CHAPTER 9

Integrating Physics and Particles

After completing this chapter, you will be able to:

- Understand how to approximate integrals with Euler Method and Symplectic Euler Integration
- Approximate Newtonian motion formulation with Symplectic Euler Integration
- Understand the needs for and implement collision detections of bounding rectangles and circles
- Resolve interpenetrating collisions based on a numerically stable relaxation method
- Understand, implement, and work with particle systems

Introduction

In the previous chapter, you experienced building the illumination component by simulating the propagation of light energy in a game scene. Recall that only selected objects participated in the simulation. For instance, in a scene, only `IllumRenderable` objects can be illuminated by the light sources, while others such as `SpriteRenderable` objects are not. In a similar fashion, the physics component you will learn about in this chapter simulates the transfer of energy between selected objects. Just as illumination, the physics component of a game engine is also a large and complex area of game engine design, architecture, and implementation. With this in mind, you will develop the physics component with the same approach you used for all the previous game engine components you have implemented. That is, by focusing on the core functionality of the component, you increase the game engine's ability to create a diverse set of 2D games.

Note In games, the functionality of simulating energy transfer is often referred to as *physics*, *physics system*, or *physics component*. This convention is followed in the book.

Physics in 2D games can be seen in a wide variety of genres from simple Pong-like reaction-based games to more advanced Angry Birds–style block destruction games. Both of these games and everything in between (such as platformers) rely on specific physical behavior to implement their core gameplay. You will implement a physics system that can support a large variety of gameplay based on an architecture that supports expansion.

Particles are textured images displayed at positions with no defined dimensions that often participate in the physics simulation. The textured images facilitate representation of interesting effects while positions without dimensions allow simple and efficient physics simulations. A particle system controls and strategizes the generation of particles with specific appearance and behavior. A particle system provides a game engine with the ability to implement dynamic-looking reactions. This includes common effects such as explosions and effects associated with casting spells. Building upon the physics component, you will create a particle system that can support diverse gameplay requirements.

Physics Overview

Physics simulations in games involve three core concepts: movement, collision detection, and collision resolution. The proper implementation based on these concepts enables believable scenarios where objects physically interact with each other in the game world. To satisfy the real-time interaction requirement, the energy transfer simulations are approximated with optimization compromises.

Movement

Movement is the description of how object positions are maneuvered around the game world. Mathematically, movement can be formulated in many ways. In Chapter 6, you experienced working with movement where you continuously accumulated a velocity to an object's position. Although desired results can be achieved, mathematically this is problematic because a velocity and a position are different types of quantities and strictly speaking cannot. be combined. In practice, you have been working with describing movement based on displacements.

- $p_{new} = p_{current} + displacement$

Movements governed by the displacement formulation become restrictive when it is necessary to change the quantity being displaced over time. Newtonian mechanics address this restriction by considering time in movement formulations.

- $v_{new} = v_{current} + \int a(t)\,dt$
- $p_{new} = p_{current} + \int \mathrm{v}(\mathrm{t})\,\mathrm{d}t$

where v(t) is the velocity that describes the change of position over time and a(t) is the acceleration that describes the change of 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 the acceleration is occurring in. Acceleration is changed by the forces acting upon the object. For example, if you were to throw a ball into the air, the gravitational force on the earth would affect the object's acceleration over time, which in turn would change the object's velocity.

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. 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.

- $v_{new} = v_{current} + a_{current} * dt$
- $p_{new} = p_{current} + v_{current} * dt$

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.

- $v_{new} = v_{current} + a_{current} * dt$
- $p_{new} = p_{current} + \boldsymbol{v}_{\boldsymbol{new}} * dt$

Recall that the game loop you have created has a fixed time step, where the `update()` function is called 60 times per second consistently. In typical implementations, the Symplectic Euler Integration method is invoked once for each `update()` function call, and the *dt* quantity can simply be the fixed time step. This means that the integration approximation occurs once during each time interval. In this way, an object in continuous motion is approximated by a discrete set of positions. Figure 9-1 shows a moving object being approximated by the object at three different time steps. However, in a continuous motion, the approximation places the object in only one of these three positions. This is a consequence of the discrete time step nature of a game engine. The most notable ramifications of this approximation are in detecting collisions. You can see one such problem in Figure 9-1; 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.

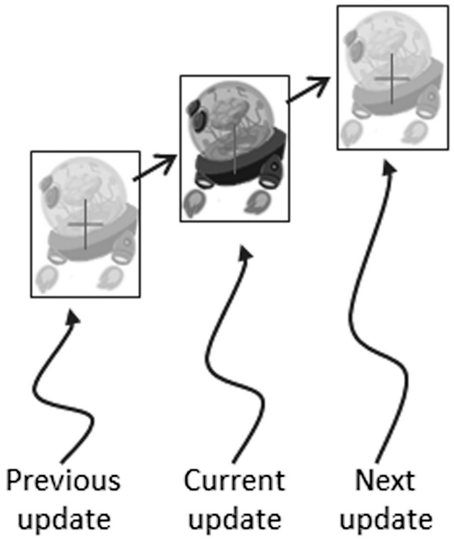

Figure 9-1. *Approximating object position with Symplectic Euler Integration over three distinct time steps*

Collision Detection

Collision detection is a vital and potentially a costly piece of physics simulation that can impact performance. For example, if you want to detect the collisions between five enemies, in the worst case you must perform four detection computations for the first enemy, followed by three computations for the second, two for the third, and one for the forth. In general, without dedicated optimizations, in the worst case you must perform $O(N^2)$ operations to detect the collisions between N objects.

■ **Note** Optimizations for collision detection typically exploit the proximity of game objects in the scene by organizing these objects either with a spatial structure such as uniform grid or quad-tree or into coherent groups such as hierarchies of bounding colliders. To focus on the core concepts and avoid unnecessary complications, these optimizations will not be covered.

You have experienced computing the collisions between objects from Chapters 6 with the `BoundingBox` utility class and the per-pixel collision detection algorithm. Although those methods can successfully detect the situation when objects overlap in space, they are not capable of computing the details of the collision. The details of a collision become important when it is necessary to resolve undesirable collision results.

Figure 9-2 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 numerical integration place the two objects over each other. This is another example ramification of the `update()` function being called at discrete time 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.

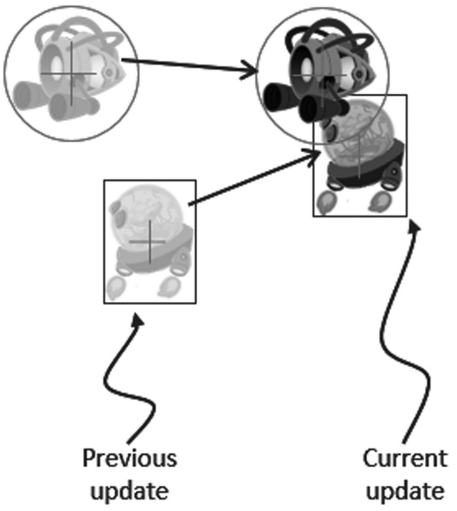

Figure 9-2. *Interpenetration of objects after an update*

Collision Resolution

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 situation. Notice that there are no collision resolution processes in the real world where collisions are strictly governed by the law of physics and interpenetration of rigid objects would never occur. Resolutions of collisions are relevant only in the virtual game world, where movements are approximated and impossible conditions may occur but can be resolved in ways that are desirable to the 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 the object's position so that it is no longer penetrating 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 interpenetration can result in new penetrations with other close 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 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 relaxation where interpenetrated objects are separated incrementally via repeated applications of the Projection Method in a single update cycle. With relaxation, the number of times that the Projection Method is applied is referred to as *relaxation iterations*. For example, by default the engine sets relaxation iterations to 15. This means that within one `update()` function call, after the movement integration approximation, the collision detection andk resolution procedures will be executed 15 times.

■ **Note** The Impulse Method was popularized and championed by Erin Catto with the popular Box2D physics game engine component. You can view the source code at `http://box2d.org/`.

Detecting Collisions

You have already worked with axis-aligned bounding boxes in Chapter 6. For flexibility and to better bound objects of different shapes, you will also support bounding circles in your game engine. The collision between game objects will be determined based on the collision results between bounding boxes and circles. You can position a bounding shape or shapes on your object regardless of the object's center position, and you can use multiple bounding shapes for detecting collisions with the object. Focusing on the core concepts and clarity of implementation, you will implement the straightforward system with one bounding shape centered on each game object. You can expand and improve this basic system after mastering the concepts involved.

■ **Note** This book covers the collision detection only between axis-aligned rectangles and circles. There are algorithms, such as the popular Separating Axis Theorem (SAT) and the Gilbert-Johnson-Keerthi (GJK), that are capable of detecting the collisions between any general convex shapes in 2D. These algorithms are computationally intensive and typically use simple bounding colliders such as bounding boxes or circles as a first pass.

The Rigid Shape Bounds Project

This project demonstrates how to detect collisions between bounding boxes and circles. For debugging purposes, the new bounding shapes can be drawn with the `RenderableLine` class. You can see an example of this project running in Figure 9-3. The source code to this project is defined in the `Chapter9/9.1.RigidShapeBounds` folder.

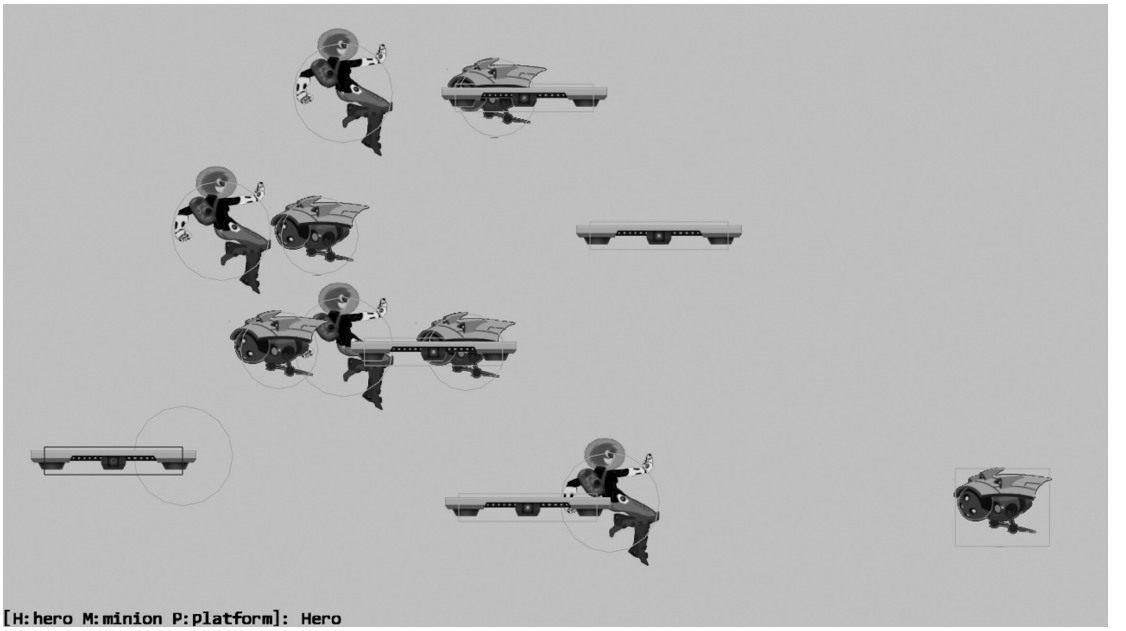

Figure 9-3. *Running the Rigid Shape Bounds project*

The controls of the project are as follows:

- *H/M/P keys*: Select one of `Hero`, `Minion`, or `Platform`. The selected object will not be drawn; instead, only its bounds will be drawn for better visual inspection of collision detection results.
- *Left mouse button press and move in the game window*: Drags the selected object (with only the bounding shape drawn) for collision detection. A detected collision causes the bounding shapes to be drawn in red.

The goals of the project are as follows:

- To derive the basic infrastructure to support collision detection between rigid shapes
- To understand and implement collision detection algorithms between bounding boxes and circles
- To lay the foundation for building a physics component

You can find the following external resources in the `assets` folder: the `fonts` folder that contains the default system fonts and two texture images (`minion_sprite.png`, which defines the sprite elements for the heroes and the minions, and `platform.png`, which defines the platforms).

Creating the Rigid Shape Base Class

You will begin by defining a base class for the bounding rectangle and circle. The base class will define all functionality that is common to the two bounding shapes. Note that a new class, `RigidRectangle`, is created for detecting collisions with bounding circles. Although similar, the existing functionality of the `BoundingBox` class is left unchanged to avoid affecting the rest of the engine that currently utilizes the `BoundingBox` class.

1. Start by creating a new subfolder called `Physics` under the `src/Engine` folder. In the `Physics` folder, create a new file called `RigidShape.js`.

2. Edit `RigidShape.js` and define the `eRigidType` enumerated data type to distinguish between the different types of rigid shapes.

```
RigidShape.eRigidType = Object.freeze({
    eRigidAbstract: 0,
    eRigidCircle: 1,
    eRigidRectangle: 2
});
```

3. Define the constructor for the rigid shape, which takes the transform and contains a `LineRenderable` object for drawing the center position of the rigid shape. The `mDrawBounds` variable defines whether the bounds should be drawn.

```
function RigidShape(xform) {
    this.mXform = xform; // this is typically from gameObject
    this.kPadding = 0.25   // size of the position mark
    this.mPositionMark = new LineRenderable();
    this.mDrawBounds = false;
}
```

4. Create a `draw()` function to draw an X at the center position of the rigid shape by utilizing the transform and the constant padding created in the constructor. Notice that the same `LineRenderable` is used for both pieces of the X to save memory. This may seem like an extraneous detail; however, when drawing many bounding boxes and circles, the savings can quickly add up.

```
RigidShape.prototype.draw = function (aCamera) {
    if (!this.mDrawBounds) {
        return;
    }
```

```
        //calculation for the X at the center of the shape
        var x = this.mXform.getXPos();
        var y = this.mXform.getYPos();

        this.mPositionMark.setFirstVertex(x - this.kPadding, y + this.kPadding);
                                                                    //TOP LEFT
        this.mPositionMark.setSecondVertex(x + this.kPadding, y - this.kPadding);
                                                                //BOTTOM RIGHT
        this.mPositionMark.draw(aCamera);

        this.mPositionMark.setFirstVertex(x + this.kPadding, y + this.kPadding);
                                                                    //TOP RIGHT
        this.mPositionMark.setSecondVertex(x - this.kPadding, y - this.kPadding);
                                                                //BOTTOM LEFT
        this.mPositionMark.draw(aCamera);
    };
```

5. Create an empty `update()` function for subclasses to override and add get and set accessors.

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

RigidShape.prototype.rigidType = function () {
    return RigidShape.eRigidType.eRigidAbstract;
};

RigidShape.prototype.getPosition = function() {
    return this.mXform.getPosition();
};
RigidShape.prototype.setPosition = function(x, y ) {
    this.mXform.setPosition(x, y);
};
RigidShape.prototype.getXform = function () { return this.mXform; };
RigidShape.prototype.setXform = function (xform) { this.mXform = xform; };
RigidShape.prototype.setColor = function (color) { this.mPositionMark.
setColor(color); };
RigidShape.prototype.getColor = function () { return this.mPositionMark.
getColor(); };
RigidShape.prototype.setDrawBounds = function(d) { this.mDrawBounds = d; };
RigidShape.prototype.getDrawBounds = function() { return this.mDrawBounds; };
```

Creating the Rigid Rectangle Object

With the base abstract class for rigid shapes defined, you can now create your first rigid object, the rigid rectangle.

1. Under the `Physics` folder, create a new file called `RigidRectangle.js`.
2. Edit this file to create a constructor that initializes a width property and a height property and a `LineRenderable` object for drawing the sides of the rectangle. Make sure to inherit from the `RigidShape` base class.

```
function RigidRectangle(xform, w, h) {
    RigidShape.call(this, xform);
    this.mSides = new LineRenderable();
```

```
    this.mWidth = w;
    this.mHeight = h;
}
gEngine.Core.inheritPrototype(RigidRectangle, RigidShape);
```

3. Override the `draw()` function to draw the bounds of the rectangle when `mDrawBounds` is true. Remember to call the superclass `draw()` function to draw the center position of the shape.

```
RigidRectangle.prototype.draw = function (aCamera) {
    if (!this.mDrawBounds) {
        return;
    }
    RigidShape.prototype.draw.call(this, aCamera);
    var x = this.getPosition()[0];
    var y = this.getPosition()[1];
    var w = this.mWidth/2;
    var h = this.mHeight/2;

    this.mSides.setFirstVertex(x - w, y + h);   //TOP LEFT
    this.mSides.setSecondVertex(x + w, y + h); //TOP RIGHT
    this.mSides.draw(aCamera);
    this.mSides.setFirstVertex(x + w, y - h);   //BOTTOM RIGHT
    this.mSides.draw(aCamera);
    this.mSides.setSecondVertex(x - w, y - h); //BOTTOM LEFT
    this.mSides.draw(aCamera);
    this.mSides.setFirstVertex(x - w, y + h);   //TOP LEFT
    this.mSides.draw(aCamera);
};
```

4. Define the get and set accessors and override the `rigidType()` function to return the appropriate value.

```
RigidRectangle.prototype.rigidType = function () {
    return RigidShape.eRigidType.eRigidRectangle;
};
RigidRectangle.prototype.getWidth = function () { return this.mWidth; };
RigidRectangle.prototype.getHeight = function () { return this.mHeight; };
RigidRectangle.prototype.setColor = function (color) {
    RigidShape.prototype.setColor.call(this, color);
    this.mSides.setColor(color);
};
```

Creating the Rigid Circle Object

You can now implement the rigid circle object based on a similar overall structure.

1. Under the `Physics` folder, create a new file called `RigidCircle.js`.
2. Edit this file to create a constructor that initializes a radius property and create a `LineRenderable` for drawing the bounds of the circle. This class must also inherit from the `RigidShape` base class.

```
function RigidCircle(xform, r) {
    RigidShape.call(this, xform);
    this.kNumSides = 16;
    this.mSides = new LineRenderable();
    this.mRadius = r;
}
gEngine.Core.inheritPrototype(RigidCircle, RigidShape);
```

3. Override the `draw()` function to draw the circumference of the circle when `mDrawBounds` is true. Take note that the circumference of the circle is drawn as 16 individual line segments.

```
RigidCircle.prototype.draw = function (aCamera) {
    if (!this.mDrawBounds) {
        return;
    }
    RigidShape.prototype.draw.call(this, aCamera);

    // kNumSides forms the circle.
    var pos = this.getPosition();
    var prevPoint = vec2.clone(pos);
    var deltaTheta = (Math.PI * 2.0) / this.kNumSides;
    var theta = deltaTheta;
    prevPoint[0] += this.mRadius;
    var i, x, y;
    for (i = 1; i <= this.kNumSides; i++) {
        x = pos[0] + this.mRadius * Math.cos(theta);
        y = pos[1] +  this.mRadius * Math.sin(theta);

        this.mSides.setFirstVertex(prevPoint[0], prevPoint[1]);
        this.mSides.setSecondVertex(x, y);
        this.mSides.draw(aCamera);

        theta = theta + deltaTheta;
        prevPoint[0] = x;
        prevPoint[1] = y;
    }
};
```

4. Define the get and set accessors and override the `rigidType()` function to return the appropriate value.

```
RigidCircle.prototype.rigidType = function () {
    return RigidShape.eRigidType.eRigidCircle;
};
RigidCircle.prototype.getRadius = function () {
    return this.mRadius;
};
RigidCircle.prototype.setColor = function (color) {
    RigidShape.prototype.setColor.call(this, color);
    this.mSides.setColor(color);
};
```

Detecting Collisions Between the Rigid Objects

Thus far you have implemented simple shapes that can be drawn on the canvas. You can now add collision detection behaviors to these shapes to determine whether they are colliding. You need to define functionality to detect collisions between two rectangles, between two circles, and between a rectangle and a circle. You will also add functions for determining whether a position is within the bounds of a shape.

Collision Detection Between a Rectangle and a Circle

Follow these steps:

1. In the `Physics` folder, create a new file called `RigidShape_Collision.js`. This file defines collision-related functions of the `RigidShape` class. The separate source code files ensure readability.

2. Create the `collidedRectCirc()` function to compute the collision between a rectangle and a circle. As illustrated in Figure 9-4, there are three distinct cases when a rectangle and a circle collide. The left and middle sketches of Figure 9-4 illustrate the first two cases where either the center of the rectangle is within the bounds of the circle or the center of circle is within the bounds of the rectangle. The third case on the right of Figure 9-4 illustrates two colliding shapes with centers outside the bounds of the other shape.

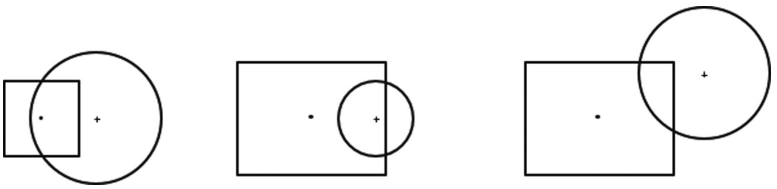

Figure 9-4. *The three cases of rectangle and circle interpenetration*

You can see this logic reflected in the following code where the first two simple cases are tested in the beginning of the function. For the third and more general case, Figure 9-5 uses two circles with centers at position V_{2a} and V_{2b} that interpenetrate the rectangle with a center at V_1 to illustrate two cases of this computation. Circle V_{2a} collides the rectangle from the top side, and V_{2b} collides from the right side. In both cases, the `vFrom1to2` vector connects the corresponding centers, and the size of the `normal` vector determines the collision condition: the collision is true when the length of the `normal` vector is less than the corresponding circle radius. The `normal` vector can be computed by subtracting `vec` from `vFrom1to2`. The `vec` vector is the result of clamping the corresponding component of the `vFrom1to2` vector by the collision side of the rectangle. For example, in the case of V_{2a} the y-component of the $\texttt{vFrom1to2}_a$ vector is clamped by the y-value of the top side of the rectangle to form $\texttt{vec}_a$. Similarly, the x-component of the $\texttt{vFrom1to2}_b$ vector is clamped by the x-value of the right side of the rectangle to form $\texttt{vec}_b$.

```
RigidShape.prototype.collidedRectCirc = function(rect1Shape, circ2Shape) {
    var rect1Pos = rect1Shape.getPosition();
    var circ2Pos = circ2Shape.getPosition();

    if (rect1Shape.containsPos(circ2Pos) || (circ2Shape.containsPos(rect1Pos))) {
        return true;
    }
```

```
    var vFrom1to2 = [0, 0];
    vec2.subtract(vFrom1to2, circ2Pos, rect1Pos);
    var vec = vec2.clone(vFrom1to2);

    var alongX = rect1Shape.getWidth() / 2;
    var alongY = rect1Shape.getHeight() / 2;

    vec[0] = this.clamp(vec[0], -alongX, alongX);
    vec[1] = this.clamp(vec[1], -alongY, alongY);

    var normal = [0, 0];
    vec2.subtract(normal, vFrom1to2, vec);

    var distSqr = vec2.squaredLength(normal);
    var rSqr = circ2Shape.getRadius() * circ2Shape.getRadius();

    return (distSqr < rSqr);
};
```

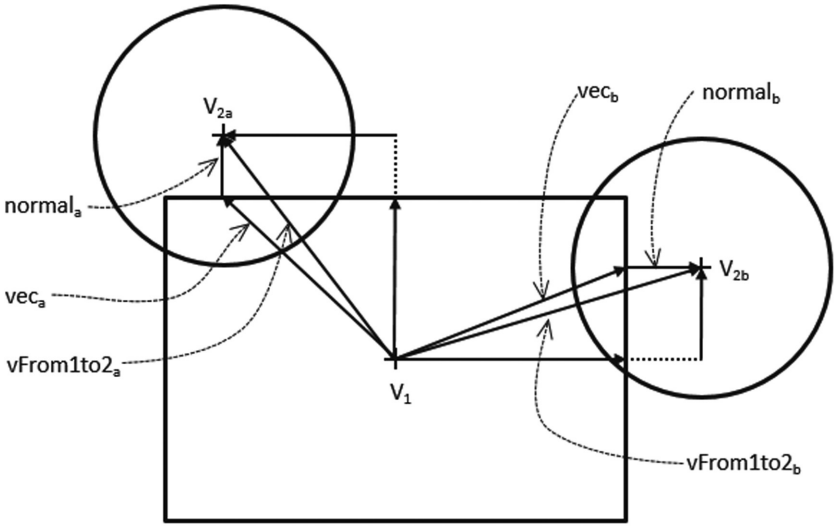

Figure 9-5. *Two examples of general case of rectangle/circle collision*

3. Define the `clamp()` function to clamp a value between minimum and maximum values.

```
RigidShape.prototype.clamp = function (value, min, max) {
    return Math.min(Math.max(value, min), max);
};
```

Collision Detection Between Two Rectangles

Follow these steps:

1. In the Physics folder, create a new file, RigidRectangle_Collision.js, to define the collision-related functions for the RigidRectangle object.

2. Create the collidedRectRect() function to receive two RigidRectangle objects and return whether the two have collided. Since the rectangles are axis-aligned, the collision condition can be determined by X and Y bounds check, similar to the boundCollideStatus() function of the BoundingBox class.

```
RigidRectangle.prototype.collidedRectRect = function(r1, r2) {
    var r1Pos = r1.getPosition();
    var r1MinX = r1Pos[0] - r1.getWidth() / 2;
    var r1MaxX = r1Pos[0] + r1.getWidth() / 2;
    var r1MinY = r1Pos[1] - r1.getHeight() / 2;
    var r1MaxY = r1Pos[1] + r1.getHeight() / 2;

    var r2Pos = r2.getPosition();
    var r2MinX = r2Pos[0] - r2.getWidth() / 2;
    var r2MaxX = r2Pos[0] + r2.getWidth() / 2;
    var r2MinY = r2Pos[1] - r2.getHeight() / 2;
    var r2MaxY = r2Pos[1] + r2.getHeight() / 2;

    return ((r1MaxX > r2MinX) && (r1MinX < r2MaxX) &&
            (r1MaxY > r2MinY) && (r1MinY < r2MaxY));
};
```

3. Define a collided() function for the RigidRectangle class to detect collisions between two arbitrary rigid shapes.

```
RigidRectangle.prototype.collided = function(otherShape) {
    var status = false;
    switch (otherShape.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = this.collidedRectCirc(this, otherShape);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = this.collidedRectRect(otherShape, this);
            break;
    }
    return status;
};
```

4. Create the containsPos() function to determine whether a given position is within the bounds of the rectangle.

```
RigidRectangle.prototype.containsPos = function (pos) {
    var rPos = this.getPosition();
    var rMinX = rPos[0] - this.getWidth() / 2;
    var rMaxX = rPos[0] + this.getWidth() / 2;
```

```
    var rMinY = rPos[1] - this.getHeight() / 2;
    var rMaxY = rPos[1] + this.getHeight() / 2;

    return ((rMinX < pos[0]) && (rMaxX > pos[0]) &&
            (rMinY < pos[1] && rMaxY > pos[1]));
};
```

Collision Detection Between Two Circles

Follow these steps:

1. In the Physics folder, create a new file, RigidCircle_Collision.js, to define the collision-related functions for the RigidCircle object.

2. Create the collidedCircCirc() function to receive two RigidCircle objects and return whether the two have collided. The collision condition is true when the distance between the two circle centers is less than the sum of their radii. To avoid the expensive square root function when computing a vector magnitude, the squared values are compared instead.

```
RigidCircle.prototype.collidedCircCirc = function(c1, c2) {
    var vecToCenter = [0, 0];
    vec2.sub(vecToCenter, c1.getPosition(), c2.getPosition());
    var rSum = c1.getRadius() + c2.getRadius();
    return (vec2.squaredLength(vecToCenter) < (rSum * rSum));
};
```

3. As in the case for RigidRectangle, define a collide() function to support collisions with an arbitrary rigid shape.

```
RigidCircle.prototype.collided = function(otherShape) {
    var status = false;
    switch (otherShape.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = this.collidedCircCirc(this, otherShape);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = this.collidedRectCirc(otherShape, this);
            break;
    }
    return status;
};
```

4. Create the containsPos() function to determine whether a given position is within the bounds of the circle.

```
RigidCircle.prototype.containsPos = function(pos) {
    var dist = vec2.distance(this.getPosition(), pos);
    return (dist < this.getRadius());
};
```

Modifying the Game Object

You can now modify the GameObject class to begin building the support for the eventual physics component of the game engine.

1. Edit GameObject.js and declare a new variable: mPhysicsComponent in the constructor.

```
function GameObject(renderableObj) {
    this.mRenderComponent = renderableObj;
    this.mVisible = true;
    this.mCurrentFrontDir = vec2.fromValues(0, 1);  // the current front direction
    this.mSpeed = 0;
    this.mPhysicsComponent = null;
}
```

2. Define simple get and set accessors for the physics component.

```
GameObject.prototype.setPhysicsComponent = function (p)
                                           { this.mPhysicsComponent = p; };
GameObject.prototype.getPhysicsComponent = function ()
                                           { return this.mPhysicsComponent; };
```

3. Modify the update() and draw() functions to support the physics component if it exists.

```
GameObject.prototype.update = function () {
    // simple default behavior
    var pos = this.getXform().getPosition();
    vec2.scaleAndAdd(pos, pos, this.getCurrentFrontDir(), this.getSpeed());

    if (this.mPhysicsComponent !== null) {
        this.mPhysicsComponent.update();
    }
};

GameObject.prototype.draw = function (aCamera) {
    if (this.isVisible()) {
        this.mRenderComponent.draw(aCamera);
    }
    if (this.mPhysicsComponent !== null) {
        this.mPhysicsComponent.draw(aCamera);
    }
};
```

Testing the Collision Detection

It is important to create objects with both rectangle and circle bounds to verify the correctness of collision detection functionality.

Modifying the Game Objects

Modify the Hero and Minion objects to define rigid objects as their physics component.

1. Modify the Hero.js file to define a RigidCircle to be its physics component.

```
function Hero(spriteTexture, atX, atY) {
    // ... identical code to previous project

    var r = new RigidCircle(this.getXform(), 9);
    r.setColor([0, 1, 0, 1]);
    r.setDrawBounds(true);
    this.setPhysicsComponent(r);
}
```

2. Edit the Minion.js file to define either a RigidCircle or a RigidRectangle to be its physics component.

```
function Minion(spriteTexture, atX, atY) {
    // ... identical code to previous project

    var r;
    if (Math.random() > 0.5) {
        r = new RigidCircle(this.getXform(), 7);
    } else {
        r = new RigidRectangle(this.getXform(), 17, 14);
    }
    r.setColor([0, 1, 0, 1]);
    r.setDrawBounds(true);
    this.setPhysicsComponent(r);
}
```

3. Create a new GameObject type, Platform, with a RigidRectangle defined as its physics component. Because of the similarity with the previous code, the details are not shown here.

Modifying the MyGame Object

To properly test and observe the results of your collision detection implementation, begin with a new MyGame.js file and implement the usual necessary functions of the constructor: initialize(), update(), and draw(). Please refer to the MyGame.js file for the details of the implementation. The interesting elements of this example are in the _detectCollision() function. This function is called by the update() function to compute the collision between the selected object against all other objects in the scene. The colliding objects will have their corresponding RigidShape objects drawn in red.

```
MyGame.prototype._detectCollision = function () {
    var i, obj;
    this.mCollidedObj = null;
    var selectedRigidShape = this.mSelectedObj.getPhysicsComponent();
```

```
        for (i = 0; i<this.mAllObjects.size(); i++) {
            obj = this.mAllObjects.getObjectAt(i);
            if (obj !== this.mSelectedObj) {
                if (selectedRigidShape.collided(obj.getPhysicsComponent())) {
                    this.mCollidedObj = obj;
                    this.mCollidedObj.getPhysicsComponent().setColor(this.kCollideColor);
                } else {
                    obj.getPhysicsComponent().setColor(this.kNormalColor);
                }
            }
        }
};
```

Observations

Run the project to test your implementation. Notice that the texture for the selected object is not drawn. This is to ensure that you can clearly observe and verify that the edges of colliding bounds are touching when the bound color changes to red. Cycle through the H, M, and P keys to observe the same three objects of the many instances are always selected. Now you can move the mouse with a left button click to move the bounds of the selected object. Notice that you can simultaneously collide with the bounds of multiple objects. Lastly, take note that the bounding shapes may not completely enclose the `GameObject` textures, as in the case of the `Hero` object where part of Dye's head and leg are outside of the bounding circle. How tightly a bounding shape should enclose an object should be determined by the gameplay requirement and not by the geometries.

Resolving Collisions

With a functioning collision detection system, you can now begin implementing collision resolution and support more natural and realistic behaviors. Focusing on the core concepts and building a stable system, you will continue to work with axis-aligned rigid rectangles and circles and avoid the complications involved in computing angular impulse resolutions and detecting collisions between rotated rectangles. For this reason, in your implementation, objects will not rotate as a response to collisions. However, the concepts and implementation approach described in this section can be generalized to support object rotation collision responses.

■ **Note** Interested readers can refer to the following links on how rotation for impulse resolution can be supported: `https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/` or `http://gamedevelopment.tutsplus.com/series/how-to-create-a-custom-physics-engine--gamedev-12715` .

The Rigid Shape Impulse Project

This project demonstrates how to combine and implement projection and impulse-based collision resolution for collided rigid rectangles and circles. Additionally, this project implements the Symplectic Euler Integration for object movements. You can see an example of this project running in Figure 9-6. The source code of this project is located in the `Chapter9/9.2.RigidShapeImpulse` folder.

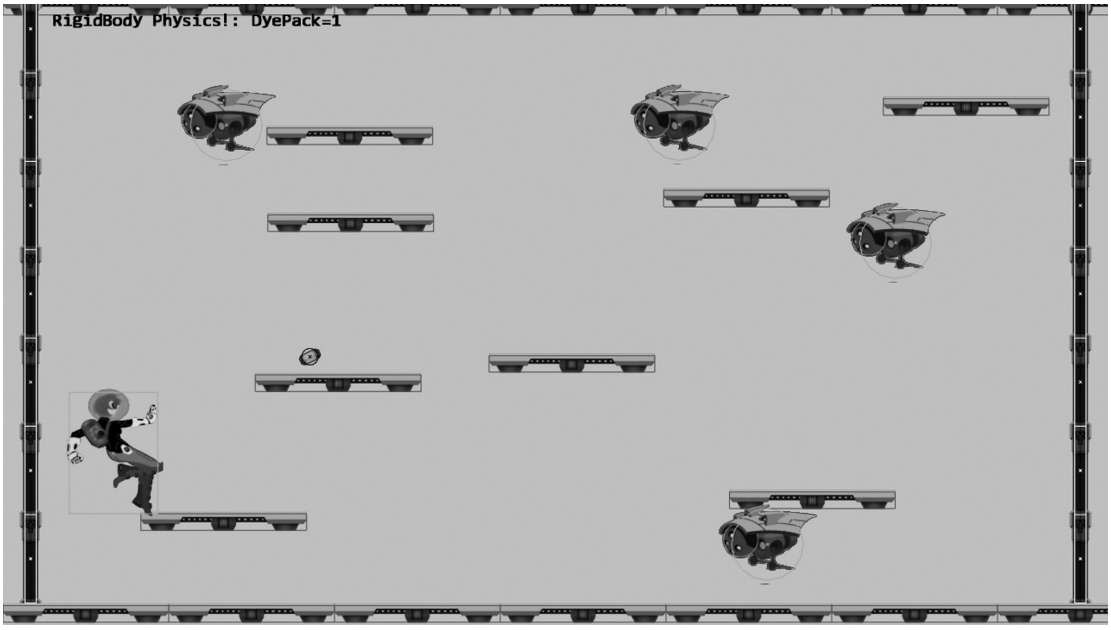

Figure 9-6. *Running the Rigid Shape Impulse project*

The controls of the project are as follows:

- *WASD keys*: Moves the `Hero` object
- *Left mouse button click in the game window*: Sends a `DyePack` chasing after the `Hero`
- *For testing the stability of collision resolution*:
  - *Z key*: Creates dumb minions with white bounds under the current mouse pointer position
  - *X key*: Selects the dumb minion under the current mouse pointer position; moving the mouse with the X key pressed will move the selected dumb minion
  - *C key*: Agitates all dumb minions with a random velocity; press and hold the C key to increase the agitation
  - *V key*: Clears all dumb minions

The goals of the project are as follows:

- To experience implementing movements based on Symplectic Euler Integration
- To understand the need for and experience the implementation of collecting collision information
- To build a physics component with relaxation support

You can find the following external resources in the `assets` folder: the `fonts` folder that contains the default system fonts and four texture images (`minion_sprite.png`, which defines the sprite elements for the hero and the minions; `platform.png`, which defines the platforms and floor and ceiling tiles; `wall.png`, which defines the walls; and `dye_pack.png`, which defines the dye packs).

Implementing Symplectic Euler Integration

For objects to collide and the resulting collisions resolved, their movements must be properly simulated. Recall the discussion from the beginning of this chapter that such movements can be described by the Newtonian motion equations, and the solutions can be approximated by the Symplectic Euler Integration.

- $v_{new} = v_{current} + a_{current} * dt$
- $p_{new} = p_{current} + v_{new} * dt$

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. You will modify the `RigidShape` class for this implementation.

Modifying the RigidShape Class

Modify the `RigidShape` class to include variables for mass, restitution (bounciness), velocity, friction, and acceleration, as shown in the following code. Notice that the inversed of the mass value is actually stored for computation efficiency.

```
function RigidShape(xform) {
    this.mXform = xform; // this is typically from gameObject
    this.kPadding = 0.25; // size of the position mark
    this.mPositionMark = new LineRenderable();
    this.mDrawBounds = false;

    // physical properties
    this.mInvMass = 1;
    this.mRestitution = 0.8;
    this.mVelocity = vec2.fromValues(0, 0);
    this.mFriction = 0.3;
    this.mAcceleration = gEngine.Physics.getSystemAcceleration();
}
```

Creating the Rigid Shape Behavior

You can now add the behavior to the rigid shape object for numerical integration.

1. Under the `src/Engine/Physics` folder, add a new file and name it `RigidShapeBehavior.js`. Remember to load this new source file in `index.html`.
2. Define an `update()` function to apply Symplectic Euler Integration to the rigid shape where the updated velocity is used for computing the new position. Notice that the use of the inverse mass in computing the velocity is a deviation from classical physics. This is a convenient trick that affords you the ability to switch off an object's motion by initializing both the velocity and inverse mass to zero.

```
RigidShape.prototype.update = function () {
    var dt = gEngine.GameLoop.getUpdateIntervalInSeconds();

    // Symplectic Euler
    //    v += (1/m * a) * dt
    //    x += v * dt
    var v = this.getVelocity();
    vec2.scaleAndAdd(v, v, this.mAcceleration, (this.getInvMass() * dt ));

    var pos = this.getPosition();
    vec2.scaleAndAdd(pos, pos, v, dt);
};
```

3. Create get and set accessors for each of the newly added variables. Note that the `setMass()` function actually computes the inverse value or initializes `mInvMass` to zero.

```
RigidShape.prototype.getInvMass = function () { return this.mInvMass; };
RigidShape.prototype.setMass = function (m) {
    if(m > 0) {
        this.mInvMass = 1/m;
    } else {
        this.mInvMass = 0;
    }
};
RigidShape.prototype.getVelocity = function () { return this.mVelocity; };
RigidShape.prototype.setVelocity = function (v) { this.mVelocity = v; };
RigidShape.prototype.getRestitution = function () { return this.mRestitution; };
RigidShape.prototype.setRestitution = function (r) { this.mRestitution = r; };
RigidShape.prototype.getFriction = function () { return this.mFriction; };
RigidShape.prototype.setFriction = function (f) { this.mFriction = f; };
RigidShape.prototype.getAcceleration = function () { return this.mAcceleration; };
RigidShape.prototype.setAcceleration = function (g) { this.mAcceleration = g; };
```

Extracting Collision Information

To support proper resolution, at each collision you need to compute a collision depth and the corresponding collision normal. 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. For example, in Figure 9-5 the distance defined by subtracting the length of the `normal` vector from the circle radius is the collision depth, and the `normal` vector is the collision normal because the collision depth is measured along this direction. It is always the case that any interpenetration can be resolved by moving the objects along the collision normal by collision depth distance. The `RigidShape` collision functions must be modified to compute for this information.

Creating the CollisionInfo object

For convenience, create a new object for recording the collision information.

1. Under the `src/Engine/Utils` folder, add a new file and name it `CollisionInfo.js`. Remember to load this new source file in `index.html`.

2. Define the constructor and the accessors for the collision depth and normal.

```
function CollisionInfo() {
    this.mDepth = 0;
    this.mNormal = vec2.fromValues(0, 0);
}
CollisionInfo.prototype.setDepth = function (s) { this.mDepth = s; };
CollisionInfo.prototype.setNormal = function (s) { this.mNormal = s; };

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

Modifying the RigidShape Collision

You can now edit the `RigidShape_Collision.js` file to extract the collision information when a rectangle and circle collide. Figure 9-5 illustrates the general strategy of computing the collision normal, `normal`, and collision depth (subtracting the length of `normal` from the radius). As discussed previously, the `normal` vector is derived from `vec`, the result of clamping the components of the `vFrom1to2` vector by the colliding side of the rectangle. As illustrated in Figure 9-7, the only slight complication is when the circle center is inside the bounds of the rectangle, where instead of clamping, you must extend corresponding `vFrom1to2` components to compute the `vec` vector. For example, in Figure 9-7, the x-component of `vFrom1to2` must be extended to reach the x-value of the rectangle right boundary to form the `vec` vector. Notice that in all cases, the computed `normal` is a vector from the collision rectangle bound toward the circle center. For this reason, in the case of Figure 9-7, the `normal` vector must be reversed to represent collision normal.

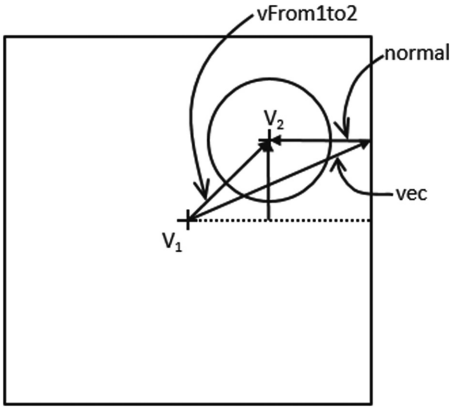

Figure 9-7. *Circle center in the bounds of the colliding rectangle*

```
RigidShape.prototype.collidedRectCirc = function(rect1Shape, circ2Shape, collisionInfo) {
    var rect1Pos = rect1Shape.getXform().getPosition();
    var circ2Pos = circ2Shape.getXform().getPosition();
    var vFrom1to2 = [0, 0];
    vec2.subtract(vFrom1to2, circ2Pos, rect1Pos);

    var vec = vec2.clone(vFrom1to2);

    var alongX = rect1Shape.getWidth() / 2;
    var alongY = rect1Shape.getHeight() / 2;

    vec[0] = this.clamp(vec[0], -alongX, alongX);
    vec[1] = this.clamp(vec[1], -alongY, alongY);

    var isInside = false;
    if (rect1Shape.containsPos(circ2Pos))  {
        isInside = true;
        // Find closest axis
        if (Math.abs(vFrom1to2[0] - alongX) < Math.abs(vFrom1to2[1] - alongY)) {
            // Clamp to closest side
            if (vec[0] > 0) {
                vec[0] = alongX;
            } else {
                vec[0] = -alongX;
            }
        } else { // y axis is shorter
            // Clamp to closest side
            if (vec[1] > 0) {
                vec[1] = alongY;
            } else {
                vec[1] = -alongY;
            }
        }
    }

    var normal = [0, 0];
    vec2.subtract(normal, vFrom1to2, vec);

    var distSqr = vec2.squaredLength(normal);
    var rSqr = circ2Shape.getRadius() * circ2Shape.getRadius();

    if (distSqr > rSqr && !isInside) {
        return false; //no collision exit before costly square root
    }

    var len = Math.sqrt(distSqr);
    var depth;
```

```
    vec2.scale(normal, normal, 1/len); // normalize normal
    if (isInside) { //flip normal if inside the rect
        vec2.scale(normal, normal, -1);
        depth = circ2Shape.getRadius() + len;
    } else {
        depth = circ2Shape.getRadius() - len;
    }

    collisionInfo.setNormal(normal);
    collisionInfo.setDepth(depth);
    return true;
};
```

Modifying the RigidRectangle Collision

Edit the `RigidRectangle_Collision.js` file to compute for collision information when a collision occurs.

1. Modify the `collidedRectRect()` function to calculate the collision depth and normal. In the following code, the collision normal is chosen to be along either the x- or y-direction with the corresponding depth.

```
RigidRectangle.prototype.collidedRectRect = function(r1, r2, collisionInfo) {
    var vFrom1to2 = vec2.fromValues(0, 0);
    vec2.sub(vFrom1to2, r2.getPosition(), r1.getPosition());
    var xDepth = (r1.getWidth() / 2) + (r2.getWidth() / 2) - Math.abs(vFrom1to2[0]);
    if (xDepth > 0) {
        var yDepth = (r1.getHeight() / 2) + (r2.getHeight() / 2) -
                                                        Math.abs(vFrom1to2[1]);
        if (yDepth > 0)  {
            //axis of least penetration
            if (xDepth < yDepth) {
                if (vFrom1to2[0] < 0) {
                    collisionInfo.setNormal([-1, 0]);
                } else {
                    collisionInfo.setNormal([1, 0]);
                }
                collisionInfo.setDepth(xDepth);
            } else {
                if (vFrom1to2[1] < 0) {
                    collisionInfo.setNormal([0, -1]);
                } else {
                    collisionInfo.setNormal([0, 1]);
                }
                collisionInfo.setDepth(yDepth);
            }
            return true;
        }
    }
    return false;
};
```

2. Modify the `collided()` function to include the `collisionInfo` parameter. Take note that the default collision depth value is initialized to zero.

```
RigidRectangle.prototype.collided = function(otherShape, collisionInfo) {
    var status = false;
    collisionInfo.setDepth(0);
    switch (otherShape.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = this.collidedRectCirc(this, otherShape, collisionInfo);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = this.collidedRectRect(this, otherShape, collisionInfo);
            break;
    }
    return status;
};
```

Modifying the Rigid Circle Collision

Similarly to the `RigidRectangle` collision, the `RigidCircle_Collision.js` file must also be edited for extracting the collision information.

1. Modify the `collidedCircCirc()` function to calculate the collision depth and normal. In the following code, `vFrom1to2` defines the vector between the two circle centers. The collision depth is simply the difference between the length of the `vFrom1to2` vector and the sum of the circle radii, while the collision normal is simply `vFrom1to2`. Notice that when the two circles overlap completely, collision normal is defined to be along the positive y-direction.

```
RigidCircle.prototype.collidedCircCirc = function(c1, c2, collisionInfo) {
    var vFrom1to2 = [0, 0];
    vec2.sub(vFrom1to2, c2.getPosition(), c1.getPosition());
    var rSum = c1.getRadius() + c2.getRadius();
    var sqLen = vec2.squaredLength(vFrom1to2);
    if (sqLen > (rSum * rSum)) {
        return false;
    }
    var dist = Math.sqrt(sqLen);

    if (dist !== 0) { // overlapping
        vec2.scale(vFrom1to2, vFrom1to2, 1/dist);
        collisionInfo.setNormal(vFrom1to2);
        collisionInfo.setDepth(rSum - dist);
    }
    else //same position
    {
        collisionInfo.setDepth(rSum / 10);
        collisionInfo.setNormal([0, 1]);
    }
    return true;
};
```

2. Modify the `collided()` function to support the new `collisionInfo` parameter. Once again, note that the depth of the collision is initialized to zero by default and that the collision normal returned from `collidedRectCirc()` must be reversed for the `RigidCircle` object.

```
RigidCircle.prototype.collided = function(otherShape, collisionInfo) {
    var status = false;
    var n;
    collisionInfo.setDepth(0);
    switch (otherShape.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = this.collidedCircCirc(this, otherShape, collisionInfo);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = this.collidedRectCirc(otherShape, this, collisionInfo);
            n = collisionInfo.getNormal();
            n[0] = -n[0];
            n[1] = -n[1];
            break;
    }
    return status;
};
```

Defining the Engine Physics Component

A new physics engine component can now be defined to coordinate collision detections and collision responses via the relaxation iteration loops. To begin, follow the pattern of defining an engine component.

1. In the `src/Engine/Core` folder, create a new file and name it `Engine_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 following the JavaScript Module Pattern as follows:

```
var gEngine = gEngine || { };  // initialize the variable while ensuring it is
not redefined

gEngine.Physics = (function () {
    var mPublic = { };
    return mPublic;
}());
```

3. Define variables for relaxation count, current relaxation loop count, has collided collision check, position correction rate, system acceleration, and collision information. In addition, create an initialization function to allocate and initialize the shared `mCollisionInfo` object.

```
var mRelaxationCount = 15;                  // number of relaxation iteration
var mRelaxationOffset = 1/mRelaxationCount; // porportion to apply when scaling friction
var mPosCorrectionRate = 0.8;               // percentage of separation to project objects
var mSystemtAcceleration = [0, -50];        // system-wide default acceleration

var mRelaxationLoopCount = 0;               // the current relaxation count
var mHasOneCollision = false;               // detect the first collision

var mCollisionInfo = null;                  // information of the current collision

var initialize = function() {
    mCollisionInfo = new CollisionInfo();   // to avoid allocating this constantly
};
```

Resolving Collisions and Applying Friction

Define functions to resolve interpenetrating objects by the Projection Method and to slow down the moving objects by applying friction.

1. Define the function `_positionalCorrection()` to apply the Projection Method to correct the positions of the colliding objects. Calculate the `correctionAmount` in the collision normal direction as a function of the collision depth and mass of the respective objects. Note that the actual correction amount is inversely proportional to the mass of the object and scaled by the `mPosCorrectionRate`. For this reason, an object with infinite mass (`invMass` of 0) will not be positional corrected. In other words, this object will be unmovable (for example, a platform). The scaling by `mPosCorrectionRate` means that the interpenetration will not be resolved completely. Rather, it is expected that, through relaxation, the `_positionalCorrection()` function will be invoked multiple times in each game loop update.

```
var _positionalCorrection = function (s1, s2, collisionInfo) {
    var s1InvMass = s1.getInvMass();
    var s2InvMass = s2.getInvMass();
    var num = collisionInfo.getDepth() / (s1InvMass + s2InvMass) * mPosCorrectionRate;
    var correctionAmount = [0, 0];
    vec2.scale(correctionAmount, collisionInfo.getNormal(), num);

    var ca = [0, 0];
    vec2.scale(ca, correctionAmount, s1InvMass);
    var s1Pos = s1.getPosition();
    vec2.subtract(s1Pos, s1Pos, ca);

    vec2.scale(ca, correctionAmount, s2InvMass);
    var s2Pos = s2.getPosition();
    vec2.add(s2Pos, s2Pos, ca);
};
```

2. Create a function for applying friction to a colliding rigid shape. As illustrated in Figure 9-8, fiction is simulated by applying a deceleration in the tangent direction, $\hat{T}$, of the collision. The actual deceleration size is scaled by the `mRelaxationOffset` to account for multiple invocations from relaxation.

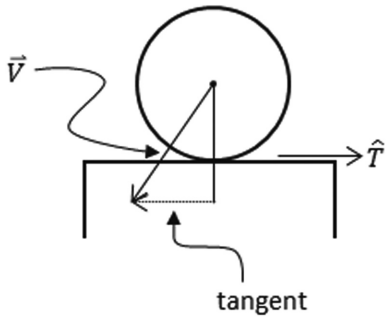

Figure 9-8. *The tangent direction of a collision*

```
// n is the collision normal
// v is the velocity
// f is the friction
// m is the invMass
var _applyFriction = function(n, v, f, m) {
    var tangent = vec2.fromValues(n[1], -n[0]);  // perpendicular to n
    var tComponent = vec2.dot(v, tangent);
    if (Math.abs(tComponent) < 0.01)
        return;

    f *= m * mRelaxationOffset;
    if (tComponent < 0) {
        vec2.scale(tangent, tangent, -f);
    } else {
        vec2.scale(tangent, tangent, f);
    }
    vec2.sub(v, v, tangent);
};
```

3. You can now create the `resolveCollision()` function to implement the collision resolution between two colliding objects. The resolution procedure requires access to the two objects and the corresponding collision information. Notice that the function is defined based on the abstract `RigidShape` and thus is able to resolve the collisions between any combinations of `RigidCircle` and `RigidRectangle`. The following are the steps to resolve the collision between two `RigidShapes`:

    a. Steps A sets `mHasOneCollision` to `true` to ensure that that relaxation loop will continue.

    b. Steps B calls to `_positionalCorrection()` to apply positional correction to the objects to push them apart by 80 percent (by default) of the collision depth.

    c. Step C calls to `_applyFriction()` to dampen the tangent component of the object velocities.

d. Step D calculates the relative velocity between the two objects by subtracting them. This relative velocity is important for computing the impulse that pushes the objects apart.

e. Step E computes `rVelocityInNormal`, the component of the relative velocity vector that is in the collision normal direction. This component indicates how rapidly the two objects are moving toward or away from each other. If `rVelocityInNormal` is greater than zero, then the objects are moving away from each other and impulse response will not be necessary.

f. Step F computes the impulse magnitude, `j`, based on `rVelocityInNormal`, restitution (bounciness), and the masses of the colliding objects. This impulse magnitude value will be used to modify both velocities to push them apart.

Figure 9-9 depicts the details of step F. `S1` and `S2` are the two colliding `RigidShapes`, where the `normal` vector is the collision normal. Note that this vector always points from object `S1` toward `S2`. The vectors `s1V-before` and `s2V-before` are the velocity vectors of the corresponding objects before step F, and the `s1V-after` and `s2V-after` are the impulse modified velocities after the step. Notice the two `-after` velocities would bring the corresponding objects away from the interpenetration.

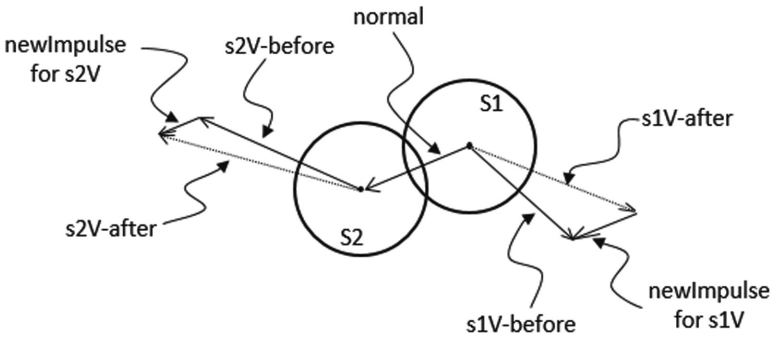

Figure 9-9. *Computing new object velocities based on collision normal and impulse*

```
var resolveCollision = function (s1, s2, collisionInfo) {
        // Step A: one collision has been found
        mHasOneCollision = true;

        // Step B: correct positions
        _positionalCorrection(s1, s2, collisionInfo);

        // Step C: apply friction
        var s1V = s1.getVelocity();  // collision normal direction is _against_ s2
        var s2V = s2.getVelocity();
        var n = collisionInfo.getNormal();
        _applyFriction(n, s1V, s1.getFriction(), s1.getInvMass());
        _applyFriction(n, s2V, -s2.getFriction(), s2.getInvMass());
```

```
    // Step D: compute relatively velocity of the colliding objects
    var relativeVelocity = [0, 0];
    vec2.sub(relativeVelocity, s2V, s1V);

    // Step E: examine the component in the normal direction
    var rVelocityInNormal = vec2.dot(relativeVelocity, n);
    if (rVelocityInNormal > 0) { //if objects moving apart ignore
        return;
    }

    // Step F: compute and apply response impulses for each object
    var newRestituion = Math.min(s1.getRestitution(), s2.getRestitution());
    var j = -(1 + newRestituion) * rVelocityInNormal;
    j = j / (s1.getInvMass() + s2.getInvMass());

    var impulse = [0, 0];
    vec2.scale(impulse, collisionInfo.getNormal(), j);

    var newImpulse = [0, 0];
    vec2.scale(newImpulse, impulse, s1.getInvMass());
    vec2.sub(s1V, s1V, newImpulse);

    vec2.scale(newImpulse, impulse, s2.getInvMass());
    vec2.add(s2V, s2V, newImpulse);
};
```

Adding the Relaxation Loop for Stability

With all the components of collision resolutions defined, you are now ready to define the relaxation support. Recall relaxation allows the collision resolution process to resolve the interpenetrations gradually over a predetermined number of times, in this case a default of 15. In general, larger relaxation iterations would result with a more natural appearance and more stable collision system. The associated computation cost is the main drawback with large relaxation iterations.

1. Continue with editing the `Engine_Physics.js` file and define the `beginRelaxation()` and `continueRelaxation()` functions to manage the relaxation iteration loop. Relaxation iterations can terminate when there are no more collisions detected.

```
var beginRelaxation = function() {
    mRelaxationLoopCount = mRelaxationCount;
    mHasOneCollision = true;
};
var continueRelaxation = function() {
    var oneCollision = mHasOneCollision;
    mHasOneCollision = false;
    mRelaxationLoopCount = mRelaxationLoopCount - 1;
    return ((mRelaxationLoopCount > 0) && oneCollision);
};
```

2. Define a `processObjObj()` function to support the relaxation of resolving the collision between two individual `GameObject` instances. The relaxation loop will continue only if the two objects continue to collide.

```
// Rigid Shape interactions: two game objects
var processObjObj = function(obj1, obj2) {
    var s1 = obj1.getPhysicsComponent();
    var s2 = obj2.getPhysicsComponent();
    if (s1 === s2)
        return;
    beginRelaxation();
    while (continueRelaxation()) {
        if (s1.collided(s2, mCollisionInfo)) {
            resolveCollision(s1, s2, mCollisionInfo);
        }
    }
};
```

3. Define a `processObjSet()` function to process the collision resolution between a `GameObject` instance and a `GameObjectSet`. Notice the nested loops. In the worst case, with the outer loop iterating at constant 15 times, the run time of this function is $O(N)$, where N is the size of the `RigidShape` set. It is important to be mindful of the cost of collision resolution.

```
// Rigid Shape interactions: a game object and a game object set
var processObjSet = function(obj, set) {
    var s1 = obj.getPhysicsComponent();
    var i, s2;
    beginRelaxation();
    while (continueRelaxation()) {
        for (i=0; i<set.size(); i++) {
            s2 = set.getObjectAt(i).getPhysicsComponent();
            if ((s1 !== s2) && (s1.collided(s2, mCollisionInfo))) {
                resolveCollision(s1, s2, mCollisionInfo);
            }
        }
    }
};
```

4. Define the `processSetSet()` function to process the collision resolution between two instances of `GameObjectSet`. Notice the triple nested loops. In the worst case, even with the outer loop iterating at constant 15 times, the run time of this function is $O(NM)$, where N and M are the sizes of the respective `GameObjectSet`. In other words, this function runs in quadratic times, and any large numbers of N or M will quickly degrade the performance of the system to non-real-time. It is important to design your system to avoid large set-to-set collisions.

```
// Rigid Shape interactions: two game object sets
var processSetSet = function(set1, set2) {
    var i, j, s1, s2;
    beginRelaxation();
```

```
    while (continueRelaxation()) {
        for (i=0; i<set1.size(); i++) {
            s1 = set1.getObjectAt(i).getPhysicsComponent();
            for (j=0; j<set2.size(); j++) {
                s2 = set2.getObjectAt(j).getPhysicsComponent();
                if ((s1 !== s2) && (s1.collided(s2, mCollisionInfo))) {
                    resolveCollision(s1, s2, mCollisionInfo);
                }
            }
        }
    }
};
```

5. Create get and set accessors. Notice that since the relaxation rate is a percentage, it should remain between zero and one, and note that the relaxation count should never be below zero.

```
var getSystemAcceleration = function() { return mSystemAcceleration; };
var setSystemAcceleration = function(g) { mSystemAcceleration = g; };
var getRelaxationCorrectionRate = function() { return mPosCorrectionRate; };
var setRelaxationCorrectionRate = function(r) {
    if ((r <= 0) || (r>=1)) {
        r = 0.8;
    }
    mPosCorrectionRate = r;
};
var getRelaxationLoopCount = function() { return mRelaxationCount; };
var setRelaxationLoopCount = function(c) {
    if (c <= 0)
        c = 1;
    mRelaxationCount = c;
    mRelaxationOffset = 1/mRelaxationCount;
};
```

6. Provide public access to the functions.

```
var mPublic = {
    initialize: initialize,
    resolveCollision: resolveCollision,
    beginRelaxation: beginRelaxation,
    continueRelaxation: continueRelaxation,
    getSystemAcceleration: getSystemAcceleration,
    setSystemAcceleration: setSystemAcceleration,
    getRelaxationCorrectionRate: getRelaxationCorrectionRate,
    setRelaxationCorrectionRate: setRelaxationCorrectionRate,
    getRelaxationLoopCount: getRelaxationLoopCount,
    setRelaxationLoopCount: setRelaxationLoopCount,
    processObjObj: processObjObj,
    processObjSet: processObjSet,
    processSetSet: processSetSet
};
return mPublic;
```

Supporting the Engine Physics Component

A few minor modifications are required to integrate the physics component into the engine.

1. Edit the `Engine_Core.js` file by adding the following code to the `initializeEngineCore()` function to initialize the physics component of the engine:

```
gEngine.Physics.initialize();
```

2. Edit the `Engine_GameLoop.js` file by adding the following variables and functions to gain access to the time interval between frames. This is the delta time used in your numerical integration.

```
var kFrameTime = 1 / kFPS;
var kMPF = 1000 * kFrameTime; // Milliseconds per frame.

var getUpdateIntervalInSeconds = function () {
    return kFrameTime;
};

var mPublic = {
    start: start,
    stop: stop,
    getUpdateIntervalInSeconds: getUpdateIntervalInSeconds
};
```

Testing Impulse Resolution

The implementation should be tested in two ways, first to ensure moving objects collide and behave naturally and second to ensure the collision resolution system is stable when there are many objects in proximity. The implementation of the testing system has three interesting details.

- A new object type, `Wall`, is defined in the `Wall.js` file to represent the bounding walls of the world.
- It is important to take note to call the `RigidShape.setMess()` function to set the mass of stationary objects to zero, such as the `platform` and the `wall`.
- The new physics engine functionality is invoked from the `MyGame._physicsSimulation()` function defined in the `MyGame_Physics.js` file.

The rest of the testing system is largely similar to previous projects, and the detailed code listing will not be presented. Please refer to the source code files for the details of the implementations.

Observations

Run the project to test your results. Notice that right off the bat the hero character falls gradually to the floor and stops with a slight rebound. This is a clear indication that the base case for numerical integration, collision detection, and resolution all are operating as expected. Now notice the wandering `Minion` objects; they interact properly with the platforms and the walls of the game world with soft bounces and no apparent interpenetrations. Now, use the WASD keys to navigate the `Hero` object; make

sure to collide the Hero into the Minion objects and observe the apparent transfer of energies. A right-click releases home-in DyePack objects. This is to verify the proper functioning of the rest of the GameObject class. Notice that when the DyePack objects collide with the Platforms or with the Minions, they properly skirt the edges of the RigidShape. Additionally, observe that the relatively small mass of a DyePack object resulting in the DyePack has little effect on the Minion's trajectory when they collide. However, when many DyePack objects collide with the same Minion object at the same time, the results can have a significant effect on a Minion's movement.

The stability of the system can be tested by pressing the Z key, where many dumb Minion objects with white RigidShape bounds will be spawned. Take note that these white-bounded Minion objects are meant for testing system stability and will collide only with the Platform objects and among themselves. Notice how the white-bounded Minion objects settle and stack at the bottom of the scene; the lack of jitter in the objects here demonstrates the stability of the system. Now, press the C key a few times to excite the white-bounded Minions and watch the system settle back into a stable state. Alternatively, you can press the X key and select and move a white-bounded Minion to observe how the system settles back into a stable state after a local disturbance. Be careful with the number of white-bounded Minions you spawn. The quadratic runtime means the system performance can degrade quickly with even a modest number of Minions. You can press the C key to clear all the white-bounded Minions.

Particles and Particle Systems

A particle is a textured position with no defined dimensions. This description may seem contradictory because you have learned that a texture is an image and images are always defined by a width and height and will definitely occupy areas. The important clarification is that the game engine logic processes a particle as a position with no area, and the drawing system displays the particle as a texture. In this way, even though an actual displayed area is shown, the width and height dimensions of the texture are ignored by the underlying logic.

In addition to a position, a particle also has properties such as life span, color (for tinting the texture), and size (for scaling the texture). A particle system is a collection of particles with the same set of properties. By strategically spawning, manipulating the properties of, and removing particles in a particle system, interesting visual effects such as fire or explosions can be simulated.

In this section, you will create a simple and flexible particle system that includes the basic functionality required to achieve common effects, such as explosions and spell effects. Additionally, you will implement a particle shader to properly blend your particles within your scenes. The particles will collide and interact accordingly with the RigidShape objects.

The Particles Project

This project demonstrates how to implement a particle system to simulate explosion-like effects and to interact with the RigidShape object within the scene. You can see an example of this project running in Figure 9-10. The source code of this project is located in the Chapter9/9.3.Particles folder.

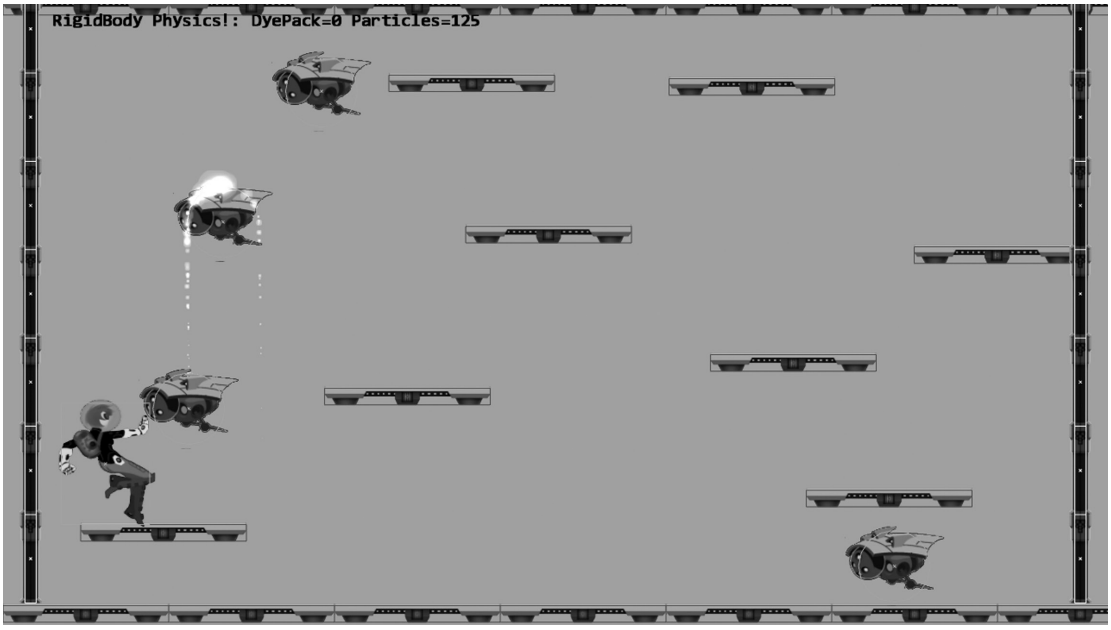

Figure 9-10. *Running the Particles project*

The controls of the project are as follows:

- *WASD keys*: Moves the `Hero` object
- *Left mouse button click in the game window*: Sends a `DyePack` chasing after the `Hero`
- *Z key*: Spawns particles at the current mouse position

The goals of the project are as follows:

- To understand the details of how to draw a particle and define its behaviors
- To experience implementing a particle system
- To build a particle engine component that supports interaction with `RigidShape`

You can find the following external resources in the `assets` folder: the `fonts` folder that contains the default system fonts and four texture images (`minion_sprite.png`, which defines the sprite elements for the hero and the minions; `platform.png`, which defines the platforms and floor and ceiling tiles; `wall.png`, which defines the walls; `dye_pack.png`, which defines the dye packs; and `particle.png`, which is used to draw one particle).

In the following exercise, you will first learn about the support for drawing a particle object. After that, you will examine the details of how to create an actual particle object and define its behaviors. Lastly, you will create a new game engine component to manage the interaction of the particles with the `RigidShape` objects.

A particle is a new type of object for your game engine and requires the entire drawing system support, including custom GLSL shaders, default sharable shader instance, and a new `Shader`/`Renderable` pair.

Creating GLSL Particle Fragment Shader

Particles are textured positions with no areas. However, as discussed, your engine will draw each particle as a textured rectangle. For this reason, you can simply reuse the existing texture vertex shader `TextureVS.glsl`. When it comes to the actual computation of each pixel color, a new fragment shader, `ParticleFS.glsl`, must be created to properly simulate the intense brightness of fire and explosions. The new fragment shader aggressively composites colors to create oversaturation, where the resulting values of the RGB components quickly become the maximum displayable value of 1.0.

1. Under the `src/GLSLShaders` folder, add a new file and name it `ParticleFS.glsl`.

2. Similar to the texture fragment shader defined in `TextureFS.glsl`, you need to declare `uPixelColor` and `vTexCoord` to receive these values from the game engine and define the `uSampler` to sample the texture.

```
precision mediump float;
    // sets the precision for floating point computation

// The object that fetches data from texture.
// Must be set outside the shader.
uniform sampler2D uSampler;

// Color of pixel
uniform vec4 uPixelColor;

// The "varying" keyword is for signifing that the texture coordinate will be
// interpolated and thus varies.
varying vec2 vTexCoord;
```

3. Implement the main function to accumulate colors aggressively.

```
void main(void)  {
    // texel color look up based on interpolated UV value in vTexCoord
    vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

    vec3 r = vec3(c) * c.a * vec3(uPixelColor);
    vec4 result = vec4(r, uPixelColor.a);

    gl_FragColor = result;
}
```

Defining a Default ParticleShader Instance

You can now modify the engine to support the initializing, loading, and unloading of a new `ParticleShader`.

1. Begin by adding a variable for the particle shader in the `Engine_DefaultResources.js` file located in the `src/Engine/Core/Resources` folder. Also, define an accessor function as shown here:

```
var kParticleFS = "src/GLSLShaders/ParticleFS.glsl";
var mParticleShader = null;
var getParticleShader = function () { return mParticleShader };
```

2. Now instantiate a new particle shader in the `_createShaders()` function, as shown here:

```
var _createShaders = function(callBackFunction) {
    gEngine.ResourceMap.setLoadCompleteCallback(null);
    // ... identical to previous projects
    mParticleShader = new TextureShader(kTextureVS, kParticleFS);
    callBackFunction();
};
```

3. In the `initialize()` function, add the following code to properly load the file:

```
gEngine.TextFileLoader.loadTextFile(kParticleFS,
    gEngine.TextFileLoader.eTextFileType.eTextFile);
```

4. In the `cleanUp()` function, add the following line of code to unload the file when it is no longer needed:

```
gEngine.TextFileLoader.unloadTextFile(kParticleFS);
```

5. Don't forget to add the get accessor function to `mPublic()` so that it can be accessed.

```
getParticleShader: getParticleShader,
```

Creating the ParticleRenderable Object

Recall that a `Shader/Renderable` pair of objects must be defined to interface the GLSL shader to the game engine. With the default `ParticleShader` object defined to interface to the GLSL `ParticleFS` shader, you can now create a new `Renderable` object to support the drawing of particles. Fortunately, the detailed behaviors of a particle, or a textured position, is identical to that of a `TextureRenderable` with the exception of the different shader. As such, the definition of the `ParticleRenderable` object is trivial.

1. Under the `src/Engine/Renderables` folder, add a new file and name it `ParticleRenderable.js`. Remember to load this new source file in `index.html`.

2. Define the new renderable object as a simple subclass of the `TextureRenderable`, with the exception of a different default shader.

```
function ParticleRenderable(myTexture) {
    TextureRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.
getParticleShader());
}
gEngine.Core.inheritPrototype(ParticleRenderable, TextureRenderable);
```

Defining the Particle and Particle Game Object

With the infrastructure for drawing a particle object defined, it is time to create the actual particle and define its behaviors.

Creating a Particle

To support moving particles around the scene, their movements can be approximated with the Symplectic Euler Integration.

1. Begin by creating a new subfolder called `Particles` under the `src/Engine` folder. In this folder, create a new file called `Particle.js`. Remember to load this new source file in `index.html`.

2. Create the constructor and add variables for position, velocity, acceleration, drag, and drawing variables for marking its position (for debugging).

```
function Particle(pos) {
    this.kPadding = 0.5;   // for drawing particle bounds

    this.mPosition = pos;  // this is likely to be a reference to xform.mPosition
    this.mVelocity = vec2.fromValues(0, 0);
    this.mAcceleration = gEngine.Particle.getSystemtAcceleration();
    this.mDrag = 0.95;

    this.mPositionMark = new LineRenderable();
    this.mDrawBounds = false;
}
```

3. Create a `draw()` function to draw the position of the particle with a simple X. This function is to support the debugging of a particle system.

```
Particle.prototype.draw = function (aCamera) {
    if (!this.mDrawBounds) {
        return;
    }
    //calculation for the X at the particle position
    var x = this.mPosition[0];
    var y = this.mPosition[1];
    this.mPositionMark.setFirstVertex(x - this.kPadding, y + this.kPadding);
                                                                    //TOP LEFT
    this.mPositionMark.setSecondVertex(x + this.kPadding, y - this.kPadding);
                                                                //BOTTOM RIGHT
    this.mPositionMark.draw(aCamera);
    this.mPositionMark.setFirstVertex(x + this.kPadding, y + this.kPadding);
                                                                    //TOP RIGHT
    this.mPositionMark.setSecondVertex(x - this.kPadding, y - this.kPadding);
                                                                //BOTTOM LEFT
    this.mPositionMark.draw(aCamera);
};
```

4. You can now implement the `update()` function for calculating the particle's position. This is a straightforward implementation of the Symplectic Euler Integration. The `mDrag` variable simulates drags on the particles.

```
Particle.prototype.update = function () {
    var dt = gEngine.GameLoop.getUpdateIntervalInSeconds();
    // Symplectic Euler
    //    v += a * dt
    //    x += v * dt
    var p = this.getPosition();
    vec2.scaleAndAdd(this.mVelocity, this.mVelocity, this.mAcceleration, dt);
    vec2.scale(this.mVelocity, this.mVelocity, this.mDrag);
    vec2.scaleAndAdd(p, p, this.mVelocity, dt);
};
```

5. Define simple get and set accessors. These functions are straightforward and are not listed here.

Creating the ParticleGameObject

With the actual particle object defined, you can now create the `ParticleGameObject` class to abstract the particle's behavior and to support the drawing of the particle.

1. Under the `src/Engine/Particles` folder, add a new file and name it `ParticleGameObject.js`. Remember to load this new source file in `index.html`.

2. Define a constructor to initialize variables for manipulating the color, size, and life span of the particle.

```
function ParticleGameObject(texture, atX, atY, cyclesToLive) {
    var renderableObj = new ParticleRenderable(texture);
    var xf = renderableObj.getXform();
    xf.setPosition(atX, atY);
    GameObject.call(this, renderableObj);

    var p = new Particle(xf.getPosition());
    this.setPhysicsComponent(p);

    this.mDeltaColor = [0, 0, 0, 0];
    this.mSizeDelta = 0;
    this.mCyclesToLive = cyclesToLive;
}
gEngine.Core.inheritPrototype(ParticleGameObject, GameObject);
```

3. Create a function for setting the final color of the particle. This function computes the required rate of change such that the color of the particle will be constantly changing during each update cycle unit the particle expires.

```
ParticleGameObject.prototype.setFinalColor = function(f) {
    vec4.sub(this.mDeltaColor, f, this.mRenderComponent.getColor());
    if (this.mCyclesToLive !== 0) {
        vec4.scale(this.mDeltaColor, this.mDeltaColor, 1/this.mCyclesToLive);
    }
};
```

4. Define accessor functions to set the particle size change rate and to test whether the particle has expired. An expired particle should be removed from the system.

```
ParticleGameObject.prototype.setSizeDelta = function(d) {  this.mSizeDelta = d; };
ParticleGameObject.prototype.hasExpired = function() { return (this.mCyclesToLive
                                                              < 0); };
```

5. Add an `update()` function to change the color and size of the particle.

```
ParticleGameObject.prototype.update = function () {
    GameObject.prototype.update.call(this);

    this.mCyclesToLive--;
    var c = this.mRenderComponent.getColor();
    vec4.add(c, c, this.mDeltaColor);

    var xf = this.getXform();
    var s = xf.getWidth() * this.mSizeDelta;
    xf.setSize(s, s);
};
```

Creating the ParticleGameObjectSet

To work with a collection of particles, you can now create the `ParticleGameObjectSet` to support conveniently looping over `ParticleGameObjects`.

1. Under the `src/Engine/Particles` folder, add a new file and name it `ParticleGameObjectSet.js`. Remember to load this new source file in `index.html`.

2. Define `ParticleGameObjectSet` to be a subclass of `GameObjectSet`.

```
function ParticleGameObjectSet() {
    GameObjectSet.call(this);
}
gEngine.Core.inheritPrototype(ParticleGameObjectSet, GameObjectSet);
```

3. Override the `draw()` function of `GameObjectSet` to ensure particles are drawn with additive blending. Recall that the default `gl.blendFunc()` setting utilizes the alpha channel value to implement transparency. This is referred to as *alpha blending*. In this case, the `gl.blendFunc()` setting results in a simple accumulation of colors without considering the alpha channel. This is referred to as additive blending. As mentioned, to properly simulate the intense brightness of fire and explosions, it is desirable to aggressively accumulate colors such that the RGB components can quickly reach the maximum displayable value of 1.0. Additive blending facilitates the aggressive accumulation.

```
ParticleGameObjectSet.prototype.draw = function (aCamera) {
    var gl = gEngine.Core.getGL();
    gl.blendFunc(gl.ONE, gl.ONE);  // for additive blending!
    GameObjectSet.prototype.draw.call(this, aCamera);
    gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA); // restore alpha blending
};
```

4. Override the `update()` function to ensure expired particles are removed.

```
ParticleGameObjectSet.prototype.update = function () {
    GameObjectSet.prototype.update.call(this);

    // Cleanup Particles
    var i, obj;
    for (i=0; i<this.size(); i++) {
        obj = this.getObjectAt(i);
        if (obj.hasExpired()) {
            this.removeFromSet(obj);
        }
    }
};
```

Defining the Engine Particle Component

With the drawing and behavior infrastructures defined, you can now define the engine component to manage the particle system.

1. In the `src/Engine/Core` folder, add a new file and name it `Engine_Particle.js`. Remember to load this new source file in `index.html`.
2. Define the particle component following the JavaScript Module Pattern as follows:

```
var gEngine = gEngine || { };
    // initialize the variable while ensuring it is not redefined

gEngine.Particle = (function () {
    var mPublic = { };
    return mPublic;
}());
```

3. Define variables for system acceleration and a common calculation space for heavily used variables.

```
var mSystemAcceleration = [0, -50.0];

// the follows are scratch workspace for vec2
var mFrom1to2 = [0, 0];
var mVec = [0, 0];
var mNormal = [0, 0];
```

4. Add a function to resolve the collision between a circle and a particle. This can be accomplished by a simple radius check. Particles inside a circle are projected out. Notice that, in this case, with a zero-area particle, collision resolution is straightforward to handle and that there is no need for relaxation.

```
var resolveCirclePos = function (circShape, particle) {
    var collided = false;
    var pos = particle.getPosition();
    var cPos = circShape.getPosition();
    vec2.subtract(mFrom1to2, pos, cPos);
    var dist = vec2.length(mFrom1to2);
    if (dist < circShape.getRadius()) {
        vec2.scale(mFrom1to2, mFrom1to2, 1/dist);
        vec2.scaleAndAdd(pos, cPos, mFrom1to2, circShape.getRadius());
        collided = true;
    }
    return collided;
};
```

5. Similar to the case of a circle, add the `resolveRectPos()` function to resolve the collision between a rectangle and a particle. Once again, particles inside a rectangle are simply projected out along the nearest x- or y-boundary.

```
var resolveRectPos = function (rectShape, particle) {
    var collided = false;
    var alongX = rectShape.getWidth() / 2;
    var alongY = rectShape.getHeight() / 2;

    var pos = particle.getPosition();
    var rPos = rectShape.getPosition();

    var rMinX = rPos[0] - alongX;
    var rMaxX = rPos[0] + alongX;
    var rMinY = rPos[1] - alongY;
    var rMaxY = rPos[1] + alongY;

    collided = ((rMinX<pos[0]) && (rMinY<pos[1]) &&
                (rMaxX>pos[0]) && (rMaxY>pos[1]));

    if (collided) {
        vec2.subtract(mFrom1to2, pos, rPos);
        mVec[0] = mFrom1to2[0];
        mVec[1] = mFrom1to2[1];

        // Find closest axis
        if (Math.abs(mFrom1to2[0] - alongX) < Math.abs(mFrom1to2[1] - alongY))  {
            // Clamp to closest side
            mNormal[0] = 0;
            mNormal[1] = 1;
            if (mVec[0] > 0) {
                mVec[0] = alongX;
```

```
            } else {
                mVec[0] = -alongX;
            }
        } else { // y axis is shorter
            mNormal[0] = 1;
            mNormal[1] = 0;
            // Clamp to closest side
            if (mVec[1] > 0) {
                mVec[1] = alongY;
            } else {
                mVec[1] = -alongY;
            }
        }

        vec2.subtract(mVec, mVec, mFrom1to2);
        vec2.add(pos, pos, mVec);  // remember pos is particle position
    }
    return collided;
};
```

6. Define a function for processing the collision between an object and a set of particles.

```
// Rigid Shape interactions: a game object and a set of particle game objects
var processObjSet = function(obj, pSet) {
    var s1 = obj.getPhysicsComponent();  // a RigidShape
    var i, p;
    for (i=0; i<pSet.size(); i++) {
        p = pSet.getObjectAt(i).getPhysicsComponent();  // a Particle
        s1.resolveParticleCollision(p);
    }
};
```

7. Define a function to process the collision between a `GameObjectSet` and a set of particles.

```
// Rigid Shape interactions: game object set and a set of particle game objects
var processSetSet = function(objSet, pSet) {
    var i;
    for (i=0; i<objSet.size(); i++) {
        processObjSet(objSet.getObjectAt(i), pSet);
    }
};
```

8. Finally, implement the necessary get and set accessors for the system's acceleration and remember to add the necessary functions to `mPublic`.

```
var getSystemAcceleration = function() { return mSystemAcceleration; };
var setSystemAcceleration = function(g) { mSystemAcceleration = g; };
```

```
        var mPublic = {
            getSystemAcceleration: getSystemAcceleration,
            setSystemAcceleration: setSystemAcceleration,
            resolveCirclePos: resolveCirclePos,
            resolveRectPos: resolveRectPos,
            processObjSet: processObjSet,
            processSetSet: processSetSet
        };
        return mPublic;
```

Interacting Particles with RigidShape

To properly support the interaction between `Particle` objects and `RigidShape` objects, edit the `RigidShape_Collision.js` file and define the following function to define the corresponding collision function:

```
RigidShape.prototype.resolveParticleCollision = function(aParticle) {
    var status = false;
    switch (this.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = gEngine.Particle.resolveCirclePos(this, aParticle);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = gEngine.Particle.resolveRectPos(this, aParticle);
            break;
    }
    return status;
};
```

Testing the Particle System

The test should verify two main goals. First, the implemented particle system is capable of generating visually pleasant effects. Second, the particles are capable of colliding with `RigidShape` objects and resulting in expected behaviors. The test case is based mainly on the previous project with one simple modification—invoking the `_createParticle()` function when the Z key is pressed. Please consult the source code for the details of the implementation.

The `_createParticle()` function listed in the following configures and creates particles. There are two important observations. First, the `random()` function is used many times to configure each created `Particle`. Particle systems utilize large numbers of similar yet slightly different particles to build and convey a sense of the desired visual effect. It is important to avoid patterns of any sort, and this use of randomness is an important rule to follow. Second, there are many seemingly arbitrary numbers used in the configuration, such as setting the life of the particle to be between 30 and 230 or setting the final red component to a number between 3.5 and 4.5. This, unfortunately is the nature of working with particle systems: there is quite a bit of ad hoc trying. Commercial game engines typically alleviate this difficulty by releasing a collection of preset values for their particle systems. In this way, game designers can fine-tune specific desired effects by tweaking the provided presets.

```
MyGame.prototype._createParticle = function(atX, atY) {
    var life = 30 + Math.random() * 200;
    var p = new ParticleGameObject(this.kParticleTexture, atX, atY, life);
    p.getRenderable().setColor([1, 0, 0, 1]);
```

```
    // size of the particle
    var r = 5.5 + Math.random() * 0.5;
    p.getXform().setSize(r, r);

    // final color
    var fr = 3.5 + Math.random();
    var fg = 0.4 + 0.1 * Math.random();
    var fb = 0.3 + 0.1 * Math.random();
    p.setFinalColor([fr, fg, fb, 0.6]);

    // velocity on the particle
    var fx = 10 - 20 * Math.random();
    var fy = 10 * Math.random();
    p.getPhysicsComponent().setVelocity([fx, fy]);

    // size delta
    p.setSizeDelta(0.98);

    return p;
};
```

Observations

Run the project and press the Z key to observe the generated particles. It appears as though there is combustion occurring underneath the mouse pointer. Hold the Z key and move the mouse pointer around slowly to observe the combustion following the mouse as though there is an engine generating flames under the mouse pointer. If you move the mouse pointer quickly, you will observe individual pink circles changing color while dropping toward the floor. Particle systems must be fine-tuned for individual situations. Now, hold the Z key and move the mouse pointer over any of the other `RigidShape` objects, such as one of the `Minions`. Observe that particles are projected to the boundary and continue to behave as expected. In addition, notice that the particles skirt the `RigidShape` object bounds when collided, just as you would expect.

Particle Emitters

With your current particle system implementation, you can create particles at a specific point and time. These particles when created can move and change based on their properties. However, particles can be created only when there is an explicit state change such as a key click. This becomes restricting when it is desirable to persist the generation of particles after the state change, such as an explosion that persists for a short while after the dye pack collides with the hero. A particle emitter addresses this issue by defining the functionality of generating particles over a time period.

The Particle Emitters Project

This project demonstrates how to implement a particle emitter for your particle system to support emitting particles over time. You can see an example of this project running in Figure 9-11. The source code of this project is located in the `Chapter9/9.4.ParticleEmitters` folder.

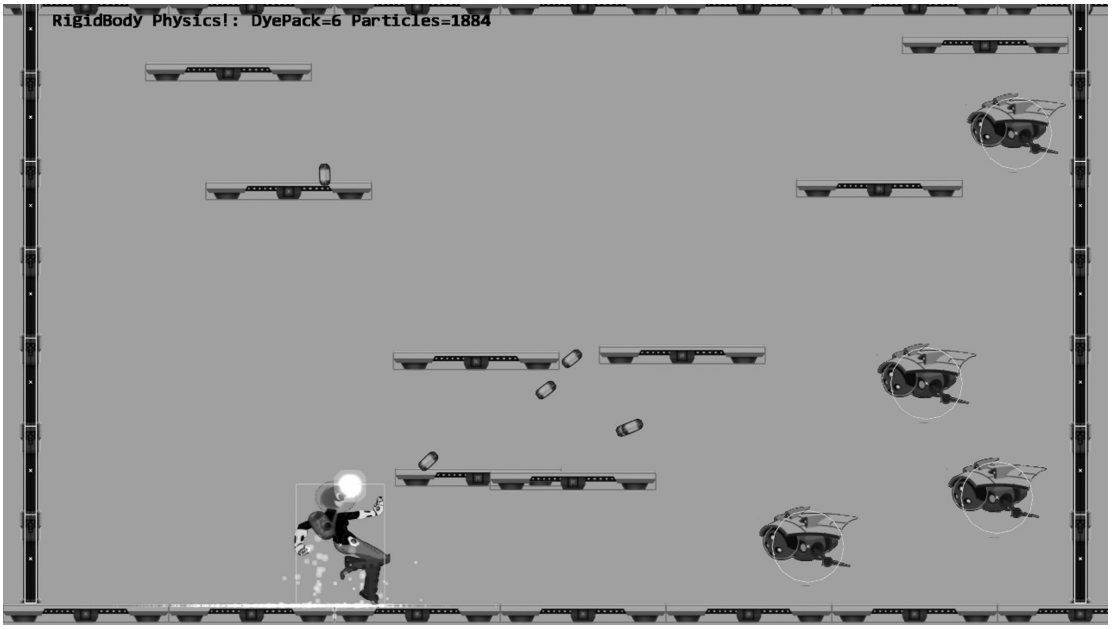

Figure 9-11. *Running the Particle Emitters project*

The controls of the project are as follows:

- *WASD keys*: Moves the `Hero` object
- *Left mouse button click in the game window*: Sends a `DyePack` chasing after the `Hero`
- *Z key*: Spawns particles at the current mouse position

The goals of the project are as follows:

- To understand the need for particle emitters
- To experience implementing particle emitters

You can find the following external resources in the `assets` folder: the `fonts` folder that contains the default system fonts and four texture images (`minion_sprite.png`, which defines the sprite elements for the hero and the minions; `platform.png`, which defines the platforms and floor and ceiling tiles; `wall.png`, which defines the walls; `dye_pack.png`, which defines the dye packs; and `particle.png`, which is used to draw one particle).

Creating the ParticleEmitter Object

Recall that when working with particles, it is important to avoid patterns. In this case, as the `ParticleEmitter` object generates new particles over time, it is important to build in randomness to avoid any patterns.

1. Under the `src/Engine/Particles` folder, add a new file and name it `ParticleEmitter.js`. Remember to load this new source file in `index.html`.

2. Create a constructor to define and set the default number of particles to emit at each cycle, the emitter's position, the number of particles left to be emitted, and `mParticleCreator`, the callback function for actual particles creation.

```
function ParticleEmitter(pos, num, createrFunc) {
    // Smallest number of particle emitted per cycle
    this.kMinToEmit = 5;

    // Emitter position
    this.mEmitPosition = pos;   // this can be a reference to a xform.mPosition

    // Number of particles left to be emitted
    this.mNumRemains = num;

    this.mParticleCreator = createrFunc;
}
```

3. Define a function to return the functioning status for the emitter. When there are no more particles to emit, the emitters should be removed.

```
ParticleEmitter.prototype.expired = function () { return (this.mNumRemains <= 0); };
```

4. Create a function to actually emit particles. Take note of the randomness in the number of particles that are actually emitted and the invocation of the `mParticleCreator()` callback function. With this design, there would be no patterns in the number of particles that are created over time. In addition, the emitter defines only the mechanisms of how, when, and where particles will be emitted and does not define the characteristics of the created particles. The function pointed to by `mParticleCreator` is responsible for defining that.

```
ParticleEmitter.prototype.emitParticles = function (pSet) {
    var numToEmit = 0;
    if (this.mNumRemains < this.kMinToEmit) {
        // If only a few are left, emits all of them
        numToEmit = this.mNumRemains;
    } else  {
        // Otherwise, emits about 20% of what's left
        numToEmit = Math.random() * 0.2 * this.mNumRemains;
    }
    // Left for future emitting.
    this.mNumRemains -= numToEmit;
    var i, p;
    for (i = 0; i < numToEmit; i++) {
        p = this.mParticleCreator(this.mEmitPosition[0], this.mEmitPosition[1]);
        pSet.addToSet(p);
    }
};
```

Modifying the Particle Game Object Set

The defined ParticleEmitter object needs to be integrated into ParticleGemeObjectSet to manage the emitted particles.

1. Edit the ParticleGameObjectSet.js file; in the constructor, define and initialize a particle emitter set.

```
this.mEmitterSet = [];
```

2. Define a function for instantiating a new emitter. Take note of the func parameter; this is the callback function that is responsible for the actual creation of individual Particle objects.

```
ParticleGameObjectSet.prototype.addEmitterAt = function (p, n, func) {
    var e = new ParticleEmitter(p, n, func);
    this.mEmitterSet.push(e);
};
```

3. Modify the update function to loop through the emitter set to generate new particles. Expired emitters are removed.

```
ParticleGameObjectSet.prototype.update = function () {
    // ... identical to previous project

    // Emit new particles
    for (i=0; i<this.mEmitterSet.length; i++) {
        e = this.mEmitterSet[i];
        e.emitParticles(this);
        if (e.expired()) {
            this.mEmitterSet.splice(i, 1);
        }
    }
};
```

Testing the Particle Emitter

This is a straightforward testing of the correct functioning of the ParticleEmitter object. The two interesting implementation details are as follows:

1. In the Hero.js file, at the end of the update() function, when a collision between a dye pack and the hero is detected, a new ParticleEmitter object is instantiated with the func callback function.

```
for (i=0; i<dyePacks.size(); i++) {
    obj = dyePacks.getObjectAt(i);
    // chase after hero
    obj.rotateObjPointTo(p, 0.8);
    if (obj.pixelTouches(this, collisionPt)) {
        dyePacks.removeFromSet(obj);
        allParticles.addEmitterAt(collisionPt, 200, func);
    }
}
```

The func parameter is defined in the update() function of the MyGame.js file as follows:

```
var func = function(x, y) { this.createParticle.call(this, x, y); };
```

where the createParticle() function is identical to the one from the previous project.

2. The RigidShape and Particle system interaction is triggered from the end of the _physicsSimulation() function defined in the MyGame_Physics.js file as follows:

```
MyGame.prototype._physicsSimulation = function() {
    // ... identical to previous project

    // Particle system collisions
    gEngine.Particle.processSetSet(this.mAllMinions, this.mAllParticles);
    gEngine.Particle.processSetSet(this.mAllPlatforms, this.mAllParticles);
};
```

As previously, the rest of the implementation is straightforward and will not be listed to avoid unnecessary distraction. Please refer to the source code in the src/MyGame folder for details.

Observations

Run the project and see the same behavior as in the previous project. The only exception is the new explosion effects after the dye packs collide with the Hero object. Notice how each explosion is triggered by the collision and persists for a short while before disappearing gradually. Comparing this effect with the one resulting from a short tapping of the Z key, observe that without a dedicated particle emitter, the explosion seems to have fizzled before it begins.

Summary

This chapter led you in building a simple yet flexible physics infrastructure for your game engine. Focusing on the implementation of core concepts, you learned to approximate Newtonian motion integrals with the Symplectic Euler Integration and how to build a numerically stable physics simulation that includes relaxation, collisions detection computation between axis-aligned rectangles and circles, extraction of collision normal and depth, and collision resolution using the Projection and Impulse methods. In addition, you have integrated a basic particle system with your physics engine with particles that interact with the RigidShape objects in the system. Through working with your particle system, you have learned that appropriate use of randomness is important and that creating interesting visual effects requires hands-on experience and fine-tuning iterations.

Chapter 8 complemented Chapter 5 by covering the simulation of illumination to improve the fidelity of the drawn game objects. In turn, this chapter augmented Chapter 6 by presenting the approximation of Newtonian motions to support the increase in behavioral complexity of the interacting game objects. The next chapter will extend Chapter 7 and describe more advanced topics in working with the camera.

Game Design Considerations

The puzzle level I've been building in the examples to this point has focused entirely on creating an understandable and consistent logical challenge. I've avoided burdening the exercise with any kind of visual design, narrative, or setting (design elements traditionally associated with enhancing player presence) to ensure you're thinking only about the rules of play without introducing superfluous distractions. However, as you create core game mechanics, it's important to understand how certain elements of gameplay can contribute directly to presence (recall that *presence* is the player's sense of feeling as if they're connected to the game world). The logical rules and requirements of core mechanics often have no effect on presence until they're paired with an interaction model, audiovisual design, or game setting, although some elements of game mechanics can directly influence presence. As discussed in Chapter 8, lighting is one example of a typically presence-enhancing visual design element that can also be used directly as a core game mechanic; object physics is similarly a presence-enhancing design element that's perhaps even more frequently connected directly to core game mechanics.

Our entire experience in the physical world is governed by physical laws, and when you model physics in virtual worlds, it makes an immediate and natural connection to the physical world and will therefore frequently strengthen presence. An example of objectphysics enhancing presence but not contributing to design would be destructible environments that have no direct impact on gameplay. If you're playing a first-person shooter, for example, and shoot at crates and other game objects that respond by realistically exploding on impact or if you throw a ball in the game world that bounces in a reasonable approximation of how a ball would bounce in the physical world, these are examples of physics being used purely to enhance presence but not necessarily contributing to the game mechanic. If you're playing Angry Birds, however, and launching one of the birds from your slingshot into the game space and must aim your shot based on the physics-modeled parabolic arc the bird follows upon launch, as shown in Figure 9-12, that's an example of physics being used as both a core element of gameplay while also enhancing presence. In fact, any game that involves jumping a hero character in an environment with simulated gravity is an example of physics contributing to both presence and the core mechanic, so most platformer games utilize physics as both a core mechanic and a presence-enhancing design element.

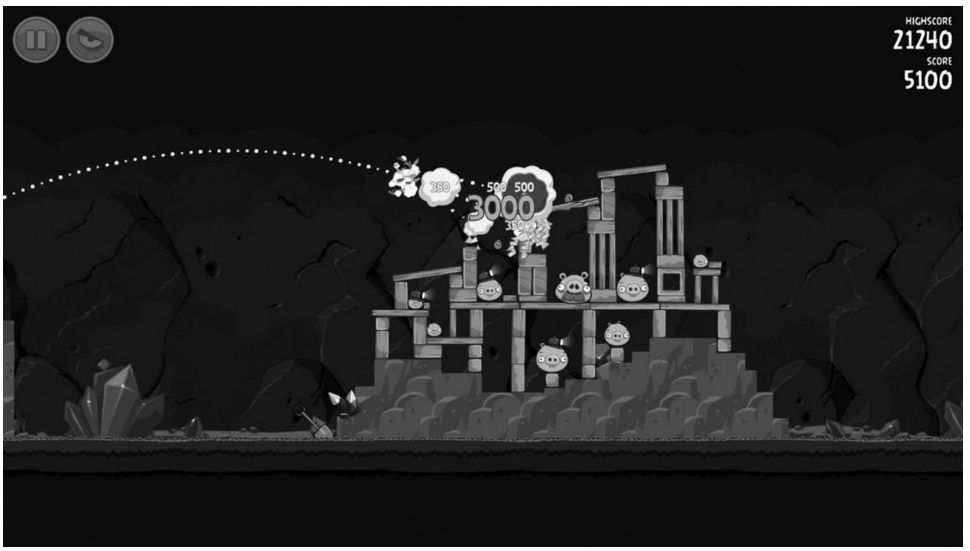

Figure 9-12. *Rovio's Angry Birds requires players to launch projectiles from a slingshot in a virtual world that models gravity, mass, momentum, and object collision detection. The game physics are a fundamental component of the game mechanic and enhance the sense of presence by assigning physical world traits to virtual objects*

The Rigid Shape Bounds and Rigid Shape Impulse projects introduce you to the powerful ability of physics to bring players into the game world. Instead of simply moving the hero character like a screen cursor, the player is now experiencing simulated inertia, momentum, and gravity requiring the same kind of predictive assessments around aiming, timing, and trajectory that would exist when manipulating objects in the physical world, and objects in the game are now colliding in a manner similarly anchored in physical experience. Even though specific values might take a detour from the physical world in a simulated game space (e.g., lower or higher gravity, more or less inertia, and the like), as long as the relationships are consistent and reasonably analogous to our physical experience, presence will typically increase when these effects are added to game objects.

As with earlier projects, it's possible to use basic behaviors to explore new game mechanics. Imagine, for example, a game level where the hero character in the Rigid Shape Impulse project was required to push all the robots into a specific area within a specified time limit while avoiding being hit with the tracking projectiles. Imagine the same level without physics, and it's a different experience (although not necessarily inferior). The Verlet Particles and Particle Emitters projects provide an example where object physics contribute to presence but don't necessarily have a direct impact on the game mechanic. If the tracking projectiles explode on impact without any kind of simulated particle effect (perhaps using a sprite-based animation to show an explosion), it will potentially be less effective as a presence-enhancing effect than a particle simulation based on a physically accurate model of particle dispersion (again, because we experience combustion in the physical world in a way that follows the laws of physics).

At this point in the discussion it's worth stepping back and interrogating the notion that presence is *necessarily* increased by modeling physical world experience in game environments. While modeling physical behavior in virtual environments is a common and high-confidence method of connecting players to game worlds, there are frequently other style choices that can be quite effective at bringing players into the game. In the case of a particle emitter versus an animated sprite to show an explosion, consider a game featuring a comic book style and displaying a "BLAM!" graphic to signal an explosion. Objects don't show the word "BLAM!" when they explode in the physical world, of course, but the highly stylized use of "BLAM!" in the context of a comic book aesthetic can be quite effective at bringing players into the game world. Many AAA studios have embraced the notion that high-fidelity environments featuring physically accurate materials and robust physics are critical for enhancing presence, although the recent surge of independent studios working in a myriad of different styles is challenging this notion.

I left the level design in Chapter 8 with an interesting two-stage mechanic that focused almost exclusively on abstract logical rules and therefore hadn't yet considered elements that would add presence to the experience and bring players into the game world. Recall the current state of the level in Figure 9-13.

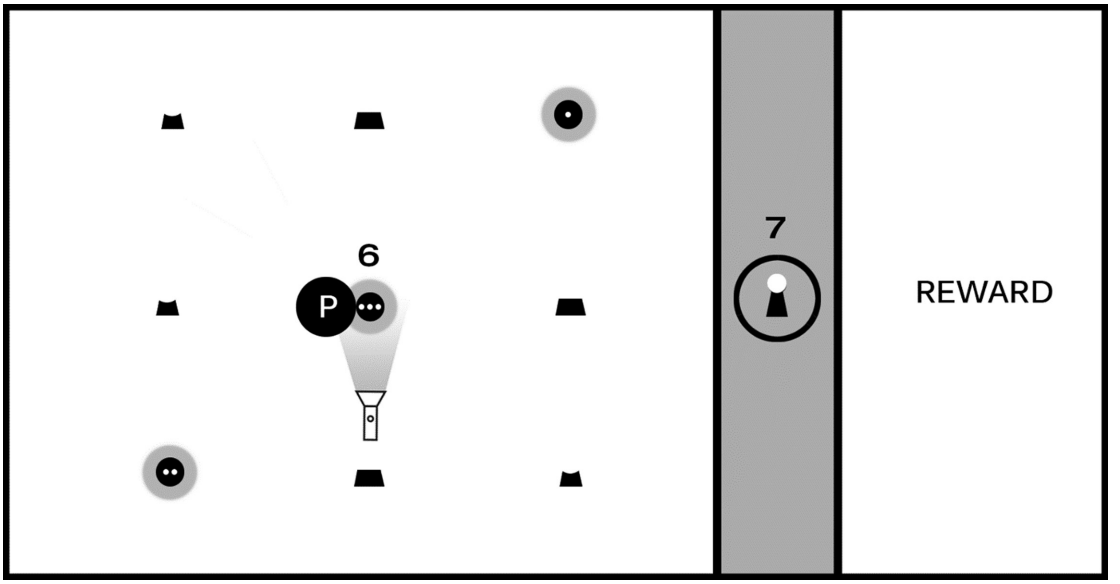

Figure 9-13. *The level as it currently stands includes a two-step puzzle first requiring players to move a flashlight and reveal hidden symbols; the player must then activate the shapes in the correct sequence to unlock the barrier and claim the reward*

There is, of course, some sense of presence conveyed by the current level design. The barrier preventing players from accessing the reward is "impenetrable" and conveyed by a virtual wall, and the flashlight object is "shining" a virtual light beam that reveals hidden clues in the manner perhaps that a UV light in the real world might reveal special ink. Presence is fairly weak at this stage, however, as you have yet to place the game experience in a setting or context and the intentionally generic shapes don't provide the game with any kind of visual identity the player can relate to. If you decided to take this mechanic further, now would be a reasonable time to begin exploring the game's setting. This example used a flashlight metaphor to reveal hidden symbols, but it's now possible to abstract the mechanic's logical rules from the current implementation. This mechanic can be described as follows: "The player must explore the environment to find tools required to assemble a sequence in the correct order."

For the next iteration of the mechanic, let's revisit the interaction model and evolve it from purely a logic puzzle to something a bit more active that makes use of object physics. Figure 9-14 changes the game screen to include a jumping component.

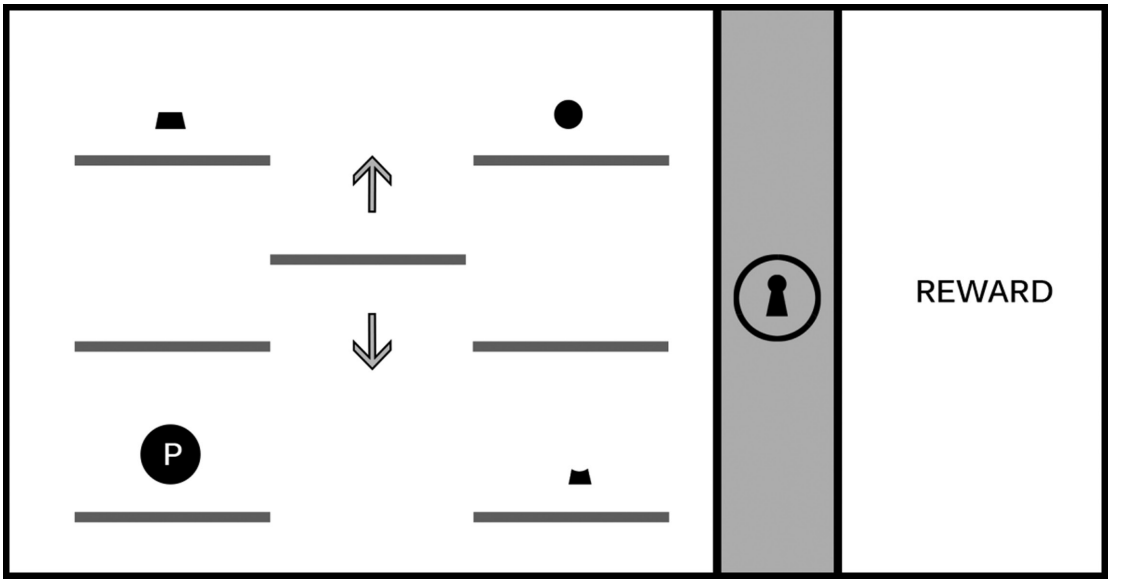

Figure 9-14. *The game screen now shows just one instance of each part of the lock (top, middle, bottom), and the hero character moves in the manner of a traditional jumping 2D platformer. The six platforms on the left and right are stationary, and the middle platform moves up and down, allowing the player to ascend to higher levels. (This image assumes the player is able to "jump" the hero character between platforms on the same level but cannot reach higher levels without using the moving platform.)*

The mechanic is evolving to include a dexterity challenge (in this case, timing the jumps), yet it retains the logical rules from the earlier iteration: the shapes must be activated in the correct order to unlock the barrier blocking the reward. Imagine the player experiences this screen for the first time; they'll begin exploring the screen to learn the rules of engagement for the level, including the interaction model (the keys and/or mouse buttons used to move and jump the hero character), whether missing a jump results in a penalty (for example, the loss of a "life" if the hero character misses a jump and falls off the game screen), and what it means to "activate" a shape and begin the sequence to unlock the barrier.

Figure 9-14 is the beginning of an interesting (although still basic) platformer puzzle, but you've now simplified the solution compared to the earlier iteration, and the platformer jumping component isn't especially challenging. Recall how adding the flashlight in Chapter 8 increased the logical challenge of the original mechanic by adding a second kind of challenge requiring players to identify and use an object in the environment as a tool; you can add a similar second challenge to the platformer component, as shown in Figure 9-15.

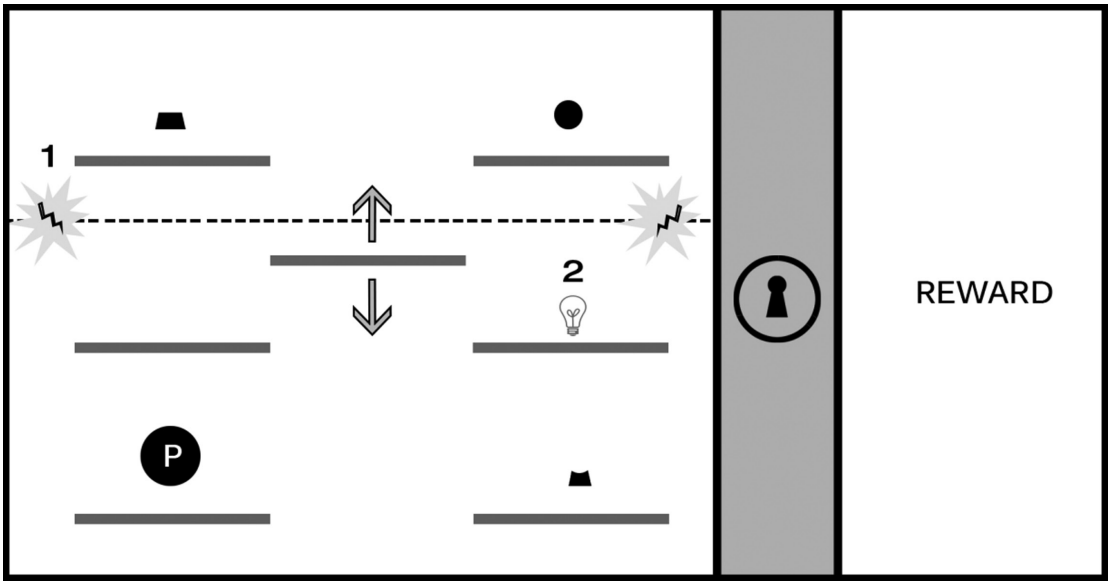

Figure 9-15. *The introduction of a force field blocking access to the upper platforms (#1) can significantly increase the challenge of the platformer component. In this design, the player must activate the switch (represented with a lightbulb in #2) to disable the force field and reach the first and third shapes*

The introduction of a force field opens a number of interesting possibilities to increase the challenge. The player must time the jump from the moving platform to the switch before hitting the force field, and the shapes must be activated in order (requiring the player to first activate top right, then the bottom right, and then the top left). Imagine a time limit is placed on the deactivation when the switch is engaged and that the puzzle will reset if all shapes aren't activated before the force field is reengaged.

You've now taken an elemental mechanic based on a logical sequence and adapted it to support an action platformer experience. At this stage, the mechanic is becoming more interesting and beginning to feel more like a playable level, but it's still lacking setting and context, both of which will influence the further evolution and implementation of this particular mechanic. This is a good opportunity to explore the kind of story you might want to tell with this game. Are you interested in a sci-fi adventure, perhaps a survival horror experience, or maybe a series of puzzle levels with no connected narrative? The setting you choose will not only help inform the visual identity of the game but can also guide your decisions on the kinds of challenges you create for players (that is, are "enemies" in the game working against you, or will you keep the emphasis on solving logic puzzles?). A good exercise to practice connecting a mechanic to a game setting is to pick a place (for example, the interior of a space ship) and begin contextualizing the mechanic in that scenario and defining the elements of the challenge in a way that make sense for the setting. For a spaceship setting, perhaps something has gone wrong and you must make your way from one end of the ship to the other while neutralizing security lasers through the clever use of environment objects. Experiment with applying the spaceship setting to the current mechanic and adjusting the elements in the level to fit that theme: lasers are just one option, but can you think of other uses for the mechanic that don't involve the unlocking sequence? Practice working with the structure of the mechanic from the examples and try applying it to a range of different settings to begin building your comfort for taking abstract mechanics and applying them to specific settings.

Remember, including object physics in your level designs isn't a universal requirement. Game design is infinitely flexible, and sometimes you may want to subvert or completely ignore the laws of physics in the game worlds you create. The final quality of your game experience is the result of how effectively you harmonize all nine elements of game design; it's not about the mandatory implementation of any one design option. Your game might be completely abstract and involve shapes and forms shifting in space in a way that has no bearing on the physical world, or it might take place in an analogous real-world environment; yet, your mechanic may not relay on object physics at all, and your use of color, audio, and narrative might still combine to create an experience with a strong presence for players. However, if you find yourself with a game environment that seeks to convey a sense of physicality by making use of objects that people can relate to objects found in the physical world, it's worth exploring how object physics might enhance the experience.