Simulating the Rigid World

# 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 with B, C, D, and E. With A and B results available, next you must perform three collision detections between the second object, B with objects C, D, and E; followed by two collisions for the third object C, then, one for object D. In this way, without dedicated optimizations, you must perform operations to detect the collisions between objects.

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: the detailed and computationally intensive algorithm, or the narrow phase method, are only 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 uniform grid or 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 the 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 can be resolved by moving the colliding objects along the collision normal by the collision depth 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 from 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 otherShape is a RigidRectangle and call the collideCirCirc() function in the case of a RigidCircle. 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 (Case A), collision with centers of the two circles located at different positions (Case B), and the two centers located at exactly the same position (Case C). The following code shows Case A, the detection of no collision. Case A is very similar to the case illustrated in Figure 9-2.

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

let vFrom1to2 = [0, 0];

// Case A: 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;

}

// Cases B and C to follow

1. When a collision is detected, if the two circle centers are located at different positions (Case B), 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.

//… continue from the previous step

if (dist !== 0) {

// Case B: 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);

}

//… details in the next step

1. The last case for two colliding circles is when both circle centers are located at exactly the same position (Case C). 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.

//...continue from the previous step

if (dist !== 0) {

//...identical to previous step

} else {

let n = [0, -1];

// Case C: 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 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 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 implemented circle collision detection, build 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 implementing the algorithm to detect collisions between rectangles.