Synthesizing Realistic Human Motions Using Motion Graphs

by

Ling Mao, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2011

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Ling Mao

August 29, 2011

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Ling Mao

August 29, 2011

Acknowledgments

Foremost I would like to express my gratitude to my supervisor Dr. Rozenn Dahyot, for her advice, help, good humour and above all patience. I would also thank our course coordinator John Dingliana for some useful advice. Finally, many thanks to my family and friends for supporting me while doing this work.

The data used in this project was obtained from mocap.cs.cmu.edu. The database was created with funding from NSF EIA-0196217.

Ling Mao

University of Dublin, Trinity College
September 2011

Synthesizing Realistic Human Motions Using Motion Graphs

Ling Mao
University of Dublin, Trinity College, 2011

Supervisor: Rozenn Dahyot

Animating realistic human motions plays an important role in the game and movie industries. Motion capture provides a reliable way for acquiring realistic motions. However, motion capture data has proven to be difficult to modify, which limits the utility of motion capture in practice. We propose to investigate ways for making the motion capture data reusable so that new motions can be generated.

This paper provides a motion graph based system for synthesizing realistic motions. The motion graph is a directed graph where edges are pieces of motions and nodes are points for joining edges. First of all, various motions are chosen from the motion capture database as the input. Then a motion graph is constructed by splitting original motion streams and building transitions between similar poses. Similar poses are found by evaluating similarity metric. Transitions, which are also motion clips, are made by linearly interpolating original motion pieces. Finally, a branch and bound search

algorithm is applied to extract an optimal "*GraphWalk*" satisfying user requirements.

We show the practical application of this system to path synthesis, which is to generate new motions along paths. Also, we show the interactive control over the motions with the help of graphical user interface, including changing motions between different types, drawing paths and setting directions for the character to follow.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Character animation has gained its weight in today's game and movie industries, among which human animation is the most challenging task because of its complex motions and emotions. From the primitive key-framing animation, developers have introduced many ways trying to create realistic human motions. Recently, motion capture has proven to be a reliable way to accomplish this task. With multiple sensors attached at certain places on an actual human body, it captures the real motion data and then translates the data onto a digital model for reconstruction. Motion capture is now popularly used by game developers and movie producers.

However, motion capture is limited in more common applications, as the data has proven to be difficult to modify and editing techniques only apply small reliable changes to a motion. Imagine that if the motion capture database can not provide sufficient similar motions required by users, then there is no choice but to capture more data. The motion capture is a complex and time consuming process, and its devices may not be available. Motion synthesis is then developed to make the motion capture data reusable by generating new motions. Also it enables the dynamic synthesis of motions and the interactive control of three-dimensional avatars. Due to these features, motion synthesis is now a popular technique for human animation in game and movie industries.

One goal of motion synthesis is to generate new motions desired in practical applications while retaining the realism of motion capture. Motion graphs built from motion capture data have emerged as a promising technique, in that it can generate

long motions while preserving the original motion data as much as possible.

The motion graph is a directed graph that contains edges and nodes. Edges present motion clips both from original motion data and transitions that are automatically created by linearly interpolating original similar motions. Nodes are the points joining relative edges. This simple graph structure allows editing original motion streams by splitting them, inserting nodes, and then creating transitions between the newly inserted nodes. In order to generate new motions, a *GraphWalk* is performed by traversing the graph, appending one edge after another. Therefore, motions can be as long as possible, and also can be whatever we want as long as sufficient original data is provided.

To generate good quality motions, the motion graphs should be with good quality, which indicates good connectivity and smooth transitions. A well connected motion graph requires as many transitions as possible, to ensure more choices when searching the graph for desired motions. Nonetheless, keeping the smoothness of transitions may exclude certain number of potential transitions. Balancing these two criteria has proven to be difficult according to current research.

1.1 Goal

The objective of our project is to synthesize realistic human motions using motion graphs. Mainly, we try to accomplish three tasks. The first is to obtain the original motion data from the database which is stored in BVH files. The *.bvh* is a file format that records the skeleton hierarchy information and the motion data. It represents a skeletal human body and a motion clip with certain number of frames. Thus certain processing including parsing the BVH file and interpreting its data is required.

Second comes to the construction of good quality motion graphs. To detect candidate transitions, we use similarity metric combined with the idea of point cloud window, which takes into account both the static posture difference and motion dynamics difference. We also takes different thresholds in similarity metric when dealing with some different motion types. We apply common linear interpolation and spherical linear interpolation to keep the smoothness of transitions. Finally, the motion graph is pruned to be a strongly connected component.

The final task requires a graph search policy to search motions through the graph

that satisfy user requirements. We use efficient branch and bound search method generally. Also a quick step-first search is embedded to ensure quick transitions between different motion types.

With graphical user interface, we intend to apply our framework to path synthesis, that is generating motions following the paths. We implement user interactive control over motions by drawing paths, setting directions and choosing different motion types to perform.

1.2 Document Outline

The structure of this document is as follows:

- Chapter 2 reviews the related work in the area of motion synthesis, as well as character animation.
- Chapter 3 provides an overview of the framework design, including system tasks and graphical user interface.
- Chapter 4 describes the system implementations in details.
- Chapter 5 presents the result motions generated from our framework, and evaluates the performance.
- Chapter 6 draws an conclusion of the current project and set directions for future exploration.

Chapter 2

State of the Art

This chapter gives an overview of the related research. Section 2.1 introduces briefly different methods to create character animations. Section 2.2 illustrates the three main techniques for motion synthesis. Section 2.3 focuses on the techniques of motion graph based synthesis, and gives detailed published algorithms aiming at constructing good motion graphs for practical use. Section 2.4 follows section 2.3, and further introduces the methods of motion search through the motion graph for generating final motions. Section 2.5 gives an overview of the software related in character animation. Section 2.6 draws a summary of state of the art and gives a brief introduction of techniques in this paper.

2.1 Character Animation

In computer animation, a human body is usually treated as a rigid skeleton that drives a deformable skin. To create an animated character, one has to model the skeleton geometry and create motion specifications for character actions [1, 2, 3]. The motion specifications are the translation values of the root joint of the skeleton and the rotation values of all other joints over time. Generally, there are four methods to create animations: key-frame animation, procedural animation, behavioural animation and motion capture.

Key-frame animation is a traditional method, where key poses are first specified and then in-between motions are generated by interpolating the key frames. Computer

techniques automate the second process using spline interpolation [4, 5, 6, 7]. This method gives artists the ultimate control over every detail of the motion. However, it is one of the most time consuming methods, considering the production of key frames that yield convincing physics and achieve desired effects.

Procedural animation method produces animation by running a physically based simulation of the entire system. Compared with key-frame animation, less degree of control is allowed, but less time consumes and higher degree of realism can be achieved. There are two main approaches proposed: kinematics simulation and dynamics simulation. Kinematics directly manipulates body joints and segments without considering its mass and forces acting on it. This approach mainly relies on bio-mechanical knowledge and combines forward and inverse kinematics for computing motions [8, 9, 10]. Unlike kinematics, dynamics studies the motion by applying the forces to the body with mass. One popular way is controller-based dynamics that uses controllers to control the joints during simulation [11, 12]. While this control-based approach require a user to specify a large number of control parameters, which resulting a difficult task, the space-time constrained optimization approaches stand out [13], by optimizing a physically-based objective function to yield motions satisfying physics constraints and user specifications.

In behavioral animation, the animation is generated by modeling and simulating the mental process of character. The behavior is specified in terms of intents and goals. Goals are the results of animation, while intents are in turn driven by these goals. In order to satisfy intents, the character needs a motion model to control its own locomotion. Additionally, the character uses a virtual sensor to sense the environment and the result of its own actions, thus setting up a feedback control loop. The most popular application of behavioral animation is animating groups [14], which is to specify individual member to animate a group.

Motion capture has been recently developed and popularly used for generating animations. It is a process of recording motion data directly from live performers for an approximate skeletal hierarchy and then translating the data onto a digital model for character reconstruction. Thus, the animation generated by this method is extremely realistic, as probably some motion details can be acquired. However, it consumes effort to capture the data considering the availability of devices, the complexity of capture process and the quality of performers' actions. So further processing has developed

into a research area for reusing the motion capture database to generate new motions, called motion synthesis.

In summary, different method applies to different requirement. Key-framing enables animators to artistically control over characters. Procedural animation produces physically realistic characters quickly. Behavioral animation is used to animating group behaviors regardless of high precision of an individual character. Motion capture is specially used when physically realistic motions are strictly desired and the motions can be acted out by skilled performers. Besides, motion capture database can also be reused.

2.2 Motion Capture Based Synthesis Techniques

As mentioned before, motion capture to generate motions is largely limited by the availability of devices, the complexity of capture process and the quality of performers' actions. To make it widely applicable, the motion capture data needs to be made re-usable. Thus, motion synthesis is developed to generate new motions with existing motion capture database, which also enables the control over motions to meet the user specifications. Generally, motion synthesis involves three stages: obtaining user requirements, searching database and generating motions. Research work in motion synthesis can be divided into three categories: multi-target interpolation based synthesis, model construction based synthesis and motion graphs based synthesis.

2.2.1 Multi-target Interpolation Based Synthesis

Multi-target interpolation based synthesis refers to the interpolation among a set of example motions. Bruderlin and Williams [15] introduced this method with dynamic time-warping to blend between motions, and displacement mappings to alter motions such as grasps. In addition, multi-resolution filtering method is applied to generate variations of a motion [15], by separating frequency bands of a motion and then adjusting gains of the different bands. For example, increasing the gain of the high frequency band produces jittery motions while increasing the gain of low frequency band creates exaggerated motion. In another way, Perlin [16, 17] replaced high frequency components with random noise and used noise functions to simulate personality and

emotion in existing animation.

Motions can be parameterized. Rose et al. [18]. They proposed a verbs/adverbs system to realize motion parameterizations. Verbs are used to represent parameterized motions, and adverbs are used for the parameters that control verbs. To create new verbs, interpolation is applied among similar motions with the use of radial basis functions to generate variations. Then the verbs are combined with other verbs to form a verb graph by linearly blending smooth transitions between them, resulting in a substantial repertoire of expressive behaviors. Besides, this system requires inverse kinematics constraint key-frames as input. This parameterization system allows convenient modifications of motions in real time and enables interactive control over motions.

Desired motions can be linearly interpolated directly from a subset of examples [19]. However, to increase the precision of the selection of data subset, Wiley and Hahn [19] resampled the motion data to a regular rectangular or cylindrical grid of the parameter space, and new motions can then be synthesized by traversing the sampled grid in the desired order and interpolating between motion samples corresponding to these grid points.

As mentioned above, the quality of motion interpolated largely depends on the example motions chosen, so one prerequisite of the method is synchronization of motions, which is that the motions should be aligned in global position and rotation, they should have similar cadence and length and they should have similar constraint frames. Early effort used manual input which is not efficient. Kovar and Gleicher [20] proposed a novel data structure, named registration curve, which encapsulates the relationships involving the time, local coordinate frame and constraint states of an arbitrary number of input motions. This method can expand the class of motions that can be successfully blended automatically.

2.2.2 Model Construction Based Synthesis

Another popular approach to motion synthesis is to construct statistical models. The inspiration of this method is the fact that there are correlations among numerous features of the data. For example, a point characterized by a particular time and frequency band will depend upon points close to it in time in other frequency bands.

Pullen and Bregler [24] modeled the correlations with a kernel-based representation of the joint probability distributions of the features. Then the synthesis is initiated by sampling randomly from the probability distributions and completed by an iterative technique to maximize the probability of occurrence of each value.

State machine learning is a prominent approach in statistical model based motion synthesis. Brand and Hertzmann [25] demonstrated an HMM (hidden Markov model) derived scheme, called style HMM (SHMM), to learn and generate style variations in motions. The motions captured data here is processed by constructing abstract states which each represent entire sets of poses. They introduced a cross-entropy optimization framework that makes it possible to learn SHMM from a sparse sampling of unlabeled style examples. The generated SHMM is used to synthesize the final motion.

The combination of PCA (principle component analysis) and K-means clustering can also be contributed to motion synthesis to construct local linear models. Hodgins et al [26] preprocessed the motion database by splitting it into smaller motions representing low dimensional data, then converting the segments to high dimensionality, and reducing them by PCA and clustering using K-means. Thus the database becomes a hierarchy of clusters of motions segments which can be represented as local linear models. Proper local model is then found to be used for generating a motion.

2.2.3 Motion Graphs Based Synthesis

Motion graph is a newly developed method and has emerged as a very promising technique for automatic motion synthesis for both interactive control applications [27] and for off-line sketch-based motion synthesis [27, 21, 29, 30, 31]. A motion graph is constructed by extracting a set of motion clips from database according to user requirements and then building transitions between all similar motion frames. Finally, new motions are created by a traversal in the graph. The motion graph is constructed with nodes, representing motion clips, and edges, representing transitions. The inspiration of constructing motion graphs came from a simple idea to synthesize motions by cutting and pasting parts from various motion clips in database together. Thus, this method can preserve original motion as much as possible, which means details from captured motion can be less probably lost. In addition, it is an intuitive method for novice to start the research in motion synthesis.

2.3 Motion Graphs

Considering the simple graph structure, ability to generate long motions while preserving as much original motion as possible, and fully automatic solution to motion synthesis problems, motion graph has been popularly developed and used by researchers, especially by novices. However, there are two fundamental issues that should be taken into considerations. One is that to generate good quality of motion, the motion graph should be well connected and the transitions built should be smooth. But actually achieving both criteria simultaneously is difficult, because good connectivity requires transitions between less similar poses from different behaviors, while transitions between very similar poses from similar behaviors will result in new motions with good quality. The other is the selection of motions for constructing motion graph. In order to construct a motion graph, a subset of motions is selected from large database according to real application. On one hand, a small size of motion graph constructed is easier for fast search for desired motions. On the other hand, the motion graph needs to contain enough data to ensure good quality of transitions and to satisfy user requirements. Based on these issues, researchers have developed several solutions.

2.3.1 Motion Graph Construction

The standard motion graph [21] used similarity metric to find similar poses as candidate transitions and then linear blends to create smooth transitions. The similarity metric is represented by distance function, which means the smaller the value of distance function is, the more similar the poses are. Most distance functions are based on Euclidean distances [27, 21]. While the use of linear blends may violate the constraints in original motion, Kovar and Gleicher [21] correct this by using constraint annotations in the original motions.

Blending techniques are popular for creating smooth transitions [18, 27, 21, 33]. However, some issues should be taken into account: the alignment of motions, the length of blending window and the value of blending weights. Another way to create quick transitions is proposed by Ikemoto et al [32]. They first split the motions into short segments which are then clustered into groups based on similarity. Then similar motion segments are found, after which the most natural transitions are searched among all possible interpolations of these segments. Here, discriminative classifiers are defined

to distinguish nature and unnatural motions.

Move trees is a variant of the stand motion graph, also called structured motion graph, which is commonly used in game industries to generate human motion [34, 35]. Unlike standard motion graph, move trees uses FSM (finite state machine) to represent all the motion clips as behaviors and define the logic transitions between the behaviors. A search tree of states is built in the FSM. Motions are generated automatically by a planning algorithm that performs a global search of the FSM for good transitions and thus a sequence of behaviors that meet user requirements. The move trees can produce motions with controlled motion quality and transition time in that the good design of FSM allows all potential transitions needed for new motions, and captured motion clips are optimized with manual work to ensure the transition quality. Thus this technique is well-suited for real-time games.

There have developed several extensions for motion graph construction cooperating with other techniques such as interpolations. The interpolated motion graph (IMG) proposed by Safonova and Hodgins [31] is created by interpolating all possible poses with matching contact patterns in the standard motion graph (MG). And the final motion is created by searching the IMG for an optimal path that satisfies user specifications. The basic idea of IMG came from their intention to create synthesized motions by an interpolation two time-scaled paths through a motions graph. The graph created is much better with smaller search space and high quality, because it only contains natural poses and velocities from the original motions and the interpolation of segments with matching contact patterns. However, the quality of IMG still depends on that of MG, because only when MG contains quick and smooth transitions can IMG has good transitions.

Safonova and Zhao [37] continued the work in optimizing motion graph, intending to balance between graph connectivity and smooth transitions, when they developed a well-connected motion graph (wcMG). Different from IMG, the wcMG directly add quick and smooth transitions to MG, thus it uses exactly the same representation as a MG. They achieved this by computing a set of interpolated poses from the original poses before creating the graph from both original poses and a set of interpolated poses. The pre-processed interpolated poses ensure the smooth transitions in motion graph, while MG may have plausible transitions in that similar poses found not similar enough to create transitions. Furthermore, a set of motions can also be computed

in the pre-processing with different interpolation weights that will contain many more similar poses allowed for transitions. Finally, the set of interpolated poses is reduced to a subset by minimizing the interpolated poses number while guaranteeing the physical correctness of the transitions and the connectivity of the original poses. The wcMG in practice outperform MG in that it achieves better connectivity and smoother motions, thus leading to more realistic visualization, and it requires no post-processing, which saves time and effort.

Another extension of motion graph is called parametric motion graph introduced by Heck and Gleicher [36]. The construction of the parametric motion graph (PMG) differs from the standard motion graph mainly in the following two ways. While the nodes in MG represent the motion clips from database, the nodes in PMG represent entire parametric motion spaces that produce short motions through blending-based parametric synthesis. The blending-based parametric synthesis generates any motion accurately from an entire space of motions by blending together the examples from that space. In addition, while the edges in MG are built as transitions between individual motion clips, the edges in PMG encode valid transitions between source and destination parameterized motion spaces. Thus the edges encode the range of parameters of the target space that a motion from the source space can transition to, and correct transitions between valid pairs of the source and destination motions, which is solved by a sampling based method. Based on the structure, PMG can efficient organize large number of example motions that can be blended to produce final motion streams, and allow interactive authoring controllable characters.

2.3.2 Selection of Motions

The other problem to be solved in motion graph construction mentioned above is the selection of motions. In comparison, it has been found less solutions. Early effort was mostly devoted to manual selection, which is manually selecting a subset of motions from large database based on user requirements to construct motions graphs. However, this has proved to be difficult for synthesizing a special motion and obey the automatic concept of computing. A dynamic approach was proposed by Cooper et al [39] that it uses active learning to identify which motion sequence the user should perform next during capturing sessions, and then updates a kinematic character controller after

each new motion clip is acquired. This active learning method helps automatically identify specific tasks that the character controller may perform poorly, and thus avoid the difficulty of manually determinations of which motions to capture. However, this system does not make formal guarantees of being able to perform every task from every start state. Some tasks may be impossible while others may have not been captured.

Kovar and Gleicher [22] introduced a method to automatically extract logically similar motion segments from a large database and use them to construct a continuous parameterized interpolation motion space. New motions are created for new parameter values inside the space. To find the logically similar motions that are numerically dissimilar, their search method employs a novel similarity metric to find close motions and then uses them as intermediaries to find more distant motions. The novel distance metric complement the standard one in that it analyzes the time correspondences of the frames themselves. Whats more, the search method is not started from scratch each time; instead a match web is pre-computed, which is a compact and efficiently searchable representation of all possibly similar motion segments. Unfortunately, while their work can generate motions for a particular behavior very well, it does not scale to multiple distinct behaviors. And it doesnt make guarantees to satisfy user requirements.

To make motion graph more practically used, the selection of a subset of motions needs to be simple and fully automatic for users, especially novices. Safonova et al [38] cast the selection problem as a search for a minimum size sub-graph from a large motion graph representing the motion capture database and propose an efficient algorithm, called the Iterative Sub-graph Algorithm (ISA), to find a good approximation to the optimal solution. They first select a small set of key motions that are required in real applications but fail to generate a good motion graph because of lacking quick and smooth transitions. Then additional motions in the motion database are found that result in good connectivity between the key motions, while the size of the motion graph is kept as small as possible. As mentioned in their work, this step corresponds to a well-defined theoretical problem of finding a minimum size sub-graph from a very large graph, which is a NP-hard problem but optimized approximately by employing ISA in their work.

2.4 Motion Search

After the motion graph is constructed, the next step is to search for the motions required by applications. Generally, there are two main ways to solve this problem: off-line search and on-line search. Existed on-line search methods are implemented with reinforcement learning.

2.4.1 Off-line Motion Search

The off-line motion synthesis takes the predefined user requirements and searches a motion graph with global search techniques for a motion meeting user requirements. The quality of result motions also depends on the searching algorithms, and the efficiency of the algorithm influences the efficiency of motion synthesis. Several algorithms have been developed.

An early algorithm developed by Kovar et al [21] uses branch and bound to drive the character to follow a sketched path. They cast the motion search problem to finding an optimal *GraphWalk* from all possible *GraphWalks*. The *GraphWalk* is a motion generated by placing edges in the graph one after another. The branch and bound method can make this optimization process more efficient in that it reduces the number of *GraphWalks*. However, it does not change the fact that the search process is inherently exponential. Arikan et al [29] proposed another search method, based on dynamic programming at several scales, to search for a motion with user specified annotations.

A searching method, called A* algorithm, has been developed by Safonova and Hodgins [31] as an efficient way for optimal or near-optimal solutions. For globally optimal search, they developed two techniques that minimize the effort of search. The first technique compresses the motion graph into a practically equivalent but much smaller graph, by removing states and transitions that would not be part of an optimal solution. The second computes an informative heuristic function that guides the search towards states that more likely appear in an optimal solution.

2.4.2 Reinforcement Learning

Unlike off-line search method taking predefined user specification, on-line search method takes user input in real time. In Lee's work [28], a pre-computation is provided to define how an avatar moves for each control input and the avatar's current state. During run-time, their framework allows interactive control over motions for a set of control inputs with least time cost. McCann and Pollard [41] introduced a model of user behaviours integrated for better immediate control.

2.5 Software Overview

With the interest in developing 3D character animation, there have emerged multiple software tools and APIs. Tools like Autodesk 3ds Max, Autodesk Maya, Autodesk MotionBuilder and Blender provide powerful 3D modelling, animation, rendering and compositing utilities. APIs typically like OpenGL and Direct3D, provide many commands for 3D rendering. Lately, a new rendering engine called OGRE (objected-oriented graphics rendering engine) brings a general solution for graphic rendering.

OGRE is written in C++ and supports APIs like OpenGL and Direct3D and operating systems like Windows, Linux and Mac OS X. It is designed to minimize the effort required for rendering 3D scenes, thus automatically accomplishes common requirements like rendering the state management, spatial culling and dealing with transparency. The main features of OGRE can be illustrated as follows:

- Support vertex and fragment programs along with custom shaders written in Cg, DirectX9 HLSL, GLSL and assembler.
- Support multiple material techniques.
- Support texture files: PNG, JPEG, TGA, BMP, and DDS.
- Support flexible mesh data formats. Separate the concept of vertex buffers, index buffer, vertex declarations and buffer mappings. Scene manager has support for Progressive meshes (LOD) that is automatically or manually generated.
- Support sophisticated skeletal animation and flexible shape animation.

- Scene graph based engine where nodes allow objects to be attached to each other and follow each others movement. Support a wide variety of scene managers like Octree, BSP.
- Has multiple shadow rendering techniques, both modulated and additive, stencil and texture based.
- Has a compositing manager with a scripting language and full screen post-processing for special effects such as HDR, blooming, saturation, brightness, blurring and noise.
- Has a particle system with extensible rendering and customizable effectors and emitters.
- Has content exporter tools for most 3D modelers including 3ds Max, Maya, Blender, LightWave, Milkshape, Sketchup and more.

There also have developed multiple graphical user interface libraries, among which CEGUI is popularly used by game developers. Written in C++, it provides windowing and widgets for graphics APIs or engines where such functionality is not natively available. Always, OGRE and CEGUI work together, and perform strong corporation in animation applications.

2.6 Summary

The technique of motion capture to create character animation has stood out among various techniques, as it can produce realistic animation reliably. However, it is limited by the incapability of modifying motion capture data, the availability of capturing devices, and its time consuming capture work. Therefore, motion synthesis has been developed to reuse the motion capture data for creating new motions. The most challenge of motion synthesis is how to create the motions as much realistic as possible. Many different algorithms have been developed trying to create realistic motions while keeping flexible interactive control over the motions, and they can be generally divided into three categories: multi-target interpolation based synthesis, model construction

based synthesis, and motion graph based synthesis. Motion graph has been popularly used by researchers, especially by novices, because of the simple graph structure and the ability to generate long motions while preserving as much original motion as possible.

Based on the current research work in motion graphs, a framework is provided in this paper, trying to construct good quality motion graphs and efficiently search motions through the graph that meet user specifications. We take the classical motion graph construction method introduced by Kovar [21], and implement branch and bound search over the graph for an optimal solution. The optimization criteria is to approach the path travelled by the character to the path specified by users as much as possible. Finally, we intend to generate long motions that can follow the paths and change between different motion types.

We use OGRE and CEGUI to help our graphical user interface (GUI) design. OGRE mainly contributes to rendering the scene and CEGUI enables flexible user interface design. We design the GUI to enforce interactive control over motions and visualize the results.

Chapter 3

Framework Design

This chapter gives an overview of the framework design, including the requirements, system tasks, and a graphical user interface designed for the convenient interactive control and visualization of our motion synthesis application.

3.1 System Tasks

Generally, there are three main modules in the system implementation: loading original motion data, constructing motion graphs, and searching motions. In details, they can be divided into six steps as follows:

1. Load original motions in BVH files from motion capture database, according to motion loading configurations. The configuration tells which BVH files are needed and what frames of the motion are loaded. It also defines the descriptive information with certain labels to describe what the motion type is, like {*run*, *normal*}, {*walk*, *fast*}, etc.

2. Construct the motion graph by splitting original motion streams and creating transitions between similar poses.

3. Prune the motion graph, and then extract sub graphs. Each sub graph presents one motion type.

4. Set user controls interactively with graphical user interface.

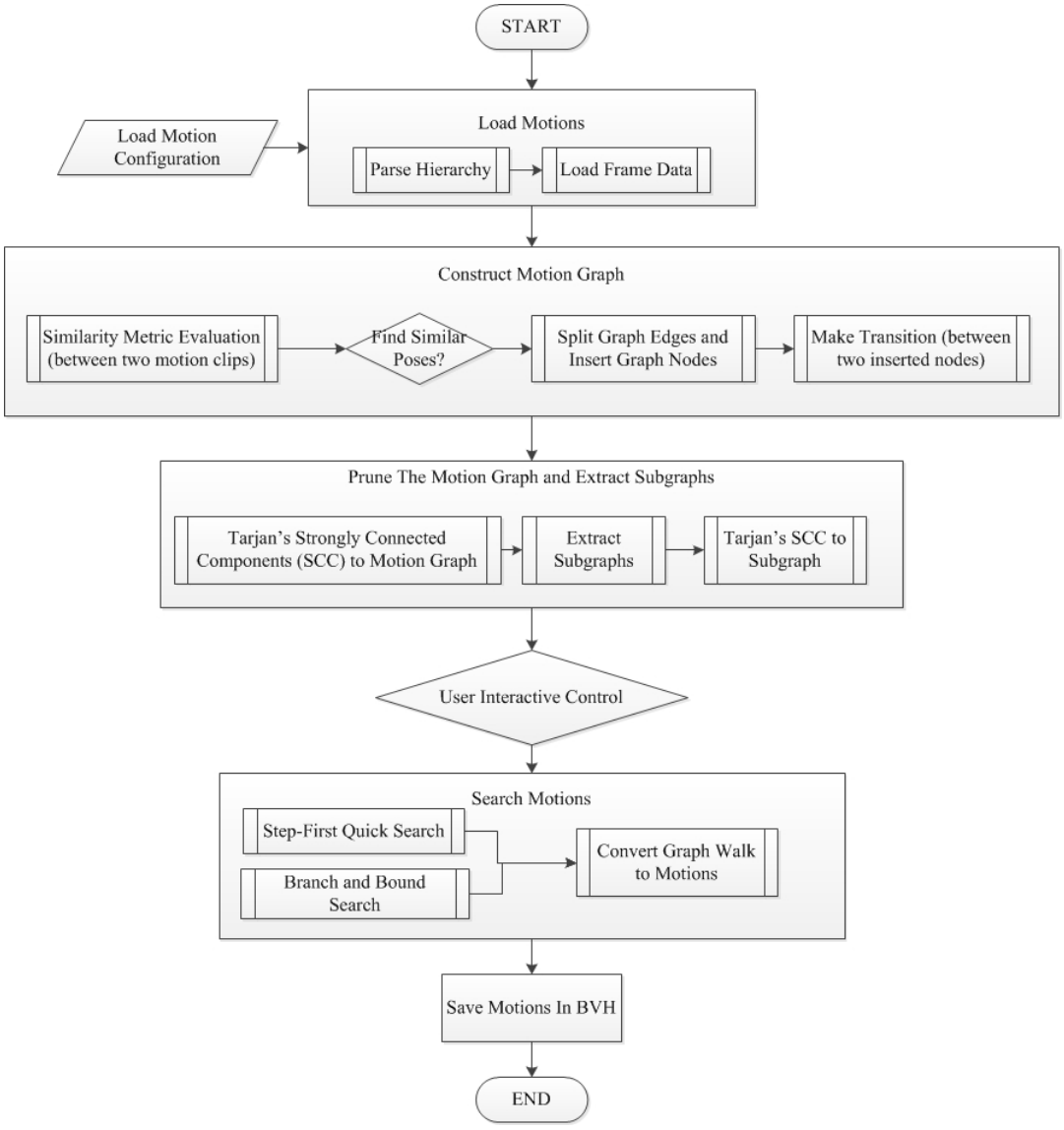

Figure 3.1: System Design.

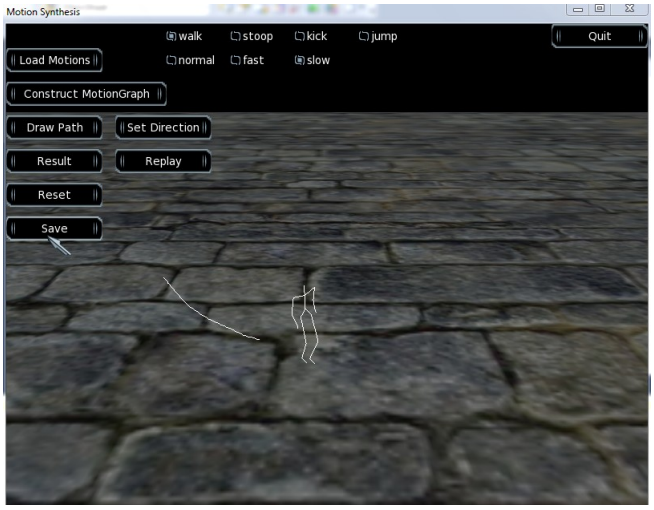

Figure 3.2: Rendering Window.

5. Search motions by traversing the motion graphs.

6. Save generated motions in BVH files.

Each step contains several sub-processes. The next chapter will describe how to implement each task in details. Figure 3.1 presents the system tasks in a flow chart.

3.2 Graphical User Interface

In addition, we need a rendering window to display the motions performed by a character for evaluation. Also, in the window, users should be able to set interactive controls over the motions. Therefore, we design a graphical user interface with CEGUI in Ogre. As mentioned in Section 2.5, Ogre is a powerful rendering engine and CEGUI is a popular GUI library, and their combination work makes our rendering window attractive. Figure 3.2 shows a screen-shot of our rendering window.

The scene features are rendered using Ogre. The widgets, including push buttons and check boxes, are designed with CEGUI.

- **Push button**. It will be fired under clicked, and the event attached to it then will be executed. Push button is a useful widget that can help users control the system tasks in real time. We put several push buttons here to set the tasks including:

- "Load Motions": load motions from motion capture database. Execute step 1 in above system tasks.
- "Construct MotionGraph": construct motion graphs from the loaded motion data. Step 2 and step 3 will be accomplished.
- "Draw Path": draw paths by dragging the mouse. It requires the character to move following the path.
- "Set Direction": set a direction by planting a point on the ground. The direction is from where the character is to the point planted. It requires the character to move towards the direction.
- "Result": begin the motion search, and then display the current result motion once the search is finished.
- "Relay": replay the motion. Because the motion may be generated step by step with several user controls input in the middle, we can use this button to replay it from the very start.
- "Reset": reset the scene by removing all paths and direction points. Also the current character stops moving and all the motions generated before are deleted.
- "Save": Save the motion generated currently in a BVH File.
- "Quit": stop running and quit the rendering window.

- **Check box**. It is fired when got checked or unchecked from user input. Each box represents one descriptive label. Users choose the type of motion to be generated by clicking to check the corresponding boxes.

Chapter 4

Implementation

This chapter provides a detailed explanation of how we accomplish the tasks described in the last chapter. Figure 4.1 provides a class diagram illustrating the relationship between tasks, and how each task contributes to the system. Below some classes will be described in details.

4.1 Motion Load and Save

The input and output of the system are both BVH files which are downloaded from CMU Graphics Lab Motion Capture Database [42]. But what is BVH? How does it record the motion? How can we read the data from it, and how can the data been interpreted for the practical use? The following gives the answers.

4.1.1 What is BVH File

The name BVH stands for Biovision hierarchical data. A motion capture services company, called Biovision, developed this file format to provide the skeleton hierarchy information in addition to the motion data. A BVH file contains two sections: hierarchy and motion. In the hierarchy section, it provides the skeleton hierarchy of a human body, as well as the local translation offsets of the root and the joints. In the motion section, it provides the root's global translation and rotation values, and the joints' local rotation values relative to their parents. It perfectly defines a character's motion. However, it lacks a full definition of the basis pose (no rotation offset is defined).

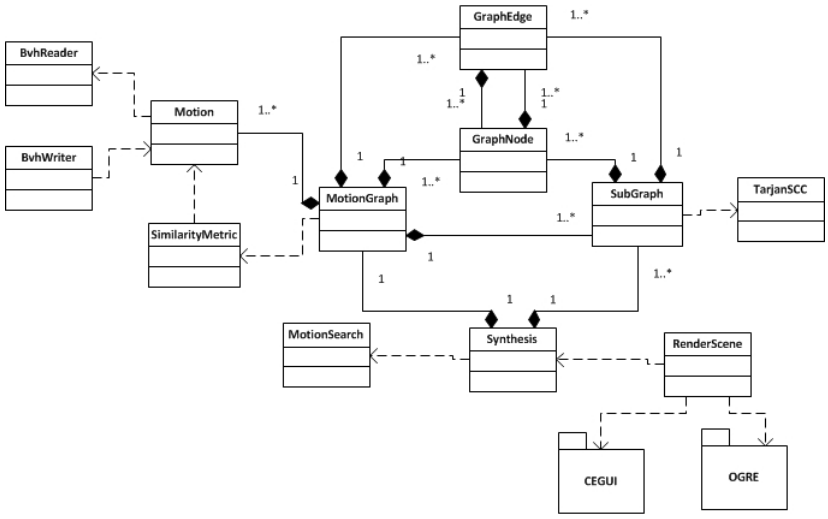

Figure 4.1: System Class Diagram

The data in this file can be read and written as file streams in C++ programming. The only issue is that the data in the hierarchy section is recorded in a table format.

4.1.2 Parsing BVH File

An example BVH file is provided in Appendix A. The first section begins with the keyword **HIERARCHY**. The next line starts with the keyword **ROOT**, followed by the name of the root segment. A BVH file permits more than one hierarchy. Another hierarchy is also denoted by the keyword **ROOT**. Following **ROOT** are the segments: **JOINT** and **End Site**. The BVH format is a recursive definition. Each segment of the hierarchy contains some data relevant to just that segment. Then it recursively defines its children inside accolades.

Each segment has some data started by keywords **OFFSET** and **CHANNELS**. Following **OFFSET** is the 3-dimensional vector (x, y, z). For different segments, the data following **CHANNELS** varies. **ROOT** has six channels, providing global translation (x, y, z) and rotation (x, y, z). **JOINT** only has three channels, defining the local rotation relative to its parent. And **End Site** does not have any channels. The order of the rotation channels appears strange, as it goes from z to y and finally to x. In some files, the order varies for different segments. It may go from z to x and finally to y. This order must be kept unchanged when parsing the hierarchy, because

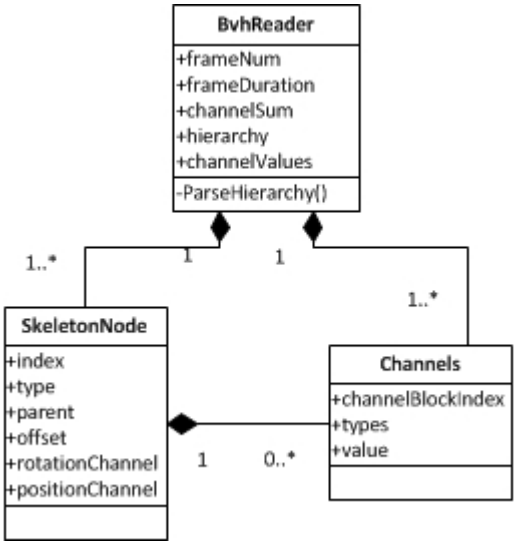

Figure 4.2: BVH Data Structure. *SkeletonNode*: root/joint node. It has position channels(*positionChannel*) and rotation channels(*rotationChannel*). Joint's *positionChannel* is null. *type*: indicates node type root/joint. In *Channels*, *channelBlockIndex*: the place of the *Channels* in the list of channels in motion data; *value*: 3D vector value (x, y, z) of the channel; *types*: the order of channel value x, y and z. In *BvhReader*, *hierarchy*: a vector container containing all root/joint nodes; *channelValues*: an array($N * M$) storing all channels in the motion data (N is frame number, and M is *Channels* number in one frame); $frameNum$: total frame number(N); $frameDuration$: sampling rate; *channelSum*: total channel number(M).

it should match the motion data provided in motion section.

The motion section begins with the keyword **MOTION** on a line by itself. This line is followed by a line that uses the keyword **Frames:** and provides the number of frames. On the next line is the **Frame Time:** indicating the sampling rate of the motion data. In the example of Appendix A, the rate is given as 0.0083333 (120 frames a second). The rest is the actual motion data containing position and rotation channels. Each line is just one sample(one posture), and the channel values appear in the order as that in the hierarchy section.

To describe a motion from the BVH file, we mainly need a hierarchy data structure, shown in Figure 4.2, and a motion data container. The hierarchy data structure should enable keeping track of each joint's parent. Also a mapping index is needed to map the channels in the hierarchy to the channels in the motion data.

4.1.3 Interpreting The Data

After extracting information from the BVH file, a motion can be represented. However, BVH file only provides the global position and rotation of the root, and the local rotations of the joints relative to their parents. We still need the global position of each joint segment if we want to draw a character.

To calculate the position of a joint segment, first we create a transformation matrix M_l from the local translation and rotation information. For any joint the translation information will simply be the offsets as defined in the hierarchy section. The rotation data comes from the motion section. For the root, the translation will be the sum of its offset and its global translation from the motion section. Unlike other motion data files, BVH does not account for scales. As BVH uses the Euler angle to represent the rotation about an axis, we need to calculate the rotation by quaternion:

$$Q = Q_z Q_y Q_x \tag{4.1}$$

where Q_x, Q_y and Q_z are the rotations about the axis x, y, and z respectively, in quaternion presentation. The order should be strictly followed, as quaternion multiplication is non-commutative. Using Ogre maths library, we can simply transform the positional vector and rotational quaternion to transformation matrix. Once the local transformation is created, concatenate it with the local transformation of its parent, then its grandparent, and so on.

$$M_{global} = M_{root}...M_{parent}M_l \tag{4.2}$$

Because Ogre uses column vectors when applying matrix multiplications, the transformation implemented by the matrices happens right-to-left. So in (4.2) the joint is transformed first by M_l, then M_{parent}, ..., and finally M_{root}. The order is vital since matrix multiplication is not commutative.

The motion section in BVH contains certain number of frames. Each frame is a posture represented with position and rotation channels. A posture stays unchanged if we translate it along the floor plane or rotate it about the vertical axis. This is called 2D alignment transformation. After this transformation, only root global position and

rotation Q_{old} will be changed:

$$X_{new} = X_{old} * \cos angle + Z_{old} * \sin angle + X_{trans} \tag{4.3}$$

$$Z_{new} = -X_{old} * \sin angle + Z_{old} * \cos angle + Z_{trans} \tag{4.4}$$

$$Q_{new} = Q_{old} * Q_{(y,angle)} \tag{4.5}$$

where (X_{trans}, Z_{trans}) is the translation along the floor plane, and $angle$ is the angle rotated about the vertical axis. (X_{old}, Z_{old}) is the original position of the posture on the floor, and (X_{new}, Z_{new}) is the new one after the 2D transformation. $Q_{(y,angle)}$ is the quaternion representing a rotation about the vertical axis by $angle$. Q_{old} is the posture's original rotation and Q_{new} is the new one. The quaternion multiplication in (4.5) indicates applying rotation $Q_{(y,angle)}$ to Q_{old}.

4.2 Motion Graph Construction

We obtain the original motion clips by loading BVH files as the original data for motion graph construction. In addition to the hierarchy and the motion data, a motion clip can also be identified with descriptive labels describing what the motion is, like "walk", "run" and "jump". The descriptive information is set in motion loading configurations, see Section 3.1, and plays an important role in later motion search.

In general, the motion graph is a directed graph containing edges and nodes. Edges are the motion clips and nodes are points joining edges. With the original motion clips on hand, a motion graph is initialised with all these motion clips as edges and placing two nodes at the beginning and at the end of each edge, see Figure 4.4. An edge in the graph can be split, and then a node will be inserted to join the two split edges. Two individual nodes can also be connected with a transition edge. Unlike original motion clips, the transition is a motion clip that is created by linear interpolation of original motions. Figure 4.3 shows the motion graph data structure.

In motion synthesis, creating a transition is one of the most difficult but important tasks. For example, a character should perform a smooth transition from walking to running, while this transition needs to take some time to keep realistic. Linear interpolation is a commonly used method to keep this transition as smooth as possible.

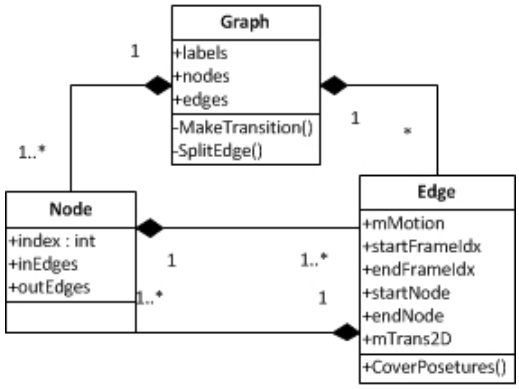

Figure 4.3: Motion Graph Data Structure. A graph contains multiple edges and nodes, as well as descriptive labels describing what kinds of motions this graph includes. A node is attached with incoming edges and outgoing edges, and an edge has a start node and an end node. Also a node is numbered ($index$) for identification. In addition, an edge is a motion clip that clipped from $startFrameIdx$ to $endFrameIdx$ of the corresponding motion ($mMotion$). Also when an edge is made from transition, it has a 2D transformation matrix ($mTrans2D$).

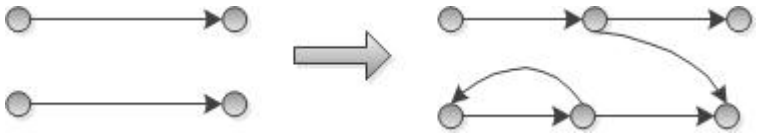

Figure 4.4: A trivial motion graph. (left) The motion graph is initialised with two initial clips. (right) The edge can be split and a new node is inserted. A new edge (transition) can be created connecting two individual nodes.

It could keep both the features of walking and running while balancing them with a factor. The following two sections provide our solution, including detecting candidate transitions from original motion clips and creating transitions between them with linear interpolation.

4.2.1 Candidate Transitions Detection

Smooth transitions should be created by linearly interpolating similar poses. Therefore, detecting candidate transitions is detecting pieces of similar poses from original motion clips. We use similarity metric to measure the similarity between poses.

The similarity metric is a distance function $D(A_i, B_j)$, where A_i is the pose in ith frame of motion A, and B_j is the pose in jth frame of motion B. The smaller this distance value is, the higher the possibility to make a transition from A_i to B_j is. From Section 4.1 we know that a skeleton pose is represented by positional and rotational vectors of root and joints. Thus, to measure the distance function, we calculate the squared Euler distances between corresponding roots or joints in the two poses and then sum them up. However, a good distance function not only takes into account the static posture difference but also the motion dynamics difference. Imagine, for example, that two walking poses are said to be similar when only considering the former factor, but one tends to step the left foot while the other tends to step the right foot. To address this, we adopt the point cloud metric proposed by Kovar and his colleagus [21].

Here, we treat each root or joint on the skeleton as a point. Then points on the skeleton of one pose compose a small point cloud. To calculate the distance $D(A_i, B_j)$, we consider the point clouds formed over two windows of frames of user defined length L. One is formed by extracting L neighbour frames after A_i and the other by extracting L neighbour frames before B_j, see Figure 4.5. The two windows of point clouds can capture the motion dynamics bordered at one frame, as well as the global positions of root and joints of all postures involved. The distance function sums up all squared Euler distances between corresponding points, and a small distance value indicates a high possibility to make a transition from A_i to B_j. The length of the transition edge is the size of the window, therefore $D(A_i, B_j)$ is affected by every pair of poses that makes a pose in the transition edge. In practice, we define the window length L as one third of a second.

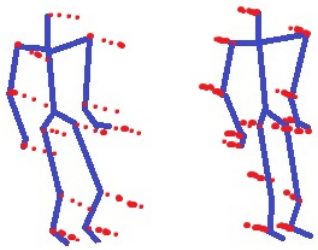

Figure 4.5: Point Clouds of Two Poses. one bordered at the beginning of one posture and other bordered at the end of the other posture. A similarity metric should also consider the motion dynamics difference.

Before we calculate the distance function, we first need to make a 2D transformation to B_j so that the two poses are aligned in the same coordinate system, as mentioned in Section 4.1.3. To find an appropriate 2D transformation $T_{\theta,(x_0,z_0)}$ for B_j, we take into account all points in the widows:

$$\theta = \arctan \frac{\sum_i \omega_i (x_i z_i' - x_i' z_i) - \frac{1}{\sum_i \omega_i}(\bar{x}\bar{z'} - \bar{x'}\bar{z})}{\sum_i \omega_i (x_i x_i' + z_i z_i') - \frac{1}{\sum_i \omega_i}(\bar{x}\bar{x'} + \bar{z}\bar{z'})} \tag{4.6}$$

$$x_0 = \frac{1}{\sum_i \omega_i}(\bar{x} - \bar{x'}\cos\theta - \bar{z'}\sin\theta) \tag{4.7}$$

$$z_0 = \frac{1}{\sum_i \omega_i}(\bar{z} + \bar{x'}\sin\theta - \bar{z'}\cos\theta) \tag{4.8}$$

where ω_i represents both the joint weight, telling how important the joint should be considered, and the frame weight, telling how much the frame in the window affects the result. ω_i results from the multiplication of these two factors. We use Gaussian function to weight the frames which taper off from A_i to the end of the window B_j. Also we use joint weights in Table 4.1 which possibly optimized the result after most tests. (x_i,z_i) is the global position of the point in the point cloud of motion A, and (x_i',z_i') corresponds to the point of motion B. $\bar{x} = \sum_i \omega_i x_i$ and the other barred terms are defined similarly. Finally, the distance function is:

$$\sum_i \omega_i \| p_i - T_{\theta,(x_0,z_0)} p_i' \|^2 \tag{4.9}$$

Table 4.1: Joint Weights for Calculating Distance Function in Simialriy Metric

Joint	Hip(L/R)	Knee(L/R)	Ankle(L/R)		
Weight	0.4	0.4	0.9		
Joint	lowerback	Chest	Chest2	lowerneck	Neck
Weight	0.8	0.1	0.1	0.1	0.8
Joint	Shoulder(L/R)	Elbow(L/R)	Others		
Weight	0.4	0.2	0		

where p_i is one point in the point cloud window of motion A, and p_i' corresponds to the point of motion B.

In principle, we calculate the distance function for every pair of frames in the original motion data. The result of comparing motion A with motion B is a 2D error matrix, see Figure 4.6. To extract candidate transitions from the error matrix, we find all local minima using the algorithm illustrated in Algorithm 1.

Actually, not all local minima in the error matrix indicate high-quality transitions. A value can be treated as a local minima only if it is lower than its neighbours, but the value itself may be high. Therefore, we need a threshold to extract those good enough candidate transitions. To find an appropriate threshold, we should consider the difference of two motions. If the two motions are close, like fast walk and slow walk, the threshold needs to be low. Otherwise, it is high. We define a low threshold and a high threshold to address this issue. Finally, good transitions are extracted from similarity metric. At the same time, the 2D alignment transformation is also recorded as we need to align motions when making a transition or appending one motion clip to another, as the $mTrans2D$ in motion graph data structure (Figure 4.3). We will discuss this later.

Algorithm 1 Algorithm for Finding Local Minima in 2D Matrix

for all point $P_{i,j}$ in 2D error matrix **do**
  if $Value_{P_{i,j}} >$ value of any point around $P_{i,j}$ (3*3 neighbourhood) **then**
    $OutputValue_{P_{i,j}} = MaxValue$.
  else
    $OutputValue_{P_{i,j}} = Value_{P_{i,j}}$.
  end if
end for

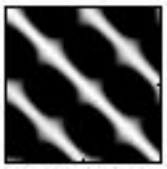

Figure 4.6: 2D Error Matrix. The entry (i,j) is the similarity value that measures the possibility of making transition from ith frame of motion A to $(j + L - 1)th$ frame of motion B, where L is the window length of point clouds. White values corresponds to low distance function value (high similarity value). The local most white values imply possible candidate transitions.

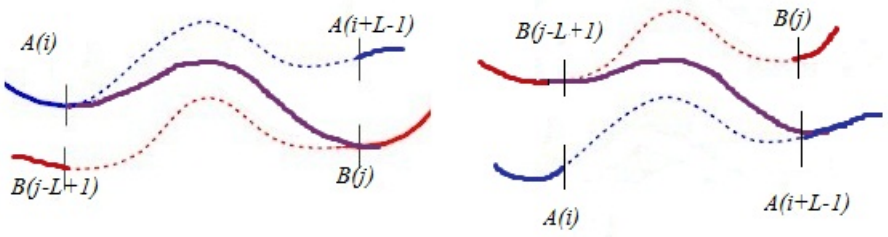

Figure 4.7: Make transitions between motion A and B. (left) Transition from motion A to motion B. (right) Transition from motion B to motion A. If the transition is from A to B, motion B should make a 2D transformation to align with motion A. Otherwise, motion A makes the 2D transformation to align with motion B.

4.2.2 Transition Creation

If $D(A_i, B_j)$ meets the requirement to be a candidate transition, then two transitions can be created. Both transitions result from blending frames A_i to A_{i+L-1} with frames B_{j-L+1} to B_j. The difference is that one is the transition from motion A to motion B (the transition goes from A_i to B_j), while the other is from motion B to motion A (the transition goes from B_{j-L+1} to A_{i+L-1}), see Figure 4.7. The following describes how to make a transition from motion A to motion B.

To start with, the frames in motion B need to make a 2D alignment transformation, making sure that they are in the same coordinate system with the frames in motion A, as mentioned in the end of Section 4.2.1. The transforming method is provided in

Section 4.1.3. We use linear interpolation for blending frames, that is on kth frame C_k of the transition C we perform common linear interpolation on root positions, and spherical linear interpolation on rotations of root and joints:

$$P_{C_k} = \alpha(k) * P_{A_{i+k}} + (1 - \alpha(k)) * P_{B_{j-L+1+k}} \tag{4.10}$$

$$Q^p_{C_k} = Slerp(\alpha, Q^p_{A_{i+k}}, Q^p_{B_{j-L+1+k}}) \tag{4.11}$$

where P_{C_k} is the root position on the frame C_k, and $Q^p_{C_k}$ is the pth joint rotation. Other barred terms are defined similarly. The interpolation factor α is computed as follows in order to ensure it goes from 1 to 0 when k goes from 0 to $L-1$:

$$\alpha(k) = 2(\frac{k+1}{L})^3 - 3(\frac{k+1}{L})^2 + 1 \tag{4.12}$$

Each original motion is labelled with descriptive information describing what the motion is. When making a transition, its descriptive labels are the union of both original motions' descriptive labels $Label_A \cup Label_B$. Therefore, in motion search(Section 4.4) we know that the transition edge contains both features of the original motions.

4.3 Pruning The Graph and Extracting Subgraphs

The motion graph constructed in Section 4.2 can not be used for motion search directly, because it may have some *deadends* and *sinks*. A *deadend* is the node that does not belong to any cycle in the graph, and a *sink* is the node that can only reach part of the total numbers of nodes in the graph. These two kinds of nodes may probably cause the motion search to be halted when no successor is found. An example of rough motion graph is shown in Figure 4.8. So we need to extract a strongly connected component(SCC), resulting in a motion graph where every node can reach to all other nodes in the graph. We use the method proposed by Tarjan [40] to get SCCs for the motion graph, see Algorithm 2, and then extract the largest SCC which contains the most nodes.

Moreover, to make the motion search (Section 4.4) more efficiently, we extract subgraphs from the motion graph, each subgraph contains one motion type with one descriptive label set. Hence to extract a subgraph for a descriptive label set *Label*,

Algorithm 2 Tarjan's Strongly Connected Components Algorithm.

```
input: graph G=(V,E)
output: set of strongly connected components (sets of vertices)

index:=0
S:=empty
for all v in V do
  if v.index is underfined then
    strongconnect(v)
  end if
end for

function: strongconnect(v)
// Set the depth index for v to the smallest unused index
v.index:=index
v.lowlink:=index
index:=index+1
S.push(v)

// Consider successors of v
for all (v,w)in E do
  if w.index is undefined then
    //Successor w has not yet been visited, recursive on it.
    strongconnect(w)
    v.lowlink := min(v.lowlink,w.lowlink)
  else if w is in S then
    // Successor w is in stack S and hence in the current SCC
    v.lowlink := min(v.lowlink,w.index)
  end if
end for

// If v is a root node, pop the stack and generate an SCC
if v.lowlink = v.index then
  start a new strongly connected component
  repeat
    w :=S.pop()
    add w to current strongly connected component
  until (w = v)
  output the current strongly connected component
end if
end function
```

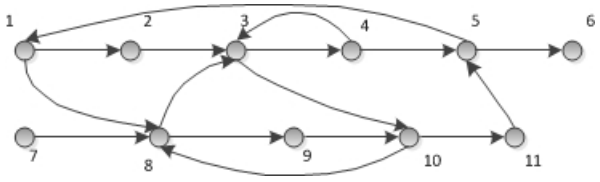

Figure 4.8: A Rough Motion Graph. The motion graph is constructed from original motion clips by splitting edges, inserting nodes and making transitions, where $(1, 2, 3, 4, 5, 8, 9, 10, 11)$ is the largest strongly connected component. (6) is a *deadend*, and (7) is a *sink*.

we search edges that carry *Label*. Similarly, to avoid *deadend* and *sink*, we find the largest SCC for the subgraph, the same work as that for the motion graph before. The result subgraph must be part of the original motion graph, and not all edges and nodes carrying the corresponding descriptive label set are included in the subgraph.

4.4 Motion Search

By this stage, we have got a motion graph and its subgraphs, each graph labelled with descriptive information indicating what kinds of motions it contains. The next stage is to perform a $GraphWalk$ traversing the graphs for an optimal stream of edges that meets user requirements. Then convert the $GraphWalk$ to displayable motion. Hence the contents in this section include: how to extract an optimal $GraphWalk$, how to calculate optimization criteria for $GraphWalk$ evaluation, and how to convert $GraphWalk$ to a displayable motion.

4.4.1 Extracting Optimal *GraphWalk*s

As a motion graph is a directed graph, we can generate a continuous motion stream randomly by appending edges one after another. Apparently, this is not what we want, because we can not control where it goes. For example, a motion is performing walk when suddenly it poses tending to jump and then poses back to walk. Thus, we need a technique to extract an optimal $GraphWalk$ from the motion graph that can yield a realistic motion conforming to user specifications. The start node is randomly chosen.

We cast the extraction as a search problem, and use branch and bound to increase

the search efficiency. An optimization criteria $g(w, e)$, discussed in Section 4.4.2, is supplied here to evaluate the additional error accrued by appending one edge e to the existing $GraphWalk$ w. To evaluate the motion search, we add a $g(w, e)$ each time we append an edge. The total error of a $GraphWalk$ is:

$$f(w) = f([e_1, e_2, ..., e_n]) = \sum_{i=1}^{n} g([e_1, e_2, ..., e_{i-1}], e_i) \tag{4.13}$$

where $[e_1, e_2, ..., e_{i-1}]$ is the existing $GraphWalk$ when appending the edge e_i. Because each time we add an edge, the total error never decreases, $g(w, e)$ is a non-negative value. This is supplied as an important inspiration to apply the branch and bound search.

When dealing with searching an optima through a directed graph, the naive solution is step-first search which compares all possible completed $GraphWalk$s for a best one. However, this performs low efficiency, especially when a long motion stream is required, as the number of $GraphWalk$s will grow exponentially. Using branch and bound, we can save much more time in that it can cull any branch incapable of yielding an optima, see Figure 4.9. As mentioned before, $f(w)$ never decreases when appending an edge, thus we treat $f(w)$ as the lower bound of $f(w + e)$. $f(w + e)$ indicates the error accumulated by appending and edge e to the current $GraphWalk$ w. During searching, if we meet a node of which the lower bound is larger than that of the current found optimal $GraphWalk$, the search is halted for the branch of that node and proceeds to the next un-searched branch. Apparently, we need to keep track of current optimal $GraphWalk$ and store its lower bound.

Although we use branch and bound method, we can not change the fact that the number of $GraphWalk$ searched grows exponentially. Generating long motion stream still costs large amount of time. We try to shorten the time by generating a $GraphWalk$ incrementally. At each step, we only extract an optimal $GraphWalk$ of N frames using Branch and Bound. Then only the first M frames of this $GraphWalk$ can be retained as part of the final optimal $GraphWalk$ required, and the last retained node is chosen as the start node for next branch and bound search. In practice, we set $N = 90$ (about $\frac{3}{4}$ of a second), and $M = 40$ (about $\frac{1}{3}$ of a second). We only allow the searched frame number to exceed the parameter once and then finish the search process, as it can not

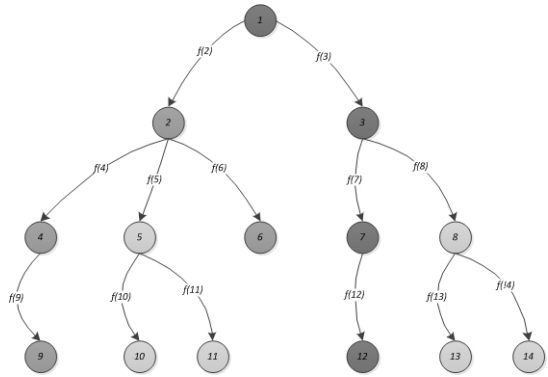

Figure 4.9: Branch and Bound Search. $f(n)$ is the lower bound which represents the accumulated error of the $GraphWalk$ reaching node n. Therefore the lower bound of a child node is calculated by adding the lower bound of its parent with the additional error $g(w, e)$ accrued by appending the edge from its parent to itself. Given that $f(5) > f(9)$, $f(6) > f(9)$, $f(9) > f(12)$, and $f(12) < f(8)$, the branch and bound search order here is [1, 2, 4, 9, 5, 6, 3, 7, 12, 8], and the optimal result is [1, 3, 7, 12]. Because $f(5) > f(9)$, we halt the search for the branch of node 5, the same with node 8. Because $f(9) > f(12)$, the $GraphWalk$ [1, 3, 7, 12] replaces [1, 2, 4, 9] as the current optimal one.

probably be the right N or M after appending an edge.

Moreover, in order to avoid the $GraphWalk$ passing through different motion types, we only allow searching for one motion type in a certain time duration. Thus, we extract subgraphs from the motion graph (Section 4.3). To perform one type of motion, we only search the subgraph G_1 attached with corresponding descriptive label set(L_1) using branch and bound described before. When changing to a different type(L_2), we map the last node retained from searching G_1 to the node in the large motion graph G as the start node, and then start a step-first search in G. We complete any possible $GraphWalk$ that ends with an edge containing L_2, and then find the shortest one with the smallest number of frames, which indicates the quickest transition from the previous motion type L_1 to the current L_2. Considering that the last node retained from the shortest $GraphWalk$ may not exist in the subgraph G_2 of the current motion type, we can not directly start a branch and bound search in that subgraph. So we then start from that node to search G for the edges that only contain L_2 until we meet a node that exists in G_2. Finally, we map that node to the node in G_2 and begin a

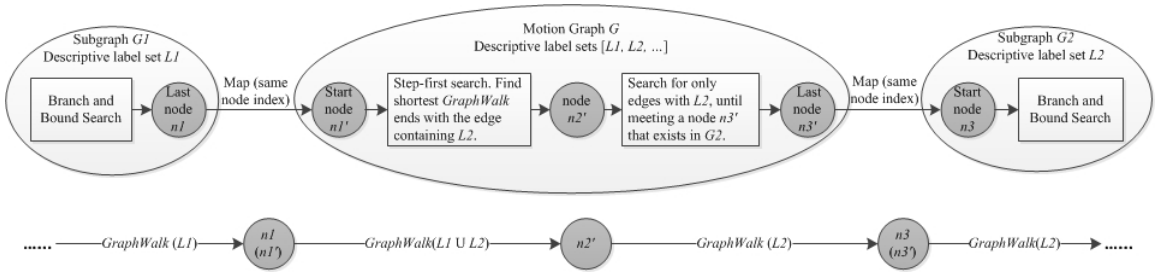

Figure 4.10: Process For Transition Between Two Motion Types. (up) Programming design. (down) Extracted $GraphWalk$.

new branch and bound search. Figure 4.10 describes the transition process.

4.4.2 Optimization Criteria (Path Synthesis)

We need an optimization criteria to value $g(w, e)$, the additional error accrued by appending an edge to the existing $GraphWalk$, as well as a terminating condition for motion search. Usually, this is user specified. To address this, we apply path synthesis, generating motions following a path.

By drawing a path P with the user interface (Section 4.5), we collect the data of that path $P(v_i, l_i)$, where v_i is the ith point on the path and l_i is the arc length from the start point to v_i. The basic idea of path synthesis is that we make the actual path P' travelled by the character most close to P, by evaluating the deviation of P' to P.

To compute $g(w, e)$, we project the character's root position v' on the ground at each frame, forming a piecewise linear curve. We compute the arc length $l'_{(e,j)}$ from the start of path P' to $v'_{(e,j)}$ which is the root position at jth frame on edge e, and then find the point v on path P of which arc length l is the same with $l'_{(e,j)}$. The point v is computed by linear interpolating v_{i-1} and v_i with the factor β:

$$\beta = \frac{l'_{(e,j)} - l_{i-1}}{l_i - l_{i-1}} \tag{4.14}$$

We compute the squared distance between v and $v'_{(e,j)}$, and then sum up the squared

distances for all frames on edge e, resulting in the evaluation criteria $g(w, e)$:

$$g(w, e) = \sum_{j=1}^{n} \|P'(v'_{(e,j)}, l'_{(e,j)}) - P(v, l'_{(e,j)})\|^2 \tag{4.15}$$

To calculate the global position $v'_{(e,j)}$ when appending the edge e to the current $GraphWalk$, we need to apply a 2D alignment transformation to e, making sure that they are in the same coordinate system. In addition, we need to keep track of the information of last frame in the last edge of the existing $GraphWalk$ for computing $g(w, e)$ of the current appended edge e, including the global root position and the arc length from the start, as well as the accumulated 2D alignment transformation. We will discuss the accumulated 2D transformation later in Section 4.4.3.

The terminating condition of $GraphWalk$ extraction is that the total arc length of path P' exceeds that of path P. Also, during branch and bound search process, if the arc length of the current frame exceeds that of path P, the corresponding point is mapped to the end point of path P.

In case, the character may stand still without an incentive to move forward, because it can accrue zero error by staying there. To avoid this, we introduce a small amount of forward progress γ on each frame. Therefore, we replace $l'_{(e,j)}$ in Equation 4.14 with $\max(l'_{(e,j)}, l'_{(e,j-1)} + \gamma)$. In practice, we set $\gamma = 0.1$.

4.4.3 Converting *GraphWalk* to Motion

After extracting an optimal $GraphWalk$, we are close to the end. One thing left is to convert the $GraphWalk$ to a continuous motion stream. As the $GraphWalk$ consists of edges which are pieces of motion, a motion can be generated by placing these pieces one after another in the order as they are stored in $GraphWalk$. The only issue is to place them at the correct position and orientation, which is why we need to record the 2D transformation for each candidate transition detected (Section 4.2.1). In other words, the frames should be aligned in the same coordinate system by applying a 2D transformation T as long as a $GraphWalk$ contains one transition. Start from the beginning of the $GraphWalk$, where T is identity, T is multiplied by a new 2D transformation T_i each time exiting a transition edge. The frames on the next edge are aligned with the current T.

4.5 User Interactive Control

One essential objective of motion synthesis is to generate new motions that meet user specifications. With the graphical user interface designed, we allow interactive control over motions by setting requirements, including selecting motion types, drawing path and setting direction.

- **Motion Type Selection:** As described in Section 4.4.1, we only perform one type of motion in a time duration to avoid the uncertainty that the motion changes from one type to another. Therefore, we select one motion type labelled with descriptive information displayed in the GUI. The choices of descriptive information includes all motion types existed in the motion graph constructed in Section 4.2.

- **Path Drawing:** When moving a mouse to draw the path, the point P_i is captured at each frame the rendering window runs. We capture the global position v_i of the point and then compute the accumulated arc length l_i by:

$$l_i = l_{i-1} + \|v_i - v_{i-1}\| \tag{4.16}$$

  l_0 is defaulted as zero. Then the data $P(v_i, l_i)$ is used in 4.4.2 for path synthesis.

- **Direction Setting:** This is similar to path synthesis. When a direction is set by planting a point on the ground, a path is automatically drawn from the current location of the character towards the point. The following processing is the same as path synthesis.

Chapter 5

Evaluation

This chapter presents some examples of result motions generated from our framework, and provides details in the test and evaluation of system efficiency and performance.

5.1 Results

Generally, we show the results of changing the motion between different types, as well as the application in path synthesis.

5.1.1 Transition Creation

To visualize the transitions created in motion graph construction, we load several different motion types from the database and construct the motion graph. In runtime, we control the character to change between different motion types. Figure 5.1 shows a motion that is generated from the original motions including walk, run and jump. It makes transitions from jump motion to walk motion and then to run motion. The FPS in the picture is 15. From it we can see that the motion can change between different types continuously with a high degree of smoothness.

5.1.2 Artifacts

In common sense, we know that real human motion usually contains foot-plants, which are periods of time when a foot or part thereof remains in a fixed position. Imagine that

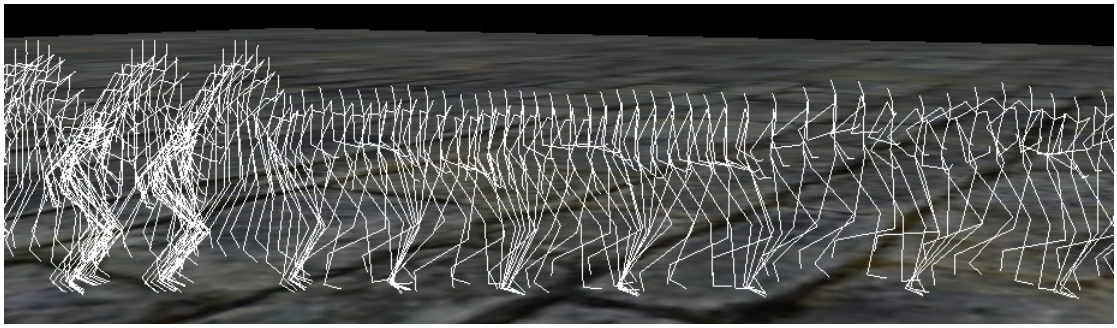

Figure 5.1: Result motion Changing between Different Types.

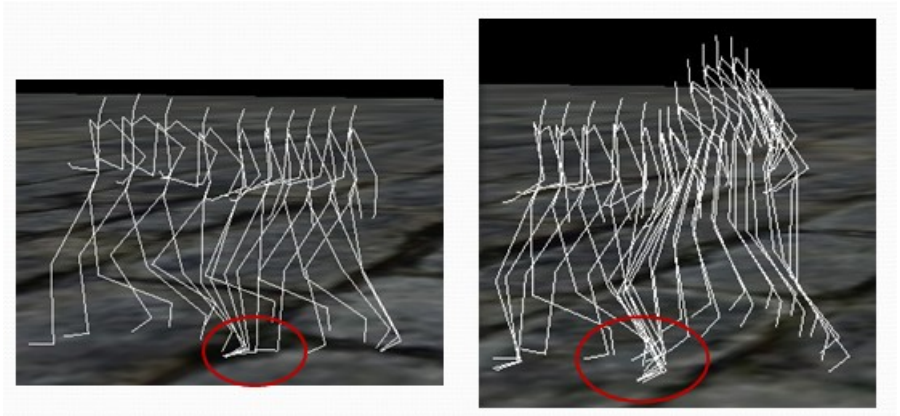

Figure 5.2: Linear blends Caused Artifacts. (left) When the character makes transitions from walking to running, the foot appears slipping in the red circle. (right) The character changes from jump to walk, in the red circle foots are shown not planted in fixed positions in the time duration.

when a character is walking, his left foot should be planted on the ground when the right foot is stepping forwards. However, our result motions present a kind of artifact that the foot is slipping when it ought to be planted, see Figure 5.2. This artifact is caused by linear blending the frames when making transitions, as only low-frequency changes are added to motions while high-frequency details are ignored. In the area of motion editing, this artifact is called *Footskate* [23].

One published solution [21, 23] to the artifacts is to attach constraint annotations to some frames, providing the information that whether the frame contains such a foot-plant issue and how many frames this foot-plant lasts for. Then from the fixed foot

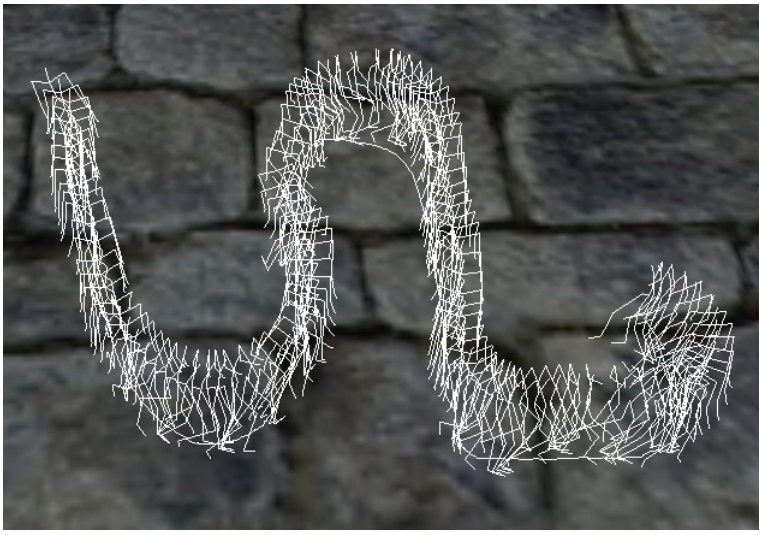

Figure 5.3: Path Synthesis Result 1. The generated walk motion follows the path specified ideally. There are three original motions: “walk straight”, “turn left”, “turn right”.

positions, they used an IK solver to get other corresponding joint and root positions. Another solution takes keytimes that specify periods during which inverse kinematic constraints need to be enforced [18].

5.1.3 Path Synthesis

Path synthesis is an application to our framework that enforces user specifications on motions by drawing a path. The generated motion is supposed to be as close to the path specified as possible. Figure 5.3 shows an ideal result motion that perfectly follows the path. It is generated from three original motions: “walk straight”, “turn left”, “turn right”.

However, sometimes the path travelled by the character can not fit that drawn by users well, because there is not enough original motion data to provide the appropriate motions, see Figure 5.4. However, as our path synthesis tries to find an optimal *GraphWalk* that fits the specifications, the motion shown in Figure 5.4 can finally find a way to turn right in order to follow the path.

Similarly, when a direction is set for the character, it can move towards that direc-

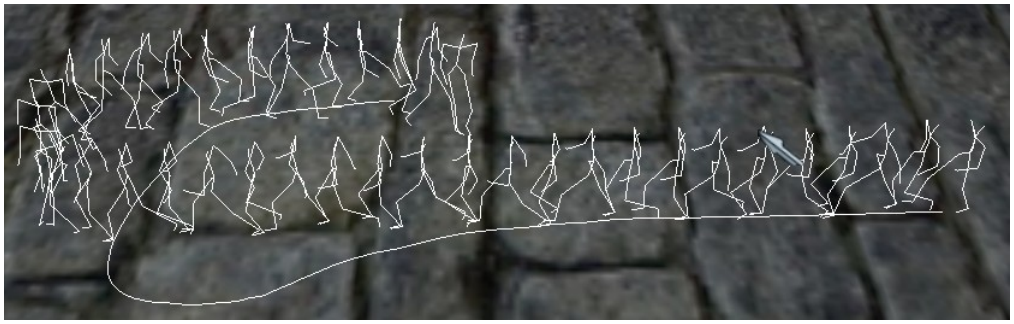

Figure 5.4: Path Synthesis Result 2. The path travelled by the character does not fit that specified by users well. The original motions only contain "walk straight" and "turn right".

tion as long as the original motion data contains the appropriate motions, see Figure 5.5.

5.2 Test

In this section, we test the performance of our framework under different sampling rates of frames.

In motion graph construction, we use similarity metric for candidate transition detection. We compare each pair of frames from the original motion database, which is time consuming work. Therefore, we sample the original loaded data. However, the framework performance including motion graph construction and motion search will be influenced under different sampling rates. In order to evaluate this and find a suitable rate for our framework, we test different sampling rates by evaluating the quality of motion graph, as well as the time consumed in graph construction and motion search respectively.

The BVH files [42] we use adopt the sampling rate of 120 (120 frames a second). First we use three motions: "walk straight"(282 frames), "turn left"(650 frames), "turn right"(400 frames). Table 5.1 compares five rates (120, 60, 30, 20, 15), among which rate 120 and rate 15 can not finish the corresponding task in time. The other three rates can result in almost the same motions shown in the bar chart (Figure 5.6).

Of course, the efficiency is affected by CPU speed, but we are testing in the same

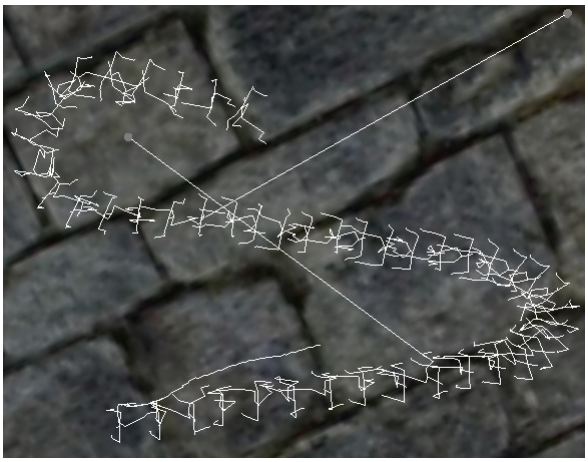

Figure 5.5: Set Direction For The Character. The character can walk towards the new direction set. The original motions contain: "walk straight", "turn left", "turn right".

Table 5.1: Sampling Rate Evaluation: Comparison of Time Cost

Rate(frames/sec)	Time cost in graph construction(s)	Time cost in motion search(s)
120	*not finished in 1h*	/
60	500	5
30	65	60
20	15	330
15	12	*not finished in 1h*

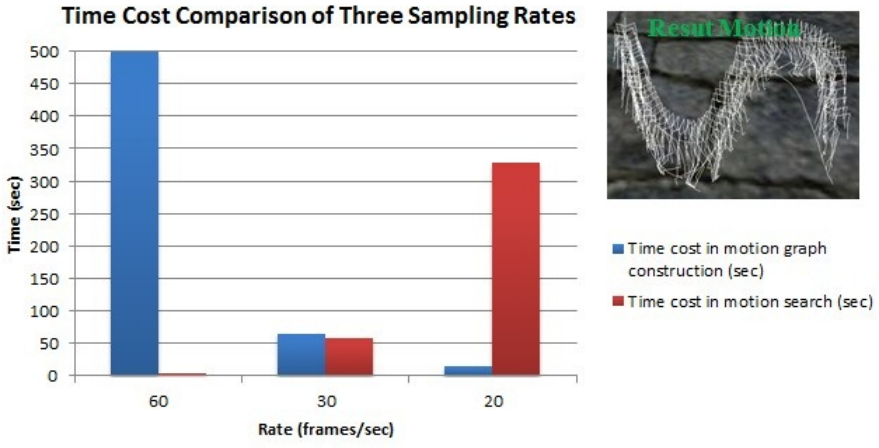

Figure 5.6: Bar Chart of Table 5.1.

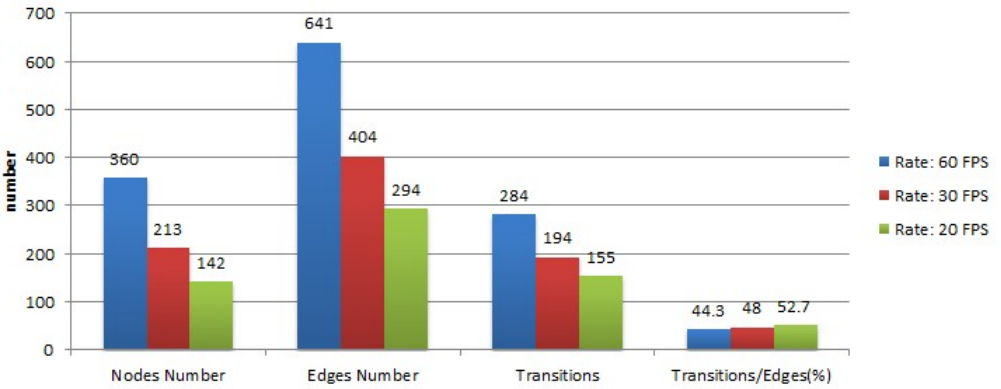

Figure 5.7: Sampling Rate Evaluation: Comparison of Graph Size.

computer, so it does not influence the comparison in Table 5.1. In comparison, we can see that high frequency of sampling will spend more time in graph construction but less time in motion search, while low frequency shows the reverse performance. Because higher sampling frequency (more frames) will cause more pairs of frames involved in similarity metric, resulting in more time costs. In order to test the connectivity of each motion graph (the largest strongly connected component retained from Section 4.3) under its corresponding sampling rate, we compare the numbers of nodes, edges and transitions, see Figure 5.7. From the chart, we see that the lower the sampling frequency is, the less nodes, edges and transitions the motion graph has, therefore the smaller the motion graph's size is. The percentages of the transition numbers are similar. While lower sampling rate needs less time in motion graph construction, the motion graph is not well-connected as there are less nodes and edges, resulting in adding more difficulty to motion search. Besides, from Figure 5.6, we find a sampling rate of 30 can balance the time costs in graph construction and in motion search well, and the time is not that much also. Thus in our framework, we adopt this sampling rate.

However, a low sampling frequency will certainly lose some details from the original motion data, as the original BVH files from CMU library [42] take the rate of 120. To address this, we compare our result under the rate of 30 with the result under the rate of 120, see Figure 5.8. Actually, we can not see the difference apparently, the transitions made under the two sampling rates appear nearly the same, except for the artifacts caused by linear interpolation. However, we can not deny the fact that some

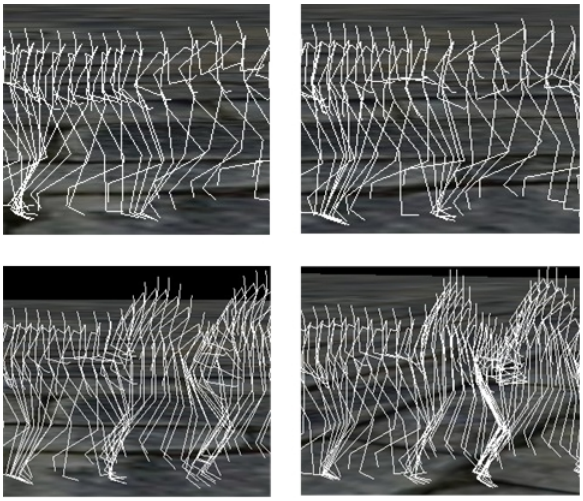

Figure 5.8: Sampling Rate Evaluation: Comparison of Result Motions. (upleft)walk-to-run motion under sampling rate 120. (upright)walk-to-run motion under sampling rate 30. (downleft)walk-to-jump motion under sampling rate 120. (downright)walk-to-jump motion under sampling rate 30.

original motion data will be ignored under low sampling frequency. In fact, we can find other ways to improve the efficiency of motion graph construction, and at the same time keep high frequency of sampling as it keeps more original motion data, which will be a future exploration.

Chapter 6

Conclusions and Future Work

In this paper, we have provided a framework for generating continuous, realistic, long human motions automatically. This framework includes constructing a motion graph that encapsulates connections among original motion clips, which are obtained from BVH files of CMU library [42], and then searching motions through the graph that satisfy user requirements. We apply the framework to path synthesis. Also a graphical user interface is provided for interactive control over motions and visualization of results.

The result motions are continuous and highly realistic. They can make transitions to any other motions with a high degree of smoothness, provided that there are similar enough poses existing between the two transferred motions. In addition, they can follow paths and directions specified by users as long as original motion data provides enough motions. However, without the appropriate post-processing, the motions present artifacts caused by linear blending of frames. The typical one, called *Footskate*, is that the foot slips on the ground while it needs to be planted. In future work, we will try to use constraint annotations describing which frames contain such a foot-plant and how long this constraint lasts for. The fixed foot positions on the ground are then used in post-processing to compute other joints and root positions with an IK solver. Another idea without post-processing can also be taken into consideration that uses inverse kinematic constraints in the linear interpolation process.

Another bottleneck of our framework is the time consumed in detecting candidate transitions for motion graph construction. This process compares each pair of N frames

in the original motion clips chosen from database and therefore involves $O(N^2)$ operations. However, the fact is that only a few candidate transitions exist in the large database. In future work, we can explore a way to divide the motion clips into different categories based on their contract information with the environment, like foot touching ground, hand hanging up, etc. Then in similarity metric, we only compare those frames in the same categories to speed up the candidate transition detection.

Moreover, our framework manually chooses the required motions from database. We can take a further step in the exploration to automatically choose appropriate motions from a large database.

Finally, due to the controllable feature of graph structure, in principle, we can generate motions following more user requirements. Therefore, in future work, we can develop more applications above path synthesis.

Appendix A: An Example of BVH File

```
HIERARCHY
ROOT Hips
{
        OFFSET 0.000000 0.000000 0.000000
        CHANNELS 6 Xposition Yposition Zposition Zrotation Yrotation Xrotation
        JOINT LHipJoint
        {
                OFFSET 0.000000 0.000000 0.000000
                CHANNELS 3 Zrotation Yrotation Xrotation
                ...
                        End Site
                        {
                                OFFSET 0.000000 -0.000000 1.112490
                        }
                ...
        }
        JOINT RHipJoint
        {
                OFFSET 0.000000 0.000000 0.000000
                CHANNELS 3 Zrotation Yrotation Xrotation
                ...
                        End Site
                        {
```

```
                    OFFSET 0.000000 -0.000000 1.112490
                }
            ...
        }
        ...
}
```

Appendix B

Affixed is a disc containing the source code and a video for the project, as well as some related files.

Bibliography

[1] N. I. Badler, C. B. Phillips, and B. L. Webber. *Simulating Humans: Computer Graphics Animation and Control*. Oxford University Press, New York, Oxford, 1993.

[2] Richard E. Parent. *Computer animation: algorithms and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001

[3] A.Watt and M.Watt. *Advanced Animation and Rendering Techniques*. Theory and Practice. Addison-Wesley Publishing Company, 1999.

[4] Aseem Agarwala, Aaron Hertzmann, David H. Salesin, and Steven M. Seitz. *Keyframe based tracking for rotoscoping and animation*. ACM Trans. Graph., 23(3):584-591, 2004.

[5] Doris H. U. Kochanek and Richard H. Bartels. *Interpolating splines with local tension, continuity, and bias control*. In Computer Graphics (Proceedings of SIGGRAPH 84), volume 18, pages 33-41, July 1984.

[6] Alexander Kort. *Computer Aided In-betweening*. In NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering, pages 125-132, New York, NY, USA, 2002.

[7] Scott N. Steketee and Norman I. Badler. *Parametric keyframe interpolation incorporating kinetic adjustment and phrasing control*. In SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pages 255-262, New York, NY, USA, 1985.

[8] Ronan Boulic, Nadia Magnenat-Thalmann, and Daniel Thalmann. *A global human walking model with real-time kinematic personification.* Vis. Comput., 6(6):344-358, 1990.

[9] Ronan Boulic and Daniel Thalmann. *Combined direct and inverse kinematic control for articulated figure motion editing.* Comput. Graph. Forum, 11(4):189-202, 1992.

[10] D. Zeltzer. *Motor control techniques for figure animation.* IEEE Computer Graphics and Applications, 2(9):53-59, 1982.

[11] J. K. Hodgins, W. L. Wooten, D. C. Brogan, and J. F. O'Brien. *Animating human athletics.* In SIGGRAPH 95 Proceedings, Annual Conference Series, pages 71-78. ACM SIGGRAPH, Addison Wesley, August 1995.

[12] J. Laszlo, M. van de Panne, and E. Fiume. *Limit cycle control and its application to the animation of balancing and walking.* In SIGGRAPH 96 Proceedings, 99 Annual Conference Series, pages 155-162. ACM SIGGRAPH, ACM Press, August 1996.

[13] Andrew Witkin and Michael Kass. *Spacetime constraints.* In SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques, pages 159-168, New York, NY, USA, 1988. ACM Press.

[14] CraigW. Reynolds. *Flocks, herds, and schools: A distributed behavioral model.* In Computer Graphics (Proceedings of SIGGRAPH 87), volume 21, pages 25-34, July 1987.

[15] Bruderlin, A., and Williams, L. *Motion signal processing.* In Computer Graphics (Aug. 1995), pp. 97-104. Proceedings of SIGGRAPH 95.

[16] Ken Perlin. *Improving noise.* In SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, pages 681-682, New York, NY, USA, 2002. ACM Press.

[17] Perlin, K. *Real time responsive animation with personality.* IEEE Transactions on Visualization and Computer Graphics 1, 1 (Mar. 1995), 5-15.

[18] Rose, C., Cohen, M., and Bodenheimer, B. 1998. *Verbs and adverbs: Multidimensional motion interpolation*. IEEE Computer Graphics and Application 18, 5, 32.40.

[19] Douglas J.Wiley and James K. Hahn. 1997. *Interpolation synthesis of articulated figure motion*. IEEE Computer Graphics and Applications, 17(6):39-45.

[20] L. Kovar and M. Gleicher. *Flexible automatic motion blending with registration curves*. In Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, (San Diego, California), pp. 214-224, Eurographics As-sociation, 2003.

[21] Lucas Kovar, Michael Gleicher, and Fred Pighin. *Motion graphs*. ACM Trans. on Graphics, 21(3):473-482, 2002.

[22] Lucas Kovar and Michael Gleicher. *Automated extraction and parametrization of motions in large data sets*. ACM Trans. on Graphics, 23(3):559-568, 2004.

[23] Lucas Kovar, Michael Gleicher, and John Schreiner. 2002. *Footskate cleanup for motion capture editing*. Tech. rep., University of Wisconsin, Madison.

[24] Pullen, K., and Bregler, C. 2000. *Animating by multi-level sampling*. In IEEE Computer Animation Conference, CGS and IEEE, 36.42.

[25] Brand, M., and Hertzmann, A. 2000. *Style machines*. In Proceedings of ACM SIGGRAPH 2000. 183.192.

[26] J. Barbi, A. Safonova, J. Y. Pan, C. Faloutsos, J. K. Hodgins, and N. S. Pollard. *Segmenting motion capture data into distinct behaviors*. In Proceedings of Graphics Interface 2004, (London, Ontario, Canada), pp. 185-194, Canadian Human-Computer Communications Society, 2004.

[27] 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 Trans. on Graphics, 21(3):491-500, 2002.

[28] Jehee Lee and Kang Hoon Lee. *Precomputing avatar behaviour from human motion data*. In ACM SIGGRAPH/Eurographics Symp. on Comp. Animation, pages 79-87, 2004.

[29] Okan Arikan, David A. Forsyth, and James F. O'Brien. *Motion synthesis from annotations*. ACM Trans. on Graphics, 22(3), 2003.

[30] Anthony C. Fang and Nancy S. Pollard. *Efficient synthesis of physically valid human motion*. ACM Trans. on Graphics, 22(3):417-426, 2003

[31] Alla Safonova and Jessica K. Hodgins. *Construction and optimal search of interpolated motion graphs*. In ACM Trans. Graph., page 106, 2007.

[32] Leslie Ikemoto, Okan Arikan, and David Forsyth. *Quick transitions with cached multi-way blends*. In ACM Symposium on Interactive 3D Graphics, pages 145-151, 2007.

[33] Sang Il Park, Hyun Joon Shin, and Sung Yong Shin. *On-line locomotion generation based on motion blending*. In ACM SIGGRAPH/Eurographics Symp. On Comp. Animation, pages 105-112, July 2002.

[34] Manfred Lau and James J. Kuffner. *Behavior planning for character animation*. In 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pages 271-280, August 2005.

[35] John Buchanan Mark Mizuguchi and Tom Calvert. *Data driven motion transitions for interactive games*. In Eurographics 2001 Short Presentations, 2001.

[36] Rachel Heck and Michael Gleicher. *Parametric motion graphs*. In ACM Symposium on Interactive 3D Graphics, pages 129-136, 2007.

[37] Liming Zhao and Alla Safonova. *Achieving Good Connectivity in Motion Graphs*. In 2008 SIGGRAPH/ Eurographics Symposium on Computer Animation.

[38] Liming Zhao, Aline Normoyle, Sanjeev Khanna and Alla Safonova. *Automatic Construction of a Minimum Size Motion Graph*. In 2009 SIGGRAPH/ Eurographics Symposium on Computer.

[39] Seth Cooper, Aaron Hertzmann, and Zoran Popovi'c. *Active learning for real-time motion controllers*. In SIGGRAPH '07: ACM SIGGRAPH 2007 papers, page 5, New York, NY, USA, 2007.

[40] Tarjan, R. E. *Depth-first search and linear graph algorithms*. SIAM Journal on Computing 1(2): 146-160, 1972.

[41] James McCann and Nancy S. Pollard. *Responsive characters from motion fragments*. ACM Trans. on Graphics (SIGGRAPH 2007), 2007.

[42] http://mocap.cs.cmu.edu/.