# Real-Time Game Physics Technical Addendum

This document is a supplement to the real-time physics chapter in the 2nd edition of *Introduction to Game Development.* The text below was adapted from the 1st edition of the book. Here we include an extension of the discussion of theoretical and numerical physics to include generalized (but frictionless) rigid body motion.

Note that equation numbers 1 through 19 (some of which are referenced here) are contained in the printed chapter. Equations 20 and higher are contained in this document. Figure numbers referenced in this document refer to figures located in this document, e.g., there are no references here to figures contained in the printed text.

# Generalized Translational Motion

## *General Rigid Bodies vs. Spherical Particles*

We will now generalize our discussion to include arbitrary sized and shaped rigid bodies, rather than perfectly smooth spherical particles. There is exactly one reason for the smooth sphere restriction so far: we were able to completely avoid considering the issue of rotational motion. This is important: *all of the equations presented previously are perfectly valid for rigid bodies of any shape that are of finite size.*

Despite the fact that the previous equations support arbitrarily shaped objects, they describe the translational physics of a single point on the object (*e.g.*, the center of a sphere). Rigid body objects fill a volume in space, and so we must find an appropriate point on the object for use in a simulation. It is standard practice to choose the object's *center-of-mass* to be the reference point for translational motion. The reason for this choice is that it removes *inertial coupling* that would otherwise make the translational equations dependent on rotation and the entire system more difficult to solve.

Equation 20 defines the location of the center-of-mass for a rigid body.

$$\mathbf{p}_{center-of-mass} = \frac{1}{mass}\iiint_{Vol} \rho \mathbf{r} dxdydz \tag{20}$$

The density, $\rho$ is the mass per unit volume, with units of type mass per the cube of distance. *The SI units for density are kilograms per meter cubed (kg/m$^3$).* The variable, **r**, is the vector from a known reference point to the location of a differential element of the object's mass. The resulting center-of-mass location is calculated relative to the same reference point. Note that density might be a constant, but in general it might also vary with the differential element position, **r**. For example, if you are computing the center-of-mass of an object made partly of steel and partly of plastic, the integration over the plastic parts would use a different density from the integration over the steel parts. For

arbitrarily-shaped objects, Equation 20 can be difficult to evaluate. Brian Mirtich [Mirtich96] and David Eberly [Eberly03] have documented robust techniques for evaluating the center-of-mass of triangle mesh objects, which are extremely useful for game development.

Figure 1 illustrates a rigid body and its center-of-mass, and shows the standard symbol for center-of-mass. For physics simulation, we normally choose a local object-aligned coordinate system with its origin located at the center-of-mass. This same local coordinate system can be used for collision detection and rendering, as well as physics simulation.

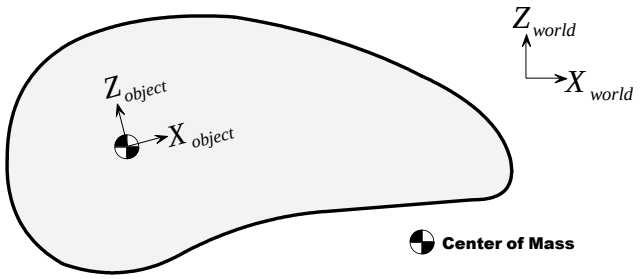


Figure 1. The center-of-mass of an object.

We will now consider a variety of non-constant external forces that contribute to the generalized motion of an object. Any combination of these forces might be acting on an object at a given time. You can obtain the net force acting on an object, $\mathbf{F}_{net}$, by simply adding all applied forces together. $\mathbf{F}_{net}$ is exactly the value of $\mathbf{F}$ to be used in the state derivative vector for numerical integration. If $\mathbf{F}_{net}$ has zero magnitude, the object has zero acceleration and is said to be in *translational equilibrium*, though it may still be moving at a nonzero velocity.

## *Linear Springs*

The first generalized force that we will consider is due to a spring connecting two objects. Figure 2 illustrates a spring that is connecting two objects.

The spring is connected to one of the objects at location $\mathbf{p}_{e1}$, and to the other object at location $\mathbf{p}_{e2}$, the endpoints of the spring. The length of the spring is simply the Euclidian distance between the two endpoints. A spring has a so-called *rest length*, $l_{rest}$, which defines the length of the spring when it is neither compressed nor stretched. The spring exerts zero force when its length is its rest length. When the spring is stretched to be longer than $l_{rest}$, it applies an attraction force to each of the objects. When the spring is compressed to be shorter than $l_{rest}$, it applies a repulsion force to each of the objects. Equation 21 presents the simplest realistic model of spring force, *Hooke's Law*, in which the force is a linear function of the displacement from $l_{rest}$.

$$\mathbf{F}_{spring} = k\left(l - l_{rest}\right)\hat{\mathbf{d}} \tag{21}$$

The variable, *k*, is the *spring stiffness*, a measure of the strength of the spring. *The stiffness is measured in units of type force per unit length. The SI units for spring stiffness are Newton's per meter.* The variable, *l*, is the current length of the spring, and the vector variable, $\hat{\mathbf{d}}$, is a unit length vector in the direction from $\mathbf{p}_{e1}$ to $\mathbf{p}_{e2}$. The spring force, $\mathbf{F}_{spring}$, is applied to object 1 at location $\mathbf{p}_{e1}$. An equal but opposite force, $-\mathbf{F}_{spring}$, is applied to object 2 at location $\mathbf{p}_{e2}$.

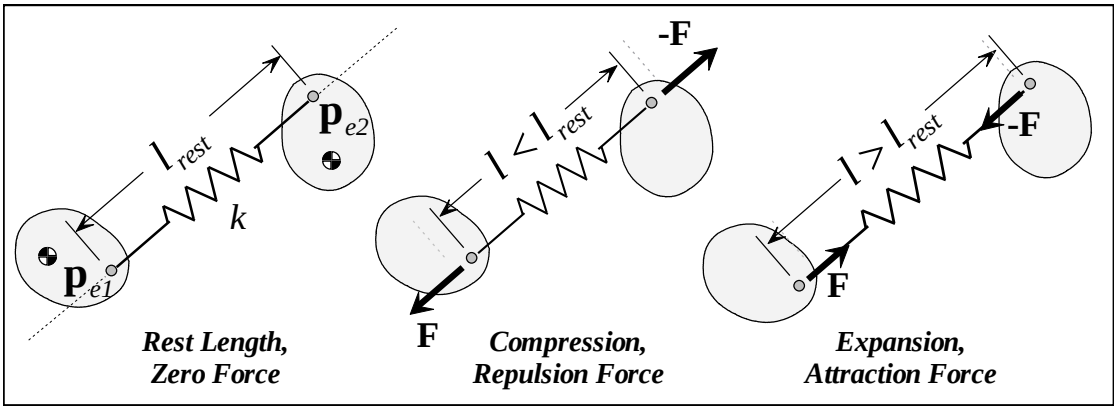



Figure 2. The linear spring.

## *Viscous Damping*

Viscous damping is a dissipative force (one that reduces kinetic energy) acting on objects moving at low speeds through fluids such as air, water, and oil. Mechanical damping devices called *dashpots* generate viscous damping forces, and are often utilized to reduce vibrations in machines, vehicle suspension systems, etc. Dashpots apply a damping force to the objects to which they are connected, as shown in Figure 3.

The viscous damper applies forces along the damper axis. The magnitude of the forces is related to the relative velocity of the objects along the damper axis. Equation 22 defines the force. The parameter, *c*, is the damping coefficient, *measured in units of type mass per unit time. The SI units for damping coefficient are kilograms per second.* The parameter, $\hat{\mathbf{d}}$ is a unit vector in the direction from $\mathbf{p}_{e1}$ to $\mathbf{p}_{e2}$.

$$\mathbf{F}_{damping} = c\left(\left(\mathbf{V}_{ep2} - \mathbf{V}_{ep1}\right)\cdot\hat{\mathbf{d}}\right)\hat{\mathbf{d}} \tag{22}$$

The damping force, $\mathbf{F}_{damping}$, is applied to object 1 at location $\mathbf{p}_{e1}$. An equal but opposite force, $-\mathbf{F}_{damping}$, is applied to object 2 at location $\mathbf{p}_{e2}$.

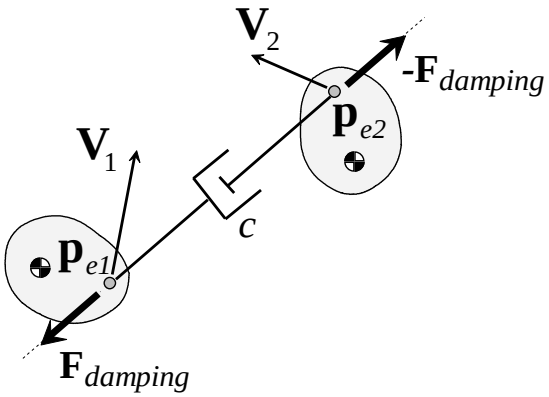


Figure 3. The dashpot damper. If objects approach each other along the line between the points where the damper is connected, the damping force is repulsive.

## *Aerodynamic Drag*

An object traveling through a fluid, such as air or water, experiences a drag force that acts in the opposite of the object's velocity through the fluid. Equation 23 provides a simple approximation for this aerodynamic drag. Here, $C_D$ is the drag coefficient, which has no units. Typical values for non-streamlined objects range from 0.1 to 0.4. The variable $S_{ref}$ is a representative front-projected area of the object. For game objects, other than aircraft, choose $S_{ref}$ to be the cross-section area of a bounding sphere for the object, and assume the drag force acts at the center-of-mass of the object. In Equation 23, the variable $\rho$ is the mass density of the fluid through which the object is traveling. The reference, [Rhodes05], provides a comprehensive overview of aerodynamic forces.

$$\mathbf{F}_{drag} = -\frac{1}{2}\rho|\mathbf{V}|^2 C_D S_{ref} \frac{\mathbf{V}}{|\mathbf{V}|} \tag{23}$$

## *Surface Friction*

When two objects make contact, either during a collision, while in resting contact, or during sliding contact, the objects potentially exert a force on each other within the contact plane. This tangential force is called *friction*. The behavior of the friction force is rather complex. Observe that if you apply a horizontal force to an object at rest on a surface, the object does not begin moving unless the force exceeds a threshold. Once the force exceeds the threshold, the object begins moving, often abruptly. When the object is moving, the force required to keep the object moving is less than the force required to cause the initial motion. This observation illustrates the presence of a variable *static friction* when the object is at rest, and a *dynamic friction* when the object is in motion.

Coulomb developed the most common model of friction in the year 1781. You may be familiar with *Coulomb friction* from your prior studies. Using the Coulomb model, the

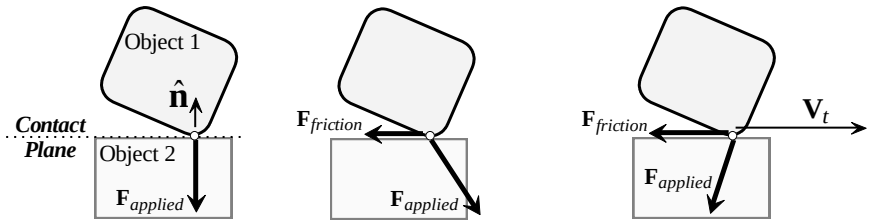
magnitude of static friction is equal to the component of an external force, $\mathbf{F}_{applied,}$ applied between the objects in the contact plane up to a maximum magnitude of $\mu_s|\mathbf{F}_n|$, where $\mu_s$ is a *static friction coefficient*, and $\mathbf{F}_n$ is the component of the applied force parallel to $\hat{\mathbf{n}}$, given by $\mathbf{F}_n = \hat{\mathbf{n}}(\mathbf{F}_{applied} \cdot \hat{\mathbf{n}})$. The magnitude of dynamic friction, generated when there is relative motion in the contact plane between the two objects, is given by $\mu_d|\mathbf{F}_n|$. Here, $\mu_d$ is the *dynamic friction coefficient*. The friction coefficients are functions of the material properties of the two objects that are in contact. For example, the value of $\mu_s$ between two objects made of wood ranges from around 0.2 to around 0.75. The value of $\mu_d$ is usually smaller than $\mu_s$. The difference between the coefficients leads to a discontinuity in the magnitude of friction force at the moment the objects begin to slide past one another, and this discontinuity can cause a difficulty in numerical simulations. *The friction coefficients have no units.*

There are three basic scenarios for two objects in contact with each other, shown in Figure 4. The following are the conventions used in the figure and equations to follow: $\mathbf{F}_{applied}$ is the total force, less friction, applied by object 1 onto object 2; $\mathbf{V}_t$ is the tangential component of the relative velocity of object 1 moving past object 2; $\hat{\mathbf{n}}$ is the contact normal measured outward from object 2; the resulting friction force, $\mathbf{F}_{friction}$ as calculated is applied on object 1, so that the net force on object 1 becomes $\mathbf{F}_{net}=\mathbf{F}_{applied}+\mathbf{F}_{friction}$. By Newton's Third Law of Motion, $-\mathbf{F}_{friction}$ is applied on object 2, so there is no need to calculate the friction force twice. The tangential relative velocity, $\mathbf{V}_t$, is given by $\mathbf{V}_t = (\mathbf{V}_1 - \mathbf{V}_2) - \hat{\mathbf{n}}((\mathbf{V}_1 - \mathbf{V}_2)\cdot\hat{\mathbf{n}})$.

An intuitive example of an external force, $\mathbf{F}_{applied}$, applied by one object onto another is simply the weight of an object resting on a horizontal surface. The resting object applies a force equal to its weight on the surface in the direction $-\hat{\mathbf{n}}$, and from Newton's Third Law of Motion, the surface applies an equal but opposite force back on the object. If you exert an additional horizontal force on the object, attempting to slide the object, $\mathbf{F}_{applied}$ would be the sum of the object's weight plus the additional horizontal force.

If $\mathbf{V}_t$ is zero, the friction force is given by Equation 24. Note that this equation guarantees that the magnitude of static friction never exceeds the Coulomb maximum of $\mu_s|\mathbf{F}_n|$. The tangential component of the applied force, $\mathbf{F}_t$, is given by $\mathbf{F}_t = \mathbf{F}_{applied} - \mathbf{F}_n$.

$$\mathbf{F}_{friction} = -\frac{\mathbf{F}_t}{|\mathbf{F}_t|}\min\left(\mu_s|\mathbf{F}_n|, |\mathbf{F}_t|\right) \tag{24}$$




$\mathbf{F}_{applied}$ parallel to $\hat{\mathbf{n}}$
No relative velocity in contact plane
$\mathbf{F}_{friction}$=<0,0,0>

$\mathbf{F}_{applied}$ not parallel to $\hat{\mathbf{n}}$
No relative velocity in contact plane
$\mathbf{F}_{friction}$ given by Eq. 24

$\mathbf{V}_t$ is nonzero
$\mathbf{F}_{friction}$ given by Eq. 25

Figure 4. Friction acting on object 1 at a contact point for three scenarios. The force, $\mathbf{F}_{applied}$, is applied by object 1 onto object 2.

If $\mathbf{V}_t$ is zero, and $\mathbf{F}_t$ exceeds $\mathbf{F}_{friction}$ in magnitude, the objects will begin to accelerate tangentially past one another. For the case when $\mathbf{V}_t$ is nonzero, the friction force is given by Equation 25. Note here that $\mathbf{F}_{applied} \cdot \hat{\mathbf{n}}$ is always negative, and so friction acts in a direction opposite $\mathbf{V}_t$. Friction is a dissipative force when the objects are in relative motion, and acts to reduce the kinetic energy of the two objects.

$$\mathbf{F}_{friction} = \frac{-\mathbf{V}_t}{|\mathbf{V}_t|} \mu_d |\mathbf{F}_n| \tag{25}$$

It is interesting to note that it is possible for dynamic friction to act in the same direction as the tangential applied force, rather than always against it. For example, consider a passenger train headed on a deadly course towards a bridge recently destroyed by a villain. Our hero has managed to set the brakes so that the wheels are no longer turning. Sliding, Coulomb friction acts to slow the train down. But friction is insufficient to stop the train in time. Our hero might tie one end of a chain to the train, then hold the other end of the chain while bracing herself against a building, or a perhaps a mountain. The force that the chain applies to the train acts opposite the train's velocity, in the same direction as the friction force. Thus, the applied force and the friction force act in the same direction, both contributing to the deceleration of the train and saving of lives.

Friction is a surprisingly difficult force to comprehend. You may find it useful to explore other technical documents related to implementing physics for games, such as the Essential Math tutorials mentioned in the printed text, other presentations on real-time physics presented at the Game Developer's Conference (GDC), and published online at http://www.gamasutra.com. Archives of presentations from past GDC conferences can be found online at http://www.gdconf.com.

## *A Simple Spring-Mass-Damper Soft-Body Dynamics System*

To understand better how you might go about using these various forces, consider a fun example. Using the results of this section and the prior section, you can construct a simple soft-body dynamics simulator. Simply create a polygon mesh with an interesting shape. You can create the mesh in code or using a digital content creation modeling package. For this system, you will use physics to update the position of the vertices of the mesh. For the physics system, create a particle at the location of each vertex of the mesh, and assign a mass to the particle. Then, create a spring and a damper between unique pairs of particles. The spring rest lengths should be equal to the initial distance between particles. Figure 5 illustrates this configuration, for a 2D model. The arrangement extends naturally to 3D. It is important that you include springs that connect particles on opposite sides of the mesh, to prevent the shape from collapsing; however, for complex meshes you don't necessarily have to have springs between every unique pair.

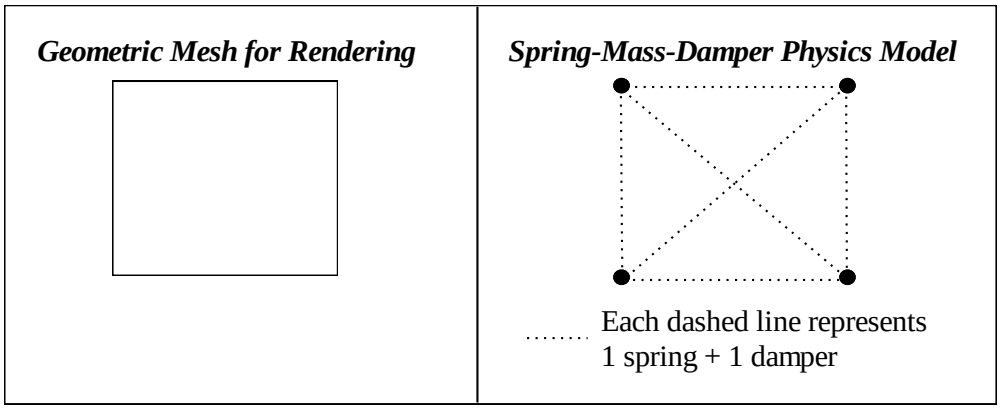



Figure 5. A simple soft-body model of a mesh, represented by a collection of particles connected by springs and dampers.

As a way of experimenting with this type of simple soft-body model, consider the following pseudo-code, which initializes the object in midair, with an initial velocity of zero. The forces acting on the particles include gravity, as well as the spring and damper forces.

Listing 8. A simple spring-mass-damper soft-body dynamics system.

```
void main()
{
     N = number of particles; // # of verts in visual model
     Vector3D cur_S[2*N];     // S(t+delta_t)
     Vector3D prior_S[2*N];   // S(t)
     Vector3D S_derivs[2*N];  // dS/dt
     Vector3D g(0.0, 0.0, -9.81);  // gravity
     float mass[N];           // mass of particles
     float k[N][N]; // spring constant between particles
     float lrest[N][N];       // spring rest lengths
```

```
float c[N][N]; // damper constant between particles
float delta_t = 0.02;    // physics time step, seconds
float game_time;         // current game time, seconds
float prev_game_time; // game time at previous frame
float physics_lag_time=0.0; // time since last update
Float init_height;  // initial height above the ground

// initialize the particles
for (i = 0; i < N; i++)
{
     mass[i] = 1.0f;     // mass = 1 kg

     // set initial linear momentum to be zero
     cur_S[2*i] = Vector3D(0,0,0);

     // assign initial position from visual model
     cur_S[2*i+1] = position of model vertex i;

     // update the initial position to reflect
     // init_height
     cur_S[2*i+1].z += init_height;
}

// initialize a spring and damper between every pair
// of particles.

for (i = 0; i < N; i++)
{
     for (j = 0; j < N; j++)
     {
          // configure the spring
          k[i][j] = 10.0f;// stiffness = 10 N/m

          // configure the damper
          c[i][j] = 0.1f; // damping coef = 0.1 kg/s

          // configure the rest length
          lrest[i][j] = length of vector
                         (cur_S[2*j+1] -
                          cur_S[2*i+1]);
     }
}

// runtime loop, derived from Listing 6.
while (main game loop)
```

```
     {
          update game_time;
          physics_lag_time += (game_time – prev_game_time);
          while (physics_lag_time > delta_t)
          {
               DoPhysicsSimulationStep(delta_t);
               physics_lag_time = physics_lag_time –
                                  delta_t;
          }
          prev_game_time = game_time;

          // once physics has updated the particle
          // positions, we must transfer these to the
          // visual model for rendering.
          for (i = 0; i < N; i++)
          {
               update visual model vertex i position to
               be cur_S[2*i + 1];
          }

          render the visual model;
     }
}

void DoPhysicsSimulationStep(delta_t)
{
     See the listings in the printed text.

     For better stability, modify the code to use
     velocity-less Verlet integration for the position
     updates and explicit Euler to update particle
     velocities.
}

Vector3D CalcForce(i)
{
     Vector3D d, SForce, DForce, RelativeVel;
     Vector3D Force_Net = 0.0f;

     // Initialize the net force by calculating the
     // force due to gravity, the particle's weight.
     Force_Net += mass[i] * g;

     // compute the spring and damper forces
     for (j = 0; j < N; j++)
```

```
    {
        // compute unit vector from particle i
        // to particle j, and the current length
        // of the spring
        d = cur_S[2*j+1] - cur_S[2*i+1];
        length = d.Length();

        d.Normalize(); // Make d unit length

        // compute the spring force using equation 20.
        // i is attracted to j if the current length is
        // greater than the rest length, repelled from j
        // if the current length is less than rest length
        SForce = k[i][j] * (length – lrest[i][j]) * d;

        // compute the damping force. First we need the
        // relative velocity
        RelativeVel = (cur_S[2*j]/mass[j]) –
                          (cur_S[2*i]/mass[i]);

        // from here, we calc the damping force using
        // equation 21. If object j is moving away from
        // object i, the force on object j draws i
        // towards j, otherwise the force repels i away
        // from j.
        DForce = c[i][j] * RelativeVel.DotProduct(d) * d;

        // increment the net force
        Force_Net += SForce;
        Force_Net += DForce;
    }

    return(Force_Net);
}
```

Jeff Lander [Lander99] has created a simple demo program based on the approach outlined in the listing above. Jeff's code is available for download on the Internet. Although imperfect, it is something you can use as a comparative example. Certainly, read Lander's article to find out what lessons he learned.

# Rotational Motion

The physics of rotational motion are analogous to the kinematics of particle or center-of-mass translational motion. Torque, the analog of force, causes an angular acceleration.

Angular acceleration causes a change in angular velocity. Angular velocity causes a change in orientation, the rotational analog of position. We begin our analysis of rotational motion with Equation 26, the rotational analog to Equation 8.

$$\frac{d}{dt}\mathbf{L}(t) = \boldsymbol{\tau}(t) \tag{26}$$

Here, the vector $\boldsymbol{\tau}(t)$ is the *torque*, sometimes called the *moment*, or *moment of force*. *Torque is measured in units of type force times distance. The SI units for torque are Newton-meters.* Torque is calculated at a point about which an object is expected to rotate, and is related to a force applied to the object. Torque is nonzero when the force acts along a line that does not intersect the point where torque is being calculated. Equation 27 gives the mathematical definition of torque, where **r** is the vector from the point about which torque is being calculated to the point where the force causing torque is being applied. From the Equation, note that torque has a direction that is perpendicular to the force vector and **r**.

$$\boldsymbol{\tau} = \mathbf{r} \times \mathbf{F} \tag{27}$$

Figure 6 illustrates the generation of torque about the center-of-mass of a rigid body, due to a force, **F**, applied at a point, **P**, on the body. Here, **r** is the vector from the center-of-mass to the point **P**.

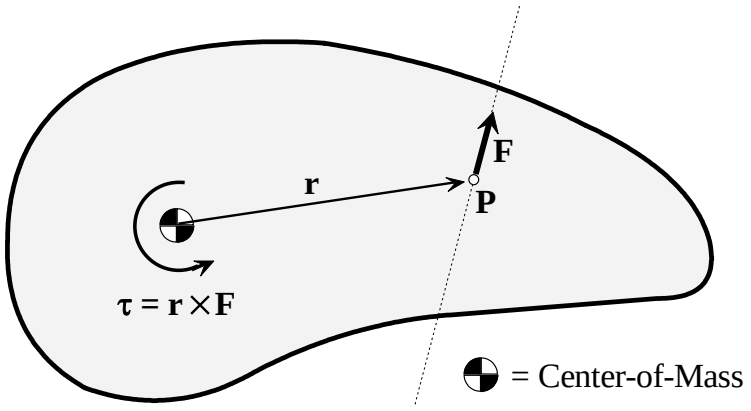



Figure 6. The relationship between force and torque. The arc arrow indicates that the torque due to **F** causes a counterclockwise rotation, *e.g.*, a positive rotation about a vector pointing out of the page, the direction of the torque vector.

Before continuing, let's consider the effect of torque on an object. We would like the concept to be somewhat intuitive. The effect of torque is fairly intuitive when it acts on an object that initially is not rotating, and so we will consider that case here. Consider the classic children's outdoor play toy, the seesaw. A seesaw in play rotates back-and-forth about the center fulcrum. If you approach a seesaw that is not in use, it will be resting, not rotating. When you sit on the end of the seesaw, your weight acts straight down near

one end, and the vector **r** from the fulcrum to your body points in a generally horizontal direction. Your weight results in a torque about the center of the fulcrum that acts to one side, parallel to the ground and perpendicular to the seesaw and your weight vector. It happens that the torque applied is exactly parallel to the axis of rotation about the fulcrum. This, in a nutshell, is the effect of torque on an object that is not initially rotating: the torque causes the object to begin rotating about the torque axis. For objects that are rotating with fairly small angular velocities, the result is similar: large torques change the axis of rotation to be approximately the torque axis. Objects with large angular momentum exhibit behavior that is less intuitive, called *gyroscopic precession*, in response to an applied torque, and even respond in a non-intuitive fashion when no torque at all is being applied (*torque-free precession*). We will not discuss these high angular momentum behaviors here.

If an object is moving freely in space, torque is calculated at the center-of-mass, and rotation is about the center-of-mass. If an object is constrained in some way, torque should be calculated about some other point that represents the possible axis of rotation. For example, if an object is constrained by a hinge, torque should be measured at a point along the hinge axis. In this chapter, we are only concerned with torque about the center-of-mass.

The vector, **L**(*t*), is called *angular momentum*, the rotational analog to linear momentum. Equation 28 states the mathematical definition of angular momentum. *Angular momentum is measured in units of type mass times distance squared per unit time. The SI units for angular momentum are kilogram-meters-squared per second (kg-m$^2$/s).*

$$\mathbf{L} = \mathbf{J}\boldsymbol{\omega} \tag{28}$$

The variable **J** is a symmetric 3x3 matrix called the *inertia tensor*, the analog of mass. The terms of the inertia tensor describe the distribution of mass throughout the volume of a rigid body. (**J** is represented by the variable **I** in many texts; however, we use **J** here to avoid confusion with the identity matrix.) Equation 29 defines the inertia tensor, which is measured in world coordinates.

$$\mathbf{J} = \begin{bmatrix} J_{xx} & J_{xy} & J_{xz} \\ J_{xy} & J_{yy} & J_{yz} \\ J_{xz} & J_{yz} & J_{zz} \end{bmatrix} \tag{29}$$

The diagonal components of the inertia tensor are called *moments of inertia,* and the off-diagonal elements are called *products of inertia*. *The inertia tensor is measured in units of type mass times distance squared. The SI units for the inertia tensor are kilogram-meters-squared (kg-m$^2$).* Equations 30 and 31 define the moments of inertia and products of inertia, respectively. Here, the variables $r_x$, $r_y$, and $r_z$ are the components of a vector **r** from the object's center-of-mass to a differential element of mass in the object.

$$J_{xx} = \iiint_{Vol} \rho\left(r_y^2 + r_z^2\right)dxdydz \; ; \; J_{yy} = \iiint_{Vol} \rho\left(r_x^2 + r_z^2\right)dxdydz \; ; \; J_{zz} = \iiint_{Vol} \rho\left(r_x^2 + r_y^2\right)dxdydz \quad (30)$$

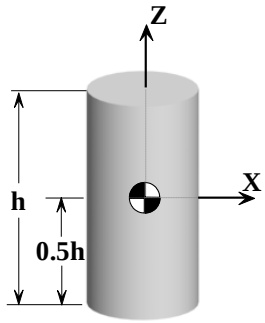
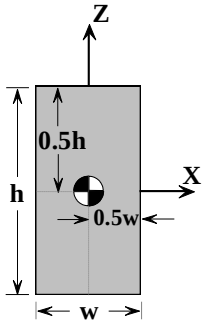
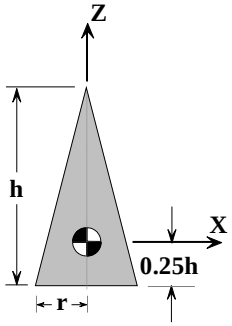
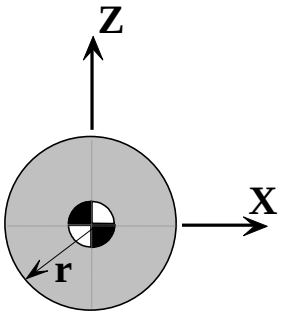
$$J_{xy} = \iiint_{Vol} \rho r_x r_y dxdydz \; ; \; J_{xz} = \iiint_{Vol} \rho r_x r_z dxdydz \; ; \; J_{yz} = \iiint_{Vol} \rho r_y r_z dxdydz \quad (31)$$

As with center-of-mass, for general-shaped objects these equations can be tedious to evaluate. Table 1 provides equations for the inertia tensor elements for a few simple shapes. The products of inertia for these objects are zero. The references [Mirtich96] and [Eberly03] provide methods for robustly evaluating the inertia tensor and center-of-mass for the arbitrary triangle meshes that are common to game development.

For physics simulation, as presented herein, the inertia tensor must be represented in the inertial reference frame, or the game's world coordinate system. If the object is rotating, the inertia tensor in this coordinate system will change as the object rotates. To avoid an expensive recomputation for every numerical integration step, it is best to compute the inertia tensor, and its inverse, $\mathbf{J}^{-1}$, in the object's local coordinate system, and then transform that tensor into world space at every integration step using Equation 32. Note that the same transformation is applied to both the tensor and its inverse, where **R** is the 3x3 object-to-world rotation matrix.

$$\mathbf{J} = \mathbf{R}\mathbf{J}_{object\_space}\mathbf{R}^T \; ; \; \mathbf{J}^{-1} = \mathbf{R}\mathbf{J}^{-1}_{object\_space}\mathbf{R}^T \quad (32)$$

Table 1. Inertia tensor values and center-of-mass locations for primitive shapes with constant density.

| Cylinder | Rectanguloid | Cone | Sphere |
|---|---|---|---|
|  |  |  |  |
| $r$ = radius<br>$J_{xx} = \frac{1}{12}m\left(3r^2 + h^2\right)$<br>$J_{yy} = J_{xx}$<br>$J_{zz} = 0.5mr^2$ | $d$ = depth<br>$J_{xx} = \frac{1}{12}m\left(d^2 + h^2\right)$<br>$J_{yy} = \frac{1}{12}m\left(w^2 + h^2\right)$<br>$J_{zz} = \frac{1}{12}m\left(d^2 + w^2\right)$ | $J_{xx} = \frac{3}{20}m\left(r^2 + \frac{h^2}{4}\right)$<br>$J_{yy} = J_{xx}$<br>$J_{zz} = \frac{3}{10}mr^2$ | $J_{xx} = J_{yy} = J_{zz} = \frac{2}{5}mr^2$ |

The orientation of an object can be represented by either $\mathbf{R}$ or by a unit quaternion, $q$. Each of these is an analog to position. The angular velocity of an object, $\boldsymbol{\omega} = \langle \omega_x, \omega_y, \omega_z \rangle$, measured in world space, is the analog of velocity. The direction of angular velocity is the parallel to the axis of the object's rotation, and its magnitude is the rate of rotation. *Angular velocity is measured in units of type angle per unit time (angle/time). The SI units for angular velocity are radians per second (rad/s).*

We can use equations 26 through 32 to perform the rotational portion of our numerical physics simulation. For a single rigid body, the state vector for explicit Euler integration, with both translational and rotational states included, can be $\mathbf{S}_i = \langle m_i \mathbf{V}_i, \mathbf{p}_i, \mathbf{L}_i, \mathbf{R}_i \rangle$, or $\mathbf{S}_i = \langle m_i \mathbf{V}_i, \mathbf{p}_i, \mathbf{L}_i, q_i \rangle$ if you choose a unit quaternion to represent the object's orientation state. The state derivative entries for $m_i\mathbf{V}_i$ and $\mathbf{p}_i$ are the same as for the no rotation case. The state derivative entry for $\mathbf{L}$ is given by Equation 26, *e.g.*, it is the net torque, $\boldsymbol{\tau}_{net}$, calculated as the sum of torques due to the applied forces in Equation 33.

$$\boldsymbol{\tau}_{net} = \left( \sum_{i=1}^{N_{springs}} \mathbf{r}_{spring,i} \times \mathbf{F}_{spring,i} \right) + \left( \sum_{i=1}^{N_{dampers}} \mathbf{r}_{dmp,i} \times \mathbf{F}_{dmp,i} \right) + \mathbf{r}_{contact} \times \mathbf{F}_{friction} + \cdots \qquad (33)$$

Equation 34 defines the value of the state derivative of $\mathbf{R}$ when the orientation state is represented as an object-to-world rotation matrix, and Equation 35 defines the state derivative of $q$ when the orientation state is represented by a unit quaternion. Take care to note the special representation of the angular velocity as a quaternion with its real component equal to 0.0.

$$\frac{d}{dt}\mathbf{R}(t) = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \mathbf{R}(t) \qquad (34)$$

$$\frac{d}{dt}q(t) = \frac{1}{2}\omega(t)q(t), \quad \text{with} \quad \omega = 0 + i\omega_x + j\omega_y + k\omega_z \qquad (35)$$

With the state vector and state derivative vector in place, you can use your favorite numerical integrator to solve for the updated state vector.

There are two post-processing steps that you must perform to properly simulate rotation. First, since the state derivative of orientation, given by Equation 34 or 35, requires that we know the angular velocity, it is necessary to compute the angular velocity *after* the integration step, by solving Equation 28 for the angular velocity and transforming the

object space inertia tensor into world space using the second part of Equation 32. This step is given below as Equation 36.

$$\boldsymbol{\omega}(t+\Delta t) = \mathbf{R}(t+\Delta t)\mathbf{J}^{-1}_{object\_space}\mathbf{R}^{T}(t+\Delta t)\mathbf{L}(t+\Delta t) \quad (36)$$

Once the updated angular velocity is known, you will be able to prepare for the next integration step.

Second, a correction step must be applied to **R** or *q* every few frames. When simulating rotation, **R** is expected to be orthogonal and *q* is expected to be unit length. Floating point round off and truncation error will cause these to drift over time. Every few frames, you must ensure that **R** is orthogonal by performing a Gram-Schmidt orthonormalization ([Eberly04], [Golub96]) or perform a Euclidian normalization of *q*.

## *The Simulation Loop with Support for Rigid Body Rotation*

As discussed above, the simulation loop for rigid body dynamics with rotational motion is somewhat more complex than pure translational motion. Listing 9 outlines the steps required to configure such a system, and illustrates the simulation loop without collision detection and response. As in prior code listings, comments within the code will clarify the process. Note that in practice collisions must be handled in a manner similar to Listing 5.

Listing 9. Simulation Loop with Rotation

```
void main()
{
     N = number of rigid bodies
     Matrix33 JObj[N];   // inertia tensors in object space
     Matrix33 JObjInv[N];     // inverse inertia tensors in
                              // object space
     Matrix33 J;              // temporary inertia tensor
                              // in world space
     Float mass[N];           // rigid body masses
     Vector3D cur_S[3*N];     // velocity, position, and
                              // angular momentum states
     Quaternion cur_q[N];     // orientation states as
                              // quaternions
     Vector3D prior_S[3*N];   // prior vel, position,
                              // angular momentum states
     Quaternion prior_q[N];   // prior orientation state
     Vector3D S_derivs[3*N];  // dS/dt for vel, pos,
                              // angular mom
     Quaternion q_derivs[N];  // dS/dt for orientation
```

```
Vector3D cur_w[N];       // current angular velocities
matrix33 R;              // temporary rotation matrix
int iCounter = 0;        // counter to tell us when to
                         // renormalize the
                         // orientation quaternion

// initialize the rigid bodies
for (i = 0; i < N; i++)
{
     mass[i] = mass of rigid body i;

     // The initial center-of-mass position and
     // linear momentum are the same as the
     // translational only case.
     cur_S[3*i] = initial linear momentum of i
     cur_S[3*i+1] = initial position of i's
                          center-of-mass;

     // The rotational state variables are dependent
     // on the inertia tensor, so we calculate that
     // here.

     JObj[i] = compute inertia tensor of rigid body i,
               in the local object space of i;

     // since we need it later, compute and store the
     // inverse inertia tensors.

     JObjInv[i] = JObj[i].Inverse();

     // Set the initial orientation of object i. This
     // is a quaternion represented in world space.

     cur_q[i] = current orientation as a unit
                    quaternion;

     // the initial angular velocity is here assumed
     // to be zero, but it might be nonzero.

     cur_w[i] = Vector3D(0,0,0);

     // compute initial angular momentum given angular
     // velocity using equation 28 and 32. This
     // requires that we first compute a 3x3 rotation
     // matrix cur_q[i].
```

```
        R = cur_q[i].ConvertToRotationMatrix();

        // From here we can compute the initial world
        // space inertia tensor using Equation 32

        J = R * JObj[i] * R.Transpose();

        // Now we can compute the initial angular
        // momentum using Equation 28.
        cur_S[3*i+2] = J * cur_w[i];
    }

    // Game simulation/rendering loop
    while (game simulation is running)
    {
        update game_time;

        // update the physics
        physics_lag_time += (game_time - prev_game_time);
        while (physics_lag_time > delta_t)
        {
            DoPhysicsSimulationStep(delta_t);
            physics_lag_time = physics_lag_time -
                                        delta_t;
        }
        prev_game_time = game_time;

        // occasionally renormalize the orientation
        // quaternion
        if (++iCounter == 5)
        {
            iCounter = 0;
            for (i = 0; i < N; i++)
                cur_q[i].Normalize();
        }

        // render the scene
        for (i = 0; i < N; i++)
            Render rigid body i at position
                cur_S[3*i+1],
                and orientation cur_q[i];
    }
}
```

```
// Update the physics
void DoPhysicsSimulationStep(delta_t)
{
     copy cur_S to prior_S
     copy cur_q to prior_q

     // temp_w is a temporary quaternion version of the
     // angular velocity, as shown in Equation 35.
     Quaternion temp_w;

     // calculate the state derivative vectors
     for (i = 0; i < N; i++)
     {
          // state derivative for translational
          // linear momentum and position are the
          // same as for the non-rotation case
          S_derivs[3*i] = CalcForce(i);
          S_derivs[3*i+1] = prior_S[3*i] / mass[i];

          // state derivative for angular momentum
          // is the net torque, given by Equations
          // 26 and 33.
          S_derivs[3*i+2] = CalcTorque(i);

          // state derivative for the orientation
          // is given by Equation 35.
          temp_w.Set(0, cur_w[i].x, cur_w[i].y,
                        cur_w[i].z);

          q_derivs[i] = 0.5 * temp_w * cur_q[i];
     }

     // integrate the equations of motion. Vector
     // states first.
     ExplicitEuler(3*N, cur_S, prior_S, S_derivs, delta_t);

     // Followed by the quaternion orientation state
     ExplicitEuler(N, cur_q, prior_q, q_derivs, delta_t);

     // We are not done yet. We have the updated state, but
     // we need to compute the new angular velocity using
     // Equation 36. We do this in a loop since it has to
     // be done for all objects
     for (i = 0; i < N; i++)
     {
```

```
            // We first need a rotation matrix for time
            // t + delta_t.
            R = cur_q[i].ConvertToRotationMatrix();

            // We next compute the inverse inertia tensor
            // in world space, which is used in Eq. 36.
            J = R * JObjInv[i] * R.Transpose();

            // We are now in a position to update the
            // angular velocity using Equation 36.
            cur_w[i] = J * cur_S[3*i+2];
        }

        // Now we're done with the integration!

        // By integrating the equations of motion, we have
        // effectively moved simulation time forward by
        // delta_t.
        t = t + delta_t;
}
```

## *A Brief Word About Integrators and Different State Variable Types*

If you look carefully at Listing 9, you will see that the call to *ExplicitEuler* looks identical regardless of whether or not the state vectors and derivatives contain Vector3D objects or Quaternion objects. And you may wonder how you can actually create a single integrator function that can integrate state variables that are of different types. There are a couple of ways to accomplish this. One approach is to flatten all state variables, *e.g.*, vectors and quaternions, into an array of floating point values, and use an integrator that simply integrates an array of floating point state values. Another approach, using object-oriented programming, is to derive all state variables from a base State class, and ensure that all concrete state classes overload the operators required by the integrator: -, +, and *.

Let's take a look at the first approach, in which the state variables are flattened to an array of floats. A Vector3D object can be represented as an array of 3 floats, and a Quaternion object can be represented as an array of 4 floats. Listing 10 shows how you might represent a collection of object states that follow this approach, along with the call to *ExplicitEuler*. The state variable vector includes $m\mathbf{V}$, $\mathbf{p}$, $\mathbf{L}$, and $q$ in a single floating point array.

Listing 10. Object States Flattened Into an Array of Floating Point Values

```
float cur_S[13*N];       // current state
float prior_S[13*N];     // prior state
float S_derivs[13*N];    // state derivatives
```

```
// for object i
cur_S[13*i + 0] = linear momentum x component;
cur_S[13*i + 1] = linear momentum y component;
cur_S[13*i + 2] = linear momentum z component;
cur_S[13*i + 3] = position x component;
cur_S[13*i + 4] = position y component;
cur_S[13*i + 5] = position z component;
cur_S[13*i + 6] = angular momentum x component;
cur_S[13*i + 7] = angular momentum y component;
cur_S[13*i + 8] = angular momentum z component;
cur_S[13*i + 9] = orientation real component;
cur_S[13*i + 10] = orientation imaginary i component;
cur_S[13*i + 11] = orientation imaginary j component;
cur_S[13*i + 12] = orientation imaginary k component;

prior_S[13*i + 0] through prior_S[13*i + 12] is similar;

S_derivs[13*i + 0] = Net force x component;
S_derivs[13*i + 1] = Net force y component;
S_derivs[13*i + 2] = Net force z component;
S_derivs[13*i + 3] = Velocity x component;
S_derivs[13*i + 4] = Velocity y component;
S_derivs[13*i + 5] = Velocity z component;
S_derivs[13*i + 6] = Net torque x component;
S_derivs[13*i + 7] = Net torque y component;
S_derivs[13*i + 8] = Net torque z component;
S_derivs[13*i + 9] = 0.0;
S_derivs[13*i + 10] = angular velocity x component;
S_derivs[13*i + 11] = angular velocity y component;
S_derivs[13*i + 12] = angular velocity z component;

// The following call integrates the vector values and
// quaternion values all in one call
ExplicitEuler(13*N, new_S, prior_S, S_derivs, delta_t);
```

Using an object-oriented approach, the integrator parameters would be specified as arrays of a base object class type, then derive all state variable types from the base class. The base class in this case must define pure virtual functions for the basic mathematical operators -, +, and *, and the state variable classes must each provide concrete implementations of those operators. If you choose to use C++, and store the states in STL vectors, then the state variable classes will also need to provide assignment operators and copy constructors to make the STL container classes happy. Listing 11 shows example code in pseudo-C++.

Listing 11. Object-oriented State Classes and Integrator

```
class State
{
     const State &operator+(const State &Other) = 0;
     const State &operator-(const State &Other) = 0;
     const State &operator*(const State &Other) = 0;
     const State &operator*(const float fFactor) = 0;

};

class Vector3D : public State
{
     const Vector3D &operator+(const Vector3D &Other);
     const Vector3D &operator-(const Vector3D &Other);
     const Vector3D &operator*(const Vector3D &Other);
     const Vector3D &operator*(const float fFactor);

     float m_fX, m_fY, m_fZ;  // components of vector
};

class Quaternion : public State
{
     const Quaternion &operator+(const Quaternion &Other);
     const Quaternion &operator-(const Quaternion &Other);
     const Quaternion &operator*(const Quaternion &Other);
     const Quaternion &operator*(const float fFactor);

     float m_fR;              // real component
     float m_fi, m_fj, m_fk;  // imaginary components
};

void main()
{
     ArrayContainer<State> cur_S;
     ArrayContainer<State> prior_S;
     ArrayContainer<State> S_derivs;
     for (i = 0; i < N; i++)
     {
          cur_S.Add(Vector3D(linear momentum of i));
          cur_S.Add(Vector3D(position of i));
          cur_S.Add(Vector3D(angular momentum of i));
          cur_S.Add(Quaternion(orientation of i));

          // prior_S and S_derivs follow similarly
```

```
     }

     // in the simulation loop, just call the integrator
     // by passing references to cur_S, prior_S, and
     // S_derivs.
}

void ExplicitEuler(ArrayContainer<State&> &new_S,
     const ArrayContainer<State&> &prior_S,
     const ArrayContainer<State&> &S_derivs, delta_t)
{
     unsigned int N = new_S.size();
     for (i = 0; i < N; i++)
     {
          new_S[i] = prior_S[i] + delta_t * S_derivs[i];
     }
}
```

There certainly are other ways in which you might build an object-oriented numerical simulator; however, one benefit of the approach shown in Listing 11 is that the numerical integrator method need only exist in one location in your source code. This makes the code fairly easy to maintain.

# Collision Response Revisited

Now that we understand how to simulate the motion of objects undergoing rotational motion, it is worth revisiting impulse-momentum-based collision response. Figure 7 illustrates a generalized, frictionless rigid body collision. Because there is no friction, the collision impulse acts through the point of impact in the direction of a unit normal vector at the point of impact. As shown in the figure, the line of action of the impulse does not necessarily intersect the centers-of-mass of the objects involved, resulting in an impulsive torque that changes the rotational motion state of the objects.

The rotational analog to the linear impulse-momentum equation is the *angular impulse-momentum equation*, given by Equation 37 for frictionless collisions between two objects.

$$\mathbf{L}_1^+ = \mathbf{L}_1^- + \mathcal{J}(\mathbf{r}_1 \times \hat{\mathbf{n}}) \;\; ; \quad \mathbf{L}_2^+ = \mathbf{L}_2^- - \mathcal{J}(\mathbf{r}_2 \times \hat{\mathbf{n}}) \tag{37}$$

To compute generalized frictionless collision response, we must solve Equations 9-12 and 37 for the impulse. Equation 38 gives the resulting impulse value, which models both translational and rotational effects. The impulse vector is given by Equation 11.

To compute the post-collision linear and angular momentums, simply apply the impulse from Equations 38 and 11 to Equations 9, 10, and 37. Note that Equation 38, when substituted into Equation 11, simplifies to Equation 13 when the line joining the two centers-of-mass intersects the impact point and is parallel to the contact normal, since all the cross products in Equation 38 become zero. This is to be expected, since there is no impulsive torque in this case.

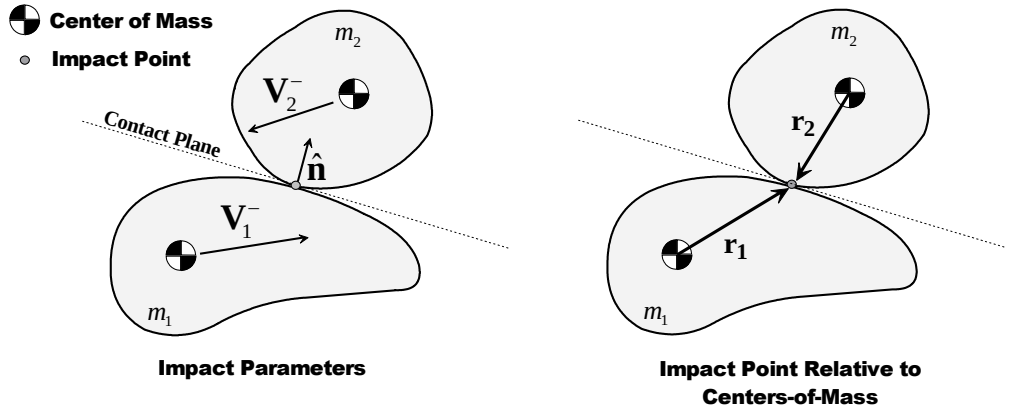



Figure 7. Generalized frictionless collision.

$$\Lambda = \frac{-(1+\varepsilon)\left(\hat{\mathbf{n}}\cdot\left(\mathbf{V}_1^- - \mathbf{V}_2^-\right) + \boldsymbol{\omega}_1^- \cdot \left(\mathbf{r}_1 \times \hat{\mathbf{n}}\right) - \boldsymbol{\omega}_2^- \cdot \left(\mathbf{r}_2 \times \hat{\mathbf{n}}\right)\right)}{\frac{1}{m_1} + \frac{1}{m_2} + \left(\left(\mathbf{r}_1 \times \hat{\mathbf{n}}\right)^{\mathrm{T}} \mathbf{J}_1^{-1}\left(\mathbf{r}_1 \times \hat{\mathbf{n}}\right) + \left(\mathbf{r}_2 \times \hat{\mathbf{n}}\right)^{\mathrm{T}} \mathbf{J}_2^{-1}\left(\mathbf{r}_2 \times \hat{\mathbf{n}}\right)\right)} \tag{38}$$

# Bringing It All Together

You now should have a good understanding of the major factors that cause and affect the motion of general rigid bodies, as well as a basic understanding of the numerical integration techniques that can be used to approximately solve for the motion of a collection of objects over time. Whether you are implementing simple spherical particle physics or general rigid body physics, your basic process is rather straightforward, as outlined below:

- Choose a numerical integrator (or integrators), which will determine the state vector and the number of prior states you must track.
- Choose a representation for the state vector, **S**, and derivative vector, $d\mathbf{S}/dt$ and/or $d^2\mathbf{S}/dt^2$.
- Choose a set of initial conditions for your rigid bodies, and assign these into **S**.
- If your selected numerical integrator requires more than one prior state, use explicit Euler integration to initialize the entire set of prior states. Alternatively, initially set all prior states equal to the initial conditions.

- During the simulation for each physics step, if collisions are detected and you are using impulse-momentum-based collision response, resolve them using the instantaneous impulse-momentum equations and update the state properties for the objects involved before continuing with the general update of all objects. Alternatively, if you are using penalty-force-based collision response, compute the penalty force on objects involved in collisions based on current interpenetration depths at the time *t*, and add the penalty force to the net force applied to each object. Then, update all objects in one step, including those in collision.
- During the simulation for each physics step:
    - Copy the current state $\vec{S}$ into a temporary state array to be given to the integrator as one of the prior states. Copy other prior states if necessary.
    - Calculate or copy the state derivative vector for each object.
    - Call the integrator(s) to update the state vector.
    - If simulating rotational motion, update the angular velocity every frame, and renormalize the orientation matrix and/or orientation quaternion every few frames.

# Additional References


[Eberly03] Eberly, David, "Polyhedral Mass Properties (Revisited)," http://www.magic-software.com/Documentation/PolyhedralMassProperties.pdf, January 2003.

[Eberly04] Eberly, David, *Game Physics*, Morgan Kaufmann, 2004.

[Golub96] Golub, Gene H., and Charles F. Van Loan, *Matrix Computations Third Edition*, Johns Hopkins, 1996.

[Lander99] Lander, Jeff, "Collision Response: Bouncy, Trouncy, Fun," article in *Game Developer Magazine*, article available online at http://www.gamasutra.com/features/20000208/lander_01.htm, code available online at http://www.gdmag.com/src/mar99.zip, March 1999.

[Mirtich96] Mirtich, Brian, "Fast and Accurate Computation of Polyhedral Mass Properties," *Journal of Graphics Tools*, Volume 1, Issue 2, February 1996. (Online at http://www.cs.berkeley.edu/~jfc/mirtich/massProps.html)

[Rhodes05] Rhodes, Graham, "Back of the Envelope Aerodynamics for Game Physics," *Game Programming Gems 5*, Charles River Media, 2005.