Fall 2014

Self-collision avoidance through keyframe interpolation and optimization-based posture prediction

Richard Kennedy Degenhardt III
University of Iowa

SELF-COLLISION AVOIDANCE THROUGH KEYFRAME INTERPOLATION AND OPTIMIZATION-BASED POSTURE PREDICTION

By

Richard Kennedy Degenhardt III

A thesis submitted in partial fulfillment
of the requirements for the Master of
Science degree in Electrical and Computer
Engineering in the Graduate College of
The University of Iowa

December 2014

Thesis Supervisor: Professor Karim Abdel-Malek

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

_____

MASTER'S THESIS

_____

This is to certify that the Master's thesis of

Richard Kennedy Degenhardt III

has been approved by the Examining Committee for
the thesis requirement for the Master of Science degree in
Electrical and Computer Engineering at the December 2014 graduation.

Thesis Committee: _____
Karim Abdel-Malek, Thesis Supervisor

_____
Jasbir Arora

_____
Guadalupe Canahuate

To my grandmother, Margaret Degenhardt,
who encouraged her children to attend an institution of higher learning,
thereby instilling a desire for knowledge within her family
for generations to come

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Professor Karim Abdel-Malek, for providing me with great opportunities in such an interesting field. I would like to thank Kimberly Farrell for her guidance and significant contributions to the implementation of this work. Further thanks to my family, co-workers, and friends, for their support and encouragement. A special thanks to Chloe Metzger, for her help with editing and invaluable moral support. Lastly, I would like to thank the members of my thesis committee: Professor Karim Abdel-Malek, Professor Jasbir Arora, and Professor Guadalupe Canahuate, for their time and feedback.

ABSTRACT

Simulating realistic human behavior on a virtual avatar presents a difficult task. Because the simulated environment does not adhere to the same scientific principles that we do in the existent world, the avatar becomes capable of achieving infeasible postures. In an attempt to obtain realistic human simulation, real world constraints are imposed onto the non-sentient being. One such constraint, and the topic of this thesis, is self-collision avoidance. For the purposes of this topic, a posture will be defined solely as a collection of angles formed by each joint on the avatar. The goal of self-collision avoidance is to eliminate the formation of any posture where multiple body parts are attempting to occupy the exact same space. My work necessitates an extension of this definition to also include collision avoidance with objects attached to the body, such as a backpack or armor. In order to prevent these collisions from occurring, I have implemented an effort-based approach for correcting afflicted postures. This technique specifically pertains to postures that are sequenced together with the objective of animating the avatar. As such, the animation's coherence and defining characteristics must be preserved. My approach to this problem is unique in that it strategically blends the concept of keyframe interpolation with an optimization-based strategy for posture prediction. Although there has been considerable work done with methods for keyframe interpolation, there has been minimal progress towards integrating a realistic collision response strategy. Additionally, I will test this optimization-based approach through the use of a complex kinematic human model and investigate the use of the results as input to an existing dynamic motion prediction system.

PUBLIC ABSTRACT

The research presented in this thesis provides a strategy for implementing realistic self-collision avoidance on a virtual avatar. The goal of this self-collision avoidance method is to prevent multiple parts of the body from occupying the exact same space. This method not only accounts for limbs of the body, but is also dynamic enough to account for clothing and equipment that is placed on the avatar. This strategy can be of great use when animating any articulated figure in a virtual environment and is targeted specifically for the fields of animation and digital human modeling.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EQUATIONS

CHAPTER 1

INTRODUCTION

1.1 Digital Human Modeling with Santos®

Digital human modeling (DHM) is a broad term often used to describe the analysis of complex situations using the software representation of humans. This term can be applied both in the context of DHM as a technology and as a fundamental research area (Zhang and Chaffin, 2005). As a technology, DHM is a means to provide useful information or feedback through the use of a digital prototype. As a research area, DHM predicts human behavior through the development of mathematical models that allow for computer graphic visualization.

The primary benefit of DHM is its proactive approach to the human element of design. For example, the implementation of DHM technology in the field of product design can allow for easier and earlier identification of ergonomics-related problems. Often, DHM technology can even eliminate the need for added design steps, like creating physical mock-ups or testing on real human subjects. In this thesis, DHM is implemented through the virtual human Santos® with the goal of analyzing task-based human performance. To help achieve this goal, the Santos® environment relies on a novel, optimization-based philosophy of human modeling (Abdel-Malek, Yang, Kim, Marler, Beck, Swan, Frey-Law, Mathai, Murphy, Rahmatallah, and Arora, 2007). This approach empowers the digital human to perform, unaided, in a physics-based world.

Santos® was developed to be a comprehensive human model capable of predicting dynamic human motion. The digital models are based on human body scans of specifically targeted anthropometry and morphology and incorporate 109 degrees of freedom. The mathematical model for the Santos® skeleton was developed based on the Denavit-Hartenberg method for kinematic and dynamic analysis. The prediction is achieved using optimization formulations that are governed by human performance measures and constrained through restrictions imposed by the skeleton, the laws of physics, and the environment.

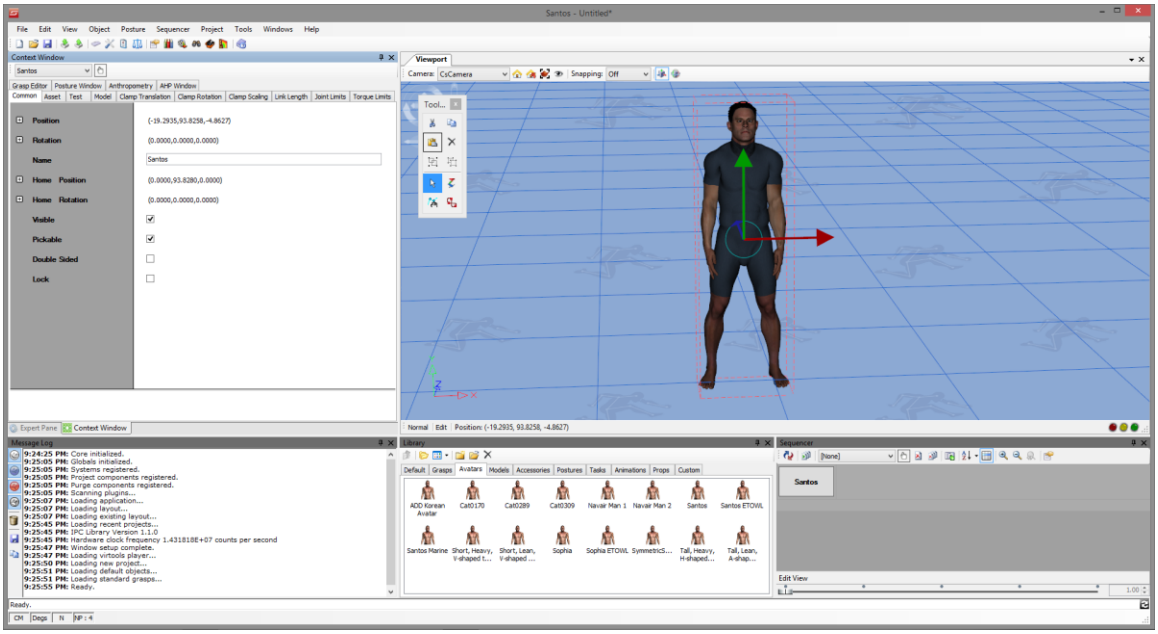

Figure 1.1: The Santos® digital human modeling software environment.

1.2 Motivation

There is an ever-evolving demand across a variety of industries for the ability to evaluate the human aspects of design within a computer-simulated environment. While Santos® offers a realistic human model with accurate posturing capabilities, it can also necessitate extensive motion capture data when trying to dynamically predict motion. This becomes especially important when trying to predict dynamic human motion for a single task with a variety of different equipment configurations. Currently, the Santos® model is being used within a branch of the US Military to prototype equipment and analyze performance across a range of tasks.

Figure 1.2: Several of the tasks that are being displayed through the use of Santos®.

A simple example of task-based prediction would be a digital soldier that walks up a flight of stairs with a specific type of rifle, armor vest, or backpack. The software enables the user to import digital equipment models, which can then be added to the virtual soldier. Each task that can be analyzed within the software requires motion capture data as input to the motion prediction system. The motion capture process does

not take into account every possible gear configuration that a soldier can have. Instead, it relies on the prediction system to provide the cause and effect of different configurations. While this method works for altering the motion to account for added forces on the body, it fails to consider how the added items can physically obstruct the motion. If, for example, an ammo pouch were to be attached to one side of the torso, only the weight of the pouch would be accounted for when dynamically predicting motion, not the space it occupies. If the avatar's arm were to move through this occupied space in the motion capture animation, it would likely move through the same space in the resulting predicted motion, causing unrealistic self-collision.

Figure 1.3: A case where the predicted motion contained self-collision.

It is infeasible to supply a motion capture animation for each of the thousands of possible equipment configurations that the software can create, and doing so would greatly diminish the dynamic nature of the software. After all, one of the advantages of digital human modeling is that it eliminates the need for the real human subjects that

motion capture necessitates. One possible solution is to require only one base animation for each task, and then modify it to account for any physical obstructions on the body. If this were done in such a way that the realism of the motion could be maintained, the dynamic nature of the software would be preserved. In order for this to be useful, the base animation for each task would need to be modified so that all collisions with equipment are avoided without compromising the objective of the motion. Thus, if the avatar is holding a two-handed weapon in the original motion capture, its hands must also remain in contact with the weapon during all further posture adjustments. To enforce objectives like this, task-specific constraints would be needed within the motion prediction to prevent any unwanted behaviors.

1.3 Literature Review

1.3.1 Keyframe Interpolation

Keyframe interpolation has been used by animation studios since the 1930s. Animators define a motion as a disjointed set of poses and rely on interpolation to fill in the frames between each pose (Nebel, 1999). In this case, a pose defined by the animator represents a keyframe. This same method is also frequently applied to articulated figures in 3D animation. The primary drawback to this approach is that it doesn't ensure that the postures created between keyframes are realistic, meaning that the animator still has to go back and check each interpolated frame to correct any unrealistic postures. Although other motion control methods exist within animation (Thalmann, 1991), keyframe interpolation still remains popular due to the simplicity of its requirements.

To reduce the "check and correct" burden that keyframe interpolation places on the animator, several different posture-building tools have been created. Kinematics and dynamic constraints are often used within virtual reality applications to create more realistic motion. Although these same strategies can be successfully applied to the field of 3D animation, they do not account for collisions between the limbs of an articulated figure. Nebel (1999) proposes an incomplete algorithm to automatically generate keyframes that the animator previously had to add manually. This algorithm does not make any assumptions towards the makeup of the articulated figure, so only a rigid skeleton and an associated collision detection system is needed. Figure 1.4 provides a better understanding of how this recursive method for collision avoidance works.

K_0, K_1, keyframes specified at the time steps T_0, T_1

CheckAndCorrect((K_0, T_0), (K_1, T_1))
{
    Interpolate between K_0 and K_1
    Get for each time step between T_0 and T_1 the list of self-collisions
    If (self-collision)
        Select the first collision to be corrected ($T_{Collision}$, ColType)
        Move the limbs at $T_{Collision}$ in order to correct the collision
        Create a sub-keyframe ($K_{Collision}$, $T_{Collision}$)
        CheckAndCorrect((K_0, T_0), ($K_{Collision}$, $T_{Collision}$))
        CheckAndCorrect(($K_{Collision}$, $T_{Collision}$), (K_1, T_1))
}

Figure 1.4: Produces collision free motion between two keyframes (Nebel, 1999).

This algorithm supplies a general strategy for the implementation of self-collision avoidance. It is an empty scheme that allows for the use of any generic interpolation and collision detection methods. The algorithm takes advantage of geometric properties to automatically fix some of the collision frames, and will continue until all frames have

been freed of self-collision. Along with implementing his own self-collision detection method, Nebel (1999) also asserts a plane constraint on the limbs of the articulated figure to ensure that the obstacle is passed. This constraint keeps the new suggested positions of the limbs on the path of the interpolation curve to create a more fluid motion. Figure 1.5 shows how the positions of a limb can be constrained to a plane that is defined by the interpolation splines.

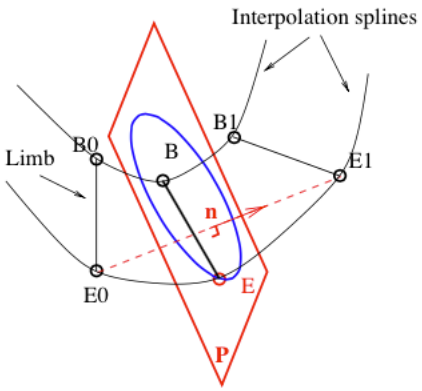

Figure 1.5: New positions generated by the plane constraint (Nebel, 1999).

Other models have since been adapted to this method of keyframe interpolation (Abend, 1982; Flash and Hogan, 1985). Nebel (2000) attempts to extrapolate models from neuroscience to obtain more realistic self-collision avoidance. These models make an effort to explain how the central nervous system coordinates movements for multi-joint limbs. One model shows that hand speed profiles have peaks before and after passing an obstacle (Abend, 1982). Another proposes a minimum jerk model that simulates point-to-point movements based on the minimization of the rate of change of hand acceleration (Flash and Hogan, 1985). These models were used within the recursive "check and correct" algorithm and were validated for simple obstacle avoidance cases

(Nebel, 2000). When compared to motions performed by real humans reaching around a simple obstacle, the results were considered to match over 30% of the time.

Liu and Cohen (1995) also proposed a model of keyframe interpolation that attempts to maintain a physically plausible motion. Their method, which they refer to as keyframe optimization, helps to fill the gap between simple keyframe systems and other optimization-based systems. This approach considers all user specified keyframe positions to be final, and it aids the interpolation process through the use of higher level constraints. These constraints primarily deal with velocity and specifically pertain to the limbs of the body and the center of gravity. The final component is a simplified optimization process that is used to solve the keyframes. This is particularly useful when a character needs to perform an athletic motion and inertial forces start playing a significant role. Although the authors do not explicitly state the results of this process, they do assert that the method creates graceful and realistic animations. It is also worth noting that this system does not incorporate contact or collision avoidance.

1.3.2 Inverse Kinematics

The use of inverse kinematics (IK) methods for simulating complex human interactions has become standard in virtual reality applications. Collision avoidance within IK is often accomplished by integrating collision response vectors. This approach tends to work on a frame-by-frame basis, though it will often fail to ensure a coherent motion (Nebel, 2000). An alternative approach would be a complete search in configuration space for a path free of collisions. This method would have an exponential

cost search that would grow with the number of degrees of freedom and still require some form of naturalness be imposed on the motion (Zhao, Liu, and Badler, 2005).

Zhao et al (2005) propose a method of sternum-rooted IK that uses a data-driven approach to path planning for upper torso avoidance. Data-driven motion generation relies on a database of human postures, and results in motions that are often very natural. The issue with this approach, however, is that it heavily depends on the amount of data collected. It should be noted that there are methods for improving the flexibility of the data set through motion interpolation (Rose, Bodenheimer, and Cohen, 1998). Zhao et al (2005) use the data-driven approach combined with a strength model to reduce the search complexity for a path and impose naturalness onto the motion. Even though this strategy works well for simple path planning where a natural connecting path exists, it still does not guarantee completeness and is not recommended for use with systems that require a more complex solution.

1.3.3 Collision Detection

The literature presented thus far has primarily focused on the problem of formulating a collision response. However, a collision handling system must also solve the problem of collision detection. Most collision detection algorithms rely on testing geometrical intersections to determine if a collision is present. These geometries are often broken into hierarchical representations of a character in order to localize the areas where a collision occurs and to improve the efficiency of the algorithm (O'Sullivan, Radach, and Collins, 1999).

One representation of such an algorithm is the use of sphere trees (Palmer and Grimsdale, 1995; Hubbard, 1995, 1996; Quinlan, 1994). It is common practice within computer graphics to use spheres to approximate objects. This is because spheres provide simple intersection tests and have the property of rotational invariance. Using sphere hierarchies, any non-convex object can be filled with spheres to provide varying degrees of approximation. O'Sullivan and Dingliana (1999) use four levels of sphere filling, each level providing a closer and more defined approximation of the object. These four levels can be collectively referred to as a sphere tree, which is generated automatically for an object. The goal of this multi-level approach is to start with the least-defined approximation and then eliminate the need to search the subsets contained within the next levels.

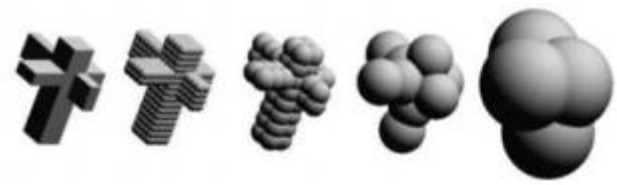

Figure 1.6: The multi-level approach to sphere filling (O'Sullivan and Dingliana, 1999).

O'Sullivan and Dingliana (1999) generate the sphere representation of an object through recursive octree subdivision. This process first requires the smallest possible bounding cube for the object to be obtained. Once the object is completely encompassed by this bounding cube, the cube is then subdivided into eight equal partitions known as octants. If a resulting partition contains any part of the object, it becomes a node of the octree and will also be subdivided. This octant subdivision continues recursively until

the desired level of approximation is reached. Each cube node will then be replaced with a sphere that has its radius set to the smallest value capable of completely encompassing the cube.

Once the sphere representation of an object has been obtained, it is ready to be used within the collision detection system. This system enforces an all-pairs table that tracks the bounding box of every entity in the scene. When two entities are found to be colliding based on the bounding boxes, they are added to the active collisions list. All pairs in the active collisions list are then processed by an intersection-testing algorithm to determine whether or not they are actually colliding. This intersection test simply checks to see if the distance between two spheres is less than the sum of the radii. If two entities are found to be colliding, they are added to the real collision list and will be handled by a collision response strategy. O'Sullivan and Dingliana (1999) use this collision detection method to provide varying degrees of collision response data that is cleverly prioritized by visual perception. They note, however, that the model is very specific and doesn't take advantage of several factors that are truly representative of human motion.

1.4 Objectives

The general objective of this thesis is to introduce a new technique for implementing self-collision avoidance within the dynamic motion prediction system of Santos®. In order to do so, a method for keyframe interpolation is developed that can be adapted to take advantage of the pre-existing work done within the Santos environment. Because a validated, optimization-based method for posture prediction already exists

within the software, this can be leveraged to impose naturalness onto the interpolated postures that are created. This method for posture prediction also allows for the creation of constraints that will be used to keep the motion continuous. More specifically, this approach will attempt to solve some of the issues encountered in the current literature through the use of the concepts outlined in the remainder of this section.

1.4.1 Keyframe Interpolation

A modified version of the generic keyframe interpolation strategy introduced by Nebel (1999) was developed to be used with a custom collision handling system. Correcting some of the animation frames through interpolation will limit the number of times the more expensive collision response method of posture prediction is needed. This attempt to confine posture prediction to frames targeted by recursive bisection will also prove to help maintain a coherent motion. Because large groups of collision frames could lead to loss of important motion characteristics, evaluation points are used to prevent the interpolation from over-smoothing the motion.

1.4.2 Sphere-Based Collision Detection

A collision detection method is implemented that uses sphere-based body groupings to distinguish between the torso and limbs of the avatar. This method is low cost and allows for a dynamic modification of the groups to account for equipment added to the body. The avatar will have a set amount of predetermined body spheres, while any added equipment will need to be filled with spheres before it can be grouped

appropriately. To automate this sphere-filling process, an equipment import tool has been developed that automatically attempts to fill a model using a best-fit strategy for a number of spheres provided by the user.

1.4.3 Collision Response through Posture Prediction

As previously discussed, the existing method for posture prediction is being leveraged to impose naturalness onto the interpolated postures. This approach, like those that use kinematics equations in the literature (Zhao, Liu, and Badler, 2005), can be problematic when trying to produce a fluid motion. To account for this issue, a plane constraint is imposed on the position of specifically targeted joints. This constraint will always be in the direction of the most recently defined motion curve. These plane constraints, combined with the smoothing effect of keyframe interpolation, mitigate the motion inconsistencies created by frame-by-frame posture prediction.

CHAPTER 2

BACKGROUND

2.1 Kinematic Human Modeling in Santos®

The Santos® software environment provides a physics-based model that can be used within a multi-objective, optimization-based problem to formulate predicted postures (Yang, Rahmatalla, Marler, Abdel-Malek, and Harrison, 2007). That is to say, a high degree-of-freedom human model is used within the software to realistically approximate a full body skeletal posture that is governed by the same factors of physics that we encounter in the real world.

2.1.1 The 55-DOF Santos® Model

The Santos® model uses rigid links connected by kinematic joints to represent the human skeletal system (Yang et al, 2007). It's important to note the difference between a kinematic joint and an anatomical joint. An anatomical joint, like those found on a real human, can have multiple kinematic joints. For example, the wrist contains two kinematic joints because it is mechanically capable of rotating about two different axes. In order to accurately model the entire human body, a system with a large number of kinematic joints is necessary. This is what is commonly referred to as a high degree-of-freedom (DOF) model. The Santos® software uses a 55-DOF kinematic model that can be seen in Figure 2.1.

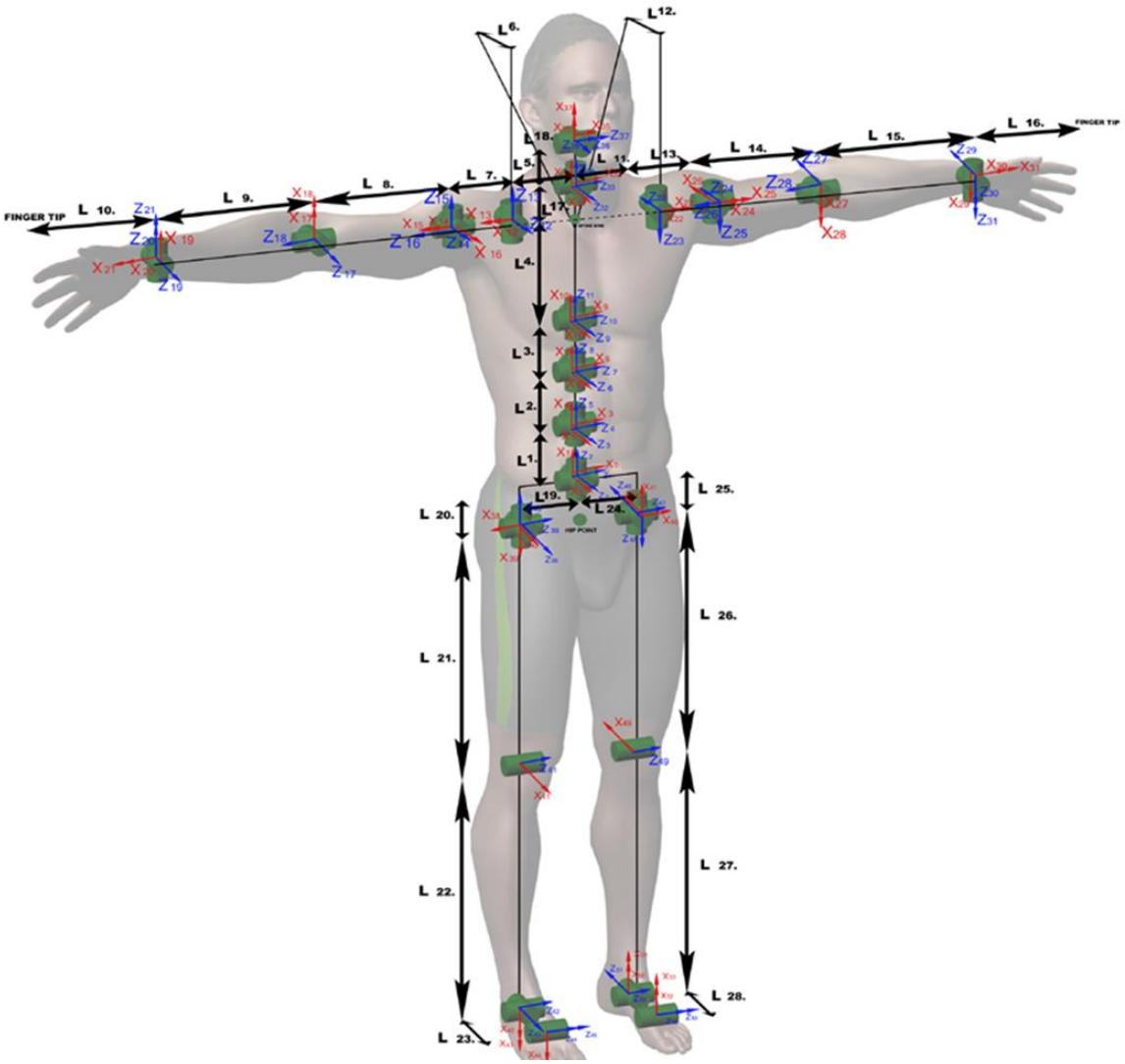

Figure 2.1: The 55-DOF Santos kinematic model.

The Denavit-Hartenberg (DH) method is used with this model to represent the transformations between joints (Denavit and Hartenberg, 1955). Because the human body is a structure that is arranged through a series of rigid links connected by joints, the DH method is necessary to address the motion of the system (Abdel-Malek and Arora, 2013). Additionally, this method has proven to be a useful tool for modeling human biomechanics, because it can efficiently represent the transformation through four parameters (Farrell, 2005). Developing the Santos® model for the DH method required a

local coordinate frame to be embedded at each DOF. For each frame, the i^{th} z-axis controls the motion for the $(i + 1)^{th}$ DOF. Farrell (2005) describes the four parameters of the method that are used to obtain the position and orientation of frame i with respect to frame $i – 1$ as follows:

- the angle θ_i between the $(i\text{-}1)^{th}$ and i^{th} x-axis about the $(i\text{-}1)^{th}$ z-axis
- the distance d_i from the $(i\text{-}1)^{th}$ to the i^{th} x-axis along the $(i\text{-}1)^{th}$ z-axis
- the angle α_i between the $(i\text{-}1)^{th}$ and i^{th} z-axis about the i^{th} x-axis
- the distance a_i from the $(i\text{-}1)^{th}$ to the i^{th} x-axis along the i^{th} x-axis

Referring to Figure 2.1, the coordinate frames required for supplying the parameters of the DH method can be seen with each DOF. It is this use of the DH method with a high-DOF human model that allows for the prediction of human posture within the software.

2.1.2 Optimization-Based Posture Prediction

Posture prediction attempts to find a configuration of joint angles that allows the human body to achieve a specific objective. A common example of this is reaching a fingertip towards a target point in space. A high-DOF model of a human will likely yield multiple solutions to this problem. An optimization-based approach to posture prediction will look for the most realistic posture that satisfies the objective. Santos® uses a multi-objective optimization, which combines multiple human performance measures as multiple objectives in the optimization problem. These performance measures can include minimization of factors like perturbation from a neutral posture, joint displacement, and potential energy. Depending on the objective, there may be a

preference for one factor over another. Figure 2.2 shows how the Santos® software can combine a variety of performance measures and assign weighting values for each.

Figure 2.2: The result of MOO-based posture prediction for the index finger acting as the end-effector and reaching a target point in space.

The possible performance measures are not limited to those shown in Figure 2.2, as many others can be developed. Optimization-based posture prediction merely introduces a framework for the optimization of these cost functions on the task being completed. A cost function, for the purpose of this formulation, is any human performance measure that is to be minimized or maximized (Abdel-Malek and Arora, 2013). In order to properly set up the optimization problem, constraints and design variables will also be needed. The design variables are the position and orientation of the segmented body links, while the constraints are any mathematical bounds to the problem. For example, a constraint can impose a distance tolerance between a target point and an

end-effector. An end-effector is the part of the body that will attempt to reach the target point. In Figure 2.2, the end-effector is the tip of the index finger, and the global position of this end-effector is obtained through Equation (2.1), expressed in terms of the transformations given by Equation (2.2).

$$\mathbf{x}(\mathbf{q}) = \left(\prod_{i=1}^{n} {}^{i-1}\mathbf{T}_i \right) \mathbf{x}_n \qquad \text{(Equation 2.1)}$$

$${}^{i-1}\mathbf{T}_i = \begin{pmatrix} \cos\theta_i & -\cos\alpha_i \sin\theta_i & \sin\alpha_i \sin\theta_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\alpha_i \cos\theta_i & -\sin\alpha_i \cos\theta_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{(Equation 2.2)}$$

Within Equation (2.1), $\mathbf{x}(\mathbf{q})$ represents the global position vector and $\mathbf{x}_n$ is the position of the end-effector with respect to the n^{th} frame. The transformation matrix describes the position and orientation of the i^{th} frame in terms of the $(i-1)^{th}$ frame. These frames are embedded locally at each DOF, and the terms of this transformation matrix are the DH parameters discussed previously.

The optimization problem is then set up to find the generalized joint variables that represent the optimum posture for a task. It attempts to satisfy the cost functions while also operating within a feasible space that is governed by the constraints. Abdel-Malek and Arora (2013) present the optimization problem using the following equations:

$$Find: \boldsymbol{q} \in R^{DOF} \quad Minimize: \textit{Discomfort, Effort,} \text{ etc.} \qquad \text{(Equation 2.3)}$$

$$Subject\ to: \left\| x(\boldsymbol{q})^{end-effector} - x^{targetpoint} \right\|^2 \leq \varepsilon \qquad \text{(Equation 2.4)}$$

$$and \quad q_i^L \leq q_i \leq q_i^U; \qquad i = 1, 2, \ldots, DOF$$

Within Equation (2.3), **q** represents the vector of generalized joint variables that will be calculated. The first part of Equation (2.4) describes the distance constraint that will be used to require contact between the end-effector and target point. This constraint represents distance squared, and ε is a positive number that is used to approximate zero. The second part of the equation is used to impose limits on the joints so that the resulting posture remains realistic while trying to reach the target point. For the purposes of this thesis, only a basic understanding of optimization-based posture prediction is necessary. The real significance, with regards to self-collision avoidance, is that it supplies a validated method for imposing realism onto a collision response.

2.1.3 Minimizing Effort

Although there are several performance measures available within optimization-based posture prediction, this implementation will primarily focus on minimizing effort. Effort will be defined as total displacement of all joints from the initial posture. This differs from the discomfort measure, which is a measure of total displacement from a posture that is deemed comfortable. For the purposes of this thesis, effort will likely be a more useful and reliable measure than discomfort, especially with a motion like jumping vertically where certain frames will not have an easily defined comfortable posture.

Equation (2.5) (Abdel-Malek and Arora, 2013) provides a measure of effort that can be minimized within posture prediction. Here, $q_i^{initial}$ represents the initial set of joint variables for the avatar prior to posture prediction. The effort is determined by the summation of the distances from the initial set, where each joint is assigned a weighting value (γ_i). The weighting values are an important component for adding realism to the prediction, as they provide a priority for joint articulation. The assigning of higher values will result in more contribution to the sum and have a stronger effect on the prediction. Put simply, this will keep joints of higher weight values closer to the initial posture, with the weight values for the joints having already been previously determined (Farrell and Marler, 2004).

$$f_{effort}(q) = \sum_{i=1}^{n} \gamma_i \left(q_i - q_i^{initial}\right)^2 \qquad \text{(Equation 2.5)}$$

Because this strategy is trying to enforce the initial posture, it is beneficial to have a reasonable and realistic initial set of joint variables. If, for example, the initial posture was obtained through motion capture, this method would be working towards a more realistic set of joint variables. When used on a frame-by-frame basis, as demonstrated in this thesis, it will also help posture prediction follow the objectives of a motion.

2.1.4 Validation of the Santos® Model

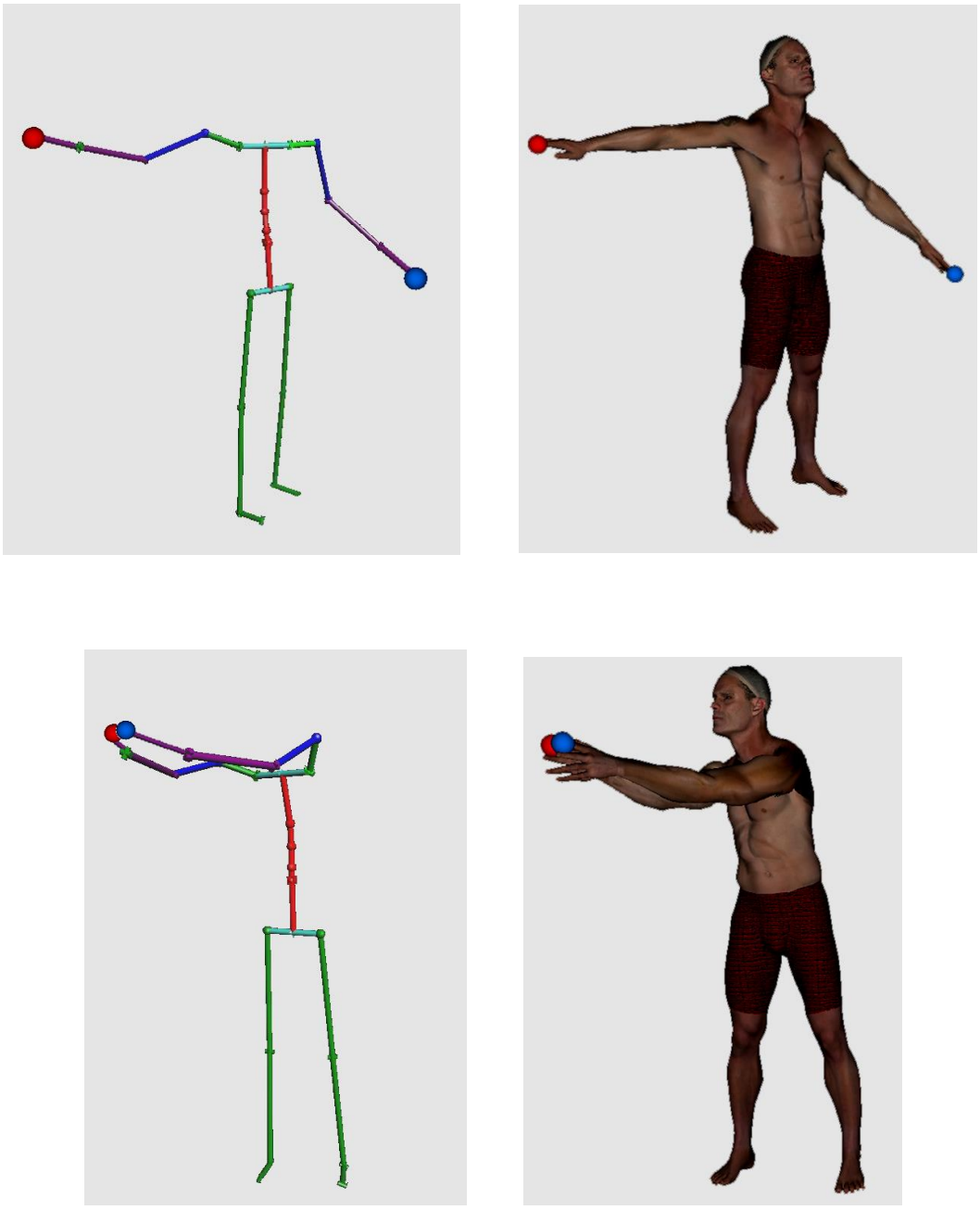

Figure 2.3: Some of the validation results for optimization-based posture prediction. The images on the left side are the motion capture results while the images on the right side are the results of posture prediction (Farrell, 2005).

In order to validate the results of optimization-based posture prediction with a high-DOF kinematic model, a motion capture tracking system was used with real human subjects (Farrell, 2005). The motions of these subjects were mapped to a skeletal model and recorded to a results file. After post-processing the files to account for subject anthropometry, they can be used to directly compare motion on a high-DOF skeletal model. The results of this validation (Figure 2.3) showed that optimization-based posture prediction is capable of providing realistic postures. Although the solution provided by posture prediction depends on the performance measures chosen in the optimization, the validation showed that this approach allows those measures to be easily adjusted and adopted to models with multiple end-effectors.

2.2 Predictive Dynamics

Predictive dynamics is the approach used within Santos® to simulate human motion. Although the self-collision avoidance strategy described within this thesis does not make use of predictive dynamics, it will provide a form of input to the system. For this reason, it is useful to have a basic understanding of the predictive dynamics methodology.

Considerations like human dynamics and the laws of physics are used within predictive dynamics to provide realistic human motion (Abdel-Malek and Arora, 2013). Similar to optimization-based posture prediction, the predictive dynamics approach relies on the same basic components in order to set up the optimization formulation. The primary difference with dynamics is that it defines the performance measures and

constraints with the goal of recovering the real motion of a dynamic system (Xiang, Chung, Kim, Bhatt, Rahmatalla, Marler, Arora, and Abdel-Malek, 2010). Nevertheless, the problem still consists of the same core components: design variables, cost functions, and constraints.

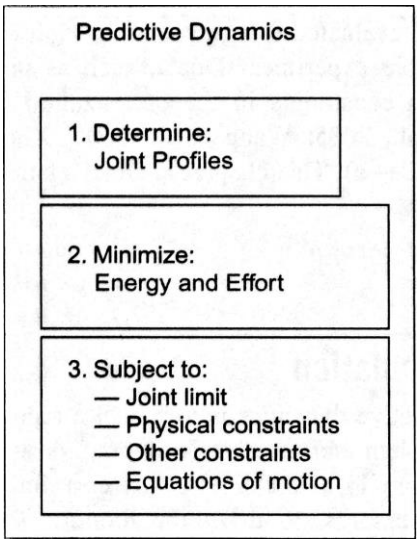

Figure 2.4: A component view of the predictive dynamics problem (Abdel-Malek and Arora, 2013).

The cost functions used within predictive dynamics are similar to those used with optimization-based posture prediction. The goal of these functions is to drive the motion prediction based on minimizing factors, like energy and effort. This minimization must still adhere to all constraints specified for the particular motion. The distinguishing constraint of predictive dynamics is that it incorporates the general equations of motion. These equations describe the dynamics of our world, and attempt to solve for an unknown motion and the forces behind it. As shown in Figure 2.4, the goal is to determine the joint profiles. These joint profiles represent the angles created by the body segments as a function of time (Abdel-Malek and Arora, 2013).

One of the posture prediction performance measures, which was previously mentioned, related to the minimization of deviation from an initial reference posture (Zhang and Chaffin, 2005). A similar approach can be taken with predictive dynamics, but with a reference motion instead of a reference posture. Because the reference motion can be obtained through motion capture with real human subjects, it serves as a valuable performance measure for providing a realistic solution. It is important to note that this is a cost function and must satisfy the constraints imposed by the joint limits and the equations of motion.

Although predictive dynamics is capable of incorporating a method for self-avoidance, the current implementation offers no way to avoid obstacles that are attached to the body. From a software standpoint, the ability to attach objects to the body and use dynamics to predict the motion with these new forces taken into consideration is useful. While the system is very capable of supplying realistic results with regards to the effect of the new forces, it does not account for the space occupied by the objects. By using a reference motion as a cost function, the issue is further compounded by minimizing the deviation from a motion where the added obstacles do not exist. One possible solution to this problem, which is the topic of this thesis, is to modify the reference motion to account for the obstacles that have been added to the body. The resulting motion should provide a more useful minimization to be used as a cost function. However, there are still a few potential pitfalls with this strategy, which will be discussed further in the next chapter.

CHAPTER 3

APPROACH

3.1 Integration into Predictive Dynamics

The goal of this work is to provide the predictive dynamics system with a motion that is free of self-collision. As discussed in the previous chapter, this motion will be used as part of a cost function within the system. Unfortunately, this process is not as straightforward as just fixing the self-collisions in a motion capture file and then passing it to the predictive dynamics system. Because the system will modify the motion to account for the forces on the body, it is entirely possible that the fixed motion will be brought back into collision.

If, for example, a situation calls for the use of predictive dynamics for simulating a task where a human model is walking forward with a backpack and an armor vest. Although a motion capture file of a test subject walking forward is available for use as the reference motion, the action was performed by a subject that was not wearing a backpack or an armor vest. Because the reference motion is part of only one of potentially many cost functions within the system, and the forces of the objects will still be applied through dynamics, the motion remains acceptable to use. The problem, however, is that the vest in the simulation has pockets protruding from the sides and they collide with the model's arms while walking. To fix this issue, the reference motion capture file will be run through the self-avoidance system to transform it into a motion that is free of these collisions, allowing the model's arms to cleanly navigate around the

protrusions of the vest. This new, and hopefully improved, reference motion can now be used within predictive dynamics. If, however, the force on the body created by the backpack is great enough to cause significant deviation from the reference motion, it is likely that the arms will again be moved into collision with the vest protrusions.

There is a simple, yet costly, solution to this problem. First, predictive dynamics will need to be used with the unmodified reference motion in order to obtain a result where the forces have already been applied. This resulting motion will then be run through the self-avoidance system to fix any collisions. Because the method may have altered the motion in such a way that violates the predictive dynamics constraints, it would need to be fed back into predictive dynamics again and used to obtain the final solution. Running the entire predictive dynamics system twice is often too costly in terms of performance for practical uses, especially for the purposes of self-collision avoidance.

Fortunately, this implementation can make use of an existing neural network that is built around a vast library of predictive dynamics results (Bataineh, 2012). This neural network was created with the goal of improving the performance of predictive dynamics by providing a better initial guess to the system. It can also improve self-avoidance by providing an initial guess motion that accounts for added forces to the body, and thereby necessitates only running predictive dynamics once. Although this method still leaves room for error, the performance of the motion prediction system as a whole is greatly increased. To recap, the self-collision avoidance system takes the output of the neural

network, removes all self-collisions from the motion, and passes it onto the system to be used for motion prediction.

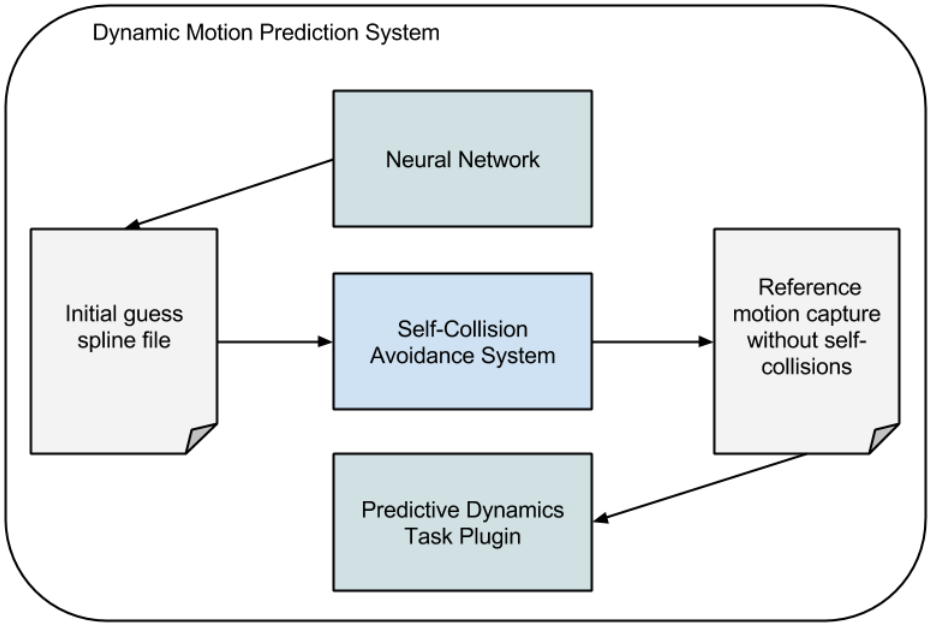

Figure 3.1: A component view of the dynamic motion prediction system.

3.2 Keyframe Interpolation with Posture Prediction

The previous section focused on how the self-collision avoidance system is used to aid the motion prediction of predictive dynamics. This section, and those that follow, will deal only with what happens within the self-collision avoidance system. The basics of the algorithm will be discussed along with a detailed breakdown of the collision avoidance strategy and the constraints created for posture prediction.

3.2.1 Recursive Bisection with Interpolation

The idea of keyframe interpolation through recursive bisection was presented by Nebel (1999). As shown in the literature review, the general structure already exists for a recursive algorithm that implements this form of self-avoidance. Using this underlying idea, a method for keyframe interpolation was developed that works around joint angle sets obtained from optimization-based posture prediction. In order to manipulate the animation, the input file is converted into a series of joint rotation curves. Each degree-of-freedom present on the avatar contains its own rotation curve for the animation, and each curve contains a key for every frame—which is represented by a single posture—in the animation.

Figure 3.2 provides a basic outline of the algorithm used to generate self-collision free motions. This method takes the bounds of an animation segment as input, and finds all collision groups within this segment. A collision group occurs when multiple consecutive frames are found to contain self-collisions. Once all of the collision groups are obtained, the recursive loop begins. For each collision group, the centermost frame is found and run through a modified version of optimization-based posture prediction that will account for self-avoidance. The keys for all surrounding frames within the joint rotation curves are then removed and replaced by interpolated values. Because the animations maintain a high frame rate, these curve keys can be accurately evaluated through linear interpolation (Equation 3.1). The only frame known to be free of collision at this point is the bisecting frame, though it is likely that several other surrounding

frames have also been fixed as a result of the interpolation. This entire process continues until the motion has been modified such that no self-collisions remain.

For a value x in the interval (x_0, x_1), the value y along the straight line is given from:

$$\frac{y-y_0}{x-x_0} = \frac{y_1-y_0}{x_1-x_0} \qquad \text{(Equation 3.1)}$$

```
RecursiveAvoid( startFrame, endFrame )
{
    allGroups = FindAllCollisionGroups( startFrame, endFrame )

    for each collisionGroup in allGroups
    {
        bisectFrame = the midpoint frame that bisects the group
        for each frame in collisionGroup
        {
            if the frame is the bisectFrame
            {
                RunSelfAvoidance( frame )
            }
            else
            {
                for each joint rotation curve
                {
                    remove the curve key for the frame
                }
            }
        }

        for each frame that is not bisectFrame
        {
            for each joint rotation curve
            {
                interpolate along the curve to add a key back for the frame
            }
        }

        previousFrame = the frame before the bisectFrame
        nextFrame = the frame after the bisectFrame

        RecursiveAvoid( startFrame, previousFrame )
        RecursiveAvoid( nextFrame, endFrame )
    }
}
```

Figure 3.2: The recursive method used to fix self-collisions through posture prediction and interpolation.

Because it is often difficult to visualize the motion in terms of joint rotation curves, it may be helpful to think of it in terms of a two dimensional path of motion. This

path will be for a joint on the body that will travel through an area of collision. The steps of the algorithm can now easily be broken down and displayed visually. Figure 3.3 shows the steps using this visual format for a simplified case.

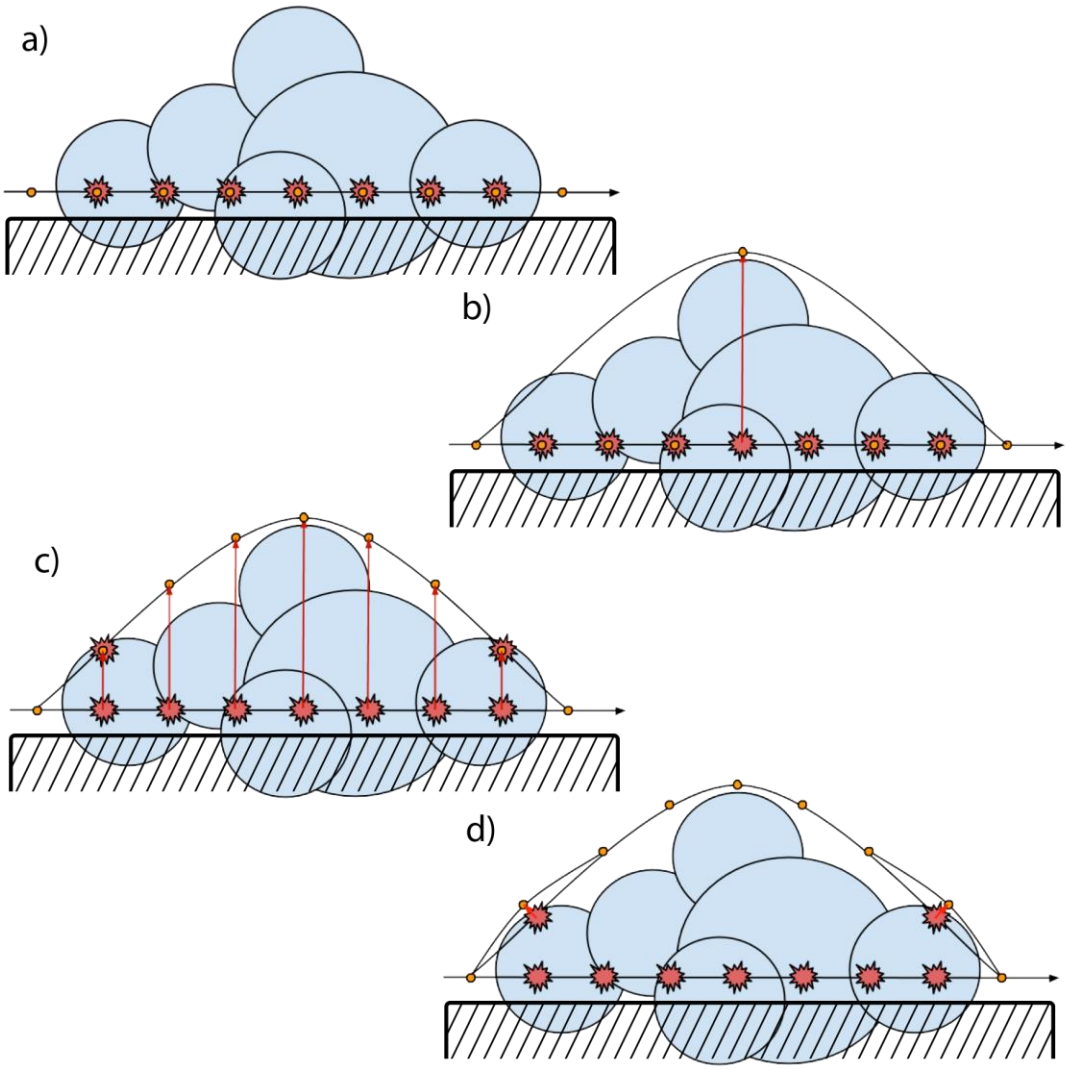

Figure 3.3: The blue circles represent collision spheres, and the yellow dots represent a joint position within a frame of the animation, where a) the path of motion creates a group of collisions, b) the middle frame is fixed through the implementation of posture prediction, c) the surrounding frames are removed and replaced by interpolated values, and d) the process recursively repeats for remaining collisions.

3.2.2 Using Evaluation Points to Prevent Over-Smoothing

Over-smoothing of an animation occurs when a large group of frames are interpolated such that characteristics of the original motion are lost. This has the potential to happen any time a large group of collisions is found, and is a result of all frames around the initial bisection having been replaced by interpolated values. While it is preferable to replace these frames with postures that are free of collision, this cannot be done if replacement comes at the cost of important characteristics of the motion. Fortunately, for the purposes of this thesis, each predictive dynamics task developer specifies evaluation points within the animation. They choose these points in time based around criteria they deem important to the representation of the motion. This is similar to how a 3D animator would choose frames for keyframe interpolation.

These evaluation points are very useful for identifying a relatively small selection of frames where the values in between can be interpolated without losing important motion characteristics. In other words, the likelihood of the motion being over-smoothed by interpolation can be mitigated by using this selection of frames. This is done by implementing a quick pre-processing phase prior to the recursive bisection. Within this phase, both the group of developer-chosen frames and the first and last frame of the animation are run through self-collision detection. If a self-collision is detected for any frame, it will be fixed using the modified optimization-based posture prediction method. This will ensure that the collision groups within the animation never get large enough to allow interpolation to over-smooth desirable characteristics.

3.3 Collision Detection

The collision response method chosen for this self-avoidance approach is best suited for a simple and low cost strategy for collision detection. Because the collision response does not require information about the current collisions, it is only necessary to know if they exist within the frame. As the literature mentioned, sphere-based intersection tests provide a low cost option for collision detection. The method implemented here will use body-based sphere groups and dynamically add object spheres to the group of the parented limb.

3.3.1 Body-Based Sphere Groups

A list of self-avoidance spheres and a grouping logic for those spheres exist within the data file for each avatar. The five basic body groups that will be considered include the torso and head, the left arm, the right arm, the left leg, and the right leg. Each sphere is parented to a joint and assigned a radius and offset position. In total, the avatar contains 39 body spheres split among the five groups. The file also contains logic for several spheres pairs that are not enforced between the groups, most notably the overlap where the legs and arms meet the torso.

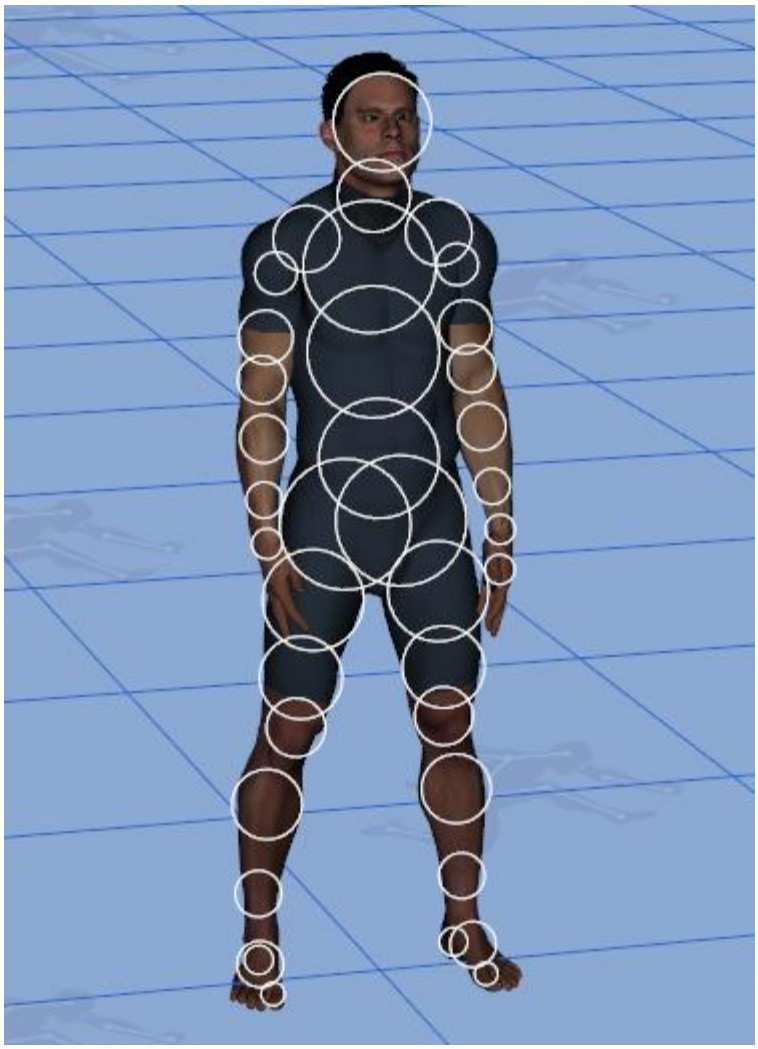

Figure 3.4: The self-avoidance body spheres of the avatar.

Once all of the spheres have been defined for the body, the self-collision detection test becomes fairly simple. First, a list of enforced sphere pairs is built between the five groups. This is done by adding every possible pair that could qualify as a collision between any two groups. Say, for example, the objective is to know when the sphere located in the wrist collides with a sphere located in the other four body groups. This means that the list contains a pair for every possible pairing of the wrist sphere with each other sphere in the other four body groups. The collision detection then becomes a

simple intersection test between every sphere pair within this list. Because the radius and position of every sphere is known, the intersection test consists of comparing the sum of the radii of the two spheres with the distance between the centers. Essentially, if the sum of the radii is greater than the distance between the center points, it is safe to determine that the two spheres are not intersecting. Although there is a considerably large number of sphere pairs in the list, the simple intersection test ensures that performance remains acceptable.

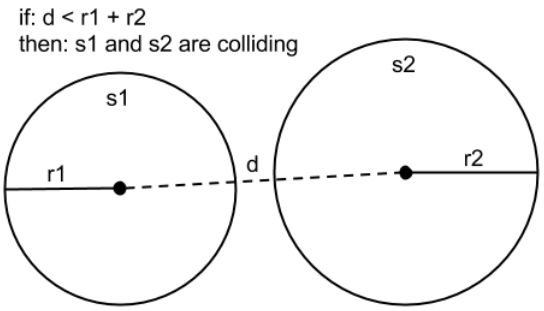

Figure 3.5: Two dimensional view of sphere intersection test.

3.3.2 Body-based Object Sphere Filling

Not only is it useful to avoid self-collision between body parts, but it is also useful to avoid collision with objects that are attached to the body. For this reason, a method has been developed for filling objects with spheres and adding those spheres to the appropriate body group. If, for example, a pouch is added to the torso of the avatar, the pouch will be filled with spheres that will then be added to the torso's body sphere group. From that point on, any time a sphere located within the arm collides with a sphere of the

pouch, it will be detected as a collision that should be handled by the collision response strategy.

A method for sphere-filling of objects has been developed in order to further support a continuous avoidance strategy. The actual sphere-filling method is not unique, but where standard approaches fill every mesh of the model individually, this method wraps all of the meshes in a cocoon-like casing which will then be filled with spheres. This takes a potentially complex shape and greatly simplifies it. One issue this form of sphere-filling can prevent deals with non-continuous motions created by avoidance of annulated objects.

Figure 3.6: A torus-shaped object parented to the avatar.

Figure 3.6 shows the type of object that could create this issue. It occurs because the optimization-based posture prediction can provide a solution that positions a limb

inside the torus-shaped object. If a motion involved swaying the arms forward and backward, posture prediction would likely alternate between predicting collision-free frames outside of the torus object with predicting collision-free frames inside of the object. This would either lead to an unrealistic snapping motion through the torus object, or a very inconvenient and inefficient swaying of the arm. Because the optimization-based posture prediction does not factor in the equations of motion, the problem would need to be simplified in order to maintain realism. By filling objects with spheres based on a convex envelope, the gaps of the object are filled, and a simpler shape for avoidance is created. By applying this method to the torus-shaped object of the previous figure, the hole becomes filled with spheres and the avatar navigates the arm around the entire object during motion. It is worth noting that these special error cases are difficult to create, so filling objects in the conventional manor would not be considered detrimental to the system.

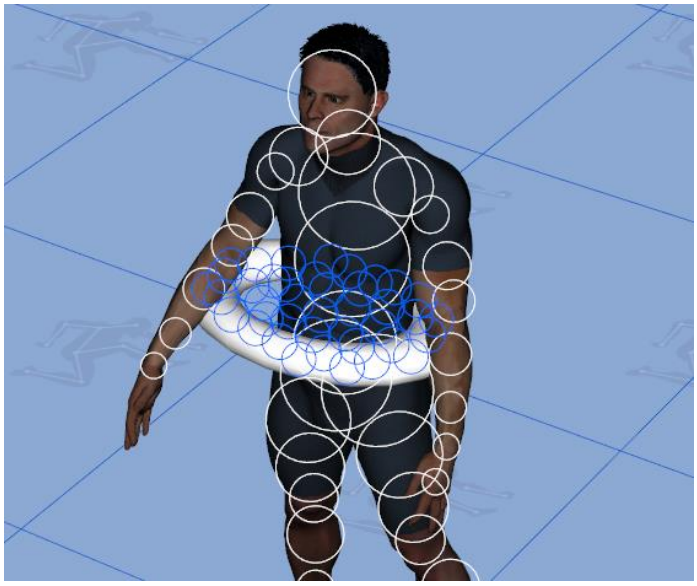

Figure 3.7: The torus-shaped object that is filled based on a convex envelope.

3.3.3 Collision Detection Implementation

The implementation of collision detection within the self-avoidance algorithm is used to find the collision groups present in an animation segment. The method only needs to know whether or not a single collision is present for a particular frame. The following figure shows the logic for building the collision groups that will are used with the recursive bisection method.

```
FindAllCollisionGroups( startFrame, endFrame )
{
    collisionGroups = the list used to store collision groups
    for each frame from startFrame to endFrame
    {
        apply the posture of the frame to the kinematic model

        if a collision is detected
        {
            if not previously colliding
            {
                recognize this point as the start of a collision group
            }
            or if the frame is the last frame in the entire animation
            and it was previously colliding
            {
                fix the frame and add the group to collisionGroups
            }
        }
        else
        {
            if previously colliding
            {
                recognize this point as the end of the collision group
                add the group to collisionGroups
            }
        }
    }
}
```

Figure 3.8: The strategy for grouping self-collisions.

3.4 Self-Avoidance Constraints

In order to maintain a realistic motion and adhere to certain task-based objectives, it is necessary to implement custom constraints for the optimization-based posture prediction. The first of which is a plane constraint for positioning joints, which will help the posture prediction maintain a fluid path of motion for the limbs. For specific tasks, other constraints will also need to be added if this method for self-avoidance is to be used practically. One such task-based objective that has been implemented is a constraint for holding a two-handed weapon while walking.

3.4.1 Plane Constraints

The need for plane constraints becomes apparent as soon as posture prediction is freely run on a frame-by-frame basis. Plane constraints are used to drive the predictions along the path of motion, and without this driving force, the motion can easily become disjointed. This occurs because the optimization problem within posture prediction knows nothing about the frames before or after the current frame, and there is nothing that enforces coherence between frame predictions. Without a plane constraint, the posture will be solved based only on minimization of effort, which will likely create large gaps between frames. Figure 3.9 gives a two-dimensional representation of the problem through the same simplified visualization method used previously.

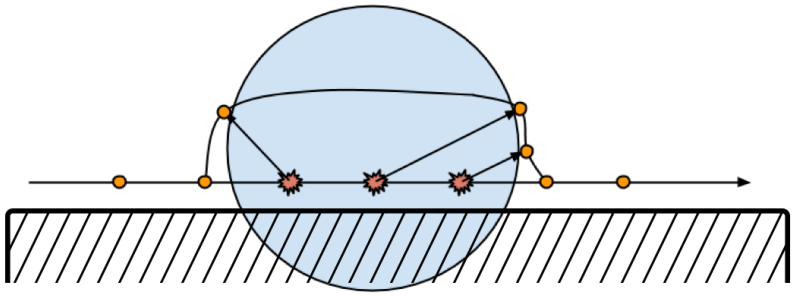

Figure 3.9: How posture prediction could potentially fix all frames of the motion without actually fixing the path of the motion.

Although interpolation will likely mitigate this effect by evenly evaluating some of the collisions frames, a method for enforcing the path of motion is needed. A joint-based plane constraint has been implemented by placing an unbounded plane in the position of a selected joint and setting the orientation of the plane to be the direction of motion (Equation 3.2). Next, posture prediction is run with the position of the selected joint constrained to somewhere along the plane. The joint will be the end-effector in this instance, and the distance between it and the plane will be minimized through Equation (3.3), as demonstrated by the accompanying figure (Farrell, 2005). To allow for a simpler optimization, a dot product representation of $\cos^2\theta$ is used.

$$n_i = |x_i(q) - x_{i+1}(q)| \qquad \text{(Equation 3.2)}$$

Where, $x_i(q)$ is the joint position at the i^{th} frame

$$Distance^2 = |x_k(q) - p_k|^2 \cos^2\theta \leq \varepsilon \qquad \text{(Equation 3.3)}$$

$$= |x_k(q) - p_k|^2 \left(\frac{n_k \cdot (x_k(q) - p_k)}{|n_k||x_k(q) - p_k|}\right)^2 \leq \varepsilon$$

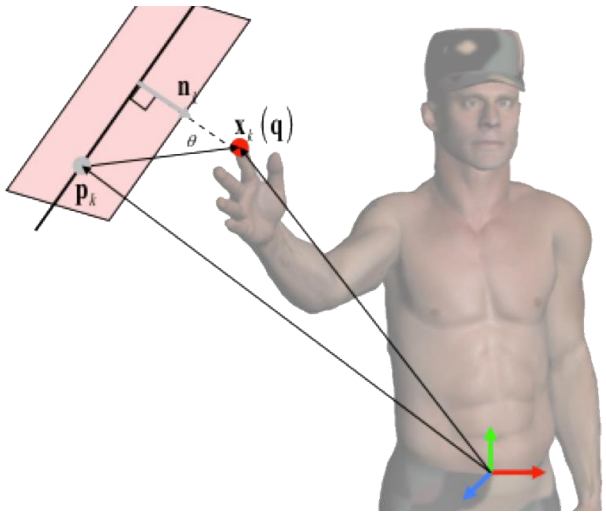

Figure 3.10: The plane constraint distance is minimization (Farrell, 2005).

A common example of this implementation is adding a plane constraint to the elbow joint in order to avoid torso protrusions while walking. Figure 3.11 shows the effect of running optimization-based posture prediction with the plane constraint parented to the elbow joint.

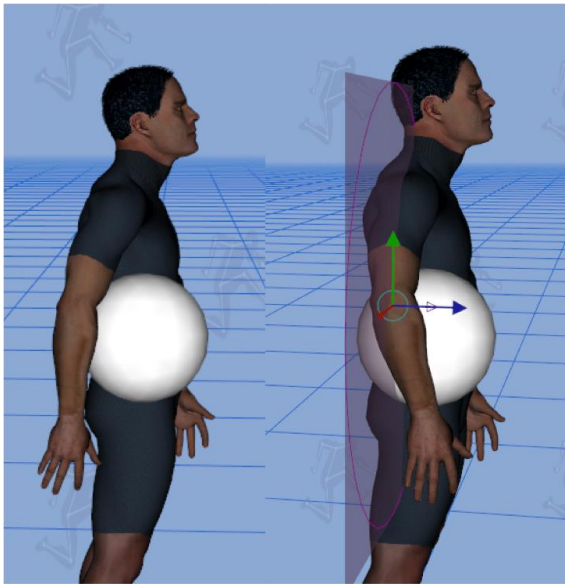

Figure 3.11: The posture on the left was obtained through standard posture prediction, whereas the posture on the right used the plane constraint with posture prediction.

Within this implementation of self-avoidance, the use of the plane constraint is targeted at bisecting frames of collision groups. The goal is to maximize the chance of surrounding collision frames being fixed through interpolation, and also to enforce navigation around objects. Through recursive bisection, newly interpolated joint rotation curves are constantly being created. These new curves provide safer joint positions for posture prediction and an updated direction of motion used for orienting the plane constraint. This strategy will keep posture prediction from suggesting rogue postures that are not consistent with the path of motion for a joint.

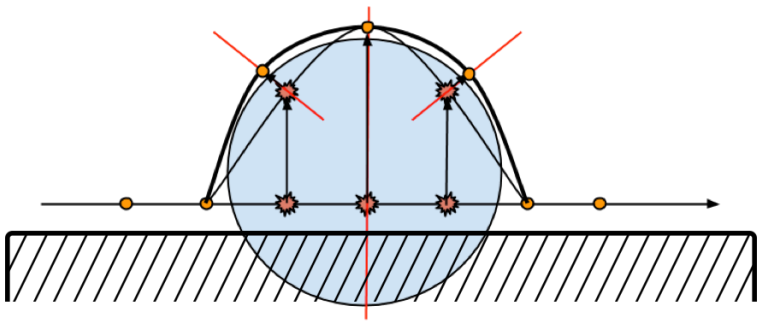

Figure 3.12: The red lines represent the use of directional plane constraints with posture prediction for a simple collision group.

3.4.2 Task-Based Constraints

A task-based constraint is one that is specifically implemented to ensure that posture prediction does not unknowingly guide the motion in a way that violates an objective of the animation. Several of the tasks that are the focus of this work require the avatar to hold a two-handed weapon. This weapon will generally be parented to only one hand in the scene, so it is entirely possible for self-avoidance adjustments to cause an

undesirable loss of contact with the opposite hand. To prevent this, a constraint can be added to posture prediction that mandates this contact. Self-avoidance and realistic weapon contact can be simultaneously maintained through the strategic placement of an end-effector and target point in the scene, and by capturing the opposite hand direction with respect to the weapon.

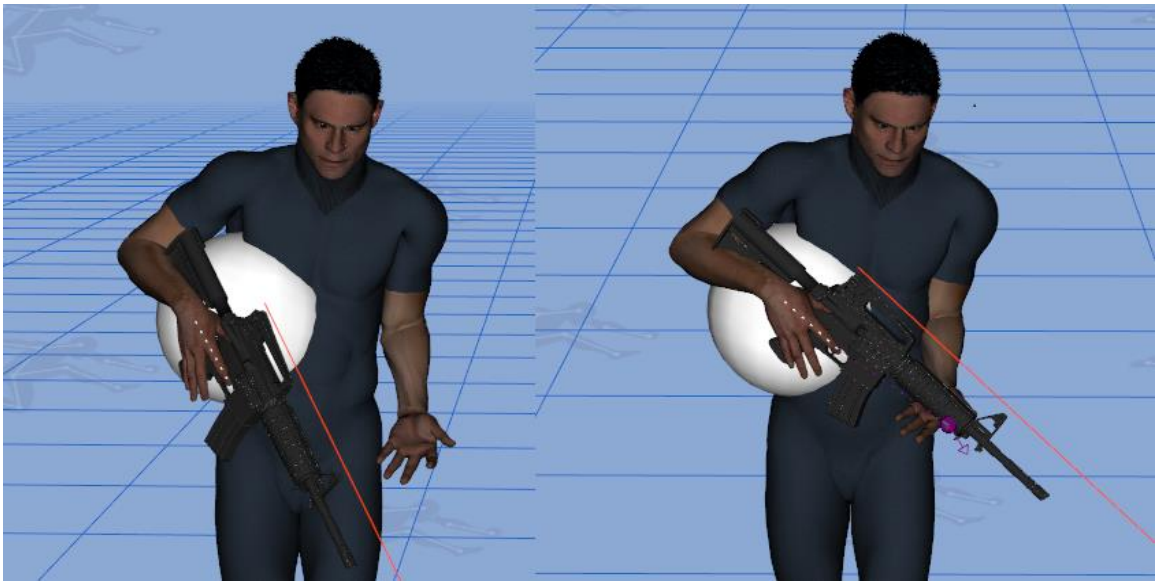

Figure 3.13: How the two-handed weapon constraint prevents loss of contact with the weapon.

In order to implement this constraint, a target point is parented to the trigger finger of the avatar. This target point is then offset from the parented finger and positioned at the opposite hand point of contact. An end-effector is also placed at this point of contact and parented to the opposite hand. The optimization-based posture prediction will attempt to minimize the distance between the target point and end-effector to prevent the loss of weapon contact. Task-based constraints are fairly simple for the developer to create, and once the constraint has been added to the system, it can easily be used with any future animation that warrants it.

3.4.3 Constraint Implementation

In order to manage the constraints used with the optimization-based posture prediction, there is a structure that allows for dynamically adding and updating them based on the task. This means it becomes possible to pick and choose which constraints are needed for specific tasks and call a generic update method for those within the recursive bisection. These updates are done right before posture prediction provides the new set of joint angles, as shown in the following figure.

```
RunPostureAvoidance( frame )
{
    for each plane constraint
    {
        update the plane constraint for the frame
    }

    for each task constraint
    {
        update the task constraint for the frame
    }

    run posture prediction on the kinematic model

    for each joint rotation curve
    {
        replace the key for the frame with the values of the predicted posture
    }
}
```

Figure 3.14: The general strategy for updating plane and task-based constraints, running posture prediction, and replacing the frame values.

CHAPTER 4

RESULTS

4.1 Upper Body Self-Avoidance with a Simple Obstacle

One of the primary motivating factors for the work of this thesis is to allow for upper body self-collision avoidance. This is especially useful in animations that contain significant amounts of arm movement. For the initial tests, a walking forward animation and a vertical jump animation were used. Each of these animations contain arm movement that would interfere with objects attached to the side of the torso. Figures 4.1 and 4.2 show the effectiveness of the algorithm with avoiding a sphere model that has been attached to the torso. The plane constraints were focused on the elbows, and the optimization-based posture prediction focused on minimizing effort.

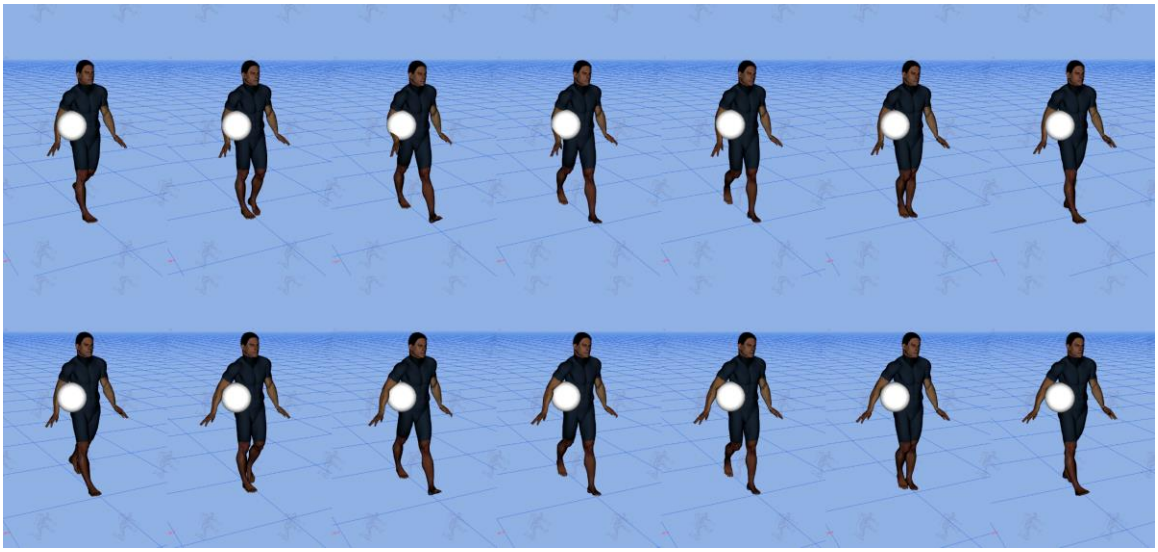

Figure 4.1: A standard walking animation where the top frames were played directly from the animation file, and the bottom frames were processed for self-collision avoidance.

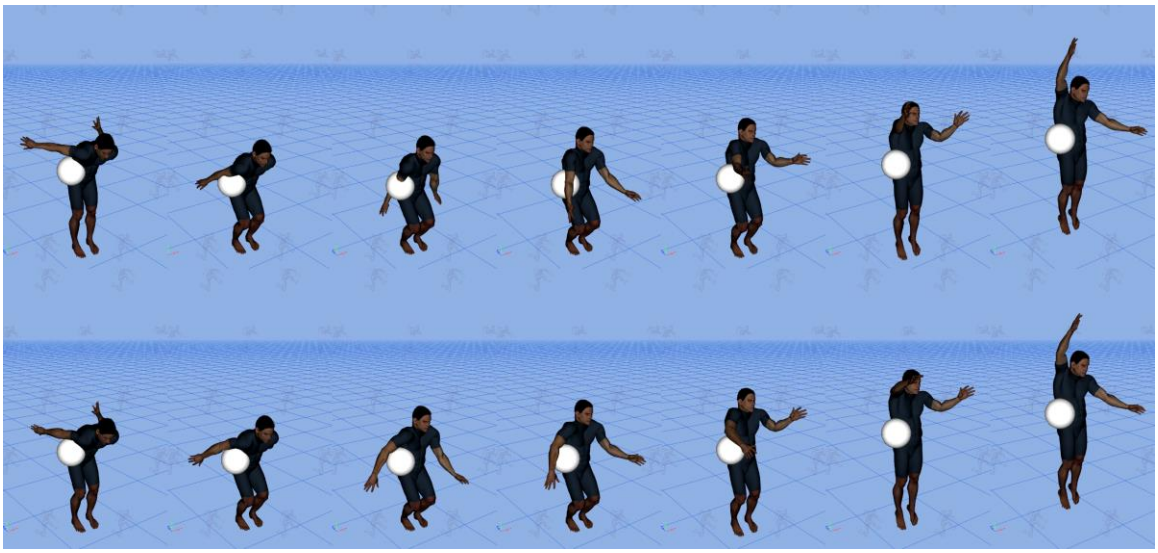

Figure 4.2: A vertical jump animation where the top frames were played directly from the animation file, and the bottom frames were processed for self-collision avoidance.

4.2 Upper Body Self-Avoidance with Complex Obstacles

Although it is difficult to gauge the fluidity of a motion through a picture, the results show a smooth avoidance of the obstacle in both cases. Despite this result showing promise, the obstacle used for these tests is greatly simplified and not an accurate representation of equipment that would need to be avoided in practical use. For the next test, the aforementioned equipment sphere filling method is used to create more realistic jumping and walking scenarios. The jumping motion avoids collision on both sides of the body with no noticeable breaks in coherence. The walking motion also avoids collision successfully, but exposes the need for a two-handed weapon constraint. Figure 4.4 shows how adjusting for self-collision avoidance can potentially hinder objectives of the task, and Figure 4.5 shows how the two-handed weapon constraint can

prevent this. Again, the plane constraints were focused on the elbows, and posture prediction was focused on minimizing effort.

Figure 4.3: The avatar adjusting posture to avoid collision during the take-off phase of the vertical jump animation.

Figure 4.4: Multiple views of the animation for walking with a weapon. The two views on the left show the animation with no self-avoidance, and the views on the right demonstrate how self-avoidance pulls the weapon away from its desired position in the opposite hand.

Figure 4.5: The walking animation with the two-handed weapon constraint added, the opposite hand is once again kept in the guiding position.

4.3 Full Body Avoidance for Complex Motions

The results obtained to this point have only pertained to upper body avoidance between the arms and the torso, but the legs should also be considered. Self-collisions involving the legs are far less common, but also far more difficult to handle. This is because the arms rarely form points of contact with the ground or need to support the weight of the avatar. Currently, there exists no performance measure implemented within the optimization-based posture prediction that will account for the balance of the avatar or the limited range of motion created by ground contact. This means that as soon as a collision occurs with a limb that is supporting the weight of the avatar, nothing will enforce the limb to continue supporting the weight and thereby maintain a realistic posture. This became apparent when an animation was tested that involved the avatar

transitioning between prone and standing postures. If posture prediction minimizes effort across all joints in the body like had been done previously, the legs will assume unrealistic positions. More specifically, the results show that they lose contact with the ground entirely. Interestingly, once the joints of weight bearing limbs are frozen, the upper body and remaining limbs still avoid collision in a realistic manner.

Figure 4.6: Allowing posture prediction to minimize effort across all joints. The left posture is the frame from the animation file that contains self-collisions and the right posture uses the avoidance method on this frame.

Figure 4.7: Freezing the lower body joints and predicting posture.

Intuitively, it can be reasoned that in a real world scenario, these limbs would actually behave in a similarly restricted fashion. If, like in Figures 4.6 and 4.7, the knee

and foot belonging to the same leg form points of contact with the ground, that leg will have a fairly limited range of motion. Once a limited range of motion is forced onto the lower body, a seemingly realistic avoidance strategy can once again be obtained. This gives us the basis for the assumption that if optimization-based posture prediction had a performance measure capable of restricting range of motion for weight bearing joints, the recursive algorithm would be capable of handling the scenario in a realistic way.

4.4 Implementation with Predictive Dynamics

The motivating factor for the work of this thesis is to supply the predictive dynamics optimization formulation with a reference motion that is free of self-collisions. The objective function specific to minimizing the distance from the reference motion must be weighted enough within the formulation to keep the solution from moving back into self-collision. A situation may also be encountered where the reference motion violates the equations of motion, and the optimization process brings the solution back into self-collision. Of the cases tested for walking and jumping, however, this did not seem to be an issue.

Upon further analysis of the predictive dynamics optimization problem, some clarity is provided. Because the only other objective function being minimized is a weighted sum of joint torque limits over time, the reference motion is given significant consideration within the formulation. Although it is at the discretion of the predictive dynamics task developer, the reference motion objective function is generally weighted evenly with joint torque minimization. As the weight assigned to the reference motion

decreases, the chance for a reintroduction of self-collisions into the motion will likely increase.

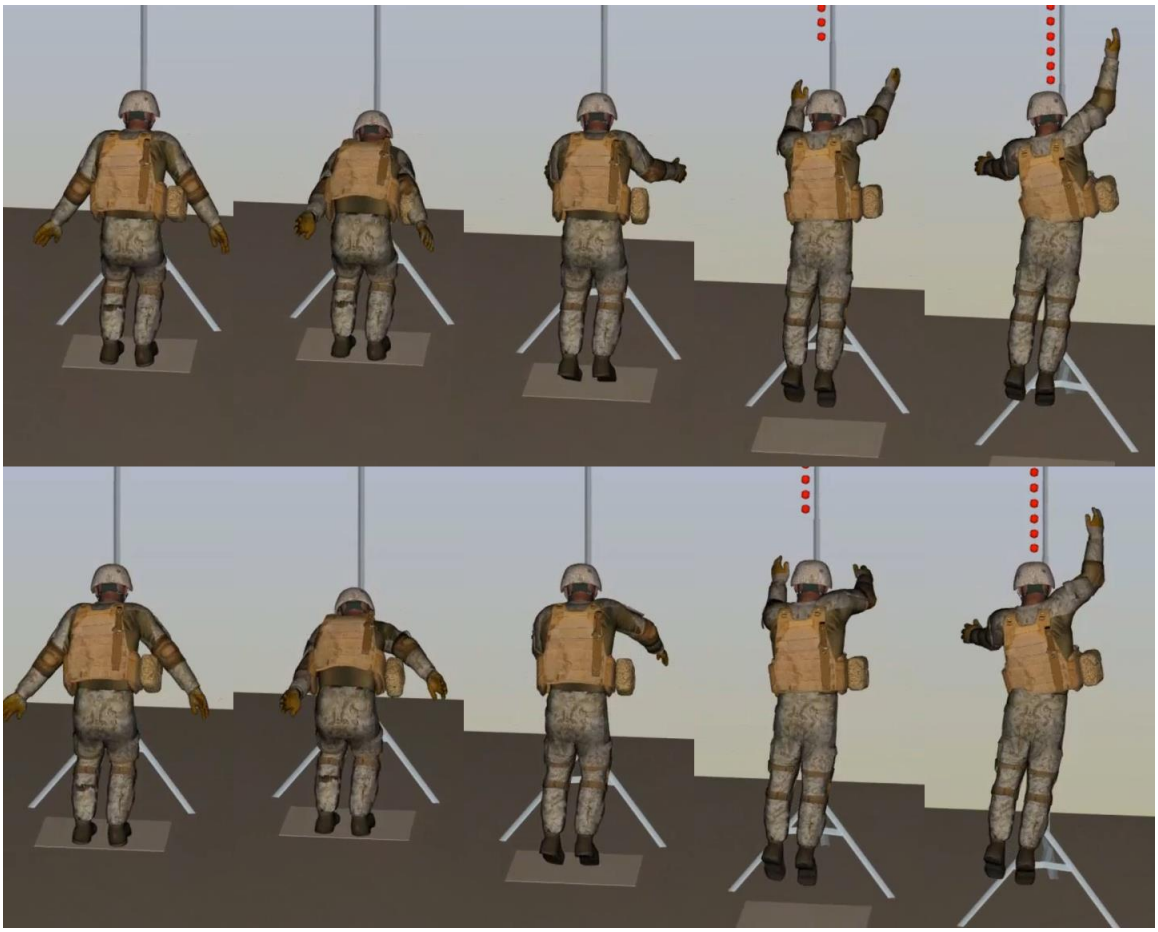

Figure 4.8: The vertical jump predictive dynamics task, where the jump shown on the bottom used a reference motion processed for self-avoidance.

CHAPTER 5

CONCLUSIONS

In this thesis, it has been shown that keyframe interpolation and optimization-based posture prediction have the potential to work well together towards the goal of processing animations for self-collision avoidance. Relying on the minimization of effort as a performance measure has proved to be especially successful for upper body avoidance. Although there are issues with weight-bearing limbs, it stands to reason that this could be overcome with further research towards additional performance measures.

As a self-collision avoidance method specific to predictive dynamics reference motions, the current implementation should provide a reliable approach to upper body avoidance if the motion is relatively upright. More specifically, this method works especially well for scenarios like walking, running, ascending stairs, and jumping, where upper body avoidance is generally the issue. However, because there is potential for the predictive dynamics system to reintroduce collisions, processing the reference motion for self-avoidance cannot be considered a completely reliable means for the predictive dynamics output. The alternative approach would be to integrate a method for self-avoidance into the predictive dynamics system. While this may provide more reliable results, it could also significantly decrease the performance of predictive dynamics. For this reason, a low cost option for self-avoidance may be of significant value here.

This work also has potential within the field of animation, were the use of keyframe interpolation is common. This method could allow animators to automate self-

avoidance for general motions where minor adjustments are needed. For example, if an articulated figure needed to avoid a newly added body obstacle, the animators would not need to go back and adjust all keyframes to account for this change. Instead, they could use a single reference motion and continually process this motion for self-avoidance anytime a wardrobe or upper-body change has been made. The task-based constraints may also be of some use in this field, as they allow the user to modify an entire animation around newly introduced objectives. These constraints are fairly simple to develop from a programmatic standpoint and would be especially useful to animators if they could be created through non-programmatic means.

With regards to the effectiveness of the plane constraints, the tests show that the posture prediction transitions smoothly between frames. The plane constraints also ensure that obstacles are passed by the limbs through natural avoidance strategies. If, however, a flat, plane-like obstacle were added to the body, it is entirely possible that the avoidance strategy would encounter issues. Not only would this object be difficult to represent through a sphere approximation for collision detection, but it may be narrow enough to fall between frames. Fortunately, objects this flat are rarely attached to the body in ways that would cause the issue to present itself, especially in practical scenarios. Still, it should be recognized that situations like these may exist, and that the current implementation is not absolute.

Introducing the idea of task-based constraints served as an effective means for preserving objectives within the original animation. These constraints are not only

conceptually simple, but this implementation makes it fairly easy for the developer to add them to an animation. Optimization-based posture prediction as a means for collision response is what made the creation of these constraints possible, and is therefore unique to this implementation. It is not inconceivable to imagine an entire suite of generic task-based objectives becoming available to the developer as this implementation is used with more animations. However, because adding these constraints will likely make the optimization problem within posture prediction more difficult, a decrease in performance should be expected.

Although this algorithmic approach to self-collision avoidance cannot be considered absolute, it can still have significant use in the fields of animation and virtual reality. Essentially, the current implementation is not only providing an effective and low cost means for upper body self-avoidance, but also a framework for a more complete solution to be implemented through further research.

5.1 Future Work

The most useful improvement to this approach would be a way to realistically enforce ground contact within the optimization-based posture prediction. The effort performance measure already offers a convenient means for this addition because it assigns a weighting value to each joint. If this weighting value were to be modified based on contact with the ground, the optimization may encourage a more realistic solution. As a result, it would then become less desirable to move the weight-bearing limbs when adjusting the posture for self-avoidance. Another option would be to limit

the joint ranges of motion based on contact with the ground. This would no longer be through the use of a performance measure within the optimization formulation, it would instead be defined as a constraint. Regardless of the implementation method chosen, more research would be needed to determine exactly how the joints should behave as a result of ground contact.

An improvement that could be made to the keyframe interpolation algorithm would be to add a velocity constraint. Liu and Cohen (1995) introduce a way to relax speed and timing, and this algorithm could benefit from a similar approach. Because collisions are fixed in groups, the bounding frames of these groups are not changed from the original motion. Thus, the avatar must still assume those postures at the originally given time frames. If the predicted path of motion for the collision group greatly extends the travel distance of a limb, the resulting motion may contain unrealistic velocity. If a velocity constraint were introduced that limited the allowable change in rotation of the joints between frames, the violating frames could then be removed and interpolated to relax the timing. Additionally, this change would likely alter the path of motion in such a way that it appears more realistic.

REFERENCES

Abdel-Malek, K., Yang, J., Kim, J.H., Marler, T., Beck, S., Swan, Colby, Frey-Law, L., Mathai, A., Murphy, C., Rahmatallah, S., and Arora, J., 2007, "Development of the Virtual-Human Santos®", *Digital Human Modeling*, Lecture Notes in Computer Science, Vol. 4561, pp. 490-499.

Abdel-Malek, K. and Arora, J., 2013, *Human Motion Simulation: Predictive Dynamics*, Academic Press, 1st Edition.

Abend, W., Bizzi, E., and Morasso, P., 1982, "Human arm trajectory formation", *Brain*, 105, pp. 331-348.

Bataineh, Mohammad, 2012, *Artificial Neural Network for Studying Human Performance*, M.S. Thesis, University of Iowa, Iowa.

Denavit, J. and Hartenberg, R.S., 1955, "A kinematic motation for lower-pair mechanisms based on matrices", *Journal of Applied Mechanics*, Vol. 77, pp. 215-221.

Farrell, K. and Marler, R.T., 2004, "Optimization-based kinematic models for human posture", *SAE Digital Human Modeling for Design and Engineering* (June 14-16, 2005), Iowa City, IA.

Farrell, K., 2005, *Kinematic Human Modeling and Simulation using Optimization-Based Posture Prediction*, M.S. Thesis, University of Iowa, Iowa.

Flash, T. and Hogan, N., 1985, "The coordination of arm movements: an experimentally confirmed mathematical model", Journal of Neuroscience, Vol. 5, pp. 1688-1703.

Hubbard, P.M., 1995, "Collision Detection for Interactive Graphics Applications", *IEEE Transactions on Visualization and Computer Graphics*, Vol. 1, No. 3, pp. 218-230.

Hubbard, P.M., 1996, "Approximating Polyhedra with Spheres for Time-Critical Collision Detection", *ACM Trans. on Graphics*, Vol. 15, No. 3, pp. 179-210.

Liu, Z., and Cohen, M.F., 1995, "Keyframe motion optimization by relaxing speed and timing", *1995 Eurographics Workshop on Animation*, Maastrich, Holland.

Nebel, J.-C., 1999, "Keyframe interpolation with self-collision avoidance", *Computer Animation and Simulation '99*, Eurographics, Springer Computer Science, pp. 77-86.

Nebel, J.-C., 2000, "Realistic collision avoidance of upper limbs based on neuroscience models", Eurographics '2000, Interlaken, Switzerland, Vol. 19, No. 3.

O'Sullivan, C., Radach, R., and Collins, S., 1999, "A model of collision perception for real-time animation", *Computer Animation and Simulation '99*, Eurographics, Springer Computer Science, pp. 67-76.

O'Sullivan, C. and Dingliana, J., 1999, "Real-Time Collision Detection and Response using Sphere-Trees", *Proc. Spring Conference on Computer Graphics, Slovokia*, 83-92.

Palmer, I.J. and Grimsdale, R.L., 1995, "Collision Detection for Animation using Sphere-Trees", *Computer Graphics Forum*, Vol. 14, No. 2, pp. 105-116.

Quinlan, S., 1994, "Efficient Distance Computation between Non-Convex Object", *Proceedings International Conference on Robotics and Automation*, pp. 3324-3329.

Rose, C., Bodenheimer, B. and Cohen, M., 1998, "Verbs and Adverbs: Multidemensional Motion Interpolation Using Radial Basis Functions", *IEEE Computer Graphics and Applications*, Vol. 18, pp. 32-40.

Thalmann, N. Magnenat and Thalmann, D., 1991, "Complex models for animating synthetic actors", IEEE Computer Graphics and Applications, Vol. 11, No. 5, pp. 32-44.

Xiang, Y., Chung, H., Kim, J., Bhatt, R., Rahmatalla, S., Yang, J., Marler, T., Arora, J., and Abdel-Malek, K., 2010, "Predictive dynamics: an optimization-based novel approach for human motion simulation", *Structural and Multidisciplanary Optimization*, Volume 41, Issue 3, pp. 465-479.

Yang, J., Rahmatalla, S. ,Marler, T., Abdel-Malek, K., and Harrison, C., 2007, "Validation of Predicted Posture for the Virtual Human Santos®", *Digital Human Modeling*, Lecture Notes in Computer Science, Vol.4561, pp. 500-510.

Zhang, X. and Chaffin, D.B., 2005, "Digital Human Modeling for Computer-Aided Ergonomics", *Handbook of Occupational Ergonomic*s, CRC Press.

Zhao, L., Liu, Y., and Badler, N., 2005, "Applying empirical data on upper torso movement to real-time collision-free reach tasks", *SAE Transactions Journal of Passenger Cars – Mechanical Systems.*

APPENDIX

RECURSIVE BISECTION AVOIDANCE CODE

```
/*****************************************************************************
 *
 * Source       :   DnSelfAvoidanceWriter
 * File         :   DnSelfAvoidanceWriter.cs
 * Additional   :   Class for writing motion capture files modified for SA
 * Project      :   Silicon
 * Author       :   rkd, Oct. 31, 2013
 *
 * Copyright (C) 2012-13 Virtual Soldier Research University of Iowa
 * All rights reserved. Duplication in any medium, electronic or otherwise is
 * prohibited without express authorization from VSR
 *
 *****************************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Core;
using Dynamics.Deprecated;
using SCG = System.Collections.Generic;

namespace Dynamics
{
   /**
    * Write modified motion capture file from spline or mocap input
    */
    public class DnSelfAvoidanceWriter : DnTextWriterBase
    {
       #region Private Members

       //!< The currently picked avatar
       private static CsAvatar ms_kAvatar = null;
       //!< The global position for the avatar
       private CsVector4f m_vGlobalPosition;
       //!< The position for the avatar
       private CsVector4f m_vPosition;
       //!< The joint curves for the originial mocap
       private static SCG.Dictionary<int, CsCurve> ms_kOriginalJoints;
       //!< The joint curves for the modified mocap
       private static SCG.Dictionary<int, CsCurve> ms_kModifiedJoints;
       //!< The plane constraints being used
       private SCG.List<_PlaneConstraint> m_kPlaneConstraints;
       //!< The task constraints being used
       private SCG.List<_ITaskConstraint> m_kTaskConstraints;
       //!< The MOCAP file
       private CsGenericParser m_kMOCAPFile;
       //!< The # of frames in the output MOCAP file
       private static int ms_iMOCAPFrames;
       //!< The play speed of the output MOCAP
```

```
private float m_fMOCAPSpeed;
//!< The spline as a generic data file
private CsGenericData m_kSplineData;
//!< The spline joint curves
private DnJointSplineCollection m_kSplineJoints;
//!< The spline with start and end time
private DnJointSpline m_kSpline;
//!< The spline evaluator
private DnICurveEvaluator m_kSplineEval;
//!< True if mocap, false if spline for input
private bool m_bMocap;
//!< The percentage of collision group smoothing
private float m_fSmoothing = 0.0f;
//!< The required group size to run posture
private int m_iMinGroupSize = 0;
//!< The evaluation points
private List<int> m_kEvalPoints;
//!< The weapon constraint, true if used
private bool m_bWeapon = false;
//!< Use elbow plane constraints
private bool m_bElbows = false;
//!< Use wrist plane constraints
private bool m_bWrists = false;
//!< Lower bounds for joint angles
private int m_iLowerBounds = 6;
//!< Upper bounds for joint angles
private int m_iUpperBounds = 35;

#endregion

/**
 * Constructor for MOCAP
 */
public DnSelfAvoidanceWriter(CsAvatar kAvatar, CsGenericParser kMocapFile,
   string sFilename, bool bElbows, bool bWrists, bool bWeapon, float
   fSmooth, int iMinBuffer, List<int> kEvalPoints)
{
   m_sFileName = sFilename;
   // get the MOCAP file, delta time, and frame count
   m_kMOCAPFile = kMocapFile;
   m_fMOCAPSpeed = Convert.ToSingle(m_kMOCAPFile.GetWord());
   ms_iMOCAPFrames = Convert.ToInt32(m_kMOCAPFile.GetWord());
   // set the avatar and the task
   ms_kAvatar = kAvatar;
   m_vGlobalPosition = new CsVector4f();
   if (kAvatar != null)
   {
      m_vPosition = ms_kAvatar.GetPosition(true);
   }

   m_bMocap = true;
   m_bElbows = bElbows;
   m_bWrists = bWrists;
   m_bWeapon = bWeapon;
   m_fSmoothing = fSmooth / 100f;
   m_iMinGroupSize = (iMinBuffer * 2) + 1;
   m_kEvalPoints = kEvalPoints;
}
```

```
/**
 * Constructor for Spline
 */
public DnSelfAvoidanceWriter(CsAvatar kAvatar, CsGenericData kSplineFile,
   string sFilename, int iFrames, float fSpeed, bool bElbows, bool
   bWrists, bool bWeapon, float fSmooth, int iMinBuffer, List<int>
   kEvalPoints)
{
   m_sFileName = sFilename;
   // get the Spline file data, setup the evaluator
   m_kSplineData = kSplineFile;
   m_kSplineJoints = new DnJointSplineCollection();
   m_kSpline = m_kSplineJoints.AddData(m_kSplineData.Data, 0.0f);
   m_kSplineEval =
      m_kSplineJoints.GetEvaluator(eDnJointEvaluationType.Angle);
   m_fMOCAPSpeed = fSpeed;
   ms_iMOCAPFrames = iFrames;
   // set the avatar
   ms_kAvatar = kAvatar;
   m_vGlobalPosition = new CsVector4f();
   if (kAvatar != null)
   {
      m_vPosition = ms_kAvatar.GetPosition(true);
   }

   m_bMocap = false;
   m_bWeapon = bWeapon;
   m_bElbows = bElbows;
   m_bWrists = bWrists;
   m_fSmoothing = fSmooth / 100f;
   m_iMinGroupSize = (iMinBuffer * 2) + 1;
   m_kEvalPoints = kEvalPoints;
}

/**
 * Creates and formats a string to be written to a file
 * @return a string to be written to a file
 */
public override string WriteText()
{
   // set posture prediction settings
   ms_kAvatar.SetProp("PostureWeightEffort", new CsDouble(1.0));
   ms_kAvatar.SetProp("PostureWeightJoint", new CsDouble(0.0));
   ms_kAvatar.SetProp("PostureStrengthPercentile", new CsDouble(5.0));

   // builds the string representing the modified MOCAP file
   StringBuilder kStringBuilder = new StringBuilder();

   // create the animation curves that represent the MOCAP file
   ms_kOriginalJoints = _CreateAnimCurves();
   // create a copy of the animation curves to be modified
   ms_kModifiedJoints = _CreateAnimCurves();
   // creates all the plane constraints that will be used with posture
   prediction
   m_kPlaneConstraints = _CreatePlaneConstraints();

   // get the current avatar posture, to apply back after avoidance has
```

```
been run
float[] fJointArray =
   CsCore.AvatarSystem.GetJointAngleArray(ms_kAvatar);

// creates all task based constraints that will be used with
posture prediction
m_kTaskConstraints = new List<_ITaskConstraint>();
_CreateTaskConstraints();

// sets up the body spheres in case they have been cleared
CsPosturePredict.Predict(ms_kAvatar);

// use evaluation points to free up important frames of the animation
_PreProcess();

// passes all MOCAP frames into recursive bisection avoidance
_RecursiveAvoid(0.0f, (float)(ms_iMOCAPFrames - 1));

// reapply initial posture
CsCore.AvatarSystem.SetJointAngleArray(ms_kAvatar, fJointArray);

// finished using plane constraints, so destroy
_DestroyPlaneConstraints();

// finished using task constraints, so destroy
_DestroyTaskConstraints();

// start building the MOCAP file with the generic header info
kStringBuilder.AppendLine("# Joint profiles in rad");
kStringBuilder.AppendLine("# first row is delta time");
kStringBuilder.AppendLine("# second row is the number of increments");
kStringBuilder.AppendLine("# next row is the initial Q values, followed
   by the Q values for time 2 and so on...");
kStringBuilder.AppendLine("# G1    G2      G3      G4      G5      G6
   Q1      Q2      Q3      Q4      Q5...");
kStringBuilder.AppendLine((m_fMOCAPSpeed).ToString("F6"));
kStringBuilder.AppendLine(ms_iMOCAPFrames.ToString());

float[] fGPR = new float[6];
float[] fJoints = new float[49];
// for each frame in the mocap animation
for (int i = 0; i < ms_iMOCAPFrames; i++)
{
   // first three values are global position
   float fTx =
      ms_kOriginalJoints[0].Data.FindClosestKey((float)i).Value;
   fGPR[0] = fTx;
   float fTy =
      ms_kOriginalJoints[1].Data.FindClosestKey((float)i).Value;
   fGPR[1] = fTy;
   float fTz =
      ms_kOriginalJoints[2].Data.FindClosestKey((float)i).Value;
   fGPR[2] = fTz;

   // next 3 are global rotation
   float fRx =
      ms_kOriginalJoints[3].Data.FindClosestKey((float)i).Value;
   fGPR[3] = fRx;
```

```
            float fRy =
               ms_kOriginalJoints[4].Data.FindClosestKey((float)i).Value;
            fGPR[4] = fRy;
            float fRz =
               ms_kOriginalJoints[5].Data.FindClosestKey((float)i).Value;
            fGPR[5] = fRz;

            string sAvoidPosture = fGPR[0].ToString("F6") + "\t" +
               fGPR[1].ToString("F6") + "\t" + fGPR[2].ToString("F6") + "\t" +
               fGPR[3].ToString("F6") + "\t" + fGPR[4].ToString("F6") + "\t" +
               fGPR[5].ToString("F6") + "\t";

            for (int k = 6; k < 54; k++)
            {
               sAvoidPosture = sAvoidPosture +
               ms_kModifiedJoints[k].Data.FindClosestKey((float)i).Value
               .ToString("F6") + "\t";
            }
            sAvoidPosture = sAvoidPosture +
               ms_kModifiedJoints[54].Data.FindClosestKey((float)i).Value
               .ToString("F6");

            kStringBuilder.AppendLine(sAvoidPosture);
         }

         return kStringBuilder.ToString();
      }

      #region Preprocessing phase

      /**
       * Single pass avoidance around evaluation frames
       */
      private void _PreProcess()
      {
         List<int> kRemoveList = new List<int>();
         // process each evaluation frame for self-avoidance
         foreach (int iFrame in m_kEvalPoints)
         {
            // apply frame from the modified mocap curves
            _ApplyCurvePosture(iFrame, false);

            // if a collision is detected
            if (_CollisionDetected())
            {
               // run posture prediciton to fix the frame
               _RunPostureAvoidance(iFrame);
            }
            else
            {
               // remove frame from list if not in collision
               kRemoveList.Add(iFrame);
            }
         }

         // remove frames not in collision
         foreach (int iRemove in kRemoveList)
         {
```

```
            m_kEvalPoints.Remove(iRemove);
        }

        int iStart;
        int iFinish;
        // interpolate surrounding frames if they're in collision
        foreach (int iFrame in m_kEvalPoints)
        {
            iStart = iFrame;
            iFinish = iFrame;
            // work backwards from the collision frame
            for (int i = iFrame - 1; i >= 0; i--)
            {
                _ApplyCurvePosture(i, false);
                if (_CollisionDetected())
                {
                    iStart = i;
                    // remove all keys for this time step
                    for (int iJoint = m_iLowerBounds; iJoint < m_iUpperBounds;
                        iJoint++)
                    {
                        CsCurve kCurve;
                        if (ms_kModifiedJoints.TryGetValue(iJoint, out kCurve))
                        {
                            kCurve.Data.RemoveKey(kCurve.Data.GetKey((float)i).Time);
                        }
                    }
                }
                else // terminate
                {
                    i = -1;
                }
            }

            // work forwards from the collision frame
            for (int j = iFrame + 1; j <= (ms_iMOCAPFrames - 1); j++)
            {
                _ApplyCurvePosture(j, false);
                if (_CollisionDetected())
                {
                    iFinish = j;
                    // remove all keys for this time step
                    for (int iJoint = m_iLowerBounds; iJoint < m_iUpperBounds;
                        iJoint++)
                    {
                        CsCurve kCurve;
                        if (ms_kModifiedJoints.TryGetValue(iJoint, out kCurve))
                        {
                            kCurve.Data.RemoveKey
                              (kCurve.Data.GetKey((float)j).Time);
                        }
                    }
                }
                else // terminate
                {
                    j = ms_iMOCAPFrames;
                }
            }
```

```
            // add back frames through recursive bisection and cardinal
            evaluation
            _CardinalBisection(iStart, iFinish);
        }
    }

    #endregion

    #region Recursive Bisection

    /**
     * Recursively runs Posture Prediction and interpolates the animation
     * curves to avoid self collisions
     * @param   fStart the starting time frame for a group of frames
     * @param   fEnd the ending time frame for a group of frames
     */
    private void _RecursiveAvoid(float fStart, float fEnd)
    {
        CsCore.Log.Debug("Recursion called with bounds " + fStart + " to " +
            fEnd);
        // flag used to determine if a collision is occuring
        bool bColliding = false;
        // temporary storage for the start frame in collision groups
        int iStartFrame = 0;

        // stores all collision groups between fStart and fEnd
        SCG.List<_SelfCollision> kGroups = new SCG.List<_SelfCollision>();

        // find all collision groups
        for (int iFrame = (int)fStart; iFrame <= (int)fEnd; iFrame++)
        {
            // apply frame from the modified mocap curves
            _ApplyCurvePosture(iFrame, false);

             // if a collision is detected
             if (_CollisionDetected())
             {
                // if not currently colliding
                if (!bColliding)
                {
                    // record the start frame of the collision
                    iStartFrame = iFrame;
                    bColliding = true;
                }
                // if the last frame in the entire animation is in collision
                // it needs to be fixed in order for the group to be completed
                if (iFrame == (ms_iMOCAPFrames - 1) )
                {
                    // fix the first frame and last frame and finish the group
                    if (iStartFrame == 0)
                        _RunPostureAvoidance(0);
                    _RunPostureAvoidance(iFrame);
                    kGroups.Add(new _SelfCollision(iStartFrame + 1, iFrame - 1));
                    bColliding = false;
                }
             }

             // if no collision is detected
```

```
            else
            {
                // if previously colliding
                if (bColliding)
                {
                    // add the collision group to the list of groups
                    kGroups.Add(new _SelfCollision(iStartFrame, iFrame - 1));
                    bColliding = false;
                }
            }
        }

        CsCore.Log.Debug("Collision groups found: " + kGroups.Count);

        // process each group of collisions
        foreach (_SelfCollision kGroup in kGroups)
        {
            // get the time key that bisects the group
            float fBisectTime = kGroup.StartFrame + ((kGroup.EndFrame -
                kGroup.StartFrame) / 2);
            float fMidPoint = 0.0f;

            // this is ugly, but guarentees the middle time we use has a key
            associated with it
            CsCurve kSampleCurve;
            if (ms_kOriginalJoints.TryGetValue(0, out kSampleCurve))
            {
                fMidPoint = kSampleCurve.Data.FindClosestKey(fBisectTime)
                    .Time;
            }
            else
            {
                CsCore.Log.Error("No middle frame found for self-avoidance, the
                impossible happened.");
                return;
            }

            // buffer the group based on smoothing percentage
            _SelfCollision kBufferedGroup = _BufferGroup(kGroup);
            // update the bounding frames for the group
            kGroup.StartFrame = kBufferedGroup.StartFrame;
            kGroup.EndFrame = kBufferedGroup.EndFrame;

            // for each frame in the current collision group
            for (int iGroupFrame = kGroup.StartFrame; iGroupFrame <=
                kGroup.EndFrame; iGroupFrame++)
            {
                // ensures a minimum frame buffer around posture prediction
                if (kGroup.EndFrame - kGroup.StartFrame >= m_iMinGroupSize)
                {
                    // if the frame is the midpoint
                    if ((float)iGroupFrame == fMidPoint)
                    {
                        // apply the midpoint frame for posture prediction
                        _ApplyCurvePosture(iGroupFrame, false);
                        // run posture prediciton to fix the frame
                        _RunPostureAvoidance(iGroupFrame);
                    }
```

```
                // if the frame is not the midpoint
                else
                {
                    // remove all keys for this time step
                    for (int iJoint = m_iLowerBounds; iJoint < m_iUpperBounds;
                        iJoint++)
                    {
                        CsCurve kCurve;
                        if (ms_kModifiedJoints.TryGetValue(iJoint, out kCurve))
                        {
                            kCurve.Data.RemoveKey
                            (kCurve.Data.GetKey((float)iGroupFrame).Time);
                        }
                    }
                }
            }
        }

        // add back frames through recursive bisection and evaluation
        //_CardinalBisection(kGroup.StartFrame, kGroup.EndFrame);
        _QuadraticPolyFit(kGroup.StartFrame, (int)fMidPoint,
            kGroup.EndFrame);

        // now that the midpoint is fixed, get the previous and next
        points
        float fPrevPoint = fMidPoint - 1.0f;
        float fNextPoint = fMidPoint + 1.0f;

        // recursively check for collisions in new evaluated keys
        if (kGroup.StartFrame != fMidPoint)
        {
            _RecursiveAvoid(kGroup.StartFrame, fPrevPoint);
        }
        if (kGroup.EndFrame != fMidPoint)
        {
            _RecursiveAvoid(fNextPoint, kGroup.EndFrame);
        }
    }
}

#endregion

#region Private Helper Methods

/**
 * Y = AX^2 + BX + C
 * @param iStart the first frame of the group
 * @param iMid the middle frame of the group
 * @param iEnd the last frame of the group
 */
 private void _QuadraticPolyFit(int iStart, int iMid, int iEnd)
 {
    // foreach joint curve
    for (int iJoint = m_iLowerBounds; iJoint < m_iUpperBounds; iJoint++)
    {
        CsCurve kCurve;
        if (ms_kModifiedJoints.TryGetValue(iJoint, out kCurve))
```

```
        {
            float fX1 = iStart;
            float fY1;
            if (kCurve.Data.GetKey(fX1) == null)
            {
                fY1 = kCurve.Evaluate(fX1);
                kCurve.Data.AddKey(fX1, fY1);
            }
            else
            {
                fY1 = kCurve.Data.GetKey(iStart).Value;
            }

            float fY2 = kCurve.Data.GetKey(iMid).Value;
            float fX2 = iMid;

            float fX3 = iEnd;
            float fY3;
            if (kCurve.Data.GetKey(fX3) == null)
            {
                fY3 = kCurve.Evaluate(fX3);
                kCurve.Data.AddKey(fX3, fY3);
            }
            else
            {
                fY3 = kCurve.Data.GetKey(iEnd).Value;
            }

            // find A, B, and C
            float fA = ((fY2 - fY1)*(fX1 - fX3) + (fY3 - fY1)*(fX2 - fX1)) /
                ((fX1 - fX3)*((fX2 * fX2) - (fX1 * fX1)) + (fX2 - fX1)*((fX3
                * fX3) - (fX1 * fX1)));
            float fB = ((fY2 - fY1) - (fA * ((fX2 * fX2) - (fX1 * fX1)))) /
                (fX2 - fX1);
            float fC = fY1 - (fA * (fX1 * fX1)) - (fB * fX1);

            float fY, fX;
            for (int i = iStart + 1; i < iMid; i++)
            {
                fX = i;
                fY = (fA * (fX * fX)) + (fB * fX) + fC;
                kCurve.Data.AddKey(fX, fY);
            }

            for (int j = iMid + 1; j < iEnd; j++)
            {
                fX = j;
                fY = (fA * (fX * fX)) + (fB * fX) + fC;
                kCurve.Data.AddKey(fX, fY);
            }
        }
    }
}

/**
 * Uses recursive bisection and cardinal evaluation to add back frames in
 * group
 * @param iStart the first frame of the group
```

```
* @param iEnd the last frame of the group
*/
private void _CardinalBisection(int iStart, int iEnd)
{
   // get the time key that bisects the group
   float fBisectTime = iStart + ((iEnd - iStart) / 2);
   float fMidPoint = 0.0f;

   // this is ugly, but guarentees the middle time we use has a key
   associated with it
   CsCurve kSampleCurve;
   if (ms_kOriginalJoints.TryGetValue(0, out kSampleCurve))
   {
      fMidPoint = kSampleCurve.Data.FindClosestKey(fBisectTime).Time;
   }
   else
   {
      CsCore.Log.Error("No middle frame found for self-avoidance, the
         impossible happened.");
      return;
   }

   // evaluate for each curve at fTime and add as a data key
   for (int iJoint = m_iLowerBounds; iJoint < m_iUpperBounds; iJoint++)
   {
      CsCurve kCurve;
      if (ms_kModifiedJoints.TryGetValue(iJoint, out kCurve))
      {
         // if key doesn't already exist, so if not the PD fixed frame
         if (kCurve.Data.GetKey(fMidPoint) == null)
         {
            kCurve.Data.AddKey(new CsCurveKey(fMidPoint,
            kCurve.Evaluate(fMidPoint)));
         }
      }
   }

   // now that the midpoint is fixed, get the previous and next points
   float fPrevPoint = fMidPoint - 1.0f;
   float fNextPoint = fMidPoint + 1.0f;

   // recursively check for collisions in new evaluated keys
   if (iStart != fMidPoint)
   {
      _CardinalBisection(iStart, (int)fPrevPoint);
   }
   if (iEnd != fMidPoint)
   {
      _CardinalBisection((int)fNextPoint, iEnd);
   }
}

/**
 * Updates a group to add buffered frame bounds based on preset
 * smoothing percentage
 * @param kGroup the collision group to buffer for smoothing
 */
 private _SelfCollision _BufferGroup(_SelfCollision kGroup)
```

```
{
    // set frame buffer as percentage of group size
    float fFrameBuffer = (kGroup.EndFrame - kGroup.StartFrame) *
        m_fSmoothing;
    fFrameBuffer = (float)Math.Round(fFrameBuffer, 0);

    CsCore.Log.Debug("Frame buffer of " + fFrameBuffer + " used");
    // if the buffer extends beyond first frame
    if (kGroup.StartFrame - fFrameBuffer < 0.0f)
    {
        kGroup.StartFrame = 0;
    }
    else
    {
        kGroup.StartFrame -= (int)fFrameBuffer;
    }

    // if the buffer extends beyond the last frame
    if (kGroup.EndFrame + fFrameBuffer > (ms_iMOCAPFrames - 1))
    {
        kGroup.EndFrame = (ms_iMOCAPFrames - 1);
    }
    else
    {
        kGroup.EndFrame += (int)fFrameBuffer;
    }

    return kGroup;
}

/**
 * Updates the plane constraints and runs posture prediction, updating
 * the modified curve
 * @param iFrame the frame to be used as a key value for the curve
 */
private void _RunPostureAvoidance(int iFrame)
{
    // update all involved plane constraints
    foreach (_PlaneConstraint kPlane in m_kPlaneConstraints)
    {
        kPlane.Update(iFrame);
    }

    // update all involved task constraints
    foreach (_ITaskConstraint kTask in m_kTaskConstraints)
    {
        kTask.Update(iFrame);
    }

    // run posture prediction
    CsPostureResults kResults = CsPosturePredict.Predict(ms_kAvatar);

    CsCore.Log.Debug("Ran Posture Prediction on frame " + iFrame);

    // for the upper body, replace the curve frame with posture results
    frame
    for (int iJoint = m_iLowerBounds; iJoint < m_iUpperBounds; iJoint++)
    {
```

```
            CsCurve kCurve;
            if (ms_kModifiedJoints.TryGetValue(iJoint, out kCurve))
            {
               kCurve.Data.GetKey((float)iFrame).Value =
                  (float)kResults.m_dPosture[iJoint].Value;
            }
         }
      }

      /**
       * Applies the posture at the specified frame in the mocap curves
       * @param   iFrame the frame in the curves to apply
       */
       private static void _ApplyCurvePosture(int iFrame, bool bMocap)
       {
          float[] fJointArray =
             CsCore.AvatarSystem.GetJointAngleArray(ms_kAvatar);

          // get each joint rotation and apply to the avatar
          for (int iJoint = 0; iJoint < 55; iJoint++)
          {
             CsCurve kCurve;
             if (bMocap)
             {
                if (ms_kOriginalJoints.TryGetValue(iJoint, out kCurve))
                {
                   fJointArray[iJoint] =
                      kCurve.Data.GetKey((float)iFrame).Value;
                }
             }
             else
             {
                if (ms_kModifiedJoints.TryGetValue(iJoint, out kCurve))
                {
                   fJointArray[iJoint] =
                      kCurve.Data.GetKey((float)iFrame).Value;
                }
             }
          }

          CsCore.AvatarSystem.SetJointAngleArray(ms_kAvatar, fJointArray);
       }

      /**
       * Checks the current avatar posture for sphere collisions
       * @return true if collisions, false if not
       */
      private bool _CollisionDetected()
      {
         // get the initial joint angles for the mocap frame
         CsArray<CsDouble> kInitialAngles =
            (CsArray<CsDouble>)ms_kAvatar.GetProp("JointAngle");

         if (CsPosturePredict.FindSelfCollisions(ms_kAvatar, kInitialAngles))
         {
            return true;
         }
```

```
            return false;
        }

        /**
         * Converts the selected motion capture file into a dictionary of curves
         * that can be evaluated
         * @return  The dictionary of curves that represents the MOCAP data
         */
        private SCG.Dictionary<int, CsCurve> _CreateAnimCurves()
        {
            SCG.Dictionary<int, CsCurve> kJoints =
                new SCG.Dictionary<int, CsCurve>();

            // initialize all of the joint curves
            for (int iJoint = 0; iJoint < 55; iJoint++)
            {
                kJoints.Add(iJoint, new CsCurve(new CsCurveData(), new
                    CsCurveEvaluatorLinear()));
            }

            // if we're creating the curves from a MOCAP file
            if (m_bMocap)
            {
                // for each frame in the mocap animation
                for (int iCurrentLine = 0; iCurrentLine < ms_iMOCAPFrames;
                iCurrentLine++)
                {
                    // skip to the current line
                    if (m_kMOCAPFile.PositionLine((uint)iCurrentLine + 2))
                    {
                        // the first 3 are global position
                        float fTx = Convert.ToSingle(m_kMOCAPFile.GetWord());
                        float fTy = Convert.ToSingle(m_kMOCAPFile.GetWord());
                        float fTz = Convert.ToSingle(m_kMOCAPFile.GetWord());

                        if (iCurrentLine == 0)
                        {
                            m_vGlobalPosition.Set(fTx, fTy, fTz);
                        }
                        fTx -= m_vGlobalPosition.X;
                        fTy -= m_vGlobalPosition.Y;
                        fTz -= m_vGlobalPosition.Z;

                        CsVector4f vPos =
                            CsCore.UnitManager.JointAnglesToPosition(fTx, fTy, fTz);
                        vPos += m_vPosition;

                        // next 3 are global rotation
                        float fRx = Convert.ToSingle(m_kMOCAPFile.GetWord());
                        float fRy = Convert.ToSingle(m_kMOCAPFile.GetWord());
                        float fRz = Convert.ToSingle(m_kMOCAPFile.GetWord());
                        CsVector4f vRot =
                            CsCore.UnitManager.JointAnglesToRotation(fRx, fRy, fRz);

                        float[] fGPR = { fTx, fTy, fTz, fRx, fRy, fRz };
                        for (int iGPR = 0; iGPR < 6; iGPR++)
                        {
                            CsCurve kCurve;
```

```
                if (kJoints.TryGetValue(iGPR, out kCurve))
                {
                    kCurve.Data.AddKey(new CsCurveKey((float)iCurrentLine,
                    fGPR[iGPR]));
                }
            }

            // step through all the joints in the mocap frame
            for (int iJoint = 6; iJoint < 55; iJoint++)
            {
                // get the joint rotation angle and try to add it to the
                respective curve
                float fJoint = Convert.ToSingle(m_kMOCAPFile.GetWord());
                CsCurve kCurve;
                if (kJoints.TryGetValue(iJoint, out kCurve))
                {
                    kCurve.Data.AddKey(new CsCurveKey((float)iCurrentLine,
                       fJoint));
                }
            }
        }
        else
        {
            CsCore.Log.Warning("Failed to position line within the text
               file");
        }
    }
}
else // if we're creating the curves from a spline file
{
    // for each frame in the mocap animation
    for (int iCurrentLine = 0; iCurrentLine < ms_iMOCAPFrames;
    iCurrentLine++)
    {
        // figure out the time for the evaluate
        float fTime = iCurrentLine * (m_kSpline.EndTime /
           ms_iMOCAPFrames);
        float fVTtoMM = (float)CsCore.UnitManager.VTtoMM * 1000;

        // the first 3 are global position
        float fTx = m_kSplineEval.Evaluate(1, fTime) * fVTtoMM;
        float fTy = m_kSplineEval.Evaluate(2, fTime)* fVTtoMM;
        float fTz = -m_kSplineEval.Evaluate(0, fTime) * fVTtoMM;

       CsVector4f vPos = CsCore.UnitManager.PositionToJointAngles
         (new CsVector4f(fTx, fTy, fTz));
        fTx = vPos.X;
        fTy = vPos.Y;
        fTz = vPos.Z;

        // the next 3 are global rotation
        CsVector4f vTemp = new CsVector4f();
        vTemp.X = -m_kSplineEval.Evaluate(4, fTime);
        vTemp.Y = -m_kSplineEval.Evaluate(5, fTime);
        vTemp.Z = m_kSplineEval.Evaluate(3, fTime);
        vTemp = VsCore.VtConvertFromDynamicsAngle(vTemp);
        vTemp = CsCore.UnitManager.RotationToJointAngles(vTemp);
```

```
            float fRx = vTemp.X;
            float fRy = vTemp.Y;
            float fRz = vTemp.Z;

            float[] fGPR = { fTx, fTy, fTz, fRx, fRy, fRz };
            for (int iGPR = 0; iGPR < 6; iGPR++)
            {
               CsCurve kCurve;
               if (kJoints.TryGetValue(iGPR, out kCurve))
               {
                  kCurve.Data.AddKey(new CsCurveKey((float)iCurrentLine,
                  fGPR[iGPR]));
               }
            }

            // step through all the joints
            for (int iJoint = 6; iJoint < 55; iJoint++)
            {
               // get the joint rotation angle and try to add it to the
               respective curve
               float fJoint = m_kSplineEval.Evaluate(iJoint, fTime);
               CsCurve kCurve;
               if (kJoints.TryGetValue(iJoint, out kCurve))
               {
                  kCurve.Data.AddKey(new CsCurveKey((float)iCurrentLine,
                  fJoint));
               }
            }
         }
      }

      return kJoints;
   }

   /**
    * Creates the task based constraints
    */
   private void _CreateTaskConstraints()
   {
      if (m_bWeapon)
      {
         _WeaponConstraint kWeaponConstraint = new _WeaponConstraint();
         m_kTaskConstraints.Add(kWeaponConstraint);
      }
   }

   /**
    * Destroys the task based constraints
    */
   private void _DestroyTaskConstraints()
   {
      foreach (_ITaskConstraint kTask in m_kTaskConstraints)
      {
         kTask.Destroy();
      }
   }

   /**
```

```
    * Creates the plane constraints that the avoidance methods will use
    * @return the list of plane constraints
    */
    private SCG.List<_PlaneConstraint> _CreatePlaneConstraints()
    {
        SCG.List<_PlaneConstraint> kList = new SCG.List<_PlaneConstraint>();

        if (m_bElbows)
        {
            _PlaneConstraint kElbowLeft = new _PlaneConstraint("Elbow_Left_1",
                eCsMEEJoints.Elbow_Left_1);
            _PlaneConstraint kElbowRight = new _PlaneConstraint("Elbow_Right_1",
                eCsMEEJoints.Elbow_Right_1);
            kList.Add(kElbowLeft);
            kList.Add(kElbowRight);
        }
        if (m_bWrists)
        {
            _PlaneConstraint kWristLeft = new _PlaneConstraint("Elbow_Left_1",
                eCsMEEJoints.Wrist_Left_1);
            _PlaneConstraint kWristRight = new _PlaneConstraint("Elbow_Right_1",
                eCsMEEJoints.Wrist_Right_1);
            kList.Add(kWristLeft);
            kList.Add(kWristRight);
        }

        return kList;
    }

    /**
    * Destroys all plane constraints
    */
    private void _DestroyPlaneConstraints()
    {
        foreach (_PlaneConstraint kPlane in m_kPlaneConstraints)
        {
            kPlane.Destroy();
        }
    }

    #endregion

    #region Helper Classes

    /**
    * Plane constraint class primarily used for updating constraints on a
    * frame by frame basis
    */
    private class _PlaneConstraint
    {
        private CsMarker kMarker;
        private CsEndEffector kEndEffector;
        private string _sName;

        /**
        * Constructor
        */
        public _PlaneConstraint(string sName, eCsMEEJoints eJoint)
```

```
{
    _sName = sName;
    // create the marker
    kMarker = new CsMarker();
    kMarker.ParentAvatar(ms_kAvatar.ID,
        CsCore.AvatarSystem.GetJointInfo(eJoint), null);
    kMarker.SetPosition(VsCore.AvGetBodyPartPos(ms_kAvatar.ID, sName,
        true), true);
    kMarker.SetProp("Name", new CsString(sName + "_Marker"));
    kMarker.SetProp("Marker Type", new CsEnum(eCsConstraintType.Plane));
    kMarker.SetProp("Rotation", new CsVectorVariant(new CsVector4f(0.0f,
        0.0f, 0.0f)));
    CsCore.World.Add(kMarker);
    // create the end effector
    kEndEffector = new CsEndEffector();
    kEndEffector.ParentAvatar(ms_kAvatar.ID,
        CsCore.AvatarSystem.GetJointInfo(eJoint), null);
    kEndEffector.SetPosition(VsCore.AvGetBodyPartPos(ms_kAvatar.ID,
        sName, true), true);
    kEndEffector.SetProp("Name", new CsString(sName + "_EE"));
    kEndEffector.SetProp("Target", new CsID(kMarker.ID));
    kEndEffector.SetProp("TargetType", new
        CsEnum(eCsTargetType.Object));
    CsCore.World.Add(kEndEffector);
}

/**
 * Updates the constraint based on the frame
 */
public void Update(int iFrame)
{
    // apply the joint rotations at the next frame (previous if last
    frame)
    _ApplyCurvePosture(((iFrame == (ms_iMOCAPFrames - 1)) ? (iFrame - 1)
    : (iFrame + 1)), false);
    CsVector4f vEnd = VsCore.AvGetBodyPartPos(ms_kAvatar.ID, _sName,
        true);

    _ApplyCurvePosture(iFrame, false);
    CsVector4f vStart = VsCore.AvGetBodyPartPos(ms_kAvatar.ID, _sName,
        true);

    CsVector4f vDirection = vEnd - vStart;

    kMarker.SetProp("Rotation", new CsVectorVariant(vDirection));
}

/**
 * Destroys the plane constraint
 */
public void Destroy()
{
    kMarker.Node.Dispose();
    kMarker = null;

    kEndEffector.Node.Dispose();
    kEndEffector = null;
}
```

```
}

/**
 * Self collision class used to store start and end times of collision
 * groups
 */
private class _SelfCollision
{
   private int _iStartFrame;
   private int _iEndFrame;

   // Constructor
   public _SelfCollision(int iStart, int iEnd)
   {
      _iStartFrame = iStart;
      _iEndFrame = iEnd;
   }

   public int StartFrame
   {
      set { _iStartFrame = value; }
      get { return _iStartFrame; }
   }

   public int EndFrame
   {
      set { _iEndFrame = value; }
      get { return _iEndFrame; }
   }
}

#region Task Constraints

/**
 * Interface to allow for a generic call to all task constraints update
 * and destroy
 */
interface _ITaskConstraint
{
    void Update(int iFrame);
    void Destroy();
}

/**
 * Weapon constraint for tasks
 */
private class _WeaponConstraint : _ITaskConstraint
{
   private CsMarker kMarker;
   private CsEndEffector kEndEffector;

   // constructor
   public _WeaponConstraint()
   {
      ms_kAvatar.SetProp("LeftHandDirection", new
         CsEnum(eCsHandDirectionType.Hand_Direction));
      _ApplyCurvePosture(ms_iMOCAPFrames - 1, true);
```

```
            // create the marker
            kMarker = new CsMarker();
            kMarker.ParentAvatar(ms_kAvatar.ID,
               CsCore.AvatarSystem.GetJointInfo
               (eCsMEEJoints.FingerMiddle_Right1_1), null);
            kMarker.SetPosition(VsCore.AvGetBodyPartPos(ms_kAvatar.ID,
               "FingerMiddle_Left1_1", true), true);
            kMarker.SetProp("Name", new CsString("Weapon_Marker"));
            kMarker.SetProp("Marker Type", new
               CsEnum(eCsConstraintType.BodyPoint));
            CsCore.World.Add(kMarker);
            // create the end effector
            kEndEffector = new CsEndEffector();
            kEndEffector.ParentAvatar(ms_kAvatar.ID,
               CsCore.AvatarSystem.GetJointInfo
               (eCsMEEJoints.FingerMiddle_Left1_1), null);
            kEndEffector.SetPosition(VsCore.AvGetBodyPartPos(ms_kAvatar.ID,
               "FingerMiddle_Left1_1", true), true);
            kEndEffector.SetProp("Name", new CsString("Weapon_EE"));
            kEndEffector.SetProp("Target", new CsID(kMarker.ID));
            kEndEffector.SetProp("TargetType", new
               CsEnum(eCsTargetType.Object));
            CsCore.World.Add(kEndEffector);
         }

         /**
          * Updates the constraint based on the frame in the MOCAP
          * NOTE: Not needed for this constraint.
          */
          public void Update(int iFrame) { }

         /**
          * Destroys all world objects used to create the constraint
          */
          public void Destroy()
          {
             ms_kAvatar.SetProp("LeftHandDirection", new
                CsEnum(eCsHandDirectionType.None));

             kMarker.Node.Dispose();
             kMarker = null;

             kEndEffector.Node.Dispose();
             kEndEffector = null;
          }
      }

      #endregion

      #endregion
   }
}
```