Physics

# Movement

Movement is the description of how object positions change in the simulated world. Mathematically, movement can be formulated in many ways. In Chapter 6, you experienced working with movement where you continuously accumulated a velocity to an object’s position. As illustrated in Figure 9-19 and the following equation, you have been working with describing movement based on constant displacements.



Figure 9-19. Movement Based on Constant Displacements

A movement that is governed by the constant displacement formulation becomes restrictive when it is necessary to change the amount that is displaced over time. Newtonian mechanics address this restriction by considering time in the movement formulations, as seen in the following equations.

These two equations implement a Newtonian based movement where is the velocity that describes the change in position over time and is the acceleration that describes the change in velocity over time.

Notice that both velocity and acceleration are vector quantities encoding both the magnitude and direction. The magnitude of a velocity vector defines the speed, and the normalized velocity vector identifies the direction that the object is traveling. An acceleration vector lets you know whether an object is speeding up or slowing down and the change of travelling direction. Acceleration is changed by the forces acting upon an object. For example, if you were to throw a ball into the air, the gravitational force would affect the object’s acceleration over time, which in turn would change the object’s velocity.

## Explicit Euler Integration

The Euler method, or Explicit Euler Integration, approximates integrals based on initial values. This is one of the most straightforward and thus a good beginning point to learn about integration approximation. As illustrated in the following two equations, in the case of the Newtonian movement formulation, the new velocity, , of an object can be approximated as the current velocity, , plus the current acceleration, , multiplied by the elapsed time. Similarly, the object’s new position, , can be approximated by the object’s current position, , plus the current velocity, , multiplied by the elapsed time.

The left diagram of Figure 9-20 illustrates a simple example of approximating movements with Explicit Euler Integration. Notice that the new position, , is computed based on the current velocity, . While the new velocity, , is computed to move the position for the next update cycle.



Figure 9-20. Explicit (Left) and Symplectic (Right) Euler Integration

## Symplectic Euler Integration

You will implement the Semi-Implicit Euler Integration or Symplectic Euler Integration, where intermediate results are used in subsequent approximations. The following equations show Symplectic Euler Integration. Notice that it is nearly identical to the Euler Method except that the new velocity, , is being used when calculating the new position, . This essentially means that the velocity for the next frame is being used to calculate the position of this frame.

The right diagram of Figure 9-20 illustrates that with the Symplectic Euler Integration, the new position is computed based on the newly computed velocity, .

## The Rigid Shape Movements Project

You are now ready to implement Symplectic Euler Integration to approximate movements. The fixed time step, , formulation conveniently allows the integral to be evaluated once per update cycle. This project will guide you through completing the rigid shape component to support movement calculations. You can see an example of this project running in Figure 9-21. The source code to this project is defined in chapter9/9.5.rigid\_shape\_movements.

In addition to implementing Symplectic Euler Integration, this project also guides you to define attributes required for collision simulation and response, such as mass, inertia, friction, and restitution. As will be explained, each of these attributes will play a part in the simulation of object movements and collision responses. This straightforward information is presented here to avoid distracting the discussions of the more complex concepts to be covered in the subsequent projects.



Figure 9-21. Running the Rigid Shape Movements project

The controls of the project are identical to the previous project with a single addition of the V and H key commands in behavior control:

* **Behavior control:**

**V key**: Toggle motion of all objects

**H key**: Inject random velocity to all objects

G key: Randomly create a new rigid circle or rectangle

* **Draw control**

C key: Toggle the drawing of all CollisionInfo

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 complete the implementation of RigidShape classes to include relevant physical attributes
* To implement movements based on Symplectic Euler Integration

In the following, you will first define relevant physical attributes to complete the RigidShape implementation and then focus on integrating Symplectic Euler Integration support for approximating movements.

### Completing the RigidShape Implementation

As mentioned, in order to allow focused discussions of the more complex concepts in the later sections, the attributes for supporting collisions and the corresponding supporting functions are introduced in this project. These attributes are defined in the rigid shape classes.

#### Modifying the RigidShape Class

Edit rigid\_shape.js in the src/engine/rigid\_shape folder.

1. In the constructor of the RigidShape class, define variables representing acceleration, velocity, angular velocity, mass, rotational inertia, restitution (bounciness), and friction. Notice that the inverse of the mass value is actually stored for computation efficiency (by avoiding an extra division during each update calculation). Additionally, notice that a mass of zero is used to represent a stationary object.

class RigidShape {

constructor(xf) {

this.mXform = xf;

this.mAcceleration = physics.getSystemAcceleration();

this.mVelocity = vec2.fromValues(0, 0);

this.mType = "";

this.mInvMass = 1;

this.mInertia = 0;

this.mFriction = 0.8;

this.mRestitution = 0.2;

this.mAngularVelocity = 0;

this.mBoundRadius = 0;

this.mDrawBounds = false;

}

1. Define the setMass() function to set the mass of the object. Once again, for computational efficiency the inversed of the mass is store. Setting the mass of an object to zero or negative is a signal that the object is stationary with zero acceleration and will not participate in any movement computation. Notice that when the mass of an object is changed you would need to call updateInertia() to update its rotational inertia, mInertial. Rotational inertia is geometric shape specific and that the implementation of updateIntertia() function is a subclass specific responsibility.

setMass(m) {

if (m > 0) {

this.mInvMass = 1 / m;

this.mAcceleration = physics.getSystemAcceleration();

} else {

this.mInvMass = 0;

this.mAcceleration = [0, 0]; // to ensure object does not move

}

this.updateInertia();

}

1. Define getter and setter functions for all of the other corresponding variables.

getInvMass() { return this.mInvMass; }

getInertia() { return this.mInertia; }

setInertia(i) { this.mInertia = i; }

getFriction() { return this.mFriction; }

setFriction(f) { this.mFriction = f; }

getRestitution() { return this.mRestitution; }

setRestitution(r) { this.mRestitution = r; }

getAngularVelocity() { return this.mAngularVelocity; }

setAngularVelocity(w) { this.mAngularVelocity = w; }

setAngularVelocityDelta(dw) { this.mAngularVelocity += dw; }

getVelocity() { return this.mVelocity; }

setVelocity(x, y) {

this.mVelocity[0] = x;

this.mVelocity[1] = y;

}

flipVelocity() {

this.mVelocity[0] = -this.mVelocity[0];

this.mVelocity[1] = -this.mVelocity[1];

}

getAcceleration() { return this.mAcceleration; }

setAcceleration(x, y) {

this.mAcceleration[0] = x;

this.mAcceleration[1] = y;

}

1. For the convenience of debugging, define a function, getCurrentState(), to retrieve variable values as text, and a function, userSetsState(), allow a user to set the variables.

getCurrentState() {

let m = this.mInvMass;

if (m !== 0)

m = 1 / m;

return "M=" + m.toFixed(kPrintPrecision) +

"(I=" + this.mInertia.toFixed(kPrintPrecision) + ")" +

" F=" + this.mFriction.toFixed(kPrintPrecision) +

" R=" + this.mRestitution.toFixed(kPrintPrecision);

}

userSetsState() {

// keyboard control

let delta = 0;

if (input.isKeyPressed(input.keys.Up)) {

delta = kRigidShapeUIDelta;

}

if (input.isKeyPressed(input.keys.Down)) {

delta = -kRigidShapeUIDelta;

}

if (delta !== 0) {

if (input.isKeyPressed(input.keys.M)) {

let m = 0;

if (this.mInvMass > 0)

m = 1 / this.mInvMass;

this.setMass(m + delta \* 10);

}

if (input.isKeyPressed(input.keys.F)) {

this.mFriction += delta;

if (this.mFriction < 0)

this.mFriction = 0;

if (this.mFriction > 1)

this.mFriction = 1;

}

if (input.isKeyPressed(input.keys.R)) {

this.mRestitution += delta;

if (this.mRestitution < 0)

this.mRestitution = 0;

if (this.mRestitution > 1)

this.mRestitution = 1;

}

}

}

#### Modifying the RigidCircle Classes

As mentioned, the rotational inertia, mInertial, is a specific to geometric shape and must be modified by the corresponding classes.

1. Edit rigid\_circle\_main.js in the src/engine/rigid\_shapes folder to modify the RigidCircle class to define the updateInertia() function. This function calculates the rotational inertia of a circle when its mass has changed.

updateInertia() {

if (this.mInvMass === 0) {

this.mInertia = 0;

} else {

// this.mInvMass is inverted!!

// Inertia=mass \* radius^2

// 12 is a constant value that can be changed

this.mInertia = (1 / this.mInvMass) \* (this.mRadius \* this.mRadius) / 12;

}

};

1. Update the RigidCircle constructor and incShapeSize() function to call the updateInertia() function.

constructor(xf, radius) {

super(xf);

… identical to previous code …

this.updateInertia();

}

incShapeSizeBy(dt) {

… identical to previous code …

this.updateInertia();

}

#### Modifying the RigidRectangle Classes

Modifications similar to the RigidCircle class must be defined for the RigidRectangle class.

1. Edit rigid\_rectangle\_main.js in the src/engine/rigid\_shapes folder to define the updateInertia() function.

updateInertia() {

// Expect this.mInvMass to be already inverted!

if (this.mInvMass === 0)

this.mInertia = 0;

else {

//inertia=mass\*width^2+height^2

this.mInertia = (1 / this.mInvMass) \* (this.mWidth \* this.mWidth + this.mHeight \* this.mHeight) / 12;

this.mInertia = 1 / this.mInertia;

}

};

1. Similar to the RigidCircle class, update the constructor and incShapeSize() function to call the updateInertia() function.

constructor(xf, width, height) {

super(xf);

… identical to previous code …

this.updateInertia();

}

incShapeSizeBy(dt) {

… identical to previous code …

this.updateInertia();

}

### Defining System Acceleration and Motion Control

With the RigidShape implementation completed, you are now ready to define support for movement approximation.

Define a system-wide acceleration and motion control by adding appropriate variables and access functions to physics.js in the src/engine/components folder. Remember to export the newly defined functionality.

let mSystemAcceleration = [0, -20]; // system-wide default acceleration

let mHasMotion = true;

// getters and setters

function getSystemAcceleration() { return vec2.clone(mSystemAcceleration); }

function setSystemAcceleration(x, y) {

mSystemAcceleration[0] = x;

mSystemAcceleration[1] = y;

}

function getHasMotion() { return mHasMotion; }

function toggleHasMotion() { mHasMotion = !mHasMotion; }

… identical to previous code …

export {

// Physics system attributes

getSystemAcceleration, setSystemAcceleration,

getHasMotion, toggleHasMotion,  
  
 … identical to previous code …

}

### Accessing the Fixed Time Interval

In your game engine the fixed time step, , is simply the time interval of the game loop updates. Now, edit loop.js in the src/engine/core folder to define and export the update time interval.

onst kUPS = 60; // Updates per second

const kMPF = 1000 / kUPS; // Milliseconds per update.

const kSPU = 1/kUPS; // seconds per update

… identical to previous code …

function getUpdateIntervalInSeconds() { return kSPU; }

… identical to previous code …

export {getUpdateIntervalInSeconds}

### Implementing Symplectic Euler Integration in the RigidShape class

You can now integrate Symplectic Euler Integration movement approximation into the rigid shape classes. Since this movement behavior is common to all types of rigid shapes, the implementation should be located in the base class, RigidShape.

1. In the src/engine/rigid\_shapes folder, edit rigid\_shape.js to define the travel() function to implement Symplectic Euler Integration for movement. Notice how the implementation closely follows the listed equations where the updated velocity is used for computing the new position. Additionally, notice the similarity between linear and angular motion where the location (either a position or an angle) is updated by a displacement that is derived from the velocity and time step. Rotation will be examined in detailed in the last section of this chapter.

travel() {

let dt = loop.getUpdateIntervalInSeconds();

// update velocity by acceleration

vec2.scaleAndAdd(this.mVelocity, this.mVelocity, this.mAcceleration, dt);

// p = p + v\*dt with new velocity

let p = this.mXform.getPosition();

vec2.scaleAndAdd(p, p, this.mVelocity, dt);

this.mXform.incRotationByRad(this.mAngularVelocity \* dt);

}

1. Modify the update() function to invoke travel() when the object is not stationary, mInvMass of 0, and when motion of the physics component is switched on.

update() {

if (this.mInvMass === 0)

return;

if (physics.getHasMotion())

this.travel();

}

### Modifying MyGame to Test Movements

The modification to the MyGame class involves supporting new user commands for toggling system-wide motion, injecting random velocity, and, setting the scene stationary boundary objects to rigid shape with zero mass. The injecting of random velocity is implemented by the randomizeVelocity() function defined in my\_game\_bounds.js file.

All updates to the MyGame class are straightforward. To avoid unnecessary distraction, the details are not shown. As always, you can refer to the source code files in the src/my\_game folder for implementation details.

### Observations

You can now run the project to test your implementation. In order to properly observe and track movements of objects, initially motion is switched off. You can type the V key to enable motion when you are ready. When motion is toggled on, you can observe a natural-looking free-falling movement for all objects. You can type G to create more objects and observe similar free-fall movements of the created objects.

Notice that when the objects fall below the lower platform they are re-generated in the central region of the scene with a random initial upward velocity. Observe objects move upwards until the y-component velocity reaches zero, and then they begin to fall downwards as a result of gravitational acceleration. Typing the H key injects new random upward velocities to all objects resulting in objects decelerating while moving upwards.

Try typing the C key to observe the computed collision information when objects overlap, or, interpenetrate. Pay attention and note that as objects travel through the scene interpenetration occurs frequently. You are now ready to examine and implement how to resolve object interpenetration in the next section.