Implementing the 2D Physics Engine Core

In the previous chapter, you implemented functionality to support basic drawing operations. Drawing is the first step to constructing your physics engine because it allows you to observe the output while continue to expand the engine’s capabilities. In this chapter, two critical components for 2D physics simulations, the core engine loop and rigid shape class, will be examined and added to the engine. The core engine loop, or the engine loop, allows the engine to control and handle the real time interaction of game objects, while the rigid shape class abstracts and hides the detailed information such as positions and rotation angles that are required for future physics calculations.

This chapter begins with the brief coverage a simple vector calculation library. It is assumed that you have background in basic vector operations in 2D space, and thus the required code is provided without extensive conceptual explanations. The chapter then introduces you to a rigid shape class, a critical abstraction that will encapsulate all the information of an object that is required in a physics simulation. For example, as will be introduced in the next chapter, information such as width, height, center position, mass, inertia, friction. This information presented through the rigid shape class will be utilized throughout the engine’s evolution into a fully featured 2D game physics library. For this chapter you will begin with the creation the rigid shape class that only contains information for drawing of the object onto the canvas. Lastly, you will be introduced to one of the more important components of the physics engine, the core engine loop.

After completing this chapter, you will be able to:

* Control the position and rotation of objects based on user keyboard input
* Simulate gravity that affects all objects in the scene and the ability to toggle gravity on and off
* Select and display the properties of a specific object
* Reset the scene

# Vector Calculation Library

Physics simulation requires a vector library to represent object positions and orientations, and to support the computations involved in the simulation that changes these quantities. The computation involved in 2D physics simulations are basic vector operations including: addition, subtraction, scaling, cross product, etc. For this reason, you will create a simple Vec2 vector math library to be included in all subsequent projects.

## Creating the Library

In this step, you will create a new file within a new Library folder to support all the required calculations.

1. Create new folder name Lib inside the SiteRoot (or public\_html) folder by right-clicking and creating a new folder.
2. Create a new JavaScript File within the Library folder by right-clicking the Lib folder. Name the file Vec2.js.
3. Open the new Vec2.js file for editing.
4. Add the Vec2 constructor.

var Vec2 = function (x, y) {

this.x = x;

this.y = y;

};

1. Add all the operation functions to support basic vector operations

Vec2.prototype.length = function () {

return Math.sqrt(this.x \* this.x + this.y \* this.y);

};

Vec2.prototype.add = function (vec) {

return new Vec2(vec.x + this.x, vec.y + this.y);

};

Vec2.prototype.subtract = function (vec) {

return new Vec2(this.x - vec.x, this.y - vec.y);

};

Vec2.prototype.scale = function (n) {

return new Vec2(this.x \* n, this.y \* n);

};

Vec2.prototype.dot = function (vec) {

return (this.x \* vec.x + this.y \* vec.y);

};

Vec2.prototype.cross = function (vec) {

return (this.x \* vec.y - this.y \* vec.x);

};

Vec2.prototype.rotate = function (center, angle) {

//rotate in counterclockwise

var r = [];

var x = this.x - center.x;

var y = this.y - center.y;

r[0] = x \* Math.cos(angle) - y \* Math.sin(angle);

r[1] = x \* Math.sin(angle) + y \* Math.cos(angle);

r[0] += center.x;

r[1] += center.y;

return new Vec2(r[0], r[1]);

};

Vec2.prototype.normalize = function () {

var len = this.length();

if (len > 0) {

len = 1 / len;

}

return new Vec2(this.x \* len, this.y \* len);

};

Vec2.prototype.distance = function (vec) {

var x = this.x - vec.x;

var y = this.y - vec.y;

return Math.sqrt(x \* x + y \* y);

};

With these functions defined, it is now possible to operate on vectors to calculate and manipulate the position, size, and orientation of objects drawn on the canvas. It is expected that you understand these elementary operators. Do not forget to include the new library in the project by adding the new file into the index.html using <script> tag like so:

<script type="text/javascript" src="Lib/Vec2.js"></script>

# Physics Engine and Rigid Shapes

This book focuses on primitive objects that do not change shape during their physical interactions, or objects that are rigid. 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.

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.

The physics engine you will build is based on simulating the interactions between rigid circles and rectangles. This simulation consists of four fundamental steps:

1. Implementing motions
2. Detecting collisions
3. Resolving the collisions
4. Deriving responses to the collisions

The rest of this chapter leads you to build the infrastructure to represent simple rigid circles and rectangles. Next chapter presents the intricate details of collision detection, while the last chapter discusses motion approximation, collision resolution, and collision responses.

## The Rigid Shape Project

This project demonstrates how to implement the basic infrastructure to encapsulate the characteristics of a rigid body. You can see an example of this project running in Figure 2-1.

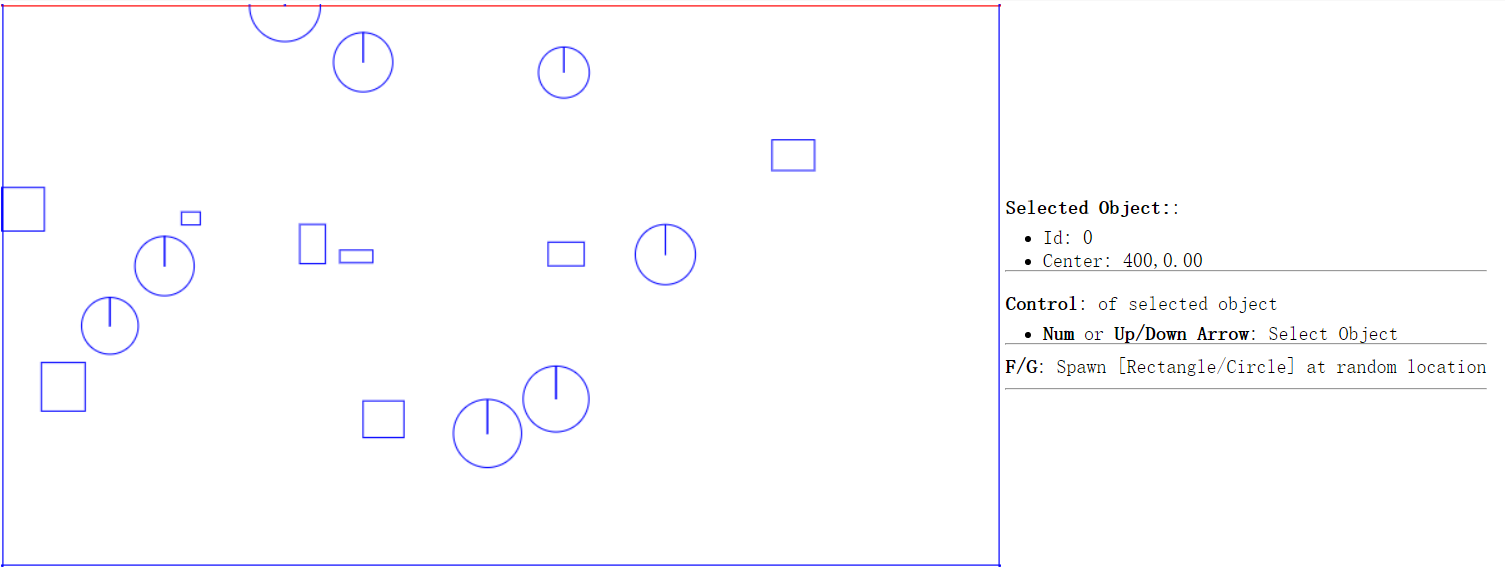


Figure 2-1. Running the Rigid Shape Project.

The source code to this project is defined in the Rigid Shape Project folder.

Project Goals:

* 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.

### The List Object in Engine Core

You will begin by defining a list object, mAllObjects, to keep track of all defined rigid shapes. As you will see in the next chapter, the mAllObjects list allows the simulation of physical interaction among all defined rigid shapes. To conveniently support the simulation computation, the mAllObjects list is defined in the gEngine.Core component.

1. Edit Core.js and add the following line inside gEngine.Core. This creates a list for keeping track of all defined rigid shapes.

var mAllObjects = [];

1. Update the mPublic variable in the Core.js to allow access to the newly define list object. This is accomplished in the following code snippet.

var mPublic = {

mAllObjects: mAllObjects,

mWidth: mWidth,

mHeight: mHeight,

mContext: mContext

};

### 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.

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 2-2. 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();

};

### Modify the User Control Script

You will modify the UserControl.js file for testing the new functionality.

1. Edit the UserControl.js file in the SiteRoot (or public\_html) folder.
2. Add the gObjectNum variable, an index to the mAllObjects array representing the currently selected object. Notice that this variable is defined before the definition of the userControl function and is a variable with global scope.

var gObjectNum = 0;

1. Within the userControl function, define supports for the creation of random rectangles and circles with the F and G keys.

if (keycode === 70) { // f

var r1 = new Rectangle(new Vec2(Math.random()\*width\*0.8, Math.random()\*height\*0.8),

Math.random() \* 30+10, Math.random() \* 30+10);

}

if (keycode === 71) { //g

var r1 = new Circle(new Vec2(Math.random()\*width\*0.8, Math.random()\*height\*0.8),

Math.random() \* 10 + 20);

}

1. Within the userControl function, define supports for selecting an object index based on the up/down arrows and the 0 to 9 keys.

if (keycode >= 48 && keycode <= 57) { //number

if (keycode - 48 < gEngine.Core.mAllObjects.length)

gObjectNum = keycode - 48;

}

if (keycode === 38) { //up arrow

if (gObjectNum > 0)

gObjectNum--;

}

if (keycode === 40) { // down arrow

if (gObjectNum < gEngine.Core.mAllObjects.length-1)

gObjectNum++;

}

### Integrate into the Core

You can now modify the Core.js file to integrate and test the newly defined functionality. Your modification will invoke the drawing of all created rigid shapes, and update the User Interface (UI) to properly reflect the state of the application. For now, the drawing will be accomplished through a simple and continuously loop of calling the appropriate drawing functions, or the engine loop. In the next section of this chapter, you will implement a more advanced engine loop to handle the physics engine’s calculations.

1. Open Core.js in the Engine core folder for editing.
2. Create a new runGameLoop function. In runGameLoop, call the windows.requestAnimationFrame to specify the function for the next frame redraw. Additionally, invoke two other functions, the draw and updateUIEcho functions, to draw all the defined rigid shapes and to receive user keyboard entries.

var runGameLoop = function () {

requestAnimationFrame(function () {

runGameLoop();

})

updateUIEcho();

draw();

};

1. Define the updateUIEcho function to update the HTML to display the proper state of the application.

var updateUIEcho = function () {

document.getElementById("uiEchoString").innerHTML = "<p><b>Selected Object:</b>:</p>" +

"<ul style=\"margin:-10px\">" +

"<li>Id: " + gObjectNum + "</li>" +

"<li>Center: " + mAllObjects[gObjectNum].mCenter.x.toPrecision(3) + "," +

mAllObjects[gObjectNum].mCenter.y.toPrecision(3) + "</li>" +

"</ul> <hr>" + "<p><b>Control</b>: of selected object</p>" +

"<ul style=\"margin:-10px\">" +

"<li><b>Num</b> or <b>Up/Down Arrow</b>: SelectObject</li>" +

"</ul> <hr>" +

"<b>F/G</b>: Spawn [Rectangle/Circle] at random location" + "<hr>";

};

1. Add the draw function to iterate through and invoke the corresponding draw functions of the rigid shapes in the mAllObjects list. The strokeStlye property is set such that only the currently selected object is drawn in red while the rest are in blue.

var draw = function () {

mContext.clearRect(0, 0, mWidth, mHeight);

var i;

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

mContext.strokeStyle = 'blue';

if (i === gObjectNum)

mContext.strokeStyle = 'red';

mAllObjects[i].draw(mContext);

}

};

1. Define support to initialize the engine loop when the script runs for the first time.

var initializeEngineCore = function () {

runGameLoop();

};

1. Allow public access to the initializeEngineCore function by including it in the mPublic variable.

var mPublic = {

initializeEngineCore: initializeEngineCore,

mAllObjects: mAllObjects,

mWidth: mWidth,

mHeight: mHeight,

mContext: mContext

};

### Define the Initial Scene

You can now define a bounded empty environment to test the new functionality.

1. Create new file under the SiteRoot (or public\_html) folder, and name it MyGame.js.
2. Edit this file by creating a new function named MyGame. Inside this function use the new rigid shape object you just implemented to create the four bounds that define border for future physics simulation.

function MyGame() {

var width = gEngine.Core.mWidth;

var height = gEngine.Core.mHeight;

var up = new Rectangle(new Vec2(width / 2, 0), width, 3);

var down = new Rectangle(new Vec2(width / 2, height), width, 3);

var left = new Rectangle(new Vec2(0, height / 2), 3, height);

var right = new Rectangle(new Vec2(width, height / 2), 3, height);

}

Note that you can modify the initial scene by editing this function. This can become handy in the following chapters when you want to test the performance of the physics simulation.

### Modify the index.html File

To include the new functionality, you need to always remember to include and call them inside the index.html file.

1. Open the index.html file for editing.
2. Modify the body tag to support the handling of keyboard events, define the initial testing environment by instantiating a new MyGame object, and initialize the engine loop by calling the initializeEngineCore.

<body onkeydown="return userControl(event);"

onload="var game = new MyGame();

gEngine.Core.initializeEngineCore()">

1. Add a new table row for echoing the application state.

<table style="padding: 2px">

<tr>

<td>

<div><canvas id="canvas"></canvas></div>

</td>

<td>

<div id=”uiEchoString”> </div>

</td>

</tr>

</table>

1. Remember to include all the new scripts with the <script> tag.

<script type="text/javascript" src="RigidBody/RigidShape.js"></script>

<script type="text/javascript" src="RigidBody/Circle.js"></script>

<script type="text/javascript" src="RigidBody/Rectangle.js">

</script><script type="text/javascript" src="EngineCore/Core.js"></script>

<script type="text/javascript" src="MyGame.js"></script>

<script type="text/javascript" src="UserControl.js"></script>

You can now fun the project and test your new object class implementations. It should look like Figure 2-1.

## Observation

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.

# The Core Engine Loop

One of the most important characteristics of any physics engine is the support of seemingly intuitive and continuous interactions between the objects and the graphical simulation elements. In reality, these interactions are implemented as a continuous running loop that receives and processes the calculations, updates the object states, and renders the objects. This constantly running loop is referred to as the *engine loop*.

To convey the proper sense of intuitiveness, each cycle of the engine loop must be completed within a normal human’s reaction time. This is often referred to as *real time*, which is the amount of time that is too short for humans to detect visually. Typically, real-time can be achieved when the engine loop is running at a rate of higher than 40 to 60 cycles in a second. Since there is often one drawing operation in each loop cycle, the loop cycle’s rate can also be expressed as frames per second (FPS), or the *frame rate*. An FPS of 60 is a good target for performance. This is to say, your engine must process calculations, update the object states, and then draw the canvas all within 1/60th of a second!

The loop itself, including the implementation details, is the most fundamental control structure for an engine. With the main goal of maintaining real-time performance, the details of an engine loop’s operation are of no concern to the rest of the physics engine. For this reason, the implementation of an engine loop should be tightly encapsulated in the core of the engine with its detailed operations hidden from other elements.

## Engine Loop Implementations

An engine loop is the mechanism through which logic and drawing are continuously executed. A simple engine loop consists of processing the input, updating the state of objects, and drawing those objects, as illustrated in the following pseudocode:

initialize();

while(game running) {

input();

update();

draw();

}

As discussed, an FPS of 60 or higher is ideal to maintain the sense of real-time interactivity. When the game complexity increases, one problem that may arise is when sometimes a single loop can take longer than 1/60th of a second to complete, causing the game to run at a reduced frame rate. When this happens, the entire game will appear to slow down. A common solution is to prioritize which operations to emphasis and which to skip. Since correct input and updates are required for an engine to function as designed, it is often the draw operation that is skipped when necessary. This is referred to as *frame skipping*, and the following pseudocode illustrates one such implementation:

elapsedTime = now;

previousLoop = now;

while(game running) {

elapsedTime += now - previousLoop;

previousLoop = now;

input();

while( elapsedTime >= UPDATE\_TIME\_RATE ) {

update();

elapsedTime -= UPDATE\_TIME\_RATE;

}

draw();

}

In the previous pseudocode listing, UPDATE\_TIME\_RATE is the required real-time update rate. When the elapsed time between the engine loop cycle is greater than the UPDATE\_TIME\_RATE, the update function will be called until it is caught up. This means that the draw operation is essentially skipped when the engine loop is running too slowly. When this happens, the entire game will appear to run slowly, with lagging play input response and frames skipped. However, the game logic will continue to be correct.

Notice that the while loop that encompasses the update function call simulates a fixed update time step of UPDATE\_TIME\_RATE. This fixed time step update allows for a straightforward implementation in maintaining a deterministic game state.

## The Core Engine Loop Project

This project demonstrates how to incorporate a loop into your engine and to support real-time simulation by updating and drawing the objects accordingly. You can see an example of this project running in Figure 2-3. The source code to this project is defined in the Core Engine Loop Project folder.

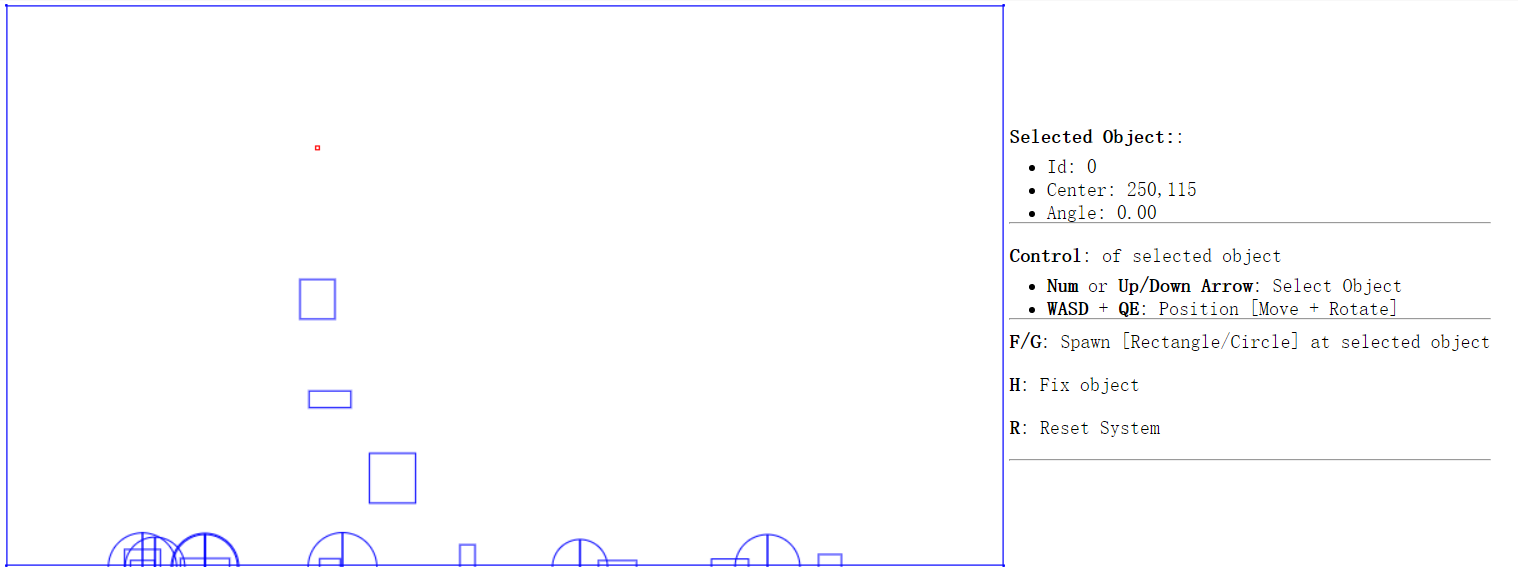


Figure 2-3. Running the Core Engine Loop Project.

The goals of the project are as follows:

* To understand the internal operations of an engine loop
* To implement and encapsulate the operations of an engine loop
* To gain experience with continuous update and draw to simulate animation

### Implement the Engine Loop Component

The engine loop component is a core engine functionality and thus should be implemented as a property of the gEngine.Core. The actual implementation is similar to the pseudocode listing discussed.

1. Edit the Core.js file.
2. Add the necessary variables to determine the loop frequency.

var mCurrentTime, mElapsedTime, mPreviousTime = Date.now(), mLagTime = 0;

var kFPS = 60; // Frames per second

var kFrameTime = 1 / kFPS;

var mUpdateIntervalInSeconds = kFrameTime;

var kMPF = 1000 \* kFrameTime; // Milliseconds per frame.

1. Update the runGameLoop function to keep track of the elapsed time between frames and to ensure that the update function is called at the frame rate frequency.

var runGameLoop = function () {

requestAnimationFrame(function () {

runGameLoop();

});

//compute how much time has elapsed since the last RunLoop was executed

mCurrentTime = Date.now();

mElapsedTime = mCurrentTime - mPreviousTime;

mPreviousTime = mCurrentTime;

mLagTime += mElapsedTime;

// Make sure to update the game the appropriate number of times.

// Update only every Milliseconds per frame.

// If lag larger then update frames, update until caught up.

while (mLagTime >= kMPF) {

mLagTime -= kMPF;

update();

}

updateUIEcho();

draw();

};

1. Modify the updateUIEcho function to print out additional relevant application state information like how to rotate and move the selected rigid shape. The code in bold is the only addition to the function.

var updateUIEcho = function () {

document.getElementById("uiEchoString").innerHTML =

// … identical to previous project

mAllObjects[gObjectNum].mCenter.y.toPrecision(3) + "</li>" +

"<li>Angle: " + mAllObjects[gObjectNum].mAngle.toPrecision(3) + "</li>" +

"</ul> <hr>" +

"<p><b>Control</b>: of selected object</p>" +

"<ul style=\"margin:-10px\">" +

"<li><b>Num</b> or <b>Up/Down Arrow</b>: SelectObject</li>" +

"<li><b>WASD</b> + <b>QE</b>: Position [Move + Rotate]</li>" +

"</ul> <hr>" +

"<b>F/G</b>: Spawn [Rectangle/Circle] at selected object" +

"<p><b>H</b>: Fix object</p>" +

"<p><b>R</b>: Reset System</p>" +

"<hr>";

};

1. Create a new function named update, which will call the update function of every rigid shape defined.

var update = function () {

var i;

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

mAllObjects[i].update(mContext);

}

};

### Extend the Rigid Shape Classes

You are going to modify the rigid shape base class, and the both Rectangle and Circle classes to support the implementation of simple behavior. While the update function is defined in the rigid shape base class to be invoked by the game engine loop, the detailed implementation of update must necessarily be subclass specific. For instance, a circle object implements moving behavior by changing the values in its center while a rectangle object must change all of the values in the vertex and face normal arrays to simulate the same movement behavior.

#### Rigid Shape Base Class

1. Edit the RigidShape.js file.
2. Define the update function to be called by the engine loop and implement the simple falling behavior by changing the center position with a constant y-direction vector. Notice that the free fall behavior is only applied when the shape is within the vertical bounds of the canvas.

RigidShape.prototype.update = function () {

if (this.mCenter.y < gEngine.Core.mHeight && this.mFix !== 0)

this.move(new Vec2(0, 1));

};

Subclasses are responsible for defining the mFix variable and the move function to control if the shape is fixed where it should not follow the falling behavior and to implement the moving of the shape. It should be emphasized that this rigid shape movement behavior is included here for testing purposes only and will be removed in the next project. Actual physics based movement of rigid shape objects and the associated physical quantities including velocity and acceleration will be introduced and discussed in Chapter 4.

Note that by default the canvas coordinate defines the origin, (0, 0), to be located at the top left corner, and positive y direction to be downwards. For this reason, to simulate gravity, you will move all object in the positive y-direction.

#### The Circle Class

The Circle class is modified to implement movements.

1. Edit the Circle.js file.
2. Define the mFix instance variable to enable or disable the falling behavior.

var Circle = function (center, radius, fix) {

// … code similar to previous project

this.mFix = fix;

// … code similar to previous project

1. Add a move function to define how a circle is moved by a vector--adding the movement vector to the center and the mStartpoint.

Circle.prototype.move = function (s) {

this.mStartpoint = this.mStartpoint.add(s);

this.mCenter = this.mCenter.add(s);

return this;

};

1. Add rotate function to implement the rotation of a circle. Note that since a circle is infinitely symmetrical, a rotated circle would appear identical to the original shape. The mStartpoint position allows a rotated reference line to be drawn to indicate angle of rotation of a circle.

// rotate angle in counterclockwise

Circle.prototype.rotate = function (angle) {

this.mAngle += angle;

this.mStartpoint = this.mStartpoint.rotate(this.mCenter, angle);

return this;

};

#### The Rectangle Class

Similar to the circle class, the Rectangle class must be modified to support the new functionality.

1. Edit the Rectangle.js file.
2. Define the mFix instance variable to enable or disable the falling behavior.

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

// … code similar to previous project

this.mFix = fix;

// … code similar to previous project

1. Define the move function by changing the values of all vertices and the center.

Rectangle.prototype.move = function (v) {

var i;

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

this.mVertex[i] = this.mVertex[i].add(v);

}

this.mCenter = this.mCenter.add(v);

return this;

};

1. Define the rotate function by rotating all over the vertices and recomputing the face normals.

Rectangle.prototype.rotate = function (angle) {

this.mAngle += angle;

var i;

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

this.mVertex[i] = this.mVertex[i].rotate(this.mCenter, angle);

}

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();

return this;

};

### Modify User Control Script

You will need to extend the userControl function defined in the UserControl.js file to support movements, rotation, disable/enable gravity, and reset the entire scene.

1. Edit the UserControl.js file.
2. Add statements to support moving, rotating, and toggling of gravity on the selected object.

// move with WASD keys

if (keycode === 87) { //W

gEngine.Core.mAllObjects[gObjectNum].move(new Vec2(0, -10));

}

if (keycode === 83) { // S

gEngine.Core.mAllObjects[gObjectNum].move(new Vec2(0, +10));

}

if (keycode === 65) { //A

gEngine.Core.mAllObject[gObjectNum].move(new Vec2(-10, 0));

}

if (keycode === 68) { //D

gEngine.Core.mAllObjects[gObjectNum].move(new Vec2(10, 0));

}

// Rotate with QE keys

if (keycode === 81) { //Q

gEngine.Core.mAllObjects[gObjectNum].rotate(-0.1);

}

if (keycode === 69) { //E

gEngine.Core.mAllObjects[gObjectNum].rotate(0.1);

}

// Toggle gravity with the H key

if (keycode === 72) { //H

if(gEngine.Core.mAllObjects[gObjectNum].mFix === 0)

gEngine.Core.mAllObjects[gObjectNum].mFix = 1;

else gEngine.Core.mAllObjects[gObjectNum].mFix = 0;

}

1. Add a statement to reset the scene.

if (keycode === 82) { //R

gEngine.Core.mAllObjects.splice(5, gEngine.Core.mAllObjects.length);

gObjectNum = 0;

}

1. Modify object creation statements of the G and F keys such that the new object is created at the location of the currently selected object, rather than a random position.

if (keycode === 70) { //f

var r1 = new Rectangle(new Vec2(gEngine.Core.mAllObjects[gObjectNum].mCenter.x,

gEngine.Core.mAllObjects[gObjectNum].mCenter.y),

Math.random() \* 30 + 10, Math.random() \* 30 + 10);

}

if (keycode === 71) { //g

var r1 = new Circle(new Vec2(gEngine.Core.mAllObjects[gObjectNum].mCenter.x,

gEngine.Core.mAllObjects[gObjectNum].mCenter.y),

Math.random() \* 10 + 20);

}

### Update the Scene

To test the implemented engine loop and object movements, you will create an initial selected object to the scene. This initial object will be serve as the cursor position for spawning created rigid shapes. This can be accomplished by editing the MyGame.js file and creating an initial object.

function MyGame() {

var width = gEngine.Core.mWidth;

var height = gEngine.Core.mHeight;

var r1 = new Rectangle(new Vec2(width / 2, height / 2), 3, 3, 0);

var up = new Rectangle(new Vec2(width / 2, 0), width, 3, 0);

var down = new Rectangle(new Vec2(width / 2, height), width, 3, 0);

var left = new Rectangle(new Vec2(0, height / 2), 3, height, 0);

var right = new Rectangle(new Vec2(width, height / 2), 3, height, 0);

}

## Observation

Run the project to test your implementation. You will see that the scene is almost the same to that of the previous project except for the small initial cursor object. See that you can change the selected object, and thereby the cursor object, with the 0 to 9, or the up and down arrow keys. Type F and G keys to see that new objects are created at the cursor object location and they always follow the falling behavior. This real-time smooth falling behavior indicates that the engine loop has been successfully implemented. You play around with its position with the WASD, QE, and H keys move, rotate and toggle gravity on the selected object. You may also notice that without movement of the cursor object, the newly created objects are clustered together which can be confusing. That is because the physics simulation has yet to be defined. In the next chapter you will learn about and implement collision detection as a first step to remedy the clustered object problem.

# Summary

In this chapter, you have implemented basic rigid body classes. 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 chapter, you will learn about other physical quantities including mass, inertia, friction, and restitution. The engine loop project introduced you to the basics of a continuous update loop that supports real time per-shape computation and enables visually appealing physics simulations. In the next chapter you will begin learning about physics simulation by first examine the collision between rigid shapes in detail.