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

July 2015
(For Graduation August 2015)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

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 using keyframing. In a keyframe animation, certain "key" parts of the animated sequence are specified, with the remaining frames filled in, or "tweened" using an automated interpolation method or manual frame addition. For 2D animation, this occurs as a series of images which are played back in order to produce the animation. In 3D, keyframe animations are performed on a 3D model.

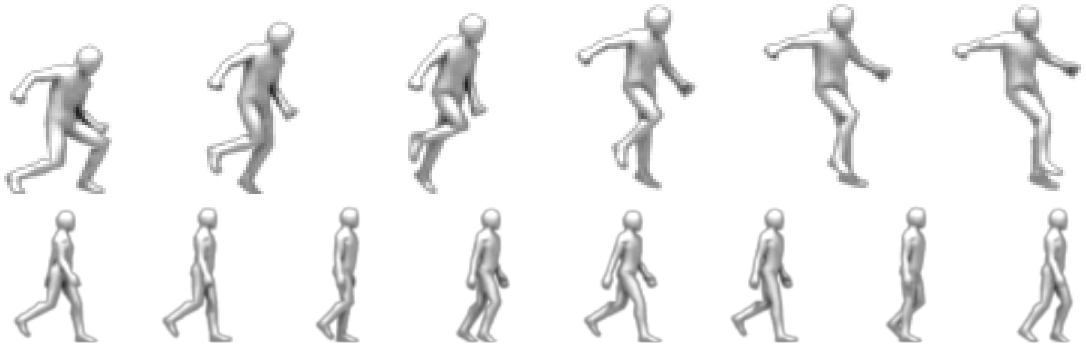


Figure 1.1: This example shows a 2D sprite sheet used to produce a jumping animation for a stick figure character. The frames in this case are laid out in a single image for demonstration purposes, progressing in order starting with frame 0, the frame farthest left in this sprite sheet.

3D models are described as a mesh, a collection of primitive polygons (i.e. quadrilaterals or triangles) which are stored as vertices. This mesh describes what is drawn, including any texture, color, and other material information. Along with the mesh, a skeleton, or rig, is stored. The rig describes a heirarchical structure of bones and accompanying joints. Each vertex is given a series of weights describing the impact each joint has on its transformation. This allows many vertices, and

therefore many polygons, to be transformed at once in organized groups, simplifying the problem of animating the model to a matter of transforming the skeleton in the desired manner. To animate this 3D model, an artist specifies keyframes of the animation by positioning the skeleton at different time steps. The stored keyframes, instead of an image, are the transformations of each joint at this frame or step of the animation, which a rendering or game engine can interpolate between to produce the final result.

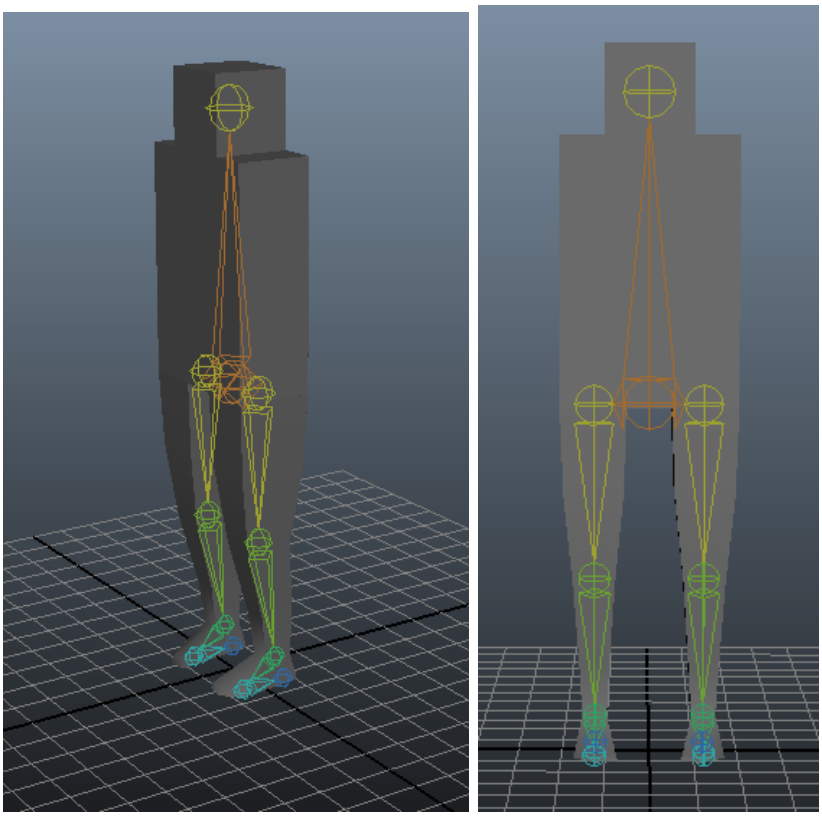


Figure 1.2: Above is an example of a character model in AutoDesk Maya . The character skin or mesh is shown in gray, with a rig shown in multicolor. While it is visualized as a series of spherical joints with connecting solids, the rig itself does not have a visual component in practice. The rig acts as a skeleton, deforming the mesh of the character model to make animation easier. Additional tools such as deformer groups and inverse kinematics 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.

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 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 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, which is not possible with manual keyframe animations.

We present a controller that takes a skeleton as input, with additional parameters describing the character, which produces a sequence of poses for a keyframe animation. The additional character parameters describe the character's mass as well as the constraints placed on each joint to prevent unnatural rotations. Weight is specified per-joint to allow for calculation of the character's center of mass. Our controller works with the initial flex and takeoff stage of the jumping motion. The motion is calculated by modeling muscles as simple springs attached to the skeleton at 2 points. Spring constants for the muscles are determined by the user, which are applied in a linear spring calculation to determine the approximate change from rest length required to achieve a particular force. This change from rest length approximates the flexion required, which can be used to calculate a plausible amount of bend for the wind-up motion of a jump.

To control the motion and maintain plausibility, calculations are performed to determine the character's center of mass and supporting polygon. As the character should 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 poly-

gon. Flexion proceeds with the character bending progressively, maintaining balance while moving to achieve the desired spring force in its leg muscles.

The next stage of the motion, the take-off where the character releases from the ground, uses the potential energy of the spring-muscles and accelerates the character's center of mass upward. Application of force works from the joints and muscles closest to the center of mass outwards through the skeleton. While not handled by our controller, the character would then proceed through the in-air portion of the jump, where the acceleration changes due to gravity as well as other forces 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. Other work has handled landing with a similar approach. [1]

Our controller is made to be a module, able to be used with other controllers as part of a larger system. 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. Additionally, certain cases during the duration cause the controller to stop early, for example a mid-air collision mid jump which would require a separate controller to handle this case. [2]

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 composed 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 for hand or arm movements. 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. 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 keyframe. Keyframes 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 framerate 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 keyframes can be sparse and there are tools for aiding in generation, this process requires heavy manual input for each animation for each character. Characters that move differently due to differences in weight or body makeup must be animated separately, requiring an artist to perform similar, time consuming work. With a simulation based approach this work could be reduced to setting variables such as weight, height, and musculature.

1.2 Contributions

This thesis describes a controller which simulates a jumping motion on a character. The generated animation is created to be plausible in appearance, though it may not be a physically accurate representation. Specifically, we contribute a model for the windup and take-off phases of a jump and created a controller using

this model in Unity3D .

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. The system developed leverages the provided features through the Unity3D scripting interface. 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 game objects with attached Transform components which allow arbitrary transformations.

CHAPTER 2
PREVIOUS WORK

2.1 Background

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. 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 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 [3].

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 [4].

Physical simulations utilize a rigid-body character with a user-defined skeleton to find optimal poses based on desired conditions. 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 [2]. Hodgins et al. created several controllers for running, vaulting, and bicycling, creating realistic motions and secondary motion using rigid bodies and spring-mass simulations [5]. Geijtenbeek and Pronost provide a detailed review of physics based simulations [6].

2.1.1 Existing Technologies

Unity3D mecanim applies constructed animations of various types to similar skeletons. This requires your animation to be constructed manually or generated through a system before it can be applied.

3ds max footsteps offer a method of positioning feet and producing rudimentary animations, but require manual input and tweaking for a a full animation.

CHAPTER 3
ANIMATION

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

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 due to unknown reasons. 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.

3.1 Setup and Inputs

The character in our system consists of a mesh, a skeleton, 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 more heavily on the lower body motion of the figure. A more complex human character, or even a humanoid non-human character 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 producing two triangles.

Our mesh was created using AutoDesk Maya , positioning the vertices in groups 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 3.1.2, 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 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 keyframe in an animation storing information about each joint instead of each vertex.

3.1.1 User Specified 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 multiplicative 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.

Figure 3.1: Examples of jumps with poorly selected times.

Error allowance in our simulation is a widely used value indicating a percent error tolerance. This tolerance is used for determining the allowable difference

Table 3.1: Values for calculated necessary velocity given air and windup times for a skeleton with muscle k values around 20000.

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. Below is an example of the inequality using the error allowance.

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

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. 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 calculation is as follows.

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

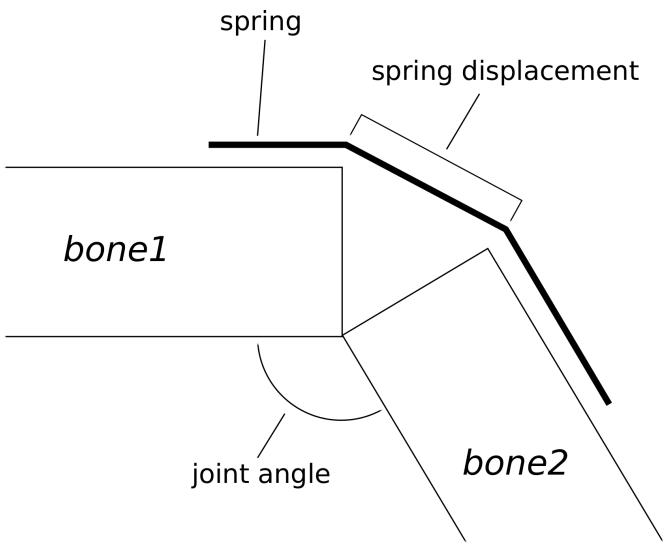


Figure 3.2: Visual representation of the setup of bones and muscles for a single joint. This illustrates the method by which we calculate the spring displacement, taking the bones as rigid blocks which separate and stretch the string as the joint bends.

3.1.2 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.

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, which makes the constraints some-

what more complicated. Constraints 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.

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, as seen in [3]. 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.

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.

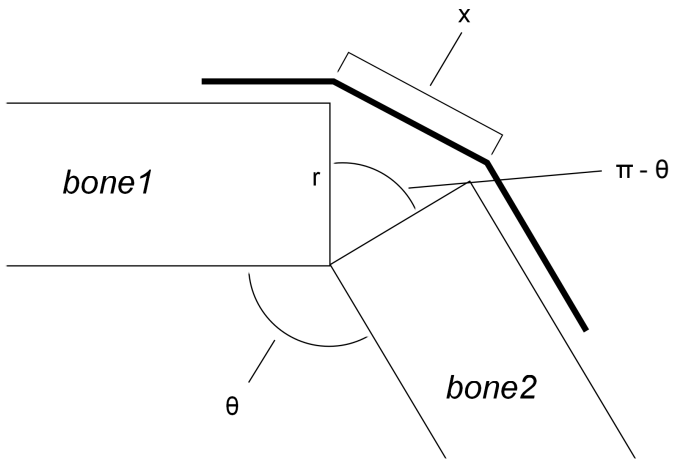


Figure 3.3: Above is a visual representation of the angles used to calculate the spring displacement. The angle of the joint, θ, is used to calculate the displacement s using the bone width r and the law of sines.

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 3.2 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 utility purposes we have each muscle calculate its utilization, giving a value in $[0, 1]$. 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.

3.2 Center of Mass and 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 proscribed 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 without using its legs 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.

3.3 Inverse Kinematic Solving

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, the joint angles and positions of the knee and ankle are solved for. 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.

Table 3.2: Joint angle constraint values used for each joint, with accompanying images of expected motion range.

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. 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. This algorithm is shown in 3.1.

Algorithm 3.1: Given chain of joints C, move joint E to position D using cyclic coordinate descent (TODO cite CCD references). 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.

function SINGLECHAINIK(C, D, E)
  repeat
    for all Joints R between E and the root, starting with the end E **do**
      $\theta_R = \mathbf{RD} \times \mathbf{RE}$
    end for
  until *Desired number of iterations performed* or **E** *is close enough to* **D**
end function

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 must be determined, through a minimum acceptable distance. 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.

3.4 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. This method failed for unknown reasons. Torques on the joints results 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. [3] 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.

3.4.1 Path Estimation

Figure 3.4: Example of a path estimation.

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.

The user specifies a desired time (t_{air}), which 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. This is useful as the desired animation can be more easily adjusted to fit a desired time as an in-game animation or to fit a particular storyboard for an animated film sequence.

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 displacement ($\mathbf{x} - \mathbf{x_0}$), 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.

3.4.2 Windup

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 3.1.2. 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.

3.4.2.1 Sampling

In order to determine the position of the character that produces this acceleration, we sample uniformly over the region in which the character is balanced. 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. 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 as for the samples but for the current state of the skeleton. 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 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

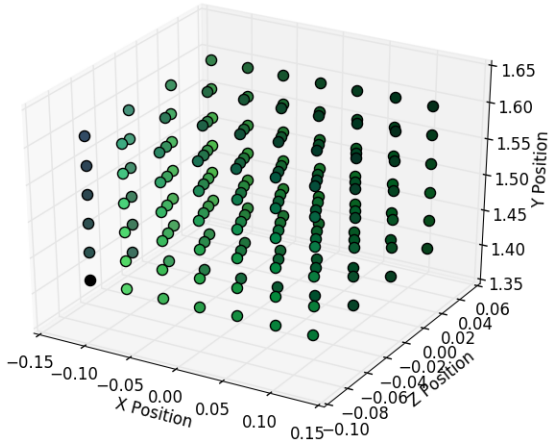


Figure 3.5: Pictured above is a plot of instant linear acceleration samples at different pelvis positions within the balanced region. Color indicates the magnitude, with the red, green, and blue channels corresponding to the x, y, and z components respectively of the acceleration sample. Lighter color 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.

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.

3.4.3 Thrust and Takeoff

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 as described in 3.1.2. At full extension, 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.

3.5 Energy Based Calculation

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.

3.5.1 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 3.4.1. Instead of

Like with the torque-based method, we took samples in the region where the character maintains balance as in 3.4.2.1. 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.

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

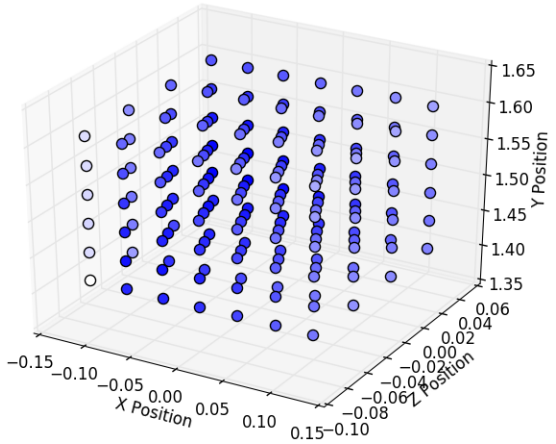


Figure 3.6: A graph of samples for the character with all muscle k values set to 20000. The pelvis positions are plotted, with the color of the data point indicating the energy. A darker point indicates a higher total elastic energy calculated for the skeleton when the pelvis is positioned at that point.

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.

3.5.2 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.

3.5.3 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 3.1.2, 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. 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.

3.6 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

4.1 Motion Visualization

For visualizing motion of a character or figure, there are a limited selection of different techniques. Most common is a sequence of frames in which a character is posed, either in a still sequence or as a video. As this is a final goal of our system, this is a valid visualization, but fails to provide a simple comparison between one animated sequence and another. This is desirable for qualifying or quantifying performance of the system. A sequence of still images is also space-consuming, which can be undesirable for print formats or even digital formats where length or size of document is an issue.

Specific markers can be 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.

4.2 Summary

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. [7] 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.

	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	1×10^9	1×10^9	1×10^9	1×10^9	1×10^9	1×10^9

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 5.5 and **??**. 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. 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.

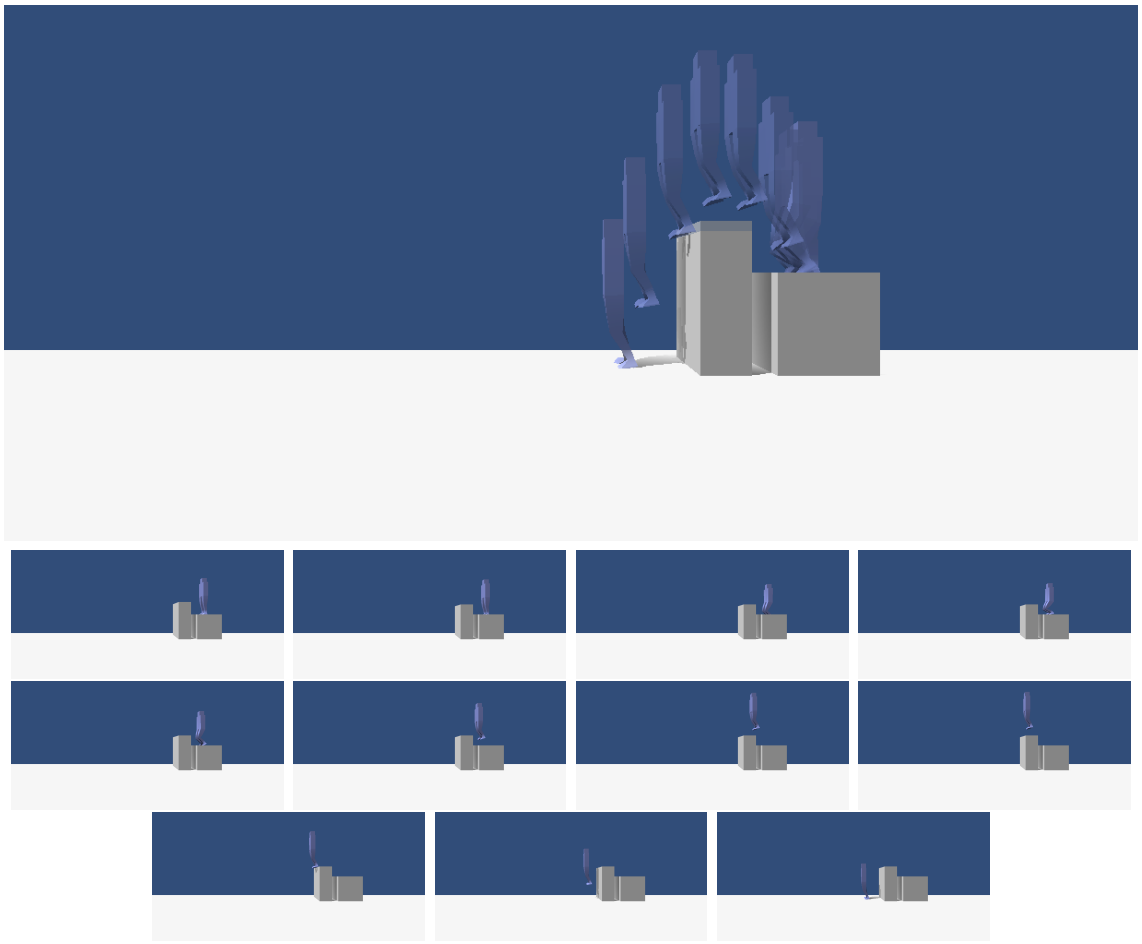


Figure 5.1: Pictured is an animation of a complex scene, in which the character must jump from on top of a box, over another box and land on the ground. The first image in the figure shows the frames composited into one image to visualize the full motion, while the remaining images show the individual frames.

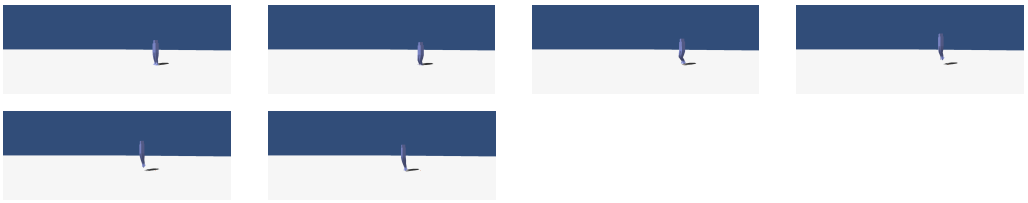
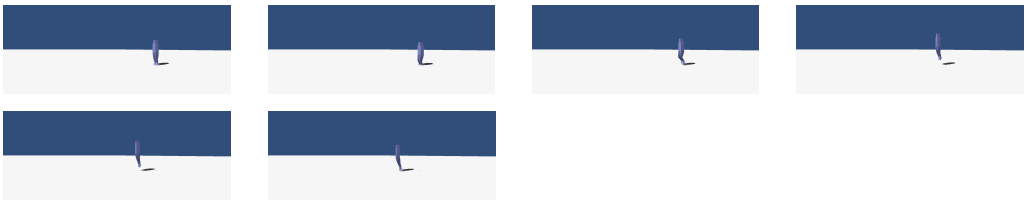
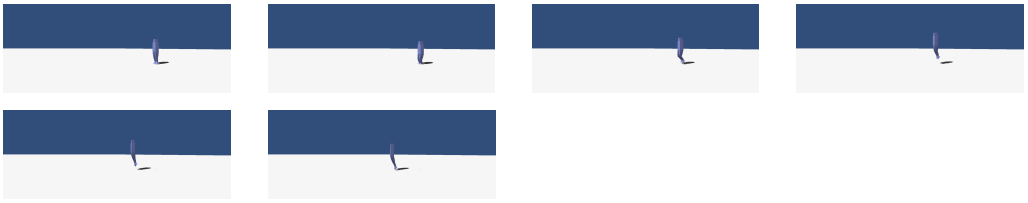
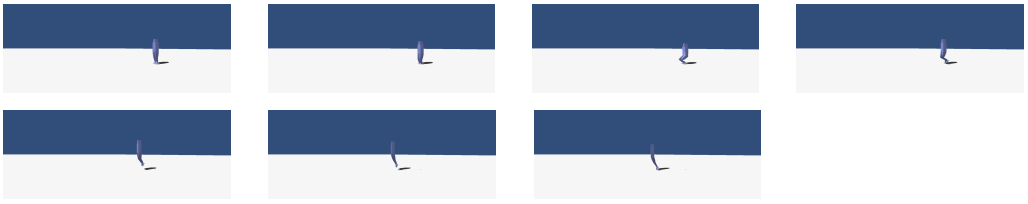
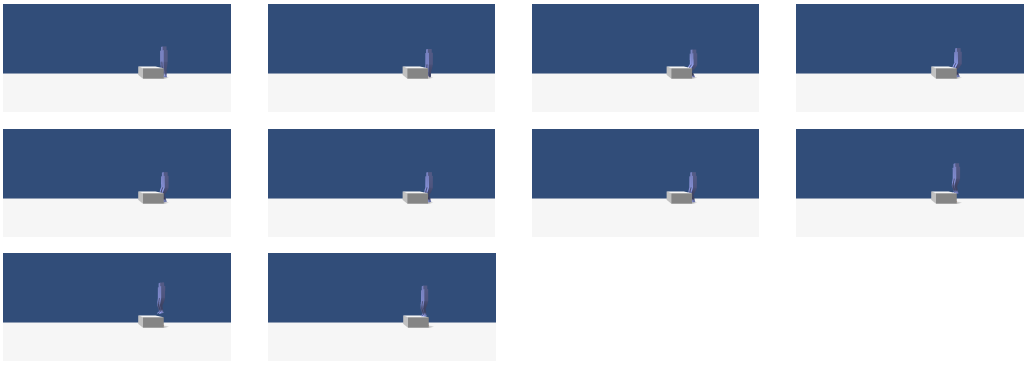
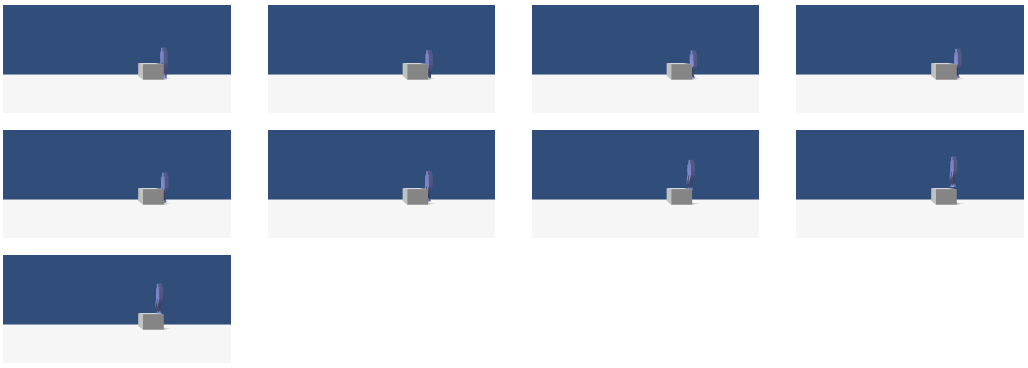
1m forward	
1.3m forward	
1.6m forward	
1.9m forward	
0.5m box	
0.75m box	

Table 5.2: Table of forward jumping motions

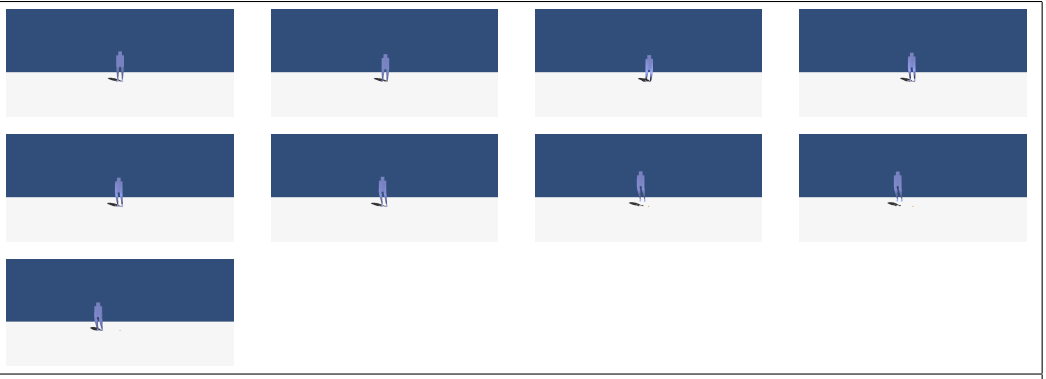
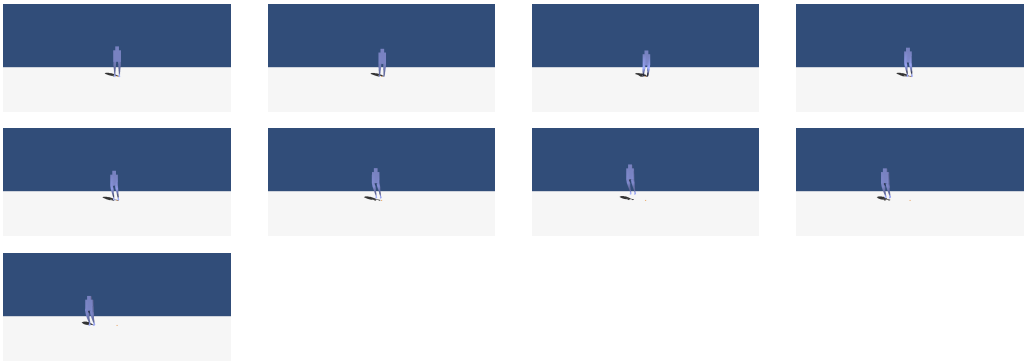
1m right	
1.3m right	
1.6m right	

Table 5.3: Table of rightward jumping motions.

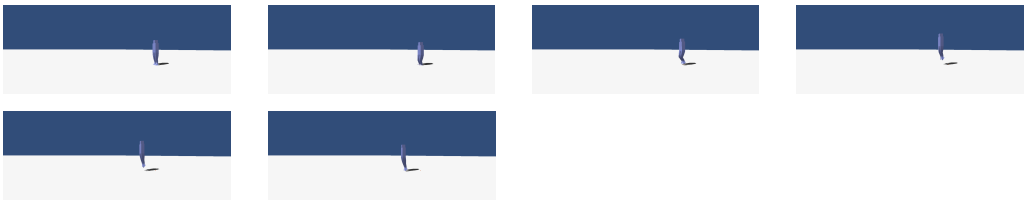
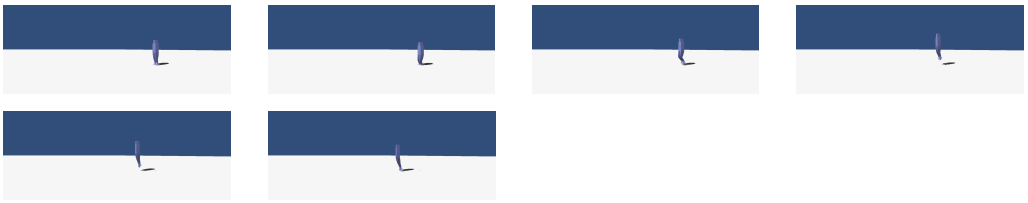
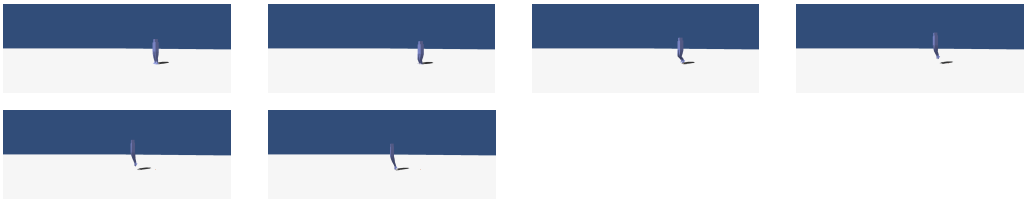
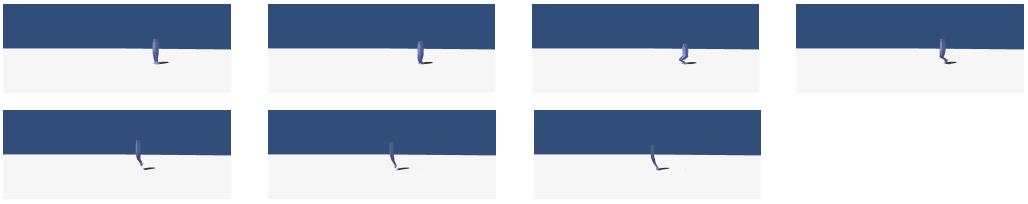
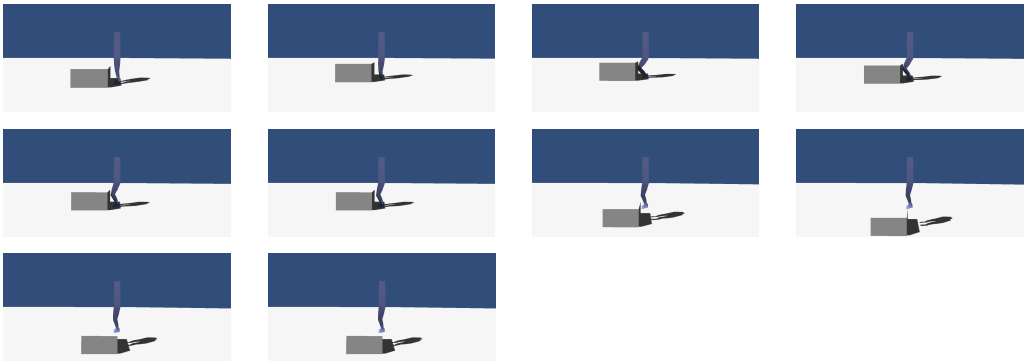
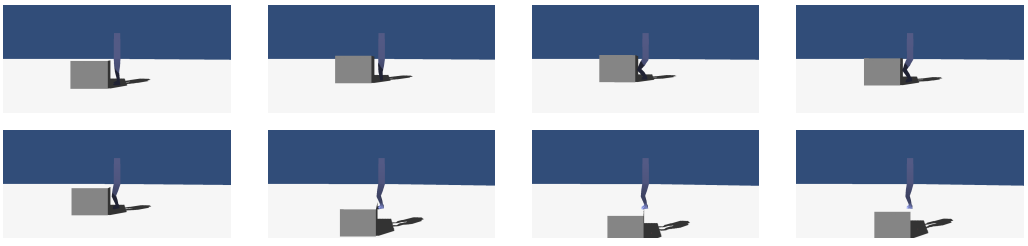
1m forward	
1.3m forward	
1.6m forward	
1.9m forward	
0.5m box	
0.75m box	

Table 5.4: Table of forward jumping motions

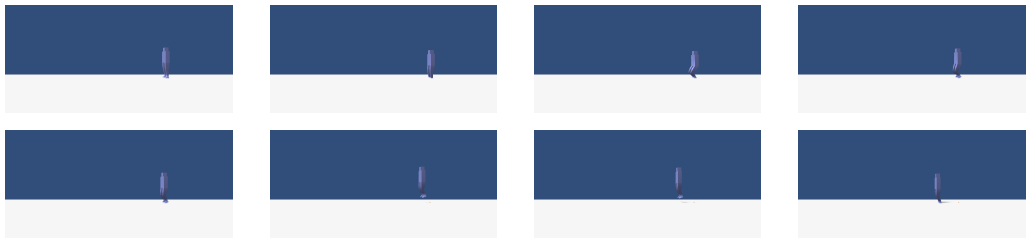
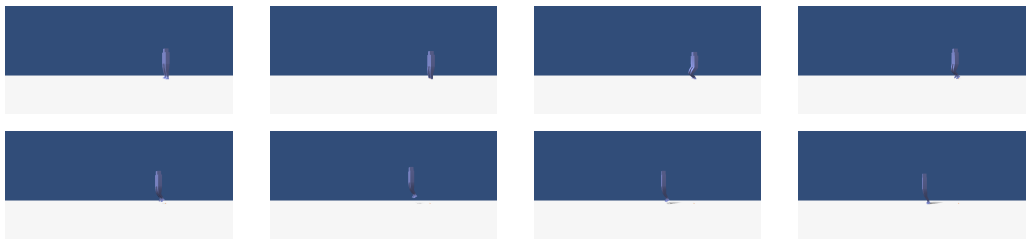
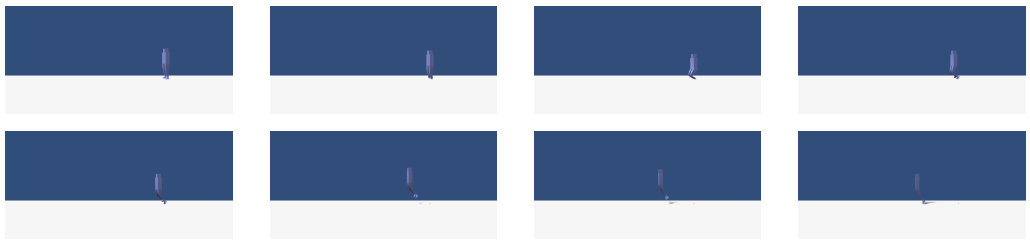
1.6m forward (global $k = 20000$)	
1.9m forward (global $k = 20000$)	
1.6m forward (varying $k = 20000$)	
1.9m forward (varying $k = 20000$)	

Table 5.5: Table of forward jumping motions

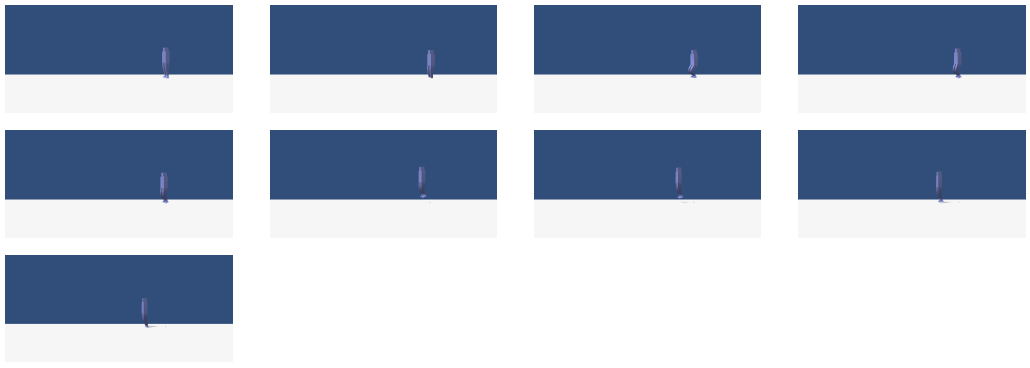
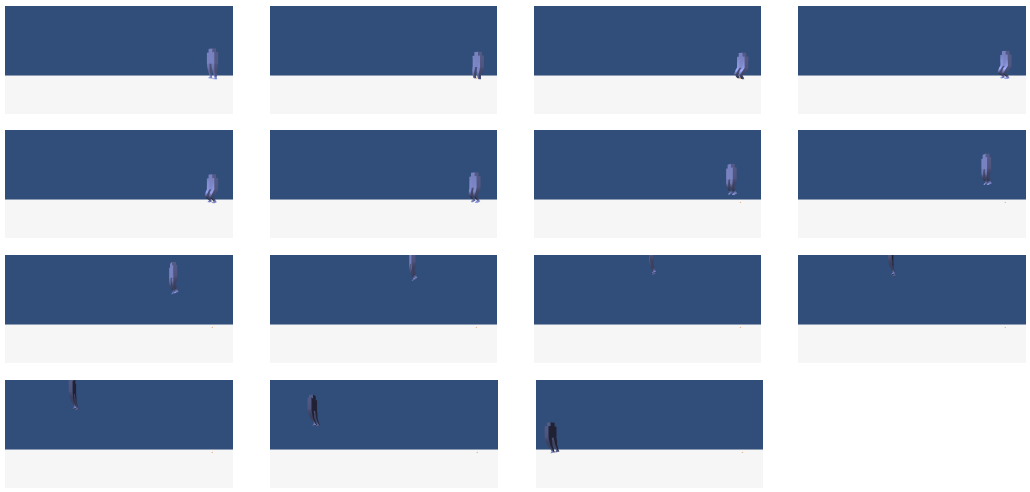
1m forward	
10m forward	

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.

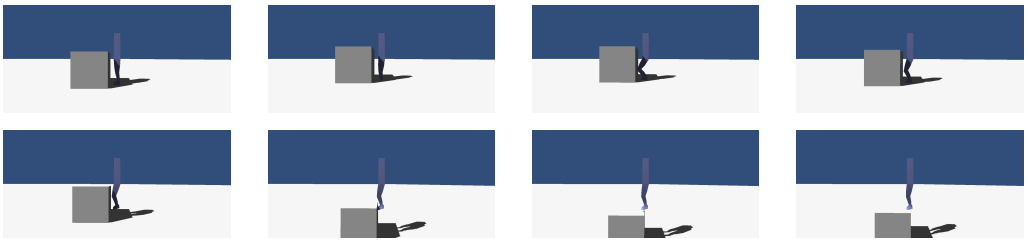
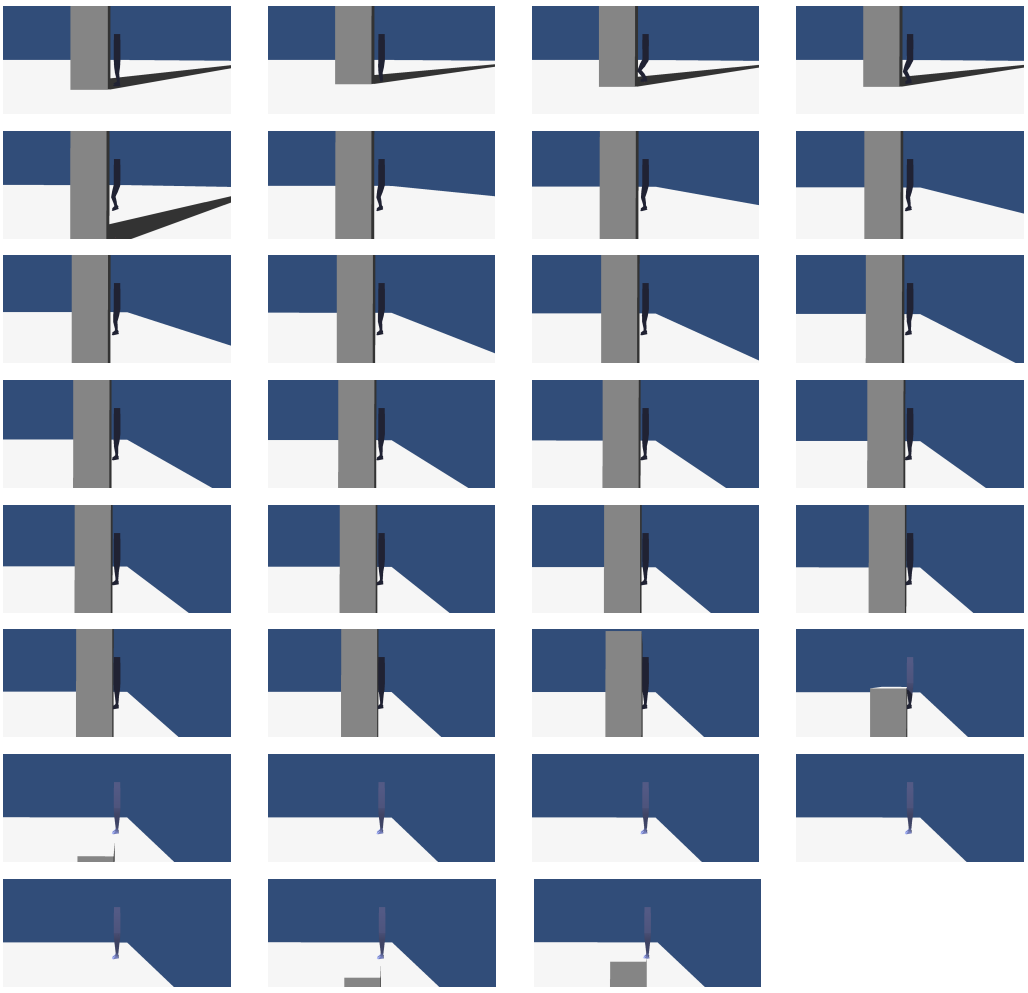
1m box	
100m box	

Table 5.7: Animations of 1m and 100m box jumps for the super human.

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

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 keyframe animation in a format such as FBX, which could be utilized in video games. A 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.

CHAPTER 6
FUTURE WORK AND CONCLUSION

- learning model - slight adjustments to different situations can be more easily generated through the use of a learning model such as in XYZ [1, 3].
- upper body affects, non-negligible effect of upper body movements on acceleration of CoM
- expand to non-bipedal
- more complex muscle and motion model, trials to determine level of effect on simulation (is it worthwhile)
- in-air phase: add complexity, can we jump to create vaulting motions, mid-jump push off from objects (jump and push off a wall), rotations, or flips.

6.1 Conclusion

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] 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

[3] 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.

[4] 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

[5] 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

[6] 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

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