# Final year project: Animation software

# Introduction

# Background

## Motivation

* Interest in how software is designed
* Interest in animation
* Something 2D to keep it simple

## 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 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 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.

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*.

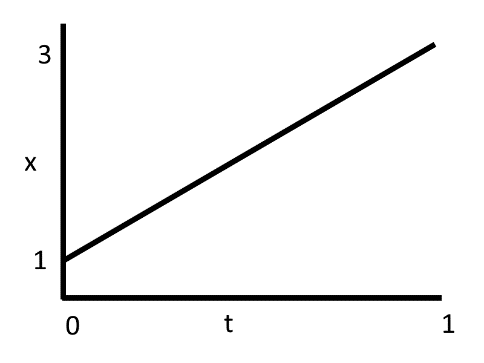


Fig 1. Blending function for x

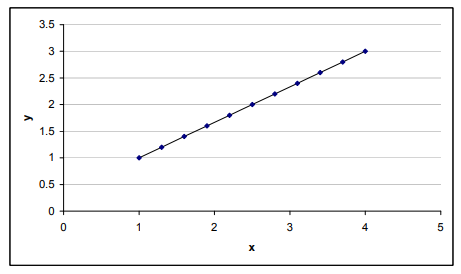


Fig 2. From page 5. <https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf>

### 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

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.

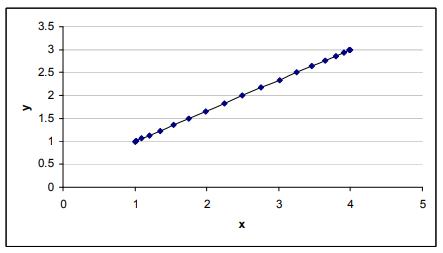
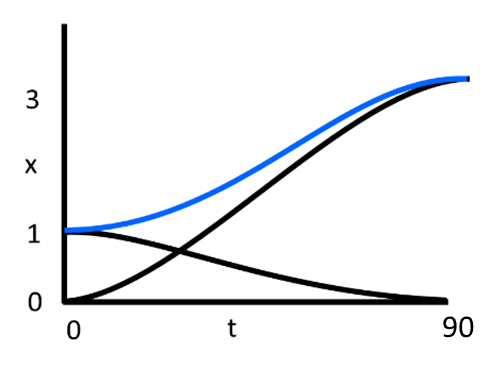


Fig 4. From page 10. <https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf>

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.

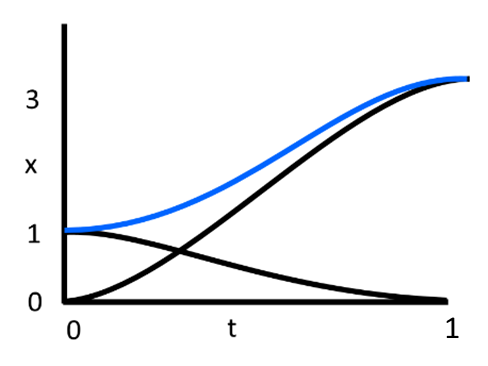


Fig 5. Blending function for x. The 2 black curves are and

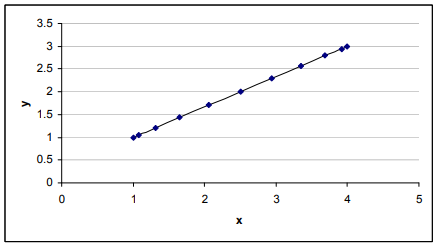


Fig 6. From page 14. <https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf>

These methods of interpolation are useful for certain circumstances and give us 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 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.

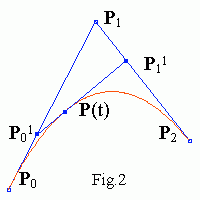


Fig 7. Example of quadratic Bezier curve with interpolation. From: <https://www.ibiblio.org/e-notes/Splines/bezier.html>

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 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 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.

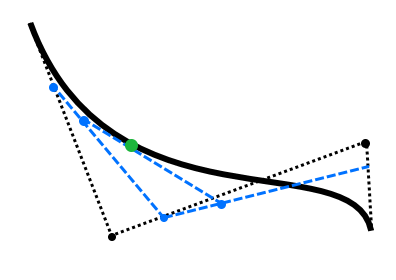
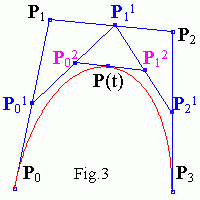


Fig 9. Example of cubic Bezier curve

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 start to become more computationally expensive since were 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.

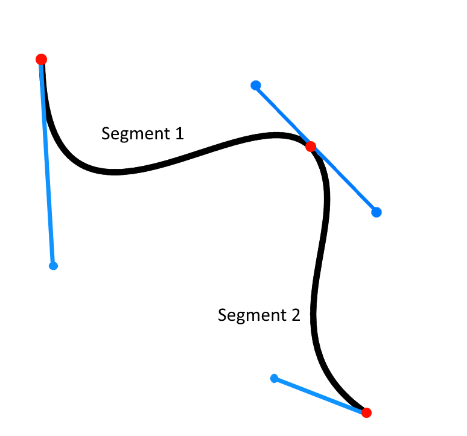


Fig 10. Piecewise Bezier curve

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 of 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.

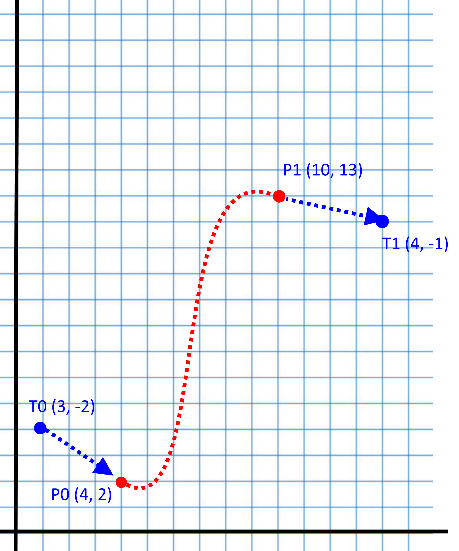


Fig 11. Hermite Curve

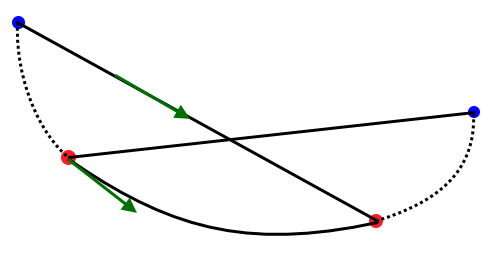
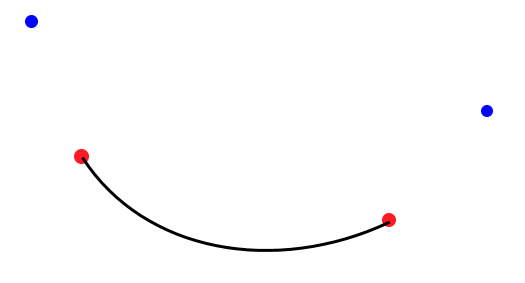
To go from individual Hermite splines to Catmull-Rom requires replacing the tangents. This is achieved by creating a tangent from adjacent control points e.g. the tangent for P1 would be (P2 – P0). We can also control the tension of the curve by adding a constraint to the tangent, α, which controls the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

Fig 12. Catmull-Rom spline



P-1

P0

P1

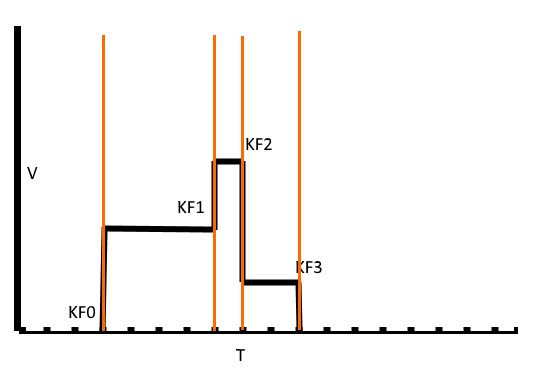
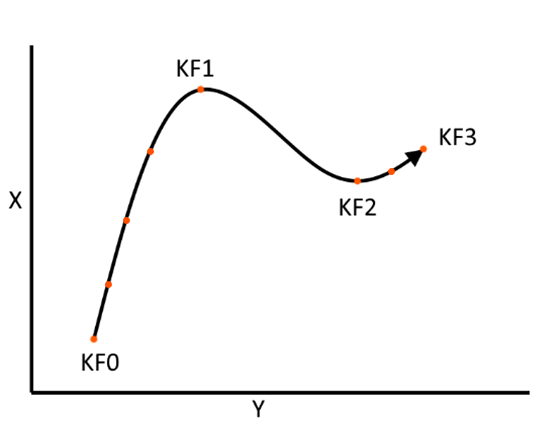
P2

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 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.

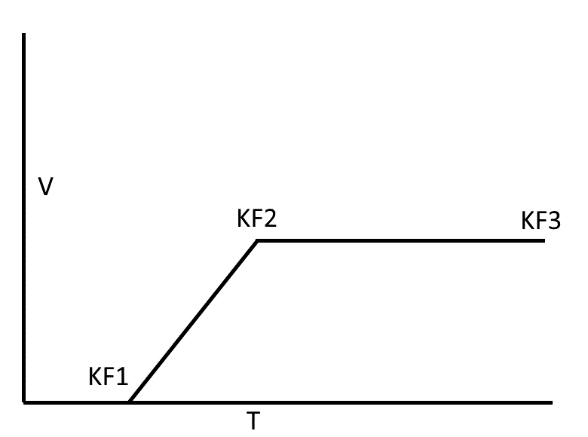
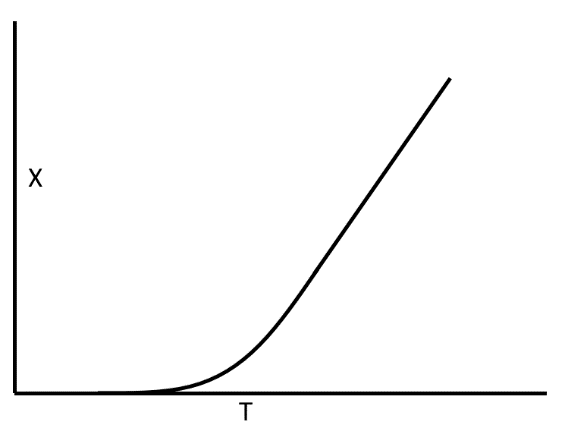
****

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.

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

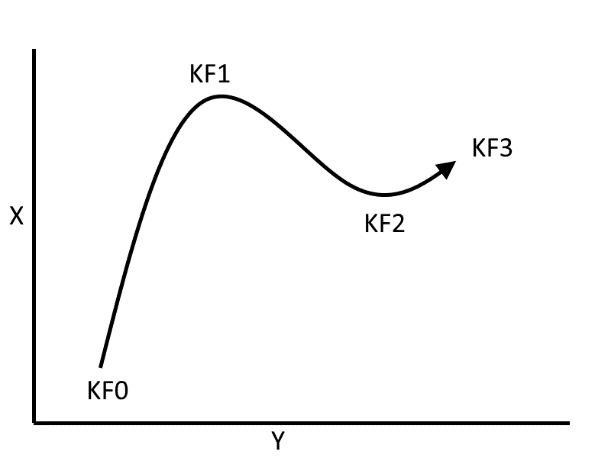
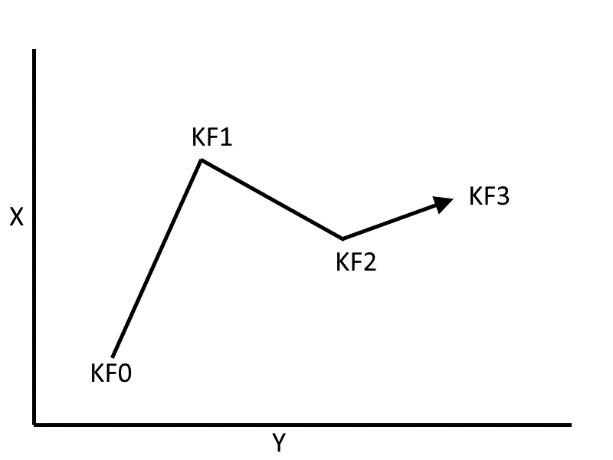
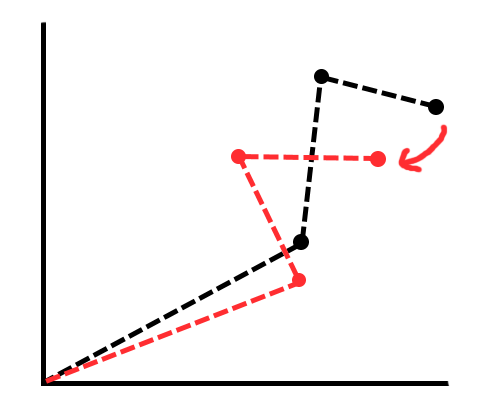


Fig 15. Example of straight and curved paths that an animated object might follow

## 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 the move the “hand” or end-effector and have the “arm” follow smoothly without breaking.

**Figure 1: 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, 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

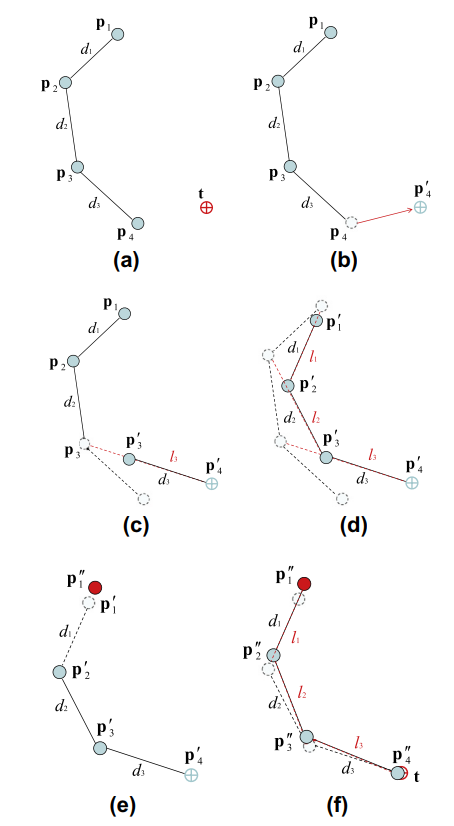
There are multiple ways to solve inverse kinematics including: algebraic methods and iterative methods such as the Jacobian inversion method and FABRIK method. The algebraic solution involves calculating the end-effector position using trigonometry, however, if the degree of freedom value increases so 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 row 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 joints position (O). To do this we translate them by the difference in rotation multiplied by a timestep (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 bodies 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 and much smaller amount 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 2. From: Aristidou, A. and Lasenby, J. (2011). FABRIK: A fast, iterative solver for the Inverse Kinematics problem. *Graphical Models*, 73(5), pp.243-260.**

## 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

* Upon starting this project, I created a backlog with set milestones to complete

## Development

* Started with the draw shape tool
* Then the transform tool. Started with translation. Moved onto scaling and realised using sfml shapes weren’t the right choice for a transformable object due to how the origin position works. Created quad part class as an underlying “shape” for actual shapes to be transformed onto. Quad part provides better control over 4 vertices and a centre position that in return makes coding scaling much simpler. Next came rotation which is now just a case of doing matrix multiplication.
* Added a colour tool
* Added joint functionality. Quad parts keep track of parent and child objects, this results in a tree like structure that could be used for Inverse Kinematics
* Whilst making the tools I needed a way to switch between them e.g. using buttons. The buttons them self were simple to implement, however, having objects respond to the buttons being clicked added some complications.
* Call back functions seemed like the most straightforward way of giving functionality to a button. If a button is clicked call a function. Thus, came the event handler and signals.
* Starting with signals, these contains a map of call back functions that can be notified. A signal is given a class to act as a trigger that other classes can connect call back functions to. The complicated part was making signals be generic as to work for different functions with varying arguments.
* The event handler was created as a result of UI interactions with input e.g. the mouse. Lets say I have a button: having to pass said mouse position and button event to all UI elements that need them could result in a lots functions that pass information up and down components of classes. The event handler is a singleton class which can bypass these “levels” by registering signals to an instance of the event handler. This makes sense from the programmer’s point of view since I if I want to add a button, I shouldn’t have to worry about also having to pass input data to it.
* This also resulted in a rework of how I handle input. Before, I SFML would poll the events and I would code the response inside a switch case. Now the input handler has multiple signals that are registered to the event handler for specific events such as: mouse moved, mouse wheel scrolled, and mouse click that are notified on inside the switch case – keeping the code compact.
* From one point of view this makes passing data from input to UI to a corresponding action much easier. However, say we have 2 classes that need to send and receive data from each other, if both register the signals to the event handler and then the other class tries to connect a call back to that signal, the code will result in an error since one of those signals doesn’t exist yet since the classes are made one after the other. Therefore, as to keep the code clean, only classes that need to use the event handler should use it i.e. the input handler and any corresponding classes that use input; but not for handling events between classes, in which their signals can be connected wherever their owner is declared.
* Next is the, not to be underestimated, task of adding animation via key framing. Making this proved difficult as there isn’t exactly many examples of “How to code key framing” out on the internet. So I started with a way to record shape data from the Quad Part class in a key frame manager class. Which also presented another problem: if I am to store data for each shape, I need a way to give the data back to said shape. Pointers seem like a good option, however, this could result in a lot of memory management to keep the pointers valid since shapes could be deleted. The solution is a shapes container class that has several methods for dealing with shapes e.g. new shape, get shape and get all shapes. These methods all deal with an id for shapes that remain constant as long as that shape exists.

## Testing and Report

# Design

## Introduction

## UML

## UI Designs

# Testing

## Testing Strategy

## Results

# Conclusion