CONTROLLER FOR JUMPING ANIMATIONS TO ACHIEVE TARGET POSITIONS

By

Ian Charles Ooi

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:

Barbara Cutler, Thesis Adviser

Charles Stewart, Member

Shawn Lawson, Member

Rensselaer Polytechnic Institute
Troy, New York

November 2015
(For Graduation December 2015)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank Rosa Tung and Jim McCarthy for acting as consultants for existing artistic methods as well as usage of tools which allowed me to produce the mesh and rig used in this thesis and showed me an artist's view of animation. I would like to thank Jon Holman for discussing the mathematics of optimization problems with me, which helped to direct my choice for how to solve the problems.

ABSTRACT

Animating a character for video games and films is difficult and time consuming, requiring hours of artist labor to produce each animation. These animations are set and inflexible, requiring changes to the animation or sometimes fully new animations to suit new characters or situations for natural looking movements. Jumping is one such animation, where the size, mass, strength, and environment affect the movement of the character. Traditionally these animations are produced by manually posing the character for certain key frames and interpolating between the frames to produce a smooth animation. The more detailed or lengthy an animation, the more work required to specify it. Physics-based simulation for animation production can reduce this work, creating animations for a variety of situations based on constants set for the character and environment. These animations can then be easily recreated or adjusted for different environments by changing the constants set for generation.

This thesis work presents a simulation-based method of control for a character, focusing on the lower body, to produce jumping animations for a variety of situations and body parameters. Two methods of simulation are described, one using a torque calculation and the other using an energy calculation to determine poses for the character. Our simulation takes as input a mesh representing the character, a tree of joints describing the skeleton, a set of muscles, mass assigned to each limb of the body, and a description of the desired path through desired timings, gravity, and desired displacement. An inverse kinematic solver is used to aid in posing the character.

Contributions of this thesis include an implementation of a simulation to produce jump animations in Unity3D, a description of character poses based on torque as well as another based on energy, a sampling-based method for choosing a target position, and visualization of the produced animations in several ways to aid in debugging, analysis and presentation.

CHAPTER 1
INTRODUCTION

Animations of human characters are used heavily in video games, movies, and other fields. Especially with the increasing usage of complex environment traversal in both film and video games, many similar animations of athletic motions must be created with small changes to tune the motion to the particular situation, environment, and character. Creation of such animations is largely done by hand by artists, posing the character for each time step of the animation. In 2D animation this takes the form of a sprite sheet as shown in Figure 1.1. These sprites may be drawn by hand or generated through 3D models.

In 3D artists will frequently use a method called key framing, a modification of a method used for producing 2D animations. In a key frame animation, certain "key" parts of the animated sequence are specified, with the remaining frames filled in in a process called "in betweening or "tweening," using an automated interpolation method or manual frame addition. For 2D animation, the artist will need to generate the intermediate images themselves or have a program interpolate between images. In 3D, key frame animations are performed on a 3D model, storing transformation data about the model for each frame and playing back the animation by repeating the transformations, interpolating between them to produce smooth animations.

Specifying these animation frames is work intensive, taking up significant time and resources to produce for a single character. Additionally, similar animations may need to be produced for slightly different scenarios, with only minor modifications required. These minor modifications can be made to fit a different setting, such as a character jumping on Earth or on the moon, or can be for different characters, such as a large person moving in contrast with a small child. Though the movement itself may be similar, manual changes must be made, requiring artist time which could be spent generating new assets.

Recent work in animation generation seeks to automate this process, replacing the manual process with a procedural one. Physics-based simulations can be used

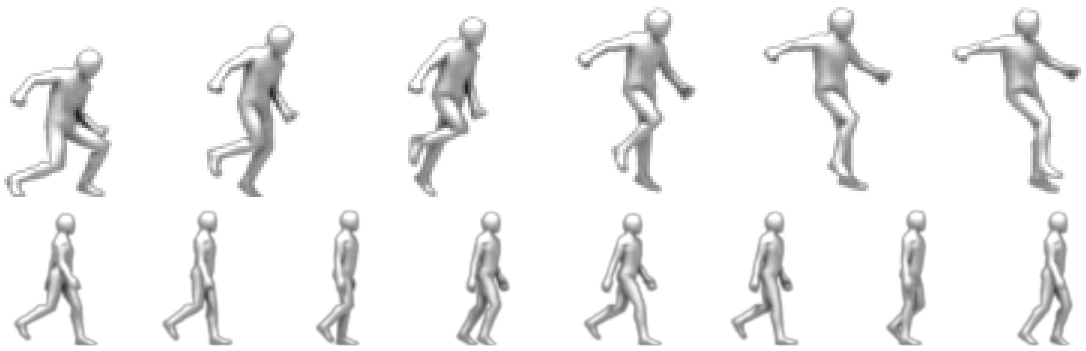
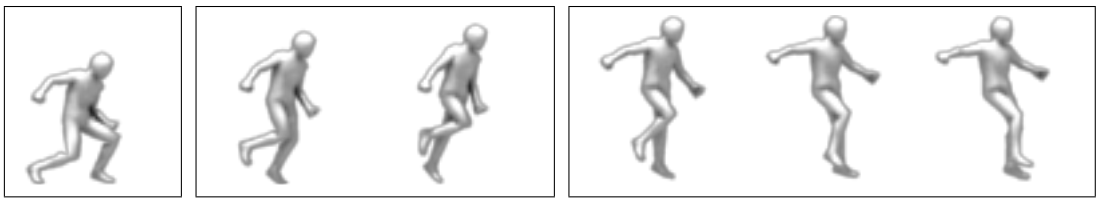


Figure 1.1: This example shows a 2D sprite sheet used to produce a jumping animation (top row) and a walking animation (bottom row) for a stick figure character. The frames in this case are laid out in a single image for demonstration purposes, progressing in order starting with frame 0, the frame farthest left in this sprite sheet. An artist may create key frames and then fill in intermediary poses to produce a full sprite sheet as above. These sprite sheets were created by Clint Bellanger using a 3D model and are licensed under Creative Commons Attribution 3.0, retrieved from opengameart.org.

to produce controllers for the skeleton, determining joint positions and rotations for keyframes automatically. Not only does this reduce the effort involved in the creation process, but this also provides a basis for dynamic interaction between a character's animation and the environment. With manual keyframe animations this is not possible, as any specific interaction between a character and the environment must be manually created by an animator

We present a controller that takes a skeleton as input, with additional parameters describing the character. We then simulate a jumping motion on the character, determining a sequence of poses based on the character's muscles to produce a keyframe animation. The additional character parameters describe the character's mass, muscles, the constraints placed on each joint to prevent unnatural rotations, and a description of the jump indicating desired time and distance. Mass is specified per-limb, with each mass stored in the joint object affecting the limb to allow for calculation of the character's center of mass.

We divide a jump into five stages: path estimation, windup, thrust, in air maneuvering, and landing. Our controller works with the initial path estimation,



(a) Windup **(b) Takeoff** **(c) In-Air**

Figure 1.2: Above is an example 2D sprite animation of a simple human character jumping, with the jump divided into the windup, takeoff, and in-air stages. A landing is absent from this sequence. The windup consists of a short crouch to prepare for the jump, creating an opportunity for the body to extend and thus accelerate. The takeoff performs this acceleration, shifting its center of mass forward past its feet and the character becomes airborne. During the in-air phase the character moves its body to control the fall, in this case spreading its arms and shifting its feet from behind its pelvis to in front of its pelvis.

windup, and thrust stages of the motion as shown in Figure 1.2. Poses are calculated by modeling muscles as Hookean springs attached to the skeleton at 2 points and crossing a joint as shown in Figure 1.3. Spring constants for the muscles are specified by the user, allowing our simulation to animate characters of various strengths and body makeups. Spring displacement from rest is calculated using the bend in the joint and constants describing the skeleton. Using the spring displacement, a particular pose of a joint is related to the usage of the character's muscle and the elasticity of the connective tissue of the muscle and joint.

Two approaches are described in this thesis, one using torque and one using elastic energy. As the springs contract, they produce forces on the bones, which result in torques at each joint. These torques are then used to calculate a windup pose for the character by calculating resultant linear acceleration for the character's center of mass, which is compared to a calculated necessary acceleration to make the jump. This linear acceleration is then used to compute the motion of the takeoff and in-air phases of the jump, which are sampled to create frames of an animation. Our second approach uses the spring displacement to calculate the elastic potential energy of each muscle, which is modeled by a single linear spring. By assuming

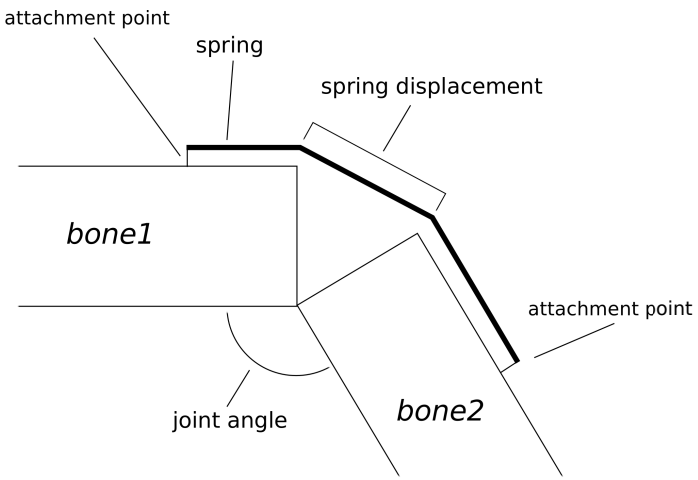


Figure 1.3: Visual representation of the setup of bones and muscles for a single joint. This illustrates the method by which we calculate the spring displacement, taking the bones as rigid blocks which separate and stretch the string as the joint bends.

perfect conservation of energy, we compare the total elastic potential energy of the character's muscles to a calculated necessary kinetic energy for the jump to be completed.

To control the motion and maintain plausibility, we calculate the character's center of mass and supporting polygon each frame. This allows the character to maintain balance while flexing its joints, the position of the character's joints are adjusted to keep the center of mass positioned over the supporting polygon. While flexing, the character constantly checks the position of its center of mass against the calculated supporting polygon, ensuring the center of mass is over the support and as close to the center as possible.

We use an inverse kinematic solver to help determine poses of intermediate joints. This allows us to find the necessary pose for the ankle and knee of the character given the position of its foot and hip. Many solutions in this region are possible, so we constrain the skeleton such that only those poses where the joints are within human ranges of motion are possible.

The next stage of the motion is the thrust stage where the character releases from the ground by using the potential energy of the spring-muscles to accelerate

its center of mass upward. This is handled by applying the calculated accelerations from the windup stage to the character's pelvis to accelerate the center of mass, with the inverse kinematic solver determining the poses of the legs as they unbend. Balance must be maintained, and the relative rate of rotation of the different joints of the leg must be balanced with each other to maintain foot position while the body is transformed. As the skeleton is a tree with its root at the character's pelvis, this can be a challenge, necessitating the use of the inverse kinematics solver. Due to the nature of the inverse kinematics algorithm used, the extension propagates from the hip to the feet.

The character then proceeds through the in-air portion of the jump, where the acceleration changes due to gravity before they finally land. We assume a simple trajectory for the in-air phase, though more complex motions with turns, flips, or interaction with the environment could be created. Our simulation ends when the character's feet contact the ground, ending the in-air phase. Other work, such as that by Ha et al., has handled landing and creating a separate controller to handle this is beyond the scope of this thesis. [1]

Our controller is made to be a module, able to be used with other controllers as part of a larger system. Its modularity is in the form of detecting and handling its state at each stage, consisting of its position, velocity, acceleration, pose, muscle state, and any collisions or forces applied. The controller performs actions when it has a response to its current state and ends control when it no longer has an appropriate response to allow a separate controller to handle the situation. This allows each controller to do a smaller job well. Several such controllers can be connected to produce more complex animations or animation sequences, utilizing bounded starting and finishing conditions for the character as well as bounds on expected conditions during operation to allow handling of stimuli during operation.

1.1 An Example of Animation Creation

To motivate the necessity for an automated animation method, we present an example of the method for producing an animation by hand. First a mesh and rig must be produced, which we also require for our simulation. This is a time and

skill intensive process, requiring the artist to exercise their technical and creative knowledge and ability to create the character's figure, the mesh, and to create and attach a skeleton, the rig. The mesh is constructed of faces which are in turn made of vertices, and each vertex must be assigned a weight for each joint of the rig to indicate the way the vertex should transform when the joint in the rig is transformed. These joints serve not only as a joint in the anatomical sense, but as a tracked point in the skeleton. The joints are connected by bones, but the information is stored at the joint, which sometimes necessitates a joint to be placed in a non-anatomical way. For example, in our rig we have a joint placed at the head, which not only allows us to rotate the head but tracks the position. A similar situation arises with the toe and heel, where a joint is used to facilitate the placement of a connecting bone. Once the mesh and rig are created and the weights for the rig are assigned to the mesh, an animator may create controls to manipulate several joints at once, such as creating a controller to facilitate grasping motions in which many bones of the hand must be manipulated at once. These controls are formed from an object such as a simple spline to which several joints are constrained, allowing movement of several joints through manipulation of the control object.

Once the rig and mesh are set up, an animator must manipulate the mesh to pose the model. Study of real subjects may be used to help ensure realism, suc For a jumping motion they would need to decide how they want to start the animation to allow blending from other movements such as walking, running, or standing. They must then position the model and record a key frame. Key frames are hand-created frames of the animation which a renderer or game engine may interpolate between to produce a final animation, allowing an animator to create and store few frames, while still creating a 60 frame per second animation in the final setting. This also allows few frames to be stored to produce any frame rate of animation, while also reducing the storage requirements in exchange for a minor increase in computation, which is an extra interpolation for each joint in the character.

While the key frames can be sparse and there are tools for aiding in generation, this process requires heavy manual input for each animation for each character, as well as prior knowledge of the motion to be produced. Characters that move dif-

ferently due to differences in weight or body makeup must be animated separately, requiring an artist to perform similar, time consuming work. For example, a character with extremely strong legs may bend less, applying the same force as a weaker character over a shorter distance to achieve the same acceleration. A heavier character would need to apply much more force to achieve the same acceleration and thus jump height as a lighter character, necessitating stronger muscles or a greater distance over which the force is applied, thus requiring a deeper squat than the light character. With a simulation-based approach this work could be reduced, especially repeated work, to setting variables such as weight, height, and musculature for the different characters and generating the set of animations desired. As techniques and technologies improve, a simulation could be used in place of a stored key frame animation, creating the animation in place based on its environment and situation, thus shifting the burden on an animator from preparing poses and keyframes to adjusting body dimensions, weight, strength, and constraints.

For a jump specifically, a human loads the muscles in their legs, quickly moving to a particular bend based on the feedback they feel in their muscles and joints, as well as balance feedback and their knowledge of the jumps they have performed in the past. The appearance of bearing and lifting weight is difficult to achieve, as the animator must visually match the poses to their knowledge of bodies or to a visual of a similar human to their character performing the motion. Simulation allows a computer to calculate movement based on physical aspects of the character, incorporating knowledge such as the character's weight and body make up, allowing an animator to produce animations of unfamiliar motions with a higher degree of physical plausibility.

1.2 Contributions

This thesis describes a controller which simulates a jumping motion of a character. The generated animation is created to be plausible in appearance, using a simplified physical representation. We sacrifice some physical accuracy in favor of speed and simplicity of creating, describing, and implementing the simulation. Specifically, we contribute a description for the windup, thrust, and in air phases of

a jump and created a controller using a physical simulation and an implementation in Unity3D. In short, our contributions are:

- A controller which simulates a windup and thrust phase of a jump, moving the character from the ground into the air
- A description of character poses based on torque generated by muscles
- A description of character poses based on elastic energy of the muscles
- A sampling-based method for choosing a pose given muscles and a desired output
- Visualizations of the animations and values for analysis and presentation

1.2.1 Unity3D

To help separate the work performed for this thesis from features provided by Unity3D, we give a brief description of what Unity3D provides. Unity3D was chosen due to familiarity, lack of cost, and the selection of features available. While we chose Unity3D, this system could be implemented in other engines, requiring that the engine provides rendering, physics, and scene management capabilities, which are provided by most game engines.

Unity3D is a game engine which we used to develop this system. It provides infrastructure for rendering, scene management, skeletal animation, asset import and management, lighting, and scripting. Our simulation leverages the provided features through the Unity3D scripting interface, which allows a developer to write scripts in C# to use the functions available. We developed scripts for calculation of muscle forces, as well as for applying the force to an imported model. Models were created in AutoDesk Maya and imported using Unity3D's asset import as Unity3D GameObjects with attached Transform components which allow arbitrary transformations. The skeleton, specified in AutoDesk Maya, was imported as a tree with each joint as a separate GameObject node of the tree, with a parent GameObject as the root node, which also was parent to the GameObjects carrying the created mesh.

Unity3D tracks and manages each GameObject, running the update loop for the user. Unity3D uses a component-object model, where GameObjects are given behaviors by having a particular Component. Each Component has several different update functions for different stages of the engine's pipeline such as `FixedUpdate` for fixed spacing of calls instead of actually calling once per frame or `OnPostRender` for operations performed after rendering, but the main function is `Update`, which is called every frame. Components allow rendering and simulation behaviors to be separated from each other, decoupling them from each other and the objects. The update loop handles rendering related updates as well as running `Update` for each component on the object. Our simulation is implemented as several Component scripts attached to the character, with additional Components to carry extra information with each joint of the skeleton and Components on each hip for inverse kinematic solving. The scene manager provided by Unity3D allows for simple access and manipulation of these Components and their objects, managing many useful data structures, namely the tree of all objects present in the scene.

Unity3D also provides matrix and vector math libraries as well as generalized math libraries to aid in physics calculations and subsequent transformations on the objects. There are rigid body and physics components available, but for the purposes of our simulation we performed the physics calculations ourselves to gain greater control. We utilized Unity3D ray casts for detecting collisions between the character's feet and the floor.

For managing the numerous user-specified values, we use the Unity3D Inspector. Unity3D provides the Unity Editor for creating and managing scenes and objects within, which is depicted in Figure 1.4. Within the Editor the Inspector is a window showing the Components attached to the selected GameObject. Unity3D utilizes serialization to allow public class variables to be set and modified through the Inspector, providing a form of fields into which numbers can be entered. Using this, we allow the user to specify the desired constants for the simulation.

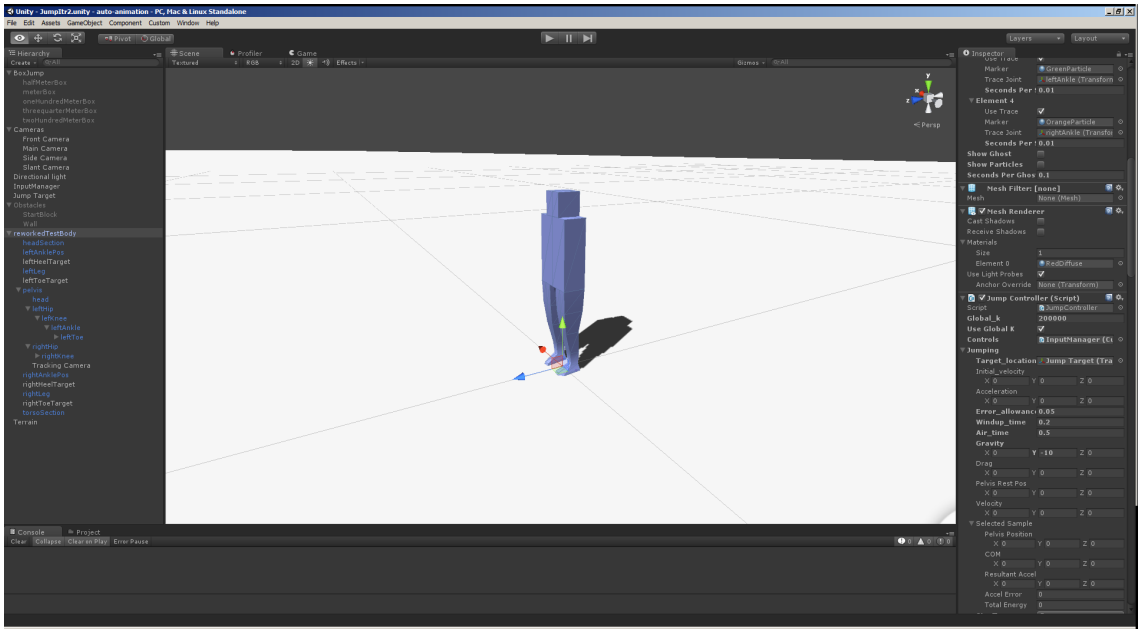


Figure 1.4: Pictured is the Unity3D Editor. At left a representation of the scene called the Hierarchy, allowing for visualization and manipulation of the tree of objects that make up the scene. At right is the Inspector, showing the interface for setting and editing the values specified in our simulation, the Jump-Controller Component. In the center is a visual of the scene, allowing for intuitive placement and manipulation of objects, showing the character. At the bottom is a window showing console output and errors. These windows may be moved by the user.

CHAPTER 2
PREVIOUS WORK

2.1 Background

2.1.1 Muscle-based Simulation

Producing athletic animations for human characters is difficult. One method, motion capture is used for production of realistic animations for human athletics and other motions, however it requires the collection of information for each motion and does not adapt to the virtual environment. Muscle-based approaches produce realistic motions which adapt to the environment, using a complex model of the musculo-skeletal structure.

Geijtenbeek et al. [2] use a rough, user-created muscle routing on a skeleton to produce various gaits that are learned based on the velocity and environment. The muscle routing is optimized to remain within a physical region while providing optimal forces on the skeleton based on freedom of motion of the skeletal joints and the calculated optimal length of the muscle. This model is then used to compute sequences of muscle activations, modeling neural signals, which produce the final animations. This method is effective, producing good results in various levels of gravity on at least 10 different bipedal skeletons.

2.1.2 Rigid Body Simulations

Instead of complex muscle systems, some physical simulations utilize a rigid-body character with a user-defined skeleton to find optimal poses based on desired conditions other than muscle simulation. Ha et al. utilize such a scheme to generate landing motions for human characters based off linear velocity, global angular velocity, and angle of attack. The system chooses either a feet first or hands first landing strategy and moves into a roll to reduce stress on the body using principles from biomechanics and robotics. A sampling method is applied to determine successful conditions, producing bounding planes for the data. The movement is broken into stages of airborne and landing, in which the character re-positions for the designated

landing strategy, and executes the landing strategy respectively. Each of these is separated into impact, roll, and get-up stages. Movement and joint positions are produced using PID servos [1].

Other work on producing such controllers was produced by Faloutsos et al. who described a method of composing such controllers by giving pre-conditions, post-conditions, and intermediate state requirements. The composed controllers are then chosen at each step based on the current pose and which controller is deemed most suitable [3]. Hodgins et al. created several controllers for running, vaulting, and bicycling, creating realistic motions and secondary motion using rigid bodies and spring-mass simulations [4]. Geijtenbeek and Pronost provide a detailed review of physics based simulations [5].

2.1.3 Inverse Kinematics

Inverse kinematics approaches attempt to generate the motion based on a desired final position, determining the skeletal position by solving the system given constraints. Koga et. al use path planning, inverse kinematics, and forward simulation to generate animations of arm motions for robots and humans working cooperatively. They produce arm manipulations that avoid collisions and result in final positions and orientations for specified parts of the arm to produce motions such as a human putting on glasses and a robot arm and human cooperating to flip a chessboard [6].

2.1.4 Commercial Software

Several technologies exist to similarly aid in animation production. Unity3D MecAnim applies constructed animations of various types to similar skeletons, providing joint constraints and muscle definitions in a similar manner to our simulation. MecAnim provides functionality for constraining range of motion and blending between existing animations, utilizing existing clips to produce complex animations in a manner similar to the composable controllers described by Faloutsos et al., as well as inverse kinematics solving. Due to a lack of understanding of the features and limitations, as well as what level of control is available, we chose not to utilize

MecAnim for our current implementation. As discussed in chapter 6, future work would ideally take advantage of MecAnim.

3ds Max footsteps offer a method of positioning feet and producing walk, run, and jump cycles based on number of parameters. Without knowledge of the algorithm, analysis is difficult, but it seems to produce animations by specifying timing and parameters about the stride. Parameters defined include stride width, length, and height. Our simulation seeks to produce more natural looking animations through use of a muscle simulation. The footstep style of animation may be preferable to artists however as it gives very strong control over the timing and spacing of the individual events of the animation, such as foot falls.

CHAPTER 3
SIMULATION METHOD FOR A JUMPING MOTION

In this chapter we describe our method for simulating a character. We begin by describing the motion, breaking it into stages in Section 3.1. In Section 3.2 we then describe the setup of the problem and inputs to our simulation. As part of our description of the inputs to the simulation, we describe the representation of a skeleton in detail in Section 3.2.3 as well as the other user specified constants in Section 3.2.2. Following the problem setup, we describe our inverse kinematics solver in Section 3.4, and balancing of the character in Section 3.3. We then describe two methods of simulation: the first using torque on a joint to describe the motion in Section 3.5 and the second using energy of the muscles to describe the motion in Section 3.6. The stages of jumping are described for each of these calculation paths.

3.1 Overview of a Jumping Motion

Jumping is the acceleration of a character's center of mass upward. Acceleration is applied in excess and in opposition to gravity, resulting in the character breaking contact with the ground and traveling a short distance before contact is re-established. This acceleration results due to the character pushing against the ground, first bending to create space and lengthen muscles, then extending, contracting the muscles.

Jumping motions can be divided into several stages. First is the lead-up or wind-up stage in which the character flexes, preparing their muscles for contraction and providing space for their body to extend. This takes the form of a slight crouch. Next comes the thrust stage, in which the character extends and exerts a force against the ground to accelerate upward. The character pushes against the floor with their feet, the contractions of the muscles causing joints to unbend and as a result displace the character's center of mass causing work to be done. This extension and resulting thrust is caused by the contraction of muscles, which produce torques on the skeleton.

Once the character has broken contact with the ground, they travel through the air, their velocity decreasing steadily due to gravity, air resistance, and other forces until they eventually land. Once they regain contact with the ground, the character absorbs or disperses the kinetic energy of their jump.

We attempted two main approaches to producing a jump motion through muscle simulation. The first method focused heavily on the torques produced by the muscles, but ultimately failed to converge, and consistently moved towards a solution that appears incorrect, where the character begins to lean towards the right. These calculated values were also extremely small as compared to the calculated required values, despite repetitions of the math derivation and careful debugging. The second method focused on the kinetic energy and performed calculations to match the energy of the muscles, which simplifies the movement and produced a successful simulation.

3.2 Model Initialization and Setup

3.2.1 Creation of the Model and Rig

The character in our system consists of a mesh, a skeleton or *rig*, and a controller which has itself several sub-components. First we constructed a mesh, a 3D visual representation of our character. Our mesh is a simple, blocky humanoid, lacking arms in order to focus on the motion of the lower body. A more complex human character, or non-human bipedal characters could be substituted. This mesh consists of vertices, which each have a position as well as other information not relevant to our simulation such as normal and texture coordinate, which are used by Unity3D for displaying the mesh. This mesh can also be referred to as a character model, but in the context of this simulation we will refer to it as the mesh. Three vertices form a triangular face, though these are often created as quadrilaterals by the artist as the topology of the model can be simpler to work with due to the grid patterns formed as compared to triangular meshes. In the case of quad meshes, the mesh is often treated as a triangle mesh by the game engine or renderer, with each quad split into two triangles.

Our mesh was created using AutoDesk Maya, positioning the vertices in groups

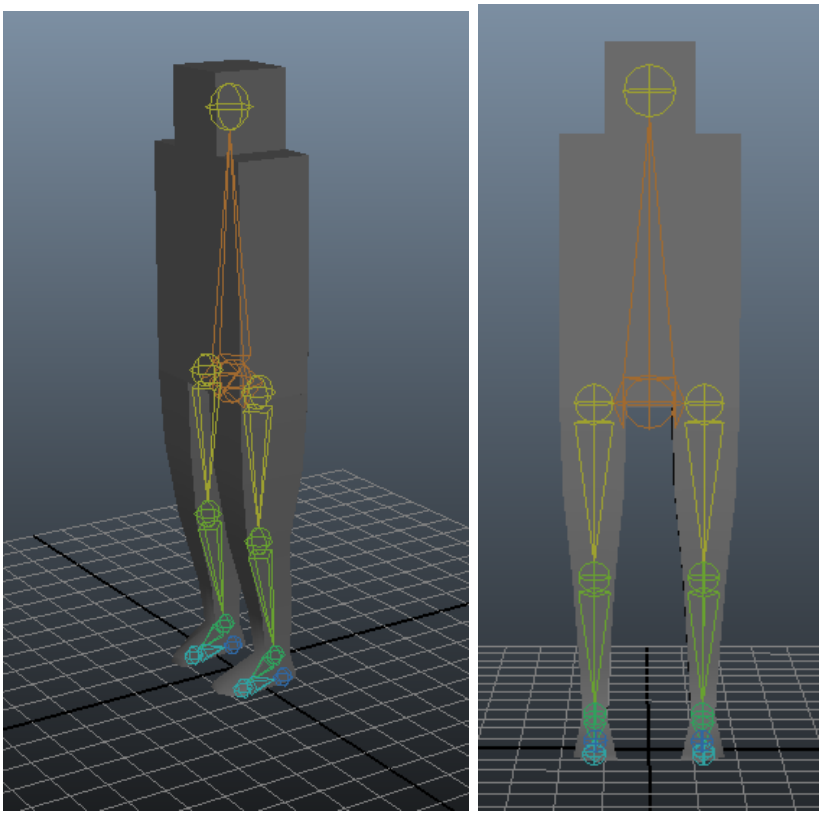


Figure 3.1: Above is an example of a character model in AutoDesk Maya, the character used for our simulation. The character skin or mesh is shown in gray, with a rig shown in multicolor. While it is visualized as a series of spherical joints with connecting solids, the rig itself does not have a visual component in practice. The rig acts as a skeleton, deforming the mesh of the character model to make animation easier. Additional tools such as user-specified skeletal controls and inverse kinematic handles can be used to further simplify the creation of animations for artists. While these tools simplify movement of the model, the artist still must position each joint for each frame of the animation, which is then stored for later playback. This rig contains 11 bones and 12 joints: a pelvis, two hips, two knees, two ankles, two toes, two heels, and one head. In this visual, orange bones and joints are connected directly to the root, the pelvis, yellow are the first level of the tree, green the second, light blue the third, and dark blue the fourth level of the tree.

or individually, using several rectangular prisms as a base. Using the edge loop tool, more vertices were inserted, with loops of edges circling the torso and legs to produce the final shape. This mesh was then rigged, meaning a skeleton was added. As described in further detail in section 3.2.3, joints were positioned individually relative to the mesh, attempting to mimic the positioning of joints in the human skeleton. The connections were made simply, with each joint acting as a ball and socket joint, meaning that constraints needed to be specified at a later stage to facilitate hinge joints such as the knee and ankle and that complex, multi-boned structures as found in the human foot and shin were simplified to a single bone connecting two joints. The hierarchy of joints for the skeleton was rooted at the pelvis with three children: one leading to the upper body and one to each hip. From there a single joint was used for the manipulation of the upper body, with separate joints for each hip, knee, ankle, toe, and heel. Though there is no movement in the toe, placing a bone for the foot requires an end joint for the bone to connect to.

Joints are then associated with the mesh through weight painting, in which each vertex of the mesh is assigned a weight for each joint. This weight designates if and how much a vertex transforms when a joint is moved. Each vertex must be assigned a weight for each joint of the model. Careful weight assignment is highly important as this determines the behavior of the character's "skin" when they move, affecting how the mesh twists or bends as well as which parts move with which bones. The joints may then be used to manipulate the mesh to produce animations, with each key frame in an animation storing information about each joint instead of each vertex.

3.2.2 Environmental and Jump Constants

Within the controller there are a number of constants the user can specify, outside of the character itself. These specify constants for the simulation environment as well as some constants describing the animation to be produced. Our only environmental constant is gravity, which we specify as -10 $\frac{m}{s^2}$, where the negative indicates the downward direction. Other constants include the air time, windup time, error allowance, and constants for the PID controllers which specify a mul-

tiplicative factor for the proportional, integral, and derivative components of each PID controller.

The times indicate how much time the character is expected to spend in the air and winding up for the jump. Air time in our case consists of the portion of the animation where the character's feet are not in contact with the ground plane. Windup time refers to the time in which the character has their feet on the ground and is in the process of accelerating their mass upwards as the initial takeoff portion of the jump. A long windup time gives a very slow, exaggerated jumping motion while a short windup gives a very rapid, clipped motion. While we allow the user to specify any time for both air and windup, in practice there is a limited range of values that are possible for the character. Values outside of the reasonable range produce indeterminate or strange behaviors in the simulation, such as either failing to find a muscle load that can feasibly produce the jump or attempting to produce the jump and failing partway.

Error allowance in our simulation is a widely used value indicating a percent error tolerance. This tolerance is used for determining the allowable difference between the desired values of either resultant linear acceleration and desired acceleration for the torque based simulation, or calculated kinetic energy and total elastic energy for the energy based simulation. The allowable difference is used for choosing samples and determining if the character has satisfied either the energy or acceleration requirements to complete the windup stage. The same error allowance is also used to determine if the limb usage is greater than a minimum to complete the windup stage, which is used to force a degree of bend in extreme cases, where very little bend is required as the muscles are extremely strong or the jump distance is very small. This minimum is applied to compensate for the contraction rates of muscles, which would require a higher degree of bend than our muscle model does, leading to a more realistic simulation. We use an $\epsilon = 0.05$ for calculating our percent error allowance, and a separate, floating point epsilon of 0.001 for comparison of floating point numbers. Below is an example of the inequality using the error allowance.

$$E_{kinetic} - E_{elastic} \leq \epsilon E_{kinetic}$$

Input $\mathbf{t_a}$ (s)	Calculated $\mathbf{a}$ $(\frac{m}{s^2})$	Calculated $\mathbf{v_0}$ (m)
0.1	(0, 2.5, 50)	(0, 0.5, 10)
0.2	(0, 5, 25)	(0, 1, 5)
0.3	(0, 7.5, 16.67)	(0, 1.5, 3.33)
0.4	(0, 10, 12.5)	(0, 2, 2.5)
0.5	(0, 12.5, 10)	(0, 2.5, 2)
0.6	(0, 15, 8.33)	(0, 3, 1.67)
0.7	(0, 17.5, 7.14)	(0, 3.5, 1.43)
0.8	(0, 20, 6.25)	(0, 4, 1.25)
0.9	(0, 22.5, 5.56)	(0, 4.5, 1.11)
1.0	(0, 25, 5)	(0, 5, 1)
1.1	(0, 27.5, 4.55)	(0, 5.5, 0.91)
1.2	(0, 30, 4.17)	(0, 6, 0.83)
1.3	(0, 32.5, 3.85)	(0, 6.5, 0.77)
1.4	(0, 35, 3.57)	(0, 7, 0.71)
1.5	(0, 37.5, 3.33)	(0, 7.5, 0.67)

Table 3.1: Values for calculated necessary velocity given air and windup times for a skeleton with muscle k values around 20000. This table shows calculated necessary acceleration and velocity for the character given constant jump displacement of $x - x_0 = (0, 0, 1)m$, gravity $g = (0, -10, 0)\frac{m}{s^2}$, and windup time $t_w = 0.2s$, where windup time refers to the amount of time the force of the jump is applied to the character. Values are calculated with variable desired air time t_a in range $[0.1, 1.5]s$ with a step of $0.1s$, where v_0 is the velocity when the character leaves the ground, and a is acceleration required to reach v_0 from rest.

Lastly, we utilize proportional-integral-derivative (PID) controllers, which require 3 constants for each controller. A PID controller modifies an input based on error in 3 ways: proportional to the error, proportional to the integral of the error, and proportional to the derivative of the error [7]. This allows changes to the system to have gains directly related to the error through the proportional component, eliminate steady-state error with the integral component, and to adjust gain with the derivative to improve stability of the changes. Our system does not require an integral component, meaning our controllers are just PD controllers with their constant for I set to 0. For our controllers, as we work in discrete frames, our controller

	Windup	Balance
k_p	0.25	1
k_i	0	0
k_d	0.25	1

Table 3.2: Above are the proportional (k_p), integral (k_i), and derivative (k_d) constants for the two PID controllers used in our simulations. One controller handles feedback control of the bend of the character, working with the error between the required output of the muscles and the desired output while the second handles the balance of the character, minimizing the distance of the character's center of mass from the center of the character's supporting polygon through feedback control. These constants act as a weight on the different components of the controller, adjusting the rate of control and the rate of control relative to the error, constant error, and change in error through the proportional, integral, and derivative components respectively.

calculation is as follows.

$$u(t) = k_p e(t) + k_i \sum_{\tau=0}^{t} e(\tau) + k_d \left(e(t) - e(t-1)\right)$$

3.2.3 Skeleton, Joints, and Muscles

For the purposes of animation, a joint is an object with an associated position, associated transformation, a parent joint, and some number of child joints. In the case of the root, the parent joint is absent and in the case of the end joints such as tips of the fingers there are no child joints. Each child joint is connected to the parent by a rigid bone, which protrudes from the parent at a given resting angle. These joints are structured in a tree, as the parent and child joints imply, with the root of the tree at the pelvis. This tree serves as a hierarchy for transformations. Figure 3.1 shows the skeleton for our character.

Joints are associated with a set of vertices from the mesh to be animated. Each of these may be associated with multiple joints, and are assigned a weight for each joint which acts as a scale factor for the transformations performed on the vertex. When a joint is transformed, the transformation is propagated to the children, with the parent as the origin of the child node's coordinate system.

In addition to the animation-related functions, joints in our system handle a number of other functions and values. Each joint keeps track of its constraints on rotation. Rotations are performed axis-angle, that is a rotation is specified as a rotation θ degrees around an arbitrary axis $\mathbf{e} \in \mathbb{R}^3$. This makes the constraints somewhat more complicated as compared to euler angles in exchange for simpler, rotations.

Constraints, however, are specified through euler angles: pitch, roll, and yaw. These correspond to a rotation about the x, z, and y axes respectively in Unity3D's coordinate system. When rotating using a traditional rotation matrix through Euler angles, constraints simply prevent any of the angles from exceeding the bounds. The other issue was how to constrain a joint when rotation about a certain axis was fully prohibited, such as the knee joint which can only rotate about the x-axis. To constrain the axis-angle rotation thus, simply zero the undesirable component of the vector. Constraining the axis-angle rotation to a region with degree minima and maxima for rotations around the x, y, and z axes requires definition and constraint to the region of a sphere the rotation constraints covers. Instead of solving this complex problem, we instead clamp the euler angles to their constrained regions after each rotation, forcing the controller and the inverse kinematics solver to operate within the region and thus reducing the possible solutions for each to only those solutions within the region.

Along with constraints, each joint tracks a weight, which allows distribution of weight over the body. This distribution affected the torque simulation heavily as the torque of each joint resulted in a different angular momentum depending on the mass distribution. The energy simulation considered the character as a rigid body, not considering the changes occurring due to weight distribution except for balance issues and the effects on the muscles as described shortly. Joints also provide functions for calculating the direction to the next joint for muscle calculations, and a utility function for returning to a resting position.

Several joints form a muscle. Though any number of joints is possible, our muscles only utilize 3 joints each. These joints serve as anchor points for the muscles, with each anchor point specified as a number in $[0, 1]$ between the center joint, i.e.

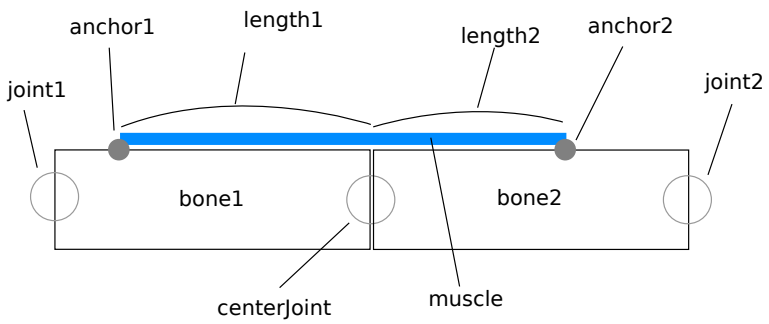


Figure 3.2: Pictured is a diagram of how a muscle is defined. Three joints are used: joint1, joint2, and centerJoint, where centerJoint is the joint the muscle crosses. The center of rotation for the joint is conceptualized as at the edge of the bone as shown above, with the movement modeled after connecting a pair of blocks with a rubber band, though the joint is positioned at the center of the bone. As the bone used to calculate the muscle forces is only loosely related to the bone of the character's rig, this does not introduce errors as the model is kept internally consistent with only the angle and relative positions of the joints used for the calculation.

the joint the muscle crosses, and the other joint the bone connects to. In this way three joints describe the path of a muscle in our system. The muscle itself is a linear spring, obeying Hooke's law. This gives force (F) and elastic energy ($E_{elastic}$) as

$$F = -ks$$
$$E = \frac{1}{2}ks^2$$

where k is the spring constant for the muscle and s is the displacement of the spring. The negation of k in the force calculation indicates the force restores the spring to rest. These anchor points and the method of muscle specification are shown in Figure 3.2. While the muscle in Figure 3.2 is at rest when the joint is straightened, the rest angle can be specified arbitrarily to handle, for example, the right angles of the muscles crossing the ankles and hips, as the hip muscles span from pelvis to hip to knee.

Intuitively, this models the bones as blocks, with a spring representing a combination of the muscle and tendons connecting the blocks. When the joint rotates,

the spring necessarily stretches, producing a restoring force attempting to retract the spring to its initial length. This then produces a torque on the blocks involved, causing a torque on the blocks at the joint. As one block is more strongly anchored than the other, i.e. the part of the joint chain closer to the foot and thus the ground is less able to move freely as more of the character's weight is resting on it, one bone rotates about the joint center. The user is free to then restrict this motion further by specifying ranges of motion for the joint to mimic human capabilities.

Considering a muscle as a spring models a muscle at maximum activation. A relaxed muscle has very little contractual or restoring force, and thus a low value for k, while a flexed muscle has a large k. While the muscle model could take muscle activation level into account, we chose the simpler system, allowing the user to specify k for their particular case. This gives more control for the animation, but also allows for more errors and issues. The results with poorly chosen k values resemble the results with poorly chosen times.

Along with the s values, a muscle keeps track of its center joint, anchor joints, and the bone width. A muscle has two anchor points and crosses the center joint. The positions of the anchors are specified as values in the range $[0, 1]$ along the bone between the center joint and the anchors. Bones are considered to be rectangular prisms with a square cross-section, with a width r specified by the user. Spring displacement can then be calculated directly from the angle of the joint using these constants and our assumptions as illustrated in Figure 1.3 with the equation below derived from the law of sines:

$$s = \frac{r \sin(\pi - \theta)}{\sin \frac{\theta}{2}}$$

We derive this equation by calculating the supplementary angle, opposite the spring displacement. This angle can therefore be expressed as $\pi - \theta$. The sum of the remaining angles of the triangle formed by the spring displacement and bone ends can then be expressed as $\frac{\theta}{2}$. These angles are shown in Figure 3.3. We assume uniform bone width, which allows a simpler calculation of the angles.

For debugging purposes and for calculating the extension of the limb we have

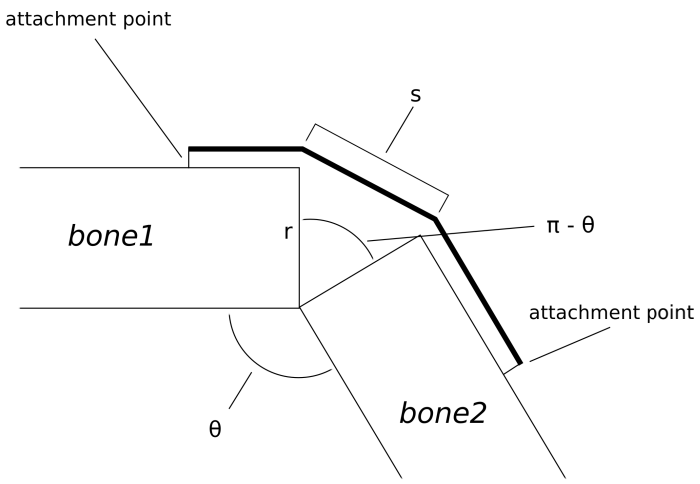


Figure 3.3: Above is a visual representation of the angles used to calculate the spring displacement. The angle of the joint, θ, is used to calculate the displacement s using the bone width r and the law of sines. Angles for the joint are restricted to the range $(0, \pi)$, with values outside of this range producing an undefined s. As a bend in the reverse direction, considering rotation about each axis separately, produces an equivalent situation, the absolute value of the angle is used and the equivalent angle within the range is found. For negative angles, an adjustment is made in the direction to account for the difference in the direction of the forces produced after calculating the displacement, reversing the sign on the resulting torque or altering the direction of acceleration for the energy calculation.

each muscle calculate its utilization, giving a value in $[0, 1]$. Utilization is a number used to determine if a muscle is being stretched and producing a force, indicating if the muscle is too weak or is simply not being used by the simulation. This utilization is calculated as the dot product of the normalized vectors between the center joint and the anchor points. The value is then scaled from $[-1, 1]$ to $[0, 1]$ by adding 1 and dividing by 2. A modification to this function is required for joints at rest at angles other than 0 or π radians, instead using a comparison to the stored rest angle of the joint.

As an additional debugging tool and method of determining the expected ranges of values, we calculated the values in Table 3.3. By testing different ranges

of values and comparing with some expected values, we determined empirically a range of values that should produce "normal" human simulation, i.e. expected strength values required for several different heights given body mass and proportions. An military study of male aviators in 1988 provided information on mass, mass distribution, and the dimensions of different body sections. Range of motion data was obtained from Boone and Azen who compared clinical measurements with estimations in the handbook of The American Academy of Orthopaedic Surgeons as a reference [8].

3.3 Center of Mass and Maintaining Balance

The center of mass (CoM) is calculated as the centroid of the character. More specifically, joint positions are averaged, with a weight assigned to each joint based on the weight of the limb associated. The CoM must be recalculated with each update to the character's pose as the shift in weight changes the position.

Using the calculated CoM, the balance of the character can be determined by the position of the CoM relative to the supporting polygon of the character. The supporting polygon is a polygon determined by the points of contact of the character with the ground or other supports which provide a normal force to counteract gravity and other external forces. During the windup and thrust phases, the character maintains contact with the ground through their feet, with the outer edges of the feet forming two sides of a quad, a line between the two feet at the toes forming a third, and a line between the heels of the character forming the fourth side. This polygon should be parallel to the ground plane, and is positioned at the bottom of the feet. If the character's center of mass is over this supporting polygon, the character is balanced.

To quantify balance, the vector between the center of the supporting polygon and the position of the CoM is measured. This vector is then projected into the same plane as the supporting polygon, giving a 2 dimensional error between where the CoM is currently and where it would need to be to be perfectly centered. The PD controller then attempts to minimize the magnitude of this vector by moving in the prescribed direction while bending to achieve the desired force, constraining

the number of solutions possible to provide the desired force.

As the character bends for the windup and extends for thrusting, the character must rebalance with minimal change in flexion, i.e. height of the hips, in order to maintain the calculated load on the muscles. For these cases, the upper body is used to rebalance. The character bends forwards, backwards, left, and right to shift the weight of the upper body to offset the balance error created by the lower body pose. This is performed using a PD controller, with the error input as the balance error as calculated above, and the output as the amount to rotate in each direction subject to skeletal constraints.

3.4 Inverse Kinematic Solving for Ankle and Knee Position

As the skeleton is a hierarchy assumed to be rooted at the hip, a problem arises with applying rotations to joints. To keep a character's feet rooted to the floor as is expected, positions must be solved for using inverse kinematics. Given the desired position for the hip, and the desired position of the foot, we solve for the joint angles and positions of the knee and ankle. Constraints are placed on each joint, limiting the range of motion to an expected range as well as limiting the axes about which each joint can rotate, preventing unnatural directions of motion. These values are specified per joint and can be edited by the user to simulate varied levels of flexibility or alternate body shapes.

A solution to the joint positions is found greedily using these constraints, and gradient descent method which works on single-chains of joints. A single chain of joints is a sequence of joints in which each joint has a single child and a single parent, with one root joint and one leaf joint which lack a parent and child respectively. This hierarchy may be specified as a subtree of the skeleton in which these conditions hold true, as in our simulation where the joints from the hip to the heel are considered separately for each leg. Given the hierarchy of joints and a desired position for one of the non-root nodes of the chain, cyclic-coordinate descent (CCD) is used to determine rotations of the joints between the joint in question and the root that will minimize the distance between the joint in question and the desired position as done by Lander [9, 10]. This algorithm is shown in Algorithm 3.1, with a visual

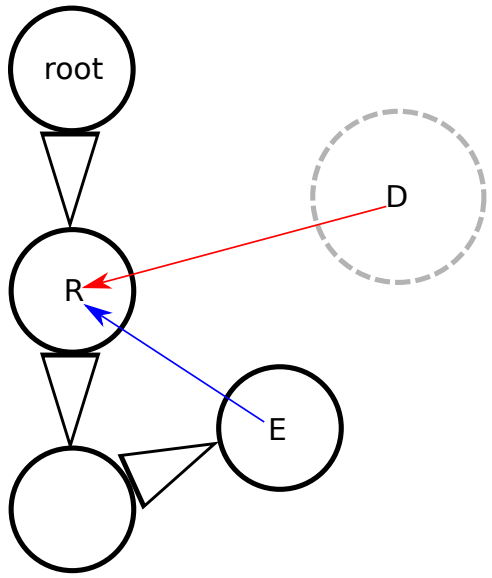


Figure 3.4: Depicted is a chain of joints, with a root joint, an end joint E, and two joints in the middle of the chain. To move the end joint E, which may for example be a hand or foot, to position D, iterate over the chain of joints and calculate the vectors between the current and end joint positions as well as the vector between the current joint position and the destination for the end joint. The dot product of the normalized vectors then gives the cosine of the angle between these vectors. Rotating R by this angle will then align the vectors, moving E towards position D. The dotted gray circle shows an example target area in which the joint E is considered "close enough" to position D, allowing the algorithm to terminate if the center of E is anywhere within that circle. If D is farther away than the total length of the joint chain, the algorithm is set to terminate after a user-specified number of iterations, in our case 30.

representation of a calculation for a single joint in the chain shown in Figure 3.4.

This approach is simpler to implement than other approaches such as the pseudo-inverse of the Jacobian for the specific case of single chains of joints. In addition, this approach allows some flexibility, specifically in constraints of the joints. As each joint is addressed individually instead of the system as a whole, any constraints placed on the joint can easily be accounted for by simply preventing the joint from rotating out of the desired range while the rest of the system continues to move as close as possible to the solution. One downside is that a halting condition

Algorithm 3.1: Given chain of joints C, move joint E to position D using cyclic coordinate descent. This process iteratively moves joint E closer to the location D, concentrating on each joint R in the chain one at a time and solving the geometric problem of minimizing distance between E and D by rotating R. D is the desired position of the body part, such as where the toe should be placed and E is the joint that should be moved to the desired position. The vectors RD and RE are the vectors between the positions of joints R and E, and joint R and the desired location D in $\mathbb{R}^3$.

```
function SingleChainIK(C, D, E)
    repeat
        for all Joints R between E and the root, starting with the end E do
            θ = cos⁻¹ (RD · RE)
            Rotate R by θ
        end for
    until Desired number of iterations performed or E is close enough to D
end function
```

must be determined, through a minimum acceptable distance. Additionally, the algorithm does not halt if D is farther from the root position of the chain than the length of the chain, i.e. it is farther away than the length of the limb. To handle the case where the joint cannot be moved within this minimum distance, a maximum number of iterations must be designated. In practice, 100 iterations is enough to converge, with as few as 30 working well for our simulation.

3.5 Torque-Based Simulation

One method of simulation we attempted used springs placed along the length of each limb to produce a torque on the joints of the character. Torques on the joints result from the force of the muscle pulling a bone to rotate about the joint, resulting in a complex system of motion with each bone rotating around the joints. These rotations combine to move the body in a direction, allowing a character's control to be centered around degree and timing of muscle activation.

Our method for this type of simulation was inspired by the other work in complex muscle based simulation for bipeds such as in Geijtenbeek et al. [2]. In our case, we take the muscle as constantly activated, which means that its spring constant defines the strength of the muscle, allowing the user to set the activation manually. This muscle is then stretched to produce the desired torque by bending

the affected limb. The restoring force of the spring-muscles produce torques which are used to calculate angular momentum, and from that linear momentum. Linear momentum is then used to calculate the resulting acceleration.

3.5.1 Calculation of Required Velocity and Acceleration Given Time Constraints

Before calculations relating to the model's skeleton are performed, an initial estimate of the jump path is performed. The estimate uses a forward kinematic calculation to determine the velocity required to move an object through the air from the initial position of the model to a final position. This path is specified indirectly by the user by setting time in air, gravity, and desired displacement in three dimensional space, which necessarily describe a parabolic path assuming that displacement and gravity are not 0, and that gravity is an attractive force pulling the character toward the ground. While perhaps somewhat less intuitive than positioning the character exactly how the artist wants it for each frame, this allows a wide range of physically plausible, simple jumps to be produced. In such cases, this reduces work excepting special case jumps, such as those found in Warner Brothers' Looney Toons.

The user specifies a desired time (t_{air}) that indicates the time the character will spend airborne during the animation, i.e. the time between when the character's feet break contact with the ground and when they regain contact with the ground. Changes to this time affect the animation length as well as modify the peak height of the jump, as a longer time will necessarily require the character to be airborne longer, and thus be higher in the air. While this reduces the direct control the artist has, this provides a simple way to control animation length.

The initial velocity is given by a manipulation of a simple kinematic equation

$$\mathbf{x} = \mathbf{x_0} + \mathbf{v_0} t_{air} + \frac{1}{2}\mathbf{a} t_{air}^2$$

derived from the relationships between acceleration ($\mathbf{a}$), velocity ($\mathbf{v}$), and displace-

ment $(\mathbf{x} - \mathbf{x_0})$, namely

$$\mathbf{v} = \frac{d\mathbf{x}}{dt}$$
$$\mathbf{a} = \frac{d\mathbf{v}}{dt}$$

.

This relationship gives us the velocity, given a known start position $(\mathbf{x_0})$, destination $(\mathbf{x})$, acceleration $(\mathbf{a})$, and time (t_{air}). As this describes the path of the character once it breaks contact with the ground, the time referred to only covers the time between the character breaking contact with the ground and when their feet touch again at the end of the jump The only force, and therefore acceleration acting upon the character while in the air is gravity, thus giving

$$\mathbf{v_0} = \frac{\mathbf{x} - \mathbf{x_0}}{t_{air}} - \frac{\mathbf{g}t}{2}$$

to describe the initial velocity our character has upon breaking contact with the ground, which then decays over the course of the jump due to gravity to produce a parabolic path. The equation considers the character as a point mass traveling in a vacuum, meaning there is no consideration of friction, drag, or rotational movements. Acceleration can then be determined as

$$\begin{aligned} a &= \frac{dv}{dt} \\ &= \frac{\mathbf{v_f} - \mathbf{v_i}}{t_{windup}} \\ &= \frac{\mathbf{v_0} - v_i}{t_{windup}} \end{aligned}$$

where $\mathbf{v_i}$ is the initial velocity of the character before the jump calculations began. This accommodates jumps where the character is already moving.

3.5.2 Windup Animation of the Character Based on Required Acceleration

Using the calculated velocity and acceleration, we compare the desired values to achieve a particular path with the values produced by the muscles on the skeleton. To compute a resultant instant linear acceleration, we start from the muscle and calculate the torque resulting from the muscle's contraction. First we calculate the scalar force of the muscle-spring as $F = -ks$ where s is the spring displacement as calculated in section 3.2.3. Using the scalar force, we calculate the torque and the angular momentum. Torque magnitude can be calculated as

$$\tau = rF$$

where r in this case is the scalar distance between the pivot point and the location of force application. In our case, this is the distance from the center of the joint to the major anchor point, which we consider to be the anchor point highest in the hierarchy, which is the bone most expected to move. The direction of the torque is the normal vector to the plane of rotation, calculated as the cross product between the directions to each anchor point which gives the plane of rotation about the joint.

Angular momentum is derived from torque, as

$$\tau = \frac{d\mathbf{L}}{dt}$$

where τ is torque and $\mathbf{L}$ is the angular momentum. The angular momentum can then be calculated as τt_{windup}. Angular momentum is used to calculate linear momentum by finding the tangent to the circle at the moment of takeoff. The magnitude of the linear momentum is then calculated as

$$p = \frac{L}{r}$$

which then gives the acceleration as follows

$$\mathbf{a} = \frac{1}{m}\left(\frac{d\mathbf{p}}{dt}\right)$$

where m is the mass of the character.

3.5.2.1 Sampling of Torque Values at Uniformly Distributed Pelvis Positions

In order to determine the position of the character that produces this acceleration, we sample uniformly over a region in which the character is balanced, excluding some points where the character may be balanced in favor of a simpler region. A sample is described by the position of the pelvis joint, the resulting linear momentum, and a vector describing the displacement of the character's center of mass from the center of its support to indicate balance as described above. Samples were restricted to a box defined by a rectangle around the character's feet with a height reaching the position of the character's pelvis when standing at rest. Any sample outside of this volume was assumed to put the character off balance due to the lack of upper body definition in our simulation.

Samples were then taken at uniformly distributed positions in this region. The calculated acceleration was projected onto the desired acceleration vector through the dot product, with values greater than or equal to the desired acceleration's magnitude indicating a plausible solution. These plausible solutions are then ordered by their balance error, calculated as the difference in position between the center of mass and the center of the supporting polygon. The first result of this list is then chosen as the candidate answer and the pelvis is moved towards this sample. At each iteration this is repeated until the desired acceleration is achieved. A plot of samples for a particular set of values, in this case all muscle constants set to $k = 20000$, is shown in Figure 3.5. An equation for the described calculation is shown below.

$$E_{accel} = \mathbf{a_{desired}}^2 - \mathbf{a_{desired}} \cdot \mathbf{a_{calc}} \leq \epsilon \left(\mathbf{a_{desired}}^2\right)$$

Each iteration the error for the skeleton is calculated by comparing the current state of the skeleton to the desired acceleration and a sample is selected. If the error is above the tolerance, a new position for the hip is calculated using proportional-derivative control, where the new position for the iteration of the controller, $u(i)$, is

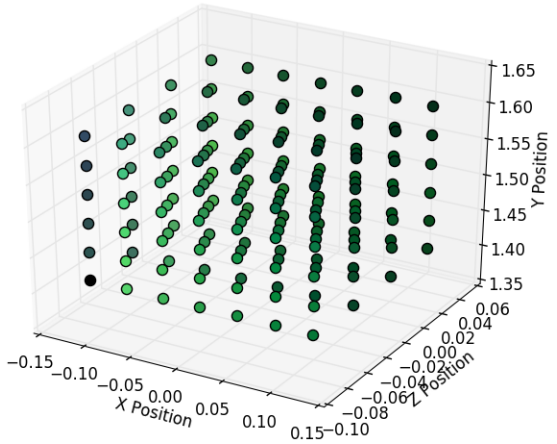


Figure 3.5: A plot of instant linear acceleration samples at different pelvis positions within the balanced region. The axes shown correspond to the axes of the world, with x, y, and z of the plot corresponding to the x, y, and z position in space. Color indicates the magnitude, with the red, green, and blue channels corresponding to the x, y, and z components respectively of the acceleration sample. Lighter color indicates a larger magnitude. This is a clear example of the failures of this simulation method, as the samples seem to continuously increase from the one corner of the bounding box to the opposite diagonally, which does not produce a plausible animation as the character heavily favors one side, even with symmetric strengths.

calculated as

$$u(i) = k_p E_{all}(i-1) + k_d(E_{all}(i-1) - E_{all}(i-2))$$

where i is the iteration, and k_p and k_d are weights which determine the rate of change. The input to this PD controller is calculated by selecting a sample from the set of samples based on the acceleration error of the sample. After filtering all plausible candidates from the samples, the candidate which minimizes balance error is selected and the vector between the current pelvis position and the sample pelvis position is given to the PD controller.

The hip is repositioned based on the output of the PD controller, and the inverse kinematics component then iterates to calculate the positions of the remaining leg joints, assuming the feet should remain in the same position on the ground and the pelvis should remain at the chosen position. This chooses a solution for the intervening joints. At the next iteration, these new joint positions and angles are then used to compute the instant linear acceleration, center of mass, balance error, and acceleration error which are then passed back to the PD controller until the error in acceleration is below the tolerance.

3.5.3 Extension of the Character's Body and Takeoff from Ground

Upward acceleration is animated by calculating the angular accelerations of the joints given the forces acting upon them and the resulting torques. Torque is the change in angular momentum over time, allowing for the acceleration to be calculated as the mass can be assumed to be constant:

$$\tau = \frac{dL}{dt} = m\frac{dv_\theta}{dt} = ma_\theta$$

where τ is the torque of a joint, L is the angular momentum, m is the mass of the limb, which must take into account the mass of the rest of the body which is also moved by the limb, v_θ is the angular velocity and a_θ is the angular acceleration. This results in the below equation for determining angular acceleration.

$$a_\theta = \frac{\tau}{m}$$

Using angular acceleration, the intermediary poses for the model can be determined at each time step. For each frame, an explicit Euler integration is performed to determine first angular velocity and finally angle. As the character continues to extend, a check is made for if the character has yet reached full extension by calculating muscle utilization as described in Section 3.2.3. At full extension, when the limbs are no longer in use, the character can no longer accelerate in the direction of the jump, and the character breaks contact with the ground to enter the in-air phase.

As the windup phase of the torque based simulation fails, it is difficult to quantify or qualify the success and failure of this portion of the simulation. For values where the simulation could not complete the windup phase, the in-air portion is unavailable and for values where the windup phase completes, the values are erratic and fail to produce valid poses, often resulting in extreme contortions of the model.

3.6 Energy-Based Simulation

Another way we simulated jumping motions was energy based. We assumed that the kinetic energy of the character traveling at a calculated velocity from its start position to the destination position is equivalent to the summed elastic potential energies of the leg muscles. The direction of the velocity is determined by the direction the center of mass is accelerated after the windup phase, and we assume that this is achievable given that the character can achieve the desired energy.

Change in direction is accomplished through usage of the feet and shift of weight during the acceleration and windup phases, applied through balance and inverse kinematic calculations. If a character wishes to move forward, they shift their weight back farther during windup, allowing them to accelerate forward farther before becoming airborne. In the same way, shifting to one side can allow the character to accelerate in the opposite direction for non-forward jumps. As with the torque-based simulation, the jump follows the stages of path estimation, windup, thrust, in-air, and landing.

3.6.1 Path Estimation and Windup

Kinetic energy is calculated as

$$E_k = \frac{1}{2}mv^2$$

where m is the mass of the character and v is the velocity. Mass is a constant specified by the user, and we calculate the desired velocity and acceleration as in the torque based path estimation as described in section 3.5.1.

Like with the torque-based method, we took samples in the region where the character maintains balance as in section 3.5.2.1. Samples were collected at regular intervals within a bounding box defined by the character's supporting polygon, the ground, and the the height of the character's pelvis at full extension. The pelvis was repositioned and the resulting elastic potential energy measured. Candidate samples were selected by finding all samples with

$$E_{kinetic} - E_{elastic} \leq \epsilon(E_kinetic)$$

and the candidate with the lowest incurred balance error was selected.

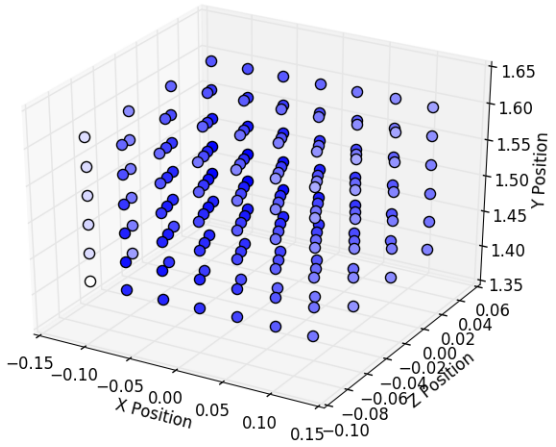


Figure 3.6: A graph of samples for the character with all muscle k values set to 20000. The pelvis positions are plotted, with the color of the data point indicating the energy. A darker point indicates a higher total elastic energy calculated for the skeleton when the pelvis is positioned at that point.

Our sampling method does not guarantee an optimal solution, but can be seen as guaranteeing an approximate minimum. We choose a sample with minimum balance error and iteratively approach it, which greedily finds the nearest sample to the rest pose which satisfies the energy equivalence, $E_{kinetic} = \sum E_{elastic}$. While not guaranteed to be the minimum, this guarantees that we find a near-minimum value assuming the region is continuous. Figure 3.6 is a graph of samples for a particular

run with all muscle k values set to 20000. Color indicates the calculated energy for the sample point, with higher saturation of blue indicating a higher energy value for the sample. The higher energy samples tend to be clustered towards the center of the sample group, which meets expectations as most humans will bend to a position such that their hip is between its normal position when standing and the level of their knees.

Table 3.3 shows examples of estimated values produced by the muscle crossing the knee joint given several different k values over a range of joint angles. By comparing the summed calculated energy output with the desired, we check if the current pose produces enough elastic energy for the intended motion.

Inputs				**Outputs**		
$\mathbf{k}(\frac{kg}{s^2})$	$\mathbf{r}(m)$	$\theta_{\mathbf{deg}}$	$\theta_{\mathbf{rad}}$	$\mathbf{s}(m)$	$\mathbf{F}(N)$	$\mathbf{E}(J)$
200000	0.05	170	2.96	0.0087	1743.110	7.59
200000	0.05	160	2.79	0.0173	3472.960	30.15
200000	0.05	150	2.61	0.0258	5176.380	66.98
200000	0.05	140	2.44	0.0342	6840.400	116.97
200000	0.05	130	2.26	0.0422	8452.360	178.60
200000	0.05	120	2.09	0.0500	10000.000	250.00
200000	0.05	110	1.91	0.0573	11471.500	328.98
200000	0.05	100	1.74	0.0642	12855.700	413.17
200000	0.05	90	1.57	0.0707	14142.100	500.00
20000	0.05	170	2.96	0.0087	174.311	0.75
20000	0.05	160	2.79	0.0173	347.296	3.01
20000	0.05	150	2.61	0.0258	517.638	6.69
20000	0.05	140	2.44	0.0342	684.040	11.69
20000	0.05	130	2.26	0.0422	845.236	17.86
20000	0.05	120	2.09	0.0500	1000.000	25.00
20000	0.05	110	1.91	0.0573	1147.150	32.89
20000	0.05	100	1.74	0.0642	1285.570	41.31
20000	0.05	90	1.57	0.0707	1414.210	50.00
24000	0.05	90	1.57	0.0707	1697.050	60.00
16000	0.05	90	1.57	0.0707	1131.370	40.00
40000	0.05	90	1.57	0.0707	2828.420	100.00

Table 3.3: Above is a table of different displacement (s), force (F), and energy (E) values calculated given a constant bone width of $0.05m$, varying k and varying angle of the joint. These values are for a single knee, with one muscle crossing the joint.

3.6.2 Solutions to the Energy Assignment Problem

The setup of the problem resembles a quadratic programming problem. In this problem, we minimize:

$$\mathbf{x}^T\mathbf{Q}\mathbf{x}$$

subject to

$$\mathbf{x}^T\mathbf{P}\mathbf{x} \leq r$$

where $x \in R^6$ and $\mathbf{Q}$ and $\mathbf{P}$ are square, 6×6 matrices. $\mathbf{Q}$ for our problem is a diagonal matrix, where the diagonals contain the spring constants $\{k_1, k_2, ..., k_6\}$ and $\mathbf{P}$ is $-\mathbf{Q}$. We can introduce additional constraints in the form of maximum and minimum values for all $x_i \in \mathbf{x}$, where the minimum and maximum are calculated at the minimum and maximum rotations of the joint, producing the extremes in spring displacement.

This problem of quadratically constrained quadratic programming is NP-hard and as such we sought a simpler problem or approximation to solve to obtain a solution. As with the torque method, we choose a sampling-based approach as there is a known limit on the possible solutions in our case: balance.

3.6.3 Thrust, In-Air, and Landing

Thrust for the energy based simulation works off of explicit euler integration. The calculated acceleration from the path estimation stage is used to accelerate the character over the duration of t_{windup}. At each time step, the pelvis is translated by $v_{takeoff}$, which begins at 0 and gains $\mathbf{a}dt$ per frame. Each frame the character's extension is checked using the function described in section 3.2.3, and when it falls below a tolerance of 0.1, the character is considered fully extended and the in air phase begins. During the thrust, the movement can sometimes proceed faster than the IK solver can converge, so we pause the simulation briefly and only iterate the solver until it converges.

The in air phase proceeds similarly, with the character translated by the calculated velocity from the path estimate each frame. Velocity is modified by $\mathbf{g}dt$ where $\mathbf{g}$ is the acceleration due to gravity, resulting in the desired parabolic path

of the player through the air. Ray casts are performed to check if the character's feet have contacted a surface to land on. A ray cast is used instead of the collision detection of a capsule collider as is normally used for physics interactions with characters in Unity3Das it provides more information about the distance from surfaces. This allows the prevention of intersection errors where the rougher shape of the collider, which is usually a box, sphere or capsule, causes the character to fail to collide properly and intersects a shape. Performing per-triangle collision would also solve this issue, but as characters are expected to contain tens of thousands of triangles. These ray casts are made from above toe, heel, and middle of the foot, starting above and passing through the points to produce a longer ray for more stable results. Once one of these ray casts indicates the feet have touched such a surface, the character enters the landing phase.

A landing controller is beyond the scope of this thesis, and thus we end the simulation as the character's feet touch a suitable surface. As described in chapter 2, other controllers can be composed with this controller to produce a full animation, handling the complexities of the landing motion as a separate, modular controller.

3.7 Summary

The character is modeled as a mesh with a skeleton and muscles. The skeleton was created as a tree of joints with the root at the hip. Muscles were placed along the limbs, crossing a single joint and anchored to the bones on either side. Force and energy outputs of the muscles were calculated as linear springs, with spring displacements calculated from the angle of the joint. An inverse kinematic solver was used to aid in creating poses.

Two methods were used for the simulation, one using torque based calculations and the other using energy based calculations. Each method proceeded through the stages of path estimate, windup, thrust, in air, and landing, with similar calculations used to produce an error function to quantify character poses. In the torque based simulation, error was calculated by using the muscles to calculate torques, and from the torques accelerations. In the energy based simulation, error was calculated as the difference between elastic potential energy and target kinetic energy.

Samples were collected using these error functions, with the selected best sample minimizing both the balance error and either the acceleration or energy errors for the torque and energy simulations respectively. A PD controller iteratively moved the character's pelvis position towards the selected sample until the error between the calculated values for the frame and the desired values were below a tolerance. After the character moved its pelvis to the chosen sample position to reduce the error below tolerance, it performed the thrust and in air portions of the jump treated as a rigid body, with the simulation ending when the character's feet touch the ground.

CHAPTER 4
VISUALIZATION

In this chapter we motivate the need for good visualizations, as well as the difficulty of visualizing character animation in print format as well as video. We then discuss techniques we utilized to visualize our data for presentation as well as for debugging and analysis.

4.1 Motivation and Inspiration

Showing a motion in a static medium such as print presents numerous challenges. The static image or images must convey a sense of time that is understandable, such that a viewer may intuit the direction and rate of movement. Especially for a complex object such as a figure, occlusion can obstruct information, and prevent understanding of the motion of hidden portions of the body. The projection of a 3D scene can also create similar issues as occlusion, creating ambiguity in depth and obscuring motions in some dimensions.

We drew from several sources for inspiration on how to visualize our results. A video by KORB created for the CCTV Documentary Channel shows "motion sculptures," in which the people in the scene leave trails of material as they move. These sculptures very cleverly captured the movement of the body throughout the space of the video, creating aesthetically pleasing, if somewhat difficult to parse, visuals.

Another film of a similar nature is Choros by Michael Langan and Terah Maher. Images of a single dancer follow her through her her movements, leaving a traceable pattern of movement. This technique was inspired by chronophotography, a precursor to video which utilizes multiple successive photographs or multiple exposures on the same film to visualize movement of a figure or object. These can be laid out in animation strips or superimposed to create a single image.

4.2 Motion Visualization

Markers were used to highlight motion of particular parts of the body, such as the pelvis or center of mass. Other indicators placed on or around the figure can indicate other values, such as arrows to represent vectors of force. This however can result in clutter within the images, scene, or frame of video, occluding or distraction from the primary animation. Copies of the character could be left behind to produce an after image effect such as in the videos discussed in section 4.1. We used these techniques in conjunction with each other, enabling and disabling them as the situation required.

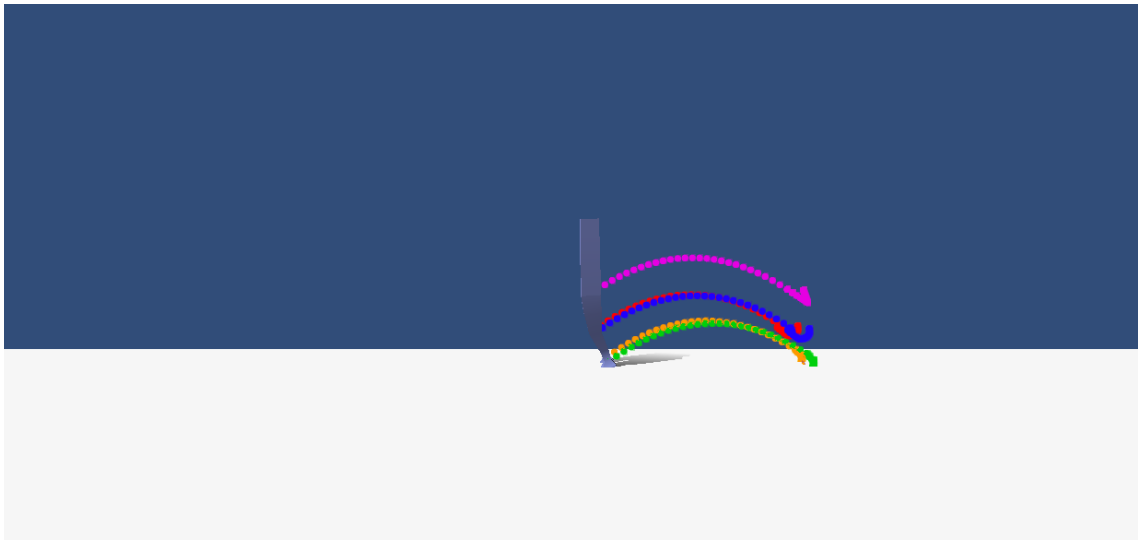


Figure 4.1: Above is an example of the markers tracking joints. With a high sample rate, a curve forms trailing along the path of the joint. Markers were placed at a rate of 10 per second of simulation time.

Markers were implemented as camera-facing "billboard" planes with a texture that were spawned with a user-specified frequency, following an arbitrary joint of the character. Each frame the orientation of the planes adjusts such that it faces the main active camera, allowing very little geometry to be used to produce an always visible visual to be placed in the scene. We used one of these for each joint in the legs as well as one for the pelvis in order to track the paths of the joints. This visualization was an interesting one to view, and gave a similar effect to motion sculptures, but was very difficult to understand without a reference to know which

color of marker corresponded to which joint. Even with knowledge of which joint followed which trail of markers, the visualization was difficult to understand, though it gave a good overview of the whole motion. These marker trails are shown in Figure 4.1.

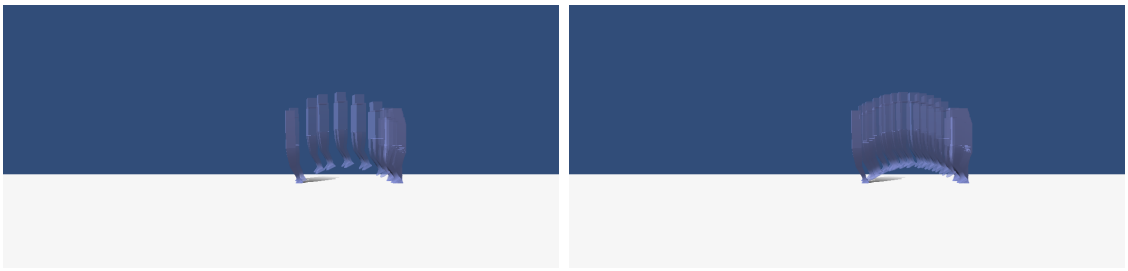


Figure 4.2: Pictured above are examples of a ghost image visualization achieved by placing copies of the character model at a rate of 1 per 0.5s (pictured left) and 1 per 0.2s (pictured right).

We used a ghost image visualization as well in 2 ways: leaving copies of the character behind at a user defined rate and layering collected frame data. In the first method, we make a copy of only the model and necessary skeleton components, leaving out the extra data such as mass, constraints, and muscles used in our simulation, and match the positions and rotations of each of its joints to the character at the current time. The ghosts use a semi-transparent material to help differentiate between them and the model, as well as to provide some clarity as to each ghost's pose and combat the issues of occlusion. Examples of this visualization are shown in Figure 4.2.

An alternate method of forming the ghost image visualization was to layer numerous collected frames in an image editing program. This method was very work intensive, requiring each layer to be matte painted by hand. Algorithms and techniques in image processing and computer vision for automating this process, but for our purposes it was necessary to manually select the desired region in each image that should be visible in the final combined image. To reduce the negative effects of occlusion, layers above the first were given an opacity of 75%. Though work intensive, this produced an excellent visual for still media to show the path of the entire motion, but was a poor visual in many cases for showing detail of the movement as images often occlude each other. Examples of this visualization are

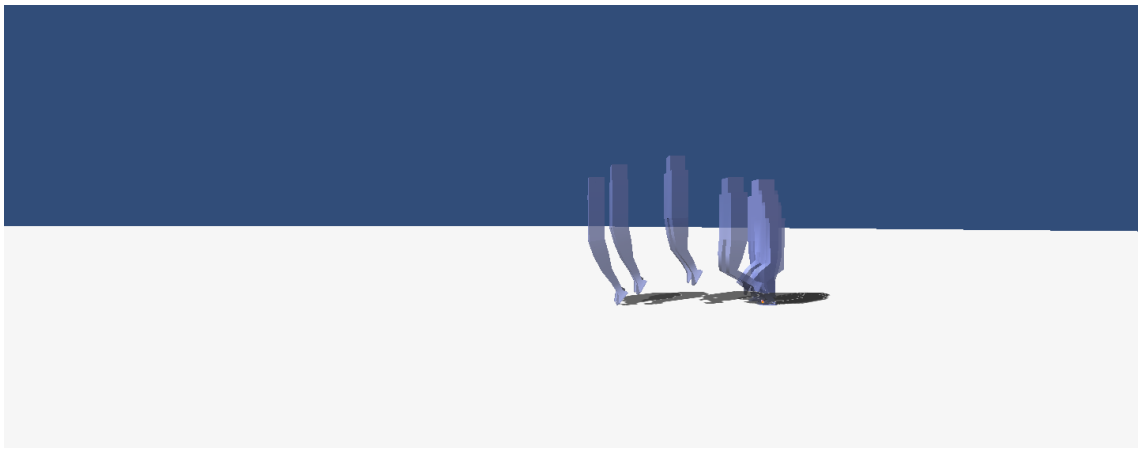


Figure 4.3: An example of the composited frames visualization, in which several frames of the animation are layered on top of each other, and matte painted to produce a combined image with all frames superimposed.

shown in Figure 4.3 and in Figure 5.1. As a compliment to this visual, we used animation strips, in which the frames were simply presented adjacent to each other in order from left to right. These strips are also found in the data tables presented in Section 5.1 and in Figure 5.1.

For debugging visuals, we used a feature in Unity3D called Gizmos, which allowed debug drawing of primitive shapes. We used small spheres of various colors to show sample positions, target positions for the IK solver, corners of the supporting polygon, and the position of the center of mass. Rays were used to visualize velocity, acceleration, distance to the destination, and value of the samples. A screenshot of our Gizmos active for an energy based simulation are shown in Figure 4.4.

Similar to the Gizmos, we utilized a feature called Handles to ease setting and visualization of the joint constraints on the skeleton. Unlike a Gizmo, a Handle can be manipulated by the user to affect values in the simulation before run time. As shown in Figure 4.5, three handles were used to adjust pitch, yaw, and roll, color coded as red, green, and blue. Colors were chosen to correspond to the colors of the reference axes placed in the scene to aid the user, showing the red, green, and blue handles as limiting rotation about the x, y, and z axes respectively.

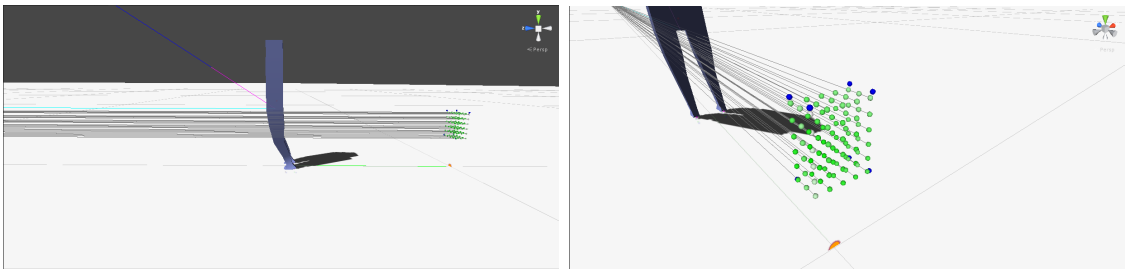


Figure 4.4: Pictured above are screenshots of Gizmos in the Unity3D Editor. The green points illustrate the sample field taken at the beginning of the jump, while the gray lines show the energy measured at this sample. The rays beginning at the player's pelvis show the magnitude and direction of the acceleration (magenta) and velocity (blue) of the character. The green line at the player's feet shows the displacement from the start position to the destination, and the cyan line starting at the player's pelvis shows the calculated kinetic energy. At right, a closer view of the samples is shown to show the blue dots which outline the balanced region which the samples were restricted to. The character is distanced from these samples as the Gizmos are drawn at the character's start position, and the character in this scene has completed its jump and is at the destination position. The orange particle is a marker placed at the start position.

4.3 Summary

In this chapter we discussed some difficulties in visualizing animations in both animated and non-animated settings. We presented some sources of inspiration and discussed the different methods we used to visualize data for presentation, analysis, and debugging. Methods we used for analysis and presentation included trails of markers, ghost images created by duplicating the player model in the scene, layering frames to create a single image, and animation strips. Our debugging visualizations took advantage of Unity3D utilities, using Gizmos and Handles to allow the user to see information about the simulation, as well as to more easily adjust constraints on the skeleton.

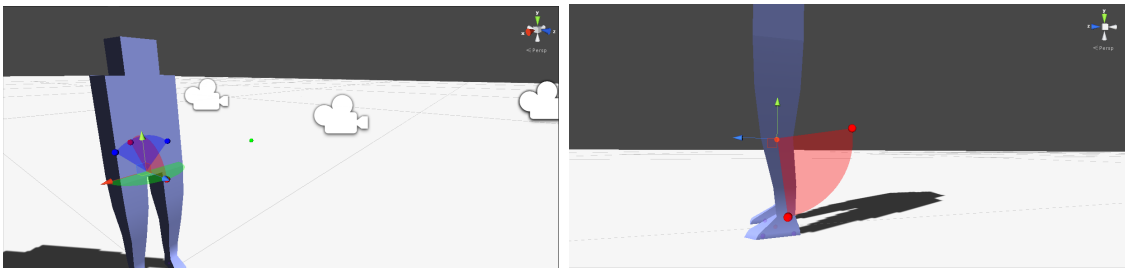


Figure 4.5: Shown above are Handles, a tool similar to Gizmos but with the addition of changing a value. In this case, the Handles affect the minimum and maximum rotation for the joints. Handles are only visible when the corresponding object is selected by the user, reducing clutter. All three Handles are shown in the left image, allowing the user to set the constraints for pitch (red), yaw (green), and roll (blue) as the pelvis is set to allow rotations about all three axes. At right, we show the Handle for adjusting the constraints of the left knee, which is only able to rotate about the x-axis (pitch).

CHAPTER 5
RESULTS

The windup PD controller was given constants of $k_p = 0.25$ and $k_d = 0.25$. These were chosen empirically to offset irregularities do to time step and slow convergence in the inverse kinematic solver. With higher values, the translation of the pelvis results in the character's feet embedding in the ground plane due to too great a movement in a single frame. The feet then fail to adjust as the inverse kinematic solver cannot converge quickly enough. These values were found to generally produce smooth windup phases without compromising on speed of the animation too much.

Muscle spring constants were tested in various configurations, with several trials run for each set of constants for varying distances, directions, and situations. Animations were created for forward jumps between 1m and 2m for the normal human values and at 1m, 10m, and 100m for the super human. A jump onto a box was also simulated, with 0.5m and 0.75m boxes for the normal human and 1m and 100m boxes for the super human. The normal human muscle constants were additionally used for sideways jumping animations, as well as a more complex scene in which the character is made to jump from on top of a box, over an obstacle before finally landing on the ground.

Jump animations appear plausible, and the spring values result in forces similar to a human muscle. Human muscles have about $30\frac{N}{cm^2}$ force per cross-sectional area. [11] The character's leg thickness is about $0.20m$ forward to back narrowing towards the knee, with a left-right thickness of around $0.15m$. If we assume that skin is about $0.002m$ thick (2mm) and about 15% of the remainder is subcutaneous fat, we are left with a $0.1683m$ by $0.1258m$ cross section. This gives an axial cross sectional area of approximately $0.021m^2$. We use a bone width of $0.05m$, giving a cross sectional bone area of $0.0025m^2$ which leaves an area of $0.019m^2$ of non-bone muscle. If half of this is extensor muscle, then we have an approximate cross sectional area of $0.0095m^2$ or $95cm^2$. This means that the estimated maximum isometric force for the muscle is $F = \left(30\frac{N}{cm^2}\right)(95cm^2) = 2850N$. With a k of 20000,

our muscle produces

$$F = -k\left(\frac{r\sin(\pi-\theta)}{\sin\frac{\theta}{2}}\right)$$

which for a joint bend of $\frac{\pi}{2}$ radians is $1414N$. A more accurate k value for the single muscle would be 40000 but 20000 suffices as the smaller muscles of the calf are modeled as overly strong. The k value was determined empirically, following the assumption that an average person has a max long jump in the range of 1.5m to 2m. A possible explanation is that our simulated character can perfectly execute the movement, maintaining balance and applying force. There is a non-trivial technical aspect to performing a jump, meaning that a human with the capability to perform a longer jump may not be capable of maximizing their range due to imperfect technique.

The produced animations are plausible and recognizable as jump animations. The character loads its limbs appropriately, giving the appearance of weight, and extends its knees, hips, and lastly ankles to show thrust corresponding to the usage of its muscles. As in a real jump, the character extends its ankles last, the calf muscle providing the final thrust of the motion.

The simulation runs in an interactive frame rate, with a delay on start up for the calculation of the sample field. As long as the mass and the muscle constants remain the same, the sample field need not be re-calculated. The sample field calculation is linear in the number of samples taken. A major bottleneck aside from populating the sample field is the convergence of the inverse kinematic solver. The solver is guaranteed to run every frame, usually for many iterations unless it has already converged. While the cost is manageable due to the linear nature of the algorithm, the main issue is when convergence does not occur and the simulation must either stop and wait for the solver to catch up, or continue on and risk corrupting the simulation due to misplaced joints or limbs. There are two major cases of this occurring: the character's feet sinking into the ground instead of the knees bending during windup and the character's feet prematurely breaking contact with the ground during thrust. Our simulation suspends its activities when a compromising situation is detected, iterating the inverse kinematic solver until the issue

is resolved. The solver usually converges within a few frames, but this can add undesired time to the simulation and undesired frames to the animation.

Frame data was collected at a rate of 1 frame per 0.1s of simulation time, giving a frame rate of 10 frames per second. This granularity was used to ensure capture of any irregularities in the simulation and to ensure changes in pose were recorded as shorter trials ran in under 1s of simulation time, though the real time was several seconds. Finer granularity was found to have little benefit. We show one of five frames in the tables discussed in section 5.1 to reduce the size of the animation strips presented and thereby reduce the size of our tables.

	Left Hip	Left Knee	Left Ankle	Right Hip	Right Knee	Right Ankle
Global, Normal	20000	20000	20000	20000	20000	20000
Varying, Normal	20000	24000	16000	20000	24000	16000
Uneven Global, Normal	16000	16000	16000	24000	24000	24000
Uneven Varying, Normal	24000	28000	20000	16000	20000	12000
Global, Super	1×10^{10}	1×10^{10}	1×10^{10}	1×10^{10}	1×10^{10}	1×10^{10}

Table 5.1: This table shows muscle spring constants (k) values used for several trial runs. Each column represents a muscle, described by the center joint which indicates the joint the muscle crosses and affects. Each row represents a different trial, with a set of k values. Global runs used a uniform k for one or both sides of the body, while varying runs used different spring constants for each muscle. Uneven runs were meant to mimic a character with an injury or other source of imbalance where one leg was significantly stronger than the other.

5.1 Output Animations

Several trials were run with different, empirically determined k values as shown in Table 5.1. The destination position was chosen for each to demonstrate the range of motions possible with our simulation. Trials can be divided into several types: forward jumps, sideways jumps, and box jumps. In a forward jump, the character starts from standing and jumps forward to a destination. A sideways jump is the same, except requiring the character to jump to the right without first turning. Box jumps required the character to jump from standing to land on an obstacle in front of them. Additionally, a more complex scene was constructed, in which the character starts standing on a box. The character then jumps off the box, over and

obstacle, and lands on the floor below.

The complex scene is pictured in Figure 5.1. The box on which the character starts is 1m in height, and the obstacle is 1.4m in height. To achieve the path shown, the destination was set at $(0, 1.5, 1)$, with the starting position at $(0, 0, 0)$. Muscle spring constants were set for all muscles globally as $k = 20000$.

Forward jumps are shown in Tables 5.2, 5.4, and 5.6. For the super human, jump destinations were set 1m and 100m in front of the character's starting position. For the normal human, jumps were set at 1m, 1.3m, 1.6m, and 1.9m, shown in Table 5.2 for a globally set k and in Table 5.4 for a varying k. Box jumps are also included in these tables, with 1m, 10m, and 100m boxes for the super human and 0.5m and 1m for the normal human. Animations were produced for normal human strength with both a globally set $k = 20000$, where each muscle had the same constant, and varying with $k = 20000 \pm 4000$ as shown in Table 5.1.

In Table 5.5 we show forward jumping motions of 1.6m and 1.9m where the character has uneven strength in their legs. These values were chosen similar to the normal human run. There is little effect, which is why lengths shown were only at the extreme of distance possible for the character. However, the character's right leg can be seen to dangle in the trial with varying k values, indicating that more load is on the left leg.

To demonstrate the flexibility of a simulation-based animation, we also include a jump to the side. The destination was specified as 1m, 1.3m, and 1.6m to the right, i.e. the direction $(1, 0, 0)$ relative to the character, of the character's start position. Instead of the pelvis thrusting forward and up, in these trials the pelvis thrust is to the side and upwards as expected. These animations are shown in Table 5.3 and were collected with a global $k = 20000$.

5.2 Limitations

Our system has a number of limitations and failure cases. First is that there are many constants to be specified, which is work intensive but gives freedom to make wide changes to the animation by tuning parameters

There are numerous small issues with the calculations caused by strange or

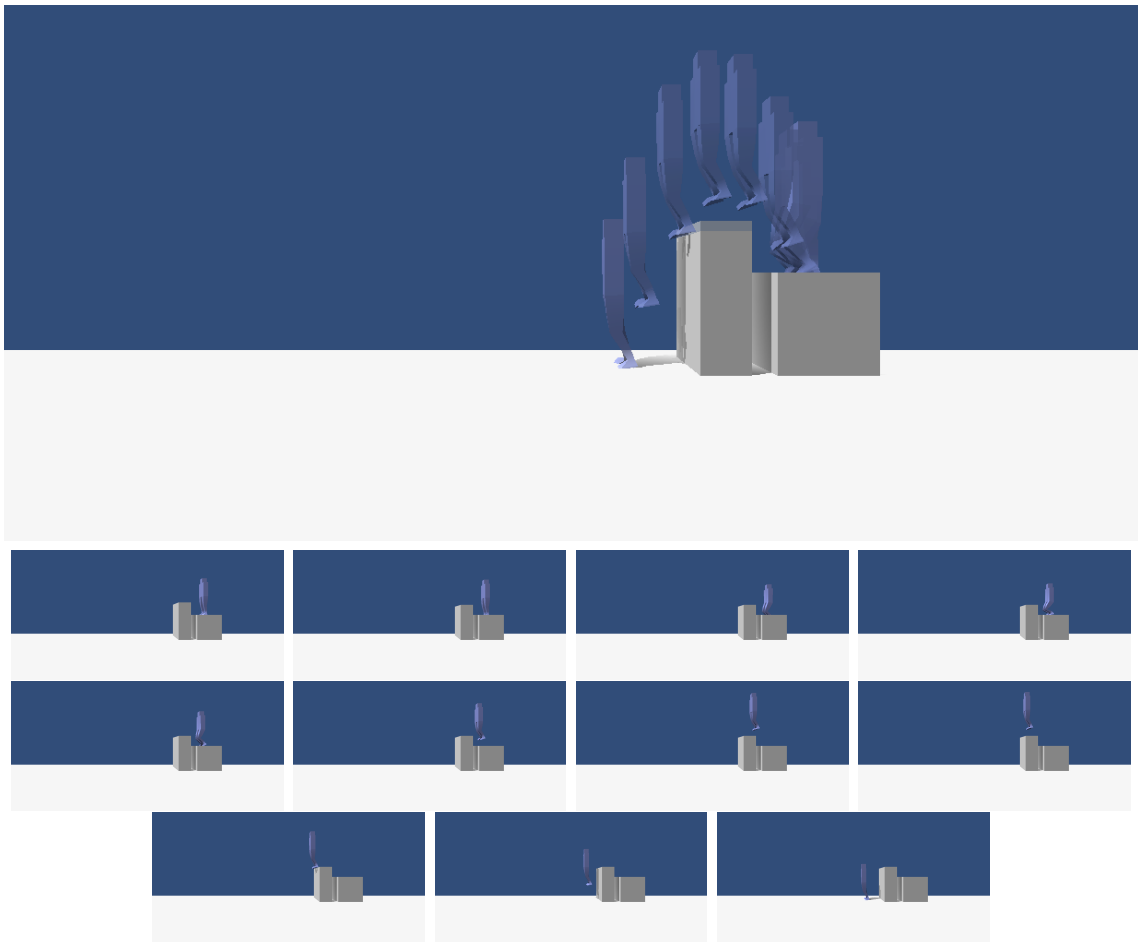


Figure 5.1: Pictured is an animation of a complex scene, in which the character must jump from on top of a box, over another box and land on the ground. The first image in the figure shows the frames composited into one image to visualize the full motion, while the remaining images show the individual frames. This run used $t_{windup} = 0.2s$ and $t_{air} = 0.5s$.

unexpected interactions. Foremost is the behavior of rotations and angle measurement in Unity3D. Angles are read and interacted with as Euler angles, pitch, roll, and yaw, or rotation about the x, z, and y axes respectively. Rotations, however, are stored and calculated by the engine in the form of quaternions. Due to the conversions between the two, and various manipulations that occur in the scene, this can result in angles not restricted to $[-360, 360]$ degrees, and can result in jumps between positive and negative angles. A solution would be to restrict the angles to positive angles in the range $[0, 360]$ for all calculations and manipulations, but this

makes specifying constraints more complex.

Problems with angle also arise as the angle of the joint does not necessarily reflect the angle between the bones. For example, when the knee is fully extended, the joint angle stored in the object is 0, but the angle between the bones is π radians. A solution is to calculate the angle between the bones of the joint when needed using the dot product of the vectors between the joint and its parent and the joint and its child. We did not realize that there was still an issue with angle specification as the issue was balanced out as the supplementary angle was used erroneously, but a fully correct implementation would be desirable for true consistency. The data was collected with the flawed implementation, though as previously stated it produced similar results.

Movement of the upper body is very minimal, and is quite unlike a human performing a long jump. This is likely due to the restriction of pelvis reposition to the region over the supporting polygon. Humans frequently move their pelvises far behind their supporting polygon, compensating using the weight of their upper body. The usage of rapid movement of the upper body to aid in acceleration is also a factor we do not consider, such as the effects of arm swing on a jump.

As our simulation was focused on the movements leading to the airborne phase, the handling of in air maneuvers and landing are overly simplistic. After the character becomes airborne, the pelvis will generally be displaced in the direction of acceleration relative to the feet. The character should maneuver while airborne such that their feet are in front of their body to prepare for landing. For landing, a reverse of the windup for the energy simulation could be used, loading the muscles in the legs to offset the kinetic energy the character has from the jump, converting it to elastic energy in the muscles. Landing and airborne phases should be handled ideally by a separate controller.

Our inverse kinematic solver is also very simple, and brings its own issues due to this simplicity. This algorithm was chosen specifically to minimize time, effort, and resources spent on the inverse kinematic component, in favor of the simulation itself. With a different inverse kinematic solver, better results could be achieved.

The output of our simulation is currently image frames as well as the direct

visualization through Unity3D, as opposed to a key frame animation in a format such as FBX, which could be utilized in video games. An AutoDesk Maya plugin that runs our simulation would also be more useful for allowing creation of animations if a real-time frame rate cannot be achieved.

5.3 Summary

Plausible animations were created, with empirically determined k values around 20000. Analysis shows different values would be theoretically more realistic, but the empirically determined values still produced reasonable animations. Animations were produced for a normal strength human as well as a super human for forward standing jumps, sideways standing jumps, and box jumps. A scene was constructed in which the character jumped from on top of a box, over an obstacle, and landed on the ground below to show a more complex scene. We discussed in this chapter the values used and method for collecting data, and presented sets of frames collected from several simulations with a variety of scenes. Animations depicted forward, sideways, and box jumps for a normal human range of strength and for a super human strength. An animation of a character jumping over an obstacle was also presented.

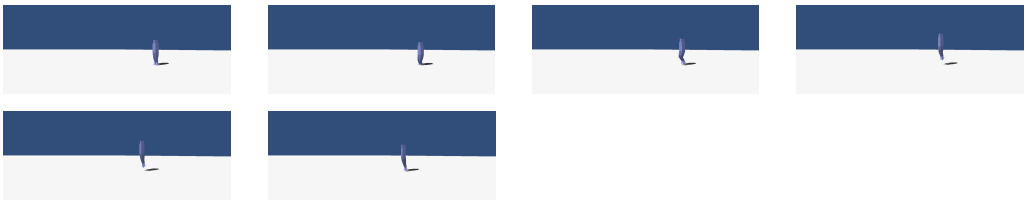
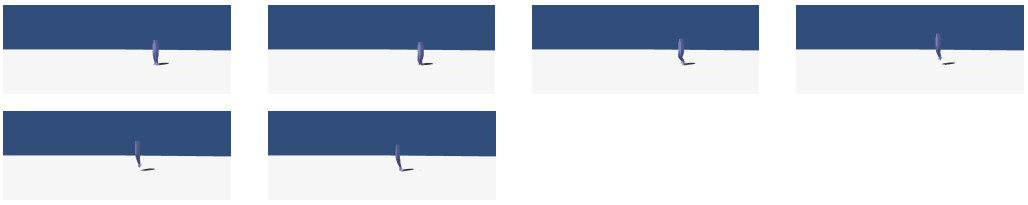
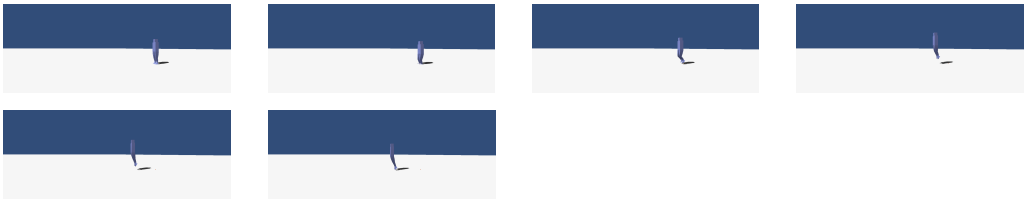
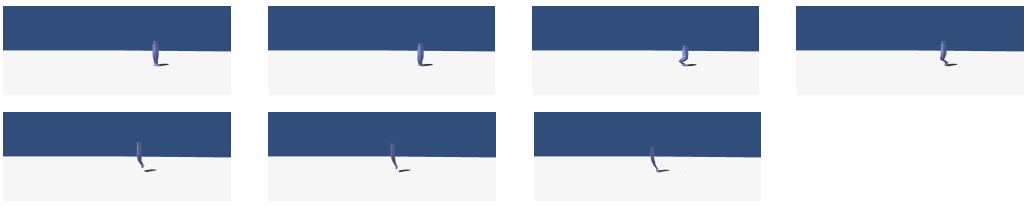
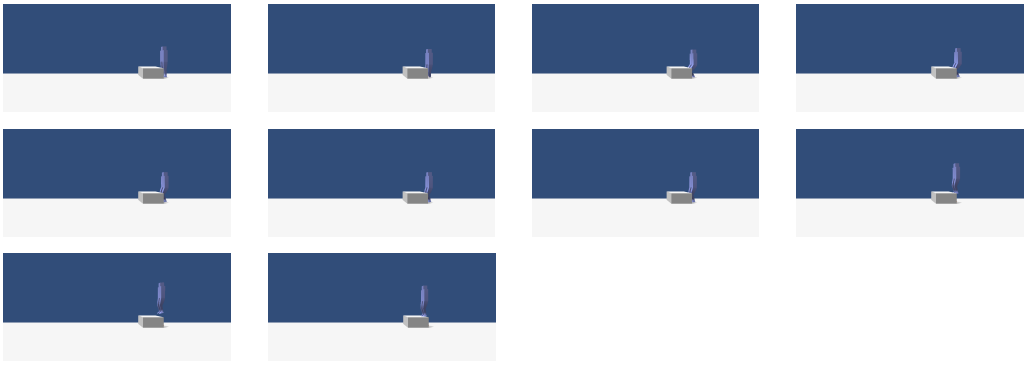
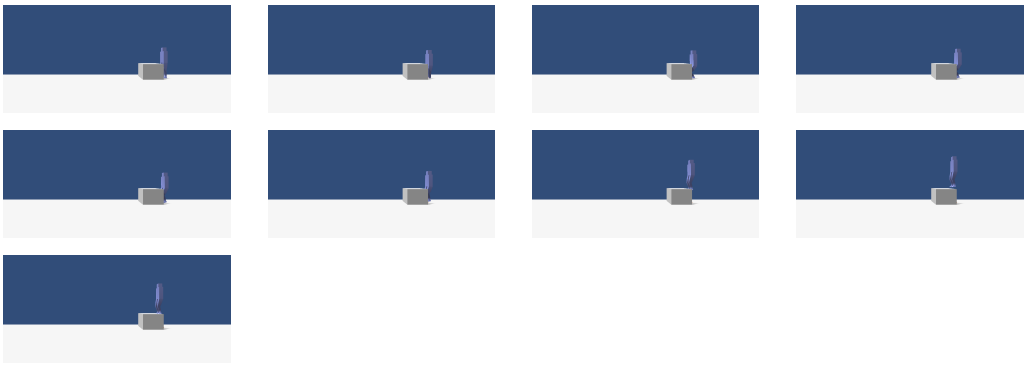
1m forward	
1.3m forward	
1.6m forward	
1.9m forward	
0.5m box	
0.75m box	

Table 5.2: Table of forward and box jump motions for a character with global $k = 20000$, $t_{windup} = 0.2s$, and $t_{air} = 0.5s$. The boxes were placed 0.75m in front of the character, with the character's target destination set 0.3m in front of the character on top of the box.

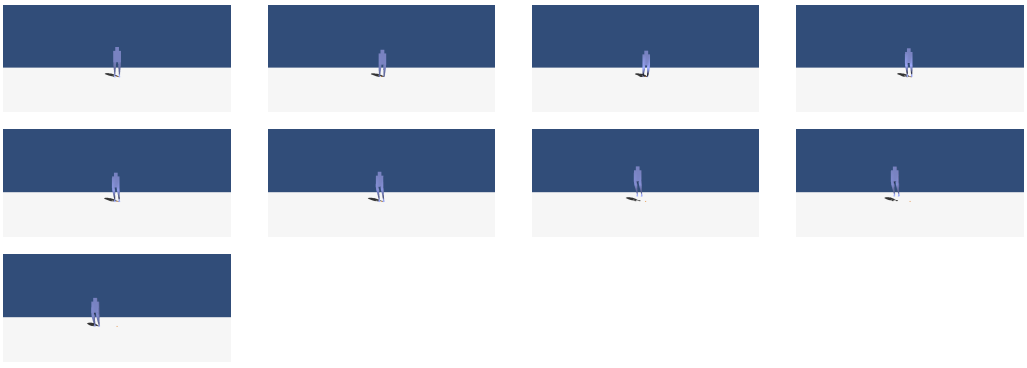
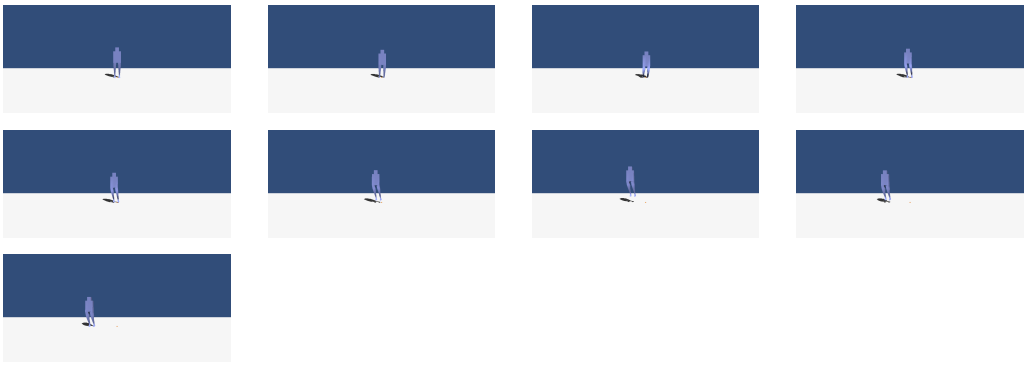
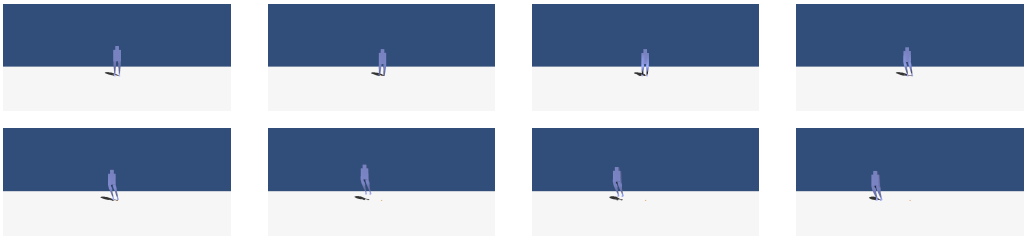
1m right	
1.3m right	
1.6m right	

Table 5.3: Table of right jump motions for a character with global $k = 20000$, $t_{windup} = 0.2s$, and $t_{air} = 0.5s$. The target was placed at the distance listed in the table in the direction $(1, 0, 0)$ relative to the character.

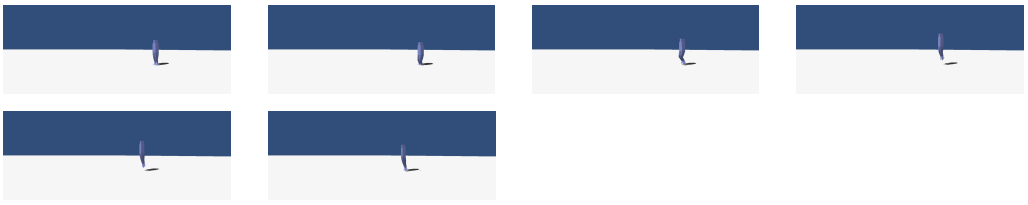
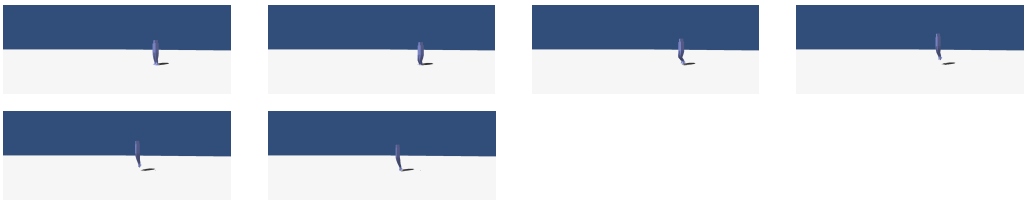
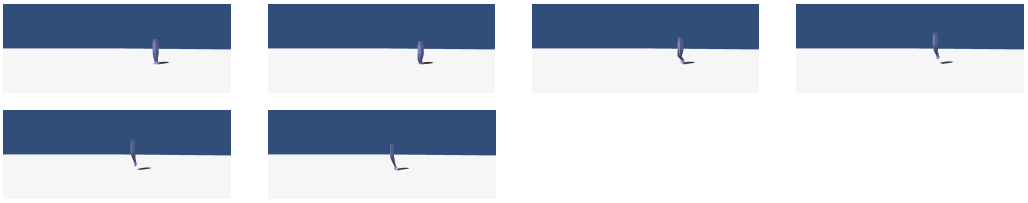
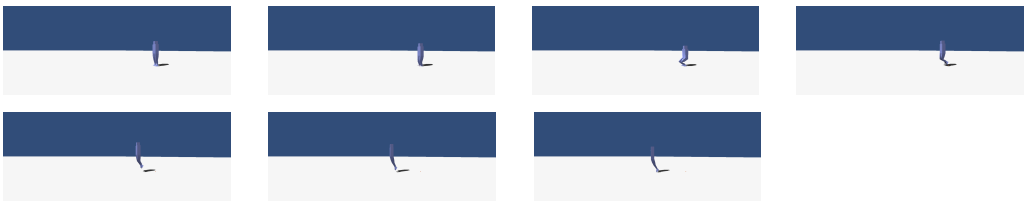
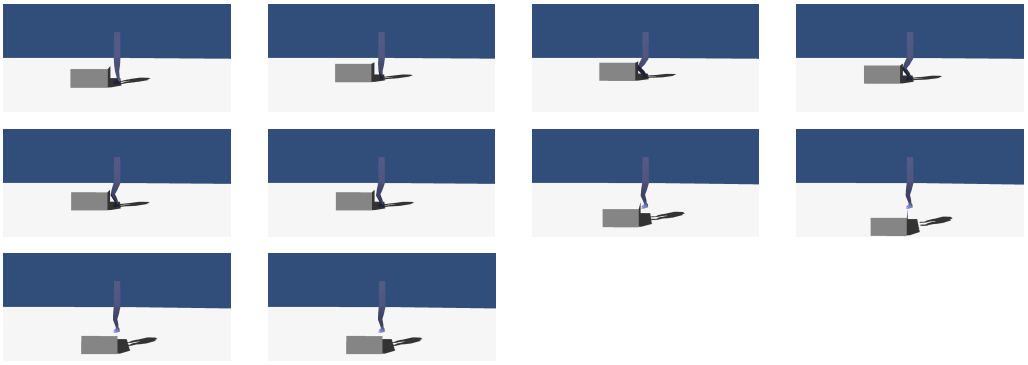
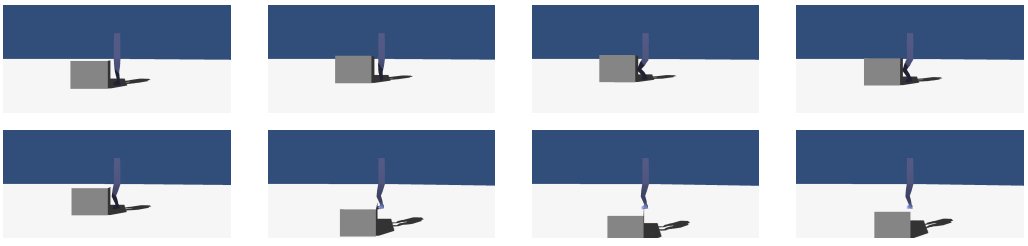
1m forward	
1.3m forward	
1.6m forward	
1.9m forward	
0.5m box	
0.75m box	

Table 5.4: Table of forward and box jump motions for a character with varying $k \approx 20000$, $t_{windup} = 0.2s$, and $t_{air} = 0.5s$. The boxes were placed 0.75m in front of the character, with the character's target destination set 0.3m in front of the character on top of the box.

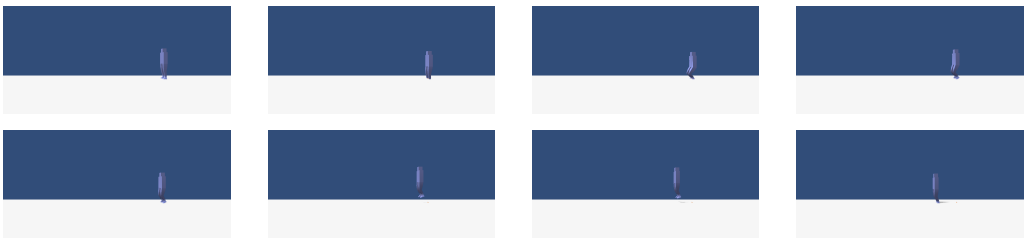
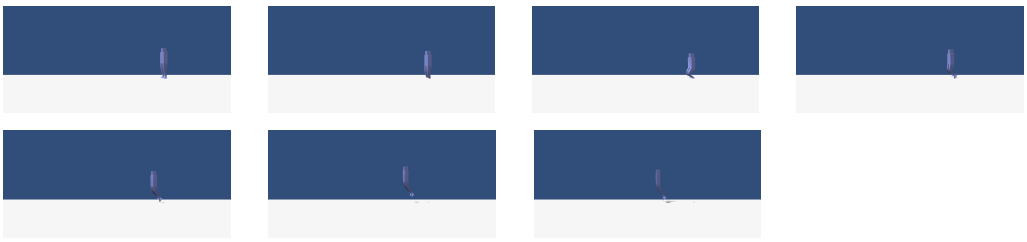
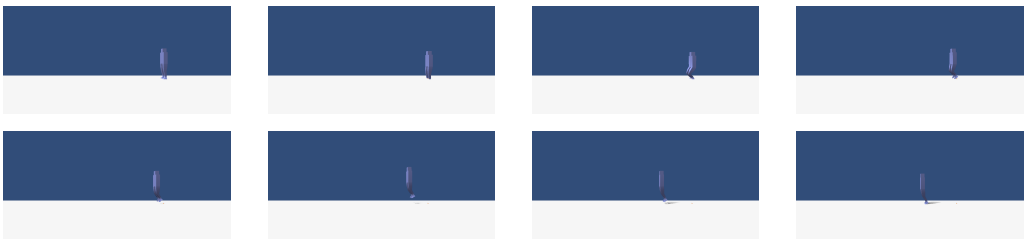
1.6m forward (global $k = 20000$)	
1.9m forward (global $k = 20000$)	
1.6m forward (varying $k = 20000$)	
1.9m forward (varying $k = 20000$)	

Table 5.5: Table of forward jumping motions with uneven muscle strengths between legs, using $t_{windup} = 0.2s$ and $t_{air} = 0.5s$.

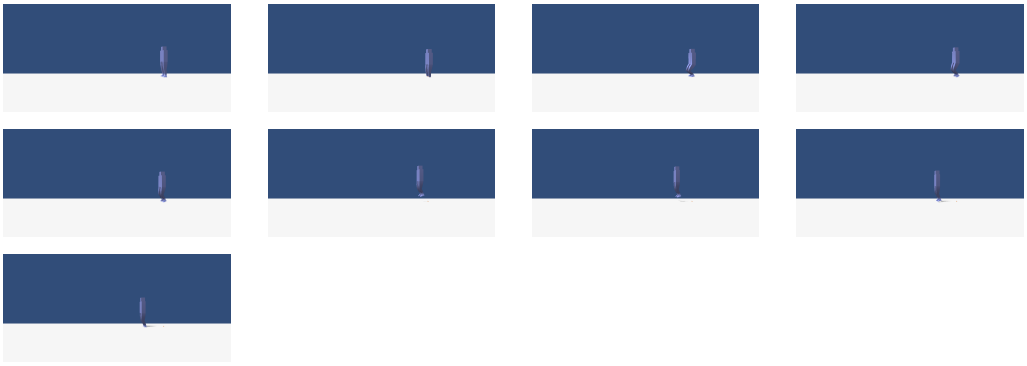
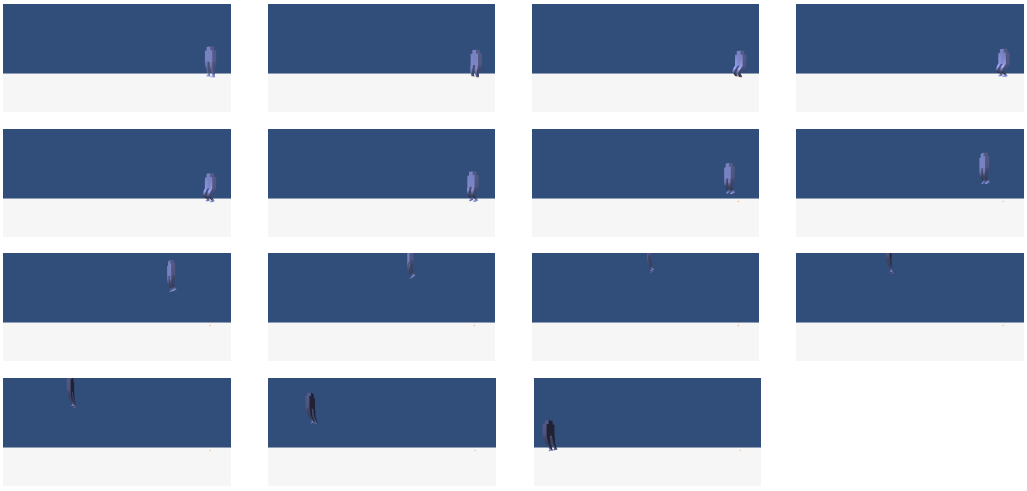
1m forward	
10m forward	

Table 5.6: Above are generated frame sequences for the super human trial, where the k values were chosen such that the character could leap over a tall building, a 100m tall box. Animations above were generated for 1m, 10m, and 100m forward jumps. The 100m forward jump is not pictured due to the difficulty of capture, as either the jump was out of the range of the camera or the camera was too far to clearly see the animation.

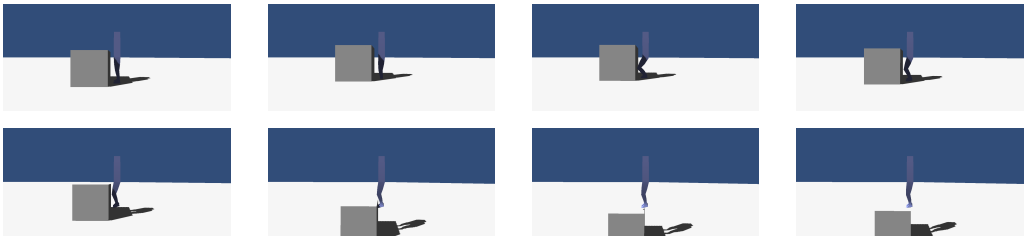
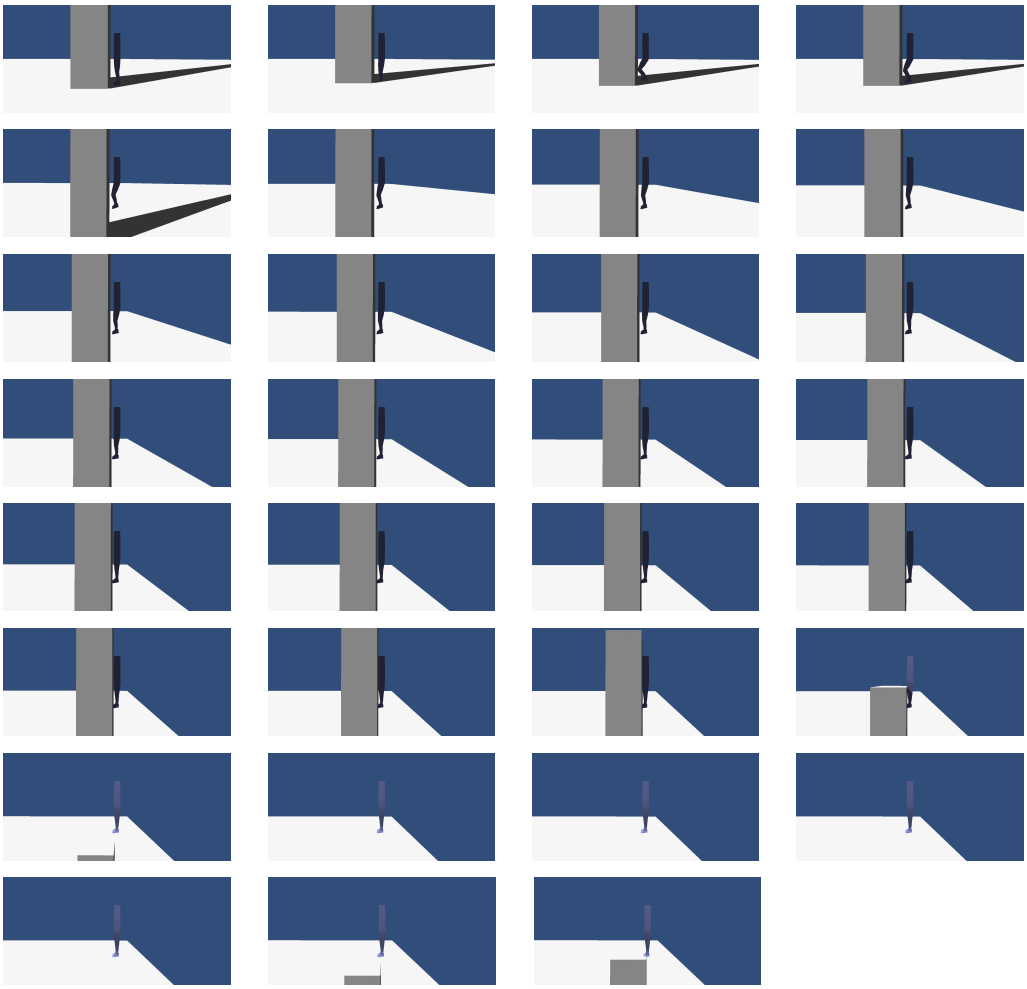
1m box	
100m box	

Table 5.7: Animations of 1m and 100m box jumps for the super human. This run uses $t_{windup} = 1s$ and $t_{air} = 5s$.

CHAPTER 6
FUTURE WORK AND CONCLUSION

In this chapter we discuss future work and conclude our discussion of research on this simulation. Section 6.1 discusses possible and planned work on this topic, as well as potential future work in this and closely related lines of study. Section 6.2 concludes this document with final thoughts and a summary of the document.

6.1 Future Work

We planned some further work that we decided was beyond the scope of this thesis. Our current system does not allow much flexibility with specification of the path the character travels for its jump. Jumping path estimation could be performed based on a policy. Possible policies are achieving a height while jumping to a target destination, pathing to clear an object or intersect with an object, follow a path defined by the user, and jumping with a user specified velocity or speed. These policies would require a smarter handling of the in air phase of the jump, which would be best implemented as a secondary controller to allow controller composition for more complex motions. A more complex in air controller would ideally handle cases such as acrobatics, in air maneuvers, and checking for and handling collisions.

Current work exists for landing motions such as described in chapter 2. Ha et al. describes an example of one such controller for a falling and landing motions. Incorporation of such other controllers would allow creation of more complex animations. A separate controller could also be used for improving the motions of the upper body for each of these phases. This could be used to create complex freerunning animations such as vaults and wall runs which are becoming prevalent in video games such as Mirror's Edge.

To help with choosing values, a learning model could be applied. Animations could be marked as successful and desirable by humans to train an algorithm to choose desirable constants for the muscles given target destinations. Machine learning could also be applied for learning a function to determine muscle load in the

windup phase of our simulation. Intuitively the situation seems to fit a learning model well, but more study would be required.

Both simulations were solved using a sampling solution, but could have been solved using an optimization problem. Solving the optimization problem, such as the quadratic program in 3.6.2 would likely provide a better solution and would give stronger guarantees of optimality. This would likely decrease performance.

Our animation output is currently images. A more desirable animation output would be key frames storing the positions and orientations for each joint of the character's skeleton, which could then be used in a game or video as a pre-baked animation. An implementation as a plugin for AutoDesk Maya could also be more desirable, as it could then be incorporated into an artist's work flow. We chose not to use AutoDesk Maya initially for an implementation initially due to familiarity with Unity3D and so that we could obtain live visuals of the simulation with debugging information easily as the simulation was performed and animation played.

Another option would be to utilize MecAnim, a feature of Unity3D. We chose not to use this feature while performing the initial research, due to lack of understanding of the limitations and features available in MecAnim. After completing further research, there are many components of MecAnim that would improve our simulation and increase its ease of incorporation into game development and animation workflows. Our simulation could be re-tooled to output animation clips for MecAnim, and our constraint system replaced with the one provided by MecAnim. MecAnim muscles do not provide the functionality required, so our muscle component would need to be modified and reapplied to the skeleton. As is, our simulation is very close to producing output that could be used as an input to MecAnim, making this a promising direction for future work on the implementation.

6.2 Conclusion and Summary

In this thesis, we discussed the need for a more efficient way to produce character animations for video games and film. We then presented a simulation based approach for creating such animations for a jumping motion of a character to reach a given target position. Our system used two different types of simulation: torque

based and energy based. The torque based simulation failed to produce good results, but we collected frame data for a variety of situations using the energy based simulation. We then described our methods for visualizing the animations to quantify and qualify the performance, giving visual information in an animated format and still format.

References

[1] S. Ha, Y. Ye, and C. K. Liu, "Falling and landing motion control for character animation," *ACM Trans. Graph.*, vol. 31, no. 6, pp. 155:1–155:9, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2366145.2366174

[2] T. Geijtenbeek, M. van de Panne, and A. F. van der Stappen, "Flexible muscle-based locomotion for bipedal creatures," *ACM Transactions on Graphics*, vol. 32, no. 6, 2013.

[3] P. Faloutsos, M. van de Panne, and D. Terzopoulos, "Composable controllers for physics-based character animation," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 251–260. [Online]. Available: http://doi.acm.org/10.1145/383259.383287

[4] J. K. Hodgins, W. L. Wooten, D. C. Brogan, and J. F. O'Brien, "Animating human athletics," in *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '95. New York, NY, USA: ACM, 1995, pp. 71–78. [Online]. Available: http://doi.acm.org/10.1145/218380.218414

[5] T. Geijtenbeek and N. Pronost, "Interactive character animation using simulated physics: A state-of-the-art review," *Computer Graphics Forum*, vol. 31, no. 8, pp. 2492–2515, 2012. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2012.03189.x

[6] Y. Koga, K. Kondo, J. Kuffner, and J.-C. Latombe, "Planning motions with intentions," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '94. New York, NY, USA: ACM, 1994, pp. 395–408. [Online]. Available: http://doi.acm.org/10.1145/192161.192266

[7] S. Bennett, "Nicholas minorsky and the automatic steering of ships," *IEEE Control Systems Magazine*, vol. 4, no. 4, pp. 10–15, November 1984.

[8] D. C. Boone and S. P. Azen, "Normal range of motion of joints in male subjects." *The Journal of Bone & Joint Surgery*, vol. 61, no. 5, pp. 756–759, 1979. [Online]. Available: http://jbjs.org/content/61/5/756

[9] J. Lander, "Oh my god, i inverted kine!" *Game Developer*, pp. 15–22, September 1998.

[10] J. Lander, "Making kine more flexible," *Game Developer*, pp. 15–22, November 1998.

[11] P. M. McGinnis, *Biomechanics of Sport and Exercise*. Champaign, IL, USA: Human Kinetics, March 2013.

[12] i. Pei-shan Xie1 and i. Xing Cai2, "Research on computer modeling of human movement." *Advanced Materials Research*, no. 926-930, pp. 4190 – 4193, 2014. [Online]. Available: http://libproxy.rpi.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=aci&AN=96297571&site=eds-live&scope=site

[13] S. Jain and C. K. Liu, "Controlling physics-based characters using soft contacts."

[14] M. Alexander and A. Honish, "Footwork for the volleyball block."

[15] T. Amasay, "Static block jump techniques in volleyball: Upright versus squatting start position."

[16] S. Jain, Y. Ye, and C. K. Liu, "Optimization-based interactive motion synthesis," *ACM Trans. Graph.*, vol. 28, no. 1, pp. 10:1–10:12, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1477926.1477936

[17] R. Lobietti, S. Fantozzi, R. Stagni, and S. G. Coleman, "A biomechanical comparison of jumping techniques in the volleyball block and spike."

[18] S. Coros, P. Beaudoin, and M. van de Panne, "Robust task-based control policies for physics-based characters," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 170:1–170:9, Dec. 2009. [Online]. Available: http://doi.acm.org/10.1145/1618452.1618516

[19] H. C. Sun and D. N. Metaxas, "Automating gait generation," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 261–270. [Online]. Available: http://doi.acm.org/10.1145/383259.383288

[20] J. Babič and J. Lenarčič, *Vertical Jump: Biomechanical Analysis and Simulation Study.* InTech, June 2007, ch. 31.

[21] F. E. Zajac, M. R. Zomlefer, and W. S. Levine, "Hindlimb muscular activity, kinetics and kinematics of cats jumping to their maximum achievable heights," *The Journal of Experimental Biology*, vol. 91, pp. 73–86, 1980.

[22] F. E. Zajac, "Thigh muscle activity during maximum-height jumps by cats," *Journal of Neurophysiology*, vol. 53, no. 4, pp. 979–994, 1985.

[23] J. Hayes, "Creating character animation assets," pp. 2–3, November 1999. [Online]. Available: http://www.gamasutra.com/view/feature/131796/creating_character_animation_assets.php