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 game engines the functionality of simulating energy transfer is often referred to as 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, or vehicle physics. A believable physical behavior and interactions of game objects have become key elements of many modern PC and console games, and more recently, browser and smartphone games. For example, the bouncing of a ball, the wiggling of a jelly block, the ripples on a lake, or the skidding of a car. The proper simulation and realistic renditions of these 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 speed, tire properties, etc.; the ripples on a lake depends on the cause, size of the lake, etc.; wiggle of a jelly block depends on density, initial deformation, etc. Even in the very simple case, 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 jelly block (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 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 followings:

* 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 steps that together implements 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 of this chapter, the Rigid Shape and Bounds project
* Collisions of the rigid shapes: Examines and implements the mathematics to accurately collide circle and rectangle RigidShape objects. An important concept to be learned is that in the digital world, rigid shapes can overlap and it is essential to capture the details of this overlap in a CollisionInfo object. The topics on collision will be covered by three projects focusing separately on collisions between circle shapes: the Circle Collision and Collision Info project; collisions between rectangle shapes: the Rectangle Collision project; and lastly, collisions between circle and rectangle shapes: the Rectangle and Circle Collisions project.
* Simulate physical motion: Approximates integrals that describe motions in the discretized world. The topic on motion will be covered by the Rigid Shape Movements project.
* Collision positional correction: Observes that interpenetration between colliding rigid shapes often occurs and implements the numerically stable relaxation loops 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, derives approximation, and implements the solution. 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, attaching rigid circles on spaceships and performing rigid body physics simulation of 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 contacts. 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 usually bounded by a simple geometry, e.g., a circle, where the potentially expensive collision computation is only invoked when the bounds of 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 and 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 and the particle.png image, you can find the following external resource files in the assets folder:

* minion\_sprite.png 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 GameObjct, 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 to these functions are not relevant to the physics simulation and are 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() function.

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

Since Renderable can be rotated freely, the rigid shapes that represent these Renderable objects must also be rotated freely. In the case of Renderable objects, the actual rotation is accomplished via vertex multiplication with appropriate transformation matrix in the WebGL vertex shader. For rigid shapes, this rotation must be computed explicitly.

Now, 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.

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;

}

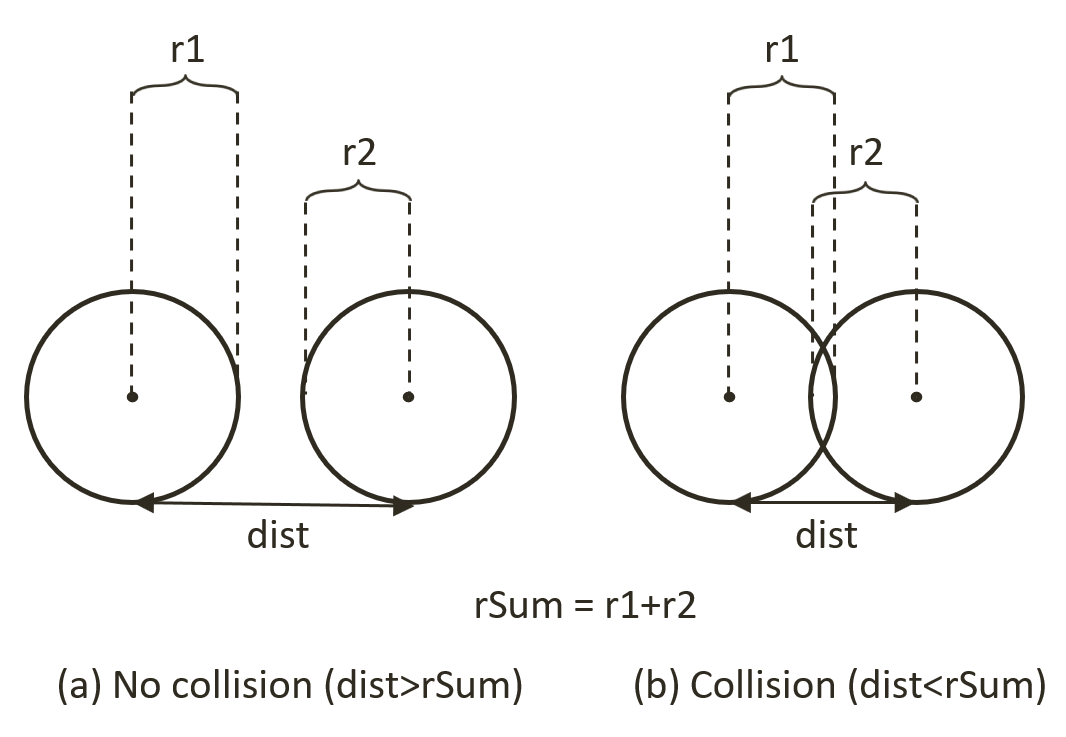


Figure 9-2. Circle Collision Detection: (a) No collision (b) Collision detected.

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

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 position and rotation of the rigid rectangle is defined by the Transform referenced by mXform, where 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() functions. 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 vertices and face normals must be re-computed (rotateVertices() calls computeFaceNormals()).

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 normal 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 has 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, including 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. Replace 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, circles and rectangles, with combinations of Renderable types including, TextureRenderable, SpriteRenderable, and SpriteAnimateRenderable. The proper functioning of these combinations can demonstrate the correctness of RigidShape implementation and allow visual examination of the appropriateness and 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 where the supported 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 Z/X rotation 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 will encompass different combinations of Renderable and RigidShape types, and at the same time, allow visual examination of approximating complex texture shapes with simple geometric circles and rectangles.

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 main functionality of constructor, init(), draw(), update(), etc. of MyGame is 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 actual more accurate and costlier 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 the Hero in the center. The Z/X and Y/U 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 and the corresponding RigidShape, while the Y/U keys only changes the dimension of the RigidShape. This allows the designer to control how tightly to wrap the Renderable with the corresponding RigidShape. Try typing 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. Now, 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 contacts 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 computation 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 addition to collision checks between every object a detailed collision detection algorithm involves intensive computations. 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 and rule out those that are physically far apart from each other that clearly cannot possibly collide. This allows the detailed and computationally intensive algorithm, or the narrow phase method, to be deployed for objects that are physically closed to each other.

A popular broad phase method uses axis-aligned bounding boxes (AABB) or bounding circles to approximate proximity of objects. As detailed in Chapter 6, AABBs are excellent for approximating objects that are aligned with the major axes, but, have 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 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 work with collision conditions in CollisionInfo classes
* To understand and implement circle collision detection algorithm

### Defining the CollisionInfo Class

A new class must be defined to record RigidShape collision interpenetration situation 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 …

}

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 update to support collisions. Since the abstract base shape, RigidShape, does not contain actual geometric information, the actual collision functions only need to 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 a RigidRectangle. If so, for now, call the collideCirCirc() function in the case of a RigidCircle as currently 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 normalFrom2to1 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

With the circle-to-circle collision detection implemented, you can now define the physics component to trigger the collision computation.

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 GameObjectSet objects.

// 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);

let p = obj.getXform().getPosition();

this.mTarget.getXform().setPosition(p[0], p[1]);

}

1. Modify the draw() function to draw the created CoolisionInfo 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 and as such remember to use the left/right-arrow keys to select 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 Y/U 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 is in collision 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 the 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 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 exist 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 B2 or B4 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.

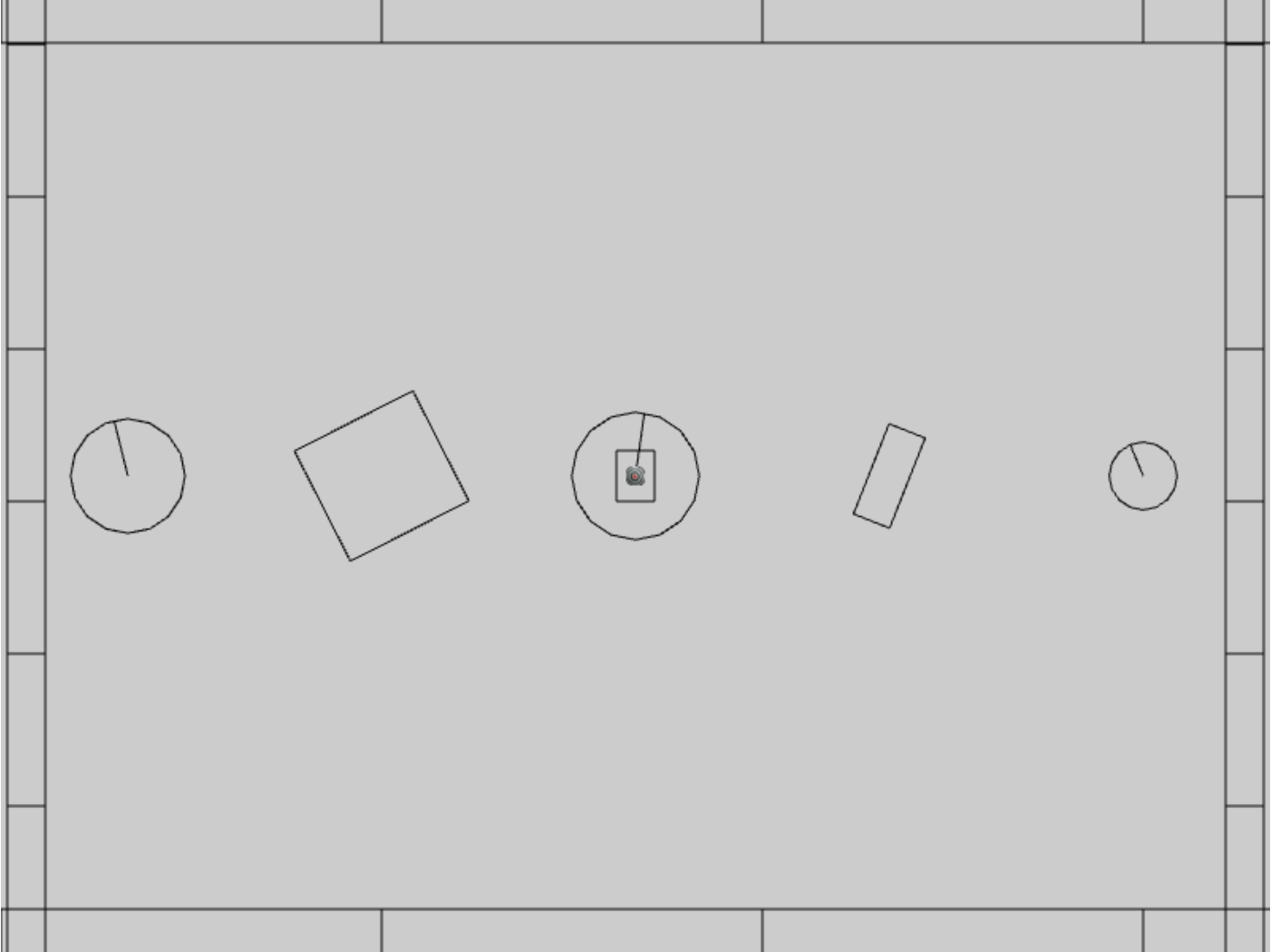


Figure 9-14. Running the Rectangle Collisions project

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 required for 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. Nota 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 collidedRectRect() 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.collidedRectRect = 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 collidedRectRect() 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 there is no overlap between the shapes. Type the Z/X keys to rotate and the Y/U 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 is will be resolved in the next project.

## Collision Between Rectangles and Circles

The support point approach to computing collision detection 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 from the Edge determines if collision has occurred.

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

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

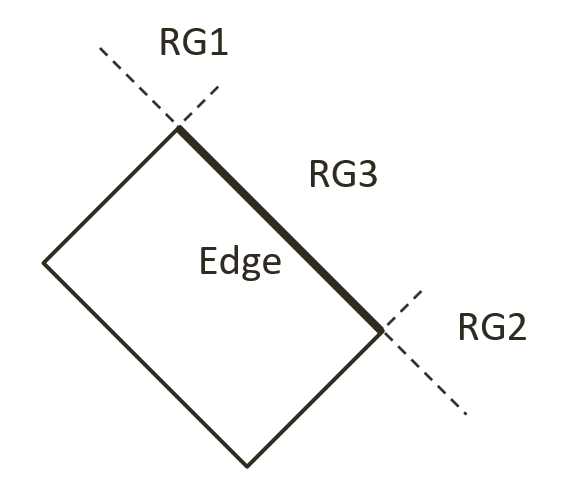


Figure 9-15. The Three Regions Outside a Given Edge of a Rectangle

## 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 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. Now, 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;

}

The right diagram of Figure 9-17 shows that collision occurs when the length of vector v1 is less than the circle radius, and in this case, the collision normal is simply along the vector v1, and collision depth is the difference between the radius and dist, the length of vector v1.



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 listing 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 the nearest edge and handle if center is inside

if (!outside) {

… Implementation of Step B: The circle center is insde the rectangle

return;

}

… Implementation of Steps C1 to C3: Circle center is outside

return true;

};

1. Step A, compute the nearest edge. The nearest edge can be found by computing the perpendicular distances between the circle center to each of the edges 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 shows marching through all of the vertices, computing the vector from the vertex to the circle center, and projecting 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 normal and thus will result in negative projected length. This is in contrast to the right diagram of Figure 9-18, when the center is outside of the rectangle then, 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.