Believable Automatically Synthesized Motion by Knowledge-Enhanced Motion Transformation

F. Sebastian Grassia

August 21, 2000

CMU-CS-00-163

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:

Andrew Witkin (chair)

Joseph Bates

Randy Pausch

Edwin Catmull, Pixar Animation Studios

This research was sponsored by a National Science Foundation (NSF) Graduate Fellowship and a Schlumberger Foundation Collegiate Award Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, Schlumberger, or the U.S. government.

Keywords: computer animation, automatic motion synthesis, motion editing, motion transformation, knowledge based animation, believable character animation

Abstract

Automatic synthesis of character animation promises to put the power of professional animators in the hands of everyday would-be filmmakers, and to enable rich behaviors and personalities in 3D virtual worlds. This goal has many difficult sub-problems, including how to generate primitive motion for any class of action (run, jump, sigh, etc.) satisfying any goals and in any style (e.g. sad, hurried, like George walks, etc.), how to parameterize these actions at a high level while allowing detailed modifications to the result, and how to combine the primitive motions to create coherent sequences and combinations of actions. Previous approaches to automatic motion synthesis generally appeal to some combination of physics simulation and robotic control to generate motion from a high-level description. In addition to being generally difficult to implement, these algorithms are limited to producing styles that can be expressed numerically in terms of physical quantities.

In this thesis we develop a new automatic synthesis algorithm based on motion transformation, which produces new motions by combining and/or deforming existing motions. Current motion transformation techniques are low to mid level tools that are limited in the range and/or precision of deformations they can make to a motion or groups of motions. We believe these limitations follow from treating the motions as largely unstructured collections of signals. Consequently, the first contribution of our work is to create a powerful, general motion transformation algorithm that combines the strengths of previous techniques by structuring input motions in a way that allows us to combine the effects of several transformation techniques.

To utilize this algorithm in an automatic setting, we must be able to encapsulate control rules in primitive motion generators. We accomplish this by developing the "motion model," which contains rules for transforming sets of example motions for a specific class of action. We show that because the example motions already contain detailed information about the action, the rules can be formulated on general properties of the action, such as targets/goals, rather than low-level properties such as muscle activations. This not only makes the rules relatively easy to devise, but allows a single motion model to generate motion in any style for which we can provide a few example motions. In the course of our experimentation we developed fifteen different motion models for humanoid character animation, several of which possess multiple styles (mainly derived from motion captured data).

After developing motion models, we continue to utilize knowledge encapsulation to address the problems of combining the output of motion models sequentially (segueing) and simultaneously (layering), collectively known as "transitioning." Because of the action-specific knowledge we store in motion models, we are able to create much richer and higher quality transitions than past approaches.

Our current results enable us to animate coherent stories, iteratively refining our initial directorial-like specification, in near-real-time. Some of our results can be viewed at http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter8 .

Dedication

To my parents, Frank and Frances Grassia

Acknowledgements

This work would not have been possible without the help and support of many people. I would especially like to thank the members of my committee. I feel amazingly lucky to have had the opportunity to work with them. In particular, my advisor Andy Witkin helped me develop the mathematical tools and outlook necessary to address hard animation problems. His ever-flowing fountain of good ideas and fresh perspectives helped me over many a rough spot. Finally, he taught me how to communicate mathematical material clearly and effectively. I wish to think Joe Bates for the opportunity to develop many of my ideas in collaboration with Zoesis and all the wonderful people there. The experience of working closely with Joe and Zoesis's group of focussed, kindred souls on challenging graphics problems was one of the most rewarding of my academic career. I would like to thank Randy Pausch not only for his contributions in evaluating the results of this work, but also for bringing amazing and desperately needed organization to the thesis and my professional life, for pushing me to establish effective lines of communication to my committee when I needed them more than I realized, and for being an all-around inspiration for getting things done. I wish to thank Ed Catmull for sharing some of his amazing insight into the nature of the problems I attacked and the solutions for which I was looking. Our conversations early in the development of the thesis influenced all of the subsequent choices that determined the direction of the work.

I also wish to thank my family and friends, whose understanding, love, and belief enabled me to persist through the low spots in the seven year journey that is finally concluding, and for sharing with me the joys along the way and at the culmination.

Table of Contents

1 Introduction

Current computer technology has reached a level that allows a team of professionals to create virtually any special effect, digital character, or virtual world they can imagine. In particular, the massive success of movies such as *Toy Story* (1995) and *Star Wars: The Phantom Menace* (1999) demonstrates that we can create, animate, and render believable digital characters that captivate an audience throughout a feature-length film. However, the effort and skill required to do so are immense, requiring thousands of person-hours from artists, animators, and engineers.

Now that we have the technology to make it *possible* to bring digital characters to life, the next technological frontier is making the process *easier*. In this thesis we will focus on simplifying the process of animating digital characters. The difficulty of character animation stems from both the amount of specification required to make characters move – the skeletal pose alone of a humanoid character (excluding hands and face) requires 50 to 60 scalar values – and the deep understanding of the way people and animals move that is necessary to create animation specifications that result in pleasing, believable motions.

Graphics researchers have borrowed, adapted, or developed many techniques that try to ease these difficulties by either reducing or abstracting the specification required to pose a character, or by encapsulating physical laws of motion or other knowledge into algorithms. For instance, *inverse kinematics* (IK) [Badler

1987][Isaacs 1987] abstracts the specification needed for posing, by allowing the animator to place *constraints* on a pose (such as "the foot goes here and the hand goes over there") and relying on the computer to figure out what combinations of the 50 or so skeletal joint angles will satisfy the constraints. IK does not, however, help much at making characters move believably, since it is still entirely up to the animator to determine where the foot should move and how quickly it should get there.

Other methods attack the ambitious goal of *automatic motion synthesis*, in which the computer generates the entire animation from a high-level specification, such as a director might give to actors. Any such approach must provide the computer with some means of automatically choosing natural looking, believable motions from among all the possible motions that satisfy the high-level description. Most automatic approaches guide the computer with physical laws (such as Newton's laws) and dynamics, because all motions that real people and animals perform conform to physical laws, and because physics is straightforward to express in digital form. One approach builds virtual robotic controllers [Hodgins 1995] that afford high-level parameterizations such as a path and velocity for running, then numerically simulating the robot in a virtual world governed by physics to produce motion. Another approach casts animation as a large *spacetime* optimization [Witkin 1988], utilizing physics, a model of the character's actuator system (*i.e.* muscles), and a numerical measurement of the motion that measures its quality (such as power expenditure), to compute the *optimal* motion that satisfies a set of high-level constraints on the motion.

These and other, related approaches are potentially powerful, but share several limitations. First, they tend to be very difficult to program, and often do not generalize either to complicated characters or diverse actions. Second, they are extremely computationally expensive, and thus make iterative refinement of an animation difficult, and real-time response is out of the question. Lastly, and in our reasoning most limiting, is the adherence to strict physical realism: animators routinely violate physics in order to create more powerful personalities and draw the viewer's attention.

There is a class of motion synthesis algorithms that can reproduce an unlimited range of useful and interesting styles for any type of motion. Rather than producing new motions from square one (as do the automatic methods), these algorithms *transform* an existing motion to satisfy a different set of high-level constraints, while preserving the style of the original performance. There are currently a broad array of transformation tools, from those that apply simple, keyframe-like deformations [Witkin 1995], to those that adapt all or parts of the spacetime optimization to the task of deforming motion [Gleicher 1997][Popović 1999b], to producing new motions by interpolating between many example motions [Wiley 1997][Rose 1998]. These tools are very promising, but currently suffer from a variety of problems, ranging from inability to satisfy precise constraints, to limited range of operation (*i.e.* cannot deform the motions very far), to the inability to consider more than one style of a particular type of action at a time. We believe that these problems all stem from attaching little to no meaning to the motions upon which the transformation operators work.

In this thesis we develop a method for automatic character animation that combines motion transformation techniques with a structured means of attaching knowledge to primitive motion generators. Previous approaches to automatic motion synthesis are limited to producing motion in styles that can be described mathematically. By using motion transformation as the basis for our algorithm, we can automatically synthesize motion in any style for which we possess a few example performances. The single, guiding principle of all aspects of our work is that we can make automatic motion synthesis tractable by attaching expert knowledge to otherwise unstructured example motions. This knowledge takes the form of descriptive annotations on the example motions and rules for how to combine and transform example motions. Our approach consists of three main parts.

First, we develop a toolkit and methodology for motion transformation that combines the strengths of previous transformation operators and extends the range of deformations we can apply to any set of motions. We present updates to existing operators such as blending, space warping, and time warping, and also a novel optimization-based IK algorithm and constraint satisfaction engine based upon it. Our ability to improve and extend transformation methods hinges on attaching structured knowledge to the motions we transform.

Second, we develop the *motion model*, a new abstraction for encapsulating domain-specific rules relevant to a particular type of action (such as walking, jumping, peering, *etc.*). The rules specify how to transform a set of example or *base motions,* using our transformation toolkit, from a high-level goal specification (such as "jump from here to there, going so high"). We are able to reason about and manipulate gross aspects of the motion (such as goals and targets) only because we utilize transformation and example motions, which contain the specific details of the motion. Utilizing this information, we are also able to fill in any information not contained in a task specification, so a potential user can provide much or little detail on the constraints for the motion. Additionally, because the rules are formulated to extract information from the base motions, a single motion model can utilize an infinite number of sets of base motions, each representing a different style of performing the action.

Third, we develop algorithms for combining the primitive motions produced by motion models. The most basic motion combination is the *segue*, in which we transition from one performing motion into another. We present methods for calculating both the duration and the geometry of the segue motion, utilizing information gleaned from base motions and new metrics. We then consider other means of combining motions and present new techniques for layering motions, so that a character can perform multiple actions simultaneously, and pausing and holding poses while performing other actions.

Ultimately, we present a prototype animation system, *Shaker*, that incorporates these ideas and presents an interface in which a user builds an animation by adding primitive motions (*i.e.* motion models) to a developing script, and incrementally refining both the primitive motions and transitions using any combination of the high and low level tools we provide.

1.1 Motivation & Goals

It is clear that there are many benefits to be gained from making believable character animation easier to create. The work in this thesis is motivated primarily by three specific applications:

- **Autonomous/Guided Agents.** Animating computer or user controlled digital characters in real-time is perhaps the “poster child” application for good automatic animation methods, since it is simply not possible for an animator to animate a character (much less dozens of them) in real-time in ever-changing situations. Given the absolute necessity of maintaining a minimum frame-rate, quality is often a secondary concern in this area. We would like to eliminate this trend.
- **Visual Storytelling.** One of the great promises of the digital revolution is to let people tap into creativity that previously was blocked off for artificial and/or economic reasons. The camcorder helped to make amateur filmmaking/storytelling more accessible, but still left the would-be filmmaker limited by what she could place in front of her camcorder and what she could communicate to her actors. Automatic animation methods should enable even an amateur to animate her own stories with her own characters in a visual medium.
- **Animation Prototyping.** Professional animators are artists, and as such demand complete control over their medium. Since automatic methods generally tend to reduce detailed control over the end result, it is more difficult to make a case for their application to a production environment. If, as we hope, we can successfully integrate detailed controls with high-level specification, our methods might make animators more productive by quickly giving them a rough prototype of an animation, allowing them to focus most of their time on tweaking it to get it just right.

Examining these applications in detail, we have identified the following five sub-problems of automatic motion synthesis that we must address, which illustrate the complexity and scope of our goal:

1. For any particular class of motion (*eg.* walking, jumping, yawning), we must be able to achieve believability across a wide range of stylistic and functional variation. Functional variation deals with, for example, walking along any path, over any terrain, stepping over specific obstacles, *etc.*, while stylistic variation specifies whether the character is strutting, sneaking, or walking normally, or walking like Jack walks or Sandra walks. Note that while such an *atomization* of motion into primitive actions is common in graphics research, its validity is not universally accepted [Sengers 1998]. We hope to provide more evidence that, when accompanied by rich operators for combining motions, atomizing them is valid.
2. To maximize ease of use without sacrificing flexibility, we must provide a small, relevant set of control parameters for each class of motion (*eg.* controlling a jump by its takeoff and landing spots, height, and directions in which the character is facing upon beginning and ending the jump), while

still allowing very detailed adjustments to the motion, if so desired. Previous approaches almost invariably contain a tradeoff between the ease of control and amount of control.

3. Generating primitive motions of arbitrary classes is only half of our task, since nearly all interesting tasks and stories involve not just a single action, but sequences and combinations of actions. Therefore we must also be able to generate transitions between motions that are both believable and stylistically consistent with the surrounding motions.

4. Most animation tasks will also involve interaction among characters and interaction between characters and their environments. Our motion synthesis algorithms must be aware of the eventuality of interaction and account for it.

5. Since our fundamental tool is motion transformation, then unless we mean to restrict the user to only those styles of motion we initially provide, we must ensure that the system is straightforwardly extensible – it must be able to incorporate new example motions easily, and (with understandably more effort) entirely new classes of action.

While solving these problems, we will also try to remain cognizant of two further, secondary goals. First is that computational speed is always important, particularly for the real-time applications above; we consider this secondary because continual hardware advances give us a doubling of speed every one and one half years. Second is that to enable effective, powerful storytelling we must, as Loyall points out, "allow for the unique" [Loyall 1997], which is to say that we must be able to endow characters with identifiably unique personalities, and that no character should ever perform the same action in exactly the same way. We will address this concern to the extent that we solve problem 5 above, but we are aware that ultimately we must incorporate variation *within* styles as well.

Finally, before we move on to discussing our solution to these problems, we wish to comment on the term "believable," which appears not only in the title of this thesis, but throughout the document as well. We are interested in generating *believable* character animation, as opposed to strictly *realistic* animation. "Believable" here has the same meaning proposed by Bates in his term "believable agent" [Bates 1994] – that although we know the animated character is not real, we are willing to suspend our disbelief and invest emotionally in the character. Whereas realistic animation is defined solely by physics and biomechanics, believable animation, while generally adhering to a consistent set of rules, may not follow "true physics" because devices such as exaggeration and highly stylized behaviors serve to create more engaging characters (just as good acting involves much more than simply simulating real people). As we will see later, our experimental domain is human motion capture data, and one of our evaluation criteria is realism, but these decisions were purely pragmatic, in that it was much easier to obtain motion captured data than quality 3D hand-animation. Our methods are limited neither to humanoid animation nor to "earth physics."

1.2 Algorithm Outline

Solving the problems we have described represents a complete solution to the problem of synthesizing rich, involved character animation automatically from high-level specification. We believe that no single, neat algorithm can ever hope to address the problem in its entirety. Our approach is based on a handful of concepts and algorithms, unified by the incorporation of expert knowledge at all levels. In this section we will broadly describe the components of this approach, beginning at the bottom, with the process of creating a better motion transformation algorithm. We will then move on to describe how we create modular, automatic, primitive motion generators based on that algorithm. Finally, we will discuss how we transition between and combine the motions created by the primitive motion generators.

We begin with a primitive motion that we wish to transform to meet a new set of goals. We observe that for actions with a wide task domain (such as throwing, reaching, and jumping), no existing deformation tool can convincingly transform a single motion over the entire domain. Using blending/interpolation methods, however, we can mix *multiple* example motions together – by sampling the domain space with multiple examples we can produce reasonable intermediate motions by temporally aligning and blending the examples. Therefore our first step is to choose a set of *base motions* that span the domain of the action in which we are interested, and annotate them so that we can align them. In contrast to prior approaches, we are able to allow a great deal of flexibility in choosing the base motions, because of the added structure we attach to them (described below).

Interpolation methods excel at reproducing a wide range of task domain variation, but falter at meeting precise goals (such as "pick up the ball at $<x, y, z>$ coordinates"). Therefore we utilize motion interpolation to compute a motion that lies in the general neighborhood of our goal, then apply a combination of motion deformation operators to enforce precise, instantaneous pose constraints on the motion. Chief among these tools is *motion warping* [Witkin 1995], which deforms a motion by adding a smooth, time-varying offset to the motion of each joint such that at certain, specified keyframes, the sum of the joints' original values (in the original motion) plus the offsets results in a specific pose that meets the desired constraints. Based on the supposition that much of what determines *style* is a combination of timing and high-frequency content in a motion, motion-warping tends to preserve the style of the motion it is deforming (for small magnitude deformations), because it does not alter the original timing, and the smoothness of the offset does not perturb the high-frequency content of the motion.

This new two-step transformation process combines the strengths of each of the individual components. It does not, however, address the residual weakness of both components, which is that they generally upset important, time-persistent geometric qualities of the original motions. For instance, blending a long-stride walk with a short-stride walk will indeed produce a medium-stride walk, but one in which the feet slide along and/or penetrate the ground rather than remaining firmly planted, as they were in each example motion. In order to correct such problems, we need a means of specifying and enforcing these qualities. To do this we introduce the concept of geometric *invariants*, which specify persistent geometric constraints on the motion that must be maintained under any deformation. In this role (we will introduce others later) we use invariants to deform the motion further by filtering it through a new, fast, constraint satisfaction engine that ensures all the invariants are maintained throughout the motion. This engine attempts to preserve the style of its input motion by satisfying the invariants via IK and constrained optimization, which results in a motion that deviates minimally from the input according to a criterion we will describe in chapter 4.

What we have described so far represents a general, flexible motion transformation algorithm that an animator could use to produce an abundance of motions from a small set of examples. However, we are interested in *automatic* motion synthesis, so we need a means of guiding this algorithm much as an animator would. We develop the *motion model* as a fundamental abstraction of a particular class of primitive motion. The motion model for a particular action class (*e.g.* jumping, pointing, throwing, *etc.*) encapsulates the definition of the base motions[1] for the action, all of the invariants, an appropriate high-level parameterization of the action, and rules for guiding the transformation algorithm on the base motions to produce a motion that satisfies any parameter setting. All of the knowledge encoded in a motion model is general to the class of action – for instance, the "jump" motion model specifies that there are the named events "liftoff" and "touchdown," and that in-between these events, the feet are not fixed to the ground, and the character's center of mass follows a ballistic trajectory. However, it does *not* specify when, precisely, these events should occur, nor does it specify where the feet are located prior to and subsequent to the airborne phase. The motion model gathers this specific information from its base motions and their annotations. This means that we can achieve different *styles* of performance simply by swapping one set of base motions for another; while motion model definitions require a programming language, styles are simple, annotated text files, and we can change styles dynamically while continuing to satisfy the goals contained in the motion model's parameters.

Although we created more than a dozen complete motion models in this thesis, the focus and main contribution of the work was not in producing any particular motion model, but rather developing a structured approach to creating them, which we will describe in chapter 5.

[1] The definitions of the base motions describe in general terms the goal that each base motion should achieve. For instance, the definitions of the four base motions of our *jump* motion model are "broad jump as far as possible," "short hop upwards," "jump as high straight up as possible," "jump up onto a raised platform."

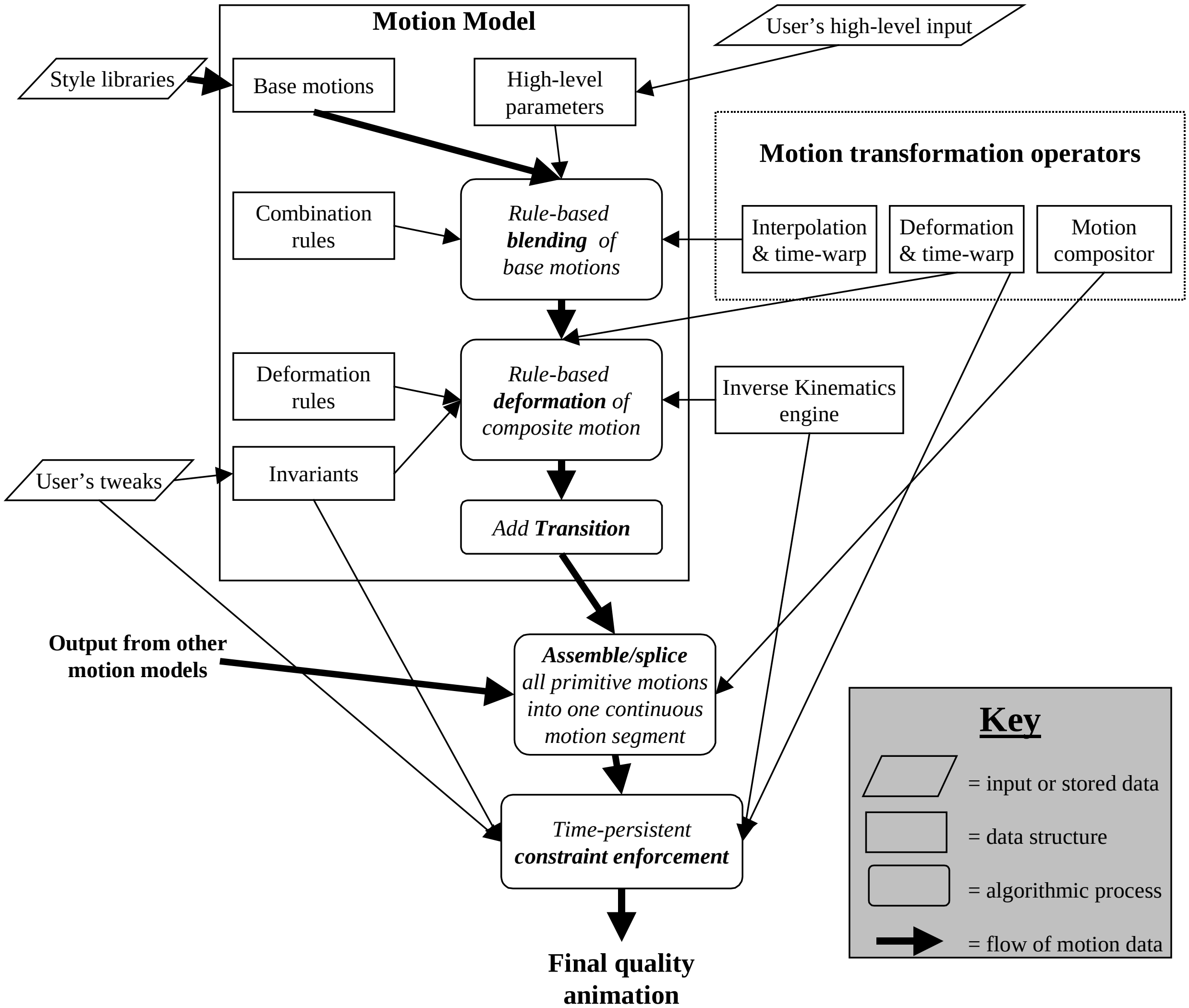


Figure 1-1: Structure and algorithm organization.

Interesting stories do not, however, consist of single actions, but rather sequences and combinations of motions. We therefore continue to use the information inside motion models to create "transition segments" that last from when one action has completed its task to when the next action achieves its first goal (as defined by the respective motion models). Although we cannot, in general, utilize base motions directly (since the starting pose for a transition may be arbitrarily different from the starting pose of the base motions), we develop a computation for determining the duration of the transition by extrapolating from the base motions, and propose a method for capturing the style of the base motions for the transition by means of spectral decomposition of the base motions.

Utilizing the same invariants we introduced above, we are also able to develop a richer concept of layering motions simultaneously than is found in previous work. We present techniques for blended and differential layering, in addition to the usual "replace" layering, along with a new data structure based on the concept of a “motion stack” that, along with invariants, is necessary for general layering. The motion stack allows us to combine the signals that comprise the actual motions in new ways, and invariants not only reprise their role in assuring that critical aspects of the blended motions survive the combination, but also provide guidance as to *when* specific motions can actually be combined. The motion stack data structure also allows us to provide a new type of motion transition, a “hold,” in which an action executes while trying to hold the terminating pose of the previous action.

This leads to the specific algorithms and structures that are represented schematically in Figure 1-1. Each motion model is designed by an expert using the API we have developed, and consists of base motions, motion combination and deformation rules, geometric invariants, and a high-level parameterization. The API includes a language for easily building motion expressions of transformed motions (operators include blending, time and space warping, segmenting, splicing, and others), invariants and other abstractions for dealing with footsteps, balance and interaction[2], and tools for extracting information from base motions and computing adjusted poses using IK.

The combination rules specify how the base motions should be combined to create a motion that approximated the goal represented by the high-level parameters (for example, throwing a ball at a specified target). If the designer chose a simple, dense set of base motions, this can be as simple as a radial basis function interpolation scheme such as that described by Rose [Rose 1998]. If the designer chose a more compact (and thus more efficient) set of base motions, the combination rules will execute some form of geometric reasoning on the parameter settings to determine a good combination. For example, given the four throwing base motions *throw-near, throw-far, throw-up,* and *throw-down,* the *throw* motion model first blends the *throw-near* and *throw-far* motions according to the distance from the thrower to the target (accounting also for the desired throwing velocity), and then blends into the result either *throw-up* or *throw-down* depending on the angle of elevation of the target.

The deformation rules specify a set of deformations to apply to the resulting motion that will cause the motion to meet important instantaneous goals of the motion. The result of the previous motion combination is an expression in our API’s motion language, and the result of the further deformations will also be a motion expression. Since the combination should have created a motion that is not too different from the desired motion, the motion model can utilize simple deformation operators such as motion warping, pivoting, and reflecting, using IK and the motion model’s geometric invariants to compute goal poses. Invariants for a *throw* include releasing the ball in the proper direction, gazing at the target, and ensuring the character’s feet do not slip throughout. In our throwing example, we deform the combination motion by pivoting the motion partway through the action so that the character steps into the lateral direction of the target, thus releasing the ball in the correct lateral direction.

The simple deformation operators utilized in the last step are incapable of maintaining persistent invariants (such as feet not slipping), so the final result of the motion model must be filtered through a constraint enforcement engine that enforces the motion model's own invariants. Several such engines exist [Gleicher 1997, Lee 1999], but we have developed a new, faster algorithm that satisfies the constraints in a single pass of IK, pre-filtering the motion with motion-warped poses at all constraint activations/deactivations to preserve continuity in the resultant motion. Before we can perform this constraint filtering, however, we must combine the contributions from all motion models into a single, seamless animation. One element of the API (called a *clip-composite*) enables individual motion segments to be glued together in sequence and layered one on top of another. Making this composite seamless requires that we possess smooth, natural transitions between motions.

The last, self-contained aspect of the API is a transition generator, integrated into the motion language, so that a transition between two motions is simply a motion expression that can be inserted into the clip-composite between the two. The transition generator queries each motion model to determine when best to end the preceding motion and when to enter the subsequent motion, which results in a beginning pose (plus instantaneous velocity) and an ending pose (plus velocity). We have developed an algorithm that uses a mass-based measure of the distance between two poses and information from the motion models and their motions to determine how long the transition between the two poses should last. The actual motion performed during the transition should be in a style consistent with those of the surrounding motions. We have adapted motion warping to this purpose, but it is only applicable in certain situations. In other cases, we currently use simple Hermite interpolation of the poses, which ignores all high-frequency content of the style. We propose a method based on a Laplacian decomposition of motion that we hope to use to graft on the high-frequencies of the surrounding motions to the Hermite pose interpolation.

1.3 Contributions and Results

In this thesis we will show that by adding structure and domain-specific knowledge to motion transformation, we are able to solve a number of problems in character animation:

- By encapsulating motion transformation rules in motion models, we significantly extend the range of motions we can produce by transforming example motions using relatively simple operators. Motion interpolation can cover a wide range of task-space variation, given the right base motions, but performs poorly at meeting and maintaining precise geometric constraints. Motion warping and other deformation operators can transform a motion to meet a precise set of new instantaneous constraints while preserving the high-frequency spectrum of the original motion, but are limited to a small range of deformations. A motion model designer is able to specify how to combine transfor-

[2] Our current API defines these only for bipeds, because the motion data on which we worked was entirely human mo-

mation operators to use each to its best advantage, according to general principles we will provide. Moreover, we define a compositional language for building motion expressions, in which motions and operators for combining and deforming them are functionally equivalent. Thus a motion model designer can easily compose rich transformations consisting of many deformations and combinations of raw base motions.

- The structure we apply to motions allows us to separate the common from the unique – motion models abstract the control structure and rules of motion for a particular action, while styles fit unique performances of the action into the more general structure. Consequently, our methods are the first that allow us to transform entire families of task-related, but stylistically different motions in the same way. Use of the same control structure (*viz.* motion model) means that we can maintain all of a user's high and low-level animation specification (goals/targets, tweaks, and all other parameter settings) while radically changing the style in which a character performs a motion.

- By adding geometric invariants that specify the crucial geometric properties of a primitive class of motion that must be preserved throughout transformation, we are able to integrate high-level parametric control of animation with very low-level fine tuning capabilities. In addition to their duties in transforming and combining primitive motions, these invariants allow the user to make arbitrary keyframe or space-warp modifications to the animation without perturbing its defining characteristics, since all modifications are filtered through the invariants.

We will also investigate transition synthesis and other motion combinations to unprecedented depth. This is the first work that considers computing not just the geometry of a transition, but also the transition duration. We will present a method that uses a mass-displacement measure of pose distance and the information contained in prototype transitions to compute transition durations that depend on both the magnitude of the transition and the style of the segued motions. We will demonstrate the superiority of this computation to existing methods, and suggest an extension to it that applies machine learning to improve its quality further. We will also explore new operators for combining motions, including the *differential layering* operator that allows us, for example, to perform any action while shivering (by only layering the high-frequencies from a shivering motion onto the action), and the *hold* operator, that causes an actor to perform an action while trying to maintain a given pose as much as possible.

Lastly, we will introduce a new constraint satisfaction technique that combines motion warping with a single pass of IK. It is able to satisfy constraints exactly (where the constraints do not conflict), and, given a comparable IK algorithm, is much faster than existing methods. Although it provides no guarantee of inter-frame coherence, we have never observed it to introduce discontinuities.

---

tion capture data. It should be straightforward to derive equivalent abstractions for quadrupeds, arthropods, *etc.*

To test our ideas and explore the significance of these contributions, we built a humanoid animation system called *Shaker* that allows a would-be animator to construct high-level scripts consisting of sequences and combinations of motion models. Shaker provides fifteen different actions (*i.e.* motion models): *Jump, Reach, Throw, Peer, Foot-shuffle, Ah-ha!, Arms-akimbo, Catch-breath, Check-watch, Fist-Shake, Foot-stomp, Head-shake, Relief-wipe, Sigh,* and *Shrug.* All of the base motions for our motion models came from motion captured data, and for many of the motion models, we possessed multiple styles of performance.

We used Shaker to create many small animations to test various concepts, and also several longer animations that tell simple short stories. In the online animation section for this chapter, http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter8**,** we present several examples of animations we created with Shaker; each is explained in detail in the online page. We also conducted an informal user test to evaluate the quality of animations we were able to create with Shaker. Employing a modified Turing test, we observed that users were only able to select a pure motion captured motion from among three synthesized motions 28% of the time. We will discuss these and all other results in greater detail in chapter 8.

1.4 Document Layout

In the next chapter we will explore past animation work in more detail and relate it to our own work. In chapter 3 we will present a detailed overview of our algorithms along with a deeper discussion of the animation problems we address. Next, chapter 4 describes all of the fundamental algorithms and tools (both new and adapted) that we utilize in our animation engine. Chapter 5 explains the heart of that engine, motion models, describing them structurally, operationally, and from a design perspective. In chapter 6 we proceed to discuss transitions and other types of motion combinations, building on the structure of the motion model. Chapter 7 presents the details of tying motion models together with transitions, interaction between motion models and between characters, and fitting the whole together with our constraint satisfaction algorithm. In chapter 8 we present our experimental results, including the prototype animation system we built to test our ideas, several animations demonstrative of the breadth and quality of animations we were able to achieve, the results of a user test in which subjects judged the quality of our animations, and potential limitations of our approach. Finally, in chapter 9 we conclude, re-examining our contributions, how well we achieved our goals, and several interesting directions for future work. We also include two appendices that give structural schematics of the major abstractions in the thesis.

2 Background

Researchers have been developing sophisticated and powerful algorithms for character animation for many years. Our own work relies on various of these approaches not only for inspiration, but also for some of our building blocks. In this chapter we will explore a range of previous work, beginning with the most fundamental (and most frequently used) techniques of keyframing and inverse kinematics, then examining a smattering of examples from each of three broad categories of algorithms: automatic motion synthesis, motion transformation, and motion combinations. At the end of the chapter we will consider the place of the work in this thesis with respect to the existing body of work.

Keyframe interpolation is one of the oldest techniques for character animation, and still de facto standard for serious work. Assuming that a character model can be posed by specifying a fixed set of numbers called *degrees of freedom* or DOFs (as we will describe in detail in chapter 4), keyframing enables the animator to pose a character at specified *keyframes* along the animation timeline, allowing the computer to generate the intermediate poses by interpolating the DOFs. The animator can place unlimited keyframes, and can also affect the inter-keyframe interpolation. Thus keyframing gives the animator as much control over the animation as desired, which largely accounts for its continued pre-eminence. It also requires the animator to exercise great skill in determining the precise timing and motion of each DOF, and great patience in dealing with a potentially enormous amount of specification (circa 2000 character models possess anywhere from fifty to hundreds of DOFs). For these reasons keyframing remains a tool usable only by those with a great deal of talent and time.

In the last twenty years, numerous formulations of *inverse kinematics* (IK) have been developed to make the posing subtask of keyframing easier [Isaacs 1987, Girard 1987, Badler 1987, Maciejewski 1990, Phillips 1990]. IK draws inspiration (and much mathematics) from robotic control algorithms, and allows the animator to directly specify positions and orientations of a character's bodyparts, letting the computer determine the values the DOFs should take to fulfill these constraints. Particularly for tasks that require precise placement of feet and hands, IK is dramatically more intuitive and efficient than direct manipulation of the DOFs. Simple systems involving only short, single chains of bodyparts can utilize closed-form analytical methods for computing the DOF values that will satisfy the position and/or orientation constraints; these algorithms are fast and stable, although obviously limited. For the more general case of multiple, interacting constraints on characters with branching structure we must utilize more sophisticated mathematics, such as numerical optimization [Gill 1981, Press 1993]. IK algorithms that use such mathematics are powerful and flexible, but notoriously difficult to make robust. As IK is an important component of our approach, we present our own formulation in chapter 4; we have also discussed the relative merits of several different formulations elsewhere [Grassia 2000].

2.1 Approaching Automatic Motion Synthesis

While IK eases the task of posing characters (and can even be used to animate them by pulling bodyparts along trajectories through time), it relieves the animator neither of the need to specify the subtle timing relationships of various bodypart movements, nor of the need to control all bodyparts all the time. The goal of so-called *automatic motion synthesis* techniques is to enable the animator to give a very high-level description of a motion or set of motions, allowing the computer to handle all of the low-level details such as precise body movements and timing. In this section we will examine several different approaches to this goal.

2.1.1 Procedural and Robotic Controllers

Following the most basic rule of thumb in computer science, "divide and conquer," we can break down the task of creating algorithms for the production of arbitrary animations into the task of making many algorithms, each of which can produce a *specific* type of motion (such as walking, reaching, jumping, *etc.*) and worry about combining their results later. Each of these smaller algorithms becomes a robotic *motion controller*, possessing some suitable high-level parameterization that enables an animator to produce motion for a range of situations. There have been many architectures proposed for constructing robotic motion controllers, which can be broken down into those that incorporate dynamics [Bruderlin 1989, McKenna

1990, Hodgins 1995, van de Panne 1996] and those that are purely kinematic [Ko 1993, Bruderlin 1994, Perlin 1995].

Dynamics-based controllers promise physical realism at the cost of long execution times, since they must invoke a physical simulation to create an animation. Hodgins, for example, constructs a finite state machine that guides a robotic controller during a repeated action, such as running. In each state, proportional derivative (PD) controllers drive the robot/character's joints towards a particular pose, which characterizes the state (such as the right foot push-off phase of a run cycle). The controllers function as muscles for the character (which possesses a realistic human mass distribution), and obey muscle activation limits. The motion produced by the controller is a function of both the poses associated with the states and the detailed activation functions for the PD controllers in each state; these must both be adjusted by hand to make the character maintain its balance and respond to parameters (such as turning radius). While professional animators can provide input to generating the poses that drive the state machines, it is very difficult to apply their skills to tuning PD controllers to achieve stability, much less incorporate stylistic variation.

Kinematic controllers execute rapidly, and are possibly easier for an animator to control, since they do not function on muscle activations and inertia, but rather strictly on poses. Perlin synthesizes motion curves for each of a character's joints by summing textural noise with sine and cosine functions. The global position of a character is determined by defining the pose relative to whichever of the feet happens to be in contact with the ground. Perlin created several interesting and expressive dance steps using this approach, but his are not true parameterized controllers, so the end user must specify animations in terms of sines and cosines.

As we have hinted, all of these controller schemes suffer from two shortcomings. First, they are very difficult to create. While several of the approaches provide "general" schemes for constructing new motion controllers, these schemes are generally founded on mathematics or control theory that is all but inaccessible to professional animators, who are the group that almost certainly should be creating the motion controllers, as they possess unequaled knowledge about how things should move. Second, it is without exception difficult if not impossible to incorporate stylistic variation into the output of such controllers.

Our motion models can be considered kinematic motion controllers. We address the problems above by using structured motion transformation as the basis for our controllers. Because example motions already contain all of the DOF functions that would otherwise require muscle activation schedules or arbitrary procedural generators, professional animators building motion models can focus on *altering* existing motions to create new ones, a task with which they are well acquainted. Because we attach meaning and structure to the example motions that describe the motion and its goals in general terms, we can allow many styles in a single motion controller simply by adding different example motions.

2.1.2 Simulation and Physics-Based Optimization

Drawing inspiration directly from nature, researchers have attempted to faithfully simulate physics in order to produce highly realistic animation from minimal specification. Simulation techniques have successfully animated interacting rigid bodies [Moore 1988, Baraff 1989, Baraff 1991] and clothing [Baraff 1998, DeRose 1998]. Straight simulation involves representing a system/character/world as a set of geometrical objects (polygonal, implicit, particles) that possesses position, orientation, and linear and angular velocities. The animator gives the objects initial positions and velocities, and specifies some forces to act on them (gravity, wind, springs, *etc.*). The simulation algorithm then applies Newton's laws to incrementally simulate forward in time how the objects would move in response to the forces, generally accounting for collisions, and possibly friction as well. This process works well when the objects being simulated have no actuators or motivators of their own; when muscles become involved (as they do for character animation), simulation algorithms need some other entity to tell them when and how the muscles should activate, and are therefore not truly applicable to character animation by themselves. Simulation techniques are also difficult to control, because one cannot in general specify the goal state one would like the system to achieve, but rather only the starting state.

One approach to addressing these problems is *spacetime constraints* [Witkin 1988, Cohen 1992]. This technique allows the animator to specify an arbitrary number of full or partial pose constraints (*i.e.* keyframes) through which the character will pass while executing a motion that is dynamically sound (according to the mass distribution and muscle activation limits that must accompany the character). Spacetime computes the muscle activations over the entire animation simultaneously, by solving a large, constrained optimization. In this optimization appear the constraints from the poses the animator specified, the muscle activation limits, and Newton's second law expressed in terms of the muscle activations for which the algorithm is solving. If there is a solution to this set of constraints, there are typically many, so the algorithm incorporates a numerical *objective function* that is simultaneously minimized as the constraints are satisfied. One can thus guide the solution by varying the objective function; an objective function that is used often is power consumption, so that the character tends to exert himself as little as possible to accomplish the motion. The presence of the physics constraints in the formulation provides significant power in the traditionally difficult area of producing athletic motions, since such motions are governed primarily by physics. However, spacetime encounters problems in scaling to characters of humanoid complexity, and allows only for stylistic variation that can be encoded numerically in the objective function.

In contrast, our work virtually ignores physics, relying instead on the realism of the example motions it uses and the ability of our transformation operators to preserve that realism. Thus, while athletic motions may be the most difficult for us to handle, we are able to easily reproduce motions that are defined and driven more by culture, emotion, or kinematics (motion involving little energy transfer). Also, we are able to produce exaggerated, flamboyant, or other far-fetched, non-realistic motions simply by using example motions that contain those properties, whereas spacetime would need to determine some way of encoding these properties numerically in an objective function.

2.1.3 Learning and Other High-Dimensional Searches

Spacetime optimization performs a localized optimization, *i.e.* it looks for a suitable animation that is in the neighborhood (in terms of the values of all of its free parameters) of some initial starting animation. Other researchers have endeavored to broaden the range of animations that can be automatically produced for a given motion by performing a global optimization or search. Both the genetic algorithm [Ngo 1993, Gritz 1997] and simulated annealing [Auslander 1995, Grzeszczuk 1995] have been utilized to learn/select controllers for locomotion of various simple characters. Approaches to date have been limited to animating locomotion, largely due to the necessity for an evaluation function that can automatically determine how well a particular controller can accomplish its goal. Sims [Sims 1994] took the analogy to evolution one step further, and evolved the structure of the characters themselves.

These techniques operate on the premise that in a very high dimensional space of numerical encodings of animation controllers, good or desirable controllers reside in small, localized families, and families may be scattered across a large area. To discover these families automatically, one uses a global search algorithm (such as the genetic algorithm or simulated annealing) that begins with a randomly assembled population of controllers (each encoded in a dense, numeric vector). The algorithm generates potential new generations of populations by mutating and/or recombining the current population's members. Each new controller and the old controllers are then evaluated according to how well it accomplishes its task (*e.g.* how far a locomotion controller moves its character), and only the best performing are kept for the next generation. This process continues until the generations cease to improve over a number of generations, or computation resources are exhausted.

Due to the high dimensionality of the typical search space, these techniques incur massive computation times. Although the searches theoretically pass through all dimensions of stylistic variation as well as competence at performing the given task, we have no means of encoding style into the evaluation functions. The only means currently available for achieving useful stylistic variation is via regular animator intervention throughout the optimization process: instead of relying solely on the evaluation function to guide the direction of the search, the animator manually chooses interesting directions from those currently being explored by the algorithm. Furthermore, while Grzeszczuk successfully evolved controllers that enabled a shark to follow a path while swimming, evolving more varied parameterizations for controllers is difficult.

2.2 Motion Transformation

All of the techniques discussed so far attempt to produce motion or controllers from first principles. A consequence this approach, which we have noted in reference to each technique, is a difficulty in achieving varied styles of performance because of the need to specify style numerically. As we have already mentioned in the Introduction, an alternative to "raw" motion synthesis is to utilize existing motions, *transforming* them to suit new needs. The goal of all methods following this approach is that in transforming motions, we are able to preserve their style, *whatever* it may be.

We classify transformation methods into three categories: deformation, blending/interpolation, and signal processing, and examine prior work in each area. A starting point for all of these techniques is to address a motion as being composed of a set of *motion curves*, where each motion curve is a function of time that specifies what one of the character's joints (or DOFs) does over the course of the motion. We will discuss several of the techniques in greater detail in chapter 4, since they form the core of our own transformation engine.

2.2.1 Deformation

One way to transform a motion is to deform or bend parts of it so that the result meets certain instantaneous or persistent constraints. The first technique to accomplish this is *motion warping* [Witkin 1995, Bruderlin 1995], which is capable of satisfying instantaneous pose constraints. The technique, which can operate in both the spatial domain and the time domain, allows the animator to add any number of new keyframes to an existing animation without regard to the structure of its motion curves. The algorithm computes smooth displacement functions to *add onto* all of the affected motion curves such that the sum of the displacement and the original motion curves results in a motion that passes through all of the new keyframes. Under the supposition that high frequencies and temporal inter-relationships in the motion curves account for much of the perceived style of a motion, warping preserves the style because the displacements are computed to contain as few high frequencies as possible.

One shortcoming of the original motion warping technique is its inability to enforce continuous constraints, which can lead to, for example, skidding feet when transforming a walking motion. Gleicher [Gleicher 1997] addresses this by applying parts of the spacetime constraints apparatus, utilizing the optimization to solve for the displacement curves in the presence of persistent geometric constraints. He is able to scale spacetime to characters of humanoid complexity in part by eliminating all of the physics considerations, so the result has no claims to dynamical soundness. Popović [Popović 1999b] also uses spacetime, and is able to retain the physics by simplifying the structure of the character itself for any given motion to be transformed, then applying spacetime to deform the motion of the simplified character, finally mapping the changes back onto the original character. Although it does not operate at interactive rates, this technique can alter physical parameters such as gravity as well as continuous constraints to produce an impressive range of dynamically sound[3] motions. Finally, Lee [Lee 1999] applies multi-scale basis functions to

[3] Only the motion of the simplified model is truly dynamically sound. Discrepancies may be introduced in the mapping back to the original character.

Gleicher's formulation, replacing the spacetime optimization with a less expensive multi-pass inverse kinematics optimization.

The major shortcoming of all of these deformation-based techniques is the limited range of deformations that can believably be applied to any single motion. The movement involved in picking up something on the ground is so different from the movement necessary to pick something off of a shelf overhead is so different that, even with physics to guide it, no deformation algorithm could successfully warp one into the other. Also, because the original animations are treated as a mostly unstructured set of motion curves, all alterations are intimately tied to the particular motion – and thus style – so that style is not one of the parameters of motion we are able to change easily. The solution to these problems, we believe, is to combine deformation techniques with other transformation algorithms that can combine *multiple* motions.

2.2.2 Blending and Interpolation

We can address the major shortcoming of deformation-based methods by addressing not single motions, but sets of related motions, and blending or interpolating between them. For instance, given a small hop and an energetic jump upwards, we can synthesize a jump of any energy level between the two by interpolating by different amounts between the two. Wiley [Wiley 1997] developed an interpolation scheme based on dense, regular sampling of the space of motions we wish to reproduce. Rose [Rose 1998] adapted more sophisticated mathematics to enable multi-dimensional interpolation of quaternion valued motion curves that allowed more freedom and sparsity of sampling. In Rose's scheme, primitive actions were called *verbs*, which defined a specific set of example motions. Each example was characterized by a set of *adverbs*, for instance, a "reach" motion might have *x, y,* and *z* adverbs, and a walk might have *happy-sad* and *knowledgable-clueless* adverbs. The motion builder can supply any number of example motions, provided each can be characterized by a value for each adverb corresponding to the verb. The end user can then specify any combination of verb and adverb settings, and the interpolation mathematics (*viz.* radial basis functions) will synthesize a corresponding motion by combining the examples in real-time.

There are two major drawbacks of these and all interpolation-based techniques. First, as we add parameters by which we would like to control the resulting animations, the number of motion samples we need grows exponentially. Second, it is difficult to attain or maintain precise geometric constraints, so interpolation alone is not suited for precise work. In our work we use interpolation for its strength – achieving a wide range of motions by combining multiple examples – while incorporating other transformation algorithms (such as deformation operators) to make up for its weaknesses.

2.2.3 Signal Processing

As motion curves are in some sense simply signals in the same sense as a radio transmission, several researchers have reasoned that we can create interesting animations by applying various signal processing techniques to motion curves. Unuma [Unuma 1995] applied Fourier analysis to cyclic motions such as walking and running, and was able to extract the difference between a dejected walk and a normal walk, and then able to apply the difference to a run, for example. It has yet to be shown how the approach can be generalized to non-cyclic motions, however. Bruderlin [Bruderlin 1995] applied Laplacian decomposition to arbitrary motions to decompose each motion curve into frequency bands. He then presented a collection of techniques that provided a “graphic equalizer” control over the motion. Such manipulations allowed interesting variations in the style of the motions, but provided no means of achieving new task-space goals, nor any real means of characterizing the variations. We will, however, make mention again of the Laplacian decomposition in chapter 6, where we hope to utilize it for synthesizing transitions.

2.3 Combining Primitive Actions

All of the work we have discussed so far (except for that of Rose) addressed only the problem of synthesizing single, primitive motions. Interesting animations and the creation of believable personalities will almost always involve many motions, executed in sequence and in tandem. We will examine several mechanisms for combining motions.

Traditional animators have long known that subtle movements and mannerisms are crucial to portraying engaging, convincing personalities [Thomas 1981, Lasseter 1987]. In the research community, Morawetz [Morawetz 1990] was among the first to explicitly recognize this by formulating *secondary actions* – independent, primitive motions representing mannerisms and gestures – and structuring an animation so that the secondary motions could be grafted on top of some underlying “main” action, such as walking. We will extend these ideas to our concept of *layering* motions, in chapter 6.

To be able to layer motions, we must first be able to *transition* motions, *i.e.* change a character’s movements from performing one action to performing another action in a believable, natural way. Perlin’s early animation work [Perlin 1995] introduced blending to transition between two motions, in which we overlap the two motions between which we wish to transition, and perform the animation equivalent of a cross-dissolve. This was extended somewhat in Perlin’s later work [Perlin 1996], and further yet by the “Verbs and Adverbs” work of Rose [Rose 1998] by the addition of *action buffering*, in which a transition that causes bodily interpenetration can be repaired by inserting an intermediate pose (or, in Rose’s formulation, intermediate motions) through which the blend will pass. The buffered action is constructed in knowledge of the motions between which it will be inserted, and is crafted to avoid all body collisions. However, even with these extensions, blending has a very limited range of applicability for transitions, as we will discuss in chapter 6. To produce more realistic transitions over a wider range, Rose [Rose 1996] applied the spacetime constraints apparatus to synthesizing new motion to “fill in the gap” between two motions. This work also introduced an interesting language for building expressions of motions and motion clips – small snippets of animations that may contain any subset of the full animations motion curves over any time interval, which we will discuss in chapter 3.

Assuming that we possess an adequate means of generating primitive motions and transitioning between them, we still need a means for organizing motions to enable content providers to construct animation scripts and author character personalities. This becomes crucial when we wish to design real-time, interactive systems with computer controlled characters. Perlin's work on Improv [Perlin 1996] (which has recently evolved into a commercial product) allows the animator to define primitive behaviors and easily script them together into continuous animation. Both Bryan Loyall's work (in conjunction with the Oz project) [Loyall 1997] and Bruce Blumberg's work (in conjunction with the Alive! project) allow autonomous characters to be directed at the goal level, capable of generating their own detailed scripts from their desires and the state of the environment they inhabit. These systems enable "character builders" to design complete, autonomous characters at all levels, from low level motor control systems, to goal coordination and arbitration layers, to the definition, modification, and reaction of the character's desires and goals to his own agenda and events that occur independently in the virtual world. Such ambitious projects have focussed on the higher level aspects of the problem, leaving room for symbiosis with work such as ours, which would be considered to address the low level motor control aspect of a character.

Finally, the Motion Factory [Zhu 1995] is a commercial authoring environment and real-time 3D animation engine that utilizes robotic path planning and inverse kinematics techniques to generate dynamic, goal-driven animation. It addresses many of the "intermediate level" issues of motion automatic motion synthesis such as path planning and collision avoidance. While it provides a character builder with considerable power and convenience, the quality of its results, particularly with respect to meeting precise constraints, is lacking. We surmise (details on the proprietary system are unavailable) that this is due in part to the robotic origins of Motion Factory's algorithms, and due also in part to the severe constraints placed on those algorithms by the necessity of real-time performance.

While the problem of automatic script generation is an important one, it is beyond the scope of this work. Although we will develop our own structure for scripting primitive motions, we view our work as a "back end" animation engine that could easily be plugged into any of the systems described in the last paragraph to render a dynamic script into high quality motion.

2.4 Situating Our Work

The work in this thesis draws inspiration from many of the techniques presented here, and can be classified in two different ways. First, we can consider it as a new technique for motion transformation. By attaching structure and rule-based knowledge to example motions, we are able to combine the benefits of several existing transformation techniques, and add the ability to deal with style as a transformation parameter. Second, we can consider it as a new methodology for creating motion controllers (via motion transformation) that are highly composable, thus forming the most complete attempt at automatic motion synthesis to date.

3 Knowledge-Enhanced Motion Transformation

In this chapter we give an overview of the entire thesis, beginning with the problem we set out to solve and our guiding philosophy while attacking it. We then describe all of the pieces and concepts that make up our knowledge-based approach, and give an explanation of all the types of knowledge utilized therein. Finally, we explain how all of the pieces fit together and use the knowledge to create a coherent automatic animation engine.

3.1 Goal and Philosophy

In Chapter 1 we described all the facets of the animation problem that this work addresses. Briefly recapping, our goal is to develop a means of character animation in which we compose primitive actions both sequentially and simultaneously to produce rich, engaging behaviors. The primitive actions should be parameterized by a reasonably small set of task-level parameters that span the full range of functional variation for the action, as well as stylistic parameters that allow the action to be performed in any manner realizable by a performance artist or master animator. When the primitive actions are composed, the transitions between them must be not only natural, but maintain the style of the actions being composed, and allow control over the level of coordination among the actions. Finally, introducing new actions and styles of performance should be straightforward; we desire that introducing new actions would be in the domain of an animation *technical director* working in conjunction with a *master animator*, while the introduction of new styles should be within the grasp of the advanced end-user. In the computer animation industry, a *technical director* is a computer scientist or software engineer with deep knowledge of some portion of the mathematics and algorithms underlying computer graphics, and a *master animator* is an accomplished and knowledgeable animator who would, for instance, animate a main character in an animated movie.

Such a goal statement is fairly broad, and nearly all previous work on automatic animation generation can be seen, in a certain light, as an attempt at fulfilling it. The specific approach developed in this thesis was guided by many further requirements, beliefs, and underlying assumptions, some of which have been discussed in previous chapters, but which we now collect and describe here. Knowledge of these issues up front will aid both in understanding some of the choices we made, and in judging our success.

- **Requirements of believability.** The end goal of character animation is to tell a story, whether it be a traditional, linear story as told in movies, or an interactive, evolving story as in the best of today's computer games. In theater and cinema, bringing a character to life and making it engaging may require an actor to move, speak, and emote in ways that "real" people seldom or never do. The same is true in animation, which is fairly obvious to any observer of successful cartoons. However, the animation research community places a great emphasis on dynamically sound and "optimal" motion; this is mainly pragmatic, since dynamical soundness and optimality (in terms of energy consumption) are easily quantifiable, thus giving researches a numeric measure of how "good" their animation is. Subjective measures such as believability and "engaging" are difficult to quantify and use as a basis for comparison of different approaches, and so are generally eliminated from consideration.

  We believe, however, that physical realism and physically based metrics have very little to say about how to generate or quantify stylistically interesting and engaging animation. Moving beyond physics is so vitally important to producing good animation that we consider physics to be one tool among many, which the animator can choose to use, ignore, or supplement on a case by case basis, as needed. This will, of course, make evaluation of our work more difficult, which we will address in Chapter 8.

- **No free ride.** We do not believe that it is possible, at current or foreseeable levels of technology, to devise an algorithm from "first principles" alone that can produce good parameterized character animation allowing for significant stylistic variation. There are no promising leads for developing meaningful mathematical descriptions of emotion or style suitable for guiding spacetime algorithms; trying to program motion/muscle controllers via cognitive modeling/learning is perhaps as difficult as the general AI problem; emergent algorithms must become much more sophisticated (and complex) to be useful for evolving useful parameterizations of style along with the motions they produce. It is also unclear how a master animator could contribute significantly to the development of these techniques, since their methods are so different from those with which animators are familiar.

We believe there must be significant human input in the process of developing engaging actions and useful parameterizations for them, and that the humans whose hard work we would most like to capitalize upon are master animators. While we agree that it is both misguided and unlikely to try to "put John Lasseter in a box[4]" [Catmull 1997], animators more than any other group understand the requirements of believability, so capitalizing on their work and even a small part of their knowledge seems the most logical place to begin in designing a knowledge-based animation algorithm.

- **Linear knowledge complexity.** When considering the types of knowledge we can incorporate into our composable primitive actions, we desire only "linear" knowledge. By linear we mean that the knowledge need make no specific assumptions about any action other than the one for which it is defined. So, for instance, the knowledge responsible for generating a good transition into a *jump* action should not depend specifically on whether the action preceding the *jump* is a *walk* or a *wave*. By considering such "pair-wise" or higher kinds of knowledge, we could make the job of designing transition algorithms much easier, but we would cripple the extensibility of the approach, since each time a new action is added to an existing set of *n* actions, we would need to create *n* new transition rules for transitioning into the new action from any existing action, as well as augment the transition rules of the *n* existing actions to handle transitioning from the new action into each existing action.

- **Important characteristics of animation.** We have already stated and will soon state again that our low-level animation engine is based on motion transformation. In order for any transformation algorithm to produce an animation that is equally as good as the input starting motion, we must determine the quantities and qualities of the animation that, when preserved, ensure that both the style and realism of the original are also transmitted. Realism, which deals largely in physical plausibility, is relatively straightforward to define and preserve, to varying degrees, through imposition of constraints that enforce time-varying geometric or physical relationships (such as feet not skidding or slipping while walking). Style and its preservation, however, are more nebulous. However, based on our own observation, and corroborated by the investigations of Bruderlin and Williams [Bruderlin 1995], Witkin and Popović [Witkin 1995], and Unuma *et al* [Unuma 1995], we posit that the quantities that most determine visible style are the timing and coordination of the sub-actions that comprise the action (for instance, in walking, the individual steps, arm swings, and hip sway), and the high-frequency content of the animation, which contains most of the subtle details of the

[4] Creativity and the ability to draw upon vast and diverse experiences and observations are integral ingredients for creating unique, engaging animations. It would be virtually impossible to cull some set of rules or other knowledge from John Lasseter (or any other animator) that could, working from a description of some character, create a set of animated behaviors that bring the character believably and uniquely to life.

motion. Assuming we use transformation algorithms that preserve these quantities (see discussion in Chapter 4), the quality of our results will be an affirmation or denial of this hypothesis.

- **Only rigid-body animation.** In this work we are only considering rigid-body animation, the motion of a character's skeleton. Bringing a conventional humanoid character convincingly to life depends heavily on animating the "soft" parts of the face. Experience with current state-of-the-art character animation software [Hash 1999] gives us confidence that many of the principles developed in this work can be applied to soft-body animation. However, since the underlying infrastructure is radically different from that used in rigid-body animation, it is beyond the scope of this thesis to experiment with soft-bodies.

3.2 An Approach to Knowledge Based Motion Synthesis

As we have already mentioned, motion synthesis algorithms that generate motions from scratch, whether it be via physics-based optimization or robotic-style controllers, face the daunting task of characterizing and parameterizing style mathematically. Motion transformation algorithms, on the other hand, are able (under the assumptions listed above) to bypass this problem by simply preserving the style of a previously created or performed motion while deforming the motion to meet new task level goals.

In this thesis we attempt to solve the animation problem as defined at the beginning of this chapter by taking motion transformation as our fundamental tool, and addressing the current shortcomings listed in the Introduction and Background chapters (*i.e.* no method of grouping, reusing and parameterizing motions by function; limited range of useful deformation; only cursory consideration of how to combine separate motions into a cohesive animation) by applying structure to the animation problem that will allow us to systematically reason about and apply knowledge to the motion synthesis process.

3.2.1 Motion Models

We begin by defining an animation as a set of actions performed by an actor – "actor" is simply the name of our abstraction for an animatable character; each action is performed in one particular (of possibly many) styles, and actions may overlap, cooperate, or execute sequentially. Working from this definition, we provide a basic unit of animation, the *motion model*, which is a modular, parameterized motion generator capable of producing motion that executes a single type or class[5] of action (*e.g.* walk, jump, sigh) in any situation, and in any number of styles, for any actor[6]. The motion model for a particular action class con-

[5] We use "class" in a very generic sense here. We will refine more specific uses in the next section.
[6] Motion models are designed to dynamically gather all character-specific information from an actor at run-time, and can thus operate on a wide range of actors. However, the raw motion data contained in styles must be adapted to characters of greatly differing body proportions or types before they can be successfully applied to characters other than the one they were originally animated on. Gleicher [Gleicher 1998] and Popović [Popović 1999b] address this problem; we do not.

tains knowledge that specifies the important aspects of the action (*e.g.* in a *throw* motion, the throwing arm cocks, attacks, and then releases in the direction being aimed at), defines a useful set of parameters for controlling the action, and determines how the action changes in response to different parameter settings.

Motion models generate motion through transformation. Although alternate generators such as dynamics simulation or procedural motion controllers can serve as the guts of a particular motion model class, most motion models for character animation are designed to take advantage of our knowledge-based motion transformation algorithm, which consists of two parts:

1. **Internal.** Inside a motion model, rules specific to the motion model translate the current parameter settings into a recipe for combining and deforming **base motions**, which are annotated, previously performed or animated examples of the motion model's action. These recipes can include any of the following transformations of base motions: selection, splices, time-varying blends, time dilations, reflections, and motion warps. The exact recipe depends on both the motion model class and the current parameter settings, and is the first application of specific knowledge to guide the motion transformation process. For example, a *throw* motion model might have two base motions, one for a short throw, and one for a long throw. So, depending on the distance of its *target* parameter from the actor, the recipe blends the two base motions to produce a throw that is more or less energetic, then applies appropriate warps on the throwing arm at the "cock" and "release" times to make sure it points towards the target. Of course, if the actor is left-handed and the base motions were performed right-handed, the recipe would first reflect the base motions.

   When the "generator" recipe is done, a combination of general and specific *transition rules* determine how the standalone motion is integrated into the larger animation. This is a multifaceted problem, which we will discuss in more detail below in section 3.2.3.

   Finally, the motion model must ensure that the important geometrical relationships within the action, which we call **invariants**, are maintained throughout the motion. A common invariant in character animation is that generally when a foot is providing support, it should neither slide across nor penetrate the floor. Many single transformation algorithms, such as motion warping and blending/interpolation do not have the capability to enforce invariants; since our algorithm combines the strengths of multiple algorithms, we can. However, since each motion model in the animation will specify invariants, and since invariants from different motion Models may overlap in time and thus interact, it is not possible for each motion model to enforce its invariants in isolation. Instead, the last act that is "internal" to a motion model is the exporting of its invariants so that when all motion models have executed their internal stage, we can complete the animation by moving on to the second stage.

2. **External.** In order to produce the final-quality animation from the assembled and spliced animations of all the individual motion models comprising the animation, we must ensure that at each frame, all of the invariants active at the frame are satisfied. Since an invariant is basically a geomet-

ric constraint, this process amounts to a constrained optimization in which we try to deform the animation as little as possible (in order to maximally preserve its style), and is itself a kind of motion transformation algorithm. We introduce several existing algorithms as well as our own new, inexpensive algorithm, in section 3.5.

Generating believable motion from a parametric description of the motion is one facet of the purpose and knowledge encoding of a motion model – the other is *providing* a parameterization of the motion that is useful both for initially scripting an animation and subsequently editing it. We define a good parameterization as one that, for a goal-oriented action, provides coverage of the entire goal domain with a reasonably small number of logical parameters. Naturally, "good" parameterizations differ for each motion model class, depending on the complexity of the motion models' goals; there are, however, some parameters common to all motion models, one example being the parameters that specify and fine-tune the transitions into and out of the motion in the animation. Furthermore, the individual parameterizations share additional uniformity in that the *types* of all parameters belong to a small set for which we provide standardized manipulation widgets or direct manipulation interfaces, as appropriate. Although we *can* construct motion models that support multiple "stylistic" parameters (*e.g.* step-length, hip-sway, and speed for a *walk* motion), we advocate bundling most stylistic parameterization into the single "style" parameter, described in the next section; doing so not only provides uniformity among motion model parameterizations, but can also greatly simplify the implementation of the motion models.

In Chapter 5, we will describe in detail how motion models operate and how they can be created in a structured fashion. We will provide worked examples and describe an API for motion model construction.

3.2.2 Styles

A **style** specifies a particular performance of a given action; meaningful dimensions of stylistic variation include emotion, energy (spirited vs. lethargic, *etc*), and personal style (George's run, John's walk). In the context of motion models, a style is a particular set of base motions and accompanying annotations; the motion model is created in the knowledge that there will be many styles associated with it. Although there are limitations on how a style can differ from the description of an action contained in a motion model, the room for interesting variation is generally quite large. One of the strengths of this work is that all of the styles associated with a motion model are interchangeable with respect to the task-level specification of a developing animation. That is, if the user has, for example, defined a convoluted and detailed path for an actor to *walk* down in a *happy* style, and then decides the actor is actually feeling a bit mischievous and should be *sneaking* instead, the user can simply switch the style from *happy* to *sneaking* in assurance that the goals imposed by the previously defined walk-path will still be satisfied while sneaking.

3.2.3 Motion Combinations

Of paramount importance in constructing interesting, rich behaviors and animations is the ability to combine, layer, and segue primitive actions expressively and controllably. Let us first consider just the simple case of the segue, where we move from performing one action to performing another action. Even here, there are many decisions for an automatic system to make, and possible dimensions of variation that a particular situation may require. For instance, *when* in the first action should we begin the transition, at what point in the second action are we transitioning *to*, and *how long* should the transition be? Should there be a *pause* before beginning the second motion? Should we try to *hold the pose* of the first motion while performing the second? And most importantly, how do we generate natural, stylistically consistent motion over the transition? When we expand our scope to include layering of motions (so that multiple actions can be performed at the same time, provided it makes sense to do so, *i.e.* one cannot walk and sit simultaneously), we add more aspects. If one of the motions completes before the other, how and over what timeframe do we either resume performing just the remaining action, or introduce another action to replace the completed one? How can we optimize the animation for the *combination* of the actions – consider the situation of a man trying to grab an apple hanging from a tree branch: he will be *reaching* while *jumping*, and the interaction between the two separate motions strongly affects the overall motion; for example, the act of reaching while in midair will cause the man to pivot about his center of mass, which should cause him to land in a different orientation than he otherwise would have; furthermore, the range of spots that can be reached while in midair depends on the trajectory of the jump, which in turn depends on the landing spot and height of the jump; thus, changing the goal of the reach action should initiate an arbitration whereby the parameters of the jump, if free to change, actually do so to best accommodate the reach. Many of these issues have not been recognized in previous work; we believe this thesis to be the first graphics work to consider the totality of these problems, which we refer to as "**motion combinations**," in depth.

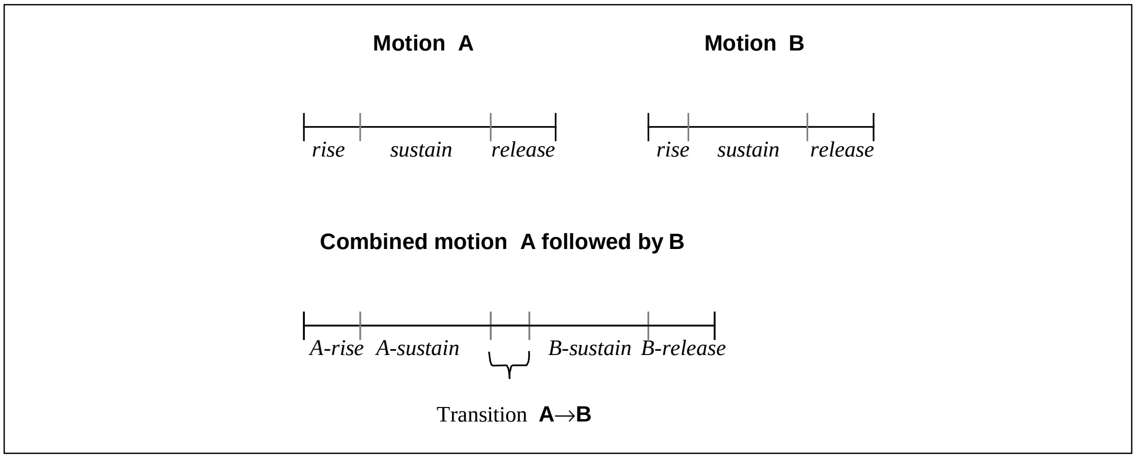


Figure 3.2: Transition Entrypoints. We wish to segue motion A into motion B, which are shown in interval form (time along the horizontal axis). The transition begins when A would have begun its *release*, and produces motion that achieves the pose at the end of B's *rise*. The transition replaces both the *release* phase of A and the *rise* phase of B.

We will now outline our solutions to these transitioning problems, which we will discuss at length in the motion combinations chapter (Chapter 6).

- **Transition entrypoints.** The user should have final say over exactly when a transition begins and ends, but we want the automatic system to generate defaults that make sense most of the time. To the degree that most actions are goal-oriented and consist of a "rising action" phase, a "main goal" phase, and a "falling action" phase, common sense dictates that we should begin the transition from motion A to motion B when we have finished the main part of A (that is, at the end of the "main goal" phase), and end it at the beginning of the main part of B (carry the transition through the "rising action" of B). These relationships are demonstrated in Figure 3.2. Each motion model knows when these phases occur in their motions, and transitions use these values by default. The user can specify an offset from these defaults for each transition, which gives ultimate control.

- **Transition duration.** Our observations of people in motion have led us to the conclusion that the duration of what we consider to be the "transitional motion" is highly dependent on not only the classes of the motions involved (*i.e.* motion model), but also the styles of the motions. Let us for the moment simplify the transitioning problem to generating motion that takes the actor from pose "*a*" (the pose the actor holds at the beginning of the transition in motion A) to pose "*b*" (the pose we wish the actor to have at the end of the transition in motion B). We have identified three different classes of transitions with respect to duration: *fast-as-possible*, where we just wish to move into the goal pose *b* as quickly as possible subject to how fast the actor can actually move – this is frequently applicable when motion B is an athletic action; *constant*, where the duration of the "rising action" phase of motion B determines the transition duration – we have found this to be applicable only for certain styles of performance involving exaggeration brought on (typically) by timidity or fear; *proportional*, where we take from the "rising action" phase of B the *pacing* of the transition, adjusting the actual duration based on how much movement occurred in the "rising action" compared to how much motion is required to move from *a* to *b* – this type of transition is the most frequently occurring. The *proportional* transitions require both a measure of motion and a function for translating differences in motion distance into difference in durations. For the former, we develop a measure of *pose-distance* in Chapter 4, based on the mass displacement formula developed by Popović [Popović 1999a]; for the latter, we propose a learning algorithm that can utilize example transitions or corrections from the animator, discussed in the motion combinations chapter.

- **Pauses and Holds.** Pauses and holds are aspects of transitions not previously considered at all in the graphics literature, but which we believe are important enough to effective storytelling that they should be explicitly accounted for. Pauses are often critical when an actor experiences a sudden change of mind or dawn of realization: a character begins to raise his hand to argue and emphasize a contentious point, but halfway through realizes he was wrong and is making a fool of himself; he stops in mid-rise and mid-breath, pauses for half a second, then drops his arm and clamps his mouth shut. Holds are important when an action is inserted into the middle of an otherwise continuous sequence of motions. Let us assume we have two types of actions: *look*, which causes the actor to look in some direction while in a standing-up position, and a *jump*, which causes the actor to jump from one place to another. Now we have an animation where the actor performs two successive jumps (across rocks in a stream, perhaps), and the default transition between them causes the actor to launch into the second jump right from the absorption-crouch of the first, without rising back to a standing position in-between. But now the actor hears something behind him as he is finishing the first jump, and pauses to look over his shoulder – during the *look*, we may not want the actor to rise to his feet while looking back, but rather stay as much in the crouch as possible, with the foreknowledge of the upcoming second jump. Our provisions for these mechanisms are dependent on the details of how we assemble motions from motion models, so we will delay further discussion of them until we have covered the appropriate material.

- **Motion Generation.** A motion model may be constructed so that it can begin acting from any pose and velocity the actor may possess; however, the more common case, particularly for the type of motion models we propose, is that it is well-suited to beginning its action from one of a small set of poses. This means that when the previous action segues into the current one, the actor may be in a pose quite different from any of those the motion model generating the action knows about. Therefore we must be able to generate motion that takes the actor *consistently* from the pose at the end of the previous action to a known pose in the current action, which is typically the pose at the end of the "rising action" phase of the motion. "Consistently" refers to the style of the generated transitional motion: if in both the previous action and the current action the actor is trembling with fear, then the transitional motion should be as well.

Stylistic considerations aside, most of the time the only purpose of transitional motion is to quickly and naturally get from one pose to another – there is no strong dependence on the actual action into which we are segueing, thus no dependence on the motion model. Therefore, we are able to lift the task of generating transitional motion from the motion models themselves (although they are always able to, if they wish), moving it up to actor-class-specific **Transition Generators**, which use rules and signal processing to produce transitional motion between any pair of actions an actor may perform. "actor-class-specific" means we would create a different transition generator for each type of actor (*e.g.* humanoid, quadruped, etc.); as with most constructs in our scheme, transition generators can be specialized, by either more specific types (*e.g.* equine and bovine quadrupeds) or by specific individuals (*e.g.* the transition generator specific to Mr. Ed). The rules contained in transition generators pertain mainly to avoiding awkward or physically impossible configurations during the transition motion, such as interpenetration of bodyparts and loss of balance. The signal processing elements allow us to incorporate the high frequency content of the surrounding actions into the transitional motion.

In the motion combinations chapter we will also discuss why we need such an elaborate scheme for transitional motion when several techniques, such as blending [Perlin 1995] and spacetime [Rose 1996] already exist.

- **Layering Motions.** When we consider the possibility of layering actions so that multiple actions can execute simultaneously, we must address several additional issues. First, since the start of a layered action generally does not coincide with the end of the underlying action[7], we need a new approach to determining and specifying good starting points. We have identified three different rules for temporally placing the "goal" time of the layered action: so that it coincides with the goal time of the underlying action; at time *x*, where *x* may be absolute or relative to the goal time of the action upon which it is layered; to optimize some spatial quantity – for example, the goal of the layered action should occur when the action upon which it is layered brings some named bodypart closest to a specified goal condition. Motion models are free to decide which of these rules best applies to their motion (we give guidelines for choosing among them in chapter 6), and the animator can always override the system's decision.

  Secondly, we need both a language and mechanism for decomposing, mixing, and reassembling motions. Whereas with simple segues we transition one entire set of motion curves into another, when layering, the motion curve controlling a particular joint may come from either or a combination of both actions (and their corresponding sets of motion curves). Furthermore, the particular combination of the underlying and layered motions will be different for different joints. This information, which is derived from knowledge of how important each joint is to performing a specific action, is captured in motion models. For instance, the *walk* motion model knows that the joints pertaining to the legs and feet are crucial to successful walking, and thus cannot be shared with a layering motion, while the joints in the upper body are more negotiable, and can be shared or suborned by a layering motion. The means of actually recombining the constituent motions is provided by our motion expression language, described in section 3.3.2.

Lastly, layering introduces the problem of needing to resume motions. Without layering, each action has a single transition associated with it – the transition from the previous action into itself. With layering, however, it is generally the case that the layered action will not end at the same time as the underlying action; when each action ends, it may segue into another action, or yield control completely of the joints to which it contributes. When either of the actions yields control, the other must assume it (or else some joints will have no animation), necessitating a second transition over which the remaining motion resumes doing what it would have done if the other motion were not present.

With the ability to modularize and parameterize primitive actions in a number of styles and combine them together in powerful ways, all that remains for our approach to be complete is consideration of how animations are actually specified, and what mechanisms we can provide for editing and tweaking the animation beyond the level of control provided by motion model parameters. In section 3.5 we will discuss these issues in the context of the animation engine we have derived from the concepts just presented; but first we present, in more concrete terms, the kinds of knowledge we have successfully incorporated into our transformation-based animation algorithm.

3.3 Meta-Knowledge (Organization)

In subsequent chapters we will describe in detail how various types of knowledge are derived, implemented, and incorporated into the different parts of our animation algorithm. It will aid in the understanding of each chapter, however, to already have a broad view of all the different types of knowledge we consider, and how they fit together.

---

[7] Imposing a hierarchical ordering on layered motions is necessary for motion models to be able to execute properly, as we will see in chapters 6 and 7.

Before we can begin discussing the specific types of knowledge that are encoded in various parts of the algorithm, we must describe several important choices about the way we structure and organize the animation problem. These choices shape the "animation language" in which subsequently described knowledge is encoded, so we think of them as meta-knowledge.

3.3.1 Class Hierarchies

Any successful knowledge based algorithm depends on the ability to specify general knowledge that can be applied in any situation, while also retaining the ability to override that general knowledge for more specific cases where targeted, more applicable knowledge is available. From system architecture and software engineering standpoints, this is especially important, since it allows us not only to reduce code redundancy (knowledge applied to many individual cases is extracted to a single, more general rule), but also provides a clean mechanism for extending the knowledge database to handle new situations.

We have found that the object-oriented abstraction of hierarchical classes is especially well suited to encoding animation knowledge. A class defines a set of objects that share a common set of behaviors (typically in the form of functions and queries) and structure. Classes may be derived from other classes, in which case all objects of the derived class inherit the behaviors of the "parent" class, but the derived class is free to override any or all of those behaviors, and also is able to add new behaviors specific to the derived class and any further classes derived from it. For more information on classes and objects, see Stroustrup [Stroustrup 1994]. As we will describe shortly, we use class hierarchies extensively in the abstraction for the physical representation of actors, and in creating motion model libraries. Finally, although our prototype implementation uses C++, the use of the hierarchical class abstraction is not tied to our use of C++; other, more flexible object-oriented languages, like CLOS [Keene 1989], would afford us the same abilities, but with a different set of practical tradeoffs (which aren't really relevant).

3.3.2 Clip Animation Language

Since we are primarily concerned with manipulating and combining motions to form new motions, the choice of how to represent motions and motion operators is critically important. In their paper on spacetime transitions, Rose *et. al.* [Rose 1996] develop a language for combining motions in which primitive motions, called *Motion Units* and consisting of an ordered set of motion curves, can be chopped into bits and recombined using boolean set membership operators. Most importantly, the output of any of these splicing operations is itself a Motion Unit, and can be further spliced just as a primitive motion.

We adopt a similar idea, but extend it to consider not just recombinations of motions, but also deformations of motions. As we will describe in Chapter 4, we define a class for motions called **clip**, and every type of motion or manipulation of motions is a class derived from clip. Some examples are clip-primitive for imported motion captured sequences, clip-space-warp and clip-time-warp, which perform geometric and temporal motion warping of clips, and clip-composite, which performs sophisticated blending and splicing of clips. Because each possible deformation of a clip is itself a clip, it is very easy for motion models to create output motions from base motions by successively refining them – each refinement not needing to know the specific type of the motion(s) upon which it operates; the resultant motion is simply a tree whose internal nodes are motion clip manipulations and whose leaves are clip-primitives. This point is truly the crux of the Clip, because it unifies all motion transformations into a function-composition language that treats transformations and deformations of motions the same way it treats primitive motions.

Because motion models produce clips, they could themselves be cast as members of the clip language. However, our intent for clips is that they be as lightweight as possible, since the structure of motion expressions comprising an animation may change continuously as an animation develops, and thus clip expressions are created and destroyed frequently. Motion models are fairly heavy objects, and we only destroy one when its action has been deleted from the script.

3.4 Knowledge Encoding

All specific knowledge is associated with a particular high-level concept or data abstraction, which determines the scope of the knowledge in the animation process. The four concept/abstractions we make use of are the *character class,* the *motion model,* the *style,* and the *actor.* We will now describe the scope of each of these abstractions, as well as the types of knowledge that can be attached to them.

3.4.1 Per Character Class Knowledge

As we will describe in Chapter 4, the physical representation of each actor is a **character model**, which contains mathematical descriptions of both the skeleton (which determines how the actor can move) and geometry (which determines what the actor looks like). A character model can be generic, in which case there is no other information associated with it. However, we can also construct a character model that conforms to one of a set of pre-defined (but easily extensible) specialized classes of character model that provides useful information. We define two such classes, **Humanoid** and **Rigid-Body**, which contain the following types of knowledge:

- **Physical Structure.** All humanoids have the same skeletal structure and bodyparts[8], and the Humanoid class applies a consistent naming scheme to them. Thus, regardless of actor Dave's size or shape, any motion model acting on him can reliably query the position of his "head" or place a constraint on the position of his "left foot".

[8] Obviously, this is not always true. Humans may have fewer (or more) than the standard limbs. Furthermore, we may wish to design characters that are humanoid (*i.e.* move and behave much like humans), but not human – possessing tails, antennae, two heads, *etc.* Chapter 4 discusses how we deal with this.

- **Common Handles.** Handles are points or orientations on the character model that can be pulled using inverse kinematics to effect movement. By pre-defining the most common types of handles, we facilitate the creation of actor-independent motion models. Some of the handles defined for the Humanoid class are used for planting the actor's feet, maintaining the actor's balance and controlling the actor's direction of gaze.

- **Inverse Kinematics Algorithms.** In our implementation we employ a powerful, general-purpose inverse kinematics algorithm. However, it is possible to construct special-purpose algorithms that use knowledge of the character class to improve efficiency and/or results. We will discuss this topic briefly in Chapter 4.

- **"Macros" and Sub-motions.** There are some elements that are pervasive of most actions that actors of a certain character class perform. We can, therefore, make the tasks of describing, creating, and controlling motions more efficient by extracting these elements into the character class, available for all motion models to draw upon. The Humanoid class defines the concept and functionality of a footstep (the functionality includes maintaining a non-sliding foot-plant while the foot is on the ground), and also makes available maintenance of static balance while at least one foot is planted.

- **Transition Rules.** One of the major challenges of believable motion synthesis is ensuring that transitions between motions are plausible (*i.e.* no interpenetration of bodyparts). Since interpenetration-avoidance depends on the physical structure of the actor, the character model is a logical place to store knowledge that can guide our transition generators in this matter.

In short, the character class defines the "virtual actor type" for which a motion model is written; the motion model can then be successfully applied to any actor of that type. Furthermore, although we do not explore it in this thesis, character class hierarchies can be manipulated to map entire libraries of motions onto very different types of actors. For example, if we wanted to make a dog perform the same actions as a human, we would instance the dog actor from a character class "Humanoid-Dog," which is derived (inherits from) the class Humanoid; if all of the named handles and degrees of freedom defined in Humanoid are properly mapped into corresponding attributes in Humanoid-Dog, then any motion that can be applied to a Humanoid can also be applied to a Humanoid-Dog.

3.4.2 Per Motion Model Knowledge

The motion model is the central concept of our approach, responsible for the majority of the motion synthesis, and consequently, repository of the most and varied knowledge. Some of the most important types, which will be discussed again in Chapter 5, are the following:

- **Base Motions.** Our formulation of motion models generates new motion by combining and deforming existing base motions. Most non-gestural motions (and even some gestural motions) can have significant task-level variation – for instance, all the possible targets of a *reach*, a *jump*, or a *throw*. The very first knowledge a designer puts into a motion model is the number and kind of base motions it will contain, along with the rules for combining the base motions to satisfy any setting of the task-level parameters. This decision also determines the structure of styles for the motion model, since a style is basically a performance of the base motions.

- **Timing Structure.** Most actions have a rhythm and several "phases" they pass through in executing. By identifying the phases of a motion, we make it easier to control the timing of the motion as a whole (as, for example, to exaggerate it), and make it possible to automatically pick good times at which to begin and end transitions between motions. We have found that many motions fit well into a timing structure similar to the four phase "attack, sustain, decay, release" format used to describe sound synthesis.

- **Invariants.** Invariants are created by the motion model to specify aspects of the motion that must be preserved throughout both the transformation of the base motions into the synthesized motion, *and* during any subsequent layering or "tweaking" of the resulting motion. For instance, the *standing jump* motion model specifies invariants that keep the feet firmly planted on the ground during the launch and landing phases, and that keep the actor's center of mass on a parabolic trajectory while in flight. Invariants also serve as a means of exchanging information between motion models, and make the tweaking process easier.

- **Parameterization.** The parameters afforded by the designer for the motion model determine the "smart controls" over the final motion. By "smart controls" we mean the following: the animator is free to tweak the motion produced by the motion model *ad infinitum* using Motion Warping tools, but the only benefits of the motion model available during this process are the invariants; changing the motion via the motion model parameters, however, allows all of the knowledge contained in the motion model to be brought to bear on how to best alter the motion to accommodate the parameters.

- **Sub-Motion Importance.** When we want to allow multiple actions to execute simultaneously, the animation system must decide first whether the desired combination is possible (for instance, it is not possible to walk and jump at the same time, but it is possible to walk and chew gum), and second, where the motion for each bodypart is coming from when different motion models each want it to do something different. Each motion model definition specifies how important each bodypart is to the goal of its motion, and to what degree it would be willing to share or relinquish control of the bodypart to another, simultaneously executing motion model. Then, dynamically, as more actions are added on top of existing actions, the added motion models query the existing motion models to determine whether they can acquire the resources they need to execute.

- **Transition Rules.** The motion model contains several types of information that help to generate better transitions. We have identified several different means of determining the proper duration for a transition, and the motion model specifies which of these means are appropriate for its motion. The motion model may also be capable of identifying certain starting states (the pose or sub-motion produced by the previous motion model at its transition into the current one) and produce a different transition accordingly. For example, normally when transitioning into a jump, we perform a windup before launching; but when executing a series of hops (a hop is just a short *jump*), there is no need to windup between hops – we pass directly from impact-absorption of one hop to the launch of the next.

3.4.3 Per Style Knowledge

The knowledge contained in a style describes how its raw motion curves conform to the motion model's definition of the base motions, and can also provide refinements to some of the information contained in the motion model.

- **Motion Model Conformance.** The style provides a description of each of its base motions that "fills in the blanks" in the template defined by the motion model. It specifies the actual times at which each of the motion phases begins and ends, and when appropriate, states whether the action is right-handed or left-handed. It also describes any footsteps that occur in each base motion.
- **Added Invariants.** Some styles may contain natural motion invariants not considered by the motion model (because they do not exist in all styles). Therefore, it is possible for a style to describe additional invariants that the motion model will enforce, but only when performing in that particular style.
- **Refined Transition Information.** Some transition information maintained by the motion model, particularly the transition duration determination, can vary according to style.

3.4.4 Per Actor Knowledge

The most specific kinds of knowledge we consider allow us to attach behaviors to individual actors. Most of these items address the problem of creating unique personalities that pervade and dictate every action the actor performs, especially important when using an animation engine as a substrate for real-time, interactive agents. As such, they truly belong in the realm of the next-higher level of control in an autonomous agent control hierarchy: the behavior system or "brain" [Loyall 1997]. We mention them here because we do consider them, and our actor abstraction provides logical hooks to these types of knowledge.

- **Persistent Style.** A character designer can define moods or other parameterized concepts that determine what styles each or all motion models should use by default. Thus, if George's mood becomes "angry", all subsequently generated motion models would automatically select "angry" or "curt" styles.[9] The result could function similarly to "adverbs" in the work of Rose [Rose 1998].
- **Meta Style.** There are many other aspects to personal style that transcend our concept of a style, which can be expressed as per-actor data attachments. We can attach triggers and sub-scripts that specify that Gustav always wipes his hands on his pants before shaking hands, and reaches for the sky before looking at his watch. We can also specify that Mia is left-handed – since handedness is recognized by the Humanoid character class and all motions that involve a dominant hand specify on a per-style basis which hand *is* dominant in the base motions, the motion model can flip a motion to make it right or left handed as necessary.

3.5 The Animation Engine

Although we will discuss how the methods developed in this thesis can be applied to real-time control of avatars or agents in Chapter 9, the focus of the work and experiments we actually performed is the interactive creation of character animation, with attention given to providing maximal flexibility and customizability. As we will describe in the results chapter (Chapter 8), the kind of animation interface we aim to provide is one in which the animator creates a detailed script[10] for the animation interactively, by selecting primitive actions from motion libraries, configuring them using the built-in parameters, and composing them using any of the transition mechanisms we described in the preceding sections. This is all done using high level controls, and the animation engine fills in all the low-level details and manipulations. On top of this, however, is an additional interface for fine-tuning the motion produced by the system, allowing the operations of space-warping, time-warping, and smart keyframing, all of which we will discuss below.

In this section we will describe both the structure and control/information flow of our animation engine. From the previous sections of this chapter, we have motion models that produce motion curves for a single or repeated action, plus invariants and a control apparatus for modifying the goals and style of the motion. We also have the ability to create motion segments that transition one action into another and layer actions on top of each other. To form a complete animation system, however, we still require a means of coordinating and combining motion models and assembling their individual sets of motion curves into a continuous animation. We developed two abstractions for this purpose, the **clip-composite**, and the **director**.

---

[9] The recognition of "angry" or "curt" styles could depend on enumeration or settings of "mood potentiometers" attached to the styles, along the lines of "Comic Chat." [Kurlander 1996]

[10] Here and in all subsequent uses of the word "script," we mean to draw as close an analogy as possible to a screenplay, which contains the high level instructions that a director uses to guide his actors.

The clip-composite is, as its name suggests, a subclass of the clip family for representing motions. It serves as the single container into which the motion curves from all motion models for an actor are spliced; thus we can query it for the pose of the actor at any time in which the animation is defined. The clip-composite supports the notions of both sequential and hierarchical composition of motions (in the form of other clips), so that a motion model can add its motion into the clip-composite either segued from or layered on top of the motion of another motion model. It accepts externally created transition generators as sources of the transitions between motion segments from different motion models, and allows motions to be mixed at the granularity of bodyparts (as we would need to successfully layer a *point-at* on top of a *walk*: the motion of the legs, feet, and hips come primarily from the *walk*, while those of the pointing arm and head come primarily from the *point-at*, and others are a more even mixture of the two). These topics and the clip-composite will be discussed in depth in Chapters 6 and 7.

The final abstraction we require is the director, which owns and coordinates all stages of animation production, and all actors participating in the animation. Every object and character in a scene is an actor, and must be placed under a director's control before it can be animated. An actor consists primarily of the physical description of the character (character model – section 3.4.1 and Chapter 4), the partially sorted list of motion models driving the actor (discussed below and in Chapter 7), and the set of active invariants for the actor (section 3.4.2 and Chapter 5). Once the director has control of an actor, it performs the following services for it:

- channels all input from the graphical interface to the appropriate motion models
- can calculate poses and values of handles using inverse kinematics
- initiates recomputation of the animation from its constituent motion models when necessary
- maintains, for each actor, a clip-composite called **rough-cut** into which the raw motion curves from each motion model are assembled, and a clip-primitive called **final-cut** that contains the cleaned up, invariant-satisfied version of the animation

Figure 3.3 shows the flow of control in the animation engine from the time new input is received by the director, to the time the updated final-cut is ready. The director accepts three different types of input: changes to motion model parameters; additions, deletions, and changes in the temporal and layering relationships of motion models in the "script" for any actor; fine adjustments or "tweaks" to the final animation, in the form of changes to invariant values or imposition of keyframes through which the animation must pass. Any of these inputs can come from graphical interaction, a saved animation, or a higher-level script generating entity. We will first explain the workings of the engine in response to motion model parameter input, followed by the differences in response to the other two types of input.

3.5.1 Response to Parameter Input

Step 0: Clear Existing Animation

When a parameter affecting a motion model is changed, the first action the director takes is to clear away all vestiges of the currently computed animation, which includes resetting the final-cut and rough-cut, and removing all active invariants. It may seem wasteful to recompute the entire animation when we only change (for instance) the goal or style of the last move in a long sequence of actions; we do so for two reasons: 1) the parameter optimization performed in the next step may cause propagation of changes in both temporal directions from the altered motion model, 2) our implementation of invariants and other data structures would complicate the process of partially re-evaluating the animation – the performance gains would not be worth the added implementation time, and would not alter the fundamental results. This step is not represented in Figure 3.3 since it is really a "pre-step."

Step 1: Parameter Arbitration

The structure we apply to the motion synthesis problem (motion models, invariants, and parameter classes) allows multiple independent motion generators (*i.e.* the motion models) to communicate with each other in standardized ways. One way we take advantage of this is by optimizing sets of free motion model parameters for the *combination* of motions, as we alluded to in section 3.2 with the *reaching-while-jumping* combination. This optimization, which makes use of invariants and other globally defined handles, is described in Chapter 7. After this step, all motion model parameters except timing values will be set; timing values generally depend on the detailed geometry of the motions, and so cannot be determined in advance of the motions.

Step 2: Execute Motion Models in Partial Order

Once the motion model parameters are set, the motion models must all execute, in order to apply their rules to the generation of new motion curves. A motion model that segues from another must be able to query its predecessor for its pose and time at the transition point (among other things) in order to correctly compute and situate its own motion – similarly for a layered motion model with respect to the motion model upon which it is layered. These dependencies establish a partial evaluation order for the motion models of any given actor; one of the director's jobs is to maintain this ordering, and to execute the motion models according to it in this step. There are four fairly distinct stages of a single motion model's execution, as shown in Figure 3.3.

Step 2a: Recompute Motion Curves

The first and generally most substantial phase of motion model execution is the rule-based translation of parameter values and base motions into a clip that performs the action of the motion model. We will describe this process in general in Chapter 5, and give specifics for all motion models we implemented in the Appendix. One important point to note is that the motion contained in the clip, while generally (but not necessarily) satisfying its goals and invariants at its key timepoints, will not satisfy the invariants at all

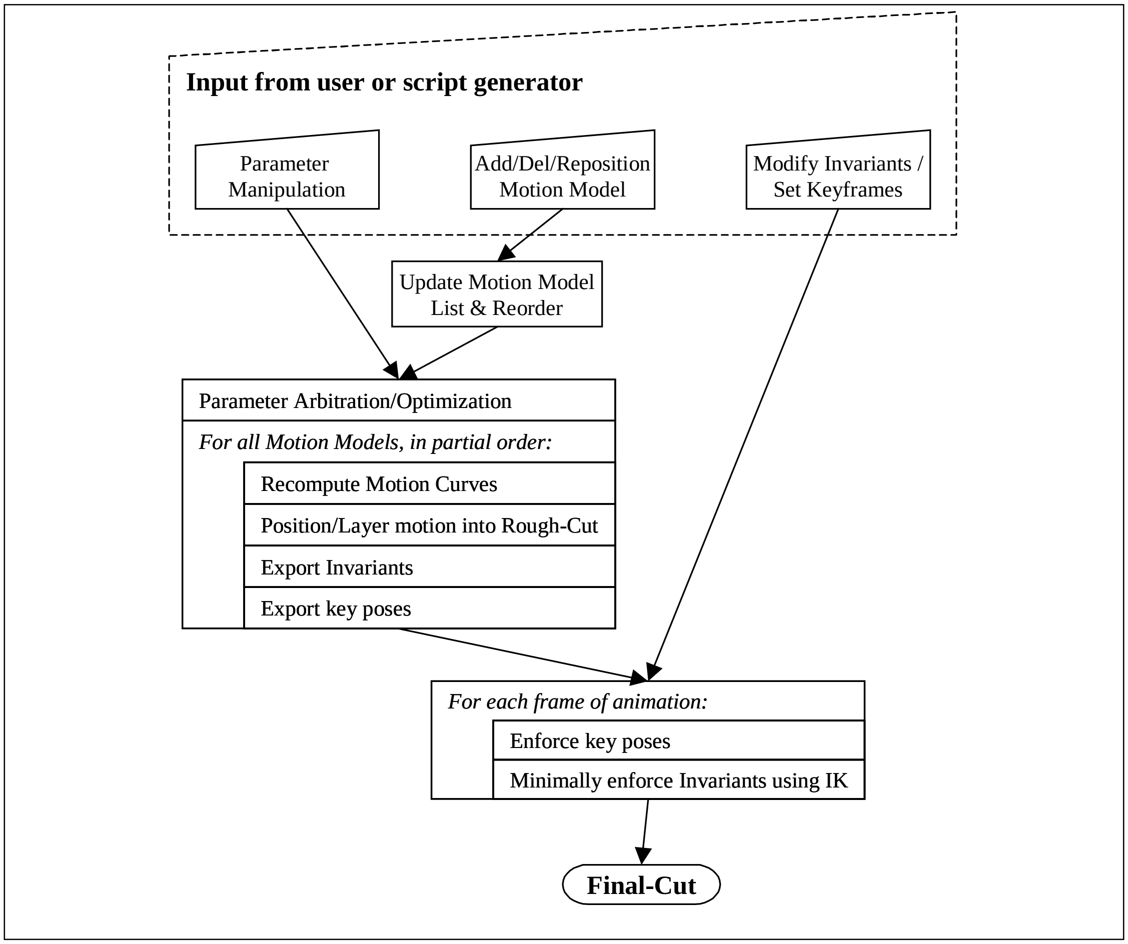


Figure 3.3: Control Flow. Input to the animation engine causes the final quality animation (final-cut) to be recomputed. Motion model parameter changes cause the motion model script to be re-executed, while additions, deletions or relative changes of the constituent motion models first require the execution order of the script to be updated. Once the individual actions of all the motion models have been spliced into one continuous animation, rough-cut, invariants are satisfied at every frame to produce final-cut. Invariants and keyframes may be set without requiring motion models to recompute their primitive motions, since they affect only the cleaning up of rough-cut into final-cut.

times. This is by design, since we only wish to run our constraint satisfaction engine on rough-cut, after all motion models have contributed both their motions and invariants to the mix.

Step 2b: Position/Layer Motion into rough-cut

Once the general motion has been computed, we must situate it in the composite motion, rough-cut. To do this we must first compute the motion that will smoothly transition from the previous (if segued) or underlying (if layered) motion to the current. The motion model's transition generator performs this task, as well as determining the duration of the transition, as we will describe in Chapter 6. With the transitional motion generated and all the timing parameters calculated, we can then add the motion into the clip-composite, rough-cut.

Step 2c: Export Invariants

Even during the calculation of its motion curves, the motion model is aware of all the invariants that must be maintained through all its various phases. However, in order to satisfy the invariants simultaneously with the invariants of any other motion models that may be layered with the current, we must first export the invariants of *all* the motion models to the director, so that it can initiate a single constraint satisfaction sweep through the animation *after* all motions have been recalculated and added into rough-cut.

Step 2d: Export Key Poses

Our main facility for fine-grained control of the final animation is to allow the user to place an unlimited number of keyframe poses through which (subject to continued invariant satisfaction) the actor must pass at specific times. In order to satisfy these additional pose constraints while still preserving the style of the motion produced by the motion models, we use space-warping on top of rough-cut, *before* it is passed through the constraint satisfaction engine, and then also add the pose constraints to the set of constraints that must be satisfied by that engine. Space-warping helps to preserve animation continuity, and maintains the high frequency content of the motion models' results, while the imposition of the poses as constraints guarantees that the poses will actually be met, if they are consistent with the invariants.

As we will discuss in the next Chapter, one of the secrets of successful use of space-warping is properly setting the range of influence (in time) of each warp key. In most cases, when working with a space-warping tool, an animator must place at least three warp keys to achieve a desired affect (one for the actual modification, and two to bracket its area of effect). However, the knowledge motion models possess about the important key-times within their actions can serve to automatically bracket most modifications the user would make[11]. Therefore, the last phase of a motion model's execution is to compute the desired pose at each of its key-times and export those poses as bracketing keys to the clip-space-warp layered on top of rough-cut. The keyframe poses we have been talking about, once initially created, are attached to the base motion model whose timespan intersects the time of the pose, and these poses are also exported by the motion model at this time.

Step 3: Cleaning up the Motion

Once all the motion models have executed, rough-cut contains the "general" motion, but neither invariants nor user-placed keyframe poses may be satisfied everywhere. The most general method of accomplishing this would be to use a spacetime constraints optimization, like that of Gleicher [Gleicher 1997], or the theoretically cheaper IK-based hierarchical optimization of Lee [Lee 1999]. However, we have achieved satisfactory results using an even simpler and cheaper method that performs a single IK pass

[11] By simply adding a separate clip-space-warp on top of the "automatic" one described here, we can enable the user to place all his own brackets while tweaking; he would simply select the layer into which his space-warp keys are placed.

through a version of the animation that has already been space-warped for continuity. We will describe the method and motivation for this approach in Chapter 7.

Step 3a: Enforcing Key Poses

This step is really just the setup phase of the constraint enforcement. Since the key poses have already been placed into the clip-space-warp on top of rough-cut, all that remains is for the director to make sure that all the proper invariants and other constraints are activated at each frame of the animation.

Step 3b: Minimally Enforce Invariants using IK

Finally, we produce the "ready for rendering" version of the animation, final-cut, one frame at a time by running an optimization-based inverse kinematics algorithm (described in the next chapter) on the pose in the corresponding frame in the space-warped rough-cut. The IK algorithm changes the pose minimally so that it satisfies all the constraints active at the frame.

Our end result, final-cut, is then a point-sampling (in time) of the true animation, which can be resampled at any time using interpolation. Our sampling rate is 30 frames per second, which could result in some loss of detail for very fast motions; however, it would be a simple matter, affecting no other component of the engine other than the director itself, to increase the sampling density to any desired level.

3.5.2 Response to Other Input

Changes to the Motion Model Script

The second type of input to our animation engine affects the existence of motion models and their temporal and layering relationships. When the user adds or removes an action, changes the ordering of actions, or changes a transition relationship, the animation must be recomputed just as it was for motion model parameter changes. However, these modifications affect the partial evaluation ordering for the affected actor, so, as is shown in Figure 3.3, before re-executing the motion models, the director must update the partial evaluation order of the motion models.

Modifying Invariants & Placing Keyframe Poses

The third and last type of input to our engine is modification of the keyframe poses, which we discussed in step 2d above. We neglected to mention there that, in addition to instantaneous pose constraints, the user could also manually set the values of any invariants. For instance, if one of an actor's footfalls while walking needs to be just a bit more to the right, the user can drag the foot at any frame within the footfall, and because an invariant governs the footfall, the position of the foot will be adjusted at all frames during the footfall.

All of these changes affect only the constraint satisfaction (Cleanup) step of the animation engine, so the response to them skips the motion model re-execution step.

4 Basic Tools and Abstractions

In this chapter, we describe all of the basic concepts and tools upon which the “core” of the thesis builds. This material breaks down into four main areas:

1. abstractions for the “physical” representations of virtual 3D characters
2. issues surrounding representations of motion
3. the design and functionality of motion transformation operators
4. computations involving actor poses

Many of these topics have been addressed in the literature before; in our presentation, we will discuss the needs and experiences that drove us towards the designs we developed, and note novel contributions where applicable.

4.1 Character Models and Actors

Character models and **actors** are closely related abstractions in this thesis. A character model defines the geometry and animatable structure of a virtual 3D character, and defines a number of operations and services associated with characters. These include facilities for posing and moving the character. An actor is essentially a character model plus motion. As the name implies, an actor is all about a performance: it contains a character model defining the physical representation of the character, as well as the static or developing script that specifies the actions to be performed by the character in an animation. We will now describe each in more detail. Please note that the material in section 4.1.1 is a review of established techniques for character modeling, not a contribution of this thesis.

4.1.1 Character Models

Since we are primarily animating humanoids in this thesis, we will gear our discussion of character models to them; however, the logic and algorithms apply equally well to robots, most animals, and rigid bodies. Ignoring, as we are, the "soft body" characteristics and motions (such as muscles, fat, and skin), the broad motion with which this thesis is concerned maps almost directly to the motion of the underlying skeleton. Therefore, a good representation of the skeleton will suffice to represent the motions in which we are interested. The relevant characteristics of a human skeleton are that it is comprised of rigid links that are connected to each other by rotational joints that do not translate with respect to each other (to a very good approximation). Thus the entire skeleton assembly has an overall position and orientation, and many connected, rigid parts that pivot, twist, and swing with respect to each other.

We could choose to model a skeleton very physically, starting with a collection of bones represented as rigid 3D models each of which has a position and orientation, and attaching them together with explicit constraints at the joints that would prevent them from separating or from rotating in illegal ways (for instance, a wrist cannot twist). With this model, the degrees of freedom (DOFs) of the character are the positions and orientations of each bone in the body. This is awkward because anytime we move any bone (through even the simplest of control techniques), we must reapply and solve (potentially) all of the constraints that hold the bones together.

A more convenient representation imposes a hierarchical relationship on the bones, in which the position and orientation of each bone is specified relative to a "parent" bone. Since we have stipulated that bones do not slide with respect to each other at the joints, a bone's configuration at any time can be specified by a fixed translation (thus not involving any DOFs) and a rotation relative to the parent bone (which is in turn defined in terms of *its* parent, all the way up to the root, which has full translational and rotational DOFs). In effect, we have eliminated the need for constraints to hold the bones together by implicitly removing the translational DOFs from all bones in the hierarchy save the root. This behavior is exactly what we get from a *transformation hierarchy*, such as that implemented in OpenGL [Neider 1993] or Inventor [Wernecke1994].

We refer to our own implementation of transformation hierarchies as **character models**. They support all of the standard functionality of hierarchies, such as computing positions and orientations of points on the body (for example, sensor positions in a motion capture setup), and displaying some version of the character[12], which itself performs mainly “compute current position of point on body” operations on a hierarchy. Let us say we wish to compute the current position of the center of a character’s hand in a 2D stick-figure hierarchy (all math and procedures extend straightforwardly to 3D). Define $\mathbf{p}$ as the position of the center of the hand with respect to the wrist joint, which is the center of rotation for the hand. We compute the current, world space coordinates $\mathbf{p}_w$ of the hand by computing and concatenating transformation matrices at each joint between the point’s origin bodypart (here, the hand) and the root of the hierarchy, which is often set to be the hips.

$$\mathbf{p}_w = \mathbf{T}_w(x, y)\mathbf{R}_w(\theta_w)\,\mathbf{T}_s\mathbf{R}_s(\theta_s)\,\mathbf{T}_c\mathbf{R}_c(\theta_c)\,\mathbf{T}_b\mathbf{R}_b(\theta_b)\,\mathbf{T}_f\mathbf{R}_f(\theta_f)\,\mathbf{T}_h\mathbf{R}_h(\theta_h)\,\mathbf{p}$$

The above equation is the formula we use to compute the world space position of the hand, and represents a matrix-vector multiplication, where the matrix is a product (composition) of many matrices. The sequence of **T**ranslations and **R**otations beginning on the right and moving to the left are the fixed **t**ranslation that positions the **h**and’s base at the origin and **r**otates by θ_h , then positions and rotates the **f**orearm, then on to the **b**icep, **c**hest, **s**tomach, and finally, the global rotation and translation at the **w**aist. This computation can be performed rapidly for many different points on the character’s body by recursively traversing the hierarchy using a transformation stack like the one provided by OpenGL. For greater detail on the mathematics of transformation hierarchies, see Foley [Foley 1992].

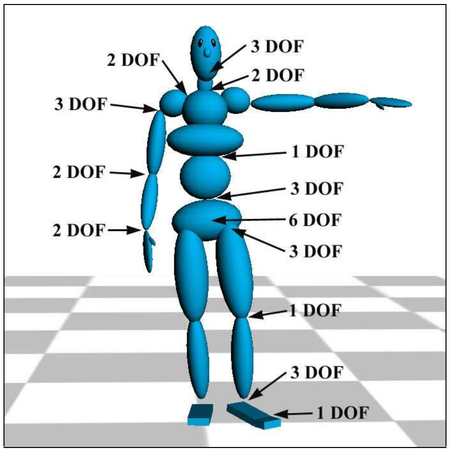
Our character models go beyond the basic hierarchy functionality to embed several kinds of knowledge that aid in both defining and animating characters. We will now describe each kind, and in the process define the concrete character used to create and perform all of the animations in this thesis.

4.1.1.1 Joint Types and Limits

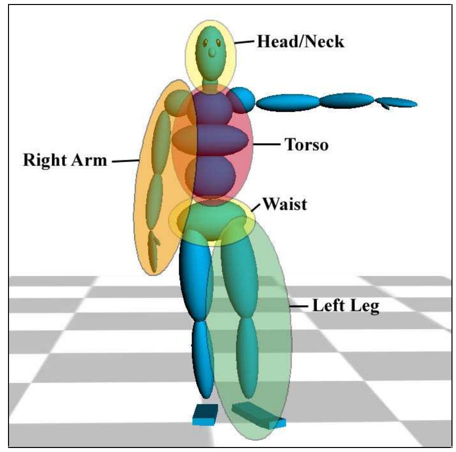
To aid in the process of defining character models and placing meaningful limits on their legal motion ranges, we have pre-defined a number of types of joints that can be used to attach bodyparts in a transformation hierarchy. A *Joint* is a specific sequence of transformations that allow for movement in some or all of the six rigid-body DOFs, *plus* a suitable specification of angular limits on the allowable motion of the joint, tailored to the needs of articulated systems. We have defined the following types of joints:

- **Rigid-Body:** This joint provides all six rigid-body DOFs, and does not allow any limits on the motion. We use this joint for all simple, rigid objects, and at the root of more articulated characters, such as humanoids.

[12] We need not model or draw actual bones. The hierarchy itself gives us the requisite rigid-limb behavior, and we are free to draw any geometry to represent the limbs, and even, as previously mentioned, layer more sophisticated “secondary” structures on top of the hierarchy, such as muscles whose shapes are driven by the joint angles. However, it is actually desirable *not* to draw faithful reconstructions of the bones, since this might draw the viewer’s or animator’s attention away from the study of motion itself.



(a) Joint DOFs



(b) Bodypart-Groups

Figure 4.1: Physical Structure of Humanoid. In part (a) we label every joint (save the duplicates in the left arm and right leg) with the number of DOFs it possesses in our humanoid model, which also uniquely identifies which of our four Joint types it represents. At the elbow, the two DOFs are the bend of the arm and the twist about the axis of the lower arm that in humans actually occurs midway down the forearm. The top two segments of the torso form a single rigid unit that is allowed to bend forward/backward with respect to the abdomen. The neck can pivot but not twist. In part (b) we show the constitution of the Bodypart-groups that define the granularity at which we mix motions, discussed in section 4.1.1.2.

- **Hinge:** A Hinge joint is a one DOF joint that allows for limited (or unlimited) motion about a single, specified rotation axis. We use Hinge joints at a humanoid's knees.

- **Pivot:** A Pivot joint allows for rotation about any two of the three rotation axes at the joint. For example, at a humanoid's wrist, we use a Pivot joint to allow (in Euler angle parlance) yaw and pitch, but no roll.

- **Ball-and-Socket:** This joint recognizes the need for sophisticated joint limits to accurately model joints like shoulders and leg/hips. It breaks the motion into two components, each of whose ranges can be limited independently: a twist about the major joint axis (*e.g.* the upper-arm bone), and a two DOF swing of that axis. We describe this joint in detail in our previous work on rotation parameterizations [Grassia 1998].

With these joint types one can rapidly define a character that behaves fairly well under inverse kinematics operations (inverse kinematics is the main reason for the necessity of joint limits). We have constructed a character model of a humanoid male possessing 49 DOFs, which we show in Figure 4.1(a), along with the specifications of all of its joints. Because we endeavored to make the process of mapping our motion captured data to a character as simple as possible, and because we did not want to address the problem of mapping motion from one character to another, we used this character model for all of our motion models and examples.

4.1.1.2 *Classes Can Be Hierarchical*

Everything we have described so far is applicable to any generic character model that we may create. In terms of abstractions, a character model is a generic class; however, we can use the concept of sub-classing to embed useful knowledge about the character's structure and manipulation into the character model itself. For instance, we have defined a sub-class of character model called **humanoid** that defines the following information for any character created using it:

- **Bodypart Structure:** Humanoid specifies that there are a fixed number of articulated bodyparts in a humanoid, and gives them names that can be used for easy retrieval and specification of constraints and joints. The twenty-one named bodyparts are: Waist, Abdomen, Chest, Neck, Head, L_Thigh, L_Shin, L_Foot, L_Toes, R_Thigh, R_Shin, R_Foot, R_Toes, L_Shoulder, L_Bicep, L_Forearm, L_Hand, R_Shoulder, R_Bicep, R_Forearm, R_Hand.
- **Named Groups:** A crucial concept to our treatment of motion combinations is the structural granularity at which we allow them to be mixed. That is, do we allow a motion to be constructed by taking the motion of the wrist from motion A, the motion of the forearm from motion B, *etc.*, or, at the other end of the spectrum, dictate that all motion comes from motion A or from motion B? If it is feasible, it behooves us to find an intermediate granularity, since the smaller we make the granularity, the greater the overhead we incur in specifying and evaluating motions. By studying the ways in which various bodyparts commit themselves to different actions, we grouped the 21 humanoid bodyparts into seven groups: Waist, Torso, Head_Neck, Left_Leg, Right_Leg, Left_Arm, Right_Arm, which we have enumerated in Figure 4.1(b).
- **IK handles:** A specialized class such as humanoid defines, for each bodypart, standard position, orientation, and direct joint access handles for each bodypart to facilitate creating constraints used in IK. These handles (specifically, the position handles) can be tailored to the geometry of each actor instantiated (see below) from humanoid, which allows motion models and other high-level entities to manipulate motions without needing to know the exact geometry of the actor. For instance, we define the standard position handle of the foot to be the "ball of the foot", which can then be used, regardless of how thick, wide, or, long an actor's foot actually is, to place the foot firmly on the ground and pivot it with predictable results.

4.1.2 Instancing Actors

Although it is more specific than a character model, a humanoid is still just a *class*, which is to say it is not a concrete object. In order to animate a character, we must first instantiate a concrete actor from an abstract character model (or sub-class such as humanoid). This instantiation process adds several elements crucial to animation:

- **Geometry:** During actor instantiation we add the actual, movable geometry to the character model. In our prototype system this amounts to adding simple geometric primitives with mass estimates, for a result like that shown in Figure 4.1. Ultimately, the geometry would likely be a deformable mesh that acts as a skin that moves along with the underlying "bone structure" that the character model defines.
- **Handle Annotations:** We also add concrete values for each of the IK handles defined by the specific sub-class of character model.
- **Script & Final-Cut:** Finally, each actor possesses and maintains its own animation script – the dynamic set of motion models that specify what the character will be doing at every point in time. Corresponding to the script is the actor's reserved place in the *final-cut*, the frame-by-frame final version of the animation in which all invariants have been satisfied.

4.2 Motion Representation

The number and nature of a character's DOFs, as specified by a character model, determine the nature of a *pose*: a pose is a complete set of values for a character model's DOFs, resulting in a total-body posture. We can then define an animation in terms of poses: an animation is a pose-valued function of time. The mathematical representations we use for poses and the motion functions comprising an animation have a fundamental impact on the complexity, stability, and performance of the manipulation and deformation algorithms at the heart of this thesis. We have examined many of the issues one must weigh when choosing suitable representations, presenting our findings elsewhere [Grassia 1998][Grassia 2000]. In this section, we will describe the representations we use in this thesis, along with rationale for the major components.

4.2.1 Poses

A character model may utilize many different types of joints (section 4.1.1.1), and thus require many different types of DOFs; for example, a Hinge joint requires a single scalar Euler angle, while a Rigid-Body joint requires a 3D vector for position, and a 3D exponential map vector [Grassia 1998] for the free rotation. The simplest possible structure for a pose would result from simply packing all of the varied DOFs sequentially in a prescribed order into a single, high dimensional vector. This representation is efficient of both time and space: it uses the fewest numbers possible to encode the pose, and realizing the pose from this mathematical representation simply involves unpacking the vector back out into the individual joints. It is, however, ungainly in several respects pertinent to motion manipulation: first, when we try to construct motion functions from it, each DOF type will require its own interpolant and blending operators; second, if we change the type of any joint to add or subtract a DOF (either in the course of developing the character or designing a similar one), all previously computed poses and motion functions become invalid.

We choose instead a more unified representation, based on the conclusion that most, if not all, of the characters we are currently interested in animating can be modeled as transformation hierarchies whose roots possess what we call a Rigid-Body joint, and all of whose interior joints are purely rotational. This being the case, we represent every joint the same way – as a single quaternion representing a rotation – and complete the pose with a single 3D position vector for the translation at the root. This representation is less efficient, since a one-DOF Hinge joint uses an entire 4D quaternion. In addition, each joint type must be able to convert to and from quaternions and its native set of DOFs. However, these are small matters compared to what we have gained: an elegance and simplicity for all of our many motion transformation and deformation algorithms, and complete freedom to restructure and adapt the internal joint structure of our character models without losing or invalidating all previously recorded poses and motions. We have found the latter property particularly fortuitous, as we discovered several times over the course of the thesis work that one or another of our character's joints lacked sufficient DOFs to execute certain motions, and were able to painlessly change the type of one or more joints to add the necessary DOFs.

Particularly in light of our promotion of the exponential map as a rotation representation [Grassia 1998], it may seem odd that we use quaternions to represent joints. There are two reasons for this choice: quaternions have an established calculus that is well understood; quaternion interpolation (SLERP [Shoemake 1985] or more sophisticated techniques) still provides the best interpolation of rotations.

4.2.2 Motions, Motion Functions, and Clips

Given a physical structure for character models and representations for DOFs and poses, we are ready to define our ultimate basic data type, the motion. We could simply define a motion as a sequence of poses or as a pose-valued function; however, although we do frequently access the motion in this fashion, most of our manipulation and combination operators require that we also be able to address a motion in units smaller than an entire pose. The most common subunit of a motion is a motion function that Witkin and Popović call a *motion curve* [Witkin 1995], which defines the value of a single DOF over the entire time span of the motion. However, since, as we have already noted, 3D rotations are not well-described and animated by collections of scalars and scalar-valued functions, we specialize motion curves into *Quaternion Curve* and *V3 Curve*. A Quaternion Curve is simply a quaternion-valued function of time that can accurately represent the motion of a one, two, or three DOF rotational joint. A V3 Curve is a function of time that returns a 3D (position) vector. All of the motion transformation algorithms we will describe in the next section can be expressed easily in terms of these motion curves.

We can now define our basic unit of motion, the **clip**. A clip is an abstract data type that can represent an arbitrary-length motion for a particular type of character model – a clip can be instantiated for any type of character model, and is then associated with that type alone for its lifetime. This association is necessary because the particular type of character model specifies the ordering and number of joints, which the clip needs to determine the number and meaning of its constituent motion curves. The basic properties and functionalities defined for all subclasses of clip are the following:

- Every clip contains or deforms a Quaternion Curve for each joint, plus a single V3 Curve for the global translation of the character's root.
- These motion curves are organized according to the bodypart-groups discussed in section 4.1.1.2 and Figure 4.1(b). A clip may define only a partial pose or mix other motions by body-part-group – for example, a clip may contain motion for only the Head_Neck and the Right_Arm, in which case it will contain *all* the motion curves associated with those groups. It is not possible for a clip to contain, for example, motion curves for the left wrist and elbow without also containing the curves for the left shoulder and collarbone.
- A clip can load its pose or partial pose into any compatible character model at any time in which the clip is defined.
- A clip can return the value of any joint or the global translation at any time in which the clip is defined.
- A clip can return the time interval over which it is active.
- Clips are composable (exactly how depends on the specific sub-class of clip), allowing us to easily create and destroy sophisticated motion expressions.

The last point is truly the crux of the clip, because it unifies all motion transformations into a function-composition language that treats transformations and deformations of motions the same way it treats primitive motions. Each type of transformation we wish to support (warping, blending, *etc.*) becomes a sub-class of clip that, in addition to performing its unique deformation on some set of component clips, itself supports the same generic clip operations listed above.

4.3 Our Versions of Motion Transformation Algorithms

In this section we describe all of the motion transformation algorithms we use in this work, discussing, where appropriate, differences from "typical" implementations of such algorithms and problems we encountered. Most of the basic algorithms are part of the clip motion language. The last, however, is inverse kinematics and constraints, which is an integral part not only of our final-quality motion synthesis engine, but also of many of the transformation algorithms that are clips. We deal mainly with the general function of each algorithm and specific problems in this chapter – for more detail on the clip structure and parameters, please see the appendix.

4.3.1 Primitive Motions

The raw materials for all motion expressions in this work are previously animated motion clips[13]. The **clip-primitive** is a sub-class of clip that can read a motion from a file in the form of motion curves. The motion curves can be either cubic splines representing keyframed animations, or frame-by-frame samples from motion capture. To reduce file size and memory footprint, we actually store rotations as 3D exponential map vectors; when the motion curve is queried for its value at some time, it converts the values of the surrounding frames into quaternions and SLERPs to get the desired quaternion value.

We can also use a clip-primitive to store an animation we have dynamically generated. After the clip-primitive has been created with a specified number of frames' worth of storage, we can record an animation by placing the actor associated with the clip in the pose he should have at frame x and then tell the clip-primitive to capture the pose from the actor at frame x for all x in the desired animation. The "final-cut" we introduced in the last chapter is a clip-primitive, which we use to store the final version of the animation we are creating.

4.3.2 Warping

Our most heavily used deformation tools are derived from what we described in chapter 2 as motion warping. We implemented space-warping, time-warping, and pivoting, encountering numerous problems not addressed in the literature. We discuss these problems and present our solutions, but still believe more work is required on these techniques.

4.3.2.1 Space Warping

For general-purpose geometric deformation of animation clips, we implemented motion warping, which we described in Chapter 2. However, because (with the exception of the global translation) our motion curves are quaternion curves in S^3, we cannot simply add displacements. Instead, we must use quaternion multiplication, the (rough) equivalent in S^3 of addition in R^3. Therefore, given initial motion curve $\mathbf{q}(t)$ and displacement curve $\mathbf{d}(t)$, both of which, in our implementation, are spherical Beziers [Shoemake 1985], the resultant motion warp is:

$$\mathbf{q}'(t) = \mathbf{d}(t) \circ \mathbf{q}(t)$$

Where '$\circ$' denotes quaternion multiplication, and the order of the operands is significant because quaternion multiplication does not commute. We have encountered several issues not well explored in the literature while trying to use motion warping effectively. For the purposes of this thesis, we would *like* to assume that motion warping, along with the other low-level transformation operators, is a well-oiled black box that we do not need to develop significantly. Since this is not the case, we will briefly explore the most

[13] There is no reason why we cannot use procedurally generated motions, but we have not explored them.

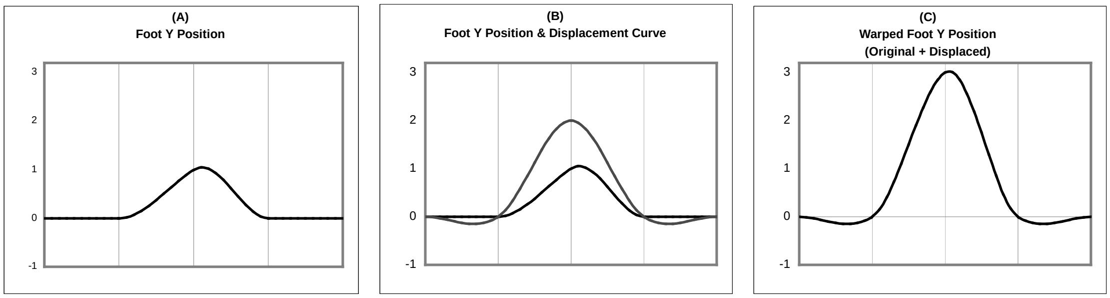


Figure 4.2: Anticipation/Overshoot in space-warping. These figures depict foot height over time. The vertical lines show where the warp keys have been placed. In (A) we see the original height function. We decide the foot actually needs to achieve height 3.0 rather than 1.0 at the middle key; this results in a motion warp with value 2.0 at the middle key, and 0.0 at all the other keys. In (B) we have overlaid the displacement curve computed by the motion warp: the requirement of C1 continuity at the warp keys causes negative displacements in anticipation of and following the positive displacement. In (C) we see that this results in the foot sinking below the floor in the displaced height function.

significant issue we encountered, although we do not claim to have the best solution, and believe more work is required.

From our own and others' limited investigations, it seems that our perceptual systems are very adroit at noticing sudden changes in motion. This fact is one of the motivations behind the typical formulation of motion warping, in which the displacement consists of one or more continuous functions that possess the same domain as the motion they are modifying. When a new warp key is introduced, or an existing key modified, the tangents at the adjacent keys are automatically adjusted to preserve continuity of the displacement function, as shown in Figure 4.2. This causes a combination of anticipation and overshoot in the motion, due to the segments preceding and following the modified segment in the motion warp. For instance, imagine that the displacement in Figure 4.2 is to be applied to the height of an actor's foot, which in the original motion is flush with the ground. Clearly, then, the negative displacements in the map during the segments preceding and following the modified segment will cause the foot to pass through the floor, which we do not desire. In this particular example, we can repair the damage by applying per-frame IK afterwards, but we have encountered many other situations in which we cannot easily repair this damage. There are times when this behavior *might* be desirable, such as when we increase the extension of, say, a tennis swing with motion warping, and would not be dismayed to see a deepened windup and follow-through of the swing.

This means that we must solve two problems: how can we create motion warps that contain no anticipation/overshoot, and when do we choose to use such maps rather than the "standard" maps? We have two choices in addressing the first problem, both of which result in the loss of C1 continuity, but one of which maintains G1 continuity. As shown in Figure 4.3(B)**,** one solution is break the tangents at the keys surrounding the edited or inserted key: the "outside" tangents are left unmodified, while the "inside" tangents are calculated using Catmull-Rom rules. This remedy sacrifices all first-order continuity of the motion

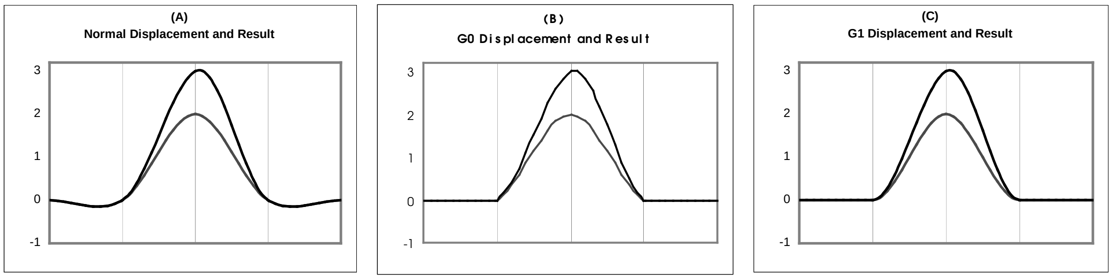


Figure 4.3: Eliminating Anticipation/Overshoot. In part (A) we show the same motion warp (lower curve) and warped result (upper curve) as in Figure 4.2. In part (B) we eliminate the anticipation/overshoot by introducing a G1 discontinuity at the keys on either side of the modified key. Note that this results in a visible tangent discontinuity of the resulting warped curve. In part (C) we instead maintain G1 continuity in the motion warp, which preserves continuity of the original in the results.

warp, but produces the same motion inside the interval in which the key is placed as that of the original technique. The second solution, also shown in Figure 4.3(C), also leaves the outside tangents unchanged, but rather than recomputing new inside tangents using the newly inserted or modified key, simply scales down the equivalent of the outside tangent for use as the inside tangent. This solution maintains G1 continuity in the motion warp, but will generally cause a sharper rise in the motion towards the inserted key's value. It also introduces a new DOF, the scale factor for the inside tangents, which allows a tradeoff between more natural rise-times and continuity of the map.

The question of when to use each technique is substantially more involved, and we have not explored it deeply. Currently, we employ the traditional motion warping for defining the initial, key-time warp on the motion produced by motion models (this will be explained in Chapter 7). When this motion warp must be altered in response to user-placed tweaks, we use the G1 method above, by default, and allow the user to switch to either of the other two methods if desired.

We implement this algorithm in the **clip-space-warp**, which allows full-body warp keys to be placed by specifying a desired pose at some time, or warps on individual bodyparts by specifying either the desired value or displacement.

4.3.2.2 Time warping

Time-warping addresses the problem of altering the original timing of a motion by either stretching or compressing various portions of an animation in the time domain. When formulating space-warping, we needed to implement the warp as a *displacment* that we added onto the original motion curve, because the motion curve possessed a complicated shape that we wished to preserve. Warping in the time domain is simpler in that the function/signal we wish to warp is the simple linear scalar function $y(x) = x$. Therefore, rather than compute a displacement to add onto this function, we simply form a direct mapping between warped time and unwarped time like so: given a set $\alpha_1 .. \alpha_n$ of original key-times and a set of corresponding desired final key-times $\gamma_1 .. \gamma_n$, we produce a smooth, invertible mapping that maps from the *warped final* *key-times* to the original key-times and also continuously maps all times in the final domain into the original range. We compute the warp with the warped times as the domain because nearly all accesses of the time-warp are requests for motion curve values in final animation time (*i.e.* warped time). In the "Motion Warping" paper [Witkin 1995] the authors use cardinal splines to interpolate the key-time constraints, noting that this may cause time in the range of the warp (*i.e.* original time) to run backwards as time in the domain moves forward. They claim this is not a problem and may even be desirable. In our applications, however, where we will frequently use time-warping to completely align multiple base motions so that we can mix them together, we cannot allow time to run backwards.

This means we require that the time-warp induced by the keys, mapping final times to original times, be either monotonically increasing or decreasing. To meet this constraint, we first check to make sure all of the time-warp keys are monotonically increasing or decreasing – if not, we signal a major problem, as the creator of the time-warp is not using it correctly. We then create an initial time-warp function by fitting Catmull-Rom splines through the key-times. As with natural splines, this may cause time to run backwards, even with monotonic keys. Therefore we check each piece/segment of the spline, reducing the magnitudes of the tangents until the segment is monotonic.

Given our starting assumptions that all of the key-times are monotonically increasing or decreasing, and that the tangents for the cubic spline segments that interpolate the keys were initially computed using Catmull-Rom rules, then the condition on monotonicity of the segment is that the extrema of the cubic interpolating function lie *outside* the intended domain of the segment, *i.e.* beyond the keys that define and bound the segment. This condition is sufficient because if an interval of a function contains no extrema, then the function cannot "change direction" in the interval – it must monotonically increase, decrease, or remain constant (if the two endpoints of the interval possess the same value). Furthermore, if there *are* extrema inside the segment, they can be moved outside the segment (resulting in monotonicity) by reducing the magnitude of the tangents of the cubic spline. Since the monotonicity condition is non-linear, we did not attempt to derive an analytical solution to this problem. Instead, we iteratively reduce the tangent magnitudes and recheck the monotonicity of the segment. Each iteration involves few operations, and in practice we require only a few iterations to fix bad intervals. Furthermore, this operation need only be performed once for each time-warp. We note that this process trades off higher-order continuity of the warp function for monotonicity; we have not noticed this to be problematic.

We implement this method of time-warping in the **clip-time-warp**, which, in addition to methods for setting up a time-warp and effectively resampling the clip on top of which it is layered, contains methods for mapping from original time to warped time and vice-versa.

4.3.2.3 Pivoting

One of the most common deformations we make is to reposition and reorient motions. Every base motion is performed with the actor standing in a certain spot and facing in a specific direction. Before we can even think about blending base motions, we must ensure that all the base motions for a particular motion model are aligned – the actor starts out in as close to the same position and orientation as possible. Further, once we have blended the base motions, the very first deformation we generally make to the result is to reposition/reorient it according to the position and orientation parameters of the motion model.

One of the first things we discovered in our initial experiments with motion warping is that trying to accomplish this repositioning (which we refer to as *pivoting*) using straight motion warping does not work for the majority of motions of character models whose roots are their hips. The reason for this is that the pivoting operation involves displacing both the global orientation and global translation of the actor, and the displacement in rotation will be centered, by default, at the origin of the space in which the motion is defined, ignoring the interaction between global position and orientation on the pose. This results in a pivoted motion that faces in the correct direction, but slides around in ways the original motion did not.

What we must do is pivot (*i.e.* rotate) around a fixed point of the original animation, and only then perform any repositioning, if it has been specified. We therefore define a **clip-pivot** that takes, in addition to the clip to be pivoted, three arguments: **pivotPoint**, the position of a fixed point (usually the ball of one of the actor's feet) on the actor's body at the beginning of the clip; **rotate**, specifies how much to rotate around the pivotPoint; **finalPos**, the position to which pivotPt should be moved after applying the rotation. The deformation created by a clip-Pivot affects only the global position and orientation of its base clip, like so:

$$\begin{aligned}&\mathbf{rot}'(t) = \mathbf{rotate} \circ \mathbf{rot}(t)\\&\mathbf{pos}'(t) = \mathbf{finalPos} + \left(\mathbf{rotate} \circ \left(\mathbf{pos}(t) - \mathbf{pivotPoint}\right)\right)\end{aligned}$$

In other words, rather than add independent displacements to the global position and orientation motion curves, the above formulae rotate the position motion curve about a fixed point (pivotPoint) by the same rotation applied to the orientation motion curve before adding on the position displacement.

4.3.3 Blending/Interpolation

The blending of multiple motions is an integral part of our fundamental motion synthesis algorithm, both in primitive motion generation and in combining primitive actions together. Our choice of quaternion motion curves complicates this problem, because the only published, robust means of directly blending more than two quaternions is the sophisticated radial basis function blending presented by Rose [Rose 1998]. Rather than trying to adapt that technique to our needs, we decided to implement only binary blending, which can utilize the simple SLERP [Shoemake 1985]. To blend more than two motions, we construct a binary tree of blends. For instance, assume we have a binary operator SLERP(A, B, perc) that produces a motion-curve-wise blend that is 'perc' percent of the way from motion A to B. Now, if we have three motions, A, B, and C, and we wish to produce a blend that is equal parts of all three, we could do it like so:

SLERP(C, SLERP(A, B, 50%), 33%)

We devised two different blending operators from the binary blender – one that varies the blend over time, and one that varies the blend over bodypart-groups. The **clip-blend** operator allows its operand clips to be precisely temporally situated with respect to each other, defining some region of temporal overlap. For each such overlap created, one also provides a scalar function whose domain is the temporal overlap region, and whose range is [0, 1] (we use Hermite curves); this function specifies (for all motion curves) the blending percentage between the first operand clip and the second at each moment. Thus we can create blends that vary over time – this is particularly useful for stitching a cyclic motion together from different (but related) components. As we will see later, we also use this operator for the *jump* motion model.

Our second blending operator is the **clip-mix**, which takes, in addition to its two clip operands, a vector that specifies a constant blend factor for each bodypart-group of the clips. Our primary use for the clip-mix is to layer some amount of a motion on top of another motion – for instance, to layer a hand-wave on top of a walk: although the hand-wave may define motion curves for the hips and legs, we almost certainly do not want to include any contribution from them to the mix/layering; we will include some contribution from the torso, and a large (possibly 100%) contribution for the waving arm and head/neck. This blend factor can be in the range [0, 1], producing the same blending behavior as that of the clip-blend. For example, to mix the *wave* motion with the *walk*, where a blend factor of zero results in pure *walk* and a blend factor of one results in pure *wave*, we might use the following vector:

[hips = 0, r-leg = 0, l-leg = 0, torso = 0.3, head-neck = 1, r-arm = 1, l-arm = 1]

Sometimes when layering, however, we do not care about the *absolute* poses contained in the layered motion, but wish only to capture its relative or high frequency motion – our *difference* blend allows us to do just that. By specifying a blend factor in the range [100, 101], we signify that instead of directly blending the second motion with the first, we *add on* to the first motion, at each time *t*, a weighting of the difference between the pose at time *t* and the pose at time 0 of the second motion. The weighting for this difference is (blend factor – 100). The rationale for this "difference" mode of blending comes from the motivating application of the clip-mix, the layering of motions. We can use the difference blend to extract the "shiver" from a motion in which the actor holds a pose while shivering, and layer the shiver onto any other motion.

4.3.4 Reflection

We often want to reflect a motion along some axis in the world space, usually along the axis that corresponds to an actor's right/left axis. We would do this for one of two reasons: for actions that make approximately symmetric use of the body, we can reduce the required number of base motions by substituting, for example, *look-right* and its left/right mirror for *look-right* plus *look-left*; for when we wish to include a "handedness" parameter for a motion model without wanting to double the required number of base motions. Since motion models ensure that all of their base motions are aligned such that the actor faces along one of the coordinate axes, we can accomplish both of the above goals while restricting reflections to the coordinate axes.

With this restriction, it is easy to reflect our quaternion motion curves: if we wish to reflect a quaternion along the X axis, we simply negate the Y and Z vector components of the quaternion; to reflect along the Y axis, we would negate the X and Z vector components. To reflect the global translation of a clip along an axis, we negate the single component of the 3D vector associated with the axis. However, we have not yet accounted for the so-called lateral inversion effect of mirrors in which right and left are reversed. If we were actually interested in creating a true reflection of the actor performing a motion, we would need to physically reflect the actor's structure right-to-left, and thus could not implement reflection as a simple filter operation. However, we are only interested in reflecting the motion, not the actor's structure; therefore, to complete the reflection we need only swap the reflected motion curves of laterally symmetric bodyparts (*e.g.* swap left arm with right arm, left leg with right leg, and torso and head with themselves). This brings up an item of knowledge packaged with character models that we omitted from our earlier discussion: each character model class specifies, for each bodypart, its laterally symmetric bodypart.

The **clip-mirror** uses this information to reflect a given clip along one of the coordinate axes (*i.e.* across the plane orthogonal to the axis).

4.3.5 Splicing and Dicing

When we begin to address the problem of assembling a consistent, continuous animation by segueing and layering multiple primitive actions, we quickly discover a need for several specialized motion operators. We have already discussed the clip-mix, which plays an important role in layering. When using a clip-blend to blend a partial motion (one that does not provide motion curves for all bodyparts) with a complete motion, we also need a means of selecting only those motion curves in the full motion that are present in the partial motion. We can accomplish this with a **clip-select**, which makes accessible only a specified set of motion curves in a motion, by enumerating the bodypart-groups that should be present. Thus the two operands to the clip-blend would be the partial motion and a clip-select based on the complete motion that uses the same set of bodypart-groups contained in the partial motion.

The final type of specialized clip is the **clip-composite**, which is also by far the most sophisticated and special-purpose. Its purpose is to serve as the medium into which all motion expressions that form an animation are spliced, forming a continuous whole. Each actor in an animation only creates one clip-composite, which we named in the previous chapter the *rough-cut*. Its detailed operation represents a large part of our work on segueing and layering; as the parts relating to layering, in particular, are somewhat involved, we will delay a deep discussion of its operation until chapter 6, where we will already be in the context of transitions. For now we will give a simplified definition sufficient to carry us through the motion models chapter.

A clip-composite, when created (or reset), is essentially an empty timeline, much as one would find in any video-editing program. Motion models add their motions (in the form of clips) into their actor's clip-composite as *segments.* A segment is simply a continuous sub-section of an existing clip, which we endow with a specific position in the clip-composite timeline. The segment can be added as a "stand-alone" segment if it is positioned earliest in time on the timeline, or it can be added to segue from a segment already in the timeline, or layered on top of one or more segments, blocking out the segments "below" it for its duration. When we add a segment to be segued or layered, we can include a separate "glue" segment that can overlap both the previous/underlying segment and the newly added segment, serving as a *transition* segment between them. This glue segment is, of course, also a clip; in fact, it is a specialized class known as a **clip-transition**, whose form and method we will discuss in chapter 6 when we address transitions. With the contribution of the clip-transition, the clip-composite stitches all of the motions produced by an actor's individual motion models into one animation.

4.3.6 Inverse Kinematics

Our final transformation operation is inverse kinematics (IK), which is not cast as a member of the clip family since it operates on poses rather than motions. It is, however, very useful as a front-end to specifying space-warp keys, and is crucial to our constraint satisfaction engine (as we will see in chapter 7). As we discussed in chapter 2, the literature provides many different formulations and approaches to IK. The work in this thesis grew out of our own investigations into devising IK algorithms suitable for motion synthesis, and several aspects of the results of those investigations are unique. Therefore, we will now describe the IK algorithm used in this thesis, with emphasis on the aspects that are new to the field.

IK solves the problem of generating a pose for an articulated figure or other system of linkages from a set of position and orientation constraints applied at specific places on the figure or linkage. Typically, the numerical problem of determining values for the figure's DOFs that will satisfy the constraints is underdetermined, because there are far fewer constraints than DOFs. A popular means of addressing this problem is to additionally specify an *objective function* of the DOFs that must be minimized simultaneously to the constraints being satisfied. Our IK algorithm is based on this *constrained optimization* approach.

Any IK problem can be broken down into several sub-problems that can be solved more or less independently. We will discuss each of the following sub-problems in sequence, and follow with several practical considerations and caveats that we encountered in developing our algorithm:

- Given a set of constraints, how do we form a problem that can be solved numerically to produce a complete pose for our figure or linkage?
- How can we efficiently and robustly solve the numerical problem posed in the first step?
- How do we form an objective function that will produce natural-looking solutions?

- What form of constraints are we allowed to create?

To the best of our knowledge, the original idea for the solver presented in section 4.3.6.2 is attributed to Andrew Witkin and David Baraff. For reasons that will become clear in the presentation, they refer to it as the "square root of W" method.

4.3.6.1 Problem Setup

We begin by gathering the DOFs of all of our actors into a single state vector $\mathbf{q}$ of dimension n. We wish to control the system by specifying a set of k scalar, non-linear equality constraints $\mathbf{C}(\mathbf{q}) = \mathbf{0}$, which will allow us to (among other things) specify the position and/or orientation of various bodyparts. Solving directly for a $\mathbf{q}$ that satisfies the constraints is unduly difficult because the constraint equations are non-linear. Instead, we draw inspiration from interactive use of IK, in which the user pulls on constraint handles, in effect specifying a *velocity* with which the constraints must move. We begin with the character possessing an initial state $\mathbf{q}_0$ (whatever pose the actor happens to be in). We then iteratively compute a *direction* in which the actor should move in order to satisfy the velocity constraints we have specified, and then use the computed direction to integrate a simple ordinary differential equation (ODE) to produce a new state $\mathbf{q}_{next}$. To control velocities, we begin by differentiating the constraint equations with respect to time:

$$\frac{\partial \mathbf{C}}{\partial t} = \frac{\partial \mathbf{C}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial t} = \mathbf{J}\dot{\mathbf{q}}$$

$$\mathbf{J}\dot{\mathbf{q}} = -\alpha \mathbf{C}(\mathbf{q}) \tag{4.1}$$

In the above, $\dot{\mathbf{q}}$ is the instantaneous velocity of the actor in configuration space, for which we will be solving during each integration of our ODE. The *Jacobian* $\mathbf{J}$ expresses the linear, differential relationship between velocities in configuration space and velocities in the constraint space, and thus $\mathbf{J}\dot{\mathbf{q}}$ is the velocity of the constraints. To satisfy the original position or orientation constraints by controlling their velocities, we simply command them to move in the opposite direction of their current violations! This is expressed by the linear equation (4.1), in which the constant α is a scaling coefficient less than 1.0, so the vector $-\alpha\mathbf{C}(\mathbf{q})$ represents a small velocity in the opposite direction of the current constraint violations.

Equation (4.1) is a linear equation for $\dot{\mathbf{q}}$, our unknowns, which means it is solvable. However, since we generally specify far fewer constraints than we have DOFs, the problem is massively under-specified. This means that there are many $\dot{\mathbf{q}}$'s that will make equation (4.1) true, and we must find a means of choosing between them. Even a brief amount of experimentation quickly informs us that an excellent metric to begin with is one that minimizes the amount of "thrashing about" the actor does on his way to satisfying the constraints. It seems the simplest way of achieving this is by choosing the *smallest* $\dot{\mathbf{q}}$ at each step.

It just so happens that the numerical optimization literature supplies various *least-squares* matrix equation solving algorithms for underdetermined systems (such as we have), which produce a solution vector that minimizes the difference of the solution from the initial guess supplied to the algorithm. That is, if we wish to solve $\mathbf{b} = \mathbf{Mx}$ for $\mathbf{x}$ and supply an initial value for $\mathbf{x}$ of $\mathbf{x}_0$, then the solution will be the $\mathbf{x}$ that minimizes

$$\left\| \mathbf{x} - \mathbf{x}_0 \right\|_2$$

If we always start out with $\mathbf{x}_0 = \mathbf{0}$, then we get the solution with the smallest absolute magnitude. If we were to apply such an algorithm (such as conjugate gradients [Press 1992] or LSQR [Paige 1982]) directly to equation (4.1), we would indeed get the smallest magnitude $\dot{\mathbf{q}}$. However, we would quickly discover that minimizing $\dot{\mathbf{q}}$ directly does not give us very good results, because it ignores the differences in scaling between the various DOFs. If we measured length in microns and angle in radians, our actor would almost never change his global position, preferring instead to rotate. If we measured the bend and twist of his torso in radians and the swing of his shoulder in degrees, he would twist fully about without moving his shoulder to reach a target in front of him. We require a *scale-invariant* metric and the framework in which to minimize it. In section 4.3.6.3 we will describe how to derive a scale-invariant metric known as the *mass matrix*, which measures the total displacement of all the actor's bodyparts. To actually use this metric, we will specify, in addition to our constraints, a quadratic *objective function* of our unknowns $\dot{\mathbf{q}}$ that we will simultaneously minimize while satisfying the velocity constraints:

$$E\left(\dot{\mathbf{q}}\right) = a\dot{\mathbf{q}}\mathbf{M}\dot{\mathbf{q}} + \mathbf{b}\dot{\mathbf{q}} + c \tag{4.2}$$

In (4.2), a and c are scalars, $\mathbf{M}$ is a (generally symmetric) matrix that we will compute as the mass matrix, and $\mathbf{b}$ is a vector that we can utilize to simulate the effect of springs acting on our actor. In the absence of constraints, we would minimize equation (4.2) by setting its gradient to zero:

$$\nabla E\left(\dot{\mathbf{q}}\right) = a\mathbf{M}\dot{\mathbf{q}} + \mathbf{b} = \mathbf{0} \tag{4.3}$$

and solving the resulting linear system of equations for $\dot{\mathbf{q}}$. In the presence of the constraints, we instead require that ∇E point entirely along a direction forbidden by the constraints. Gleicher presents a method for doing this based on *Lagrange multipliers* that involves inverting $\mathbf{M}$ [Gleicher 1992], while Cohen presents a somewhat different formulation of the entire problem, using *sequential quadratic programming* (SQP) that solves a larger linear system but does not require us to invert $\mathbf{M}$ [Cohen 1992]. In the next section we will present an alternative technique that is more numerically stable than Gleicher's and faster than Cohen's. For now we simply assume that we can compute our $\dot{\mathbf{q}}$ that satisfies the velocity constraints and subject to them, minimizes the quadratic objective function *E*.

Once we have calculated $\dot{\mathbf{q}}$, we integrate through the simple ODE

$$\mathbf{q}_{next} = \mathbf{q}_{curr} + \triangle t\,\dot{\mathbf{q}} \quad (4.4)$$

to get a new $\mathbf{q}$, the result of moving the actor differentially in the direction specified by the velocity constraints. If we are interested in satisfying position or orientation constraints, we repeat the process of moving the constraints in the opposite direction of the constraint violations until $\|\mathbf{C}(\mathbf{q})\|_2 < \varepsilon$.

4.3.6.2 The Solver Technique

It will not affect the mathematics if we step back for a moment and think of our problem in somewhat physical terms, in which the constraints are mechanisms that must exert force on bodyparts to keep the constraints satisfied. For instance, a "nail" constraint nails one point on a bodypart to a point in space – the nail constraint must exert a force to keep the bodypart in place; in nature, this force generally comes from friction and compressive forces. If we think about our constraints as exerting a "virtual force" $\mathbf{f}_c$ that keeps them satisfied (and brings them into satisfaction if they are not already), then one way of stating our optimality criteria for ∇E in the presence of constraints is that it be totally *comprised* of the virtual constraint force:

$$\mathbf{f}_c = \mathbf{M}\dot{\mathbf{q}} + \mathbf{b}$$

$$\dot{\mathbf{q}} = \mathbf{M}^{-1}(\mathbf{f}_c - \mathbf{b}) \quad (4.5)$$

This does not seem to help us, at first glance, since we now have *two* unknowns, but we can eliminate $\dot{\mathbf{q}}$ by substituting equation (4.5) into equation (4.1) like so:

$$\mathbf{J}\mathbf{M}^{-1}(\mathbf{f}_c - \mathbf{b}) = -\alpha\mathbf{C}(\mathbf{q}) \quad (4.6)$$

$$\mathbf{J}\mathbf{W}\mathbf{b} - \alpha\mathbf{C}(\mathbf{q}) = \mathbf{J}\mathbf{W}\mathbf{f}_c \quad (4.7)$$

In equation (4.7), $\mathbf{W}$ is the inverse of $\mathbf{M}$, and $\mathbf{f}_c$ is the only unknown, for which we will now be solving. Once we have computed $\mathbf{f}_c$, we can calculate $\dot{\mathbf{q}}$ using equation (4.5), since everything else in the equation is known.

How do we solve equation (4.7) for $\mathbf{f}_c$? Least-squares iterative solvers will allow us to solve for $\mathbf{f}_c$ just as they allowed us to solve equation (4.1) for $\dot{\mathbf{q}}$. However, to achieve the scaling specified by our objective function *E*, we wish for the solver to minimize not the absolute magnitude of $\mathbf{f}_c$ (which is what we would get by default), but the following quantity:

$$\tfrac{1}{2}\mathbf{f}_c\mathbf{W}\mathbf{f}_c \quad (4.8)$$

To achieve this, we make use of Cholesky decomposition [Press 1992], which computes the "square root" $\widehat{\mathbf{M}}$ of a symmetric, positive definite matrix $\mathbf{M}$, such that $\widehat{\mathbf{M}}\widehat{\mathbf{M}}^{\mathrm{T}} = \mathbf{M}$.

Applying Cholesky decomposition to $\mathbf{W}$, we introduce a new variable $\mathbf{y}$:

$$\mathbf{y} = \widehat{\mathbf{W}}^{\mathrm{T}}\mathbf{f}_c, \qquad \text{therefore,}$$

$$\mathbf{f}_c = (\widehat{\mathbf{W}}^{\mathrm{T}})^{-1}\mathbf{y} \tag{4.9}$$

Substituting equation (4.9) into equation (4.7) gives us the following:

$$\mathbf{JWb} - \alpha\mathbf{C}(\mathbf{q}) = \mathbf{J}(\widehat{\mathbf{W}}\widehat{\mathbf{W}}^{\mathrm{T}})(\widehat{\mathbf{W}}^{\mathrm{T}})^{-1}\mathbf{y}$$

multiplicative canceling occurs, leaving us with

$$\mathbf{JWb} - \alpha\mathbf{C}(\mathbf{q}) = \mathbf{J}\widehat{\mathbf{W}}\mathbf{y} \tag{4.10}$$

Now, feeding this matrix equation into a least-squares solver, with $\mathbf{y}_0 = \mathbf{0}$ will produce a solution that minimizes

$$\|\mathbf{y}\|_2 = \mathbf{y}^{\mathrm{T}}\mathbf{y} = (\mathbf{f}_c\widehat{\mathbf{W}})(\widehat{\mathbf{W}}^{\mathrm{T}}\mathbf{f}_c) = \mathbf{f}_c\mathbf{W}\mathbf{f}_c$$

which is exactly what we wanted. Having computed $\mathbf{y}$, we can calculate $\mathbf{f}_c$ using equation (4.9), and, as discussed above, compute $\dot{\mathbf{q}}$ from $\mathbf{f}_c$. The computational speed of this formulation is generally limited by the time required to invert and Cholesky factor the mass matrix **M**, which is firmly O(n^3), whereas the linear system of equations in (4.10) is sparse and can be solved quite rapidly[14]. However, as we will see at the end of the next section, for certain cases in which we are interested, we can reduce this time to O(n).

4.3.6.3 Kinetic Energy as Objective Function and Mass Matrix

The stated goal of our objective function is to produce scale-invariant results that cause the actor to move as little as possible in order to satisfy the constraints. If we think about every particle in the actor's body as having an infinitesimal mass *m*, then we want each individual particle to move as little as possible. This is equivalent to stating that we want our actor to expend as little virtual kinetic energy as possible. Kinetic energy expenditure is a better measure of how much a character moves than $\|\dot{\mathbf{q}}\|$ because it does not depend on the parameterization of the character's DOFs, only on the "physical" structure of the character – *i.e.* its mass distribution. Thus we want to minimize kinetic energy expenditure, which just happens to be a quadratic function of velocity:

---

[14] It is difficult to characterize the complexity of the sparse solver in "big-O" notation. In practical use, we have measured our LSQR solver to require on average twenty to thirty percent of *n* iterations to converge. Each iteration involves several sparse matrix-vector multiplications, each of whose execution time depends on the number of non-zero entries in **J**, which is generally below thirty percent of n^2.

$$\min \quad KE = \tfrac{1}{2} m v^2 \tag{4.11}$$

For this to be useful, we must express it in terms of $\dot{\mathbf{q}}$. Let us, for a moment, consider just one of the particles that make up our actor's body. Recall from section 4.1.1 that the world-space position of a point in a transformation hierarchy can be expressed as the (fixed) position of the point in its local bodypart reference frame multiplied by a sequence of transformation matrices. Since that composite transformation is a function of $\mathbf{q}$, the joint angles, we will call it $\mathbf{R}(\mathbf{q})$, or just $\mathbf{R}$ for short; thus the world space position of our particle is $\mathbf{Rp}$. Now let us take its time derivative:

$$\frac{\partial}{\partial t}\mathbf{Rp} = \frac{\partial \mathbf{R}}{\partial \mathbf{q}}\frac{\partial \mathbf{q}}{\partial t}\mathbf{p} = \frac{\partial \mathbf{R}}{\partial \mathbf{q}}\dot{\mathbf{q}}\mathbf{p}$$

Substituting this expression for the particle's velocity into equation (4.11) gives us the kinetic energy of a single particle:

$$KE = \tfrac{1}{2} m \left(\frac{\partial \mathbf{R}}{\partial \mathbf{q}}\dot{\mathbf{q}}\mathbf{p}\right)^{\mathrm{T}} \left(\frac{\partial \mathbf{R}}{\partial \mathbf{q}}\dot{\mathbf{q}}\mathbf{p}\right) \tag{4.12}$$

To calculate the KE of the actor as a whole, we must sum the energies of all its constituent particles, which, assuming constant density ρ over the actor's body, translates into a volume integral:

$$KE = \tfrac{1}{2} \int_{Volume} \rho \left(\frac{\partial \mathbf{R}}{\partial \mathbf{q}}\dot{\mathbf{q}}\mathbf{p}\right)^{\mathrm{T}} \left(\frac{\partial \mathbf{R}}{\partial \mathbf{q}}\dot{\mathbf{q}}\mathbf{p}\right) \tag{4.13}$$

which is indeed a quadratic function of $\dot{\mathbf{q}}$. The Hessian of this quantity (second derivative with respect to $\dot{\mathbf{q}}$) is an n by n matrix $\mathbf{M}$ known as the mass matrix:

$$\mathbf{M} = \frac{\partial^2 KE}{\partial \dot{\mathbf{q}}^2} = \tfrac{1}{2} \int_{Volume} \rho \left(\frac{\partial \mathbf{R}}{\partial \mathbf{q}}\mathbf{p}\right)^{\mathrm{T}} \left(\frac{\partial \mathbf{R}}{\partial \mathbf{q}}\mathbf{p}\right) \tag{4.14}$$

To actually compute this quantity efficiently, we turn the integral over the entire body into a sum of integrals for the individual bodypart/nodes in the hierarchy, each of which can be computed analytically. We illustrate by summing over bodyparts j and keeping track of the indices of differentiation that we previously omitted for simplicity:

$$\begin{aligned} KE &= \tfrac{1}{2} \sum_j \int_{Volume} \rho \left(\frac{\partial \mathbf{R}_j}{\partial \mathbf{q}_r}\dot{\mathbf{q}}\mathbf{p}\right)^{\mathrm{T}} \left(\frac{\partial \mathbf{R}_j}{\partial \mathbf{q}_s}\dot{\mathbf{q}}\mathbf{p}\right) \\ \frac{\partial^2 KE}{\partial \dot{\mathbf{q}}_r \partial \dot{\mathbf{q}}_s} &= \sum_j \int_{Volume} \rho \left(\frac{\partial \mathbf{R}_j}{\partial \mathbf{q}_r}\mathbf{p}\right)^{\mathrm{T}} \left(\frac{\partial \mathbf{R}_j}{\partial \mathbf{q}_s}\mathbf{p}\right) \\ &= \sum_j \operatorname{trace}\left(\left(\frac{\partial \mathbf{R}_j}{\partial \mathbf{q}_r}\right)^{\mathrm{T}} \boldsymbol{\tau}_j \frac{\partial \mathbf{R}_j}{\partial \mathbf{q}_s}\right) \end{aligned} \tag{4.15}$$

where $\boldsymbol{\tau}_j$ is the *mass matrix tensor* for node *j*, computed as the integral of outer-product of position with itself, times the node mass:

$$\boldsymbol{\tau}_j = m_j \int \mathbf{p}\mathbf{p}^{\mathrm{T}} dx\, dy\, dz \tag{4.16}$$

which we can calculate analytically for simple shapes like ellipsoids and boxes, and compute via a variety of other methods for more complex shapes [Baraff 1995]. Note that this tensor remains constant as long as the approximate shape of the bodypart remains constant – it does not depend on the current orientation of the bodypart at all. Equation (4.15) gives us an exact formula for calculating each element of the mass matrix **M**. Note that **M** is symmetric, and that we can compute all of its elements in a single descent of the transformation hierarchy by accumulating tensors and derivatives up the tree as we recurse out of the descent.

When we use this mass matrix in the equations of the previous two sections, we achieve IK performance that causes the actor to expend minimal kinetic energy while satisfying the velocity constraints. In practice, this simulates a perfectly weighted mannequin suspended in water in zero gravity; if we pull on the actor's hips, the hips follow, but every appendage tries to stay in its place, trailing behind to various degrees. While this is well-behaved and a perfect simulation of an articulated mannequin with no internal motivators of its own, we quickly discovered it was not very practical or useful in trying to control the actor's pose interactively with a few constraints. Fortunately, we discovered that a diagonal approximation to the mass matrix (a matrix that contains the diagonal elements of the mass matrix, and zero everywhere else) results in appealing, scale-independent movement for the actor. Our intuition as to why the diagonal approximation is more useful for interactive posing than the true energy minimization is that the approximation transmits the structure of the transformation hierarchy to the user, whereas the energy minimization hides it. Transmitting the hierarchy's structure is useful because it allows us to manipulate higher-up portions of the hierarchy without perturbing the DOFs of lower-down parts; for instance, if we pull on the shoulder, the arm will follow along (whereas under energy minimization, the hand would try to remain at the same position in space, as would the rest of the arm to varying degrees). As an added bonus, the diagonal approximation requires only linear time to compute, invert, and Cholesky factor, whereas the full mass matrix requires quadratic time to compute, and cubic time to invert and factor.

4.3.6.4 Near Singular Configurations

This technique, like all IK techniques based on numerical optimization, becomes very ill-conditioned[15] at near singular configurations. We experimented with many techniques and tricks to mitigate the resulting "thrashing" behavior that results (the solver produces extremely large joint-angle velocities). Our current solution is to use the LSQR solver [Paige 1982], which incrementally computes a lower bound on the condition number of the system as it iterates its solution; in ill-conditioned cases, the condition number is very high – therefore we can cut off the solver when it is in trouble and either do nothing or relax the system (by damping the system, switching to a penalty method, or altering the constraints to get out of the near singular configuration). Cutting off the solver completely does not completely eliminate the thrashing, but mitigates it and brings it quickly back under control when the constraints cease to conflict. Adding sufficient damping to the system (adding a constant positive term to each diagonal element of the system) eliminates the thrashing, but causes all constraints to become sluggish and sloppy. Turning all constraints into penalties would result in somewhat similar behavior as damping; however, switching only the *problematic* constraint(s) into penalties produces generally acceptable behavior. We have confirmed this by manually switching to penalty constraints – doing so automatically requires determining which constraints are the problematic ones, which we have not yet implemented. Although near-singular configurations seem to arise often when the user is manipulating the character directly with IK, we have yet to encounter a problem with them when using IK from within motion models to pose characters.

4.3.6.5 Enforcing Joint Limits

Joint limits are, by nature, inequality constraints, and our differential solver is only equipped to handle equality constraints. We must, therefore, activate joint limit constraints as equality constraints at the joint limit boundary, but only when the original inequality constraint would be otherwise violated. We do this for each quadratic sub-problem (*i.e.* computation for $\dot{\mathbf{q}}$), so that we never produce a state that is in violation of the joint limits. This proceeds as follows: at the beginning of each sub-problem, we initialize the *active set* of joint limit constraints to the empty set. We then compute a $\dot{\mathbf{q}}$ and perform a trial integration of equation (4.4) and check to see whether any joint limit constraints would be violated. If all joint limit constraints are satisfied, we are done and we accept the new $\dot{\mathbf{q}}$ and $\mathbf{q}$. For each joint limit constraint that *is* violated, we add the constraint to the active set of constraints as an equality constraint that will cause the joint to move exactly to the edge of its allowed motion range. We then resolve the quadratic sub-problem, with the active set of joint limit constraints added to the set of constraints that must be satisfied by the solver; we repeat this process until no further constraints are added to the active set during an iteration. Although in theory this could cause each sub-problem to be solved as many times as there are joints, statistics generated from our typical usage of our prototype systems shows that, on average, each sub-problem is solved between one and two times.

---

[15] For a good introduction to the problems of singular and near-singular configurations, as well as dealing with ill-conditioned systems, see Maciejewski [Maciejewski 1990].

4.3.6.6 *Constraints for IK*

The only element still lacking from our formulation is how we actually form useful constraint equations. This area has received much attention in the literature, so we will only sketch the form of our constraints and refer to Witkin [Witkin 1987, Witkin 1990] and Welman [Welman 1993] for details on implementing them. Since we can constrain anything we can express in equation form (and take derivatives of it), we are limited only by the aspects of an actor that we can express as calculable quantities. We restrict ourselves to geometric quantities such as positions, orientations, and joint values of actors' bodyparts, as well as an actor's center of mass.

With these primitives and additionally regular scalars, vectors, and vector-valued functions of time, we can create constraints that nail a point on the actor's body to a particular point in space or sweep the point along a trajectory in time, require that a bodypart orient like another bodypart or according to an independent quaternion, require that a point remain on one side of a plane, and require that an actor's center of mass remain fixed while he performs other actions.

For instance, to nail a point $\mathbf{p}$ on bodypart j to the location $\mathbf{v}$ in world space, we would create the constraint

$$\mathbf{C}(\mathbf{q}) = \mathbf{R}_j(\mathbf{q})\mathbf{p} - \mathbf{v} = \mathbf{0}$$

which is actually a set of three scalar constraints, stating that each of the x, y, and z components of the bodypart point and point in space must be equal. Computing derivatives on such constraint equations is simplified by the use of *auto-differentiation*, which is described in the references we mentioned in the preceding paragraphs.

4.4 Poses, Pose distance, and its uses

The last basic concept we need to define is that of quantifying and controlling the difference between two poses. When we discuss how to calculate the duration of a transition between two motions in chapter 6, we will utilize the "pose distance" between the starting and ending poses in the computation. Also, our constraint satisfaction engine must be able to compute a solution that satisfies the constraints while deviating as little as possible from the rough motion computed by the motion models, which means that at every instant of the animation we are minimizing the distance between the pose dictated by the motion models and the constrained pose.

Before we can begin to discuss computing pose distance, we need to define how we measure it. Since we have already defined a pose as the collection of joint angles that define the configuration of a character model, we might define the pose distance as the Euclidean norm of the difference between the two pose vectors, *i.e.* given poses $\mathbf{q}_1$ and $\mathbf{q}_2$, the distance d between them is:

$$d = \left\| \mathbf{q}_1 - \mathbf{q}_2 \right\|_2$$

which has the attraction of cheap computation. Unfortunately, this metric completely ignores the scaling effect of the joints on the pose. For instance, it would count a 45 degree rotation of the wrist as having an equal impact on the pose distance as a 45 degree bend at the waist. Since the mass matrix provides scale-invariance from just this sort of discrepancy, we could consider altering the computation to:

$$d = \left(\left(\mathbf{q}_1 - \mathbf{q}_2 \right)^{\mathrm{T}} \mathbf{M} \left(\mathbf{q}_1 - \mathbf{q}_2 \right) \right)^{\frac{1}{2}}$$

This provides a much better measure, but is still problematic because the mass matrix must be computed at just one of the two poses, and truly measures energy expenditure only differentially in the neighborhood of the pose. Thus, the larger the true distance between the poses, the greater distortion there will be due to the local nature of the mass matrix.

Popović derives an elegant alternative that measures the sum of the squares of mass displacements in moving from one pose to another [Popović 1999a]. Following Popović's derivation, we define the mass displacement of a single bodypart/node in the transformation hierarchy as:

$$E_{dm} = \int_i \rho \left(\mathbf{x}_i - \overline{\mathbf{x}}_i \right)^2 dx\, dy\, dz$$

where we are integrating over all points i in the node, ρ is the density for the node, and $\mathbf{x}$ is the position of a body point $\mathbf{p}$ in the first pose, while $\overline{\mathbf{x}}$ is its position in the second pose. We compute $\mathbf{x}$ just as we would the world space position of any body point in node j:

$$\mathbf{x}_i = \mathbf{R}_j \mathbf{p}_i$$

where $\mathbf{R}_j$ is the transformation matrix that brings the point into world space. Note that for all the points $\mathbf{p}$ within node j, the same transformation $\mathbf{R}_j$ transforms into world space. This fact allows us to make the following simplifications, leading to an efficient method of computation:

$$\begin{aligned}
E_{dm_j} &= \int_i \rho_j \left(\mathbf{R}_j \mathbf{p}_i - \overline{\mathbf{R}}_j \mathbf{p}_i \right)^2 dx\, dy\, dz \\
E_{dm_j} &= \int_i \rho_j \left(\mathbf{R}_j \mathbf{p}_i \mathbf{R}_j \mathbf{p}_i - 2 \mathbf{R}_j \mathbf{p}_i \overline{\mathbf{R}}_j \mathbf{p}_i + \overline{\mathbf{R}}_j \mathbf{p}_i \overline{\mathbf{R}}_j \mathbf{p}_i \right) dx\, dy\, dz \\
E_{dm_j} &= \mathrm{trace}\left(\mathbf{R}_j \boldsymbol{\tau}_j \mathbf{R}_j^{\mathrm{T}} - 2 \mathbf{R}_j \boldsymbol{\tau}_j \overline{\mathbf{R}}_j^{\mathrm{T}} + \overline{\mathbf{R}}_j \boldsymbol{\tau}_j \overline{\mathbf{R}}_j^{\mathrm{T}} \right) \\
E_{dm_j} &= \mathrm{trace}\left(\mathbf{R}_j \boldsymbol{\tau}_j \left(\mathbf{R}_j^{\mathrm{T}} - 2 \overline{\mathbf{R}}_j^{\mathrm{T}} \right) + \overline{\mathbf{R}}_j \boldsymbol{\tau}_j \overline{\mathbf{R}}_j^{\mathrm{T}} \right)
\end{aligned}$$

where $\boldsymbol{\tau}_j$ is the exact same mass matrix tensor defined in equation (4.16). In fact, the entire formulation of E_{dm} is nearly identical to that of kinetic energy expenditure, except that instead of measuring *differential* motion with $\dot{\mathbf{q}}$, we are now measuring finite displacements $\Delta\mathbf{q}$. To compute the total mass-square displacement of the transformation hierarchy, we simply sum the contributions from each bodypart/node.

However, for a formula we develop in chapter 6 to compute the duration of a transition, we will be interested not in the sum of mass-squared displacements, but in the sum of actual displacements:

$$d = \int_i \rho \sqrt{\left(\mathbf{x}_i - \overline{\mathbf{x}}_i\right)^2}\, dx\, dy\, dz$$

Unfortunately, the efficient formulation of the mass-square computation depends on summing the displacements-squared: we cannot allow a square root inside the integral. Thus we cannot compute the true mass displacement efficiently enough for it to be useful. We therefore compute the root mean-square displacement, which has the same units as mass displacement, and which we have verified for the ellipsoid and box using *Maple*[16] that it is asymptotically equivalent to (and typically within a few percent of) the true mass displacement:

$$d_{approx} = \hat{\mathrm{E}}_{dm} = m\sqrt{\int_i \left(\mathbf{x}_i - \overline{\mathbf{x}}_i\right)^2 dx\, dy\, dz} \tag{4.17}$$

We use this function to compute the pose distance between poses for calculating transition timings.

4.4.1 Minimal Displacement from a Pose in IK

Popović uses his mass-square displacement function directly in his objective function to produce IK solutions that deviate minimally from a given rest pose. He is able to do this because he uses the Lagrangian formulation of SQP [Cohen 1992], which allows functions of $\mathbf{q}$ for its objective. Since we do not use SQP, but rather opt for the speedier, differential IK formulation that uses objective functions of $\dot{\mathbf{q}}$, we cannot use it[17] as is. However, we can make a further approximation, defining

$$\mathbf{q} = \mathbf{q}_{last} + k\dot{\mathbf{q}}$$

that is, defining $\mathbf{q}$ as a differential function of $\dot{\mathbf{q}}$, and assuming simple Euler integration, taking k as the integration step size. If we do so, we can use the chain rule to compute the first and second derivatives with respect to $\dot{\mathbf{q}}$ that we require of an objective function:

---

[16] *Maple* is an automatic mathematical calculation program similar to *Mathematica*.
[17] When we are trying to minimize the difference between two poses, it does not matter whether we minimize the mass-squared displacement or the mass displacement, so we use the simpler mass-squared displacement.

$$E_{dm_j} = \mathrm{trace}\left(\mathbf{R}_j \boldsymbol{\tau}_j \left(\mathbf{R}_j - 2\overline{\mathbf{R}}_j\right)^{\mathrm{T}} + \overline{\mathbf{R}}_j \boldsymbol{\tau}_j \overline{\mathbf{R}}_j^{\mathrm{T}}\right)$$

$$\frac{\partial E_{dm_j}}{\partial \dot{\mathbf{q}}_s} = \mathrm{trace}\left(k \frac{\partial \mathbf{R}_j}{\partial \mathbf{q}_s} \boldsymbol{\tau}_j \left(\mathbf{R}_j - 2\overline{\mathbf{R}}_j\right)^{\mathrm{T}} + \mathbf{R}_j \boldsymbol{\tau}_j k \frac{\partial \mathbf{R}_j^{\mathrm{T}}}{\partial \mathbf{q}_s}\right)$$

$$= 2k\ \mathrm{trace}\left(\left(\mathbf{R}_j - \overline{\mathbf{R}}_j\right) \boldsymbol{\tau}_j \frac{\partial \mathbf{R}_j^{\mathrm{T}}}{\partial \mathbf{q}_s}\right)$$

$$\frac{\partial^2 E_{dm_j}}{\partial \dot{\mathbf{q}}_s \partial \dot{\mathbf{q}}_r} = 2k^2\ \mathrm{trace}\left(\frac{\partial \mathbf{R}_j}{\partial \mathbf{q}_r} \boldsymbol{\tau}_j \frac{\partial \mathbf{R}_j^{\mathrm{T}}}{\partial \mathbf{q}_s} + \left(\mathbf{R}_j - \overline{\mathbf{R}}_j\right) \boldsymbol{\tau}_j \frac{\partial^2 \mathbf{R}_j^{\mathrm{T}}}{\partial \mathbf{q}_s \partial \mathbf{q}_r}\right)$$

However, computation of this Hessian significantly complicates our streamlined formulation, and requires second derivatives with respect to $\mathbf{q}$, which we utilize for no other purpose. We therefore experimented with several other, cheaper objectives, and settled on the following objective derived from our first approach to computing the pose distance:

$$E(\mathbf{q}) = \frac{n^2}{k^2}\left(\mathbf{q} - \overline{\mathbf{q}}\right)^{\mathrm{T}} \left(\mathbf{M}_d + \mathbf{N}\right)\left(\mathbf{q} - \overline{\mathbf{q}}\right)$$

which makes use of the same differential definition of $\mathbf{q}$ in terms of $\dot{\mathbf{q}}$ we utilized above. $\overline{\mathbf{q}}$ is the pose from which we wish to minimally deviate, $\mathbf{M}_d$ is the diagonal approximation to the mass matrix, $\mathbf{N}$ is a diagonal matrix whose diagonal elements are all n, the coefficient of restitution, and k is, as before, the integration step size.

5 Motion Models

In Chapter 3 we described the basics of motion models, how they operated, and how they fit into the context of producing animation. Now, given the knowledge gained from Chapter 4 about the low-level motion transformation algorithms and motion language at our disposal, we are ready to examine motion models in depth.

Motion models are motion generators designed from the start for interacting with other generators to produce rich combinations and sequences of motions. In this chapter, however, we will focus on the structure and design of individual motion generators, each tasked with reproducing a specific type of motion. We will address the concepts and rules pertaining to the *combination* of individual motions in the following chapters. We will, however, where appropriate, note where content from the subsequent chapters is relevant to a structure or operation being described.

Since, in Chapter 3, we have already given an overview of how motion models work, we will begin an in-depth discussion immediately. In section 5.1 we will describe motion models from a structural standpoint, giving detailed definitions of each of the major components (other than combination rules) that comprise motion models. In section 5.2 we will progress to an operational viewpoint, and describe how the various components combine to generate motion via transformation. Then, in section 5.3, knowing what a *well-designed* motion model looks and works like, we address the issues of *how* to design a good motion model, discussing rule design and an application programming interface (API) to standardize and simplify motion model construction. Throughout the chapter we will illustrate all major concepts by developing and referring to a *throw* motion model, which is simple enough in structure to be easily understood, and yet possesses interesting transformation rules and invariants. Finally, in section 5.4 we describe the detailed information about motion models and the clip language that appears in the appendix.

5.1 Components

We begin by enumerating and explaining the major structural components of motion models, including our accumulated experience on selecting and designing each component. These components are, in the order in which we will discuss them: base motions, parameterizations, styles, invariants, and motion transformers. The necessity for, and specific composition of each of these structures is determined by the nature of the motion we wish to generate – specifically: the goal(s) and goal-related variation of the motion, and the important kinematic properties of the motion. Let us, then, define our example motion model in these terms.

Our *throw* motion model should be able to generate any type of humanoid one-handed throw, right or left handed[18], in at least the throwing styles *overhand, underhand, side-arm,* and *frisbee-toss*. The main goal of a *throw* is to cause some thrown object (hereafter the "ball", although it need not be an actual ball) to hit a goal-target; the primary aspect of this goal is the position of the target relative to the actor (hereafter the "thrower") doing the throwing, but to completely control the throw, we may also specify: whether the ball hits the target on its rising or falling arc; the acceleration due to gravity; either the horizontal speed possessed by the ball at the time of impact *or* the height that the ball reaches (would reach) at the apex of its trajectory.

Observation of example motions in each of the styles mentioned reveals (not surprisingly) that the motion of the throwing arm determines the pacing and general form of the throw[19]. There may or may not be one or more footsteps taken during the throw, the head/eyes generally track the target through all phases of the motion (save possibly the follow-through), and the non-throwing arm may be thrust out in the general direction of the target during the windup.

---

[18] Save, perhaps, for the most complicated, multi-stage pitches of the most flamboyant of pitchers (*e.g.* the 1980's Dodgers pitcher Fernando Valenzuela).

[19] We do not intend to imply causality here, rather simply observed correlation.

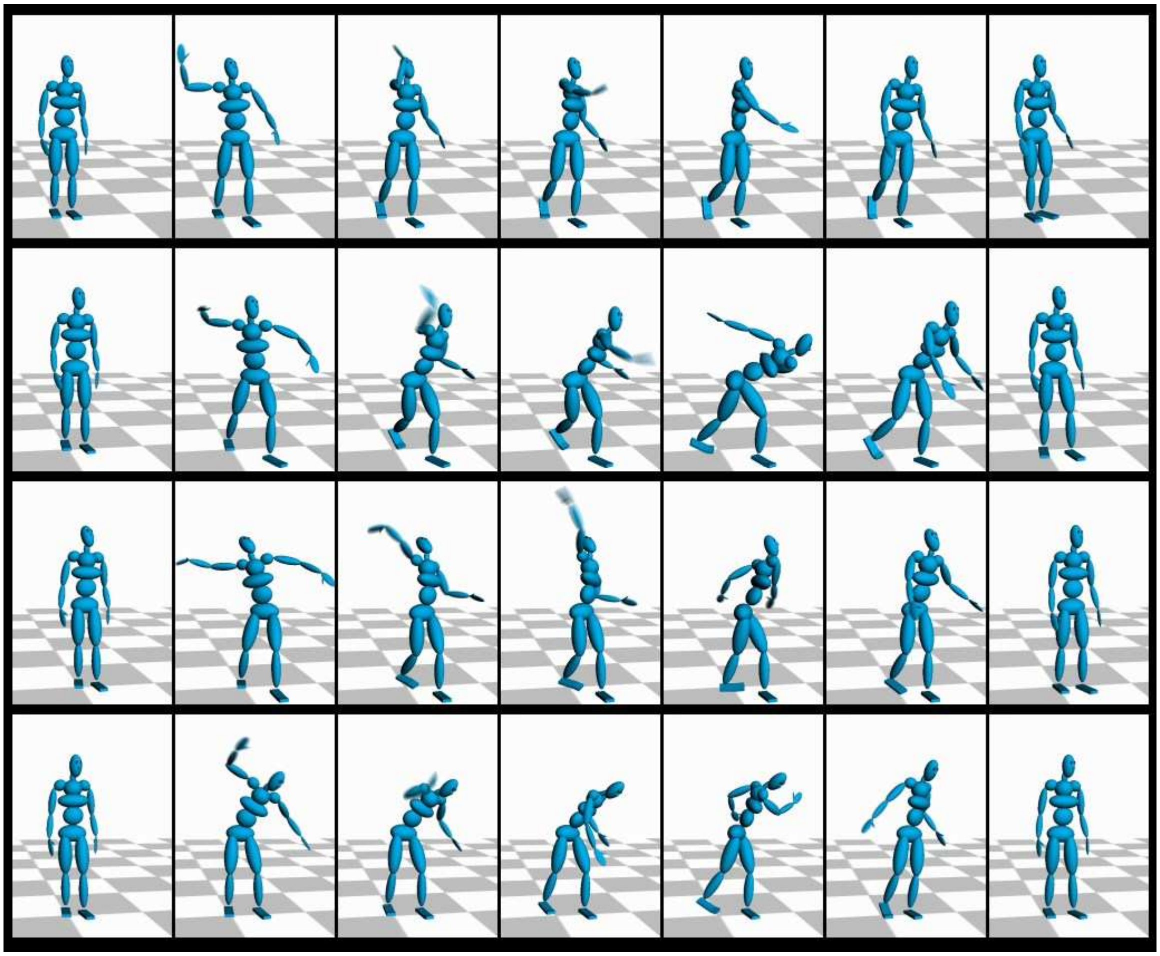


Figure 5-1: The four base motions of *throw*. Here we depict the four base motions defined by the *throw* motion model, in the style *overhand*. The top row contains (non-uniformly spaced in time) frames from the **near** base motion, with motion blur applied to indicate rapid movement. In the subsequent rows, we have the **far**, **up**, and **down** base motions. The actual animations reside at http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter5.

5.1.1 Base Motions

The most fundamental and important component of any motion model is the set of base motions from which all motion is generated. The concept of defining a set of base (or *basis* [Rose 1996]) motions that span a subspace of the task or goal space is not new to this thesis. Any motion synthesis algorithm that relies on interpolation or blending of motions (such as the work of Wiley [Wiley 1997] and Rose [Rose 1998]) must do so. Because our motion synthesis algorithm utilizes additional operators other than interpolation, we gain additional flexibility in choosing a spanning basis of motions, and the ability to annotate each base motion with information about the motion that will guide the motion transformation process.

A base motion is first and foremost a collection of motion curves, one per joint or DOF, all defined over the same time interval. Applied to an appropriate character model as a unit, these motion curves will cause the character model to perform a specific single action. Secondly, a base motion describes the action it represents, both *implicitly*, through its name/position, and *explicitly*, through annotations that accompany the motion curves.

5.1.1.1 Implicit Description

The implicit description of the action comes from the process of determining what needs to be captured and represented by the complete set of base motions for a motion model. In our *throw* motion model, we span the task variation space with four base motions (our rationale will be described below, generally at the end of this section, and more specifically in section 5.3.2), which are depicted in Figure 5-1:

1. **Near** – throwing a ball at a target in front of the thrower, not too far away
2. **Far** – throwing a ball at a distant target, thus more energetically
3. **Up –** throwing a ball very high overhead
4. **Down –** throwing a ball straight down

Thus we know, for each of the four base motions, approximately what kind of action each contains, just by noting the name it was given during the selection process.

5.1.1.2 Explicit Description

The explicit annotations describe the commonalities among the base motions, as well as aspects of motion unique to each. The *form* of each kind of annotation that can be present in a base motion is determined by the motion model with which it is associated. The actual information contained in each annotation is derived from the motion curves themselves (manually, or potentially automatically). The primary annotation, universal to all base motions, is the temporal breakdown of the motion into distinct sub-phases. The motion model itself specifies the number and meaning of the sub-phases, and the per base motion annotations state where each phase falls in its motion. For instance, the *throw* motion model consists of five sub-phases, demarked by six key-times: **start** – just as thrower is about to begin the motion; **windup** – the time at which the throwing arm is maximally cocked and forward throwing motion is about to commence; **release** – the ball is released; **follow-through –** the last instant of the follow-through motion; **settle** – thrower has approximately settled into to a rest or terminating pose; **end** – motion ceases. The frames in Figure 5-1 correspond roughly to these key-times, with an additional (non-key) frame added between **windup** and **release** to illuminate the rapid motion during that phase.

These timing annotations are crucial for several reasons:

- They enable us to align multiple base motions (using time warping) so that they can be properly blended. If we tried to interpolate non-aligned motions, we could inadvertently blend part of a windup with part of a throw-forward, resulting in strange motion or no motion at all.

- They allow the motion model to make deformations to the motion at precisely the correct time. For instance, to ensure that the throw is in the right direction, we need to apply space warp keys to properly align the arm at *windup* and *release*.
- As we will discuss in the next chapter, they allow motion models to reason about when to best begin transitioning into/from following/preceding motions, respectively.

We have found that choosing an effective set of phases for a new motion model is fairly intuitive if the motion can at all be thought of as achieving some goal, which seems to be true of even many gestural motions. In fact, although the exact names we give the key-times may vary somewhat from motion model to motion model, many motions, both gestural and non-, can be effectively described as four phases, with key-times whose meanings are the following: **start, achieve-goal, release-goal, settle, end**.

Several other simple annotations further characterize important gross aspects of the motion. Part of the knowledge available to all legged character model classes is the concept of a footstep, including the sub-phases (liftoff, heel-strike, toe-strike, *etc.*) and invariants involved in defining and enforcing each of these phases. Since the underlying functionality of the API for all legged-creature motion models knows what to do when it encounters footsteps, we can deal with their presence or lack thereof in any base motion by simply annotating when each occurs (the geometry can be inferred automatically). Another, similar annotation, if applicable, is whether the performance is right or left handed.

We can also use annotations to extend, to a certain degree, the nature of a motion for a particular style of performance by adding or deleting invariants. For instance, Joe's particular style of throwing sidearm involves bringing his non-throwing hand to his waist and keeping it there throughout the motion. We can accommodate this style within our "generic" *throw* motion model by adding an annotation to each of the base motions that defines an invariant that will keep the hand on the waist.

5.1.1.3 Canonical and Animation Time

A direct result of annotating base motions with key-times is that we can make a distinction between a motion model's *canonical time* and the current *animation time*. Animation time is "absolute" time, measured in the current animation under construction. Many types of units are possible – we chose integer frame numbers, with the stipulation that 30 frames = 1 second. Canonical time, is measured relative to the key-times of a motion model as a real number in the range [0 .. *number-of-key-times* – 1]. A non-integer canonical time t refers to the time that is $(100 * (t - \lfloor t \rfloor))$ percent of the way between the key-times numbered $\lfloor t \rfloor$ and $\lceil t \rceil$. For instance, the canonical time at which the **release** of a throw occurs is 2.0, since **release** is the third key-time of a *throw*. We use canonical time whenever we wish to temporally position something (whether it be a tweak, invariant, or transition) relative to the motion enacted by the motion model. Motion models store most times as canonical times, since this allows us to make dramatic changes to the makeup and ordering of actions in an animation without perturbing events within an action (*i.e.* motion model).

5.1.1.4 Choosing Base Motions

Picking a good set of spanning base motions for a motion model requires foreknowledge of the possible variation in the goal or task space, as well as other factors such as the importance of bounding the number of base motions. As such, it will probably not be possible to automate this process, although the computer should be able to pick an optimal set of *m* base motions from a larger set of *n* base motions. Assuming we have the resources, we can always simplify this decision by using the proven brute-force approach of dense, regular sampling in the goal-space used by Wiley [Wiley 1997]. However, since we have additional motion tools at our disposal (*viz.* inverse kinematics, space warping, and reflection), we can considerably pare down the number of required base motions to those that cannot be believably achieved by warping other base motions.

This may sound somewhat nebulous, but we can make it more concrete by considering the nature of most goal-directed actions. Many actions involve an actor and a target, the thing being acted upon, which may or may not come into actual contact with the actor. Regardless, the target has a relative position to the actor, and this is generally the main dimension of task-space variation of the action. If the action is symmetric, *i.e.* both halves of the actor's body are doing approximately the same thing, then we can discriminate the target position into five general categories, which gives us five base motions: **in-front, in-back, up, down,** and **to-the-side**. If it makes a difference whether the target is close at hand or farther away, we can prefix each of the preceding with *near* and *far*, doubling the number of base motions to ten. If the action is asymmetric (*e.g.* performed using one hand only), we replace the single **to-the-side** base motion with **adjacent-side** and **opposite-side**, for a total of six (or twelve) base motions.

We do not claim that this set is universal; many motion models we designed have considerably fewer base motions, although none had more. Sometimes other properties of the target require further discrimination of base motions; for instance, the mass of the target matters for a *pick up* motion model. However, consideration of the directionality/relative position of the target to the actor worked well as a starting point for the motion models created for this thesis. If we consider a six base motion set to be minimal[20], we have a rough framework for where to place additional base motions to reduce our reliance on the motion transformations. When considering whether to sample densely or sparingly, consider the pros and cons of dense sampling:

- Using a dense sampling of the task space makes it easier to implement a motion model that will give good results, assuming a good multi-dimensional interpolation scheme. The fewer base motions we

[20] For many humanoid actions, there are actually only five necessary base motions, since they do not consider targets behind the actor.

use, the more sophisticated the motion model must be in determining how to combine and deform the base motions to cover the task space.

- Regardless of how well compressed or parameterized the motion curves of base motions are, denser sampling obviously will lead to more memory intensive motion synthesis, and in the foreseeable future, memory bandwidth is a limiting factor on performance.
- Dense sampling makes acquisition and characterization of new styles more difficult.

In this work, a combination of limited resources and an eagerness to see how far we could push the underlying motion transformation algorithms led us towards choosing smaller base motion sets in general. When starting from the minimal end of the spectrum, we may discover during the course of building and testing the motion model that we actually need more base motions to achieve acceptable results. This happened with our *jump-both-feet* motion model, where we discovered we needed to add a **jump-up** motion to our initial **hop, high,** and **broad** jumps. Therefore it is extremely useful to have the ability to acquire additional motions on demand during the motion model definition process. In hindsight, during our motion capture session, we would have recorded an additional two base motions for the *throw*: **far-up,** and **far-down**.

5.1.2 Parameterizations

Once we have addressed the primary issue of spanning the task space with base motions, we can move on to consider how we would like to enable the user or higher level control systems to control the motion. We do this even before devising algorithms for selecting, combining, and deforming base motions because the inputs to those algorithms are determined by our choice of control parameters. We begin by describing the possible types for control parameters:

- **Position** – a 3D vector describing a position in space, such as the takeoff spot for a *jump.*
- **Orientation –** a rotation (which we store as a quaternion, but which could be any set of rotation parameters) that describes an orientation, or, with the addition of constraints, a direction. An example is the direction one is facing when beginning to *walk.*
- **Boolean –** a true or false value, such as whether the actor keeps his gaze on the target while *reach*ing for it.
- **Enumeration –** one of a set of discrete options, such as whether a *wave* is right or left-handed, or selection of the style in which the motion is to be performed.

- **Scalar –** a single floating point value, such as the height a ball should achieve at its apex during a *throw*.
- **Handle –** an IK handle that identifies both a specific object or bodypart of an actor, and a position on the object (as of a contact point). One example is the point of contact by which we *reach* for and acquire an object.
- **Trajectory –** a path in 3D space, such as the path along which one *walk*s.

Although some of these types are compound types built up from other types, we give each the status of a primitive type. We do this for two reasons: first, each of these types corresponds directly to a heavily used parameter type in our motion synthesis engine, and thus little to no translation is required between user control parameters and engine parameters; second (and most importantly), each primitive type has an associated interaction method specifically tailored to it, making use of direct manipulation wherever possible. We specify positions by 3D direct manipulation of bodyparts or proxy widgets; we alter orientations using arcball controllers [Shoemake 1994]; we create handles via object picking in the animation workspace; we manipulate the control points of trajectories in the workspace just like individual positions; because they do not generally have a direct spatial correspondence, we specify booleans, scalars, and enumerations using standalone 2D widgets – radio buttons for booleans, sliders for scalars, and choice/menu widgets for enumerations. Ultimately, the interaction technique we provide for each parameter is nearly as important to the success of the motion model as the semantics of the parameter. However, since polished usability is not the main focus of this work, we settled on these techniques as representative of those in common usage in current 3D applications [Hash 1999][Maya 1999].

When creating a user interface to a motion controller (the parameterization combined with the interaction techniques mentioned above *are* the interface to the motion model), we face two conflicting demands: we must provide numerous and/or sophisticated enough controls to affect all aspects of the motion, should the user desire; and we must make the interface simple, intuitive, and easy to use. Emphasizing the former typically leads to a system for “power users,” such as (in other domains) Emacs, Adobe Photoshop, and Unix, which afford immense and efficient manipulative power in their respective domains, but whose interfaces are arcane and difficult to learn. Emphasizing the latter leads to a system for novices, making it very easy to perform simple manipulations, but affording no mechanisms for sophisticated operations, some examples of which are Microsoft Notepad and Paint. It is unusual to see a system that can function successfully in both arenas.

Our use of motion transformation, coupled with the practice of hierarchical parameterizations, allows us to span much of the spectrum between power user and novice in the interfaces we create for motion models. There are four principal reasons why this is so:

1. Our concept of a “style” bundles what would otherwise be a plethora of stylistic parameters into a single parameter. Such a parameter reduction, while simplifying an interface, would generally result in

a loss of expressive degrees of freedom. However, since we can always add a new style to achieve a desired nuance, we do not sacrifice flexibility for this simplicity.

2. Since the motion produced by motion models is derived from example motions, we are able to compute default values for all parameters left unspecified by the user that are consistent with the currently selected style. Thus the user typically need only modify the parameters he truly wishes to change.

3. Although the concept is certainly not new, we are able to use parameter hierarchies to create and hide (until desired) successively more sophisticated layers of controls. As per point 2 above, the deeper levels of controls receive default values computed from the parameters that are in use and the example motions. We can then present a simple, basic set of controls to the novice user, without sacrificing, for example, fine control over how one motion transitions into another, or control over the placement of each footfall in a walk.

4. Even if the parameter set we design for a particular motion model falls short of providing complete control over every aspect of the motion (which it almost invariably will), the motion produced by motion models can be further refined using techniques similar to keyframing. The techniques, along with their interface, are discussed in Chapter 7.

In our experimental motion models, we have implemented a two-layer control scheme (plus the keyframe-based tweaking mechanism just mentioned). The first layer presents all the basic controls, some common to all motion models, others specific to each. The common controls are name, style, gross position and orientation of the actor at the beginning of the motion, and the motion model from which the motion transitions or is layered upon. The specific parameters cover all dimensions of task-level variation envisioned by the designer. Although we cannot give any hard and fast rules, we have developed a few guidelines that seem to work well, which we will present in the next paragraph. The second layer, accessible by expanding the first, consists also of common parameters that enable minute control over the transition into the motion from the preceding motion. A third layer we would certainly want in a non-prototype system is a set of controls for selectively disabling any or all of the motion's invariants.

Parameter Name	Parameter Type	Parameter Description
Position	*position*	Position of hips in default starting pose
Orientation	*orientation*	Direction actor faces in default starting pose
Target-Pos	*position*	World-space position of target
Target-Obj	*handle*	Actor/object to strike (mutually exclusive of Target-Pos)
Apex-Height	*scalar*	Absolute height ball achieves (would achieve) at apex
Strike-Rising	*boolean*	Does ball strike target on while rising in its arc?
Style	*enumeration*	One of: *overhand, sidearm, underhand, frisbee-toss*

Figure 5-2: First-layer parameters for *throw* motion model. These are the actual parameters we implemented for the *throw* motion model. We chose not to provide a "horizontal striking speed" parameter; had we done so, it would have been mutually exclusive of Apex-Height just as Target-Pos and Target-Obj are mutually exclusive – *i.e.* when one is explicitly set by the user, the other is ignored.

To develop the specific first-layer parameters, we begin by classifying the motion as either a "gesture" or an "action." A gesture (such as *wave, sigh, catch-breath*, *check-watch*, *scratch-head*) is generally dominated by style, and requires few task-level parameters, some of which may be: duration, handedness, target (applicable, for instance, to *wave*). An action generally has a wider space of task variation, which we parameterize by considering the physics and geometry of the action. Our *throw* example falls into the action category (see Figure 5-2). As we mentioned at the beginning of this chapter, the motion is primarily driven by the relative location of the target to the thrower; in addition to the position and orientation of the thrower, we allow the user to specify the target as either a position in space, or a handle on another object in the scene. However, we may also care about the trajectory followed by the ball, which we can completely specify as follows: whether the ball hits the target on its rising or falling arc (boolean); the acceleration due to gravity (scalar); either the horizontal speed possessed by the ball at the time of impact (scalar) *or* the height that the ball reaches (would reach) at the apex of its trajectory (scalar). In a system where we modeled the hands of an actor and performed sophisticated physics simulations, we might also provide a "spin" parameter. We applied the same reasoning to parameterize *peer, jump-both-feet, reach,* and other motion models, which we will describe later.

5.1.3 Styles

The style is a consistently performed set of base motions plus annotations that is interchangeable with other styles associated with the same motion model. It is one of the primary contributions of this work to the field of motion transformation, since all previous work has focussed on transforming specific motions or sets of motions.

We have already discussed most of the relevant aspects of styles in section 5.1.1, since each is primarily a collection of base motions. The macro-structure of a style is largely dictated by the motion model for which the style is defined: which and how many base motions comprise it; additionally, the style can specify additional invariants that should be enforced, and supply for certain parameters default values specific to the style that cannot be derived from the example motions themselves. The micro-structure of every style is identical: for each base motion we describe the motion curves themselves, the values of the key-times, the frame rate of the data in the motion curves, and other annotations required by the motion model (handedness, footsteps, target location, *etc.*).

While this separation of control apparatus (*i.e.* motion model) from motion data (*i.e.* style) gives us an unprecedented degree of expressive freedom in a motion controller, it is not infinite. A style can only deviate in content and structure so far from what the designer of the motion model originally envisioned as typical of the type of action represented by the motion model. For instance, while our *throw* motion model can accommodate overhand, underhand, sidearm, and frisbee-toss styles of throwing, it will probably not produce the same quality results for a discus-toss style, because a discus throw has many more phases and more complicated motion than any of the “common” throws.

We would like to point out that because styles are separate entities from motion models and require no programming to create, they allow for a new niche in animation content providing, akin to professional model-making in the 3D modeling business. For while we would require a fair amount of progress beyond the research in this thesis to enable non-programmers to define a motion model, we firmly believe any competent animator can, with our existing technology (plus some slightly more polished tools) create any number of styles for existing motion models. Current work contracted to 3D character animators is generally for a very specific action and situation, and not intended or suited for reuse. But the growing areas of immersive virtual environments, realistic games, and simulations will require vast amounts of engaging, yet spontaneously reactive motions for their avatars and characters. Styles can help to fill this vacuum by enabling power-user animators to create not just “static animations”, but content that can be used in dynamic environments.

5.1.4 Motion Combiners and Filters

To create motions that satisfy the goals specified by the motion model parameters, we deform and combine individual base motions from a particular style. The particular “motion expression” representing the end-result depends on both the type of motion model and the current parameter settings. The language in which the motion expression is built, however, is common to all motion models – it is precisely the set of clip classes detailed in Chapter 4, which form a functional language for building motions.

The set of combination, deformation, and filtering operators in Chapter 4 is flexible and powerful enough to create a wide variety of motion expressions, and due to its object-oriented/hierarchical design, it is also easily extensible. Of all the motion models created for this thesis, only one required us to create a new type of clip to successfully express its synthesized motion. The *jump-both-feet* motion model defines a **clip-trajectory** operator that, given the takeoff and landing poses and apex height of a jumper, computes and enforces the resulting parabolic trajectory of the jumper's center of mass. It would have been possible to enforce the trajectory solely as an invariant on the center of mass; however, invariants (as we will describe shortly) are not enforced until later in the motion synthesis pipeline, and we need the actual trajectory *while constructing* the motion expression so that we can adjust the airborne timing of the jump to match the physics of ballistic trajectories. Note that we do not *need* to enforce the trajectory in the clip-trajectory (because we require only the trajectory itself to adjust the timing), so the short-circuit of our motion synthesis pipeline is for convenience only.

Continuing with our ongoing example, the *throw* motion model makes typical use of clips in creating its basic motion expression. We depict the final motion expression, comprised of ten layers of clips, in Figure 5-3. In a preprocessing step common to all motion models, we use clip-pivots to align all of the base motions so that, in their starting poses, they are geometrically aligned[21] and can be meaningfully blended. For a given set of parameters, we choose a subset of the aligned base motions to be combined, temporally align them using clip-time-warps, then combine the results using one or more clip-mixes (we could also use clip-blends or a more general multi-dimensional interpolation scheme). Two more clip-pivots and a clip-blend allow us to position and orient the motion according to its starting conditions and make it step into the proper throwing direction (the last is skipped if the style contains no footsteps). Subject to continuously enforcing invariants, we achieve the final, stand-alone throwing motion by applying a clip-space-warp that uses IK-computed poses to ensure the throwing arm points in exactly the right direction at the release time, and a clip-time-warp to dilate the interval from **windup** to **follow-through** so that the hand's speed at **release** matches the initial speed demanded by the desired ballistic trajectory.

5.1.5 Invariants

With the structural components described so far, we can mix, blend, and broadly deform base motions, but we have no mechanism for making an actor achieve precise instantaneous or sustained geometric goals. We must, for instance, be able to ensure that feet do not slip during foot plants, and that an actor's eyes/head continuously track a moving object while gazing at it. Invariants are our general-purpose mechanism for specifying and enforcing geometric constraints, both in the single persistent constraint satisfaction process applied to the entire animation (section 5.2.5), and as the means for generating poses to define space-warp deformations within motion models. They also serve as a primary means of communication between motion models.

[21] We will define what "aligned" means in section 5.2.2

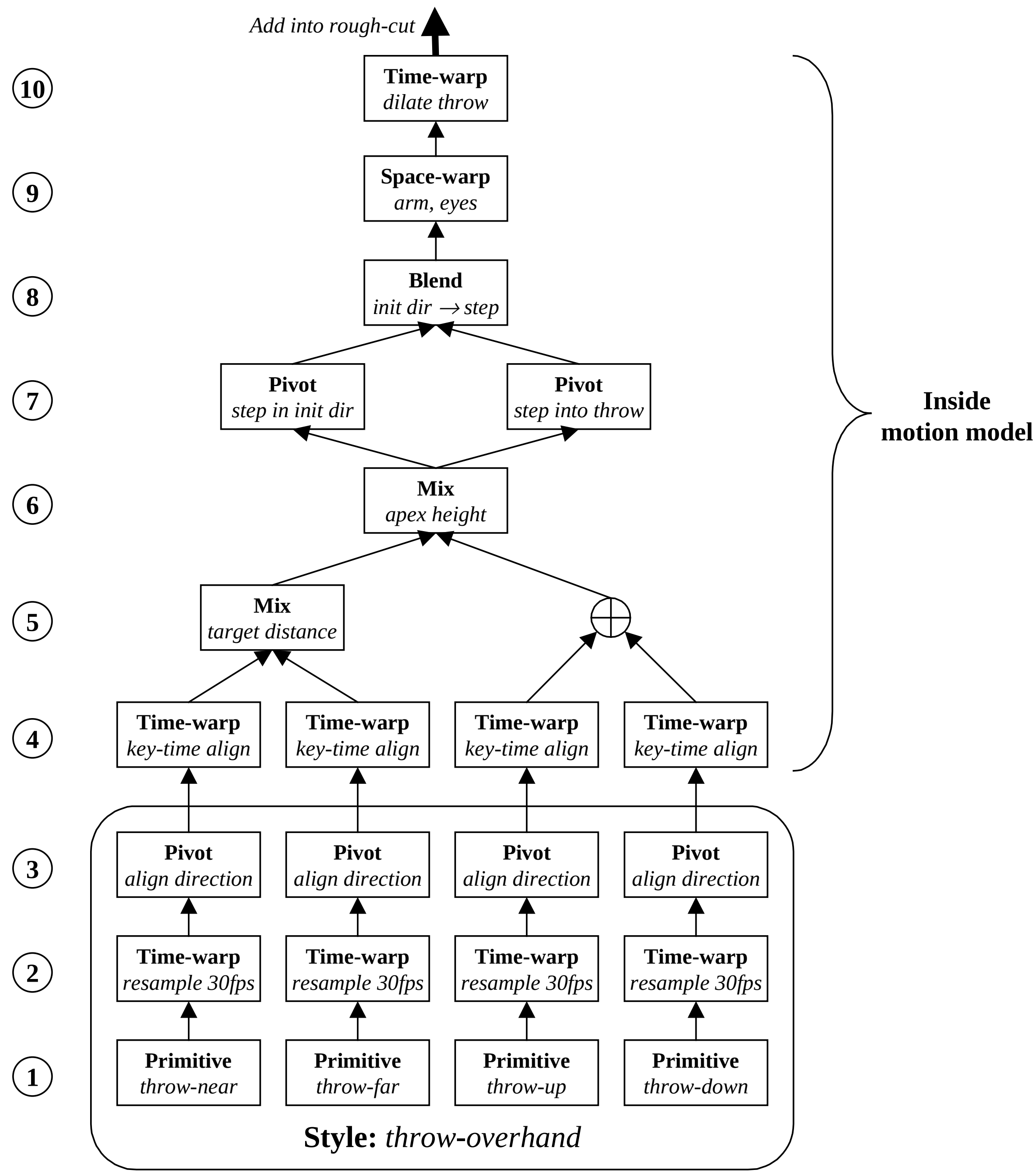


Figure 5-3: Clip motion expression for *throw.* This directed graph represents all of the transformation operators that contribute to creating a throw from any particular parameter settings. Layers 1-3 occur only once, inside the style object (in this case, *throw-overhand*). Layer 2 is necessary because the throwing motions in layer 1 contain 50 frames per second (fps) of data (since throwing involves large velocities), while our system operates at 30 fps. Layer 3 aligns all four base motions so that they begin facing in the same direction. Layer 4 temporally aligns the base motions so that we can interpolate them. In layer 5, the "or" operator does not represent any structure in the graph we create, merely that we only select one of *throw-up* or *throw-down* to combine with the interpolation of *throw-near* and *throw-far* in layer 6. Layers 7 and 8 accomplish the gradual step into the throwing direction from the initial direction the actor faces at the beginning of the action. Layer 9 uses poses generated by the motion model's invariants to ensure the actor regards the target at the key-times, and that his arm is aligned with the throwing direction at windup and release. Finally, layer 10 dilates the interval between windup and follow-through so that the hand has the correct speed when releasing the ball. At this point the motion is ready to be inserted into the rough-cut, along with its transition.

5.1.5.1 Geometric Constraints

At their core invariants are geometric constraints on functions of actors' bodyparts: positions, orientations, joint angles – in short, functions of all the types of handles described in Chapter 4 in our discussion of IK. An invariant is something that doesn't change with the passage of time or some other process; our invariants represent geometric expressions that must be satisfied over some (possibly zero) interval in the animation. Possibly the second most important aspect of the definition of a motion model (after deciding on its base motions) is specifying the invariants for the motion, since this is, in effect, stating what is truly important in the motion.

Our *throw* motion model, for instance, contains several invariants. First, like all humanoid motion models, it contains a set of position and orientation invariants that enable us to maintain foot plants; we will describe these in detail in section 5.3.1 when we discuss API's for motion models. Secondly, we define an *aim-at* invariant that is active instantaneously at the ball-release that allows us to adjust the position of the throwing arm so that the ball is launched towards the target. Finally, we enforce another aim-at invariant on the actor's gaze direction so that he looks at the target throughout. One might, at first, wonder how an instantaneous constraint on the throwing arm can result in a satisfactory throwing *motion.* The answer, which will be explained in detail shortly in section 5.2.5, is that our animation engine uses invariants not just to enforce pose constraints, but to define a continuous deformation of the underlying motion; thus, the effect of even an instantaneous invariant propagates through time to affect the whole throwing motion. In fact, this invariant (and many like it in other motion models) *must* be instantaneous – if we enforced it over the entire throwing motion, we would need to specify the precise direction in which the arm should be pointing at every instant of time, which would be practically impossible to do without eradicating many of the subtle motions of the arm present in the base motions.

5.1.5.2 Inter- Motion Model Communication

Invariants also facilitate communication and cooperation between motion models in two ways. First, all invariants applicable to a given actor are owned and managed by the actor, and shared by all the motion models contributing to the actor's motion. This allows a motion model interested in a particular invariant to easily discover which other motion models active for its actor care about the same invariant. This is especially important when attempting to layer motions, since a motion model that defines an invariant crucial to its operation cannot be layered on top of another motion model that defines the same invariant, but requires it to have a different value.

Second, invariants are the principle tokens used in parameter arbitration and optimization, which we will discuss in the next section. Most invariants involve a single IK handle, such as a control point on an actor bodypart, or the orientation of a bodypart; we can, therefore, associate the *value* of the handle under a given motion at a particular time with the invariant. Motion models use this capacity of invariants to determine, for instance, important aspects of the initial pose the actor possesses when the motion model begins its motion.

5.2 Operation – Five Stages

We will now present a detailed explanation of how a motion model uses the five components from the last section, together with its current parameter settings, to generate motion that satisfies the goals. In its full generality, the problem of generating even a single, primitive motion (as is done by a motion model) is extremely complex, and describing it all at once would result in a presentation that would be, at best, difficult to follow. Therefore, we have structured this document to describe the problem in phases, each phase adding a bit more of the full problem to the picture. In the remainder of this chapter, we will discuss the basic problem of a motion model generating its motion in isolation from other motion models and other actors, for use in an offline animation (*i.e.* not real-time or using a dynamically generated script). In subsequent chapters, as we introduce the topics of transitioning, layering, inter-actor interaction, and dynamic scripting, we will revisit the basic motion model operation, showing how the algorithms need to change to accommodate the more general problem.

We can break basic motion model operation down into five consecutive stages: parameter setting and optimization, base motion selection and combination, compositing the motion into the "working animation" (the *rough-cut* from Chapter 3), exporting control handles to be used for parameter manipulation and tweaking, and finally the transformation algorithm that deforms the motion to satisfy the invariants. We will now describe each stage in detail. Since some stages deal heavily in concepts we will introduce in later chapters, we will need to allude to the upcoming material, but we will keep it to a minimum. One may notice that these five steps break down the animation pipeline somewhat differently than the four steps of motion model execution presented in chapter 3.5. This is a function only of the level at which we are describing the same algorithms.

5.2.1 Parameter Setting/Optimization

Before a motion model can begin to synthesize an animation from its base motions, it must have concrete values for all of its motion parameters. But since one aspect of our solution to enabling rapid prototyping of animation is to allow the user to leave many of the parameters unspecified, we must be able to compute reasonable default values for all free variables. There are two levels to this problem; the first level, which we will discuss in this section, is the fairly straightforward problem of "filling in the blanks" using base motions as a guide; the second is the optimization problem of simultaneously determining values for the free parameters of all motion models so that the total motion is optimally coordinated. We will delay discussion of this problem until Chapter 7.

5.2.1.1 Default Parameter Values

We can compute default values for most motion parameters by using the parameters that *have* been specified to select and combine values from the base motions. For instance, when specifying a *throw* motion, we commonly omit specification of parameter Apex-Height (referring to Figure 5-2). As we will demonstrate in section 5.3.2, by examining the relationship between the intended target and the targets of the base motions, we determine how much influence each base motion will have on the final motion (the formula will be discussed in the next section). We then use these weightings to form a weighted sum of the height values associated with the base motions. For discrete parameters such as Strike-Rising, we use the weightings to determine which base motion value best fits the situation: if the throw will be predominantly downwards (*i.e.* contains a heavy weighting of the *down* base motion), then the arc will have no rising arc at all; if the throw will be predominantly upwards (*i.e.* contains a heavy weighting of the *up* base motion), then the ball will strike on its rising arc; otherwise, it will not. This formula comes from the same basic observations we use to determine how to combine the base motions.

Some default parameter values depend not on the base motions, but rather on the initial conditions for the motion model. Without needing to discuss transitions in detail, we can state that a motion model is cognizant that its actor may be in any pose and position at the beginning of the motion model's execution. Furthermore, in the context of a multiple motion model animation, the user should not need to specify the gross position and orientation parameters (Position and Orientation in Figure 5-2), since they can be calculated as a function of the actual starting pose and the default starting pose for the motion model. This calculation is common to and identical for all motion models designed for a certain character model class. When we present our API for motion models later in this chapter, we will describe the calculation specific to humanoid motions.

5.2.1.2 Legal Parameter Ranges

Because each motion model possesses knowledge of the kind of motion it is meant to be performing, we can generally perform a sanity check on all parameter values. Often the base motions themselves will define the boundaries of realizable motions, as the base motions *far-up*, *far-down, far-forward, far-left,* and *far-right* represent the furthest *reach* in each cardinal direction that an actor can actually accomplish while maintaining balance. In other cases, basic physiology and common sense suffices: even though our *fist-shake* gesture consists of a single base motion (and thus provides no legal range of targets at which the actor can vent his anger), we know that it would require an extremely limber person to hit targets directly behind them.

What action to take on discovering an illegal parameter value should be, to a certain degree, up to the user. We can allow the user to choose from a variety of actions:

- Clip the offending parameter value to the boundary of its legal range, and optionally inform the user that such a clipping occurred.

- Ignore the illegality and create the best motion possible, even though it may be implausible.
- In a future system, explore the addition or combination of other motions that bring the parameter into the legal range (for example, adding a *jump* to a *reach* when the target is too high above the actor's head).

5.2.2 Base Motion Selection, Combination and Deformation

When all of the parameters possess values, we have a complete description of the desired motion, and we can begin to translate the description into a rough motion. With the exception of certain motion models that create motion through physical simulation (such as that used by a ball while in a ballistic trajectory), all motion models follow the same two-part course: selection and combination of base motions, followed by deformation of the blended motion to meet invariants at key times. We stress that although we use invariants to generate IK poses for the deformation phase here, this is distinct from the persistent constraint satisfaction that occurs later in the pipeline. Much of the "content" of a motion model goes into the specific combination and deformation formulae for each motion model, and we will discuss strategies for creating these formulae in section 5.3, the Design section of this chapter. However, although the precise formulae differ between motion models, the basic structure and function is identical for all.

5.2.2.1 Combining Base Motions

Except for most gestures, motion models typically possess multiple base motions that span the parameter space of the motion. To create a motion that satisfies an arbitrary set of parameter values, we will need to begin with some combination of the base motions. There are five steps in doing so:

1. **Geometrically align base motions (Pre-process).** Before we can consider combining any of our base motions, we must ensure that they are compatible with each other within the context of the action they represent. The major area in which they may initially be incompatible is in the gross position and alignment of the whole motion. For instance, consider a scenario in which our actor performed the *look-forward* base motion while facing along the positive X-axis, but performed the *look-up* base motion while facing along the negative X-axis. A combination of the two intending to achieve a "look slightly upwards" motion will cause the actor to face in a different direction that depends on how far he is looking upwards. This is clearly undesirable. Therefore, when a style is first used, we examine all of its base motions to ensure that, in their initial poses, the motions all face in the same direction, and some identified point on the actor (we generally use the ball of the right foot, but any point will do) is in the same position in world space. We can accomplish this using a clip-pivot and some simple measurements for each base motion, and it need be performed only once.

2. **Generate base motion weightings.** Using the formulae specific to each motion model, translate the parameter values into a weighting vector for the base motions, the sum of whose components is 1.0. In our implementation, we do not allow negative weightings (which we would use to extrapolate) for any base motion, but this restriction could be removed if we improved our interpolation algorithm.

3. **Temporally align base motions.** Before we can blend any base motions together, they must all be time-warped so that their key-times are coincident. If footsteps are an integral part of the Motion but are not directly reflected in the key-times, the motion model designer may also use the timings of foot-lifts and strikes in the time-warp. We determine the actual times to which we will warp each key-time by weighting the key-times of each individual base motion using the weighting vector from step 1. Once we have computed the "actual times", we construct a clip-time-warp for each base motion that maps the base motion's key-times into the actual times.

4. **Combine base motions using the weighting vector.** With the base motions temporally aligned, we can finally blend/interpolate them. One could adopt a very general interpolation scheme such as the radial basis functions used by Rose [Rose 1998]. We have chosen a simpler scheme based on the clip-blend and clip-mix operators developed in Chapter 4. Of course, since these are binary blending operators, we will need to decompose our *n* base motion weighting operation into a sequence of binary operations. This is generally straightforward as the sequence follows directly from the logic used to determine the weightings in the first place, which we will see in section 5.3.2.

5. **Repeat, shifted in time, for repetitive motions.** For actions that repeat a basic motion or variations of a motion, we must repeat steps two through four for each repetition. A stipulation on the base motions for such motion models is that each base motion is recorded/captured in the context of a series of such motions, and that the base motion includes some "run-off" at the beginning and the end. This allows us to stitch all of the repetitions together simply by overlapping them slightly and clip-blend'ing them together. Examples of such actions are walking, hammering, and chopping with a knife.

5.2.2.2 Deforming a Blended Motion to Meet Precise Goals

If a motion model was designed using a brute-force sampling approach (defines multiple base motions along each dimension of possible variation), it is conceivable (although not guaranteed) that a blending of the base motions exactly meets the goals of the motion. The far more common case, however, is that the blended motion will have the gestalt of the desired motion, but will not meet the goals, for one of two reasons. First, motion interpolation is imprecise because the values of bodypart handles are nonlinear functions of the character model's joint angles. The consequences can be illustrated by an example: we have two *reach* base motions: one that reaches a target directly in front of the actor, and one that reaches a target 90 degrees to the right of the actor. If our desired target lies somewhere 60 degrees to the right of the actor, we would probably decide the best blending of the two motions is 1/3 of the *front* reach, and 2/3 of the *right* reach (since 60 is 2/3 of the way to 90). However, due to the non-linearity of the positioning functions and the complexities of human posture, the resulting motion will not exactly meet the goal.

Second, there may be residual variation between the blended and desired motion along parametric axes the base motions intentionally do not span. For example, the base motions for our *throw* motion model do not provide any variation in the target's lateral position, which is the reason *throw* requires only four base motions. This is indicative of our general experimentation in replacing dimensions of sampling with dimensions of transformation, as discussed earlier in this chapter.

Having established the need for motion deformation, we turn now to our deformation tools. When addressing deviations from the desired motion caused by imprecise interpolation, the combination of automatic IK-generated poses at specific key-times (dictated by the motion model's invariants) and a clip-space-warp suffice to correct the motion to achieve its goals naturally, since the deviation is typically small and well within the range that space-warping can handle. When addressing deviation due to intentional under-sampling along a parametric axis, we may require more powerful or specialized deformation operators from the clip family. Returning to the *throw* motion model, we will illustrate one such deformation. Before we do, however, we would like to point out that of all the motion models we created in this work, only the *throw* and *jump-both-feet* required this more sophisticated and non-automatable[22] deformation, and these are precisely the two motion models where we truly pushed on the envelope of reducing the number of base motions.

We derive the deformation that accounts for lateral variation of a *throw* target from two observations of human throwing. The first is that we generally take at least one step forward with the leg opposite the throwing arm, just prior to releasing the ball. The second is that this step is taken in the direction in which we are throwing the ball. The deformation we would like to achieve, therefore, is one that makes the actor step in the lateral direction of the target, rotating the entire body (and thus the throwing arm) along with it. This means that when the actor releases the ball we want him to be pivoted as if he had been facing in the throwing direction all along (applying a clip-pivot into the **target** direction). However, since the actor begins the throwing motion facing in the initial direction (applying a clip-pivot into the **initial** direction), we must somehow smoothly transition from one motion to the other. We do this by completely overlapping the two motions in time and applying a clip-blend that transitions from the **initial-direction** clip-pivot at the time of the first foot liftoff to the **target-direction** clip-pivot at the time of the prior-to-release step. Of course, there are limits to how much we can pivot while stepping, and due to the throwing style or layering of motions there may *be* no footsteps at all. We can account for any remaining difference in the actual and desired throwing direction with the standard IK/space-warp deformation applied to the direction of the throwing arm at the time of release, using an aim-at invariant.

5.2.3 Compositing Motion into the Rough-Cut

After the first two stages of motion model execution, we have a motion that begins in the correct general position and orientation and meets the basic goals dictated by the parameters. It does not, however, begin in the exact pose dictated by the completion of a predecessor motion (if any), nor does it satisfy its own extended invariants (those invariants that are active over some extended time interval). If the motion model were, in fact, the *only* motion model in the animation, all we would need to do at this point would be to run IK on each frame (or whatever other constraint-satisfaction algorithm we choose, such as spacetime) to complete the animation. However, motion models are designed to function in the presence of other motion models to create seamless, multi-action animations. The presence of other motion models requires that the action of one motion model naturally transitions into the motion of another. While the algorithms for creating transitions warrant an entire chapter's discussion (Chapter 6), the third stage of motion model execution deals largely with the consequences of having transitions at all.

We begin with a refresher on the rough-cut, which you will recall from Chapters 3 & 4 is the sole instance of a specialized data structure called a clip-composite. A clip-composite allows us to arrange the individual results of many motion models (in the form of clips) into a single, continuous animation by letting us also add special clips positioned between motion model motions to transition between them. It allows us to do this individually for each bodypart-group[23], and allows motions to be layered on top of each other. The main job of this third stage is to correctly position the motion model's "final" motion into the rough-cut, ensuring that it smoothly and naturally fits with the motion already in the rough-cut, most notably the motion temporally preceding the current motion model's motion. This process has several sub-steps, and is the same for all motion models:

0. **Specify invariants in initial time frame.** Although this action has already taken place in the previous step (in order to properly deform the blended motion), we list it here because it has ramifications on later sub-steps. The key facts to note are that the invariants are specified in a timeframe that does not account for transitions, and that, once specified, invariants are not destructible (for the current animation computation) because they are shared with other motion models.

---

[22] Automatic generation of code to deform motions suffering only from imprecise interpolation is entirely conceivable. Automatic generation of code to deform motions requiring more sophisticated deformations is not likely, given that it requires creativity and an understanding of the desired motion to design the deformation.

[23] Recall that a bodypart-group, defined for a particular character model class, represents either the set of individual bodyparts that make up a unit that functions as a whole (as the hand, forearm, upper-arm, and shoulder make up the Right-Arm group), or the set of motion curves that drive those bodyparts, depending on context.

1. **Compute duration of transition from predecessor motion.** Computing the duration of the transition from the previous motion into the current one is described in detail in the next Chapter. What we need to know about it now is that the transition duration will henceforth specify the timing between the beginning of the motion and its attainment of its first goal – the "rising action" portion of the motion described in section 3.2.3.

2. **Remap invariants and re-specify final deformation.** If the transition duration differs from the duration of the initial rising action phase (which it generally will), then the timing of the entire motion will shift, and we must remap the activation times of the invariants to match the new timing. We must also, at this time, re-specify the final clip-space-warp deformation from the last step, for reasons that will become clear when we discuss layering in the next chapter. One might ask why we do not simply wait until after we have computed the transition duration to specify the invariants and deformation in the first place, thus eliminating the need for remapping. The answer is that the algorithm for computing transition durations relies on obtaining the correct, goal-satisfied pose for this motion at the end of the rising action interval; we can only provide the correct pose after imposing the invariants and specifying the clip-space-warp deformation.

3. **Compute the motion curves for the clip-transition.** The final thing we must do before we can add the motion into the rough-cut is compute the actual motion curves that, over the transition interval, will smoothly segue from the previous motion into the current one. Since the transition is, of course, itself a motion, the motion curves are placed in (and actually computed by) a clip-transition; this will be discussed in the next chapter. The reason we must compute the motion curves for the transition separately from the transition timing also has to do with layering.

4. **Add the motion into the Rough-Cut**. Since we now have the motion, the transition into the motion, and all of the invariants properly aligned in time, we can add the motion and its transition into the rough-cut.

After completing these steps the motion is incorporated into the larger animation, and the only task left to complete the animation itself is to satisfy the invariants of all motion models using our constraint satisfaction engine. Some of the concepts introduced in later chapters, such as layering and inter-actor interaction, will require minor modifications or additions to the actions in this step of motion model execution, but the core actions will remain unchanged.

5.2.4 Update and Export Control/Tweak Handles

As we mentioned earlier in section 5.1.2, each motion model parameter is strongly tied to an interaction technique. Most of the information required to engage in the interaction is stored either explicitly or implicitly (in its type) with each parameter. We have found that, particularly for the direct manipulation interaction techniques, the interaction is most naturally performed if, when the parameter is activated for editing, we update the display presented to the user so that it reflects the state of the animation at the time most relevant to the parameter. For instance, when the user activates the Target-Pos or Target-Obj parameters for editing, we would like to display the instant in the animation at which the thrower currently releases the ball, as this most accurately reflects the “state of the world” for purposes of aiming. Note that this is *not* the time at which the actor actually takes aim, but if the actor is supposed to hit a point in space near a moving target, we would like to see where the target will be when the actor releases the ball. Similarly, in an animation where the actor locomotes over a wide area, when the user wishes to change the initial position of the actor as he begins walking (the Position parameter), we do not want to display the actor at the end of his travels, but rather at the beginning. Therefore, the motion model must set the “edit time” of each of its parameters, based on its current situation in the rough-cut (which simply determines an offset) and the decision of the motion model designer on what time within the motion model is appropriate. Some parameters, such as the position of a static object/actor, do not possess any logically best editing time, and for these we can specify that the display does not need to be changed when the parameter is activated for editing.

In addition to control parameters, we also must update and activate any tweak constraints. Recall that a tweak is a geometric constraint (instantaneous or extended) that the user may create via direct manipulation of the actors at specific times in the animation. Tweaks are meant to provide very fine control over the animation and effect small changes in it; they are ignored during motion model execution, affecting only the final, invariant satisfaction step. Once created, a tweak attaches itself to the motion model most strongly influenced by it, and that motion model translates the tweak’s time-of-application into its own canonical time so that, as the animation is further modified, the tweak will be applied at the same time relative to the motion it is modifying. At this point in the motion model’s execution, therefore, we must translate all tweaks’ application times back into the updated animation time in which the motion model is situated, and export the tweak itself to the final stage in the animation pipeline, the deformation of the animation to achieve invariant satisfaction.

5.2.5 Deformation to Satisfy Invariants

Every motion model in an animation executes the above four steps individually (except for the global parameter optimization which is, of course, performed communally for all motion models), in a partial order determined by predecessor/successor relationships between motion models. The final step, cleaning up the motion so that invariants are satisfied at every frame, is performed only once, acting on the entire animation collected in the rough-cut. We must make this separation between per-motion model work and one-for-all work for two reasons: invariants and tweaks associated with a motion model may have an influence on neighboring (temporally and with respect to layering) motion models, and satisfying the invariants of one motion model in isolation of the others could lead to discontinuities at the transition interfaces.

Specifically, the goal of this final stage of the animation pipeline is to minimally deform the animation present in the rough-cut so that it meets the constraints imposed by all the invariants of all motion models throughout the animation, *and* introduces no discontinuities into the animation. We have already noted that we could utilize any of several pre-existing engines to solve the constraint satisfaction problem, such as Gleicher's space-time constraints engine [Gleicher 1997] or Lee's iterative IK hierarchical b-spline algorithm [Lee 1999]. Since we are ultimately interested in real-time motion synthesis as well as offline content creation, we decided to design a new constraint satisfaction engine that focuses on speed, at the possible expense of some quality. Due to time constraints, we were not able to compare its performance to that of the others mentioned here; we leave judgment of its quality to the evaluation chapter (8). We will describe the engine itself in detail when we talk in chapter 7 about synthesizing a coherent animation from all of the pieces presented in these preceding chapters.

5.3 Design

Given that motion models produce high-quality motion from high quality base motions (which the reader can judge for himself from the movies at the accompanying web site), the success of the whole approach rests on two criteria: Can the intended users actually achieve results similar to those we presented more easily than they could using currently available techniques? Are the methods presented here extensible – that is, can we create a large library of motion models and styles with a reasonable amount of work, and achieve results similar to those we presented here? We will answer both of these questions as best we can in chapter 8, but we would like to briefly focus on the second question here, since its answer largely depends on the process of designing and implementing new motion models.

Our response to this question consists of two parts. First, in section 5.3.1, we describe a set of tools intended to aid in motion model development. These tools include implementations of many of the lower level algorithms necessary for motion synthesis, including IK, transition generators, and our motion expression language. In combination with a structured template for building motion models, these tools allow the motion model designer to focus almost exclusively on the primary task of encoding motion transformation rules. Second, in section 5.3.2 we try to give greater insight into several aspects of the design process by presenting guidelines for choosing and designing the components of motion models described in section 5.1, and by discussing in much greater depth the design of rules for motion combination and transformation introduced in section 5.2.2, using the *throw* motion model as an example.

5.3.1 An API for Motion Models

The ideal for a motion model from a design point of view is an abstraction that efficiently and easily encapsulates an animator's knowledge about how things move and how different motions are related, and uses that knowledge to guide motion transformation. Even assuming (as we do) that it is possible to extract useful motion transformation rules from animator knowledge, the current state of human-computer interaction necessitates that some programming will be required to express the rules in a form usable within a motion model. This is why we believe that, except in rare cases, motion models can best be created by a two-person team consisting of an animator and a technical director (TD), *i.e.* a person who understands graphics and animation, possesses strong programming skills, but most likely does not have animator skills. To keep the programming aspect of motion model creation as minimal and simple as possible, we provide an application programmer's interface (API) to motion models. In the remainder of this section, we will present various concepts, elements, and features of this API, making distinction between what is implemented in our prototype system and what would be contained in the ideal.

Elements of the API Available to All Motion Model Classes

- **Motion Parameters.** We provide built in facilities for handling all aspects of interaction with parameters of the types mentioned in this chapter. The motion model designer need do no more than declare each parameter and provide bounds (if applicable) on its value. The current value will then be available whenever the motion model recomputes its motion.

- **Clip Motion Operators.** Obviously, the motion operators comprising the clip family are a significant part of the API. We found that a small number (eleven) of general operators sufficed to create all of the motion models in this thesis save one; the operators are blend, composite, hold, mix, pivot, primitive, reflect, repeater, select, space-warp, and time-warp. Given that the structure of a clip is both compact and very well defined (see appendix), it is also straightforward to derive new operators as necessary, as we did for the *jump-both-feet* motion model.

- **Flexible Transition Generators.** We have developed a flexible transition generator, in the form of a member of the clip family, that is not motion model specific, which we have used for all of our motion models. A small amount of motion model specific guidance can be supplied to guide the generator, as we will describe in the next chapter, but all of the transition generation, as well as the facilities for compositing the motion into the rough-cut, are part of the API.

- **IK Poses.** Two very common operations used in motion models are querying the value of an IK handle while an actor is in the pose specified by a particular clip at some time, and computing an entire pose that minimally deforms an initial pose to satisfy a set of IK constraints. We provide easy access to these functions, using the same engine utilized by the global constraint satisfaction algorithm.

- **Collision Detection and Avoidance.** Collision avoidance, detection, and response is one of the major omissions from our prototype system. Currently, we provide half-plane constraints that keep a point (presumably on a bodypart) on one side of a mathematical plane, which can be used to prevent some types of interpenetration. However, we have documented several cases where the simple blending of two base motions (each of which is fine by itself) will result in a motion that contains interpenetration of bodyparts. In offline applications, this is not a significant problem, since we can quickly eliminate the interpenetration with a minor tweak, in most cases. For demanding real-time applications, however, the system itself must be able to detect when interpenetration occurs, and how to alter the motion appropriately. There are many collision detection schemes that we could incorporate into our approach (although they are computationally expensive) [Moore 1988, Baraff 1992]. Collision avoidance is a more difficult problem, although it has been studied extensively in the robotics and AI communities [Latombe 1991, Koga 1994]. At a bare minimum, we would desire that the API allow us to easily determine which bodyparts of which actors are intersecting in a given set of poses. Any attempts at providing collision avoidance mechanisms must be (at least) actor specific, and possibly motion model specific as well.

- **Physics Simulation.** In our current prototype, physics simulation is applied only selectively in isolated places to produce motion curves. For instance, within the motion model that determines the position and orientation of a Rigid Body, when it is detected that the object has been released from another actor's grasp, it applies differential equations to compute the ballistic trajectory of the object, storing the trajectory as motion curves. However, it should be straightforward to incorporate forward dynamics simulation into our constraint satisfaction engine, making selective or total application of physical laws and forces to specific actors an integral part of the API. We will describe our ideas in this area in the "future work" section of chapter 9.

- **Style.** An implicit but important aspect of the API is that we do not need to do any programming to add a style for a motion model class. The API provides a "translator" object that reads the definition of any style from a file and converts it to a set of base motions and annotations, then indexes the style by name with the motion model class for which it is defined. From the motion model's point of view (and thus the designer's), all styles are represented by the same type of translator object, which is queried to retrieve base motions and annotations.

- **Motion Model Template.** The necessary functionality of a motion model is highly structured, which has allowed us to construct a basic template that we used to construct all of our motion models after the first few. This insures that no requisite functionality is inadvertently omitted from the motion model, and speeds up the coding of the more mundane functionality.

Elements of the API Specific to Humanoid Class

An important part of the API is the ability to specialize the basic motion model class to encapsulate knowledge and functionality that becomes available to all motion models derived from the specialized class. The most thoroughly developed specialized class in our prototype is the **humanoid**, which contributes the following to the API for motions designed for humanoids:

- **Footsteps.** Specifying and managing the motion of the feet is one of the more complicated subtasks in creating human motion. Since we model the foot as a jointed assembly (hinge joint at the ball of the foot), there are many possible combinations of invariants that can be used to specify and constrain the foot's motion in different situations. We have attempted to reduce the complexity by defining a small set of states in which the foot can be at any time, with each state corresponding to a specific set of invariants. They are as follows:

  1. **Heel-Lock:** Weight is on the heel, and the foot can pivot about the heel. Uses position and orientation invariants at the heel of the foot.

  2. **Toe-Lock:** Weight is on the ball of the foot, about which the foot can pivot. The heel is free to rise and fall, although it is constrained from passing through the floor via a half-plane constraint on the heel. Uses position and orientation invariants at the ball of the foot, plus the previously mentioned half-plane constraint.

  3. **Free:** The foot is not in contact with the ground, and no invariants apply, save those utilized for collision avoidance.

  This is obviously a vast simplification, and as such, it will perform poorly when applied to some types of motion. For example the action of rolling onto the side of one's foot, as people sometimes do while standing idly, would be difficult to describe using any of our three states. Nevertheless, we have found these three states sufficient for controlling all of the motions contained in the thesis. Furthermore, this simplification allows us to very succinctly describe the motion of the feet that is present in any particular example motion, as we must for the base motions in a style. Given that each foot is always considered to be in one of the three states, and that the actual motion of the foot is contained in the motion curves, all we need do in order to effectively control the motion of the feet is to supply the times at which state transitions occur.

  We simplify the description task further by defining the **footstep** annotation, which contains four elements: **foot -** the foot whose motion is being described; **liftoff -** the time in the base motion at which the foot enters the **free** state; **heel-strike –** the time at which the foot enters the **heel-lock** state; **toe-strike –** the time at which the foot enters the **toe-lock** state. Any of the temporal elements can take on the special symbol ***not-present***, which, for **liftoff** means that the foot begins the motion midair, and for **heel-strike** and **toe-strike** means the foot does not pass through the corresponding state between its own and the next **liftoff**.

This abstraction makes annotating the foot motion in any base motion, from standing to walking to jumping, quick and easy. Also, because we propagate the abstraction into the motion models, we facilitate several important computations. One such computation is the time-warping of base motions that allows us to blend them properly (as described in section 5.2.2.1 above): sometimes the correspondence between footsteps in the various base motions is important, but (for any of several reasons) this information is not reflected in the key-times of the corresponding motion model. Therefore, we provide a "footstep-aware" base motion blending function that can incorporate the timing information contained in the **footstep** data structures of each base motion when computing the temporal correspondence between base motions.

- **Stance Analyzer.** A motion model, by default, generally assumes that when it takes control of its actor, the actor is in approximately the same stance as it would be if there had been no preceding motion. By "stance" we mean the support provided by the feet – that is, that the same feet are in contact with the ground – and the starting position and orientation of the actor as he begins the motion. The existence of the **footstep** throughout motion computation allows us to easily detect when discrepancies between actual and desired stance exist. The API provides a function that analyzes the discrepancy (if any), and either inserts, deletes, or extends footsteps during the first phase of the motion to eliminate the discrepancy.

  This function then uses the foot positions and the direction in which the actor is facing at the end of the preceding motion to determine how best to match the starting position and orientation of the motion model's motion to the preceding motion. We will discuss several subtleties involved in this computation in the next chapter when we describe how to compute the geometric aspect of transitions.

- **Balance Invariant.** A principal cause of awkward poses (and thus, motions) is ignoring the need of a humanoid to maintain static balance while performing non-locomotive actions. Utilizing our techniques, this typically occurs when the actor is asked to achieve a goal that is completely outside the span of a motion model's base motions, in which case we rely on a combination of extrapolation and IK-induced deformation to achieve the goal. In the example of a *reach*, for instance, this may cause the actor's center of mass to pass out of the area for which the actor's stance can provide support; in a true simulation of a humanoid, the actor would fall over, but in our system does not because the motion model has not told him to do so. We can prevent such erroneous situations by providing a balance invariant, similar to that of Phillips [Phillips 1991]. A balance invariant (which our current API does not provide) would determine the actor's area of support on the ground by examining the stance, and then apply a restoring force or constraint on the actor's center of mass whenever its projection onto the ground leaves the area of support. Each motion model would then give the user the option of enabling or disabling the balance invariant. Naturally, both the **footstep** and all the concepts that make use of it (stance, balance, *etc.*) could easily be extended to quadrupeds, monopods, or hexapods.

- **Gesture.** Alternately to the definition of a gesture that we gave in section 5.1.2, a useful structural definition is as a motion model that utilizes only a single base motion. This definition is useful because a number of simplifications to the structure and operation of a motion model occur in this case, since the motion model no longer needs to blend base motions. We have codified and factored out these simplifications into a further specialization of the humanoid class, called a **gesture**, from which we can quickly derive new gestural motion models. Using the gesture class, we were able to create five new motion models from annotated base motions in a single afternoon.
- **Specialized Transition Generators.** We will discuss in the next chapter how to create transition generators tailored to a specific actor, motion model, or style. Some aspects of transition generation are common to most or all humanoid actions; for instance, rules for avoiding self-intersection, and general rules for deriving transition durations when no more-specific information is available. These can be collected into a "humanoid transition generator", which motion models are free to specialize further.

5.3.2 Rule Design and Tool Usage

Having the basic design of the motion model API in hand, we can proceed to describe in detail the process of building motion models. We have already presented most of our findings and experience in how to choose motion model components in section 5.1. However, motion transformation is at the heart of motion models, and we have not yet given any detailed instruction on how to transform base motions into successful final motions. In the remainder of this section we will present several rules and guidelines for using the API, derived from lessons we learned in the process of creating more than a dozen motion models. Also, as we introduce new concepts in later chapters, we will add to these rules where appropriate.

5.3.2.1 Angular vs. Cartesian Interpolation

A very common parameter for action-oriented motion models is a target position or object. Of the motion models we have implemented, *throw, reach, jump-both-feet, peer, foot-shuffle,* and *fist-shake* possess such a parameter or parameters, which the motion model must translate into a blending of base motions. The first step in this translation is to compute a weighting for each base motion in the blend, and this requires an interpolation scheme by which we judge the target parameter's relationship to the base motion targets. Some common examples of interpolation schemes are Cartesian linear, bilinear, and trilinear interpolation, for interpolating one, two, and three dimensional quantities, respectively. For the task at hand, we are actually interested in the *inverse* of the interpolation function at the heart of each scheme: each of the above schemes defines functions for computing a value given weights for the proper number of base val-

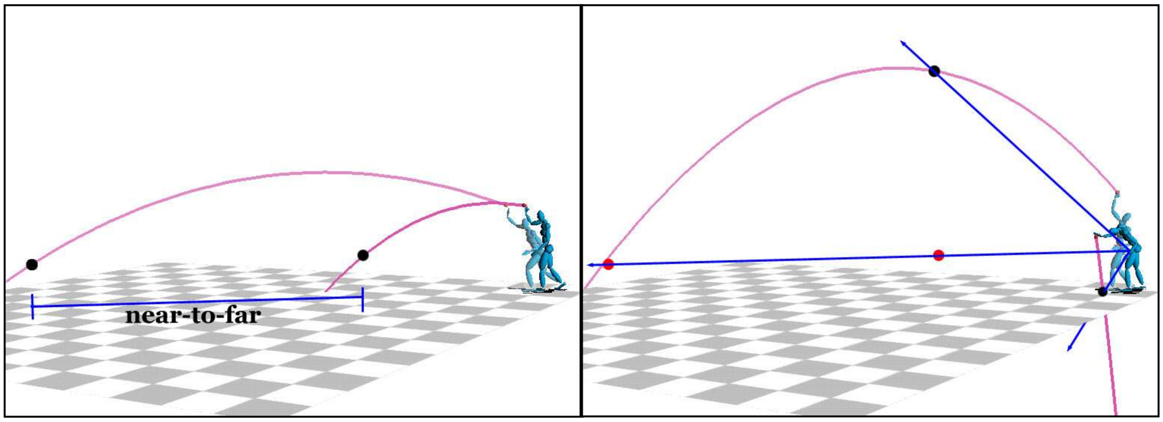


Figure 5-4: Computing *throw* base motion weights. On the left, we show the thrown trajectories of the **near** and **far** base motions, with the intended targets denoted as large dots. We compute the relative weighting of **near** to **far** by projecting the intended target onto the segment **near-to-far** and applying the inverse of linear interpolation. On the right, we show the thrown trajectories of the **up** and **down** base motions and their targets, as well as the targets of the **near** and **far** base motions along the horizontal line. The dark lines show the angles of elevation (calculated from the rest-state hips as origin) of each target, which we use in computing the relative weighting of **up** or **down** to the blend of **near/far**.

ues, while we are interested in computing the weights; however, the inverse is generally completely dependent on the scheme itself.

We have found that two different general schemes are useful in computing base motion weights: Cartesian rectilinear, and angular. Actions that involve moving the body or a part of the body *to* a specific place, such as *reach, jump-both-feet,* and *foot-shuffle* conform easily to trilinear or bilinear Cartesian schemes, since the full position of the target in a 3D lattice (*reach, jump-both-feet*) or a plane (*foot-shuffle*) is significant to the motion. Actions that involve pointing or aiming at something, such as *peer* and *fist-shake* differentiate their base motions by the direction of the target with respect to the actor, and do not depend on the distance of the target from the actor; thus an angular expression of the target with respect to the base motions (spherical coordinates without the radial component) is more appropriate than a rectilinear one. Sometimes, as we will see for *throw* in the next rule, a combination of Cartesian and angular interpolants best fits the base motions.

5.3.2.2 Multidimensional Blending

Combining multiple base motions according to arbitrary sets of parameters involves both the problem of computing base motion weights from the parameters, and then devising a means of computing a composite motion from the weighted base motions. We have already begun to discuss the weight-computation problem, and we will now complete our treatment with our solution for the *throw* motion model.

If we have constructed a motion model with numerous base motions that densely sample the task space defined by the motion model's parameterization, then a tri-linear grid-based (rectilinear or angular) interpolation scheme such as that developed by Wiley [Wiley 1997] would suffice to compute base motion weights. In a motion model such as *throw*, where we have distilled a large task space into just four base motions, we must find another solution, one that considers the relationship of each base motion to the task space. We begin by noting the dimensions of variation contained in the base motions: since there is no lateral variation in the target among the base motions, we are left with distance-to-target, as represented by the difference between the **near** and **far** base motions, and the angular elevation of the target, in an arc that encompasses all four base motions (with **near** and **far** taken together). These metrics are represented in Figure 5-4.

We use the distance-to-target metric to compute the ratio of **near** to **far** weighting, *nf* like so, where **targetPos** is the desired target parameter:

Let $$\begin{aligned}\mathbf{nearFar} &= \mathbf{target}_{far} - \mathbf{target}_{near}\\ \mathbf{relTarget} &= \mathbf{targetPos} - \mathbf{target}_{near}\end{aligned}$$ then $$nf = \frac{\mathrm{proj}\left(\mathbf{relTarget} \rightarrow \mathbf{nearFar}\right)}{\left|\mathbf{nearFar}\right|}$$

If we allow extrapolation of base motions, we can allow *nf* to take on any value. Currently, we limit our base motion blending to interpolation, so we clip *nf* to the range [0, 1]. This computation is typical of our Cartesian schemes. For instance, *reach* performs a similar computation along each of its principal axes, after first determining the two base motions between which the target lies in each dimension.

However, we still need to determine the weights of **up** and **down**. We observe first that a throw should never contain contributions from *both* **up** and **down**, since we would pass through a purely forward throw in-between. On the right in Figure 5-4, we show how the angle of elevation of the target, as measured from the actor’s hips in a neutral stance, can be used to determine the ratio of either **up** or **down** to a purely forward throw. This works well for generally downward throws – specifically, throws where the target is below the thrower’s hips *and* the Apex-Height parameter indicates that the trajectory contains only a downward component. However, if we measure the upward component only by the target position, the throwing motion will not change in response to how high we throw the ball on its way to the target. Therefore, if Apex-Height indicates that the trajectory has both an upward and a downward component, we measure the angular elevation of the *apex* of the trajectory rather than the target. We can now compute the base motion weights:

If trajectory has upward and downward components:

$upF =$ ratio of angular elevation of trajectory apex to **up**’s apex

$weight_{near} = (1.0 - upF)\ (1.0 - nf)$

$weight_{far} = (1.0 - upF)\ nf$

$weight_{up} = upF$

$weight_{down} = 0$

If trajectory has only a downward component:

$downF =$ ratio of angular elevation of target apex to **down**’s target

$weight_{near} = (1.0 - downF)\ (1.0 - nf)$

$weight_{far} = (1.0 - downF)\ nf$

$$weight_{up} = 0$$
$$weight_{down} = downF$$

Note that this represents a convex sum of the base motions, since the weights sum to 1.0 and there are no negative weights. We will include the weighting formulae for each motion model we have constructed, in the Appendix. For now we move on with *throw*: now that we have the weightings, we can actually blend the base motions according to the weights. If our motion curves described Euclidean, linear functions, then we could perform a simple component-wise linear combination to achieve the blend. However, since we use quaternion-valued functions[24] for our motion curves, we cannot (Grassia and Shoemake [Grassia 1998, [Shoemake 1985] each describe important aspects of this situation). We have two viable alternatives: a mathematically sophisticated non-Euclidean blending function such as the radial basis function employed by Rose [Rose 1998], or constructing a blending hierarchy from binary blending operators based on Shoemake's SLERP operation [Shoemake 1985], like our clip-mix and clip-blend. For the sake of expediency and simplicity, we have chosen the latter.

In this situation, where we are creating a blend of the entire set of motion curves and the blend is constant over the duration of the motion, the differences between clip-mix and clip-blend are irrelevant. Therefore we will describe the blending of the *throw* base motions using a generic SLERP operator, which takes two sets of motion curves plus a scalar between zero and one, and returns a set of motion curves each of whose members is a curve that is 'the scalar' percent of the way between the corresponding motion curves in the original sets. The blended motion for *throw* in terms of the intermediate coefficients defined above is then:

If trajectory has upward and downward components:

SLERP(SLERP(**near, far**, *nf*), **up,** *upF*)

If trajectory has only a downward component:

SLERP(SLERP(**near, far**, *nf*), **down,** *downF*)

If the motion model contained more axes of variation, the blend would be correspondingly deeper. For instance, *reach*, which blends along three axes, contains three levels of SLERPs.

5.3.2.3 Interpolation vs. Warping vs. Style for Capturing Variation

One of the central questions that arises in designing a motion model is which of the several transformation mechanisms we provide is most appropriate for capturing all of the variation in the action. As we have already stated in section 5.1.1, we can capture infinite variation by increasing the dimensionality and density of our base motion sampling and employing interpolation. However, for performance, acquisition cost,

---

[24] Of course, the global position function for each motion is linear, and we can and do employ simple linear combinations for them.

and storage reasons, we benefit from relying as little as possible on interpolation of large data sets. Therefore, we will now present some guidelines as to the types of variation for which we can employ the other main techniques at our disposal, space-warping and styles.

The combination of inverse kinematics (or spacetime constraints) and geometric motion warping give us a powerful tool for smoothly deforming motions to meet new goals. Since we have already discussed the problems of scale-mapping in Chapter 4, we will limit our current discussion to the more common displacement-mapping style of motion warping, which is the algorithm employed by our clip-space-warp. As a rule, the quality of result obtained from warping degrades with the magnitude of the requested displacement. This is a consequence of more distantly-related motions differing in subtle ways not recoverable by IK algorithms. For instance, suppose we have a hammering motion where the actor is using a tack hammer on a small nail, but we need a hammering motion for a large, stubborn nail, which will be much more energetic. If we were to apply motion warping to the original motion, we would employ IK on the pose where the hammer is raised, and cause it to be raised considerably higher. Now, a good IK algorithm may produce a pose that involves a deformation not only to the entire hammer arm, but also to the back. However, (especially in an automatic setting) it will almost certainly not add any motion to the other arm, or the head, both of which we would expect to see in an energetic hammer swing. Therefore the motion of the hammer arm will not match that of the rest of the body.

A less obvious but more insidious artifact of motion warping is that its performance depends also on the *direction* of the desired displacement relative to the original motion. If we consider a motion such as a step or a swing of the arm, we can easily see that the main motion is occurring in a specific direction. If we now add a displacement either in the same direction as the motion or in a direction transverse to the motion, the resulting motion will look fine. However, if we add a displacement in the *opposite* direction to the motion, attempting to either compress the extremes of the motion or reverse its direction, the resulting motion will overshoot its goal (displacement key) before achieving it, which is quite unnatural looking. This is caused by the additive nature of motion warping, and its effect is illustrated in the online examples, at http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter8**,** the third example in the section titled "Warping for Transitions: Good and Bad." Keeping the magnitude of the displacement small mitigates this effect.

In this thesis we propose the use of styles for capturing most forms of stylistic variation in motions. The observant reader will notice that while approaches such as Verbs and Adverbs [Rose 1998] allow base motions of arbitrary styles to be blended at will, we have specifically limited our motion models to allow application of only one style at a time. The reason for this limitation is that styles contain discrete annotations, such as added invariants and footsteps that may not be consistent or interpolable with the discrete annotations of other styles. We leave it to future work to devise a means of resolving these potential inconsistencies, but for this thesis enforce the concept of single-style application. This means that we should not employ styles to capture any variation for which we may be interested in intermediate values. For example, if we had captured performances of throwing both a light object and a heavy object and made the performances into the two styles *throw-light* and *throw-heavy*, we would not be able to generate a believable throw for a "middle weight" object. However, if we instead doubled the number of *throw* base motions from four to eight, we could enable a "ball weight" parameter and generate believable throws over the spectrum from light to heavy objects simply by adding another layer to our base motion interpolation scheme. It is far less likely that we would be interested in a throw that is "part sidearm, part overhand," so it is safe to encode these variations as styles, as we have done.

5.3.2.4 Persistence of Invariants

We actually use invariants for three different purposes inside a motion model: to communicate among motion models, to maintain geometric constraints and invariants over extended time intervals, and as the language for expressing the pose deformations that drive motion warping. We are concerned now with the last of these purposes, and wish to give some guidelines on the usage of invariants in this role. Let us consider a simple example in the *fist-shake* motion model, where the actor shakes his fist at some target, which is a user parameter. Since this motion model is implemented as a gesture from a single base motion, we will need an aim-at invariant on the arm doing the shaking in order to control the direction in which the shaking is occurring. Our first impulse might be to effect any change in direction by applying the aim-at invariant over the entire shaking duration; however, this would result in a significant loss of high frequency content from the motion, since the shaking naturally causes the arm's direction to fluctuate *around* the principle direction-of-aim. In practice, this results in a motion that looks markedly strange. However, because our transformation engine utilizes motion warping, we can simply apply the invariant instantaneously at the beginning of the shaking duration and at the end. Because the activation of an invariant causes a displacement key to be created in the transformation problem passed to our global constraint solver, this results in a constant displacement being applied to the motion over the shaking duration, leaving the high frequencies intact. We successfully employ this technique in the *reach* and *throw* motion models as well.

5.4 Further Detail

In the appendices we will provide a schematic of the specific structural and functional components of motion models, as well as the clip operators that form our transformation language. For each of the motion model, actor, director, and clip classes, we will list and describe the main data structures and methods. We will also include some sample motion model source code.

6 Motion Combinations

Other than the details of our constraint satisfaction engine, we now know how to generate primitive actions via motion models. Most interesting behaviors and animations, however, consist of many primitive actions, sometimes executed in concert, other times sequentially. In this chapter we will consider various ways in which we may need to combine motions, and study how we can execute these combinations automatically, using only per-primitive-motion knowledge. By examining existing approaches and their shortcomings, we will see that in order to automatically combine motions effectively, we must have a certain amount of knowledge about the motions being combined – exactly what motion models give us.

This study of “motion combinations” fundamentally deals with transitioning an actor seamlessly and consistently from performing one action to performing another. The simplest form of transitioning is segueing, in which we transition from a completed action into a new action that is the sole focus of the actor’s attention. To create good segues, we must consider not just the *geometry* of the segue (*i.e.* the motion curves that dictate the actor’s trajectory during the segue), but also the *timing* of the segue, each in light of the styles in which the two motions are performed.

In section 6.1 we will begin by precisely defining a transition and the qualities of the two adjoining motions we should attempt to preserve. We will examine the existing approaches to segueing and note their shortcomings, then attack separately the problems of transition timing and transition geometry. In section 6.1.3 we will describe how we are able to use the knowledge contained in motion models to calculate better values for transition entrance/exit times and durations, and suggest a simple machine-learning approach that might allow us to do even better. In section 6.1.4 we will describe our current solutions to producing transition geometry, and also propose a method that could address the shortcomings of our current best-solutions. We conclude our discussion of segues by addressing some architectural issues in section 6.1.5, and the need to avoid actor self-intersection during transitions in section 6.1.6.

In section 6.2 we move beyond simple segues to consider the layering of multiple motions on top of each other, to be executed simultaneously. We will see that this involves transitions, but also adds constraints on when motions can and should be combined in this way (sections 6.2.1 and 6.2.2). In section 6.2.3 we will present the different operators we developed for combining layered motion curves, and discuss the conditions and granularity of motion by which we can apply them. We end our discussion of layering by introducing *resumptions* in section 6.2.3.3 – secondary transitions that we must compute and apply to the terminus of layered transitions so that they can rejoin the motions upon which they are layered.

Finally in section 6.3, we will discuss the necessity and implementation of *pausing* motions in mid-execution and *holding* the terminal pose of one motion while executing another. Throughout the chapter, we will generally limit the depth of our discussion to the concepts and their application; many of the details about the algorithms fit into the next chapter's presentation of synthesizing complete animations from motion models.

We will reiterate it at relevant points throughout the chapter, but we want to mention now that most of the examples for this chapter are in the form of online animations, since subtleties of timing and motion are difficult to convey in print. The examples are located at:

http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter6.

6.1 Transitioning One Motion into Another

Our most basic motion combination, segueing, consists of little more than transitioning between motions. Since transitioning is integral to other types of combinations as well, we will spend the bulk of this section analyzing transitioning in general, rather than segueing specifically.

After defining our approach to transitioning, we will consider which qualities of the surrounding motions we may want to preserve during a transition. We will next examine several previously published techniques for transitioning and explain why we find them unsuitable to our needs. Finally, we will present new techniques we utilize and propose for computing both the geometry and timing of transitions.

Approach: **transitioning –** Given two motions A and B and our desire to switch from performing motion A to performing motion B, we define transitioning as a two-phase process. In the first stage, we choose the boundaries of the transition – that is, the time-point in A's motion curves at which we will cease performing A and begin the transition, and the time-point in B's motion curves at which we wish for the transition to end, leaving the actor fully executing B. In the second stage we calculate a set of motion curves that matches the boundary conditions from the first stage with specified C0, C1, or C2 continuity, as well as the duration of the transition motion curves.

6.1.1 Important Qualities to Consider and Preserve

When we generate motion to "fill the gap" between two consecutive motions, we wish the transition motion to have three properties: it must be physically plausible, it should be stylistically consistent with the motions we are bridging, and it should be believable. These properties are very similar to those we desire of the primitive actions produced by motion models, so it should not be surprising that we will quantify them similarly. Depending on the algorithm we use to generate transitions, the motion may be newly generated rather than transformed from existing motion; however, we will still talk about preserving characteristics of the motions being bridged, since that should be a goal of synthesized motion.

Physical plausibility imposes the same requirements on transitions as for motion models: the motion should not cause any unwanted interpenetration of bodyparts; joint limits should be honored; and limb accelerations should be within the range of the actor's muscles. Of course, we also need to be able to selectively ignore any of these criteria since the animator should always have the final word, but by default, an automatically generated transition should follow them.

Preserving the style of the bridged motions involves maintaining the overall pacing or timing, and also preserving the high frequency details of the motions. These are the same characteristics that we defined in chapter 3.1 as being the key elements for our motion transformation engine to preserve. We now have a new dimension to consider, however, since the style of the just-finishing motion may be different from that of the just-starting motion. If, for example, we are transitioning from a trembling *reach* to a trembling *drop-object*, then it is clear that the transition should include trembling. But if the *reach* is performed in a confident style, then the proper course of action is less clear.

At a higher level, we should also be concerned that the transition conveys the thought behind the change of actions, and represents a believable personality. This is one of the more challenging tasks performed by animators, and it is difficult even to define its main properties. It does, however, rely heavily on the timing and pacing of the transition. Obviously, without actually knowing what the actor is thinking, an automatic system cannot hope to reliably produce motion that conveys that thought. We *can* ensure that we provide the tools and hooks for adjusting the timing and pacing, so that an animator or a system that does model actors' minds (such as the *Hap* system of Loyall [Loyall 1997]) can customize the transition appropriately.

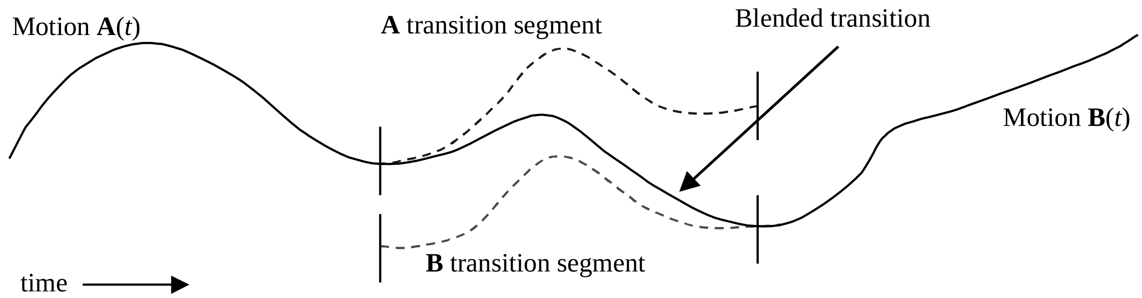


Figure 6-1: Blended Transitions. (Recreated from [Rose 1998]) Motion A's main action enters from the left, and its transition segment is shown with the upper dashed line. Motion B's action begins with its transition segment, and continues to the right. Blending the two transition segments according to a sigmoid produces the smooth continuation of A into B that is shown as a solid line.

6.1.2 Existing Approaches and Problems

In this section we describe the transition generation algorithms in use prior to our work, and explain why we found them unsatisfactory. We will see that in some cases the algorithms suffer from fundamental limitations, while in others we could mitigate or eliminate the problems by applying more structure to motions, as we do with motion models.

6.1.2.1 Blending

A simple and widely used method of transitioning is the "overlap and blend" technique, described by Perlin [Perlin 1995] in the context of textural motion synthesis, and later by Rose [Rose 1998] for use in motion transformation. This technique assumes we have already solved the first stage of the transition problem – *i.e.* we have segmented motion A into a main part and a transition part, and similarly for B, a transition part and a main part. We then create the transition by overlapping the two transition intervals and blending them according to a sigmoid or other smoothly increasing function that rises from 0 (indicating the blend consists of 100% motion A) to 1.0 (the blend consists of 100% motion B). If the two transition intervals are not the same duration (which they generally are not), we first time-warp each interval to a common duration, such as the mean of the individual durations. This is depicted Figure 6-1**.** The resulting transition is as smooth as the segments from which it is blended, and it phases out the high frequency detail of motion A just as it phases in the detail of motion B, although in pessimal cases, cancellation of high frequency detail can occur (if, for example, the troughs of one motion curve align with similarly shaped crests of the other).

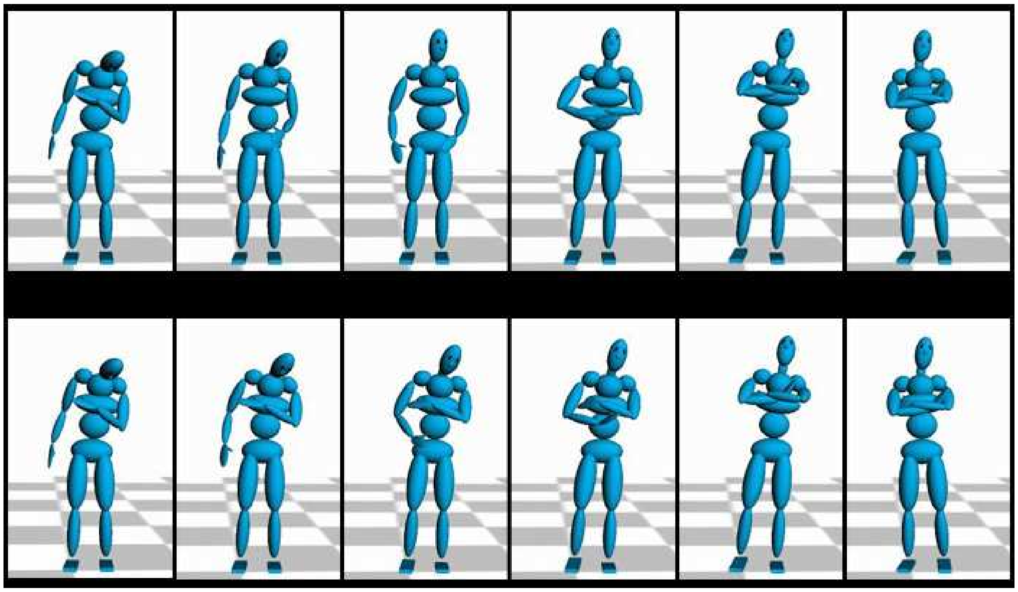


Figure 6-2: Blending Vs. Pose-to-Pose Interpolation. In the top row we see six frames (increasing in time left-to-right, but not consecutive) from the *check-watch* to *arms-akimbo* transition, computed using blending. Note that in the second and third frames the actor lowers his left arm while raising his right. In the bottom row, we see the same transition, computed by interpolating the transition start and end poses using Hermite interpolation. In this sequence the left arm moves directly into a tucked position. The animations of these transitions are available in http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter6.

When the corresponding motion curves in the two transition intervals are roughly the same shape, differing primarily by an offset (as is the case in Figure 6-1**,** and in general when we are transitioning from one repeated action to a similar one – like taking multiple steps in a walk cycle), the resulting blended transition is generally plausible and decisive. To alleviate occasional interpenetration between bodyparts, Perlin [Perlin 1996] added the ability to attach a "buffering action" to any primitive action, which is executed upon any transition into or out of the action. For example, the transition between a "hands behind back" motion and any motion where the hands are in front of the actor may cause the hands to pass through the actor's body. Therefore, we would attach a "hands at sides" buffering action to "hands behind back," so that whenever the actor transitions out of "hands behind back" he first moves his hands to his sides.

However, we encountered two crucial, undocumented problems in using blending for generating automatic transitions. First, if the motion contained in A's transition interval is substantially different from the motion in B's (which is a common case), the resulting transition causes the actor to appear indecisive and awkward, continuing to perform action A for some time after deciding to begin action B. Consider the situation demonstrated in Figure 6-2, where the actor segues from looking at his watch to folding his arms akimbo. The end of the *check-watch* action has the actor returning his hand to his side, while the beginning of the *fold-arms* action raises the arms from a neutral stance to the folded position. The most logical time to begin the transition in action A is at the point where the actor is about to lower his arms, since this is when he is "done" looking at his watch – in an interactive setting, we do not even have any control over the beginning of the transition. However, as we see in Figure 6-2, if we blend *any* of the motion from action A following this time point with motion from B, the actor will first lower his watch-arm partly back to his side before bringing it back up to the fold. This looks indecisive and unnatural – the watch-arm should move directly from its held "check watch" position into the folded position (try this yourself and see what feels most natural). Under these circumstances, the only way we could produce an acceptable blend would be to blend not an interval, but the held "check watch" *pose* from action A, and perform a search in action B to find the time to begin the transition at which the watch-arm is closest to its held position in action A. This procedure is not only expensive, but is not guaranteed to improve matters any, since the result of the pose-matching search may collapse B's transition interval to a single instant.

Second, because blending does not allow us to separate the geometry of the motions from the timing, blends of intervals that move significantly different distances over different times will produce very unnatural motion. For instance, consider the transition between the two motions in Figure 6-3, where we transition from a "bent over, catching one's breath" motion into a short hop. The hopping motion begins from a neutral pose and achieves its natural transition end-point, the crouch from which it will spring upwards, in a quick 11 frames. The "catch breath" motion has a natural transition interval beginning with the instant at which it begins straightening up, lasting for 41 frames until it approximately straightens. The resulting problem is that no matter what we set the final transition duration to be, the speed at which the actor moves will be unnatural at either the beginning or the end of the transition. If we set the duration to something in the range of 15 to 20 frames, then the end of the transition will look fine, but the actor will whip upwards while straightening at the beginning, since the blend consists primarily of the "catch breath" motion at the beginning, and it has been sped up by more than a factor of two. If we instead made the transition last 30+ frames, the beginning would look better, but at the end the actor would appear to freeze and move in slow motion, since it consists mainly of a motion that has been slowed down by a factor of three. In this case, we can improve the transition by pushing back the beginning of the transition in the "catch breath" motion until the actor is nearly straightened out already. However, to effect this solution in general would require the same kind of expensive pose-matching search described in the previous paragraph – and just as it was not guaranteed to work there, it also may fail here.

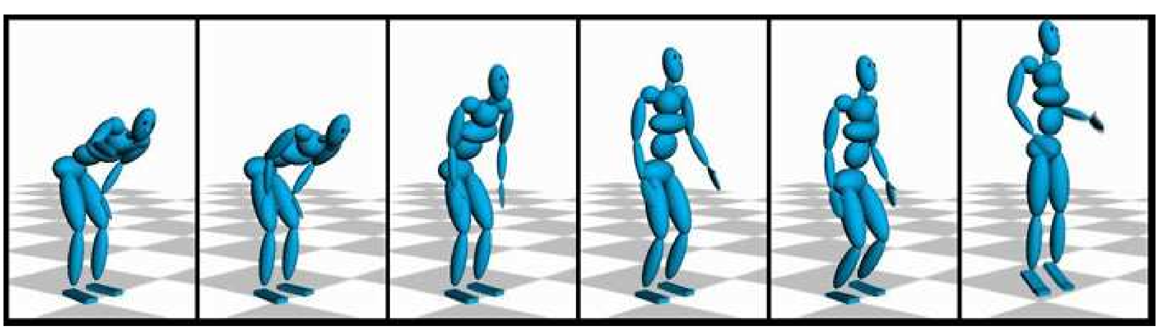


Figure 6-3: Blending from *catch-breath* to a hopping *jump*. The animation of this transition is on this chapter's web page at http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter6.

6.1.2.2 *Pose to Pose Interpolation*

One might complain that it is ludicrous to even consider blending to transition between substantially different motions when there is an even simpler approach that works better in such situations. Many motions, like *check-watch* and *fold-arms*, consist mainly of achieving some pose, then holding it (more or less), ending when the pose is released. If this is the case, the rising action and falling action phases of the motions are relatively unimportant, serving only to bring the actor into the held pose or release him from it. Therefore, why not simply segue between the two actions by interpolating between the held pose of the first action and the held pose of the second, using a spline or other simple parameterized curve per motion curve? The bottom row of Figure 6-2 demonstrates how this technique produces a more natural transition than blending in the *check-watch* to *arms-akimbo* case. We can even use the tangent information in the two motions to match velocity as well as pose at the endpoints of the transition. This technique has been in use for several years, but is rather too simple to have been published.

This technique's performance is predictable. It produces simple, smooth transitions, and will never suffer from the "indecision" that can plague blending. Whenever the surrounding motions are not as smooth or simple (*e.g.* containing high frequencies in their motion curves), the interpolated transitions will not match. The technique also affords us no means of determining the proper duration of the transition. Since it always discards the information and detail contained in the two motions, it will perform worse than blending when the two motions actually are similar to each other.

6.1.2.3 *Spacetime*

Rose [Rose 1996] proposed the use of spacetime optimization to solve the geometric calculations of the transitioning problem. Given a final pose and joint-angle velocities for motion A, and a beginning pose and joint-angle velocities for motion B, plus a desired duration for the transition, Rose applies the entire spacetime constraints apparatus to produce highly physically realistic[25] motion that smoothly bridges the gap between the poses. This is an interesting result in many respects, but it has a number of shortcomings that render it inappropriate for our goals.

First is the limitation imposed by guiding the solution mainly by physics. Without the adaptation of several signal decomposition and analysis tools to spacetime, we would often lose the style of the bridged motion during the transition. For instance, a spacetime transition bridging two trembling motions would not tremble. Also, while it is generally not a problem for simulated humans, character animation in general often demands non-physically realistic motion.

Second, spacetime optimization methods have not been shown to scale well to characters of humanoid complexity. In his work with spacetime, Popović notes that the spacetime optimization frequently fails to converge for humanoids with roughly 50 DOFs [Popović 1999a]. Even if convergence were not an issue, spacetime problems of this magnitude are computationally expensive, and will not be viable for real-time uses for at least another two generations of computer hardware.

6.1.2.4 Finite State Machines

In the "Verbs and Adverbs" paper [Rose 1998], Rose proposes to address all of the problems inherent in the use of blending for transitions by extending Perlin's action buffering to the authoring of rich finite state machines governing the progression of actions during a transition. His key observation is that all of the problems of blending cease to be problems if the two motions being bridged are not that different. Therefore, we can allow the system architect or user to specify, for each pair of actions between which we might want to transition, a sequence of *other* actions that should be performed in between, with blending being used to generate the transitions in the resulting sequence.

Given a sufficient number of intermediate primitive actions, each performed in all the styles in which we are interested, this technique will work well and rapidly. However, it does not scale well, since each time we add a new primitive action to a library of n actions, we may need to not only insert new supporting intermediate actions, but also add $\mathbf{O}(n)$ new nodes to the finite state machine governing transitions.

6.1.3 Computing Transition Timing

One immediately obvious pattern of the existing approaches is their ad hoc treatment of computing transition durations. We believe the principal cause of this condition is that all of the approaches deal with the motions as largely unstructured clips. Because we are transitioning between motion models, however, we possess structure and example data that we can exploit to compute transition durations that reflect the motions involved. In particular, we will utilize the design strategy of enumerating the key-times of a motion roughly according to the "**start, achieve-goal, release-goal, settle, end"** model we described in Chapter 4.1.1.2, and the way of thinking about recording base motions that it implies. Because of this, nearly every motion model can produce a standalone motion that achieves the motion's goal from some start state, performed in the specific style we have selected. In other words, we have a stylistically and physically correct example of how to achieve the action's goal *and* how long it took to do so from that start state, which we may be able to transform to create a transition, or at least extract hints and characteristics that we can apply to generating the transition. We will refer to this example as the *prototype transition* of each motion model.

In this section we will describe how we currently use the information contained in motion models to determine the timing of transitions, and also how extending the scope of information we store in motion models and actors with machine learning techniques could improve the quality of our results even further.

---

[25] The realism is modulo body interpenetration, since we currently possess no spacetime formulation that can detect and

6.1.3.1 Beginning and Ending Transitions

For the case of segueing motions sequentially, our key-time annotations provide us with excellent information from which to choose the boundaries of the transition. When automatically computing transitions, the most logical time to begin the transition[26] is when the A motion has completed its goal, at which point the actor proceeds directly into achieving the B motion's goal. Thus, we select the A motion's **release-goal** as the time at which to begin the transition in motion A, and choose the B motion's **achieve-goal** as the time at which the transition will end in motion B. This automatic selection based on standard motion model key-times generally produces results that correspond to a smooth progression of actor thought, without lapses or interruptions of goal achievements. There may be times, however, when an interruption or delayed response is just what the animator desires; to accommodate these needs, part of the standard complement of motion model parameters are offsets that are applied to the automatically computed transition start and end points.

When we extend our consideration to including transitions in which one motion is *layered* on top of another, the logical time at which to begin the transition is less clear, particularly when the underlying motion is repetitive, *e.g.* layering a *reach* on top of a *walk* across the room. We will discuss strategies for layered transitions in section 6.2.3.1.

6.1.3.2 Transition Durations

Methods for computing the duration of transitions are conspicuously absent in the literature, which we have found puzzling, since the transition duration has proven to be one of the most important factors to believability in our experiments. Rose claims that generally transitions of between .4 and .6 seconds work well [Rose 1996], but this is actually only true for transitions performed at a reasonably brisk pace and in which the bodyparts involved move only a moderate distance. How long a transition should take to complete depends on a number of factors:

- *Style* – Since the style of a motion can affect its pacing, it should also affect the transition; if we're transitioning into a slow moving motion, the transition should be correspondingly slow.
- *Pose distance* – It takes an actor longer to move a farther distance than a shorter one.
- *Torque limits* – Some transitions should be performed as fast as possible, and the "speed limits" are determined by the strength of the actor's muscles.

The primary basis for our computation of a transition's duration is motion B's prototype transition, which we described at the beginning of this section (6.1.3). It gives us a concrete example of how long it should take, in the particular style selected, to transition from the motion's starting pose to the goal (transition end) pose. Our task is how to use that information to answer the question of how long it should take to

---

correct this condition.

transition from the transition's starting pose to the goal pose. We have experimented with several techniques:

1. **Constant time –** Regardless of the pose in which the transition begins, it will last as long as the prototype transition. We have found this to be useful mainly for simple gestures; it has the advantage of being exceptionally fast to compute.
2. **Fast as possible** – Within the actor's muscular speed limits, perform the transition as fast as possible. We will discuss how we measure speed and speed limits below. This measure performs well for transitioning between athletic actions.
3. **Proportionate to pose distance –** What we truly want is a single measure that takes into account style, muscular limits, *and* the distance covered by the transition. Our first attempt at such a measure conjectures that it should take an actor twice as long to move twice the distance as contained in the prototype transition (and half as long to move half as far). We use the pose distance metric $\hat{\mathrm{E}}_{dm}$ defined in chapter 4.4 to measure the pose distance traversed by the prototype transition and between the starting and ending poses of the new transition. While this linear relationship between distance and elapsed time almost always holds true on a macro scale (*e.g.* it takes twice as long to walk two flat meters as it takes to walk one), we have found it less robust in predicting pose changes not involving locomotion. It performs best when the style dictates a slow to moderate pace for the transition; it often computes incorrect durations for transitions that should be fast moving.

In current practice, none of the above methods works satisfactorily for all motions or all styles. We therefore have made available to the motion model and style designers all three methods. We have found that for a particular style, and sometimes, for all styles of a motion model, one or another of the above methods produces good results most of the time. Each motion model specifies which method to use by default, and styles can contain an annotation that allows them to override the default; determining which method requires a little experimentation – we can easily provide a test suite to evaluate each method. When the automatically computed duration is unacceptable, the animator can use one of the standard motion model parameters to manually set the transition duration.

6.1.3.3 Learning Transition Durations

Even though it is easy to adjust a poor transition duration, we would still like to have a single, reliable method for automatically computing durations – particularly for real-time motion synthesis, since it affords us no opportunity to manually tweak our results. The inadequacy of our existing methods stems from their over-simplified assumptions about the relationship between pose distance and duration – that it is either constant or linear. Although we are unable to verify it experimentally in time for the thesis, we propose the following hierarchical scheme for computing durations.

---

[26] In the absence of any further information about the actor's state of mind

Each character model class (such as humanoid) defines a scalar valued function of the ratios of current and prototype pose distances,

$$\mathrm{S}_{pd}\left(\frac{\hat{\mathrm{E}}_{dm\ A\to B}}{\hat{\mathrm{E}}_{dm\ proto}}\right)$$

such that the transition duration for the A to B transition is

$$dur_{A\to B} = \mathrm{S}_{pd}\ dur_{proto}$$

i.e. a scaling of the prototype's duration. Each specific actor (like George or Sally) can provide its own S'_{pd} that overrides the default S_{pd}, and each style can contain an S''_{pd} that overrides both of the other measures[27]. Our hope is that since S_{pd} operates on ratios rather than absolute distances, and produces scale factors rather than absolute durations, it's application will be fairly broad and we will not often need to define lower nodes on the function hierarchy.

How do we compute S_{pd}? Through one of two methods, we learn the function. If we possess sufficient motion data wherein the actor performs the same motion but from different starting poses, we can acquire measurements automatically. Otherwise, we can develop a test suite of different starting poses for a transition into a particular motion, compute the transition geometry automatically, and ask the master animator to supply an appropriate duration for each case. Once we have the data, we create a function through any of a number of regression algorithms, such as polynomial function fitting or neural nets. Data input for a more specific function (S'_{pd} or S''_{pd}) can be percolated upwards to more general functions to influence them. However, such training sessions should occur *before* creating any serious animation, since changes to the functions could alter the timing of existing animations.

6.1.3.4 Per Bodypart-Group Timing

In order to simplify the presentation, we have omitted from the entire discussion of transitions thus far one crucial, but complicating aspect. All of the calculations we have so far described – pose distance, muscular torque limits, and transition durations – must be performed separately on each bodypart-group. There are three main reasons why this is necessary:

1. It does not make sense to talk about the torque or speed limit of an entire actor, and while we could deal with individual joints (incurring more overhead), it is reasonable to measure limits on sets of coordinated joints, as we will see shortly.

---

[27] One may notice that we have not allowed motion models to define their own scaling functions. We reason that specializing by actor is more useful than by general action class, and allowing specialization for both would cause conflicts. We do allow specialization by style, reasoning that specific styles would be developed for each actor.

2. Most limbed creatures move in a coordinated fashion, with, for example, arms and legs moving simultaneously. If we tried to use pose distance of the entire body as a metric for determining how long it should take to move that distance, we are ignoring coordination, with potentially grievous consequences. For instance, suppose our prototype transition involves mostly arm movement, which displaces *x* mass units. Now we must compute a transition in which the same *x* movement occurs in the arms, but one of the legs also moves by *x*. This transition should take about the same time as the prototype, since the arms and legs move simultaneously in coordination; however, the full-body pose distance for this transition is 2*x*, which erroneously leads our algorithms to believe the actor has twice as far to move, so should take a longer time to do it.

3. Many times (particularly when layering is involved) a transition will not involve the actor's entire body, and since, as we have already explained in chapter 4 that the bodypart-group is our atomic unit of motion combination within a clip, we must be able to compute transitions over arbitrary subsets of an actor's bodypart-groups.

We will now describe how this per bodypart consideration affects each of the three computations listed above.

Pose Distance

Instead of asking how much mass the actor displaces in moving from pose A to pose B, we are actually asking the subtly different set of questions: how much mass did the (*e.g.*) left arm move due to the action of only its own muscles? This question is different because if the actor bends at his hips, his left arm moves along with the torso, but the induced displacement is included only in the displacement computation for the torso, *not* the left arm. This difference, which necessitates some changes to the recursive descent algorithm we use to compute $\hat{E}_{dm}$, makes sense because it factors out the structure of the transformation hierarchy on the computation; also, if we sum the displacement measurements of all the individual bodypart-groups, we get the same value as computing the displacement over the entire body.

"Speed" limits

If we were to use common measures of muscular limits based on individual joints, we would need to convert our results from the joint domain to the bodypart-group domain, since it is there that we combine motions. Trying to define a "composite force/torque" limit for a bodypart-group such as an arm is tricky, however, because one would need to account for the differences in scaling between the various joints. We achieve a similar effect by simply measuring and limiting the amount of its own mass the arm can move per unit time. This involves only the same pose distance computation described in the preceding paragraph, and gives us what we call "mass speed," leading to enforcement of "mass speed limits". To define the actual mass speed limits, we feed snippets of fast moving motion to an analyzer that computes, for each bodypart-group, the ratio of pose distance moved to elapsed time. This need be done only once per character model class, although individual actors can supply their own limits.

Transition Durations

When we use any of the duration computation methods that involve pose distance, we actually calculate a time for each bodypart-group independently. This results in a different duration for each bodypart-group participating in the transition; since we desire a single duration for the entire transition, we must somehow reduce the individual durations to one. We have had success using two techniques:

- **Fastest Legal** – Use the shortest computed individual duration, but adjust it upwards if necessary if any mass speed limits would be violated.
- **Mean** – Use the mean of the individual durations.

However, computing per bodypart-group durations using ratios of pose distances, as we have described in the last two sections, fails when the prototype transition does not contain significant information about all of the bodyparts. In this case, information equals movement: if, say, the left leg moves hardly at all in a prototype transition, then any transition that contains moderate to significant left leg movement will produce a very large input for the S_{pd} (or other) function, since the ratio of current to original mass displacement will be very high. For the linear pose distance computation described in section 6.1.3.2, this will always produce an erroneously high transition duration for the left leg – the left leg did not truly participate in the prototype transition, so we should not expect to be able to base any computations on it.

If we are instead using a learned S_{pd} function, we have several options. If the function was trained on motions in which the left leg actually contributed to the prototype motions, then the computation will fail just as the simple linear scaling does. If we train a S''_{pd} for each style, then we have already added in enough extra information for the computation to succeed (*viz.* we have considered numerous starting poses already). However, we claimed that one of the benefits of formulating S_{pd} in terms of ratios was that we should not need to define many S''_{pd}s. We can preserve this claim by noting that in these cases, the bodypart-group does not really contribute to the motion, so we can appeal to more generic information sources to compute a feasible duration. Therefore, we need only supply a single extra set of functions R_{pd}, which operate directly on pose distance, and produce durations of "normal" movements for each bodypart-group.

It might seem that a simpler solution would be to compute, rather than the mean of the individual bodypart-group durations, a weighted sum, with greater weighting given to those groups that move more in the prototype transition. This would be mistaken, however, since then any movement that occurs in the current transition among the lightly-weighted groups would be essentially disregarded, thus potentially leading to abnormally fast transition durations.

6.1.3.5 *Effects of Single Entrance/Exit Times?*

Most human motions, at least, involve coordination of multiple limbs and bodyparts. Even so, it is rare that different limbs begin and end particular coordinated motions at *exactly* the same time. One might then wonder whether our decision to characterize transitions as completely starting at a single instant and ending at a single instant would produce artifacts. In practice we have not noticed this to be the case, and we reason that is so because as long as we maintain both geometric and stylistic continuity across the transition boundaries, each limb proceeds at a pace dictated by the geometry and its velocity at the beginning of the transition. Thus the only condition under which all limbs will perceptibly begin in lock step is when the transition begins from a perfectly still pose. Furthermore, if the animator desires a greater degree of independence in the motion of the limbs, he can either use layering (in which case the transition will only involve a subset of the actor's limbs), or tweak the final results with the warping facilities we will describe in chapter 7.

6.1.4 Some New Approaches for Transition Geometry

The transition duration is crucial to achieving an effective transition, but just as important is the geometry of the transition, *i.e.* the shape of the actual motion curves that will drive the actor's movement throughout the transition. Given our dissatisfaction with the published methods, we tried several new approaches, trying to preserve as many of the quantities listed in section 6.1.1 as possible.

One of the primary goals is to preserve the high frequency details of a particular style throughout a transition. In section 6.1.1 we mentioned the dilemma of deciding which of motion A's or B's style to use, if the styles are different. We noted that blending combines them, by default, gradually easing out A's details while easing in B's. As can be seen from our definition and use of a "prototype transition" in our discussion of transition durations, we have taken a different approach. Reasoning that the beginning of a transition represents a conscious shift of thought into anticipating the upcoming action, we endeavor to incorporate the style of the B motion into the transition.

6.1.4.1 *Warp to Boundary Conditions*

The simplest method of computing a transition geometry that preserves the details of the B motion's style is to space-warp the prototype transition: we use the motion curves of the prototype transition (appropriately time-warped according to the transition duration), inserting a space-warp key at the beginning to match the pose in which motion A terminates, and a zero-displacement key at the end to match the transition-end pose. This method works well, subject to the same limitations of space-warping as a general deformation tool. Namely, it produces artifact-free results when the A-termination-pose is not too different from the starting pose of the prototype transition.

Another potential problem with this technique is that it guarantees only C0 continuity at the beginning boundary of the transition. We can increase the smoothness of this boundary either by overlapping the A motion and blending it with the warped motion for a short time beyond the boundary, or by using the tangent information at the boundary in the A motion to place a third space-warp key shortly after the first that will more nearly match the transition to the A motion. The former approach can suffer from the same problems we described for blending as a general transition mechanism, and the latter can introduce high frequencies into the transition, due to the close spacing of the keys at the beginning. In those cases where space-warping the prototype transition is otherwise applicable, we have not noticed any discontinuities at the beginning of the transition, even without application of either of the measures we just described. This is because of the general timing structure we apply to motions. When a motion fits into the attack/sustain/decay/release model, we generally begin a transition out of the motion at the end of the sustain phase, at which time the actor is usually holding a pose, about to begin releasing it – therefore, we notice no sudden momentum shifts since the actor begins a new motion essentially from rest. If the A motion causes the actor to be moving rapidly at the transition boundary, we would definitely notice the discontinuity.

6.1.4.2 Generate New Motion in Consistent Style

Space-warping is simple, fast, and preserves the B motion's style quite well, but even if continuity is not an issue, it is still only applicable under certain circumstances. The pose to pose interpolation scheme of section 6.1.2.2 is a more general scheme, which we can vary in several ways: we can interpolate the motion curves uncoupled from each other (*i.e.* each one separately), or coupled through IK or spacetime, and we can try to incorporate the high frequencies of the B motion or not.

In our current prototype system, we have employed the simplest of these: uncoupled interpolation via single Hermite cubic segment (or the corresponding spherical Beziers on S^3) per motion curve, interpolating the pose and velocity at the transition endpoints. In our scheme, we do incorporate the tangent values from motion B at the end of the transition, and utilize the timing information we gleaned from the prototype transition. As mentioned above, this technique produces believable results when the prototype transition contains little high frequency content. Hermite interpolation would fail, of course, in the same example we pointed out as a limitation of the spacetime technique, transitioning from one trembling motion to another, since the trembling would completely drop out during the transition.

We have formulated a strategy, as yet untried, for synthesizing new motion for a transition that incorporates the high frequency detail of the prototype transition. Simple interpolation provides a good basis for the low frequency components of a transition, over a very wide range of starting poses. We propose to take this solution and graft on high frequencies extracted from the prototype transformation using the Laplacian transform, whose application to decomposing motion curves into frequency bands was described by Bruderlin [Bruderlin 1995]. The result should contain most of the detail preserved by warping, but contain none of the low frequency artifacts warping can often introduce. There are some problems yet to solve, such as what to do when the duration of the prototype transition does not match that of the computed transition: possibilities include compressing/dilating the prototype (thus shifting the frequencies), select a time-window from which to extract frequencies (if the current transition is shorter than the prototype), or try to adapt a statistical texture synthesizer such as that of De Bonet [De Bonet 1997] to producing new frequency curves similar to the originals (if the current transition is longer than the prototype).

6.1.4.3 Current, Per Bodypart-Group Scheme

Even if we achieved our goal of providing motion model designers with a single automatic transition computation algorithm that produced good results 100% of the time, the "good" result will not always be what the animator desires, so we provide to the motion model designer, the style designer, and ultimately, the animator, a variety of choices. The motion model designer will specify the most broadly applicable method for its action, while individual styles and the animator can override these defaults. We emphasize again that we can only provide these options because of the structure motion models provide us. The current set of techniques is:

- **Blend –** Use traditional blending for all participating bodypart-groups.
- **Warp –** Use space-warping for all participating bodypart-groups.
- **Hermite –** Use Hermite interpolation for all participating bodypart-groups.
- **Best –** For each participating bodypart-group considered individually, if the deformation at the beginning of the transformation that would be induced by a space-warp is less than 10% of the total mass of the bodypart-group, then use space-warping for that group. Otherwise, use Hermite interpolation. The deformation we measure is the pose-distance (section 4.4) between the transition start pose and the start pose of the prototype transition.
- **Warp named –** If the prototype transition motion of a few bodypart-groups contains high frequency content that is crucial to the believability of the transition, but the other bodypart-groups are of less importance, we can specify that space-warping be used for only those groups, while Hermite interpolation is used for the others.

Except for testing purposes, we use exclusively the last two methods in our motion models. If the high frequency replacement method works as planned, we can eliminate all methods but 'Blend' from the above list.

6.1.5 Clip-Transition

Our main unit of animation is the clip, and a transition (or resumption –see below) should be thought of as a "glue" clip that joins two other clips together. To formalize this relationship and simplify motion model design, we supply a sub-class of clip called clip-transition. In addition to the normal clip functionality, the clip-transition actually creates appropriate transition motion using the rules and concepts presented in this chapter. When a motion model needs to create a transition for itself, it need only pass to a clip-transition the two clips involved, the relevant entrance and exit times, and its choice of methods for computing the transition duration (from section 6.1.3) and the transition geometry (from section 6.1.4). The clip-transition will then create the motion and inform the motion model of the transition duration; the clip-transition can then be abutted or overlapped with the motions between which we are transitioning to create a seamless animation.

Our decision to abstract this functionality into a class separate from the motion model is motivated by our guiding principal of minimizing the exchange of class-specific knowledge between motion models. Because the transition is subsequently used only as a clip, a motion model designer is free to supplement or ignore the standard clip-transition. For instance, a *run* motion model can check to see whether its preceding motion was a *walk*, producing a blended, in-motion transition if so, and starting up from rest if not.

6.1.6 Avoiding Self-Intersection

Avoiding actor self-intersection (not to mention intersection with the environment) in full generality and with high fidelity is computationally expensive and difficult to do well. It is well beyond the scope of this thesis. We mention this fact here because self-intersection is more of a problem for transitions than for motion models, which contain more specific knowledge (in the form of invariants and multiple examples of legal motions) on how to avoid it. We will describe the two basic approaches we have considered, and then the current extent of our provisions.

The more attractive alternative, from a computational standpoint, builds on Perlin's action buffering. Similarly to Rose's extensions, we would build a large catalog of beginning and ending transition poses between which simple interpolation would produce self-intersection. Unlike Rose, the "states" between the beginning and ending will consist simply of intermediate poses through which either a space-warp or low-frequency Hermite interpolation must pass. This method is computationally attractive because it does not require any collision checks, which are expensive, nor iterative refinement of the motion. However, it has potential fidelity problems, as well as complexity issues.

Perlin bases his scheme on recognizing problematic partial poses: for example, "hands behind back" transitioning to "hands out front", which would cause the hands to pass through the body without buffering in a "hands at sides" pose. However, we cannot ignore what the rest of the body is doing in concert. For instance, the partial pose transition "hands at sides" to "hands out front" is a straightforward motion, free of complications – if the actor is standing up straight. If, instead, the actor is squatting down on his haunches, his hands will almost certainly pass through his legs. This makes the problem combinatorial in nature, and increases both the dimensionality and the width of the problem domain. To determine the best set of buffering poses for a particular bodypart-group (*i.e.* partial pose) we must decide whether to consider its motion as part of a larger, coordinated motion (consider both arms together, or individually?), and then factor in the motion of all the other bodypart-groups, even those not participating in the transition. The motion model into which we are transitioning can generally make the former decision. For the latter, we can organize the many partial poses into "pose bounding hierarchies," using our pose distance metric to descend the hierarchies and find the closest matching poses in our database efficiently.

The second problem is that the fidelity of this technique is only medium. Its purpose is to prevent major errors in moving from one broad class of poses to another. But it has no means of detecting or correcting smaller scale errors. For instance, after we add in high frequency details to a basic arm trajectory produced by the action buffering, it may be that the elbow intersects the torso a slight but noticeable amount. Since the technique has no way of reviewing its results, we cannot fix this problem (within the technique).

Our other alternative (also speculative), which can also be used as a secondary patch on top of action buffering, is to preview the transition motion, scanning for self-intersections using collision detection, and repair problems when encountered. Scanning for collisions is straightforward, if expensive. Repairing the problem is more difficult. Eliminating the instantaneous interpenetration is fairly simple, provided our collision detection algorithm provides us with a point on each of the two participating bodyparts that is inside the other bodypart: we simply apply a force on each point in the surface normal direction of the other point until the bodyparts separate. We must, however, also engage a collision avoidance algorithm over the actor's entire body as we apply the separating forces, so that the offending bodyparts do not inadvertently penetrate *other* bodyparts while eliminating one intersection. Because it incorporates no real motion knowledge, this technique can easily produce feasible, but ugly motions. It is truly intended only for repairing small penetration violations; the action buffering or higher level control algorithm must be responsible for coordination of mostly collision-free path planning.

Using this spot-elimination ability to create plausible transition motion is a much more difficult part of the problem. Simply running the above algorithm at every frame of the transition would not only remove existing high frequencies from the motion, but may also result in motion where a limb heads for a collision course with part of the body, then skims along the surface when the collision is imminent. Somehow, the information about where the major intersections occur must be communicated to a motion planner that can update the motion on a broad scale. This is an entire field of research. Instead, since the action buffering technique functions much like a gross-scale planner, we would try using the spot-elimination functionality to repair only the small-scale intersections that may still exist in the action buffered result.

In our prototype system, we have implemented only a very limited form of action buffering. We have not found this to be too burdensome during the animation process because of the ease with which tweaks can be applied. When we notice a self-intersection problem, we can generally correct it by interactively eliminating the intersection using IK, thus creating a tweak that adjusts the surrounding animation accordingly (tweaks will be explained in detail in the next chapter). For real-time systems in the near future, basic action buffering is probably the best we can hope to do.

6.1.7 Examples

The difference in results between the various transition generation algorithms presented here involve differences in timing and at times subtle differences in trajectories. These cannot be communicated well in a static medium, so we must refer the reader to our accompanying website for illustrative examples beyond the example in section 6.1.2.1, which did lend itself to the static page. The relevant animations are at **http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter6,** and include: the *check-watch* to *fold-arms* transition, using both blending and pose to pose interpolation; the *catch-breath* into *hop* transition using blending; and several examples demonstrating the circumstances under which space-warping is effective for computing transition geometry**.**

6.2 Layering Motions- Performing Multiple Actions Simultaneously

Most creatures possess the ability to perform multiple actions at the same time - if not in reflection of multiple trains of thought, then at least in coordinated pursuit of a single goal. For instance, while I am walking towards my car, I press a button on my keychain that unlocks the doors, and while a monkey is swinging through the trees it is also scouting for danger and its next handhold. Without this ability to "layer" motions on top of each other temporally, our actors can follow only a single, linear course of action, which will lead to disjointedness when, for example, an actor must stop walking to wave at someone, then resume walking. In this section we will explore the necessary components of layering, and describe our approach to implementing it. We include several examples of layered motions in the online animations at: **http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter6**

6.2.1 When Motions Can be Layered

First let us define layering and our goals in layering. One motion (the *layered motion*) is layered on top of another (the *underlying motion*) if their temporal extents overlap in time, and each motion pursues its goals simultaneously. An underlying motion can have many motions layered directly on top of it, and each layered motion can serve as an underlying motion to other motions layered on top of it, *ad infinitum*. Thus we are defining a hierarchy or "stack" of motions; each level in the stack can consist of a single motion or a *stream* of multiple motions connected temporally by segues. The need for the hierarchy in layering that we have just defined arises from the necessity of establishing an execution order for the layered motion models. The consequences of the this hierarchical relationship of layered motions is twofold:

- **Feasibility of layering**. A motion can only be layered onto the existing stack if it can acquire the resources it needs from the actor in order to actually achieve its goal during the time in which it is scheduled to execute. "Resources" are basically the use (sole or shared) of actor bodyparts that are integral to the motion, and "acquire" refers to the fact that some bodyparts may already be in use by other motions further down in the stack (and thus, by definition, of higher precedence), who may be unwilling to share or relinquish the bodyparts because they are necessary resources to the pursuit of *their* goals. The animator or higher level control system can structure motions in the hierarchy in any manner they please, keeping in mind that a motion model can generally only reference motion models below itself in the hierarchy[28]. We have found it useful to structure the hierarchy according to importance of the motions: the "primary" motion (often some form of locomotion) resides at the base of the hierarchy, while secondary motions are layered on top of it or other secondary motions.
- **Temporal relationships**. We will discuss below all of the ways in which we might want to decide when a layered motion should begin executing, but one of the simplest ways is defined by the hierarchy: we can specify the entrance time of the layered motion as an offset from either the entrance time or exit time of the underlying motion.

Even more so than with our techniques for computing segues, layering depends on the structure provided by motion models. This is due to the necessity of defining, tracking, and acquiring resources, and the need to specify in detail how each motion best layers on other motions.

6.2.2 When Motions Should be Layered

An action must be layered on top of another action (and therefore *should* be layered) whenever it must begin execution before the other has finished. Due to the way in which we have structured motions, this may not always be possible or make sense (as described in the previous section); however, most combinations that do make sense are feasible (*eg.* jumping while walking is not allowed because both motions require exclusive use of the legs, but this combination does not truy make sense, either).

6.2.3 *How* Motions Can be Layered

In previous work [Perlin 1995] [Rose 1996], layering has always been executed such that the layered action, after an appropriate blend-transition, assumes total control of the bodyparts in which it is interested. When we refer to bodyparts here, we mean the motion curves driving the bodyparts. We have explored three different means of combining underlying and layered motion curves, which can be utilized independently for each bodypart-group participating in the layering:

1. **Replace**. Just as in previous approaches, the replace combination method transitions from the underlying motion curves to those of the layered motion. This method is rather limited, since such "all or none" combinations may make it difficult to maintain the style of the underlying motion while performing the layered one. For instance, let us suppose we wanted to layer a *reach* motion on top of a *stooped-walk* motion. The *reach* involves the torso to some degree, but was most likely performed with the actor beginning and ending in an upright position. Therefore, when it is layered on the *walk*, the actor will come out of his slouch to perform the *reach*, only settling back into a stooped-over position after concluding the *reach*.

2. **Blend**. Blending allows us to drive the bodyparts with contributions from both the underlying and layered motion curves. This allows us to generate motions that are part layered, part underlying, and thus can solve the problem we noted with the replace method. However, we would not be able to apply this method in general without having invariants at our disposal, because mixing any amount of the underlying motion in with the layered motion will cause the task-level constraints of the layered motion to be violated (*e.g.* the actor will reach a different target than if the *reach* were layered using the replace method). Invariants repair this damage while maximally preserving the style of the (blended) motion curves. Blending weights are specific to each motion model, bodypart-group, and potentially motion model parameters. We will discuss these weights and how they are combined in a layering hierarchy in section 6.2.3.2.

3. **Differential**. When the exact position of a bodypart is not critical to the layered motion, but the high-frequency content of its motion is important, we can add just its *differential* motion onto the underlying motion. We do this by specifying a timepoint in the layered motion at which the system samples the motion curves to produce a reference pose. Then, whenever the layer hierarchy is sampled, we generate poses for each bodypart by adding onto the underlying motion a weighting of the difference between the layered motion curves at the sample time and the reference pose. In other words, for each quaternion motion curve:

$$\mathbf{final} = \frac{\mathbf{layered}_{curr} \circ \mathbf{layered}^{-1}_{reference}}{weight} \circ \mathbf{underlying}$$

   where '$\circ$' denotes quaternion multiplication, and division of a quaternion by a scalar means to divide the magnitude of the quaternion's rotation by the scalar.

[28] The potential exception is in the parameter optimization scheme we propose in section 9.3.5, in which information

We often use this method for the root bodypart, since we generally do not want the overall position of the hips in the layered motion to affect the underlying motion.

6.2.3.1 Computing Transition Boundaries

Automatically determining when a segue transition should occur is fairly easy - when the motion from which we are segueing has completed its action. The "best" time at which to begin a layered motion can depend not just on the structure of the underlying motion, but also on its detailed geometry and the goals of both underlying and layered motions. We can motivate the different classes of transition criteria we have identified with some illustrative examples.

1. We wish to animate an actor trying to grab something high overhead by jumping and reaching for it. We can express this as a *reach* layered on top of a *jump*. Since the actor is trying to reach as high as possible, he should achieve the goal of the *reach* just as he reaches the apex of his *jump*. Therefore we wish the layered *reach*'s transition to be computed such that the aforementioned alignment of goals occurs.

2. We script an actor to walk across the room along a path, passing by a table (this is accomplished with a single application of a *walk* motion model). We then decide the actor should pick up a newspaper lying on the table as he walks past the table, so we layer a *reach* on top of the walk. The animation system should select a time to begin the transition into the *reach* such that the newspaper will actually be within reach when the actor reaches for it. Therefore, the system must determine the point in time at which the *walk* brings the actor closest to the table, and compute the transition start time such that the goal of the *reach* occurs simultaneously. In other words, the system must optimize a function of a derived parameter (here, the actor's center of mass) over the underlying motion; the function being optimized is the distance between the actor's center of mass (as a function of time) and the target of the *reach*.

3. In the same example of the actor walking past a table and picking up a newspaper, using the direct result of the "closest to target" optimization above might produce an awkward transition: let us assume the actor is reaching with his right hand, and that the newspaper will be somewhat in front of him when he reaches for it. As he walks towards it, his arms will most probably be swinging back and forth to some degree. Since the reach will involve forward-swinging motion of his right arm, it would look awkward if he were to begin the reach while his arm is in mid-swing backwards, since a jerk would ensue. Of course the "closest to target" optimization knows nothing about arm swings, and so might allow such a jerky transition to occur. Minimizing the transition jerk requires us to find the time to begin the transition at which the velocity of the arm in the underlying motion most

---

can propagate up and down a hierarchy.

nearly matches that of the arm in the layered motion[29]. In fact, just as the actor does not need to be *exactly* at his closest approach to the newspaper when the grab occurs, so too does the jerk not need to be exactly minimized - beginning the transition anytime during a forward arm swing will be acceptable. Therefore we could compute the transition start time by optimizing a two-term function of time: the first term is a exponential-bowl shaped function of distance-to-target that does not contribute much when the actor is within an arm's reach of the target, and the second term measures the difference in velocity between the arm's underlying and layered motions. In practice, it may improve performance to precondition this computation by using the pure "closest to target" solution as a starting point.

These considerations led us to identify the following three means of specifying/computing the time at which a layered motion should begin execution:

1. **Offset**. The simplest method allows us to specify the starting time as an offset to the starting time of the underlying motion, in units of animation time or in the canonical units of the underlying motion. The layered motion is thus anchored to the underlying motion, its start time moving in step with any changes to the underlying motion. A simple variation on this direct specification is to use absolute animation time rather than relative to the underlying. The only limitation on this method is that *some* underlying motion must exist (but not necessarily that of the specific motion model on which the layered motion is layered) at the time when the layered motion is to begin its execution.
2. **Goal Positioning**. Rather than specifying when the layered motion should begin, we can specify when we want it to achieve its goal, again specified as a relative offset to the underlying motion. This allows us to handle example one above and many other situations.
3. **Optimizing on Underlying Motion**. The final method allows us to specify an objective function of time that can contain terms involving bodypart positions and velocities in the underlying motion. The solution resulting from minimizing the objective function is then either the start time or the goal time of the layered motion.

The more involved, optimization-based calculations are useful as a minor timesaver in an offline animation system - in most cases the animator would want to fine-tune the transition start time anyway. Also, in an interactive system where motion models are being used to generate motion for an avatar in direct response to input from a user, the beginnings of transitions are determined directly by the timing of the user's input. The situation under which these more powerful and costly methods are most justified is when mo-

[29] In real life, the actor would probably adjust his stride in anticipation of the pickup. We may be able to incorporate such reasoning using our proposed technique for inter-motion model parameter optimization, which we discuss in section 9.3.5.

tion models are being used as a back-end to AI or script driven avatars, where it is extremely useful to be able to specify plans in which timing is expressed relative to cooperating goals.

Accordingly, we have only implemented the first method in our prototype system. It allowed us to test the effectiveness of layering efficiently and rapidly. Goal Positioning, while simple in itself, causes complications to our current framework for ordering motion models for execution. Therefore we delayed its implementation as well as that of the optimization method. We will discuss this limitation, its causes, and remedies, in the next chapter.

6.2.3.2 The Granularity and Legality of Sharing

For layering to be useful, we must be able to select, replace, or mix parts of full-body motions. As we have already mentioned, we use the bodypart-group as the granularity for these operations. This has an impact on all of the aspects of layering that we have discussed up to this point. We will now discuss how to specify and execute the sharing of motion resources.

Another item of knowledge built into each motion model is a per-bodypart-group annotation that specifies how important the motion of that group is to the motion model's action, and to what degree the motion model is willing to share it in a layering situation. The annotation becomes attached to the motion produced by each motion model, and is thus defined and available in all potentially underlying motions. The annotation has both a qualitative and quantitative component. The quantitative part, to which we will refer as *importance*, is a number from zero to one hundred. The qualitative part, which determines how the importance is interpreted and acted upon, is one of the following tags:

- **Control-Required**: The motion model requires exclusive use of the bodypart-group in order to be able to work properly. If the importance of the underlying motion for this group is greater than that of this motion model, the motion model cannot be layered.
- **Control-Optional**: The motion model is willing to provide all the motion for this group, but does not require it. If its importance is greater than that of the underlying motion, it will provide the motion, otherwise it will do nothing.
- **Blend-Required**: The motion model is willing to blend its motion for this group with that of other motion models. It *must* be able to blend its motion with the underlying motion (and so will fail to execute if the underlying motion's annotation for this group is **Control-Required**), and is willing to blend (but not relinquish complete control) with any motions layered on top of itself. The relative importances of the two motions determine the blend ratio.
- **Blend-Optional**: Like the previous tag, but the group is not so critical to the action that the motion model will fail to execute if the underlying motion is unwilling to share.

- **Add-Required & Add-Optional**: The meanings of these tags are analogous to the **blend** tags, except instead of blending, they specify the differential combination operator.

We use the first four tags extensively in most motion models. For example, consider a *point* motion model in which the actor points at something with his right hand. The legs are fairly unimportant, so they earn *control-optional* tags. The hips and torso both receive *blend-optional* tags since they can contribute to the motion, but do not need to. The right arm gets *control-required*, while the left arm and head get *control-optional*. The *add-* tags are special purpose, and consequently get less frequent usage.

When a layered motion model executes, it must determine whether it can access all of the bodypart-groups that its annotations specify it requires. In order to do so, it must query *all* of the motions underlying it during any portion of its would-be extent in the animation – if any of them require exclusive use of a bodypart-group that the motion model needs, the motion model cannot execute at that time.

Assuming the motion model is able to utilize all the bodypart-groups it needs, it must also check for invariant access. The very nature of invariants precludes their being simultaneously shared – a bodypart cannot be in two places at once, nor can it have two different orientations at the same time. Therefore, we enforce the rule that at any time in the animation, only one motion model can utilize each invariant. This means that if any motion underlying the layered motion model is using an invariant the motion model requires in order to execute properly, the motion model cannot execute at all.

The class motion model provides a method `Gather_Layered_Resources` that performs both these checks, given the motion model's per-bodypart-group annotations and a list of all the motions underlying the motion model.

6.2.3.3 Resumptions

In a linear sequence of motions, there need be only one transition associated with each motion - the transition from the previous motion into the (current) motion. However, when we layer motions, the layered motion and the underlying motion will not, in general, end at the same time. This means that at some time one or the other motion will need to provide data for all of the motion curves and invariants previously generated by the combination of the two motions. This in turn necessitates a transition from what the terminating motion was doing to what the remaining motion will do. As we noted at the beginning of this section, a level in the "motion stack" need not consist of a single layered motion, but may be a segued stream of motions. In this case, we are concerned not with the ending of the initial layered motion, but with the termination of the last motion in the stream rooted at the layered motion.

This switching of control is, of course, a transition, which we distinguish from normal transitions as a *re-sumption*, since it involves transferring control back to a motion already in the motion stack rather than to a newly rising one. Resumptions also differ from normal transitions in how they compute their durations and handle invariants. Throughout this section we will refer to the example in Figure 6-4, in which the actor is re-orienting himself, then locating a target towards which he is going to jump; since pivoting does not require his full attention, he begins looking around before completing the pivot. However, he finishes pivoting before he finishes looking around, so the jump transitions from the peer, after the pivot has passed full control to the peer via a resumption.

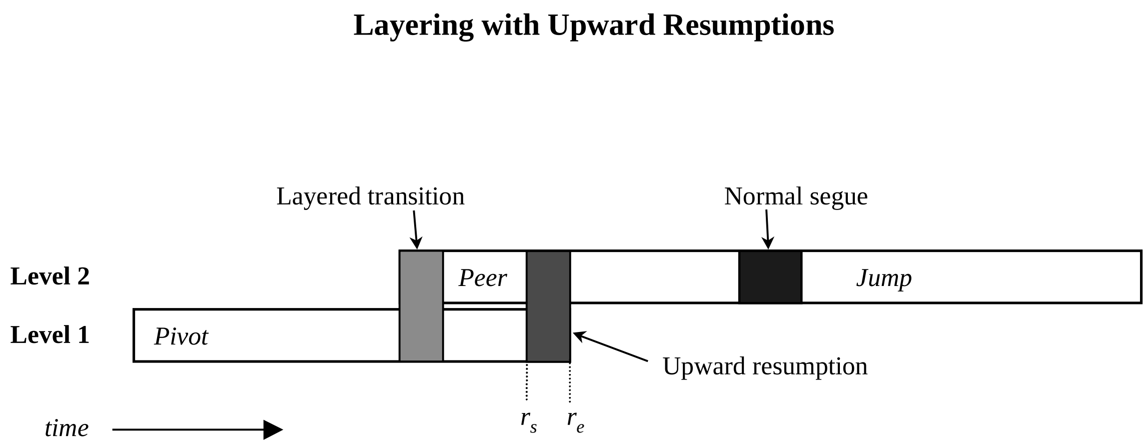


Figure 6-4: Motion Stack Containing Upward Resumption. In a motion stack like the one depicted here, successive layers build upwards. This motion stack represents a motion combination in which the actor repositions himself, looks around while still repositioning, continues to look around after settling into a new position, then jumps.

Resumption Duration

The duration of a resumption is more difficult to calculate than that of a normal transition because the ending pose of the resumption is "floating." In a normal transition, the starting and ending poses are known, and are independent of the duration of the transition. In a resumption, while the starting time and pose are fixed, the ending pose is dependent on the duration, since the ending pose comes from the resuming motion. For instance, in Figure 6-4, the resumption from the *pivot* to the *peer* begins at time r_s, and will continue until time r_e. Now r_s is fixed in time as the **release-goal** key-time in the *pivot*, so the exact pose from which we will begin the resumption is also fixed. However, each possible value for r_e corresponds to a different pose in the *peer*, so we have that:

$$duration = \mathrm{F}\left(pivot\left(r_s\right), peer\left(r_e\right)\right)$$

and

$$r_e = r_s + duration$$

where ‘F’ represents any of the analytical pose-distance computations we introduced earlier in section 6.1.3.2. Since the transition duration thus depends indirectly on itself, we violate the assumption of these computations that the final pose is fixed. In fact, the pose-distance equations cannot, in general, be solved for resumptions. The best we can do is to sample a space of transition durations (for example, centered on the duration of the transition *into* the layered motion), and select the one that is closest to the duration that the pose-distance equations would have computed for the same starting and ending poses.

Once we *do* have a duration for the resumption, we also have the ending pose, so we can compute the geometry of the resumption. Happily, the geometry of a resumption is computed identically to that of a normal transition.

Resumption Invariant Handling

As we mentioned in section 6.2.3.2, invariants can only be controlled by a single motion model at any time, and layered motion models are never allowed to usurp control of an invariant from any underlying motion. So when a layered motion terminates prior to the underlying motion, the downward (in the motion stack) resumption is simple with respect to invariants: the layered motion simply relinquishes control of all of its invariants when the resumption begins – they were only important to the layered motion, and the underlying motions maintain uninterrupted control of all of their invariants. However, when the resumption is instead an "upward" resumption as in Figure 6-4, the situation is more complicated.

When a layered motion executes, all of the motions underlying it have already executed, thus producing their motion curves and claiming their invariants. If executed in isolation, both the *pivot* and the *peer* in Figure 6-4 would activate foot position and orientation invariants to properly place the feet and prevent them from sliding. Because the *peer* is layered, however, when it executes, it discovers that an underlying motion already owns the foot invariants, and since the *peer* is really only interested in the actor's upper body and head, it simply notes that it should not activate its own foot invariants. But when the *pivot* terminates, relinquishing control of the actor's entire body to the *peer*, the *peer* must activate its own foot invariants. Therefore, the layered motion must, while executing, determine whether any of the motions underlying it terminate before it does, resulting in upward resumptions. If so and the terminating motion controls invariants that the layered motion *can* control, then the layered motion must activate those invariants at the point the underlying motion begins its resumption.

Lest the reader think this extra complication does not buy us much, consider that upward resumptions can occur quite frequently. Due to our (well justified, we believe) decision to begin and end transitions at the same instant for all bodyparts, upward resumptions are the sole means we possess to initiate an action while another action is still executing, and then continue the script from the just-initiated action (as in Figure 6-4).

6.3 Necessary Variations of Basic Transition

Simple segues express a linear progression of actions in which the actor executes each action uninterrupted, then moves on to the next action, forgetting about the first. Layering, as we have just seen, can allow the actor to execute multiple actions simultaneously. Frequently, however, we will require more sophisticated direction of the actor in order to convey specific emotions or trains of thought. Two motion operators that afford us further control over the actor are *pauses* and *holds*.

6.3.1 Pause

Often we will wish to pause an actor's progress in mid-action. For example, an unexpected thought may occur to an actor while raising a fork to his mouth, causing him to halt his progress momentarily. Or we may wish him to hold a particular pose within an action before completing the action. We afford pauses by layering a clip-Timewarp on top of the "final" version of each motion model's motion - the version that gets added in to the rough-cut. For each pause associated with the motion model, we insert a pair of keys in the time-warp that temporally stretches a single frame of the motion to fill the duration of the pause. It is then this stretched version of the motion that we add into the rough-cut. We must also, however, adjust the timing of the motion's invariants - any invariant active at the beginning of the pause should be active throught the duration of the pause, and any invariant that begins after the pause must have its activation and deactivation times shifted by the duration of the pause.

Like tweaks, invariants, and any other temporally relevant quantities attached to motion models, we track the activation time of pauses in the motion model's canonical time. In canonical time, a pause has only the canonical time equivalent of a single frame's duration (the single frame that we stretched out to create the pause). To simplify the mapping from animation time to canonical time and vice versa, we utilize the same time-warp we constructed to effect the pause: in mapping from canonical time to animation time, we simply apply the time-warp to the computation described in chapter 5, and in mapping from animation time to canonical time, we *first* apply the inverse of the time-warp to the animation time, using the result as the starting point for the animation-to-canonical computation from chapter 5. We could opt for a somewhat simpler implementation of pausing in which we layer just a single clip-Timewarp on top of the rough-cut, inserting all pauses there. However, this would only allow us to pause either all the action or none of it - by attaching pauses to individual motion models, we can, for instance, pause the action of a squatting motion, while allowing the gaze-at action layered on top of it to run continuously.

6.3.2 Hold

When we cause an actor to pause, we make him hold a particular pose or partial pose, with the intent of completing resuming the action that we are interrupting. In some situations, we desire the actor to instead hold a pose (typically a terminal pose of an action) while proceeding to perform the next action in the actor's script. For instance, consider a script in which the actor squats down, then picks up an item from the ground, then picks up another item from the ground. The most natural means of performing this sequence would involve the actor squatting down, then *remain* in a squatting position while picking up the two items from the ground. The *pick-up* motion model, however, prefers to begin and end in a standing pose, so a straight segue of these three actions will cause the actor to squat down, then rise back up while reaching for the first object, and remain with hips in the air as he proceeds to the second *pick-up*. To achieve the desired performance, we would like to force the actor to try to maintain the terminal pose of the *squat* as much as possible while picking up the two items.

To achieve this behavior, which we refer to as a "hold transition" (selectable as an parameter among the transition controls), we employ a motion blending mechanism similar to that used in layering. The idea is that we will create a composite motion that is partly the held pose from the preceding motion, and partly the motion of the action that we are next executing. The ratio of held pose to new motion is calculated on a per-bodypart-group basis, according to how necessary the bodypart-group is to successful execution of the action. We use the same rating of the importance of each bodypart-group to the motion that we did for layering motions. Thus the actor will tend to move only those bodyparts crucial to accomplishing the action's goal, otherwise more-or-less holding the previous pose. This automatic blending provides a decent starting point, but ultimately we would require the ability to individually specify the blend ratio for each bodypart-group.

Given the relationship of our methodology for holding to that of layering, one might wonder why we need a separate mechanism for holding when we could achieve much the same effect by pausing the "held" action and layering the "holding" action on top of it. Aside from the obvious fact that the paused motion will eventually resume, and must transition into some other action, there is a subtler difference involving invariants. When we use the hold mechanism, the held motion terminates at the transition, and along with it, all of its invariants. If we instead pause the held motion, all of its invariants are still active, so the holding action may not have access to all of the actor's invariants, and therefore may not even be able to execute. Thus the hold mechanism is more generally applicable. There is an example comparing regular segueing, pausing + layering, and holding in the animation section for this chapter, **http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter6.**

7 Synthesizing Animations from Motion Models

In the previous chapters we have discussed the main elements necessary to animate a single actor, namely motion models and transitions. In the interests of clarity, however, we have omitted many of the details involved in doing so, such as the mechanics of our constraint satisfaction engine, how we incorporate parameter editing and low level "tweaking" of an animation, and how we effectively and efficiently stitch motions together into a seamless, coherent animation. In discussing only a single actor's motion, we have also neglected any possibility for interaction between actors, which is generally crucial for effective storytelling.

In this chapter we will address all of these issues, which together fill in the gaps between the main ideas in this work, resulting in a functioning character animation algorithm. We will begin by discussing exactly how motion models are ordered, executed, segued, and layered for a single actor. We then move on to consider the extensions necessary to allow multiple actors to interact. We follow this in section 7.3 with a presentation of our constraint satisfaction algorithm, and then in section 7.4 we discuss how to incorporate high and low level editing into our motion synthesis pipeline. Finally, in section 7.5, we propose changes to our fundamental formulation that would enable real-time animation synthesis, discussing the impact of such changes and the benefits to be gained in doing so.

7.1 Composing Motions

We will begin with the task of composing the primitive motions produced by motion models. This involves organization of the multiple motion models that comprise an actor's script and specific data structures such as the clip-composite for stitching motions together. The organization and data structures we present were designed with simplicity and efficiency of the production animation task in mind. Along the way we will revisit the limitations induced by this architecture, to which we alluded in the previous chapter.

7.1.1 Motion Model Organization and Execution

Every time a motion model parameter changes in response to a user editing operation, the motion model must re-execute in order to produce new motion that reflects the new parameter values. Since the new motion will generally possess different geometry and timing than the old, any motion models that have dependencies on the altered motion model, whether it be through segueing or layering (or, as we will see shortly, inter-actor interaction), must also re-execute to adapt to the changes. Since each directly dependent motion model may also change upon re-execution, the re-execution must propagate to all motion models indirectly dependent on the altered motion model, which, generally speaking, is all motion models to its right in the animation timeline, and all motion models above it in the motion stack. Similarly, whenever a new motion model is inserted into or an existing motion model deleted from a script in progress, re-execution must propagate rightward, and upward. We will eventually detail precisely what occurs during re-execution, but first we will describe how motion models are organized within an animation, since it is clear that correct re-execution depends on determining the dependencies between motion models.

7.1.1.1 Motion Stack as Script

The script for each actor, consisting of a set of motion models and their compositional relationships, is maintained independently by an actor object. As we outlined in chapter 4, the actor also contains and manages all resources that must be shared by the motion models in the actor's script. These resources include both the rough and final versions of the actor's animation, all of the invariants active for the actor, and the actual geometric representation of the actor (*i.e.* its Character-Model).

The actor explicitly constructs and maintains the motion stack associated with the hierarchical relationships between layered motions, and uses it to efficiently determine those relationships and use them to correctly order script re-executions. The motion stack consists of an arbitrary number of levels, each level layered upon the one below it. The lowest level can contain only a single stream of motions, while all higher levels can contain multiple streams (a stream, introduced in the last chapter, is a continuous sequence of motions connected only by segues). Within a level, streams are ordered by increasing start time of the first motion model in each stream. Figure 7-1 shows an example of multiple streams in a single level.

Since layered motion models depend on their directly underlying motion for start-time determination, and on *all* underlying motions for resource allocation and motion curve blending, all underlying motion models to any given layered motion model must have already executed before the motion model is allowed to execute. The motion stack allows an actor object to easily execute its motion models in a correct order by traversing the motion stack left to right, bottom to top. The only complication arises when there are multiple streams on a level that may vie for resources. The first stream to execute will acquire the resources, and if the other streams overlap temporally with the first *and* require the same resources, they may be barred from the resources, and thus not allowed to execute. We determine precedence among streams by the entrance time of their lead motions, and therefore must be able to compute the entrance times of these motions *before* we actually execute them. This is the main reason we do not currently support the "goal positioning" method of specifying a layered motion model's entrance time (described in section 6.2.3.1), since it requires the layered motion model to execute before it can determine the entrance time.

7.1.1.2 Managing Invariants

The actor is also responsible for coordinating the use of invariants among all motion models in an actor's script. The actor maintains all possible invariants for a character, and grants access to individual motion models over their timespans of activation; invariants are identified by type, affected bodypart, and, where appropriate, point of application. Because invariants are, at their core, geometric constraints specifying trajectories for bodyparts to follow through time, the actor must ensure that the outdated portions of trajectories are removed from affected invariants before a re-execution of motion models can occur.

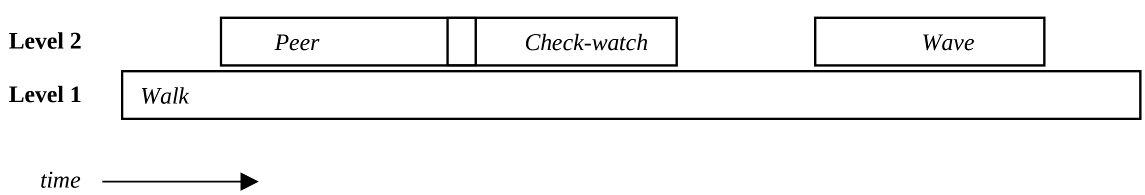


Figure 7-1: Multiple Streams in a Single Motion Stack Level. Recalling that levels in the motion stack are determined by the motion upon which a secondary motion is specified in reference to (*i.e.* layered upon), it is possible to have multiple streams in a single level. Above, two streams are layered on top of a *walk* action. The first stream contains two segued motion models (*peer* followed by *check-watch*), while the second stream consists of a single action, *wave*.

7.1.2 Basic Integration of Transitions and Layering

We discussed the concepts of transitioning and layering in the last chapter, but we have been ignoring the data structures and algorithms necessary to actually splice the many individual clips together. We will now explain the clip-composite, the class of clip designed specifically for this purpose. The composite splices together the entire motion of a single actor. Each actor maintains one, which we have referred to as the *rough-cut* of the animation.

If all we were interested in doing was segueing motions together, the composite would be as simple as the clip-blend, which allows us to overlap a series of full clips and blend from one to another over the course of the overlap. The introduction of layering, however, introduces several further levels of complexity. First of all, we must address the motion not as a single unit, but in bodypart-group chunks, since motions are layered and composed by bodypart-group. We add motion segments[30] to a composite individually for each bodypart-group, and it maintains a separate data structure for each bodypart-group, which we will refer to as a *group-motion*.

Second, each group-motion must be a type of motion stack. This stack is somewhat more restrictive than the motion stack implemented by actors, since here only one stream can exist per level; any segment added into the group-motion that overlaps any existing segment will be put into a higher level in the stack. Also, when a motion model adds a segment to a composite, it adds it in relation to a segment already in the relevant group-motion: when adding in a motion that segues from another, it references the segment of its predecessor, and the segment goes into the same level as the predecessor in the stack; when adding in a layered motion, it references the segment of its underlying motion, and the segment goes into the next free (unused) level in the stack. These measures allow the composite to find, for each group-motion, which motion to use at every instant of the animation: it begins at the top level and filters its way down, looking for a level that contains motion at the specified time; when it finds none, it looks to lower levels; however, in the face of upward resumptions, there may be no lower levels, in which case it looks upward.

Finally, because of the blending and differential operators we provide for combining layered motions with underlying motions, the composite must be self-referential, in that group-motions may contain segments whose motion expressions (directed graphs of clips) may now contain the composite itself as a node, thus causing a cycle. The reason for this situation is straightforward: when we blend a layered motion, we must blend it not with its immediately underlying motion, but with the *entire* motion built up of all the levels in the motion stack beneath it, and the composite is the only structure that contains the entirety of this motion.

The means of accommodating this necessity while avoiding infinite loops in our data structures is to acknowledge that when we reference the composite as an operand of a clip-mix or a clip-blend to effect a layered blend or differential, we wish to reference not the finished version of the composite, but only, for each group-motion, those levels of the stack *below* the level into which the blend or mix will be added. This necessitates two practices:

---

[30] We define a motion segment (or just plain segment) as a clip from which we only access the motion of a single bodypart-group. Segments come from the main motion produced by motion models, and also from transitions and resumptions.

1. All underlying motions to a layered motion must already be executed and added into the composite before the layered motion executes. In the previous section we explained how to do so, for this and other reasons.
2. Every access of *any* clip's value at a particular time must be augmented by an optional argument that is a vector of level numbers, one per bodypart-group. All clips except composites simply pass this parameter on to their operand clips (if any). A composite will use this argument, if present, to limit its access of its own segments in each group-motion to those in levels below the level numbers contained in the vector. When a motion model creates a blend or mix, it gives the appropriate level-number-vector to the clip, which it will store and use when accessing its operands. The blended clip will ultimately be added into the composite, and reside in the level above the level number dictated by the vector in each group-motion, so we will encounter no cycles in accessing the complete motion.

Incorporating Transitions

As we mentioned in the last chapter, we generally package the transition between two motions into its own, separate clip. The composite is designed to accommodate this arrangement: when we add a main motion segment into the composite, we also provide a clip to be used as a transition between the preceding motion in the stack-level (or underlying motion, in the case of layering) and the new motion. The transition clip can be overlapped with both preceding and new segments, with blending (determined by a supplied scalar function), or can simply abut them.

7.2 Inter-Actor Interaction

Up to the point of enforcing persistent constraints, we now know how to coordinate and combine multiple motion models to create a seamless animation for a single actor. However, much animation we may be interested in creating involves interaction between multiple actors. "Interaction" involves primarily two things: coordinated (and potentially cooperative) motion, and intentional physical contact (possibly sustained) between actors. Although dialog between people can often have an impact on the peoples' motions (body language, gesticulation, *etc.*), it is beyond the scope of this thesis to address such subtle interaction.

7.2.1 Coordination

Although we have not yet implemented it in our prototype, we believe we can support a fair degree of coordination between actors with slight additions to our animation framework. Actors can easily share the same goal when the animator selects the same target object for both actors' scripts. In cases where the target is not a concrete object, but rather an abstract location or other value, we can create a non-rendering “virtual” object[31] that we assign as the target so that when we move it, both actors will adjust their motions correspondingly.

We can enable temporal coordination between actors by extending our specification of transition start times to include, in addition to the existing absolute and predecessor-relative modes, a mode in which the start time of one actor’s motion is specified relative to an event in another actor’s script. Of course, this may cause one actor to run out of scripted motion while waiting for an event to trigger a transition into another action. We can handle this in one of two ways: first, make the actor automatically go into a pause at the end of the motion prior to the motion awaiting a trigger event, or second, allow the actor’s script to contain optional “filler” motion that will execute until the trigger event occurs.

7.2.2 Contact

The more challenging aspect of actor interaction is intentional contact, which we distinguish from accidental contact, because the latter should generally be handled by the collision avoidance/response mechanism. Physical contact is challenging to incorporate because its presence can dramatically alter what an actor is capable of accomplishing with any particular motion model. For example, keeping in mind that all objects in an animation are animatable actors, imagine a rock lying on the ground, executing a *motionless* motion model, when a humanoid comes along, picks up the rock, and throws it. If not for the contact with the humanoid, the rock would just lie there. With the contact, it traverses a complicated trajectory. The situation is even more interesting when contact occurs between two self-motivated actors. If one actor grabs another’s arm and pulls, it should cause a complex reaction in the other’s behavior, altering either the actor’s course (if moving) or stance (if stationary).

Handling contact in full generality is beyond the scope of this thesis. However, we have addressed the self-motivated actor / passive object type of contact, within a framework that we believe could be applied to solving the more difficult problem of self-motivated / self-motivated contact. Our approach centers on a means of establishing temporary connections between actors, which allow them to communicate directly and arbitrate for motion control. In our approach, intentional contact is initiated by one actor, and “received” by another. When the instigating motion model (*i.e.* the motion model of the initiating actor that actually causes the contact) executes and decides it wants to make contact with another actor, it creates a *connection* object that becomes attached to both its own actor object, and the actor of the recipient actor.

7.2.2.1 Initiating Contact

The connection object serves both as a description of the contact between the actors, and as a communication channel between the actors. Currently, the only type of connection we model is an approximation of a grasp, which is described as contact points on both initiator and responder, plus a relative orientation of

[31] The concept of using non-rendering, abstract objects as targets for constraints is standard fare in commercial animation packages.

the recipient to the initiator; contact points are described as IK handles. In addition to the physical contact type, each connection is created with a certain *intention*, to which the recipient gets to respond, as well as the time at which the contact is initiated. The intention specified by the initiator is one of the following:

- *Initiator-Controls*: The initiator would like to move about as it pleases, with the recipient adjusting its motion to compensate and maintain the contact. This is the usual intention whenever a humanoid picks up a passive object, for example.
- *Recipient-Controls*: The initiator would prefer to just react to the motion of the responder, so even though the initiator instigates the contact, the recipient drives it. This would be the intention when an actor grabs a hook that is swinging across a room, hoping to get a free ride.
- *Shared*: The independent motion of both actors should be factored into maintaining the contact. This would be the intention of contact established while shaking hands, for example.

When the initiator creates a connection, part of its initialization informs the recipient of the pending contact and the intent of the initiator. The recipient may respond with the type of contact that it is actually willing to support, based on the motion model it is executing at the time the contact is established. It can respond with any of the same intentions used by the initiator, and additionally the following, to all of which the initiator must be able to adapt:

- *Recipient-Stationary*: The recipient is unable to move, therefore the initiator must adjust its motion to keep the contact stationary.

7.2.2.2 Maintaining Contact

Throughout the time during which the connection is active, the connection adds constraints into the active set (joining the constraints imposed by invariants) that will maintain the contact: our grasp-type connection adds an object-to-object position nail constraint and an object-to-object orientation matching constraint. Regardless of the motion in the rough-cuts of the actors involved, the constraints will cause the constraint satisfaction engine to alter the motions sufficiently to maintain contact. The system determines the DOFs of the constraints according to the final intention of the contact: if the recipient is stationary or it is in control, only the DOFs of the initiator are active in the constraints; if the initiator is in control, only the DOFs of the recipient are active; otherwise (shared control), the DOFs of both initiator and recipient are active.

7.2.2.3 Terminating Contact

The initiator also terminates the contact (or chooses not to, if the contact is to be permanent). This often occurs in a different motion model than the instigator that initiated the contact. For instance, when an actor reaches down to pick up a stone in order to throw it, it is the *reach* motion model that initiates the contact, but the *throw* that terminates it. This is why connections are associated with actors and not motion models.

When the initiator terminates the contact, it deactivates all of the contact constraints, and it informs the recipient of the termination, through the connection. The connection is able to provide both parties with the final position, orientation, and linear and angular velocities of the contact point upon termination. A passive recipient's motion model can use this information to create a ballistic trajectory for itself. This is exactly what a *motionless* motion model does for a Rigid-Body recipient actor/object.

7.2.2.4 Effect of Contact on Motion Model Execution

When the initiator creates a connection, it must know the exact position of the recipient at that time; therefore, the recipient must execute its script up to that point in order to be able to compute its precise pose. However, once the connection is established, it will generally alter the motion of the recipient after the time of contact, which means the recipient must *re*-execute its script, at least from the starting time of the motion model that responded to the connection, onwards. In a scenario involving multiple contacts between self-motivating actors, this re-execution may go back and forth multiple times, with much of the effort being wasted.

In our current implementation, we limit contact to that initiated by a self-motivated actor, visited upon an otherwise motionless rigid-body. In this case we can avoid wasted effort by always executing motivated actors' scripts first; each contact initiated will cause at most one re-execution of a passive actor's script, which in general is just a single *motionless* motion model.

In the more general case, we would ideally like to execute all actors' scripts only up until the time of the first contact, establish the contact, then proceed with all executions until the making/breaking of the next contact, *etc.* In this fashion, no executions would be wasted. In fact, switching from entire-script executions to limited lookahead executions is one of the major aspects of moving to a real-time architecture. At the end of this chapter, we will discuss how a real-time architecture can thus make contact more efficient.

7.3 Constraint Satisfaction Engine

7.3.1 The Problem

When all actors have executed their motion models, composing the results into clip-composites, we are ready to produce the final version of the animation, which incorporates all of the constraints specified by invariants and connections. We wish to create the animation that is, for each actor, as close to the motion contained in the rough-cuts as possible, while ensuring that all constraints are satisfied at every frame during which they are active. Additionally, we do not wish to introduce any discontinuities into the animation.

This is truly a spacetime problem (modulo the physics), and has been solved as such by Gleicher [Gleicher 1998] and Popović [Popović 1999b]. However, even Gleicher's stripped-down algorithm and the subsequent reformulation by Lee [Lee 1999] as a multi-resolution, multi-pass IK solver are too computationally expensive for our needs. What we would like is to perform just a single pass of IK on our animation to satisfy the constraints. Combined with a well-optimized IK algorithm, this should be fast enough for editing reasonably sized animations at interactive rates, and is also best-suited to a real-time architecture in which speed is even more important.

As Gleicher notes, however, simply performing an IK pass to alter an existing animation will often introduce discontinuities into a previously smooth animation. Consider trying to alter a reaching motion in which the actor reaches for a target 12 inches in front of him, and we wish to change the target's distance to 18 inches in front of him. The most reasonable way to specify this deformation is by placing an *instantaneous* position constraint on the actor's hand at the time he acquires his target, nailing the hand 18 inches in front of the actor at that time. But now consider what will happen as we run IK on each frame consecutively, leading up to the target acquisition: since there are no active constraints before the target-acquisition frame, the actor simply does what he did in the original motion, which is to reach for a target 12 inches in front of him. At frame *target acquisition – 1*, his hand is at $12 - \varepsilon$ inches. Then, at frame *target acquisition*, the constraint activates, and pulls his hand to the desired position of 18 inches in front of himself, causing a leap of six inches in a single frame – a gross discontinuity.

The problem, of course, is that per-frame IK provides no means of factoring inter-frame coherency or smoothness into its solutions, whereas spacetime does, both implicitly (in the form of continuous basis functions for solution representation) and explicitly (in the form of acceleration minimization terms in its objective function). We could try to explicitly propagate the deformation by changing the instantaneous constraint into a trajectory the actor's hand must follow from the time it begins the reaching motion until it acquires its new target; however, this would generally result in a large loss of high frequency content, so it is to be avoided.

7.3.2 A Fast Solution for Motion Models

If we want to satisfy our constraints in a single IK pass without introducing noticeable discontinuities, we need a means propagating a low frequency description of the deformations induced by the constraints *before* we perform the IK pass, and without overly perturbing the high-frequency content of the animation we are deforming. We present a solution that capitalizes on several properties of motion models and on two further observations:

1. Most discontinuities that would arise in deforming an animation using per-frame IK occur at frames in which constraints are activated or deactivated.
2. Within its range of applicability (discussed in chapter 4), space-warping provides exactly the sort of deformation-propagation we require.

In general terms, what we will do is place a space-warp over the rough-cut, then use IK to compute the actor's pose at each frame in which any constraint activates or deactivates, and insert the resulting pose as a key in the space-warp. When we have performed this over the entire rough-cut, the animation produced by the space-warp will move the actor smoothly from deformed pose to deformed pose, maintaining high-frequency content[32], although persistent constraints will not be satisfied over their entire activation intervals. Now, assuming that the constraints themselves change smoothly over time (if they are persistent), we can run per-frame IK on the space-warped rough-cut without worrying about the kinds of discontinuities we described above.

To complete the algorithm, we must discuss some specifics of applying space-warping and IK. In general, we strive to place as few warp keys as possible, in order to perturb the original animation as little as possible. We must make sure, however, that we place enough. In particular, when we compute a pose at a frame because, for instance, a constraint on the actor's hand activates, we must not only place warp keys for all DOFs between the hand and the actor's root (*i.e.* hips), but also for any other DOFs involved in persistent constraints that may be active. For instance, if foot constraints had already been activated and were still active at that frame, we would also place warp keys for all DOFs in the feet and legs. This is necessary because the foot constraints contributed to the pose we just generated, so their effect must be propagated to keep the pose consistent.

Additionally, we must make clear that the IK algorithm we are applying in this algorithm utilizes the same minimal-pose-deviation objective function we defined at the end of chapter 4. At each frame, the pose we try to maintain is that provided by the space-warp, not the rough-cut.

There are several caveats and items we wish to make explicit about the use and performance of this algorithm:

- **Complexity.** Since we must first apply IK at each constraint activation/deactivation, the algorithm is technically not a single pass of IK. However, there are typically far fewer constraint activations than frames in the animation, and the performance should be much better than Lee's, which requires a minimum of three to four complete passes, plus signal processing.
- **Generality.** As we discussed in chapter 4, there are definite limits on the kinds and extent of deformations that space-warping can believably achieve. Therefore this algorithm should not be considered a true general-purpose deformation/transformation operator, capable of transforming an animation subject to arbitrary constraints. In our context, however, in which motion models generate motion that is already close to its goals, we should never be over-taxing the space-warping algorithm.

---

[32] Unless, of course, the constraints are too numerous and densely packed in time, in which case even spacetime solvers would fail to preserve the high frequencies.

- **Quality Guarantees.** We can guarantee that if the constraints are satisfiable, our algorithm will satisfy them at every frame[33]. Like Lee, however, if we wish to satisfy the constraints exactly, we can make no strict guarantees on the continuity of the solution. We can supply anecdotal evidence that in all of our experimentation, we have never observed a discontinuity caused by our IK constraint solver.

The result of running the constraint satisfaction engine is a pose per-actor, per-frame. Each actor maintains a clip-primitive with storage enough for its actor's entire animation, called the *final-cut*, and places each frame's pose into it for rapid playback and inter-frame interpolation. An added bonus of our algorithm is that once we calculate the space-warp, we can run the per-frame IK independently for each frame. This allows us to enhance interactivity when constructing long animations in a production setting. When the user makes an edit or other change to the animation, it may, with our current IK algorithm and computer speeds, take several seconds to run IK on the entire animation. If the user wishes to play back the animation after making the change, we have no choice but to pause interaction while the IK algorithm executes. However, if the user is instead making a series of changes, or (for example) moving the camera around after the edit, we can run the IK pass incrementally in the background, a few frames at a time. Thus the user can continue his interaction.

7.4 Editing and Propagation of Changes

We can now synthesize an animation from a high-level description consisting of hierarchies of motion models and their individual parameter settings. We have yet to discuss, however, how to edit an animation, and how to record the changes. As we outlined in chapter 3, the user can edit the animation at two levels, via setting motion model parameters to effect general and sweeping changes, and via "tweeks" to fine-tune the result. We have already discussed the types of changes we can make using motion model parameters, as well as suggestions for interaction techniques. What remains to discuss is how parameter changes propagate when an altered motion model is part of a script, and the means and mechanisms for performing tweaks.

7.4.1 Parameter Propagation

Earlier in this chapter we discussed how we might automatically adjust sets of motion model parameters to make combinations of motions work together more effectively. Ignoring such optimizations for a moment, we still must address the fact that motion models that are related in any way (via a transition of any

[33] Although we can easily and continuously degrade the quality for faster refresh by easing the stopping criteria for the IK algorithm.

sort or a chain of transitions) will generally possess interdependent parameters. For example, let us consider a three-action script, in which an actor bends down to *reach* for something, then *peers* at it, and finally *throws* it. All three motion models have parameters that allow the user to set the actor's general position and orientation. However, when motion models segue from other motion models, these parameters become dependent on those of the predecessor, since the following motion model must begin where the previous left off. This creates a forward temporal dependence for these parameters, since their values in the first motion model propagate forward in time to all following motion models.

We are left with a decision to make regarding these parameters for the later motion models: do we allow the user to edit them? If so, how do we reconcile the changes with earlier motion models? Let us imagine the user changes the "gross position" parameter of the *peer* motion model. There are only two ways in which we can reconcile this change with the (unchanged) gross position of the preceding *reach*: 1) we can change the gross position of the *reach* to match that of the new *peer* (*i.e.* back-propagate the changes), or 2) we can insert footsteps or other motion between the *reach* and the *peer* to change position from one to the other. We are unsure that there is a correct choice for all circumstances, or that there is any criterion for automatically choosing one over the other. We can, however, give the user the choice. In our current implementation, we avoid this issue by deactivating for editing any parameters that are dependent on those of preceding motion models. This results in a cleaner interface, but requires the user to seek the first motion model in a chain of dependents for editing propagating parameters.

7.4.2 Tweaking

We allow three forms of tweaking to the basic animation produced by a set of motion models. Each involves slight modifications to the last stages of our animation pipeline as we have presented it so far. The three types of tweaks are explicit keyframing, time adjustments, and warp adjustments.

7.4.2.1 Explicit Keyframing

The user can create persistent IK constraints or traditional per-DOF keyframes that are then added into the constraint set solved for by the constraint engine of the last section. In exactly the same way, the handles associated with invariants can be manipulated to change the invariants' values, as described at the end of chapter 3. Changes to invariants require that the actors' scripts be re-executed, but the addition or editing of other constraints requires only that we re-execute the constraint satisfaction pass.

The only other modification necessary to afford this tweak is that the added constraints and keyframes, while specified in animation time (*i.e.* frame numbers) become attached to the motion models active at their time of activation. We do this so that if the user later makes more fundamental changes to the animation, the tweak will still be applied to the same action to which it was initially applied, regardless of how the overall animation timing changes.

7.4.2.2 Time Adjustments

We can also allow the user to compress and expand the animation's timing and flow by interposing a time-warp between the rough-cut and final-cut. Since we do not wish to allow the animation to run backwards at any point, we must use a monotonicity-enforced time-warp, as described in chapter 4.3.2.1. Because this change sits on top of the motion model scripts (which are unaltered by this tweak), we must translate the timing of all interaction events with the user: we filter parameter editing times (for instance) through the time-warp before presenting them to the user, and we filter further tweak activation times through the inverse of the time-warp before passing them to the motion models.

7.4.2.3 Warp Adjustments

A subtler form of editing than imposition of keyframed constraints (which replaces the underlying animation) is to merely specify additional poses or partial poses through which the actors must pass, while maintaining as much of their original motion as possible. This is exactly the problem space-warping addresses, and fits in perfectly with our constraint satisfaction engine.

There is one subtlety in this problem, and that is in determining the timespan of influence of each warp pose the user specifies. We have attempted to do auto-key-insertion, utilizing the same space-warp already used by the constraint engine, and causing the motion models to (prior to editing) insert warp keys at their key-times, as appropriate. The idea behind this strategy is that the key-times of the actions being performed by the motion models form logical propagation barriers. For instance, if the user nudges an actor's elbow while the actor is reaching forward during a *reach*, it seems logical to bound the deformation induced by the nudge to the interval between when the *reach* begins and when it achieves its goal.

In our own experimentation using our system, we have had mixed success with this strategy. Most of the difficulty we have encountered stems from the fact that much of the time, we can accomplish our goal best by placing a tweak right at an existing warp key, but our interactive tools for locating and navigating the automatically inserted keys are lacking. Enhancing these tools would definitely enhance the value of this strategy. We might also try other approaches to auto-key-insertion, such as adding a new space-warp on top of the constraint engine's for each (or sets of) warp pose the user adds. This would force the user to manually specify the timespan of influence of the tweak, but affords more freedom.

Like the keyframed constraint tweaks described above, we attach added warp keys to underlying motion models by translating their activation times into the canonical times of the relevant motion models, so that the tweaks will stay with the actions as the animation is further altered.

7.5 A Real-Time Architecture for Motion Models

Occasionally in the last two chapters we have referred to limitations of our current architecture for animation, and stated that they could be removed by adapting motion models to a real-time architecture. While it is technically future work (and will be mentioned again there), we would like to explain our ideas on this topic, since it is relevant to material already presented, and we have developed these ideas fairly deeply. Many of the ideas (particularly the auto-pruning motion tree described below) developed from a several-month collaboration with Zoesis Inc. in designing their real-time character animation subsystem, which is currently in active use in 3D games and interactive narrative.

We will begin by defining exactly what we mean by a real-time architecture. The architecture we have been describing throughout this document, which we have occasionally referred to as a "production architecture" is designed for creating and editing potentially long, entire animations. Therefore it gives us access to an actor's entire script, and renders (executes motion models and performs IK cleanup) the entire animation before allowing us to play it back at any speed, any number of times. A real-time architecture, by contrast, expects to receive a single-use, dynamically changing script, fed to it in small chunks. It must constantly produce new frames for display, moving the animation timeline forward in step with real-time. Its view of the future script is imperfect, since the entity feeding it script may revise its plans mid-action, and may not know, at any time, what it will be doing more than a few seconds into the future. Additionally, a real-time architecture should be able to provide more comprehensive information flowing from the motion models up to the higher-level control system. Interaction techniques and tweak mechanisms are less important because the automatically generated motion is generally consumed as is.

The three consequences of these characteristics that have the greatest impact on the architecture design are: reduced future information and consideration of future script; the potential need to revise a motion model's motion while its current motion is already being consumed; performance constraints of maintaining a minimum frame-rate. This leads us to an architecture in which each actor's script consists of a small queue of motion models representing the current and near-future actions. Once animation time has proceeded past the last frame to which a motion model contributes, it is discarded. Similarly, we maintain only a few frames of final-cut cache for each actor; the size of the cache is determined by the "Non-interruptible Animation Chunk" (NAC), which we estimate to be around 2-3 frames ($1/15^{th}$ to $1/10^{th}$ of a second)[34].

While playing back the finalized animation contained in the current NAC, the system is simultaneously computing the next NAC-sized bit of animation. This process and its results are more complicated than for the production architecture, because new specification from the animation driver (AI, user input, *etc.*) may not only specify new actions to perform in the near or far future, but also changes to the goals and style of the currently running motion models. The former poses no problems: additional actions join the current actions in the script, and execute along with them in the current NAC computation. The latter, however, requires a fundamental addition to the animation pipeline.

The problem is that motion models will not, in general, produce motion satisfying new goals that is consistent with previously generated motion satisfying previous goals. For instance, if an actor is in the middle of *reach*ing for a target when informed that the target has changed position. We can re-execute the *reach* with the new target and align the resultant motion temporally with the old motion, but were we to simply switch from the old to new motion, a discontinuity would be noticeable. Our only recourse is to transition from the old motion to the new. There are, in fact, two different cases to consider.

In the first case, either the altered goal is quite different from the original, or the original motion has passed a "point of no return," beyond which it must finish executing. An example of the former is if the actor is midway through reaching for a target on the floor when informed that the target is actually on a shelf above his head. An example of the latter is any time subsequent to takeoff in a *jump*. We group these cases together because we handle them in the same way: rather than attempt to replace them, we instead terminate them (former) or let them end naturally (latter), then create a normal transition into the revised action, which will segue from the original. In the case of the former, the result will be a perceived discontinuity, which should be acceptable since it results from a sudden and drastic change of plan. The latter case will result in a delayed reaction to the new instruction, which also should be understandable since it results from physical limitations.

In the second case, the new goals are not so radically different from the originals, and we have the opportunity to make the change seamlessly and with minimal loss of high-frequency detail. An example of this situation is a change to the path down which an actor is walking. To do so, we first re-execute the motion model with the new parameters/goals; however, we insert its motion, along with that of all the other motion models in the script, into a *new* rough-cut, without destroying or resetting the old. We proceed further to the first stage of the constraint satisfaction cleanup stage, in which we impose the constraint activation/deactivation space-warp on top of the new rough-cut. Before completing the cleanup, however, we create a new "base" animation by creating a blended transition from the old motion to the new.

This process may occur anew at each NAC computation, so the "old" animation may not be a single space-warped rough-cut, but rather a binary tree of clip-blends whose left-hand children (first operand) are other clip-blends representing earlier versions of the animation, and whose right-hand children (second operand) are single space-warped rough-cuts, which were the "new" animation during the NAC computation in which they were created. Since the duration of the transition from old to new is bounded, we can continuously prune this tree from the bottom-left upwards, replacing a node by its right child when the transition into that child's motion is complete.

---

[34] Recent correspondence with Joe Bates of Zoesis suggests that, in fact, a NAC longer than 1/30th of a second may cause noticeable hesitation in characters' reaction times in some situations.

There are several interesting consequences of this architecture when compared to that of the production architecture:

- Since there are fewer motion models in a script, it is now more practical to allow goal positioning of layered transitions (section 6.2.3.1), as well as richer inter-actor contact.
- Because the animation is computed in small steps, and because we now have a more established method for dealing with changes to a motion model's operating environment, it becomes much easier to incorporate physical simulations into an animation with full interaction than with the scripted animation.
- We would need to experiment to determine the computational bottleneck in this architecture. In the production architecture, practice shows that IK is clearly the limiting factor. Due to the potential cost of evaluating the clip-blend/rough-cut tree and the increased number of motion model re-executions, this may not be the case in a real-time architecture.

8 Results

We have now finished our presentation of the concepts and algorithms developed in this work. In this chapter we will describe how we evaluated those ideas and the claims we made about them in chapter 1. A significant portion of the effort expended on this thesis went into creating a working animation system based on motion models that enabled us to test, at least informally, our claims regarding the efficacy of motion models, the ease with which we can create animations, and the quality of those animations. In section 8.1 we will present our prototype animation system, *Shaker*, describing the extent to which it implements the ideas we presented in earlier chapters, and how one uses it to create animations.

To obtain a sufficient number and variety of base motions to test our ideas, we employed motion capture. In section 8.2 we will describe several aspects of the process, including the capture technology, resampling/filtering algorithms, and mapping techniques.

In section 8.3 we describe two user tests we conducted in order to test the quality of individual motion models, and the quality of entire animations, including transitions. We will describe the goal and setup of each test, and present the results with our conclusions about them. Finally, in section 8.4 we will discuss the limitations of our approach.

8.1 Shaker – A Prototype Implementation

We built a substantial system in C++ in order to experiment with the concepts in this thesis and actually create animations. We call this prototype animation system *Shaker*, as it is an evolution of and companion to our earlier, IK-based animation system, *Mover*. In this section we will first describe Shaker's overall intent and design, then enumerate the specific features (corresponding to concepts from this thesis) it does and does not implement, including the actor types and motion models it supports. We will then explain certain aspects of Shaker's interface in order to convey our idea of how to create animations, and so the reader can gauge how much effort is required to make an animation. We will end the section with links to and descriptions of several illustrative animations we made using Shaker.

8.1.1 Design

Our goals in designing Shaker were threefold:

1. To test the richness and quality of animations we could create with a handful of motion models

2. To demonstrate by example that we can actually create an extensible set of good motion models for a variety of actions, and that the effort involved is not prohibitive

3. To demonstrate that, given the high degree of control motion models provide over an animation, the amount of specification actually required to create an animation is small – this is the strongest claim we can make about the ease-of-use of our approach without carrying out user studies, which we were ill-equipped to do

We already described the control flow of a production animation system based on motion models in chapter 3.2. In Figure 8-1 we see a structural layout of Shaker, which implements the control flow with which we are already acquainted. Each animation is managed by a single director object, which owns and coordinates all of the actors involved in the animation (recall that all physical objects in the animation are actors), and handles all I/O and user interaction. Each actor in turn manages all of the motion models that comprise a single actor's script. These major object classes utilize various support libraries, including those for IK, clip- operators, style libraries, basic drawing, character models, and a stripped-down *Scheme*-like language for saving/restoring animations.

In support of goal 2 above, we attempted to simplify motion model creation by implementing most of the API we described in chapter 4.3. The first few motion model classes we created, before we had developed the API, each took weeks to complete (starting from fully mapped base motions). By the end of our implementation phase, supported by the API and some experience, we could add a simple gestural motion model in about an hour[35], and a more complicated motion model such as *throw* in about a week.

[35] We implemented *foot-stomp, relief-wipe, shrug,* and *sigh* in a single afternoon.

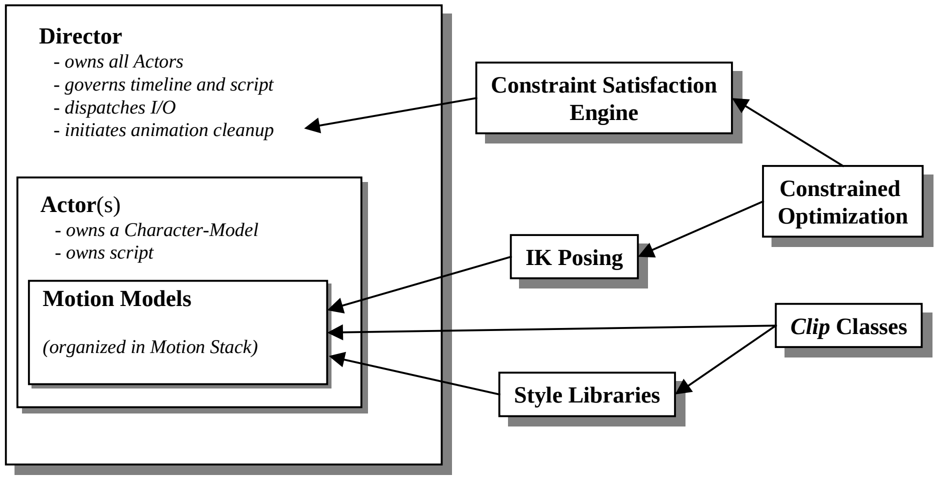


Figure 8-1: Structural Diagram of Shaker. Nesting denotes ownership of the inner object by the outer. Arrows denote use of a module by another.

8.1.2 Features Implemented

Shaker implements the following features of the animation system we have been describing

- **Motion model API.** Except as noted in the next section, Shaker implements the API in section 4.3.
- **Support for multiple Actors.** An animation can contain many actors. In Shaker, actors are drawn from one of three predefined types, all composed of simple geometry. The three types are *humanoid male*, as an instance of the character model class humanoid, *ball*, as an instance of rigid-body, and *cube*, also a rigid-body. When instantiating an actor from one of these types, the user can set its dimensions and color.
- **Motion model scripting.** Shaker allows animation scripts that include unlimited segueing and layering of actions, as described in chapter 6. The underlying machinery can add, delete, or move motion models in the script at any time, although the graphical interface for moving motion models or changing their layering relationships is unimplemented.
- **Tweaking.** Shaker provides one layer of space-warp with pre-inserted keyframes into which the user can insert further warp keys (as in chapter 7.4.2), generated through IK posing or direct joint manipulation.
- **Actor interaction.** Currently, Shaker supports a limited form of connection. In order for a motion model to be able to receive a connection, it must be explicitly created to be cognizant of connections (ultimately, this should be a generic part of the API). Currently, the only motion model so created is the *motionless* class associated with rigid-body actors. Initiating or terminating a connection *must* be an explicit part of a motion model class, and is currently part of the *throw* and *reach* classes.

- **Motion models.** Shaker currently contains the following fifteen motion model types defined for humanoid, plus *motionless* for rigid-body, from which to construct animations:
  - *Jump-both-feet.* A standing jump from any position and orientation to any other position and orientation, achieving a specified height at the jump's apex. Possesses two styles, one completely keyframed, the other mostly motion captured (one of the base motions was too badly distorted, so we substituted a keyframed animation for it).
  - *Reach.* Can touch a particular point in space, or acquire a ball or cube (if in reach) by selecting a contact point on the object. Only simple, contact-based grasping is performed, since humanoid does not model articulated fingers and thumbs (when the actor's palm touches an object, the object sticks to it until the actor lets it go). Possesses a single style, *reach-normal*.
  - *Throw.* Performs a throwing motion towards a given target (abstract position or target object) with control over the ballistic trajectory. If actor currently holds a rigid-body (as a result of a *reach*), the object will traverse the trajectory. Possesses the styles *throw-overhand* and *throw-sidearm*.
  - *Peer.* Looks at a given target (abstract position or target object) for a specified duration. Since humanoid does not possess articulated eyes, *peer* fixes gaze direction completely via manipulation of the actor's head's orientation. Possesses styles *peer-normal*, *peer-stealthy* (a slow, exaggerated motion), and *peer-fast* (whips about to locate the target).
  - *Foot-shuffle.* Allows the actor to shift a bit in any direction or pivot slightly in either direction by taking a series of small, shuffling steps. Possesses a single style.
  - *Ah-ha*. A *gesture* in which the actor swings up his hand (in substitution of an outstretched index finger) in response to a sudden flash of inspiration.
  - *Arms-akimbo.* A *gesture* that folds the actor's arms and sways for while.
  - *Catch-breath*. A *gesture* in which the actor bends over and pants in exhaustion.
  - *Check-watch.* A *gesture* that causes the actor to gaze at his wrist.
  - *Fist-Shake.* A *gesture* in which the actor shakes his fist in anger at a selectable target.
  - *Foot-stomp.* A *gesture* in which the actor stomps his foot, as if in a tantrum.
  - *Head-shake.* A *gesture* in which the actor shakes his head back and forth, as if in negation. Possesses three styles: *casual, sad,* and *disbelief.*
  - *Relief-wipe.* A *gesture* in which the actor wipes the back of his hand across his brow, in relief.
  - *Sigh.* A *gesture* in which the actor performs an exaggerated sigh.
  - *Shrug.* A *gesture* in which the actor performs a shoulder shrug.

8.1.3 Features Not Implemented

Due to time constraints, we were unable to incorporate some of the ideas we have presented into our prototype. From the motion model API, we implemented neither significant collision detection nor a balance invariant. We did not test the transition duration learning algorithm, the frequency-band transition geometry synthesizer, or the inter-motion model parameter optimizer. We were disappointed that we did not have time to implement a *walk* motion model, since its presence would have broadened the scope of interesting animations we would have been able to make. Unfortunately, we ran short on time, and the *walk* would have taken longer to create, partly because it would have been the first repetitive motion model, but largely because the data we acquired for walking was in bad shape and would have required great efforts to make usable (see section 8.2.3).

8.1.4 Animating with Shaker

One of our main motivations for this work is to enable more people to partake in visual storytelling by lowering the bar to creating interesting, high-quality animations. The best automatic motion synthesis in the world would be useless to this task without an easy to use interface. While this thesis does not pretend to be a thorough investigation of interface design, we have given much thought to this problem, and tried to reflect our ideas in Shaker. An interface to the functionality provided by this work must contain at least four basic functionalities:

I. A means of constructing and editing the general script for the animation – *i.e.* the set of motion models it contains.

II. A way to access and edit the parameters of all of the motion models.

III. A means of placing, editing, and removing tweaks.

IV. A means of playing back and navigating through an animation (*i.e.* virtual camera control).

Figure 8-2 shows a screen snapshot of Shaker's user interface, with various graphical areas labeled for reference. Area 1 is the *script window*, in which the user constructs the script for each actor, addressing task I above. Actors are arranged vertically down the left hand side of the scrollable and resizable window, and each actor's script extends to the right of the actor's icon. In the prototype, the representation of each script element (*i.e.* action) is simply iconic – a box long enough to contain the name the user gives each particular instantiation of a motion model. By clicking on any icon in the script window, the user activates and brings to the foreground a *motion model control panel* containing widgets that enable editing all of the parameters of the motion model instantiation (including switching its style among any of those defined for it), forming our solution to task II above. Area 2 shows an example of a motion model control panel for a *jump* motion model.

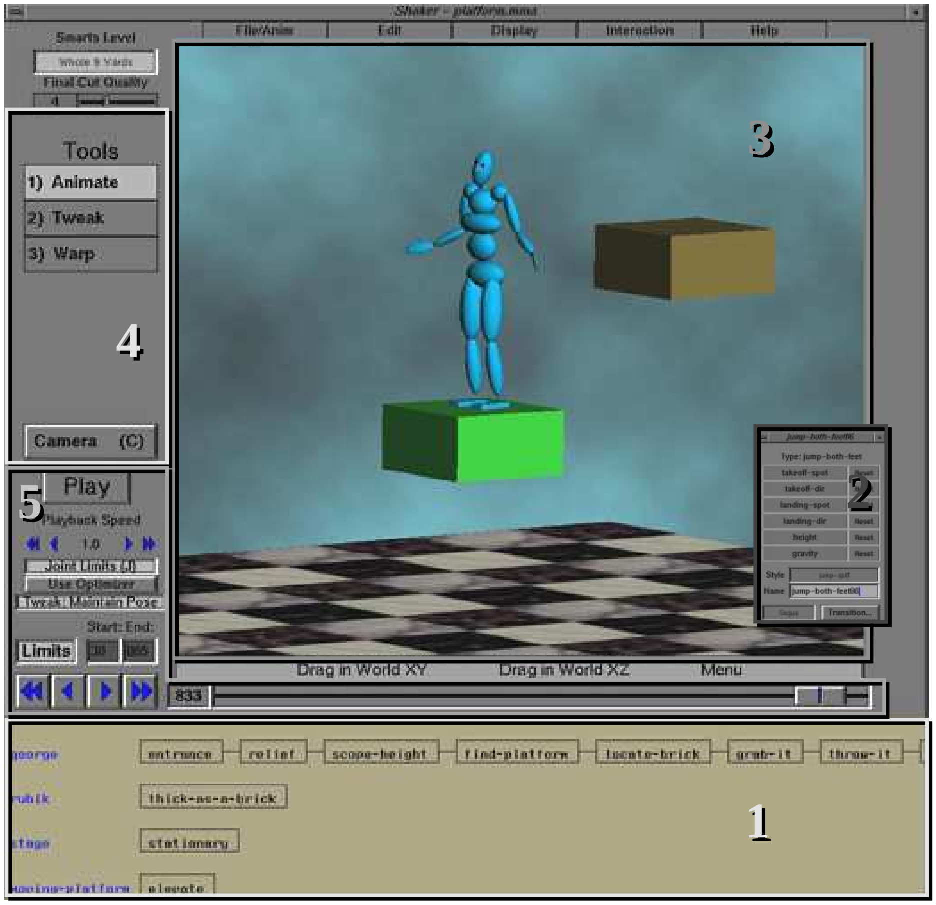


Figure 8-2: The Shaker Interface. See text below for description.

Dialog boxes allow the user to add actors and motion model instantiations to the script via menu selection; motion models can be added as segued or layered off of any motion model in the script. Ideally, given more development time, we would tie the length of a script element to the duration of the motion model instantiation and a zoom factor, and the horizontal dimension of the script window would function as a timeline. Dragging a script element would cause its transition timing and relationship parameters to change. We have since discovered this paradigm to be very similar to how non-linear video editing interfaces function, and this seems a fitting metaphor.

Area 3 is the main work area, where we perform all direct manipulation, camera control, and viewing of the animation. To enable unambiguous direct manipulation of motion model parameters, the user chooses which parameter he wishes to set, by activating it in the corresponding motion model control panel, which displays the appropriate manipulation widget in the main work space, which the user can then select and drag.

Since motion model parameter manipulation and IK tweaking involve similar operations, we always clearly place the user in one of two modes: *Animation* mode, in which the user can edit motion model parameters, and *Tweak* mode, in which the user can add tweaks corresponding to any invariant type the system supports. If, while in tweak mode, the user activates a motion model parameter from a control panel, the system automatically switches to animation mode. The user can switch between modes using the buttons in area 4 (or keyboard shortcuts), and also activate a quaternion arcball-based camera [Shoemake 1994] for changing the view.

Finally, area 5 provides basic transport controls for playing back the animation, including a scrollable timeline and valuators for setting the playback speed.

8.1.5 Examples

We used Shaker to create many small animations to test various concepts, and also several longer animations that tell simple short stories. We used two of the longer animations in our empirical animation quality test, which we will describe and present in section 8.3. In the online animation section for this chapter, http://www.cs.cmu.edu/~spiff/thesis/animations.htm#chapter8**,** we present several examples of animations we created with Shaker. Each is explained in detail in the online page, but we will give brief descriptions here as well.

The first example shows the powerful changes to an animation we can make using motion model parameters. It first shows the animation we would get by simply concatenating five motion models together. It then shows the resulting animation from making six simple direct-manipulation changes to motion model parameters.

The second animation shows two versions of a sequence of jumps, in which the style of jump in the second version is different from that in the first. We made the change with a simple menu selection, and all goals of the animation are still met.

The third animation shows three different humans reaching for objects and throwing them. This animation demonstrates the wide range of targets that both the *reach* and *throw* motion models are capable of accepting. We animated all three actors using the same sequence of steps, then changed the style for the green actor to throw sidearm.

The fourth animation demonstrates the results of applying several tweaks to a jump animation to make it into a spread-eagle. We can also see the need for collision detection in this example, as the actor's arm passes through his knee while landing.

The final animation is a longer scene that could take place in an adventure story or game (with higher quality props, of course!), in which the hero enters a "room," surveys his situation, then takes some action.

8.2 Motion Capture Data Acquisition

Although we are very interested in applying our approach to hand-animated motion, we based the motion libraries developed for this thesis on motion capture data, for two principle reasons. First, there was the matter of economics: while working directly with an animator would undoubtedly have been more rewarding, it would have been logistically difficult and potentially expensive. We were, however, easily able to inquire at several motion capture studios, and able to get a day's studio time for a fee that fit our budget. Second, using motion capture data gave us the potential to evaluate our results more objectively. If we were to show someone animation that was transformed from a Disney-style motion, all we can ask the viewer is whether they liked it or not. If, however, we show someone animation transformed from motion captured data, we can ask how convincingly human it looks, particularly in comparison to the motion captured motions themselves. We will make use of this in our empirical evaluation.

8.2.1 Studio Setup and Technology

We acquired our data at Lamb and Company, Minneapolis, Minnesota. Their capture technology utilized the Flock of Birds™ magnetic system, in which one attaches 16 sensors to the actor's body. Each sensor returns its position and absolute orientation at approximately 50 Hz. We employed a fairly typical sensor placement, with one sensor on the head, one on the front chest, one mid-back, one at the tail-bone, and one on each shoulder, forearm, back-of-hand, thigh, shin, and top of foot near the toes.

Magnetic capture technology requires that a pre-set stage, containing no metal in the vicinity, be carefully calibrated beforehand. Calibration involves measuring the magnetic field emitted by a powerful magnet, at regular intervals over a lattice defined around the stage. Our stage was limited to approximately 9 by 5 feet, which precluded capturing any running or running-jump motions.

8.2.2 Mapping Data to Our Humanoid

The typical method for mapping motion captured marker data onto a digital character is to create "nail" constraints on the character that correspond to the sensor positions on the live actor's body, then use IK or spacetime to make the virtual sensors align with the positions of the real ones, dragging the digital body along with them, thus posing it. One of the attractions of magnetic capture over optical (in addition to price) is that it gives us orientations for each sensor, in addition to position. This makes mapping the data onto a digital character much easier: if all we have is the position of each sensor over time (as with optical tracking), then it is crucial that the positions of the virtual sensors with respect to the character's body closely match the positions of the actual sensors on the actor's body, which is difficult to do. Orientation data, however, requires no such careful correspondence because, discounting rippling of the flesh, the relative orientation of a sensor is the same across an entire bodypart.

We utilized our own IK animation system, *Mover*, to perform the mapping. We first filtered and resampled the raw data, using exponential coordinates [Grassia 1998] for the orientation data so that we were able to employ b-splines for both position and orientation data. Computing the mapping turned out to be somewhat of an art. Given that much of the sensor data was redundant given a fair digital approximation of a human, with joint limits and appropriate articulations, *and* that the correspondence between actor and character was not perfect, it became a process of trial and error to determine, for a particular motion, what subset of the sensor positions and orientations, when mapped via constraints to the character, would produce the most faithful reproduction of the original motion.

8.2.3 Quality of Results

Unfortunately, two factors conspired to degrade the quality of our motion captured data. First was that Lamb and Co. had just moved to a new space, and the area in which their stage was erected was not free of metal, and was not sufficiently calibrated, given the degree of deformation induced in the magnetic field by the metal. Therefore, many of our motions, particularly those that ranged over the stage, such as our walking motions, contained substantial distortions of some sensor positions and orientations over some portion of the motion. Second, while we were forewarned that magnetic technology could not sample fast-moving motions at above 60 Hz (with which we were prepared to cope), we had not counted on the sensors producing large amounts of noise when moving fast. This affected particularly the position readings of the sensors, which in effect became useless during parts of the throwing, jumping, and limited running motions we performed. We discovered these problems in the data only after the data was delivered to us some weeks after the shoot, and we were unable to undertake a second capture session due to cost.

8.3 Animation Quality User Test

A principal metric of evaluation for this work is the quality of results it produces. Quality of animation is difficult to measure objectively, as discussed by Hodgins [Hodgins 1997] and others. Ideally, we wish to measure the quality of motions produced by individual motion models over a range of input parameter values, and separately measure the quality of transitions our system generates. Given our starting data, the first task is the easier of the two, since each base motion of a motion model is actually a motion captured performance, so we can compare “real” motions to synthesized motions. We cannot do the same for transitional motion, since we had not conceived of such a test at the time we acquired our data, and so have no “real” transitions to which we can compare our synthesized ones.

Let us focus first on evaluating individual motion models. There are two possible routes of evaluation. One possibility is to numerically measure the difference between two motions, one real and one synthesized, that accomplish the same goal. Measurements could be conducted both kinematically, or, perhaps, more usefully, dynamically. Unfortunately, due to the very nature of our transformation-based algorithm and the strategy we used to pick motions to capture, if we have a captured motion that accomplishes a certain task, then it is contained as a base motion within a motion model, so if we ask the motion model to generate a motion that satisfies the same goals as the base motion, it will essentially just spit the base motion back out. Thus we would be comparing a real motion to itself, not evaluating the transformation capabilities of the motion model.

This leaves us with the second evaluation method, which is a kind of "Turing Test of Motion" in which we present a subject with two or more motions, one of which is a real (base) motion, and the others which have been synthesized by a particular motion model, and ask the subject to pick out the real motion. As noted by Hodgins [Hodgins 1997], the presentation method for the motions can have a crucial impact on the subject's impressions. Rendering the motions onto as realistic a humanoid as possible would be problematic, since there are many human DOFs and secondary motions that we do not model (faces, hands, hair, *etc*.). For our test, we chose to present the motions on the same articulated figure that we used to map the motion captured data in the first place (pictured in Figure 4.1). This representation has the advantages of being a recognizable humanoid without forcing the subject to require (for instance) facial expressions of it, since it does not have an articulated or realistic face.

Formulating the test on an intermediate representation of a humanoid also addresses one of our chief concerns, which is how to insulate the performance of motion models from the quality of the input base motions. As we will see in the results of our second test, the quality (or, more precisely, the lack thereof in cases) of some of our base motions was discernible to the subjects. However, we can use the results of this test to answer not the question of whether our motion models produce realistic motion, but rather the question of whether they degrade the quality/realism of the base motions upon which they are built. This is a far better question for us to answer, because it has meaning for the application of motion models to cartoonish or other non-realistic motion as well the simulation of humans.

Evaluating the quality of our automatically generated transitions is much more difficult. Since all of our captured motions are of primitive actions, we have no "real" transitions with which to compare. We considered showing a sequence of actions to the subject, and asking him if he could pick out the transitions, not telling him how many there were. However, just because a subject can identify a transition does not necessarily mean the quality of the transition is low. We also considered specifically pointing out a transition to the subject, and asking him to rate its quality on a scale of one through ten. This also is problematic because he has nothing with which to compare it. We finally settled on a less transition-specific test in which we show the subject a longer, coherent animation consisting of many motion models transitioned together, and ask the subject to point out any problems he sees in the animation, judging it as a simulation of a human. For each problem he encounters, he is asked to record the approximate time at which it occurs and a brief description of the problem. We reason that we can utilize the results to gain insight on the quality of transitions, since we know when the transitions occur in the animation, and therefore can determine how many of the problems are due to transitions.

8.3.1 Experimental Setup

We conducted a user test at Carnegie Mellon to evaluate the quality of animations we could produce with Shaker. We must emphasize that while we took as many measures as possible to ensure rigor in the test (described below), it was by no means a formal user study, so the results should be considered anecdotal. We will now describe the test, in general to specific details; we actually gave two versions of the test to two different groups of subjects. Static versions of these tests can be viewed at http://www.cs.cmu.edu/~spiff/thesis/test/test1.htm and http://www.cs.cmu.edu/~spiff/thesis/test/test2.htm. You can see the answers to the Turing tests by clicking the "Finish!" button – you do not need to supply an email address.

If you wish to take the test, please do so now, since the answers are given below. **Be warned**, however, that each version will download approximately 30M of video data, and will only display properly on a PC or Macintosh that has the Apple QuickTime™ player version 3.0 or higher installed (available for free download from http://www.quicktime.com). We have reproduced the static layout of the main parts of the test in Figure 8-3 and Figure 8-4 for quick reference.

As we hinted in the previous paragraph, we conducted the test inside a web browser, since we have excellent tools for designing web-based video presentations, and collecting data from web-based forms is straightforward. Since we did not have the resources to create a 3D player for our animations, nor the time to convert our output to an existing 3D format, we chose to render our test animations to 2D video. This format has the advantage of being embeddable in a web page, so that the subject could view the entirety of each video without losing context of the others, and could rapidly switch between related videos. The test consisted of two parts, displayed on the same web page consecutively, corresponding to the two tests we described above. We administered the tests in a controlled environment on our own machines, onto whose hard-drives we pre-loaded the videos, for rapid and smooth playback.

Part one of the test implemented the long-form animation evaluation. We actually created two versions of the test, each using a different animation/story, and our server decided which version to give each subject based on how many of each version had already been completed by previous subjects. One of the two stories is depicted in Figure 8-3, in which the hero tries to hit the red box by throwing balls at it, but the box keeps evading his throws. In the second story (not pictured, but also viewable from our website), the hero approaches a large crevasse with much trepidation and jumps over it safely.

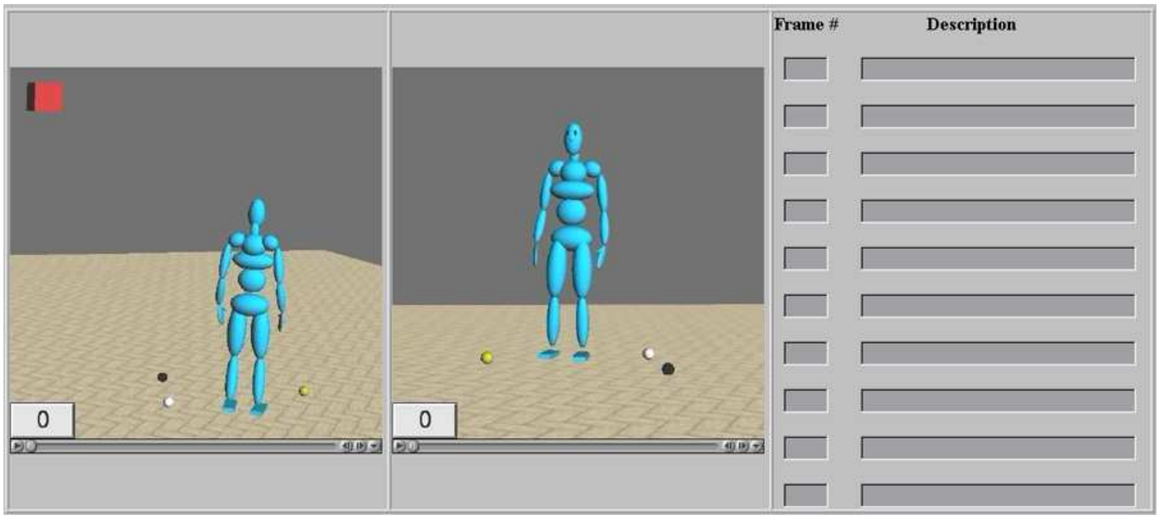


Figure 8-3: One of two possible "Part One" tests. The accompanying instructions appear *above* this section of the page in the browser:

> Below are two different views of the same animation. The one on the left shows all of the action, and the one in the middle shows a close-up of the "hero", whose movements we are interested in evaluating. In the lower left-hand corner of each animation is a counter that shows the frame number at each instant of the animation. You can view each animation by pressing the small play button in the far-lower-left corner of each animation window or by double-clicking in the window, and you can also drag the oval-shaped slider to view any instant of the animation.
>
> We want to determine how human-like the hero moves. Please evaluate the quality of the animation by using the inputs to the right of the animations to note any places in which the hero's motion looks unnatural. Enter the approximate frame number at which the unnaturalness occurs in the column labeled "**Frame #**", and give a brief description of what bothers you next to it in the column labeled "**Description**".
>
> **IMPORTANT:** Please do not feel obliged to use all of the problem reporting entries: report only those instances in which you truly think the motion is unnatural. If you find more than ten, please let Sebastian know! Also, you may ignore small penetrations of the hero's feet through the floor (*i.e.* do not report them as problems), and you may also ignore the fact that he uses "sticky hands" to pick up objects, since he has no fingers. When you are done, please scroll down to the second section.

Part two of the test corresponded to the motion Turing test, and possessed the same structure for both versions of part one. We used the same four groups of motions each time, but within each group, we presented the four animations in a random order to each user. We tested the *jump* motion model, the *throw* motion model with both overhand and sidearm styles, and the *reach* motion model. We limited ourselves to four in order to bound the time required to take the test (generally fifteen to twenty minutes), and chose these four because they employ the most sophisticated transformations, thus best testing our methodology. In each group, one motion was a pure base motion, and each of the others incorporated as large a deformation as possible along one or more parametric dimensions of variation, within the intended parameter range. The deformation we endeavored to maximize was a composite of two parts: first, we tried to ensure that the combination of base motions (section 5.2.2.1) produced by the motion model's rules was not strongly weighted to any single base motion, *i.e.* we chose no examples that were very close to the goals/parameters of base motions. Second, we also maximized the further deformation applied to the blended motion (section 5.2.2.2), where possible. For instance, in the *throw* examples we set the position of the target so as to cause the hero to throw in substantially different lateral directions, thus exercising the pivot deformations we described in section 5.2.2.2.

As you may notice from the web pages or figure captions, we specifically did not try to explain to the subjects *how* we created the synthesized motions, because we wanted them to bring as few preconceptions and prejudices to their judgments as possible.

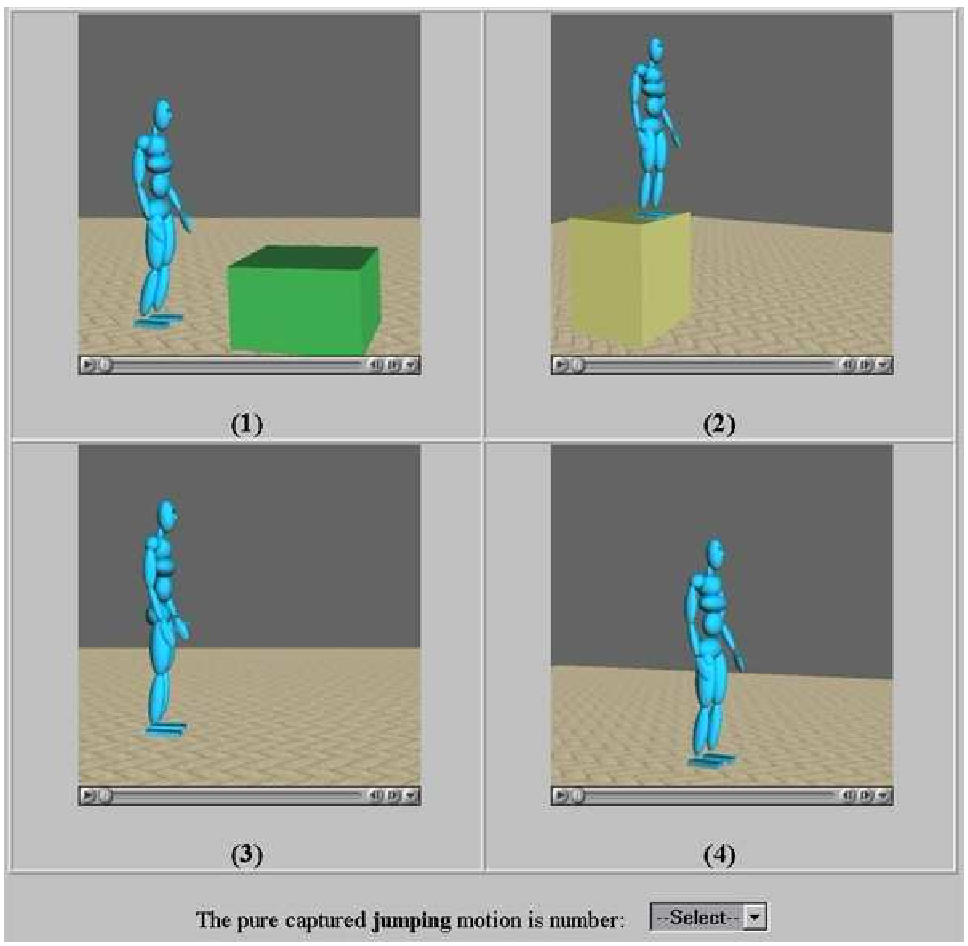


Figure 8-4: One permutation of the "Part Two" test of the *jump* motion model. The following instructions appeared above a sequence of four such groupings.

In the second (and last!) section, you will be presented with four sets of four similar motions each. In each set, one of the four motions is the motion of a live actor, captured and mapped onto the hero's body as directly as possible. It should therefore be of the highest quality. The other three motions were synthesized. Note that the same caveats from Part One about foot and hand motion apply here.

Your task is to decide, for each set, which is the motion of the live actor. Each motion is labeled "1", "2", "3", or "4", and there is a field below each set in which you should select the number of the motion you chose as the "live" motion. If you cannot see a difference in quality between the four motions, just guess! When you are done with all four sets, please make sure you have entered your email address at the bottom, and click the "Finish!" button.

8.3.2 Test Results

We hosted our test over a period of five days, inviting members of the CMU community in general to participate, with a raffle for a $50 gift certificate as incentive. Forty-five subjects took one or the other form of the test, with about three-quarters of the subjects being computer science students or professors, the rest comprised largely of administrative assistants, non-CS students, and one visiting alumnus.

8.3.2.1 Motion Turing Test Results

The results of the modified Turing test were generally encouraging, with one unexpected and interesting occurrence.

1. **Jump**. Eighteen out of forty-five, or 40% of the subjects selected the actual captured motion, which, in the online version of the tests, is choice 1.
2. **Throw-Sidearm**. Only four out of forty-five, or 9% of the subjects selected the actual captured motion, which is choice number 1 as well. Twenty-six subjects (57%) picked choice 2. We noticed this trend developing during the testing period, and asked one subject why he had chosen it. He replied that it was the fastest, most energetic of the four throws, so he figured it was least likely to be computer generated animation, which he generally associated with slow, robotic movements.
3. **Reach**. Eighteen out of forty-five, or 40% of the subjects selected the actual captured motion, which, in the online version of the tests, is choice 4.
4. **Throw-Overhand**. Thirteen, or 29% of the subjects selected the actual captured motion, which is choice 2.

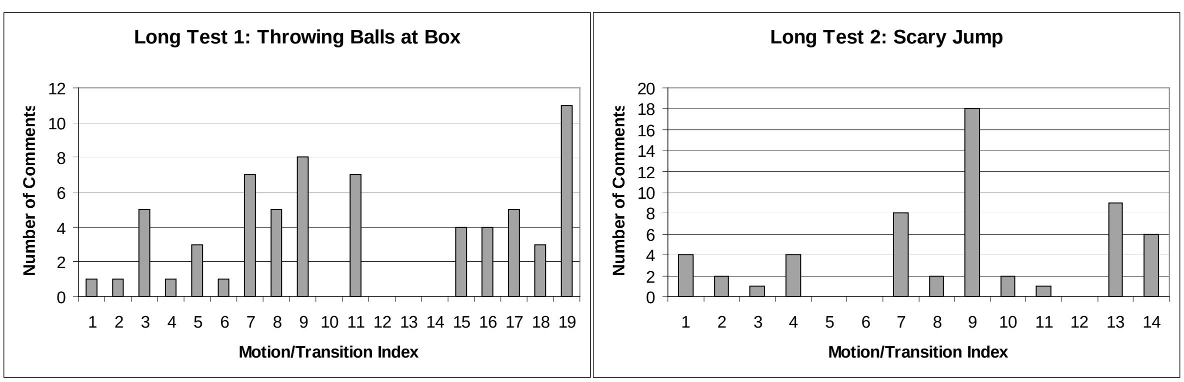


Figure 8-5: Tallied comments for long form motion tests. For each of the two tests, we placed all of the comments into bins according to the motion segment to which they most directly applied. Test 1 consisted of 10 motion models, with 9 transitions between them. Motion models were given bins with odd indices in increasing chronological order, and transitions were assigned to even indices. Thus index 1 on the left represents the actor looking at the red box, while index 2 represents him transitioning into looking down at the yellow ball. Test 2 consisted of 7 motion models, labeled similarly. Index 14 does not correspond to any motion segment, but rather represents criticisms about "directorial" decisions we made.

The mean number of correct selections per subject was 1.13, with a standard deviation of 0.93. Thus the average subject was only able to distinguish between captured and synthesized motions only 28% of the time. We take this as a reasonable validation of the initial quality of our motion models.

8.3.2.2 Long-Form Animation Results

The long-form animation test represented an extreme torture test of our methods, because we required the subjects to examine the animation in minute detail, using the "scrub bar" of the video viewer to traverse the animation frame-by-frame and at multiple speeds. Subjects thus discovered and reported subtle "problems[36]" of the animation that they might not have noticed if the animation were presented in a movie or game. Our primary aim in giving this test was to get an idea (although not as quantitative as for individual motions) of the quality of our transitions by observing whether people complained more about the motions generated by transitions or by motion models. As Figure 8-5 shows, we had noticeably fewer complaints about transitions than about the motion generated by motion models. This is obviously not conclusive evidence, since it could mean there were fewer problems in the transitions, *or* that the problems in the transitions were simply less pronounced than those in the motion models. As we mentioned previously, this is the best we can do without motion captured data as a basis for comparison.

The test also gave us valuable insight into how people judge motion. We placed our first use of the word "problems" in quotation marks in the previous paragraph because many of the things people took issue with were actually critiques of our original motion captured performances! We will now provide some detail about each of the two animations and the critiques of them.

Test 1: Throwing Balls at a Box

The first animation was a scene in which the hero sees a floating red box some distance away, and tries to hit it with balls he finds on the ground at his feet. The box, however, keeps evading his well-aimed throws, and he eventually gives up in despair. We keyframed the box's animation, since all of our motion models apply to humanoids. The hero's script consisted of ten motion models connected sequentially via simple segues: *peer* (up), *peer* (down), *reach* (down), *throw*, *peer* (up), *peer* (down), *reach* (down), *throw*, *peer* (up), *head-shake*. To achieve the state of surprise the hero evinces on noticing the box move, we terminated the *throw* motions early to transition into the *peer* motions where he stares at the box.

Twenty-three subjects critiqued this animation, with a total of 66 comments, for an average of just under three comments per subject. The most common criticism applied to the dejected *head-shake* motion (index 19 in Figure 8-5), and stated that the hero's head hangs too low while shaking. This, however, *is* the actual motion of the actor who performed it, who has a very long neck. The second most common criticism is a legitimate critique about our *peer* motion model (indices 1, 3, 9, 11, and 17). We asked the subjects to judge the animation as one performed by a human, and humans have articulated eyeballs, which they fully

[36] We shall explain why we quote *problems* shortly.

employ when locating a target. Our hero, however, has fixed, button eyes, and consequently utilizes only his head and neck to orient his eyes at a selected target. Thus, as the subjects noticed, he tends to cock his head oddly to gaze at some targets. Finally, we note that the transition that elicited the most complaints in either animation was the transition from the first *throw* to the surprised *peer* (index 8), which is somewhat of an off-balance lurch, due to the early termination of the *throw*. Both this transition and the other early-termination *throw*-to-*peer* transition, suffer from lack of consideration of balance and dynamics, which we discuss further in the “Limitations” section below.

Test 2: A Scary Jump

In our second animation, the hero shuffles up to the edge of a precipice, takes the measure of the cliff and the other side, then shakes his head in uncertainty. Deciding to go anyway, he jumps to the other side; amazed that he made it, he rapidly looks back down into the chasm he just crossed, then wipes the back of his hand across his brow in relief. It consists of seven motion models segued together: *foot-shuffle*, *peer* (down), *peer* (forward), *head-shake*, *jump*, *peer* (back/down), *relief-wipe*. In this animation we utilize the “hold pose” variant of the basic transition, holding the “peer down” pose throughout the “peer forward” and “uncertain head shake” motions that follow. For dramatic purposes, we made the distance between the cliffs be quite large; therefore, the jump the hero executes is on the border of believability for a human, thus we expected more complaints about the jump.

As you can see in Figure 8-5, we did indeed get many critiques of the jump (index 9). There were several valid complaints stemming from the facts that we were forcing the motion model to extrapolate from its base motions, and that the primary base motion being utilized was keyframed. We also had several amusing complaints. One subject believed the feet should leave the ground at a more staggered interval, while another thought they should leave closer together. Three subjects thought the hero moved too slowly through the air, when in fact the *jump* motion model computed his parabolic trajectory and its timing in strict accordance with earth gravity. The same criticism of the *head-shake* (index 7) that we saw in the first animation was also repeatedly leveled here. Another aspect of the original actor’s performances that was repeatedly criticized (particularly in the last *peer* and *head-shake* – indices 11 and 13) was that the hand/arm motion was too exaggerated in the follow-through.

8.4 Limitations

Like any research, this work was in large part an exploration. In particular, we tried to push on two current frontiers in animation:

1. How much can we limit our dependence on dynamics in getting realistic results in animating humans? Enforcing accurate dynamics is one of the most expensive operations in character animation today.

2. How much more generally applicable and efficient can we make motion transformation methods by applying detailed structure to the problem?

We believe our work has made substantial progress on both of these fronts, and uncovered several interesting lines of further inquiry. We have also been able to discern limits, both current and future, of our approach.

8.4.1 Dynamics & Stability

Since we are not interested in biomechanical modeling or optimization, our sole possible interest in dynamics is its aid in creating natural appearing motion. The hypothesis we tested in support of our inquiry along frontier 1 above is whether, given dynamically sound base motions, our methods can combine and transform them without noticeably perturbing the dynamic properties. The answer we divined is both yes and no.

Our results provide a fairly strong affirmative answer to the hypothesis above. Using only very simple and inexpensive dynamics calculations (such as ballistic trajectories), our initial motion models produce motion realistically indistinguishable from captured base motions a majority of the time, over the intended parametric range of variation. Obviously, this would not hold true were we to introduce contacts or large collisions unanticipated by the motion models, but no current dynamics algorithm is able to produce more than convincing passive responses to such occurrences, either (whereas the goal is to alter the character's *intentional* action in response).

We have, however, catalogued deficiencies in transitions that arise from lack of consideration of the dynamics of motion. This is perhaps not surprising, since we generally must synthesize *new* motion for transitions without the benefit of transforming existing motion according to domain-specific rules, as do motion models. We observe problems primarily when a transition must bridge a large gap in poses involving (usually) a shifting of the actor's weight/balance. We see two cases of this in the first long-form animation test, "Throwing balls at a box." In both cases (indices 8 and 16 on the left in Figure 8-5), we transition from a *throw* to a *peer*, beginning the transition before the *throw* has completed its recovery phase, thus leaving the actor in a somewhat awkward pose. The transition into the *peer* occurs as somewhat of a stiff lurch, and results in an awkward stance.

There are two factors at work here. First, our simple Hermite interpolation method of computing the transition geometry ignores interaction and coupling between various bodyparts. Adding in parts of the frequency spectrum from the prototype transition, as we suggested in chapter 6.1.4.2, might relieve some of the unnaturalness, but we believe only dynamics or a more sophisticated, annotated database of prototype transitions could truly solve this problem. Second is a problem of *stability,* which we observe also in the last *reach* to *throw* transition (index 14) in the same animation (although apparently no-one else found it offensive). In the first two cases, the transition does not realize that the stance it leaves the feet in is awkward, and even if it did, it does not know how to realistically insert a step that could reposition the feet. In the last case, the transition begins by stepping forward with the left leg, which, from the starting pose of reaching down to the left, causes the actor to become very unbalanced. Adding the balance operator introduced in the motion model API in chapter 5 would help here, but we also need a general purpose "stepper" generator that is tied to a stance analyzer for transitions, capable of inserting a believable step into a transition when it would increase the stability or reduce the awkwardness of the following motion.

8.4.2 Continued Need for Experts

An animation system built on well-crafted motion models gives an unprecedented ability to non-animators to create highly detailed character animations. However, while parts of the motion model creation process can be automated or semi-automated, we believe motion models for non-trivial motions will require at least the input of an expert animator[37]. We think this is actually a positive, since animators can supply insight and knowledge that would essentially require a solution to the general AI problem to derive automatically. Nevertheless, the strong preference in Computer Science is for automatic algorithms that eliminate the need for human guidance.

---

[37] It *may* be possible to reduce dependence on a technical director/programmer by expanding the scope of the API and expressing it as, for instance, a structured graphical programming language.

9 Conclusion

We have now described all of the concepts and algorithms constituent to this work, and presented the results of our experimentation. In conclusion, we will reiterate our contributions to 3D character animation, as we first stated in chapter 1. Then, in section 9.2, we will revisit the motivating applications for this work, and detail how well our results pertain to each. Finally, in section 9.3, we will explore some of the avenues of future investigation raised by our work.

9.1 Contributions

In this thesis we showed that by adding structure and domain-specific knowledge to motion transformation, we are able to solve a number of problems in character animation:

- By encapsulating motion transformation rules in motion models, we significantly extend the range of motions we can produce by transforming example motions using relatively simple operators. Motion interpolation can cover a wide range of task-space variation, given the right base motions, but performs poorly at meeting and maintaining precise geometric constraints. Motion warping and other deformation operators can transform a motion to meet a precise set of new instantaneous constraints while preserving the high-frequency spectrum of the original motion, but are limited to a small range of deformations. A motion model designer is able to specify how to combine transformation operators to use each to its best advantage, according to general principles we provided. Moreover, we have defined a compositional language for building motion expressions, in which motions and operators for combining and deforming them are functionally equivalent. Thus a motion model designer can easily compose rich transformations consisting of many deformations and combinations of raw base motions.

- The structure we apply to motions allows us to separate the common from the unique – motion models abstract the control structure and rules of motion for a particular action, while styles fit unique performances of the action into the more general structure. Consequently, our methods are the first that allow us to transform entire families of task-related, but stylistically different motions in the same way. Use of the same control structure (*viz.* motion model) means that we can maintain all of a user's high and low-level animation specification (goals/targets, tweaks, and all other parameter settings) while radically changing the style in which a character performs a motion.

- By adding geometric invariants that specify the crucial geometric properties of a primitive class of motion that must be preserved throughout transformation, we are able to integrate high-level parametric control of animation with very low-level fine tuning capabilities. In addition to their duties in transforming and combining primitive motions, these invariants allow the user to make arbitrary keyframe or space-warp modifications to the animation without perturbing its defining characteristics, since all modifications are filtered through the invariants.

We also investigated transition synthesis and other motion combinations to unprecedented depth. This is the first work of which we are aware that considers computing not just the geometry of a transition, but also the transition duration. We presented a method that uses a mass-displacement measure of pose distance and the information contained in prototype transitions to compute transition durations that depend on both the magnitude of the transition and the style of the segued motions. We demonstrated the superiority of this computation to existing methods, and suggested an extension to it that applies machine learning to improve its quality further. We also explored new operators for combining motions, including the *differential layering* operator that allows us, for example, to perform any action while shivering (by only layering the high-frequencies from a shivering motion onto the action), and the *hold* operator, that causes an actor to perform an action while trying to maintain a given pose as much as possible.

Lastly, we introduced a new constraint satisfaction technique that combines space-warping with a single pass of IK. It is able to satisfy constraints exactly (where the constraints do not conflict), and, given a comparable IK algorithm, is much faster than existing methods. Although it provides no guarantee of inter-frame coherence, we have never observed it to introduce discontinuities.

9.2 Applications

In chapter 1, we presented the major motivating applications for this work: autonomous agents, visual storytelling, and fast animation prototyping. We will now discuss how well our results can be applied to each of these tasks and which areas need further work.

9.2.1 Autonomous/Guided Agents

With the modifications we described in chapter 7.5, our architecture is quite well suited to serving as the "motion/action generator" component of both autonomous agents and user-guided avatars. In both cases, a higher level component of the system (AI "brains" or user input) supplies a dynamically evolving script for the actor(s), with limited or imperfect knowledge of future actions. Generally, new instructions will be intermittent; this poses no problems to our architecture since layered motions, once completed, resume back into whatever action was previously executing. We have recent, compelling evidence that the general strategy we proposed in section 7.5 is sound, as the results of our previous collaboration with Zoesis in this area are "very pleasing" [Bates 2000].

To be practical, however, our approach must improve in two areas. First, we require a faster IK algorithm. As we have already mentioned, IK is currently the bottleneck of our algorithm, and we estimate that IK must be sped up by at least a factor of five for the entire animation engine to function within the 2-10% CPU allocation that motion generation is generally given within game engines, for instance.

Second, collision detection and avoidance, almost completely absent in our current system, play at least two separate, important roles. Let us consider the "platform" genre of video game, in which the user controls an avatar who runs, jumps, manipulates things, and possibly fights (*e.g.* Abe's Oddysee, Spyro the Dragon, Tomb Raider). When the user directs the avatar to pick up some object or push a button, the system must compute a collision free path to the target, if one exists. However, much of the skill required to play these games lies in precisely controlling the direction and velocity of jumping and running actions so as to clear hurdles or move from platform to platform. In these instances, if the specified jump would cause the avatar to smack into a wall, we do not wish to correct it, but rather detect it and trigger the correct crashing response. Thus, in some circumstances we require only simple collision detection and response, while in others, full collision avoidance.

One difference between the autonomous agent and typical user-controlled avatar is in the desired parameterization of motion models. The type of goal planning that most AI systems perform fits well into the kinds of parameterizations we have described here. The kind of control typically afforded by games, however, generally specifies directions of travel/manipulation rather than goal-targets. Therefore we would also need to rework the parametric interfaces to applicable motion models.

After many years' of experience working on the problems of building reactive, expressive 3D agents from the ground up, Joe Bates feels that in the long run, the split between mind and motor control should be eliminated, to keep the NAC short enough to mimic living creatures, and to make it easier for character builders to cross boundaries. We believe that tighter integration of the motor control (*i.e.* animation subsystem) and mind of an agent is crucial to enabling character builders to easily build rich characters with complex personalities and behaviors. However, we also believe that the limiting factor on the size of the NAC is the performance of the animation system itself, and that eliminating the boundary between mind and motor control will not allow us to shrink the size of the NAC substantially.

9.2.2 Visual Storytelling

The application our current design most directly fits is that of simplifying the process of telling stories through animation. While much work remains to be done to enable the non-sophisticate to be able to create a weekly or monthly comic, our work addresses one of the three principle subproblems involved: animating sophisticated characters via high-level, directorial-like scripting. The other two subproblems address the availability of 3d character models and worlds, respectively. Both can be solved by more powerful creation tools, or by the availability of large libraries (optimally customizable) of models.

Of course, the relevance of our approach to this problem hinges on the quality of our results and the ease of use of our methods. We have made our cases for both of these properties in section 9.1.

9.2.3 Animation Prototyping

The same properties that make our approach useful for visual storytelling could allow a practiced user or professional animator to put his ideas into (rough) motion in a short amount of time. This has potential application in feature animation production, principally in two areas. First is animating crowd scenes. In an animated movie, imbuing main characters with rich personalities will almost always require an animator's careful attention in every shot, if not every frame; thus the benefit of our techniques is questionable. For animating large numbers of characters, none of which our attention is drawn to for long, motion models provide a convenient and compact means of specifying the characters' motion.

The second potential application is in blocking out animation quickly to see how a particular scene plays out, or to communicate an idea to a director. The need for this ability is unclear, however, since traditional storyboards seem to function well for communicating story ideas, and good animators can generally see a scene in their heads from only a little motion. If our tweaking tools prove powerful enough, it is *possible* that animators would find use in animating a scene by fleshing it out with motion models, then tweaking the results to get just the look they desire. This is largely untested, since we lacked the animation skill to do more than use them as a proof of concept.

9.3 Future Work

The work in this thesis is but a foundation and beginning to what we hope is a substantial step forward in lowering the threshold to creating high quality character animation. As we have mentioned throughout this document, much work remains to be done. In addition to those items we described as "part of the thesis, but not yet implemented" (such as the machine learning-based transition durations and signal decomposition-based transition geometries), we will discuss here some of the more crucial and/or interesting extensions to this work that we have considered.

9.3.1 Animating the Rest of the Body

Animating a character's face is crucial to effective storytelling. It may be possible to adapt some of our methods to facial animation, although it would be challenging. The only facial articulations that can be controlled by angular parameters are the eyeballs and the jaw: the majority of facial movement arises from fleshy skin movement due to deformation of the underlying muscles. Animators generally achieve this motion by directly controlling the vertices or control points of the surface representation of the model's skin, and there may be hundreds, or even thousands of such controls.

The need for individual manipulation of each of these controls for every facial movement would make animating sophisticated expressions untenable. Commercial animation systems already offer higher levels of control to make the task more manageable. Animation Master [Hash 1999], for example, allows any combination of skeletal and skin deformations from a canonical base pose to be stored in a *pose*. Multiple amounts and types of deformation can be specified at different values of a *pose parameter*, creating a primitive that produces different poses at different values of the pose parameter. While animating, the user can keyframe values of this parameter to produce motion along the deformation path defined by the pose primitive. Thus complicated poses can be encapsulated, reused, and animated for things like expressions and mouth positions associated with phonemes.

At this level of abstraction away from the raw control points, it may make sense to apply our structure for encapsulating and parameterizing timing and coordination knowledge, as well as transitions between expressions or partial expressions. A scheme like the one described above easily accommodates operations such as segueing and layering.

We believe that other types of bodily animation, such as hair/fur movement and bodily muscular deformation can be best generated by other techniques. Animating these elements is a rich topic of research; some of the interesting approaches have utilized physical simulation [Anjyo 1992, Wilhelms 1997, Van Gelder 1997], and some animator visual programming to build deformation functions directly into the character's skin-mesh [Hash 1999].

9.3.2 Better Auto-Keying and Tweaking Controls

Our current auto-keying approach to placing tweaks, in which the effects of a tweak are bounded by automatically pre-inserted warp-keys, is quite cumbersome. To achieve understandable results, the user must be aware of where in the timeline the auto-keys are, but is unable to get this information from the “tweak mode” display. Also, it may be that the user can only obtain the desired effect by placing his own boundaries for the warp induced by the tweak. As we discussed in section 7.4.2.3, we need not only a more informative interface to tweaking, but also the ability to add entirely new layers of space-warp on top of the auto-keyed layer.

9.3.3 Dynamics and Simulation

In section 8.4.1 we discussed the need to incorporate dynamics into the motion synthesis of articulated characters. We will undoubtedly also wish to incorporate many other types of physics-based simulations into our animations, such as rigid-body simulations, particle systems and continuum equations for simulating fluids, smoke, fire, hair/fur, and other phenomena. If the interactions between our scripted actors and these simulations are to be only one-way, *i.e.* actor-to-simulation, then we can perform the simulations at any point after the animated characters’ motions have been computed. If we wish for simulated systems to affect scripted characters, then we must utilize the real-time formulation of our animation engine that we described in section 7.5, and weaken the meaning of a “Non-interruptable Animation Chunk” (NAC). Now, after executing the next NAC in sequence, we incrementally execute all active simulations, frame by frame, inside the NAC. If any collision between a simulated element and a scripted actor occurs, we shorten the duration of the NAC so that it ends at the time at which the collision (or other type of interaction) occurs. Thus when the next NAC (no longer quite non-interruptable!) executes, it will do so in knowledge of the interaction, and can adjust accordingly.

9.3.4 Collisions

As we discussed above in section 9.2.1, appropriate collision detection and response is a crucial element of extending our work to serving as the animation engine for autonomous or guided avatars. Enforcing non-penetration is also useful in direct manipulation interfaces, such as ours. Various methods exist for performing efficient collision detection [Moore 1988, Baraff 1992]. These techniques also generally address physically realistic collision response for rigid, non-interpenetrating bodies, and we could incorporate them into our framework in the same manner we discussed in the previous section for incorporating simulations. Calculating the correct response for self-motivated characters, however, is another matter.

Fully automating actor responses to arbitrary collisions with the environment and other actors is a difficult AI problem, since a given actor’s response will depend not only on the physical parameters of the contact, but also on the actor’s personality and state of mind. Agent architectures such as Hap [Loyall 1997] are designed to handle “interrupts” (such as collisions) from the motor control layer, and allow a character builder to program responses to the interrupts. Motion models whose action is intimately tied to collision and contact, such as grappling or couples-dancing, will explicitly model and/or anticipate these events and contain rules for responding to them appropriately. For handling incidental collisions, we envision providing two minimalist, yet flexible measures.

The first is to label the times at which collisions occur as addressable events. The animator can utilize these tags to trigger transitions to "collision response" motion models, or a higher-level scripting mechanism can change an agent's subsequent goals or actions. The second is to provide a default collision response generator that treats the colliding objects as unmotivated, altering the motion of each to prevent penetration as would the rigid-body methods we mentioned at the beginning of this section.

9.3.5 Optimizing for Combinations

We have thought extensively about how we might automatically improve the quality of motion combinations. Although we have not implemented any of these ideas, we present them now in slightly more depth than other items of future work.

Between the level of explicitly constructing goal-driven plans and low-level motion synthesis, there lies a potential for optimizing the combination of motions specified by a script to work together and influence each other in appropriate ways. The optimization problem could become arbitrarily complicated and far ranging, so we will define a bounded range of effects and quantities that may be included. We will begin with a motivating example, then describe the scope of the problem we intend to solve.

Consider, as we did in chapter 3, a script in which the animator tells the actor to *jump*, and then, while jumping, *reach* for a target overhead – in other words, a *reach* layered on a *jump*. The goal of a "motion optimization" involving the two motions would be to make the actor jump so as to bring himself as close as possible to the target, thus bringing it in reach, and for him to acquire the target (if possible) when closest to it.

The first bound on any such optimization must be that it does not invalidate or nullify anything that the animator has explicitly specified. This means that the optimization can only set or change motion model parameters that the animator has not set herself. The second bound is in how goals and requests can be specified to the optimization. We allow non-linear equality and inequality constraints involving the values of a motion model's invariants evaluated at specific or general times, plus the global position and orientation of the actor under each motion model's control. For instance, in the example above, the *reach*, realizing that it is layered and thus not in control of the actor's root position, may make the request (in the form of a constraint equation) that its underlying motion (the *jump*) bring the actor's root as close as possible to a particular point in space (which it has computed itself as a point from which it is able to reach the target) sometime during its execution.

When all motion models in the script have made all such requests, we can perform a numerical optimization over the free parameters of the motion models to best satisfy all requests, *i.e.* generate a least-squares-error solution. Assuming that we will solve the optimization problem using techniques similar to those used in our IK algorithm, described in chapter 4, all that remains is to determine how to express the invariants as differentiable functions of the motion model parameters, and what form of objective function to use in order to guide the solution. We must state that this material has not been implemented, as it was considered ancillary to the main contributions of the thesis.

Invariant values depend directly on IK handles, which in turn depend on the pose of an actor at some given time, which in turn depends on the actor's motion, which depends on application of the motion model rules to the parameters. In theory, therefore, we can express an invariant's value as a composition of many functions that bottoms out at the motion model parameters. In practical use, this would require instrumenting the entire clip library and the motion model synthesis rules for computing chain-rule derivatives. Since motion model rules can currently contain arbitrary code (but might not in the future), this prospect is not appealing.

An alternative is to manually construct simplified expressions for the invariants in terms of a motion model's current motion and parameter set. For instance, there is a nearly linear relationship between the root's Y position during flight and the *height* parameter, and an equally simple linear expression for the root's X and Z positions in terms of the *takeoff-spot* and *landing-spot*. Such expressions are sure to be less accurate than those of the preceding paragraph, and creating such expressions may seem odious and overburdening of the motion model designer. In truth, however, we foresee this optimization as being used mainly for gross optimizations of locomotion trajectories in order to simplify or enable layered motions' goals. In such cases, the position and possibly orientation of the root are generally the only handles participating in the optimization.

A suitable objective function for the optimization will incorporate two terms: the first, to eliminate the effects of differences in scale between parameters, can be constructed from similar principals to those used to construct the mass matrix from chapter 4. The second term expresses how willing a motion model is to alter each of its parameters, based on their current values. Each motion model knows the legal ranges of each of its parameters, and when their values go beyond the boundaries, they will make it more difficult to move them further.

9.3.6 Specializing & Combining Motion Models

We designed the framework in this thesis to encapsulate and reuse the work of the master animator. However, interesting possibilities exist for enabling the end-user to do the same, at a different level. The first is enabling the user to create new performance styles by tweaking existing ones. We can enable this extension simply by guiding the user through a process of applying similar tweaks to each of the base motions for a particular style, and then associating the tweaked motions with a newly created style. It may also be possible, in some circumstances, to automatically apply the tweaks the user has added to a *particular* instantiation of a motion model to *all* of the motion model's base motions.

Another method by which the user could encapsulate his own work is to create a new type of motion model out of combinations of existing motion model types. For example, the user may animate a "preening" behavior in which the actor *lean*'s over a counter to *peer* at a mirror, *stroking* his hair, and baring his teeth (with a *bare-teeth* motion model). The user could then group all of these motion models together to form a new "primitive" behavior encapsulated by a *preen* motion model. The *preen* motion model would possess the union of the parameters and invariants of each of its constituent motion models, as well as the schematic of how the component motion models should be combined. The first pass at enabling this ability would utilize a generic "combination" class of motion model, which, when instantiated, itself instantiates copies of all its constituent motion models, and creates its own clip-composite in which to assemble the composite animation to export to the rough-cut.

Bibliography

[Anjyo 1992] Ken-ichi Anjyo and Yoshiaki Usami and Tsuneya Kurihara. *A simple method for extracting the natural beauty of hair*, Computer Graphics (Proceedings of SIGGRAPH 92), *26 (2)*, pp. 111-120 (July 1992, Chicago, Illinois). Edited by Edwin E. Catmull.

[Auslander 1995] Joel Auslander and Alex Fukunaga and Hadi Partovi and Jon Christensen and Lloyd Hsu and Peter Reiss and Andrew Shuman and Joe Marks and J. Thomas Ngo. *Further Experiences with Controller-Based Automatic Motion Synthesis for Articulated Figures*, ACM Transactions on Graphics, *14(4)*, pp. 311-336 (October 1995).

[Badler 1987] Norman I. Badler and Kamran H. Manoochehri and Graham Walters. *Articulated figure positioning by multiple constraints*, IEEE Computer Graphics & Applications, *7 (6)*, pp. 28-38 (June 1987).

[Baraff 1989] David Baraff. *Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies*. In Computer Graphics (SIGGRAPH 89 Proceedings), volume 23, pages 223–232, July 1989. (p. 24).

[Baraff 1991] David Baraff. *Coping with friction for non-penetrating rigid body simulation*. In Computer Graphics (SIGGRAPH 91 Proceedings), volume 25, pages 31–40, July 1991. (p. 24).

[Baraff 1992] David Baraff and Andrew Witkin. *Dynamic simulation of non-penetrating flexible bodies*, Computer Graphics (Proceedings of SIGGRAPH 92), *26 (2)*, pp. 303-308 (July 1992, Chicago, Illinois). Edited by Edwin E. Catmull.

[Baraff 1995] David Baraff. Course notes from course: *Physically Based Modeling*, organizer Andrew Witkin, SIGGRAPH 95, Computer Graphics Course Notes, Annual Conference Series (July 1995, Los Angeles, California).

[Baraff 1998] David Baraff and Andrew Witkin. *Large Steps in Cloth Simulation*. In Michael Cohen, editor, Computer Graphics (SIGGRAPH 98 Proceedings), pages 43–54, July 1998. (p. 24).

[Bates 1994] Joseph Bates. *The Role of Emotion in Believable Agents*, Communications of the ACM, Volume 37 , Issue 7 (1994), pp 122-125, (July 1994).

[Bates 2000] Joseph Bates. Personal communication.

[Blumberg 1995] Bruce M. Blumberg and Tinsley A. Galyean. *Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 47-54 (August 1995, Los Angeles, California). Addison-Wesley. Edited by Robert Cook.

[Bruderlin 1989] Armin Bruderlin and Tom Calvert. *Goal Directed, Dynamic Animation of Human Walking*. Proceedings of SIGGRAPH 89, Computer Graphics Proceedings, Annual Conference Series, pp. 233-242 (August 1989).

[Bruderlin 1994] Armin Bruderlin and Chor Guan Teo and Tom Calvert. *Procedural Movement for Articulated Figure Animation*. In Computers and Graphics, volume 18, #4, pp. 453-461. Elsevier Science Ltd., 1994.

[Bruderlin 1995] Armin Bruderlin and Lance Williams. *Motion Signal Processing*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 97-104 (August 1995). Addison-Wesley. Edited by Robert Cook.

[Catmull 1997] Ed Catmull. Personal communication.

[Cohen 1992] Michael F. Cohen. *Interactive spacetime control for animation*. In Computer Graphics (SIGGRAPH 92 Proceedings), volume 26, pages 293–302, July 1992. (p. 25).

[De Bonet 1997] Jeremy S. De Bonet. *Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Images*, Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, pp. 361-368 (August 1997, Los Angeles, California). Addison-Wesley. Edited by Turner Whitted.

[DeRose 1998] Tony DeRose, Michael Kass, and Tien Truong. *Subdivision Surfaces in Character Animation*. In Michael Cohen, editor, Computer Graphics (SIGGRAPH 98 Proceedings), pages 85–94, July 1998. (p. 24).

[Foley 1992] James Foley and Andries van Dam and Steven Feiner and John Hughes. *Computer Graphics: Principles and Practice*. 2nd Edition. Addison-Wesley Publishing Company.

[Gill 1981] Philip E. Gill and Walter Murray and Margaret H. Wright. *Practical Optimization*. 1981. Academic Press.

[Girard 1987] Michael Girard. *Interactive Design of 3D Computer-Animated Legged Animal Motion*. In IEEE Computer Graphics and Applications, June 1987. Pages 39-50.

[Gleicher 1992] Michael Gleicher and Andrew Witkin. *Through-the-lens camera control*, Computer Graphics (Proceedings of SIGGRAPH 92), *26 (2)*, pp. 331-340 (July **1992**, Chicago, Illinois). Edited by Edwin E. Catmull.

[Gleicher 1997] Michael Gleicher. *Motion Editing with Spacetime Constraints*, 1997 Symposium on Interactive 3D Graphics, pp. 139-148 (April 1997). ACM SIGGRAPH. Edited by Michael Cohen and David Zeltzer.

[Gleicher 1998] Michael Gleicher. *Retargeting Motion to New Characters*, Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, pp. 33-42 (July 1998, Orlando, Florida). Addison-Wesley. Edited by Michael Cohen.

[Grassia 1998] F. Sebastian Grassia. *Practical Parameterization of Rotations Using the Exponential Map.* Journal of graphics tools. A K Peters, Ltd. Volume 3, Number 3, 1998. pp 29-48.

[Grassia 2000] F. Sebastian Grassia. *Motion Editing: Mathematical Foundations*, Course notes from course 26, *Motion Editing: Principles, Practice, and Promise*, organizer Mike Gleicher, SIGGRAPH 2000, Computer Graphics Course Notes, Annual Conference Series (July 2000, New Orleans, Louisiana).

[Gritz 1997] Larry Gritz and James K. Hahn. *Genetic Programming Evolution of Controllers for 3-D Character Animation*, Genetic Programming 1997: Proceedings of the Second Annual Conference, pp. 139-146 (July 1997, Stanford University, CA, USA). Morgan Kaufmann.

[Grzeszczuk 1995] Radek Grzeszczuk and Demetri Terzopoulos. *Automated Learning of Muscle-Actuated Locomotion Through Control Abstraction*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 63-70 (August 1995, Los Angeles, California). Addison-Wesley. Edited by Robert Cook.

[Hash 1999] Martin Hash and others. *Martin Hash's Animation Master: Reference Guide*. Hash, Inc. Vancouver, WA. 1999.

[Hodgins 1995] Jessica K. Hodgins and Wayne L. Wooten and David C. Brogan and James F. O'Brien. *Animating Human Athletics*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 71-78 (August 1995, Los Angeles, California). Addison-Wesley. Edited by Robert Cook.

[Hodgins 1997] Jessica K. Hodgins and James F. O'Brien and Jack Tumblin. *Do Geometric Models Affect Judgments of Human Motion?*, Graphics Interface '97, pp. 17-25 (May 1997). Canadian Human-Computer Communications Society. Edited by Wayne A. Davis and Marilyn Mantei and R. Victor Klassen.

[Isaacs 1987] Paul M. Isaacs and Michael F. Cohen. *Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics*, Computer Graphics (Proceedings of SIGGRAPH 87), *21 (4)*, pp. 215-224 (July 1987, Anaheim, California). Edited by Maureen C. Stone.

[Keene 1989] Sonya E. Keene with Dan Gerson. Object-oriented programming in Common LISP : a programmer's guide to CLOS. Addison-Wesley. 1989.

[Ko 1993] Hyeongseok Ko and Norman I. Badler. *Straight line walking animation based on kinematic generalization that preserves the original characteristics*, Graphics Interface '93, pp. 9-16 (May 1993, Toronto, Ontario, Canada). Canadian Information Processing Society.

[Koga 1994] Yoshihito Koga and Koichi Kondo and James Kuffner and Jean-Claude Latombe. *Planning Motions with Intentions*, Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series, pp. 395-408 (July 1994, Orlando, Florida). ACM Press. Edited by Andrew Glassner.

[Kurlander 1996] David Kurlander and Tim Skelly and David Salesin. *Comic Chat*, Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series, pp. 225-236 (August 1996, New Orleans, Louisiana). Addison-Wesley. Edited by Holly Rushmeier.

[Lasseter 1987] John Lasseter. *Principles of Traditional Animation Applied to 3D Computer Animation*, Computer Graphics (Proceedings of SIGGRAPH 87), *21(4)*, pp. 35-44 (July 1987, Anaheim, California). Edited by Maureen C. Stone.

[Latombe 1991] J.C. Latombe. *Robot motion planning*. Kluwer Academic Publisheres, 1991.

[Lee 1999] Jehee Lee and Sung Yong Shin. *A Hierarchical Approach to Interactive Motion Editing for Human-Like Figures*, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, pp. 39-48 (August 1999, Los Angeles, California). Addison-Wesley Longman. Edited by Alyn Rockwood.

[Loyall 1997] A. Bryan Loyall. *Believable Agents: Building Interactive Personalities.* Carnegie Mellon University Computer Science Technical report #CMU-CS-97-123. 1997.

[Maciejewski 1990] Anthony A. Maciejewski. *Dealing with the Ill-Conditioned Equations of Motion for Articulated Figures.* IEEE Computer Graphics and Applications. Vol. 10 (3). pp. 63-71. May 1990.

[Maya 1999] Maya user's guide. Alias-Wavefront software. 1999.

[McKenna 1990] Michael McKenna and David Zeltzer. *Dynamic Simulation of Autonomous Legged Locomotion*. Proceedings of SIGGRAPH 90, Computer Graphics Proceedings, Annual Conference Series, pp. 29-38 (August 1990).

[Moore 1988] Matthew Moore and Jane Wilhelms. *Collision Detection and Response for Computer Animation*, Computer Graphics (Proceedings of SIGGRAPH 88), *22 (4)*, pp. 289-298 (August 1988, Atlanta, Georgia). Edited by John Dill.

[Morawetz 1990] Claudia L. Morawetz and Thomas W. Calvert. *Goal-Directed Human Animation of Multiple Movements*, Graphics Interface '90, pp. 60-67 (May 1990). Canadian Information Processing Society.

[Neider 1993] Jackie Neider and Tom Davis and Mason Woo. *OpenGL Programming Guide*. Reading, Mass. Addison-Wesley, 1993.

[Ngo 1993] J. Thomas Ngo and Joe Marks. *Spacetime Constraints Revisited*, Proceedings of SIGGRAPH 93, Computer Graphics Proceedings, Annual Conference Series, pp. 343-350 (August 1993, Anaheim, California). Edited by James T. Kajiya.

[Paige 1982] C. C. Paige and M. A. Saunders. *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM TOMS 8(1), 43-71 (1982).

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

[Perlin 1996] Ken Perlin and Athomas Goldberg. *IMPROV: A System for Scripting Interactive Actors in Virtual Worlds*, Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series, pp. 205-216 (August 1996, New Orleans, Louisiana). Addison-Wesley. Edited by Holly Rushmeier.

[Phillips 1990] Cary B. Phillips and Jianmin Zhao and Norman I. Badler. *Interactive Real-time Articulated Figure Manipulation Using Multiple Kinematic Constraints*, 1990 Symposium on Interactive 3D Graphics, *24 (2)*, pp. 245-250 (March 1990). Edited by Rich Riesenfeld and Carlo Séquin.

[Phillips 1991] Cary B. Phillips and Norman I. Badler. *Interactive behaviors for bipedal articulated figures*, Computer Graphics (Proceedings of SIGGRAPH 91), *25 (4)*, pp. 359-362 (July 1991, Las Vegas, Nevada). Edited by Thomas W. Sederberg.

[Popović 1999a] Zoran Popovic. *Motion Transformation by Physically Based Spacetime Optimization*, Doctoral thesis. Carnegie Mellon University Department of Computer Science. Technical Report number CMU-CS-99-106. June, 1999.

[Popović 1999b] Zoran Popović and Andrew Witkin. *Physically Based Motion Transformation*, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, pp. 11-20 (August 1999, Los Angeles, California). Addison-Wesley Longman. Edited by Alyn Rockwood.

[Press 1993] William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery. *Numerical Recipes in C: the art of scientific computing*. 2nd Edition, Cambridge Press, 1993.

[Rose 1996] C. Rose, B. Guenter, B. Bodenheimer, and M. Cohen. *Efficient Generation of Motion Transitions using Spacetime Constraints*. In Computer Graphics (SIGGRAPH 96 Proceedings), pages 147–154, 1996. (p. 25).

[Rose 1998] Charles Rose and Michael F. Cohen and Bobby Bodenheimer. *Verbs and Adverbs: Multidimensional Motion Interpolation*, IEEE Computer Graphics & Applications, *18(5)*, pp. 32-40 (September - October 1998).

[Sengers 1998] Phoebe Sengers. *Anti-boxology: agent design in cultural context*. Doctoral thesis. Carnegie Mellon University Department of Computer Science. Technical Report number CMU-CS-98-151. August, 1998.

[Shoemake 1985] Ken Shoemake. *Animating Rotation with Quaternion Curves*, Computer Graphics (Proceedings of SIGGRAPH 85), *19 (3)*, pp. 245-254 (July 1985, San Francisco, California). Edited by B. A. Barsky.

[Shoemake 1994] Ken Shoemake. *Arcball Rotation Control*, Graphics Gems IV, pp. 175-192 (**1994**, Boston). Academic Press. Edited by Paul S. Heckbert.

[Sims 1994] Karl Sims. *Evolving Virtual Creatures*, Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series, pp. 15-22 (July 1994, Orlando, Florida). ACM Press. Edited by Andrew Glassner.

[Stroustrup 1994] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley. Reading, MA. 1994.

[Thomas 1981] Frank Thomas and Ollie Johnston. *The Illusion of Life*. Hyperion publishers, New York. 1981.

[Unuma 1995] Munetoshi Unuma and Ken Anjyo and Ryozo Takeuchi. *Fourier Principles for Emotion-based Human Figure Animation*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 91-96 (August 1995). Addison-Wesley. Edited by Robert Cook.

[Van Gelder 1997] Allen Van Gelder and Jane Wilhelms. *An Interactive Fur Modeling Technique*, Graphics Interface '97, pp. 181-188 (May **1997**). Canadian Human-Computer Communications Society. Edited by Wayne A. Davis and Marilyn Mantei and R. Victor Klassen.

[Welman 1993] Chris Welman. *Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation.* Master's Thesis, Simon Frasier University, 1993. ftp://fas.sfu.ca/pub/cs/theses/1993/ChrisWelmanMSc.ps.gz .

[Wernecke 1994] Josie Wernecke. *The Inventor Mentor*. Reading, Mass. Addison-Wesley, 1994.

[Wilhelms 1997] Jane Wilhelms and Allen Van Gelder. *Anatomically Based Modeling*, Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, pp. 173-180 (August **1997**, Los Angeles, California). Addison-Wesley. Edited by Turner Whitted.

[Wiley 1997] Wiley, Doug and Hahn, James, *Interpolation Synthesis of Articulated Figure Motion*, IEEE Computer Graphic and Applications, November/December 1997, Volume 17, No. 6, pp. 39-45.

[Witkin 1987] Andrew Witkin and Kurt Fleischer and Alan Barr. *Energy Constraints on Parameterized Models*, Computer Graphics (Proceedings of SIGGRAPH 87), *21 (4)*, pp. 225-232 (July 1987, Anaheim, California). Edited by Maureen C. Stone.

[Witkin 1988] Andrew Witkin and Michael Kass. *Spacetime Constraints*, Computer Graphics (Proceedings of SIGGRAPH 88), *22 (4)*, pp. 159-168 (August 1988, Atlanta, Georgia). Edited by John Dill.

[Witkin 1990] Andrew Witkin and Michael Gleicher and William Welch. *Interactive Dynamics*, 1990 Symposium on Interactive 3D Graphics, *24 (2)*, pp. 11-21 (March 1990). Edited by Rich Riesenfeld and Carlo Séquin.

[Witkin 1995] Andrew Witkin and Zoran Popović. *Motion Warping*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 105-108 (August 1995, Los Angeles, California). Addison-Wesley. Edited by Robert Cook.

[Zhu 1995] David Zhu, Craig Becker, Yotto Koga, James Kuffner, and Jean-Claude Latombe. http://www.motionfactory.com

A *Clip* Library Specifications

In this and the following appendix, we relate detailed structure about the main data structures pertaining to the thesis. Although we do not present any actual code or pseudocode, we do not hide the fact that the data structures are written in C++.

A.1 Generic Clip Class

This is the structure and functionality common to all members of the clip family. The reader may note that there is no “re-execute” or “re-evaluate” functionality discussed. To avoid unimportant complexity, we decided to implement clips as single evaluation objects. That is, once the initial parameters for a clip are specified, there is generally no established means of changing the parameters. Therefore, each time a motion model executes, it first destroys its previous clip expression, then constructs a new one from its most primitive elements (*i.e.* base motions).

A.1.1 Data Elements

- `const char * modelClass;` Each clip is associated with a specific character model class, which defines the meaning of the motion curves the clip contains, and how they can be applied to a character model in order to make it move. In particular, the model class defines how many motion curves the clip will contain, and the names of the bodyparts to which they will be applied in a character model, and the set of bodypart-groups into which the motion curves are collected. Each clip stores the name of the character model class with which it is associated in this element.

- `class BitVector groups;` A `BitVector` is a simple packed bit-array class that allows its elements to be set, tested, and combined with other `BitVector`'s via boolean operations. The clip data element `groups` reflects whether each bodypart-group is actually present in the clip. That is, does the clip believe it should provide motion curves for the bodyparts in the group?

A.1.2 Functional Elements

- `virtual void GetMotionCurveValue(int jointIndex, double time, double *val, int *belowLevs=NULL);` This is the main function of all clips: returning the value of a particular joint/bodypart at a given time. The mapping of character model to clip defines a mapping of joints to integer identifiers; `jointIndex` specifies the joint whose value in the clip at time `time` the clip should copy into `val`. `belowLevs` is passed to the operands of the clip in most cases (we will describe exceptions below for clip-mix and clip-composite.

- `double GetMotionCurveVelocity(int jointIndex, double time, V3 *vel=NULL, int *belowLevs=NULL);` Use finite differences to evaluate the angular velocity of the joint, rather than its value. We use this to compute the final and beginning velocities of clips between which we wish to create transitions using Hermite interpolation.

- `virtual bool FullComplement(double time);` Returns true if and only if every member of `groups` is present at time `time`.

- `virtual void LoadModelFromClip(double time, CharacterModel *model);` Cause the clip to evaluate all of its motion curves at `time`, and place those values it defines (according to its `groups` element) into the specified character model `model`.

- `virtual const class BitVector &DefinedGroups(double time);` Returns a copy of `groups`, evaluated at `time`. The function of many clips depends on the makeup of their operand clips. A clip-blend, for instance, takes different actions depending on whether one or both of its overlapping operand clips defines a value for a particular bodypart-group. A clip, therefore, typically invokes this method on its operand clips during the process of determining its own value.

- `virtual void TimeInterval(double &start, double &end);` Compute and return the timespan, in animation time, over which the clip is defined. For all clip classes except clip-primitive, this involves, at some point, querying the clip operands.
- `ClipAnim(ClipAnim *operand);`
  `ClipAnim(CharacterModel *model);`
  `ClipAnim(int numCurves, int numGroups, const char *mClass);` As we described in section A-1.1, each clip is defined for a specific class of character model. However, different types of clips are created under different circumstances with different kinds of information. Subclasses can provide any of the three above sets of data to define, in the parent clip class, the character model class.

A.2 Clip-Primitive

A clip-primitive is the only leaf-node type in the motion expressions we build. It expresses motions read from files as clips, and can also store dynamically computed animations (built up one frame at a time).

A.2.1 Data Elements

- `class Differentiable **curves;` A `Differentiable` is our most generic class representing a function of time that is differentiable with respect to some vector of state. We make clip-primitives very general by allowing their motion curves to be arbitrary `Differentiable`'s. The element `curves` is the primary data for a clip-primitive, consisting of an array of size {number of joints/bodyparts of character model for which clip-primitive is defined} of pointers to `Differentiable.`
- `int numFrames;` The number of frames of data contained in the motion curves.

A.2.2 Functional Elements

- `ClipPrimitive(CharacterModel *model, Differentiable *globPos, Differentiable **jointRots, Bodypart **corrJoints);` To create a clip-primitive from existing motion curves, as we do when parsing a style from a datafile, we use this constructor. `globPos` contains the motion curve for `model` 's global translation, and `jointRots` is an array of motion curves whose size (but not ordering) must be that of the joints/bodyparts of `model`. `corrJoints` specifies, for each element of `jointRots`, the corresponding bodypart in `model`, so that the clip-primitive can arrange the elements of `jointRots` in the correct order.

- `ClipPrimitive(CharacterModel *model, int startTime, int numFrames);` Recalling that we also use clip-primitive to store the final, IK-filtered version of the animation for rapid playback, we can create a "blank" clip-primitive for the character model class of `model` using this constructor.

- `void TakeSnapshot(int frameTime, CharacterModel *model);` If the clip-primitive was created using the second constructor above, we can use this method to stuff the pose currently held by `model` into the motion curves at time `frameTime`.

A.3 Clip-Time-Warp

Clip-time-warp implements time warping as a direct mapping from warped times to original times.

A.3.1 Data Elements

- `Clip *warped;` The clip whose time domain is being warped.

- `double *oTimes, *wTimes;` These are two arrays of the same size, represeting the time-warp keys. `wTimes`[*i*] defines the animation time that will be mapped into `warped` 's domain at time `oTimes`[*i*].

- `struct Tans {double l,r;} *tans;` We actually compute the time-warp as a direct mapping of `wTimes` into `oTimes`, which we accomplish with a Hermite spline. `tans` stores the tangent values corresponding to each key in `wTimes` and `oTimes`.

A.3.2 Functional Elements

- `ClipTimewarp(Clip *toWarp);` We always create a clip-time-warp to modify an existing clip. The newly created clip-time-warp contains no time-warp keys, and thus does not deform the time domain at all.

- `void AddTimewarpKey(double origTime, double warpedTime);` We define a time-warp by adding mapping keys to it. After each new key is added, we recompute the entire mapping function, *i.e.* rebuild the Hermite spline whose value knots are `wTimes` and `oTimes`, and whose tangents are `tans`. We do so while ensuring that time does not flow backwards, as we described in section 4.3.2.2.

- `double OrigTimeFromWarped(double warpedTime);` This is the most frequently used operation of a clip-time-warp, and simply involves evaluating the Hermite spline that defines the mapping from warped time into original time. Clip-time-warp's implementation of `GetMotionCurveValue` simply invokes `GetMotionCurveValue` on `toWarp` with the provided time mapped through `OrigTimeFromWarped`. If `warpedTime` falls to the left or right of the extreme keys defined for the time-warp, the time-warp at the relevant extreme key is applied.

- `double WarpedTimeFromOrig(double origTime);` The inverse operation of `OrigTimeFromWarped`, this requires that we invert the spline mapping, which involves cubic root finding.

A.4 Clip-Space-Warp

Clip-space-warp implements motion warping for both 3D position curves and quaternion curves. One non-standard addition we developed is that when successive warp keys are very close together (our threshold is less than 3 frames apart), we perform linear interpolation of the displacement curve between the two keys (equivalent to setting the interior tangent lengths to zero). Such closely-spaced keys are legitimate in our system, but catmull-rom interpolation generally produces wild gyrations on such a small time scale.

A.4.1 Data Elements

- `Clip *warped;` Like clip-time-warp, clip-space-warp, is defined by an existing clip that it will deform.
- `class Spacewarp **warps;` Each motion curve of the underlying clip is warped independently. This element is an array of `Spacewarp` objects, each of which space-warps a single orientation, position, or scalar function. Position curves use 3D Hermite splines to encode the displacement, which is added to the base curve. Orientation curves (quaternion valued) encode the displacement as a spherical bezier curve, which is applied to the base curve by quaternion multiplication, as we described in section 4.3.2.1.

A.4.2 Functional Elements

- `ClipSpacewarp(Clip *toWarp, BitVector *onlyTheseGroups);` A newly created space-warp can (by default) warp all motion curves of `toWarp`, or only those bodypart-groups specified in `onlyTheseGroups`.
- `void SetWarpPose(double time, V3 &trans, Quat rotations[]);` We can specify a warp key at a specific time by providing the values each joint should possess in the warp.
- `void SetWarpPose(double time, CharacterModel *model);` We can also specify a full-body warp by instructing the space-warp to read the current joint values directly from an appropriate character model.

- `class Spacewarp &GetSpacewarp(int jointIndex);` If we wish to place warp keys only at selected joints, we can access the individual `Spacewarp`'s directly.

- `void SetZeroWarp(double time);` When we wish to limit the effect of a particular warp key, it is often useful to specify "zeros," keys before and after the key of interest that specify zero displacement.

- `void RemoveAllWarpsAtTime(double time);`

- `void RemoveAllWarps(void);` Resets the entire space-warp

A.5 Clip-Pivot

The clip-pivot correctly rotates an entire motion about a fixed point, accounting for the correlation between global position and orientation that space-warping, for example, would ignore.

A.5.1 Data Elements

- `Clip *clip;` The clip to be pivoted.

- `V3 pivotPt, offset, newPos;` The element `pivotPt` is the fixed point in the space `clip` inhabits, `offset` represents the displacement to add to the fixed point after rotating, and `newPos` is the value of the global translation computed by `ComputePivot` below.

- `Quat pivot, newRot;` The element `pivot` is the rotation to be applied to `clip`, and `newRot` is the value of the global rotation computed by `ComputePivot` below.

A.5.2 Functional Elements

- `ClipPivot(Clip *toPivot, V3 &pivotPt, Quat pivotRot, V3 &finalPos);` Define a pivot to be applied to `toPivot`, where `pivotPt` is the fixed point about which the rotation will occur, `pivotRot` is the pivot rotation, and `finalPos` is the location to which the fixed point should be translated after performing the rotation.

- `void ComputePivot(double time);` Whenever `GetMotionCurveValue` is invoked on a clip-pivot, it executes this method to compute the pivoted values of the global translation and rotation of `clip` (which are the only motion curves affected by the pivot). The global rotation is simply rotated by `pivot`, while the global translation first subtracts `pivotPt`, then rotates by `pivot`, and finally adds on `offset`.

A.6 Clip-Mirror

A clip-mirror allows us to reflect a motion across any of the three coordinate axes. In order to properly account for lateral inversion, it assumes that a character model knows, for each bodypart, what its lateral-reflection counterpart is. For instance, the lateral counterpart of the left shoulder is the right shoulder, while the lateral counterpart of the neck is itself. To actually produce a mirror image of an actor performing a motion, we would need to reflect the actual geometry as well, but here we are interested only in the motion curves.

A.6.1 Data Elements

- `ClipAnim *clip;` The clip to be reflected.
- `int *mirrorParts;` The clip-mirror allocates and computes, in its constructor, the lateral counterpart of each bodypart, and stores its integer label here for quick computation later.
- `CoordAxes mirrorAxis;` The axis along which we reflect.

A.6.2 Functional Elements

- `ClipMirror(class CharacterModel *model, ClipAnim *toMirror, CoordAxes mirrorAxis);` Reflect motion `toMirror` across axis `mirrorAxis` (one of X, Y, or Z), using `model` as a guide for lateral inversion. To reflect a position across, for example, the X axis, we negate its X component. To reflect a quaternion rotation across the X axis, we negate the Y and Z components of the quaternion. We then swap the motion curves of lateral counterparts to complete the reflection.

A.7 Clip-Select

The clip-select makes available only certain specified motion curves from a motion that contains a potentially larger set of motion curves.

A.7.1 Data Elements

- `Clip *clip;` The clip whose motion curves we wish to select
- `BitVector presentCurves;` Specifies which bodypart-groups are selected in `clip`.

A.7.2 Functional Elements

- `ClipSelect(Clip *clip, class BitVector &selectedGroups);` Create a selection of `clip`. All attempted references of motion curves not belonging to bodypart-groups specified in `selectedGroups` will raise an exception.

A.8 Clip-Composite

The clip-composite is the special purpose splicing mechanism that allows us to build up complicated combinations of layered motions. It consists of an array of *layers* for each bodypart-group of the character model type associated with the clip-composite, with layer 0 being the primary or base layer. Each layer consists of a single sequence of motion *segments* for a bodypart-group. When the clip-composite must evaluate itself at a given time, it finds the *highest* level in each bodypart-group's array that contains a segment defined at the evaluation time, and queries the clip comprising that segment for motion curves. This makes sense for the primary purpose of the clip-composite, which is to allow layering: successively layered motions over-shadow the motions upon which they are layered.

Recall from chapter 6 that layered motions can be blended with the underlying motion, and note that the underlying motion is generally the entire layer array in the rough-cut clip-composite at the time the layered motion is evaluated. This means that the rough-cut can be an operand to a clip-mix or clip-blend that will itself eventually be added into the rough-cut. If we naively allowed such a cyclic expression, an evaluation of the rough-cut would result in an infinite loop. This explains the need for `belowLevs` in the parameter set to `Clip::GetMotionCurveValue`, as it specifies a set of levels (one for each bodypart-group) below which in the layer array we should look in a clip-composite to evaluate it at the given time. Thus when the rough-cut is an operand to a clip-mix, we provide the clip-mix with a `belowLevs` that contains the index (for each bodypart-group) of the current top of the layer array. Thus, no matter how many further layers are added to the rough-cut, when the clip-mix accesses it as an operand with `belowLevs`, it will reference the same layers, regardless of whether the clip-mix itself subsequently appears in higher levels of the layer array.

A.8.1 Data Elements

- ```
  struct Segment {
      struct Transition {
          double        blendStart, blendPhase2;
          Clip          *glue;
          class Hermite *blend;
      } begin, resume;
      double         start, end, srcOffset;
      Clip           *clip;
      Quat           *valCache;
      int            fJoint, lJoint, layerLevel;
  ```
  ```

```
   struct Segment *next, *prev;
};
```

Clip-composite defines the internal class `Segment`, whose elements closely mirror the parameters to `AddSegment` (explained below).

- ```
  struct GI {int start, end, topLayer, maxLayers;
             Segment **layers, *currPos;
             double  currTime; };
  GI         *segments;
  ```

  This element is the layer array we discussed at the beginning of this section.

A.8.2 Functional Elements

- ```
  int AddSegment(int DOFGroup, ClipAnim *clip, double srcStartTime,
                 double dstStartTime, double duration, ClipAnim *glue,
                 class Hermite *blend, double blendStartTime,
                 double endBlendPhase1, ClipAnim *base,
                 int baseLayerLevel, bool layer);
  ```

  Adds part or all of the bodypart-group `DOFGroup` in `clip` as a segment in the clip-composite, relative to some clip `base` that has already been added to the clip-composite, in level `baseLayerLevel`. The portion of `clip` to be added is defined by `srcStartTime` and `duration`, and will be placed in the layer array starting at time `dstStartTime`. The new segment is supplied with a transition as well, in the form of `glue`, a clip (generally a clip-transition) that sits in front of the new segment, starting at `blendStartTime`, blending between `base` and itself until `endBlendPhase1`, and then blending from itself to `clip` according to the scalar Hermite blending function `blend`. If `layer` is `true`, then `clip` will be layered off of `base`, and be placed into a new level in the layer array. Otherwise, it is considered to segue from `base`, and is placed in the same level as `base`.

- ```
  void AddResumption(int DOFGroup, int layerLevel,
                     double layerStartTime, ClipAnim *glue,
                     Hermite *blend, double blendStartTime,
                     double endBlendPhase1, double resumeTime);
  ```

  Add a resumption to an existing segment in the layer array for a given bodypart-group `DOFGroup`. The segment is identified by its level in the layer array (`layerLevel`) and its start time (`layerStartTime`). The remainder of the parameters simply define a transition from the segment at time `blendStartTime` to whatever segment is below it (or above it if there is none below it) at time `resumeTime`.

- `void RemoveAllSegments(void);` Completely clears the clip-composite of all segments for all bodypart-groups.
  ```

- `void TopLayerList(int *topLayers);` Stores in `topLayers` the indices of the top-most layers currently residing in each bodypart-group's layer array. This is how we get the necessary information to blend the rough-cut into a clip-mix.

A.9 Clip-Mix

Clip-mix allows us to mix exactly two motions, which are presumed to be defined over the same time range. The two motions are blended together, with different weightings for each bodypart-group. As discussed in chapter 6.2.3, the weight can specify whether the second motion should be blended with or differentially added to the first motion.

A.9.1 Data Elements

- `struct MixGroup {int start, end; double mix; };` Simply stores the indices of the first and last motion curves associated with a bodypart-group, along with the mix weighting for that group.
- `Clip *clip[2];` The two clips being mixed.
- `int *bLev[2];` The beneathLevels values for the two clips.
- `MixGroup *mix;` The mix weightings, as described above.

A.9.2 Functional Elements

- ```
  ClipMix(CharacterModel *model, Clip *clip1, Clip *clip2,
          double *mixGroups, double startTime, int *beneathLev1,
          int *beneathLev2);
  ```
  Defines a mix between `clip1` and `clip2`, in which `mixGroups` contains the bodypart-group weighting factors. `beneathLev1` and `beneathLev2` are as described above in section A.8.

A.10 Clip-Blend

The clip-blend is simply a scaled-down version of clip-composite (or rather, clip-composite is a scaled-up version of clip-blend) that allows no layering. At most two segments can be overlapped at any time, and blending according to a single Hermite curve interval is applied between the two segments over the duration of their overlap. Blending is performed over the entire defined set of motion curves. This much simpler version suffices for tasks such as smoothing out stitched-together cycles by overlapping and blending their end/beginning portions, and for time-varying blends between two motions.
  ```

B Motion Model Specifications

The motion model class (`MotionModel`) contains many elements and methods. As the purpose of this appendix is to give a detailed flavor of what is in a motion model, we have decided to show most of the detail, but only explain the most relevant parts. To show less would be hiding too much, and to explain more would muddle understanding of the overall design. The details we do not show pertain only to the processes of writing and reading descriptions of motion models from files, and supporting direct manipulation of motion model parameter values, neither of which is central to the main ideas of the thesis.

B.2 Data Elements

- ```
  struct Basis {
     Basis(int numTimeParams);
     virtual ~Basis();
     virtual void Release(void);

     MMStyle::BaseMotion  *baseMotion;
     class ClipTimewarp   *timewarp;
     class ClipBlend      *splice;
     bool                  loopable;
     double               *ssTimeVals; // ss = "single shot"
   };
  ```
  ```

The `Basis` is the internal data structure for a base motion. When the style of a motion model is set via `SetStyle`, the motion model queries the relevant `MMStyle` to retrieve a `MMStyle::BaseMotion` for each base motion, and builds a `Basis` for each. The `Basis` acts as a wrapper for the raw base motion, providing the time-warp that we use to temporally align base motions for blending, and storing the timing information of the motion.

- ```
  enum ParamType {SCALAR, POSITION, ORIENTATION, ENUM, BOOLEAN, HANDLE,PATH};
  struct Parameter {
     char      *label;   // short name of parameter
     ParamType  type;
     int        dim;           // vector dimension (0 for enumerated type)
     bool       isFree,  // ~(explicitly set)
                dependent,     // value actually depends on some other MM
                active;  // currently has interaction focus?

     union pValue {            // the actual value of the parameter
         V3 v3; Quat quat; double scalar; int enumerated; bool boolean;
         class ControlPtBox *handle; class V3Curve *path;
     } val;

     union editData {    // data pertaining to the interaction method
         struct {
            class Constraint *handle, // the actual manipulation constraint
                             *pivot;  // provides positional anchor value
            bool              editRigidly;
            bool              useGenericController;
         } dManip;
         float    bounds[3];
     } editData;
     int     editTime;   // time at which interaction occurs
     struct objs *activeBut, *resetBut;

     Parameter();
     ~Parameter();
  };
  ```

The `Parameter` is the data structure used to define and maintain the main motion model parameters. The bulk of a `Parameter` is devoted to enabling graphical manipulation of the parameter. The actual value of the parameter is contained in the union element `val`. Each specific motion model class defines its own parameters, but registers them at the generic `MotionModel` level so that detailed knowledge is not required to manipulate or examine them.

- ```
  struct InvariantVal {
     class Actor::Invariant *inv;
     struct ValInterval {
         union  Val { double scalar; V3 v3; Quat quat; } val;
         double cStart, cEnd;
         bool   set, openLeft, openRight;
     } *values;
     int   numValues, numUsed, priority;
     bool              active;
     MotionModel      *takenFrom;

     InvariantVal(class Actor::Invariant *invar, int numIntervals);
     ~InvariantVal(void);
  ```

```
   void CreateInterval(int indx, double start, double end,
                       bool openLeft, bool openRight);
 };
```

The actual constraints associated with invariants exist independently of any motion model, maintained by the `Actor` the motion model is animating. Beyond the act of registering its interest in the invariant with the `Actor`, the `MotionModel` maintains a set of "values" for the invariant. That is, for each interval the invariant is active during the motion model's duration, an `InvariantVal` stores, the temporal specification of the interval and the value the invariant should possess. When a user tweaks an invariant's value, the tweaked value is stored in the appropriate interval value, and it is noted that the interval has been explicitly set.

- ```
  struct RemapData { double curr, mapTo; struct RemapData *next; };
  struct TweakList {
     class Constraint      *tweak;
     RemapData             *timePoints;
     struct TweakList      *next;

     TweakList(class Constraint *con);
  };
  ```

  All other tweaks (*i.e.* manipulations of bodyparts not controlled by invariants) are attached to a motion model and stored as `TweakLists`. The tweak consists of a geometric `Constraint` that contains the desired value(s) of the tweak, and a record of the application times of the tweaks (*i.e.* key-times) in the motion model's canonical time so that the tweak can adjust with the motion model as its parameters change.

- ```
  class Director    *director;
  CharacterModel    *model;
  class MMStyle     *mmStyle;
  ```

  The `director` is the maintainer of the entire animation, from which the motion model receives certain services, such as computation of IK poses. The `model` is the character for which the motion model is generating animation, and `mmStyle` is the currently selected style for the motion model.

- ```
  class MotionModel *predecessor, *successor;
  ```

  These are the direct predecessor and successor of the motion model in the script, related to this motion model through transitions. For a layered motion, predecessor is the motion model from which the motion model is layered.

- ```
  InvariantVal     **invariants;
  int                numInvariants;
  ```

  This is the array of all invariants in which the motion model is interested, whether it actually uses them in a particular motion model execution or not.

- ```
  int                startTime, duration, entranceTime,
                     actionTime, predecessorEndTime, beginTransTime;
  ```
  ```

These elements describe the transition of the motion model from its predecessor. `startTime` and `duration` describe the entire motion model; `beginTransTime` is when, in animation time, the transition begins; `entranceTime` is the time in the motion model relative to its own start, at which the transition ends; `actionTime` is the time in animation time at which the transition ends; `predecessorEndTime` is the time at which the predecessor officially terminates.

- **`int pauseDuration, pauseStartAnim;`**
  `double pauseStart;`

  Our implementation currently allows only one pause per motion model, mainly because of interface issues in specifying multiple pauses. These elements describe the start and duration of the pause. The pause is then applied to the motion model by applying a time-warp to its final motion that stretches the single frame beginning at `pauseStartAnim` to fill the entire duration of the pause. Therefore, there is technically some movement during the pause, but it is generally not noticeable.

- **`int *currTimes;`**

  This array of animation times contains the final values of the motion model's key-times, after the motion model has completely generated its motion and situated it in the rough-cut.

- **`TweakList *tweaks, *tweakTail;`**

  A list containing all of the tweaks relevant to this motion model.

- **`class ClipTransition *transGenerator, *resumeGenerator;`**

  Each motion model must contain separate transition generators for its (main) transition and its potential resumption, since the generator itself is a clip that functions as the "glue" for the motion when added into the rough-cut (as described in section A.8.2).

- **`class Clip *currMotion;`**

  The final motion produced by the motion model, which gets added into the rough-cut.

- **`class BitVector groups;`**

  This element specifies which bodypart-groups of `currMotion` are actually valid and contribute to the rough-cut. This will be less than the entire body for layered motions.

- **`class BitVector **keyWarpGroups;`**

  Before performing the final IK constraint satisfaction on the rough-cut, we place a space-warp over the entire rough-cut, as we described in chapter 7.3. Each motion model places warp keys at each of its key-times. Depending on the action, it may be inappropriate to place keys on all bodyparts. For instance, the *check-watch* motion model should not place keys on the actor's legs. The motion model stores this information in `keyWarpGroups`, an array of `BitVector's` that specifies, for each key-time, the bodypart-groups that should participate in the warp.

- **`Parameter **allParameters;`**

`int                numParameters;`

The array of all motion model parameters that we mentioned at the beginning of this section.

B.2 Functional Elements

- **`virtual void SetStyle(class MMStyle *style);`**

  This function sets the style of a motion model. Every style the system knows about will, when actually used in a session, be read from a file and stored in a `MMStyle`. In this function, the motion model first abolishes its association with its current style, releasing its `Basis` structures. It then queries the style for each of its base motions, creating `Basis` structures for each. Finally, it digests any other annotations the style provides, such as the timing of footplants, explicit values for parameters, or added invariants.

- **`void AddToAnimation(MotionModel *toFollow, bool layered);`**

  When a new motion model is created or read from a file and has been defined, its first action is to insert itself into the script as controlled by the director. Motion models are always inserted into the script relative to an existing motion model (unless it is the first action taken). ToFollow is the motion model that the new motion model will follow, and layered specifies whether the relationship is a layering or segue. This function uses the two following functions to insert the motion model into the script.

- ```
  virtual void ProceedFrom(class MotionModel *preceedingMotion,
                           bool layered);
  ```

  This is a script-level operation that causes the motion model to execute relative to `preceedingMotion`. If `layered` is `true`, then the motion model will be layered off of `preceedingMotion`, otherwise it will segue from it. Any motion models transitioning from the motion model will still do so. If `layered` is `false` and any motion model currently segues from `preceedingMotion`, it will henceforth segue from the last motion model in the sequence currently rooted at this motion model.

- ```
  virtual void Precede(class MotionModel *followingMotion,
                       bool layered);
  ```

  We maintain the entire script for an actor in the links between motion models. Therefore, when a motion model inserts itself after a motion model in the script, it must also inform that motion model it has done so. This is accomplished by invoking `Precede` on the predecessor with itself as the argument.

- ```
  virtual void AddToRoughCut(class ClipComposite *roughCut,
                             MMList *motionsBelow);
  ```

  When the director issues a re-execution of the animation, each actor calls this function on every motion model in its script, in the order dictated in chapter 7. Four things occur in this function:
  ```

1. The motion model completely recomputes its own motion, using `RecomputeMotion`, described below. This process includes computing its entrance transition, and activating all of its invariants.

2. For each bodypart-group over which the motion model actually has control, we add to `roughCut` both the just-computed motion and the transition.

3. If the motion model is part of a layered sequence (but is not the first motion in the sequence), then for each bodypart-group its predecessor controlled but the motion model itself does not control, we cause the predecessor to compute and insert a resumption into the `roughCut`.

4. Update the timing of all tweaks (using `RemapTweaks`, below), and export them into the animation.

- ```
  virtual void AddWarpKeys(class ClipComposite *roughCut,
                           class ClipSpacewarp *edits);
  ```

  Once all of the motion models have executed and added their motions and transitions to the `roughCut`, the actor makes another pass through the motion models, invoking this function on each (order of execution is `not` important), which causes them to insert space-warp keys into `edits` at each key-time and at the activation and deactivation of each invariant and tweak. It is then `edits` upon which we run per-frame IK, as described in chapter 7. We cannot do this earlier while executing `AddToRoughCut` because to compute the pose that forms the basis of the space-warp, all of the invariants and tweaks that will affect the pose must be activated, and we cannot guarantee this until all motion models have completed `AddToRoughCut`.

- ```
  virtual void AddConstraint(class Constraint *cons,
                             class ClipSpacewarp *warps);
  ```

  The active operator for all tweaks is a constraint. When a new tweak is created interactively or read from a file, this function adds it to a motion model's list of known tweaks. `AddConstraint` also re-computes the actor's pose subject to the new tweak, and inserts the pose as a space-warp key in warps.

- ```
  virtual bool ConstraintAlteration(class Constraint *cons,
                                    class Bodypart *bod, int time,
                                    class ClipSpacewarp *warps);
  ```

  After the user has made an interactive modification to a tweak (either changing its value or removing it altogether), this function causes the motion model to update its own information about the tweak.

- ```
  class Clip *CurrentMotion(void);
  ```

  This function returns the motion last computed by the motion model. It does *not* cause a re-execution of the motion model, which only occurs in AddToRoughCut.

- ```
  virtual void GetTimeInterval(int &startTime, int &duration);
  ```

  Determine and return the entire time interval over which the motion model is active, from the moment its transition begins, to the moment its successor's transition begins. This is used by actors to deter-
  ```

mine which motion model lie underneath other motion models in the motion stack (important for determining the feasibility of layering).

- **`int CanonicalTimeToAnimTime(double canon);`**

  Using the last-computed values of the motion model's key times, convert a time in the motion model's canonical time to animation time.

- **`double AnimTimeToCanonicalTime(int anim);`**

  The reverse.

- **`void SetBaseLayer(MotionModel *bl);`**

  All motions in a sequence beginning with a layered motion are themselves also layered. However, they do not, at first, know this, since each motion models is aware only of its immediate predecessor and successor, as well as any motions layered off of it. When a layered motion executes, one of the first things it does is invoke SetBaseLayer on its successor (if any), which informs the successor that it is, indeed, layered.

- **`void RemapTweaks(void);`**

  When a tweak is created, its application time is translated into the motion model's canonical time and stored as such. This allows the tweak to maintain its time of application *relative to* the timing of the motion model, regardless of how much the motion model's timing changes. `RemapTweaks` causes all tweaks to recompute their animation application times from their canonical application times and the current values of the motion model's key-times.

- ```
  void ComputeTransition(class ClipComposite *prevMotion,
                         int beginT, int goalT,
                         ClipTransition::TimingMethod timeMeth,
                         ClipTransition::TimingResolution resMeth);
  ```

  While recomputing its motion, a motion model must account for transitioning from its predecessor motion. `ComputeTransition` invokes the motion model's `ClipTransition` to create the transition. It acquires the beginning pose for the transition from `prevMotion` (an alias for the rough-cut). `beginT` and `goalT` specify the times in the motion model's motion at which we can find the beginning and ending poses for the prototype transition. `timeMeth` instructs the transition generator on which method to use for computing the transition's duration (fixed, fastest, or proportional, as described in chapter 6), while `resMeth` instructs it on how to resolve the durations computed for each bodypart-group into a single value (mean, fastest-legal, slowest). The computation is performed for only those bodypart-groups the motion model actually controls.

- ```
  virtual void ComputeResumption(class ClipComposite *resuming,
                                 class BitVector *groups, int beginT);
  ```

Functions much like `ComputeTransition`, but uses the motion model's resumption generator to generate a resumption transition. The `groups` parameter is computed by and passed from the motion model's successor's `AddToRoughCut`.

- **`virtual void RecalculateMotion(class ClipComposite *roughCut, MMList *motionsBelow);`**

  This is the function inside of which most of the computation discussed in chapter 5 occurs. This includes releasing storage for the "old" version of the motion, calculating default values for any unspecified motion model parameters, transforming the base motions to satisfy the parameters' goals, activating the motion model's invariants, and computing the motion model's transition. The two parameters for this function are necessary for layering. `roughCut` is necessary for determining the actor's pose, since no single motion model may be entirely in control of the actor. `motionsBelow` is a list of motion models passed to `GatherLayeredResources`, described below.

- **`void GatherLayeredResources(const int *desiredDGs, MMList *below, int *mixFormula);`**

  The first thing a layered motion must do when re-executing is determine whether it can actually acquire the resources it needs to execute. To do so, it must have a list of all of the motion models that lie beneath it in the motion stack, which the actor computes, and is passed to `GatherLayeredResources` as below. If all of these motion models are willing to relinquish or share control of the bodypart-groups the layered motion model requires (as expressed in `desiredDGs`) and relinquish any requisite invariants, then the layered motion model can execute, and the blending factors for the layered motion curves are returned in `mixFormula`.

- **`ClipAnim *MixLayeredMotion(Clip *curr, ClipComposite *roughCut, int *mix);`**

  Once a layered motion model's blending factors have been computed in `GatherLayeredResources,` we compute the actual blended composite in `MixLayeredMotion` by creating a clip-mix with the motion model's current motion as one operand, and the rough-cut as the other. It is at this point that we compute the `belowLevs` discussed in section A.8.