Simulating the Rigid World

# Interpenetration of Colliding Objects

Although no movements yet, once start moving, objects can easily overlap, or interpenetrate, start by building representation for this interpenetration and later to refine and correct this. …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.

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

**SHOULD THIS BE BEFORE or After CollisionInfo** 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.

# 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. **NEED a new diagram on circle/circle**

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 Collisions and CollisionInfo 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. This project introduces the rigid shapes and the bounds. You can see an example of this project running in Figure 9-X2. The source code to this project is defined in chapter9/9.2.circle\_collisions\_and\_ colllision\_info.

Figure 9-X1. Running the CollisionInfo and Circle Collisions project

The controls of the project are as follows, for both scenes:

* **This and that**
* **This and that**.

The goals of the project are as follows:

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

You can find the following external resource files in the assets folder: this file and tht file (no changes)

### The Physics Component and Broad Phase Detection

You Physics.js, focus on the loop

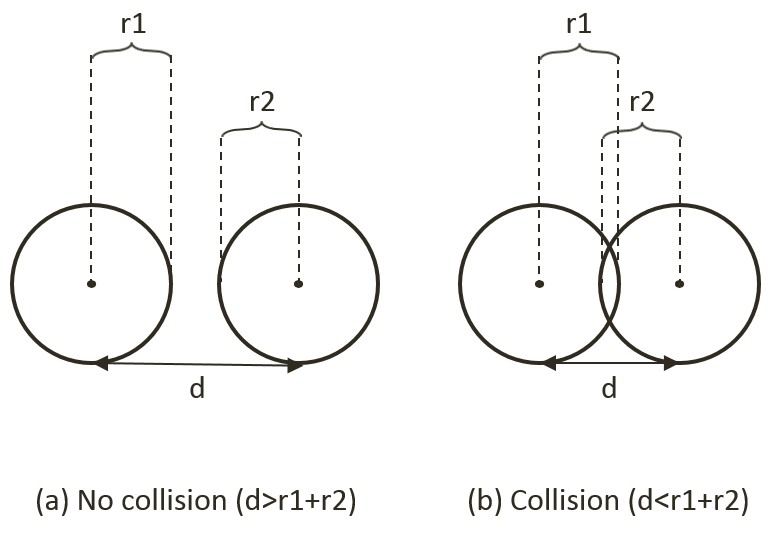


Figure 3-4. Circle Collision Detection: (a) No collision (b) Collision detected.

#### Export Physics to the Client

integrating texture functionality into the engine involves modifying the engine access file, index.js. Edit index.js and add in the following import and export statements to grant client access to this and that

### Define Collision Info

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;

};

### Testin

### The collision

### Observations

Run 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.