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 (Step 1), collision with centers of the two circles located at different positions (Step 2), and 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;

}

// 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);

}

//… 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 (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 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 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 and 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, derives a line (or axis) that is perpendicular to the edge, project all edges of the two convex shapes onto this line, and compute for 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 shape edges onto these two lines, 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 edges 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

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. This is illustrated in Figure 9-12 for the face normal A1 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 This project will guide you to implement 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.

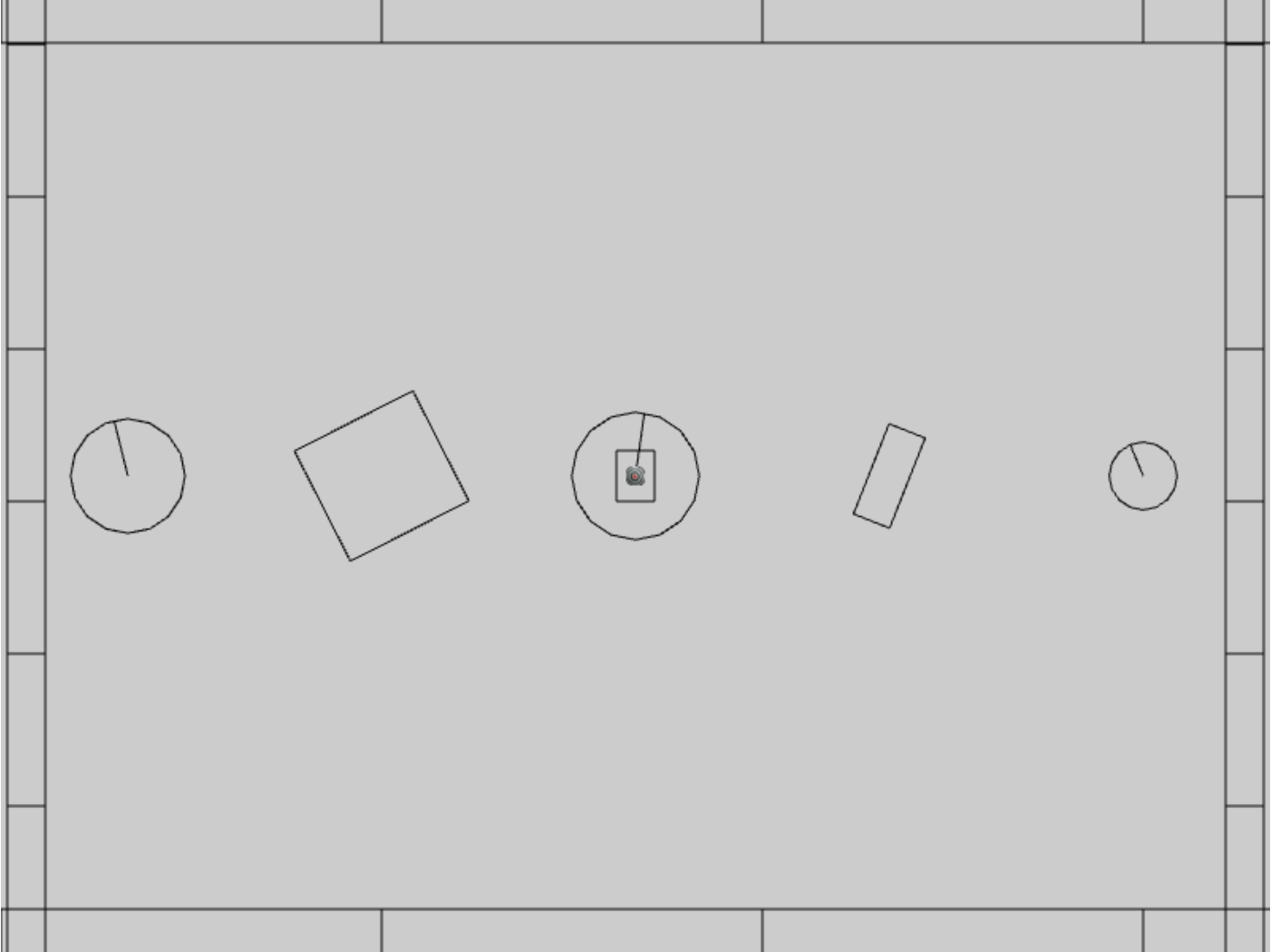


Figure 9-14. Running the Rectangle Collisions project

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 circles and rectangles and 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 completed collision detection infrastructure from the previous project, 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: 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.

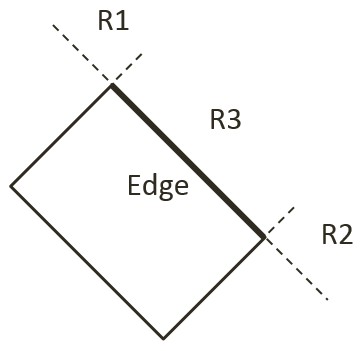


Figure 9-15. The Three Regions Outside a Given Edge of a Rectangle

## 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.