Collision Response: Bouncy, Trouncy, Fun

I was all set this month to start talking about how to handle collision response. It seemed to be the next logical step from the discussion last month on methods for detecting collisions between 3D objects ("When Two Hearts Collide," February 1997). I thought I could just have these objects that you could move around, make collide, and then watch their responses. Yeah, collision response, that will be great! Then I thought, "How am I going to get these objects flying around in the first place?" Well, I could give each object an initial velocity and they would collide. But, I would need world boundaries for those objects to bounce off of so they would stay in play. To direct the objects, I need to be able to apply force. Suddenly, instead of a nice collision demo, I had designed ASTEROIDS. All I wanted was a little demonstration of a fairly simple concept and instead I ended up applying forces and acceleration to particles. I had stumbled on the big "D" word: Dynamics.

That's alright. I will not be afraid. I always say, "Turn a problem into an opportunity." For months, I've been considering picking up where Chris Hecker left off his "Behind the Screen" columns back in June 1997. Chris had created a very interesting rigid body dynamics simulation and spearheaded the use of hardcore physics in the game development community. However, physics is a huge field full of fertile topics that can be distilled into nice column-sized pieces. So once more good friends, into the breach.

What's So Dynamic About It?

When I was writing about inverse kinematics back in September, I was only really interested in kinematics: that is, the study of motion without regard to the forces that cause it. Dynamics, I said, concerns how forces are used to create motion, and I didn't want to open up that can of worms. Well, the can is now open and the worms are climbing all over.

I'm going to have to recap a bit, but I suggest you go back and reread Chris's column from the January 1997 *Game Developer*, "Physics, the Next Frontier." If you don't have the magazine handy, the article is available on the Definition Six web site at http://www.d6.com/users/checker.

This month, I'm going to focus on particle dynamics. What is particularly important about particle dynamics is the relationship between force, f, the mass of a particle, m, and the acceleration of that particle, a. This can be stated in the familiar Newtonian notation as $f = ma$. You may recall from Chris's column that the acceleration of a particle is the derivative with respect to time of the velocity of that particle, v. Likewise, the velocity of the particle is the derivative with respect to time of the position of the particle, x. You can see how this relationship works in Eq. 1.

$$f = ma$$
$$a = \frac{dv}{dt} = \dot{v} = \ddot{x} = \frac{f}{m}$$
$$v = \frac{dx}{dt} = \dot{x}$$

(Eq. 1)

So, let me state the problem I'm trying to solve. Given a set of forces acting on a particle at time t, where will that particle be after a small amount of time has passed? It's clear that with the value of the force and the mass of the particle, I can obtain the acceleration of the particle. If I integrate that acceleration with respect to t, I'll end up with the new velocity of the particle. If I integrate again, I get the new position. Easy, right?

The structure for a particle is in Listing 1. It's easier to store $1/m$ for the particle because this is what I need in the equations. The forces that act on the particle accumulate in the f term. With this information, I can integrate the dynamic system forward in time to establish a new position for the particle. This process involves solving ordinary differential equations. Fortunately,

LISTING 1. *The particle type.*

```
// TYPE FOR A PHYSICAL PARTICLE IN THE SYSTEM
struct tParticle
{
        tVector pos;            // Position of Particle
        tVector v;              // Velocity of Particle
        tVector f;              // Total Force Acting on Particle
        float   oneOverM;       // 1 / Mass of Particle
};
```

Many have told Jeff that his top is made of the rubber and bottom of the spring. Bounce him and Darwin 3D a note at jeffl@darwin3d.com

LISTING 2. *My simple Euler intergrator.*

```
////////////////////////////////////////////////////////////////////////////
// Function:  Integrate
// Purpose: Calculate new Positions and Velocities given a deltatime
// Arguments: DeltaTime that has passed since last iteration
// NoteS:   This integrator uses Euler´s method
////////////////////////////////////////////////////////////////////////////
void CPhysEnv::Integrate( float DeltaTime)
{
/// Local Variables ///////////////////////////////////////////////////////
  int loop;
  tParticle *source,*target;
////////////////////////////////////////////////////////////////////////////
  source = m_CurrentSys; // CURRENT STATE OF PARTICLE
  target = m_TargetSys;  // WHERE I AM GOING TO STORE THE NEW STATE
  for (loop = 0; loop < m_ParticleCnt; loop++)
  {
    // DETERMINE THE NEW VELOCITY FOR THE PARTICLE
    target->v.x = source->v.x + (DeltaTime * source->f.x * source->oneOverM);
    target->v.y = source->v.y + (DeltaTime * source->f.y * source->oneOverM);
    target->v.z = source->v.z + (DeltaTime * source->f.z * source->oneOverM);

    // SET THE NEW POSITION
    target->pos.x = source->pos.x + (DeltaTime * source->v.x);
    target->pos.y = source->pos.y + (DeltaTime * source->v.y);
    target->pos.z = source->pos.z + (DeltaTime * source->v.z);

    source++;
    target++;
  }
}
```


Chris's column described a numerical method of solving these problems. Listing 2 contains code that uses the simplest numerical integrator, known as Euler's method, to compute the new state of the system. The great thing about this integrator is that it's simple to implement and understand. However, because it's a simple approximation, it's subject to numerical instability, as we will see later.

You Can't Force Me to Move, Can You?

I now have a method for dynamically moving particles around in a realistic fashion. However, to get anything interesting to happen, I need to get things moving. This requires the application of some brute force, or several forces. But what kinds of forces do I want to apply to my little particles?

Well, the obvious force that has been applied to objects in games since the beginning of computer simulations is gravity. When I wrote the article on particle systems back in July 1998 ("The Ocean Spray in Your Face"), I had a very simple system for applying a force such as gravity. This time, however, I want to be a bit more physically realistic. Gravity is a constant force that is being applied to all particles. In order to realistically simulate gravity, force must be added into the particle's force accumulator every system update. In general, this force is a vector pointing down along the y axis. However, there's nothing to stop a simulator from having a gravity vector that points in a different direction. In fact, one of the very cool things about having a good physical simulation is that gravity can change and things will still "look" correct. This realistic look may not occur if you are trying to hand animate an object.

Putting the Bounce Back in my Bungie

Now, gravity was a pretty obvious force to apply to particles. But what else can I do? A loose connection of points isn't really all that interesting to watch even if it is simulated with accurate physics. It would be much more entertaining if I could connect those particles to form structures.

What about stretching a spring between two particles? This procedure is actually easy to implement. Hook's spring law (Eq. 2) is a pretty good way of representing the forces that a spring exerts on two points.

$$f_a = -k_s\left(|L| - R\right) + k_d \frac{\dot{L} \bullet L}{|L|}\frac{L}{|L|}$$
$$f_b = -f_a$$
$$L = a - b$$
$$\dot{L} = v_a - v_b$$

(Eq. 2)

This formula represents the force applied to particles *a* and *b*; the distance between these particles, *L*; the rest length of the spring, *r*; the spring constant or "stiffness", k_s; the damping constant, k_d; and the velocity of the particles, *v*. The damping term in the equation is needed in order to sim-

LISTING 3. *A damped spring force.*

```
p1 = &system[spring->p1];
p2 = &system[spring->p2];
VectorDifference(&p1->pos,&p2->pos,&deltaP);          // Vector distance
dist = VectorLength(&deltaP);
// Magnitude of deltaP

Hterm = (dist - spring->restLen) * spring->Ks;        // Ks * (dist - rest)

VectorDifference(&p1->v,&p2->v,&deltaV);              // Delta Velocity Vector
Dterm = (DotProduct(&deltaV,&deltaP) * spring->Kd) / dist;  // Damping Term

ScaleVector(&deltaP,1.0f / dist, &springForce);       // Normalize Distance Vector
ScaleVector(&springForce,-(Hterm + Dterm),&springForce);  // Calc Force
VectorSum(&p1->f,&springForce,&p1->f);                    // Apply to
Particle 1
```

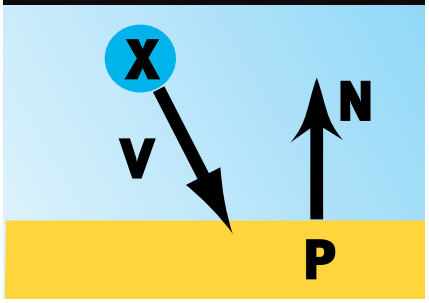

FIGURE 1. *A particle colliding with a plane.*

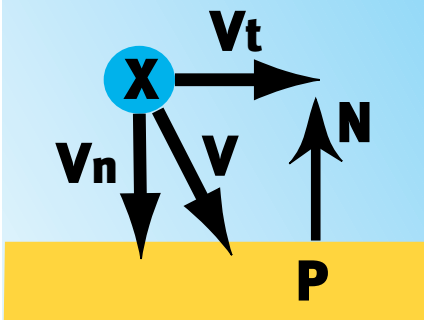

FIGURE 2. *Components of a collision.*

ulate the natural damping that would occur due to the forces of friction. This force, called viscous damping, is the friction force exerted on a system that is directly proportional and opposite to the velocity of the moving mass. In practice, the damping term lends stability to the action of the spring. The code applying the spring force on two particles is in Listing 3.

Other Forces

Viscous drag should be applied to the entire system. A drag is a great way of making the particles look as though they are floating around in oil. It also adds numerical stability to the system, meaning that the particles won't bounce around too much. A viscous drag force is applied by multiplying a damping constant, K_d, with the velocity of the particle and subtracting that force from the accumulator.

Momentary forces are also very useful for interacting with the simulation. I've used a spring tied to a particle and attached the mouse to drag the object around. A force applied to a particle can be used to create a motor or other source of motion.

You can also make some interesting effects by locking a particle. That is, by turning off the simulation for a particular particle, it becomes fixed and can act as an anchor point. (You can achieve the same effect by causing the particle to have an infinite mass. In the simulator, simply set the particle's mass to zero.) Immobilizing one particle like this creates many possibilities for creating complex simulations.

Finally, Back to Collision

Whew, now that I have a nice dynamic particle simulator, I can start talking about collision detection and response again. The simplest form of collision detection that I can add to this simulation is point-to-plane collision. With particles, it will be easy. Last month, I discussed the use of the dot product to determine whether a point has collided with a plane. Take a look at Figure 1.

Particle X with a velocity vector V is moving towards plane P with a normal N. I know that a collision of some sort occurred if $(X\text{-}P) \bullet N < \varepsilon$, where ε is some small threshold near zero. If that value is $< -\varepsilon$, then the particle has passed through the wall, penetrating it. That won't make my simulator happy, so if a particle is penetrating any boundary, it's necessary to back up the simulator a little and try again. If the dot product is just very near zero, then I have what is called a contact and I need to check further.

A particle in contact with a boundary may not be colliding with that boundary if the particle is moving away from the boundary. The relative velocity of the two bodies is checked by calculating $N \bullet V$. If that value is less than zero, the two bodies are in colliding contact and I need to resolve the collision.

To resolve the collision, I need to calculate two more vectors. They represent the motion parallel and tangential to the normal of collision. Take a look at Figure 2.

The normal of collision is simply the normal to the plane. I calculate the velocity after the collision with Eq. 3.

$$V_n = (N \bullet V)N$$
$$V_t = V - V_n$$
$$V' = V_t - K_r V_n \qquad \text{(Eq.3)}$$

In this equation, K_r is the coefficient of restitution. This is the amount of the normal force, V_n, that is applied to the resulting force. If K_r is 1, I have a totally elastic collision. If it is 0, the particle sticks to the plane.

Building with Sticks

Now that I have this nifty particle simulator where I can attach particles with springs and apply forces to them, it's time to build something. Let me start with a simple block such as the one in Figure 3.

Each of the edges of the object is a spring connecting the vertices. Unfortunately, if I run this object through the simulator, I end up with a big heaping mess. The mess occurs

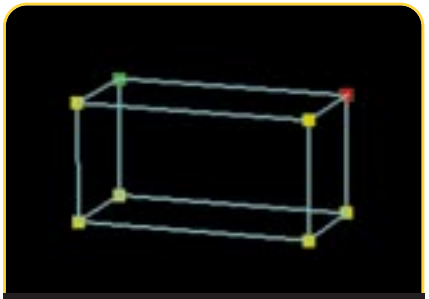

FIGURE 3. *A simple dynamic cube.*

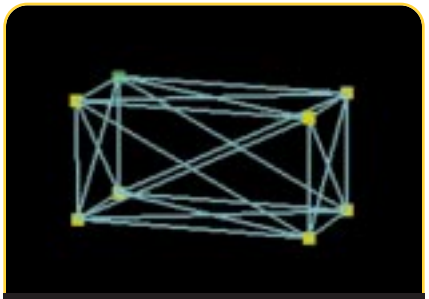

FIGURE 4. *A stable cube.*

FIGURE 5. *A cube out of control.*

because the springs connecting the vertices aren't enough to provide stability for the cube. In order to create a cube that won't collapse, it's necessary to put crossbeam supports on each face of the cube (Figure 4).

Creating objects this way feels more like constructing a bridge than 3D modeling. You find yourself adding struts and crossbeams all over the place.

FOR FURTHER INFO

- Baraff, David, and Andrew Witkin. "Physically Based Modeling," SIGGRAPH Course Notes, July, 1998, pp. B1-C12. I built my first particle dynamics simulator after seeing an article by David Barraff a couple of years ago. For this article, I used one source of his and Andrew Witkin's in particular.
- Hecker, Chris. "Behind the Screen." *Game Developer*, October 1996 – June 1997. Credit for the ideas and some of the methods of simulation go to Chris Hecker. I have tried to base my code on many of his ideas so it will be familiar to readers. His excellent series of articles on rigid body physics got me and many others excited about real-time physics. Hopefully, I can continue to build on this tradition. Also available on Chris's web site at http://www.d6.com.

You will need several good math and physics books if you really want to get into this topic. Here are a few that I used in this article.

- Beer and Johnston. *Vector Mechanics for Engineers: Dynamics*, Sixth Edition, WCB/McGraw-Hill, New York, 1997.
- Mullges and Uhlig. *Numerical Algorithms with C*, Springer-Verlag, New York, 1996
- Acton, Forman S. *Numerical Methods that Work*, Harper and Row, New York, 1970. This last book was a useful little book my father had from his days of working on guidance systems. Now I am using it to make virtua-jello. Go figure.
- Doug DeCarlo at the University of Pennsylvania wrote an application for X-Windows called XSpringies that allows you to simulate 2D particle-spring interactions. You can check this out from his website at http://www.cis.upenn.edu/~dmd/doug.html or get the program at ftp.cis.upenn.edu/pub/dmd/xspringies/xspringies-1.12.tar.Z

Leave a face open and it behaves correctly. The face without the crossbeam supports is more likely to collapse.

Bring Me Stability or Bring My Program Death

I mentioned before that by using a simple Euler integrator, I'm sacrificing numerical stability for ease and speed of calculation. You may wonder, however, what happens when the system becomes unstable. There's a really easy way to find out what will happen. Remember the spring coefficient that was applied to the particles? This coefficient represented the stiffness of the springs used. If I set that value fairly high because I want really stiff springs, the little Euler integrator cannot handle it. If you run that cube I had with stiff springs, you may see something like Figure 4 or something equally interesting. The still frame doesn't do it justice. This is a rigid body way out of control.

There's a solution to combat this instability beyond, "Don't do that" — it's to give my integrator an upgrade. Euler's method is simply not sophisticated enough to handle problems such as this. Next month, I will take a look at how I can improve the integrator with something a little more stable.

Kid in a Gummi Bear Store

I really find in fun to play with this simulator. It's very satisfying to bring in shapes and play with making them stable and tweaking the spring and gravity settings. You then can fling the objects all around and bounce them off the walls. There are many more variables that can be added to the simulator. Other forces such as contact friction can be added. Some interactive features such as pinning vertices would make it more fun. But I think we're on our way to a really fantastic Jello-land simulator. Next month, I also plan on adding support for multiple bodies as well as object-to-object collision. Check out the source code and demo application on the *Game Developer* web site at http://www.gdmag.com. It will allow you to load in your own shapes, connect them with springs, and play around with the simulator. ■