Simulating the World with RigidShapes

After completing this chapter, you will be able to:

* Recognize the significant computational complexity and cost of simulating real-world physical interactions
* Understand that typical game engine physics components approximate physical interaction based on simple geometries such as circles and rectangles
* Implement accurate collisions of circle and rectangular geometric shapes
* 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 behavior of rigid bodies in the real-world

# Introduction

In a game engine the functionality of simulating energy transfer is often referred to as the physics, physics system, physics component, or physics engine. Game engine physics components play an important role in many types of games. The range of topics within physics for games is broad and includes but is not limited to areas such as rigid body, soft body, fluid dynamics, and vehicle physics. Believable physical behaviors and interactions of game objects have become key elements of many modern PC and console games, as well as, more recently, browser and smartphone games. For example, the bouncing of a ball, the wiggling of a gelatinous cube, the ripples on a lake, or the skidding of a car. The proper simulation and realistic renditions of these physical phenomena are becoming common expectations.

Unfortunately, accurate simulations of the real-world can involve details that are overwhelming and require in-depth disciplinary knowledge where the underlying mathematical models can be complicated and the associated computational costs prohibitive. For example, the skid of a car depends on its speed, the tire properties, etc.; the ripples on a lake depends on its cause, the size of the lake, etc.; the wiggle of a gelatinous cube depends on its density, the initial deformation, etc. Even in the simplest of cases, the bounce of a ball depends on its material, the state of inflation, and theoretically, even on, the particle concentrations of the surrounding air. Modern game engine physics components address these complexities by restricting the types of physical interaction and simplifying the requirements for the simulation computation.

Physics engines typically restrict and simulate isolated types of physical interaction and do not support general combinations of interaction types. For example, the proper simulation of a ball bouncing (rigid body) often will not support the ball colliding and jiggling a gelatinous cube(soft body), or, accurately simulate the ripple effects caused by the ball interaction with fluid (fluid dynamics). That is, typically a rigid body physics engine does not support interactions with soft body objects, fluids, or vehicles. In the same manner, a soft body physics engine usually does not allow complex interactions with rigid body or other types of physical objects.

Additionally, physics engines typically approximate a vastly simplified interaction model while focusing mainly on attaining visually convincing results. The simplifications are usually in the forms of assumptions on object geometry and physical properties with restrictive interaction rules applied to a selective subset in the game world. For example, a rigid body physics engine typically simplifies the interactions of objects in the following ways:

* assumes objects are continuous geometries with uniformly distributed mass where the center of mass is located at the center of the geometric shape
* approximates object material properties with straightforward bounciness and friction
* dictates that objects do not change shape during interactions
* limits the simulation to a selective subset of objects in the game scene

Based on this set of assumptions, a rigid body physics simulation, or a rigid body simulation, is capable of capturing and reproducing many familiar real-world physical interactions such as objects bouncing, falling, and colliding. For example, a fully inflated bouncing ball or a simple Lego block bouncing off of a desk and landing on a hardwood floor. These types of rigid body physical interactions can be reliably simulated in real-time as long as deformation does not occur during collisions.

Objects with uniformly distributed mass that do not change shape during interactions can be applicable to many important and useful scenarios in games. In general, rigid body physics engines are excellent for simulating moving objects coming into contact with one another such as a bowling ball colliding with pins, or, a cannon ball hitting an armored plate. However, it is important to recognize that with the given set of assumptions, a rigid body physics simulation does not support the following:

* objects consisting of multiple geometric parts, e.g., an arrow
* objects with non-trivial material properties, e.g., magnetism
* objects with non-uniform mass distribution, e.g., a baseball bat
* objects that change shapes during collision, e.g., rubber balls

Of all real-world physical object interaction types, rigid body interaction is the best understood, most straightforward to approximate solutions for, and least challenging to implement. This chapter focuses only on rigid body simulation.

## Chapter Overview

Similar to illumination functionality, the physics component of a game engine is also a large and complex area of game engine design, architecture, and implementation. With this in mind, you will develop the rigid body physics component based on the same approach for all the previous game engine components. That is analyzing, understanding, and implementing individual steps to gradually realize the core functionality of the component. In the case of the physics component, the main ideas that encompass the rigid body simulation include the following.

* Rigid Shape and Bounds: Defines the RigidShape class to support an optimized simulation by performing computation on separate and simple geometries instead of the potentially complex Renderable objects. This topic will be covered by the first project, the Rigid Shape and Bounds project.
* Collision Detection: Examines and implements the mathematics to accurately collide circle and rectangle RigidShape objects. An important concept to be understand is that in the digital world, rigid shapes can and often do overlap, thus, it is essential to retain the details of this overlapping event in a CollisionInfo object. The topics covered on collision detection will be discussed by three separate projects, each focusing on a unique collision interaction. They include:
  + the collisions between circle shapes: the Circle Collision and Collision Info project
  + the collisions between rectangle shapes: the Rectangle Collision project
  + the collisions between rectangle and circle shapes: the Rectangle and Circle Collisions project.
* Movement: Movement or the simulation of physical motion approximates integrals that describe motions in a world that is updated at fixed intervals. The topic on motion will be covered by the Rigid Shape Movements project.
* Interpenetration of Colliding Objects: Observes that interpenetration between colliding rigid shapes occurs and addresses it by implementing a numerically stable relaxation loop to incrementally correct the situation. This topic is presented in the Collision Position Correct project.
* Collision Resolution: Models the responses to collision with the Impulse Method. The Impulse Method will be covered in two projects, first the simpler case without rotations in the Collision Resolution project, and finally with considerations for rotation in the Collision Angular Resolution project.

# Rigid Shapes and Bounds

The computation involved in simulating the interactions between arbitrary rigid shapes can be algorithmically complicated and computationally costly. For these reasons, rigid body simulations are typically based on a limited set of simple geometric shapes. For example, rigid circles and rectangles. In typical game engines, these simple rigid shapes can be attached to geometrically complex game objects for an approximated simulation of the physical interactions between those game objects. For example, you could attach rigid circles onto a group of spaceships(assets) and perform a rigid body physics simulation on the rigid circles to approximate the physical interactions between the spaceships.

From real-world experience you know that simple rigid shapes can interact with one another only when they come into physical contact. Algorithmically, this observation is translated into detecting collisions between rigid shapes. For a proper simulation, every shape must be tested for collision with every other shape. In this way, the collision testing is an operation, where is the number of shapes that participate in the simulation. As an optimization for this costly operation, rigid shapes are often bounded by a simple geometry, e.g., a circle, where the potentially expensive collision computation is only invoked when the bounds of the shapes overlap.

## The Rigid Shapes and Bounds Project

This project introduces the RidigShape classes with a simple circular bound for collision optimization. The defined RigidShape class will be integrated into the game engine where each GameObject object will have references to both a Renderable and a RigidShape object. The Renderable object will be drawn showing the players a visually pleasing gaming element while the RigidShape will be processed in the rigid shape simulation approximating the behavior of the GameObject object. You can see an example of this project running in Figure 9-1. The source code to this project is defined in chapter9/9.1.rigid\_shapes\_and\_bounds.



Figure 9-1. Running the Rigid Shapes and Bounds project

The controls of the project are as follows:

* **Behavior control:**

G key: Randomly create a new rigid circle or rectangle

* **Draw control**

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

The goals of the project are as follows:

* To define the RigidShape classes and integrate with GameObject.
* To demonstrate that a RigidShape represents a corresponding Renderable geometry on the same GameObject.
* To lay the foundation for building a rigid shape physics simulator.
* To define an initial scene for testing the physics component.

In addition to the system font folder, you can find the following external resource files in the assets folder:

* minion\_sprite.png is for the minion and hero objects
* platform.png and wall.png are the horizontal and vertical boarder objects in the test scene
* target.png is displayed over the currently selected object

### Setting up Implementation Support

You will begin building this project by first setting up implementation support. First, organize the engine source code structure with new folders for anticipation of increases in complexity. Second, define debugging utilities for visualization and verification of correctness. Third, extend library support for rotating rigid shapes.

#### Organizing the Engine Source Code

In anticipation for the new components, in the src/engine folder create the components folder and move the input.js component source code file into this folder. This folder will contain the source code for physics and other components to be introduced in later chapters. You will have to edit camera\_input.js, loop.js, and index.js to update the source code file location change of input.js.

#### Supporting Debug Drawing

It is important to note that only a Renderable object, typically referenced by a GameObject, is actually visible in the game world. Rigid shapes do not actually exist in the game world, they are defined to approximate the simulation of physical interactions of corresponding Renderable objects. In order to support proper debugging and verification of correctness, it is important to be able to draw and visualize the rigid shapes.

1. In the src/core folder, create debug\_draw.js, import from LineRenderable, and define supporting constants and variables for drawing simple shapes as line segments.

import LineRenderable from "../renderables/line\_renderable.js";

let kDrawNumCircleSides = 16; // for approx circumference as line segments

let mUnitCirclePos = [];

let mLine = null;

1. Define the init() function to initialize the objects for drawing. The mUnitCirclePos are positions on the circumference of a unit circle, and mLine variable is the line object that will be used for drawing.

function init() {

mLine = new LineRenderable();

mLine.setPointSize(5); // make sure when shown, its visible

let deltaTheta = (Math.PI \* 2.0) / kDrawNumCircleSides;

let theta = deltaTheta;

let i, x, y;

for (i = 1; i <= kDrawNumCircleSides; i++) {

let x = Math.cos(theta);

let y = Math.sin(theta);

mUnitCirclePos.push([x, y]);

theta = theta + deltaTheta;

}

}

1. Define the drawLine(), drawCrossMarker(), drawRectangle(), and drawCircle() functions to draw the corresponding shape based on the defined mLine object. The source code for these functions is not relevant to the physics simulation and is not shown. Please refer to the project source code folder for details.
2. Remember to export the defined functions.

export {

    init,

    drawLine, drawCrossMarker, drawCircle, drawRectangle

}

##### Initialing the Debug Drawing Functionality

Edit loop.js, import from debug\_draw.js and call the init() function after all asynchronous loading promises are fulfilled in start().

import \* as debugDraw from "./debug\_draw.js";

… identical to previous code …

async function start(scene) {

… identical to previous code …

// Wait for any async requests before game-load

await map.waitOnPromises();

// With all resources loaded, it is now possible to initialize

// System internal functions that depends on resources (shaders, etc.)

debugDraw.init(); // drawing support for rigid shapes, etc.

… identical to previous code …

}

**Note** A valid alternative for initializing debug drawing is in the createShaders() function of the shader\_resources module after all the shaders are created. However, importing from debug\_draw.js in shader\_resources.js would create a circular import: debug\_draw imports from LineRenderable that attempts to import from shader\_resources.

#### Updating the gl-matrix Library

The gl-matrix library currently supports vertex translations with its vec2 addition and vertex scaling with its vec2 scalar multiplication but does not support vertex rotations. Edit src/lib/gl-matrix.js file and define the vec2.rotateWRT() function to support rotating a vertex position, pt, by angle with respect to the ref position. Following the convention of gl-matrix, the first parameter of the function, out, returns the results of the operation.

vec2.rotateWRT = function(out, pt, angle, ref) {

var r=[];

vec2.subtract(r, pt, ref);

vec2.rotate(r, r, angle);

vec2.add(r, r, ref);

out[0] = r[0];

out[1] = r[1];

return r;

};

### Defining the RigidShape Base Class

You are now ready to define RigidShape to be the base class for the rectangle and circle rigid shapes. This base class will encapsulate all the functionality that is common to the two shapes.

1. Start by creating a new subfolder, rigid\_shapes, in src/engine. In this folder, create rigid\_shape.js, import from debug\_draw, and define drawing colors and the RigidShape class.

import \* as debugDraw from "../core/debug\_draw.js";

let kShapeColor = [0, 0, 0, 1];

let kBoundColor = [1, 1, 1, 1];

class RigidShape {

... implementation to follow …

}

export default RigidShape;

1. Define the constructor to include instance variables shared by all subclasses. The xf parameter is typically a reference to the Transform of the Renderable represented by this RigidShape. The mType variable will be initialized by subclasses to differentiate between shape types, e.g., circle vs rectangle. The mBoundRadius is the radius of the circular bound for collision optimization, and mDrawBounds indicates if the circular bound should be drawn.

constructor(xf) {

this.mXform = xf;

this.mType = "";

this.mBoundRadius = 0;

this.mDrawBounds = false;

}

1. Define appropriate getter and setter functions for the instance variables.

getType() { return this.mType; }

getCenter() { return this.mXform.getPosition(); }

getBoundRadius() { return this.mBoundRadius; }

toggleDrawBound() { this.mDrawBounds = !this.mDrawBounds; }

setBoundRadius(r) { this.mBoundRadius = r; }

setTransform(xf) { this.mXform = xf; }

setPosition(x, y) { this.mXform.setPosition(x, y); }

adjustPositionBy(v, delta) {

let p = this.mXform.getPosition();

vec2.scaleAndAdd(p, p, v, delta);

}

\_shapeColor() { return kShapeColor; }

\_boundColor() { return kBoundColor; }

1. Define the boundTest() function to determine if the circular bounds of two shapes have overlapped. As illustrated in Figure 9-2, a collision between two circles can be determine by comparing the sum of the two radii, rSum, with the distance, dist, between the centers of the spheres. Once again, this is a relatively efficient operation designed to precede the costlier accurate collision computation between two shapes.

boundTest(otherShape) {

let vFrom1to2 = [0, 0];

vec2.subtract(vFrom1to2, otherShape.mXform.getPosition(), this.mXform.getPosition());

let rSum = this.mBoundRadius + otherShape.mBoundRadius;

let dist = vec2.length(vFrom1to2);

if (dist > rSum) {

// not overlapping

return false;

}

return true;

}

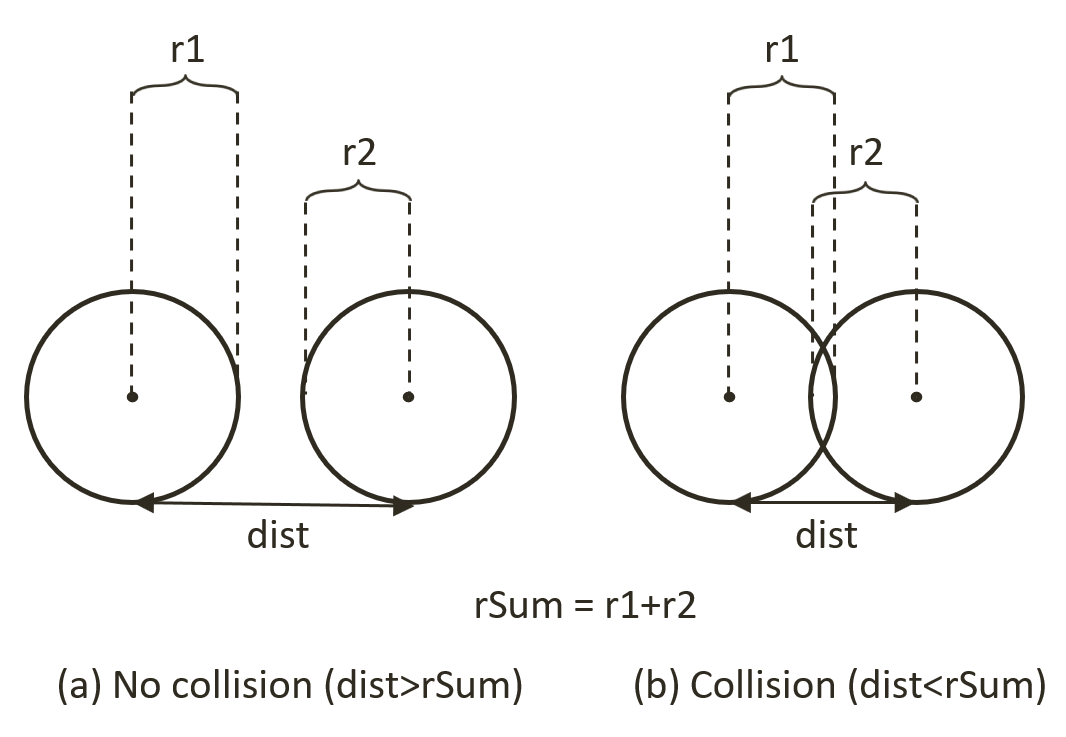


Figure 9-2. Circle Collision Detection: (a) No collision (b) Collision detected.

1. Define the update() and draw() functions. For now update() is empty. When enabled, the draw() function draws the circular bound and a “X” marker at the center of the bound.

update() { // nothing for now }

draw(aCamera) {

if (!this.mDrawBounds)

return;

debugDraw.drawCircle(aCamera, this.mXform.getPosition(), this.mBoundRadius, this.\_boundColor());

debugDraw.drawCrossMarker(aCamera, this.mXform.getPosition(),

this.mBoundRadius \* 0.2, this.\_boundColor());

}

### Defining the RigidRectangle Class

Renderable objects encode geometric information of a shape based on a Transform operator being applied on the unit square. For example, a rotated rectangle is encoded as a scaled and rotated unit square. As you have experience, this representation, where vertices of the unit square remain constant together with the matrix transformation support from the GLSL vertex shader, is effective and efficient for supporting the drawing of transformed shapes.

RigidShapes are Renderable objects designed for interactions where the underlying representation must support extensive mathematical computations. In this case, it is more efficient to explicitly represent and update the vertices of the underlying geometric shape. For example, instead of a scaled and rotated square, the vertex positions of the rectangle can be explicitly computed and stored where the values are then readily available for mathematical computations. For this reason, RigidRectangle will define and maintain the vertices of a rectangle explicitly.

With the abstract base class for rigid shapes defined, you can now create the first concrete rigid shape, the RigidRectangle class. In anticipation of complex collision functions, the implementation source code will be separated into multiple files. For now, create the rigid\_rectangle.js as the access file and import from the rigid\_rectangle\_main.js which will implement the core RigidRectangle functionality.

1. In the src/rigid\_shapes folder, create rigid\_rectangle.js to import from rigid\_rectangle\_main.js and to export the RigidRectangle class. This is the RigidRectangle class access file where users of this class should import from.

import RigidRectangle from "./rigid\_rectangle\_main.js";

export default RigidRectangle;

1. Now, create rigid\_rectangle\_main.js in the src/rigid\_shapes folder to import RigidShape and debugDraw, and define RigidRectangle to be a subclass of RigidShape.

import RigidShape from "./rigid\_shape.js";

import \* as debugDraw from "../core/debug\_draw.js";

class RigidRectangle extends RigidShape {

... implementation to follow …

}

export default RigidRectangle;

1. Define the constructor to initialize the rectangle dimension, mWidth by mHeight, and mType. It is important to recognize that the vertex positions of the rigid rectangle are controlled by the Transform referenced by mXform. In contrast, the width and height dimensions are defined independently by mWidth and mHeight. This dimension separation allows the designer to determine how tightly a RigidRectangle should wrap the corresponding Renderable. Notice that the actual vertex and face normal of the shape are computed in the setVertices() and computeFaceNormals() functions. The definition of face normal will be detailed in the following steps.

constructor(xf, width, height) {

super(xf);

this.mType = "RigidRectangle";

this.mWidth = width;

this.mHeight = height;

this.mBoundRadius = 0;

this.mVertex = [];

this.mFaceNormal = [];

this.setVertices();

this.computeFaceNormals();

}

1. Define the setVertices() function to set the vertex positions base on the dimension defined by mXform. As illustrated in Figure 9-3, the vertices on the rectangle is defined as index 0 being the top-left, 1 being top-right, 2 being bottom-right, and index 3 corresponds to the bottom-left vertex position.

setVertices() {

this.mBoundRadius = Math.sqrt(this.mWidth \* this.mWidth + this.mHeight \* this.mHeight) / 2;

let center = this.mXform.getPosition();

let hw = this.mWidth / 2;

let hh = this.mHeight / 2;

// 0--TopLeft;1--TopRight;2--BottomRight;3--BottomLeft

this.mVertex[0] = vec2.fromValues(center[0] - hw, center[1] - hh);

this.mVertex[1] = vec2.fromValues(center[0] + hw, center[1] - hh);

this.mVertex[2] = vec2.fromValues(center[0] + hw, center[1] + hh);

this.mVertex[3] = vec2.fromValues(center[0] - hw, center[1] + hh);

}



Figure 9-3. The Vertices and Face Normals of a Rectangle.

1. Define the computeFaceNormals() function. Figure 9-3 shows that the face normals of a rectangle are vectors that are perpendicular to the edges and point away from the center of the rectangle. In addition, notice the relationship between the indices of the face normals and the corresponding vertices. Face normal index-0 points in the same direction as the vector from vertex 2 to 1. This direction is perpendicular to the edge formed by vertices 0 and 1. In this way, the face normal of index-0 is perpendicular to the first edge, and so on. Notice that the face normal vectors are normalized with length of 1. The face normal vectors will be used later for determining collisions.

computeFaceNormals() {

// 0--Top;1--Right;2--Bottom;3--Left

// mFaceNormal is normal of face toward outside of rectangle

for (let i = 0; i < 4; i++) {

let v = (i + 1) % 4;

let nv = (i + 2) % 4;

this.mFaceNormal[i] = vec2.clone(this.mVertex[v]);

vec2.subtract(this.mFaceNormal[i], this.mFaceNormal[i], this.mVertex[nv]);

vec2.normalize(this.mFaceNormal[i], this.mFaceNormal[i]);

}

}

1. Define the dimension and position manipulation functions. In all cases, after the manipulation the rectangle, the vertices and face normals must be re-computed (rotateVertices() calls computeFaceNormals()). In all cases, it is critical to ensure that the vertex positions and the state of mXform are consistent.

incShapeSizeBy(dt) {

this.mHeight += dt;

this.mWidth += dt;

this.setVertices();

this.rotateVertices();

}

setPosition(x, y) {

super.setPosition(x, y);

this.setVertices();

this.rotateVertices();

}

adjustPositionBy(v, delta) {

super.adjustPositionBy(v, delta);

this.setVertices();

this.rotateVertices();

}

setTransform(xf) {

super.setTransform(xf);

this.setVertices();

this.rotateVertices();

}

rotateVertices() {

let center = this.mXform.getPosition();

let r = this.mXform.getRotationInRad();

for (let i = 0; i < 4; i++) {

vec2.rotateWRT(this.mVertex[i], this.mVertex[i], r, center);

}

this.computeFaceNormals();

}

1. Now, define the draw() function to draw the edges of the rectangle as line segments, and the update() function to update the vertices of the rectangle. The vertices and face normals must be re-computed because, as you may recall from the RigidShape base class constructor discussion, the mXfrom is a reference to the Transform of a Renderable object, the game may have manipulated the position or the rotation of the Transfrom. To ensure RigidRectangle consistently reflect the potential Transform changes, the vertices and face normals must be re-computed at each update.

draw(aCamera) {

super.draw(aCamera); // the cross marker at the center

debugDraw.drawRectangle(aCamera, this.mVertex, this.\_shapeColor());

}

update() {

super.update();

this.setVertices();

this.rotateVertices();

}

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Defining the RigidCircle Class

You can now implement the RigidCircle class with a similar overall structure to that of RigidRectangle.

1. In the src/rigid\_shapes folder, create rigid\_circle.js to import from rigid\_circle\_main.js and to export the RigidCircle class. This is the RigidCircle class access file where users of this class should import from.

import RigidCircle from "./rigid\_circle\_main.js";

export default RigidCircle;

1. Now, create rigid\_circle\_main.js in the src/rigid\_shapes folder to import RigidShape and debugDraw, and define RigidCircle to be a subclass of RigidShape.

import RigidShape from "./rigid\_shape.js";

import \* as debugDraw from "../core/debug\_draw.js";

class RigidCircle extends RigidShape {

... implementation to follow …

}

export default RigidCircle;

1. Define the constructor to initialize the circle radius, mRadius, and mType. Similar to the dimension of a RigidRectangle, the radius of RigidCircle is defined by mRadius and is independent from the size defined by the mXfrom. Note that the radii of the RigidCircle, mRadius, and the circular bound, mBoundRadius, are defined separately. This is to ensure future alternatives to separate the two.

constructor(xf, radius) {

super(xf);

this.mType = "RigidCircle";

this.mRadius = radius;

this.mBoundRadius = radius;

}

1. Define the getter and setter of the dimension.

getRadius() { return this.mRadius; }

incShapeSizeBy(dt) {

this.mRadius += dt;

this.mBoundRadius = this.mRadius;

}

1. Define the function to draw the circle as a collection of line segments along the circumference. To properly visualize the rotation of the circle, a bar is drawn from the center to the rotated vertical circumference position.

draw(aCamera) {

let p = this.mXform.getPosition();

debugDraw.drawCircle(aCamera, p, this.mRadius, this.\_shapeColor()); // the circle object

let u = [p[0], p[1] + this.mBoundRadius];

// angular motion

vec2.rotateWRT(u, u, this.mXform.getRotationInRad(), p);

debugDraw.drawLine(aCamera, p, u, false, this.\_shapeColor()); // show rotation

super.draw(aCamera); // draw last to be on top

}

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Modifying the GameObject Class to Integrate RightShape

Recall from the discussions in Chapter 6, the GameObject class is designed to encapsulate the visual appearance and behaviors of objects in the game scene. The visual appearance of a GameObject is defined by the referenced Renderable object. Thus far, the behaviors of a GameObject have been defined and implemented as part of the GameObject class in the forms of an ad hoc traveling speed, mSpeed, and simple autonomous behavior, rotateObjPointTo(). You can now replace these ad hoc parameters with the upcoming systematic physics component support.

1. Edit GameObject.js to remove the support for speed, mSpeed, as well as the corresponding setter and getter functions and the rotateObjPointTo() function. Through the changes in the rest of this chapter, the game object behaviors will be supported by the rigid body physics simulation. Make sure to leave the other variables and functions alone, they are defined to support appearance and to detect texture overlaps, pixelTouches().
2. In the constructor define new instance variables to reference to a RigidShape, and to provide drawing options.

class GameObject {

constructor(renderable) {

this.mRenderComponent = renderable;

this.mVisible = true;

this.mCurrentFrontDir = vec2.fromValues(0, 1); // this is the current front direction of the object

this.mRigidBody = null;

this.mDrawRenderable = true;

this.mDrawRigidShape = false;

}

... implementation to follow …

}

1. Define getter and setter for mRigidBody, and, functions for toggling drawing options.

getRigidBody() { return this.mRigidBody; }

setRigidBody(r) { this.mRigidBody = r; }

toggleDrawRenderable() { this.mDrawRenderable = !this.mDrawRenderable; }

toggleDrawRigidShape() { this.mDrawRigidShape = !this.mDrawRigidShape; }

1. Refine the draw() and update() functions to respect the drawing options, and, to delegate GameObject behavior update to the RigidShape class.

draw(aCamera) {

if (this.isVisible()) {

if (this.mDrawRenderable)

this.mRenderComponent.draw(aCamera);

if ((this.mRigidBody !== null) && (this.mDrawRigidShape))

this.mRigidBody.draw(aCamera);

}

}

update() {

// simple default behavior

if (this.mRigidBody !== null)

this.mRigidBody.update();

}

1. Edit the game\_object\_set.js file to modify the GameObjectSet class to support the toggling of different drawing options for the entire set.

… identical to previous code ...

toggleDrawRenderable() {

let i;

for (i = 0; i < this.mSet.length; i++) {

this.mSet[i].toggleDrawRenderable();

}

}

toggleDrawRigidShape() {

let i;

for (i = 0; i < this.mSet.length; i++) {

this.mSet[i].toggleDrawRigidShape();

}

}

toggleDrawBound() {

let i;

for (i = 0; i < this.mSet.length; i++) {

let r = this.mSet[i].getRigidBody()

if (r !== null)

r.toggleDrawBound();

}

}

### Testing of RigidShape Functionality

RigidShape is designed to approximate and to participate on behalf of a Renderable object in the rigid shape simulation. For this reason, it is essential to create and test different combinations of RigidShape types, which includes: circles and rectangles, with all combinations of Renderable types, more specifically, TextureRenderable, SpriteRenderable, and SpriteAnimateRenderable. The proper functioning of these combinations can demonstrate the correctness of the RigidShape implementation and allow you to visually examine the suitability as well as the limitations of approximating Renderable objects with simple circles and rectangles.

The overall structure of the test program, MyGame, is largely similar to previous projects where the details of the source code can be distracting and is not listed here. Instead, the following describes the tested objects and how these objects fulfill the specified requirements. As always, the source code files are located in src/my\_game folder and the supporting object classes are located in src/my\_game/objects folder.

The testing of imminent collisions requires the manipulation of the positions and rotations of each object. The WASDObj class, implemented in wasd\_obj.js, defines the WASD movement and X/Z rotation control of a GameObject. The Hero class, a subclass of WASDObj implemented in hero.js, is a GameObject with a SpriteRenderable and a RigidRectangle. The Minion class, also a subclass of WASDObj in minion.js, is a GameObject with SpriteAnimateRenderable and is wrapped by either a RigidCircle or a RigidRectangle. Based on these supporting classes, the created Hero and Minion objects encompass the different combinations of Renderable and RigidShape types, allowing you to visually inspect each RigidShape’s accuracy at representing a complex texture’s bounds. The vertical and horizontal bounds in the game scene are GameObject instances with TextureRenderable and RigidRectangle created by the wallAt() and platformAt() functions defined in my\_game\_bounds.js file. The constructor, init(), draw(), update(), etc. of MyGame are defined in the my\_game\_main.js file with largely identical functionality as in previous testing projects.

### Observations

You can now run the project and observe the created RigidShape objects. Notice that by default, only RigidShape objects are drawn. You can verify this by typing the T key to toggle on the drawing of the Renderable objects. Notice how the textures of the Renderable objects are bounded by the corresponding RigidShape instances. You can type the R key to toggle off the drawing of the RidigShape objects. Normally, this is what the players of a game will observe, with only the Renderable and without the RigidShape objects being drawn. Since the focus of this chapter is on the rigid shapes and the simulation of their interactions, the default is to show the RigidShape and not the Renderable objects.

Now type the T and R keys again to toggle back the drawing of RigidShape objects. The B key shows the circular bounds of the shapes. The more accurate and costly collision computations to be discussed in the next few sections will only be incurred between objects when these bounds overlap.

You can try using the WASD key to move the currently selected object around, by default with the Hero in the center. The X/Z and U/Y keys allow you to rotate and change the dimension of the Hero. Toggle-on the texture, with the T key, to verify that rotation and movement is applied to both the Renderable as well as its corresponding RigidShape, and that the U/Y keys only change the dimension of the RigidShape. This allows the game designer/developer control over how tightly to wrap the Renderable with the corresponding RigidShape. You can type the left/right-arrow keys to select and work with any of the objects in the scene. Finally, the G key creates new Minion objects with either a RigidCircle or a RigidRectangle.

Lastly, notice that you can move any selected object to any location, including overlapping with another RigidShape object. In the real-world, the overlapping, or interpenetration, of rigid shape objects can never occur while in the simulated digital world this is an issue that must be addressed. With the functionality of the RigidShape classes verified, you can now examine how to compute the collision between these shapes.

# Collision Detection

In order to simulate the interactions of rigid shapes, you must first detect which of the shapes are in physical contact with one another, or, which are the shapes that have collided. In general, there are two important issues to be addressed when working with rigid shape collisions: computation cost and the situations when the shapes overlap, or interpenetrate. In the following, the broad and narrow phase methods are explained as an approach to alleviate the computational cost, and collision information is introduced to record interpenetration conditions such that they can be resolved. This and the next two subsections detail the collision detection algorithms and implementations of circle-circle, rectangle-rectangle, and circle-rectangle collisions.

## Broad and Narrow Phase Methods

As discussed when introducing the circular bounds for RigidShape objects, in general every object must be tested for collision with every other object in the game scene. For example, if you want to detect the collisions between five objects, A, B, C, D, and E; you must perform four detection computations for the first object, A, against objects B, C, D, and E. With A and B’s results computed, next you must perform three collision detections between the second object B, against objects C, D, and E; followed by two collisions for the third object, C, then finally, one for the fourth object, D. The fifth object, E, has already been tested against the other four. This testing process, while thorough, has its drawbacks. Without dedicated optimizations, you must perform operations to detect the collisions between objects.

In rigid shape simulation, a detailed collision detection algorithm involving intensive computations is required. This is because accurate results must be computed to support effective interpenetration resolution and realistic collision response simulation. A broad phase method optimizes this computation by exploiting the proximity of objects to rule out those that are physically far apart from each other(within the virtual world space) and thus, clearly, cannot possibly be colliding with each other. This allows the detailed and computationally intensive algorithm, or the narrow phase method, to be deployed for objects that are physically close to each other.

A popular broad phase method uses axis-aligned bounding boxes (AABB) or bounding circles to approximate the proximity of objects. As detailed in Chapter 6, AABBs are excellent for approximating objects that are aligned with the major axes, but, has limitations when objects are rotated. As you have observed from running the previous project with the B key typed, a bounding circle is a circle that centers around and completely bounds an object. By performing the straightforward bounding box/circle intersection computations, it becomes possible to focus only on objects with overlapping bounds as the candidates for narrow phase collision detection operations.

There are other broad phase methods that organize objects either with a spatial structure such as a uniform grid or a quad-tree, or into coherent groups such as hierarchies of bounding colliders. Results from broad phase methods are typically fed into mid phase and finally narrow phase collision detection methods. Each phase narrows down candidates for the eventual collision computation, and each subsequent phase is incrementally more accurate and more expensive.

## Collision Information

In addition to reporting if objects have collided, a collision detection algorithm should also compute and return information that can be used to resolve and respond to the collision. As you have observed when testing the previous project, it is possible for RigidShape objects to overlap in space, or interpenetrate. Since real-world rigid shape objects cannot interpenetrate, recording the details and resolving RigidShape overlaps is of key importance.

As illustrated in Figure 9-4, the essential information of a collision and interpenetration include: collision depth, normal, start, and end. The collision depth is the smallest amount that the objects interpenetrated where the collision normal is the direction along which the collision depth is measured. The start and end are beginning and end positions of the interpenetration defined for the convenience of drawing the interpenetration as a line segment. It is always true that any interpenetration of convex objects can be resolved by moving the colliding objects along the collision normal by the collision depth magnitude or the distance from the start to the end position.



Figure 9-4. Collision Information

## The Circle Collisions and CollisionInfo Project

This project builds the infrastructure for computing and working with collision information based on collisions between circles. You can see an example of this project running in Figure 9-5. The source code to this project is defined in chapter9/9.2.circle\_collisions\_and\_colllision\_info.



Figure 9-5. Running the CollisionInfo and Circle Collisions project

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

* **Behavior control:**

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

The goals of the project are as follows:

* To understand the strengths and weaknesses of broad phase collision detection
* To build the infrastructure for computing inter-circle collisions
* To define and work with collision conditions via the CollisionInfo class
* To understand and implement circle collision detection algorithm

### Defining the CollisionInfo Class

A new class must be defined to record RigidShape interpenetration situations as illustrated in Figure 9-4.

1. In the src/engine/rigid\_shape folder, create the collision\_info.js file, import from debugDraw, declare the drawing color to be magenta, and define the CollisionInfo class.

import \* as debugDraw from "../core/debug\_draw.js";

let kInfoColor = [1, 0, 1, 1]; // draw the info in magenta

class CollisionInfo {

... implementation to follow …

}

export default CollisionInfo;

1. Define the constructor with instance variables that correspond to those illustrated in Figure 9-4 for collision depth, normal, and a start and end positions.

constructor() {

this.mDepth = 0;

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

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

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

}

1. Define the getter and setter for the variables.

getDepth() { return this.mDepth; }

setDepth(s) { this.mDepth = s; }

getNormal() { return this.mNormal; }

setNormal(s) { this.mNormal = s; }

getStart() { return this.mStart; }

getEnd() { return this.mEnd; }

setInfo(d, n, s) {

this.mDepth = d;

this.mNormal[0] = n[0];

this.mNormal[1] = n[1];

this.mStart[0] = s[0];

this.mStart[1] = s[1];

vec2.scaleAndAdd(this.mEnd, s, n, d);

}

1. Create a function to flip the direction of the collision normal. This function will be used to ensure that the normal is always pointing towards the object that is being tested for collision.

changeDir() {

vec2.scale(this.mNormal, this.mNormal, -1);

let n = this.mStart;

this.mStart = this.mEnd;

this.mEnd = n;

}

1. Define a draw() function to visualize the start, end, and collision normal in magenta.

draw(aCamera) {

debugDraw.drawLine(aCamera, this.mStart, this.mEnd, true, kInfoColor);

}

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Modifying the RigidShape Classes

RigidShape classes must be updated to support collisions. Since the abstract base shape, RigidShape, does not contain actual geometric information, the actual collision functions must be implemented in the rectangle and circle classes.

#### Modifying the RigidRectangle Class

For readability, collision support will be implemented in a separate source code file, rigid\_rectangle\_collision.js.

1. Modify rigid\_rectangle.js to import from the new source code file.

import RigidRectangle from "./rigid\_rectangle\_collision.js";

export default RigidRectangle;

1. In the src/engine/rigid\_shapes folder, create the rigid\_rectangle\_collision.js file, import CollisionInfo and RigidRectangle, and define the collisionTest() function to always return a collision failed status. Collisions with RigidRectangle shape will always fail until the next subsection.

RigidRectangle.prototype.collisionTest = function (otherShape, collisionInfo) {

let status = false;

if (otherShape.mType === "RigidCircle") {

status = false;

} else {

status = false;

}

return status;

}

1. Remember to export the extended RigidRectangle class for the clients.

export default RigidRectangle;

#### Modifying the RigidCircle Class

Modify the RigidCircle source code files in exactly the same manner as that of RigidRectangle: edit rigid\_circle.js to import from rigid\_circle\_collision.js. Now, you are ready to implement circle-circle collision detection.

1. In the src/engine/rigid\_shape folder, create the rigid\_circle\_collision.js file, import RigidCircle, and define the collisionTest() function to always return a collision failed status if the otherShape is not a RigidCircle, otherwise, call and return the status of collideCircCirc(). For now, a RigidCircle does not know how to collide with a RigidRectangle.

import RigidCircle from "./rigid\_circle\_main.js";

RigidCircle.prototype.collisionTest = function (otherShape, collisionInfo) {

let status = false;

if (otherShape.mType === "RigidCircle") {

status = this.collideCircCirc(this, otherShape, collisionInfo);

} else {

status = false;

}

return status;

}

1. Define the collideCircCirc() function to detect the collision between two circles and to compute the corresponding collision information when a collision is detected. There are three cases to the collision detection: no collision (step 1), collision with centers of the two circles located at different positions (step 2), and collision with the two centers located at exactly the same position (step 3). The following code shows step 1, the detection of no collision, notice that this code also corresponds to the cases as illustrated in Figure 9-2.

RigidCircle.prototype.collideCircCirc = function (c1, c2, collisionInfo) {

let vFrom1to2 = [0, 0];

// Step 1: Determine if the circles overlap

vec2.subtract(vFrom1to2, c2.getCenter(), c1.getCenter());

let rSum = c1.mRadius + c2.mRadius;

let dist = vec2.length(vFrom1to2);

if (dist > Math.sqrt(rSum \* rSum)) {

// not overlapping

return false;

}

… implementation of Steps 2 and 3 to follow …

}

1. When a collision is detected, if the two circle centers are located at different positions (step 2), the collision depth and normal can be computed as illustrated in Figure 9-6. Since c2 is the reference to the other RigidShape, the collision normal is a vector pointing from c1 towards c2, or in the same direction as vFrom1to2. The collision depth is the difference between rSum and dist, and the start position for c1 is simply c2’s radius distance away from the center of c2 along the negative mFrom1to2 direction.



Figure 9-6. Details of a Circle-Circle Collision

// Step 1: refer to previous step

if (dist !== 0) {

// Step 2: Colliding circle centers are at different positions

vec2.normalize(vFrom1to2, vFrom1to2);

let vToC2 = [0, 0];

vec2.scale(vToC2, vFrom1to2, -c2.mRadius);

vec2.add(vToC2, c2.getCenter(), vToC2);

collisionInfo.setInfo(rSum - dist, vFrom1to2, vToC2);

}

… implementation of Step 3 to follow …

1. The last case for two colliding circles is when both circle centers are located at exactly the same position (step 3). In this case, the collision normal is defined to be the negative y-direction, and the collision depth is simply the larger of the two radii.

// Step 1: refer to previous step

if (dist !== 0) {

// Step 2: refer to previous step

} else {

let n = [0, -1];

// Step 3: Colliding circle centers are at exactly the same position

if (c1.mRadius > c2.mRadius) {

let pC1 = c1.getCenter();

let ptOnC1 = [pC1[0], pC1[1] + c1.mRadius];

collisionInfo.setInfo(rSum, n, ptOnC1);

} else {

let pC2 = c2.getCenter();

let ptOnC2 = [pC2[0], pC2[1]+ c2.mRadius];

collisionInfo.setInfo(rSum, n, ptOnC2);

}

}

### Defining the Physics Component

You can now define the physics component to trigger the collision detection computations.

1. In the src/engine/components folder, create the physics.js file, import CollisionInfo and declare variables to support computations that are local to this file.
2. Define the collideShape() function to trigger the collision detection computation. Take note the two tests prior to the actual calling of shape collisionTest(). First, check to ensure the two shapes are not actually the same object. Second, call to the broad phase boundTest() method to determine the proximity of the shapes. Notice that the last parameter, infoSet, when defined will contain all CollisionInfo objects for all successful collisions. This is defined to support visualizing the CollisionInfo objects for verification and debugging purposes.

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

let hasCollision = false;

if (s1 !== s2) {

if (s1.boundTest(s2)) {

hasCollision = s1.collisionTest(s2, mCInfo);

if (hasCollision) {

// make sure mCInfo is always from s1 towards s2

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

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

mCInfo.changeDir();

// for showing off collision mCInfo!

if (infoSet !== null) {

infoSet.push(mCInfo);

mCInfo = new CollisionInfo();

}

}

}

}

return hasCollision;

}

1. Define utility functions to support the game developer: processSet() to perform collision determination between all objects in the same GameObjectSet; processObjToSet() to check between a given GameObject and objects of a GameObjectSet; and, processSetToSet() to check between all objects in two different GameObjectSets.

// collide all objects in the GameObjectSet with themselves

function processSet(set, infoSet = null) {

let i = 0, j = 0;

let hasCollision = false;

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

let s1 = set.getObjectAt(i).getRigidBody();

for (j = i + 1; j < set.size(); j++) {

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

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

}

}

return hasCollision;

}

// collide a given GameObject with a GameObjectSet

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

let j = 0;

let hasCollision = false;

let s1 = obj.getRigidBody();

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

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

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

}

return hasCollision;

}

// collide between all objects in two different GameObjectSets

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

let i = 0, j = 0;

let hasCollision = false;

for (i = 0; i < set1.size(); i++) {

let s1 = set1.getObjectAt(i).getRigidBody();

for (j = 0; j < set2.size(); j++) {

let s2 = set2.getObjectAt(j).getRigidBody();

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

}

}

return hasCollision;

}

1. Now, export all the defined functionality.

export {

// collide two shapes

collideShape,

// Collide

processSet, processObjToSet, processSetToSet

}

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Modifying the MyGame to Test Circle Collisions

The modifications required for testing the newly defined collision functionality is rather straightforward.

1. Edit my\_game\_main.js, in the constructor define the array for storing CollisionInfo and a new flag indicating if CollisionInfo should be drawn.

constructor() {

super();

… identical to previous code …

this.mCollisionInfos = [];

… identical to previous code …

// Draw controls

this.mDrawCollisionInfo = true; // for now, supports showing of collision info

… identical to previous code …

}

1. Modify the update() function to trigger the collision tests.

update() {

… identical to previous code …

if (this.mDrawCollisionInfo)

this.mCollisionInfos = [];

else

this.mCollisionInfos = null;

engine.physics.processObjToSet(this.mHero, this.mPlatforms, this.mCollisionInfos);

engine.physics.processSetToSet(this.mAllObjs, this.mPlatforms, this.mCollisionInfos);

engine.physics.processSet(this.mAllObjs, this.mCollisionInfos);

}

1. Modify the draw() function to draw the created CollisionInfo array when defined.

draw() {

… identical to previous code …

if (this.mCollisionInfos !== null) {

for (let i = 0; i < this.mCollisionInfos.length; i++)

this.mCollisionInfos[i].draw(this.mCamera);

this.mCollisionInfos = [];

}

… identical to previous code …

}

1. Remember to update the drawControlUpdate() function to support the C key for toggling of the drawing of the CollisionInfo objects.

drawControlUpdate() {

let i;

if (engine.input.isKeyClicked(engine.input.keys.C)) {

this.mDrawCollisionInfo = !this.mDrawCollisionInfo;

}

… identical to previous code …

}

### Observations

You can now run the project to examine your collision implementation between RigidCircle shapes in the form of the resulting CollisionInfo objects. Remember that you have only implemented circle-circle collisions. Now, use the left/right-arrow keys to select and work with a RigidCircle object. Use the WASD keys to move this object around to observe the magenta line segment representing the collision normal and depth when it overlaps with another RigidCircle. Try typing the U/Y keys to verify the correctness of CollisionInfo for shapes with different radii. Now, type the G key to create a few more RigidCircle objects. Try moving the selected object and increase its size such that it collides with multiple RigidCircle objects simultaneously and observe that a proper CollisionInfo is computed for every collision. Finally, note that you can toggle the drawing of CollisionInfo with the C key.

You have now implemented circle collision detection, built the required engine infrastructure to support collisions, and verified the correctness of the system. You are now ready to learn about Separating Axis Theorem (SAT), and implement a derived algorithm to detect collisions between rectangles.

## Separating Axis Theorem

The Separating Axis Theorem (SAT) is the foundation for one of the most popular algorithms used for detecting collision between general convex shapes in 2D. Since the derived algorithm can be computationally intensive, it is typically preceded with an initial pass of the broad phase method. The SAT states that:

Two convex polygons are not colliding if there exists a line (or axis) that is perpendicular to one of the given edges of the two polygons that when projecting all edges of the two polygons onto this axis results in no overlaps of the projected edges.

In other words, given two convex shapes in 2D space, iterate through all of the edges of the convex shapes, one at a time. For each of the edges, derive a line (or axis) that is perpendicular to the edge, project all the edges of the two convex shapes onto this line, and compute for the overlaps of the projected edges. If you can find one of the perpendicular lines where none of the projected edges overlaps, then the two convex shapes do not collide.

Figure 9-7 illustrates this description using two axes-aligned rectangles. In this case, there are two lines that are perpendicular to the edges of the two given shapes, the X and Y axes.



Figure 9-7. A Line Where Projected Edges Do Not Overlap

When projecting all of the edges of the shapes onto these two lines/axes, note that the projection results on the Y-axis overlaps, while there is no overlap on the X-axis. Since there exists one line that is perpendicular to one of the rectangle edges where the projected edges do not overlap, the SAT concludes that the two given rectangles do not collide.

The main strength of algorithms derived from the SAT is that for non-colliding shapes it has an early exit capability. As soon as an axis with no overlapping projected edge is detected, an algorithm can report no collision and does not need to continue with the testing for other axes. In the case of Figure 9-7, if the algorithm began with processing the X-axis, there would be no need to perform the computation for the Y-axis.

### A Simple SAT Based Algorithm

Algorithms derived based on the SAT typically consists of four steps. Note that this algorithm is applicable for detecting collisions between any convex shapes. For clarity, in the following explanation each step is accompanied with a simple example consisting of two rectangles.

* **Step 1 Compute Face Normals**: Compute the perpendicular axes, or face normals for projecting the edges. Using rectangles as an example, Figure 9-8 illustrates that there are four edges and each edge has a corresponding perpendicular axis. For example, A1 is the corresponding axis for and thus is perpendicular to the edge eA1. Note that in your RigidRectangle class, mFaceNormal, or face normals, are the perpendicular axes A1, A2, A3, and A4.



Figure 9-8. Rectangle Edges and Face Normals

* **Step 2 Project Vertices**: Project each of the vertices of the two convex shapes onto the face normals. For the given rectangle example, Figure 9-9 illustrates projecting all vertices onto the A1 axis from Figure 9-8.



Figure 9-9. Project Each Vertices onto Face Normals (shows A1)

* **Step 3 Identify Bounds**: Identifies the min and max bounds for the projected vertices of each convex shape. Continue with the rectangle example, Figure 9-10 shows the min and max positions for each of the two rectangles. Notice that the min/max positions are defined with respect to the direction of the given axis.



Figure 9-10. Identify the Min and Max Bound Positions for Each Rectangle

* **Step 4 Determine overlaps**: Determines if the two min/max bounds overlap. Figure 9-11 shows that the two projected bounds do indeed overlap. In this case, the algorithm cannot conclude and must proceed to process the next face normal. Notice that as illustrated in Figure 9-8, processing of face normal B1 or B3 will result in a deterministic conclusion of no collision.



Figure 9-11. Test for Overlaps of Projected Edges (shows A1)

The given algorithm is capable of determining if a collision has occurred with no additional information. Recall that after detecting a collision, the physics engine must also resolve potential interpenetration and derive a response for the colliding shapes. Both of these computations require additional information--the collision information as introduced in Figure 9-4. The next section introduces an efficient SAT-based algorithm that computes support points to both inform the true/false outcome of the collision detection and serve as the basis for deriving collision information.

### An Efficient SAT Algorithm: The Support Points

As illustrated in Figure 9-12, a support point for a face normal of shape-A is defined to be the vertex position on shape-B where the vertex has the most negative distant from the corresponding edge of shape-A. The vertex SA1 on shape-B has the largest negative distant from edge eA1 when measured along the A1 direction, and thus SA1 is the support point for face normal A1. The negative distance signifies that the measurement is directional and that a support point must be in the reversed direction from the face normal.



Figure 9-12. Support Points of Face Normals

In general, the support point for a given face normal may be different during every update cycle and thus must be recomputed during each collision invocation. In addition, and very importantly, it is entirely possible for a face normal to not have a defined support point.

#### Support Point May Not Exist for a Face Normal

A support point is defined only when the measured distance along the face normal has a negative value. For example, in Figure 9-12 the face normal B1 of shape-B does not have a corresponding support point on shape-A. This is because all vertices on shape-A are positive distances away from the corresponding edge eB1 when measured along B1. The positive distances signify that all vertices of shape-A are in frontof the edge eB1. In other words, the entire shape-A is in front of the edge eB1 of shape-B and thus the two shapes are not physically touching, and thus they are not colliding.

It follows that, when computing the collision between two shapes, if any of the face normals does not have a corresponding support point, then the two shapes are not colliding. Once again, the early exit capability is an important advantage--the algorithm can return a decision as soon as the first case of undefined support point is detected.

For convenience of discussion and implementation, the distance between a support point and the corresponding edge is referred to as the support point distance and this distance is computed as a positive number. In this way, the support point distance is actually measured along the negative face normal direction. This will be the convention followed in the rest of the discussions in this book.

#### The Axis of Least Penetration and Collision Information

When support points are defined for all face normals of a convex shape, the face normal of the smallest support point distance is the axis leading to the least interpenetration. Figure 9-13 shows the collision between two shapes where supports points for all of the face normals of shape-B are defined: vertex SB1 on shape-A is the corresponding support point for face normal B1, SB2 for face normal B2, and so on. In this case, SB1 has the smallest corresponding support point distance and thus the face normal B1 is the axis that leads to the least interpenetration. The illustration on the right on Figure 9-13 shows that in this case, support point distance is the collision depth, face normal B1 is collision normal, support point SB1 is the start of the collision, and the end of the collision can be readily computed, it is simply SB1 offset by collision depth in the collision normal direction.



Figure 9-13. Axis of Least Penetration and The Corresponding Collision Information

#### The Algorithm

With the background description, the efficient SAT-based algorithm to compute the collision between two convex shapes, A and B, can be summarized as:

Compute the support points for all the face normals on shape-A

If any of the support points is not defined, there is no collision

If all support points are defined, compute the axis of least penetration

Compute the support points for all the face normals on shape-B

If any of the support points is not defined, there is no collision

If all support points are defined, compute the axis of least penetration

The collision information is simply the smaller collision depth from the above two results. You are now ready to implement the support point SAT algorithm.

## The Rectangle Collisions Project

This project will guide you through the implementation of the support point SAT algorithm. You can see an example of this project running in Figure 9-14. The source code to this project is defined in chapter9/9.3.rectangle\_collisions.

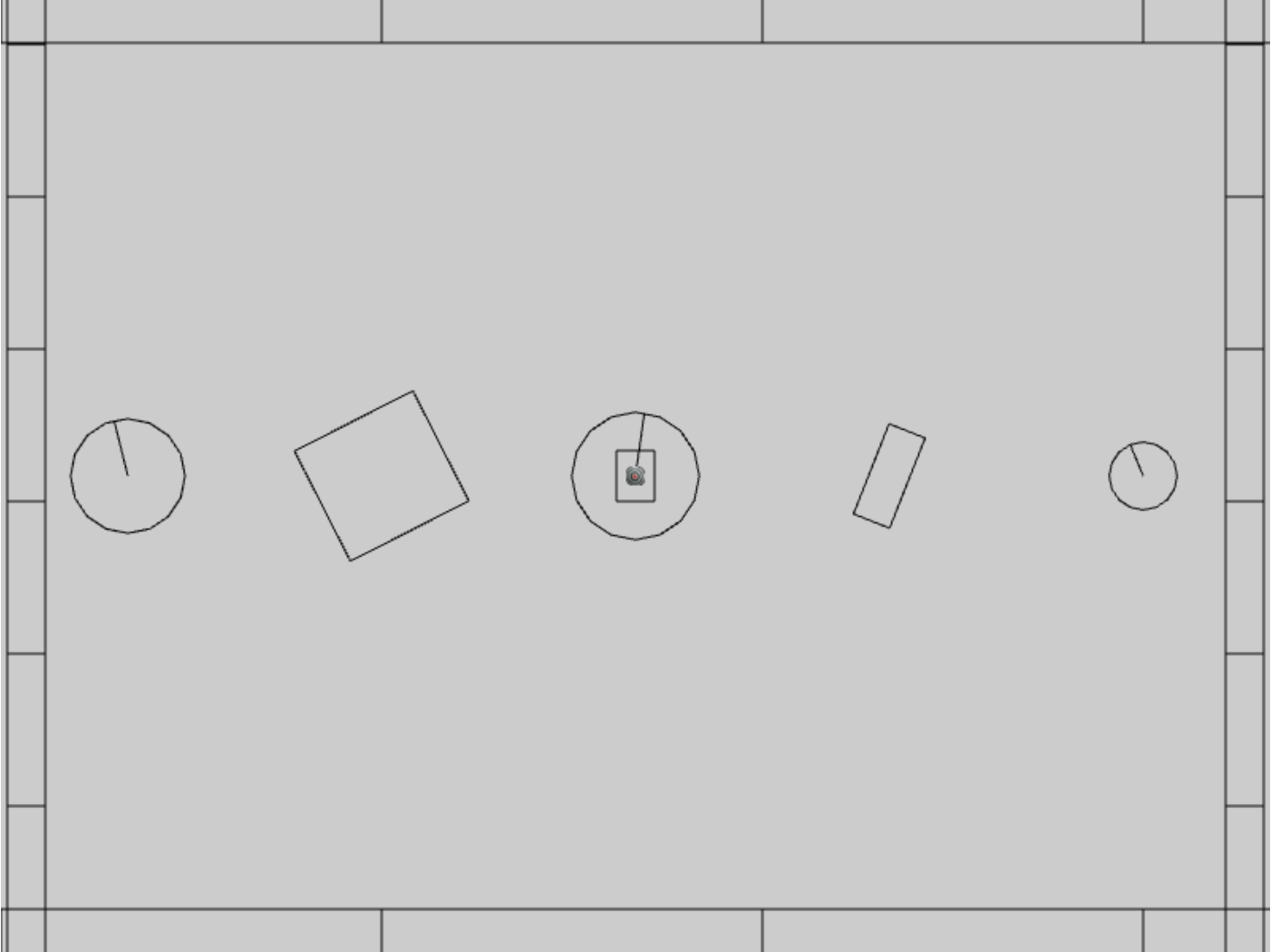


Figure 9-14. Running the Rectangle Collisions project

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

* **Behavior control:**

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

The goals of the project are as follows:

* To gain insights into and implement the support point SAT algorithm
* To continue with completing narrow phase collision detection implementation.

After this project your game engine will able to collide between circle shapes and between rectangles shapes while still not supporting collisions between circle and rectangle shapes. This will be one step closer to completing the implementation of narrow phase collision detection for rigid shapes. The remaining functionality, detecting circle-rectangle collisions, will be covered in the next subsection.

### Implementing the Support Point SAT

With the collision detection infrastructure from the previous project completed, the only modification required is to append the new functionality to the RigidRectangle class. Recall that the source code file rigid\_rectangle\_collision.js was created for the implementation of rectangle collision.

1. In the src/engine/rigid\_shapes folder, edit rigid\_rectangle\_collision.js to define local variables. These are temporary storage during computations, they are statically allocated and reused to avoid the cost of repeated dynamic allocation during each invocation.

class SupportStruct {

constructor() {

this.mSupportPoint = null;

this.mSupportPointDist = 0;

}

}

// temp work area to save memory allocations

let mTmpSupport = new SupportStruct();

let mCollisionInfoR1 = new CollisionInfo();

let mCollisionInfoR2 = new CollisionInfo();

1. Create a new function findSupportPoint() to compute a support point based on, dir, the negated face normal direction, ptOnEdge, a position on the given edge (e.g., a vertex). The listed code marches through all the vertices; compute vToEdge, the vector from vertices to ptOnEdge; project this vector onto the input dir; and record the largest positive projected distant. Recall that dir is the negated face normal direction, and thus the largest positive distant corresponds to the furthest vertex position. Note that it is entirely possible for all of the projected distances to be negative. In such cases, all vertices are in front of the input dir, a support point does not exist for the given edge, and thus the two rectangles do not collide.

RigidRectangle.prototype.findSupportPoint = function (dir, ptOnEdge) {

// the longest project length

let vToEdge = [0, 0];

let projection;

mTmpSupport.mSupportPointDist = -Number.MAX\_VALUE;

mTmpSupport.mSupportPoint = null;

// check each vector of other object

for (let i = 0; i < this.mVertex.length; i++) {

vec2.subtract(vToEdge, this.mVertex[i], ptOnEdge);

projection = vec2.dot(vToEdge, dir);

// find the longest distance with certain edge

// dir is -n direction, so the distance should be positive

if ((projection > 0) && (projection > mTmpSupport.mSupportPointDist)) {

mTmpSupport.mSupportPoint = this.mVertex[i];

mTmpSupport.mSupportPointDist = projection;

}

}

}

1. With the ability to locate a support point for any face normal, the next step is the find the axis of least penetration with the findAxisLeastPenetration() function. Recall that the axis of least penetration is the support point with the least support point distant. The listed code loops over the four face normals, finds the corresponding support point and support point distance, and records the shortest distance. The while-loop signifies that if a support point is not defined for any of the face normals then the two rectangles do not collide.

RigidRectangle.prototype.findAxisLeastPenetration = function (otherRect, collisionInfo) {

let n;

let supportPoint;

let bestDistance = Number.MAX\_VALUE;

let bestIndex = null;

let hasSupport = true;

let i = 0;

let dir = [0, 0];

while ((hasSupport) && (i < this.mFaceNormal.length)) {

// Retrieve a face normal from A

n = this.mFaceNormal[i];

// use -n as direction and the vertex on edge i as point on edge

vec2.scale(dir, n, -1);

let ptOnEdge = this.mVertex[i];

// find the support on B

// the point has longest distance with edge i

otherRect.findSupportPoint(dir, ptOnEdge);

hasSupport = (mTmpSupport.mSupportPoint !== null);

// get the shortest support point depth

if ((hasSupport) && (mTmpSupport.mSupportPointDist < bestDistance)) {

bestDistance = mTmpSupport.mSupportPointDist;

bestIndex = i;

supportPoint = mTmpSupport.mSupportPoint;

}

i = i + 1;

}

if (hasSupport) {

// all four directions have support point

let bestVec = [0, 0];

vec2.scale(bestVec, this.mFaceNormal[bestIndex], bestDistance);

let atPos = [0, 0];

vec2.add(atPos, supportPoint, bestVec);

collisionInfo.setInfo(bestDistance, this.mFaceNormal[bestIndex], atPos);

}

return hasSupport;

}

1. You can now implement the collideRectRect() function by computing the axis of least penetration with respect to each of the two rectangles and choosing the smaller of the two results.

Rectangle.prototype.collideRectRect = function (r1, r2, collisionInfo) {

var status1 = false;

var status2 = false;

// find Axis of Separation for both rectangle

status1 = r1.findAxisLeastPenetration(r2, collisionInfoR1);

if (status1) {

status2 = r2.findAxisLeastPenetration(r1, collisionInfoR2);

if (status2) {

// if both of rectangles are overlapping, choose the shorter normal as the normal

if (collisionInfoR1.getDepth() < collisionInfoR2.getDepth()) {

var depthVec = collisionInfoR1.getNormal().scale(collisionInfoR1.getDepth());

collisionInfo.setInfo(collisionInfoR1.getDepth(),

collisionInfoR1.getNormal(),

collisionInfoR1.mStart.subtract(depthVec));

} else {

collisionInfo.setInfo(collisionInfoR2.getDepth(),

collisionInfoR2.getNormal().scale(-1),

collisionInfoR2.mStart);

}

}

}

return status1 && status2;

}

1. Complete the implementation by modifying the collisionTest() function to call the newly defined collideRectRect() function to compute the collision between two rectangles.

RigidRectangle.prototype.collisionTest = function (otherShape, collisionInfo) {

let status = false;

if (otherShape.mType === "RigidCircle") {

status = false;

} else {

status = this.collideRectRect(this, otherShape, collisionInfo);

}

return status;

}

### Observations

You can now run the project to test your implementation. You can use the left/right-arrow keys to select any rigid shape and use the WASD keys to move the selected object. Once again you can observe the magenta collision information between overlapping rectangles, or overlapping circles. Remember that this line shows the least amount of positional correction needed to ensure that there is no overlap between the shapes. Type the X/Z keys to rotate and the U/Y keys to change the size of the selected object and observe how the collision information changes accordingly.

At this point, only circle-circle and rectangle-rectangle collisions are supported so when circles and rectangles overlap, there are no collision information shown. This will be resolved in the next project.

## Collision Between Rectangles and Circles

The support point algorithm does not work with circles because a circle does not have identifiable vertex positions. Instead, you will implement an algorithm that detects collisions between a rectangle and a circle according to the relative position of the circle’s center with respect to the rectangle.

Before discussing the actual algorithm, as illustrated in Figure 9-15, it is convenient to recognize that the area outside an edge of a rectangle can be categorized into three distinct regions by extending the connecting edges. In this case, the dotted lines separated the area outside the given edge into: RG1, the region to the left/top; RG2, the region to the right/bottom; and RG3, the region immediately outside of the given edge.

With this background, the collision between a rectangle and a circle can be detected as follows:

* **Step A**: Compute the edge on the rectangle that is closest to the circle center.
* **Step B**: If the circle center is inside the rectangle: collision is detected.
* **Step C**: If circle center is outside

**Step C1**: If in Region RG1: distance between the circle center and left/top vertex determines if a collision has occurred.

**Step C2**: If in Region RG2: distance between the circle center and right/bottom vertex determines if a collision has occurred.

**Step C3**: If in Region RG3: perpendicular distance between the center and the Edge determines if a collision has occurred.

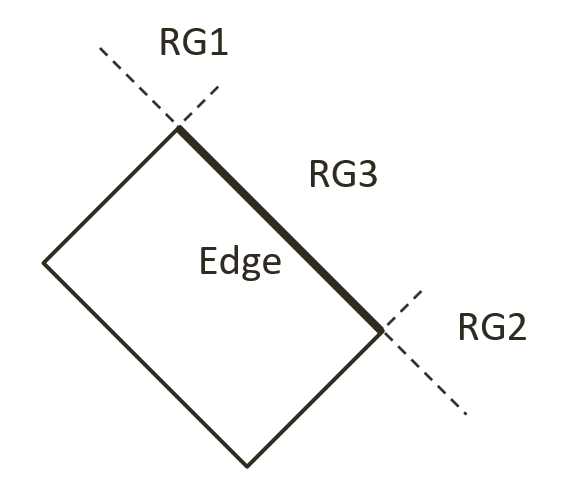


Figure 9-15. The Three Regions Outside a Given Edge of a Rectangle

## The Rectangle and Circle Collisions Project

This project guides you in implementing the described rectangle-circle collision detection algorithm. You can see an example of this project running in Figure 9-16. The source code to this project is defined in chapter9/9.4.rectangle\_and\_circle\_collisions.



Figure 9-16. Running the Rectangle and Circle Collisions project

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

* **Behavior control:**

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

The goals of the project are as follows:

* To understand and implement the rectangle circle collision detection algorithm.
* To complete the narrow phase collision detection implementation for circle and rectangle shapes.

### Defining Rectangle and Circle Collision

Once again, with the completed collision detection infrastructure the only modification required is to append the new functionality. This will be implemented in the RigidRectangle class. For readability of the rather involved algorithm, a new source code file, rigid\_rectangle\_circle\_collision.js, will be created for implementation.

1. Update the RigidRectangle access file to import from the latest source code file. In the src/engine/rigid\_shapes folder, edit rigid\_rectangle.js to replace the import to be from the latest source code file.

import RigidRectangle from "./rigid\_rectangle\_circle\_collision.js";

export default RigidRectangle;

1. In the same folder, create the rigid\_rectangle\_circle\_collision.js file to import from rigid\_rectangle\_collision.js such that new collision function can be appended to the class.

import RigidRectangle from "./rigid\_rectangle\_collision.js";

1. Define a new function, checkCircRecVertex() to process regions RG1 and RG2. As illustrated in the left diagram of Figure 9-17, the parameter v1 is the vector from vertex position to circle center. The right diagram of Figure 9-17 shows that a collision occurs when dist, the length of v1, is less than r, the radius. In this case, the collision depth is simply the difference between r and dist.

RigidRectangle.prototype.checkCircRecVertex = function(v1, cirCenter, r, info) {

// the center of circle is in corner region of mVertex[nearestEdge]

let dist = vec2.length(v1);

// compare the distance with radius to decide collision

if (dist > r)

return false;

let radiusVec = [0, 0];

let ptAtCirc = [0, 0];

vec2.scale(v1, v1, 1/dist); // normalize

vec2.scale(radiusVec, v1, -r);

vec2.add(ptAtCirc, cirCenter, radiusVec);

info.setInfo(r - dist, v1, ptAtCirc);

return true;

}



Figure 9-17. Left: Condition when center is in region RG1. Right: The corresponding collision information

1. Define collideRectCirc() function to detect the collision between a rectangle and a circle. The following code shows the declaration of local variables and the five major steps, A to C3, that must be performed. The details of each steps are discussed in the rest of this subsection.

gle.prototype.collideRectCirc = function (otherCir, collisionInfo) {

let outside = false;

let bestDistance = -Number.MAX\_VALUE;

let nearestEdge = 0;

let vToC = [0, 0];

let projection = 0;

let i = 0;

let cirCenter = otherCir.getCenter();

… implementation of Step A: Compute nearest edge, handle if center is inside …

if (!outside) {

… implementation of Step B: Circle center is insde rectangle …

return;

}

… implementation of Steps C1 to C3: Circle center is outside rectangle …

return true;

};

1. Step A, compute the nearest edge. The nearest edge can be found by computing the perpendicular distances between the circle center and each edge of the rectangle. This distance is simply the projection of the vector, from each vertex to the circle center, onto the corresponding face normal. The listed code iterates through all of the vertices computing the vector from the vertex to the circle center, and projects the computed vector to the corresponding face normal.

// Step A: Compute the nearest edge

while ((!outside) && (i<4)) {

// find the nearest face for center of circle

vec2.subtract(vToC, cirCenter, this.mVertex[i]);

projection = vec2.dot(vToC, this.mFaceNormal[i]);

if (projection > bestDistance) {

outside = (projection > 0); // if projection < 0, inside

bestDistance = projection;

nearestEdge = i;

}

i++;

}

As illustrated in the left diagram of Figure 9-18, when the circle center is inside the rectangle all vertex to center vectors will be in the opposite directions of their corresponding face normals and thus will result in negative projected lengths. This is in contrast to the right diagram of Figure 9-18, when the center is outside of the rectangle. In this case at least one of the projected lengths will be positive. For this reason, the “nearest projected distance” is the one with the least negative value and thus is actually the largest number.



Figure 9-18. Left: Center inside the rectangle will result in all negative projected length. Right: Center outside the rectangle will result in at least one positive projected length

1. Step B, if the circle center is inside the rectangle, then collision is detected and the corresponding collision information can be computed and returned.

if (!outside) { // inside

// Step B: The center of circle is inside of rectangle

vec2.scale(radiusVec, this.mFaceNormal[nearestEdge], otherCir.mRadius);

dist = otherCir.mRadius - bestDistance; // bestDist is -ve

vec2.subtract(ptAtCirc, cirCenter, radiusVec);

collisionInfo.setInfo(dist, this.mFaceNormal[nearestEdge], ptAtCirc);

return true;

}

1. Step C1, determine and process if the circle center is in Region RG1. As illustrated in the left diagram of Figure 9-17, Region RG1 can be detected when v1, the vector between the center and vertex is in the opposite direction of v2, the direction of the edge. This condition is computed in the following listed code.

let v1 = [0, 0], v2 = [0, 0];

vec2.subtract(v1, cirCenter, this.mVertex[nearestEdge]);

vec2.subtract(v2, this.mVertex[(nearestEdge + 1) % 4], this.mVertex[nearestEdge]);

let dot = vec2.dot(v1, v2);

if (dot < 0) {

// Step C1: In Region RG1

return this.checkCircRecVertex(v1, cirCenter, otherCir.mRadius, collisionInfo);

} else {

… implementation of Steps C2 and C3 to follow …

}

1. Steps C2 and C3, differentiate and process for Regions RG2 and RG3. The listed code performs complementary computation for the other vertex on the same rectangle edge for Region RG2. The last region for the circle center to be located in would be the area immediately outside the nearest edge. In this case, the bestDistance computed previously in step A is the distance between the circle center and the given edge. If this distance is less than the circle radius then a collision has occurred.

if (dot < 0) {

// Step C1: In Region RG1

… identical to previous code …

} else {

// Either in Region RG2 or RG3

// v1 is from right vertex of face to center of circle

// v2 is from right vertex of face to left vertex of face

vec2.subtract(v1, cirCenter, this.mVertex[(nearestEdge + 1) % 4]);

vec2.scale(v2, v2, -1);

dot = vec2.dot(v1, v2);

if (dot < 0) {

// Step C2: In Region RG2

return this.checkCircRecVertex(v1, cirCenter, otherCir.mRadius, collisionInfo);

} else {

// Step C3: In Region RG3

if (bestDistance < otherCir.mRadius) {

vec2.scale(radiusVec, this.mFaceNormal[nearestEdge], otherCir.mRadius);

dist = otherCir.mRadius - bestDistance;

vec2.subtract(ptAtCirc, cirCenter, radiusVec);

collisionInfo.setInfo(dist, this.mFaceNormal[nearestEdge], ptAtCirc);

return true;

} else {

return false;

}

}

}

#### Calling the Newly Defined Function

The last step is to invoke the newly defined function. Note that the collision function should be called when a circle comes into contact with a rectangle, as well as when a rectangle comes into contact with a circle. For this reason, you must modify both the RigidRectangle class in rigid\_rectangle\_collision.js, and the RigidCircle class in rigid\_circle\_collision.js.

1. In the src/engine/rigid\_shapes folder, edit rigid\_rectangle\_collision.js, modify the collisionTest() function to call the newly defined collideRectCirc() when the parameter is a circle shape.

RigidRectangle.prototype.collisionTest = function (otherShape, collisionInfo) {

let status = false;

if (otherShape.mType === "RigidCircle") {

status = this.collideRectCirc(otherShape, collisionInfo);

} else {

status = this.collideRectRect(this, otherShape, collisionInfo);

}

return status;

}

1. In the same folder, edit rigid\_circle\_collision.js, modify the collisionTest() function to call the newly defined collideRectCirc() when the parameter is a rectangle shape.

RigidCircle.prototype.collisionTest = function (otherShape, collisionInfo) {

let status = false;

if (otherShape.mType === "RigidCircle") {

status = this.collideCircCirc(this, otherShape, collisionInfo);

} else {

status = otherShape.collideRectCirc(this, collisionInfo);

}

return status;

}

### Observations

You can now run the project to test your implementation. You can create new rectangles and circles, move and rotate them to observe the corresponding collision information.

You have finally completed the narrow phase collision detection implementation and can begin to examine the motions of these rigid shapes.

# 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 added a displacementto the position of an object. 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 approximations 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. With Symplectic Euler Integration, instead of current results, intermediate results are used in subsequent approximations and thus better simulates the actual movement. 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 rest of this section, you will first define relevant physical attributes to complete the RigidShape implementation. After that, 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 specific to each shape’s geometry such that the implementation of updateIntertia() is a responsibility of the subclass.

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. These functions are straightforward and are not listed here.
2. For the convenience of debugging, define a function, getCurrentState(), to retrieve variable values as text, and a function, userSetsState(), to allow interactive manipulations of 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 Class

As mentioned, the rotational inertia, mInertial, is 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 Class

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 the 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 implement the Symplectic Euler Integration movement approximation in 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 of the 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 interpenetration occurs frequently as objects travel through the scene. 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 sized) 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 laws 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 should 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 resolution 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 remains in continuous 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. 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 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 collision detection is performed only when at least one of the objects is 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 when 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,positional correction, 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, positional correction is off, and showing of collision information is on. For these reasons, you will observe the created rigid shapes clumping in the central region of the game scene with many associated magenta colored collision information lines.

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 colored collision depth line 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 one update the object will move past 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(), these 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. After the mechanics behind simple impulse resolution are fully understood and implemented, the complications associated with angular impulse resolutions will be examined in the next section.

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 from 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: 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 to 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) for 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: computes 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: uses 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 only linear velocities are considered and 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 F/N 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 bounciness 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 solution 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 the 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 .

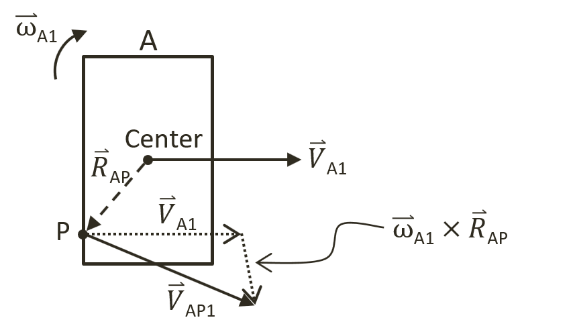


Figure 9-28 Linear Velocity at a Position in the Presence of Rotation

**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 . By now you know that the linear velocities at point before the collision for the two objects are as follows,

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

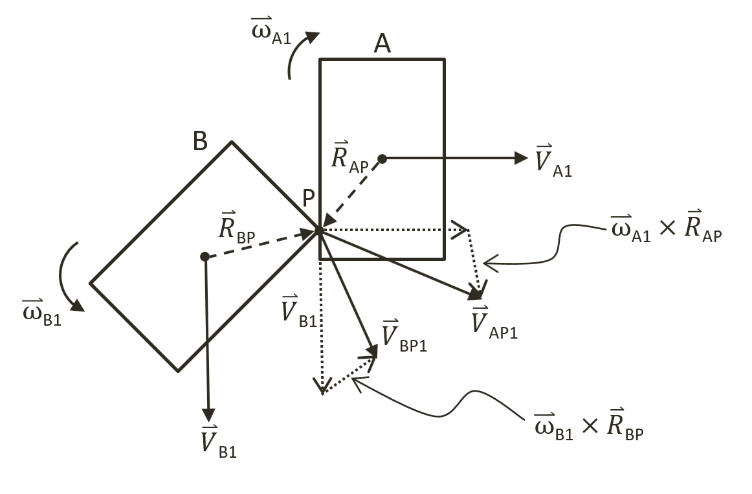


Figure 9-29 Colliding Shapes with Angular Velocities

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 and begin with Equation (21) for object A.

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), on the right-hand-side, the term with the dot-product 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.

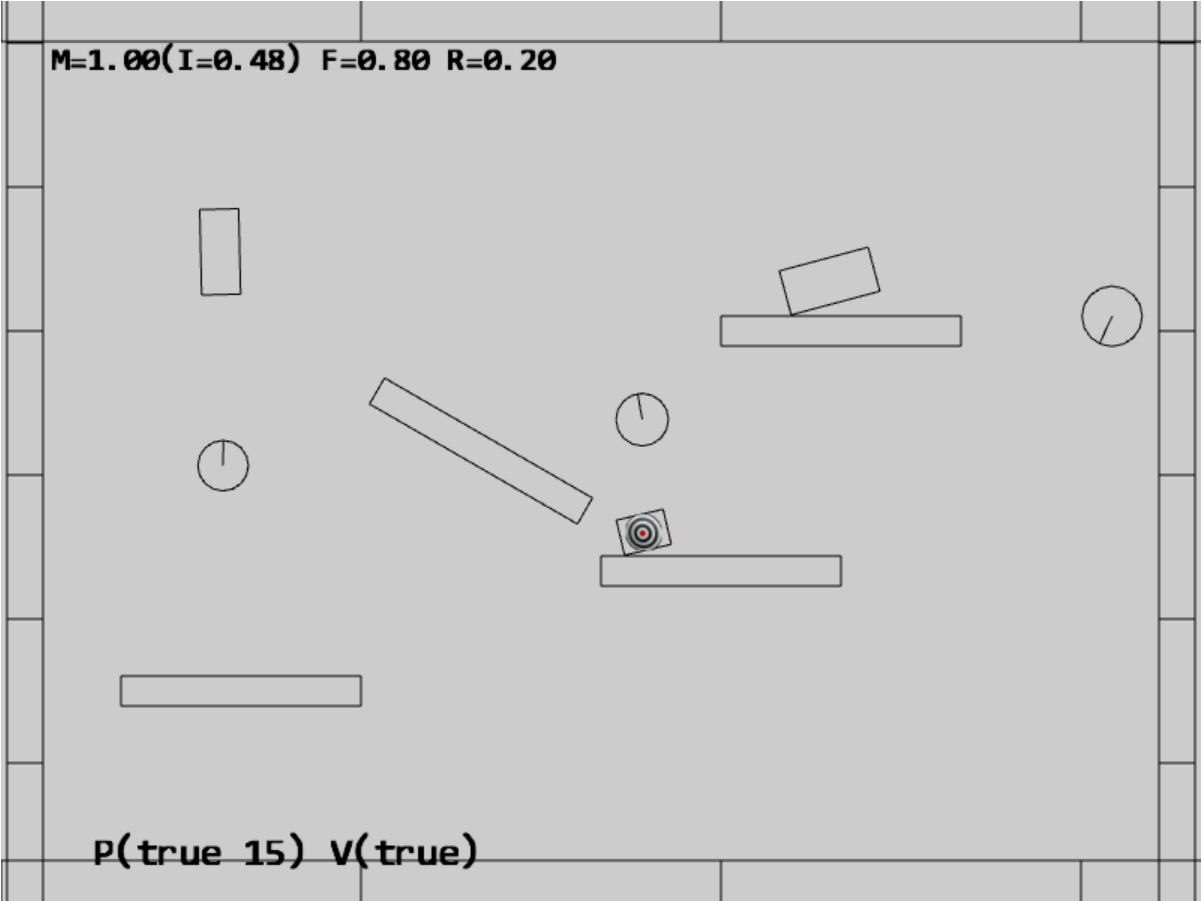


Figure 9-30. Running the Collision Angular Resolution project

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**: Inject random velocity to all objects

G key: Randomly create a new rigid circle or rectangle

* **Draw control**

C key: Toggle the drawing of all CollisionInfo

T key: Toggle textures on all objects

R key: Toggle the drawing of RigidShape

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

* **Object control:**

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

WASD keys: Move the selected object

Z/X key: Rotate the selected object

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

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

The goals of the project are as follows:

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

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

### Updating the Physics Component

To properly integrate angular impulse, you 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 (9), 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 the normal direction. A positive normal direction component signifies that 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 the 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 to be 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 in 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 inject random velocities to all objects and observe the collisions. Notice that the collisions with the high-mass circle are almost like collisions with stationary walls/platforms. The inversed mass and rotational inertia modelled by the Impulse Method is capable of successfully capturing the collision effects of objects with different masses.

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

# Summary

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

A step by step derivation of the relevant formulae for the simulations is followed by a detailed guide to the building of a functioning system. You have learned to extract collision information between shapes, formulate and compute shape collisions including 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 interface functions, 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 related to behaviors 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:

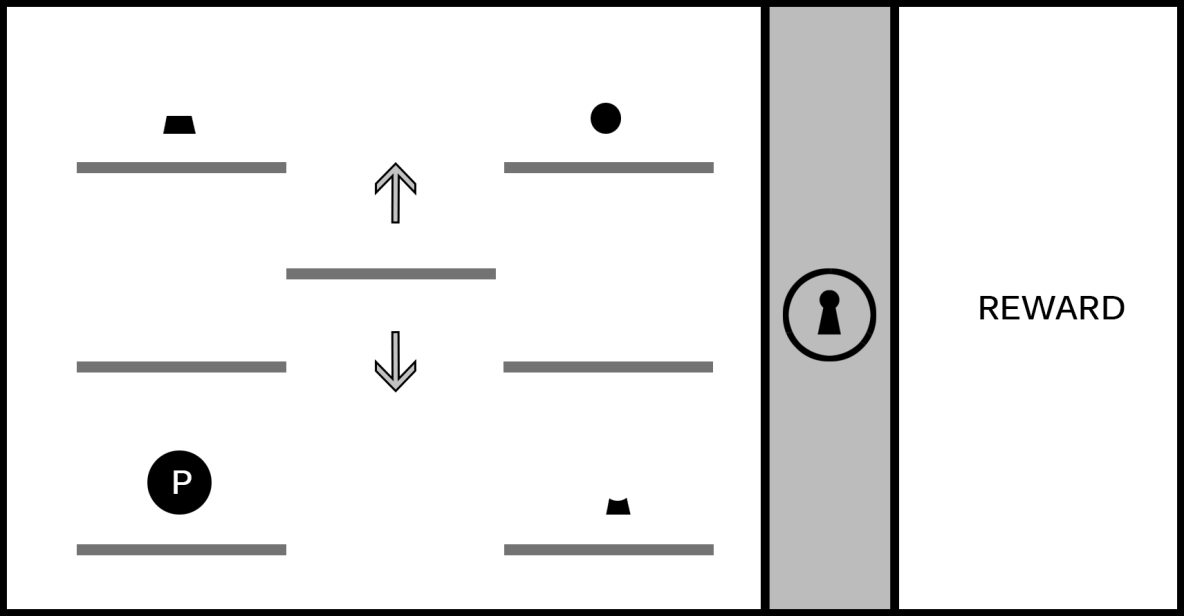


Figure 9-33. The game screen now shows just one instance of each part of the lock (top, middle, bottom), and the hero character moves in the manner of a traditional jumping 2D platformer. The six platforms on the left and right are stationary, and the middle platform moves up and down, allowing the player 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:

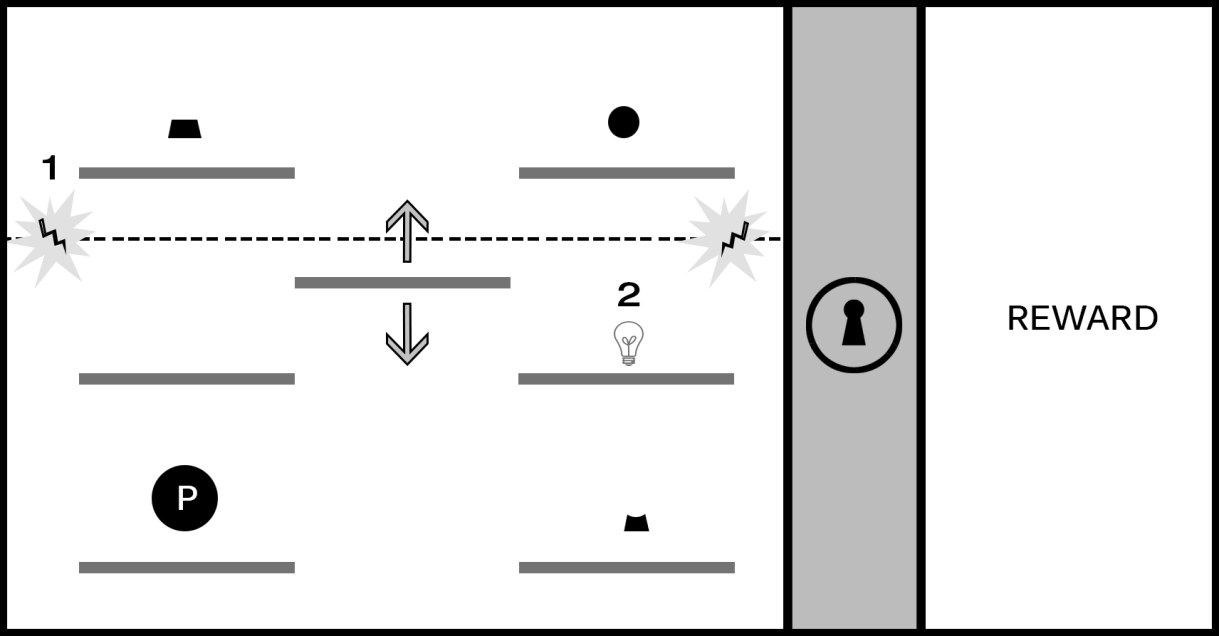


Figure 9-34. The introduction of a force field blocking access to the upper platforms (#1) can significantly increase the challenge of the platformer component. In this design, the player must activate the switch (represented with a lightbulb in #2) to disable the force field and reach the first and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