ANIMATION OF 3D AVATARS

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2019

Student id: 10328541

Department of Computer Science

Contents

Word Count: 14876

List of Figures

Abstract

Creating realistic animations involving human motion is a complex task. Motion capture technology can be used to record motion from the real world, but generally, the results cannot be reused in different settings, which makes its adoption very expensive. This project presents an implementation of the motion graph technique. Motion graph is a graph-like data structure that can be used to synthesise motion from a motion capture database. Motion is extracted as a graph path, where edges represent transitions between two animations, and vertices are original motion capture clips. A simple linear interpolation is used to transition from one motion to another, which means that the motions must be similar. The purpose of the motion graph is to find optimal transition points between multiple animations so that interpolation can be used to produce natural-looking motion. Two graph traversal algorithms are proposed to extract motion. The first is a greedy algorithm that creates random graph walks, while the second looks for graph paths that can connect a list of vertices in a specific order.

Experiments were conducted to determine the correctness of the motion blending method and to evaluate the ability of the graph traversal algorithms to produce valid animations. The experimental results show that the implemented system can generate realistic-looking animations from the motion capture data.

Declaration

No portion of the work referred to in this dissertation 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

I would like to thank my family and friends for supporting me through this journey. Thank you from the bottom of my heart.

Chapter 1

Introduction

The main subject area of this project is 3D computer animation of digital human characters. The rapid evolution of computer graphics has made it possible to render photorealistic images of humans. However, progress was much slower in animation, due to the many subtleties of the human motion. Because people interact with one another every day, when a person moves in a slightly odd way, we can immediately notice that something is not right. This feeling perfectly translates to computer-generated animations, especially for everyday actions such as walking or running. Therefore, realistic motion is crucial to the overall believability of the presentation.

Traditionally, in 2D animation, keyframing is one of the most popular techniques for creating animation. In keyframing, the animator draws the primary poses (or frame) of an object, while other workers draw the missing frames (in between of keyframes). The same technique and principle apply to 3D character animation [Las87], but in this case, the computer generates the missing frames using interpolation algorithms. Because of the increased dimensionality from 2D to 3D, keyframing becomes a very time-consuming process. It is possible that depending on the complexity, "weeks of work are needed for a few seconds of animation" [HT00]. Therefore, it is common to use motion capture to speed up the animation and, at the same time, obtain more realistic results by directly capturing a real person's performance.

However, motion capture is expensive due to the hardware requirements and infrastructure, which makes it unfeasible to solely rely on it. Hence, researchers have tried to reuse the motion capture data to syntheses new motion that was not part of the original data. Motion graph [Luc02] is the technique implemented in this project. It is based on the homonymous data structure that, once constructed, can be used to generate new motion sequences. Although, methods that simulate the physical properties of

the character can adapt to changes in the environment, the variety of the synthesised motion is still strictly dependent on the amount and diversity of the motion capture data. This problem is shared among all motion synthesis methods.

Research in motion synthesis is exciting because it is still fundamentally an "unsolved problem" [GSC+15], meaning that current solutions work well only when restrictions are applied. There are many applications of motion synthesis: path synthesis can be used in video games to control the character interactively. Crowd simulation can be animated automatically by blending an arbitrary sequence of motion. Physics simulated characters can be trained to perform specific movement and have the knowledge transferred to a real robot [PAZA17].

1.1 Aims

This work aims to implement a motion synthesis technique that use motion capture data to generate new motion sequences.

1.2 Objectives

The following objectives have been identified in order to achieve the project aim:

- Use OpenGL to create a 3D scene where the animation can take place.
- Create a parser for the motion capture data.
- Implement a generic skeleton system, that can be used for any skeleton type.
- Develop a motion playback system, that play both motion capture data as well as generated motion.
- Implement the motion graph data structure.
- Develop search algorithms to extract motion from the motion graph.
- Test and evaluate the system in different scenarios.

1.3 Chapter structure

This dissertation is structured in the six chapters.

- Chapter 1 introduces the subject area, as well as the aims and objectives of the project.
- Chapter 2 covers the background and literature review. This includes a description of motion capture and motion synthesis techniques, followed by a short math section.
- Chapter 3 describes the research methods, which includes the tools used for the implementation, the design of the software, and the description of the experiments.
- Chapter 4 covers the implementation process, describing each step in detail.
- Chapter 5 shows the results of the experiments and provides an overall evaluation of this work.
- Chapter 6 contains the conclusions of the dissertation. More specifically: summary of the results, reflection, and suggestions about future work.

Chapter 2

Background

The purpose of this chapter is to present the research area of this project, with an overview of the different methods that can be used to capture and synthesise motion. At the end of the chapter, a brief section on some fundamental mathematical concepts is included, since they will be relevant in the following chapters.

2.1 Motion Capture

The process of recording and digitising motion into a format that can be read and Interpreted by computer software is called motion capture. Over the decades, this technology has become increasingly important, so much so, that it is expected in almost every entertainment product (films, video games, advertising, etc.) that has to deal with animating a digital creature [Men11]. Given that motion capture is at the heart of this project and most of the motion synthesis techniques, it is appropriate to give an introduction about its history, techniques, and applications.

2.1.1 History

The origins of motion capture are directly related to that of computer graphics. In fact, only at the start of the 1980s, computers were powerful enough to render 3D scenes with textures. With the advent of 3D animations, a need for realistic human animations emerged. Although experimental results were first obtained in research settings as early as the late 1970s [Men11], the first commercial use of motion capture happened in the 1984 commercial "Brilliance", directed by Robert Abel. The animation was a 30 seconds long shot of a fully digital woman made of chrome advertising canned food.

Figure 2.1: PDI exoskeleton suit for the movie Toys. Photo taken from [Men11].

Markers were drawn on the actress body with a marker pen, then she was photographed from different point of views. The idea was to calculate the difference between the markers at incremental frames, and the combination of the various angles was used to localise each marker so that it could be used in an 18 joints skeleton model. The localisation was, at that time, a very time-consuming task since it involved comparing each pair of marker at every point of view.

One of the pioneers in motion capture technology was (now defunct) Pacific Data Images (PDI). In 1988, PDI built a wearable exoskeleton (figure 2.1) suit which had an array of optical potentiometers sensors placed on the joints position. The exoskeleton was used for the *Waldo* Muppet character on the *The Jim Henson Hour* show, and later in the movie *Toys*, in 1992.

During the 1990s, optical motion capture became the most common technique in the entertainment industry [PEG$^+$09]. Companies such as Biovision offered motion capture services to use their proprietary optical system and software. These systems were based on special markers, tracked by multiple high-speed infrared cameras. The video game industry is one of the biggest early adopters of motion capture, so much so that "today it is a given that any video game that has human characters is expected to have used motion capture" [Men11].

Motion capture developed in the same period (1980-2000) as computer vision. Hence, researchers have tried to apply vision-based techniques to solve motion capture. Earlier methods were, mostly, based on optical flow which uses the velocity of each region of the human body over multiple frames. The velocities converge to a

global average that is then grouped into blobs, from which it is possible to estimate the location of the joints. Although assumptions, like motion pattern (e.g. walking) and small changes between frames, were necessary in order to find a valid solution [JC99]. Other computer vision-based techniques include probabilistic methods, and more recently, deep learning methods.

2.1.2 Motion capture types

There many technologies that can be used to capture motion. However, all of them can be classified, based on the position of the source and sensors, in the categories below.

- Inside-in: no external sources are needed since the sensors can work individually and are placed directly on the joints. Typically, accelerometers and gyroscopes are used.
- Inside-out: external source emits signals that the sensors placed on the body can detect and use to locate itself. These systems follow the same principle as the GPS system, in which sensors get their position using the broadcast signal of external sources (satellites).
- Outside-In: similar to inside-out system. However, the tracking is done externally, since the markers on the body do not collect any information. Optical system using infrared cameras are the most popular example in this category.

Optical methods

Optical motion capture uses an array of calibrated cameras to track a set of reflective markers. Each camera emits an infrared (or visible red) light source, which is then reflected by spherical markers that can be from a few millimetres in size up to a few inches, depending on the situation. The resolution and the frame rate of the camera determine the quality of the capture. Resolution range from 180x180 pixels to 16 megapixels, while capture rate from 30 frame per second (fps) up to 2000 [Men11]. Typically, a higher resolution results in a lower frame rate.

Optical mocap can be very accurate, but they are also sensitive to changes. If a camera is moved during the recording process, the result becomes skewed and unusable. That is why recalibration is always necessary before shooting a scene, which uses an object with known dimensions. The minimum number of cameras is two, although a bigger capture area requires more, even up to a few hundred. The purpose of

Figure 2.2: Motion capture studio with optical cameras setup. Image from [Rok19].

using multiple cameras is to reduce occlusion of the markers, and if the cameras lose line of sight for more a few frames, an operator must manually reassign it. Due to the high sensitivity to light, optical systems can only be used indoors in uniform lighting conditions and away from lighting sources of the same wavelength [Vic16]. Figure 2.2 shows an optical motion capture system using visible red light as a source, and external lights were purposely dim to avoid interference with the source lights.

Once the recording is done, data must be processed to be usable. Depending on the number of cameras, the amount of data collected can be significant. The post-processing involves three steps: first is the image segmentation of the footage, leaving only the markers visible, which can be achieved using a threshold value based on the intensities of the markers, as well as, information from neighbour frames. The second step is to compute the local 2D coordinates for every camera, which are then combined together in step three, which uses all the different point of views to produce the final 3D coordinates for each marker.

Electromechanical suits

At the opposite side of the spectrum, compared to optical mocap, are the electromechanical suits. These devices are based on inertial sensors packed together in MEMS (Micro Electro-Mechanical Systems), which include accelerometer, gyroscope, and potentiometers. The basic idea is to place several MEMS across the body (at joints location) and measure their angles. This has the advantage, being an inside-in system, to

Figure 2.3: Xsens full body motion capture suit. Image from [Xse19].

be completely free from occlusion, and also gives more flexibility since it can be used anywhere, independent of lighting and terrain condition. However, trade-offs include: inability to track root position since only angles are measured; lower maximum refresh rates. An example of this type of system is the Xsens MVN link suit shown in figure 2.3.

Local positioning systems

A local positioning system (LPS) is a motion capture solution that can be considered as a "miniaturised GPS". Although this type of mocap system is not fully mature yet, it is one of the most promising technology to be the next state-of-art [Men11]. LPSs are inside-out systems that use an external emitter (radio frequencies, laser, or other) and an array of sensors that can detect the source. By counting the time between each detection and knowing either the position of the emitters or that of the neighbour sensors (relative to each other), it is possible to accurately calculate of the sensor's position in space using trigonometry.

Radio frequency-based systems can operate in large areas like football field (up to 25000 m^3) with an accuracy of 1-2 mm [Men11] at 250 updates per second. Laser-based systems have recently been commercialised with the release of modern virtual reality headsets. The HTC Vive headset (developed by Valve) introduced an LPS laser-based tracking system with an accuracy of 1.5 mm, a sampling rate of 1000 Hz, and

up to 100 m^2 area (in the second generation model) [Kre16].

Vision-based methods

Vision-based motion capture uses computer vision techniques to track and estimate a human pose. Vision-based methods are similar to optical mocap, but with a significant difference: vision does not rely on markers for tracking, instead only raw pixel data (sometimes in addition to depth data) is used. The most appealing promise of this type of mocap is the ability to use conventional cameras in any kind of environment, without hardware or space restrictions. A vision-based system for motion capture can be built following the structure below [MG01].

- Initialisation: set initial parameters of the camera, scene, and model (joints). This process can be simplified if assumptions are made, for example, we can assume there is no camera motion, no occlusion, a predefined motion pattern etc.
- Tracking: this step concerns the identification of the human figure, or rather, the segmentation of the object of interest (humans) from the background. Generally, tracking is a solved problem in constrained scenarios, where the background is relatively easy to discriminate [GJH01], but results are still not optimal in real-life scenes, where conditions are unknown [ZN03]. Tracking is greatly simplified if the camera is equipped with a depth sensor, an example of such a system is the RGB-D front-facing camera of iPhone X [LCM+19], and the Kinect [Zha12].
- Pose estimation: involves finding the skeleton parameters, which are the joints orientation. This process can operate frame-by-frame or use temporal information from surrounding frames.

Although vision-based motion capture promises fewer drawbacks compared to its marker-based counterparts, it is, however "not yet clear exactly what accuracy can be achieved and whether such systems can be effectively and routinely utilised" [CECS18].

2.1.3 Applications

Motion capture is used across a wide range of industries. These include entertainment, defence, medicine, law, and sports.

Entertainment

Video games is, most certainly, the market that saw the highest adoption of motion capture technology. Almost every game with human character is expected to use mocap to make faster and more realistic animations. Even during the early days, when motion capture was still in its infancy and expensive, game developers were among the first to use this tool, while the more mature television and film markets were still sceptical. This can be explained by the fact that video games were still primitive and not able to handle high-quality motion capture, especially with the hardware constraints of the time, leading to their symbiotic development.

Defence and security

The military and police use motion capture in training simulations in conjunction with virtual reality headsets. These simulations often do not include full-body motion capture, but only the weapon and a small number of tracking points on the body. Fighter pilots have, for a long time, been using a tracking system for their helmet to augment aim and operational awareness [Men11].

Medicine

In the medical field, motion capture is known as *3D motion analysis*, and it is primarily used in orthopaedics to study how joint movement interact with bones, and the locomotion mechanism of people and animals (gait analysis). Perhaps surprising, medicine has always had the biggest user-base of motion capture [Men11].

Law

Reconstruction animations using motion capture can be presented in court to the jury as evidence. This is a way to supporting the narration of the events from the point of view of the witnesses. However, the animation must conform to a strict standard in order to admissible; for example, it has to be simple enough that it cannot be used to easily influence the jury [Sch07].

Sports

Training of elite athletes is constantly monitored to ensure correct performance and to help improve their techniques. Among all the data, motion is undoubtedly one of the

primal metrics, especially in sports like swimming, gymnastics, and running.

2.2 Motion synthesis

Before introducing the various techniques for motion synthesis, we need to define what is meant by motion synthesis. Motion synthesis is the process of, automatically, creating new animation that satisfies a set of desired constraints. Early work involved editing individual animations, with the goal of adapting them to different scenarios and environments. Motion warping [WP95, BW95] gives good result only for small changes in the overall motion, and since it uses an approach similar to image warping [BN92], extreme warps are distorted. In this project, we are interested in the synthesis of entire clips of motion.

It is possible to loosely categorise the methods in the literature, in two main approaches: "data-driven and physics-based methods" [GSC+15]. The former category relies entirely on the data and typically requires a significant number of high-quality motion capture data [LL06]. The latter methods are based on physics simulation and are generally more complex since characters have to interact with the virtual world following real laws of motion. As an analogy, we can think of the former approach as puppeteering, while the latter approach is closer to that of robotics.

2.2.1 Data-driven methods

Data-driven methods use motion capture data to construct new sequences of motion by splitting the original clips into many smaller segments. Transitions are created between these segments using interpolation algorithms, which effectively makes it possible to jump between different motions in a relatively smooth and convincing manner (depending on the similarity of the motions).

One of the first paper on data-driven methods was introduced by A. Lamouret and M. van de Panne in 1996 [LvdP96]. Although their model only had 5 degrees of freedom (DOF), by comparison in this project, we have a skeleton with 62 DOF; the underlining process remained unchanged. Figure 2.4 shows the general structure of a data-driven method: the original motion clips are examined to produce a data structure that cuts each motion into chunks, then new motion is generated by combining these pieces. The success of data-driven methods in commercial applications (e.g. in video games), is due to their low computational cost once the data structure is generated, an

example of real-time interactivity are [LCR+02] and [AF02]. Currently, data-driven methods can be classified as graph-based or statistical, however these labels are not mutually exclusive, since there is no formal boundary.

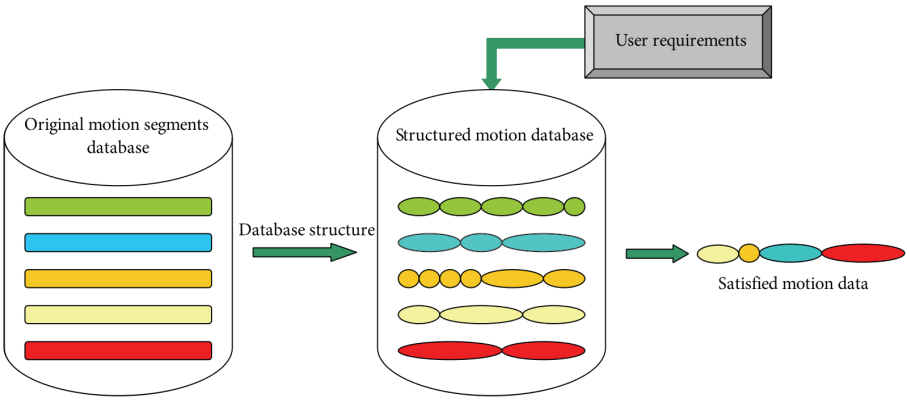


Figure 2.4: Data-driven motion synthesis process, credits to [WCW14].

2.2.2 Graph-based methods

Graph-based methods take advantage of graph-like data structures in order to navigate between the motion clips. As previously mentioned, the original clips are split into smaller segments, each represented as nodes, while transitions between them are edges. Motion is extracted using a search algorithm on the graph. As a result, graph-based methods are also referred to as *planning and search methods* [LK05]. Work following this approach include [GJSS01, AF02, Luc02, LK05, HG07, MC12].

Motion graph

Motion graph [Luc02] is arguably the most popular technique in this category since the original paper is one of most frequently cited with, at the time of writing, over 1600 citations. The main contribution of this work is the automatic construction of the graph. The authors introduced a distance metric that makes it possible to compare arbitrary motions in order to locate suitable transition points between them. This metric determines how similar two motions are, which is achieved by incorporating derivative information using a point cloud representation of the character's movement. The point cloud is formed over a window of frames, each of which consists of a smaller point cloud obtained from downsampling of the character's mesh. In this way, derivative

information is included in the calculation. However, the point cloud needs to be normalised because poses remain the same no matter the direction the character is facing or its position on the floor.

As shown in figure 2.6b, the motion graph is created as a directed graph where each edge represent a transition, and each node is a segment of the original motion and also serves as the connecting point to other segments. Transitions are generated using linear interpolation, which is "sufficiently reliable if two motions are close" [Luc02]. Once the graph is constructed, motion can be extracted by searching the graph. Additional constraints can be applied to the search. For example, the paper describes a path synthesis process that searches the graph for motions that satisfy the path constraint through optimisation. Figure 2.5 shows a synthesised motion along a path, obtained by searching the motion graph.

Drawbacks of motion graphs include: scalability for the construction of the graph in large motion databases, this is mainly derived from the computation of the distances which involves every fair of frames; a threshold must be manually set by the user, this value regulates the quality of the transitions, meaning that setting a lower value will select better transitions, however, it is also possible that the graph will not have any edge that meets the threshold.

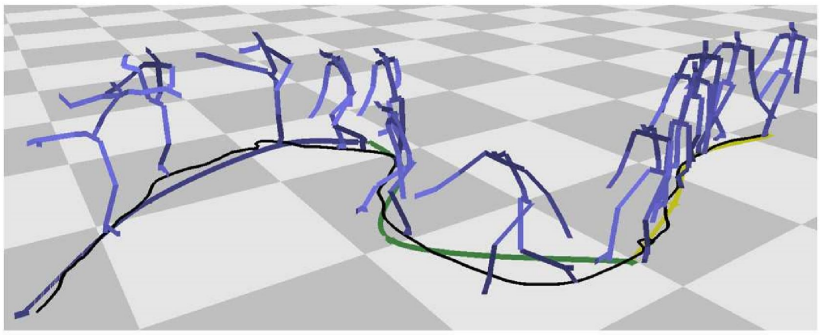
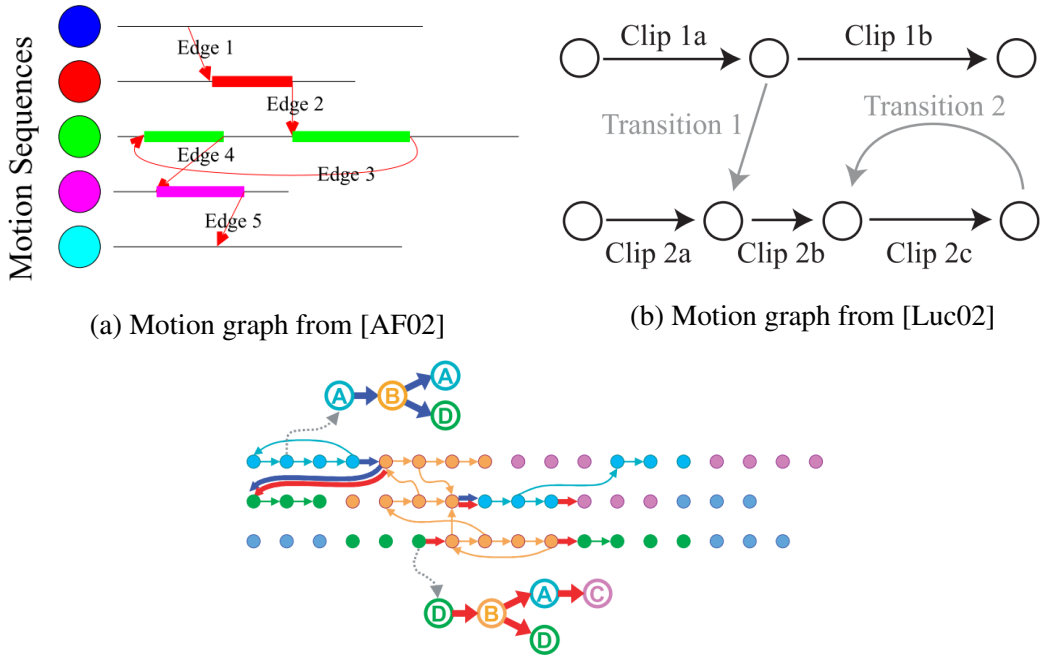


Figure 2.5: Example of a synthesised sequence using motion graphs (image from [Luc02]).

Motion graph variants

Other researchers have proposed different graph construction and search techniques. In [AF02], nodes represent the original motion and edges only serve to connect two frames. The transition happens using a *localised smoothing* function on ± 30 frames around the transition point and makes the motion to be continuous. A randomised



(a) Motion graph from [AF02]

(b) Motion graph from [Luc02]

(c) Motion graph from [LCR+02]

Figure 2.6: Motion graph variants.

search algorithm is used instead of branch and bound (as in [Luc02]), which creates accessory graphs with different levels of details in order to limit the search complexity. Secondary graphs with lower detail are created by clustering the edges in the distance matrix (matrix with the distance of every pair of frames between two motions), and *k-means* is used to reduce half of the edges at each level, effectively obtaining "an instance of Tree-Structured Vector Quantization" [AF02].

A two-layer architecture is presented in [LCR+02] and [SBS02], where the lower layer is a Markov process that depicts transitions with probabilities; the higher layer is a statistical model that analyses clusters of similar frames. The clusters can be connected to each other, forming *cluster trees*, which in turn form the *cluster forest*. Motion is extracted by first finding a cluster path in the higher layer, then a local search on the lower layer (more detailed) selects the motion with the highest probability. The clusters are depicted in figure 2.6c with different colours.

Figure 2.6 shows a comparison between different construction techniques used to build motion graphs. These graph structures are proposed in [AF02, Luc02, LCR+02].

Finite state machine

An alternative to graph-like data structure is presented in [LK05], where a finite state machine (FSM) is used instead. In addition, this method also groups clips of motions in behaviours, which are then represented in the FSM as behaviour states, then the $A*$ algorithm is used to search the resulting tree of the states. Compared to motion graphs, FSM has the advantage of being much more efficient because of their smaller number of states and can guarantee completeness which means that it will always find a solution if it exists. Making this method appealing for game engines [Gre13]. However, FSM requires the segmentation of motion clips into sets of behaviours, and since motion can be ambiguous, it is a time-consuming task. Methods for automatic segmentation were presented in [JAJ+04]. More recent efforts have been made to automate the process of defining the relationship between the states using machine learning techniques [Gil09].

2.2.3 Statistical models

"Statistical models have been frequently used to capture the underlying structure of a large collection of motion data" [LL06]. *Hidden Markov Models (HMM)* can be used for learning complex behaviours with temporal dependencies [GJH01]. This means that it is possible to build at least some kind of understanding of the motion, but since the human motion is intrinsically high-dimensional, dimensionality reduction techniques have been introduced.

Principal component analysis (PCA) is used to project the data into a lower dimension representation, while preserving the structure of the data. Then, the lower-dimensional data can be clustered, for example, using *K-means*. Finally, a *Markov chain* is used to construct a transition graph, which can be used to generate motion.

Tanco and Hilton [HT00] use two layers: the lower level is a Markov chain, used to produce a coarse sketched motion; the higher level is an HMM to generate the final motion. Real-time interaction have been achieved by constructing more efficient *cluster forest* in [SBS02] and [LCR+02], as previously discussed.

Development in manifold learning algorithms has inspired researchers to adopt such techniques to enhance results in motion synthesis. Manifold algorithms are used when the input data is non-linear in the overall structure, but is locally linear, in a data point's neighbourhood. Kovar and Gleicher adopted a manifold strategy to find "logically similar motions that are numerically dissimilar by using close motion as

intermediaries to find more distant motion" [Kov04].

An extension of *ISOMAP* (unsupervised algorithm used to extract non-linear embeddings [TDSL00]) named *ST-ISOMAP*, where *ST* stands for *Spatial-Temporal*, was developed to "extract motion primitives" [JM02]. Subsequently, the *LLE* [RS00] (Locally Linear Embedding) algorithm was used to generate new sequences of motion by extracting a path in the LLE embedding space [JFLM07]. Another manifold technique called *Gaussian process dynamical models* (GPDM) is a latent variable model that can encode high-dimensional motion data to a latent space, while preserving temporal information [WFH08].

With the recent success of deep learning methods in computer vision tasks, researchers began experimenting with these ideas in the field of motion synthesis. One of the first exciting work in this direction is [HKJ17], where the authors presented a novel neural network architecture named *Phase-Functioned Neural Network (PFNN)*, which uses cyclic motions as training data. The PFNN takes as input the environment's mesh, a phase value, and the last state of the characters to produce a realistic motion that adapts to both displacements in the environment, as well as input control from the user. With this technique, new motion can be computed in real-time once the model is trained. However, being a deep learning method, it requires a large amount of motion data, which needs to be pre-processed to be cyclic.

2.2.4 Physics-based methods

Physics simulation can be used to generate motion. This approach is radically different from data-driven methods, drawing inspiration from robotics, biomechanics, and control theory, the character constantly interact with the environment and motion are the result of forces and torque applied to the joints. Although physics-based techniques have been around for many decades ([AG85, Wil87]), their popularity outside of the research space still lacks due to some critical drawbacks.

Traditionally, physics-based methods have greatly suffered from stylistic limitation of movement [NF02], since motion is dictated by multiple motion controllers, which have to work together to produce composite motion, it can appear robotic and look unnatural. This also leads to a less reactive response from interactive control inputs from the user. Simulating a complex system such as the human body articulation is a very difficult task, and requires a reasonable amount of knowledge about "multi-body dynamics, numerical integration, biomechanics and optimisation theory" [GP12], and it is also computationally expensive, meaning that compromises must be made on

either the number of simulated parts or number of characters.

There are three main components in a character physics simulation: *physics engine, physics-based character, motion controllers*.

Physics engine

The physics engine is responsible for continuously updating the simulation by a small timestep at every iteration, effectively applying all the forces to compute the new position. The tasks of the physics engine are collision detection, dynamics simulation, and numerical integration.

Collision detection checks if objects intersect with each other, which is important because interaction only happens after a collision has been detected. However, collision between different parts of the character (self-collision) is disregarded for simplicity [GP12].

Dynamics is the process of calculating the angular accelerations of the objects in the simulation. The *Newton-Euler* equation describe how to compute the linear velocity and angular velocity of a rigid-body (non-elastic). The simulated body is logically held together by a number of joints, but to enforce this in the system, one must either apply additional forces to the limbs in opposite directions or limit the DOF (degree of freedom) [GP12], such that the movement is always within the constraints.

Numerical integration is the task of applying the accelerations from the dynamics simulation step to update the objects' velocities and positions in space. Integration requires a timestep value, which is used to specify how much the simulation is advancing. A higher value results in a less accurate simulation (potentially leading to instability), while a smaller value makes the simulation more accurate, at the cost of performance.

Physics-based character

Traditional characters are based on a skeleton system, which has geometric meshes attached to it. Physics-based characters "require mass properties and joint constraints, as well as actuators to enable motion control" [GP12]. Meaning that characters have, apart from the visible mesh, a collision geometry with mass information made of simple primitives (for performance) that closely resembles the original mesh.

The motion is generated by actuators, which located in the joint positions. The actuators are the digital equivalent of *servomotors* in a physical robot, and serve to

apply torque on the joints. An alternative method, taking direct inspiration from biology, is a system based on muscle contraction [WGF07]. However, due to its additional complexities, physics character based on muscle contraction still remain unpopular [GVDPVDS13].

Motion controllers

To make the character perform certain actions (e.g. running, jumping etc.), an appropriate motion controller must be designed for each action. Naturally, there are many possible approaches that one can follow to implement a controller.

In early works, controllers were mostly handcrafted and based on traditional control theory, and were not able to perform many skills as well as being sensitive to perturbations [FvdPT01, YLvdP07], which has been partially solved by learning a nonlinear probabilistic model from perturbed example sequences [YL10].

Neural networks can be used to learn a mapping between the sensors information (the state of the character) and the actuators. This means that a control policy is learned automatically by the network, but we still need to supply a objective function which specifies what motion to learn.

More recently, researchers have achieved very impressive results using deep reinforcement learning (RL) algorithm. DeepLoco [PBYV17], learned to walk in dynamic environments, while accomplishing specific goals and demonstrated robustness from external disturbances. Making RL trained controllers potentially suitable to be used in real robots [PAZA17].

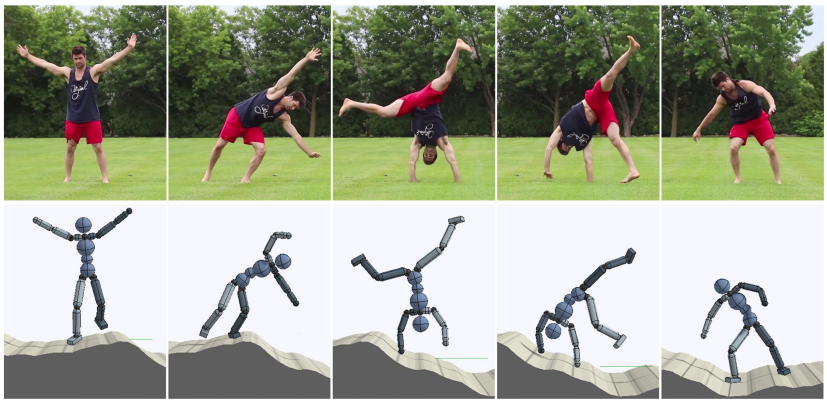


Figure 2.7: Example of learning motion directly from video (image from [PKM$^+$18]).

2.2.5 Hybrid methods

Hybrid methods use both motion data and physics simulation to produce motion. The idea is to have the advantages of both approaches: achieve a more natural-looking motion using motion capture data and a responsive character to environmental perturbations.

There are two main ways of combining physics system and motion data: the most obvious one is to simply have both at the same time, meaning that when there is no perturbation we can play the reference motion, and switch to the physics-based character when a reaction is needed [ZMCF05]. Another way is to integrate a statistical model [GSC+15] that maps the reference motion capture to a *force field*. Such system has been implemented in [WMC11] using a non-linear Gaussian process. The results of the latter method show "realistic adaptations to perturbations that are not present in the training data, characters with varying properties and terrains" [WMC11].

Deep reinforcement learning is employed in Deepmimic [PALvdP18], which improved the results by imitating motion capture clips, making the controllers able to perform a large repertoire of motions skills. This idea was quickly extended by using video as training data (figure 2.7). Similar results were also achieved by other researchers in [LPY16, MTT+17]. While the promises of RL methods are fascinating, limitations include high computational cost of training and the design of specific behavioural policies.

2.3 Mathematical concepts

This section serves as a quick reminder to the essential mathematical concepts of computer animation and computer graphics, which have been used in the implementation of the project.

2.3.1 Transformation Matrices

To successfully animate a skeleton we need to be able to translate, rotate, and scale every bone appropriately. In computer graphics every transformation in 3D space can be represented by a $4x4$ matrix with homogeneous coordinates [dV15]. The advantage of using matrices is that, it is possible to combine different transformations by simply multiplying their respective matrices.

Translation

A translation of a point by a distance $d = (d_x, d_y, d_z)$ is a simple sum of its coordinates by the components of d. In a matrix form, we have the following:

$$T(d_x, d_y, d_z) = \begin{Bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{Bmatrix} \tag{2.1}$$

Scaling

Like translation, scaling is also a quite straightforward operation:

$$S(s_x, s_y, s_z) = \begin{Bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix} \tag{2.2}$$

Rotation

In the case of the rotation, the matrix transformation is more complex because it is composed of three distinct rotations, one for each axis. This means that, in order to describe a generic rotation in 3D space, we need three separate matrices: R_x, R_y, and R_z.

$$R_x(\theta) = \begin{Bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix} \tag{2.3}$$

$$R_y(\theta) = \begin{Bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix} \tag{2.4}$$

$$R_y(\theta) = \begin{Bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix} \tag{2.5}$$

Where θ is the Euler angle of the axis of rotation. Using Euler angles, although intuitive, presents a significant disadvantage known as the "gimbal lock". This is caused by the alignment of two rotational axes to become parallel, and hence we lose a degree of freedom.

2.3.2 Quaternion

Another way of representing 3D rotations is by using quaternions instead of Euler angles. Quaternios are an extension of complex numbers, and are especially useful for animation tasks involving rotations on an arbitrary axis. A quaterion is formed by two components:

$$q = (s, v) \tag{2.6}$$

where s is a real scalar, and v a vector that has imaginary components.

$$v = ia + jb + kc \tag{2.7}$$

where (a, b, c) are components of the vector v, and (i, j, k) have the following properties:

$$i^2 + j^2 + k^2 = -1 \tag{2.8}$$

$$ij = -ji = k \tag{2.9}$$

To rotate an point by an angle θ around the axis u (unit vector), we have to define the quaternion:

$$q = \left(cos\left(\frac{\theta}{2}\right), usin\left(\frac{\theta}{2}\right) \right) \tag{2.10}$$

and its inverse:

$$q^{-1} = \left(cos\left(\frac{\theta}{2}\right), -usin\left(\frac{\theta}{2}\right) \right) \tag{2.11}$$

The rotated point p' can then be calculated as:

$$p' = qrq^{-1} \tag{2.12}$$

where $r = (0, p)$ is a quaternion with its real part set to zero and its vector part is equal to the original point p.

There a few advantages of using quaternions over Euler angles:

- quaternions do not suffer from the gimbal lock problem;
- no longer requires applying a different rotation for each axis;
- can be combined with simple multiplication;
- interpolation yields better results [Sho85] (useful for blending motions);
- can be easily converted to a rotation matrix.

Chapter 3

Research methods

The purpose of this chapter is to provide, to the reader, a description of the tools and an overview of the implemented software's architecture. Firstly a description of the encoding of the motion capture data is given. Secondly is a summary of the tools, including programming language, OpenGL, and external libraries. Finally, an overview of the software's design is presented, as well as the experiments which will be the focus in the following chapters.

3.1 Motion capture data

This project uses the motion capture database of "Carnegie Mellon University graphics lab" [Lab02]. The database is available for download to anyone who wishes to use it. The data is organised by subject or the type of action performed. A subject is a particular configuration of the skeleton based on the body of the actor involved in the motion capture process. Two types of files are needed to play an animation: a *.asf* and a *.amc* file. These are text-based formats designed to be readable by humans. This encoding is based on the Acclaim motion capture format [Lan98], developed by the homonymous game company.

3.1.1 Skeleton structure

The skeleton structure is encoded in the *.asf* file as plain text. It consists of a root node, which is the only one that has position value, and a collection of bones with different lengths. Keywords are used to divide information. For example, the keyword **":root"** signals the beginning of the root properties, while lines following **":bonedata"** are

individual bone information.

Because bones properties are relative and do not have an absolute position, the skeleton needs to have a hierarchy system in order to move the bones to the correct position. The keyword **":hierarchy"** specifies the part of the file where relationship information is listed. Bones are related through parent-child references, which creates a tree structure (figure 3.1). This means that a bone in a lower hierarchical position inherits all the transformations of its parent.

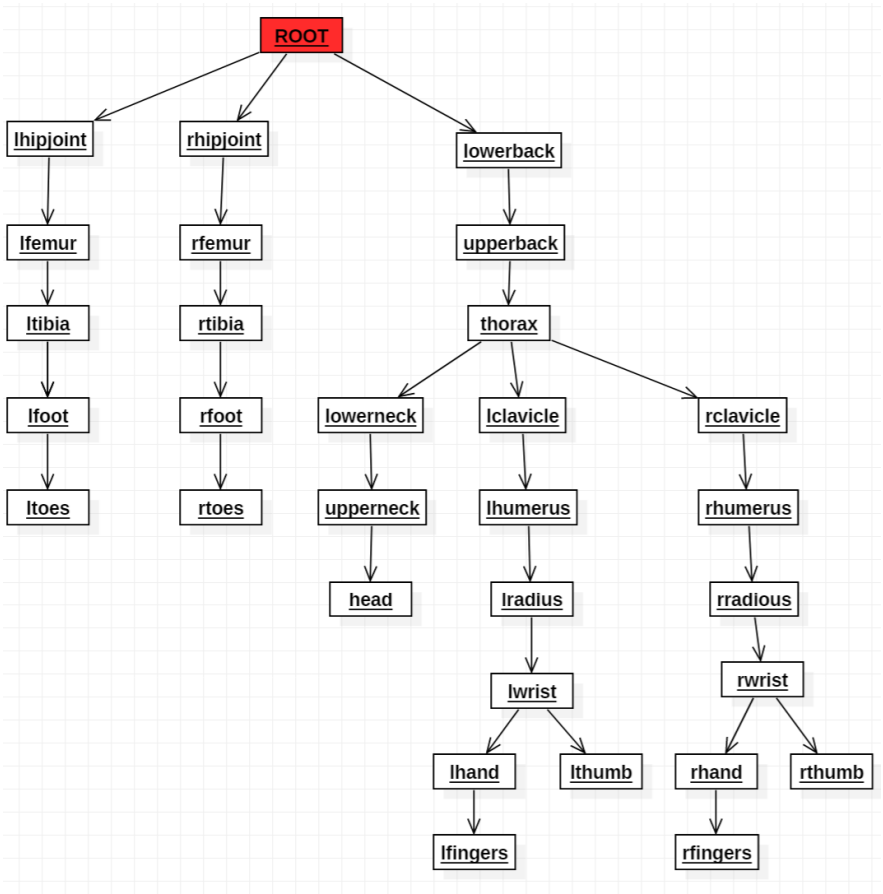


Figure 3.1: The skeleton as tree structure of bones.

3.1.2 Motion data

The *.amc* file contains the motion data, which is represented on a frame-by-frame basis. Each frame corresponds to a single pose. A pose has the global position of the root node and rotation values for each bone, meaning that it has full control over the skeleton's position and posture.

A single *.amc* file can contain an arbitrary number of frames, which are separated by small time interval that depends on the sampling rate of the motion capture system. Motion in the CMU database is all sampled with a frequency of 120 Hz.

3.2 OpenGL development environment

This section describes the tools used in this project. In particular, it provides an overview of OpenGL and all the related libraries.

3.2.1 C++ programming language

The programming language chosen for this project is C++. C++ is designed to fully support the object-oriented paradigm while retaining the compatibility with the C programming language. This combination makes it very versatile since efficiency is not sacrificed for the higher-level features, like other similar languages (e.g. Java). Although OpenGL can be used with more modern programming languages (e.g. Python, Ruby, Java etc.), the ability of C++ to operate closer to the hardware still makes it the preferred language for game engines and graphics-based applications [Gre13].

3.2.2 OpenGL

OpenGL is a graphics API specification, currently developed by the Khronos Group consortium. Since it is only a specification, the task of implementing the actual functions of the interface is up to the graphics chip manufactures, and are specific to that hardware. However, as a loyalty free API, OpenGL can be implemented and distributed by anyone who wishes to develop it. For example, in the Linux operating system, there are versions developed by the open-source community.

3.2.3 Immediate mode and core profile

An important distinction must be made between immediate mode OpenGL (version 1.X and 2.X) and OpenGL core profile (versions above 3.2). The former is only supported in older versions, and it has a fixed graphics pipeline, which makes it easy to use but at the expense of flexibility. For example, the user can easily draw primitives (e.g. cubes, spheres, toruses) with a single command, and does not have to define the transformation matrices, but rather he or she has to use a built-in matrix stack. This is

not ideal for our purposes since skeletal animation requires a series of complex matrix operations as described in 4.2.

On the other hand, OpenGL core profile leaves most of the operations to the developer, all transformation matrices must be specified, as well as all the vertices of an object, as normalised coordinates. Other major improvements are the programmable shaders and vertex buffers. This project uses the OpenGL core profile (version 3.3).

3.2.4 OpenGL core profile overview

OpenGL could be described as a state machine, since function calls depend on the current state, meaning that the same function will behave differently if different values were set before its call. An example is the draw function that can draw either points, lines or triangles, depending on the set variable.

Model vertices are stored in an array called vertex buffer object (VBO) and then sent to the GPU in a single transfer to limit bus overhead between CPU and GPU. However, OpenGL also requires us to define a vertex array object (VAO) which acts as a wrapper of the VBO, with additional information on how to read the VBO (size and offset in memory).

Unlike the older version of OpenGL, in core profile, there are no default shaders, so it is the developer's responsibility to implement them. Shaders are generally small programs written in GLSL (a C-like language), they contain graphics operations for matrices and vectors. There are three types of shaders: vertex shaders, geometry shaders, and fragment shaders. Although, only the vertex shader and fragment shader are required by OpenGL. Vertex shader determines the position of the vertices and texture coordinates; fragment shaders calculate the final colour of each pixel, e.g. modify colour based on the lighting information.

3.2.5 Coordinate system

OpenGL employs several coordinate systems to define an object's final position on the screen. Figure 3.3 shows how these coordinates interact with each other, and the necessary operations that are required to move from one to another.

Local space

The local space is the original space of an object's vertices coordinates, that is the initial position when a object is created.

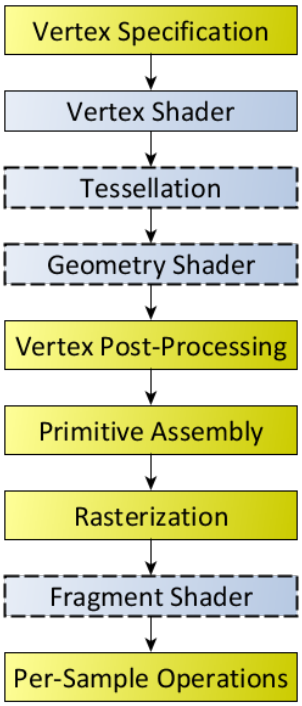


Figure 3.2: The OpenGL rendering pipeline. Image from [Khr19]

World space

World space is the coordinate system of all the objects in the scene. Each object coordinates in the world space is calculated multiplying the model matrix to the local coordinates. A model matrix corresponds to scaling, rotating, and translating an object and can be defined as:

$$M = S * R * T \tag{3.1}$$

where S, R, T are the scale, rotation, and translation matrices respectively.

View space

The view space coordinates are coordinates relative to the position and orientation of a camera.

Clip space

The clip space has normalised device coordinates (NDC) which are in the range of -1.0 to 1.0. Clip space is used to distinguish the visible coordinates by the camera and those that are not. A coordinate is visible only if it falls into the normalised range if

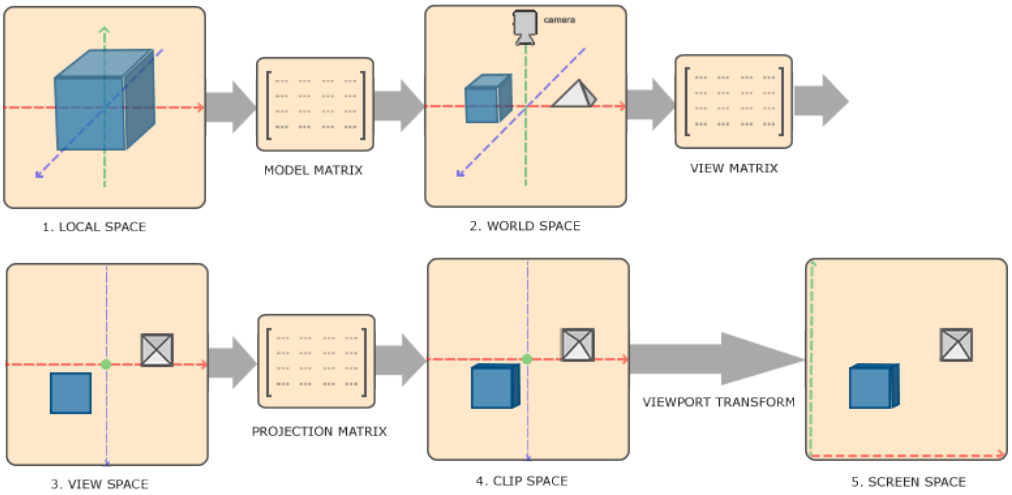


Figure 3.3: OpenGL coordinate system. Image from [dV15].

not, it will be discarded and not shown in the end. The value in this space is determined by the projection matrix, which can be either orthographic or perspective. The visible portion of the scene is called frustum.

Screen space

Screen space is the final coordinate system (in pixels) of objects on the computer screen.

3.2.6 External libraries

Besides the standard STL library of C++, this project has integrated the ImGui graphical user interface and a number of utility files for OpenGL. Even though the developed software is not intended for commercial use, a GUI has been proved to be necessary during development. An interactive interface gives additional information and more control over the inputs, which makes it easier to understands the behaviour of the program, and thus, easier to debug.

Dear ImGui interface

Dear ImGui (or just ImGui) is a popular open-source graphical user interface for C++, mainly aimed at game tool creation. It uses the immediate mode paradigm, which is a type of GUI architecture that favours simplicity, giving the developer the ability to

create a UI window anytime and anywhere in the code. ImGui offers many default widgets like buttons, sliders, labels, colour pickers etc.

Traditional GUI relies on the retainer mode paradigm, in which everything reacts with interrupt signals(events) that trigger the linked callback. Retainer mode GUIs are, as a result, more complicated to use since GUI object must be manually created and destroyed. While in immediate mode, there are almost no states to track because the UI is drawn from scratch on every frame.

Miscellaneous

A number of utility classes have been included, and modified, in the project. The original source files are from [dV15], these file include:

- The Camera class encapsulates the parameters of the camera such as position, orientation vectors, and the amount of yaw and pitch to apply.
- The Shader class encapsulates the shader programs (vertex shader and fragment shader) which are loaded form file at the beginning of the main function.
- The Model class is used to load external 3D meshes in *.obj* format.

3.3 Project design

This section will present the overall design of the software component of the project. This is necessary to understand the implementation details in the next chapter, as well as the code base (see appendix C for access).

3.3.1 Detailed requirements

Application

- **Rendering of 3D objects**: the application must be capable of rendering 3D primitives such as cubes, spheres, cylinders, and planes with different sizes, positions, and shaders.
- **Reference plane**: the floor on which the animation takes place. It is used to help the user understand the position and orientation of the character, relative to the world.

- **Camera system**: the user must have the ability to move the camera direction and position.
- **GUI**: a graphical user interface must be integrated so that the user can interact with the application and visualise useful information.

Motion playback system

- **Read motion capture files**: the application has to parse both the *.asf* file containing the skeleton structure, and the *.amc* file containing the motion data.
- **Construct the skeleton**: a data structure that stores the properties of an *.asf* file as a set of bones. The skeleton must have a 3D representation for each bone.
- **Apply motion to the skeleton**: motion must be applied on a frame-by-frame basis on each bone.
- **Motion playback controls**: the user can stop and resume the animation, as well as skip to any specific frame.
- **Selection of motion file**: a file selection system to change or add motion files.

Motion synthesis system

- **Generate point cloud**: approximate the skeleton's mesh as a point cloud. A deterministic algorithm is required since we need corresponding point between different point clouds.
- **Distance matrix**: a way to compare two motions, to identify which frames are the most similar. This is useful to find transition points.
- **Blending two motions**: linear interpolation to smoothly transition between one motion to another.
- **Motion graph creation and visualisation**: the application should create a motion graph given a list of motions and a threshold value.
- **Generate new sequences of motion**: using the motion graph to generate new motion. Which can be achieved through graph traversal methods.

3.3.2 Architecture

The software is described from two perspectives: *logical view and process view*. The Logical view is concerned with how the main classes interact with one another. Process view deals with the behaviour of the system at runtime. This methodology is inspired by the *4+1* view model [Kru95] framework, which has two additional views: *deployment and physical*. Due to the relatively small size of the project (in terms of code), it was decided that the last two views were unnecessary.

Logical view

Figure 3.4a shows a UML diagram of the main classes that are part of the motion playback system. A brief description for each of them is given below.

- Pose: contains the new position of the root, and rotation angle (as quaternion) for each joint.

- Animation: instances of this class are used to encapsulate the motion, which is stored as a series of poses. Poses are ordered sequentially and have a corresponding frame number so that the first pose corresponds to frame 1, second pose to frame 2, and so on.

- PointCloud: encapsulates an arbitrary point cloud as a vector of 3D points and the method to compare to another point cloud.

- Bone: stores the properties of a bone segment, which are length, position and orientation. Bone hierarchy is preserved with a parent-child relationship.

- Skeleton: is composed of a collection of bones. It is responsible for updating the bones with a new pose instance. It also produces a point cloud by approximating the bone's mesh.

The motion graph is represented by the *MotionGraph* class (figure 3.4b), which encapsulates a number of vertices (*Vertex* objects) and edges (*Edge* objects). The Adjacent class is used to store the graph as an adjacency list: each vertex has multiple edges coming out. Vertex objects correspond to the original animations, while edge objects correspond to transitions.

Process view

The behaviour of the system is defined by the main function, which is in control of checking for inputs (from keyboard and mouse), updating the states, and rendering all 3D objects and the user interface. These tasks have to be executed many times per second to give the feeling of a reactive real-time simulation. This type of process is referred to as a game loop pattern [Gre17].

Figure 3.5 shows an activity diagram of the process view. Because ImGui uses the immediate mode approach, polling is required to get an event (e.g. button pressed). One of the main purposes of the GUI is to initiate the creation of the motion graph. To create a motion graph, a list of motion capture animations and a threshold must be specified. Whether the graph is created of not, a skeleton is always present in the 3D scene.

The rendering process involves drawing the GUI, skeleton, and the environment (i.e. floor, lights, and background). Once the rendering is completed, the system checks the time it took to execute all the commands: if *dt* (delta time) is less than the desired frame time, it means that the simulation is going too fast. Therefore, the system waits for synchronisation. The projects adopt a frame rate of 120 fps (frame per second), which is the same as the sample rate of the motion capture data. The process is terminated only when the exit command is given.

3.4 Experimental design

This section describes the experiments that will be conducted to evaluate the project. The purpose of these experiments is to find out which scenarios the system struggles with, and in which it works best.

3.4.1 Experiment 1 - motion blending

The objective of this experiment is to test the effectiveness of the motion blending technique. To achieve this, we need a way to measure the blending error at different transition points. A straightforward method to compute the error would be to compare the generated transition to the optimal transition. However, the optimal transition data is not available, but it is possible to estimate the error using a distance metric [Luc02] (see section 4.3 for details).

In this experiment, transition points will be set manually, with incremental errors.

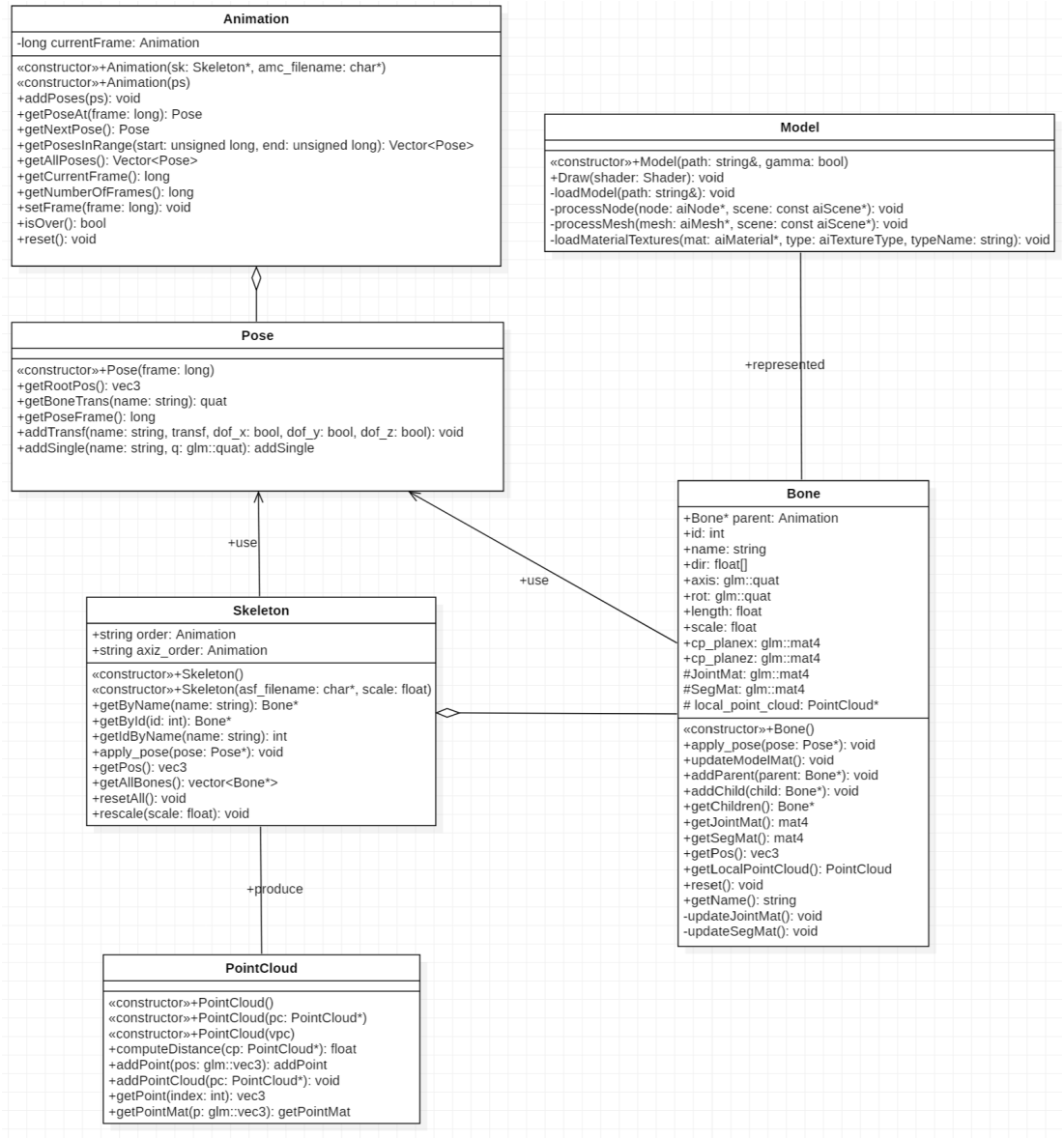
The hypothesis is that a blending with lower distance (or error) will be realistic than one with higher error. Therefore, a threshold between “good“ and “bad” transitions should exist.

3.4.2 Experiment 2 - random graph walks

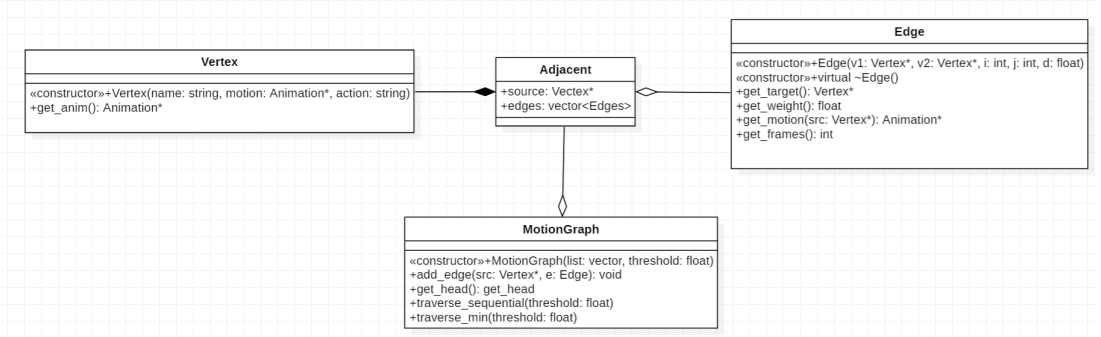
Since graph walks are used to extract motion from the motion graph, any path in the graph can be converted to continuous motion. This experiment is designed to test the validity of the previous statement. In particular, several motion graphs, with varying sizes, are tested with random graph walks. Edges (which represent the transitions) are chosen in a semi-random fashion, by randomly picking one from a pool of top candidates. An edge is considered a top candidate if its cost is among the best 20% (see section 4.7.1). The quality of the graph walk is determined by the average cost of the edges (which are the transitions).

3.4.3 Experiment 3 - chronological graph walk

The purpose of this experiment is to test the more sophisticated algorithm for creating graph walks. Opposite to the random walk, traditional graph search algorithms can be adapted to a motion graph. Branch and bound is used to look for the best path that can cover the graph in chronological order (see section 4.7.2). The order refers to the sequence of nodes that the graph walk has to satisfy. For example, we may want the character to start the animation by running, then do a roll, and then get up and walk away. Again, the cost metric used is the point cloud distance metric (section 4.3). Here the assumption is that it produces a more cohesive motion sequence than an arbitrary random graph walk. More specifically, the chronological path should have a lower average and maximum cost.



(a) Playback system architecture



(b) Motion graph architecture

Figure 3.4: UML class diagrams of main classes.

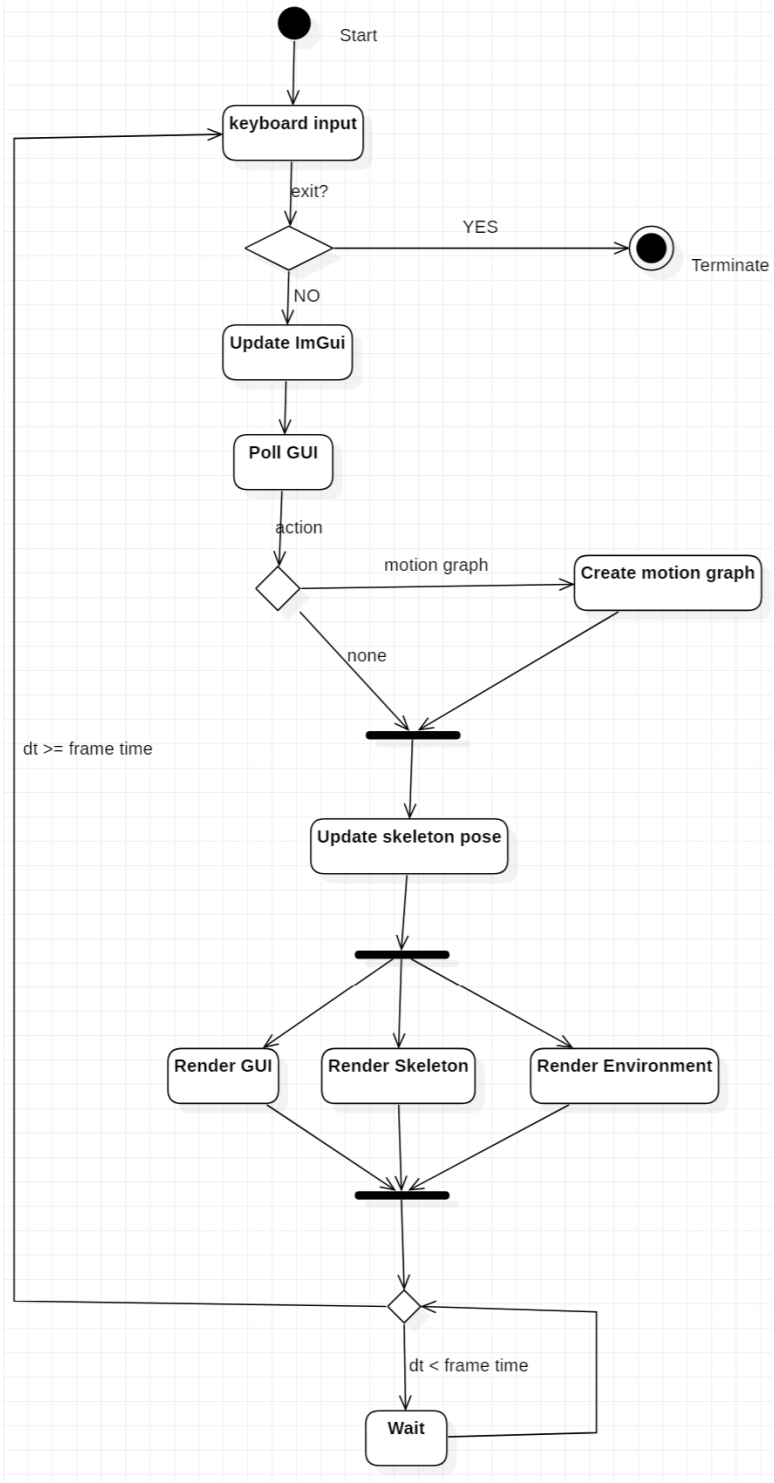


Figure 3.5: Activity diagram of the process view.

Chapter 4

Implementation

This chapter is focused on the implementation details and the challenges faced during the development of the project.

4.1 Building the skeleton

The character is represented as a humanoid skeleton made of a series of bones, which are segments that are connected with each other with joints in between. The *.asf* (3.1) file includes information about how to construct the skeleton, which depicts it as a hierarchical structure. This means that bone properties are relative to their parent. Bone properties include length, axis of rotation, offset direction and orientation. The skeleton is implemented using a tree structure, in which the root node is the "root bone", although it is not really a bone, in the sense that it cannot be represented as a segment. However, every other bone is dependent on the root bone since it gives the global position and orientation, and serves as the starting point of the skeleton building process.

Joints are at the beginning of each bone segment; they are position at the beginning of the bone and are used to create the joint articulation. Hence, by rotating the *humerus* joint, the entire arm is rotated. From the root, as seen in figure 3.1, there are three connecting bones (*lhipjoint, rhipjoint, lowerback*) that have different directions and lengths, then the endpoint of these segments (direction applied to the length) are used as starting point for the bones in the next hierarchical level of the tree. In other words, lower nodes in the tree inherent the transformations of every connected parent node.

4.2 Animation

Once the skeleton is constructed, the animation is applied to it in the form of *poses*. An animation is made of a series of frames, each of which defined by a single pose. Just like a person posing still for a photo to be taken, a skeleton pose is a set of rotations (and translations in case of the root) that can be applied to the skeleton in order to get the desired "photo pose" of the skeleton.

The poses for each frame are specified in the *.amc* file, which contains rotational information of every bone as Euler angles, but is converted to quaternions for convenience. The process of setting a new pose for a skeleton starts with the root bone, which sets the new absolute position and orientation. Then for each remaining bone, only rotation is applied on top of the parent's transformation. The resulting model matrix M (which defines the absolute position and orientation) is calculated as follows [Gle99]:

$$M = P * C * R * Cinv; \tag{4.1}$$

where P is the parent's model matrix with an offset equal to the length of the bone, C is a matrix obtained from the axis values, $Cinv$ its inverse, and R is the rotation matrix from the pose's quaternion. A cylinder is used to represent the bone segment, and it is positioned between point a, the position from the model matrix M, and point b, the offset of M into the direction defined at the skeleton's creation with the bone's length. Using linear algebra, it is possible to find the angle of rotation to align a vector between two points. First, we calculate the axis of rotation by computing the cross product of the initial axis of the cylinder $v = (0,1,0)$ and the vector difference $d = a - b$. The angle of rotation θ can be calculated using simple trigonometry:

$$\theta = cos^{-1}(\frac{v \cdot d}{length}) \tag{4.2}$$

Once the segment is aligned, it is translated in the middle of point a and b and then scaled accordingly.

Note that equation 4.1 uses the OpenGL notation for matrix multiplication, which is right to left.

4.3 Distance metric

In order to choose a good transition point between two motions, one must have a way to measure how close these two motions are for each pair of frames. In particular, we need a distance metric that can capture not only the difference in the current posture but also the meaning with respect to the action performed in the motion. A naïve way to address this problem is to use a vector norm of the pose's parameters, which is a vector composed of the root's position and orientation for each joint in the skeleton. This method results in a loss of information about the context of the movement, such as:

- spatial translation in the 2D plane (moving the character on the floor, XZ) should be irrelevant when comparing poses since the character can perform very similar motion and move in different directions.
- Derivative information like acceleration and velocity cannot be captured if we only consider one pose at a time.
- The hierarchical nature of the skeleton means that a bone's position depends on the orientation of its joint as well as the orientation of all the parent joints. For example, the final position of the forearm, after a 40-degree rotation of the elbow joint, will differ dramatically based on the orientation of the shoulder joint.

The distance metric used in this project is the same used in [Luc02], which is the distance between point clouds over a set window of frames. A point cloud of a pose is the result of a subsampling operation on the character's mesh, which produces a number of points that approximates the 3D shape of the character. However, the construction of the point clouds differs from [Luc02], since they place the points around the joints, while this implementation places the points in both joints and bone segments. Figure 4.1 shows the visualisation of two different point clouds, formed over a total of 81 frames each.

This metric successfully tackles all of the previous shortcomings: the spatial problem on the 2D plane is solved by simply not applying the XZ transformations; derivative information is naturally captured because multiple cloud point (of a single pose) is combined together in a larger, window-sized, cloud point that has information of both previous and following poses. More precisely, the distance $D(A_i, B_j)$ between motion A at frame i and motion B at frame j, with window of k frames, is computed in the following way:

1. select frames from A_{i-k} to A_{i+k} and B_{j-k} to B_{j+k}, so that for each motion we consider $2k+1$ frames, with k frames before and after the selected frames (A_i and B_j).

2. Create a point cloud for every pose inside the window of frames ($2k+1$). Each point cloud is created by sampling along every bone at a regular interval, similar to placing dots on a person's body with a marker. Optionally, one could also apply weights to every point in order to make the metric more sensitive to certain parts of the body, to add additional constrains.

3. Compute sum of square distance for every pair of corresponding points of the point clouds. Note that since a motion does not change when it is translated in the XZ plane or rotated on the Y axis (vertical), it is essential to remove these transformations. This can be achieved by computing the point cloud while the character moves in place.

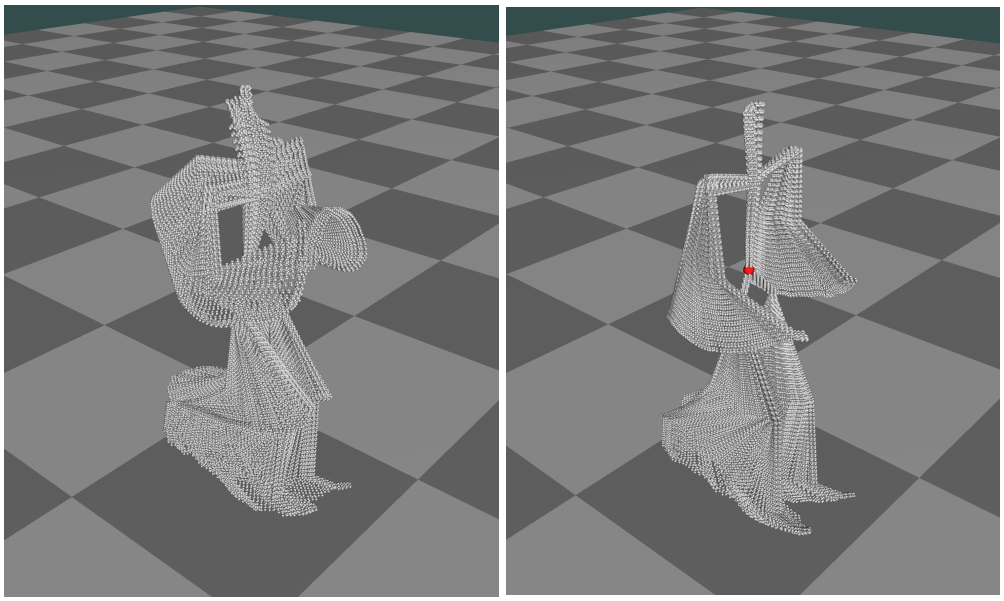
The use of this window of frames captures the derivative information that would have been lost if only a single frame is considered. The order of the derivatives depends on the size of the value k, a high value of k will produce higher-order derivatives and vice versa. The value proposed in this project is 40 frames, which means a total window size of 81 frames, about a $2/3$ of a second.

4.3.1 Distance matrix

The distance matrix is a tool for comparing two motions. It tells us where these motions are similar to each other using the previously defined distance metric. The distance for every possible transition point is computed so that it is possible to choose an ideal transition point. However, the distance matrix is also very useful for visualisation purposes, as shown in figure 4.2, distances are normalised based on the minimum and maximum values in order to produce a bitmap image, in which the darker pixels indicate greater distance while lighter grey values indicate to more similar motions.

The computation of the distance matrix is the most time-consuming process in this project since it must be computed for every pair of motion. It is, for this reason, essential to maximise the efficiency of the calculation. A straightforward algorithm to compute the distance matrix can be described in the following steps:

1. get the point cloud of every frame of motion A and B, then save it in the respective vector.



(a) (b)

Figure 4.1: Point clouds of two different motions, both using 81 total frames ($k = 40$). a) Point cloud of a running animation. b) Point cloud of a walking animation.

2. Iterate over the point cloud vector of motion A and build, for each frame, a cumulative point cloud which is the combination of all the single point cloud within the window size $2k+1$.

3. The distance can then be obtained by computing the sum of square distance of the cumulative point cloud for each pair of frame (A_i, B_j).

If each local point cloud (points of a single pose) has m points, the point cloud of a window would have $m*(2k+1)$ total points, making the complexity of the algorithm (in terms of the number of operations) to be:

$$O(N_A * N_B * m * (2k+1)) \tag{4.3}$$

where N_A and N_B are the number of the frames of motion A and B, respectively.

4.3.2 More efficient computation of the distance matrix

The inefficiency of the above method can be observed in step 2 and 3; much of the computation is wasted recalculating the distance of the same points. For example, the point clouds of A_x and A_{x+1} share almost the same window of frames. In fact, they contain $A_x = [x-k, ..., x+k]$, and $A_{x+1} = [x-k+1, ..., x+k+1]$, which means they differ

Figure 4.2: Visualisation of a distance matrix, where pixel correspond to a pair of frames. The brighter areas indicate a lower distance where motions are more similar. Local minimas are represented as green dots.

by only two elements while sharing $2k-1$. Therefore, we need to exploit this property to reduce the number of operations. The distance $D(A_i, B_j)$ can be reformulated as:

$$D(A_i, B_j) = D(A_{i-1}, B_{j-1}) - d(A_{i-k-1}, B_{j-k-1}) + d(A_{i+k}, B_{j+k}) \tag{4.4}$$

where D is the distance between the window size point clouds, and d is the distance between the smaller point clouds of the single poses. Using equation 4.4 to compute the distance, it is possible to reduce the computational overhead by a factor of $(2k+1)$, resulting in the following complexity:

$$O(N_A * N_B * m) \tag{4.5}$$

Effectively, this optimisation translates in at least an order of magnitude in performance gain, depending on the value of k.

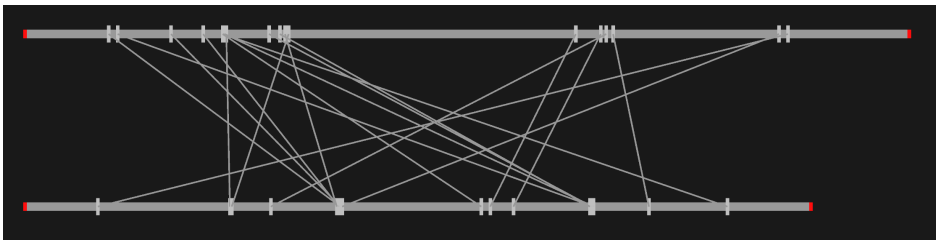


Figure 4.3: Transition edges between two motions.

4.4 Finding transition points

Our goal is to find good transition points from one motion to another. Having computed the distance matrix, we know the distance for each pair of frames, in which the lowest distance translates to the best transition point. However, it is not always optimal to choose the best transition point, since we want the system to be flexible. By using a threshold value, it is possible to find all those transition points that are at least considered "good enough". The threshold is determined by the user because it can be varied depending on the situation, setting a small value will create more believable transitions, which may be necessary for running motions since people see it every day, but for other motions like ballet dancing, it is sufficient to use a lower value. When the threshold is applied, only the point where the distance is lower is considered as candidate transition points.

In addition to the threshold mechanism, we use local minima instead of selecting every valid point (with no discrimination) that has the distance below the threshold. Choosing the local minima as transition candidates have the effect of removing redundant points. In fact, similar distances tend to cluster around the same area in a distance matrix, which is natural since motion does not change a lot frame by frame. Local minimas are defined as points that have a lower distance value than all of their eight neighbours in the distance matrix, and are represented as green dots in figure 4.4 (only two are present because a threshold is applied). As a result, the local minima is the best choice when there are many similar points around a specific area of interest. Figure 4.3 shows all possible transitions between two motions.

4.5 Motion blending

Motion blending is the task of creating believable transitions from one motion to another. In our case, once the transition points are found, we must smoothly transition to

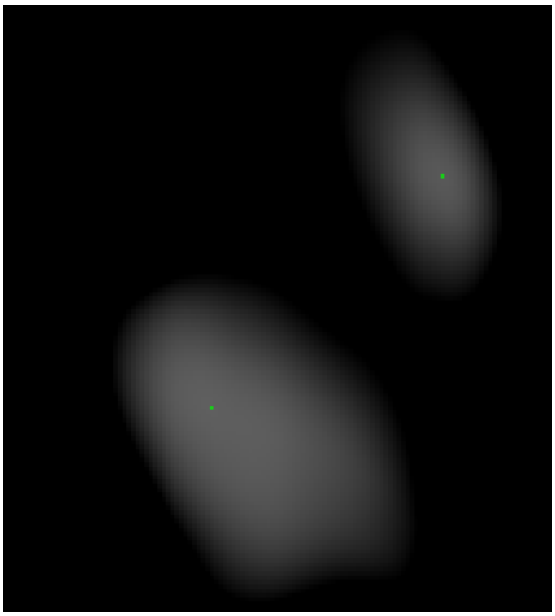


Figure 4.4: Distance matrix with two clusters. Choosing the local minimas (green dots) as transition points greatly reduces the number of transitions.

the target motion by creating intermediate poses that are a mixture of both motions.

The blending technique adopted in this project is the linear interpolation of the poses' parameters, i.e. root position and joints orientation. Although linear interpolation is often considered a weak solution for animations that are nothing alike (e.g. running and jumping), it is, however, proven to be optimal for motions that are sufficiently similar [Luc02]. Therefore, by combining linear blending and the distance metric, we can achieve realistic transitions, given that the motions are "close" at least in some points.

Firstly, the root position is linearly interpolated, which involves interpolating two points a and b using the following equation:

$$R_t = (1-t) * a + t * b \tag{4.6}$$

where $t \in [0,1]$, so that $R_0 = a$, and $R_1 = b$. Since we are using a window of frames $2k+1$, the blending must interpolate frames A_{i-k} to A_{i+k} with frames B_{j-k} to B_{j+k}. The intermediate root position and joint rotations are computed as:

$$R_t = \alpha(t) R_{A_{i-k+t}} + [1-\alpha(t)] R_{B_{j-k+t}} \tag{4.7}$$

$$q_t^i = slerp(q^i_{A_{i-k+t}}, q^i_{B_{i-k+t}}, \alpha(t)) \tag{4.8}$$

where $\alpha(t)$ is the continuity constraint that maps t to $[0,1]$, since t is the transition frame $(-1 < t < 2k+1)$, and q_t^i is the rotation of the joint i at the frame t of the transition. The continuity constraint used is a polynomial, proposed in [Luc02]:

$$\alpha(t) = 2(\frac{t+1}{k})^3 - 3(\frac{t+1}{k})^2 + 1 \tag{4.9}$$

other constraints have been proposed in [KG04], [RGBC96] and [LCR+02].

4.6 Constructing the motion graph

The motion graph is created using a list of input animations, and it is implemented as an adjacency list. A vertex represents an animation, and it can have an arbitrary number of edges, which are transitions to other vertices or to itself. An edge contains information regarding the target vertex, the transition point, and a weight value which is the point cloud distance associated with that transition. The number of vertices is determined by a number of animations, while the number of edges per vertex is equal to the number of local minimas in the distance matrix of each pair of motion. For example, suppose that we construct a motion graph with 5 animations, where each pair of animations has 10 local minima points. The result is a graph with 5 vertices, each of which will have 50 edges (10 edges targeting each motion). Figure 4.5 illustrates a motion graph constructed using 5 motions.

An alternative graph structure could have been developed following the original implementation in [Luc02], where animations are split each time a new edge is inserted, thus adding a new vertex every time. This creates a more complex graph to navigate since it has a much larger number of nodes, and it also requires to create and store the transition motion for each edge, as the reference to the original animation is lost. In larger motion graphs, storing each motion blend in memory could be an issue, due to the exponential number of edges. The implementation proposed here does not suffer from this issue since motion can be blended at any time, given a pair of vertex and edge.

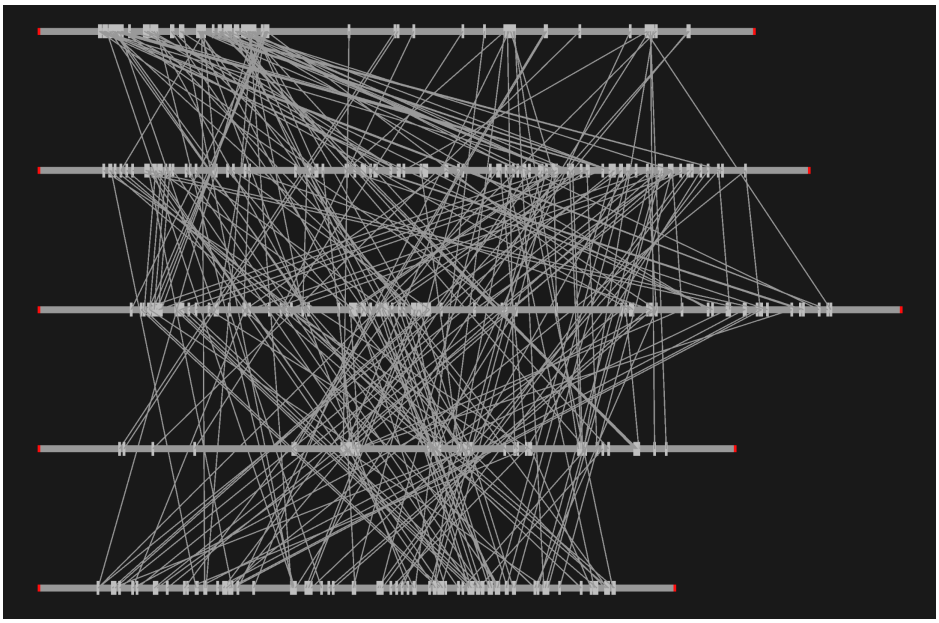
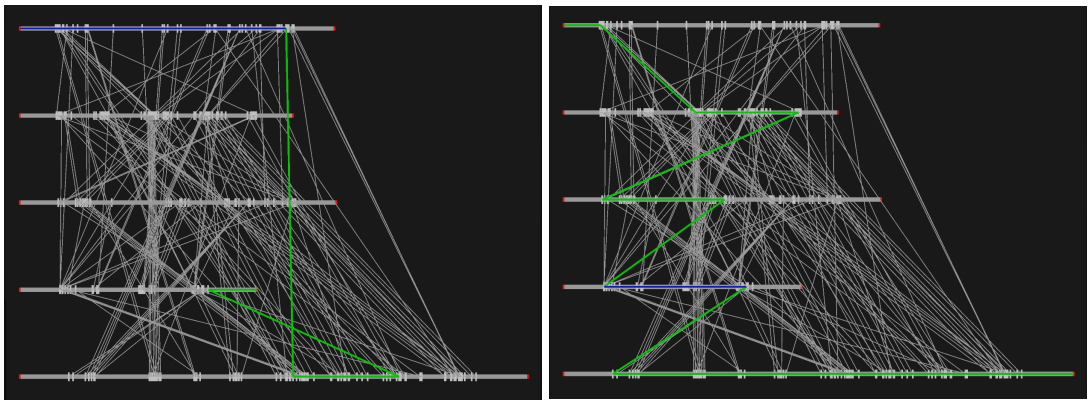


Figure 4.5: Motion graph constructed from 5 animations.

4.7 Generating motion

Once the motion graph is constructed, motion can be extracted by traversing the graph. The obvious way to do this is to use a traditional graph traversal algorithm such as Dijkstra's shortest path or other similar methods. While this approach may seem appealing at first, it is flawed in the sense that the generated motion does not convey any particular meaning, since there is no objective specified. Instead, we can design graph walks such that the desired output is produced.

One possible objective of the search is to traverse each animation in a sequential way; this produces a sequence of motion that follows a user-defined order. For example, the user can specify an ordered list of animations that starts with walking, running, sneaking, and finish with a jump. This traversal method requires a path that connects each motion to the next on the list, which can be problematic if additional constraints are considered, like a threshold value that removes certain edges, or a minimum amount of frames that each motion has to last before going to the next. Therefore, the presence of a valid graph walk is not always guaranteed. In this project, a greedy search algorithm as well as a sequential graph walk approach is presented.



(a) Greedy path (b) Sequential path

Figure 4.6: Sample motion graph paths using the greedy (a) and the sequential (b) algorithms.

4.7.1 Greedy search

The greedy approach always selects the most preferable (the one with the lowest distance) transition point. In testing this algorithm, it has been observed that the graph walk generated is very short, even when bigger motion graphs are used. The average number of transitions is only around 2 to 3 edges. However, given the right conditions, the algorithm can loop through the graph indefinitely since it always selects an edge if there is one available.

For these reasons, a modification was introduced to create more worthwhile sequences. Instead of starting the walk at the same point every time, a randomised mechanism picks from a pool of the best candidates (the top 5% of the valid edges). The edges to be selected have to be valid, which means that the starting frame of the transition must equal or greater than the current position, due to the architecture of the graph, as described in the previous section. To avoid infinite walks, a maximum number of transitions is set, so that if the search stops when this limit is reached.

Figure 4.6a shows a graph walk using the greedy approach. Note that the path does not cover the entire graph, because it finishes when there are no valid edges to select.

4.7.2 Sequential graph walk

The sequential method returns a graph walk that covers the input animations in order. This can be useful, for example, in the context of crowd movement by blending a variety of walking motions. The sequential graph walk is implemented by finding a path that can cover every vertex of the graph since the number of edges is exponential;

it is unfeasible to explore all the possible path. Therefore, a heuristic method has been adopted: perform a set number of trial and error attempts to find a possible path, if there is no path after n trials then the algorithm returns the longest path that has been found so far. At each iteration, edges are filtered using the threshold, and the way an edge is selected to be inserted in the path is by assigning a probability that depends on the distance, a lower distance translates to a higher probability of selection while high distances have very low probabilities. This ensures that, first, the same edges are not chosen every time, which gives more variety, and second, it makes sure that better transitions are always preferred overall.

Figure 4.6b shows the path of a sequential graph walk. Compared to the greedy approach, it spans across all graph vertices, forming a longer path.

Chapter 5

Results and evaluation

This chapter presents the experimental results of the project and an evaluation of the work. First, the motion blending technique is tested to understand how it is affected by the distance metric. Then, results for the two graph traversal algorithms are presented. In the second section, the overall results and limitations are discussed.

5.1 Experimental results

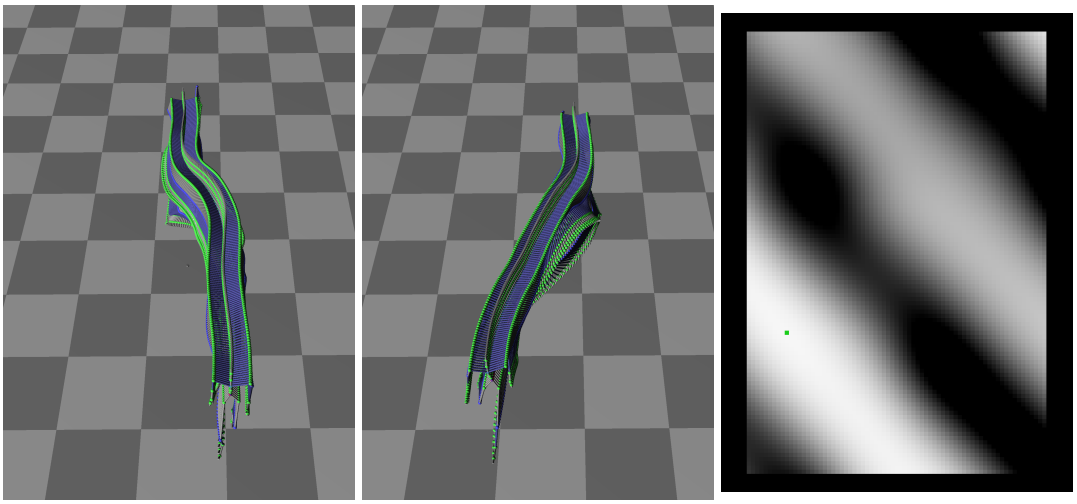
The following results were obtained on a machine running the Linux operating system (Ubuntu 18.04), equipped with an AMD Ryzen 1700 CPU at 3 GHz and 16 GB of memory.

5.1.1 Motion blending

To test the effectiveness of the motion blending technique, two cases are analysed:

- A high similarity case where the motions share attributes such as direction, speed, and overall posture. The selected motions are two running animations that have a slight veer, one to left and other to the right (figure 5.1).

- A low similarity case where the motions are very different. To explore this case, a walking animation and a ballet animation were chosen (figure 5.5). Combining a simple motion to a highly complex one.



(a) (b) (c)

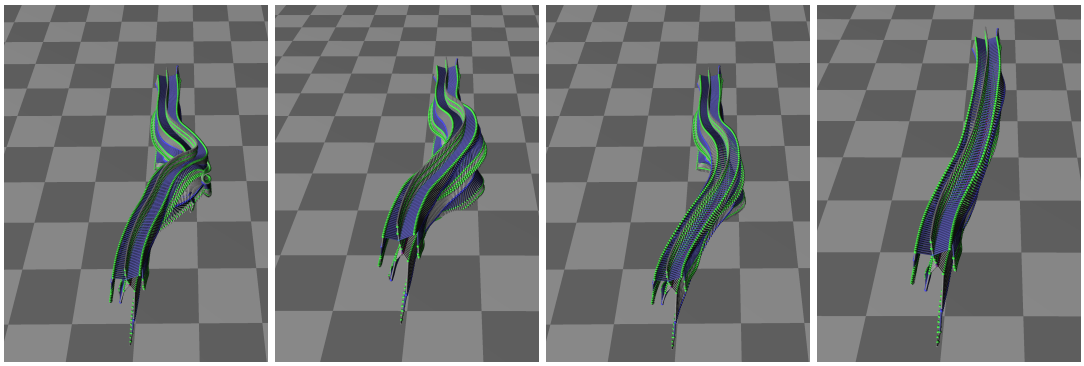
Figure 5.1: (a) Flow of the run and veer left animation. (b) Flow of the run and veer right animation. (c) Distance matrix of (a) and (b)

High similarity

Figure 5.1 shows the motions depicted as flows of all the single poses (or frames) displayed at once. We define motion A as the motion on the left (figure 5.1a), and motion B one the right (figure 5.1a). The file name of A is *127_13.amc*, while B is *127_14.amc*, with 185 and 151 number of frames respectively.

The distance matrix of motion A and B, shown in figure 5.1c, reveals that all the distances converge to a single local minima. Although usually there are several local minimas, this phenomenon can be explained by the fact that A and B are similar throughout the distance matrix, which does not favour the formation of clusters. Note that in figure 5.1c, a threshold of $t = 2000$ is applied to distance matrix to see in which area we have the most similarity.

Figure 5.2 shows the results of the blending of four different transition points, from the lowest to the highest distance. The results suggest that varying the distance d has a significant impact on the complexity of the interpolated motion. The transition with the lowest distance ($d = 70.15$) shows much more intricacy than the one with the highest value ($d = 2871.20$). The latter follows an almost completely straight path, losing all the details from both the original motions. The former was able to combine the left veer and the right veer, from motion A and B, respectively. The intermediate values ($d = 568.51$ and $d = 1064.73$) of figure 5.2 shows a less pronounced veer effect to the left compared to the lowest d value. However, the same end position is reached in all



(a) $d = 70.15$ (b) $d = 568.51$ (c) $d = 1064.73$ (d) $d = 2871.20$

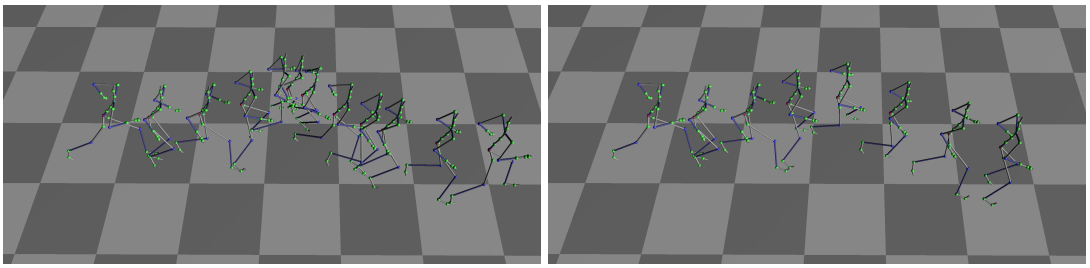
Figure 5.2: Blending results of the two running motions at different transaction distances.

cases. Further information can be deduced from figure 5.3, it shows the same blending animation of figure 5.2 but with a side view. From this point of view, it is possible to notice a correlation between the distance metric and the quality of the transition. In particular, it shows a snapshot of the poses at an interval of 20 frames each. The reason why the low distance transition ($d = 70.15$) appears to have more snapshots is that its transition point is more favourable, meaning that it produces a longer overall animation (i.e. choosing the blend a frame towards the end of A, to a frame at the beginning of B). At the opposite end, the resulting animation of the high-value transitions is very short due to the poor location of the transition (i.e. blending towards the end of B).

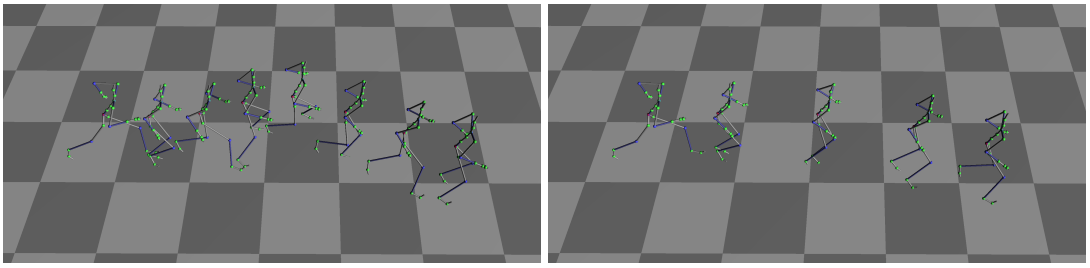
Low similarity

In this case, the motions are a generic walk, and a pirouette, using the animation *5_01.amc* (597 frames) for the former, and *5_11.amc* (590 frames) for the latter. Figure 5.5a shows the walking motion, which is quite straightforward. Figure 5.5b shows the more complex ballet motion; it is represented as a composition of frame-by-frame poses to captures the inherent speed of the pirouette.

The transition point chosen is at the location $319 - 277$ (frame of the first motion, and frame of the second motion), with a distance of 2444.85. Although the transition is associated with a large distance, it is still a low value relative to its distance matrix (figure 5.5c), which has a threshold of 6000. The result of the blending is in shown figure 5.4, where the technique appears to be able to transition to a completely different motion in a stable way. Of course, given the high distance value, the result is not without some defects. Although it can be hard to tell from still images, when the



(a) $d = 70.15$ (b) $d = 568.51$



(c) $d = 1064.73$ (d) $d = 2871.20$

Figure 5.3: Side view of the blending results of two running motions at different distances.

animation is played, clear sliding artefacts start to appear. Transition errors are more accentuated if we choose a transition point with an even higher distance, as shown in figure A.1 of the appendix A.

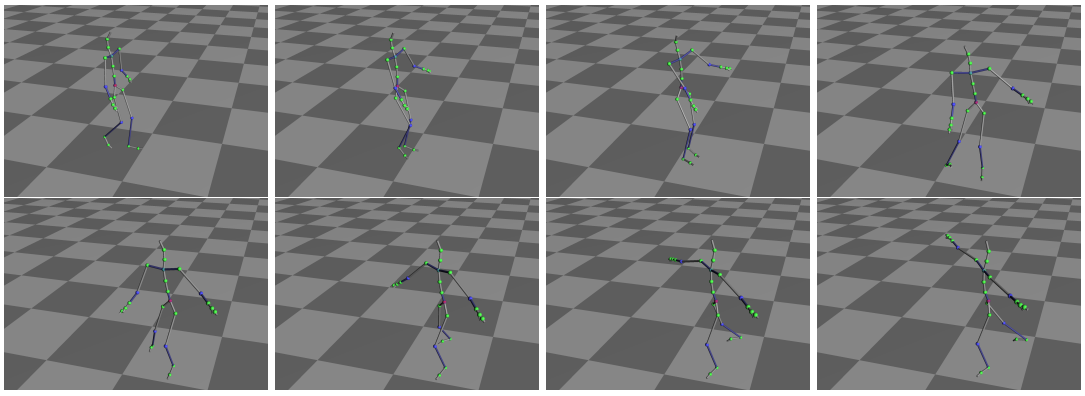
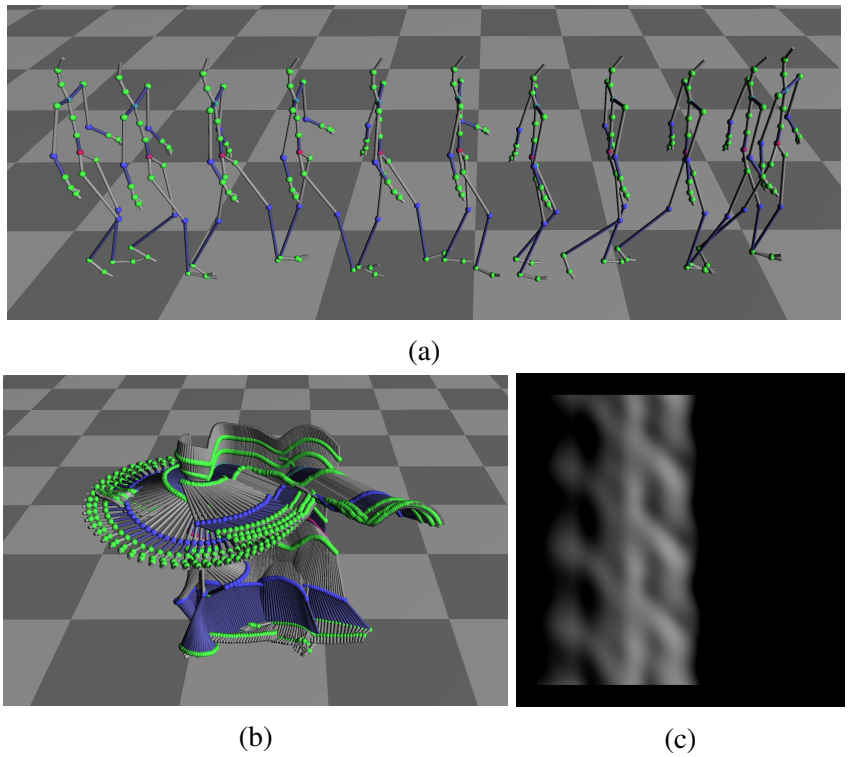


Figure 5.4: Result of blending a walk motion and a pirouette. The animation is depicted using snapshots of 10 frames from left to right. The distance of the transition is 2444.45.



(a)

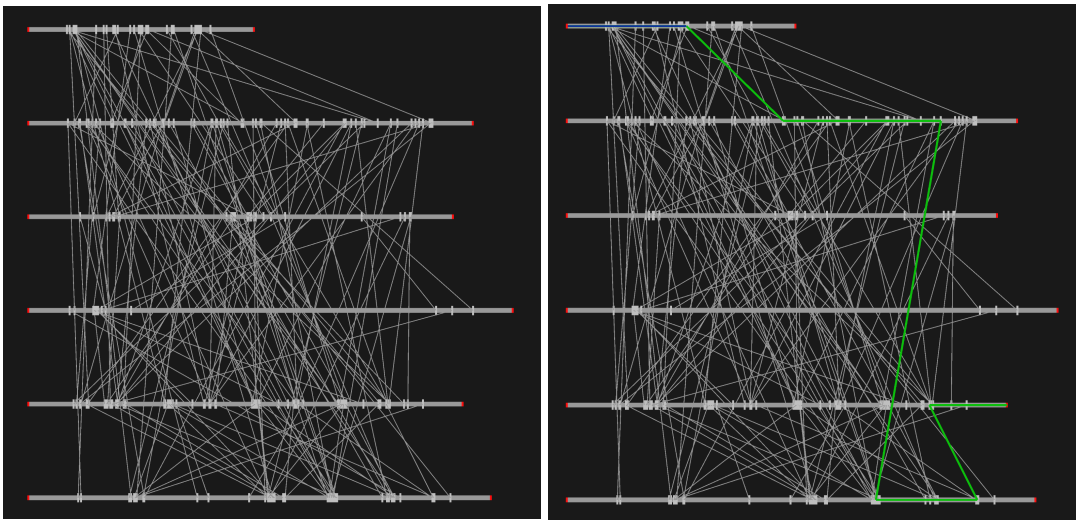
(b) (c)

Figure 5.5: (a) A straight walking motion. (b) The flow of a pirouette. (c) Distance matrix between motion (a) and (b). A threshold of 6000 is applied to the matrix.

5.1.2 Random graph walk

This experiment tests the feasibility of the motion graph to create meaningful connections between an arbitrary number of motions. The testing motion graph contains six animations involving locomotion. More specifically, these motions are *walk, run, duck, sneak, jump up and down, and jump on one leg*, which accounts for 2623 combined frames, taking 336 seconds to compute. The graph is made of 6 vertices and 380 edges, which have the following properties:

- maximum edge distance is $d = 20743.55$;
- minimum edge distance is $d = 18.92$;
- average edge distance is $d = 1834.15$;
- the most connected vertex has 96 edges;
- the least connected vertex has 38 edges;
- the average connection per vertex is 63;



(a) Motion graph (b) Greedy graph path

Figure 5.6: (a) Representation of the motion graph. A threshold of $t = 1000$ is applied. (b) Random graph path using the greedy algorithm. The green line lines represent the path, while the single blue line indicates the starting point.

Random walks are calculated using the greedy algorithm described in section 4.7.1. Generally, it tries to guess a good candidate transition by selecting a random sample in the top edges pool. As a result, the computed graph path changes every time the algorithm is executed. The generated path can vary in the number of transitions, the overall length of the animation, and the quality of the transitions.

Figure 5.7 shows a path made of 3 transitions, with an average distance of 197.75. This translates to an animation of 482 frames that starts with running, and transitions to walking, jumping on one leg, and finishes with a jump on both feet. Only a single jump from the *jump up and down* animation is included because the last transaction is connected towards the end of the original motion. The transitions are:

1. *run* to *walk*, at frames 129-193, $d = 145.04$.

2. *walk* to *one leg jump*, at frames 418-113, $d = 366.075$.

3. *one leg jump* to *jump*, at frames 429-379, $d = 82.13$.

Two vertices are not visited by the random graph walk because a dead end is reached in the last vertex. However, running the greedy algorithm 20 times, produced on average very similar results to the one above. The average number of transitions was 3 (in a range of 1 to 5), with an average distance of $d = 214.64$. Alternative paths are shown in figure A.3 in the appendix.

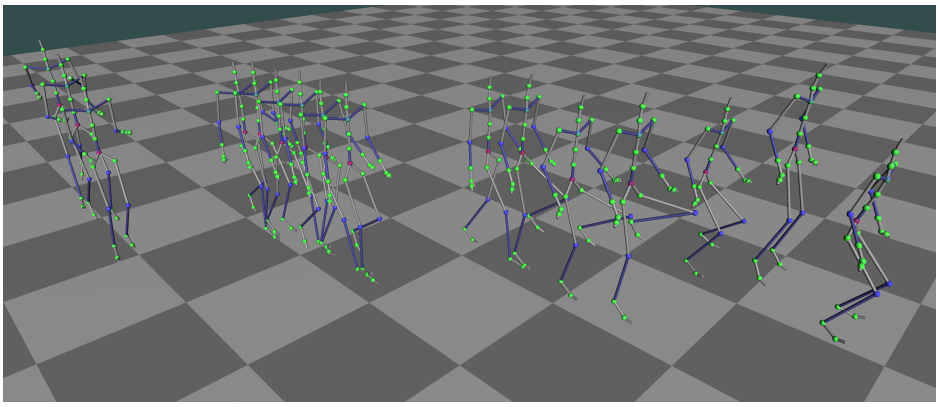


Figure 5.7: Generated motion sequence extracted from the motion graph using the random graph path. An offset is applied at each transition to aid in the visualisation.

5.1.3 Chronological graph walk

In this experiment, the sequential algorithm (from section 4.7.2) is used to compute a path that connects all the motions in a specific chronological order. The same motion graph from the previous experiment is used, making comparison between the two methods easier.

Figure 5.8 shows that the sequential algorithm can find a path (in green) to visit all the vertices of the motion graph. This creates a much longer animation than a random walk: 982 frames long (about 8.6 seconds) compared to 482 (4.0 seconds). Since the vertices are ordered, based on the desired sequence, the chronological walk follows a top to bottom path. Except for the last vertex, all vertices are visited for a significant chunk of frames relative to their individual lengths, meaning that their corresponding motion is part of the synthesis. The path includes the following transitions:

- *run* to *walk*, at frames 125-226, $d = 20.22$;
- *walk* to *sneak*, at frames 425-91, $d = 516.24$;
- *sneak* to *duck*, at frames 241-72, $d = 389.47$;
- *duck* to *jump*, at frames 449-89, $d = 692.66$;
- *jump* to *one leg jump*, at frames 379-429, $d = 82.13$.

Figure 5.9 shows the synthesised animation, separated with an offset at every transition in order to have a clearer representation of the different types of motion. Each snapshot of the skeleton is 30 frames apart. The generated animation starts with the

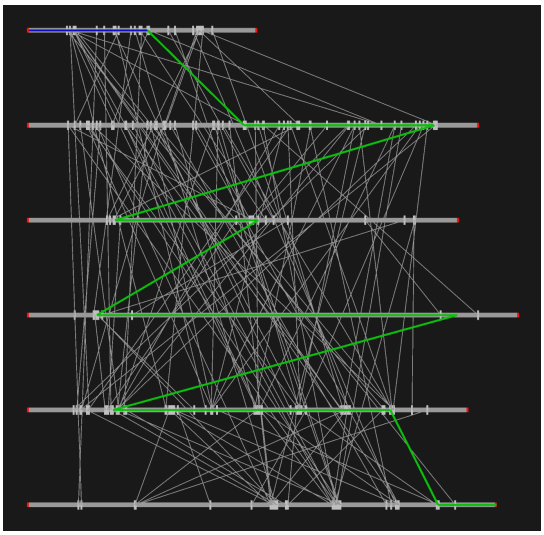


Figure 5.8: Chronological graph path using the sequential algorithm. The green line lines represent the path, while the single blue line indicates the starting point.

original running motion for 125 frames (1 second); blends to walking motion for 199 frames (1.7 seconds); blends to the sneaking motion for 150 frames (1.3 seconds); blends to the ducking motion for 377 frames (3.1 seconds); blends to jumping for 110 frames (0.9 seconds); finishes with a single, one leg jump for 21 frames (0.18 seconds).

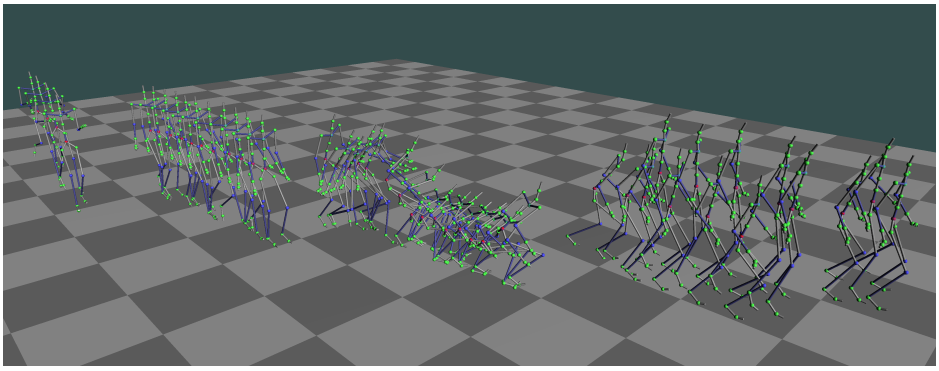


Figure 5.9: Synthesised motion extracted from the motion graph using the chronological graph path. An offset is applied at each transition to aid in the visualisation.

5.2 Evaluation

This section gives a discussion of the results obtained in the experiments, strengths and limitations of the current state of the project are presented.

5.2.1 Motion blending

The experiments showed that motion blending using linear interpolations can produce realistic results if the motions are similar enough. The distance metric implemented in this project can successfully discriminate a good transition point from bad ones. The results from the high similarity test, illustrates the direct correlation between the distance value and the quality of the blending, low distance values (less than 1000) produce better animations than a high distance (more than 1000). In low similarity cases, when the motions share no resemblances (the case of walking and ballet), linear interpolation is not sufficient. In particular, the direction and orientation of the characters play a significant role: if two characters go in opposite directions, it is improbable that a low distance transition point exits between the two. This issue can be observed in the distance matrix of the walking motion and the pirouette in figure 5.1c, where the left side is completely dark because the ballet dancer was going (and facing) in the opposite direction of the walking motion. In fact, the transition in figure 5.4 happens in a location where the ballet dancer turns around in the same direction as the walking.

5.2.2 Motion graph implementation

The experimental results demonstrate that a correct implementation of the motion graph data structure has been developed. Compared to the original implementation of motion graph [Luc02], the representation of the graph is more compact, with the same number of vertices as the number of animations, instead of splitting animations into smaller clips creating a graph with many more vertices.

Transition points are found by selecting the local minimas of the distance matrix. Although this does not guarantee that a transition point will be low distance, however, it identifies plausible locations for good transitions.

The advantage of using motion graphs is that it is possible to use any graph algorithm to look for motions. The random graph walk experiment proves that any graph walk can be converted into a sequence of motion. The results show that a very reasonable animation can be produced even from a random path, although short in length.

On the other hand, the chronological graph walk produces more meaningful sequences by restricting the order of traversal, which restricts the search problem by a large margin. This is important because it means that it can be applied to real-time application, such as interactive control in video games. The performances of the graph traversal algorithm also limits the size of the motion graph that we can build, since a

large motion graph is of little use if we cannot compute a proper path.

The current motion graph implementation also supports online construction, which makes it possible to keep building the graph as new motion data comes in, without the need for recomputing the old connections. This feature is particularly useful for updating the motion graph with relevant data.

The most appropriate use for the system implemented in this project (in the current state), would be crowd simulation. Even though the quality of the synthesised motion can be natural-looking, it still is not flexible enough, since precise control of the animation is lacking. In the case of the greedy algorithm, no control is available except for the starting animation. The sequential algorithm only offers a coarse level of control, i.e., it is possible to choose the overall structure of the animation, but finer details, like position constraints at a certain location, are currently not achievable. However, it is reasonable to say that, with some minor improvements, the system can be used to control the character's movement interactively. For example, if we would carefully design a motion graph with an idle animation that is properly connected to many locomotion nodes in the graph, a relatively simple algorithm can be developed to blend the appropriate motion when an input from the user is received. If the input indicates a left turn, we can blend to the *turn left* vertex using the next available transition.

5.2.3 Limitations

Although the objectives of the project have been met, the following limitations still need to be addressed:

- The motion capture data has to be manually categorised, with labels that can be used to describe the motion. This is both time-consuming and not ideal since motion is naturally ambiguous. Moreover, in the CMU motion capture database, it is sometimes necessary to edit the motion files in order to delete unwanted frames.

- The interpolation technique, at the moment, does not account for the difference in the direction and orientation of the target motion clip. Rather, only the root position (of the target motion) is normalised so that the transition does not slide if there is an offset between the two motions. Direction and orientation must also be considered in order to combine all types of motion. For example, in the current interpolation, it is not possible to apply a *walk and turn left* motion twice to back a U-turn.

- Motion graphs have edges that lead to dead ends, meaning that after transitioning to that point, there are no further transitions available. Self-transition can be generated to mitigate this issue.

- Generally, in data-driven methods, such as motion graph, it is very difficult to have the character respond to the environment. This can be a problem, in case there are obstacles in the scene.

- The time complexity of the motion graph is exponential, due to the need to compute the distance matrix for every pair of motion. However, the algorithm can be parallelised to improve performance. Since there are no concurrency issues, a multi-core version would be straightforward to implement. Another performance boost can be achieved by running on graphics cards (e.g. using CUDA or OpenCL).

Chapter 6

Conclusion

This work started with a study of motion capture technology, which is essential for understanding the research area of motion synthesis. Because the aims of the project were to implement a system, from scratch, that is capable of automatically generating new sequences of motion, a deep level of knowledge needed to be achieved.

The first step was to build a 3D application in OpenGL, to animate a virtual skeleton using motion capture data. This application would later be the foundation of the main system that we want to create. A skeleton system was implemented using an object-oriented approach by abstracting the structure in an arbitrary configuration of bones objects. Then, a parser was developed to read any motion file from the CMU motion capture database.

The main challenge was the implementation of the motion graph technique. Because it involves the development of many sub-systems, such as the point cloud distance metric, motion blending, and tools to interact and visualise the graph structure. A reliable distance metric is vital for the correct construction of a motion graph since it determines how the vertices in the graph are connected. In particular, the distance must account for derivative information: it is not enough to compare just the pose, but its context plays a much bigger role. Hence the developed metric accounts for a window of motion, before and after the pose of interest. Motion blending is extensively used once the motion graph is constructed; it is implemented using linear interpolation of the position and rotation of the skeleton.

Two graph traversal algorithms have been developed to extract motion from the graph. The greedy algorithm computes a random graph path, but it does so by selecting from a pool of edges with relative low distance. Although a random path is not that meaningful, it can be used to prove the correctness of the motion graph. In the

experimental results, we observed that any random walk produces a valid sequence of motion. The other developed graph traversal algorithm is the sequential algorithm, which returns a graph path that follows a specific chronological order. The sequential algorithm was designed with the objective of being computationally inexpensive. Because of the exponential growth of the motion graph when additional motions are added, traditionally graph search algorithms are not feasible to use for larger graphs. On the other hand, the sequential algorithm significantly reduces the search space of the graph by imposing a predefined order. Results show that this algorithm can successfully find a path connecting all the vertices.

As the project grew in both size and complexity, a graphical user interface became necessary to interact with the application. More specifically, visualisation tools were developed to display the distance matrix and the motion graph. This has been proven to augment the productivity and correctness of the implementation, essentially making debugging much easier and less time-consuming. Other tools were developed to interact with the 3D scene: moving the camera; playback controls; file selection and toggle of various options.

Overall, the work on this project progressed without many complications, apart from the beginning stage, when a lot of code was necessary to build the motion playback system. In conclusion, I believe that this project was successful since all the initial objectives have been met. However, improvements can always be made.

6.1 Further work

Further work include the following topics:

- the distance metric can be improved using as set of weights based on the importance of particular parts of the body. For example. if the motion of the upper body was deemed more important, then it is possible to tweak the weights so that it becomes more sensitive to that specific region.

- More complex blending algorithms can be developed, which would increase the fidelity of the transitions. Possibly, registration curves [KG03] can be implemented.

- Alternative graph traversal algorithms can be implemented based on specific need. Because there is no general purpose solution for all types of motion, we have to tailor the algorithm to the problem.

- A physics-based system can be integrated into the existing implementation. The combined system would be capable of producing both realistic and reactive motion. A possible approach could have both systems (motion graph and physics simulation) running in parallel, and switch to one or the other based on the context. For example, when a collision is detected we would switch to the physics system.

Bibliography

[AF02] Okan Arikan and D. A. Forsyth. Interactive motion generation from examples. *ACM Transactions on Graphics*, 21(3):483–490, 2002.

[AG85] William W Armstrong and Mark W Green. The dynamics of articulated rigid bodies for purposes of animation. *The visual computer*, 1(4):231–240, 1985.

[BN92] Thaddeus Baier and Shawn Neely. Feature-Based Image Metamorphosis, 1992.

[BW95] Armin Bruderlin and Lance Williams. Motion signal processing. pages 97–104, 1995.

[CECS18] Steffi L Colyer, Murray Evans, Darren P Cosker, and Aki IT Salo. A review of the evolution of vision-based motion analysis and the integration of advanced computer vision methods towards developing a markerless system. *Sports medicine-open*, 4(1):24, 2018.

[dV15] Joey de Vries. Learn opengl. *Licensed under CC BY*, 4, 2015.

[FvdPT01] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. (1):251–260, 2001.

[Gil09] Marco Gillies. Learning finite-state machine controllers from motion capture data. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):63–72, 2009.

[GJH01] Aphrodite Galata, Neil Johnson, and David Hogg. Learning variable-length Markov models of behavior. *Computer Vision and Image Understanding*, 81(3):398–413, 2001.

[GJSS01] Min Gyu, Choi Jehee, Lee Sung, and Yong Shin. A Randomized Approach to Planning Biped Locomotion with Prescribed Motions. *Work*, 2001.

[Gle99] Michael Gleicher. *Acclaim ASF/AMC*, 1999. `http://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/ASF-AMC.html`.

[GP12] Thomas Geijtenbeek and Nicolas Pronost. Interactive character animation using simulated physics: A state-of-the-art review. In *Computer Graphics Forum*, volume 31, pages 2492–2515. Wiley Online Library, 2012.

[Gre13] Jason Gregory. Game engine architecture. *Choice Reviews Online*, 47(05):47–2616–47–2616, 2013.

[Gre17] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2017.

[GSC+15] Shihui Guo, Richard Southern, Jian Chang, David Greer, and Jian Jun Zhang. Adaptive motion synthesis for virtual characters: a survey. *The Visual Computer*, 31(5):497–512, 2015.

[GVDPVDS13] Thomas Geijtenbeek, Michiel Van De Panne, and A Frank Van Der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics (TOG)*, 32(6):206, 2013.

[HG07] Rachel Heck and Michael Gleicher. Parametric motion graphs. page 129, 2007.

[HKJ17] DANIEL HOLDEN, TAKU KOMURA, and JUN SAITO. Phase-Functioned Neural Networks for Character Control. *ACM Transactions on Graphics, Vol. 36*, 36(4):491–533, 2017.

[HT00] Adrian Hi and Luis Molina Tanco. Realistic Synthesis of Novel Human Movements from a Database of Motion Capture Examples. *Proceedings Workshop on Human Motion*, pages 137–142, 2000.

[JAJ+04] Jernej Barbic, Alla Safonova, Jia-Yu Pan, Christos Faloutsos, Jessica K. Hodgins, and Nancy S. Pollard. *Segmenting motion capture data into distinct behaviors*. 2004.

[JC99] Aggarwal J.K. and Q. Cai. Human Motion Analysis: A Review. *Computer Vision and Image Understanding*, 73(3):428–440, 1999.

[JFLM07] Chao Jin, Thomas Fevens, Shuo Li, and Sudhir Mudur. Motion learning-based framework for unarticulated shape animation. *Visual Computer*, 23(9-11):753–761, 2007.

[JM02] Odest Chadwicke Jenkins and Maja J Mataric. Deriving action and behavior primitives from human motion data. In *IEEE/RSJ international conference on intelligent robots and systems*, volume 3, pages 2551–2556. IEEE, 2002.

[KG03] Lucas Kovar and Michael Gleicher. Flexible automatic motion blending with registration curves. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 214–224. Eurographics Association, 2003.

[KG04] Lucas Kovar and Michael Gleicher. Automated extraction and parameterization of motions in large data sets. *ACM Transactions on Graphics*, 23(3):559, 2004.

[Khr19] Khronos Group Inc. Rendering pipeline overview, 2019. [Online; accessed August 5, 2019].

[Kov04] Lucas Kovar. *Automated methods for data-driven synthesis of realistic and controllable human motion*. 2004.

[Kre16] Oliver Kreylos. Lighthouse tracking examined. 2016.

[Kru95] Philippe B Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.

[Lab02] CMU Graphics Lab. *CMU Motion Capture Database*, 2002. `http://mocap.cs.cmu.edu`.

[Lan98] Jeff Lander. Working with Motion Capture file formats. 1998.

[Las87] John Lasseter. Principles of traditional animation applied to 3d computer animation. In *ACM Siggraph Computer Graphics*, volume 21, pages 35–44. ACM, 1987.

[LCM+19] Michael C LeCompte, Scotty A Chung, Mahta Mirzaei McKee, Travis G Marshall, Bart Frizzell, Mandy Parker, A William Blackstock, and Michael K Farris. Simple and rapid creation of customized 3-dimensional printed bolus using iphone x true depth camera. *Practical radiation oncology*, 2019.

[LCR+02] Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins, and Nancy S. Pollard. Interactive control of avatars animated with human motion data. *ACM Transactions on Graphics*, 21(3), 2002.

[LK05] Manfred Lau and James J. Kuffner. Behavior planning for character animation. Number July, page 271, 2005.

[LL06] Jehee Lee and Kang Hoon Lee. Precomputing avatar behavior from human motion data. In *Graphical Models*, volume 68, pages 158–174, 2006.

[LPY16] Libin Liu, Michiel Van De Panne, and Kangkang Yin. Guided Learning of Control Graphs for Physics-Based Characters. *ACM Transactions on Graphics*, 35(3):1–14, 2016.

[Luc02] Frederic Pighin Lucas Kovar, Michael Gleicher. Motion Graphs. pages 2012–2014, 2002.

[LvdP96] Alexis Lamouret and Michiel van de Panne. Motion synthesis by example. In *Computer Animation and Simulation96*, pages 199–212. Springer, 1996.

[MC12] Jianyuan Min and Jinxiang Chai. Motion graphs++. *ACM Transactions on Graphics*, 31(6):1, 2012.

[Men11] Alberto Menache. *Motion Capture Primer*. 2011.

[MG01] Thomas B. Moeslund and Erik Granum. A survey of computer vision-based human motion capture. *Computer Vision and Image Understanding*, 81(3):231–268, 2001.

[MTT+17] Josh Merel, Yuval Tassa, Dhruva Tb, Sriram Srinivasan, Jay Lemmon, Ziyu Wang, Greg Wayne, and Nicolas Heess. Learning human behaviors from motion capture by adversarial imitation. 2017.

[NF02] Michael Neff and Eugene Fiume. Modeling tension and relaxation for computer animation. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 81–88. ACM, 2002.

[PALvdP18] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. DeepMimic. *ACM Transactions on Graphics*, 37(4):1–14, 2018.

[PAZA17] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-Real Transfer of Robotic Control with Dynamics Randomization. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8, 2017.

[PBYV17] Xue Bin Peng, Glen Berseth, Kangkang Yin, and Michiel Van De Panne. DeepLoco. *ACM Transactions on Graphics*, 36(4):1–13, 2017.

[PEG+09] Rick Parent, David S Ebert, David Gould, Markus Gross, Chris Kazmier, Charles John Lumsden, Richard Keiser, Alberto Menache, Matthias Müller, F Kenton Musgrave, et al. *Computer animation complete: all-in-one: learn motion capture, characteristic, point-based, and Maya winning techniques*. Morgan Kaufmann, 2009.

[PKM+18] Xue Bin Peng, Angjoo Kanazawa, Jitendra Malik, Pieter Abbeel, and Sergey Levine. SFV: Reinforcement Learning of Physical Skills from Videos. 37(6), 2018.

[RGBC96] Charles Rose, Brian Guenter, Bobby Bodenheimer, and Michael F. Cohen. Efficient generation of motion transitions using spacetime constraints. (April):147–154, 1996.

[Rok19] Rokoko Electronics. Rokoko motion capture library, 2019. [Online; accessed August 25, 2019].

[RS00] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[SBS02] Hedvig Sidenbladh, Michael J. Black, and Leonid Sigal. Implicit Probabilistic Models of Human Motion for Synthesis and Tracking. pages 784–800, 2002.

[Sch07] Damian Schofield. Animating and interacting with graphical evidence: Bringing courtrooms to life with virtual reconstructions. In *Computer Graphics, Imaging and Visualisation (CGIV 2007)*, pages 321–328. IEEE, 2007.

[Sho85] Ken Shoemake. Animating rotation with quaternion curves. In *ACM SIGGRAPH computer graphics*, volume 19, pages 245–254. ACM, 1985.

[TDSL00] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[Vic16] Vicon. *Vicon Vantage Reference Manual*, 2016. `https://www.prophysics.ch/wp-content/uploads/2017/06/ViconVantageReference.pdf`.

[WCW14] Xin Wang, Qiudi Chen, and Wanliang Wang. 3D human motion editing and synthesis: A survey. *Computational and Mathematical Methods in Medicine*, 2014, 2014.

[WFH08] Jack M. Wang, David J. Fleet, and Aaron Hertzmann. Gaussian process dynamical models for human motion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):283–298, 2008.

[WGF07] Rachel Weinstein, Eran Guendelman, and Ronald Fedkiw. Impulse-based control of joints and muscles. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):37–46, 2007.

[Wil87] Jane Wilhelms. Using dynamic analysis for realistic animation of articulated bodies. *IEEE Computer Graphics and Applications*, 7(6):12–27, 1987.

[WMC11] Xiaolin Wei, Jianyuan Min, and Jinxiang Chai. Physically valid statistical models for human motion generation. *ACM Transactions on Graphics (TOG)*, 30(3):19, 2011.

[WP95] Andrew Witkin and Zoran Popovic. Motion Warping. *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques - SIGGRAPH '95*, pages 105–108, 1995.

[Xse19] Xsens Technologies B.V. Xsens mvn link motion capture suit, 2019. [Online; accessed August 26, 2019].

[YL10] Yuting Ye and C. Karen Liu. Synthesis of responsive motion using a dynamic model. *Computer Graphics Forum*, 29(2):555–562, 2010.

[YLvdP07] KangKang Yin, Kevin Loken, and Michiel van de Panne. SIMBICON. *ACM Transactions on Graphics*, 26(3):105, 2007.

[Zha12] Zhengyou Zhang. Microsoft kinect sensor and its effect. *IEEE multimedia*, 19(2):4–10, 2012.

[ZMCF05] Victor Brian Zordan, Anna Majkowska, Bill Chiu, and Matthew Fast. Dynamic response for motion capture animation. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 697–701. ACM, 2005.

[ZN03] Tao Zhao and Ramakant Nevatia. Bayesian human segmentation in crowded situations. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 2, pages II–459. IEEE, 2003.

Appendix A

Additional results

A.1 Motion blending

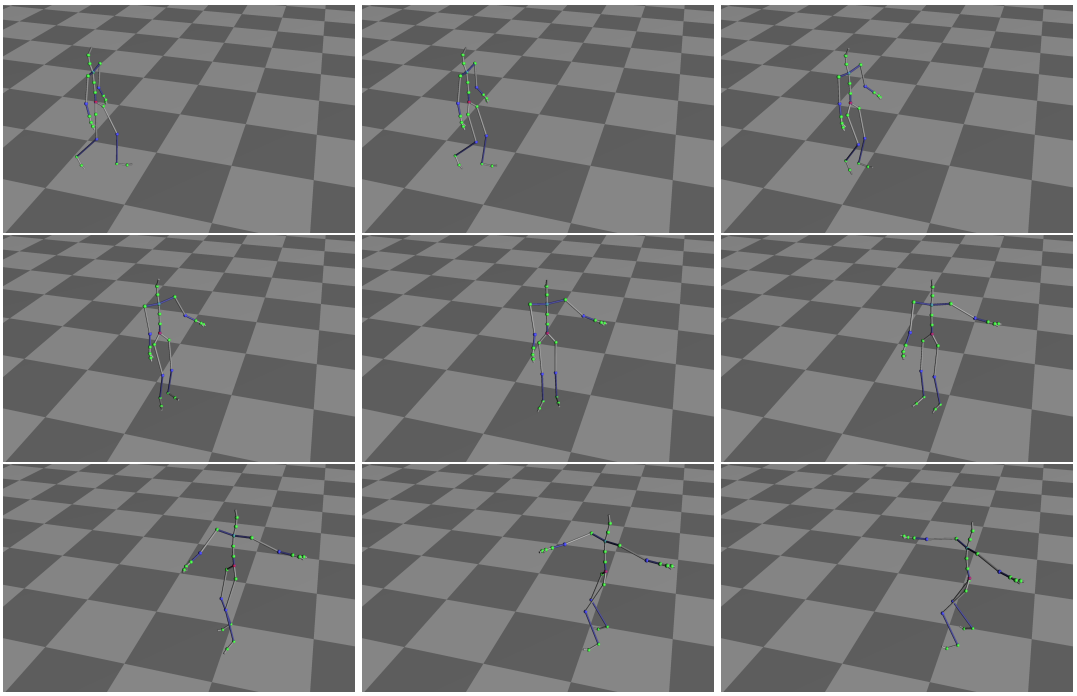
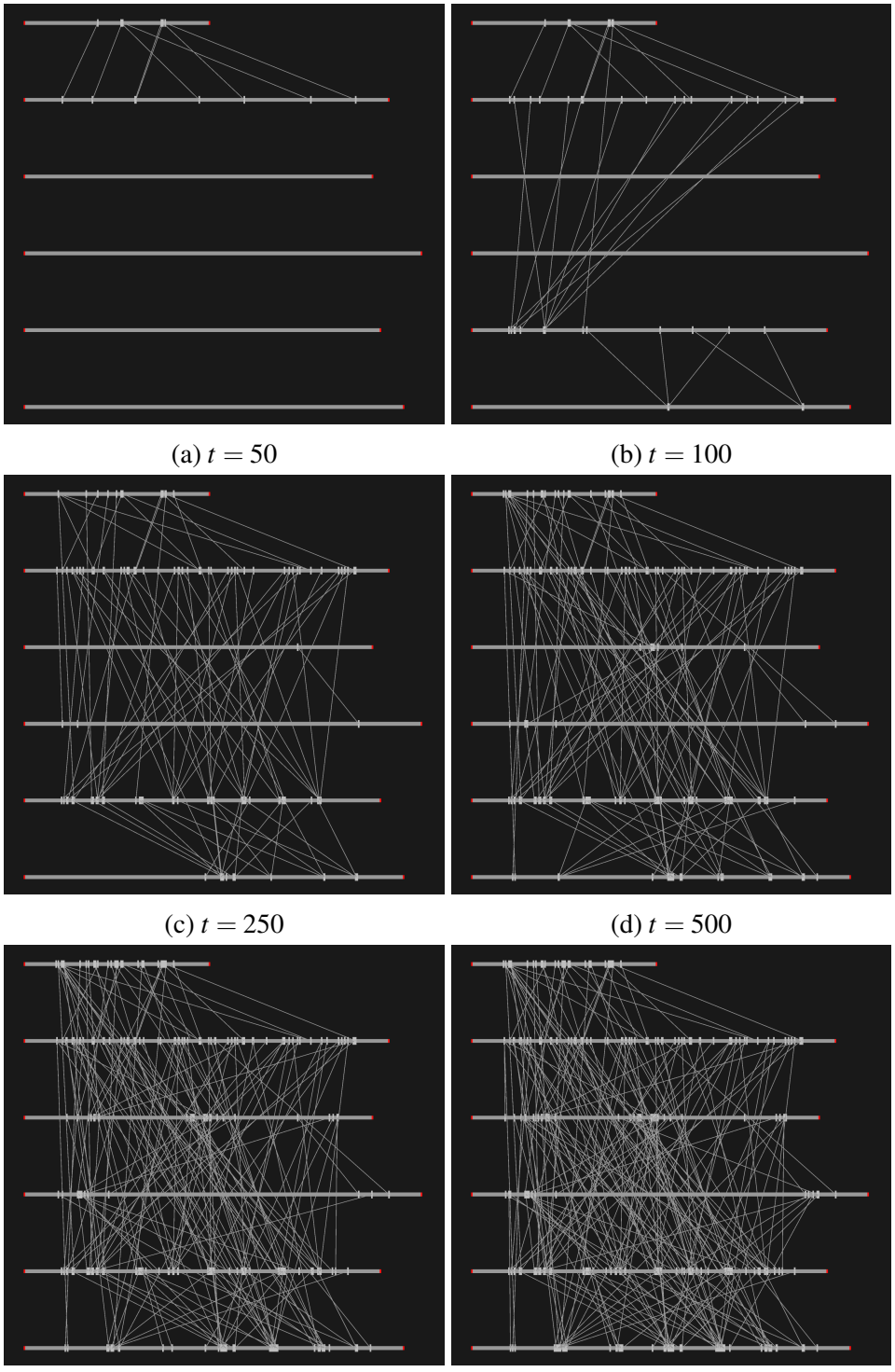


Figure A.1: Result of blending a walk motion and a pirouette. The animation is depicted using snapshots of 10 frames. The distance of the transition is 4682.45. Clear sliding issue are visible.

A.2 Motion graph



(a) $t = 50$ (b) $t = 100$

(c) $t = 250$ (d) $t = 500$

(e) $t = 1000$ (f) $t = 250000$

Figure A.2: Motion graph with various threshold values.

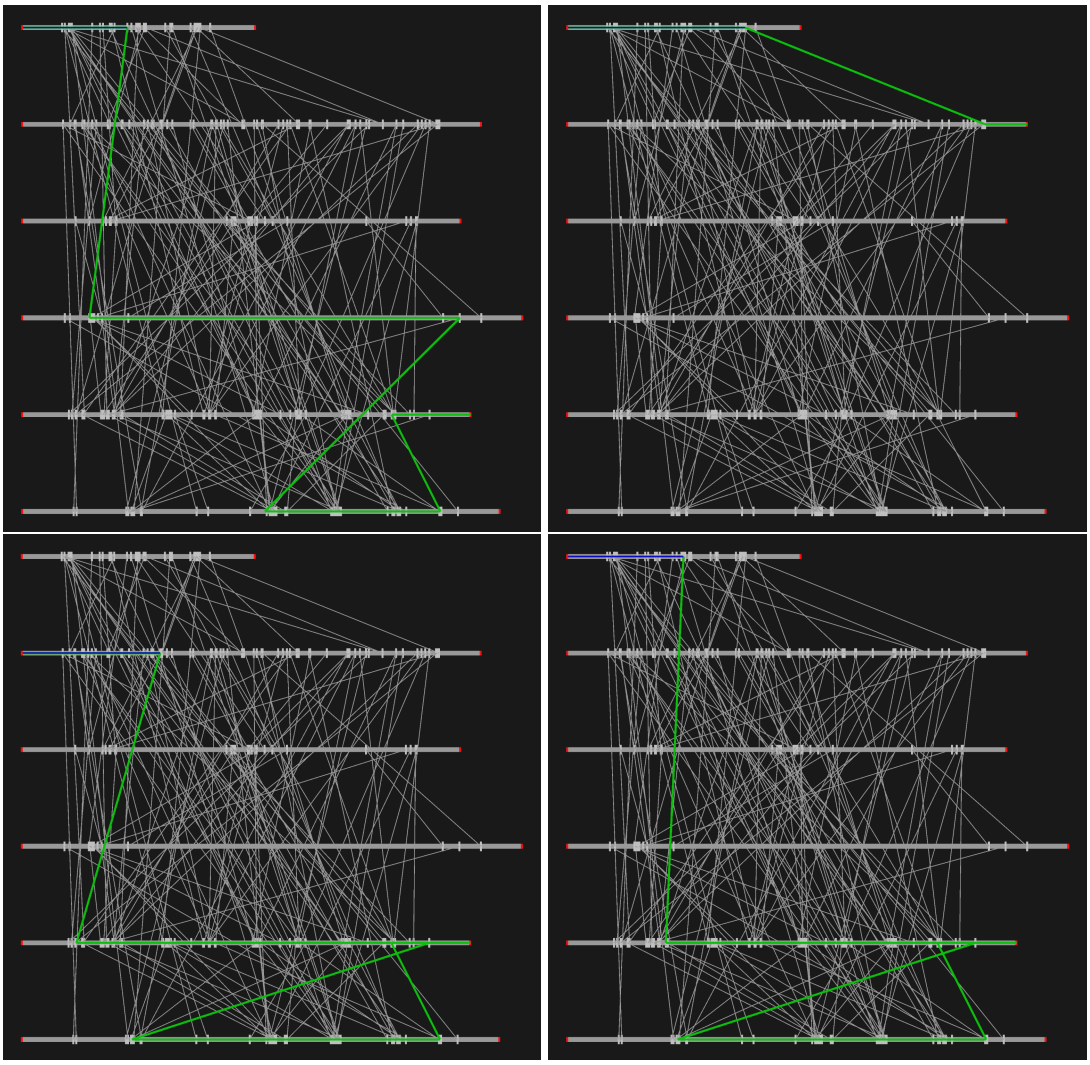


Figure A.3: Alternative random graph paths on the motion graph.

Appendix B

User interface

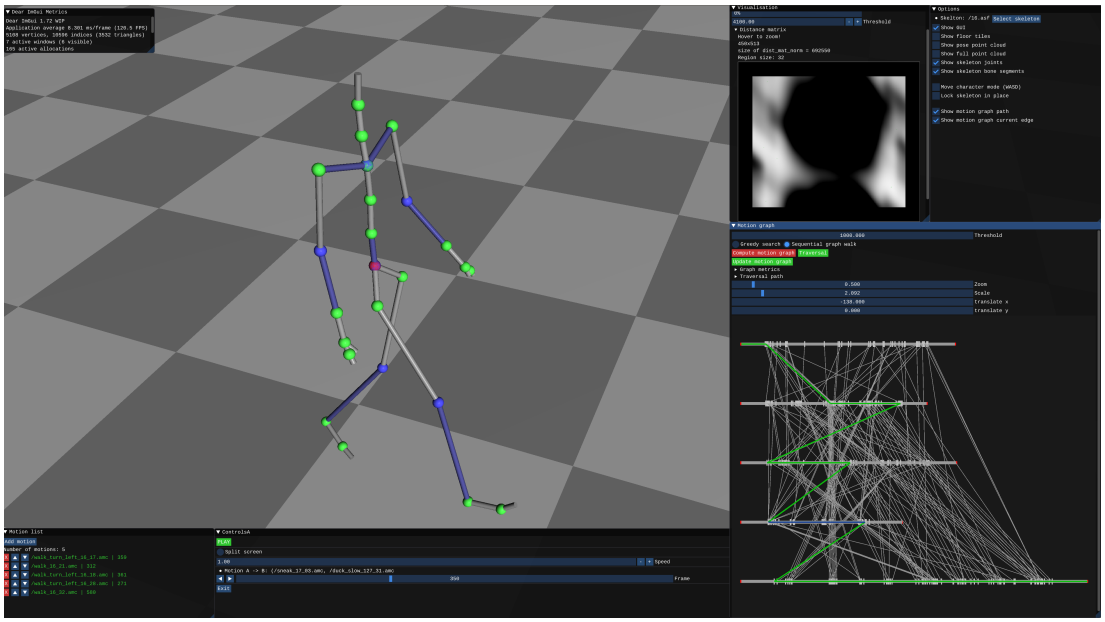


Figure B.1: A screenshot of the project's GUI used to interact with the application.

Appendix C

Github Repository

The code base of the project can be accessed at the following url:

`https://github.com/SudoHead/motion-graphs`