**Incorporating Collision Detection**

In the context of 2D video games, the fundamentals of a physical simulation involves movements of rigid shapes, collisions of the moving shapes, and responses after the collisions. In the previous chapter, you defined the rigid shape classes and a core engine loop to support basic drawing, update operations, and simple movements of rigid shapes. In this chapter, you will learn about and implement the detection of rigid shape collisions and compute the necessary information, such that in the next chapter you can begin resolving and implement the responses to the collisions. The proper implementation based on these concepts enables believable scenarios when objects physically interact with each other in the simulated world.

This chapter focuses on the foundations of detecting collisions, including how to approximate the detection, a theory for exact detection of colliding rectangles and circles in any orientations, and essential information to capture after detecting a collision to support resolution of interpenetration and proper responses to collisions. You will implement this system in a step by step manner, from a simple broad phase collision detection method, to the more accurate and computationally more costly Separating Axis Theorem (SAT). In this way, at each step the collision detection will become more accurate and be applicable to more general cases until your solution is ready to be used in the next chapter for resolving and responding to collisions. The final result of this chapter will be a collision detection system that can detect collisions between rigid rectangles and circles of any size and in orientation where the information required for resolving and responding to the collisions are computed and available.

After completing this chapter, you will be able to:

* Appreciate the significant computational cost of detecting object collisions.
* Optimize object collision detection with broad phase collisions to avoid unnecessary computations.
* Understand that, in a computer simulation, rigid bodies can interpenetrate during a collision and that this interpenetration must be resolved.
* Learn and use the Separating Axis Theorem (SAT) to detect rigid body collisions.
* Compute the necessary information to support efficient. In the next chapter, you will learn about effective resolution of rigid body interpenetration using this computed information.
* Implement an efficient collision detection algorithm that is based on SAT.
* Detect collisions between rigid rectangles and circles accurately.

# Interpenetration of Colliding Objects

As illustrated in Figure 3-1, the fixed update time step introduced in previous chapter means object positions in continuous motion is approximated by a discrete set of positions. The most notable ramifications of this approximation are in detecting collisions.



Figure 3-1: A Rigid Square in Continuous Motion

You can see one such problem in Figure 3-1; imagine a thin wall existed in the space between the current and the next update. You would expect the object to collide and stop by the wall in the next update. However, if the wall were thin enough, the object would essentially pass right through it as it jumped from one position to the next. This is a common problem faced in many game engines. A general solution for these types of problems can be algorithmically complex and computationally intensive. It is typically the job of the game designer to mitigate and avoid this problem with well-designed (for example, appropriate size) and well-behaved (for example, appropriate traveling speed) game objects.

Figure 3-2 shows two objects colliding after a time step. Before the time step, the objects are not touching. However, after the time step, the results of the movement simulation place the two objects over each other.



Figure 3-2: The Interpenetration of Colliding Objects

This is another example ramification of fixed update time step with discrete intervals. In the real world, given that the objects were solid, the two would never interpenetrate. This is where details of a collision must be computed such that the interpenetrating situation can be properly resolved.

# Collision Detection

Collision detection is a vital and potentially a costly piece of physics simulations that can impact performance significantly. For example, if you want to detect the collisions between five objects, in the worst case you must perform four detection computations for the first objects, followed by three computations for the second, two for the third, and one for the fourth. In general, without dedicated optimizations, in the worst case you must perform operations to detect the collisions between objects.

In addition to reporting if a collision has occurred, a collision detection algorithm should also support the computation of information that can be used to resolve and respond to the collision. This information can include penetration depth, and the normal vector of penetration. It is important to compute this information accurately such that the collision can be effectively resolved and the response properly computed to simulate the real world. Remember that object interpenetration does not happen in real world, thus the computed information are only approximation of the actual law of physics.

# Broad Phase Method

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 are only deployed for objects that are physically closed to each other.

A popular broad phase method uses bounding boxes/circles to approximate collisions between all objects. A bounding box is an x/y-axes aligned rectangular box that completely bounds a given object. The term x/y-axes aligned refers to the fact that the four sides of a bounding box are parallel to the horizontal x-axis and to the vertical y-axis. Similarly, 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 narrow down the candidates for detailed collision detection operations to only those with colliding bounds.

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.

This chapter only introduces you to the bounding circle broad phase collision method followed by a narrow phase algorithm that is based on the Separation Axis Theorem (SAT).

## The Broad Phase Method Project

This project demonstrates how to implement a broad phase collision detection method using bounding circles. You can see an example of this project running in Figure 3-3. The source code to this project is defined in the Broad Phase Method Project folder.

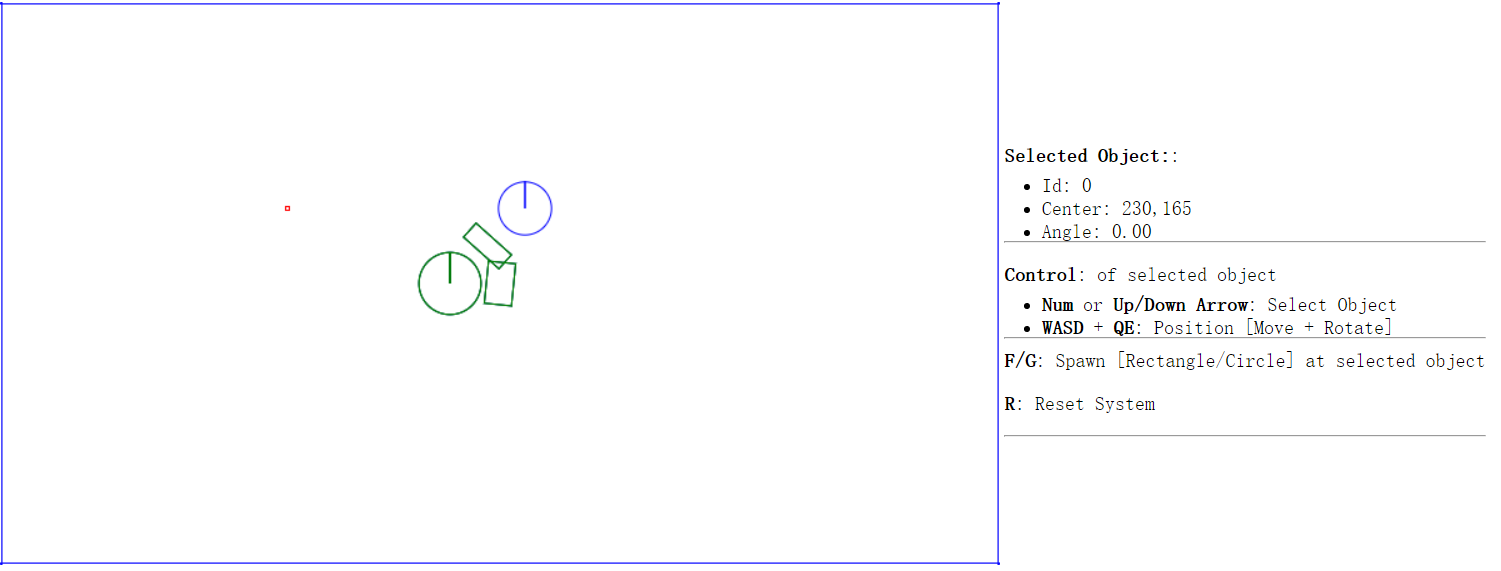


Figure 3-3. Running the Broad Phase Method Project.

Project goals:

* To understand the implementation of bounding circle collision detection
* To understand the strengths and weaknesses of broad phase collision detection
* To lay the foundation for building a narrow phase collision detection algorithm

### Define the Physics Engine Component

A physics engine component can now be defined to support the collision detection computations. To begin, follow the steps of defining an engine component.

1. In the SiteRoot/EngineCore (or public\_html/EngineCore) folder, create a new file and name it Physics.js. This file will implement the physics engine component. Remember to load this new source file in index.html.
2. Define the physics component in a similar fashion as you defined gEngine.Core:

var gEngine = gEngine || { };

gEngine.Physics = (function () {  
 var mPublic = {

};  
 return mPublic;  
}());

1. Create a collision function within gEngine.Physics to test the intersection of bounding circles between all objects in the mAllObjects list. Notice the nested loops that test every object against each other for collision and that the colliding objects are drawn with green color.

var collision = function () {

var i, j;

for (i = 5; i < gEngine.Core.mAllObjects.length; i++) {

for (j = i + 1; j < gEngine.Core.mAllObjects.length; j++){

If (gEngine.Core.mAllObjects[i].boundTest(gEngine.Core.mAllObjects[j])) {

gEngine.Core.mContext.strokeStyle = 'green';

gEngine.Core.mAllObjects[i].draw(gEngine.Core.mContext);

gEngine.Core.mAllObjects[j].draw(gEngine.Core.mContext);

}

}

}

};

1. Add public variable within mPublic to allow access to the collision function.

var mPublic = {

collision: collision

};

### Invoke the Physics Collision and Update the UI

Edit the Core.js file in the SiteRoot/EngineCore (or public\_html/EngineCore) folder.

1. Invoke the collision computation from the runGameLoop function within the core engine loop.

//….identical to previous project

while (mLagTime >= kMPF) {

mLagTime -= kMPF;

gEngine.Physics.collision();

update();

}

//….identical to previous project

1. Modify the updateUIEcho function to remove support for the H button. The gravity on/off functionality is no-longer required.

//...identical to previous project

"<b>F/G</b>: Spawn [Rectangle/Circle] at selected object" +

"<p><b>H</b>: Fix object</p>" + // remove this line

"<p><b>R</b>: Reset System</p>" +

### Modify Rigid Shape Classes

Now you can modify all the files inside the rigid shape folder to support a bounding circle test for the broad phase collision detection method.

1. You need to modify rigid shape base class. Open RigidShape.js under the folder SiteRoot/RigidBody (or public\_html/RigidBody).
2. Add the mBoundRadius variable to the RigidShape constructor. This is the radius of the bounding circle for the rigid shape.

this.mBoundRadius = 0;

1. Define a new prototype function, and name it boundTest, a function that will test if two bounding circles have collided. The most straightforward way to detect the collision between two circles is to determine if the distance between the two centers is less than the sum of the radii. The scenario is depicted in Figure 3-4.

RigidShape.prototype.boundTest = function (otherShape) {

var vFrom1to2 = otherShape.mCenter.subtract(this.mCenter);

var rSum = this.mBoundRadius + otherShape.mBoundRadius;

var dist = vFrom1to2.length();

if (dist > rSum) {

return false; //not overlapping

}

return true;

};

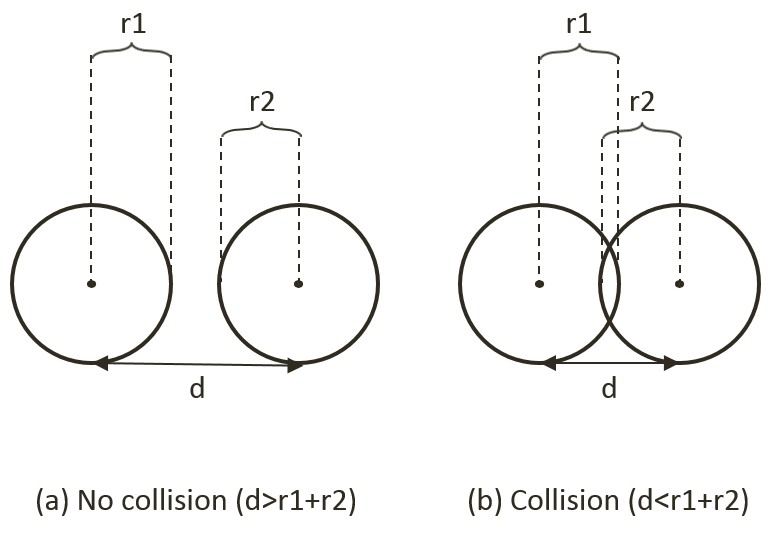


Figure 3-4. Circle Collision Detection: (a) No collision (b) Collision detected.

1. You also need to remove the movement testing code that was defined update function of the RigidShape base class.

RigidShape.prototype.update = function () { };

1. Next, modify the Circle.js file in the same folder to initialize the value for the mBoundRadius variable in the constructor. The bounding circle of a rigid circle shape has the same radius as the rigid shape. Remember to remove the mFix variable.

this.mBoundRadius = radius;

**this.mFix** **=** **fix;** //remove this line

1. Modify the Rectangle.js file for a similar purpose, to initialize the mBoundRadius variable in the constructor. In this case, the bounding circle for a rectangle rigid shape is defined as half of the diagonal distance of the rectangle. Once again, remember to remove the unused mFix variable.

this.mBoundRadius = Math.sqrt(width\*width + height\*height)/2;

## Observation

Run the project to test your implementation. Notice that by default, objects are created in the same location, have bounding circles that overlap, and thus are drawn in green color. You can select an object and move/rotate it to observe the green color changing back to black when there are no overlaps of their corresponding bounding circles. Now, create a rectangle and a circle, and move them apart. Rotate the rectangle and move it close to each other, but without actually touching, the circle. You may notice that although the two shapes are not touching and yet both are drawn in green. That is because the collision bound for the rectangle is a circle, which overestimates the bounds of the object as shown in Figure 3-5. This is the most important drawback with this broad phase method: though efficient, it is inaccurate. This issue will be remedied by the SAT algorithm to be introduced in a later section.

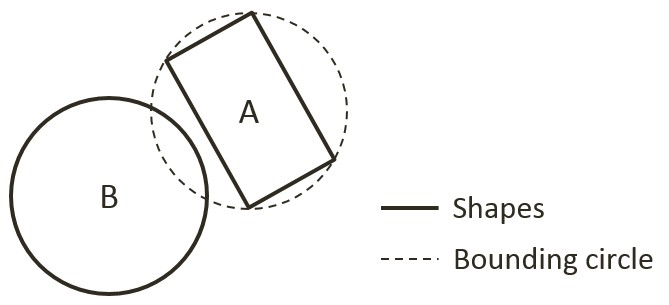


Figure 3-5. False positive collision between Rectangle-A and Circle-B

# Collision Information

With the broad phase collision method implemented, you can now begin the process of defining narrow phase methods for detecting the collision between different rigid shapes. As discussed earlier, information regarding the specifics of a collision must be computed to support proper resolution of interpenetration and response. As illustrated in Figure 3-6, the essential information of a collision includes: 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.

This section leads you to develop the infrastructure for computing and working with collision information based on collisions between rigid circle shapes--a straightforward extension to the previous project. After this section, with the proper support for storing and accessing collision information, the Separating Axis Theorem (SAT) will be introduced and implemented.



Figure 3-6. Collision Information.

## The Circle Collision Detection Project

This project builds the infrastructure for computing and working with collision information based on collisions between circles. As will be discussed, collision information records the specific details of a collision for resolving interpenetration and generating responses. Notice that the bounding circle based broad phase collision detection method computes the exact collision detection solution for rigid circle shapes. For this reason, this project can take advantage of previous project and focus on computing and working with collision information. You can see an example of this project running in Figure 3-7. The source code to this project is defined in the Circle Collision Detection Project folder.

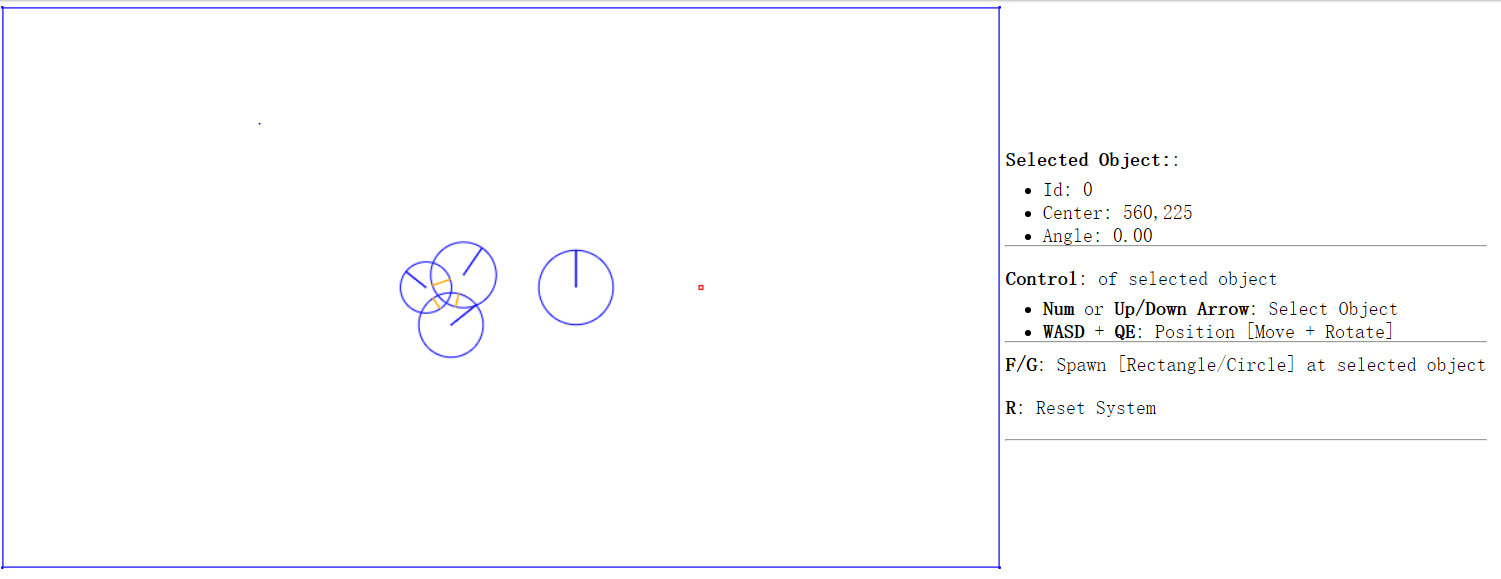


Figure 3-7. Running the Circle Collision Detection Project.

Project goals:

* To define collision information
* To build the infrastructure for computing and working with collision information
* To compute and display collision information for circles

### Define Collision Information Object

A new class must be defined to support the storage of collision information.

1. Under the SiteRoot/Lib (or public\_html/Lib) folder, create a new file and name it CollisionInfo.js. Remember to load this new source file in index.html.
2. Define the constructor of the object to contain collision depth, collision normal, and a start and end positions. These are the beginning and ending positions of a collision interpenetration.

function CollisionInfo() {

this.mDepth = 0;

this.mNormal = new Vec2(0, 0);

this.mStart = new Vec2(0, 0);

this.mEnd = new Vec2(0, 0);

}

1. Define the getter and setter for the object.

CollisionInfo.prototype.setNormal = function (s) { this.mNormal = s; };

CollisionInfo.prototype.getDepth = function () { return this.mDepth; };

CollisionInfo.prototype.getNormal = function () { return this.mNormal; };

CollisionInfo.prototype.setInfo = function (d, n, s) {

this.mDepth = d;

this.mNormal = n;

this.mStart = s;

this.mEnd = s.add(n.scale(d));

};

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

CollisionInfo.prototype.changeDir = function () {

this.mNormal = this.mNormal.scale(-1);

var n = this.mStart;

this.mStart = this.mEnd;

this.mEnd = n;

};

### Compute Collision Information Between Two Circles

In the previous project you implemented the functionality for detecting collisions between two circles. In the following, you will amend the computation of collision information to include the information gained from circle collisions.

1. Create a new file under the SiteRoot/RigidBody (or public\_html/RigidBody) folder, name it Circle\_collision.js. This file will contain the implementation of colliding a rigid circle shape with other rigid shapes.
2. Define the collisionTest function to collide a rigid circle shape with another RigidShape object. Notice that the actual collision testing function is shape specific. For now, a circle only knows how to collide with a circle and will always return false for any other shapes.

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

var status = false;

if (otherShape.mType === "Circle")

status = this.collidedCircCirc(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, collision with centers of the two circles located at different, and at exactly the same positions. The following code shows the detection of no collision. The details are depicted in Figure 3-7, vFrom1to2 is the vector pointing from center of c1 to center of c2; rSum is the sum of the radii, and dist is the distance between the centers of two circles.

Circle.prototype.collidedCircCirc = function (c1, c2, collisionInfo) {

var vFrom1to2 = c2.mCenter.subtract(c1.mCenter);

var rSum = c1.mRadius + c2.mRadius;

var dist = vFrom1to2.length();

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

return false; //not overlapping

}

// … details in the following steps

};

1. A collision is detected when dist, the distance between the centers of the two circles, is less than the sum of the radii. In this case, if the two circles do not have centers located at the exact same position, the collision depth and normal can be computed. As illustrated in Figure 3-8, 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 simple c2’s radius distance away from the center of c2 along the normalFrom2to1 direction.



Figure 3-8. Details of a Circle-Circle Collision.

//… continue from the previous step

if (dist !== 0) {

// overlapping but not same position

var normalFrom2to1 = vFrom1to2.scale(-1).normalize();

var radiusC2 = normalFrom2to1.scale(c2.mRadius);

collisionInfo.setInfo(rSum - dist, vFrom1to2.normalize(), c2.mCenter.add(radiusC2));

}

//… details in the next step

1. The last case for two colliding circles is when both circle's centers are located in exactly the same position. In this case, as shown in the following code, 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 {

//same position

if (c1.mRadius > c2.mRadius)

collisionInfo.setInfo(rSum, new Vec2(0, -1),

c1.mCenter.add(new Vec2(0, c1.mRadius)));

else

collisionInfo.setInfo(rSum, new Vec2(0, -1),

c2.mCenter.add(new Vec2(0, c2.mRadius)));

}

### Case for Collision with a Rectangle

The collision computations for a rectangle will be covered later in this chapter. For now, an empty structure will be defined to avoid runtime errors.

1. Create a new file under the SiteRoot/RigidBody (or public\_html/RigidBody) folder, name it Rectangle\_collision.js.
2. Add the following code to the file to return a false condition for all collisions with a rectangle rigid shape for now.

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

var status = false;

if (otherShape.mType === "Circle")

status = false;

else

status = false;

return status;

};

### Modify Physics Engine Component

You can now modify the physics component to support the computation of collision information when computing circle to circle collisions.

1. Edit EngineCore/Physics.js to support the drawing of collision information and to call the newly defined rigid shape collisionTest function.
2. For debugging and testing purposes, define the drawCollisionInfo function to draw the collision depth and normal as an orange colored line over the rigid shape.

var drawCollisionInfo = function (collisionInfo, context) {

context.beginPath();

context.moveTo(collisionInfo.mStart.x, collisionInfo.mStart.y);

context.lineTo(collisionInfo.mEnd.x, collisionInfo.mEnd.y);

context.closePath();

context.strokeStyle = "orange";

context.stroke();

};

1. In the collision function, first create a collisionInfo object to record the details of collisions. After the broad phase boundTest returns true, the details for the collision must be determined by calling the rigid shape collisionTest function you just defined.

//….identical to previous project

var collisionInfo = new CollisionInfo();

for (i = 0; i < gEngine.Core.mAllObjects.length; i++) {

for (j = i + 1; j < gEngine.Core.mAllObjects.length; j++) {

if (gEngine.Core.mAllObjects[i].boundTest(gEngine.Core.mAllObjects[j])) {

if (gEngine.Core.mAllObjects[i].collisionTest(gEngine.Core.mAllObjects[j], collisionInfo)) {

// … details in the next step

}

}

//….identical to previous project

1. When a collision is deemed valid, it is important to ensure that the collision normal is always in the direction towards the object being tested. As illustrated in the following code, this can be determined by the sign of the dot product between the collision normal and the vector defined by the centers of the colliding objects. drawCollisionInfo function is called to draw the corresponding collision information.

//… continue from the previous step

if (gEngine.Core.mAllObjects[i].collisionTest(gEngine.Core.mAllObjects[j], collisionInfo)) {

**//make sure the normal is always from object[i] to object[j]**

if (collisionInfo.getNormal().dot(gEngine.Core.mAllObjects[j].mCenter.subtract(

gEngine.Core.mAllObject[i].mCenter)) < 0) {

collisionInfo.changeDir();

}

//draw collision info (a black line that shows normal)

drawCollisionInfo(collisionInfo, gEngine.Core.mContext);

**}**

//… identical to previous project

## Observation

Run the project to test your implementation. Notice that when you create two circles, their collision is no longer indicated by a change of color. Instead orange lines are drawn inside the colliding circles to indicate the corresponding collision depth and normal. You can create and observe the collision information drawn on all colliding circles. The collision information will be used to resolve collision interpenetrations. Lastly, observe that collision information is absent from rigid rectangle shapes. This is because you have not implemented the functionality and that the corresponding collisionTest function always returns false. The next two projects will guide you through the implementation of collision computation with rigid rectangle shape.

# 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 overly computationally intensive for real time systems, it is typically preceded with an initial pass of broad phase method, as introduced in the previous section. 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 you can iterate through all of the edges of the convex shapes, one at a time. For each of the edges, compute 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 3-9 illustrates this description using two axes-aligned rectangles. In this case, there are two lines that are perpendicular to the two given shapes, the X and Y axes.



Figure 3-9. There exist a projection that does 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 3-9, 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:

* **Step 1 Compute Face Normals**: Compute the perpendicular axes, or face normals for projecting the edges. As illustrated in Figure 3-10, a rectangle has 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 rigid rectangle implementation, mFaceNormal, or face normals, are the perpendicular axes A1, A2, A3, and A4.



Figure 3-10. Rectangle Edges and Face Normals

* **Step 2 Project Vertices**: Project each of the vertices of the two convex shapes onto the face normals. Figure 3-11 illustrates this projection of all vertices onto the A2 axis from Figure 3-10.



Figure 3-11. Project each vertex onto face normals (example shows A1)

* **Step 3 Identify Bounds**: Identifies the min and max bounds for the projected vertices of each convex shape. Continue with the previous rectangle example, Figure 3-12 shows identifying 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 3-12. Identify the min and max bound positions for each rectangle.

* **Step 4 Determine overlaps**: Determines if the two min/max bounds overlap. Figure 3-13 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 the drawing on the right of Figure 3-10, process of face normal B1 will result in a deterministic conclusion of no collision.



Figure 3-13. Test for overlap for every axis of projection (example using 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 3-6. 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 3-14 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 3-14. 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, the face normal B1 of shape-B in Figure 3-14 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 front* of 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 3-15 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 3-15 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 3-15. 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 point 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 Collision Project

This project will guide you to implement the support point SAT algorithm. You can see an example of this project running in Figure 3-16. The source code to this project is defined in the Rectangle Collision Project folder.

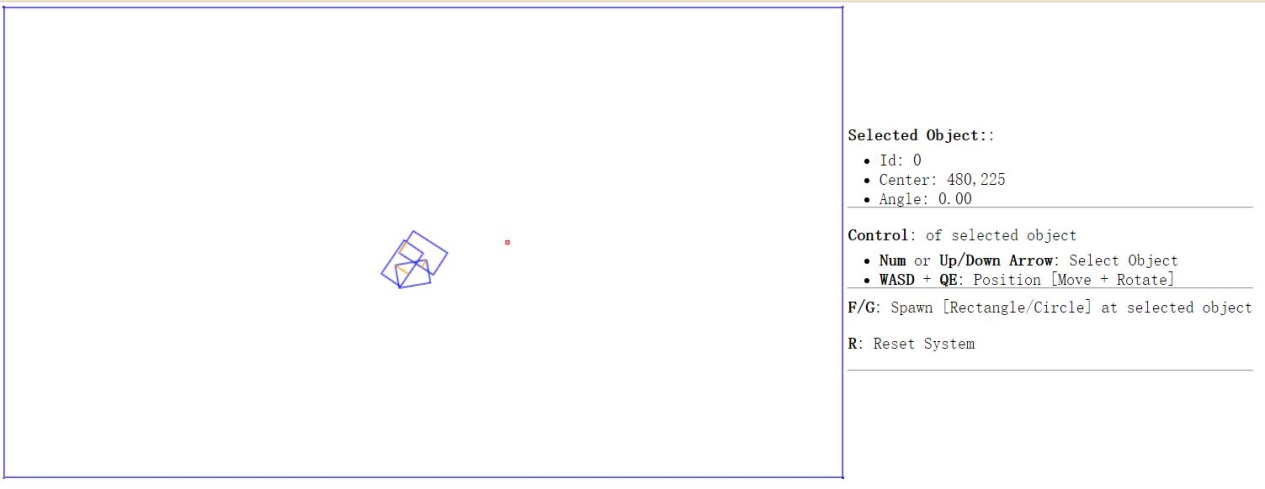


Figure 3-16. Running the Rectangle Collision Project.

Project goals:

* To gain insights into and implement the support point SAT algorithm

### Modify Rectangle Collision

Begin by modifying the Rectangle\_collision.js file to implement the collision detection between rectangles.

1. Edit the Rectangle\_collision.js file in the RigidBody folder.
2. 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 following 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. Additionally, 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.

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

//the longest project length

var vToEdge;

var projection;

// initialize the computed results

tmpSupport.mSupportPointDist = -9999999;

tmpSupport.mSupportPoint = null;

//check each vector of other object

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

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

projection = vToEdge.dot(dir);

//find the longest distance with certain edge

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

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

tmpSupport.mSupportPoint = this.mVertex[i];

tmpSupport.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 by implementing the findAxisLeastPenetration function. Recall that the axis of least penetration is derived based on the support point with the least support point distant. The following 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.

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

var n;

var supportPoint;

var bestDistance = 999999;

var bestIndex = null;

var hasSupport = true;

var i = 0;

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

// Retrieve a face normal from A

n = this.mFaceNormal[i];

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

var dir = n.scale(-1);

var ptOnEdge = this.mVertex[i];

// find the support on B

// the point has longest distance with edge i

otherRect.findSupportPoint(dir, ptOnEdge);

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

//get the shortest support point depth

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

bestDistance = tmpSupport.mSupportPointDist;

bestIndex = i;

supportPoint = tmpSupport.mSupportPoint;

}

i = i + 1;

}

if (hasSupport) {

//all four directions have support point

var bestVec = this.mFaceNormal[bestIndex].scale(bestDistance);

collisionInfo.setInfo(bestDistance, this.mFaceNormal[bestIndex], supportPoint.add(bestVec));

}

return hasSupport;

};

1. You can now implement the collidedRectRect function by computing the axis of least penetration with respective 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.

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

var status = false;

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

status = false;

} else {

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

}

return status;

};

## Observation

You can now run the project to test your implementation. Try creating multiple rectangles with the F key. You can see an orange line representing collision information (collision depth, in the collision normal direction, from start to end) when two or more rectangles collide. Remember that this line shows the least amount of positional correction needed to resolve the collision. Use to up and down arrows to select and rotate the rectangles and observe how the collision info changes accordingly. At this stage you have implemented collision detection between a circle and a circle, as well as a rectangle and another rectangle. If you try to collide a rectangle and a circle, no collision info is generated because you have not implemented support for this type of collision. 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 3-17, 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 nearest edge (the edge on the rectangle that is closest to the circle center).
* **Step B**: If circle center is outside

**Step B1**: If in Region R1: distance between the circle center and left/top vertex from the Edge determines if collision has occurred.

**Step B2**: If in Region R2: distance between the circle center and right/bottom vertex from the Edge determines if collision has occurred.

**Step B3**: If in Region R3: perpendicular distance between the center and the Edge determines if collision has occurred.

* **Step C**: If the circle center is inside the rectangle: collision is detected.

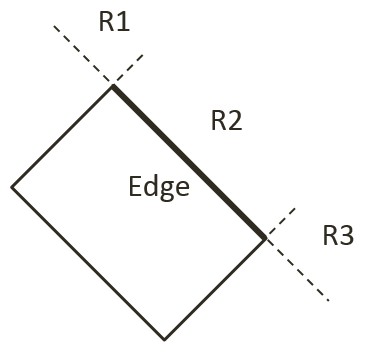


Figure 3-17. The Three Regions Outside a Given Edge of a Rectangle.

## The Rectangle Circle Collision Project

This project guides you in implementing the described rectangle-circle collision detection algorithm with detailed discussions for each of the steps. You can see an example of this project running in Figure3-18. The source code to this project is defined in the Rectangle Circle Collision Project folder.

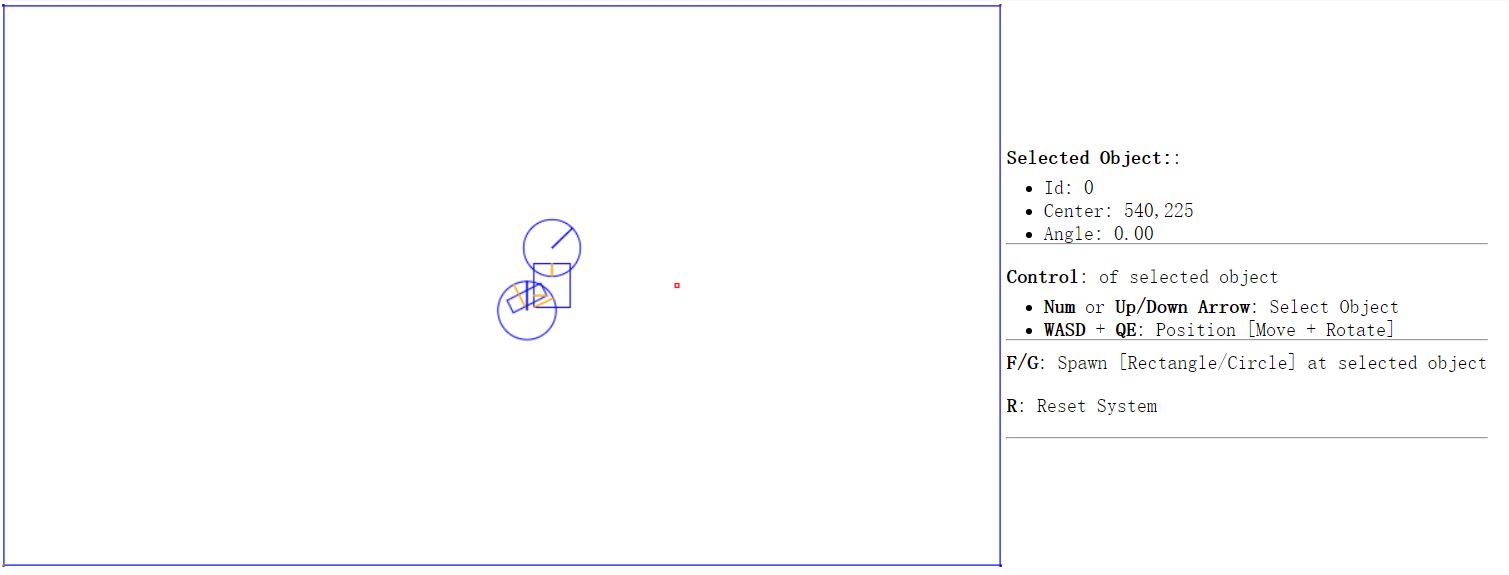


Figure 3-18. Running the Rectangle Circle Collision Project.

Project goals:

* To understand and implement the rectangle circle collision detection algorithm.

### Modify Rectangle Collision

You are going to implement the described algorithm in the Rectangle\_collision.js file.

1. Edit the Rectangle\_collision.js file in the RigidBody folder.
2. Create a new function, collidedRectCirc, to detect the collision between a rectangle and a circle. Accordingly, there will be five major steps in this function. The following listing collapsed all of the steps with detailed to be filled-in in the rest of this section.

Rectangle.prototype.collidedRectCirc = function (otherCir, collisionInfo) {

// **Step A**: Compute the nearest edge

if (!inside) {

// **Step B1**: If center is in Region R1

// **Step B2**: If center is in Region R2

// **Step B3**: If center is in Region R3

} else {

// **Step C**: If center is inside

}

return true;

};

1. **Step A**: Compute the nearest edge. The nearest edge can be computed 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 between each vertex and the circle center onto the corresponding face normal. The following 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 normals.

// **Step A**: Compute the nearest edge

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

//find the nearest face for center of circle

circ2Pos = otherCir.mCenter;

v = circ2Pos.subtract(this.mVertex[i]);

projection = v.dot(this.mFaceNormal[i]);

if (projection > 0) {

//if the center of circle is outside of rectangle

bestDistance = projection;

nearestEdge = i;

inside = false;

break;

}

if (projection > bestDistance) {

bestDistance = projection;

nearestEdge = i;

}

}

As illustrated in Figure 3-19, one interesting observation is that 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, when the center is outside of the rectangle then, at least one of the projected length is positive. For this reason, the “nearest projected distance” is the one with the least negative value and thus is actually the largest number.



Figure 3-19. (a) Center inside the rectangle will result in all negative projected length. (b) Center outside the rectangle will result in at least one positive projected length.

1. **Step B1**: if center is outside of the rectangle and in Region R1. As illustrated in Figure 3-20-a, the Region R1 can be detected when v1, the vector between the center and the edge vertex is in the opposite direction of v2, the direction of the edge. This is to say, the center of the circle is in Region R1 when the dot product of those two vectors is negative. Figure 3-20-b 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 3-20. (a) Condition when Center is in Region R1. (b) The corresponding collision information.

// **Step A**: Compute the nearest edge (*details discussed*)

if (!inside) { //the center of circle is outside of rectangle

// **Step B1**: if ceter is in Region R1

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

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

var v1 = circ2Pos.subtract(this.mVertex[nearestEdge]);

var v2 = this.mVertex[(nearestEdge + 1) % 4].subtract(this.mVertex[nearestEdge]);

var dot = v1.dot(v2);

if (dot < 0) { // Region R1

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

var dis = v1.length();

//compare the distance with radium to decide collision

if (dis > otherCir.mRadius)

return false;

var normal = v1.normalize();

var radiusVec = normal.scale(-otherCir.mRadius);

collisionInfo.setInfo(otherCir.mRadius - dis, normal, circ2Pos.add(radiusVec));

} else { // Not in Region R1

// … *details to follow* …

// **Step B2**: If center is in Region B2

if (…) { // in Region R2

// … *details to follow* …

} else { // not in Region R2

// **Step B3**: If center is in Region B3

// … *details to follow* …

}

}

} else { // *else of (!inside)*

// **Step C**: If center is inside the rectangle

// … *details to follow* …

}

1. **Step B2**: if the center is outside of the rectangle and in Region R2. The following code complements that of Step B1, with the only difference being the direction of v2, the vector along the edge. In this case, the vector along the edge is in the opposite direction as compared for working with Region R1.

// **Step A**: Compute the nearest edge (*details discussed*)

if (!inside) {

// **Step B1**: If center is in Region R1 (*detailed discussed*)

} else {

// **Step B2**: If center is in Region R2

//the center of circle is in corner region of mVertex[nearestEdge+1]

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

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

var v1 = circ2Pos.subtract(this.mVertex[(nearestEdge + 1) % 4]);

var v2 = v2.scale(-1);

var dot = v1.dot(v2);

if (dot < 0) {

var dis = v1.length();

//compare the distance with radium to decide collision

if (dis > otherCir.mRadius)

return false;

var normal = v1.normalize();

var radiusVec = normal.scale(-otherCir.mRadius);

collisionInfo.setInfo(otherCir.mRadius - dis, normal, circ2Pos.add(radiusVec));

} else {

// **Step B3**: If center is in Region B3

// … *details to follow* …

}

1. **Step B3**: If the center is in Region R3. The last possible 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, if this distance is less than the circle radius then collision occurred.

// **Step B3**: If center is in Region B3

//the center of circle is in face region of face[nearestEdge]

if (bestDistance < otherCir.mRadius) {

var radiusVec = this.mFaceNormal[nearestEdge].scale(otherCir.mRadius);

collisionInfo.setInfo(otherCir.mRadius - bestDistance,

this.mFaceNormal[nearestEdge], circ2Pos.subtract(radiusVec));

} else {

return false;

}

1. **Step C**: If the circle center is inside the rectangle, then collision is detected and the corresponding collision information can be computed and returned.

if (!inside) {

//… *conditions for Region R1, R2, and R3 as discussed*

} else {

//the center of circle is inside of rectangle

var radiusVec = this.mFaceNormal[nearestEdge].scale(otherCir.mRadius);

collisionInfo.setInfo(otherCir.mRadius - bestDistance,

this.mFaceNormal[nearestEdge], circ2Pos.subtract(radiusVec));

}

return true;

};

1. The last step is to modify the collisionTest function to call the newly defined collision function accordingly.

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

var status = false;

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

status = this.collidedRectCirc(otherShape, collisionInfo);

} else {

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

}

return status;

};

## Observation

You can now run the project to test your implementation. You can create rectangles and circles, move and rotate them to observe the corresponding collision information represented by orange lines. Rotate colliding rectangles to observe the collision information adapting to the shape’s rotation. That is because the calculated collision information is depend on the position of vertex and face normal of the rectangle. However, when you rotate a colliding circle, the collision information does not change. That is because the calculated collision information is only dependent on the circle's center position and its radius. For this reason, the rotation of a circle does not change its collision information.

# Summary

At this stage, your physics engine simulation is capable of detecting collisions accurately, and computing the appropriate collision information when rigid objects collide. You have been introduced to broad phase method, the Separating Axis Theorem, and support points for efficiently detecting collisions of convex shapes. You have implement algorithms based on these concepts that successfully detect collisions and compute the associated information necessary for resolving any potential interpenetrations. The next chapter will introduce you to some elementary physics about movements, and how to use the computed collision information for simulating a real-world physics interactions in 2D space by properly resolving collisions.