Simulating the World with RigidShapes

After completing this chapter, you will be able to:

* Recognize the significant computational complexity and cost of detecting object collisions
* Understand that typical game engine physics components approximate physical interaction based on simple geometries such as circles and rectangles
* Implement accurate collisions between 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 responses of rigid bodies in the real-world

# INTRODUCTION

In games, the functionality of simulating energy transfer is often referred to as physics, physics system, physics component, or, physics engine. Physics engines 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 physics behavior and interactions of game objects has 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 flowing of a river, 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 flow of a river depends on drop, debris, 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 probably even on the particle concentrations of the surrounding air. Modern game engine physics components address these complexities by restricting interaction types and simplifying the simulation computation requirements.

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 hitting and wiggling the jelly block, or, properly simulate water flowing but do not support the water flowing and interfering with a skidding car.

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 among a selective subset in the simulation. For example, a rigid body physics engine typically simplifies the interactions of objects in the following ways:

* Assumes object are continuous geometries with uniformly distributed mass.
* 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 your 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, approximate solutions for and implement. This chapter focuses only on rigid body simulation.

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 using the rigid body physics simulation of the rigid circles to approximate the physical interactions between the spaceships.

Intuitively, you know that two rigid shapes can interact with one another when they come into a physical contact. Algorithmically, this simple intuition translates 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 actual potentially expensive collision computation is only invoked when the bounds of shapes overlap.

## The Rigid Shapes and Bounds Project

This project introduces the RigidShape classes with circular bounds for collision optimization. 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 selected object

Z/X key: Rotate selected object

Y/U key: Increase/decrease RigidShape size of selected object

The goals of the project are as follows:

* To define and the RigidShape classes and integrate with GameObject.
* To lay the foundation for building a rigid shape physics simulator.
* To understand the relationships between RigidShape classes and the engine core functionality.
* 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. The 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, target.png indicating the currently selected object.

You will begin building this project by first organizing the engine for the anticipated increase in source code complexity, and defining debugging utility for visualization and verification of correctness.

### Organizing the Engine

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 upcoming 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 the rigid shape do not actually exist in the game world, they are defined to support the simulation of physical interactions of game objects. Only game objects are actually defined and visible in the game world. 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 from the shader\_resources module after all the shaders are created in the createShaders() function. 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 gl-matrix Library

Since game objects can be rotated freely, the rigid shapes that represent these game objects must also be rotated freely. In the case of GameObject or Renderable objects, the actual rotation is accomplished via vertex multiplying 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 can 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 default 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 shared instance variables. The xf parameter should be a reference to the Transform of the GameObject represented by this RigidShape. The mType variable will be initialized by subclasses to differentiate between a circle and a rectangle. The mBoundRadius is 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 shape overlap. Once again, this is the relatively efficient test for avoiding the more costly 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, and when enabled, draws the circular bound with a “X” marker at the center position.

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

1. In src/rigid\_shapes folder, create rigid\_rectangle.js to import from rigid\_rectangle\_main.js and to export the RigidRectangle class. This is 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 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. 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 next step.

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 corresponds to the top-left, 1 top-right, 2 bottom-right, and index 3 corresponds to the bottom-left vertex position.

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

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);

}

1. Define the computeFaceNormals() function. Figure 9-2 also shows that face normals of a rectangle are vectors that are perpendicular to the edges and point away from the center of the rectangle. Notice that the face normal vectors are normalized with length of 1. In addition, notice the relationship between the rectangle vertices and the corresponding face normals. 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 in the direction that is pointing away from the rectangle and is perpendicular to the first edge, and so on. 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. Notice that 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. Lastly, 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. Recall from the RigidShape base class constructor discussion that the mXfrom is a reference to the Transform instance of a Renderable object, where the game may manipulate and change. 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();

}

### Defining the Rigid Circle Class

You can now implement the rigid circle object based on an overall structure that is similar to that of the rigid rectangle.

1. Under the RigidBody folder, create a new file and name it Circle.js.
2. Edit this file to create a constructor that initializes the radius of the circle, the rigid body type as Circle, and a mStartpoint position for the purpose of drawing a reference line to visualize the rotation angle of a circle. Initially, without rotation, the reference line is vertical connecting the center of the circle to the top of the circumference. Changing the rotation angle of the circle will result in this line being rotated.

var Circle = function (center, radius) {

RigidShape.call(this, center);

this.mType = "Circle";

this.mRadius = radius;

// The start point of line in circle

this.mStartpoint = new Vec2(center.x, center.y - radius);

};

1. Similar to the Rectangle class, you must include the following code to ensure that the Circle class properly inherits from the RigidShape base class.

var prototype = Object.create(RigidShape.prototype);

prototype.constructor = Circle;

Circle.prototype = prototype;

1. Distinct from that of the rectangle, the arc function of the context is used to draw the circle onto the canvas. In addition, you need to draw the rotation reference line from the center to the mStartpoint, the top of the circle.

Circle.prototype.draw = function (context) {

context.beginPath();

//draw a circle

context.arc(this.mCenter.x, this.mCenter.y, this.mRadius, 0, Math.PI \* 2, true);

//draw a line from start point toward center

context.moveTo(this.mStartpoint.x, this.mStartpoint.y);

context.lineTo(this.mCenter.x, this.mCenter.y);

context.closePath();

context.stroke();

### Modifying the GameObject Class to Integrate RightShape

GameObject update

in integrating texture functionality into the engine involves modifying the engine access file, index.js. Edit index.js and add in the following import and export statements to grant client access to this and that

### Testing of RigidShape Functionality

Edit Must test, and in aniticipating for future expansion blah blah blah in the src/my\_game/objects folder:

1. Craete wasd\_obj.js, support convenient keyboard movement for testing purposes. UYPDATE TO SUPPORT CIRCLE or RECTANGLE rigidshape

/\* File: WASD\_Obj.js

\*

\* Defines the keyControl function

\*/

"use strict"; // Operate in Strict mode such that variables must be declared before used!

var kWASDDelta = 0.3;

import engine from "../../engine/index.js";

class WASDObj extends engine.GameObject {

constructor() {

super();

}

keyControl() {

let xform = this.getXform();

if (engine.input.isKeyPressed(engine.input.keys.W)) {

xform.incYPosBy(kWASDDelta);

}

if (engine.input.isKeyPressed(engine.input.keys.S)) {

xform.incYPosBy(-kWASDDelta);

}

if (engine.input.isKeyPressed(engine.input.keys.A)) {

xform.incXPosBy(-kWASDDelta);

}

if (engine.input.isKeyPressed(engine.input.keys.D)) {

xform.incXPosBy(kWASDDelta);

}

if (engine.input.isKeyPressed(engine.input.keys.Z)) {

xform.incRotationByDegree(1);

}

if (engine.input.isKeyPressed(engine.input.keys.X)) {

xform.incRotationByDegree(-1);

}

}

}

export default WASDObj;

1. Edit hero.js, demonstrate support for basic textured Renderable, UPDATE TO create circle/rigidshape

class Hero extends WASDObj {

constructor(spriteTexture) {

super(null);

this.kDelta = 0.3;

this.mRenderComponent = new engine.SpriteRenderable(spriteTexture);

this.mRenderComponent.setColor([1, 1, 1, 0]);

this.mRenderComponent.getXform().setPosition(50, 40);

this.mRenderComponent.getXform().setSize(3, 4);

this.mRenderComponent.setElementPixelPositions(0, 120, 0, 180);

var r = new engine.RigidRectangle(this.getXform(), 3, 4);

this.setRigidBody(r);

this.toggleDrawRenderable();

this.toggleDrawRigidShape();

}

}

export default Hero;

1. Create minion.js test and demonstrate support for sprite animated objects

Now, ready to support creation of test environment. Explain what to be created, explain what is in update. Explain separate source code files. Now start,

1. Create my\_game.js …
2. Thios and that

### Observations

Run and see, try this and that. RigidShape copy You can now run the project to test your implementation. Notice the four bounding borders and the text output to the right that prints instructions for the user and echoes the application state which includes the index of the selected object. Pressing the F or G key generates a rectangle or circle at a random position with a random size. This drawing simulation seems rather similar to the previous project. The main differences are in the object abstraction and drawing mechanism--RigidShape class definition and engine loop monitoring user input and drawing of all defined objects. In the next project you will evolve the engine loop to support the changing of rigid shape states including allowing the user to change the attributes of each of the rigid shapes in the scene and simple simulation of falling objects.

Although, only simple position, orientation, and drawing are supported, these classes represent a well-defined abstraction, hides implementation,details, and thus supports future integration of complexity. In the next section, define collision info and begin with simple collision of circles.

* Simulate gravity that affects all objects in the scene and the ability to toggle gravity on and off
* Optimize object collision detection with broad phase collisions to avoid unnecessary computations.
* Understand that, in a computer simulation, rigid bodies can interpenetrate during a collision and that this interpenetration must be resolved.
* Learn and use the Separating Axis Theorem (SAT) to detect rigid body collisions.
* Compute the necessary information to support efficient. In the next chapter, you will learn about effective resolution of rigid body interpenetration using this computed information.
* Detect collisions between rigid rectangles and circles accurately.
* Understand how to approximate integrals with Euler Method and Symplectic Euler Integration
* Approximate Newtonian motion formulation with Symplectic Euler Integration
* Resolve interpenetrating collisions based on a numerically stable relaxation method

Complete the physics engine in simulating the collisions and responses of rigid circles and rectangles