PARAMETER SPACES, SPACETIME CONTROL AND MOTION GRAPHS FOR AUTOMATING THE ANIMATION OF VIDEOGAME CHARACTERS

by

Lorne McIntosh
B.Sc., Simon Fraser University, 2007

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

in the
School of Interactive Arts and Technology
Faculty of Communication, Art and Technology

SIMON FRASER UNIVERSITY
Fall 2011

APPROVAL

Name: Lorne McIntosh

Degree: Master of Science

Title of Thesis: Parameter Spaces, Spacetime Control and Motion Graphs for Automating the Animation of Videogame Characters

Examining Committee:

Chair:

Dr. Halil Erhan
Assistant Professor,
School of Interactive Arts and Technology

Steve DiPaola
Senior Supervisor
Associate Professor,
School of Interactive Arts and Technology

Dr. Philippe Pasquier
Supervisor
Assistant Professor,
School of Interactive Arts and Technology

Dr. Thomas Calvert
External Examiner
Professor Emeritus,
School of Interactive Arts and Technology

Date Approved:

Abstract

Character animations are a crucial part of many interactive applications, from training simulations to videogames. As these applications have become more sophisticated, the growing number of character animations required has made standard animation techniques like key-framing and motion-capture increasingly expensive and time-consuming. Procedurally generating animations appears to offer a solution. This thesis extends and combines work from several areas of procedural animation to create an end-to-end system for the automatic generation of character animations for interactive applications. Specifically, our architecture pairs Spacetime Control, used to automatically generate new physically-valid clips of character animation, with a data-driven playback technique, used to automatically generate continuous streams of character motion from these clips in real-time. Our approach exploits the natural parameterization present in videogames and character motion to organize and automate the procedural generation of large quantities of character animation. It also supports rapid-prototyping, easily handles animation design changes, and may potentially be operated from start to finish by a single user. We demonstrate this architecture with a working implementation and show results from an example scenario starring a humanoid character capable of dozens of generated motions including standing, walking, running, turning and stepping.

Keywords: procedural animation; character animation; videogames; parameter spaces; physically-based; spacetime control; optimal control; nonlinear optimization; motion graphs

Acknowledgments

I am extremely grateful to have had the opportunity to work with my senior advisor Steve DiPaola. Steve is a passionate and knowledgeable teacher, an inspiring researcher, and a patient and open mentor who gave me the "globally-optimal" balance of guidance and freedom throughout my studies at SIAT. I am also very grateful to my advisor Philippe Pasquier, without whose initial enthusiasm and encouragement I may never have tackled the complex subject of Spacetime Control.

Many thanks go to my parents Terence and Margaret for their constant love and encouragement (not to mention all the years of free room and board). Special thanks go to my brother Tom for all of his welcome distractions, impromptu guitar-solos and beer-runs etc. Also, thanks to my friends Colin Mingus, Andrew Hawryshkewich and others for breaking me out of my thesis-cave occasionally for some much-needed hiking and videogames.

I would also like to thank Carl Laird and Andreas Wächter for creating IPOPT [WB06]—the open source nonlinear optimization routine that out-performed every commercial package I tried. I'm also grateful to Alex Champandard, for graciously providing me with the AISandbox [Cha11] source code and the excellent Motion Graph implementation contained therein. Thanks to Kevin Lake and BlendSwap.com for providing the character model of "Chip" the mannequin.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

From their humble beginnings, videogames have seen an immense growth in popularity, and today are considered a mainstream form of entertainment, rivalling movies and music. Along with this increased popularity has come an astounding increase in sophistication and complexity. In the 1970s, popular games like Pong [Alc72] wowed audiences with their blocky 2D graphics and beeping sound effects. Animated characters were rare. By contrast, today's games are set in detailed and expansive 3D worlds, and are populated with hundreds of life-like characters. Clearly, the bar has been raised.

1.1.1 State of the Art

Assets are the digital building-blocks that make up interactive productions like videogames. They include characters, props, scenery, animations, sound effects, music, and dialog. To deliver a high-fidelity experience, modern 3D videogames typically require many thousands of such assets in very high quality. Among these, the character assets are arguably the most important in games with characters—especially in those starring a player-controlled character. In such games, the player-character at a minimum—and possibly many other characters—are on-screen throughout virtually the entire playtime. This places a great deal of importance on the assets that determine how these characters sound, look, and move. In this thesis we concentrate our discussion on the latter of these—the way in which characters move.

Modern 3D character animation systems for interactive media like videogames typically operate by playing back canned sequences or "clips" of character animation data. For instance, a walk-cycle is an example of a single sequence or clip of character animation. Each animation clip is typically created individually, using one of two techniques: *keyframing* or *motion-capture*.

Keyframing

In the case of the former, animations are created manually by an animator using the methods offered by an animation software package—chiefly keyframing. Historically, this is based on traditional hand-drawn animation where a senior animator would draw the main poses or "keys" and an assistant animator would create the in-between drawings between the key drawings. In modern 3D computer animation, the process is much the same. An animator specifies the positions and rotations of a character's body parts at several essential (key) moments in time. For instance, to create an animation of a character kicking something, the animator might first create a keyframe with the character's legs together, a second with the character's right leg rotated back, and finally the last keyframe with his right leg rotated forwards. These keyframes are then interpolated by the computer to produce all the "in-between" frames necessary for smooth-looking motion. Creating animations in this way typically requires artistic ability, some knowledge of human and animal movement, experience with the animation software package being used, and a great deal of practice and patience. The benefits are very precise control over the final animation, and complete freedom to produce any desired motion—including highly stylistic and physically impossible ones.

Motion Capture

In the case of the latter, the animation data is recorded from the performance of a live actor—typically a human actor, though animals have also been used. Various systems have been developed to accomplish this task, but the majority rely on triangulating the actor's movement using data received from many carefully calibrated cameras installed around a small "capture space". The cameras typically track markers worn by the actor in order to allow for easier identification of the position and orientation of each of his body parts in physical space, though markerless systems are now emerging. Motion capture is what drives

most examples of truly convincing character animation today.

Despite its immense popularity in the videogame industry, and its potential for incredibly realistic results, motion capture technology has numerous disadvantages:

- Special equipment, software and trained personnel are necessary and can be prohibitively expensive—especially for individuals and small studios.
- Captured animations must generally be processed or cleaned up, often manually, to make them suitable for use in an interactive production.
- Only motions that can actually be performed inside the capture-space can be recorded. This generally excludes the possibility of capturing dangerous stunts, uncooperative animals, imaginary creatures, motions that defy the laws of physics, or motions that require more room than the capture-space allows for.
- Making substantial changes to a motion-captured animation is difficult. In general they must be performed and captured again, or painstakingly edited by an animator.

Playback Processing

Regardless of the way in which these sequences or clips of character animation are produced, they must somehow be played back at run-time. For either of these techniques, keyframing or motion capture, there are a variety of ways in which the animation data may be stored, grouped into sequences or clips, and then processed for real-time playback. For instance, with keyframed skeletal animation, the rotational data at the joints on keyframes could be saved in a lookup-table per sequence or clip. To play a clip back, the real-time game logic would decide which clip is necessary and which frame should be used, and then read the corresponding frame data from the lookup-table. The game's scene graph would then be modified to reflect the character's new pose on this frame—possibly with an interpolation being performed before, if it was not a keyframe. Additionally, there could be a "skinning" or enveloping process to deform the character's rendered mesh to match the skeleton's pose in real-time. This is just one of many possible playback processes for keyframing and motion capture, which typically has as its goals: speed, reuse and making efficient use of the graphics hardware API. Regardless of the specific technique used, non-procedural animation, such as keyframing and motion capture, has the character animation data pre-made (i.e. canned) into sequences which then use one of these playback techniques at run-time. In this thesis,

we refer to these sequences as animation clips or just "clips". We also use the term animation clip to refer to a grouped sequence of animation data, or an animation sequence, even when it has been procedurally generated.

1.1.2 Problems in Paradise

It was once computational power that limited the scope of interactive productions like videogames. Commodity processors operated too slowly to render truly convincing 3D graphics, sound-effects, and music together in real-time. With ever-increasing transistor-densities and clock-speeds, and with the advent of highly-parallel dedicated graphics processing units (GPUs) though, that bottleneck has largely been cleared. Indeed, modern videogames may now render scenes that compete well with the artistic and technical prowess of the film industry. Instead, the limiting factor today appears to be man-power for creating assets.

The ever-increasing quantity and quality of the character assets required for populating modern videogames are making the conventional character animation techniques and playback processing techniques inadequate. It is our view that conventional character animation systems for videogames suffer from several major problems:

- They are time-consuming to use, as everything must be done laboriously by programmers, and animators or motion-capture actors, all who need extensive training.
- They require extensive up-front planning and foresight. Because of the tight interconnectedness of character animations, an iterative design process—adding new animations as they are realized to be necessary—does not work well. This makes rapid-prototyping difficult.
- They require complete reconstructions of the entire animation database for seemingly small changes, (such as the addition of a single requirement like, "the character now carries a sword in his right hand"). Every animation must be edited manually by an animator to meet the new requirement, or motion captured again from scratch.
- They are nearly impossible for any single user to operate as they require the expertise of both programmers and animators. Tight coordination between the two is critical, but difficult to achieve.

1.1.3 The Case for Procedural Content

Procedural asset generation appears to offer a solution to many of the problems plaguing conventional asset creation techniques. Rather than relying on human designers to manually specify the details of every asset, procedural systems take an algorithmic approach, exploiting the computational power of the computer to produce assets in a more automated fashion according to rules and parameters. Human designers work with such procedural systems at a higher level, specifying the rules and parameters that govern asset-creation rather than the assets themselves.

1.2 Contributions

This thesis presents an architecture for the procedural generation and playback of 3D character animations for interactive productions like videogames. We combine work from the field of Spacetime Control (used to automatically generate new physically-valid clips of character animation) with work from the field of data-driven playback techniques (used to automatically generate continuous streams of character motion from these clips in real-time). Our major contribution is a demonstration that this combination of techniques can produce an effective end-to-end system for the procedural animation of videogame characters. Additionally, we present a novel parameterized approach to organizing the animation generation process that is particularly amenable to character animation for videogames. Lastly, our Spacetime Control formulation considers the creation of cyclic (i.e. looping) animations in a manner that is more robust than has been done previously.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 performs a literature review of relevant techniques in the area of procedural character animation. Chapter 3 presents the design of our character animation system architecture. Chapter 4 describes the details of our particular implementation of this architecture. Chapter 5 presents an example character animation scenario, demonstrates the use of our approach in solving it, and discusses the results. Finally Chapter 6 draws some conclusions, and suggests several areas for future research.

Chapter 2

Related Work

As with most difficult problems in computer graphics and artificial intelligence, procedural character animation has been approached from many different perspectives. The following sections provide an overview of the approaches that are most relevant to this thesis—specifically *physically-based techniques*, *data-driven techniques* and *hybrid techniques* that combine aspects of both.

2.1 Physically-Based Techniques

To animate a physically-embodied character, it could be argued that the most logical approach is to model and simulate the physics of the character's body. This approach ensures that the motions produced are at least physically-plausible; basic laws such as gravitational attraction and conservation of energy will be obeyed. In general, this lends a degree of realism—though not necessarily intelligence or grace—to the resulting animation.

Within physically-based techniques, there are two main approaches that differ in the way physical laws and control are applied: *controlled forward-dynamics* and *Spacetime Control*.

2.1.1 Controlled Forward-Dynamics

The forward-dynamics of a character's body are relatively trivial to simulate, and in fact this is commonly done in many modern videogames to produce a "ragdoll" effect for unconscious characters. At each discrete time step t, the rigid-body equations of motion are integrated to produce a new state for the character at $t + 1$. Without any means to influence this

simulation, the character will naturally crumple to the floor under the effect of gravity, appearing lifeless and limp.

Synthesizing motion for conscious characters in this way however is vastly more difficult. Conscious characters must appear to sense their environment, think about and plan their actions, and ultimately use their virtual muscles to move in a natural-looking way. This necessitates the creation of a motion-controller—a "brain" for the character that will complete a sensory-motor feedback loop. The creation of such motion-controllers represents a difficult problem though, and has been an active area of research for decades.

Many researchers have created motion-controllers manually using standard techniques from robotics like state-machines, proportional-derivative control, trajectory tracking, inverted pendulum models, and inverse kinematics. Controllers for a wide variety of typical human motions such as walking, running, jumping and crouching etc. have been created in this way [LvdPE96, CBvdP10]. Controllers for more acrobatic motions such as vaulting, tumbling, and diving etc. have also been produced [HWBO95, Woo98]. Despite these successes, engineering new controllers with these techniques remains difficult and typically requires careful design and a deep understanding of the desired motions. Additionally, though these motion-controllers may be robust, the animations they produce tend to be somewhat stiff and robotic-looking, and there's often no obvious way of addressing this in the controller design.

To overcome these problems, some researchers have applied techniques from Artificial Intelligence to problem of controller design. Many have applied Genetic Algorithms to evolve Neural Network-based motion-controllers [RM01, Gut04, VM08, CBOP09, AF09]. This biologically-inspired approach promises a high degree of automation, natural-looking results and real-time controller performance. As Genetic Algorithms and Neural Networks continue to improve, it may yet become the dominant method in procedural character animation. For now though, the process of evolving Neural Network controllers generally falls short of expectations. Fitness functions are notoriously difficult to specify, and even relatively simple behaviours like walking are slow and difficult to evolve. For instance, the bipedal walking controllers of Allen and Faloutsos were only stable "for 5-10 meters before toppling" [AF09]. [1]

[1] Our own attempts to evolve a bipedal walking controller were even less successful. The evolutionary process eventually discovered a physical glitch that, when properly exploited, would launch characters 20m through the air. With the discovery of flight, walking quickly became an antiquated mode of transportation.

As an aside, it is worth noting that in the realm of videogames, controlled forward-dynamics techniques like these are somewhat unlikely to be adopted in the short-term. Game designers generally prefer to have a high-degree of reliability and predictability in the way their characters will move—especially the player-character. With controlled forward-dynamics techniques, there is always the possibility of a controller failure. Though entirely realistic, it could make for a rather frustrating game-play experience if the hero occasionally tripped or stumbled at inopportune moments.

Additionally, despite some success [FPT01], composing multiple controllers together to create complex motion sequences remains an open and difficult problem.

2.1.2 Spacetime Control

A different approach to applying physical-laws to procedural character animation is known as Spacetime Control. This approach is alternatively called by the names Spacetime Constraints and Optimal Control, but in this thesis we will use the term Spacetime Control. Originally from the field of robotics, it was adapted and introduced to the computer graphics community by Witkin and Kass [WK88]. In Spacetime Control, the motion of a character is specified at a high-level with constraints (for instance, "Avoid all obstacles and never use more than 200 N of thrust"), and an objective function (for instance, "Maximize the total distance traveled around the track"). The system uses these specifications to automatically find motion trajectories that minimize or maximize the objective function whilst satisfying the constraints. Many researchers have successfully created novel character animations from scratch using Spacetime Control. For instance, Fang and Pollard created gymnastically-themed animations for a humanoid character [FP03], and Wampler and Popović created walk and run-cycle animations for characters with a variety of body-types [WP09]. In this thesis, we use Spacetime Control to automatically generate animations in parameterized sets for our characters.

Despite these successes, Spacetime Control is a very computationally expensive technique, which generally makes it inappropriate for use in interactive applications like videogames. We work around this limitation by combining Spacetime Control with a data-driven technique for real-time playback.

2.2 Data-Driven Techniques

There are many character animation techniques which require the input of pre-made animation clips. We refer to these as data-driven organizational and playback techniques. Data-driven techniques process a set of input animations to provide new animations not in the original dataset, or useful methods of playing the animations back.

2.2.1 Move Trees

Move Trees [Kin98, MBC01] (which are, in fact, directed graph structures despite their name) represent the conventional videogame industry approach to using clips of character animation data for real-time playback. In these graphs, each node represents a unique clip of character animation, and the edges between nodes describe which clips may follow each other. Transitions between clips are generally only allowed at the ends of animation clips. The animation clips, the structure of the Move Tree itself and the game logic used to navigate through the Move Tree at run-time are all carefully planned before development begins and then created manually. This reliance on manual effort makes large Move Trees expensive to produce, and also somewhat fragile. Adding, changing or removing animation clips is often difficult because of the tight interconnectedness of the animations.

2.2.2 Motion Graphs

Numerous researchers have investigated the possibility of automatically building directed graph structures like Move Trees, and automatically searching them to find useful motion sequences [AF02, LCR*02, KGP02]. This automatic approach has been termed Motion Graphs.

Motion Graphs were introduced by Kovar et al. [KGP02], and provide an automatic method of producing continuous streams of character motion in real-time, given a database of animation clips as input. The Motion Graph itself is a directed-graph structure where each edge represents a clip of character animation, and each node serves as a transition point connecting these clips. Continuous streams of character motion may be produced by simply walking the graph, playing the clips of animation data encountered along each edge.

Arikan et al. [AFO03] present an approach which provides functionality essentially similar to that of a Motion Graph, wherein pre-made animation clips are re-combined as necessary. Their approach however allows for the combination of multiple animation clips

simultaneously—for instance, "jump and catch while running". Though reasonably efficient, the approach does not yet work in real-time, and is thus unsuitable for use in interactive productions.

Zhao and Safonova [ZS09] create Motion Graphs with better connectivity and smoother transitions by generating additional clips via the interpolation of existing ones. In this thesis, we use the key elements of their approach in our Motion Graph implementation.

Lee et al. [LWB*10] create a more flexible derivative of Motion Graphs which they call Motion Fields. The technique is capable of creating motions at run-time that are not explicitly defined in the input animation data.

2.3 Hybrid Techniques

Many techniques, including ours, take a hybrid approach to character animation, combining aspects of both physically-based and data-driven animation into a single system in order to overcome perceived limitations of one or the other.

Some researchers have successfully applied Spacetime Control to the editing of pre-made or motion-captured animations. For instance, transitions have been created between short segments of motion-captured animation [RGBC96], and motion-captured animation has been adapted to suit different scenarios [MKHK08, PW99].

Liu et al. [LHP05] incorporate a more sophisticated bio-mechanical model into their Spacetime Control formulation, accounting for muscle preferences, spring forces at joints and contact points, and variable joint stiffness. The values for these extra parameters are automatically estimated from analyzing motion-capture data. Different animations in the same style as the original can then be synthesized.

Chapter 3

Architecture

This chapter explains our architecture in general, and provides background information on the techniques used. Chapter 4 provides the details of our particular implementation of this architecture.

3.1 Overview

Our system takes a hybrid approach to character animation, combining aspects of both physically-based and data-driven techniques. Specifically, our architecture pairs Spacetime Control, used to automatically generate new clips of character animation, with a data-driven playback technique, used to automatically generate continuous streams of character motion from these clips in real-time. The major benefits of this architecture are:

- It exploits the natural parameterization present in videogames and character motion to organize and automate the procedural generation of large quantities of character animation.
- It supports rapid-prototyping and sweeping design changes. Animations can be quickly added, changed, and removed en masse.
- It may be operated from start to finish by a single user—potentially even a user with little experience in programming or animation.

Figure 3.1 shows an overview of our architecture. From left to right, the first two blocks represent the physically-based stages of the architecture. In the first block, the characters'

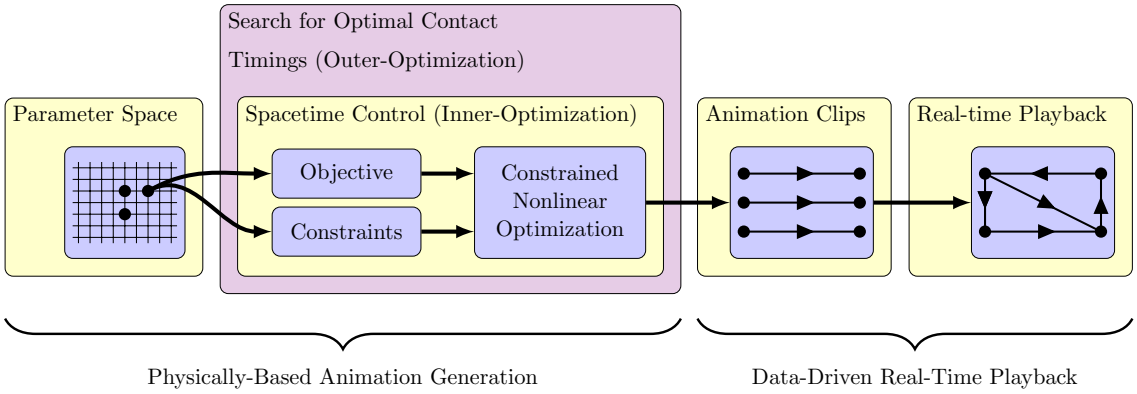


Figure 3.1: Architecture Overview

bodies, their equations of motion, and one or more "Parameter Spaces" are defined by the user. These Parameter Spaces are sampled at discrete intervals to generate a set of constraints and objectives that describe each desired animation.

In the second block, two optimization routines—one nested inside the other—are executed. The outer optimization performs a derivative-free search for the optimal animation length and ground contact timings. The inner optimization is a standard Spacetime Control routine which takes the constraints and objective function for each animation as input, and generates clips of animation data as output. This nested design is similar to that used by Wampler and Popović [WP09].

The last two blocks represent the data-driven stage of our architecture. The output of the optimization routines is simply a (potentially large) database of animation clips. In most respects, these clips are equivalent to those that could be created during a motion-capture session, or animated by an artist using a conventional 3D animation software suite.

In the final block of our architecture, the generated animation clips are used for real-time in-game playback. We avoid specifying any particular playback technique, as we foresee each application will have its own unique requirements in a character animation system, and a 'one size fits all' approach is likely misguided. However, we discuss some techniques that we think are particularily suited to the automated nature of this architecture.

3.2 Spacetime Control

Spacetime Control forms the central animation-generating component of our architecture. It is a powerful method of generating animation, because it requires only a high-level mathematical description of a character's body and its equations of motion, what the character must do (e.g. "Move from point A to point B in 5 seconds"), and how the character should do it (e.g. "Minimize the energy expended"). From these descriptions, physically valid motion is then produced automatically. In the sections that follow, we describe the basic Spacetime Control technique in more detail. Specific details of our implementation of it are discussed in Chapter 4.

3.2.1 Constrained Nonlinear Optimization

In Spacetime Control, the generation of physically-valid character motion is phrased as a constrained nonlinear optimization problem of the form,

$$\begin{aligned} &\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x), \\ &\text{subject to} \quad l \leq \begin{pmatrix} x \\ c(x) \end{pmatrix} \leq u \end{aligned} \tag{3.1}$$

where x is a vector of problem variables, $f(x)$ is a nonlinear objective function to be minimized, $c(x)$ is a vector of nonlinear constraint functions, and l and u are vectors of lower and upper bounds placed on the variables and constraint functions.

Given a constrained optimization problem in the form of (3.1), there are a wide variety of algorithms that may be applied to solve it. In general, these algorithms follow one of two different approaches: Interior-Point methods (an example of which is the program IPOPT [WB06]) and Sequential Quadratic Programming methods (an example of which is the program SNOPT [GMS05]). The internal functioning of these algorithms is a large topic unto itself, and lies outside the scope of this thesis. The interested reader is referred to Bertsekas [Ber99] for a complete treatment of constrained nonlinear optimization techniques. For our purposes though, it is enough to know that $f(x)$ and $c(x)$ must typically be twice continuously differentiable functions, and that either method will return a solution vector for the problem variables x. The solution vector will be a *locally* minimal point with regards to $f(x)$, and will respect the constraints $l \leq x \leq u$ and $l \leq c(x) \leq u$. Note that a *globally*

minimal point is possible, but not guaranteed.

3.2.2 Physical Constraints

To write our character animation problem in the form required by Equation (3.1), the character's body (and the laws of physics that govern its movement) are modeled mathematically, and written as a set of "physical constraints" in the nonlinear optimization problem. Depending on the specifics of the character to be modeled, a wide variety of different modeling techniques may be appropriate. For instance, characters with articulated skeletons (like many bipeds and quadrupeds) might be most efficiently modeled as systems of rigid-bodies connected by powered revolute joints. Other characters that chiefly bend or squish might be better represented using a soft-body approach. Flying and swimming characters may need to be physically modeled in a way that accounts for fluid-dynamics. In this thesis, we are chiefly interested in characters with articulated skeletons and we therefore take the rigid-body approach.

Characters with articulated skeletons are commonly approximated using systems of rigid-bodies connected to each other at specific points (i.e. revolute or spherical joints). For example, see Figure 3.2 which depicts a planar bipedal character, constructed from 5 rigid-bodies, connected by 4 revolute joints. The effects of muscles can be modelled by allowing the character to impart torques on the rigid-bodies across each joint. Joint limits can be modelled by constraining the minimum and maximum angles between the connected bodies. If contacts between the character and the environment are required, they can be modeled as temporary joints—active for just the duration of the contact.

Classical mechanics literature [Gol80] provides several different formalisms that can be used to derive the equations of motion for such a character. Most notable among these are the Newton-Euler, Lagrangian, Hamiltonian and Kane's Method formalisms. The choice of formalism is largely a practical one as they are all essentially equivalent—the laws of physics are faithfully modelled in each. Depending on the specifics of an implementation though, one may be more computationally tractable, easier to use, or otherwise more desirable than another—and we therefore avoid specifying one in our architecture. Section 4.2.2 describes our use of the Newton-Euler formalism to automatically derive the equations of motion for characters in our implementation. For a greatly more thorough treatment of classical mechanics, we refer the interested reader to Goldstein [Gol80].

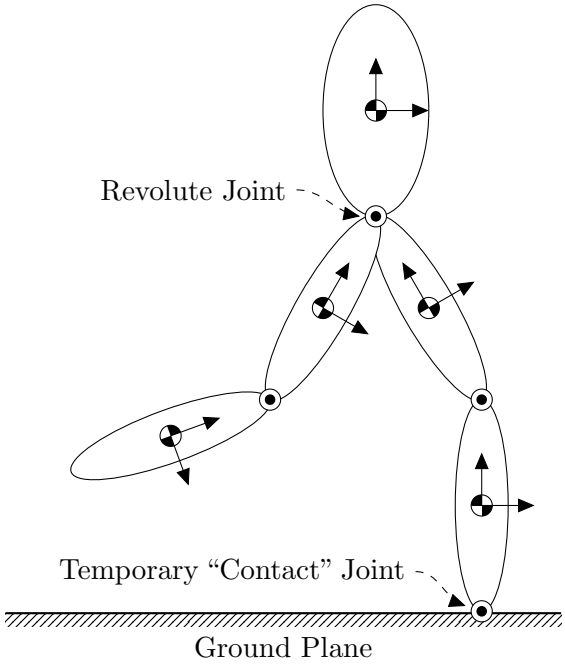


Figure 3.2: A planar biped, constructed from 5 rigid-bodies connected by 4 revolute joints. A 5th revolute joint provides a temporary contact point with the ground plane.

3.2.3 Animation Specifiers

What we have described until this point is nothing more than an elaborate way to generate physically-correct motion. But exactly *what* motion we will get is unknown. Without additional constraints or an objective function to further specify a desired animation, the constrained nonlinear optimization routine is likely to return an undesirable motion. We refer to these types of constraints and objective functions generically as "animation specifiers".

Similar to the physical constraints discussed above, animation constraints are simply mathematical inequalities of the form $l \leq x \leq u$ or $l \leq c(x) \leq u$, and may therefore express virtually anything physical involving the character. As a simple example, the positions and rotations of the character's body segments, or the relative joint angles between body segments could be constrained to particular ranges on particular frames. In this way, a desired animation may be loosely "keyframed" using constraints. The nonlinear optimization routine then does the "inbetweening" to construct physically-valid motion around the specified values.

The Spacetime approach to animation is really much more powerful than this though as it allows complex animation specifications that conventional keyframing does not. For example, it allows writing constraints on forces and torques, as well as on time-derivatives like the velocity and acceleration of various quantities. Constraints may also be written using a "statistical" approach. For example, one could constrain a character to maintain an *average* rotation of 0° over the course of an animation. Additionally, regions of space could be specified for the character to avoid, or to stay within, creating a virtual obstacle course. The possibilities are seemingly endless.

Objective functions provide another means by which to specify what motion will be generated. Rather than specify a hard-constraint, they merely request that a particular function $f(x)$ be minimized. In Spacetime Control, it is a common practice to minimize the sum of squared "muscle" torques exerted by the character, but any function of the problem variables will work. For instance, a character could be given preferences regarding the joint angles of its limbs. A wide range of limb movement would remain possible if necessary, but the character would otherwise attempt to keep limbs at their preferred angles. Smooth, fluid motion could be requested by minimizing the sum of squared joint angle accelerations. Slow, stiff movements could be requested by minimizing the sum of squared joint angle velocities. Again, the possibilities are seemingly endless.

It is often useful to specify multiple simultaneous objective functions. However, an astute reader will note that constrained nonlinear optimization requires just a single objective function. To handle multiple simultaneous objective functions we take the common step of writing our final objective as the weighted-sum of all individual objective functions. Mathematically that is,

$$f(x) = \sum_i (f_i(x) \cdot w_i) \tag{3.2}$$

where f(x) is the final objective function written to the constrained nonlinear optimization problem, $f_i(x)$ is the ith objective function, and w_i is the relative weight of the ith objective function.

In our experience, we have found that most animation specifications can actually be written in either form—as constraints or as objective functions—though typically one is more convenient or more effective than the other.

3.2.4 A Simple Example

In this section we provide a gentle walk-through of the Spacetime Control technique using the simplest possible "character"—the Spacetime Particle introduced by Witkin and Kass [WK88]. An implementation with fully articulated humanoid characters is explored in Chapter 4.

The Spacetime Particle is simply a point-mass that is free to move in a 2-dimensional Euclidean space by applying force to itself with a "jet pack". A gravitational field applies constant acceleration to the particle in one dimension.

To aid in writing the equations for this example, we define some common terms here which will appear in our equations:

Term	Definition	Value used
h	time (in seconds) between each frame	0.05 s
f_n	total number of frames in the animation	21
d_n	number of dimensions in the Euclidean space	2
m	mass (in kilograms) of the particle	2 kg
$q_{f,d}$	particle's position (in meters from origin) at frame f in dimension d	*variable*
$Q_{f,d}$	particle's jet force (in Newtons) at frame f in dimension d	*variable*
Q_{max}	maximum allowed jet force (in Newtons)	30 N
g_d	acceleration (in m/s^2) of gravity in dimension d	$(0, -9.81)$

Table 3.1: Some common terms which will appear in our equations.

$q_{f,d}$ and $Q_{f,d}$ are designated as the variables for the constrained nonlinear optimization routine to determine. The main constraints governing the motion of our character are then derived as follows. We start with Newton's second law of motion,

$$F_{net} = m \cdot a \tag{3.3}$$

where F_{net} is the sum of the forces acting on an object, m is the mass of the object, and a is the acceleration of the object. Ignoring acceleration for now, we can substitute our terms for F_{net} and m into Equation (3.3). In our case, F_{net} is the sum of the jet force $Q_{f,d}$ and the constant force of gravity $m \cdot g_d$ acting on the sphere. m is simply the total mass of the sphere (which we have already defined as m). This substitution yields:

$$Q_{f,d} + m \cdot g_d = m \cdot a \tag{3.4}$$

Turning our attention towards the acceleration term now, we note that acceleration is the second time-derivative of position. We can thus use finite difference formulas to approximate it:

$$\dot{q}_{f,d} = \frac{q_{f,d} - q_{f-1,d}}{h} \tag{3.5}$$

$$\ddot{q}_{f,d} = \frac{q_{f+1,d} - 2q_{f,d} + q_{f-1,d}}{h^2} \tag{3.6}$$

Substituting these relations into Equation (3.4) yields our final $d_n(f_n - 2)$ physical constraints:

$$\begin{gathered} Q_{f,d} + m \cdot g_d = m\frac{q_{f+1,d} - 2q_{f,d} + q_{f-1,d}}{h^2}, \\ 0 < f < f_n - 1, \quad 0 \leq d < d_n \end{gathered} \tag{3.7}$$

With the physical constraints for our character specified, we are simply left with the task of specifying some animation constraints (what the character must do) and an objective function (how the character should do it). In this example, we will have the character begin at the origin $(0, 0)$ with some upwards velocity, and end at another point $(4, 4)$ with the same upwards velocity. To do this, we write some constraints on the character's position on the first two and last two frames of the animation:

$$\begin{aligned} q_0 &= (0, 0) \\ q_1 &= (0, 0.5) \\ q_{f_n-2} &= (4, 3.5) \\ q_{f_n-1} &= (4, 4) \end{aligned} \tag{3.8}$$

Finally, we choose an objective function that will minimize the magnitude of the jet force used by the character over the course of the animation:

$$\begin{gathered} Minimize: \\ \sum_{\substack{0 \leq f < f_n \\ 0 \leq d < d_n}} {Q_{f,d}}^2 \end{gathered} \tag{3.9}$$

When solved with a constrained nonlinear optimization routine, the result is a vector of $q_{f,d}$ and $Q_{f,d}$ values. The $q_{f,d}$ values represent the movement of the character and are what we are chiefly interested in. When rendered appropriately, these values produce an

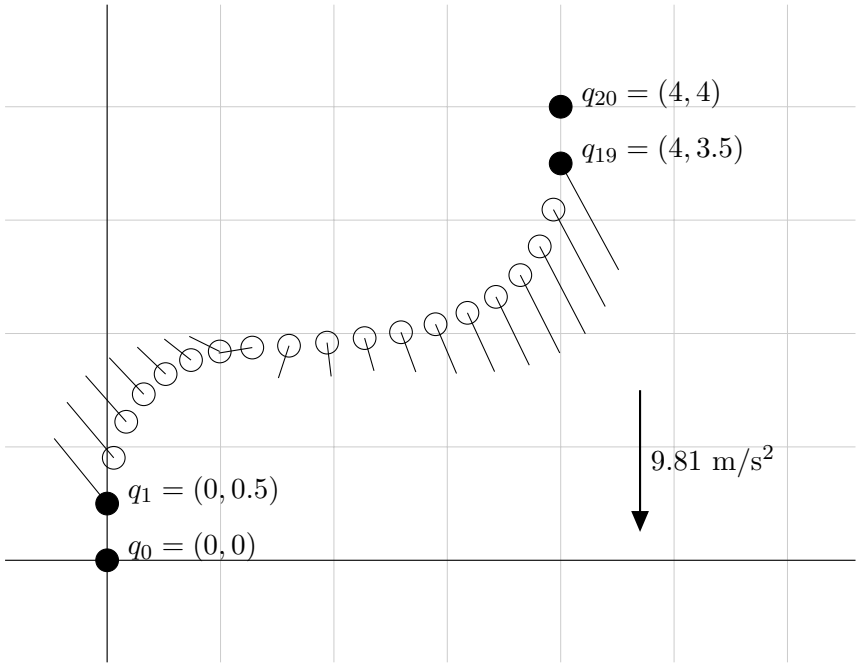


Figure 3.3: A plot showing the resulting Spacetime Particle motion. The particle's positions on the first two and last two frames are fixed by constraints. Jet forces are visualized as vectors drawn at the particle's position.

animation similar to the one depicted in Figure 3.3. The $Q_{f,d}$ values represent the jet forces applied by the character on each frame, and although not strictly necessary for playing the animation, may nonetheless prove useful. For instance, when the animation is used in-game a fiery particle effect could be rendered to visualize the jet. The $Q_{f,d}$ values could be used in order to render this effect with an appropriate direction and magnitude on every frame.

3.2.5 Cyclic Animations

In videogames and other media, the use of cyclic (i.e. looping) animation clips is a common practice. For example, most animators are familiar with the creation of walk-cycles and run-cycles for characters. These cyclic animation clips are especially convenient for game developers because they are relatively short, and yet may be trivially played back to produce endless streams of motion. For this reason, our architecture considers the creation of cyclic animation clips as a core feature.

Our architecture supports the creation of what we term *perfectly cyclic* and *transformed*

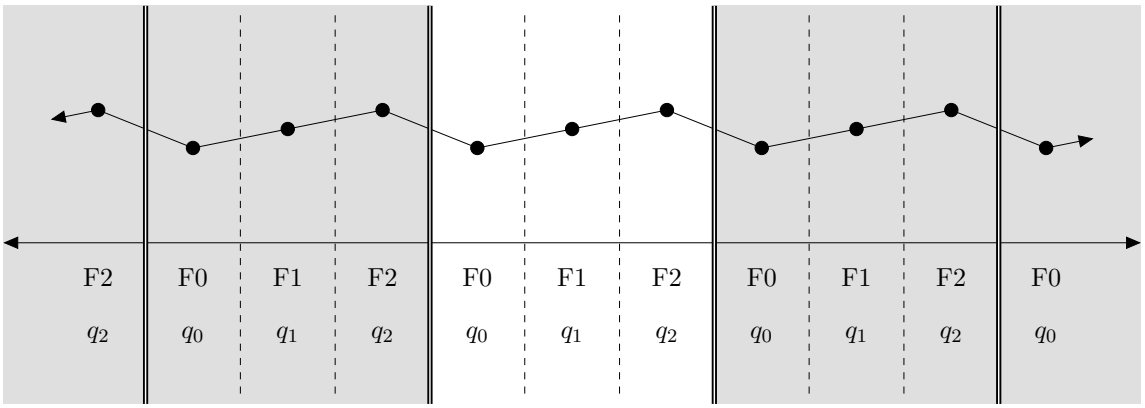


Figure 3.4: A simple "perfectly cyclic" animation with 3 frames.

cyclic animation clips. We define perfectly cyclic to mean that the character starts and ends the clip with the same world transform and velocity. This is achieved by simply applying a Euclidean-definition modulo operator [Bou92] to temporal expressions wherever they appear in the constraints and objectives. The temporal expression is used as the dividend, and the total animation length is used as the divisor. For example, using our Spacetime Particle example from above (see Section 3.2.4), the temporal variable is frame index f, so we could rewrite the equations of motion (3.7) as,

$$Q_{f \bmod f_n,d} + m \cdot g_d = m\frac{q_{f+1 \bmod f_n,d} - 2q_{f \bmod f_n,d} + q_{f-1 \bmod f_n,d}}{h^2} \qquad 0 \le f < f_n, \quad 0 \le d < d_n \tag{3.10}$$

This has the intended effect of looping the time dimension, causing motion to repeat once per animation clip length. Note that there are more equations of motion than before, since we now include equations for the first and last frames. Figure 3.4 shows an example of a perfectly cyclic animation clip which has been "unrolled" several times (the gray areas in the figure) to better show its cyclic behavior. It depicts a dot which rises over 3 frames (F0, F1 and F2), and then quickly returns to its original position as the next cycle begins. The appropriate expression for the dot's position on each frame is written under the frame label.

The "perfectly cyclic" approach is generally only useful for producing animations where the character is stationary overall and may return to its initial state with relative ease. Suitable examples include motions like hand-waving, standing idle, and jumping on the spot. An interesting workaround to this limitation was explored by Wampler and Popović [WP09] who

created walk-cycles and run-cycles for their animals using a "treadmill" approach. In their work, characters remain stationary overall while their feet slide backwards at a prescribed velocity whenever they are in contact with the ground plane. This approach allows the authors to write their physical constraints in the "perfectly cyclic" manner described above, and yet allows for the generation of locomotion animations, where it appears the character is moving. Although convenient for some motions, like constant-velocity locomotion, we have found this approach lacking on two counts:

- The character's momentum is essentially neglected. For constant-velocity locomotion, where a character is simply maintaining his direction and speed, this is largely inconsequential. However, in animations where a character's velocity is changing (i.e. the character is turning, speeding up or slowing down), correct momentum becomes critical to realistic motion. Imagine a human character making a tight turn while sprinting. Realistically, this character should "lean into the turn" in order to remain balanced. However, on a circular treadmill no such lean is required as the character has negligible momentum.

- Specifying an appropriate speed and direction for the treadmill at every moment becomes difficult in complex animation scenarios. For example, in a motion where the character is required to stand idle and then jump forward 1m, the treadmill's speed would need to be accelerated from 0 m/s in the moments preceding the jump, and then decelerated in the moments after landing. Exactly when and how it should accelerate to achieve the desired animation may not be readily apparent.

For these reasons we introduce the *transformed cyclic* approach. This approach allows the creation of motions where the character starts and ends the clip with different world transforms and velocities. Critically though, all of the character's body parts start and end the clip with the same character-space (i.e. relative) transforms and velocities, and thus the clip still appears "perfectly cyclic" in character-space.

To do this, we introduce a spatial transform T and its inverse T^{-1}. The transform may only have translational and rotational components, and it is applied to the character as a whole in world-space. When writing constraints that reference the temporal variable at times after the last frame of the animation clip, T is applied to the value. Similarly, when referencing the temporal variable at times before the first frame of the animation clip, T^{-1} is applied to the value.

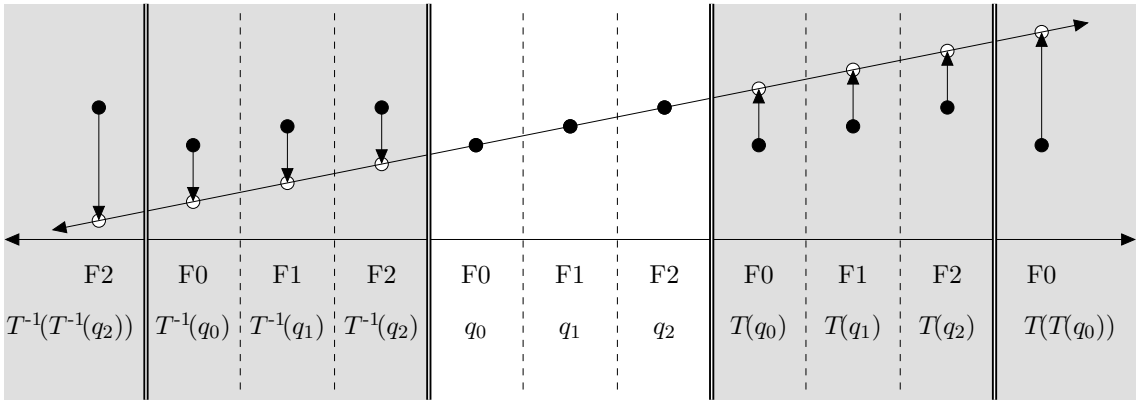


Figure 3.5: A simple "transformed cyclic" animation with 3 frames.

For example, using our Spacetime Particle example from above (see Section 3.2.4), the temporal variable is frame index f, so we could rewrite the equations of motion (3.7) as,

$$Q_{f,d} + m \cdot g_d = m\frac{q_{f+1,d} - 2q_{f,d} + q_{f-1,d}}{h^2},$$
$$0 < f < f_n - 1, \quad 0 \leq d < d_n$$

$$Q_{f,d} + m \cdot g_d = m\frac{q_{f+1,d} - 2q_{f,d} + T^{-1}(q_{f-1 \bmod f_n,d})}{h^2},$$
$$f = 0, \quad 0 \leq d < d_n \tag{3.11}$$

$$Q_{f,d} + m \cdot g_d = m\frac{T(q_{f+1 \bmod f_n,d}) - 2q_{f,d} + q_{f-1,d}}{h^2},$$
$$f = f_n - 1, \quad 0 \leq d < d_n$$

This has the intended effect of looping the time dimension, causing motion to repeat once per animation clip length, with the exception that T is applied. By writing different transformations T, different animations can be achieved. For instance, a character can be made to move forward, by writing T as a translation in the desired direction, or to turn by writing T as a rotation. In each case, the resulting animation will contain a physically-valid motion in which the character moves to achieve the transform over the course of the animation. In the trivial case where T is the identity transform, our transformed cyclic approach degenerates into the perfectly cyclic approach described above.

Figure 3.5 shows an example of a transformed cyclic animation clip which has been "unrolled" several times (the gray areas in the figure) to better show its cyclic behavior. It depicts a dot which rises over 3 frames (F0, F1 and F2), and continues to rise infinitely as subsequent iterations of the loop are played. The appropriate expression for the dot's position on each frame is written under the frame label. In this case, the transformation T simply translates the dot upward, and is represented with an arrow showing how the dot's position is transformed.

When playing back a transformed cyclic animation, the same transform T must be applied to the character's world transform cumulatively each time playback returns to the beginning of the clip. Doing so completes the illusion that the character is moving overall.

3.3 Optimal Contact-Timings

The standard Spacetime Control approach, as we have described it, requires any contacts that a character's body makes with the environment to be explicitly modeled before the nonlinear optimization can proceed. The exact timing of the contact (i.e. over which time steps it occurs), the bodies involved (i.e. what part of the character and what part of the environment contact each other), and the exact nature of the contact (e.g. elastic or inelastic, and the coefficient of friction etc.) must be specified. All of this is necessary because contacts represent discontinuities in the dynamics of the system. They must therefore be modeled explicitly using constraints such that the equations of motion change instantaneously at the moment a contact begins or ends.

As an example, one might create a jumping animation for a bipedal character by specifying that both of the character's feet must be in contact with the ground plane during the first 20 and last 20 frames of an animation (for an illustration of these contact timings, see Figure 3.6). Ground reaction forces would be enabled on these frames to push upwards on the character at the points of contact to achieve this. During the intervening frames, intersection with the ground plane would be forbidden, and ground reaction forces would be disabled. The nonlinear optimization routine would therefore produce a jumping motion to satisfy these constraints.

Manually specifying the appropriate timings for every contact becomes a remarkably difficult task though—especially when more complex animation scenarios are considered. Imagine the difficulty in specifying the contact timings for a galloping quadruped or a

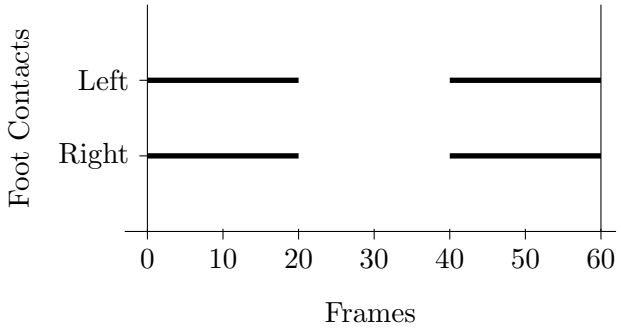


Figure 3.6: An example of contact timings that would produce a jumping motion for a bipedal character. The character would be airborne between frames 20 and 40.

limping biped. Relatively small differences in contact timings may produce significantly different motion, which may spell the difference between extremely natural and extremely unnatural-looking motion.

Rather than leave the contact timings for each Spacetime Control problem to be specified by the user, our architecture prescribes the use of a derivative-free optimization routine to determine suitable values automatically. In our own implementation we use the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [HO96] routine. This optimization is performed as an outer loop around the Spacetime Control optimization in a manner similar to that of Wampler and Popović [WP09]. It attempts to find a set of contact timings that minimize the value of the objective function returned from the inner Spacetime Control optimization. Additionally, the length of the animation itself may be determined by this outer optimization if desired.

3.4 Organizing Animation Generation

An important feature of our architecture is the organization of the animation generation process.

3.4.1 Character Animation with Parameters

Our organizational system stems from acknowledging the often-parameterized nature of videogames and of character motions themselves. Designers typically create games with

parameters—it is only natural in such a highly procedural medium. In games with characters, the various ways in which the characters can move (e.g. walking or flying, fast or slow, healthy or limping) are often central to the game's mechanics, and are therefore parameterized and given much consideration by designers. For instance, in the game Space Invaders [Nis78], the movement speed of the invaders is a parameter that is slowly increased in order to escalate the difficulty over the course of each level. In a modern remake with compelling graphics, each change in speed would require a suitable change in the character's animation, making "movement speed" an example of a parameter that affects character animation in the game. Space Invaders also has several different types of invaders. In our hypothetical remake, we could claim that the invaders with tentacles need different animations than the ones with antennae, making "body type" a second parameter affecting character animation. In general, the more parameters that affect the physical state of a character, the more unique animations are required to properly animate that character under all circumstances.

It is important to note that for animators this represents a full-factorial design problem—a so-called combinatorial explosion. That is, for each possible combination of values across all of the parameters, a unique animation may be required. Mathematically, we could say that the upper limit on the total number of required animations is,

$$i_0 \times i_1 \times \ldots \times i_n \tag{3.12}$$

where i is the number of possible values that the nth parameter can take.

For a more involved example, let us consider a hypothetical 3D action adventure game. This game has 3 characters: a knight, an ogre and a dragon. Each of these characters can move at 3 speeds (walking, jogging and sprinting), turn corners at 7 different rates ($90°/s$ left, $60°/s$ left, $30°/s$ left, $0°/s$, $30°/s$ right, $60°/s$ right, and $90°/s$ right), and carry 1 of 2 different loads (a heavy treasure chest or nothing at all). By Equation (3.12) we can see that this seemingly simple example with just 4 parameters has necessitated the creation of an extraordinary 126 unique animations[1]. Using traditional keyframing or motion-capture animation techniques, creating this quantity of animation would place an enormous burden

[1]In truth, even 126 animations would be considered relatively few by modern videogame standards. For example, the game Assassin's Creed (Ubisoft, 2007) uses over 12,000 animations for the main character alone [Gam07].

on the development team.

In practice, by using some tricks and a little common-sense, it is often possible to get away with creating fewer animations than Equation (3.12) would dictate (which is why we say it is the "upper limit" on the total number of required animations). For instance, if a character has an axis of symmetry, then animations may be mirrored in that axis, eliminating the need for both "left" and "right" versions of every animation. Additionally, if several characters share similar bodies and styles of movement, the same animations may be used for all of them. These same tricks can also be used in our architecture.

3.4.2 Parameter Spaces

Our approach to handling this combinatorial explosion is not to avoid it, but to embrace it. To this end, we introduce the idea of an n-dimensional "Parameter Space", as a natural way of organizing the automatic generation of all the character animations necessary in an application. The number of dimensions n in the Parameter Space is equal to the number of parameters that affect character animation in the final application. Every point in the Parameter Space (defined by an n-length coordinate vector of parameter values) represents an "animation specification" that could potentially be used to generate an actual animation. We use the term "animation specification" since there is, as of yet, no concrete motion data present—the animation exists only abstractly as a vector of parameter values that specify some properties that the generated animation should satisfy.

Parameters, and their corresponding dimensions in a Parameter Space, may be discrete or continuous. Discrete parameters may only hold values from a finite set (e.g. carrying one of: briefcase, backpack, or nothing), while continuous parameters may hold any real value (e.g. movement speed in m/s).

For an example of a Parameter Space, see Figure 3.7 which illustrates a simple 2-dimensional Parameter Space parameterized on Forward speed and Crouching. Forward speed is a continuous parameter measured in m/s, and Crouching is a discrete parameter with just two possible values: On and Off. The bold horizontal lines represent areas that are part of the Parameter Space and may be sampled. Every point in this Parameter Space may be identified uniquely with a coordinate vector of parameter values. For example, the coordinate vector [2.5m/s, On] identifies a point in the Space that specifies the character should move forward at 2.5m/s, in a crouched position.

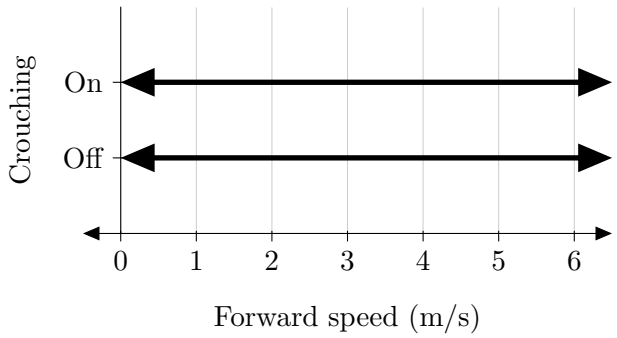


Figure 3.7: An example of a 2D Parameter Space parameterized on Forward speed (continuous) and Crouching (discrete).

From Parameter Spaces to Constraints and Objectives

A Parameter Space is really just a means to an end. To be useful to us, a Parameter Space must be sampled, and animations must be generated to satisfy the vector of parameter values at each sampled point. Parameter values themselves however are not sufficiently descriptive for constrained nonlinear optimization to proceed. The values may hold some particular meaning for the designer who specified them, but to the computer they mean little. For instance, the physical meaning of "Crouching" is indeterminate. We must therefore provide a way of transforming parameter values into the concrete physical constraints and objectives that are necessary for constrained nonlinear optimization. To accomplish this, each dimension in a Parameter Space is associated with a function that produces a set of zero or more physical constraints and objectives for every possible value in that dimension. In this way, the exact physical meaning of parameter values like "Crouching" are specified. The union of these n sets is then taken as the final set of constraints and objectives used in the constrained nonlinear optimization problem.

By way of example, let us again consider the Parameter Space introduced in Figure 3.7. A point from this 2-dimensional Parameter Space would be transformed into a set of constraints and objectives using 2 functions. The first would generate constraints and objectives for the Forward Speed parameter and could be written as,

$$ForwardSpeed(p0) = \{torso_{f_n-1,0} = torso_{0,0} + p0 \cdot f_n \cdot h\} \tag{3.13}$$

where $p0$ is the value of the Forward Speed parameter, and $torso_{f,d}$ is the position of the

character's torso on frame f in dimension d. We assume that dimension 0 is the appropriate "forward" dimension. This constraint should therefore force the character to move forward by a distance that necessitates an average speed of $p0$ m/s.

The second function would generate constraints and objectives for the Crouching parameter and could be written as,

$$Crouching(p1) = \begin{cases} \{head_{f,1} \leq 1.2 \mid 0 \leq f < f_n\} & \text{for } p1 = \text{On} \\ \varnothing & \text{for } p1 = \text{Off} \end{cases} \tag{3.14}$$

where $p1$ is the value of the Crouching parameter, and $head_{f,d}$ is the position of the character's head on frame f in dimension d. We assume that dimension 1 is the appropriate up/down dimension, and that the character is already constrained to stay on or above a floor plane located at 0 in this dimension. If $p1$ is On, the returned set of f_n constraints will therefore force the character to crouch in order to keep his head below the 1.2m "ceiling". If $p0$ is Off, no constraints or objectives (i.e. an empty set) are returned, leaving him free to stand upright.

Finally, the union of these two sets,

$$ForwardSpeed(p0) \quad \cup \quad Crouching(p1) \tag{3.15}$$

is taken as the final set of constraints and objectives used in the constrained nonlinear optimization problem.

Sampling a Parameter Space

Parameter Spaces must be sampled at discrete intervals in order to generate desired animation clips. Exactly how this is done is left in the hands of the designer, and there are several factors to be considered.

If a Parameter Space has any continuous parameters, then of course sampling at discrete intervals in those dimensions is necessary for our approach to be computationally tractable. For example, in Figure 3.8 one can see the designer has decided to sample the continuous Forward speed parameter in increments of 0.5 m/s. Additionally, some regions of a Parameter Space may specify infeasible or otherwise undesirable animations. In Figure 3.8, the designer has decided not to sample areas of the space where Forward speed $\geq$ 2 m/s

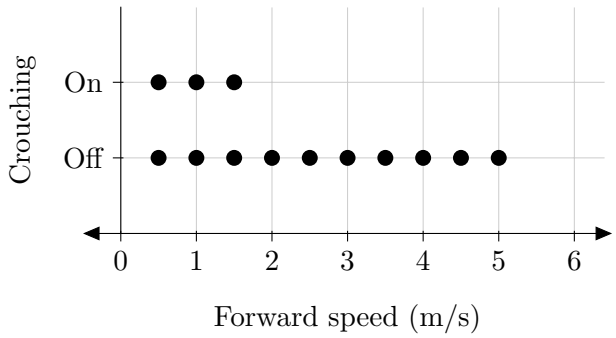


Figure 3.8: An example of how the Parameter Space from Figure 3.7 may be sampled at discrete intervals to generate desired animations.

and Crouching = On because he only wants the character to move slowly while crouched. Finally, some regions of a Parameter Space may need to be sampled at higher or lower resolution than others. For instance, a region where small changes in parameter values result in significant changes in motion may need to be sampled at more frequent intervals in order to adequately capture the details of the region.

Multiple Characters

In the preceding sections we alluded that "character" can be treated as just another parameter in a Parameter Space. This is an appealing notion since it implies that once an animation has been specified for one character, an equivalent animation may be generated for any other character, just by sampling the Parameter Space in the normal fashion. In truth, treating character as just another parameter is not quite so straightforward, and requires a little thought.

Difficulties with this approach arise because the constraint and objective generating functions for the animation-related Parameter Space dimensions must inevitably "know" something about the character. That is, in order to specify the movement of a character, these functions must be able to reference the variables $q_i(t)$ and $Q_i(t)$ that control the state of the character. However, different characters have different variables to match their different morphologies, and so the problem is revealed. How would one specify animation constraints like "Crouching" or "Limping" in a way that is completely character-agnostic? Dogs, snakes, fish and humans would all require different interpretations of these concepts—if they applied at all.

We offer a partial solution to this problem. For characters that are reasonably similar in morphology and style of motion, we continue to treat "character" as just another parameter in a Parameter Space. We abstract small differences in morphology by using a generic naming convention for character body parts. In this way, "equivalent" body parts, across all characters in the Parameter Space, are referenced by constraints and objectives in a consistent manner. For instance, "pelvis", "tail" and "left hind upper leg" would all be equally applicable to canine, horse, reptile and rodent characters. Constraints and objectives that were written for one, could be applied to all. Some care may need to be taken to ensure that constraints and objectives "scale" properly with the characters. For example, it would be unrealistic to expect a mouse to achieve the same running speed as a horse. Scaling may be accomplished through appropriate sampling of the Parameter Space for each character, or may be built into the units of each dimension. For instance, instead of specifying speed in an absolute unit like m/s, the dimension could use a relative unit that accounts for the character's leg length and muscle strength.

For characters that are significantly different in morphology (or style of motion), we use a divide and conquer approach. We split the Parameter Space into multiple separate Spaces, such that the characters within each Space are similar enough in morphology to each other that the first technique may be used effectively. For instance, all the bipeds in a game might be placed in one Parameter Space, the quadrupeds into a second and the snakes into a third.

3.5 Animation Clip Database and Real-Time Playback

When all the Parameter Spaces have been sampled, and each point has been converted into a constrained nonlinear optimization problem and solved, the result is a database of animation clips—one animation clip for each point sampled from the Parameter Spaces.

In most respects, these clips are equivalent to those that could be created during a motion-capture session, or animated by an artist using a conventional 3D animation software suite. The major differences are:

- The clips have been procedurally generated, and can therefore be easily re-generated en masse to meet changing design requirements.
- The clips were generated procedurally via Spacetime Control, and therefore contain a significant amount of meta-data about the physical dynamics of the motion. For

instance, the exact "muscle" torques exerted by the character, ground reaction forces, and ground contact timings are known explicitly.

- The clips have been sampled from a larger Parameter Space, and are therefore conceptually related to each other in known ways. In essence, each clip is "tagged" with meta-data about what motion the character is performing. This data can be used at run-time, to aid in the selection of clips that will achieve a particular result.

- Due to the precise nature of Spacetime Control, the clips can have higher technical accuracy than is usually possible with motion-capture, and do not require any clean-up like noise removal, cyclification, foot sliding correction, or penetration correction.

In the final block of our architecture, the generated animation clips are used for real-time in-game playback. We avoid specifying any particular playback technique, as we foresee each application will have its own unique requirements in a character animation system, and a 'one size fits all' approach is likely misguided. A wide variety of suitable techniques are available to choose from though, from straightforward animation clip playback, to more automated techniques like Motion Graphs [KGP02], Well-Connected Motion Graphs [ZS09], and Motion Fields [LWB*10], among many others [LCL06, MP07, SH07, TLP07]. In our own implementation we explore the use of Well-Connected Motion Graphs.

Chapter 4

Implementation

4.1 Overview

In order to demonstrate our architecture, we have developed a complete implementation of it, capable of generating a wide variety of 3D character animations. This chapter describes the design of that implementation, and the particular techniques and technologies used.

Much like the architecture itself, our implementation may be viewed as having two distinct parts. The first part is responsible for *animation generation* while the second part is responsible for *real-time playback* of the generated animation clips.

4.2 Animation Generation

The animation generating part of our implementation consists chiefly of an object-oriented library of Python classes which provide an API through which the user may define characters and parameter spaces, and ultimately generate and export animation clips. To derive the characters' equations of motion, and perform other symbolic manipulations, we make use of the Sympy library [Sym10] for Python. Constrained nonlinear optimization is performed by the program IPOPT [WB06], which is interfaced with our Python library through AMPL [FGK90] for convenience. The derivative-free "outer" optimization, responsible for determining contact timings, is performed by the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [HO96] library for Python.

The following sections provide a tour of our Python library. We briefly demonstrate its usage, and explain some of its internal operation.

In what follows, we use a right-handed coordinate system with the positive y-axis pointing up. We use standard gravity of -9.81 m/s^2 on the y-axis, and the character stands on an infinite XZ ground plane. Our Euler-angles use the Tait-Bryan convention with a Y-X-Z ordering.

4.2.1 Defining Characters

The first step in creating character animations with our system is, of course, to define the characters. Characters are defined as arbitrary systems of rigid-bodies connected by powered joints. To allow this, our Python library provides an API through which the user may instantiate rigid-bodies and joints, and then add these to character objects.

Rigid-Bodies

Since many human and animal body parts are of approximately ellipsoidal shape, our rigid-bodies are modeled as ellipsoids. A rigid-body is constructed by providing the axis-aligned length, width and height of the ellipsoid, as well as its total mass. The ellipsoid is constructed with its geometric center (and thus its center of mass) at the origin in the rigid-body's local coordinate system. This convention makes writing the equations of motion for the body simpler. The constructor also accepts a string which is used as the "generic" name for the body part. If multiple characters have identically-named rigid-bodies, then constraints and objectives that are written for one character may generally be used for the others as well (see Section 3.4.2 for more information). As an example, here we instantiate two rigid-bodies that make up a bipedal character's leg:

```
thigh_left = RigidBody (
    Name = "L_leg_upper", Mass = 6.41,
    Diameter = [0.16, 0.42, 0.16] )

calf_left = RigidBody (
    Name = "L_leg_lower", Mass = 3.13,
    Diameter = [0.11, 0.43, 0.11] )
```

Beyond simply storing the data provided to their constructors, our rigid-bodies are also responsible for several other tasks. The first is instantiating a vector of Sympy symbols (i.e. variables) $q(t)$ that will be used to represent the state of the rigid-body in each degree of

freedom. In our work, we use 6 degrees of freedom (3 translational and 3 rotational), and therefore $q(t)$ is defined as,

$$q(t) = [x(t),\ y(t),\ z(t),\ \phi(t),\ \theta(t),\ \psi(t)] \tag{4.1}$$

where x, y, z are the translational coordinates of the body, ϕ, θ, ψ are the rotational (i.e. Euler-angle) coordinates of the body and t is the temporal variable (a discrete frame index, in our case). These 6 state expressions are eventually treated as variables for the constrained nonlinear optimization routine to solve for. Rigid-bodies also expose these state variables individually as tx, ty, tz, rx, ry, and rz respectively, which makes it easier to refer to them when writing constraints. So for instance, `thigh_left.tz(0)` and `thigh_left.q[2](0)` would both refer to the Z translation of the thigh on frame 0.

Rigid-bodies are also responsible for calculating their principal mass-moments of inertia I_ϕ, I_θ and I_ψ. In our implementation, rigid-bodies are ellipsoids and therefore their principal moments of inertia are calculated as,

$$\begin{aligned} I_\phi &= (m/5)(b^2 + c^2), \\ I_\theta &= (m/5)(a^2 + c^2), \\ I_\psi &= (m/5)(a^2 + b^2) \end{aligned} \tag{4.2}$$

where m is the mass of the rigid-body and a, b, and c are the radii of the ellipsoid in the x, y and z axes respectively.

Finally, rigid-bodies define a method, `get_intersection_constraint()`, that returns a constraint enforcing non-intersection between themselves and a given sphere. This is used by the character to prevent intersection among all its rigid-bodies. Since our rigid-bodies are axis-aligned ellipsoids centered at the origin in local coordinates, the necessary constraint may be written as,

$$1 \leq \frac{{S_x}^2}{(a + S_r)^2} + \frac{{S_y}^2}{(b + S_r)^2} + \frac{{S_z}^2}{(c + S_r)^2} \tag{4.3}$$

where S_x, S_y and S_z are the given x, y and z coordinates of the sphere in the local rigid-body space, and S_r is the given radius of the sphere.

Joints

Joints are responsible for constraining the movement of rigid-bodies in one or more degrees of freedom. For example, a spherical (i.e. ball and socket) joint eliminates the 3 translational degrees of freedom, allowing only rotation about the point of constraint. A revolute (i.e. hinge) joint eliminates 5 degrees of freedom, allowing rotation on just a single axis. To enforce their constraints in a physically-correct manner, joints exert constraint forces in each degree of freedom they constrain.

In our implementation, each joint is required to implement a common interface that allows us to automatically derive the appropriate equations of motion in a consistent way. Each joint instantiates a vector of Sympy symbols (i.e. variables) $\lambda(t)$ that represents the constraint forces applied by the joint in each degree of freedom it constrains. Additionally each joint implements 2 methods: `get_state_constraints()` which returns a list, equal in length to $\lambda(t)$, of the constraints imposed by the joint on the involved rigid-bodies' states $q(t)$, and `get_force_constraints()` which returns a list of zero or more constraints on the constraint forces $\lambda(t)$ themselves.

Using this interface we implement two joint types: *body joints* and *contact joints*, both of which behave principally like spherical joints (i.e. eliminating translational movement).

Body Joints

Body joints are used to connect two body parts together (for example, a knee joint constrains the upper and lower legs). If desired, these joints can also apply "muscle" torque to allow characters to move, and can enforce rotational limits. They are constructed by providing references to the 2 rigid-bodies that are constrained, as well as the local-coordinates on each body where the constraint occurs. Minimum and maximum joint angles on each axis, and the maximum "muscle" torque that can be exerted by the joint are also provided. To continue our example from above, here we create a knee joint to connect the upper and lower leg of our bipedal character. The specified rotation limits prohibit relative rotation of the two bodies except in the positive direction on the Z axis, much like a human knee joint.

```
joint_knee_left = JointBody (
    Name = "L_knee",
    BodyA = thigh_left, PointA = [0.0, -0.21, 0.0],
    BodyB = calf_left, PointB = [0.0, 0.215, 0.0],
    RotationLimits = [[0,0], [0,0], [0, 2.8]],
    TorqueLimit = 180 )
```

Body joints constrain all 6 degrees of freedom and therefore define their constraint force vector $\lambda(t)$ as,

$$\lambda(t) = [f_x(t),\ f_y(t),\ f_z(t),\ \tau_\phi(t),\ \tau_\theta(t),\ \tau_\psi(t)] \tag{4.4}$$

where f_x, f_y, f_z are the translational forces, τ_ϕ, τ_θ, τ_ψ are the rotational torques.

The `get_state_constraints()` method returns 6 corresponding constraints,

$$\begin{aligned}
\forall\, t:&\quad 0 \leq X_{BodyA}(t)(PointA)_x - X_{BodyB}(t)(PointB)_x \leq 0,\\
\forall\, t:&\quad 0 \leq X_{BodyA}(t)(PointA)_y - X_{BodyB}(t)(PointB)_y \leq 0,\\
\forall\, t:&\quad 0 \leq X_{BodyA}(t)(PointA)_z - X_{BodyB}(t)(PointB)_z \leq 0,\\
\forall\, t:&\quad \phi_{min} \leq \phi_{BodyA}(t) - \phi_{BodyB}(t) \leq \phi_{max},\\
\forall\, t:&\quad \theta_{min} \leq \theta_{BodyA}(t) - \theta_{BodyB}(t) \leq \theta_{max},\\
\forall\, t:&\quad \psi_{min} \leq \psi_{BodyA}(t) - \psi_{BodyB}(t) \leq \psi_{max}
\end{aligned} \tag{4.5}$$

where X_i is a spatial transform that converts coordinates from the local-space of rigid-body i to world-space. The first three constraints constrain the world-space position of PointA on BodyA to equal the world-space position of PointB on BodyB. The last three constraints constrain the relative rotations of BodyA and BodyB to be within the specified joint rotation limits.

The `get_force_constraints()` method returns a single constraint, limiting the sum of squared torques applied by the joint:

$$\forall\, t:\quad \tau_\phi(t)^2 + \tau_\theta(t)^2 + \tau_\psi(t)^2 \leq TorqueLimit^2 \tag{4.6}$$

Contact Joints

Contact joints are used to define points on a character that may come into contact with the ground plane over the course of an animation. For example, contact joints might be placed on a character's hands and feet, thus allowing him to do cartwheels. Placing contact joints on his hands and knees would allow him to crawl etc. Unlike body joints, contact joints have a temporally-varying element to them. Contact joints contribute different constraints to each time step of the Spacetime Control problem depending on whether or not they are active in that time step. Before a Spacetime Control problem can be solved, the subset of time steps in which each contact joint is active must be explicitly defined. This can be done manually to produce a particular gait, or automatically to minimize an objective function as discussed in Section 4.2.5. In either case, we separate the definition of a contact joint from the specification of its contact timings—the character defines the contact joint, and the animation defines its contact timings.

Contact joints are constructed by providing a reference to the rigid-body which will contact the ground plane, and the point in local-coordinates on that body where the contact will occur. Additionally, a coefficient of static friction (μ) for the contact is provided.[1] This is used to ensure that the character does not take any action that would cause the contact to break free of static friction (i.e. exceed traction) and begin sliding. Here we add a contact joint to the bottom end of our bipedal character's lower leg:

```
joint_foot_left = JointContact (
    Name = "L_foot",
    Body = calf_left, Point = [0.0, -0.215, 0.0],
    Friction=0.5 )
```

Contact joints constrain 4 degrees of freedom and therefore define their constraint force vector $\lambda(t)$ as:

$$\lambda(t) = [f_x(t),\ f_y(t),\ f_z(t),\ \tau_\theta(t)] \tag{4.7}$$

[1] In a more robust implementation, the coefficient of friction might be specified with the contact timings, thus allowing it to vary per contact (or even per time step) if necessary.

The `get_state_constraints()` method returns 4 corresponding constraints

$$\begin{aligned}
\forall\, t \in \mathit{JointActive} :\quad & 0 \leq && X_{\mathit{Body}}(t)(\mathit{Point})_x - X_{\mathit{Body}}(t-1)(\mathit{Point})_x && \leq 0, \\
\forall\, t \in \mathit{JointActive} :\quad & 0 \leq && X_{\mathit{Body}}(t)(\mathit{Point})_y && \leq 0, \\
\forall\, t \in \mathit{JointActive} :\quad & 0 \leq && X_{\mathit{Body}}(t)(\mathit{Point})_z - X_{\mathit{Body}}(t-1)(\mathit{Point})_z && \leq 0, \\
\forall\, t \in \mathit{JointActive} :\quad & 0 \leq && \theta_{\mathit{Body}}(t) - \theta_{\mathit{Body}}(t-1) && \leq 0
\end{aligned} \tag{4.8}$$

where *JointActive* is the set of time steps for which the contact joint is active. The first and third constraints constrain the world space position of the contact point at time step t, to equal the world-space position of the contact point at time step $t-1$ on the x and z axes, thus preventing the Body from sliding while in contact. The second constraint constrains the world-space position of the contact point on the y-axis to zero (i.e. on the ground plane). Finally, the fourth constraint constrains the θ rotation of the Body at time t to equal the θ rotation of the Body at time $t-1$, thus preventing the Body from twisting while in contact. This is done so that our point contact behaves more like an area contact, simulating a foot.

The `get_force_constraints()` method returns 6 constraints:

$$\begin{aligned}
&\forall\, t \in \mathit{JointActive} :\quad 0 \leq (\mu f_y(t))^2 - f_x(t)^2 - f_z(t)^2, \\
&\forall\, t \in \mathit{JointActive} :\quad 0 \leq f_y(t), \\
&\forall\, t \notin \mathit{JointActive} :\quad 0 \leq f_x(t) \leq 0, \\
&\forall\, t \notin \mathit{JointActive} :\quad 0 \leq f_y(t) \leq 0, \\
&\forall\, t \notin \mathit{JointActive} :\quad 0 \leq f_z(t) \leq 0, \\
&\forall\, t \notin \mathit{JointActive} :\quad 0 \leq \tau_\theta(t) \leq 0
\end{aligned} \tag{4.9}$$

where again, *JointActive* is the set of time steps for which the contact joint is active. The first constraint limits the ground reaction forces to be within a "static friction cone". The second constraint ensures that the ground reaction forces only push upwards on the character (never pulling him down). The last 4 constraints simply ensure that when the contact joint is not active, no forces or torques are applied.

Characters

With the rigid-bodies and joints defined, character objects may now be created. Each character is constructed with a name, and references to the rigid-bodies and joints it is comprised of are added to it. To continue our simple example, we could add the legs, knee joint, and contact joint we made previously to a new character:

```
char_knight = Character(Name = "BoldyBraveSirRobin")
char_knight.add_body(thigh_left)
char_knight.add_body(calf_left)
char_knight.add_joint(joint_knee_left)
char_knight.add_joint(joint_foot_left)
# And so on, for the rest of the bodies and joints...
```

In addition to storing references to the rigid-body and joint objects it is comprised of, characters are responsible for writing a set of constraints that prevent intersection among their rigid-bodies (which one will recall, we model as ellipsoids). We accomplish this only approximately using the `get_intersection_constraint()` method defined in our rigid-body class (see Section 4.2.1). Each rigid-body is approximated as a sphere inscribed inside its actual ellipsoid, and constraints are written to prevent this sphere from intersecting any of the other ellipsoids on any frame. This simplification makes the necessary constraints easier to derive at the expense of the occasional self-intersection.

Finally, characters are responsible for deriving their own equations of motion, to which we dedicate our discussion in the next Section, 4.2.2.

4.2.2 Deriving Equations of Motion

We derive the equations of motion for characters in our implementation using a straightforward Newton-Euler approach, with redundant coordinates. As shown above, our character class stores references to the rigid-body and joint objects it is comprised of. When a character is used in an animation, the character object is responsible for inspecting its rigid-bodies and joints, and deriving the appropriate equations of motion for itself.

Ignoring the character's joints for now, the equations of motion for an individual rigid-body may be written as follows. To keep these equations simple, we write them with respect to a coordinate frame whose origin coincides with the rigid-body's center of mass. To begin, the rigid-body object is queried for its vector $q(t)$ containing its 6 state expressions

(see Equation 4.1). The velocities of these terms, $\dot{q}(t)$ are defined using the central finite difference formula,

$$\dot{q}(t) = \frac{q(t+1) - q(t-1)}{2h} \tag{4.10}$$

and the accelerations, $\ddot{q}(t)$ are defined using the second order central finite difference formula,

$$\ddot{q}(t) = \frac{q(t+1) - 2q(t) + q(t-1)}{h^2} \tag{4.11}$$

where h is the duration of a single time step. Additionally, each rigid-body is queried for its scalar mass m, and its principal mass-moments of inertia, I_ϕ, I_θ, and I_ψ (see Equation 4.2).

With these terms defined, we can write the rigid-body equations of motion (the Newton-Euler equations) as,

$$\begin{aligned}
\forall\, t\ : \quad & f_x(t) = m\ \ddot{x}(t) \\
\forall\, t\ : \quad & f_y(t) = m\ (\ddot{y}(t) - 9.81) \\
\forall\, t\ : \quad & f_z(t) = m\ \ddot{z}(t) \\
\forall\, t\ : \quad & \tau_\phi(t) = I_\phi\ \ddot{\phi}(t) + (I_\psi - I_\theta)\ \dot{\theta}(t)\ \dot{\psi}(t) \\
\forall\, t\ : \quad & \tau_\theta(t) = I_\theta\ \ddot{\theta}(t) + (I_\phi - I_\psi)\ \dot{\psi}(t)\ \dot{\phi}(t) \\
\forall\, t\ : \quad & \tau_\psi(t) = I_\psi\ \ddot{\psi}(t) + (I_\theta - I_\phi)\ \dot{\phi}(t)\ \dot{\theta}(t)
\end{aligned} \tag{4.12}$$

where f_x, f_y, f_z, τ_ϕ, τ_θ and τ_ψ are placeholders for the translational forces acting on x, y and z and the rotational torques acting on ϕ, θ and ψ respectively. The first three equations are a straight-forward application of Newton's second law of motion (see Equation (3.3)). The last three are Euler's equations, and contain an extra term to account for the fictitious forces that arise due to our use of a non-inertial (i.e. rotating) frame of reference. Considering each rigid-body in isolation as we have been, these placeholder forces and torques are always zero, and the movement of the rigid-body follows inevitably from its initial conditions. However, our rigid-bodies are connected by joints, and these joints apply forces and torques to the bodies in order to constrain their relative motions. To write the necessary expressions for these forces, it is no-longer sufficient to consider each rigid-body in isolation—we will need to consider the system as a whole.

To begin, each rigid-body is queried for its state vector $q(t)$, and these are compiled into

a single large state vector $\mathbf{q}(t)$

$$\begin{aligned}\mathbf{q}(t) = [\quad & x_0(t),\ y_0(t),\ z_0(t),\ \phi_0(t),\ \theta_0(t),\ \psi_0(t), \\ & x_1(t),\ y_1(t),\ z_1(t),\ \phi_1(t),\ \theta_1(t),\ \psi_1(t), \\ & \cdots, \\ & x_n(t),\ y_n(t),\ z_n(t),\ \phi_n(t),\ \theta_n(t),\ \psi_n(t) \quad]\end{aligned} \tag{4.13}$$

where n is the number of rigid-bodies in the character.

Additionally, the `get_state_constraints()` method is called on each of the character's joints and the resulting constraint expressions (ignoring their lower and upper bounds) are compiled into a single constraint expression vector $\mathbf{C}(t)$.

The next step is to take the Jacobian of $\mathbf{C}(t)$ with respect to $\mathbf{q}(t)$. The Jacobian is simply a large matrix with one row for every constraint expression in $\mathbf{C}(t)$, and one column for every state expression in $\mathbf{q}(t)$. Element j,k is the first-order partial derivative of the jth constraint expression in $\mathbf{C}(t)$ with respect to the kth state expression in $\mathbf{q}(t)$. The Jacobian $\mathbf{J}$ is therefore written as

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{C}_0}{\partial \mathbf{q}_0} & \frac{\partial \mathbf{C}_0}{\partial \mathbf{q}_1} & \cdots & \frac{\partial \mathbf{C}_0}{\partial \mathbf{q}_k} \\ \frac{\partial \mathbf{C}_1}{\partial \mathbf{q}_0} & \frac{\partial \mathbf{C}_1}{\partial \mathbf{q}_1} & \cdots & \frac{\partial \mathbf{C}_1}{\partial \mathbf{q}_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{C}_j}{\partial \mathbf{q}_0} & \frac{\partial \mathbf{C}_j}{\partial \mathbf{q}_1} & \cdots & \frac{\partial \mathbf{C}_j}{\partial \mathbf{q}_k} \end{bmatrix} \tag{4.14}$$

We also compile a large vector of joint constraint forces $\boldsymbol{\lambda}(t)$, with each joint contributing its constraint force vector $\lambda(t)$. Using these vectors we can now write expressions for the place-holder forces we used above ($f_x(t)$, $f_y(t)$, $f_z(t)$, $\tau_\phi(t)$, $\tau_\theta(t)$ and $\tau_\psi(t)$):

$$F_k(t) = \mathbf{J}^T \cdot \boldsymbol{\lambda}(t) \tag{4.15}$$

where $F_k(t)$ is the force applied on the kth degree of freedom (corresponding to the kth state variable in $\mathbf{q}(t)$), and $\mathbf{J}^T$ is the transpose of $\mathbf{J}$.

The character now declares its state variables $\mathbf{q}(t)$ and force variables $\boldsymbol{\lambda}(t)$ in the Space-time Control problem. Finally, it writes its physical constraints (4.12) and joint constraints

(obtained by calling `get_state_constraints()` and `get_force_constraints()` on each of its joints) to the Spacetime Control problem.

4.2.3 Defining Parameter Spaces

With the characters defined, the user now specifies—using Parameter Spaces—the animations the system should generate. We take a straight-forward approach to defining and sampling Parameter Spaces in our implementation. Our Python library provides an API through which the user may instantiate Parameter Spaces, add dimensions to them, and then generate and solve the resulting Spacetime Control problems.

Parameter Spaces are initially constructed by simply providing a name for the Space. For example:

```
ps_example = ParameterSpace(Name = "Example")
```

Dimensions are then added to the Parameter Space using its `add_dimension()` method which accepts a list of "Specifiers". In our API, Specifier is an abstract base-class for several other classes, including Constraint, Objective, Character, ContactTiming, and AnimationTiming. Each Specifier in the list passed to `add_dimension()` simultaneously defines a value in the new dimension to be sampled, and provides the corresponding constraints or objective functions to be used when sampling it. For example, using the one Specifier that has already been introduced—Character—we can create a "character dimension" in our Parameter Space with 3 possible values:

```
ps_example.add_dimension( [char_knight, char_princess, char_ogre] )
```

When sampled, the character object will automatically write its various state variables and physical constraints, as described in Section 4.2.2, to the Spacetime Control problem. By adding more dimensions, we can begin to specify the motions these characters should perform. Here we add a dimension with just a single value—a Constraint specifier that causes the character's torso to begin the animation at the origin on the X and Z axes.

```
ps_example.add_dimension( [
    Constraint(Name = "torsoBegin",
        lb = 0, c = torso.tx(0)**2 + torso.tz(0)**2, ub = 0)
] )
```

And here we add a third dimension with 49 values generated in a loop. Each value specifies a particular position for the character's torso on the X and Z axes at the end of the animation (Frame 29, we assume in this case).

```
ps_example.add_dimension( [
    Constraint(Name = "torsoEnd",
        lb = 0, c = (torso.tx(29) - x)**2 + (torso.tz(29) - z)**2, ub = 0)
    for x in range(-3,4) for z in range(-3,4)
] )
```

We could also add a fourth dimension with a user-specified animation length and framerate,

```
ps_example.add_dimension( [AnimationTiming(Length = 1.0, FPS = 30)] )
```

And finally we could add a fifth dimension with user-specified contact timings for each contact joint,

```
ps_example.add_dimension( [
    ContactTiming({
        joint_foot_left:[(0.0, 0.5)], #contact starting at 0%, lasting for 50%
        joint_foot_right:[(0.5, 0.5)] #contact starting at 50%, lasting for 50%
    })
])
```

To generate all the animations specified by a Parameter Space, the user simply calls the Space's `generate()` method and provides the name of the constrained nonlinear optimization solver to use. For example, in this case we use the IPOPT solver:

```
ps_example.generate(solver = "ipopt")
```

The `generate()` method always samples the dimensions of a Parameter Space in a fully combinatorial fashion, as described by Equation (3.12). This simplifies our system somewhat (eliminating the need for the user to specify desired and undesired regions of each Parameter Space), but does occasionally result in infeasible or otherwise undesirable animations specifications being made. In the infeasible cases, the animations are simply never solved (the optimization routine returns an error). In the undesirable cases, the

user may manually delete the resulting animations from the database if he wishes. When generated, our ps_example Parameter Space will produce $3 \times 1 \times 49 \times 1 \times 1 = 147$ unique animations.

4.2.4 Inner Optimization with IPOPT and AMPL

When the `generate()` method of a Parameter Space is called, the Parameter Space is sampled, and a set of Specifier objects is collected for each point in the Space. Together, each set of Specifier objects completely describes an animation to be produced, including the character, constraints, objective functions and timing information.

At this point, we use IPOPT [WB06] and AMPL [FGK90] to solve the resulting Spacetime Control problems. IPOPT is an open source software package that uses an interior point method for the optimization of large-scale nonlinear problems of the form in (3.1). It is thus well-suited to our needs. However, IPOPT and similar solvers typically require some additional information beyond just the variables, constraints and objective function that form the problem definition. For instance, IPOPT requires information regarding the problem structure, including the number of non-zeros and the sparsity structure for both the Jacobian of the constraints, and the Hessian of the Lagrangian function. It also requires functions for evaluating the gradient of the objective function and the Jacobian of the constraints among others. While we could likely compute this information in Python using Sympy, we find it easier to interface with IPOPT through AMPL.

AMPL is an algebraic modeling language for describing large-scale optimization problems, and an associated interpreter for translating these problems into the form required by solvers like IPOPT. Of particular importance to us, AMPL automatically computes the problem structure and the necessary derivatives required by IPOPT. Additionally, AMPL attempts to perform some simplification of the problem by eliminating unnecessary variables and constraints, and the result is often a smaller problem that solves in less time. These features make AMPL very attractive to us, and we therefore use it as an intermediary between our Python library and IPOPT.

To solve our Spacetime Control problem with AMPL and IPOPT, a string for holding the AMPL problem is created, and each of the Specifier objects mentioned above writes itself to the problem using the appropriate AMPL syntax. In the case of low-level Specifiers like Constraints and Objectives, this process is fairly straightforward. Constraints are trivially written in AMPL format with a minimum of effort. Meanwhile, Objective functions are

collected and a weighted sum of their values, as per Equation (3.2), is added to the AMPL problem as the final objective function. Higher-level Specifiers like Character perform their own processing to ultimately generate low-level Constraint and Objective Specifiers which are then handled as before. In addition, higher-level Specifiers may also introduce variables. For instance, our Character Specifier is responsible for writing the vectors of state and force variables defined by its RigidBody and Joint objects respectively.

With the AMPL problem written, a pipe is opened to the AMPL executable, and the problem string is transmitted. AMPL performs its preprocessing of the problem, calls the IPOPT solver executable on it, and finally returns the results to our Python library using another pipe. If the solver was successful, the results are saved to our database of animation clips.

In order to speed up the generation of animations, we parallelize this optimization process at the level of individual Spacetime Control problems. Each Spacetime Control problem is completely independent of the others, and they may therefore be solved simultaneously by running multiple AMPL and IPOPT executables. This makes our approach scale well with the number of processing units available.

4.2.5 Outer Optimization with CMA-ES

As shown briefly in Section 4.2.3, our implementation allows the user to explicitly specify the animation length and contact timings for animations if desired. Suitable values for these quantities are not always easy to determine manually though, and so our implementation provides a facility for determining them automatically using the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [HO96] library for Python. If the user fails to specify animation length or contact-timings manually, then this facility is automatically invoked per Spacetime Control problem to determine the unspecified values.

CMA-ES is a derivative-free method for the numerical optimization of nonlinear optimization problems. We use it as an outer optimization loop around our Spacetime Control optimization. For the sake of this outer optimization we make the assumption that each contact joint has, at most, one contact per animation. We can therefore express our contact timings using two variables per contact joint—the starting time of the contact and the duration of the contact—both expressed as a percentage of the total animation length. If the outer optimization is to determine the animation length, then we also introduce a variable to express the length of the animation in seconds. We provide an initial starting point for these

variables that spaces the contacts evenly over the length of a 1 second animation. All the variables are given lower bounds of 0.0 and upper bounds of 1.0. The variable for animation length is scaled by a factor of 3 (allowing up to 3 second animations to be produced). We use values for the CMA-ES parameters of $\lambda = 16$, $\mu = 8$, and $\sigma = 1/3$. At each iteration of the CMA-ES optimization, the values of the objective functions returned from our inner Spacetime Control optimizations are provided as fitnesses. According to the new contact timing and animation length variables determined by CMA-ES, we produce new Spacetime Control problems, solve them, and the process is repeated. After 100 iterations, we take the best solution found as the final animation clip.

4.2.6 Exporting Animation Clips

When all the points in a Parameter Space have been sampled, and the corresponding Spacetime Control problems have been solved, the result is a large database of state and force vectors that describe the dynamics (i.e. motions and forces) of the characters' rigid-bodies through time in each animation.

To be widely useful though, these animations must be exported to a standard skeletal animation format so they may be handled by conventional animation software and real-time 3D graphics engines. Standard skeletal animation systems represent character motion data as a hierarchy of transforms or "bones". In practice there is little difference between the rigid-bodies in our representation and these bones—both represent parts of a character's body for the sake of describing their motions. The chief difference is simply that the bones in a skeletal animation system are arranged in a hierarchy, whereas the rigid-bodies in our representation are not.

We therefore provide a simple facility to convert our "flat" rigid-body representation into a hierarchy on demand. The user specifies a rigid-body that will become the root of the hierarchy. For the purpose of exporting animation clips, this is usually the rigid-body that represents the character's torso or pelvis, though any rigid-body will suffice. A breadth-first traversal of the character is then performed, starting at the root body, and proceeding through all the rigid-bodies connected to it by BodyJoints, and then through all the bodies connected to those bodies, and so forth, until every rigid-body in the character has been visited. As it proceeds, this traversal yields parent and child pairs, allowing us to express our character as a hierarchy of rigid-bodies. If desired, the transforms of child bodies are then expressed relative to the transforms of their parent bodies.

Using this "automatic rooting" facility, we export our character animations to two skeletal animation file formats. The first of these, *Biovision Hierarchy (BVH)*, is a standard format for exchanging motion capture data, and is widely supported by a variety of 3D software packages. We use the BVH format in conjunction with the open-source 3D software package Blender [Ble11], to inspect our animation clips once they are generated. The second format we export to, *Ogre3D Skeleton XML*, is the native skeletal animation format for the open-source real-time 3D rendering engine, Ogre3D [Str09]. Our Motion Graph implementation, provided by the AISandbox (see Section 4.3.1), is based on Ogre3D and relies on it to handle various low-level tasks like animation loading and playback. We therefore export our animation clips to its preferred format.

4.3 Real-Time Playback

The real-time playback portion of our implementation consists chiefly of the Motion Graph technique [KGP02]. Additionally, we augment this with some key features of the Well-Connected Motion Graph technique [ZS09] for achieving smoother transitions and better connectivity within the graph. Motion Graphs are a well-studied data-structure (e.g. Reitsma and Pollard [RP07] provide an extensive evaluation) and we feel their highly-automated nature provides a good fit for the purpose of demonstrating our architecture.

4.3.1 Motion Graphs

Motion Graphs were introduced by Kovar et al. [KGP02], and provide an automatic method of producing continuous streams of character motion in real-time, given a database of animation clips as input. The Motion Graph itself is a directed-graph structure where each edge represents a clip of character animation, and each node serves as a transition point connecting these clips (see Figure 4.1). Continuous streams of character motion may be produced by simply walking the graph, playing the clips of animation data encountered along each edge.

To construct a Motion Graph, a database of animation clips—typically motion captured, but in our case generated procedurally via Spacetime Control—is provided as input. Additionally, the skinned character mesh (i.e. the polygons that are actually rendered and moved via the character's bones) is provided. This database is then analyzed to determine, for every frame in every clip, its similarity to every other frame in the database. For a number of

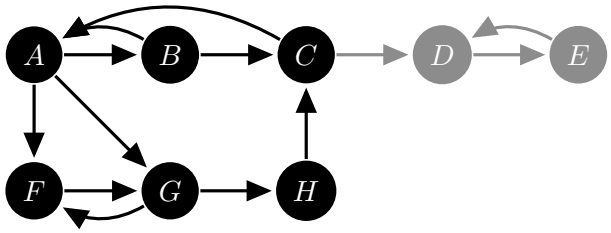


Figure 4.1: A simple Motion Graph. Nodes D and E are not part of the largest Strongly Connected Component and would be discarded.

reasons, the naive approach of directly comparing the rotations of bones fails to adequately capture similarity between frames. Instead, the vertices that make up the character's mesh are treated as a "Point Cloud" that moves in response to the character's bones (see Figure 4.2). The sum of squared distances between corresponding points in this cloud becomes the primary metric of frame-similarity. Additionally, the similarity metric measures differences in the velocities of these vertices between frames, while accounting for differences in the overall translation and rotation of the character between frames. When this analysis is complete, the result is a 2D similarity-map for every pair of frames in the database (for example, see Figure 4.3). The locally minimal points on this map are determined, and those locally minimal points that meet a designer-specified threshold are selected to become transition points in the Motion Graph. As a final step, to remove dead-ends and other poorly connected parts of the graph, the largest Strongly Connected Component (SCC) of the Motion Graph is computed, and any edge that does not connect two nodes in this component is discarded (again, see Figure 4.1).

At run-time, walks on a Motion Graph may be produced in any number of ways to generate useful character motion. For instance, Kovar et al. [KGP02] demonstrate the use of a search technique similar to A* [HNR68] for finding walks that minimize an arbitrary metric. Specifically they find walks that will follow a user-specified path sketched on the ground plane. In our implementation, we are content to simply perform random walks of the graph—thus realizing the "elaborate screen saver" imagined by Kovar et al.

In our implementation we make use of the AISandbox [Cha11] which provides—among many other things—a working Motion Graph implementation. We split our animation clip database into sets—one for each character—and build separate Motion Graphs for each of them.

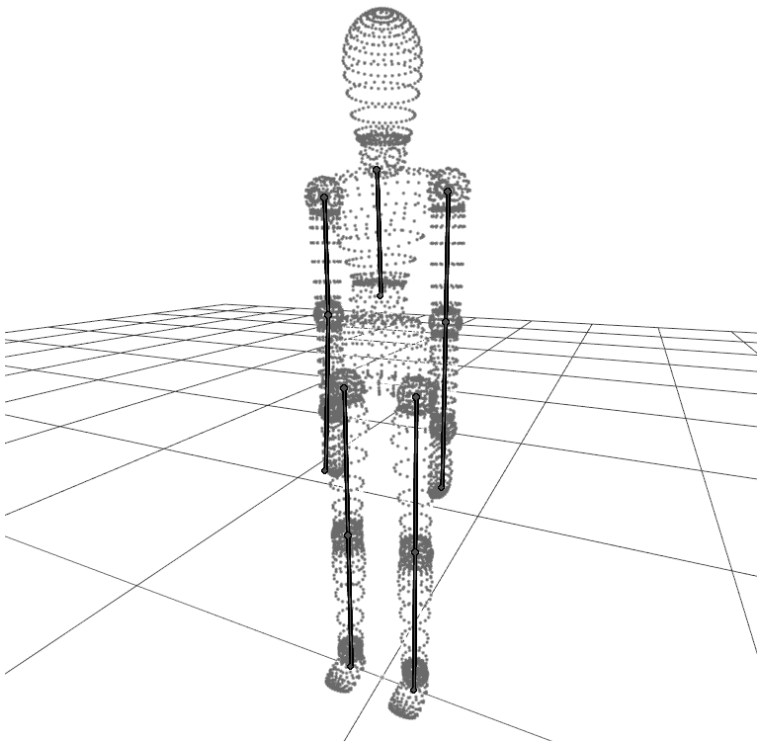
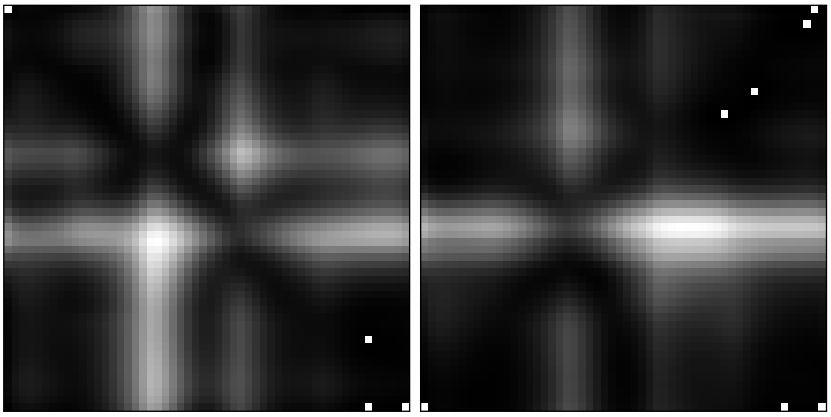


Figure 4.2: The vertices that make up a character's mesh form a "Point Cloud" that is used to determine similarity between frames. The thick black lines represent the character's bones.



(a) (b)

Figure 4.3: Some typical examples of Motion Graph similarity-maps. Darker pixels represent greater similarity between frames. We use single white pixels in otherwise dark regions to mark the locally-minimal points.

4.3.2 Smoother Transitions and Better Connectivity

The standard Motion Graph algorithm, as described above, often suffers from a lack of either smooth transitions, or good connectivity, and improving one is usually at the expense of the other. In our own experiments we found that it was often difficult to build Motion Graphs that included all of the input motion data, without tolerating some very noticeable discontinuities during transitions. For example, our run-cycle animation clips contain substantially different motions with much higher velocities than any of the other animation clips in our database (e.g. walk-cycles and stepping etc.). The Motion Graph algorithm was therefore unable to create many smooth transitions between these run-cycles and the other clips. Sampling our Parameter Spaces at higher resolutions, thus producing greater numbers of similar animation clips, often helped. This was especially true for achieving better connectivity between clips sampled from the same Parameter Space with continuous-valued parameters. However, it did less to help in the case of clips sampled from different Spaces, or from Spaces with discrete-valued parameters (e.g. crouching and not crouching) where sampling at intermediate values was not possible. Additionally the extra time required to generate so many nearly identical animation clips via Spacetime Control was inefficient.

To improve this situation, we borrow the key feature of the Well-Connected Motion Graph technique proposed by Zhao and Safonova [ZS09]. Specifically, we interpolate all motion segments that share the same character and contact configuration in the manner they describe. This has the desired effect of producing greater numbers of similar frames, allowing smoother transitions and better connectivity in the Motion Graph. Most importantly though, it operates across all the animation clips (regardless of the particulars of their respective Parameter Spaces), and it does so in a much more computationally efficient manner than Spacetime Control allows.

Chapter 5

Results

5.1 Some Examples

In order to demonstrate our architecture, we have used our implementation of it to produce several Parameter Spaces of animation for a humanoid character "Chip" with 22 degrees of freedom (see Figure 5.1). The resulting 36 animation clips were then used to build a Motion Graph that produces random walks of smooth-looking animation in real-time. All of these animations were produced with a time step of 0.05 seconds (i.e. 20 frames per second).

The first of our Parameter Spaces, "WalkRunTurn-Cyclic" produces 24 cyclic animations that cover walking, running and turning-on-the-spot motions. This Space is parameterized on just two key dimensions: forward speed (in m/s) and turn rate (in rad/s). Figure 5.2 shows the 1 straight and 8 left turn walking animations we sampled from this Space. Figure 5.3 shows the 1 straight and 3 left turn running animations. For brevity, we neglect to show the right versions of all these animations, since they are essentially identical, only mirrored.

One will note that there are only 7 running animations produced (out of the 17 sampled) as many of the Spacetime Control problems with faster turn rates fail to solve. Running while making extremely tight turns is a difficult, if not impossible, task for most real-world animals, and so this result is to be expected. Such failures could even be considered a benefit as physically implausible animations are removed automatically—designers need not worry about specifying *only* physically-plausible motion when designing Parameter Spaces. If these animations were truly desired though, the constraints in the Spacetime Control problem could be loosened to allow more solutions to be found. For example, increasing the

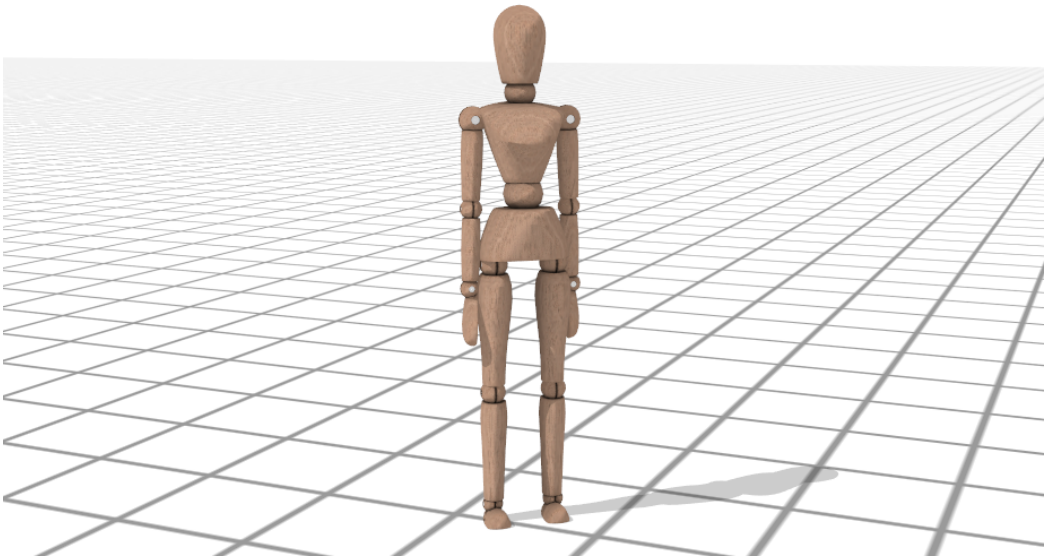
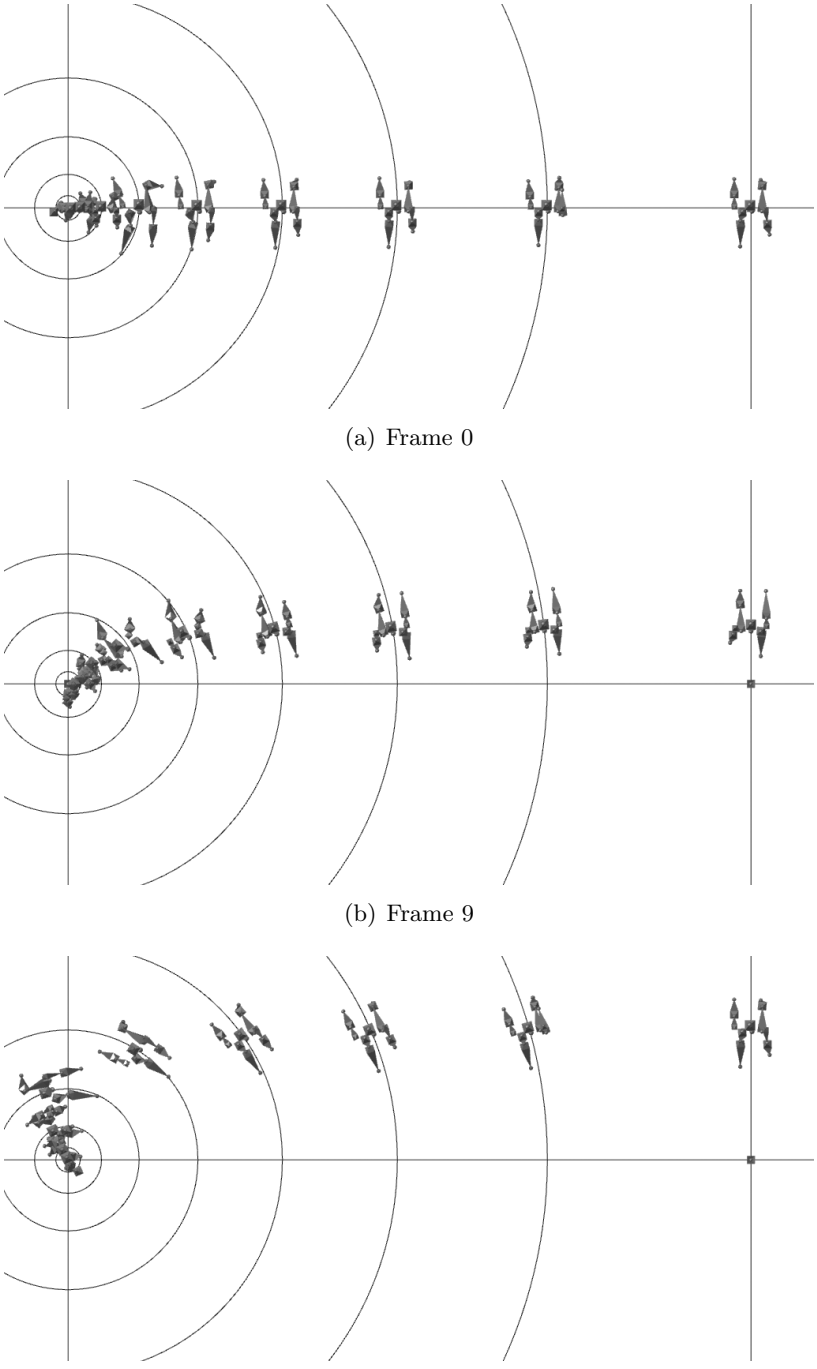


Figure 5.1: Chip the mannequin inside our game-engine environment

character's muscle torque limits and increasing the coefficient of friction between his feet and the ground plane would likely have the desired effect.

As another example, we created a "WalkSpin-Cyclic" Parameter Space that produces 4 cyclic animations in which the character moves in a straight line while spinning. This Space is parameterized on 2 key dimensions: forward-speed (0.5 m/s and 1 m/s) and spin direction (clockwise and counter-clockwise). It is assumed that the character will always make 1 full rotation over the course of the animation (otherwise the animation cannot be cyclic). Figure 5.4 shows the 1 m/s, counter-clockwise animation we sampled from this Space. Since our CMA-ES optimization is limited to assuming at most one contact per contact joint, and since 4 steps were required for this motion, we specified the contact times manually.

Finally we created a "Stepping-Cyclic" Parameter Space that produces 8 cyclic animations in which the character begins in an idle standing pose, takes 2 steps in some direction, and returns to an idle pose. The character maintains his orientation while doing this. This Space is parameterized on 2 key dimensions that resemble polar coordinates: the direction the steps are in (in radians), and the distance traveled from the starting point (in meters). Figure 5.5 shows five of the animations we sampled from this Space. Again, since our CMA-ES optimization is limited to assuming at most one contact per contact joint, we specify the



(a) Frame 0

(b) Frame 9

(c) Frame 20

Figure 5.2: Nine walking animation clips sampled from our "WalkRunTurn-Cyclic" Parameter Space (viewed from above). The animation clip on the far right walks in a straight line, while the others turn at different rates to the left.

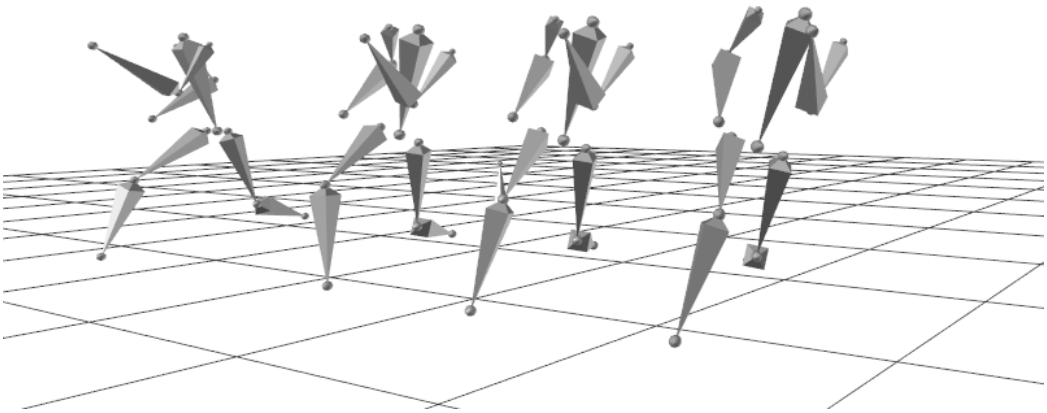


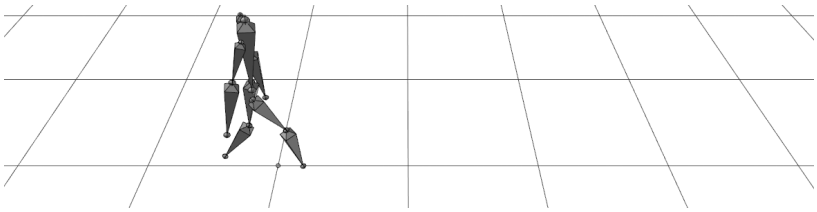
Figure 5.3: Four running animation clips sampled from our "WalkRunTurn-Cyclic" Parameter Space. Note how the characters lean into the turn (especially the characters on the right with the highest turn rates).

contact times manually.

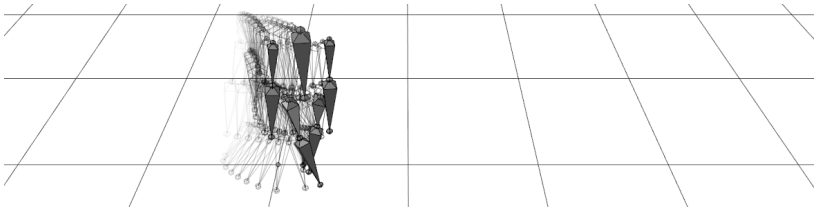
In the above experiments we used a desktop PC with a 4-core Intel 3.4 GHz processor. We found that it takes the system about 10 minutes to solve a single Spacetime Control problem. Given 4 cores, this means we can solve about 4 Spacetime Control problems in parallel in 10 minutes. To perform a full CMA-ES optimization of an animation (for optimal contact timings and animation length) takes much longer—typically between 24 and 48 hours.

5.2 Handling a Design Change

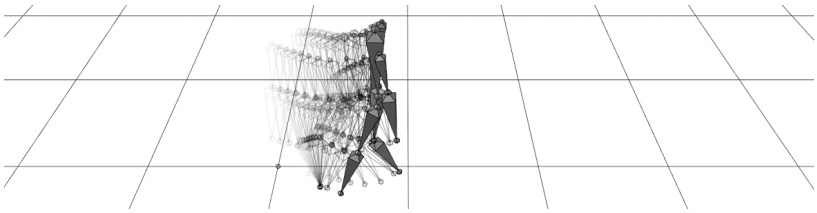
After creating the 36 example animations described above, we decided it would be interesting to create alternate versions of all these animations in which the character has a limp. To make this addition was remarkably easy. We added an extra dimension with 2 values (Healthy and Limping) to each of the 3 Parameter Spaces. The Healthy value has no specifiers and the Limping value has a constraint that places an upper bound on the ground reaction force that can be exerted on the character's right foot. Specifically the force is limited such that his right foot can only support 0.6 times his full body weight:



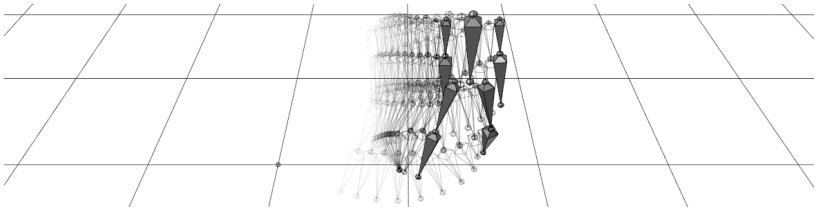
(a) Frame 0



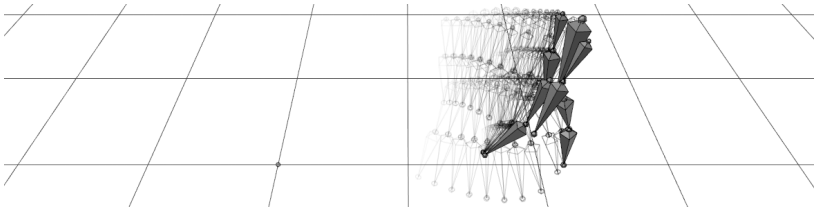
(b) Frame 10



(c) Frame 20

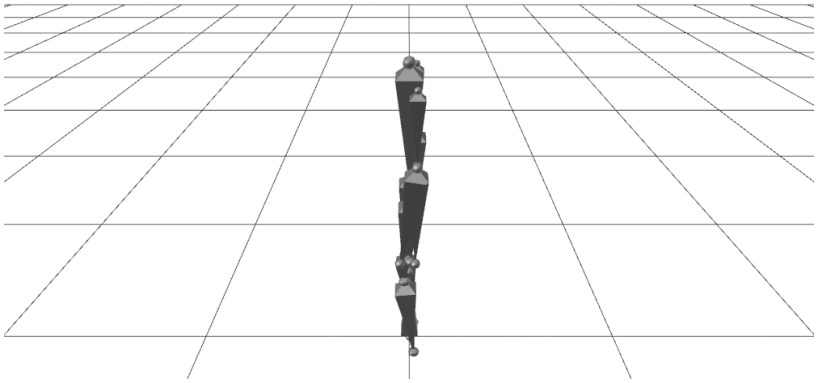


(d) Frame 30

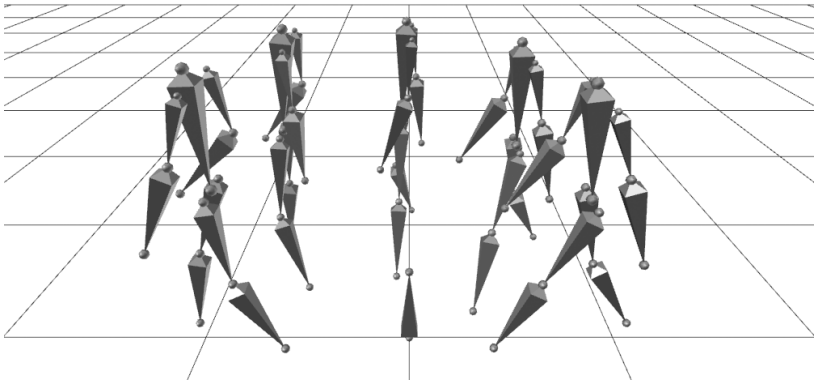


(e) Frame 40

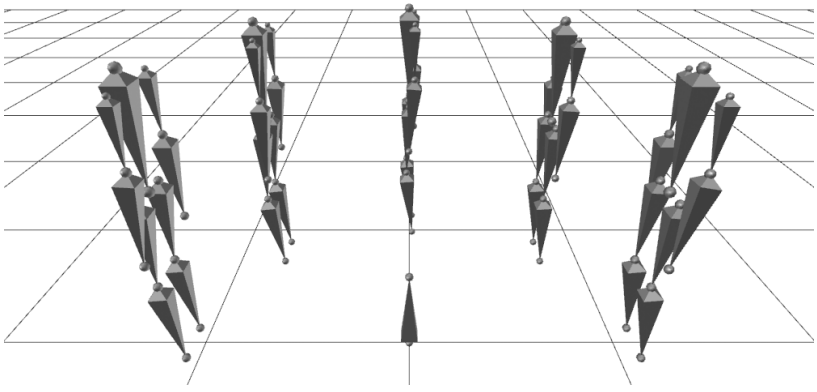
Figure 5.4: A single character animation sampled from our "WalkSpin-Cyclic" Parameter Space. The character spins counter-clockwise while walking in a straight line to the right. He performs one complete revolution.



(a) Frame 0



(b) Frame 23



(c) Frame 36

Figure 5.5: Five character animations sampled from our "Stepping-Cyclic" Parameter Space (the remaining 3 animations that would complete the circle are not shown for clarity)

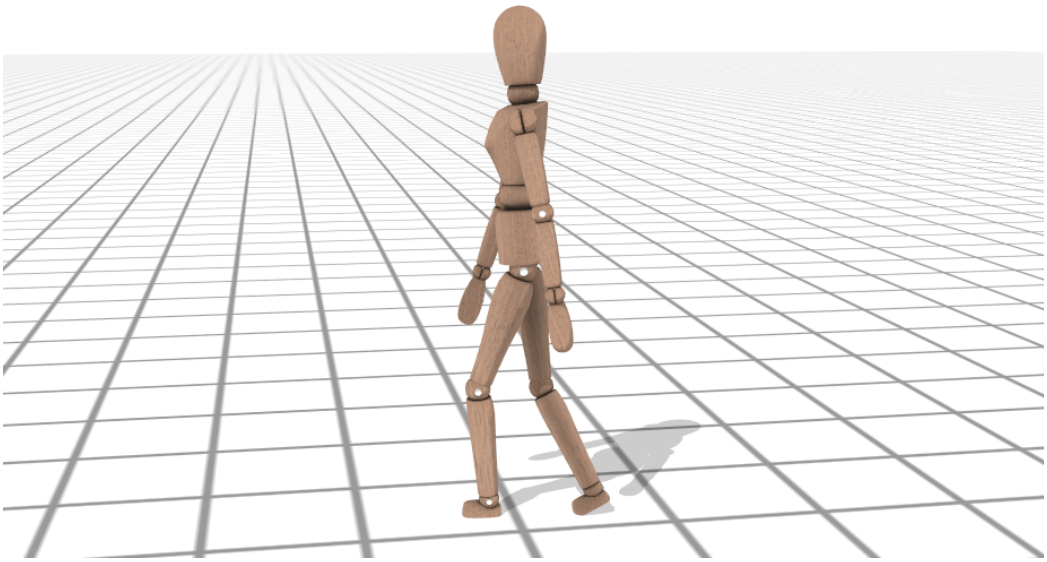


Figure 5.6: Chip performs one of his walk-cycle animations in the Motion Graph.

```
parameter_space.add_dimension([
    None,
    Constraint(Name = "rightLegLimp",
        c = joint_foot_right.fy(t), ub = char_chip.get_mass() * 9.81 * 0.6)
])
```

After running the animation generation process again, we had limping versions of all the animations, (with the exception of the running animations which all failed to solve when given the limping constraint). In the case of our "WalkRunTurn-Cyclic" Parameter Space in which the contact times were optimized by CMA-ES, the limping versions have a distinctly asymmetric pattern of contacts. To our untrained eyes it appears very much like a human limp. In the other two Spaces, the contact-timings remain fixed, but there is nonetheless a discernible asymmetry to the motion. While the character has his weight on the right (injured) leg, he lets his body drop slightly by bending his right knee—presumably to lessen the ground reaction force on that foot.

Chapter 6

Conclusion

In this thesis we have presented an architecture for automating character animation in interactive productions like videogames. We take a hybrid approach, combining the physically-based technique of Spacetime Control for generating animation clips, with the data-driven technique of Motion Graphs for real-time interactive playback of them. This architecture avoids many of the problems we perceive in conventional character animation systems. Namely, it does not rely on teams of talented animators or trained motion capture personnel to create each animation individually. Rather, a single animator may specify spaces of animations to be made using parameters. The system then samples and generates animations from these spaces en masse. The procedural nature of our system allows for rapid-prototyping and sweeping design changes. Animations may be quickly added, changed, and removed en masse without extensive re-work. We created a working implementation of this architecture and demonstrated its use in a typical videogame animation scenario. Specifically we showed how standing, walking, running, and stepping motions for a humanoid character could be produced. Additionally we showed how a new design requirement—that the character also perform limping versions of all these actions—could be easily realized.

The animations produced by our implementation are not of the same quality as those produced by talented animators or by motion-capture technology. Given our simplistic bio-mechanical model, we could hardly expect them to be. Nonetheless our results show some compelling humanoid motion, and given the extremely discerning eye that humans have for the nuances of humanoid motion, this bodes well for other morphologies. We suspect that using a more sophisticated bio-mechanical model (e.g. [Yam05]), would result in more natural-looking motion.

Additionally, we have not shown a particularly great breadth or variety of character animations. Our examples have all been from the domain of humanoid locomotion. We chose this domain because it seems most applicable to modern videogames wherein characters are often humanoid and often required to stand, walk, run and jump etc. These sorts of motions also require little in the way of domain-specific modeling (e.g. acting skills, emotions, intricate manipulation of objects etc.) and are therefore well-suited to a coarse physical optimization. Whether Spacetime Control, in general, will extend well into other domains of character animation remains to be seen.

6.1 Future Work

6.1.1 Optimal Contact-Timings

In our system we use CMA-ES to optimize the contact-times of our characters' feet with the ground plane. To do this, we make the assumption that each foot will contact the ground either once, or not at all during each animation. Though reasonably efficient, this approach overly constrains the space of possible solutions, and prevents the creation of certain animations. For instance, our system will not create skipping (characterized by two contacts per foot per cycle). We could add additional variables to the CMA-ES optimization to account for more contacts, but computational efficiency becomes a problem. An interesting direction for future research would thus be to explore alternative encodings and optimization algorithms. Using Compositional Pattern Producing Networks (CPPN) [Sta07] produced by Genetic Algorithms seems like a particularly interesting solution. By exploiting principles like symmetry and repetition, CPPNs may be able to more easily generate a variety of useful contact-timings.

Alternatively, some new research by Erez [Ere11] demonstrates an "invertible" contact model for Spacetime Control. This formulation represents contacts in a fully-continuous way, allowing for the optimization of contacts as part of the regular Spacetime Control optimization. The author demonstrates this approach by creating a run-cycle for a humanoid with 31 degrees of freedom. Using this contact model in our system would make the outer CMA-ES optimization loop unnecessary, and could enable a greater variety of animations to be made.

6.1.2 Self-Intersection

As described in Section 4.2.1, we currently enforce non-intersection between characters' body parts only approximately. A sphere inscribed inside each ellipsoid is forbidden to intersect any of the other ellipsoids on any frame. This is insufficient on two counts. First, it would be preferable to forbid the ellipsoid itself from intersecting any of the other ellipsoids on any frame. We did not pursue this, as it was not immediately evident to us how the necessary constraints could be derived. Several authors offer solutions though [WWK01, AG03]. The second is a more subtle problem. Occasionally we find an animation clip in which the state at frame f is feasible, and the state at $f + 1$ is feasible, but there is no way to reach $f + 1$ from f without an intersection occurring during the brief interval between the two frames. Using smaller time steps would help in this regard, but this quickly becomes computationally expensive, and it only makes an intersection less likely—not impossible. It would thus be interesting to explore more definitive approaches. Using swept-volumes to forbid intersections between frames seems like a likely solution, though the formulation of such equations in a fully-differentiable way may prove difficult.

6.1.3 Transition Animations

In our implementation, we use the Well-Connected Motion Graph approach [ZS09] for generating transitions between individual clips of animation. This approach has worked reasonably well for us. The interpolated animations are computationally inexpensive to produce, usually physically-valid (as shown by Safonova and Hodgins [SH05]), and do not contain foot-sliding errors. However, the approach does suffer from a number of problems. The interpolated animation clips are not *always* physically-valid, and they may also contain serious self-intersection errors (e.g. when interpolating between two clips where the character's hand is in front of his torso in one, and behind his torso in the other). Additionally, the approach tends to create a vast number of animation clips, which greatly slows the creation and processing of the Motion Graph.

As an alternative, it would be interesting to explore the use of Spacetime Control for generating transition animations. Rose et al. [RGBC96] describe such an approach. More recently, Ren et al. [RZS10] describe an approach that we feel is particularly well-suited to our design. The authors use Spacetime Control to generate short transition animations that are chosen to minimize the average transition time between frames in the Motion Graph.

Bibliography

[AF02] Arikan O., Forsyth D. A.: Synthesizing constrained motions from examples. *ACM Transactions on Graphics 21*, 3 (2002), 483–490.

[AF09] Allen B. F., Faloutsos P.: Evolved controllers for simulated locomotion. In *Motion in Games*, vol. 5884. Springer, Berlin, Heidelberg, 2009, pp. 219–230.

[AFO03] Arikan O., Forsyth D. A., O'Brien J. F.: Motion synthesis from annotations. *ACM Trans. Graph. 22*, 3 (2003), 402–408.

[AG03] Alfano S., Greer M. L.: Determining if two solid ellipsoids intersect. *Journal of Guidance, Control, and Dynamics 26*, 1 (Jan. 2003), 106–110.

[Alc72] Alcorn A.: *Pong*. Atari Incorporated, 1972.

[Ber99] Bertsekas D. P.: *Nonlinear Programming*, 2nd ed. Athena Scientific, Belmont, Massachusetts, 1999.

[Ble11] Blender Development Team: *Blender: Open source 3D content creation suite*. The Blender Foundation, 2011. `http://www.blender.org/` (accessed Aug. 22, 2011).

[Bou92] Boute R. T.: The euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems 14*, 2 (Apr. 1992), 127–144.

[CBOP09] Clune J., Beckmann B., Ofria C., Pennock R.: Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *CEC 2009, Trondheim, Norway, May 2009* (May 2009), IEEE Press.

[CBvdP10] Coros S., Beaudoin P., van de Panne M.: Generalized biped walking control. In *ACM Transactions on Graphics (TOG)* (New York, NY, USA, July 2010), vol. 29, ACM, pp. 130:1–130:9.

[Cha11] Champandard A. J.: *The AI Sandbox^{TM}*. AiGameDev.com, 2011. `http://aisandbox.com/` (accessed Aug. 14, 2011).

[Ere11] Erez T.: *Optimal Control for Autonomous Motor Behavior*. Ph.D. thesis, Washington University, Saint Louis, Missouri, May 2011.

[FGK90] Fourer R., Gay D. M., Kernighan B. W.: A modeling language for mathematical programming. *Management Science 36*, 5 (May 1990), 519–554.

[FP03] Fang A. C., Pollard N. S.: Efficient synthesis of physically valid human motion. In *ACM SIGGRAPH 2003 Papers* (San Diego, California, 2003), ACM, pp. 417–426.

[FPT01] Faloutsos P., Panne M. v. d., Terzopoulos D.: Composable controllers for physics-based character animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 251–260.

[Gam07] GameTrailers: Assassin's Creed - animations interview [video, interview with Elspeth Tory], Oct. 2007. `http://www.gametrailers.com/video/animations-interview-assassins-creed/26510` (accessed Aug. 20, 2011).

[GMS05] Gill P. E., Murray W., Saunders M. A.: SNOPT: an SQP algorithm for Large-Scale constrained optimization. *SIAM Rev. 47*, 1 (2005), 99–131.

[Gol80] Goldstein H.: *Classical Mechanics*, 2nd ed. Addison-Wesley, Reading, Massachusetts, July 1980.

[Gut04] Gutta P.: *An Investigation Towards Robot Balancing Using Recurrent Neural Networks and Evolutionary Algorithms*. M.Sc. thesis, Chalmers University of Technology, Goteborg, Sweden, 2004.

[HNR68] Hart P. E., Nilsson N. J., Raphael B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2 (July 1968), 100–107.

[HO96] Hansen N., Ostermeier A.: Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation, 1996* (May 1996), IEEE, pp. 312–317.

[HWBO95] Hodgins J. K., Wooten W. L., Brogan D. C., O'Brien J. F.: Animating human athletics. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), SIGGRAPH '95, ACM, pp. 71–78.

[KGP02] Kovar L., Gleicher M., Pighin F.: Motion graphs. In *ACM Transactions on Graphics (TOG)* (New York, NY, USA, July 2002), vol. 21, ACM, pp. 473–482.

[Kin98] Kines M.: Planning and directing motion capture for games. *Game Developer Magazine* (1998). `http://www.gamasutra.com/features/20000119/kines_01.htm` (accessed Aug. 29, 2011).

[LCL06] Lee K. H., Choi M. G., Lee J.: Motion patches: building blocks for virtual environments annotated with motion data. In *ACM Transactions on Graphics (TOG)* (New York, NY, USA, 2006), SIGGRAPH '06, ACM, pp. 898–906.

[LCR*02] Lee J., Chai J., Reitsma P. S. A., Hodgins J. K., Pollard N. S.: Interactive control of avatars animated with human motion data. In *ACM Transactions on Graphics (TOG)* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 491–500.

[LHP05] Liu C. K., Hertzmann A., Popović Z.: Learning physics-based motion style with nonlinear inverse optimization. *ACM Trans. Graph. 24*, 3 (2005), 1071–1081.

[LvdPE96] Laszlo J., van de Panne M., Eugene F.: Limit cycle control and its application to the animation of balancing and walking. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 155–162.

[LWB*10] Lee Y., Wampler K., Bernstein G., Popović J., Popović Z.: Motion fields for interactive character locomotion. In *ACM Transactions on Graphics (TOG)* (New York, NY, USA, Dec. 2010), vol. 29, ACM, pp. 138:1–138:8.

[MBC01] Mizuguchi M., Buchanan J., Calvert T.: Data driven motion transitions for interactive games. In *Eurographics 2001 Short Presentations* (2001), vol. 2, p. 6.

[MKHK08] Multon F., Kulpa R., Hoyet L., Komura T.: From motion capture to Real-Time character animation. In *Motion in Games*. 2008, pp. 72–81.

[MP07] McCann J., Pollard N.: Responsive characters from motion fragments. In *ACM SIGGRAPH 2007 papers* (San Diego, California, 2007), ACM, p. 6.

[Nis78] Nishikado T.: *Space Invaders*. Taito Corporation, 1978.

[PW99] Popović Z., Witkin A.: Physically based motion transformation. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), ACM Press/Addison-Wesley Publishing Co., pp. 11–20.

[RGBC96] Rose C., Guenter B., Bodenheimer B., Cohen M. F.: Efficient generation of motion transitions using spacetime constraints. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM, pp. 147–154.

[RM01] Reil T., Massey C.: Biologically inspired control of physically simulated bipeds. *Theory in Biosciences 120*, 3 (Dec. 2001), 327–339.

[RP07] Reitsma P. S. A., Pollard N. S.: Evaluating motion graphs for character animation. *ACM Transactions on Graphics 26*, 4 (Oct. 2007), 18.

[RZS10] Ren C., Zhao L., Safonova A.: Human motion synthesis with optimization-based graphs. *Computer Graphics Forum 29*, 2 (May 2010), 545–554.

[SH05] Safonova A., Hodgins J. K.: Analyzing the physical correctness of interpolated human motion. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation - SCA '05* (Los Angeles, California, 2005), p. 171.

[SH07] Safonova A., Hodgins J. K.: Construction and optimal search of interpolated motion graphs. *ACM Trans. Graph. 26*, 3 (2007), 106.

[Sta07] Stanley K. O.: Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines 8* (June 2007), 131–162.

[Str09] Streeting S.: *Ogre3D: Open source graphics rendering engine.* Torus Knot Software, 2009. `http://www.ogre3d.org/` (accessed Aug. 22, 2011).

[Sym10] SymPy Development Team: *SymPy: Python library for symbolic mathematics.* 2010. `http://www.sympy.org/` (accessed Aug. 5, 2011).

[TLP07] Treuille A., Lee Y., Popović Z.: Near-optimal character animation with continuous control. *ACM Transactions on Graphics 26*, 3 (July 2007), 7.

[VM08] Valsalam V., Miikkulainen R.: Modular neuroevolution for multilegged locomotion. In *Proceeedings of the Genetic and Evolutionary Computation Conference* (2008).

[WB06] Wächter A., Biegler L. T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming 106*, 1 (2006), 25–57.

[WK88] Witkin A., Kass M.: Spacetime constraints. *SIGGRAPH Comput. Graph. 22*, 4 (1988), 159–168.

[Woo98] Wooten W. L.: *Simulation of Leaping, Tumbling, Landing, and Balancing Humans.* Ph.D. thesis, Georgia Institute of Technology, 1998.

[WP09] Wampler K., Popović Z.: Optimal gait and form for animal locomotion. *ACM Trans. Graph. 28*, 3 (2009), 1–8.

[WWK01] Wang W., Wang J., Kim M.: An algebraic condition for the separation of two ellipsoids. *Computer Aided Geometric Design 18*, 6 (July 2001), 531–539.

[Yam05] Yamaguchi G. T.: *Dynamic Modeling of Musculoskeletal Motion: A Vectorized Approach for Biomechanical Analysis in Three Dimensions*. Birkhäuser, Sept. 2005.

[ZS09] Zhao L., Safonova A.: Achieving good connectivity in motion graphs. *Graphical Models 71*, 4 (July 2009), 139–152.