# 2D Key Frame Animation Software

**By Sam Knight – P16184152**

**Computer Games Programming**

**De Montfort University**

# 1 CONTENTS

# 2 INTRODUCTION

Animation software provides various features that artists and animators use today. This report will look at key features used by modern animation tools and document the steps taken to develop these features.

# 3 BACKGROUND

## 3.1 MOTIVATION

The motivation behind making a 2D animation piece of software stemmed from an interest in drawing and animation, and a challenge to myself to implement features such as various interpolation techniques for smooth blending of key frames and Inverse Kinematics for moving and controlling connected shape skeletons. Completing this project should provide a better understanding of the structure of software that will be applicable to future projects.

## 3.2 KEYFRAME INTERPOLATION

### 3.2.1 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 animator's needs. These needs might include having an animated car move with 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 object's keyframe positions and controlling the speed of the object along that path whilst being tied to a specific frame rate.

### 3.2.2 Moving between two points: Linear Interpolation

The simplest form of interpolation is linear interpolation that, given two points, 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 time, *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.

$$X(t) = x_1 + t(x_2 - x_1)$$

$$Y(t) = y_1 + t(y_2 - y_1)$$

$$0 \leq t \leq 1$$

Below, Figure 1 , is a visual representation of X(t) and Figure 2 shows a line made from both equations with regular intervals of *t*.

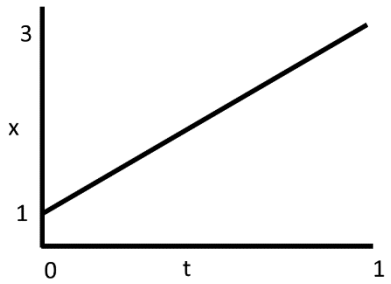


*Figure 1 Blending function for x (Unless otherwise referenced, all imagery displayed was created by me)*

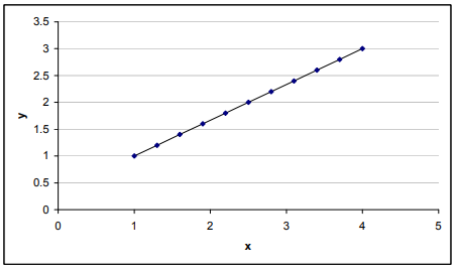


*Figure 2 (Nait-Charif, 2011/12)*

### 3.2.3 Non-linear Interpolation

Non-linear interpolation is different from linear interpolation in that the ratio of spacing in time, *t,* won't necessarily be the same as the output. This can create smooth motion like how a physical object accelerates and

decelerates 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.

### 3.2.4 Trigonometric interpolation

$$X(t) = x_1 cos^2(t) + x_2 sin^2(t)$$

$$Y(t) = y_1 cos^2(t) + y_2 sin^2(t)$$

$$0 \leq t \leq \frac{\pi}{2}$$

Below, Figure 3, is an example of X(t) and how $x_1 cos^2(t)$ and $x_2 sin^2(t)$ add together to give interpolated values between 1 and 3 (The blue curve). Figure 4 is the resulting interpolated values between the two end points.

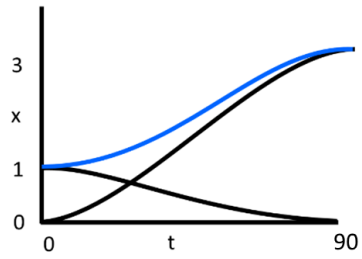


*Figure 3 Blending function for x. The 2 black curves are $x_1 cos^2(t)$ and $x_2 sin^2(t)$.*

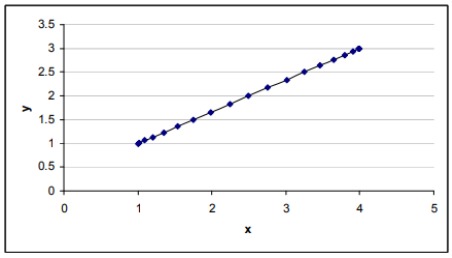


*Figure 4 (Nait-Charif, 2011/12)*

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

### 3.2.5 Cubic interpolation

$$V_1 = 2t^3 - 3t^2 + 1 \qquad V_2 = -2t^3 + 3t^2$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} V_1 + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} V_2$$

$$0 \leq t \leq 1$$

One main difference is that the parameter *t* is between 0 and 1 rather than 0 and $\frac{\pi}{2}$ which is a lot more useable compared to radians.

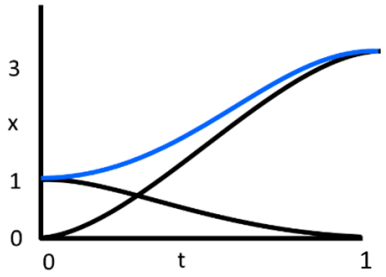


*Figure 5 Blending function for x. The 2 black curves are $2t^3 - 3t^2 + 1$ and $-2t^3 + 3t^2$*

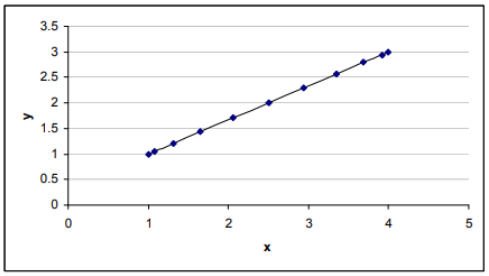


*Figure 6 (Nait-Charif, 2011/12)*

These methods of interpolation are useful for certain circumstances and give us 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 endpoints. What if we wanted to generate points that result in a curved path rather than a straight path?

### 3.2.6 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 endpoints, a quadratic Bezier curve has three control points and a cubic Bezier curve has four control points. Bezier curves can keep increasing to the $n^{th}$ 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 endpoints – this is called DeCasteljau's algorithm. One set of endpoints is directly taken from the in-between points of the other two sets of endpoints and the resulting curve is made from the linear interpolated points between the resulting endpoints.

$$P_0{}^1 = (1-t)P_0 + tP_1, \qquad P_1{}^1 = (1-t)P_1 + tP_2$$

$$P(t) = (1-t)P_0{}^1 + tP_1{}^1$$

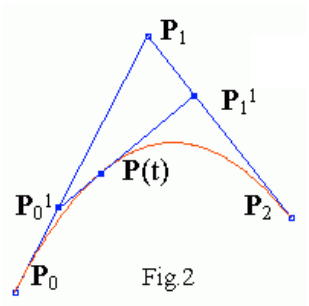


*Figure 7 Example of quadratic Bezier curve with interpolation. (Demidov, 2001)*

The above calculation is enough to code this curve, however, the whole thing can be put into one equation. Sub in $P_0{}^1$ and $P_1{}^1$:

$$P(t) = (1-t)[(1-t)P_0 + tP_1] + t[(1-t)P_1 + tP_2]$$

$$P(t) = (1-t)^2P_0 + 2(1-t)tP_1 + t^2P_2$$

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

$$P(t) = \sum_{i=0,n} B_i{}^n(t)P_i$$

$$B_i{}^n(t) = C_n{}^i t^i (1-t)^{n-i}$$

$$C_n{}^i = \frac{n!}{i!\,(n-i)!}$$

$$N = n + 1$$

$$N = number\ of\ control\ points$$

If we make $P_1$ 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, we 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 diagrams below.

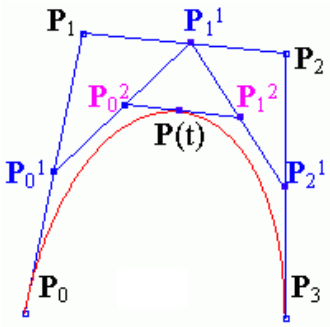


*Figure 8 Example of cubic Bezier curve from: (Demidov, 2001)*

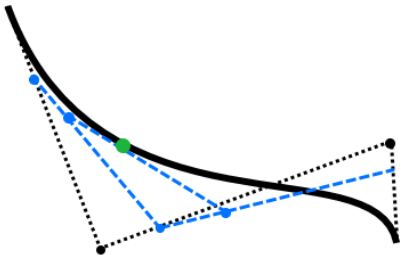

*Figure 9 Example of cubic Bezier curve*

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 start to become more computationally expensive since we're exponentially increasing the number of iterations where we linearly interpolate. However, there is a workaround, Bezier and other curves methods can be pieced together to create longer curves without exponentially increasing the cost to compute.

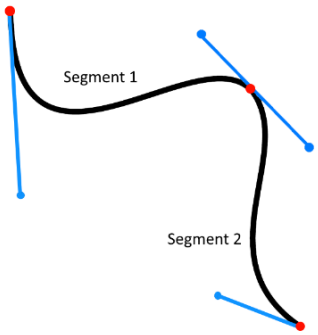


*Figure 10 Piecewise Bezier curve*

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 of control as they can be easily manipulated using control points to push and pull the curve in certain ways. However, what if we wanted a curve that goes through all the points we give it?

## 3.3 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.

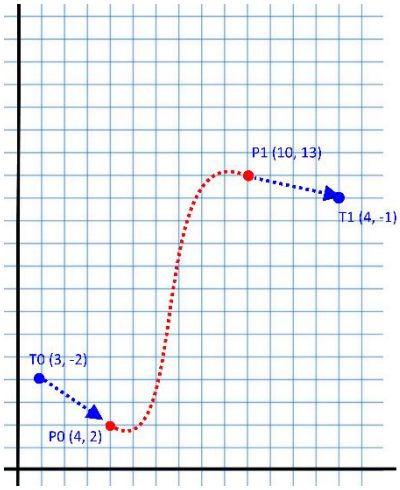


*Figure 11 Hermite Curve*

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 $P_1$ would be $(P_2 - P_0)$. We can also control the tension of the curve by adding a constraint to the tangent, α, which controls the magnitude.

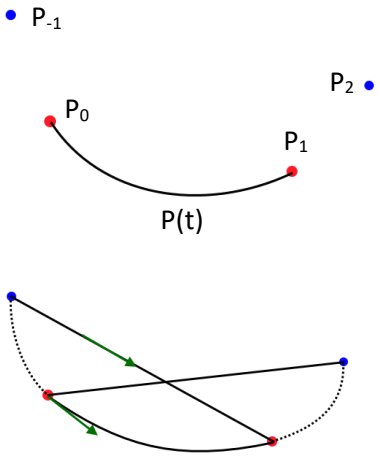


*Figure 12 Catmull-Rom spline*

$$[P(t)] = [t^3 \quad t^2 \quad t \quad 1] * \begin{bmatrix} -\alpha & 2-\alpha & \alpha-2 & \alpha \\ 2\alpha & \alpha-3 & 3-2\alpha & -\alpha \\ -\alpha & 0 & \alpha & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} P_{-1} \\ P_0 \\ P_1 \\ P_2 \end{bmatrix}$$

*Figure 13 Interpolation equation for Catmull-Rom spline at time t.*

### 3.3.1 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 can't 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 keyframe's time or position. Since we don't want the path to physically change we are left with giving up control of the keyframe times.

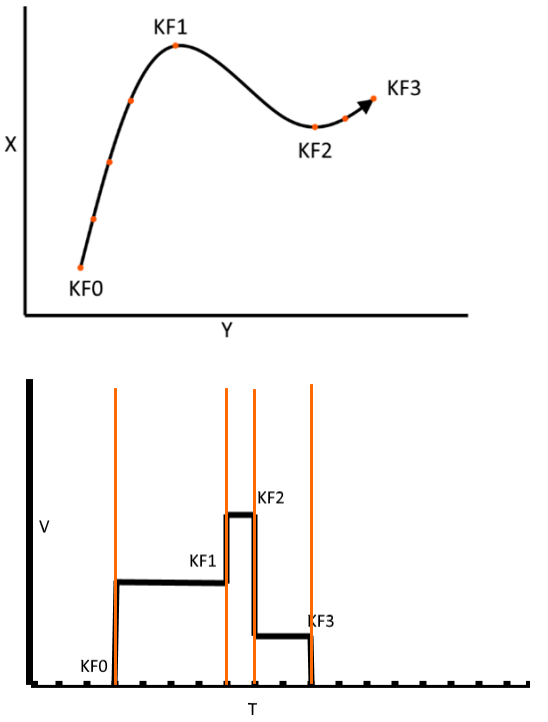


*Figure 14 First graph shows position, 2nd graphs shows velocity over time of the first graph*

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

We now have two ways of creating curves with reasonable amounts of control, however, there are still issues that need solving. We have a way to interpolate between two points linearly and along multiple types of curve. Let's 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. Figure 15 shows how you could plot speeds on a graph and get the resulting motion over time on the right.

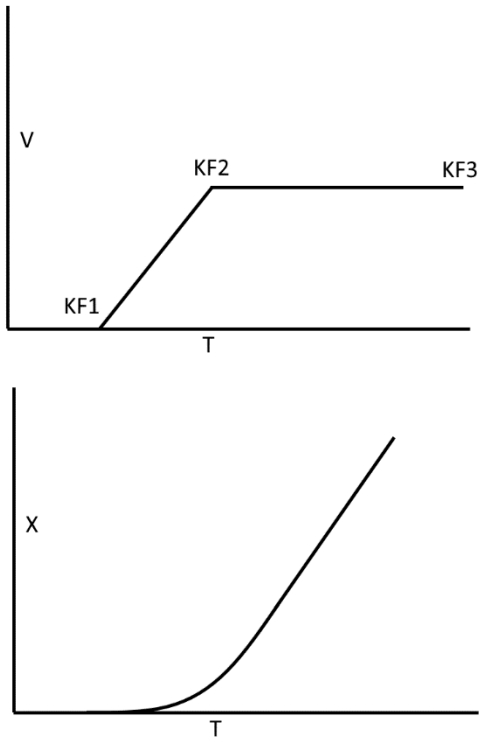


*Figure 15 Example of using linear interpolation for controlling speed and the resulting curve in motion*

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

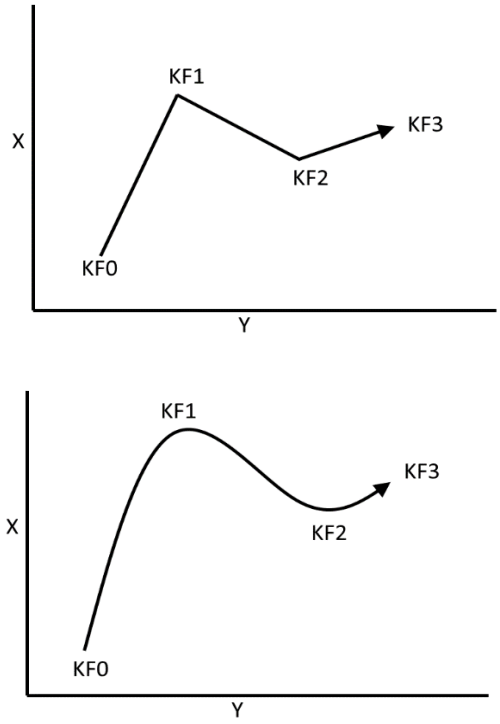


*Figure 16 Example of straight and curved paths that an animated object might follow*

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

## 3.4 INTERPOLATION

### 3.4.1 Context

My animation project will have 2D shapes linked together by joints to form an articulated chain or "arm" that can be moved for a desired key frame. These individual links can be rotated around their joints, however, manipulating the whole "arm" whilst maintaining a realistic form would take tedious careful movements. This is where inverse kinematics comes in handy. The goal with inverse kinematics is to move the "hand" or end-effector and have the "arm" follow smoothly without breaking.

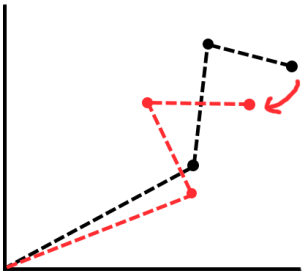

*Figure 17 example of IK*

There is also forward kinematics which achieves the opposite, given the joints of the chain you can calculate where the end effector is. However, in computer software a lot of positions of elements are already known e.g. given a square, you as the programmer define its position telling it where to be rendered – so this won't help.

### 3.4.2 Articulated Body

An articulated body can be thought of like a hierarchical tree structure made of links that are connected by joints. Links are simply connections between joints, however, there are multiple types of joints such as revolute and prismatic. A revolute joint is a joint that rotates its link and a prismatic joint extends and contracts its link. The initial joint is called the root or base, there can be multiple links from a root joint, for example, a torso of a humanoid body could be considered a root joint with arms and legs connected to it – but the root is still considered the first in the chain. The end of a chain is called the end-effector and there can be multiple end-effectors. Furthermore, the end-effector is used to control the joints via inverse kinematics. Articulated bodies can be expressed in terms of Degrees of Freedom (DOF), in 3D an articulated body could have a high overall DOF since each joint can have a maximum of three axes of rotation. However, in 2D only one axis of rotation exists so the overall DOF will be low in comparison.

### 3.4.3 Solving methods for Inverse Kinematics

There are multiple ways to solve inverse kinematics including: algebraic methods and iterative methods such as the Jacobian inversion method and FABRIK method. The algebraic solution involves calculating the end-effector position using trigonometry, however, if the DOF value increases, so do the steps required to solve the problem. Iterative methods work by solving the problem multiple times, each time getting a little bit closer to the intended goal. The Jacobian method is one of the earlier iterative methods that translates each joint by calculating the change in rotation of each joint using a Jacobian matrix. Firstly, calculate the difference in rotation for each joint: to do this we need the difference between the target (T) and the end-effector (E) and the inverse Jacobian.

$$dO = VJ^{-1}$$

However, the inverse of a matrix can only be calculated if the matrix has the same number of rows as columns and the determinant is not zero otherwise it cannot exist. There is an alternative that we can use for an approximation of $J^{-1}$ which is $J^T$ the Jacobian transpose.

$$dO = VJ^T \qquad V = T - E$$

The Jacobian matrix can be calculated from the cross products of the axis of rotation of a

joint ($R_i$) and the difference in positions of the joint ($P_i$) and the end effector (E). Each term is a vector.

$$J = [R_1 \times (E - P_1), \dots, R_i \times (E - P_i)]$$

Now we know the change angles for each joint we can update each joint's position (O). To do this we translate them by the difference in rotation multiplied by a time step (h).

$$O \mathrel{+}= dO * h$$

These steps are then repeated until the end effector is as close to the target as desired. The Jacobian method, however, requires a lot of computational power due to its use of matrices and cross product.

Another iterative method is the FABRIK method (Forward and Backward Reaching Inverse Kinematics). This method focuses on solving the inverse kinematic problem using only positions of joints and how to move them toward a subsequent target. There are two main steps to the FABRIK method called Forward Reaching and Backward Reaching. Forward reaching starts by making the end-effector equal to the target followed by finding where the previous joint lies on a line between the end-effector and said previous joint. This is repeated for each joint down the chain and results in the how chain being disconnected from the original root position.

This is fixed by Backward Reaching which repeats the Forward Reaching step but in reverse, starting by moving the root joint back. If the target is within the articulated body's full length, the body will smoothly reach toward its goal after several iterations. The FABRIK method has a much lower computational cost when compared to methods like Jacobian since it doesn't handle any rotation, requires a much smaller number of calculations to be made per iteration and produces much more natural and stable movements.

Figure 2: (a) to (d) demonstrates how each link is repositioned along a line between its previous joint and the goal/subsequent joint during Forward Reaching. (e) and (f) show Backward Reaching.

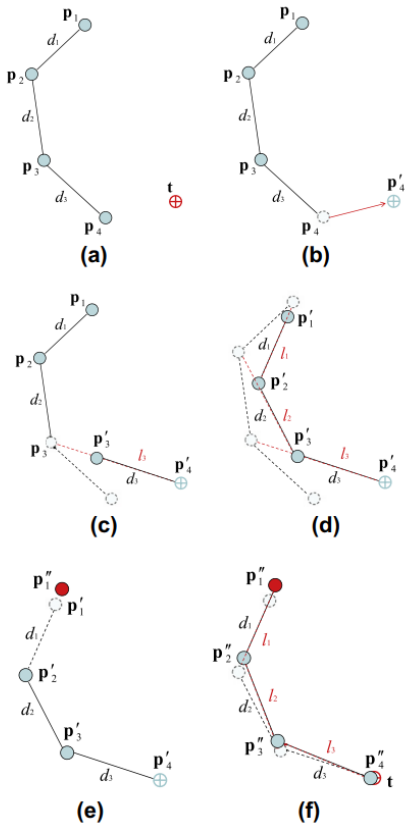


*Figure 18 Example of IK using forward and backward reaching (Andreas Aristidou, 2011)*

# 4 CRITICAL ANALYSIS

## 4.1 SPECIFICATION

The project started with planning the necessary steps and goals to create a piece of functioning animation software. The backlog in **Error! Reference source not found.** shows the list of key features to implement into the animation software. The project will be realised using the agile project management and work will be carried out in sprints. The backlog contains key tasks to be completed with various properties: story (Required to progress), priority (What should be done first) and points (value that relates to how difficult the task will be). During the sprints tasks will be completed and the points will be totalled to see how much progress is being made, if

the point total stays low the project isn't making good progress, if the point totals are consistently high then the project is progressing smoothly. These points are estimates and can be subject to change depending on whether or not tasks remain uncompleted resulting in progress being halted even if the sprints' point total is high. The log of sprints will be appended in Appendix 2.

## 4.2 Development

The first sprint started with creating a means to draw shapes. This requires having something to draw to such as a canvas and a way to render shapes to it. The canvas is simply a SFML render texture that provides an area and the shapes are visible only rendered inside the canvas rectangle. The canvas doesn't own the shapes and are therefore stored in their own separate STL vector that allows for them to be easily passed around. To create shapes we need a way to handle the mouse events. The shape tool class essentially creates a new shape when the left mouse button is clicked, the shape can then be dragged to a desired size, then act of drawing ends when the left mouse button is released. To make drawing shape easier, a camera class was implemented that builds on an SFML view to provide a way to move around and zoom in and out of the canvas.

Moving onto the 2nd sprint we need to add more tools, such as a transform tool, which requires a way to switch between tools. Thus, a Tool Box class is added that has a pool of all the available tools and upon calling the appropriate method the corresponding tool becomes active. With this we can begin on the transform tool: the transform tool is a relatively large and complex tool compared to the drawing tool since it is being designed to handle all 3 transformations: translation, scaling and rotation. The first transformation to be added was translation, which simply involved moving the shape relative to the mouse. The next transformation was scaling which brought about some problems with using SFMLs default shapes. To achieve a desirable scaling from each corner and each side required setting the origin of that shape to the opposite side/corner which also repositions the shape. This results in a lot of complexity and just seemed a bit pointless. It looked like making a custom shape that could handle vertices and position separately would work much better. During this sprint a colour tool was implemented. To achieve a simple implementation, I would need some way of picking values for red, green and blue, for the shape's RGB values. This could be done in various ways but having 3 separate sliders seemed like a good start.

The 3rd sprint continues to add another tool, finishes off the transform tool and refactoring of shapes. Since the transform tool required the basic shape to be remade that's where I started. A custom quadrilateral shape that separates its position from its vertices and implements custom transformation methods. Now that we can easily transform a shape the next feature to add is joint functionality and a joint tool. When initially planning how to implement these features I wanted them to provide the base functionality for implementing Inverse Kinematics in the future which requires a chain like tree structure of shapes so that I can tell what shapes are connected to. The first idea was to have a class keep track of these connections; however, it quickly became obvious that this wouldn't work well. Therefore, the next option would be to let the shapes keep track of these connections themselves: each shape can have a parent shape and multiple child shapes that can be used to iterate along a chain of joint shapes. The joint tool would, therefore, need to handle the linking of these shapes by firstly selecting a shape, that will be the parent of a 2nd shape, and then selecting a 2nd shape that will be the child of the first shape. This addition to the Quad Part class also requires a slight refactor to so that when its transformation methods are called, and it

applies them to the child shapes that are joint to the current shape. The only thing this custom shape cannot do is be any other shape than a quadrilateral, so as an extra level we need to render an SFML shape on top of the Quad Part every frame matching its position and size. Therefore, I created an object class that inherits from Quad Part and contains a polymorphic SFML shape that can currently be a SFML rectangle or a custom ellipse shape. So far, switching between tools has been done using a rather simple button class that isn't very generic and customisable. The next task then was to refactor the buttons so that they can display a custom icon. I also wanted buttons to be easy to link up to functionality so I decided to implement call-back functionality that allows you to pass a function pointer to the button; this function will then be called when the button is clicked.

The 4th sprint involved a lot of refactoring to use new features that I initially thought would provide some very useful functionality to the project – easy to use event handling. Essentially, I wanted to provide a way for classes to be able to listen for events and do something when that event happens and hopefully prevent messy code structure as a result due to being able to keep certain classes separate. The initial functionality was event listening, after doing some research I found the Signal/Slot pattern that provides a nice implementation of the observer pattern. Essentially, a class can have a "signal" (an event that could happen) whilst another class has a call-back function that can be considered a "slot". The function pointer of the call-back can be connected to the signal and, therefore, when the signal (event) is triggered, all connected call-back functions are called. This now allows classes to interact with each other whilst remaining separate since neither class **needs** to know about the other. As an extra step I wanted this system to work with input and remove the need for passing input directly to UI elements such as buttons, since most button examples I've seen don't seem to require mouse input every frame by the programmer. This makes a lot of sense since from the programmer's point of view all they want is a functioning button and not how to get it to work i.e. by passing its required input. To fully hide this from view it requires a singleton class that can pass input from an input handler to the button element behind the scenes. The result is an event handler class that acts as a map for signals to register to, call-back methods can be connected by specifying a name string that is associated with the desired signal. Since the event handler and signals handle call-back functions that can have varying parameters (also, all call-backs must return void, since if a slot were to return: any other slots would end up not being called) resulting in both classes being very reliant on templating to be generic. We also need to refactor the way we handle input since it has become quite complex and these signals should now reduce this complexity. Beforehand the input was all handled in the main loop and all classes that take input would end up making the input handling loop bigger. To reduce the amount of code in the main loop we move the input handling to a separate class that also implements signals and the event handler instance. This results in a much smaller input polling loop since all we do is notify the corresponding signals that their input event has happened (rather than having to deal with passing input to *n* classes – we just notify 1 signal). The event handler and signals now handle passing the data to all the relevant call-back functions in various classes without anyone having to see it. However, using singletons always seems to have shortcomings, in this case it is apparent when two classes have events that the other listens to. If both classes register their signals to the event handler, and then both classes try to connect call-back functions to the other class's signal on construction, depending on which class is declared first, one of those signals will not exist yet resulting in an error. Therefore, the event handler isn't exactly

programmer safe when being used for purposes other than the simple button elements, as a result the event handler shouldn't be used by default and a class's signals should have call-back methods connect outside of the class/below the class declaration (and below both declarations if trying to connect two classes).

Still on the 4th sprint the next task is to try and implement a way to animate the objects. Which sounds simple, however, it proved to be very complicated. There aren't really any guides on how to set up this kind of structure so I would have to implement this from scratch. Starting with the easy part, we need a way to record and load data into/from an object using their positions, vertex positions and rotation.

Moving onto the 5th sprint, the next step is how to store this data, which in turn brought another problem to light, that is that if we store the object's data how do we give said data back to the object. Pointers looked like the simple solution, however, since we currently used a vector to store an object, if we delete an object, then the pointer could become invalid and we can no longer give data back to a shape. This meant that either I would have to do some memory management, which I thought would complex and code would rely on it working properly, or I add a system to associate ID numbers with each object so that they can be retrieved. I opted for what I thought would be simpler and a potentially more reliable ID system. Since each shape required an ID the "Shapes" container class needs to handle the creation, deletion and retrieval of objects to generate valid ID numbers.

Now that we have a way of retrieving and loading object data, we can move back onto how to store this data. Having a "Key Frame Manager" class that has a Set that contains shape key frame data (a struct that contains an ID and a set of key frame structs that each contain a frame number and the object data) store this data so that it can perform operations on it seemed like a viable option. Then the basic operation methods needed to be implemented i.e. recording a key frame; deleting a key frame; and setting the current frame.

I refactored this code several times since it was difficult deciding on a STL container to use to store the object key frame data since I needed to perform most of the STL operations such as adding an element, deleting a specific element, iterating over the elements and searching for a specific element. Since an animation could contain many objects and key frames the container needs to provide these operations as fast as possible – especially when playing the animation. I initially tried a vector since it provides very fast iteration, however, it doesn't provide any look-up methods that are required for error handling and deletion. I also tried a map and set, however, these don't provide nearly the iteration speed that a vector would. After looking for some kind of container that could provide the best of both containers, I came across Boost's containers and the flat map/set stood out since they both have an underlying vector – resulting in fast look-up and iteration speed at the cost of insertion and deletion speed. Since the insert and delete methods are called not nearly as frequently as the look-up/iteration, this seemed like a valid option.

Once the basic operations for the key frame manager had been implemented it would be relatively simple to add functionality for the animation to be recorded and run. However, just adding the functionality means we can immediately use it. Interacting with and displaying the amount of data an animation can produce requires a complex piece of UI. Looking at most modern software that implements key frame animation; they have a form of timeline that key frames are displayed to. Since our data structure consists of shapes and their corresponding key frame data,

creating a UI to mimic this would make sense.

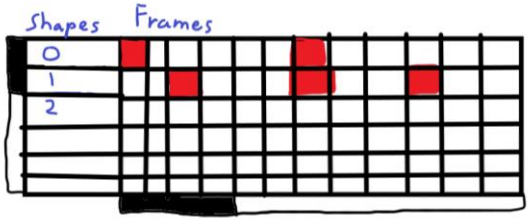


*Figure 19 Diagram of the timeline user interface*

In Figure 19 you can see that on the left the shapes are represented by their ID numbers and on the right the key frames are represented by the red squares. To keep it manageable the timeline and key frame manager work with frame numbers rather that frame times. Although, frame times can still be calculated and end up being displayed at the bottom of the time line. This wasn't my first idea for the time line but was the first concrete and planned out attempt at implementing it. Initially I wanted to have some sort of classic scrollable UI but that relies on rendering anything inside it to a texture and without adding some complex code to add and remove key frame information either side of the currently visible part of the texture, this isn't very feasible to do in the short term – let alone make it work smoothly. On the other hand, the UI in Figure 19 only displays a set number of frames at one time; the current frames being shown can be changed by moving the slider and would shift all those red rectangles accordingly. Currently the code is inefficient: when sliding the frame slider there is noticeable frame rate drops due to the way the frame slots currently update. Basically, all the slots are reupdated every update frame that the slider bar moves. In hindsight a better option would be to add code to detect when the slider bar moves enough to have moved the frames along by 1 and then update the frame slots from an iterator that walks along the data with the slider bar – thus not iterating over all the data. Then I moved onto linking the key frame manager and the time line; this essentially involved sending the shape-key-frame data to the time line so that it can deal with it. Finally, certain events were added to function in conjunction with the key frame manager such as recording or overwriting a keyframe whenever a shape is drawn or transformed. Also, to show that objects can be animated a current key frame slider can be moved along the timeline that results in the key frame manager updating and interpolating shapes positions between key frames.

## 4.3 End notes

This project had a lot of planned features that perhaps were too much to achieve in the given time, however, the base goal of being able to create, connect and animate objects was reasonably achieved – albeit with occasional bugs. In hindsight the amount of time needed to complete the basic implementation of key frame animation was severely underestimated and could be improved. Currently, there are play, stop and record buttons that provide no functionality. Also, a form of reverse signal is needed for all UI elements; clicking anywhere with the draw tool will create a new shape, this is undesirable and fills up the time line with pointless objects. Essentially, anything that needs this kind of prevention needs to be notified when the mouse is over any UI element. Finally, Inverse Kinematics and most forms of interpolation did not get implemented, however, the ground work for them has been set up and is ready to be used. IK can easily make use of the object tree structure created by joints and the key frame manager just needs to implement the different forms of interpolation calculations.

# 5 SOFTWARE STORYBOARD

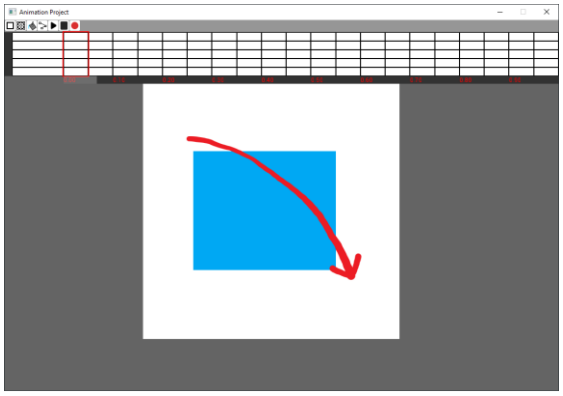

*Figure 20 How to draw a shape*

Initially the draw shape tool is already selected, so we don't need to push any buttons yet. To draw a shape: click-and-drag with the left mouse button to draw a rectangle and release to finish. Note: there was a bug that causes an error if the draw tool is selected first, however, following this guide should avoid any errors.

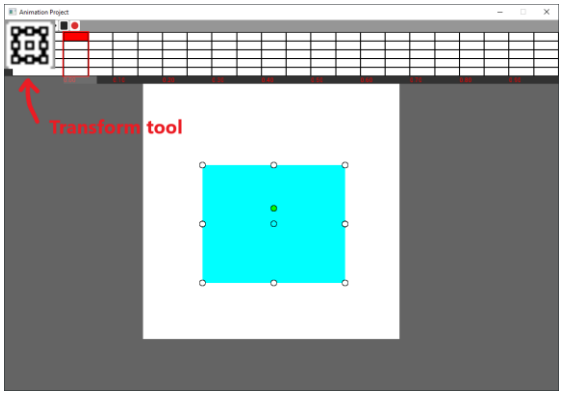


*Figure 21 transforming a shape*

Firstly, select the transform tool in the top left, then click on the rectangle to select it and the transform handles should appear. To resize the rectangle: click-and-drag on any of the 8 handles located on the edge of the rectangle to resize it. To rotate, click-and-drag the green handle in a circle motion. To reposition the origin of rotation, click-and-drag the handle located next-to the green handle/centre of the rectangle.

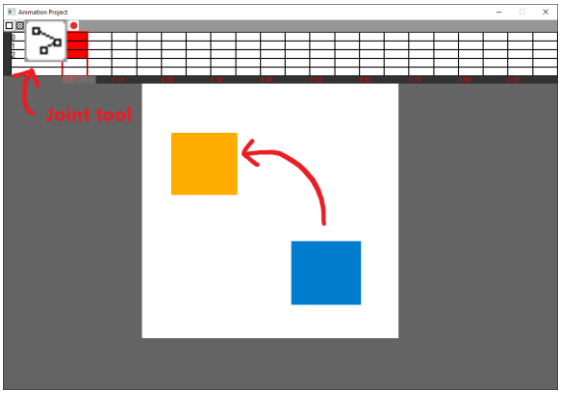


*Figure 22 Creating a joint*

To a create a joint, first select the joint tool in the top left. To connect two shapes, first click-and-release on the shape that you want the joint to anchor to, then click-and-release on the other shape to connect it to the first.

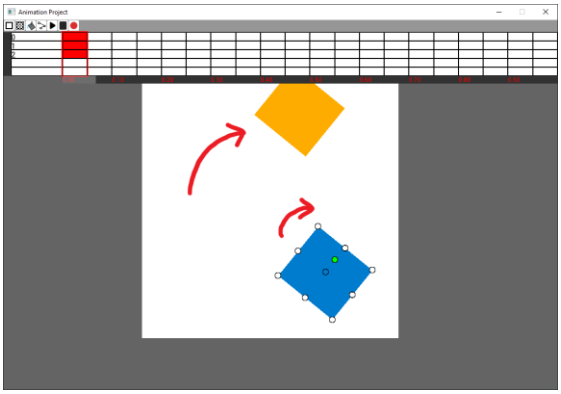

*Figure 23 Rotating a connected shape*

Now that we have connected two shapes, we can use the parent shape (blue) to apply transformations to the child shape (yellow) around the parent shape's (blue) origin point. Above we use the transformation tool to rotate the blue shape and the yellow shape rotates accordingly.

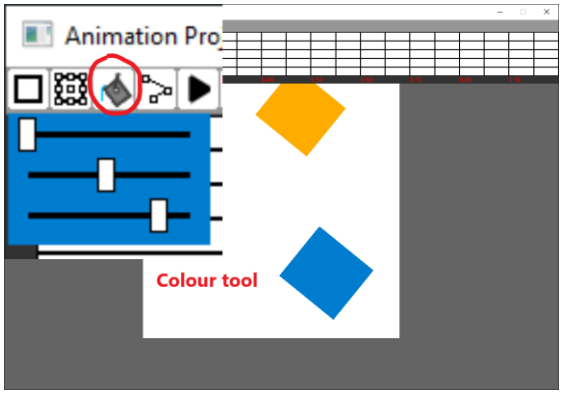


*Figure 24 Applying a colour*

To apply a colour, select the paint bucket button, then click-and-drag the 3 slider handles to select the desired amount of red, green and blue values; then click on a shape to apply that colour.

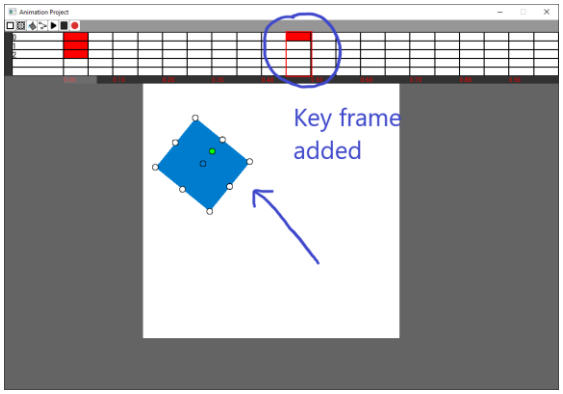


*Figure 25 Recording a key frame*

Key frames are recorded whenever a shape is created or transformed (currently only translations). To record a key frame at a specific time, click-and-drag the red current frame slider rectangle to the desired time and then move or create a shape in that frame – a red key frame should appear.

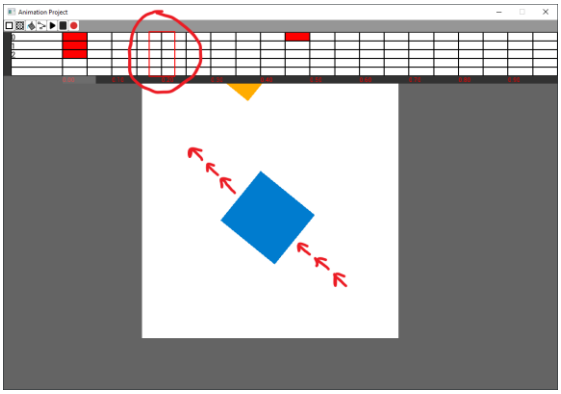

*Figure 26 Interpolating between key frames*

To see the shape move between key frames, simply click-and-drag the red current-key-frame slider between the recorded key frames.

# 6 Conclusion

Overall, I think this project achieved what I wanted it to do, however, it is missing most of the key features I initially wanted to add such as IK and more complex forms of interpolation. In my Motivation statement I mentioned my interest in learning and implementing interpolation, IK techniques and software features/techniques.

Overall, I think I at least managed to learn about all the features that I wanted to implement and managed to add some software patterns like signal-slots/observer, factory and singleton. I didn't manage to properly add interpolation and IK but I have now acquired the understanding needed to add them to the software. What I learnt doing this project is that having a good understanding of the features helps when structuring the software e.g. adding joints in preparation for IK. Also, I learnt that certain features need careful planning and can take much longer to implement than anticipated. That experience has been really valuable and will help me in my future work.

Also, due to underestimating the learning curve and the resulting amount of time needed to get the software to an acceptable

point for submission, extensive end of project testing wasn't possible whilst also keeping to the deadline. Again, this has been a valuable lesson for me, it's has taught me that when estimating how long a project will take, it's likely to take longer and that must factored into the planning.

That said, it does add several features that I didn't initially plan that turned out to be very useful for software in a number of ways. For instance: Signals-Slot pattern and call-back implementation and event handling help with sending messages via an easy-to-use implementation; transform tool handles and the key frame time line were features I had to add as I found they were necessary to enable interaction with another feature – that wasn't in my initial plan. As a result, this took up an unforeseen amount of time, having to learn and experiment with features I hadn't anticipated – but this meant I did learn more than I initially set out to.

# 7 APPENDICES

*Appendix 1 Initial backlog*

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | | Project Backlog | | | | | |
| 2 | | Tasks | Story | Priority | Story Points | Status | Date Completed |
| 3 | 1 | Create shapes | yes | high | 20 | Not Started | |
| 4 | 2 | Delete shapes | yes | medium | 20 | Not Started | |
| 5 | 3 | Move tool | yes | high | 40 | Not Started | |
| 6 | 4 | resize tool | yes | high | 40 | Not Started | |
| 7 | 5 | Rotate tool | yes | high | 40 | Not Started | |
| 8 | | | | | | | |
| 9 | 6 | Undo | no | low | 100 | Not Started | |
| 10 | 7 | Redo | no | low | 100 | Not Started | |
| 11 | | | | | | | |
| 12 | 8 | Joint tool | yes | high | 60 | Not Started | |
| 13 | 9 | Rotate joint tool | yes | high | 60 | Not Started | |
| 14 | 10 | Move 'limb' tool (using Inverse Kinematics) | Yes | High | 100 | Not Started | |
| 15 | 11 | Selection tool | no | low | 100 | Not Started | |
| 16 | | | | | | | |
| 17 | 12 | Timeline viewer | yes | medium | 100 | Not Started | |
| 18 | 13 | Record keyframe | yes | high | 30 | Not Started | |
| 19 | 14 | Delete keyframe | yes | medium | 50 | Not Started | |
| 20 | 15 | Current time slider | no | low | 50 | Not Started | |
| 21 | | | | | | Not Started | |
| 22 | 16 | Save animation data | yes | high | 100 | Not Started | |
| 23 | 17 | load animation data | yes | high | 80 | Not Started | |
| 24 | | | | | | Not Started | |
| 25 | 18 | apply textures to shapes | yes | low | 30 | Not Started | |
| 26 | 19 | Remove texture from shape | yes | low | 15 | Not Started | |
| 27 | | | | | | | |
| 28 | 20 | Add Easing selection for keyframes with presets (Linear being default) | yes | low | 100 | Not Started | |
| 29 | 21 | Add basic acceleration interpolation curves | no | low | 30 | Not Started | |
| 30 | 22 | Add elastic interpolation curves to presets | no | low | 40 | Not Started | |
| 31 | 23 | Add bounce interpolation curves to presets | no | low | 40 | Not Started | |
| 32 | | | | | | | |
| 33 | 24 | Export as gif | no | low | 100 | Not Started | |

*Appendix 2 Sprint log*

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | Sprint | Task | Points | Date | Total |
| 2 | 1 | Canvas | 10 | 22/11/2019 | |
| 3 | 1 | Camera | 15 | 22/11/2019 | |
| 4 | 1 | Itool | 15 | 22/11/2019 | |
| 5 | 1 | Shape tool | 20 | 22/11/2019 | 60 |
| 6 | | | | | |
| 7 | 2 | ToolBox | 30 | 13/12/2018 | |
| 8 | 2 | Transform tool | 30 | 13/12/2018 | |
| 9 | 2 | Colour tool | 20 | 13/12/2018 | 80 |
| 10 | | | | | |
| 11 | 3 | Joint tool | 50 | 14/02/2019 | |
| 12 | 3 | Transform tool | 30 | 14/02/2019 | |
| 13 | 3 | QuadPart | 50 | 14/02/2019 | |
| 14 | 3 | Object | 5 | 14/02/2019 | |
| 15 | 3 | Buttons | 10 | 14/02/2019 | 135 |
| 16 | | | | | |
| 17 | 4 | Signals | 40 | 07/03/2019 | |
| 18 | 4 | Event handler | 40 | 07/03/2019 | |
| 19 | 4 | Input handler | 15 | 07/03/2019 | |
| 20 | 4 | Refactored buttons an | 40 | 07/03/2019 | |
| 21 | 4 | Keyframe mananger | 60 | 07/03/2019 | 195 |
| 22 | | | | | |
| 23 | 5 | Shapes container | 40 | 20/03/2019 | |
| 24 | 5 | Keyframe manager | 70 | 20/03/2019 | |
| 25 | 5 | Time line | 100 | 20/03/2019 | 210 |

# 8 BIBLIOGRAPHY


Andreas Aristidou, J. L., 2011. *Forward And Backward Reaching Inverse Kinematics.* [Online]
Available at: http://www.andreasaristidou.com/FABRIK.html
[Accessed 3 4 2019].

Barinka, L. & Berka, I. R., 2002. *CESCG 2002.* [Online]
Available at: http://old.cescg.org/CESCG-2002/LBarinka/paper.pdf
[Accessed 3 4 2019].

Bermudez, L., 2017. *Unity3DAnimation.* [Online]
Available at: https://medium.com/unity3danimation/overview-of-inverse-kinematics-9769a43ba956
[Accessed 3 4 2019].

Bermudez, L., 2017. *Unity3DAnimation: Overview of Jacobian IK.* [Online]
Available at: https://medium.com/unity3danimation/overview-of-jacobian-ik-a33939639ab2
[Accessed 3 4 2019].

Bernudez, L., 2017. *Unity3DAnimation.* [Online]
Available at: https://medium.com/unity3danimation/create-your-own-ik-in-unity3d-989debd86770
[Accessed 3 4 2019].

Demidov, E., 2001. *An Interactive Introduction to Splines: Bezier spline curves.* [Online]
Available at: https://www.ibiblio.org/e-notes/Splines/bezier.html
[Accessed 3 4 2019].

Lasenby, J. & Aristidou, A., 2011. FABRIK: A fast, iterative solver for the Inverse Kinematics problem.. *Graphical Models,* 73(5), pp. 243-260.

Nait-Charif, D. H., 2011/12. *Maths for Computer Graphics.* [Online]
Available at: https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf
[Accessed 3 4 2019].

Project, G. J., n.d. *University of Calgary.* [Online]
Available at: https://pages.cpsc.ucalgary.ca/~jungle/587/pdf/5-interpolation.pdf
[Accessed 3 4 2019].

Winkle, L. V., 2009. *CodePlea: Introduction to Splines.* [Online]
Available at: https://codeplea.com/introduction-to-splines
[Accessed 3 4 2019].

Winkle, L. V., 2009. *CodePlea: Simple Interpolation.* [Online]
Available at: https://codeplea.com/simple-interpolation
[Accessed 3 4 2019].