NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A REAL-TIME ROPE MODEL SUITABLE FOR GAME ENGINE USAGE

by

Randy A. Garrido

September 2004

Thesis Advisor:	Michael J. Zyda
Second Reader:	Joseph A. Sullivan

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2004	Master's Thesis

4. TITLE AND SUBTITLE: A Rope Model	**5. FUNDING NUMBERS**
6. AUTHOR(S) Randy Ando Garrido	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited	

13. ABSTRACT (maximum 200 words)

This thesis attempts to lay a foundation for producing a real-time rope model suitable for game engine usage. The model presented here is only one of the many possible approaches in modeling a rope. The basic premise used was derived from Erkin Tunca's source code. The concept is then attempted on the Open Dynamics Engine (ODE) built by Russell Smith.

This work shows promise but much still needs to be done. This thesis only scratches the surface on the subject. In addition, ODE is primarily designed for (articulated) rigid bodies. Therefore, the next step is to create a deformable body (the rope) in ODE.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Rope Model, Rope, Simulation, Virtual Environment, Mass-Spring Method	75
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A REAL-TIME ROPE MODEL SUITABLE FOR GAME ENGINE USAGE

Randy A. Garrido
Major, United States Army
B.A., University of Guam, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION

from the

NAVAL POSTGRADUATE SCHOOL
September 2004

Author:	Randy A. Garrido
Approved by:	Michael J. Zyda Thesis Advisor
	Joseph A. Sullivan Second Reader
	Rudy Darken Chair, MOVES Academic Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis attempts to lay a foundation for producing a real-time rope model suitable for game engine usage. The model presented here is only one of the many possible approaches in modeling a rope. The basic premise used was derived from Erkin Tunca's source code. The concept is then attempted on the Open Dynamics Engine (ODE) built by Russell Smith.

This work shows promise but much still needs to be done. This thesis only scratches the surface on the subject. In addition, ODE is primarily designed for (articulated) rigid bodies. Therefore, the next step is to create a deformable body (the rope) in ODE.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This thesis would have not been possible if it weren't for Erkin Tunca and Russell Smith. Their works are the basis of this paper. I thank you both, and I hope you are pleased with my effort.

I would also like to thank my Advisor, Prof. Michael Zyda, and my Second Reader, CDR Joseph Sullivan. I have always felt welcome in your offices for questions, comments, or inquiries; at the same time, you gave me room for creativity. I learned more than I expected in writing this paper. Thank you.

To all my instructors at NPS, you all made an impact not only in my education, but also in my professional and personal life. I may not remember everything I have learned, but I will always have my toolbox with me wherever I go. Thanks to all of you.

Most of all, I would like to extend my gratitude to my wife, Jennifer. I thank you for your love, support, and understanding. And to my daughter, Abigail, whom I felt bad for every time I wasn't able to play with her. Maybe now I'll have more time to play with you. I love you both.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THE ROPE MODEL

> …the important goals are believability (the programmer can cheat as much as he wants if the player still feels immersed) and speed of execution (only a certain time per frame will be allocated to the physics engine). In the case of physics simulation, the word believability also covers stability; a method is no good if objects seem to drift through obstacles or vibrate when they should be lying still, or if cloth particles tend to blow up. [1]

Modeling or simulating a rope in a virtual environment in real time requires it not only be believable, it must also have adequate speed for proper execution. This challenge is not limited to the physics behind the behavior of a rope; it is also in its rendering at runtime. Naturally, the computer's processing speed impacts this dilemma greatly. The faster the computer's processing speed the bigger an algorithm it can handle, the faster it can be executed, and the more realistic the rendering will be.

Current available stand-alone computer systems have more memory and faster computing speed. Despite this dramatic improvement, unfortunately, it is still practically impossible to create a "perfect" simulation of a rope and have it rendered in real time. It still requires a delicate balance between believability and computational speed.

The rope is an example of a deformable body. Deformable bodies differ from rigid bodies in that deformable bodies can change shape and size while rigid bodies cannot. The rope is a one-dimensional deformable body. [2]

There are two known techniques used in simulating these types of objects; the Finite Element Method (FEM) and the Mass-Spring System. FEM has its foundation in numerical methods. FEM is very consistent and powerful, but it is also computationally intensive. [3] This method, despite its advantage in being

able to render a more accurate depiction of a rope, will obviously not meet the requisite for real time rendering because of its computational need.

The second technique, the mass-spring method, is less intensive than the FEM method. However, this method can still be intensive in the sense that it has to be continually computed. In addition, the larger the quantity of particles (or masses) involved, the harder the computer processor will have to work. This is a limitation that will be discussed further in the last chapter.

B. MOTIVATION

There are physics engines, such as one developed by Havok, that are able to strike the balance between believability and computational speed. The author(s) of these codes managed to simulate a rope that may lack accuracy, but are believable enough to the users that the lack of fidelity is easily overlooked.

Since Havok is a commercially available software, the end-user has to pay a hefty sum in order to get the license to use their product. Unfortunately, this is financially prohibitive. The ultimate goal of this paper is to create open source codes on the subject for academic purposes.

The immediate goals of this thesis are: (1) to lay the foundation for development of an open source code for a rope model stable enough to be utilized in a simulated environment at runtime, (2) to identify issues and challenges in applying the concept to ODE, and (3) to make recommendations and approaches for future work on the subject. Given the scope of the subject being covered, the main focus of this project is to primarily produce codes that can be used for demonstration and that meet the objectives per aforementioned. These codes can then be dissected and scrutinized for better understanding and future study.

The source codes written for this thesis are not presented as the only ways to simulate a rope model, but are the most logical approaches found given the available resources. They are only a few of the many and the approaches taken are far from perfect. Issues were encountered in the process. Some were

resolved, others were not. Some that were resolved created new issues, and others were simply noted for future work.

C. CHALLENGES WITH MODELING A ROPE

Regardless of the technique used, the difficulty in modeling a rope is primarily with its requirement for intense and continuous computation. In Erkin Tunca's code, for instance, a rope consisting of 50 particles would have 49 springs that attach the particles to one another (see Figure 1). Each particle must then be computed for all the factors that affect it, like the constant downward pull of gravity and the push or pull force of the spring.

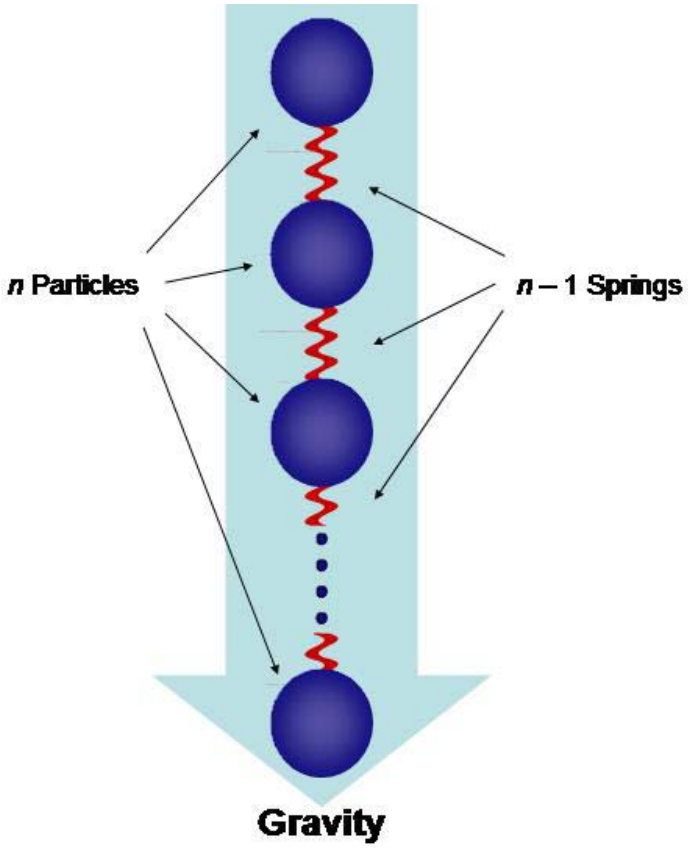

Figure 1. Mass–Spring Connections

If one end of the rope is fixed in mid-air, the rest of the rope will hang beneath it. Even though it is not moving, there is a constant gravitational pull downward for all particles in the rope. The springs, on the other hand, pull on the particles together in order to maintain their distances.

This constant pushing and pulling between gravity and springs alone requires constant computation and adjustment. Now, multiply that 50 times, and one can understand how intensive it can be, and that's just with the rope alone. One has to remember that interactive simulation programs have other objects in the virtual environment that require computation as well.

To complicate the matter even more, add collision handling for all the particles in the rope. For this example, the first particle in the rope has to be compared with the other 49 particles one at a time to check if they are about to collide. The second particle has to be compared to the other 48, the third to the other 47, and so on. (In addition, we have to be able to collide the rope with other objects in the virtual environment besides the rope.)

That means if the solution is done in a straightforward manner, the algorithm will require $50 \cdot 49 \cdot 48 \cdot \cdots \cdot 1$ times of look-up for potential collision. That's approximately equal to 3.04141e+64. This scenario (again) is only limited to the rope alone.

The number of look-ups grows exponentially as the number of particles gets larger. And if there are any other objects in the simulation, they have to be added to the number of particles as well. When two objects are not about to collide, collision detection exits without doing anything and moves on to other objects. However, the amount of lookup or visit to the function that utilizes collision detection will still be large.

The second half of collision handling is collision response. This is only used when two objects are about to collide. Collision response may not have as numerous a call as collision detection, but can be quite complex.

There are three options in dealing with collision response: the use of kinematic response; "penalty method"; and calculation of an impulse force. These three will be discussed further in the following chapter.

Another area of concern is with the accuracy in computation. Even if double precision is used (for computation), there is a gradual decrease in the accuracy of the computation. Therefore, it has to be approximated, resulting in reduced fidelity.

As previously mentioned, it's a complicated balancing act between believability and computational speed. These demands for computation combined with the desire to generate a realistic-looking rope simulated at runtime definitely present a challenge.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND INFORMATION

A. SOURCE CODES UTILIZED

1. Lesson40.cpp

This source code can be found at http://nehe.gamedev.net. It is the latter of a two-part tutorial written by Erkin Tunca. Lesson 39 is a simple physical simulation engine that defines the motion of a mass (which is referred to as particle in this paper) with the application of force in a three-dimensional environment. [4]

Lesson 40 expanded on Lesson 39's concept and used its concept to create a logical approach in defining a rope's behavior. [5] Tunca added the spring as a constraining force attached to a pair of particles. He cleverly transitioned from Lesson 39 to Lesson 40 and created a simulation of a rope in real time.

Lesson40.cpp provided an algorithm that was fairly easy to comprehend. This source code provided an excellent source of information using the concept of mass-spring method for simulating a rope. Although it included a graphic depiction of a rope, it was done in a limited virtual environment. Drawing a line between the positions of two masses creates the rope. These masses are paired sequentially by springs.

Tunca's rope does not interact with any other object other than the ground. The collision between the rope and the ground is a very simplified version of collision detection and collision response. It is assumed that the focus of this work is primarily to show the rope's behavior.

2. Open Dynamics Engine (ODE)

Russell Smith created the ODE.. Smith and several other contributors are continually improving this physics engine. The current version available is ODE version 0.5 and it can be found at http://ode.org. [6]

The ODE is the other source utilized in this thesis. ODE is a physics engine that provides an excellent library for simulating articulated rigid bodies. This engine is claimed to be fast, flexible, robust, and has built-in collision detection. [7]

ODE is designed for simulating vehicles, legged creatures, and moving objects in the virtual environments. These articulated bodies are created when rigid bodies are connected together with various kinds of joints. The ODE is designed for interactive and simulation use. ODE does this by utilizing an integrator that is stable enough to prevent the simulation from going out of control. More emphasis was placed on speed and stability over physical accuracy in the development of ODE. [8]

Fortunately, ODE has a built-in collision detection system. It has a list of collision primitives (spheres, boxes, etc.) and more are expected to come. ODE uses the concept of "spaces" in order to facilitate speed in identification of potentially intersecting objects. [9]

ODE uses the concept of hard contacts. It's a non-penetrating constraint that prevents two bodies in the virtual environment from overlapping when they collide. Virtual spring is another method that can be used for addressing this issue, but Smith opted to use the concept of hard contact rather than virtual springs. Although used by many simulators, a virtual spring is difficult to accomplish. It is also extremely prone to errors. [10]

B. PHYSICS AND MATH BEHIND THE ROPE MODEL

The concept applied in this thesis consists of the mass-spring method for the first list of examinations that appear in Chapter III. The concept used has been derived from Tunca's code since it provided a simple yet effective rendition of a rope. The physics and math applied to the rope's behavior are also consistent and stable.

On one hand, the lack of several physical factors in Tunca's model, such as rotation, angular momentum, and collision detection (that are in ODE) made it less accurate; on the other hand, it adds to the model's simplicity and made the

code easier to understand. The absence of these features may have reduced the accuracy of the rope's physics, but it had much less impact to its believability of the rope's behavior.

The basic concept behind Tunca's algorithm is also stable enough to demonstrate certain features that are consistent with a rope's basic physical behavior. With some modifications, three versions were developed that show the strength behind this concept. However, there were some modifications that were used to "tweak" the code in order to make the visual rendition of the rope appear realistic.

The second list used for examination is in Chapter IV. The pre-existing function ball-and-socket joint from ODE is primarily used in this chapter. This is the closest thing that resembles a spring in the first list of experiments. There are several types of joint connections provided by ODE. However, the only type of joint that would fulfill the requirement to simulate a rope would be the ball-and-socket type.

1. Particles

An object in a simulated world often has a point within itself that serves as a reference for its position in the virtual environment. This point of reference is often tied to the object'sobjects center of mass, or sometimes center of gravity. The object's constant mass is also important in defining the behavior of the object since it is an integral part that affects the object's velocity, force, and interaction with other objects

The particle used by this model only exists in theory. It does not occupy any space, and it is simply defined as an infinitely small point in a 3-dimensional world. The important aspect about the particle is its movement, and it is defined by this point. The particle's movement is described by the relationship between the quantity of the mass (m), force (f), acceleration (a), velocity (v), and position.

The equation used to derive the particle's movement is the formula $f = ma$. Mass is often given and assumed as a constant value. The derivation shows the equation that is of relevance to simulating the mass in the rope model:

$$f = ma$$

$$f = m\left(\frac{v}{t}\right)$$

$$f = m\left(\frac{d}{t}\right)\left(\frac{1}{t}\right)$$

$$f = m\left(\frac{d}{t^2}\right)$$

$$d = \frac{f \cdot t^2}{m} \qquad (1)$$

Distance d is the distance traveled or the difference between the previous and the current position. This is expressed in terms of $x-$, $y-$, and $z-$axis. The mass' position is continually updated. Looking at the equation, it is also important to note that the value for mass cannot be lower than a certain threshold; otherwise, this will cause instability to the algorithmic computation. This subject will be covered in Chapter V.

The derived equation may at first seem inconsistent with Tunca's code in Figure 15. Actually, the two are the same. The two derivations taken may be different, yet they both arrived at the same end state as shown below:

$$f = m\left(\frac{v}{t}\right)$$

$$ft = mv$$

$$v = \frac{f \cdot t}{m}$$

$$d = \frac{f \cdot t^2}{m} \quad (\text{or } d = v \cdot t) \qquad (2)$$

Another interesting aspect to the derived equation is its reliance on time and the “cumulative” forces applied to the particle. the mass, however, remains constant. Therefore, the distance traveled is proportional to the force (f) applied multiplied to time squared (t^2). Mathematically speaking, t is actually dt since it is a small increment in time (expressed in milliseconds).

2. Springs

Spring is a force used as a constraint in order to maintain a given distance between two particles. The equation for the spring used in this paper starts where force (f) is equal to the negative stiffness $(-k)$ multiplied by the separation distance (x), such as shown below:

$$f = -k \cdot x$$

However, this does not involve maintaining a certain distance between the two particles. An addition of another variable d, the desired distance between two particles, is needed to accomplish this as shown below:

$$f = -k \cdot (x - d) \qquad (3)$$

The spring then applies the resulting force to both particles it is “attached” to. If the current distance between two particles is longer than the desired distance, the forces applied to both particles are equal and toward each other; if the distance is shorter, the forces applied are also equal but are in the opposite direction. And if the current distance is equal to the desired distance, there is no force applied.

3. Collision Handling

Tunca's code, in manner of speaking, did address the issue of collision. However, his approach starts with simply limiting the vertical component of each particle not to go below the height of the ground. Since there are only two objects in the illustrated virtual environment, he simply added forces to mimic bouncing and friction directly on the rope. His approach suffices the need it was intended for, but it cannot be expanded if other objects are added.

ODE, on the other hand, has the functions that address collision handling. Sample codes are also provided for better understanding on how to utilize the collision detection in ODE. "As objects move relative to one another, there are two issues that must be addressed: (1) detecting the occurrence of collision and (2) computing the appropriate response to those collisions." [11]

Collision detection is used to consider the movements of objects relative to one another. A basic approach to collision detection involves testing for collision by determining whether two objects will intersect at a specific point at a specific instance in time. A more sophisticated form tests the movement of one object relative to other objects for overlap during a finite time interval. These methods require computations that can become involved if complex geometries are being considered. [12]

Collision response has three common options: kinematic response, penalty method, and calculation of impulse force. Kinematic response is quick and easy. For particles and spherical shaped objects, this response produces good visual results. The penalty method maintains non-penetration by introducing a temporary, non-physically based force. This method is typically used when the response to collision occurs at a time step when penetration is detected. The strength in using penalty method is in its ease of computation and of incorporating the force into the computational algorithm that is used to simulate rigid body movement. The third option, calculation of impulse force, is a more precise way of inserting force into the system and is typically used when time is backed up to the point of first contact. [13]

In modeling a rope, collision handling is a very complicated matter. The rope is not one object, but a compilation of objects. The particles are joined by springs. Each particle needs to be “wrapped” and centered in a sphere (or another primitive object). The spheres’ diameter is the rope’s diameter.

The purpose of the each sphere is to “occupy” space. It gives the particles volume and provides the geometry for collision handling. Since the spheres are always in close proximity to one another, each particle often collide to the other particles to its immediate left and right (or top and bottom).

The springs sometimes do not have enough force to prevent the spheres from penetrating one another. One immediate solution to this is to do nothing and allow the penetration. If the fault is unnoticeable and the simulation is still believable, then the end result can be acceptable. It need not be perfect, only appear to be.

THIS PAGE INTENTIONALLY LEFT BLANK

III. UNDERSTANDING THE MASS-SPRING METHOD USING TUNCA'S CODE AS AN EXAMPLE

A. TUNCA'S CODE

Tunca's code is basically stripped down to the bare necessities. The virtual environment he created only has two objects in it. Some may find this program simple, but it serves as an excellent source of information if one is only interested in studying how to simulate the behavior of a rope.

B. CLASSES INVOLVED

The main classes that make the simulation work are RopeSimulation, Mass (what is referred to as Particle in this paper), Spring, and Simulation. Vector3D is another class used by this program; however, its primary purpose is to simply contain the location of each mass in terms of the three axes and to facilitate the ease of doing the necessary vector arithmetic computations with the use of overloaded operators and a function.

1. RopeSimulation

For every time step, the sequence of events begins with RopeSimulation. The function operate() contains three steps that generate the movement of each mass in the rope. Whenever this function is called and given a value dot, the first thing that happens is it initializes all the value of the force for all the masses to zero, as shown in Figures 2, 6, and 9.

```
virtual void operate(float dt)
{
    init();
    solve();
    simulate(dt);
}
```

Figure 2. Function operate() in the class RopeSimulation

If the force for each mass is not reset to zero, it will accumulate or the force will continue to exist. This is not a desired effect, since it will cause the mass to move indefinitely. The accumulation of forces that are oriented in the same direction will result in a corresponding increase in velocity.

```
void solve()
{
    for (int a = 0; a < numOfMasses - 1; ++a)
    {
        springs[a]->solve();
    }

    for (a = 0; a < numOfMasses; ++a)
    {
        masses[a]->applyForce(gravitation * masses[a]->m);

        masses[a]->applyForce(-masses[a]->vel * airFrictionConstant);

        if (masses[a]->pos.y < groundHeight)
        {
            Vector3D v;

            v = masses[a]->vel;
            v.y = 0;


            masses[a]->applyForce(-v * groundFrictionConstant);

            v = masses[a]->vel;
            v.x = 0;
            v.z = 0;


            if (v.y < 0)
                masses[a]->applyForce(-v * groundAbsorptionConstant);

            Vector3D force = Vector3D(0, groundRepulsionConstant, 0) *
                (groundHeight - masses[a]->pos.y);

            masses[a]->applyForce(force);
        }

    }


}
```

Figure 3. Function solve() in the class RopeSimulation

Next, it calls on the function solve() as shown in Figure 3 above. This function calls on each spring and for each spring to compute the forces required to apply for the paired masses. See the section on class Spring to understand how the forces are applied to the masses.

In another for-loop, it then applies forces derived from gravity and air friction. If the mass is lower than the ground height, it is given a zero velocity

value for its y-component so that the ground friction applied is only in the x-z plane. This simulates the friction generated between the masses in the rope and the ground.

If the velocity of the mass has a downward direction, the ground absorption is applied to the mass' y-component force. Finally, the difference between the height of the mass and the ground is multiplied to the ground's repulsion constant. The result of this application equates to the mass bouncing off the ground.

The third and last step called is the function simulate() in Figure 4. First, it calls the function simulate() in Simulation class, which then calls the function simulate() of each mass (see Figures 5 & 8). The velocity and position values of each mass are updated.

```
void simulate(float dt)

{
    Simulation::simulate(dt);

    ropeConnectionPos += ropeConnectionVel * dt;

    if (ropeConnectionPos.y < groundHeight)
    {
        ropeConnectionPos.y = groundHeight;
        ropeConnectionVel.y = 0;
    }

    masses[0]->pos = ropeConnectionPos;
    masses[0]->vel = ropeConnectionVel;
}
```

Figure 4. Function simulate() in the class RopeSimulation

Next, the position of the initial mass (ropeConnectionPos) in the rope is updated. It is also checked to ensure that it stays above the ground. The new values for the position and velocity (ropeConnectionVel) of the first mass are saved.

By changing the values of ropeConnectionPos and ropeConnectionVel this way, it bypasses the use of force in order to move the initial mass to the

desire place. Ideally, the particle should be moved with force. Unfortunately, this is not as practical as it may seem.

The first question is how much force is necessary to move the particle from point A to point B. The combined mass of the other particles have to be considered as well since they will drag the initial particle down. Also, the length of each time increment may be controllable, but the number of iteration is not. So, this factor is unknown as well. Having two unknown variables will have one making a "one size fits all" standard approximation for all conditions. This is not good and must be avoided if possible.

2. Simulation

The function simulate() in the class Simulation simply iterates thru the list of masses involved as shown in Figure 5. It calls on the function simulate() of each mass and pass the value dt it received, as shown in Figures 5 and 8. The function init() simply calls on the function init() in each mass.

```
virtual void simulate(float dt)
{
    for (int a = 0; a < numOfMasses; ++a)
        masses[a]->simulate(dt);
}
```

Figure 5. Function simulate() in the class Simulation

```
virtual void init()
{
    for (int a = 0; a < numOfMasses; ++a)
        masses[a]->init();
}
```

Figure 6. Function init() in the class Simulation

3. Spring

The function solve() initially determines the distance between the two masses. It then applies the modified spring equation (discussed in Chapter II) to derive the force needed to maintain the distance between the two connected masses. The frictionConstant is then added to reduce the forces before they are applied to the pair of masses.

```
void solve()
{
    Vector3D springVector = mass1->pos - mass2->pos;

    float r = springVector.length();

    Vector3D force;

    if (r != 0)
        force += (springVector / r) * (r - springLength) * (-springConstant);

    force += -(mass1->vel - mass2->vel) * frictionConstant;


    mass1->applyForce(force);
    mass2->applyForce(-force);
}
```

Figure 7. Function solve() in the class Spring

4. Mass

```
void simulate(float dt)
{
    vel += (force / m) * dt;


    pos += vel * dt;


}
```

Figure 8. Function simulate() in the class Mass

Mass contains the value of the “mass” and its position. In order to move, the mass receives two things: force and time. Given these two, the distance traveled by the mass can be derived. The way the code is written, whenever the function update() is called in the main file, the time is incremented into a constant

value. Therefore, it is the amount of force applied that determines the distance traveled; and if the force is equal to zero, then the mass does not move. The derivation of velocity and position is shown in Figure 8.

Figure 9 below shows how the force of each mass is initialized.

```
void init()
{
    force.x = 0;
    force.y = 0;
    force.z = 0;
}
```

Figure 9. Function init() in the class Mass

5. Lesson40.cpp

The above steps are activated in the main file Lesson40.cpp. When executed, the function update() receives the time that has elapsed since the function was last visited. It receives the time in terms of milliseconds.

There are two key functions that make the simulation of the rope possible in the main file pertaining to the simulation of the rope; the scene is updated, then it is drawn. In the function update(), if the correct key is pressed, the variable ropeConnectionVel is given a value depending on what axis it pertains to. This change is then passed on to the function setRopeConnectionVel() in the class RopeSimulation. It updates the variable ropeConnectionVel in ropeSimulation. This velocity value will cause the first mass in the rope to move. It will then cause a domino-effect to the rest of the rope and it will move accordingly.

The time received by update() is in millisecond; therefore, it is defined as a positive integer. It has to be converted into a fraction of a second, namely dt. It is then divided by a variable maxPossible_dt to figure out how much iteration of 0.002 seconds there are. The integer 1 is added to make sure that there is at least one iteration.

The value of dt is then changed to the result of dividing dt with the number of iterations. This is done to update the value of dt since 1 has been added to the number of iterations. The result is for the latest dt to be always less than 0.002 second. The updated values for the number of iterations and dt are then applied in a for-loop that calls on the function operate() as shown in Figure 10 below:

```
void Update (DWORD milliseconds)
{
    if (g_keys->keyDown [VK_ESCAPE] == TRUE)
        TerminateApplication (g_window);

    if (g_keys->keyDown [VK_F1] == TRUE)
        ToggleFullscreen (g_window);

    Vector3D ropeConnectionVel;

    // Keys Are Used To Move The Rope
    if (g_keys->keyDown [VK_RIGHT] == TRUE)
        ropeConnectionVel.x += 3.0f;

    if (g_keys->keyDown [VK_LEFT] == TRUE)
        ropeConnectionVel.x -= 3.0f;

    if (g_keys->keyDown [VK_UP] == TRUE)
        ropeConnectionVel.z -= 3.0f;

    if (g_keys->keyDown [VK_DOWN] == TRUE)
        ropeConnectionVel.z += 3.0f;

    if (g_keys->keyDown [VK_HOME] == TRUE)
        ropeConnectionVel.y += 3.0f;

    if (g_keys->keyDown [VK_END] == TRUE)
        ropeConnectionVel.y -= 3.0f;

    ropeSimulation->setRopeConnectionVel(ropeConnectionVel);

    float dt = milliseconds / 1000.0f;

    float maxPossible_dt = 0.002f;


    int numOfIterations = (int)(dt / maxPossible_dt) + 1;
    if (numOfIterations != 0)
        dt = dt / numOfIterations;

    for (int a = 0; a < numOfIterations; ++a)
        ropeSimulation->operate(dt);
}
```

Figure 10. Function update() in the main class Lesson40.cpp

Every time the function operate() is called, it goes through the process as previously mentioned. This is done as many times as the number of iterations allows in the for-loop. According to Tunca, 2 milliseconds is the maximum value that can be used for time increment; otherwise, the errors in computation will compound and cause imprecision. This is one way to produce instability to the

program and cause for it to “explode.” Also, by initializing the forces back to zero in init(), the error(s) in computation is inhibited from compounding and from producing instability to the algorithm.

```
void Draw (void)
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();

    // Position Camera 40 Meters Up In Z-Direction.
    // Set The Up Vector In Y-Direction So That +X Directs To Right
    // And +Y Directs To Up On The Window.
    gluLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Draw A Plane To Represent The Ground (Different Colors To Create A Fade)
    glBegin(GL_QUADS);
        glColor3ub(0, 0, 255);
        glVertex3f(20, ropeSimulation->groundHeight, 20);
        glVertex3f(-20, ropeSimulation->groundHeight, 20);
        glColor3ub(0, 0, 0);
        glVertex3f(-20, ropeSimulation->groundHeight, -20);
        glVertex3f(20, ropeSimulation->groundHeight, -20);
    glEnd();

    // Start Drawing Shadow Of The Rope
    glColor3ub(0, 0, 0);
    for (int a = 0; a < ropeSimulation->numOfMasses - 1; ++a)
    {
        Mass* mass1 = ropeSimulation->getMass(a);
        Vector3D* pos1 = &mass1->pos;

        Mass* mass2 = ropeSimulation->getMass(a + 1);
        Vector3D* pos2 = &mass2->pos;

        glLineWidth(2);
        glBegin(GL_LINES);
            glVertex3f(pos1->x, ropeSimulation->groundHeight, pos1->z);
            glVertex3f(pos2->x, ropeSimulation->groundHeight, pos2->z);
        glEnd();
    }
    // Drawing Shadow Ends Here.

    // Start Drawing The Rope.
    glColor3ub(255, 255, 0);
    for (a = 0; a < ropeSimulation->numOfMasses - 1; ++a)
    {
        Mass* mass1 = ropeSimulation->getMass(a);
        Vector3D* pos1 = &mass1->pos;

        Mass* mass2 = ropeSimulation->getMass(a + 1);
        Vector3D* pos2 = &mass2->pos;

        glLineWidth(4);
        glBegin(GL_LINES);
            glVertex3f(pos1->x, pos1->y, pos1->z);
            glVertex3f(pos2->x, pos2->y, pos2->z);
        glEnd();
    }
    // Drawing The Rope Ends Here.

    glFlush ();
}
```

Figure 11. Function draw() in the main class Lesson40.cpp

After the positions of the masses have been updated, the draw() function uses a pointer in order to get the paired masses attached by the springs. It gets these positions and draws lines between them and connects them one pair at a time. The result is a simple simulation of a rope. See Figure 11.

The shadow of the rope is done in the same manner as the rope with one exception. The vertical value of each mass is fixed to the same value as that of the ground. This creates an illusion that there is a light source that cast the rope's shadow on the ground; while in reality, the shadow is simply drawn on the ground.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. MODIFICATIONS TO TUNCA'S CODE

The focus of this chapter is to demonstrate the rope's behavior under three different modifications to Tunca's code: an "improved" version of the rope, a weighted particle sliding down the rope, and a jump rope.

A. DEVELOPING AN IMPROVED ROPE MODEL

The "improvement" made in this version is done by removing the attributes that made the rope look elastic. This modification has been achieved by making several critical alterations to the original code. First, the spring's internal friction has been removed. Second, the time increment used has been reduced from 2 to 1 millisecond. And third, the coefficient, or stiffness, of the spring was increased from 10,000 to 20,000.

This increase in spring's "stiffness" was not possible with the original code. It would have caused instability to the algorithm of the code. This third modification may have been a mere change of value in the variable, but it had a tremendous impact on the elasticity of the rope.

```
void solve()
{
    Vector3D springVector = mass1->pos - mass2->pos;

    float r = springVector.length();

    Vector3D force;

    if (r != 0)
        force += (springVector / r) * (r - springLength) * (-springConstant);

    force += -(mass1->vel - mass2->vel) * frictionConstant;


    mass1->applyForce(force);
    mass2->applyForce(-force);
}
```

Figure 12. Function solve() in the class Spring

There were other changes made, but they were minor and mainly done to enhance the simulation by adding to its believability. These were simple changes to the values of certain variables in order to create a more “realistic” rendition of a simulated rope.

The purpose of internal friction in the class Spring is to reduce the forces being applied to the particles. In effect, it slows the movement of the particles as they adjust to maintain their distances from one another. This is done with the variable named frictionConstant as shown in Figure 12.

Also, what this friction does is allow the springs to stretch or contract longer than necessary, thus it enhances the effect of being springy or rubbery. This effect to the graphic rendering of the rope is quite obvious when the program is running. A snapshot of this effect is shown in figure 13 below:

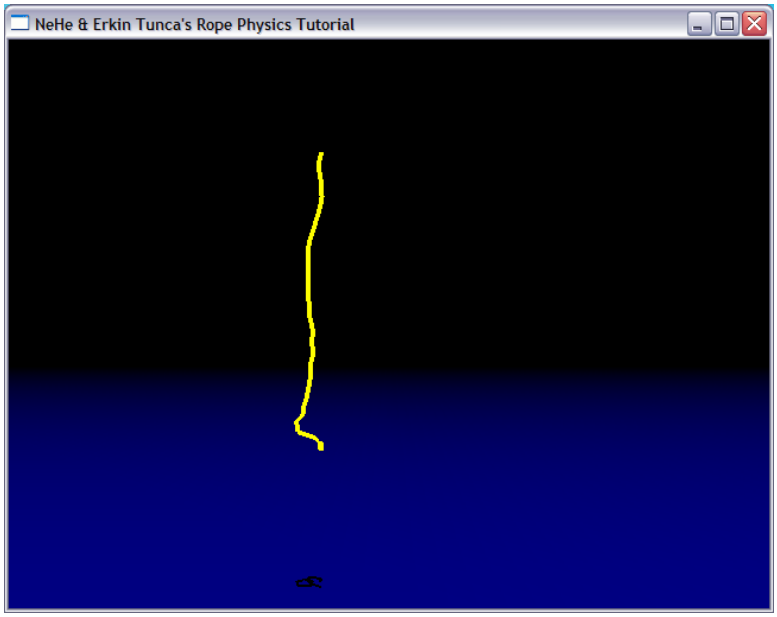

Figure 13. Snapshot of a shortened “NeHe Rope”

The other factor that’s been changed is the time increment used in computing the positions of each particle. Figure 14 shows lower portion of the function update() and how dt is utilized. The modification made simply converts the time received to an integer and use it in the for-loop to iterate the function operate() as shown in Figure 15.

```
    float dt = milliseconds / 1000.0f;

    float maxPossible_dt = 0.002f;


    int numOfIterations = (int)(dt / maxPossible_dt) + 1;
    if (numOfIterations != 0)
        dt = dt / numOfIterations;

    for (int a = 0; a < numOfIterations; ++a)
        ropeSimulation->operate(dt);
}
```

Figure 14. Time increment in update()

```
int   iterations = (int) milliseconds;

for (int a = 0; a < iterations; ++a)
    ropeSim->operate(0.001f);
```

Figure 15. Modification to function update()

One assumption in the modified version is that time marches forward and that the time received by the function update() is always larger than zero. If there ever is a situation where there is a need to go in reverse, this algorithm will bypass the operation of that particular occurrence. It'll skip it since the for-loop requires that the number of iterations be larger than zero.

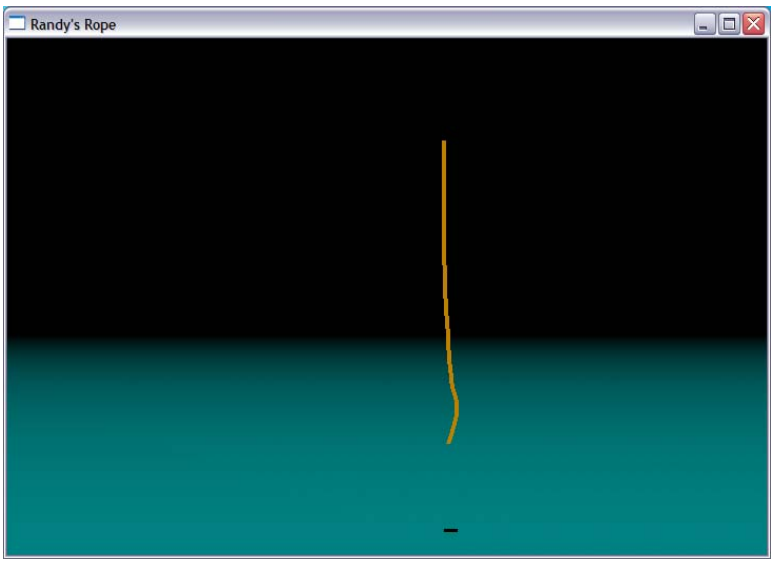

Figure 16. Snapshot of the "Improved Rope"

By removing the internal friction, reducing the time increment, and increasing the “stiffness” coefficient, the simulation of the rope has improved and is a more realistic-looking. It doesn’t stretch like the original version does. Also, after every movement, it doesn’t take the rope as long a time for it to come to a complete stop as shown in Figure 16.

The rationale for the original methods used on the three aforementioned subjects may have something to do with the computer processing capability available when the original code was written. Tunca noted that he used a computer that has a 500Mhz processing speed. The computer used for this paper has a 3.06Ghz processing speed.

The use of maxPossible_dt allows for the least amount of iterations. The lower the iterations the lesser the time it needs to execute the function, the faster the algorithm. maxPossible_dt also allows the largest time increment possible per iteration. Since the forces applied to each mass is done sequentially two at a time, the larger the time increment, the faster the error accumulates, and the more prone the algorithm is to “exploding.”

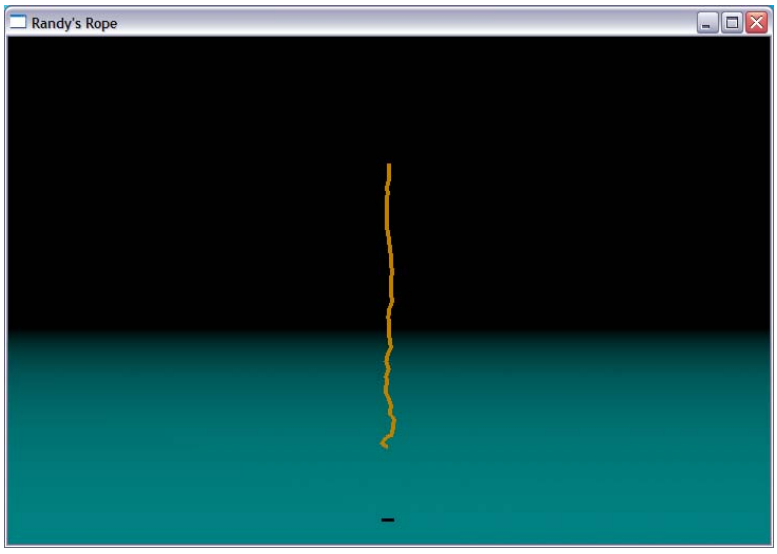

Figure 17. Jarring Effect in the “Improved Rope”

There is, however, one drawback found in the “improved” version. If the rope is yanked, up or down, quick enough, it creates a jarring pattern that is not consistent with a rope’s physical behavior. There’s no bending stiffness in the rope that prevents this from happening. The original version didn’t need an algorithm to create the stiffness since it’s slowed down by elasticity. The “kink” exist with the original version a well, but it is less obvious than the one in the improved version. The kink in the rope is shown in Figure 17.

This condition may have been due to the spring being too stiff. Therefore, forces being applied to the pair of particles have to be reduced. Consequently, returning the spring’s internal friction mitigated this condition. The force being applied to the pair of particles have been slowed enough to reduce the kink. This made the rope a bit springy, but not as noticeable as the previous version. This elasticity is more obvious in the rope that contains the weighted particle.

B. DEVELOPING A SLIDING WEIGHTED PARTICLE ON A ROPE

In theory, if a person is rappelling down a rope, his mass is added to the mass on the part of the rope he’s currently attached to. This added weight affects the movement of the rope. Relative to any given particle along the rope, his momentum will require considerably more force to move in any direction; and because of his inertia, he will also require more force to slow down or stop. His weight will also keep the rope above him taut proportional to his weight, while the part of the rope below him will still be dangling.

Unfortunately, this is not a straight-forward physics translation with the mass-spring method. Since Tunca’s code uses an elastic spring, the addition of a heavier mass only exaggerated the rope’s elasticity. It also prolonged the movement of the rope. There were times when the weighted particle moved in slow motion.

There were two attempts made to investigate the possibility of adding this scenario to the improved rope. The first try was a straightforward approach. The other reduced the vertical component of the weighted particle; in effect, the weight has been reduced.

Inside the class Particle, the function simulate() generates the velocity and position for a given particle. This function is crucial in rendering the graphics for the simulation. The position generated here is what is used for the position of each particle, which are in turn are used for rendering the simulated rope. See Figure 18.

```
void simulate(float dt) {
    velocity += (force / mass) * dt;
    position += velocity * dt;
}
```

Figure 18. Function simulate() in Mass

There are three variables that affect the value of velocity (that ultimately affects the value of the position). However, mass is the variable that is of real relevance in order to simulate a weighted mass on the rope. The first approach introduced to mitigate this condition is accomplished by creating an additional function. The new method applied is direct and simply multiplies the value of the mass 100 times as shown in Figure 19:

```
void simulateHeavyParticle(float dt) {
    velocity += (force / (mass*100)) * dt;
    position += velocity * dt;
}
```

Figure 19. First version of function simulateHeavyParticle() in Mass

How this function is utilized is shown in Figure 20. The function simulate() in RopeSimulation first checks where the weighted mass needs to be applied. The rest of the function is exactly the same with that of the original code.

Unfortunately, using the newly created function has an unwanted side-effect. As shown in Figure 21, the weighted mass has an excessive “bouncing around.” Although the rope does not stretch like a rubber band, the force applied primarily by gravity still causes the rope to be stretched. The force applied by the spring causes it to be pulled back. This tugging between the weight (due to gravity) and the spring causes the unwanted bouncing.

```
void simulate(float dt)
{
    // simulate the movement of each particle in the rope
    for (int a = 0; a < numOfParticles; ++a)
    {
        // this if-else statement applies the correct simulation
        // to the particle that contains the weigted mass

        if (a != downTheRope)
            particles[a]->simulate(dt);
        else
            particles[a]->simulateHeavyParticle(dt);
    }

    ropeConnectionPos += ropeConnectionVel * dt;

    // check to make sure that the first particle doesn't
    // go below the floor

    if (ropeConnectionPos.y < groundHeight)
    {
        ropeConnectionPos.y = groundHeight;
        ropeConnectionVel.y = 0;
    }

    particles[0]->position = ropeConnectionPos;
    particles[0]->velocity = ropeConnectionVel;
}
```

Figure 20. Function simulate() in RopeSimulation

The bouncing produced by this method is greatly influenced by the weighted mass' distance from the initial mass. The farther their distance from one another, the more pronounced the bouncing. However, this bouncing is still obvious even at closer distance to the initial mass.

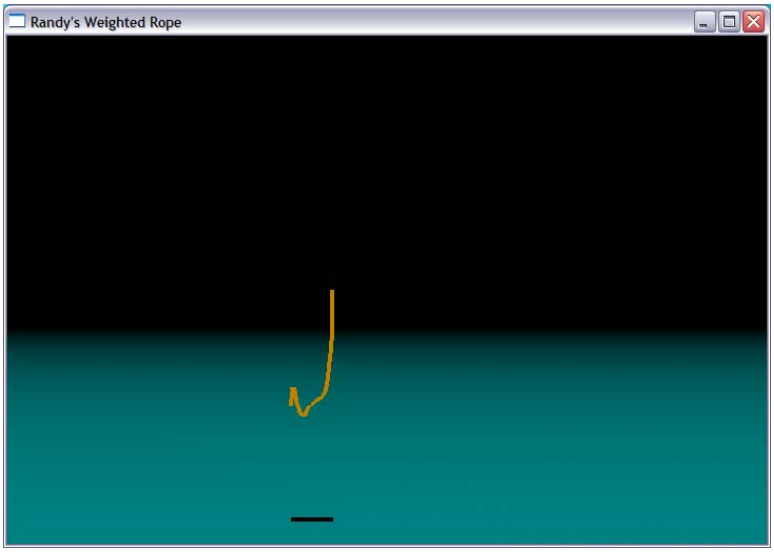

Figure 21. Snapshot of Weighted Mass with an existing y-component

In hindsight, this side effect is consistent with the use of spring in an environment that has gravity. Therefore, to mitigate this condition, one solution is to either reduce the effect of gravity or remove the effect of the spring. Removing the effect of the spring is not possible; it's what keeps the distance between particles.

Therefore, the only option is to reduce the effect of gravity on the weighted mass. The reduced mass only pertain to the y-component of the weighted mass. The value for this component is equal to the regular mass in the rope. This mass cannot be equal to zero since it is a divisor to the force applied. The resulting velocity will go to infinity and cause the simulation to go out of control. This is done as shown in Figure 22.

```
void simulateHeavyParticle(float dt) {

    velocity.x += (force.x / (mass*100)) * dt;
    velocity.y += (force.y / (mass)    ) * dt;
    velocity.z += (force.z / (mass*100)) * dt;


    position += velocity * dt;
}
```

Figure 22. Second version of function simulateHeavyParticle() in Mass

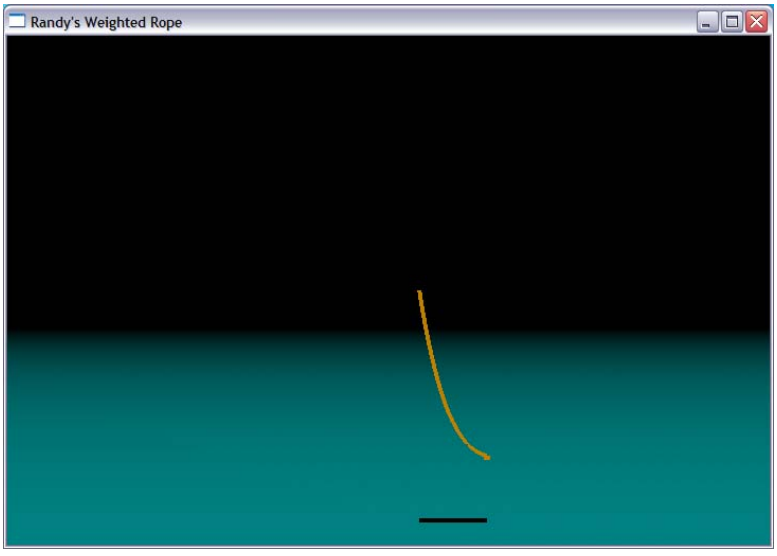

Figure 23. Snapshot of Weighted Mass with y-component reduced

Making the vertical component's value equal to that of the regular mass may have solved the concern about bouncing but it created a new one. The side effect of this new method is not obvious until the weighted mass is near the bottom end of the rope. Then the excessive swaying becomes more pronounced (as shown in Figure 23).

One possible solution sought pertaining to the original question may be to combine the two methods. The idea is to utilize the second method when the rope is moving, and use the first once it stops. The theory is that the excessive swaying will be reduced once the first method is applied.

Unfortunately, this approach is not as easy or as straightforward as it seems. The question of when is the right time to switch from one method to the other is a tricky issue. What is the trigger for the switch? What is the trigger to switch back? These are just some of the questions that need to be addressed.

C. DEVELOPING A JUMP ROPE

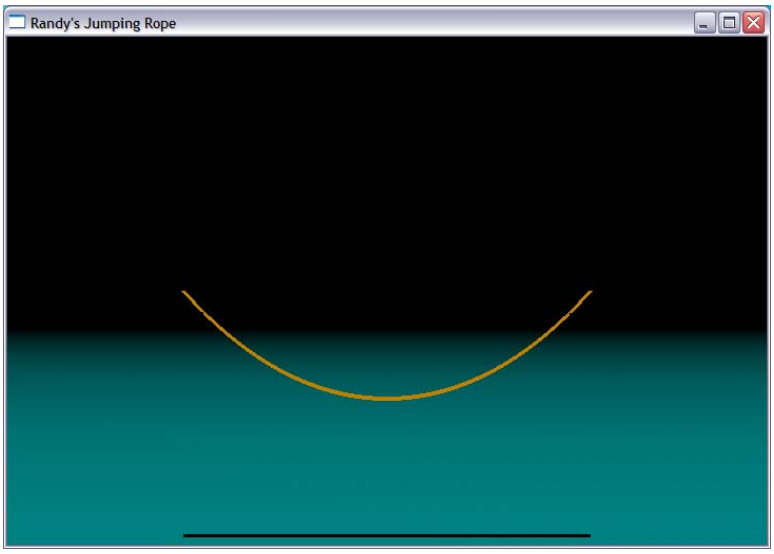

Figure 24. Snapshot of the Jump Rope

The jump rope is actually simpler to develop that it seems. Conceptually, all that is needed is setting the two ends to two fixed points in "mid-air" and have

the two ends rotated perpendicular to the orientation of the rope. In this scenario, the distance between the two points is the original distance when the rope was initialized.

However, applying this concept directly to the improved version of the rope has an undesirable result. Since the springs are too stiff, the curve is minimal. There are two approaches that can mitigate this condition; first, during initialization, plot the positions of the particles along the line created by the equation $y = ax^2 + b$, where a and b influence the horizontal and vertical components of the rope, respectively . See Figure 25 below.

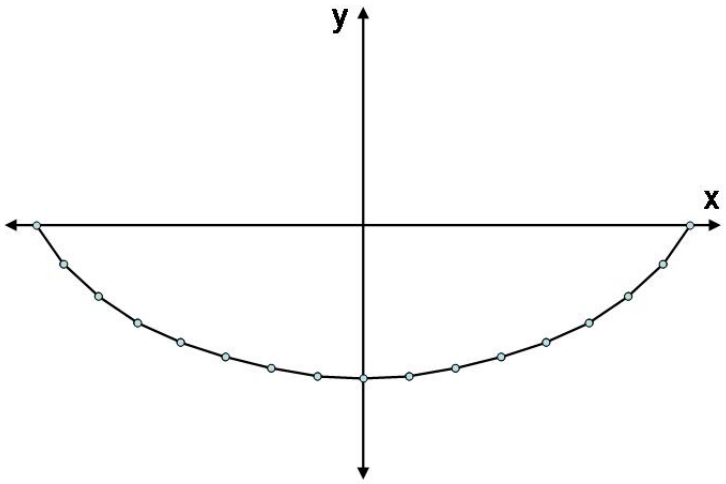

Figure 25. Particles Aligned on Equation $y = ax^2 + b$

The other way is the one applied in this paper. It is created by simply decreasing the stiffness of the spring. This results in the middle of the rope curving downward. This bow is consistent with any line or cable that bends due to the influence of gravity, as shown in Figure 24.

There is one thing, however, that is not easily created. In order to simulate a jump rope, the correct angular speed and radius have to be chosen to the ends of the rope. If the radius is too short, regardless of the angular speed, the desired result will not be achieved. Also, if the angular speed is too slow, regardless of the radius' length, the desired effect will not be achieved either.

```
if (g_keys->keyDown [VK_UP] == TRUE)
    ropeSim->rotateEnds(1);

if (g_keys->keyDown [VK_DOWN] == TRUE)
    ropeSim->rotateEnds(0);

if (g_keys->keyDown [VK_LEFT] == TRUE)
    ropeSim->setRadius( 0.0025f);

if (g_keys->keyDown [VK_RIGHT] == TRUE)
    ropeSim->setRadius(-0.0025f);
```

Figure 26. Function update() in Rope

```
RopeSimulation(int   nOP,
               float m,
               float sConstant,
               float sLength,
               float aFConstant,
               float gRConstant,
               float gFConstant,
               float gAConstant,
               float gHeight)
{
    groundHeight             = gHeight;
    airFrictionConstant      = aFConstant;
    groundFrictionConstant   = gFConstant;
    groundRepulsionConstant  = gRConstant;
    groundAbsorptionConstant = gAConstant;

    gravitation     = Vector3D(0, -9.81f, 0);
    sl              = sLength;
    numOfParticles  = nOP;
    degree          = 0;

    particles       = new Particle*[numOfParticles];
    springs         = new Spring*[numOfParticles - 1];


    for (int i = 0; i < numOfParticles; ++i)
    {
        particles[i]           = new Particle(m);
        particles[i]->position = Vector3D(sl*(i-(nOP/2)), 0.0f, 0.25f);
    }

    ropeConnectionPos = particles[0]->position;

    for (int j = 0; j < numOfParticles-1; ++j)
    {

        springs[j] = new Spring(particles[j],
                                particles[j+1],
                                sConstant,
                                sl);
    }

    degree = 0.0f;
    radius = 0.25f;
}
```

Figure 27. RopeSimulation Constructor

As shown in Figure 27, when the (jump) rope object is initialized, the particles are created parallel to the x-axis where y=0 and z=0.25. The rope is perpendicular with, and the two ends are equidistant to, the y-z plane. It starts from the negative side of the x-axis and ends to its positive side.

Before the springs are attached, the position of the first particle (ropeConnectionPos) is set equal to its initial position. The values for the initial angle and radius are set. The position of the last particle, the other end of the rope, is not done in the constructor. Initializing the position of the last particle in the constructor was never attempted. Instead, it was done in the function simulate() as shown in Figure 28.

```
void simulate(float dt)
{
    int i;

    for (i = 0; i < numOfParticles; ++i)
        particles[i]->simulate(dt);

    ropeConnectionPos += ropeConnectionVel * dt;

    particles[0]->position = ropeConnectionPos;
    particles[0]->velocity = ropeConnectionVel;

    particles[numOfParticles-1]->position = Vector3D(ropeConnectionPos.x+(sl*(numOfParticles)),
                                                     ropeConnectionPos.y,
                                                     ropeConnectionPos.z);
    particles[numOfParticles-1]->velocity = ropeConnectionVel;
}
```

Figure 28. Function simulate() in RopeSimulation

The function that rotates the ends of the rope is shown in Figure 29. The integer value it receives is used for deciding whether the ends of the rope go clockwise or counterclockwise. The if-statement that resets the value of degree back to zero is not necessary. This value can go as high or as low as the computer system will allow.

The last thing done is set the values for the initial particle's position on a circumference parallel to the y-z plane. The position for the other end would have the same values for the y- and z-components. As previously mentioned, its position is updated in the function simulation(). Figure 30 shows how the size of the radius is increased or decreased.

```
void rotateEnds(int direction)
{

    if (degree >= 360.f)
        degree = 0.f;

    if (direction == 1)
        degree += 0.1;
    else
        degree -= 0.1;

    ropeConnectionPos.z = radius * cos(degree);
    ropeConnectionPos.y = radius * sin(degree);
}
```

Figure 29. Function rotateEnds() in RopeSimulation

```
void setRadius(float rad)
{
    radius += rad;
}
```

Figure 30. Function setRadius() in RopeSimulation

THIS PAGE INTENTIONALLY LEFT BLANK

V. APPLYING AND TESTING THE CONCEPT TO ODE

Given the concept from Chapter IV, there are two ways to create a rope model using ODE: the first is by attempting to transfer a modified version of the mass-spring method to physics engine; the second is to apply the concept by working with existing functions in ODE. The first option is the more desirable of the two. However, given the complexity of the material and the limited resources available, the second choice is the more logical option if the intent is to simply generate a simulation of a rope.

A. A ROPE MODEL IN ODE

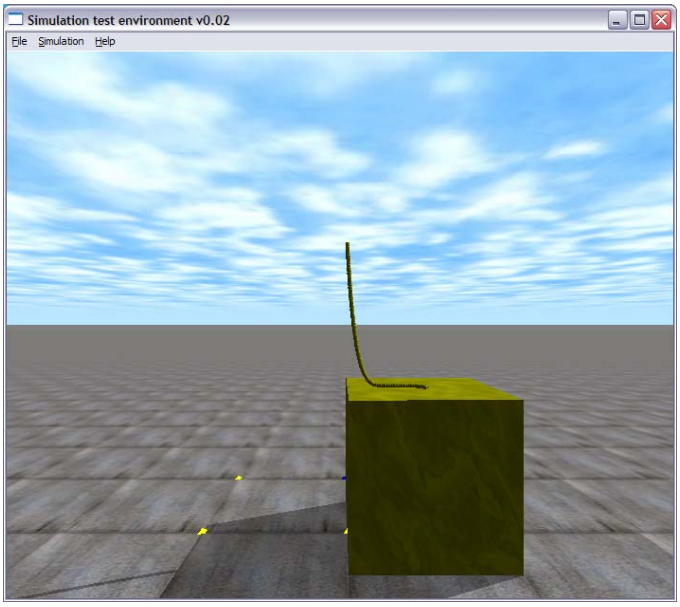

Figure 31. Rope In ODE

The rope simulation depicted in Figure 31 is done by creating a number of spheres (or nodes) and joining them successively with ball-and-socket joints. There are other types of joints available in ODE. However, the ball-and-socket joint is the only one that provides the required degrees of freedom for the connections between the spheres.

The image shown in Figure 32 illustrates what happens when only the actual spheres connected by the joints are drawn. The rope shown in Figure 31 included the “hollow” spheres that served to connect the nodes; they created an illusion of a solid and continuous rope. The hollow spheres are drawn in a straight line between nodes. These spheres are referred to as hollow since they do not interact with any other object in the environment. They penetrate through any object that’s in their path.

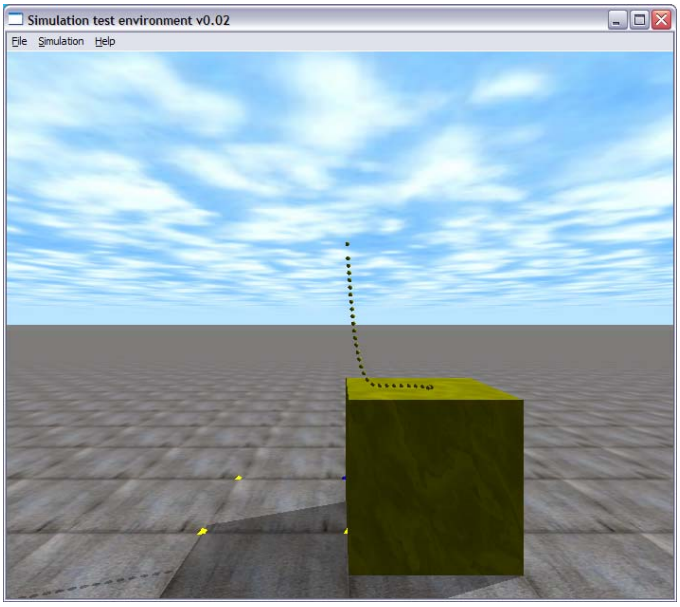

Figure 32. Rope (nodes only) In ODE

The intent in this segment is to simply demonstrate a rope in a virtual environment using ODE. Unlike the previous chapter, three items are in this environment: the rope, the ground, and the box. The static box is introduced for the rope to interact with.

The commands for moving the rope around are shown in a separate (command) window. The keys are used to control the movement of the first node that hangs in mid-air. The rest of the rope follows and hangs below it due to gravity. The directions available for each of the first node’s step are limited to the three axes. The code is written such that movements can only be done in one direction (from the three axes) at a time.

Since gravity pulls down any object that contains a mass, a method has to be devised in order to keep the rope suspended in mid-air. Unfortunately, there is no specific function available in the physics engine that will facilitate this need. dBodySetPosition() is only good for initializing objects in the virtual environment.

The function can probably be applied during real time to objects that are not connected to anything, but it does not work properly for the rope model. If the distance between the first node's current position to the desired position is far enough, this function will "yank" the first node and leave the rest of the rope behind.

The only function left that will do the "trick" is dBodyAddForce(). So, the solution used in this paper goes back to the equation $f = ma$. In Chapters II, III, and IV, the distance is derived by using force, mass, and time. In this section, distance, mass, and time are given to solve for the force needed to move the first node from point A to point B. The revised formula is shown below:

$$f = m\left(\frac{d}{t^2}\right)$$

ODE uses a time step to keep the simulation in motion relative to real time. This time step and mass are both defined prior to the execution of the program, and both are part of the solution. The third part, distance, is derived from the current position and the desired position. The current position is obtained by using dBodyGetPosition(). The desired position is either the last desired position or is the one updated by the user by pressing certain keys on the keyboard.

With the three aforementioned components, the first node of the rope is kept at a fixed point and moves at the control of the user. However, the solution has a minor side effect that is sometimes noticeable. The method produced makes an adjustment to the first node every time step. The lag in computer processing sometimes causes a visible displacement of the first node every time step. This is almost negligible from a distance; however, if viewed at close

range, one can sometimes see the “jerking” phenomenon caused by the constant adjustment to fix the first node in one place. Reducing gravity in half (from -0.5 to -0.25) reduced the jerking of the node. The adjustments being made are smoother.

B. A SLIDING WEIGHTED SPHERE ON A ROPE

Two versions were produced for this scenario. Unlike the previous chapter, the issue or concern here is not only with the effect of gravity, it is also with the ability to transfer the heavier sphere from node to node (or to slide down the rope). The first version has the weighted sphere located at a fixed point on the rope; the other has the weighted sphere sliding down the rope.

1. Weighted Sphere at a Fixed Point (on the Rope)

This setting was first created in order to examine the behavior of the rope before continuing with the development of the sliding heavier sphere. This way, unwanted results or behaviors can be isolated and addressed prior to combining all the components. On the other hand, desired features can also be identified and transferred to the final product.

Again, a box, non-static this time, is created as another object for the rope to interact with. If the rope is to be used as a model for an interactive simulation, its interaction with other objects has to be observed as well. The mass of the box and the weighted sphere is 1.0, while the value for the rest of the spheres is 0.005. The influence of gravity on all objects in the simulated environment was reduced to half the size of its original value. This was done to minimize the stretching on the rope.

There are three conditions (based on time step) that were observed based on this setting. Given the aforementioned setup values, the three time steps observed are 0.075, 0.050, and 0.025.

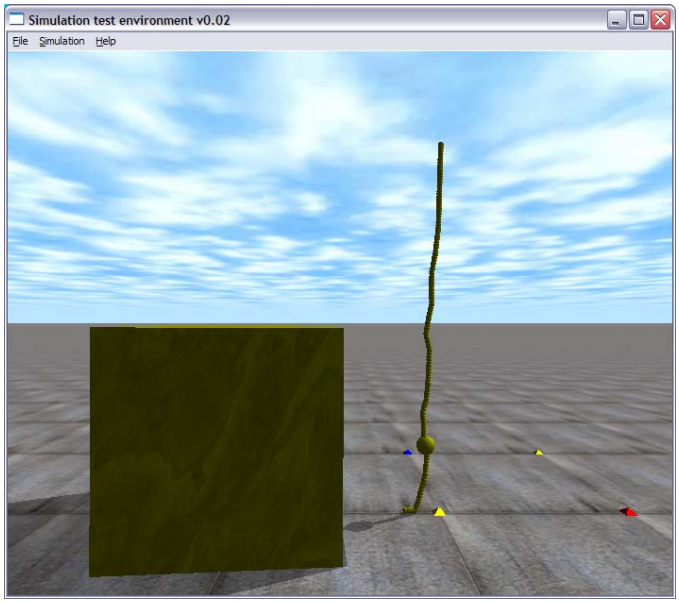

Figure 33. Rope With Weighted Sphere at 0.075 Time Step

With 0.075 time step, the elasticity in the rope cannot keep up with the gravity's pull on the weighted object. The motion of the weighted sphere is also slow. It also caused an inconsistent physical behavior with the rope. The adjustments of the joints caused the "wiggling" shown in Figure 33.

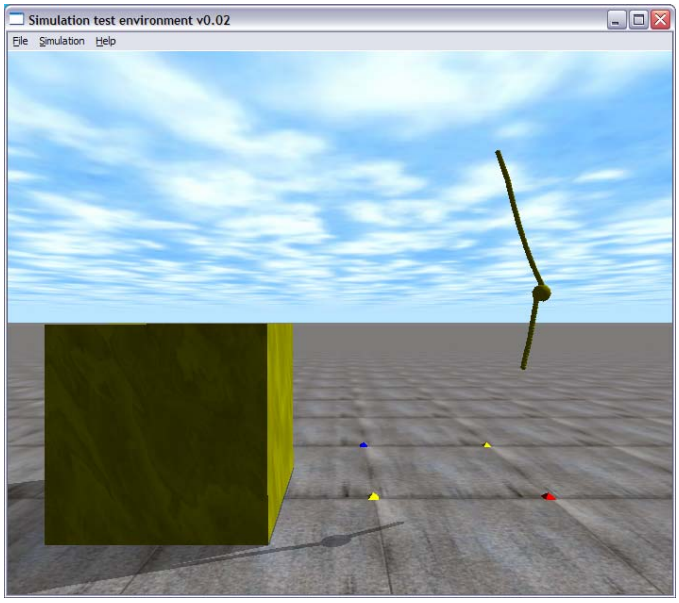

Figure 34. Rope With Weighted Sphere at 0.050 Time Step

Changing the time step to 0.050 reduced the wiggling on the rope. The rope is also less elastic, so it is not as stretched as in the previous condition. On the downside, the simulation has also slowed down even more.

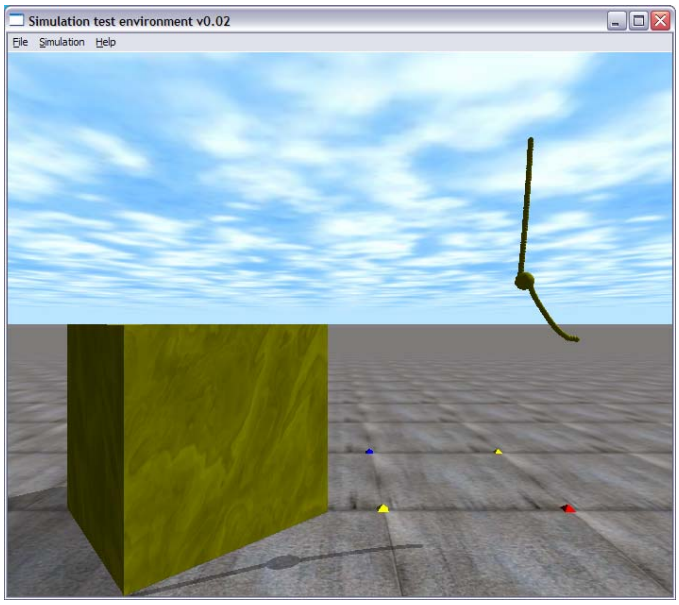

Figure 35. Rope With Weighted Sphere at 0.025 Time Step

A pattern emerges for this condition. The lesser the time step, the lesser the rope's elasticity and the slower the heavier sphere's motion. As expected, when the time step was reduced to 0.025, the rope is even less elastic. If the wiggling exists, it's unnoticeable. The motion of the heavier sphere, however, slowed even further. Like in the previous chapter, reducing the influence of gravity on the weighted sphere (no matter the size) created a "slow motion" effect on the heavier sphere.

On the other hand, the weighted sphere's interaction with the box on all three time steps was consistent. They did not penetrate one another. Instead, when the weighted sphere is given enough momentum (similar to a wrecking ball), the box will bounce back when hit.

2. Sliding Weighted Sphere

Several attempts were made to enable the weighted sphere to slide down the rope. The image of the larger sphere moving down the rope was achieved; however, the physical behavior was not. The rope did not behave as if there was an object of larger mass that is tugging on it at the position of the weighted node. It behaves as if the larger sphere still has the same mass as the rest of the spheres in the rope. The larger sphere also partially penetrates through the box and the floor.

When the larger sphere hits the box with speed similar to the fixed version, the impact to the box is minimal. The box moves as if it were hit by one of the smaller spheres. To mitigate this condition the mass of the box was reduced to that of the mass of the regular spheres. This made the heavier sphere's impact with the box more pronounced; unfortunately, the same can be said of any sphere on the rope.

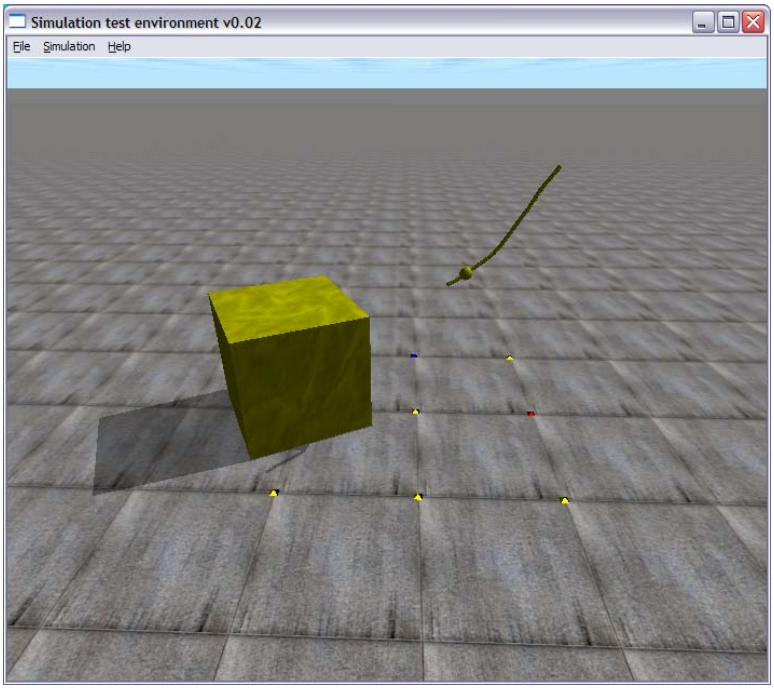

Figure 36. Rope With a Sliding Weighted Sphere

The functions dBodyGetMass(), dBodySetMass(), dMassAdjust(), and dMassAdd() were all used in different combinations. Unfortunately, none yielded the desired results. The conclusion, in regards to these functions, is that the mass is one element that is not designed to be changed during runtime. Using several combinations of the aforementioned functions even caused instability to the program. Overall, none can be declared a successful attempt in creating a believable rope simulation with a sliding heavier, larger sphere.

C. A JUMP ROPE

The attempt in creating a jump rope in ODE cannot be considered a success either. Although the concept is similar with that of the previous chapter, the "extra" elasticity on the rope creates a new situation that needs to be addressed. The mechanics of the rope work and if the computer's processor is not taxed, the simulation will initially run fine. However, after a few loops, the middle part of the rope often starts to lag behind.

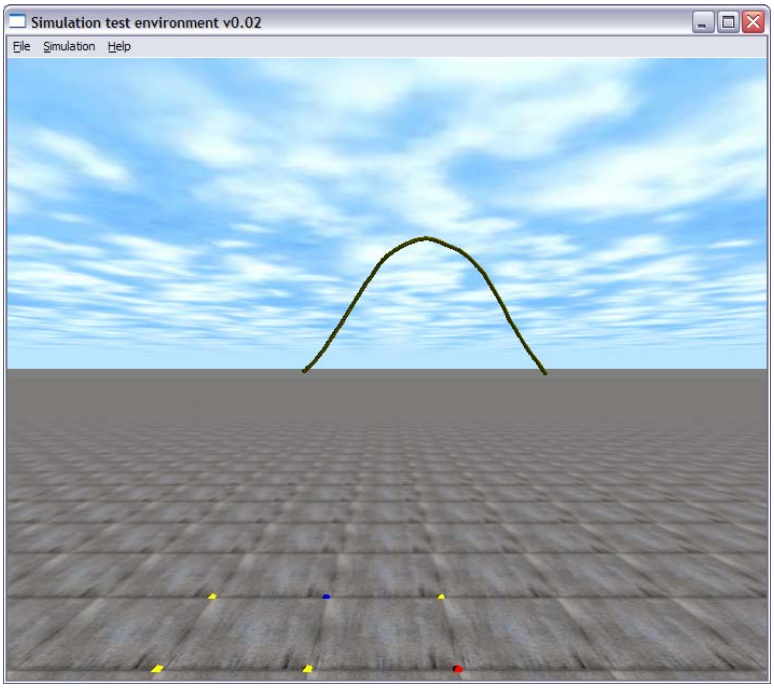

Figure 37. Snapshot of Jump Rope In ODE

The source code written for this situation allows the user to adjust the radius and rotational speed in real time. This is, however, not the answer to this dilemma, but at least the user has the ability to make adjustment if he/she chooses.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

This work has tremendous potential but much still needs to be done. One has to remember that ODE was not designed for deformable objects. An attempt to create a rope model consisting of "rigid bodies" of spheres did not fair as hoped. The illusion was not complete primarily due to the elasticity of the joints and the computational limit created by excessive bodies. The other option listed in Chapter V is the only option available.

There are other issues that make the transfer of concept from Tunca's code to ODE a challenge. A few are listed below along with a couple of potential solutions. Other areas that may be of importance to the subject are covered as well for future studies.

Although it may seem that the concerns addressed are of no relevance to the ultimate goal, the concept should remain the same. It is the concept that may be of relevance, as some of these conditions may surface again.

A. LIMITATIONS

1. Number of Particles (or Spheres)

With the improved version of Tunca's code in Chapter IV, the number of particles included can be as much as 200 with no ill-effect to the rendering of the rope. At 300 particles, the slow down in rendition (or slow motion) is obvious. The number of particles can still be increased. However, the increase in the number of particles is in proportion to the decrease in rendition speed.

As for ODE, using rope_1.cpp as the base source code, the maximum number of (sphere) bodies for the rope is 59. At this number, the simulation is susceptible to error and locking up, or the program terminating itself. In fact, between 53 and 59 bodies, immediately after the program has started, if the only thing done is to slowly lower the first node down, the program will lock up and display an error message.

When 60 bodies are created, an error message appears immediately. It is obvious that as the number of bodies increase, the harder the computer's processing unit has to work. The issue is not directly associated with the algorithm; rather, it is with the volume of computations that needs to be accomplished before the next time step. If the processor can't keep up with the demand, an error message is produced and the program locks up or disengages.

At 52, it takes a little bit more work to induce error to the system. The rope used in Figures 31 and 32, has 30 bodies. As long as there is no other program running concurrently, this amount is fairly stable for the computer system used for this thesis.

The instability is caused by the accumulated errors in computation, compounded computational demands, or a combination of both. Time step has to be decreased in order to eliminate this instability. However, decreasing the time step also decreases the simulation speed.

2. Thresholds on Mass, Force, and Time (in Tunca's Code)

In Tunca's code, distance is derived by using force, time, and mass. Looking back at equation 1, one can see how the three factors affect the distance traveled by the particle. Most of all, as m approaches zero, d goes to infinity. Therefore, m cannot equal zero.

Given that t is 0.001 second (sec) and m is 0.05 kilogram (kg), force is the only factor that is not constant. As for the force, the spring has the largest effect on the position of the mass. The biggest influence in the spring is its "stiffness." Therefore, the stiffness value is of biggest concern.

Using equation 3, $d = -20,000 \cdot (x - d) \cdot 0.00002$, or $d = -0.4 \cdot (x - d)$. So, the distance the particle will travel primarily depends on its current distance relative to the desired distance (multiplied by the magnitude 0.4). This is a reasonable amount of force since it is not only applied to one particle, but on both particles that the spring is attached to.

Using a magnitude of 0.5 may seem logical at first, but with the exception of the two particles at the ends, each particle is connected to two springs. These particles are adjusted twice. Therefore, the better approach would be to apply a force of lower magnitude and gradually ease them into the right distance from one another. A magnitude larger than or equal to 0.5 will produce an over-correction.

Unfortunately, this does not address the jarring condition described in Chapter IV. Reducing the stiffness of the spring does not alleviate this condition. Reinstating the effect of the spring's internal friction does. However, small it may be, it also brought some elasticity back to the rope.

This is one of the drawbacks in using a spring. If the spring is weak, it's elastic. If it's too stiff, the spring will produce an appearance similar to a chain link. It is up to the developer to decide how much of each component needs to be applied for the simulation to be considered believable.

Another interesting aspect to the source code is the mass and its impact on the algorithm's stability. Mass approaching the value zero also causes instability to the program. The size of the mass limits the value of the numerator ($f \cdot t^2$).

Given that time increment is 0.001 (sec) and the spring's stiffness is 20,000, the algorithm is considerably stable at 0.05 kg per particle. If the mass is reduced to 0.02 kg, the number of particles at which signs of instability in the rope will appear is around 60. So, increasing the size of the mass allows an increase to the spring's stiffness and the program's stability. The reverse is also true.

3. Computer Processing Speed

In Tunca's code, decreasing time increment has the same effect as increasing the size of the mass, or in another way, enabling the algorithm to reduce the size of the mass. However, time increment is dependent on the speed of the processor used.

For this thesis, the increment was reduced from 0.002 sec to 0.001 sec. The processor used is adequate enough to go through the iteration every millisecond. Running the program with a slower processor will likely still be possible. However, the penalty would come from the speed at which the simulation is rendered.

As for ODE, there is no guarantee to the stability of the codes written. Depending on the system's processing speed and the concurrent tasks at hand, there is a possibility that any of the codes can lock up or cause an error. This issue requires further study to understand, and not so much as to directly mitigate this condition, but to understand and define what limitations need to be imposed on the source codes or to the system requirement.

4. Challenge in Transferring the Concept from Tunca's Code to ODE

In ODE, the type of joint analogous to the spring is the ball-and-socket joint. This joint is elastic by design. Stiff springs and stiff forces are considered bad. [14] This aspect was left as is and was not studied. Therefore, the rationale behind the elasticity is not realized in this thesis.

In ODE, the size of the time step dictates the tradeoff between speed and the combined aspects of accuracy and stability. The smaller the time step, the more accurate and stable the simulation. [15] This is also true with Tunca's code. However, where the two differs is how time is used by both. ODE utilizes a constant time step, while Tunca's code uses a variable one.

With a constant time step, ODE forces all algorithms to finish before it moves on to the next time step. This makes the program more accurate and stable at smaller time step. However, it gets penalized on the speed at which it is rendered. Increasing the time step makes it faster, but gets penalized on accuracy and stability. The larger the time step, the smaller the tolerance for error.

In the modified version of Tunca's code, the increment is one millisecond. The number of times it is iterated is based on the time it receives from the system. Stability is increased by reducing the time increment from 2 to 1 millisecond. If instability occurs, the rationale will most likely come from the size of the mass.

Utilizing the concept of spring (from Tunca's code) directly to ODE will not work due to the difference in how time increment is applied by the two. As shown in Figure 15, the position of each particle is updated every millisecond. The number of times this is done is based on the number of iterations. The number of iterations received is the time elapsed in milliseconds from the system's time. For an Euler method, Tunca's algorithm is very stable and accurate. In ODE, however, time step is analogous to the number of iterations (in Tunca's code). The particles (or spheres) are updated every time step, not every millisecond. This is the source of the challenge.

B. POSSIBLE MITIGATIONS TO CURRENT ISSUES

Some of the issues encountered for the source codes written for both Tunca's and ODE may surface again. Therefore, the following proposed solutions are presented for the following specific areas.

1. Variable Friction Value for Excessive Swaying

When a pendulum swings, the only force acting on it is gravity, yet its movement has a vector component perpendicular to gravity and sways from left to right. If no force is further applied, air friction (drag) and gravity will slow the pendulum down and bring it to a complete stop. Air friction is already accounted for in Tunca's code. Since the code has been modified, gravity's effect on the horizontal motion is not.

This trigonometric relationship between the horizontal and vertical vectors is one idea that can be applied to the weighted particle to mitigate its excessive

swaying. Since the influence of gravity is already reduced in the weighted particle, the variable force (or friction) needs to be applied to the horizontal components only. Its value must be proportional to the weighted particle's horizontal velocity. Also, this friction must only be active when the first node stops moving. Otherwise, this added friction will produce excessive and unwanted dragging of the weighted particle.

2. Applying Force in Sliding Weighted Sphere in ODE

As described in Chapter V, attempts to create a sliding weighted sphere did not succeed. The attempts were primarily focused on the mass. However, mass is only one variable in equation 2. What is important is the desired result on the left side of the equation which is the distance traveled.

If mass is left untouched and focus is placed instead on the force, there may be a way to slow the weighted sphere down, or speed it up if necessary. As far as colliding with other bodies, the same force needs to be increased in order to simulate a heavier mass on impact.

C. FUTURE WORKS AND STUDIES

1. Possibility of Applying Interpolation

The "hollow" spheres described in Figure 32 are drawn in a linear manner between nodes. This sometimes gives the appearance of a connection of links and not a rope. To create a more realistic-looking rope, a cubic interpolation can be used to lay the path for the positions of the additional spheres.

2. Possibility of Using Vertices and Triangle Mesh

The rope generated in Chapter V "is a body of bodies." Rigid bodies created in ODE have certain attributes that are not needed in simulating a rope. Removing properties such as orientation, angular velocity, and inertia matrix from

each body should not produce adverse effects to the rope's simulation. In fact, removing these features should increase the speed at which the bodies are computed. However, the computing speed gained is unknown at this time.

Also, in rendering the rope in ODE, the first sphere is used in place of all the spheres in the rope. This was done intentionally for visual reasons. If the actual spheres were drawn in the rendering phase, the simulation will show the spheres rotating independent of one another. The first node does not rotate most of the time. Even when it does, at least all of the bodies in the rope will have a synchronous rotation.

Generating a rope model, such as the one done in Chapter V, is not the ideal way. At close range, one can see the spheres that make up the rope. In addition the hollow sphere bodies can penetrate through any protruding part of any body, such as the edge of the box or within the rope itself.

Also, in certain conditions, the box can get wedged in between two connected spheres bodies. Turning all the spheres in the rope into "solid" is not an option either. So, a better method needs to be investigated since this approach is not acceptable.

One possible solution to the aforementioned dilemma is applying the concept of vertices as shown in Figure 38. The blue green and light green disks are analogous to the actual and hollow sphere bodies, respectively. Here, however, the key components are not the disks, but the vertices generated from the disks.

As described previously, the positions of the generated disks can be derived from the line (or equation of a line) generated by interpolating the positions of the actual disks. The disks, both actual and generated, have to be tangent to the line. This can be solved by using the first derivative of the interpolated line. From the disks, vertices can be generated. Texture mapping can then be applied to these vertices.

For collision detection, triangle mesh (or trimesh) can be used to wrap around the rope. Also, it may be possible to only partially cover the rope's surface. If the trimeshes are distributed, such as in the examples shown in

Figure 39, penetration of the uncovered parts should be almost impossible. If there is a penetration, it would be minimal, and may actually be better in a sense that it will mimic indentation. This would be consistent with a real rope since ropes are often made of pliable or soft material.

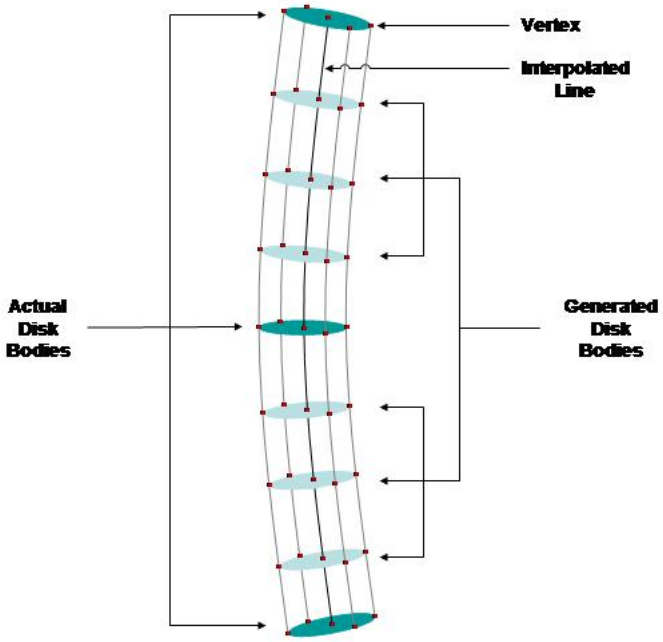

Figure 38. Rope Using Disks to Generate Vertices

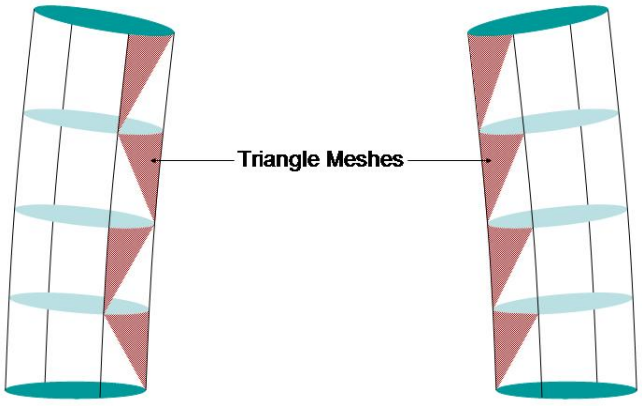

Figure 39. Triangle Meshes for Collision Detection (1)

One does not have to be limited to the trimesh examples shown in the previous page. These are just ideas. One may consider the example in Figure 40 if less accuracy is acceptable or desired. The benefit of larger trimeshes includes the further reduction in their quantity.

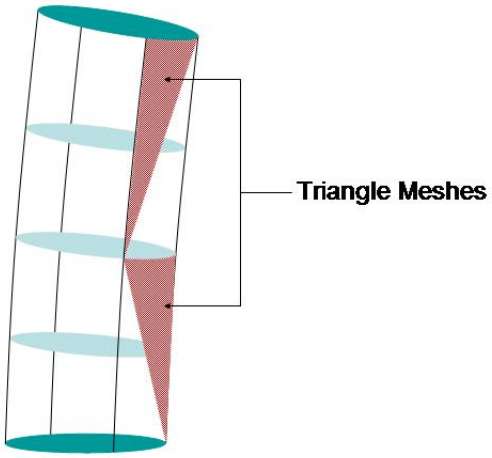

Figure 40. Triangle Meshes for Collision Detection (2)

ODE has a feature called space. Space is utilized to speed up collision detection by performing a process called collision culling.[16] By combining this feature with trimeshes, collision detections should be minimized. The trimeshes shown in Figures 39 and 40 can even be made to overlap, since it is possible to ignore collision detection between objects that are in the same space.

Another interesting approach in developing a rope is using diagonal springs to connect the disks as shown in Figure 41. The springs will not only maintain the distance between disks, they will also maintain the positions of vertices of one disk relative to vertices of the adjacent disks.

In effect, it will influence the rope's lateral movement. This setup will add torque resistance between disks. It will also create a tension that will resist bending or that will produce curling.

However, numerous springs will require a faster computational processor. One possible solution may be a combination of all the aforementioned that enables the features when needed, and disables when not. It is also possible to examine each situation on a case-by-case basis and use only the solutions that are needed.

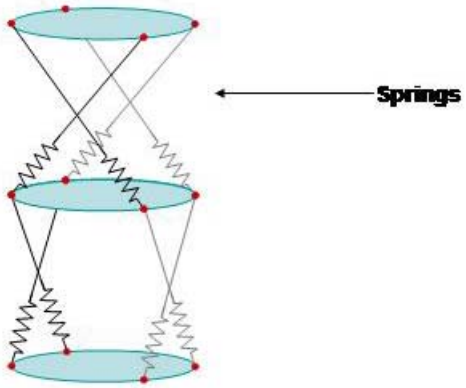

Figure 41. Springs Connecting Disks

LIST OF REFERENCES

1. Thomas Jakobsen, “Advanced Character Physics,” <http://www.ioi.dk/Homepages/thomasj/publications/gdc2001.htm> (1 August 2004).

2. Andrew Bowell, “Ready to Wear,” CGI Magazine, January 2002, <http://oldsite.havok.com/news/coverage/01-01-02.html> (1 August 2004).

3. Bowell.

4. Erkin Tunca, “Rope Simulation,” <http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=40> (1 August 2004).

5. Tunca.

6. Russell Smith, “Open Dynamics v0.5 User Guide” <http://ode.org/ode-0.5-userguide.html#sec_1_1_0> (29 May 2004).

7. Smith, 1.

8. Smith, 1.

9. Smith, 1.

10. Smith, 1.

11. Rick Parent, Computer Animation: Algorithm and Techniques (San Francisco: Morgan Kaufman Publishers, 2002), 216.

12. Parent, pg. 216.

13. Parent, pg. 217.

14. Smith, 74.

15. Smith, 73.

16. Smith, 52.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Robert McGhee
   Naval Postgraduate School
   Monterey, California