Completing the Physics Engine and Rigid Shape Component

In the previous chapter, you have implemented algorithms to detect collisions between rigid circles and rectangles. In addition to the boolean condition of whether a collision has indeed occurred, the algorithms you have implemented also computed information that tells you important details--the collision information, which includes, the interpenetration depth and normal direction. In this chapter, you will further expand the physics engine by using the collision information to correct the interpenetration condition, and learn about simulating collision responses that resemble real-world rigid shape behaviors. Initially, your responses will be in linear motion, and finally you will support objects rotating as a result of collisions.

To begin with this last phase of the investigation, you will first amend the rigid shape classes to support proper simulation of Newtonian motion and to include relevant physical attributes to allow the simulation of energy transfers between colliding objects. After you implement movements in the physics engine together with the collision detection algorithms from the previous chapter you can start resolving collisions. Collisions are resolved by correcting the interpenetration state of the rigid shapes, and instituting a proper response. Interpenetrations will be corrected by moving the colliding shapes apart such that they do not overlap and collision responses will be instituted based on the Impulse Method to simulate the transfer of both linear and angular momentum.

After completing this chapter, you will be able to:

* Understand how to approximate integrals with Euler Method and Symplectic Euler Integration
* Approximate Newtonian motion formulation with Symplectic Euler Integration
* Resolve interpenetrating collisions based on a numerically stable relaxation method
* Compute and implement responses to collisions that resembles the responses of rigid bodies in the real-world
* Complete the physics engine in simulating the collisions and responses of rigid circles and rectangles

# Movement

Movement is the description of how object positions change in the simulated world. Mathematically, movement can be formulated in many ways. In previous chapters, you experienced working with movement where you continuously changed the position of an object with a constant value, or a displacement. Although desired results can be achieved, mathematically this is problematic because a velocity and a position are different types of quantities with different units and the two cannot be simply combined. As illustrated in Figure 4-1 and the following equation, in practice, you have been working with describing movement based on constant displacements.



Figure 4-1. Movement Based on Constant Displacements

A movement that is governed by the constant displacement formulation becomes restrictive when it is necessary to change the amount that is displaced over time. Newtonian mechanics address this restriction by considering time in the movement formulations, as seen in the following equations.

These two equations implement a Newtonian based movement where is the velocity that describes the change in position over time and is the acceleration that describes the change in velocity over time.

Notice that both velocity and acceleration are vector quantities encoding the change in magnitude and direction. The magnitude of a velocity vector defines the speed, and the normalized velocity vector identifies the direction that the object is traveling. An acceleration vector lets you know whether an object is speeding up or slowing down via its magnitude and the direction that the acceleration is occurring in. Acceleration is changed by the forces acting upon an object. For example, if you were to throw a ball into the air, the gravitational force of the earth would affect the object’s acceleration over time, which in turn would change the object’s velocity.

## Explicit Euler Integration

The following two equations show that the Euler method, or Explicit Euler Integration, approximates integrals based on initial values. Though potentially unstable, this is one of the simplest and thus a good beginning point to learn about integration approximation methods. As illustrated in the following two equations, in the case of the Newtonian movement formulation the new velocity, , of the object can be approximated as the current velocity, , plus the current acceleration, , multiplied by the amount of elapsed time. Similarly, the object’s new position, , can be approximated by the object’s current position, , plus the current velocity, , multiplied by the amount of elapsed time.

**Note** An example of a numerically unstable system is one where under gravitational force a bouncing ball slows down but never stops jittering and, in some cases, may even start bouncing again.

The left diagram of Figure 4-2 illustrates a simple example of approximating movements with Explicit Euler Integration. Notice that the new position is computed based on the current velocity, , while the new velocity , is computed to move the position for the next update cycle.



Figure 4-2. Explicit (Left) and Symplectic (Right) Euler Integration

## Symplectic Euler Integration

In practice, because of system stability concerns, Explicit Euler Integration is seldom implemented. This shortcoming is overcome with the method you will be implementing, known as the Semi-Implicit Euler Integration or Symplectic Euler Integration, where intermediate results are used in subsequent approximations. The following equations show Symplectic Euler Integration. Notice that it is nearly identical to the Euler method except that the new velocity, , is being used when calculating the new position, . This essentially means that the velocity for the next frame is being used to calculate the position of this frame.

The right diagram of Figure 4-2 illustrates that with the Symplectic Euler Integration, the new position is computed based on the newly computed velocity, .

# Implementing Symplectic Euler Integration and Defining Attributes to Support Collision Response

You are now ready to implement Symplectic Euler Integration. The fixed time step update function architecture of the game engine allows the quantity to be implemented as the update time interval and the integral to be evaluated once per update cycle.

In addition to implement Symplectic Euler Integration, this project also defines the attributes and their corresponding accessor and getter functions. Though relatively straightforward, these functions are presented here to avoid distracting the discussions of the more complex concepts to be covered in the subsequent projects.

You will modify the RigidShape class for this implementation.

## Rigid Shape Movements Project

This project will guide you through completing the rigid shape component to support movement calculations and collision responses. In addition to implement Symplectic Euler Integration, the information that you are going to add includes the attributes required for collision simulation and response, such as mass, inertia, friction, and restitution. As will be explained, each of these attributes will play a part in the calculation of simulating object movements and collision responses based on Euler integration. You can see an example of this project running in Figure 4-3. The source code to this project is defined in the Rigid Shape Component Project folder.

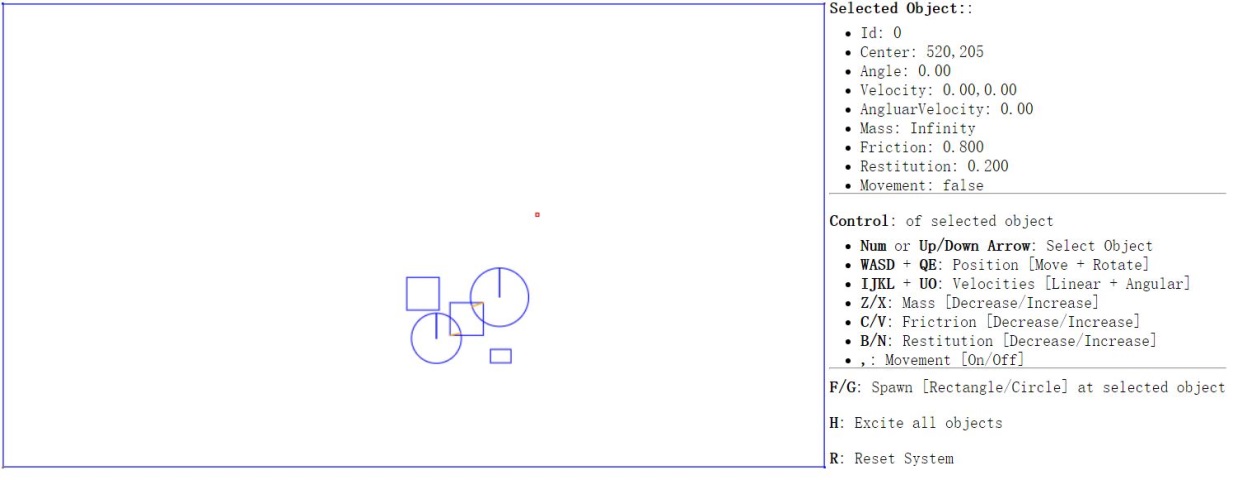


Figure 4-3. Running the Rigid Shape Component Project.

Project Goals:

* To experience implementing movements based on Symplectic Euler Integration
* To complete the implementation of RigidShape classes to include relevant physical attributes
* To build the infrastructure for responding to collisions

### Implement Symplectic Euler Integration

You must define movement support and constants in the core of the engine and in rigid shape.

#### Modify the Engine Core

Let’s start with the engine core:

1. Modify the Core.js file to include two more instance variables in the constructor. The first to support applying gravity on all objects, and the second to enable/disable object movements.

var mGravity = new Vec2(0, 10);

var mMovement = false;

1. Update the mPublic variable to allow external access to the newly defined instances.

var mPublic = {

initializeEngineCore: initializeEngineCore,

mAllObject: mAllObject,

mWidth: mWidth,

mHeight: mHeight,

mContext: mContext,

mGravity: mGravity,

mUpdateIntervalInSeconds: mUpdateIntervalInSeconds,

mMovement: mMovement

};

#### Modify the RigidShape Class

Modify the RigidShape class constructor to support velocity, angular velocity, and acceleration, as shown in the following code.

function RigidShape(center, mass, friction, restitution) {

this.mCenter = center;

this.mVelocity = new Vec2(0, 0);

this.mAcceleration = gEngine.Core.mGravity;

//angle

this.mAngle = 0;

//negetive-- clockwise

//positive-- counterclockwise

this.mAngularVelocity = 0;

this.mAngularAcceleration = 0;

gEngine.Core.mAllObject.push(this);

}

#### Implement Symplectic Euler Integration

You can now add the behavior to the rigid shape object for numerical integration. Continue with the RigidShape base class, complete the update function to apply Symplectic Euler Integration to the rigid shape where the updated velocity is used for computing the new position. Notice the implementation similarities between linear and angular motion. In both cases, the velocities are updated before the results are being applied to the displacements. Rotation will be examined in detailed in the last section of this chapter.

RigidShape.prototype.update = function () {

if (gEngine.Core.mMovement) {

var dt = gEngine.Core.mUpdateIntervalInSeconds;

//v += a\*t

this.mVelocity = this.mVelocity.add(this.mAcceleration.scale(dt));

//s += v\*t

this.move(this.mVelocity.scale(dt));

this.mAngularVelocity += this.mAngularAcceleration \* dt;

this.rotate(this.mAngularVelocity \* dt);

}

};

### Define Attributes to Support Collision Simulation and Response

As mentioned, in order to allow focused discussions of the more complex concepts in the later sections, the attributes for supporting collisions and the corresponding supporting functions are introduced in this project. These attributes are defined in the rigid shape class.

#### Modify the RigidShape Class

Now it’s time for the RigidShape class:

1. Modify the RigidShape class constructor again. This time to support mass, restitution (bounciness), and friction, as shown in the following code. Notice that the inverse of the mass value is actually stored for computation efficiency (by avoiding an extra division during each update calculation). Additionally, notice that a mass of zero is used to represent a stationary object.

function RigidShape(center, mass, friction, restitution) {

this.mCenter = center;

this.mInertia = 0;

if (mass !== undefined)

this.mInvMass = mass;

else

this.mInvMass = 1;

if (friction !== undefined)

this.mFriction = friction;

else

this.mFriction = 0.8;

if (restitution !== undefined)

this.mRestitution = restitution;

else

this.mRestitution = 0.2;

this.mVelocity = new Vec2(0, 0);

if (this.mInvMass !== 0) {

this.mInvMass = 1 / this.mInvMass;

this.mAcceleration = gEngine.Core.mGravity;

} else {

this.mAcceleration = new Vec2(0, 0);

}

//angle

this.mAngle = 0;

//negetive-- clockwise

//positive-- counterclockwise

this.mAngularVelocity = 0;

this.mAngularAcceleration = 0;

this.mBoundRadius = 0;

gEngine.Core.mAllObject.push(this);

}

1. Define a function, updateMass, to support changing of the mass during runtime. Notice that the updateInertia function is empty. This reflects the fact that rotational inertia is shape specific and the actual implementation would be the responsibility of individual subclasses (Rectangle and Circle).

RigidShape.prototype.updateMass = function (delta) {

var mass;

if (this.mInvMass !== 0)

mass = 1 / this.mInvMass;

else

mass = 0;

mass += delta;

if (mass <= 0) {

this.mInvMass = 0;

this.mVelocity = new Vec2(0, 0);

this.mAcceleration = new Vec2(0, 0);

this.mAngularVelocity = 0;

this.mAngularAcceleration = 0;

} else {

this.mInvMass = 1 / mass;

this.mAcceleration = gEngine.Core.mGravity;

}

this.updateInertia();

};

RigidShape.prototype.updateInertia = function () {

// subclass must define this.

// must work with inverted this.mInvMass

};

#### Modify the Circle and Rectangle Classes

Next, modify the Circle and Rectangle classes:

1. Modify the Circle class to implement the updateInertia function. This function calculates the rotational inertia of a circle when its mass is changed.

Circle.prototype.updateInertia = function() {

if (this.mInvMass === 0) {

this.mInertia = 0;

} else {

// this.mInvMass is inverted!!

// Inertia=mass \* radius^2

// 12 is a constant value that can be changed

this.mInertia = (1 / this.mInvMass) \* (this.mRadius \* this.mRadius) / 12;

}

};

1. Update the Circle object constructor to call the new RigidShape base class, and to accept relevant parameters of physical attributes. Remember to call the newly defined updateInertia for initialization.

var Circle = function (center, radius, mass, friction, restitution) {

RigidShape.call(this, center, mass, friction, restitution);

this.mType = "Circle";

//...identical to previous project

this.updateInertia();

};

1. Modify the Rectangle class to implement its updateIntertia function.

Rectangle.prototype.updateInertia = function() {

// Expect this.mInvMass to be already inverted!

if (this.mInvMass === 0)

this.mInertia = 0;

else {

//inertia=mass\*width^2+height^2

this.mInertia = (1 / this.mInvMass) \* (this.mWidth \* this.mWidth + this.mHeight \* this.mHeight) / 12;

this.mInertia = 1 / this.mInertia;

}

};

1. Update the Rectangle constructor in a similar manner to the Circle class to accept the relevant parameters of physical attributes and to invoke the newly defined shape specific updateIntertia function.

var Rectangle = function (center, width, height, mass, friction, restitution) {

RigidShape.call(this, center, mass, friction, restitution);

this.mType = "Rectangle";

this.mWidth = width;

this.mHeight = height;

//...indetical to previous project

this.updateInertia();

};

#### Modify the updateUIEcho Function

Since the engine has become more powerful and flexible, you want the UI to display the corresponding attributes and allow user to control these for testing purposes. Modify the updateUIEcho function in the Core.js file to print out all the options of user control.

var updateUIEcho = function () {

document.getElementById("uiEchoString").innerHTML =

"<p><b>Selected Object:</b>:</p>" +

"<ul style=\"margin:-10px\">" +

"<li>Id: " + gObjectNum + "</li>" +

"<li>Center: " + mAllObject[gObjectNum].mCenter.x.toPrecision(3) +

"," + mAllObject[gObjectNum].mCenter.y.toPrecision(3) + "</li>" +

"<li>Angle: " + mAllObject[gObjectNum].mAngle.toPrecision(3) + "</li>" +

"<li>Velocity: " + mAllObject[gObjectNum].mVelocity.x.toPrecision(3) +

"," + mAllObject[gObjectNum].mVelocity.y.toPrecision(3) + "</li>" +

"<li>AngluarVelocity: " + mAllObject[gObjectNum].mAngularVelocity.toPrecision(3) + "</li>" +

"<li>Mass: " + 1 / mAllObject[gObjectNum].mInvMass.toPrecision(3) + "</li>" +

"<li>Friction: " + mAllObject[gObjectNum].mFriction.toPrecision(3) + "</li>" +

"<li>Restitution: " + mAllObject[gObjectNum].mRestitution.toPrecision(3) + "</li>" +

"<li>Movement: " + gEngine.Core.mMovement + "</li>" +

"</ul> <hr>" +

"<p><b>Control</b>: of selected object</p>" +

"<ul style=\"margin:-10px\">" +

"<li><b>Num</b> or <b>Up/Down Arrow</b>: Select Object</li>" +

"<li><b>WASD</b> + <b>QE</b>: Position [Move + Rotate]</li>" +

"<li><b>IJKL</b> + <b>UO</b>: Velocities [Linear + Angular]</li>" +

"<li><b>Z/X</b>: Mass [Decrease/Increase]</li>" +

"<li><b>C/V</b>: Frictrion [Decrease/Increase]</li>" +

"<li><b>B/N</b>: Restitution [Decrease/Increase]</li>" +

"<li><b>,</b>: Movement [On/Off]</li>" +

"</ul> <hr>" +

"<b>F/G</b>: Spawn [Rectangle/Circle] at selected object" +

"<p><b>H</b>: Excite all objects</p>" +

"<p><b>R</b>: Reset System</p>" +

"<hr>";

};

### Modify the userControl function

For testing purposes, you want to update the UserControl.js file to allow the modification of game engine attributes during runtime. Add the following cases to the userControl function.

//… identical to previous project

if (keycode === 73) //I

gEngine.Core.mAllObject[gObjectNum].mVelocity.y -= 1;

if (keycode === 75) //k

gEngine.Core.mAllObject[gObjectNum].mVelocity.y += 1;

if (keycode === 74) //j

gEngine.Core.mAllObject[gObjectNum].mVelocity.x -= 1;

if (keycode === 76) //l

gEngine.Core.mAllObject[gObjectNum].mVelocity.x += 1;

if (keycode === 85) //U

gEngine.Core.mAllObject[gObjectNum].mAngularVelocity -= 0.1;

if (keycode === 79) //O

gEngine.Core.mAllObject[gObjectNum].mAngularVelocity += 0.1;

if (keycode === 90) //Z

gEngine.Core.mAllObject[gObjectNum].updateMass(-1);

if (keycode === 88) //X

gEngine.Core.mAllObject[gObjectNum].updateMass(1);

if (keycode === 67) //C

gEngine.Core.mAllObject[gObjectNum].mFriction -= 0.01;

if (keycode === 86) //V

gEngine.Core.mAllObject[gObjectNum].mFriction += 0.01;

if (keycode === 66) //B

gEngine.Core.mAllObject[gObjectNum].mRestitution -= 0.01;

if (keycode === 78) //N

gEngine.Core.mAllObject[gObjectNum].mRestitution += 0.01;

if (keycode === 188) //’

gEngine.Core.mMovement = !gEngine.Core.mMovement;

if (keycode === 70) //f

var r1 = new Rectangle(new Vec2(gEngine.Core.mAllObjects[gObjectNum].mCenter.x,

gEngine.Core.mAllObjects[gObjectNum].mCenter.y),

Math.random() \* 30 + 10, Math.random() \* 30 + 10,

Math.random() \* 30, Math.random(), Math.random());

if (keycode === 71) //g

var r1 = new Circle(new Vec2(gEngine.Core.mAllObjects[gObjectNum].mCenter.x,

gEngine.Core.mAllObjects[gObjectNum].mCenter.y),

Math.random() \* 10 + 20, Math.random() \* 30,

Math.random(), Math.random());

if (keycode === 72) { //H

var i;

for (i = 0; i < gEngine.Core.mAllObject.length; i++) {

if (gEngine.Core.mAllObject[i].mInvMass !== 0)

gEngine.Core.mAllObject[i].mVelocity =

new Vec2(Math.random() \* 20 - 10, Math.random() \* 20 - 10);

}

}

//… identical to previous project

## Observation

Run the project to test your implementation. Create a few objects in the scene; you can examine the attributes of your selected object. Notice that when you enable the movement by pressing the comma ( , ) key, the objects with higher downward initial velocity will drop faster because of the gravitational force or acceleration. Now create an object and set its initial y-velocity to negative. Observe that the object will move upwards until the y-component velocity reaches zero, and then it will start to fall downwards as a result of gravitational acceleration. You can also change the object’s initial x-velocity and observe the motion of a projectile. Another interesting case to try is to create a few objects and excite them by pressing the ‘H’ key. Observe how all the objects move according to their own velocities. You may see objects that move beyond the scene boundary. This is because at this point the physics engine does not support collision resolution. This will be remedied in the next section.

# Resolving Interpenetrations

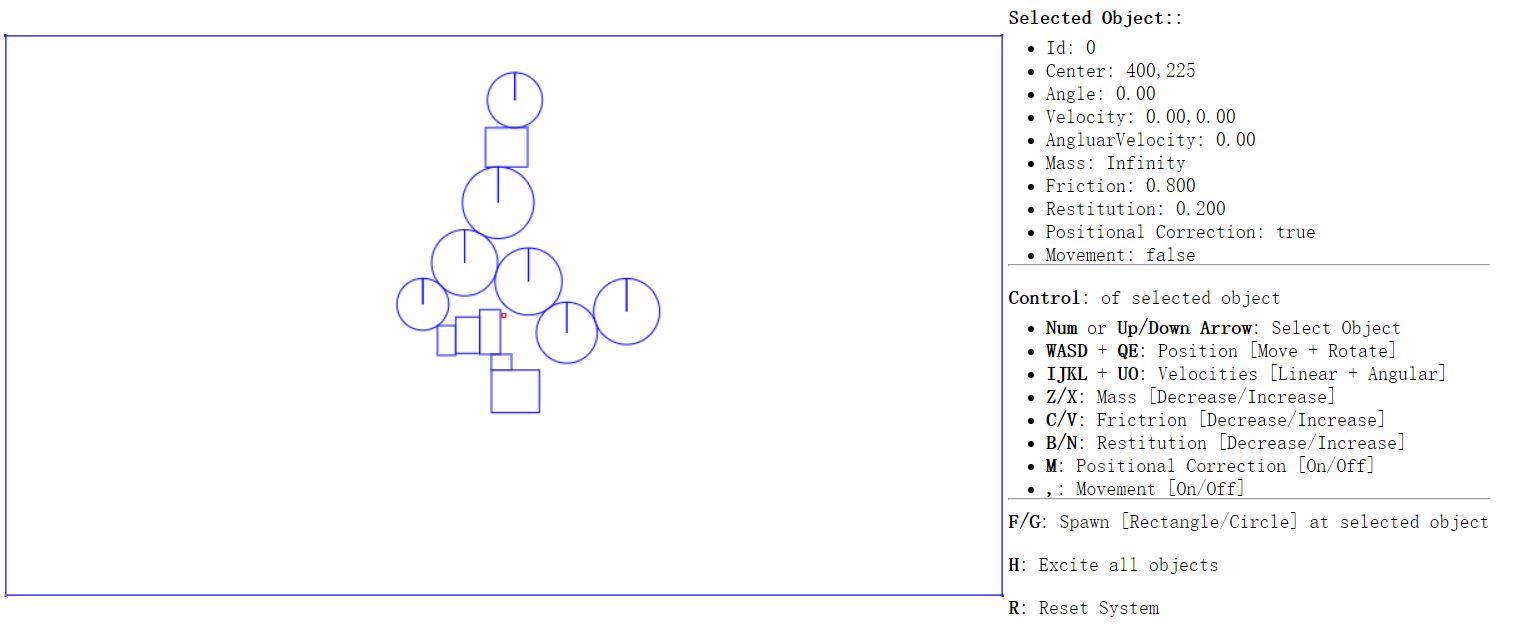
In the context of game engines, collision resolution refers to the process that determines how objects respond after a collision, including strategies to resolve the potential interpenetration situations that may occur. Notice that there are no collision resolution processes in the real world where interpenetration of rigid objects can occur since collisions are strictly governed by the law of physics. Resolutions of interpenetrations are relevant only in a virtual simulated world, where movements are approximated and impossible conditions may occur but can be resolved in ways that are desirable to the developer or designer.

In general, there are three common methods for responding to interpenetrating collisions. The first is to simply displace the objects from one another by the depth of penetration. This is known as the Projection Method since you simply move an object’s position so that it no longer penetrates the other. While this is simple to calculate and implement, it lacks stability when many objects are in proximity and resting upon one another. The simple resolving of one pair of interpenetrating objects can result in new penetrations with other nearby objects. However, this is still a common method for simple engines or games with simple object interaction rules. For example, in the Pong game, the ball never comes to rest on the paddles or walls and continuously remains in motion by bouncing off any object it collides with. The Projection Method is perfect for resolving collisions for these types of simple object interactions. The second method is known as the Impulse Method, which uses object velocities to compute and apply impulses to initiate the objects to move in the opposite directions at the point of collision. This method tends to slow down colliding objects rapidly and converges to relatively stable solutions. This is because impulses are computed based on the transfer of momentum, which in turn has a damping effect on the velocities of the colliding objects. The third method is known as the Penalty Method, which models the depth of object interpenetration as the degree of compression of a spring and approximates an acceleration to apply forces to separate the objects. This last method is the most complex and challenging to implement.

For your engine, you will be combining the strengths of the Projection and Impulse Methods. The Projection Method will be used to separate the interpenetrating objects, while the Impulse Method will be used to apply small impulses to reduce the object velocities in the direction that caused the interpenetration. As described, the simple Projection Method can result in an unstable system, such as objects that sink into each other when stacked. You will overcome this instability by implementing a relaxation loop where interpenetrated objects are separated incrementally via repeated applications of the Projection Method in a single update cycle. With a relaxation loop, the number of times that the Projection Method is applied is referred to as *relaxation iterations*. During each relaxation iteration, the Projection Method reduces the interpenetration incrementally by a fixed percentage of the total penetration depth. For example, by default the engine sets relaxation iterations to 15, and during each relaxation iteration the interpenetration is reduced by 80%. This means that within one update function call, after the movement integration approximation, the collision detection and resolution procedures will be executed 15 times. While costly, the repeated incremental separation ensures a stable system under normal circumstances. However, the 15 relaxation iterations may not be sufficient when the system undergoes sudden large changes. For example, if a large number of significantly overlapped objects, e.g., 100 overlapped circles, were to be added to the system simultaneously, then the 15 relaxation iterations may not be sufficient. This situation can be resolved by increasing the relaxation iterations at the cost of a loss in performance. From our experience, under normal operation conditions, a relaxation iteration of around 15 is a balanced trade-off between accuracy and performance.

## Positional Correction Project

This project will guide you through the implementation of the relaxation iterations to incrementally resolve inter-object interpenetrations. You are going to use the collision information computed from previous chapter to correct the position of the colliding objects. You can see an example of this project running in Figure 4-4. The source code to this project is defined in the Positional Correction Project folder.

Figure 4-4. Running the Positional Correction Project.

Project Goals:

* To appreciate the importance of the computed collision information
* To implement positional correction with relaxation iteration
* To understand and experience implementing interpenetration resolution

### Update the Physics Engine

This project will only modify Physics.js because this is the file that implements the details of collisions.

1. Edit Physics.js and add in the following variables to support the correction of positions incrementally via the relaxation iterations.

//...identical to previous project

gEngine.Physics = (function () {

var mPositionalCorrectionFlag = true;

// number of relaxation iteration

var mRelaxationCount = 15;

// percentage of separation to project objects

var mPosCorrectionRate = 0.8;

//… identical to previous project

var mPublic = {

collision: collision,

mPositionalCorrectionFlag: mPositionalCorrectionFlag

};

return mPublic;

}());

1. Modify the collision function to include an enclosing relaxation iteration loop over the collision detection loop.

var collision = function () {

var i, j, k;

for (k = 0; k < mRelaxationCount; k++) {

for (i = 0; i < gEngine.Core.mAllObject.length; i++) {

//...identical to previous project

}

}

};

1. Create a new function in gEngine.Physics and name it positionalCorrection. This function reduces the overlaps between objects by the predefined constant mPosCorrectionRate with a default value of 80%. To properly support object momentum in the simulation, the amount in which each object moves is governed by their corresponding masses. For example, upon the collision of two objects the object with a larger mass will generally move by an amount that is less than the object with smaller mass. Notice that the direction of movement is along the collision normal as defined in the collisionInfo structure.

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

var s1InvMass = s1.mInvMass;

var s2InvMass = s2.mInvMass;

var num = collisionInfo.getDepth() / (s1InvMass + s2InvMass) \* mPosCorrectionRate;

var correctionAmount = collisionInfo.getNormal().scale(num);

s1.move(correctionAmount.scale(-s1InvMass));

s2.move(correctionAmount.scale(s2InvMass));

};

1. Create another function and name it resolveCollision. This function receives two RigidShape objects as parameter, and determines if the collision detected should be positionally corrected. As pointed out previously, objects with infinite mass, or zero inversed mass, are stationary and will not participate in positional correction after a collision.

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

if ((s1.mInvMass === 0) && (s2.mInvMass === 0))

return;

// correct positions

if(gEngine.Physics.mPositionalCorrectionFlag)

positionalCorrection(s1, s2, collisionInfo);

};

1. Finally, you should call the newly defined resolveCollision function from within the collision function when a collision is detected. You can invoke resolveCollision after calling the drawCollisionInfo function.

var collision = function () {

var i, j, k;

var collisionInfo = new CollisionInfo();

for (k = 0; k < mRelaxationCount; k++) {

//….identical to previous project

drawCollisionInfo(collisionInfo, gEngine.Core.mContext);

resolveCollision(gEngine.Core.mAllObject[i], gEngine.Core.mAllObject[j], collisionInfo);

//… identical to previous project

Note that the drawCollisionInfo function is a drawing operation and strictly speaking, does not belong within the update loop in the collision function part of the system. Additionally, this draw operation is invoked within the core of relaxation loop iterations, which is computationally expensive. Fortunately, this function is for debugging purposes and will be commented out after this project.

## Observation

Run the project to test your implementation. Create a few objects in the scene. Notice that with the ‘M’ key you can control whether the newly created objects overlap. Now, reset the scene with the ‘R’ key then create some objects followed by enabling movement. You will notice small amounts of interpenetration happening and when left alone objects may begin to sink below the bottom of the scene. Select any of the objects to notice the ever increasing negative y-velocity component. During each update cycle, all objects’ y-velocities are changed by gravitational acceleration and yet the positional correction relaxation iterations are preventing them from moving downwards. By disabling the movement, you will notice overlaps disappearing completely as positional correction will not be countered anymore. The ever increasing y-velocities of the objects are a serious concern when attempting to create a stable system. Continuously increasing/decreasing numbers will result in unstable and unpredictable behavior, as witnessed in the objects sinking below the bottom boundary. In the following sections you will learn about the Impulse Method to further improve collision resolutions.

# Resolving Collisions

With a functioning positional correction system, you can now begin implementing collision resolution and support behaviors that resemble real-world situations. In order to focus on the core functionality of a collision resolution system, including understanding and implementing the Impulse Method and ensuring system stability, you will continue to work with axis-aligned rigid rectangles. The complications associated with angular impulse resolutions will be examined in the next section, after the mechanics behind linear impulse resolution are fully understood and implemented.

In the following discussion, the rectangles and circles will not rotate as a response to collisions. However, the concepts and implementation described generalize to support rotational collision responses. This project is designed to help you understand the basic concepts of impulse based collision resolution with axis-aligned shapes.

## Formulating the Impulse Method

You will formulate the solution for the Impulse Method by first reviewing how a circle can bounce off of a wall and other circles in a perfect world. This will subsequently be used to derive an approximation for an appropriate collision response. Note that the following discussion focuses on deriving the formulation for the Impulse Method and does not attempt to present a review on pure Newtonian Mechanics. Here is a brief review of some of the relevant terms.

* Mass: is the amount of matter in an object, or how dense an object is.
* Force: is any interaction or energy imparted on an object that will change the motion of that object.
* Relative Velocity: is the difference in velocity between two travel shapes.
* Coefficient of Restitution: the ratio of relative velocity after and before a collision. This is a measure of how much of the kinetic energy remains for the object to rebound from one another, or, bounciness.
* Coefficient of Friction: a number that describes the ratio of the force of friction between two bodies. In your very simplistic implementation, friction is applied directly to slow down linear motion or rotation.
* Impulse: accumulated force over time that can cause a change in the velocity. For example, resulting from a collision.

### Decomposing the Velocity in a Collision

Figure 4-5 illustrates a circle A in three different stages. At stage 1 the circle is traveling at velocity towards the wall on its right. At stage 2 the circle is colliding with the wall. At stage 3 the circle has been reflected and is traveling away from the wall with velocity .

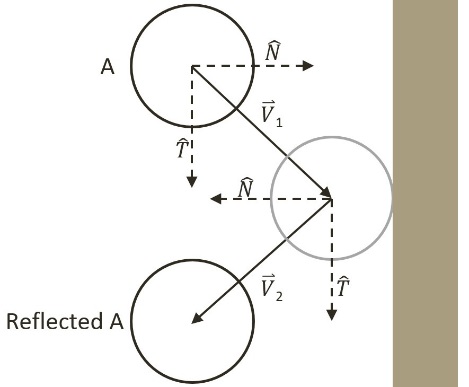


Figure 4-5 Collision Between a Circle and a Wall in a Perfect World.

Mathematically, this collision and its response can be described by decomposing the initial velocity, , into the components that are parallel, or tangent , and perpendicular, or normal , to the colliding wall. As seen in the following equation.

In a perfect world with no friction and no loss of kinetic energy, after the collision, the component along the tangent direction will not be affect while the normal component will be simply reversed. In this way, the reflected vector can be expressed as a linear combination of normal and tangent components of as followed.

Notice the negative sign in front of the component. You can see in Figure 4-5, that the component for vector points in the opposite direction of that of as a result of the collision. Notice also that the tangent component, , is still pointing in the same direction since it is parallel to the of the wall and is unaffected by the collision. This demonstrates a vector reflection.

### Relative Velocity of Colliding Shapes

This decomposition of vectors into the normal and tangent directions of the collision also applies in the general cases when the colliding shapes are both in motion. For example Figure 4-6 illustrates two traveling circle shapes, A and B, colliding.



Figure 4-6 Collision Between Two Circles

In the case of Figure 4-6, before the collision, shape A is traveling with velocity while shape B with velocity . The normal direction of the collision, , is defined to be the vector between the two circle centers and the tangent direction of the collision, , is the vector that is tangential to both of the circles at the point of collision. To resolve this collision, the velocities for shape A and B after the collision, and , must be computed.

The relative velocity between shapes A and B is defined as follows.

The collision vector decomposition can now be applied to the normal direction of the relative velocity where the relative velocity after the collision is .

* ***(1)***

The coefficient of restitution, , models the real-world situation where some kinetic energy is changed to some other form of energy during the collision. Notice that all variables on the right-hand-side of Equation (1) are defined, as they are known at the time of collision, and that the normal component of the relative velocity after the collision of shapes A and B, , is also defined. It is important to remember that,

* .

You are now ready to approximate and , the velocities of the colliding shapes after the collision.

### Approximating the Impulse Response

Accurately describing a collision involves complex considerations including factors like energy changing form, or frictions resulting from different material properties, etc. Without considering these advanced issues, a simplistic description of a collision that occurs on a shape is, a constant mass object changing its velocity from to after contact with another object. Conveniently, this is the definition of an impulse, as can be seen in the following.

Or, when solving for ,

Take a step back from the math and think about what this formula states. It makes intuitive sense. It states that the change in velocity is inversely proportional to the mass of a shape. In other words, the more mass a shape has, the less its velocity will change after a collision. The Impulse Method implements this observation, and for the normal component, it defines the velocities after a collision for shapes A and B, and , to be as followed. In this case, , and are the masses of Shapes A and B.

Subtracting the above two equations computes the normal component of relative velocity.

Recall that, is simply , and that, is , this equation simplifies to the following.

Substituting Equation (1) to the left-hand-side and remembering that and are perpendicular so that the following equation can be derived.

Collecting terms, and solving the formula for , the impulse in the normal direction gives you the following.

* ***(2)***

Finally, the impulse in the tangent direction, , can be derived in a similar manner the results of which follow.

* ***(3)***

The coefficient of friction, , is a simplistic approximation of friction.

## The Steps for Resolving Collisions

You are now ready to modify the resolveCollision function in the Physics.js file to implement the collision resolution between two colliding shapes. The resolution procedure requires access to the two RigidShape objects and the corresponding collision information. The following are the detailed steps involved:

* **Step A**: make sure at least one of the colliding shapes is not static (an inverse mass that is not equal to 0).
* **Step B**: invoke the positional correction function to snap the shapes apart by a percentage of the interpenetration depth. Recall that in your implementation, the colliding shapes will be pushed apart by a default of 80% of the interpenetration depth.
* **Step C**: compute the relative velocity between the two shapes. As presented in the derivation, the relative velocity is essential for computing the impulse in the normal and tangent directions.
* **Step D**: compute the component of the relative velocity that is in the collision normal direction. This component indicates how rapidly the two shapes are moving toward or away from each other. A positive value indicates that the shapes are moving away from each other and impulse response will not be necessary.
* **Step E**: compute the impulse in the normal direction based on results from previous step, restitution (bounciness), and the masses of the colliding shapes.
* **Step F**: compute the impulse in the tangent direction.
* **Step G**: apply impulses to modify the normal and tangent components of the shapes’ velocities to simulate the reflection of both shapes after the collision as well as friction.

The normal and tangent components of the impulse accomplish distinct purposes in simulating the results of a collision. The normal component simulates the bounciness of shapes, while the tangent component handles the friction. As illustrated in Figure 4-7, when a ball is tossed from the left towards the right, its initial spinning direction will determine the motion after the collision with the floor. On the left of Figure 4-7 the ball has an initial counter-clockwise spin while the ball on the right of the figure has an initial clockwise spin. At the point of collision with the floor, the tangent impulse component, modified by the respective friction force, will either reduce or increase the right-ward linear velocity of the ball depending on its initial spinning direction. This particular functionality will be implemented in the following section on rotational collision response. However, take note that regardless of the objects rotation upon collision the heights of the ball, after the collision, are equal to each other. This is a result of friction only affecting the tangent impulse component while the restitution affects the normal impulse component.

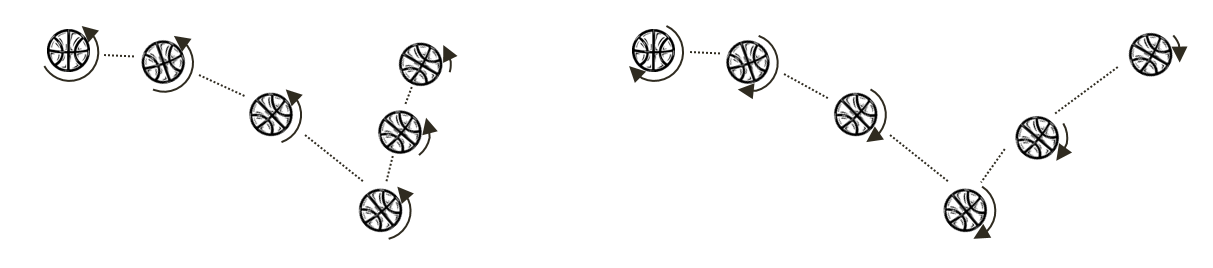


Figure 4-7. Tangent Component Impulse and Friction.

## Collision Impulse Project

This project will guide you through implementing the outlined steps to create a function that resolves the collision between axis-aligned shapes using the Impulse Method. You can see an example of this project running in Figure 4-8. The source code to this project is defined in the Collision Impulse Project folder.



Figure 4-8. Running the Collision Impulse Project.

Project Goals:

* To understand the details of Impulse Method computations
* To build a system that resolves the collision between colliding shapes

### Modify the Physics Engine Component

To properly support collision resolution, you only need to modify the physics.js file to implement the previously outlined steps.

1. Open the Physics.js file and go to the resolveCollision function.
2. After positional correction, you will begin the implementation by computing the collision normal, the relative velocity, coefficient of restitution and friction of the colliding shapes.

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

if ((s1.mInvMass === 0) && (s2.mInvMass === 0))

return;

// correct positions

if (gEngine.Physics.mPositionalCorrectionFlag)

positionalCorrection(s1, s2, collisionInfo);

var n = collisionInfo.getNormal();

var v1 = s1.mVelocity;

var v2 = s2.mVelocity;

var relativeVelocity = v2.subtract(v1);

// Relative velocity in normal direction

var rVelocityInNormal = relativeVelocity.dot(n);

// if objects moving apart ignore

if (rVelocityInNormal > 0)

return;

// compute and apply response impulses for each object

var newRestituion = Math.min(s1.mRestitution, s2.mRestitution);

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

//… details in the following steps

};

1. Compute the impulse in the direction of the collision normal based on Equation (2).

//...continue from the previous step

// Calc impulse scalar

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

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

//… details in the next step

1. Apply the impulse to the velocities of the colliding shapes.

//...continue from the previous step

//impulse is in direction of normal ( from s1 to s2)

var impulse = n.scale(jN);

// impulse = F dt = m \* △v

// △v = impulse / m

s1.mVelocity = s1.mVelocity.subtract(impulse.scale(s1.mInvMass));

s2.mVelocity = s2.mVelocity.add(impulse.scale(s2.mInvMass));

//… details in the next step

1. Compute the direction that is tangent to the collision normal.

//... continue from the previous step

var tangent = relativeVelocity.subtract(n.scale(relativeVelocity.dot(n)));

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

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

//… details in the next step

1. Compute the impulse, jT, in the direction that is tangent to the collision normal based on Equation (3), and apply the impulse to the velocities of the colliding shapes.

//...continue from the previous step

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

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

// friction should less than force in normal direction

if (jT > jN) jT = jN;

//impulse is from s1 to s2 (in opposite direction of velocity)

impulse = tangent.scale(jT);

s1.mVelocity = s1.mVelocity.subtract(impulse.scale(s1.mInvMass));

s2.mVelocity = s2.mVelocity.add(impulse.scale(s2.mInvMass));

### Defining an Initial Rectangle in Mygame.js

You need to modify Mygame.js file to define an initial rectangular RigidShape object for testing purposes. Edit Mygame.js and add the following code to define a stationary rectangle with infinite mass.

function MyGame() {

//...identical to previous project

var r2 = new Rectangle(new Vec2(200, 400), 400, 20, 0, 1, 0);

//...identical to previous project

}

## Observation

You should test your implementation in two ways. First, ensure that moving shapes collide and behave naturally. Second, ensure the collision resolution system is stable when there are many shapes that are in close proximity. You also can test the collision resolution between regular shapes and shapes with infinite mass.

Notice that the scene now has a platform like shape. This is a shape with infinite mass that can be tested for collision resolution with other regular moving shapes. Now make sure movement is switched on with the comma ( , ) key and create several rectangle and circle shapes with the ‘F’ and ‘G’ keys. Notice that the shapes fall gradually to the floor and their motions stop with a slight rebound. This is a clear indication that the base case for Euler Integration, collision detection, and resolution all are operating as expected. Press the ‘H’ key to excite all shapes. Notice the wandering shapes interact properly with the platforms and the walls of the game world with soft bounces and no apparent interpenetrations. In addition, pay attention to the apparent transfer of energy during collisions. Try adjusting the shape attributes, for example, the mass, and observe what happens when two shapes with very different masses collide. Notice that the shape with more mass does not change its trajectory much after the collision. Lastly, notice that the shapes do not rotate as a result of collision. That is because your current implementation only considers the linear velocity of the shapes. In the next project you will improve the resolution function to consider angular velocity changes as a result of collisions.

The stability of the system can be tested by increasing the number of shapes in the scene. The relaxation loop count of 15, continuously pushes interpenetrating shapes apart by 80% of the interpenetration depth during each iteration in addition to the impulse correction that is calculated. For example, you can switch off movement and positional corrections with the “,” and ‘M’ keys and create multiple, e.g., 10 to 20, overlapping shapes at the exact same position. Now enable position correction with the M key and notice that after a short pause the shapes will appear again with no interpenetrations.

# Supporting Rotation in Collision Response

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, .

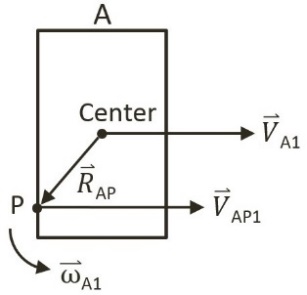


Figure 4-9 Linear Velocity at a Position in the Presence of Rotation

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.

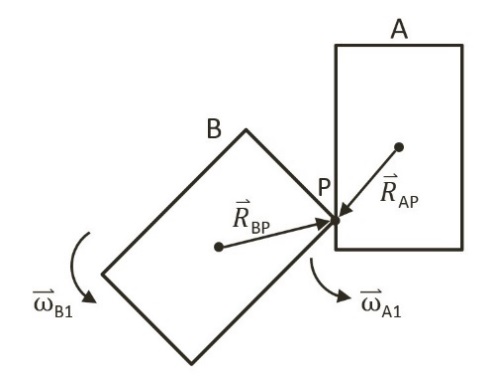


Figure 4-10 Angular Velocities of two Colliding Shapes

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.

You are now ready to implement Impulse Method collision response with support for rotation, or angular impulse.

## Angular Impulse Project

This project will guide you through the implementation of angular impulse. You can see an example of this project running in Figure 4-11. The source code to this project is defined in the Angular Impulse Project folder.

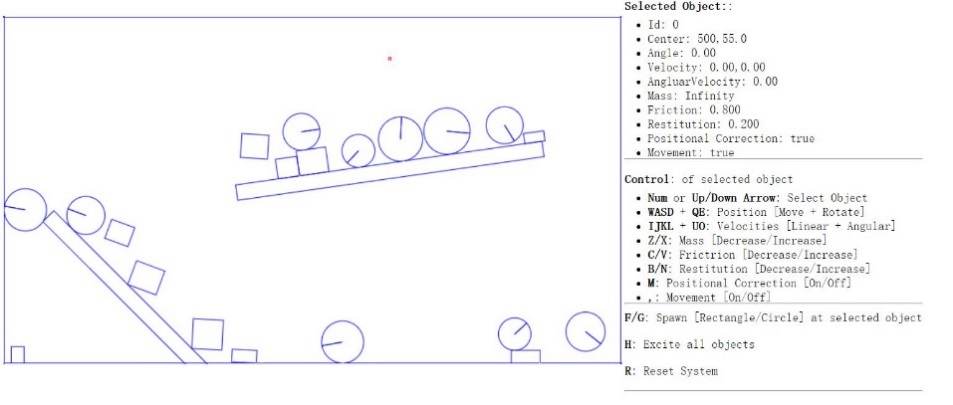


Figure 4-11. Running the Angular Impulse Project.

Project Goals:

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

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;

## Observation

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.