AN ABSTRACT OF THE THESIS OF

Robert W. Rose for the degree of Master of Science in

Electrical and Computer Engineering presented on September XX, 2006.

Title: Evolving Character Controllers for Collision Preparation

Abstract approved: _____

Ronald A. Metoyer

Full dynamic control of physically simulated characters is the holy grail of video games and other interactive applications. Recent advances in the field have presented controllers that can handle the balance and fall response after a collision. This thesis addresses the development of a controller that handles the response *before* a collision—reflexive posturing that protects the character from impact. To solve this problem we present a technique that uses genetic algorithms to optimize a pose controller based on analysis from a pain measurement system. Applying our technique we develop collision preparatory controllers that defend a virtual character in several different styles from a variety of threats.

Evolving Character Controllers for Collision Preparation

by

Robert W. Rose

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September XX, 2006
Commencement September 2006

Master of Science thesis of Robert W. Rose presented on September XX, 2006.

APPROVED:

Major Professor, representing Electrical and Computer Engineering

---

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Robert W. Rose, Author

Master of Science thesis of Robert W. Rose presented on September XX, 2006.

APPROVED:

Major Professor, representing Electrical and Computer Engineering

---

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Robert W. Rose, Author

ACKNOWLEDGEMENTS

I would like to thank...

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF APPENDIX FIGURES

DEDICATION

Insert dedication here...

Chapter 1 – Introduction

Realism in interactive applications is largely dependent upon how effectively the movements of human characters are conveyed. Since we are all inherently familiar with the properties of human motion, irregularities in human movement in an interactive application are easy to identify. Generating realistic human motion is a difficult problem however. Motion capture is a relatively simple way to record and "play-back" human movements, but the system fails when the character must interact with its environment or perform a task unanticipated by the motion capture data. Physically simulated characters allow a character to interact realistically with its environment, but controlling physically simulated characters is difficult. The problem of generating human motion for collisions with an environment is particularly challenging, such as the problem of a character getting hit in the head with a fast-moving object. Motion capture is not a suitable technique due to the variety of interactions in this class that are possible, the ethical ramifications of potentially injuring a motion capture actor, and the ability of motion capture to record an "authentic" motion given an actor's anticipation—knowing a priori that an object will be thrown at their head. Physical simulation is an attractive means of solving this problem because it would allow the character to respond in a dynamic and realistic manner without the need of motion capture (and potentially hurting a motion capture actor), but what of the preparatory pose the character

should make before they are hit?

In this thesis we present a technique for developing a controller for a physically simulated character that prepares itself for collision with a fast-moving object—an element that is often missing from video games and interactive applications. We have developed a system for measuring pain perceived by a virtual character and use this as the basis for a genetic algorithm optimization of a pose controller. Applying our technique we develop collision preparatory controllers that defend a virtual character in several different styles from a variety of threats.

1.1 Problem Statement

The goal of this work is to develop a collision preparatory controller that can protect a virtual character from a variety of impacts with fast-moving objects. To develop this controller we focus our approach on using genetic algorithms to optimize a physically simulated pose controller. Automated optimization requires a fitness function to evaluate a controllers effectiveness; for this we explore the development of a *pain measurement* system that judges controllers based on the amount of pain the character perceives after impact. We also explore redundant protection mechanisms in order to achieve more natural responses.

The main technical challenges of this work are:

- *Evaluating a controller's ability to protect a virtual character.* When a controller places the character in a protective pose and is then hit by an object, we need an automated method to judge how *well* the controller defended the

character.

- *Developing controllers that protect the character in different ways.* People protect themselves from collisions in many different ways. For our work we wish to develop controllers that protect the character using different objectives for defense, such as blocking using the hands or far away from the face.

1.2 Techniques

Developing a character controller requires first a simulation environment for the character to reside in. We use a physical simulation package to aid us in building a robust character model that interacts with its environment in a physically realistic manner. On top of this system we have designed a higher-level joint control mechanism utilizing PD-servos that moves the character in a way that emulates the movement of humans.

Evolution of the character controller is accomplished using genetic algorithms. Expressing a solution as a set of desired joint angles, genetic algorithms optimize these joint angles according to a fitness function we have developed. The fitness function measures the ability of the control solution to protect the character by observing the amount of *pain* perceived by the character upon impact with the object.

Measuring the pain perceived by a virtual character is a technique we have developed and present in this work. Our technique is rooted in observations of human

dodging and protective poses when under the threat of a projectile object. Humans protect certain parts of the body more than others, and also produce generalized, reflexive motions when threatened—our technique attempts to encapsulate these attributes.

1.3 Contributions

The contributions of this work are summarized as follows:

- *Ball and socket joint control system.* We have developed a robust ball and socket joint control system suitable for controlling a physically simulated character model. We use this system for controlling our character's head, torso and shoulders.

- *Collision preparatory pose evaluation.* Our work utilizes a pain measurement system we have developed that measures how effectively a pose prepares a virtual character for an impending collision with an object. The system evaluates poses on the basis of how much *pain* the character perceives upon impact.

- *Testing environment for collision preparation.* We have developed a robust system for repeated testing and simulation of a physically simulated character undergoing evolution in search of a collision preparatory pose. The system performs repeated testing on controllers under various modes of *joint failure* according to a *threat grid* (described in Chapter 4).

1.4 Thesis Overview

Following a discussion of background and related work in Chapter 2, we introduce our human model and control system in Chapter 3. In Chapter 4 we present our system for evolving character controllers for collision preparation, including how we test, evaluate, and generate controllers. Chapter 5 assesses the performance of our system, and we conclude in Chapter 6 with a discussion of our findings and opportunities for future work.

Chapter 2 – Background

2.1 Physically Simulated Humans

Physical simulation became a popular area of interest to computer graphics researchers in the mid 1980's. At the time, the state of the art method for animating computer graphics was by *keyframing*, a technique where scenes are generated by interpolating between the "key frames" as set by an animator[1]. As computer animation scenes became more complex however, the job of the animator became more difficult. Scenes where objects are intended to collide and interact with each other in a physically realistic manner were cumbersome or impossible to keyframe. It was recognized at the time that if the computer could *physically simulate* these aspects of a scene then this would relieve the animator of an unwieldily task and yield more realistic looking animation.

Although physical simulation systems had been in use for research in mechanics, the goals of a mechanics researcher and a computer animation researcher were quite different. In computer animation, performance was valued over simulation accuracy—so long as the results looked realistic. Early physical simulation systems for computer animation focused on optimizing impact dynamics and collision detection in ways that were not necessarily accurate, but looked convincing [10].

[1]The term "keyframing" is borrowed from traditional hand animation, where "keyframe artists" draw the main poses of a character, and "tweeners" draw the in-between frames [28].

(Today the goals are very much the same: *performance* and *believability* is more important than correctness).

As physical simulation systems progressed the desire to animate more complex scenes increased. If one could animate realistically a few falling boxes, why not simulate an entire human? Several researchers developed models for a set of objects connected by joints—known as *articulated characters*—that attempt to model the human body (and other creatures) inside a physical simulation [2, 29]. These types of human models have found a home in video games as "ragdoll" characters—uncontrolled models that simulate a lifeless character. While these models look realistic at a glance, their inability to direct their own behavior fails to remain life-like under careful observation. Dynamic control of physically simulated characters that looks natural is the holy grail of video games and other interactive applications.

2.2 Controlling Physically Simulated Humans

Physically simulating a human body presents a control challenge. In keyframing, a human character is animated by specifying key joint angles over time, but in a purely physical simulation context the character is animated by specifying torques (at the joints) over time. A system must be devised to provide these torques for the character so as to make it perform the desired action.

Borrowing from control work done in the field of robotics [21], early control systems focused on developing algorithms that generate torques that, when applied

to the character model, resulted in human-like motion. Bruderlin et al., Raibert et al. and Laszlo et al. developed systems for controlling walking gaits in physically simulated humans [4, 22, 14]. Takashima presented a technique for controlling a gymnast on a high bar [27]. Later, Hogins et al. developed a complete system for animating human athletics tasks including running, cycling, and launching from a vaulting horse [11].

A disadvantage of these control systems were that the parameters used to influence their behavior were developed in an ad hoc manner. Once the function used to perform the control was determined, it was up to the researcher to come up with values for the function that worked optimally with the given character model. If it were possible to automatically generate controllers to perform given tasks then this would free the researcher from an incredibly time consuming task—and possibly result in more robust controllers.

In this work, we focus on the problem of automatically generating a controller to perform a single task. Automatically generating systems that *combine* controllers has also been researched: De Garis presented a system that uses Neural Networks to switch between controllers, the weights of which are learned using Genetic Algorithms [5]. More recently, Faloutsos et al. created a framework that uses Support Vector Machines to learn how to switch between controllers with amazing results [6]. Both approaches are appealing because they develop the high-order control mechanism automatically, rather than being hand-tuned by the designer. We focus our efforts on similar learning-based approaches, but only for a single controller.

2.3 Genetic Optimization of Human Character Controllers

Learning a character controller is appealing because it relieves the burden of developing the character controller from the researcher. In this section we look at approaches for learning a character controller using genetic algorithms and genetic programming.

2.3.1 Genetic Algorithms

Genetic algorithms are an optimization technique inspired by evolution [7]. Genetic algorithms operate on *chromosomes*, string encodings of solutions to the problem being optimized. Pools of chromosomes are tested against a *fitness function* and then each is assigned a *fitness* value. Once each has a fitness value the the chromosomes are "mated" together and then "mutated" to generate a new pool of chromosomes. This process repeats until a satisfactory minima has been reached.

Genetic algorithms have been used to develop controllers for physically simulated controllers by pairing them with neural networks [20]. In this approach, the chromosome is used to store the weights of the neural network. The neural network is then ran and the outputs are applied as torques to the simulated character. During the simulation the character's ability to perform the given task is rated using a fitness function, and the resulting fitness is used during the selection process to create the next generation. Optimizing the weights of a neural network using genetic algorithms was applied successfully by de Garis to optimize control of a 2D walking stick figure [5]. Smith applied the technique to a fully articulated

3D humanoid model and developed controllers that could walk and balance [26].

Genetic algorithms have also been applied in other ways to develop control systems for physically simulated characters. Sims used genetic algorithms to optimize not only the control system of a virtual creature, but also creature morphology [25]. Roberts et al. and Wyeth et al. have used genetic algorithms not to develop a complete controller, but rather to optimize existing ones [23, 31]. In our work we use genetic algorithms to optimize desired joint angles, not necessarily a complete control system.

2.3.2 Genetic Programming

A related technique for developing character controlers is genetic programming, which deals with the optimization of a computer program [13]. Programs are generated at random, tested individually, and then combined and mutated to create a new generation of programs. The process is similar to genetic algorithms, except the basis for optimization is the program itself rather than parameters to one. An advantage of genetic programming over genetic algorithms is that the search space is not necessarily fixed in size—a program may grow to any length necessary, but a chromosome remains at fixed length.

Gritz et al. used genetic programming to evolve Lisp programs that control 3D articulated characters [8, 9]. Using their approach they were successful developing a hopping/walking controller for a 4 segment, 3 degree of freedom character. They were also able to develop a pointing and gesturing controller for a 28 degree of

freedom humanoid character. Wolff has developed a virtual register machine language to perform genetic programming with, and has used it to develop walking and balance controllers for a 14 degree of freedom humanoid character [30].

2.4 Collision Response

A challenge with controlling a virtual character in a physical simulation is that contact with its environment can lead to unexpected or undesirable outcomes. For instance, when a character is struck by an object and is knocked off balance, this may place the controller into a state that it was not intended to handle. Managing these unanticipated collisions so that the controller can transition back to a desirable state in a realistic manner enhances the utility of physically simulated control.

Zordan et al. developed a dynamics control system that interacts with its environment by tracking motion capture data [32]. Using motion capture examples, the control mechanism applies torques to the character's joints so that it follows the motion capture. This allows a physically simulated character to perform complex tasks without the need of training or a complex controller.

In Zordan's work, reactions to contact with the environment (such as being hit by an object), were handled by relaxing the parameters that controlled how closely the character tracked the motion capture data. After a certain time interval had passed, the parameters were tightened back to their original values so that character's movement matched more closely the motion capture example. With

the addition of a balance controller, the character was able to be struck by an object, absorb the impact of the object briefly while remaining balanced and then resume normal operation. This system worked as long as the force of impact was not so great as to exceed the capabilities of the balance controller.

For situations where an excessive force would knock a character off balance, Mandel developed biomechanically inspired fall controllers that ease the character into transitional rest states [16]. When the character is struck by a force, the controller takes protective measures to brace the character for impact with the ground. After the character is resting on the ground the system places the character back under kinematic motion capture control.

A shortcoming of Mandel's approach was its inability to transition the character into a motion capture example that gracefully handled falling. For example, the character would merely brace itself for hitting the ground rather than trying to roll out of it. Zordan et al. later revisited this problem area and introduced a technique that blended the physical simulation control of the character falling into a nearby motion capture example of the character absorbing a fall [33]. This system worked by physically simulating the character falling after impact, but before the character physical simulation is allowed to run its course the character is transitioned to the nearest motion capture example. This allows the character to handle any number of collisions in a stylistic manner controlled by an animator rather than by a controller developed by a researcher.

Natural Motion has developed a commercial product Endorphin that unifies dynamic response controllers under a single framework [19]. Their system combines

custom reaction controllers with a multiple-pass simulation technique to generate responses to dynamic interactions. While they are aiming for real-time response controllers in the near future, their current approach is purely offline and targets motion capture data generation.

None of these systems attempt to take into account what happens to the character moments before the collision; they all take over the instant *after* the character has been impacted. We as humans do not always behave in this way however—we usually duck or attempt to shield ourselves. In this work we explore a technique to develop poses that protect the virtual character in a realistic manner.

Chapter 3 – Dynamics Model

3.1 Human Model

Our character models a human from the waist-up. It is made up of 9 segments connected by 8 joints with 18 degrees of freedom (DOF) total between the joints. Each body segment is simulated as a box with uniform density. The same density is used for each body segment. A diagram of our physically simulated human model is given in Figure 3.1.

Our testing environment builds the character model at runtime as specified by a character description file. The character description file format was developed for this work to aid in quickly prototyping character models and character model attributes. The final character description file that was used for this work and a specification of the file format is provided in Appendix A.

The character model is physically simulated using the NovodeX Physics SDK [1], a commercially available physical simulation package for the Microsoft Windows platform. We use NovodeX to simulate the dynamics of our character model and how it (physically) interacts with objects placed in our virtual world. NovodeX ensures that collisions behave appropriately and that the integrity of our joints are maintained by making sure two bodies connected at a joint do not drift outside the plausible bounds of the joint.

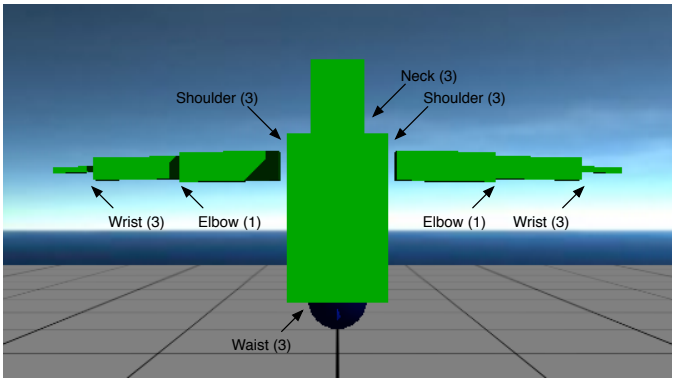

Figure 3.1: Physically simulated character model

Although NovodeX provides an API for controlling one and two DOF joints, it does not provide a mechanism to control a three DOF ball and socket joint. For this work we have developed our own ball and socket joint controller, which we describe in the following section.

3.2 Ball and Socket Joint Control

At the neck, hips and shoulders our human character model's body segments are connected by a joint that allows rotation about three axis with no translational freedom. This type of joint, known as a *ball and socket joint*, is diagrammed in Figure 3.2. Although these joints in real humans are not as simple as ball and socket joints[1], for the purposes of this work (and most computer animation tasks) simulating them as ideal ball and sockets suffices; we are more concerned with

[1]For example, the human shoulder is made up of four joints connecting the sternum, clavicle, scapula humerus and rib cage, each with three rotational DOF and three slight translation DOF [cite kinesiology book]. Accurately simulating a single human shoulder would require a 24 DOF joint model with an enormous number of constraints!

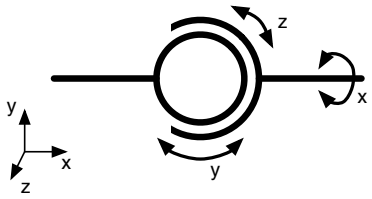

Figure 3.2: Ball and Socket Joint

visual plausibility and computational efficiency than accuracy.

3.2.1 One DOF PD-Servo

To simulate muscles moving the joints of our virtual character we use a *proportional-derivative servo* (PD-servo). The PD-servo is a popular mechanism used to control physically simulated humanoid characters in computer animation [11, 14, 32, 16, 33]. A PD-servo for a one DOF joint takes the joint's angle (θ) and angular velocity ($\dot{\theta}$) and applies a torque (τ) to it to move the joint to a desired angle (θ_{des}) according to the following equation:

$$\tau = k(\theta_{des} - \theta) + k_d(\dot{\theta_{des}} - \dot{\theta}) \tag{3.1}$$

where k and k_d are tunable stiffness and damping gains (respectively). Typically we desire no angular velocity at the end of the joint movement process ($\dot{\theta_{des}} = 0$), so we rewrite the PD-servo equation as:

$$\tau = k(\theta_{des} - \theta) - k_d(\dot{\theta}) \tag{3.2}$$

While the PD-servo is a poor approximation of the human skeletomuscular system, its computational simplicity makes it attractive as a means to control a virtual human.

The gains used for a PD-servo determine how the joint will behave and how realistic the motion generated will look. The stiffness gain will determine the power of the controller while the damping gain will adjust the controller's smoothness. Selecting gains that work well together and that are appropriate for the bodies connected to the joint can be a tricky endeavor. Over-powered, under-damped gains will create controllers that oscillate; under-powered, over-damped gains will create controllers that slowly (or never) reach their desired joint angle. Selecting gains for this work was done ad hoc, but others have reported success using heuristic methods for generating gains [32].

3.2.2 Three DOF PD-Servo

To control a ball and socket joint we require a PD-servo that can operate on a three DOF joint. We wish to take the desired joint orientation (a combination of the desired angles in each DOF) and use this to compute a three dimensional torque to apply to the joint that will move the joint to its desired orientation. If we rewrite the PD-servo equation as the difference between the desired and current joint orientation ($\Delta\vec{\Theta}$) and the difference between the desired angular velocity and

current angular velocity ($\Delta\dot{\vec{\Theta}}$):

$$\vec{\tau} = k(\Delta\vec{\Theta}) + k_d(\Delta\dot{\vec{\Theta}}) \tag{3.3}$$

then we can extend the PD-servo's behavior to support control in any number of rotational DOF, not just one. As before, we typically desire no angular velocity at the end of the joint movement process, so the difference between the desired and current angular velocity of the joint will be the inverse of the current angular velocity yielding:

$$\vec{\tau} = k(\Delta\vec{\Theta}) - k_d(\dot{\vec{\Theta}}) \tag{3.4}$$

The vector $\Delta\vec{\Theta}$ is a three dimensional vector encoding the difference between the desired orientation of the joint and the current orientation of the joint as Euler angles. Deriving this value by taking the difference between the desired orientation and the current orientation angles as Euler angles directly would be inappropriate because vector subtraction of Euler angles would not always yield the shortest path between the two orientations (due to redundancies in Euler angle representations). Instead, $\Delta\vec{\Theta}$ can be found by taking the difference of the two orientations encoded as quaternions and then converting the result to Euler angles.

Take the desired orientation of the joint as a quaternion[2] q_d and the current orientation of the joint as a quaternion q_c. The inverse of q_d is written as q'_d and the inverse of q_c as q'_c. The orientation difference between these two quaternions

[2]An excellent introduction to quaternions and their application in computer graphics can be found in [24].

is:

$$q = q_c q'_d q'_c \tag{3.5}$$

This operation will yield the shortest orientation between the two quaternions when the rotations they represent are in the same hemisphere. If the two quaternions are in opposite hemispheres (when their dot product is negative), we must invert q_c. If the orientations are in opposite hemispheres and we fail to invert q_c then the orientation difference will find the "long route" which results in an unstable controller.

To convert the resulting quaternion to a vector representing its orientation as Euler angles we find the "axis-angle" representation of the quaternion and then multiply the axis by the angle. Representing a quaternion as $(\vec{v}, w)$, the Euler angle representation of the quaternion is:

$$\vec{r} = 2 \cdot cos^{-1}(w) \cdot \frac{\vec{v}}{|\vec{v}|} \tag{3.6}$$

This operation is also referred to as finding the "moment" of a quaternion. The moment is used as a unit torque to rotate an object into the orientation expressed by the quaternion.

Finding the current angular velocity of the joint as Euler angles is achieved through a similar process. If we take the difference of the current joint orientation as a quaternion and the orientation of the joint at a previous time interval as a quaternion we will have a quaternion that represents the orientation change of the joint over the last time interval. We take this value, convert it to Euler angles,

and then divide by the amount of time transpired over the time interval to find the joint's angular velocity as Euler angles.

3.2.3 Controller Stability

In theory, the controller system described above should make a perfect joint controller. In practice however, it is far from perfect. Inaccuracies and minute precision loss in the physical simulation cause instabilities that must be addressed.

A primary source of controller instability is due to precision loss in very low-torque situations. As the joint zeros in on its desired orientation, the controller has to apply less torque to move the joint. Unfortunately, due to precision loss, a torque that may ideally be the "final" torque to move the joint into position may actually move the joint slightly beyond its target orientation. At this point the controller reverses direction and applies an opposite torque to attempt to move the joint to the target orientation. Again, precision loss may cause the joint to slide slightly past its target. This situation causes low magnitude, high frequency oscillations in the joint controller.

Low magnitude, high frequency oscillations are dealt with by smoothing the value given to the PD-servo for the joint's angular velocity. This is accomplished by keeping a running average of the angular velocity over a small time window rather than using an instantaneous snapshot of the angular velocity between the current time step and the previous. Keeping a running average of the angular velocity over just the last two frames significantly reduces oscillations, at the cost

of slightly stalling the joint's initial movement. Care must be taken when selecting the size of the window however; too large a window can cause the controller to be under-damped and introduce low frequency oscillations.

Another source of instability stems from the physics of rotating a body connected to a joint about its twist axis, which, depending upon the inertial properties of the body, may require substantially less torque than rotating the body about one of its other axis. Recall that our PD-servo equation we use to control a 3 DOF ball and socket joint uses the same k and k_d parameters regardless of the axis we are desiring to rotate the joint about. For certain joints, this may cause rotation about the twist axis to be significantly more powerful than rotations about other axis.

To address the power imbalance brought on by using a single set of k and k_d parameters for each joint a torque damping step was introduced. After we have derived the torque (τ) from the PD-servo equation we reduce the torque in the direction of the twist axis ($\vec{t}$). This is accomplished by the following:

$$d = d_r \cdot |\vec{\tau} \cdot \vec{t}| \tag{3.7}$$

$$\tau_x = \tau_x \cdot (1 - d \cdot t_x) \tag{3.8}$$

$$\tau_y = \tau_y \cdot (1 - d \cdot t_y) \tag{3.9}$$

$$\tau_z = \tau_z \cdot (1 - d \cdot t_z) \tag{3.10}$$

where d_r is a fraction between 0 and 1 specifying how much we should diminish control about the joint's twist axis. Like other tunable parameters we use, care

must be taken when selecting d_r as too high a value can cause the control system to be under-damped and introduce low frequency oscillations. For most joints we typically a d_r of 0.9.

A "catch all" method that can be employed to improve controller stability is to simply disable the controller once the desired orientation has been reached. When the magnitude of the difference between the desired and current joint orientations has gone below a threshold and the joint's angular velocity has also gone below a threshold the PD-servo control system can be disabled and the joint can be locked in place (by either the physical simulation software or otherwise). A problem with this approach is that it may not look that visually pleasing for joints that have other joints connected to them—the solution is to make the threshold for when the PS-servo is shut off very small. This approach also requires another system outside the PD-servo controller that decides when to turn the controller back on (i.e., when a force is acted upon the body, etc).

3.2.4 Joint Orientations

Up until now, we have only talked about the orientation of a *joint*. But what does that mean exactly? The orientation of a joint is the difference between the orientation of the bodies connected by the joint and their initial orientation. This concept is best illustrated by example.

Take the bodies connected by a ball and socket joint in Figure 3.3. Say that when these bodies were both created they were oriented down the x axis (z facing

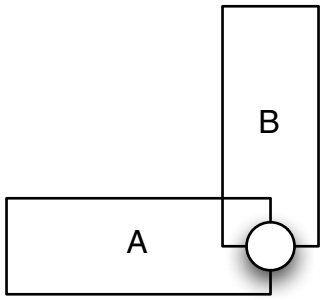

Figure 3.3: Example: Joint initial orientation

outward), so body A has an initial orientation of $(0, 0, 0)$ and B of $(0, 0, \frac{\pi}{2})$. If we call these orientations of the bodies the initial orientation of the joint then the joint's orientation in this configuration would be $(0, 0, 0)$.

To find the orientation of the joint in other positions we will need to store a rotation transform for each body that will put it into the rotational space of its initial pose with the joint. For A there's nothing to do, its initial orientation in "joint space" *is* its initial orientation. For B, the rotational transform will be the inverse of its initial orientation with the joint: $(0, 0, -\frac{\pi}{2})$. We store these inverse transforms as quaternions q_{Ainv} and q_{Binv}.

Now the objective is to determine the orientation of the joint when its in the configuration shown in Figure 3.4. Both bodies A and B are now in their initial orientations of $(0, 0, 0)$. To find the orientation of the joint we take the difference of the orientations of the bodies relative to their initial orientations with the joint:

$$q_A = q_{Ainv} q_{Acurrent} q'_{Ainv} \tag{3.11}$$

$$q_B = q_{Binv} q_{Bcurrent} q'_{Binv} \tag{3.12}$$

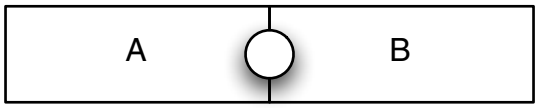

Figure 3.4: Example: Joint second orientation

$$q_{joint} = q_A q_B q'_A \tag{3.13}$$

where $q_{Acurrent}$ and $q_{Bcurrent}$ are the current orientations of the bodies A and B respectively. Since B's current orientation is $(0, 0, 0)$ and its inverse transform q_{Binv} is $(0, 0, -\frac{\pi}{2})$ (as Euler angles, see above) this puts the orientation of the joint (q_{joint}) at $(0, 0, -\frac{\pi}{2})$.

As stated earlier, we wish to control our ball and socket joints by specifying desired joint angles for each DOF (Section 3.2.2). Let's say the joint from the previous example is a three DOF ball and socket joint; the first DOF being along the x axis, the second along the y axis, and the third along the z axis. To convert desired angles in each DOF to a desired joint orientation we express each desired angle as a quaternion and multiply them together:

$$q = q_z(q_y q_x q'_y)q'_z \tag{3.14}$$

The order in which the desired angles for each DOF are multiplied together shown here is completely arbitrary, the operation can be done in any order, as long is it remains consistent for each joint[3]. Also note that we do not have to specify the DOF axis orthogonally, and that we can have any number of rotational DOF.

[3]For human readability, we prefer to use the conventional x, y, z order

3.2.5 Joint Limits

If we wish to impose joint limits in a particular DOF we may do so when we encode the quaternion representing the rotation along that DOF axis. If the angle given for the desired rotation exceeds the specified bounds, it can be truncated to fit within them:

$$\theta = min(\theta_{min}, max(\theta_{max}, \theta)) \tag{3.15}$$

where θ is the angle of rotation about the DOF axis. We call this setting a "soft limit" on the joint's rotational freedom.

While soft limits will prevent a *desired* joint angle from exceeding the thresholds, it still may not prevent the PD-servo from applying a torque to the joint that would place it outside the limits. (For example, a k k_d pair may generate a subtle oscillation around the desired orientation, exceeding a soft limit before it stabilizes). To solve this we impose "hard limits" on the PD-servo. When the joint orientation exceeds the boundaries set by the soft limits, we increase the k and k_d constants used for the joint so that it strengthens the joint around the boundaries. This gives the joint a bit less "play" around the boundaries and a more life-like appearance. This concept was inspired by human joints, where one could imagine cartilage in a joint as setting a soft limit, and bone sockets as setting a hard limit.

3.2.6 Joint Torque

The function of a joint in a physical simulation is to enforce a set of constraints. The joint ensures that the bodies connected to it will not drift outside of the boundaries set by the joints constraints. A ball and socket joint enforces a spherically connected relationship between two bodies at a fixed point on each. We have spoken previously of "applying a torque" to a joint, but a joint itself can't really apply a torque to itself—it applies it to the bodies connected to it.

To apply a torque to a joint we apply the torque to the first body connected to the joint and the inverse of the torque to the second body connected to the joint. In a multi-threaded physical simulation like NovodeX it's critical that the torque only be applied once per simulation time step—applying the torque multiple times per time step results in over-torquing the joint, which leads to instabilities.

Unfortunately we can't always apply equal and opposite torques to each body. As described previously, the initial orientation of the body in the rotational space of the joint may not be the initial orientation of the body in the rotational space of the simulation. Each torque must be put into the rotational space of the joint for each body:

$$\vec{\tau_A} = q_{Ainv}(\vec{\tau})q'_{Ainv} \tag{3.16}$$

$$\vec{\tau_B} = q_{Binv}(-\vec{\tau})q'_{Binv} \tag{3.17}$$

where q_{Ainv} and q_{Binv} are the inverse transforms from the previous section that put the current orientation of each body into the rotational space of the joint.

3.3 Mesh Contact

During our evolutionary process we use the physical simulation to throw an object at our virtual character. When the character is impacted by the object, we need to determine the closest triangle on the character's triangle mesh to the collision (the motivation for this is discussed in the next chapter). However, because the character is physically simulated as a set of connected boxes, the contact point in the physical simulation is on a box.

Each box that comprises the virtual character has associated with it a triangle mesh (for simplicity, it was decided that these meshes would not be interconnected). When the object thrown comes in contact with one of the boxes belonging to the character we receive a callback from the physical simulation software. We then find the closest triangle in the character's triangle mesh belonging to the box using a point-triangle distance algorithm.

A shortcoming of this system is that we do not test all triangles in all triangle meshes against the point of contact to see which is closest—we assume that the box hit will contain the triangle that was closest—when it may actually be possible for a triangle on another mesh to be closer. Testing all triangles in all meshes would substantially increase processing time however, so we feel the tradeoff is acceptable.

Chapter 4 – Evolution of Character Controllers for Collision Preparation

Our evolutionary process generates controllers that attempt to defend our virtual character from an incoming projectile by placing the character into a pose that prepares it for collision. This chapter describes the process by which we test, evaluate, and evolve controllers.

4.1 Testing Controllers

Before describing how controllers are evaluated and evolved, it's first important to understand the process used to test a controller. Every controller generated must go through the same rigorous testing process before it is scored and handed to the evolutionary process for optimization.

Each test performed takes as input a projectile threat and the parameters necessary for the controller to attempt to defend the character from the threat. In this section we describe the process by which a controller is tested.

4.1.1 Testing Overview

The evolutionary process generates a large set of controllers. A controller's task is to defend our virtual character from the projectile. Each controller is tested in several stages according to a *joint failure* system (described later). During each stage the controller is tested multiple times using a *threat grid* (also described later). The performance of the controller during each test is computed by a fitness function that measures the amount of pain perceived by the virtual character.

After every controller has been evaluated in the current set, (we refer to a set of controllers as a *generation*), the evolutionary process takes the best controllers of the set and uses them as the basis to generate a new set. This cycle repeats until the evolutionary process decides it has found the best possible controller.

Figure 4.1 describes the algorithm CONTROLLER-TEST we follow to test an individual controller. There are 4 stages to the testing process. Each stage tests the controller 9 times according to the threat grid, for a total of 36 tests per controller. The output of the testing algorithm is the *fitness* of the controller, a scalar value used by the genetic algorithm as the basis for optimization.

4.1.2 Test Configuration

The input to the testing process is a controller and a projectile. The controller is specified by desired joint angles, and the projectile is specified by a direction vector ($\vec{d}$) and a speed (v). We assume all projectiles are aimed at the character's head and that all projectiles originate the same distance away from the character. For

```
 inputs: d⃗, direction of projectile
         v, speed of projectile
         s, a solution
         FITNESS-FN, function to evaluate performance of a solution
         CONTROLLER, takes a solution and applies it to the character
initialize fitness to 0
for all joint failure scenarios do
  for all threat grid scenarios do
    initialize CONTROLLER with s
    modify d⃗ according to threat grid
    launch projectile from d⃗ at speed v
    while simulation is running do
      at time t_delay start CONTROLLER
      evaluate FITNESS-FN, add result to fitness
    end while
  end for
end for
```

Figure 4.1: The CONTROLLER-TEST Algorithm

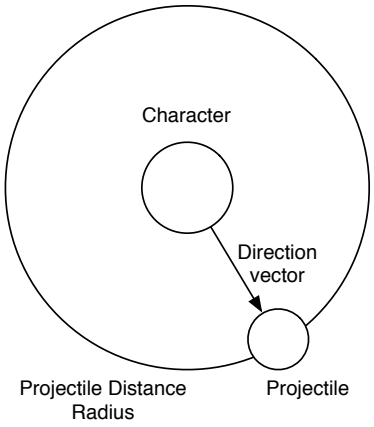

Figure 4.2: Projectile setup

simplicity's sake, we also make the projectile immune to gravity. This configuration is diagramed in Figure 4.2.

For each test, we begin with the character in a neutral position (see Figure 4.3). When the simulation begins, the projectile is launched towards the character. The controller is not allowed to begin operating until a specified time delay (t_{delay}) has elapsed. After t_{delay} the controller is applied to the physical simulation and the joints begin to move towards their desired joint angles. The controller uses a ball and socket joint control system that was described in Chapter 3.

Varying t_{delay} may result in different solutions as some defensive poses take more time to reach than others. This delay was inspired by observations of human blocking; depending upon how much time we have to respond to a threat, we may choose a different kind of preparatory pose.

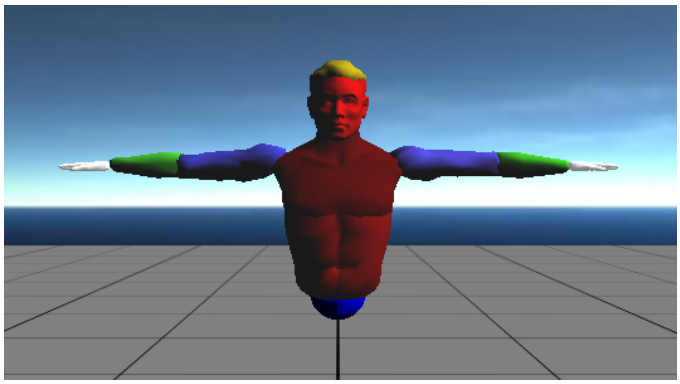

Figure 4.3: Character model mesh. The model is color-coded to show each pain multiplier area.

4.1.3 Joint Failure

As humans we rarely block an incoming projectile aimed at our face using only one arm. Unless we are bursting with confidence we typically protect our face using both arms. If the projectile is especially menacing we usually take additional measures to protect our face, such as turning or ducking our head [12, 15]. Intuitively, humans and animals tend to take multiple protective measures as part of a defensive "backup strategy" in case one or more limbs failed to perform their task. Inspired by this we introduced *joint failure* into our evolutionary process so that a controller is rewarded for using both arms and moving the head.

To simulate joint failure, we test a controller in multiple stages. During the first stage, the controller is allowed to behave normally. In the second stage we cause the character's right shoulder joint to fail–any input given to the right shoulder's joint controller is ignored. In the third stage we cause the left joint shoulder to fail. In the final stage, both shoulder joints fail.

4.1.4 Threat Grid

When we encounter a fast projectile threat aimed at our head, we typically do not take any chances. As described in the previous section, we use every means necessary to block the threat from hitting our sensitive areas: we use both hands, and we turn or duck our heads. We also tend to be very general in our approach to blocking the projectile: it's unusual for us to treat the threat as if we know *exactly* where it is going to hit us on our face. For example, we would never block only the very tips of our noses from a baseball, we would attempt to shield as much of our face as possible.

To prevent the evolutionary process from optimizing for a very specific point of impact, each solution is tested according to a *threat grid*. The threat grid modifies the direction of the projectile slightly so that the controllers are rewarded for taking a more general approach to blocking the projectile. Rather than optimizing on a specific threat case, it's encouraged to optimize on a set of cases.

The threat grid is a 3x3 grid that slightly alters the direction of the projectile; this results in 9 tests. During each test a different square is used in the threat grid to rotate the direction vector by the amount specified by the square. For example, the top-left corner of the threat grid rotates the direction vector up and to the left, whereas the bottom-right corner rotates the direction vector down and to the right. This concept is illustrated in Figure 4.4. In addition to the rotation given by the square, we also add a small amount of noise, or *jitter*. The combination of the threat grid and jitter causes the optimization process to find more general

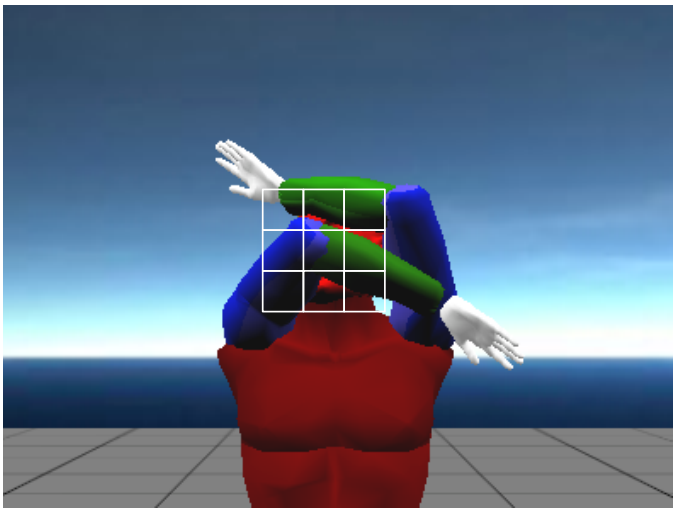

Figure 4.4: Threat grid, from the point-of-view of the projectile. (Exaggerated for demonstration purposes—the actual threat grid is much smaller).

solutions because it never anticipates the precise direction of the threat. Without the threat grid the controller is rewarded for developing solutions that reflect the projectile at an oblique angle[1].

4.1.5 Early-out Condition

A test concludes after a specified amount of time has elapsed or an early-out condition has been reached, such as the ball hitting the ground or the fitness exceeding a tuned threshold. Early-outs significantly improve the rate at which we converge on a solution.

[1]While these solutions are technically possible as a means of defending oneself, the specificity of their application makes them unrealistic as a means of blocking a projectile.

4.2 Evaluating Controllers

Our evolutionary process generates controllers that attempt to defend our virtual character from an incoming projectile. In order for the evolutionary process to optimize these controllers, we need a mechanism to gauge a controller's performance. This section describes how we evaluate a controller's *fitness*, the metric by which we measure how well a controller defends our virtual character.

A controller's fitness is determined by simulating the controller several times and measuring each test using our fitness function, FITNESS-FN. The fitness function measures the controller's ability to defend our virtual character during a test. (The testing process was described in the Section 4.1). The results of each test are combined to make up the overall fitness for the controller.

The goal of our controller is to defend our virtual character from an impending collision with a projectile. To score how well the controller performs its task, we rate the controller's performance by three metrics:

1. **Pain**: How effectively did the controller minimize the amount of pain perceived by the virtual character?

2. **Energy**: How much energy did the controller use to defend the character?

3. **Distance**: How far away did the controller block the projectile from the character's head?

The fitness function continuously evaluates the controller on these metrics while it is being tested. Each metric contributes a score; every time step these scores are

summed to determine the overall fitness of the controller:

$$fitness = pain + torque + distance \tag{4.1}$$

The fitness value is used by the genetic algorithm to optimize the controller as described in section X. In this section we explain the derivations of the *pain*, *torque*, and *distance* metrics.

4.2.1 Pain Metric

The goal of a controller is to defend our virtual character from an impending collision. Defending the character means to find a pose that causes the least amount of harm to come to the character. We measure harm using the *pain metric*, a scalar value that quantifies the amount of pain the virtual character perceived after it was hit.

Measuring pain in humans is a tricky problem—the recognized tools in the field of Medicine for evaluating pain are purely subjective [18, 17]. It has even been suggested that accurate measurement of pain will never be possible [3]. Since no system exists in the real world for quantifying the pain perceived as the result of a collision with an object[2], we were faced with no choice other than to invent one for our virtual character.

Our system for measuring the pain perceived by the virtual character is based on the simple premise that getting hit in some parts of your body hurts more than

[2]The creation of such a system would be quite sadistic!

others. Getting hit in your eyes would hurt more than your nose, which would hurt more than your face, etc. We encode areas of the character's body based on these relative sensitivities to pain. We call these areas *pain regions*.

Pain regions are implemented by colorizing areas of the character's triangle mesh with different color codes. When the projectile impacts the character model, we find the nearest triangle in the mesh and use this color to look up the pain multiplier (c_p) for the region. Figure 4.3 shows the pain region colorization on the character model. The table of pain multipliers we use is given in Figure 4.5.

When the character model is hit by the projectile, we take the point of impact (p_i) and pass this to the pain multiplier mapping function (P). P returns the pain multiplier for the region that was nearest to the impact. After we have the pain multiplier c_p we apply this to the force of impact $\vec{f_i}$ to determine the pain metric:

$$pain = \sum_i P(\vec{p_i}) \cdot |\vec{f_i}| \tag{4.2}$$

The pain metric is the primary means by which we evaluate a controller's ability to protect the virtual character.

4.2.2 Energy Metric

We reward a controller for discovering efficient solutions by using the *energy metric*. The energy metric observes the amount of energy being used by the controller while it is bringing the character model into its defensive position.

Point of Impact ($\vec{p_i}$)	Multiplier (c_p)
Hands	1.0
Chest	2.0
Upper Arm	2.0
Lower Arm	3.0
Stomach	4.0
Head	5.0
Face	10.0
Nose	14.0
Eyes	15.0

Figure 4.5: Pain Multipliers

We calculate the energy metric by accumulating the torque the controller applies at each of the character's joints every simulation time step:

$$energy = \sum_j c_t \cdot |\vec{\tau_j}| \tag{4.3}$$

where $\vec{\tau_j}$ is a torque vector applied at joint j and c_t is the multiplier that we use to scale the relative importance of the energy metric.

By picking a large value for c_t we encourage the evolutionary process to converge on solutions that use a low amount of energy—this prevents solutions where the controller grossly over-compensates its preparatory pose. However, picking *too* large a value for c_t has undesirable consequences—the energy metric overtakes the pain metric, and then the optimizer converges on solutions that only minimize energy. We used a value of 0.001 for c_t.

4.2.3 Distance Metric

It is desirable for us to obtain different styles of defensive postures, so we have introduced the *distance metric* to aid us in this task. The distance metric scores the controller based on how far away from the character's head it blocked the projectile. It is derived by taking the distance between the first point of impact ($\vec{p_0}$) and the center of the character's head ($\vec{h}$) and scaling this by the distance multiplier c_d:

$$distance = c_d \cdot \textsc{Round}(|\vec{p_0} - \vec{h}|) \tag{4.4}$$

We only consider the first point of impact due to precision errors in the physical simulator. When the projectile impacts the character the physical simulator may generate multiple points of impact during the process of resolving the collision. Accumulating the distance metric by handling multiple impacts would create inconsistencies between two impacts that appear identical.

The distance between the point of impact and the character's head is rounded to reduce the effect the threat grid system (described previously) has on the distance metric. The threat grid causes our test process to lose its determinism; rounding the distance helps keeps the distance metric less effected by the indeterminism.

Altering the scale of c_d will result in different styles of defensive poses being chosen by the optimization process. A large value will select poses that block the projectile far away from the character's head; a small value, due to the influence of the energy metric, will select minimal energy poses that block the projectile closer to the character's head. Like the energy metric, picking too large a value for c_d

will overpower the other metrics and result in poses that only block the projectile far from the character's head (and don't necessarily minimize pain). For our work we set c_d to 0.015.

4.3 Generating Controllers

In the first section of this chapter we described the process by which a controller is tested. In the second section we detailed how a controller is evaluated during a test and deriving its *fitness*. In this section we reveal how these controllers are generated.

To generate our controllers we use a genetic algorithm, an optimization process inspired by evolution [7]. Genetic algorithms operate on sets of *chromosomes*—string encodings of solutions to the problem being optimized. Pools of chromosomes, known as *generations*, are tested and evaluated according to a *fitness function*. After every chromosome in a generation has been be given a *fitness*, a selection process recombines and mutates the best candidates according to their fitness to create the next generation. This process repeats until a suitable candidate has been found.

This section describes:

- **Chromosome Format**: How we encode our controller solutions as chromosomes.
- **Selection Process**: How we select chromosomes for creating new generations.

- **Mutation and Crossover**: Our mutation process we use for creating new generations.
- **Determining End Condition**: How we know when to stop our optimization process.

4.3.1 Chromosome Format

Our chromosome encoding system interprets a chromosome as a string of unsigned integers which we refer to as *genes*. A gene corresponds directly to a desired joint angle using the following mapping:

$$f = 2\pi \cdot \frac{i}{i_{max}} - \pi \tag{4.5}$$

where f is the desired joint angle (in radians), i is the unsigned integer representation of the gene and i_{max} is the maximum unsigned integer that can be stored by the gene. Since our character model has 20 degrees of freedom, and we require a desired joint angle for each, our chromosome consists of 20 genes.

4.3.2 Crossover and Mutation

To create new chromosomes, we recombine previous chromosomes using *crossover* and *mutation*. Crossover takes two "parent" chromosomes and divides each at a split point. Part of one parent is then appended to the other to create a new

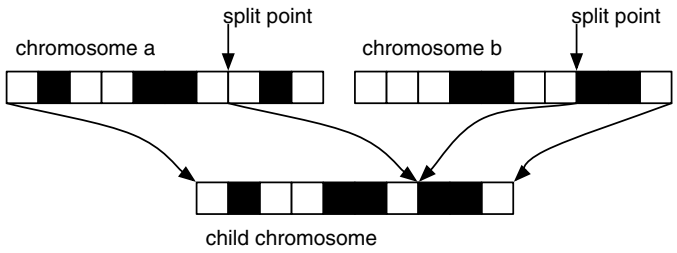

Figure 4.6: Chromosome Crossover between Two Parents

chromosome (see Figure 4.6). For our implementation, we perform crossover at the gene level. Taking part of a gene would effectively be splitting a joint angle into parts–a meaningless operation.

After a new chromosome has been created by crossover, we perform mutation on it. Our mutator takes as input a number of genes (n) and a distance (d) by which to modify a gene. A random gene is then selected and its value is increased or decreased by random amount modulo the given distance. The mutator may be called several times with different n and d values.

4.3.3 Tournament Selection Process

After an entire generation of chromosomes has been tested and a fitness value has been assigned to each, we use a tournament selection process to decide which chromosomes will continue on to the next generation and which will be used for creating new chromosomes. Our tournament process selects a group of n chromosomes from the current generation and sorts them on their fitness. The top m chromosomes are passed immediately on to the next generation unmodified (this

is called *elitism*). The remaining $n - m$ chromosomes are discarded. This process repeats until we have added enough chromosomes to the new generation to equal the previous.

To generate new chromosomes for the next generation, we take the chromosomes that were selected by elitism (the first m chromosomes), and then recombine them by using crossover. The resulting chromosomes are then mutated and added to the new pool. (Mutation and crossover rules were described in the previous section). Because recombination by crossover requires a pair of chromosomes, the size we choose for m must be an even number.

4.3.4 Determining End Condition

The genetic algorithm stops after a predetermined number of generations have been tested or when the system believes it has found an optimal solution. To determine if an optimal solution was found we compute the average fitness for each generation and apply a standard deviation over a window of prior generations. When the standard deviation falls below a tuned threshold we stop the simulation.

Chapter 5 – Assessment

5.1 Genetic Algorithm Parameters

Before we could begin experimenting with our collision preparation system it was important that we first develop a set of parameters for our genetic algorithm. The parameters used for mutation, the population size of each generation, and the tournament size used during selection all will effect the speed and outcome of the evolutionary simulation. An ideal set of parameters would find an optimum solution in the least time possible.

It's important to distinguish between *local* optimums and *global* optimums. Due to the nature of our problem domain, there is no way for us to be sure if a particular solution is a global optimum. During the genetic algorithm's process it may find several different local optimums—all with similar fitness values. When it comes time to pick a solution however, we only consider the solution with the best fitness value.

5.1.1 Mutation Scheme

The first set of parameters that we developed were the parameters used for the mutation system. The mutation algorithm takes as input a number of genes (n) and a distance (d) by which to modify the gene (this may be performed any number

Name	Algorithm
A	MUTATE($n = 10$, $d = 200$) MUTATE($n = 2$, $d = 1000$) MUTATE($n = 1$, $d = i_{max}$)
B	MUTATE($n = 10$, $d = i_{max}$)
C	MUTATE($n = 5$, $d = 1000$) MUTATE($n = 5$, $d = i_{max}$)
D	MUTATE($n = 20$, $d = 200$) MUTATE($n = 2$, $d = 1000$) MUTATE($n = 1$, $d = i_{max}$)

Figure 5.1: Mutation Schemes

of times, see Section 4.3.2). We experimented with four different schemes, given in Figure 5.1.

Each of the four mutation schemes were tested using a population size of 400 and a tournament size of 4. The results of these tests are given in Figure 5.2. After observing the results of the four mutation scheme tests it was decided to use mutation scheme A for subsequent evolutionary simulations. Mutation scheme A appeared to converge on optimal solutions faster than the other schemes.

5.1.2 Population Size and Tournament Size

The next set of parameters that needed to be determined were the population and tournament sizes we would use for our experiments. These parameters were developed by running a series of simulations from a population size of 100 to 800 in increments of 100, each with a tournament size of 4 and 8. Mutation scheme A was used throughout all simulations. The results are given in Figures 5.3 through

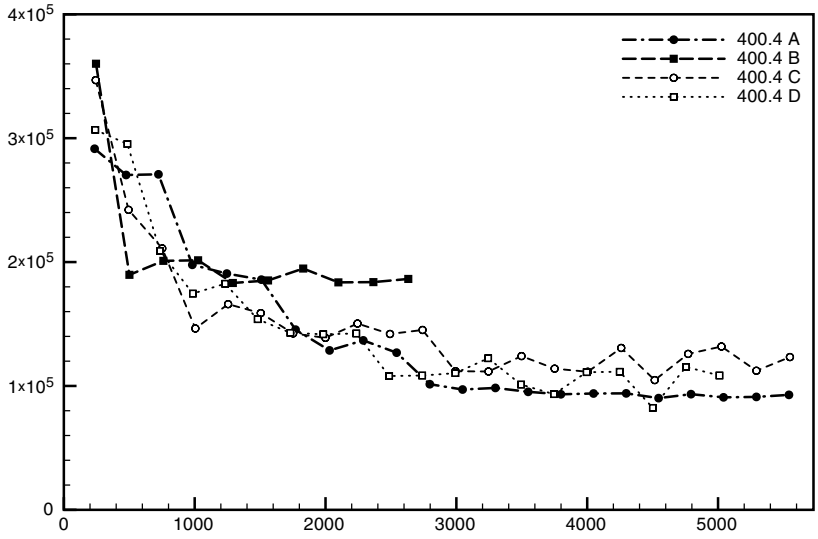

Figure 5.2: Fitness vs. Time. Mutation Schemes A, B, C, D. Population Size: 400. Tournament Size: 4.

5.6.

Comparing equal population size groups together we observed that tournament size did not have a reliable effect on the speed of convergence. A tournament size of 8 had an adverse affect on convergence for *some* population groups, but it appeared to improve convergence in others. Gritz et al [9] has reported that the size of the tournament seems to have little effect on the speed of convergence, so for our simulations we arbitrarily decided to use a tournament size of 4.

Overall we observed that large population size groups converged on a solution in a fewer number of generations than small population size groups. To evaluate which population size was the best to use for our experiments, we decided to plot our results over *time* rather than *generations*. While large population size groups

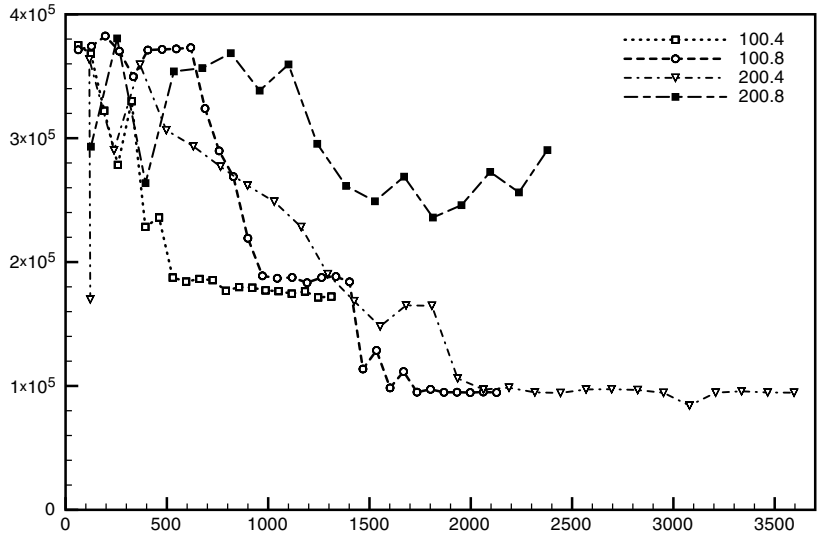

Figure 5.3: Fitness vs. Time. Mutation Scheme A. Population Sizes: 100, 200. Tournament Sizes: 4, 8.

converged in fewer generations, they took longer to simulate a generation than smaller population size groups. For example, the groups with a population size of 800 began to converge on a solution within 7 generations, but they took roughly 4000 seconds to get to that point. Conversely, the population size 400 groups began to converge in 11 generations, but it only took them about 3000 seconds to do so.

Another observation of the simulation results was that although some parameter sets took longer to converge on an optimal solution than others, most of them—with a few exceptions—did eventually converge. The sets that did not find an optimal solution most often fell victim our automated end-condition analysis (see Section 4.3.4), and we noted that if we started them back up, they did

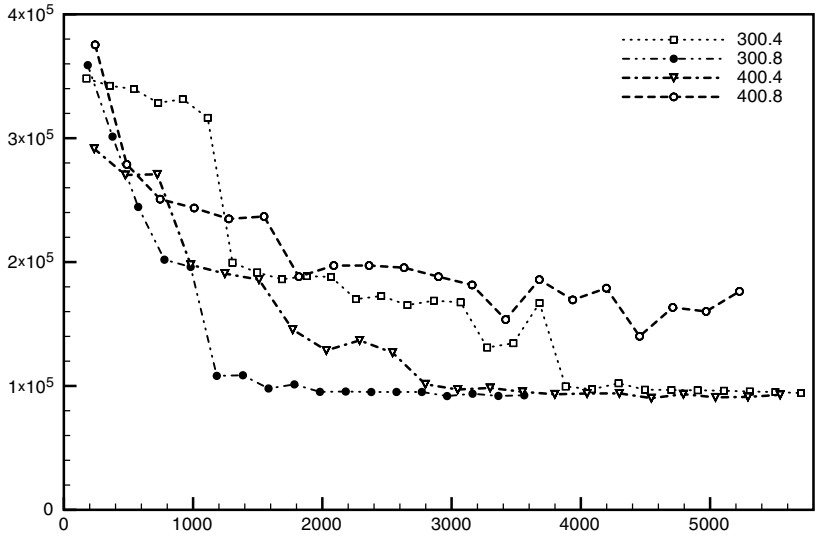

Figure 5.4: Fitness vs. Time. Mutation Scheme A. Population Sizes: 300, 400. Tournament Sizes: 4, 8.

eventually find one[1].

To aid in determining the best population size the fitness of the simulations over time were fit to spline curves and plotted together (see Figure 5.7). Here one can see that, for the most part, each simulation run gradually converges towards an optimal solution, but some converge faster and at a steadier rate than others. In the end we choose a population size of 400 because population sizes under 400 bounce around too much before finding a solution, and population sizes over 400 take too long to find a solution. 400 appears to be the best trade-off between simulation time and reliability of convergence.

[1]Sometimes, after many more hours of simulation!

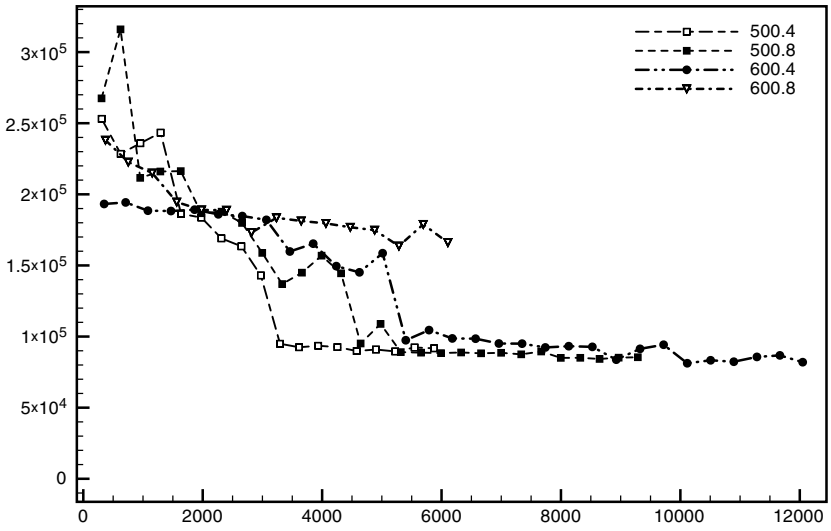

Figure 5.5: Fitness vs. Time. Mutation Scheme A. Population Sizes: 500, 600. Tournament Sizes: 4, 8.

5.2 Results

A variety of experiments were done to test the performance and flexibility of the collision preparation system. For each experiment the system was tested at several different angles varying from $-60°$ to $60°$ on the transverse plane (see Figure 5.8). We evaluate the system first as originally presented followed by modifications that place an emphasis on using the hands for blocking.

To improve determinism and accuracy of experiments, the physical simulation was set to run at a fixed time step of 0.0025s (400Hz). Larger time steps (greater than 0.005s) occasionally exhibited damper instabilities in the ball joint control system we developed. With graphic rendering disabled, the average simulation run

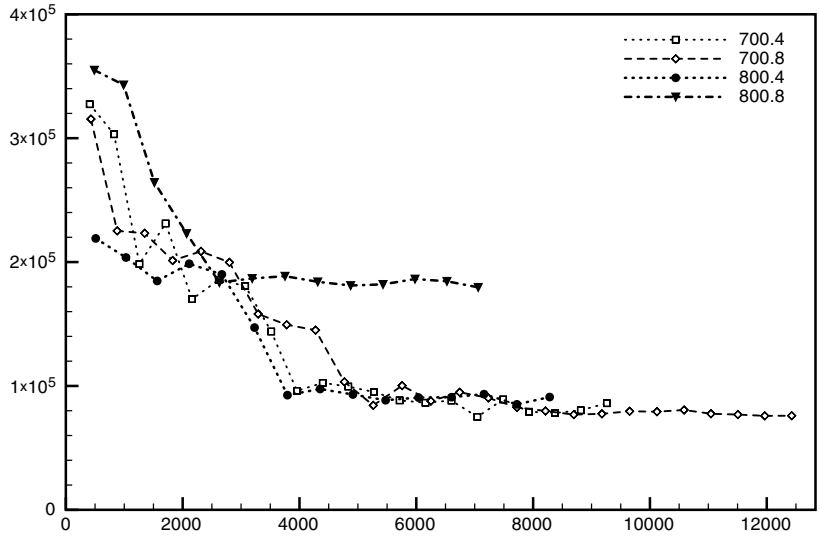

Figure 5.6: Fitness vs. Time. Mutation Scheme A. Population Sizes: 700, 800. Tournament Sizes: 4, 8.

took approximately 80 minutes to complete on an AMD Athlon 2600+ processor.

5.2.1 Collision Preparation System

The collision preparation system was able to find poses that protected the virtual character for a variety of threat angles. During evolution we observed that the system typically started with one arm blocking, then gradually transitioned to using two arms and ducking the head, as is shown in Figure 5.9. Results for various threat angles are given in Figure 5.10. (All figures are ordered left to right, top to bottom).

At certain threat angles the collision preparation system was able to converge

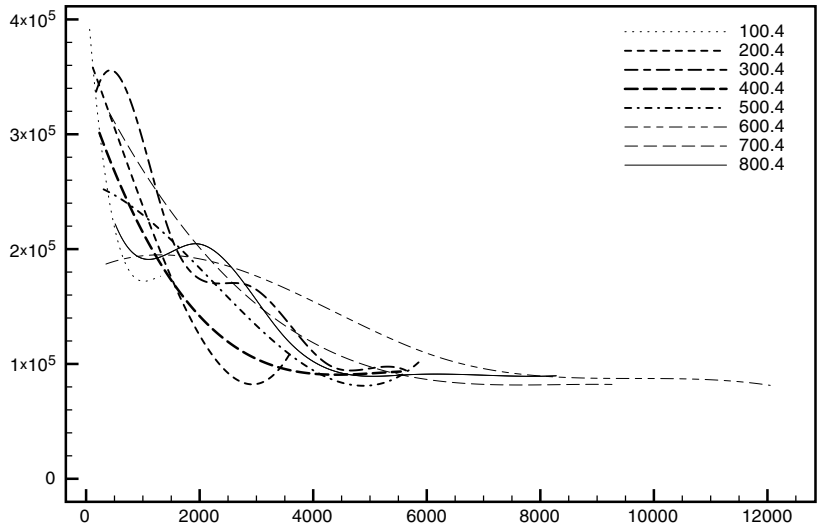

Figure 5.7: Fitness vs. Time. Spline Fit. Mutation Scheme A.

on a solution, but it did not look very convincing. For example, at a wide angle of 60° the system tended to converge on solutions that only used one arm. We believe this was due to two shortcomings in our system: the inability of the character model to twist its torso very far, and excessive inter-collision between the arms and head. Because our arms and head were simulated as rigid body blocks, the arms were not able to wrap around the face to opposite sides, which would allow the hands to aid in blocking the projectile.

The joint failure system was essential in aiding the evolutionary process converge quickly on a solution. Figure 5.11 shows the joint failure process at work: first both arms are allowed to operate, than each is failed, and finally neither arms are allowed to operate. The joint failure system forced the evolutionary process

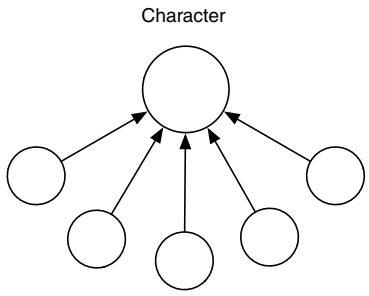

Figure 5.8: Projectile angles used in experiments (top-down view)

to try and find solutions that used both hands and moved the head to protect the face.

Overall, the majority of the solutions the system found looked very convincing—the character exhibited a reflex-like behavior that used all of the models faculties including both arms and turning the head. Solutions that failed to look convincing were ones in which the controller had the character block using the back of the hands or the back of the forearms. This problem could be overcome by creating new pain regions for these areas and configuring the pain multiplier to be higher.

5.2.2 Hand Emphasis

Although our system was successful in developing poses that defend a virtual character from an impending collision, the poses that it found may not have had the characteristics that we were anticipating. In an effort to seek out different types of poses we experimented with adjusting the parameters of the evolutionary process and attributes of the character model. One such pose that we desired was to have the controller block exclusively using its hands.

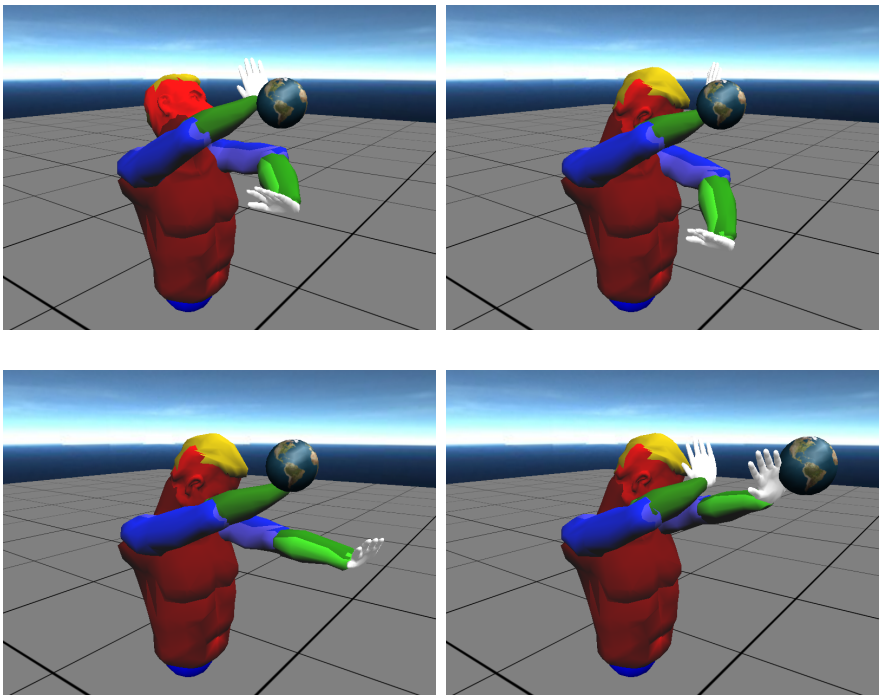

Figure 5.9: Controller at various stages of evolution

Our first attempt at having the controller block more with its hands was to adjust the distance metric constant c_d (see Section 4.2.3) so that the fitness function would place a higher emphasis on the distance between where the projectile was blocked and the character's head. The success of this approach was limited. Because the emphasis was placed on purely the distance from the head the controller would converge on solutions where the fingers were pointed out towards the projectile, which had limited applicability and looked very unrealistic.

In another approach we dropped the distance metric completely and modified the character model so that the arms did not collide with the projectile in the physical simulation. This approach was very successful in quickly developing con-

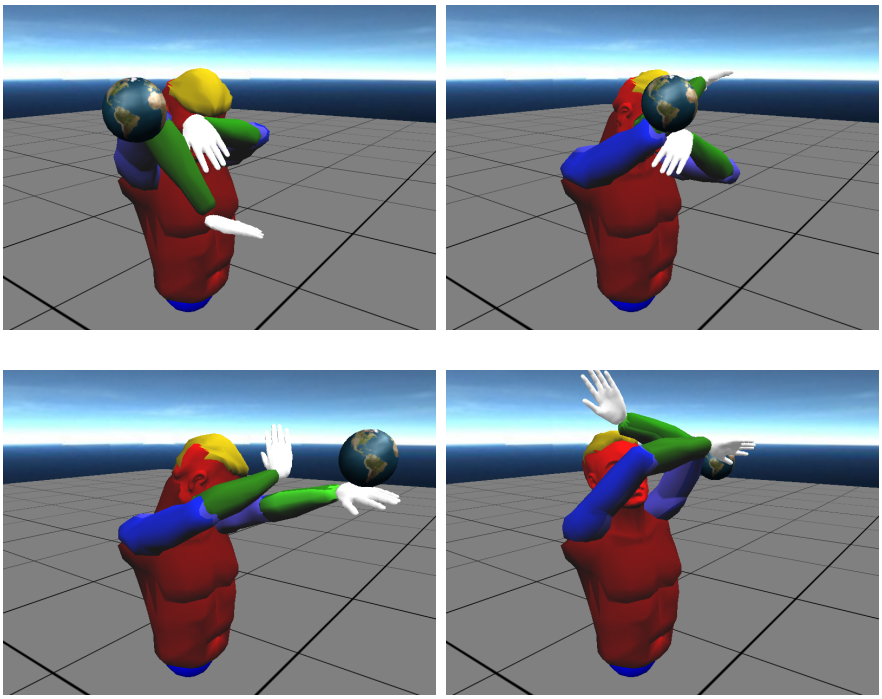

Figure 5.10: Results at various angles

trollers that blocked exclusively using the hands. Figure 5.12 shows the results for experiments where the model had hands-only collusion.

5.3 Discussion

We were successful evolving controllers that placed a virtual character in a variety of different defensive poses. The quality of our controllers was determined by our ability to create a fitness function that evaluated the effectiveness of generated controllers, and tuning parameters that were used by the fitness function. The pain metric, and the pain multipliers that went with it, was central to the fitness

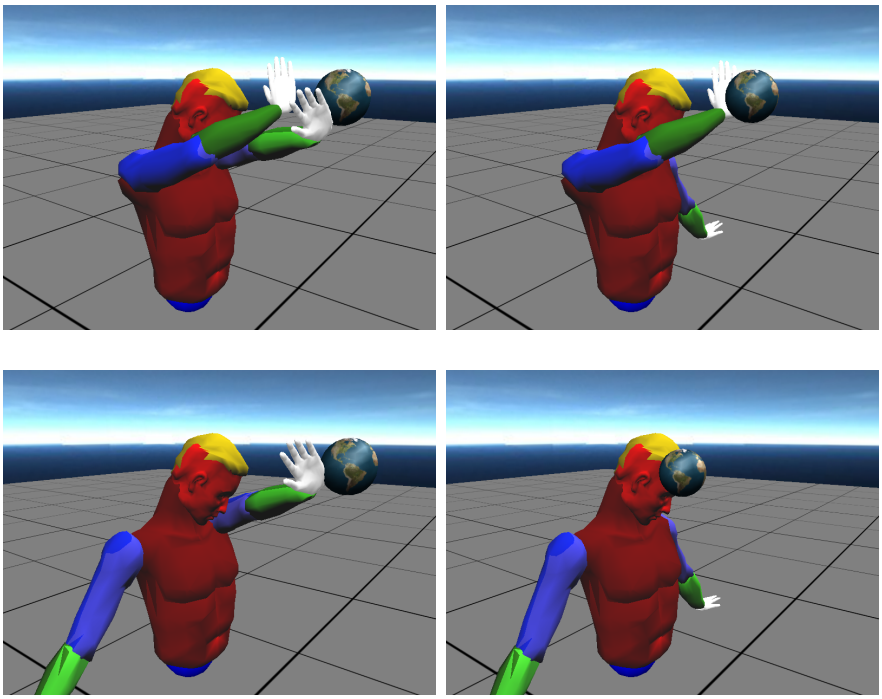

Figure 5.11: Four stages of simulated joint failure

function's effectiveness. By expanding the number of pain regions and defining more specific pain multipliers, it's feasible that using this system any kind of collision preparatory pose could be automatically generated.

Using our system, a content creator can develop poses for a virtual character without the use of motion capture. By modifying the parameters to the evolutionary process or the collision preparation system an animator could generate an endless supply of convincing poses in an automated fashion. Researchers can apply the control and optimization techniques here to a variety of character control problems.

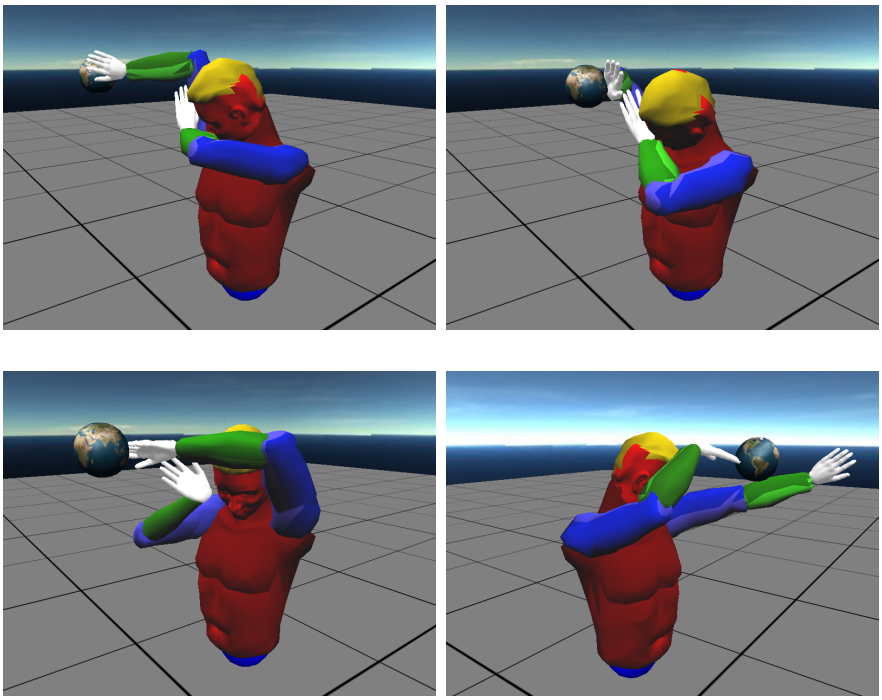

Figure 5.12: Results at various angles with hands-only collision

Chapter 6 – Conclusion

6.1 Summary

Realism in interactive applications is largely dependent upon how effectively the movements of human characters are conveyed. Since we are all inherently familiar with the properties of human motion, irregularities in human movement in an interactive application are easy to identify. Currently motion capture is the predominant technique for replicating human movement, but handling the infinite set of possibilities for human movement is not feasible under this technique. Physically simulating human characters and using controllers to generate dynamic human motion presents a rich solution for handling scenarios motion capture can not anticipate. Recent advances in physically simulated characters handle dynamic situations such as falling or being hit be an object. In this thesis we have presented a technique for generating physically simulated controllers that *prepare* a character for a collision in a reflexive, physically realistic manner.

Our technique for generating collision preparatory poses uses genetic algorithms to optimize a joint angle controller based on evaluation from a *pain measurement* system. The pain measurement system color-codes a character model with *pain regions* corresponding to the "sensitivity" of the character model to forces resulting from impacts in these areas. The testing process tests controllers generated by the

genetic algorithm multiple times according to a *joint failure* and *threat grid* system that encourage general solutions that defend the character using all of the faculties of the character model. By using this system, defensive poses may be automatically generated without the use of motion capture.

6.2 Future Work

6.2.1 Energy Metric Enhancements

The energy metric (Section 4.2.2) contributed to the evaluation of the controller by penalizing it for using excessive torque to move the character into the preparatory pose and maintain the pose after impact with the object. What the energy metric fails to consider is the way in which torque is being used. In humans, certain poses are chosen because they are better for "bracing" oneself for maintaining the pose after impact.

For example, our system occasionally generated poses that blocked using the back of the hands. This is not a common defensive pose in humans because the forearm muscle group that pulls the back of the hand is not as strong as the group that pulls the front of the hand. In the future, the energy metric should be enhanced to consider not only the amount of torque that is being applied, but also the direction it is being applied in—in this case, it should penalize the controller for using muscle groups that pull the back of the hands.

6.2.2 Joint Limit Penalties

Our system failed to consider solutions that placed the character into positions where the force from impact the object causes the character model to exceed its joint limitations. In situations where the controller made the character block using the back of its hands, impact with the object may have caused the hands to rotate in such a way that brought them almost in contact with the forearms. In humans this might cause the wrist joint to fracture.

To correct situations that in reality would cause a joint to fracture the system should penalize controllers that cause the character to exceed its joint limits (see Section 3.2.5). A possible solution might be to dramatically increase the energy metric multiplier (Section 4.2.2) when the joint exceeds a soft limit. A more sophisticated solution may be to model the amount of pain perceived when a joint exceeds a threshold, similar to how the current pain system models pain caused by forces external to the character model.

6.3 Towards Ultimate Realism

Since our control mechanism uses genetic algorithms to optimize desired joint angles, and the joints are moved to those angles using a ball-and-socket joint controller, the resulting motion does not look very human if it is slow or allowed to control the character for too long a time. Our system however is only designed to control the character for a brief period of time, and the gains used by the controller are "stiff" so that the character moves too quickly to notice any irregularities. Our

problem domain—reflexive responses to projectile threats—lends itself especially well to such a control scheme. Reflexive responses in humans are quick, "knee-jerk" reactions that require little higher-level brain power. Optimizing a pose controller such as ours will probably not work as well in other higher-order control domains.

The reason our system works well is because our task environment, even though it is a dynamic physically simulated environment, has only one episode from the point of view of the controller. The controller is presented with a threat (a direction and a velocity), and produces only one response to that threat (desired joint angles). The control system requires no decision making other than performing the low-level control task of moving the character into the position specified by the desired joint angles. In most control systems for physically simulated characters the input to the controller is continuously changing (current joint angles, forces, contacts, etc.), and the controller *is* required to perform some decision making (such as what desired joint angles to output). In this situation using genetic algorithms alone would not be a solution.

To achieve more realistic human motion for less reflexive situations (such as walking or falling) the control scheme must continuously process its sensors and produce new outputs. Here, genetic algorithms are useful for optimizing more complex control systems such as neural networks that produce torque or desired joint angle outputs. Genetic programming can also be used to generate a program that produces desired joint angles. The performance of an automated search algorithm becomes a question of the size of the search space. The larger the neural network or genetic program, the more realistic a solution may be; but a larger search space

means a smaller likelihood a solution can be found in a reasonable amount of time (if at all).

As you increase the size of the search space, you also increase the number of possible solutions—and just because a solution is *possible* doesn't mean it's necessarily a *desired* solution. A larger search space requires a better evaluation system (fitness function, in the case of genetic algorithms). In addition, with human motion an animator may desire to have more artistic control over the "style" of the motion being generated. Giving an animator the ability intuitively specify the types of desired solutions—and having an intelligent evaluation system help develop those solutions—is the *true* goal of evolving character controllers.

In this thesis we have presented a very small contribution to the larger goal of evolving the ultimate character controller: measuring perceived pain so that a character may evolve reflexive collision preparatory behaviors. Integrating pain perception with other types of evolved control systems may lead to more realistic human motion. Hopefully one day an animator will be able to specify a motion for a virtual character such as tackling or throwing a punch, and an evolutionary process can automatically re-target the motion in such a way that minimizes (or maximizes) pain for the characters involved. After all, virtual humans have feelings too!

APPENDICES

Appendix A – Character Description

For this work we developed a character model file format for describing articulated models that will undergo evolution. Our file format describes the hierarchy of body segments within the character model, joint location between body segments, joint limits, and parameters for our joint servos.

The file begins with the `bones` section. The `bones` section contains one or more `bone` sections. Each `bone` describes the name of the bone (`name`), the bone's parent (`parent`), and the location of the joint connecting the bone to its children (`joint`). Additionally, each `bone` may contain up to three `dof` sections to describe the degrees of freedom of the bone's joint, a `dim` section that gives the depth and height of the bone, and a `pd` section that specifies the parameters for the PD-servo that will drive the joint. The relationships between these sections are given in Figure A.1.

- `name` - The name of the bone. If the bone name ends in '`.mesh`' then a triangle mesh file is used to render to render the bone and perform point collision when evaluating the pain metric.
- `parent` - The name of the bone's parent (root bone is empty).
- `joint` - The position of the bone's end effector (x, y, z) and orientation of the bone's children (r_x, r_y, r_z).

```
bones {
  numbones
  bone {
    name { bonename }
    parent { bonename }
    joint { x y z rx ry rz }
    origin { x y z }
    dof { x y z limits { min max } }
    ...
    dim { height depth }
    pd { k kd }
  }
  ...
}
```

Figure A.1: Character Model File Format

- `origin` - The origin of the bone (x, y, z) relative to its parent. This is useful with bones such as the upper arm, where the parent's end effector (the chest) would be located at the neck but the axis of rotation should be at the shoulder.

- `dof` - The axis of rotation (x, y, z) for one rotational degree of freedom of the bone's joint. A bone may have up to three rotational degrees of freedom.

- `limit` - The joint limits $(min, max$ - in radians) for one rotational degree of freedom of a bone's joint.

- `dim` - The width of the bone is given by the distance between the end effector and the origin. This attribute gives the dimensions $(height, depth)$ of the bone. The box formed by the width, height and depth of the bone are used

for collision in the physical simulation.

- `pd` - The k and kd parameters used for powering the joint's PD-servo.

The character model file used for our simulations is given below.

```
bones {
  9
  bone {
      name { Hips }
      parent {  }
      joint { 0.000000 5.000000 0.000000 0.000000 0.000000 0.000000 }
  }
  bone {
      name { chest.mesh }
      parent { Hips }
      joint { 0.0 5.52 0.0 0.0 0.0 0.0 }
      dof { 1.0 0.0 0.0 limits { -0.05 0.05 } }
      dof { 0.0 1.0 0.0 limits { -0.5 0.5 } }
      dof { 0.0 0.0 1.0 limits { -0.05 0.05 } }
      dim { 3.3 2.17 }
      pd { 5000 100 }
  }
  bone {
      name { head.mesh }
      parent { chest.mesh }
      joint { 0.0 2.45 0.0 0.0 0.0 0.0 }
      dof { 1.0 0.0 0.0 limits { -0.5 0.7 } }
      dof { 0.0 1.0 0.0 limits { -0.5 0.5 } }
      dof { 0.0 0.0 1.0 limits { -0.4 0.4 } }
      dim { 1.76 1.56 }
      pd { 100 10 }
  }
  bone {
      name { upperarmr.mesh }
      parent { chest.mesh }
      joint { 5.35 -1.0 -0.5 0.0 0.0 0.0 }
```

```
    origin { 2.0 -1.0 -0.5 }
    dof { 1.0 0.0 0.0 limits { -1.2 1.2 } }
    dof { 0.0 1.0 0.0 limits { -2.2 1.2 } }
    dof { 0.0 0.0 1.0 limits { -1.2 1.2 } }
    dim { 1.0 1.0 }
    pd { 400 10 }
}
bone {
    name { lowerarmr.mesh }
    parent { upperarmr.mesh }
    joint { 2.96 0.0 0.0 0.0 0.0 0.0 }
    dof { 0.0 1.0 0.0 limits { -2.4 0.0 } }
    dim { 0.77 0.63 }
    pd { 300 5 }
}
bone {
    name { handl.mesh }
    parent { lowerarmr.mesh }
    joint { 1.4 0.0 0.0 0.0 0.0 0.0 }
    dof { 0.0 1.0 0.0 limits { -0.5 0.5 } }
    dof { 0.0 0.0 1.0 limits { -1.0 1.0 } }
    dim { 0.2 0.4 }
    pd { 100 1 }
}
bone {
    name { upperarml.mesh }
    parent { chest.mesh }
    joint { -5.35 -1.0 -0.5 0.0 0.0 0.0 }
    origin { -2.0 -1.0 -0.5 }
    dof { 1.0 0.0 0.0 limits { -1.2 1.2 } }
    dof { 0.0 1.0 0.0 limits { -1.2 2.2 } }
    dof { 0.0 0.0 1.0 limits { -1.2 1.2 } }
    dim { 1.0 1.0 }
    pd { 400 10 }
}
bone {
    name { lowerarml.mesh }
    parent { upperarml.mesh }
```

```
        joint { -2.96 0.0 0.0 0.0 0.0 0.0 }
        dof { 0.0 1.0 0.0 limits { 0.0 2.4 } }
        dim { 0.77 0.63 }
        pd { 300 5 }
    }
    bone {
        name { handr.mesh }
        parent { lowerarml.mesh }
        joint { -1.4 0.0 0.0 0.0 0.0 0.0 }
        dof { 0.0 1.0 0.0 limits { -0.5 0.5 } }
        dof { 0.0 0.0 1.0 limits { -1.0 1.0 } }
        dim { 0.2 0.4 }
        pd { 100 1 }
    }
}
```

Bibliography

[1] Ageia Technologies Inc. The Novodex physics software development kit, 2006. ageia.com.

[2] William W. Armstrong and Mark W. Green. The dynamics of articulated rigid bodies for purposes of animation. *The Visual Computer*, 1(4):231–240, 1985.

[3] H. Beecher. The measurement of pain. *Pharmacological Reviews*, 9:59–209, 1957.

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

[5] Hugo de Garis. Genetic programming: building artificial nervous systems using genetically programmed neural network modules. In B. W. Porter and R. J. Mooney, editors, *Machine Learning: Proceedings of the Seventh International Conference*, pages 132–139, Austin, TX, 21-23 1990. Morgan Kaufmann, Palo Alto, CA.

[6] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 251–260. ACM Press, 2001.

[7] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[8] Larry Gritz and James K. Hahn. Genetic programming for articulated figure motion. *Journal of Visualization and Computer Animation*, 6(3):129–142, 1995.

[9] Larry Gritz and James K. Hahn. Genetic programming evolution of controllers for 3-D character animation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo,

David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 139–146, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[10] James K. Hahn. Realistic animation of rigid bodies. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 299–308, New York, NY, USA, 1988. ACM Press.

[11] Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O'Brien. Animating human athletics. *Computer Graphics*, 29(Annual Conference Series):71–78, 1995.

[12] Sheila M. King, Caroline Dykeman, Peter Redgrave, and Paul Dean. Use of a distracting task to obtain defensive head movements to looming visual stimuli by human adults in a laboratory setting. *Perception*, 21(2):245–259, 1992.

[13] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[14] Joseph Laszlo, Michiel van de Panne, and Eugene Fiume. Limit cycle control and its application to the animation of balancing and walking. *Computer Graphics*, 30(Annual Conference Series):155–162, 1996.

[15] F.-X. Li and M. Laurent. Dodging a ball approaching on a collision path: effects of eccentricity and velocity. *Ecological Psychology*, 13(1):31–47, 2001.

[16] Michael Mandel. Versatile and interactive virtual humans: Hybrid use of data-driven and dynamics-based motion synthesis. Master's thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2004.

[17] Margo McCaffery and Chris Pasero. *Pain: Clinical Manual*. Mosby Inc., 1999.

[18] R. Melzack. The McGill pain questionnaure: Major properties and scoring methods. *Pain*, 1:277–299, 1975.

[19] Natural Motion, Ltd. Dynamic motion synthesis, 2005. naturalmotion.com.

[20] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press, 2000.

[21] Marc H. Raibert. *Legged robots that balance*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1986.

[22] Marc H. Raibert and Jessica K. Hodgins. Animation of dynamic legged locomotion. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 349–358, New York, NY, USA, 1991. ACM Press.

[23] Jonathan Roberts, D. Kee, and G. Wyeth. Improved joint control using a genetic algorithm for a humanoid robot. In *Proc. Astralasian Conference on Robotics and Automation*, Brisbane, December 2003.

[24] Ken Shoemake. Animating rotation with quaternion curves. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 245–254, New York, NY, USA, 1985. ACM Press.

[25] Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM Press, 1994.

[26] Russell Smith. *Intelligent Motion Control with an Artificial Cerebellum*. PhD thesis, University of Auckland, Auckland, New Zealand, 1998.

[27] S. Takashima. Dynamic modling of a gymnast on a high bar. In *IEEE Internation Workshop on Intelligent Robots and Systems*, pages 955–962, Osaka, Japan, 1990.

[28] F. Thomas and O. Johnson. *Disney Animation: The Illusion of Life*. Abbeville, New York, 1984.

[29] J. Wilhems and B. Barsky. Using dynamic analysis to animate articulated bodies such as humans and robots. In *Graphics Interface 85*, pages 97–104, Montreal, 1985.

[30] Krister Wolff and Peter Nordin. Learning biped locomotion from first principles on a simulated humanoid robot using linear genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2003*. Morgan Kauffman, 2003.

[31] Gordon Wyeth, Damien Kee, and Tak Fai Yuk. Evolving a locus based gait for a humanois robot. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, Las Vegas, October 2003.

[32] Victor B. Zordan and Jessica K. Hodgins. Motion capture-driven simulations that hit and react. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 89–96. ACM Press, 2002.

[33] Victor Brian Zordan, Anna Majkowska, Bill Chiu, and Matthew Fast. Dynamic response for motion capture animation. *ACM Trans. Graph.*, 24(3):697–701, 2005.