# Rigid Body Motion
# An Introduction

**by**

**Brent M. Dingle**
Texas A&M University
Fall 2004

## Abstract

This paper is a brief introduction to modeling rigid body motion. It is intended for the beginning computer scientist who is seeking to learn more about how to write programs that model the physical reality of the world. Most of the concepts are based on those presented in the course at Siggraph99 by David Baraff. The advantage this paper offers is a summary of a full implementation written in C++ based pseudocode. In particular it will circumvent the necessity for a quadratic problem solver to establish rest contacts for simple contact scenarios. In doing this explicit methods will be described detailing how to use impulse forces to keep an object in a rest state. In describing this a large amount of detail is given to the procedures used in updating the system. It is worth noting that these methods assume a very simple system of rigid bodies. As the system becomes more complex, more complex methods may be necessary. The material here should serve only as a starting point, not a final project.

# 1 Introduction

The topic of rigid body motion is a common one in computer graphics. It is used in a variety of fields for modeling and entertainment. This paper is intended for the beginning computer scientist who is seeking to learn more about how to write programs that model the physical reality of the world.

The material presented here is a brief outline of C++ based pseudocode that should prove useful in the creation of a simple physics based modeler involving rigid body motion. This basis should be sufficient to begin coding regardless of the intent of the programmer, whether it is for scientific or entertainment purposes will be of little consequence. The code herein is common among most all the tasks and should serve as a good starting point for implementing a program to model rigid body motion. It is expected that the reader is familiar with the Siggraph99 course notes of David Baraff [Bar1999]. It is further assumed the reader has already created a math library containing the ability to create and use vectors of length three along with 3x3 matrices and quaternions.

# 2 The World

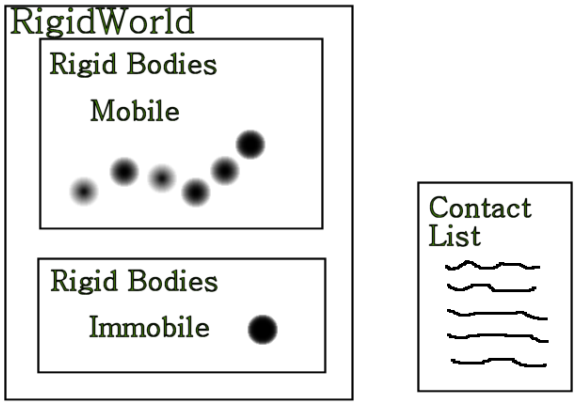
Writing a program to simulate anything requires a model of that which is to be simulated. For now we want to model objects in the real world. While drawing pictures of such things is a model, it is not good enough for us. We want to actually create objects that look and behave as they would in the real world. Unfortunately there are a great many objects and types of objects in the real world so sadly we must limit ourselves.

While gases and liquids and people and fire and smoke and such things are interesting the most common object we come into contact with can usually be classified as a rigid body. These are solid objects that do not bend or change form easily, such as a chair, a TV, a floor or a wall. Notice these objects do not bend or flap, thus clothing and paper do not count. So with this restriction to solid objects we will be excluding some types of objects, however we will still be able to model a great many.

With this in mind we will now begin to layout a data structure. As we must have a world we will call our world RigidWorld. Within this world will be RigidBodies.

Having this picture in mind we now begin to think of how the rigid bodies must interact. Clearly they will come into contact with each other, so perhaps we will need a way to represent contact points. It is also clear that some objects, such as the earth are not likely to react to smaller things, such as a chair. Or in a similar fashion the reaction between two balls colliding is likely to be different than a ball hitting a wall. Specifically each ball's state will likely be altered by a collision between them, however a wall's state is not likely to be altered by a (small) ball colliding with it. So this might give us an idea that we have two types of objects, mobile and immobile.

To summarize we will draw a picture of how things might be modeled in the most general way:




Notice the contact list would be a list of which objects are in contact with what other objects. This model is about as generalized as you can get. Now we need to flesh out how the objects will be described and how they will actually behave and why.

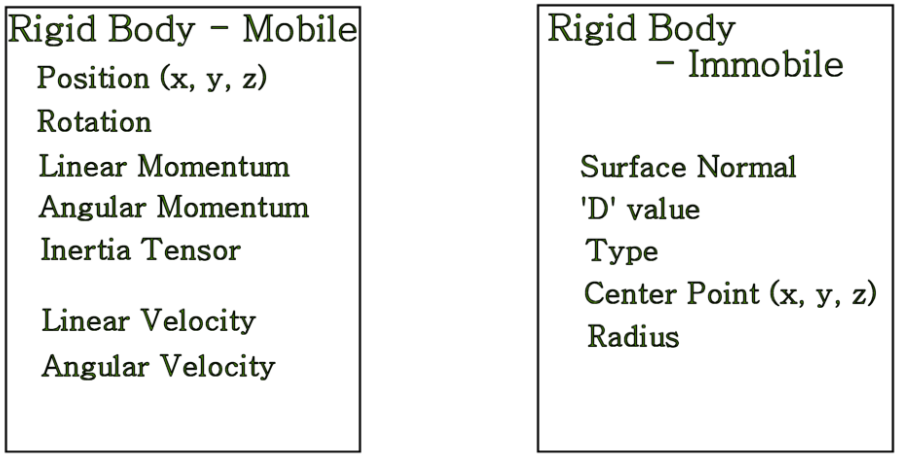
# 3 The Objects

Having a description of our world, we must now find a way to describe our objects. Clearly every object must have a position. Just as clearly it must have an orientation (i.e. which direction does it face). It also seems likely that the mobile objects may have a velocity, as they can move, and most objects have a mass (or weight). Of course we can also say an object has a color, a texture and may be reflective, but for now we will not worry about such "trimmings." All of these properties can be summarized as the *state* of the object.

Having listed all the obvious properties of an object we must remember some simple physics to complete our property list. Since we know some objects can move, something must be making them move (an object at rest tends to stay at rest…). So perhaps we can associate a list or a sum of forces acting on each object into the object's state. Thus we start reading some papers [Bar1999] and learn that the momentum of an object can be used in such a way. In this reading we also learn that for angular momentum we will need an Inertia Tensor which describes how the mass of the body is distributed. This also reminds us there are two types of velocity: angular and rotational.

Of course our immobile objects will be slightly different. For the most part these objects will be things such as floors and walls. Thus they all are planes. Recalling that a plane can be represented by the equation $Ax + By + Cz + D = 0$, where (A, B, C) is the normal of the plane and D will identify where in space the plane is, all we need for our immobile objects are the surface normal and the 'D' value. For simplicity we will think of them as infinite in all directions, but note they need not be. To indicate this we will add a center point on the plane and a radius. We also include a type to denote whether it is a circular or square plane, with radius as given or side length equal to 2*radius.

This seems like a fairly complete list of properties of an object. To summarize we can refer to the below pictures for the model we will use for the *state* of our two types of rigid bodies:




# 4 Contact – Algorithm

It would seem that we now have a functional representation of the objects in our world. Yet we have not thought about how the objects might interact. Looking back at section 2 we see we have a contact list to represent which objects are in contact with which objects, but how will we determine what should be in this list?

This is a very complicated subject area with many different answers. In the simplest terms the contact list must be able to provide the following information for any given contact between two objects:

- which two objects are in contact
- at what point (and edge and face) are they in contact
- at what angle did they meet (what is the normal of the contact point)
- what type of contact is it (vertex-face, edge-edge, etc)

For simplicity we will assume all of our mobile objects have bounding boxes, in fact we will assume they are actually boxes. We will also not concern ourselves with the collision of mobile objects with mobile objects. Thus the only contact that will be possible will be a vertex-face contact and the contact points will be where the vertices of our mobile objects come into contact with the plane(s) of our immobile objects. This is for ease of discussion, all the methods that will be described in this paper will function correctly if the above contact information is accurately maintained and used appropriately. Recall the intent of this paper is an introduction. If this simple collision type can be dealt with by the physics engine, it should be able to deal with any type of collision, as long as the collision is detected and identified correctly.

With the assumptions we have made our collision detection routine is easy to implement. This will not likely be the case in a fully functional modeler, but for the purpose of illustration it is sufficient. The general algorithm would be something like:

```
 1. Set the collision state as clear.
 2. For each mobile object, A
 3.   For each immobile object, B
 4.      For each vertex of A check
 5.        Calculate the distance from the vertex to the immobile plane
 6.            if the distance < -epsilon then
 7.              record the contact information
 8.              and flag a penetrating state
 9.           else if (–epsilon <= the distance < epsilon) then
10.              record the contact information
11.              if (not in a penetrating state) then
12.                 flag a colliding or rest state (depending on vel)
13.              else if (the distance > epsilon) and  (not in a
                 penetrating nor collision nor rest state) then
14.                 flag a clear state
```

Of course there are more optimal approaches but the above technique is straightforward and should be easy to implement. As long as the number of mobile objects, *m*, and the number of immobile objects, *n*, is small then the O(*m***n*) running time will not be a large problem. More details of the above algorithm will be given later.

# 5 Data Structures

With our world and its objects defined we may now present some C++ class-like structures to represent our world. These are very similar to those presented in [Bar1999], this is so those notes can be used to expand upon these:

```
CrigidBody
{
   // The constant quantities
   double      mass;  // likely in grams or kilograms
   Cmatrix3x3  Ibody; // inertia tensor of the rigid body

   // The state quantities
   Cvector3    pos;   // position of center of mass
   Cquaternion q;     // rotation as a quaternion
   Cvector3    P;     // linear momentum
   Cvector     L;     // angular momentum

   // The derived quantities
   Cmatrix3x3  I_inverse;  // inverse of current I
   Cvector3    vel;        // linear velocity
   Cvector3    omega;      // angular velocity

   // “artificial” quantities for drawing and collision detection
   Cvector3    vert_list[max_verts];
   Cvector3    vert_normal[max_verts];
   long        surf_triangle_list[max_triangles]
   long        num_verts, num_triangles;
}
```

```
Cimmobile
{
   Cvector3 normal;   // the A, B, C of eq: Ax + By + Cz + D = 0
   double   D;
   double   radius;
   long     type;     // circle, square, etc.

   double   sqrt_ABC; // useful in collision detection if precalc'd.
}

Ccontact
{
   long rigid_a;   // index of body A
   long rigid_b;   // negative index of immobile object B minus 1
                   // the negative will denote an immobile object
   Cvector3 point;    // The point of contact
   Cvector3 normal;   // The normal of the collision plane
   long type;         // type of contact, vert-face, edge-edge, etc.

   // As collision types increase, more may need to be added here
}

CrigidSys
{
   CrigidBody  body[max_bodies];
   Cimmobile   immobile[max_immobile_bodies];
   Ccontact    contact[max_contacts];
   long        num_bodies;
   long        num_immobiles;
   long        num_contacts;
   long        collision_state;
   double      y0[state_size * max_bodies];
   double      yfinal[state_size * max_bodies];
   double      max_time_step;
   double      cur_step;

   // Various member functions, some can be written more efficiently
   void StateToArray(CrigidBody *rb, double *y);
   void ArrayToState(CrigidBody *rb, double *y);
   void ArrayToBodies(double *y);
   void BodiesToArray(double *y);
   void ComputeForceAndTorque(CrigidBody *rb);
   void dydt(double *y, double *ydot);
   void ddt_StateToArray(CrigidBody *rb, double *ydot);
   void rk4(double *y, double *yout, long n);

   void ComputeImpulse();
   void ComputeRestImpulse();
   Cvector3 PointVelocity(long body_idx, Cvector3 point);
   void CheckForCollisions();
   void ImmobileCheck(long a, long im_idx);

   void InitStates(...);
   void Update();
   void Display(...);
}
```

# 6 Initialize

At this point we have created a model of the world we wish to simulate and we have created the data structures necessary to implement that model on a computer. What remains is the details of getting everything to actually run. You will notice we have hinted at some things to come in the member functions of CrigidSys. However, we must start at the beginning.

The first thing we will need to do is to initialize the state of all our objects. This will be done in a call to CrigidSys::InitStates(). This function likely will require a variety of parameters depending on how you want things initialized. As in many cases throughout this paper we simplify as much as possible. Here we assume InitStates takes one parameter and that is the size of the maximum time step. Specifically InitStates would be declared:

```
void CrigidSys::InitStates(double step_size)
```

Within this function we will assume at least one mobile object's state is initialized and one immobile object's state is initialized. In fact it might look something like this:

```
void CrigidSys::InitStates(double step_size)
{
   max_time_step = step_size;

   num_bodies    = 0;
   num_immobiles = 0;
   num_contacts  = 0;

   // set vertices, inertia tensor, etc to represent a cube
   body[num_bodies].MakeCube(2);   // side length is 2
   body[num_bodies].mass = 5;      // 5 kg
   body[num_bodies].pos  = <0, 3, 7>
   num_bodies++;

   // create a floor at z = 0 (the xy-plane)
   immobile[num_immobiles].normal = <0, 0, 1>  // z is up
   immobile[num_immobiles].D = 0;
   num_immobiles++;

   BodiesToArrays(yfinal);
}
```

Notice that BodiesToArrays() is assumed almost identical to the description provided in [Bar1999]. The function itself basically copies the state variables of every body into a one dimensional array. This is done so that the array might later be sent to an ODE solver (see the Update and Display section below).

# 7 Update and Display

It is assumed somewhere in the main() body of the program a timer is being run. As time passes we will expect the world to change from its initial state. This means that every *n* (micro)seconds we will call CrigidSys::Update(). This call will be followed immediately by a call to CrigidSys::Display(). Again depending what is being modeled and how the display is being done various parameters may be passed to these functions. Staying with simple, we will assume that none are necessary. Notice this concept of time is a slight deviation from the presentation given in [Bar1999], which used a simple for-loop to mark time. Unlike using an explicit for-loop we will be relying on a system timer with no predetermined stop time. So somewhere in the main program we have an OnTimer() function that is called every *n* (micro) seconds. This function might be:

```
OnTimer(int eventID)
{
   world.Update();    // assume world is type CrigidSys
   world.Display();
}
```

Our Update() function would then be:

```
void CrigidSys::Update()
{
   double delta_t;
   long   i;
   // copy the current state array into y0
   for (i = 0; i < state_size * max_bodies; i++)
      y0[i] = yfinal[i];

   cur_step = max_time_step;  // rk4 expects cur_step set correctly
   rk4(y0, yfinal, state_size * max_bodies);
   ArrayToBodies(yfinal);
   CheckForCollisions();

   if (collision_state == penetrating)
   {
      // Back step to point of collision or rest
      delta_t = 0.5 * cur_step;
      cur_step -= delta_t;
      while (collision state != colliding and != rest)
      {
         rk4(y0, yfinal, state_size * max_bodies);
         ArrayToBodies(yfinal);
         CheckForCollisions();
         delta_t *= 0.5;
         if (collision_state == penetrating) cur_step -= delta_t;
         if (collision_state == clear)       cur_step += delta_t;
      }
   } // end if penetrating state
   // Resolve the collision
   if (collision_state == colliding)
   {
      ComputeImpulse();      // directly alters the state of the bodies
      BodiesToArrays(yfinal);
   }
} // end Update
```

The functions BodiesToArrays() and ArrayToBodies() are virtually identical to those presented in [Bar1999]. Respectively they store the state of every body into an array and take an array and translate it into the state variables of each body.

Note yfinal[] and the actual state of the bodies will always be the same at the exit point of the Update() function. These are used at various points and places in the code to be backups and updates of one another. Notice also that the bodies contain more information than what is actually stored in yfinal[]. The y0[] array is only used in the Update() function which allows us to easily step back to what we know is a valid state, thus allowing the back stepping in time.

The rk4 is a Runge Kutta 4, ODE solver. The code for it would be:

```
void CrigidSys::rk4(double *y, double *yout, long n)
{
   long i;
   double xh, hh, h6;

   // Set up first derivs for current state (redundant)
   dydt(y, tmp_dydx);

   hh = cur_step * 0.5;
   h6 = cur_step / 6.0;
   xh = hh;
   for (i=0; i<n; i++)              // ------ first step
   {
      yt[i] = y[i] + hh * tmp_dydx[i];
   }

   dydt(yt, dyt);                   // ------- second step
   for (i=0; i<n; i++)
   {
      yt[i] = y[i] + hh * dyt[i];
   }

   dydt(yt, dym);                   // ------- third step
   for (i=0; i<n; i++)
   {
      yt[i] = y[i] + cur_step * dym[i]
      dym[i] += dyt[i];
   } // end for

   dydt(cur_step, yt, dyt           // ------ fourth step
   for (i=0; i<n; i++)
   {
      yout[i] = y[i] + h6 * (tmp_dydx[i] + dyt[i] + 2.0*dym[i]);
   }
} // end rk4
```

Where `tmp_dydx, dym, dyt, yt` are all arrays of size [`state_size * max_bodies`] of type double. This code is based on that presented in the book Numerical Recipes in C [Pre1987].

The dydt function is used to calculate the derivative of the state described by the y[] array sent to it. It would look something like:

```
void CRigidSys::dydt(double y[], double ydot[])
{
   long i;

   // Put data into body[]
   ArrayToBodies(y);

   for (i = 0; i < num_bodies; i++)
   {
      ComputeForceAndTorque(t, &body[i]);

      // note: when colliding some objects may be at rest too
      if ( (collision state == colliding)
            || (collision state == rest) ) // so NO penetration
      {
         ComputeRestImpulse(i);
      }

      ddt_StateToArray(&body[i], &ydot[i * STATE_SIZE]);
   }
} // end dydt
```

In the above ComputeForceAndTorque() and ComputeRestImpulse() will alter the state of body[i]. The function ddt_StateToArray will then set the portion of ydot[] that corresponds to body[i] to the necessary values to represent the rate of change of the state of body[i] based upon the forces and impulses just applied. Notice the call to ComputeRestImpulse() will only apply impulses necessary to keep a body at rest (e.g. counteract gravity to keep the object from sinking into the floor). Notice unlike the call to ComputeImpulse() in the Update() function, these rest impulses are nice enough that they can work with the ODE, and thus only a true collision state will ever ‘temporarily’ skip over the ODE.

The ddt_StateToArray() is virtually identical to that presented in [Bar1999]. Notice that throughout this paper the use of a quaternion representation has been assumed, but the simple rotation matrix should also work.

This leaves us with the details of calculating the impulse forces. However before we do that we will need to demonstrate exactly how we are thinking the contacts are being set.

# 8 Contact – Pseudocode

Previously in section 4 we presented the general algorithm we can use to establish our contact information and collision state. Here we will be more specific and present the pseudocode that could be used to implement the algorithm. Understand this is not necessarily the most efficient way to perform this operation and we have limited our types of collision to one particular kind. This is all done for simplicity and clarity. Assuming all collisions are processed in a similar manner to the one below, everything should still work. Specifically, the above Update() procedure will still produce the correct results in behavior and motion (within a given tolerance).

Here is the pseudocode for the functions to check for a collision:

```
void CrigidSys::CheckForCollisions()
{
   long a, im_idx;

   ml_NumContacts = 0;            // assume no collisions
   m_CollisionState = clear;

   // For now, only worry about floors and walls
   a = 0;
   while ((a < ml_NumActive) && (m_CollisionState != penetrating))
   {
      for (im_idx = 0; im_idx < ml_NumImmobile; im_idx++)
      {
         ImmobileCheck(a, im_idx);
      }

      a++;
   } // end while a and NO PENETRATING

} // end CheckForCollisions




void CRigidSys::ImmobileCheck(long a, long im_idx)
{
   Cvector3 tmp_vert[max verts];
   Cvector3 pt_vel;
   Cvector3 padot;
   Cvector3 point;
   double   dist;
   double   rel_vel;
   long     i;
```

```
// look at EACH and EVERY vertex of body[a] - to set COLLIDE correctly
// penetrate will just backstep time, so only 1 penetrate matters,
// collides are special as are rests.
   for (i=0; i < body.num_verts; i++)
   {
      // RotMat is set in ArrayToState() via ArrayToBodies()
      // It may also be set by: RotMat = body[a].q.MakeMatrixFromQ();
      tmp_vert[i] = body[a].RotMat * body[a].vert_list[i];
      tmp_vert[i] += body[a].pos;
   }

   for (i=0; i < body[a].num_verts; i++)
   {
      point = tmp_vert[i];
      dist = immobile[im_idx].PointDist(point);

      if (dist < -DEPTH_EPSILON)
      {
         contacts[num_contacts].rigid_a = a;
         contacts[num_contacts].rigid_b = -im_idx - 1;
         contacts[num_contacts].point = point;
         contacts[num_contacts].normal = immobile[im_idx].normal;
         contacts[num_contacts].type = vert_face;
         collision_state = penetrating;
         num_contacts ++;
      }
      else if ((dist < depth_epsilon)
               && (collision_state != penetrating))
                  // don't overwrite penetrate with collide
      {
         contacts[num_contacts].rigid_a = a;
         contacts[num_contacts].rigid_b = -im_idx - 1;
         contacts[num_contacts].point = point;
         contacts[num_contacts].normal = immobile[im_idx].normal;
         contacts[num_contacts].type = vert_face;

         // Now is it colliding or just resting?
         padot = PointVelocity(a, contacts[num_contacts].point);
         rel_vel = contacts[num_contacts].normal * padot;
         if (rel_vel < -velocity_thresh) // colliding
         {
            collision_state = colliding;
         }
         else if ((rel_vel < velocity_thresh)
                  && (collision_state != colliding))
         {
            collision_state = rest;
         }

         num_contacts++;
      }

   } // end for i
} // end ImmobileCheck
```

Notice that if RotMat is not a member of our CrigidBody class then we will need to create a temporary variable of type Matrix3x3 named RotMat and set it to be body[a].q.MakeMatrixFromQ(). This will store the 3x3 rotation matrix that is equivalent to the quaternion rotation in the matrix RotMat. This will allow us to find the vertex of a given object in terms of world coordinate space, which is necessary for collision detection.

Also notice that a negative index is being assigned to rigid_b to indicate that it is an immobile object. We can recover the index of the immobile object by adding one to rigid_b and then taking the negative.

The function PointDist() is assumed to return the distance the point sent is from the given immobile object's plane. This would be written something like:

```
double CImmobile::PointDist(CVector3 point)
{
   double ret_val;
   if (sqrtABC != 0)
   {
      ret_val = normal.x * point.x
                + normal.y * point.y
                + normal.z * point.z
                + D;

      ret_val /= sqrtABC;
   }
   else
      ret_val = 999999999999999;

   return ret_val;
}
```

PointVelocity() is assumed to return the velocity of the point on the object indexed:

```
Cvector3 CrigidSys::PointVelocity(long body_idx, CVector3 &point,)
{
   Cvector3 result;
   result = body[body_idx].vel;
   result += (body[body_idx].omega ^ (point - body[body_idx].pos));

   return result;
}
```

where v1 ^ v2 means v1 crossed with v2.

# 9 Linear Impulse Forces

Having now seen how the contact points and information about them could be calculated and stored along with a collision state, we must now find a way to use that information. To begin this we must realize there are two different times that an impulse force will need to be applied: when two objects collide and when one object is resting on another. Which kind of contact is prevalent will be stored in the collision_state variable. Recall this variable has four states: penetrating, colliding, resting and clear. The penetrating state was dealt with in the Update() function by back stepping in time. The clear state requires no special processing. So only the colliding and resting states remain to be processed. Notice that in a state of penetrating, collisions and resting may also be occurring. Likewise in a state of colliding, resting may also be happening. In a clear state none of the other three may be occurring.

We also should be aware that there will need to be two types of impulse forces applied: linear and angular. For this section we will focus on the linear impulse forces and leave angular impulse forces for the next section.

Linear impulse forces are probably the easiest to understand. In practice it is useful to think of them as a force that is acting on all points of the rigid body, but will be applied to the center of mass of the rigid body (this corresponds to the concept and usage of linear momentum). In this simple sense linear impulse forces on rigid bodies behave the same as such forces would on point-mass particles.

In the case of a rest state we know the scenario is such that the object has no velocity in the direction of the object it is in contact with – effectively it is resting on the second object. For example in the case of a rubber ball sitting on the floor, we expect the ball to stay sitting on the floor. It should not sink into the floor or bounce off the floor, even though gravity is pulling down on it. The reason it does not move is because the floor exerts an upward force exactly equal to the force of gravity. It is this upward force that we must model using a linear impulse force.

In the case of a collision state we can consider the example of a rubber ball hitting the floor. In this case we would expect it to bounce back off the floor. To achieve this effect in our model, if such a collision is detected we will need to apply a linear impulse force to the ball to “reverse” the direction of the ball. Likewise if a domino falls to the floor we would expect it to bounce off the floor. Again we would need to apply a linear impulse force to the domino to “reverse” its direction. The details of this will be given shortly, but first we must address the difference between these two scenarios by presenting the concept of an angular impulse force.

# 10 Angular Impulse Forces

In the above two examples the bounce off the floor can be achieved by a linear impulse force. However, that would only be sufficient in the case of the rubber ball. To illustrate this consider what happens when you drop the ball straight down on the floor – it bounces straight back up. There are no other changes to the ball (neglecting friction). If you drop the domino perfectly "flat" with the floor, so all its corners hit the floor at exactly the same time, it too will bounce straight back up. However if you orient the domino so that only one corner hits the floor first, it will not bounce straight back up with the same orientation. In fact it will bounce all around from one corner to another, to another, though it will still be bouncing off the floor on each contact.

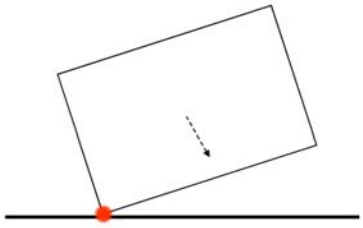
To model the effect of the domino we will need to calculate a linear impulse to apply to the rigid body (the domino) based on the point of contact. This linear impulse will effectively cause the domino to bounce up a little. However we must also calculate an angular impulse to cause the rotation that is induced by the collision. So for each contact point we will need to calculate a linear impulse component and an angular impulse force component. We will then use these impulse forces to alter the body's linear and angular momentum respectively.

This all seems straightforward. However, there are some other scenarios to consider. Specifically after the bounce off the first corner, the domino will be spinning. This means when the next corner hits the floor, not only is there a linear momentum driving it into the floor, but there may be angular momentum as well. We will need to be certain both are "reversed" properly, which is not that difficult.

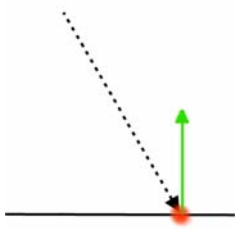
The tricky part comes when we want to allow a spinning body to stay in resting contact. For example, say we have determined the points of contact of one body, A, on another body B are a resting contact, however body A is rotating. Obviously we want to make sure body A does not rotate into body B, but we also want to allow body A to rotate (spin) on top of body B. The details of this are covered in later sections.

# 11 Calculating Colliding Impulse Forces

Calculating impulse forces is simple. Calculating the correct impulse forces is a little trickier. We will begin with the simple case of a linear impulse force for a vertex face collision, as illustrated below:



So the box is falling not straight down, but down and to the right as the dashed arrow indicates. We will assume the box is NOT rotating and one corner of the box has just hit the floor. The force diagram would look something like this:



Where the vertical line is the surface normal of the floor and the dashed line is the linear velocity of the box. We effectively want to reverse the component of the dashed line that is parallel to the surface normal. To accomplish this we compute the relative velocity between the corner of the box and the floor. Since the floor is not moving this is the velocity of the box. The impulse force required to achieve this reversal of direction is derived in [Bar1999] to be such that:

*normal* = the unit surface normal
*coeff_rest* = coefficient of restitution of the box
*I_inverse_box* = the inverse of the box's inertia matrix
*numerator* = – (1 + co*eff_rest*) * relative velocity
*ra* = (corner point of contact) – (box's center of mass)
*term1* = 1 / mass of the box
*term3* = normal * ( (*I_inverse_box*) * (*ra* ^ *normal*)) ^ *ra*)
*j* = *numerator* / (*term1* + *term3*)
*force* = *j* * *normal*

where '*' means dot product and '^' means cross product.

To use this we would add *force* to the box's linear momentum and the box's angular momentum would change from zero to *ra* ^ *force*. Notice that the box's coefficient of restitution will be zero if the box does not bounce at all and will be one if the box is extremely bouncy.

If we change the scenario slightly and assume the box is rotating, very little changes in the procedure. The relative velocity would be found to be:

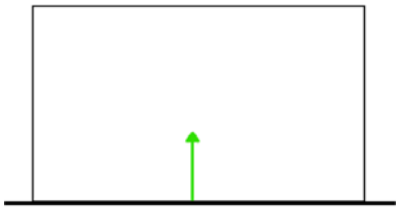
*omega* = angular velocity of the box
*padot* = (box's linear velocity) + (*omega* ^ *ra*)
*rel_vel* = *normal* * *padot*

The other calculations remain the same. To use this impulse force we again would add *force* to the box's linear momentum and add (*ra* ^ *force*) to the box's angular momentum. The coding details of this can be found later in this document in the Impulse Code section in the ComputeImpulse() function.

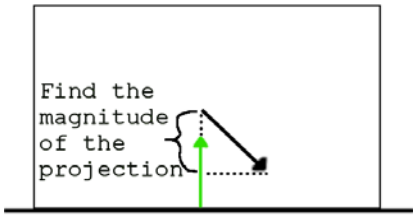
# 12 Calculating Rest Impulse Forces

Having taken care of the impulse forces needed for colliding points we are left only to find the impulse forces needed for resting points. In this discussion it is important to recognize the simplicity of the system we are considering. In this simple system there are likely cases that are not going to be addressed, however the methods presented here will still be valid, but they may need some functionality added in to deal with more complex scenarios.

We begin with the simple scenario of a block sitting in a resting state on the floor:



We will assume there is no rotation in any direction. This means we only need to prevent any linear momentum into the floor. To do this we need to calculate the magnitude of the momentum (force) into the floor and subtract it away. Or rather apply an impulse force in the direction of the surface normal which is equivalent to the force acting on the block in the downward direction of the surface normal. To find this impulse force consider the below force diagram, where a force is acting on the block pushing it down and to the right:




We want to find the projection of the current linear momentum onto the normal and then subtract that projection from the current linear momentum. To do that we set things up as follows:

*normal* = the surface normal of the floor
*magnitude* = the magnitude of the normal
*P* = the block's total linear momentum (equiv. to the force acting on the block)
*force* = ( (*P* * *normal*) / *magnitude* ) * *normal*;

To apply this impulse force we subtract *force* from the block's linear momentum. So we are done with the easy case.

If we allow the block to rotate we must think about what should happen. Consider the below two scenarios and assume that both blocks are found to be in resting states and both blocks have some angular momentum (this can and does happen):

In the first case, with two contact points, clearly the only rotation allowed would be around the surface normal. However in the second case, where there is only one contact point, we would expect the box to have a rotation such that the lower right corner would come down and touch the surface.

This leaves us with a problem to figure out. It appears that sometimes we want to remove all rotation except that around the surface normal and sometimes we do not want to remove this rotation. We must now find a way to determine when we cancel the rotation and when we do not. To make things easy we will assume in both cases it is okay to apply the linear impulse, as described above, to keep the contact points in a rest state.

To determine if we must cancel any angular momentum we will pretend that we do not need to cancel. We will then advance a time step and see what happens. If things end up in a penetrating state then we know we must cancel all the angular momentum except around the surface normal. If things end up in a colliding or rest state (or clear state) then we do not need to cancel any of the angular momentum. This forward time step is done for each contact point in a rest state.

Now that we have determined when we need to cancel some of the angular momentum we are left with the task of how. Fortunately this is as easy to do as the linear momentum was. Consider that *L* is the angular momentum of the block. Then *L* is a vector of three values which represents the line about which the block is rotated. The magnitude and direction of *L* indicates the degree of rotation around this line and the direction (clockwise or counterclockwise) of this rotation.

Thus all we need to do is project the current angular momentum onto the surface normal and set the angular momentum to equal this project. This will cancel all rotation except that around the surface normal, maintaining the correct direction and magnitude. Specifically we say:

$$L = ( (L * normal) / magnitude ) * normal;$$

# 13 Impulse Code

Finally we come to the last functions necessary to complete our simulation. These would be the member functions: CalcImpulse() and CalcRestImpulse(). The first is called only from our Update() function when we are in a colliding state and the second is only called from our dydt() function when we are in a colliding or rest state. For an explanation of how these functions operate see the above sections on impulse forces.

```
void CrigidSys::ComputeImpulse()
{
   Cvector3 padot;        // point on a  velocity
   Cvector3 ra;           // line from a to center of mass
   Cvector3 normal;       // normal of collision plane
   Cvector3 force;
   Cmatrix3x3 omega_star;
   Cquaternion qdot;
   long a, b, i;
   double term1, term2, term3, term4;
   double rel_vel, numerator, j;
   bool need_recompute;

   // For a colliding state we know we have a point such that one
   // object is within -DEPTH_EPSILON to DEPTH_EPSILON of another
   // If the objects are NOT moving toward one another this is a
   // REST state, however if they are moving toward each other then
   // we count it as a collision.

   need_recompute = false; // only if we have a real collision

   for (i = 0; i < num_contacts; i++)
   {
      a = contacts[i].rigid_a;
      b = contacts[i].rigid_b;
      normal = contacts[i].normal;

      if (b < 0)  // b is immobile object
      {
         PointVelocity(a, contacts[i].point, padot);
         pbdot.x = 0;  // immobile objects can't have velocity
         pbdot.y = 0;
         pbdot.z = 0;
         ra = contacts[i].point - body[a].pos;

         rel_vel = normal * padot;    // pbdot is known zero

         if (rel_vel < -VEL_THRESH) // colliding
         {
            numerator = -(1 + coeff_rest) * rel_vel;

            term1 = 1.0 / body[a].mass;
            term2 = 0;  // immobile objects have infinite mass
            term3 = normal * ( (body[a].I_inv * (ra ^ normal)) ^ ra);
            term4 = 0;
```

```
      // j = magnitude of impulse force
      j = numerator / (term1 + term3
      force = j * normal;

      // Apply impulse to the bodies (in this case just to A)
      body[a].P += force;
      body[a].L += (ra ^ force);

      // Recompute auxiliary variables
      //  -- AFTER all collisions are done
      need_recompute = true;

   } // end if colliding

   else if (rel_vel < VEL_THRESH) // resting
   {
      // zero out the force acting in the normal direction
      // of the surface find the projection of the linear
      // force (momentum) on the surface normal
      force = ((body[a].P * normal) / (normal.Mag()) ) * normal;
      body[a].P -= force;

      need_recompute = true;

      // Alter L as needed - see ComputeRestImpulse for reasoning
      CVector3 tmp_omega;
      tmp_omega.x = body[a].I_inv[0][0] * body[a].L.x +
                    body[a].I_inv[0][1] * body[a].L.y +
                    body[a].I_inv[0][2] * body[a].L.z;

      tmp_omega.y = body[a].I_inv[1][0] * body[a].L.x +
                    body[a].I_inv[1][1] * body[a].L.y +
                    body[a].I_inv[1][2] * body[a].L.z;

      tmp_omega.z = body[a].I_inv[2][0] * body[a].L.x +
                    body[a].I_inv[2][1] * body[a].L.y +
                    body[a].I_inv[2][2] * body[a].L.z;

      padot = (tmp_omega ^ ra);
      rel_vel = normal * padot;

      if (rel_vel < -VEL_THRESH)
      {
         // this means current rotation WOULD cause a collision,
         // so it must be altered, else it is doing nothing of
         // concern to this resting point so leave it
         force = ((body[a].L * normal) / normal.Mag() ) * normal;
         body[a].L = force;
      }
   } // end resting

} // end if b was immobile

else   // B is not immobile
{
   // at the moment can't happen
}
```

```
   } // end for i

   if (need_recompute)
   {
      // Recompute auxiliary variables
      for (i=0; i < num_bodies; i++)
      {
         body[i].vel   = body[i].P / body[i].md_Mass;
         body[i].omega.x = body[i].I_inv[0][0] * body[i].L.x +
                           body[i].I_inv[0][1] * body[i].L.y +
                           body[i].I_inv[0][2] * body[i].L.z;

         body[i].omega.y = body[i].I_inv[1][0] * body[i].L.x +
                           body[i].I_inv[1][1] * body[i].L.y +
                           body[i].I_inv[1][2] * body[i].L.z;

         body[i].omega.z = body[i].I_inv[2][0] * body[i].L.x +
                           body[i].I_inv[2][1] * body[i].L.y +
                           body[i].I_inv[2][2] * body[i].L.z;
      } // end recomputing
   } // end if need_recompute

} // end ComputeImpulse




void CrigidSys::ComputeRestImpulse(long cur_body_idx)
{
   Cvector3 padot;
   Cvector3 ra;
   Cvector3 normal;        // normal of collision plane
   Cvector3 force;
   Cmatrix3x3 omega_star;
   Cquaternion qdot;
   long a, b, i;
   double rel_vel;
   bool need_recompute;

   need_recompute = false; // only if we have a real collision

   for (i = 0; i < num_contacts; i++)
   {
      a = contacts[i].rigid_a;
      b = contacts[i].rigid_b;

      if ((a == cur_body_idx) || (b == cur_body_idx) )
      {
         normal = contacts[i].normal;

         if (b < 0)  // immobile object
         {
            PointVelocity(a, contacts[i].m_p, padot);
            pbdot.x = 0;  // immobile objects can't have velocity
            pbdot.y = 0;
```

```
      pbdot.z = 0;
      ra = contacts[i].m_p - body[a].pos;

      rel_vel = normal * padot;    // pbdot is known zero

      if (rel_vel < VEL_THRESH) // resting
      {
         // zero out the force acting in the normal
         // direction of the surface find the projection of
         // the linear force (momentum) on the surface normal
         force = ((body[a].P * normal) / normal.Mag() ) * normal;
         body[a].P -= force;

         need_recompute = true;

         // Cancel all rotate EXCEPT that which goes about
         // the surface normal - i.e. get the rotation about
         // the normal and set this body's rotation to JUST that.
         // But first find out IF the rotation will cause this
         // point to be forced into the surface
         // Notice the above canceled all the force for linear
         // momentum (which means gravity and such gets suspended
         // for the ENTIRE object - which is why getting this
         // angular stuff correct matters)
         CVector3 tmp_omega;
         tmp_omega.x = body[a].I_inv[0][0] * body[a].L.x +
                       body[a].I_inv[0][1] * body[a].L.y +
                       body[a].I_inv[0][2] * body[a].L.z;

         tmp_omega.y = body[a].I_inv[1][0] * body[a].L.x +
                       body[a].I_inv[1][1] * body[a].L.y +
                       body[a].I_inv[1][2] * body[a].L.z;

         tmp_omega.z = body[a].I_inv[2][0] * body[a].L.x +
                       body[a].I_inv[2][1] * body[a].L.y +
                       body[a].I_inv[2][2] * body[a].L.z;

         padot = (tmp_omega ^ ra);
         rel_vel = normal * padot;

         if (rel_vel < -VEL_THRESH)
         {
            // this means current rotation WOULD cause a
            // collision,so it must be altered, else it is doing
            // nothing of concern to this resting point so
            // leave it
            force = ((body[a].L * normal) /normal.Mag()) *normal;
            body[a].L = force;
         }

      } // end if resting

   } // end if b was immobile

   else   // B is not immobile
   {
      // at the moment can't happen
```

```
            }
         } // end if a or b = cur_body_idx
      } // end for i

      if (need_recompute)
      {
         // Recompute auxiliary variables
         for (i=0; i < ml_NumActive; i++)
         {
            body[i].md_Vel   = body[i].P / body[i].md_Mass;
            body[i].md_Omega.x = body[i].I_inv[0][0] * body[i].L.x +
                                 body[i].I_inv[0][1] * body[i].L.y +
                                 body[i].I_inv[0][2] * body[i].L.z;

            body[i].md_Omega.y = body[i].I_inv[1][0] * body[i].L.x +
                                 body[i].I_inv[1][1] * body[i].L.y +
                                 body[i].I_inv[1][2] * body[i].L.z;

            body[i].md_Omega.z = body[i].I_inv[2][0] * body[i].L.x +
                                 body[i].I_inv[2][1] * body[i].L.y +
                                 body[i].I_inv[2][2] * body[i].L.z;
         } // end recomputing
      } // end if need_recompute

} // end ComputeRestImpulse
```

# 14 Conclusion

So we come to the end of all the pseudocode and to the end of this paper. Hopefully it has helped in the first steps of creating a program to model rigid body motion. From here you should be able to continue to add in new features and allow for more complex interactions between the objects. Of a small note, you may wish to incorporate gravity and other force fields into the simulation. To do this, in your ComputeForceAndTorque() function you would likely want something along the lines of:

```
   if (GravityOn)
   {
      rb->force.z -= GravityForce * rb->md_Mass;
   }
```

There are many topics that could be discussed more and there are many features that could be added. From this point it should be a relatively straightforward task to allow objects to collide with objects – at least for the vertex to face collisions. It is strongly encouraged that if you are interested in expanding upon the code presented here and in [Bar1999] you look for more sources of information. An obvious direction to go would be to learn more about collision detection. You might also want to investigate how to add in friction or wind resistance. These are just a couple possibilities. Computer graphics is an enormous field of study, the possibilities are almost boundless.

# Bibliography


[Bar1999] Baraff, David and Andrew Witkin *Physically Based Modeling Course Notes,* SIGGRAPH 1999, Los Angeles, CA, USA.

[Bou2002] Bourg, David M. Physics for Game Developers, O'Reilly and Associates, 2002.

[Car1992] J. H. E. Cartwright & O. Piro, *"The dynamics of Runge-Kutta methods,"* International Journal on Bifurcation and Chaos 2, 427-449, 1992.

[Lin1998] Lin M. and S. Gottschalk, *"Collision Detection between Geometric Models: A Survey,"* Proceedings of IMA Conference on Mathematics of Surfaces , 1998.

[Mir1995] Mirtich, Brian and John Canny, *"Impulse-based Simulation of Rigid Bodies,"* Proceedings of ACM Interactive 3D Graphics Conference, 1995.

[Pre1987] Press, Teukolsky, Vettering, and Flannery, Numerical Recipes in C, Cambridge University Press, 1987.