Appendix

Completing the Lunar Lander Application

As you've gone through the book, I've slowly been contributing toward your Lunar Lander game where it has made sense. I haven't completed it, though, and this is deliberate. Where would be the fun in providing you with the finished game?! The knowledge you've gained through the book in iOS, its SDK, and Xcode will allow you to complete the game and publish it—or you may want to start a completely different game.

However, it wouldn't be fair to leave you hanging, so in this appendix I discuss some of the elements you may want to consider adding to the game. The source code associated with the book includes code fragments that complement this chapter to help you complete your game.

Implementing the Game Physics

To ensure that your lander observes the laws of physics, you need to implement a simple physics engine for it. You can find a reference describing the physics required at `www.physicsclassroom.com/class/newtlaws/u2l4a.cfm`. Don't worry; I take you through the basics next.

Gravity

This simulation is quite simple. First the forces of gravity are applied:

```
Y = Y + Gravity * TotaleSecondsOfThrust;
```

Here the `Y` value that controls the lander's height is adjusted by multiplying the `Gravity` figure (a double value of 9.8) by the number of seconds that thrust is applied, calculated by detecting the timestamp when the thrust key is held down, and the seconds elapsed from when it is released. You can use the `NSTimeInterval()` method to achieve this.

Thrust

You then need to apply the force of the thrust according to thrust and rotation. You need to use the `Math.h` include for this, but you have to implement your own angle-normalization macro, because the equivalent of `NormalizeAngle()` in .NET doesn't exist in Objective-C. No matter, it's easily done. First define this macro:

```
#define NormalizeAngle(x) ((return x % 360)+(x < 0 ? 360 : 0))
```

Then adjust your x and x axis according to the force of thrust:

```
Y = Y - Math.sin(NormalizeAngle(rotation + M_PI / 2)) * totalseconds * thrustspeed;
X = X - Math.cos(NormalizeAngle(rotation + M_PI / 2)) * totalseconds * thrustspeed;
```

The variable rotation obviously represents the angle of rotation for the lander, as affected by the user pressing the left and right keys and the `totalseconds` values (the number of seconds the thrust key is held down). The `thrustspeed` value is a constant value of 20 indicating how powerful the thrust is.

Finally, don't forget to reduce your fuel figure as the engine is thrusting.

Rotation

You won't cover rotating both left and right because they mirror each other. Let's look at what you need to do to rotate left. This method does of course assume the existence of a fuel tank, initialized with an initial value (here the value 1000) and that you pass the total number of seconds the rotation key was held down. Your method looks as follows:

```
// rotateLeft - rotate the lander left
- (void)rotateLeft:(float) totalseconds
{
    // You can't thrust if you've no fuel
    if (fuel <0)
        return;

    // Rotate left
    rotationMomentum -= M_PI / 2.0f * totalseconds;

    // Burn fuel
    fuel -= 0.5 // fuel burn evey time you thrust
}
```

The obvious difference for rotating right is that you increase the `rototationMomentum` value rather than decrease it.

Enabling User Interaction

The user is obviously required to interact with the game, although how you do this is up to you. Some options are to detect keypresses, or use onscreen images to detect a tap, or even use the accelerometer to detect the device being tipped left or right. In any case, you need to interpret the actions, and the following keys are used to interact with

the lander. Doing nothing simply lets the lander drop to the ground as gravity is applied. For your lander, you do the following.

- If the up arrow is pressed, onscreen or on the keyboard, the thruster is being fired.
- If the left arrow is pressed, onscreen or on the keyboard, the lander is rotating left.
- If the right arrow is pressed, onscreen or on the keyboard, the lander is rotating right.

Catching Game Events

You need to monitor some additional game events: for example, detecting when you hit the ground and the speed at which you hit. This is fairly straightforward using a combination of lunar lander y position and ground height and using the speed of the lander as an indicator of whether you're moving too fast.

The method only works when the ground is flat. If you decide to implement a Moon surface that has different gradients, you need to detect this in your game—maybe looking at the pixel values surrounding the lander.

Handling the Graphics

Your Lunar Lander implementation has already made provision to draw the graphics for the game by implementing the `drawRect` method, although the default implementation simply draws the same graphic (the nonthrusting lander) at the x and y location.

You need to adjust this code to be representative of the other states, including when the game is running, looking for when it should be thrusting or just dropping, and drawing at the angle appropriate for the lander. The angle is obviously in relation to the rotate-left or -right keypresses.

To rotate your image, you could use the `rotateAt()` method in Microsoft .NET, but such a method doesn't exist in Objective-C. Instead, you can use a simple macro to convert between degrees and radians, the measure used by the `UImage rotate` equivalent called `transform`. First define a macro something like the following:

```
#define degreesToRadians(x) (M_PI * (x) / 180.0)
```

Then add the following line, assuming the `deg` variable represents the degrees you wish to rotate by:

```
myView.transform = CGAffineTransformMakeRotation(degreesToRadians(deg) );
```

If you're interested in knowing more, the post at `www.platinumball.net/blog/2010/01/31/iphone-uiimage-rotation-and-scaling/` has a lot of detail about how to do image rotation and scaling.

Displaying a High Score

This should be straightforward because you covered how to create high-score code in Chapter 8, and you also looked at different mechanisms for displaying a dynamic user interface in Chapter 6. Combining these with the high-score mechanics of your game allows you to create a high-score chart that can be displayed when the game is at the main menu.

Resources

In addition to the information provided in this appendix to help you complete the Lunar Lander game, and the additional code supplied with the book, the following resources may help you complete and/or customize your game:

- *PhET Lunar Lander:* A number of resources including an online version of the game, provided as part of a teaching aid. `http://phet.colorado.edu/en/simulation/lunar-lander`.
- *LunarView.java:* A Java-based implementation of the Lunar Lander game for the Android mobile device. `http://developer.android.com/resources/samples/LunarLander/src/com/example/android/lunarlander/LunarView.html`.
- *Code Project:* A .NET-based implementation of the Lunar Lander game, written in C#. `http://www.codeproject.com/KB/game/lunarlander.aspx`.
- *History of Lunar Lander:* Wikipedia's history of the Lunar Lander game. `http://en.wikipedia.org/wiki/Lunar_Lander_(video_game)`.

Index

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X, Y, Z