uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Jean-Christian Delannoy
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Modeling Articulated Bodies with Distributed Mass for Virtual Interactive Environments

TITRE DE LA THÈSE / TITLE OF THESIS

E.M. Petriu
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

N. Georganas

G. Wainer

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

MODELING ARTICULATED BODIES WITH DISTRIBUTED MASS FOR VIRTUAL INTERACTIVE ENVIRONMENTS

by

Jean-Christian Delannoy

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M.A.Sc. degree in Electrical Engineering

Ottawa-Carleton Institute for Electrical & Computer Engineering
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-18411-0
Our file Notre référence
ISBN: 978-0-494-18411-0

Abstract

Dynamic simulation is a powerful application of contemporary computers, it has uses that range from design and manufacturing to simulation and entertainment products. This work investigates dynamic simulation of articulated bodies in interactive virtual environments. Previous techniques for both modeling articulated bodies in virtual environments as well as simulating their motion are qualitatively compared for the twin goals of correctness of results and computational efficiency. Implementations of the Verlet system and the method of stiff springs for articulated bodies are quantitatively compared when used for bodies with segments of varied mass in interactive virtual environments. Finally, a novel adaptation of the Verlet technique for simulating constrained and articulated bodies is presented. Through this adaptation to the Verlet algorithm we are able to simulate bodies with links of inconsistent mass more faithfully than with previous implementations.

Table of Contents

List of Figures

Chapter 1

Introduction

Recent advances in graphics technologies and computing power have made possible the recent boom in virtual reality applications. Highly detailed 3D computer models and environments are now commonly used in design, manufacturing, simulation and entertainment products. Large-scale studios are releasing movies consisting entirely of computer animation, a form of virtual environment. The 3D video game industry that is based almost entirely on virtual environments is booming and now generates more revenue than the movie industry in North America. Aerospace companies are able to create new vehicles without ever creating a single practical prototype because they can test virtual versions of their designs using virtual prototypes in virtual environments. Training new pilots now costs a fraction of the previous cost thanks to relatively inexpensive virtual cockpits that do not consume a single drop of gas instead of million dollar jets. These are all considered virtual environments since one of their primary objectives is always to immerse the user to the point of practically forgetting they are in a synthetic world.

Virtual environments (VE's) have come a long way thanks to sustained advances in the field of computer rendering of 3D images. Contemporary computing equipment can render millions of polygons, one of the basic building blocks of 3D environments, in a fraction of a second and the work created is getting closer and closer to being photo realistic. However, while these environments are often successful at appearing realistic when time remains still, as the mobile 3D objects are set into motion and begin touching and interacting with their surroundings, their motions often appear unnatural. This brings us to the field of physical modeling, or dynamic simulation, the technology used to mimic the behaviors and reactions of real world objects in virtual environments. In interactive environments, users want the ability to interact with as much of the world as possible. If a user kicks a virtual box on the floor, they want that box to slide on the floor and slowly come to a stop due to friction. If a user throws a ball in the air, he wants it to slowly decelerate and fall right back on to the ground. Since it would be impossible to predict all of the different ways in which a user would interact with

his virtual environment, it becomes necessary to implement laws into the virtual environment that manage how objects will move and interact with each other, much like the physical laws of the real world. All of these reactions and interactions with virtual objects must be an accurate representation of what would occur in the real world if the user is to believe that the environment is real, or for the user to become immersed in the virtual environment. While the field of rendering has made magnificent progress in the last little while, physics modeling is trailing behind. Many virtual environments still display flagrant problems due to insufficient physical modeling. Some of the symptoms include objects going through one another, and objects reacting improperly when pushed. These problems are quite noticeable since all of us as human beings are experts at identifying when objects do not correctly mimic their motions in our world because we see all interactions on a daily basis.

The first solution to the problem of creating realistic motion in virtual interactive environments is to have a library of pre-determined (or scripted) movements for all objects. In this solution which is still in use today for many applications, an animator will create a library of movements that an object can do. All of the objects will then become interactive but only for the specific set of pre-determined movements. For example, a box in the virtual environment could have movements defined for sliding and for being flattened. The box has only 2 behaviors. If a user in the virtual environment makes use of a tool to squash the box, the movement of the box being flattened can be played back. However, if the user attempts to do anything else with the box, such as cutting it in half, the box will not react to the action. In whatever way a user decides to interact with the box, it will only ever be able to react using one of those 2 defined options. This process does create results that appear correct in certain cases, but it also has many drawbacks:

- **All motion must be hand-keyed by an animator.**

To create the motion using this solution it is necessary for an animator to create keyframes for the motion of the object using a 3D editing tool. This process is extremely time-consuming since the object's location must be manually specified at as many points in time as possible.

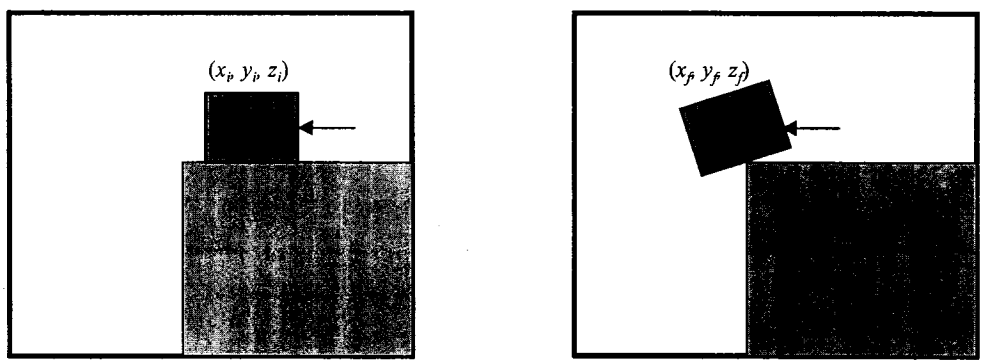


Figure 1.1. Animator-specified location of the box at various times.

This process has been in use for many years in different areas but as our virtual environments become more and more complex it will become very difficult to scale this solution to the very large amount of objects we wish to simulate and also the many ways in which we want to allow a user to interact with objects within the virtual environment.

- **The motion created does not rigorously follow our physical laws.**

The physical world around us that we are attempting to mimic follows very strict physical laws. These laws include gravity, friction and inertia and all have a bearing on how objects move and interact with the rest of the world. While a good animator will have the ability to approximate all of these laws, he will never be able to precisely reproduce in a virtual environment the actual trajectories of moving objects, as they would move in the real world.

- **Motions are all scripted.**

Since all motion is scripted, i.e.: an animator has specified where a specific object would be at a specific time in a specific environment, the motion can only be applied in this one specific case. If either the object or the location were slightly different, it would be necessary to author another set of data. A consequence of having scripted motion is that it is very difficult to apply to an interactive virtual environment. To apply scripted motion to an interactive environment would require having a very vast library of reactions for every interactive object in the environment.

Due to these limitations, creating scripted motion in VE's (virtual environments) is practically never used for rigid body motion in interactive environments. Dynamically simulating constrained and articulated bodies is much more complex however, and so many applications choose to not simulate their articulated bodies and use a solution which makes use of scripted motion instead. Virtual environments need to become much more adaptive and dynamic if they want to truly immerse their users within their synthetic environment. As an alternative to scripted methods, this thesis will begin by demonstrating ways to dynamically simulate the motion of objects in virtual interactive environments. This approach allows motion to be generated automatically in real-time without knowledge of the configuration of the environment.

While the problems with rigid bodies are well-understood, virtual environments have now evolved to the point of having all sorts of cloth (flags, shirts) and articulated bodies such as strings, cranes and robotic arms. This thesis will present and discuss approaches for modeling articulated bodies with distributed mass for virtual interactive environments. An *articulated* body will be the opposite of a *rigid* body. Articulated bodies are 3D objects that contain at least one joint, in opposition to rigid bodies where each part of the object will maintain the same relative position to the object's origin at all times.

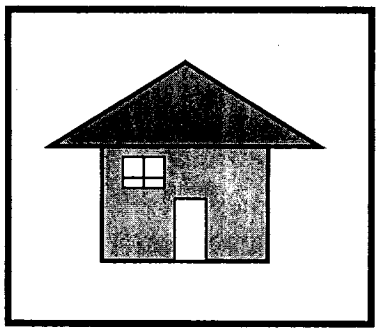


Figure 1.2a. A rigid object.

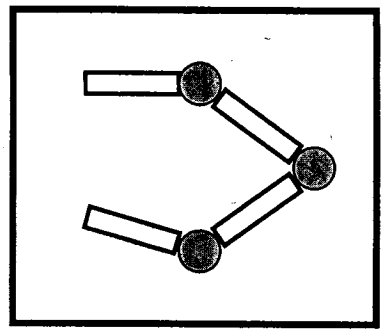


Figure 1.2b. An articulated object.

This thesis will also present techniques for modeling the mass of rigid and articulated objects, and for dynamic simulation of the models shown. An object with distributed mass is the opposite of an object represented as a point mass. In *distributed mass* modeling, the mass of an object is distributed over its entire volume. In *point mass* modeling, a single point in 3D and a scalar representing the quantity are enough to define the mass of an object.

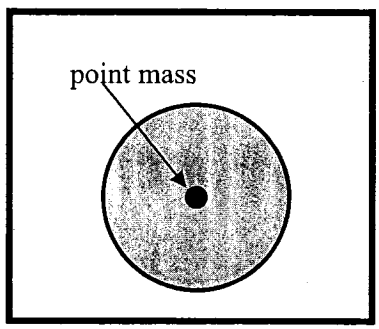


Figure 1.3a. A ball modeled using a point mass.

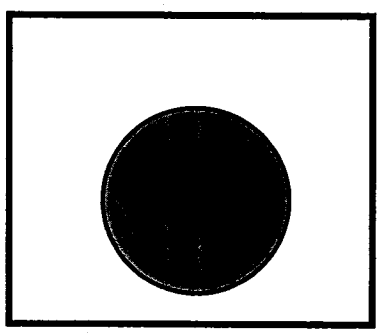


Figure 1.3b. A ball modeled using a mass distributed over the ball's volume.

The thesis will begin by presenting a method for physically based modeling and simulation of rigid bodies before moving on to the state of the art for dynamic simulation of articulated bodies in interactive virtual environments. Four different techniques for articulated bodies will be presented and discussed:

- Verlet integration combined with Gauss-Seidel iteration for solving constraints
- The method of stiff springs
- Lagrange multipliers (or using the constraint Jacobian matrix)
- Featherstone's algorithm.

Each of these methods will be analyzed; their advantages and disadvantages when implemented for interactive virtual environments will be discussed. In the last chapter, a novel adaptation of the Verlet technique for simulating constrained and articulated bodies is presented. Through this adaptation to the Verlet algorithm we are able to simulate inconsistent mass throughout articulated bodies much more faithfully than with the previous implementations. The adaptation concerns the step of constraint resolution which is a very important

part of dynamic simulation for articulated bodies; it is the resolution of constraints that is responsible for keeping the various links of the body connected together amongst other things. In previous implementations of the Verlet system, the constraints were solved without any regard for the weight of the limbs, making it impossible to simulate faithfully an articulated body with varying mass throughout. Our new implementation takes into account the mass of each link separately when it is resolving the physical constraints of the system; this allows us to simulate things such as a chain with a large mass at the end with much more believable results. To complete, a comparison of experimental results obtained through the method of stiff springs and the novel adaptation of Verlet integration will be made. The two techniques will be compared for the believability of the output, computational complexity and the capacity to simulate various types of constraints. Each comparison will be made in the context of virtual interactive environments.

The thesis will focus on physical simulations applied to entertainment products such as 3D animation or video games, and will therefore advantage quickness of computation and correct cosmetic look over strict accuracy or preciseness of the results. Most 3D animation tools and computer video games need to run in real-time on a wide array of desktop computers. Because of this, the algorithms and simulation techniques that they use must be as optimal as possible; both in the amount of memory that is used and accessed, and in the number of CPU cycles required to run the simulation. Note that the amount of memory that is used is almost as important as the number of CPU cycles required; this is due to the fact that a single cache data miss on modern computers can results in hundreds of wasted cycles as that data is brought in from long-term storage all the way to the CPU's registers. This emphasis on performance will hit accuracy the most since many of the algorithms will use certain approximations in an effort to speed up the computations, in doing so also reducing the accuracy of the simulation.

The main competitor of physical simulations in 3D virtual environments for entertainment products is key-framed animation. The technique in which an animator will specify the position and orientation of the 3D objects at various points in time (keyframes); these can then be replayed in the VE to give the illusion of real motion. This technique has some advantages and some disadvantages. One of the important advantages is that it is very simple to implement and does not consume many CPU cycles. Some of the disadvantages are that it consumes lots of memory and is very difficult to adapt to different situations. Because every single motion is created beforehand by an animator, all of this data must be kept in memory so that it can be retrieved at any time. The other disadvantage is that the objects can only move and react in ways that have been previously specified by the animator. If we take the example of a virtual man being hit by a ball and falling to the ground; an animator could animate the man being hit by a ball coming from the front and then the man falling to the ground. This animation could be played in the VE if the man is hit by a ball from the front, but if he were hit by a ball from behind him he would not know how to react; doing nothing at all or playing the animation with the ball coming from the front (which would look ridiculous). Physical simulations on the other hand allow for

objects to react realistically to any situation in their environment. They use more CPU cycles to update the simulation but also use very little memory.

Chapter 2

Physically Based Modeling for Rigid Bodies

In contrast to virtual reality, where synthetic environments are created without any regard to the real world, *Virtualized Reality* is a synthetic reality where object properties, attributes, and reactions are based on sensor data gathered from the real world. These properties give the synthetic environment a realism which is often very difficult to achieve when objects are designed and conceived without real world counterparts. While *virtualized reality* environments are always held back to a certain extent by the preciseness and granularity of modern data sensors to model the world around us exactly, it remains that a faithful representation can be achieved using a subset of the physical laws and properties of our surroundings.

This section will begin by presenting techniques and models for representing real world objects in virtual environments. While it is still not possible (or necessary) to represent an object precisely in a virtual environment since each object would need to be represented by the millions of atoms of which it is composed, we will choose a model of the object which will provide results which are faithful enough for most simulations. This section will then go on to demonstrate techniques which make use of the model provided to dynamically simulate how the objects would move and interact with the environment surrounding them.

2.1 Modeling of Object Properties

All of the objects in our world have their own properties. Examples of these properties are an object's mass, its elasticity, and its shape. These properties are quite important since they are the starting point to determine how objects will move and react with their surroundings.

It is both impossible and uneconomical to model the totality of the physical properties of objects. Scientists are at the stage of discovering object properties at a sub-atomic level; this is a much finer grain than what is needed for most of our virtual environments. Supposing it would even be possible to physically model the entirety of an object's properties, it would be practically impossible to dynamically simulate such an environment interactively and in real-time using state-of-the-art computers. It is therefore necessary to choose a subset of the attributes of objects and only model these with the level of detail sufficient for our needs. Using simple models to represent complex entities will be a recurring theme when attempting to simulate environments at interactive rates.

2.1.1 Position

The position consists of a 3-element vector representing the location in the world relatively to the world's origin.

2.1.2 Orientation

In three-dimensional space, objects have three degrees of freedom (DOF) for their rotation. Since any orientation has three DOF's, any representation will have at least 3 scalars. The two most popular methods for representing an object's orientation in a 3D environment are quaternions and Euler angles.

With Euler angles, the three scalars represent the angles of rotation around the three world axes. These angle triplets are referred to as *Euler* angles. The three axes are also referred to as the *roll*, *pitch* and *yaw* axes, corresponding respectively to the rotation around the x, y, and z-axes.

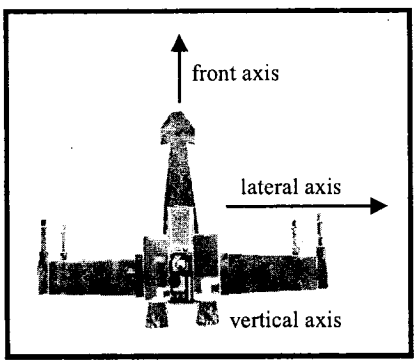


Figure 2.1a. Relative axes.

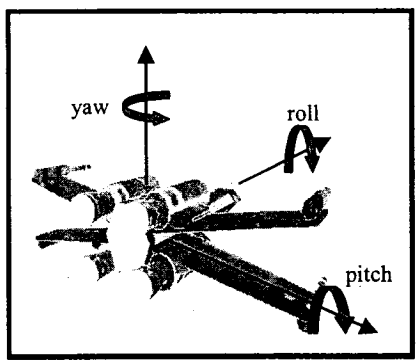


Figure 2.1b. Roll, pitch and yaw.

Quaternions were born from the realization that any orientation of the object's local basis can be created by a single rotation around an arbitrary axis. A quaternion is a quadruple of the form:

$$\mathbf{q} = \alpha_0 + \alpha_1 i + \alpha_2 j + \alpha_3 k \tag{2.1}$$

The quaternion behaves as a four-dimensional vector space with respect to addition and scalar multiplication. Its algebra is a generalization of complex numbers. See [Fon00] for further discussion and analysis about these two techniques for representing orientation.

2.1.3 Mass

The mass of an object will be modeled by a single value at the center of mass (center of gravity) for the object. The total mass of a body is simply the sum of the masses of all particles making up the body, where the mass of each particle is obtained as its mass density multiplied by its volume. By simplifying complicated bodies with non-uniform mass distribution by a single total mass, we eliminate many calculations and still achieve comparable results. This thesis will deal with mass modeling of articulated bodies in later chapters.

The center of gravity of an object is a vector where each coordinate is obtained by doing a weighted average of the particle's center of gravity along with their respective weights (2.2).

Center of gravity: (x_c, y_c, z_c)

$$x_c = (\sum x_i m_i) / \sum m_i \qquad (2.2)$$

$$y_c = (\sum y_i m_i) / \sum m_i$$

$$z_c = (\sum z_i m_i) / \sum m_i$$

2.1.4 Restitution Coefficient

When objects collide with one another, there is always energy lost due to deformations caused in both objects. This is what happens when a vehicle crashes into a wall. The moving vehicle is full of kinetic energy before the collision and has none after the collision; the energy of the moving vehicle is used to deform the vehicle itself; the deformation dissipates the object's kinetic energy. Our simulation simplifies this problem by having non-deformable bodies which retain their shapes after collisions. Witkin and Baraff present techniques for deformable bodies in [BW92]. However, the model must still represent the capacity of objects to absorb impact; otherwise two vehicles crashing into one another will simply bounce off each other. The restitution coefficient of an object will be a scalar between 0 and 1, which represents the amount of energy the object loses on collisions. A value near 0 will mean that the object will absorb lots of energy during collisions, much like a pillow; a value near 1 will mean that the object will absorb very little energy during a collision, like a rubber ball.

2.1.5 Linear Drag Coefficient

The linear drag coefficient determines the magnitude of the air friction along the object's front axis (i.e.: a fighter jet would have a low linear drag coefficient since it is very aerodynamic). It consists of a negative number between 0 and –1 and is multiplied by the component of the velocity parallel to the object's front axis to obtain the drag. This drag is then applied to the object to modify its trajectory and speed.

2.1.6 Orthogonal Drag Coefficient

The orthogonal drag coefficient determines the magnitude of the air friction along the object's right and top axes. The orthogonal drag coefficient also consists of a negative number between 0 and –1 and is multiplied by the component of the velocity along the object's front and top axes. The orthogonal drag coefficient is considered separately from the linear drag coefficient to account for the anisotropic property of objects which are much more aerodynamic when moving in certain specific directions.

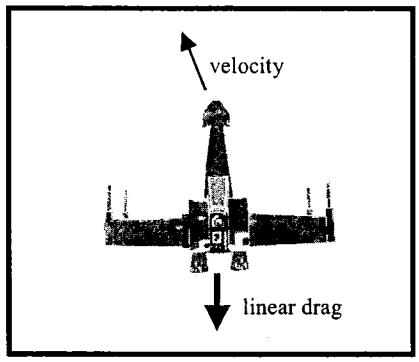


Figure 2.2a. Linear drag.

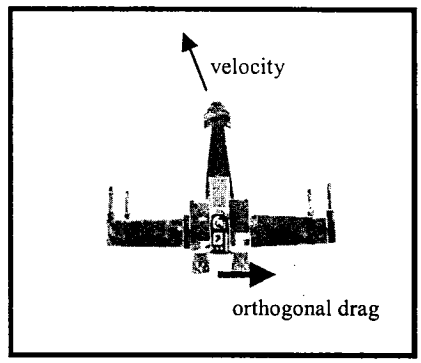


Figure 2.2b. Orthogonal drag.

Any sports car or fighter jet will have a very low linear drag coefficient, meaning that little energy is lost due to friction as it moves forward, but will usually have much higher orthogonal drag coefficients. The linear and orthogonal drag coefficients are actually only approximations used to compute the drag on the object. A more accurate representation would require drag coefficients from many different perspectives on the vehicle since it will always be different as it moves in various directions. The linear/orthogonal model however, is sufficient for most purposes.

2.1.7 Moment of Inertia

The *moment of inertia* is the name given to rotational inertia, the rotation analog of mass for linear motion such as defined in Newton's second law. It relates the torque acting on a rigid body to the body's angular acceleration:

$$\tau = I\dot{\omega} \tag{2.3}$$

τ : Total torque applied ω : Angular acceleration I : Inertia tensor

Any moment of inertia must be defined with respect to an axis of rotation. A point mass creates a moment of inertia with respect to an axis that is equal to the product of the mass times the distance from the axis squared.

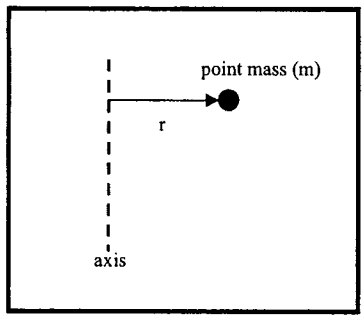


Figure 2.3. Inertia is equal to $I = mr^2$

Of course, in a virtual 3D world a body can rotate around any arbitrary axis. The angular velocity of most bodies will also have anisotropic properties: the moment of inertia will vary in relation to the axis around which it is rotating. Therefore it is incorrect to model the moment of inertia by a single scalar value. Instead a 3x3 matrix called the *inertia tensor* [BW97] will be required.

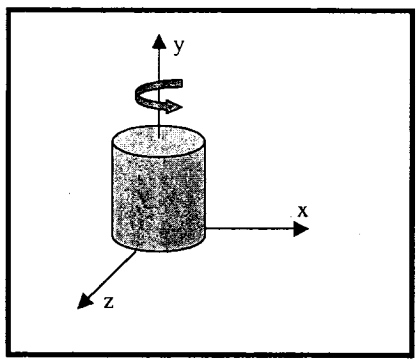


Figure 2.4a. A cylinder turning around the y-axis.

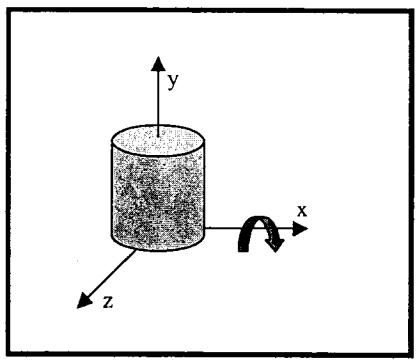


Figure 2.4b. A cylinder turning around the x-axis. (the moment of inertia will be different than in a)

To derive the formula for the inertia tensor we begin by looking at the angular moment of a rigid body composed of a set of point masses. The angular momentum is equal to the sum:

$$\mathbf{L}(t) = \sum \mathbf{r}_i(t) \wedge \mathbf{p}_i(t) \tag{2.4}$$

$\mathbf{L}(t)$: Angular momentum $\mathbf{p}(t)$: Linear momentum

Where $r_i(t)$ represents the position of the i^{th} particle, $p_i(t)$ represents the momentum of the i^{th} particle, and the summation is taken over all the particles belonging to the system. The linear momentum can then be replaced by:

$$\mathbf{p}_i(t) = m_i \mathbf{v}_i(t) = m_i \omega(t) \wedge \mathbf{r}_i(t) \tag{2.5}$$

This will yield the following expression for the angular momentum:

$$\mathbf{L}(t) = \sum_{i=1}^{n} m_i \mathbf{r}_i(t) \wedge [\omega(t) \wedge \mathbf{r}_i(t)] \tag{2.6}$$

Using a well known vector identity

$$\mathbf{P} \times (\mathbf{Q} \times \mathbf{P}) = \mathbf{P} \times \mathbf{Q} \times \mathbf{P} = \mathbf{P}^2\mathbf{Q} - (\mathbf{P} \cdot \mathbf{Q})\mathbf{P} \tag{2.7}$$

This can then be written in the form:

$$\mathbf{L}(t) = \sum_{i=1}^{n} m_i \left(\mathbf{r}_i^2(t)\omega(t) - [\mathbf{r}_i(t) \cdot \omega(t)] \cdot \mathbf{r}_i(t)\right) \tag{2.8}$$

The *i*-th component of L can be written:

$$\mathbf{L}_i = \sum_n m_n \left[\mathbf{r}_n^2 \omega_i - (\mathbf{r}_n)_i \sum_{j=1}^{3} (\mathbf{r}_n)_j \omega_j \right] \tag{2.9}$$

Making use of the *Kronecker delta* (δ) we are allowed to write the formula as:

$$\mathbf{L}_i = \sum_n m_n \sum_{j=1}^{3} \left[\mathbf{r}_n^2 \omega_j \delta_{ij} - (\mathbf{r}_n)_i (\mathbf{r}_n)_j \omega_j \right] \qquad \text{Where} \quad \omega_i = \sum_{j=1}^{3} \omega_j \delta_{ij} \tag{2.10}$$

$$\mathbf{L}_i = \sum_{j=1}^{3} \omega_j \sum_n m_n \left[\delta_{ij} \mathbf{r}_n^2 - (\mathbf{r}_n)_i (\mathbf{r}_n)_j \right]$$

The sum over *n* will be the (*i, j*) entry in the 3×3 inertia matrix.

$$\mathbf{I}_{ij} = \sum_n m_n \left(\delta_{ij} \mathbf{r}_n^2 - (\mathbf{r}_n)_i (\mathbf{r}_n)_j \right) \tag{2.11}$$

This then allows us to express the angular momentum as:

$$\mathbf{L}_i = \sum_{j=1}^{3} \omega_j \mathbf{I}_{ij} \qquad\qquad \mathbf{L}(t) = \mathbf{I}\omega(t) \tag{2.12}$$

The inertia tensor, *I*, used in Newton's second law is computed as follows:

$$I = \begin{bmatrix} \sum_{i=1}^{n} m_i (y_i^2 + z_i^2) & -\sum_{i=1}^{n} m_i x_i y_i & -\sum_{i=1}^{n} m_i x_i z_i \\ -\sum_{i=1}^{n} m_i x_i y_i & \sum_{i=1}^{n} m_i (x_i^2 + z_i^2) & -\sum_{i=1}^{n} m_i y_i z_i \\ -\sum_{i=1}^{n} m_i x_i z_i & -\sum_{i=1}^{n} m_i y_i z_i & \sum_{i=1}^{n} m_i (x_i^2 + y_i^2) \end{bmatrix} \tag{2.13}$$

The diagonal entries in the matrix are the *moments of inertia* in relation to the *x*, *y* and *z* coordinate axes. The off-diagonal entries are called the *products of inertia*.

2.1.8 Velocity

The velocity ($\mathbf{v} = \dot{\mathbf{x}}$) of an object is represented as a 3-element vector in the virtual world. The $\dot{x}$, $\dot{y}$ and $\dot{z}$ components of the vector represent the object's current velocity along the x, y and z axis. The vector will be in distance units over time.

2.1.9 Angular Velocity

The angular velocity of an object is represented as a vector. The x, y and z components describe the speed at which the object is rotating around its longitudinal, lateral and vertical axes respectively.

2.2 Dynamic Simulation of Rigid Bodies

The previous sections outlined the techniques and methods used when modeling real world objects in virtual environments. This section will describe methods that use the model provided in an effort to simulate the dynamics of the environment.

When dealing with rigid bodies in virtual environments, collisions between objects are an important aspect to keep in mind. There are two parts to the problem; the first is to detect that two objects are attempting to occupy the same volume, resulting in a collision; this is known as *collision detection*. The second part consists of computing the response that both objects will have to the new collision; this is known as *collision response*. Both of these areas are currently very active research areas [BW92, Bar89, Van04, Bou02] and will not be covered in depth here.

2.2.1 Simulation Loop

At the core of every interactive virtual reality application is a loop that gets executed over and over while the program is running. This loop manages everything from the creation of objects in the world, processing the input from the user, updating/moving the objects and finally displaying the objects on screen. The speed at which this loop executes will determine how many frames per second can be displayed on the screen. Dealing with such repetitive tasks is the strength of modern computers and this structure for interactive environments has been around since the early days of computer simulations; it continues to be in use today.

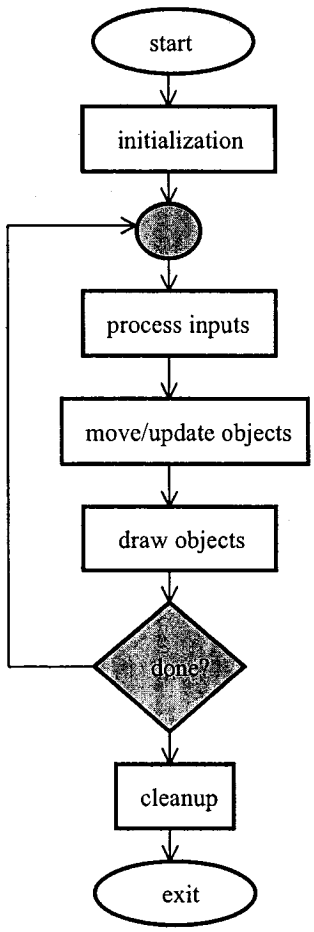


Figure 2.5. A typical program loop.

The processing input stage will observe the state of the apparatus which is used for interfacing with the virtual environment. For example, if a user could interact with the objects in the world using a keyboard, the processing input state will simply observe if any of the keys have been pressed. This is also the stage where virtual interactive forces will be applied on certain objects. For example, in a virtual environment which contains a spaceship that is controlled via the spacebar; the processing input stage will apply a virtual force to the spaceship when the spacebar is pressed down.

The move/update stage is responsible for summing all of the forces which were applied during the previous stage. It will also apply automatically occurring forces from the environment such as gravity and friction. The summed forces on each object will then be used to update the velocities, orientations and positions of each object. Constraints are also taken into account as the objects are moved and updated; otherwise non-realistic motion will be generated. If many articulated bodies are in the simulation, resolution of the constraints will occupy a large portion of this stage.

Once the positions and orientations of the objects have been updated, these can be drawn to the screen to be viewed by the users. This regular drawing of objects to the screen (once every program loop) is what gives the impression of constant and regular motion to the user.

If the simulation is done, it will perform cleanup routines at the following stage on the executing machine. If the simulation is not done, it will return to the stage of processing input again, thus restarting the loop.

While this structure has been around for many years, it is beginning to disappear in favor of structures which allow for multi-threading more easily, such as in [BW98].

2.2.2 Particle Dynamics

We begin by presenting simulation techniques for particles; a particle being an infinitesimally small object with a position and velocity but no orientation or angular velocity. The *state* of a particle at any time consists of the variables which can be modified by the dynamic simulation; in our case these are the position ($\mathbf{x}(t)$) and velocity ($\mathbf{v}(t)$) of the particle. These are concatenated in what is known as the *state vector* at time *t*:

$$\mathbf{Y}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{v}(t) \end{bmatrix} \tag{2.14}$$

For systems which contain many particles, the state vector for the system will be composed of the positions and velocities of each of the particles, all queued one after the other in the same vector.

$$\mathbf{Y}(t) = \begin{bmatrix} x_1(t) \\ v_1(t) \\ \dots \\ x_n(t) \\ v_n(t) \end{bmatrix} \tag{2.15}$$

The simulation will consist of two main steps. During the first step, all of the forces acting on the particle will be computed and the result will be used to compute the acceleration affecting the particle using Newton's equation relating force to acceleration ($f = ma$). During the second step, the acceleration and current velocity of the particles will be fed into a differential equation solver which will output the new position and velocity of the particle. Newton's second law of dynamics ($f = ma$) is used as a basis for the entire simulation. In the simulation, the mass of the particle is modeled as a simple scalar and the force is defined as a vector sum of all forces affecting the particle at time *t*. The aforementioned forces are all taken from our virtual world; examples of these could consist of gravity, friction, constraints, springs, etc. It is the intention of the simulation to

compute the new positions of the particles at any time. The accelerations will be computed from the total force acting on the particles; this will then be used to obtain the new velocity and position at the next time step.

The change in the state vector *Y(t)* over time is given by (2.16).

$$\frac{d}{dt}\mathbf{Y}(t) = \frac{d}{dt}\begin{bmatrix}\mathbf{x}(t)\\ \mathbf{v}(t)\end{bmatrix} = \begin{bmatrix}\mathbf{v}(t)\\ \mathbf{F}(t)/m\end{bmatrix} \tag{2.16}$$

Using this equation, we will have the ability to know the change in velocity and position of the particle at time *t*. The system will need to be provided with the initial conditions to properly obtain the new coordinates of the particle. The actual implementation or technique to track the change of *Y(t)* over time will be done by a numerical differential equation solver (many of these will be presented later on).

2.2.3 Rigid Body State Vector

A rigid body differs from a particle in many respects. To begin, it has a volume which must be modeled by collision geometry. A rigid body is also defined by its orientation and angular velocity. The concept of state vector used for particle dynamics will also be utilized for tracking the movement and configuration of rigid bodies, the only difference will lie in the information contained in the state vector.

A particle in the world can only be translated from its origin whereas a rigid body can not only be translated but can also undergo rotations. This will introduce two new variables required to model the state of the object: the current orientation and the angular velocity. To model the location of the body in world space we will continue to use *x(t)* for the translation of the body; the rotation will be described by a 3x3 matrix *R(t)*. These variables (*x(t)* and *R(t)*) are the *spatial variables* of the rigid body, the variables that define the current configuration of the object in the environment. *R(t)* will specify the rotation of the body around the center of mass. A fixed vector *r* in body space will be at the vector *R(t)r* at time *t* after rotation. For any point on the rigid body, we will obtain the new position by first executing the displacement caused by rotation followed by the movement caused by linear translation. In the following equation, the first operand of the addition consists of the movement caused by rotation and the second operand is the movement caused by linear translation (2.17).

$$\mathbf{r}(t) = \mathbf{R}(t)\mathbf{r}_0 + \mathbf{x}(t) \tag{2.17}$$

The velocity $\dot{r}(t)$ of the point is obtained by differentiation of this equation with respect to time (2.18).

$$\begin{aligned}\dot{\mathbf{r}}(t) &= \omega(t) * \mathbf{R}(t)\mathbf{r}_0 + \mathbf{v}(t) \\ &= \omega(t) * \big(\mathbf{R}(t)\mathbf{r}_0 + \mathbf{x}(t) - \mathbf{x}(t)\big) + \mathbf{v}(t) \\ &= \omega(t) * \big(\mathbf{r}(t) - \mathbf{x}(t)\big) + \mathbf{v}(t)\end{aligned} \tag{2.18}$$

$$= \omega(t) \times \left(\mathbf{r}(t) - \mathbf{x}(t)\right) + \mathbf{v}(t)$$

Decomposition of the final equation demonstrates how the velocity consists of two components:

- the linear component $\mathbf{v}(t)$
- the angular component $\omega(t) \times \left(\mathbf{r}(t) - \mathbf{x}(t)\right)$

2.2.4 Linear Momentum

To obtain the complete system of equations required to execute the dynamic simulation we will also need formulae for the linear and angular momentum of the rigid body are also required. The linear momentum of a particle is defined as:

$$\mathbf{p} = m\mathbf{v} \qquad \begin{array}{l} p : \text{linear momentum} \\ m : \text{mass} \\ v : \text{velocity} \end{array} \qquad (2.19)$$

For a rigid body, the linear momentum consists of the sum of the product of mass and velocity for each particle.

$$\mathbf{P}(t) = \sum m_i \dot{\mathbf{r}}_i(t) \qquad (2.20)$$

It has been shown previously that the equation for the velocity of a particle is actually.

$$\dot{\mathbf{r}}(t) = \mathbf{v}(t) + \omega(t) \times \left(\dot{\mathbf{r}}_i(t) - \mathbf{x}(t)\right) \qquad (2.21)$$

Therefore the total linear momentum of the body becomes

$$\begin{aligned} \mathbf{P}(t) &= \sum m_i \dot{\mathbf{r}}_i(t) \qquad (2.22) \\ &= \sum \left(m_i \mathbf{v}(t) + \omega(t) \times \left(\mathbf{r}_i(t) - \mathbf{x}(t)\right)\right) \\ &= \sum \left(m_i \mathbf{v}(t) + \omega(t) \times \sum m_i \left(\mathbf{r}_i(t) - \mathbf{x}(t)\right)\right) \end{aligned}$$

However, most models for rigid bodies will not define the mass of the object as a set of particles each with their own mass. They will prefer instead to use a simple scalar value to represent the mass of the entire object. Our simulation will benefit greatly if we can express the linear momentum of the body as a function of the total mass. Baraff and Witkin [BW97] noted that the sum of the products of mass and position for each particle is equal to zero if we are using the center of mass of the object as the origin for the coordinates of each particle.

$$\sum m_i \mathbf{r}_i(t) = 0 \qquad (2.23)$$

Using this property, the equation for linear momentum becomes.

$$\mathbf{P}(t) = \sum m_i \mathbf{v}(t) = \left(\sum m_i\right)\mathbf{v}(t) = \mathbf{M}\mathbf{v}(t) \qquad (2.24)$$

As a side note, since M is a constant we can also define the first derivative of the velocity as

$$\dot{\mathbf{v}}(t) = \frac{\dot{\mathbf{P}}(t)}{\mathbf{M}} \tag{2.25}$$

By comparing to $f = ma$, we can derive an equation which equates the derivative of linear momentum to the total force on a rigid body.

$$\dot{\mathbf{P}}(t) = \mathbf{F} \tag{2.26}$$

2.2.5 Angular Momentum

Angular momentum is also very useful for dynamic simulation, mostly because it is *conserved in nature*. In contrast, angular velocity can change radically over short periods of time; it is therefore not *conserved in nature*. Linear momentum was described as $P(t) = Mv(t)$, similarly the total angular momentum is described by equation (2.27).

$$\mathbf{L}(t) = \mathbf{I}(t)\omega(t) \tag{2.27}$$

L(t) : angular momentum
I(t) : inertia tensor
ω(t) : angular velocity

Similar to the relation between the derivative of linear momentum and total force acting on a rigid body, the relationship between $L(t)$ and the total torque $(\tau(t))$ applied to a rigid body can be expressed as:

$$\dot{\mathbf{L}}(t) = \tau(t) \tag{2.28}$$

2.2.6 Equations of Motion

We now have all the parts required to define the state vector $Y(t)$ of a rigid body. While the state vector for a particle contained only position and velocity, the state vector for a rigid body will be:

$$\mathbf{Y}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{R}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{bmatrix} \tag{2.29}$$

Y(t) : rigid body state vector
x(t) : position
R(t) : orientation
P(t) : linear momentum
L(t) : angular momentum

Note that the velocity used in the state vector for the particle has been replaced with formulations for momentum. Using momentum in the state vector leads to simpler equations than if velocity were used [BW97]. This is works well since the equations which relate velocity and momentum are quite simple.

The derivative of the state vector for rigid bodies then becomes:

$$\frac{d}{dt}\mathbf{Y}(t) = \frac{d}{dt}\begin{bmatrix}\mathbf{x}(t)\\ \mathbf{R}(t)\\ \mathbf{P}(t)\\ \mathbf{L}(t)\end{bmatrix} = \begin{bmatrix}\mathbf{v}(t)\\ \omega(t) * \mathbf{R}(t)\\ \mathbf{F}(t)\\ \tau(t)\end{bmatrix} \tag{2.30}$$

Now that we have defined our formula for the first derivative of the state vector, all that is required to begin the simulation are the initial values for the state vector. The simulation will pass the derivative of the state vector ($d/dt\, Y(t)$) to an ordinary differential equation solver to evaluate the rate of change for the bodies or particles at any time. This rate of change will then be used to extrapolate the state vector for the body or particles at any point in time. It is to be noted that the simulation may chose to evaluate the differential equation as often as is necessary and at whatever point in time it desires to compute a more accurate result. (We will see later that some simulations require more frequent updates than others.) This brings up the interesting fact that since the force vector ($F(t)$) is a part of the differential equation, and that this one may be evaluated at any time, we must have the ability to also determine the value of the total force vector at any and all times during the dynamic simulation.

2.3 Bounding Volumes

A bounding volume of a model is a simple shape which encapsulates or covers the entire model; it is how the object is viewed when collision detection is performed between objects. Bounding volumes are also knows as *collision geometry* and are used in interactive virtual environments since collision tests between all of the polygons which make up the visual representation of the object would be computationally expensive and would only offer slightly better results than with a lower-resolution geometry. A bounding volume should fit the visual model as closely as possible so the majority of collisions with the bounding volume would also be collisions with the model itself.

It should be pointed out that the goals of having the collision geometry fit the model as closely as possible and of having as few polygons as possible are mutually exclusive. On one hand, having a bounding volume that fits the model as closely as possible is advantageous. For a shape defined by polygons, this would mean using as *many* polygons as possible. On the other hand, we want collision tests between bounding volumes to be as inexpensive as possible. In a volume using polygons, all of the faces of one volume must be tested against all of the polygons of the other model to determine if a collision exists. Therefore, we want the bounding volume to have as *few* polygons as possible.

The most popular bounding volumes used in virtual reality will be presented and the advantages and disadvantages of each are discussed. A larger set of bounding volume types is presented in [Van04].

2.3.1 Bounding Spheres

Spheres are the simplest commonly used bounding volume type. The sphere can be represented using only 4 scalar values: 3 for representing the coordinates of the sphere's center and another for the radius. The center point of the sphere should be the origin of the object's model (the average of all of the model's vertices) and the radius of the sphere will need to be equal to the distance between the model's origin and the vertex which is the furthest away.

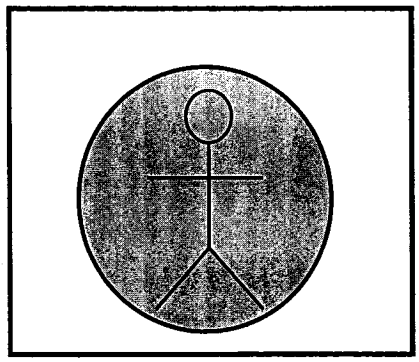


Figure 2.6. A 3D object and its bounding sphere.

To detect a collision between two bounding spheres is extremely easy: if the distance between two object's origins is less than the sum of the radii of both spheres, a collision is occurring. The disadvantage of using bounding spheres is that the bounding volume does not fit the object very tightly since real-world objects rarely have spherical shapes.

2.3.2 Bounding Cylinders

Bounding cylinders are very similar to bounding spheres. Much like bounding spheres, the bounding cylinder must always retain the same orientation in the world, the height of the cylinder remaining along the *up* direction in the environment. They are represented using only 5 scalars (instead of 4 for bounding spheres): 3 for representing the coordinates of the cylinder's center, one for the radius and another for the height. The cylinder is a popular choice for bounding volume since it is much more accurate than the sphere at approximating the human body's shape. Detecting collisions between two cylinders is slightly more complicated than detecting collisions between two spheres. After the distance is taken between the center of both cylinders, only the value that is extended over the world's ground plane must be taken. If this value is less than the sum of the radii of both objects, then we must proceed to the second phase of testing the heights of both cylinders.

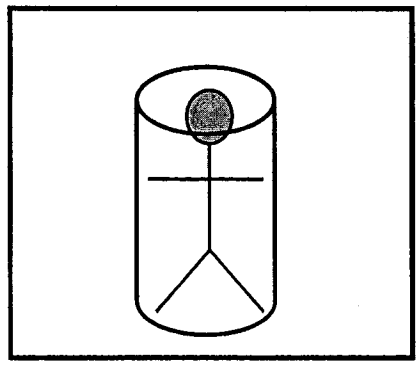


Figure 2.7. A 3D object and its bounding cylinder.

Testing for collisions between bounding cylinders takes slightly more primitive operations than for bounding spheres. The main advantage is that their shape is better adapted for representing the human body.

2.3.3 Axis-Aligned Bounding Boxes (AABB)

The axis-aligned bounding box is a 6-sided volume which contains the model entirely and where all sides are coplanar with the three world planes (*xy*, *yz*, *xz*). AABB's are very simple to compute since every one of the faces is the maximum or minimum value for the coordinate axes when testing all of the object's vertices. They typically fit bodies approximately as well as bounding spheres; only they can be tested faster than bounding spheres for intersections.

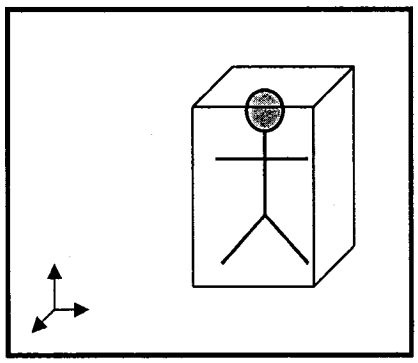


Figure 2.8a. A 3D model and its AABB.

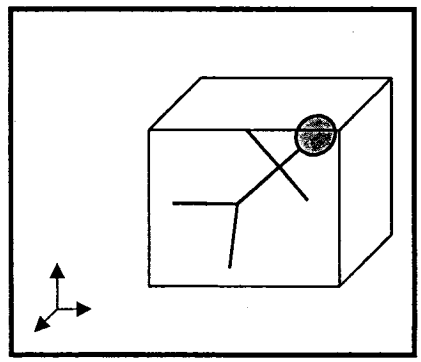


Figure 2.8b. A 3D model and its AABB (notice how the box must expand to account for the new orientation of the object).

2.3.4 Oriented Bounding Boxes (OBB)

Oriented bounding boxes contain 6 sides or 3 pairs of coplanar sides that do not have to be coplanar with the world's *x*, *y and z*-axes. The bounding box is free to change orientation as the model's orientation changes. In fact, the bounding box will maintain the same orientation as that of the model as it moves.

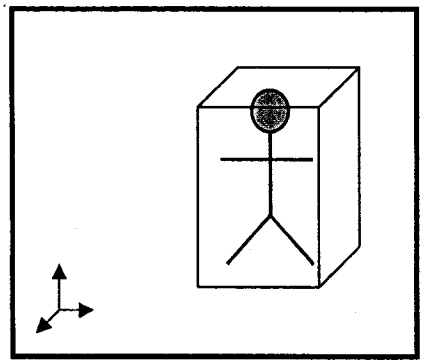


Figure 2.9a. A 3D model and its OBB.

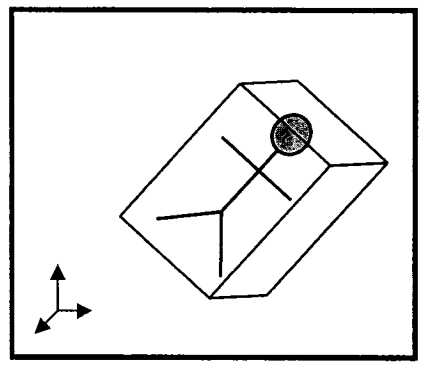


Figure 2.9b. A 3D model and its OBB (noticed how the box maintains the same orientation as the object).

Finding the smallest OBB based on an object's 3D model is difficult, however reasonably tight-fitting boxes can be obtained using heuristics [Van04]. OBB's are not low-storage representations and can use up to 200 primitive operations for intersection testing (compared to 11 for bounding spheres). Their main advantage is that they provide a much more tight-fitting volume type than other bounding volumes.

2.3.5 Polygon Mesh

The last bounding volume to be discussed is the polygon mesh. The polygon mesh consists of the same set of polygons that is used to render the model itself, or a version of the model with a lower level of detail. This type of volume uses no additional storage space since all the necessary information is already contained in the render model itself. It is obviously the most tight-fitting bounding volume possible; there are no spaces in the bounding volume which are not part of the model itself. The main disadvantage is in the intersection testing which takes many more primitive operations than for any other type of bounding volume.

While many of these representations offer very different advantages and disadvantages, it is possible to use them in combination to gain the best of both worlds. For example, many applications choose to do hierarchical intersection testing [Del02] where a cheap intersection test with a very coarse volume is done first and if the test yields a positive result, more expensive tests at finer detail are computed. For example, in [DPW03], the intersection tests begin by verifying the bounding spheres, if the result is positive then a much more expensive OBB test is done. There is no free lunch however; as the disadvantage of combining methods is that multiple bounding volume representations of the object must be stored in memory.

Many other bounding volume representations make use of *space partitioning*, in which the object is subdivided into smaller convex regions called cells; each cell then maintains a list of references to smaller cells which it contains. These representations are categorized based on how the cells are defined and how the relationships between cells are modeled. In *voxel grids* the model is partitioned into uniform rectangular cells, forming a three-dimensional array of cells. The technique is best for rectangular objects of uniform density and size. This

representation does not accommodate models with local densities however. *Octrees*, *k-d trees* and *binary space partitioning* (BSP) trees are all recursive space partitioning techniques which are adaptive to local densities in a model. These techniques also make use of cells and hierarchical structures for relations between cells; the major difference is that the size and shape of the cells can vary. These techniques are used in many three-dimensional video games.

2.4 Collision Detection

Two main equations are used to determine post-collision velocities of any number of colliding bodies. These equations are the law of conservation of momentum (equation 2.31) and the law of conservation of energy (equation 2.32). The conservation of momentum law states that the sum of the momenta of the colliding bodies before the collision will be equal to the sum after the collision. The momentum of a body is obtained by multiplying its mass with its velocity.

$$\sum_{i=1}^{m} m_i v_i^- = \sum_{i=1}^{m} m_i v_i^+ \tag{2.31}$$

The elasticity of collision law accounts for the amount of energy lost during the collision. The coefficient of restitution is a value between 0 and 1, which represents how much of the energy is lost during the collision.

$$e = \frac{-\left(v_1^+ - v_2^+\right)}{\left(v_1^- - v_2^-\right)} \tag{2.32}$$

No energy is lost in a perfectly elastic collision therefore the restitution coefficient will be 1; whereas in a perfectly inelastic collision it will be 0 since all energy will be lost.

But before any processing can be done to handle a collision, the collision itself must be detected. This is dealt with in a hierarchical manner: collision detection using spheres and, if the result of the sphere test was positive, collision detection using *oriented bounding boxes* (OBB). Oriented bounding boxes are a set of polygons which encapsulate the 3D model in a much more accurate way than the sphere. The OBB will move and rotate along with the 3D model they represent in the collision world.

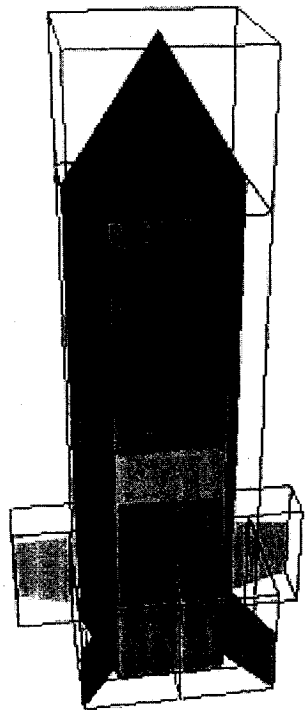


Figure 2.10. 3D model with oriented bounding box (OBB).

The collision response can also be thought of as consisting of two parts. The linear collision response which deals with the linear velocities of objects and the angular collision response which deals with the angular velocities of the objects.

When dealing with linear collision response the collision detection algorithm will only need to supply the normal of the collision plane upon detecting a collision. During a collision, the two colliding objects will not undergo any modification to their momentum which is orthogonal to the collision plane, therefore the speed used in both the elasticity and momentum equations will be the speed of the object along the collision normal, the component of the velocity which is orthogonal to the collision normal will not be affected at all. When dealing with both linear and angular collision response we will need to know not only the normal of the collision plane but also the point of collision.

2.4.1 Spherical Collision Detection

In this first implementation the volume of every object is represented by a sphere with a radius equal to the maximum distance between any of the vertices of the 3D model and the model's center. Collisions occur whenever the distance between the center of two objects is less than or equal to the sum of the radii of the two objects. The collision normal is simply the normalized vector going from the center of one object to the other.

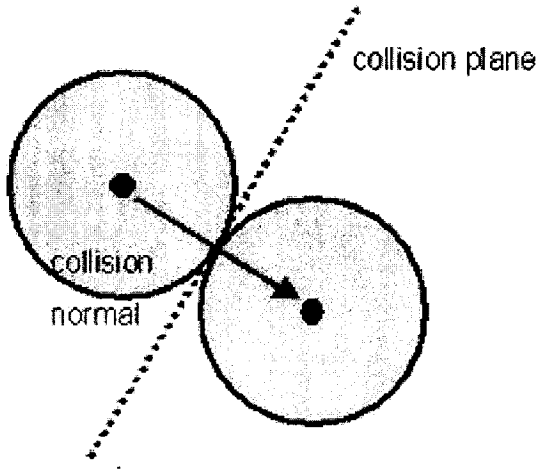


Figure 2.11. A collision between 2 spheres in 3D.

2.4.2 OBB Collision Detection

If a collision has been detected using the method with spheres we can then proceed to our second level of collision detection which is more computationally expensive but also much more accurate. In this method the volume of each object does not consist of a sphere but a set of oriented bounding boxes (OBBs), 3D boxes that rotate when the model rotates. Detecting a collision with these shapes is more involved than it was with spheres. Here is the sequence of events when verifying for collisions of object A against object B:

- For every set of four neighboring vertices of object A, setup a plane equation ($Ax+By+Cz+D=0$) with the normal heading outwards.
- For every edge of object B check if the line intersects with any of A's planes.
- If an intersection point is found, check to see if the intersection point is actually on the polygon and not just on the plane. (A plane ranges from negative infinity to positive infinity whereas a polygon is limited in its bounds.)
- If the intersection point is on the polygon, there is a collision.
- The collision normal is the normal of the traversed plane.

The collision detection routine therefore consists of two hierarchic levels: collision detection with spheres is done first since it is quick and provides a good estimate; if a collision is detected with the spheres then the second level of collision detection is done with the oriented bounding boxes method. The OBB collision detection involves much more logic and calculations than the sphere collision test but is also much more precise than the previous level. Creating a hierarchical collision detection routine provides the best of both worlds, efficient and precise results since the heavier OBB collision detection routine is only called when two objects are near.

2.5 Collision Response

Collision response is responsible for determining the post-collision velocities of objects that have just been involved in a collision. With linear collision response, the angular velocities of the colliding objects is not taken into account or modified. Linear and angular collision response bears its name obviously because it takes into account and modifies not only the linear velocity of objects but also their angular velocity. For example, a baseball bat floating in outer space that is hit at one of its extremities will not only start moving away, it will also gain angular velocity and start spinning on its center of mass.

2.5.1 Linear Collision Response

The *impulse method* is used to determine the post-collision velocities of two objects involved in a collision. Much like a real impulse, the method quantifies an indefinitely large force acting for a very short time (a collision) and produces a finite change of momentum. The impulse method is based on the equations of conservation of momentum and elasticity. It consists of calculating a simple scalar as a function of the velocity and mass of both objects and then using this scalar to modify the velocities of both entities. The impulse scalar is calculated as shown in equation 2.33.

$$J = \frac{(-v_1 \cdot n - v_2 \cdot n)(e+1)}{\left(\frac{1}{m_1} + \frac{1}{m_2}\right)} \tag{2.33}$$

e: restitution coefficient
n : collision normal
m : object mass
v : object velocity

The post-collision velocities of both objects are then modified using this scalar.

$$v_1 = v_1 + \frac{J * n}{m_1} \qquad v_2 = v_2 - \frac{J * n}{m_2} \tag{2.34}$$

Note that for the second object the impulse is subtracted instead of being added to the velocity. This confirms that equal but opposing forces are applied to the objects during the collision.

2.5.2 Linear and Angular Collision Response

Linear and angular collision response will modify both the linear and angular velocities of the objects in the VE. Characteristic equations for this type of collision response are the same as they were for the linear collision (conservation of momentum, energy) except that the angular velocity, the moment of inertia and the distance

from the point of collision to the center of mass will now also affect the momentum of the objects. This algorithm uses the momentum of the object at the specific point of contact, which is dependent on the angular velocity as well as the relative positioning of the point of contact. The moment of a rotating body is defined as shown in equation 2.34.

$$mv_{total} = mv_{linear} + I(\omega \wedge r) \tag{2.34}$$

I: moment of inertia
ω: angular velocity
r: vector from center of mass to collision point

The impulse method is used once again to bring the number of arithmetic operations to a minimum. The equations of conservation of momentum and elasticity, used in conjunction with the complete momentum formula create the complete version of the impulse method as shown in equation 2.35.

$$J = \frac{-(v_1 \cdot n - v_2 \cdot n)(e+1)}{\frac{1}{m_1} + \frac{1}{m_2} + \left(\frac{(r_1 \wedge n) \wedge r_1}{I_1} + \frac{(r_2 \wedge n) \wedge r_2}{I_2}\right) \cdot n} \tag{2.35}$$

The impulse (J) here will be used to modify the linear and angular velocities of the objects in our virtual environment. Once again the impulse will be added to the first object but subtracted from the second object (equations 2.36-2.37) since both bodies receive opposing forces.

$$v_1 = v_1 + \frac{J * n}{m_1} \qquad \omega_2 = \omega_2 + \frac{J * (r_2 \wedge n)}{I_2} \tag{2.36}$$

$$\omega_1 = \omega_1 + \frac{J * (r_1 \wedge n)}{I_1} \qquad \omega_2 = \omega_2 + \frac{J * (r_2 \wedge n)}{I_2} \tag{2.37}$$

Chapter 3

Physically Based Modeling for Articulated Bodies

This section builds on the work introduced in the first section regarding physical simulation of rigid bodies by presenting techniques for articulated bodies. This work investigates the use of physical simulation as a means to achieving believable motion of articulated figures and objects in virtual reality environments. A group of alternative algorithms suitable are presented and qualitatively compared.

3.1 Terminology and Background

Prior to going into analysis of the algorithms, it is useful to present some terminology and specialized techniques that will appear in many instances. These are presented to give the reader a quick background to support the algorithms presented.

3.1.1 Degrees of Freedom

Degrees of freedom are the number of coordinates over which an object has the liberty to move. For example, a particle in a 3d environment has three degrees of freedom since it can move along the x, y and z-axes. Rigid bodies in virtual environments typically will have six DOF's (degrees of freedom): three to represent the position of the object in the world, and another three to represent the object's orientation.

Degrees of freedom are also intrinsically linked with joints; different types of joints will allow different types of motion. In articulated bodies, each part of an object will be joined to the rest of the body via a joint. The type of joint will determine the extent to which the segment may move. Here is a quick rundown of types of joints.

- *Prismatic joint*: a prismatic joint represents a single translational degree of freedom along a specified axis between two bodies.

- *Revolute joint*: a revolute joint constrains the body connected to a single rotational degree of freedom around a specified axis.
- *Spherical joint*: a spherical joint represents three rotational degrees of freedom around a single pivot point.
- *Universal joint*: a universal joint is a spherical joint that can rotate in any way as long as the axis of rotation is orthogonal to a plane formed by both bodies and the joint.

The degrees of freedom of an articulated system are slightly more complex to visualize. Most virtual environments consist of worlds which are three dimensional therefore the bodies that lie within often have 6 degrees of freedom: three components for the location of the object, and another three for the orientation of the object in the world. In articulated bodies, if one of the parts was to be disconnected from the group, it would also have six degrees of freedom but because it is attached to other parts of the body, its constraints, will remove many of its degrees of freedom.

3.1.2 Generalized and Augmented Coordinates

When faced with the choice of how to simulate the constraints which hold various members of an articulated body together, ultimately only two options are offered to us. On one hand, we can model the constraints by reducing the number of coordinates required to uniquely describe the system configuration. On the other hand, we can maintain all of the coordinates of each member in the system, and resort to additional complementary forces to maintain the constraints. The former is known in the literature as *reduced* or *generalized* coordinates and the latter is known as *augmented* coordinates. This section will describe both methods in depth while outlining the positive and negative aspects of each. This will serve as a basis for the practical algorithms presented later on which will make use of these techniques.

In a reduced-coordinate modeling of a constrained system with m degrees of freedom, a set of constraints will have the ability to remove c of these degrees of freedom (3.1). The remaining ($n = m - c$) degrees of freedom will be used to describe the configuration of the system.

$$n = m - c \tag{3.1}$$

m: maximal coordinates of the system.
n: reduced coordinates.
c: scalar coordinates which describe the constraint forces.

For example, in the system shown in figure 3.1, consisting of 2 revolute joints, there are a total of 6 coordinates. Each segment contains positional coordinates along the x and y axes of the graph and one rotational coordinate to describe the orientation of the segment. These 9 original degrees of freedom are called the set of *maximal* coordinates.

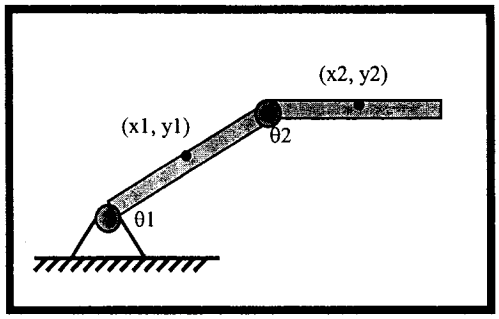


Figure 3.1: System represented by *maximal* coordinates

In a generalized coordinate formulation of this system, only 4 coordinates would be required to describe the system-state. The positional parameters for the contact with the ground and the angles of both joints (note that the length and size of both members is not considered since they cannot be modified by the simulation).

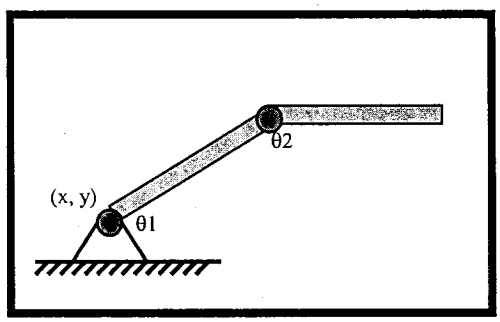


Figure 3.2: System represented by *reduced/generalized* coordinates

While the generalized coordinate formulation of the system would only require 4 coordinates to describe the system, the augmented-coordinate model requires 6 (as in figure 3.1).

The first problem often encountered in systems that use reduced coordinates is that to parameterize the m maximal coordinates in terms of n generalized coordinates can be an intractable problem in many cases. In other cases it can be impossible to obtain such parameterization. However, the types of articulated bodies which interest us most in virtual environments are loop-free articulated bodies and these are generally very easy to parameterize.

In an augmented-coordinate system using a straightforward set of m maximal coordinates the constraints are enforced by introduction of complementary forces. The basis for each of these constraint forces is known beforehand and is used to compute the Lagrange multipliers of the system. The use of Lagrange multipliers in dynamic simulation is very interesting because it allows an arbitrary number of constraints to be used simultaneously. They also allow the use of constraints which are not solely based on maintaining relative positions between segments, such as non-holonomic constraints (to be covered shortly). For example, velocity-

based constraints can be represented using augmented coordinates in conjunction with Lagrange multipliers but inherently cannot be modeled using reduced-coordinate systems.

Considering performance, in a generalized coordinate formulation, $O(n^3)$ time is required to compute the acceleration of the n coordinates. Whereas for loop-free articulated bodies the accelerations of the n generalized coordinates can be found in O(n) time [PAK00]. More recently, an O(n) solution for computing Lagrange multipliers on sparse acyclic constraint systems has also been discovered [Bar96] by using simple techniques for efficiently solving sparse-matrices.

3.1.3 Holonomic Constraints

A system is said to be holonomic if the number of degrees of freedom equals the number of generalized coordinates in a set of complete and independent generalized coordinates. In other words, a particularity of holonomic systems is that when all of the joints return to their original configuration, it also guarantees that the root of the system returns to the original system position. More generally, the system outcome for a non-holonomic system is path-dependent. This same guarantee cannot be applied to non-holonomic constraints. For example, when riding a two-wheeled cart, a return to the original internal (wheel) configuration does not guarantee a return to the original system (cart) position.

Another way to differentiate between holonomic and non-holonomic constraints is by classifying the types of constraints which are in the system. If the constraints are dependent only on time and the position of various entities in the system, the constraints are said to be *geometric*. Systems which contain only geometric constraints are said to be holonomic. If a constraint is dependent on time, position of coordinates in the system and velocities of components in the system, the constraint is said to be *kinematic*. Systems which contain at least one kinematic constraint are said to be non-holonomic.

Holonomic constraints can also be sub-divided into two categories. The system is said to be *scleronomic* if time does not appear in the set of constraint equations of the system, otherwise the holonomic system is said to be *rheonomic*.

3.1.4 Lagrange Multipliers

Lagrange multipliers are useful for solving problems of constrained optimization. The method of Lagrange multipliers is a powerful tool when one wishes to minimize or maximize a function that is subject to certain conditions or constraints. The method consists of applying the following:

$f(x, y)$: Function to minimize/maximize
$g(x, y) = 0$: Constraint equation
λ : Lagrange multiplier

$$\nabla f(x,y) = \lambda \nabla g(x,y) \tag{3.2}$$

$$\frac{\delta f(x,y)}{\delta x}\vec{i} + \frac{\delta f(x,y)}{\delta y}\vec{j} = \lambda \frac{\delta g(x,y)}{\delta x}\vec{i} + \lambda \frac{\delta g(x,y)}{\delta y}\vec{j}$$

The goal is to optimize a function of several variables that is subject to one or more constraints by setting further functions of the variables to given values. The Lagrange multiplier is introduced to reduce the constrained problem to an unconstrained problem which may then be solved using the gradient method.

3.1.5 Stiff Differential Equations

The problem of stiff differential equations appears when we are attempting to solve an ODE (ordinary differential equation) that has a solution that is varying slowly at a high level, but there are nearby solutions that vary rapidly. Figure 3.3 displays a solution to a stiff differential equation. Notice how nearby solutions vary wildly in the following graph of the solution to the equation. The top graph represents the solution to the different equation; the second graph is the same solution to the equation only it has been zoomed in considerable.

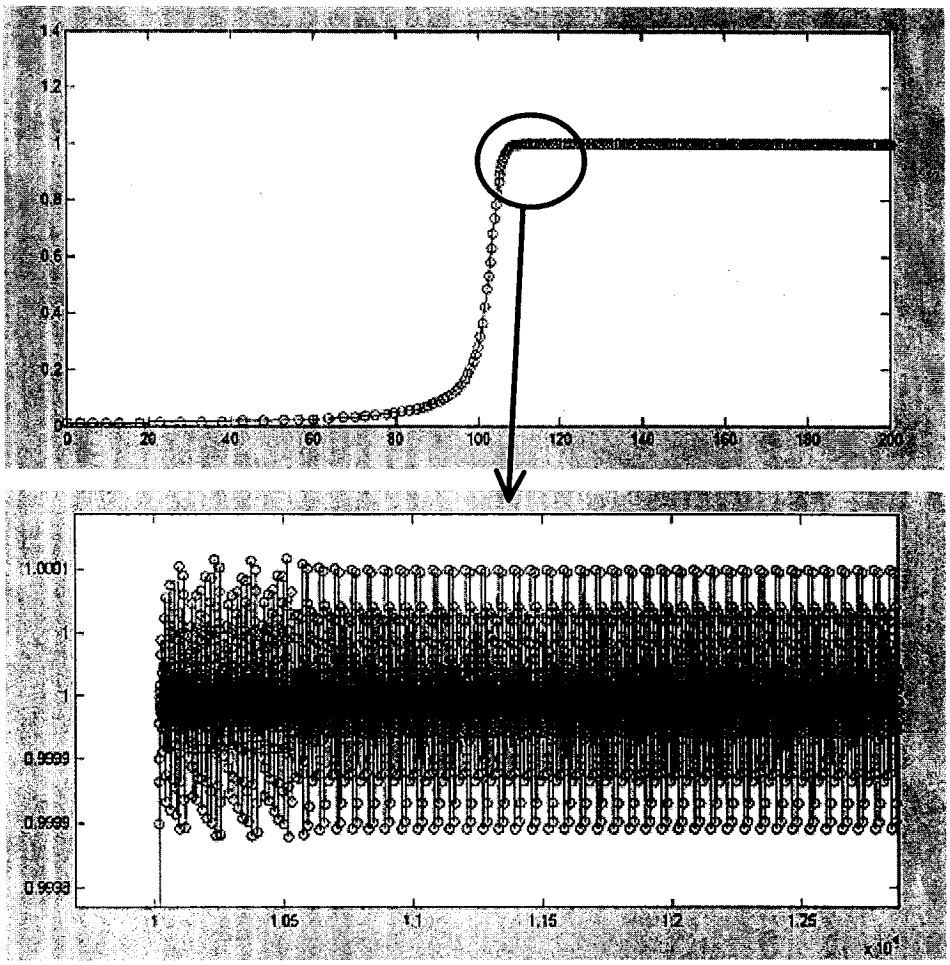


Figure 3.3: Solution to a stiff ODE obtained by *explicit* integration (above) and zoomed-in to a specific segment (below).

The actual solution to this ODE does not vary such as it is shown in figure 3.3; the variations are due to the approximations done in the numerical integrator used to solve the equation. To solve this system with satisfactory results the solver must take very small steps; methods for non-stiff problem can solve stiff differential equations, they simply take much longer to do it. The problem is therefore an efficiency issue, an especially important issue in real-time interactive virtual environments. The alternative is to use *implicit* integration methods (in contrast to the *explicit* method used above). Implicit methods work by using matrix operations to solve a system of simultaneous linear equations that predict the evolution of the solution. These methods do more work at every time step but are capable of taking much larger steps without introducing instability into the solution. Figure 3.4 displays the solution calculated by an implicit method to the same problem as previously. Note how the time-steps are much larger between each calculated point.

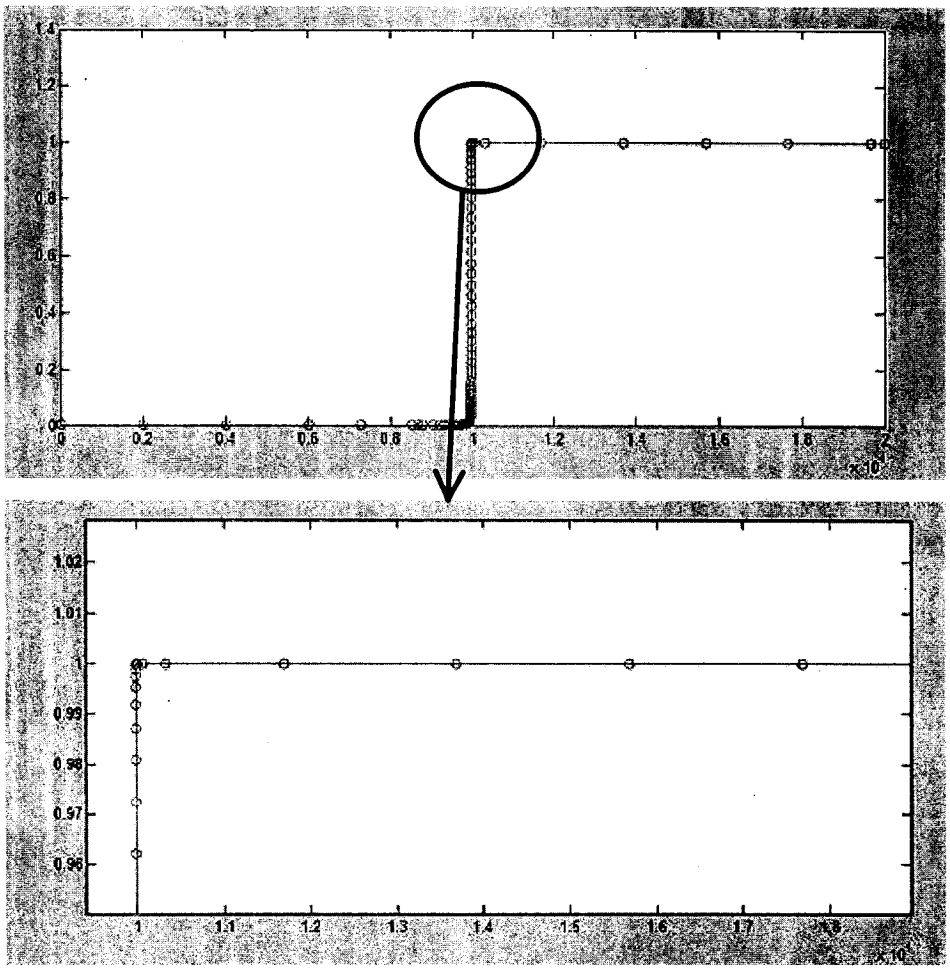


Figure 3.4: Solution to a stiff ODE obtained by *implicit* integration (above) and zoomed-in to a specific segment (below).

Baraff [Bar96] has shown that implicit methods outperform explicit methods thanks to this ability to take larger steps when dealing with stiff equations.

3.1.6 Implicit/Explicit Integration Methods

Explicit integrators solve problems of large systems of equations (most often in the form of matrices) by considering each equation in the system independently. They are written in such a way that all unknown values in a coupled system can be updated independently. For example, if one were simulating the motion of a large piece of cloth modeled as a large network of particles where each particle is connected to four neighbors via a rigid stick the explicit integrator would solve for the position of each particle individually. As one particle is updated, this newly computed position would affect other particles which may have been solved earlier by the algorithm, and the new position of the particle may have been computed using the state of other particles in the system that have not been updated yet. This problem of particles becoming out-of-sync with others in the same

system is often tolerated. Explicit methods of integration create results which are not extremely precise but they are often used regardless due to the fact that the implementation of an explicit integrator is much simpler than that of an implicit integrator. Problems with explicit integrators may also arise however due to numerical rounding off errors as the inter-related values in an articulated body become out-of-sync with each other. The error, which accumulates over each time step, causes the system to become unstable and erratic.

In a system that uses implicit integrators, the algorithm will use matrix operations to solve a system of simultaneous linear equations at every time step. This yields results that always remain in sync, at the cost of having to do more work per time step. In comparing implicit and explicit integration methods, it is true that explicit methods require fewer computations to be done at a single time step. Baraff [Bar96] has shown however that implicit methods are faster since they can handle much larger time steps than explicit methods, which tend to become unstable if they are not updated often enough. The ability to support larger steps in implicit methods overshadows the larger amount of computations which must be made at every update.

3.1.7 The Constraint Jacobian Matrix

When utilizing augmented coordinates as the formulation for the system state, a separate set of equations is required to represent the constraints contained in the system. Each joint is defined as a constraint equation which is a function of coordinates and variables. These coordinates and design variables are concatenated in a vector $\mathbf{q}$ which is known as the *spatial variables vector*. Each $\mathbf{q}$ vector will have 7x1 dimensions: a 3x1 Cartesian coordinate vector for the object's position and a 4x1 quaternion orientation vector. The system of joints is then defined as a system of constraint equations called $\mathbf{C}$.

Each joint in the articulated body will be represented by its own constraint equation $\mathbf{C}^i$. In figure 3.1, our system with two segments and 2 revolute joints (one which is attached to the ground), we find 3 different constraints and therefore also three different constraint equations (3.3).

$$\mathbf{C}^0(\mathbf{q}_g) = 0 \qquad (3.3)$$

$$\mathbf{C}^1(\mathbf{q}_g, \mathbf{q}_{l1}) = 0$$

$$\mathbf{C}^2(\mathbf{q}_{l1}, \mathbf{q}_{l2}) = 0$$

The first constraint is a function of the spatial variables of the ground and describes that the ground must remain immobile. The second constraint is a function of the $\mathbf{q}$ vectors for the ground and the first link. The remaining constraint is based off of spatial variables of the first and second links and represents the attachment between the two links.

In the case of our examples, the constraints signify that the members of the articulated body must remain together. Mathematically this means that the second derivative of $\mathbf{C}$ must be equal to 0 ($\ddot{\mathbf{C}} = 0$) if we assume that the initial conditions are $\mathbf{C} = \dot{\mathbf{C}} = 0$. To achieve this result, we must solve the system for a constraint force $\hat{\mathbf{Q}}$ which, when added to the externally applied force $\mathbf{Q}$, guarantees $\ddot{\mathbf{C}} = 0$. To solve the system, derivatives of $\mathbf{C}$ must be computed such as in equation (3.4).

$$\dot{\mathbf{C}} = \frac{\partial \mathbf{C}}{\partial \mathbf{q}} \dot{\mathbf{q}} \tag{3.4}$$

The matrix $\partial \mathbf{C} / \partial \mathbf{q}$ is called the *Jacobian* of $\mathbf{C}$. The constraint Jacobian matrix will often have a sparse structure. The scarcity is a direct result that each constraint/particle pair will correspond to only one block in the matrix. This block will only be non-zero if the constraint is dependent on the particles. An example is presented in figure 3.5.

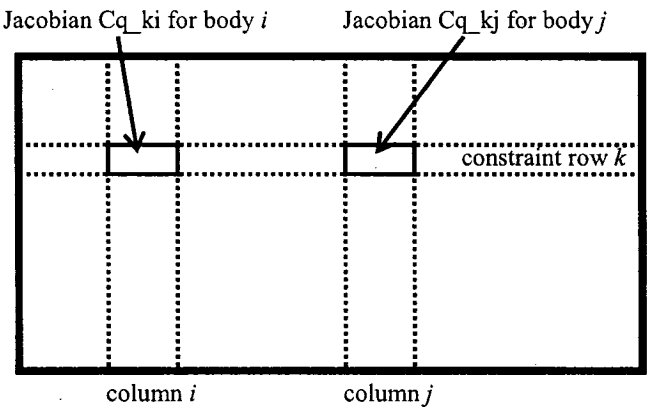


Figure 3.5. Constraint Jacobian matrix.

The rows of the Jacobian correspond to the constraint equations. The columns included in the system correspond to $\mathbf{C}_q$, the partial derivative of $\mathbf{C}$ with respect to each design parameter $\mathbf{q}$.

3.2 Verlet Integration

The first method presented for physically based modeling of articulated bodies is a combination of *Verlet* integration with *Gauss-Seidel* iteration, a method for solving systems of equations which iteratively converges towards the solution. The algorithm presented is fast, iterative and quite stable. It is also used frequently when simulating molecular dynamics [GHWS91]. Although the concepts and techniques used have been known for quite some time, this technique for physical simulation in virtual environments has only only gained wide

popularity since Thomas Jakobsen presented it at the 2001 *Game Developers Conference* [Jak03]. This technique has since been applied in many programs requiring physical simulation at interactive rates.

In the following section, an analysis of the Verlet integration algorithm is presented as an alternative to those adopted in virtualized environments in the past. It is relatively simple to implement, is numerically stable, is capable of time/accuracy trade-offs and is efficient enough for execution at interactive rates on even low-performance machines.

3.2.1 Verlet Integration

At the heart of this technique for simulation lies a particle system from which both rigid and articulated bodies can be derived. In typical systems, both the position and velocity of a particle are used and stored in memory to represent the current state of the particle. A particularity of the verlet system is that it makes use of a *velocity-less* representation of a particle. The current velocity of the simulated particles is not referenced or stored anywhere in the implementation, however we will see later on that both the current and previous positions of the particle are stored and used.

The verlet integration algorithm makes use of Newton's second law; the force on an object is equal to its mass multiplied by its acceleration. In each time-step of the simulation, the forces are gathered to compute the acceleration of the object. Typical simulations will then use this acceleration to compute a new velocity for the body. This velocity is then used to compute a new position for the object. This is simple forward Euler integration (3.5).

$$\begin{aligned} \mathbf{v}_{new} &= \mathbf{v} + \mathbf{a}\Delta t \\ \mathbf{x}_{new} &= \mathbf{x} + \mathbf{v}\Delta t \end{aligned} \tag{3.5}$$

In a verlet system we will instead store the current and previous positions of the particle to be simulated. First of all, we store the current position of the particle in a temporary variable since it will be needed later on. We then use the simple formula for computing the position of the object used in forward Euler integration. The velocity is then replaced by the other formula for computing the updated velocity. Verlet integration will then approximate the velocity by the difference between the current position and the previous location.

$$\begin{aligned} temp &= \mathbf{x}_t \\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \mathbf{v}\Delta t \\ &= \mathbf{x}_t + \mathbf{v}_t t + \mathbf{a}\Delta t^2 \\ &\cong \mathbf{x}_t + (\mathbf{x}_t - \mathbf{x}_{t-1}) + \mathbf{a}\Delta t^2 \\ &= 2\mathbf{x}_t - \mathbf{x}_{t-1} + \mathbf{a}\Delta t^2 \\ \mathbf{x}_t &= temp \end{aligned} \tag{3.6}$$

Finally, the previous position variable (*temp*) is set to the value of the current position from the beginning of this update in the simulation. Note that between lines 3 and 4 of equation 3.6, the term $(\mathbf{v}_t t)$ is simply approximated by the difference between the previous and current positions $(\mathbf{x}_t - \mathbf{x}_{t-1})$, this approximation reduces the accuracy of the simulation but by doing so it simplifies the resolution of the equation greatly since the velocity of the particles must no longer be computed or tracked.

The verlet system is not meant to be extremely accurate; the approximation of the velocity by the difference of the previous and current positions introduces some error into the simulation. The advantage of the verlet system is that it is quite stable, meaning that simulations using verlet will rarely ever become out of sync or explode to infinite values (an effect which can be noticed when using systems of stiff springs). The stability comes from the fact that the velocity is implicitly given, it is consequently much harder for the velocity and position to become out of sync. As an added bonus, drag can be introduced very easily into the system by modifying the factor of 2 in the equation to compute the new velocity. Values lower than 2 will have an effect of lowering the velocity (the approximation of the velocity) at every step of the simulation. This in turn has the effect of creating an illusion of drag on the particle or body.

3.2.2 Gauss-Seidel Iteration for Solving Constraints

The verlet integration method allows us to simulate the motion of rigid bodies and particles but to model and simulate articulated bodies we must also solve their constraints. Many different techniques will be analyzed to solve this problem; starting with a recursive method called *Gauss-Seidel iteration*, or *relaxation*.

For the simulation of a simple stick or rod in a virtual environment many implementations model the stick as a spring. However, if the elasticity of the spring is set very low (to properly model a rigid stick) simple integration techniques will become unstable. On the other hand, if the spring is made weak the stick will appear to become shorter or longer as it becomes constrained, which leads to unnatural behavior. The *Gauss-Seidel* technique, also called *relaxation*, uses a concept similar to a spring whose stiffness goes to infinity. When this is done, the system suddenly becomes solvable in a stable way with a very simple and fast approach.

The approach taken in relaxation is one of pure repetition. In a system with multiple constraints like an articulated body, the algorithm works by simply solving a first constraint independently of all others. If the particle which was just evaluated underwent a transformation, then the other particles which are connected to it are also displaced to satisfy the constraint which specifies the distance between both particles. This process consists of a single iteration in Gauss-Seidel. After one integration step, the constraints affecting some of the particles might have become invalid due to the transformations to solve other constraints. In order to obtain the correct distance once again, the particle's constraints are solved once again by pushing the particles directly away from each other or by pulling them closer together. The constraint solver can loop as many times as the simulation requires as it converges towards the actual solution.

A major advantage of the verlet system is that it supports trade-offs between time of computations and quality of results. The virtual reality simulation may choose to do only a single loop to satisfy the constraints, yielding results which are not very precise; or the system could execute many loops to solve the constraints, yielding very accurate results. The time/accuracy trade-off comes from this ability to stop the simulation at any time. Stopping early will not ruin anything but the results might appear somewhat sloppier.

3.2.3 Verlet Integration Example

This example will consist of a simple stick in a box and the simulation will be run to satisfy the constraint which specifies the length of the stick as well as the constraint that the stick remain within the box.

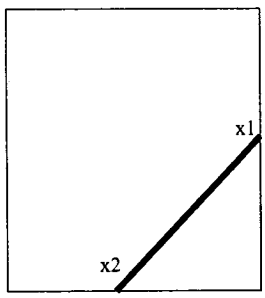


Figure 3.6. Example of a stick in a box.

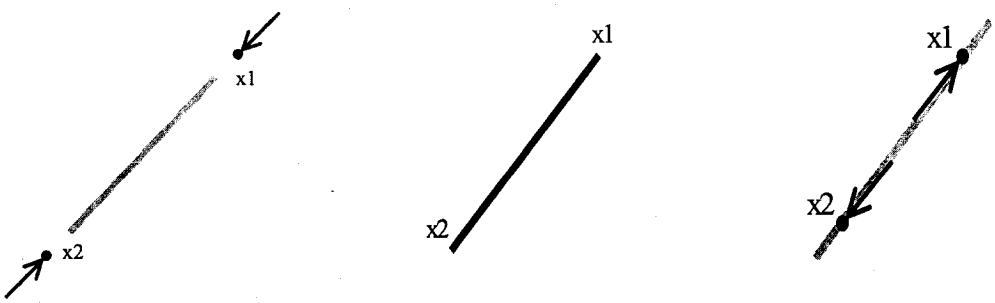
The first constraint of maintaining the points within the box is expressed as

$$x_i \geq 0 \qquad \text{and} \qquad x_i \leq 500 \qquad \text{for } i = 1, 2$$

The second constraint is that the two particles (end points) of the stick must remain exactly the same distance from each other (zero elasticity). This constraint is expressed as follows.

$$|\mathbf{x}_2 - \mathbf{x}_1| = 100 \tag{3.7}$$

As the two particles of the stick are brought closer together or further apart we will accommodate the distance constraint by dividing the difference amongst both particles. If the distance is too large both particles will be brought closer together, if the distance is too short both particles will be brought apart.



Distance too large Correct distance Distance too short

Figure 3.7. Constraint on the stick to maintain its length.

The constraints for maintaining the particles inside the box are simply enforced as follows:

$$\mathbf{x} = v\min(\mathbf{x}, \{0,0,0\}) \qquad (3.8)$$
$$\mathbf{x} = v\max(\mathbf{x}, \{500,500,500\})$$

The simulation can then simply loop over both constraint solvers until it obtains a result which is acceptable.

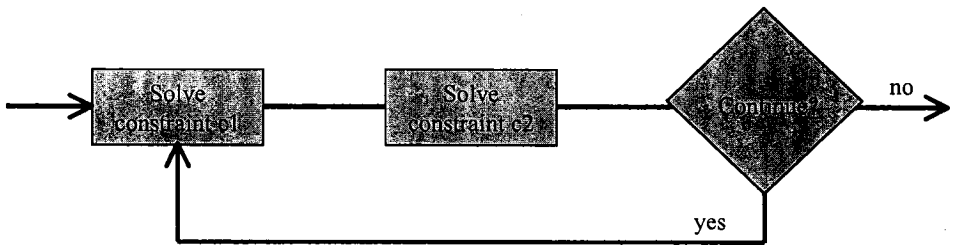


Figure 3.8. Loop for solving the constraints on the stick.

Many different factors can be used to determine if the simulation should run another loop. The decision can be based on time if the simulation needs to provide results within a specified amount of time, much like in a hard real-time system. Some implementations will cache the result from the previous iteration and will only end the iterations if the current result of the relaxation has only changed very slightly in relation to the previous result. The solution can also be made to converge towards the solution even faster through the use of *over-relaxation* techniques such as in [TAP97].

3.2.4 Gauss-Seidel Applied to Matrix Operations

Many virtual reality simulations represent the constraints on rigid and articulated bodies in matrix form. The Gauss-Seidel method for solving the n equations of a linear system of equations begins by splitting the (n x n) square matrix into three portions, the main diagonal D, the lower half L (all elements in the matrix above the diagonal are equal to 0) and the upper half U (all elements below the diagonal are equal to 0). The result is as follows (3.9):

$$\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U} \qquad (3.9)$$

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \cdots & \cdots & \cdots \\ a_{1,n} & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} a_{1,1} & \cdots & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & a_{n,n} \end{bmatrix} - \begin{bmatrix} 0 & \cdots & 0 \\ \cdots & \cdots & \cdots \\ -a_{n,1} & \cdots & 0 \end{bmatrix} - \begin{bmatrix} 0 & \cdots & -a_{1,n} \\ \cdots & \cdots & \cdots \\ 0 & \cdots & 0 \end{bmatrix}$$

The Gauss-Seidel iteration theorem then states that the solution can be obtained using the following iteration scheme:

$$\mathbf{Ax} = \mathbf{B} \qquad (3.10)$$

$$\mathbf{AP} = \mathbf{B}$$

$$\mathbf{P}_{k+1} = (\mathbf{D} - \mathbf{L})^{-1} \cdot \mathbf{U} \cdot \mathbf{P}_k + (\mathbf{D} - \mathbf{L})^{-1} \cdot \mathbf{B}$$

With a carefully chosen $\mathbf{P}_0$, the system will eventually converge to the solution $\mathbf{P}$.

3.2.5 Articulated Bodies

Combining the Verlet integration technique along with Gauss-Seidel iteration, we have the building blocks for physically simulating articulated bodies. As was shown earlier, at the heart of this simulation is a particle system. While having capabilities for simulating simple rigid bodies, the rigid body is specialized for particle systems. We will therefore be constrained to modeling all of our articulated bodies as systems of particles with rigid ("stick") constraints between them.

Spherical joint

Spherical joints with 3 degrees of freedom are modeled by simply letting two rigid bodies share a single particle.

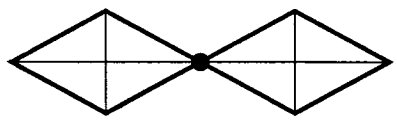


Figure 3.9: A spherical joint modeled with particles.

Revolute joint

Revolute joints with one degree of freedom are modeled by letting two rigid bodies share two particles. The rotation freedom will then be around the axis that intersects both particles.

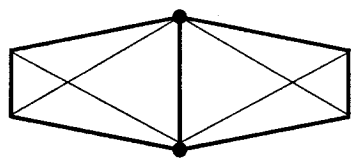


Figure 3.10: A revolute joint modeled with particles.

Prismatic joint

Prismatic joints (with one translational degree of freedom) are less intuitive to visualize. To model this type of joint it is necessary to introduce equality constraints. These equality constraints are essentially virtual "stick" constraints with specific attributes. For a prismatic joint, two equality constrains must be added, namely that the particle on the side of the joint with the end-effector must remain collinear with the previous particle and the joint itself.

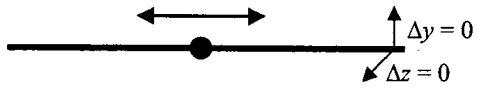


Figure 3.11: A prismatic joint modeled with particles.

The simulation enforces that there is no movement between two particles along two of the three axes. In the figure above, it is the y and z axes.

3.2.6 Analysis

Combining a stable integrator along with an iterative approach to solving constrains works well but is not without its caveats. In this section we will discuss the advantages and caveats of the consequences of using this method.

Accuracy

Verlet integration is a wonderfully stable integration technique for obtaining the updated positional parameters of objects in virtual environments. It executes very quickly since it contains very few primitive operations. Where the Verlet integration technique fails to shine is in precision. In the equation to obtain the updated position of the particle it is noted that $(x - x^*)$ is used to approximate the current velocity of the particle. The following graphs compare the accuracy of the technique to a *Runge-Kutta* technique of order 2.

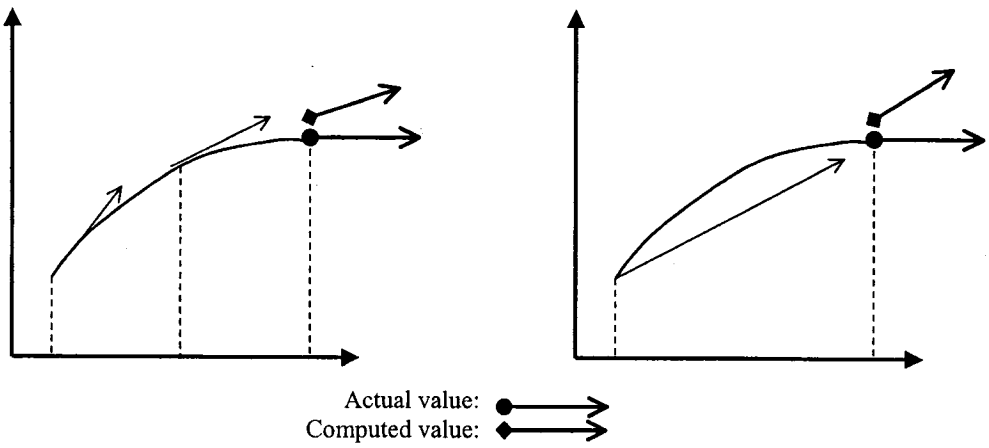


Figure 3.12a: Runge-Kutta order 2 integration.

Figure 3.12b: Verlet integration.

Close examination of these graphs reveals that the estimated velocity of the particle is more precise when it is generated by RK2 integration than when it is done via the Verlet scheme. Therefore we may conclude that Verlet integration is not extremely precise.

Time/accuracy trade-off

It is difficult to discuss the accuracy of the Gauss-Seidel iteration technique for solving constraints without also mentioning performance. Since it is an iterative approach the precision of the result will be highly dependent on the number of iterations computed. This gives the algorithm a very useful time/accuracy trade-off. If a certain level of inaccuracy is allowed or if the physical simulation has a fixed number of cycles over which it may execute, the simulation can be allowed to run at higher rates by reducing the number of iterations done on each object.

The caveat in using this technique however is it can take many iterations to output a very precise answer. Also, it is extremely difficult to obtain an exact solution using this technique since it will converge towards the actual solution but rarely outputs an exactly correct answer. Therefore, if a very accurate answer is required from the simulation, it is best to use a non-iterative technique for solving the constraints. If the speed of execution of the simulation is more important than the exactitude of the results, the Gauss-Seidel iteration technique is a valid option for solving systems of constraints.

3.3 Stiff Springs

Another popular method for modeling constraints is the method of *stiff springs*. Techniques for accurately simulating springs have been known for many years. The idea behind stiff springs is to apply an extremely strong string between two connected bodies to simulate a joint or constraint which exists between the two bodies. We will offer a qualitative assessment of this technique combined with both implicit and explicit integrators.

The origin of stiff springs lies in simulations for springs or rope that is modeled as many point masses connected with springs such as in figure 3.13.

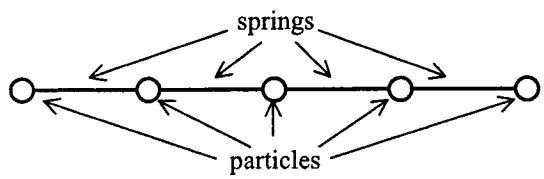


Figure 3.13: A rope modeled using particles and springs.

In essence, this model of a string consists of many particles, each connected to the next by a spring. Each spring is defined by two constraints:

- k: A constant used to represent the stiffness of the spring.

- d: A positive distance value at which a spring remains steady.

The force generated by the spring on each particle can be defined as:

$$\mathbf{f} = -k \cdot (\mathbf{x} - d) \qquad x\text{: distance from the mass to the point it is bound to.} \qquad (3.11)$$

Like any other physical system, the spring must also deal with the loss of energy due to friction but for simplicity we will put this term aside. This technique is well known [Bou02] and has been in use for many years.

3.3.1 Stiff Differential Equations

The basic goal of stiff springs is not to model a spring but to model a spherical joint. However, the physics and mathematical formulae required to properly simulate a spherical joint are quite complex in comparison to those of springs. Even though the mathematics are much more complex, the results between both techniques are fairly similar; the difference being that the connecting rod between the joints will vary in length if springs are used, whereas it will retain its exact length in other models. For this reason, computer scientists have typically used springs to model spherical joints. Eliminating difficulties arising from the rod's length being modified, the stiffness coefficient on the spring is made to be extremely large.

The increase in stiffness of the springs creates a system of differential equations (which must be solved to advance the simulation) that is said to be *stiff*. Sets of differential equations can be stiff as soon as we are dealing with more than a single differential equation. The problem occurs where there are two or more very different scales of the independent variable on which the dependent variables are changing. Reference [PTVF02] provides an example encountered in integrating a stiff equation. It is supposed that the two gray lines, one solid and the other dashed, represent two solutions to the equations. The integration, represented as the dotted sequence of segments, exhibits unstable results such as those shown here when simple integration schemes are used.

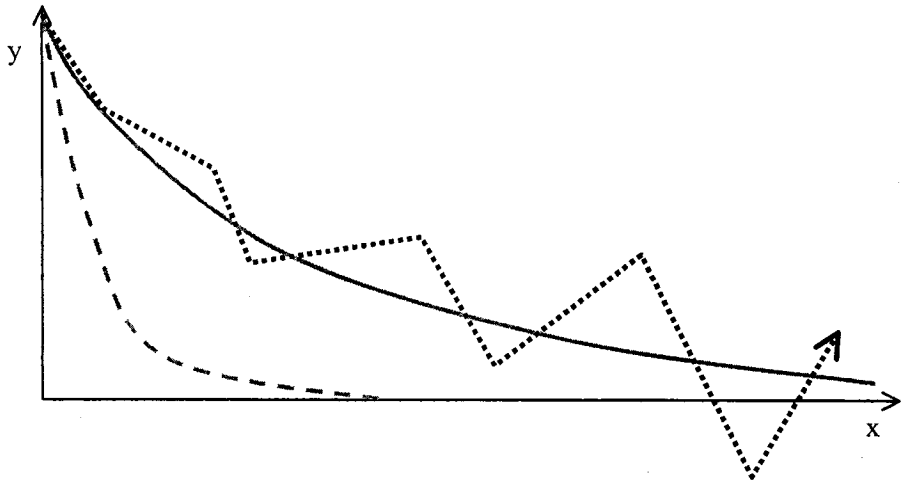


Figure 3.14: Instability example when integrating stiff equations. The solid and dashed gray lines are two solutions to the equation. The unstable integration is shown as the erratic black dotted sequence of segments.

Many virtual reality models represent a particle using both its velocity and position, this means that we are dealing with one differential equation for the velocity and another for the position. These types of models that use 2 or more differential equations that all need to be integrated to make the simulation advance are all possibly susceptible to stability problems when explicit integrators are used. Two popular methods exist to solve this problem. The first solution is to simplify the model to use a single variable instead of many to represent the position and velocity of moving objects. This creates a system with a single differential equation, which will therefore not be prone to stability problems. Verlet integration [Jak03] is an example of this, as Jakobsen eliminates the velocity variables in the system by estimating it using the difference between the current and previous positions of the object. The system then requires just a single differential equation.

The second solution for solving instability problems is to use an *implicit* integration method. A function which evaluates the new version of a value explicitly based on the current value of the variable is said to be *explicit*; the opposite is an *implicit* integration.

3.3.2 Stiff Springs with Explicit Integration

Forward Euler integration is an explicit integration technique that is very popular due to its simplicity. It computes the new value of a variable by simply adding the product of the rate of change (the first derivative) and the step size to the previous value for the variable.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\dot{\mathbf{y}}_n \qquad h\text{: step size} \qquad (3.12)$$

The method is called *explicit* because the new value (y_{n+1}) is given explicitly in terms of the old value (y_n).

Consider a 2D example for a single particle in which the constraint is that the y-coordinate must always remain at zero. This constraint is modeled by adding the following force $f = -ky(t)$ where k is a very large positive constant. By having a very large spring constant the particle will never move too far from the line $y = 0$ since the term $-ky(t)$ will always be pulling it back towards zero. It is assume that there is also a spring attached to the x component which creates a force of $-x(t)$. The state vector and its first derivative are as follows for this equation.

$$\mathbf{Y}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{y}(t) \end{bmatrix} \qquad \dot{\mathbf{Y}}(t) = \frac{d}{dt}\begin{bmatrix} \mathbf{x}(t) \\ \mathbf{y}(t) \end{bmatrix} = \begin{bmatrix} -\mathbf{x}(t) \\ -k\mathbf{y}(t) \end{bmatrix} \tag{3.13}$$

We expect the particle to be pulled towards the origin of the coordinate system with a noticeably stronger attraction to the line $y = 0$. The next step in the simulation consists of solving the equation. We begin by utilizing an explicit integration approach, Euler's method. Using Euler, if we take a step of size h, we get:

$$\begin{aligned} \mathbf{Y}_{new} &= \mathbf{Y}_0 + h\dot{\mathbf{Y}}(t_0) \\ &= \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{y}_0 \end{bmatrix} + h\begin{bmatrix} -\mathbf{x}_0 \\ -k\mathbf{y}_0 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}_0 - h\mathbf{x}_0 \\ \mathbf{y}_0 - hk\mathbf{x}_0 \end{bmatrix} \\ &= \begin{bmatrix} (1-h)\cdot\mathbf{x}_0 \\ (1-hk)\cdot\mathbf{y}_0 \end{bmatrix} \end{aligned} \tag{3.14}$$

It becomes quickly apparent that if $|1 - hk| > 1$, the new y-component which is computed will have an absolute value larger than $|y_0|$. It will be further from the line $y = 0$ than when the simulation began! The constraint was that y remain on the line $y = 0$ and yet it will become further and further from its desired position as the simulation advances. Thus the simulation is said to be unstable. For the system to remain stable using this integration technique requires the imposition of specific admissible ranges for the time steps.

$$\begin{aligned} 1 > (1 - hk) > -1 \\ hk < 2 \end{aligned} \tag{3.15}$$

In other words, the time step must always remain less than $2/k$. This is a perfect example of stiff differential equations such as was presented earlier. If k is a large number and an explicit integration is used, then every step taken will need to be very small. This could have the effect of slowing down the simulation substantially since many updates must be computed to arrive at a certain time in the simulation. This problem can be reduced (larger time steps can be allowed) by using sophisticated explicit integration methods but the problem will

remain. Using an explicit integration, there will always be a maximum time step which the simulation must respect; otherwise the system will have stability problems.

3.3.3 Stiff Springs with Implicit Integration

Many of the stability problems caused by forward Euler integration can be solved by using implicit (backward) Euler integration. The concept is very similar to forward Euler integration (explicit) with the difference being that the current rate of change is used instead of the rate of change on the previous update.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\dot{\mathbf{y}}_{n+1} \qquad h\text{: step size} \qquad (3.16)$$

The method is called *backward* Euler since it uses a forward Euler step to run the system backward in time from the output state. This imparts an additional layer of consistency since an output state must be found whose derivative at least points back to where the value came from. This is in opposition to forward Euler which does not take notice of wildly changing derivatives and therefore proceeds forward quite blindly.

We then arrive at a simple implicit method for solving the differential equation. It is based on conceptually taking a step backwards and is given by the following equation (3.17):

$$\mathbf{Y}_{new} = \mathbf{Y}_0 + hf(\mathbf{Y}_{new}) \qquad (3.17)$$

We are essentially finding a point Y_{new} such that if time ran backwards, then the state vector would end up at Y_0. The problem lies in that it is impossible in most cases to evaluate f for Y_{new}. An approximation based on a Taylor series is used instead of $f(Y_{new})$.

$$\Delta\mathbf{Y} = \mathbf{Y}_{new} - \mathbf{Y}_0 \qquad (3.18)$$

$$\mathbf{Y}_0 + \Delta\mathbf{Y} = \mathbf{Y}_0 + hf(\mathbf{Y}_0 + \Delta\mathbf{Y})$$

Therefore,

$$\Delta\mathbf{Y} = hf(\mathbf{Y}_0 + \Delta\mathbf{Y}) \qquad (3.19)$$

The value $f(Y_0 + \Delta Y)$ is also approximated by

$$f(\mathbf{Y}_0 + \Delta\mathbf{Y}) = f(\mathbf{Y}_0) + f'(\mathbf{Y}_0)\cdot\Delta\mathbf{Y} \qquad (3.20)$$

We end with the equation

$$\Delta\mathbf{Y} = h\left(f(\mathbf{Y}_0) + f'(\mathbf{Y}_0)\cdot\Delta\mathbf{Y}\right) \qquad (3.21)$$

Which can be re-written as

$$\Delta\mathbf{Y} = \left(\frac{1}{h}\mathbf{I} - f'(\mathbf{Y}_0)\right)^{-1} f(\mathbf{Y}_0) \qquad (3.22)$$

Applied to the example shown earlier

$$f(\mathbf{Y}(t)) = \begin{bmatrix} -\mathbf{x}(t) \\ -k\mathbf{y}(t) \end{bmatrix} \tag{3.23}$$

We can solve

$$\Delta\mathbf{Y} = -\begin{bmatrix} \frac{h}{h+1}\mathbf{x}_0 \\ \frac{h}{1+kh}k \cdot \mathbf{y}_0 \end{bmatrix} \tag{3.24}$$

It can be seen that using this implicit integration technique will take longer to compute per update in the simulation than the simple explicit integration technique shown earlier. However, this integration allows for much longer time steps to be taken. For example, if the time step reaches towards infinity, the example would produce the following results (3.25).

$$\lim_{n\to\infty} \Delta\mathbf{Y} = \lim_{n\to\infty} -\begin{bmatrix} \frac{h}{h+1}\mathbf{x}_0 \\ \frac{h}{1+kh}k \cdot \mathbf{y}_0 \end{bmatrix} = -\begin{bmatrix} \mathbf{x}_0 \\ \frac{k\mathbf{y}_0}{k} \end{bmatrix} = -\begin{bmatrix} \mathbf{x}_0 \\ \mathbf{y}_0 \end{bmatrix} \tag{3.25}$$

We achieve our desired value in only one time step. This is the beauty of the explicit integrator, the ability to take very large steps and retain stability. This advantage makes up for the complexity of computations required at every update. Although every update will be slightly longer to compute, much larger time steps will be possible. Many techniques are available to speed up the complexity of computations required for explicit integration. Of note, methods for quickly solving sparse matrices have been used and recently documented [TAP97].

Implicit integration has been in use for many years now. Baraff and Witkin proposed an implementation [BW98] to be used in the development of cloth simulations. Their implementation produces accurate results and the simulation also runs at approximately the same speeds for very different stiffness coefficients on the springs. This is quite the opposite for explicit integration implementations in which the speed of the simulation depends on stiffness coefficients. The implementation of Baraff and Witkin still suffers from the odd instability problem however. They fix this problem by introducing *adaptive time stepping* into their system. Essentially, this consists of simulating the cloth with a certain step size, followed by searching the resulting state for drastic changes in stretch. If drastic changes are found, the proposed state is discarded, the step size is reduced and the simulation is run again. This technique works well for offline simulations but would undoubtedly cause problems in real-time and interactive virtual environments where the simulation must run at a quick and consistent pace.

The implicit method presented here is said to be *first-order accurate*. It is a robust technique but the results of most simulations can be refined by using higher-order methods of integration such as:

- Generalizations of the Runge-Kutta method
- Generalizations of the Bulirsch-Stoer method
- Predictor-corrector methods.

3.4 Lagrange Multipliers

The previous section on using stiff springs to maintain constraints worked by forcing particles to *want* to remain at a certain distance from each other. Once it becomes apparent that particles strayed from the proper distance from one another, stiff springs were introduced to bring them back to the proper distance; in other words, attempt to *fix* the constraints. The spring forces introduced are, however, no different than all the other forces already contained in the simulation. The spring forces have to compete for influence against user-interaction forces, gravity, etc. The stiff springs are given very large restitution coefficients in an attempt to win the battle with other acting forces and retain the most influence in the simulation. As was seen earlier, the large spring coefficients give rise to stiff differential equations which are difficult to solve numerically. Another fundamental difficulty with stiff springs is that the constraint forces only affect the bodies by dealing with displacement. When constraints are broken, a stiff spring is inserted between two particles in an attempt to offset the forces pulling the particles apart. For these reasons, stiff springs and other penalty methods are sloppy and imprecise constraint mechanism that are difficult to control.

The technique of *Lagrange multipliers* approaches the problem by introducing forces to maintain the constraints. These constraint forces directly affect the acceleration of the particles in a way which maintains constraints between connected bodies. While the penalty methods maintained constraints by correcting erroneous displacements, the technique of Lagrange multipliers maintains constraints by affecting the acceleration of particles. In other words, penalty methods only reacted when constraints were broken, by bringing the particles back together. The method of Lagrange multipliers inserts constraint forces to maintain constraints before they are broken.

Lagrange multipliers are important for computer graphics applications because they bypass the difficult and often intractable problem of parameterizing the degrees of freedom of a system. They are also valued for interactive computer graphics applications because they allow the combination of an arbitrary set of both holonomic and nonholonomic constraints. Finally, many other dynamics formulations can be found which use techniques similar to Lagrange multipliers:

- Newton-Euler
- Gibbs-Appel
- D'Alembert

- Gauss' Least Constraint Principle.

3.5 The Featherstone Algorithm

In contrast to the previously demonstrated methods which all made use of *augmented* or *maximal* coordinates, the *Featherstone* algorithm is based on a *reduced* or *generalized coordinate* method. This means the bodies in the system were described using the maximum number of coordinates. When using augmented coordinates, each part of the articulated body is represented by all of its state variables such that the specific part of the body can be defined in world space independently of any and all other bodies to which it is connected. Since this creates more coordinates than there are degrees of freedom, the bodies are not automatically confined to their *manifold*; the set of configurations which respect the constraints of the system. Constraint forces (i.e. the constraint Jacobian matrix) are therefore needed to enforce the system's constraints. Such as seen previously, numerical drift problems regarding the constraint forces will cause the system's constraints to be broken over time. A plethora [BW97, BW98] of methods are available to overcome these problems. Augmented coordinate methods can be advantageous since they allow an arbitrary number of constraints to be applied. They also allow for the use of nonholonomic constraints, such as constraints linking the velocities or accelerations of multiple parts of articulated bodies. A simple example of this is two gears connected together where one has a radius twice the size of the other. The constraint would be that a point on the perimeter of the larger gear would have a velocity twice that of a point on the perimeter of the smaller gear. Some also find augmented coordinate methods easier to understand than reduced coordinate methods. However, one of the major disadvantages to using augmented coordinates is that they generally have larger time complexities; $O(n^2)$ is a typical result.

The Featherstone algorithm is a *reduced coordinate* method, also known as *generalized* coordinates. Reduced coordinate methods use a number of coordinates to describe the system which is equal to the number of degrees of freedom. In this sort of system, a single part of an articulated body cannot be defined exclusively using its reduced coordinates since the only coordinates used will be those required to define the state of the body relative to its parent object. Since the configuration of objects will always be defined in relation to the state of its parent, none of the constraints can be broken; the system will always remain within its manifold and will never be configured in an invalid state. Since there are so few coordinates required to describe the system, this also means that the integration required to compute the coordinates' location is kept to a minimum. For this reason, reduced coordinate methods suffer a lower time complexity, often $O(n)$.

The main advantage of the Featherstone algorithm is its reduced time complexity. Since it is a reduced coordinate system it is impossible for the articulated body take an invalid state; the constraints are automatically enforced. The reduced number of coordinates required to describe the system means that there are fewer variables to compute and integrate over time, giving this algorithm a major speed advantage over

augmented coordinate formulations. It typically takes $O(n^2)$ time to solve the $n \times n$ matrix resulting from constraint forces in algorithms such as with Lagrange multipliers, although Baraff [Bar96] has recently shown that this can be done in O(n) time by making use of specialized techniques for sparse matrices.

However, augmented coordinate methods are generally considered to be more flexible than reduced coordinate methods such as that of Featherstone. Augmented methods allow for arbitrary system structures while Featherstone will only handle articulated bodies which have a tree-like shape. It was previously largely believed that Featherstone could not support kinematic loops, but Erleben [Erl05] has recently shown that this is possible. Importantly, augmented methods also allow quick and simple system structure changes on the fly. In contrast, reduced coordinate methods prefer that bodies be stored in trees that must be reconfigured when anything changes; a complex and time consuming process. Finally, the Featherstone algorithm only allows for geometric constraints (holonomic systems) which make it very difficult to model friction, contact constraints or to model connected gears.

Chapter 4

Modeling and Simulating Distributed Mass

This section will begin by discussing the modeling of an articulated object's mass for the four previously mentioned simulation techniques. It will also discuss how these representations are utilized inside the algorithms for each of the methods. A novel adaptation of the Verlet algorithm that generates more believable motion on articulated bodies in which the links have unequal mass is presented. Finally, experimental results from implementations of the stiff spring method and the adapted Verlet integration method will be compared. The comparisons will be done for articulated bodies with links of unequal mass in the context of interactive virtual environments.

The novel approach that will be presented is a means to simulate dynamics on articulated bodies in which each link is not of the same mass. Two different methods to model the mass of each of these links are presented here.

4.1 Modeling Distributed Mass on a Rigid Body

4.1.1 Collection of Point Masses

The simplest way to model mass on an object will be to use a collection of point masses on the segment of the body. Each of these point masses will consist of a weight and a location that is within the space of the object.

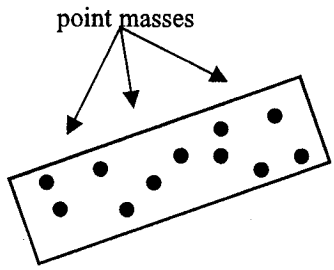


Figure 4.1: Object modeled as a collection of point masses.

Since all of the particles of one segment are part of one rigid structure, they will all maintain the same linear acceleration. To calculate this acceleration of a segment we will only require a single mass point. This mass point will be calculated as previously by adding the sum of all particles to obtain the total weight, and a weighted average will be used to obtain the position of the center of mass. The moments and products of inertia for each segment are computed using the formula for the inertia tensor presented earlier. Given the location of each point mass in relation to the center of mass (x_i, y_i, z_i) and the mass of each (m_i).

$$I = \begin{bmatrix} \sum_{i=1}^{n} m_i(y_i^2 + z_i^2) & -\sum_{i=1}^{n} m_i x_i y_i & -\sum_{i=1}^{n} m_i x_i z_i \\ -\sum_{i=1}^{n} m_i x_i y_i & \sum_{i=1}^{n} m_i(x_i^2 + z_i^2) & -\sum_{i=1}^{n} m_i y_i z_i \\ -\sum_{i=1}^{n} m_i x_i z_i & -\sum_{i=1}^{n} m_i y_i z_i & \sum_{i=1}^{n} m_i(x_i^2 + y_i^2) \end{bmatrix} \tag{4.1}$$

The total torque for one segment is obtained by adding the torque cause by each mass point. Note also that the total mass and total moments of inertia for the articulated body taken as a whole do not interest us at this point.

4.1.2 Volumetric Mass

Another solution to modeling distributed mass on a rigid body or a segment of an articulated body is to associate a continuous density to the body. Computing the total mass for this model is then done by integrating the density inside the volume.

$$m = \int \rho(\mathbf{r}) dV \tag{4.2}$$

Similarly, this method requires us also to integrate the density over the volume of the segment to obtain the moments of inertia with respect to a given axis. Typically, a simulation will compute the moments of inertia with respect to the three Euler angles of the body.

$$I = \int \rho(\mathbf{r}) r_{\perp}^2 dV \tag{4.3}$$

In this formula $\rho(\mathbf{r})$ is the density and $r_{\perp}$ is the perpendicular distance from the axis of rotation. The anisotropic properties of inertia oblige us to model it as a tensor (or matrix). The equation thus becomes [Wei05]:

$$I = \int_V \rho(x,y,z) \begin{bmatrix} y^2+z^2 & -xy & -xz \\ -xy & z^2+x^2 & -yz \\ -xz & -yz & x^2+y^2 \end{bmatrix} dxdydz \tag{4.4}$$

It is modeled as a matrix since the inertia of a body will be dependent on which axis it is spinning around. The matrix elements in the diagonal are called the *moments of inertia* and the others are often called the *products of inertia*.

4.1.3 Discussion

The model that uses volumetric mass requires us to perform multiple integrations, these are much more time consuming than the simple operations associated with the model of a collection of point masses. Given a change in mass parameters at run-time, the recalculations for mass and inertia would be faster for the model of a collection of point masses than it would be for a continuous mass distribution.

However, the computations of mass and inertia can be statically generated once before program execution, but this prohibits any modifications to the mass of the object to occur. Using volumetric mass to generate the values for the center of mass and moments of inertia is sometimes seen as providing an unnecessarily high level of detail for some interactive applications. It is also true that the discretization of mass can introduce accuracy errors into the model. The choice of model should be done based on the precision of results that must be achieved.

4.2 Simulating Distributed Mass

The following section deals with how dynamic physics simulations use their models and representations of virtual objects to create motions and behaviors that are physically believable. This thesis is mostly concerned with articulated bodies and therefore the constraints (joints or articulations) on the bodies will be our main focus for this section.

An important fact to note with the four different simulation methods presented in this thesis is that there is an important qualitative difference in how each of these techniques resolves their constraints. Of the four methods presented there are three which resolve constraints by introducing forces into the simulation whereas one method will resolve the constraints by directly manipulating the position and orientation of the limbs in the

articulated body. Indeed, the Verlet algorithm will modify object positions directly whereas the other three methods will only modify the state of the objects indirectly, by adding constraint forces into the system. Discussion of both of these categories will be done separately.

4.2.1 Methods Using Forces to Maintain Constraints

The method of stiff springs will simulate the constraints of an articulated body by introducing forces into the simulation when constraints have been broken; these are said to be *penalty forces*. The method of Lagrange multipliers (or the constraint Jacobian matrix) also functions by introducing constraint forces into the system only it will insert these to maintain the constraints *before* they are broken. But regardless of when these forces are inserted to help maintain the joints correctly, once they are a part of the simulation they can be dealt with just like any other force. Since distributed mass is still modeled as a scalar mass value positioned at the center of mass, it will only affect the stage of the simulation where the applied force to an object is transformed into linear and angular accelerations. The steps involved to compute acceleration can execute just as previously only it will make use of new values for the mass and center of mass that have been computed based on how the mass is distributed and modeled.

Because the constraint forces are dealt with just as any other force, it allows these techniques to carry on following very strict physical laws such as the one relating acceleration and force through mass. The errors in this system remain the errors in the model (this is true for any virtual environment) itself and in how the constraint forces are computed. When comparing how the forces are generated for stiff springs and Lagrange multipliers, the latter creates more faithful forces. This is due to the fact that stiff springs are simply reacting and trying to correct constraints once they are broken. The algorithm counts on constraints becoming broken; they are fixed later on. The constraint Jacobian matrix however inserts forces to maintain constraints before they are broken; in essence the constraints should never be broken.

An added advantage to these methods is that they view each object of the articulated body as a whole; which is how they actually are in the real world. The Verlet method will force us to decompose each body into a series of particles. This creates the problem of having to distribute mass across the particles and does not allow dealing with each link independently.

4.2.2 Verlet Integration

The method of Verlet integration does not resolve constraints by inserting forces into the system. Rather it makes use of a technique called relaxation, Gauss-Seidel iteration of Jacobi (based on how it is implemented) to resolve multiple constraints simultaneously. When using relaxation to resolve the constraints certain particles will have their positions modified directly without making use of any value of force or mass (see section 3.2 for details).

Because the position of objects can be modified directly it is frequently the case that the linear and angular velocities of the object are out of sync with the manner in which an object is moving. When the simulation is about to update there will be a resulting force on an object, this force is then used to calculate the acceleration which is then in turn used to compute the velocity of the object. At this stage a velocity has been computed and the position of the object is updated; this is the first stage of an update in the simulation. In the second stage the objects undergo rotations and translations while the constraints are resolved using relaxation. Because the position of the objects is modified directly, the velocity that was computed in the first stage of the simulation is no longer accurate. This is the reason why Verlet systems do not keep track of the velocity of the objects; they are said to be *velocity-less* representations.

Simple forward and backward Euler integration can also not be used because it makes use of the velocity of objects as part of their state that is used and updated every frame. Verlet integration will instead only store the current and previous positions of an object and will use the difference between the two as an approximation of velocity. In equation 4.5, $x_t - x_{t-1}$ estimates the previous velocity of the object.

$$x_{t+1} = 2\mathbf{x}_t - \mathbf{x}_{t-1} + \mathbf{a}\Delta t^2 \tag{4.5}$$

This creates a small error in the simulation since the velocity that is used is not extremely accurate. Just as Verlet systems do not store the velocity of an object, they will not store the angular velocity. Most models used with Verlet systems will represent everything using a particle-based framework that does not require any orientations or angular velocities. Instead of using position and orientation to represent an object in space, Verlet systems will model objects by placing particles with mass at the end points of the object. Figure 4.2 demonstrates the differences in how an object is modeled for the Verlet system.

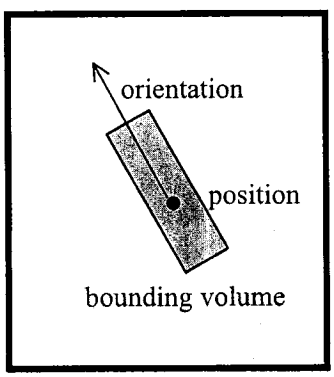


Figure 4.2a: Object modeled for a typical physical simulation.

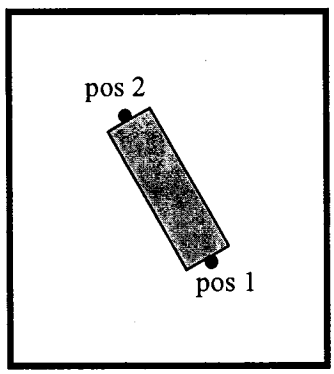


Figure 4.3b: Object modeled for a Verlet system.

In the Verlet system, the two particles making up the rod will remain at the proper distance from one another by addition of a constraint in the system. During the second phase of an update frame when the joint constraints are resolved via relaxation, the constraints to keep all particles at a certain distance from one another will also be resolved. However, there will be intervals of time where all constraints are not resolved very accurately.

This will cause some of the particles to be slightly out of place in relation to one another; they could either be too far or too close to one another. In the case of this model where the extremities of the object are modeled by particles, it will have the effect of slightly stretching or compressing the object itself. This is a considerable disadvantage to using this model.

The usage of particles to represent objects in the Verlet system also makes it difficult to control moments of inertia in the simulation. The particles that make up an object are still all considered and updated independently. The constraints that hold the particles of one body together are the same as the constraints that connect two bodies together via a joint, and they are handled in the exact same way. Each particle is handled independently and does not contain an orientation or an angular velocity; it only consists of a position and a mass. Because all the movements are simply linear movement of particles it becomes very difficult to insert angular friction into the simulation or specify a certain value for the moment of inertia of a body. By specifying unequal weights to the particles of a single body it is possible to give a different value of inertia to a body but it is only ever modified indirectly through the weights of the particles.

Because velocity and angular velocity are not properties of an object in the Verlet system, a different technique is required for simulating drag on a body. Drag is typically created by adding a force into the system that is proportional to the current velocity of an object multiplied by a negative constant. One option to add friction will be to use an approximation of the velocity as a base in the computation of drags that is then added as a force into the system. The value $x_t - x_{t-1}$ can be used to approximate the current velocity just as in equation 4.5. Another method proposed by [Jak03] to insert drag on a particle is to simply modify the 2 with 1.99 or any lower number in equation 4.5. By doing so, a small amount of energy will be lost in the system, giving the impression of air friction. In any case, both methods for creating friction in the Verlet system are slightly flawed. Both rely only on approximations of the current velocity and it is therefore difficult to be very precise in the amount of drag that a body contains.

In previous articles the constraint resolution step in Verlet system has not accounted for the mass of the links simulated, the following section presents a novel technique for Gauss-Seidel constraint resolution that is relative to the mass of each limb.

4.3 Weighted Constraint Resolution

For the Verlet system, constraints between two connected bodies are maintained by simply translating the two connected particles. The first step consists of calculating a vector that is the separation between the two particles; the second step is to set the position of both particles to be equal to the mid-point of the separating vector. The result is that now both connected particles hold the same position and the constraint is maintained.

Following this, other constraints are then fixed and will break once again the condition for this connected body. These steps will be repeated many times in the relaxation algorithm.

$$pos_1' = pos_2' = \frac{pos_1 + pos_2}{2} \tag{4.6}$$

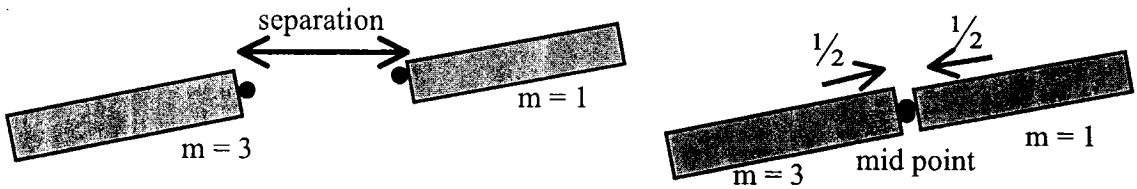


Figure 4.3: Constraint resolution in Verlet systems.

The number of iterations that are performed on all constraints can be affected by many factors. The number of iterations can be a fixed amount; it can keep on iterating until the delta from one iteration to the next is lower than a certain epsilon, or in the case of a hard real-time system, iterations can be performed until the solver is required to return a result.

Previous implementations of the Verlet system do not take the mass of each particle into account during the step where constraints are resolved via relaxation. A novel approach is presented here in which a weighted constraint resolution step is performed to factor in the mass of each body. In the case of two bodies that are disjoint when they should be connected, instead of translating both bodies by the same amount each is translated by an amount that is inversely proportional to their mass. An example is presented in figure 4.4.

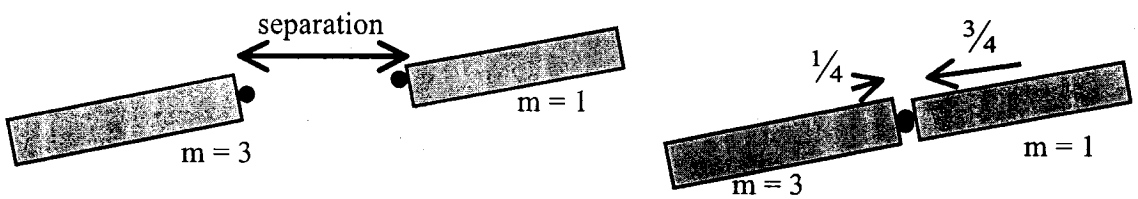


Figure 4.4: Weighted constraint resolution for Verlet systems.

In the simulation, equation 4.6 presented earlier will be replaced by the following equations.

$$pos_1' = pos_1 + \frac{vDelta * m_2}{m_1 + m_2} \qquad pos_2' = pos_2 + \frac{vDelta * m_1}{m_1 + m_2} \tag{4.7}$$

In these equations, the mass of each body is now properly taken into account to represent how more energy is required to displace an object with a large mass in comparison to a lighter object.

If we take an example of a simulation containing two rods that are connected at one end, in a Verlet simulation there are times when the two rods will become disjoint, one case in which this could occur is if a large force is applied to only one of the two segments. During the first phase of the simulation one of the links would translate over a certain distance while the other one would remain in its position. We can compare the results obtained when resolving the constraints of both bodies being connected via the regular method and when using weighted constraint resolution by viewing figures 4.3 and 4.4.

In the simulation performed with regular constraint resolution, both objects traveled the same distance to resolve the constraint. In the simulation performed with weighted constraint resolution, the object with most mass only traveled a fraction of the distance that the lighter object traveled. The result that was obtained with weighted resolution resembles much more the results that would have been obtained if the method of stiff springs was used. The spring would apply the same force (but in opposite directions) to both objects and the object with less mass would travel a longer distance before the condition of having both bodies attached was resolved.

To further prove the value of the technique presented, we have compared experimental results obtained from a simulation using stiff springs and one using Verlet integration with weighted constraint resolution. The main focus is on simulating articulated bodies with links of unequal mass, therefore we varied mass across the links that make up the articulated bodies. Our experimental results indicate that the Verlet system with weighted constraint resolution generates an output that is similar to those obtained with stiff springs but without any of the stability problems associated with it. The experimental results are presented and discussed in the following section.

4.4 Experimental Results

4.4.1 Implementation of Stiff Springs

Implementation of a simulation for an articulated body using stiff springs was based off the framework for rigid bodies presented in [DPW03]. The creation of the stiff spring class was fairly straightforward as it only required pointers to joints on the inboard and outboard objects. The springs could then be added to the physics simulator where they would apply forces to both of their objects based on how far apart they were situated.

A simple forward Euler integrator was used for the physics engine. This caused problems early on since as the spring constant grew, the articulated body showed signs of instability. In our case, instability began occurring when the spring value was around 0.027 N/cm. Section 3.3 outlines how instability is caused by a combination of large spring constants and also large step sizes. This was observed during our experimentation; when the program was run on a low-end PC, the spring constant had to be reduced because the program needed to take larger time steps (each step took a longer period to compute). Because of this instability when time steps are

large, if the application were to be released to a wider public and therefore a larger range of PC's, two options could be taken to avoid simulations blowing up due to instability:

- Put a maximum value on the step size that can be taken; if a certain PC is not fast enough for this step size, the entire simulation would be slowed down.
- Set minimum requirements on the type of PC that can run the simulation; to ensure that the step size never goes below a certain value.

On our test setup consisting of 5 links of unequal mass connected in a linear fashion, the program outputted an average of 602[1] frames per second with most graphical treatments turned off.

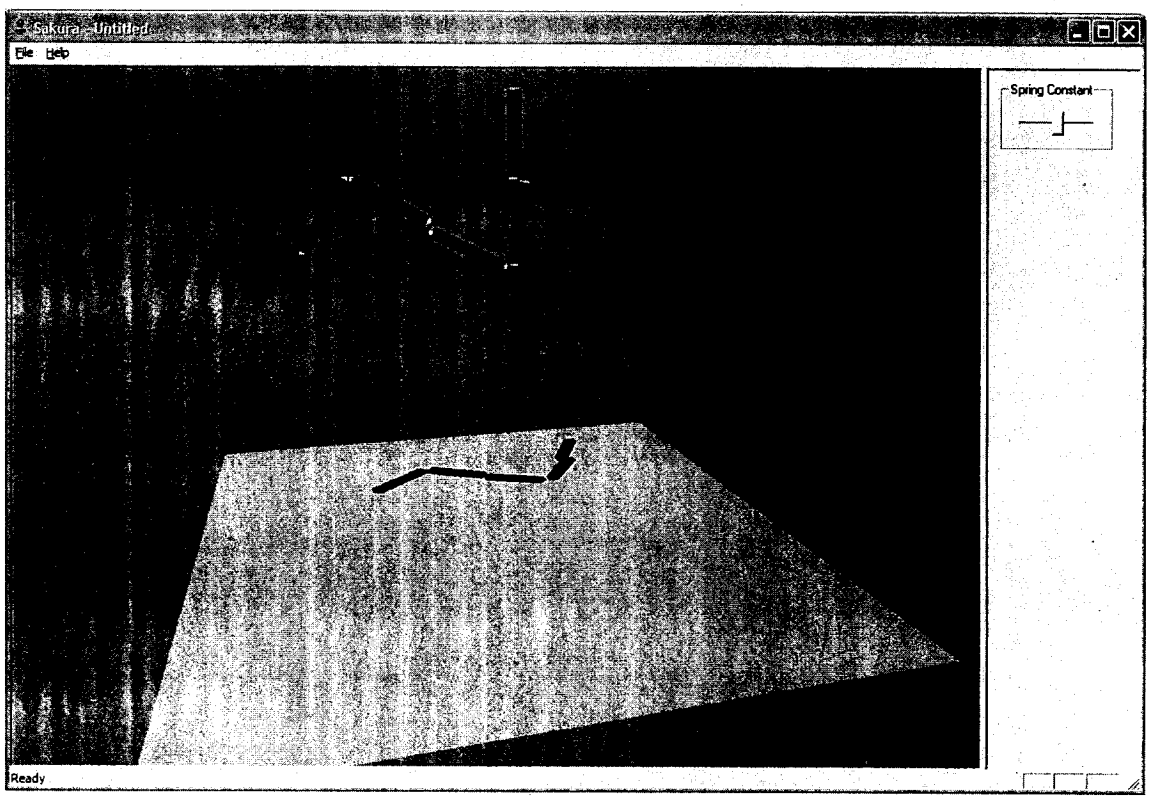


Figure 4.5: Screenshot of stiff spring simulation.

When observing the graphical results on screen one anomaly that is very noticeable is that the constraints of the bodies remaining connected together is not met entirely. By observing figure 4.5 a slight separation between each body can be noticed. The problem is most apparent on the connection between the first and second links at the top. The separation is the worst at that location because it is the springs that are supporting the most weight; it must support the weight of the four lower links while in comparison the lowest spring in the hierarchy must only support a single link. This problem could be partially solved by increasing the spring constants but this sometimes caused instability in the simulation. It quickly became a balance between having a simulation that is

[1] Tests were performed on an AMD Athlon™ XP 1800+ with 768 MG of RAM.

stable for long step sizes and having less extension on the stiff springs. Some stability issues could have been solved by using an implicit integrator such as backward Euler integration (instead of forward Euler integration) but this was not explored for the prototype.

4.4.2 Implementation of a Verlet System with Weighted Constraint Resolution

For the Verlet system implementation, the physics simulator presented in [DPW03] was thrown away because it was deemed to have too many differences with the particle-based system used for Verlet integration. A new class was created for representing the current state of an object. This class did not contain angular or linear velocity, an orientation or even drag coefficients; instead it contained a collection of particles each with their own position and mass, and a series of constraints pertaining to the relative positioning of each particle.

The Euler integrator which transferred acceleration to velocity to position updates was also thrown away in favor of the Verlet integration formula presented in [Jak03]. The simulator worked consistently and was very stable throughout the experimentation period. Even for tests on lower-end workstations, the simulation remained stable. The Verlet system was also able to maintain all the constraints between bodies with ease; there was never any noticeable gap between the connected bodies.

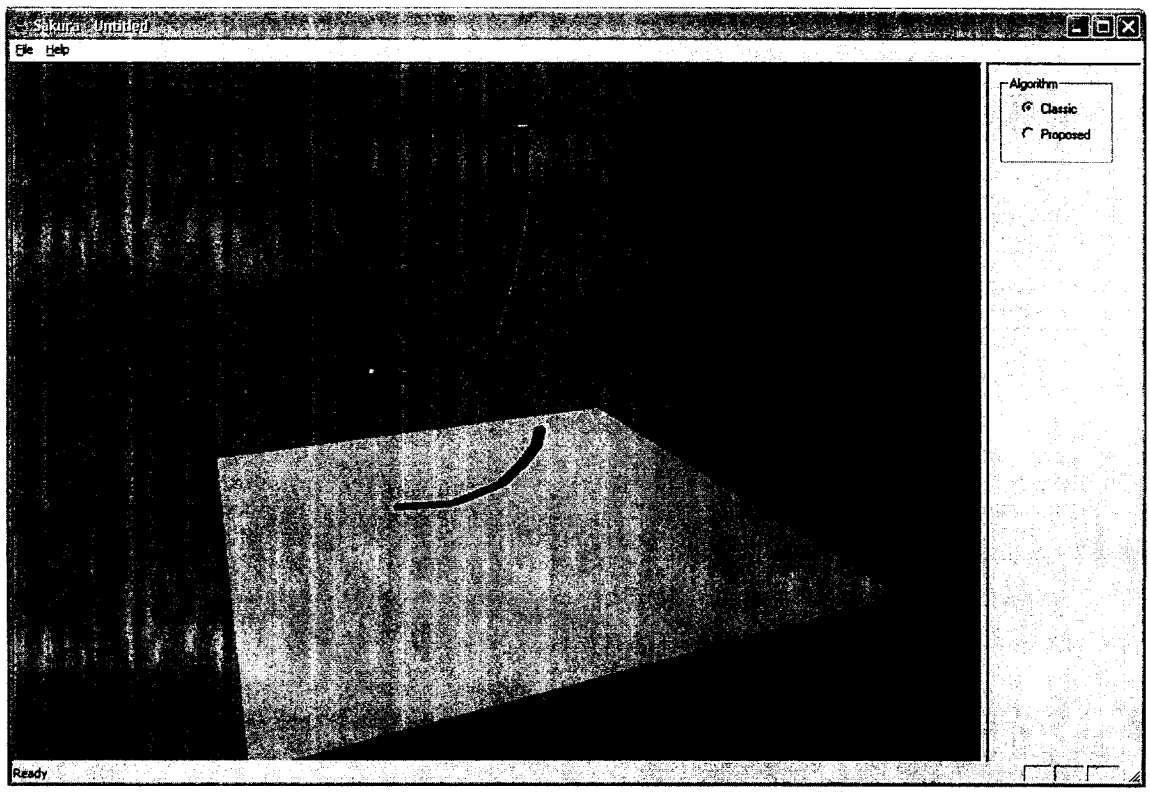


Figure 4.6: Screenshot of Verlet simulation with weighted constraint resolution.

It is difficult to categorize the performance of the Verlet system because its computation period can be tuned to be longer or shorter based on the amount of iterations taken during the relaxation period. The program outputted 597 frames per second when only 5 iterations were performed with the constraint solver; it outputted 30 frames per second[1] for 1000 iterations with the solver.

The program also contains a run-time switch to toggle between traditional constraint resolutions [Jak03] and the weighted resolution approach presented in section 4.2. For the experimentation, the four top-most links had the same mass while the link on the bottom had its mass doubled. With the traditional approach to constraint resolution, from simply viewing the motion of the body it is difficult to observe any difference in mass between the objects. With the approach of weighted constraint resolution, a difference in how the articulated body behaves can be observed. Notably the end link (with a larger mass) seems to have more control over the global motion of the articulated body, the links also take a longer period before becoming motionless due to the large amount of weight at the end of the structure; neither of these phenomena were noticeable with the simple constraint resolution.

A significant disadvantage of the Verlet system is that it was extremely difficult to tune or modify the value of drag for the objects in the virtual environment. This was done by reducing the factor 2 in equation 4.5 but it was often hit-and-miss. Modifying the number of iterations done by the constraint solver also seemed to have a noticeable effect on air drag but this was not investigated further.

4.4.3 Comparison and Discussion of Results

In this section, the experimental results obtained from the physical simulators built with stiff springs and a Verlet system containing weighted constraint resolution are compared. While it is difficult to quantify or compare the motion performed by the links of both simulations, the differences between the two will be explained qualitatively. The two implementations are also compared on computational complexity and on the control and capabilities of the algorithm in question.

In terms of computational complexity both algorithms performed similarly. The stiff springs algorithm created an average of 600 frames per second while the Verlet system with 5 iterations of constraint resolution gave similar results. The Verlet system could also trade-off some speed for accuracy; a more precise result was obtained when 1000 iterations were performed by the constraint solver but the program could only output 30 frames per second on average. In this area there is no clear winner between the two algorithms. An interesting experiment would be to implement various other integration algorithms for the stiff springs method and compare results again. Other algorithms such as Runge-Kutta could create better results but would also take a longer period to complete.

---

[1] Tests were performed on an AMD Athlon™ XP 1800+ with 768 MG of RAM.

In both simulations the larger mass on the final link could be noticed by the motion of the links. The final object in the articulated body had much more pull than any of the other limbs and therefore the lighter links often had to modify their trajectory to follow the heavier mass. One area where the Verlet system greatly outperformed stiff springs was in the ability to keep the joint constraints satisfied. In the stiff spring simulation the constant could never be increased enough to allow the resolution of the joint constraints; gaps were noticeable between links which should have been connected. The Verlet system had no troubles with the joint constraints. It is also interesting to compare the behavior of the Verlet simulation when it uses normal constraint resolution and when it uses weighted constraint resolution. In the first case, there is no noticeable difference in the behavior of the system if the masses of the various objects are unequal. This is due to the fact that when the constraints are resolved, the mass of each object is not factored in to the modifications in any way. When weighted constraint resolution is used, objects of larger mass will be noticed because they have more influence over the global movement of the articulated body; the other links have a tendency to *follow* the heavier links much more.

Both implementations have their advantages and disadvantages regarding control and capabilities. The difficulty in controlling moments of inertia and air friction for the Verlet system are one of the drawbacks. The moment of inertia is difficult to specify because the model does not store the orientation of the body, only a set of particles. The air friction is difficult to specify also with much precision because it would be calculated with only an approximation of the current velocity. Since the model that is used for the stiff springs contains both orientation and velocities for the body, moment of inertia and air friction can be specified with ease. In terms of types of joints (revolute, rotational, prismatic) it is the Verlet system that is most effective. As outlined in section 3.2, many types of constraints can be created by simply arranging the particles that make up the body in certain ways. No special logic is required in the program and no virtual forces must be created to maintain these constraints. It is conceivable for these constraints to be created using stiff springs only new types of springs would need to be created. For example, in the case of a prismatic joint, a spring that creates penalty forces for only two of the three world axes would be required in order to allow movement along one axis but not along the two others.

Tests for the scalability of both implementations were also performed to determine how both would perform when simulating larger amounts of links. The results of these tests are presented in figures 4.7 and 4.8. The method of stiff springs is able to maintain a very high number of frames; it can also maintain this level until the simulation reaches approximately 40 connected limbs at which point the frame rate begins to drop following a linear curve. The Verlet technique begins at approximately the same level as that of elastic springs only it begins to drop following a linear slope as links are added to the simulation. This leads us to conclude that the method of stiff springs is much more scaleable than the Verlet technique, but both methods are comparable for small amounts of connected segments.

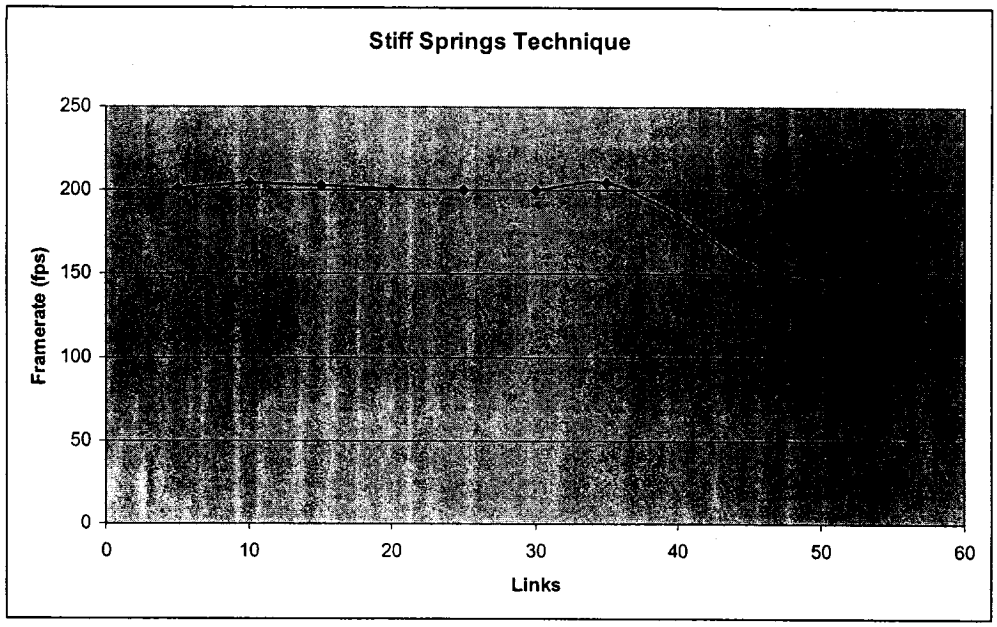


Figure 4.7: Number of frames per second achieved in our implementation of stiff springs based on the number of links in the simulation.

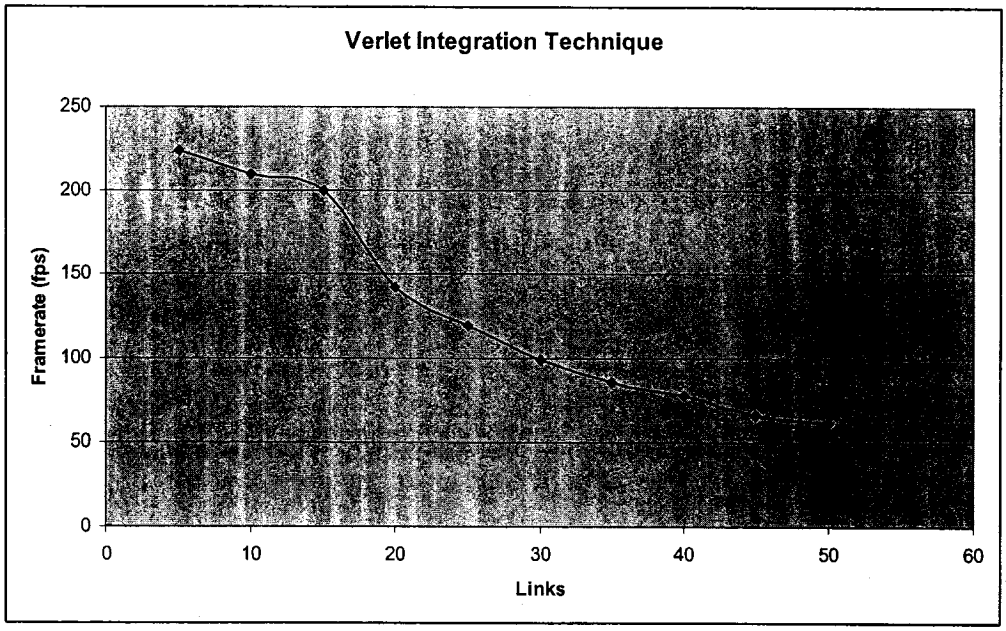


Figure 4.8: Number of frames per second achieved in our implementation of verlet integration based on the number of links in the simulation.

In conclusion, neither of the two techniques is a clear winner over the other. If the requirements of the virtual environment are that results be very precise and controllable, an approach of stiff springs may be better over a Verlet system. If the virtual environment needs an algorithm that allows for time-accuracy trade-offs and can handle multiple types of joints, a Verlet system is the preferred option. For virtual environments that are used

for real-time 3D animation packages of for interactive video games we believe that the Verlet system is the clear winner. Both these types of applications are meant to run on as many desktop computers as possible, meaning that it is very important to pay attention to the amount of memory that the application uses as well as the number of CPU cycles that it requires. Both algorithms presented use approximately the same amount of memory but the Verlet algorithm uses a lot less CPU cycles and is much more stable that the simulation that uses stiff springs to model articulations. Because of this we believe that the Verlet algorithm presented is the better option when performing physical simulations in real-time and interactive environments that run on performance-limited machines.

4.5 Development of a 3D Articulated Humanoid under the Impact of an Elastic Ball

4.5.1 Rationale

To place the discussion and implementations of the preceding chapters in context, an interactive program for animating and modeling a 3D articulated humanoid under the impact of an elastic ball was developed. The program makes use of the Verlet technique for simulating the articulated body along with the novel algorithm shown in section 4.3 for resolving constraints between limbs and for handling collisions. The software allows the user to propel an elastic ball towards a 3D humanoid model and observe how the model reacts as it falls to the ground passively. The animation of the humanoid falling to the ground is computed in real-time based on the impact parameters with the ball. Here are some of the parameters that can be modified by the user:

- The velocity and direction of the elastic ball.
- The radius of the ball.
- The elasticity of the ball.
- The type of human body (thin, medium or wide).
- The friction coefficient between the body and the ground.

The ultimate goal is to develop a model of the human body skeleton that is accurate enough to be used to gather data about how it would react to certain types of collisions in certain situations. If a model was developed to be accurate enough in how it simulates the human body, then thousands of dollars could be saved each year by replacing real people testing real impacts by virtual actors in virtual environments. For example, money could be saved by replacing crash test dummies used in cars by collision testing by virtual humanoids in virtual cars. The model could be used in any situation for observing the reactions of the human body to certain objects, at a fraction of the cost that it would take to create the real environments. This program consists of a first step towards achieving this goal, by modeling the skeleton of the human body via weighted limbs and spherical articulations. Another step that would be required to more accurately model the human body would be to simulate how the muscles of the human body react to certain types of situations. In this demo we model the articulations and bones (the skeleton) of the human body but none of the muscles. Once the ball collides with

the humanoid model, each articulation becomes entirely passive and only reacts to gravity, collisions with the ground and constraints with other limbs. In other words, the model makes no effort whatsoever to return to the relaxed standing state; it could be considered to be essentially a dead body. Future versions of the software should look into adding muscular resistance to each of the articulations.

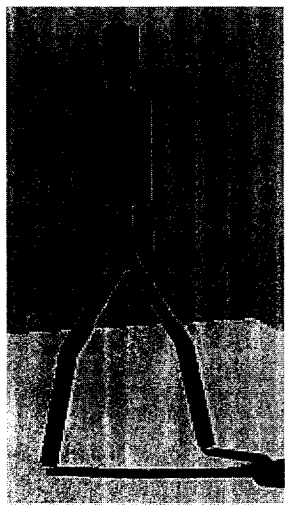
4.5.2 Humanoid Body Model

The model that was used for the human body was fairly simple; it consisted of 9 limbs which were all connected via spherical joints. The following limbs were represented:

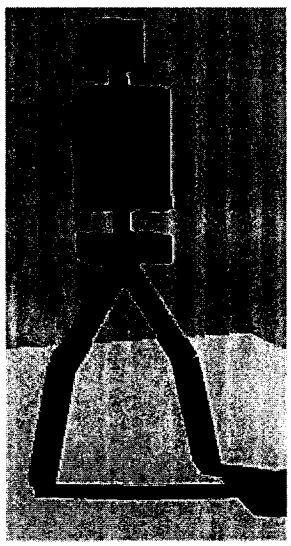
- head
- neck
- upper torso
- spine
- basin
- right upper leg
- right tibia
- left upper leg
- left tibia

The model did not contain any arms due to a limitation of how the model of a limb was implemented: The model only allowed for articulations to be placed at either extremity of the segment. This made it impossible to place arms on the sides of the upper torso. This constraint of only having the ability to connect other limbs at an extremity of another limb could be fixed in later versions of this program.

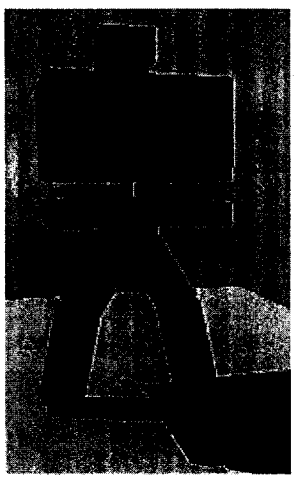
Each limb is modeled as a cylinder that has its own height and width. Each limb also contains a value for mass that is used by the simulation to compute how it will react to various collisions and forces from the environment. The mass of each limb in each body type is what causes each type of body to react differently to collisions with the ball. The height and width of the limbs are specified via values in the code but the mass was calculated as being proportional to the volume taken by each limb. Modifying any of these properties at run-time is possible but the program does not take advantage of this functionality. If the simulation needed to represent limbs that become broken and lose some parts from them, it would possible to model this by simply reducing the mass of the segment in question during run-time. The software also allows the user to choose between 3 different body types: thin (light weight), medium and wide (heavy weight). These allow the user to observe how various types of bodies would react to the impact. The proportions and mass of each limb in each of these body types is represented in figure 4.9.



Thin Body Type	
Limb	**Mass (lb)**
Head	20.0
Neck	5.0
Upper torso	84.0
Spine	5.0
Basin	21.0
Right upper leg	30.0
Right tibia	30.0
Left upper leg	30.0
Left tibia	30.0



Medium Body Type	
Limb	**Mass (lb)**
Head	25.0
Neck	5.0
Upper torso	140.0
Spine	5.0
Basin	35.0
Right upper leg	45.0
Right tibia	45.0
Left upper leg	45.0
Left tibia	45.0



Wide Body Type	
Limb	**Mass (lb)**
Head	40.0
Neck	5.0
Upper torso	350.0
Spine	25.0
Basin	87.5
Right upper leg	90.0
Right tibia	90.0
Left upper leg	90.0
Left tibia	90.0

Figure 4.9: Comparison of the 3 different body types and their properties.

Having the ability to specify the mass of each limb was very important to obtain better simulation results. If the limbs were of the same mass in each body type the differences when colliding with the thin (light) and wide (heavy) bodies would not be very noticeable. It is the different masses in the bodies that make them react differently to the collisions.

The articulations between each limb are modeled as spherical joints. This causes some problems in the simulation since there are articulations that move and bend in ways that the human body cannot. For example the knee articulation can be bent both front to back and back to front; this is not possible for the human body. Adding the ability to model different types of articulations could be made in future versions of the software.

4.5.3 Implementation Details

The program was a continuation of the work presented in [DPW03]. It was built from the ground up in C++ with OpenGL for some rendering capabilities. It allows for real-time modifications of object parameters and is entirely interactive. In [DPW03] the user could interact with the environment by applying forces to various objects via key presses. In this program, the user interacts with the simulation by modifying properties of the ball (via sliders on the right panel) and by selecting the body type to be used (via push buttons on the right panel). The user can also specify the velocity at which the ball should be launched and the friction coefficient between the body and the ground. Finally, the camera view in the perspective window can be manipulated in a similar way to most 3D animation and modeling packages. Hitting the left or right mouse keys and dragging will change the perspective of the camera; using the mouse wheel will cause the camera to zoom in and out.

Another point that can be noticed is that the body is motionless and standing upright until the ball hits it. What is in fact happening is that no physics are being simulated on the body up until it first collides with the ball. Once the collision with the ball occurs, each articulation is turned on and the physics simulator begins computing new positions for it. This is done for two reasons; the first is that it would have taken lots of effort to apply the appropriate forces to each limb that would allow the model to remain perfectly upright; it was deemed out of scope for this project. The second reason is to optimize the simulation and have it run at a faster rate by not simulating objects that are at rest. The simulation saves many CPU cycles by not computing any simulation updates for objects that are at rest; these objects will *wake* and start being simulated by the engine as soon as another object collides with them or when a new force is applied.

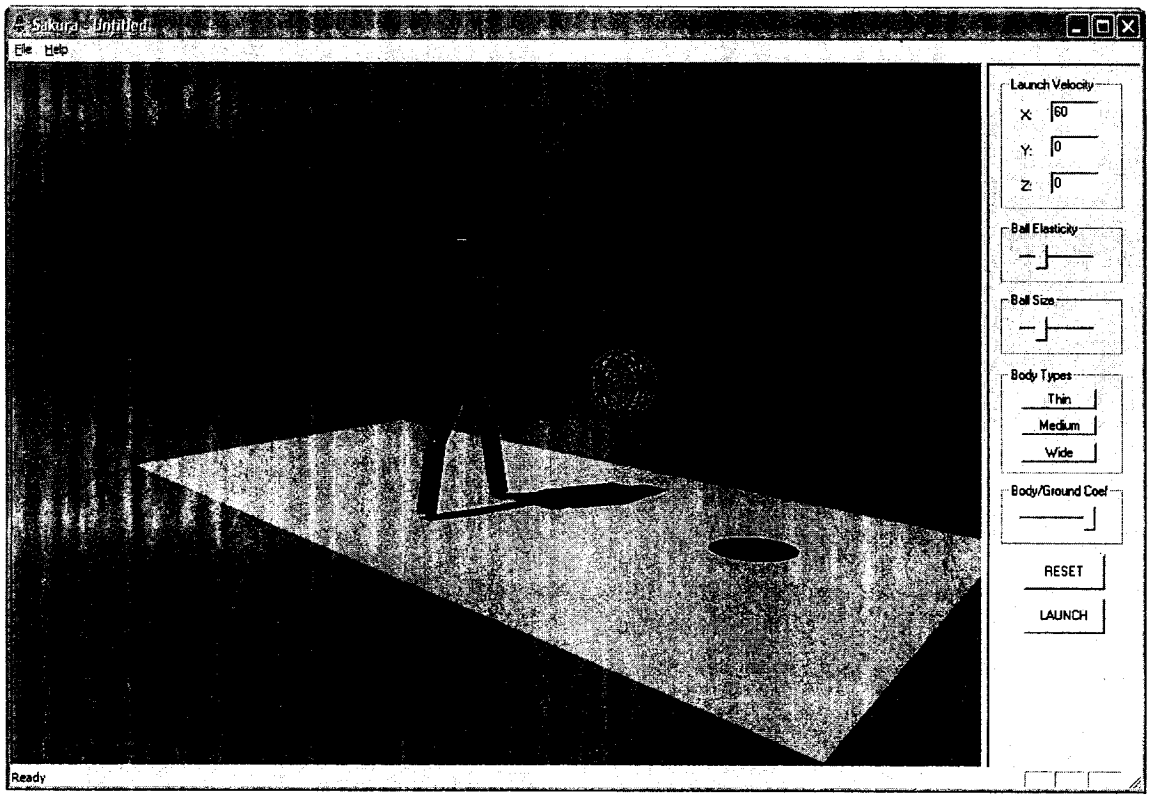


Figure 4.10: Screenshot of the user interface.

4.5.4 Experimental Results

The novel algorithm for resolving constraints in the Verlet algorithm presented in section 4.3 helps to accurately simulate the different mass in each limb of the humanoid body in this case. When the ball is propelled to hit either leg of the body, the difference in mass between the leg and the torso is noticeable because the leg will go flying backwards or sideways while the torso may move only a little. If the mass in the torso were very small (smaller than the mass in the legs) and the ball was hit on the leg, the entire body would be pulled backward by the leg at the same rate. The following section discusses some other representative tests that were done using the software demo.

- **Comparison of a humanoid being hit at small and large velocities.**

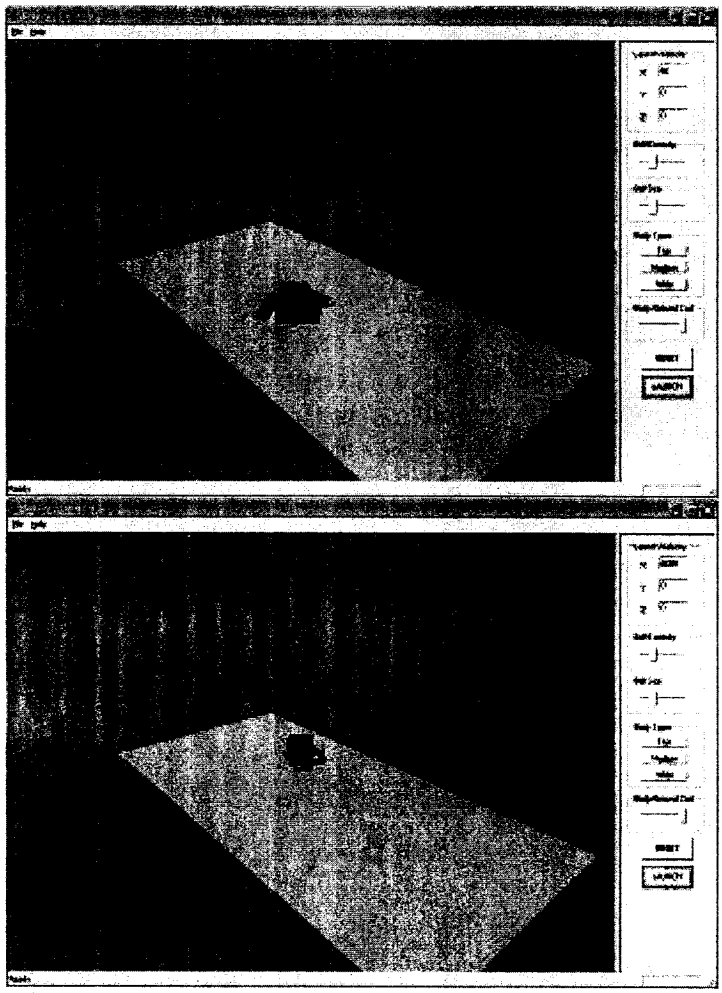


Figure 4.11: Comparison of a humanoid being hit at small (top image) and large (bottom image) velocities.

Figure 4.11 demonstrates the post-collision states of the humanoid in two different tests. In the first test, the body was hit by a ball with a very small velocity therefore not much energy was transferred over to the body and it did not travel a large distance before coming to rest. In the second test, the body was hit by a ball with a very large velocity therefore the body traveled a much larger distance before coming to rest. All of the other parameters remained the same in the simulation. This test demonstrates how a ball with a larger velocity will transfer more energy to the humanoid body.

- **Comparison of a humanoid being hit by balls with a low/large elasticity coefficient.**

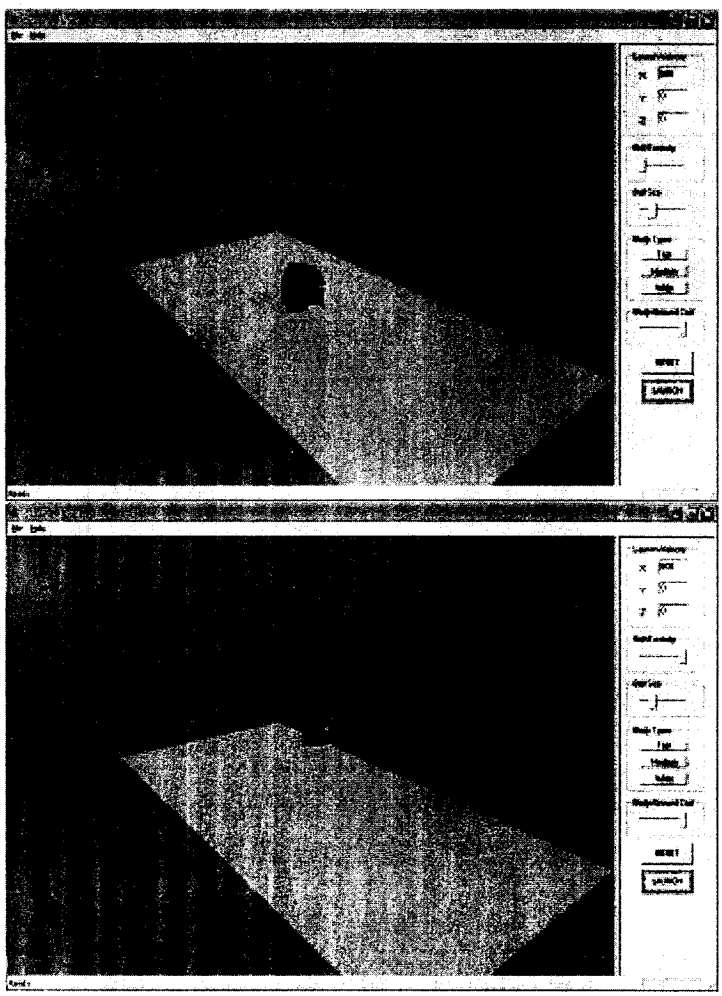


Figure 4.12: Comparison of a humanoid being hit by a ball with a low (top image) and large (bottom image) elasticity coefficient.

Figure 4.12 demonstrates the post-collision states of the humanoid in two different tests. In the first test, the body was hit by a ball with a very small elasticity coefficient meaning that lots of energy was lost (or absorbed by the ball) during the collision and therefore not much energy was transferred over to the body and it did not travel a large distance before coming to rest. In the second test, the body was hit by a ball with a very large elasticity coefficient; very little energy was lost in the collision therefore the body traveled a much larger distance before coming to rest. All of the other parameters remained the same in the simulation. This test demonstrates how a ball with a larger elasticity coefficient will not absorb much energy during collisions; instead more energy will be transferred to the humanoid body.

- **Comparison of a humanoid being hit by light and heavy balls.**

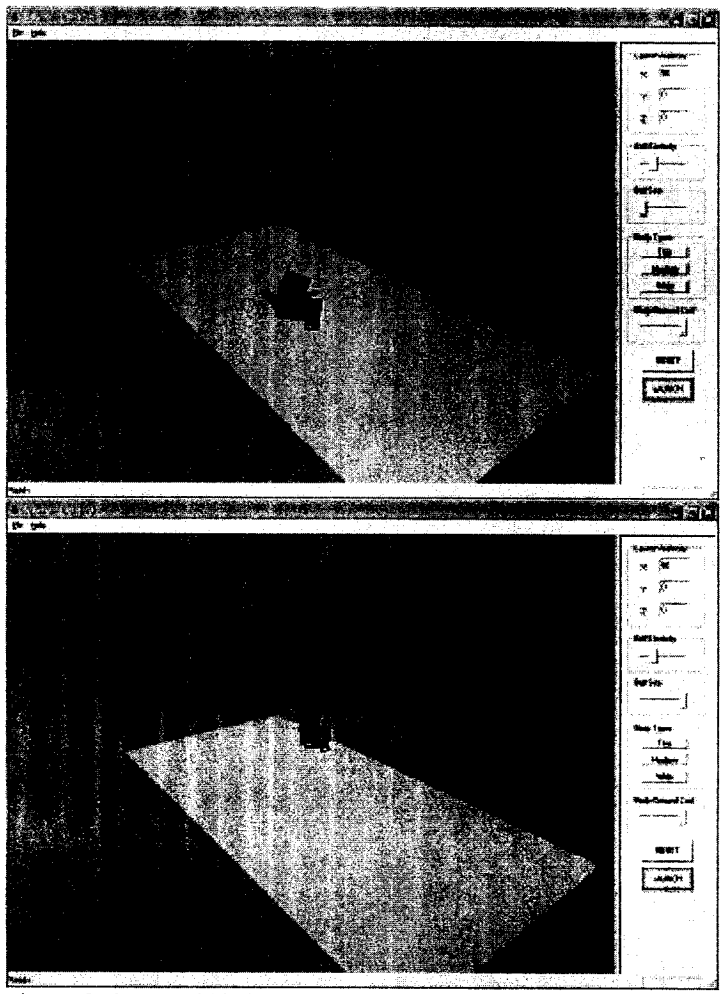


Figure 4.13: Comparison of a humanoid being hit by a small/light ball (top image) and a large/heavy (bottom image) ball.

Figure 4.13 demonstrates the post-collision states of the humanoid in two different tests. In the first test, the body was hit by a ball with a very small radius/mass meaning it only contained very little energy and momentum; when it collided, the humanoid body only traveled a very short distance. In the second test, the body was hit by a large ball with lots of mass; this ball was traveling at the same velocity as the small ball in the previous test only it carried much more energy because of its larger mass. When it collided with the humanoid skeleton much more energy was transferred therefore the body traveled a larger distance. All of the other parameters remained the same in the simulation. This test demonstrates how a ball with a larger mass will carry more energy before a collision, creating a more disruptive impact.

- **Comparison of light (thin) and heavy (wide) humanoids being hit by a ball at the same velocity.**

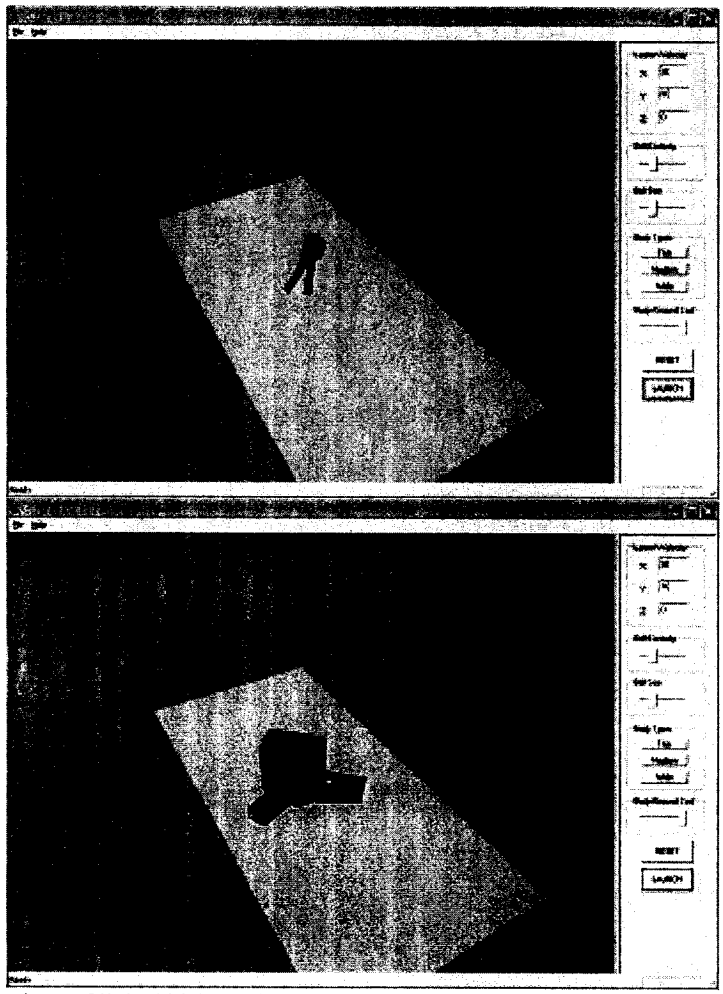


Figure 4.14: Comparison of light (thin) and heavy (wide) humanoids being hit by a ball at the same velocity.

Figure 4.14 demonstrates the post-collision states of the humanoid in two different tests. In the first test, a light (thin) body configuration was used and in the second test a very heavy (wide) humanoid body was used. In the first test, the body traveled a much larger distance after being hit by the ball than in the second test when a much more heavy body was used. All other parameters in the simulation remained the same for both tests. This demonstrates how an articulated body with a larger mass will travel a shorter distance than an object with a smaller mass when it undergoes a similar collision.

The program created also contains some other advantages; it is able to run at very quick frame-rates, achieving over 100 frames per second[1]. The simulation is very stable (limbs will never go flying all over the place) thanks to the Verlet algorithm that is used. These properties of the software being stable, interactive and running at very rapid frame-rates make the engine appropriate for use in real-time 3D virtual environments and 3D video games. The physics engine has already been utilized in a 3D racing game, *XRA: Xtreme Racing Action*, which was entered in the 2003 *Independent Games Festival* (www.igf.com), a competition for independent game developers held every year at the *Game Developer's Conference* (www.gdconf.com).

[1] Tests were performed on an Intel Pentium™ 1.60 GHz with 512 MG of RAM.

Chapter 5

Conclusion

We have examined various solutions to the problem of physically simulating articulated bodies. Representations for distributed mass are also mentioned before describing how these fit in with various physical simulations. Introducing distributed mass into the methods of stiff springs, Lagrange multipliers and the Featherstone algorithm requires little to no changes at all. For the technique of Verlet integration, the adaptation for distributed mass is a simple method of weighted constraint resolution, but it has not previously been made explicit.

Various methods of representing bodies, time-stepping simulations, integrating differential equations and representing and resolving constraints have been discussed, and their respective advantages and disadvantages analyzed. Implementation of two of the approaches for physically simulating articulated bodies have been briefly described along with their results. Of the methods presented some have been know for quite some time, notably the method of Lagrange multipliers has been known for quite some time but has only recently become popular for interactive applications thanks to the continuing increases in CPU power available on contemporary workstations. Papers and articles written years ago dismissed this method due to its computationally complexity, but this is no longer true today.

After studying several different implementations for physical simulations, some things have become apparent. A first observation is that all models and algorithms have problems of one type or another; whether these are restrictions in the types of constraints that can be modeled, or accuracy and performance problems, each one has its issues and not one approach seems uniformly superior to the others. Each solution provides advantages that certain applications could benefit from but an all-around solution without disadvantages still does not exist. Verlet integration has accuracy problems, stiff springs can only model certain types of constraints, Lagrange

multipliers are very computationally expensive and so is the Featherstone method. But even so, all of these methods have proven to be very effective in the right types of applications.

Our observations also lead us to believe that physical simulation of articulated bodies is very useful as a replacement for key framed animations in virtual environments for 3D animation or for computer video games. Many gaming consoles and computers available today have increased in the computational power that they have, but still have very limited amount of run-time memory with slow access times. This makes the solution of using physical simulation for generating motion much more appealing since it uses very little memory (while key framed data uses lots) and only slightly more computational power. Physical simulation also has the ability of generating any type of motion; whatever is needed for a particular situation whereas when using key framed animations each type of situation must be accounted for individually. Also, key framed animations will always be limited by the amount of animations that can be created by animators and by how many animations can fit in memory, while there is no limit to the types of motion that can be generated via simulation. Another advantage to using physical simulations is that no animators are required, all the motion and reactions of the virtual articulated body gets generated at run-time. For these reasons we believe that using physical simulation of articulated bodies is a good alternative to key framed animation in virtual environments for 3D animation and interactive video games.

For the field of computer animation and interactive video games we believe that our novel adaptation of the Verlet integration technique for simulation of articulated bodies with distributed mass is the most well-suited. The algorithm uses a very small amount of memory and also uses only a very small amount of CPU cycles while producing a result that is very visually believable and appealing. The algorithms of Lagrange and Featherstone both use much larger amounts of CPU cycles and the method of stiff springs does not create simulations that are believable because it allows for many articulation constraints to be broken temporarily. Finally when comparing physical simulation with the Verlet algorithm to the technique of simply using key-framed animations to give motion to the objects of the VE; we believe that our proposed algorithm will offer results that are much more adaptable to various types of situations all the while using much less memory and only slightly more CPU cycles. These advantages should be very valuable in the field of computer video games because these must have the ability of using very little memory and running very quickly on many different types of machines to allow for real-time user interaction.

Physical simulations for articulated bodies have evolved by leaps and bounds in the last years, making possible achievements that were not even thought of years ago. Many industries have benefited from this; visual special effects for movies can now procedurally animate millions of particles and hair on characters, vehicle makers can more faithfully create virtual prototypes of new products, and video game makers can embed new levels of realism into their virtual worlds. Many years ago, techniques for simulating rigid bodies were perfected, and as

we gain expertise in simulating articulated bodies researchers are beginning to look at the next logical steps such as simulation of deformable bodies [JF03].

5.2 Directions

There are plenty of problems still to be addressed. All of the methods described provide results which are good but have very little controls to modify the outputs. Whether it be artists and animators who wish to modify a result to make it more visually appealing, or a designer who must simulate more faithfully a certain phenomenon; users will require tools to modify the behavior of the simulation. The two major ways in which a user can modify the simulation are via the disposition and type of the joints, and changing the mass distribution. Controls that are slightly more subtle, such as inserting friction into a revolute joint for example, will need to be added as we move forward.

Another area of research is how to merge traditional key-framed animation in with procedural animation. Also, simply simulating the limbs of a falling human without any account for the underlying muscles and organs will not produce realistic results. How can realism in muscle-driven bones be achieved? Certain companies and researchers have begun tackling this problem, notably the *NaturalMotion* Company has released a product that simulates human movement via a physical simulation that controls muscles in a virtual human body.

Finally, would it be possible to insert into the simulation a certain level of artificial intelligence that would control the muscles of a virtual body to help it achieve certain movements? In such as case, could we create virtual actors that do not need to be animated, but only need to be given high-level commands.

Appendix A: Code Printouts for Stiff Springs Demo

```
/*****************************************************************
File name: IDynamic.h

Author(s): Jean-Christian Delannoy

Description: Interface for a dynamic (movable) object

documentation:

version: 0.001

*****************************************************************
*****************************************************************
TODO:


*****************************************************************
CHANGES:

JCD 03/10/2005 Created


*****************************************************************
*****************************************************************
Notes:



*****************************************************************
*****************************************************************
*****************************************************************
*****************************************************************/

#include "stdafx.h"
#include "IDynamic.h"
```

```
IDynamic::IDynamic() :
      vVelocity(0.0f, 0.0f, 0.0f),
      vAngularVelocity(0.0f, 0.0f, 0.0f),
      fMass(5.0f),
      fMomentInertia(10.0f),
      vTotalForce(0.0f, 0.0f, 0.0f),
      vTotalTorque(0.0f, 0.0f, 0.0f),
      fLinearFriction(0.025f),
      fAngularFriction(0.075f)
{
}

// Update velocity (both linear and angular)
void IDynamic::Update(const float& elapsedTime_)
{
      // Update forces.
      CVector vAcceleration;
      CVector vAngularAcceleration;

      vAcceleration = (vTotalForce / fMass);
      vAngularAcceleration = (vTotalTorque / fMomentInertia);

      vVelocity += (vAcceleration * elapsedTime_);
      vAngularVelocity += (vAngularAcceleration * elapsedTime_);

      // Update positions.
      Translate(elapsedTime_);
      Rotate(elapsedTime_);
}

void IDynamic::ResetForces()
{
      vTotalForce.SetXYZ(0.0f, 0.0f, 0.0f);
      vTotalTorque.SetXYZ(0.0f, 0.0f, 0.0f);
}

void IDynamic::ApplyFriction()
{
      // Apply the linear friction.
      CVector vFriction = -(vVelocity * fLinearFriction);
      ApplyLinearForce(vFriction);

      // Add angular friction to the ouboard joint.
      vFriction = -(vAngularVelocity * fAngularFriction);
      ApplyTorque(vFriction);
}

void IDynamic::ApplyLinearForce(const CVector& vForce)
{
      vTotalForce += vForce;
}

// Applies the specified force at the specific application point.
// (ApplicationPoint) is in relative coordinates.
void IDynamic::ApplyForce(CVector& vApplicationPoint, CVector& vForce)
{
```

```
	CVector vNormal = vApplicationPoint;
	vNormal.Normalize();

	CVector vNorForce = vNormal * (vForce.DotProduct(vNormal));
	ApplyLinearForce(vNorForce);

	CVector vTemp = vApplicationPoint.CrossProduct(vForce - vNorForce);
	ApplyTorque(vTemp);
}

void IDynamic::ApplyTorque(const CVector& vTorque)
{
	vTotalTorque += vTorque;
}

void IDynamic::Rotate(const float& elapsedTime_)
{
	vOrientation = vPreviousOrientation + vAngularVelocity *
elapsedTime_;
	CVector vDeltaOrientation = vOrientation - vPreviousOrientation;

	// Only rotate when necessary, it is not free!
	//if(fabs(vDeltaOrientation.x) > DO_NOT_ROTATE_THRESHOLD ||
	//	fabs(vDeltaOrientation.y) > DO_NOT_ROTATE_THRESHOLD ||
	//	fabs(vDeltaOrientation.z) > DO_NOT_ROTATE_THRESHOLD)
	{
		//rotate the model's axes.
		vTop.Rotate(vDeltaOrientation.x, vRight);
		//vFront.Rotate(vDeltaOrientation.x, vRight);

		vRight.Rotate(vDeltaOrientation.z, vFront);
		vTop.Rotate(vDeltaOrientation.z, vFront);

		vPreviousOrientation = vOrientation;
	}
}

void IDynamic::Translate(const float& elapsedTime_)
{
	vPosition += (vVelocity * elapsedTime_);
}

void IDynamic::Translate(const CVector& vTranslation_)
{
	vPosition += vTranslation_;
}

void IDynamic::RenderTotalForce()
{
	// Render the tail.
	glDisable(GL_LIGHTING);
	glDisable(GL_TEXTURE_2D);
	glColor3f(0.7f, 0.7f, 0.7f);

	// Get the points.
	CVector vForcePoint = vPosition + (vTotalForce * 50000.0f);
```

```
	// Draw the force.
	glBegin(GL_LINES);
		glVertex3f(vPosition.x, vPosition.y, vPosition.z);
		glVertex3f(vForcePoint.x, vForcePoint.y, vForcePoint.z);
	glEnd();

	glEnable(GL_TEXTURE_2D);
	glEnable(GL_LIGHTING);
}

void IDynamic::RenderTotalTorque()
{
	// Render the tail.
	glDisable(GL_LIGHTING);
	glDisable(GL_TEXTURE_2D);
	glColor3f(0.0f, 0.0f, 1.0f);

	// Get the points.
	CVector vForcePoint = vPosition + (vTotalTorque * 50000.0f);

	// Draw the force.
	glBegin(GL_LINES);
		glVertex3f(vPosition.x, vPosition.y, vPosition.z);
		glVertex3f(vForcePoint.x, vForcePoint.y, vForcePoint.z);
	glEnd();

	glEnable(GL_TEXTURE_2D);
	glEnable(GL_LIGHTING);
}
```

```
/************************************************************************
File name: CPhysicsEngine.cpp

Author(s): Jean-Christian Delannoy

Description: The physics manager for the stiff spring simulation.

documentation:

version: 001

************************************************************************
************************************************************************
TODO:


************************************************************************
CHANGES:
      JCD 03/10/2005 CREATED


************************************************************************
************************************************************************
Notes:



************************************************************************
************************************************************************
************************************************************************
************************************************************************/

#include "stdafx.h"
#include "CPhysicsEngine.h"

CPhysicsEngine::CPhysicsEngine()
{
      fLastTime = -1;
      fConstElapsedTime = -1.0f;
}

CPhysicsEngine::~CPhysicsEngine()
{
}

void CPhysicsEngine::ApplyPhysics()
{
      float fCurrentTime = CTimer::Instance()->getTimeElapsed();

      // Get time elapsed.
      if (fLastTime < 0 || (fCurrentTime - fLastTime) > 1000)
      {
            fLastTime = CTimer::Instance()->getTimeElapsed();
            return;
      }

      float fElapsedTime;
```

```
	if (fConstElapsedTime > 0)
		fElapsedTime = fConstElapsedTime;
	else
		fElapsedTime = fCurrentTime - fLastTime;

	// Set the previous time.
	fLastTime = fCurrentTime;

	// Apply gravity and friction forces here.
	int numobjects = vDynamicObjects.size();
	const CVector vGravity(0.0f, -0.00098f, 0.0f);
	for (int i = 0; i < numobjects; ++i)
	{
		vDynamicObjects[i]->ApplyLinearForce(vGravity);
		vDynamicObjects[i]->ApplyFriction();
	}

	// Apply the spring forces here.
	int numsprings = vSpringConstraints.size();
	for (int i = 0; i < numsprings; ++i)
	{
		vSpringConstraints[i]->Simulate();
	}

	// Move dynamic objects (account for forces and move objects)
	for (int i = 0; i < numobjects; ++i)
	{
		vDynamicObjects[i]->Update(fElapsedTime);
	}
}

void CPhysicsEngine::ResetForces()
{
	// Reset all the forces.
	int numobjects = vDynamicObjects.size();
	for (int i = 0; i < numobjects; ++i)
	{
		vDynamicObjects[i]->ResetForces();
	}
}

void CPhysicsEngine::AddDynamicObject(IDynamic* pDynamicObject_)
{
	vDynamicObjects.push_back(pDynamicObject_);
}

void CPhysicsEngine::RemoveDynamicObject(IDynamic* pObject_)
{
	int size = vDynamicObjects.size();
	if (size < 1)
		return;

	// Check the first object.
	std::vector<IDynamic*>::iterator iter = vDynamicObjects.begin();
	if (*iter == pObject_)
		vDynamicObjects.erase(iter);
```

```
	// Check the other objects.
	for(int i = 1; i < size; ++i)
	{
		++iter;
		if (*iter == pObject_)
			vDynamicObjects.erase(iter);
	}
}

void CPhysicsEngine::AddSpringConstraint(CSpringConstraint*
pSpringConstraint_)
{
	vSpringConstraints.push_back(pSpringConstraint_);
}

void CPhysicsEngine::RemoveSpringConstraint(CSpringConstraint*
pSpringConstraint_)
{
	int size = vSpringConstraints.size();
	if (size < 1)
		return;

	// Check the first object.
	std::vector<CSpringConstraint*>::iterator iter =
vSpringConstraints.begin();
	if (*iter == pSpringConstraint_)
		vSpringConstraints.erase(iter);

	// Check the other objects.
	for(int i = 1; i < size; ++i)
	{
		++iter;
		if (*iter == pSpringConstraint_)
			vSpringConstraints.erase(iter);
	}
}
```

```
///*****************************************************************
//File name: CSpringConstraint.cpp
//
//Author(s): Jean-Christian Delannoy
//
//Description: A class for encapsulating spring constraints
//
//documentation: none
//
//version: 0.001
//
//*****************************************************************
//*****************************************************************
//TODO:
//
//
//
//*****************************************************************
//CHANGES:
//
//JC 29/01/2006 Created
//
//
//*****************************************************************
//*****************************************************************
//Notes:
//
//
//*****************************************************************
//*****************************************************************
//*****************************************************************
//*****************************************************************/
//
#include "stdafx.h"
#include "CSpringConstraint.h"

CSpringConstraint::CSpringConstraint(CCylinder* inboardJoint_)
{
	inboardJoint = inboardJoint_;

	fFrictionConstant = 0.1f;
	fSpringConstant = 0.0027f;
}

CSpringConstraint::~CSpringConstraint()
{
}

void CSpringConstraint::AddOutboardJoint(CCylinder* outboardJoint_)
{
	vOutboardJoints.push_back(outboardJoint_);
}

void CSpringConstraint::Simulate()
{
	CVector vInboardTail = inboardJoint->GetTail();
```

```
	// Update each connected joint.
	int size = vOutboardJoints.size();
	for (int i = 0; i < size; ++i)
	{
		CVector vOutboardHead = vOutboardJoints[i]->GetHead();
		CVector vSeparatingDist = vOutboardHead - vInboardTail;

		// Calculate the spring force value.
		float fForce = vSeparatingDist.length() * fSpringConstant;
		CVector vForce = vSeparatingDist;
		vForce.SetLength(fForce);

		// Apply the force. (the application point must be relative
tothe object's center.
		CVector vAppPoint = vInboardTail - inboardJoint-
>GetPosition();
		inboardJoint->ApplyForce(vAppPoint, vForce);

		// Invert the force and apply it to the outboard joint.
		vForce = -vForce;
		vAppPoint = vOutboardHead - vOutboardJoints[i]->GetPosition();
		vOutboardJoints[i]->ApplyForce(vAppPoint, vForce);
	}
}

void CSpringConstraint::Render()
{
	// Render the tail.
	glDisable(GL_LIGHTING);
	glDisable(GL_TEXTURE_2D);
	glColor3f(0.0f, 1.0f, 0.0f);

	// Get the points.
	CVector vInboardJointTail = inboardJoint->GetTail();

	// Update for all outboard joints.
	int size = vOutboardJoints.size();
	for (int i = 0; i < size; ++i)
	{
		CVector vOutboardJointHead = vOutboardJoints[i]->GetHead();

		// Draw the spring.
		glBegin(GL_LINES);
			glVertex3f(vInboardJointTail.x, vInboardJointTail.y,
vInboardJointTail.z);
			glVertex3f(vOutboardJointHead.x, vOutboardJointHead.y,
vOutboardJointHead.z);
		glEnd();
	}

	glEnable(GL_TEXTURE_2D);
	glEnable(GL_LIGHTING);
}

void CSpringConstraint::SetSpringConstant(int nSpringConstant_)
{
	const float SPRINGMIN = 0.0002f;
```

```
        const float SPRINGMAX = 0.007f;

        float percentage = (static_cast<float>(nSpringConstant_)) / 100.0f;
        fSpringConstant = SPRINGMIN + percentage * (SPRINGMAX - SPRINGMIN);
}
```

Appendix B: Code Printouts for Verlet Integration Demo

```
/*************************************************************
File name: IDynamic.h

Author(s): Jean-Christian Delannoy

Description: Interface for a dynamic (movable) object

documentation:

version: 0.001

*************************************************************
*************************************************************
TODO:


*************************************************************
CHANGES:

JCD 03/10/2005 Created


*************************************************************
*************************************************************
Notes:



*************************************************************
*************************************************************
*************************************************************
*************************************************************/

#include "stdafx.h"
#include "IDynamic.h"
```

```
IDynamic::IDynamic() :
      fMass(5.0f),
      vTotalForce(0.0f, 0.0f, 0.0f),
      fLinearFriction(0.025f)
{
}

void IDynamic::ResetForces()
{
      vTotalForce.SetXYZ(0.0f, 0.0f, 0.0f);
}

void IDynamic::ApplyLinearForce(const CVector& vForce)
{
      vTotalForce += vForce;
}

void IDynamic::RenderTotalForce()
{
      // Render the tail.
      glDisable(GL_LIGHTING);
      glDisable(GL_TEXTURE_2D);
      glColor3f(0.7f, 0.7f, 0.7f);

      // Get the points.
      CVector vForcePoint = vPosition + (vTotalForce * 50000.0f);

      // Draw the force.
      glBegin(GL_LINES);
            glVertex3f(vPosition.x, vPosition.y, vPosition.z);
            glVertex3f(vForcePoint.x, vForcePoint.y, vForcePoint.z);
      glEnd();

      glEnable(GL_TEXTURE_2D);
      glEnable(GL_LIGHTING);
}
```

```
/************************************************************************
File name: CPhysicsEngine.cpp

Author(s): Jean-Christian Delannoy

Description: The physics manager.

documentation:

version: 001

************************************************************************
************************************************************************
TODO:


************************************************************************
CHANGES:
      JCD 03/10/2005 CREATED


************************************************************************
************************************************************************
Notes:



************************************************************************
************************************************************************
************************************************************************
************************************************************************/

#include "stdafx.h"
#include "CPhysicsEngine.h"

CPhysicsEngine::CPhysicsEngine()
{
      fLastTime = -1;
      fConstElapsedTime = -1.0f;
}

CPhysicsEngine::~CPhysicsEngine()
{
}

void CPhysicsEngine::ApplyPhysics(int nAlgoType_)
{
      float fCurrentTime = CTimer::Instance()->getTimeElapsed();

      // Get time elapsed.
      if (fLastTime < 0 || (fCurrentTime - fLastTime) > 1000)
      {
            fLastTime = CTimer::Instance()->getTimeElapsed();
            return;
      }

      float fElapsedTime;
```

```
	if (fConstElapsedTime > 0)
		fElapsedTime = fConstElapsedTime;
	else
		fElapsedTime = fCurrentTime - fLastTime;

	// Set the previous time.
	fLastTime = fCurrentTime;

	// Apply gravity and friction forces here.
	int numobjects = vCylinders.size();
	const CVector vGravity(0.0f, -0.00098f, 0.0f);
	CVector vTempGravity;
	for (int i = 0; i < numobjects; ++i)
	{
		vTempGravity = vGravity * vCylinders[i]->GetMass();
		vCylinders[i]->ApplyLinearForce(vTempGravity);
	}

	// Move dynamic objects (account for forces and move objects)
	for (int i = 0; i < numobjects; ++i)
	{
		vCylinders[i]->Update(fElapsedTime);
	}

	// Resolve all the constraints in the system via relaxation.
	ResolveConstraints(nAlgoType_);
}

void CPhysicsEngine::ResolveConstraints(int nAlgoType_)
{
	for (int j = 0; j < 1000; ++j)
	{
		// Resolve for the "connector" constraints.
		int numconstraints = vConstraints.size();
		for (int i = 0; i < numconstraints; ++i)
		{
			vConstraints[i]->Simulate(nAlgoType_);
		}

		// Resolve for the internal constraints.
		int numobjects = vCylinders.size();
		for (int i = 0; i < numobjects; ++i)
		{
			vCylinders[i]->Simulate();
		}
	}
}

void CPhysicsEngine::ResetForces()
{
	// Reset all the forces.
	int numobjects = vCylinders.size();
	for (int i = 0; i < numobjects; ++i)
	{
		vCylinders[i]->ResetForces();
	}
}
```

```
void CPhysicsEngine::AddDynamicObject(CCylinder* pCylinder_)
{
	vCylinders.push_back(pCylinder_);
}

void CPhysicsEngine::RemoveDynamicObject(CCylinder* pObject_)
{
	int size = vCylinders.size();
	if (size < 1)
		return;

	// Check the first object.
	std::vector<CCylinder*>::iterator iter = vCylinders.begin();
	if (*iter == pObject_)
		vCylinders.erase(iter);

	// Check the other objects.
	for(int i = 1; i < size; ++i)
	{
		++iter;
		if (*iter == pObject_)
			vCylinders.erase(iter);
	}
}

void CPhysicsEngine::AddConstraint(CRelaxationConstraint* pConstraint_)
{
	vConstraints.push_back(pConstraint_);
}

void CPhysicsEngine::RemoveConstraint(CRelaxationConstraint* pConstraint_)
{
	int size = vConstraints.size();
	if (size < 1)
		return;

	// Check the first object.
	std::vector<CRelaxationConstraint*>::iterator iter =
vConstraints.begin();
	if (*iter == pConstraint_)
		vConstraints.erase(iter);

	// Check the other objects.
	for(int i = 1; i < size; ++i)
	{
		++iter;
		if (*iter == pConstraint_)
			vConstraints.erase(iter);
	}
}
```

```
/*************************************************************
File name: CRelaxationConstraint.cpp

Author(s): Jean-Christian Delannoy

Description: A constraint that is resolved by relaxation.

documentation: none

version: 0.001

*************************************************************
*************************************************************
TODO:


*************************************************************
CHANGES:

JC 04/14/2006 Created.


*************************************************************
*************************************************************
Notes:


*************************************************************
*************************************************************
*************************************************************
*************************************************************/

#include "stdafx.h"
#include "CRelaxationConstraint.h"

CRelaxationConstraint::CRelaxationConstraint(CCylinder* inboardJoint_)
{
	inboardJoint = inboardJoint_;
	fFrictionConstant = 0.1f;
}

CRelaxationConstraint::~CRelaxationConstraint()
{
}

void CRelaxationConstraint::AddOutboardJoint(CCylinder* outboardJoint_)
{
	vOutboardJoints.push_back(outboardJoint_);
}

void CRelaxationConstraint::Simulate(int nAlgoType_)
{
	// The constraint we are trying to enforce here is that the tail of
	// the inboard joint must be at the same position as the head of the
	// outboard joint.
```

```
	int size = vOutboardJoints.size();

	if (inboardJoint->GetImmovable())
	{
		for (int i = 0; i < size; ++i)
		{
			CVector vInboardTail = inboardJoint->GetTailParticle();
			vOutboardJoints[i]->SetHeadParticle(vInboardTail);
		}
	}
	else
	{
		if (nAlgoType_ > 0)
		{
			// New school algorithm.
			for (int i = 0; i < size; ++i)
			{
				float inboardMass = inboardJoint->GetMass();
				CVector vInboardTail = inboardJoint-
>GetTailParticle();
				float outboardMass = vOutboardJoints[i]-
>GetMass();
				CVector vOutboardHead = vOutboardJoints[i]-
>GetHeadParticle();

				CVector vMidPoint = ((vOutboardHead *
outboardMass) + (vInboardTail * inboardMass)) / (inboardMass +
outboardMass);

				inboardJoint->SetTailParticle(vMidPoint);
				vOutboardJoints[i]->SetHeadParticle(vMidPoint);
			}
		}
		else
		{
			// Old school algorithm.
			for (int i = 0; i < size; ++i)
			{
				CVector vInboardTail = inboardJoint-
>GetTailParticle();
				CVector vOutboardHead = vOutboardJoints[i]-
>GetHeadParticle();
				CVector vMidPoint = (vOutboardHead + vInboardTail)
/ 2.0f;

				inboardJoint->SetTailParticle(vMidPoint);
				vOutboardJoints[i]->SetHeadParticle(vMidPoint);
			}
		}
	}
}

void CRelaxationConstraint::Render()
{
	// Render the tail.
	glDisable(GL_LIGHTING);
	glDisable(GL_TEXTURE_2D);
```

```
	glColor3f(0.0f, 1.0f, 0.0f);

	// Get the points.
	CVector vInboardJointTail = inboardJoint->GetTailParticle();

	// Update for all outboard joints.
	int size = vOutboardJoints.size();
	for (int i = 0; i < size; ++i)
	{
		CVector vOutboardJointHead = vOutboardJoints[i]-
>GetHeadParticle();

		// Draw the spring.
		glBegin(GL_LINES);
			glVertex3f(vInboardJointTail.x, vInboardJointTail.y,
vInboardJointTail.z);
			glVertex3f(vOutboardJointHead.x, vOutboardJointHead.y,
vOutboardJointHead.z);
		glEnd();
	}

	glEnable(GL_TEXTURE_2D);
	glEnable(GL_LIGHTING);
}
```

Appendix C: Code Printouts for 3D Articulated Humanoid Demo

```
/************************************************************************
File name: CPhysicsEngine.cpp

Author(s): Jean-Christian Delannoy

Description: The physics manager.

documentation:

version: 001

************************************************************************
************************************************************************
TODO:


************************************************************************
CHANGES:
      JCD 03/10/2005 CREATED


************************************************************************
************************************************************************
Notes:



************************************************************************
************************************************************************
************************************************************************
************************************************************************/

#include "stdafx.h"
#include "CPhysicsEngine.h"
```

```
CPhysicsEngine::CPhysicsEngine()
{
	fLastTime = -1;
	fConstElapsedTime = -1.0f;
}

CPhysicsEngine::~CPhysicsEngine()
{
}

void CPhysicsEngine::ApplyPhysics(int nAlgoType_)
{
	float fCurrentTime = CTimer::Instance()->getTimeElapsed();

	// Get time elapsed.
	if (fLastTime < 0 || (fCurrentTime - fLastTime) > 1000)
	{
		fLastTime = CTimer::Instance()->getTimeElapsed();
		return;
	}

	// Get and set the time.
	float fElapsedTime;
	fElapsedTime = (fConstElapsedTime > 0) ? fConstElapsedTime :
(fCurrentTime - fLastTime);
	fLastTime = fCurrentTime;

	// Apply gravity and friction forces here.
	int numobjects = vCylinders.size();
	const CVector vGravity(0.0f, -0.00098f, 0.0f);
	CVector vTempGravity;
	for (int i = 0; i < numobjects; ++i)
	{
		vTempGravity = vGravity * vCylinders[i]->GetMass();
		vCylinders[i]->ApplyLinearForce(vTempGravity);
	}

	// Update the ball also.
	if (ball)
	{
		if (ball->ApplyGravity)
		{
			vTempGravity = vGravity * ball->GetMass();
			ball->ApplyLinearForce(vTempGravity);
		}
	}

	// Move the cylinders and apply collision detection.
	collisionDetector.Move_ProcessCollisions(vCylinders, ball,
fElapsedTime);

	// Resolve the joint and cylinder constraints (via relaxation).
	ResolveConstraints(nAlgoType_);
}

void CPhysicsEngine::ResolveConstraints(int nAlgoType_)
{
```

```
	for (int j = 0; j < 100; ++j)
	{
		// Resolve for the "connector" constraints.
		int numconstraints = vConstraints.size();
		for (int i = 0; i < numconstraints; ++i)
		{
			vConstraints[i]->Simulate(nAlgoType_);
		}

		// Resolve for the internal constraints.
		int numobjects = vCylinders.size();
		for (int i = 0; i < numobjects; ++i)
		{
			vCylinders[i]->Simulate();
		}
	}
}

void CPhysicsEngine::ResetForces()
{
	// Reset all the forces.
	int numobjects = vCylinders.size();
	for (int i = 0; i < numobjects; ++i)
	{
		vCylinders[i]->ResetForces();
	}
	ball->ResetForces();
}

void CPhysicsEngine::AddDynamicObject(CCylinder* pCylinder_)
{
	vCylinders.push_back(pCylinder_);
}

void CPhysicsEngine::RemoveDynamicObject(CCylinder* pObject_)
{
	int size = vCylinders.size();
	if (size < 1)
		return;

	// Check the first object.
	std::vector<CCylinder*>::iterator iter = vCylinders.begin();
	if (*iter == pObject_)
		vCylinders.erase(iter);

	// Check the other objects.
	for(int i = 1; i < size; ++i)
	{
		++iter;
		if (*iter == pObject_)
			vCylinders.erase(iter);
	}
}

void CPhysicsEngine::AddConstraint(CRelaxationConstraint* pConstraint_)
{
	vConstraints.push_back(pConstraint_);
```

```
}

void CPhysicsEngine::AddBall(ICollideableSphere* pBall_)
{
	ball = pBall_;
}

void CPhysicsEngine::RemoveConstraint(CRelaxationConstraint* pConstraint_)
{
	int size = vConstraints.size();
	if (size < 1)
		return;

	// Check the first object.
	std::vector<CRelaxationConstraint*>::iterator iter =
vConstraints.begin();
	if (*iter == pConstraint_)
		vConstraints.erase(iter);

	// Check the other objects.
	for(int i = 1; i < size; ++i)
	{
		++iter;
		if (*iter == pConstraint_)
			vConstraints.erase(iter);
	}
}
```

```
/*************************************************************
File name: CArticulatedStructure.cpp

Author(s): Jean-Christian Delannoy

Description: A class for encapsulating an articulated structure of
cylinders.

documentation: none

version: 0.001

*************************************************************
*************************************************************
TODO:


*************************************************************
CHANGES:

JC 11/02/2006 Created.


*************************************************************
*************************************************************
Notes:


*************************************************************
*************************************************************
*************************************************************
*************************************************************/

#include "stdafx.h"
#include "CArticulatedStructure.h"

const float VALUEZ = 20.0f;

CArticulatedStructure::CArticulatedStructure()
{
      bApplyImpulse = false;
      vForceToApply.SetXYZ(0.0f, 0.0f, 0.0f);
      nBodyType = 1;

      float currentHeight = 70.0;
      // Add all the cylinders to the vector.
      CCylinder cylinder;

      // The head. (0)
      cylinder.SetHeight(10.0f);
      vCylinders.push_back(cylinder);

      // The neck. (1)
      cylinder.SetHeight(5.0f);
      vCylinders.push_back(cylinder);
```

```
        // The upper torso. (2)
        cylinder.SetHeight(28.0f);
        vCylinders.push_back(cylinder);

        // The spine. (3)
        cylinder.SetHeight(5.0f);
        vCylinders.push_back(cylinder);

        // The basin. (4)
        cylinder.SetHeight(7.0f);
        vCylinders.push_back(cylinder);

        // The left upper leg. (5)
        float height = 30.0f;
        cylinder.SetHeight(height);
        vCylinders.push_back(cylinder);

        // The right upper leg (6)
        cylinder.SetHeight(height);
        vCylinders.push_back(cylinder);

        // The left lower leg. (7)
        cylinder.SetHeight(height);
        vCylinders.push_back(cylinder);

        // The right lower leg. (8)
        cylinder.SetHeight(height);
        vCylinders.push_back(cylinder);

        // Sets the position and other parameters.
        Reset();

        // The neck is attached to the head.
        CRelaxationConstraint* constraintA = new
CRelaxationConstraint(&vCylinders[0]);
        constraintA->AddOutboardJoint(&vCylinders[1]);
        vConstraints.push_back(constraintA);

        // The upper torso is attached to the neck.
        CRelaxationConstraint* constraintB = new
CRelaxationConstraint(&vCylinders[1]);
        constraintB->AddOutboardJoint(&vCylinders[2]);
        vConstraints.push_back(constraintB);

        // The spine is attached to the upper torso.
        CRelaxationConstraint* constraintC = new
CRelaxationConstraint(&vCylinders[2]);
        constraintC->AddOutboardJoint(&vCylinders[3]);
        vConstraints.push_back(constraintC);

        // The basin is attached to the spine.
        CRelaxationConstraint* constraintD = new
CRelaxationConstraint(&vCylinders[3]);
        constraintD->AddOutboardJoint(&vCylinders[4]);
        vConstraints.push_back(constraintD);

        // The left upper leg is attached to the basin.
```

```
	CRelaxationConstraint* constraintE = new
CRelaxationConstraint(&vCylinders[4]);
	constraintE->AddOutboardJoint(&vCylinders[5]);
	vConstraints.push_back(constraintE);

	// The right upper leg is attached to the basin.
	CRelaxationConstraint* constraintF = new
CRelaxationConstraint(&vCylinders[4]);
	constraintF->AddOutboardJoint(&vCylinders[6]);
	vConstraints.push_back(constraintF);

	// The left lower leg is attached to the left upper leg.
	CRelaxationConstraint* constraintG = new
CRelaxationConstraint(&vCylinders[5]);
	constraintG->AddOutboardJoint(&vCylinders[7]);
	vConstraints.push_back(constraintG);

	// The right lower leg is attached to the right upper leg.
	CRelaxationConstraint* constraintH = new
CRelaxationConstraint(&vCylinders[6]);
	constraintH->AddOutboardJoint(&vCylinders[8]);
	vConstraints.push_back(constraintH);

	//const int NUMCYLINDERS = 5;
	//for (int i = 0; i < NUMCYLINDERS; ++i)
	//{
	//	cylinder.SetHeight(20.0f);
	//	cylinder.SetRadius(1.5f);
	//	cylinder.SetMass(5.0f);
	//	cylinder.SetPosition(CVector(0.0f, currentHeight, VALUEZ));
	//	currentHeight -= 20.0f;
	//	vCylinders.push_back(cylinder);
	//}

	// Configure the constraints.
	//for (int i = 0; i < NUMCYLINDERS - 1; ++i)
	//{
	//	CRelaxationConstraint* constraint = new
CRelaxationConstraint(&vCylinders[i]);
	//	constraint->AddOutboardJoint(&vCylinders[i + 1]);
	//	vConstraints.push_back(constraint);
	//}



	// Add each cylinder to the physics engine.
	int numCylinders = vCylinders.size();
	for (int i = 0; i < numCylinders; ++i)
	{
		vCylinders[i].SetImmovable(true);
		physicsEngine.AddDynamicObject(&vCylinders[i]);
	}

	// Add the constraint to the physics engine.
	int numConstraints = vConstraints.size();
	for (int i = 0; i < numConstraints; ++i)
	{
```

```
            physicsEngine.AddConstraint(vConstraints[i]);
      }

      // Initialize the ball.
      ball.SetMass(5.0f);
      ball.SetRayon(5.0f);
      CVector vStartingPos(80.0f, 25.0f, 20.0f);
      ball.SetPosition(vStartingPos);
      ball.SetPreviousPosition(vStartingPos);
      physicsEngine.AddBall(&ball);

      // Default values for the keys.
      keyRight = keyLeft = keyDown = keyUp = keyPaused = keyShift =
keyEnter = false;
      focusCylinder = 1;
}

CArticulatedStructure::~CArticulatedStructure()
{
      int numConstraints = vConstraints.size();
      for (int i = 0; i < numConstraints; ++i)
      {
            delete vConstraints[i];
      }
}

void CArticulatedStructure::Update(int nAlgoType_)
{
      if (!keyPaused)
      {
            // Reset all the objects.
            physicsEngine.ResetForces();

            // Apply any user controlled forces.
            const float forceScalar = 0.025f;
            if (focusCylinder < vCylinders.size() && focusCylinder >= 0)
            {
                  if (keyRight)
                  {

      vCylinders[focusCylinder].ApplyLinearForce(CVector(forceScalar,
0.0f, 0.0f));
                        //notice<<"Applying RIGHT force."<<"endl";
                  }
                  if (keyLeft)
                  {

      vCylinders[focusCylinder].ApplyLinearForce(CVector(-forceScalar,
0.0f, .0f));
                        //notice<<"Applying LEFT force."<<"endl";
                  }
                  if (keyUp)
                  {

      vCylinders[focusCylinder].ApplyLinearForce(CVector(0.0f, 0.0f, -
forceScalar));
                        //notice<<"Applying UP force."<<"endl";
```

```
                }
                if (keyDown)
                {

      vCylinders[focusCylinder].ApplyLinearForce(CVector(0.0f, 0.0f,
forceScalar));
                      //notice<<"Applying DOWN force."<<"endl";
                }
                if (keyShift)
                {
                      ball.ApplyLinearForce(CVector(-forceScalar * 5.0f,
0.0f, 0.0f));
                }
                if (bApplyImpulse)
                {
                      ball.ApplyLinearForce(vForceToApply);
                      bApplyImpulse = false;
                }
                if (keyEnter)
                {
                      Reset();
                }
          }

          // Reset all objects and apply physics to all elements in the
scene.
          physicsEngine.ApplyPhysics(nAlgoType_);
      }
}

void CArticulatedStructure::KeyInput(UINT nChar, bool bKeyDown_)
{
      if (nChar == VK_RIGHT)
            keyRight = bKeyDown_;

      if (nChar == VK_LEFT)
            keyLeft = bKeyDown_;

      if (nChar == VK_UP)
            keyUp = bKeyDown_;

      if (nChar == VK_DOWN)
            keyDown = bKeyDown_;

      if (nChar == VK_SHIFT)
            keyShift = bKeyDown_;

      if (nChar == (VK_RETURN))
            keyEnter = bKeyDown_;

      if (nChar == VK_SPACE && bKeyDown_)
            keyPaused = !keyPaused;

      if (nChar == VK_F1 && bKeyDown_)
            focusCylinder = 1;
      if (nChar == VK_F2 && bKeyDown_)
            focusCylinder = 2;
```

```
        if (nChar == VK_F3 && bKeyDown_)
                focusCylinder = 3;
        if (nChar == VK_F4 && bKeyDown_)
                focusCylinder = 4;
        if (nChar == VK_F5 && bKeyDown_)
                focusCylinder = 5;
        if (nChar == VK_F6 && bKeyDown_)
                focusCylinder = 6;
        if (nChar == VK_F7 && bKeyDown_)
                focusCylinder = 7;

        // Make the top cylinder no longer immovable.
        if (nChar == VK_F8 && bKeyDown_)
        {
                int numCylinders = vCylinders.size();
                for (int i = 0; i < numCylinders; ++i)
                        vCylinders[i].SetImmovable(false);
        }

        // Turn on/off the debug spheres.
        if (nChar == VK_F9 && bKeyDown_)
        {
                int numCylinders = vCylinders.size();
                for (int i = 0; i < numCylinders; ++i)
                        vCylinders[i].ToggleDebugSpheres();
        }
}

void CArticulatedStructure::Render(float floorHeight)
{
        int numCylinders = vCylinders.size();
        for (int i = 0; i < numCylinders; ++i)
        {
                vCylinders[i].Render(floorHeight, (focusCylinder == i));
        }

        // Draw the ball also
        ball.DrawCollisionSphere();
        ball.DrawShadow(floorHeight);

        // render the right force (if applied).
        if (keyRight)
        {
                CVector vEnd = (vCylinders[focusCylinder].GetHeadParticle() +
vCylinders[focusCylinder].GetTailParticle()) / 2.0f;
                CVector vSource = vEnd + CVector(-15.0f, 0.0f, 0.0f);
                RenderArrow(vSource, vEnd);
        }

        // render the left force (if applied).
        if (keyLeft)
        {
                CVector vEnd = (vCylinders[focusCylinder].GetHeadParticle() +
vCylinders[focusCylinder].GetTailParticle()) / 2.0f;
                CVector vSource = vEnd + CVector(15.0f, 0.0f, 0.0f);
                RenderArrow(vSource, vEnd);
        }
```

```
	// render the up force (if applied).
	if (keyUp)
	{
		CVector vEnd = (vCylinders[focusCylinder].GetHeadParticle() +
vCylinders[focusCylinder].GetTailParticle()) / 2.0f;
		CVector vSource = vEnd + CVector(0.0f, 0.0f, 15.0f);
		RenderArrow(vSource, vEnd);
	}

		// render the down force (if applied).
	if (keyDown)
	{
		CVector vEnd = (vCylinders[focusCylinder].GetHeadParticle() +
vCylinders[focusCylinder].GetTailParticle()) / 2.0f;
		CVector vSource = vEnd + CVector(0.0f, 0.0f, -15.0f);
		RenderArrow(vSource, vEnd);
	}

	//for (int i = 1; i < numCylinders; ++i)
	//{
	//	vCylinders[i].RenderTotalForce();
	//}

	// No need to render the constraints because they will always be of
length 0.
	//int numConstraints = vConstraints.size();
	//for (int i = 0; i < numConstraints; ++i)
	//{
	//	vConstraints[i]->Render();
	//}
}

void CArticulatedStructure::RenderArrow(CVector vSource, CVector vEnd)
{
	// Disable lighting and texturing.
	glDisable(GL_LIGHTING);
	glDisable(GL_TEXTURE_2D);
	glColor3f(0.5f, 0.5f, 1.0f);
	glLineWidth(13.0f);

	// Draw the arrow line.
	glBegin(GL_LINES);
		glVertex3f(vSource.x, vSource.y, vSource.z);
		glVertex3f(vEnd.x, vEnd.y, vEnd.z);
	glEnd();

	// Draw the arrow head.
	CVector vDifference = vEnd - vSource;
	CVector vArrowHeadPointA = vEnd - (vDifference / 3.0f);
	CVector vArrowHeadPointB = vArrowHeadPointA + CVector (0.0f, 4.0f,
0.0f);

	glBegin(GL_LINES);
		glVertex3f(vEnd.x, vEnd.y, vEnd.z);
		glVertex3f(vArrowHeadPointB.x, vArrowHeadPointB.y,
vArrowHeadPointB.z);
```

```
        glEnd();

        // Invert the second arrow head and draw it.
        vArrowHeadPointB = vArrowHeadPointA - CVector (0.0f, 4.0f, 0.0f);
        glBegin(GL_LINES);
              glVertex3f(vEnd.x, vEnd.y, vEnd.z);
              glVertex3f(vArrowHeadPointB.x, vArrowHeadPointB.y,
vArrowHeadPointB.z);
        glEnd();

        glLineWidth(1.0f);

        // Enable lighting and texturing once again.
        glEnable(GL_TEXTURE_2D);
        glEnable(GL_LIGHTING);
}

// Set up an impulse to be sent later.
void CArticulatedStructure::LaunchBall(CVector vForce)
{
        vForceToApply = vForce;
        bApplyImpulse = true;
        ball.ApplyGravity = true;
}

// Reset the position of the ball and the articulated body.
void CArticulatedStructure::Reset()
{
        // Reset the ball.
        CVector vStartingPos(80.0f, 25.0f, 20.0f);
        ball.SetPosition(vStartingPos);
        ball.SetPreviousPosition(vStartingPos);
        ball.ApplyGravity = false;

        // Setup mass and parameters.
        if (nBodyType == 0)
        {
              // Thin.
              vCylinders[0].SetRadius(4.0f);      // The head. (0)
              vCylinders[1].SetRadius(2.0f);      // The neck. (1)
              vCylinders[2].SetRadius(6.0f);      // The upper torso. (2)
              vCylinders[3].SetRadius(2.0f);      // The spine. (3)
              vCylinders[4].SetRadius(6.0f);      // The basin. (4)
              vCylinders[5].SetRadius(2.0f);      // The left upper leg. (5)
              vCylinders[6].SetRadius(2.0f);      // The right upper leg (6)
              vCylinders[7].SetRadius(2.0f);      // The left lower leg. (7)
              vCylinders[8].SetRadius(2.0f);      // The right lower leg.
(8)

              vCylinders[0].SetMass(20.0f);
              vCylinders[1].SetMass(5.0f);
              vCylinders[2].SetMass(84.0f);
              vCylinders[3].SetMass(5.0f);
              vCylinders[4].SetMass(21.0f);
              vCylinders[5].SetMass(30.0f);
              vCylinders[6].SetMass(30.0f);
              vCylinders[7].SetMass(30.0f);
```

```
            vCylinders[8].SetMass(30.0f);
      }
      else if (nBodyType == 1)
      {
            // Medium.
            vCylinders[0].SetRadius(5.0f);      // The head. (0)
            vCylinders[1].SetRadius(2.0f);      // The neck. (1)
            vCylinders[2].SetRadius(10.0f);     // The upper torso. (2)
            vCylinders[3].SetRadius(2.0f);      // The spine. (3)
            vCylinders[4].SetRadius(10.0f);     // The basin. (4)
            vCylinders[5].SetRadius(3.0f);      // The left upper leg. (5)
            vCylinders[6].SetRadius(3.0f);      // The right upper leg (6)
            vCylinders[7].SetRadius(3.0f);      // The left lower leg. (7)
            vCylinders[8].SetRadius(3.0f);      // The right lower leg.
(8)

            vCylinders[0].SetMass(25.0f);
            vCylinders[1].SetMass(5.0f);
            vCylinders[2].SetMass(140.0f);
            vCylinders[3].SetMass(5.0f);
            vCylinders[4].SetMass(35.0f);
            vCylinders[5].SetMass(45.0f);
            vCylinders[6].SetMass(45.0f);
            vCylinders[7].SetMass(45.0f);
            vCylinders[8].SetMass(45.0f);
      }
      else
      {
            // Wide.
            vCylinders[0].SetRadius(8.0f);      // The head. (0)
            vCylinders[1].SetRadius(2.0f);      // The neck. (1)
            vCylinders[2].SetRadius(25.0f);     // The upper torso. (2)
            vCylinders[3].SetRadius(2.0f);      // The spine. (3)
            vCylinders[4].SetRadius(25.0f);     // The basin. (4)
            vCylinders[5].SetRadius(6.0f);      // The left upper leg. (5)
            vCylinders[6].SetRadius(6.0f);      // The right upper leg (6)
            vCylinders[7].SetRadius(6.0f);      // The left lower leg. (7)
            vCylinders[8].SetRadius(6.0f);      // The right lower leg.
(8)

            vCylinders[0].SetMass(40.0f);
            vCylinders[1].SetMass(5.0f);
            vCylinders[2].SetMass(350.0f);
            vCylinders[3].SetMass(25.0f);
            vCylinders[4].SetMass(87.5f);
            vCylinders[5].SetMass(90.0f);
            vCylinders[6].SetMass(90.0f);
            vCylinders[7].SetMass(90.0f);
            vCylinders[8].SetMass(90.0f);
      }

      // Reset the head.
      vCylinders[0].SetPosition(CVector(0.0f, 70.0f, VALUEZ));

      // The neck. (1)
      vCylinders[1].SetPosition(CVector(0.0f, 62.50f, VALUEZ));
```

```
        // The upper torso. (2)
        vCylinders[2].SetPosition(CVector(0.0f, 45.0f, VALUEZ));

        // The spine. (3)
        vCylinders[3].SetPosition(CVector(0.0f, 25.0f, VALUEZ));

        // The basin. (4)
        vCylinders[4].SetPosition(CVector(0.0f, 15.0f, VALUEZ));

        // The left upper leg. (5)
        CVector vHead(0.0f, 20.0f, VALUEZ);
        CVector vLimb(0.0f, -vCylinders[5].GetHeight(), 0.0f);
        vLimb.Rotate(30.0f, CVector(1.0f, 0.0f, 0.0f));
        CVector vTail = vHead + vLimb;
        vCylinders[5].SetTailParticle(vTail);
        vCylinders[5].SetHeadParticle(vHead);

        // The right upper leg (6)
        vHead.SetXYZ(0.0f, 20.0f, VALUEZ);
        vLimb.SetXYZ(0.0f, -vCylinders[6].GetHeight(), 0.0f);
        vLimb.Rotate(-30.0f, CVector(1.0f, 0.0f, 0.0f));
        vTail = vHead + vLimb;
        vCylinders[6].SetTailParticle(vTail);
        vCylinders[6].SetHeadParticle(vHead);

        // The left lower leg. (7)
        vCylinders[7].SetPosition(CVector(0.0f, 5.0f, VALUEZ - 16.0f));

        // The right lower leg. (8)
        vCylinders[8].SetPosition(CVector(0.0f, 5.0f, VALUEZ + 16.0f));

        // Make each cylinder immovable.
        int numCylinders = vCylinders.size();
        for (int i = 0; i < numCylinders; ++i)
        {
                vCylinders[i].SetImmovable(true);
                vCylinders[i].SetTailPrevPosition();
                vCylinders[i].SetHeadPrevPosition();
        }
}

void CArticulatedStructure::SetBallRadius(float fRadius_)
{
        ball.SetRayon(fRadius_);
}

void CArticulatedStructure::SetBallElasticity(float fElasticity)
{
        ball.SetRestitutionCoef(fElasticity);
}

void CArticulatedStructure::SetSpringConstant(int nSpringConstant_)
{
        // Not implemented for relaxation constraints.
}
```

References

[Bar89] D. Baraff. Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies. *Proceedings of SIGGRAPH '89*, pages 223-231, July 1989.

[Bar96] D. Baraff. Linear-Time Dynamics using Lagrange Multipliers. *Proceedings of SIGGRAPH '96*, pages 137-146, August 1996.

[Bou02] D.M. Bourg. *Physics for Game Developers*. O'Reilly & Associates, Sebastopol, California, January 2002.

[BW92] D. Baraff, A. Witkin. Dynamic Simulation of Non-penetrating Flexible Bodies. *Proceedings of SIGGRAPH '92*, pages 303-308, July 1992.

[BW97] D. Baraff, A. Witkin. *Physically Based Modeling: Principles and Practice*. Siggraph 1997 course notes, 1997.

[BW98] D. Baraff, A. Witkin. Large Steps in Cloth Simulation. *Siggraph 98 Proceedings*, pages 43-54, July 1998.

[Cai00] L.W. Cai. *A Crash Course on Lagrangian Dynamics*. Department of Mechanical and Nuclear Engineering, Kansas State University.

[Del02] J.C. Delannoy. *Physics Modeling for Virtual Interactive Environments*. DISCOVER Laboratory, University of Ottawa, December 2002.

[DPW03] J.C. Delannoy, E. M. Petriu, P. Wide. Mechanics Modeling for Virtual Interactive Environments. *Proceedings of HAVE 2003*, pages 55-59. Ottawa, ON, Canada, Sept. 2003.

[Erl05] K. Erleben. *Multibody Dynamics Animation*, Department of Computer Science, Copenhagen University, 2005.

[Fon00] D. Fontijne. *Rigid Body Simulation and Evolution of Virtual Creatures*. University of Amsterdam, Amsterdam, The Netherlands, August 2000.

[GHWS91] H. Grubmuller, H. Heller, A. Windemuth, K. Schulten. Generalized Verlet Algorithm for Efficient Molecular Dynamics Simulations with Long-Range Interactions. *Molecular Simulation*, volume 6, pages 121-142. University of Illinois at Urbana-Champaign, 1991.

[Jak03] T.Jakobsen. *Advanced Character Physics*. Gamasutra, January 2003.

[Kok04] E. Kokkevis, *Practical Physics for Articulated Characters*, Game Developers Conference 2004 Proceedings, 2004.

[LCWV99] W. Liao, A. Choudhary, D. Weiner, P. Varshney. *Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes*. EECS Department, Syracuse University, New York, ECE Department, Northwestern University, Evanston, 1999.

[Len04] E. Lengyel. *Mathematics for 3D Game Programming & Computer Graphics*. Charles River Media, Hingham, Massachusetts, 2004.

[MHG98] W.G. McCallum, D. Hughes-Hallett, A.M. Gleason. *Multivariable Calculus*. John Wiley & Sons Inc., New York, New York, 1998.

[Mir96] B. Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*, doctoral dissertation, University of California at Berkeley, 1996.

[NC99] D.D. Nelso, E. Cohen. Interactive Mechanical Design Variation for Haptics and CAD. *Proceedings of Eurographics '99*, 1999.

[PAK02] D.K. Pai, U.M. Ascher, P.G. Kry. *Forward Dynamics Algorithms for Multibody Chains and Contact*. Department of Computer Science, University of British Columbia, Vancouver, Canada, 2002.

[PTVF02] W. Press, S. Teukolsky, W. Vetterling, B. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 2002.

[Rho00] G.S. Rhodes. *Stable Rigid-body Physics*. Applied Research Associates Inc., Raleigh, North Carolina, United States of America.

[Sha05] A. A. Shabana. *Dynamics of Multibody Systems*, Cambridge University Press, New York, NewYork, 2005.

[ST00] D. Stewart, J.C. Trinkle. *An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Coulomb Friction*. Department of Mathematics, University of Iowa, Iowa City, Iowa.

[TAP97] J.C. Tannehill, D.A. Anderson, R.H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Taylor & Francis, Philadelphia, PA, 1997.

[Van04] G. Van Den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufman Publishers, San Francisco, California, 2004.

[Wei05] E. Weisstein. *Eric Weisstein's World of Physics: Moment of Inertia*, Wolfram Research, scienceworld.wolfram.com, 2005.

[Wel93] C. Welman. *Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation*. Simon Fraser University, Burnaby, British Columbia, 1993.

[Wu92] X. Wu. A linear-time simple bounding volume algorithm. In *Graphics Gems III*, pages 301-306. Academic Press, Boston, 1992.

[JF03] D. L. James, K. Fathalian. Precomputing Interactive Dynamic Deformable Scenes. *Robotics Institute*, Carnegie Mellon University, September 2003.