Physics

# Movement

Movement is the description of how object positions change in the simulated world. Mathematically, movement can be formulated in many ways. In Chapter 6, you experienced working with movement where you continuously accumulated a velocity to an object’s position. As illustrated in the following equation and in Figure 9-19, you have been working with describing movement based on constant displacements.



Figure 9-19. Movement Based on Constant Displacements

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

These two equations represent Newtonian based movements 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 both the 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 as well as the changes in the objects travelling directions. 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 would affect the object’s acceleration over time, which in turn would change the object’s velocity.

## Explicit Euler Integration

The Euler method, or Explicit Euler Integration, approximates integrals based on initial values. This is one of the most straightforward approximation for integrals. As illustrated in the following two equations, in the case of the Newtonian movement formulation, the new velocity, , of an object can be approximated as the current velocity, , plus the current acceleration, , multiplied by the elapsed time. Similarly, the object’s new position, , can be approximated by the object’s current position, , plus the current velocity, , multiplied by the elapsed time.

The left diagram of Figure 9-20 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 9-20. Explicit (Left) and Symplectic (Right) Euler Integration

## Symplectic Euler Integration

You will implement 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 9-20 illustrates that with the Symplectic Euler Integration, the new position is computed based on the newly computed velocity, .

## The Rigid Shape Movements Project

You are now ready to implement Symplectic Euler Integration to approximate movements. The fixed time step, , formulation conveniently allows the integral to be evaluated once per update cycle. This project will guide you through working with the RigidShape class to support movement approximation with the Symplectic Euler Integration. You can see an example of this project running in Figure 9-21. The source code to this project is defined in chapter9/9.5.rigid\_shape\_movements.



Figure 9-21. Running the Rigid Shape Movements project

The controls of the project are the same as previous with additional commands to control the behaviors and the mass of selected object:

* **Behavior control:**

**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**: Increase/decrease the mass of the selected object

The goals of the project are as follows:

* To complete the implementation of RigidShape classes to include relevant physical attributes
* To implement movement approximation based on Symplectic Euler Integration

In addition to implementing Symplectic Euler Integration, this project also guides you to define attributes required for collision simulation and response, such as mass, inertia, friction, etc. As will be explained, each of these attributes will play a part in the simulation of object collision responses. This straightforward information is presented here to avoid distracting discussion of the more complex concepts to be covered in the subsequent projects.

In the following section, you will first define relevant physical attributes to complete the RigidShape implementation. Afterwards you will focus on building Symplectic Euler Integration support for approximating movements.

### Completing the RigidShape Implementation

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 classes.

#### Modifying the RigidShape Class

Edit rigid\_shape.js in the src/engine/rigid\_shape folder.

1. In the constructor of the RigidShape class, define variables representing acceleration, velocity, angular velocity, mass, rotational inertia, restitution (bounciness), and friction. 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.

class RigidShape {

constructor(xf) {

this.mXform = xf;

this.mAcceleration = physics.getSystemAcceleration();

this.mVelocity = vec2.fromValues(0, 0);

this.mType = "";

this.mInvMass = 1;

this.mInertia = 0;

this.mFriction = 0.8;

this.mRestitution = 0.2;

this.mAngularVelocity = 0;

this.mBoundRadius = 0;

this.mDrawBounds = false;

}

1. Define the setMass() function to set the mass of the object. Once again, for computational efficiency the inversed of the mass is store. Setting the mass of an object to zero or negative is a signal that the object is stationary with zero acceleration and will not participate in any movement computation. Notice that when the mass of an object is changed you would need to call updateInertia() to update its rotational inertia, mInertial. Rotational inertia is geometric shape specific and that the implementation of updateIntertia() function is a subclass specific responsibility.

setMass(m) {

if (m > 0) {

this.mInvMass = 1 / m;

this.mAcceleration = physics.getSystemAcceleration();

} else {

this.mInvMass = 0;

this.mAcceleration = [0, 0]; // to ensure object does not move

}

this.updateInertia();

}

1. Define getter and setter functions for all of the other corresponding variables.

getInvMass() { return this.mInvMass; }

getInertia() { return this.mInertia; }

setInertia(i) { this.mInertia = i; }

getFriction() { return this.mFriction; }

setFriction(f) { this.mFriction = f; }

getRestitution() { return this.mRestitution; }

setRestitution(r) { this.mRestitution = r; }

getAngularVelocity() { return this.mAngularVelocity; }

setAngularVelocity(w) { this.mAngularVelocity = w; }

setAngularVelocityDelta(dw) { this.mAngularVelocity += dw; }

getVelocity() { return this.mVelocity; }

setVelocity(x, y) {

this.mVelocity[0] = x;

this.mVelocity[1] = y;

}

flipVelocity() {

this.mVelocity[0] = -this.mVelocity[0];

this.mVelocity[1] = -this.mVelocity[1];

}

getAcceleration() { return this.mAcceleration; }

setAcceleration(x, y) {

this.mAcceleration[0] = x;

this.mAcceleration[1] = y;

}

1. For the convenience of debugging, define a function, getCurrentState(), to retrieve variable values as text, and a function, userSetsState(), allow a user to set the variables.

getCurrentState() {

let m = this.mInvMass;

if (m !== 0)

m = 1 / m;

return "M=" + m.toFixed(kPrintPrecision) +

"(I=" + this.mInertia.toFixed(kPrintPrecision) + ")" +

" F=" + this.mFriction.toFixed(kPrintPrecision) +

" R=" + this.mRestitution.toFixed(kPrintPrecision);

}

userSetsState() {

// keyboard control

let delta = 0;

if (input.isKeyPressed(input.keys.Up)) {

delta = kRigidShapeUIDelta;

}

if (input.isKeyPressed(input.keys.Down)) {

delta = -kRigidShapeUIDelta;

}

if (delta !== 0) {

if (input.isKeyPressed(input.keys.M)) {

let m = 0;

if (this.mInvMass > 0)

m = 1 / this.mInvMass;

this.setMass(m + delta \* 10);

}

if (input.isKeyPressed(input.keys.F)) {

this.mFriction += delta;

if (this.mFriction < 0)

this.mFriction = 0;

if (this.mFriction > 1)

this.mFriction = 1;

}

if (input.isKeyPressed(input.keys.R)) {

this.mRestitution += delta;

if (this.mRestitution < 0)

this.mRestitution = 0;

if (this.mRestitution > 1)

this.mRestitution = 1;

}

}

}

#### Modifying the RigidCircle Classes

As mentioned, the rotational inertia, mInertial, is a specific to geometric shape and must be modified by the corresponding classes.

1. Edit rigid\_circle\_main.js in the src/engine/rigid\_shapes folder to modify the RigidCircle class to define the updateInertia() function. This function calculates the rotational inertia of a circle when its mass has changed.

updateInertia() {

if (this.mInvMass === 0) {

this.mInertia = 0;

} else {

// this.mInvMass is inverted!!

// Inertia=mass \* radius^2

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

}

};

1. Update the RigidCircle constructor and incShapeSize() function to call the updateInertia() function.

constructor(xf, radius) {

super(xf);

… identical to previous code …

this.updateInertia();

}

incShapeSizeBy(dt) {

… identical to previous code …

this.updateInertia();

}

#### Modifying the RigidRectangle Classes

Modifications similar to the RigidCircle class must be defined for the RigidRectangle class.

1. Edit rigid\_rectangle\_main.js in the src/engine/rigid\_shapes folder to define the updateInertia() function.

updateInertia() {

// 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. Similar to the RigidCircle class, update the constructor and incShapeSize() function to call the updateInertia() function.

constructor(xf, width, height) {

super(xf);

… identical to previous code …

this.updateInertia();

}

incShapeSizeBy(dt) {

… identical to previous code …

this.updateInertia();

}

### Defining System Acceleration and Motion Control

With the RigidShape implementation completed, you are now ready to define support for movement approximation.

Define a system-wide acceleration and motion control by adding appropriate variables and access functions to physics.js in the src/engine/components folder. Remember to export the newly defined functionality.

let mSystemAcceleration = [0, -20]; // system-wide default acceleration

let mHasMotion = true;

// getters and setters

function getSystemAcceleration() { return vec2.clone(mSystemAcceleration); }

function setSystemAcceleration(x, y) {

mSystemAcceleration[0] = x;

mSystemAcceleration[1] = y;

}

function getHasMotion() { return mHasMotion; }

function toggleHasMotion() { mHasMotion = !mHasMotion; }

… identical to previous code …

export {

// Physics system attributes

getSystemAcceleration, setSystemAcceleration,

getHasMotion, toggleHasMotion,  
  
 … identical to previous code …

}

### Accessing the Fixed Time Interval

In your game engine the fixed time step, , is simply the time interval in between the calls to the loopOnce() function in the game loop component. Now, edit loop.js in the src/engine/core folder to define and export the update time interval.

const kUPS = 60; // Updates per second

const kMPF = 1000 / kUPS; // Milliseconds per update.

const kSPU = 1/kUPS; // seconds per update

… identical to previous code …

function getUpdateIntervalInSeconds() { return kSPU; }

… identical to previous code …

export {getUpdateIntervalInSeconds}

### Implementing Symplectic Euler Integration in the RigidShape class

You can now integrate Symplectic Euler Integration movement approximation into the rigid shape classes. Since this movement behavior is common to all types of rigid shapes, the implementation should be located in the base class, RigidShape.

1. In the src/engine/rigid\_shapes folder, edit rigid\_shape.js to define the travel() function to implement Symplectic Euler Integration for movement. Notice how the implementation closely follows the listed equations where the updated velocity is used for computing the new position. Additionally, notice the similarity between linear and angular motion where the location (either a position or an angle) is updated by a displacement that is derived from the velocity and time step. Rotation will be examined in detail in the last section of this chapter.

travel() {

let dt = loop.getUpdateIntervalInSeconds();

// update velocity by acceleration

vec2.scaleAndAdd(this.mVelocity, this.mVelocity, this.mAcceleration, dt);

// p = p + v\*dt with new velocity

let p = this.mXform.getPosition();

vec2.scaleAndAdd(p, p, this.mVelocity, dt);

this.mXform.incRotationByRad(this.mAngularVelocity \* dt);

}

1. Modify the update() function to invoke travel() when the object is not stationary, mInvMass of 0, and when motion of the physics component is switched on.

update() {

if (this.mInvMass === 0)

return;

if (physics.getHasMotion())

this.travel();

}

### Modifying MyGame to Test Movements

The modification to the MyGame class involves supporting new user commands for toggling system-wide motion, injecting random velocity, and, setting the scene stationary boundary objects to rigid shapes with zero mass. The injecting of random velocity is implemented by the randomizeVelocity() function defined in my\_game\_bounds.js file.

All updates to the MyGame class are straightforward. To avoid unnecessary distraction, the details are not shown. As always, you can refer to the source code files in the src/my\_game folder for implementation details.

### Observations

You can now run the project to test your implementation. In order to properly observe and track movements of objects, initially motion is switched off. You can type the V key to enable motion when you are ready. When motion is toggled on, you can observe a natural-looking free-falling movement for all objects. You can type G to create more objects and observe similar free-fall movements of the created objects.

Notice that when the objects fall below the lower platform they are re-generated in the central region of the scene with a random initial upward velocity. Observe the objects move upwards until the y-component velocity reaches zero, and then they begin to fall downwards as a result of gravitational acceleration. Typing the H key injects new random upward velocities to all objects resulting in objects decelerating while moving upwards.

Try typing the C key to observe the computed collision information when objects overlap, or, interpenetrate. Pay attention and note that as objects travel through the scene interpenetration occurs frequently. You are now ready to examine and implement how to resolve object interpenetration in the next section.

# Interpenetration of Colliding Objects

The fixed update time step introduced in the previous project means that the actual location of an object in a continuous motion is approximated by a discrete set of positions. As illustrated in Figure 9-22, the movement of the rectangular object is approximated by placing the object at the three distinct positions over three update cycles. The most notable ramification of this approximation is in the challenges when determining collisions between objects.



Figure 9-22: A Rigid Square in Continuous Motion

You can see one such challenge in Figure 9-22. Imagine a thin wall existed in the space between the current and the next update. You would expect the object to collide and stop by the wall in the next update. However, if the wall was sufficiently thin, the object would appear to pass right through the wall as it jumped from one position to the next. This is a common problem faced in many game engines. A general solution for these types of problems can be algorithmically complex and computationally intensive. It is typically the job of the game designer to mitigate and avoid this problem with well-designed (for example, appropriate size) and well-behaved (for example, appropriate traveling speed) game objects.

Figure 9-23 shows another, and more significant, collision related challenge resulting from fixed update time steps. In this case, before the time step the objects are not touching. After the time step, the results of the movement approximation place the two objects where they partly overlap. In the real world, if the two objects are rigid shapes or solids then the overlap, or interpenetration, would never occur. For this reason, this situation must be properly resolved in a rigid shape physics simulation. This is where details of a collision must be computed such that interpenetrating situations like these can be properly resolved.



Figure 9-23: The Interpenetration of Colliding Objects

## Collision Position Correction

In the context of game engines, collision resolution refers to the process that determines object responses after a collision, including strategies to resolve the potential interpenetration situations that may have occurred. Notice that in the real-world, interpenetration of rigid objects can never occur since collisions are strictly governed by the law of physics. As such, resolutions of interpenetrations are relevant only in a simulated virtual world where movements are approximated and impossible situations may occur. These situations must be resolved algorithmically where both the computational cost and resulting visual appearance must be acceptable.

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 positions of objects such that they no longer overlap. While this is simple to calculate and implement, it lacks stability when many objects are in proximity and overlap with each other. In this case, the simple resolving of one pair of interpenetrating objects can result in new penetrations with other nearby objects. However, the Projection Method is still often implemented in simple engines or games with simple object interaction rules. For example, in a game of Pong, 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, the Impulse Method, uses object velocities to compute and apply impulses to cause 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, the Penalty Method, 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 compute 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, in a single update cycle, interpenetrated objects are separated incrementally via repeated applications of the Projection Method.

With a relaxation loop, each application of the Projection Method is referred to as a relaxation iteration. 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 each relaxation iteration reduces the interpenetration 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.

## The Collision Position 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 project to correct the position of the colliding objects. You can see an example of this project running in Figure 9-24. The source code to this project is defined in chapter9/9.6.collision\_position\_correction.



Figure 9-24. Running the Collision Position Correction project

The controls of the project are identical to the previous project with a single addition of the P key command in behavior control:

* **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**: Increase/decrease the mass of the selected object

The goals of the project are as follows:

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

### Updating the Physics Component

The previous projects have established the required simulation infrastructure including the completion of the RigidShape implementation in the previous project. You can now focus on the details of positional correction logic which is localized and hidden in the core of the physics component in the physics.js file in the src/engine/components folder.

1. Edit physics.js to define variables and the associated getters and setters for positional correction rate, relaxation loop count, and, toggling the positional correction computation. Make sure remember to export the newly defined functions.

let mPosCorrectionRate = 0.8; // percentage of separation to project objects

let mRelaxationCount = 15; // number of relaxation iteration

let mCorrectPosition = true;

function getPositionalCorrection() { return mCorrectPosition; }

function togglePositionalCorrection() { mCorrectPosition = !mCorrectPosition; }

function getRelaxationCount() { return mRelaxationCount; }

function incRelaxationCount(dc) { mRelaxationCount += dc; }

… identical to previous code …

export {

… identical to previous code …

togglePositionalCorrection,

getPositionalCorrection,

getRelaxationCount,

incRelaxationCount

}

1. Define the positionalCorrection() function to move and reduce the overlaps between objects by the predefined rate, mPosCorrectionRate. To properly support object momentum in the simulation, the amount in which each object moves is inversely proportional to their masses. That is, upon collision, an object with a larger mass will be moved by an amount that is less than the object with a smaller mass. Notice that the direction of movement is along the collision normal as defined in by the collisionInfo object.

function positionalCorrection(s1, s2, collisionInfo) {

if (!mCorrectPosition)

return;

let s1InvMass = s1.getInvMass();

let s2InvMass = s2.getInvMass();

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

let correctionAmount = [0, 0];

vec2.scale(correctionAmount, collisionInfo.getNormal(), num);

s1.adjustPositionBy(correctionAmount, -s1InvMass);

s2.adjustPositionBy(correctionAmount, s2InvMass);

}

1. Modify the collideShape() function to perform positional correction when a collision is detected. Notice that objects of collisions are only performed between objects with non-zero masses.

function collideShape(s1, s2, infoSet = null) {

… identical to previous code …

if ((s1 !== s2) && ((s1.getInvMass() !== 0) || (s2.getInvMass() !== 0))) {

if (s1.boundTest(s2)) {

hasCollision = s1.collisionTest(s2, mCInfo);

if (hasCollision) {

vec2.subtract(mS1toS2, s2.getCenter(), s1.getCenter());

if (vec2.dot(mS1toS2, mCInfo.getNormal()) < 0)

mCInfo.changeDir();

positionalCorrection(s1, s2, mCInfo);

… identical to previous code …

}

return hasCollision;

}

1. Integrate a loop in all three utility functions, processObjToSet(), processSetToSet(), and processSet(), to execute relaxation iterations in performing the positional corrections.

function processObjToSet(obj, set, infoSet = null) {

let j = 0, r = 0;

let hasCollision = false;

let s1 = obj.getRigidBody();

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

for (j = 0; j < set.size(); j++) {

let s2 = set.getObjectAt(j).getRigidBody();

hasCollision = collideShape(s1, s2, infoSet) || hasCollision;

}

}

return hasCollision;

}

function processSetToSet(set1, set2, infoSet = null) {

let i = 0, j = 0, r = 0;

let hasCollision = false;

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

… identical to previous code …

}

return hasCollision;

}

// collide all objects in the GameObjectSet with themselves

function processSet(set, infoSet = null) {

let i = 0, j = 0, r = 0;

let hasCollision = false;

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

… identical to previous code …

}

return hasCollision;

}

### Testing Positional Correction in MyGame

The MyGame class must be modified to support the new P key command, to toggle off initial motion and positional correct, and, to spawn initial objects in the central region of the game scene to guarantee initial collisions. These modifications are straightforward and details are not shown. As always, you can refer to the source code files in the src/my\_game folder for implementation details.

### Observations

You can now run the project to test your implementation. Notice that by default, motion is off, showing of collision information is on, and, positional correction is off. For these reasons, you will observe the created rigid shapes clumping in the central region of the game scene with many associated magenta collision information.

Now, type the P key and observe all of the shapes being pushed apart with all overlaps resolved. You can type the G key to create additional shapes and observe the shapes continuously push each other aside to ensure no overlaps. A fun experiment to perform is to toggle off positional correction, followed by typing the G key to create a large number of overlapping shapes and then to type the P key to observe the shapes pushing each other apart.

If you switch on motion with the V key you will first observe all objects free falling as a result of the gravitational force. These objects will eventually come to a rest on one of the stationary platforms. Next, you will observe the magenta collision depth increasing continuously in the vertical direction. This increase in size is a result of the continuously increasing downward velocity as a result of the downward gravitational acceleration. Eventually, the downward velocity will grow so large that in an update the object will move pass the resting platform and appear to fall right through the platform. What you are observing is precisely the situation discussed in Figure 9-22. The next subsection will discuss responses to collision and address this ever-increasing velocity.

Lastly, notice that the utility functions defined in the physics component, the processSet(), processObjToSet(), and processSetToSet() functions, these functions are designed to detect and resolve collisions. While useful, these functions are not designed to report on if a collision has occurred--a common operation supported by typical physics engines. To avoid distraction from the rigid shape simulation discussion, functions to support simple collision detection without responses are not presented. At this point, you have the necessary knowledge to define such functions and it is left as an exercise for you to complete.

# Collision Resolution

With a proper 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 begin by examining collision responses without rotations. The complications associated with angular impulse resolutions will be examined in the next section, after the mechanics behind simple 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 can be generalized in a straightforward manner to support rotational collision responses. This project is designed to help you understand the basic concepts of impulse based collision resolutions.

## 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 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 travelling shapes.
* Coefficient of Restitution: the ratio of relative velocity after and before a collision. This is a measurement of how much kinetic energy remains after an object bounces off 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.

**Note** Object rotations are described by their angular velocities and will be examined in the next section. In the rest of this section, the term velocity is used to refer to the movements of objects, or, their linear velocity.

### Components of Velocity in a Collision

Figure 9-25 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 and at stage 3 the circle has been reflected and is traveling away from the wall with velocity .



Figure 9-25 Collision Between a Circle and a Wall in a Perfect World

Mathematically, this collision and the response can be described by decomposing the initial velocity, , into the components that are perpendicular and parallel to the colliding wall. In general, the perpendicular direction to a collision is referred to as the collision normal, , and the direction that is tangential to the collision position is the collision tangent . This decomposition can be seen in the following equation.

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

Notice the negative sign in front of the component. You can see in Figure 9-25, that the component for vector points in the opposite direction of that of as a result of the collision. Additionally, notice that in the tangent direction , continues to point in the same direction. This is because the tangent component is parallel to the of the wall and is unaffected by the collision. This analysis is true in general for any collisions in a perfect world with no friction and no loss of kinetic energy.

### Relative Velocity of Colliding Shapes

The decomposition of vectors into the normal and tangent directions of the collision can also be applied to the general case of when both of the colliding shapes are in motion. For example, Figure 9-26 illustrates two traveling circle shapes, A and B, coming into a collision.



Figure 9-26 Collision Between Two Traveling Circles

In the case of Figure 9-26, before the collision object A is traveling with velocity while object 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 objects A and B after the collision, and , must be computed.

The post-collision velocities are determined based on the relative velocity between the two shapes. The relative velocity between shapes A and B is defined as follows.

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

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

The restitution, , and friction, , coefficients model the real-world situation where some kinetic energy is changed to some other forms of energy during the collision. The negative sign of Equation (1) signifies that after the collision, objects will travel in the direction that is opposite to the initial collision normal direction. Equation (2) says that after the collision, friction will scale back the magnitude where objects will continue to travel in the same tangent direction only at a lower velocity. Notice that all variables on the right-hand-side of Equations (1) and (2) are defined, as they are known at the time of collision. It is important to remember that:

* .

Where the goal is to derive a solution for and , the individual velocities of the colliding objects after a collision. You are now ready to model a solution to approximate and .

**Note** The restitution coefficient, e, describes bounciness or the proportion of the velocity that is retained after a collision. A restitution value of 1.0 would mean that velocities will be the same from before and after a collision. In contrast, friction is intuitively associated with the proportion lost, or the slow down after a collision. For example, a friction coefficient of 1.0 would mean infinite friction where a velocity of zero will result from a collision. For consistency of the formulae, the coefficient f in Equation (2) is actually 1 minus the intuitive friction coefficient.

### The Impulse

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 contacting with another object. Conveniently, this is the definition of an impulse, as can be seen in the following.

Or, when solving for ,

* ***(3)***

Remember that the same impulse also causes the velocity change in object B, only in the opposite direction,

Or, when solving for ,

* ***(4)***

Take a step back from the math and think about what this formula states. It makes intuitive sense. The equation states that the change in velocity is inversely proportional to the mass of an object. In other words, the more mass an object has, the less its velocity will change after a collision. The Impulse Method implements this observation.

Recall that Equations (1) and (2) describe the relative velocity after collision according to the collision normal and tangent directions independently. The impulse, being a vector, can also be expressed as a linear combination of components in the collision normal and tangent directions, and ,

Substituting this expression into Equations (3) and (4) results in the following,

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

Note that and are the only unknowns in these two equations where the rest of the terms are 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.

**Note** The and vectors are normalized and perpendicular to each other. For this reason, the vectors have a value of 1 when dotted with themselves, and a value of 0 when dotted with each other.

#### Normal Component of the Impulse

The normal component of the impulse, , can be solved by performing a dot product with the vector on both side of Equations (5) and (6),

Subtracting the above two equations results in the following:

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

Substituting Equation (1) for the left-hand-side to derive the following equation:

Collecting terms and solving for , the impulse in the normal direction, resulting in the following:

* ***(7)***

#### Tangent Component of the Impulse

The tangent component of the impulse, , can be solved by performing a dot product with the vector on both side of Equations (5) and (6),

Following the similar steps as in the case for the normal component, subtracting the equations, and recognizing is and is , to derive the following equation:

Now, substituting Equation (2) to the left-hand-side leaves the following:

Finally, collecting terms and solving for , or the impulse in the tangent direction results in the following:

* ***(8)***

## The Collision Resolution Project

This project will guide you through resolving a collision by calculating the impulse and updating the velocities of the colliding objects. You can see an example of this project running in Figure 9-27. The source code for this project is defined in chapter9/9.7.collision\_resolution.



Figure 9-27. Running the Collision Resolution project

The controls of the project are identical to the previous project with additional controls for restitution and friction coefficients:

* **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 the Impulse Method
* To implement the Impulse Method in resolving collisions

### Updating the Physics Component

To properly support collision resolution, you only need to focus on the physics component and modify the physics.js file in the src/engine/components folder.

1. Edit physics.js and define the resolveCollision() function to resolve the collision between RigidShape objects, a and b, with collision information recorded in the collisionInfo object.

function resolveCollision(b, a, collisionInfo) {

let n = collisionInfo.getNormal();

// Step A: Compute relative velocity

let va = a.getVelocity();

let vb = b.getVelocity();

let relativeVelocity = [0, 0];

vec2.subtract(relativeVelocity, va, vb);

// Step B: Determine relative velocity in normal direction

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

//if objects moving apart ignore

if (rVelocityInNormal > 0) {

return;

}

// 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);

// Step D: Determine the effective coefficients

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

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

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

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

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

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

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

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

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

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

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

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

}

The listed code follows the solution derivation closely.

1. Steps A and B: compute the relative velocity and its normal component. When this normal component is positive, it signifies that two objects are moving away from each other and thus collision resolution is not necessary.
2. Step C: computes the collision tangent direction and the tangent component of the relative velocity.
3. Step D: use the averages of the coefficients for impulse derivation. Notice the subtraction by one when computing the newFriction for maintaining consistency with Equation (2).
4. Step E: follows the listed Equations (7) and (8) to compute the normal and tangent components of the impulse.
5. Step F: solves for the resulting velocities by following Equations (5) and (6).
6. Edit collideShape() to invoke the resolveCollision() function when a collision is detected and position corrected.

function collideShape(s1, s2, infoSet = null) {

let hasCollision = false;  
 if ((s1 !== s2) && ((s1.getInvMass() !== 0) || (s2.getInvMass() !== 0))) {

if (s1.boundTest(s2)) {

hasCollision = s1.collisionTest(s2, mCInfo);

if (hasCollision) {

… identical to previous code …

positionalCorrection(s1, s2, mCInfo);

resolveCollision(s1, s2, mCInfo);

… identical to previous code …

};

### Updating MyGame for Testing Collision Resolution

The modifications to the MyGame class are trivial, mainly to toggle both motion and positional correction to be active by default. Additionally, initial random rotations of the created RigidShape objects are disabled because at this point collision response does not support rotation. As always, you can refer to the source code files in the src/my\_game folder for implementation details.

### Observations

You should test your implementation in three ways. First, ensure that moving shapes collide and behave naturally. Second, try changing the physical properties of the objects. Third, observe the collision resolution between shapes that are in motion and shapes that are stationary with infinite mass (the surrounding walls and stationary platforms). Remember that with only linear velocities considerations, rotations will not result from collisions.

Now, run the project and notice that the shapes fall gradually to the platforms and floor with their motions coming to a halt after slight rebounds. This is a clear indication that the base case for Euler Integration, collision detection, positional correction, and resolution all are operating as expected. Press the H key to excite all shapes and the C key to display the collision information. Notice the wandering shapes and the walls/platforms interact properly with soft bounces and no apparent interpenetrations.

Use the left/right-arrow to select an object and adjust its restitution/friction coefficients with the N/F and up/down-arrow keys. For example, adjust the restitution to 1 and friction to 0. Now inject velocity with the H key. Notice how the object seems extra bouncy and, with a friction coefficient of 0, seems to skid along platforms/floors. You can try different coefficient settings and observe corresponding bouncy and slipperiness.

The stability of the system can be tested by increasing the number of shapes in the scene with the G key. The relaxation loop count of 15

continuously and incrementally pushes interpenetrating shapes apart during each iteration. For example, you can toggle off movement and positional corrections with the V and P keys and create multiple, e.g., 10 to 20, overlapping shapes. Now toggle on motion and positional corrections and observe a properly functioning system.

In the next project you will improve the resolution function to consider angular velocity changes as a result of collisions.

# 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 .

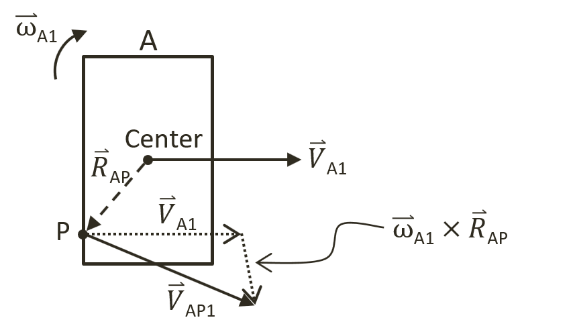


Figure 9-28 Linear Velocity at a Position in the Presence of Rotation

**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)***

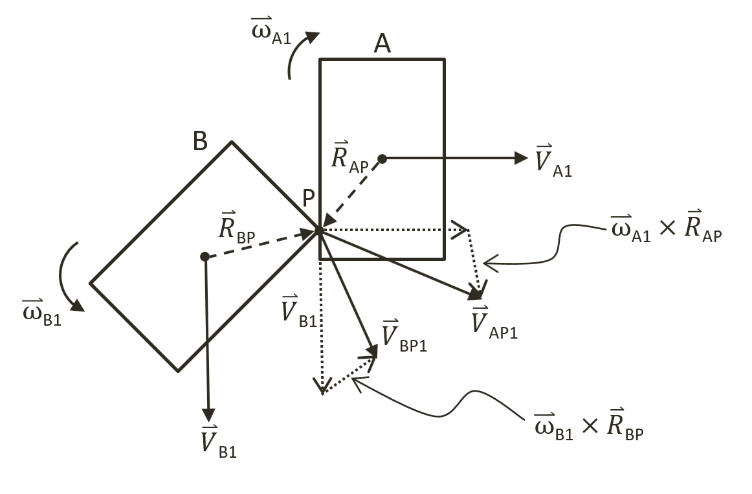


Figure 9-29 Colliding Shapes with Angular Velocities

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 that 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 are relisted for convenience 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 as shown:

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 (22) 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 definition of triple scalar product identity, this identity states that 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 the following:

* ***(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 collection of the terms with a dot-product, Equation (23) becomes the following:

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)***

Subtracting Equation (25) from (24) results in the following.

Substituting Equation (16) followed by (13) on the left-hand-side, and Equation (15) on the right-hand-side you get the following.

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.

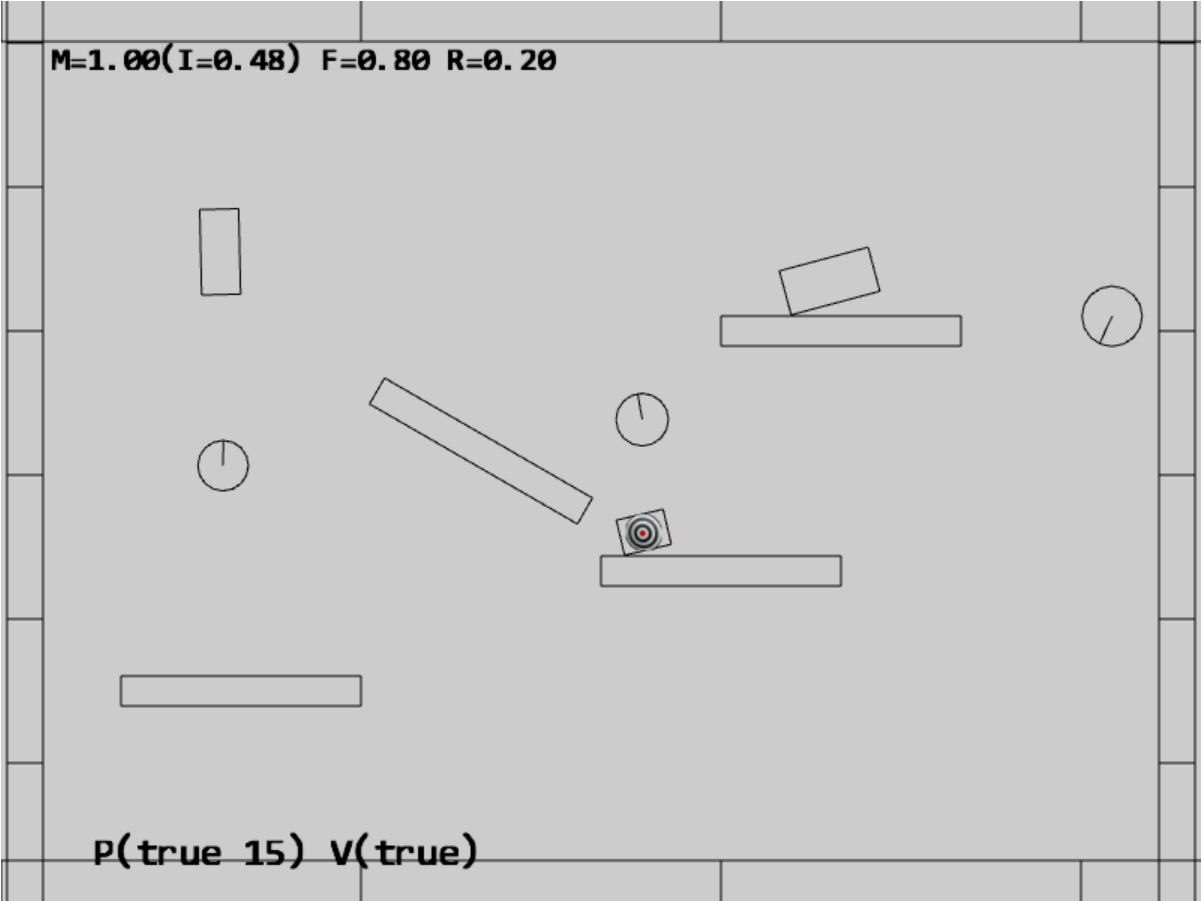


Figure 9-30. Running the Collision Angular Resolution project

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**: Apply 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 situations 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 a collision does not even cause the high-mass circle to roll. Now, type the H key to apply 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 and missing some convenient features, 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.

## Game Design Considerations

The puzzle level in the examples to this point has focused entirely on creating an understandable and consistent logical challenge; we’ve avoided burdening the exercise with any kind of visual design, narrative, or fictional setting (design elements traditionally associated with enhancing player presence) to ensure we’re thinking only about the rules of play without introducing distractions. However, as you create core game mechanics it’s important to understand how certain elements of gameplay can contribute directly to presence; the logical rules and requirements of core game mechanics often have a limited effect on presence until they’re paired with an interaction model, sound and visual design, and a setting. As discussed in Chapter 8, lighting is an example of a presence-enhancing visual design element that can also be used directly as a core game mechanic, and introducing physics to game world objects is similarly a presence-enhancing technique that’s perhaps even more often directly connected to game play.

Our experience in the real world is governed by physics, so it stands to reason that introducing similar behaviors in a game might be expected to enhance presence. An example of object physics enhancing presence but not necessarily contributing to design could be destructible environments that have no direct impact on gameplay: in a first-person shooter, for example, if the player shoots at crates and other game objects that respond by realistically exploding on impact, or if they throw a ball in the game world that bounces in a reasonable approximation of how a ball would bounce in the physical world, these are examples of physics being used purely to enhance presence but not necessarily contributing to game play. If a player is engaging with a game like Angry Birds, however, and launches one of the birds from their slingshot into the game space and they need to time the shot based on the physics-modeled parabolic arc the bird follows upon launch (as shown in Figure 9-31), this is an example of physics being used as both a core element of gameplay while also enhancing presence. In fact, any game that involves jumping a character or other game object in an environment with simulated gravity is an example of physics contributing to both presence and the core mechanic, so many platformer games utilize physics as both a core mechanic and a presence-enhancing design element.



Figure 9-31. Rovio’s Angry Birds requires players to launch projectiles from a slingshot in a virtual world that models gravity, mass, momentum, and object collision detection. The game physics are a fundamental component of the game mechanic and enhance the sense of presence by assigning physical world traits to virtual objects.

The projects in Chapter 9 introduce you to the powerful ability of physics to bring players into the game world. Instead of simply moving the hero character like a screen cursor, the player can now experience simulated inertia, momentum, and gravity requiring the same kind of predictive assessments around aiming, timing, and forward trajectory that would exist when manipulating objects in the physical world, and game objects are now capable of colliding in a manner familiar to our physical world experience. Even though specific values might take a detour from the real world in a simulated game space (e.g., lower or higher gravity, more or less inertia, and the like), as long as the relationships are consistent and reasonably analogous to our physical experience presence will typically increase when these effects are added to game objects. Imagine, for example, a game level where the hero character was required to push all the robots into a specific area within a specified time limit while avoiding being hit by projectiles. Imagine the same level without physics and it would of course be a very different experience.

We left the level design in Chapter 8 with an interesting two-stage mechanic focused almost exclusively on abstract logical rules and hadn’t yet incorporated elements that would add presence to the experience and bring players into the game world. Recall the current state of the level in Figure 9-32:



Figure 9-32. The level as it currently stands includes a two-step puzzle first requiring players to move a flashlight and reveal hidden symbols; the player must then activate the shapes in the correct sequence to unlock the barrier and claim the reward.

There is, of course, some sense of presence conveyed by the current level design: the barrier preventing players from accessing the reward is “impenetrable” and represented by a virtual wall, and the flashlight object is “shining” a virtual light beam that reveals hidden clues in the manner perhaps that a UV light in the real world might reveal special ink. Presence is frankly weak at this stage of development, however, as we have yet to place the game experience in a setting and the intentionally generic shapes don’t provide much to help a player build their own internal narrative. Our current prototype uses a flashlight-like game object to reveal hidden symbols, but it’s now possible to decouple the game mechanic’s logical rules from the current implementation and describe the core game mechanic as as “the player must explore the environment to find tools required to assemble a sequence in the correct order.”

For the next iteration of our game let’s revisit the interaction model and evolve it from purely a logic puzzle to something a bit more active that makes use of object physics. Figure 9-33 changes the game screen to include a jumping component:

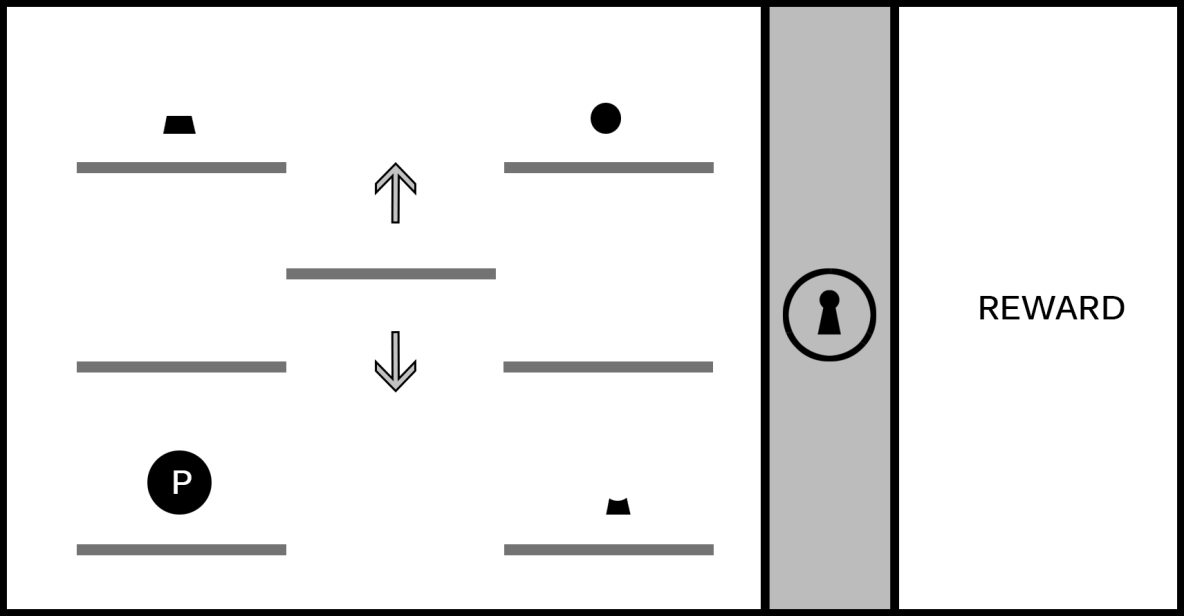


Figure 9-33. The game screen now shows just one instance of each part of the lock (top, middle, bottom), and the hero character moves in the manner of a traditional jumping 2D platformer. The six platforms on the left and right are stationary, and the middle platform moves up and down, allowing the player to ascend to higher levels. (This image assumes the player is able to “jump” the hero character between platforms on the same level but cannot reach higher levels without using the moving platform.)

We’re now evolving game play to include a dexterity challenge -- in this case, timing the jumps -- yet it retains the same logical rules from the earlier iteration: the shapes must be activated in the correct order to unlock the barrier blocking the reward. Imagine the player experiences this screen for the first time; they’ll begin exploring the screen to learn the rules of engagement for the level, including the interaction model (the keys and/or mouse buttons used to move and jump the hero character), whether missing a jump results in a penalty (for example, the loss of a “life” if the hero character misses a jump and falls off the game screen), and what it means to “activate” a shape and begin the sequence to unlock the barrier.

The game now has the beginning of an interesting (although still basic) platformer puzzle, but we’ve also now simplified the solution compared to our earlier iteration and the platformer jumping component isn’t especially challenging as shown in Figure 9-33. Recall how adding the flashlight in Chapter 8 increased the logical challenge of the original mechanic by adding a second kind of challenge requiring players to identify and use an object in the environment as a tool; we can add a similar second challenge to the platformer component, as shown in Figure 9-34:

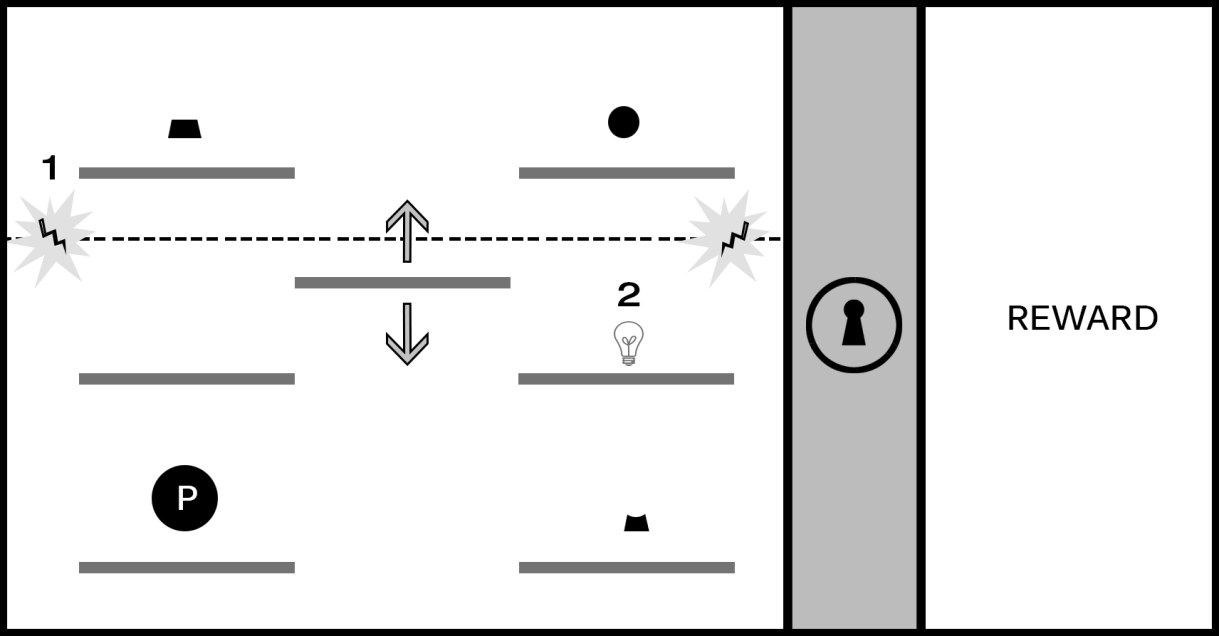


Figure 9-34. The introduction of a force field blocking access to the upper platforms (#1) can significantly increase the challenge of the platformer component. In this design, the player must activate the switch (represented with a lightbulb in #2) to disable the force field and reach the first and third shapes.

The introduction of a force field opens a variety of interesting possibilities to increase the challenge. The player must time the jump from the moving platform to the switch before hitting the force field, and the shapes must be activated in order (requiring the player to first activate top right, then the bottom right, and then the top left). Imagine a time limit is placed on the deactivation when the switch is flipped and that the puzzle will reset if all shapes aren’t activated before the force field is reengaged.

We’ve now taken an elemental mechanic based on a logical sequence and adapted it to support an action platformer experience. At this stage of development, the mechanic is becoming more interesting and beginning to feel more like a playable level, but it’s still lacking setting and context; this is a good opportunity to explore the kind of story we might want to tell with this game. Are we interested in a sci-fi adventure, perhaps a survival horror experience, or maybe a series of puzzle levels with no connected narrative? The setting will not only help inform the visual identity of the game but can also guide decisions on the kinds of challenges we create for players (for example, are “enemies” in the game working against the player, will the game play continue focusing on solving logic puzzles, or perhaps both?). A good exercise to practice connecting a game mechanic to a setting is to pick a place (for example, the interior of a space ship) and begin exploring game play in that fictional space and defining the elements of the challenge in a way that make sense for the setting. For a game on a spaceship perhaps something has gone wrong and the player must make their way from one end of the ship to the other while neutralizing security lasers through the clever use of environment objects. Experiment with applying the spaceship setting to the current game mechanic and adjusting the elements in the level to fit that theme: lasers are just one option, but can you think of other uses of our game mechanic that don’t involve an unlocking sequence? Try applying the game mechanic to a range of different environments to begin building your comfort for applying abstract game play to specific settings.

Remember also that including object physics in level designs isn’t always necessary to create a great game; sometimes you may want to subvert or completely ignore the laws of physics in the game worlds you create. The final quality of your game experience is the result of how effectively you harmonize and balance the nine elements of game design, it’s not about the mandatory implementation of any one design option. Your game might be completely abstract and involve shapes and forms shifting in space in a way that has no bearing on the physical world, but your use of color, audio, and narrative might still combine to create an experience with a strong presence for players. However, if you find yourself with a game environment that seeks to convey a sense of physicality by making use of objects that people will associate with things found in the physical world, it’s worth exploring how object physics might enhance the experience.