Physics

# Interpenetration of Colliding Objects

The fixed update time step introduced in the previous project means that the actual location of an object in a continuous motion is approximated by a discrete set of positions. As illustrated in Figure 9-22, the movement of the rectangular object is approximated by placing the object at the three distinct positions over three update cycles. The most notable ramification of this approximation is in the challenges when determining collisions between objects.



Figure 9-22: A Rigid Square in Continuous Motion

You can see one such challenge in Figure 9-22. 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 was sufficiently thin, the object would appear to pass right through the wall 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 9-23 shows another, and more significant, collision related challenge resulting from fixed update time steps. In this case, before the time step the objects are not touching. After the time step, the results of the movement approximation place the two objects where they partly overlap. In the real world, if the two objects are rigid shapes or solids then the overlap, or interpenetration, would never occur. For this reason, this situation must be properly resolved in a rigid shape physics simulation. This is where details of a collision must be computed such that interpenetrating situations like these can be properly resolved.



Figure 9-23: The Interpenetration of Colliding Objects

# Collision Position Correction

In the context of game engines, collision resolution refers to the process that determines object responses after a collision, including strategies to resolve the potential interpenetration situations that may have occurred. Notice that in the real-world, interpenetration of rigid objects can never occur since collisions are strictly governed by the law of physics. As such, resolutions of interpenetrations are relevant only in a simulated virtual world where movements are approximated and impossible situations may occur. These situations must be resolved algorithmically where both the computational cost and resulting visual appearance must be acceptable.

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 positions of objects such that they no longer overlap. While this is simple to calculate and implement, it lacks stability when many objects are in proximity and overlap with each other. In this case, the simple resolving of one pair of interpenetrating objects can result in new penetrations with other nearby objects. However, the Projection Method is still often implemented in 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, the Impulse Method, 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, the Penalty Method, 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 compute 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, in a single update cycle, interpenetrated objects are separated incrementally via repeated applications of the Projection Method.

With a relaxation loop, each application of the Projection Method is referred to as a relaxation iteration. 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 each relaxation iteration reduces the interpenetration 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.

## 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 project to correct the position of the colliding objects. You can see an example of this project running in Figure 9-24. The source code to this project is defined in chapter9/9.6.collision\_position\_correction.



Figure 9-24. Running the Collision Position Correction project

The controls of the project are identical to the previous project with a single addition of the P key commands in behavior control:

* **Behavior control:**

**P key**: Toggle penetration resolution for all objects

**V key**: Toggle motion of all objects

**H key**: Inject random velocity to all objects

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 implement positional correction with relaxation iteration
* To appreciate the importance of and work with the computed collision information
* To understand and experience implementing interpenetration resolution

### Updating the Physics Component

The previous projects have established the required simulation infrastructure including the completion of the RigidShape implementation in the previous project. You can now focus on the details of positional correction logic which is localized and hidden in the core of the physics component in the physics.js file in the src/engine/components folder.

1. Edit physics.js to define variables and the associated getters and setters for positional correction rate, relaxation loop count, and, toggling the positional correction computation. Make sure remember to export the newly defined functions.

let mPosCorrectionRate = 0.8; // percentage of separation to project objects

let mRelaxationCount = 15; // number of relaxation iteration

let mCorrectPosition = true;

function getPositionalCorrection() { return mCorrectPosition; }

function togglePositionalCorrection() { mCorrectPosition = !mCorrectPosition; }

function getRelaxationCount() { return mRelaxationCount; }

function incRelaxationCount(dc) { mRelaxationCount += dc; }

… identical to previous code …

export {

… identical to previous code …

togglePositionalCorrection,

getPositionalCorrection,

getRelaxationCount,

incRelaxationCount

}

1. Define positionalCorrection() function to move and reduce the overlaps between objects by the predefined rate, mPosCorrectionRate. To properly support object momentum in the simulation, the amount in which each object moves is inversely proportional to their masses. That is, upon collision, an object with a larger mass will be moved by an amount that is less than the object with a smaller mass. Notice that the direction of movement is along the collision normal as defined in by the collisionInfo object.

function positionalCorrection(s1, s2, collisionInfo) {

if (!mCorrectPosition)

return;

let s1InvMass = s1.getInvMass();

let s2InvMass = s2.getInvMass();

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

let correctionAmount = [0, 0];

vec2.scale(correctionAmount, collisionInfo.getNormal(), num);

s1.adjustPositionBy(correctionAmount, -s1InvMass);

s2.adjustPositionBy(correctionAmount, s2InvMass);

}

1. Modify the collideShape() function to perform positional correction when a collision is detected. Notice that objects of collisions are only performed between objects with non-zero masses.

function collideShape(s1, s2, infoSet = null) {

… identical to previous code …

if ((s1 !== s2) && ((s1.getInvMass() !== 0) || (s2.getInvMass() !== 0))) {

if (s1.boundTest(s2)) {

hasCollision = s1.collisionTest(s2, mCInfo);

if (hasCollision) {

vec2.subtract(mS1toS2, s2.getCenter(), s1.getCenter());

if (vec2.dot(mS1toS2, mCInfo.getNormal()) < 0)

mCInfo.changeDir();

positionalCorrection(s1, s2, mCInfo);

… identical to previous code …

}

return hasCollision;

}

1. Integrate a loop in all three utility functions, processObjToSet(), processSetToSet(), and processSet(), to execute relaxation iterations in performing the positional corrections.

function processObjToSet(obj, set, infoSet = null) {

let j = 0, r = 0;

let hasCollision = false;

let s1 = obj.getRigidBody();

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

for (j = 0; j < set.size(); j++) {

let s2 = set.getObjectAt(j).getRigidBody();

hasCollision = collideShape(s1, s2, infoSet) || hasCollision;

}

}

return hasCollision;

}

function processSetToSet(set1, set2, infoSet = null) {

let i = 0, j = 0, r = 0;

let hasCollision = false;

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

… identical to previous code …

}

return hasCollision;

}

// collide all objects in the GameObjectSet with themselves

function processSet(set, infoSet = null) {

let i = 0, j = 0, r = 0;

let hasCollision = false;

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

… identical to previous code …

}

return hasCollision;

}

### Testing Positional Correction in MyGame

The MyGame class must be modified to support the new P key command, to toggle off initial motion and positional correct, and, to spawn initial objects in the central region of the game scene to guarantee initial collisions. These modifications are straightforward and details are not shown. As always, you can refer to the source code files in the src/my\_game folder for implementation details.

### 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.

**As an exercise, support report collision status without positional correct.**

No collision response, when motion is on, downwards velocity continues to increase, object will eventually drop through. This is a problem will be address in the next projectz