Workshop Series: Impulse Response

Benjamin Kenwright[1]*

Abstract

This practical focuses on impulse collision responses. Impulses are a computationally fast and simple approach for creating rigid contacts. The practical introduces the principles behind impulse and an uncomplicated implementation.

Keywords

Newton's Laws, Momentum, Contacts, Penetration Resolution, Collision Response, Impulses, Rigid-Body, Real-Time, Video Games, C++
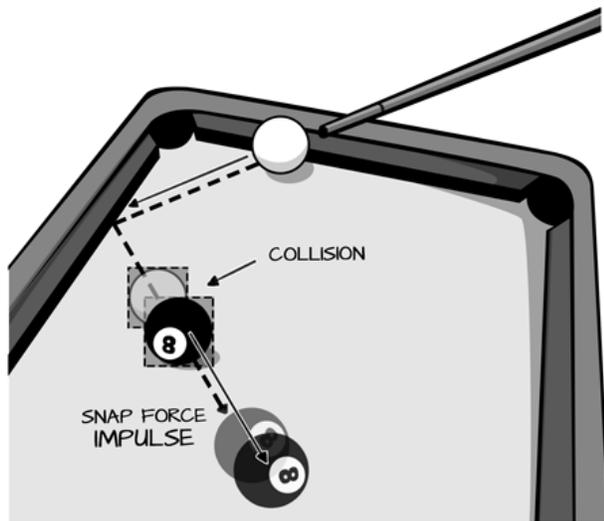
Figure 1. Impulses - Constraints (e..g, collisions) enforced using impulse force based upon changes in momentum, while the mass remains constant.

Contents

Introduction

Physics   A quick and approximate rigid body physics engine is challenging and valuable. The fundamental principles, i.e., Newton's Laws and numerical integration, enable us to move objects around the environment in a believable manner, while collision detection allows us to determine if two objects have intersected. In this practical, we discuss how to proceed when we have identified that two objects have collided in order to move them apart using impulses. As you might expect, the solution is to give each intersecting object a nudge in the direction away from the collision.

Recap   In the previous practicals, we have discussed how to identify when an intersection has occurred, and how to calculate the data required to resolve an intersection. The data which we have calculated consists of:

- The contact point - i.e., where the intersection was detected - this is typically inside one or more of the objects.
- The contact normal - i.e. the direction vector along which the intersecting object must move to resolve the collision.
- The penetration depth - i.e. the distance along the contact normal that the intersecting object must move so that it is no longer intersecting.

1. Overview

Principles and Concepts   Remember that, at this point, a rigid body physics engine is still dealing with all the simulated objects - the new physical state of the simulated objects is not drawn to the screen until after all the physics update is complete, so the player will not see any of the intersections between objects, if they are successfully resolved.

Question The question which is addressed in this practical is: how do we use this collision data to move the intersecting objects apart?

A simple solution would be to simply move the objects along the collision normal by a distance equal to the penetration depth, by directly changing the position vectors. This is known as the **projection method**, and while it will suffice for a simple simulation, it has some fairly obvious drawbacks related to the objects' velocities. If the objects are moved apart without changing their velocities then they will just continue along the same path during the next physics update and are likely to intersect again; alternatively, if the objects are moved apart and the velocity are set to zero then the simulation feels very unrealistic, as objects tend to bounce off one another rather than stop dead on first contact. We clearly need a solution which affects the velocities and/or accelerations of the objects rather than directly altering the positions.

Algorithms which directly alter the velocities of the intersecting objects are known as **Impulse Methods**, whereas algorithms which directly a inject the acceleration of the bodies are known as **Penalty Methods**. Penalty methods use spring forces to, in effect, pull the objects away from each other by affecting the acceleration through Newton's second law ($F = ma$). Impulse methods use instantaneous nudges, or impulses, to push the objects apart by directly controlling their velocities. In this practical, we will concentrate on impulse methods.

Summary
- Projection methods - control the position of the intersecting objects directly.
- Impulse methods - control the velocity of the objects, i.e. the first derivative.
- Penalty methods - control the acceleration of the objects, i.e. the second derivative.

2. Impulses

The impulse method allows us to directly affect the velocities of the simulated objects which have intersected. This is achieved through the application of an impulse, which can be thought of as an immediate transfer of momentum between the two bodies. Impulse is a term defined by classical physics as the accumulated force applied to a body over a specific amount of time (it is therefore measured in Newton seconds Ns). The impulse J is defined in terms of force F and time period Δt as:

$$J = F\Delta t \qquad (1)$$

We know from Newton's second law, that $F = ma$, and we can also write the acceleration a as the rate of change of velocity v. Substituting these values into the equation for impulse gives us:

$$
\begin{aligned}
J &= F\Delta t \\
&= ma\Delta t \\
&= m\frac{\Delta v}{\Delta t}\Delta t \\
&= m\Delta v
\end{aligned} \qquad (2)
$$

where m is mass, v is velocity, F is force, a is acceleration, Δt is the time-step, and J is our impulse. We also know that momentum is equal to the product of mass and velocity, so an impulse is equivalent to the change in momentum. Our plan then is to give colliding objects a nudge, by changing their velocity by an amount equal to:

$$\Delta v = \frac{J}{m} \qquad (3)$$

The question then, is how to calculate the impulse J generated when two bodies collide.

3. Calculating a Linear Impulse

First, we will discuss what we would like to happen to the velocities of two colliding objects - namely, we want the bodies to bounce of one another. We will consider the simple case of two spheres colliding, as shown in the Figure 2 below - sphere a is moving with velocity v_a, while sphere b has velocity v_b; the collision normal is $\hat{n}$. We want to calculate the impulse J.
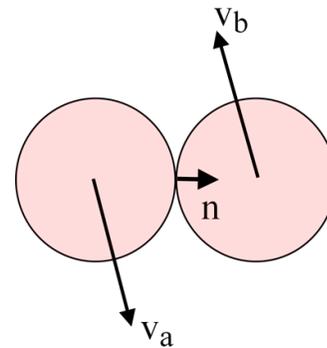
Figure 2. Linear Impulse - Single impulse between two rigid body objects.

The impulse is generated by the velocity at which the two spheres have collided so we are interested in the relative velocity of the two objects, which we will label v_{ab}. The component of the relative velocity which caused the collision is along the normal vector, so we calculate the dot product (identified as ·) of the relative velocity and the normal:

$$v_{ab} = v_a - v_b \quad v_n = v_{ab} \cdot \hat{n} \qquad (4)$$

The velocity along the normal after the collision depends on the coefficient of elasticity ε. A coefficient of 1.0 means the collision will be purely elastic, so all the velocity is transferred, whereas a coefficient of 0.0 is purely non-elastic, so

no velocity is transferred. A purely non-elastic collision will result in the two bodies staying together (i.e. no bounce); a purely elastic collision is a perfect bounce so no damping or slowing down occurs. Your simulation is likely to require a number somewhere in between (e.g., 0.7). Quite often different object types in a game will have different coefficients of elasticity which are stored as a member of the object class, in the same way as the mass is.

The coefficient of elasticity is the factor by which the velocity before the collision is multiplied to calculate the velocity after the collision. Hence:

$$v_n^+ = -\varepsilon v_n^- \tag{5}$$

where v_n is the relative velocity along the normal, and the $+$ and $-$ indicate pre and post values (i.e., before and after the collision). If we substituting for v_n:

$$(v_a^+ - v_b^+) \cdot n = -\varepsilon (v_a^- - v_b^-) \cdot \hat{n} \tag{6}$$

Remember, the $+$ and $-$ denote the state of the bodies after and before the collision respectively. Furthermore, the negation of the velocity is because we want to push the two bodies back apart in the opposite direction to their colliding velocity.

We also need to think about the momentum of the two bodies. You will remember from the first tutorial in this series on Newtonian mechanics that the total momentum must remain constant in any collision. However our plan to resolve the collision is to "inject" some momentum into the system. Hence we need to ensure that the overall additional momentum is equal to zero, which is achieved by making the momentum used to nudge the second body, the exact opposite of that used to nudge the first. This is shown in the equations below, showing the relationship between momentum before and after the collision for each body, where J is the injected impulse along the normal vector n.

$$m_a v_a^+ = m_a v_a^- + J\hat{n}$$
$$m_b v_b^+ = m_b v_b^- + J\hat{n} \tag{7}$$

The injected momentum which is equal and opposite, not the velocity as that will be affected by the mass of the object and therefore unequal for differently sized objects. Combining the last three equations, allows us to solve for the impulse J:

$$J = \frac{-(1-\varepsilon)v_{ab} \cdot n}{n \cdot n(\frac{1}{m_a} + \frac{1}{m_b})} \tag{8}$$

where J is a scalar impulse, ε is the coefficient of restitution (between 0 and 1), v_{ab} is the relative velocity, $\hat{n}$ is the contact normal, m_a and m_b are the mass of each body. The impulse J in turn allows us to calculate the velocities of the two bodies after the collision:

$$v_a^+ = v_a^- + \frac{J}{m_a}\hat{n}$$
$$v_b^+ = v_b^- - \frac{J}{m_b}\hat{n} \tag{9}$$

So we can now calculate the velocity at which two colliding bodies should move away from one another in a manner which feels believable, as it is based on Newtonian mechanics, and which incorporates a "bounciness" factor for different types of object in the form of the coefficient of elasticity. Remember, as ever, that the velocities we are discussing are three dimensional vectors, i.e., the bodies can move along all three axes of the simulated world. They can also rotate around all three axes, so next we need to consider how to account for rotational elements in collision response.

4. Calculating an Angular Impulse

Let's add some spin to our two colliding spheres. The angular velocity of each is ω_a, ω_b and the radius is r_a, r_b respectively, as shown below in Figure 3:
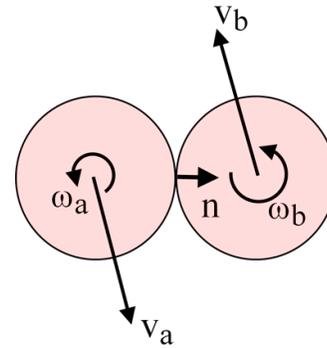
Figure 3. Angular Impulse - Single angular impulse between two rigid body objects.

In order to realistically add angular motion to a collision response, we need some more information about the actual contact point between the two objects - specifically the velocity of that point on each of the objects. This is the sum of the object's linear velocity, and the extra tangential velocity v_t created by the fact that the point is rotating around the object's centre. The velocity at a point which is distance r from the centre on an object turning with angular velocity ω is:

$$v_t = \omega r \tag{10}$$

Recall that rotational velocity is measured in radians, and there are 2π radians in a full revolution; similarly, the distance around that revolution is $2\pi r$. So the velocity of the contact point v_c on each object is:

$$v_{ca} = v_a - \omega_a r_a$$
$$v_{cb} = v_b - \omega_b r_b \tag{11}$$

Again we need to ensure that angular momentum is conserved, so:

$$I_a \omega_a^+ = I_a \omega_a^- + r_a \times J\hat{n}$$
$$I_b \omega_b^+ = I_b \omega_b^- + r_b \times J\hat{n} \tag{12}$$

The angular momentum of a body is the product of the angular velocity ω and the inertia tensor I. J is the impulse which we are calculating and n is the collision normal. Solving our equations for J, taking into account the angular momentum, leads to a much more complicated looking calculation for the impulse:

$$J = \frac{-(1-\varepsilon)v_{ab}\cdot\hat{n}}{\hat{n}\cdot\hat{n}(\frac{1}{m_a}+\frac{1}{m_b})+[(I_a^-1(r_a\times\hat{n}))\times r_a+(I_b^-1(r_b\times\hat{n}))\times r_b]\cdot\hat{n}} \tag{13}$$

This allows us to calculate the velocities of the two bodies after the collision, as well as the angular velocities:

$$\begin{aligned}
v_a^+ &= v_a^- + \frac{J}{m_a}\hat{n} \\
v_b^+ &= v_b^- - \frac{J}{m_b}\hat{n} \\
\omega_a^+ &= \omega_a^- + \frac{r_a\times J\hat{n}}{I_a} \\
\omega_b^+ &= \omega_b^- + \frac{r_b\times J\hat{n}}{I_b}
\end{aligned} \tag{14}$$

These equations allow us to write an algorithm in C++ to believably simulate two objects colliding and bouncing off one another using Newtonian mechanics. **Both linear and angular movement** are accounted for, and the elasticity of the collision is also incorporated.

5. More Complex Shapes

The simple case of two colliding spheres has been used to illustrate the algorithms employed in impulse method collision response. The algorithms are perfectly suited to more complex three-dimensional shapes. In fact, you should notice that there are no assumptions about the shapes made in the calculations. When implementing the code, you will see that all calculations are carried out in three dimensions, so for example the distance of a contact point from the centre of an object is a vector containing three values - it does not matter whether that point is on the surface of a sphere or a more complex shape. Similarly the relationship between angular velocity and linear velocity is irrespective of the object shape, it is based purely on the distance of the point of interest from the centre of rotation.

6. Implementation

The aim of this practical session is to expand your physics engine to react to collisions between simulated objects. We will implement the impulse method for both linear and angular motion. To demonstrate that the collision tests are working, we will add functionality to the project which bounces colliding objects off one another.

Listing 1. Rigid Body Impulse (Angular and Linear)

```cpp
// First ..
// Recap of common Rigid Body information...
class RigidBody
{
public:
  // ***** LINEAR ******
  Vector3   m_centre;   // Center Of Rigid Body
  float   m_invMass;
  Vector3   m_linVelocity;
  Vector3   m_forces;

  // ***** ANGULAR *****
  Matrix4   m_invInertia;
  Vector3   m_angVelocity;
  Quaternion  m_orientation;
  Vector3   m_torques;

  // ***** COMBINED *****
  Matrix4   m_matWorld;
  Matrix4   m_worldInvInertia;
  //...
  //  Common functions, such as AddForce(), Integrate(), ←
      GetPosition()..
};


// Second..
// Actual collision impulse implementation method
//←
    _____

//
//                    -(1+e)(relv.norm)
// j =  ←
    _____

//     norm.norm(1/Mass0 + 1/Mass1) + (sqr(r0 x norm) / ←
        Inertia0) + (sqr(r1 x norm) / Inertia1)
//
//←
    _____

static
void AddCollisionImpulse( RigidBody&      c0,
        RigidBody&      c1,
        const Vector3&   hitPoint,
        const Vector3&   normal,
        float      penetration)
{
  // Some simple check code.
  float invMass0 = c0.m_invMass;
  float invMass1 = c1.m_invMass;

  const Matrix4& worldInvInertia0 = c0.m_worldInvInertia;
  const Matrix4& worldInvInertia1 = c1.m_worldInvInertia;

  // Both objects are non movable
  if ( (invMass0+invMass1)==0.0 ) return;

  Vector3 r0 = hitPoint - c0.m_centre;
  Vector3 r1 = hitPoint - c1.m_centre;

  Vector3 v0 = c0.m_linVelocity + Cross(c0.m_angVelocity, ←
      r0);
  Vector3 v1 = c1.m_linVelocity + Cross(c1.m_angVelocity, ←
      r1);

  // Relative Velocity
  Vector3 dv = v0 - v1;

  // If the objects are moving away from each other we dont ←
      need to apply an impulse
  float relativeMovement = -Dot(dv, normal);
  if (relativeMovement < -0.01f)
  {
    return;
```

```
66    }
67
68    // NORMAL Impulse
69    {
70      // Coefficient of Restitution
71      float e = 0.0f;
72
73      float normDiv = Dot(normal, normal) *
74            ( (invMass0 + invMass1) +
75        Dot( normal,
76        Cross( Transform( Cross(r0, normal), ←
          worldInvInertia0), r0) +
77        Cross( Transform( Cross(r1, normal), ←
          worldInvInertia1), r1) ) );
78
79      float jn = −1*(1+e)*Dot(dv, normal) / normDiv;
80
81      // Hack fix to stop sinking − bias impulse proportional to ←
          penetration distance
82      jn = jn + (penetration*1.5f);
83
84      c0.m_linVelocity   += invMass0 * normal * jn;
85      c0.m_angVelocity   += Transform(Cross(r0, normal * jn), ←
          worldInvInertia0);
86
87      c1.m_linVelocity   −= invMass1 * normal * jn;
88      c1.m_angVelocity   −= Transform(Cross(r1, normal * jn),←
          worldInvInertia1);
89    }
90
91    DBG_CHECKVECTOR( c0.m_linVelocity )
92    DBG_CHECKVECTOR( c0.m_angVelocity )
93    DBG_CHECKVECTOR( c1.m_linVelocity )
94    DBG_CHECKVECTOR( c1.m_angVelocity )
95
96    // TANGENT Impulse Code
97    #if 1
98    {
99      // Work out our tangent vector, with is perpendicular
100     // to our collision normal
101     Vector3 tangent(0,0,0);
102     tangent = dv − (Dot(dv, normal) * normal);
103     tangent = Normalize(tangent);
104
105     float tangDiv = invMass0 + invMass1+
106
107           Dot( tangent,
108             Cross((Cross(r0, tangent) * worldInvInertia0), r0←
        ) +
109             Cross((Cross(r1, tangent) * worldInvInertia1), r1←
        )
110           );
111
112     float jt = −1 * Dot(dv, tangent) / tangDiv;
113     // Clamp min/max tangential component
114
115     // Apply contact impulse
116     c0.m_linVelocity += invMass0 * tangent * jt;
117     c0.m_angVelocity += Transform(Cross(r0, tangent * jt), ←
          worldInvInertia0);
118
119     c1.m_linVelocity −= invMass1 * tangent * jt;
120     c1.m_angVelocity −= Transform(Cross(r1, tangent * jt), ←
          worldInvInertia1);
121   }
122   #endif
123   // TANGENT
124
125   DBG_CHECKVECTOR( c0.m_linVelocity )
126   DBG_CHECKVECTOR( c0.m_angVelocity )
127   DBG_CHECKVECTOR( c1.m_linVelocity )
128   DBG_CHECKVECTOR( c1.m_angVelocity )
129 }// End AddCollisionImpulse(..)
```

7. Summary

In this practical, we have introduced the concept of collision response with particular focus on impulse methods, i.e. a method which directly affects the velocities of colliding objects in order to resolve that collision.

8. Exercises

This practical only gives a brief taste of collision impulse principles. As an exercise for the student to help enhance their understanding:

Intermediate
- Disable tangential impulses and see what happens
- Implement a large scene with a vast number of bodies interacting (e.g., >100)
- Create stacks of rigid body objects
- Do items come to a complete rest (e.g. jiggling?)

Advanced
- Try and construct impulse constraints to join bodies together

Acknowledgements

Recommended Reading

Code Complete: A Practical Handbook of Software Construction, Steve McConnell, ISBN: 978-0735619678

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

Game Inverse Kinematics: A Practical Introduction (2nd Edition) Kenwright. ISBN: 979-8670628204

Kinematics and Dynamics Paperback. Kenwright. ISBN: 978-1539595496

Game Collision Detection: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1511964104

Game C++ Programming: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1516838165

Computational Game Dynamics: Principles and Practice (Paperback). Kenwright. ISBN: 978-1501018398

Game Physics: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1471033971

Game Animation Techniques: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1523210688