Design and Integration of Controllers for Simulated Characters

by

Michael Firmin

B.Sc., Colorado School of Mines, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE in The Faculty of Graduate and Postdoctoral Studies (Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA (Vancouver) November 2014
© Michael Firmin 2014
Abstract

Developing motions for simulated humanoids remains a challenging problem. While there exists a multitude of approaches, few of these are reimplemented or reused by others. The predominant focus of papers in the area remains on algorithmic novelty, due to the difficulty and lack of incentive to more fully explore what can be accomplished within the scope of existing methodologies. We develop a language, based on common features found across physics based character animation research, that facilitates the controller authoring process. By specifying motion primitives over a number of phases, our language has been used to design over 25 controllers for motions ranging from simple static balanced poses, to highly dynamic stunts. Controller sequencing is supported in two ways. Naive integration of controllers is achieved by using highly stable pose controllers (such as a standing or squatting) as intermediate transitions. More complex controller connections are automatically learned through an optimization process. The robustness of our system is demonstrated via random walkthroughs of our integrated set of controllers.
Preface

The CMA-ES algorithm discussed in Chapter 6 was originally created by N. Hansen and A. Ostermeier [14]. The variant adapted for this work originates from [17]. An adapted version of this dissertation has been submitted to the Computer Graphics Forum journal. Figures from this thesis labeled as “used with permission” are copyright and are reused with permission from the authors of the cited papers. The rest of the work is original and developed by the author Michael Firmin, who discussed with Dr. Michiel van de Panne.
# Table of Contents

Abstract ....................................................... ii

Preface ....................................................... iii

Table of Contents ............................................ iv

List of Figures ............................................... vii

1 Introduction ............................................... 1
   1.1 A Brief History of Animation ......................... 1
   1.2 Physics Based Animation ............................. 2
   1.3 Contributions ......................................... 2
   1.4 Organization ......................................... 3

2 Related Work ............................................... 5
   2.1 Controlling Simulated Humanoids ................... 5
   2.2 Controller Transitions ............................... 7
   2.3 Domain Specific Languages ......................... 8

3 Language Features ......................................... 9
   3.1 Motion Primitives ..................................... 9
      3.1.1 Proportional Derivative Control ............... 9
      3.1.2 Virtual Force Control ......................... 11
      3.1.3 Inverse Kinematics ............................ 12
      3.1.4 Joint Symmetry ................................ 13
   3.2 Feedback ............................................. 14
      3.2.1 PD Controller Modification ..................... 14
List of Figures

1.1 ASIMO and BigDog Robots ............................. 3
1.2 Graph of currently implemented controllers ............. 4

2.1 SIMBICON locomotion controllers ....................... 6
2.2 Feature-based control applied to various characters .... 6
2.3 Example of feature based control ........................ 7

3.1 Example of a PD Controller .............................. 10
3.2 Example of virtual forces ................................ 11
3.3 Example of inverse kinematics ........................... 13

4.1 Part listing .............................................. 25
4.2 Joint listing ............................................ 26

5.1 Integrated Controller Graph .............................. 28
5.2 Step up script ........................................... 29
5.3 Step up motion ........................................... 30
5.4 Stand script ............................................. 31
5.5 Standing motion .......................................... 32
5.6 Robustness of backflip controller ....................... 33
5.7 Robustness of locomotion controllers ................... 34
5.8 Motion sequence with environmental features .......... 35
5.9 Sequence of connected controllers ...................... 36
5.10 Random motion sequences .............................. 37
5.11 Selection of controllers ................................ 37

6.1 Walk to forward walkover transition .................... 44
6.2 Skip to walk transition ........................................ 44
6.3 Walk to stand transition ........................................ 44
6.4 Forward walkover to run transition ......................... 45
6.5 Forward walkover with obstacle ............................ 45

7.1 Example from Inventing on Principle ..................... 48
Chapter 1

Introduction

1.1 A Brief History of Animation

Animation techniques have improved significantly since the days of hand-drawn scenes and characters seen in classic movies. Traditional animation, whether done by hand or computer aided, relies on the keyframing technique. In keyframing, a series of key frames are defined that represent the overall motion. Then, the frames in between these key frames are interpolated. In hand drawn traditional animation, the main artist would draw the key frames, while the in-between frames would be filled in by others. Computer aided traditional animation leaves the job of interpolation up to an automatic algorithm.

Keyframing has a few drawbacks: it is tedious, and it lacks physical realism, leaving this in the hands of the artist. In addition, interpolation can create jumpy and awkward transitions between key frames.

A more recent approach to animation is motion capture. In motion capture, motion trajectories of actors are recorded and applied to the animated characters. Because actions are performed by real people, this approach produces physically realistic motions and is also less time-consuming than keyframing for complex motions.

However, motion capture also has a number of shortcomings. The hardware and software to record and edit motion capture data is expensive and therefore not widely available to the public. Small errors in the recorded motion data can be tedious to fix. Also, motions created using motion capture are limited by the actions that can be performed by the actor. To create superhuman motions would require a cumbersome amount of data manipulation. The actor and animated character must also have similar
1.2 Physics Based Animation

A newer approach to animation, physics-based character animation, attempts to solve the issues present in traditional animation and motion capture. In this style of animation, a physically realistic character is created within a simulated world. The character is given realistic masses, joints, joint limits, and other physical properties. Motions are created by controlling the torques at each joint throughout the simulation. This method guarantees a physically realistic animation while allowing characters of any size and shape to be created. However, the difficulty in creating stable and reproducible control strategies has prevented its adoption outside of academia. For example, in the video game industry, physics-based character animation has predominantly been used to create ragdoll motions for characters as no control strategies are needed in this case.

Robotics provides another driving force behind physics-based animation. A physically simulated character can be likened to a robot in that they are both driven by torques generated at the joints. Many of the control strategies used to generate motions for a simulated character could be applied to a robot as well. Locomotion controllers have been successfully created for robots such as the bipedal Honda ASIMO [2], and Boston Dynamics’ quadrupedal BigDog [26].

1.3 Contributions

Perhaps the most significant hindrance with physics-based animation is the lack of a unified standard for developing controllers. While there exists a multitude of approaches, many are never reused or reimplemented by others. This work strives to correct this by

1. Designing a simple language to facilitate the authoring of a wide range of controllers for a planar character
2. Creating a collection of controllers (see Figure 1.2) and motion sequences using them.

3. Providing an optimization framework to

   (a) learn new transitions between controllers
   (b) stylize or smooth existing controller transitions
   (c) determine how to modify a given controller or transition to adjust to perturbations in the environment, character state, or goal trajectory.

1.4 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we discuss related work in the physics-based character animation and robotics fields. Chapter 3 discusses the desired features of a control language. A detailed description of our control language is given in Chapter 4. Next, Chapter 5 presents the controllers and motion sequences authored using our language. In Chapter 6, we describe our optimization framework and results obtained using it. Chapter 7 serves as a conclusion and provides directions for future work. Appendix A presents the scripts for many of the controllers authored in our language, while Appendix B describes how the hierarchy of controllers is represented and stored at runtime. Finally,

Figure 1.1: ASIMO and BigDog Robots. Left: Honda ASIMO, Right: Boston Dynamics’ BigDog
the Accompanying Video demonstrates many of our controllers, motion sequences, and optimization results.

Figure 1.2: Graph of currently implemented controllers. Diamond nodes represent dynamic actions while rectangular nodes are static poses. Solid arrows are naïve transitions while white arrows are a result of the optimization scheme, and dashed arrows are corrective transitions that are utilized upon failure of a controller.
Chapter 2

Related Work

There now exists a wealth of literature on controlling motion for simulated humanoids. Our discussion is focused on three topics. First, we describe the common control primitives and techniques used by the physics-based animation and robotics communities. Next, we describe previous work toward generating controller transitions and longer motion sequences. Finally, we describe how domain specific languages have contributed to other fields.

A full review of the current state of physics-based character animation is beyond the scope of this thesis, and can be found in [11]

2.1 Controlling Simulated Humanoids

One classic approach to designing controllers is to break complex motions into simpler motion phases. This idea is commonly expressed as a finite state machine and can be seen in many related works [16, 18, 19, 27]. In SIMBICON [37], Yin et al create a number of locomotion controllers by splitting more complex motions into a small number of phases, which, when repeated cyclically, generate robust motions.

Manipulation of a character’s joint space has been a common approach to physics-based simulation. Proportional derivative controllers, which produce a torque to achieve a user specified angle at each joint, can be seen in many works [15, 16, 37]. Virtual force control through Jacobian transpose has been crucial in the development of controllers for both robotics [24, 30] and physics-based animation [5, 13]. While many existing control frameworks operate primarily in joint space, others attempt to abstract motion into high-level tasks or features. In feature-based control, motions are tied to character features such as the center of mass or global angle of rotation.
This approach generally involves an underconstrained inverse dynamics problem, in which an optimization, constrained to the laws of motion, is solved at every timestep with objective functions related to the high level features. The free variables at each step are the spatial body accelerations, joint angular accelerations, and control torques. As this method uses abstract concepts, such as the center of mass, it can be extended to characters of varying shapes and sizes, as shown in Figure 2.2. Another approach involves the creation of physics-based controllers from motion captured data and has been explored in [7, 19, 29]. Our approach attempts to improve upon previous works by unifying the common features seen in physics-based character animation into an intuitive language for authoring controllers.

Figure 2.2: Feature-based control applied to various characters. Feature-based control can be easily extended to characters of varying shapes and sizes. Reproduced from [9] with permission.
2.2 Controller Transitions

While a great deal of research has been put into creating robust controllers for various motions, significantly less work has been put into finding stable transitions between them, especially for motions with little to no similar features. The learning of a designated transition controller from motion capture data has been explored in Sok et al’s [29]. In [19], Lee et al.’s controllers track reference motion capture data. Controller sequencing is supported by combining two reference motions, and warping the second to create a smooth transition.

In [6], Coros et al. transition between similar locomotion controllers for quadrupeds by linearly interpolating between the two. Recent works including [5, 22, 31, 36, 37] show how successful transitions between similar locomotion controllers can emerge with little extra work.

However, transitions between more dynamic or different controllers can pose a much harder problem. Ha et al. explored the idea of planning out a sequence of poses in order to transition to the desired pose at the beginning of a new action in [13]. The works of [10] and [35] focus on creating transitions between a small set of carefully designed controllers. In the former, Faloutsos et al. attempt to determine a set of pre-conditions for a given controller that would result in a successful transition to it. Transitions be-
between the complex, highly dynamic tasks of running and obstacle clearance are explored in Liu et al’s Terrain Runner [20]. In [12], Ha et al. create transitions through a concatenation of separate controllers. Here, they optimize the controller to be transitioned to with respect to a family of transitioning controllers, then choose the most likely candidate controller from the family.

Creating robust transitions between different controllers can allow a physically simulated character to be manipulated through higher-level, task based control. For instance, the user can specify a simple task such as walking to a given point, and the framework would figure out a sequence of controllers to achieve this goal. This idea is explored by Coros et al. in [4], using locomotion controllers as input. Alternatively, da Silva et al achieve composition of controllers using an interpolation scheme, linear Bellman combination, in [8].

In general, these approaches have been limited to connecting controllers that are similar in style or structure, such as locomotion controllers. Little work has been done in transitioning between highly dynamic motions, or motions that are drastically different in style.

2.3 Domain Specific Languages

Existing domain specific languages or tools have been shown to greatly influence their fields by simplifying the design process, and engaging a much broader community into the problem solving process. Examples of this include Renderman [32] for rendering problems and FoldIt [3] for protein folding. The Robot Operating System [25] is a general set of software libraries and tools for helping to build robot applications.
Chapter 3

Language Features

In our work, we aim to consolidate common features into one easy to use scripting language that can be used to author a wide variety of motions for a simulated humanoid. In the following sections, we describe the basic structure of controllers designed in our language, discuss the motion primitives we have chosen to include, and present various feedback rules that have been incorporated. This chapter presents an overview of the concepts used in designing our language; for an in-depth description of the language and its syntax, refer to Chapter 4.

3.1 Motion Primitives

An important consideration when designing a language is the choice of features that will serve as the basic control primitives. We want these features to be simple enough that somebody without extensive knowledge in the field can use them effectively, but still expressive enough so that the language can achieve its purpose. For our language, these primitives correspond to the basic actions that the character can take in any given phase. In addition, we would like these primitives to be based on concepts that are already widely utilized by the motion control community. Here, we describe many of the simple motion primitives that a user can specify in our language.

3.1.1 Proportional Derivative Control

The Proportional Derivative (PD) Controller is denoted by

\[ \tau_c = k_p(\theta_a - \theta_d) - k_d\dot{\theta}_a \]
where $\tau_c$ is the calculated control torque for a joint, $\theta_a$ the joint's current angle, $\theta_d$, the desired angle, and $k_p, k_d$ position and velocity gains, respectively. PD controllers have long been one of the primary motion primitives used in control research. They are simple enough to allow novice users to specify a desired joint angle in the local frame, while also allowing expert users to tweak the gains to attain better results. Our PD Controller implementation also allows the user to specify a time for the desired angle to be realized. The system linearly interpolates the desired joint angle between the initial and goal angles over the given time.

Additionally, the user can specify a desired angle for a given body part with respect to the world frame. An example usage of this is to keep the body upright by specifying a desired angle on the upper torso. The user specifies which joint(s) should be active to help realize the desired angle.
3.1.2 Virtual Force Control

The Virtual Force primitive allows users to specify a desired force upon a given body part. The character achieves this force virtually by manipulating internal control torques along a chain of joints between the given part and a specified base joint. The Jacobian Transpose is used to determine control torques based on the desired virtual force:

\[ \tau_c = J^T F \]

Virtual Forces abstract control so that users can specify a desired force in Cartesian coordinates. This can be extended to balance and speed control, gravity compensation, or forces specified on end effectors.

![Figure 3.2: Example of virtual forces. VF Controller acting on the legs to apply an upward force at the center of mass](image)

The equation for virtual force is derived by relating the power \( P \) generated by a force \( F \) along a chain of joints

\[ P = F^T \cdot \nu \]
for \( v \) the velocity at the point where the force is applied. Given that this same power can be generated internally by joint torques \( \tau \), we have

\[
P = \tau^\top \cdot \omega
\]

for angular velocity \( \omega \). Equating the two, along with the relation \( v = J \cdot \omega \), gives

\[
P^\top \cdot v = \tau^\top \cdot \omega
\]

\[
J^\top \cdot J = \tau^\top
\]

(3.1)

By creating simple linear control laws to adjust the desired force, virtual forces can be easily extended to feedback laws for achieving balance, velocity control, and gravity compensation.

For the 2D example in Figure 3.2, we have

\[
J = \begin{bmatrix}
\frac{\partial v_x}{\partial \omega_a} & \frac{\partial v_x}{\partial \omega_b} & \frac{\partial v_x}{\partial \omega_c} & \frac{\partial v_x}{\partial \omega_d} \\
\frac{\partial v_y}{\partial \omega_a} & \frac{\partial v_y}{\partial \omega_b} & \frac{\partial v_y}{\partial \omega_c} & \frac{\partial v_y}{\partial \omega_d}
\end{bmatrix}
\]

(3.2)

where \( v_x \) and \( v_y \) are the velocities at the point where we wish to apply the force, in this case the center of mass, and \( \omega_a, \omega_b, \omega_c, \omega_d \) are the angular velocities of the waist, hip, knee, and ankle, respectively.

Equation 3.2 reduces to

\[
J = \begin{bmatrix}
y_a - y_F & y_b - y_F & y_c - y_F & y_d - y_F \\
x_F - x_a & x_F - x_b & x_F - x_c & x_F - x_d
\end{bmatrix}
\]

where \((x_F, y_F)\) is the point in Cartesian coordinates where the force is applied.

### 3.1.3 Inverse Kinematics

Inverse kinematics takes as input a desired location either in world space or relative to a base joint, and computes the joint angles necessary to move a given body part to that location. This provides another level of abstraction
by allowing users to specify a target location in Cartesian coordinates.

Our system uses cyclic coordinate descent in order to determine the necessary angles along a chain of joints between the part being moved and a specified base joint. In cyclic coordinate descent, the position objective is minimized by repeatedly optimizing each joint angle variable along the chain, one at a time. A detailed description of this technique can be found in [34]. The optimal angles are then achieved using PD Controllers to produce necessary joint torques.

![Figure 3.3: Example of inverse kinematics. IK controller acting to bring the right hand and left foot to the red dots](image)

### 3.1.4 Joint Symmetry

The symmetry primitive allows the user to specify that a given joint should match the angle of another. This is useful when the exact angle of the symmetric counterpart is unknown, such as when it is adjusted by feedback laws.
3.2 Feedback

Another desired feature for control is the ability to provide more global real time feedback. We accomplish this in two separate ways. The first is a simple PD Controller modifier based on the joint space feedback rules used in SIMBICON [37], while the second, based on virtual forces, provides feedback in the Cartesian frame.

3.2.1 PD Controller Modification

The PD Controller modifier is given by

$$\theta_d = \theta_{d0} + c_d d + c_v v$$

where $\theta_d$ is the modified desired joint angle for the PD controller, $\theta_{d0}$ the initial desired angle, $d$ the horizontal distance between center of mass and a given body part, $v$ the center of mass velocity, and $c_d, c_v$ gain parameters. This form of feedback control is used primarily in the SIMBICON-like locomotion scripts.

3.2.2 Virtual Force Feedback

The second type of feedback employs virtual forces. Virtual force feedback can be used to control both character balance and velocity. For balancing, we use a simple linear controller

$$F = c_d d + c_v v$$

to determine a virtual force to apply to the center of mass. This force is realized virtually, using the joint chain running between the ankles and upper torso. Here, $d$ represents the distance between the center of mass and the center of pressure, $v$ is the center of mass velocity, and $c_d$ and $c_v$ are gain parameters. By changing the definition of $d$, we can use the same feedback law to direct the character’s center of mass to other desired locations, such as directly above one of the ankles. This idea is shown in
the stair climbing scripts by using virtual forces to reposition the center of mass over the leading foot during double stance.

Similarly, by manipulating $v$, we can regulate the character’s velocity. Here, we replace $v$ with $(v_d - v_a)$, where $v_d$ is a desired velocity, and $v_a$ the current velocity. This will create a virtual force on the center of mass, attempting to ‘push’ or ‘pull’ the character until it reaches the desired velocity.

3.3 Hierarchical Phase-Based Structure

One of the most prevalent features in controller design is the finite state-machine. In this, a complex motion is broken down into a number of simpler motions, or phases. For example, a backflip motion could be broken into a squat phase followed by jumping, aerial, and landing phases.

For this reason, short motion phases represent the fundamental unit of a controller designed in our language. In each phase the user specifies a number of motion primitives, such as a PD controller or Virtual Force on a given joint or body part, any desired feedback laws, and a number of transition conditions for moving on to the next phase.

Another important feature in designing controllers is the ability to easily re-use controller elements. If a user creates a squat motion, and wants to use it in both a backflip and hop motion, this should be simple to achieve. We support this through a hierarchical controller structure, where any given controller can include another. In our previous example, the user would first design the squat controller. Then, that controller can be included as an identical phase in both the backflip and hop controllers.

This idea also facilitates the parameterization of controller scripts. A user can start by designing a simple walking script. Then, by passing in different sets of parameters, he can modify the walk for different styles of walking without having to reimplement the original script. This idea can also be extended to parameterizing for a given feature, such as the center of mass velocity.
3.3.1 Phase Transitions

For a controller model based on finite state machines, the ability to determine when to transition between phases is essential. Our language offers the user a number of different phase transitions they can use:

1. **Time** - The simplest form of transition. Transitions after a given amount of time has elapsed since entering the phase.

2. **Contact** - Switch phases after a specified body part has come into (or broken) contact with the ground or another environmental object. This is useful in situations such as switching from an aerial phase to a landing phase in a jump controller.

3. **Stable** - Switch phases after the character has become stable, i.e. after all joint velocities as well as the center of mass velocity have dropped below some given tolerance. This is useful for highly dynamic phases/controllers where slight disturbances in the input state can cause the controller to fail.

4. **Fallen** - Switch transitions after the character’s torso orientation has passed a given limit. This is useful in determining if the character has fallen or a controller has failed.

5. **Iterations** - The controller will switch to the next phase after a given number of iterations of the phase.

Each phase in a controller supports multiple transitions, which are specified in an order of importance. The controller will check each specified transition condition in order, and if it succeeds, switch to the given phase for that specific transition. This design helps to create branches in a controller, allowing the character to choose different actions based on a number of parameters. For instance, in the landing phase of the backflip script, two phase transitions are specified. First, if the character’s upper torso has exceeded a certain global angle range (fallen forward), then the controller switches to phase which calls a crawling script. Second, if the character...
has fallen backward, then the controller switches to a phase which calls the supine script. Finally, if neither fallen transition takes place, the character waits until it is stable and then transitions to a rising phase.
Chapter 4

Language Specifications

We unify the features discussed in the previous chapter into a scripting language. The language is implemented using Boost’s xpressive grammar building library [23]. Scripts are parsed when the system is initially loaded, as opposed to being interpreted on the fly.

In this chapter, we will provide a detailed description of the language for the interested reader. This will involve an in-depth discussion of the scripting format and design choices as well as provided functions and parameters. The more casual reader may prefer to skip to Chapter 5 in which we demonstrate results and controllers we have authored.

For code examples, we will use <...> to specify values defined by the user, and [...] for optional sections of code.

4.1 Script Format

The basic script format uses a hierarchical block structure. The highest level block encloses the remainder of the script.

```
1 [SCRIPTS <script1>, <script2>, ..., <scriptN>;]
2
3 BEGINSCRIPT name(parameters)
4 ...
5 ENDSCRIPT
```

The user specified values here are the name of the script, primarily used for display purposes, and the parameters list, a comma separated list of input parameters. If the script requires no parameters, this is replaced by void.

The optional SCRIPTS statement is where the user lists the filenames of all scripts required for this one to be executed. This is analogous to the
#include statement found in C and C++. After encountering this line, the system will read in all scripts in this list.

4.1.1 Phases and Transitions

The next lower level of abstraction is the phase block

```
PHASE <n>
<COMMANDS>
ENDPHASE
```

The user specifies the phase number \( n \), a digit used to refer to this phase. The COMMANDS section is made up of transition statements, motion primitives, or a call to another script.

The user can call another script using

```
> <scriptname>(<parameters>);
```

**scriptname** is the filename of the script the user wants to call. This script must have been previously listed in the SCRIPTS statement at the beginning of the file.

**parameters** is again a comma separated list of values to be passed to the new script. It must have the same length as the list of input parameters in the new script’s definition. If the scripts takes no parameters, **void** is used instead. For example, if the user wants to call the stand.s script with a parameter of 0.1 (time spent in the controller, in this case), they would write

```
> stand (0.1);
```

When the system encounters a switch statement, it pushes the called script onto the stack, along with its parameters. When the called script returns, it is popped off the stack, and the calling script continues.

Every phase must also have at least one transition. A transition is defined as

```
TRANSITION to(<N>).after(<type> <parameters>);
```

In this, \( N \) is the phase to transition to, **type** is the type of transition, along with any specified space separated **parameters** for that transition type.
The transition types are as follows:

1. **time <dt>** - The simplest form of transition will switch to the next phase after the amount of time given by dt

2. **contact <part>** - The contact-based transition will execute the phase switch after the given part has contacted the ground or any other object in the environment. A listing of part and joint primitives is given in section [4.1.3](#).

3. **nocontact <part>** - Similar to the contact type, except the transition is made when contact is broken.

4. **stable <tol>** - The stable transition compares the angular velocities of each joint, as well as the center of mass velocity of the character against the specified tolerance tol. The controller will transition to the next phase if all these values are below the tolerance.

5. **fallen <range>** - The fallen transition will transition if the global angle of the upper torso is out of some range. For an angle \( r \) (specified in radians), range is specified as \( > r \) for an upper bound, \( < r \) for a lower bound, and \( \pm r \) for a symmetric range from \(-r\) to \(r\).

6. **anglerange <lower> <upper>** - Similar to the fallen transition, but allows the user to specify a range of angles. When the character’s upper torso leaves this range, the transition is executed.

7. **iterations <N>** - The iterations transition will transition after \( N \) repetitions of the current phase. This is mainly helpful when the associated phase consists of a call to another script.

8. **complete** - Transition after all actions have been performed. This is equivalent to iterations 1. This is again most useful when the associated phase is calling another script. For a phase consisting of standard motion primitives, the complete transition would be immediately executed.
9. **keypress <key>** - The transition will execute after the user presses a key.

10. **keyrelease <key>** - The transition will execute after a given key is released. This, along with **keypress** could potentially be used to create interactive games using our system.

### 4.1.2 Primitives

The motion primitives section of a phase is contained within an action block

```
ACTIONS
  <PRIMITIVE>
ENDACTIONS
```

where **PRIMITIVE** is any one of the following motion primitives.

1. **PD Controller** - Generates a torque to move a given joint to a given angle.

   Form: `POSE <joint>(<angle>[, <kp>, <kd>]);`

   Parameters:

   - **joint**: the joint this PD controller acts on
   - **angle**: the desired angle,
   - **kp, kd**: position and velocity gains, respectively.

2. **PD Controller Feedback Term** - Simbicon style modifier for PD Controllers. Mostly deprecated, having been replaced by Virtual Force Feedback, but left in for the sake of compatibility with old scripts. Adjusts the PD Controller’s desired angle to keep the character’s center of mass located directly above a given part or joint.

   Form: `POSEFB <joint>(<cd> <cv>).on(<part|joint> <on>);`

   Parameters:

   - **joint**: The joint to apply the feedback to
   - **cd, cv**: Position and velocity gain parameters
part|joint: Whether this feedback controller is based on a part or joint
on: The part or joint to keep the center of mass positioned above.

3. **Linearly Interpolated PD Controller** - Generates a torque to move a given joint to a given angle over a given amount of time.

Form: LIPOSE <joint>(<angle>[, <kp>, <kd>]).time(<t>);

Parameters:
- joint: the joint this PD controller acts on
- angle: the desired angle,
- kp, kd: position and velocity gains, respectively.
- t: time over which the angle is interpolated between the initial and desired angles.

4. **Global PD Controller** - A (virtual) PD controller that generates a torque to move a body part to a specified global angle.

Form: VPD <part>(<angle>[, <kp>, <kd>]).joint(<joint>);

Parameters:
- part: The body part to move
- angle: The globally defined angle to move the part to
- kp, kd: Position and velocity gains, respectively
- joint: The joint that generates the torque to move the part

5. **Virtual Force Controller** - Generates a given force internally using Jacobian Transpose

Form: VF (<x,y,z>).on(<part>).by(<joint>);

Parameters:
- x,y,z: The force to generate in Cartesian coordinates.
- part: The part that should apply the force.
**joint**: The base joint. The joint chain between this and the specified **part** will be used to generate the desired force.

6. **Inverse Kinematics Controller** - Generates joint angles to move a given body part to a given spatial location, specified locally or globally.

   Form: `IK <part>.target<local|global>(<x,y,z>) .base(<joint>) [.tolerance(<tol>)];

   Parameters:
   - **part**: The part to be moved
   - **local|global**: Whether the target is specified in global or local (to the base joint) coordinates.
   - **x,y,z**: The position to move the given part to.
   - **joint**: The base joint. The chain of angles used to move the part is from this to the given part.
   - **tol**: Tolerance, how close the part needs to be to the specified location before stopping the iteration process.

7. **Virtual Force Feedback** - Balance feedback generated through virtual forces. Applies a force to the given part in order to move the center of mass above a given point.

   Form: `VFFB <part>({cd}, {cv}).by(<joint>).over(<part|joint|cop>);

   Parameters:
   - **part**: The part the force is applied to. Generally the upper torso for full body balance.
   - **cd, cv**: Position and velocity gains, respectively.
   - **joint**: The base joint for the virtual force controller.
   - **part|joint|cop**: The part or joint to move the center of mass above. Or ‘cop’ for the center of pressure.

8. **Match Angle** - Applies a PD controller to match this joint to another, adding a simple method for requiring symmetry.
Form: MATCH <joint1>.to(<joint2>[,<kp>,<kd>][.time(<t>)]);

Parameters:

joint1: The joint to apply the PD controller to.

joint2: The joint whose angle should be matched.

kp, kd: Position and velocity gain parameters, respectively.

t: If a time is specified, a linearly interpolated PD Controller is used instead.

9. **Virtual Force Speed Feedback** - Uses virtual forces to help control center of mass velocity

Form: VSFSB <part>(<kd>).by(<joint>).speed(<speed>);

Parameters:

part: The part the force is applied to. Generally the upper torso.

kd: Gain parameter

joint: The base joint for the virtual force controller.

speed: The desired center of mass velocity

The following commands are controller level commands, and *are not* enclosed in an ACTIONS...ENDACTIONS block. They are generally the only command in a phase that transitions on completion.

1. **Save state** - Saves the character state, including center of mass position and velocity, and joint angles and angular velocities.

Form: SAVE <name>;

Parameters:

name: The name of the output state file.

2. **Load state** - Loads the character state, and resets all internal and external torques and forces.

Form: LOAD <name>;
Parameters:

name: The name of the state file to load.

4.1.3 Key Term Listing

Part Listing

The names used to refer to each body part are shown in Figure 4.1.

1. head - Head
2. neck - Neck
3. uTorso - Upper Torso
4. rArm - Right Arm
5. lArm - Left Arm
6. rForearm - Right Forearm
7. lForearm - Left Forearm
8. rHand - Right Hand
9. lHand - Left Hand
10. lTorso - Lower Torso
11. Pelvis - Pelvis
12. rThigh - Right Thigh
13. lThigh - Left Thigh
14. rShin - Right Shin
15. lShin - Left Shin
16. rFoot - Right Foot
17. lFoot - Left Foot

Figure 4.1: Part listing

Joint Listing

The names used to refer to each joint are shown in Figure 4.2.

1. neck2head - Joint connecting neck and head
2. uTorso2neck - Joint between uTorso and neck
3. rShoulder - Right Shoulder, connects rArm and uTorso
4. **lShoulder** - Left Shoulder, connects **lArm** and **uTorso**
5. **rElbow** - Right Elbow, connects **rForearm** and **rArm**
6. **lElbow** - Left Elbow, connects **lForearm** and **lArm**
7. **rWrist** - Right Wrist, connects **rHand** and **rForearm**
8. **lWrist** - Left Wrist, connects **lHand** and **lForearm**
9. **1Torso2uTorso** - Joint connecting **1Torso** and **uTorso**
10. **pelvis2lTorso** - Joint connecting **pelvis** and **1Torso**
11. **rHip** - Right Hip, connects **rThigh** and **pelvis**
12. **lHip** - Left Hip, connects **lThigh** and **pelvis**
13. **rKnee** - Right Knee, connects **rShin** and **rThigh**
14. **lKnee** - Left Knee, connects **lShin** and **lThigh**
15. **rAnkle** - Right Ankle, connects **rFoot** and **rShin**
16. **lAnkle** - Left Ankle, connects **lFoot** and **lShin**

Figure 4.2: Joint listing
Chapter 5

Integrated Controller Graph

Our language has been used to author over 25 controllers for motions ranging from simple poses such as a balanced stand to dynamic motions such as a backflip or kip. Figure 5.1 shows all of our controllers, as well as how they may be connected. In section 5.1, we present the scripts for a selection of the controllers we’ve developed, and describe the workflow to produce them. For more scripts, refer to Appendix A. In section 5.3, we discuss how controllers may be naively connected to create long sequences of motion.

Our controllers are simulated using the open source physics engine, Open Dynamics Engine (ODE) [28].

5.1 Example Controllers

5.1.1 Step-Up

We now show an example script for a step-up motion and explain the thought process and workflow used to design it. The finished motion can be seen in Figure 5.3.

In simple english, we wish to accomplish the following:

1. Stand in a balanced state
2. Lift the right foot up
3. Move the right foot over the step
4. Move the right foot down onto the step

The script for this motion is shown in Figure 5.2.

The first line in the script lists all of the controllers called by this script, which lets the system know it should load these as well. This is comparable
Figure 5.1: Integrated Controller Graph. Diamond nodes represent dynamic actions while rectangular nodes are static poses. Solid arrows are naïve transitions while white arrows are a result of the optimization scheme, and dashed arrows are corrective transitions that are utilized upon failure of a controller.

to an `#include` statement in C. In this case, we include the stand script, because we would like to start and end from it, thereby extending the range of controllers that can be connected to this one through direct transition.

The actual script begins on line 3. It is given a name and a list of parameters—in this case, \( x \) and \( y \)—which correspond to the height of the block that the character will step onto.

In the first phase—lines 4 through 7—the system calls the standVFFB.t (stand with virtual force feedback) script, with an input parameter 0.1. When the stand script has finished running, the system transitions to phase
SCRIPTS standVFFB.t;
BEGINSCRIPT step_up(x, y)
PHASE 0
> standVFFB.t (.1);
TRANSITION to(1).after(complete);
ENDPHASE
PHASE 1
ACTIONS
IK rFoot.targetlocal(0.1, -.6, 0).base(rHip);
VFFB uTorso(3000. , 250.).by(lAnkle).over(cop);
ENDACTIONS
TRANSITION to(2).after(time .2);
ENDPHASE
PHASE 2
ACTIONS
IK rFoot.targetglobal(x,y,-10)
.base(rHip)
.tolerance(0.001);
VPD rFoot(0.0, -300, -30).joint(rAnkle);
VFFB uTorso(3000, 250).by(lAnkle).over(lFoot);
ENDACTIONS
TRANSITION to(3).after(time .3);
ENDPHASE
PHASE 3
ACTIONS
POSE rHip(0.0).time(5);
VPD rFoot(0.0, -1000, -100).joint(rAnkle);
VFFB uTorso(3000, 250).by(lAnkle).over(lFoot);
ENDACTIONS
TRANSITION to(4).after(contact rFoot);
ENDPHASE
...

Figure 5.2: Step up script

1 of the original step up controller.

In the second phase, we would like to begin raising the right foot so that it can be placed on the next stair. This could be done in a number of ways. We could specify PD controllers on the right hip, knee, and ankle, but this would lead to much trial and error trying to place the foot exactly where we want it. Another method would be to specify an upward force on the right foot using virtual forces. This only requires one parameter, but it would be difficult to determine when we would like to stop moving the foot upward. We choose to use the inverse kinematics primitive, shown on line 10. This allows us to specify a (x,y,z) location to move the foot to—in this case, given
in the local coordinate frame of the right hip.

To ensure that the character remains balanced, we also include a virtual force feedback primitive (line 11). We use parameters for position and velocity gain of 3000 and 250 respectively, and specify that the character should use the chain of joints between the center of mass and left ankle in order to keep the center of mass positioned over the center of pressure. We specify a transition out of this phase after a sufficient time duration has passed to bring the foot upwards.

Now that the right foot has been raised, we would like to position it over the step. Here, we again use inverse kinematics (line 17). This time, the goal location is specified in global coordinates. The script parameters $x$ and $y$ represent the position of the top center of the step, in global Cartesian coordinates. Since the inverse kinematics primitive does not control the global angle of the base joint, we also include a virtual, or global, PD controller (line 20). We specify the desired angle—0.0, or horizontal—so that the foot may remain oriented correctly with the block.

In phase 3, we bring the foot down so that it rests flat on the block. This is done by applying a PD controller at the right hip, and transitioning out of the phase as soon as the foot makes contact with the block.

The remainder of the script (which is omitted) shifts the character’s weight to its right foot with a VFFB primitive, then lifts the left foot to join the right in a similar fashion.

Through this process we have designed a successful initial—albeit jerky—stepping up motion, shown in Figure 5.3. The motion can be smoothed or stylized by fine tuning gains and other parameters, either by
hand or through an optimization technique.

5.1.2 Stand

The stand script serves two purposes. First, rise to a simple standing pose; then, balance in that pose until stable.

```
BEGINSCRIPT STAND(dt, dt2)

PHASE 0

ACTIONS

POSE neck2head(0.1000.100.),
    uTorso2neck(0.1000.100.),
    waist(0.600.10.),
    lTorso2uTorso(0.600.60.);

POSE rElbow(0.0.1000.100.),
    lElbow(0.0.1000.100.),
    rWrist(0.0.100.10.),
    lWrist(0.0.100.10.);

LIPOSE rShoulder(0.).time(dt);
LIPOSE lShoulder(0.).time(dt);
LIPOSE rHip(0.0).time(dt);
LIPOSE lHip(0.0).time(dt);

LIPOSE rKnee(0.).time(dt);
LIPOSE lKnee(0.).time(dt);
LIPOSE rAnkle(0.).time(dt);
LIPOSE lAnkle(0.).time(dt);

VFFB uTorso(2000.250.).by(rAnkle).over(cop);
VFFB uTorso(2000.250.).by(lAnkle).over(cop);

ENDACTIONS

TRANSITION to(1).after(time dt2);

ENDPHASE

PHASE 1

ACTIONS

VFFB uTorso(2000.250.).by(rAnkle).over(cop);
VFFB uTorso(2000.250.).by(lAnkle).over(cop);

ENDACTIONS

TRANSITION to(-1).after(stable .01);

ENDPHASE

ENDSCRIPT
```

Figure 5.4: Stand script
This script is divided into two phases. In the first phase, we set simple PD controllers on all joints that do not contribute to the standing motion. This serves the purpose of bringing those joints to a known rest state. Also, linearly interpolated PD controllers are used to move all joints between the ankles and hips to angles of zero, forcing the character upright. The user specified parameter $\Delta t$ controls the speed at which the character achieves this standing pose. Finally, virtual force feedback is used to keep the character balanced during the rising motion.

After the character has risen, he enters the second phase. Here, the only primitives used are feedback laws to keep the character balanced. This phase serves the purpose of waiting until the character has become stable, as specified by the transition. The controller exits after achieving this stability.

Because of the simplicity of this controller, accompanied by the feedback, it is generally successful, as long as the initial character state has the center of mass relatively close to the center of pressure (horizontally). This, combined with the fact that the output character state is fairly consistent, makes the stand controller an integral part of connecting sequences of motions, as discussed in section 5.3. The finished result is shown in Figure 5.5.

A selection of other controllers produced in our language can be seen in Figure 5.11.
5.2 Controller Robustness

There are a number of situations in which a given controller can fail. Perhaps one of the most predominant issues is that of significant perturbations of the initial character state. The domain of initial states which will cause a controller to achieve its desired goal is known as the basin of attraction to that controller.

In general, smoother, slower motions with significant feedback built in have larger basins of attraction, while highly dynamic controllers are more sensitive to their initial state. For instance, the squat and stand controllers generally have very little dynamic motion as the feet never have to leave the ground. Virtual force feedback terms help to control the center of mass and maintain stability.

As it starts and ends from the highly stable squatting state, the backflip is surprisingly stable for small changes in initial state and environment. Figure 5.6 shows the backflip controller performed on various terrain slopes. SIMBICON style locomotion controllers are also very robust, as they have significant feedback to control foot placement. Figure 5.7 shows a number of locomotion controllers over terrain of varying slope.

Figure 5.6: Robustness of backflip controller. Top Left: 7.5 degree slope (successful). Top Right: 8 degree slope (fails initially). Bottom: -4 degree slope (fails on landing)
5.3 Controller Transitions

While our language facilitates the design of scripts for individual motions, connecting these controllers is a more challenging problem.

A naïve approach is to design the majority of controllers to start from a single pose, so that they may be connected by using the given pose as an intermediate controller. For example, when we design the hop motion, the first step is to call the stand script, which consists of a simple pose complete with balance feedback. In addition, it waits until the character has achieved stability for the joints and for center of mass before exiting and returning to the main hop motion. In this way, we guarantee that the hop starts from a very specific initial state, as defined by the stand script. The process of transitioning from another controller to the hop then simply requires designing other scripts to end at the stand pose—or close enough that the stand script will be successful.

As an example, if we wish to connect the walk controller to the hop controller, we use the intermediate walk-to-stop controller, which slowly brings the walk motion to a standstill, after which the stand motion may
then be called. Once the character has reached stability in the stand, the hop can be successfully executed. The stand script is chosen as an intermediate because it is highly stable and provides sufficient feedback. Other scripts, including the squat and SIMBICON-like locomotion scripts, are also highly stable and make for good intermediaries.

Through this process, we have been able to create a number of motion sequences though our controller graph, as shown in Figures 5.9 and 5.10. Figure 5.8 shows a sequence of the step up and hop controllers interacting with the environment.

While this process is often successful, enabling us to create long sequences of connected controllers such as that in Figure 5.1, it often requires small stability tolerances, which leads to lengthy pauses while the character attempts to achieve stability. In addition, it can fail for dynamic motions, which would require unrealistic tolerances.

An alternative is to employ an optimization to automatically create new transitions. This method is explored in the next chapter.
Figure 5.9: Sequence of connected controllers. Top: Hard coded sequence through the integrated controller graph. Bottom: Corresponding simulation.
Figure 5.10: Random motion sequences. Motion sequences produced through random walkthroughs of the integrated controller graph. The first fails on the kip, while the second fails on the handstand.

Figure 5.11: Selection of controllers. Top: Walk, Middle: Backflip, Bottom: Forward Walkover
Chapter 6

Learning Transitions

6.1 Inspiration

In the previous chapter, we discussed how long sequences of motion can be created through a system of transitions between controllers. In general, most controllers are highly dependent upon the initial character state. Because of this, many transitions will either fail or cause the transitioned controller to fail. A naïve way to overcome this is to design a number of transitioning controllers, which end at a stable character state. Then, other controllers can be designed from this output state. By using these highly stable transitioning controllers, long sequences of motion can be created. However, these can still fail for transitions to highly dynamic controllers, and also lead to lengthy pauses while the character achieves stability in joint space.

To overcome this, we propose an optimization framework based on mutating existing controllers through derivative-free evolutionary methods to create new transitions so that longer, more fluid, motion sequences can be easily designed.

Our optimization framework is able to assist with the creation of transitions in a number of ways:

Learn New Transitions

While our integrated controller graph (Figure 5.1) already has a number of transitions between similar controllers (for instance, many of the SIMBI-CON controllers are robust to naïve transitions), the process of designing direct transitions between other motions can be difficult and often produces awkward motions.

Our transition optimization framework provides a simple way to
automatically learn new transitions between controllers. This greatly expands the number of paths that can be taken when creating longer motion sequences.

**Smooth Existing Transitions**

Many of the transitions designed through a direct, naïve approach are awkward and involve long pauses while the character attempts to achieve stability in joint space. Our optimization framework can be used to require minimal energy or time spent in these transitions in order to eliminate the pauses and make the transitions more realistic.

**Correct Perturbed Transitions**

Finally, many of the naïve transitions are only successful for a given initial character state. Even a small perturbation in the character’s state can cause the transition to fail. This is also true in the case of perturbations in the environment, such as when performing the motions on a slope. Our framework can be used to modify the transitioning controller in order to correct the transition for these perturbations.

6.2 Optimization

6.2.1 Unknowns

In our optimization scheme, we are attempting to optimize over specified phases of a given input controller. To do this, we optimize all numerical parameters that are used in the phase specification, in other words, those used in motion primitives, feedback laws, and transitions.

For example, in the PD Controller primitive, the desired joint angle, gains, and duration are all included in the unknowns. In virtual force feedback laws, the gain parameters are subject to optimization. For transitions, the phase time or stability tolerance are both optimized. Integer values such as the number of iterations before transitioning or categorical parameters such as the given body part or joint are not optimized. The set of motion
primitives present is not changed. In this way, we keep the original intent of the controller, and simply modify the exact values used. Our unknown vector $x$ is simply a list of all of these numerical values.

6.2.2 Objective Functions

Our overall function $E$ is given as the weighted sum of a number of individual objective terms

$$E = \sum_{i}^{n} w_i F_i(x)$$

where $F_i$ are individual objective functions and $w_i$ their corresponding weights.

We split the individual objectives into two groups, those that are always present during the optimization, and others that are optional. The primary goal of the first group is for the character to match a known successful trajectory upon switching to the new controller. We define this using as three separate objective terms. The second group helps to provide stylized or smoother transitions.

The first of the objective terms is trajectory of the center of mass. We define this objective term as

$$F_1 = \sum_{t=0}^{n} (c_{a,t} - c_{d,t})^2$$

where $c_{d,t}$ is the position of the center of mass in the known trajectory at time $t$, and $c_{a,t}$ is the position of the center of mass in the current iteration of the optimization.

To avoid possible local minima where the center of mass trajectory is matched, but in a different manner than intended, we add in an objective term to track the global orientation of the character.

$$F_2 = \sum_{t=0}^{n} (\phi_{a,t} - \phi_{d,t})^2$$
where $\phi_{a,t}$ is the angle of the upper torso in world coordinates in the current iteration at time $t$, and $\phi_{d,t}$ is the desired angle at time $t$.

Finally, we also track the local joint angles of the character

$$ F_3 = \sum_{i=0}^{m} \sum_{t=0}^{n} (\theta_{i,a,t} - \theta_{i,d,t})^2 $$

with $\theta_{i,a,t}, \theta_{i,d,t}$ the joint angles of the $i$th joint at time $t$ in the current iteration of the optimization and the desired trajectory respectively.

Together, the objective functions $F_1, F_2, \text{and} F_3$ encode a target trajectory that the optimized controller should follow.

We allow the user to specify extra objective functions, such as limiting the time spent in a phase or minimizing energy cost. For minimizing the time spent in phase $p$, we use the objective function

$$ F_{time} = t_p $$

where $t_p$ is the time spent in phase $p$. The objective function for minimizing energy cost is given as

$$ F_{energy} = t_p \sum_{i}^{\text{joints}} \tau_i^2 $$

where $\tau_i$ is the control torque of joint $i$, and $t_p$ the time spent in phase $p$.

These additional objective functions allow for more stylistically pleasing transitions and tend to be useful in improving existing transitions.

### 6.2.3 Covariance Matrix Adaptation

Our choice of optimization method is a variant of Covariance Matrix Adaptation known as (1+1)-CMA-ES [17]. In (1+1)-CMA, only one offspring is generated for each parent, replacing it if it has a better fitness. Instead of keeping track of an evolution path, as in standard CMA-ES, we keep track of an averaged success rate, and adjust the global step size in accordance with this. If the success rate is low, the step size should be decreased, and vice versa. (1+1)-CMA-ES has the benefit of being easy to implement, while
maintaining the performance of standard, non-elitist CMA-ES.

The full optimization strategy can be summarized as follows: In each generation of the optimization, we create one child set of unknowns based on a normal distribution around the current optimal unknowns and scaled by the (1+1)-CMA-ES step size. After running the simulation with the child parameters, the objective function is evaluated, and if it has a better fitness than the previous optimal, the child set of unknowns replaces the parent. Finally, the success rate is updated and used to determine the new step size. The process is then repeated until the fitness drops below a pre-defined tolerance, or a designated limit for the number of generations is reached.

6.3 Results

For most transitions, the (1+1)-CMA-ES optimization would converge within 200 generations, providing favorable results. An initial step size of 0.002 was used for the majority of motions. The overall time spent optimizing each transition is dependent upon the length of the goal trajectory and the transitioning controller. Goal trajectories were generally between 2 and 5 seconds, leading to optimization times ranging from 10 minutes to an hour. All optimizations were performed on Linux Mint 16 with an Intel Core i5 CPU and an Nvidia GTX285 GPU.

6.3.1 Walk to Forward Walkover

While the transition from the forward walkover into the walk motion (or the majority of SIMBICON-based controllers, for that matter), generally works fine, the reverse naïve transition fails.

Through our optimization framework, we have created a new transition between the walk and forward walkover controllers as in Figure 6.1.

6.3.2 Skip to Walk

During one of the random walkthroughs of the integrated controller graph, we found that the initial character state into the skipping controller had
been perturbed just enough so that when the character tried to transition to the walking motion, the motion failed.

After optimization, the controller was modified to adapt for the perturbations, allowing for a successful transition from the perturbed skip to the walk, shown in Figure 6.2.

### 6.3.3 Walk to Stand

An initial attempt to transition straight from the walking controller results in the character falling over because his feet suddenly stop while the center of mass still has forward momentum. After optimization, the character learns to slow down during the last few cycles of the walk controller, resulting in a successful transition, as shown in Figure 6.3.

### 6.3.4 Forward Walkover to Run

Transitioning from the forward walkover to the run is successful with a naïve approach, but awkward. Upon transitioning, the character begins to run backward in order to regain his balance, before resuming to the standard run. After optimization, the character retains his forward velocity, smoothly entering the run after transitioning. The center of mass trajectory demonstrating this transition is shown in Figure 6.4.

### 6.3.5 Forward Walkover over Box

When a box is added to the scene, the forward walkover controller that works on flat ground is unsuccessful. Using our optimization framework, we adapt the controller so that it still follows the successful trajectory, as shown in Figure 6.5.
Figure 6.1: Walk to forward walkover transition. Top: Initial (failed) attempt. Bottom: Successful optimized transition.

Figure 6.2: Skip to walk transition. The transition between the skip and walk controllers is optimized. Left: Initial (failed) transition. Right: Optimized transition.

Figure 6.3: Walk to stand transition. The transition between the walk and stand controllers is optimized. The top image shows the initial attempt, and the lower image shows the successful transition after optimization.
Figure 6.4: Forward walkover to run transition. The transition between the forward walkover controllers and run controllers is optimized. The center of mass trajectory is shown, with $X$ representing the transition. The naïve approach on the left shows the character running backward slightly to regain his balance. The optimized result is on the right.

Figure 6.5: Forward walkover with obstacle. Optimization of the forward walkover when an obstacle is added. Top: Failed initial attempt. Bottom: Successful modified controller
Chapter 7

Conclusion

Authoring controllers for humanoid motion is still a challenging problem, with no clearly defined standard for their design. Current approaches continue to focus on algorithmic novelty, while the capabilities of existing techniques has still not been explored to its full potential. This work attempts to address these problems by defining a universal controller authoring language that is simple enough to be intuitive, while still being comprehensive, employing many common features of existing approaches. We have demonstrated how the language can be used to design a large number of controllers for a planar humanoid character. We have further shown how these controllers can be integrated through a system of designing and optimizing transition motions.

There is still work that needs to be pursued for a universal motion design language to be even more successful. Crowd sourcing is an obvious future direction. A readily available, web based version of our framework in which novice and professional users alike could design and submit motions would be essential for creating a vast database of controllers. In addition, the framework should be able to automatically detect how newly submitted controllers could be integrated with existing ones in the database. Our framework of learning new transitions between controllers is a start to this, but it is still limited in that it only handles controllers that start from a specified character state. One idea to tackle this problem is to learn a model of the basin of attraction of valid input states for a controller [10]. Another possibility is to interpolate between existing transition controllers that have been shown to be successful for different character states.

The idea of creating controllers from motion captured data has been explored in [7, 19, 29]. This presents another possible future direction for
our work, in combining our optimization framework with motion capture data. Given that a center of mass trajectory could be determined from the mocap data, our system could be used to optimize an existing controller to match the captured motion, as well as create transitions into the motion.

Another interesting direction is to further generalize the language to create primitives at a higher level. An idea similar to Ha and Liu’s control rigs could map lower level controllers such as PD or Jacobian transpose control to more intuitive, human-readable instructions, such as those used by a coach when teaching his students [12]. This could open up the language to novice users who could issue a simple instruction as they would to another human being. Generalizing the language even further could allow the user to specify tasks such as “move to a given location,” allowing the system to automatically find a path through the integrated controller graph which would achieve the desired result.

While generalizing primitives even further will increase the power and ease of designing motions, some motions, such as those that are highly dynamic, will likely still involve tedious parameter tuning to produce a successful, robust motion. In its current state, every time a script in our language is modified, the simulation has to be restarted in order to see the change. For complex or longer motions, this process can be incredibly time consuming. One possible future direction for this work is a live scripting environment, similar to that presented by Bret Victor’s 2012 talk at CUSEC, Inventing on Principle [33]. Victor presents a game involving a simple character, which can be paused at any time, and a ghost trail of the character as it would be simulated is shown.

Then, the user can modify the code that controls the simulation, and the trail updates accordingly. The ability to see what a simple change or addition to a script in real time would have the potential to greatly reduce the time spent designing controllers. This could be extended to simultaneously show the simulated controller for a number of perturbations of the initial character state, which could help to design more robust controllers. Applying these ideas to a physically simulated character could prove more challenging. For instance, every small change in the code would require a
Figure 7.1: Example from Inventing on Principle. The ‘future trail’ of the character updates as the code is changed in real time.

new simulation from the given timepoint, which is far less trivial for a physically simulated N-link character than for the simple sprite shown in Victor’s talk.
Bibliography


Appendices
Appendix A

Scripts

A.1 Stable Pose Scripts

A.1.1 Squat

```plaintext
# File: squatVFFB.s

# Description: Performs a squat motion
# Inputs: dt – Time to achieve and stay in squat pose
BEGINSCRIPT SQUATPOSE(dt)

PHASE 0

ACTIONS

LIPOSE neck2head(0.,1000.,100.).time(dt);
LIPOSE uTorso2neck(0.,1000.,100.).time(dt);
LIPOSE lTorso2uTorso(0.,600.,10.).time(dt);
LIPOSE waist(0.,600.,10.).time(dt);
LIPOSE rShoulder(-1.7).time(dt);
LIPOSE lShoulder(-1.7).time(dt);
LIPOSE rHip(-1.6).time(dt);
LIPOSE lHip(-1.6).time(dt);
LIPOSE rKnee(2.).time(dt);
LIPOSE lKnee(2.).time(dt);
LIPOSE rElbow(-.2,50.,30.).lElbow(-.2,50.,30.);
POSE rWrist(-.2,5.,3.).lWrist(-.2,5.,3.);
VFFB uTorso(2000.,250.).by(rAnkle).over(cop);
VFFB uTorso(2000.,250.).by(lAnkle).over(cop);

ENDACTIONS

TRANSITION to(-1).after(fallen x 1.2);
TRANSITION to(1).after(time dt);

ENDPHASE

PHASE 1

ACTIONS

VFFB uTorso(2000.,450.).by(rAnkle).over(cop);
VFFB uTorso(2000.,450.).by(lAnkle).over(cop);

ENDACTIONS

TRANSITION to(-1).after(fallen x 1.2);
```

55
TRANSITION to (−1). after (stable .01);
ENDPHASE
ENDSCRIPT

A.1.2 Stand

FILE: standVFFB.t
DESCRIPTION: Rises and balances in a stand state
Inputs: dt – Rise duration; dtt – Controller duration
BEGINSCRIPT STANDINGPOSE(dt, dtt)

PHASE 0
ACTIONS
POSE neck2head(0.,1000.,100.), uTorso2neck(0.,1000.,100.),
waist(0.,600.,10.), lTorso2uTorso(0.,600.,60.);
POSE rElbow(0.0,1000.,100.), lElbow(0.0,1000.,100.);
POSE rWrist(0.0,1000.,10.), lWrist(0.0,1000.,10.);
LIPOSE rShoulder(0.).time(dt);
LIPOSE lShoulder(0.).time(dt);
LIPOSE rHip(0.0).time(dt);
LIPOSE lHip(0.0).time(dt);
LIPOSE rKnee(0.).time(dt);
LIPOSE lKnee(0.).time(dt);
LIPOSE rAnkle(0.).time(dt);
LIPOSE lAnkle(0.).time(dt);
VFFB uTorso(2000., 250.).by(rAnkle).over(cop);
VFFB uTorso(2000., 250.).by(lAnkle).over(cop);
ENDACTIONS
TRANSITION to (1). after (time dtt);
ENDPHASE
PHASE 1
ACTIONS
VFFB uTorso(2000., 250.).by(rAnkle).over(cop);
VFFB uTorso(2000., 250.).by(lAnkle).over(cop);
ENDACTIONS
TRANSITION to (−1). after (stable .01);
ENDPHASE
ENDSCRIPT

A.2 Dynamic Scripts

A.2.1 Backflip

FILE: backflip.t
DESCRIPTION: Performs a backflip
Precondition: Squat Pose
#Inputs: void
SCRIPTS balancehighsquat.t, squat.t, rise.t, stand.t, squatVFFB.t;

SET TORQUELIMIT = 1000

BEGINSCRIPT backflip(void)
PHASE 0
   ACTIONS
   POSE rShoulder(-2.2,300.,30.), lShoulder(-2.2,300.,30.);
   VF(-900,3500,0.).on(uTorso).by(rAnkle);
   VF(-900,3500,0.).on(uTorso).by(lAnkle);
   END ACTIONS
   TRANSITION to (1). after (time .2);
ENDPHASE

PHASE 1
   ACTIONS
   POSE rAnkle(-1), lAnkle(-1);
   POSE rKnee(2.3), lKnee(2.3);
   POSE rHip(-2.), lHip(-2.);
   POSE rShoulder(-2.), lShoulder(-2.);
   POSE rElbow(-1.6), lElbow(-1.6);
   POSE waist(.7), lTorso2uTorso(.7);
   END ACTIONS
   TRANSITION to (2). after (contact rFoot);
ENDPHASE

PHASE 2
   TRANSITION to (4). after (time .2);
   ACTIONS
   POSE rAnkle(-.5), lAnkle(-.5);
   POSE rKnee(2.3,1000.,100.), lKnee(2.3,1000.,100.);
   POSEFB rAnkle(2.2,.5).on(part lFoot);
   POSEFB lAnkle(2.2,.5).on(part lFoot);
   END ACTIONS
ENDPHASE

PHASE 3
   > balancehighsquat.t(5);
   TRANSITION to (4). after (complete);
ENDPHASE

PHASE 4
   > squatVFFB.t(2);
   TRANSITION to (-1). after (complete);
ENDPHASE
ENDSCRIPT

A.2.2 Forward Hop

#File: fwdhop.s
# Description: Performs a bunny hop. standVFB.t called to stabilize

# Inputs: void

SCRIPTS standVFB.t;

BEGINSCRIPT fwdhop(void)

PHASE 0
> standVFB.t(1, 1);
TRANSITION to (1). after (complete);
ENDPHASE

PHASE 1
  ACTIONS
  POSE rElbow(-1.7), lElbow(-1.7);
  END ACTIONS
  TRANSITION to (2). after (time 1);
ENDPHASE

PHASE 2 # Start leaning forward
  ACTIONS
  POSE rElbow(-4, 80., 30.), lElbow(-4, 80., 30.);
  POSE rShoulder(.5, 80., 30.), lShoulder(.5, 80., 30.);
  POSE rHip(-6, 80., 30.), lHip(-6, 80., 30.);
  POSE rKnee(.3, 120., 30.), lKnee(.3, 120., 30.);
  POSE rAnkle(-4, 80., 30.), lAnkle(-4, 80., 30.);
  END ACTIONS
  TRANSITION to (3). after (time .5);
ENDPHASE

PHASE 3 # Bend knees
  ACTIONS
  POSE rHip(-1.3,180., 30.), lHip(-1.3,180.,30.);
  POSE rKnee(1.2, 150., 30.), lKnee(1.2,150.,30.);
  POSE rAnkle(-.5, 100., 10.), lAnkle(-.5, 100., 10.);
  END ACTIONS
  TRANSITION to (4). after (time .3);
ENDPHASE

PHASE 4 # Jump!
  ACTIONS
  POSE rShoulder(-.5, 80., 30.), lShoulder(-.5,80.,30.);
  VPD uTorso(.75,300.,30.). joint (lHip);
  VPD uTorso(.75,300.,30.). joint (rHip);
  VF (-200, -3000, 0). on (rFoot). by (rHip);
  VF (-200, -3000, 0). on (lFoot). by (lHip);
  POSE rAnkle(0);
  POSE lAnkle(0);
  END ACTIONS
  TRANSITION to (5). after (nocontact rFoot);
ENDPHASE

PHASE 5 # mid jump
  ACTIONS
  POSE rHip(-0.7,1000.,100.), lHip(-0.7,1000.,100.);
POSE rKnee (1.1, 1000., 100.), lKnee (1.1, 1000., 100.);
POSE rAnkle (-.4), lAnkle (-.4);
POSE rShoulder (-1.7, 80., 30.), lShoulder (-1.7, 80., 30.);
ENDED
ACTIONs
TRANSITION to (7). after (contact rFoot);
ENDED
PHASE 6 # landing
ACTIONs
POSE rHip (-1.1, 1000., 100.), lHip (-1.1, 1000., 100.);
POSE rKnee (1.1, 1000., 100.), lKnee (1.1, 1000., 100.);
POSE rAnkle (-.3, 1000., 100.), lAnkle (-.3, 1000., 100.);
ENDED
ACTIONs
TRANSITION to (7). after (time .2);
ENDED
PHASE 7
> standVFFB.t(1, 1);
TRANSITION to (-1). after (fallen x .6);
TRANSITION to (-1). after (complete);
ENDED
ENDPHASE
ENDSCRIPT

A.2.3 Forward Walkover

FILE: fwdhandspring.s
DESCRIPTION: Performs a forward walkover.
INPUTS: void
SCRIPTS step.t, walk.s;
BEGINSCRIPT fwdhandspring(void)
PHASE 0 # start with a walk step
> step.t(void);
TRANSITION to (1). after (complete);
ENDED
PHASE 1 # start leaning forward
TRANSITION to (2). after (time .5);
ACTIONs
POSE rShoulder (-1.7, 150., 30.), lShoulder (-1.7, 150., 30.);
POSE lKnee (0.05), rKnee (.5);
POSE lHip (.3, 200., 30.), rHip (-1, 200., 30.);
POSE rAnkle (0.), lAnkle (.4);
POSE rElbow (-.5), lElbow (-.5);
ENDED
ACTIONs
PHASE 2 # lean forward
TRANSITION to (3). after (time .2);
ACTIONs
POSE waist (.3), lTorso2uTorso (.3);
POSE rShoulder (-1.8), lShoulder (-1.8);
POSE rWrist (-1.5), lWrist (-1.5);
POSE lKnee (0.05,500,30), rKnee (.3);
POSE rHip (-1.2,200,30), lHip (.3,200,30);
POSE rAnkle (.2), lAnkle (.4);
POSE rElbow (-.2), lElbow (-.2);
ENDACTIONS

PHASE 3 # contact hands to ground
TRANSITION to (4). after (contact lHand);
ACTIONS
POSE rShoulder (-2.3), lShoulder (-2.3);
POSE rWrist (-1), lWrist (-1);
POSE rHip (-1.5), lHip (.3);
POSE rAnkle (.8);
ENDACTIONS

PHASE 4 # kick up
TRANSITION to (5). after (time .3);
ACTIONS
POSE rShoulder (-2.5,500,30), lShoulder (-2.5,500,30),
rElbow (-.5), lElbow (-.5),
rWrist (-1.5), lWrist (-1.5),
rHip (-1.6), lHip (1.6,300,200),
rKnee (0.), lKnee (0.),
rAnkle (.4), lAnkle (0.);
ENDACTIONS

PHASE 5 # balance
TRANSITION to (6). after (time .4);
ACTIONS
POSE waist (-2,300,100), lTorso2uTorso (-2,300,100);
POSE rShoulder (-2.5,300,30), lShoulder (-2.5,300,30),
rElbow (-.8), lElbow (-.8),
rWrist (-1.7), lWrist (-1.7),
rAnkle (.2), lAnkle (.2),
rKnee (.5), lKnee (.5);
POSE rHip (-1.4), lHip (2.3,300,200);
ENDACTIONS

PHASE 6 # prepare for landing
TRANSITION to (7). after (contact lFoot);
ACTIONS
POSE rHip (-.3), lHip (2.,1000,100);
POSE rKnee (.4), lKnee (1.4),
waist (-.4), lTorso2uTorso (-.4),
rShoulder (-2.7), lShoulder (-2.7);
POSE rAnkle (.4), lAnkle (-.8);
PHASE 7 # land
TRANSITION to (8) after (time 1):

ACTIONS
POSE waist (−1.600, 60.), lTorso2uTorso (0.600, 60.);
POSE rKnee (1.3), lKnee (.3), rHip (−.1), lHip (.2);
POSE rAnkle (−.0), lAnkle (−.7);
POSE rShoulder (0., 100., 30.), lShoulder (0., 100., 30.),
   rElbow (0.), lElbow (0.),
   rWrist (0.), lWrist (0.);
ENDACTIONS
ENDPHASE

PHASE 8
> walk.s(void);
TRANSITION to (−1). after (complete);
ENDPHASE
ENDSCRIPT

A.2.4 Kip

#File: kip.t
#Description: Performs a kip.
#Precondition: Supine
#Inputs: void
SCRIPTS squatVFFB.t;
BEGINSCRIPT kip(void)
PHASE 0 #Retract
ACTIONS
POSE neck2head (0., 1000., 100.), uTorso2neck (0., 1000., 100.),
lTorso2uTorso (0., 1000., 100.), waist (0., 1000., 100.),
rWrist (−1.57), lWrist (−1.57);
LIPOSE rAnkle (0.).time (.6);
LIPOSE lAnkle (0.).time (.6);
LIPOSE rKnee (2.8).time (.6);
LIPOSE lKnee (2.8).time (.6);
LIPOSE rHip (−1.4).time (.6);
LIPOSE lHip (−1.4).time (.6);
LIPOSE rShoulder (−2.6).time (.6);
LIPOSE lShoulder (−2.6).time (.6);
LIPOSE rElbow (−2.27).time (.6);
LIPOSE lElbow (−2.27).time (.6);
ENDACTIONS
TRANSITION to (1). after (time 1);
ENDPHASE
PHASE 1 #Retract more, throw feet
ACTIONS
   POSE lKnee (0.), rKnee (0.);
   POSE lHip (-1.7), rHip (-1.7);
   VF (0., 0500., 0).on(rFoot).by(rHip);
   VF (0., 0500., 0).on(lFoot).by(lHip);
ENDACTIONS
TRANSITION to (2). after (time .3);
ENDPHASE

PHASE 2 #Throw Lower Body
ACTIONS
   POSE rAnkle (-1.0), lAnkle (-1.0);
   POSE rKnee(1.505,1000.,100.), lKnee(1.505,1000.,100.);
   POSE rHip (.78), lHip (.78);
   VF (0., -4000., 0.).on(rHand).by(rShoulder);
   VF (0., -4000., 0.).on(lHand).by(lShoulder);
ENDACTIONS
TRANSITION to (3). after (contact rFoot);
ENDPHASE

PHASE 3 #Throw Upper Body
ACTIONS
   POSE waist (0.4,600.,60.);
   POSE rAnkle (-.4,1000.,100.), lAnkle (-.4,1000.,100.);
   POSE rShoulder (-1.7), lShoulder (-1.7);
   POSE rElbow (-0.,50.,30.), lElbow (-0.,50.,30.);
   POSE rWrist (-2,5.,3.), lWrist (-2,5.,3.);
   LIPOSE rHip(-1.1, 1000., 100.).time(.5);
   LIPOSE lHip(-1.1, 1000., 100.).time(.5);
   VFFB uTorso (1800., 100.).by(rAnkle).over(cop);
   VFFB uTorso (1800., 100.).by(lAnkle).over(cop);
ENDACTIONS
TRANSITION to (4). after (time 1);
ENDPHASE

PHASE 4
   > squatVFFB.t (2);
   TRANSITION to (0). after (fallen < -.6);
   TRANSITION to (-1).after (fallen > 1.2);
   TRANSITION to (-1).after (complete);
ENDPHASE
ENDSCRIPT

A.3 SIMBICON Style Scripts

A.3.1 Simbicon Base

#File: simbicononce.t
#Description: Base script for SIMBICON-style motions

62
#Inputs: See SIMBICON paper

BEGINSCRIPT simbicon (dt, cde, cdo, cve, cvo, tor, swhe, swho, swke, swko, stke, stko, ankle)

PHASE 0

ACTIONS

POSE neck2head (0.0), uTorso2neck (0.0);
POSE waist (0.0, 600., 60.), lTorso2uTorso (0.0, 600., 60.);
POSE lWrist (0.0, 5, 3), rWrist (0.0, 5, 3);
POSE rKnee(swke), rAnkle(ankle);
POSE lKnee(stke), lAnkle(ankle);
POSE rShoulder (.3,100,30), rElbow (0,100,30);
POSE lShoulder (-.3,100,30), lElbow (-.4,100,30);
VPD uTorso(tor,300,30.).joint(lHip);
COMFB2 cde cvo j rAnkle
VPD rThigh(swho,-300.,-30).joint(rHip).flags(s);

ENDACTIONS

TRANSITION to (1). after (time dt);

ENDPHASE

PHASE 1

TRANSITION to (2). after (contact rFoot);

ACTIONS

POSE rKnee(swko), lKnee(stko);
VPD uTorso(tor,300,30.).joint(lHip);
COMFB2 cde cvo j rAnkle
VPD rThigh(swho,-300.,-30).joint(rHip).flags(s);

ENDACTIONS

ENDPHASE

PHASE 2

TRANSITION to (3). after (time dt);

ACTIONS

POSE lKnee(swke),
   rShoulder (-.3,100,30), lShoulder (.3,100,30),
   rElbow (-.4,100,30), lElbow (0,100,30);
VPD uTorso(tor,300,30.).joint(rHip);
COMFB2 cde cvo j lAnkle
VPD lThigh(swho,-300.,-30).joint(lHip).flags(s);

ENDACTIONS

ENDPHASE

PHASE 3

TRANSITION to (-1). after (contact lFoot);

ACTIONS

POSE lKnee(swko), rKnee(stko);
VPD uTorso(tor,300,30.).joint(rHip);
COMFB2 cde cvo j lAnkle
VPD lThigh(swho,-300.,-30).joint(lHip).flags(s);

ENDACTIONS

ENDPHASE

63
A.3.2 Two Phase Simbicon Base

```plaintext
#File: simbicon2phaseonce.t
#Description: Base script for 2 phase SIMBICON-style motions
#Inputs: See SIMBICON paper
BEGINSCRIPT simbicon(dt, cde, cdo, cve, cvo, tor, 
     swh, swk, stk, swa, sta)
PHASE 0
  ACTIONS
  POSE neck2head(0.0), waist(0.0), 
     uTorso2neck(0.0), iTorso2uTorso(0.0), 
     lWrist(0.0, 5.3), rWrist(0.0, 5.3), 
     rKnee(swk), rAnkle(swa), lKnee(stk), lAnkle(sta), 
     rShoulder(.1,100,30), rElbow(-1.7,100,30), 
     lShoulder(-.1,100,30), lElbow(-1.7,100,30),
  VPD uTorso(tor,300,300).joint(lHip);
  COMFB2 cde cve j rAnkle
  VPD rThigh(swh,-300,-30).joint(rHip).flags(s);
ENDACTIONS
TRANSITION to(1).after(time dt);
ENDPHASE

PHASE 1
  ACTIONS
  POSE lKnee(swk), rKnee(stk), lAnkle(swa), rAnkle(sta), 
    rShoulder(-.1,100,30), rElbow(-1.7,100,30), 
    lShoulder(.1,100,30), lElbow(-1.7,100,30),
  VPD uTorso(tor,300,300).joint(rHip);
  COMFB2 cde cve j lAnkle
  VPD lThigh(swh,-300,-30).joint(lHip).flags(s);
ENDACTIONS
TRANSITION to(-1).after(time dt);
ENDPHASE
ENDSCRIPT
```

A.3.3 Walk

```plaintext
#File: walkonce.t
#Description: Walk, calls SIMBICON
#Inputs: void
SCRIPTS simbicononce.t;
BEGINSCRIPT walk(void)
PHASE 0
   > simbicononce.t(.3, 0.0, -2.2, -.2, 0.0, 0.0, 
      -.4,.7, 1.1,.05, 0.05, 0.1, -.2);
ENDSCRIPT
```
A.3.4 Fast Walk

# File: fastwalkonce.t
# Description: Fast Walk, calls SIMBICON
# Inputs: void
SCRIPTS simbicononce.t;

BEGINSCRIPT fastwalk(void)
PHASE 0
> simbicononce.t(.2, 0.0, -.2, -.2, 0.0, 0.1, 
 -.73, .7, 1.83, .05, 0.05, 0.1, -.2);
TRANSITION to(-1).after(complete);
ENDPHASE
ENDSCRIPT

A.3.5 Run

# File: runonce.t
# Description: Run, calls SIMBICON 2 phase
# Inputs: void
SCRIPTS simbicon2phaseonce.t;

SET TORQUELIMIT = 370

BEGINSCRIPT run(void)
PHASE 0
> simbicon2phaseonce.t(.2, 0.0, 0.0, -.2, -.2, 
 0.14, -1.08, 2.18, 0.05, -.2, -.27);
TRANSITION to(-1).after(complete);
ENDPHASE
ENDSCRIPT

A.3.6 Skip

# File: skip.t
# Description: Skip
# Inputs: void
SET TORQUELIMIT = 170

BEGINSCRIPT skip(void)
PHASE 0
  ACTIONS
  POSE neck2head(0.0), waist(0.0, 600., 60.),
POSE lWrist (0.0, 5.3), rWrist (0.0, 5.3);
POSE lKnee (1.75), lKnee (.19);
POSE rKnee (-2.0), rKnee (-2.0);
POSE rShoulder (.3, 100, 30), rElbow (0, 100, 30);
POSE lShoulder (-.3, 100, 30), lElbow (-.4, 100, 30);
VPD uTorso (0.05, 300, 30. joint (lHip);
COMFB 0. -0.4 j rAnkle
VPD rThigh (-1.04, -300, -30). joint (rHip). flags (s);

ENDACTIONS
TRANSITION to (1). after (time .19);
ENDPHASE

PHASE 1
TRANSITION to (2). after (time .12);

ACTIONS
POSE rKnee (2.18), lKnee (.05);
POSE lAnkle (1.0);
VPD uTorso (0.05, 300, 30.). joint (lHip);
COMFB 0. -0.4 j rAnkle
VPD rThigh (-2.25, -300, -30). joint (rHip). flags (s);

ENDACTIONS

ENDPHASE
PHASE 2
TRANSITION to (3). after (time 0.26);

ACTIONS
POSE rKnee (2.09), lKnee (.05);
POSE lAnkle (-.2);
VPD uTorso (0.05, 300, 30.). joint (lHip);
COMFB -0.04 j rAnkle
VPD rThigh (-2.44, -300, -30). joint (rHip). flags (s);

ENDACTIONS

ENDPHASE

PHASE 3
TRANSITION to (4). after (contact rFoot);

ACTIONS
POSE rKnee (.25), lKnee (.10);
POSE lAnkle (-.2);
VPD uTorso (0.05, 300, 30.). joint (lHip);
COMFB -.18 -.37 j rAnkle
VPD rThigh (.46, -300, -30). joint (rHip). flags (s);

ENDACTIONS

ENDPHASE

PHASE 4
ACTIONS
POSE lWrist (0.0, 5.3), rWrist (0.0, 5.3);
POSE lKnee (1.75), rKnee (.19);
POSE lAnkle (-.2), rAnkle (-.2);
A.3.7 Walk to Stop

# File: walk2stop.t
# Description: Connects walk and stand.
# Inputs: SIMBICON cd, cv gains.
BEGINSCRIPT stop(cde, cve)
PHASE 0

POSE lShoulder (.3,100,30), lElbow (0.100,30);
POSE rShoulder (-.3,100,30), rElbow (-.4,100,30);
VPD uTorso (0.05,300,30.).joint (rHip);
COMFB2 0. -0.4 j lAnkle
VPD lThigh (-1.04, -300., -30). joint (rHip). flags (s);
ENDACTIONS
TRANSITION to (5). after (time .19);
ENDPHASE

PHASE 5
TRANSITION to (6). after (time .12);
ACTIONS
POSE lKnee (2.18), rKnee (.05);
POSE rAnkle (1.00);
VPD uTorso (0.05,300,30.).joint (rHip);
COMFB2 0. -.4 j lAnkle
VPD lThigh (-2.25, -300., -30). joint (rHip). flags (s);
ENDACTIONS
ENDPHASE

PHASE 6
TRANSITION to (7). after (time .26);
ACTIONS
POSE lKnee (2.09), rKnee (.05);
POSE rAnkle (-.2);
VPD uTorso (0.05,300,30.).joint (rHip);
COMFB2 0. -.04 j lAnkle
VPD lThigh (-2.44, -300., -30). joint (rHip). flags (s);
ENDACTIONS
ENDPHASE

PHASE 7
ACTIONS
POSE lKnee (.25), rKnee (.10);
POSE rAnkle (-.2);
VPD uTorso (0.05,300,30.).joint (rHip);
COMFB2 -.18 -.37 j lAnkle
VPD lThigh (.46, -300., -30). joint (rHip). flags (s);
ENDACTIONS
TRANSITION to (-1). after (contact lFoot);
ENDPHASE
ENDSCRIPT
TRANSITION to (1). after (time .3);

ACTIONS

POSE neck2head (0.0), waist (0.0, 600., 60.),
    uTorso2neck (0.0), lTorso2uTorso (0.0, 600., 60.);
POSE lWrist (0.0, 5.3), rWrist (0.0, 5.3);
POSE rKnee (−0.5), rAnkle (−0.5);
POSE lKnee (1.3);
POSE lAnkle (−0.2);
POSE rAnkle (−0.2);
POSE rKnee (−0.7);
POSE rHip (0.0, 1000., 100.);
POSE uTorso (1000., 450.). by (rAnkle). over (rFoot);
COMFr2 cde cve j rAnkle
VFD rThigh (−.5,−300.,−30.). joint (rHip). flags (s);

ENDACTIONS
ENDPHASE

PHASE 1

TRANSITION to (2). after (time 1);

ACTIONS

POSE rAnkle (0.0);
POSE rHip (0.0, 1000., 100.);
POSE uTorso (1000., 450.). by (rAnkle). over (rFoot);
VFD rThigh (−.5,−300.,−30.). joint (rHip). flags (s);

ENDACTIONS
ENDPHASE

PHASE 2

TRANSITION to (3). after (time 1);

ACTIONS

POSE neck2head (0.0), waist (0.0, 600., 60.),
    uTorso2neck (0.0), lTorso2uTorso (0.0, 600., 60.);
POSE rAnkle (−0.5);
POSE rKnee (0.0);
POSE rHip (0.0);
POSE uTorso (1000., 450.). by (rAnkle). over (rFoot);
VFD rThigh (−.5,−300.,−30.). joint (rHip). flags (s);

ENDACTIONS
ENDPHASE

PHASE 3

TRANSITION to (−1). after (time 2);

ACTIONS

POSE rAnkle (−0.05);
MATCH lAnkle.to (rAnkle). time (1);
MATCH lHip.to (rHip). time (1);
MATCH lKnee.to (rKnee). time (1);
VFFB uTorso (1000., 450.). by (rAnkle). over (cop);

ENDACTIONS
A.4 Crawl Scripts

A.4.1 Crawl

```plaintext
#File: crawl.t
#Description: High level crawl script.
#Inputs: number of iterations
SCRIPTS stand_to_hak.t, crawl_base_once.t;
BEGINSCRIPT crawl(itors)
PHASE 0
  > stand_to_hak.t(void);
  TRANSITION to(1).after(complete);
ENDPHASE
PHASE 1
  > crawl_base_once.t(.3);
  TRANSITION to(-1).after(iterations iters);
ENDPHASE
ENDSCRIPT
```

A.4.2 Stand to Hands and Knees

```plaintext
#File: stand_to_hak.t
#Description: Transition from stand to Hands and Knees
#Inputs: void
BEGINSCRIPT STAND_TO_HAK(void)
PHASE 0 # start to squat
  ACTIONS
    POSE lTorso2uTorso(0., 600., 60.),
        uTorso2neck(0.), neck2head(0.);
    POSE waist(0.1, 600., 60.);
    LIPOSE rAnkle(-1.4).time(1);
    LIPOSE lAnkle(-1.4).time(1);
    LIPOSE rKnee(1.6).time(1);
    LIPOSE lKnee(1.6).time(1);
    LIPOSE rHip(-1.4).time(1);
    LIPOSE lHip(-1.4).time(1);
    POSE rShoulder(-1.7,50.,30.), lShoulder(-1.7,50.,30.);
    POSE rElbow(-0.,50.,30.), lElbow(-0.,50.,30.);
    POSE rWrist(-2.,5.,3.), lWrist(-2.,5.,3.);
    POSEFB rAnkle(2.2,2.).on(joint lAnkle),
        lAnkle(2.2,2.).on(joint lAnkle);
ENDACTIONS
```
TRANSITION to (2). after (time 1);
ENDPHASE

PHASE 1
TRANSITION to (2). after (time 0.5);
ACTIONS
POSE rAnkle(-1.4), lAnkle(-1.4);
ENDACTIONS
ENDPHASE

# landing with hands

PHASE 2
TRANSITION to (3). after (time 0.8);
ACTIONS
POSE rAnkle(-1.4), lAnkle(-1.4);
POSE lKnee(1.3,300.,100.), rKnee(-1.3,300.,30.);
POSE rShoulder(-1.4,1000.,100.), lShoulder(-1.4,1000.,100.);
POSE rElbow(-1.,1000.,100.), lElbow(-1.,1000.,100.);
POSE rWrist(-.8,5.,3.), lWrist(-.8,5.,3.);
ENDACTIONS
ENDPHASE

PHASE 3
TRANSITION to (-1). after (time 0.4);
ACTIONS
POSE lKnee(1.8,1000.,100.), rKnee(-1.8,1000.,100.);
POSE rAnkle(-1.,1000.,100.), lAnkle(-1.1,1000.,100.);
POSE rElbow(-1.,1000.,100.), lElbow(-1.,1000.,100.);
ENDACTIONS
ENDPHASE

ENDSCRIPT

A.4.3 Crawl Base

# File: crawl_base_once.t
# Description: Perform a crawl cycle
# Inputs: dt – Phase 0.2 duration
BEGINSCRIPT crawl(dt)

PHASE 0
ACTIONS
POSE neck2head(-1.0,1000.,100.), waist(-2.0,1000.,100.),
    uTorso2neck(0.0,1000.,100.), lTorso2uTorso(0.0,1000.,100.);
POSE rHip(-2.3,1000.,100.), lHip(-1.7,1000.,100.);
POSE rKnee(1.95,1000.,100.), lKnee(1.5,1000.,100.);
POSE rAnkle(0.0,1000.,100.), lAnkle(-2.1,1000.,100.);
POSE rShoulder(-1.1,1000.,100.), lShoulder(-0.7,1000.,100.);
POSE rElbow(-0.8,1000.,100.), lElbow(-1.7,1000.,100.);
POSE rWrist(-0.8,1000.,10.), lWrist(-0.5,1000.,10.);
ENDACTIONS
TRANSITION to (1). after (time dt);
ENDPHASE
PHASE 1
ACTIONS
POSE rHip (−2.3, 1000., 100.), lHip (−1.4, 1000., 100.);
POSE rKnee (2.0, 1000., 100.), lKnee (1.1, 1000., 100.);
POSE rAnkle (−7.1000.,100.), lAnkle (−5.1000.,100.);
POSE rShoulder (−0.7,1000.,100.), lShoulder (−1.6,1000.,100.);
POSE rElbow (−0.9,1000.,100.), lElbow (−0.6, 1000.,100.);
POSE rWrist (−9.1000.,10.), lWrist (−6.1000.,10.);
ENDACTIONS
TRANSITION to (2). after (contact lHand);
ENDPHASE
PHASE 2
ACTIONS
POSE rHip (−1.9, 1000., 100.), lHip (−2.3, 1000., 100.);
POSE rKnee (1.6,1000.,100.), lKnee (2.2,1000.,100.);
POSE rAnkle (−7.1000.,100.), lAnkle (0.0,1000.,100.);
POSE rShoulder (−0.7,1000.,100.), lShoulder (−1.05,1000.,100.);
POSE rElbow (−1.6,1000.,100.), lElbow (−0.5, 1000.,100.);
POSE rWrist (−0.3,100.,10.), lWrist (−1.2,100.,10.);
ENDACTIONS
TRANSITION to (3). after (time dt);
ENDPHASE
PHASE 3
ACTIONS
POSE rHip (−1.5, 1000., 100.), lHip (−2.1, 1000., 100.);
POSE rKnee (1.1,1000.,100.), lKnee (1.7,1000.,100.);
POSE rAnkle (−7.1000.,100.), lAnkle (−2.1000.,100.);
POSE rShoulder (−1.5,1000.,100.), lShoulder (−0.7,1000.,100.);
POSE rElbow (−0.7,1000.,100.), lElbow (−0.9, 1000.,100.);
POSE rWrist (−0.8,100.,10.), lWrist (−1.1,100.,10.);
ENDACTIONS
TRANSITION to (−1). after (contact rHand);
ENDPHASE
ENDSCRIPT

A.5 Handstand Scripts

A.5.1 Handstand

#File: handstand.t
#Description: Perform a handstand
#Inputs: dt – time spent in handstand
SCRIPTS step.t;

BEGINSCRIPT handstand (dt)
PHASE 0 # start with a walk step
  > step.t(void);
  TRANSITION to(1).after(complete);
ENDPHASE
PHASE 1 # start leaning forward
  TRANSITION to(2).after(time .5);
  ACTIONS
    POSE rShoulder (−1.7, 150., 30.), lShoulder (−1.7, 150., 30.),
        lKnee(0.05), rKnee(.5),
        lHip (.3, 200., 30.), rHip(-1,200., 30.),
        rAnkle(0.), lAnkle(.4), rElbow(-.5), lElbow(-.5);
ENDACTIONS
ENDPHASE
PHASE 2 # lean forward
  TRANSITION to(3).after(time .2);
  ACTIONS
    POSE waist (.3), lTorso2uTorso (.3);
    POSE rShoulder (−1.6), lShoulder (−1.6),
        rWrist (−1.5), lWrist(−1.5),
        lKnee(0.05, 500., 30), rKnee(.3),
        rHip (−1.2, 200., 30.), lHip(.3, 200., 30.),
        rAnkle(.2), lAnkle(.4),
        rElbow(−.28), lElbow(−.28);
ENDACTIONS
ENDPHASE
PHASE 3 # contact hands to ground
  ACTIONS
    POSE rShoulder (−2.75), lShoulder (−2.75);
    POSE rWrist (−1.3), lWrist (−1.3),
        rHip (−1.2), lHip (.3), rAnkle (.8);
    POSE rElbow (−.6), lElbow (−.6);
ENDACTIONS
  TRANSITION to(4).after(contact lHand);
ENDPHASE
PHASE 4 # kick up
  ACTIONS
    POSE rShoulder (−2.85, 1000., 100.), lShoulder (−2.85,1000.,100.);
    POSE rElbow(−.7, 1000., 100.), lElbow (−.7,1000.,100.);
    POSE rWrist (−1.25, 300., 30.), lWrist (−1.25, 300., 30.);
    POSE rHip (−.9), lHip (−.7, 400., 200.);
    POSE rKnee (−.7, 1000., 100.), lKnee (.7,1000.,100.);
    POSE rAnkle(.0), lAnkle(.0);
ENDACTIONS
  TRANSITION to(5).after(time .3);
ENDPHASE
PHASE 5 # balance
  ACTIONS
POSE waist (0.0,2000,100.), lTorso2uTorso (-0.1,000.100.);
POSE rHip (-7,1000,100.), lHip (-7,1000,100.);
POSE rKnee (1.1000,100.), lKnee (1.1000,100.);
POSE rWrist (-1.2,300.,30.), lWrist (-1.2,300.,30.);
POSEFB rHip (-2.0,0).on (part rHand);
POSEFB lHip (-2.0,0).on (part rHand);
POSEFB rWrist (2.0,1.0).on (part rHand);
POSEFB lWrist (2.0,1.0).on (part rHand);

ENDACTIONS
TRANSITION to (-1).after (anglerange -2 2.8);
TRANSITION to (-1).after (time dt);
ENDPHASE
ENDSCRIPT

A.5.2 Handstand to Supine

#File: handstand2supine.t
#Description: Roll from handstand to supine
#Inputs: void
SCRIPTS stand2handstand.t;

BEGINSCRIPT supine (void)
PHASE 0
ACTIONS
LIPOSE rShoulder (-2.).time (1);
LIPOSE lShoulder (-2.).time (1);
LIPOSE rElbow (-1.57).time (1);
LIPOSE lElbow (-1.57).time (1);
LIPOSE rWrist (-1.57,300.,30.).time (1);
LIPOSE lWrist (-1.57,300.,30.).time (1);
POSE rHip (-1.57), lHip (-1.57);
LIPOSE neck2head (1.).time (1);
LIPOSE uTorso2neck (1.).time (1);
LIPOSE waist (1.).time (1);
LIPOSE lTorso2uTorso (1.).time (1);
ENDACTIONS
TRANSITION to (-1).after (anglerange 0 3);
TRANSITION to (1).after (time 1.2);
ENDPHASE
PHASE 1
ACTIONS
POSE neck2head (0.0), uTorso2neck (0.0),
waist (0.0,600.,60.), lTorso2uTorso (0.0,600.,60.);
POSE rShoulder (0.0), lShoulder (0.0);
POSE rElbow (0.0), lElbow (0.0);
POSE rWrist (0.0), lWrist (0.0);
POSE rHip (0.0), lHip (0.0);
POSE rKnee(0.0), lKnee(0.0);
POSE rAnkle(0.0), lAnkle(0.0);
ENDACTIONS
TRANSITION to(-1).after(anglerange 0 3);
TRANSITION to(-1).after(time 1);
ENDPHASE
ENDSCRIPT

A.5.3 Walk on Hands

#File: handstandwalk.t
#Description: Walk on hands, SIMBICON based
#Precondition: Handstand
#Inputs: void
BEGINSCRIPT walk_on_hands(void)
PHASE 0
ACTIONS
LIPOSE rHip(0.).time(1);
LIPOSE lHip(0.).time(1);
LIPOSE rKnee(0.).time(1);
LIPOSE lKnee(0.).time(1);
LIPOSE waist(0.,1000.,60.).time(1);
VFFB uTorso(1000.,50.).by(rWrist).over(cop);
VFFB uTorso(1000.,50.).by(lWrist).over(cop);
ENDACTIONS
TRANSITION to(-1).after(anglerange -2 2);
TRANSITION to(1).after(time 1);
ENDPHASE
PHASE 1
ACTIONS
VFFB uTorso(2000.,250.).by(lWrist).over(cop);
LIPOSE rElbow(-1.7).time(1);
ENDACTIONS
TRANSITION to(-1).after(anglerange -2 2.8);
TRANSITION to(2).after(time 1);
ENDPHASE
PHASE 2
ACTIONS
VFFB uTorso(2000.,250.).by(lWrist).over(lHand);
LIPOSE rElbow(-3.).time(1);
LIPOSE rShoulder(-2.9).time(1);
LIPOSE rWrist(-5.,300.,30.).time(1);
LIPOSE lShoulder(-2.5).time(1);
ENDACTIONS
TRANSITION to(-1).after(anglerange -2 2.8);
TRANSITION to(3).after(contact rHand);
ENDPHASE
PHASE 3

ACTIONS

VFFB uTorso(1000., 1000.).by(lWrist).over(cop);
VFFB uTorso(1000., 1000.).by(rWrist).over(cop);
ENDACTIONS

TRANSITION to(-1).after(anglerange -2 2.8);
TRANSITION to(4).after(time 1);
ENDPHASE

PHASE 4

ACTIONS

VFFB uTorso(2000., 200.).by(rWrist).over(rHand);
VFFB uTorso(300., 30.).by(lWrist).over(rHand);
ENDACTIONS

TRANSITION to(-1).after(anglerange -2 2.8);
TRANSITION to(5).after(time 1);
ENDPHASE

PHASE 5

ACTIONS

LIPOSE rShoulder(-2.7).time(1);
LIPOSE rWrist(-1.3,300.,30.).time(1);
LIPOSE lElbow(-1.7).time(1);
VFFB uTorso(2000., 600.).by(rWrist).over(rHand);
ENDACTIONS

TRANSITION to(-1).after(anglerange -2 2.8);
TRANSITION to(6).after(time 1);
ENDPHASE

PHASE 6

ACTIONS

VFFB uTorso(2000., 600.).by(rWrist).over(rHand);
MATCH lShoulder.to(rShoulder).time(1);
MATCH lElbow.to(rElbow).time(1);
MATCH lWrist.to(rWrist, 300., 30.).time(1);
ENDACTIONS

TRANSITION to(-1).after(anglerange -2 2.8);
TRANSITION to(-1).after(time 1);
ENDPHASE

ENDSCRIPT
Appendix B

Controller Hierarchy

Here, we briefly describe how the hierarchical system of controllers is represented during execution.

The basic system uses four structures representing the controller, phase, action, and transition. The controller class contains its filename, a list of phases, and the index of the current phase

```java
1  class Controller()
2  {
3      string name;
4      array phases;
5      int curr_phase;
6  };
```

The phase class consists of the phase number, a list of actions to be taken on this phase, and a list of transitions to be checked.

```java
1  class Phase()
2  {
3      int num;
4      array actions;
5      array transition;
6  };
```

Each type of action is descended from the Action class, and must have a performAction function, which determines the torques generated whenever the action is called.

```java
1  class Action()
2  {
3      virtual void performAction();
4      virtual void reset();
5  };
```

Finally, a transition object is descended from the Transition class, and
must have a check() function that evaluates whether the transitioning conditions are true or not, and a phase index to transition to.

```cpp
class Transition {
    virtual void check();
    int to;
};
```

After the controllers are read in, and the basic structures set up, they need to be stored and handled by the system in an efficient manner. There are two primary data structures used, a map for storing all controllers that have been read in, and a stack for storing the current hierarchy of controllers.

The storage map uses the controller’s name as the key, and the controller data structure as the entry. When the system comes across a phase that calls another controller, the new controller is located in the map, and pushed to the top of the stack. After the controller exits, it is popped from the stack, and the previous controller resumes at the phase it left off at.

When the controller stack is empty, the character goes into a ragdoll mode, where no control torques are generated.