Simulating the Rigid World

# Resolving Angular Components of Collisions

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 relevant 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. From basic mechanics, rotational inertia is also known as the angular mass, or rotational inertia. It determines the torque needed 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.

## Integrating Newtonian Mechanics for Rotation

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 shape A, is actually the velocity of the shape at its center location. In the absence of rotation, this velocity is constant throughout the shape and can be applied to any position. However, as illustrated in Figure 4-9, when the movement of a shape 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, .

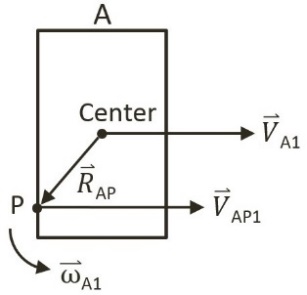


Figure 4-9 Linear Velocity at a Position in the Presence of Rotation

**Note**: Angular velocity is a vector that is perpendicular to the linear velocity. In this case, as linear velocity are defined on the X/Y plane, is a vector in the z direction since objects rotate around their center of mass. For simplicity, in your implementation, will be stored as a simple scalar representing the z-component magnitude of the vector.

## Formulating Impulse Method with Rotation

Similar to the case for linear impulse response, it is also true that change in angular velocity after a collision is inversely proportional to the rotational inertia. As illustrated in Figure 4-10, for shapes A and B with rotational inertia of and ; and initial angular velocities of and ; after a collision the angular velocities, and , are defined as follows.

Where and are positional vectors from each shape’s center of rotation to the point of collision, ; and is the normal of collision.

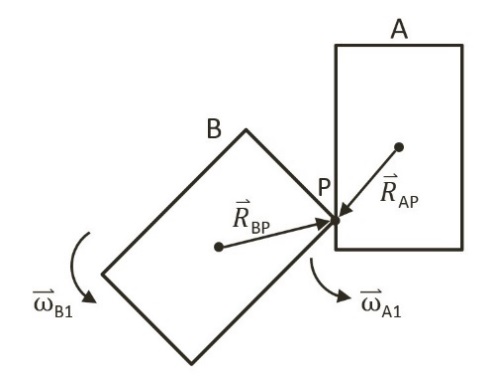


Figure 4-10 Angular Velocities of two Colliding Shapes

Recall that the Impulse Method formulation is derived based on decomposing the relative velocity after the collision, , into normal and tangent directions. With , being the relative velocity from before the collision, Equation (1) from previous section is repeated in the following.

Note that this equation was derived before the considerations for rotation and the formulation assumes that the velocity for each shape is constant over the entire shape. In order to support rotation, this equation must be generalized and solved at the point of collision, .

* ***(4)***

In this case, and are relative velocities at collision position , from before and after the collision where the following is still true for these vectors.



As previously derived, it is now possible to substitute the following equations together with the definition of the relative vectors into Equation (4) and solve for the impulse, .

Though tedious, the simplification algebra is relatively straightforward, and the resulting impulse in the collision normal direction, , can be expressed as followed.

* ***(5)***

Similar to the case in linear response, the impulse in the tangent direction, , can be derived and expressed as followed.

* ***(6)***

Once again, the coefficient of friction, , is a simplistic approximation of friction. In addition, note that since and are vectors in the X/Y plane, in implementation  
  is a scalar representing the z-component magnitude of the resulting vector.

## The Collision Angular Resolution Project

This This project will guide you through the implementation of angular impulse. You can see an example of this project running in Figure 9-X2. The source code to this project is defined in chapter9/9.8.collision\_angular\_resolution.

Figure 9-X1. Running the Collision Angular Resolution project

The controls of the project are as follows, for both scenes:

* **This and that**
* **This and that**.

The goals of the project are as follows:

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

### Modify the Physics Engine

To To implement angular impulse, in the resolve collision function, you only need to modify the Physics.js file to implement the generalized formulation derived.

1. Edit the Physics.js file and go to resolveCollision function that you have created in the previous projects.
2. It is important to compute the velocities at the collision position, and . In the following, r1 and r2 are the and positional vectors for shapes A and B. Notice that in the implementation, the collision position, , is simply the mStart position in the collisionInfo. The variables v1 and v2 are the actual and vectors.

var resolveCollision = function (s1, s2, collisionInfo) {

//..identical to previous project

var n = collisionInfo.getNormal();

//the direction of collisionInfo is always from s1 to s2

//but the Mass is inversed, so start scale with s2 and end scale with s1

var start = collisionInfo.mStart.scale(s2.mInvMass / (s1.mInvMass + s2.mInvMass));

var end = collisionInfo.mEnd.scale(s1.mInvMass / (s1.mInvMass + s2.mInvMass));

var p = start.add(end);

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

var r1 = p.subtract(s1.mCenter);

var r2 = p.subtract(s2.mCenter);

//newV = V + mAngularVelocity cross R

var v1 = s1.mVelocity.add(new Vec2(-1 \* s1.mAngularVelocity \* r1.y,

s1.mAngularVelocity \* r1.x));

var v2 = s2.mVelocity.add(new Vec2(-1 \* s2.mAngularVelocity \* r2.y,

s2.mAngularVelocity \* r2.x));

var relativeVelocity = v2.subtract(v1);

// Relative velocity in normal direction

var rVelocityInNormal = relativeVelocity.dot(n);

//..details in the next step

};

1. The next step is to compute the impulse in the collision normal direction, , according to Equation (5).

//...identical to previous project

//...continue from previous step

var newFriction = Math.min(s1.mFriction, s2.mFriction);

//R cross N

var R1crossN = r1.cross(n);

var R2crossN = r2.cross(n);

// Calc impulse scalar

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

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

jN = jN / (s1.mInvMass + s2.mInvMass +

R1crossN \* R1crossN \* s1.mInertia +

R2crossN \* R2crossN \* s2.mInertia);

//...details in the next step

1. Now, update the angular velocity according to the Impulse Method formulation introduced.

s1.mAngularVelocity -= R1crossN \* jN \* s1.mInertia;

s2.mAngularVelocity += R2crossN \* jN \* s2.mInertia;

//...details in the next step

1. Now, compute the impulse in the collision tangent direction, , according to Equation (6).

//…identical to previous project

//relativeVelocity.dot(tangent) should less than 0

tangent = tangent.normalize().scale(-1);

var R1crossT = r1.cross(tangent);

var R2crossT = r2.cross(tangent);

var jT = -(1 + newRestituion) \* relativeVelocity.dot(tangent) \* newFriction;

jT = jT / (s1.mInvMass + s2.mInvMass +

R1crossT \* R1crossT \* s1.mInertia +

R2crossT \* R2crossT \* s2.mInertia);

//...identical to previous project

1. Finally, update the angular velocity based on the tangent direction impulse

s1.mAngularVelocity -= R1crossT \* jT \* s1.mInertia;

s2.mAngularVelocity += R2crossT \* jT \* s2.mInertia;

## Observations

Run the project to test your implementation. The shape that you insert into the scene should now be rotating, colliding, and responding 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. 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 keep up real time performance.

# Summary

This chapter has guided you through understanding the foundation behind a working physics engine. A step by step derivation of the relevant formulae for the simulations followed by detailed guide to the building of a functioning system. You have computed the movement of shapes, resolved interpenetrations after collisions, implemented resolution based on the Impulse Method for shapes both linearly and rotationally. Now that you have completed your physics engine, you can integrate the system into almost any 2D game engine. Additionally, you can test your implementation by supporting other shapes. You can also carefully examine the system and identify potentials for optimization and further abstractions. Many improvements to the physics engine are still possible.

This chapter led you in building a simple yet flexible physics infrastructure for your game engine. Focusing on the implementation of core concepts, you learned to approximate Newtonian motion integrals with the Symplectic Euler Integration and how to build a numerically stable physics simulation that includes relaxation, collisions detection computation between axis-aligned rectangles and circles, extraction of collision normal and depth, and collision resolution using the Projection and Impulse methods.