Simulating the Rigid World

# Angular Components of Collision Responses

Now that you have a concrete understanding and have successfully implemented the Impulse Method for collision responses with linear velocities, it is time to integrate the support for the more general case of rotations. Before discussing the details, it is helpful to relate the correspondences of Newtonian linear mechanics to that of rotational mechanics. That is, linear displacement corresponds to rotation, velocity to angular velocity, force to torque, and mass to rotational inertia, or, angular mass. Rotational inertia determines the torque required for a desired angular acceleration about a rotational axis. The following discussion focuses on integrating rotation in the Impulse Method formulation and does not attempt to present a review on Newtonian Mechanics for Rotation. Conveniently, integrating proper rotation into the Impulse Method does not involve derivation of any new algorithm. All that is required is the formulation of impulse responses with proper consideration of rotational attributes.

## Collisions with Rotation Consideration

The key to integrating rotation into the Impulse Method formulation is recognizing the fact that the linear velocity you have been working with, e.g., velocity of object A, is actually the velocity of the shape at its center location. In the absence of rotation, this velocity is constant throughout the object and can be applied to any position. However, as illustrated in Figure 9-28, when the movement of an object includes angular velocity, , its linear velocity at a position , , is actually a function of the relative position between the point and the center of rotation of the shape, or the positional vector .

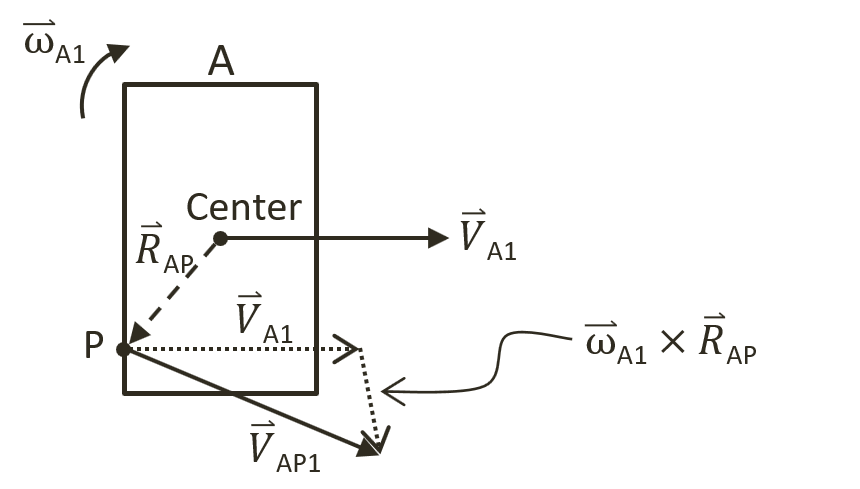


Figure 9-28 Linear Velocity at a Position in the Presence of Rotation

**Note**: Angular velocity is a vector that is perpendicular to the linear velocity. In this case, as linear velocity is defined on the X/Y plane, is a vector in the z direction. Recall from discussions in the Introduction Section of this chapter, the very first assumption made was that rigid shape objects are continuous geometries with uniformly distributed mass where the center of mass is located at the center of the geometric shape. This center of mass is the location of the axis of rotation. For simplicity, in your implementation, will be stored as a simple scalar representing the z-component magnitude of the vector.

Figure 9-29 illustrates an object B with linear and angular velocities of and colliding with object A at position . At this point, you know that the linear velocities at point before the collision for the two objects are as follows,

* ***(9)***
* ***(10)***

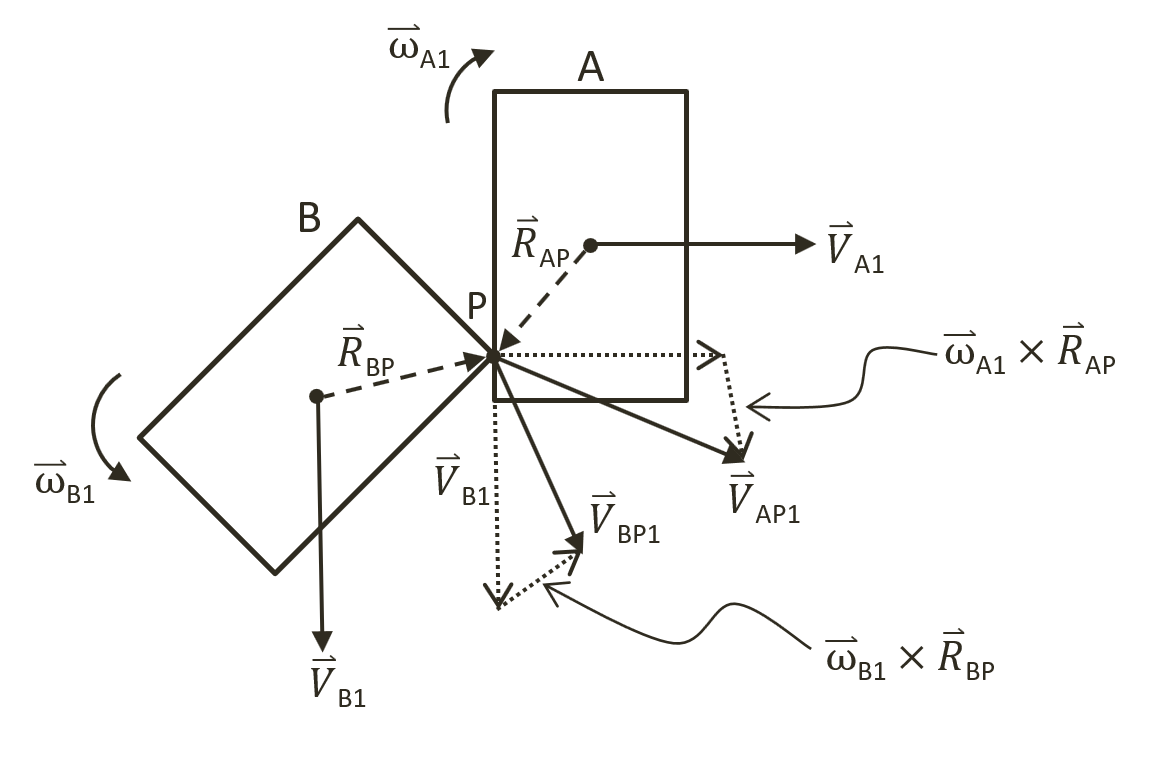


Figure 9-29 Colliding Shapes with Angular Velocities

After the collision, the linear velocity at the collision position can be expressed as follows,

* ***(11)***
* ***(12)***

Where, and , and and , are the linear and angular velocities for objects A and B after the collision, and, the derivation of a solution for these quantities is precisely the goal of this section.

## Relative Velocity with Rotation

Recall from the previous section, the definition of relative velocity from before and after a collision between objects A and B are defined as follows,



These velocities are analyzed based on components in the collision normal and tangent directions in Equations (1) and (2), and for reference convenience relisted in the following.

* ***(1)***
* ***(2)***

These equations are derived without considering rotation and the formulation assumes that the velocity is constant over the entire shape. In order to support rotation, these equations must be generalized and solved at the point of collision, .

* ***(13)***
* ***(14)***

In this case, and are relative velocities at collision position from before and after the collision. It is still true that these vectors are defined by the difference in velocities for objects A and B from before, and , and after, and , the collision at the collision position on each object,

* ***(15)***
* ***(16)***

You are now ready to generalize the Impulse Method to support rotation and to derive a solution to approximate the linear and angular velocities: , , , and .

## Impulse Method with Rotation

Continue with the Impulse Method discussion from the prevision section, that after the collision between objects A and B, the Impulse Method describes the changes in their linear velocities by an impulse, , scaled by the inverse of their corresponding masses, and . This change in linear velocities is descripted in Equations (3) and (4), relisted as follows,

* ***(3)***
* ***(4)***

In general, rotations are intrinsic results of collisions and the same impulse must properly describe the change in angular velocity from before and after a collision. Remember that inertial, or rotational inertial, is the rotational mass. In a manner similar to linear velocity and mass, it is also the case that the change in angular velocity in a collision is inversely related to the rotational inertia. As illustrated in Figure 9-29, for objects A and B with rotational inertia of and , after a collision the angular velocities, and , can be described as follows, where and are the positional vectors of each object,

* ***(17)***
* ***(18)***

Recall from the previous section that it is convenient to express the impulse as a linear combination of components in the collision normal and tangent directions, and , or,

Substituting this expression into Equation (17) results in the following,

In this way, Equations (17) and (18) can be expanded to describe the change in angular velocities caused by the normal and tangent components of the impulse, as follows.

* ***(19)***
* ***(20)***

The corresponding equations describing linear velocity changes, Equations (5) and (6), are relisted in the following,

* ***(5)***
* ***(6)***

You can now substitute Equations (5) and (19) into Equation (11), and, Equations (6) and (20) into Equation (12),

* ***(21)***
* ***(22)***

It is important to reiterate that the changes to both linear and angular velocities are described by the same impulse, . In other words, the normal and tangent impulse components and in Equations (21) and (21) are the same quantities and these two are the only unknowns in these equations where the rest of the terms are values either defined by the user, or, can be computed based on the geometric shapes. That is, the quantities , , , , , , and , are defined by the user, and,, , and can be computed. You are now ready to derive the solutions for and .

**Note** In the following derivation, it is important to remember the triple scalar product identity where given vectors, , , and, , the following is always true:

### Normal Components of the Impulse

The normal component of the impulse, , can be approximated by assuming that the contribution from the angular velocity tangent component is minimal and can be ignored, and isolating the normal components from Equations (21) and (22). For clarity, you will work with one equation at a time.

Now, ignore the tangent component of the angular velocity and perform a dot product with the vector on both sides of Equation (21) to isolate the normal components,

Carry out the dot products on the right-hand-side, recognizing is a unit vector and is perpendicular to , and, let , then, this equation can be re-written as,

* ***(23)***

The vector operations of the right-most term in Equation (23) can be simplified by applying the triple scalar product identity and remembering that, ,

* =

With this manipulation and collecting the terms with dot-product, Equation (23) becomes,

From Equation (9), the dot-product term is simply ,

* ***(24)***

Equation (22) can be processed through an identical algebraic manipulation steps, by ignoring the tangent component of the angular velocity and performing a dot product with the vector on both side of the equation, the following can be derived,

* ***(25)***

Subtract Equation (25) from (24) results in,

Substitute Equation (16) followed by (13) on the left-hand-side, and Equation (15) on the right-hand-side,

Lastly, collect terms and solve for ,

* ***(26)***

### Tangent Component of the Impulse

The tangent component of the impulse, , can be approximated by assuming that the contribution from the angular velocity normal component is minimal and can be ignored, and isolating the tangent components from Equations (21) and (22) by performing a dot product with the vector to both sides of the equations.

Now follow the exact algebraic manipulation steps as when working with the normal component the impulse in the tangent direction, , can be derived and expressed as followed.

* ***(27)***

## The Collision Angular Resolution Project

This project will guide you through the implementation of general collision impulse response that supports rotation. You can see an example of this project running in Figure 9-30. The source code to this project is defined in chapter9/9.8.collision\_angular\_resolution.

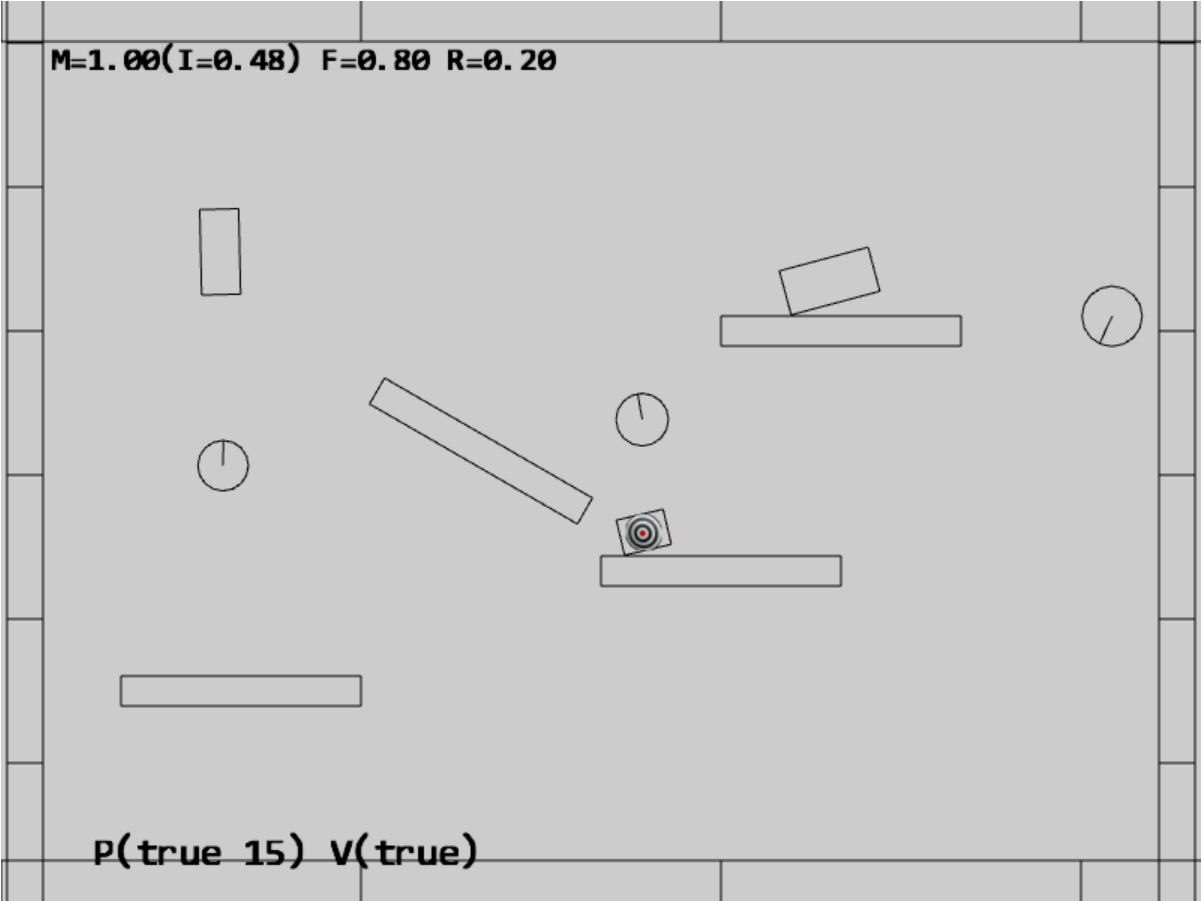


Figure 9-30. Running the Collision Angular Resolution project

The controls of the project are identical to the previous project:

* **Behavior control:**

P key: Toggle penetration resolution for all objects

**V key**: Toggle motion of all objects

**H key**: Inject random velocity to all objects

G key: Randomly create a new rigid circle or rectangle

* **Draw control**

C key: Toggle the drawing of all CollisionInfo

T key: Toggle textures on all objects

R key: Toggle the drawing of RigidShape

B key: Toggle the drawing of the bound on each RigidShape

* **Object control:**

Left/right-arrow key: Sequence through and select an object

WASD keys: Move the selected object

Z/X key: Rotate the selected object

Y/U key: Increase/decrease RigidShape size of the selected object, this does not change the size of corresponding Renderable object

Up/down-arrow key + M/N/F: Increase/decrease the mass/restitution/friction of the selected object

The goals of the project are as follows:

* To understand the details of angular impulse
* To integrate rotation into your collision resolution
* To complete the physics component

**Note** The cross product between a linear velocity on the x-y plane, , and, an angular velocity along the z-axis, , , is a vector on the x-y plane.

### Updating the Physics Component

To properly integrate angular impulse, you would only need to replace the resolveCollision() function in the physics.js file of the src/engine/components folder. While the implementation closely follows the algebraic derivation steps, it is rather long and involved. To facilitate understanding and for clarity, the following details the implementation in steps.

function resolveCollision(b, a, collisionInfo) {

    let n = collisionInfo.getNormal();

    // Step A: Compute relative velocity

    … implementation to follow …

    // Step B: Determine relative velocity in normal direction

… implementation to follow …

    // Step C: Compute collision tangent direction

… implementation to follow …

    // Step D: Determine the effective coefficients

    … implementation to follow …

    // Step E: Impulse in the normal and tangent directions

… implementation to follow …

    // Step F: Update velocity in both normal and tangent directions

    … implementation to follow …

}

1. Step A: Compute relative velocity. As highlighted in Equations (15), in the presence of angular velocity, it is important to determine the collision position (Step A1), and compute linear velocities and at the collision position (Step A2).

// Step A: Compute relative velocity

let va = a.getVelocity();

let vb = b.getVelocity();

// Step A1: Compute the intersection position p

// the direction of collisionInfo is always from b to a

// but the Mass is inverse, so start scale with a and end scale with b

let invSum = 1 / (b.getInvMass() + a.getInvMass());

let start = [0, 0], end = [0, 0], p = [0, 0];

vec2.scale(start, collisionInfo.getStart(), a.getInvMass() \* invSum);

vec2.scale(end, collisionInfo.getEnd(), b.getInvMass() \* invSum);

vec2.add(p, start, end);

// Step A2: Compute relative velocity with rotation components

// Vectors from center to P

// r is vector from center of object to collision point

let rBP = [0, 0], rAP = [0, 0];

vec2.subtract(rAP, p, a.getCenter());

vec2.subtract(rBP, p, b.getCenter());

// newV = V + mAngularVelocity cross R

let vAP1 = [-1 \* a.getAngularVelocity() \* rAP[1], a.getAngularVelocity() \* rAP[0]];

vec2.add(vAP1, vAP1, va);

let vBP1 = [-1 \* b.getAngularVelocity() \* rBP[1], b.getAngularVelocity() \* rBP[0]];

vec2.add(vBP1, vBP1, vb);

let relativeVelocity = [0, 0];

vec2.subtract(relativeVelocity, vAP1, vBP1);

1. Step B: Determine relative velocity in normal direction. A positive normal direction component signifies the objects are moving apart and the collision is resolved.

// Step B: Determine relative velocity in normal direction

let rVelocityInNormal = vec2.dot(relativeVelocity, n);

//if objects moving apart ignore

if (rVelocityInNormal > 0) {

return;

}

1. Step C: Compute collision tangent direction and the tangent direction component of the relative velocity.

// Step C: Compute collision tangent direction

let tangent = [0, 0];

vec2.scale(tangent, n, rVelocityInNormal);

vec2.subtract(tangent, tangent, relativeVelocity);

vec2.normalize(tangent, tangent);

// Relative velocity in tangent direction

let rVelocityInTangent = vec2.dot(relativeVelocity, tangent);

1. Step D: Determine the effective coefficients by using the average of the colliding objects. As in the previous project, for consistency friction coefficient is one minus the values form the RigidShape objects.

// Step D: Determine the effective coefficients

let newRestituion = (a.getRestitution() + b.getRestitution()) \* 0.5;

let newFriction = 1 - ((a.getFriction() + b.getFriction()) \* 0.5);

1. Step E: Impulse in the normal and tangent directions, these are computed by following Equations (26) and (27) exactly.

// Step E: Impulse in the normal and tangent directions

//R cross N

let rBPcrossN = rBP[0] \* n[1] - rBP[1] \* n[0]; // rBP cross n

let rAPcrossN = rAP[0] \* n[1] - rAP[1] \* n[0]; // rAP cross n

// Calc impulse scalar

// the formula of jN can be found in http://www.myphysicslab.com/collision.html

let jN = -(1 + newRestituion) \* rVelocityInNormal;

jN = jN / (b.getInvMass() + a.getInvMass() +

rBPcrossN \* rBPcrossN \* b.getInertia() +

rAPcrossN \* rAPcrossN \* a.getInertia());

let rBPcrossT = rBP[0] \* tangent[1] - rBP[1] \* tangent[0]; // rBP.cross(tangent);

let rAPcrossT = rAP[0] \* tangent[1] - rAP[1] \* tangent[0]; // rAP.cross(tangent);

let jT = (newFriction - 1) \* rVelocityInTangent;

jT = jT / (b.getInvMass() + a.getInvMass() +

rBPcrossT \* rBPcrossT \* b.getInertia() +

rAPcrossT \* rAPcrossT \* a.getInertia());

1. Step F: Update linear and angular velocities. These updates follow Equations (5), (6), (19), and (20) exactly.

// Update linear and angular velocities

vec2.scaleAndAdd(va, va, n, (jN \* a.getInvMass()));

vec2.scaleAndAdd(va, va, tangent, (jT \* a.getInvMass()));

a.setAngularVelocityDelta((rAPcrossN \* jN \* a.getInertia() + rAPcrossT \* jT \* a.getInertia()));

vec2.scaleAndAdd(vb, vb, n, -(jN \* b.getInvMass()));

vec2.scaleAndAdd(vb, vb, tangent, -(jT \* b.getInvMass()));

b.setAngularVelocityDelta(-(rBPcrossN \* jN \* b.getInertia() + rBPcrossT \* jT \* b.getInertia()));

## Observations

Run the project to test your implementation. The shapes that you insert into the scene now rotate, collide, and respond in fashions that are similar to the real world. A circle shape rolls around when other shapes collide with them, while a rectangle shape should rotate naturally upon collision. The interpenetration between shapes should not be visible under normal circumstances. However, two reasons can still cause observable interpenetrations. First, a small relaxation iteration, or second, your CPU is struggling with the number of shapes. In the first case, you can try increasing the relaxation iteration to prevent any interpenetration.

With the rotational support you can now examine the effects of mass differences in collisions. With their abilities to roll, collisions between circles are the most straightforward to observe. Wait for all objects are stationary and use the arrow key to select one of the created circles, type the M key with up-arrow to increase its mass to a large value, e.g., 20. Now, select another object and use the WASD key to move and drop the selected object on the high-mass circle. Notice that the high-mass circle does not have much response to the collision, for example, chances are the collision not even cause the high-mass circle to roll. Now, type the H key to inject random velocities to all objects and observe the collisions. Notice that the collisions with the high-mass circle are almost like collisions with stationary walls/platforms. The inversed mass and rotational inertia modelled by the Impulse Method is capable of successfully capturing the collision effects of objects with different masses.

Now your 2D physics engine implementation is completed. You can continue testing by creating additional shapes to observe when your CPU begins to struggle with keeping up real time performance.

# Summary

This chapter has guided you through understanding the foundation behind a working physics engine. The complicated physical interactions of objects in the real-world are greatly simplified by focusing only on rigid body interactions, or rigid shape simulations. The simulation process assumes that objects are continuous geometries with uniformly distributed mass where their shapes do not change during collisions. The computationally costly simulation is performed only on a selected subset of objects that are approximated by simple circles and rectangles.

A step by step derivation of the relevant formulae for the simulations is followed by detailed guide to the building of a functioning system. You have learned to extract collision information between shapes, formulate and compute shape collisions include the Separating Axis Theorem, approximate Newtonian motion integrals with the Symplectic Euler Integration, resolve interpenetrations of colliding objects based on numerically stable gradual relaxations, and derive and implement collision resolution based on the Impulse Method.

Now that you have completed your physics engine, you can carefully examine the system and identify potentials for optimization and further abstractions. Many improvements to the physics engine are still possible. This is especially true from the perspective of supporting game developers with the newly defined and powerful functionality. For example, most physics engines also support straightforward collision detections without any responses. This is an important missing functionality from your physics component. While your engine is capable of simulating collisions results, as is, the engine does not support responding to the simple, and computationally much lower cost, question of if objects have collided. As mentioned, this can be an excellent exercise.

Though simple with interface functions that can be friendlier, your physics component is functionally complete and capable of simulating rigid shape interactions with visually pleasant and realistic results. Your system supports intuitive parameters, including: object mass; acceleration; velocity; restitution; and friction; that can be straightforwardly related to the behavior of objects in the real-world. Though computationally demanding, your system is capable of supporting a non-trivial number of rigid shape interactions. This is especially the case if the game genre only required one or a small set, e.g., the hero and friendly characters, interacting with the rest of the objects, e.g., the props, platforms, and enemies.