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 Position Correction

In the context of game engines, collision resolution refers to the process that determines how objects respond after a collision, including strategies to resolve the potential interpenetration situations that may occur. Notice that there are no collision resolution processes in the real world where interpenetration of rigid objects can occur since collisions are strictly governed by the law of physics. Resolutions of interpenetrations are relevant only in a virtual simulated world, where movements are approximated and impossible conditions may occur but can be resolved in ways that are desirable to the developer or designer.

In general, there are three common methods for responding to interpenetrating collisions. The first is to simply displace the objects from one another by the depth of penetration. This is known as the Projection Method since you simply move an object’s position so that it no longer penetrates the other. While this is simple to calculate and implement, it lacks stability when many objects are in proximity and resting upon one another.

The simple resolving of one pair of interpenetrating objects can result in new penetrations with other nearby objects. However, this is still a common method for simple engines or games with simple object interaction rules. For example, in the Pong game, the ball never comes to rest on the paddles or walls and continuously remains in motion by bouncing off any object it collides with. The Projection Method is perfect for resolving collisions for these types of simple object interactions.

The second method is known as the Impulse Method, which uses object velocities to compute and apply impulses to initiate the objects to move in the opposite directions at the point of collision. This method tends to slow down colliding objects rapidly and converges to relatively stable solutions. This is because impulses are computed based on the transfer of momentum, which in turn has a damping effect on the velocities of the colliding objects. The third method is known as the Penalty Method, which models the depth of object interpenetration as the degree of compression of a spring and approximates an acceleration to apply forces to separate the objects. This last method is the most complex and challenging to implement.

For your engine, you will be combining the strengths of the Projection and Impulse Methods. The Projection Method will be used to separate the interpenetrating objects, while the Impulse Method will be used to apply small impulses to reduce the object velocities in the direction that caused the interpenetration. As described, the simple Projection Method can result in an unstable system, such as objects that sink into each other when stacked. You will overcome this instability by implementing a relaxation loop where interpenetrated objects are separated incrementally via repeated applications of the Projection Method in a single update cycle.

With a relaxation loop, the number of times that the Projection Method is applied is referred to as relaxation iterations. During each relaxation iteration, the Projection Method reduces the interpenetration incrementally by a fixed percentage of the total penetration depth. For example, by default the engine sets relaxation iterations to 15, and during each relaxation iteration the interpenetration is reduced by 80%. This means that within one update function call, after the movement integration approximation, the collision detection and resolution procedures will be executed 15 times. While costly, the repeated incremental separation ensures a stable system under normal circumstances.

However, the 15 relaxation iterations may not be sufficient when the system undergoes sudden large changes. For example, if a large number of significantly overlapped objects, e.g., 100 overlapped circles, were to be added to the system simultaneously, then the 15 relaxation iterations may not be sufficient. This situation can be resolved by increasing the relaxation iterations at the cost of a loss in performance. From our experience, under normal operation conditions, a relaxation iteration of around 15 is a balanced trade-off between accuracy and performance. The Collision Position Correction Project

## The Collision Position Correction Project

This project will guide you through the implementation of the relaxation iterations to incrementally resolve inter-object interpenetrations. You are going to use the collision information computed from previous chapter to correct the position of the colliding objects.You can see an example of this project running in Figure 9-X2. The source code to this project is defined in chapter9/9.6.collision\_position\_correction.

Figure 9-X1. Running the Collision Position Correction 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 appreciate the importance of the computed collision information
* To implement positional correction with relaxation iteration
* To understand and experience implementing interpenetration resolution

### Update the Physics Engine

This project will only modify Physics.js because this is the file that implements the details of collisions.

1. Edit Physics.js and add in the following variables to support the correction of positions incrementally via the relaxation iterations.

//...identical to previous project

gEngine.Physics = (function () {

var mPositionalCorrectionFlag = true;

// number of relaxation iteration

var mRelaxationCount = 15;

// percentage of separation to project objects

var mPosCorrectionRate = 0.8;

//… identical to previous project

var mPublic = {

collision: collision,

mPositionalCorrectionFlag: mPositionalCorrectionFlag

};

return mPublic;

}());

1. Modify the collision function to include an enclosing relaxation iteration loop over the collision detection loop.

var collision = function () {

var i, j, k;

for (k = 0; k < mRelaxationCount; k++) {

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

//...identical to previous project

}

}

};

1. Create a new function in gEngine.Physics and name it positionalCorrection. This function reduces the overlaps between objects by the predefined constant mPosCorrectionRate with a default value of 80%. To properly support object momentum in the simulation, the amount in which each object moves is governed by their corresponding masses. For example, upon the collision of two objects the object with a larger mass will generally move by an amount that is less than the object with smaller mass. Notice that the direction of movement is along the collision normal as defined in the collisionInfo structure.

var positionalCorrection = function (s1, s2, collisionInfo) {

var s1InvMass = s1.mInvMass;

var s2InvMass = s2.mInvMass;

var num = collisionInfo.getDepth() / (s1InvMass + s2InvMass) \* mPosCorrectionRate;

var correctionAmount = collisionInfo.getNormal().scale(num);

s1.move(correctionAmount.scale(-s1InvMass));

s2.move(correctionAmount.scale(s2InvMass));

};

1. Create another function and name it resolveCollision. This function receives two RigidShape objects as parameter, and determines if the collision detected should be positionally corrected. As pointed out previously, objects with infinite mass, or zero inversed mass, are stationary and will not participate in positional correction after a collision.

var resolveCollision = function (s1, s2, collisionInfo) {

if ((s1.mInvMass === 0) && (s2.mInvMass === 0))

return;

// correct positions

if(gEngine.Physics.mPositionalCorrectionFlag)

positionalCorrection(s1, s2, collisionInfo);

};

1. Finally, you should call the newly defined resolveCollision function from within the collision function when a collision is detected. You can invoke resolveCollision after calling the drawCollisionInfo function.

var collision = function () {

var i, j, k;

var collisionInfo = new CollisionInfo();

for (k = 0; k < mRelaxationCount; k++) {

//….identical to previous project

drawCollisionInfo(collisionInfo, gEngine.Core.mContext);

resolveCollision(gEngine.Core.mAllObject[i], gEngine.Core.mAllObject[j], collisionInfo);

//… identical to previous project

Note that the drawCollisionInfo function is a drawing operation and strictly speaking, does not belong within the update loop in the collision function part of the system. Additionally, this draw operation is invoked within the core of relaxation loop iterations, which is computationally expensive. Fortunately, this function is for debugging purposes and will be commented out after this project.

### Observations

Run the project to test your implementation. Create a few objects in the scene. Notice that with the ‘M’ key you can control whether the newly created objects overlap. Now, reset the scene with the ‘R’ key then create some objects followed by enabling movement. You will notice small amounts of interpenetration happening and when left alone objects may begin to sink below the bottom of the scene. Select any of the objects to notice the ever increasing negative y-velocity component. During each update cycle, all objects’ y-velocities are changed by gravitational acceleration and yet the positional correction relaxation iterations are preventing them from moving downwards. By disabling the movement, you will notice overlaps disappearing completely as positional correction will not be countered anymore. The ever increasing y-velocities of the objects are a serious concern when attempting to create a stable system. Continuously increasing/decreasing numbers will result in unstable and unpredictable behavior, as witnessed in the objects sinking below the bottom boundary. In the following sections you will learn about the Impulse Method to further improve collision resolutions.