Volume xx (200y), Number z, pp. 1–14

Controller Design for Multiskilled Bipedal Characters

M. Firmin and M. van de Panne

University of British Columbia, Canada

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.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Animation—Physical Simulation

1. Introduction

Physics based animation is an approach to animation in which characters, humanoid or otherwise, are simulated within a virtual environment. While it has many benefits over traditional animation techniques—keyframing and motion capture—there are still many obstacles to overcome before it can be adopted by the animation community at large.

One predominant issue is that while there are a multitude of approaches to designing new and complex motions, few are reused, reimplemented, or integrated with existing techniques. For example, many papers have explored—in depth—the creation of complex motions such as backflips, rolls, and locomotion, but there has been little work that aims to integrate the motions and methods.

Motion controller design for simulated characters and robots alike generally involves complex algorithms and fine tuning of many parameters before achieving a motion that is both physically realistic and stylistically appealing. A simple method of authoring controllers that is accessible to experienced and novice users alike has the potential to greatly further the number of motions a framework for physically simulated characters can produce.

1.1. Overview of Approach and Contributions

To help overcome the prevalent issues in physics based animation, we propose a simple controller authoring language, incorporating common features from research in the area. We argue that such a language should be comprehensive enough to allow for the development of complex motions, while also being approachable for novice users.

The primary contribution of this paper is our take on such a language. Our design is based around a finite state machine model, which allows users to design motions by breaking complex motions into smaller ones. The lowest level vocabulary of the language is derived from the commonly seen aspects of physics animation research, such as proportional derivative or virtual force controllers. These motion primitives can be abstracted in order to allow novice users to create controllers based on intuitive concepts such as joint angles or forces specified in Cartesian coordinates, while also being open to fine tuning by expert users.

In addition, we contribute a novel method for integrating the controllers developed using our language. This is done by learning new inter-controller transitions through optimization.

Our language has been used to design over 25 controllers for various motions ranging from simple static poses, to highly dynamic motions such as backflips and forward walkovers. We also demonstrate the connectivity of our controllers by random walkthroughs of our integrated set of controllers.

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 paper, and can be found in [GP12]

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 [RH91, LvdPF96, HW98, LKL10]. In SIMBICON [YLvdP07], Yin et al create a number of locomotion controllers by splitting more complex motions into a small number of phases, which when repeated cyclically, to 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 [HP97, HW98, YLvdP07]. Virtual force control through Jacobian transpose has been crucial in the development of controllers for both robotics [SADM94, PCT*01] and physics based animation [CBvdP10, HYL12]. 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 [MZS09, dLMH10, ABFHdL14]. 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.

Another approach involves the creation of physics based controllers from motion captured data. This has been explored in [SKL07, DSAP08, LKL10].

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 [SKL07]. In [LKL10], 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 [CKJ*11], Coros et al. transition between similar locomotion controllers for quadrupeds by linearly interpolating between the two. Recent works including [YLvdP07, MDLH10, YL10, CBvdP10, TLC*10] show how successful transitions between similar locomotion controllers can emerge with little extra work. In [WFH10], Wang et al. create recovery controllers for various locomotion controllers. While normally used for correcting deviations from the set of acceptable states for the controller, the authors also show how transitions can be created by "recovering" from the transitioning controller into the new one.

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 [HYL12]. The works of [FVdPT01] and [Woo98] 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 between the complex, highly dynamic tasks of running and obstacle clearance are explored in Liu et al's Terrain Runner [LYvdPG12]. In [HL14], 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 [CBvdP09], using locomotion controllers as input. Alternatively, da Silva et al achieve composition of controllers using an interpolation scheme, linear Bellman combination, in [DSDP09]

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 [Ups89] for rendering problems and FoldIt [CKT*10] for protein folding. The Robot Operating System [QCG*09] is a general set of software libraries and tools for helping to build robot applications.

Closer to this work, domain specific languages such as Improv [PG96] and the Smart Object format [KT99] allow a user to script simple interactions between characters and environmental objects, as well as define simple kinematic motions. In [BK13], Backman and Kallmann create a visual language for designing physics-based controllers using motion graphs.

3. Language Features

In our work, we aim to consolidate these 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 layout of our language, discuss the motion primitives we have chosen to include, and present various feedback rules that have been incorporated into the language.

In this section, we briefly describe the language and its features, and present a few example controllers and motions. A more in depth description of the language, as well as the code for all of the controllers designed in it, can be found in the appendices of [Fir14].

3.1. 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 or she 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.

3.2. Motion Primitives

Another 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.

1. **Proportional Derivative Controller**
   The proportional Derivative (PD) Controller is denoted by

   $$\tau_c = k_p(\theta_a - \theta_d) - k_d\dot{\theta}_a \quad (1)$$

   where τ_c is the calculated control torque for a joint, θ_a the joint's current angle, θ_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 system also allows the goal angle for the PD Controller to be linearly interpolated between the current angle and the specified angle over a given duration of time. This allows the user to specify a time for the desired angle to be realized, while creating a smooth, linear motion between the two angles.
2. **Global PD Controller**
   The global PD controller takes as input the desired angle of a given body part specified with respect to the world frame. An example usage 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. **Virtual Force**
   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:

$$\tau_c = J^T F \tag{2}$$

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.

4. **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.
   Our system uses cyclic coordinate descent, as outlined in [Wel93], in order to determine the necessary angles along a chain of joints between the part being moved and a specified base joint. The desired angles are then achieved using PD Controllers to produce necessary joint torques.
5. **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.

3.3. 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 [YLvdP07], while the second, based on virtual forces, provides feedback in the Cartesian frame.

The PD Controller modifier is given by

$$\theta_d = \theta_{d0} + c_d d + c_v v \tag{3}$$

where θ_d is the modified desired joint angle for the PD controller, θ_{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.

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

$$F = c_d d + c_v v \tag{4}$$

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.

3.4. 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 contacted 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. **No Contact** - Switch phases after contact between the ground and a given body part has been broken. For example, switching from a preparation phase to an aerial phase in a jump controller.
4. **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.
5. **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.
6. **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.

3.5. Case Study: Step-Up Motion

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 1.

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 as follows

```
1  SCRIPTS standVFFB.t;
2
3  BEGINSCRIPT step_up(x, y)
4  PHASE 0
5      > standVFFB.t(.1);
6      TRANSITION to(1).after(complete);
7  ENDPHASE
8  PHASE 1
9      ACTIONS
10         IK rFoot.targetlocal(0.1, -.6, 0).base(rHip);
11         VFFB uTorso(3000., 250.).by(lAnkle).over(cop);
12     ENDACTIONS
13     TRANSITION to(2).after(time .2);
14 ENDPHASE
15 PHASE 2
16     ACTIONS
17         IK rFoot.targetglobal(x,y,-10)
18             .base(rHip)
19             .tolerance(0.001);
20         VPD rFoot(0.0,-300,-30).joint(rAnkle);
21         VFFB uTorso(3000, 250).by(lAnkle).over(lFoot);
22     ENDACTIONS
23     TRANSITION to(3).after(time .3);
24 ENDPHASE
25 PHASE 3
26     ACTIONS
27         POSE rHip(0.0).time(5);
28         VPD rFoot(0.0,-1000,-100).joint(rAnkle);
29         VFFB uTorso(3000, 250.).by(lAnkle).over(lFoot);
30     ENDACTIONS
31     TRANSITION to(4).after(contact rFoot);
32 ENDPHASE
33 ...
```

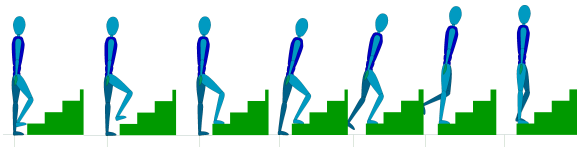

Figure 1: *step up script*

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 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 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 1 of the original step up controller.

In phase 2, 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 fasion.

Through this process we have designed a successful initial—albeit jerky—stepping up motion, shown in Figure 1. The motion can be smoothed or stylized by fine tuning gains and other parameters, either by hand or through an optimization technique.

When designing a controller, we generally start by outlining what each phase should do. Then, each phase is authored in succession, fine tuning parameters until we are happy with the phase. Finally, after all phases have been authored to create an initial approximation to the desired motion, we go back through the controller and further fine tune for realism,

style, or robustness from different starting states or character perturbations. While it is generally fairly straightforward to create an initial approximation for a motion, fine tuning it can be tricky and time-consuming.

3.6. Case Study: Somersault

Another motion created in our language is the somersault, as shown in Figure 2. To create this motion, we start by modifying the forward walkover script as both motions have similar beginnings. When running this initial script, we immediately see that the character needs to be more bent over to ready himself for a rolling motion. Therefore, we adjust the angles for the hips, shoulders, and ankles until we see the character place his hands closer to the feet.

Next, we notice that the character needs to adjust his back and neck in order to provide a better arch for achieving a roll. By adjusting these joint angles, we achieve a start to a rolling motion, but the character is still unable to complete the roll and end on his feet. To solve this, we adjust the hips and knees so that he forms a tighter ball and rolls onto his feet, while also bringing the arms around to push off the ground. Finally, we call the squat-with-virtual-force-balance-feedback script in order to achieve balance, and end in a state from which various other motions can be started.

From start to finish, the entire motion took less than a half hour to create, and the majority of this time was spent trying to perfect the final stage of the roll so that the character would end on his feet, rather than falling backward or forward. Most adjustments were achieved by directly manipulating the desired angles with PD Controllers, while virtual force feedback was included for balance at the end of the motion.

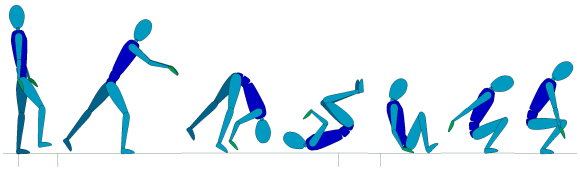

Figure 2: *Somersault script*

3.7. Language Re-use

We support re-use of our language in three primary ways. First, a controller written in our language can serve as a template for future controllers. If a user wants to design a motion similar to one already in the database, they can use the existing one as a base, modifying it appropriately. For example, if the user wanted to design a handstand motion, he could start from the forward walkover controller as the two will likely have similar structure.

Second, individual controllers are commonly re-used during the development of other motions. For instance, the step controller serves as a lead-in to both the handstand and forward walkover motions, and the squat controller is used to balance the character at both the beginning and end of the backflip controller.

Finally, a controller can be reused with a different set of parameters to create a new motion. This is shown in the SIMBICON-like locomotion scripts. The walk, fast walk, walk-in-place, and backward walk controllers all consist of a single call to the same four-phase SIMBICON script with a unique set of parameters in order to create a different motion. For instance, the walk script is given as below. For exact descriptions of the parameters, please refer to the original SIMBICON paper [YLvdP07].

```
1  SCRIPTS simbicon.t;
2
3  BEGINSCRIPT walk(void)
4  PHASE 0
5  #            (dt, cde, cdo,  cve, cvo, tor, swhe,
6  #            swho, swke, swko, stke, stko, ankle)
7  > simbicon.t(.3, 0.0, −2.2, −.2, 0.0, 0.0, −.4,
8                .7,  1.1,  .05,  0.05, 0.1,  −.2  );
9  TRANSITION to(0).after(complete);
10 ENDPHASE
11 ENDSCRIPT
```

By simply changing these parameters, different locomotion scripts can be authored.

3.8. 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 intermediate controllers.

While this process is often successful, enabling us to create long sequences of connected controllers such as that in Figure 5, 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 section.

4. Optimizing for New Transitions

We propose a method to automatically learn transition controllers that connect existing controllers, based on Covariance Matrix Adapatation [Han06], an evolutionary, derivative-free optimization technique. Our transition optimization approach takes as input two controllers: A and B. Then, by mutating specified phases within controller A, it produces a new controller that can successfully transition from A to B. This process is illustrated in Figure 3

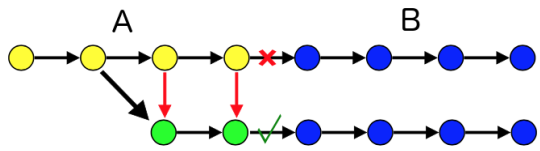

Figure 3: *The last two phases of Controller A (represented by the first four circles) are mutated to connect to Controller B (represented by the last four circles).*

4.1. 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 a simply list of all of these numerical values.

4.2. Objective Function

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

$$E = \sum_{i}^{n} w_i F_i(x) \tag{5}$$

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 trajectory that is known to be successful 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

$$F_1 = \sum_{t=0}^{n} (c_{a,t} - c_{d,t})^2 \tag{6}$$

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.

$$F_2 = \sum_{t=0}^{n} (\phi_{a,t} - \phi_{d,t})^2 \tag{7}$$

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

$$F_3 = \sum_{i=0}^{m} \sum_{t=0}^{n} (\theta_{i:a,t} - \theta_{i:d,t})^2 \tag{8}$$

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

Together, the objective functions given in equations 6,7, and 8 encode a target trajectory (5) that the optimized controller should follow. For most optimizations, weights of $w_1 = 100, w_2 = 1$, and $w_3 = 1$ were used. The first objective term F_1 was weighted significantly higher than the others, as it provided the main trajectory to follow, while the other two primarily helped to avoid local minima.

The goal trajectory can be created by simply recording the trajectory of a script that is known to work, given some initial state. For example, suppose the user wishes to connect a forward walkover script to a walking script. They can run the walking script from the initial standing state (which is known to be successful), and generate the trajectory from there. In addition, it is easy to create a transition from one script to a point partway through another (such as connecting the handstand to a point halfway through the forward walkover), by removing the unwanted sections of the trajectory.

In addition to the basic trajectory objective, 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

$$F_{\text{time}} = t_{\text{p}}$$

where t_{p} is the time spent in phase p. The objective function for minimizing energy cost is given as

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

where τ_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 are useful in improving existing transitions.

4.3. (1+1)-CMA-ES

As the objective function defined in section 4.2 has no well-defined gradient, we consider a derivative-free optimization technique, Covariance Matrix Adaptation (CMA) [Han06]. CMA is an evolutionary algorithm well suited to solving discontinuous, non-convex problems, which has shown to be successful for similar problems in previous works [CKJ*11, HL14, TGLT14, ASvdP13].

We opt to use (1+1)-CMA-ES [ISH06], an elitist variant to the original CMA. 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 proven to be successful in previous works [AvdP13] and has the benefit of being easy to implement, while still maintaining the performance of the original CMA.

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.

The (1+1)-CMA algorithm from [ISH06], as adapted for our simulation, is reprinted below.

Data: x_{parent}, $C = I$, $\bar{p}_{\text{succ}} = p_{\text{succ}}^{\text{target}}$, p_c, tol
Result: x_{parent}
while $f(x_{parent}) > tol$ **do**
  determine A such that $C = AA^{\top}$;
  $z \sim \mathcal{N}(0, I)$;
  $x_{\text{child}} \leftarrow x_{\text{parent}} + \sigma Az$;
  updateStepSize($\sigma, \lambda_{\text{succ}}, \bar{p}_{\text{succ}}$);
  if $f(x_{child}) \leq f(x_{parent})$ **then**
    $x_{\text{parent}} \leftarrow x_{\text{child}}$;
    updateCov($C, Az, \bar{p}_{\text{succ}}, p_c$);
  end
end

Algorithm 1: Transition Optimization Algorithm

The variables x_{parent} and x_{child} represent the optimization parameters of the parent and child generations, respectively. The remainder of the variables are the covariance matrix C, averaged success rate $\bar{p}$, target success rate $p_{\text{succ}}^{\text{target}}$, evolution path p_c, and step size σ. λ_{succ} is a binary variable with value 1 if $f(x_{\text{child}}) < f(x_{\text{parent}})$, and 0 otherwise. The function $f(x)$ represents the evaluated objective functions upon running the transition simulation with parameters x. Details of the functions updateStepSize to update σ and updateCov to update the covariance matrix C can be found in [ISH06].

5. Results and Discussion

5.1. Implementation

Our simulation is implemented using the Open Dynamics Engine [SO03] for a 17 link planar character. The controller authoring language is designed using Boost's xpressive grammar building library [Nie07]. The scripted simulations run in real time, while the transition optimization framework runs offline.

For most transitions, the (1+1)-CMA-ES optimization converges 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.

The number of phases necessary for convergence is highly

Sequence 1	Sequence 2
stand	stand
hands and knees	stand to hands and knees
crawl	hands and knees
crawl to squat	crawl
squat	crawl to squat
rise	squat
stand	backflip (failed landing)
step	crawl
forward walkover	crawl to squat
walk	squat,
stop	backflip
stand	squat
step	rise
handstand	stand
walk on hands	step
handstand	handstand
handstand to supine	walk on hands
supine	handstand
hands and knees	handstand to supine
crawl	supine
crawl to squat	kip (failure)
squat	
stand	
step	
handstand (failure)	

Figure 4: *Random Motion Sequences*

dependent on the length and number of parameters in each phase. Transitions from controllers with simple phases, such as the walk to stand transition, would require optimizing over more phases. All results were optimized using four to ten phases from the transitioning script, except for the walk-to-stand transition, which was optimized over 4 full walk cycles, or 16 phases.

After converging to a solution, the system automatically generates and writes out a new script with the optimized parameters. This script can then be run in real time in the same way as any standard script written in the language.

5.2. Controllers

Our language has been used to design over 25 separate controllers ranging from simple, static, balancing controllers such as the stand and squat controllers to more complex, dynamic controllers including the backflip and kip. Figure 5 shows a chart of our implemented controllers. Figure 6 shows many of the controllers we have authored sequenced together in one long motion, as well as the corresponding path through the controller graph. We encourage the reader to consult the accompanying video for examples of all controllers and transitions created through our framework.

Random walkthroughs of our integrated controllers graph produce the sequences of motions given in Figure 4.

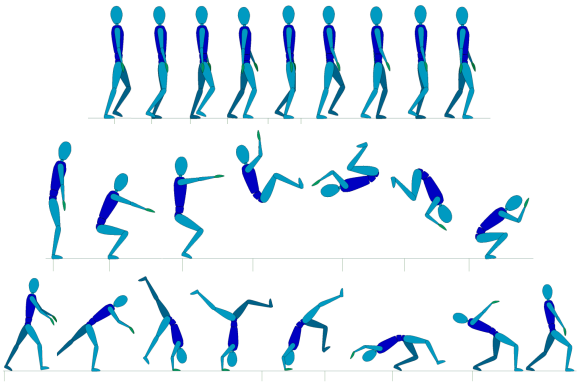

Figure 7: *Selection of Controllers. Top: Walk, Middle: Backflip, Bottom: Forward Walkover*

5.3. 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 tend to have larger basins of attraction, while highly dynamic controllers are much 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 8 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 9 shows a number of locomotion controllers over terrain of varying slope.

5.4. Transitions

We develop a number of new transitions to the integrated controller graph using our optimization framework, as well as improve upon existing transitions between controllers for style or smoothness.

Figures 10 and 11 illustrate newly optimized transitions. The naïve transition results in the character falling over in both cases. After optimization, the transitions are successful and smooth. In these, the character learns to adjust his walk to prepare for the upcoming motion, a forward walkover in the first example, and coming to a sudden stop in the second.

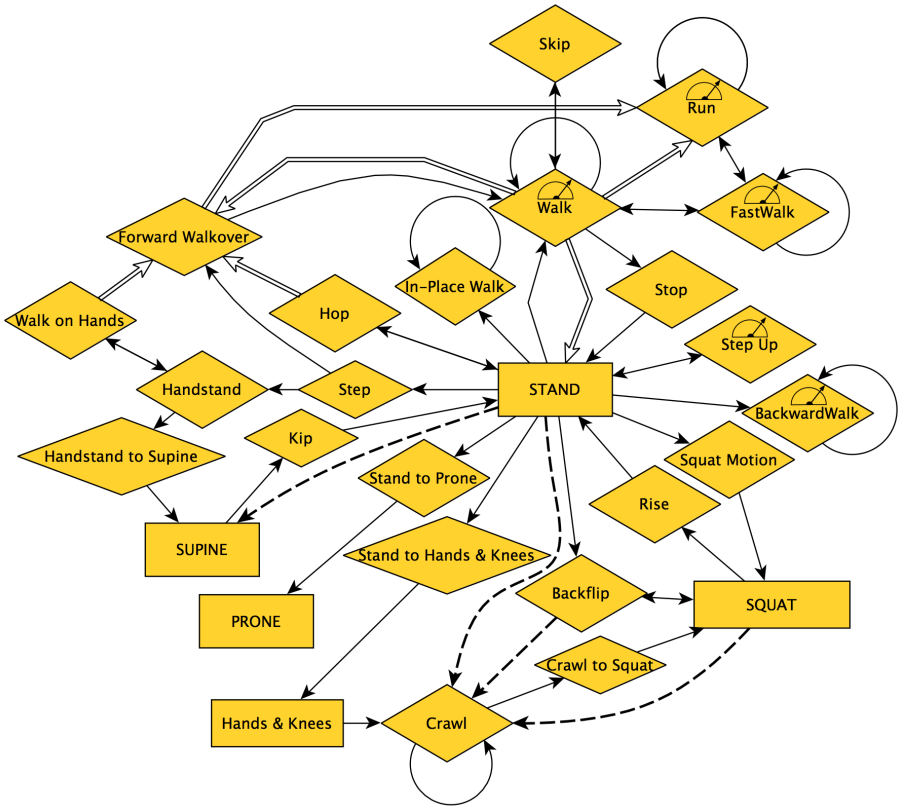

Figure 5: *Graph of currently implemented controllers. Diamond nodes represent dynamic actions while rectangular nodes are static poses. Solid arrows are naïve transitions while white arrows are the result of an optimized transition, and dashed arrows are corrective transitions that are utilized upon failure of a controller. The ⏲ symbol represents a controller that can be parameterized (based on velocity for SIMBICON-like controllers, and step height and width for the step up controller).*

Figure 12 shows how our framework can be used to improve a transition that is successful, but unrealistic. The naïve attempt to transition between the walk and run controllers results in an awkward jump during the transition where center of mass velocity is reduced from the walk velocity of 1.2m/s to around 0.8m/s. After optimization, the character learns to lean forward and take short rapid steps during the walk in order to smoothly transition into the run, increasing walk velocity to be similar to that of the run controller, from about 1.0m/s to 2.0m/s.

Figure 13 shows another example of a transition that is smoothed through our optimization framework. Here, the character transitions from a walking handstand directly into the end of the forward walkover motion. The naïve transition, while successful, is awkward as the character sways back and forth on his wrists before regaining forward momentum. After optimization, he learns to orient his legs and torso in the direction of the walkover, thereby producing a much smoother, cleaner transition. In this example, the additional objective function of minimizing energy was also used.

In Figure 14, the framework achieves a better transition between the forward walkover and run controllers. The initial attempt causes the character to jog backward for several steps in order to regain his balance. After optimization, the character smoothly transitions from the walkover into a run.

Finally, our transition optimization framework can also be used to determine how to adjust a controller when a previously successful transition has failed due to a change in initial character state. For example, a naïve transition between the skip and walk controllers is successful in most attempts, but under certain perturbations of the character state seen in random walkthroughs of the controller graph, it will fail. Figure 15 shows how our optimization framework has been used to correct this, by adjusting the skip controller so that the transition into the walk is successful.

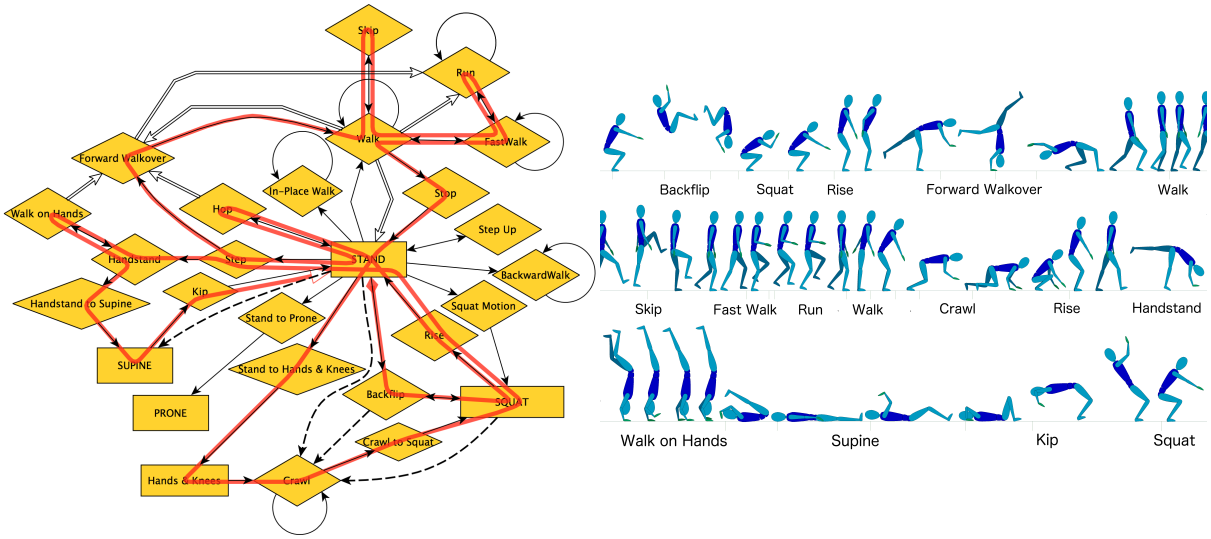

Figure 6: *Sequence of connected controllers. The path taken through our controller graph is shown on the left, and the corresponding visual simulation on the right.*

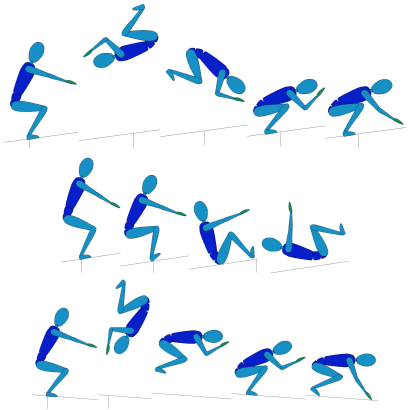

Figure 8: *Robustness of backflip controller. Top: 7.5 degree slope (successful). Middle: 8 degree slope (fails initially). Bottom: -4 degree slope (fails on landing)*

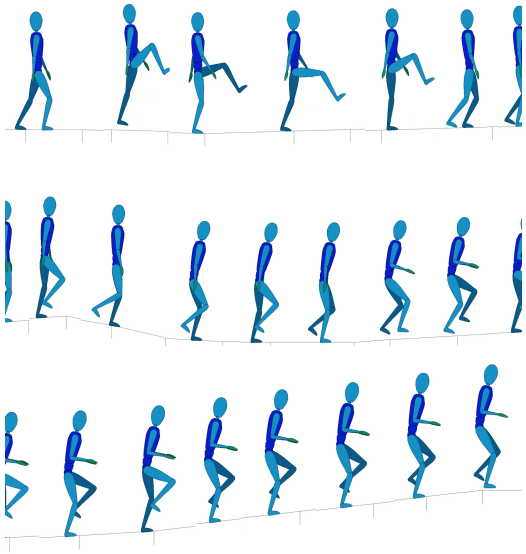

Figure 9: *SIMBICON style controllers (walk, skip, fastwalk, run) over varying terrain*

6. Conclusions and Future Work

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 much to be considered before a language such as the one we have presented could be adopted by the community at large. Perhaps one of the most significant

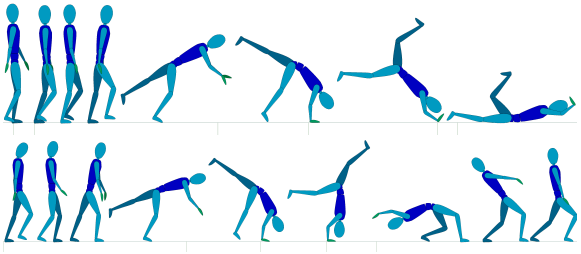

Figure 10: *Optimizing the transition between the walk and forward walkover controllers. The top sequence shows the initial attempt, and the lower sequence shows the successful transition after optimization*

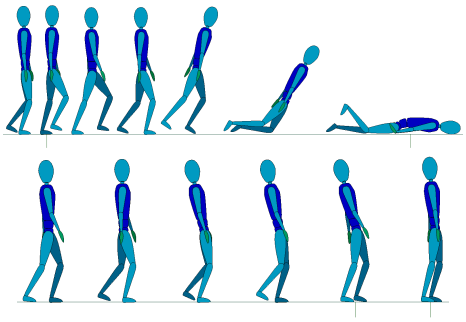

Figure 11: *Optimizing the transition between the walk and stand controllers. The top sequence shows the initial attempt, and the lower sequence shows the successful transition after optimization*

questions is whether it is necessary to introduce an entirely new domain specific language, or to build upon an existing scripting language such as Python, Javascript, or Lua. By building on an existing language, we could utilize existing features, such as debugging, error logging, and extensibility. However, it is hard to keep an existing language focused and stop divergence from the original goal. In creating a new language, we try to eliminate distractions and present the user with no more than they need: a set of control primitives that can be used to author a wide range of diverse motions. Whether its final form is an entirely new language, or as a library on top of an existing one, we believe there is significant value in a universal control authoring language that inspires reuse and robustness.

Another extension to this work could be to create a GUI interface such as that described in [BK13] to the language. While a universal domain specific language for motions control can help to bring controller design to a wider audience, users with little to no programming experience may have trouble picking it up. However, care must be taken to not over-simplify the language by doing so, thereby limiting the range of possiblities provided by it.

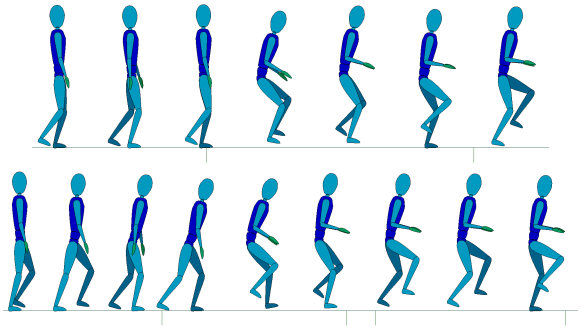

Figure 12: *Optimizing the transition between the walk and run controllers. controllers. The top sequence shows the initial attempt, which is successful but awkward, and the lower sequence shows the smoother transition after optimization*

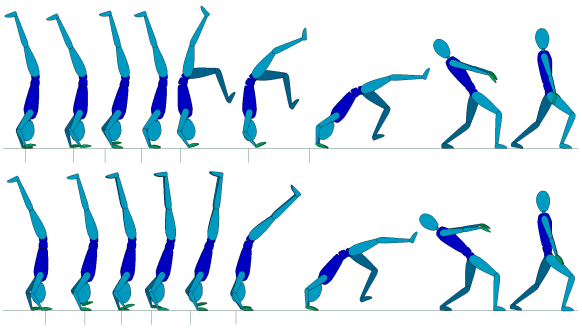

Figure 13: *Optimizing the transition between the walking handstand and forward walkover controllers. The top sequence shows the naïve transition, which is successful but awkward, and the lower sequence shows the smoother transition after optimization*

The use of hierarchical controllers, while greatly helping to promote re-use and templating in the language, can also be a hinderance. If the user wishes to change a parameter in a parent script, he or she must carefully consider all controllers that build off that script. One simple change can often cause the child controllers to fail where they succeeded before. Controller hierarchies should be used sparingly and only when there is a strong reason for them such as for a family of similar controllers, such as in the SIMBICON-based locomotion scripts.

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 transistions between controllers is a start to

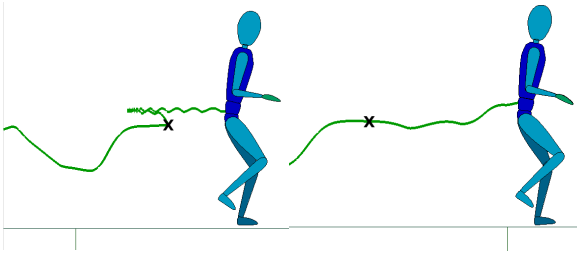

Figure 14: *Optimizing the transition between the forward walkover controllers and run controllers. The center of mass trajectory is shown, with **X** representing the transition. The naïve approach on the left shows the character running backward slightly to regain his balance. The optimized result is on the right.*

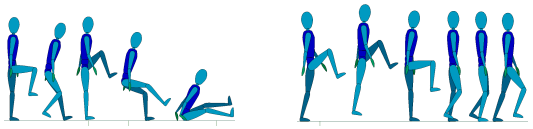

Figure 15: *Optimizing to correct the transition between the skip and walk controllers. The sequence on the left shows the failed attempt encountered during a random walkthrough of the controller graph. The right sequence shows the successful, corrected transition after optimization*

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 [FVdPT01]. Another possibility is to interpolate between existing transition controllers that have been shown to be successful for different character states.

At the time of writing, our language is limited to a planar character. However, extending the language to support three dimensional characters should be fairly straightforward. The most significant hinderance would be the number of additional parameters that would be introduced when a character has free range in a three dimensional space, which would likely increase the amount of time spent on fine tuning these parameters. The transition optimization framework could potentially benefit from being extended into 3D, as there would be more values contributing to the objective trajectory, helping to define it better, leading to a more constrained problem. While the language is currently limited to motions for a bipedal, humanoid character, we believe that many of the concepts presented in this work could be extended to other characters, such as quadrupeds.

The idea of creating controllers from motion captured data has been explored in [SKL07, DSAP08, LKL10]. 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 [HL14]. 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 designing controllers in our language, the majority of time and effort is spent on fine tuning parameters in order to create a robust and realistic motion. This is especially true for longer motions, where the entire motion would have to be replayed everytime a parameter is changed. A possible direction for future work builds on the live coding examples described in Bret Victor's talk Inventing on Principle [Vic12]. In these examples, Victor demonstrates a coding environment in which users can update a script and see in real-time how their changes will affect not only the present, but also the past and future states in the simulation. We believe that a similar idea applied to our work could greatly reduce the amount of time a user spends tuning parameters to achieve the perfect motion.

References

[ABFHdL14] AL BORNO M., FIUME E., HERTZMANN A., DE LASA M.: Feedback control for rotational movements in feature space. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 225–233. 2

[ASvdP13] AGRAWAL S., SHEN S., VAN DE PANNE M.: Diverse motion variations for physics-based character animation. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2013), ACM, pp. 37–44.

[AvdP13] AGRAWAL S., VAN DE PANNE M.: Pareto optimal control for natural and supernatural motions. In *Proceedings of the Motion on Games* (2013), ACM, pp. 7–16. 8

[BK13] BACKMAN R., KALLMANN M.: Designing controllers for physics-based characters with motion networks. *Computer Animation and Virtual Worlds 24*, 6 (2013), 553–563. 3, 12

[CBvdP09] COROS S., BEAUDOIN P., VAN DE PANNE M.: Robust task-based control policies for physics-based characters. In *ACM Transactions on Graphics (TOG)* (2009), vol. 28, ACM, p. 170. 2

[CBvdP10] COROS S., BEAUDOIN P., VAN DE PANNE M.: Generalized biped walking control. *ACM Transactions on Graphics (TOG) 29*, 4 (2010), 130. 2

[CKJ*11] COROS S., KARPATHY A., JONES B., REVERET L., VAN DE PANNE M.: Locomotion skills for simulated quadrupeds. *ACM Transactions on Graphics (TOG) 30*, 4 (2011), 59. 2, 8

[CKT*10] COOPER S., KHATIB F., TREUILLE A., BARBERO J., LEE J., BEENEN M., LEAVER-FAY A., BAKER D., POPOVIĆ Z., ET AL.: Predicting protein structures with a multiplayer online game. *Nature 466*, 7307 (2010), 756–760. 3

[dLMH10] DE LASA M., MORDATCH I., HERTZMANN A.: Feature-based locomotion controllers. *ACM Transactions on Graphics (TOG) 29*, 4 (2010), 131. 2

[DSAP08] DA SILVA M., ABE Y., POPOVIĆ J.: Simulation of human motion data using short-horizon model-predictive control. In *Computer Graphics Forum* (2008), vol. 27, Wiley Online Library, pp. 371–380. 2, 13

[DSDP09] DA SILVA M., DURAND F., POPOVIĆ J.: Linear bellman combination for control of character animation. In *Acm transactions on graphics (tog)* (2009), vol. 28, ACM, p. 82. 2

[Fir14] FIRMIN M.: *Design and integration of controllers for simulated characters*. Master's thesis, University of British Columbia, 2014. 3

[FVdPT01] FALOUTSOS P., VAN DE PANNE M., TERZOPOULOS D.: Composable controllers for physics-based character animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 251–260. 2, 13

[GP12] GEIJTENBEEK T., PRONOST N.: Interactive character animation using simulated physics: A state-of-the-art review. In *Computer Graphics Forum* (2012), vol. 31, Wiley Online Library, pp. 2492–2515. 2

[Han06] HANSEN N.: The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*. Springer, 2006, pp. 75–102. 7, 8

[HL14] HA S., LIU C. K.: Iterative training of dynamic skills inspired by human coaching techniques. *ACM TRANSACTIONS ON GRAPHICS* (2014). 2, 8, 13

[HP97] HODGINS J. K., POLLARD N. S.: Adapting simulated behaviors for new characters. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 153–162. 2

[HW98] HODGINS J. K., WOOTEN W. L.: *Animating human athletes*. Springer, 1998. 2

[HYL12] HA S., YE Y., LIU C. K.: Falling and landing motion control for character animation. *ACM Transactions on Graphics (TOG) 31*, 6 (2012), 155. 2

[ISH06] IGEL C., SUTTORP T., HANSEN N.: A computational efficient covariance matrix update and a (1+ 1)-cma for evolution strategies. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation* (2006), ACM, pp. 453–460. 8

[KT99] KALLMANN M., THALMANN D.: *Modeling objects for interaction tasks*. Springer, 1999. 3

[LKL10] LEE Y., KIM S., LEE J.: Data-driven biped control. *ACM Transactions on Graphics (TOG) 29*, 4 (2010), 129. 2, 13

[LvdPF96] LASZLO J., VAN DE PANNE M., FIUME E.: Limit cycle control and its application to the animation of balancing and walking. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM, pp. 155–162. 2

[LYvdPG12] LIU L., YIN K., VAN DE PANNE M., GUO B.: Terrain runner: control, parameterization, composition, and planning for highly dynamic motions. *ACM Trans. Graph. 31*, 6 (2012), 154. 2

[MDLH10] MORDATCH I., DE LASA M., HERTZMANN A.: Robust physics-based locomotion using low-dimensional planning. *ACM Transactions on Graphics (TOG) 29*, 4 (2010), 71. 2

[MZS09] MACCHIETTO A., ZORDAN V., SHELTON C. R.: Momentum control for balance. In *ACM Transactions on Graphics (TOG)* (2009), vol. 28, ACM, p. 80. 2

[Nie07] NIEBLER E.: Boost. xpressive, 2007. 8

[PCT*01] PRATT J., CHEW C.-M., TORRES A., DILWORTH P., PRATT G.: Virtual model control: An intuitive approach for bipedal locomotion. *The International Journal of Robotics Research 20*, 2 (2001), 129–143. 2

[PG96] PERLIN K., GOLDBERG A.: Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM, pp. 205–216. 3

[QCG*09] QUIGLEY M., CONLEY K., GERKEY B., FAUST J., FOOTE T., LEIBS J., WHEELER R., NG A. Y.: Ros: an open-source robot operating system. In *ICRA workshop on open source software* (2009), vol. 3. 3

[RH91] RAIBERT M. H., HODGINS J. K.: Animation of dynamic legged locomotion. In *ACM SIGGRAPH Computer Graphics* (1991), vol. 25, ACM, pp. 349–358. 2

[SADM94] SUNADA C., ARGAEZ D., DUBOWSKY S., MAVROIDIS C.: A coordinated jacobian transpose control for mobile multi-limbed robotic systems. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on* (1994), IEEE, pp. 1910–1915. 2

[SKL07] SOK K. W., KIM M., LEE J.: Simulating biped behaviors from human motion data. In *ACM Transactions on Graphics (TOG)* (2007), vol. 26, ACM, p. 107. 2, 13

[SO03] SMITH R., OTHERS: Open dynamics engine, 2003. 8

[TGLT14] TAN J., GU Y., LIU C. K., TURK G.: Learning bicycle stunts. *ACM TRANSACTIONS ON GRAPHICS 33*, 4 (2014). 8

[TLC*10] TSAI Y.-Y., LIN W.-C., CHENG K. B., LEE J., LEE T.-Y.: Real-time physics-based 3d biped character animation using an inverted pendulum model. *Visualization and Computer Graphics, IEEE Transactions on 16*, 2 (2010), 325–337. 2

[Ups89] UPSTILL S.: *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., 1989. 3

[Vic12] VICTOR B.: Inventing on principle, 2012. 13

[Wel93] WELMAN C.: *Inverse kinematics and geometric constraints for articulated figure manipulation*. PhD thesis, Simon Fraser University, 1993. 4

[WFH10] WANG J. M., FLEET D. J., HERTZMANN A.: Optimizing walking controllers for uncertain inputs and environments. In *ACM Transactions on Graphics (TOG)* (2010), vol. 29, ACM, p. 73. 2

[Woo98] WOOTEN W. L.: Simulation of leaping, tumbling, landing, and balancing humans. 2

[YL10] YE Y., LIU C. K.: Optimal feedback control for character animation using an abstract model. *ACM Transactions on Graphics (TOG) 29*, 4 (2010), 74. 2

[YLvdP07] YIN K., LOKEN K., VAN DE PANNE M.: Simbicon: Simple biped locomotion control. In *ACM Transactions on Graphics (TOG)* (2007), vol. 26, ACM, p. 105. 2, 4, 6