CHAPTER 5

Ladybug Chase

Games are among the most exciting mobile phone apps, both to play and to create. The recent smash hit Angry Birds was downloaded 50 million times in its first year and is played more than a million hours every day, according to Rovio, its developer. (There is even talk of making it into a feature film!) While we can't guarantee that kind of success, we can help you create your own games with App Inventor, including this one involving a ladybug eating aphids while avoiding a frog.

What You'll Build

With the Ladybug Chase app shown in Figure 5-1, the user can:

- Control a ladybug by tilting the phone.
- View an energy-level bar on the screen, which decreases over time, leading to the ladybug's starvation.
- Make the ladybug chase and eat aphids to gain energy and prevent starvation.
- Help the ladybug avoid a frog that wants to eat it.

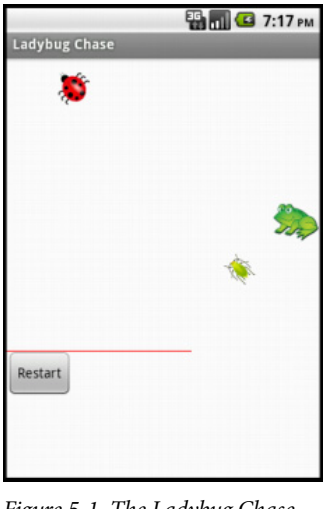

Figure 5-1. The Ladybug Chase game in the Designer

What You'll Learn

You should work through the MoleMash app in Chapter 3 before delving into this chapter, as it assumes you know about procedure creation, random-number generation, the **ifelse** block, and the `ImageSprite`, `Canvas`, `Sound`, and `Clock` components.

In addition to reviewing material from MoleMash and other previous chapters, this chapter introduces:

- Using multiple `ImageSprite` components and detecting collisions between them.
- Detecting phone tilts with an `OrientationSensor` component and using it to control an `ImageSprite`.
- Changing the picture displayed for an `ImageSprite`.
- Drawing lines on a `Canvas` component.
- Controlling multiple events with a `Clock` component.
- Using variables to keep track of numbers (the ladybug's energy level).
- Creating and using procedures with parameters.
- Using the **and** block.

Designing the Components

This application will have a `Canvas` that provides a playing field for three `ImageSprite` components: one for the ladybug, one for the aphid, and one for the frog, which will also require a `Sound` component for its "ribbit." The `OrientationSensor` will be used to measure the phone's tilt to move the ladybug, and a `Clock` will be used to change the aphid's direction. There will be a second `Canvas` that displays the ladybug's energy level. A Reset button will restart the game if the ladybug starves or is eaten. Table 5-1 provides a complete list of the components in this app.

Table 5-1. All of the components for the Ladybug Chase game

Component type	Palette group	What you'll name it	Purpose
`Canvas`	Basic	`FieldCanvas`	Playing field.
`ImageSprite`	Animation	`Ladybug`	User-controlled player.
`OrientationSensor`	Sensors	`OrientationSensor1`	Detect the phone's tilt to control the ladybug.
`Clock`	Basic	`Clock1`	Determines when to change the `ImageSprites'` headings
`ImageSprite`	Animation	`Aphid`	The ladybug's prey.
`ImageSprite`	Animation	`Frog`	The ladybug's predator.
`Canvas`	Basic	`EnergyCanvas`	Display the ladybug's energy level.
`Button`	Basic	`RestartButton`	Restart the game.
`Sound`	Media	`Sound1`	"Ribbit" when the frog eats the ladybug.

Getting Started

Download the images of the ladybug, aphid, and frog from the book's website (*http://examples.oreilly.com/0636920016632/*). You'll also need to download the sound file for the frog's ribbit.

Connect to the App Inventor website and start a new project. Name it "LadybugChase" and also set the screen's title to "Ladybug Chase". Open the Blocks Editor and connect to the phone. Add the images you found or created, as well as the sound file, to the Media panel.

If you will be using a phone, you'll need to disable autorotation of the screen, which changes the display direction when you turn the phone. On most phones, you do this by going to the home screen, pressing the menu button, selecting Settings, selecting Display, and unchecking the box labeled "Auto-rotate screen."

Animating the Ladybug

In this "first-person chewer" game, the user will be represented by a ladybug, whose movement will be controlled by the phone's tilt. This brings the user into the game in a different way from MoleMash, in which the user was outside the phone, reaching in.

Adding the Components

While previous chapters have had you create all the components at once, that's not how developers typically work. Instead, it's more common to create one part of a program at a time, test it, and then move on to the next part of the program. In this section, we will create the ladybug and control its movement.

- Create a `Canvas` in the Component Designer, name it `FieldCanvas`, and set its `Width` to "Fill parent" and its `Height` to 300 pixels.
- Place an `ImageSprite` on the `Canvas`, renaming it `Ladybug` and setting its `Picture` property to the (live) ladybug image. Don't worry about the values of the `X` and `Y` properties, as those will depend on where on the canvas you placed the `ImageSprite`.

As you may have noticed, `ImageSprites` also have `Interval`, `Heading`, and `Speed` properties, which we will use in this program:

- The `Interval` property, which you can set to 10 (milliseconds) for this game, specifies how often the `ImageSprite` should move itself (as opposed to being moved by the `MoveTo` procedure, which you used for MoleMash).

- The `Heading` property indicates the direction in which the `ImageSprite` should move, in degrees. For example, 0 means due right, 90 means straight up, 180 means due left, and so on. Leave the `Heading` as-is right now; we will change it in the Blocks Editor.
- The `Speed` property specifies how many pixels the `ImageSprite` should move whenever its `Interval` (10 milliseconds) passes. We will also set the `Speed` property in the Blocks Editor.

For more details on image sprites, see Chapter 17.

The ladybug's movement will be controlled by an `OrientationSensor`, which detects how the phone is tilted. We want use the `Clock` component to check the phone's orientation every 10 milliseconds (100 times per second) and change the ladybug's `Heading` (direction) accordingly. We will set this up in the Blocks Editor as follows:

1. Add an `OrientationSensor`, which will appear in the "Non-visible components" section.
2. Add a `Clock`, which will also appear in the "Non-visible components" section, and set its `TimerInterval` to 10 milliseconds. Check what you've added against Figure 5-2.

Adding the Behavior

Moving to the Blocks Editor, create the procedure `UpdateLadybug` and a **Clock1 .Timer** block, as shown in Figure 5-3. Try typing the names of some of the blocks (such as "Clock1.Timer") instead of dragging them out of the drawers. (Note that the operation applied to the number 100 is multiplication, indicated by an asterisk, which may be hard to see in the figure.) You do not need to create the yellow comment callouts, although you can by right-clicking a block and selecting Add Comment.

The `UpdateLadybug` procedure makes use of two of the `OrientationSensor`'s most useful properties:

- `Angle`, which indicates the direction in which the phone is tilted (in degrees).
- `Magnitude`, which indicates the amount of tilt, ranging from 0 (no tilt) to 1 (maximum tilt).

Multiplying the `Magnitude` by 100 tells the ladybug that it should move between 0 and 100 pixels in the specified `Heading` (direction) whenever its `TimerInterval`, which you previously set to 10 milliseconds in the Component Designer, passes.

Although you can try this out on the connected phone, the ladybug's movement might be both slower and jerkier than if you package and download the app to the phone. If, after doing that, you find the ladybug's movement too sluggish, increase the speed multiplier. If the ladybug seems too jerky, decrease it.

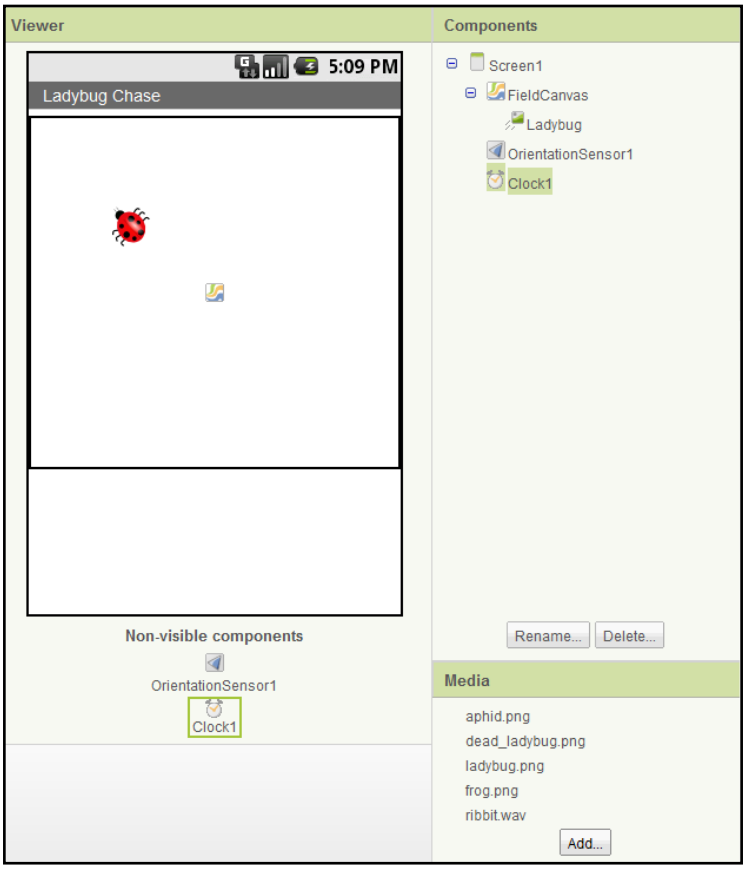

Figure 5-2. Setting up the user interface in the Component Designer for animating the ladybug

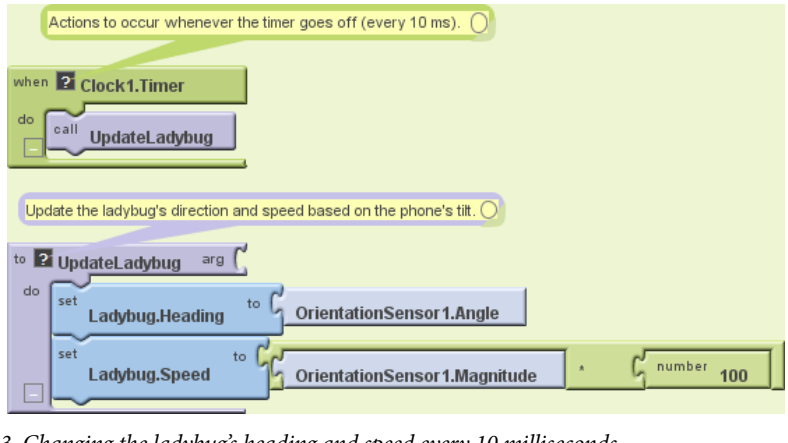

Figure 5-3. Changing the ladybug's heading and speed every 10 milliseconds

Displaying the Energy Level

We will display the ladybug's energy level with a red bar in a second canvas. The line will be 1 pixel high, and its width will be the same number of pixels as the ladybug's energy, which ranges from 200 (well fed) to 0 (dead).

Adding a Component

In the Designer, create a new `Canvas`, placing it beneath `FieldCanvas` and naming it `EnergyCanvas`. Set its `Width` property to "Fill parent" and its `Height` to 1 pixel.

Creating a Variable: Energy

In the Blocks Editor, you will need to create a variable `energy` with an initial value of 200 to keep track of the ladybug's energy level. (As you may recall, we first used a variable, `dotSize`, in Chapter 2's PaintPot app.) Here's how to do it:

1. In the Blocks Editor, in the Built-In column, open the Definitions drawer. Drag out a **def variable** block. Change the text "variable" to "energy".
2. If there is a block in the socket on the right side of **def energy**, delete it by selecting it and either pressing the Delete key or dragging it to the trash can.
3. Create a **number 200** block (by either starting to type the number 200 or dragging a **number** block out of the Math drawer) and plug it into **def energy**, as shown in Figure 5-4.

Figure 5-4. Initializing the variable energy to 200

Figure 5-5 shows how creating the variable also added blocks to the My Definitions drawer to set or get the value of **energy**.

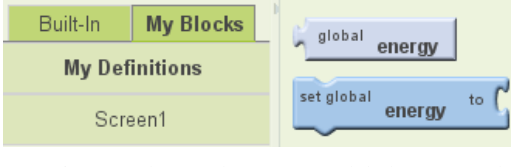

Figure 5-5. View of the My Definitions drawer showing new global energy and set global energy blocks

Drawing the Energy Bar

We want to communicate the energy level with a red bar whose length in pixels is the energy value. To do so, we could create two similar sets of blocks as follows:

1. Draw a red line from (0, 0) to (`energy`, 0) in `FieldCanvas` to show the current energy level.
2. Draw a white line from (0, 0) to (`EnergyCanvas.Width`, 0) in `FieldCanvas` to erase the current energy level before drawing the new level.

However, a better alternative is to create a procedure that can draw a line of any length and of any color in `FieldCanvas`. To do this, we must specify two arguments, `length` and `color`, when our procedure is called, just as we needed to specify parameter values in MoleMash when we called the built-in `random integer` procedure. Here are the steps for creating a `DrawEnergyLine` procedure, which is shown in Figure 5-6.

1. Go to the Definition drawer and drag out a **to procedure** block.
2. Click its name (probably "procedure1") and change it to "DrawEnergyLine".
3. Go back to the Definition drawer and drag out a **name** block, snapping it into the *arg* (short for *argument*) socket. Click its name and change it to "color".
4. Repeat step 3 to add a second argument and name it "length".
5. Fill in the rest of the procedure as shown in Figure 5-6. You can find the new **color** and **length** blocks in the My Definitions drawer.

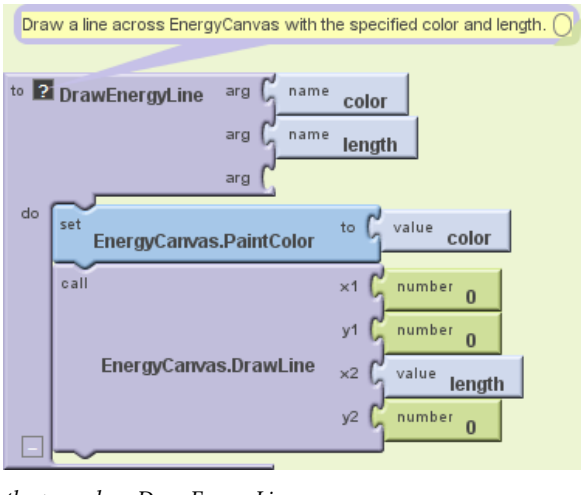

Figure 5-6. Defining the procedure DrawEnergyLine

Now that you're getting the hang of creating your own procedures, let's also write a `DisplayEnergyLevel` procedure that calls `DrawEnergyLine` twice, once to erase the old line (by drawing a white line all the way across the canvas) and once to display the new line, as shown in Figure 5-7.

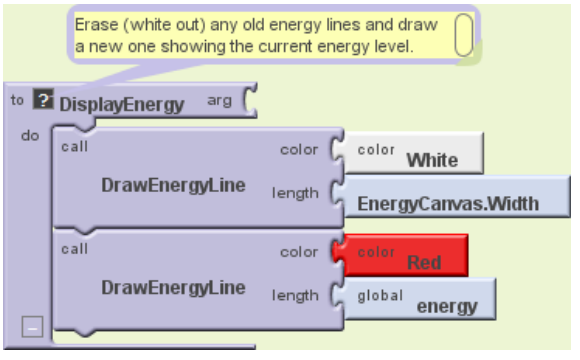

Figure 5-7. Defining the procedure DisplayEnergyLevel

The `DisplayEnergyLevel` procedure consists of four lines that do the following:

1. Set the paint color to white.
2. Draw a line all the way across `EnergyCanvas` (which is only 1 pixel high).
3. Set the paint color to red.
4. Draw a line whose length in pixels is the same as the energy value.

Note. *The process of replacing common code with calls to a new procedure is called refactoring, a set of powerful techniques for making programs more maintainable and reliable. In this case, if we ever wanted to change the height or location of the energy line, we would just have to make a single change to* `DrawEnergyLine`*, rather than making changes to every call to it. For more information on procedures, see Chapter 21.*

Starvation

Unlike the apps in previous chapters, this game has a way to end: it's over if the ladybug fails to eat enough aphids or is eaten by the frog. In either of these cases, we want the ladybug to stop moving (which we can do by setting `Ladybug.Enabled` to false) and for the picture to change from a live ladybug to a dead one (which we can do by changing `Ladybug.Picture` to the name of the appropriate uploaded image). Create the `GameOver` procedure as shown in Figure 5-8.

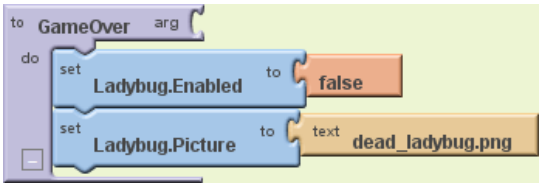

Figure 5-8. Defining the procedure GameOver

Next, add the code outlined in red in Figure 5-9 to `UpdateLadybug` (which, as you may recall, is called by `Clock.Timer` every 10 milliseconds) to:

- Decrement its energy level.
- Display the new level.
- End the game if energy is 0.

Test your app. *You should be able to test this code on your phone and verify that the energy level decreases over time, eventually causing the ladybug's demise. If you want to restart the application, press the "Connect to Device..." button in the Blocks Editor.*

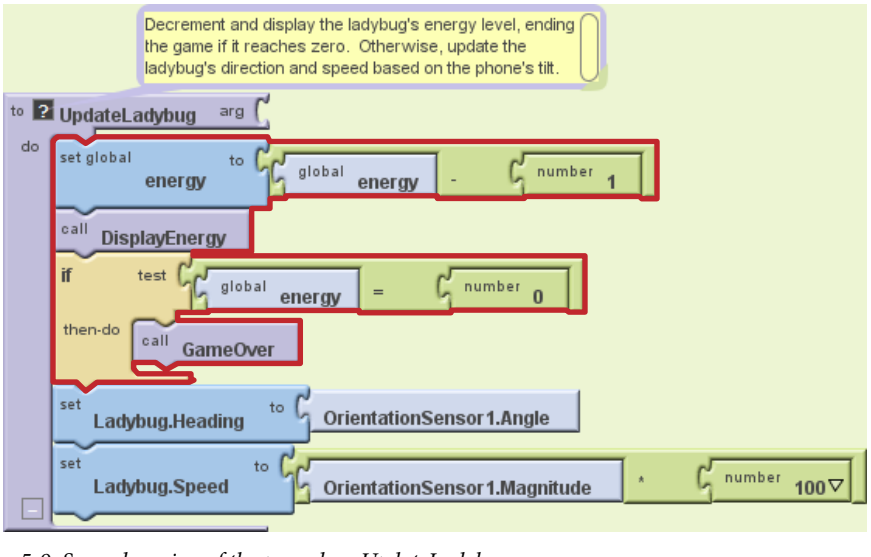

Figure 5-9. Second version of the procedure UpdateLadybug

Adding an Aphid

The next step is to add an aphid. Specifically, an aphid should flit around `FieldCanvas`. If the ladybug runs into the aphid (thereby "eating" it), the ladybug's energy level should increase and the aphid should disappear, to be replaced by another one a little later. (From the user's point of view, it will be a different aphid, but it will really be the same `ImageSprite` component.)

Adding an ImageSprite

The first step to add an aphid is to go back to the Designer and create another `ImageSprite`, being sure not to place it on top of the ladybug. It should be renamed `Aphid` and its properties set as follows:

1. Set its `Picture` property to the aphid image file you uploaded.
2. Set its `Interval` property to 10, so, like the ladybug, it moves every 10 milliseconds.
3. Set its `Speed` to 2, so it doesn't move too fast for the ladybug to catch it.

Don't worry about its `X` and `Y` properties (as long as it's not on top of the ladybug) or its `Heading` property, which will be set in the Blocks Editor.

Controlling the Aphid

By experimenting, we found it worked best for the aphid to change directions approximately once every 50 milliseconds (5 "ticks" of `Clock1`). One approach to enabling this behavior would be to create a second clock with a `TimerInterval` of 50 milliseconds. However, we'd like you to try a different technique so you can learn about the **random fraction** block, which returns a random number greater than or equal to 0 and less than 1 each time it is called. Create the `UpdateAphid` procedure shown in Figure 5-10 and add a call to it in **Clock1.Timer**.

How the blocks work

Whenever the timer goes off (100 times per second), both **UpdateLadybug** (like before) and **UpdateAphid** are called. The first thing that happens in **UpdateAphid** is that a random fraction between 0 and 1 is generated—for example, 0.15. If this number is less than 0.20 (which will happen 20% of the time), the aphid will change its direction to a random number of degrees between 0 and 360. If the number is *not* less than 0.20 (which will be the case the remaining 80% of the time), the aphid will stay the course.

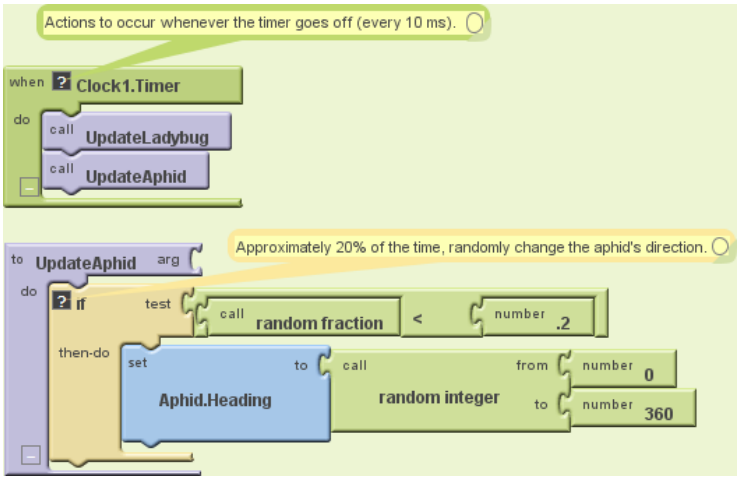

Figure 5-10. Adding the procedure UpdateAphid

Having the Ladybug Eat the Aphid

The next step is having the ladybug "eat" the aphid when they collide. Fortunately, App Inventor provides blocks for detecting collisions between `ImageSprite` components, which raises the question: what should happen when the ladybug and the aphid collide? You might want to stop and think about this before reading on.

To handle what happens when the ladybug and aphid collide, let's create a procedure, `EatAphid`, that does the following:

- Increases the energy level by 50 to simulate eating the tasty treat.
- Causes the aphid to disappear (by setting its `Visible` property to false).
- Causes the aphid to stop moving (by setting its `Enabled` property to false).
- Causes the aphid to move to a random location on the screen. (This follows the same pattern as the code to move the mole in MoleMash).

Check that your blocks match Figure 5-11. If you had other ideas of what should happen, such as sound effects, you can add those too.

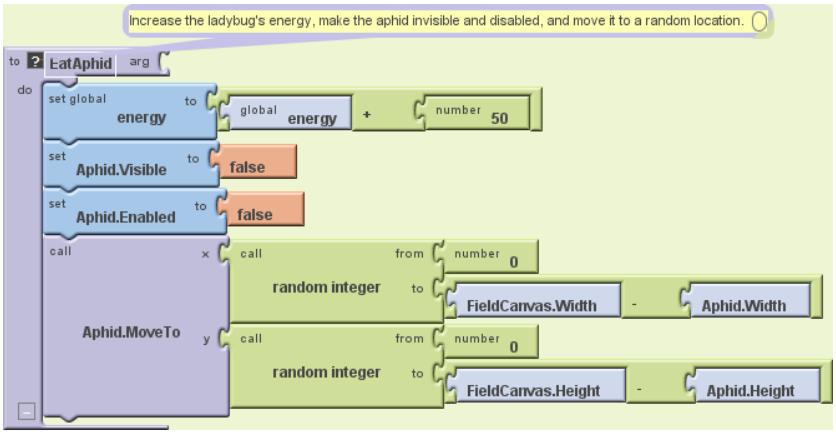

Figure 5-11. Adding the procedure EatAphid

How the blocks work

Whenever **EatAphid** is called, it adds 50 to the variable `energy`, staving off starvation for the ladybug. Next, the aphid's `Visible` and `Enabled` properties are set to false so it seems to disappear and stops moving. Finally, random *x* and *y* coordinates are generated for a call to **Aphid.MoveTo** so that, when the aphid reappears, it's in a new location (otherwise, it will be eaten as soon as it reemerges).

Detecting a Ladybug–Aphid Collision

Figure 5-12 shows the code to detect collisions between the ladybug and the aphid. Note that when you add a condition to the "and" block, a new test socket appears.

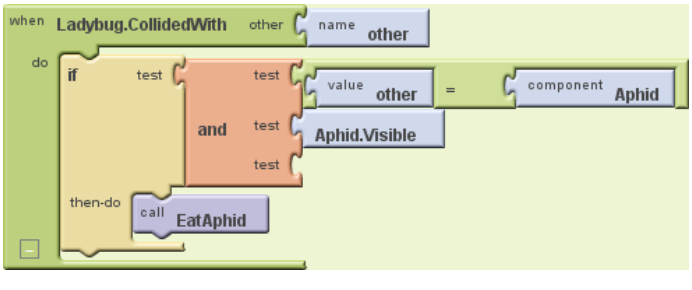

Figure 5-12. Detecting and acting on collisions between the ladybug and aphid

How the blocks work

When the ladybug collides with another `ImageSprite`, **Ladybug.CollidedWith** gets called, with the parameter "other" bound to whatever the ladybug collided with. Right now, the only thing it can collide with is the aphid, but we'll be adding a frog

later. We'll use *defensive programming* and explicitly check that the collision was with the aphid before calling **EatAphid**. There's also a check to confirm that the aphid is visible. Otherwise, after an aphid is eaten but before it reappears, it could collide with the ladybug again. Without the check, the invisible aphid would be eaten again, causing another jump in energy without the user understanding why.

Note. *Defensive programming is the practice of writing code in such a way that it is still likely to work even if the program gets modified. In Figure 5-12, the test* ***other = Aphid*** *is not strictly necessary because the only thing the ladybug can currently collide with is the aphid, but having the check will prevent our program from malfunctioning if we add another* `ImageSprite` *and forget to change* ***Ladybug.CollidedWith****. Programmers generally spend more time fixing bugs than writing new code, so it is well worth taking a little time to write code in a way that prevents bugs.*

The Return of the Aphid

To make the aphid eventually reappear, you should modify `UpdateAphid` as shown in Figure 5-13 so it changes the aphid's direction only if it is visible. (Changing it if it's invisible is a waste of time.) If the aphid is not visible (as in, it has been eaten recently), there is a 1 in 20 (5%) chance that it will be reenabled—in other words, made eligible to be eaten again.

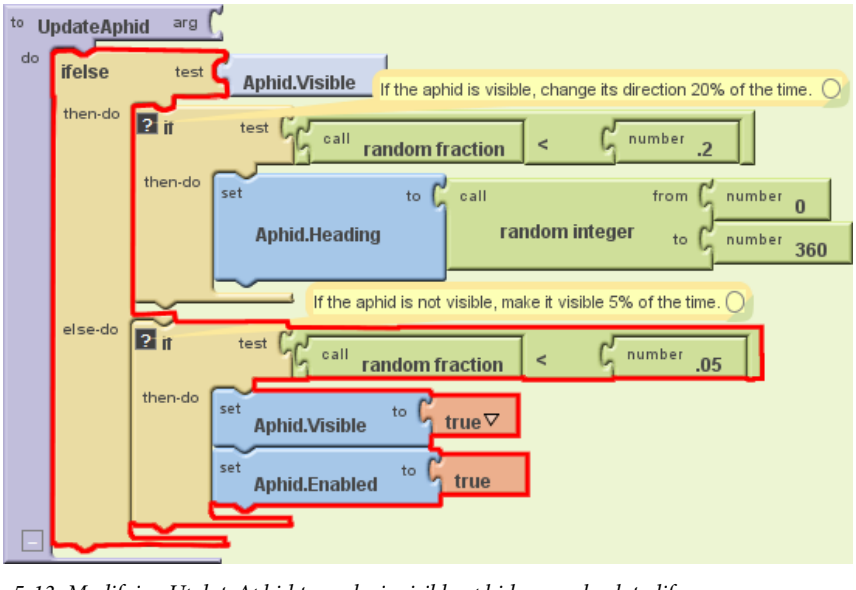

Figure 5-13. Modifying UpdateAphid to make invisible aphids come back to life

How the blocks work

`UpdateAphid` is getting pretty complex, so let's carefully step through its behavior:

- If the aphid is visible (which will be the case unless it was just eaten), `UpdateAphid` behaves as we first wrote it. Specifically, there is a 20% chance of its changing direction.
- If the aphid is not visible (was recently eaten), then the "else-do" part of the **ifelse** block will run. A random number is then generated. If it is less than .05 (which it will be 5% of the time), the aphid becomes visible again and is enabled, making it eligible to be eaten again.

Because `UpdateAphid` is called by `Clock1.Timer`, which occurs every 10 milliseconds, and there is a 1 in 20 (5%) chance of the aphid becoming visible again, the aphid will take on average 200 milliseconds (1/5 of a second) to reappear.

Adding a Restart Button

As you may have noticed from testing the app with your new aphid-eating functionality, the game really needs a Restart button. (This is another reason why it's helpful to design and build your app in small chunks and then test it—you often discover things that you may have overlooked, and it's easier to add them as you progress than to go back in and change them once the app is "complete.") In the Component Designer, add a `Button` component underneath `EnergyCanvas`, rename it "RestartButton", and set its `Text` property to "Restart".

In the Blocks Editor, create the code shown in Figure 5-14 to do the following when the `RestartButton` is clicked:

1. Set the energy level back to 200.
2. Reenable the aphid and make it visible.
3. Reenable the ladybug and change its picture back to the live ladybug (unless you want zombie ladybugs!).

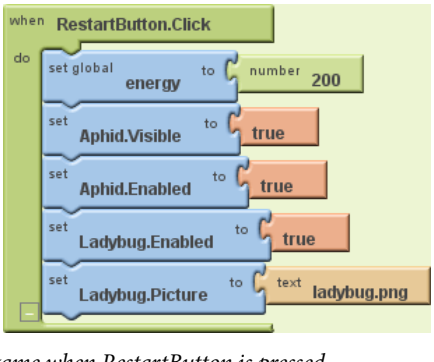

Figure 5-14. Restarting the game when RestartButton is pressed

Adding the Frog

Right now, keeping the ladybug alive isn't too hard. We need a predator. Specifically, we'll add a frog that moves directly toward the ladybug. If they collide, the ladybug gets eaten, and the game ends.

Having the Frog Chase the Ladybug

The first step to having the frog chase the ladybug is returning to the Component Designer and adding a third `ImageSprite`—`Frog`—to `FieldCanvas`. Set its `Picture` property to the appropriate picture, its `Interval` to 10, and its `Speed` to 1, since it should be slower-moving than the other creatures.

Figure 5-15 shows `UpdateFrog`, a new procedure you should create and call from `Clock1.Timer`.

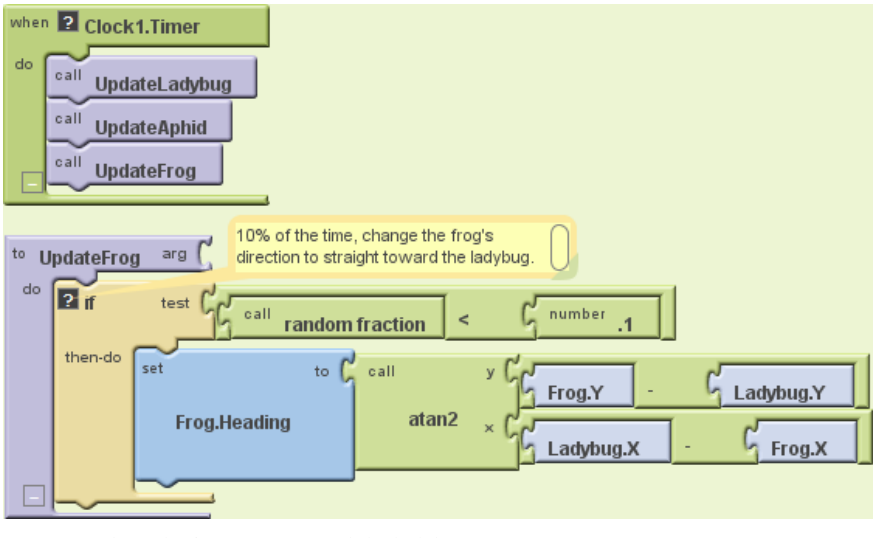

Figure 5-15. Making the frog move toward the ladybug

How the blocks work

By now, you should be familiar with the use of the **random fraction** block to make an event occur with a certain probability. In this case, there is a 10% chance that the frog's direction will be changed to head straight toward the ladybug. This requires trigonometry, but don't panic—you don't have to figure it out yourself! App Inventor handles a ton of math functions for you, even stuff like trig. In this case, you want to use the **atan2** (arctangent) block, which returns the angle corresponding to a given set of *x* and *y* values.

(For those of you familiar with trigonometry, the reason the y argument to **atan2** has the opposite sign of what you'd expect—the opposite order of arguments to subtract—is that the *y* coordinate increases in the downward direction on an Android Canvas, the opposite of what would occur in a standard *x*–*y* coordinate system.)

Having the Frog Eat the Ladybug

We now need to modify the collision code so that if the ladybug collides with the frog, the energy level and bar goes to 0 and the game ends, as shown in Figure 5-16.

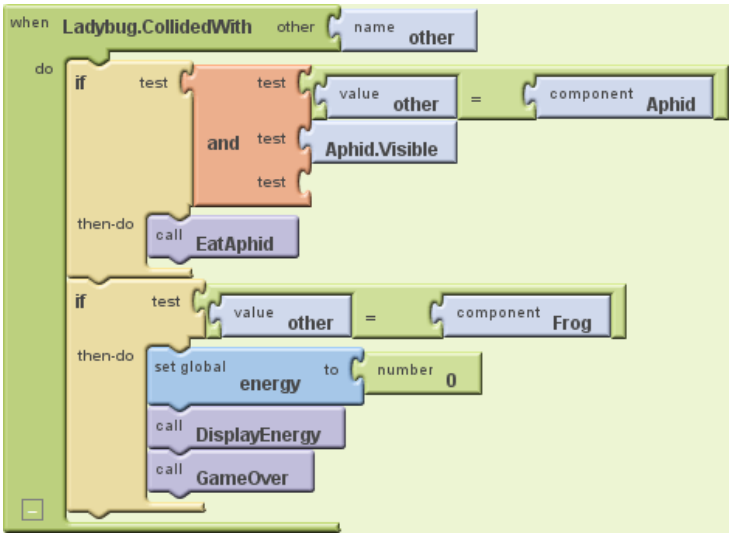

Figure 5-16. Making the frog eat the ladybug

How the blocks work

In addition to the first **if**, which checks if the ladybug collided with the aphid, there is now a second **if**, which checks if the ladybug has collided with the frog. If the ladybug and the frog collide, three things happen:

1. The variable `energy` goes down to 0, since the ladybug has lost its life force.
2. **DisplayEnergy** is called, to erase the previous energy line (and draw the new—empty—one).
3. The procedure we wrote earlier, **GameOver**, is called to stop the ladybug from moving and changes its picture to that of a dead ladybug.

The Return of the Ladybug

RestartButton.Click already has code to replace the picture of the dead ladybug with the one of the live ladybug. Now you need to add code to move the live ladybug to a random location. (Think about what would happen if you didn't move the ladybug at the beginning of a new game. Where would it be in relation to the frog?) Figure 5-17 shows the blocks to move the ladybug when the game restarts.

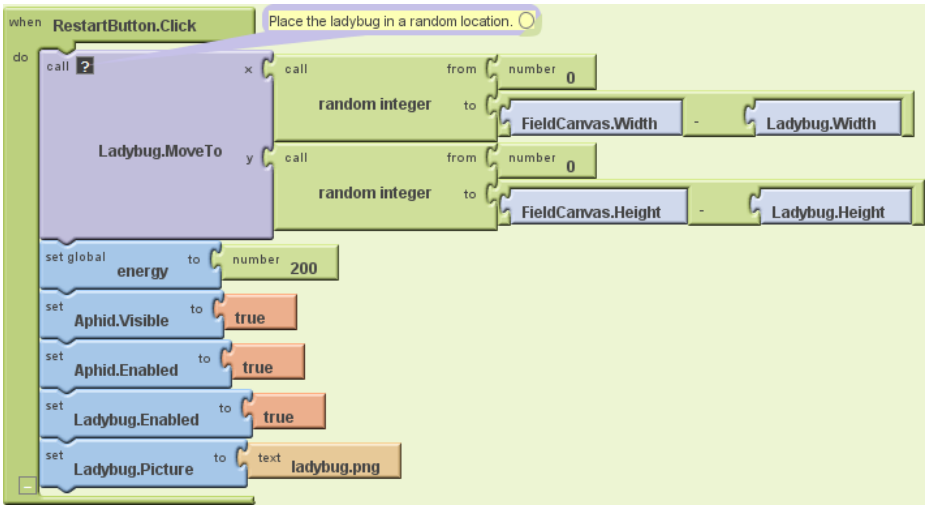

Figure 5-17. The final version of RestartButton.Click

How the blocks work

The only difference between this version of **RestartButton.Click** and the previous version is the **Ladybug.MoveTo** block and its arguments. The built-in function **random integer** is called twice, once to generate a legal *x* coordinate and once to generate a legal *y* coordinate. While there is nothing to prevent the ladybug from being placed on top of the aphid or the frog, the odds are against it.

Test your app. *Restart the game and make sure the ladybug shows up in a new random location.*

Adding Sound Effects

When you tested the game, you may have noticed there isn't very good feedback when an animal gets eaten. To add sound effects and tactile feedback, do the following:

1. In the Component Designer, add a `Sound` component. Set its `Source` to the sound file you uploaded.
2. Go to the Blocks Editor, where you will:
    a. Make the phone vibrate when an aphid is eaten by adding a **Sound1 .Vibrate** block with an argument of 100 (milliseconds) in **EatAphid**.
    b. Make the frog ribbit when it eats the ladybug by adding a call to **Sound1.Play** in **Ladybug.CollidedWith** just before the call to **GameOver**.

Variations

Here are some ideas of how to improve or customize this game:

- Currently, the frog and aphid keep moving after the game has ended. Prevent this by setting their `Enabled` properties to false in **GameOver** and back to true in **RestartButton.Click**.
- Display a score indicating how long the ladybug has remained alive. You can do this by creating a label that you increment in **Clock1.Timer**.
- Make the energy bar more visible by increasing the `Height` of `EnergyCanvas` to 2 and drawing two lines, one above the other, in `DrawEnergyLine`. (This is another benefit of having a procedure rather than duplicated code to erase and redraw the energy line: you just need to make a change in one place to change the size—or color, or location—of the line.)
- Add ambiance with a background image and more sound effects, such as nature sounds or a warning when the ladybug's energy level gets low.
- Have the game get harder over time, such as by increasing the frog's `Speed` property or decreasing its `Interval` property.
- Technically, the ladybug should disappear when it is eaten by the frog. Change the game so that the ladybug becomes invisible if eaten by the frog but not if it starves to death.
- Replace the ladybug, aphid, and frog pictures with ones more to your taste, such as a hobbit, orc, and evil wizard or a rebel starfighter, energy pod, and Imperial starfighter.

Summary

With two games now under your belt (if you completed the MoleMash tutorial), you now know how to create your own games, which is the goal of many new programmers or wannabes! Specifically, you learned:

- You can have multiple `ImageSprite` components (the ladybug, the aphid, and the frog) and can detect collisions between them.
- The tilt of the phone can be detected by the `OrientationSensor`, and the value can be used to control the movement of a sprite (or anything else you can imagine).
- A single `Clock` component can control multiple events that occur at the same frequency (changes in the ladybug's and frog's directions), or at different frequencies, by using the **random fraction** block. For example, if you want an event to occur approximately one-fourth (25 percent) of the time, put it in the body of an **if** block that is only executed when the result of **random fraction** is less than .25.
- You can have multiple `Canvas` components in a single app, which we did to have both a playing field and to display a variable graphically (instead of through a `Label`).
- User-defined procedures can be defined with parameters (such as "color" and "length" in `DrawEnergyLine`) that control the behavior, greatly expanding the power of procedural abstraction.

Another component useful for games is `Ball`, which only differs from `ImageSprite` in having the appearance of a filled circle rather than an arbitrary image.

CHAPTER 6

Paris Map Tour

In this chapter, you'll build an app that lets you create your own custom guide for a dream trip to Paris. And since a few of your friends can't join you, we'll create a companion app that lets them take a virtual tour of Paris as well. Creating a fully functioning map app might seem really complicated, but App Inventor lets you use the `ActivityStarter` *component to launch Google Maps for each virtual location. First, you'll build an app that launches maps for the Eiffel Tower, the Louvre, and Notre Dame Cathedral with a single click. Then you'll modify the app to create a virtual tour of satellite maps that are also available from Google Maps.*

What You'll Learn

This chapter introduces the following App Inventor components and concepts:

- The `Activity Starter` component for launching other Android apps from your app. You'll use this component here to launch Google Maps with various parameters.
- The `ListPicker` component for allowing the user to choose from a list of locations.