## 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: R1, the region to the left/top; R2, the region to the right/bottom; and R3, 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**: Edge = 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 R1: distance between the circle center and left/top vertex from the Edge determines if collision has occurred.

**Step C2**: If in Region R2: distance between the circle center and right/bottom vertex from the Edge determines if collision has occurred.

**Step C3**: If in Region R3: perpendicular distance between the center and the Edge determines if collision has occurred.

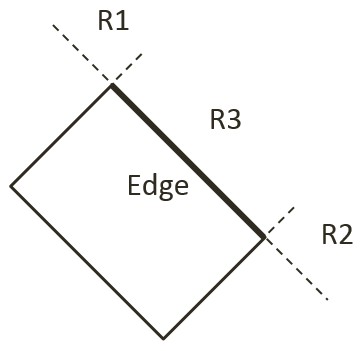


Figure 9-15. The Three Regions Outside a Given Edge of a Rectangle

## 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 R1 and R2. 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 R1. 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, Step A to Step 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();

// Step A: Compute the nearest edge and handle if center is inside

if (!outside) {

// Step B: The circle center is insde the rectangle

return;

}

// Circle center is outside

// Steps C1 to C3

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 R1. As illustrated in the left diagram of Figure 9-17, Region R1 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 R1

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

} else {

… Steps C2 and C3

}

1. Steps C2 and C3, differentiate and process for Regions R2 and R3. The listed code performs complementary computation for the other vertex on the same rectangle edge for Region R2. 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 R1

… identical to previous step …

} else {

// Either in Region R2 or R3

//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 R2

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

} else {

// Step C3: In Region R3

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 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 examine the motions of these rigid shapes.