Physics Tutorial 7: Collision Response – Penalty Methods

Summary

We discuss Penalty Methods for collision response. This is achieved by introducing simulated springs between bodies, which push them apart through force and acceleration. The method is shown to be useful for soft bodies and cloth simulation.

New Concepts

Springs, Soft Bodies, Cloth.

Introduction

In the previous tutorial we discussed how to move two colliding bodies apart by applying an impulse in order to directly affect their velocities. In this tutorial we introduce an alternative approach; we directly affect the acceleration of the colliding bodies by applying a force to push them apart. This is known as the Penalty Method. It is achieved by introducing the effects of a coiled spring between the two bodies; the force of the spring expanding pushes the bodies apart. Obviously we are not talking about adding actual springs into the simulation, and rendering them, but we utilise the physical laws of springs to achieve a realistic effect for moving intersecting objects apart.

This approach becomes especially useful when we apply it to soft bodies. So far we have only considered rigid body simulation (i.e. the shape of each object has remained constant while its position and orientation has varied); even the animated characters from the graphics course have consisted of a hierarchical structure of rigid bodies as far as the physics simulation is concerned. A soft body, on the other hand, can change its shape. A classic example of a soft body is cloth – imagine a curtain blowing in the breeze, a football net reacting to a ball kicked into it, or a superhero's cape flowing behind as she runs. Soft bodies are typically modelled as a network of interconnected nodes, and the connections between those nodes are simulated by the physics engine as springs.

Springs

In the real world a spring tends to be a coiled up strand of metal. We are interested in simulating the properties of a spring when it is used to connect two objects; these properties can be summarised as

- When the spring is compressed it forces the two objects apart.
- When the spring is elongated it forces the two objects toward one another.

Basically the spring continually tries to return to its rest length. This is expressed mathematically as:

$$F = -kx$$

where k is the spring constant (a measure of how difficult the spring is to stretch), x is the difference between the rest length of the spring and its current length, and F is the ensuing force at each end of the spring. The equation is known as *Hooke's Law of Elasticity*, which you probably studied during Physics classes in school. Note the negative sign, which shows that this is a restorative force – i.e. it is trying to pull the spring back to its equilibrium length. A higher value of the spring constant k means the spring requires a higher force to stretch or compress it; more to the point for our purposes, a higher value of k will result in a greater force pushing our intersecting objects apart. As ever, for the purposes of our physics engine, the distance and force in the equation are three dimensional vectors, but we will discuss them in simpler terms for clarity.

If this were the sum total of our spring simulation, then the objects which are connected by the spring would continue to bounce back and forth for ever. This is of course unrealistic and not the intention of the simulation. In the real world, an object on the end of a spring will gradually slow down and come to a rest. Hence we need to introduce a *damping factor* into our algorithm.

The damping factor is higher at larger velocities of the object, and a proportional force is introduced via the *damping coefficient* c. The equation for our spring force now becomes

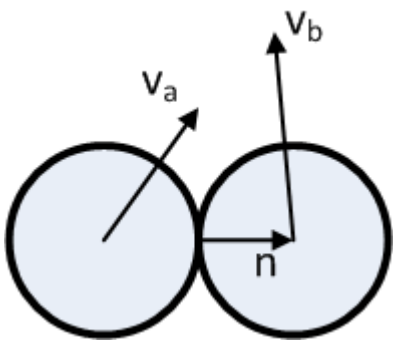
$$F = -kx - cv$$

where v is the velocity of the object along the axis of the spring.

So we have a simple equation which describes how a spring can force two objects apart, or draw them together, while taking into account a realistic damping factor to bring the objects to rest.

Springs for Collision Response

We will again consider the simple case of two colliding spheres from the previous tutorial. Our intention here is to address any intersection identified in the collision response routines by attaching the intersecting objects to either end of a virtual spring, and then rely on the resulting spring forces to push the objects back apart.



We again have the three pieces of intersection information from the collision detection routines; namely the contact point, the contact normal n and the penetration depth d.

The relative velocity is again simple to calculate ($v_{ab} = v_a - v_b$), and we are interested in the component of the relative velocity in the direction of the collision normal (i.e. the dot product of v_{ab} and n. The coefficient of elasticity is k and the damping factor is c, giving the equation:

$$F = -kd - c(n \cdot v_{ab})$$

An equal and opposite force is applied to each object. However, often one of the objects attached to the spring is actually a fixed point (for example, part of the environment), in which case the force on that end of the spring is ignored, and only the free object is affected.

Remember that this method results in a force, which is then applied in the same way as other game forces – i.e. the resulting acceleration is calculated from $F = ma$. The impulse method resulted in a direct change to velocity, so the mass of the objects was taken into account in calculating the impulse; with the penalty method the mass is taken into account when calculating the acceleration from the force.

The choice of values for the two coefficients k and c is vital, and can result in a range of desired collision response types. The higher the value of the damping factor c, the less bouncy the collision, while the higher the value of the elasticity coefficient, the more solid the objects will feel. For example, a low value of k and a low value of c will feel like a trampoline (a big bounce on a soft surface), while a low value of k with a high value of c will feel more like a swamp (sinking into a soft surface). You will need to experiment with values to tune the effect that you want.

Implementation of Springs for Collision Response

This section contains C++ code for simulating a spring class for collision response. Note that the code assumes that the rigid body class includes a method for applying a force to the object, and this method is used as the standard way for moving objects around in the force-based physics simulation.

```
1
2  class Spring_c
3  {
4  public:
5     RigidBody_c *m_rbA;
6     RigidBody_c *m_rbB;
7     Vector3  m_localPosA;
8     Vector3  m_localPosB;
9     float    m_length;   //rest length
10    float    m_ks;    //stiffness
11    float    m_kd;    //damping
12
13    Spring_c( RigidBody_c* rbA, Vector3 localPosA,
14          RigidBody_c* rbB, Vector3 localPosB )
15    {
16
17       m_ks = 30.0f;  //default
18       m_kd = 10.0f;
19
20       m_rbA = rbA;
21       m_rbB = rbB;
22       m_localPosA = localPosA;
23       m_localPosB = localPosB;
24
25       //get the actual world position of the springs
26       Vector3 p0 = Transform( m_localPosA, m_rbA.GetOrientation() ) +
27           m_rbA.GetPosition();
28       Vector3 p1 = Transform( m_localPosB, m_rbB.GetOrientation() ) +
29           m_rbB.GetPosition();
30       m_length = (p1 - p0).Length();
31    }
```

```
32
33      void Update(float dt)
34      {
35          //Work out the world pos of each spring point
36          Vector3 p0 = Transform( m_localPosA, m_rbA.GetOrientation() ) +
37              m_rbA.GetPosition();
38          Vector3 p1 = Transform( m_localPosB, m_rbB.GetOrientation() ) +
39              m_rbB.GetPosition();
40
41          //Work out the err
42          float err =  (p1 - p0).Length() - m_length;
43
44          Vector3 linVelA = m_rbA->GetLinearVelocity();
45          Vector3 linVelB = m_rbB->GetLinearVelocity();
46
47          Vector3 forceDirection = Vector3.Normalise(p1 - p0);
48
49          //Calculate the force from the spring (inc damping)
50          Vector3 force = forceDirection (err * m_ks -
51             Vector3.Dot( forceDirection, (linVelA - linVelB) * m_kd );
52
53          m_rbA->AddForce(p0, force * 0.5f);
54          m_rbB->AddForce(p0, -force * 0.5f);
55      }
56  };
```

Spring Class

The penalty method for collision response in the physics simulation is more straightforward to implement than the Impulse method, and it has the advantage of directly utilising the force-based movement implementation. However great care must be taken so that the results don't feel as though there are actual springs connecting things together; without that care objects may bounce around in an unrealistic manner.

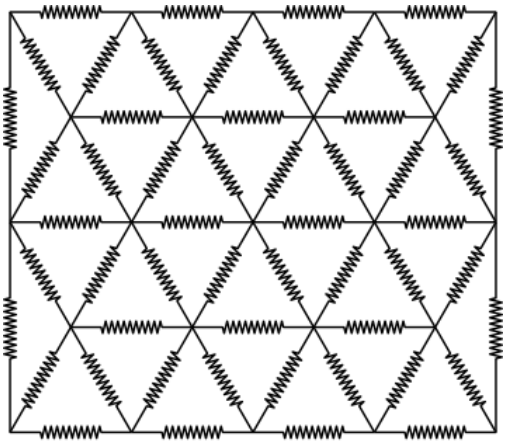
Soft Bodies

So far we have simulated each object as a single item in the simulation (either a particle or a rigid body), or as a hierarchical skeleton of items which are treated by the physics engine as distinct entities. This approach is perfect for most entities which are to be simulated in a game, as we do not anticipate that many objects will be required to change shape as the simulation progresses. Remember that a rigid body can change it's position and orientation, but not its shape. In the cases when we do require a body to change its shape, we need a new approach known as soft body simulation – springs can be utilised to provide soft-bodies in the physics simulation.

A soft body can be modelled as a network of nodes interconnected by simulated springs. The diagram shows a piece of cloth simulated in this way.

The rest position of the cloth has no tension in the springs, so the mesh is evenly spaced. Applying a force to one part of the mesh will cause the surrounding springs to stretch, and to pull the connected nodes which in turn stretch the next layer of springs attached to them. As the force is removed the springs contract and bring the overall mesh back to its rest position. A three dimensional deformable object (such as a jelly) can be simulated similarly, with a three dimensional array of nodes. It is also common to use a one-dimensional set of noes to simulate a rope, a chain or even a strand of hair.

A common approach in simulating deformable objects using this spring-based soft body solution is to match the graphical model to the simulated model. That is to say that each vertex of the graphical model is represented by a node in the physical model, and each edge in the graphical model coincides with a spring.



It should be stressed that soft bodies are extremely expensive items to simulate both in terms of computation and memory. Each spring connecting each node must be simulated, which is obviously considerably more computationally expensive than treating the object as a single entity as happens in rigid body simulation. Similarly, as the mesh of springs and nodes must be stored in a data structure, there is also a greater cost to memory. For these reasons, soft body simulation tends to be used for specific instances in games, which give maximum impact to the player. It is also worth pointing out that this approach is not really based on any kind of real-world physics; it is a technique for providing some behaviour which looks and feels good in the game simulation (i.e. in reality a cloth isn't a network of connected springs).

There are instances where a body needs to deform, but not return to its original shape (for example a crumple zone on a car) – in those instances the spring approach is not suitable as the basic nature of a spring is that it is constantly trying to return to its rest length. In the instance of the car crumple zone, the bodywork is still modelled as a network of interconnected nodes, and the simulation knows that they have freedom of movement along a particular axis in one direction only. Consequently applying a sufficiently high force will cause the nodes to squash up together, but removing the force does not allow the nodes to revert to the uncompressed state.

Implementation of a Ball-Spring Chain

The code sample shows how to implement a chain of nodes connected by springs with freedom to move in two dimensions.

```
1
2  // Gravity acceleration
3  const static float G = -9.8f;
4
5   // Mass
6  const static float MASS = 10.0f;
7
8  // Spring parameter in Hook's law
9  const static float KS = 50.0f;
10
11 // Velocity damping parameter
12 const static float KD = 1.0f;
13
14 // Time slice
15 const static float DT = 0.1f;
16
17 const static int MAXNUMBALLS = 7;
18
19 // Spring demo class - whole demo is encapsulated within this class
20  class SpringDemo_c
```

```
{
    DArray< Vector3 > m_ballPosition;
    DArray< Vector3 > m_ballVelocity;
    DArray< Vector3 > m_ballForce;

public:
    SpringDemo_c()
    {
        // Create our balls
        for (int i=0; i<MAXNUMBALLS; i++)
        {
            m_ballPosition.Push( Vector3(0,0,0) );
            m_ballForce.Push( Vector3(0,0,0) );
            m_ballVelocity.Push( Vector3(0,0,0) );
        }

        // Fix first ball
        m_ballPosition[0].x = 100;
        m_ballPosition[0].y = 100;

        // Fix last ball
        m_ballPosition[m_ballPosition.Size()-1].x = 500;
        m_ballPosition[m_ballPosition.Size()-1].y = 100;

        // Initialize moving balls
        for (int i=1; i<m_ballPosition.Size()-1; i++)
        {
            m_ballPosition[i].x = RandomFloat(0, 450);
            m_ballPosition[i].y = RandomFloat(0, 90);
        }
    }

    void DrawBalls()
    {
        for (int i = 0; i<m_ballPosition.Size(); i++)
        {
            DrawCircle2D( m_ballPosition[ i ], 10.0f);
        }
        // Draw springs
        for (int i=0; i<m_ballPosition.Size()-1; i++)
        {
            DrawLine2D(m_ballPosition[i], m_ballPosition[i+1]);
        }

        // Calculate forces on each node
        void RecalculateBallPosition()
        {
            // Calculate the spring force
            for (int i = 1; i<m_ballForce.Size()-1; i++)
            {
                // Force from the left ball and right ball
                Vector3 f0 = KS * (m_ballPosition[i]
                    - m_ballPosition[i-1]);
                Vector3 f1 = KS * (m_ballPosition[i]
                    - m_ballPosition[i+1]);
                Vector3 F = f0 + f1;
                m_ballForce[ i ] = F - KD*m_ballVelocity[ i ];
                m_ballForce[ i ].y -= MASS*G;
```

```
79          }
80
81          // Calculate the new position of each nodes
82          for (int i = 1; i<m_ballPosition.Size()-1; i++)
83          {
84              // accelerations
85              Vector3 a = m_ballForce[ i ] * ( 1.0f / MASS );
86
87              // velocities
88              m_ballVelocity[ i ] += a * DT;
89
90              // positions
91              m_ballPosition[ i ] += m_ballVelocity[ i ] * DT;
92          }
93      }
94
95  public:
96      // Game state is kept here in the Draw update
97      // for simplicity, we assume a fixed
98      // timestep update call ( so the function is
99      // always called at the same interval )
100     void Draw()
101     {
102         RecalculateBallPosition();
103
104         DrawBalls();
105     }
106 }
```

Ball–Spring Chain

Tutorial Summary

In this tutorial we have introduced an alternative approach to collision response in the form of penalty methods, i.e. a method which directly affects the accelerations of colliding objects in order to resolve that collision. This was achieved by simulating a spring between the two bodies. It has also been shown how this approach can be used to simulated soft bodies (i.e. deformable objects which change shape when an external force is applied, but try to revert to their original shape).

This concludes the tutorial series on physics engines. We have discussed the overall aims and practicalities of a physics simulation for games, and described how a force-based Newtonian simulation is suitable to our needs. We have used numerical integration to move our objects around based on the Newtonian forces acting upon them. Once all the objects have been moved for a specific time step, we have looked at how to detect whether there have been any collisions or intersections; this work was divided into the broadphase for quickly identifying possible collisions, and the narrowphase for precisely identifying actual intersections. e have then used that information to perform some collision response, and investigated two approaches, namely the impulse method and the penalty method. The end result of the tutorial series should be a functioning physics engine which simulates both linear and angular interaction of the game objects and environment.

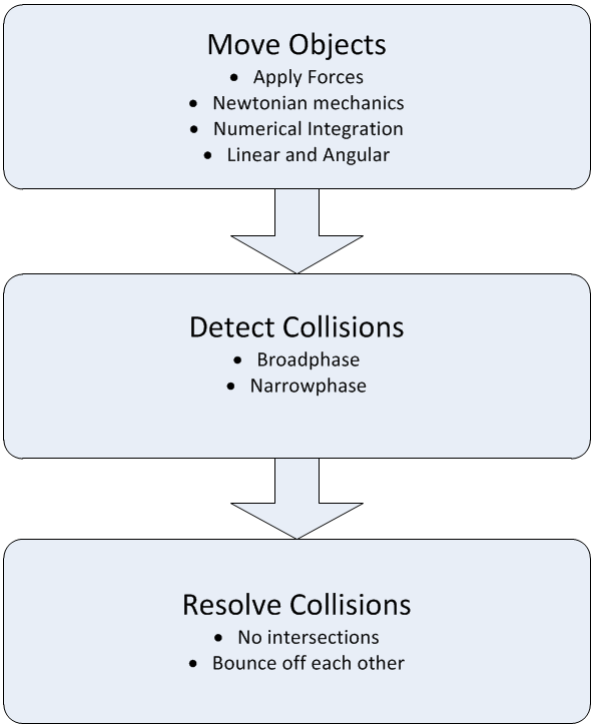


Figure 1: Physics Engine Overview