Introduction to Soft Body Physics

Skeel Keng-Siang Lee

skeel@skeelogy.com
http://cg.skeelogy.com

Contents

Introduction

In recent years, there has been a high proliferation in the usage of physics simulations in games. This might be due to the increasing need for realistic movements of the objects to match their realistic renderings. Some of these physics simulations include rigid body simulations where the objects move and rotate but do not change their shapes, and ragdoll physics where the motion of a character can be simulated under force perturbations. There has however not been a lot of soft body physics in games yet, thus it might be worthwhile to start diving into this type of simulation so that you can start implementing some soft bodies for your game.

A soft body is basically an object which changes its overall shape due to external forces acting on it. Some examples that we commonly see include cloth, balloons and jelly. A cloth drapes downwards due to gravitational forces and flutters due to wind forces. Similarly, a balloon enlarges when the internal pressure forces increase, and produces indentations when a child exerts forces on it by squeezing. These kind of natural behaviors are interesting to observe, especially in a game environment where the entertainment value is of high importance.

This tutorial has been written to help you get a quick understanding of some basic theories and implementations behind simulating soft bodies in real time. It is meant for beginners as we will start from the very basics. In an effort to show the usefulness of soft body physics, some of the examples given in this tutorial also illustrate how it can be used as a quick substitution for cases where using rigid body simulation will prove to be much more expensive.

There are four major chapters in this tutorial and we will be creating a different soft body simulation for each of them. This tutorial progressively builds an object-oriented soft body physics system, thus we will be heavily reusing and inheriting from the codes written in earlier chapters. It assumes that you have a basic understanding of object-oriented programming.

This tutorial primarily targets users of Microsoft XNA 3.0 but the ideas and algorithms can easily be implemented using other languages and APIs once you understand them.

By the end of this whole tutorial, you will be able to create these simulations:

Video 1: *An interactive cloth simulation*

[Click on image above to watch the video]

Video 2: *An interactive slinky simulation (created using a goal-based soft body)*

[Click on image above to watch the video]

Video 3: *An interactive chain simulation (created using springs and length constraints)*

[Click on image above to watch the video]

Base Codes

I have included some basic game components and classes in the project files of each chapter so that certain basic tasks can be performed easily. Some of the things that these game components can do include:

i) Loading and drawing a model from file by calling `LoadModel(”theFileName”)`

ii) Transforming of a model in the scene by setting the `Translate`, `Rotate` and `Scale` properties

iii) Creating of a camera by creating a `CameraComponent` and adding it to `game.Components`

iv) Detecting if a key is held down, just pressed or just released by calling the methods `IsKeyHeldDown()`, `IsKeyJustDown()`, `IsKeyJustUp()` respectively

v) Creating a 3D line by just creating a `Line3DComponent`, adding it to `game.Components` and then setting the `StartPosition`, `EndPosition` and `Color` properties

These base codes basically encapsulates some of the basic operations in XNA. They are meant to improve the clarity of the soft body simulation implementations, which will otherwise be affected by unnecessary codes that are there to load models, draw primitives, detect if a key is just pressed etc. These classes are pretty much the basics of XNA, so if you know your XNA well enough, it should not be too hard to understand what is written in there without me explaining them.

To use the base codes, you just need to create a new instance of each of the components and add them to the `this.Components` list in your game. This has been done for you in Game1.cs in each of the chapter files.

(Game1.cs, addition of codes in bold)

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using SkeelSoftBodyPhysicsTutorial.Main;

namespace SkeelSoftBodyPhysicsTutorial
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
```

```
//component members
CameraComponent cameraComponent;
InputComponent inputComponent;
ModelComponent modelComponent;
Line3DComponent line3DComponent;

public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    //instantiate the components and add them to this.Components
    Vector3 cameraPosition = new Vector3(0, 10, 10);
    Vector3 targetPosition = Vector3.Zero;
    Vector3 upVector = Vector3.UnitY;
    string backgroundImage = "";
    cameraComponent = new CameraComponent(this, cameraPosition,
                             targetPosition, upVector, backgroundImage);
    this.Components.Add(cameraComponent);

    bool showMouse = false;
    inputComponent = new InputComponent(this, showMouse);
    this.Components.Add(inputComponent);

    modelComponent = new ModelComponent(this);
    this.Components.Add(modelComponent);

    line3DComponent = new Line3DComponent(this);
    this.Components.Add(line3DComponent);
}

protected override void Initialize()
{
    base.Initialize();
}


protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
}

protected override void UnloadContent()
{

}

protected override void Update(GameTime gameTime)
{
    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    base.Draw(gameTime);
```

```
        }
    }
}
```

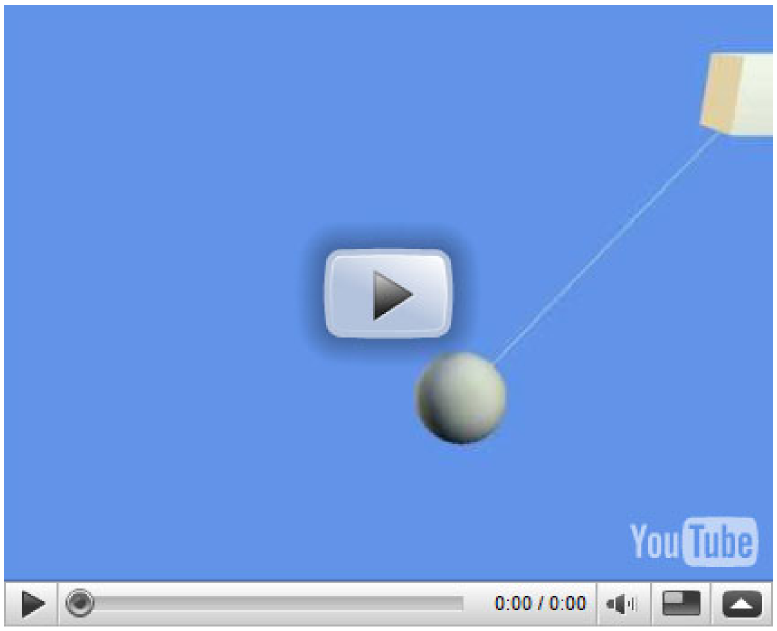
The rest of the codes above are just the standard template codes that are provided when you create a new project in XNA Game Studio, with the comments removed.

Chapter 1 - Spring Simulation

Get the source files for the start of this chapter from the SkeelSoftBodyPhysicsTutorial-Chapter1-BEGIN folder

We will start off the tutorial by implementing a simple spring simulation where two objects are attached via a spring. By learning how to create a spring simulation, you will gain enough knowledge on how to perform soft body simulations in games. This simple simulation will thus prepare you for more complicated soft body simulations in later chapters.

By the end of this chapter, you should have a simple spring simulation like this:



Video 4: *A simple spring simulation between two objects*
[Click on image above to watch the video]

It does not look particularly impressive because it is just a basic test of how to create a spring and run the simulation. However, do not get too worried about it as we will create much better-looking simulations in later chapters.

1.1 Physics Simulation Algorithm

To implement any physics simulation in games, we need to do these steps (in order):

i) Sum up all forces acting on a body/vertex to get the resultant force

ii) Find acceleration by dividing the resultant force by the object mass: a = f / m (Newton's Second Law for a constant mass)

iii) Perform numerical integration on the acceleration to get velocity/position

iv) Update the body/vertex with the new velocity/position, and repeat from (i)

At each time step, we get a new position for the simulated object, based on all the forces that are acting on it. As we update the object with this new position, we will observe its motion under the simulation.

We will cover more algorithms and formulas as we move along with the implementation, particularly on how the different types of forces are calculated and how to perform the numerical integration.

In order to implement this simulation system, there are four main groups of classes that we will be creating:

- **Simulation objects**: These pack an object (can be a model, vertex, triangle etc) together with its simulation attributes such as position, velocity and mass into one.
- **Force generators**: These represent things that can generate forces, such as a spring or gravity.
- **Integrators**: These perform numerical integration using the acceleration to find the new position/velocity, and updates the corresponding attributes in the given simulation object.
- **Simulations**: These are the main classes that will manage the whole simulation algorithm. The tasks that they will do include initializing all the variables, connecting the springs and updating the simulations.

1.2 Simulation Object Classes

Recall that we need to keep track of the position and possibly the velocity of the object undergoing simulation. It will be convenient if we pack the simulated object together with these simulation attributes. Thus, let us first start with the simulation object classes.

1.2.1 SimObject Abstract Class

We first create an abstract class called `SimObject`. This will contain necessary members and functions that will be inherited by other simulation object classes.

Using the Solution Explorer, add a new folder called SimObjects in the existing SoftBody folder. Then create a new class file and name it SimObject.cs.

```
(SimObject.cs)

using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.Main;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects
{
    public enum SimObjectType { PASSIVE, ACTIVE }

    public abstract class SimObject
    {
        private float mass;
        private SimObjectType simObjectType;
        protected Vector3 currPosition;
        protected Vector3 prevPosition;
        protected Vector3 currVelocity;

        public float Mass
        {
            get { return mass; }
            set { mass = value; }
        }

        public SimObjectType SimObjectType
        {
            get { return simObjectType; }
            set { simObjectType = value; }
        }
        public Vector3 CurrPosition
        {
            get { return currPosition; }
            set { currPosition = value; }
        }

        public float CurrPositionX
        {
            get { return currPosition.X; }
            set { currPosition.X = value; }
        }

        public float CurrPositionY
        {
            get { return currPosition.Y; }
            set { currPosition.Y = value; }
        }

        public float CurrPositionZ
        {
            get { return currPosition.Z; }
            set { currPosition.Z = value; }
        }

        public Vector3 PrevPosition
        {
            get { return prevPosition; }
            set { prevPosition = value; }
        }

        public Vector3 CurrVelocity
        {
            get { return currVelocity; }
            set { currVelocity = value; }
        }

        //-----------------------------------------------------------------

        public SimObject(float mass, SimObjectType simObjectType)
        {
            this.mass = mass;
            this.currPosition = Vector3.Zero;
            this.prevPosition = currPosition;
            this.currVelocity = Vector3.Zero;
            this.simObjectType = simObjectType;
        }
    }
}
```

A `SimObject` has a `mass` and is of a `SimObjectType` which is either active or passive. Active simulation objects have their positions and velocities determined by the forces in the system, while the passive ones do not. Passive simulation objects are still considered part of the simulation since they allow active objects to interact with them. Setting an object to be passive is useful in cases where we need to allow the user to control its position, possibly using the mouse or keyboard.

In the constructor above, we also initialize the `currPosition`, `prevPosition` and `currVelocity` variables to the zero vector.

Recall that we have to sum up the forces acting on this simulation object. It will be convenient for the object itself to store the resultant force so that the object can be passed around to different classes and the resultant force is still being tracked and stored. Thus we will create a `resultantForce` variable as a class member in the `SimObject`.

```
(class member in SimObject.cs)

        protected Vector3 resultantForce;

        public Vector3 ResultantForce
        {
            get { return resultantForce; }
            set { resultantForce = value; }
        }
```

We will also create a method to reset the resultant force. This will typically be called either at the end or at the beginning of each time step.

```
(class method in SimObject.cs)

        public void ResetForces()
        {
            this.resultantForce = Vector3.Zero;
        }
```

Finally, any class that inherits the `SimObject` class should have its own `Update` method, so we define an abstract method called `Update` to make it compulsory for derived classes to implement it.

```
(class abstract method in SimObject.cs)

        public abstract void Update(GameTime gameTime);
```

1.2.2 SimModel

Based on the `SimObject` abstract class, we shall implement our first simulation object class called `SimModel`. It is meant to wrap simulation attributes to a model that we will load from file and update the position of this loaded model at every time step. In the SimObjects folder, create a new class file named SimModel.cs.

(SimModel.cs)

```
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.Main;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects
{
    public sealed class SimModel : SimObject
    {
        private GameModel model;

        public GameModel Model
        {
            get { return model; }
            set { model = value; }
        }

        //----------------------------------------------------------------

        public SimModel(GameModel model, float mass, SimObjectType simObjectType)
            : base(mass, simObjectType)
        {
            this.model = model;
            this.currPosition = model.Translate;
            this.prevPosition = currPosition;
        }
    }
}
```

Basically we pass in a `GameModel` (one of the classes provided in the base codes), its `mass` and its `SimObjectType` when we create a `SimModel` instance. The `currPosition` and `prevPosition` variables are updated to the current position of the `GameModel`.

Since the `Update` method is defined as abstract in the base class, we will have to implement it using an override method. All we have to do for this method is to update the position of the `GameModel` according to the `currPosition`, which will be recalculated at every time step. This basically synchronizes the position of the `GameModel` with the stored position.

(class method in SimModel.cs)

```
        public override void Update(GameTime gameTime)
        {
            this.model.Translate = this.currPosition;
        }
```

That is all we have to do for this class, since the other necessary implementations are all inherited from the base class.

1.3 Force Generator Classes

Now that we have the simulation objects created, we can start to implement the force generators. Recall from the simulation algorithm in Section 1.1 that we need to sum up the forces that are acting on a body at every time step. These forces will come from the force generator classes that we will be implementing now.

1.3.1 ForceGenerator Interface

We shall start off by creating a `ForceGenerator` interface and make it compulsory for derived classes to implement a method `ApplyForce`.

Create a new folder called ForceGenerators in the SoftBody folder, and create a new class file named ForceGenerator.cs in the new folder.

(ForceGenerator.cs)

```
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.ForceGenerators
{
    public interface ForceGenerator
    {
        void ApplyForce(SimObject simObject);
    }
}
```

What we are expecting the derived classes to do when implementing the `ApplyForce` method is to use the properties of the `SimObject` (such as mass, position, velocity etc) to calculate an appropriate force and add that to its `ResultantForce` property.

1.3.2 Gravity

Let us proceed on to create our first and simplest force generator: gravity. To get the gravitational force acting on an object, we need two pieces of information, namely the mass of the body and the gravitational acceleration that it is experiencing. The gravitational force can then simply be found as

gravitational force = mass * gravitational acceleration [1]

Now, add a new class file named Gravity.cs in the ForceGenerators folder and create a `Gravity` class which inherits from the `ForceGenerator` interface:

(Gravity.cs)

```
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.ForceGenerators
{
    public sealed class Gravity : ForceGenerator
    {
        private Vector3 acceleration;

        public Vector3 Acceleration
        {
            get { return acceleration; }
            set { acceleration = value; }
        }

        public float AccelerationX
        {
            get { return acceleration.X; }
            set { acceleration.X = value; }
        }

        public float AccelerationY
        {
            get { return acceleration.Y; }
            set { acceleration.Y = value; }
        }
        public float AccelerationZ
        {
            get { return acceleration.Z; }
            set { acceleration.Z = value; }
        }

        //------------------------------------------------------------------

        public Gravity()
            : this(new Vector3(0, -9.81f, 0)) { }

        public Gravity(Vector3 acceleration)
            : base()
        {
            this.acceleration = acceleration;
        }
    }
}
```

The first constructor takes in no acceleration but creates an acceleration of -9.81 downwards by default. This is the gravitational acceleration that we experience on Earth.

We now have to implement the `ApplyForce` method. As described in Eqn. 1, we need to find the product of the object mass and gravitational acceleration, and add that to its `ResultantForce` property.

(class method in Gravity.cs)

```
        public void ApplyForce(SimObject simObject)
        {
            simObject.ResultantForce += simObject.Mass * acceleration;
        }
```

1.3.3 Medium

Next, we implement a force generator called `Medium` which represents the medium that the simulation object is in. The viscosity of the medium that the body is in will determine how much drag force it will experience. An object moving in honey, for instance, will experience more drag forces than an object in air. To simplify the implementation, we shall only represent the viscosity by a drag coefficient that can be arbitrarily adjusted by the user to get the desired drag effect.

Add a Medium.cs file in the ForceGenerators folder.

```
(Medium.cs)

using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.ForceGenerators
{
    public sealed class Medium : ForceGenerator
    {
        private float dragCoefficient;

        public float DragCoefficient
        {
            get { return dragCoefficient; }
            set { dragCoefficient = value; }
        }

        //-------------------------------------------------------

        public Medium(float dragCoefficient)
            : base()
        {
            this.dragCoefficient = dragCoefficient;
        }
    }
}
```

As usual, we have to implement the `ApplyForce` method. The drag force is usually proportional to the current velocity of the body. The faster it is moving, the more drag force it experiences. Thus, all we have to do is multiply our drag coefficient with the current velocity of the simulation object:

$$\text{drag force} = -\,\text{drag coefficient} * \text{object velocity} \qquad [2]$$

Since this is an opposing force that slows down the motion of the object, we have to negate the result.

The implementation for this is as follows:

```
(class method in Medium.cs)

        public void ApplyForce(SimObject simObject)
        {
            simObject.ResultantForce += - dragCoefficient * simObject.CurrVelocity;
        }
```

1.3.4 Spring

We now implement the last force generator, which is a spring. A spring has these attributes:

- **Stiffness**: This determines the "springiness" of the spring. The higher the stiffness value, the more force that will be generated when the spring is compressed or stretched. This causes the spring to return to its original shape faster, thus creating a stiffer behavior. This is also known as the spring constant.

- **Damping**: This determines the amount of internal drag force that the spring will experience. The damping force is parallel to the direction of the spring.

- **Rest length**: This is the length that the spring will try to maintain when undergoing compression or stretching.

In terms of implementation, a spring also needs to store the two objects are that attached to it so that its current length can be calculated at any time step.

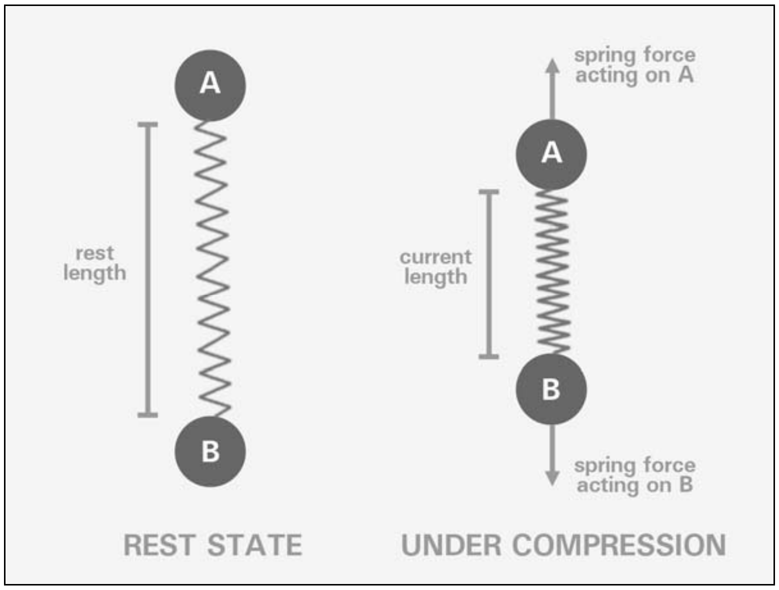


Figure 1: *Spring forces acting on the two ends of the spring when undergoing compression*

Referring to Fig. 1, the spring force experienced by object A is calculated as:

*spring force on A = - stiffness * (current length – rest length) * unit vector BA* [3]

A spring has its own damping force, in addition to the damping force caused by the medium that it is in. The difference between these two types of damping forces is that the spring damping force only acts on the spring itself while the medium damping force damps the motion of all the objects in it. The spring damping force is proportional to the relative velocities of the two objects and is usually written as:

*spring damping on A = - damping * ((velocity of A– velocity of B). unit vector BA) * unit vector BA* [4]

Note that this damping force should be in the direction from B to A. However, the difference in velocities is a vector and it may not be in that desired direction. Thus, we have to project the velocity difference vector to the direction vector from B to A. This projection is done by taking the dot product of the velocity difference vector with the unit vector from B to A (which will give the magnitude), and then multiplying that with the unit vector from B to A (which will then give the vector that we need). There is a negation in the term because this is an opposing force.

Newton's Third Law of Motion states that for every force generated, there is an equal and opposite force. Thus, the spring force experienced by object B is just the same force as experienced by object A, but in the opposite direction. This applies to the spring damping force as well.

Now that we have enough knowledge about the spring forces, let us start to implement the `Spring` class. Create a new class file called Spring.cs file in the ForceGenerators folder.

(Spring.cs)

```
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.ForceGenerators
{
    public sealed class Spring : ForceGenerator
    {
        private float stiffness;
        private float damping;
        private float restLength;
        private SimObject simObjectA;
        private SimObject simObjectB;

        public float Stiffness
        {
            get { return stiffness; }
            set { stiffness = value; }
        }
        public float Damping
        {
            get { return damping; }
            set { damping = value; }
        }

        public SimObject SimObjectA
        {
            get { return simObjectA; }
            set { simObjectA = value; }
        }

        public SimObject SimObjectB
        {
            get { return simObjectB; }
            set { simObjectB = value; }
        }

        //------------------------------------------------------------

        public Spring(float stiffness, float damping, SimObject simObjectA,
SimObject simObjectB)
            : this(stiffness, damping, simObjectA, simObjectB,
(simObjectA.CurrPosition - simObjectB.CurrPosition).Length()) { }

        public Spring(float stiffness, float damping, SimObject simObjectA,
SimObject simObjectB, float restLength)
            : base()
        {
            this.stiffness = stiffness;
            this.damping = damping;
            this.simObjectA = simObjectA;
            this.simObjectB = simObjectB;
            this.restLength = restLength;
        }
    }
}
```

The `Spring` class basically takes in the necessary attributes of a spring and stores them. It also takes in the two `SimObjects` that are attached to each end of the spring. If no rest length is supplied to the constructor, then it is just taken as the starting distance between the two `SimObjects` given.

Next, we implement the `ApplyForce` method:

```
(class method in Spring.cs)

        private Vector3 direction;
        private float currLength;
        private Vector3 force;
        public void ApplyForce(SimObject simObject)
        {
            //get the direction vector
            direction = simObjectA.CurrPosition - simObjectB.CurrPosition;

            //check for zero vector
            if (direction != Vector3.Zero)
            {
                //get length
                currLength = direction.Length();

                //normalize
                direction.Normalize();

                //add spring force
                force = -stiffness * ((currLength - restLength) * direction);

                //add spring damping force
                force += -damping * Vector3.Dot(simObjectA.CurrVelocity - 
simObjectB.CurrVelocity, direction) * direction;

                //apply the equal and opposite forces to the objects
                simObjectA.ResultantForce += force;
                simObjectB.ResultantForce += -force;
            }
        }
```

Implementing the `ApplyForce` method is a bit more involved. We first get the direction vector that points from object B to A, then we find its length and normalize it. Remember that we have to check whether a vector is the zero vector before we normalize it or else we will have a division by zero. With these information, we can calculate the spring force and spring damping forces according to Eqn. 3 and Eqn. 4 respectively. Finally, we add these forces to object A and add the equal but opposite force to object B.

Notice that we are not using the `SimObject` that has been passed in as a parameter, since we have already stored both the necessary `SimObjects` in the class itself. This is a bit of a quirkiness caused by implementing the method specified in the interface class. However, it does not cause much harm and we can just pass in `null` when we are calling this method for a `Spring` instance.

1.4 Integrator Classes

Recall that once we have the acceleration of the object, we need to perform integration with respect to time to obtain the velocity and position of the object. In a real-time environment, we are not able to perform integration in an analytical manner, meaning that you cannot just integrate an equation like what you did in your basic calculus classes (e.g. "add one to the exponent and divide the whole equation by it…"). What we have to rely on now is a numerical method called numerical integration. Such a numerical solution allows us to sample at discrete points in the integral to get an approximate solution. In our case, we sample the integral at each time step (for single-step integrators). Numerical integrators allow us to use some equations involving acceleration, velocity and position to approximate the new position of an object.

One major concept that is associated with numerical integrators is that there is always an error associated with the numerical integrator that we use. This is roughly indicated by the *order* of the integrator. The higher the order, the less error there is. However, higher order integration methods usually means more calculations and thus slower in execution.

There is a variety of integrators available for us to choose from, ranging from a Forward Euler integration method which is the fastest but not very stable (first order), to something like a Runge-Kutta Fourth Order which is very stable but very expensive in calculations as well. For games, there is a popular middle-ground technique called Verlet integration which is fast and stable (second order). For this tutorial, we shall start off with a Forward Euler integrator since it is the simplest to understand and implement, and then move on to a Verlet integrator in later chapters.

These integrators require different attributes of the object. Some require you to store the current and previous position, while some require you to store the current position and current velocity. Whatever the case is, the new position for the next time step is always generated, and that will be used to update the `CurrPosition` property of the `SimObject`.

Now that we have a brief overview of numerical integrators, we can start creating the Integrator classes. These classes basically take in the acceleration of a `SimObject` and performs numerical integration to find the new position and possibly velocity.

1.4.1 Integrator Abstract Class

As usual, we will first create an abstract class for other integrator classes to derive from. First, create a new folder called Integrators under the SoftBody folder, and create a new class file named Integator.cs in it.

(Integrator.cs)

```
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Integrators
{
    public abstract class Integrator
    {
        private Game game;

        //use a fixed time step to get predictable sim
        protected float fixedTimeStep;

        public float FixedTimeStep
        {
            get { return fixedTimeStep; }
            set { fixedTimeStep = value; }
        }

        //---------------------------------------------------------------

        public Integrator(Game game)
        {
            this.game = game;

            //set the fixed time step to target elapsed time (default at 1/60)
            fixedTimeStep = (float)game.TargetElapsedTime.TotalSeconds;
        }
    }
}
```

The `Integrator` abstract class defines a class variable called `fixedTimeStep` which stores a fixed time step value for the simulation. Simulation behavior changes as the time step changes, meaning that a spring, for instance, with the same stiffness and damping constants will behave differently when undergoing a simulation at 30 frames per second and at 60 frames per second. In order to get consistent and predictable simulations, it is better to keep it at a constant value. There are other solutions for this, such as adaptive time-step methods, but we will keep to this simple solution in our tutorial. For the `fixedTimeStep` variable, we will be using the value from `game.TargetElapsedTime` which specifies the desired time step for the game. The default value for this is 1 / 60 seconds.

Derived classes will need to implement their own `Integrate` method, so it is defined as abstract in the class.

(class abstract method in Integrator.cs)

```
        public abstract void Integrate(Vector3 acceleration, SimObject simObject);
```

1.4.2 Forward Euler Integrator

We can now start to discuss about our first numerical integrator, called the Forward Euler integrator. This is one of the most basic integrator that can be implemented.

To find the value of an attribute *y* in the next time step, the Forward Euler integrator basically finds the product of the time step *Δt* and the derivative of *y*, and adds the result to *y* at the current time step:

$$y(t + \Delta t) = y(t) + y'(t)\Delta t \qquad [5]$$

Relating that to our simulation, we have:

$$v(t + \Delta t) = v(t) + a(t)\,\Delta t \qquad [6]$$

This means that the velocity at the next time step is equals to the velocity at the current time added to the product of the time step and acceleration at the current time.

A similar equation exists for the position of the object:

$$x(t + \Delta t) = x(t) + v(t)\,\Delta t \qquad [7]$$

Now, create a new file called ForwardEulerIntegrator.cs.

(ForwardEulerIntegrator.cs)

```
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Integrators
{
    public sealed class ForwardEulerIntegrator : Integrator
    {
        public ForwardEulerIntegrator(Game game)
            : base(game) { }
    }
}
```

This class basically inherits from the `Integrator` abstract class.

We then create an override method named `Integrate`, and use Eqn. 6 and Eqn. 7 given above to obtain the new velocity and position respectively. These new attributes are updated into the given `SimObject`.

(class method in ForwardEulerIntegrator.cs)

```
        public override void Integrate(Vector3 acceleration, SimObject simObject)
        {
            //calculate new position using the velocity at current time
            simObject.CurrPosition += simObject.CurrVelocity * fixedTimeStep;

            //calculate new velocity using the acceleration at current time
            simObject.CurrVelocity += acceleration * fixedTimeStep;
        }
```

1.5 Simulation Class

We now have all the necessary classes that we need to implement the simulation algorithm discussed in Section 1.1. We shall pack the simulation algorithm into a `Simulation` class so that it is easier to create other kinds of simulations in later chapters.

Create a new folder called Simulations in the SoftBody folder, and then add a new class file named Simulation.cs.

(Simulation.cs)

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.ForceGenerators;
using SkeelSoftBodyPhysicsTutorial.SoftBody.Integrators;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Simulations
{
    public class Simulation
    {
        protected Game game;
        protected List<SimObject> simObjects = new List<SimObject>();
        protected List<ForceGenerator> globalForceGenerators = new
List<ForceGenerator>();
```

```
        protected List<Spring> springs = new List<Spring>();
        protected Integrator integrator;

        public List<SimObject> SimObjects
        {
            get { return simObjects; }
            set { simObjects = value; }
        }

        public Integrator Integrator
        {
            get { return integrator; }
            set { integrator = value; }
        }

        //------------------------------------------------------------------

        public Simulation(Game game)
        {
            this.game = game;

            //create a default integrator
            this.integrator = new ForwardEulerIntegrator(game);
        }
    }
}
```

From the algorithm stated in Section 1.1, we know that we have to iterate through lists of simulation objects, forces and also use an integrator. Thus the `Simulation` class keeps track of:

- **A list of `SimObjects`**: These are the objects that are in the simulation system. The active ones have their motion determined by the simulation system, while the passive ones do not.
- **A list of `ForceGenerators`**: These are the global force generators, such as `Gravity` and `Medium`. They will be applied to all `SimObjects` in the system.
- **A list of `Springs`**: These are considered local force generators that will apply forces which act only on the attached `SimObjects`.
- **An integrator**: This is used to integrate the acceleration of the `SimObjects` to obtain positions/velocities for the next time step.

We shall also create some methods to add the `Springs`, `ForceGenerators` and `SimObjects` into the simulation system:

```
(class methods in Simulation.cs)

        public void AddSpring(float stiffness, float damping, SimObject simObjA, 
SimObject simObjB)
        {
            Spring spring = new Spring(stiffness, damping, simObjA, simObjB);
            springs.Add(spring);
        }

        public void AddSimObject(SimObject simObject)
        {
            simObjects.Add(simObject);
        }

        public void AddGlobalForceGenerator(ForceGenerator forceGenerator)
        {
            globalForceGenerators.Add(forceGenerator);
        }
```

Now we create an `Update` method that implements the physics simulation algorithm in a step-by-step fashion.

```
(class method in Simulation.cs)

        public virtual void Update(GameTime gameTime)
        {

        }
```

First, we sum up all the forces. This is done by iterating through each `Spring` to sum up the local spring forces, and then applying the global forces to each `SimObject`.

(inside Update() in Simulation.cs)

```
            //sum all local forces
            foreach (Spring spring in springs)
            {
                spring.ApplyForce(null);  //no need to specify any simObj
            }

            //sum all global forces acting on the objects
            foreach (SimObject simObject in simObjects)
            {
                if (simObject.SimObjectType == SimObjectType.ACTIVE)
                {
                    foreach (ForceGenerator forceGenerator in globalForceGenerators)
                    {
                        forceGenerator.ApplyForce(simObject);
                    }
                }
            }
```

Next, we find the acceleration by dividing the resultant force by the mass of the object. We then ask the assigned integrator to integrate the acceleration.

(outside Update() in Simulation.cs)

```
        Vector3 acceleration;
```

(inside Update() in Simulation.cs)

```
            foreach (SimObject simObject in simObjects)
            {
                if (simObject.SimObjectType == SimObjectType.ACTIVE)
                {
                    //find acceleration
                    acceleration = simObject.ResultantForce / simObject.Mass;

                    //integrate
                    integrator.Integrate(acceleration, simObject);
                }
            }
```

Now we update the object by asking the `SimObject` to update itself.

(inside Update() in Simulation.cs)

```
            //update object
            foreach (SimObject simObject in simObjects)
            {
                simObject.Update(gameTime);
            }
```

Recall that the `SimModel` class implemented `Update` by assigning the position of the `GameModel` to the `CurrPosition` property. This essentially updates the position of the model that we see on the screen according to the newly calculated `CurrPosition`.

Finally, we reset the forces acting on the `SimObjects` so that there are no forces in the accumulative `ResultantForce` property at the start of the next time step.

(inside Update() in Simulation.cs)

```
            //reset forces on sim objects
            foreach (SimObject simObject in simObjects)
            {
                if (simObject.SimObjectType == SimObjectType.ACTIVE)
                {
                    simObject.ResetForces();
                }
            }
```

1.6 Using The New Classes To Create A Spring Simulation

Now let us create the simulation in the main game class by using all the classes that we have created. If you take a look at the Game1.cs provided in the project files, it is pretty much the same as the template version provided automatically by XNA Game Studio when you create a new XNA project. The only difference is that all the unnecessary comments have been removed, and that the necessary components have been added in (refer to the chapter on “Base Codes” for more details).

We shall first create a blank method for initializing the scene:

```
(class members in Game1.cs, addition of code in bold)

        protected override void Initialize()
        {
            InitSpringScene();
            base.Initialize();
        }

        private void InitSpringScene()
        {

        }
```

Next, we will initialize the simulation in the newly created method. We first load in a cube and a sphere. They will serve as markers to indicate both ends of the springs. These models are loaded using the `ModelComponent` from our base codes.

```
(at the top of Game1.cs)

using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

(outside InitSpringScene() in Game1.cs)

        SimModel stationaryCubeSimObj, movingSphereSimObj;

(inside InitSpringScene() in Game1.cs)

            //load in a cube and a sphere
            GameModel stationaryCube = modelComponent.LoadGameModel(@"cube");
            stationaryCube.TranslateY = 5;
            GameModel movingSphere = modelComponent.LoadGameModel(@"sphere");
```

We then create a spring simulation:

```
(at the top of Game1.cs)

using SkeelSoftBodyPhysicsTutorial.SoftBody.Simulations;

(outside InitSpringScene() in Game1.cs)

        Simulation springSim;

(inside InitSpringScene() in Game1.cs)

            //create a spring sim
            springSim = new Simulation(this);
```

Next, we pack the cube and sphere into `SimObjects` and add them into the spring simulation. The cube will be controlled by us, so it will be marked as passive. The sphere on the other hand is to be controlled by the simulation, so it has to be marked as active. The mass of the sphere is set to 1.0 after some trial and error, while the mass of the cube is set to some arbitruary value (1000.0 in this case) but the value does not matter since it is not used in the calculations.

```
(inside InitSpringScene() in Game1.cs)

            //create sim objects from the loaded models
            float sphereMass = 1.0f;
            movingSphereSimObj = new SimModel(movingSphere, sphereMass,
SimObjectType.ACTIVE);
            springSim.AddSimObject(movingSphereSimObj);
            stationaryCubeSimObj = new SimModel(stationaryCube, 1000.0f,
SimObjectType.PASSIVE);
            springSim.AddSimObject(stationaryCubeSimObj);
```

We then create a spring between the two `SimObjects` created:

```
(inside InitSpringScene() in Game1.cs)

            //attach a spring between the sim objects
            float stiffness = 8.0f;
            float damping = 0.1f;
            springSim.AddSpring(stiffness,      damping,      stationaryCubeSimObj,
movingSphereSimObj);
```

Next, we add global forces to the simulation system. We will be using the gravitational acceleration of Earth (9.81 downwards) for the gravity. For the drag coefficient value, a value of 0.5 was deemed to be appropriate after some trial and error.

```
(at the top of Game1.cs)

using SkeelSoftBodyPhysicsTutorial.SoftBody.ForceGenerators;

(inside InitSpringScene() in Game1.cs)

            //add in a global force generator: gravity
            Gravity gravity = new Gravity(new Vector3(0, -9.81f, 0));
            springSim.AddGlobalForceGenerator(gravity);
            //add in a global force generator: air
            float dragCoefficient = 0.5f;
            Medium air = new Medium(dragCoefficient);
            springSim.AddGlobalForceGenerator(air);
```

Now we create an `Integrator` and assign it to the simulation.

(at the top of Game1.cs)

```
using SkeelSoftBodyPhysicsTutorial.SoftBody.Integrators;
```

(inside InitSpringScene() in Game1.cs)

```
            //create an integrator and assign it to the sim
            ForwardEulerIntegrator integrator = new ForwardEulerIntegrator(this);
            springSim.Integrator = integrator;
```

In order to visualize the spring, we shall also draw a line from the cube to the sphere that represents the spring. A line has already been created when we include `Line3DComponent` into `game.Components`, so all we have to do now is to set the positions of the two ends, and also the color of the line.

(inside InitSpringScene() in Game1.cs)

```
            //init the line from cube to sphere that represents the spring
            line3DComponent.StartPosition = stationaryCube.Translate;
            line3DComponent.EndPosition = movingSphere.Translate;
            line3DComponent.Color = Color.White;
```

Now, we need to update the simulation and the line in the `Update` method. All we have to do is to ask the simulation to update itself, and then move the two ends of the line to the new positions calculated for the sphere and cube simulation objects.

(inside Update() in Game1.cs, addition of code in bold)

```
        protected override void Update(GameTime gameTime)
        {
            //update the sim
            springSim.Update(gameTime);

            //update line
            line3DComponent.StartPosition = stationaryCubeSimObj.CurrPosition;
            line3DComponent.EndPosition = movingSphereSimObj.CurrPosition;

            base.Update(gameTime);
        }
```

If you compile and execute the program now, you should have a stationary cube and a moving sphere that is attached via a spring, which means that the simulation that we have coded works!

Let us now do the last bits of coding to allow the user to control the stationary cube.

(class methods in Game1.cs, addition of code in bold)

```
        protected override void Update(GameTime gameTime)
        {
            //poll for input
            HandleInput(gameTime);

            //update the sim
            springSim.Update(gameTime);

            //update line
            line3DComponent.StartPosition = stationaryCubeSimObj.CurrPosition;
            line3DComponent.EndPosition = movingSphereSimObj.CurrPosition;

            base.Update(gameTime);
        }

        private void HandleInput(GameTime gameTime)
        {
            if (inputComponent.IsKeyHeldDown(Keys.Right))
            {
                stationaryCubeSimObj.CurrPositionX += 0.1f;
            }
            if (inputComponent.IsKeyHeldDown(Keys.Left))
            {
                stationaryCubeSimObj.CurrPositionX -= 0.1f;
            }
            if (inputComponent.IsKeyHeldDown(Keys.Up))
            {
                stationaryCubeSimObj.CurrPositionY += 0.1f;
            }
            if (inputComponent.IsKeyHeldDown(Keys.Down))
            {
                stationaryCubeSimObj.CurrPositionY -= 0.1f;
            }
        }
```

You should now have a spring simulation that can be controlled using the keyboard, just like Video 4 that you saw in the beginning of this chapter.

In this chapter, we have looked at how a physics simulation should be done in games. We then created four main groups of classes that help us create these simulations in an object-oriented manner. Finally, we made use of all these classes to create a simple spring simulation.

This chapter is quite long because it covers a lot of basics for soft body simulations. We still have three more chapters to go but do not worry, most of the work has already been done in this chapter. In the next chapter, we shall start to implement a cloth simulation, and it will rely largely on the codes that we have created so far.

Get the source files for the end of this chapter from the SkeelSoftBodyPhysicsTutorial-Chapter1-END folder

Chapter 2 - Cloth Simulation

Get the source files for the start of this chapter from the SkeelSoftBodyPhysicsTutorial-Chapter2-BEGIN folder

In this chapter, we will be creating a simple cloth simulation. A cloth is one of the basic soft bodies that is simple to create and yet produces pleasant results.

Soft bodies, in general, is mostly about how you connect the springs between vertices. Each vertex of the model is able to move independently from others, allowing the model to deform and change its overall shape. Whether the model deforms like a soft body that you expect very much depends on how you connect the springs and the attributes of the springs. Now that we know how to create springs from the previous chapter, we will learn how to connect the springs for the plane geometry in this chapter so that a cloth simulation can be created.

By the end of this chapter, you should be able to create a cloth that looks like this:

CLOTH
CG:SKEELOGY
cg.skeelogy.com
YouTube
0:00 / 0:00

Video 5: *A cloth simulation created using simple springs and constraints*
[Click on image above to watch the video]

2.1 Constraints

Before we get into creating the cloth itself, let us first take a look at constraints and why we need them.

Constraints are basically conditions that the simulation system must try to satisfy. Some examples of constraints are length constraints where two objects try to maintain their distance apart, and point constraints where an object tries to stick to a certain point in space.

In order to understand why length constraints for cloth are so important, I have created a version of the cloth that does not use them. Notice how the whole cloth looks too elastic due to the highly springy behavior, especially at the pinned corners. It is also quite unstable at times.

CLOTH
CG:SKEELOGY
cg.skeelogy.com
NEGATIVE EXAMPLE: NO LENGTH CONSTRAINTS
YouTube
0:00 / 0:00

Video 6: *A cloth simulation without length constraints.*
The cloth tends to be too elastic and quite unstable at times.
[Click on image above to watch the video]

What we really want for a cloth simulation is to have length constraints between the vertices on top of the springs. This will help maintain the distances between the vertices and give a much more realistic behavior for the cloth simulation. We will also want to have point constraints at the corners of the cloth so that the corners can be pinned in space for the user to control.

The way we solve the constraints will be by relaxation, a very practical method that works well for games. We will iterate *n* number of times and at each iteration, we try to get each constraint to satisfy the goal. For example, for a point constraint, we could perhaps do a 10-iteration loop and at every iteration, we move the object towards the goal position as specified by the point constraint.

Why then do we want to iterate *n* times and get the constraint to move towards the goal repeatedly? The reason is that there might be many different constraints to be satisfied and by satisfying one of them, the other constraints might be violated. Imagine a simulation with two connected length constraints. During each iteration, the first constraint moves towards the goal length but by doing so, the goal length of the second constraint might have been increased. The second constraint then moves towards its goal length and in turn might have caused the goal length of the first constraint to increase. This seems like they are countering the effect of each other, but at the end of each iteration, both of them would have moved towards their goal lengths to some extent (convergence is guaranteed under the right conditions). This is the reason why we have to iterate through *n* number of times so that we are slowly relaxing the “fight” between the constraints and they will slowly converge to their goal lengths respectively.

The number of iterations to perform depends on how much processing power you have for your game. The higher the iterations, the better the constraints will be satisfied, but the more expensive it is to calculate. A moderate number of about 10 to 20 often works well to give satisfactory result, so there is usually no need to increase this number to high values like 100. In fact, in some cases, just one iteration might be enough. It is suggested that you reduce this number to a suitable value which gives visually acceptable results.

2.1.1 Constraint Interface

Create a folder called Constraints in the SoftBody folder, and add a new class file named Constraint.cs. The `Constraint` class will basically be an interface that makes it compulsory for derived classes to implement a method called `SatisfyConstraint`.

(Constraint.cs)

```
namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Constraints
{
    public interface Constraint
    {
        void SatisfyConstraint();
    }
}
```

2.1.2 Length Constraint

As mentioned earlier, length constraints are basically constraints that tries to maintain the distance between two objects.

In the Constraints folder, add a new class file named LengthConstraint.cs. A `LengthConstraint` class basically takes in two `SimObjects` and the expected distance between them.

```
(LengthConstraint.cs)

using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Constraints
{
    public sealed class LengthConstraint : Constraint
    {
        private float length;
        private SimObject simObj1;
        private SimObject simObj2;

        public float Length
        {
            get { return length; }
            set { length = value; }
        }

        //------------------------------------------------------------

        public LengthConstraint(float length, SimObject simObj1, SimObject simObj2)
        {
            this.length = length;
            this.simObj1 = simObj1;
            this.simObj2 = simObj2;
        }
    }
}
```

Next, we have to implement the `SatisfyConstraint` method. Recall that we have to satisfy the constraint at each iteration. For a length constraint, this can be done by moving both ends of the constraint by half the difference of the current length and the goal length, as shown in Fig. 2.

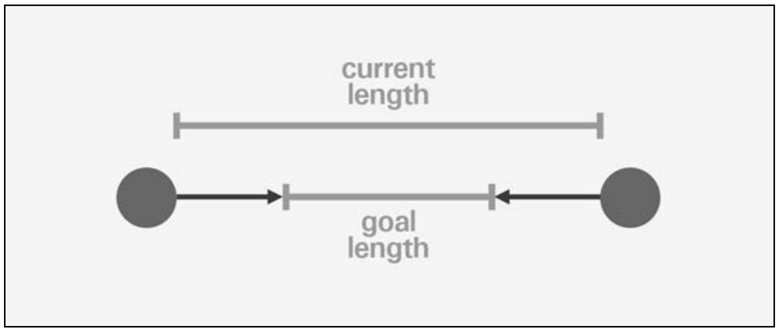


Figure 2: *Two ends of a length constraint moving towards the goal length in a single iteration*

The implementation of that is given below. Basically we calculate the direction and current distance between the two `SimObjects`. Then we normalize the direction vector, and move both `SimObjects` half the difference distance in the converging direction.

```
(class method in LengthConstraint.cs)

        Vector3 direction;
        float currentLength;
        Vector3 moveVector;
        public void SatisfyConstraint()
        {
            //calculate direction
            direction = simObj2.CurrPosition - simObj1.CurrPosition;

            //calculate current length
            currentLength = direction.Length();

            //check for zero vector
            if (direction != Vector3.Zero)
            {
                //normalize direction vector
                direction.Normalize();

                //move to goal positions
                moveVector = 0.5f * (currentLength - length) * direction;
                simObj1.CurrPosition += moveVector;
                simObj2.CurrPosition += -moveVector;
            }
        }
```

2.1.3 Point Constraint

Another useful type of constraint is a point constraint. A point constraint tries to pin an object to a certain point in space. This is useful for constraining the top two corners of our cloth later on. By using point constraints, we are also able to control the two pinned corners by changing their respective constraint position.

In the Constraints folder, create a new class file and call it PointConstraint.cs.

```
(PointConstraint.cs)

using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Constraints
{
    public sealed class PointConstraint : Constraint
    {
        private Vector3 point;
        private SimObject simObject;

        public Vector3 Point
        {
            get { return point; }
            set { point = value; }
        }

        public float PointX
        {
            get { return point.X; }
            set { point.X = value; }
        }

        public float PointY
        {
            get { return point.Y; }
            set { point.Y = value; }
        }

        public float PointZ
        {
            get { return point.Z; }
            set { point.Z = value; }
        }

        public SimObject SimModel
        {
            get { return simObject; }
            set { simObject = value; }
        }

        //------------------------------------------------------
        public PointConstraint(Vector3 point, SimObject simObject)
        {
            this.point = point;
            this.simObject = simObject;
        }
    }
}
```

Now we have to implement the `SatisfyConstraint` method. To satisfy a point constraint, all we have to do is move the object to the goal position.

```
(class method in PointConstraint.cs)

        public void SatisfyConstraint()
        {
            //move to goal position
            simObject.CurrPosition = point;
        }
```

2.1.4 Processing The Constraints In The Simulation Class

Now that we have both the `LengthConstraint` and `PointConstraint` classes created, we need to let the `Simulation` class process them.

First, we add in some variables in Simulation.cs to keep track of the `Constraints`. This consists of a list of `Constraints` and a `constraintIterations` variable that determines how many times to iterate through the `Constraints` per time step.

```
(at the top of Simulation.cs)

using SkeelSoftBodyPhysicsTutorial.SoftBody.Constraints;

(class members in Simulation.cs)

        protected List<Constraint> constraints = new List<Constraint>();
        protected int constraintIterations;

        public List<Constraint> Constraints
        {
            get { return constraints; }
            set { constraints = value; }
        }
        public int ConstraintIterations
        {
            get { return constraintIterations; }
            set { constraintIterations = value; }
        }
```

Then we have to iterate through the constraints to relax them, as described in the beginning of Section 2.1. This relaxation iteration has to be done after the integrator has solved for the new object positions, but before updating the objects.

(class method in Simulation.cs, addition of code in bold)

```
        public virtual void Update(GameTime gameTime)
        {
            //sum all local forces
            foreach (Spring spring in springs)
            {
                spring.ApplyForce(null);  //no need to specify any simObj
            }

            //sum all global forces acting on the objects
            foreach (SimObject simObject in simObjects)
            {
                if (simObject.SimObjectType == SimObjectType.ACTIVE)
                {
                    foreach (ForceGenerator forceGenerator in globalForceGenerators)
                    {
                        forceGenerator.ApplyForce(simObject);
                    }
                }
            }

            foreach (SimObject simObject in simObjects)
            {
                if (simObject.SimObjectType == SimObjectType.ACTIVE)
                {
                    //find acceleration
                    acceleration = simObject.ResultantForce / simObject.Mass;

                    //integrate
                    integrator.Integrate(acceleration, simObject);
                }
            }

            //satisfy constraints
            for (int i = 0; i < constraintIterations; i++)
            {
                foreach (Constraint constraint in constraints)
                {
                    constraint.SatisfyConstraint();
                }
            }
            //update object
            foreach (SimObject simObject in simObjects)
            {
                simObject.Update(gameTime);
            }

            //reset forces on sim objects
            foreach (SimObject simObject in simObjects)
            {
                if (simObject.SimObjectType == SimObjectType.ACTIVE)
                {
                    simObject.ResetForces();
                }
            }
        }
```

2.2 Verlet (No Velocity) Integrator

It is time to introduce another integrator called the Verlet integrator. It is known to be fast and stable (second order, as compared to first order for Forward Euler), and thus has been a popular choice for games. There are several variants of Verlet integrators available, such as Velocity Verlet and also a multi-step method called LeapFrog Verlet. However, the Verlet integrator that we are interested in is a version that totally omits the object's velocity in its calculations.

In the previous section, we have covered how to solve constraints by relaxation. If you have realized, in order to satisfy the constraints, we simply moved the objects towards their goal positions without any consideration for their velocities! This is where the Verlet integrator comes in. By explicitly omitting the velocities in the calculations, we can perform the simple shifting of positions during relaxation without worrying about the velocities, and we will still be able to solve for the new positions at each time step.

The equation for the Verlet integration is as follows:

$$x(t + \Delta t) = 2\, x(t) + x(t - \Delta t) + a(t)\, (\Delta t)^2 \qquad [8]$$

It basically uses the position in the current and previous time steps to determine the position for the next time step.

The above Verlet integration equation can be modified to simulate drag forces in the medium, where *d* is a drag coefficient between 0 and 1 inclusive.

$$x(t + \Delta t) = (2 - d)\, x(t) + (1 - d)\, x(t - \Delta t) + a(t)(\Delta t)^2 \qquad [9]$$

By using Eqn. 9, we can leave out the `Medium` force generator in our simulation later on, since the *d* variable in the equation above is a good substitute for that. We will be using this modified equation in our simulation from now on.

Let us now start implementing this Verlet integrator. In order to differentiate this Verlet integrator from the other variants, we shall name this class `VerletNoVelocityIntegrator`. In the Integrators folder, create a new class file named VerletNoVelocityIntegrator.cs.

```
(VerletNoVelocityIntegrator.cs)

using System;
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Integrators
{
    public sealed class VerletNoVelocityIntegrator : Integrator
    {
        private float drag;

        public float Drag
        {
            get { return drag; }
            set
            {
                if (value < 0 || value > 1)
                {
                    throw new ArgumentException("Air resistance must be between 0
and 1");
                }
                drag = value;
            }
        }

        public VerletNoVelocityIntegrator(Game game) : this(game, 0.005f) { }

        public VerletNoVelocityIntegrator(Game game, float drag)
            : base(game)
        {
            this.drag = drag;
        }
    }
}
```

This class basically takes in a drag value in the constructor. It will be set to 0.005 by default if no value is provided.

We now need to implement the compulsory `Integrate` method, using Eqn. 9.

(class method in VerletNoVelocityIntegrator.cs)

```
        Vector3 newPosition;
        public override void Integrate(Vector3 acceleration, SimObject simObject)
        {
            newPosition = (2 - drag) * simObject.CurrPosition
                - (1 - drag) * simObject.PrevPosition
                + acceleration * fixedTimeStep * fixedTimeStep;

            simObject.PrevPosition = simObject.CurrPosition;
            simObject.CurrPosition = newPosition;
        }
```

2.3 Simulation Vertex

Recall that in the previous chapter, we packed a model into a simulation object called `SimModel` so that the simulation attributes such as position, velocity and mass are included. For a cloth, we are interested in moving its vertices, meaning that the simulation objects are actually the vertices of the cloth itself! We will thus need to implement a new simulation class for the vertices.

In the SimObjects folder, create a new class file named SimVertex.cs.

(SimVertex.cs)

```
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.Primitives;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects
{
    public sealed class SimVertex : SimObject
    {
        private int vertexId;
        private TexturedPrimitive primitive;
        public int VertexId
        {
            get { return vertexId; }
            set { vertexId = value; }
        }

        //----------------------------------------------------------------------

        public SimVertex(float mass, SimObjectType simObjectType, int vertexId,
TexturedPrimitive primitive)
            : base(mass, simObjectType)
        {
            this.vertexId = vertexId;
            this.primitive = primitive;
            this.currPosition = primitive.GetVertexPosition(vertexId);
            this.prevPosition = currPosition;
        }
    }
}
```

The `SimVertex` class takes in the mass of the vertex, the `SimObjectType` (active or passive), the vertex id and the primitive. All these information will allow us to retrieve and set the vertex positions when we need to. The other necessary information will come from the base class `SimObject`.

We will have to implement the `Update` method as well. All we have to do is to just set the vertex position to the newly calculated position.

(class method in SimVertex.cs)

```
        public override void Update(GameTime gameTime)
        {
            primitive.SetVertexPosition(this.vertexId, this.currPosition);
        }
```

2.4 Soft Body and Cloth Simulation Classes

We will now proceed on by creating classes that represent a soft body and a cloth. A cloth is essentially a soft body, so the `ClothSim` class that we will be creating will inherit from the `SoftBodySim` class. Let us first start off by creating the `SoftBodySim` class. In the Simulations folder, create a new class file named SoftBodySim.cs.

```
(SoftBodySim.cs)

using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Simulations
{
    public class SoftBodySim : Simulation
    {
        protected SimVertex[] simVertices;

        public SimVertex[] SimVertices
        {
            get { return simVertices; }
            set { simVertices = value; }
        }

        //-------------------------------------------------------------------

        public SoftBodySim(Game game)
            : base(game) { }
    }
}
```

The `SoftBodySim` class basically inherits from the `Simulation` class, and stores a list of `SimVertex`.

Next, for our `ClothSim` class, create a new class file named ClothSim.cs in the Simulations folder.

```
(ClothSim.cs)

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using SkeelSoftBodyPhysicsTutorial.Primitives;
using SkeelSoftBodyPhysicsTutorial.SoftBody.Constraints;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Simulations
{
    public sealed class ClothSim : SoftBodySim
    {
        private TexturedPlane clothPlane;

        public ClothSim(Game game, TexturedPlane clothPlane, float clothMass,
                        float structStiffness, float structDamping,
                        float shearStiffness, float shearDamping,
                        float bendStiffness, float bendDamping)
            : base(game)
        {
            this.clothPlane = clothPlane;
            //create sim data
            CreateSimVertices(clothPlane, clothMass);

            //connect springs and add constraints
            ConnectSprings(structStiffness, structDamping,
                           shearStiffness, shearDamping,
                           bendStiffness, bendDamping);
        }
    }
}
```

The `ClothSim` class takes in a `TexturedPlane` as the cloth geometry, as well as the attributes for the different types of springs needed to link up the cloth vertices. It also creates the simulation vertices, connects the springs and adds length constraints between the vertices. The two method calls in the constructor have not been implemented yet, and will be discussed now.

The implementation for `CreateSimVertices` is as follows:

```
(class method in ClothSim.cs)

        private void CreateSimVertices(TexturedPlane clothPlane, float clothMass)
        {
            int numVertices = clothPlane.NumVertices;
            float vertexMass = clothMass / numVertices;
            simVertices = new SimVertex[numVertices];
            for (int i = 0; i < numVertices; i++)
            {
                simVertices[i] = new SimVertex(vertexMass, SimObjectType.ACTIVE, i,
clothPlane);
                this.AddSimObject(simVertices[i]);
            }
        }
```

It basically creates a `SimVertex` for each vertex of the plane, and sets it to active so that they will be processed by the simulation.

Before we discuss about the `ConnectSprings` method, we need to talk about how the springs should be connected for the cloth geometry. There are essentially three kinds of springs that are needed for a good cloth simulation:

1. **Structural springs** (Fig. 3): These are placed along each quad edge. They are the basic springs which maintain the inter-distance between the vertices in the horizontal and vertical directions.

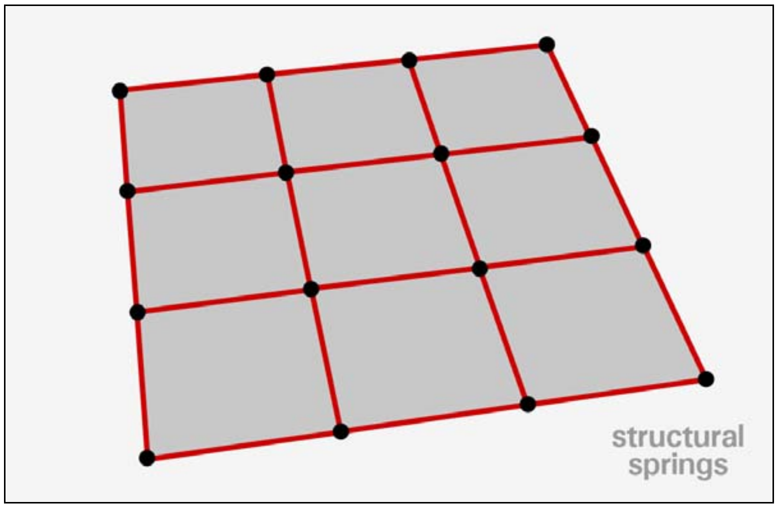


Figure 3: *Structural springs on a plane*

2. **Shear springs** (Fig. 4): These are placed along the diagonals of each quad. They help to maintain the square shape of each quad on the geometry, preventing them from collapsing into flat diamond shapes.

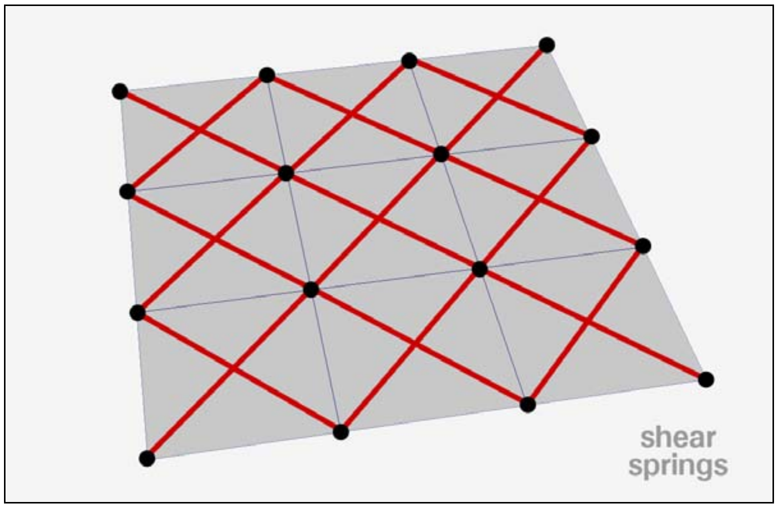


Figure 4: *Shear springs on a plane*

3. **Bend springs** (Fig. 5 and 6): These are placed horizontally and vertically as well, just like structural springs. However, they are placed between vertices that are two vertices apart. They help to control how much the cloth can bend. A high stiffness value for these springs will create a “stiffer” look (e.g. leather) while a low stiffness value will create a softer behavior (e.g. silk).

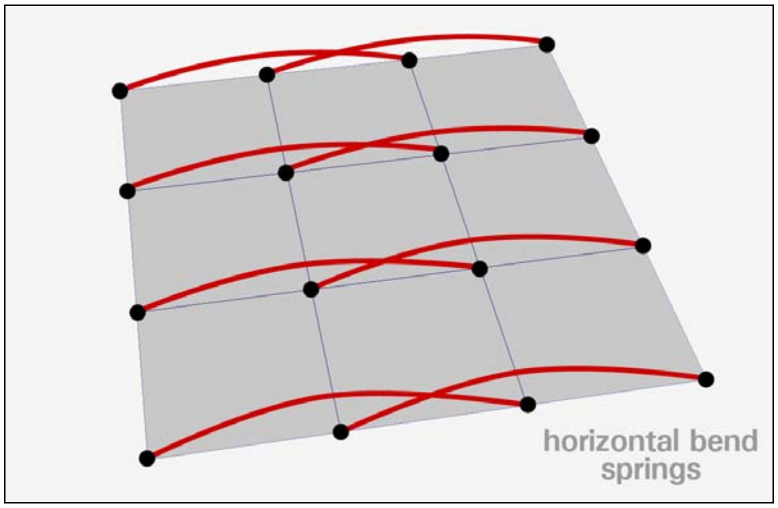


Figure 5: *Horizontal bend springs on a plane*

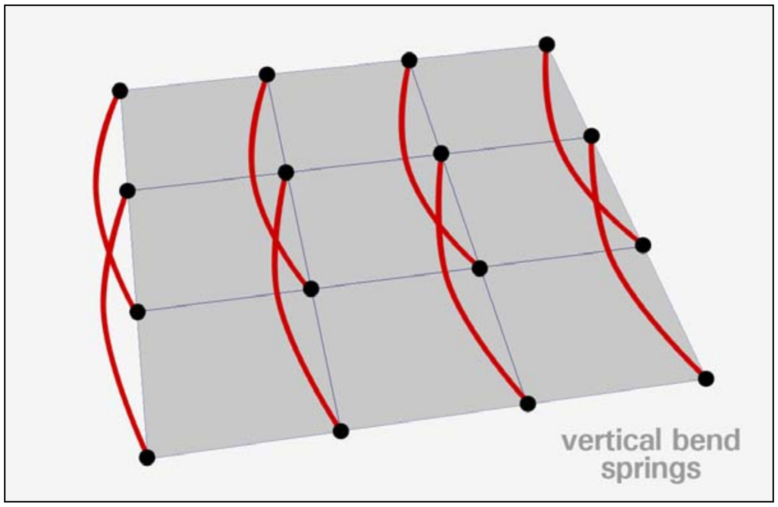
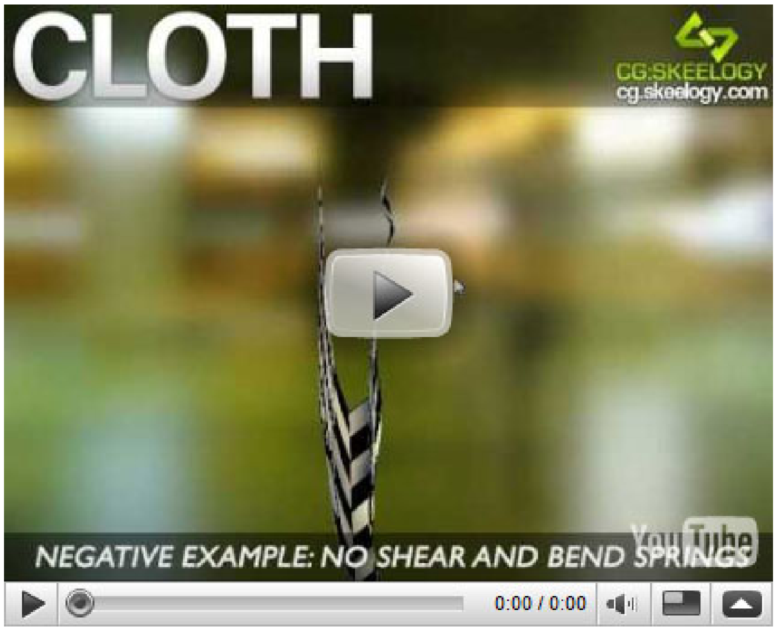
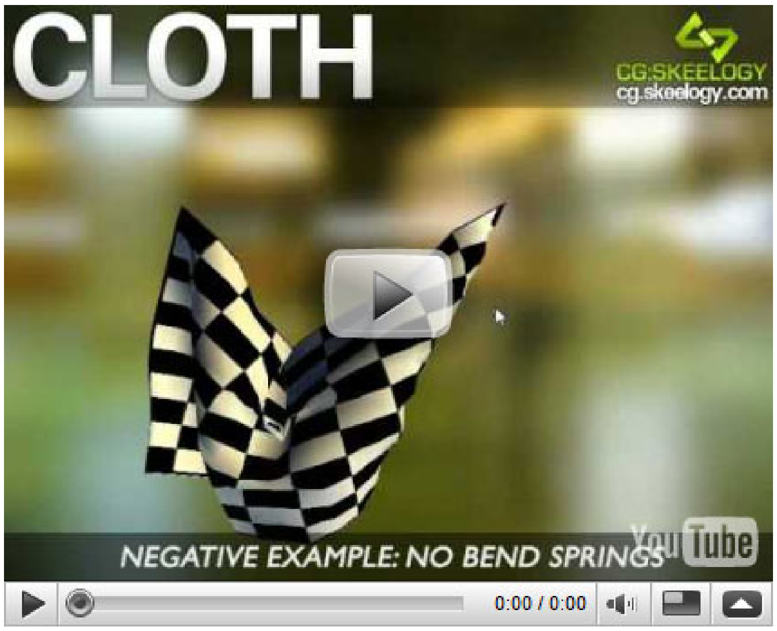


Figure 6: *Vertical bend springs on a plane*

These videos illustrate how the cloth will behave without some of these springs:



Video 7: *Cloth simulation with only structural springs (no shear and bend springs).*
The cloth is unable to maintain a good shape when draping downwards.
[Click on image above to watch the video]



Video 8: *Cloth simulation with structural and shear springs (no bend springs).*
The cloth holds its shape better but is unable to bend realistically and penetrates itself easily.
[Click on image above to watch the video]

You can probably get away with not using the bend springs in certain cases, such as a piece of cloth that is fluttering with the wind, as long as it is visually plausible. This can actually help to save some spring and constraint calculations. We will, however, be using the bend springs in our simulation.

With this new knowledge of how to connect the springs, we can start to implement the `ConnectSprings` method:

(class method in ClothSim.cs)

```
        private void ConnectSprings(float structStiffness, float structDamping,
                                    float shearStiffness, float shearDamping,
                                    float bendStiffness, float bendDamping)
        {
            for (int x = 0; x < clothPlane.LengthSegments; x++)
            {
                for (int y = 0; y <= clothPlane.WidthSegments; y++)
                {
                    //structural spring: horizontal (-)
                    int vertexAId = x + y * (clothPlane.LengthSegments + 1);
                    int vertexBId = (x + 1) + y * (clothPlane.LengthSegments + 1);
                    this.AddSpring(structStiffness, structDamping,
simVertices[vertexAId], simVertices[vertexBId]);
                    float length = (clothPlane.GetVertexPosition(vertexAId) -
clothPlane.GetVertexPosition(vertexBId)).Length();
                    this.Constraints.Add(new LengthConstraint(length,
simVertices[vertexAId], simVertices[vertexBId]));
                }
            }

            for (int x = 0; x <= clothPlane.LengthSegments; x++)
            {
                for (int y = 0; y < clothPlane.WidthSegments; y++)
                {
                    //structural spring: vertical (|)
                    int vertexAId = x + y * (clothPlane.LengthSegments + 1);
                    int vertexBId = x + (y + 1) * (clothPlane.LengthSegments + 1);
                    this.AddSpring(structStiffness, structDamping,
simVertices[vertexAId], simVertices[vertexBId]);
                    float length = (clothPlane.GetVertexPosition(vertexAId) -
clothPlane.GetVertexPosition(vertexBId)).Length();
                    this.Constraints.Add(new LengthConstraint(length,
simVertices[vertexAId], simVertices[vertexBId]));
                }
            }

            for (int x = 0; x < clothPlane.LengthSegments; x++)
            {
                for (int y = 0; y < clothPlane.WidthSegments; y++)
                {
                    //shear spring: diagonal (/)
                    int vertexAId = (x + 1) + y * (clothPlane.LengthSegments + 1);
                    int vertexBId = x + (y + 1) * (clothPlane.LengthSegments + 1);
                    this.AddSpring(shearStiffness, shearDamping,
simVertices[vertexAId], simVertices[vertexBId]);
                    float length = (clothPlane.GetVertexPosition(vertexAId) -
clothPlane.GetVertexPosition(vertexBId)).Length();
                    this.Constraints.Add(new LengthConstraint(length,
simVertices[vertexAId], simVertices[vertexBId]));

                    //shear spring: diagonal (\)
                    vertexAId = x + y * (clothPlane.LengthSegments + 1);
                    vertexBId = (x + 1) + (y + 1) * (clothPlane.LengthSegments + 1);
                    this.AddSpring(shearStiffness, shearDamping,
simVertices[vertexAId], simVertices[vertexBId]);
                    length = (clothPlane.GetVertexPosition(vertexAId) -
clothPlane.GetVertexPosition(vertexBId)).Length();
                    this.Constraints.Add(new LengthConstraint(length,
simVertices[vertexAId], simVertices[vertexBId]));
                }
            }

            for (int x = 0; x < clothPlane.LengthSegments - 1; x++)
            {
                for (int y = 0; y <= clothPlane.WidthSegments; y++)
                {
                    //bend spring: horizontal (--)
                    int vertexAId = x + y * (clothPlane.LengthSegments + 1);
                    int vertexBId = (x + 2) + y * (clothPlane.LengthSegments + 1);
                    this.AddSpring(bendStiffness, bendDamping,
simVertices[vertexAId], simVertices[vertexBId]);
                    float length = (clothPlane.GetVertexPosition(vertexAId) -
clothPlane.GetVertexPosition(vertexBId)).Length();
                    this.Constraints.Add(new LengthConstraint(length,
simVertices[vertexAId], simVertices[vertexBId]));
                }
            }

            for (int x = 0; x <= clothPlane.LengthSegments; x++)
            {
                for (int y = 0; y < clothPlane.WidthSegments - 1; y++)
                {
                    //bend spring: vertical (||)
                    int vertexAId = x + y * (clothPlane.LengthSegments + 1);
                    int vertexBId = x + (y + 2) * (clothPlane.LengthSegments + 1);
                    this.AddSpring(bendStiffness, bendDamping,
simVertices[vertexAId], simVertices[vertexBId]);
                    float length = (clothPlane.GetVertexPosition(vertexAId) -
clothPlane.GetVertexPosition(vertexBId)).Length();
                    this.Constraints.Add(new LengthConstraint(length,
simVertices[vertexAId], simVertices[vertexBId]));
                }
            }
        }
```

That is quite a lot of code, but most of the codes are actually repetitive. The structural springs are created first, followed by the shear springs and then the bend springs. Length constraints are created in the same loops.

Now that we have the springs connected, we need to update the cloth geometry in the `Update` method:

(class method in ClothSim.cs)

```
        public override void Update(GameTime gameTime)
        {
            //call base.Update() to update the vertex positions
            base.Update(gameTime);

            //recalculate the vertex normals
            clothPlane.RecalculateNormals();

            //commit the vertex position and normal changes
            clothPlane.CommitChanges();
        }
```

We first call the base class to update itself first, which will calculate all the new positions of the vertices and set the cloth vertices to those new positions. Since the vertices move independently from one another, their relative positions change over time and their normals will change as well. Thus, we need to recalculate the normals so that the rendering is faithful to the geometry changes. Note that `RecalculateNormals` has already been implemented in the base class. Finally, we need to call `CommitChanges` on the plane geometry which will update all the vertex position changes in the video RAM.

2.5 Using The New Classes To Create A Cloth Simulation

We are now ready to create the cloth in our main program file Game1.cs.

First, we initialize the scene.

```
(class method in Game1.cs, additional codes in bold)

        protected override void Initialize()
        {
            InitClothScene();
            base.Initialize();
        }

        private void InitClothScene()
        {

        }
```

We define the cloth attributes and create a plane as the cloth geometry.

```
(at the top of Game1.cs)

using SkeelSoftBodyPhysicsTutorial.Primitives;

(outside InitClothScene()in Game1.cs)

         TexturedPlane clothPlane;

(inside InitClothScene()in Game1.cs)

            //cloth attributes
            int length = 10;
            int width = 5;
            int lengthSegments = 20;
            int widthSegments = 15;

            //load in a plane
            clothPlane = new TexturedPlane(this, length, width, lengthSegments,
                                           widthSegments, @"checkerboard");
            clothPlane.Initialize();
```

In the Solution Explorer, notice that there is a folder called Primitives that contains classes that you can use to create textured planes and cylinders. I will not be explaining the codes in these classes but do note that with these classes, you will be able to create primitives and also get/set the vertex attributes such as the position and normal. If you are going to use your own primitive classes or load your own models from files, you will also need to have methods to get/set the vertex attributes.

We then define the mass of the cloth, attributes of the different springs for the cloth, and create a `ClothSim` instance:

(at the top of Game1.cs)

```
using SkeelSoftBodyPhysicsTutorial.SoftBody.Simulations;
```

(outside InitClothScene()in Game1.cs)

```
        ClothSim clothSim;
```

(inside InitClothScene()in Game1.cs)

```
            //create a cloth sim
            float clothMass = 2.0f;
            float structStiffness = 2.0f;
            float structDamping = 0.02f;
            float shearStiffness = 2.0f;
            float shearDamping = 0.02f;
            float bendStiffness = 2.0f;
            float bendDamping = 0.02f;
            clothSim = new ClothSim(this, clothPlane, clothMass, structStiffness,
                                    structDamping, shearStiffness, shearDamping,
                                    bendStiffness, bendDamping);
            clothSim.ConstraintIterations = 10;
```

Next, we create the global force generators. For our cloth simulation, this will only be gravity. Note that we will not be creating a `Medium` force generator since the drag forces will be simulated in the Verlet integrator.

(at the top of Game1.cs)

```
using SkeelSoftBodyPhysicsTutorial.SoftBody.ForceGenerators;
```

(inside InitClothScene()in Game1.cs)

```
            //add in a global forceGenerators: gravity
            Gravity gravity = new Gravity(new Vector3(0, -9.81f, 0));
            clothSim.AddGlobalForceGenerator(gravity);
```

We will constrain the top left and top right corners of the plane in order to control them later on. This is done by using `PointConstraints`.

(at the top of Game1.cs)

```
using SkeelSoftBodyPhysicsTutorial.SoftBody.Constraints;
```

```
(outside InitClothScene()in Game1.cs)

        PointConstraint topLeftCorner, topRightCorner;

(inside InitClothScene()in Game1.cs)

            //constrain the two top corners of the cloth so that we can control it
            topLeftCorner = new
PointConstraint(clothSim.SimVertices[0].CurrPosition, clothSim.SimVertices[0]);
            clothSim.Constraints.Add(topLeftCorner);
            topRightCorner = new
PointConstraint(clothSim.SimVertices[lengthSegments].CurrPosition,
clothSim.SimVertices[lengthSegments]);
            clothSim.Constraints.Add(topRightCorner);
```

We then create a Verlet integrator, with an associated drag value.

```
(at the top of Game1.cs)

using SkeelSoftBodyPhysicsTutorial.SoftBody.Integrators;

(inside InitClothScene()in Game1.cs)

            //create an integrator and assign it to the sim
            float drag = 0.01f;
            Integrator integrator = new VerletNoVelocityIntegrator(this, drag);
            clothSim.Integrator = integrator;
```

There are a few more steps regarding our cloth plane before we can execute the program. We need to call its `LoadContent` method so that it can load the necessary contents.

```
(class method, additional code in bold)

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);
            clothPlane.LoadContent();
        }
```

Next, we have to ask the cloth simulation to update at every time step.

(class method, additional code in bold)

```
        protected override void Update(GameTime gameTime)
        {
            //update the sim
            clothSim.Update(gameTime);

            base.Update(gameTime);
        }
```

Finally, we must draw the cloth at every time step so that it is visible on the screen. We shall also set the culling mode to none. The default rendering state is set to cull away the back faces of a model, which will make one side of the cloth invisible although it may be facing the camera. Once we set it to none, we will be able to see both sides of the plane.

(class method, additional code in bold)

```
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);
            GraphicsDevice.RenderState.CullMode = CullMode.None;
            base.Draw(gameTime);
            clothPlane.Draw(gameTime);
        }
```

If you compile and execute the program now, you should see a cloth draping down, with the top two corners pinned.

As usual, we will take a step further and allow the user to control the simulation. We shall use the mouse to control the two top corners of the cloth. The left button will be for moving the top left corner while the right button will be for moving the top right corner.

(class members, additional codes in bold)

```
        protected override void Update(GameTime gameTime)
        {
            //poll for input
            HandleInput(gameTime);

            //update the sim
            clothSim.Update(gameTime);

            base.Update(gameTime);
        }

        private void HandleInput(GameTime gameTime)
        {
            if (inputComponent.IsMouseHeldDown(MouseButtonType.Left))
            {
                topLeftCorner.PointX += -inputComponent.MouseMoved.X / 40;
                topLeftCorner.PointY += inputComponent.MouseMoved.Y / 40;
            }
            if (inputComponent.IsMouseHeldDown(MouseButtonType.Right))
            {
                topRightCorner.PointX += -inputComponent.MouseMoved.X / 40;
                topRightCorner.PointY += inputComponent.MouseMoved.Y / 40;
            }
        }
```

Since the screen space and the world space exist in different scales, we have to divide the `MouseMoved` property by a certain scale value. A suitable value has been found to be 40 after some trial and error.

Compile and execute the program and you should get a controllable cloth simulation just like Video 5 that you saw at the start of this chapter!

In this chapter, we have touched on a few important concepts related to cloth simulation. We have discussed about constraints, why we need them and provided an implementation that uses the relaxation method. Next we implemented the Verlet integrator that works hand in hand with the constraints that we have, since there is no velocity used or stored in the Verlet integrator. We then started to implement the `SimVertex` class, the `SoftBodySim` class and the `ClothSim` class, all of which are built on top of what we have created in the previous chapter. Finally, we created an instance of the cloth simulation.

For our cloth simulation, we have dealt with a type of soft body that involves connecting springs between neighbouring vertices to maintain the spatial distance between the vertices. In the next chapter, we shall look at another way of connecting the springs to create a different kind of soft body.

Get the source files for the end of this chapter from the SkeelSoftBodyPhysicsTutorial-Chapter2-END folder

Chapter 3 - Goal Soft Bodies: Slinky Simulation

Get the source files for the start of this chapter from the SkeelSoftBodyPhysicsTutorial-Chapter3-BEGIN folder

In this chapter, we will be focusing on the creation of a kind of soft body that I will be referring to as a goal soft body. The basic idea is that each vertex on the geometry has a goal position that it should move towards, and this is done by attaching a spring from the current position to the goal position. With a variety of spring stiffness, we will actually get a pretty decent soft body effect.

At the end of this chapter, you will be able to create a pseudo slinky simulation like this:

SLINKY
CG:SKEELOGY
cg.skeelogy.com
YouTube
0:00 / 0:00

Video 9: *A fake slinky simulation using a goal soft body*
[Click on image above to watch the video]

The slinky that you see is actually just a simple cylindrical goal soft body with an alpha texture map. This chapter also illustrates the fact that there are always tricks to help reduce calculations. It may not be physically accurate but as long as it looks believable and it runs fast, it is suitable for games.

3.1 Goal Soft Body Simulation Class

Let us start off by first looking at how we will go about creating the goal soft body. Referring to Fig. 7, we can see that there are actually two objects: a goal object and a soft body object. We attach a spring between each vertex of the soft body and the corresponding vertex of the goal object. As the goal object transforms over time, the vertices of the soft body will be pulled along, giving us the soft body effect that we need.

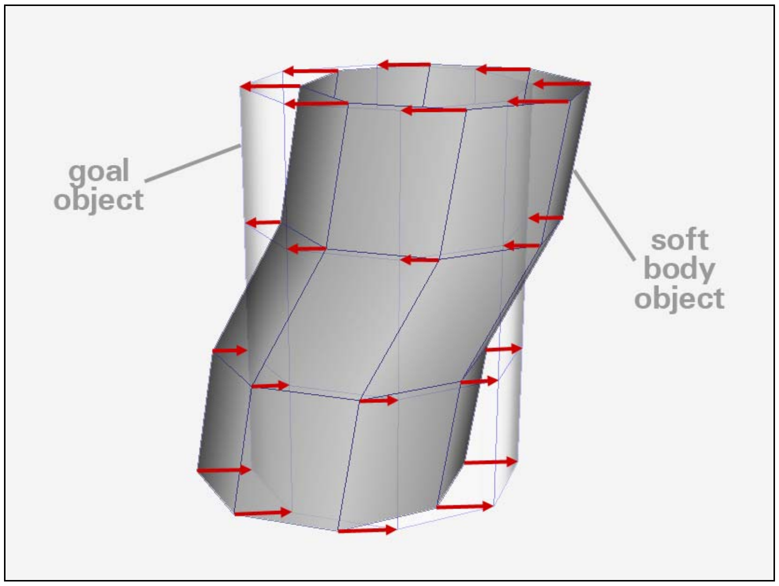


Figure 7: *The main idea behind a goal soft body.*
Each vertex has a goal position to move towards.

This is analogous to the way soft bodies are created in 3D packages like Maya, where there is a goal object and a soft body object which follows it on a per-vertex basis. We will simply create the same object twice, with one of them being the soft body object and the other one being the goal object.

Let us now create the `GoalSoftBodySim` class. In the Simulations folder, create a new class file named GoalSoftBodySim.cs.

```
(GoalSoftBodySim.cs)

using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.Primitives;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Simulations
{
    public sealed class GoalSoftBodySim : SoftBodySim
    {
        private TexturedPrimitive softBodyObject;
        private TexturedPrimitive softBodyGoalObject;

        public GoalSoftBodySim(Game game, TexturedPrimitive softBodyObject,
TexturedPrimitive softBodyGoalObject, float mass, float stiffness, float damping)
            : base(game)
        {
            this.softBodyObject = softBodyObject;
            this.softBodyGoalObject = softBodyGoalObject;

            //create sim vertices on both the goal and non-goal object
            CreateSimVertices(softBodyObject, mass);
            CreateSimVerticesForGoal(softBodyGoalObject, mass);

            //connect springs between the vertices of the goal and non-goal object
            ConnectSprings(stiffness, damping);
        }
    }
}
```

This class basically takes in two `TexturedPrimitives` (one as the goal object, one as the soft body) and the necessary spring attributes. It then creates simulation vertices for both of these objects and connects springs between them. Constraints will not be used since we want the springs to really be springy for this type of soft body.

We now implement the `CreateSimVertices` method. This method initializes the `SimVertex` array, creates a new active `SimVertex` for each slot in the array, and then adds the newly created `SimVertex` to our list of `SimObjects`. This is the same as how we create the simulation vertices for our cloth simulation in the previous chapter.

```
(class method in GoalSoftBodySim.cs)

        private void CreateSimVertices(TexturedPrimitive softBodyObject, float mass)
        {
            int numVertices = softBodyObject.NumVertices;
            float vertexMass = mass / numVertices;
            simVertices = new SimVertex[numVertices];
            for (int i = 0; i < numVertices; i++)
            {
                simVertices[i] = new SimVertex(vertexMass, SimObjectType.ACTIVE, i,
softBodyObject);
                this.AddSimObject(simVertices[i]);
            }
        }
```

The implementation for `CreateSimVerticesForGoal` is similar. The only difference is that we are creating another array to store this list of goal `SimVertex`s, and that these `SimVertex`s are set as passive.

```
(class member in GoalSoftBodySim.cs)

        private SimVertex[] simVerticesForGoal;

(class method in GoalSoftBodySim.cs)

        private void CreateSimVerticesForGoal(TexturedPrimitive softBodyObject,
float mass)
        {
            int numVertices = softBodyObject.NumVertices;
            float vertexMass = mass / numVertices;
            simVerticesForGoal = new SimVertex[numVertices];
            for (int i = 0; i < numVertices; i++)
            {
                simVerticesForGoal[i] = new SimVertex(vertexMass,
SimObjectType.PASSIVE, i, softBodyObject);
                this.AddSimObject(simVerticesForGoal[i]);
            }
        }
```

Next, we connect the springs from the soft body to the goal object by implementing the `CreateSprings` method. Note that we have only passed in one stiffness for all the springs, which will not give a decent soft body result. What we will do here is to vary the stiffness of the springs that we are creating based on the single stiffness specified. We want to start from half the stiffness for the first spring, and increasing it gradually to the full stiffness for the last spring. This is done by first finding the `increment` value and then increasing the stiffness of each subsequent spring by that amount. We will be using the same damping value for all the springs though.

(class method in GoalSoftBodySim.cs)

```
        private void ConnectSprings(float stiffness, float damping)
        {
            //find the increment step for each subsequent spring
            float increment = (0.5f * stiffness) / softBodyObject.NumVertices;

            for (int i = 0; i < softBodyObject.NumVertices; i++)
            {
                //create a gradient stiffness from 0.5 stiffness to 1.0 stiffness
                float thisStiffness = (increment * i) + 0.5f * stiffness;
                this.AddSpring(thisStiffness, damping, simVertices[i],
simVerticesForGoal[i]);
            }
        }
```

Finally, we override the `Update` method to update the normals on the geometry and commit the vertex position/normal changes to the video RAM, just like what we did in the last chapter.

(class method in GoalSoftBodySim.cs)

```
        public override void Update(GameTime gameTime)
        {
            //call base.Update() to update the vertex positions
            base.Update(gameTime);

            //recalculate the vertex normals
            softBodyObject.RecalculateNormals();

            //commit the vertex position and normal changes
            softBodyObject.CommitChanges();
        }
```

3.2 Creating A Goal Soft Body

Now we are ready to create the goal soft body in Game1.cs. First, we prepare a method to initialize the scene.

```
(class method in Game1.cs, addition of codes in bold)

        protected override void Initialize()
        {
            InitGoalSoftBodyScene();
            base.Initialize();
        }

        private void InitGoalSoftBodyScene()
        {

        }
```

We then define some attributes for a cylinder, and create a `TexturedCylinder` using the primitive classes provided. We will texture the cylinder with a checkerboard texture.

```
(outside InitGoalSoftBodyScene() in Game1.cs)

        TexturedCylinder cylinder;

(inside InitGoalSoftBodyScene() in Game1.cs)

            //cylinder attribute
            float length = 10.0f;
            float radius = 3.0f;
            int lengthSegments = 30;
            int radialSegments = 50;

            //load in the cylinder
            cylinder = new TexturedCylinder(this, length, radius, lengthSegments,
radialSegments, "checkerboard");
            cylinder.Initialize();
```

We then load in another `TexturedCylinder` as the goal object.

```
(outside InitGoalSoftBodyScene() in Game1.cs)

        TexturedCylinder goalCylinder;

(inside InitGoalSoftBodyScene() in Game1.cs)

            //load in another same cylinder as the goal
            goalCylinder = new TexturedCylinder(this, length, radius,
                                        lengthSegments, radialSegments, "checkerboard");
            goalCylinder.Initialize();
```

We then create the goal soft body simulation using the `GoalSoftBodySim` class that we have created in the previous section.

(outside InitGoalSoftBodyScene() in Game1.cs)

```
        GoalSoftBodySim goalSoftBodySim;
```

(inside InitGoalSoftBodyScene() in Game1.cs)

```
            //create a goal soft body sim
            float mass = 0.1f;
            float stiffness = 0.03f;
            float damping = 0.0f;
            goalSoftBodySim  =  new  GoalSoftBodySim(this,  cylinder,  goalCylinder,
                                                    mass, stiffness, damping);
```

For our demo, we want to have a force that can push the soft body around while the goal object stays stationary. We will do this using a `Gravity` force generator. We will set the acceleration to the zero vector first, and will change it according to how the user uses the mouse to interact with the scene later on. Logically speaking, we should actually create another class called `Push` but the implementation will be exactly the same as the `Gravity` force generator, so we shall not go through that trouble.

(outside InitGoalSoftBodyScene() in Game1.cs)

```
        Gravity push;
```

(inside InitGoalSoftBodyScene() in Game1.cs)

```
            //create a force generator that we will use to push the cylinder later
            push = new Gravity(Vector3.Zero);
            goalSoftBodySim.AddGlobalForceGenerator(push);
```

As usual, we create an integrator and assign it to the simulation.

(inside InitGoalSoftBodyScene() in Game1.cs)

```
            //create an integrator and assign it to the sim
            float drag = 0.05f;
            Integrator integrator = new VerletNoVelocityIntegrator(this, drag);
            goalSoftBodySim.Integrator = integrator;
```

Next, we need to ask both the goal object and soft body object to load their contents.

(class method in Game1.cs, addition of codes in bold)

```
        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);
            cylinder.LoadContent();
            goalCylinder.LoadContent();
        }
```

In the existing `Update` method, we have to poll for input, and also ask the goal soft body simulation to update itself.

(class method in Game1.cs, addition of codes in bold)

```
        protected override void Update(GameTime gameTime)
        {
            //poll for input
            HandleInput(gameTime);

            //update the simulation
            goalSoftBodySim.Update(gameTime);

            base.Update(gameTime);
        }
```

We want the user to use the mouse to interact with the soft body. The user will click and drag the left mouse button in the scene. Once he releases, a force will be applied on the soft body object by setting the acceleration of the `Gravity` force generator, and this magnitude will be based on how far he has dragged the mouse across the screen.

(class method in Game1.cs)

```
        Vector2 startPos, endPos;
        private void HandleInput(GameTime gameTime)
        {
            if (inputComponent.IsMouseJustDown(MouseButtonType.Left))
            {
                //mouse is just down, so store the starting position
                startPos = inputComponent.MousePosition;
            }
            else if (inputComponent.IsMouseJustUp(MouseButtonType.Left))
            {
                //mouse has just been released, so find the ending position
                endPos = inputComponent.MousePosition;

                //use distance mouse moved as the push acceleration
                push.AccelerationX = (endPos.X - startPos.X) * 3;
                push.AccelerationY = -(endPos.Y - startPos.Y) * 3;

                return;
            }

            //set the push acceleration to zero
            push.Acceleration = Vector3.Zero;
        }
```

Finally, we draw the cylinder. The culling mode is set to none so that we can see both sides of the cylinder.

(class method in Game1.cs, addition of codes in bold)

```
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);
            GraphicsDevice.RenderState.CullMode = CullMode.None;
            base.Draw(gameTime);
            cylinder.Draw(gameTime);
        }
```

You should now have a soft body simulation on a cylinder where the user can exert forces on it by clicking and dragging across the screen. This could work well for the neck or the arm of a flabby character where the fats jiggle like soft bodies.

SLINKY
CG:SKEELOGY
cg.skeelogy.com
YouTube
0:00 / 0:00

Video 10: *A goal soft body simulation*
[Click on image above to watch the video]

Let us try to make this demo slightly more presentable. We shall create an object that is cylindrical and has spring-like motion: a slinky. This can be done using a simple texture map with alpha. In the `InitGoalSoftBodyScene` method, change the texture of the cylinder to “slinky”. This texture is provided in the Content folder.

(inside InitGoalSoftBodyScene() in Game1.cs, code changes shown in bold)

```
            //load in the cylinder
            cylinder = new TexturedCylinder(this, length, radius, lengthSegments,
radialSegments, "slinky");
            cylinder.Initialize();
```

That’s it! You now have a pseudo slinky simulation that is based on a goal soft body, just like Video 9 that you saw at the beginning of the chapter.

In this chapter, we have discussed the idea behind a goal soft body and created a class to handle the spring connections. We have applied the goal soft body simulation on a cylinder, which made it look like some kind of flabby neck or arm of a fat character. We then took a step further to create a fake slinky simulation simply by changing the texture on the cylinder. This was done to show how simple tricks can be used to mimic behavior which might be costly to process in real time.

In the next chapter, we will see how we can create another pseudo simulation that looks like a rigid body simulation with collision handling!

Get the source files for the end of this chapter from the SkeelSoftBodyPhysicsTutorial-Chapter3-END folder

Chapter 4 – Faking Rigid Bodies: Chain Simulation

Get the source files for the start of this chapter from the SkeelSoftBodyPhysicsTutorial-Chapter4-BEGIN folder

In order to illustrate the usefulness of soft body simulations, we will be using a 1D soft body simulator to create a pseudo rigid body simulation in this chapter. It will comprise of a series of nodes arranged vertically and these nodes will be connected by springs. A chain segment model will be used as each of the nodes, thus giving the appearance of a long chain. Since the distances between the nodes will be restricted by length constraints, the chain will look like it is undergoing some kind of collision detection to maintain those distances.

By the end of this chapter, you should be able to create a chain simulation like this:

Video 11: *A chain simulation created using a 1D soft body simulator*
[Click on image above to watch the video]

4.1 Chain Simulation Class

We shall begin by first creating the `ChainSim` class. In the Simulations folder, create a new class file named ChainSim.cs.

(ChainSim.cs)

```
using System;
using Microsoft.Xna.Framework;
using SkeelSoftBodyPhysicsTutorial.Main;
using SkeelSoftBodyPhysicsTutorial.SoftBody.Constraints;
using SkeelSoftBodyPhysicsTutorial.SoftBody.SimObjects;

namespace SkeelSoftBodyPhysicsTutorial.SoftBody.Simulations
{
    public class ChainSim : Simulation
    {
        public ChainSim(Game game, GameModel[] ropeSegmentModels, float totalMass,
float stiffness, float damping)
            : base(game)
        {
            //create sim data
            float sphereMass = totalMass / ropeSegmentModels.Length;
            SimModel[] simObjs = new SimModel[ropeSegmentModels.Length];
            for (int i = 0; i < ropeSegmentModels.Length; i++)
            {
                simObjs[i] = new SimModel(ropeSegmentModels[i], sphereMass,
SimObjectType.ACTIVE);
                this.AddSimObject(simObjs[i]);
            }

            //attach springs between the sim objects
            for (int i = 1; i < ropeSegmentModels.Length; i++)
            {
                this.AddSpring(stiffness, damping, simObjs[i - 1], simObjs[i]);
                this.Constraints.Add(new LengthConstraint((simObjs[i -
1].Model.Translate - simObjs[i].Model.Translate).Length(), simObjs[i - 1],
simObjs[i]));
            }
        }
    }
}
```

We first create an active `SimModel` for each of the chain segments passed in, and then attach springs with constraints between the `SimModel` objects. This is pretty much the same as what how we have been creating our `Simulation` classes so far.

Next, we have to override the `Update` method.

(class method in ChainSim.cs)

```
        Vector3 currToChildVector;
        float angle;
        Vector3 rotAxis;
        public override void Update(GameTime gameTime)
        {
            //call base to perform simulation
            base.Update(gameTime);

            //orient rope segments so that they point to immediate child
            for (int i = 0; i < simObjects.Count - 1; i++)
            {
                //get vector that points to child
                currToChildVector = simObjects[i + 1].CurrPosition - 
simObjects[i].CurrPosition;
                if (currToChildVector != Vector3.Zero)
                {
                    //normalize
                    currToChildVector.Normalize();

                    //find out angle to rotate
                    angle = (float)Math.Acos(Vector3.Dot(Vector3.UnitY, -
currToChildVector));
                    angle = MathHelper.ToDegrees(angle);

                    //find out axis to rotate about
                    rotAxis = Vector3.Cross(Vector3.UnitY, -currToChildVector);
                    if (rotAxis != Vector3.Zero) rotAxis.Normalize();

                    //need to just rotate model about the zaxis for our case
                    ((SimModel)simObjects[i]).Model.RotateZ = rotAxis.Z * angle;
                }
            }

            //make the orientation of the last segment follow the parent segment
            ((SimModel)simObjects[simObjects.Count - 1]).Model.RotateZ = 
((SimModel)simObjects[simObjects.Count - 2]).Model.RotateZ;
        }
```

As usual, we need to first ask the base class to update itself, which will perform the whole simulation and provide us with the new positions. We will then rotate the chain segments so that they point to their immediate child. To do that, we need to find out the rotation angle and the rotation axis. The angle can be found by taking the dot product of the Y axis and the vector from the child to the current segment. The rotation axis, on the other hand, can be found by taking the corresponding cross product. This axis will give us the direction of rotation (clockwise or anti-clockwise) when given the angle. For our program, we will be restricting the interaction to the XY plane, thus we will only need to rotate the chain segments along the Z axis to give a plausible rigid body simulation. The last segment of the chain has no child to point to, thus we will just let its orientation follow that of its parent.

4.2 Creating A Chain Simulation

We are now ready to create our chain simulation in Game1.cs. As usual, we create a method to initialize the scene.

```
(class methods in Game1.cs, addition of codes in bold)

        protected override void Initialize()
        {
            InitChainScene();
            base.Initialize();
        }

        private void InitChainScene()
        {

        }
```

We then define the chain attributes and load in models that will represent each chain segment. The model that we will be loading in is chainSegment.fbx, which is provided in the Content folder. While loading in each segment, we scale and position them. We also rotate the even segments by 90 degrees in the Y axis to form an interlocking chain.

```
(inside InitChainScene() in Game1.cs)

            //chain attributes
            int numSegments = 30;
            float separation = 0.65f;

            //load in segments
            GameModel[] segments = new GameModel[numSegments];
            for (int i = 0; i < numSegments; i++)
            {
                segments[i] = modelComponent.LoadGameModel("chainSegment");

                //reduce the size of each segment
                segments[i].Scale = 0.3f * Vector3.One;

                //position each segment downwards incrementally to form the chain
                segments[i].TranslateY = -i * separation;

                //rotate the even segments by 90 degrees in Y axis
                segments[i].RotateY = (i % 2) * 90.0f;
            }
```

Next, we create a chain simulation.

```
(outside InitChainScene() in Game1.cs)

        ChainSim chainSim;

(inside InitChainScene() in Game1.cs)

            //create a chain sim
            float totalMass = 1.0f;
            float stiffness = 0.3f;
            float damping = 0.01f;
            chainSim = new ChainSim(this, segments, totalMass, stiffness, damping);
```

We then add in a `Gravity` as a global force generator.

```
(inside InitChainScene() in Game1.cs)

            //add in a global force generator: gravity
            Gravity gravity = new Gravity(new Vector3(0, -9.81f * 5, 0));
            chainSim.AddGlobalForceGenerator(gravity);
```

In order to control the chain, we need to constrain the head of the chain. We will thus need to create a `PointConstraint` for the head chain segment.

```
(inside InitChainScene() in Game1.cs)

        PointConstraint chainHeadPoint;

(inside InitChainScene() in Game1.cs)

            //constrain the head point so that we can control it
            chainHeadPoint = new PointConstraint(Vector3.Zero,
chainSim.SimObjects[0]);
            chainSim.Constraints.Add(chainHeadPoint);
```

We then create an integrator for the simulation system as usual.

(inside InitChainScene() in Game1.cs)

```
            //create a integrator and assign it to the sim
            float drag = 0.005f;
            Integrator integrator = new VerletNoVelocityIntegrator(this, drag);
            chainSim.Integrator = integrator;
```

That is all that we need to do to initialize the scene.

Next, we have to ask the chain simulation to update itself.

(class method in Game1.cs, addition of codes in bold)

```
        protected override void Update(GameTime gameTime)
        {
            //update the simulation
            chainSim.Update(gameTime);

            base.Update(gameTime);
        }
```

If you compile and run the simulation now, you will have a chain simulation! The different segments rotate accordingly and with the length constraints in place, this soft body simulation really does behave like a rigid body simulation but only at a fraction of the cost. This is especially true since collision detection of concave objects can be computationally very expensive.

As usual, we shall add in the method to manage the user input so that the user can interact with the scene.

(class methods in Game1.cs, addition of codes in bold)

```
        protected override void Update(GameTime gameTime)
        {
            //poll for input
            HandleInput(gameTime);

            //update the simulation
            chainSim.Update(gameTime);

            base.Update(gameTime);
        }
```

```
private void HandleInput(GameTime gameTime)
{
    if (inputComponent.IsMouseHeldDown(MouseButtonType.Left))
    {
        chainHeadPoint.PointX += -inputComponent.MouseMoved.X / 40;
        chainHeadPoint.PointY += inputComponent.MouseMoved.Y / 40;
    }
}
```

You should now have an interactive chain simulation that looks like the one in Video 11 at the start of the chapter.

In this chapter, we have seen how a simple chain simulation can be faked by using a simple 1D soft body system. Each node of the soft body system is represented with a chain segment model which is rotated to always point to the immediate child node. The length constraints added to the springs maintains the distances between the segments, making it look like collision detection is being handled. Such a simulation is fast and stable and is indeed a good substitution for the more expensive rigid body solution.

Get the source files for the end of this chapter from the SkeelSoftBodyPhysicsTutorial-Chapter4-END folder

Conclusion

We have reached the end of this very long tutorial. If you have followed along all the way until this point and got all the simulations to work, give yourself a pat on your back! You deserve it!

The goal of this tutorial was to introduce the major concepts needed to create soft body simulations in games, and to give an object-oriented implementation of these concepts. We first started with the implementation of a basic spring system and built many different classes in order to make it work, including simulation objects, force generators, integrators and simulations. We then implemented constraints in order to get a decent cloth simulation to work. Several different types of springs were used to connect the vertices of the cloth geometry together to achieve a realistic behavior. Another type of soft body was subsequently introduced where each vertex of the body has a goal position to move towards. By using an alpha texture on a cylinder, we created a slinky based on this kind of goal soft body simulation. Last but not least, we have shown an alternative use of a soft body simulation by showing how it can be used to give a visually plausible rigid body simulation. We created a chain simulation based on a 1D soft body simulator. The simulation is fast, stable and serves as a good substitute for rigid body simulations.

I hope you have learned enough from this tutorial for you to read up on and implement more advanced methods and algorithms. There are many other methods of creating soft bodies available in literature, some of which discretizes the volume of the body (e.g. Finite Element Method), while others try to preserve the volume of the body by introducing fluid calculations in them (e.g. Pressurized Soft Body), just to name a couple. You are encouraged to read up more on these other techniques once you are comfortable with the basics.

Several possible extensions are possible with the current system that we have. You could, for instance, start to create collision detection for the system and even create spatial partitions while you are at it. You could also start to implement other kinds of numerical integrators, forces (e.g. wind), and other kinds of constraints. You could even use the Graphics Processing Unit (GPU) to perform some of the calculations needed for soft body simulations! These topics had to be left out of this tutorial due to the sheer length of this tutorial.

Soft bodies will prove to be an interesting effect to be used in games in the near future, so get started with implementing your own soft body simulation systems and send me a link to your game when you are done with it! Enjoy and have fun!

Skeel (Email: skeel@skeelogy.com Website: http://cg.skeelogy.com)

References

Advanced Character Physics (http://teknikus.dk/tj/gdc2001.htm) by Thomas Jakobsen

Verlet Intergration – Wikipedia (http://en.wikipedia.org/wiki/Verlet_integration)

Devil in the Blue Faceted Dress: Real Time Cloth Animation (http://www.gamasutra.com/view/feature/3444/devil_in_the_blue_faceted_dress_.php) by Jeff Lander