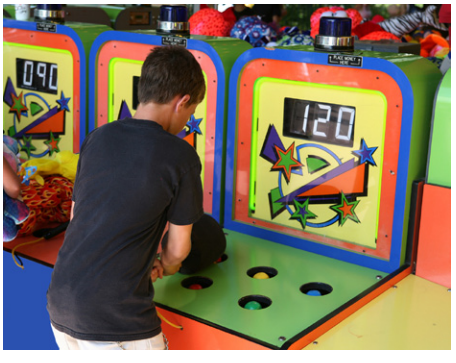
CHAPTER 3

MoleMash



This chapter shows you how to create MoleMash, a game inspired by the arcade classic Whac-A-Mole, in which mechanical critters pop briefly out of holes, and players score points by whacking them with a mallet. MoleMash was created by a member of the App Inventor team, nominally to test the sprite functionality (which she implemented), but really because she is a fan of the game.

When Ellen Spertus joined the App Inventor team at Google, she was eager to add support for creating games, so she volunteered to implement *sprites*. The term, originally reserved for mythological creatures such as fairies and pixies, emerged in the computing community in the 1970s, where it referred to images capable of movement on a computer screen (for video games). Ellen first worked with sprites when she attended a computer camp in the early 1980s and programmed a TI 99/4. Her work on sprites and MoleMash was motivated by double nostalgia—for both the computers and games of her childhood.

What You'll Build

For the MoleMash app shown in Figure 3-1, you'll implement the following functionality:

- A mole pops up at random locations on the screen, moving once every second.
- Touching the mole causes the phone to vibrate, the display of hits to be incremented (increased by one), and the mole to move immediately to a new location.
- Touching the screen but missing the mole causes the display of misses to be incremented.
- Pressing the Reset button resets the counts of hits and misses.

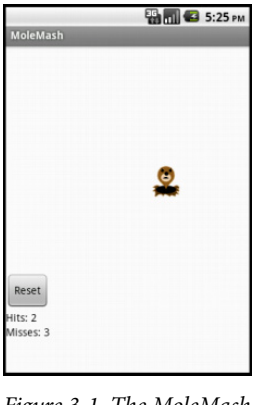


Figure 3-1. The MoleMash user interface

What You'll Learn

The tutorial covers the following components and concepts:

- The `ImageSprite` component for touch-sensitive movable images.
- The `Canvas` component, which acts as a surface on which to place the `ImageSprite`.
- The `Clock` component to move the sprite around.
- The `Sound` component to produce a vibration when the mole is touched.
- The `Button` component to start a new game.
- Procedures to implement repeated behavior, such as moving the mole.
- Generating random numbers.
- Using the addition (+) and subtraction (-) blocks.

Getting Started

Connect to the App Inventor website and start a new project. Name it "MoleMash" and also set the screen's title to "MoleMash". Open the Blocks Editor and connect to the phone.

Download this picture of a mole from this book's site (*http://examples.oreilly.com/0636920016632/*), naming it *mole.png* and noting where you save it on your computer. In the Media section of the Component Designer, click Add, browse to where the file is located on your computer, and upload it to App Inventor.

Designing the Components

You'll use these components to make MoleMash:

- A `Canvas` that serves as a playing field.
- An `ImageSprite` that displays a picture of a mole and can move around and sense when the mole is touched.
- A `Sound` that vibrates when the mole is touched.
- `Labels` that display "Hits: ", "Misses: ", and the actual numbers of hits and misses.
- `HorizontalArrangements` to correctly position the `Labels`.
- A `Button` to reset the numbers of hits and misses to 0.
- A `Clock` to make the mole move once per second.

Table 3-1 shows the complete list of components.

Table 3-1. The complete list of components for MoleMash

Component type	Palette group	What you'll name it	Purpose
Canvas	Basic	Canvas1	The container for ImageSprite.
ImageSprite	Animation	Mole	The user will try to touch this.
Button	Basic	ResetButton	The user will press this to reset the score.
Clock	Basic	Clock1	Control the mole's movement.
Sound	Media	Sound1	Vibrate when the mole is touched.
Label	Basic	HitsLabel	Display "Hits: ".
Label	Basic	HitsCountLabel	Display the number of hits.
HorizontalArrangement	Screen Arrangement	HorizontalArrangement1	Position HitsLabel next to HitsCountLabel.
Label	Basic	MissesLabel	Display "Misses: ".
Label	Basic	MissesCountLabel	Display the number of misses.
HorizontalArrangement	Screen Arrangement	HorizontalArrangement2	Position MissesLabel next to MissesCountLabel.

Placing the Action components

In this section, we will place the components necessary for the game's action. In the next section, we will place the components for displaying the score.

1. Drag in a Canvas component, leaving it with the default name Canvas1. Set its Width property to "Fill parent" so it is as wide as the screen, and set its Height to 300 pixels.
2. Drag in an ImageSprite component from the Animation group on the Palette. Place it anywhere on Canvas1. Click Rename at the bottom of the Components list and change its name to "Mole". Set its Picture property to *mole.png*, which you uploaded earlier.
3. Drag in a Button component from the Basic group on the Palette, placing it beneath Canvas1. Rename it to "ResetButton" and set its Text property to "Reset".
4. Drag in a Clock component. It will appear at the bottom of the Viewer in the "Non-visible components" section.
5. Drag in a Sound component from the Media group on the Palette. It, too, will appear in the "Non-visible components" section.

Your screen should now look something like Figure 3-2 (although your mole may be in a different position).

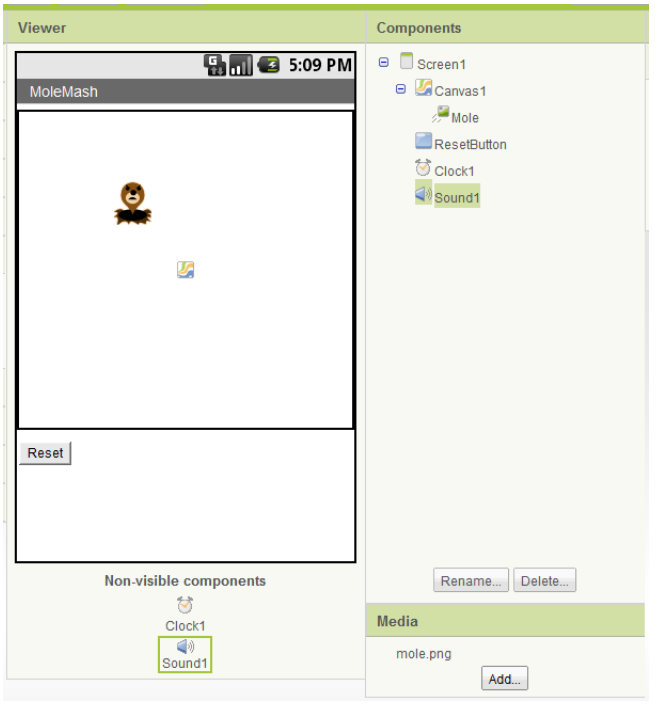


Figure 3-2. The Component Designer view of the "action" components

Placing the Label components

We will now place components for displaying the user's score—specifically, the number of hits and misses.

1. Drag in a `HorizontalArrangement` from the Screen Arrangement group, placing it beneath the `Button` and keeping the default name of `HorizontalArrangement1`.
2. Drag two `Labels` from the Basic group into `HorizontalArrangement1`.
   a. Rename the left `Label` to "HitsLabel" and set its `Text` property to "Hits: " (making sure to include a space after the colon).
   b. Rename the right `Label` to "HitsCountLabel" and set its `Text` property to "0".
3. Drag in a second `HorizontalArrangement`, placing it beneath `HorizontalArrangement1`.

4. Drag two Labels into `HorizontalArrangement2`.
    a. Rename the left `Label` to "MissesLabel" and set its `Text` property to "Misses: " (making sure to include a space after the colon).
    b. Rename the right `Label` to "MissesCountLabel" and set its `Text` property to "0".

Your screen should now look like something like Figure 3-3.

Figure 3-3. The Component Designer view of all the MoleMash components

Adding Behaviors to the Components

After creating the preceding components, we can move to the Blocks Editor to implement the program's behavior. Specifically, we want the mole to move to a random location on the canvas every second. The user's goal is to tap on the mole wherever it appears, and the app will display the number of times the user hits or misses the mole. (*Note*: We recommend using your finger, not a mallet!) Pressing the Reset button resets the number of hits and misses to 0.

Moving the Mole

In the programs you've written thus far, you've called built-in procedures, such as `Vibrate` in HelloPurr. Wouldn't it be nice if App Inventor had a procedure that moved an `ImageSprite` to a random location on the screen? The bad news: it doesn't. The good news: you can create your own procedures! Just like the built-in procedures, your procedure will show up in a drawer and can be used anywhere in the app.

Specifically, we will create a procedure to move the mole to a random location on the screen, which we will name `MoveMole`. We want to call `MoveMole` at the start of the game, when the user successfully touches the mole, and once per second.

Creating MoveMole

To understand how to move the mole, we need to look at how Android graphics work. The canvas (and the screen) can be thought of as a grid with *x* (horizontal) and *y* (vertical) coordinates, where the (*x*, *y*) coordinates of the upper-left corner are (0, 0). The *x* coordinate increases as you move to the right, and the *y* coordinate increases as you move down, as shown in Figure 3-4. The `X` and `Y` properties of an `ImageSprite` indicate where its upper-left corner should be, so the top-left mole has `X` and `Y` values of 0.

To determine the maximum available `X` and `Y` values so that `Mole` fits on the screen, we need to make use of the `Width` and `Height` properties of `Mole` and `Canvas1`. (The mole's `Width` and `Height` properties are the same as the size of the image you uploaded. When you created `Canvas1`, you set its `Height` to 300 pixels and its `Width` to "Fill parent," which copies the width of its "parent" element, the screen.) If the mole is 36 pixels wide and the canvas is 200 pixels wide, the *x* coordinate of the left side of the mole can be as low as 0 (all the way to the left) or as high as 164 (200 – 36, or `Canvas1.Width` – `Mole.Width`) without the mole extending off the right edge of the screen. Similarly, the *y* coordinate of the top of the mole can range from 0 to `Canvas1.Height` – `Mole.Height`.

Figure 3-5 shows the procedure you will create, annotated with descriptive comments (which you can optionally add to your procedure).

To randomly place the mole, we will want to select an *x* coordinate in the range from 0 to `Canvas1.Width` – `Mole.Width`. Similarly, we will want the *y* coordinate to be in the range from 0 to `Canvas1.Height` – `Mole.Height`. We can generate a random number through the built-in procedure `random integer`, found in the Math drawer. You will need to change the default "from" parameter from 1 to 0 and replace the "to" parameters, as shown in Figure 3-5.

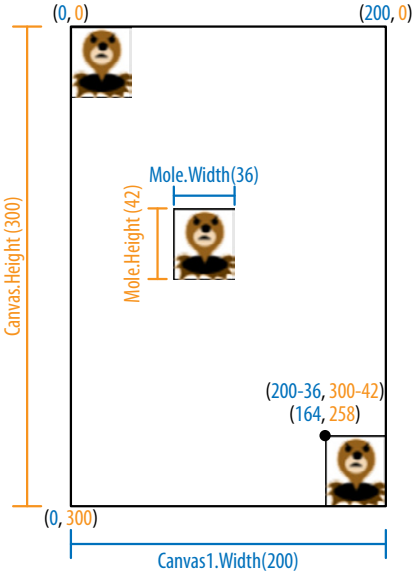


Figure 3-4. Positions of the mole on the screen, with coordinate, height, and width information; x coordinates and widths are shown in blue, while y coordinates and heights are shown in orange

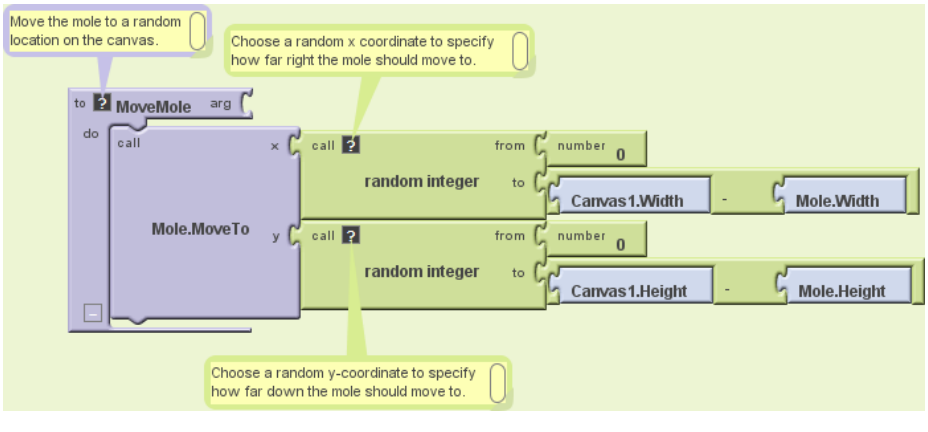


Figure 3-5. The MoveMole procedure, which places the mole in a random location

To create the procedure:

1. Click the Definition drawer under the Built-In tab in the Blocks Editor.
2. Drag out the **to procedure** block (not **to procedureWithResult**).
3. Click the text "procedure" on the new block and enter "MoveMole" to set the name of the procedure.

4. Since we want to move the mole, click the My Blocks tab, click the Mole drawer, and drag **Mole.MoveTo** into the procedure, to the right of "do." Note that we need to provide *x* and *y* coordinates.
5. To specify that the new *x* coordinate for the mole should be between 0 and `Canvas1.Width` – `Mole.Width`, as discussed earlier:
    a. Click the Built-In tab to get to the built-in procedures.
    b. Click the Math drawer.
    c. Drag out the **random integer** block, putting the plug (protrusion) on its left side into the "x" socket on **Mole.MoveTo**.
    d. Change the **number 1** on the "from" socket by clicking it and then entering 0.
    e. Discard the **number 100** by clicking it and pressing your keyboard's Del or Delete button, or by dragging it to the trash can.
    f. Click the Math drawer and drag a subtraction (-) block into the "to" socket.
    g. Click My Blocks to get to your components.
    h. Click the Canvas1 drawer and scroll down until you see **Canvas1.Width**, which you should drag to the left side of the subtraction operation.
    i. Similarly, click the Mole drawer and drag **Mole.Width** into the right side of the subtraction block.
6. Follow a similar procedure to specify that the *y* coordinate should be a random integer in the range from 0 to `Canvas1.Height` – `Mole.Height`.
7. Check your results against Figure 3-5.

To try out your **Mole.MoveTo** call, right-click the block and choose Do It. (You may need to restart the app by clicking "Connect to Device" first.) You should see the mole move on your phone screen, going to a different location each time (except in the extremely unlikely case that the random-number generator chooses the same place twice in a row).

Calling MoveMole when the app starts

Now that you've written the `MoveMole` procedure, let's make use of it. Because it's so common for programmers to want something to happen when an app starts, there's a block for that very purpose: **Screen1.Initialize**.

1. Click My Blocks, click the Screen1 drawer, and drag out **Screen1.Initialize**.
2. Click the My Definitions drawer, where you'll see a **call MoveMole** block. (It's pretty cool that you've created a new block, isn't it?!) Drag it out, putting it in **Screen1.Initialize**, as shown in Figure 3-6.

Figure 3-6. Calling the MoveMole procedure when the application starts

Calling MoveMole every second

Making the mole move every second will require the `Clock` component. We left `Clock1`'s `TimerInterval` property at its default value of 1,000 (milliseconds), or 1 second. That means that every second, whatever is specified in a **Clock1.Timer** block will take place. Here's how to set that up:

1. Click My Blocks, click the Clock1 drawer, and drag out **Clock1.Timer**.
2. Click the My Definitions drawer and drag a **call MoveMole** block into the **Clock1 .Timer** block, as shown in Figure 3-7.

Figure 3-7. Calling the MoveMole procedure when the timer goes off (every second)

If that's too fast or slow for you, you can change `Clock1`'s `TimerInterval` property in the Component Designer to make it move more or less frequently.

Keeping Score

As you may recall, you created two labels, `HitsCountsLabel` and `MissesCountsLabel`, which had initial values of 0. We'd like to increment the numbers in these labels whenever the user successfully touches the mole (a hit) or taps the screen without touching the mole (a miss). To do so, we will use the **Canvas1.Touched** block, which indicates that the canvas was touched, the *x* and *y* coordinates of where it was touched (which we don't need to know), and whether a sprite was touched (which we do need to know). Figure 3-8 shows the code you will create.

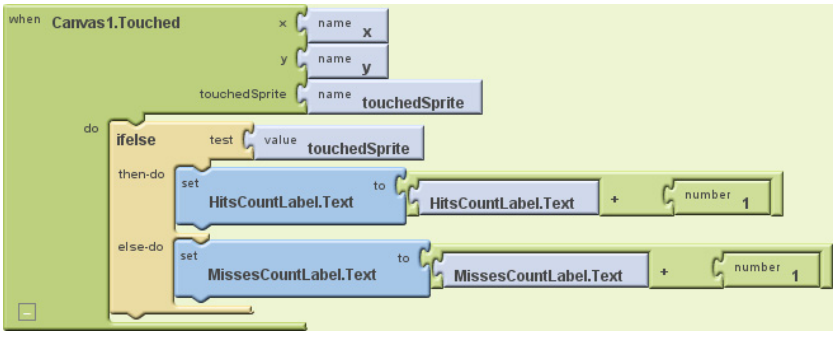


Figure 3-8. Incrementing the number of hits (HitsCountLabel) or misses (MissesCountLabel) when Canvas1 is touched

Figure 3-8's translation is whenever the canvas is touched, check whether a sprite was touched. Since there's only one sprite in our program, it has to be `Mole1`. If `Mole1` is touched, add one to the number in `HitsCountLabel.Text`; otherwise, add one to `MissesCountLabel.Text`. (The value of touchedSprite is false if no sprite was touched.)

Here's how to create the blocks:

1. Click My Blocks, click the Canvas1 drawer, and drag out **Canvas1.Touched**.
2. Click Built-In, click the Control drawer, and drag out **ifelse**, placing it within **Canvas1.Touched**.
3. Click My Blocks, click the My Definitions drawer, and drag out **touchedSprite** and place it in **ifelse**'s test socket.
4. Since we want `HitsCountLabel.Text` to be incremented if the test succeeded (if the mole was touched):
   a. From the HitsCountLabel drawer, drag out the **set HitsCountLabel.Text to** block, putting it to the right of "then-do."
   b. Click Built-In, click the Math drawer, and drag out a plus sign (**+**), placing it in the "to" socket.
   c. Click My Blocks, click the HitsCountLabel drawer, and drag the **HitsCountLabel.Text** block to the left of the plus sign.
   d. Click Built-In, click the Math drawer, and drag a **number 123** block to the right of the plus sign. Click 123 and change it to 1.
5. Repeat step 4 for `MissesCountLabel` in the "else-do" section of the **ifelse**.

Test your app. *You can test this new code on your phone by touching the canvas, on and off the mole, and watching the score change.*

Procedural Abstraction

The ability to name and later call a set of instructions like MoveMole is one of the key tools in computer science and is referred to as *procedural abstraction*. It is called "abstraction" because the caller of the procedure (who, in real-world projects, is likely to be different from the author of the procedure) only needs to know what the procedure does (moves the mole), not how it does it (by making two calls to the random-number generator). Without procedural abstraction, big computer programs would not be possible, because they contain too much code for one person to hold in his head at a time. This is analogous to the division of labor in the real world, where, for example, different engineers design different parts of a car, none of them understanding all of the details, and the driver only has to understand the interface (e.g., pressing the brake pedal to stop the car), not the implementation.

Some advantages of procedural abstraction over copying and pasting code are:

- It is easier to test code if it is neatly segregated from the rest of the program.
- If there's a mistake in the code, it only needs to be fixed in one place.
- To change the implementation, such as making sure that the mole doesn't move somewhere that it appeared recently, you only have to modify the code in one place.
- Procedures can be collected into a library and used in different programs. (Unfortunately, this functionality is not currently supported in App Inventor.)
- Breaking code into pieces helps you think about and implement the application ("divide and conquer").
- Choosing good names for procedures helps document the code, making it easier for someone else (or you, a month later) to read.

In later chapters, you will learn ways of making procedures even more powerful: adding arguments, providing return values, and having procedures call themselves. For an overview, see Chapter 21.

Resetting the Score

A friend who sees you playing MoleMash will probably want to give it a try too, so it's good to have a way to reset the number of hits and misses to 0. Depending on which tutorials you've already worked through, you may be able to figure out how to do this without reading the following instructions. Consider giving it a try before reading ahead.

What we need is a **ResetButton.Click** block that sets the values of `HitsCountLabel.Text` and `MissesCountLabel.Text` to 0. Create the blocks shown in Figure 3-9.

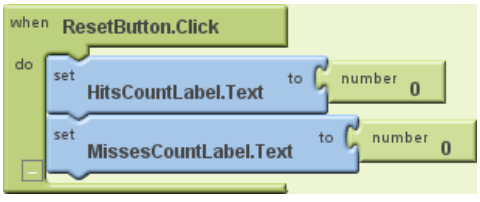


Figure 3-9. Resetting the number of hits (HitsCountLabel) and Misses (MissesCountLabel) when the Reset button is pressed

At this point, you probably don't need step-by-step instructions for creating a button click event handler with text labels, but here's a tip to help speed up the process: instead of getting your number from the Math drawer, just type 0, and the block should be created for you. (These kinds of keyboard shortcuts exist for other blocks, too.)

Test your app. *Try hitting and missing the mole and then pressing the Reset button.*

Adding Behavior When the Mole Is Touched

We said earlier that we want the phone to vibrate when the mole is touched, which we can do with the **Sound1.Vibrate** block, as shown in Figure 3-10. Note that the parameter names `x1` and `y1` are used in **Mole.Touched** because `x` and `y` have already been used in **Canvas1.Touched**.

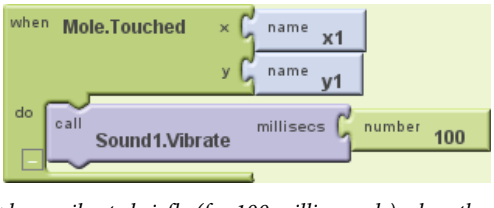


Figure 3-10. Making the phone vibrate briefly (for 100 milliseconds) when the mole is touched

Test your app. *See how the vibration works when you actually touch the mole. If the vibration is too long or too short for your taste, change the number of milliseconds in* ***Sound1.Vibrate****.*

The Complete App: MoleMash

Figure 3-11 illustrates the blocks for the complete MoleMash app.

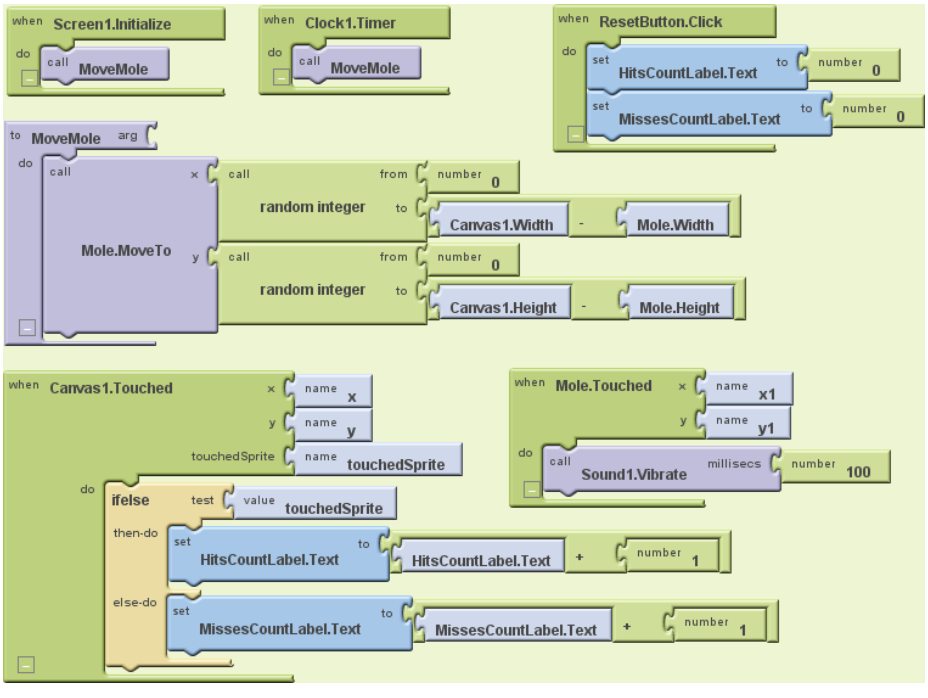


Figure 3-11. The complete MoleMash application

Variations

Here are some ideas for additions to MoleMash:

- Add buttons to let the user make the mole move faster or slower.
- Add a label to keep track of and display the number of times the mole has appeared (moved).
- Add a second `ImageSprite` with a picture of something that the user should *not* hit, such as a flower. If the user touches it, penalize him by reducing his score or ending the game.
- Instead of using a picture of a mole, let the user select a picture with the `ContactPicker` component.

Summary

In this chapter, we've covered a number of techniques useful for apps in general and games in particular:

- The `Canvas` component makes use of an x-y coordinate system, where x represents the horizontal direction (from 0 at the left to `Canvas.Width-1` at the right) and y the vertical direction (from 0 at the top to `Canvas.Height-1` at the bottom). The height and width of an `ImageSprite` can be subtracted from the height and width of a `Canvas` to make sure the sprite fits entirely on the `Canvas`.
- You can take advantage of the phone's touchscreen through the `Canvas` and `ImageSprite` components' `Touched` methods.
- You can create real-time applications that react not just to user input but also in response to the phone's internal timer. Specifically, the **Clock.Timer** block runs at the frequency specified in the `Clock.Interval` property and can be used to move `ImageSprite` (or other) components.
- Labels can be used to display scores, which go up (or down) in response to the player's actions.
- Tactile feedback can be provided to users through the `Sound.Vibrate` method, which makes the phone vibrate for the specified number of milliseconds.
- Instead of just using the built-in methods, you can create procedures to name a set of blocks (**MoveMole**) that can be called just like the built-in ones. This is called procedural abstraction and is a key idea in computer science, enabling code reuse and making complex applications possible.
- You can generate unpredictable behavior with the **random integer** block in the Math drawer, making a game different every time it is played.

You'll learn more techniques for games, including detecting collisions between moving `ImageSprite` components, in Chapter 5 (Ladybug Chase).