Simulating the Rigid World

**Add position correction and resolution as options so that physics can support detection only**

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.

you have implemented algorithms to detect collisions between rigid circles and rectangles. In addition to the boolean condition of whether a collision has indeed occurred, the algorithms you have implemented also computed information that tells you important details--the collision information, which includes, the interpenetration depth and normal direction. In this chapter, you will further expand the physics engine by using the collision information to correct the interpenetration condition, and learn about simulating collision responses that resemble real-world rigid shape behaviors. Initially, your responses will be in linear motion, and finally you will support objects rotating as a result of collisions.

you will first amend the rigid shape classes to support proper simulation of Newtonian motion and to include relevant physical attributes to allow the simulation of energy transfers between colliding objects. After you implement movements in the physics engine together with the collision detection algorithms from the previous chapter you can start resolving collisions.

# Movement

Movement is the description of how object positions change in the simulated world. Mathematically, movement can be formulated in many ways. In previous chapters, you experienced working with movement where you continuously changed the position of an object with a constant value, or a displacement. Although desired results can be achieved, mathematically this is problematic because a velocity and a position are different types of quantities with different units and the two cannot be simply combined. As illustrated in Figure 4-1 and the following equation, in practice, you have been working with describing movement based on constant displacements.



Figure 4-1. Movement Based on Constant Displacements

A movement that is governed by the constant displacement formulation becomes restrictive when it is necessary to change the amount that is displaced over time. Newtonian mechanics address this restriction by considering time in the movement formulations, as seen in the following equations.

These two equations implement a Newtonian based movement where is the velocity that describes the change in position over time and is the acceleration that describes the change in velocity over time.

Notice that both velocity and acceleration are vector quantities encoding the change in magnitude and direction. The magnitude of a velocity vector defines the speed, and the normalized velocity vector identifies the direction that the object is traveling. An acceleration vector lets you know whether an object is speeding up or slowing down via its magnitude and the direction that the acceleration is occurring in. Acceleration is changed by the forces acting upon an object. For example, if you were to throw a ball into the air, the gravitational force of the earth would affect the object’s acceleration over time, which in turn would change the object’s velocity.

## Explicit Euler Integration

The following two equations show that the Euler method, or Explicit Euler Integration, approximates integrals based on initial values. Though potentially unstable, this is one of the simplest and thus a good beginning point to learn about integration approximation methods. As illustrated in the following two equations, in the case of the Newtonian movement formulation the new velocity, , of the object can be approximated as the current velocity, , plus the current acceleration, , multiplied by the amount of elapsed time. Similarly, the object’s new position, , can be approximated by the object’s current position, , plus the current velocity, , multiplied by the amount of elapsed time.

**Note** An example of a numerically unstable system is one where under gravitational force a bouncing ball slows down but never stops jittering and, in some cases, may even start bouncing again.

The left diagram of Figure 4-2 illustrates a simple example of approximating movements with Explicit Euler Integration. Notice that the new position is computed based on the current velocity, , while the new velocity , is computed to move the position for the next update cycle.



Figure 4-2. Explicit (Left) and Symplectic (Right) Euler Integration

## Symplectic Euler Integration

In practice, because of system stability concerns, Explicit Euler Integration is seldom implemented. This shortcoming is overcome with the method you will be implementing, known as the Semi-Implicit Euler Integration or Symplectic Euler Integration, where intermediate results are used in subsequent approximations. The following equations show Symplectic Euler Integration. Notice that it is nearly identical to the Euler method except that the new velocity, , is being used when calculating the new position, . This essentially means that the velocity for the next frame is being used to calculate the position of this frame.

The right diagram of Figure 4-2 illustrates that with the Symplectic Euler Integration, the new position is computed based on the newly computed velocity, .

## The Rigid Shape Movements Project

You are now ready to implement Symplectic Euler Integration. The fixed time step update function architecture of the game engine allows the dt quantity to be implemented as the update time interval and the integral to be evaluated once per update cycle. This project will guide you through completing the rigid shape component to support movement calculations and collision responses. In addition to implement Symplectic Euler Integration, the information that you are going to add includes the attributes required for collision simulation and response, such as mass, inertia, friction, and restitution. As will be explained, each of these attributes will play a part in the calculation of simulating object movements and collision responses based on Euler integration. You You can see an example of this project running in Figure 9-X2. The source code to this project is defined in chapter9/9.5.rigid\_shape\_movements.

In addition to implement Symplectic Euler Integration, this project also defines the attributes and their corresponding accessor and getter functions. Though relatively straightforward, these functions are presented here to avoid distracting the discussions of the more complex concepts to be covered in the subsequent projects.

Figure 9-X1. Running the Rigid Shape Movements 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 experience implementing movements based on Symplectic Euler Integration
* To complete the implementation of RigidShape classes to include relevant physical attributes
* To build the infrastructure for responding to collisions

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

### Implement Symplectic Euler Integration

You must define movement support and constants in the core of the engine and in rigid shape. Loop to addin access, what else?

#### Modify the RigidShape Class

Modify the RigidShape class constructor to support velocity, angular velocity, and acceleration, as shown in the following code.

function RigidShape(center, mass, friction, restitution) {

this.mCenter = center;

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

this.mAcceleration = gEngine.Core.mGravity;

//angle

this.mAngle = 0;

//negetive-- clockwise

//positive-- counterclockwise

this.mAngularVelocity = 0;

this.mAngularAcceleration = 0;

gEngine.Core.mAllObject.push(this);

}

#### Implement Symplectic Euler Integration

You can now add the behavior to the rigid shape object for numerical integration. Continue with the RigidShape base class, complete the update function to apply Symplectic Euler Integration to the rigid shape where the updated velocity is used for computing the new position. Notice the implementation similarities between linear and angular motion. In both cases, the velocities are updated before the results are being applied to the displacements. Rotation will be examined in detailed in the last section of this chapter.

RigidShape.prototype.update = function () {

if (gEngine.Core.mMovement) {

var dt = gEngine.Core.mUpdateIntervalInSeconds;

//v += a\*t

this.mVelocity = this.mVelocity.add(this.mAcceleration.scale(dt));

//s += v\*t

this.move(this.mVelocity.scale(dt));

this.mAngularVelocity += this.mAngularAcceleration \* dt;

this.rotate(this.mAngularVelocity \* dt);

}

};

### Define Attributes to Support Collision Simulation and Response

As mentioned, in order to allow focused discussions of the more complex concepts in the later sections, the attributes for supporting collisions and the corresponding supporting functions are introduced in this project. These attributes are defined in the rigid shape class.

#### Modify the RigidShape Class

Now it’s time for the RigidShape class:

1. Modify the RigidShape class constructor again. This time to support mass, restitution (bounciness), and friction, as shown in the following code. Notice that the inverse of the mass value is actually stored for computation efficiency (by avoiding an extra division during each update calculation). Additionally, notice that a mass of zero is used to represent a stationary object.

function RigidShape(center, mass, friction, restitution) {

this.mCenter = center;

this.mInertia = 0;

if (mass !== undefined)

this.mInvMass = mass;

else

this.mInvMass = 1;

if (friction !== undefined)

this.mFriction = friction;

else

this.mFriction = 0.8;

if (restitution !== undefined)

this.mRestitution = restitution;

else

this.mRestitution = 0.2;

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

if (this.mInvMass !== 0) {

this.mInvMass = 1 / this.mInvMass;

this.mAcceleration = gEngine.Core.mGravity;

} else {

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

}

//angle

this.mAngle = 0;

//negetive-- clockwise

//positive-- counterclockwise

this.mAngularVelocity = 0;

this.mAngularAcceleration = 0;

this.mBoundRadius = 0;

gEngine.Core.mAllObject.push(this);

}

1. Define a function, updateMass, to support changing of the mass during runtime. Notice that the updateInertia function is empty. This reflects the fact that rotational inertia is shape specific and the actual implementation would be the responsibility of individual subclasses (Rectangle and Circle).

RigidShape.prototype.updateMass = function (delta) {

var mass;

if (this.mInvMass !== 0)

mass = 1 / this.mInvMass;

else

mass = 0;

mass += delta;

if (mass <= 0) {

this.mInvMass = 0;

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

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

this.mAngularVelocity = 0;

this.mAngularAcceleration = 0;

} else {

this.mInvMass = 1 / mass;

this.mAcceleration = gEngine.Core.mGravity;

}

this.updateInertia();

};

RigidShape.prototype.updateInertia = function () {

// subclass must define this.

// must work with inverted this.mInvMass

};

#### Modify the Circle and Rectangle Classes

Next, modify the Circle and Rectangle classes:

1. Modify the Circle class to implement the updateInertia function. This function calculates the rotational inertia of a circle when its mass is changed.

Circle.prototype.updateInertia = function() {

if (this.mInvMass === 0) {

this.mInertia = 0;

} else {

// this.mInvMass is inverted!!

// Inertia=mass \* radius^2

// 12 is a constant value that can be changed

this.mInertia = (1 / this.mInvMass) \* (this.mRadius \* this.mRadius) / 12;

}

};

1. Update the Circle object constructor to call the new RigidShape base class, and to accept relevant parameters of physical attributes. Remember to call the newly defined updateInertia for initialization.

var Circle = function (center, radius, mass, friction, restitution) {

RigidShape.call(this, center, mass, friction, restitution);

this.mType = "Circle";

//...identical to previous project

this.updateInertia();

};

1. Modify the Rectangle class to implement its updateIntertia function.

Rectangle.prototype.updateInertia = function() {

// Expect this.mInvMass to be already inverted!

if (this.mInvMass === 0)

this.mInertia = 0;

else {

//inertia=mass\*width^2+height^2

this.mInertia = (1 / this.mInvMass) \* (this.mWidth \* this.mWidth + this.mHeight \* this.mHeight) / 12;

this.mInertia = 1 / this.mInertia;

}

};

1. Update the Rectangle constructor in a similar manner to the Circle class to accept the relevant parameters of physical attributes and to invoke the newly defined shape specific updateIntertia function.

var Rectangle = function (center, width, height, mass, friction, restitution) {

RigidShape.call(this, center, mass, friction, restitution);

this.mType = "Rectangle";

this.mWidth = width;

this.mHeight = height;

//...indetical to previous project

this.updateInertia();

};

### Observations

You Run the project to test your implementation. Create a few objects in the scene; you can examine the attributes of your selected object. Notice that when you enable the movement by pressing the comma ( , ) key, the objects with higher downward initial velocity will drop faster because of the gravitational force or acceleration. Now create an object and set its initial y-velocity to negative. Observe that the object will move upwards until the y-component velocity reaches zero, and then it will start to fall downwards as a result of gravitational acceleration. You can also change the object’s initial x-velocity and observe the motion of a projectile. Another interesting case to try is to create a few objects and excite them by pressing the ‘H’ key. Observe how all the objects move according to their own velocities. You may see objects that move beyond the scene boundary. This is because at this point the physics engine does not support collision resolution. This will be remedied in the next section.