**2D Key Frame Animation Software**

**By Sam Knight – P16184152**

**Computer Games Programming**

**De Montfort University**

Contents

[2 Introduction 3](#_Toc5197824)

[3 Background 3](#_Toc5197825)

[3.1 Motivation 3](#_Toc5197826)

[3.2 Keyframe Interpolation 3](#_Toc5197827)

[3.2.1 Introduction 3](#_Toc5197828)

[3.2.2 Moving between two points: Linear Interpolation 3](#_Toc5197829)

[3.2.3 Non-linear Interpolation 3](#_Toc5197830)

[3.2.4 Trigonometric interpolation 3](#_Toc5197831)

[3.2.5 Cubic interpolation 4](#_Toc5197832)

[3.2.6 Creating a curve between points: Bezier Curve 4](#_Toc5197833)

[3.3 Cardinal Splines and Catmull-Rom Splines 6](#_Toc5197834)

[3.3.1 Consolidation 6](#_Toc5197835)

[3.4 Interpolation 8](#_Toc5197836)

[3.4.1 Context 8](#_Toc5197837)

[3.4.2 Articulated Body 8](#_Toc5197838)

[3.4.3 Solving methods for Inverse Kinematics 8](#_Toc5197839)

[3.5 State of the art 9](#_Toc5197840)

[3.5.1 Key Frame User Interface 9](#_Toc5197841)

[4 Critical analysis 10](#_Toc5197842)

[4.1 Specification 10](#_Toc5197843)

[4.2 Development 10](#_Toc5197844)

[4.3 Testing and Report 11](#_Toc5197845)

[5 Design 11](#_Toc5197846)

[5.1 Introduction 11](#_Toc5197847)

[5.2 UML 11](#_Toc5197848)

[5.3 UI Designs 11](#_Toc5197849)

[6 Testing 11](#_Toc5197850)

[6.1 Testing Strategy 11](#_Toc5197851)

[6.2 Results 11](#_Toc5197852)

[7 Conclusion 11](#_Toc5197853)

[8 Bibliography 12](#_Toc5197854)

# Introduction

# Background

## Motivation

The motivation behind making a 2D animation piece of software stemmed from an interest in drawing and animation, and a challenge to see if I could implement features of modern drawing/animation software myself. Completing this project should provide a better understanding of the structure of software that will be applicable to future projects.

## 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 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.

### 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 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.

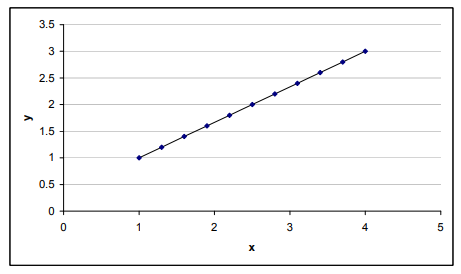
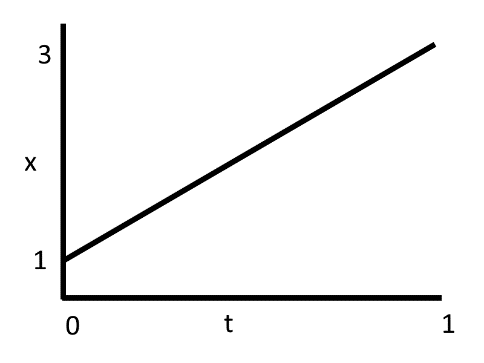
Below is a visual representation of X(t). On the right shows a line made from both equations with regular intervals of *t*. 

Figure 1 Blending function for x

Figure 2 (Nait-Charif, 2011/12)

### 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.

### Trigonometric interpolation

Below 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.

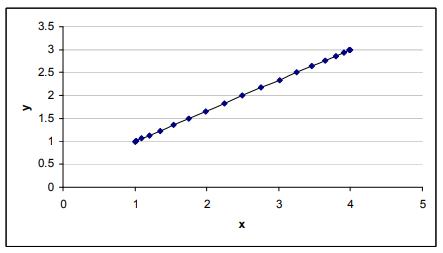
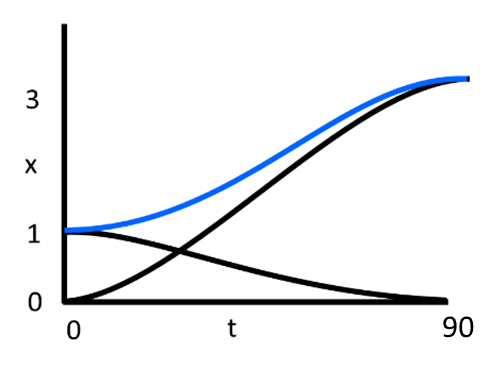


Figure 3 Blending function for x. The 2 black curves are and.

Figure 4 (Nait-Charif, 2011/12)

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

### Cubic interpolation

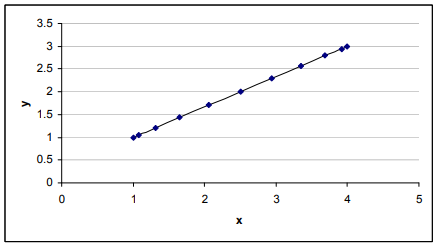
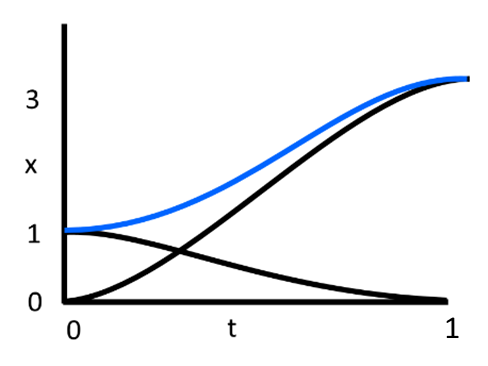
One main difference is that the parameter *t* is between 0 and 1 rather than 0 and which is a lot more useable compared to radians. 

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

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?

### 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 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 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.

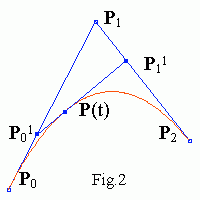
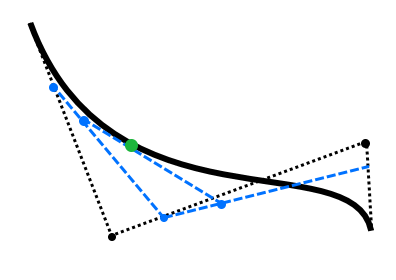
The above calculation is enough to code this curve, however, the whole thing can be put into one equation. Sub in and :

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

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.

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 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. 

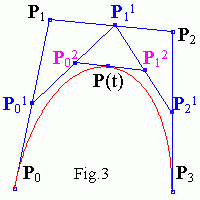


Figure 8 Example of cubic Bezier curve from: (Demidov, 2001)

Figure 9 Example of cubic Bezier curve

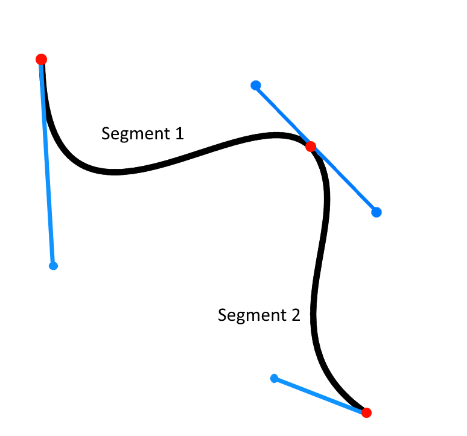
Stopping at cubic Bezier curves is a good idea, since it offers a lot of control over a curve, and any Bezier curves with an order that exceeds cubic start to become more computationally expensive since we’re exponentially increasing the number of iterations where we linearly interpolate. However, there 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?

## Cardinal Splines and Catmull-Rom Splines

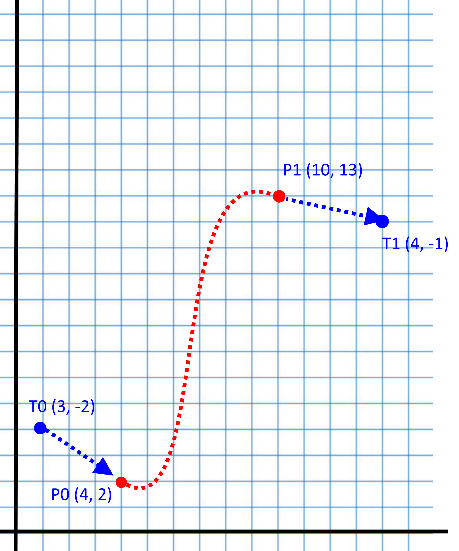
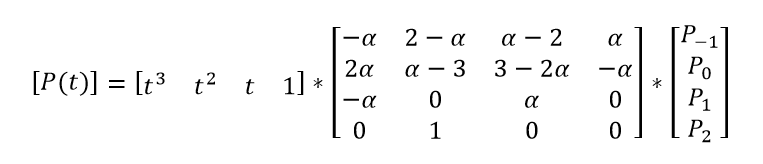
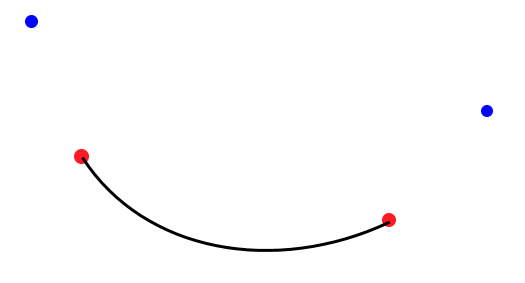
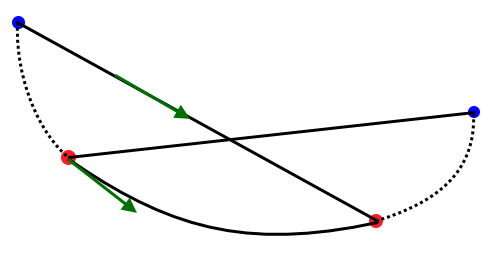
Cardinal splines are a series of connected curves, so in the piecewise Bezier example above the curve would be considered a cardinal spline. Catmull-Rom splines are a type of cardinal spline made from multiple Hermite splines. Hermite spline interpolation creates a curve using two control points and 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 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. 



P-1

P0

P1

P2

P(t)

Figure 12 Catmull-Rom spline

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

### 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 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.

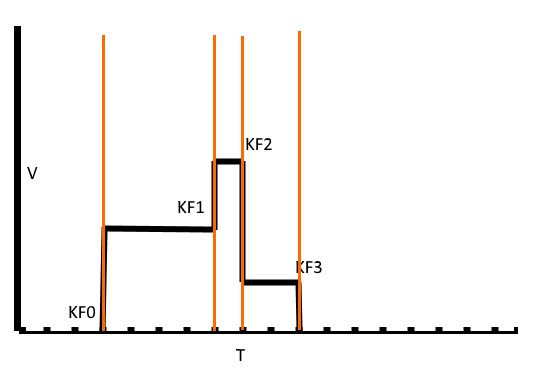
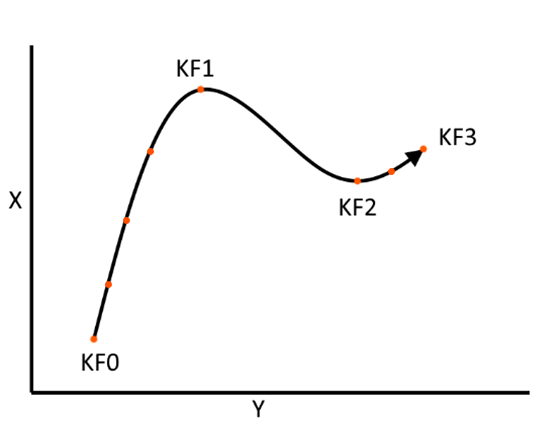


Figure 14 First graph shows position, 2nd graphs shows velocity over time of the first graph

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

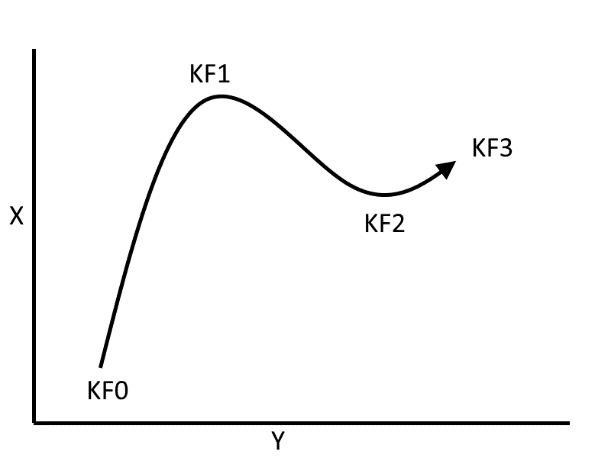
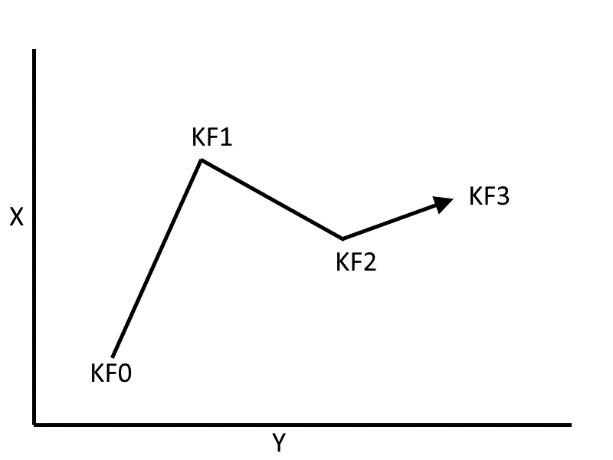


Figure 15 Example of using linear interpolation for controlling speed and the resulting curve in motion

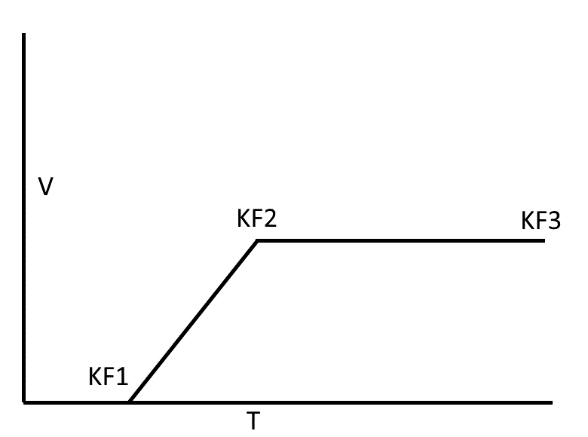
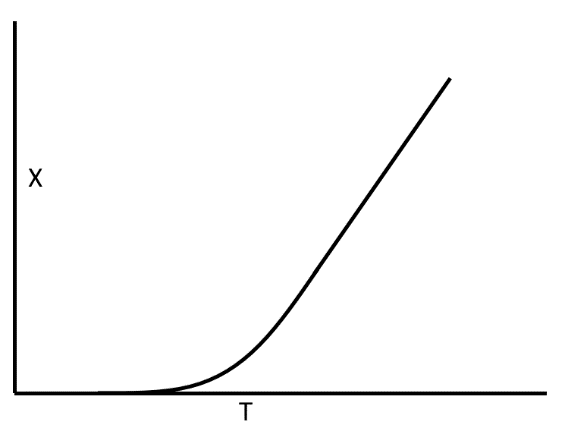


Figure 16 Example of straight and curved paths that an animated object might follow

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

## Interpolation

### Context

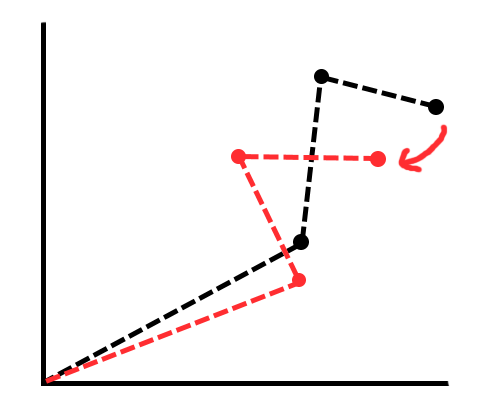
My animation project will have 2D shapes linked together by joints to form an articulated chain or “arm” that can be moved for a desired key frame. These individual links can be rotated around their joints, however, manipulating the whole “arm” whilst maintaining a realistic form would take tedious careful 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.

### 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.

### 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 degree of freedom 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.

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 JT the Jacobian transpose.

The Jacobian matrix can be calculated from the cross products of the axis of rotation of a joint (Ri) and the difference in positions of the joint (Pi) and the end effector (E). Each term is a vector.

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).

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.

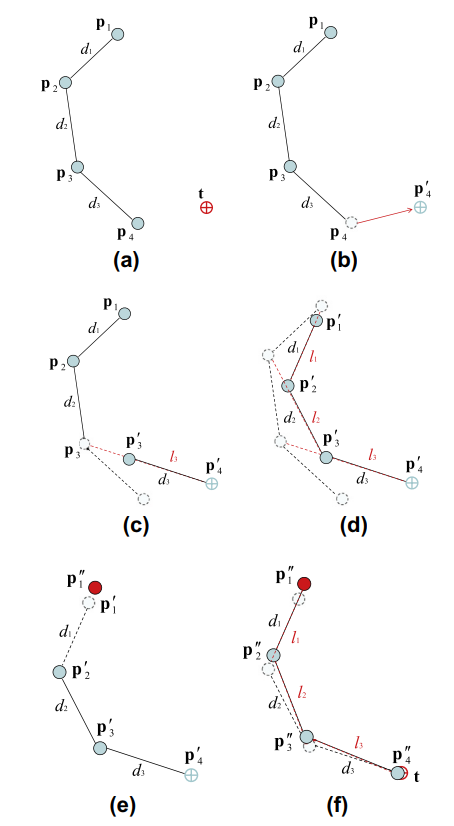
Figure 2: (a) to (d) demonstrates how each link is repositioned along a line between its previous joint and the goal/subsequent joint during Forward Reaching. (e) and (f) show Backward Reaching.

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

## State of the art

### Key Frame User Interface

* Creating a user interface that can represent the key frames on a timeline for multiple objects and handle user interactions with it isn’t a trivial task.
* Compare how modern software achieves this
* What key aspects do they have in common, are they user friendly or over complex?

# Critical analysis

## Specification

The project started with planning the necessary steps and goals to create a piece of functioning animation software. The backlog in Appendix 1 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 stay low not 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.

## 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 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 and finish drawing on release of the left mouse button. 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 shapes 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 features 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, these quickly became obvious that it wouldn’t work well. Therefore, the next option would be to let the shapes keep track of these connections themselves: each shapes 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 it applies them to the child shapes that are joint to the current shape. The only 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 makes to programs functions unclear to a user. 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 class 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 also 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 to be passed 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 arguments (all call-backs must return void since if a slot were to return, then any other slots would end up not being called) resulting in both classes being very reliant on templating for 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 classes 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 button elements (and other simple UI elements), as a result the event handler shouldn’t be used by default and signals should be connected to outside of classes (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 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 objects 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 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 with what I thought would be simpler and 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 look for some kind of container that could do 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.

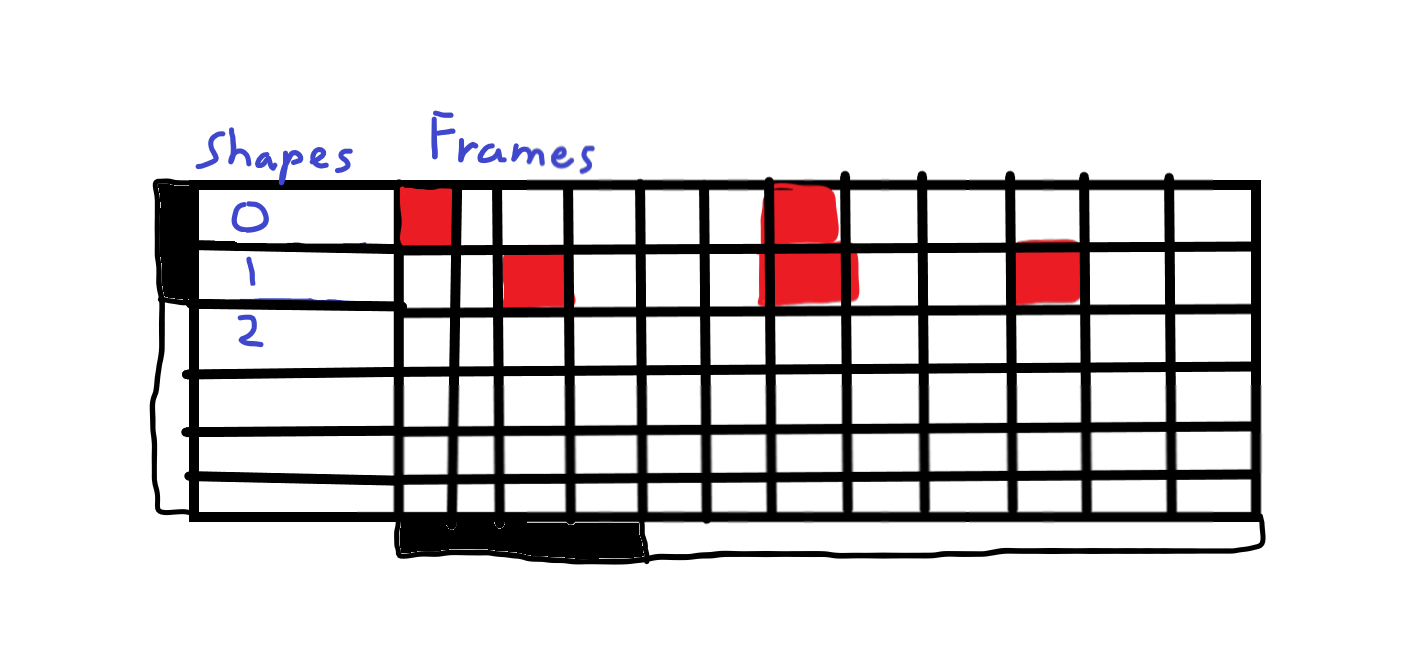
Once the basic operations for the key frame manager had been implemented it would be relatively simple to add functionality for the animation to be recorded and run. However, just adding the functionality means we can immediately use it. Interacting with and displaying the amount of data an animation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

Figure Diagram of the timeline user interface

In the 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

## Testing and Report

# Design

## Introduction

## UML

## UI Designs

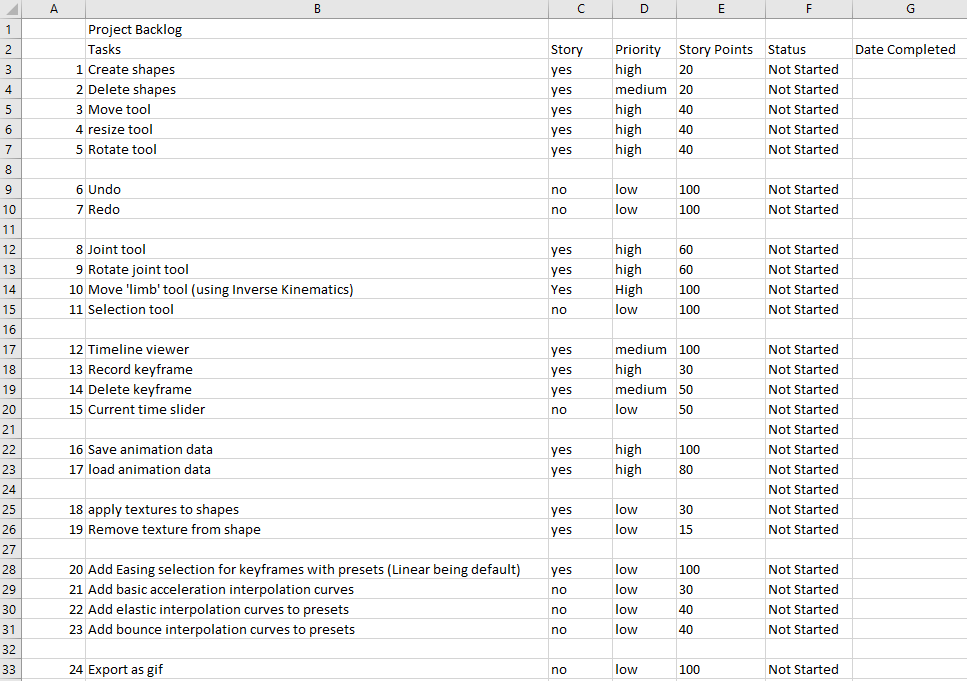
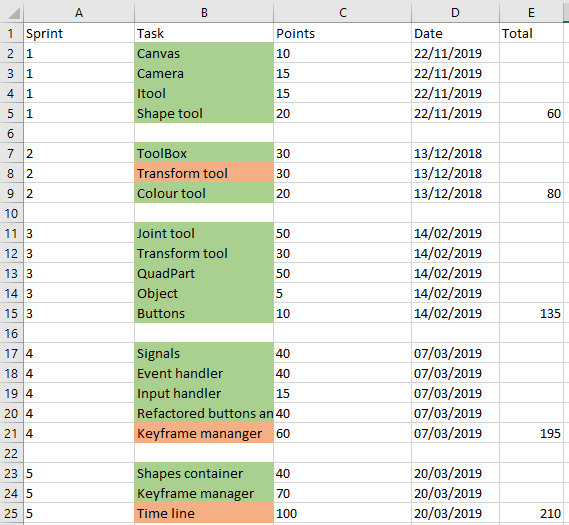
# Testing

## Testing Strategy

## Results

# Conclusion

# Appendices



Appendix 1 Initial backlog

Appendix 2 Sprint log

# 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].