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 game engine physics components typically approximate physical interaction with 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 fluid dynamics, soft-body, vehicle physics, and rigid body. A believable physics behavior and interactions of game objects has become key elements of many modern PC and console games, more recently, as well as browser and smartphone games. For example, the throwing and flight path of a javelin, the touching and wiggling of a jelly, the skidding and collision of a car with a barrier, or the shooting and collision of a basketball with the backboard. The proper simulation and realistic renditions of these are becoming common expectations.

Unfortunately, the accurate simulations of the real world can involve overwhelming details that require in-depth disciplinary knowledge where the underlying mathematical models can be unwieldy and the associated computational costs prohibitive. Among many other parameters, factors that are obvious but difficult to specify for the given examples include, javelin flight depends on weight distribution on the object and air resistance, jelly wiggle depends on density and initial deformation, and, car skid depends on speed, weight distribution, and much more.

Even in the very simple case, the bouncing of the basketball depends on its inflation state, the materials of the ball and the backboard, and may also be affected by particle concentrations of the surrounding air. For these reasons, game engines typical approximates a vastly simplified solution where visually believable results can be accomplished. The simplifications are usually in the forms of restrictive simulations among a selective subset of objects with approximated properties. For example, with the restriction that objects cannot change shape during interaction, selecting only the ball and the backboard and ignore all other objects in the game scene in the simulation, and, approximating the materials only with bouncing and frictional factors. These are the approach and the core ideas of a rigid body physics component in typical game engines.

This chapter focuses only on rigid body, a single continuous geometry with uniformly distributed mass. Rigid bodies do not change shape during physical interactions.For example, a falling Lego block bouncing off of your desk and landing on a hardwood floor would be considered an interaction between rigid objects. This type of simulation is known as a rigid body physics simulation, or more simply a rigid body simulation. **Rigid body simulation is cool because it can achieve this and that effects. The Lego block bouncing example indicated/illustrate this and that. This is one of the most commonly encountered need in game engine, and, also one of the best understood and thus most straightforward to implement simulation.**

Though powerful and capable of capturing many real-world situations, it is important to recognize the limitations of results from rigid body simulations. This means, important to recognize limitations**.** does not support, any interaction with surrounding media, e.g., air resistance of objects inflight, or in water, arrow in flight, does not support soft-body, e.g., human flesh, or objects that changes shape while in contact, e.g., contact with grass or human flesh, single geometry with heterogenous physical properties, e.g., arrow in-fight where the head and tail has different drag. **While it is true, does not support many, Covers a wide general range of physical world, block drop on to floor, pin-ball, etc. An important observation, accurate simulation of the physicl world is theoretically difficult and computationally expensive. Should use with strategic care and combine with creative design and of art etc. Limitations, arrow in flight due to different air resistance on different part flight changes, boat floating down a river, fast moving vehicle crash into a stationary barrie. Does not work, uniform mass on the entire object, air resistence; fluid, shape deformation. Note, powerful, but, also limited.**

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 physics component that you will learn in this chapter simulates the transfer of kinetic energy 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 functioning physical world. From game developer perspective, similar to extra code to enable illumation computation or shadow simulation, must include requires extra programming code to collect and group objects that participate in physics simulation.**

## 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 physics component based on the same approach for all the previous game engine components. That is, analyzing, understanding and implementing individual modules of the component and building on these modules to gradually realize the core functionality of the component.

**This section outlines the main modules that together implements the rigid body physics simulation.** Introduce separate geometry, colliding the geometries, simulating physical movements, refining and resolving collision, and finally simulating the response as a result of collision.

* Rigid Shape and Bounds: optimize physical simulation, by performing computation on separate geometries instead of the potentially complex renderables. Introduce RigidShape class to wrap, only support circle and rectangle.
* Collisions of the rigid shapes: math to collide, 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 often 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 approximating their physics simulations. 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.

Sentences on the need for bounds to

## The Rigid Shapes and Bounds Project

This project introduces the rigid shapes and the bounds. You can see an example of this project running in Figure 9-X1. The source code to this project is defined in chapter9/9.1.rigid\_shapes\_and\_bounds.

Figure 9-X1. Running the Rigid Shapes and Bounds project

The controls of the project are as follows, for both scenes:

* **This and that**
* **This and that**.

The goals of the project are as follows:

* To define the base class for all rigid shape objects.
* To lay the foundation for building a rigid shape physics simulator.
* To understand the relationships between rigid shape classes and the engine core functionality.
* To define an initial scene for testing your implement.

You can find the following external resource files in the assets folder: this file and tht file

### The Rigid Shape Base Class

You can now define a base class for the rectangle and circle shape objects. This base class will encapsulate all the functionality that is common to the two shapes. MAY NEED TO POINT OUT ROTATION for NORMAL vector. May need to include rotation

1. Start by creating a new subfolder called RigidBody under the SiteRoot (or public\_html) folder. In the RigidBody folder, create a new file and name it RigidShape.js
2. Edit RigidShape.js to define the constructor. For now the constructor only receives one vector argument representing the center of the object. The rotation angle of the rigid shape has a default value of 0. The created object is then be pushed into a global object list, mAllObjects.

function RigidShape(center) {

this.mCenter = center;

this.mAngle = 0;

gEngine.Core.mAllObjects.push(this);

}

### The Rigid Rectangle Class

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

1. Under the RigidBody folder, create a new file and name it Rectangle.js.
2. Edit this file to create a constructor that receives the center, a width and a height properties. In the constructor, define the type of rigid body as Rectangle, allocate an array to store the vertex positions of the rectangle, and a separate array to store the face normal vectors (to be discussed later).

var Rectangle = function (center, width, height) {

RigidShape.call(this, center);

this.mType = "Rectangle";

this.mWidth = width;

this.mHeight = height;

this.mVertex = [];

this.mFaceNormal = [];

};

1. In the constructor, compute the vertex positions of the rectangle using the center, width, and height information.

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

this.mVertex[0] = new Vec2(center.x - width / 2, center.y - height / 2);

this.mVertex[1] = new Vec2(center.x + width / 2, center.y - height / 2);

this.mVertex[2] = new Vec2(center.x + width / 2, center.y + height / 2);

this.mVertex[3] = new Vec2(center.x - width / 2, center.y + height / 2);

1. Next, compute the face normal vectors. As illustrated in Figure 2-2, face normals 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 is 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 index-0 is the direction pointing away from the rectangle that is perpendicular to the first edge, and so on. The face normal vectors will be used later for determining collisions.



*Figure 9-XX. The Face Normals of a Rectangle.*

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

//mFaceNormal is normal of face toward outside of rectangle

this.mFaceNormal[0] = this.mVertex[1].subtract(this.mVertex[2]);

this.mFaceNormal[0] = this.mFaceNormal[0].normalize();

this.mFaceNormal[1] = this.mVertex[2].subtract(this.mVertex[3]);

this.mFaceNormal[1] = this.mFaceNormal[1].normalize();

this.mFaceNormal[2] = this.mVertex[3].subtract(this.mVertex[0]);

this.mFaceNormal[2] = this.mFaceNormal[2].normalize();

this.mFaceNormal[3] = this.mVertex[0].subtract(this.mVertex[1]);

this.mFaceNormal[3] = this.mFaceNormal[3].normalize();

1. Ensure the newly defined Rectangle class properly inherits from the RigidShape base class by including the following code after the constructor.

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

prototype.constructor = Rectangle;

Rectangle.prototype = prototype;

1. Now you can create the draw function for the rectangle object. The strokeRect function of the context, a reference to the canvas, is invoked to accomplish this. Corresponding translation and rotation must be defined in order to draw the rectangle at the proper position and orientation. The implementation is shown as follows.

Rectangle.prototype.draw = function (context) {

context.save();

context.translate(this.mVertex[0].x, this.mVertex[0].y);

context.rotate(this.mAngle);

context.strokeRect(0, 0, this.mWidth, this.mHeight);

context.restore();

};

### 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();

### Integration with the Engine

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