Animation of Avatars using Human Motion Capture Data

Yi Xiang Ngam

Supervised By:
Dr. Aphrodite Galata

A PROJECT REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR'S OF SCIENCE
IN THE FACULTY OF COMPUTER SCIENCE

Abstract

Motion graphs are a widely used tool in various applications. This report presents a detailed exploration of the implementation of motion graphs from the ground up, including solutions to any issues encountered during the process. An experimentation phase is conducted to document the effect of parameters on performance, in order to identify the best parameters. Finally, the report evaluates the speed, accuracy, and effectiveness of motion graphs. The results of this evaluation provide insights into the efficacy of motion graphs, making this report a valuable resource for anyone seeking to implement motion graphs in their work.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgement

I would like to express my sincere appreciation to my supervisor, Dr. Aphrodite Galata, for their invaluable guidance, support, and encouragement throughout this project. Their expertise and insights have been instrumental in shaping my understanding and approach to the subject matter, and their dedication has helped me to achieve my academic goals.

I would also like to extend my gratitude to my parents, whose support and belief in my abilities have been a source of motivation. Their sacrifices have enabled me to pursue my academic aspirations.

Finally, I am grateful to my friends who have offered their encouragement and support during this journey. Their contributions have made this experience both rewarding and enjoyable.

Thank you all for your contributions to my personal and academic growth.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

1. **GUI** - Graphical User Interface
2. **Mocap** - Motion Capture

Chapter 1

Introduction

1.1 Overview

The rapid advancement of digital technology has made computer graphics an essential component in various industries, including scientific visualisations, advertising, entertainment, and virtual reality. In the field of computer animation, generating animations for interactive characters, particularly in the video game industry, poses a significant challenge that computer graphics can address. With the increasing demand for immersive experiences that feature fluid and realistic character movements, numerous methods have been developed to tackle this challenge. Among these approaches, motion graphs have emerged as a prevailing solution in recent times (Kovar et al., 2002), providing a robust and efficient technique for creating animations that look and feel natural.

Although motion graphs have become a popular technique in the video game industry and are a core concept in game engines such as Unity's Mechanim Animation System [1] and Unreal Engine's Animation Blueprint [2] there is a scarcity of resources and guides on building and deploying motion graphs from scratch. Similarly, while there are many libraries available to parse and process motion capture data, finding a comprehensive guide on parsing motion capture data from the ground up is challenging. Therefore, this project and report aim to fill that gap by providing a comprehensive overview of the theory and concepts behind motion graphs and motion capture data, along with the techniques required to construct and deploy motion graphs from the ground up. The report will also detail the problems and challenges encountered during the project's development and the solutions used to overcome them. Additionally, this report includes some practical tips and tricks when implementing motion graphs that were not discussed in the original paper, making it a valuable resource for those interested in motion graphs and their implementation.

1.2 Aims & Objectives

This project has several objectives. Firstly, this project aims to gain a comprehensive understanding of implementing and deploying motion graphs while investigating potential challenges and their solutions. Secondly, this project aims to provide readers with a comprehensive overview of theoretical concepts and practical aspects of building motion graphs from the ground up, as well as practical tips and tricks for implementation.

To achieve these aims, the project will use fundamental tools such as C++ and OpenGL to accomplish the following objectives:

- Construct a robust and interactive virtual environment
- Parse and process motion capture data
- Construct and deploy motion graphs

After the successful implementation of the program. We will test the system extensively to document the effects of various parameters on the quality and performance of the generated motions. This will allow us to find the optimal parameters for the system.

Chapter 2

Background

2.1 Rotations

Rotations are a key concept when dealing with motion capture data, as they are used in encoding joint angles. There are three main methods to represent 3D rotations: Euler angles, rotation matrices, and quaternions. Each method has its advantages and disadvantages and specific applications in computer graphics and animation.

2.1.1 Euler Angles

Euler angles are a way of representing rotations as a sequence of three angular rotations around the principal axes (x, y, and z) [3]. While they are intuitive and easy to understand, they can suffer from a problem known as gimbal lock [4]. This occurs when the alignment of two rotational axes causes the loss of one degree of freedom. As a result, you can no longer rotate the object independently around all three axes, and one of the rotation axes becomes redundant. In computer graphics and 3D animations, gimbal lock can lead to unexpected behaviour and difficulties when animating objects [5]. It can cause sudden jumps in rotation or limit the range of motion, making it challenging to create smooth and realistic animation [6].

2.1.2 Rotation Matrices

Rotation matrices are a way of describing rotations as 3x3 orthogonal matrices that transform coordinate vectors through matrix multiplication [7]. They are widely used in computer graphics and linear algebra but can be computationally expensive and require normalisation to maintain orthogonality [5]. If multiple rotations are compounded over time, the rotation matrix may lose its orthogonal properties due to floating-point arithmetic errors or rounding inaccuracies [8]. This can cause the matrix to become skewed or distorted, resulting in inaccurate transformations or scaling effects in the object being transformed [6]. To prevent this, it's necessary to normalise the rotation matrix periodically to maintain its orthogonality, ensuring it accurately represents rotations without introducing unwanted effects such as scaling or shearing. This is especially important in applications such as computer graphics [5] and robotics where precise control over transformations is critical.

2.1.3 Quaternions

Quaternions are a more advanced representation, consisting of one real part and three imaginary parts, represented as q = w + xi + yj + zk [9]. They are widely used in 3D applications, such as computer graphics [4] and robotics, to represent and manipulate rotations and orientations efficiently. Unit quaternions are used to represent 3D rotations, expressed as q = cos(θ/2) + (xi + yj + zk) * sin(θ/2), where θ is the rotation angle and (x, y, z) is a unit vector along the rotation axis. Quaternions avoid gimbal lock, provide smooth interpolations (slerp), and are computationally efficient compared to other rotation representations like Euler angles or rotation matrices. However, they can be less intuitive to understand and visualise.

2.2 Motion Capture Data

2.2.1 Skeletal Animations

Computer animation, the process of generating dynamic imagery through the manipulation of digital models or objects over time, creates a compelling illusion of motion and has become an integral component in various industries such as film, television, video games, advertising, and virtual reality. Numerous techniques exist for achieving computer animation, each possessing distinct advantages and applications, including keyframing, procedural animation, and physics-based animation. One prevalent technique is skeletal animation.

Skeletal animation necessitates the construction of an internal hierarchical structure, referred to as a skeleton, which is composed of joints and bones for a character or object. Within the skeleton, joints represent points of articulation where sections of the object can rotate or translate relative to one another, while bones provide the structural framework connecting the joints.

The following key concepts underpin the process of skeletal animation:

1. Root Joint: The starting point of the hierarchy, also known as the base or world joint, acts as the parent joint for the entire skeletal structure. It typically represents the character's centre of mass or the base of the spine.
2. Parent-Child Relationship: This relationship between joints determines the sequence in which transformations (translation, rotation, and scaling) are applied. When a parent joint moves or rotates, all its child joints and the attached geometry move or rotate with it, simulating the effect of real-world articulated structures, such as limbs and appendages.

3. Local Space: Each joint in the hierarchy has its local space, which represents the joint's position, rotation, and scale relative to its parent joint along the bone's axis. Manipulating joints in their local space allows for more intuitive control over the character's movement, as it accounts for the specific orientation and constraints of the individual joint.
4. Global Space: Also known as world space, global space represents the joint's position, rotation, and scale in the overall coordinate system of the 3D scene. The global transformation of a joint is calculated by applying its local transformation and recursively multiplying it with the global transformation of its parent joint up to the root joint.
5. Joint Constraints: To ensure more realistic and plausible animations, joint constraints can be applied to limit the range of motion for specific joints, preventing unnatural or impossible movements. These constraints often derive from the biomechanical properties of the character being animated.

2.2.2 Motion Capture Data

Motion capture, commonly known as mocap, is a technique used to record the movement of real-life actors or objects and convert it into digital animations. This process involves capturing position and orientation data from markers or sensors placed on the actor's body or the object being tracked, utilising tracking systems such as optical, magnetic, or inertial. The recorded data consists of joint angles, positions, and rotations over time.

Key aspects of motion capture include:

1. Sampling Rate: This refers to the frequency at which the position and orientation data of markers or sensors are recorded, typically measured in frames per second (fps) or samples per second (Hz). A higher sampling rate captures more detailed and accurate motion, albeit generating larger data sets.

2. Joint Angles and Positions: The primary components of motion capture data are the position and orientation (rotation) information of markers or sensors over time. Calculated joint angles and positions are derived from this data and then mapped onto the digital character's skeleton, which drives movement and deformation.

2.2.3 Motion Capture File Formats

Motion capture data is available in various formats, each with its specific structure and characteristics. Common mocap file formats include BVH (Biovision Hierarchy), FBX (Filmbox), C3D (Coordinate 3D), and AMC/ASF (Acclaim Motion Capture). This report focuses on the AMC/ASF file formats as they are the formats used in the project.

AMC/ASF is a file format developed by Acclaim Entertainment, Inc. for storing motion capture data [10]. Unlike many other file formats, the AMC/ASF file format uses two separate files: the AMC (Acclaim Motion Capture) file, which stores the motion data such as joint angles and rotations, and the ASF (Acclaim Skeleton File) file, which describes the skeletal structure including the hierarchy, joint constraints, and bone lengths of the character being animated.

2.2.3.1 The ASF File

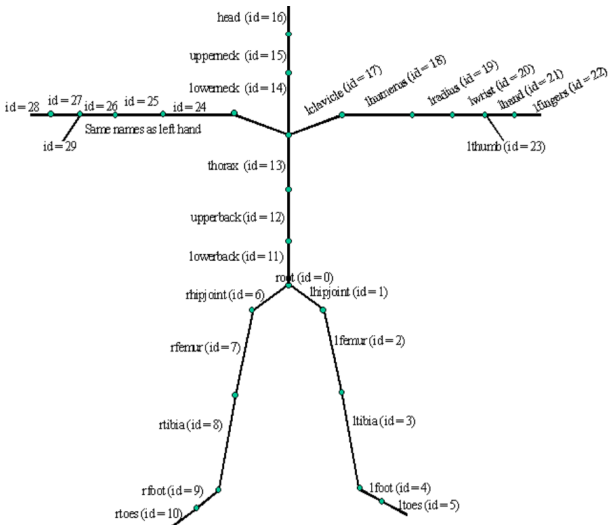


Figure 1: A typical skeleton represented by the ASF file.[1]

The Acclaim Skeleton File (ASF) is a vital tool used in motion capture applications to accurately describe the skeletal structure of a character or object. By providing critical information such as hierarchy, joint constraints, and bone lengths, it allows for the precise reconstruction of movements when used in conjunction with the corresponding Acclaim Motion Capture (AMC) file.

The ASF file is structured into several sections, which include the header, root information, hierarchy, and bone data. Each line in the file provides specific data related to the character's skeleton, making it easy to understand and work with.

1. Header: This section includes version information, the character's name, and units of measurement, as well as any metadata or comments that may be present.
2. Root Information: This section defines the root joint (base of the hierarchy) and its global position, orientation, and additional properties.
3. Hierarchy: This section specifies the parent-child relationships between joints in the skeletal structure using the `begin` and `end` keywords.
4. Bone Data: This section contains detailed information about each joint in the hierarchy, including the parent joint, direction and length of the bone connecting to the child joint, degrees of freedom (DOF) for rotations, and joint constraints.

1 Image taken from http://www.cs.cmu.edu/~kiranb/animation/StartupCodeDescription.htm

2.2.3.2 The AMC File

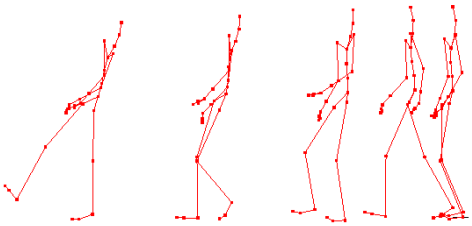


Figure 2: A typical motion represented by the ASF file.

The Acclaim Motion Capture (AMC) file is a text-based file format used to store motion data captured from real-life actors or objects. When combined with the corresponding Acclaim Skeleton File (ASF), the AMC file allows for the accurate recreation of character movements in computer animations. The following is a comprehensive description of the AMC file and guidelines on parsing it, presented in a clear and concise textbook-style manner.

The AMC file consists of several sections, including the header, followed by a series of frames, each containing motion data for the character's joints.

1. Header: This section includes version information and other metadata. Comments may also be present.
2. Frame Data: This section contains the motion data for each frame of the animation, listed in chronological order. Each frame starts with a frame number, followed by joint data for every joint specified in the ASF file's degrees of freedom (DOF) section.

2.3 Interactive Character Animations

2.3.1 History of Interactive Character Animations

Animating procedural characters involves generating character animations using algorithms, mathematical functions, and data-driven techniques, rather than relying solely on manual keyframing. This approach has evolved over time, with various techniques being developed and refined to create more natural, dynamic, and interactive character animations.

Prior to the advent of motion capture, animators primarily depended on keyframing and procedural animation techniques. Keyframing is a manual approach employed in the animation process. A notable example of a game utilising keyframing is "Prince of Persia" (1989), in which the game's creator, Jordan Mechner, filmed his brother performing acrobatics and meticulously traced over the live-action footage frame by frame to generate the character's animations [11]. Although this technique offers precise control over the animation, it can be labour-intensive and time-consuming, particularly for intricate characters and scenes.

In contrast, procedural animation involves the use of algorithms and mathematical functions to produce animation. Perlin's 1985 paper [12] introduced an early procedural animation system that harnessed noise functions to create more organic and expressive character movements. Techniques such as inverse kinematics, dynamic simulation, and physics-based methods contribute to the creation of more natural and dynamic animations. "Jurassic Park: Trespasser" (1998) serves as an example of a game that employed procedural animation for dinosaur limb movements, utilising inverse kinematics to achieve a more realistic motion. However, procedural animation often lacks the fine control and expressiveness found in keyframed animations and necessitates specialised knowledge in mathematics and physics for implementation.

The emergence of motion graphs had a significant impact on the domain of procedural character animation, addressing the shortcomings of earlier methods. In 2002, Kovar et al. [13] introduced motion graphs, a technique that leverages motion capture data to generate complex, continuous, and interactive character animations by analysing, connecting, and traversing various segments of motion data. Motion graphs facilitate the synthesis of new character animations by constructing a graph structure, connecting similar poses and velocities, and traversing different paths through the graph. This method enables the creation of intricate, continuous, and interactive animations with minimal manual intervention, offering a more efficient and flexible approach to character animation.

2.3.2 Motion Graphs

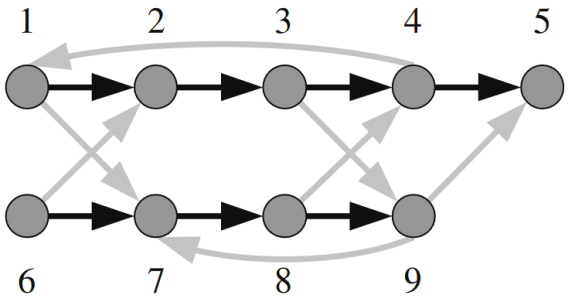


Figure 3: A simple motion graph. Black edges are frames from the original mocap database. Grey edges are transition frames connecting similar points by a blending process. (Taken from: [21])

2.3.2.1 Overview

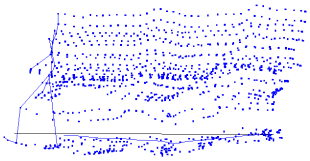
This section describes gives an overview of Motion Graphs as presented in Kovar's paper [13].

A typical motion graph consists of:

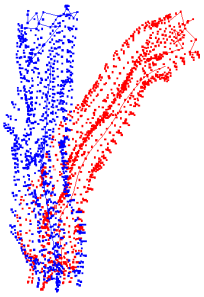
1. Motion capture data: Motion graphs start with a database of motion capture data, which consists of a collection of motion clips. Each motion clip represents a short sequence of character movements, such as walking, running, jumping, or other actions.
2. Building the graph: The motion graph is constructed by connecting motion clips based on their similarity. Each node of the graph represents the transition points. They are the points on the clip where a transition to another clip is possible. The frames in between the transition points are represented as edges in the graph
3. To create the graph, the algorithm identifies pairs of frames in different motion clips that are similar in terms of pose and velocity, then connects these frames with directed edges. These connections represent possible transitions between different clips, allowing for the generation of novel animations.
4. Transition cost: To find the best transitions between motion clips, a cost function is defined. The cost function measures the dissimilarity between two frames in terms of their pose and velocity. Low-cost transitions result in smooth, visually plausible connections between motion clips.
5. Blending: When a transition between two motion clips is chosen, the algorithm blends the frames around the transition to create a smooth connection. This blending process ensures that the generated animation does not have abrupt changes or discontinuities, resulting in more natural-looking character movements.
6. Generating animations: To create a new animation, the algorithm traverses the motion graph by following a path of connected nodes. The path can be determined randomly or based on user input, allowing for the generation of various animations using the same set of motion capture data. As the path is

traversed, the corresponding motion capture frames are concatenated, and the transitions are blended to create a continuous, realistic character animation.

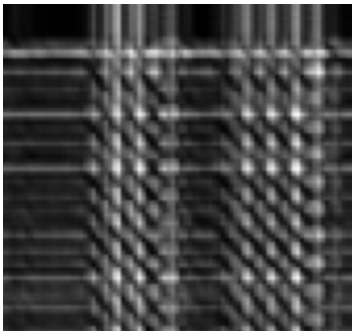
2.3.2.2 Searching for Candidate Transitions



Step 1: Find point cloud for a a pair of windows.



Step 2: Align the point clouds and calculate the sum of squared distances between each corresponding point.



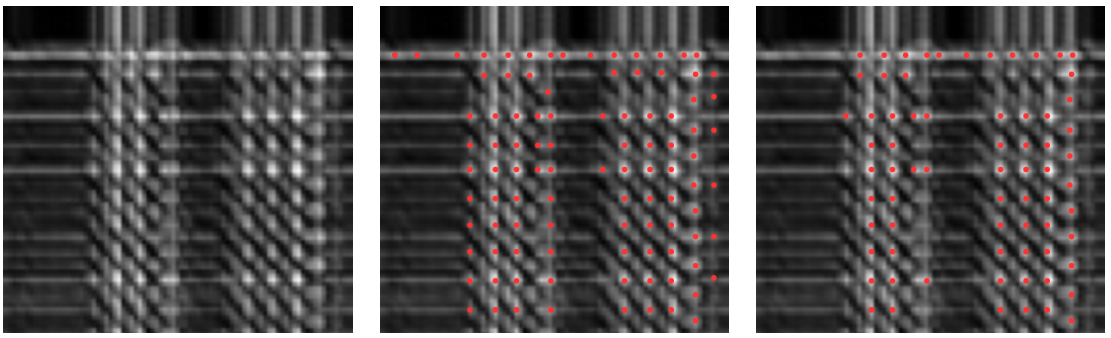
Step 3: Repeat for all pairs of windows in each clip to generate a distance map.

Figure 4: Steps to search for candidates transitions.

First, we define a window of size k as a sequence of frames bordered by frames F_i and F_i+k, where k is a user-defined constant representing the length of transitions. The effect of k will be further explored in the evaluation sections of this report. Next, for all windows in the animation database, we compare them against all other windows in the database and identify pairs of windows with a high degree of similarity, as determined by a distance function.

To calculate the distance between two windows, we first consider the point cloud generated by each clip over the window of frames. In other words, we create a point cloud of the mesh for each frame within each clip. Since we are not concerned with the original position of the clips in world space, we translate one of the point clouds to align with the other point cloud in order to find the minimum loss, considering that any arbitrary rigid 2D transformation may be applied to the point clouds.

2.3.2.3 Selecting From Candidate Transitions



Step 1: Find distance map between two frames

Step 2: Find all local minima in the distance map

Step 3: Apply thresholding to select desired points.

Figure 5: Steps to select transitions from a typical distance map.

To create a transition model, we compute the distance between every pair of windows in the database and generate a 2D distance map. We identify all the local minima of this map to extract the "sweet spots" where transitions are most favourable. However, not all local minima produce high-quality transitions, so we look for those with small error values. We can set a threshold for the error value to determine which local minima to accept. Users can either set this threshold themselves to balance between good transitions and high connectivity or use an empirically determined threshold. In the experimentation section, we will examine the effect of the threshold on the motion generated

2.3.2.4 Creating the Transitions

When the similarity between window A and window B meets the predefined threshold criteria, we create a transition by blending the frames of window A with those of window B. The initial step involves applying the suitable aligning 2D transformation to the motion. Subsequently, for frame p of the transition (-1 < p < k), we employ linear interpolation for the root positions and utilise spherical linear interpolation for joint rotations, as described by the equations provided below.

$$Root_p = p * Root_{Ap} + (1 - p) * Root_{Bp}$$
$$Quat_p = slerp(Quat_{Ap}, Quat_{Bp}, a(p))$$

Where p follows the given equation to ensure C1 continuity:

$$a(p) = 2(\frac{p+1}{k})^3 - 3(\frac{p+1}{k})^2 + 1,\ 0 \leq p < k$$

2.3.2.5 Pruning the Graph

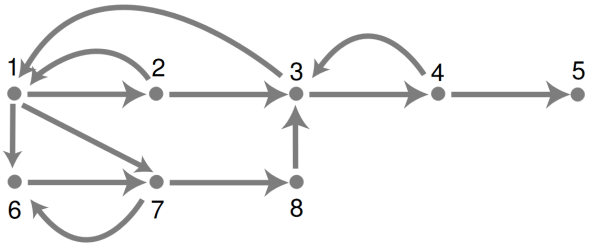


Figure 6: A simple motion graph. The largest strongly connected component is [1, 2, 3, 6, 7, 8]. Node 4 is a sink and 5 is a dead end. Source: [Kovar et al]

The current state of the graph does not guarantee that motion can be synthesised indefinitely since some nodes, called "dead ends," are not part of any cycle. When a dead end is entered, there is a limit to the amount of additional motion that can be generated. Other nodes, called "sinks," may be part of one or more cycles but can only reach a small fraction of the total number of nodes in the graph. To avoid logical discontinuities that may occur when a character transitions from one motion to another, we prune the graph.

The pruning process ensures that the graph can generate arbitrarily long streams of motion of the same type while using as much of the database as possible. To achieve this, we first compute the strongly connected components (SCCs) of each subgraph using Tarjan's algorithm. We then eliminate any edge that does not connect two nodes in the largest SCC, and nodes with no edges are discarded. We warn users if the largest SCC for a given set of labels has too few frames or if there is no way to transition between two large SCCs. In either case, the user may adjust the transition thresholds to increase the graph's connectivity.

2.3.2.6 Motion Generation

By this stage, we have finished constructing the motion graph. We will now need to extract the motion that follows a given path.

2.3.2.7 Searching for Motion

To find the graph walk that meets user requirements, we treat motion extraction as a search problem and use branch and bound to improve efficiency. The user provides a scalar function g(w, e) that evaluates the additional error when appending an edge e to

the existing path w. We define the total error of the path as the sum of the errors of all edges in the path. The user must also supply a halting condition that indicates when no additional edges should be added to the graph walk.

Our goal is to find a complete graph walk that minimises the total error. To handle the exponential growth of graph walks, we use a branch and bound strategy to eliminate branches that cannot yield a minimum. We use the current best complete graph walk as a lower bound and immediately stop any branch whose error exceeds the error of the best graph walk.

Additionally, since the branch and bound algorithm is still inherently exponential, we generate a graph walk incrementally by finding an optimal graph walk of n frames using branch and bound, then retaining a portion m of the frame found. This process continues until a complete graph walk is generated, which improves the efficiency of the overall search. The effects of parameter n (search depth) and the parameter m (keep ratio) will be further explored in the experiments section (section 4.2).

2.3.2.8 Converting Graph Walks To Motion

Because every edge on the motion graph corresponds to a piece of motion, a graph walk can be created by placing these pieces in sequence. The challenge is to ensure that each piece is oriented and located correctly. In other words, each frame of the motion needs to be transformed by an appropriate 2D rigid transformation. At the beginning of a graph walk, the transformation is the identity. Whenever the walk exits a transition edge, the transformation that moves the next clip to the end of the last clip is applied. The details of this can be found in the implementation section of the report (section 3.3.7).

Chapter 3

Implementation

3.1 Systems Design

3.1.1 Functional Requirements

This section outlines the functional requirements for the motion graphs application developed in the context of the reimplementation of Kovar et al.'s paper. The functional requirements discussed below are designed to ensure the application is user-friendly, efficient, and capable of reproducing the results outlined in the original paper.

1. Interactivity: The 3D environment generated by the motion graphs application must offer a high level of interactivity for the user. This includes the ability to navigate the virtual space, manipulate the camera to look around and interact with objects and characters within the environment. Furthermore, users should have the ability to draw paths for characters to follow, enabling them to direct character movements and actions within the environment. Ensuring interactivity enhances the user experience and fosters an engaging and immersive environment for exploration and experimentation.
2. Customizable Animations: The motion graphs application should provide users with the capability to utilise various motion capture databases containing different types of movements, allowing them to select the most appropriate motions according to their specific needs. Additionally, users should be able to configure the properties of the generated motion, such as realism, responsiveness, and performance, to create custom animations to suit their needs. This level of customization facilitates greater flexibility and adaptability in designing and implementing character animations.
3. High-quality Animations: The animations produced by the motion graphs application should be accurate and robust, ensuring that characters closely follow the user-defined path while maintaining natural and fluid movement. Smooth transitions between motion clips and minimal visual artefacts are essential for creating a realistic and engaging experience. Advanced techniques such as motion blending, interpolation, and the motion graph construction methodology proposed by Kovar et al. should be employed to achieve seamless animations, contributing to the overall visual appeal and believability of the interactive 3D environment.

4. Loading Custom Paths: The motion graphs application should provide users with the ability to import custom paths for characters to follow, granting them greater control and personalization within the interactive 3D environment. By incorporating user-defined paths, characters can be guided through specific routes or sequences, allowing users to shape the narrative, direct character actions, and achieve specific objectives or goals. This functionality enhances the user's creative input and enables a more dynamic and tailored experience within the 3D environment.

3.1.2 Non-Functional Requirements

This section outlines the non-functional requirements for the motion graphs application developed based on Kovar et al.'s paper. Non-functional requirements address aspects such as performance and usability of the application.

- Performant: The motion graphs application should exhibit real-time performance, ensuring that it runs smoothly without any noticeable delays or sluggishness. As interactive animations are a key component of the application, it is crucial that the application maintains a consistently high frame rate and responds promptly to user inputs.
- Easy to use: The motion graphs application should be intuitive and straightforward to use, allowing users to easily navigate the interface, access various features, and perform desired actions without excessive guidance or training. The user interface should be designed with clear, well-organised elements, and incorporate visual cues and feedback to guide users through the application.

3.1.3 Architecture Overview

This section provides an overview of the motion graphs application's architecture, focusing on its main components, relationships, and interactions. To illustrate the structure, we include a conceptual UML class diagram that highlights the critical components and interactions, offering a high-level understanding of the design.

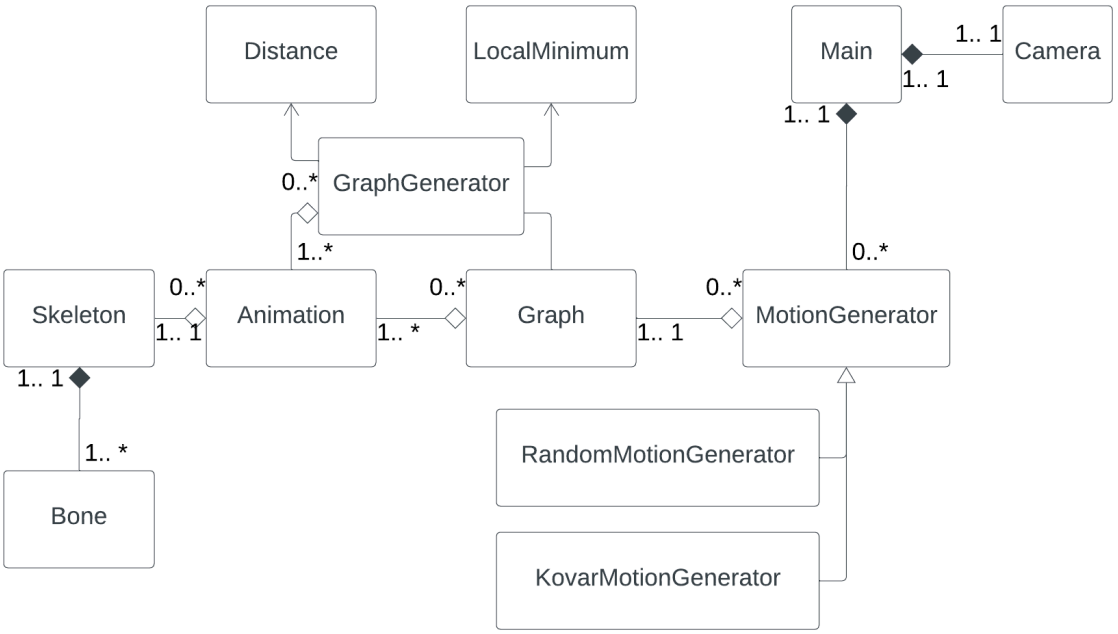


Figure 7: Simplified UML class diagram of the project.

The full UML class diagram, representing the codebase with over 3000 lines of code, is too large to fit within this section. The complete diagram is available in the appendix for readers seeking a more detailed view.

3.2 Technologies Used

Core Technologies

- C++ - The programming language is widely used in computer graphics due to its efficiency and ability to manipulate memory directly. It is commonly used for developing high-performance graphics applications and games, and is often used in conjunction with graphics libraries and frameworks to create complex 3D graphics and visual effects.
- OpenGL - The Open Graphics Library (OpenGL) is a cross-platform, open-source graphics API (Application Programming Interface) that enables us to create high-performance 2D and 3D graphics applications. It provides a set of functions for creating and manipulating graphics objects, such as textures and 3D models, and allows for low-level control over the hardware, making it popular for graphics programming and game development. OpenGL is widely used in various industries, including video games, virtual reality, CAD, and scientific visualisation.
- Visual Studio 2022 - Visual Studio 2022 is a popular Integrated Development Environment (IDE) for C++ developers. It provides a range of tools and features for writing, debugging, and deploying C++ applications, and offers improved performance and collaboration capabilities.

External Libraries

- GLFW: GLFW is an open-source, cross-platform library that provides a simple API for creating windows, contexts and surfaces, receiving input and events, and handling OpenGL rendering.
- GLAD: GLAD is a C/C++ library that loads OpenGL function pointers at runtime, making it easier to use OpenGL in modern C++ applications. Together, GLFW and GLAD provide a lightweight and efficient way to develop OpenGL applications.
- Boost: Boost is a set of C++ libraries that provides solutions for a variety of programming tasks, including data structures, algorithms, multithreading, and more.
- IMGUI: IMGUI (Immediate Mode Graphical User Interface) is a lightweight, fast, and portable library for creating graphical user interfaces in C++.
- STB: STB is a collection of single-file libraries for C and C++ that provide useful utilities for graphics programming, such as image loading and manipulation, font rendering, and more.
- GLM: GLM is a header-only C++ library for mathematics related to OpenGL programming, such as vectors, matrices, and quaternions.

3.3 Implementation Details

The practical implementation of each component introduced in the architectural overview is described in detail in this section. The report provides an explanation of how the concepts presented in section 2 were put into practice, highlighting any discrepancies between theory and implementation. Any problems encountered during implementation are discussed, and the solutions employed to overcome them are elaborated upon. Additionally, this section shares useful tips and tricks for those seeking to implement motion graphs from scratch.

3.3.1 Bone Class

The Bone Class encapsulates information pertaining to the bones, encompassing joint angles, local transformations, and global transformations. There are primarily three methods for storing rotation data: Euler Angles, Rotation Matrices, and Quaternions. Each of these representations carries its own set of pros and cons, as detailed in section 2.1. Initially, this project employed Euler Angles to depict rotations, since that is the format used in ASF/AMC files, without considering the differences between the representations. This choice resulted in difficulties when blending different frames, with instability and sporadic 360-degree rotations being the most prominent issues. Consequently, the project shifted to using quaternions, which offer greater stability and numerical accuracy, thereby resolving the problem.

3.3.2 Skeleton Class

The Skeleton class holds the skeletal information from the ASF file. To parse the ASF file, we need to read the file line by line.
For each line:

- If the line starts with a colon (':'), it indicates the beginning of a new section. Keep track of the current section for context.
- If the line starts with a hash symbol ('#'), it is a comment and can be ignored.
- Otherwise, process the line based on the current section, as follows:
    - For the 'Units', 'Version', and 'Documentation' sections, simply store the corresponding values in appropriate variables.
    - For the 'Root' section, parse the properties of the root node and store them in a Root object or variable.
    - For the 'Bones' section, parse each bone's properties and create a Bone object. Store the bones in a map collection for later use.
    - For the 'Hierarchy' section, parse the parent-child relationships and update the Bone objects with the correct parent and children pointers.

3.3.3 Animation Class

The animation class stores the contents in an AMC file, as well as methods to reconstruct each frame to construct the animation. To parse the AMC file, we need to read the file line by line.
For each line:

- If the line starts with a hash symbol ('#'), it is a comment and can be ignored.
- If the line contains a single integer, it indicates the beginning of a new frame. Keep track of the current frame number.
- Otherwise, the line specifies the joint angles for a bone in the current frame. Parse the line and read the rotation in Euler Angles based on the dof information stored in the Bone object.
- Convert the euler angles to quaternions. This is easily done with the glm::quat(eulerXYZ) method provided by the GLM library.
- Store each joint rotation of a frame in a map collection
- Store each frame in a list

Upon successfully parsing the AMC file, it is now necessary to reconstruct the animation by utilising the data obtained from both this class and the skeleton class.

Constructing the animation:
Constructing the animation utilises the bone and hierarchy information from the skeleton class, as well as the joint rotational information in the animation class to find the final transformation of each bone and the final coordinates of the skeleton vertices for each frame. Before calculating the final transformation of each joint, you need to create some initial quaternions (C, and Cinv), and offset vector B using data from the

Bone object. These transformations help convert the motion data into a form that can be used to animate the skeleton.

- Quaternion C: This quaternion is derived from the rotation axis specified in the ASF file for each segment.
- Quaternion Cinv: This is the inverse of matrix C. After you've calculated C, find its inverse and save it as Cinv.
- Vector B: This matrix is derived from the translation offset of a segment from its parent segment. The translation offset is the $direction \times length$ of the parent segment, and it represents the position of the child segment relative to its parent. For the root segment, we can use the skeleton's global position as the offset.

Calculate local transforms: For each frame, use the prepared vectors and quaternions (C, Cinv, and B) and the quaternion M (the joint rotation for the frame) to calculate a local transform (L). This local transform represents how the segment moves in relation to its parent segment.

- Since we are applying rotations and translations to obtain the local transform, we would first need to convert the quaternions and offset vectors into a compatible form. We do this by converting them into homogeneous transformation matrices.
- To compute the local transform (L), multiply the matrices in the following order:
  L = B * C * M * Cinv

Combine transforms: To find the final transformation for each segment, multiply the local transforms of the segment and its parent segments in the hierarchy. This gives you the global transformation needed to animate the skeleton.

- Combine local transforms: To calculate the global transform (G) for a segment, multiply the local transforms of the segment and its parent segments in the hierarchy. Multiply the local transforms in order, starting from the current segment and moving up to the parent segment.

  Equation: $G = L_{root} \times L_{parent1} \times L_{parent2} \times \ldots \times L_{current}$

However, a simplification can be made:

- Starting from the root segment, traverse the skeleton hierarchy, visiting each segment and its children segments in order.
- Traversing the hierarchy from the root allows us to store the global transform (G) for each segment, which can be used directly by its children segments.

  Equation: $G = G_{parent} \times L_{current}$

3.3.4 Distance Calculation Class

The method for calculating distance, as implemented, closely aligns with the approach outlined in section 2.3.2.2. Nevertheless, it is important to address several practical considerations that arise during the actual implementation process.

Performance
Computing the distance between each pair of frames proves to be a highly resource-intensive procedure, as it exhibits a complexity of O(n^2), where n represents the total number of frames in the database. Given that motion capture data usually consists of 60 to 120 frames per second, the frame count can easily approach the tens of thousands. Consequently, several solutions have been proposed to address these challenges and enhance efficiency.

Preprocessing & Storage
One practical approach is to preprocess the distances and save them to a file. This method ensures that the majority of the computationally intensive tasks are executed just once, allowing for swift loading into memory whenever motion graphs need to be generated. Nonetheless, this solution does not tackle the issue of the time-consuming nature of distance calculations.

Reducing the Sampling Resolution
Rather than comparing every individual pair of frames in the database, it is possible to process every Nth frame. For instance, sampling every 20 frames equates to sampling every 1/12th of a second, which accelerates the algorithm by 400 times, while also offering additional benefits, such as reduced storage requirements and faster processing times for subsequent steps. However, we need to be cautious when reducing the sampling too much as it can negatively affect the quality of the motion generated. The evaluation section elaborates on the influence of sampling resolution on the produced motion.

Simplifications
In our implementation, we focus on a skeleton without a mesh, which allows for certain simplifications in the distance calculation process. One simplification is instead of generating points from the surface of a mesh, we can use the skeleton vertices to produce the point clouds. This reduction results in more efficient calculations without compromising the quality of the generated motion.

Additionally, we can simplify the process by avoiding the calculation of the rigid transformation described in the paper [13]. Instead, we set the root of the skeleton for the first frame in both clips to have zero position and zero orientation. This adjustment further streamlines the distance calculation process, enabling us to generate motion graphs more quickly and efficiently while maintaining a high level of accuracy.

3.3.5 Local Minimum Class

This class provides a basic approach to identify local minima in the 2D distance map computed by the distance class. For each cell within the distance map, the algorithm evaluates its value against its eight adjacent neighbours. A cell is marked as a local minimum if its value is the lowest among its surrounding neighbours. Subsequently, the local minima are subjected to thresholding to select transitions that are a certain quality and above.

3.3.6 Graph Class

The Graph Class serves as a central repository for all the animations, nodes, edges, and transitions in the system. Its primary purpose is to maintain a comprehensive map collection that associates FrameID structs as keys with FrameVector structs as values.

Notably, in our implementation transition frames are precomputed and stored within the Graph Class. Precomputing and storing transition frames eliminates the overhead of generating transitional frames on-the-fly during frame-by-frame graph searches. This streamlined approach simplifies the playback of a graph search, as it only requires playing all the frames in a list rather than managing the complex, real-time generation of transition frames.

In this implementation, each frame is represented as a node, and all the possible subsequent frames that a node can access are connected by edges. This contrasts with the approach described in the paper, where transition points are stored as nodes and the frames between them as edges. The chosen implementation strategy prioritises simplicity and efficiency for the graph search algorithm, as it eliminates the overhead of tracking the current edge and managing actions at each transition point. As a result, the playback of a graph walk becomes more straightforward.

Each FrameID struct comprises several key attributes, including

- animationID: the ID of the frame's primary animation
- indexID: the frame index within the primary animation
- animationID2: the ID of the frame's secondary animation
- indexID2: the frame index within the secondary animation
- transitionMode: a boolean indicating whether the frame is a transition frame
- alpha: the blending value of the transition

The FrameID struct allows for the assignment of a unique identifier to every frame in the graph, including frames that transition frames that blend between two clips.

For non-transition frames, the attributes animationID2, indexID2, and alpha are set to -1, while transitionMode is set to false, as these properties are irrelevant in this context. In contrast, transition frames utilise all of the aforementioned attributes. In this case, animationID and indexID denote the frame from which the transition originates, whereas animationID2 and indexID2 signify the destination frame. The alpha parameter represents the interpolation amount for the transition.

The FrameVector struct encapsulates various information about a frame, such as joint angles and the nodes they connect with. This structure plays an essential role in the overall functionality of the Graph Class, enabling efficient organisation and access to crucial frame data. In line with the paper's methodology, the graph is pruned of nodes that are not part of the strongly connected component using a standard implementation of Tarjan's algorithm.

3.3.7 Motion Generator Class

This class is the parent class of the Random Motion Generator Class and the Kovar Motion Generator Class. This base class takes in a graph as input, and uses the graph traversal algorithms defined in the subclass to generate motion. As a result, the primary focus of this section is on highlighting the implementation common to both classes.

Frame Concatenation
To convert a given graph walk into a generated motion, it is essential to concatenate the individual frames by merging them in a sequential order. This process entails aligning the frames to preserve consistency in both the position and orientation of the character throughout the motion. This objective can be achieved by implementing suitable 3D transformations, including translation and rotation, for each frame. These transformations facilitate the maintenance of accurate spatial relationships among the character's body parts to ensure a seamless integration of the frames.

When transitioning from one clip to another, the first step is to normalise the next clip, C_o, so that it starts at the origin with zero translation and rotation.

1. Determine the orientation of the first frame in C_o and save it as matrix R_o.
2. Identify the position of the first frame in C_o and save it as matrix T_o.
3. Translate to the origin: $C_o = T_o^{-1} \times C_o$
4. Rotate to zero orientation: $C_o = R_o^{-1} \times C_o$

   (Note: It is vital to apply the inverse of translation before the inverse of rotation.)

Next, translate the clip to the target position and orientation, denoted as C_t.

6. Determine the orientation of the first frame in C_t and save it as matrix R_t.
7. Identify the position of the first frame in C_t and save it as matrix T_t.
8. Rotate to the target orientation: $C_t = R_t \times C_o$
9. Translate to the target position: $C_t = T_t \times C_t$

   (Note: In contrast to the earlier step, it is crucial to apply rotation before translation.)

In summary, to move the new clip from its original position and orientation, C_o, to the target position and orientation, C_t, use the following equation:

$$C_t = T_t \times R_t \times R_o^{-1} \times T_o^{-1} \times C_o$$

3.3.8 Random Motion Generator Class

The Random Motion Generator Class serves as a dummy class designed to randomly traverse the graph and generate arbitrary graph walks. By implementing this class first, we can ensure the functionality and reliability of the graph structure and path playing algorithms before proceeding with more complex operations. As it navigates through the graph, this class confirms the correctness of the graph's data structures and the playback mechanisms in place. Establishing this solid foundation is crucial for further development and implementation of more sophisticated motion generation classes, such as the Kovar Motion Generator Class.

3.3.9 Kovar Motion Generator Class

The Kovar Motion Generator Class is designed to process a graph object and generate a graph walk that adheres to a user-defined path. It employs the path searching algorithm outlined in the paper, along with the defined error function and appropriate halting criteria, to traverse the graph and produce the desired path. This implementation is consistent with the original paper, ensuring that the generated motion paths are both accurate and efficient.

In our implementation, the halting criteria are set as being sufficiently close to the final point and having traversed at least 95% of the user's path. The second condition is set to prevent the character from stopping when it passes over the endpoint in the middle of a graph walk.

The paper outlines a process for defining an error function, g(w, e), within the framework discussed earlier. The primary idea is to estimate the actual path P′ traversed by the character during a graph walk and measure the difference between P′ and the desired path P. The graph walk is deemed complete when the halting criteria are met.

A straightforward method for determining P′ involves projecting the root onto the floor at each frame, which forms a piecewise linear curve, also known as a polygonal chain. The error function g(w, e) is the sum of the squared distances between corresponding points on P′ and P at the same arc length from the start of its respective path. This error function is chosen for its efficiency, which is crucial for making the search algorithm practical. It encourages the character to make consistent progress along the path and travel at appropriate speeds for different actions, such as sharp turns and walking straight. Additionally, it does not give undue preference to one action over another.

By incorporating the defined error function and the specified halting criteria into the Kovar Motion Generator Class, a cohesive and efficient system for generating motion paths is achieved. This class effectively combines the data structure provided by the Graph Class with the algorithmic approach described in the paper, creating a streamlined process for searching the graph and producing a graph walk that aligns with the user's desired path.

3.3.10 Camera Class

The Camera class manages the camera's position and orientation in a 3D scene, enabling user interaction and movement in the environment. It allows the camera to move in different directions, rotate around specific axes, and look at specific targets within the scene. The class handles user input for controlling the camera's movement and updates the view matrix accordingly, ensuring the proper transformation of world coordinates to camera space for rendering.

3.3.11 Main

In this section, we will discuss the main function of the program, which features an interactive virtual world that allows users to engage with the environment in a dynamic and intuitive manner. The virtual world includes an interactive camera, enabling users to view the scene from various angles and perspectives, enhancing their overall experience.

One of the key functionalities provided within this virtual world is the ability to draw paths for the character to follow. This feature allows users to create custom paths and observe the character's movement along them, showcasing the effectiveness of the motion graph implementation in generating realistic and smooth animations.

To support these interactive features, the program also incorporates a graphical user interface (GUI). The GUI provides users with easy access to various settings and controls, enabling them to fine-tune the application's behaviour and better understand the impact of different parameters on the generated motion. Overall, the combination of the interactive virtual world, camera, path drawing capabilities, and supporting GUI

creates a comprehensive and engaging environment for users to explore the capabilities of the motion graph system.

3.3.11.1 World

This section provides a brief description of the scene, which is rendered using OpenGL. The floor is constructed from four points that describe its corners, and a self-created texture is applied to create a checkerboard pattern on the floor. A basic floor shader is used for rendering the floor, while a simple line shader is employed for the skeleton, rootline, and pathline.

The rendering process in OpenGL involves the use of Vertex Array Objects (VAOs) and Vertex Buffer Objects (VBOs) to efficiently store and manage vertex data on the GPU. VAOs store information about the vertex attributes, while VBOs hold the actual vertex data. These objects play a crucial role in streamlining the rendering pipeline and improving performance.

The Model-View-Projection (MVP) transform is used to convert the 3D coordinates of the objects in the scene into 2D screen coordinates. The model matrix defines the position, rotation, and scale of each object, the view matrix represents the camera's position and orientation, and the projection matrix determines how the 3D scene is projected onto the 2D screen. These matrices are combined to form the MVP matrix, which is then used to transform the vertices during rendering.

In the rendering loop, the skeleton vertices are obtained from the motion generator for every frame and are rendered on the screen. This process ensures that the motion of the skeleton is accurately displayed in the scene, providing a smooth and realistic animation.

3.3.11.2 GUI

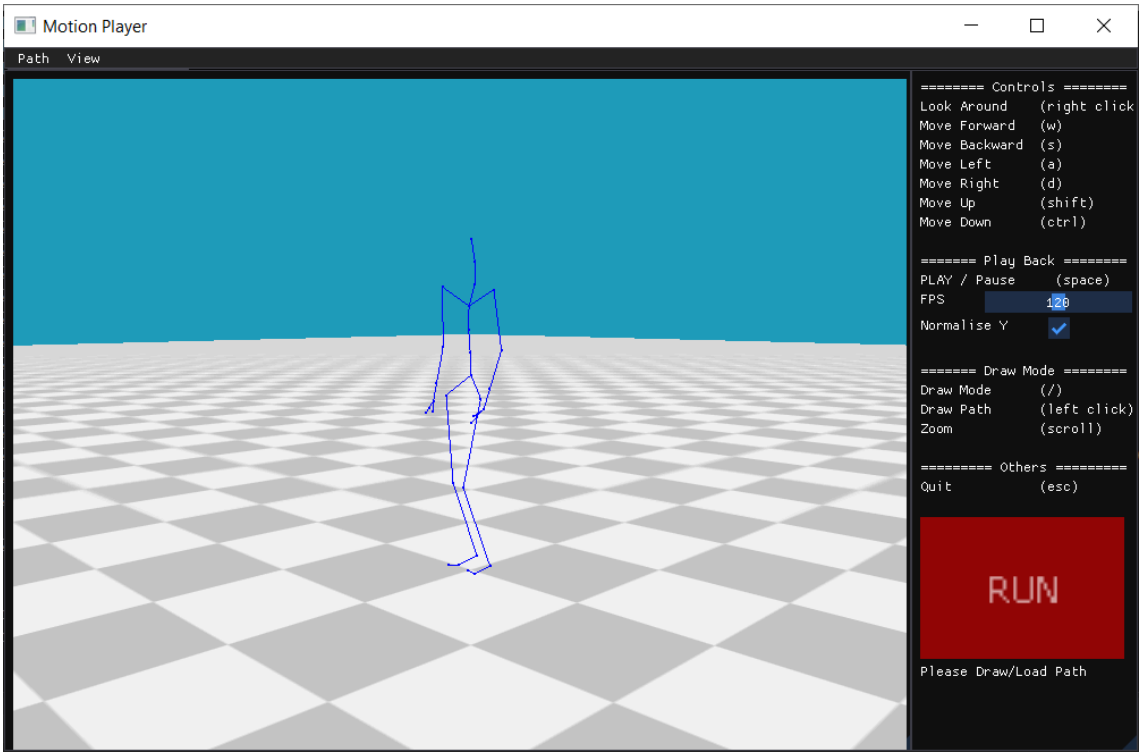


Figure 8: The GUI of the application

The GUI for the application is implemented using ImGui, a library that allows for the creation of intuitive and easy-to-use interfaces. In this application, the previous scene is rendered onto a Framebuffer Object (FBO), which is then displayed in a subwindow within the GUI. This approach enables the efficient rendering of the scene and allows users to interact with the 3D visualisation while retaining a clear view of the interface.

Another subwindow is positioned on the right side of the screen, providing a dedicated area for settings and controls. This subwindow contains various options and parameters that allow users to adjust the application's behaviour and appearance according to their preferences. By organising the settings and controls in a separate subwindow, the application maintains a clean and uncluttered interface, making it easier for users to interact with the scene and achieve the desired results.

3.3.11.3 Path Drawing

The path is defined as a list of control points. The implementation of the motion path poses an interesting challenge, as it needs to meet certain criteria. In particular, the implementation should be able to identify the position on the path corresponding to any given arc length from the start of the path due to the error function outlined in section 3.3.9

Naive Implementation

This section outlines the naive implementation first employed in the project. The user-drawn or loaded path is represented solely as a list of control points. To determine the position on the path for a given arc length 'S', a linear search is performed through the list of control points, cumulatively adding the distance between each point until the sum closely matches or equals 'S'.

However, this approach presents a few issues:

1. The search has a complexity of O(n), which is suboptimal since a typical user-drawn path usually consists of numerous control points.
2. As this algorithm would be utilised within the branch-and-bound method for path searching (i.e., executed frequently), its efficiency and practicality are significantly compromised.
3. The control points for a drawn path are affected by the coastline paradox [14], which means that the measured length of the path can vary greatly depending on the scale used for measurement, leading to inconsistencies.

An alternative solution involves precomputing the arc lengths corresponding to each control point, storing them in a list, and performing a binary search on the list. Although this results in an improved complexity of O(log n), it remains impractical for use in a performance-critical loop.

Proposed Implementation

The main idea of this solution is to have control points spaced across a fixed distance. That way, each control point is a constant arc length away from the previous control point. This allows us to index the list with O(1) complexity. The following is split into 3 sections. The first section describes how to draw the path, the second section details how to convert any arbitrary loaded path into our path, and lastly, the third section details how to find the position of a point, given an arc length distance from the start of the path. The following sections provide the pseudocode for the implementation.

Algorithm 1: Drawing the path

Data: k, a constant describing the distance between each control point
Result: path, a list of control points for the path

```
path= []
path.add(The current position of the character)
During each frame when drawing:
        E = the position of the last point in path
        P = the position of the cursor
        if distance(P - E) > k then
                vector = P - E
                vector = vector / |vector |
                new_point = x + k * vector
                path.add(new_point)
end procedure
```

The described algorithm generates the graph by monitoring the cursor's position during each frame of the drawing process. The algorithm only adds a point if the cursor is at least a certain predetermined distance away from the previous point. This approach effectively controls the distance between the control points, generating the desired path.

Algorithm 2: Convert path

Data: initial_points, a list of points describing the input path
k, a constant describing the distance between each control point
Result: path, a list of points describing the converted path

```
path= []
path.add(initial_points[0])

P = initial_points[0]
S = initial_points[0]
E = initial_points[1]
distance_remaining = k
i = 0
repeat
    if distance(P - E) >= distance_remaining then
        SE = E - S
        SE = SE / |SE|
        P = P + SE * distance_remaining
        path.add(P)
        distance_remaning = k
    else
        distance_remaining = distance_remaining - distance(P, E)
        i = i + 1

        if i + 1 >= size of initial points:
            break

        P = initial_points[i]
            S = initial_points[i]
            E = initial_points[i + 1]
    return path
end procedure
```

Fundamentally, this algorithm traverses the provided path, positioning a point at regular intervals, with each point being k units of arc length apart from the previous one. This approach effectively controls the distance between the control points, generating the desired path.

Algorithm 3: Compute Position

Data: S, the arc length from the start of the path
k, a constant describing the distance between each control point
path, a list of control points for the path
Result: P, the position along the path which is S arc length from the start of the path

```
indexf = S / k

index = floor(indexf)
remainder = indexf - index

if index + 1 < size of path then
    S = path[index]
    E = path[index + 1]

    P = S * (1 - remainder) + E * remainder

    return P
else
    return Last point in path
end procedure
```

The algorithm calculates the index "indexf" of the path by dividing the input arc length S by the constant k. As this operation yields a floating-point value rather than an integer, we identify the points indexed by the integers surrounding "indexf" and perform interpolation between these two points to obtain a precise position on the path.

Advantages

The implementation outlined above facilitates obtaining the point on the path corresponding to a given arc length with a computational complexity of O(1). By maintaining a fixed distance between control points, this algorithm also effectively mitigates the coastline paradox issue. Furthermore, this approach offers an additional benefit: it enables the user to generate smooth paths. As the path incrementally expands by a maximum length of k for each frame the user draws, the path lags behind during the drawing process, leading to the creation of smoother lines.

Chapter 4

Results and Evaluation

4.1 Evaluation

This section presents the results obtained from the motion graph implementation and evaluates them using various metrics. The evaluation aims to provide a comprehensive understanding of the performance and effectiveness of the implemented motion graph.

Qualitatively, the results can be described by observing the generated motion and assessing its smoothness, naturalness, and adherence to the given path. This type of evaluation provides valuable insight into the overall quality of the motion generated by the motion graph.

Quantitatively, there are several measures that can be used to assess the performance of the motion graph. Measures used include:

- Total Path Error: the total error between the produced path and the given path. This metric helps to determine how accurately the generated motion follows the desired trajectory.
- Average Foot Sliding per Transition Frame: Foot sliding is a phenomenon where the character's feet appear to slide on the surface instead of having solid contact with the ground. This is one of the main issues that arise when applying blending on clips to produce transitions. Minimising foot sliding is crucial for generating high-quality motion, as it contributes to the naturalness and realism of the character's movement. Hence, this metric is a good way to evaluate the quality and realism of the motion generated.
- CPU Time: the CPU time required to search the graph is another important metric for evaluating the efficiency of the motion graph implementation. This metric provides an indication of the computational resources needed for the path searching algorithm and its practicality for real-time applications.

4.1.1 Qualitative Evaluation

In this section, we present a qualitative evaluation of our reimplementation of the motion graphs algorithm, as proposed by Kovar et al. (2002). The primary objective of this algorithm is to generate character animations that follow complex and intricate paths while maintaining realistic blending and minimal visual artefacts.

To evaluate the performance of our implementation, we generated a path in the shape of the Manchester Bee icon, demonstrating the algorithm's ability to handle complex and intricate paths effectively. Additional hand-drawn paths were also used, and the generated animations closely followed the intended paths.

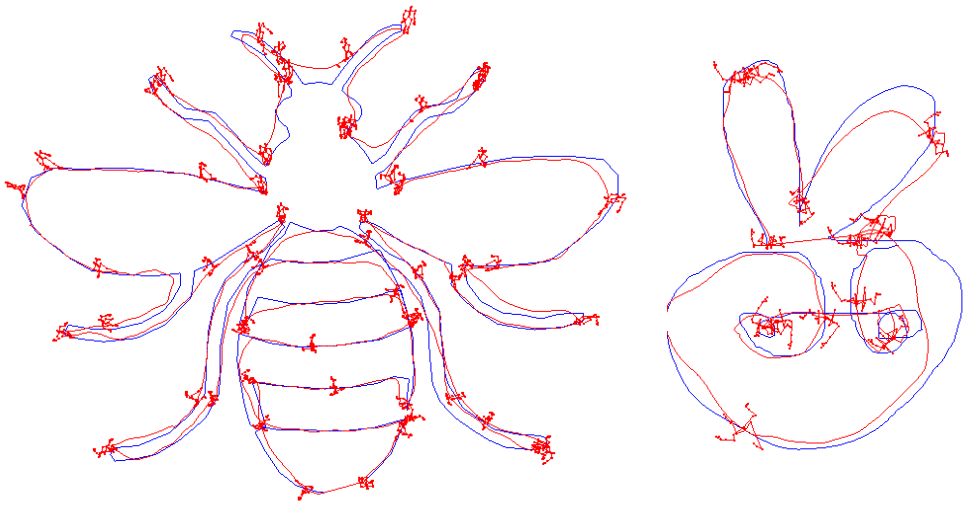


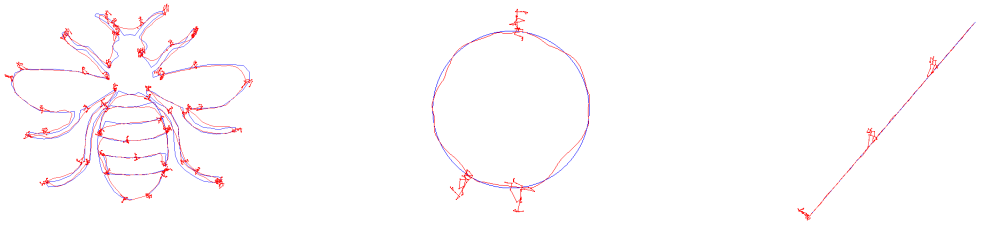
Figure 9: Generated Motion. The blue line represents the drawn path, while the red line represents the path of the character. The red line closely matches the blue line.

Our evaluation is primarily based on qualitative judgement and focuses on the following aspects:

1. Realistic blending: The generated animations exhibit no sudden shifts in velocity or joint rotations, resulting in smooth and natural character movements.
2. No discontinuities: The motion generated does not have abrupt shifts in position, ensuring a continuous and coherent animation.

4.1.2 Quantitative Evaluation

In this section, we evaluate the application quantitatively given the metrics described above. The application is run on a single-core CPU. The table below summarises the findings for various paths.



(a) (b) (c)

Figure 10: Graph walks of (a) The Manchester Bee, (b) a circle, and (c) a straight line

	Manchester Bee	Circle	Straight Line
Total Path Error	3.198e+06	2250.65	2438.29
Average Path Error per Frame	79.83	1.562	1.195
Average Footslide per Transition Frame	0.01317	0.01350	0.01472
Total CPU Time Elapsed (Seconds)	99.80	0.423	2.918
Total CPU Time Elapsed / Second of Motion Generated	0.2988	0.0352	0.1714

Table 1: Quantitative metrics for the paths in Figure 11.

Quantitatively, we have measured foot sliding and compared it against the survey conducted in the [15]. According to the paper, foot sliding below 0.021 is generally not noticeable. Our implementation's performance adheres to this criterion, indicating minimal visual artefacts like foot sliding. Furthermore, the time it takes to generate a second of motion is much lower than 1 second. This indicates that our system performs very quickly, and is able to generate motion in real-time.

4.1.3 Conclusions

In conclusion, our reimplementation of the motion graphs algorithm effectively generates character animations that follow complex and intricate paths while maintaining realistic blending, continuity, and minimal visual artefacts. Moreover, our system generates the motion swiftly, making it suitable for real-time applications. These results demonstrate the success and effectiveness of our implementation. However, there is still room for improvement by optimising the parameters for the algorithm to enhance its performance further.

4.2 Experiments

Parameter	Base Value
Threshold	60%
Transition Length	⅓ seconds / 40 frames
Search Depth	2 seconds / 240 frames
Keep Ratio	50%
Resolution	5%

Table 2: Parameter values for the base configuration used in section 4.1

This section delves into the experiments conducted to analyse the impact of different parameters on the quality and performance of the motion generated by the motion graph implementation. Through these experiments, the section aims to provide valuable insights into the effects of parameter variation on the resulting motion and to offer an interpretation of the gathered data.

The following parameters are varied during the experiments:

1. Threshold: This parameter represents the percentage of local minima accepted when computing candidate transitions. Varying the threshold value helps to understand its influence on the quality of motion and the number of feasible transitions in the graph.
2. Transition Window Size: The size of the window used to calculate the distance matrix plays a vital role in determining the granularity of the motion graph. By altering the window size, the impact on motion smoothness and the computational complexity of the graph construction can be assessed.
3. Search Depth: This parameter sets the maximum search depth for the iterative branch and bound algorithm. Investigating the effects of different search depths can provide insights into the trade-off between search accuracy and computational efficiency.
4. Keep Ratio: The proportion of frames to keep from the branch and bound algorithm determines the amount of lookahead before committing to the motion. Decreasing the keep ratio enhances the accuracy of the generated motion but comes at the cost of increased computational requirements. By varying the keep ratio, the relationship between motion accuracy and computational complexity can be examined.
5. Resolution: The sampling resolution in generating the distance matrix affects the granularity of the motion graph and the computational resources required for its construction. Exploring different resolutions helps to understand the balance between graph fidelity and computational efficiency.

This section explores the impact of varying parameters on the motion graph implementation using a predefined base configuration used in section 4.1, which is closely aligned with the original paper. The base configuration is described in Table 2. Furthermore, a fixed circular path with a radius of 25 units is used for all experiments.

Threshold	20%	40%	60%	80%	100%
Total Path Error	1.78E+06	15791	19497	6096	5944
Average Footslide per Transition Frame	0.00953	0.01457	0.01559	0.01634	0.01602
CPU Time Elapsed (Seconds)	0.19	1.518	3.419	4.71	7.396

Table 3: The effect of the threshold parameter on the motion generated

Reducing the threshold value results in an improvement in the quality and realism of the generated motion, as evidenced by the decrease in the Average Footslide per Transition Frame. However, this comes at the expense of graph connectivity, leading to a higher Total Path Error, which signifies a decreased ability to accurately follow the desired path. Furthermore, a lower threshold results in reduced CPU time, as there are fewer edges in the graph to search through. This highlights a trade-off between motion quality and graph connectivity when adjusting the threshold value.

Transition Window Size	⅙ seconds	⅓ seconds	⅔ seconds
Total Path Error	1989.95	19496.6	1.23E+06
Average Footslide per Transition Frame	0.0004	0.01559	0.01130
CPU Time Elapsed (Seconds)	75.039	3.419	0.375

Table 4: The effect of the transition window size on the motion generated.

Reducing the window size significantly enhances both the Total Path Error and Average Footslide per Transition Frame metrics. While this may initially seem promising, a qualitative analysis of the generated motion reveals that it becomes choppy, with rapid changes in speed and motion. This is a result of considering less derivative information when the window size is smaller. Conversely, increasing the window size too much can also negatively impact the graph's connectivity, as demonstrated by a rise in the Total Path Error. This suggests that finding an optimal window size is crucial for balancing motion quality and graph connectivity.

Search Depth	**120**	**180**	**240**	**360**	**480**
Total Path Error	nan	46850.5	19496.6	7567.73	7458.72
Average Footslide per Transition Frame	nan	0.01515	0.01559	0.01656	0.01631
CPU Time Elapsed (Seconds)	nan	0.68	3.419	22.279	183.471

Table 5: The effect of the search depth parameter on the motion generated.

Increasing the search depth results in an improvement of the Total Path Error, indicating better path accuracy. However, this effect eventually plateaus, as evidenced by the reduced change in error between search depths of 360 and 480. Despite this plateau, there is a significant increase in CPU time—around eight times—between these search depths. This highlights the need to balance search depth with computational cost to achieve optimal performance and accuracy.

Keep Ratio	20%	40%	50%	60%	80%	100%
Total Path Error	9975.19	10753.1	19496.6	34967.7	88418.8	143603
Average Footslide per Transition Frame	0.01584	0.01452	0.01560	0.01481	0.01480	0.01472
CPU Time Elapsed (Seconds)	4.854	2.676	3.419	2.212	2.427	3.031

Table 6: The effect of the keep ratio parameter on the motion generated.

Decreasing the keep ratio improves the Total Path Error, which can be attributed to enhanced lookahead capabilities. However, reducing the ratio too much can lead to increased CPU time due to the need for more iterations. Conversely, increasing the ratio too much also results in higher CPU time, as the algorithm must spend more effort and distance correcting errors. A 40% keep ratio is identified as a sweet spot, where Total Path Error, Average Footslide per Transition Frame, and CPU Time Elapsed are all minimised, striking an optimal balance between performance and accuracy.

Resolution	5%	10%	20%	100%
Total Path Error	19496	13089	5504	3526
Average Footslide per Transition Frame	0.01559	0.01525	0.01495	0.01442
CPU Time Elapsed (Seconds)	3.419	3.863	13.906	265.232

Table 7: The effect of the sampling resolution on the motion generated.

Decreasing the resolution leads to an increase in the Total Path Error, but it significantly reduces the CPU time. Additionally, lowering the resolution also accelerates the distance map computation by a factor of n^2. In this case, the trade-off between resolution and computational efficiency should be carefully considered based on the specific requirements and desired accuracy of the motion generation process.

Chapter 5

Summary & Conclusions

5.1 Achievements

In this project, I successfully accomplished the objectives I initially set out to achieve, gaining valuable knowledge and experience in the process. Through working with low-level tools and processes, I was able to construct a comprehensive system from the ground up, which allowed for a deeper understanding of the underlying concepts and techniques.

One of the key achievements of this project was parsing motion capture data, a crucial step in generating realistic animations. Since parsing the data is a difficult task without a lot of resources, learning this early on in the project provided the foundation upon which the motion graph system was built. Additionally, I created an interactive virtual world that enabled users to engage with the environment, observe the character's movements, and evaluate the effectiveness of the motion graph implementation.

Furthermore, I successfully implemented motion graphs, which allowed for the generation of smooth and realistic animations based on the given motion capture data. This achievement showcased the practical application of the motion graph concept in creating lifelike character movements.

Finally, I designed and implemented a graphical user interface (GUI) to support the interactive virtual world, providing users with an accessible means of controlling various settings and parameters. This addition further enhanced the user experience and facilitated a greater understanding of the motion graph system.

Overall, these achievements demonstrate the successful completion of the project's objectives and the acquisition of valuable skills and knowledge in the field of computer animation and motion graphs.

5.2 Challenges & Reflections

During the course of this project, I faced several challenges, primarily stemming from my limited experience with C++ and OpenGL prior to embarking on this endeavour. Nevertheless, this project provided an invaluable opportunity to learn a great deal about programming techniques and best practices.

Due to my initial lack of experience, the codebase turned out to be somewhat messy and inconsistent. As I progressed and learned more about proper programming practices, I realised that some of the earlier parts of the code did not adhere to these principles, which made later progress increasingly difficult and slow. This resulted in an inconsistent coding style throughout the project.

In an effort to improve the codebase, I undertook multiple refactorings to enhance its overall structure and organisation. Despite these efforts, I still believe that the codebase falls short of the level of design I would ideally like to achieve. Given more time, I would consider completely reimplementing the codebase from scratch to ensure a more efficient and well-structured design, which would help prevent the emergence of spaghetti code.

Reflecting on these challenges has allowed me to better understand the importance of adhering to best practices and maintaining a clean and well-organised codebase from the outset. This experience has provided me with valuable insights that will undoubtedly prove beneficial in future programming endeavours.

5.3 Future Work

In terms of future work, this project could be expanded upon by implementing the various improvement and variations upon motion graphs over the years and describing their results. An example of improvements are:

- **Motion synthesis from annotations [16]:** This paper allows for more control over synthesised motion by using annotated constraints, making it possible to generate motions that meet specific requirements.
- **Interactive control of avatars animated with human motion data [17]:** This work enables real-time user control over synthesised motion, making character animations more interactive and responsive.
- **Synthesising physically realistic human motion in low-dimensional, behaviour-specific spaces [18]:** This paper reduces the computational cost of generating new motions by focusing on low-dimensional spaces, optimising the motion synthesis process.
- **Fat graphs: Constructing an interactive character with continuous controls [19]:** This paper extends motion graphs by using continuous controls for character animation, allowing for smoother transitions and more diverse motion sequences.
- **Near-optimal character animation with continuous control [20]:** This work generates near-optimal character animations using continuous control and spacetime constraints, resulting in natural and efficient character animations that satisfy given constraints.

Additionally, this project could serve as a foundation for exploring various other animation algorithms, including motion matching and neural-based methods. By expanding the range of techniques, the application can potentially generate more diverse and realistic animations.

Integrating a physics engine would be another valuable enhancement, enabling the creation of physically-based animations that interact with the environment in a more convincing manner.

Furthermore, this project could be extended to develop a simple, playable game. By implementing a basic character and incorporating a third-person camera that continuously follows the character, the application could be transformed into an interactive experience. The path generation algorithm could be adjusted to generate paths for the character based on user input from a keyboard, similar to many games.

These proposed future developments not only demonstrate the potential of the current project but also open up new avenues for exploration in the realm of computer animation and game development.

5.4 Final Conclusions

In conclusion, this project has successfully met its aims and objectives, providing a solid foundation for generating interactive animations. The opportunity to work on this project has not only allowed me to gain valuable practical experience but also enhanced my knowledge and understanding of implementing a large system from the ground up.

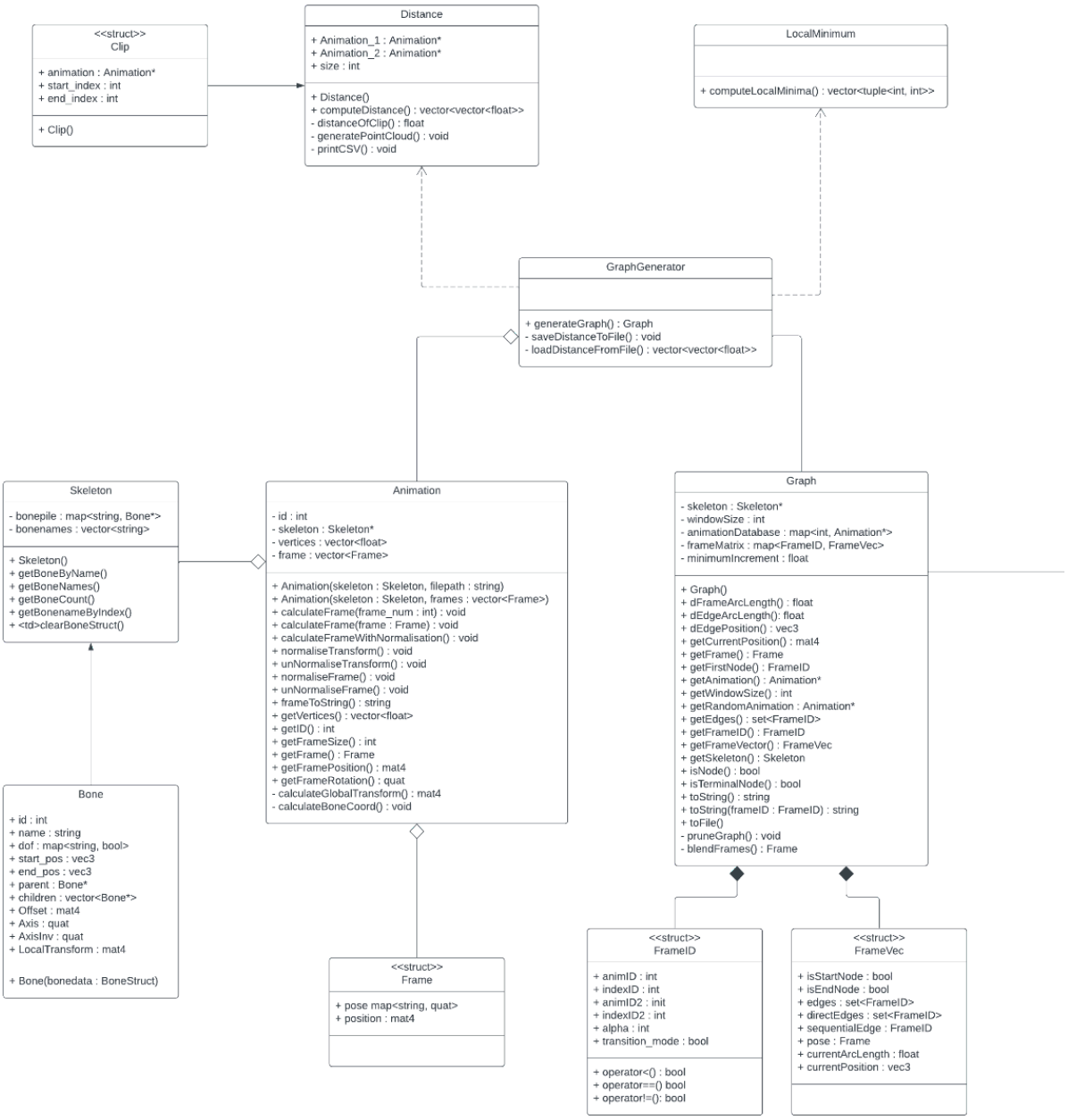
The hands-on nature of this project has been instrumental in furthering my expertise in the field of computer animation and programming. I am grateful for the opportunity to have tackled this challenging and rewarding project, and I am confident that the skills and experience gained will serve me well in future endeavours.

Bibliography

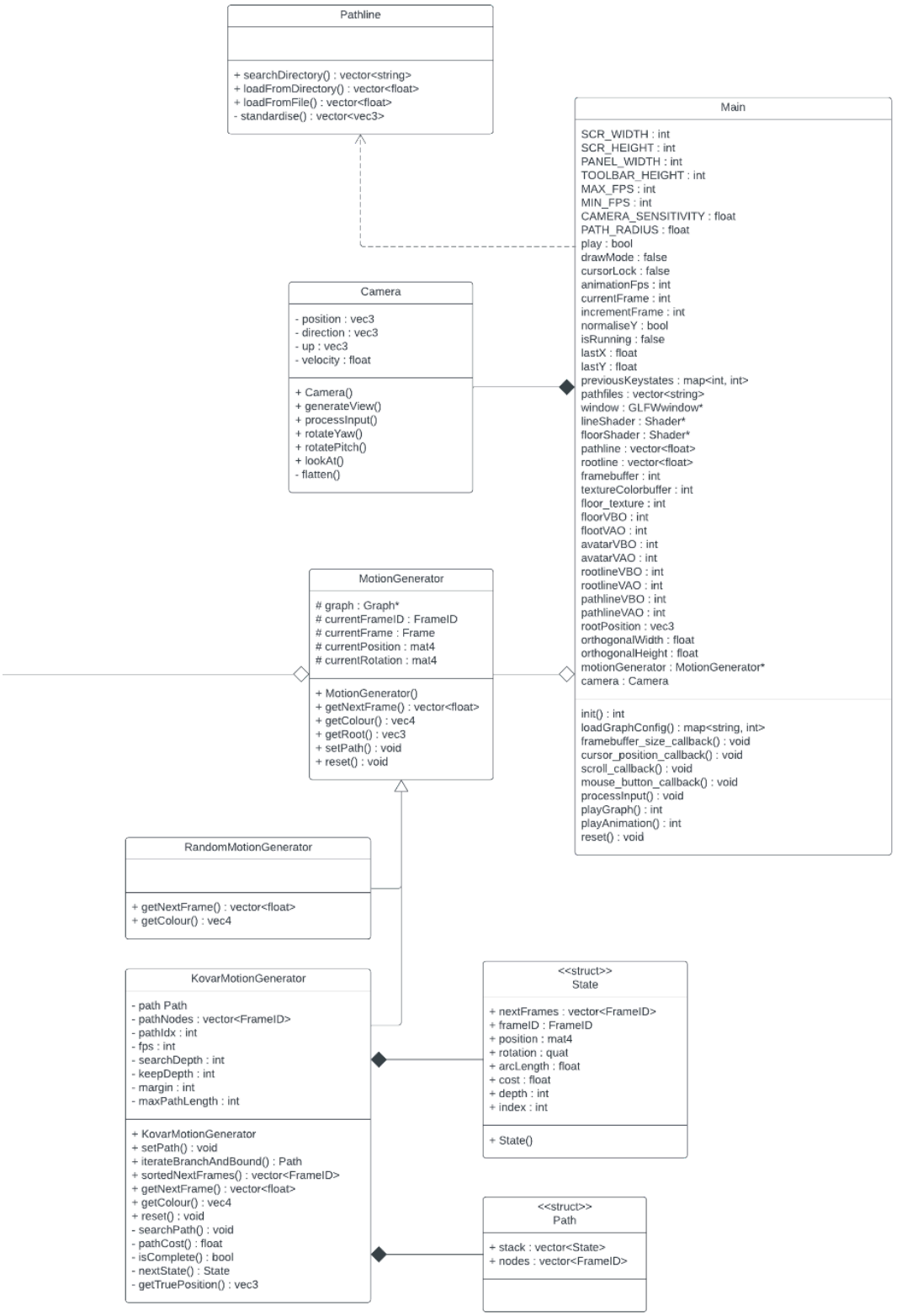
[1] “Animation Blueprints,” *docs.unrealengine.com*. https://docs.unrealengine.com/5.1/en-US/animation-blueprints-in-unreal-engine/ (accessed Apr. 23, 2023).
[2] “Unity - Manual: Mecanim Animation System,” *docs.unity3d.com*. https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html
[3] H. Goldstein, C. P. Poole, and J. L. Safko, *Classical mechanics*. United States: Pearson, 2011.
[4] K. Shoemake, “Animating rotation with quaternion curves,” *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 245–254, Jul. 1985, doi: https://doi.org/10.1145/325165.325242.
[5] R. Parent and P. Firm, *Computer animation : algorithms and techniques*. Waltham, Mass.: Morgan Kaufmann, 2012.
[6] D. H. Eberly, *3D game engine design : a practical approach to real-time computer graphics*. Boca Raton: Crc Press, 2015.
[7] B. K. P. Horn, “Closed-form solution of absolute orientation using unit quaternions,” *Journal of the Optical Society of America A*, vol. 4, no. 4, p. 629, Apr. 1987, doi: https://doi.org/10.1364/josaa.4.000629.
[8] Gene Howard Golub and C. F. Van, *Matrix computations*. Baltimore: The Johns Hopkins University Press, Cop, 2013.
[9] W. Hamilton, “LXXVIII. *On quaternions; or on a new system of imaginaries in Algebra,*” *The London, Edinburgh, And Dublin Philosophical Magazine And Journal Of Science*, vol. 25, no. 169, pp. 489–495, Jan. 1844, doi: https://doi.org/10.1080/14786444408645047.
[10] A. Menache, *Understanding motion capture for computer animation and video games*. San Diego, Ca: Morgan Kaufmann, 2000.
[11] J. Mechner, *The Making of Prince of Persia*. CreateSpace Independent Publishing Platform, 2011.
[12] K. Perlin, “An image synthesizer,” *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 287–296, Jul. 1985, doi: https://doi.org/10.1145/325165.325247.
[13] L. Kovar, M. Gleicher, and F. Pighin, “Motion graphs,” *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 473–482, Jul. 2002, doi: https://doi.org/10.1145/566654.566605.
[14] B. Mandelbrot, “How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension,” *Science*, vol. 156, no. 3775, pp. 636–638, May 1967, doi: https://doi.org/10.1126/science.156.3775.636.
[15] M. Pražák, Ludovic Hoyet, and C. O’Sullivan, “Perceptual evaluation of footskate cleanup,” *Symposium on Computer Animation*, Aug. 2011, doi: https://doi.org/10.1145/2019406.2019444.
[16] N. K. Govindaraju, B. Lloyd, S.-E. Yoon, A. Sud, and D. Manocha, “Interactive shadow generation in complex environments,” *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 501–510, Jul. 2003, doi: https://doi.org/10.1145/882262.882299.

[17] J. Lee, J. Chai, P. S. A. Reitsma, J. K. Hodgins, and N. S. Pollard, “Interactive control of avatars animated with human motion data,” *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, Jul. 2002, doi: https://doi.org/10.1145/566570.566607.
[18] A. Safonova, J. K. Hodgins, and N. S. Pollard, “Synthesizing physically realistic human motion in low-dimensional, behavior-specific spaces,” *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 514–521, Aug. 2004, doi: https://doi.org/10.1145/1015706.1015754.
[19] Hyun Song Shin and Hyun Ju Oh, “Fat graphs: constructing an interactive character with continuous controls,” *Symposium on Computer Animation*, pp. 291–298, Sep. 2006, doi: https://doi.org/10.5555/1218064.1218104.
[20] TreuilleAdrien, LeeYongjoon, and PopovićZoran, “Near-optimal character animation with continuous control,” *ACM Transactions on Graphics*, vol. 26, no. 3, pp. 7–7, Jul. 2007, doi: https://doi.org/10.1145/1276377.1276386.
[21] L. Zhao and A. Safonova, “Achieving good connectivity in motion graphs,” Graphical Models, vol. 71, no. 4, pp. 139–152, Jul. 2009, doi: https://doi.org/10.1016/j.gmod.2009.04.001.

Appendix



Appendix A.1 Left Half of the Full UML Class Diagram



Appendix A.2 Right Half of the Full UML Class Diagram