Simulating the World with RigidShapes

After completing this chapter, you will be able to:

* Recognize the significant computational complexity and cost of simulating real-world physical interactions
* Understand that typical game engine physics components approximate physical interaction based on simple geometries such as circles and rectangles
* Implement accurate collisions of circle and rectangular geometric shapes
* Approximate Newtonian motion formulation with Symplectic Euler Integration
* Resolve interpenetrating collisions based on a numerically stable relaxation method
* Compute and implement responses to collisions that resembles the behavior of rigid bodies in the real-world

# INTRODUCTION

In game engines the functionality of simulating energy transfer is often referred to as physics, physics system, physics component, or, physics engine. Game engine physics components play an important part in many types of games. The range of topics within physics for games is broad and includes but is not limited to areas such as rigid body, soft body, fluid dynamics, or vehicle physics. A believable physical behavior and interactions of game objects have become key elements of many modern PC and console games, and more recently, browser and smartphone games. For example, the bouncing of a ball, the wiggling of a jelly block, the ripples on a lake, or the skidding of a car. The proper simulation and realistic renditions of these are becoming common expectations.

Unfortunately, accurate simulations of the real world can involve details that are overwhelming and require in-depth disciplinary knowledge where the underlying mathematical models can be complicated and the associated computational costs prohibitive. For example, the skid of a car depends on speed, tire properties, etc.; the ripples on a lake depends on the cause, size of the lake, etc.; wiggle of a jelly block depends on density, initial deformation, etc. Even in the very simple case, the bounce a ball depends on its material, state of inflation, and theoretically, even on the particle concentrations of the surrounding air. Modern game engine physics components address these complexities by restricting the types of physical interaction and simplifying the requirements for the simulation computation.

Physics engines typically restrict and simulate isolated types of physical interaction and do not support general combinations of interaction types. For example, properly simulate ball bouncing but do not support the ball colliding and wiggling a jelly block, or, properly simulate ripple effects but do not support ripples caused by a skidding car. That is, typically a rigid body physics engine does not support interactions with soft body object, or fluids, or vehicles; and a soft body physics engine usually does not support interactions with rigid body objects, etc.

Additionally, physics engines typical approximate a vastly simplified interaction model where visually convincing results can be attained. The simplifications are usually in the forms of assumptions on object geometry and physical properties with restrictive interaction rules applied to a selective subset in the game world. For example, a rigid body physics engine typically simplifies the interactions of objects in the following ways:

* assumes objects are continuous geometries with uniformly distributed mass where the center of mass is located at the center of the geometric object,
* approximates object material properties with straightforward bounciness and friction,
* dictates that objects do not change shape during interactions, and,
* limits the simulation to a selective subset of objects in the game scene.

Based on this set of assumptions, a rigid body physics simulation, or a rigid body simulation, is capable of capturing and reproducing many familiar real-world physical interactions such as object bouncing and dropping. For example, a fully inflated bouncing ball where deformation does not occur during collisions, or, a simple Lego block bouncing off of a desk and landing on a hardwood floor. While a relatively small subset of the real-world physical object interaction types, rigid body simulation is the best understood; most straightforward to model, to approximate solutions for, and to implement. This chapter focuses only on rigid body simulation.

Objects with uniformly distributed mass that do not change shape during interactions can be applicable to many important and useful scenarios in games. As the described ball bouncing and Lego block dropping examples, rigid body physics engine is excellent for simulating moving objects coming into contact with one another such as a bowling ball colliding with pins, or, a cannon ball hitting an armored plate. However, it is important to recognize the limitations of rigid shape physics where the assumptions do not support:

* objects consisting of multiple geometric parts, e.g., an arrow;
* objects with non-trivial material properties, e.g., magnetism;
* objects with non-uniform mass distribution, e.g., a baseball bat; or
* objects that change shapes during collision, e.g., rubber balls.

## Implementation Considerations

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 cannot be illuminated. In a similar fashion, the rigid body physics component that you will build in this chapter simulates the transfer of kinetic energy only between selected objects. As in the case of illumination, restricting simulation to a selective collection is the result of optimizing performance. From game designer perspectives, strategic choice of objects is important to convey proper sense of a functioning physical world.

## Chapter Overview

Similar to illumination functionality, 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 rigid body physics component based on the same approach for all the previous game engine components. That is analyzing, understanding, and implementing individual steps to gradually realize the core functionality of the component. In the case of the physics component, the main steps that together implements the rigid body simulation include the following.

* Rigid Shape and Bounds: Defines the RigidShape class to support an optimized simulation by performing computation on separate and simple geometries instead of the potentially complex Renderable objects.
* Collisions of the rigid shapes: Describes the mathematics to accurately collide RigidShape objects. Introduces the CollisionInfo class to
* recognize that in the digital world objects can overlap, introduce CollisionInfo to capture the characteristic of this overlap. Only circles and SAT for rectangles, and circle/rectangle, can be expanded, by introducing new collisions
* Simulate physical motion: formulate physical movement under gravity, derive solution, and implement
* Collision position correction: game loop discrete update, means, objects in motion can overlap, or interpenetrate during a collision, something that must be resolve or corrected.
* Collision resolution: finally responses to collision.

# Rigid Shapes and Bounds

The computation involved in simulating the interactions between arbitrary rigid shapes can be algorithmically complicated and computationally costly. For these reasons, rigid body simulations are typically based on a limited set of simple geometric shapes. For example, rigid circles and rectangles. In typical game engines, these simple rigid shapes can be attached to geometrically complex game objects for an approximated simulation of the physical interactions between those game objects. For example, attaching rigid circles on spaceships and performing rigid body physics simulation of the rigid circles to approximate the physical interactions between the spaceships.

From real-world experience you know that simple rigid shapes can interact with one another only when they come into physical contacts. Algorithmically, this observation is translated into detecting collisions between rigid shapes. For a proper simulation, every shape must be tested for collision with every other shape. In this way, the collision testing is an operation, where is the number of shapes that participate in the simulation. As an optimization for this costly operation, rigid shapes are usually bounded by a simple geometry, e.g., a circle, where the potentially expensive collision computation is only invoked when the bounds of shapes overlap.

## The Rigid Shapes and Bounds Project

This project introduces the RidigShape classes with a simple circular bound for collision optimization. The defined RigidShape class will be integrated into the game engine where each GameObject object will have references to both a Renderable and a RigidShape objects. The Renderable object will be drawn showing the players a visually pleasing gaming element while the RigidShape will be processed participating in the rigid shape simulation approximating the behavior of the GameObject object. You can see an example of this project running in Figure 9-1. The source code to this project is defined in chapter9/9.1.rigid\_shapes\_and\_bounds.



Figure 9-1. Running the Rigid Shapes and Bounds project

The controls of the project are as follows:

* **Behavior control:**

G key: Randomly create a new rigid circle or rectangle

* **Draw control**

T key: Toggle textures on all objects

R key: Toggle the drawing of RigidShape

B key: Toggle the drawing of the bound on each RigidShape

* **Object control:**

Left/right-arrow key: Sequence through and select an object

WASD keys: Move the selected object

Z/X key: Rotate the selected object

Y/U key: Increase/decrease RigidShape size of the selected object, this does not change the size of corresponding Renderable object

The goals of the project are as follows:

* To define and the RigidShape classes and integrate with GameObject.
* To demonstrate that a RigidShape represents a corresponding Renderable geometry on the same GameObject.
* To lay the foundation for building a rigid shape physics simulator.
* To define an initial scene for testing the physics component.

In addition to the system font folder and the particle.png image, you can find the following external resource files in the assets folder.

* minion\_sprite.png for the minion and hero objects,
* platform.png and wall.png that are the horizontal and vertical boarder objects in the test scene, and
* target.png that is being displayed over the currently selected object.

### Setting up Implementation Support

You will begin building this project by first setting up implementation support. First, organize the engine source code structure with new folders in anticipating increases in complexity. Second, define debugging utilities for visualization and verification of correctness. And, third, extend library support for rotating rigid shapes.

#### Organizing the Engine Source Code

In anticipation for the new components, in the src/engine folder create the components folder and move the input.js component source code file into this folder. This folder will contain the source code for physics and other components to be introduced in later chapters. You will have to edit camera\_input.js, loop.js, and index.js to update the source code file location change of input.js.

#### Supporting Debug Drawing

It is important to note that only a Renderable object, typically referenced by a GameObjct, is actually visible in the game world. Rigid shapes do not actually exist in the game world, they are defined to approximate the simulation of physical interactions of corresponding Renderable objects. In order to support proper debugging and verification of correctness, it is important to be able to draw and visualize the rigid shapes.

1. In the src/core folder, create debug\_draw.js, import from LineRenderable, and define supporting constants and variables for drawing simple shapes as line segments.

import LineRenderable from "../renderables/line\_renderable.js";

let kDrawNumCircleSides = 16; // for approx circumference as line segments

let mUnitCirclePos = [];

let mLine = null;

1. Define the init() function to initialize the objects for drawing. The mUnitCirclePos are positions on the circumference of a unit circle, and mLine variable is the line object that will be used for drawing.

function init() {

mLine = new LineRenderable();

mLine.setPointSize(5); // make sure when shown, its visible

let deltaTheta = (Math.PI \* 2.0) / kDrawNumCircleSides;

let theta = deltaTheta;

let i, x, y;

for (i = 1; i <= kDrawNumCircleSides; i++) {

let x = Math.cos(theta);

let y = Math.sin(theta);

mUnitCirclePos.push([x, y]);

theta = theta + deltaTheta;

}

}

1. Define the drawLine(), drawCrossMarker(), drawRectangle(), and drawCircle() functions to draw the corresponding shape based on the defined mLine object. The source code to these functions are not relevant to the physics simulation and are not shown. Please refer to the project source code folder for details.
2. Remember to export the defined functions.

export {

    init,

    drawLine, drawCrossMarker, drawCircle, drawRectangle

}

##### Initialing the Debug Drawing Functionality

Edit loop.js, import from debug\_draw.js and call the init() function after all asynchronous loading promises are fulfilled in start() function.

import \* as debugDraw from "./debug\_draw.js";

… identical to previous code …

async function start(scene) {

… identical to previous code …

// Wait for any async requests before game-load

await map.waitOnPromises();

// With all resources loaded, it is now possible to initialize

// System internal functions that depends on resources (shaders, etc.)

debugDraw.init(); // drawing support for rigid shapes, etc.

… identical to previous code …

}

**Note** A valid alternative for initializing debug drawing is in the createShaders() function of the shader\_resources module after all the shaders are created. However, importing from debug\_draw.js in shader\_resources.js would create a circular import: debug\_draw imports from LineRenderable that attempts to imports from shader\_resources.

#### Updating the gl-matrix Library

Since Renderable can be rotated freely, the rigid shapes that represent these Renderable objects must also be rotated freely. In the case of Renderable objects, the actual rotation is accomplished via vertex multiplication with appropriate transformation matrix in the WebGL vertex shader. For rigid shapes, this rotation must be computed explicitly.

Now, edit src/lib/gl-matrix.js file and define the vec2.rotateWRT() function to support rotating a vertex position, pt, by angle with respect to the ref position.

vec2.rotateWRT = function(out, pt, angle, ref) {

var r=[];

vec2.subtract(r, pt, ref);

vec2.rotate(r, r, angle);

vec2.add(r, r, ref);

out[0] = r[0];

out[1] = r[1];

return r;

};

### Defining the RigidShape Base Class

You are now ready to define RigidShape to be the base class for the rectangle and circle rigid shapes. This base class will encapsulate all the functionality that is common to the two shapes.

1. Start by creating a new subfolder, rigid\_shapes, in src/engine. In this folder, create rigid\_shape.js, import from debug\_draw, and define drawing colors and the RigidShape class.

import \* as debugDraw from "../core/debug\_draw.js";

let kShapeColor = [0, 0, 0, 1];

let kBoundColor = [1, 1, 1, 1];

class RigidShape {

... implementation to follow …

}

export default RigidShape;

1. Define the constructor to include instance variables shared by all subclasses. The xf parameter is typically a reference to the Transform of the Renderable represented by this RigidShape. The mType variable will be initialized by subclasses to differentiate between shape types, e.g., circle vs rectangle. The mBoundRadius is the radius of the circular bound for collision optimization, and mDrawBounds indicates if the circular bound should be drawn.

constructor(xf) {

this.mXform = xf;

this.mType = "";

this.mBoundRadius = 0;

this.mDrawBounds = false;

}

1. Define appropriate getter and setter functions for the instance variables.

getType() { return this.mType; }

getCenter() { return this.mXform.getPosition(); }

getBoundRadius() { return this.mBoundRadius; }

toggleDrawBound() { this.mDrawBounds = !this.mDrawBounds; }

setBoundRadius(r) { this.mBoundRadius = r; }

setTransform(xf) { this.mXform = xf; }

setPosition(x, y) { this.mXform.setPosition(x, y); }

adjustPositionBy(v, delta) {

let p = this.mXform.getPosition();

vec2.scaleAndAdd(p, p, v, delta);

}

\_shapeColor() { return kShapeColor; }

\_boundColor() { return kBoundColor; }

1. Define the boundTest() function to determine if the circular bounds of two shapes have overlapped. Once again, this is a relatively efficient operation designed to precede the costlier accurate collision computation between two shapes.

boundTest(otherShape) {

let vFrom1to2 = [0, 0];

vec2.subtract(vFrom1to2, otherShape.mXform.getPosition(), this.mXform.getPosition());

let rSum = this.mBoundRadius + otherShape.mBoundRadius;

let dist = vec2.length(vFrom1to2);

if (dist > rSum) {

//not overlapping

return false;

}

return true;

}

1. Define the update() and draw() functions. For now update() is empty. When enabled, the draw() function draws the circular bound and a “X” marker at the center of the bound.

update() { // nothing for now }

draw(aCamera) {

if (!this.mDrawBounds)

return;

debugDraw.drawCircle(aCamera, this.mXform.getPosition(), this.mBoundRadius, this.\_boundColor());

debugDraw.drawCrossMarker(aCamera, this.mXform.getPosition(),

this.mBoundRadius \* 0.2, this.\_boundColor());

}

### Defining the RigidRectangle Class

With the abstract base class for rigid shapes defined, you can now create the first concrete rigid shape, the RigidRectangle class. In anticipation of complex collision functions, the implementation source code will be separated into multiple files. For now, create the rigid\_rectangle.js as the access file and import from the rigid\_rectangle\_main.js which will implement the core RigidRectangle functionality.

1. In the src/rigid\_shapes folder, create rigid\_rectangle.js to import from rigid\_rectangle\_main.js and to export the RigidRectangle class. This is the RigidRectangle class access file where users of this class should import from.

import RigidRectangle from "./rigid\_rectangle\_main.js";

export default RigidRectangle;

1. Now, create rigid\_rectangle\_main.js in the src/rigid\_shapes folder to import RigidShape and debugDraw, and define RigidRectangle to be a subclass of RigidShape.

import RigidShape from "./rigid\_shape.js";

import \* as debugDraw from "../core/debug\_draw.js";

class RigidRectangle extends RigidShape {

... implementation to follow …

}

export default RigidRectangle;

1. Define the constructor to initialize the rectangle dimension, mWidth by mHeight, and mType. It is important to recognize that the position and rotation of the rigid rectangle is defined by the Transform referenced by mXform, where the width and height dimensions are defined independently by mWidth and mHeight. This dimension separation allows the designer to determine how tightly a RigidRectangle should wrap the corresponding Renderable. Notice that the actual vertex and face normal of the shape are computed in the setVertices() and computeFaceNormals() functions. The definition of face normal will be detailed in the following steps.

constructor(xf, width, height) {

super(xf);

this.mType = "RigidRectangle";

this.mWidth = width;

this.mHeight = height;

this.mBoundRadius = 0;

this.mVertex = [];

this.mFaceNormal = [];

this.setVertices();

this.computeFaceNormals();

}

1. Define the setVertices() functions. As illustrated in Figure 9-2, the vertices on the rectangle is defined as index 0 being the top-left, 1 being top-right, 2 being bottom-right, and index 3 corresponds to the bottom-left vertex position.

setVertices() {

this.mBoundRadius = Math.sqrt(this.mWidth \* this.mWidth + this.mHeight \* this.mHeight) / 2;

let center = this.mXform.getPosition();

let hw = this.mWidth / 2;

let hh = this.mHeight / 2;

// 0--TopLeft;1--TopRight;2--BottomRight;3--BottomLeft

this.mVertex[0] = vec2.fromValues(center[0] - hw, center[1] - hh);

this.mVertex[1] = vec2.fromValues(center[0] + hw, center[1] - hh);

this.mVertex[2] = vec2.fromValues(center[0] + hw, center[1] + hh);

this.mVertex[3] = vec2.fromValues(center[0] - hw, center[1] + hh);

}



Figure 9-2. The Vertices and Face Normals of a Rectangle.

1. Define the computeFaceNormals() function. Figure 9-2 shows that face normals of a rectangle are vectors that are perpendicular to the edges and pointing away from the center of the rectangle. In addition, notice the relationship between the indices of the face normals and the corresponding vertices. Face normal index-0 points in the same direction as the vector from vertex 2 to 1. This direction is perpendicular to the edge formed by vertices 0 and 1. In this way, face normal of index-0 is perpendicular to the first edge, and so on. Notice that the face normal vectors are normalized with length of 1. The face normal vectors will be used later for determining collisions.

computeFaceNormals() {

// 0--Top;1--Right;2--Bottom;3--Left

// mFaceNormal is normal of face toward outside of rectangle

for (let i = 0; i < 4; i++) {

let v = (i + 1) % 4;

let nv = (i + 2) % 4;

this.mFaceNormal[i] = vec2.clone(this.mVertex[v]);

vec2.subtract(this.mFaceNormal[i], this.mFaceNormal[i], this.mVertex[nv]);

vec2.normalize(this.mFaceNormal[i], this.mFaceNormal[i]);

}

}

1. Define the dimension and position manipulation functions. In all cases, after the manipulation the rectangle vertices and face normals must be re-computed (rotateVertices() calls computeFaceNormals()).

incShapeSizeBy(dt) {

this.mHeight += dt;

this.mWidth += dt;

this.setVertices();

this.rotateVertices();

}

setPosition(x, y) {

super.setPosition(x, y);

this.setVertices();

this.rotateVertices();

}

adjustPositionBy(v, delta) {

super.adjustPositionBy(v, delta);

this.setVertices();

this.rotateVertices();

}

setTransform(xf) {

super.setTransform(xf);

this.setVertices();

this.rotateVertices();

}

rotateVertices() {

let center = this.mXform.getPosition();

let r = this.mXform.getRotationInRad();

for (let i = 0; i < 4; i++) {

vec2.rotateWRT(this.mVertex[i], this.mVertex[i], r, center);

}

this.computeFaceNormals();

}

1. Now, define the draw() function to draw the edges of the rectangle as line segments, and the update() function to update the vertices of the rectangle. The vertices and face normal must be re-computed because, recall from the RigidShape base class constructor discussion, that the mXfrom is a reference to the Transform of a Renderable object, the game may have manipulated the position or the rotation of the Transfrom. To ensure RigidRectangle consistently reflect the potential Transform changes, the vertices and face normals must be re-computed at each update.

draw(aCamera) {

super.draw(aCamera); // the cross marker at the center

debugDraw.drawRectangle(aCamera, this.mVertex, this.\_shapeColor());

}

update() {

super.update();

this.setVertices();

this.rotateVertices();

}

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Defining the Rigid Circle Class

You can now implement the RigidCircle class with a similar overall structure to that of RigidRectangle.

1. In the src/rigid\_shapes folder, create rigid\_circle.js to import from rigid\_circle\_main.js and to export the RigidCircle class. This is the RigidCircle class access file where users of this class should import from.

import RigidCircle from "./rigid\_circle\_main.js";

export default RigidCircle;

1. Now, create rigid\_circle\_main.js in the src/rigid\_shapes folder to import RigidShape and debugDraw, and define RigidCircle to be a subclass of RigidShape.

import RigidShape from "./rigid\_shape.js";

import \* as debugDraw from "../core/debug\_draw.js";

class RigidCircle extends RigidShape {

... implementation to follow …

}

export default RigidCircle;

1. Define the constructor to initialize the circle radius, mRadius, and mType. Similar to the dimension of a RigidRectangle, the radius of RigidCircle is defined by mRadius and is independent from the size defined by the mXfrom. Note that the radii of the RigidCircle, mRadius, and the circular bound, mBoundRadius, are defined separately. This is to ensure future alternatives to separate the two.

constructor(xf, radius) {

super(xf);

this.mType = "RigidCircle";

this.mRadius = radius;

this.mBoundRadius = radius;

}

1. Define the getter and setter of the dimension.

getRadius() { return this.mRadius; }

incShapeSizeBy(dt) {

this.mRadius += dt;

this.mBoundRadius = this.mRadius;

}

1. Define the function to draw the circle as a collection of line segments along the circumference. To properly visualize the rotation of the circle, a bar is drawn from the center to the rotated vertical circumference position.

draw(aCamera) {

let p = this.mXform.getPosition();

debugDraw.drawCircle(aCamera, p, this.mRadius, this.\_shapeColor()); // the circle object

let u = [p[0], p[1] + this.mBoundRadius];

// angular motion

vec2.rotateWRT(u, u, this.mXform.getRotationInRad(), p);

debugDraw.drawLine(aCamera, p, u, false, this.\_shapeColor()); // show rotation

super.draw(aCamera); // draw last to be on top

}

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Modifying the GameObject Class to Integrate RightShape

Recall from the discussions in Chapter 6, the GameObject class is designed to encapsulate the visual appearance and behaviors of objects in the game scene. The visual appearance of a GameObject is defined by the referenced Renderable object. Thus far, the behaviors of a GameObject has been defined and implemented as part of the GameObject class in the forms of ad hoc traveling speed, mSpeed, and simple autonomous behavior, rotateObjPointTo(). You can now replace these ad hoc parameters with the upcoming systematic physics component support.

1. Edit GameObject.js to remove the support for speed, mSpeed, including the corresponding setter and getter functions and the rotateObjPointTo() function. Through the changes in the rest of this chapter, the game object behaviors will be supported by the rigid body physics simulation. Make sure to leave the other variables and functions alone, they are defined to support appearance and to detect texture overlaps, pixelTouches().
2. In the constructor define new instance variables to reference to a RigidShape, and to provide drawing options.

class GameObject {

constructor(renderable) {

this.mRenderComponent = renderable;

this.mVisible = true;

this.mCurrentFrontDir = vec2.fromValues(0, 1); // this is the current front direction of the object

this.mRigidBody = null;

this.mDrawRenderable = true;

this.mDrawRigidShape = false;

}

... implementation to follow …

}

1. Define getter and setter for mRigidBody, and, functions for toggling drawing options.

getRigidBody() { return this.mRigidBody; }

setRigidBody(r) { this.mRigidBody = r; }

toggleDrawRenderable() { this.mDrawRenderable = !this.mDrawRenderable; }

toggleDrawRigidShape() { this.mDrawRigidShape = !this.mDrawRigidShape; }

1. Replace the draw() and update() functions to respect the drawing options, and, to delegate GameObject behavior update to the RigidShape class.

draw(aCamera) {

if (this.isVisible()) {

if (this.mDrawRenderable)

this.mRenderComponent.draw(aCamera);

if ((this.mRigidBody !== null) && (this.mDrawRigidShape))

this.mRigidBody.draw(aCamera);

}

}

update() {

// simple default behavior

if (this.mRigidBody !== null)

this.mRigidBody.update();

}

1. Edit the game\_object\_set.js file to modify the GameObjectSet class to support the toggling of different drawing options for the entire set.

… identical to previous code ...

toggleDrawRenderable() {

let i;

for (i = 0; i < this.mSet.length; i++) {

this.mSet[i].toggleDrawRenderable();

}

}

toggleDrawRigidShape() {

let i;

for (i = 0; i < this.mSet.length; i++) {

this.mSet[i].toggleDrawRigidShape();

}

}

toggleDrawBound() {

let i;

for (i = 0; i < this.mSet.length; i++) {

let r = this.mSet[i].getRigidBody()

if (r !== null)

r.toggleDrawBound();

}

}

### Testing of RigidShape Functionality

RigidShape is designed to approximate and to participate on behalf of a Renderable object in the rigid shape simulation. For this reason, it is essential to create and test different combinations of RigidShape types, circles and rectangles, with combinations of Renderable types including, TextureRenderable, SpriteRenderable, and SpriteAnimateRenderable. The proper functioning of these combinations can demonstrate the correctness of RigidShape implementation and allow visual examination of the appropriateness and limitations of approximating Renderable objects with simple circles and rectangles.

The overall structure of the test program, MyGame, is largely similar to previous projects where the details of the source code can be distracting and is not listed here. Instead the following describes the supporting testing objects and how these objects fulfill the specified requirements. As always, the source code files are located in src/my\_game folder where the supporting object classes are located in src/my\_game/objects folder.

Testing of upcoming collisions require the manipulation of the positions and rotations of each object. The WASDObj class, implemented in wasd\_obj.js, defines the WASD movement and Z/X rotation of a GameObject. The Hero class, a subclass of WASObj implemented in hero.js, is a GameObject with a SpriteRenderable and a RigidRectangle. The Minion class, also a subclass of WASObj in minion.js, is a GameObject with SpriteAnimateRenderable and that is wrapped by either a RigidCircle or a RigidRectangle. Based on these supporting classes, the created Hero and Minion objects will encompass different combinations of Renderable and RigidShape types, and at the same time, allow visual examination of approximating complex texture shapes with simple geometric circles and rectangles.

The vertical and horizontal bounds in the game scene are GameObject instances with TextureRenderable and RigidRectangle created by the wallAt() and platFormAt() functions defined in my\_game\_bounds.js file. The main functionality of constructor, init(), draw(), update(), etc. of MyGame is defined in the my\_game\_main.js file with largely identical functionality as in previous testing projects.

### Observations

You can now run the project and observe the created RigidShape objects. Notice that by default, only RigidShape objects are drawn. You can verify this by typing the T key to toggle on the drawing of the Renderable objects. Notice how the textures of the Renderable objects are bounded by the corresponding RigidShape instances. You can type the R key to toggle off the drawing of the RidigShape objects. Normally, this is what the players of a game will observe, with only the Renderable and without the RigidShape objects being drawn. Since the focus of this chapter are on the rigid shapes and the simulation of their interactions, the default is to show the RigidShape and not the Renderable objects.

Now, type the T and R keys again to toggle back the drawing of RigidShape objects. The B key shows the circular bounds of the shapes. The actual more accurate and costlier collision computations, to be discussed in the next few sections, will only be incurred between objects with these bounds overlap.

You can trying the WASD key to move around the currently selected object, by default the Hero in the center. The Z/X and Y/U keys allow you to rotate and change the dimension of the Hero. Toggle-on the texture, with the T key, to verify that rotation and movement is applied to both the Renderable and the corresponding RigidShape, while the Y/U keys only changes the dimension of the RigidShape. This allows the designer to control how tightly to wrap the Renderable with the corresponding RigidShape. Try typing the left/right-arrow keys to select and work with any of the objects in the scene. Finally, the G key creates new Minion objects with either a RigidCircle or a RigidRectangle.

Lastly, notice that you can move any selected object to any location, including overlapping with another RigidShape object. In the real-world, the overlapping, or interpenetration, of rigid shape objects can never occur while in the simulated digital world this is an issue that must be addressed. Now, with the functionality of the RigidShape classes verified, you can now examine how to compute the collision between these shapes.