# Motion Reuse, Synthesis and Scripting for Character Animation

*Submitted in partial fulfillment of the requirements*
*for the degree of*

Doctor of Philosophy

by

SHRINATH V. SHANBHAG

Roll No. : 99429701

Supervior:

PROF. SHARAT CHANDRAN


Kanwal Rekhi School of Information Technology

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

2007

*to my parents*

*anuradha & vasudev shanbhag*

# Approval Sheet

Thesis entitled **Motion Reuse, Synthesis and Scripting for Character Animation** by **Shrinath Shanbhag** is approved for the degree of Doctor of Philosophy

Examiners

_____

_____

_____

_____

Supervisor

_____

Chairman

_____

Date: _____

Place: _____

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY, INDIA

# CERTIFICATE OF COURSE WORK

This is to certify that Mr. Shrinath V. Shanbhag was admitted to the candidacy of the Ph.D. Degree on March 27, 2001 after successfully completing all the courses required for the Ph.D. Degree programme. The details of the course work done are given below.

| Sr. No. | **Course Code** | **Course Name** | **Credits** |
|---|---|---|---|
| 1. | CS 622 | Computer Vision | 6 |
| 2. | IT 604 | Human Computer Interaction Design | 6 |
| 3. | IT 606 | Mobile Computing | 6 |
| 4. | IT 620 | Seminar | 4 |
| 5. | IT 621 | Geographic Information System | 6 |
| 6. | IT 641 | Advanced Internet Technologies | 6 |
| 7. | IT 690 | Mini Project | 10 |

I.I.T Bombay Dy. Registrar (Academic)

Date:

## Abstract

Animated humanoid characters are a delight to watch. They find extensive use in a number of interactive computer applications such as movies, games, and strategic and ergonomic simulations. There are four main approaches of generating animation data – key frame animation, procedural animation, behavioral animation and motion capture. Of these, motion capture is a fairly recent development. Nonetheless motion capture has emerged as a popular method because of its speed, robustness and ability to capture rich motion as enacted by live performers. Irrespective of the method selected, creating animation data is a time consuming and costly process.

Institutions and individuals working with animated characters eventually end up possessing a library of motion data. This data has been painstakingly generated over a period of time. The size of this motion database increases monotonically with time. Given the cost and time associated with creating new animation, it is only natural to search for mechanisms that allow reuse of existing animation in new contexts. However motion captured animation data being iconic in nature is difficult to adapt.

Our work focuses on reuse of humanoid character animation data. We describe reuse methods that work with motion captured animation (and other animation exported to mocap compatible formats). The motion editing problems we address are:

1. Identification of transitions between different motion captured actions at arbitrary instants of time - in the absence of explicitly recorded transition sequences
2. Synthesis of looping motion of arbitrary length
3. Extending the richness of the mocap database by synthesizing new motion using motion grafting and
4. Parameterization of walk motions using geometric models to add to variety of such motions on the fly

We integrate our motion reuse techniques together as primitive operations of our scripting language. Our language allows direct access to our techniques and enables their functional composition. Our language allows scripting of actors driven by mocap animation data and narratives with embedded parallel and sequential flows.

**Keywords**: computer graphics, animation, mocap, motion capture, motion editing, motion parameterization, scripting

# Contents

# List of Figures

# Chapter 1

# Introduction

Animated characters have an alluring charm. Ever since they were introduced in animated films, generations have grown up to love and adore these characters. With the advent of computer era, animated characters have entered the digital medium. So successful has been their conquest that animated characters today, find use in applications as diverse as movies, games, strategic simulations, and ergonomics testing.

The animation requirements of a humanoid animated character vary between the extremes of physical realism and cartoony caricature. This does not depend on the application but on the mood or theme desired to be communicated by the creator. Examples of both extremes can be found in movies and games. In either case, the animations have to be believable. Human observers are adept at identifying even small deviations, as they are natural experts by virtue of having watched other people perform such motion all through their life. Creating believable animation, therefore, requires tremendous amount of skill.

There are four main approaches of generating animation data—key-frame animation, procedural animation, behavioral animation and motion capture. Of these, motion capture (*mocap*) is a fairly recent development. Nonetheless mocap has emerged as a popular method because of its speed, robustness and ability to capture rich motion as enacted by live performers. Irrespective of the method selected, creating animation data is a time consuming and costly process.

A big challenge for animators is creating new animations cost effectively and efficiently. Institutions and individuals working with animated characters eventually end up possessing a large library of motion data. This data has been painstakingly generated over a period of time. Even

then, new animation requests are the norm. For example, consider the animation of an actor that performs the actions "*run – jump – walk*" in order. Any change of order such as requiring the actor to now perform a "*walk – run – jump*" requires recreating the animation. This change can be triggered for many reasons. For instance the director may decide to change the script of a movie scene. The actor may be a game character responding to user control and environmental constraints. Given the cost and time associated with creating anew, it is only natural to search for mechanisms that allow reuse of existing animation in new contexts.

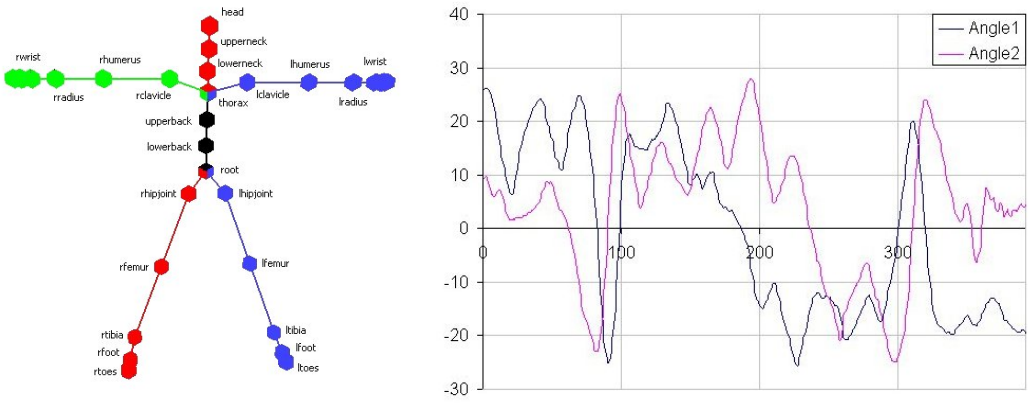
Motion reuse necessitates editing. However, motion data is iconic in nature and difficult to edit. The difficulty is akin to editing an old news video recording to synthesize live broadcast. Existing methods entail manually specifying key-frame like constraints and generating smooth displacement maps. Such editing engages the animator at a very low level and tends to be time consuming. A related problem is to select appropriate candidate clips for editing. Motion data being voluminous in nature, manual search is laborious. Effective reuse, therefore, calls for automation and high level control.

## 1.1 Mocap basics

Mocap is the fastest way of generating animation data today. Animation data is acquired from live performers. This has the advantage of capturing even individualistic nuances of a performer. Libraries of mocap clips are commercially available. Mocap is the most popular method of creating realistic humanoid animations.

A variety of technologies (mechanical, optical, electro magnetic, acoustic) are available for mocap. A sensor or an optical marker is attached to each joint of the performer, as in Figure 1.1. The sensor measures the joints degree of freedom variables (DOF's). The DOF's are, typically, the position and rotation values of the joint about a coordinate frame of reference. A data acquisition system uniformly samples each of the sensors. The recorded values form the animation data. The corresponding skeletal hierarchy is depicted by Figure 1.2(a). Mocap animation data consists of a bundle of motion signals as in Figure 1.2(b). Each signal represents a sequence of sampled values for each DOF. The sample values of the different joint DOFs at each frame determine the configuration of the articulated figure. The root of the skeletal hierarchy typically contains six DOFs—corresponding to translation and rotation about the x, y

Figure 1.1: An optical marker motion capture setup from [61].




(a) The hierarchical skeletal representation of the subject.

(b) Motion captured signals corresponding to a DOF.

Figure 1.2: Skeletal representation of humanoid and associated DOF signals.

and z axis. The rest of the nodes contain only three rotational DOFs.

A *motion* $M(t)$ is defined as a vector function that specifies the configuration of each of the skeletons $n$ joints (see figure 1.2(a)) at each point in time:

$$M(t) = (p(t), o_1(t), \ldots, o_n(t)), \tag{1.1}$$

where p is the position of the root in the global coordinate system and $o_i$ is the orientation of the $i^{th}$ joint relative to its parent's coordinate system. We use Euler angles to represent orientations. Orientations may also be represented by unit quaternions or orthonormal matrices. Motion data provides a discrete set of skeletal poses, or *frames*, $M(t_1), \ldots, M(t_k)$ that correspond to a regular sampling of the underlying motion $M(t)$. We generate root positions in between samples through linear interpolation, and intermediate joint orientations are computed through spherical

linear interpolation.

## 1.2 Goals

**Our goal in this thesis is to provide automatic, efficient, high level, reuse and synthesis methods for humanoid character motion.** Our techniques enable the following motion reuse scenarios.

1. **Motion re-sequencing:** Motion re-sequencing is a common reuse requirement. Our earlier example requiring reordering a "*run – jump – walk*" animation to a "*walk – run – jump* animation is an example of re-sequencing. Re-sequencing involves cutting and pasting motion from different clips. Visualize the animation of an actor running to kick a football and stopping a few paces later. In order to modify this clip such that the actor carries on running after the football kick, an animator may decide to cut-paste an existing *run* sequence at the end.

   Appending motions together creates a noticeable jerk at the join. In order to create a believable result, an animator needs to manually perform two tasks—(i) find the best point at which to splice motion, the kick and the run in our example, and (ii) hand edit the motion signals of both clips to smoothen out the transitions at the join. Since the animated character typically has 30 or more degrees of freedom, this amounts to a lot of manual work. What is desired instead is an automatic smooth stitch operation. *We synthesise such smooth transitions.*

2. **Cyclic motion synthesis:** Locomotive motion such as walks, runs, jumps, etc. are repetitive in nature. They character cycles through the same motion over and over again. The number of cycles of such motion required for a specific scene depends on factors like the actors travel distance. For example a character in a game may be required to walk ten steps to reach point A stop and then a further 15 steps to reach point B. Every cycle of such motion is similar. Rather than capture new motion for each scene, it is useful to synthesize such motion from existing data. The primary task in such adaptation is detecting cycle boundaries. A variable number of cycles can then be synthesized by looping through existing motion. Non-locomotive motion also contain cycles for example wav-

ing, dance steps etc. Identifying cycles and loop synthesis is useful here for extending the playback time. Today, such motions are created manually. *We identify such cyclic motion automatically and assist the animator in synthesizing the desired number of cycles.*

3. **Limb motion transplants:** Humanoid characters possess a huge action repertoire. A character can, for example, wave, punch, grasp and reach. He can perform these actions while standing, walking, running or jumping. The resulting actions for example, could be a reach while standing , "*wave while walking*," "*punch while running*" , or "*grasp while jumping*." The number of such combinations is potentially so huge that it is impractical to capture all the various combinations a priori. A useful reuse technique to synthesize new combinations is to transplant limb motions between clips. Given animations of "*wave while walking*" and "*a punch while running*," such adaptation can be used to synthesize animations of "*punch while walking*" and "*wave while running*." Simple minded grafts produce motions that look unrealistic. This is because every limb motion is inherently correlated to motions of other limbs. Another issue is the different timing characteristics of the source and destination clips. The rate at which the motion is performed may be different in the two clips. Our goal is to enable believable action combination synthesis using limb motion transplant by accounting for correlation and differing motion timing for ambulatory humanoid motion.

4. **Walk Parameterization:** Walking – preferred locomotive motion of humanoid characters – is used frequently in animation. As discussed earlier, one common adaptation of walk is to synthesize varying number of walk cycles. This type of re-synthesis allows a character to grossly attain its final goal. Further refinement is required to position the actor precisely at a desired location. This refinement occurs in natural human motion as adaptation of the stride. Sometimes a different type of refinement, viz. a change in foot lift is required. For example, a character may increase the foot lift in order to clear a small obstacle in its path without resorting to other form of locomotion such as a jump. Another example is of an actor increasing its foot lift in order to wade through ankle deep water. Our goal is to enable synthesis of such varying stride and foot lift walk from a given mocap walk sequence.

## 1.3 Contributions

This dissertation presents our motion reuse, synthesis and scripting frame work for mocap humanoid character animation. The following are **key contributions** of our work.

1. **Automatic foot-plant constraint identification (Chapter 3).** Our *foot plant identification* technique automatically identifies foot plants for motion on a horizontal plane. A majority of the mocap clips fall in this category. The foot plant detection is based on proximity of the foot to the ground plane and the relative displacement of the feet. A foot plant is accepted if the heel or the ball is within specified tolerances for both the above parameters.

2. **Automatic clip classification (Chapter 3).** Our *clip classification* is based on the sequence of foot plants detected. We observe that stand, walk, jump and run motions exhibit unique foot plant patterns. Our foot plant classifier is implemented as a finite state machine which recognizes these patterns and classifies the clips accordingly.

3. **Novel inverse kinematics solver (Chapter 6).** We present a novel adaptation of closed form two link inverse kinematics solver. We use this solver in our online synthesis of walk parameterization.

4. **Smooth synthesis of transitions and cyclic motion using cluster graphs (Chapter 4).** Our *cluster graphs* cluster frames from the input clips, based on similarity, into a cluster graph node. Each node contains frames from one or more input clips. The clustered frames are potential transition points from or to their respective clips from other clips whose frames also lie in the node.

   We generate transitions by splicing together clips at transition points defined by the cluster graph. Our synthesis blends DOF values about the transition point. The system can either use the best match or prompt the animator to specify one of top 'n' transition points. When the system is not able to find a direct transition between two clips, the animator is allowed to select transitions passing through one or more intermediate clips.

   We synthesise cyclic motion by transitioning among different frames from the same clip, situated earlier in time. Our cluster graph node groups contiguous frames of a single clip together into clip frame sequences. The presence of multiple clip frame sequences

belonging to the same clip indicates presence of cycles. We allow the animator to specify the number of loops to be synthesized. The system blends motion at the transition point as explained above. For simple loops the transition occurs between different frames of the same clip. An animator may also synthesize loops by specifying mutually recursive transitions between a set of clips.

5. **Locomotive motion grafting (Chapter 5).** Our graft synthesis allows automatic transplantation of upper body motion on to different lower body base motions. We allow grafts to be synthesized only between ambulatory humanoid motion. We further restrict the types of source and destination clips heuristically to increase graft quality. The animator chooses the the source and the destination clips of the graft along with the limb(s) to be grafted. Our system then synthesises the longest possible graft clip using the foot plant constraints to correlate and synchronise motion timings. Our experiments determine that a majority of the resulting graft clips are believable.

6. **Walk parameterization (Chapter 6).** We use a new per frame inverse kinematics (PFIK) [36] based method that synthesizes variable stride and variable lift walk and climb limb motion from a single mocap walk sequence using a kinematic walk model. We use stride and lift as the control parameters. Our use of a single motion clip complements the large database approach. Unlike most mocap based schemes our synthesis can be controlled programmatically.

   Our walk parameterization scheme works on the fly. It can be used like a filter to modify the output of any motion generation scheme in an online manner. For example, this scheme may be used to modify the walk of an interactive player controlled in-game character. Similarly, a motion planner may be used to specify our input parameter values for the non-player characters.

7. **Scripting reuse synthesis (Chapter 7).** Our scripting environment allows an animator to use our mocap editing techniques to script a digital performance. Our script consists of two parts — actor definitions and story. An actor definition associates skeletal hierarchy with motion clips and assigns them representative names. The motion clips themselves can be one of captured clips, transition clips or graft clips.

   Our story part consists of actor instantiation and the sequence of actions for each actor

instance. We allow to types of flows — parallel and sequential. Actions referenced in a parallel flow are performed simultaneously by the corresponding actor instances. Actions referenced in the sequential flow are performed in the defined sequence. We allow nesting of parallel and sequential flows.

## 1.4 System Architecture

In this section we describe our reuse and synthesis framework. *Mocap workbench* is our graphical user interface front-end that provides direct access to our methods. This toll is described in Appendix A. The accompanying CD contains a video demonstrating its operation. Our system comprises of the following four components:

1. *Clip preprocessor.* The clip preprocessor, Figure 1.3, is responsible for annotating and classifying the input clips. It comprises of two functional blocks: the foot plant annotator and the classifier. The foot plant annotator annotates the clips foot plant frames. The classifier detects and annotates the clip type—walk, run, jump and stand.

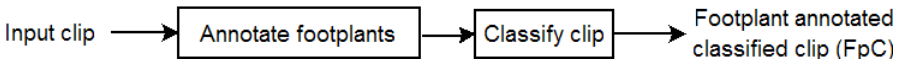



Figure 1.3: Pre processing of mocap clips.

2. *Cluster graph generator.* The cluster graph generator, Figure 1.4, builds a cluster graph for the input clip set. The *cluster graph* data structure clusters similar frames into nodes. Each node contains one or more contiguous frame sequences from the input clip set. The nodes are connected based on *natural transitions* observed in the input clips. For example if node A has frames 23–56 and node B has clips 57-89 from the same input clip, there exists a natural transition from node A to node B. This manifests itself in the cluster graph as an edge between the two nodes.

3. *Query and synthesis component.* The query and synthesis component, Figure 1.5, provides search and reuse primitives based on the cluster graph data structure. The query primitives allow an animator to search for candidate clips from the database. The synthesis primitives allow the animator to process the candidate clips for reuse. The synthesis primitives provide smooth splicing, motion grafts and walk parameterization.

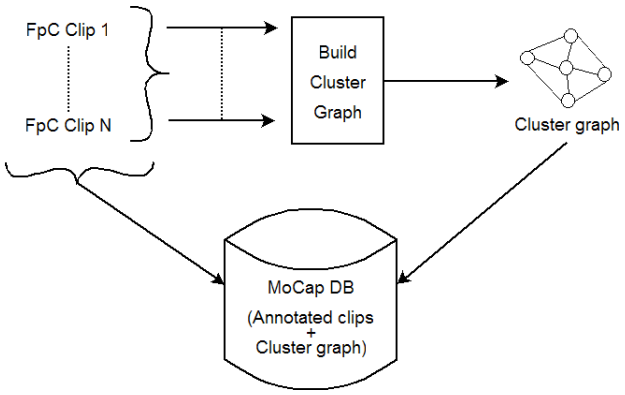



Figure 1.4: Our mocap database contains annotated clips and the cluster graph.

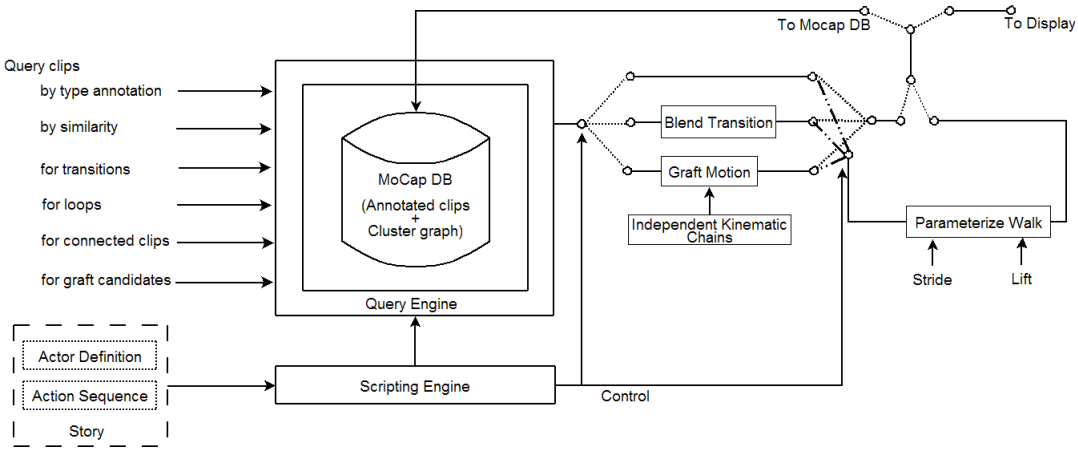



Figure 1.5: Query and synthesis component.

4. *Scripting system.* The scripting system allows an animator to use our mocap editing techniques in the context of story. The story definition allows for creation of actors and narratives. The actor definitions map actions to motion clips. The narrative allows actor instantiation and two types of narrative flow—parallel and sequential. The scripting system uses our synthesis techniques to generate the story animation.

## 1.5 Thesis Organization

The rest of the chapters of this dissertation are organized as described below. The logical relationship between chapters is depicted in 1.6.

In Chapter 1, we presents an introduction to the area of motion capture editing and define our goals.

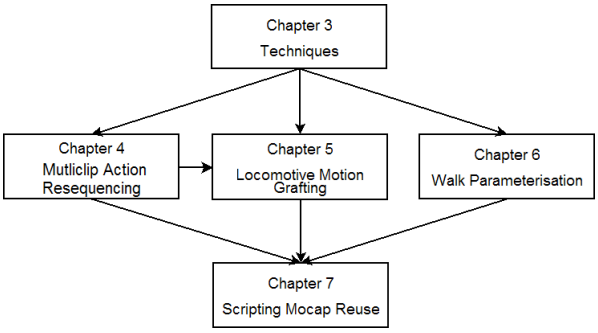



Figure 1.6: Logical relationship of chapters.

In Chapter 2, we present a review of literature in the area of motion capture editing and synthesis. We discuss the merits and disadvantages of techniques comparing them with our own technique.

In Chapter 3, we describe our basic techniques for preprocessing and output synthesis. The techniques discussed are our foot plant detection algorithm, our clip classification scheme and our interpolation synthesis method. These are fundamental to our reuse and synthesis architecture and are relied upon by different parts of our system.

In Chapter 4, we present our motion re-sequencing and motion cyclification method. We use the cluster graphs for generating inter motion transitions and cyclification. We present a number of example of motions synthesized using cluster graph. We compare cluster graphs with other graph based motion synthesis techniques.

In Chapter 5, we describe our locomotive motion grafting. We explain our notion of independent kinematic chains which forms the basis for limb transplants. We explain our use of foot plant constraints as the correlation signal. We describe our heuristic rules to restrict graft candidates. We explain our synthesis algorithm and provide examples of motion synthesised by our technique.

In Chapter 6, we describe our walk parameterization scheme. We use a simple geometric model to generate our variations. Using this model, we are able to parameterize walks using stride and foot lift as the parameters. We also describe our extension of this scheme to synthesize climbs.

In Chapter 7, we describes the scripting interface to our reuse and synthesis methods. We discuss language constructs and narrative structure supported by our scripting system.

In Chapter 8, we discuss the possible shortcoming's of our techniques. We discuss ways of

extending it and also point out challenging areas of future work in the area of motion editing and synthesis.

Appendix A, describes *"Mocap Workbench."*, a graphical front end to our reuse and synthesis techniques. We have used this application to synthesize all out output. The accompanying CD contains a video showcasing this application.

Appendix B lists the public application programming interface (API) of our reuse and synthesis library.

Appendix C describes our state machine implementation of the clip classification scheme described in Chapter 3.

Appendix D describes the method to compute homogeneous rotation matrix that rotates a point along x-axis to a specified position.

Appendix E describes the contents of the accompanying CD. The CD contains videos of our animation synthesis.

# Chapter 2

# Review of Literature

The history of research in humanoid character animation dates back more than twenty years. In this chapter we present a review of related work. We first describe general character animation techniques and then discuss work related to our techniques.

## 2.1 Character Animation

Creating an animated character involves modelling the characters geometry and creating motion specifications for the characters actions [9],[78],[114]. The geometry is constructed in a geometric modeller using built in primitives. A newer approach is to acquire the same by scanning real actors or miniature models [42]. The area of geometric modelling is mature. Animators have a wide range of tools and techniques at their disposal [105],[50], [20], [80], [102]. The advances in modelling and rendering has allowed creation of 3D animated characters. The content developer community has adopted these developments with such great success that animated content, earlier dominated by 2D hand drawn characters, today largely comprises of 3D computer generated characters.

Motion specifications breathe life into the characters geometric model. The traditional animation process employed in movies, called *cell animation*, involved hand drawing every single image frame. A large number of slightly varying images were drawn, photographed and then presented in a quick succession to create the illusion of motion [106]. Computer assisted animation has largely replaced this traditional manual process. Initial computer animation techniques

mimicked the traditional methodology. After the success of these, further research focused on creating higher level motion models, accentuating physical realism, defining motion parameterization and enabling motion editing.

*Creating animation is tedious, costly and highly skilled activity.* The challenge therefore is to develop techniques that reduce the tedium, time, cost and skill required. Reuse of existing animation plays an important part towards achieving this goal. Existing tools and techniques do not handle the reuse problem satisfactorily. Our work focuses on the area of high level reuse.

There are four main methods to create animation viz. *key-frame animation*, *procedural animation*, *behavioral animation* and *motion capture*. Each of these vary in the level at which motion is specified. They range from explicit to abstract, in that order (excluding motion capture). Explicit methods allow rich control at the expense of greater tedium and time. Abstract methods allow ease of specification whilst sacrificing control.

### 2.1.1 Key Frame Animation

Computer assisted key-frame animation mimics the manual process traditionally employed in movies. This process, called *"cell animation,"* involves hand drawing of the movies image frames, called *"cells."* As mentioned earlier, a large number of slightly varying images are drawn, photographed and then presented in a quick succession creating the illusion of motion. This process is tedious and cumbersome. The time and cost required to create animation is astonishingly large compared to that of live action. In addition, the time taken for a new animator to reach proficiency in his craft is large as well. To offset this learning cost, the *key frame* technique is employed. In key framing, senior animators draw key poses of animation spread out on the action time line. Junior animators then fill up the intermediate frames, called *in-betweens* using the key frames as guides. This process is called *in-betweening*. Computer animation techniques automate in-betweening of key-frames using spline interpolation [51], [100], [53], [1], [66].

The key-framing methods offer rich control to the animator. The animator can specify each and every nuance of a motion. The final result depends entirely on the animators skill. *Ironically this very freedom and degree of control, adversely effects the time required to generate the animation.* This is one of the most time consuming methods of generating motion data and

demands high levels of proficiency from the animator.

### 2.1.2 Procedural Animation

In procedural animation, motion is specified indirectly by either specifying velocities and accelerations or by assigning mass properties to the animated characters, and setting up appropriate forces or by specifying space and time constraints. Animation is produced by running a physically based simulation of the entire system. Here the animator has lesser degree of control then in key-framing. On the up side, animations can be produced much faster than in key-framing. A high degree of realism can be achieved by suitably configuring the physical properties of the character.

A number of motion models have been proposed to parameterize and generate locomotive motions. These can be classified into the following three categories:

1. Kinematic simulation: Kinematics is the study of motion of a body without considering its mass and the forces acting on it. For articulated human figures, this is the study of the positions, angles, speeds and accelerations of the human body joints and segments during motion. The first set of tools developed for motion specification were based on forward and inverse kinematics. Forward kinematics requires specifying the state vector of an articulated figure over time. The position of the various body joints are calculated from this specification. Inverse kinematics involves specifying joint positions and computing the joint angles.

   *Most of the kinematic approaches used for generating synthetic human locomotion rely on bio-mechanical knowledge and combine forward and inverse kinematics for computing motions.* The earliest methods devised used finite state machines inferred from bio-mechanical informations and controlled by high level parameters such as step length and step frequency [122]. The bio-mechanical knowledge was embedded as hierarchical concurrent state machines that generated the gait for the synthetic skeleton. A key posture was associated with each stage and in betweens are generated using linear interpolation. A variation of this approach is to use normalized velocities observed from experimental data [13]. *Motion produced by interpolation of forward kinematic poses can violate environmental constraints for example, floor penetration.* Collision detection coupled

with inverse kinematics techniques can be used to correct these artifacts by enforcing non-penetration constraints [14]. An excellent introduction to goal-directed motion of articulated structures is [52].

2. Dynamics simulation: Dynamics is the study of motion resulting from application of forces to bodies with mass. Like kinematics, dynamics has two main types: Forward (direct) dynamics and inverse dynamics. Forward dynamics requires the forces and torques to be specified. From these initial specifications and constraints, the system simulates the movement of masses. Inverse dynamics computes the forces and torques required to move a body along a known path. Earliest methods devised for dynamic simulation setup a system of Lagrangian motion equations [31], [49]. The equations are of the form: $F = Ma$, where $F$ represents a generalized force vector, $M$ represents a generalized mass vector and $a$ represents the acceleration. The matrix $M$ depends on the relative positions of solids and therefore changes over time, requires inverting at each time step.

   As in kinematics animation it is common to use results from bio-mechanics literature. Motion models for simulation are derived from bio-mechanical data. One such model for a walking human is that of a telescoping leg with two degrees of freedom for the stance phase and a compound pendulum model for the swing phase [17]. This model is used to compute the trajectory of the free nodes in the articulated figure. Generating motion for new behaviors and creatures is cast as trajectory optimisation problem [118].

   An alternative to specifying trajectories is to find control algorithms. Such controllers can be hand crafted or generated automatically. Motions including running, bicycling, vaulting and diving have been created using hand crafted controllers [43], [120]. While controllers exist for some types of motion, designing new controllers in general is a difficult task. Approaches to new controller design include frameworks for composing specialist controllers together to create a more general and capable control systems for dynamic characters [26]. Integrated controllers allow the character to perform motor tasks while being able to autonomously react to different situations within its environment.

   In other work, optimal control mechanisms, mixed with inverse kinematics are used to produce gaits [22]. On another trial, hybrid control techniques are used to bring interactivity and artistic techniques to physics based animation [59]. However, this system needs a lot of effort in designing and learning the appropriate interfaces for physical animation

which results in a steep learning curve.

Dynamics is also used to constrain kinematically computed motion. Here dynamics is used as a post process check to validate physical relevance of the generated motion. *Synthesizing motion kinematically is appealing because dynamics alone does not produce sufficiently realistic motion of active objects* [45].

3. Spacetime constraints: Spacetime constraint is a method to solve for the motion of a character over the entire animation time interval. It is a new formulation which permits the imposition of constraints throughout the motion interval. It freely propagates the effects of constraints backward and forward in time. *Constraints in initial, final or intermediate positions and velocities encode the goals of the motion, while constraints limiting muscle forces for preventing inter penetration define properties of the physical situation.* Newtonian physics provides a constraint relating the force and position functions that must hold at every instant in time. An objective function that determines how the motion should be performed is specified . This function is optimized using constrained optimization techniques to yield physically valid motion that achieves the goals specified the animator [118].

   An improved method for solving space-time constraints relies on "cspacetime windows." These are designed interactively, and enable the solution of a given part, in space and time of the animation [23]. This method improves upon the computational complexity of the original spacetime technique. Spacetime techniques have also been used for synthesizing biped walking animation and for motion editing [110], [34], [35].

*Animations generated using dynamics and spacetime constraints are subjectively more realistic than that generated kinematically. The primary drawbacks of this method are computational complexity and the difficulty in coming up with new motion models and combining existing ones together.* While models exist for separate actions such as walking, running, cycling etc, unifying these is still a challenge. Unifying here means composing together distinct models to create a composite motion. Another disadvantage of this scheme is the restriction imposed on artistic freedom by the physical process. While it is easy to create physically realistic animation, it is relatively difficult to create animations that follow toon laws of physics.

### 2.1.3 Behavioral Animation

The behavioral animation approach is the most abstract amongst all of the computer assisted animation methods. Here, an animation is generated by modelling and simulating the mental process of the character. The behavior is specified in terms of intents and goals. Animation results when the simulated character tries to satisfy its internal goals. The intents are in turn driven by these goals. In order to satisfy intents, the character needs locomotor skills. These skills are provided by embedding a motion model within the character. The character can then control its own locomotor actions. This motion model can either be based on procedural animation or be a specified as a simple action mapping table indexing pre-generated animation data. The key-framing or motion capture techniques may be used to generate this data. In addition, the character is augmented with a virtual sensor with which to sense the environment and the result of its own actions, thus setting up a feedback control loop.

The behaviorally animated character is best represented by a layered model such as in the figure 2.1. The top layer represents the high level goals and intents. The middle layer represents the motion model used by the character. The lower layer represents the geometric data used by the motion model. Control flows from the top layer to the bottom layer. *Behavioral animation is*

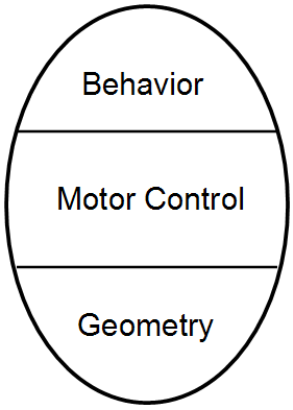



Figure 2.1: Three layered character model for behavioral animation.

*uniquely suited for animating groups such as flock of birds, schooling of fishes or crowds of humanoids* [90], [108]. It is used for situation wherein individually animation each member of the group is impractical. It is also used generating responsive characters that react to their virtual environment [84].

### 2.1.4 Motion Capture Animation

Motion capture is a technique for acquiring animation data directly from live performers. Though new, this technique is by far the fastest way of producing extremely realistic animation data. It is possible to acquire the finest nuances of the performers actions. This has helped motion capture become a very popular method. The generated motion stream is akin to a video recording. Further processing however is an area of research. The output of a motion capture session is identical to the output of key-framed skeletal animation.

### 2.1.5 Comparison of animation methods

*In a real world applications, it is common to see simultaneous application of one or more of the above animation techniques.* Each of the above serves a special requirement. Whenever the animator requires artistic control key-framing is the method of choice. For quickly setting up physically realistic base animation, the procedural methods are used. Behavioral animation is used for crowd simulation and elsewhere where a high degree of precision is not essential. For example - to generate animation for secondary characters in a scene. Motion capture is unique in that it generates physically realistic data and at incredible speeds. It is used when realistic character animation is desired and can be easily acted out by skilled performers.

## 2.2 Mocap Based Synthesis and Reuse Techniques

Motion captured data almost always needs processing before use. At the very least, motion capture systems contain online smoothing filters to alleviate the effects of sensor noise. Commonly needed processing operations include editing, re-targeting, blending, stitching, smoothing, up-sampling and down-sampling. Motion editing techniques attempt to address these requirements.

### 2.2.1 Mocap Editing

Editing implies changing some characteristic of the captured motion. *Motion may require editing for one of two main reasons - constraint adaptation and re-targeting.* Acquired motion very

often does not meet all the constraints that an animator wants. Examples include being at a particular position at a particular time accurately or synchronizing movement to another action that has been shot before. Such cases require editing to satisfy constraints. Re-targeting refers to the problem of adapting motion to different characters. These characters typically have different limb lengths. Sometimes the characters may be even less similar - for example adapting biped motion to a quadraped.

Intrinsically, motion capture yields an unstructured representation - a sequence of joint angles. Editing this kind of iconic description poses a problem analogous to that of editing a bitmapped image or a sampled hand-drawn curve. One approach to editing is to fit curves to the raw data, producing a key-frame like description that can be modified by editing the curves control points. This is not attractive as then the fit curve is likely to need at least as many control points as would have been needed to key-frame the motion manually. An alternative is to devise a scheme which allows specification of constraints in a key-frame like manner and alters the motion based on this specification. The difference here is that the original motion signal is left as is and only the desired changes are key-framed. This results in far fewer keys then would be required otherwise. Motion warping is such a scheme [119]. An animator specifies the desired configuration of the articulated figure at few key instants in time. The original motion captured signal is then either scaled or displaced to match the constraint signal at the specified key time. Correspondingly interpolating splines are created for either scaling or displacing the original signal. Scaling produces exaggerated motion while displacement better retains original motion characteristics. An analogous method to achieve the same is *motion displacement mapping* [18]. Displacement mapping method does not cater to the scaling part present in the motion warping formulation. Nonetheless displacement maps are useful and preferred over motion warps as they are easier to construct and because scaling distorts the original motion. Scaling is however, useful to exaggerate motions giving them a toony feel [113]. Convolving the the motion signals of specific joints with a signal resembling the inverse of Laplacian of Gaussian results in a *satisfactory* toony feel. This approach generates exaggeration, squash and stretch [58] with proper anticipation and follow-through motion as desired.

Satisfying every constraint curve individually can lead to motion artifacts. Spacetime formulations globally satisfy all constraints simultaneously whilst maintaining similarity to original motion is to use the [66], [34]. The spacetime techniques cast the problem as global non-linear

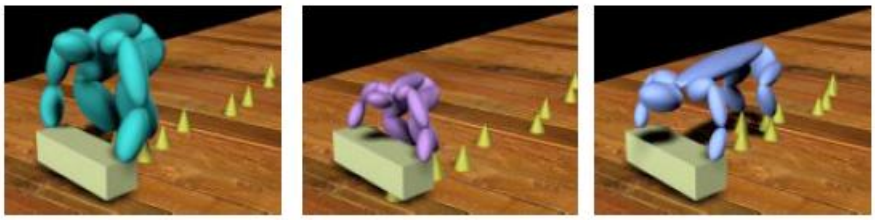


Figure 2.2: Re-targeting example. Different sized characters animated using the same source motion processed using re-targeting constraints from [34])

.

optimization problem with suitable objective functions. Non-linear optimization problems are computationally expensive. An alternative is to use a hierarchical B-spline fitting technique with inverse kinematics [63]. This technique breaks the large global optimization problem into many small local optimization problems.

Re-targeting motion involves editing motion to meet the re-targeting constraints. The methods described above can be used here. What differs is the way the constraints are arrived at. Figure 2.2 shows three characters with different proportions animated with the same motion and constrained to pick up the box at a the same time interval. Spacetime constraint and hierarchical B-spline techniques have been used for re-targeting [34], [63]. Both these techniques require constraints to be identified manually. These constraints are typically specified on joint angle values or their position values. Constraints are specified to enforce foot plants and in Figure 2.2 for positioning the hands in contact with the box. The systems then updates the unspecified joint values using inverse kinematics while simultaneously minimizing the deviation from the original motion (the objective function).

Global optimization techniques being computationally expensive are inaccessible for online use. Computer puppetry is an application wherein online motion re-targeting is required. A live actors performance is mapped on the fly to animated characters that are part of live broadcast. An algorithm incorporating fast inverse-kinematics solver with prioritized rules for constraint matching can be applied in this context [95]. The prioritization rules used here favour posture similarity to end-effector positions in the absence of scene constraints (such as external objects in contact) and favour position matching in the presence of such constraints.

Finally, motion may also be edited to correct for physical plausibility. Most motion editing methods do not validate this aspect. *While not a key requirement, as most animators flaunt laws of physics routinely, edited motion can be post processed to restore physical correctness* [94]

*when desired.* A physical model such as used in dynamic simulations is used to validate motion and derive corrective displacement map curves.

### 2.2.2 Mocap driven synthesis

Mocap driven synthesis techniques aim to synthesize new motion from existing motion captured animation. The simplest way to generate such animation is multi-target interpolation [18]. Multi-target interpolation refers to interpolation between more than one recorded values. For example, if there is a brisk walk and a tired walk recorded, one can interpolate between these two specimens to generate a whole spectrum of walks. The prerequisite to employing multi-target interpolation is synchronization of frames. Typically the constraint frames, such as frames where in the feet are planted on the ground, need to be synchronized. Without such synchronization the resulting animation will not represent plausible motion and thus find little use.

Using a variety of action clips, multi-target interpolation can be used to synthesize a large gamut of human actions, as demonstrated by the Verbs and Adverbs system [91]. In this system, Verbs, represent a collection of clips depicting one type of motion. An example of verb motion is bipedal locomotion. The variation within the verb space is parameterized by adverbs for example slow walk, fast walk, run, strut etc. The system requires each clip in a verb to start with similar poses, contain equal content[1] The system also requires the constraint key-frames to be specified as input. This allows trivial setting up of inter clip frame correspondences mentioned above. Radial basis functions are used to interpolate in the adverb parameter space . This allows generating continuous interpolated variations of motion. Transitions between verbs is also supported. This is accomplished using linear blending.

Variations of a motion can be generated using a single clip as well. Multi-resolution sampling has been used to separate frequency bands of a captured motion. Adjusting the gains of different bands upon recomposition synthesizes variations. Increasing the gain of the high frequency band, for example, produces jittery motion and that of low frequency band creates exaggerated motion. The band gains do not have to be positive. The multi-resolution method can also be employed along with multi-target interpolation. The decompositions of motion signal can either

[1] Here equal content means same number of cycles and no spurious motion (such as a head-scratch in a walk cycle).

be accomplished using Laplacian pyramid technique [18] or by using wavelet decomposition [87]. The re-synthesis process can be tweaked to generate variations such as those observed in repetitive natural motion. For example each cycle of a walk, while similar exhibits variations. Replacing high frequency components with random noise [81], [82] and selectively superimposing noise on some of the intermediate levels allows such synthesis.

Wavelet decomposition technique has also been used to drive motion from a partial specification or to improve a specified motion by substituting signals from an existing capture motion [88]. The core of this system deals with matching motion signal fragments of the driving signal. However such a search is difficult to perform with the input motion signal. The given signal is therefore decomposed into various resolution bands and bands with relatively lower frequency are used to perform signal matching. Fragments of matched signals are stitched together using linear blending.

Motion captured in one or more motion clips can be parameterized in space [116]. A regular rectangular or cylindrical grid pattern can be used for sampling the space and motion frames corresponding to each of the grid points identified. New motion can then be synthesized by traversing the sampled grid in the desired order and interpolating between motion samples corresponding to these grid points.

Motion clips can be treated as periodic signals and analysed for frequency content using Fourier analysis [109]. Adjusting the frequency components on re-synthesis generates motion variations. This method intrinsically captures inter clip frame correlation. The signals, which are treated as periodic, are assumed to lie between $-T/2$ to $T/2$ where $T$ is the period and then normalised. Once normalised Fourier functional models are generated for all clips. These can be mixed freely using interpolation or extrapolation. The synthesized output remains perfectly synchronized. Another characteristic of this method is that it allows interpolation both in the frequency and in the time domain. Output generated in either case is similar.

The methods described above directly use input motion data to synthesize the output. A different approach to mocap driven synthesis is to first learn and construct generative motion models and use these models to synthesize motion. Standard machine learning schemes such as HMM's can be applied. An HMM derived scheme called style HMM (SHMM) has been demonstrated to learn and generate style variations in motions [15]. The idea explored here is – entropy minimization to learn the input motion model, and cross entropy minimization to correlate

the different model thus learnt. Using a parameterization defined on cross entropy minimized model any of the individual motion models or interpolants can be produced. The final motion is synthesized using the generated SHMM.

A hierarchy of PCA spaces has been employed to represent a set of input walking and running motions captured at different speeds from different subjects. this is then used to synthesize walks and runs with different styles, locomotion type and personality [32]. Walk and run motion samples are captured from five subjects for various speeds. The system reduces input dimensionality using PCA. The coefficients of this first level PCA space cluster motion frames on the subject type. PCA is performed again on these first level PCA coefficients to obtain a second level PCA subspace. In this new space the coefficients cluster based on motion type. A third PCA subspace is created using the coefficients of the second level. It is observed that coefficients of the third level encode speed. A linear least square fit is constructed at this level and used for extrapolating speed. Interpolated in the second level varies locomotion type between walk and run. Interpolation in first level controls personification.

A novel way of synthesizing new motion is by cutting and pasting parts from various clips together. The cut paste can either be across the whole articulated figure or may only involve selective limbs and joints. Graph based methods are best suited for cut paste of whole figure motions. In these methods nodes represent motion clips and edges represent transitions between motions. These methods differ in the construction of the graphs, the granularity of the motion segments with each graph node and the re-synthesis technique used. Methods [56], [61], [5] treat each motion clip as a graph node. Method [44] fragments each motion clips based on motion beat analysis. Each of these methods then finds a transition from each node of the graph to every other node based on a fitness criteria. The fitness criteria measures the distance of the clips in the two nodes. A transition is recorded if for any frame pairs in the two nodes, the distance is less than a given threshold. The fitness criteria takes into consideration position, velocity and acceleration of motion at the joints while measuring the distance. The re-synthesis process comprises of a graph walk. For some applications a random walk suffices, such as for rhythmic motion synthesis [44]. The walk and therefore the synthesized animation can be controlled by specifying additional constraints. One way to specify the constraint is to sketch a path and label parts on the path where a specific motion clip is required. A search for suitable paths matching the constraint is then used to determine the nodes visited [56]. Alternatively

constraints can be specified by identifying individual frames, spatial locations that must be attained at specific times. This requires an exhaustive search for the best possible solution to be performed. However such a search using conventional methods like dynamic programming, can be impractical. A more feasible approach is to perform a randomised search on a hierarchy of graphs, [5]. Alternatively, the graph walk can be controlled interactively by presenting the animator a set of choices or by acting out the desired motion [61].

It is possible to cut paste sub-hierarchy motion. Such synthesis need to account for motion correlation across various limbs of the articulated figure. These methods are new. Few authors other than ourselves handle this case at present. One approach is to combinatorially cut and paste motion of various limbs. SVM's can be constructed and trained using valid motions, and later used to discern acceptable combinations from invalid one's [47]. Another approach is to establish correlation between two motions using dynamic time warping and transplant part upper body motion from a donor clip to the source clip [40]. This results in expanding the number of motion combinations available for reuse.

## 2.3 Transitions, Correlation and Blending

In motion captured animation, transition refers to the smooth change in playback clip. Almost every instance of transition is accomplished by blending frames from two or more different clips. The most commonly used blend function is linear interpolation. Radial basis functions, coefficients of Fourier transform and even bands of Laplacian pyramids can be linearly interpolated to construct the transition [91], [109], [18]. A blend can also be constructed by displacing the signals at the joint such that they joint smoothly. A displacement map can be used for such alteration [5].

A prerequisite for synthesizing transitions using blend is establishing inter clip frame correlation. This essentially requires identical skeletal structure and correspondence of frames with respect to time. For example, when blending walk cycles, the steps must coincide so that the feet strike the ground at the same time for corresponding parameter values. If a sad walk is at a slower pace that a happy walk and if these are blended together without first establishing a correspondence between steps, the feet will no longer stride the ground at regular intervals. In fact they are not even guaranteed to strike the ground at all. Hence it is essential to establish a

mapping between frames of the two clips. Establishing such a correlating can require warping time dimension of one or both the clips.

Dynamic time warping is a standard technique for nonlinear signal matching. The time warp procedure identifies a combination of expansions and compressions which can best "warp" the two signals together. For mocap animation, time warping is applied in the discrete time domain to register the corresponding motion signal parameter signals such as joint angles [18]. The basic dynamic time warping technique is hard to extend to multiple motion clips because of exponential computational cost. An alternative to achieve the same is to find pairwise correlation and establish a normalised mapping between all the clips. For example if $M_1$, $M_2$ and $M_3$ are clips and $S_{1->2}$, $S_{1->3}$ and $S_{2->3}$ are pairwise correlation functions. Then if $S_{1->2}(f_1) = f_2$ and $S_{1->3}(f_1) = f_3$, it is likely that $S_{2->3}(f_2)$ is also approximately $f_3$. With this assumption it is possible to remove $S_{2->3}$ and combine $S_{1->2}$ and $S_{1->3}$ into a single time warp curve. To ensure good registration a few additional constraints - continuity, causality and slope limit are required [54].

Blending of clips can generate artifacts even when motion frames are time warped. For instance consider the interpolating between motion of two walk clips - one turning left and the other turning right. It is reasonable to assume that the left and right turns cancel each other when equally blended and a straight walk is produced. This however is not the case. The resulting walk is shorter than expected as the position vectors that are blended cancel each other and produce stationary motion. In order to overcome these artifacts blending needs to be performed in aligned local coordinate frame [54].

## 2.4 Motion Annotation

Motion annotation is an integral part of mocap data processing. Annotations are required to declare motion semantics, specify constraints and to correlate with other motion clips. Semantic annotation is almost always carried out manually[91], [6]. Such annotations describe the type of motion for example a walk with a wave [6]. These annotations can be used to synthesize animations with specific motions.

Foot plant annotations are important for motion synthesis. These have traditionally been spec-

ified out semi-automatically [57]. Approaches to identify constraints such as proximity have been developed. For example the sign change of second derivative of the motion curve is a good indicator of important events. This in combination with a local neighbourhood search for geometric objects can be effectively used to determine proximity constraints [10]. Similar methods may also be adapted for detecting foot plants [33]. Sensors may sometimes be used to simultaneously capture foot plant motion from performers [30]. Software techniques for automatic foot plant are not robust enough for general use and can only be applied in limited contexts. Foot skate is a related problem and manifests itself as sliding feet when in fact the feet should be firmly planted on the ground. Once foot plants are estimated or annotated, foots Kate can be corrected by enforcing a positional constraint and using inverse kinematics to update the other joint angles [57], [46]. A recent approaches learns foot plant constrained poses from posture information in input data and classifies unknown frames based on a 21 frame input vector specifying positions of knee, ankle and toes, [33].

## 2.5 Motion Cyclification

Motion cyclification refers to one of two topics - generating cyclic motion such as walk cycles or hand wave and motion playback in a loop i.e seamless repetition of one or more connected motion fragments. Cyclic motion synthesis is inherently addressed by motion synthesis using Fourier model [109] and techniques specialised to synthesize locomotive motions [32]. Enabling playback in a loop involves determining mutually recursive transitions. A graph based motion synthesis method can be adapted to search for motions that can be looped. An alternative approach is to find a common poses shared by clips and model these as nodes of the graph. The motions that pass through these poses are then form arcs of the graph. If the poses are shared by start and end segments a motion clip, then the arc loops back to the same node. If not motion can be looped by traversing multiple nodes with common posture finally returning back to the start node [38].

## 2.6 Motion Parameterization

Parameterizing motion is a key goal of data driven motion synthesis techniques. This allows synthesis of variations in the motion. Parameterizations can modelled, learned or observed. Variations are essentially created by interpolation or extrapolation in the feature space controlled by the parameter. Parameters to be learned can exist in Euclidean space [116] [55] , frequency domain [109] or in some abstract space such as the space of HMM models [15]. Parameterization can also be defined as a search for similarity. For example searching for similar postures in a motion, while allowing more flexibility to a particular limb. This is likely to uncover all possible postures that the limb takes from a the given base posture. The different positions of the free limb can then be used as in multi-target interpolation to drive the limb animation towards one these goals. Affecting the DOF's of only this limb, leaves other parts of the animation unaffected. Such techniques can be used to parameterize motions like reaching, or kicking [55]. Parameterized action representation (PAR) [10] is a scheme to build parameterized action description. Here the parameterization includes the objects that the object interacts with. For example reach out and touch and object. The object positions are taken as the parameters. The input motion is recorded and suitably modified using inverse kinematics so as to reach a different parameter positions. The scheme builds in safeguards in the form of preconditions and postconditions to be able to select appropriate motions and parameterization.

## 2.7 Scripting

Mocap editing for reuse tends to be an interactive activity. Hence not much research exists in scripting for mocap reuse. However, the spacetime constraint based motion transition system describe in [92] contains a motion expression interpreter. In this system motions are represented as a hierarchy of expressions. Motion expressions can be of one of three types of objects: intervals, degree of freedom and motion unit. An interval is a list of scalar functions of time plus a start time and an end time. A DOF is a list of intervals that defines the value of a single joint DOF over time. Motion unit represents a collection of some but not necessarily all of the articulated bodies joint DOF's. Three kids of operations are defined on these primitives - set operations, function operations and insert/delete operations. Set operations allow time interval arithmetic and addition and deletion of DOF's. Function operations define transformations,

clipping, concatenation. Additionally functional operations can be composed with each other. Insert and delete operations allow insertion and deletion of objects. Using this language, motion operations can be specified easily. However this language is not animator friendly.

Another example of a scripted animation environment is the Improv system [82]. This system allows the animator to compose their own motion primitives and define actions based on these primitives. Further more animators are allowed to group actions in layers. This layering is used to determine mutual exclusivity of actions and parallelism. In order to create a story, the authors simple call their action definitions. This system is animator friendly. However it allows only sequential flows and does not allow for functional composition which is a powerful reuse feature.

## 2.8 Discussion

Our *cluster graph* method is a graph based data driven motion synthesis method. We differ from existing methods in two ways: (i) Our graph nodes contains motions frames at much higher granularity - typically from 1-60 frames and (ii) We allow nodes to contain motion frames from more than one clips. These two characteristics give our data structure additional properties beyond just finding transitions viz. the ability to detect cycles and the ability to correlate motion across clips. Chapter 4 explains the construction and properties of cluster graphs in detail.

Our *locomotive motion grafting* method uses a technique similar to one defined in [40]. [47] is work published prior to ours though and unpublished version of our work predates this. We differ from [47] in being able to generate grafts deterministically. The work reported by [40] is newer than ours and published independently. The key differences between this work and ours lie in the technique used to establish correlation between graft source and donor clips. While dynamic registration curves are used by the later, we use foot plant based correlation obtained using our cluster graph for establishing motion correlation. We allow selective limb transplants. Our synthesis technique synthesises all possible combinations following strict rules based on heuristics.

Our walk parameterization scheme uses a kinematic model to synthesize variations in the recorded walk. We infer parameters such as leg length, step size and stride from the speci-

fied input clip. This allows our method to work for differently proportioned characters with no change. We allow online up-scaling and down-scaling of these parameters and generate on the fly variations.

Our scripting language is designed to allow reuse while allowing animators to define actors and actions in a natural way. Additionally we allow multiple instantiation of actors. We support nested parallel and sequential flows in our story script. The primitives used for defining actions allow direct access to our reuse algorithms in an accessible way and are functionally composable.

# Chapter 3

# Basic Techniques

In this chapter we discuss our input clip preprocessing and interpolation synthesis technique. Our preprocessing annotates individual frames with foot plant information and classifies the clips based on their type. We distinguish amongst *stand*, *walk*, *run*, and *jump* motion. We compute frame attributes such as world positions, per frame relative displacement, velocity and acceleration of selected joints. We use this information in our reuse and synthesis methods described in later chapters.

## 3.1 Foot plant identification

A *foot plant* occurs when the foot of an animated character strikes the ground. The foot continues to be in contact and stationary for the duration of the foot plant. Foot plant identification is very important for motion editing operations and interpolation synthesis such as ours. Ignoring foot plants during interpolation produce visible artifacts ranging from foot skates to unnatural motion where the feet may never touch the ground. Foot skate refers to a phenomenon where the foot of a character slides on the ground, when in fact it should be firmly planted. In a motion captured sequence, the foot plant position in adjacent frames may not coincide due to noise resulting in foot skate. A technique to identify and correct foot skate is described in [57].

Foot plants are identified and annotated manually. Automatic identification of foot plants for the general case is difficult. Most simple minded algorithms fail, as humanoid characters produce infinite variations of motion that are difficult to quantify. We present an algorithm designed

to work for a limited class of animations – the class of *ambulatory humanoid motion on a horizontal plane*. Even for such restricted subset, identification is complicated by noisy input data. Our algorithm reliably detects foot plants for motions in this class, even in the presence of noise. The foot is planted when either the heel or the ball are in contact with the ground (see

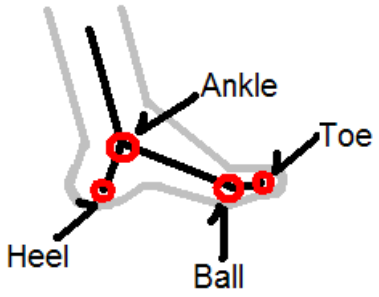
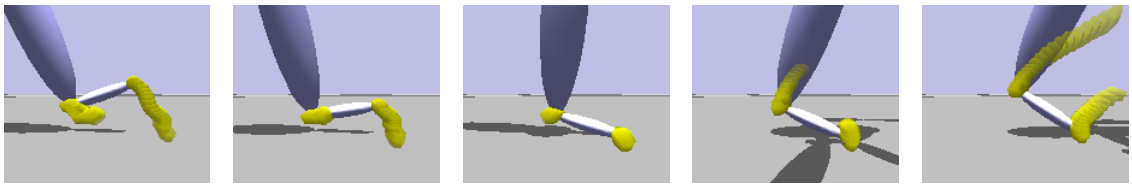



Figure 3.1: The human foot.

Figure 3.1). Figure 3.2 shows the different stages of foot plant in walks—the foot approaches for a plant, the heel strikes, the ball strikes, the heel lifts off the ground and finally the ball lifts off the ground. The frames between the heel strike and ball lift off are *foot plant frames* [28]. In practice, mocap recordings track the motion of the ankle and the toe joints. As seen in Figure 3.1, the heel is connected to the ankle by a rigid link with zero degrees of freedom. The ball and toe are sufficiently close to each other, more over, for most animations, the ball-toe link is not modelled separately. Therefore we use ankle and toe joint signals to infer foot plants.



(a) Approach (b) Heel strike (c) Ball strike (d) Heel lift off (e) Ball lift off

Figure 3.2: Foot plant stages for walks

### 3.1.1 Foot plant detection algorithm

In this section, we present our foot plant detection algorithm. Our algorithm detects foot plants on a level horizontal plane in the presence of sensor noise and foot bounce. Note that our

algorithm is designed to handle commonly occurring motions such as walk, run, jump, and stand performed on level floor. We do not address other types of humanoid motion in this work.

We use a two stage process. In the first stage our algorithm infers heel and ball strikes by observing the recorded mocap data for the ankle and toe joints respectively. Each frame is annotated with this information. In the second stage, we mark foot plants frames using these annotations.

**Inferring Heel and Ball Strikes**

Ideally, when planted, the feet are stationary and in contact with the floor. In practice because of sensor noise and calibration errors the feet may not be in line with the floor and may exhibit small movements. Our algorithm marks the ankle and toe as planted if their vertical displacements from plant positions and their linear velocities are within specified thresholds. The algorithm is as follows:

**Inputs**: $\varepsilon$: Vertical displacement threshold.

$\eta$: Velocity threshold.

$\delta t$: Sampling interval between two frames

$w$: search window.

$y_a$: Ankle Y position (vertical) when the feet is rested on the ground.

$y_t$ : Toe Y position (vertical) when the feet is rested on the ground.

For both ankle and toe joints, at each frame:

**Step 1**: Compute $p_i$ – the position of the joint at frame $i$.

**Step 2**: Compute relative displacement $\delta s_i$, instantaneous velocity $v_i$ and height of the joint relative to foot plant position, , $\delta y_i$.

$$\delta s_i = p_i - p_{i-1}$$
$$v_i = \frac{\delta s_i}{\delta t}$$

$$\delta y_i = \begin{cases} \|p_{y_i} - y_a\| & \text{for the ankle joint} \\ \|p_{y_i} - y_t\| & \text{for the toe joint} \end{cases}$$

**Step 3**: Mark joint as *planted* for this frame if

$$(\delta y_i < \varepsilon) \;\&\&\; (\delta v_i < \eta)$$

**De-bouncing joint strikes** The joint annotation obtained from step 3 is noisy, see Figure 3.3(a) and Figure 3.5. After the entire clip is processed once for joint plants, we cleanup the noise by detecting and correcting drop outs as follows:

For each frame , $i$, of the clip: if the joint is not marked planted, we construct a search window $[w_{min}, w_{max}]$ centered about $i$.

$$w_{min} = \begin{cases} 0 & \text{if (i-w)} < 0 \\ i - s & \text{(i-w)} \geq 0 \end{cases}$$

$$w_{max} = \begin{cases} i + w & \text{if (i+w)} < maxFrame \\ maxFrame & \text{if (i+s)} \geq maxFrame \end{cases}$$

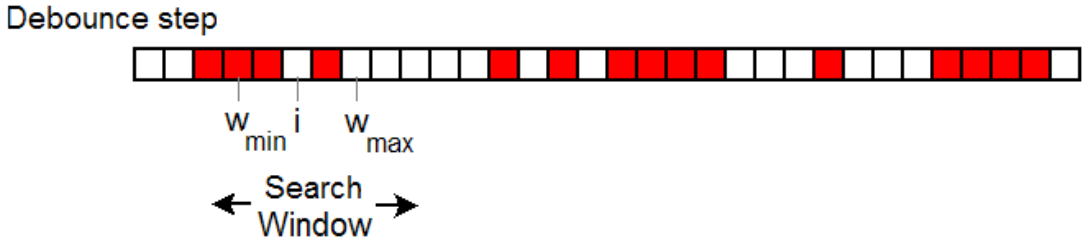
where $maxFrame$ is the index of the last frame in the clip.

$$w_{min} \leq i \leq w_{max}$$

We search within this window for existence of other plant frames. We annotate the current frame as planted if we find such frames on both sides of $i$. This process is depicted in Figure 3.3(b). The output of this algorithm for clean input is shown in Figure 3.4 and for noisy input in Figure 3.5.

Note that we do not check for the presence of spurious foot plants. This is driven by our observations. We see a lot of noise around foot strikes. However, since the foot is lifted well above the ground threshold during the swing phase, spurious foot plants are not common. In fact we have not encountered such a case in our experimental data.


(a) Noisy foot plant annotation




(b) Searching for dropouts


(c) The de-bounced signal

Figure 3.3: Figure shows the de-bouncing of joint plant annotation obtained at the end of stage 1. Red cells denote frames for which the joint is planted.

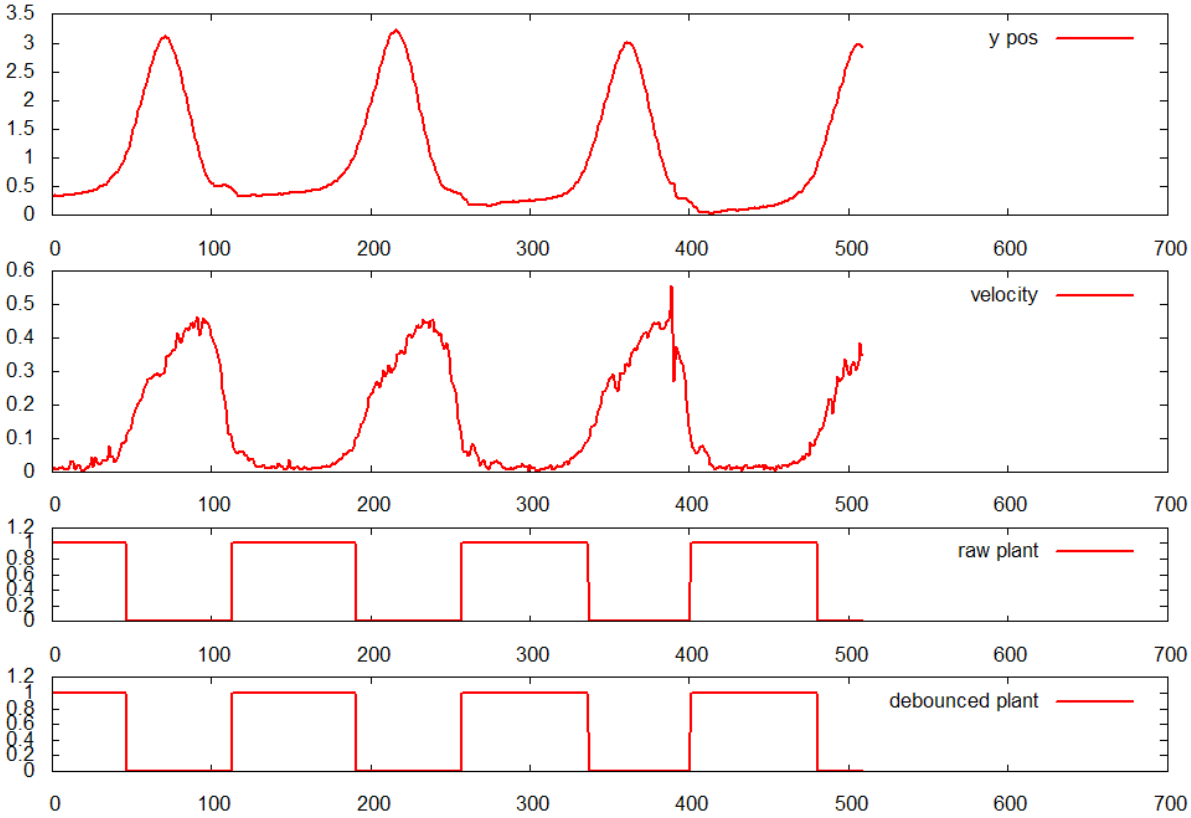



Figure 3.4: Output of algorithm for "noise free" left ankle joint signal. From top to bottom, the first graph shows the y position of the ankle. The second graph shows the velocity of the joint. The third graph shows output of stage 1. The fourth graph shows output of stage2.

**Annotating Foot plants**

The next step is to annotate foot plants. The foot is planted as long as the heel or ball is in contact with the ground. We therefore mark frames with either ankle or toe plant annotation as planted.

### 3.1.2 Experimental Results

We used motion clips from the CM mocap library as input to our system. We have tested our algorithm with several clips, sampled randomly. Table 3.2 is a representative listing of clips used for foot plant detection test. The listed sample clips contain 20,192 frames sampled at 30/60 fps for a total of about 5-7 minutes of animation. Our algorithm successfully detected foot plants for this varied set at online rates. Figure 3.7 and the accompanying video shows results of our foot plant detection scheme.

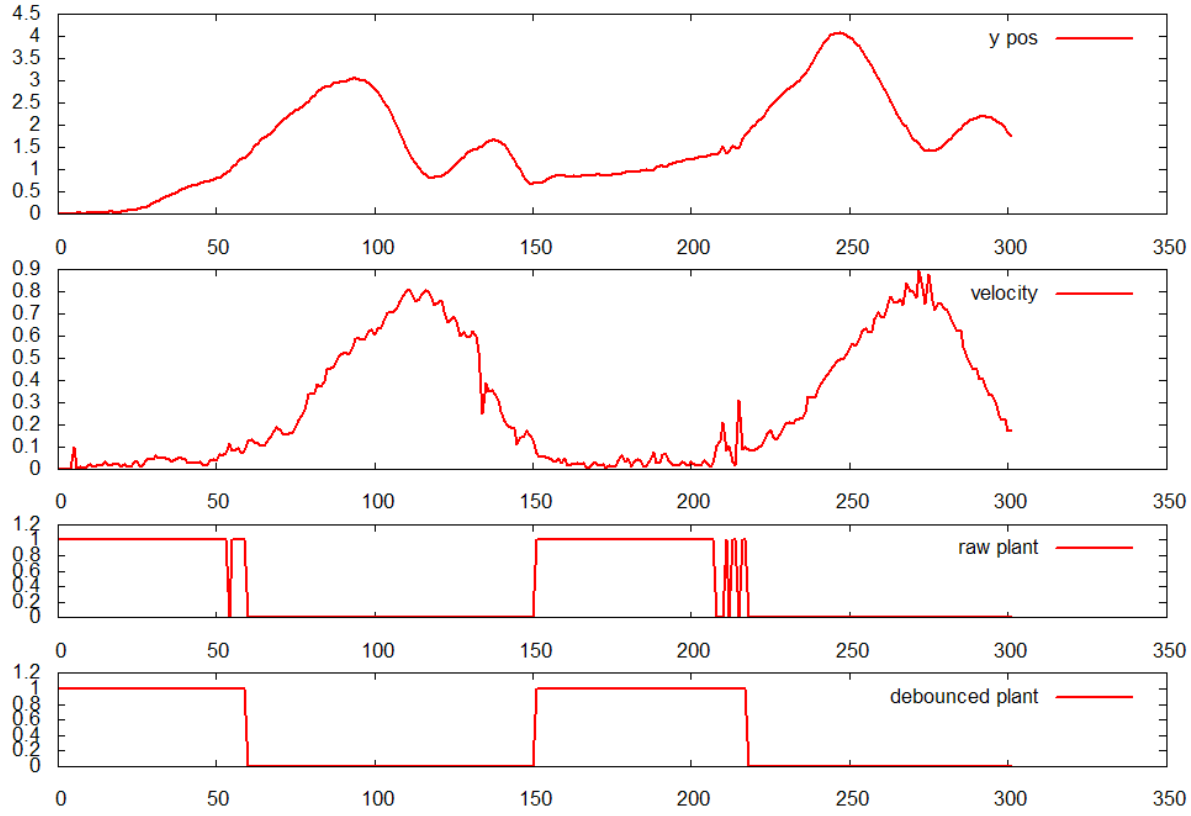



Figure 3.5: Output of algorithm for "noisy" left ankle joint signal. From top to bottom, the first graph shows the y position of the ankle. The second graph shows the velocity of the joint. The third graph shows output of stage 1. The fourth graph shows output of stage2. Notice that the bouncy signal in the third graph is corrected by the de-bounce step, as can be observed from the fourth graph.

We compared our frame level results with manually annotated foot plants. Four human observers manually annotated nine clips. Each clip was annotated by at least two observers. The clips contained 3820 frames for a total of 7640 data points (one observation for each leg plant). Before running our algorithm we adjusted the relative displacement and lift tolerance parameters by trial and error such that visually good looking results were obtained for all clips. We obtained the following results:

Table 3.1: Confusion matrix for our foot plant detection algorithm

| | Computed ($\eta = 6.8$ & $\varepsilon = 0.1$) | | |
|---|---|---|---|
| Observed | | Foot plant | Not foot plant |
| | Foot plant | 5422 | 152 |
| | Not foot plant | 167 | 2279 |

The corresponding precision, recall, specificity and accuracy values are:

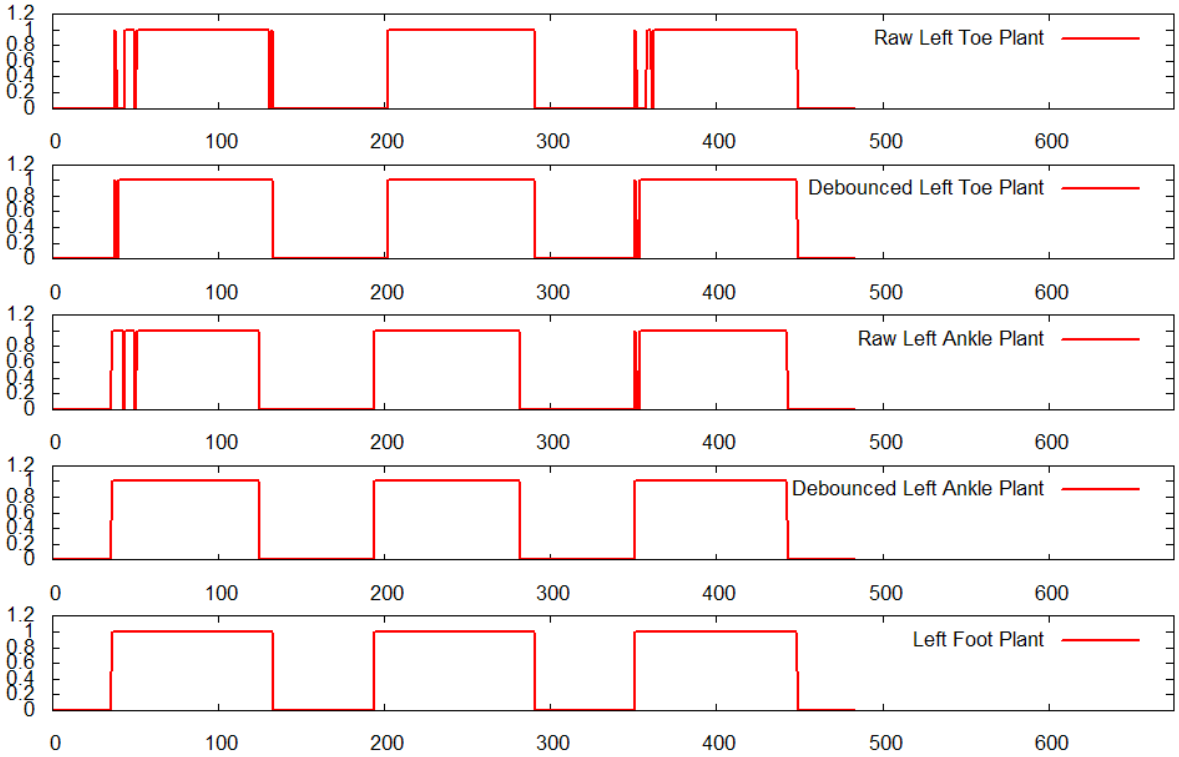



Figure 3.6: Foot plant annotation for "noisy" left foot signal of a walk along with corresponding left toe and ankle signals. From top to bottom, the first graph shows noisy toe annotation. The second graph shows de-bounced toe annotation. The third graph shows "noisy" ankle annotation. The fourth graph shows de-bounced ankle annotation. The fifth graph shows the combined foot plant annotation for the left foot.

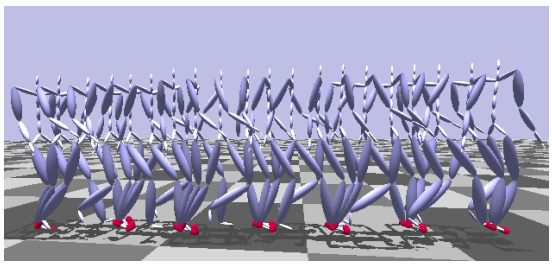
| | |
|---|---|
| Precision: 0.970120 | Recall: 0.972731 |
| Specificity: 0.931725 | Accuracy: 0.960224 |

No two human observers marked the same interval of frames as planted. However, our computed foot plants showed 97% overlap. No foot plant occurence was missed by our algorithm.

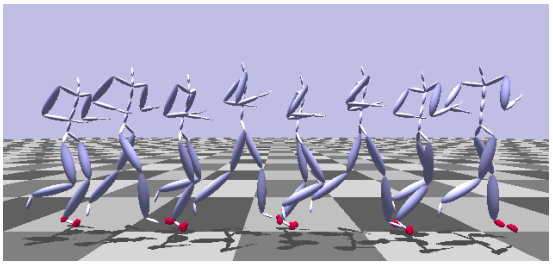
On completion of the foot plant identification stage, we have clips annotated with ankle, toe and foot plant information. Next we classify the clips into different locomotive types.

## 3.2 Clip Classification

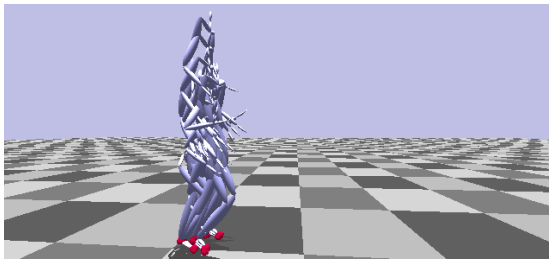
The clip classification stage differentiates the input clips in to one of the following types: *stand, walk, run, jump* and *other*. The *other* category is used to represent clips not identified by our classifier. This operation follows the foot plant identification. Our classification is based on the



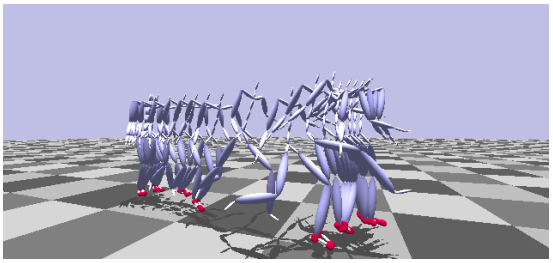
(a) Walk animation.



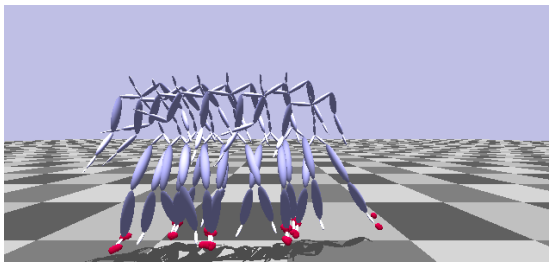
(b) Run animation.



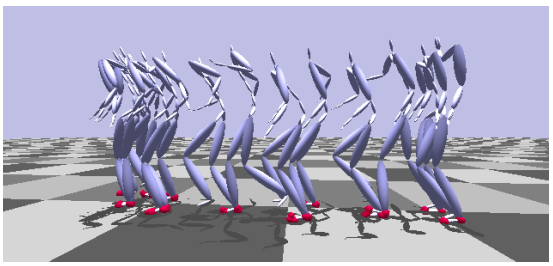
(c) Jump animation.



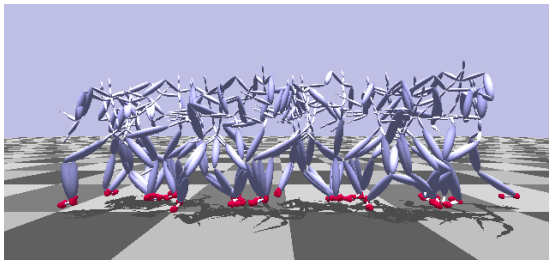
(d) Animation of a football kick.



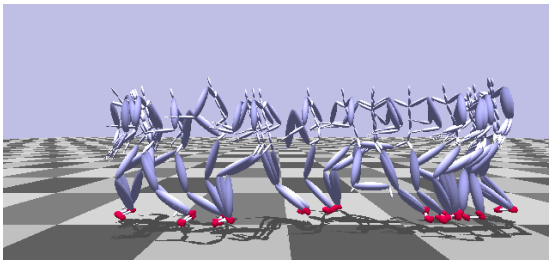
(e) Animation of a basketball dribble.



(f) Animation of hopscotch.



(g) Martial art form Mawashigeri.



(h) Martial art front kick.

Figure 3.7: Experimental results for automatic foot plant annotation algorithm. The identified foot plants are indicated by red spheres. These results are best seen in the accompanying video.

observation that each of the clip categories exhibit distinct foot plant patterns. Bio-mechanics literature defines walk as a gait wherein at least one foot is always in contact with the ground with short phases of "double support" in between when both feet are on ground [48], [28]. A

Table 3.2: Sample data used with automatic foot plant detection.

| Sr. No. | Motion Type |
|---|---|
| 1 | Basketball |
| 2 | Walk |
| 3 | Extended stride walk |
| 4 | Football |
| 5 | Jump |
| 6 | Walk |
| 7 | Run |
| 8 | Indian dance - (Bharatanatyam) |
| 9 | Banana peel slip |
| 10 | Moon walk |
| 11 | Wide leg roll |
| 12 | Martial arts - Basai |
| 13 | Martial arts - front kick |
| 14 | Martial arts - mawashigeri |
| 15 | Hopscotch |
| 16 | Macarena Dance |
| 17 | Sneak |

run is defined as a series of bouncing impacts with the ground that are usually alternated with aerial phases when neither foot is in contact with the ground [28]. We define 'stand' to be a state when both feet are on the ground for an extended period of time. We define 'jump' as hopping motion using one or both feet but not run. Note that our definition of jump includes one legged hops but excludes motions such as acrobatic pole vaults and running jumps. With these definitions, we construct regular expressions to identify each type of motion clip.

We denoting the character's foot plant state by the following tokens

'L' - The characters left foot is planted and the right is in flight

'R' - The characters right foot is planted and the left is in flight

'B' - Both of the characters feet are planted

'N' - Both of the characters feet are in flight

Using these tokens we have encode stand, walk, run and jump as:

```
Stand: B
;
Walk: (B L B R)+
| (B R B L)+
```

```
| (L B R B)+
| (R B L B)+
;
Run:  (L N R N)+
|(R N L N)+
;
Jump: (B N)+ B
| (L N)+L
| (R N)+R
;
```

In this grammar, each token represents a foot plant configuration. Transitions are caused by change in foot plant state configuration. The character continues to stay in its present state as long as the foot plant configuration does not change. In addition to this simple grammar we need two more pieces of information to determine the locomotive state of a clips frames. These are the initial state, and a way to determine change in locomotive state such as a change from walk to run. We solve the problem using a state machine.

### 3.2.1 State Machine

Our state machine comprises of four top level states, each containing entry and exit actions, internal substrates and internal transitions. The four top level states are: UNKNOWN, WALK, JUMP and RUN. Being in any of these states corresponds to being in one of their internal sub states.

The Unknown state is entered at the start of the clip and whenever an ambiguous state change is triggered by a transition from any of other three states. Later tokens are then used to infer the previously unknown state frames. This is accomplished by having the exit actions of each state update the action type of frames from entry frame to the exit frame.

See Appendix C for a complete description of our state machine implementation.

#### Determining the clip type

The clip type is determined by scanning through all its frames. Homogeneous clips – those containing frames of the same type – are classified as belonging to the common locomotive type of their frames. Heterogeneous clips – those contain frames of more than one type are classified as "*Other*."

#### Clip segmentation

Our clip classification procedure segments the clip based on their locomotive type. Our reuse and synthesis methods, however, do not use segmentation information. If required an animator may manually segment the clips and use them as inputs to our system.

#### Search by type

Our clip classification allows the animator to search the mocap database by a clips locomotive type. Instead of having a strict classification, it is possible to classify the clip by percentages. For instance in our example above (see C.2) 83% of the frames are classified as belonging to type Walk and the remaining 17% as belonging to type Jump. This clip can therefore be classified as 83% walk or 17% jump. We use strict classification for our synthesis and reuse. The % classification can be used to support fuzzy searches. A fuzzy search essentially allows an animator to query clips containing at least x% animation of a certain type. For example search for clips containing 75% run animation.

### 3.2.2 Experimental Results

Our primary use for the clip classifier is to select clips appropriate for further processing by our motion grafting scheme. As seen from the results below, our scheme is conservative and correctly identifies simple, normal locomotive motions of homogeneous type. The output does not contain *false positive* errors. This is highly desirable in order to ensure quality of our motion grafting synthesis.

Table 3.3: Clips of homogeneous locomotion type

| Sr. No. | Motion Description | Identified Loco-motion Type |
|---|---|---|
| 1 | Walk | Walk |
| 2 | Extended stride walk | Walk |
| 3 | Jump | Jump |
| 4 | Walk | Walk |
| 5 | Run | Run |
| 6 | Martial arts - front kick | Walk |
| 7 | Sneak | Walk |

Table 3.4: Clips of heterogeneous locomotion type (Other)

| Sr. No. | Motion Description | Locomotion Segments Detected |
|---|---|---|
| 1 | Basketball | Walk, Run |
| 2 | Football | Walk, Run |
| 3 | Indian dance - (Bharatanatyam) | Walk, Run |
| 4 | Hand spring | Run, Walk |
| 5 | Back flip twist and fall | Walk, Run |
| 6 | Banana peel slip | Walk, Run |
| 7 | Monkey back flip | Walk, Run |
| 8 | Moon walk | Walk, Run |
| 9 | Wide leg roll | Jump, Walk, Run |

### 3.2.3 Discussion

Our classification scheme classifies the generic locomotion type of a particular animation. The concrete type tends to be more subjective. For instance the animation of a character stomping its legs in place will be classified by our scheme as walk while a human observer may prefer to label it as stand. A sneaking motion will be classified as walk while human observers will recognize it as sneak. Some types of dance motion will be classified as walk, some as run and others as *others*. For example, our classifier thinks that the Indian dance form Bharatanatyam is a kind of walk. The classification depends on the foot plants in the motion. To the best of our knowledge, no known scheme can successfully classify all types of motion accurately. In spite of these obvious shortcomings, we find that our classifier does classify most regular input

appropriately and works well to satisfy the needs of our motion grafting synthesis.

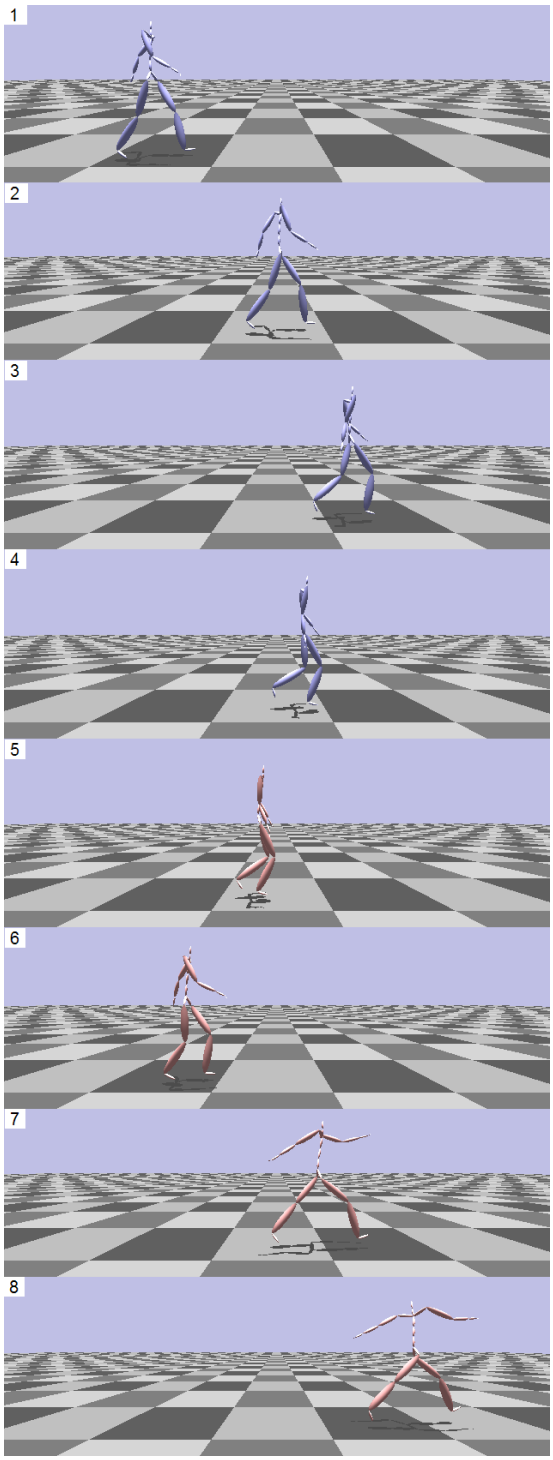



Figure 3.8: Figure show glide interpolation artifact caused by interpolation of absolute root DOF values. The images from top to bottom show progressive frames of a blend between two clips (colored blue and magenta). The interpolation starts at frame 3 (from top) and continues till frame 6. During this interval the character glides back unnaturally even though the animation is a forward back. It then moves forward again after the transition is complete (frames 7 & 8). This artifact is best demonstrated in the accompanying video.

## 3.3 Interpolation synthesis

In this section we discuss our motion interpolation scheme. We synthesise new motion by concatenation motion from different clips and blending the transitions. The transition blend is created by interpolating values of corresponding DOF's of the two clips over the blending interval. If the transition takes place at $frame_i$ in clip 1 and $frame_j$ in clip 2, then the transition interval is centered over frames $i$ and $j$. We use linear interpolation for position and displacements. We interpolate angles using spherical linear interpolation. We vary the interpolation weights linearly from clip 1 to clip 2 over the blend interval.

### 3.3.1 Interpolation Artifacts

Interpolating DOF values directly produces interpolation artifacts. For example blending between characters at different root position synthesizes an unnatural glide. An example of this artifact is shown in Figure 3.8 and in the accompanying video. Blending between a walk turning left and a walk turning right does not produce a straight walk as one might expect but results in chaotic motion. A similar artifact occurs for rotation, where the interpolation causes oscillation in the direction or speed of rotation. A solution to this problem is discussed in [54]. We use a different scheme to solve the problem. For the root joint, we interpolate between relative per frame displacements and per Y rotation angle. For each frame, we compute the relative

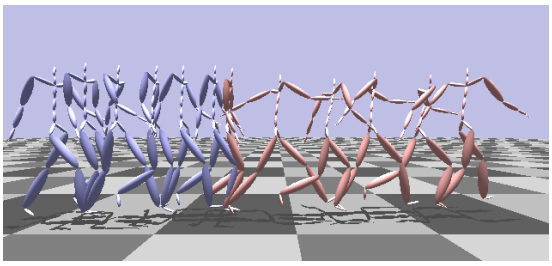


Figure 3.9: Correct synthesis free of interpolation artifacts, using relative displacements and relative Y angle interpolation for the root joints.

displacements and relative rotation of the root pose about the Y axis. For the first frame of our synthesis we use the absolute position and rotation from out start frame. For each frame thereafter, we compute the new root position by adding the relative displacement for the previous frame. Similarly we compute new Y rotation angle by adding the delta angle computed for the previous frame. For the transition blend interval, we obtain the desired displacement

and Y rotation values by interpolating the relative displacements and Y rotation angles of the corresponding frames from the two clips transition clips. For non-root joints, we interpolate absolute DOF values.The resulting synthesis is free of interpolation artifacts as shown in Figure 3.3.1 and also in the accompanying video.

# Chapter 4

# Multi-Clip Motion Re-sequencing

In this chapter we describe our multi-clip motion re-sequencing and cyclic motion synthesis techniques. We view animation as a sequence of connected motion segments joined together by smooth transitions. Changing the playback order of segments results in new animation. We describe *cluster graph's* – the data structure we use to detect motion segments and capture transition information.

## 4.1 Motivation

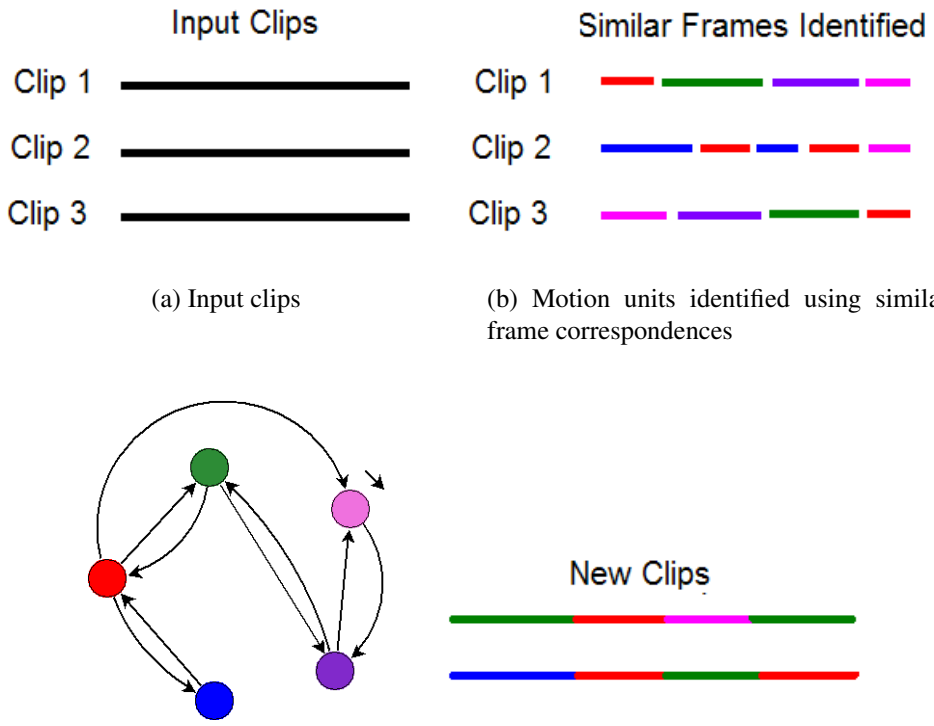
In an animation, an actor performs one or more activities. For example, consider the animation of an actor climbing a stair and entering a room. Let us say that, to accomplish this task, the actor walks four steps in front, climbs ten steps of the staircase, walks six steps turning left, stands, opens a door and walk two steps forward. The set of distinct activities he performs are – straight walk, staircase climb, left turn walk, stand and open door. Each of these can be thought of as a motion unit. *A motion unit is an animation sequence depicting a distinct activity.* The recorded motion for this animation contains the aforementioned motion units organized in a particular order. The animation transitions smoothly from one activity to another at the joins. Now consider a slight variation of the scenario above. Let the door that guards the room entrance be moved to the entrance of the staircase. The new sequence of motion units required to synthesize animation for this scenario is – straight walk (i.e. walk four steps in front), stand, open door, staircase climb, left turn walk (i.e. walk six steps turning left)

and straight walk (i.e. walk two steps forward). The set of motion units is identical to the earlier scenario but ordered differently. The new animation can, therefore, be constructed by reusing the existing motion units provided it is possible to reconnect them using smooth natural transitions. In this example, the recorded mocap has transitions straight walk to climb, climb to left turn walk, left turn walk to stand, stand to open door and open door to straight walk. In the second scenario, re-sequencing requires three new transitions which do exist in the recording, viz. straight walk to stand, open door to staircase climb and left turn walk to straight walk. Given a set of motion captured clips, it is likely that some of these "missing" transitions exist as part of other recordings. Such recorded transitions can be reused for synthesis.

Note here, that some activities are cyclic in nature, for example walk and climb. The example uses six walk cycles – four to walk up to the staircase and two cycles to enter the room. Given one walk cycle, the remaining can be synthesized by looping. Detecting cyclic motion units can therefore enable synthesizing more such cycles when required.

When capturing motion for movies, goals such as – enter the room at the top of the stair, kick the ball, jump over the hurdle – form the basis for capture. Each captured clip contains multiple motion units. Given a collection of such mocap clips, a natural way to reuse motion is to identify motion units and synthesize new animation by stitching them together in a different order. It is important to maintain smooth activity transitions during re-synthesis. As explained earlier, one way to achieve this is to reuse the transitions present in the original recording. In order to extract such transitions, it is required to detect the motion units in different clips and establish correspondence between similar ones. Once such correspondence is established, a graph of motion units can be created with different nodes interconnected using recorded transitions. Figure 4.1 illustrates this idea. An input set of clips is shown in Figure 4.1(a). Individual motion units in the input clips are color coded in Figure 4.1(b). Note that, in the figure, the seams of differently colored line segments denote the transitions between motion units (activities). For example, clip 1 contains a transitions from the motion unit coded in red to the motion unit in green. Other transitions occur from green to violet and from violet to pink. In clip 2 there exist transitions from blue to red, red to blue and red to pink. Similarly for clip 3. The graph of motion units and transitions for this set of clips is shown in Figure 4.1(c). The nodes of this graph represent motion units and the arcs represent inferred transitions. New motion is synthesized by traversing this graph in the desired order. Clips synthesised this way are shown in Figure

4.1(d).




(a) Input clips

(b) Motion units identified using similar frame correspondences

(c) The transition graph

(d) Motion re-sequencing by traversing transition graph in different order

Figure 4.1: The motion database as a graph motion units and interconnecting transitions.

Animations differ not just in the sequence of activities but also in the activities being performed. For example, consider the scenario where the actor of our staircase example, walks out of the house and kicks a ball lying just outside. Synthesizing this animation would require a kick motion unit, which is not present in the earlier recording. In such cases re-sequencing alone cannot satisfy the new requirements and new motion needs to be acquired. Similarly, new motion acquisition is required if a desired transition does not exist in existing recordings.

Games have, for long, used on the fly motion re-sequencing to generate animation. A data structure, sometimes referred to as *Move Tree*, is used for this purpose [68], [56]. However move trees are created manually. Motion for creating move trees is acquired carefully so as to assist later manual editing. The technique employed is to start off each activity from a common pose and return back to this pose at the end. Transitions are seamless as all activities start and end at the the common pose. In [38], authors describe a technique to identify one or more such common poses. This behavior of returning to the same common pose is unnatural and merely

a convenience for creating motion that allows seamless stitching at runtime. In reality, actors follow different transition paths. Our technique does not use the notion of common poses and utilizes transitions recorded in the original motion captured clips for synthesis.

In the remainder of the chapter we describe cluster graphs and their construction. Following this we describe our motion re-sequencing and cyclic motion synthesis. We conclude with a discussion of other tree based motion synthesis techniques.

## 4.2 Cluster Graph

The *cluster graph* is a data structure that detects similarity in-between arbitrary clips and records the transition information between them. It consists of nodes and edges as shown in 4.2. Each node of the cluster graph groups together similar frames from potentially different clips in the mocap database. The granularity of a cluster graph is much finer than the "activity" defined previously. The clustering is based on frame similarity rather than on activity. Such clustering segments motion into frame sequences with contained frame counts ranging from a single frame to entire clips. This is controlled by the specified similarity error threshold. In practice, an activity such as a walk cycle, gets clustered into five to ten clusters. We treat motion in a cluster as our motion unit.

In order to connect cluster graph nodes, we sort all frames within a cluster graph node by their clip and frame indices. These sorted frames are then grouped together into one or more contiguous *clip-frame* sequences. *A clip-frame sequence is a contiguous collection of frames belonging to one single mocap clip in the database.* Since frames from different clips get clustered in a node, potentially, each node contains clip-frame sequences from one or more clips. Figure 4.3 shows a cluster graph node with clip-frame sequences from different clips clustered together.

The edges between nodes are determined by the natural ordering of frames in the original clips and the clusters to which they belong. The in-transition and out-transition edges for each cluster graph node are determined as by observing the frame adjacent the first frame and the last frame a clip-frame sequences contained in this node. The in-transition edges come from the the clusters containing the prior frames. The out-transition edges go to the clusters containing the

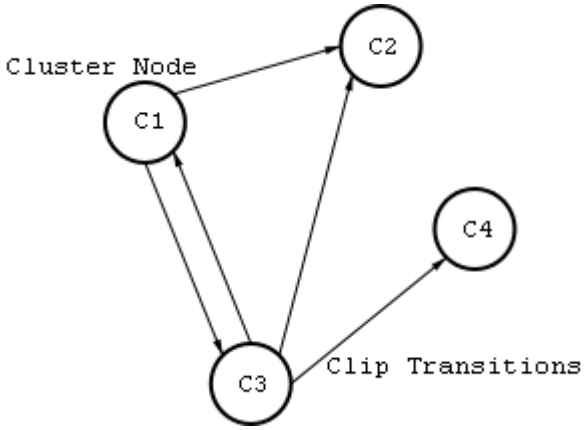



Figure 4.2: A cluster graph.

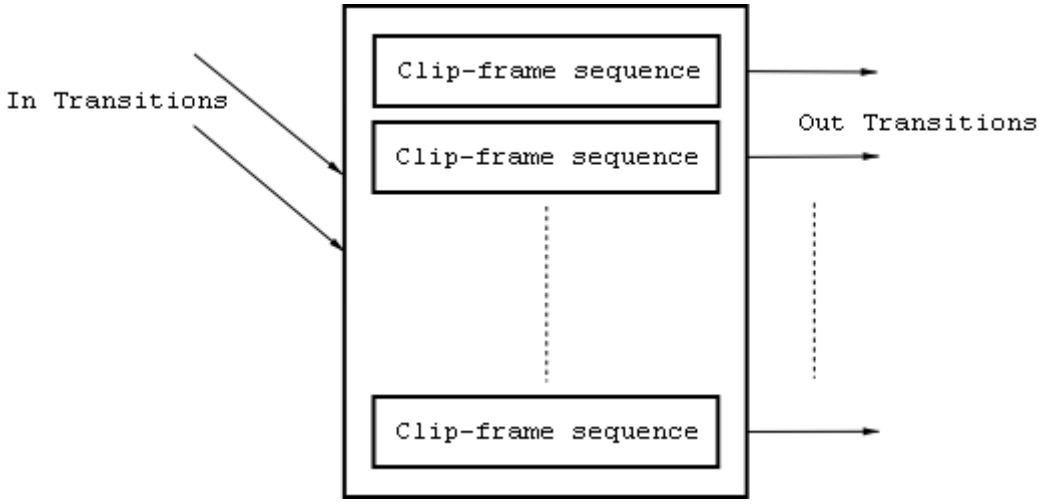



Figure 4.3: A cluster graph node.

succeeding frame [1]. For example if frame $i$ of clip A is clustered in node 1 and frame $(i+1)$ is clustered in node 2, then there exists an out-edge from node 1 into node 2 (an in-edge for node 2). The set of in-edges and out-edges of each cluster node is the set of its unique in-edges and out-edges.

### 4.2.1 Frame similarity metric

In order to detect similar motion frames, we define a frame similarity metric for comparing two frames of motion. Since each frame is essentially a vector of values, containing the root's position and the orientation of each joint, a simple approach is to compute a weighted norm of these vectors. However, this metric fails to address several important issues:

[1] The clip-frame sequence containing the first frame of a clip does not contain an in-transition edge. Similarly the clip-frame sequence containing the last frame of a clip does not contain the out-transition edge.

1. It fails to collect similar motion performed at different locations and headings. A characters motion is invariant to translation on the horizontal plane and to rotation about the vertical axis. For example see Figure 4.4, which shows a frame from the football kick sequence being performed at different positions in the scene.

2. It does not guarantee higher order continuity. Transitions matching only instantaneous frame positions demonstrate $G_0$ continuity and can be jerky. Better transitions result from good velocity and acceleration matches.

3. It fails to account for dissimilarities caused by gimbal lock artifacts. Individual raw angle values being equal does not conclusively determine an end effectors position because of gimbal lock artifacts.

4. It does not take constraint information into account. It fails to consider importance of joints. Not all end effectors contribute equally to a frames likeness. The posture of a humanoid actor depends primarily on the root, the head and the feet joints. Often interpolating hand positions generates acceptable results.

5. Similar frames in different clips can have different contact constraints. Transitioning between frames with different constraint states breaks believability.

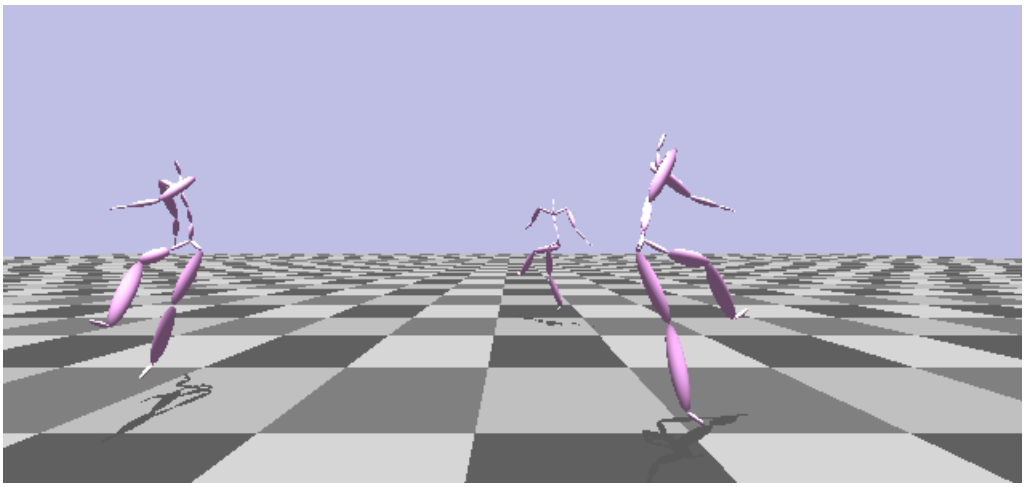


Figure 4.4: All three poses shows in this figure are identical except that they are positioned and oriented differently. Animated motion is invariant to translation along horizontal plane and rotation about the vertical axis. Directly comparing DOF values fails to cluster frames such as shown above.

Our frame similarity metric accounts for each of these issues as follows.

1. To account for invariance to translation along the horizontal plane and rotation about Y

axis, we translate all frames to a neutral reference coordinate system. This is constructed by translating the characters to $(0, y, 0)$ where $y$ is the original y position of the root and rotating the character such that its forward axis coincides with the X-axis.

2. To account for gimbal lock artifacts, we compare positions of the end-effectors in the neutral reference coordinate system.

3. To provide for better than $G_0$ continuity, we compute velocities and accelerations and include them in our frame similarity metric computation.

4. To account for unequal joint contribution, we weight the contribution of each joint.

5. To account for contact state differences we restrict frame comparisons and clustering to only those with similar foot plant states. Foot plant constraint states are explained in the Section 4.2.2.

We define the error metric as the sum of squares of the weighted Euclidean differences of positions, velocities and accelerations over all the end effector nodes. For some frames $i$ and $j$ in the motion capture database, let $p_{i_l}$ , $p_{j_l}$ denote the position, $v_{i_l}$, $v_{j_l}$ denote the velocity and $a_{i_l}$, $a_{j_l}$ the acceleration and $w_l$, $w'_l$ and $w''_l$ the weights for position, velocity and acceleration components of end effector $l$. The frame error metric, $E_{i,j}$, is computed as

$$E_{i,j} = \sum_{l=i}^{n_l} w_l (p_{i_l} - p_{j_l})^2 + \sum_{l=i}^{n_l} w'_l (v_{i_l} - v_{j_l})^2 + \sum_{l=i}^{n_l} w''_l (a_{i_l} - a_{j_l})^2 \tag{4.1}$$

### 4.2.2 Foot plant constraint states

We restrict clustering frames with dissimilar foot plant constraint states by comparing only those that have similar states. To detect the foot plant constraint states, we use the annotation created by our foot plant detection algorithm described in section 3.1. We distinguish between the following distinct foot plant states.

1. `L`: Left foot alone is planted.

2. `LB`: Both feet are planted and previous state was `L`.

3. `LN`: No feet is planted and previous state was `L`.

4. `R`: Right foot alone is planted.

5. `RB`: Boot feet are planted and previous state was `R`.

6. `RN`: No feet is planted and previous state was `R`.

7. `N`: No feet is planted and previous state was one of `LB` or `RB`.

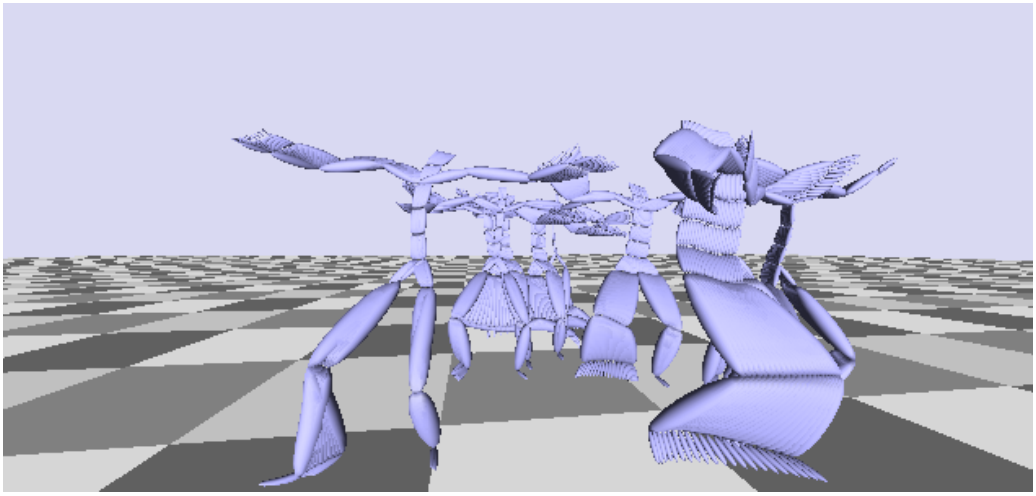


Figure 4.5: Figure shows frames from a Bharatanatyam performance clustered together using our frame similarity metric. Notice that similar frames at different positions and orientations are clustered together correctly.

Figure 4.5 shows similar poses from an Indian classical dance performance (Bharatanatyam) identified using our similarity metric.

### 4.2.3 Constructing cluster graphs

With the above similarity metric to compute distance between two frames, we use an algorithm reminiscent of Kruskal's minimum spanning tree algorithm to cluster frames. We first pre-process all $n$ given frames so that we have the distance set $D$ for all $\binom{n}{2}$ frames. Next, we sort all frames within each cluster graph node by their clip and frame indices. These sorted frames are then grouped together into one or more contiguous *clip-frame* sequences.

1. Sort $D$ into $\pi = (o_1, o_2, \ldots, o_k)$ by non-decreasing values.

2. Start with $F^0 = \{\}$.

3. Repeat Step 4 for $o_q = o_1, \ldots, o_k, k = \|D\|$.

4. Construct $F^q$ given $F^{q-1}$ as follows. Let $o_q$ correspond to frame pair $(i, j)$. If $o_q$ is small compared to a threshold, then

(a) Add frames $(i, j)$ to $F^{q-1}$ and refer to this pair as a single frame called $I_j$.

(b) Remove from $D$ all references to either $i$ or $j$.

(c) Represent joint positions of $I_j$ as average of those in $i$ and $j$.

(d) Compute distance $I_j$ for all frame pairs $(I_j, p)$, $p \neq i, k \neq j$. Insert these distances in $D$ maintaining the sorted order.

This algorithm is efficiently implemented. Specifically each insert and removal in Step 4 is $O(\log n)$.

At the end of this step we have a cluster tree with frames clustered into different nodes. Each tree node contains frames with different max error thresholds. The root of this tree represents the entire motion database. To generate the cluster graph nodes, we walk the tree and extract nodes with specified max error threshold. We then sort all frames within a cluster graph node by their clip and frame indices. These sorted frames are then grouped together into one or more contiguous *clip-frame* sequences. The edges between nodes are determined by the natural ordering of frames in the original clips and the clusters to which they belong. The in-transition and out-transition edges for each cluster graph node are determined as by observing the frame adjacent the first frame and the last frame a clip-frame sequences contained in this node. The in-transition edges come from the the clusters containing the prior frames. The out-transition edges go to the clusters containing the succeeding frame as explained earlier.

All frames contained in a cluster graph node are similar, within an error bound. However not all frame transitions have the same cost. We pre-compute good transition frames for inter clip transition by aligning clip-frame sequences within a cluster to a common alignment frame.

### 4.2.4 Alignment frames

Given any two clip-frame sequences there exist a frame pair which represent the best transition between those two clip-frame sequences. Transitioning at this frame pair generates smooth transition synthesis. Alignment frames are such best transition frames from each clip frame sequence. They represent the transition points from any other clip-frame sequence within the node.

Once motion clip-frame sequences are clustered into a cluster graph we pre-compute the align-

```
S = L_{1,2}
for i = 2 to m − 1 do
  templist = {}
  for each element (a, ..., idx) in S do
    for each element in L_i do
      if (idx, ∗) is an element in L_{i,i+1}, then
        Add (a, ..., idx, ∗) to templist.
      end if
    end for
  end for
  S = templist
end for
```

Figure 4.6: The alignment algorithm.

ment frames in each clip frame sequence within a node. A brute force way of computing alignment frames is to compute best transition frames for every clip-frame sequence combination and store the list. This requires $O(m^2)$ clip frame sequence comparisons. We use the following greedy algorithm instead which requires $O(m-1)$ clip frame sequence comparisons. First, we find the best point of transition between two motion clip-frame sequences. The decision is based on the distances between corresponding frames of the two clip-frame sequences computed using our frame similarity metric. We choose the frame pair for which the frame errors are minimal. Next, we iteratively determine an ordered $m$-tuple ($m$ is the number of clip-frame sequences in the cluster node) of frame indices, such that these frames in the corresponding clip-frame sequences are the closest matches for each other.

These alignment frames serve as good points for both in and out transitions for each clip frame sequence within the cluster. Figure 4.7 displays results of this implementation on motion sequences. The new curve formed is not only continuous at the point of transition but also smooth.

**Details**

We iteratively apply a correlation-based technique on clip-frame sequences $M_i$ and $M_{i+1}$ and get a list $L_{i,i+1}$ of $k$ most suitable pairs of frame indices. This $L_{i,i+1}$ contains several pairs of indices $(a,b)$, such that frame $f_a^i$ and $f_b^{i+1}$ resemble each other closely. Once all the $L_{i,i+1}$ are computed, we look for common indices in adjacent $L_{i,i+1}$.

If $\mathscr{S}$ is non-empty, it contains at least one index sequence of length $n$. In general, it contains the desired list of indices $j_1, j_2, \ldots, j_n$. The frames with these indices in the corresponding clip-

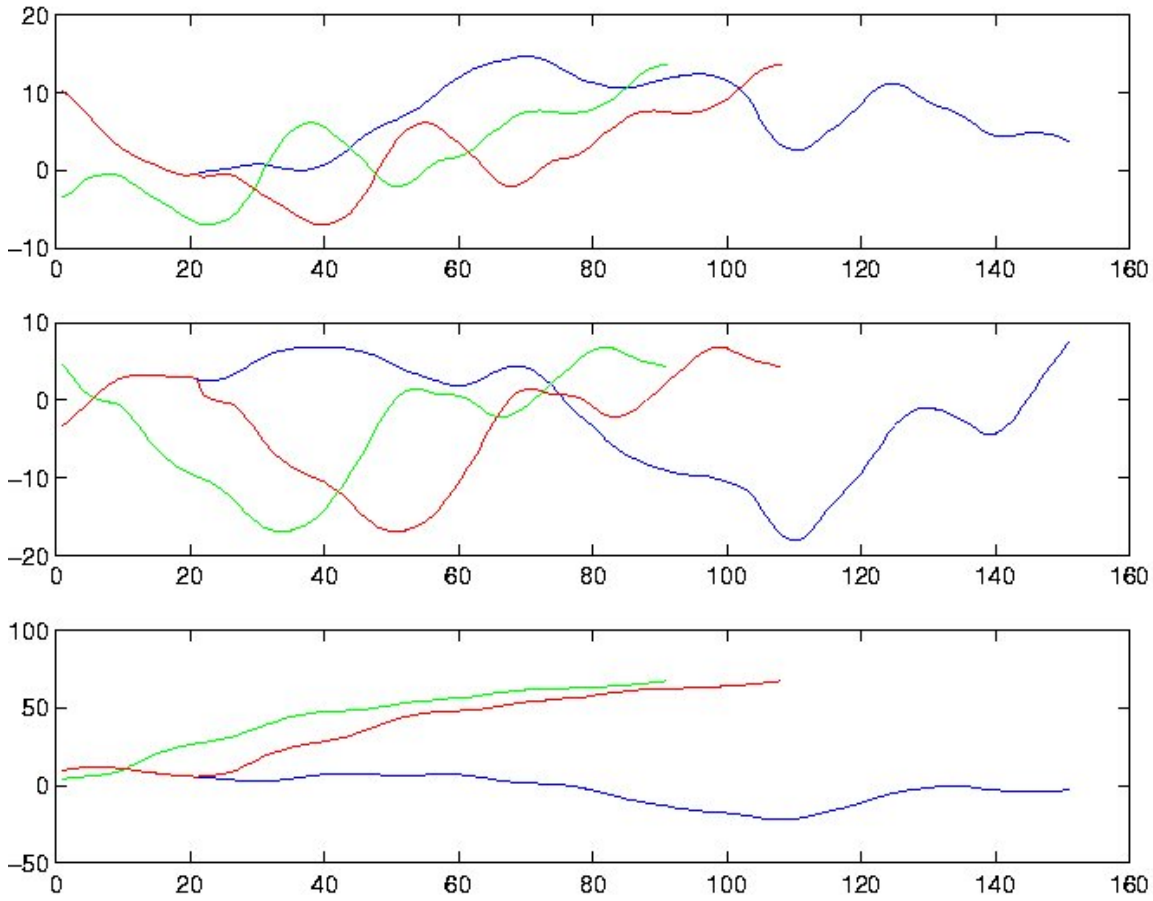



Figure 4.7: Multiple sequence transition generated using the best match alignment frame tuple. Graphs shows smooth synthesized transitions for three different DOF's. We transition from the blue to the green input sequences. The red sequence is the output sequence.

frame sequences would be closest to one another. Thus if we now want to transition from clip $M_4$ to $M_7$, we play $M_4$ till the $j_4^{th}$ frame and then switch to $(j_7+1)^{th}$ frame in clip $M_7$. The transition is without any jerks and bumps.

## 4.3 Synthesis

New motion is synthesized by walking the cluster graph and transitioning from one clip-frame sequence to another clip-frame sequence under animator control. We use the interpolation technique described in Section 3.3 to render new motion. We allow the animator to select the blending interval. Selecting a blending interval in the range of 1-2 seconds yields satisfactory results. However, it is important to account for different foot plant constraint states during blending to avoid interpolation artifacts. Our constraint handling is explained in the following section.

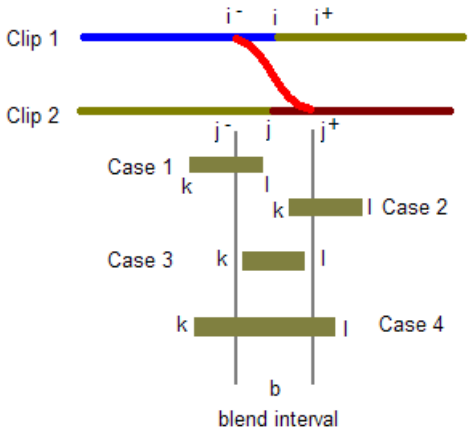



Figure 4.8: Different foot plant constraint cases that occur over the blend interval.

### 4.3.1 Handling different constraint states

We handle foot plant constraints separately. Our scheme is similar to one used in [61]. The blend interval can contain frames of different foot plant constraint states. Figure 4.8 shows the four possible combinations that result:

Suppose the transition is from $frame_i$ of clip 1 to $frame_j$ of clip 2. A blend interval $b$ is created centered over the transition frames. Let $frame_{i^-}$ and $frame_{j^-}$ denote the start of the interval and frames $frame_{i^+}$ and $frame_{j^+}$ denote the end of the blend interval on clips 1 and 2 respectively. If there is an overlap between the blending interval and the contact interval [k, l], we establish the constraint based on the following cases:

1. **Case 1:** $k \leq i^- < l < i^+$ The constraint lies over the start of the blending interval and between frame $i^-$ and frame $l$ the foot should follow the trajectory of the motion sequence before the transition.

2. **Case 2:** $j^- < k < j^+ \leq l$ The constraint lies over the end of the blending interval and between frame $k$ and frame $k^+$ the foot should follow the trajectory of the motion sequence after the transition.

3. **Case 3:** $j^- < k < l < j^+$ The contact interval is contained within the blending interval and the trajectory of the foot can be taken from either side. Our implementation chooses the closer side.

4. **Case 4:** $k \leq i^- < i^+ < l$ or $k \leq j^- < j^+ < l$. Here the constraint lies over both

boundaries and the system interpolates linearly between source and destination sequences allowing the foot to slip.

### 4.3.2 Motion re-sequencing

Every cluster node with more than one out edge represents a transition between clips. We use free-form, un-annotated input motion clips. Our synthesis, therefore, requires animator input. In theory, a random cluster graph walk can synthesize new motion. However such motion is not likely to be semantically meaningful.

To synthesize new motion, the animator selects the clips with the desired motion units. He then selects approximate transition points for each transition and queries the system for transitions to the destination motion. The system searches the cluster graph for clusters containing clip-frame sequences from the source and target motion frames near the animator specified frame. If a cluster is found containing the exact exit and entry frame, the alignment from from this cluster is used for transition. If not, a search is made for the nearest transition such that exit frame of previous action transitions to some frame in the destination motion, that lies before the start of the target action. If no such option is found, a search is made to find transitions containing intermediate clusters, passing through clips other than those suggested by the animator. This result is presented to the animator for selection. Once the desired transitions are selected, animation is synthesized taking into consideration foot plant constraints as described in previous section.

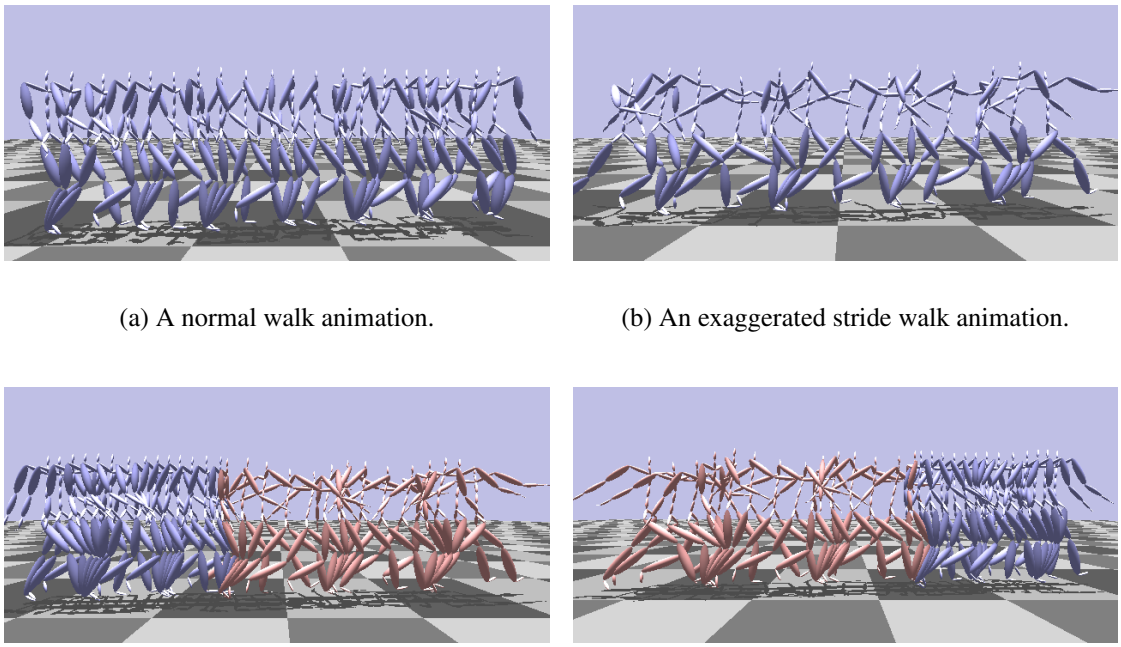
### 4.3.3 Cyclic motion synthesis

Clustering inherently detects such loops. A cluster node containing more than one contiguous clip frame sequence from the same clip indicates the presence of a loop. Sorting the frame sequences within a cluster node allows for easy identification of loop-able frames. A loop exists from every clip-frame sequence to other clip-frame sequences from the same clip, preceding it in time. For example let there be three clip-frame sequences Seq1, Seq2 and Seq3 in the cluster graph, all belonging to the same clip. Motion cycles can be synthesized by looping between the sequence combinations Seq1-Seq2, Seq1-Seq3, Seq2-Seq3[2]. The loop exists because, the

[2]Loop Seq1-Seq3 contains Seq2 as well.

frames in the ending sequence, for example,. Seq2 in loop Seq1-Seq2 are similar to frames in the start sequence, Seq1 and the frames occur at different points in time in the original clip.

To synthesize motion cycles, the animator first selects the start clip. At the desired looping frame, the animator queries our system for looping transitions. The system presents available options to the animator. Once a selection is made, the system synthesizes the loops taking into consideration foot plant constraints. The animator specifies the number of cycles to generate.



(a) A normal walk animation.

(b) An exaggerated stride walk animation.

(c) A normal walk to exaggerated stride walk transition.

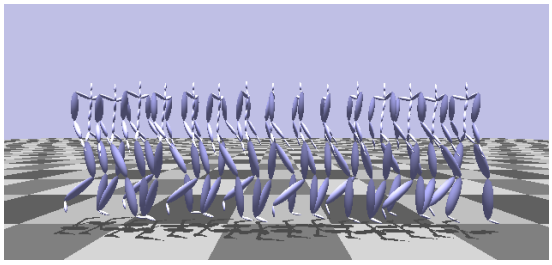
(d) A exaggerated stride walk to normal walk transition.

Figure 4.9: Screen shot of transition between a walk clip and a walk clip with exaggerated stride. This results is best viewed in the accompanying video.

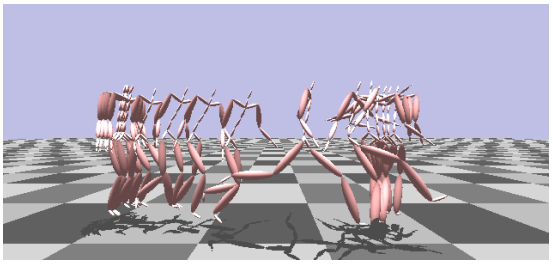
## 4.4 Experimental results and discussion

This chapter presented our technique for high level motion reuse using action re-sequencing and cyclic motion synthesis. Figure 4.9 and 4.10 show sample transitions synthesized using our techniques. Our experimental results are best viewed in the accompanying CD which contains video of our synthesized transitions and motion loops.

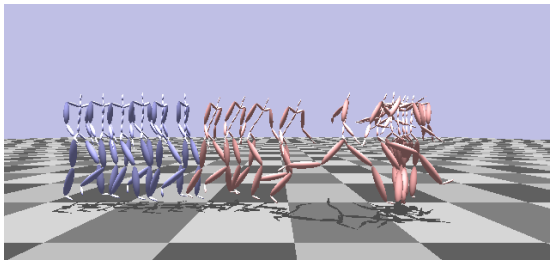
Our basic strategy is to identify similarity in motion across different clips and make use of the naturally recorded transitions between different action sequences across the input database.



(a) A normal walk animation.



(b) A football kick animation.



(c) A normal walk to football kick transition.

Figure 4.10: Screen shot of transition between a walk clip and a football kick clip. This results is best viewed in the accompanying video.

Our method for cluster graph generation is automatic and the only input required is the error threshold for each cluster node. Our synthesis is interactive and requires animator input. An alternative to human input is to traverse the cluster graph using an objective function. Such an objective function may be constructed to meet scene constraints such that the character starts from location A and ends up at location B passing through several intermediate posses. Other authors [5] and [56] have explored trees based data structures with an aim to automatically generate motion meeting specified with constraints. We argue that such motion is hard to define precisely and can only be used at best as a coarse starting point which then needs to be refined by the animator.

A number of differences exists between other tree based approaches and our work. These are as described below.

1. Transitions: While other authors synthesize transitions to improve connectivity when none exits, we only use existing recorded transitions between motions. Variety is ensured by the fact that a sufficiently large motion database will contain many of the often used action sequences. Therefore transitions for such sequences will have been already recorded.

2. Granularity of transitions: Most authors restrict the number of transitions recorded per clip. Very often the restriction is set to one. Only one other clips can be transitioned onto from any given clip. This is a severe drawback. Our implementation keeps all transitions. Since we cluster frames together in the cluster graph data structure and record only distinct transitions between cluster nodes, a large number or redundant transitions do not need to be stored. We can reconstruct actual best transition for frames within a single cluster on demand or use the pre-computed alignment frames to synthesize transition.

3. Frame similarity metric: Our frame similarity metric uses a positions, velocities and acceleration of end effectors in a neutral frame of reference. Other authors have used metrics such as generating point clouds to guarantee $C_1$ continuity. Our having accounted for velocity and acceleration, allows us to generate smooth motion.

4. Intent: We aim to provide the animator an interactive tool to enable high level motion reuse. Other authors have automated the final motion composition process. While this results in good looking motion, its final utility is not established. Investigations into the goodness of such synthesis is also a topic of research [89].

# Chapter 5

# Locomotive Motion Grafting

In this chapter we describe our technique to reuse motion data at the sub-hierarchy level. We view every mocap animation as a composition of multiple parallel actions. Our objective is to increase an actors action repertoire by splitting the mocap animation into upper and lower body actions and synthesizing all possible new combinations. We describe *motion grafting*, our heuristic driven deterministic technique to synthesize believable locomotive motion grafts.

## 5.1 Motivation

Human beings are active entities capable of performing innumerable actions. A characteristic of us humans is that we perform multiple distinct actions in parallel. For example an actor can wave one of his hands while simultaneously grasping an object with the other. The wave action can be combined with different locomotion such as walk or run. It can also be performed in different postures like sitting, standing or lying down. Notice that different types of parallelism are at play – posture level (waving while standing or sitting), base locomotion level (waving walking or running) and sub-limb level (waving while grasping an object). The performer is free to choose which actions to perform simultaneously. Our basic idea is to exploit this parallelism and extend an actors action repertoire by synthesizing new action combinations from an existing mocap database.

Every action is intrinsically parameterised by style, speed, posture and base locomotion. In the example above, the hand wave can be performed by a gentle movement of the palm or involve

exaggerated full arm movement – a matter of style. It may be performed at different speeds as in a slow wave versus a fast and vigorous one. The performer can be walking straight or with a stoop, a matter of posture. Finally he may be standing, sitting or running and not necessarily just walking. Variations of the same basic motion can depend upon internal factors such as moods of the performer and external factors such interaction contexts or physical constraints. The number of such combinations is potentially huge and poses a combinatorial problem for mocap acquisition. Table 5.1 shows an example of such combinatorial variations with two values each for parameters "locomotion," "style" and "speed."

Table 5.1: Combinations of wave and base motions.

| Locomotion | Style | Speed |
|---|---|---|
| Walk | Normal | Slow |
| Walk | Normal | Fast |
| Walk | Exaggerated | Slow |
| Walk | Exaggerated | Fast |
| Run | Normal | Slow |
| Run | Normal | Fast |
| Run | Exaggerated | Slow |
| Run | Exaggerated | Fast |

### 5.1.1 Mocap acquisition problem

As mocap performance encodes whole body motion, every desired combination of action, style, speed and posture needs to be acquired separately. This has the disadvantage that virtual actors are limited by the number of distinct variations captured. This drawback directly impacts character animation in interactive applications like games, which use minimal set of actions to drive their virtual characters. For off line applications such as movies, the problem manifests itself as an explosive growth in the number separate combinations that need to be captured. Since the action repertoire is large, it becomes impractical to capture "every conceivable" combination. However, each captured motion, does contain, actions performed in some style, speed, posture and base locomotion. An interesting alternative is to attempt re-synthesis from existing motion data.

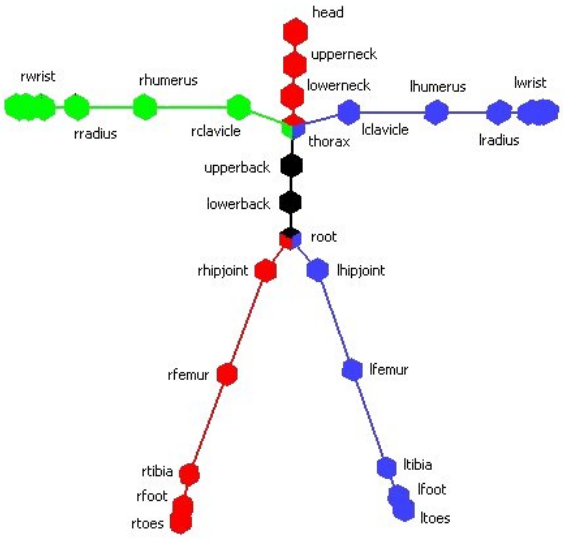
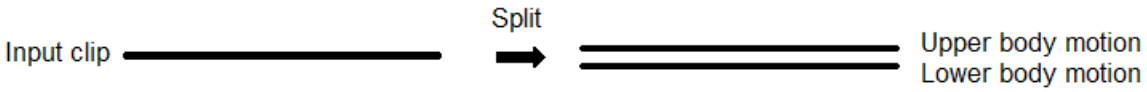


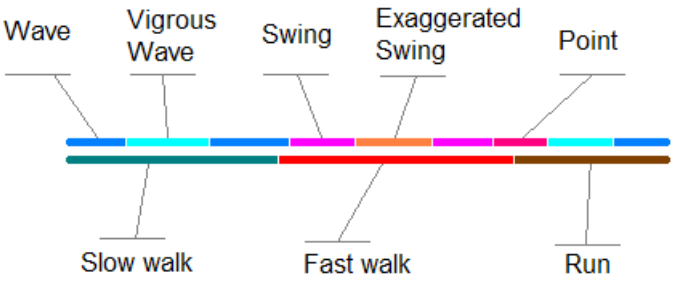

Figure 5.1: Hierarchical skeleton with independent kinematics chains demarcated.

### 5.1.2 Independent kinematic chains

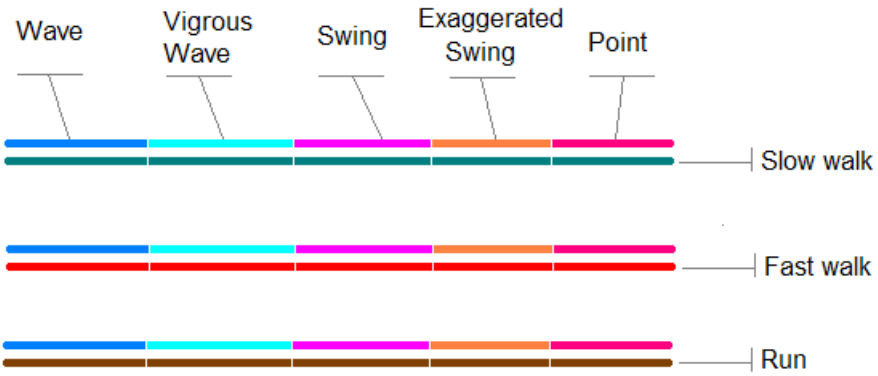
In order to perform combinatorial synthesis of parallel actions, we need to be able to extract these motions. One way to recover these is to segment motion by limbs. We observe that the skeletal hierarchy used to acquire mocap data and subsequently drive the character animation contains several kinematic chains as shown in Figure 5.1. Each of these corresponds to independently manoeuvrable part of the human body. For example nodes [root, lhipjoint, lfemur, ltibia, lfoot, ltoes] constitute a kinematics chain corresponding to the left leg. For different chains having a common root, the kinematic inter-dependency is restricted to the roots DOF variables. The chains are unaffected by DOF variables at non root nodes. We call such chains *independent kinematic chains*. The motion signals captured for each independent chain constitutes an *independent actions*. We think of a motion capture sequence to be composed of several independent actions running in parallel. However, in practice, kinematic chains exhibit strong correlation for certain motions and none at all for others. We discuss such correlation in more detail in Section 5.1.4.




(a) Split mocap clip into upper and lower body motion




(b) Annotate upper and lower body actions




(c) Synthesize all combinations of upper and lower body motions

Figure 5.2: An illustration of the basic parallel action synthesis concept.

### 5.1.3 Parallel action synthesis

We classify the different independent kinematic chains into two groups based on their correlation characteristics – upper body group and lower body group. Kinematic chains in these groups show stronger correlation within the group and weaker correlation across groups. Upper body group contains kinematic chains corresponding to the hands, chest and head. The lower body group contains chains corresponding to the legs. We split the original input motions into corresponding upper and lower body motion. Figure 5.2 illustrates the basic concept of parallel

action synthesis. Note that the motion annotation in this figure only serves to aid explanation. Our input data is un-annotated. New upper and lower body motion combinations are then combinatorially synthesized. Table 5.2 shows such combinations for motion in figure 5.2. Some of these are less natural, but still possible.

| | Wave | Vigorous Wave | Swing | Exaggerated Swing | Point |
|---|---|---|---|---|---|
| Slow Walk | Slow walk wave | Slow walk vigorous wave | Slow walk swing | Slow walk exaggerated swing | Slow walk point |
| Fast Walk | Fast walk wave | Fast walk vigorous wave | Fast walk swing | Fast walk exaggerated swing | Fast walk point |
| Run | Run wave | Run vigorous wave | Run swing | Run exaggerated swing | Run point |

Table 5.2: All combinations of upper and lower body actions for motion in Figure 5.2.

The upper body motion can be further split into motions for right arm, left arm, head and torso as shown in Figure 5.3. The re-synthesis procedure can be applied recursively at this level of subdivision.


Figure 5.3: Subdividing upper body motion further .

### 5.1.4 Cross body Correlation

An obvious way to composite parallel action is to cut and paste limb motion segments across existing mocap clips. However, naively doing so results in motions that do not look human. This is due to lack of cross-body correlation that exists in a real performance. Correlation occurs either due to the active intent or as a result of passive reflex. Example of intentional correlation is seen in movements such as relaxed walking, where arms swing out of phase with the legs for energy reasons [47]. This is a gait that is chosen by the actor, and can be broken

at will - for example, to reach out or wave when walking. Reflex correlations occur as a result of the body reacting to maintain equilibrium. For example, the arms may be extended out in order to balance a fall. If this arm movement is replaced with some other arm movement, the resulting motion may not look believable in a human. In either case the correlations play an important role in determining believability of the final motion and need to be accounted for.

### 5.1.5 Our method

While researchers in the field of behavioral animation have built systems, [11] [84], that take advantage of parallelism in actions, attempts at automatic parallel action composition are recent [97], [47], [40]. Our solution is based on a scheme that breaks down the original problem into manageable parts as shown in 5.9. Our method requires identifying independent actions. Once identified, these are available for re-compositing with different base locomotion taking into account cross body correlation. We call this recomposition process – *motion grafting*. Our scheme is deterministic and designed for locomotive motion. We restrict our graft synthesis to "homogeneous" clips containing stand, walk, run or jump motions. By homogeneous we mean that the entire clip has the same locomotive motion. The rest of this chapter describes our our correlation technique and our heuristic method for generating high quality graft motion.

## 5.2 Identifying Independent Kinematic Chains

We use independent kinematic chains 5.1.2 to extract parallel actions. Though easy to specify manually, we automatically detect kinematic chains for two main reasons. The first is that our technique is not restricted to humanoid actors. The second, skeletal models used to acquire motion are not identical. The number of joints or their names differ from model to model. We use the following algorithm:

Let $L$ be the list of leaf joints and $I$ be the list of independent kinematic chains. First set $I$ to empty and populate $L$ with leaf joints. Next,

1. Repeat steps 2-5 for each joint $j$ in $L$
2. Create a new joint list $l$ and insert $j$ into $l$

3. Repeat step 4 till $(\|j.children\| > 1)||(j == root)$

4. Insert $j.parent$ into $l$
   If $(\|j.parent.children\| > 1)$ && $(j.parent \neq root)$ insert $j.parent$ into $L$
   Set $j = j.parent$

5. Insert $l$ into $L$

$L$ contains the list of detected chains. Our algorithm detects the following six kinematics chains for the hierarchy in Figure 5.1.

Chain 1: Thorax ... Head

Chain 2: Thorax ... LWrist

Chain 3: Thorax ... RWrist

Chain 4: Root ... Thorax

Chain 5: Root ... LToes

Chain 6: Root ... RToes

Note that our definition allows number of independent kinematic chains to be more than the number of leaf nodes. Chain 4, for example, does not end in a leaf node. It is an intermediate chain. Motion of Chain 4 affects chains 1 and 3 equally. We treat all kinematic chains uniformly.

Once independent kinematic chains are identified, we split motion by filtering out signals corresponding to DOF variables of joints belonging to each chain. During synthesis we selectively re-composite signals of different chains on to corresponding chains of the base clips. For example in order to create a fast walk wave motion, we composite the DOF signals of joints corresponding to the waving hand on to the corresponding hands DOF signal of the fast walk wave segment.

We also use the independent kinematic chains to create sub-hierarchy cluster graphs. Sub-hierarchy cluster graphs are cluster graphs created for a subset of the articulated objects skeletal hierarchy. We use sub-hierarchy cluster graphs to establish cross body correlations as described in section 5.3.

## 5.3 Correlating motion

Successful graft synthesis needs to account for cross body correlation amongst the various independent kinematic chains. For each graft, we refer to the clip from which the limb motion is copied as the *graft source* and the clip on to which this motion is copied to as the *base clip*. Our objective is to correlate graft source and base clips for the duration of the graft action. Our correlation technique is based on the observation that for locomotive motion:

1. There exists a weak natural correlation between the upper body and lower body motion of humanoid actors. This correlation is weak because the behaviour is voluntary and can be broken at will.

2. There is strong correlation between the motion of the two legs.

We limit our input to locomotive motions - walk, run, stand and jump, that satisfy condition 2. We subdivide the task of establishing correlation into two steps:

> Step 1: Correlate lower body motion across clips.
>
> Step 2: Use the lower body correlation as a guide to synchronize upper body motion grafts for the corresponding lower body motion.

### 5.3.1 Foot plant based lower body correlation

It is hard to model motion correlation precisely as the notion of correlation in motions is amorphous. This is primarily so because humanoid characters are active entities and "break" correlation at will. Strict signal matching techniques such as DTW, therefore, succeed in correlating only a small subset of motion signals. We observe that for locomotive signals, upper body and lower body motions are inherently correlated by foot plants. We use this information for correlating motions for grafting.

Our "cluster graph," described in Chapter 4, automatically clusters together frames from different clips based on similarity. The clustered clip-frame sequences establish temporal correlation between their containing clips. We use this property of cluster graphs along with foot plant signals to correlate graft motions. To correlate lower body motion, we build sub-hierarchy cluster graphs for independent kinematic chains 5 and 6 which correspond to the left and the right legs.

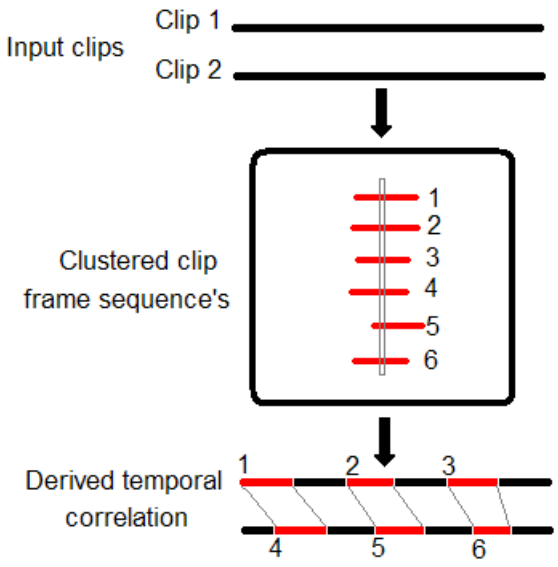



Figure 5.4: Deriving correlation from cluster graphs.

We set our error threshold to a large value such that the frame clustering reflects foot plant constraint states. Similar results can also be obtained by directly comparing foot plant annotations. Using cluster graphs allows us to control the "similarity error" between clustered frames, while direct annotation comparison does not. This control allows us to generating better quality grafts when the feet movement show larger variations for example in grafts between a small stride walk and an exaggerated stride walk.

Figure 5.4 shows clip-frame sequences labelled 1, 2, 3, 4, 5 and 6 clustered together in a cluster graph node. Sequences 1, 2 and 3 belong to input clip 1 while 4, 5 and 6 belong to input clip 2. The bottom sub-figure shows the temporal correlation between sequences (1 & 4), (2 & 5) and (3 & 6). This correlation is obtained by sorting the clip-frame sequences belonging to each clip by time and mapping each sequence of the clips in the sorted order. A linear mapping is established between frames of each clip frame sequence. This mapping provides the largest overlap between the two clips. However since every clip frame sequence within a cluster node is similar alternate correlations also exist. For example the clip frame sequences can be correlated as shown in Figure 5.5. Here the top sub-figure shows our original correlation, middle sub-figure shows correlation between sequences (1 & 5) and (2 & 6) and the bottom figure shows correlation between sequences (2 & 4) and (3 & 6). Alternate correlations are used for synchronizing upper body motion.

For motion clips with similar lower body motion, such as variations of walk, the cluster graph

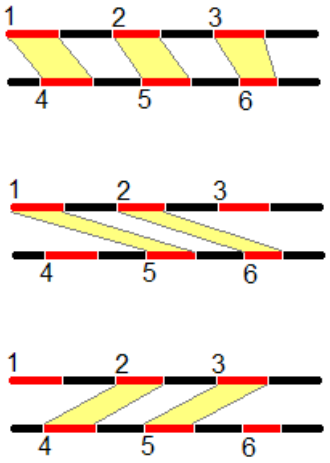



Figure 5.5: Alternate correlations.

correlates every single frame of the two clips. In general, however, all frames are not correlated automatically by the cluster graph. For example for frames from dissimilar motion clips such as walk and run, not all frames map to the same set of cluster nodes. There will be some nodes that contain frames from either only the walk clip or the run clip and vice versa. We deduce correlation for such frames using foot plant information, see Figure 5.6. Using the cluster graph we identify successive foot plant clip frame sequences in each clip. The frames lying in between these sequences are then correlated to with each other by establishing a linear mapping. Note that some frames, typically at the start of the clip till the first foot plant frame and from the last foot plant frame to the end of the clip, cannot be used for grafts as correlation these frames cannot be determined by our method.

### 5.3.2 Upper body synchronization

To synchronize upper body motion we proceed as follows.

1. Find lower body frames corresponding to the upper body transition points of the graft source and the base clip.

2. Locate the nearest cluster containing identical foot plant states.

3. If the graft source and base clip spans containing the transition points are not correlated, choose an alternative lower body correlation such that transition point spans are aligned or are the least number of frames apart from each other, see Figure 5.7 and Figure 5.8.

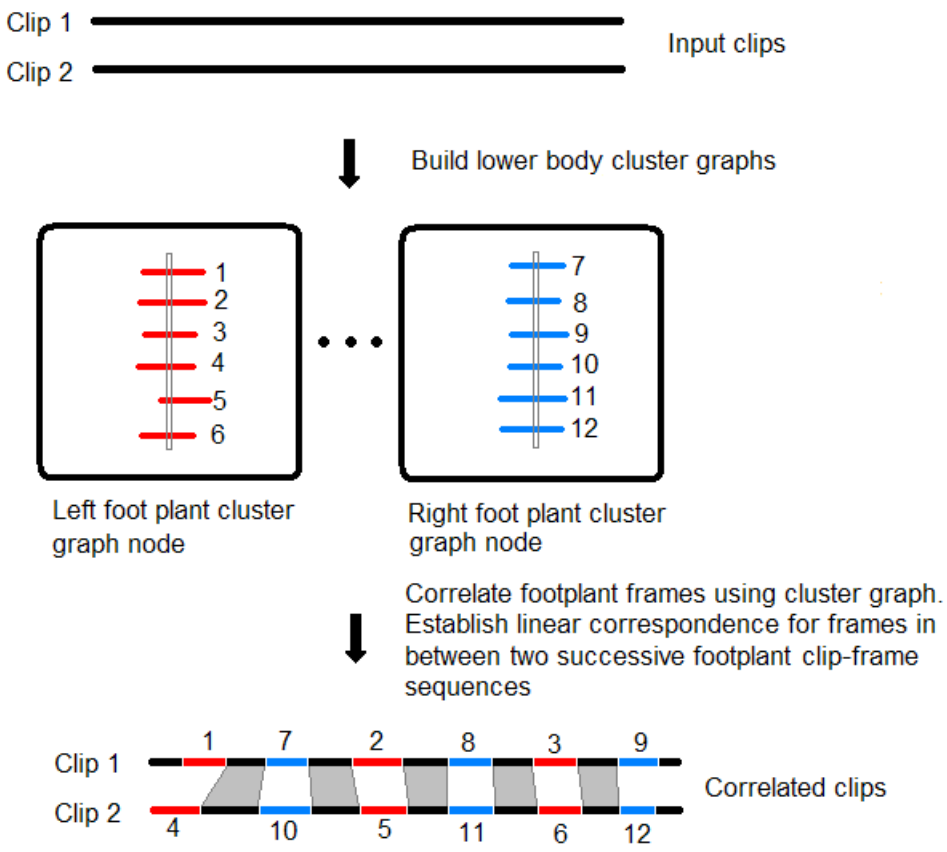



Figure 5.6: Using foot plants to establish correlation.

4. Establish linear mapping between every frame in the two clips for the entire graft action range.

There are two cases to consider - Figure 5.7 show the first case where the transition points are in phase with lower body locomotion cycle. Figure 5.8 show the second case where the transition points are out of phase with lower body locomotion cycle. For the second case, the correlation clip-frame sequences do not overlap but the frame distance between the transition frames is minimized.

## 5.4 Motion Grafting

Grafting is the process of synthesizing motion for an independent kinematic chain. We copy the graft motion from independent kinematic chains of the graft source clip and superimpose it on the corresponding chains of the base clip. Each independent kinematic chain is a list of joints.

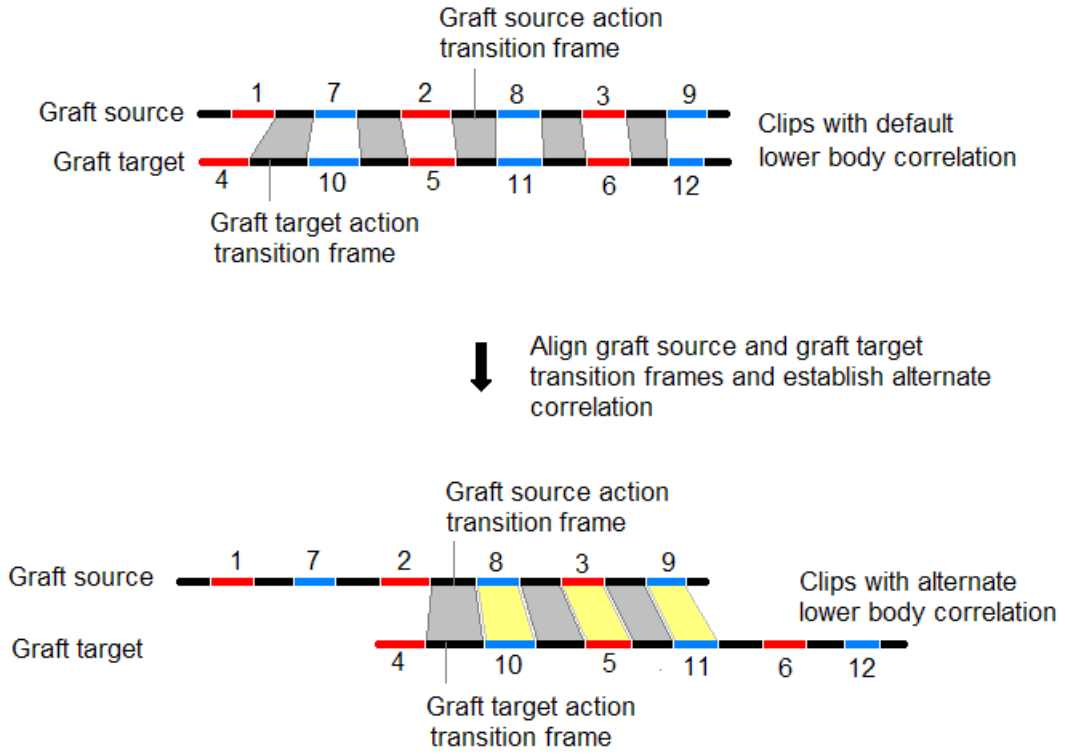



Figure 5.7: Establishing upper body correlation - Case 1 - Transition points are in phase with locomotion cycle and their containing clip-frame sequence spans are aligned.

Grafting essentially involves masking out the base clip's signal and replacing them with those from the graft source. The signals can also be blended together as in multi-target interpolation. In our implementation, we use blending for the transition interval.

A first step to creating grafts is to identify graft action in the graft source corresponding to the action to be replaced in the base clip. One way to accomplish is by manual scanning. However this approach becomes cumbersome for large motion databases. We use an automatic graft identification scheme as described below.

### 5.4.1 Identifying grafts

Our graft framework for automatically identifying and synthesizing graft motion is illustrated in Figure 5.9. We use a conservative, heuristic driven algorithm to increase quality of grafts. The input to our algorithm is a skeletal hierarchy and a free form unlabelled collection of associated

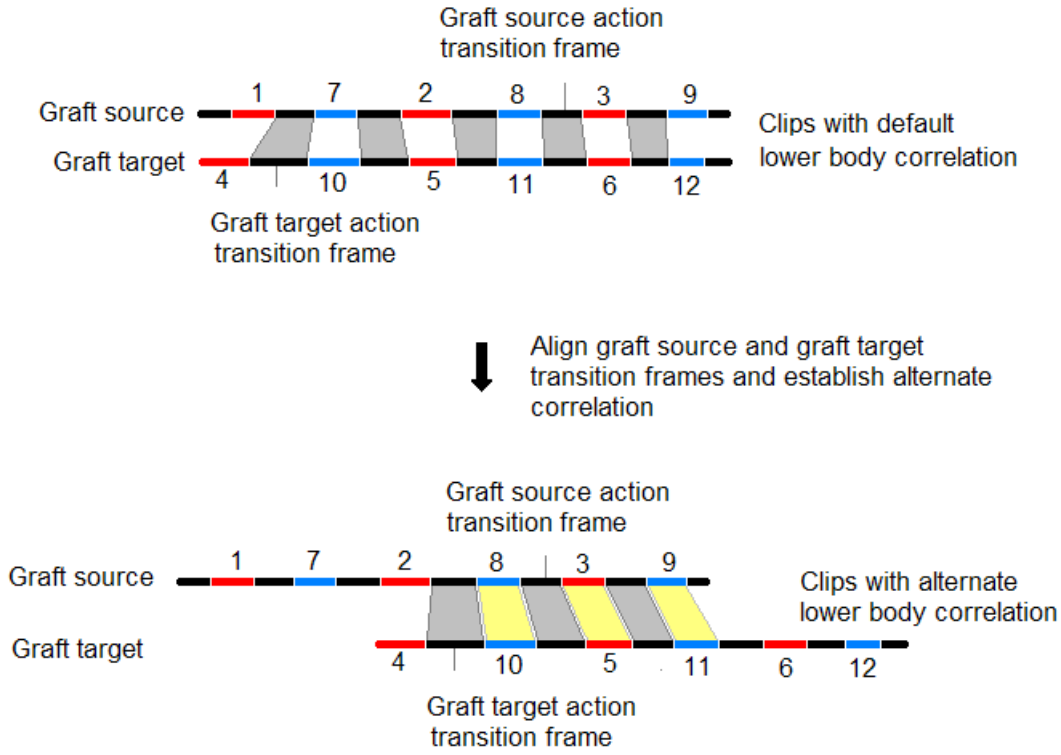



Figure 5.8: Establishing upper body correlation - Case 2 - Transition points are out of phase with locomotion cycle, their containing clip frame sequence spans are not aligned. The correlation is established such that the distance between the transition points is minimal, while maintaining lower body foot plant phase correlation.

motion clips. The output of our algorithm is all possible combinations of synthesized graft action as defined below.

1. The first step is to pre-process every clip in the mocap database to detect and annotate foot plant constraints.

2. Next we classify the clips based on the foot plant annotation into the following categories - “stand,” “walk,” “run,” “jump” and “others.” The clips of interest to us are the ones labelled “stand,” “walk,” “run” or “jump.” We discard the clips labelled “other.”

3. We then separate the upper body and lower body motion signals based on their respective independent kinematic chains while retaining their correspondence for later use.

4. Create a cluster graph for lower body motion. We use this to obtain lower body correlation. Lower body motions determine the base clips used for grafting.

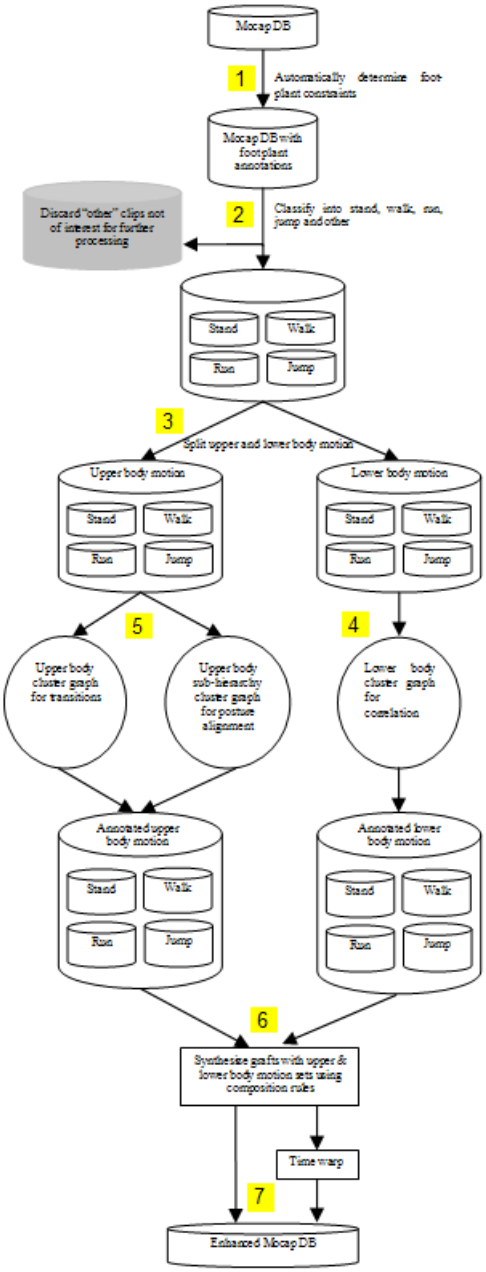



Figure 5.9: Grafting Framework. Our scheme starts with a discovery of locomotive motions from the motion capture database. After classifying independent kinematic chains, a cluster graph data structure is used to correlate seemingly different motions. Correlation is key to generate believable grafts. An optional time warp enables scaling in time.

5. We create two cluster graphs for the upper body motion sets. The first is created with the entire upper body and is used for grafting transition information. However, a subset of transitions may violate self penetration. A second cluster graph is created with just the root lower back upper back torso joint chain, to conservatively estimate safe grafts.

6. We synthesize grafts as a Cartesian product of upper body and lower body motion sets using correlation rules defined in section 5.3 and the composition rules explained in section 5.4.2.

7. As an optional last step we allow the animator to accept or reject generated clips before enhancing the motion database.

### 5.4.2 Composition Rules

Our categorization of motion results in upper and lower body motion divided into four sets corresponding to "stand," "walk," "run" and "jump." We synthesize motion grafts by taking certain conservative Cartesian products of upper and lower body motion sets:

1. (Upper body motion set "stand") X (Lower body action sets "stand," "walk," "run" and "jump").

2. (Upper body motion set "walk") X (Lower body motion sets "walk" and "run")

3. (Upper body motion set "run") X (Lower body motion sets "walk" and "run")

4. (Upper body motion set "jump") X (Lower body motion set "jump")

These rules allow composition of action within same type of motion, for example walk with walk and run with run. Other combinations are allowed based on existence of foot plant correlation. For example, walks exhibit correlation but jumps do not and grafting amongst these motions is disallowed. However both jumps and stands both do not exhibit foot plant correlation and are potential graft candidates. But we disallow this combination because jump are more unstable than stand and the grafts generated are not very believable. We allow actions in runs and walks to be composed with each other as both exhibit foot plant correlation.

## 5.5 Clip classification

Section 3.2 describes our clip classification technique in detail. We use the lower body kinematics chains for clip classification. One of the important characteristics of lower body motion is the foot-plant constraint. We use foot-plant constraint pattern matching to classify clips. We make the following observation regarding foot-plant constraints.

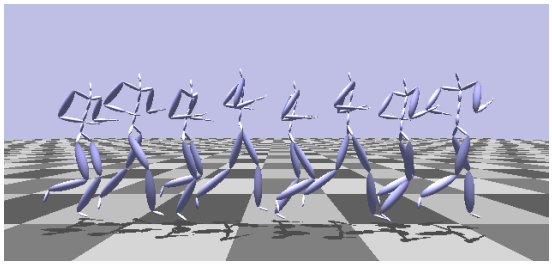
1. Standing Stationary: Both feet remain planted.
2. Walking: Alternate feet are planted passing through a double step pose.
3. Running: Alternate feet are planted with intervening stages of both feet being of the ground.
4. Jumping: Both feet are either planted or in air simultaneously.

### 5.5.1 Detecting foot plant constraints

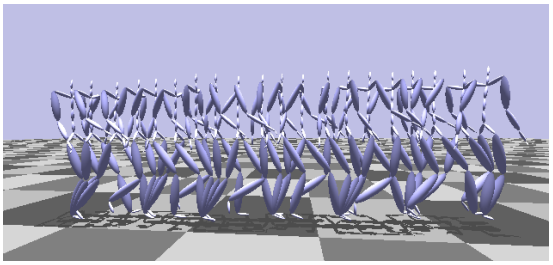
Section 3.1 describes our foot plant detection algorithm in detail. We identify foot plant constraints by first identifying frames with zero crossings for vertical displacement (the y axis in our case). We select all frames which are close to the ground, within a given threshold. This forms the seed set of foot plant frames. For most normal walk sequences, we observe that the foot is placed on the ground for more than one frame. However, in a motion captured sequence, the foot positions may not coincide exactly due to foot skate. [57] describe a technique to identify and correct foot skate. We use a simpler method. From the initial set of foot plant frames obtained above, we sequentially search in both directions and cluster frames, near the seed foot plant frame, where the magnitude of the displacement vector is below a given threshold value. We stop the search at the first frame that fails the test. We then cluster together, like foot plant frames based on their sequence in the clip.

## 5.6 Results

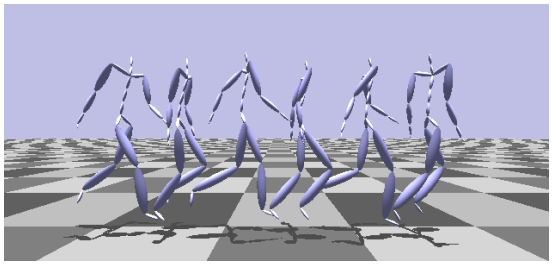
Our motion capture database, after categorization, consists of more than a 100 clips from the CMU motion capture database. Figure 5.10 and Figure 5.11 are representative of the results



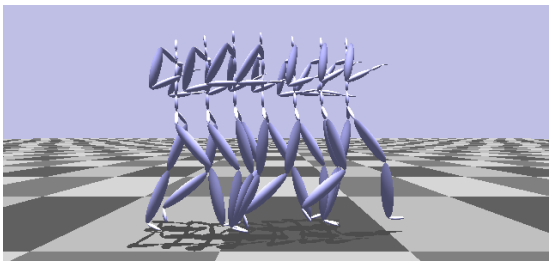
(a) A run animation.



(b) A walk animation.

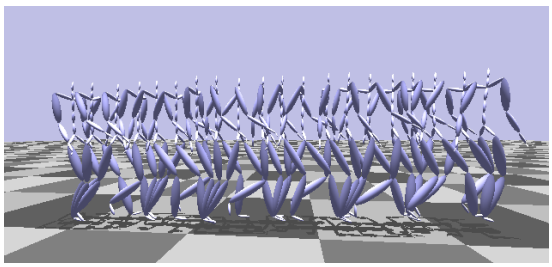


(c) Graft with walk hand movements transplanted on to run animation.

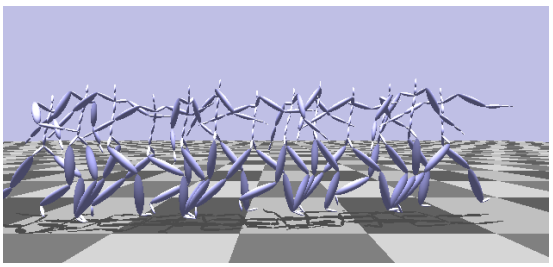


(d) Graft with run hand movements transplanted on to walk animation.

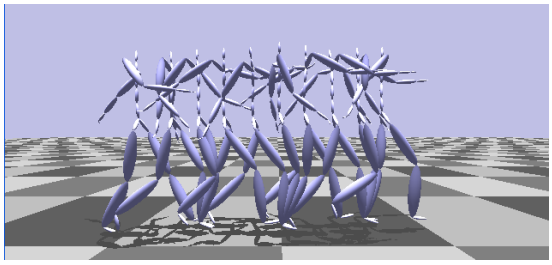
Figure 5.10: Example motion grafts with a walk clip and a run clip.



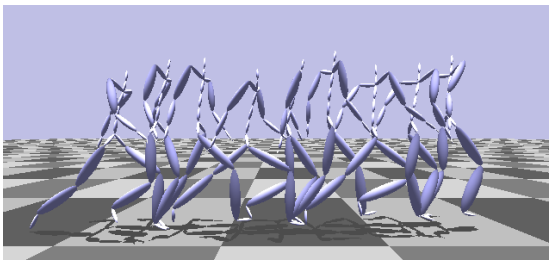
(a) A walk animation.



(b) A walk animation with exaggerated stride.



(c) Graft with exaggerated stride walk hand movements transplanted on to walk animation to produce a march like motion.



(d) Graft with walk hand movements transplanted on to exaggerated stride walk animation.

Figure 5.11: Example motion grafts with a walk clip and a walk clip with exaggerated stride.

obtained using our method. In Figure 5.10 the walk clip (top right) and the run clip (top left) are used as both the target and base clips to synthesize two new clips (bottom row). Bottom left shows the result of grafting walk hand movements on the run. Bottom right image shows the results of grafting run hand movements on the walk clip. In Figure 5.11 The normal walk clip (top left) and the exaggerate stride walk (top right)are used as both the graft and the base clip. The bottom left picture depicts arm motions from the exaggerated stride grafted on to simple walk yielding the marching like motion. Bottom right picture shows the animation resulting form grafting normal walk hand movements onto the exaggerated stride walk. As can be seen (from the accompanying videos) the results are fairly believable and smooth.

## 5.7 Comparison with concurrent work

To our knowledge motion grafting was first discussed in our own prior unpublished work [97]. An interesting implementation has been subsequently described in [47]. The work presented here complements the work in [47] in the following ways:

- For increased quality, we target only motions that have running, jumping, and walking motions. Several unsuccessful transplants are reported in [47].
- The randomization rules to generate new motion are not used in our work. Instead currently we have used a Cartesian product of upper and lower body classification to generate candidate grafts.
- Instead of using an SVM based classification to determine successful transplants, we use the intrinsic correlation available in cluster graphs.

# Chapter 6

# Walk Parameterization

In this chapter we describe our novel parameterization and synthesis of new walk and climb motions from a single motion captured sequence. Our aim is to enable adapting mocap walk motion to different scenarios. This is an essential requirement to enable mocap reuse. We parameterize walk on stride and foot lift parameters. Our use of a single motion clip complements the large database approach. Unlike most mocap based schemes our walk synthesis can be controlled programmatically. We use our technique as a post processing filter for our higher level mocap reuse methods described earlier.

## 6.1 Motivation

Walk is a fundamental humanoid motion. It is the preferred mode of locomotion. Humans walk more than they run, jump, hop or swim. Many animations feature walking. Adapting mocap walks to different scenarios may require editing for reasons such as meeting positional goals, bounding over small obstacles in the path and adapting to inclines.

1. Meeting positional goals: Figure 6.1 shows an example of walk adaptation. Suppose the original recording captures straight line walk of an actor walking from point A to point B. In a different scene, say, the actor needs to walk from point C to point D, again, in a straight line. Let $d(A,B)$ and $d(C,D)$ be distances from A to B and C to D respectively. If we choose to reuse the captured walk, then it would need adaptation if $d(A,B) \neq d(C,D)$. Let $s$ be the actors stride length in the original capture. Walk is a symmetric cyclic motion.

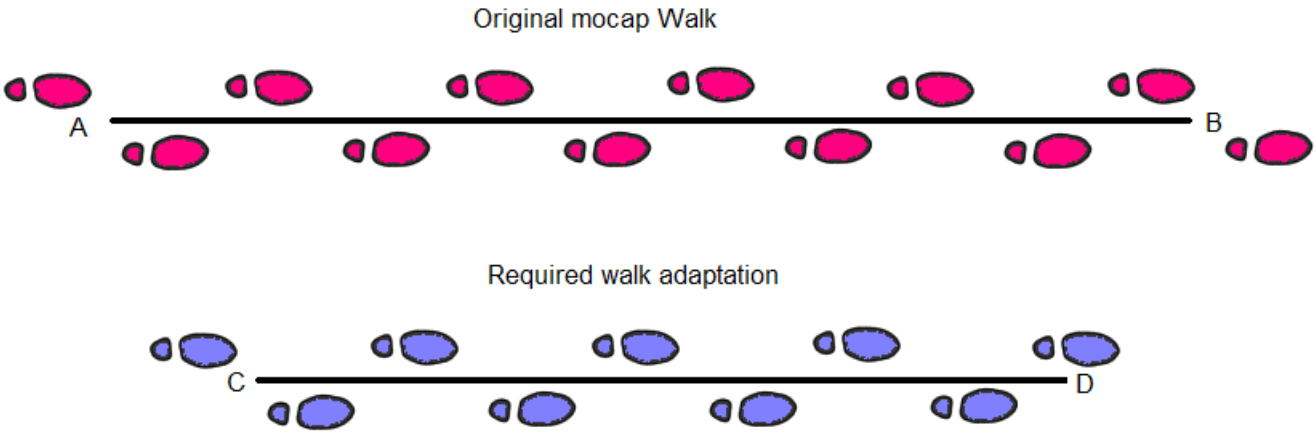
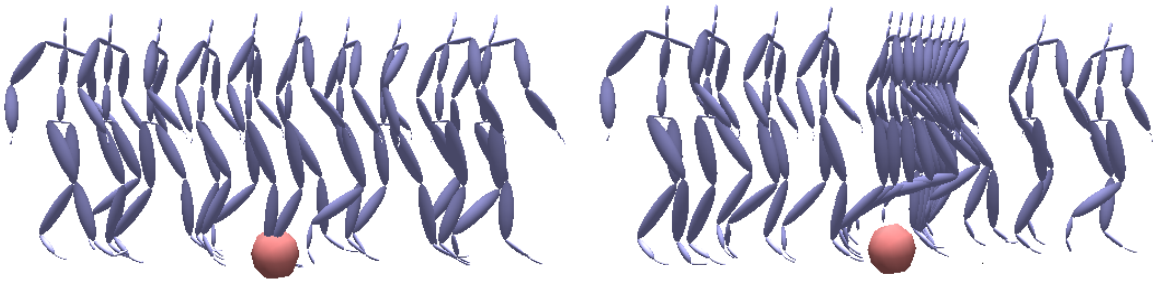



Figure 6.1: Adapting walk to meet positional constraints

Assuming that the actors stride length is uniform for all strides of the recording and if $d(A,C)$ is equal to an integral number of strides, then we may adapt by simply chopping or looping the walk by an integral number by an integral number of stride. However, if this is not the case, then the stride of the walk will need to be modified to coincide with $d(A,C)$. The modification required could be uniform or gradual, i.e. increasing or decreasing slowly from its recorded value. Similar adaptation can also be applied to ease the character into a walk from a stationary position or to transition him from walk to stand.

2. Bounding obstacles:



(a) With no adaptation, the actor walks through the obstacle.

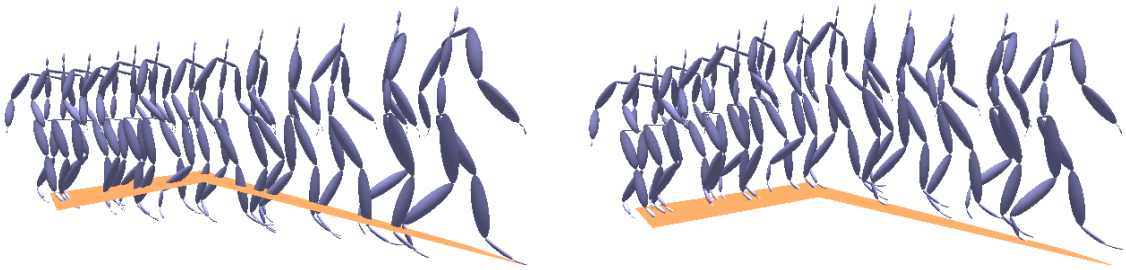
(b) The walk is adapted such that the actor bounds over the obstacle.

Figure 6.2: Adapting walk to bound over obstacles

Walking in a virtual environment full of objects sometimes calls for bounding over obstacles. For example a game character navigating through a forest scene may be required to

bound over a small log of wood. Figure 6.2 shows an example. In this case the foot lift and stride of the walk have been altered locally to step over the obstacle.

3. Adapting to inclines: Walks are commonly recorded for level horizontal planes. Yet vir-



(a) The original mocap walk on horizontal surface, needs to be adapted for use on inclines.

(b) Mocap walk displaced vertically to confirm to the ground plane exhibits visual artifacts.

Figure 6.3: Adapting walks for inclined planes.

tual environments often contain inclined planes within them. Using the walk captured for level plane for an inclined plane creates visual artifacts. The obvious method of changing the characters vertical position makes the character glide along the incline unnaturally. The footsteps penetrate into the incline or appear to be floating above the incline depending upon the vertical displacement applied. A requirement therefore, is to adapt a captured walk to an incline.

Here we describe our novel motion parameterization and synthesis of new walk and climb motions from a single motion captured sequence. Specifically, we describe a new *per frame inverse kinematics* (PFIK) [36] based method that synthesizes variable stride and variable lift walk and climb limb motion from a single motion captured walk sequence using a kinematic walk model. We use *stride* and *lift* as the control parameters.

## 6.2 Our method

Our base motion sequences are motion captured walks, such as shown in Figure 6.4. The aim is to programmatically synthesize variations using stride and lift as control parameters.

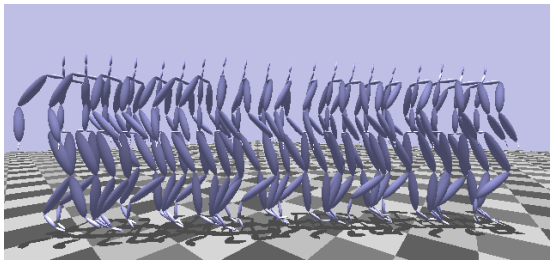


Figure 6.4: Motion captured base walk sequence.

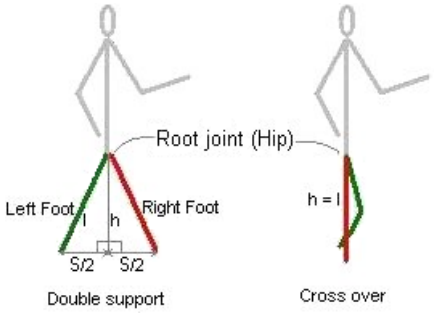



Figure 6.5: Simple Kinematic Walk Model.

### 6.2.1 Kinematic Model for Walk

Human walking is a process of locomotion in which the erect, moving body is supported first by one leg and then the other. As the moving body passes over the supporting leg, the other leg is swinging forward in preparation for its next support phase. One foot or the other is always on the ground, and during that period when the support of the body is transferred from the trailing to the leading leg there is a brief period termed as "*double support*." The stance at which the hip attains maximum ground clearance is termed as "*crossover stance*." [48] describes human walking in more detail.

Based on the above, we define a simple kinematic model for human walking that allows us to estimate hip ground clearance $h$, during the gait cycle. Figure 6.5 shows ground clearance values for the root joint. The minimum occurs during double support stance. The maximum occurs during the cross over stance. From figure 6.5 we have

$$h = \begin{cases} \sqrt{l^2 - (\frac{S}{2})^2} & \text{for double step stance} \\ l & \text{for crossover stance} \end{cases} \tag{6.1}$$

where $S$ is the stride and $l$ is the length of the legs measured from hip to heel at the crossover stance.

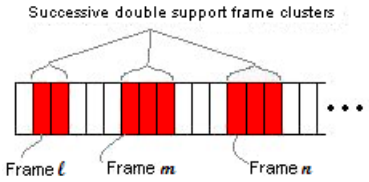



Figure 6.6: Frame $m$ and Frame $n$.

The root joint trajectory between two double support stances for the saggital plane (Y-Z plane in our case) is a sinusoidal wave [48]. We pre-process the clip to annotate double support frames and cluster contiguous sequence of double support frames as shown in Figure 6.6. If $R_i$ is the position of the root joint for the $i^{th}$ frame, $m$ is the first frame of a double support cluster and $n$ is the first frame of the immediately succeeding double step cluster, then the y position of the root joint in the interval $[m, n)$ is given by:

$$R_i.y = h_{min} + \delta h * \sin(\theta) \tag{6.2}$$

$$\text{where} \begin{cases} \theta = \frac{\frac{\pi}{2} * (i-m)}{n-m}, m \leq i < n \\ h_{min} = \sqrt{l^2 - (\frac{S}{2})^2} \\ \delta h = l - h_{min} \end{cases}$$

Given a skeletal model with leg length $l$ and a walk sequence with new desired stride $S'$, we can recompute the root's saggital plane trajectory for the interval $[m,n)$ using the above equation. The root's transverse plane (Z-X plane in our case) trajectory is retained from the original motion clip. Figure 6.7 shows the trajectories for the root joint and the left foot joint, for a half walk cycle[1]. In the second half cycle, the root joint trajectory repeats itself and the right foot follows a trajectory similar to the one shown for left foot.

### 6.2.2 Preprocessing

We pre-process the base walk sequence to identify and annotate foot plant and double support frames. Our foot plant identification technique is described in Section 3.1. We then compute per frame relative displacement vectors of root and feet joints. We also compute per frame foot lift vectors and mean foot lift vector.

[1]The full walk cycle consists of two symmetric strides, one with each foot leading.

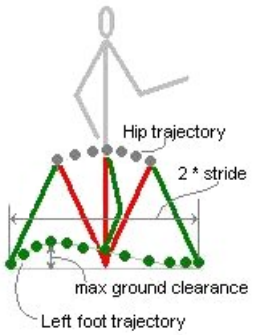



Figure 6.7: Root and foot trajectories in the saggital plane for the left foot. This constitutes one half cycle of walk. The second half cycle with the right foot leading and the left planted is symmetric and follows similar trajectory.

**Computing Relative Displacement Vector and Stride**

We compute per frame relative displacement vectors for the *root node*, *left foot node* and *right foot node*. The displacement vector is computed as follows:

Let $P_{j_i}$ and $P_{j_{(i+1)}}$ be the world positions of joint $j$ at frame $i$ and $i+1$ respectively. Then the relative displacement vector $D_{j_i}$ for joint $j$ at frame $i$ is given by:

$$D_{j_i} \leftarrow P_{j_{(i+1)}} - P_{j_i} \tag{6.3}$$

The foot stride vector $S$ is computed by adding individual frame displacement vectors of frames for either left or right foot joint from frame $l$ to frame $n$, where $l$ and $n$ are as explained in Figure 6.6. The magnitude of $S$, so computed, is twice the stride. The stride for left and right feet in a rhythmic walk motion are equal.

$$S = \frac{\sum_{i=l}^{n} D_{j_i}}{2} \tag{6.4}$$

**Computing Lift Vector**

Computing the lift vector for the base sequence, involves projecting the feet position on the ground plane, and computing the vector difference of the two positions.Let $P_{j_i}$ and $P'_{j_i}$ be the world positions of joint $j$, representing a foot, at frame $i$ and its projection on the ground plane

respectively. Then the lift vector $L_{j_i}$for joint $j$ at frame $i$ is given by:

$$L_{j_i} \leftarrow P_{j_i} - P'_{j_i} \tag{6.5}$$

Traversing through the frames sequentially, we identify frames with local maxima of the foot lift vector. We find the mean magnitude, $L_m$, of magnitudes of local maxima of foot lift vectors. This is used to determine the foot lift scaling factor as explained in Section 6.2.3.

### 6.2.3 Synthesis

In this section we describe our synthesis of variable stride, variable foot lift and climb. We synthesize lower body motion and retain the upper body animation of the original recording. If desired, the method in [94] can be applied as a post process to physically touch up upper body motion. Since the stride changes required in practice are small, our synthesis remains believable even without this step. This is best seen in our sample results video; a glimpse of which is shown in Figure 6.11. We describe our adaptations below.

#### Varying stride

The stride of the captured walk sequence is varied by directly scaling the relative displacement vectors $D_{j_i}$, with a scaling factor $s$. If the new stride length desired is $S'$ then the scale factors is given by:

$$s = \frac{S'}{|S|} \tag{6.6}$$

The displacement vectors of the root joint are also scaled by the same scaling factor. A new trajectory is obtained for the root joint and foot joints as follows:

for $i = l$ to $(n-1)$

$P_{j_{(i+1)}} \leftarrow P_{j_i} + s * D_{j_i}$

After this step, the y coordinates of root joint at each frame $i$ are adjusted using equation 6.2. Smoothly varying the scale factor $s$ creates a smoothly varying stride. This is used for synthesizing an accelerating or decelerating walk sequence. The DOF angles for each joint in the foot

kinematic chain are recomputed at each frame using a two link analytical *inverse kinematics solver*. Our IK solver is described in Section 6.3.

### Varying Foot Lift

We vary foot lift by computing new positions for the feet joints by scaling the frame foot lift vectors by a scale factor $f$. If the new max foot lift desired is $F$, then the scale factor is given by:

$$f = \frac{F}{L_m} \tag{6.7}$$

where $L_m$ is the mean magnitude of local maxima of foot lift vectors as described in Section 6.2.2. For a motion sequence containing $n$ frames, the new foot trajectory is computed as:

for $i = 0$ to $(n-1)$

$P_{j_i} \leftarrow P'_{j_i} + f * L_{j_i}$

where $P'_{j_i}$ is the projection of joint position $P_{j_i}$ on the ground plane. The computation for foot lift and stride are combined as follows:

for $i = l$ to $(n-1)$

$P_{j_{(i+1)}} \leftarrow P_{j_i} + s * D_{j_i}$

$P_{j_{(i+1)}} \leftarrow P'_{j_{(i+1)}} + f * L_{j_{(i+1)}}$

The DOF angles for each joint in the foot kinematic chain are recomputed for each frame using our IK solver.

### Synthesizing climb

Here we describe the synthesis of climb motion, along a plane inclined at angle $\theta$ with the horizontal as illustrated in Figure 6.8. The climb is synthesized as a combination of stride and lift as follows. Let $S$ be the desired stride along the inclined plane. The distance parallel to the ground plane is given by $S^{\parallel}$. The corresponding rise in height in ground level is given by $S^{\perp}$.

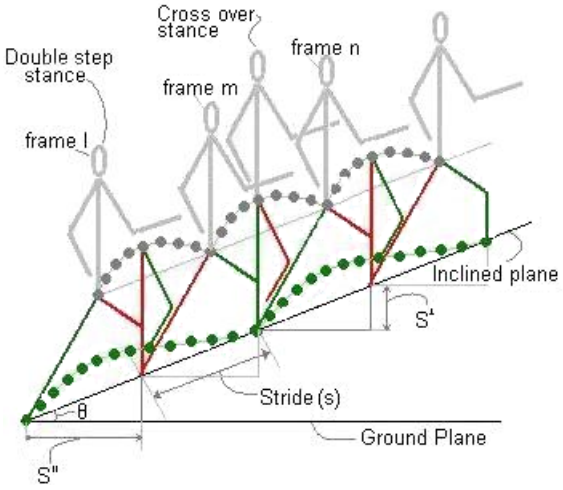



Figure 6.8: Modelling climb.

$$
\begin{aligned}
|S^{\parallel}| &= |S| * \cos(\theta) \\
|S^{\perp}| &= |S| * \sin(\theta)
\end{aligned}
\tag{6.8}
$$

For synthesizing the climb motion, we need to estimate the path of the root joint and the feet joints of our articulated body. For this we use the simple kinematic model described in section 6.2.1 as before. However we displace the root positions to account for the incline.

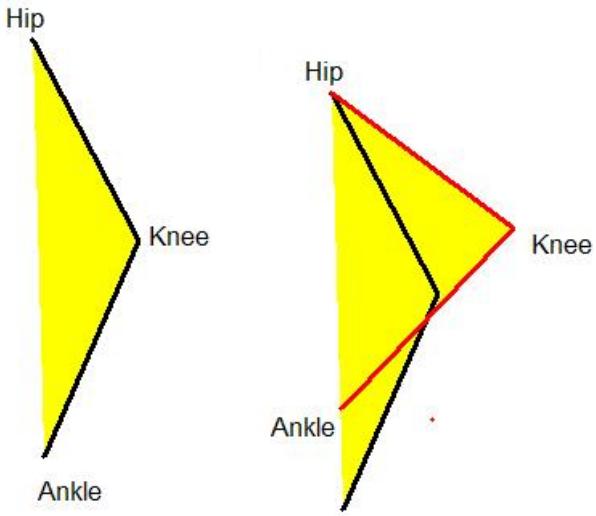
Consider successive double step frames as in frames $l$, $m$, $n$ shown in Figure 6.6. The geometry of these frames for climb is as visualized in Figure 6.8. Note that the root undergoes an additional vertical displacement of magnitude $S^{\perp}$ between each successive double step frame. We first synthesize a sequence with stride scaled to $S^{\parallel}$. We displace the root positions in this modified sequence. We incrementally displace the root in the vertical direction by $S^{\perp}$ at each double support frame cluster. The vertical displacement for in between frames is linearly interpolated as follows:

for $i = l$ to $m$

$P_{j_i}.y \leftarrow P_{j_i}.y + \frac{(i-l)*S^{\perp}}{(m-l)}$

where $P_{j_i}$ is the position of the root joint in the $i^{th}$ frame of the modified sequence.

Trajectory for the left and right foot joints are calculated in a similar manner. However, note that the left foot (shown in green color in Figure 6.8), accumulates an additional vertical displacement of magnitude $2 * S^{\perp}$ between frames $l$ and $m$. For frames $m$ to $n$, the right leg, shown in red, accumulates this vertical displacement. The legs climb alternately, with the planted leg




(a) Constraint plane defined by original Hip, Knee and Ankle joint positions.

(b) Configuration for the modified knee position.

Figure 6.9: The black line segments show the original configuration of the hip-knee-ankle joints and the red segments show the new configuration. The original hip-knee-ankle configuration defines a constraint plane. We find an IK solution for the system given a new ankle position on this plane. Our solution is constrained such that all joints continue to lie in the constraint plane.

maintaining its height. The trajectory of the left leg joint from frames $l$ to $m$ can be computed as follows:

for $i = l$ to $m$

$$P_{j_i}.y \leftarrow P_{j_i}.y + \frac{(i-l)*2*S^{\perp}}{(m-l)}$$

where $P_{j_i}$ is the position of the left leg joint in the $i^{th}$ frame of the modified sequence. The DOF angles for each joint in the foot kinematic chain are recomputed, for each frame, using our IK solver.

## 6.3 Inverse Kinematics Solver

In this section we describe our analytical inverse kinematics algorithm based on planar two link solver.

Let $H$ be the hip joint, $K$ be the knee joint and $A$ be the ankle joint. The configuration of the hip-

knee-ankle system is as depicted in figure 6.9. Our motion parameterization procedure modifies the root and the ankle positions. We need to compute the modified knee position and the new DOF angle values at the hip joint and the knee joint.

The modified root position is used to position the skeleton. We compute the "original" hip, knee and ankle world positions using the existing DOF values. These three joint positions define a constraint plane. We now displace the ankle to the modified position determined by our walk synthesis algorithm. Next we compute the new knee position using planar two link IK solver described in section 6.3.1. Once the modified knee position is known, we compute the DOF joint values at hip using the method described in 6.3.2. With the hip joint angles computed, we apply the same method to compute the DOF angles for the knee.

### 6.3.1 Planar two link solver

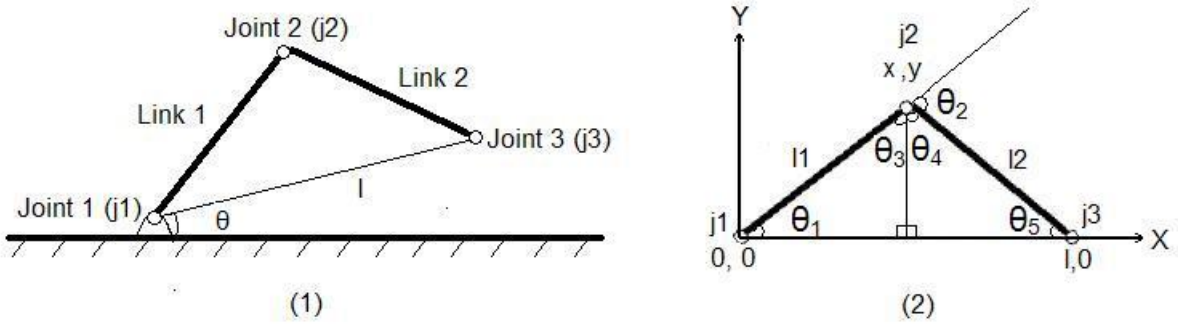



Figure 6.10: Two link mechanism. In (1) end-effector j3 is positioned coordinates (l, $\theta$) with j1 as the origin. (2) shows the setup to compute angles $\theta_1$ and $\theta_2$.

Figure 6.10 depicts a planar two link mechanism consisting of three revolute joints – $j_1$, $j_2$, $j_3$ and two links – link 1 of length $l_1$ and link 2 of length $l_2$. Joint $j_1$ is fixed to the base substrate and joint $j_3$ is free. $j_3$ is the end effector in this section. The position of $j_3$ can be computed if $\theta_1$ and $\theta_2$ are specified. The problem is to find $\theta_1$ and $\theta_2$ given the position of $j_3$ ( $l$, $\theta$).

The setup to solve for joint angles is shown in 6.10(2). Solving using simple coordinate geometry we get,

$$\theta_1 = \tan^{-1}(\frac{y}{x}) \tag{6.9}$$

$$\theta_2 = \tan^{-1}(\frac{y}{x}) + \tan^{-1}(\frac{y}{l-x}) \tag{6.10}$$

For solving the limb configuration we map the hip to joint j1, knee to j2 and ankle to j3.

### 6.3.2 Computing joint DOF angles

In this section we explain our method for computing joint DOF angles given the its world position.

The model used for mocap is organized as a skeletal hierarchy. The world position of a joint $J$ whose parent is joint $P$ is computed as follows:

Let the offset of $J$ in $P$ local coordinate system be $j(x,y,z)$ and the rotational DOF's of $P$ be $p(\theta_x,\theta_y,\theta_z)$. The world position of $J$ $j_w(x,y,z)$ is computed as

$$j_w = j * M * W$$

where $M$ is the composite homogeneous rotation matrix for $p(\theta_x,\theta_y,\theta_z)$ and $W$ is $P$'s world transformation matrix. Replacing $j * M$ by $j_M$ we get

$$j_w = j_M * W$$

Here $j_M$ is a point in $P$'s local coordinate system. Given $j_M$, a homogeneous matrix equivalent to M can be found by composing elementary transformation matrices as described in Appendix D. $j_M$ can be computed from $j_w$ when $W$ is known

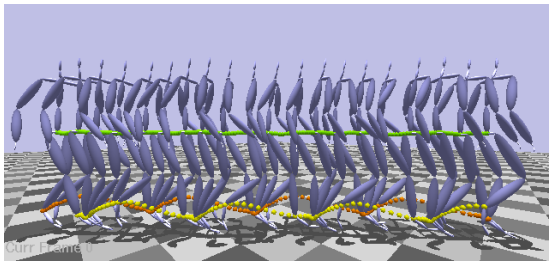
$$j_M = j_w * W^{-1}$$

Given a new world position $j'_w$ for joint $J$, the new local coordinate system position of $J$ is given by

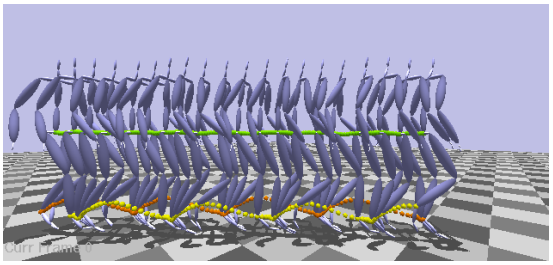
$$j'_M = j'_w * W^{-1} \tag{6.11}$$

Next we find the modified local rotation matrix $M'$ corresponding to $j'_M$, the new local coordinate position of $J$, (see Appendix D).

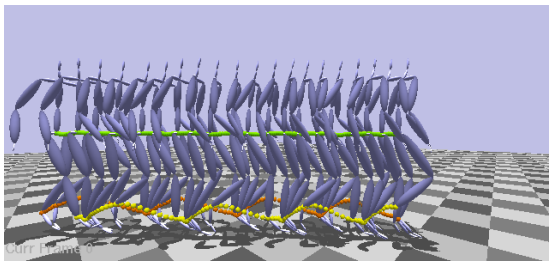
Our articulated figure however uses Euler angle representation. We therefore need to convert $M'$ to Euler angles $p(\theta'_x,\theta'_y,\theta'_z)$. This conversion is performed using standard homogeneous matrix decomposition techniques [41], [96].



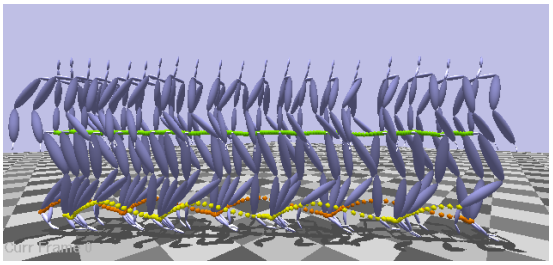
(a) A walk animation.



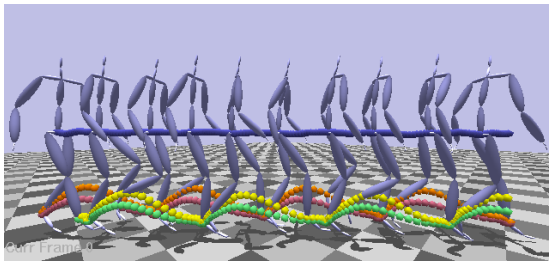
(b) Synthesized walk with uniformly shortened stride.



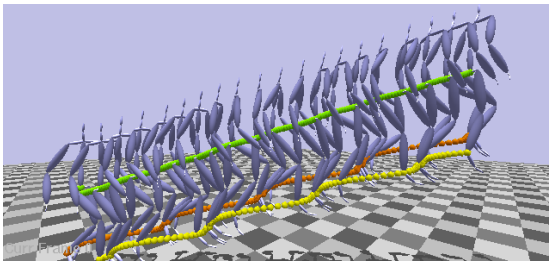
(c) Synthesized walk with gradually reducing stride.



(d) Synthesized walk with gradually increasing stride.



(e) Synthesized walk with high foot lift. The yellow and orange streamers depict the modified foot lift curve whereas the green and pink streamers show the original one.



(f) A synthesized climb.

Figure 6.11: Walk parameterization. (a) is a motion captured walk animation. (b)—(f) have been synthesized using our walk parameterization technique.

## 6.4 Experimental results and Discussion

We have described our method to parameterize a motion clip with an aim to synthesize variations thereof. Figure 6.11 and the accompanying video show results of our synthesis. We use a simple kinematic model adapted from [48] to drive the synthesis. It is now conceivable that several such parameterizations can exists for the different motions that comprise the actors motion database. Our implementation is efficient, fast and can be used online.

# Chapter 7

# Scripting

One of the goals of this work is to simplify mocap reuse. Traditional motion editing techniques operate at individual joint DOF level, making them cumbersome to use. Our methods operate at a relatively higher level. We enable animators to synthesize new animation by composing sub-clip and sub-hierarchy motion and allow further customization using walk parameterization. In this chapter we explore the application of these techniques to content creation and presentation. We define a set of primitive operations that enable flexible motion reuse and describe a simple scripting interface to specify, instantiate and control virtual actors driven by a mocap animation database. We demonstrate the use of our scripting engine to synthesize animation in story telling scenarios.

## 7.1 Motivation

We aim to make our reuse methods accessible to animators by providing a simple yet powerful scripting interface. A set of composable primitives is desired for flexibility. Our motion editing operations are best expressed in terms of motion clips and transformations. Every reuse transformation acts on one or more input clips and produces an output motion clip (see Figure 7.1). This suggests implementing motion reuse primitives as filters. Filters are commonly used in image, audio and video processing applications. They provide an efficient, extensible, composable and easy to use framework. Using the filter architecture we can express reuse scenarios as *directed acyclic filter graphs*.

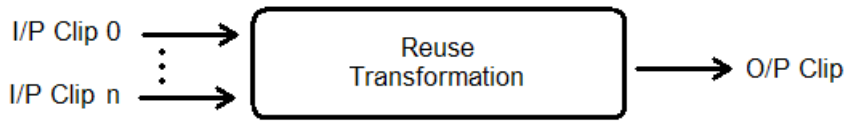



Figure 7.1: The input mocap clip is transformed by the mocap reuse transform to synthesize the output clip.

Figure 7.2 shows an example of a motion reuse filter graph instance. Here an output clip is synthesized by transforming three input clips using a sequence of simple filter transformations. The output of one filter is used as the input to the next filter in the graph. Each intermediate output clip is fed to one or more subsequent filters. The output animation is assembled by concatenating the sequence of transformed clips

The rest of this chapter describes our motion reuse primitives and our scripting language constructs.

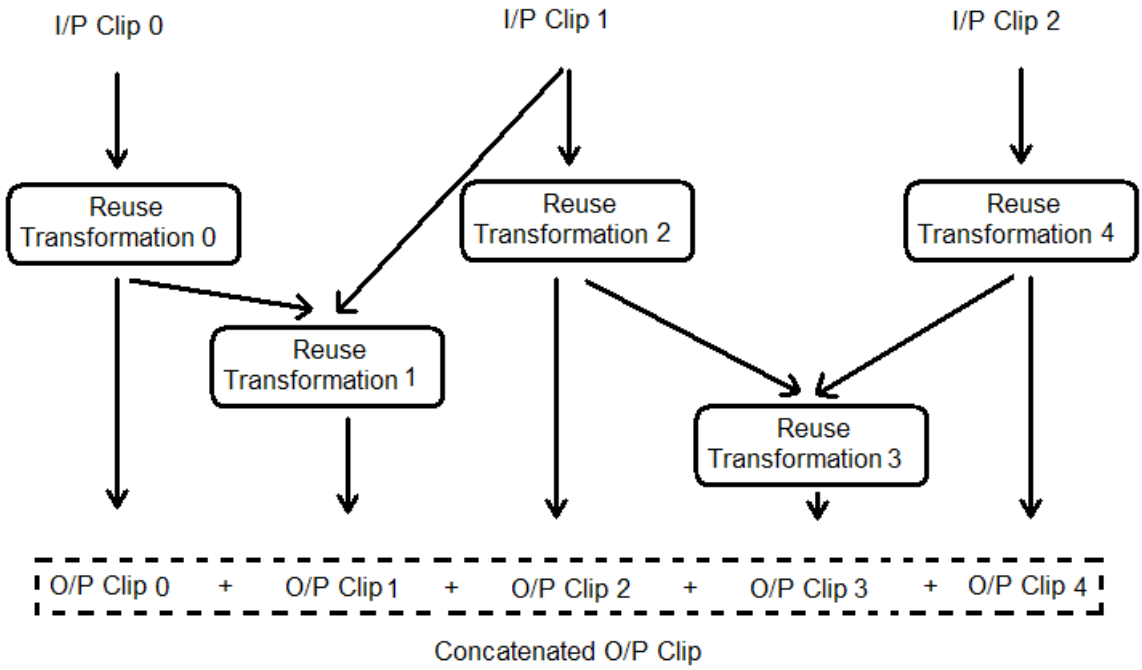



Figure 7.2: A transform filter graph depicting a motion reuse scenario. The output animation is a concatenation of individual output clips.

## 7.2 Motion Reuse Primitives

Motion reuse primitives are the building blocks of our filter graph. Each of the primitives is implemented as a filter. We define the following primitives:

1. Concatenate: This primitive takes one or more input clips and appends them in the input order.

2. Sub-Clip: This primitive takes one input clip and a start frame and an end frame. It returns a sequence of contiguous frames, between the start and the end frame,d from the input clip.

3. Re-sample: This primitive takes one clip as input. It re-samples the frames between the given start and end frame using linear interpolation to produce the desired number of output frames.

4. Loop: This primitives constructs a new clip by looping an input clips frames. It takes as input the parameters start frame, end frame, loop enter and exit frames. The clip plays from the start frame till the loop exit frame. It jumps from the loop exit frame to the loop enter frames "*loop count*" times. It terminates by playing frames up to the end frames.

5. Transition: This primitive creates a transition between the two input clips, at the specified transition points for both clips, using the specified blending window.

6. Graft: This primitive takes two clips as input the graft target and the base clip on to which the motion is to be grafted. In addition it takes as input the graft range for the source clip and the destination clip and the pointer to the graft independent kinematic Chains. The output is a smoothly grafted clip.

7. AlterWalk: This primitive takes as input a walk clip along with the desired stride, foot lift and incline parameters and returns the modified walk.

8. Load: The load filter loads an input clip from the file system. It can parse files in the bvh or asf file format.

9. Save: The save filter persists its input clip to the file system. It takes as input a single clip and its persistence filename.

10. Display: Display filter renders its input clip on to a window on the screen.

## 7.3 Scripting

To facilitate the use of our reuse primitives, we provide a simple scripting environment that allows an animator to script stories. In our environment, scripting the story involves two tasks. The first task is to create actor definitions. The second task is to instantiate actors and define the narrative. The narrative specifies a sequence of actions, for every actor, that enacts the story.

We use the following conventions in our scripting language:
'{' and '}' are used to enclose sub-structure.
'[' and ']' are used to group collections together.
The text inside '≪' and '≫' is a place holder for an actual identifier such as an *alias* or *filename*.
All strings are enclosed in double quotes(").

Figure 7.3 shows the outline of a script. It contains two top level sections – the actors collection and the story.

```
actors
[
        actor <<alias>> {...}
        ...
]
story
{
        cast {...}
        scenes
        [
                scene <<alias>> {...}
                ...
        ]
        screenplay {...}
}
```

Figure 7.3: The script outline

### 7.3.1 Actor Definition

The '`actors`' collection contains definitions of one or more actors that participate in the story. Each actors properties are defined by the '`actor`' structure.

### Actor

Figure 7.4 shows the syntax of an actor definition. The '`actor`' keyword begins the actor definition. This is followed by the name of the actor being defined. The body of the actor definition is enclosed in a pair of curly braces .

```
actor <<alias>>
{
        hierarchy <<filename>>
        geometry <<filename>>
        mocapdb
        [
                clip <<alias>> <<filename>>
                ...
        ]
        graftchains
        [
                graftchain <<alias>>
                {
                noderef <<alias>>
                ...
                }
        ]
        actions
        [
                action <<alias>>
                {
                        <<clip>>
                }
                ...
        ]
}
```

Figure 7.4: Actor definition syntax.

The following attributes are defined for each actor:

- *hierarchy*: This attribute associates the actor definition with the hierarchy that was used to capture the motions. Each actor can be associated with one hierarchy. The hierarchy is specified by referencing an 'asf' or 'bvh' file.

- *geometry*: This attribute associates the actor definition with a skin file, used for rendering the actor on the screen. The skin file is a triangle model comprising of vertices, triangle and association of vertices to joints in the hierarchy.

- *mocapdb*: This attribute is a collection that references the mocap clips that have been captured for this actor. Entries in mocapdb are of the form

```
clip <<alias>> <<filename>>
```

  where *alias* is the name to be associated with the clip *filename*. The reference is implemented using our *load* primitive described in section 7.2

- *graftchains*: This attribute is a collection of named independent kinematic chains, referred to in the script as graft chain and used for motion grafting. Each `graftchain` uses the syntax

```
graftchain <<alias>>
[
        noderef <<alias>>
        ...
]
```

  Here each graftchain is associated with an alias. The graftchain is a collection of node references where the `noderef` alias references the nodes in the `hierarchy`.

- *actions*: This attribute is a collection of action definitions that associate clips with an action name. The syntax of each action entry is described below.

**Action**

Each action associates a name action with a motion clip. The reference can either be to a mocap clip from the mocapdb or one that is derived using a filter graph instance. This struct forms the link to our motion reuse primitives. The syntax of an '`action`' is shown in Figure 7.5. The

```
action <<alias>>
{
        <<clip>>
}
```

Figure 7.5: Action syntax.

definition starts with the keyword '`action`', followed by an alias to be associated with this

action. The body of the action defines the *clip* reference. The clip reference is defined by one of the following motion reuse primitives:

- `clipref`: This primitive references a clip specified in the mocapdb. The alias refers to the in-script name associated to the mocap filename in the mocapdb.

```
clipref <<alias>>
```

- `concatenate`: This primitive references a collection of one or more *clip's* and returns a concatenated clip.

```
concatenate
[
        <<clip 1>>
        ...
        <<clip n>>
]
```

- `subclip` The subclip primitive returns a contiguous sequence of frames between the specified start and end frames from its input clip.

```
subclip
{
        <<clip>>
        startframe <<int>>
        endframe <<int>>
}
```

- `resample`: The resample primitive re-samples the input clip linearly to produce a new clip with the desired frame count.

```
resample
{
        <<clip>>
        framecount <<int>>
}
```

- `loop`: The loop primitive creates a new clip by repeating the frames between `loopstart`

and `loopend` frames, `loopcount` times.

```
loop
{
        <<clip>>
        loopstart <<int>>
        loopend <<int>>
        loopcount <<int>>
}
```

- `transition`: The transition primitive synthesizes a transition from inputclip 1's *out frame* to inputclip 2's *in frame* using the specified blend window.

```
transition
{
        <<clip 1>>
        <<clip 2>>
        outframe <<int>>
        inframe <<int>>
}
```

- `graft`: The graft primitive synthesizes a motion graft for the specified graft chains of the base clip using the graft source. The baseStart, baseEnd specify the base action frames that are to be modified. The graftStart and graftEnd frames specify the graft source action frames.

```
graft
{
        <<baseclip>>
        <<graftclip>>
        baseStartFrame <<int>>
        baseEndFrame <<int>>
        graftStartFrame <<int>>
        graftEndFrame <<int>>
        graftChains
```

```
        [
                <graft chain alias1>
                ...
        ]
}
```

- `alterwalk`: The alterwalk primitive alters the stride and foot lift of the walk and adapts the walk to an inclined plane.

```
alterwalk
{
        <<clip>>
        stride <<float>>
        footlift <<float>>
        incline <<float>>
}
```

A sample actor definition for an actor named *Phil* is shown in Figure 7.6. The '`actor`' keyword begins the actor definition. This is immediately followed by the name of the actor that is being defined, in this case *Phil*.

### 7.3.2 Story Script

The story script consists of three parts - a cast specification, a scenes collection and a screenplay. Figure 7.7 depicts the structure of the story section.

#### Cast

The cast collection specifies all actor instances that participate in the story. Each cast member is an instance of an actor defined in the `actors` collection. The *actor alias* references the actor definition and the *instance alias* names the cast member. This alias is then used to refer to the cast member in the story.

```
actors
[
        actor "Phil"
        {
                hierarchy "c:\actors\phil\phil.asf"
                geometry "c:\actors\phil\phil.geom"
                mocapdb
                [
                        clip "walk"
                        {
                                filename "c:\actors\phil\walk1.amc"
                        }
                ]
                graftchains
                [
                        graftchain "leftLeg"
                        {
                                noderef "root"
                                noderef "lhipjoint"
                                noderef "lfemur"
                                noderef "ltibia"
                                noderef "lfoot"
                                noderef "ltoes"
                        }
                ]
                actions
                [
                        action "walk"
                        {
                                clipref "walk"
                        }
                ]
        }
]
```

Figure 7.6: Sample actor definition for actor ‘Phil’

```
story
{
        cast
        [
                <<actor alias> <<instance alias>>
                ...
        ]
        scenes
        [
                scene <<alias>> {...}
        ]
        screenplay
        [
                <<scene alias>>
                ...
        ]
}
```

Figure 7.7: The story structure.

**Scene**

The `scenes` collection contains the definition of one or more individual scenes of the story. Each `scene` of the story has an initialization section and a screenplay section as shown in 7.8. The initialization section allows positioning and orientation of the cast members involved in the scene and of the camera. Of the cast members defined in the story, only those referenced in the scene' s initialization section are instantiated for the respective scenes. The actors are hidden. The camera and actor instances are repositioned at the beginning of each scene.

The scene screenplay section contains action invocations on actor instances. The action invocations can be grouped into parallel and sequential flows by using `par`, and `seq` blocks respectively. Actions invoked within a `par` block are executed simultaneously. Actions invoked within a `seq` block are invoked sequentially. `par` and `seq` sections can be nested within each other. Multiple action invocations on the same actor instance within a `par` block are executed sequentially. The screenplay section, itself, behaves like a *seq* block. Figure 7.9 shows an example with `par` and `seq` blocks. Here, four actor instances – instance 1-4 – are used. Different actions are invoke on each of the instances, viz.

- actions 1, 5 and 7 are invoked on instance 1
- actions 2, 6 and 11 are invoked on instance 2

```
scene
{
        initialize
        {
                camera
                {
                        position <<float>> <<float>> <<float>>
                        lookat <<float>> <<float>> <<float>>
                }
                <<instance name>>
                {
                        position <<float>> <<float>> <<float>>
                        lookat <<float>> <<float>> <<float>>
                }
                ...
        }
        screenplay
        {
                <<instance name>>.<<action name>>
        }
}
```

Figure 7.8: The scene structure.

- actions 3, 8 and 12 are invoked on instance 3
- actions 4, 9 and 10 are invoked on instance 4.

. These actions are grouped together into different `par` and `seq` blocks. Figure 7.10 shows the resulting action flow in time. At the scene screenplay level, the seq block is executed followed by the par block, in sequence. For the `seq` block, actions 1 and 2 are executed in parallel, followed by actions 3 and 4 executed in parallel. This is followed by actions 5 and 6 in sequence. For the subsequent `par` block, the two nested sequential blocks are executed in parallel the first containing actions 7, 8, 9 and the second containing actions 10, 11 and 12. Figure 7.10 depicts the scenario for nearly equal length action blocks. For actions with different lengths contained in a parallel block, upon completion, the actor instances with shorter actions wait till the longest action sequence is completed.

### 7.3.3 Story screenplay

The `screenplay` section of the `story` is a collection of scene alias references. The output animation is synthesised using the scene order specified in this section. At playback time, the

```
scene
{
        initialize{...}
        screenplay
        {
                seq
                {
                        par
                        {
                                <<instance 1>>.<<action 1>>
                                <<instance 2>>.<<action 2>>
                        }
                        par
                        {
                                <<instance 3>>.<<action 3>>
                                <<instance 4>>.<<action 4>>
                        }
                        <<instance 1>>.<<action 5>>
                        <<instance 2>>.<<action 6>>
                }
                par
                {
                        seq
                        {
                                <<instance 1>>.<<action 7>>
                                <<instance 3>>.<<action 8>>
                                <<instance 4>>.<<action 9>>
                        }
                        seq
                        {
                                <<instance 4>>.<<action 10>>
                                <<instance 2>>.<<action 11>>
                                <<instance 3>>.<<action 12>>
                        }
                }
        }
}
```

Figure 7.9: `par` and `seq` flows.

actions in the scene's screenplay section are invoked.

### 7.3.4 Generating the script

We generate the script and action definitions manually. The reuse primitives – transition, loop and graft – are specified by interactively querying our cluster graph. Our mocap workbench

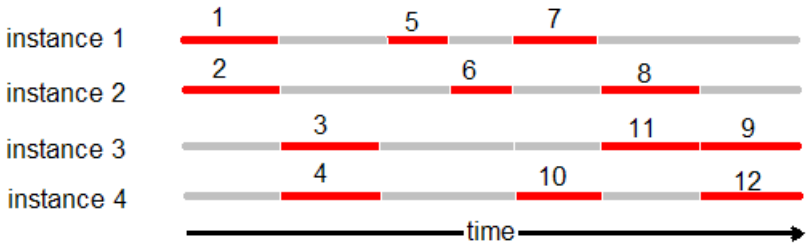



Figure 7.10: Action control flow for script in 7.9.

tool allows loading motion clips and building the cluster graph. The cluster graph is built once for each actor and reused for interactive querying. Once appropriate transition points are identified, we use our MRSS tool to synthesize and preview individual actions. The scripting engine generates its own synthesis from the script annotations. The API used for motion reuse and synthesis across all our tools is identical and packaged as a reusable dynamic link library. This API is described in Appendix B.

## 7.4 A Script Sample

```
actors
[
        actor "Footballer"
        {
                hierarchy "c:\actors\16.asf"
                geometry "c:\actors\phil\phil.geom"
                mocapdb
                [
                    clip "walk"
                    {
                        filename "c:\actors\phil\08_02.amc"
                    }
                    clip "extendedStrideWalk"
                    {
                        filename "c:\actors\phil\08_07.amc"
                    }
                    clip "footballkick"
                    {
                        filename "c:\actors\phil\11_01.amc"
```

```
    }
    clip "run"
    {
        filename "c:\actors\phil\16_36.amc"
    }
    clip "jump"
    {
        filename "c:\actors\phil\16_36.amc"
    }
]
graftchains
[
        graftchain "leftLeg"
        {
                noderef "root"
                noderef "lhipjoint"
                noderef "lfemur"
                noderef "ltibia"
                noderef "lfoot"
                noderef "ltoes"
        }
        graftchain "rightLeg"
        {
                noderef "root"
                noderef "rhipjoint"
                noderef "rfemur"
                noderef "rtibia"
                noderef "rfoot"
                noderef "rtoes"
        }
        graftchain "leftHand"
        {
                noderef "thorax"
                noderef "lclavicle"
                noderef "lhumerus"
                noderef "lradius"
                noderef "lwrist"
        }
        graftchain "rightHand"
```

```
        {
                noderef "thorax"
                noderef "rclavicle"
                noderef "rhumerus"
                noderef "rradius"
                noderef "rwrist"
        }
]
actions
[
        action "walk"
        {
                clipref "walk"
        }
        action "run"
        {
                clipref "run"
        }
        action "jump"
        {
                clipref "jump"
        }
        action "kickFootball"
        {
                clipref "footballkick"
        }
        action "turnLeft"
        {
                clipref "walkTurningLeft"
        }
        action "turnRight"
        {
                clipref "walkTurningRight
        }
        action "transtitionFromWalkToRun"
        {
                transition
                {
                        clipref "walk"
```

```
                                        clipref "run"
                                        outframe 200
                                        inframe 45
                                }
                        }
                        action "transitionFromRunToWalk"
                        {
                                transition
                                {
                                        clipref "run"
                                        clipref "walk"
                                        outframe 148
                                        inframe 23
                                }
                        }
                        action "march"
                        {
                                graft
                                {
                                        clipref "walk"
                                        clipref "extendedStrideWalk"
                                        baseStartFrame 32
                                        baseEndFrame 256
                                        graftStartFrame 19
                                        graftEndFrame 244
                                        graftchains
                                        [
                                                lefthand
                                                righthand
                                        ]
                                }
                        }
                ]
        }
]

story
{
        cast
```

```
{
        Footballer player1
        Footballer player2
        Footballer player3
        Footballer player4
}
scenes
[
        scene "KickBall"
        {
                initialize
                {
                        camera
                        {
                                pos 0, 0, 0
                                lookat 1.0, 0, 0
                        }
                        player1
                        {
                                pos 0, 0, 0
                                lookat 1.0, 0, 0
                        }
                        player2
                        {
                                pos 0, 0, 0
                                lookat 1.0, 0, 0
                        }
                        player3
                        {
                                pos 0, 0, 0
                                lookat 1.0, 0, 0
                        }
                        player4
                        {
                                pos 0, 0, 0
                                lookat 1.0, 0, 0
                        }
                }
                screenplay
```

```
                        {
                            par
                            {
                                player1.kickFootball
                                player4.march
                            }
                            par
                            {
                                player1.transitionFromWalkToRun
                                player2.jump
                                player3.run
                                player4.march
                            }
                            par
                            {
                                player1.run
                                player2.run
                                player3.run
                                player4.march
                            }
                        }
                }
        ]
        screenplay
        {
                scene "KickBall"
        }
}
```

## 7.5 Discussion

Our scripting environment is simple. We chose to incorporate as minimal features as were required to demonstrate the use of our motion reuse synthesis techniques. A production system would need to incorporate schemes for action synchronization in addition to our `par` and `seq` blocks. In our system actors remain frozen when they wait for other actors to play out their actions. An easy way to address this is to define idle actions for actors and build transitions

from most actions to and from the idle actions. The Improv system, [84], uses a similar notion of idle actions. [38] describes a technique to automatically determine common poses in a motion capture database, which may be used to transition to idle actions.

# Chapter 8

# Summary and Conclusions

Animating humanoid characters is a complex task that demands great skill and many resources. The objective of our work has been to simplify this task. Towards this end, we have successfully demonstrated our data driven synthesis techniques that generate new animation by reusing existing animation data. While our techniques are mainly targeted at mocap data, some of them may also be used with animation synthesized using other techniques.

Data driven synthesis techniques such as ours, are particularly useful because as new animations get created over time, a library of motion builds up and there exists a good likelihood that new requirements can be met from existing data. We note, however, that data driven synthesis can only generate animations similar to those that already exist. Given the fact that humanoid actors possesses rich repertoire of actions and variations - synthesizing new animation requires prior collection of such motion. Mocap makes it relatively easy to acquire these from live performers.

Most mocap acquired today targets specific scenes. This applies equally to animation created using other methods. Reusing such animation necessitates editing. Adaptation implies the ability to cut and splice parts for combinatorial synthesis and adherence to new constraints. The ability to specify these operations at a high level is a key enabler. Our main contributions to these are:

## Cut and splice synthesis at a sub-clip level

Chapter 4 shows how actions from individual clips can be resequenced using transition information inferred from mocap clips. Therein, we describe the cluster graph data structure that automatically clusters common subsequences between motions and creates a transition graph for these. We use such cluster graphs for sub-clip cut and splice synthesis. Cluster graphs inherently identify motion loops. We synthesize cyclic motion using this information. This re-synthesis method is fast and affords real time interactivity.

## Cut and splice synthesis at a sub-hierarchy level

Chapter 5 explores motion grafting. Grafts are motion splices that operate at sub-hierarchy level. We introduce the notion of independent kinematic chains and independent actions. We show how to graft animation of one independent kinematic chain on to different base clips while accounting for cross body correlation. Such correlation is important to synthesize believable grafts. We demonstrate the use of foot plant constraints to correlate upper and lower body motion. We define a set of conservative heuristics to automatically synthesize believable graft motion combinations.

## Parameterization of walk motions

Chapter 6 shows our parameterization of walks using stride and lift parameters. Our parameterization is based on a simple kinematic model amenable to on the fly application. We use a novel constrained planar analytical two link inverse kinematics solver to adapt our limb motions. Our parameterization is useful in adapting motion to spatial constraints. A special case of such constraints, which we handle separately, is the adaptation of horizontal locomotion to inclined planes.

## Scripting and functional composition architecture

Chapter 7 describes our filter graph architecture and scripting language. We use the filter graph architecture to support functional composition of motion primitives. The scripting language

forms the interface between the animator and our reuse techniques. It exposes our techniques as composable primitives. It allows definition and instantiation of actors and scenes. It supports authoring of parallel and sequential animation flows. It allows the animator to build his own vocabulary for enhanced flexibility and ease of use.

Our methods, can be improved upon in many areas. In the rest of this chapter we discuss the applicability and limitations of our methods and suggest areas for future work.

## 8.1 Future work

Development in mocap reuse and synthesis techniques promise to increase the speed of creating animation while enhancing quality and making animation accessible to novice animators. These goals may be achieved by improving upon the following:

- **Parameterization:**
  Motion parameterization is the key component to enabling reuse. Every variation of an action can be expressed as a parameterization of the base action. For example, a hand wave can be performed with the hand raised high above the head or with the hand at eye-level. The height of the end-effector can be considered as a parameter in this example. Another example is of different types of walks – fast walk, tired walk, slow walk can all be considered variations of "the normal walk." Here the parameters are a little more difficult to define. In our work, we derived the parameters, foot lift and stride, from observation. However, not all parameters are easily observable. Parameterization can benefit from using one of the following techniques:

  **Data driven parameterization**
  Machine learning methods can be employed to learn parameters. An example of this is provided by style machines [15].

  Clustering and principal component analysis may also be used to identify similar data. Our cluster graphs are useful for clustering data. Suitably modifying cluster graphs to collect similar actions can be a useful approach to parameterization. The granularity of a cluster graph node is at a sub activity level. A collection of cluster nodes are required to identify a complete activity. An improved motion comparison function can be defined

to collect similar motions with different variations. These can then be used to define a parameter space. The parameter space may have dimensions corresponding to end-effector positions, velocity, acceleration etc. Interpolation in this space can then be used to vary any or all of style, rate or positions of the end effectors.

**Model driven parameterization**

While data driven parameterization is useful for generating blended variations, model driven parameterization enables tweaking individual clips of known motion type. Our walk parameterization scheme is such an example. Dynamic simulation models exist for simulating human athletics,[43], for evaluating upper body dynamics [124], and for automatic gait animation [101]. Adaptation guided by such models may be used to synthesize parameterized variations. Methods that correct physical accuracy can be included as post process steps. Such techniques have been suggested in [27], [94], [85], [124].

- **Generalized grafts:**

  Grafts provide combinatorial synthesis, increasing the action combinations available for reuse, without requiring re-acquisition. Our motion grafting implementation is limited by the fact that we use foot plant constraints. Alternative methods of defining correlation will allow grafting between any type of motion. A dynamic time warping scheme to correlate motion is defined in [54]. A method, with goals similar to ours using the DTW scheme is described in [40]. It would be instructive to generalize or compare these correlation methods directly vis-a-vis their usefulness for grafting. A useful extension would be to correlate based on other events of interest such as zero crossings of joint accelerations or on sub-hierarchies.

- **On the fly synthesis:**

  Our technique is interactive. A reasonable goal is to extend our reuse technique to support on the fly synthesis. Such an extension could be used to drive in game characters, where motion and transitions are generated on the fly. A key problem that needs to be solved in this direction is enforcement of timing constraints. The latency between an action change request and the time it actually changes has to be bounded. Current constraint based methods use dynamic programming or randomised search strategies. These are computationally expensive and hence not of much use in this context. Motion planning strategies can also benefit from on the fly motion adaptation.

- **Behavioral animation:**

  A larger goal for animation reuse is to understand the semantics of the underlying motion. This remains yet a challenge. With better understanding of action semantics and variations, controlling animation by simulating higher level mental processes will be more amenable. Tying actions and variations to high level goals, moods and external constraints is the realm of behavioral animation. Reuse methods will prove useful in this context.

- **Tool Support:**

  Our work is applicable to areas using animated humanoid actors. We envisage that motion reuse methods like ours will become popular in the future. As more reuse methods are developed, the way in which animations are created will change. Instead of capturing motions for specific scenes, motion variations will be captured for general use and then targeted to a specific scenario. For these methods to be widely accepted, they need to be incorporated into industry standard content creation tool chains. The tools that we have developed are interactive and disjoint. They require significant user intervention. Output of one tool needs to be interpreted and reformatted for use with another tool. This process can be automated. Combining these tools and providing a user interface for script generation will result in tremendous productivity improvement. Our motion reuse library supports motion definition in industry standard .bvh and .asf file formats. Therefore integrating our tool set with standard 3D content creation programs such as 3D Studio Max, Maya, Blender and others, should be relatively simple. This will make ours and other such techniques accessible to the content creators at large, leading to wider adoption of these methods.

In conclusion, we feel that motion reuse and synthesis methods have much unexplored potential. They will become the favoured methods for generating character animation in the near future.

# Appendix A

# Mocap Workbench

## A.1 Introduction

Mocap workbench is an application written by us to provide a graphical user interface for our reuse and synthesis techniques. It allows users to view mocap animation, create displacement maps, build cluster graphs and export motion clips. Figure A.1 shows the mocap workbench.

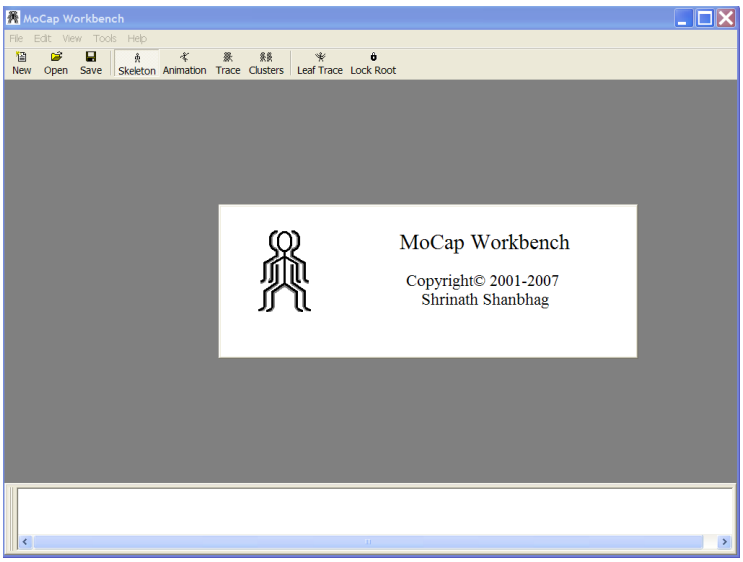



Figure A.1: The Mocap Workbench

Mocap workbench uses the concept of projects to group motions used for editing. Projects are collection of a skeletal hierarchy, related motion clips and derived information. Motion editing commands are only available in the context of a project. A user starts off by creating a blank project. The next step is to import a skeleton definition to this project. Mocap workbench

handles mocap files defined in the “bvh” and “asf/amc” formats. After importing the skeleton, the user imports motion clips that he intends to work with. This collection of skeleton and clips can be saved as a project. In addition, projects also contain cluster graph information, if one has been created for this project. Saved projects can be opened for later editing.

## A.2 Workspace

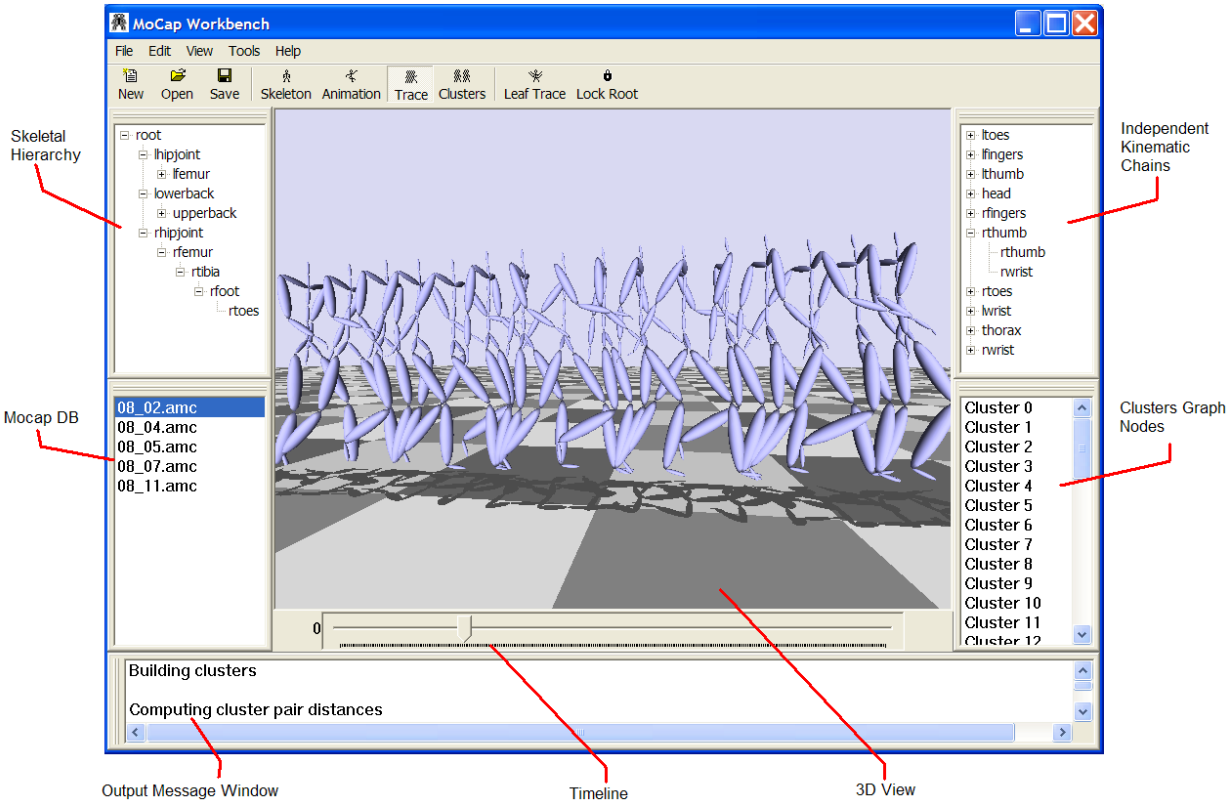



Figure A.2: Mocap workbench workspace.

Figure A.2 shows the mocap workbench workspace. This workspace is divided into multiple panes. The panes are:

1. Skeletal hierarchy pane: This pane shows the skeletal hierarchy. It allows the user to select joints for further operations.

2. Mocap database pane: This pane shows the list of clips that are imported into this project. It allows the user to select the clips for further processing or display.

3. Independent kinematic chain pane: This pane shows the independent kinematic chains inferred for the input skeletal model.

4. Cluster graph node pane: This pane shows the clusters graph nodes generated when a cluster graph is built. It allows the user to select the clusters for display or query.

5. Output pane: This pane is used to provide feedback messages. It displays information about current operations, their progress and their results. The cluster graph node attributes are displayed in this pane.

6. 3D View: This pane is used to display 3D rendering of the current clips. The user can select between skeletal view, trace view, animation and cluster view. These views are described in Section A.3.

7. Time line: The time line scroll bar controls the current frame. It updates itself to reflect the total number of frames in the current clip whenever a new clip is selected from the mocap database pane.

## A.3 Menus commands

Figure A.3 shows the menu commands of the mocap work bench.

1. File Menu: The file menu allows the user to open, save or create projects. It also allows the user to import and export mocap data. The import and export options as provide as corresponding sub-menus. The Import→Leaf Weights command is used to import joint weights used in frame similarity metric computation.

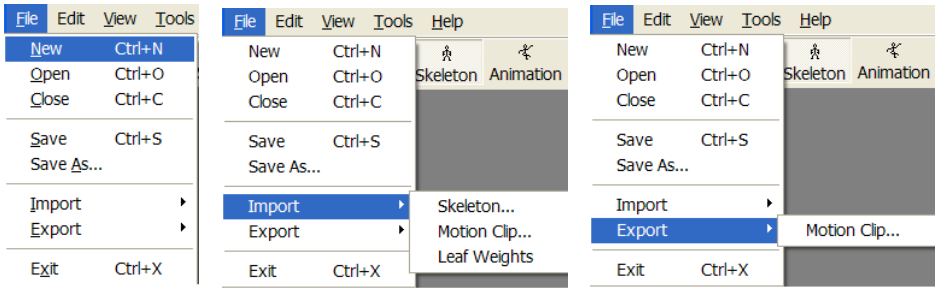
2. Edit Menu: The edit menu allows the user to view the individual DOF signals, construct motion displacements maps and motion warps and select clips for grafting. Graft chains are selected using the Edit→IK Parameters command.

3. View Menu: The view menu allows the user to select one of the following views:

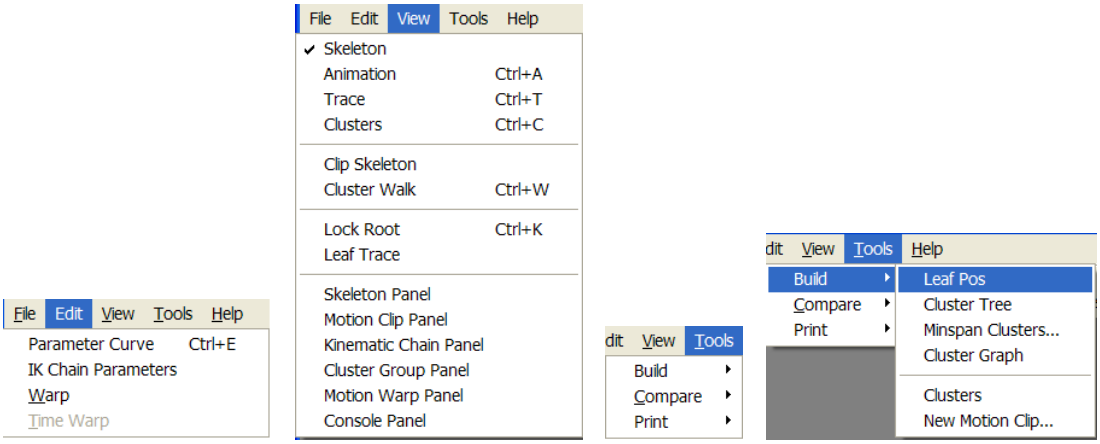
   (a) Skeletal view: This view displays the skeletal model. The display shows the current frame selected using the time line.

   (b) Animation view: This view displays animation for the currently selected motion clip. All demo animation clips were screen captured from the animation view.

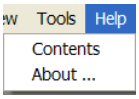
   (c) Trace view: This view displays every fifteenth frame from from the time line. This




(a) File Menu (b) Import Menu (c) Export Menu




(d) Edit Menu (e) View Menu (f) Tools Menu (g) Build Menu




(h) Help Menu

Figure A.3: Menu Commands

view has been used to generate screen shots of results depicted in this thesis.

(d) Cluster View: The cluster view displays the frames clustered in a cluster graph node. Additionally it allows the user to turn on and off the display of leaf positions traces. The Lock Root command is used to switch to relative displacement mode. This command effects both the display and computation. When switched on all computations are performed in the neutral frame of reference described in 4.2.1.

4. Tools Menu: The tools menu allows access to algorithms and attributes. The build sub menu is used for creating cluster graphs. It has commands corresponding to each step of

the cluster graph building process, see task 13 of Section A.4.

5. Help Menu: An html version of this users guide is displayed using the Help→Contents command. Program information is displayed using the Help→About command.

## A.4 Tasks

This section describes how to accomplish the tasks below in Mocap workbench.

1. **Create New Project**: Select the File→New command or the tool bar 'New' from the button.
2. **Open Existing Project**: Select the File→Open command or the 'Open' button from the tool bar.
3. **Save Project**: Select the File→Save command or the 'Save' button from the tool bar.
4. **View Skeleton**: Select the View→Skeleton command or the 'Skeleton' button from the tool bar.
5. **View Animation**:Select the View→Animation command or the 'Animation' button from the tool bar.
6. **View Trace**: Select the View→Trace command or the 'Trace' button from the tool bar.
7. **View Leaf Positions**: Select the View→Leaf Trace command or the 'Leaf Trace' button from the tool bar.
8. **View Clusters**: Select the View→Clusters command or the 'Clusters' button from the tool bar.
9. **View DOF Curve**:
   (a) Select the joint you wish to view in the skeletal hierarchy pane.
   (b) Select the Edit→Parameter Curve command. The Edit Graph Window Pops up, see Figure A.4.
   (c) Select the DOF channel in the Edit Graph Window.
10. **Import Skeleton**:

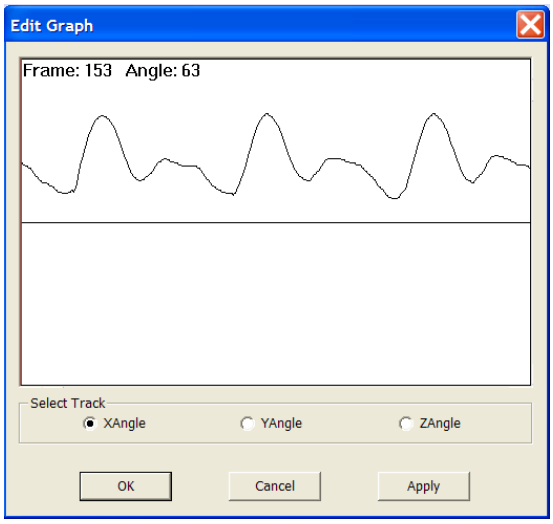



Figure A.4: Edit Graph window.

(a) Select the File→Import→Skeleton command. The Open file dialog box pops up.

(b) Select the desired skeleton file to load and Click Ok.

11. **Import Motion Clip**:

(a) Select the File→Import→Motion Clip command. The Open file dialog box pops up.

(b) Select the desired mocap file to load and Click Ok.

12. **Import Joint Weights**:

(a) Select the File→Import→Leaf Weights command. The Open file dialog box pops up.

(b) Select the desired leaf weights file to load and Click Ok.

13. **Build cluster graph** :

(a) Open a saved project or create a new one. If you created a new project, import a skeleton clip, associated motion clips files and joint weights.

(b) Select the Lock Root command to turn on neutral frame of reference computing.

(c) Select Tools→Build→Cluster Tree command. This starts the process of building the cluster tree. The output message pane displays progress. Wait for this process to complete.

(d) Select Tools→Build→Min-span Clusters command. A dialog box pops up to collect the max frame error metric, see Figure A.5. Enter desired error metric and click Ok. Output message window displays progress of the operation. Wait for this process to complete.

(e) Select Tools→Build→Clusters Graph command. The cluster node clip-frame sequences and in and out edge information is displayed in the output message box.

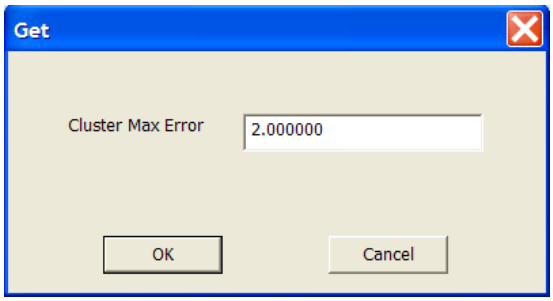



Figure A.5: Frame Error Metric dialog box.

14. **Changing View**: Select the desired view from the View menu or from the tool bar.

15. **Navigate in 3D View**: Click on the 3D pane. The following key commands are defined.
`Up Arrow`: Move forward.
`Down Arrow`: Move backward.
`Left Arrow`: Turn left.
`Right Arrow`: Turn right.
`Shift + Up Arrow`: Increase camera height.
`Shift + Down Arrow`: Decrease camera height.

16. Build displacement map: Select the File→Save command or the 'Save' button from the tool bar.

17. **Create Transitions and Loops**:

(a) Select Build→New Motion command. The "New Motion Clip" wizard window pops up, see Figure A.6.

(b) Select the save filename for the new motion.

(c) Select 'Cluster Walk' option as the creation method.

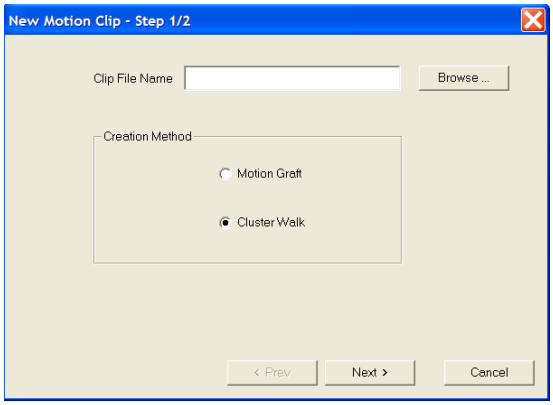



Figure A.6: New Motion Dialog.

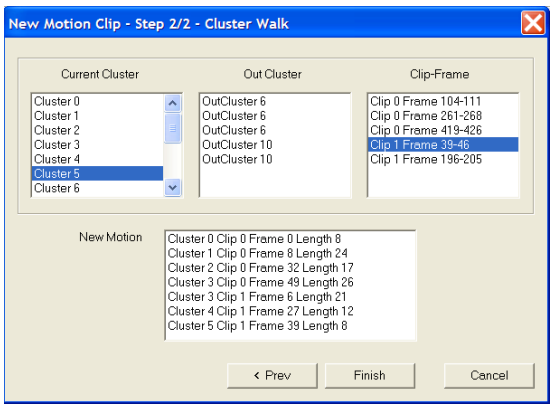



Figure A.7: Cluster Walk Dialog.

(d) Select Next to walk the cluster graph and synthesize new motion. The cluster walk dialog pops up, see Figure A.7.

(e) Highlight start cluster from the ‘Current Cluster’ List.

(f) The ‘Cur Clip-Frame Sequence’ selects the clip-frame sequences clustered in this graph node and the ‘Out Cluster’ list displays all the nodes connected to this node. Select an cluster from the ’Out Cluster List’. The ’Cur Clip-Frame Sequence’ shows clip-frame sequences leading to the selected out cluster.

(g) Select the desire transition clip-frame sequence by double clicking its entry in the ’Cur Clip-Frame Sequence’ list box. The out cluster selected becomes the selected cluster in the ’Current Cluster’ list box.

(h) Repeat steps 17e to 17h to build desired new clip.

(i) Click finish when done. The new clip is synthesized, added to the mocap database and available for selection from the motion database pane.

18. **Create Grafts**:

(a) Select the Edit→IK Chain Parameters command. The IK Chain Properties window pops up, see Figure A.8.

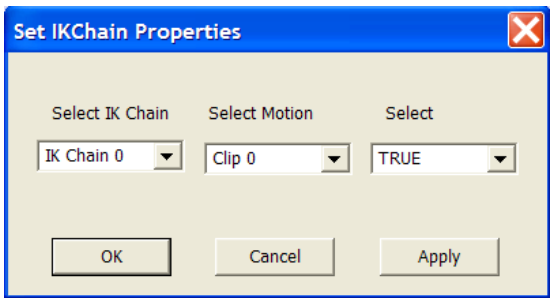



Figure A.8: IK Chain Properties Window

(b) Map independent kinematic chains to the desired motion clip using the "Select IK Chain" drop down to select the kinematics chain and "Select Motion" to select the desired graft motion.

(c) Ensure that the same clip is selected for all lower body chains. Similarly ensure that the same clip is selected across all upper body chains. The lower body clip is treated as the base motion. The upper body clip is selected as the graft motion.

(d) The third drop down allows selective grafting of motion from the graft motion clip. Set it to true for the chains where the base motion has to be replaced by the graft motion.

(e) Click OK to generate graft. The generated clips is added to the Mocap database and displayed in the mocap database pane.

19. **Export Motion Clip**:

(a) Select the File→Export→Motion Clip command. The "Save As" dialog box opens.

(b) Enter the desired filename and click Ok.

# Appendix B

# Mocap Reuse and Synthesis Library API

This appendix lists the public application programming interface (API) of our mocap reuse and synthesis library. This API consists of exported functions, data types and their public members.

## B.1 Structs and Classes

1. **Vector**: This struct. represents a three dimensional vector.

   **Attributes**

| |
|---|
| `float x;`<br>The x-component of this vector. |
| `float y;`<br>The y-component of this vector. |
| `float z;`<br>The z-component of this vector. |

**Methods**

| |
|---|
| `Vector();`<br>Constructor. Initializes the vector to (0, 0 0). |
| `Vector operator + (const Vector & v);`<br>Addition operator. |
| `Vector operator - (const Vector & v);`<br>Subtraction operator. |
| `Vector operator - (int);`<br>Unary negation operator. |
| `friend float Dot(const Vector & v1, const Vector & v2);`<br>Returns the dot product of two vectors. |
| `friend Vector Cross(const Vector & v1, const Vector & v2);`<br>Returns the cross product of two vectors; |
| `friend Vector operator * (float scale, Vector & v);`<br>Scales the vector by the specified "scale" factor. |
| `Vector operator / (float scale);`<br>Scales the vector by inverse of the specified "scale" factor. |
| `operator float ();`<br>float cast operator. Returns the magnitude of the vector. |
| `float Magnitude();`<br>Returns the magnitude of the vector. |

2. **Point**: This struct represents a point using homogeneous coordinates.

**Attributes**

| |
|---|
| `float x;`<br>The x-component of this point. |

| |
|---|
| `float y;`<br>The y-component of this point. |
| `float z;`<br>The z-component of this point. |
| `float w;`<br>The w-component of this point. |

**Methods**

| |
|---|
| `Point();`<br>Constructor. Initializes the point t0 (0, 0, 0, 1) |
| `Vector operator - (const Point & pt);`<br>Subtraction operator. Returns the vector between two points. |
| `Point operator - (const Vector & v);`<br>Translates the point by subtracting the specified vector. |
| `Point operator + (const Vector & v);`<br>Translates the point by adding the specified vector. |
| `void Homogenize();`<br>Divides attributes of point by `w`. |

3. **Matrix**: Represents a homogeneous 4x4 transformation matrix.

**Attributes**

| |
|---|
| `float val[4][4];`<br>The two dimensional array representation of the matrix. |

**Methods**

| |
|---|
| `Matrix();`<br>Constructor. Initializes the matrix to identity matrix. |

| |
|---|
| `float & Get(int row, int col);`<br>Returns the element at position specified by “row” and “col.” |
| `operator float * ();`<br>Returns a pointer to the first element of the matrix memory block. |
| `float *operator[](int row);`<br>Returns a pointer to the first element of the specified row. |
| `void Identity();`<br>Set the matrix to identity matrix. |
| `void SetTranslate(float x, float y, float z);`<br>Sets the components of the matrix to translate by the given x, y and z offsets. |
| `void SetRotateX(float angle);`<br>Set the components of the matrix to rotate about the X-Axis by the specified angle (in degrees). |
| `void SetRotateY(float angle);`<br>Set the components of the matrix to rotate about the Y-Axis by the specified angle (in degrees). |
| `void SetRotateZ(float angle);`<br>Set the components of the matrix to rotate about the Z-Axis by the specified angle (in degrees). |
| `void SetScale(float sx, float sy, float sz);`<br>Sets the components of the matrix to scale by sx, sy and sz about the origin. |
| `Matrix operator * ( const Matrix &mat);`<br>Post multiplication operator. |
| `Matrix Transpose();`<br>Returns the transpose of the matrix. |
| `Matrix & operator = (Matrix &m);`<br>Assignment operator. |

4. **Transform**: Represents the transform associated with a joint.

**Attributes**

| |
|---|
| `float m_tx;`<br>X - translation for this joint in the parents coordinate system. |
| `float m_ty;`<br>Y - translation for this joint in the parents coordinate system. |
| `float m_tz;`<br>Z - translation for this joint in the parents coordinate system. |
| `float m_angleX;`<br>X - rotation (in degrees) for this joint in the parents coordinate system. |
| `float m_angleY;`<br>Y - rotation (in degrees) for this joint in the parents coordinate system. |
| `float m_angleZ;`<br>Z - rotation (in degrees) for this joint in the parents coordinate system. |
| `float m_sx;`<br>X - scale factor for this joint in the parents coordinate system. |
| `float m_sy;`<br>Y - scale factor for this joint in the parents coordinate system. |
| `float m_sz;`<br>Z - scale factor for this joint in the parents coordinate system. |
| `float m_offX;`<br>X - offset for this joint in its own coordinate system. |
| `float m_offY;`<br>Y - offset for this joint in its own coordinate system. |
| `float m_offZ;`<br>Z - offset for this joint in its own coordinate system. |
| `char m_szRotOrder[4];`<br>Order in which rotations are to be composed. For example 'XYZ' implies R(x)*R(y)*R(z). |

**Methods**

| |
|---|
| `Transform();`<br>Constructor. |
| `Matrix GetTransform();`<br>Returns the composite transformation matrix. |
| `Matrix GetRotation();`<br>Returns the rotation matrix. |
| `Matrix GetTranslation();`<br>Returns the translation matrix; |
| `Matrix GetScale();`<br>Returns the scale matrix. |
| `Matrix GetOffset();`<br>Returns the offset matrix. |
| `Matrix GetInverseTransform();`<br>Returns the inverse of the transform matrix. |
| `Matrix GetInverseRotation();`<br>Returns the inverse of the rotation matrix. |
| `Matrix GetInverseTranslation();`<br>Returns the inverse of the translation matrix. |
| `Matrix GetInverseScale();`<br>Returns the inverse of the scale matrix. |
| `Matrix GetInverseOffset();`<br>Returns the inverse of the offset matrix. |
| `void SetRotateOrder(char *szRotOrder);`<br>Sets the rotation order of this transform. |

5. **DOF**:

   An enum identifying each unique DOF (translation, rotation, scale and offset) associated with a joint.

6. **Joint**: This class represents a node of the skeletal hierarchy that represents a joint. The hierarchy is defined by parent child relationships amongst various joints.

### Attributes

| |
|---|
| `string m_name;`<br>Name of the node. |

### Methods

| |
|---|
| `void AddChild(Joint *pJoint);`<br>Add a child node. |
| `void RemoveChild(Joint *pJoint);`<br>Remove a child node. |
| `Joint *GetChild(int i);`<br>Return the $i^{th}$ child of this node. |
| `Joint *GetChild(string name);`<br>Return the child with the specified name. |
| `int ChildCount();`<br>Returns the number of child nodes. |
| `Joint *FindDescendant(string name);`<br>Returns a descendant node with the specified node or NULL if not found. |
| `Joint *GetParent();`<br>Returns the parent node. |
| `Transform & GetAxisTransform();`<br>Returns the axis transform for this node. |
| `Transform & GetModellingTransform();`<br>Returns the current modelling transform for this node. |
| `Transform & GetModellingTransform(int clip, int frame=0, bool bTransform=true);`<br>Returns the modelling transform for the specified clip and frame. |

| |
|---|
| `Matrix GetModellingMatrix();`<br>Returns the current modelling matrix. |
| `Matrix GetModellingMatrix(int clip, int frame=0, bool bTransform=true);`<br>Returns the modelling matrix for the specified clip and frame. |
| `Matrix GetInverseModellingMatrix(int clip, int frame=0, bool bTransform=true);`<br>Returns the inverse of the modelling matrix for the specified clip and frame. |
| `Matrix GetWorldMatrix(int clip, int frame=0, bool bTransform=true);`<br>Returns the world matrix for the specified clip and frame. |
| `Matrix GetInverseWorldMatrix(int clip, int frame=0, bool bTransform=true);`<br>Returns the inverse of the world matrix for the specified clip and frame. |
| `Joint *GetRoot();`<br>Returns the root of the skeletal hierarchy. |
| `int GetClipCount();`<br>Returns the number of clips associated with this skeletal hierarchy. |
| `void AddClip(MotionClip *pClip);`<br>Adds a new motion clip to the skeletal hierarchy. |
| `void RemoveClip(MotionClip *pClip);`<br>Removes a motion clip from this skeletal hierarchy. |
| `MotionClip *GetClip(int i);`<br>Returns the specified motion clip. |
| `int GetClipIndex(MotionClip *pClip);`<br>Returns the index of the specified motion clip. |
| `void SetCurrMotionClip(int index);`<br>Sets the current motion clip for the hierarchy. |

| |
|---|
| `void NextFrame(int frame, float blendFactor);`<br>Updates the dof values to the next frame of the current clip. |
| `void NextFrame(int clipIndex, int frame, float blendFactor);`<br>Updates the dof values to the specified next frame of the current clip. Interpolates between nextFrame and (nextFrame+1) by the given blendFactor. |
| `void NextFrame(int clip1, int clip1frame, int clip2, int clip2frame, float clipBlendFactor, float frameBlendFactor);`<br>Updates the dof values. The dof values are set using trilinear interpolation between two clips The frame blend factor controls the interpolation between frames of each of the clip and the clip blend factor specifies the interpolation between clips. |
| `void NextFrame(int frame, float blendFactor, bool bRelPos);`<br>Updates the dof values to the next frame of the current clip and interpolates between frame and (frame+1) by the given blend factor. |
| `void NextFrame(MotionClip *pClip, int frame, float blendFactor);`<br>Updates the dof values to the next frame of the specified motion clip and interpolates between frame and (frame+1) by the given blend factor. |
| `float GetDOFPosX();`<br>Returns the X translation dof component. |
| `float GetDOFPosY();`<br>Returns the Y translation dof component. |
| `float GetDOFPosZ();`<br>Returns the Z translation dof component. |

| |
|---|
| `void SetDOFPosX(float val);`<br>Sets the X translation dof component. |
| `void SetDOFPosY(float val);`<br>Sets the Y translation dof component. |
| `void SetDOFPosZ(float val);`<br>Sets the Z translation dof component. |
| `float GetDOFRotX();`<br>Returns the X rotation dof component (in degrees). |
| `float GetDOFRotY();`<br>Returns the Y rotation dof component (in degrees). |
| `float GetDOFRotZ();`<br>Returns the Z rotation dof component (in degrees). |
| `void SetDOFRotX(float val);`<br>Sets the X rotation dof component (in degrees). |
| `void SetDOFRotY(float val);`<br>Sets the Y rotation dof component (in degrees). |
| `void SetDOFRotZ(float val);`<br>Sets the Z rotation dof component (in degrees). |
| `float GetDOFScaleX();`<br>Returns the X scale dof component. |
| `float GetDOFScaleY();`<br>Returns the Y scale dof component. |
| `float GetDOFScaleZ();`<br>Returns the Z scale dof component. |
| `void SetDOFScaleX(float val);`<br>Sets the X scale dof component. |
| `void SetDOFScaleY(float val);`<br>Sets the Y scale dof component. |

| |
|---|
| `void SetDOFScaleZ(float val);`<br>Sets the Z scale dof component. |
| `float GetDOFOffX();`<br>Returns the X offset dof component. |
| `float GetDOFOffY();`<br>Returns the Y offset dof component. |
| `float GetDOFOffZ();`<br>Returns the Z offset dof component. |
| `void SetDOFOffX(float val);`<br>Sets the X offset dof component. |
| `void SetDOFOffY(float val);`<br>Sets the Y offset dof component. |
| `void SetDOFOffZ(float val);`<br>Sets the Z offset dof component. |
| `void GetWorldPos(float & x, float & y, float & z, int clip=-1, int frame=0, bool bTransform=true);`<br>Gets the world position of the joint for the specified clip and frame. |

7. **MotionClip**: Represents the set of sampled DOF values for a skeletal hierarchy along with information about number of samples and the sampling rate.

   **Attributes**

| |
|---|
| `string m_name;`<br>Name of the motion clip. |
| `int m_numSamples;`<br>Number of samples. |
| `float m_sampleInterval;`<br>Time interval between two successive samples. This is the inverse of sampling frequency. |

| |
|---|
| `float *m_pSamples;`<br>Pointer to the sampled DOF values. |

8. **ClipFrame**: Represents a structure identifying a clip in an actors mocap database and a frame within it.

**Attributes**

| |
|---|
| `int m_clip;`<br>The clip index of the clip to which this frame belongs. |
| `int m_frame;`<br>The frame index. |

**Methods**

| |
|---|
| `bool operator < (ClipFrame & cf);`<br>Less than operator. Returns true when (m_clip ¡ cf.m_clip) or ( (m_clip == cf.m_clip) && (m_frame ¡ cf.m_frame)). |
| `bool operator == (ClipFrame & cf);`<br>Equality operator. Returns true if (m_clip == cf.m_clip) && (m_frame == cf.m_frame). |

9. **Sequence**: Represents a contiguous range of frames belonging to one motion clip.

**Methods**

| |
|---|
| `void Add(ClipFrame *pClipFrame);`<br>Add a clipframe to the sequence. |
| `int FrameCount();`<br>Returns the number of frames in this sequence. |
| `ClipFrame *First();`<br>Returns a pointer to the first frame of the sequence. |
| `ClipFrame *Last();`<br>Returns the a pointer to the last frame of the sequence. |
| `ClipFrame *Get(int index);`<br>Returns a clipframe pointer to the indexed frame in this sequence. |

10. **Cluster**: Represents a cluster of ClipFrames clustered together based on frame similarity.

**Attributes**

| |
|---|
| `Point *m_pCentroidLeafPos;`<br>The average centroid position of end effectors for frames in this cluster. |
| `float m_frameError;`<br>The frame error metric for frames in this cluster. |
| `vector <int> m_rgAlignmentFrames;`<br>The alignment frames for clipframe sequences in this cluster. |

**Methods**

| |
|---|
| `Cluster(int numLeafPos=0);`<br>Constructor. |
| `void SetLeafCount(int numLeafPos);`<br>Sets the leaf number of leaf joints. |
| `void AddFrame(ClipFrame *pClipFrame);`<br>Adds a clipframe to this cluster. |
| `void AddCluster(Cluster *pCluster);`<br>Adds a sub-cluster with this cluster. |
| `int FrameCount();`<br>Returns the number of frames in this cluster. |
| `int ClusterCount();`<br>Returns the number of sub clusters in this cluster. |
| `ClipFrame * Get(int index);`<br>Returns the indexed clip frame. |
| `Cluster * GetCluster(int index);`<br>Returns the indexed cluster. |
| `void BuildSequences();`<br>Builds clipframe sequences for frames in this cluster. |
| `int SequenceCount();`<br>Returns the count of clipframe sequences. |
| `Sequence *GetSequence(int index);`<br>Returns the indexed clipframe sequece. |

| |
|---|
| `bool IsClipFrameInCluster(ClipFrame *pClipFrame);`<br>Test if the specified clipframe is part of this cluster. |
| `void AddOutLink(Cluster *pCluster, int sequence);`<br>Adds an out-edge from this cluster to the specified cluster. |
| `int OutLinksCount();`<br>Returns the number of out-links. |
| `Cluster *GetOutLink(int index);`<br>Returns the indexed out-link. |
| `void AddInLink(Cluster *pCluster, int sequence);`<br>Adds an in-edge from the specified cluster to this cluster. |
| `int InLinksCount();`<br>Returns the count of in-links. |
| `Cluster *GetInLink(int index);`<br>Returns the indexed out-link. |

11. **Action**: This abstract base class represents the interface for motion clip reuse primitives. The motion primitives `clipref`, `concatenate`, `subclip`, `resample`, `loop`, `transition`, `graft`, `alterwalk` are internally implemented as classes implementing this interface.

    **Methods**

| |
|---|
| `int FrameCount();`<br>Returns the number of frames in this action. |
| `void Begin();`<br>Initializes the action. |

| |
|---|
| `bool Update(float elapsedTime);`<br>Plays out the action by incrementing the elapsedTime. |

12. **ActorDefinition**: This class represents the actor definition in the script.

    **Attributes**

| |
|---|
| `string m_name;`<br>Name of the actor. |
| `Joint *pJoint;`<br>Pointer to the actors skeletal hierarchy. |
| `vector<MotionClip *> clips;`<br>Mocap clip database for this actor. |
| `map<string, Action*> actions;`<br>Collection of actions defined for this actor. |

**Methods**

| |
|---|
| `void LoadHierarchy(char *szFileName);`<br>Loads the skeletal hierarchy from the specified file. |
| `Clip* LoadClip(char *szFileName);` |
| `void AddAction(Action *pAction);`<br>Adds an action to this actor. |

13. **ActorInstance**: Represents an instanced actor.

**Attributes**

| |
|---|
| `string m_name;`<br>Name of this actor instance. |
| `ActorDefinition *m_pActorDefinition;`<br>Pointer to the actor definition of which this is an instance. |

| |
|---|
| `float m_dx;`<br>X component of this instances world position. |
| `float m_dy;`<br>Y component of this instances world position. |
| `float m_dz;`<br>Z component of this instances world position. |
| `int m_currAction;`<br>The index of the current mocap clip being played by this instance. |

14. **IStatement**: Represents the abstract base class for a story statement. Each statement represents an action to be called on an actor instance.

**Methods**

| |
|---|
| `virtual void Begin();`<br>Initializes the statement. |
| `virtual bool Update(float elapsedTime);`<br>Plays out the action represented by this statement by incrementing the elapsed time. |

15. **ICompoundStatement**: Derives from IStatement and represents the abstract base class for a collection of story statements. The `seqblock` and `parblock` are composites of IStatement, in turn derived from IStatement. `seqblock` represents sequential flow and `parblock` represents parallel flow.

**Methods**

| |
|---|
| `virtual void Begin();`<br>Initializes the statement. |
| `virtual bool Update(float elapsedTime);`<br>Plays out the action represented by this statement by incrementing the elapsed time. |

| |
|---|
| `virtual void AddStatement(IStatement *pStatement);`<br>Plays out the action represented by this statement by incrementing the elapsed time. |

16. **SeqBlock**: Derives from ICompoundStatement and represents a collection of story statements to be executed sequentially.

17. **ParBlock**: Derives from ICompoundStatement and represents a collection of story statements to be executed in parallel.

18. **Script**: Represents a scripted story block definitions of actors and scenes.

**Attributes**

| |
|---|
| `ActorDefinitionMap m_actorDefinitions;` |
| `ActorInstanceMap m_actorInstances;` |
| `SeqBlock m_story;` |

## B.2 Functions

### B.2.1 Mocap Parsers

| |
|---|
| `Joint *ParseBVH(char *szFileName);`<br>This function parses mocap data in the BVH file format and constructs a skeletal model and motion clip. The root of the skeletal model is returned. |
| `Joint *ParseASF(char *szFileName);`<br>This function parses skeletal data in the ASF file format and constructs a the skeletal model. The root of the skeletal model is returned. |
| `MotionClip *ParseAMC(char *szFileName);`<br>This function parses motion data in the AMC file format and returns a pointer to MotionClip. |

### B.2.2 Cluster Graph

| |
|---|
| `void BuildLeafPos(Joint *pRoot, JointPtrList &leafList, vector<PointVector> &rgClipLeafPositions, JointPtrListList & ikChainList, bool bLockRoot);`<br>This function builds the leaf positions of nodes for the given clip. |
| `void BuildClusterListFromKruskalsTree(ClusterPtrList &clusterList, Cluster *pKruskalsTree, float maxError);`<br>This function traverses the min. span cluster tree and creates a list of clusters meeting the specified inter-frame error. |
| `void ComputeSequenceAlignmentFrames(Joint *pRoot, Cluster *pCluster, vector <float *> rgClips, vector <float *> rgClipsBar, vector <float *> rgClipsBarBar);`<br>Computes the sequence alignment frames for sequences in this cluster. |

```
void BuildAllClipKruskalsMinSpanClusters(
Joint *pRoot, Cluster *& rgClusters, int &
ndxNextCluster, ClipFrame *& rgClipFrames,
int & ndxNextClipFrame, JointPtrList
& rgLeafList, vector<PointVector> &
rgClipLeafPositions, ClusterPtrList & clusterList,
ClusterPairDistanceMap & distanceMap, Cluster *&
pKruskalsTree );
```

This function builds the min. span cluster tree using our clustering algorithm.

```
void BuildClusterGraph(ClusterPtrList clusterList);
```

This function builds a graph of motion transitions from the clusters in the cluster graph list created using `BuildClusterListFromKruskalsTree`.

### B.2.3 Motion Grafting

### B.2.4 Walk Parameterization

```
IK2LinkLimbSolver(Joint *pBaseJoint, Joint
*pMidJoint, Joint *pEndJoint, int clip, int frame,
Point &goalPos, bool bBVH, IK2LinkLimbSolution
&solution);
```

This function solves for the position of the midjoint and dof angles of base joint and mid joint, given the positions of base joint and the goal position for the end joint.

```
void ComputeVariableStepGoalAndRootPos(Joint *
pModel, Joint * pLeftFoot, Joint * pRightFoot,
int clipIndex, vector<Point> &rgLeftGoalPos,
vector<Point> &rgRightGoalPos, vector<Point>
&rgRootPos, float stepScale=1.0f, float
deltaStepScale=0, float liftScale=1.0f);
```

This function adjusts the walk clips stride and foot lift.

```
void ComputeStairGoalAndRootPos(Joint *pModel,
Joint *pLeftFoot, Joint *pRightFoot, int clipIndex,
vector <Point> &rgRootPos, vector <Point>
&rgLeftGoalPos, vector <Point> &rgRightGoalPos,
vector <Point> &rgSteps, float stepHeight, float
stepWidth);
```

This function adapts the given walk to the specified incline.

### B.2.5 Scripting

```
Script * ParseScript(char *szScriptFileName);
```

Parses a scripted story file and returns a pointer to the script object.

# Appendix C

# Clip Classification State Machine

This appendix describes the implementation of our clip classification scheme described in 3.2 using a state machine. The state machine comprises of four top level states, each containing entry and exit actions, internal sub-states and internal transitions. These top level states are: UNKNOWN, WALK, JUMP and RUN. Being in any of these states corresponds to being in one of their internal sub-states.

The UNKNOWN state is entered at the start of the clip and whenever an ambiguous state change is triggered by a transition from any of other three states. The UNKNOWN state has four sub-states: `Unknown_None`, `Unknown_Left`, `Unknown_Right` and `Unknown_Both`. The suffixes indicate the transitions which triggered the move to this state. For example, entry into Unknown_Left state is triggered by a change of the characters foot plant configuration via the '`L`' input token.

The Walk state has three sub-states: `Walk_Left`, `Walk_Right` and `Walk_Both`.

The Jump state has two sub-states: `Jump_None` and `Jump_Both`.

The Run state has four sub-states: `Run_Left`, `Run_Left_None`, `Run_Right`, `Run_Right_None`. The Run_Left_None and Run_Right_None states are look ahead states. The Run_Left_None state for example represents a transition caused by a foot plant configuration change from '`L`' to '`N`'. This history information is required to correctly classify the "`L N L`" and "`R N R`" jump patterns.

### The Unknown state

*E*ntry Action. Upon entry, the UNKNOWN state records the current frame number as the entry frame number.
*E*xit Actions.

1. For exits triggered by a foot plant configuration change, the unknown state updates the state of all frames from entry frame up to and not including the current frame's locomotion type with the type of the next state. i.e. if the next state to be entered is WALK, then the frame types are updated to WALK.

2. On encountering and end of clip, the state is updated to UNKNOWN. An exception to this rule is triggered when the current state is `Unknown_Both`. In this case the state of the frames are update to STAND.

. The state transitions for this state are as depicted in Figure C.1.

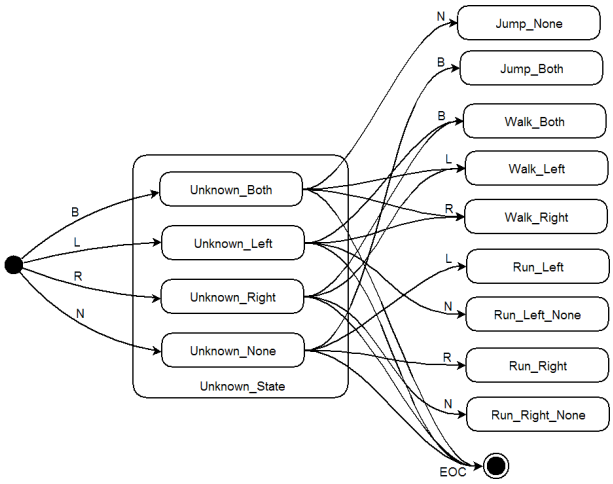



Figure C.1: Unknown-State State Transition Diagram

### The WALK, RUN and JUMP states

*E*ntry Action. Upon entry, the current frame number is recorded as the entry frame.
*E*xit Actions. Upon exit, all frames from entry frame to current frame are annotated with current state type i.e. if current state is WALK state then all frames are updated to type WALK. Similarly for RUN and JUMP states.

The state transitions for these states are as depicted in Figure C.2 for WALK state, Figure C.3 for JUMP state and Figure C.4 for RUN state. The state machine starts in one of the UNKNOWN

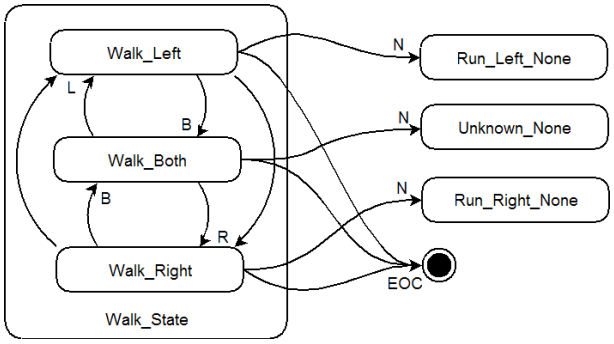



Figure C.2: Walk-State State Transition Diagram

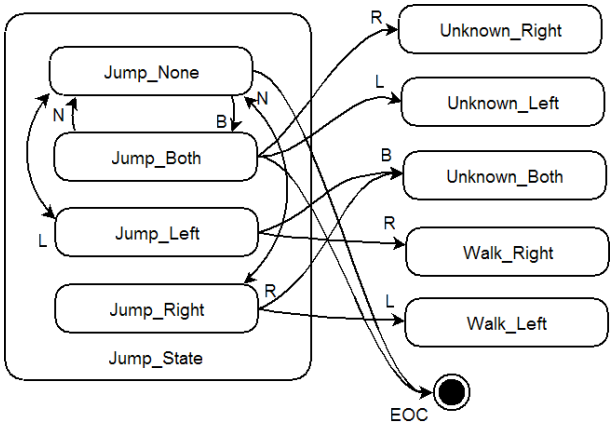



Figure C.3: Jump-State State Transition Diagram

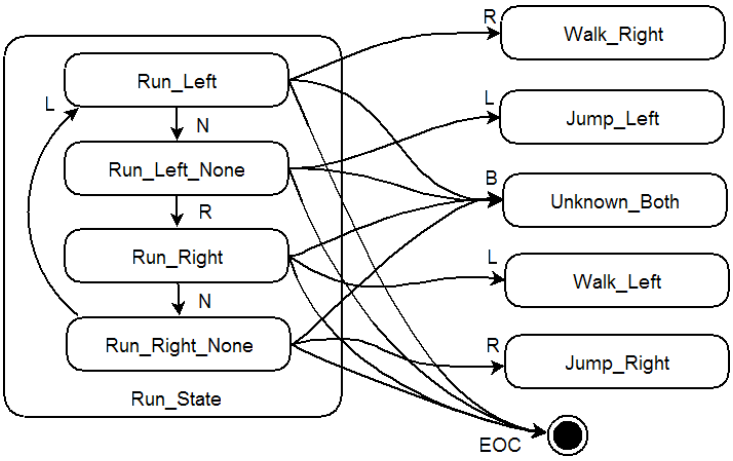



Figure C.4: Run-State State Transition Diagram

states, depending upon the characters foot plant state in the first frame. The locomotion type of the frames is “unknown” till the first transition to any of the other states. Consider a clip having

Table C.1: Sample clip with annotated foot plant states

| Frame range | Foot plant state |
|---|---|
| 1–79 | 'B' |
| 80–126 | 'L' |
| 127 –165 | 'B' |
| 166-201 | 'R' |
| 202-245 | 'B' |
| 246-275 | 'N' |
| 276- 294 | 'B' |

following foot plant constraint states as shown in Table C.1 The frames 1–79 are collected in the `Unknown_Both` state. Upon encountering an '`L`' at frame 80, the state machine transitions to the `Walk_Left` state. At this time, the exit action of the `Unknown_State` marks frames 1–79 as belonging to WALK. The state machine continues to be in the WALK state till it encounters an '`N`' token at frame 246. At this point it transitions to `Unknown_None`. This triggers the exit action of the WALK state to updates locomotive type of frames 80–245 to WALK. At frame 276 the state machine transitions from `Unknown_None` to JUMP_BOTH. At this time frames 246–275 are marked with locomotive type JUMP. After encountering frame 294, a transition is caused to the final state by 'EOC'. The exit action of the JUMP state updates the frames 276–294 to type JUMP. The frame classification at the end of this process is shown in Table C.2.

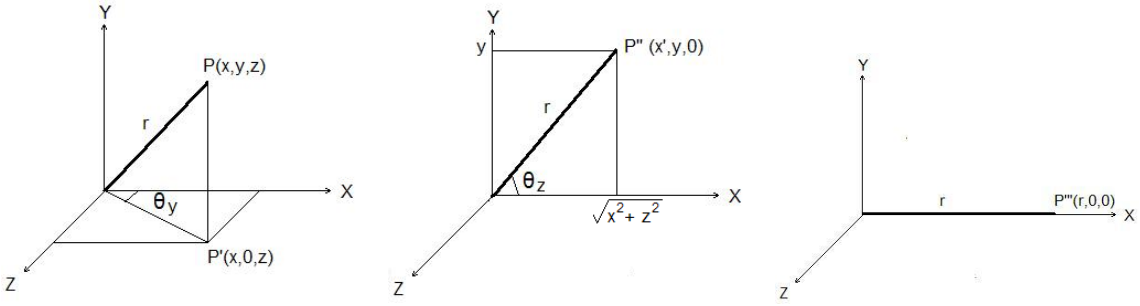
Table C.2: Locomotive type classification at the end of the state machine run.

| Frame range | Locomotive type | % Frames |
|---|---|---|
| 1–245 | Walk | 83 |
| 246–294 | Jump | 17 |

# Appendix D

# Homogeneous Rotation Matrix Computation

Given a point in $P(x,y,z)$, the homogeneous rotation matrix to rotate a point on X-axis to co-incide with $(x,y,z)$ can be computed by composing elementary rotation matrices. Figure D.1 shows the steps involved.




(a) Step 1 (b) Step 2 (c) Step 3

Figure D.1: Computing the homogeneous rotation matrix to align a point on X-axis with $P(x,y,z)$.

Let 'H' be the homogeneous rotation matrix that rotates a point on X-axis to $P(x,y,z)$. Let $\theta_y$ be the Y rotation angle, $R_y$ be the Y rotation matrix, $\theta_z$ be the Z rotation angle and $R_z$ be the Z rotation matrix.

Angles $\theta_y$ and $\theta_z$ are given by

$$\theta_y = \tan^{-1}(\frac{x}{z}) \tag{D.1}$$

$$\theta_z = \tan^{-1}(\frac{y}{\sqrt{x^2+z^2}}) \tag{D.2}$$

$R_y$ and $R_z$ are given by

$$R_y = R(-\theta_y); \tag{D.3}$$

$$R_z = R(\theta_z); \tag{D.4}$$

H is obtained as

$$H = R_z * R_y \tag{D.5}$$

# Appendix E

# CD Contents

**Mocap Reuse, Synthesis and Scripting**

**Experimental Results**

Shrinath Shanbhag
KRESIT, IIT Bombay

**Index**

| No. | Demo video |
|---|---|
| 1. | Standard Input Clips (from CMU Mocap DB) |
| 2. | Our Software (Mocap Workbench) |
| 3. | *Our Techniques*<br>i. Automatic Footplant Identification (Chapter 3)<br>ii. Interpolation Artifact (Chapter 3)<br>iii. Multi-clip Action Resequencing (Chapter 4)<br>iv. Loop Synthesis (Chapter 4)<br>v. Locomotive Motion Graft Synthesis (Chapter 5)<br>vi. Walk Parameterizations (Chapter 6)<br>vii. Scripting (Chapter 7) |

Figure E.1: Index of demo videos contained in the CD.

The accompanying CD contains videos of our animation synthesis. Figure E.1 shows a screen shot of its index page. The videos in this CD are organized by technique. The videos are encoded using "cinepack radius," "xvid" or "Windows Media" video compression, mp3 audio compression and packaged as "avi" or "wmv' 'files.

# Bibliography


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

[2] A. Ahmed, F. Mokhtarian, and A. Hilton. Parametric motion blending through wavelet analysis, 2001.

[3] A. Ahmed, F. Mokhtarian, and A. Hilton. Cyclification of human motion for animation synthesis, 2003.

[4] N. Al-Ghreimil and James K. Hahn. Combined partial motion clips. In *WSCG*, 2003.

[5] Okan Arikan and D. A. Forsyth. Interactive motion generation from examples. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 483–490, New York, NY, USA, 2002. ACM Press.

[6] Okan Arikan, David A. Forsyth, and James F. O'Brien. Motion synthesis from annotations. *ACM Transactions on Graphics*, 22(3):402–408, July 2003.

[7] G. Ashraf and K. C. Wong. Generating consistent motion transition via decoupled framespace interpolation. *Computer Graphics Forum*, 19(3), August 2000.

[8] G. Ashraf and Kok Cheong Wong. Semantic representation and correspondence for state-based motion transition. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):481–499, October-December 2003.

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

[10] Rama Bindiganavale. *Building parameterized action representations from observation*. PhD thesis, University of Pennsylvania, July 2000.

[11] Bruce M. Blumberg and Tinsley A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 47–54, August 1995.

[12] Bobby Bodenheimer, Chuck Rose, Seth Rosenthal, and John Pella. The process of motion capture – dealing with the data. In *Computer Animation and Simulation '97*, pages 3–18, September 1997.

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

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

[15] Matthew Brand and Aaron Hertzmann. Style machines. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 183–192, July 2000.

[16] Christoph Bregler, Lorie Loeb, Erika Chuang, and Hrishi Deshpande. Turning to the masters: Motion capturing cartoons. *ACM Transactions on Graphics*, 21(3):399–407, July 2002.

[17] A. Bruderlin and T. W. Calvert. Goal-directed, dynamic animation of human walking. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 233–242, New York, NY, USA, 1989. ACM Press.

[18] Armin Bruderlin and Lance Williams. Motion signal processing. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 97–104, August 1995.

[19] T. W. Calvert, J. Chapman, and A. Patla. Aspects of the kinematic simulation of human movement. *IEEE Computer Graphics and Applications*, 2(9):41–50, 1982.

[20] John E. Chadwick, David R. Haumann, and Richard E. Parent. Layered construction for deformable animated characters. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 243–252, New York, NY, USA, July 1989. ACM Press.

[21] Diane M. Chi, Monica Costa, Liwei Zhao, and Norman I. Badler. The emote model for effort and shape. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 173–182, July 2000.

[22] S. Chung and J. Hahn. Animation of human walking in virtual environments. In *Computer Animation '99*, May 1999.

[23] Michael F. Cohen. Interactive spacetime control for animation. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 293–302, July 1992.

[24] Mira Dontcheva, Gary Yngve, and Zoran Popović. Layered acting for character animation. *ACM Transactions on Graphics*, 22(3):409–416, July 2003.

[25] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. Tbag: A high level framework for interactive, animated 3d graphics applications. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 421–434, July 1994.

[26] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 251–260, August 2001.

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

[28] Claire T. Farley and Daniel P. Ferris. Biomechanics of walking and running: from center of mass movement to muscle action. *Exercise and Sport Sciences Reviews*, 26:253–285, 1998.

[29] Chris Fraley and Adrian E. Raftery. How many clusters? which clustering method? answers via model-based cluster analysis. *The Computer Journal*, 41(8):578–588, 1998.

[30] Yan Gao, Lizhuang Ma, Xiaomao Wu, and Zhihua Chen. Automatic foot-plant constraints detection shoes. In *CGIV '05: Proceedings of the International Conference on Computer Graphics, Imaging and Visualization (CGIV'05)*, pages 87–92, Washington, DC, USA, 2005. IEEE Computer Society.

[31] Michael Girard and A. A. Maciejewski. Computational modeling for the computer animation of legged figures. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 263–270, New York, NY, USA, 1985. ACM Press.

[32] P. Glardon, R. Boulic, and D. Thalmann. A coherent locomotion engine extrapolating beyond experimental data. In *Proc. of IEEE Int'l Conf. on Computer Animation and Social Agents*, pages 73–84, July 2004.

[33] Pascal Glardon, Ronan Boulic, and Daniel Thalmann. Robust on-line adaptive footplant detection and enforcement for locomotion. *Vis. Comput.*, 22(3):194–209, 2006.

[34] Michael Gleicher. Motion editing with spacetime constraints. In *1997 Symposium on Interactive 3D Graphics*, pages 139–148, April 1997.

[35] Michael Gleicher. Retargetting motion to new characters. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 33–42, July 1998.

[36] Michael Gleicher. Comparing constraint-based motion editing methods. *Graphical Models*, 63(2):107–134, March 2001.

[37] Michael Gleicher and Peter Litwinowicz. Constraint-based motion adaptation. *The Journal of Visualization and Computer Animation*, 9(2):65–94, 1998.

[38] Michael Gleicher, Hyun Joon Shin, Lucas Kovar, and Andrew Jepsen. Snap-together motion: assembling run-time animations. In *2003 ACM Symposium on Interactive 3D Graphics*, pages 181–188, April 2003.

[39] Ashraf Golam and Kok Cheong Wong. Dynamic time warp based framespace interpolation for motion editing. In *Graphics Interface*, pages 45–52, 2000.

[40] Rachel Heck, Lucas Kovar, and Michael Gleicher. Splicing upper-body actions with locomotion. *Computer Graphics Forum*, 25(3):459–466, 2006.

[41] Thomas Herter and Klaus Lott. Algorithms for decomposing 3-d orthogonal matrices into primitive rotations. *Computer & Graphics*, 17(5):517–527, 1993.

[42] A. Hilton, D. Beresford, T. Gentils, R. Smith, W. Sun, and J. Illingworth. Whole-body modelling of people from multi-view images to populate virtual worlds. *Visual Computer: International Journal of Computer Graphics*, 16(7):411–436, 2000.

[43] Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O'Brien. Animating human athletics. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 71–78, August 1995.

[44] Tae hoon Kim, Sang Il Park, and Sung Yong Shin. Rhythmic-motion synthesis based on motion-beat analysis. *ACM Transactions on Graphics*, 22(3):392–401, July 2003.

[45] Ko Hyeongseok and N. I. Badler. Animation human locomotion with inverse dynamics. *IEEE Computer Graphics and Applications*, 16(2):50–59, 1996.

[46] Leslie Ikemoto, Okan Arikan, and David Forsyth. Knowing when to put your foot down. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 49–53, New York, NY, USA, 2006. ACM Press.

[47] Leslie Ikemoto and David A. Forsyth. Enriching a motion collection by transplanting limbs. In *2004 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 99–108, July 2004.

[48] Verne Thompson Inman, Henry Ralston, and Frank Todd. *Human Walking*. Lippincott Williams & Wilkins, 1989.

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

[50] Kolja Khler, Jrg Haber, and Hans-Peter Seidel. Geometry-based muscle modeling for facial animation. In *GRIN'01: No description on Graphics interface 2001*, pages 37–46, Toronto, Ont., Canada, Canada, 2001. Canadian Information Processing Society.

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

[52] J. U. Korein and N. I. Badler. Techniques for generating the goal-directed motion of articulated structures. *IEEE Computer Graphics and Applications*, 2(9):71–81, 1982.

[53] Alexander Kort. Computer aided inbetweening. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 125–132, New York, NY, USA, 2002. ACM Press.

[54] Lucas Kovar and Michael Gleicher. Flexible automatic motion blending with registration curves. In *2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 214–224, August 2003.

[55] Lucas Kovar and Michael Gleicher. Automated extraction and parameterization of motions in large data sets. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 559–568, New York, NY, USA, 2004. ACM Press.

[56] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. *ACM Transactions on Graphics*, 21(3):473–482, July 2002.

[57] Lucas Kovar, John Schreiner, and Michael Gleicher. Footskate cleanup for motion capture editing. In *ACM SIGGRAPH Symposium on Computer Animation*, pages 97–104, July 2002.

[58] John Lasseter. Principles of traditional animation applied to 3d computer animation. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 35–44, July 1987.

[59] Joseph Laszlo, Michiel van de Panne, and Eugene L. Fiume. Interactive control for physically-based animation. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 201–208, July 2000.

[60] Jehee Lee. *A Hierarchical Approach to Motion Analysis and Synthesis for Articulated Figures*. PhD thesis, Deptartment of Computer Science, Korea Advanced Institute of Science and Technology, 2000.

[61] Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins, and Nancy S. Pollard. Interactive control of avatars animated with human motion data. *ACM Transactions on Graphics*, 21(3):491–500, July 2002.

[62] Jehee Lee and Kang Hoon Lee. Precomputing avatar behavior from human motion data. In *2004 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 79–87, July 2004.

[63] Jehee Lee and Sung Yong Shin. A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 39–48, August 1999.

[64] Yin Li, Michael Gleicher, Ying-Qing Xu, and Heung-Yeung Shum. Stylizing motion with drawings. In *2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 309–319, August 2003.

[65] I. Lim and D. Thalmann. Construction of animation models out of captured data. In *2002 IEEE International Conference on Multimedia and Expo*, pages 829–832, August 2002.

[66] Peter Litwinowicz and Lance Williams. Animating images with drawings. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 409–412, New York, NY, USA, 1994. ACM Press.

[67] Zicheng Liu, Steven J. Gortler, and Michael F. Cohen. Hierarchical spacetime control. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 35–42, July 1994.

[68] Mizuguchi M., Buchanan J., and T. Calvert. Data driven motion transitions for interactive games. In *Eurographics '01 - Short Presentation*, 2001.

[69] Nadia Magnenat-Thalmann and Daniel Thalmann. Virtual humans: thirty years of research, what next? *The Visual Computer*, 21(12):997–1015, 2005.

[70] Ramon Mas, Ronan Boulic, and Daniel Thalmann. Extended grasping behavior for autonomous human agents. In *AGENTS '97: Proceedings of the first international conference on Autonomous agents*, pages 494–495, New York, NY, USA, 1997. ACM Press.

[71] S. Ménardais, R. Kulpa, F. Multon, and B. Arnaldi. Synchronization for dynamic blending of motions. In *2004 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 325–335, July 2004.

[72] Brian Mirtich. Fast and accurate computation of polyhedral mass properties. *journal of graphics tools*, 1(2):31–50, 1996.

[73] Meinard Müller, Tido Röder, and Michael Clausen. Efficient content-based retrieval of motion capture data. *ACM Transactions on Graphics*, 24(3):677–685, August 2005.

[74] Franck Multon, Laure France, Marie-Paule Cani, and Gilles Debunne. Computer animation of human walking: a survey. *Journal of Visualization and Computer Animation (JVCA)*, 10:39–54, 1999. Published under the name Marie-Paule Cani-Gascuel.

[75] Luciana Porcher Nedel. Simulating virtual humans. Technical report, Universidade Federal do Rio Grande do Sul, 1998.

[76] J. Thomas Ngo and Joe Marks. Spacetime constraints revisited. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 343–350, New York, NY, USA, 1993. ACM Press.

[77] Richard E. Parent. Computer animation: Algorithms and techniques a historical review. In *Computer Animation 2000*, pages 86–91, May 2000.

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

[79] F. I. Parke. Parameterized models for facial animation. *IEEE Computer Graphics and Applications*, 2(9):61–68, 1982.

[80] Frederic I. Parke and Keith Waters. *Computer facial animation*. A. K. Peters, Ltd., Natick, MA, USA, 1996.

[81] Ken Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press.

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

[83] Ken Perlin. Two link inverse kinematics. GDC 2002 course notes `http://mrl.nyu.edu/~perlin/gdc/ik/talk.html`, March 2002.

[84] Ken Perlin and Athomas Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 205–216, August 1996.

[85] Zoran Popović and Andrew P. Witkin. Physically based motion transformation. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 11–20, August 1999.

[86] Martin Preston. Parallel spacetime animation. In Dimitri Terzopoulos and Daniel Thalmann, editors, *Computer Animation and Simulation '95*, pages 181–196. Springer-Verlag, 1995.

[87] Katherine Pullen and Christoph Bregler. Animating by multi-level sampling. In *Computer Animation 2000*, pages 36–42, May 2000.

[88] Katherine Pullen and Christoph Bregler. Motion capture assisted animation: Texturing and synthesis. *ACM Transactions on Graphics*, 21(3):501–508, July 2002.

[89] P. S. A. Reitsma and N. S. Pollard. Evaluating motion graphs for character navigation. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 89–98, New York, NY, USA, 2004. ACM Press.

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

[91] Charles Rose, Michael F. Cohen, and Bobby Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics & Applications*, 18(5):32–40, September - October 1998.

[92] Charles F. Rose, Brian Guenter, Bobby Bodenheimer, and Michael F. Cohen. Efficient generation of motion transitions using spacetime constraints. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 147–154, August 1996.

[93] R. M. Sanso and Daniel Thalmann. A hand control and automatic grasping system for synthetic actors. *Comput. Graph. Forum*, 13(3):167–177, 1994.

[94] Hyun Joon Shin, Lucas Kovar, and Michael Gleicher. Physical touch-up of human motions. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 194, Washington, DC, USA, 2003. IEEE Computer Society.

[95] Hyun Joon Shin, Jehee Lee, Michael Gleicher, and Sung Yong Shin. Computer puppetry: An importance-based approach. *ACM Transactions on Graphics*, 20(2):67–94, April 2001.

[96] Ken Shoemake. Euler angle conversion. In Paul Heckbert, editor, *Graphics Gems IV*, pages 222–229. Academic Press, Boston, 1994.

[97] Shanbhag Shrinath and Chandran Sharat. Parallel action synthesis for skeletally animated characters. Technical report, IIT Bombay, February 2004.

[98] Karl Sims. Evolving virtual creatures. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 15–22, July 1994.

[99] Russell Smith. *Intelligent Motion Control with an Artificial Cerebellum*. PhD thesis, University of Auckland, New Zealand, 1998.

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

[101] Harold C. Sun and Dimitris N. Metaxas. Automating gait animation. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 261–270, August 2001.

[102] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 205–214, New York, NY, USA, 1987. ACM Press.

[103] Daniel Thalmann. Physical, behavioral, and sensor-based animation. In *Proceedings of Graphicon 96*. Computer Graphics International Conference, 1996.

[104] Daniel Thalmann. Challenges for the research in virtual humans. In *Agents 2000*, 2000.

[105] N. M. Thalmann, S. Carion, M. Courchesne, P. Volino, and Y. Wu. Virtual clothes, hair and skin for beautiful top models. In *CGI '96: Proceedings of the 1996 Conference on Computer Graphics International*, page 132, Washington, DC, USA, 1996. IEEE Computer Society.

[106] Frank Thomas and Ollie Johnston. *Disney Animation – The Illusion of Life.* Abbeville Press, 1981.

[107] Deepak Tolani, Ambarish Goswami, and Norman I. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graph. Models Image Process.*, 62(5):353–388, 2000.

[108] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 43–50, July 1994.

[109] Munetoshi Unuma, Ken Anjyo, and Ryozo Takeuchi. Fourier principles for emotion-based human figure animation. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 91–96, August 1995.

[110] M. van de Panne. From footprints to animation. *Computer Graphics Forum*, 16(4):211–224, 1997.

[111] Michiel van de Panne. Making them move: Motor control for animated humans and animals. In *European Control Conference*, 1997.

[112] Jing Wang and Bobby Bodenheimer. Computing the duration of motion transitions: an empirical approach. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 335–344, New York, NY, USA, 2004. ACM Press.

[113] Jue Wang, Steven M. Drucker, Maneesh Agrawala, and Michael F. Cohen. The cartoon animation filter. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1169–1173, New York, NY, USA, 2006. ACM Press.

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

[115] Chris Wellman. *Inverse Kinematics And Geometric Constraints For Articulated Figure Manipulation*. PhD thesis, Simon Fraser University, 1989.

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

[117] Peter Jason Willemsen. *Behavior and Scenario Modelling for Real-Time Virtual Environments*. PhD thesis, Graduate College of The University of Iowa, May 2000.

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

[119] Andrew P. Witkin and Zoran Popović. Motion warping. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 105–108, August 1995.

[120] Wayne L. Wooten and Jessica K. Hodgins. Animation of human diving. *Comput. Graph. Forum*, 15(1):3–13, 1996.

[121] Qinxin Yu and Demetri Terzopoulos. Synthetic motion capture: Implementing an interactive virtual marine world. *The Visual Computer*, 15(7/8):377–394, 1999.

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

[123] Yu Zhang, Edmond C. Prakash, and Eric Sung. Hierarchical face modeling and fast 3d facial expression synthesis. In *SIBGRAPI '02: Proceedings of the 15th Brazilian Symposium on Computer Graphics and Image Processing*, pages 357–364, Washington, DC, USA, 2002. IEEE Computer Society.

[124] Victor B. Zordan and Jessica K. Hodgins. Motion capture-driven simulations that hit and react. In *ACM SIGGRAPH Symposium on Computer Animation*, pages 89–96, July 2002.

# Publications


[1] Shrinath Shanbhag, Sharat Chandran. Synthesizing New Walk and Climb Motions from a Single Motion Captured Walk Sequence. In *Proc. of Indian Conference in Vision Graphics and Image Processing (ICVGIP'04)*, Calcutta, 16-18 December 2004.

[2] Suddha Basu, Shrinath Shanbhag, Sharat Chandran. Search and Transitioning for Motion Capture Sequences. In *Proc. of ACM Symposium on Virtual Reality Software and Technology (VRST'05)*, Monterey, California, November 7-9, 2005

[3] Shrinath Shanbhag and Sharat Chandran. Grafting Locomotive Motions. In *The 14-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2006 (WSCG'06)*, Plzen, Czech Republic, January 30 - February 3, 2006.

# Acknowledgements

My sincerest thanks and gratitude to the almighty for showing me this beautiful day.

I set off on this journey seven years ago, little knowing what lay ahead. The process of earning a PhD is wonderfully amorphous and tailored to the individual. My experience, luckily, has been largely positive. The process taught me many lessons in life, changed my way of perceiving the world and ensured that I learned the art of research. Economic necessities required me to hold full time jobs all through this period. I worked at the graphics division of the National Centre for Software Technology (now CDAC Mumbai) for the first six years and at Microsoft India (R& D) Pvt. Ltd. in my final year. This meant that I was living in two worlds - the pragmatic and the pure – simultaneously. While it worked for me, this is not something I would recommend to the young researcher setting out today. I feel it is best to devote ones full time to study. In hindsight, I have thoroughly enjoyed the highs and lows that came my way, though it felt different at that time. None of this would have been possible without the cooperation and assistance of the people who were associated with me as I travelled along this path.

I am grateful to my advisor Prof. Sharat Chandran for guiding me. I thank him for his patience and for his words of encouragement he provided as I stumbled and fell innumerable times along the way. I thank Dr. Sudhir P. Mudur my ex-supervisor at National Centre for Software Technology (NCST - now CDAC Mumbai) for initiating me into computer graphics research. But for him, I would never have undertaken this journey. I thank Dr. D. B. Pathak for the motivating evening chat we had, when I first went to KRESIT to submit my PhD application form. I thank Prof. Kavi Arya and Prof. M. Karunakaran, members of my research progress committee, for their invaluable suggestions and constant encouragement.

The data used for experimentation was obtained from the CMU Graphics Lab Motion Capture Database (mocap.cs.cmu.edu). I thank them for making high quality mocap data freely available

for download.

I thoroughly enjoyed working at the VIGIL lab, IIT Mumbai. I did not have any seniors around but the enthusiasm and zeal of my juniors kept me going. I thank Biswarup Choudhury for the wonderful person he his and for lending his voice to my videos. I thank Appu Shaji for patiently answering my many questions on all things math. Thanks are also due to all lab members for making VIGIL the cool place it is.

I wish to acknowledge the support I have received from NCST and Microsoft India. I thank all my ex-colleagues at the Graphics Division of NCST for their encouragement and support. I thank Shamik Basu, my supervisor at Microsoft for allowing me to take time off from my job even during product release. I thank my colleagues from the MSSub and IDC IDSR teams at Microsoft for their invaluable cooperation. Without their support my last mile would have been in jeopardy.

I thank Vijay Ambre and Homcy Varghese from KRESIT office for being extremely helpful. They helped me accomplish all administrative tasks from far. Without their help it would have been very difficult for me to manage the two worlds that I have lived in, these last few years.

I thank all those who have been a source of inspiration, have helped me and have brought a smile to my face.

Finally, I wish to thank my parents for being the constants in my life. The have held me together in my times of despair and have endured my many tantrums from childhood till this day. This effort would not have been possible without their unconditional love and support. I dedicate this thesis to them.

Shrinath Shanbhag

January 15, 2007