PERCEPTION BASED GAIT GENERATION FOR QUADRUPEDAL CHARACTERS

A Thesis

by

JUNZE ZHOU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Tim McLaughlin
Committee Members,	Ann McNamara
	John Keyser
Department Head,	Tim McLaughlin

May 2013

Major Subject: Visualization

ABSTRACT

With the rapid expansion of the range of digital characters involved in film and game production, creating a wide variety of expressive characters has become a problem that can not be solved efficiently through current animation methods. Key-frame animation is time-consuming and requires animation expertise. Motion capture is constrained by equipment and environment requirements and is most applicable to humanoid characters. Simulation can produce physically correct motion but does not account for expressiveness. This thesis focuses on developing a more efficient animation system using a procedural approach in which the skeletal structure and characteristics of motion that communicate weight and age in quadrupeds have been isolated and engineered as user-controlled tools and modifiers to build creature shape and synthesize cyclic gait animation. This new approach accomplished the goal of quick generation of expressive characters. It is also successful in achieving real-time animation playback and adjustment.

To My Parents

ACKNOWLEDGEMENTS

This thesis would not have been possible without the help, support and patience of my committee chair, Prof. Tim McLaughlin. His guidance and knowledge helped me throughout the research and the writing of this thesis. I also thank the rest of my thesis committee, Dr. Ann McNamara and Dr. John Keyser, for their valuable advice and suggestions.

My special thanks go to Ariel Chisholm, Wei Wang, Spencer Cureton, Jorge Cereijo-Perez, and Anton Agana, without whose knowledge and assistance this thesis would not have been successful.

I thank all the members in the Perception-based Animation research group at the Department of Visualization for providing a solid foundation for this thesis.

I also thank all of the faculty, staff, and students at the Department of Visualization for their encouragement, insightful comments, and for creating such an inspiring learning environment.

Last but not the least, I would like to thank my family members, especially my parents, Lindong Zhou and Jie Xu, for encouraging me to pursue this degree, and for their infinite support throughout my life.

NOMENCLATURE

PLD	Point-light Display
FK	Forward Kinematics
IK	Inverse Kinematics
2D	2 Dimensional
3D	3 Dimensional
PCA	Principal Component Analysis
GUI	Graphical User Interface
MEL	Maya Embedded Language
API	Application Programming Interface

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Recent years have seen an increase in demand for the efficient creation of distinctive digital character animation in the film and game industry. Character animation is the process of manipulating a character rig either manually, or using an animation system, to produce the desired character performance. An efficient animation system is one that can greatly simplify the animation generation process while creating expressive motion. In contrast, inefficient animation systems complicate the process and undermine the capacity to create expressive character animation.

Digital character animation is typically achieved through key-frame animation, motion capture, or procedural systems, such as physically-based simulation. Key-frame animation is a manual process in which animators control character rigs and create key poses that eventually form performances. It produces expressive animation but is time consuming and costly. Motion capture (or mocap) records an actor's performance and transfers it onto digital character rigs. Mocap provides realistic animation but requires specific motion tracking equipment and acting expertise. It is widely used on humanoid characters but is more difficult to achieve with non-human, animal forms. Rule-based animation systems produce animation based on the imitation of real-world behaviors defined mathematically. It brings great benefits, such as simulation generates physically correct animation, but does not typically include motion elements that communicate character traits. In situations that involve large numbers and a wide variety of digital creatures, such as film series *Chronicles of Narnia* (2005 - 2010) and the video game *World of Warcraft*, the animation methods listed above become inefficient for production. To combat this problem, a more feasible approach is in needed.

This thesis aims at developing a perception based animation system to procedurally generate expressive quadrupedal locomotion. In this system, a dynamic quadruped rig is driven by mathematical functions that define animal gaits. Gait functions are acquired through biological motion experiments and analysis of real world animal footage. Users are provided with tools to modify creature skeletal configurations, interpolate between gaits, and adjust two characteristics of the generated motion (weight and age). Motion data used in this thesis is provided by the Perception Based Animation project funded by the National Science Foundation (Award #IIS-1016795)[15]. This thesis is focused on utilizing provided data to construct the target animation system.

2. BACKGROUND AND RELATED WORK

2.1 Identity Signals in Biological Motion

The usual stimuli used in biological motion experiments are Point-light Displays (PLD). A PLD representation of a specific motion is formed by moving dots that reflect the motion of key joints of the subject. This method was invented by Gunnar Johansson [11] and has proven an effective and malleable experimental method for isolating motion from shapes and surfaces in the communication of identity. Figure 2.1 shows an example of representing the progression of a human walk motion with a PLD.

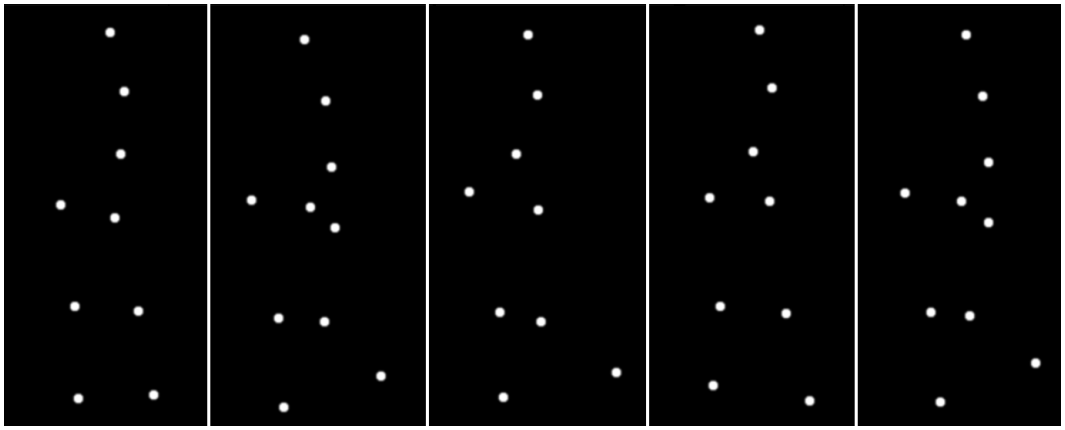

Figure 2.1: A PLD Representation of Human Walk. (side view, right to left) [24]

Biological motion contains rich information that defines the characteristics of a person or animal. Early biological motion experiments suggested that the human brain may contain mechanisms specialized for the detection of other humans from motion signals. In 1977, PLD techniques were used by Lynn Kozlowski and James

Cutting to show that human gender can be identified through walking motion [13]. Later works utilizing PLD by several other researchers has shown that information related to emotion and individual characteristics are also encoded in a person's motion [6, 29]. In 2006, Cord Westhoff and Nikolaus Troje suggested that the human brain might have more generalized detectors tuned simply to the characteristic signal generated by the feet of a locomoting animal [28]. In 2009, studies conducted by Tim McLaughlin and Ann McNamara have shown that animal motion can communicate relative age, relative weight, and the identity as predator or prey [15].

Although it is clear that identity-laden information is communicated through the motion of a person or animal, little is known about how such information is encoded.

2.2 Character Setup

Character setup is the process of creating character rigs that can be either manipulated by animators or controlled by animation systems for the purpose of defining a performance. A standard character rig involves a motion system, a control system, and a deformation system [17]. This section investigates the common practice of constructing these three systems.

2.2.1 Motion System

The first step of character setup is to build a motion system. A motion system can be considered the mechanical architecture that drives the character model [17]. It often consists of a hierarchy of articulation points commonly referred to as "joints" or "bones" by 3D software packages. Joints are connected to each other through parent/child relationships and form a skeletal structure that is similar to a real skeleton. Figure 2.2 shows an example of a digital skeleton used for a tiger character. Determining the joint pivot location is one of the most important aspects of creating a motion system as they define the areas of the character that will articulate [7].

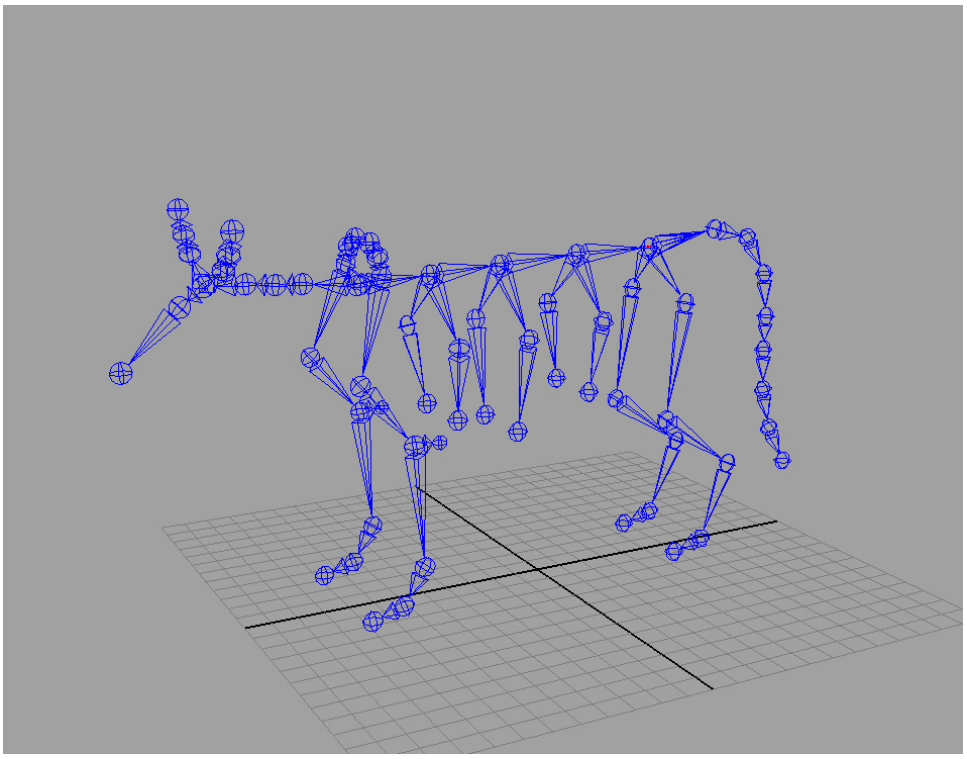

Figure 2.2: A Digital Skeleton Used for a Tiger Character

2.2.2 Control System

The second step of character setup is the creation of a control system. Control systems provide a management structure to the motion system and an interface to animation. It mainly involves the creation of kinematics and control layers.

Kinematics is a mechanical term that defines the math behind movement in a system. Utilizing the theories of kinematics in character setup can speed up animation interactivity and add realistic movements to characters that benefit from articulated joints [7]. The two widely used kinematics are Forward Kinematics (FK) and Inverse Kinematics (IK). FK is a low level approach based on simple parent/child relationships in a hierarchy of joints. It uses the joint parameters to compute the configuration of a joint chain. The position of each joint is determined recursively

by determining the position of the joints above it in the hierarchy.

In late 1980s, IK for determining mechanism motion were widely exploited in robotics [21] . In 1985, IK algorithms were adopted in character animation by Girard and Maciejewski [8]. IK is a high level, goal-directed approach to animation, meaning that animator positions the end joint, or the end-effector in the system and the system solves for the position and orientation of all the joints in the affected hierarchy [7]. Compared to FK, IK generates motion from simplified goals rather than from explicitly defined key-postures. Because IK was proven to be very effective for interactive manipulations, it quickly became one of the most important parts of character control systems.

Control layers provide a layered user-manipulation structure over the motion system and the kinematics. The most commonly used control layer for key-frame animation is a set of visible controllers. Usually, controllers are built as customized shape objects with indication of their main function. For instance, a circle controller indicates rotation and an arrow controller indicates translation toward a certain direction. Besides, a controller is also colored based on the region it is located. For example, controllers on the left side are usually colored as red and controllers on the right side are colored as blue. These rules allow the animator to easily make a distinction between different types of controllers. Figure 2.3 shows the same tiger rig with completed control system.

Multiple control layers are often involved in a complex control system. The complexity of control layers is usually determined by the animation method. For example, if a character rig is designed to be driven by a motion capture system or a physically-based simulation system, additional control layers are needed to receive data from these animation systems. Current animation methods are further discussed in section 2.3.

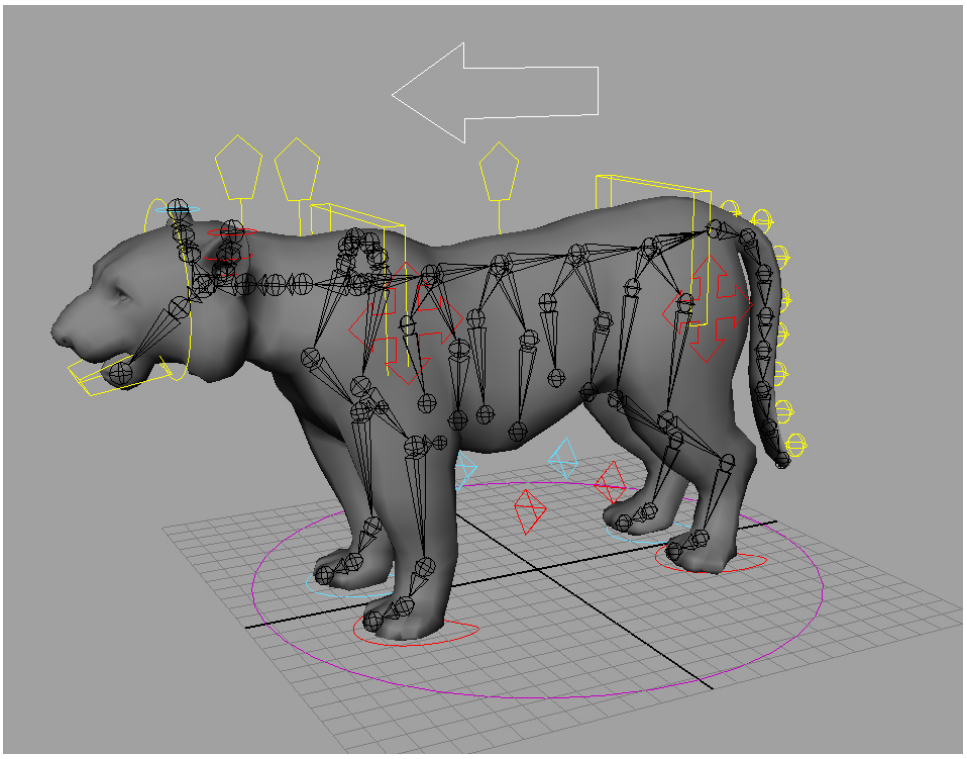

Figure 2.3: The Completed Tiger Rig

2.2.3 Deformation System

The final step of character setup is the creation of a deformation system that defines exactly how the character model deforms in motion. The four most widely used deformation methods are joint based deformation, deformers, dynamics and muscle systems. Joint based deformation utilizes the Skeleton-Subspace Deformation technique to bind a character model to a motion system. Deformers are high-level tools for dynamically changing the shape of a model. Dynamics are physically-based approaches that are used to deform a model through soft body and rigid body simulations. Muscle systems use underlying muscle objects to deform a model in a similar manner that muscles deform real skin [18].

2.3 Current Animation Methods

As mentioned in the previous section, motion is the key of communicating personality. This section investigates current animation methods and their limitations of producing believable motion.

2.3.1 Key-frame Animation

Traditional 2-Dimensional (2D) animation is done by hand. All the frames in an animation have to be drawn as key-frames and in-between frames. Since each second of film requires 24 frames, the amount of work required to create even the shortest of animated films can be tremendous. The use of computers to create human locomotion started at the end of the 1970s [30]. The idea of key-frame in traditional animation was brought into computer animation around the same time. Rather than drawing key-frames manually, 3-Dimensional (3D) computer animation is created through posing a character in 3D space as key-frames, then the computer automatically fills in the in-between frames by smoothly interpolating between key-frames.

Since key-frame animation is a manual process, it requires animators to have a thorough knowledge of animation principles, as well as how to convey personality through motion. The well-known 12 Principles of Animation is a set of rules introduced by the Disney animators Ollie Johnston and Frank Thomas in 1981 [25]. Its main purpose was to produce an illusion of characters adhering to the basic laws of physics while contributing to effective storytelling. It also dealt with characteristic issues, such as emotional timing and character appeal. In 1987, John Lasseter adapted the 12 rules to computer animation. He also pointed out that the success of character animation lies in communicating the personality, and intention of characters [14]. Later, in 1995, the first fully animated feature film *Toy Story* was created by Pixar Animation Studios.

2.3.2 Motion Capture

Motion capture refers to the process of recording of human body movement, or other movement, for immediate or delayed analysis and playback. The information captured can be as general as the simple position of the articulation joint on the character, or as complex as the deformations of the face and muscle masses. Motion capture for character animation involves the mapping of motion data onto the motion of a digital character. The mapping can be direct, such as human arm motion controlling a character's arm motion, or indirect, such as human hand and finger patterns controlling a character's skin color or emotional state.

The use of motion capture started in the late 1970's, and came to be widely used in the late 1990s. In the early 1980's, Tom Calvert attached potentiometers to a body and used the output to drive computer animated figures for choreographic studies and clinical assessment of movement abnormalities [3]. In 1985, Jim Henson Productions created Waldo C. Graphic - a computer generated puppet that could be controlled in real-time in concert with real puppets [27]. In 1988, deGraf-Wahrman Inc. developed "Mike the Talking Head" for Silicon Graphics. The head was driven by a specially built controller that allowed a single puppeteer to control many parameters of the character's face, including mouth, eyes, expression, and head position [22]. In the late 1990s, motion capture was adopted by film industry. *Final Fantasy The Spirits Within* (2011) was the first feature film made primarily with motion capture. Another major step was made by film *Lord of the Rings: Fellowship of the Ring* (2001). It created a unique character Gollum by using a combination of motion capture and key-frame animation [17]. Because of the reusability of the captured motion data and the ability to retarget to a variety of similar character models, game development is another large market for motion capture.

Motion capture produces animation with high fidelity. However, the equipment required to perform motion capture has greatly limited it's range of usage. For example, performing motion capture on real animals as "actors" is almost impossible achieve. As a result, motion capture is primarily used on animating humanoid characters in film and game production.

2.3.3 Rule-based Animation

2.3.3.1 Simulation

Physically based simulation was first used for character animation in the end of 1980s. The earliest simulation system to use inverse dynamics for character animation is proposed by Paul Isaacs and Michael Cohen in 1987 [10]. Similar systems were later implemented in several other works including the simulation of human walking and running [2, 12].

Since motion is generated in accordance with physical laws, the main benefit of simulation is that physically correct animation and interaction can be achieved very easily. However, motion generated by simulation doesn't necessarily contain elements that communicate the identity characteristics of the character. Moreover, the result is not art-directable. Therefore, simulation is usually used as a supplementary tool for character animation rather than a primary animation method.

2.3.3.2 Procedural Animation

Procedural animation is a relatively new approach for character animation. "Procedural" means that the motion of objects in 3D space is directly defined by mathematical functions. This method is used to automatically generate animation in real-time to allow a more diverse series of actions than could otherwise be created using predefined animations. Procedural animation is widely used to animate mechanical objects. However, it is not commonly used for character animation since

mathematically defining characteristic components in biological motion is a challenge that has not yet been met.

In 2002, Nikolaus Troje explored the idea of using linear combinations of sinusoidal basis functions to synthesize human walking motion. In his study, psychophysical experiments of gender recognition were conducted based on a set of holistic motion capture data. Then the result was used with Fourier Analysis and Principal Component Analysis (PCA) to extract the male and female characteristics from the motion capture data [26]. By applying multilinear data analysis techniques to registered motion examples, the multilinear motion models proposed in this study provided an efficient and feasible approach to mathematically describe character motion. It also provides the foundation and guideline for this thesis. Similar methods have been applied in later studies of gait patterns associated with sadness and depression [16] and human female fertility cycle [19]. To date, this approach has not been applied to the analysis and synthesis of animal locomotion.

This thesis presents a new approach to generating expressive quadruped animation for game and film production. Based upon the techniques of Troje and the PLD research method, this new approach is implemented as an animation system that provides the user with control over the skeletal structure of the creature and two identity characteristics of the motion (weight and age).

3. METHODOLOGY

The target system can be divided into four major components: a motion data library, a graphical user interface (GUI), a dynamic quadruped rig, and a gait generator. The relationship between them is indicated in figure 3.1. This chapter describes concepts and methods utilized to design and construct these four components.

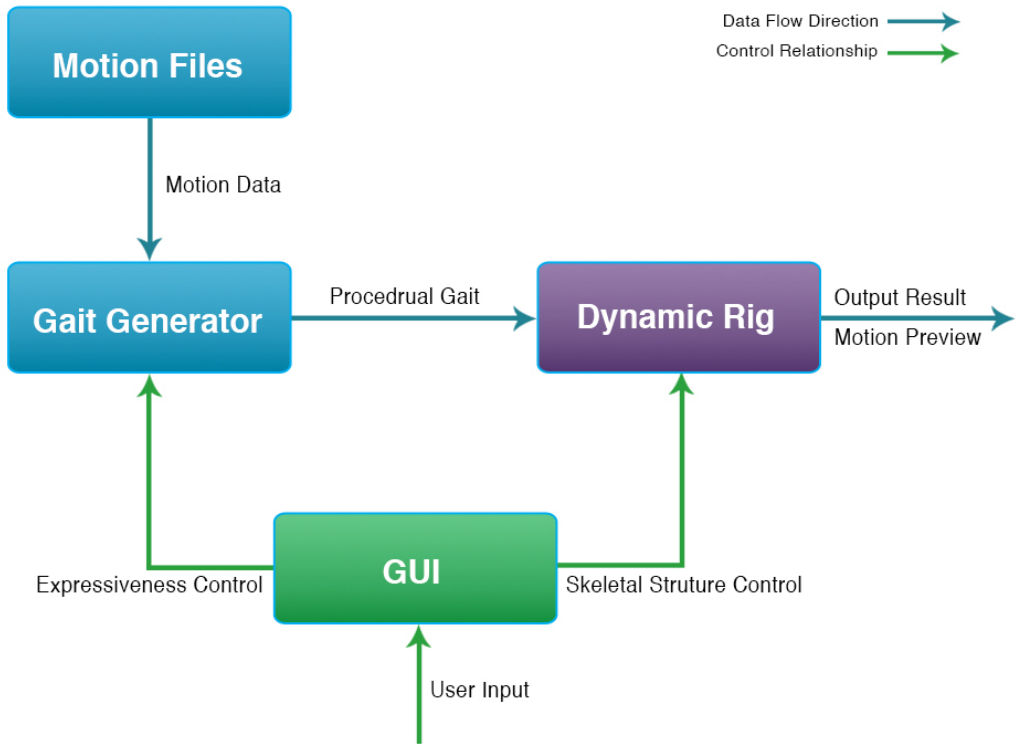

Figure 3.1: System Composition.

3.1 Procedural Motion Data

One of the greatest advantages of procedural animation is efficiency - motion is generated instantly. Therefore, procedural animation has the potential to meet the demand of quick generation of expressive motion over wide range of characters. In order to implement the target procedural animation system, two sets of motion data need to be acquired: data indicating human perception of characteristics in creature motion, and mathematical functions that describe creature gaits. Obtaining these two data sets is not in the scope of this thesis. Desired data has been obtained through the Perception Based Animation project funded by the National Science Foundation (Award #IIS-1016795) with Principal Investigator Tim McLaughlin and Co-Principal Investigator Dr. Ann McNamara [15]. Below is a brief description of the method used to obtain these two data sets.

Motion data is acquired through an approach similar to that conducted by Nikolaus Troje in 2002 [26]. First, video footage of quadrupedal animals walking, trotting, and running are collected as motion examples. These footages were taken from side view to maximize the contained motion information. Then, PLD representations of those examples are created by manually tracking key joints on the creature. Figure 3.2 shows an example of an animal footage and it's PLD representation. After that, PLDs are used as stimuli for eye tracking experiments to determine the expressiveness of the creature and key features that are necessary for conveying the expressiveness in creature motion [15]. At the same time, Fourier Analysis is performed on PLDs to extract sinusoidal basis functions that define the local rotation of each joint on the creature [4]. Rather than recording the sinusoidal functions directly, motion data used by this thesis is stored in plain text files as a series of components required to form the sinusoidal function, such as amplitudes and phases.

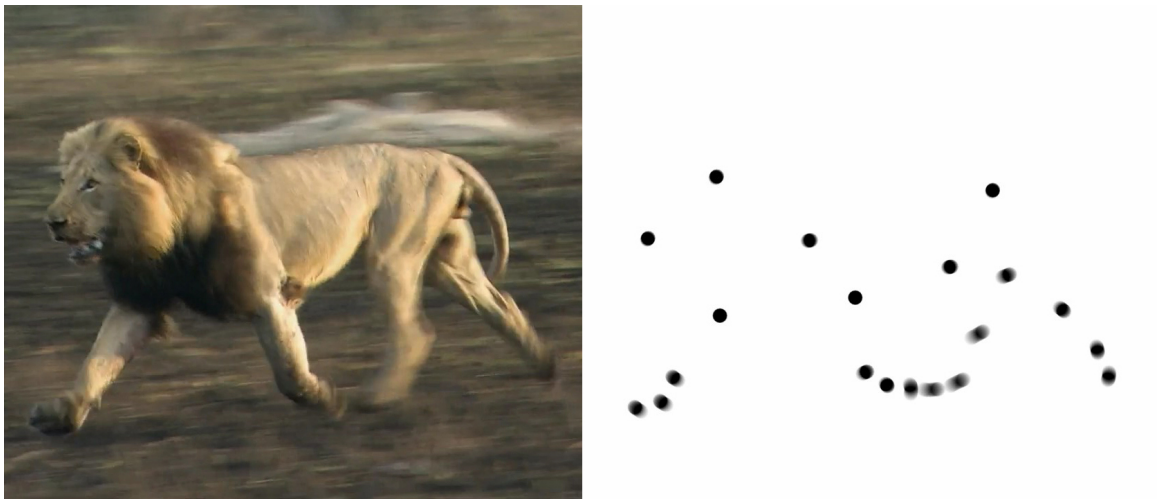

Figure 3.2: A Trotting Lion Footage (Left) and It's PLD Representation (Right).

Sinusoidal basis functions are driven by the time variable. It is important to mention that most of quadruped gaits share the same pattern across both left and right legs, the only difference is that the pattern occurs at different tempos. As a result, only one side of the motion needs to be extracted from this process. When applying motion data, joint rotation values on both legs can be obtained by utilizing the same sinusoidal function but with different time shift. In order to achieve this goal, the speed of the creature (frame per cycle) and the time shift difference (frame) between left and right leg are also recorded in the motion file. A portion of an example motion file can be found in appendix A.1.

One limitation of this method is that only 2D motion data can be acquired, since PLDs are extracted only from 2D video footages. As a result, this thesis is only focused on the generation of 2D motion although the capacity of receiving and displaying 3D motion has built into the implemented animation system.

Quadruped can be divided into three categories: unguligrade, digitigrade, and plantigrade. An unguligrade is an animal that walks on the tips of its toes, such as

horse and springbok. Similarly, a digitigrade walks on its digits and a plantigrade walks with its podials and metatarsals flat on the ground. By the time this thesis is completed, motion data is provided only for the front and hind legs of limited types of unguligrade creatures. Therefore, synthesizing the motion of other body parts and other types of quadrupeds is not included in this thesis. The motion data of these missing parts can be extracted and facilitate this thesis in the future.

3.2 Graphical User Interface

An easy-to-use Graphical User Interface (GUI) is the key visual component for interacting with the target system. Based on the system functionalities described previously, the GUI can be divided into three parts: the shape control panel, the gait control panel, and the rig control panel.

The shape control panel provides the user with tools to modify the skeletal structure of the quadruped rig, in other words, to change the joint position of the creature. Ideally, those tools should include a skeleton preset library which contains skeletal configuration of multiple basic creature types and a customization interface which allows the user to create, store, load, and modify their own skeleton configurations.

The gait control panel provides the user with controls over the type, speed, and characteristics of the procedurally generated gait. The two characteristic components involved in this thesis are weight and age. All of these tools can be visually presented as control sliders to enable easy interaction. In the real world, the characteristics of motion have an influence on the speed of the creature. For instance, a heavy creature tends to walk slower than a light creature. As a result, it is reasonable to provide users with a switch that allows them to either define the creature's speed by themselves, or let the system calculates the speed automatically based on the current characteristics.

The rig control panel provides various options to view the generated motion. This is necessary since users may require different viewing methods. For example, some users prefer viewing the animation as a PLD representation. In this situation, bones indicating the joint hierarchy and control tools should be hidden. It is also important to provide additional view options for research purposes, such as viewing the motion of the front legs in isolation.

3.3 Dynamic Quadruped Rig

The dynamic quadruped rig used in this thesis serves two main purposes: allowing the user to modify the skeletal structure of the rig and displaying the generated creature motion. This section describes the methods used to construct a generic quadruped rig that fulfills these two goals.

3.3.1 Scripted Rig Setup

The desired rig can be setup in a similar approach to animation production. This includes creating a motion and a control system. A deformation system is not involved in this study. The only motion synthesized is skeletal, which is separate from forms and surfaces. Similar to the PLD method, each joint can be visually represented as a dot, or a 3D sphere.

As mentioned in section 3.1, the motion data provided for this study consists of sinusoidal functions for defining local joint orientations. Together, they form the final procedural motion as a combination of multiple sinusoids. Since this type of data is recorded based on the joint hierarchy (or FK), the target rig needs to be setup in an FK fashion so that the data can be displayed properly. On the other hand, an input control layer should be created to receive procedural data and drive joint orientations. This control layer includes building a socket node for each joint in the skeleton hierarchy since the motion data is per-joint based. It is also necessary

to create a standard layer of controllers to allow the user to manually fine-tune each joint's rotation through the key-frame animation method.

Character setup is a linear process which ultimately produces a character rig with integrated motion and control systems. This nature of character setup usually leads to difficulty in revising the rig and adding new features to the rig once it is completed. To make the target animation system adjustable and extendable in the future, the rig is procedurally generated by scripting. This means that the rig creation process is recorded as scripts in a programming language that can be executed within the chosen 3D animation package so that future modification can be done simply through revision of the scripts.

Scripted rig setup involves manually placing guide objects in the 3D space to indicate joint locations. Scripts can then be executed to build the quadruped rig based on the provided position information.

3.3.2 Dynamically Adjusting Rig Skeleton

The reason for giving the quadruped rig a dynamic structure is so that the motion generated by the gait generator is adjustable. As a result, the user needs to have the ability to also modify the rig's skeleton configuration, specifically the joint position, so that they can create a desired creature shape that matches the motion.

A callback mechanism can be designed and utilized to make this dynamic feature possible. Callback is the process of executing a subroutine inside a program when a predefined event occurs. In the case of this study, a set of user-controlled guide objects can be used to represent the current joint locations. When a guide object is moved in 3D space, a callback function inside the system can be triggered to update the corresponding joint position and orientation to match the guide object.

The benefit of this approach is that the dynamic feature exists as an independent

function and separated from the quadruped rig itself. It provides easy control to the user and minimizes the complexity of the rig structure compared to building this feature directly into the rig.

3.3.3 Managing Customized Skeleton Structures

Users have complete control over the shape of the creature, it is therefore necessary to provide them with tools to manage customized creature shapes, including saving, loading, deleting, and mirroring functions.

To implement the saving and loading functions, an efficient method of recording joint positions in character space is needed. Recording position in character space is essential. It allows the system to customize and manage creature shapes based on the current location of the character rig in 3D space. In this way, the user can transform the whole system freely without damaging its functionality. A plain text file is a simple effective media to record joint positions. The target system stores the user-created text file into a specified location on the user's hard disk and reads it back in when called. The reading and writing process can be performed on the same set of guide objects mentioned in the previous section since it represents the current joint positions in 3D space.

Mirroring is a useful feature as most quadrupedal creatures exhibit a symmetrical skeleton structure. A mirroring function speeds up the process of customizing the creature's shape because the user only needs to adjust joints on one side of the creature and mirror their positions.

Another beneficial feature for speeding up the shape customization is to provide a creature library that contains basic presets of quadruped skeleton structures, such as a horse skeleton and a lion skeleton. When a certain preset is selected, the system can update the rig skeleton to match the target creature type. As a result, the user

can switch between different creatures instantly and use these presets as a foundation to create their own creatures.

3.4 Gait Generator

The gait generator is the key component for producing expressive gaits. It is connected to the specified control layer on the dynamic quadruped rig after the rig is completed. The process of gait generation involves reading motion data from stored motion files, calculating joint rotation values, and sending these values to the input control layer which eventually drives the motion of each joint.

3.4.1 Efficiency of Gait Generation

The efficiency of gait generation is essential to enable real-time animation preview. The whole calculation process has to be repeated every frame due to the fact that the sinusoidal function used to compute final joint rotation values are driven by the time variable. Inefficient design of gait generator can cause slow animation generation which in the end undermines real-time animation display.

As discussed previously, gait generation is a fairly onerous process to be repeated every frame. In order to optimize this process, motion data can be read in only once and stored inside the system when the system is initialized. Output plugs should be created to form a direct connection between them and the input control layer on the quadruped rig. In this way, only the joint rotation calculation has to be conducted when the time variable changes.

3.4.2 Characteristic Controls

To provide users with control over the two characteristics included in this thesis, an approach similar to the weighted interpolation method developed by Nikolaus Troje in 2002 [26] is used. To be specific, the target system can perform bilinear or

multilinear interpolation in between registered motion examples, i.e. motion data files, based upon the user input to calculate the output motion that drives the rig. Multilinear interpolation is the method of deriving output value by weighing the calculation results of multiple functions. In this study, a motion example can be determined as a heavy walk during the eye tracking experiment mentioned in section 3.1. When the user raises the weight level through the system GUI, the system can then increase the weight of the sinusoidal functions corresponded to this motion example so that the final motion appears to be heavier.

The gait type control can also be achieve through this method. For example, when the user changes the gait type from walk to trot, the system can increase the influence of trot motion examples while decreasing the influence of walk motion examples so that the final motion transits from walk to trot. This results in a linear transaction between different gait types, which is an effective way to approximate the gait transition in the real world.

In the real world, the gait type and the characteristics of the gait have significant impact on the speed of the creature. For example, a heavy creature tends to walk slower than a light creature. Conversely, speed change can lead to gait type and expressiveness change. Because of this interrelationship between gait and speed giving the user separate control over them can lead to situations in which these two components to cancel each other. For instance, if the user chose to create a heavy walk with a fast speed, the "heavy" feature could be undermined. As a result, a control mechanism needs to be established so that the user can choose to either only control the characteristics or manually define the creature's speed. In the first scenario, the system can automatically calculate the speed based upon the current characteristics. Otherwise, the speed is directly input by the user and will not be influenced by the characteristic change.

3.4.3 Use of Neutral Pose

The target system provides the user with total control over the initial joint locations of the creature, meaning that the rig can be posed with full freedom. It brings great benefit but at the same time leaves initial joint rotation values on the rig. These initial rotation values are necessary for viewing the current pose, but problematic for gait generation.

To apply motion data properly across various of poses created by the user, a neutral pose needs to be defined as a pose with zero rotation value on all the joints. Procedural data can then be applied by using this pose as an initial pose. In further detail, the system sets the current pose to neutral pose prior to applying motion data. This important step cleans initial rotation offsets generated during the shape customization process, but maintains the creature's skeleton structure, or the bone size, that the user created. When the user stops the animation, the system is able to retrieve the initial rotation offsets and put them back on to the rig to restore the customized pose.

4. IMPLEMENTATION

4.1 Development Tools and Programming Languages

4.1.1 Animation Softwares

This project has been developed using Autodesk® Maya® 2012 [1] as the 3D animation system for manipulating and viewing the animated creature and its motion. Maya is an industry standard 3D animation package that is widely used in companies and studios in the field of film and game production. Recent versions of Maya integrated both its own scripting language, Maya Embedded Language (MEL), and Python [20]. Besides, it also provides developers with C++ Application Programming Interface (API), which allows developers to access and manipulate Maya data, customize the Maya GUI, and extend Maya functionality. New features can be loaded into Maya as plug-ins and work seamlessly with the rest of Maya [9].

Qt Creator® [5] is used as the tool for designing and creating the GUI for the target system. Qt Creator is a powerful cross-platform GUI layout and forms builder. The Qt framework is utilized by a large number of production tools in the animation industry, including Autodesk Maya. As a result, the system GUI created through Qt Creator can be easily integrated into Maya interface through provided Maya UI commands.

4.1.2 Use of Programming Languages

Two programming Languages, Python and C++, are utilized in this thesis to construct the two key components of the target system: the dynamic rig and the gait generator. Python is a high-level programming language which is interpreted and emphasized on code readability. Although not complied, Python is fully dy-

namic, efficient, and easy-to-use, which makes it one of the most commonly used programming languages for scripting. Python has been integrated into Maya makes it ideal for performing most of the tasks involved in this study, including automating quadruped rig creation, executing dynamic rig functions, and loading the system GUI.

In comparison, C++ is an intermediate-level programming language that is statically typed, free-form, and compiled [23]. It is not as convenient to use as Python, but since C++ is a compiled language, it runs much faster than Python. As mentioned in section 3.4.1, the gait generator in the target system requires high efficiency in code execution due to its per-frame evaluation nature. To achieve this goal, C++ is a much better choice. For this thesis, Maya C++ API is utilized to construct the gait generator as a plug-in which is eventually embedded into Maya and connected to the quadruped rig.

4.2 Creating the System GUI

A GUI is used to manipulate the skeletal structure of the generic quadruped rig and control the animation generated by the gait generator. To reach these goals, the GUI is constructed as a standalone Maya window with a shape control panel, a gait control panel, and a rig control panel.

4.2.1 The Shape Control Panel

The completed shape control panel is shown in figure 4.1. It consists of two parts: a *Presets* tab and a *Custom Shapes* tab. The *Presets* tab contains the skeletal structure of several selected quadrupeds in the horse and cat family. It provides a basic structure for the user to use as a foundation for developing their own creature and to preview generated procedural motion.

A *Manual Mode* switch is built on the very top of the shape control panel. When

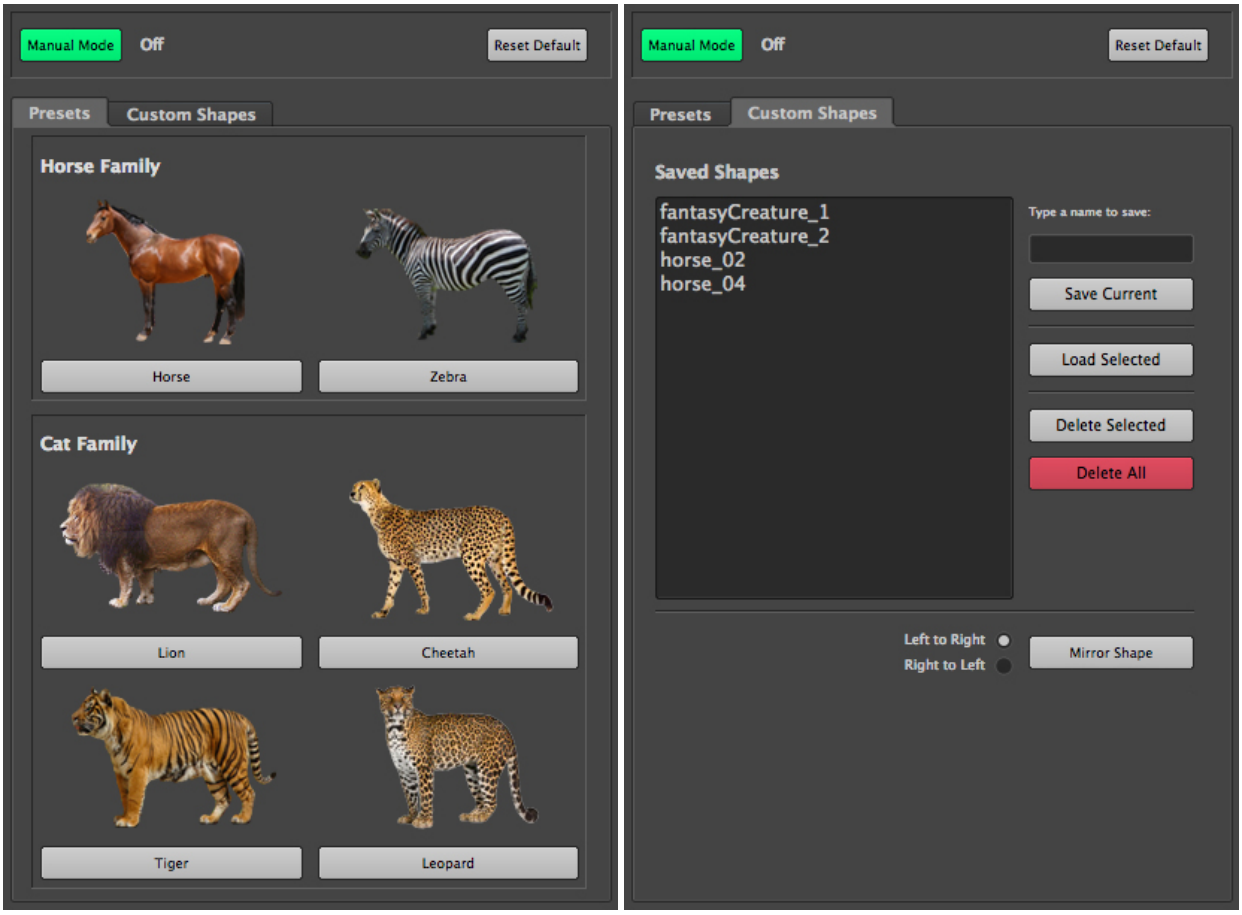

Figure 4.1: The Shape Control Panel Includes Two Tabs: The Creature Preset Tab (Left) and the Creature Customization Tab (Right).

switched on, the system enters manual shape adjusting mode, in which the user can directly reposition each joint on the quadruped rig to create their own creature structure. This process is further discussed in section 4.4. Tools are provided in the *Custom Shapes* tab for saving, loading, mirroring, and deleting custom shapes. As mentioned, custom shapes are stored in a specific folder within the system as plain text files containing joint position data. This folder is scanned during system initialization for valid text files which are then loaded into the *Saved Shapes* scroll list on the *Custom Shapes* tab. A portion of an example shape text file can be found

in appendix A.2

Each button shown on the panel is linked with a specific program routine to achieve its functionality. For example, when the *Save Current* button is pushed, the corresponding function is executed to write current joint position data into a text file in the predefined format. The code for save current shape function can be found in appendix B.1

4.2.2 The Gait Control Panel

Figure 4.2 is a snapshot of the completed gait control panel. The panel consists of three sections: the *Gait* section, the *Speed* section, and the *Characteristic* section.

The *Gait* section includes a *Generate Gait* master switch and a *Gait Type* slider. The master switch starts and stops the procedural animation generation. Users cannot view animation and adjust creature shape at the same time. To make the system more user-friendly, an automated mode query mechanism is implemented to prevent this conflict, i.e. if the manual shape adjusting mode is switched on while the system is generating motion, the animation switch will be automatically set to *Off*. Similarly, it will be set back to *On* after adjustments are completed.

The *Gait Type* slider controls a float type variable that reflects the current gait. In this thesis, walking is represented as 0.0, trotting as 5.0, and running as 10.0. For example, when the user dials the slider from 0.0 to 5.0, the current gait automatically transits from walking to trotting.

The *Speed* section contains a *Speed Multiplier* slider and three speed sliders. As mentioned in section 3.4.2, the speed and the characteristic of the motion influences each other and providing the user with total control over these two aspects may produce undesired results. To solve this issue, a switch is created for enabling user-

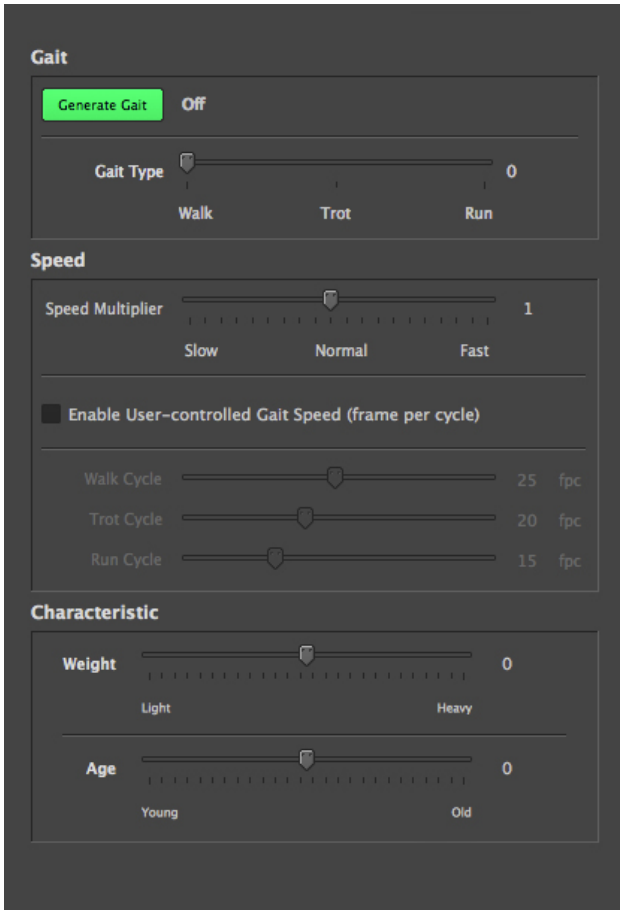

Figure 4.2: The Gait Control Panel.

controlled gait speed. If enabled, the three speed sliders will be activated for the user to individually define the walk, trot, and run speed as frames per cycle, and the *Gait Type* slider and the two characteristic sliders will have no influence on the speed of the creature. If disabled, the creature's speed will be calculated based upon the current gait type and characteristics, and the user has no direct control over the speed of the creature.

The *Speed Multiplier* slider controls a float type variable which serves as a global multiplier to the current speed of the motion. It can be used as a supplementary tool for globally adjusting the animation.

Finally, the *Characteristic* section involves two characteristic sliders controlling the two expressive components included in this thesis: weight and age. Similar to the two sliders discussed previously, they are also represented as float type variables. For example, the weight is represented as a value between -50.0 and 50.0, where -50.0 is light, 50.0 is heavy, and 0.0 represents a neutral weight. When the slider is moved toward the heavy end, the weight value increases and the motion appears to be heavier.

All the user inputs provided by the gait control panel are internally queried and used by the gait generator to control the weight of the corresponding motion examples.

4.2.3 The Rig Control Panel

The rig control panel provides various animation viewing options. As shown in figure 4.3, it allows users to adjust global rig size and joint size, hide or show controls and bones, hide or show certain parts of the creature, and turn on and off colored PLDs.

4.3 The Generic Quadruped Rig

The generic quadruped rig creation is an automated process completed through executing a series of python scripts in an orderly manner. This automated process is guided by a set of locator guide objects stored in a manually created Maya file. This section mainly discusses the key scripts used in this process, including the joint script, the control script, the geometry script, and the clean up script.

4.3.1 Initial Skeleton

The joint script creates an generic quadruped skeleton. As mentioned in section 3.4.3, a predefined neutral pose is essential for applying motion across a wide range

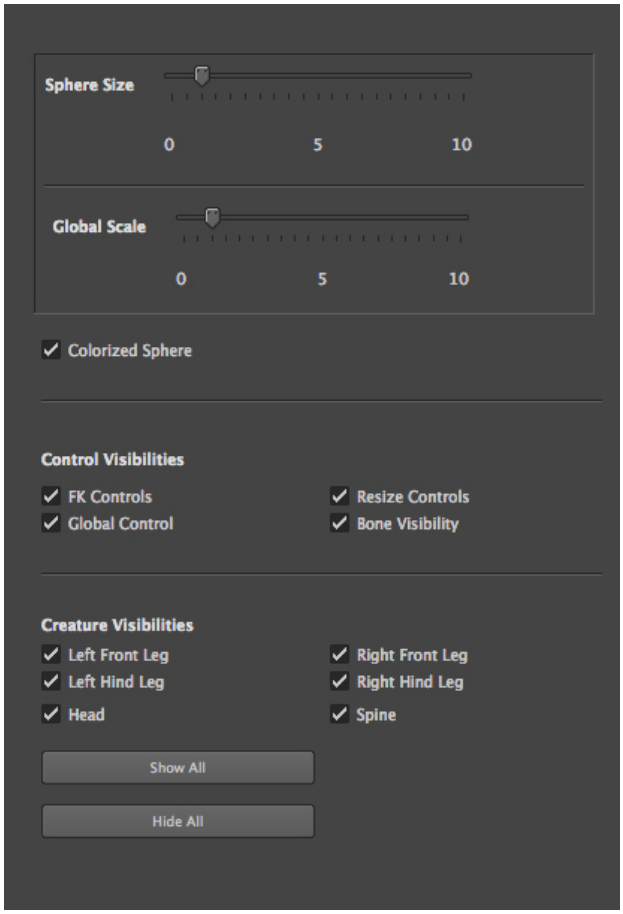

Figure 4.3: The Rig Control Panel.

of skeleton structures. The neutral pose used in this thesis is defined as the pose shown in figure 4.4, where each of the four legs forms a vertical straight line. The skeleton structure used in this thesis only contains two joints on the creature's head and spine. This is because only these joints are tracked during the motion extraction process.

In addition, the joint script constructs another skeleton that is identical to the previous one. This skeleton is used for guiding the shape customization process. Further discussion of its functionality can be found in section 4.4.

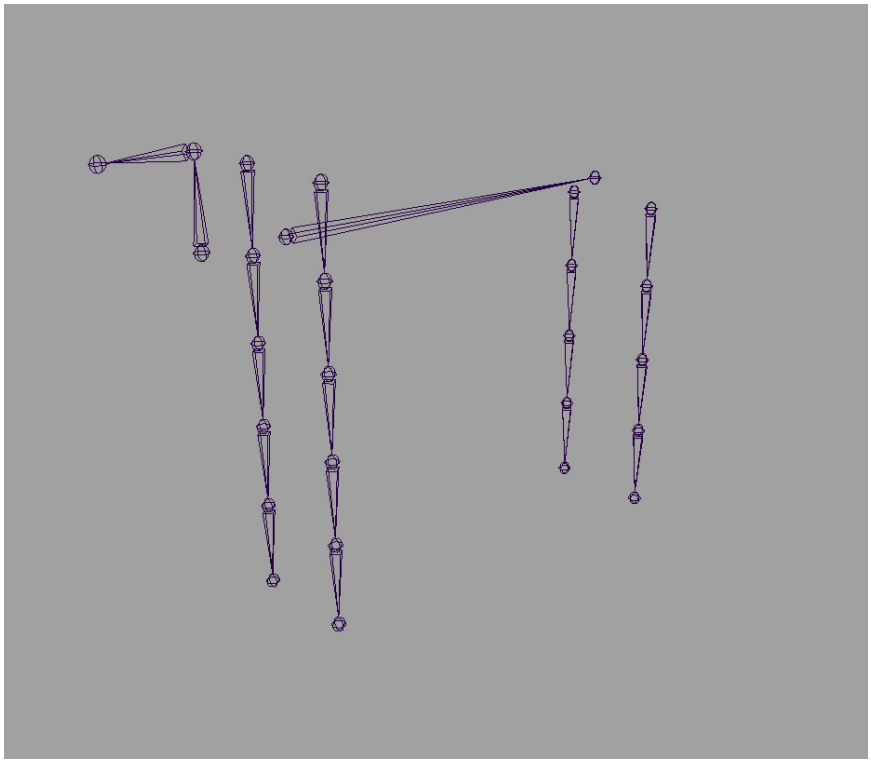

Figure 4.4: The Initial Quadruped Skeleton in Neutral Pose.

4.3.2 Establishing Control System

The control script builds multiple control layers on top of the joint hierarchy. Figure 4.5 shows the complete control layer hierarchy on the left elbow joint. The "elbow_l_CTLGRP" is the top group node. It is also used for storing translational offsets generated when the user reposition this joint. Similarly, the "elbow_l_RotOffset" is the node used for storing rotational offsets generated during the same process. The "elbow_l_CTLPRO" is the input layer for receiving procedural motion from the gait generator. At the very bottom, the "elbow_l_CTL" is the FK controller shown in figure 4.6.

The FK controller is the only control layer that is visible to the user. It controls the joint orientation directly and is constructed in the same FK fashion to the joint

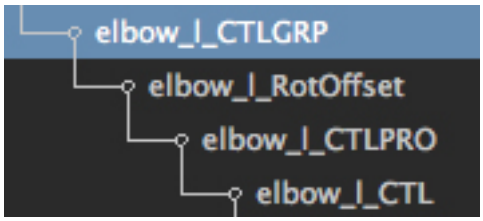

Figure 4.5: The Control Layer Hierarchy of the Left Elbow Joint.

hierarchy. They are built so that the user is able to fine-tune the procedurally generated motion. Besides the FK controller, other control layers are all remain hidden to the user and can only be directly manipulated by the system.

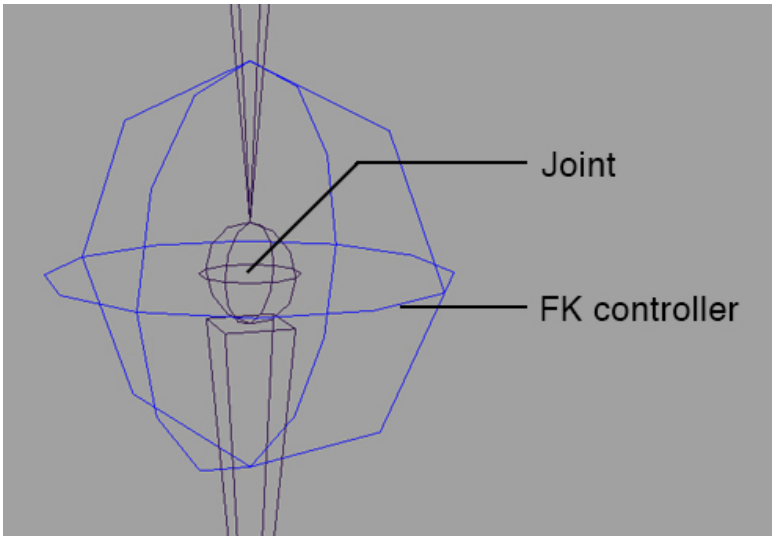

Figure 4.6: An Example of FK Controller.

In addition to establishing control layers, the control script also builds several miscellaneous components related to the shape customization function. These include a set of bone length measuring tools for querying the length of each bone and resize controls for guiding the shape customization process. A portion of the control

script is shown in appendix B.2.

4.3.3 Attaching Joint Spheres and Final Clean Up

The geometry script is executed after the completion of all control layers. It is a straightforward process which attaches a sphere geometry to each joint as the PLD object and assigning a meaningful color to the sphere. Similar to the roles of coloring controllers, color assignment for these spheres is also based on region. This thesis uses blue to represent left side, red to represent right side, and purple to represent center region. Finally, the clean up script is executed to scan the entire rig and lock internal components that should not be manipulated by the user, such as the skeleton and sphere geometries. It also locks and hides useless node attributes, such as translation attributes on the FK controller. Cleaning the rig prevents the user from accidentally breaking the system. The fully established generic quadruped rig is shown in figure 4.7.

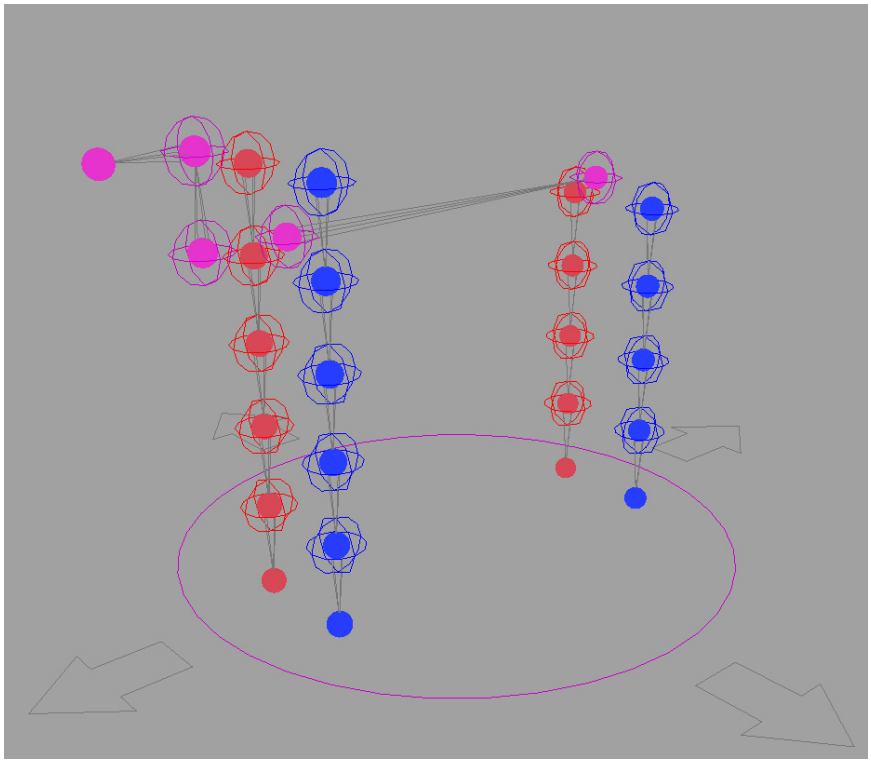

Figure 4.7: Completed Generic Quadruped Rig

4.4 Making the Rig Dynamic

The dynamic feature of the rig is implemented as a standalone callback function. It is activated by clicking the *Manual Mode* button on the system GUI and allows the user to resize the skeleton, or reposition joints, in the 3D space. The entire callback process is shown in figure 4.8. The callback script can be found in appendix B.3.

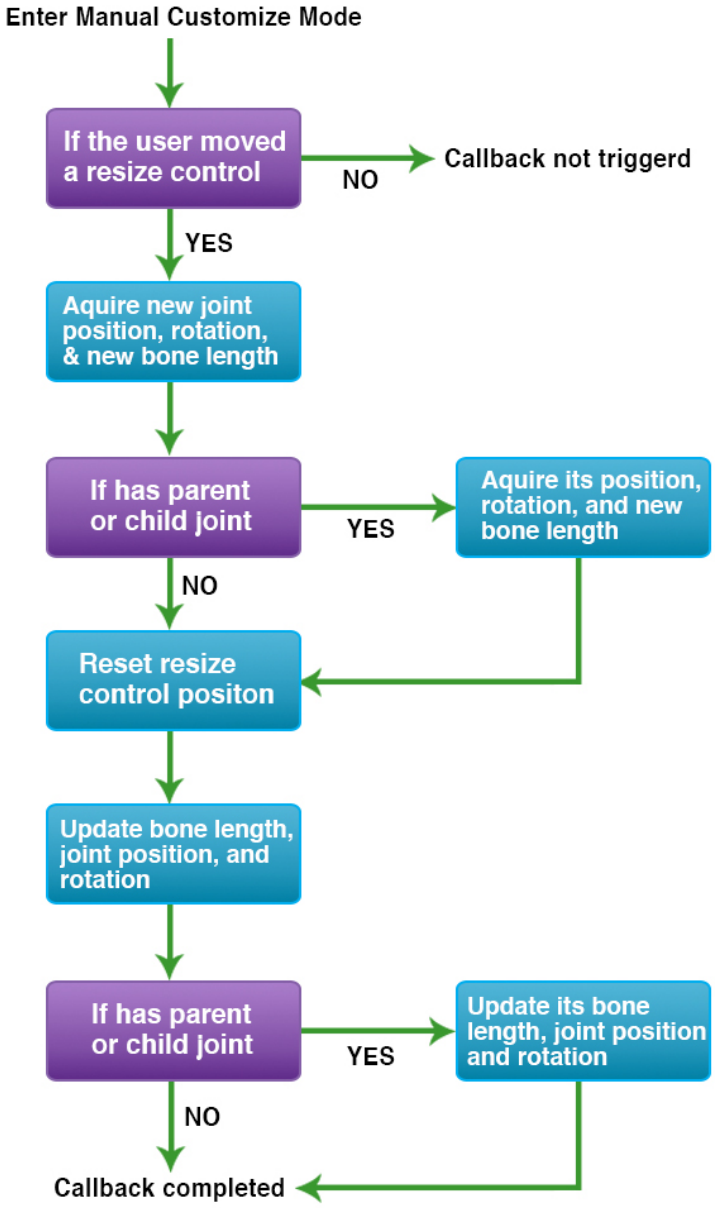

Figure 4.8: The Resize Callback Process

As shown in figure 4.9, the callback function is guided by two sets of guide objects: the resize control located at the center of each joint (visualized as green locators) and the guide joints that form a skeleton structure which is identical to the rig skeleton.

Under normal conditions, resize controls remain hidden and move along with rig joints. Only during manual customization mode can they be seen and manipulated by the user. On the other hand, the position of the guide joint is constrained to the corresponding resize control. The local rotation axis of the guide joint is set up in a way that its X axis is pointing at its child joint. Furthermore, an IK mechanism is built between every two adjacent guide joints so that the orientation of the parent joint can be automatically updated when its child joint is relocated. This keeps the parent joint's X axis always pointing at its child joint. As a result, when the user moves a resize control in 3D space, the corresponding guide joint is able to provide the callback function with new joint position and orientation information.

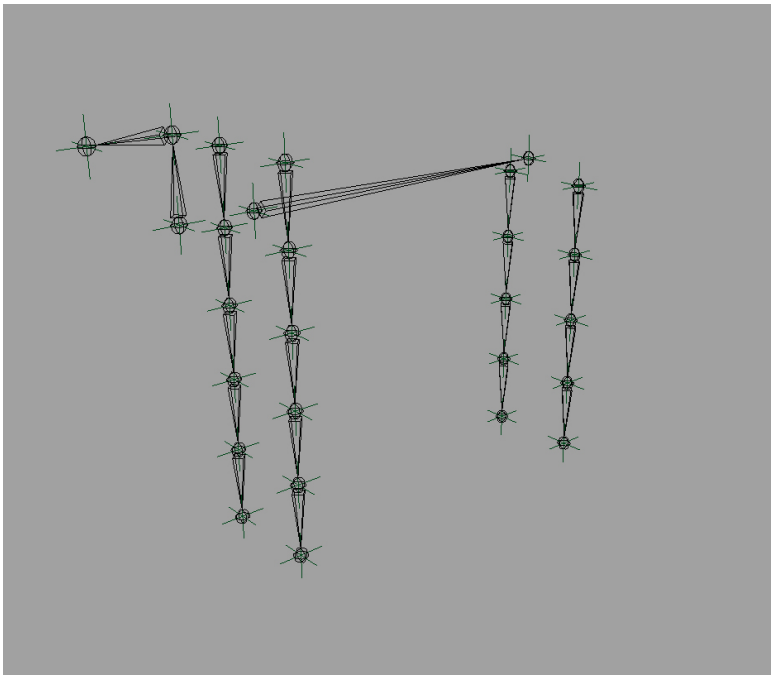

Figure 4.9: Guide Objects for Shape Customization.

The callback function is a linear process triggered and executed when a resize

control is moved to a new location. It queries the new joint position and orientation as discussed above, as well as the new bone length. The new bone length is acquired thought a series of measuring tools built into the quadruped rig. If the joint has a parent or a child joint, the same data set is also acquired from them. After all necessary data is obtained, the moved resize control is reset to its original position before the skeleton update process begins.

The bone length is the first one to be updated. It is done by scaling the joint with a multiplier calculated through dividing the new bone length with the original bone length. Secondly, the callback function updates the joint's position and orientation to match the previously acquired new data. Finally, the same update process is conducted on the joint's parent or child joint if they exist.

When the user loads a creature preset or a customized shape, the same function is executed. The only difference is that new joint position data is provided directly by the text file that contains predefined joint positions. In other words, the shape text file serves the same purpose as moving the resize control manually. Figure 4.10 shows the result of loading a custom horse shape.

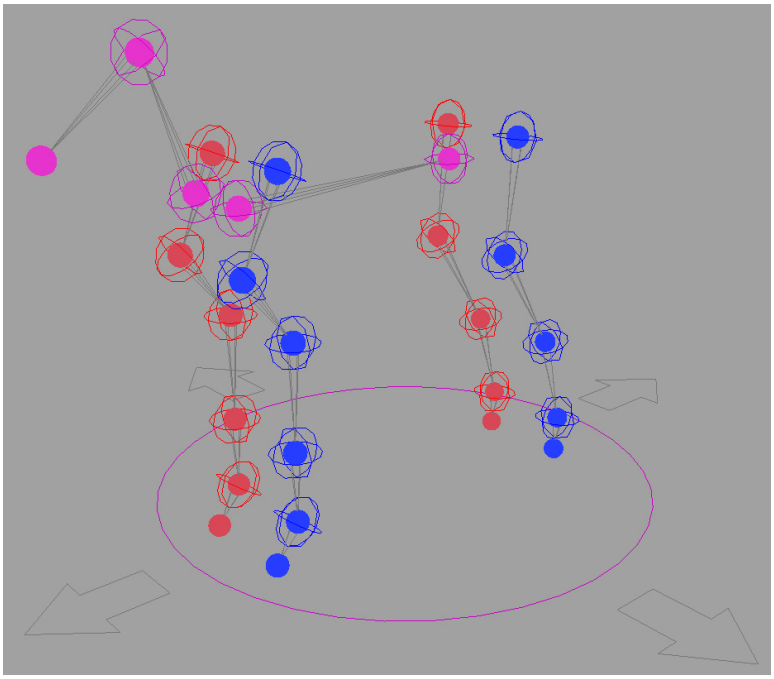

Figure 4.10: The Result of Loading a Custom Horse Shape.

4.5 Constructing the Gait Generator

The gait generator is implemented as a Maya plug-in by utilizing Maya C++ API. It includes two major functions: a data reader function and a compute function. This section is devoted to discussing how they were created and how they work together to generate expressive quadruped motion.

4.5.1 Motion Data Reader

The data reader function scans a specified system folder and reads in all the motion files within that folder one by one. As mentioned in section 3.1, the motion file used in this thesis consists of a series of components required to form the sinusoidal function used to compute joint rotations, such as amplitudes and phases. In order to read in such a motion file, a series of internal array variables is used to store these

components within the plug-in so that they can be queried without reading the same motion files repeatedly.

As a result, the data reader function is executed only once when the gait generator plug-in is loaded. This approach greatly improves the efficiency of the gait generator. The completed motion data reader function is shown in appendix B.4.

4.5.2 The Compute Function

The compute function serves to calculate the final rotation value for each joint on the creature. It first queries current time value and user inputs from the system GUI, including the gait type value, the gait speed values, the two characteristic values, and the mode of the user-controlled speed switch. Then, the gait speed is calculated based upon the current mode of the switch. If the switch is on, the gait speed is determined directly by the three gait speed values. If the switch is off, the gait speed is calculated through a weighted interpolation between the gait speed provided by the motion files. After that, the compute function calculates the joint rotation value for each motion example by computing the result of a key sinusoidal function formed by the components from the corresponding motion file. To enable overall speed adjustment, current time is scaled by the speed multiplier before being applied to the sinusoidal functions.

At this point, all motion examples are calculated and stored independently as joint rotation values. The last step, which is also the most important step, is to interpolate in between these motion examples based on the gait type and characteristic values to compute the final joint rotation value. This process is demonstrated in figure 4.11. The boxes on the left side of the figure represents the twelve motion examples used by this thesis. Within each of the three gait types (walk, trot, and run), one motion example is used to represent each of the four characteristics (light,

heavy, young, and old). A weighted interpolation is first performed between the two characteristics within the same characteristic category. For example, if the user changes the creature's weight to 20 (the entire range is from -50 to 50), then 30% of the light gait and 70% of the heavy gait forms the rotation value for the weight characteristic. Secondly, the function averages the resulted rotation values across two characteristic categories. Eventually, the final rotation value is computed by performing a similar weighted interpolation based on the user-controlled gait type value.

The compute function is executed when the time variable or the user input changes. This allows the user to view and adjust generated motion in real-time. The completed motion data reader function is shown in appendix B.5.

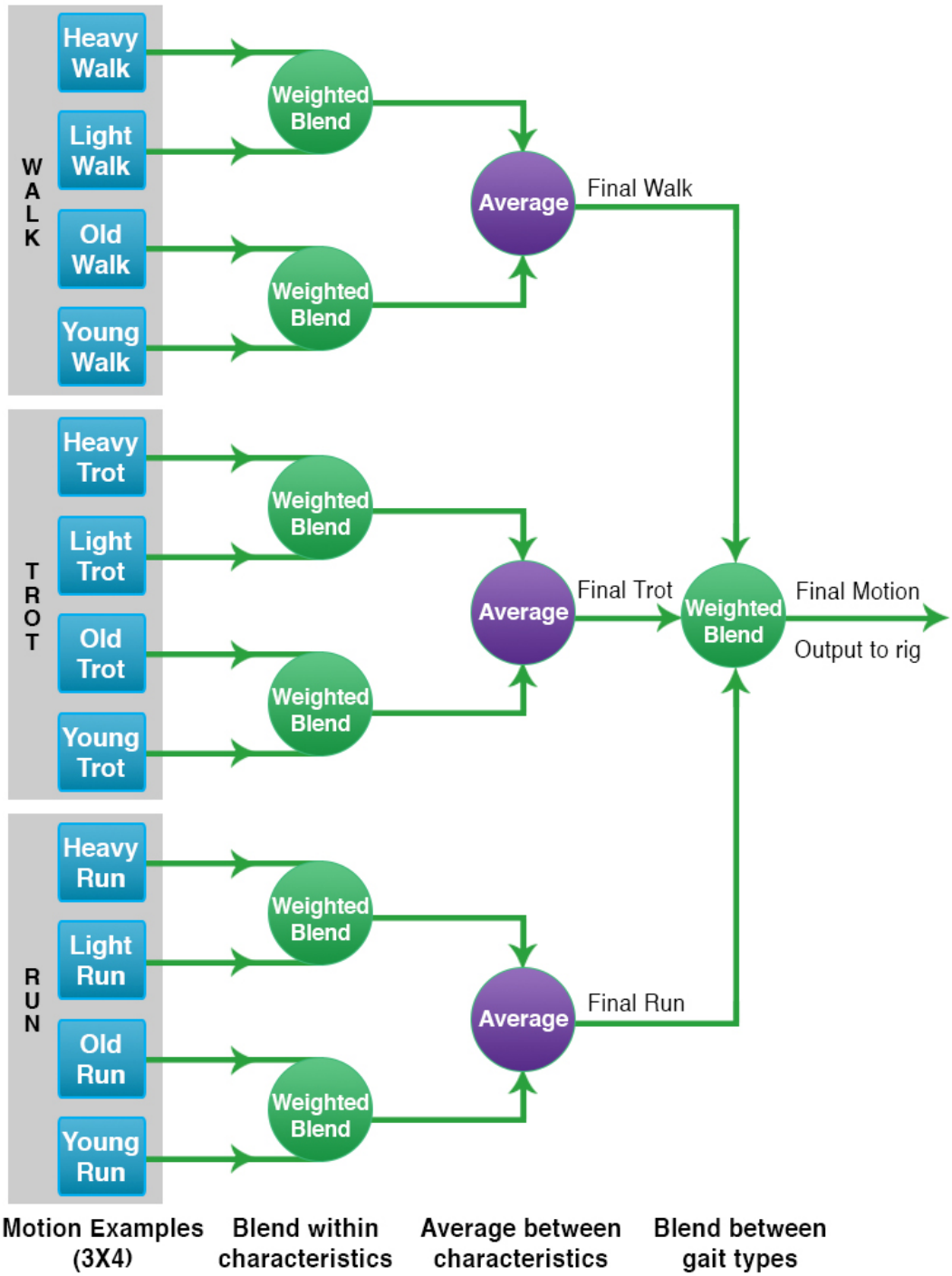

Figure 4.11: The Main Interpolation Process.

4.6 Putting Everything Together

With the system GUI, the dynamic quadruped rig, and the gait generator are completed, the final step is to integrate these three elements into one single system. To fulfill this goal, a system initialization script and a master GUI script are created using Python. When executed in Maya, the initialization script automates the process of creating the quadruped rig, loading the gait generator plug-in, and connecting attributes between the two. Besides, the master GUI script loads the GUI window and button-related functions into Maya. It also forms direct attribute connections between the GUI and the quadruped rig. These two scripts are packaged into a Maya pop-up menu and shelf buttons for easy access. Figure 4.12 shows the system implemented in Maya.

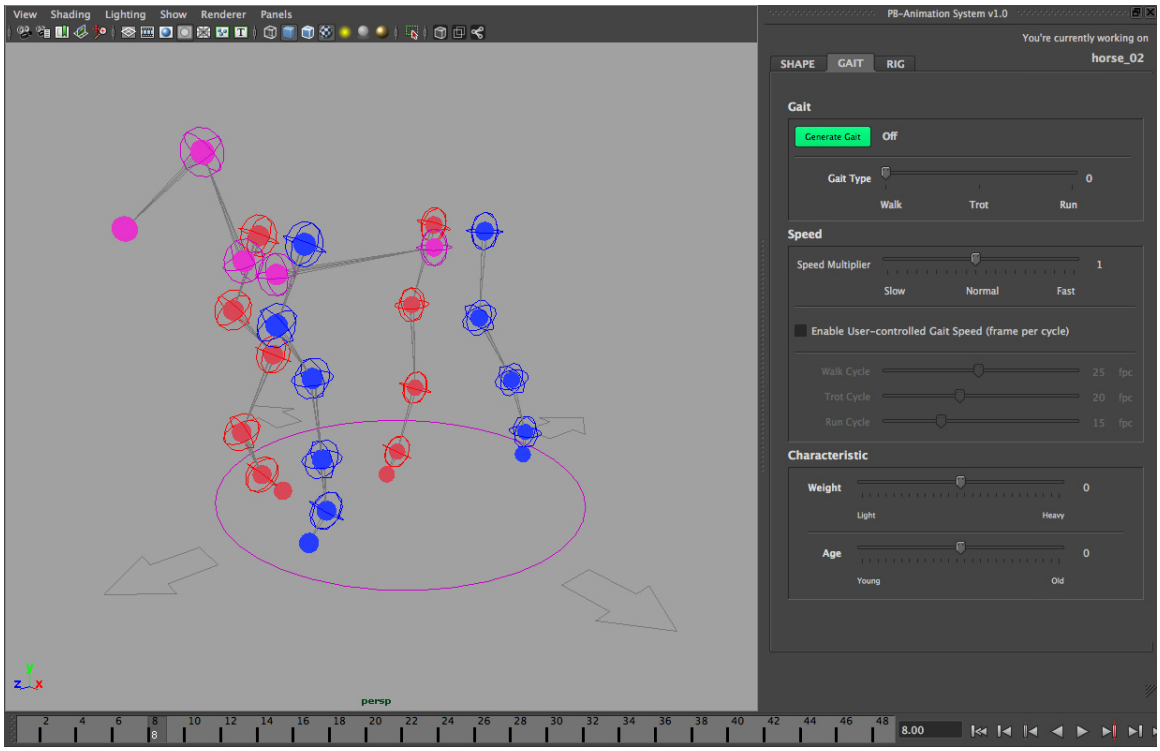

Figure 4.12: The Completed System in Maya.

5. RESULTS AND CONCLUSION

The animation system implemented for this thesis uses a procedural method to simplify the character animation creation process while maximizing the expressiveness of the generated animation by extracting characteristic signals that identify the animal from real animal motion. It is constructed within Autodesk Maya 2012 environment as a dynamic quadruped rig and a gait generator plug-in. The whole system, including its GUI, is embedded into Maya and integrated seamlessly into the Maya workflow, which is in common use in animation, visual effects, and game production.

This system is successful in synthesizing believable quadruped leg locomotion relating to the motion observed in animal video footages. The system is useable by untrained users to create distinctive quadruped shapes and gaits. In other words, the user is able to efficiently construct a unique quadrupedal character and apply a walking, trotting, or running gait onto it that expresses weight and/or age. Those qualities can be modified in real-time. Some of the creature skeletons and motion produced by the system are shown from figure 5.1 to 5.6.

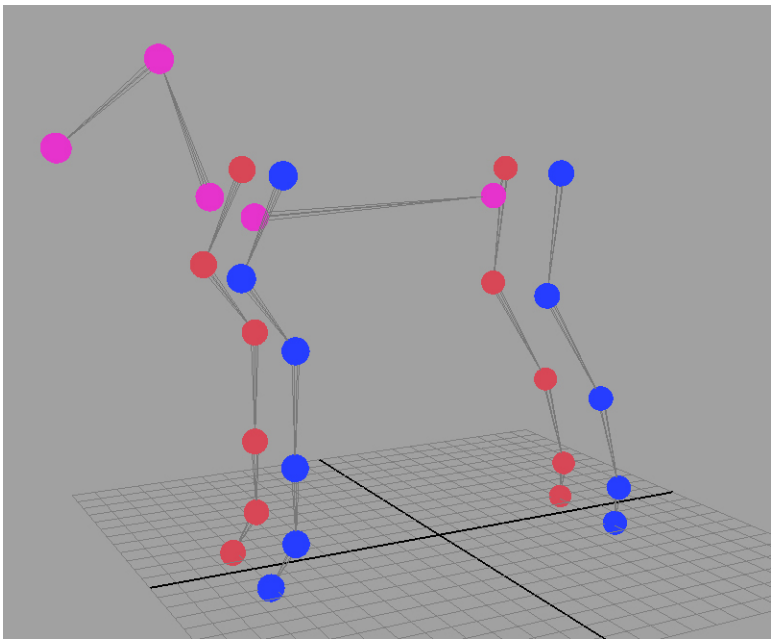

Figure 5.1: Skeleton 1 - A Horse Skeleton.

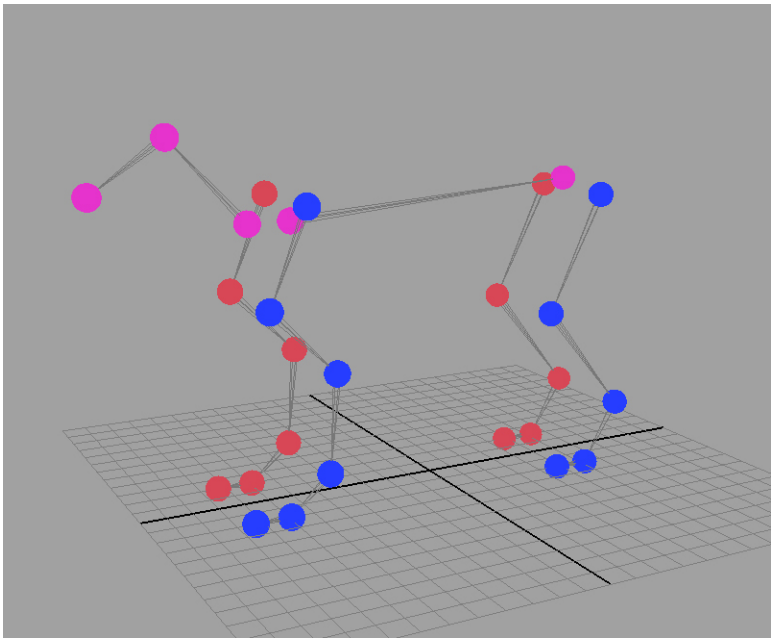

Figure 5.2: Skeleton 2 - A Lion Skeleton.

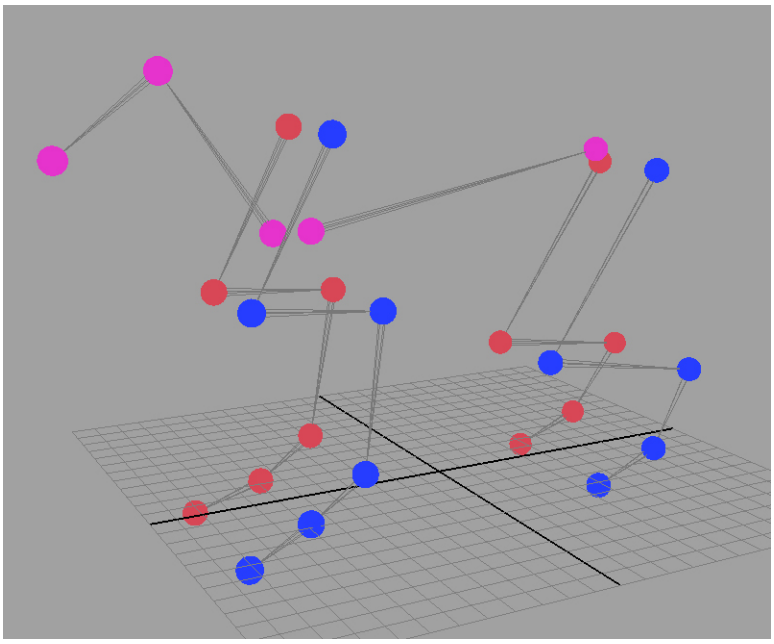

Figure 5.3: Skeleton 3 - A Fantasy Creature Skeleton.

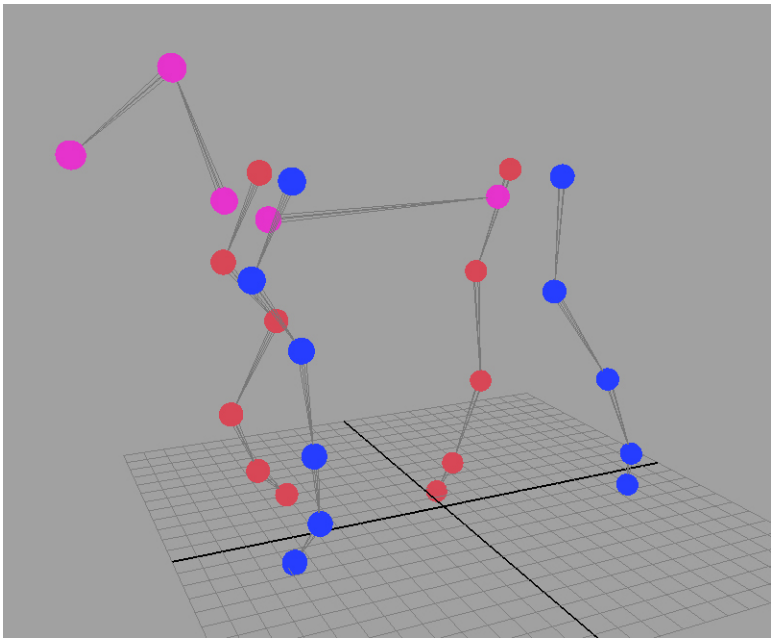

Figure 5.4: Motion Output 1 - A Heavy and Old Walk.

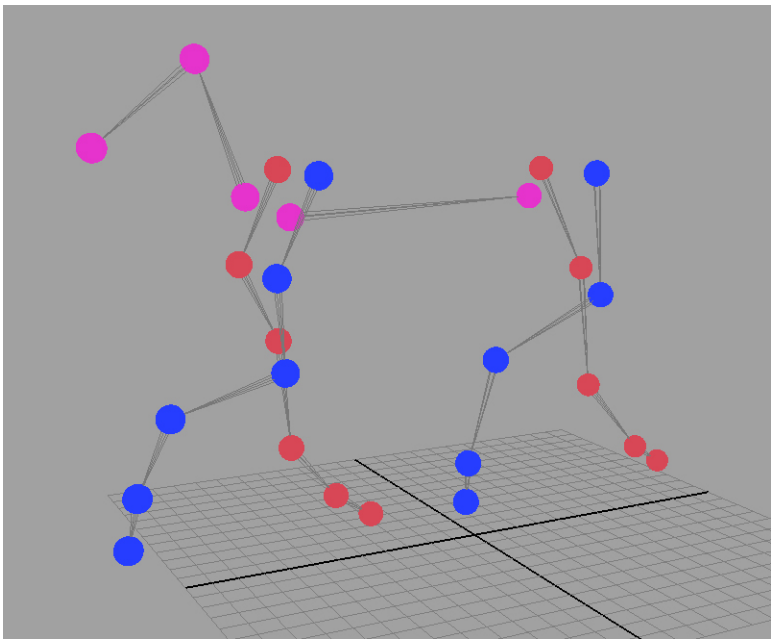

Figure 5.5: Motion Output 2 - A Light and Young Trot.

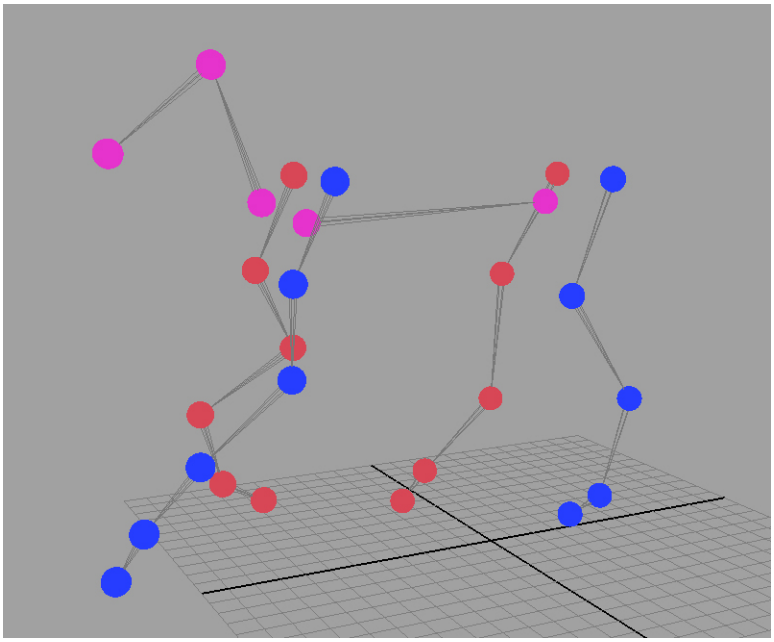

Figure 5.6: Motion Output 3 - A Light and Young Run.

6. FUTURE WORK

6.1 Other Types of Creatures and Traits

As mentioned in section 3.1, this thesis only investigates unguligrade creatures and two types of characteristics, weight and age, that are communicated through motion. It is reasonable to extend the range of study to cover other types of creatures as well, such as digitigrade quadrupeds and reptiles. Traits such as aggressiveness and emotions such as happy and sad can also be included to expand the characteristics involved in this thesis.

6.2 The Third Dimension

As mentioned in section 3.1, only two dimensional motion data is provided for this study due to limitations of the method used to acquire motion data. A more robust system would take the third dimension into consideration. Data related to the third dimension can also be obtained through analysis of real world animal motion.

6.3 Generating Standard Animation Rig

A possible use of this study is to generate cyclic quadruped locomotion for animation production. However, a production may need to manually fine-tune the procedurally generated motion. Although the system built for this thesis contains a FK control layer for this purpose, it is not efficient enough due to the limitations of the FK mechanism. Therefore, a useful extension to the system would be generating a standard IK rig based on the customized quadruped skeleton with the same procedural motion carried over.

6.4 Interaction

This thesis is only focused on producing believable quadruped locomotion. Interactions between multiple creatures and between creatures and the environment are not involved. It is possible to build a physically-based system on top of this study to achieve complex interactions. Including interaction has the potential to produce more realistic and more diverse creature motion.

REFERENCES

[1] Autodesk Inc. *Autodesk Maya*. San Rafael, CA, USA, 2012. Available at http://usa.autodesk.com/maya.

[2] Armin Bruderlin and Thomas W. Calvert. Goal-directed, dynamic animation of human walking. In *Proceedings of the 16td annual conference on computer graphics and interactive techniques*, SIGGRAPH '89, pages 233–242, New York, NY, USA, 1989. ACM.

[3] T.W. Calvert, J. Chapman, and A. Patla. Aspects of the kinematic simulation of human movement. *IEEE Computer Graphics and Applications*, 2:41–50, November 1982.

[4] Spencer Cureton. Using fourier analysis to generate believable gait patterns for virtual quadrupeds, chapter 3-4. Master's thesis, Texas A&M University, College Station, TX, USA, May 2013.

[5] Digia. *Qt Creator*. Helsinki, Finland, 2012. Available at http://qt.digia.com.

[6] Winand H Dittrich, Tom Troscianko, Stephen E G Lea, and Dawn Morgan. Perception of emotion from dynamic point-light displays represented in dance. *Perception*, 25(6):727–738, 1996.

[7] Michael Ford and Alan Lehman. *Inspired 3D Character Setup*, chapter 9. Premier Press, Cincinnati, OH, USA, 1st edition, August 2002.

[8] Michael Girard and A. A. Maciejewski. Computational modeling for the computer animation of legged figures. In *Proceedings of the 12rd annual conference*

on computer graphics and interactive techniques, SIGGRAPH '85, pages 263–270, New York, NY, USA, 1985. ACM.

[9] David A. D. Gould. *Complete Maya Programming: An Extensive Guide to MEL and the C++ API*, volume 1, chapter 1. Morgan-Kaufmann Publishers, San Francisco, CA, USA, 2003.

[10] Paul M. Isaacs and Michael F. Cohen. Controlling dynamic simulation with kinematic constraints. In *Proceedings of the 14td annual conference on computer graphics and interactive techniques*, SIGGRAPH '87, pages 215–224, New York, NY, USA, 1987. ACM.

[11] Gunnar Johansson. Visual perception of biological motion and a model for its analysis. *Perception and Psychophysics*, 14(2):201–211, 1973.

[12] Hyeongseok Ko and Norman I. Badler. Animating human locomotion with inverse dynamics. *IEEE Computer Graphics and Applications*, 16:50–59, March 1996.

[13] Lynn T. Kozlowski and James E. Cutting. Recognizing the sex of a walker from a dynamic point-light display. *Perception and Psychophysics*, 21(6):575–580, 1977.

[14] John Lasseter. Principles of traditional animation applied to 3d computer animation. In *Proceedings of the 14th international conference on computer graphics and interactive techniques*, SIGGRAPH '87, pages 35–44, New York, NY, USA, 1987. ACM.

[15] Meredith McLendon, Ann McNamara, Tim McLaughlin, and Ravindra Dwivedi. Connecting the dots: discovering what's important for creature motion. In *ACM SIGGRAPH 2009: Talks*, SIGGRAPH '09, New York, NY, USA, 2009. ACM.

[16] Johannes J. Michalak, Nikolaus F. Troje, Julia J. Fischer, Patrick P. Vollmar, Thomas T. Heidenreich, and Dietmar D. Schulte. The embodiment of sadness and depression - gait patterns associated with dysphoric mood. *Psychosomatic Medicine*, 71:580–587, 2009.

[17] Rob O'Neill. *Digital Character Development: Theory and Practice*, chapter 5. Elsevier, Burlington, MA, USA, November 2008.

[18] Benito Adán Peña. Performance-guided character bind pose for deformations, page 6. Master's thesis, Texas A&M University, College Station, TX, USA, May 2011.

[19] Meghan P. Provost, Vernon L. Quinsey, and Nikolaus F. Troje. Differences in gait across the menstrual cycle and their attractiveness to men. *Archives of Sexual Behavior*, 37:598–604, 2008.

[20] Python Software Foundation. *Python*. Available at http://python.org.

[21] Paul Richard. *Robot Manipulators: Mathematics, Programming, and Control: The Computer Control of Robot Manipulators*, chapter 2. MIT Press, Cambridge, MA, USA, 1981.

[22] Barbara Robertson. Mike, the talking head. *Computer Graphics World*, pages 15–17, July 1988.

[23] Herbert Schildt. *C++: The Complete Reference*, chapter 11. Mcgraw-Hill Osborne Media, San Francisco, CA, USA, 3rd edition, January 2003.

[24] Thomas F. Shipley and Jonathan S. Brumberg. *Markerless Motion-capture for Point-light Displays*, 2004. Available at http://astro.temple.edu/~tshipley/mocap.html.

[25] Frank Thomas and Ollie Johnston. *The Illusion of Life: Disney Animation*, chapter 3. Hyperion, New York, NY, USA, 1981.

[26] Nikolaus F. Troje. Decomposing biological motion: A framework for analysis and synthesis of human gait patterns. *Journal of Vision*, 2:371–387, 2002.

[27] Graham Walters. The story of waldo c. graphic. In *ACM SIGGRAPH 1989 Courses*, SIGGRAPH '89, Boston, MA, USA, 1989. ACM.

[28] Cord Westhoff and Nikolaus F. Troje. The inversion effect in biological motion perception: Evidence for a "life detector"? *Current Biology*, 16(8):821–824, 2006.

[29] Cord Westhoff and Nikolaus F. Troje. Kinematic cues for person identification from biological motion. *Perception and Psychophysics*, 69(2):241–253, 2007.

[30] David Zeltzer. Motor control techniques for figure animation. *IEEE Computer Graphics and Applications*, 2:53–59, September 1982.

APPENDIX A

FILE STRUCTURE

A.1 An Example of Motion Data File (Partial)

```
gaitSpeed 28.0
sideShift 13.0
#
shld_l_JNT
amp 0.418829208188 0.418895224287 ... 0.0777159882367 0.0768972420591
phase 3.13554477366 3.12949413884 ... 0.292301164687 0.291238389801
bias_init 0.91228102
t_shift 0.70833333
bias_fix 0.0893934537304
scalar_fix 0.491588813919
Ts 0.000249226888021
#
elbow_l_JNT
amp 3.31188289846 3.31268818924 ... 0.317559471586 0.310555886268
phase -0.00528405253926 -0.010573286896 ... -2.96311377224 -3.05369559847
bias_init 4.15541984
t_shift 0.45833333
bias_fix 0.237712699948
scalar_fix 0.687669784759
Ts 0.000178019205729
#
fFoot_l_JNT
amp 0.252875238719 0.252901497563 ... 0.0449747035578 0.0445147790109
phase 3.14072508903 3.13985706655 ... 0.0431395820147 0.0429736862485
bias_init 2.59282611
t_shift 0.3333333333
bias_fix -0.562946448585
scalar_fix 0.491067003711
Ts 0.000249226888021
#
fBall_l_JNT
amp 0.780617022809 0.780855989407 ... 0.160284842809 0.158493615133
phase -3.12995141802 -3.11830718496 ... -0.525018522765 -0.523346939904
bias_init 3.25784688
```

```
t_shift 0.5
bias_fix -0.211659444817
scalar_fix 0.487962128767
Ts 0.000249226888021
#
...
```

A.2 An Example of Shape Text File (Partial)

```
l|fLeg|0|1|0
tx:2.0
ty:14.7611550148
tz:7.3177832011

l|fLeg|1|0|0
tx:2.0
ty:11.3289396686
tz:8.91948369596

l|fLeg|2|0|0
tx:2.0
ty:8.6160001359
tz:6.97156484157

l|fLeg|3|0|0
tx:2.0
ty:4.34279081795
tz:7.06849824942

l|fLeg|4|0|0
tx:2.0
ty:1.47710086515
tz:7.08085323626

l|fLeg|5|0|1
tx:2.0
ty:-0.0170557639454
tz:8.09294564418

...
```

APPENDIX B

SAMPLE CODE

B.1 The Save Current Shape Function

```
def saveShape(self, *args):
        gaitState = mc.getAttr(self.allctl+'.generateGait')
        if gaitState == 1: self.gaitSwitch(gaitState) #### if generating gait, then
            stop animation

        ### get input name and test if it's valid
        sn = mc.textField(self.shapeListPath+'|name_LE', q = 1, text = 1)
        if sn == '' or sn.find(' ') != -1:
                mc.confirmDialog(t = 'Warning', m = 'Please type in a valid name! (No
                    space in name)', button = 'Got it', defaultButton = 'Got it',
                    dismissString = 'Got it')
                return

        ### append name to the scroll list
        mc.setAttr(self.allctl+'.currentShape', sn, type='string')
        mc.textScrollList(self.shapeListPath+'|shape_LIST', e=1, append=sn)
        mc.textScrollList(self.shapeListPath+'|shape_LIST', e=1, selectItem=sn)

        ### open output file
        outfile = open(self.shapePath+sn+'.txt', 'w')
        for part in ['fLeg', 'hLeg', 'spine', 'head']:
                for side in ['l', 'r']:
                        if part == 'spine' or part == 'head': side = 'c'
                        for i in range (0, len(quad[part]), 1):
                                ##### calculate joint position in character space
                                mRscWorld = mc.getAttr(refn + quad[part][i] + '_' + side
                                    + '_' +' RSC.parentMatrix[0]')
                                pRscWorld = sj.vectorXmatrix([0,0,0], mRscWorld, 1)
                                mGlobal =
                                    mc.getAttr(self.allctl+'.worldInverseMatrix[0]')
                                pRscGlobal = sj.vectorXmatrix(pRscWorld, mGlobal, 1)
                                #### wirte side, part, index, hasParent flag and
                                    hasChild flag
                                outfile.write(side)
```

```
                        outfile.write('|')
                        outfile.write(part)
                        outfile.write('|')
                        outfile.write(str(i))
                        outfile.write('|')
                        if i == 0: outfile.write(str(1))
                        else: outfile.write(str(0))
                        outfile.write('|')
                        if i == len(quad[part])-1: outfile.write(str(1))
                        else: outfile.write(str(0))
                        outfile.write('\n')

                        j = 0
                        ##### write position attribute name & value
                        for ax in ['x', 'y', 'z']:
                                outfile.write('t'+ax)
                                outfile.write(':')
                                outfile.write( str(pRscGlobal[j]) )
                                outfile.write('\n')
                                j = j+1
                        outfile.write('\n')
                if part == 'spine' or part == 'head': break

        mc.textField(self.shapeListPath+'|name_LE', e = 1, text = '') #### reset name
            text field
        if gaitState == 1: self.gaitSwitch(1-gaitState) #### if was generating gait
            before saving, then restart motion
```

```
class legCTL:
	def __init__(self):
		self._utlCTL = rgc.utlCTL()

	def setup (self, sd, part):
		###### build fk contorls
		for i in range (0, len(quad[part]), 1):
			partN = quad[part][i]+'_'+sd+'_'
			ctlN = partN+ctl


			self._utlCTL.ballCTL(1, 1, ctlN, ctlN.replace(ctl, jnt), sd,
			    'none')
			mc.group(ctlN, n = ctlN.replace(ctl, ctl+pro))
			mc.group(ctlN.replace(ctl, ctl+pro), n = ctlN.replace(ctl,
			    'RotOffset'))

			mc.select(ctlN, r=1)
			mc.addAttr (ln = 'initialOffset', at = 'double3')
			for ax in ['X', 'Y', 'Z']:
				mc.addAttr (ln = 'initialOffset'+ax, p =
				    'initialOffset', at = 'double')
			mc.setAttr (ctlN+'.initialOffset', lock = 0, keyable = 0,
			    channelBox = 0)
			offset = mc.getAttr(partN+'RotOffset.rotate')
			mc.setAttr(ctlN+'.initialOffset', offset[0][0], offset[0][1],
			    offset[0][2], 'double3')

			if i == 0: mc.parent(ctlN+grp, 'fkCtl_'+grp)
			else: mc.parent(ctlN+grp,quad[part][i-1]+'_'+sd+'_'+ctl)

		###### build resize contorls
		for i in range (0, len(quad[part]), 1):
			partN = quad[part][i]+'_'+sd+'_'
			rscN = partN+rsc
			self._utlCTL.locCTL(1, 1, rscN, rscN.replace(rsc, jnt), sd,
			    'none')
			mc.parent(rscN+grp, part+'_'+sd+'_'+rsc+grp)
			mc.pointConstraint(partN+jnt, rscN+grp, mo=1, w=1)
```

```
                ##### add iks on guide joints
                for i in range (0, len(quad[part])-1, 1):
                        partN = quad[part][i]+'_'+sd+'_'
                        rscN = partN+rsc
                        ikn = mc.ikHandle (n = partN+'Ref'+ikh, sj = partN+'Ref'+jnt,
                            ee = quad[part][i+1]+'_'+sd+'_'+'Ref'+jnt, sol =
                            'ikSCsolver', s = 0)
                        mc.parent(ikn[0], 'ikh_'+grp)
                        mc.pointConstraint(quad[part][i+1]+'_'+sd+'_'+rsc, ikn[0],
                            mo=1, w=1)
                        if i == 0: mc.parentConstraint(rscN, partN+'Ref'+jnt, mo=1, w=1)

                ###### build distance measurement tools
                for i in range (0, len(quad[part]), 1):
                        partN = quad[part][i]+'_'+sd+'_'

                        if i != (len(quad[part]) - 1):
                                self._utlCTL.disTool(sd, partN+rsc,
                                    quad[part][i+1]+'_'+sd+'_'+rsc, quad[part][i])
                        mc.parentConstraint(partN+ctl, partN+jnt, mo=1, w=1)

legctl = legCTL()
legctl.setup('l', 'fLeg')
legctl.setup('r', 'fLeg')
legctl.setup('l', 'hLeg')
legctl.setup('r', 'hLeg')
legctl.setup('c', 'head')
legctl.setup('c', 'spine')

###### setup forequarter & hindquarter groups
foreOffGrp = mc.group (em=1, n = 'foreQuarter_c_PosOffset')
foreTopGrp = mc.group(foreOffGrp, n = 'foreQuarter_c_'+ctl+grp)
mc.parentConstraint('chest_c_'+ctl, foreTopGrp, mo=0, w=1)
mc.delete(foreTopGrp+'_parentConstraint1')
mc.parent(foreTopGrp, 'fkCtl_'+grp)
mc.pointConstraint('chest_c_'+ctl, foreTopGrp, mo=1, w=1)

hindOffGrp = mc.group (em=1, n = 'hindQuarter_c_PosOffset')
hindTopGrp = mc.group(hindOffGrp, n = 'hindQuarter_c_'+ctl+grp)
mc.parentConstraint('hip_c_'+ctl, hindTopGrp, mo=0, w=1)
mc.delete(hindTopGrp+'_parentConstraint1')
```

```
mc.parent(hindTopGrp, 'fkCtl_'+grp)
mc.pointConstraint('hip_c_'+ctl, hindTopGrp, mo=1, w=1)

for sd in ['l', 'r']:
        mc.parent('shldBlade_'+sd+'_'+ctl+grp, foreOffGrp)
        mc.parent('hip_'+sd+'_'+ctl+grp, hindOffGrp)
mc.parent('neck_c_'+ctl+grp, foreOffGrp)
```

```
##### script jobs
def resize(side, part, index, hasParent, hasChild, ready, pMeTargetWorld):
        ###################################
        #### PREPARE NECESSARY DATA
        index = int(index)
        bhasParentase = int(hasParent)
        hasChild = int(hasChild)
        ready = int(ready)

        bLen = 3
        if part == 'spine': bLen = 14

        #### get name prefix
        if hasParent != 1: parent = refn+quad[part][index-1]+'_'+side+'_'
        if hasChild != 1: child = refn+quad[part][index+1]+'_'+side+'_'
        me = refn+quad[part][index]+'_'+side+'_'

        #### get CHILD joint world position
        if hasChild != 1:
                pChildTargetLocal = mc.getAttr(child+'RSC.t')
                mChildTargetWorld = mc.getAttr(child+'RSC.parentMatrix[0]')
                pChildTargetWorld = vectorXmatrix(pChildTargetLocal[0],
                    mChildTargetWorld, 1)

        #### get ME joint world position (case scriptJob)
        if ready == 1: pMeTargetWorld = getTgtWorldPos_scriptJob(side, part, index)

        #### get forequarter & hindquarter world positions (if working on spine
            joints)
        if part == 'spine':
                pForeQTargetLocal = mc.getAttr(refn+'foreQuarter_c_PosOffset.t')
                mForeQTargetWorld =
                    mc.getAttr(refn+'foreQuarter_c_PosOffset.parentMatrix[0]')
                pForeQTargetWorld = vectorXmatrix(pForeQTargetLocal[0],
                    mForeQTargetWorld, 1)

                pHindQTargetLocal = mc.getAttr(refn+'hindQuarter_c_PosOffset.t')
                mHindQTargetWorld =
                    mc.getAttr(refn+'hindQuarter_c_PosOffset.parentMatrix[0]')
```

```
        pHindQTargetWorld = vectorXmatrix(pHindQTargetLocal[0],
            mHindQTargetWorld, 1)

#### get ME, PARENT, and CHILD joint local rotations
if ready == 0:
        mMeTargetInv = mc.getAttr(me+'RSC.parentInverseMatrix[0]')
        pMeTarget = vectorXmatrix(pMeTargetWorld, mMeTargetInv, 1)
        mc.setAttr(me+'RSC.t', pMeTarget[0], pMeTarget[1], pMeTarget[2],
            'double3')
rMeRefJNT = mc.getAttr(me+'RefJNT.r')
if hasParent != 1: rParentRefJNT = mc.getAttr(parent+'RefJNT.r')
if hasChild != 1: rChildRefJNT = mc.getAttr(child+'RefJNT.r')

###################################
#### RESIZE PROCESS BEGINS
mc.setAttr(me+'RSC.t', 0, 0, 0, 'double3')

#### update PARENT rotation
if hasParent != 1: mc.setAttr(parent+'RotOffset.r', rParentRefJNT[0][0],
    rParentRefJNT[0][1], rParentRefJNT[0][2], 'double3')
#### update ME position & rotation
updateXForm(me+'CTLGRP', pMeTargetWorld, 1)
mc.setAttr(me+'RotOffset.r', rMeRefJNT[0][0], rMeRefJNT[0][1],
    rMeRefJNT[0][2], 'double3')
#### update CHILD position & rotation
if hasChild != 1:
        updateXForm(child+'CTLGRP', pChildTargetWorld, 1)
        mc.setAttr(child+'RotOffset.r', rChildRefJNT[0][0],
            rChildRefJNT[0][1], rChildRefJNT[0][2], 'double3')
#### If adjusting spine, update forequarter & hindquarter positions
if part == 'spine':
        updateXForm('foreQuarter_c_PosOffset', pForeQTargetWorld, 1)
        updateXForm('hindQuarter_c_PosOffset', pHindQTargetWorld, 1)

#### update initial offsets
updateInitOffset(side, part, index)
if hasParent != 1: updateInitOffset(side, part, index-1)
if hasChild != 1: updateInitOffset(side, part, index+1)

if ready == 1: rsReady(side, part, str(index), str(hasParent), str(hasChild),
    str(ready))
```

```
##### main functions
def go():
	for part in ['fLeg', 'hLeg', 'head', 'spine']:
		for index in range (0, len(n.quad[part]), 1):
			for side in ['l', 'r']:
				if part == 'spine' or part == 'head': side = 'c'
				base = 0
				end = 0
				if index == 0: base = 1
				if index == (len(n.quad[part]) - 1): end = 1
				rsReady(side, part, str(index), str(base), str(end),
				    str(1))
				if part == 'spine' or part == 'head': break
def stop():
	mc.scriptJob(killAll = True)
```

```
string gaitGenerator::getDataPath()
{
    char * ePathP;
    char ePath[128];
    char * projPath;

    string dataPathString;
    size_t found;

    /// get environment variables
    ePathP = getenv ("PYTHONPATH");
    strcpy (ePath, ePathP);
    projPath = strtok (ePath,":");

    // search for the environment variable contains "gaitGen"
    while (projPath != NULL)
    {
        dataPathString = string(projPath);
        found=dataPathString.find("gaitGen");
        if (found!=string::npos) break;
        else projPath = strtok (NULL, ":");
    }

    //replace "include/" with "motionData/" to get correct data path
    dataPathString = dataPathString.substr(0, dataPathString.size()-8);
    dataPathString.append("motionData/");

    return dataPathString;
}

MStatus gaitGenerator::readData(string dataPathString)
{
    MStatus status;

    // convert data path from string to char
    char dataPathChar[dataPathString.size()];
    memcpy(dataPathChar, dataPathString.c_str(), dataPathString.size()+1);

    // open data path
```

```
DIR *dir;
struct dirent *ent;
dir = opendir (dataPathChar);

// check if path is valid
if (dir == NULL)
{
    return MS::kFailure;
}

int fileNum=0, jntNum, i, j, lineCount, caseNum;
char fileName[80];
ifstream motionData;
string s, line, value;
size_t foundFile, found;

// iterate files one by one
while ( (ent = readdir (dir)) != NULL )
{
    // convert current path to string and check if it is a valid file name
    s = string(ent->d_name);
    foundFile=s.find(".txt");
    if (foundFile == string::npos) continue;

    // form correct path pointing to current file (file path = data folder path +
        file name)
    sprintf(fileName, "%s%s", dataPathChar, ent->d_name);
    motionData.open(fileName);

    // Count lines in current file
    lineCount = count(istreambuf_iterator<char>(motionData),
        istreambuf_iterator<char>(), '\n');
    lineCount ++;
    motionData.seekg (0, ios::beg); // set line pointer back to the first line

    // Read in motion data from current file
    jntNum = 0; // reset joint #, VERY IMPORTANT - otherwise will cause memory
        overrun
    for (i = 0; i < lineCount; i++)
    {
        motionData>>line;
```

```
found=line.find(".");
while (found!=string::npos)
{
    motionData>>line;
    found=line.find(".");
}

if (line == "amp") caseNum = 1;
else if (line == "phase") caseNum = 2;
else if (line == "bias_init") caseNum = 3;
else if (line == "t_shift") caseNum = 4;
else if (line == "bias_fix") caseNum = 5;
else if (line == "scalar_fix") caseNum = 6;
else if (line == "Ts") caseNum = 7;
else if (line == "gaitSpeed") caseNum = 8;
else caseNum = 0;

switch ( caseNum )
{
    case( 0 ):
        break;

    case( 1 ):
        for (j = 0; j < numComp; j++)
        {
            motionData>>value;
            amp[fileNum][jntNum][j] = ::atof(value.c_str());
        }
        break;

    case( 2 ):
        for (j = 0; j < numComp; j++)
        {
            motionData>>value;
            phase[fileNum][jntNum][j] = ::atof(value.c_str());
        }
        break;

    case( 3 ):
        motionData>>value;
        bias_init[fileNum][jntNum] = ::atof(value.c_str());
```

```
                bias_init[fileNum][jntNum] = bias_init[fileNum][jntNum]*180/PI;
                break;

            case( 4 ):
                motionData>>value;
                t_shift[fileNum][jntNum] = ::atof(value.c_str());
                break;

            case( 5 ):
                motionData>>value;
                bias_fix[fileNum][jntNum] = ::atof(value.c_str());
                break;

            case( 6 ):
                motionData>>value;
                scalar_fix[fileNum][jntNum] = ::atof(value.c_str());
                break;

            case( 7 ):
                motionData>>value;
                Ts[fileNum][jntNum] = ::atof(value.c_str());
                Fs[fileNum][jntNum] = 1./Ts[fileNum][jntNum];
                omega_delta[fileNum][jntNum] = 2.0*PI*(Fs[fileNum][jntNum]/Npts);
                jntNum++;
                break;

            case( 8 ):
                motionData>>value;
                gaitSpeed[fileNum] = ::atof(value.c_str());
                break;
        }
    }

    motionData.close();
    //motionData.clear();
    fileNum++;
}
fileCount = fileNum;

rewinddir(dir);
closedir (dir);
```

```
    return MS::kSuccess;
}
```

B.5 The Compute Function in Gait Generator

```
MStatus gaitGenerator::compute(const MPlug& plug, MDataBlock& data)
{
    MStatus status;

    //// Get input attributes
    bool gaitSwitch = data.inputValue(a_gaitSwitch, &status).asBool();
    bool lockSpeed = data.inputValue(a_lockSpeed, &status).asBool();
    float speedMultiplier = data.inputValue(a_speedMultiplier, &status).asFloat();
    float walkSpeedUsr = data.inputValue(a_walkSpeedUsr, &status).asFloat();
    float trotSpeedUsr = data.inputValue(a_trotSpeedUsr, &status).asFloat();
    float runSpeedUsr = data.inputValue(a_runSpeedUsr, &status).asFloat();
    float gaitType = data.inputValue(a_gaitType, &status).asFloat();
    float frame = data.inputValue(a_frame, &status).asFloat();

    float weight = data.inputValue(a_weight, &status).asFloat();
    float age = data.inputValue(a_age, &status).asFloat();
    float danger = data.inputValue(a_danger, &status).asFloat();

    float shldBladeOffset = data.inputValue(a_shldBladeOffset, &status).asFloat();

    ////////////////////////////////////////
    /////// Compute walk, trot, and run speed
    float walkSpeed, trotSpeed, runSpeed;
    float walkWeight, walkAge, walkFinal;
    float trotWeight, trotAge, trotFinal;
    float runWeight, runAge, runFinal;
    float Rfinal;
    float jointRotOffset;

    /// set up indices
    int iHeavyWalk = 6;
    int iLightWalk = 8;
    int iOldWalk = 7;
    int iYoungWalk = 8;

    int iHeavyTrot = 4;
    int iLightTrot = 5;
    int iOldTrot = 3;
    int iYoungTrot = 5;
```

```
int iHeavyRun = 0;
int iLightRun = 2;
int iOldRun = 0;
int iYoungRun = 1;

/// normalize weight for 2 characteristics
float wWeight = (weight+50)/100;
float wAge = (age+50)/100;

/// Calculate cycle speed
if (lockSpeed)
{
    walkSpeed = walkSpeedUsr;
    trotSpeed = trotSpeedUsr;
    runSpeed = runSpeedUsr;
}
else
{
    /// WALK Interpolation
    float walkSpeed_weight = wWeight*gaitSpeed[iHeavyWalk] +
        (1-wWeight)*gaitSpeed[iLightWalk];
    float walkSpeed_age = wAge*gaitSpeed[iOldWalk] +
        (1-wAge)*gaitSpeed[iYoungWalk];
    walkSpeed = ( (walkSpeed_weight + walkSpeed_age)/2 ) / speedMultiplier;

    /// TROT Interpolation
    float trotSpeed_weight = wWeight*gaitSpeed[iHeavyTrot] +
        (1-wWeight)*gaitSpeed[iLightTrot];
    float trotSpeed_age = wAge*gaitSpeed[iOldTrot] +
        (1-wAge)*gaitSpeed[iYoungTrot];
    trotSpeed = ( (trotSpeed_weight + trotSpeed_age)/2 ) / speedMultiplier;

    /// RUN Interpolation
    float runSpeed_weight = wWeight*gaitSpeed[iHeavyRun] +
        (1-wWeight)*gaitSpeed[iLightRun];
    float runSpeed_age = wAge*gaitSpeed[iOldRun] + (1-wAge)*gaitSpeed[iYoungRun];
    runSpeed = ( (runSpeed_weight + runSpeed_age)/2 ) / speedMultiplier;
}

////////////////////////////////////////////
/////// Compute y_est
```

```
int fileNum, ii, jntNum, side;
float time, calSpeed, calSideShift, fShldTS;
float y_est[9][2][8] = {0.0}; ////////////// fileTotal couldn't be replaced at here
                            ////////////// format: y_est[Total File #][Two
                                Sides][Total Joint #]

time = frame/24.0; //converiting frame value to real-time value. ASSUMING 1 sec =
    24 frames
if (gaitSwitch)
{
    for (fileNum = 0; fileNum < fileCount; fileNum++)
    {
        /// Get the speed type for main calculation
        if (fileNum < 3)
        {
            calSpeed = runSpeed;
        }
        if (fileNum >=3 && fileNum < 6)
        {
            calSpeed = trotSpeed;
        }
        if (fileNum >= 6)
        {
            calSpeed = walkSpeed;
        }

        /// Grab the shoulder joint's t_shift, it is used to unify the starting
            point of signals across all motion files
        fShldTS = t_shift[fileNum][1];

        for (side=0; side < 2; side++)
        {
            /// scale side shift to match the current cycle speed
            if (side == 0)
            {
                calSideShift = 0;
            }
            else
            {
                calSideShift =
                    (sideShift[fileNum]/24)*(calSpeed/gaitSpeed[fileNum]); //// if
```

```
        calculating the right side, conduct time shift
}

for (jntNum=0; jntNum < jointTotal; jntNum++)
{
    //// form the main cosine function
    for (ii=1; ii < numComp+1; ii++)
    {
        y_est[fileNum][side][jntNum] = y_est[fileNum][side][jntNum] +
            amp[fileNum][jntNum][ii-1] *
            cos(omega_delta[fileNum][jntNum] * (ii) *
            (gaitSpeed[fileNum]/calSpeed )*
            (time-t_shift[fileNum][jntNum]-fShldTS + calSideShift) +
            phase[fileNum][jntNum][ii-1]);
    }
    // apply fixing values
    y_est[fileNum][side][jntNum] = (1.0/(2*Fs[fileNum][jntNum])) *
        y_est[fileNum][side][jntNum];
    y_est[fileNum][side][jntNum] = y_est[fileNum][side][jntNum] -
        bias_fix[fileNum][jntNum];
    y_est[fileNum][side][jntNum] = y_est[fileNum][side][jntNum] *
        scalar_fix[fileNum][jntNum];
    y_est[fileNum][side][jntNum] = y_est[fileNum][side][jntNum] *
        (180.0/PI);
    y_est[fileNum][side][jntNum] = y_est[fileNum][side][jntNum] +
        bias_init[fileNum][jntNum];

    //// correct joint roation offset
    switch (jntNum)
    {
        case 0 :
            jointRotOffset = 90 + shldBladeOffset;
            break;

        case 4 :
            jointRotOffset = 90;
            break;

        default :
            jointRotOffset = 180;
            break;
```

```
                }
                y_est[fileNum][side][jntNum] = y_est[fileNum][side][jntNum] -
                    jointRotOffset;
            }
        }
    }
}

//////////////////////////////////////////
/// Open Output attributes
MArrayDataHandle h_leg_l_RZ = data.outputArrayValue( a_leg_l_RZ, &status);
CHECK_MSTATUS_AND_RETURN_IT (status);
MArrayDataHandle h_leg_r_RZ = data.outputArrayValue( a_leg_r_RZ, &status);
CHECK_MSTATUS_AND_RETURN_IT (status);

//////////////////////////////////////////
// Coumpute final rotation value for each joint and set output attributes
for (jntNum = 0; jntNum < jointTotal; jntNum++)
{
    for (side=0; side < 2; side++)
    {
        /////// WALK interpolation
        walkWeight = wWeight*y_est[iHeavyWalk][side][jntNum] +
            (1-wWeight)*y_est[iLightWalk][side][jntNum];
        walkAge = wAge*y_est[iOldWalk][side][jntNum] +
            (1-wAge)*y_est[iYoungWalk][side][jntNum];
        walkFinal = (walkWeight+walkAge)/2;

        /////// TROT interpolation
        trotWeight = wWeight*y_est[iHeavyTrot][side][jntNum] +
            (1-wWeight)*y_est[iLightTrot][side][jntNum];
        trotAge = wAge*y_est[iOldTrot][side][jntNum] +
            (1-wAge)*y_est[iYoungTrot][side][jntNum];
        trotFinal = (trotWeight+trotAge)/2;

        /////// RUN interpolation
        runWeight = wWeight*y_est[iHeavyRun][side][jntNum] +
            (1-wWeight)*y_est[iLightRun][side][jntNum];
        runAge = wAge*y_est[iOldRun][side][jntNum] +
            (1-wAge)*y_est[iYoungRun][side][jntNum];
        runFinal = (runWeight+runAge)/2;
```

```
            //// GAIT TYPE transition
            if (gaitType <= 10)
            {
                Rfinal = (1 - gaitType/10) * walkFinal + (gaitType/10) * trotFinal;
            }
            else
            {
                Rfinal = (1 - (gaitType-10)/10) * trotFinal + ((gaitType-10)/10) *
                    runFinal;
            }

            //// output rotation value
            if (side == 0)
            {
                h_leg_l_RZ.jumpToArrayElement( jntNum );
                h_leg_l_RZ.outputValue().setFloat( Rfinal );
            }
            else
            {
                h_leg_r_RZ.jumpToArrayElement( jntNum );
                h_leg_r_RZ.outputValue().setFloat( Rfinal );
            }
        }
    }

    ////////////////////////////////////////
    /// Set clean output attributes & plug
    h_leg_l_RZ.setAllClean();
    h_leg_r_RZ.setAllClean();
    data.setClean(plug);

    return MS::kSuccess;
}
```