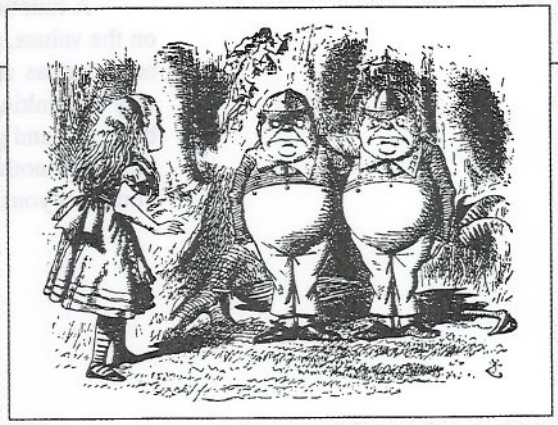
Part III
Using Functions and Control Statements

Part III
Using Functions and Control Statements

Chapter 6
Functions and *If/Else*



"I know what you're thinking about," said Tweedledum: "but it isn't so, nohow." "Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be: but as it isn't, it ain't. That's logic."

This chapter presents functions and conditional execution control statements (two key concepts in programming), as used with methods. Functions allow you to check certain conditions within a world while an animation is running. While Alice provides some built-in functions that are useful in methods, in this chapter we will look at examples of programs where we want to use a function that does not already exist in Alice. In such a situation, you can write your own function.

A function is similar to a method in that it is a collection of instructions and (like a method) is called. The purpose of a method is to perform an animation, but that of a function is to return a value. What difference does this make? Well, an animation performed by a method moves, turns, or performs some other action with objects in the world. We call this changing the state of the world. In a function, however, objects do not move, turn, or perform some other action. So, functions leave the state of the world unchanged. Mathematicians say these are pure functions.

An exciting thing about writing your own function is you can then use it (along with built-in functions) to check out a current condition in the world and make a decision about whether (or not) a method is called. In Alice, we use an *If/Else* statement as a conditional execution control statement that uses a condition for making a decision.

Section 6-1 introduces the use of functions in methods and explains how to write your own function as a method that returns a value. We need functions as a way of getting information about objects as the program is running (at runtime).

In Section 6-2, we use conditional execution (*If/Else* statement) to make decisions about whether (or not) to call a method. We will write our own Boolean function (which returns *true* or *false*) to be used as part of the execution control mechanism. In your programs, you will use decisions to make animations execute in different ways depending on a condition, such as whether an object is visible or where an object is located. In an interactive world, we can use *If/Else* to decide what is done in an event handling method.

6-1 Functions

In writing program code, we often need some information about the objects in the world and the World itself. The most commonly used properties (such as *color* and *opacity*) are listed in a properties list for the object. To get information about other properties (for example, *height*

or *width*) we use a function to ask a question. The value returned by a function can be a number, an object, a Boolean (*true* or *false*), or some other type.

A function may receive values sent in as arguments (input), perform some computation on the values, and return (send back) a value as output. In some cases no input is needed, but often values are sent in. The diagram in Figure 6-1-1 outlines the overall mechanism. One way of thinking about a function is that it is something like an ATM machine. You enter your bankcard and password as input and click on the button indicating you would like to see your balance (another input). The ATM machine looks up your account information and then tells you what your current balance is (as output).

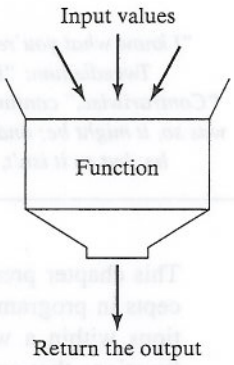


Figure 6-1-1. Overview of how a function works

Abstraction

As with class-level and world-level methods, one important benefit of a function is that it allows us to think about the overall process rather than all the nitty-gritty little details. When we use an ATM, for example, we think about getting the balance in our account—not about all the operations that are going on inside the machine. In the same way, we can call a function in our program to perform all the small actions, while we just think about what answer we are going to get. Like methods, functions are an example of abstraction—collecting lots of small steps into one meaningful idea to allow us to think on a higher plane.

Using a built-in function in a method

Alice provides built-in functions that can be used to provide information for instructions in a method. As an example, consider the world shown in Figure 6-1-2. In setting up this world, we put the toyball (Sports) on the ground next to the net (Sports) and then move it 1 meter away from the net (so that we know the ball is exactly 1 meter from the net).

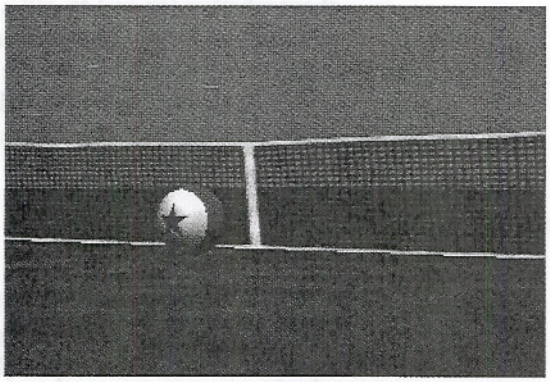


Figure 6-1-2. A world containing a toyball and a tennisNet

We want to bounce the ball over the tennis net. Do not be deceived. This is not as easy as it sounds because we cannot easily tell just by looking at the ball what its orientation is. In other words, we don't know "which way is up" in terms of the ball's sense of direction. The ball should move up and forward and then down and forward:

Actually, we are thinking about the ball's up and down motion relative to the ground, so we need to align the ball's sense of direction with the ground. Orientation is done by using an *orient to* method, as shown in Figure 6-1-3.

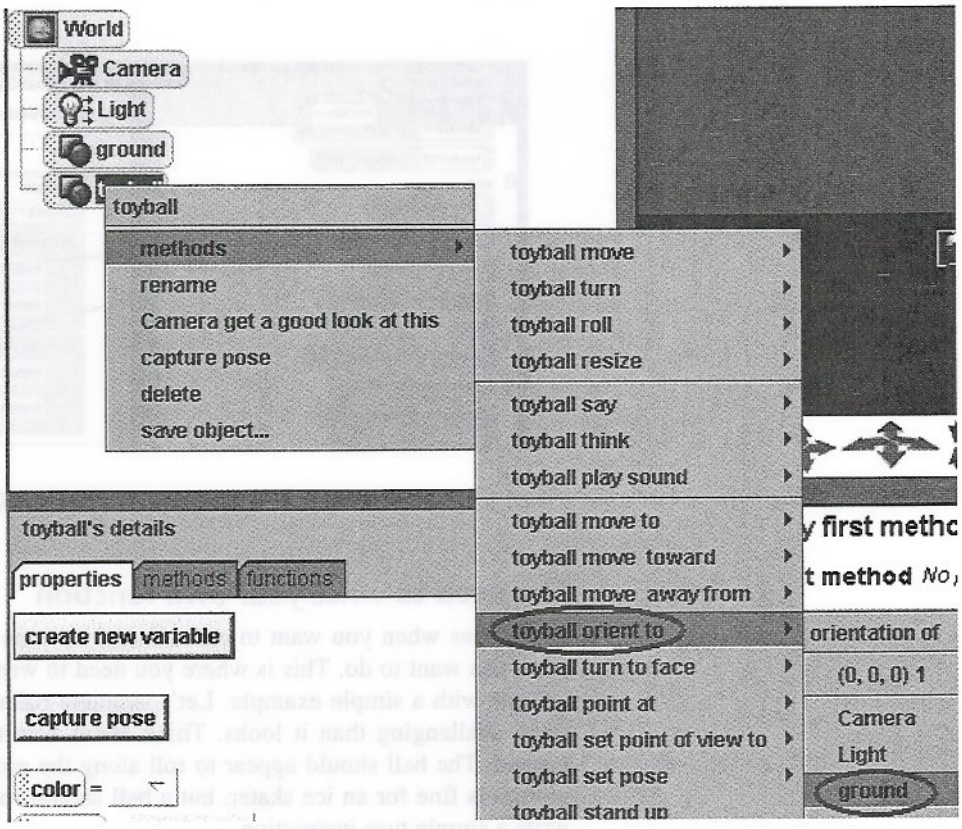


Figure 6-1-3. Using an *orient to* method to orient the ball with the ground

Having properly set up the initial scene, we can write a method to bounce the ball over the net. Since two objects are involved in this action, we will write a world-level method named *ballOverNet*.

A storyboard would look like this:

```
ballOverNet

Do in order
  toyball turn to face the net
  Do together
    toyball move up
    toyball move forward
  Do together
    toyball move down
    toyball move forward
```

We know how far to move the ball forward because we set up the world with the ball exactly 1 meter from the net. We do not know how far up to move the ball to clear the net. We can use a built-in function for the tennisNet to determine its height and then use that as the distance the ball moves up (and then back down). We don't have to think about what Alice is doing to figure out the height of the tennis net, we can just call the function and get the height. This is a 2-step process, 1) Drag the toyballs's, *move* tile into the editor and select 1 as a default distance. 2) Drag the tennis Net's *height* function tile on top of the 1. Figure 6-1-4 illustrates dragging the tennisNet's *height* function onto the *move up* and *move down* instructions.

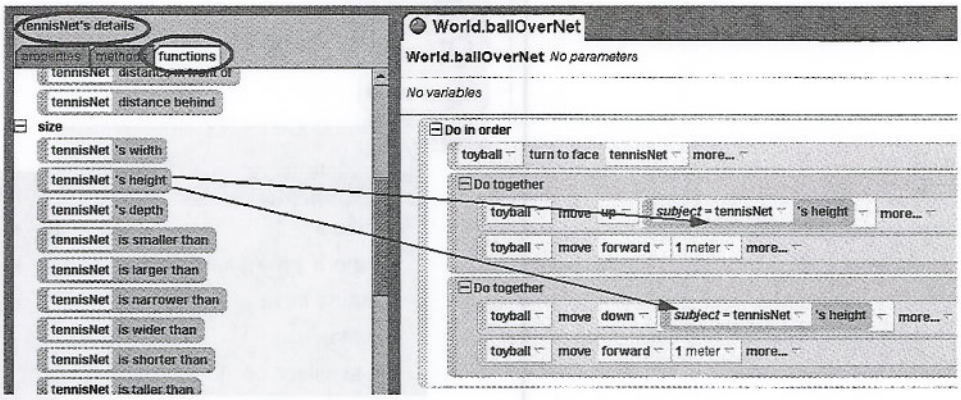


Figure 6-1-4. Dragging the built-in *height* function into a method

The need to write your own function

Sometimes when you want to call a function, none of the built-in functions will work for what you want to do. This is where you need to write your own function. It is helpful to illustrate with a simple example. Let's simulate rolling the ball forward. Once again, this is more challenging than it looks. Think about how the ball can be made to roll along the ground. The ball should appear to roll along the ground, not just glide along it. (A gliding motion is fine for an ice skater, but a ball should roll.) An obvious instruction would be to write a simple *turn* instruction.

Surprise—the turn instruction simply rotates the ball in place! (It spins the ball around, but the ball does not move along the ground.) To actually roll, the ball must turn and also move in the same direction. A storyboard for a method to simulate a rolling action (*realisticRoll*) would be:

```
realisticRoll

Do together
  move ball forward 1 meter
  turn ball forward 1 revolution
```

With this in mind, we can write a class-level method named *realisticRoll*, Figure 6-1-5.

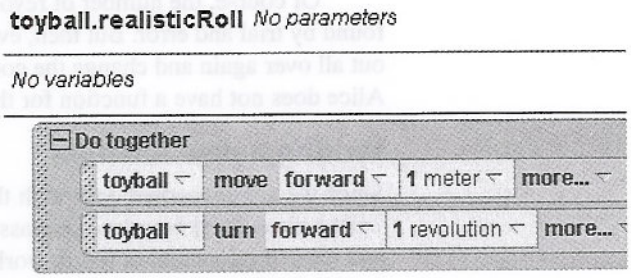


Figure 6-1-5. Move and turn instructions in a *Do together*

Our testing of this code, however, is disappointing. The effect of moving and turning the ball at the same time is that the ball ends up at the same place it started. (The turning action prevents the ball from moving forward.)

Why is this? Well, the ball is moving relative to itself, not relative to the ground. A solution to this problem is to use *asSeenBy = ground* in the *move* statement. The code in Figure 6-1-6 moves and turns the ball forward 1 meter, simulating a real-life ball rolling on the ground.

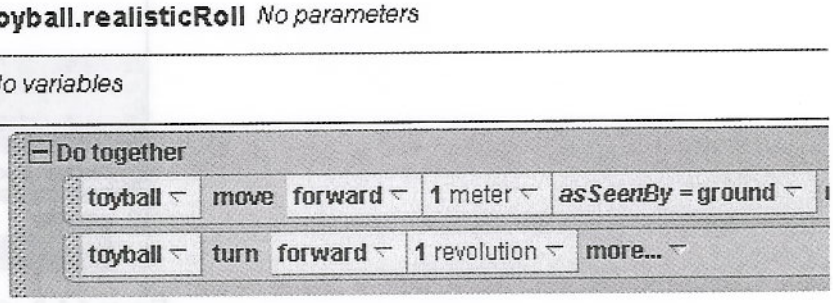


Figure 6-1-6. The toyball rolls along the ground

We guessed that 1 revolution would look realistic. But when it is run, it is not quite right. We must now think about how many revolutions the ball needs to turn in covering a given distance in a forward direction. This presents a challenge because the number of times the ball needs to turn is proportional to the ball's diameter. To cover the same forward distance, a small ball turns more times than a larger ball. In Figure 6-1-7, the larger ball covers the same distance in one revolution as the smaller ball covers in four revolutions.

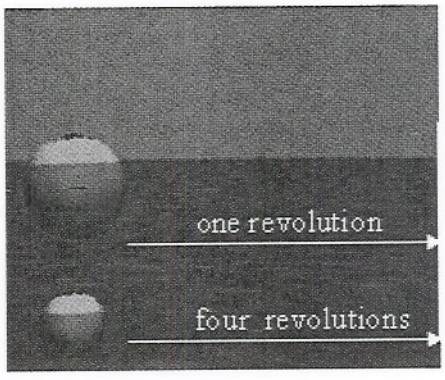


Figure 6-1-7. Distance covered by a revolution is proportional to diameter

Of course, the number of revolutions needed for the ball to roll, say, 10 meters could be found by trial and error. But then, every time the ball changed size you would have to figure it out all over again and change the code. A better way is to compute the number of revolutions. Alice does not have a function for this, so we will write our own.

Writing a new function

Since we are concerned only with the ball rolling, and no other objects are involved, we can write a class-level function. (A class-level function allows you save out the ball as a new class and reuse the function in future worlds.) To write your own class-level function, select the object in the Object tree (for a world-level function you would select the World). In the functions tab, click on the **create new function** button, as in Figure 6-1-8.

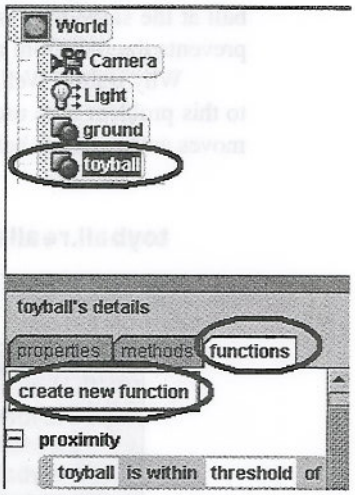


Figure 6-1-8. Creating a new function

A popup New Function box (Figure 6-1-9) allows you enter the name of the new function and select its type. Your new function is categorized by the type of information it returns. The types of functions include Number, Boolean, Object, and Other (such as String, Color, and Sound). In this section, we will write functions that return Number values. Examples and exercises later in the chapter use functions that return other types of values.

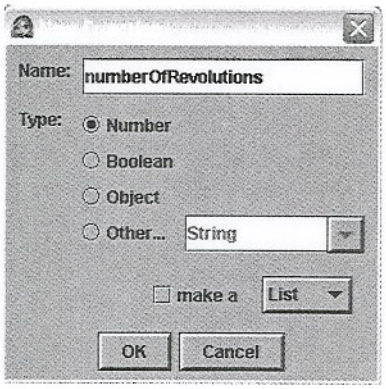


Figure 6-1-9. Enter a name and select a return type

A click on the OK button creates an editor panel where you can write the code for the function, as shown in Figure 6-1-10.

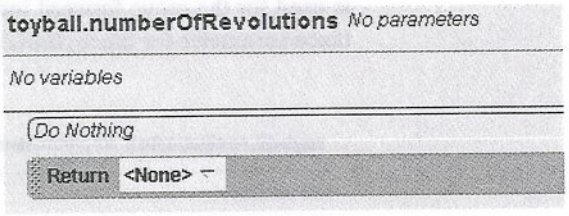


Figure 6-1-10. Editor panel for writing your own function

The Return statement

Every function must have a *Return* statement. The reason is that all functions return some information. When you first create a function, Alice automatically adds a *Return* statement at the end, reminding you that your function must return some information. You cannot remove the *Return* statement.

In our example, we want to ask the function: "How many revolutions does the ball have to make to move a given distance along the ground?" The number of revolutions depends on the distance traveled by the outside (circumference) of the ball in a single revolution, so we use the formula:

$$\text{number of revolutions} = \text{distance}/(\text{diameter} * \pi)$$

To use this formula, we need three pieces of information: the distance the ball is to roll, the diameter of the ball, and something named π. To provide this information,

- A parameter will be used to send in the distance the ball is to roll.
- The built-in function *toyball's width* will be called to get the diameter.
- The symbol π (also known as pi) is a constant value (it does not change). Pi represents the ratio of the circumference of a circle to its diameter. We will use 3.14 as the constant value of pi.

The code shown in Figure 6-1-11 implements the function. The number of rotations is computed by dividing the distance the ball is to move (the *distance* parameter) by the product of the ball's diameter (*toyBall's width,* a built-in function) and pi (described above). The *Return* statement tells Alice to send back the computed answer.

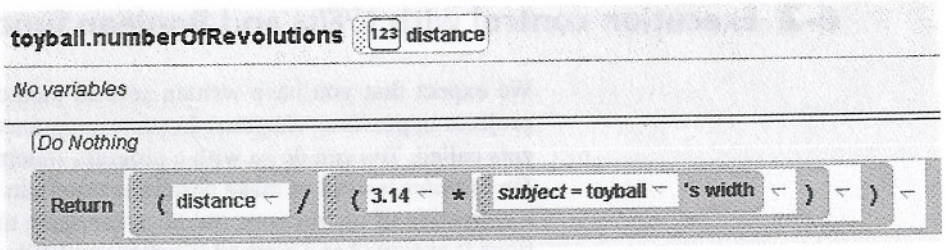


Figure 6-1-11. The *toyBall.numberOfRevolutions* function

The order of evaluation of the values in the function must be carefully arranged. Alice uses nested tiles and parentheses to emphasize the order in which the values are evaluated. The innermost expression is computed first. In this example, *3.14* is multiplied by the *toyBall's width,* and then that value is divided into the *distance*.

Calling the function

Now that the toyBall has a function named *numberOfRevolutions*, the function can be used to revise our *realisticRoll* method, as shown in Figure 6-1-12. An arbitrary distance of 10 meters is used for the move forward instruction. The same value, 10 meters, is also used as the distance parameter for the *toyBall.numberOfRevolutions* function.

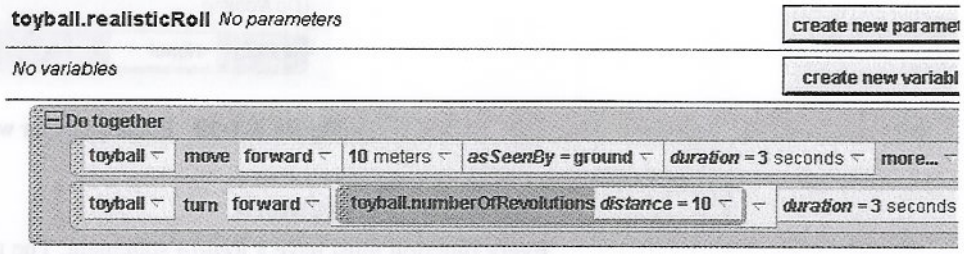


Figure 6-1-12. Calling the *numberOfRevolutions* function

Although 10 meters was used as a test value in the example above, we should really be testing our program by using low distance values (for example, −2 and 0) and also with high distance values (for example, 20). Using a range of values will reassure you that your program code works on many different values. One way to do this is to parameterize the *realisticRoll* method so it can be called with different test values, as illustrated in Figure 6-1-13.

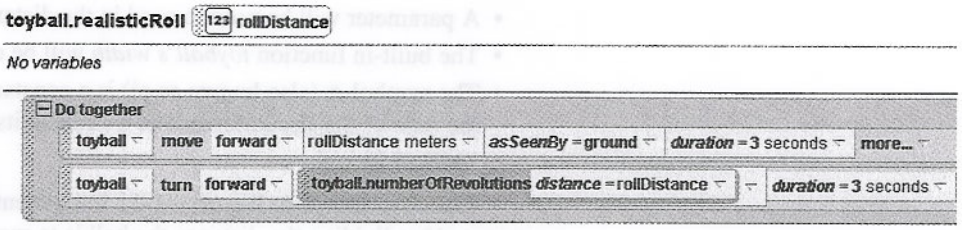


Figure 6-1-13. A parameter for *rollDistance* allows testing with different distance values

6-2 Execution control with *If/Else* and Boolean functions

We expect that you have written several methods as part of programs for exercises and projects in previous chapters. In some programs, you may want to control when a method gets called. You can do so with a program statement called a control structure. In methods, *Do in order* is used to make certain instructions run sequentially and *Do together* to make a collection of instructions run all at the same time. To control whether a block of instructions is executed or a method is called, an *If/Else* statement is used. In this section, we look at the use of *If/Else* statements in Boolean functions and makeing decisions about calling methods.

As stated previously, an *If/Else* is a statement that makes a decision based on the value of a condition as a program is running. (For simplicity, we often refer to it as an *If* statement.) Figure 6-2-1 illustrates the processing of an *If* statement. The statement checks to see whether

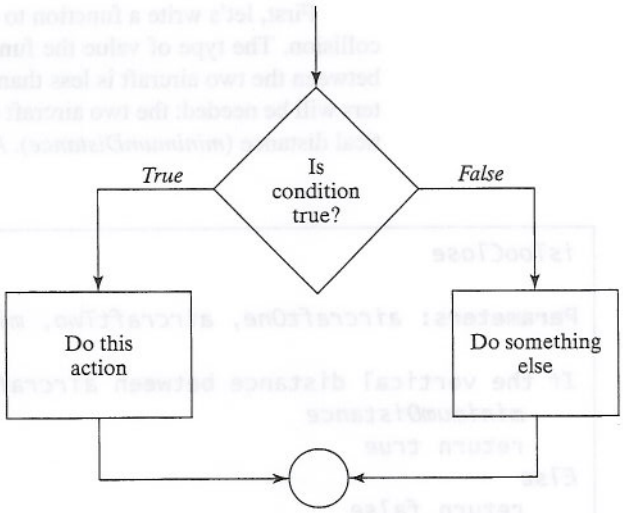


Figure 6-2-1. Describing the processing of an *If* statement

a condition is *true*. If it is, a set of program instructions are run. If the condition is *false*, a separate set of program instructions are run.

Using an *If* statement in a Boolean function

Execution control structures are a big part of game-type and simulation programs. The current conditions guide the actions of the objects in a game. In a basketball game a method is called to increase a team's score only if the ball goes through the net. In a driver-training simulation, a method is called to move the car forward only if the car is still on the road.

As an example of using an *If* statement for conditional execution, consider the world shown in Figure 6-2-2. A biplane (Vehicle) and a helicopter (Vehicle) are flying in the same fly-space at approximately the same altitude near the airport (Buildings). One function of radar and software in the control tower (Buildings) is to check for possible collisions when two vehicles are in the same fly-space. If they are too close to one another, the flight controller can radio the pilots to change their flight paths.

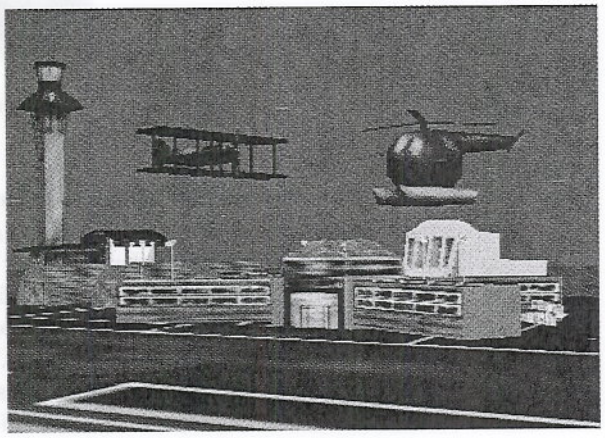


Figure 6-2-2. Fly-space collision danger

First, let's write a function to check whether the biplane and helicopter are in danger of collision. The type of value the function will return is Boolean—*true* if the vertical distance between the two aircraft is less than some minimum distance, otherwise *false*. Three parameters will be needed: the two aircraft objects (aircraftOne and aircraftTwo) and a minimum vertical distance (*minimumDistance*). A storyboard for *isTooClose* is:

```
isTooClose

Parameters: aircraftOne, aircraftTwo, minimumDistance

If the vertical distance between aircraftOne and aircraftTwo is less than
    minimumDistance
    return true
Else
    return false
```

To translate the storyboard to program code, we create a new world-level function named *isTooClose*. The function should return *true* or *false*, so Boolean is selected as the type, as shown in Figure 6-2-3.

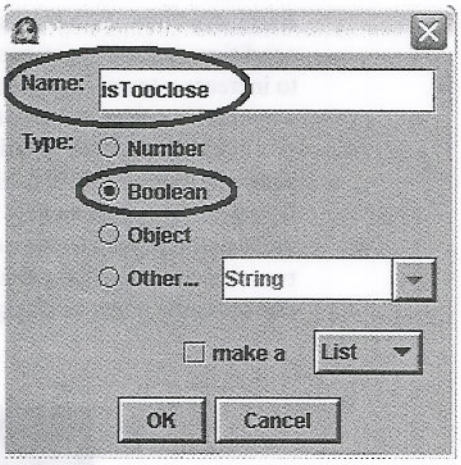


Figure 6-2-3. Creating a Boolean function

In the editor for the function, create the three parameters aircraftOne, aircraftTwo, and *minimumDistance*. Then, drag an *If/Else* tile into the editor, as in Figure 6-2-4. The condition selected from the popup menu for the *If* statement is *true*.

Although *true* was selected as the condition from the popup menu, it is acting as a placeholder. In this example the condition of the *If* statement should be "the vertical distance is less than the *minimumDistance*." To compare the vertical distance between the two aircraft to a minimum distance, we will use a built-in world-level function, the "<" (less than) operation. The "<" is one of six relational operators used in the World's build-in *math* functions, as shown in Figure 6-2-5.

A relational operator computes a *true* or *false* value based on the relationship between the two values. For example, "==" is "is equal to" and "!=" is "is not equal to."

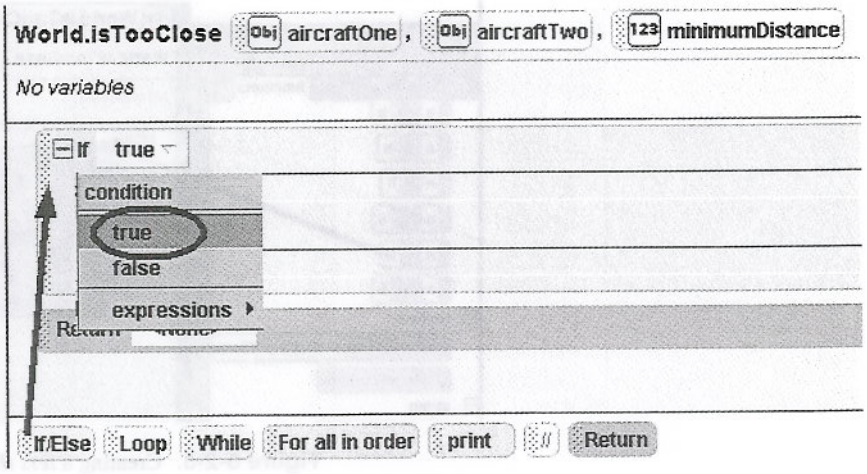


Figure 6-2-4. Dragging an *If* statement into a function and selecting *true* condition

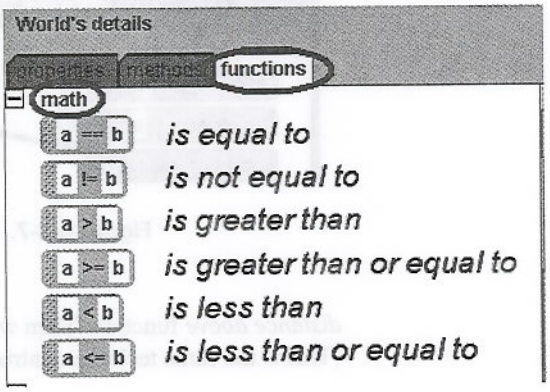


Figure 6-2-5. Relational operations in the built-in World functions

In this example, we want to check whether the distance between the two aircraft is less than a minimum distance. If this condition is true, the air traffic controller will tell the pilots to move the aircraft apart.

To create a conditional expression for the *If* statement, we drag a *less than* function ($a < b$) from the World's functions into the editor and drop it on top of *true*, as in Figure 6-2-3. From the popup menu, we select 1 (for a) and *minimumDistance* (for b). (The vertical distance between the two aircraft is not an option in this popup menu as a choice for a, so we arbitrarily selected 1 as a placeholder. We will show you how to replace the 1 with the actual vertical distance in the next paragraph.)

We want to replace the placeholder (1) with the vertical distance between the two aircraft. Since we don't know whether the helicopter is above the biplane or the biplane is above the helicopter, we use an *absolute value* function. The absolute value of 4 is 4 and the absolute value of -4 is also 4. In other words, absolute value ignores a negative sign. To use absolute value, drag the absolute value function on top of the 1, as shown in Figure 6-2-7.

Now, we want to replace the value 1 with a *distance above* function to determine the distance of aircraftOne above aircraftTwo. Of course, aircraftOne is a parameter, not an actual object. So, we select the biplane from the Object tree to serve as a specific object to create the

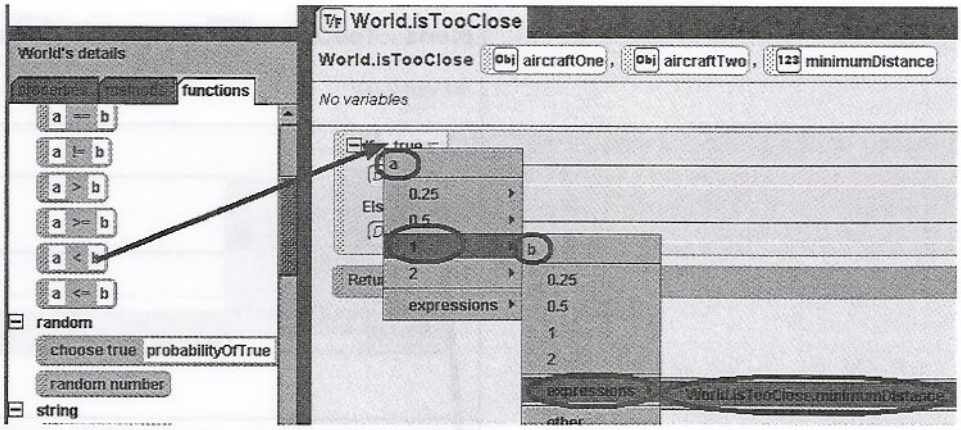


Figure 6-2-6. Creating a *less than* condition for an *If* statement

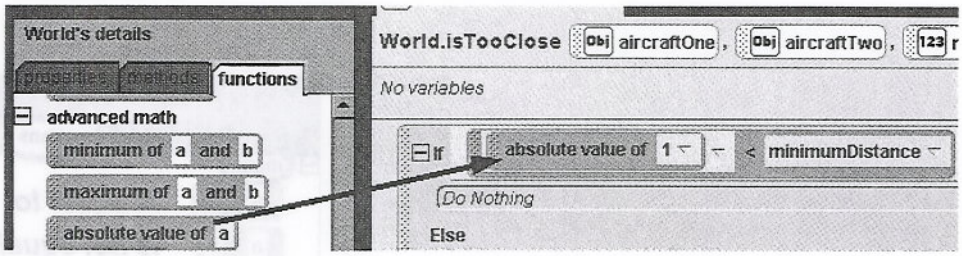


Figure 6-2-7. Drag the *absolute value* function on top of the 1

distance above function. Then we drag aircraftOne in to replace the biplane, as in Figure 6-2-8. (This is the same technique introduced in the Zeus world in Chapter 5, Section 2.)

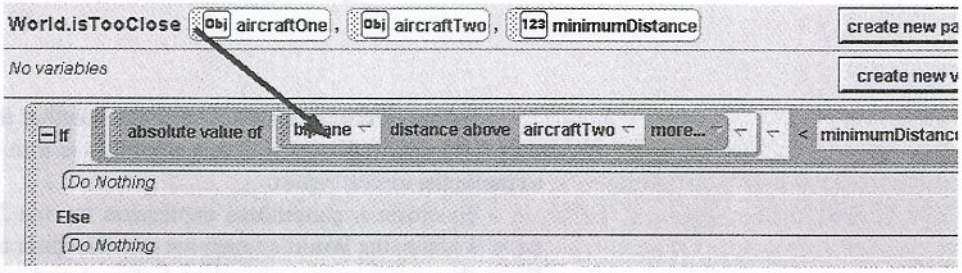


Figure 6-2-8. Creating an expression to compute the vertical distance

The vertical distance can now be compared to *minimumDistance*. Return statements are dragged into the *If* and *Else* parts of the *If* statement. If the vertical distance is less than *minimumDistance*, return *true*; otherwise return *false*. Figure 6-2-9 shows the completed *If* statement in the *isTooClose* function.

Once again, note that the nesting of tiles in a conditional expression creates an order of evaluation. First Alice computes the *distance above*, and then *absolute value* is applied to the result. For example, suppose aircraftOne is 500 meters above the ground and aircraftTwo is

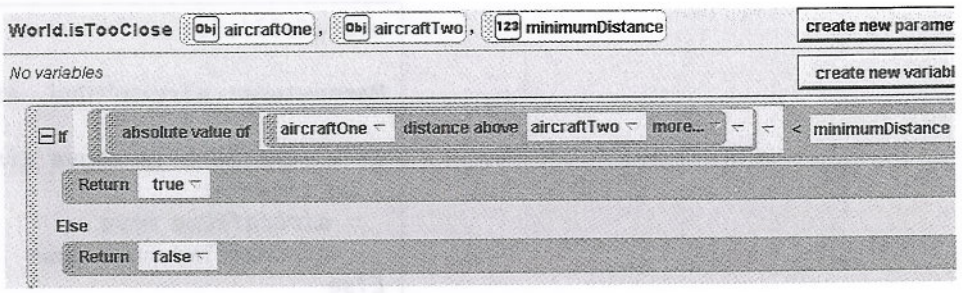


Figure 6-2-9. The *isTooClose* function

520 meters above the ground. Then, the distance of aircraftOne above aircraftTwo is –20 meters (500 − 520). The absolute value of −20 is 20, so the vertical distance between the two aircraft is 20 meters.

Using an *If* statement to control calling a method

Another common use of an *If* statement is to control whether a method is called. To show how this works, let's continue with the same example. The scenario for this world indicated that if the aircraft were too close, the flight controller should radio the pilots to change their flight path to avoid a collision. The storyboard could look like this:

```
If isTooClose
    avoidCollision
Else
    <Do nothing>
```

The *isTooClose* function is called to check whether the biplane (aircraftOne) and the helicopter (aircraftTwo) are too close to each other. If the function returns *true*, a method named *avoidCollision* is called. Otherwise, nothing is done (the method is not called). Let's pretend that the *avoidCollision* method is already written. Then, the previous storyboard can be implemented as shown in Figure 6-2-10 (10 is an arbitrary minimum distance).

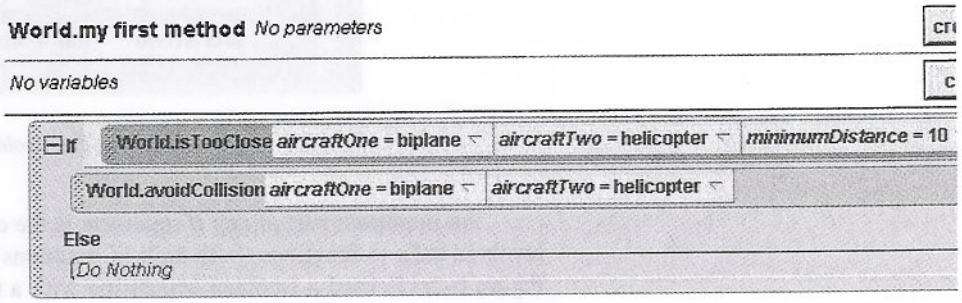


Figure 6-2-10. Controlling a call to a method

In the *avoidCollision* method, the aircraft that is above the other should move up and the lower aircraft should move down. Since we don't know which aircraft is above the other, we can use an *If* statement to check their relative heights and move each one up or down, as needed. The storyboard for the *avoidCollision* method is:

```
avoidCollision

Parameters: aircraftOne, aircraftTwo

If aircraftOne is above aircraftTwo
  Do together
    aircraftOne move up
    aircraftTwo move down
Else
  Do together
    aircraftOne move down
    aircraftTwo move up
```

The translation of this storyboard to program code is shown in Figure 6-2-11. The condition for the *If* statement calls a built-in function, *is above*, to determine which aircraft is above the other. The instructions executed depend on the value returned. If aircraftOne is above aircraftTwo, aircraftOne will move up and aircraftTwo will move down. Otherwise, the *Else* part kicks in so aircraftOne will move down and aircraftTwo will move up. Either way, the aircraft move away from one another.

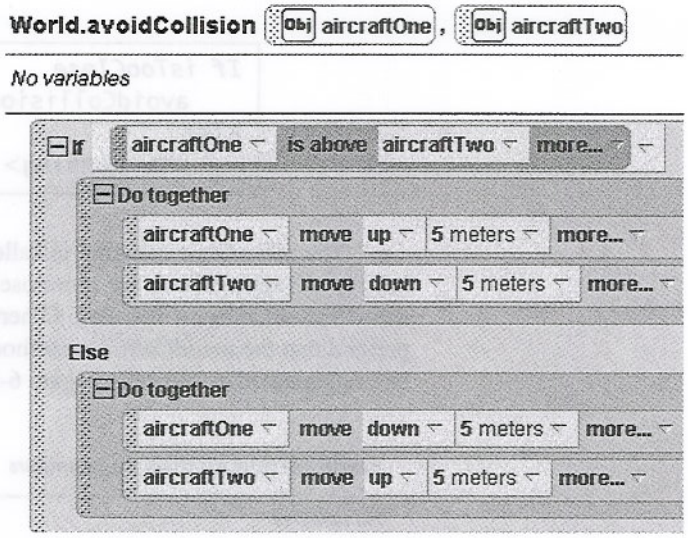


Figure 6-2-11. The *avoidCollision* method

An important part of any *If* statement is the condition. In these examples, the condition involved calls to functions (both built-in functions and our own function). One example (see Figure 6-2-11) used a Boolean expression with a relational operator. Regardless of how the condition is written, it must evaluate to *true* or *false*. The value of the condition determines whether the *If* part or the *Else* part of the *If* statement is executed.

An expression with multiple conditions

In Chapter 5, we created a Zeus interactive world where a user can choose the next target of the god's anger by clicking on a philosopher. Zeus shoots his thunderbolt at the selected philosopher, and special effects are used to show the results. (For convenience, the code for the *shootBolt* method is reproduced in Figure 6-2-12.)

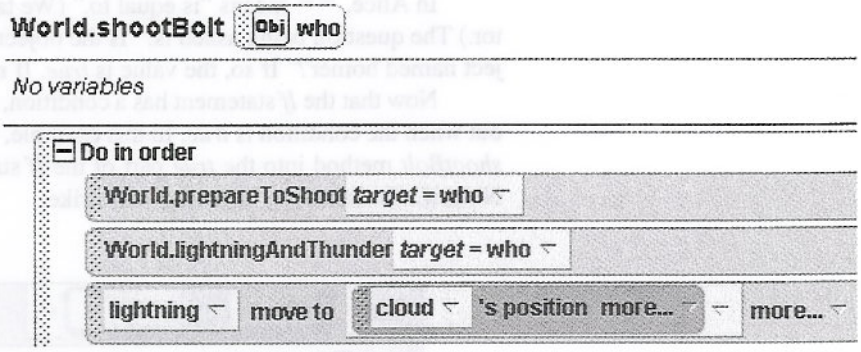


Figure 6-2-12. The *shootBolt* method (from Zeus world in Section 5-2)

We had intended that Zeus shoot the thunderbolt only at a philosopher. Unfortunately, we found that if the user clicks on any other object that happens to be in the world, Zeus zaps that object as well. We want to control the lightning so that the bolt is shot at an object only if *who* is one of the philosophers.

We have four philosophers in this world, so we will need to check whether the object clicked was one of them. This is an example where a conditional expression contains multiple conditions. To show how to use multiple conditions, we begin with a single condition to check for just one philosopher—then expand it to four philosophers.

First, drag an *If* statement into the top line of the *shootBolt* method. Next, create a condition to test whether the object clicked was homer. To create the condition *If who* == homer, drag the *who* tile on top of the *true* tile in the *If* statement. Then select *who* == homer from the popup menu, as shown in Figure 6-2-13.

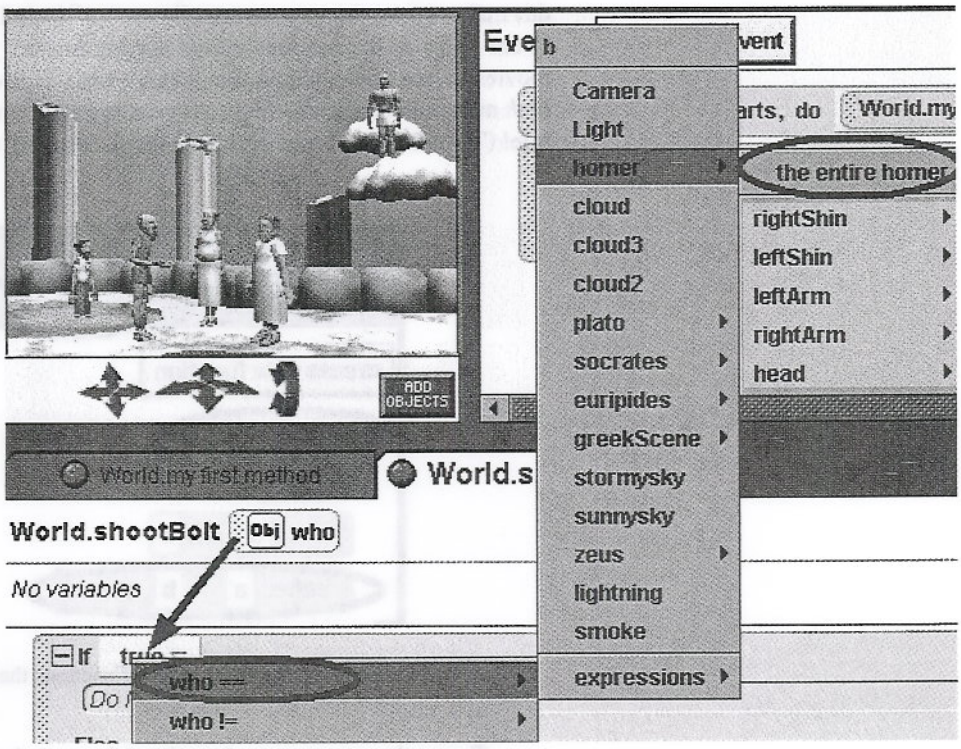


Figure 6-2-13. Selecting *who* == *homer*

In Alice, == means "is equal to." (We talked about "==" above as a relational operator.) The question being asked is: "Is the object sent to the *who* parameter the same as the object named homer?" If so, the value is *true*. If not, the value is *false*.

Now that the *If* statement has a condition, we need to supply the instructions to be carried out when the condition is *true*. In this example, we need only drag the *Do in order* block of the *shootBolt* method into the *true* part of the *If* statement. The result is shown in Figure 6-2-14. Now, if *who* is homer, lightning will strike.

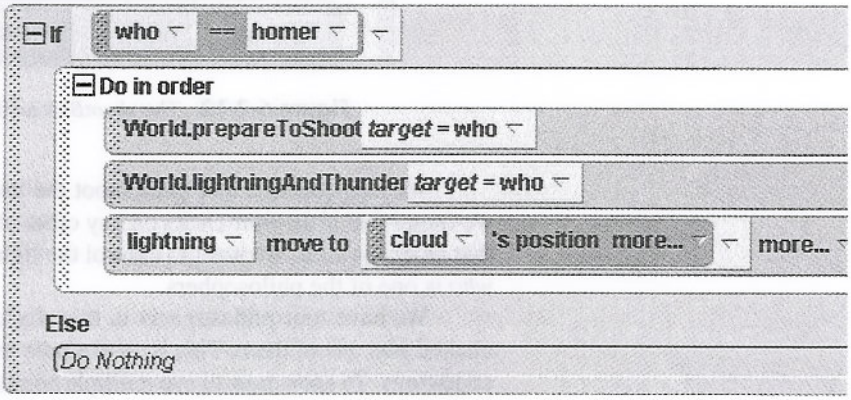


Figure 6-2-14. If *who* == homer, lightning will strike

Now it is time to expand the *If* statement's condition to allow any of the four philosophers. A logical operator "or" is needed (one of three logic operators found in the World functions list, shown in Figure 6-2-15). The *or* operation means "either this or that or possibly both." While this makes sense, the way we actually use "or" in daily life is somewhat different. When you say, "I am going to the movies or I am staying home," you mean that either you are going to the movies or you are staying home, but not both. However, the Alice expression *either a or b, or both* means that either the first statement is true, or the second statement is true, or they are both true! (The other two logic operators are discussed at the end of this section.)

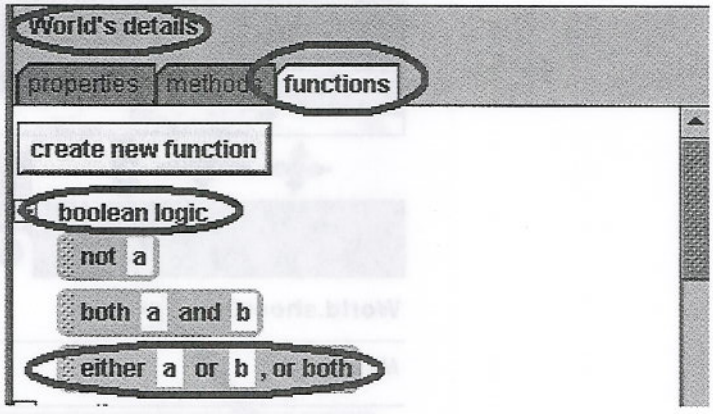


Figure 6-2-15. Functions that use logic operators

To use the *or* operator, drag the *either a or b or both* tile over the condition tile in the *If* statement. In this example, the *or* operator must be dragged into the condition tile three times

to account for all four philosophers. The code below illustrates the modified *If* statement. (The statement is broken into two lines to make it fit on the width of this page—but is all on one line in Alice.)

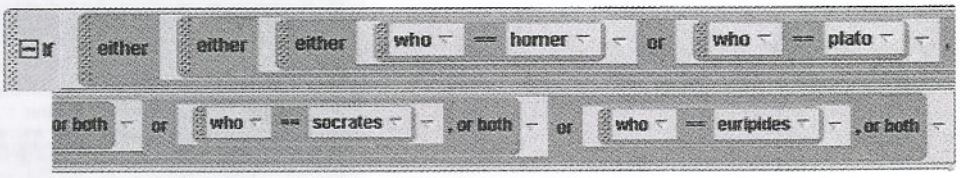


Figure 6-2-16 shows the revision of the *shootBolt* method. (Not all of the multiple conditions can be seen, because the code runs off the edge of the window.) Now, clicking on any one of the four philosophers results in Zeus shooting a thunderbolt at that philosopher, but clicking on something else in the world causes no action.

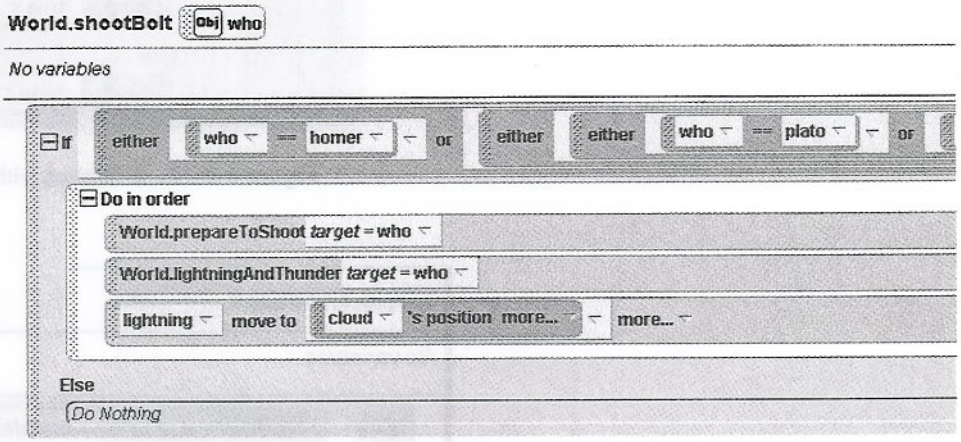


Figure 6-2-16. The *If* statement condition in the *shootBolt* method

A function using nested *If* statements

The condition in the *shootBolt* method above is somewhat complicated and difficult to read. The reason is that four different philosopher objects must be checked using three *or* operators—all in one conditional expression.

An alternative is to write a function with multiple *If* statements. Each *If* statement will check a single condition. Of course, if we find one that works, we do not need to check any further. To make this happen, we nest the *If* statements one inside another, as shown in Figure 6-2-17. The nested *If* statements check one philosopher at a time. If one of the *If* statements is *true*, the function returns *true*; otherwise it returns *false*. This example makes clear an important fact about *Return* statements in a function. As soon as a *Return* statement is executed (as a result of an *If* statement being true), a *true* value is returned and the function is all over. Any remaining statements in the function are skipped. If none of the *If* statement conditions are true, the final *Return false* is executed.

What do we gain by writing a function such as *isPhilosopher*? For one thing, the code in the function is a lot easier to understand. Also, a call to the *isPhilosopher* function can now be used as the condition for the *If* statement in the *shootBolt* method, as shown in Figure 6-2-18.

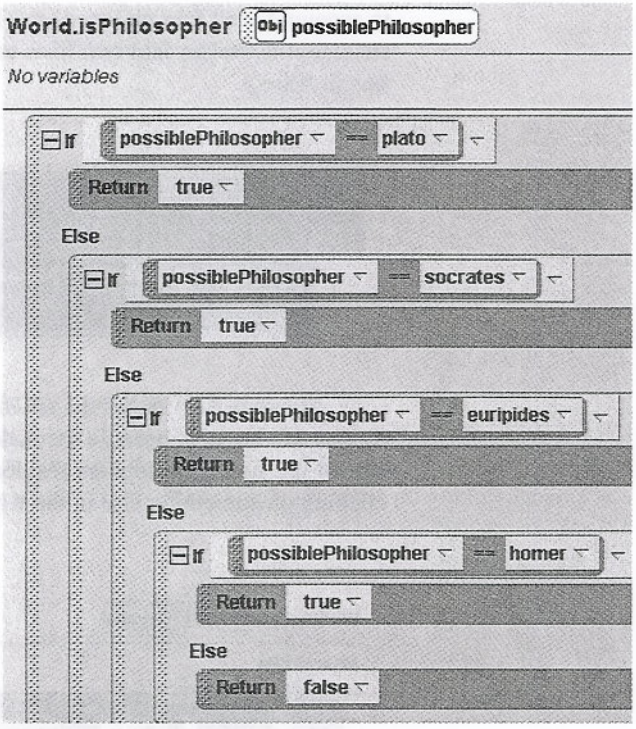


Figure 6-2-17. A function with multiple *If* statements

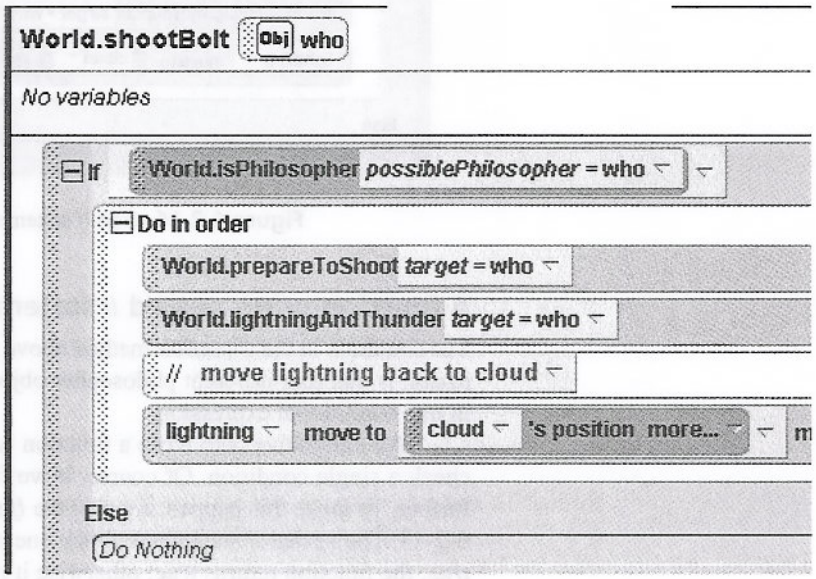


Figure 6-2-18. Calling the *isPhilosopher* function from the *shootBolt* method

Using a parameter in a condition

One last problem (that we know of) still exists in our Zeus world! Clicking on a philosopher who has already been shot by a thunderbolt results in Zeus shooting another thunderbolt at the "already-zapped" philosopher. (That seems like a waste of energy.) How can we prevent this

from happening? One solution is to use another *If* statement to make sure Zeus only zaps a philosopher who isn't already frizzled.

How can we tell if a philosopher has already been zapped? In this example, we turned a zapped philosopher a black color (to dramatize the effect of a lightning strike). The color property can be used to determine whether the object has already been struck by lightning. Once again, we have the problem of a using a parameter as an object. (We saw this problem before in Chapter 5 when we wanted to use a *move to* instruction with the parameter *who*.) We will use the same technique of dragging in an arbitrary object and then replacing the object tile with *who*.

Here are the steps:

1. Drag an *If* statement into the method. In this example, drag it into the line immediately under the first *If* statement.
2. Select one of the philosophers from the Object tree (we arbitrarily chose plato). Drag its color property onto the condition tile and select the != operator and the color black. The result should look like this:

3. Drag the *who* parameter on top of plato to allow the color of any philosopher object to be checked. Now the statement should look like the one in Figure 6-2-19.

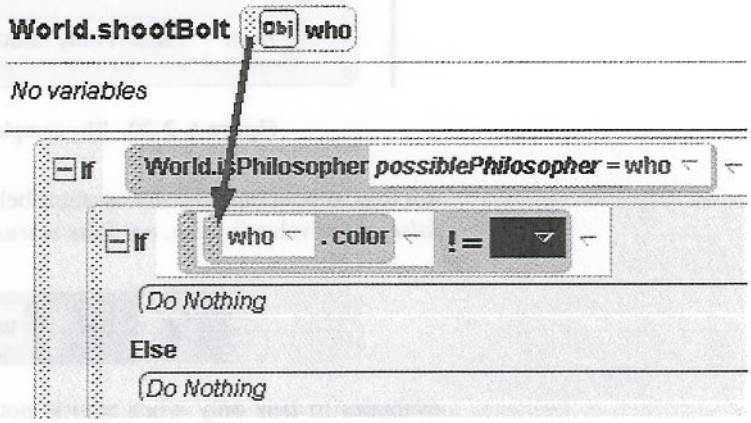


Figure 6-2-19. Replacing plato with the parameter *who*

Code is now added to the *If* and *Else* parts of the *If* statements. The completed code is illustrated in Figure 6-2-20.

The effect of the nested *If* statements is that both *If* statement conditions must be met before the *Do in order* block will be run. If the object clicked is a philosopher and the selected philosopher has not yet been struck by lightning, Zeus shoots his thunderbolt at the philosopher. On the other hand, if the user clicks on an object other than one of the four philosophers, Zeus says, "*I only shoot at philosophers*!" Finally, if the user clicks on a philosopher who has already been hit by a thunderbolt, Zeus now says, "*That philosopher is already zapped!!!*"

Other logic operators

In the Zeus program above, we used an *or* logic operator to create a condition that contained multiple parts. The *or* operator is one of three logic operators available in Alice. Another is the

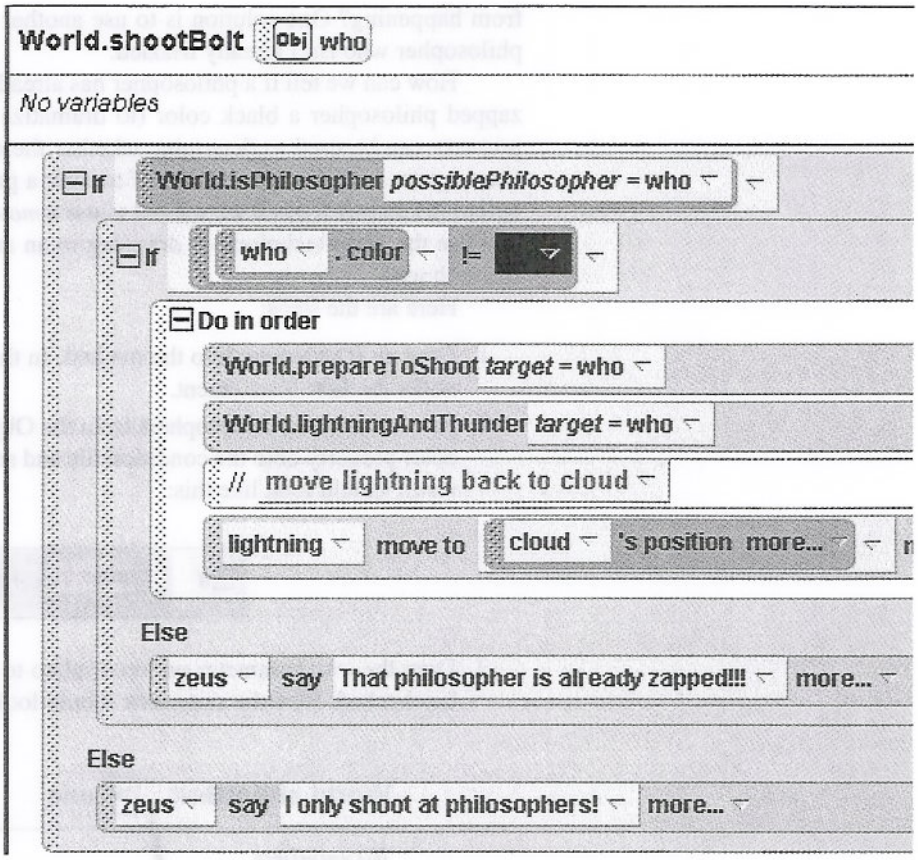
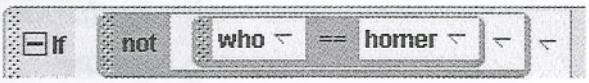


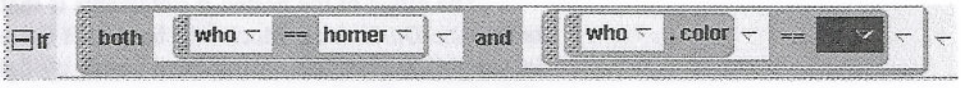
Figure 6-2-20. The complete code for the revised *shootBolt* method

not logical operator. The *not* operator behaves quite logically—if the value is *true*, *not true* is *false*; if the value is *false*, *not false* is *true*. The example



evaluates to *true* only when *who* is not homer. In other words, clicking anywhere on the screen (on one of the other philosophers, on a cloud, etc.) would cause this expression to evaluate to *true*. If the object clicked is homer, the expression evaluates to *false*.

A third logical operator is *and*. The *and* logical operator requires both Boolean expressions to be true in order to evaluate to *true*. The example



evaluates to *true* only if *who* is homer and his color has been set to blue.

It is important to be very careful with expressions containing two or more logical operators. The following expression evaluates to *true* only if *who* is homer and homer's color is black or blue.

This is because the nested tiles represent the expression:

both (who == homer) and (either who.color == blue or who.color == black, or both)

Below is an expression that looks almost the same but is very different. It evaluates to *true* if *who* is black. It will also evaluate to *true* if *who* is homer and his color is blue.

The reason this expression is different is that the logical tiles are nested differently, so the order of evaluation is different. These nested tiles represent the expression:

either (both (who == homer) and (who.color == blue)) or (who.color == black) or both

These examples point out that the level of nesting in logical expressions can be tricky. In general, we recommend not including more than one logical operator in a Boolean expression. If more are needed, we recommend using nested *If* statements. It isn't that we think nested *If* statements are better than combinations of logical operators. Instead, we just realize it can be quite difficult to understand exactly the order of evaluation when a Boolean expression has several nested logical operators.

Tips & Techniques 6
Random Numbers and Random Motion

Random numbers

Random numbers play a big role in certain kinds of computing applications. For example, random numbers are used in creating secure encryptions for transmission of information on the internet. An encrypting utility uses random numbers to create a code for scrambling the information stored in a file. Then the scrambled information is transmitted across the net. When the information arrives at the target location it must be decrypted (unscrambled). To decrypt the information, you need to know the code that was used. Various kinds of scientific simulations also make use of random numbers. For example, random numbers are used to create "what-if" situations in weather simulation programs.

To illustrate the use of random numbers in Alice, let's create an animation where an object's motion is random. A random number is created by selecting the world-level function *random number*, shown in Figure T-6-1.

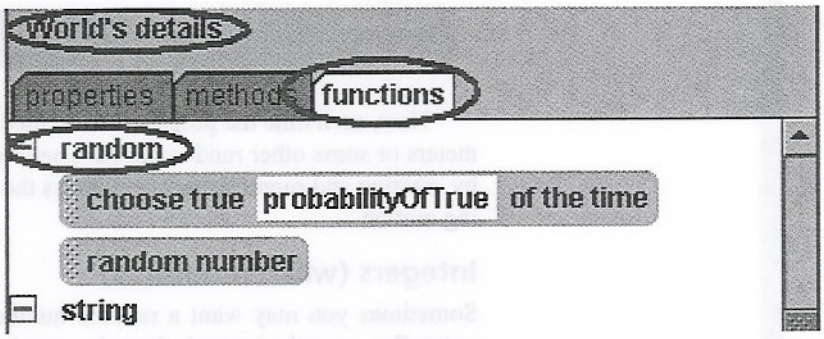


Figure T-6-1. World's *random number* functions

As an example, consider the world scene in Figure T-6-2. An Eskimo (People) and a penguin (Animals) are playing together, sliding around on the ice in a winter world.

Figure T-6-2. Pet penguin sliding on the ice

Suppose you want to write code to make the penguin slide forward a random distance. This easily is accomplished by first creating a *move* instruction and then dragging the *random number* function tile into the distance parameter. The result is shown here:

The *random number* function returns a fractional value between 0 and 1. When the above instruction is executed, the penguin will slide forward some fractional amount between 0 and 1. Although you know the range of values (somewhere between 0 and 1) you don't know the exact number. Also, each time the program is executed the distance is likely to be different. This unpredictability is what makes it random.

Selecting a range for random number

The default range of values for *random number* is 0 to 1. Suppose you want a random number between 1 and 5, instead of between 0 and 1. To specify the range of numbers for the random number function, click *more* in the *random number* tile (purple) and select the *minimum* value. Then, do the same to select a *maximum* value. The instruction with a range of 1 to 5 is shown here:

Now, each time the program is executed, the penguin will slide forward 3.8 meters or 2.3 meters or some other random amount between 1 and 4.999. We suggest that you test this out by creating and running the world. Press the **Restart** button several times to observe the sliding motion.

Integers (whole numbers)

Sometimes you may want a random number that has no digits to the right of the decimal point. For example, instead of numbers such as 3.8, or 2.3 you may want whole numbers such as 2 or 3 or 4. Numbers that are whole numbers (have no digits to the right of the decimal

point) are known as integer values. To use the *random number* function to obtain random integer values, click *more* in the *random number* tile (purple) and select *integerOnly* and *true* in the pull-down menu, as shown in Figure T-6-3. In this example, the generated random number will be 1, 2, 3, or 4.

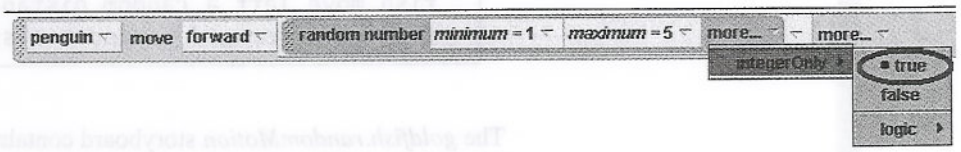


Figure T-6-3. Selecting *integerOnly* for the *random number* function

Random motion

If the *random number* function is combined with the *move* instruction, a form of random motion is created where objects move to a random location. Random motion is essential in many different kinds of games and simulations. As an example, consider the world in Figure T-6-4. The goldfish (Animals) is to swim in a random-like movement. We want to restrict the movement to a nearby position. Otherwise, successive moves would make the goldfish jump around here and there on the screen—not at all like a goldfish swimming.

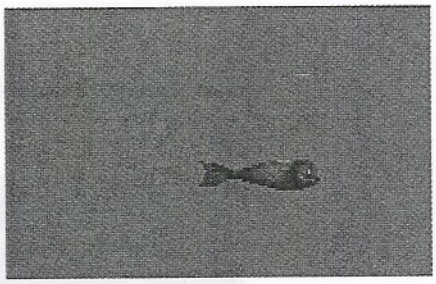


Figure T-6-4. The goldfish in a water world, for a random motion animation

Because Alice objects live in a three-dimensional world, you can write a *move* instruction that moves the goldfish in any of six possible directions (*forward, backward, left, right, up*, and *down*). Combining the *random number* function with six *move* instructions might be a bit clumsy. Let's try to make our work easier by taking advantage of negative numbers. If a negative number is used as the distance in a *move* instruction, the object is to move in the opposite direction. For example, the instruction shown next is to *move* the goldfish *left* −1 meters. When executed, this instruction will actually move the goldfish *right* 1 meter.

If we use both positive and negative distance values, only three *move* instructions are needed. We will use *up, left*, and *forward* directions. With this idea in mind, let's write a storyboard for a *randomMotion* method:

```
randomMotion
Parameters: min, max
Do together
  fish move up a random distance between min and max
  fish move left a random distance between min and max
  fish move forward a random distance between 0 and max
```

The *goldfish.randomMotion* storyboard contains a *Do together* block and three *move* instructions. In each move instruction, the random number function is called to generate a random distance. The parameters (*min* and *max*) are used to select the range of minimum and maximum values for the *random number* distance. In this example, we are going to use a *min* of −0.2 and a *max* of 0.2. So the *random number* function will return a value in the range of −0.2 to 0.2 for the move up and move left instructions. Note that *min* for the *move forward* instruction is 0. This is because goldfish do not swim backwards.

Now we can translate the storyboard into program code. The code for the *randomMotion* method is shown in Figure T-6-5. Think about how this method works. In each *move* instruction, the *random number* function is called to get a random distance. Let's say the *random number* function returns a negative distance for the *move up* instruction; the goldfish will actually move *down*. A negative distance for the *move left* instruction will move the goldfish *right*. The distance for the *move forward* instruction will always be positive, so the goldfish will always move forward.

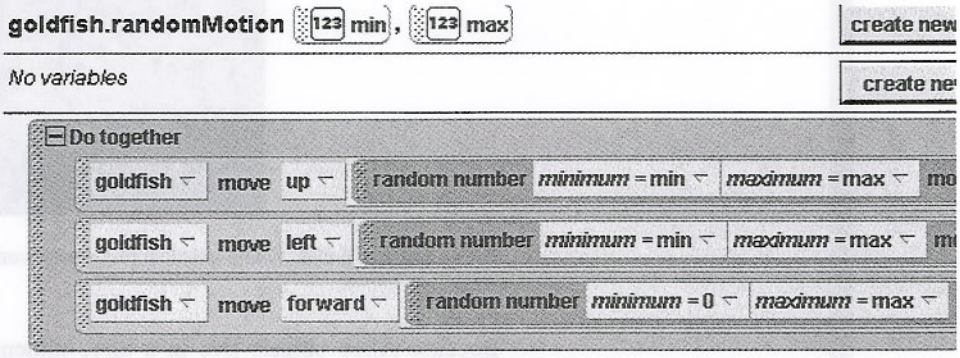


Figure T-6-5. The *randomMotion* method

The *Do together* block executes the *move up, move right,* and *move forward* all at the same time. As a result the goldfish will move to some random location within a cube of space defined by the minimum and maximum random number distances.

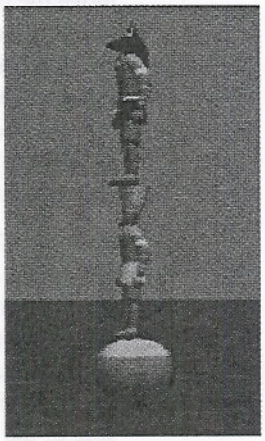
Once the *randomMotion* instruction has been written, it should be tested. To call the *randomMotion* method, *min* and *max* arguments must be passed to the method, as shown below. In this example, the *min* (−0.2) and *max* (0.2) arguments will restrict the movement of the goldfish to a random location near the current location.

Exercises and Projects

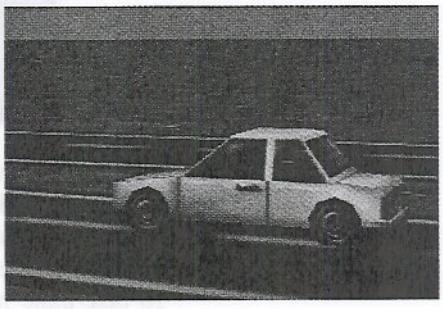
6-1 Exercises

1. *Adding Acrobats to the Rolling Ball*
   Begin by creating the rolling ball world as presented in Section 6-1. Add two objects/acrobats of your own choosing. (we used the Pharaoh and Anabas from the Egypt folder). Position them on top of the ball (Sports), one on top of the other. In the world shown below, we resized the ball to twice its size. Write a program for a circus act, where the acrobats move with the ball, staying on top of it, as the ball rolls. The acrobats should put their arms up half way, to help them balance as they move along with the rolling ball.

   Hint: Use the scene editor quad view to be certain the acrobats are standing directly on top of one another and are centered on the ball. Also, use pull-down menu methods to be sure that the acrobats and the ball all have the same orientation. (See Tips & Techniques 2 on how to use the *orient to* instruction to ensure that objects are synchronized for movement together.)

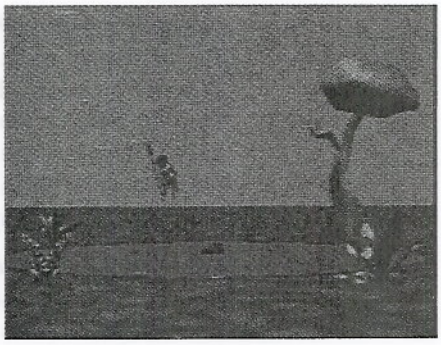


2. *Rotating Tires on a Car or Truck*
   Create a new world with a car or truck (Vehicles). Write a program to make the four wheels of the car turn forward as the car moves forward. The code should be very similar to the code used to make a ball roll forward (see the *realisticRoll* method in Section 6-1).



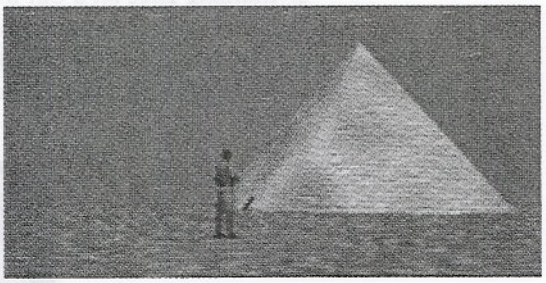
3. *Bee Scout*
   It has been a hot, dry summer and a hive of bees is in desperate need of a new supply of pollen. One bee (Animals/Bugs) has ventured far from the hive to scout for new pollen sources. A natural place to look is near ponds in the area. Set up the initial scene with a circle (Shapes) flat on the ground and colored blue to look like a pond. Add plants,

trees, and other natural scenery including some flowers (Nature). Be sure the bee is located somewhere around the edge of the pond, as shown in the screen shot here.



Write a program to animate the bee scouting the edge of a pond for flowers that are in bloom. The bee is to fly around the perimeter of the pond (circle). Write a method to move the bee scout around the perimeter of the pond in which the circumference of the circle is used to guide the motion. (Yes, *asSeenBy* could be used—but that is not the point of this exercise.) The formula for computing the circumference of a circle is $\pi \times$ the diameter of the circle. π is 3.14 and the diameter is the object's width. Write a function that computes and returns the circumference of the circle. Then have the bee fly around the perimeter of the pond by moving forward the amount of meters returned by the circumference function while turning left one revolution.

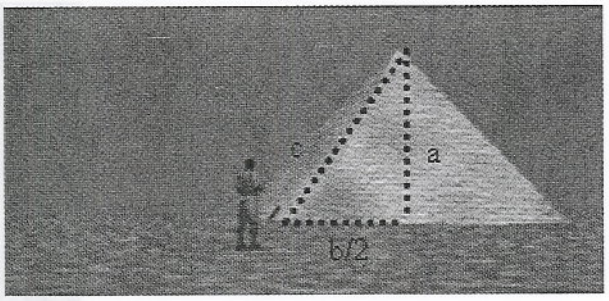
4. *Pyramid Climb*
On spring break, a student is visiting the land of the Pharaohs. The student (for example, randomGuy 1 from the People folder on the CD or Web gallery) decides to climb one of the pyramids (Egypt), starting at the bottom and moving straight up the side. Set up an initial scene consisting of a person and a pyramid, as shown in the screen shot below. Write a method to animate the *climb* so that the person's feet are always on the side of the pyramid.



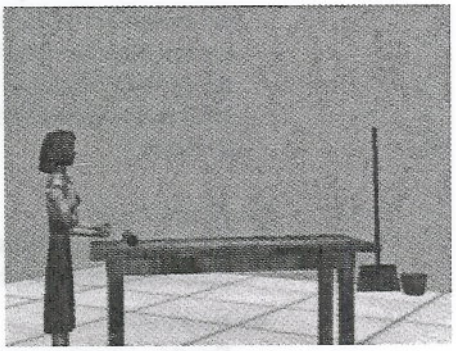
Begin by pointing the person at the pyramid and walking him/her up to edge. Then, turn the person about 1/8 of a revolution so as to lean into the climb. (Play with this leaning movement until you get a reasonable angle for the person to climb the pyramid.) After reaching the top, the person should stand up straight.

To determine how far the person must move to complete the climb, the *climb* method must call a function. The function computes the side length of the pyramid. The formula for computing the distance up the side of the pyramid is based on the Pythagorean theorem ($a^2 + b^2 = c^2$). The value of c will provide a rough estimate of how far the person should move (in a diagonal direction) up the side of the pyramid. The formula is:

$$\text{length of the pyramid's side} = \sqrt{((\text{pyramid's height})^2 + (\text{pyramid's width}/2)^2)}$$



5. *Baking Lessons*
In this exercise, a woman (People) is learning how to make rolled cookies. Before she can cut out the cookies, she needs to roll the dough with her rolling pin (Kitchen). Create a function that determines how much the rolling pin is to roll as she slides it along the table, flattening the dough. Use the function in a world that animates rolling pin rolling across the table (Furniture on CD or Web gallery).

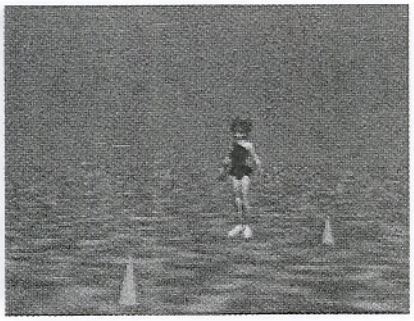


6-2 Exercises

6. *Modifications to the Zeus World*
   (a) Modify the Zeus world to make each philosopher say something different when clicked.
   - Euripides says, "Come on guys, I want to take a bath."
   - Plato says, "I call it Play Doe" and then extends his right hand to show the other philosophers his Play Doe.
   - Homer says, "By my calculations, pretzels are the optimum solid food."
   - Socrates says, "Like sands in the hour glass, so are the days of our lives."

   Use an *If* statement to determine which philosopher was clicked.
   (b) Modify the Zeus world so that if homer gets clicked and zapped by the thunderbolt, he falls over, says "d'oh", and then stands back up again (instead of turning color). Allow repeated clicking on homer, which should result in his repeated falling down and getting back up.

7. *Practice Turns*
Create a skater world, as illustrated below. Import an enhancedCleverSkater object, as designed and created in Chapter 4, Exercise 9. (If you have not created the EnhancedCleverSkater class, an iceSkater can be used from the gallery, but you will have to write your own methods to make her skate forward and skate around an object.)
For this world, write a program to make the skater practice her turns on the ice. First, place the skater 1 meter from the second cone, facing forward. She then skates forward toward

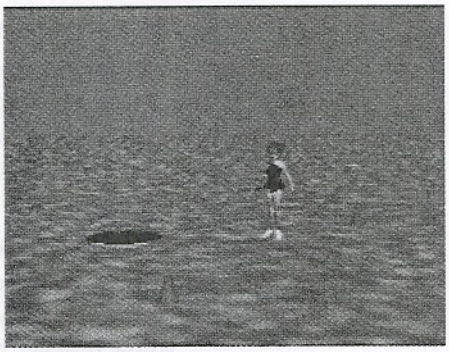


the first cone (a sliding step). When she gets close, she should skate halfway around the cone and end up facing the other way to skate back toward the other cone. Next, have the skater skate toward the other cone and when close enough make a turn around it. In this way, the skater should complete a path around the two cones.

Hint: To find out whether the skater has gotten close enough to do a half-circle turn around the cone, you can use the *is within threshold of object* function for the enhancedCleverSkater. (The phrase *is within threshold of object* evaluates to *true* if the second object is within the specified distance of the first object.) Another possibility is to use the *distance to* function and the relational operator $a < b$ (available as a world-level function) to build the logical expression "Is the skater's distance to the cone less than 2 meters?"

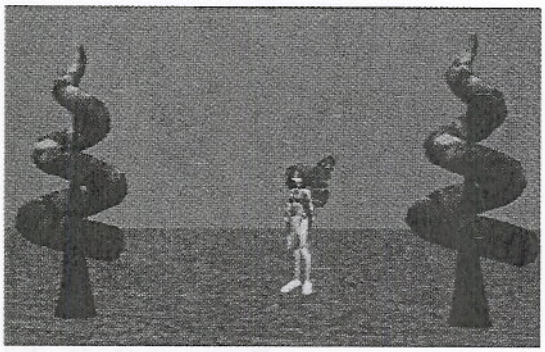
8. *Figure-Eight*
   This exercise is an extension of Exercise 7. Modify the world to have the skater complete a figure-eight around the cones.

9. *Ice Danger*
   For this exercise, you can begin with a world completed in either Exercise 7 or 8—or create a new skater world from scratch. Add a hole in the ice (a blue circle).



Make the world interactive to allow the user to use the mouse to move the hole around on the icy surface. (See Tips & Techniques 5 for using the *let mouse move objects* event.) Now, as the skater is moving across the surface of the ice, the user can move the hole into the skater's path. Modify your method that skates the skater forward to use an *If* statement that checks whether the skater is skating over the hole. If she is on top of the hole, she will drop through it. If you have sound on your computer, you may want to add a splash sound.

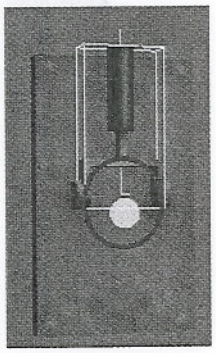
10. *Flying Between Two Trees*
Create a world with a lichenZenspider (Fantasy/Fairies) between two trees (Nature). Animate the lichenZenspider flying back and forth between the two trees. Make the world interactive so that lichenZenspider flies forward a short distance each time the user presses the enter key. LichenZenspider should move forward until she reaches a tree, then turn around to fly back toward the other tree. When she gets to the second tree, she should turn around to fly back toward the first tree. Be sure to avoid lichenZenspider's colliding with a tree.



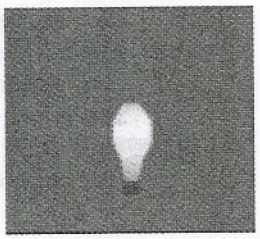
11. *Spanish Vocabulary*
A common exercise in teaching someone a new language is to show the person familiar objects and ask for the word for that object. In this exercise, you are to write a simple vocabulary builder to help someone learn the Spanish word for cat. The scene below shows a cat sitting on the grass and three Spanish words (3D text) displayed in front of the cat. The user is expected to click on the correct Spanish word for cat. A click on any other object will not work. Write a function (*isGato*) that returns *true* if the word (a 3D text object) selected is "gato," and *false* otherwise. The function will use one parameter (*objectClicked*) to send in the 3D text word clicked by the user. If the user clicks on "gato," the cat will say "Si, si!" Otherwise, the cat should turn its head left and right (indicating that the choice was incorrect).

12. *Switch*
Create a world using a Switch object (in the Controls folder of the gallery). Write a method called *flipSwitch* that handles the "when the Switch is clicked" event. It the switch is clicked, its handle will flip from up to down or down to up. Also, write a Boolean function *isHandleUp* which returns *true* if the handle is up and *false* if it is down. (*flipSwitch* will call *isHandleUp* to decide whether to turn the handle forward $\frac{1}{2}$ revolution or backwards one-half revolution.)

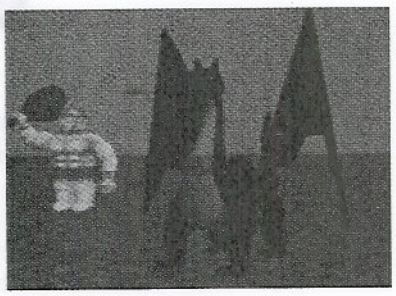
Hint: To write *isHandleUp*, some reference point is needed to test the handle's position. One way to do this is to put a small invisible circle (Shapes folder) on the center of the switch plate. When the handle is up, the handle is above the circle. When the handle is turned down, the handle is below the circle. (See Tips & Techniques 4 for details on moving an object relative to an invisible object.) Do be careful to place the circle immediately below the handles center of gravity, as illustrated below.



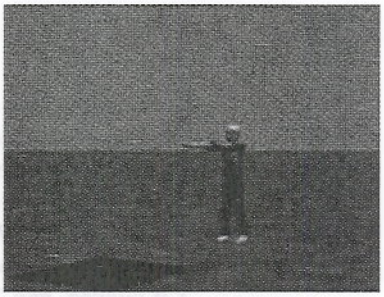
13. *Lightbulb*
Create a world with a lightbulb (Lights) and a method *turnOnOff* that turns the lightbulb on/off depending on whether it is already on/off. When the lightbulb is on, its emissive color property has a value of yellow. When the lightbulb is off, its emissive color is black. Write a Boolean function *isLightOn* that returns *true* if the light bulb is on and *false* if it is off. When clicked, the lightbulb should turn on/off.



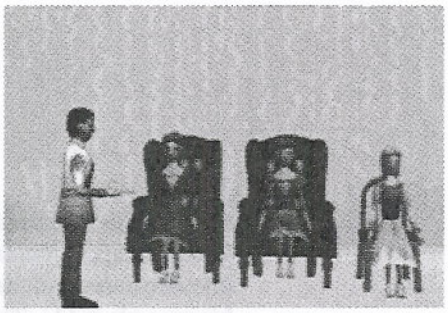
14. *Frighten Away the Dragon*
Create an initial scene of a troll and a dragon (Medieval) as shown below. The troll is trying to frighten away the dragon from his favorite hunting grounds. The troll is to rant and rave while moving toward the dragon if the two are more than 5 meters apart. The troll should move toward the dragon every time the space bar is pressed. Use your own function to find out when the troll gets too close to the dragon. When the troll is less than 5 meters away from the dragon, have the dragon fly away.



15. *Zombie World*
Create a world with a zombie (Spooky) and an open grave (a black square on the ground). In a scene from a scary movie, the zombie walks forward toward the grave and falls in. In this animation, every time the user presses the space bar, the zombie should walk forward. A Boolean function named *isAboveGrave* returns *true* if the zombie is within one-half meter of the grave. When the function returns *true*, make the zombie fall in.



16. *Does the Shoe Fit?*
A common use of animation is to prepare educational software for young children. This animation is taken from a classic children's story. The prince (People) is trying to find the woman whose foot fits the glass slipper. He has come to Cinderella's home, and Cinderella and the two stepsisters (People) are waiting to try on the shoe, as shown in an initial scene below. This world will be interactive in that the user will click on one of the women in the scene. Write a function that returns *true* if the object clicked is Cinderella and *false* otherwise. If the shoe fits (it only fits Cinderella), the prince should move toward Cinderella and ask her to marry him. The glass slipper then appears on Cinderella's foot. If one of the stepsisters is clicked, the shoe will not fit, and the prince should indicate that she is not the one for him.

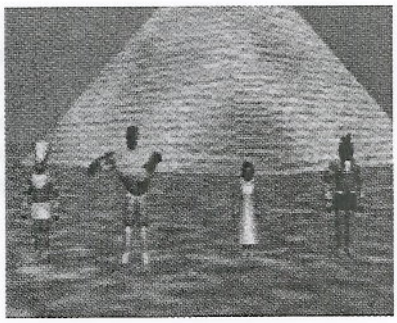


Hint: To make this program easier to write, use two glass slippers: one the prince is holding and one on Cinderella's foot. Make the glass slipper on Cinderella's foot invisible. When the prince proposes marriage, make the shoe in the prince's hand invisible and the one on Cinderella's foot visible.

Note: Do NOT use a "?" in the filename when saving a world.

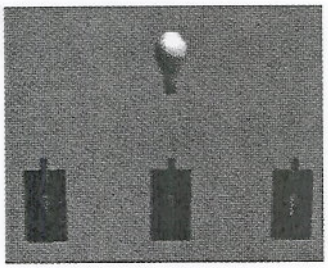
6-2 Projects

1. *Gatekeeper*
Build a world with four different objects of your choice (people, shapes, vehicles, etc.). We used objects from the Egypt folder. Position them in a lineup. The objects are facing the player and are spaced equally from one another.



In this game, one of the objects is a gatekeeper, holding a secret password to allow the user to open a hidden door into the pyramid. To find the secret password, the user must rearrange the objects in the lineup until the gatekeeper is on the far right. (If the objects in the lineup are counted from left to right, 1–2–3–4, the gatekeeper must be moved into position 4.) When an object is clicked, it switches position with the object farthest from it, 1 and 4 will switch with each other if either one is clicked, 3 will switch with 1 if clicked, and 2 will switch with 4 if clicked). Make one of the objects (in position 1, 2, or 3) the gatekeeper. You must use a Boolean function that returns *true* when the objects have been rearranged so the gatekeeper is on the far right of the lineup and *false* if it is not. When the gatekeeper is in position, display a 3D text object containing the password.

2. *Binary Code Game*
Build a world with three switches (Controls) and a lightbulb (Lights), as seen below. Set the *emissive color* property of the lightbulb to *black* (turned-off).



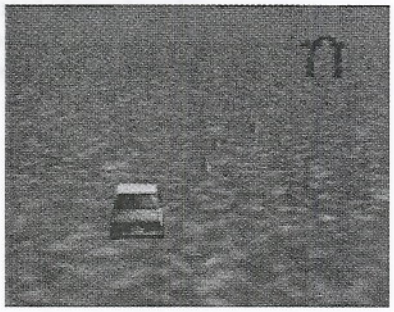
In this game, the positions of the levers on the switches represent a binary code. When a lever is up, it represents 1 (electric current in the switch is high) and when down, 0 (electric current in the switch is low). In the above world, all three levers are up, so the binary code is 111. The correct binary code is chosen at the beginning of the game. (You can enter a given code, and then have a friend try to guess it, or you can use the world-level *random number* function, as described in Tips & Techniques 6.) The idea is to have the user try to guess the correct binary code that will light up the light bulb (its emissive color will be yellow). To guess the binary code, the user will click on the levers to change their position. Each time the user clicks on a lever, it moves in the opposite direction—up (if currently down) or down (if currently up). When all three switches are in the correct position for the binary code, the bulb will turn on.

Each switch should respond to a mouse-click on the switch. If the handle is down, flip it up. If the handle is up, flip it down. To track the current position of a handle, an invisible circle can be placed on the switch and used as a point of reference. When the handle is above the circle, turn it down. When the handle is below the circle, turn it up. See Exercise 12 for more detail.

Your program must include a Boolean function that determines whether the Boolean code is correct.

Hint: You can use the color of the circles (even though they are invisible) as a flag that indicates the correct position of the lever.

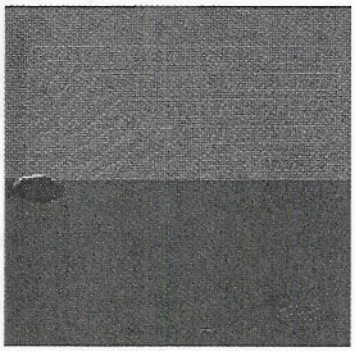
3. *Driving Test*
Create a world that simulates a driving test. The world should have a car (Vehicles), 5 cones (Shapes), and a gate (Spooky). Set up your world as shown in the image below. Also, create two 3D text-phrase objects: "You Pass" and "Try Again". Set the *isShowing* property of each text-phrase to *false*, so that they are not visible in the initial scene.



In this driver test, the user will use arrow-key presses to move the car forward, left, or right to swerve around each of the five cones. If the car hits one of the cones, the driver fails the test, the car stops moving and the "Try Again" text object is made visible. If the user manages to steer the car past all 5 cones, the car should drive through the gate and the "You Pass" text object become visible. Write a function named *isTooClose* that checks the car's distance to a cone. If the car is within 2 meters of the cone, the function returns *true*. It returns *false* otherwise. Also, write a function named *isTestPassed*, which evaluates whether the user has passed the test (i.e., whether the car has been driven past the gate).

Hint: Work under the assumption that the user will not cheat (i.e., pass all the cones and head straight through the gate). Due to the differences in the width and depth of the car, do not be concerned if part of the front or back of the car hits a cone.

4. *Phishy Move*
Phishy fish (Animals) has just signed up at swim school to learn the latest motion, the *sineWave*. Your task is to write a method to teach her the *sineWave* motion. The initial scene with a fish and water is seen here.

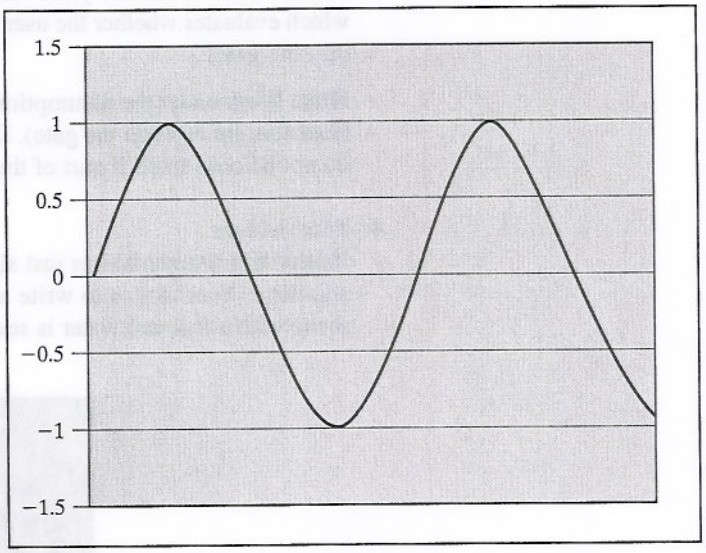


Note: This world is provided on the CD that accompanies this book. We recommend that you use the prepared world, as setting up the scene is time consuming. If you are an adventuresome soul, here are the instructions for setting up the world on your own: Use popup methods to move the fish to the world origin (0,0,and 0) and then turn the fish right one-quarter revolution. Because Phishy is partially submerged, set the opacity of the water to 30% so the fish can be seen in the water. Now, use camera controls to reposition the camera (this takes a bit of patience as the camera must be moved horizontally 180 degrees from its default position). Then, adjust the vertical angle to give a side view of the fish in the water, as seen above. The fish should be located at the far left and the water should occupy the lower half of the world view, as seen in the screen shot above.

Alice has a *sine* function that can be used to teach Phishy the *sineWave* motion. (The sine function is often used to determine the relationship between the lengths of the sides of a right triangle. For the purposes of this animation, that relationship is not really important.) If the sine function is computed over all angles in a full circle, the sine value starts at 0, goes up to 1, back through 0 to −1 and returns to 0:

Angle	Sine of the angle
0	0
45	0.707
90	1
135	0.707
180	0
225	−0.707
270	−1
315	−0.707
360	0

This function is continuous, so if sine values are plotted some multiple of times, we will see the curve repeated over and over:



Sine wave

For the *sineWave* motion, Phishy is to move in the sine wave pattern. In the world provided on the CD, Phishy has been positioned at the origin of the world. In a 2D system, we would say she is at point (0,0). To simulate the sine wave pattern, she needs to move up to height that is 1 meter above the water, then down to a depth of 1 meter below the

surface of the water (−1 meter), back up to 1 meter above the water, and so on, at the same time as she moves to the right in the water. The Alice sine function expects to receive an angle expressed in radians (rather than degrees). Write a function, named *degreesToRadians*, which will convert the angle in degrees to the angle in radians. To convert from degrees to radians, multiply the angle degrees by π and divide by 180. The *degreesToRadians* function should return the angle in radians.

Now, write a method to have Phishy move in the sine wave pattern. Remember that in the world provided on the CD, Phishy has already been positioned at the origin of the world. So, Phishy is already at the position for 0 degrees.

Hint: One way to create the sine wave pattern is to use *move to* instruction (Tips & Techniques 2). A *move to* instruction should move Phishy to a position that is (*right, up*, 0), where *right* is the radian value and *up* is the *sin(radian value)*. After adding a *move to* instruction, the built-in world level function *(right, up, forward)* can be dropped onto the target of the *move to*, allowing individual specification of each coordinate. Use *move to* instructions for angles: 45, 90, 135, 180, 225, 270, 315, and 360. For a smoother animation, make each *move to* instruction have *style = abruptly*. Using *style = abruptly* seems counter intuitive to making an animation run more smoothly. The reason is that animation instructions begin slowly (gently) and end slowly (gently) by default. If we wish to have 2 instructions appear to behave as a single instruction, we do not want each instruction to slow down as the instruction starts/ends. Hence we use the abrupt option.

5. *Cosine Wave*
   Teach Phishy how to move in a cosine wave pattern, instead of the sine wave pattern described in the preceding project.

Summary

Examples in this chapter presented ways to use functions and *If* statements in methods. In Alice, an *If* statement allows for conditional execution of a section of code (based on the value of a Boolean condition, *true* or *false*). The Boolean condition is used to determine whether the *If* part or the *Else* part of the statement will be executed at runtime. Thus, an *If* statement allows us to control the flow of execution of a section of our program code.

Built-in functions do not always meet the particular needs of a program. In this chapter, we looked at how to write your own functions that return different types of values. The benefit of writing your own functions is that you can think about the task on a higher plane—a form of abstraction. Functions that compute and return a number value make program code much less cluttered and easier to read, because the computation is hidden in a function. Such functions will be useful in many situations. For example, we might want to write a function to return the number of visible objects on the screen in an arcade-style game. The function would be called to keep track of how many objects the user has eliminated (made invisible). When only a few objects are left, we might decide to speed up the game.

Parameters can be used to allow a function to be used with different objects. As with class-level methods, you can write a class-level function and save it out with the object. This allows the function to be reused in another program.

Important concepts in this chapter

- An *If* statement is a block of program code that allows for the conditional execution of that code.
- An *If* statement contains a Boolean condition used to determine whether the segment of code will execute.

- If the Boolean condition evaluates to *true*, the *If* part of the statement is executed. If the expression evaluates to *false*, the *Else* part is executed.
- Boolean conditions may call built-in functions that return a *true* or *false* value.
- Logical operators (*and, or, not*) can be used to combine simple Boolean expressions into more complex expressions.
- Relational operators (<, >, >=, <=, ==, !=) can be used to compare values in a Boolean expression.
- Functions can be written to return a Boolean value and used in *If* statements.
- Functions can also be written to compute and return other types of values.

CHAPTER 7
Repetition: Definite and Indefinite Loops

"Are we nearly there?" Alice managed to pant out at last.
"Nearly there!" the Queen repeated.
"Why, we passed it ten minutes ago! Faster!"

In example programs, we have used *If* statements to make a choice about whether to execute a section of program code. In this way, *If* statements can be used to control program execution. This is why an *If* statement may be thought of as a control statement. In this chapter, we look at control statements where instructions and methods are repeated.

In Section 7-1, examples are presented where a *Loop* statement is used to execute a block of instructions again and again. The *Loop* statement can be used when we know exactly how many times a block of instructions should be repeated (that is, a definite number of times). We use the word count to describe the number of times a loop repeats. For this reason, a *Loop* statement is referred to as a counted loop.

In Section 7-2, the *While* statement is introduced for repeating a block of instructions where we do not know exactly how many times it should be repeated (an indefinite number of times). A *While* statement corresponds naturally to the way we say and think about certain actions in our everyday conversation. For example, we might say, "While I am driving, I will continue to keep my hands on the wheel and my eyes on the road," or perhaps something like, "While I am a member of the baseball team, I will practice batting." The *While* statement is intuitive—a way to think about actions that repeat while some condition remains true.

7-1 Loops

As our worlds become more sophisticated, the code tends to become longer. One reason is that some animation instructions and methods must be repeated to make the same action occur over and over. Perhaps you have already written programs where you needed to write a call to a method several times. In this section, we look at using a *Loop* statement to call a method repeatedly.

Using a loop for repeating a call to a method

Let's begin with a very simple world, shown in Figure 7-1-1. A bunny (Animals) has snuck into his neighbor's garden (Nature) and has spotted some nice broccoli shoots. We resized several tree (Nature) objects to a very small size to create the broccoli-like objects. The task is to write a program to animate the bunny hopping over to munch a few bites of broccoli.

Let's assume we have already written a method for the bunny named *hop*. The *hop* method enables the bunny to hop forward by moving his legs up and down at the same time as he moves forward. A possible implementation of the *hop* method appears in Figure 7-1-2. (The sound instruction is optional.)

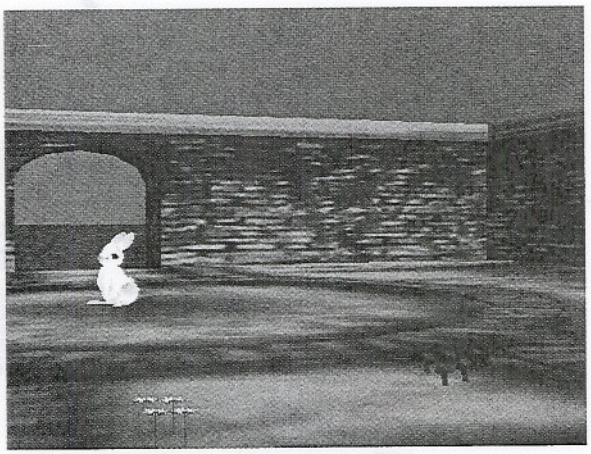


Figure 7-1-1. A bunny in the garden

bunny.hop *No parameters*

No variables

- Do together
  - bunny | play sound | World.jump (0:00.415) | *duration* = 0.5 seconds | more...
  - Do in order
    - // The bunny moves
    - Do together
      - bunny | move | up | .5 meters | *duration* = 0.25 seconds | more...
      - bunny | move | forward | .4 meters | *duration* = 0.25 seconds | more...
    - Do together
      - bunny | move | down | 0.5 meters | *duration* = 0.25 seconds | more...
      - bunny | move | forward | 0.4 meters | *duration* = 0.25 seconds | more...
  - Do in order
    - // The bunny's right foot simulates a hopping motion
    - bunny.hipR.footR | turn | forward | 0.12 revolutions | *duration* = 0.25 seconds | m
    - bunny.hipR.footR | turn | backward | 0.12 revolutions | *duration* = 0.25 seconds
  - Do in order
    - // The bunny's left foot simulates a hopping motion
    - bunny.hipL.footL | turn | forward | 0.12 revolutions | *duration* = 0.25 seconds | m
    - bunny.hipL.footL | turn | backward | 0.12 revolutions | *duration* = 0.25 seconds

Figure 7-1-2. The *bunny.hop* method

The bunny in our example world is eight hops away from the broccoli. One possible way to animate eight bunny hops is given in Figure 7-1-3. In this program, a *Do in order* block is placed in the *World.my first method*. Then, a *turn to face* instruction is used to make the bunny turn to look at the broccoli. Finally, the call to the *bunny.hop* method is dragged into the editor eight times.

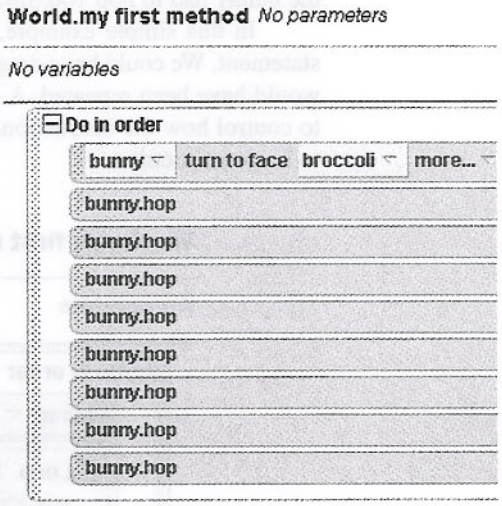


Figure 7-1-3. Eight hops

Of course, the code in Figure 7-1-3 will perform the task of making the bunny hop over to the broccoli. It was tedious, though, to drag eight *bunny.hop* instructions into the program. We want to look at a way to make our job easier by using a *Loop* statement. To create a *Loop* statement, drag the *Loop* tile into the editor. The popup menu offers a choice for the count (the number of times the loop will execute), as shown in Figure 7-1-4. In this example, the number 8 is entered as the count. Note that a loop can execute only a whole number of times.

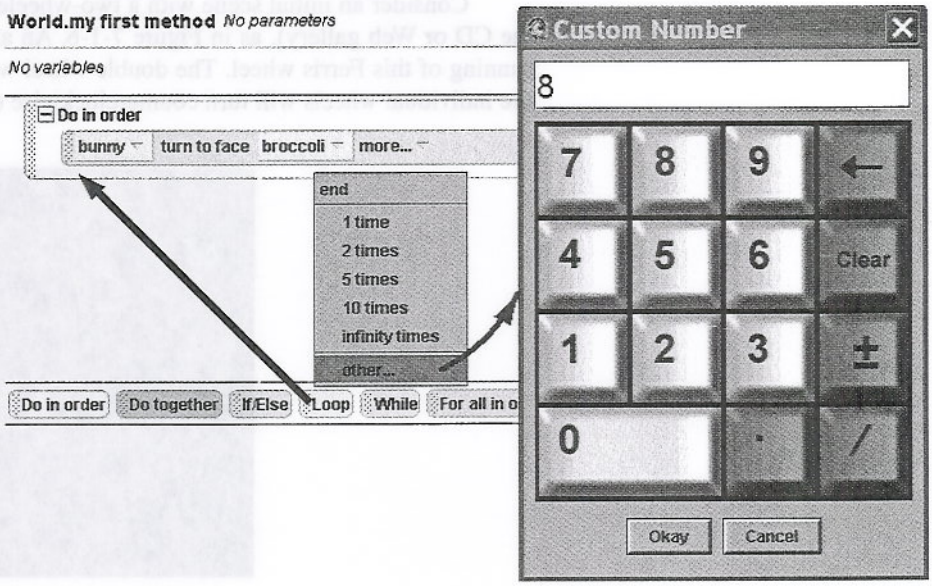


Figure 7-1-4. Selecting a count

Next, a call to the *bunny.hop* method is placed inside the *Loop* statement, as shown in Figure 7-1-5. When the program is run, the bunny will turn to face the broccoli and then hop eight times. That is, the loop will call the *bunny.hop* method eight times. The benefit of using a *Loop* is immediately obvious—the *Loop* is quick and easy to write. Although eight calls to the *bunny.hop* method is not a big deal, there are program situations where methods have to be repeated 20, 30, or even 100 times. It would be much easier to use a single *Loop* statement if the bunny had to hop 100 times!

In this simple example, we placed a single call to a method (*bunny.hop*) in the *Loop* statement. We could have dragged several instructions and method calls into the block, and all would have been repeated. A *Do in order* or *Do together* block can be nested inside the loop to control how the instructions are repeated; otherwise Alice will assume the instructions are to be done in order.

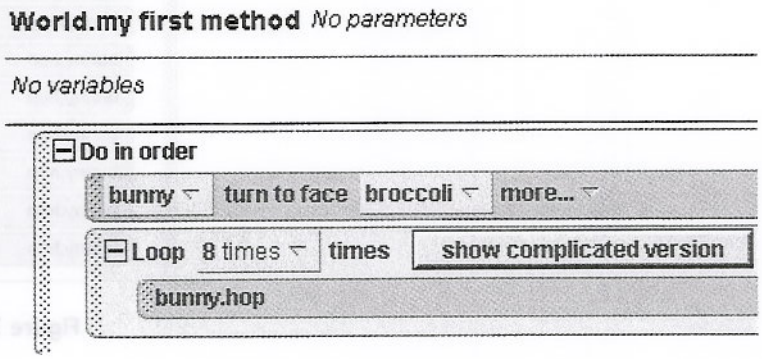


Figure 7-1-5. Using the *Loop* statement to call a method repeatedly

Nested loops

Each of the examples above used only one *Loop* statement. Of course it is possible to have several *Loop* statements in the same program. In fact, a *Loop* statement can be nested inside another *Loop* statement. In this section, we take a look at a world where this is an appropriate way to write the program code.

Consider an initial scene with a two-wheeled Ferris wheel (Amusement Park folder on the CD or Web gallery), as in Figure 7-1-6. An animation is to be written that simulates the running of this Ferris wheel. The double wheel will turn clockwise (roll right) while each of the individual wheels will turn counterclockwise (roll left).

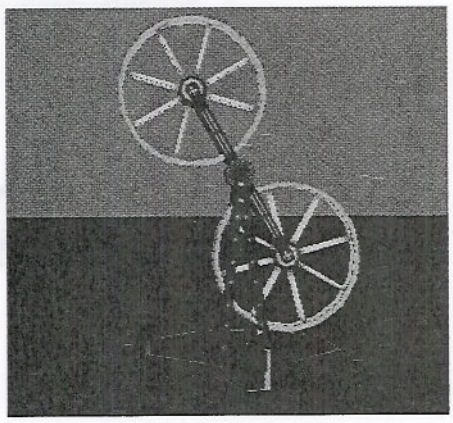


Figure 7-1-6. A Ferris wheel

Figure 7-1-7 is code for a *Loop* to roll the Ferris wheel ten times. The *style = abruptly* parameter has been used to smooth out the rotating of the entire double wheel over each repetition. (We chose the *abrupt* style because the default style, *gently*, slows down the animation instruction as it begins and ends. Because we are repeating the instruction within a loop, we want to avoid having the wheel go through a repeated slowdown, speedup, slowdown, speedup ... kind of action.)

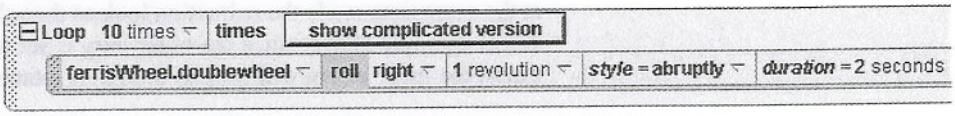


Figure 7-1-7. Rotating the double wheel 10 times

With the motion of the entire Ferris wheel accomplished, instructions can now be written to rotate each of the inner wheels counterclockwise (roll left) while the outer double wheel is rotating clockwise (roll right). The code to rotate the inner wheels is presented in Figure 7-1-8.

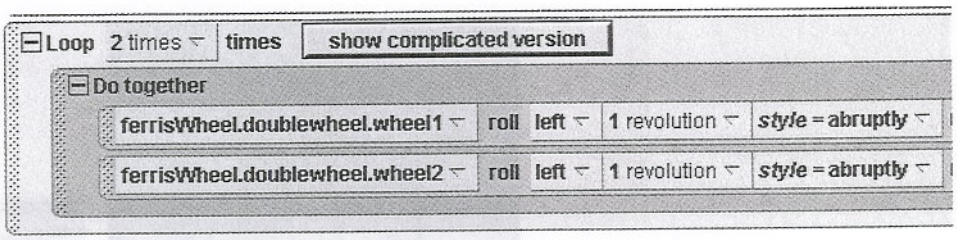


Figure 7-1-8. Rotating the inner wheels two times

Now the rotations can be combined. To increase the excitement of the ride, we will have the inner wheels rotate more frequently, perhaps twice as often as the outer wheel. So, each time the double wheel rotates one revolution clockwise, the inner wheels should rotate twice counterclockwise. Figure 7-1-9 shows the modified code.

In previous programs you have seen that actions in a *Do together* block need to be synchronized in duration. In this example the inner wheels need to rotate twice while the outer wheel rotates once. So, the duration of rotation for the outer wheel (*doublewheel*) is made to be twice the duration of rotation of the inner wheels (*doublewheel.wheel1* and *doublewheel.wheel2*).

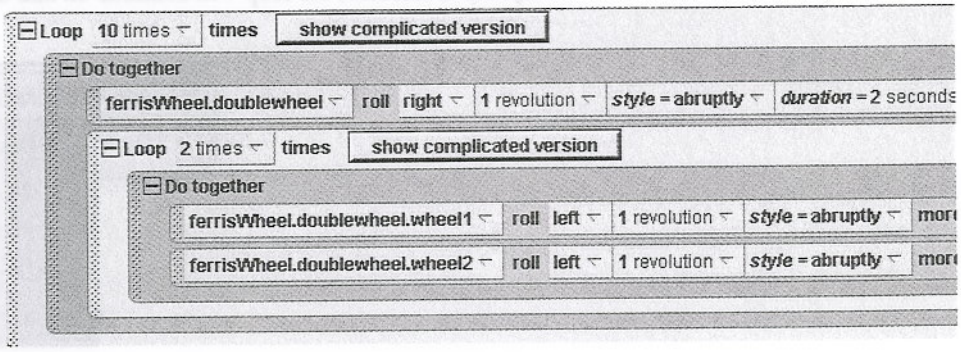


Figure 7-1-9. The complete code for the Ferris wheel

Each time the outer loop runs once, the inner loop runs twice. This means that the double wheel takes 2 seconds to complete its rotation, and the inner wheels require 1 second (the default duration) to complete their rotations, but rotate twice. In all, the inner wheels rotate 20 times counterclockwise as the outer wheel rotates 10 times clockwise.

Optical Illusion: Instructions within a *Do together* block can sometimes cancel each other out (in terms of the animated action we can observe). For example, in the First Encounter world described in Chapter 2, Section 2, turning the backLeftLegUpperJoint backward and forward at the same time made the animation look as though the leg was not turning at all. In the Ferris wheel example above, since the outer loop is set to execute only 1 time and the inner loop 2 times, the inner wheel will look as though it rotates only once.

Using a function call to compute the count

In the examples presented here, the number of times a block of code was to loop was a specific number, 2, 8, or 10. But the count can also be a function that returns a number. For example, the code in Figure 7-1-10 uses a function to determine the count for a loop. If the guy were 6 meters from the girl, the *Loop* would make the guy walk 6 times (assuming that the guy covers exactly 1 meter in his *walk* method). What would happen if the guy were 5.7 meters from the girl? The loop only executes a whole number of times. So the guy would walk only 5 times.

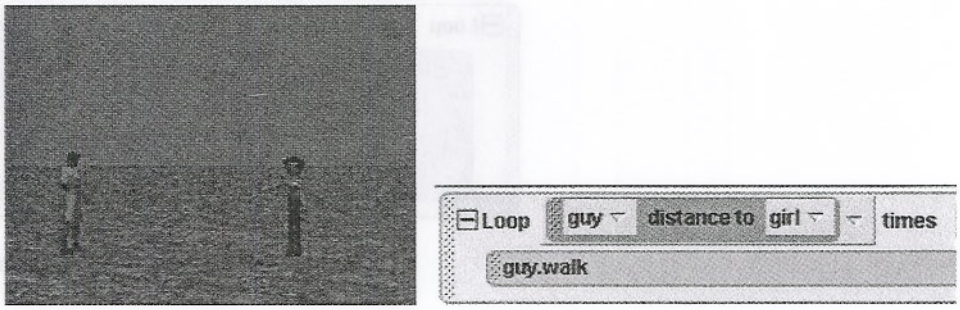


Figure 7-1-10. Using a function to determine a loop count

A tremendous advantage of using a function as the number of times to loop is that objects can be repositioned in the initial scene, and it will not be necessary to modify the code to specify the count. The function will automatically compute the distance between the guy and the girl and the loop will repeat only as many times as needed.

Infinite loop

One option in the popup menu for selecting a *Loop* count is *infinity*. (See Figure 7-1-11.) If *infinity* is selected, the loop will continue on and on until the program stops.

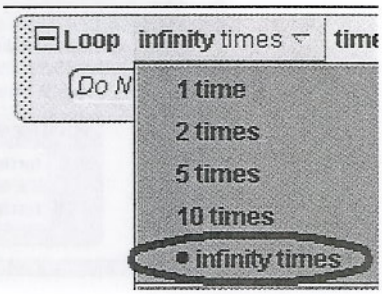


Figure 7-1-11. Selecting *infinity* as the count

Ordinarily, we advise that you avoid infinite loops. However, in animation programs an infinite loop is sometimes useful. Consider an amusement park animation where a carousel is one of the objects in the scene, as in Figure 7-1-12. To make the carousel go around in the background, call the carousel's built-in method *carouselAnimation* in an infinite loop, as shown in Figure 7-1-13.

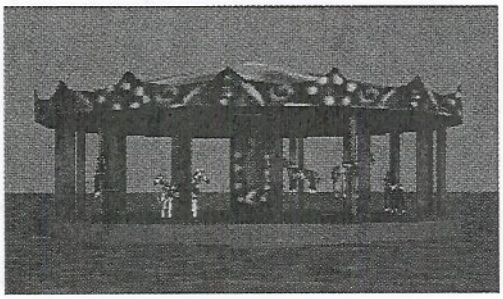


Figure 7-1-12. Carousel in infinite motion

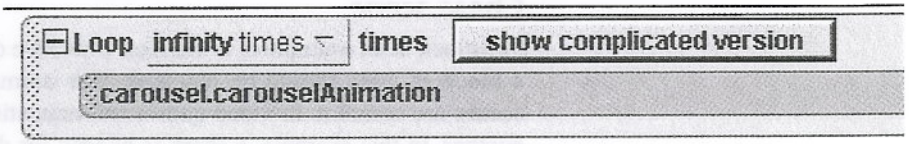


Figure 7-1-13. Loop *infinity* times

7-2 *While*—a conditional loop

The *Loop* statement requires that the programmer specify the number of times the loop is to be repeated. This could be a number, such as 10, or a function such as *boy.distance to girl*. The number of times a loop should repeat might not be known in advance. This is often the case in games and simulations, where objects move randomly. In this section, a form of looping known as the *While* statement will be introduced to handle situations where the programmer does not know (at the time the program is written) how many times the loop should be repeated.

The *While* statement is a conditional loop . One way to think about a *While* statement is:

While some condition is true perform instruction(s)

The instruction performed inside the while statement can be a single action or several actions enclosed in a *Do in order* or *Do together* block. The condition used in a *While* statement is a Boolean condition (the same type of condition used in *If* statements). The Boolean condition acts like a gatekeeper at a popular dance club. If the condition is true, entry is gained to the *While* statement and the instructions within the *While* statement are executed, otherwise the instructions are skipped. Unlike the *If* statement, however, the *While* statement is a loop. The diagram in Figure 7-2-1 illustrates how the *While* statement works as a loop.

If the condition is *true,* the instructions are performed; then the condition gets checked again. If the condition is still *true*, the instructions within the loop are repeated. If the condition has become *false*, the loop ends and Alice goes on to execute the next statement in the program. A *While* statement is useful for situations where we do not know how many times the loop should be repeated. All we need to know is the condition that determines whether the loop will be repeated.

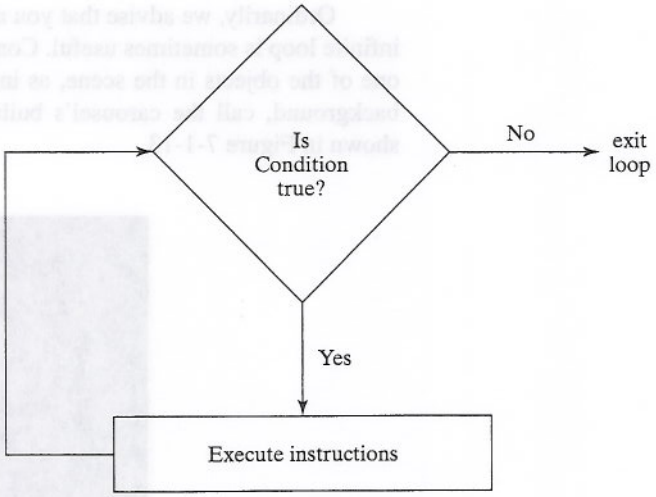


Figure 7-2-1. The *While* statement works as a loop

Chase scene

Let's look at an example of a situation where we do not know (ahead of time) how many times a block of code should be repeated. This animation is a "chase scene" simulation. Chase scenes are common in video games and animation films where one object is trying to catch another. In this example, a shark is hungry for dinner. The shark is going to chase after and catch a fleeing goldfish (Animals). OK, so this is not a gracious animation—but sharks have to eat, too! Figure 7-2-2 shows a very simple initial world.

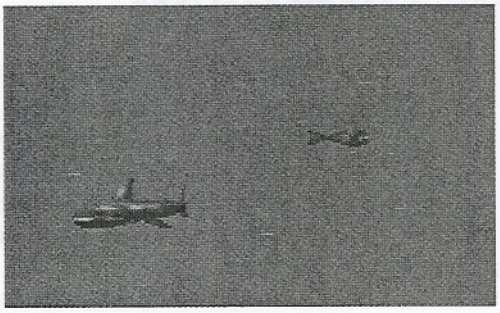


Figure 7-2-2. Chase scene

Problem

Our task is to animate the shark chasing the goldfish, as it tries to get close enough (say, within 0.5 meters) to gobble the goldfish down for dinner. Naturally, as the shark chases it, the goldfish is not standing still. Instead, it is trying to escape by moving to a random position. Of course, we want the fish to look like it is swimming (not jumping all around on the screen), so we will move it to a random position that is close to the current position.

Storyboard solution

The basic idea is that if the goldfish is more than 0.5 meters away from the shark, the shark is going to point at the goldfish and swim toward it. Meanwhile, the goldfish is moving away to a random position nearby. If the goldfish is still more than 0.5 meters away, the shark will change course and swim toward it and the goldfish will try to swim away. Eventually, when

the shark finally gets within 0.5 meter of the goldfish, the chase is over and the shark can catch and eat the goldfish.

Let's plan a method for the chase animation using the *While* statement. Think about it like this:

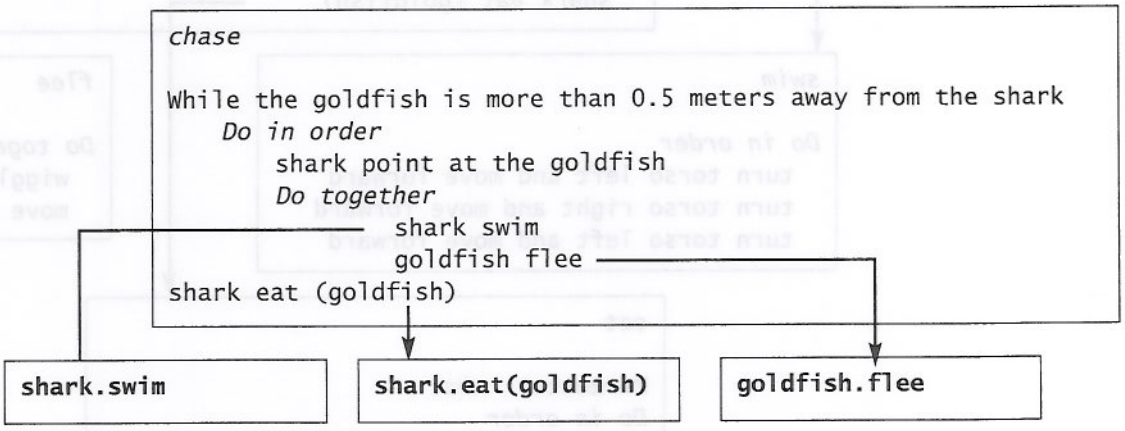
"While the shark is more than 0.5 meters away from the goldfish, move the shark toward the goldfish and, at the same time, move the goldfish to a random position nearby"

The condition in our *While* statement will be "the goldfish is more than 0.5 meter away from the shark." If this condition is true, the chase is on. A storyboard is shown here.

```
chase

Do in order
While the goldfish is more than 0.5 meters away from the shark
        Do in order
                shark point at the goldfish
                Do together
                        shark swim (toward the goldfish)
                        goldfish flee (away from the shark)
shark eat (the goldfish)
```

The condition must be *true* to allow entry into the loop. After running the instructions within the *While* statement, the condition will be evaluated again, and if the condition is still *true*, the instructions within the *While* statement will run again. This process continues until the condition finally evaluates to *false*.

In the storyboard, the *shark swim, goldfish flee*, and *shark eat* steps will each require several instructions. These actions will be written as methods.



For now, let's pretend that these methods have already been written (we can write them next). A *chase* method to implement the overall storyboard is illustrated in Figure 7-2-3.

Now, let's look at how to write the *shark.swim, goldfish.flee,* and *shark.eat* methods that are called from the *chase* method. We can use stepwise refinement to complete the storyboard, as shown next. Please note that a *duration* of zero causes the instruction to occur instantaneously, without a gradual animation of the action.

The *shark.swim* method moves the shark forward and the turns the torso of the shark right and left to simulate a swimming motion through water. Figure 7-2-4 illustrates *shark.swim*.

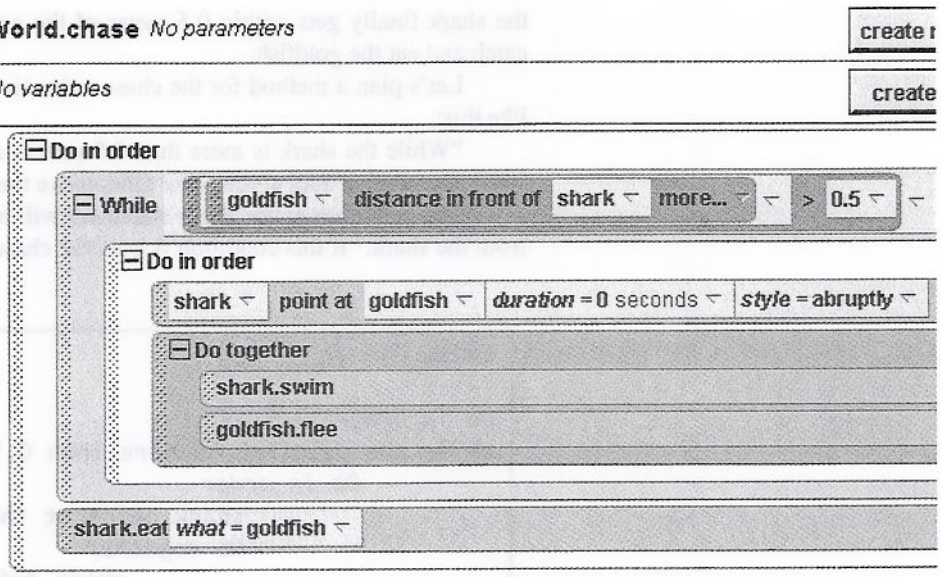


Figure 7-2-3. The *chase* method using a *While* statement

```
chase

Do in order
  While the goldfish is more than 0.5 meters from the shark
    Do in order
        Point the shark at the goldfish
        Do together
            shark swim
            goldfish flee
  shark eat (goldfish)
```

```
swim

Do in order
  turn torso left and move forward
  turn torso right and move forward
  turn torso left and move forward
```

```
flee

Do together
  wiggle tail
  move to random location
```

```
eat

Parameter: what
Do in order
  shark points at what
  shark opens jaw and moves forward
  shark closes jaw
```

The *goldfish.flee* method moves the tail of the goldfish to simulate a swim-like motion and calls the *randomMotion* method (previously defined in Tips & Techniques 6) to move the fish to a nearby random location. The *goldfish.flee* method is shown in Figure 7-2-5.

After the While statement ends, the *shark.eat* method is executed, as is illustrated in Figure 7-2-6. The method uses a parameter to specify *what* object in the scene is on the menu.

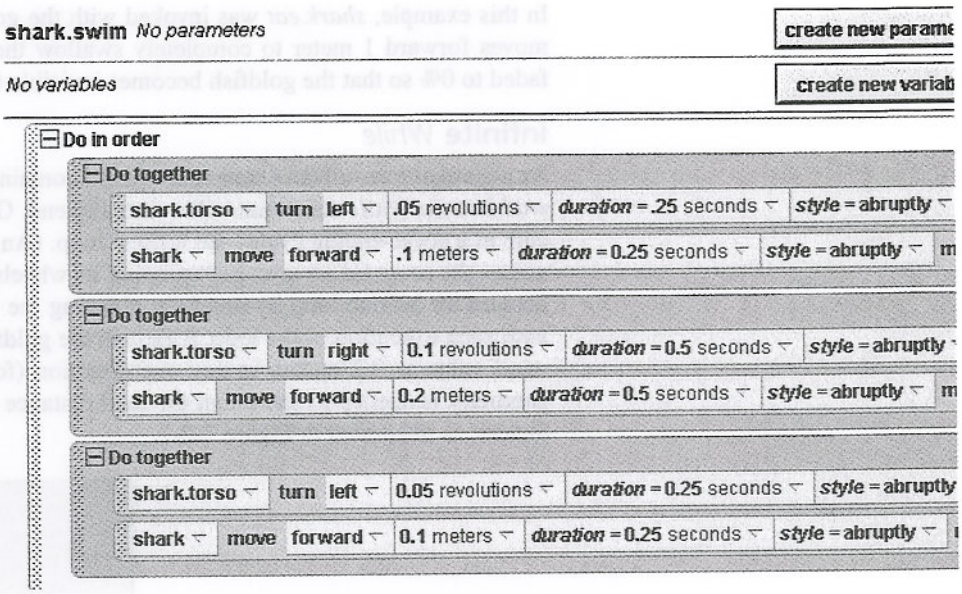


Figure 7-2-4. The *shark.swim* method gives realistic shark movement

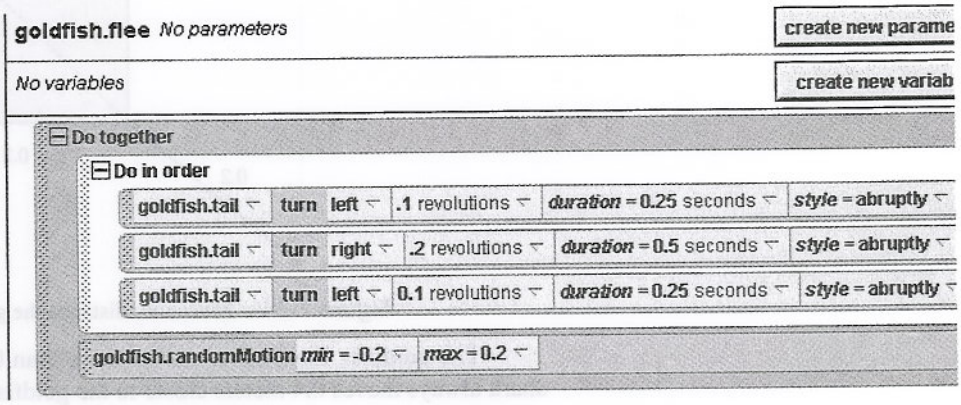


Figure 7-2-5. The *goldfish.flee* method—fish swims to a random location

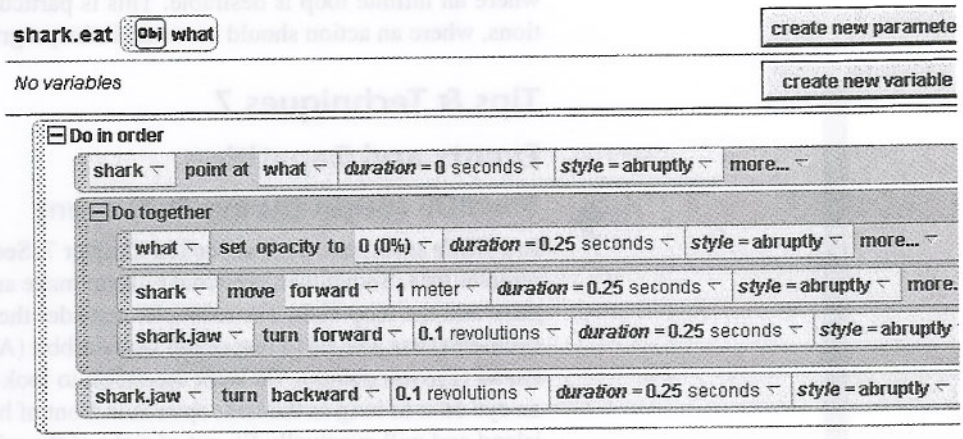


Figure 7-2-6. The *shark.eat* method

In this example, *shark.eat* was invoked with the goldfish object as the parameter. The shark moves forward 1 meter to completely swallow the goldfish. The opacity of the goldfish is faded to 0% so that the goldfish becomes invisible (disappearing into the shark's mouth).

Infinite *While*

As a general rule, a *While* statement should contain instructions that change conditions in the world so the *While* statement will eventually end. Otherwise, the loop would continue to execute in a never-ending cycle—an infinite loop. (An infinite loop is often a program error that makes the program seem to be "spinning its wheels.") In the shark chase example above, we avoided an infinite loop by carefully planning the distance the shark and the goldfish move with each execution of the loop. Although the goldfish is moving randomly, we set the maximum value to 0.2 meters in any one direction (forward, up/down, right/left). A bit of 3D geometry is needed to show that the total distance the goldfish travels must be less than .35 meters, as illustrated in Figure 7-2-7.

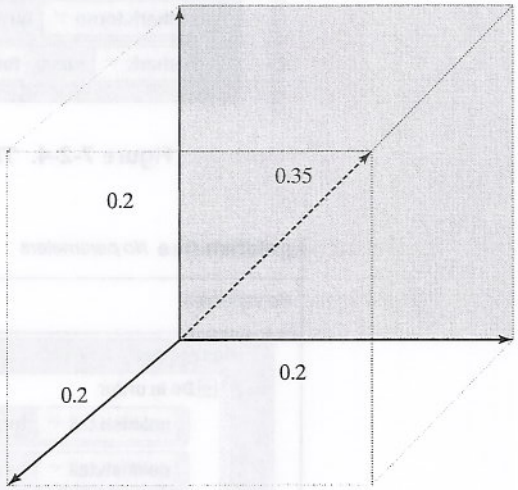


Figure 7-2-7. Maximum distance the goldfish can travel in one move

The goldfish, then, never moves more than 0.35 meters away from the shark, and the shark always moves 0.4 meters closer to the goldfish. The shark's distance advantage guarantees that the shark is eventually going to catch up to the goldfish and the loop will end.

We highly recommend that you carefully check the condition for a *While* statement to be sure the loop will eventually end. On the other hand, there are some kinds of programs where an infinite loop is desirable. This is particularly true in games and simulation animations, where an action should occur until the program shuts down.

Tips & Techniques 7

Events and Repetition

The BDE (Begin-During-End) event

In a *While* statement, as described in Chapter 7, Section 2, actions occur while some condition remains true. Sometimes, we would like to make an action occur when the condition becomes *false* and the loop ends. For example, consider the world in Figure T-7-1, where a helicopter (Vehicles) has arrived to rescue the white rabbit (Animals). The helicopter is circling over the island (Environments). We want the rabbit to look at the helicopter and turn his head to keep an eye on it as long as the helicopter is in front of him. Of course, the helicopter is circling the island and will eventually fly out of sight of the rabbit. When the rabbit can no longer see the helicopter, we want him to look at the camera.

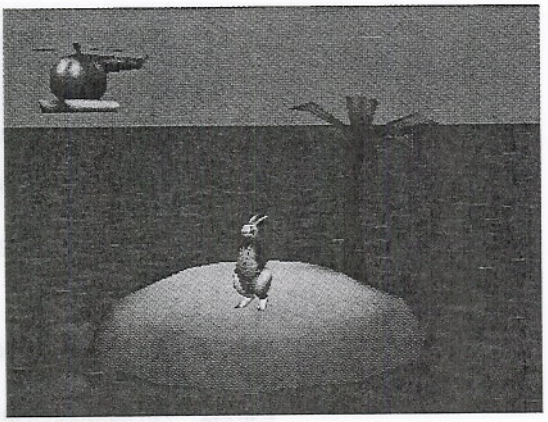


Figure T-7-1. The rabbit rescue scene

This animation has two repeated actions: a helicopter circling an island and a rabbit moving his head to watch the helicopter as it circles. Also, an action should occur as soon as the helicopter can no longer be seen by the rabbit: the rabbit looks back at the camera. The problem is how to make actions repeat (within a loop) but also make an action occur each time the loop ends. We have seen this situation before in working with interactive worlds and events. Perhaps we can create a solution to our problem by using events. In fact, Alice has a built-in *While* statement event for linking events to repeated actions. The event, highlighted in Figure T-7-2, is *While something is true*.

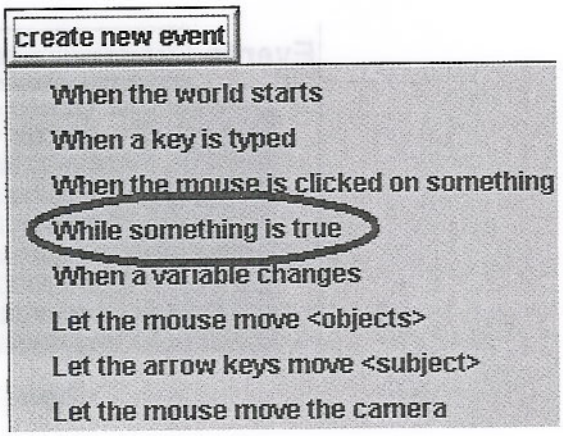


Figure T-7-2. Selecting *While something is true* in the events editor

Selecting the *While something is true* event from the pull-down menu causes a *While* event block to be added to the Events editor, as shown in Figure T-7-3. The event block automatically contains a Begin-During-End (BDE) block that allows you to specify what happens when the loop begins, during the execution of the loop, and when the loop ends.

To create a BDE behavior, we need to specify up to four pieces of information (any or all may be left as *Nothing*):

A conditional expression/function (evaluates to *true* or *false*)—The *While* statement condition is <*None*> by default, but you must replace it with a Boolean expression or function.

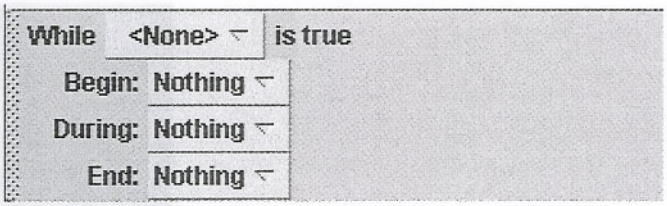


Figure T-7-3. *While* event with *Begin-During-End* (BDE)

Begin—An animation instruction or method that is to be done once, at the time the condition becomes true.

During—An animation instruction or method that is to be done repeatedly, as long as the condition remains true. Note that as soon as the condition becomes false, the animation instruction or method ceases running, even if it is in the middle of an instruction!

End—An animation instruction or method that is to be done once, at the time the condition becomes false.

In the above example, the helicopter is flying around the island. We would like to have the white rabbit's head look at (*point at*) the helicopter as soon as the helicopter is in front of the rabbit (Begin). Then, as long as the helicopter is still in front of the rabbit (During), we want the rabbit's head to continue to look at the helicopter. As soon as the helicopter is no longer in front of the rabbit (End), the rabbit should look back at the camera. The code in Figure T-7-4 accomplishes this task:

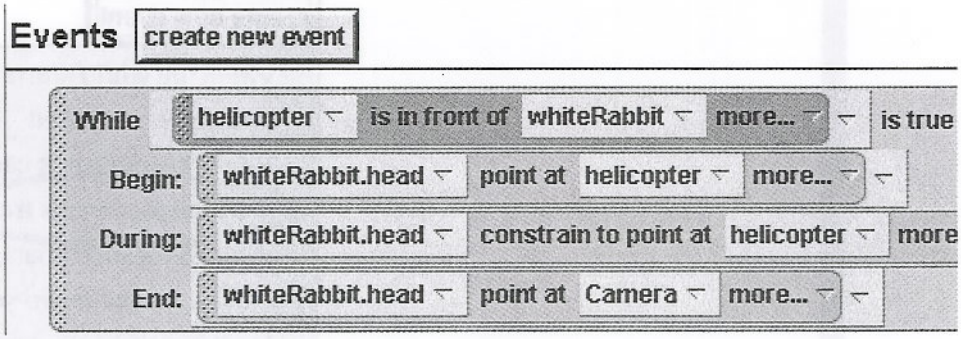


Figure T-7-4. *While statement in an event link*

The *constrain to point at* instruction is a built-in method that adjusts the position of an object to point at some other object. In this example, *constrain to point at* is used to make the rabbit's head continuously point at the helicopter as long as the *While* statement condition (helicopter is in front of whiteRabbit) is *true*.

Exercises and Projects

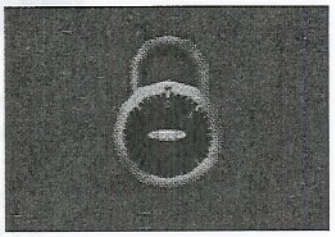
7-1 Exercises

1. *Caught in the Act*

   This exercise is to complete the bunny example in Section 7-1. You will recall that the bunny has snuck into his neighbor's garden and is hopping over to take a bite of the tempting broccoli shoots. Code was presented to make the bunny hop eight times (in a

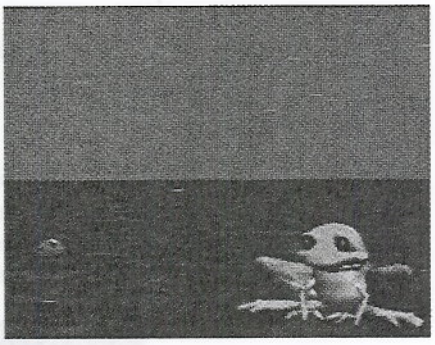
4. *Lock Combination*
Exercise 10 in Chapter 4 was to create a new class of combination lock with four class-level methods—*leftOne, rightOne, leftRevolution*, and *rightRevolution*—that turn the dial 1 number left, 1 number right, 1 revolution left, and 1 revolution right, respectively. Also, the lock has a method named *open* that opens it, and another named *close* that closes it. The purpose of this exercise is to reuse the methods created in the previous exercise. Revise the previous world. Use a loop instruction to turn the dial left 25. Then use a loop to turn the dial right and finally use a loop to turn the dial back to the left 3 times. (The combination is 25, 16, 3). Then, pop open the latch, close the latch and return the dial to zero.

Hint: Use *Wait* to make the lock pause between each turn of the dial.

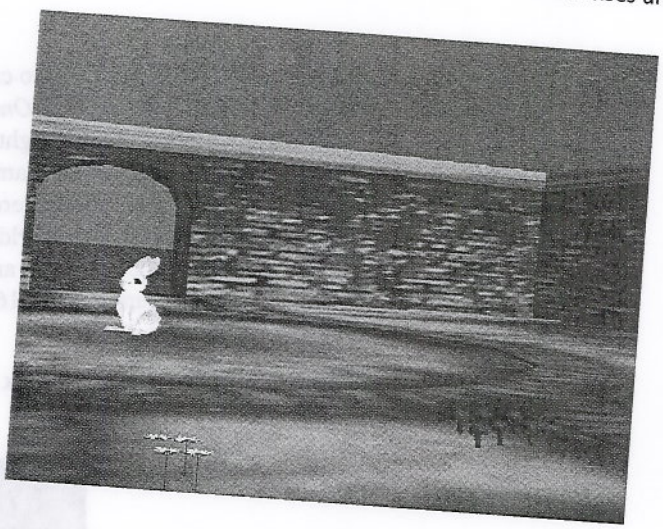


7-2 Exercises

5. *Frog and Ladybug*
Create a world with a frog (Animals) and a ladybug (Animals). Write an interactive program to allow the user to drag the ladybug around the scene. (Use a *let the mouse move objects* event.) As the ladybug is dragged around, make the frog chase after it by moving one hop at a time without colliding with the ladybug. If the user moves the ladybug within 2 meters of the frog, have the frog look at the camera and say "ribbit"—then end the animation.



6. *Bumper Cars*
Create a simulation of the bumper car ride (Amusement Park), where the cars move continuously around within the bumper arena. Add two bumper cars inside the arena. In this animation, each car should be moving forward a small amount until it gets too close to another car or to the wall, then turn the car a quarter of a revolution clockwise (to get a different direction) and continue moving forward. Use a switch (Controls) to stop and start the ride. As long as the switch is on, the ride should continue.

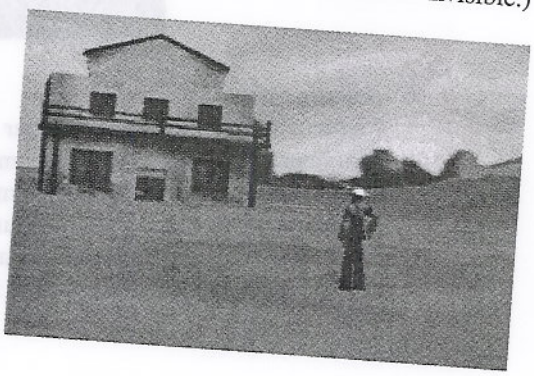


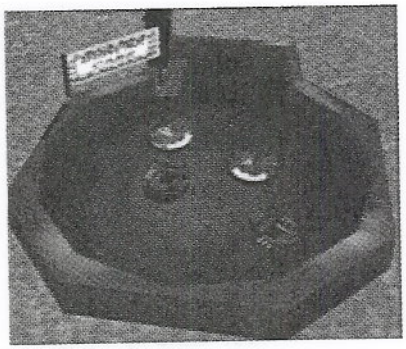
loop) over to the broccoli. Just as the bunny reaches the broccoli, Papa rabbit (Hare from Animals folder) appears at the garden gateway and taps his foot in dismay. The bunny hops a quick retreat out of the garden. Write a program to implement the bunny in the garden animation. Your code should use a loop not only to make the bunny hop over to the broccoli (see the initial scene in Figure 7-1-1) but also to hop out of the garden.

2. *Square Bunny Hop*
   This exercise explores the use of nested loops. Papa rabbit has been teaching the bunny some basic geometry. The lesson this morning is on the square shape. To demonstrate that the bunny understands the idea, the bunny is to hop in a square. Create a world with the bunny and the hop method, as described in Section 7-1. Use a loop to make the bunny hop three times. When the loop ends, have the bunny turn left one-quarter revolution. Then add another loop to repeat the above actions, as shown in the storyboard here.

```
Loop 4 times
        Loop 3 times
                bunny.hop
        turn left 1/4 revolution
```

3. *Saloon Sign*
   An old saloon (Old West) is being converted into a tourist attraction. Use 3D text to create a neon sign to hang on the front of the balcony. Then use a loop to make the sign blink 10 times. (Tips & Techniques 1 provides instructions on using 3D text. Tips & Techniques 4 shows how to make an object visible or invisible.)

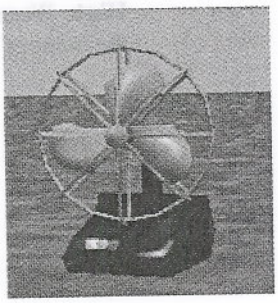




Hint: To avoid a car driving through a wall of the arena, a simple form of collision detection is needed. One way to check for a possible collision is to use the *distance to* function to compute the distance of the car to the arena. Remember that *distance to* is measured "center-to-center." In this world, a measurement from the center of the car to the center of the arena is exactly what you need. (When a car gets too far from the center of the arena, it will collide with a wall.) It is also possible to write a function that returns whether two cars are about to collide with one another. What should be done in this case?

7. *Stop and Start*
   Choose a ride object other than the carousel from the Amusement Park gallery that moves in a circular pattern (a round-and-round manner, like the Ferris wheel). Create a method that performs an animation appropriate for the ride object selected. Then, create a way to start and stop the ride using the *While something is true* event.

8. *Speed Control*
   Place a fan (Objects) in a new world. The fan has four buttons—high, medium, low, and off. Create a method that controls the speed at which the fan blades rotate, depending on which button is clicked. (You may want to use several methods instead of just one.) The blades should continue turning until the animation stops running or the user clicks the off button.



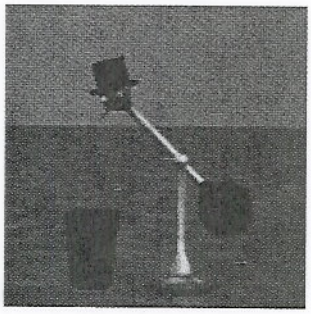
9. *Old-time Rock and Roll*
   Create a world with an old-fashioned phonograph (Objects) in it. Create methods to turn the crank and turn the record. Then, create the BDE control mechanism (Tips & Techniques 7) to call a method that plays the record at the same time the crank is turning.

10. *Wind-up Penguin*
Create a world with a wind-up penguin. This is actually a penguin (Animals) with a windUpKey (Objects) positioned against its back. The key's *vehicle* property has been set to the penguin. In this world, make the penguin waddle (or walk) around the world continuously while its wind-up key turns.

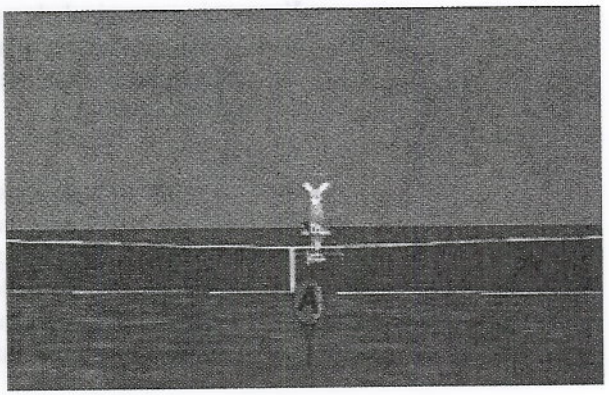
7-2 Projects

1. *Drinking Parrot*
A small toy popular with children is a drinking parrot. The parrot is positioned in front of a container of water and its body given a push. Because of the counterbalance of weights on either end of its body, the parrot repeatedly lowers its head into the water. Create a simulation of the drinking parrot (Objects). Use an infinite *Loop* statement to make the parrot drink.

Hint: We used the blender object (Objects), pushed the base into the ground, and changed the color of the blender to blue to simulate a bucket of water.



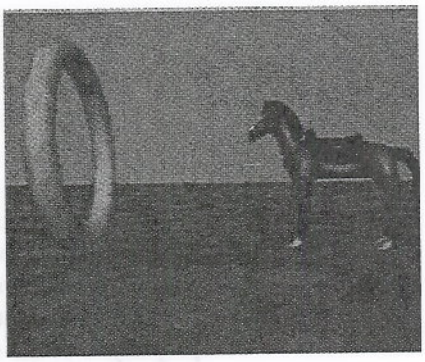
2. *Tennis Shot*
Create a tennis shot game with a kangaroo_robot (SciFi), a tennis ball, tennis net, and tennis racket (Sports). Position the tennis ball immediately in front of the robot on the other side of the net (from the camera point of view) and the tennis racket on this side of the net (from the camera point of view). The initial scene should appear as shown next.



Set up an event to *let the mouse move objects* (Tips & Techniques 5). In this game, the kangaroo_robot and the tennis ball move together left or right a random distance (use the World *random number* function) between −1 and 1 meter. Then the robot "throws" the ball across the net (the tennis ball moves up a random height and forward a given distance). The player will move the tennis racket to try to "hit" the tennis ball. A "hit" occurs when the tennis racket gets within 0.1 meter of the tennis ball. Actually, the tennis ball is virtual in this simulation and will go right through the racket even if the player manages to "hit" it. However, we will know when the player manages to "hit" the tennis ball, because the kangaroo_robot will wiggle his ears.

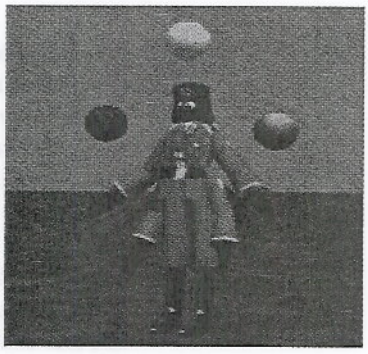
The real challenge in writing this program is to figure out whether the player is successful in moving the tennis racket close enough to the tennis ball to hit it before it goes out of sight. To make this work, use a loop where each execution of the loop moves the tennis ball up and forward only a very short distance. Each time through the loop, check to see if the racket has gotten close enough to the ball to score a hit. As mentioned above, you will need to experiment to figure out the appropriate count for the loop so as to eventually move the ball forward out of sight of the camera. When the loop ends, have the kangaroo_robot turn left one-quarter revolution and move off the tennis court signaling that the game is over.

3. *Horse through a Hoop*
The horse is in training for a circus act, in which the horse must jump through a large hoop (we used the torus from the Shapes folder). Create a world, as shown below, with a horse facing the hoop. Write a program to use a loop to have the horse trot forward several times and then jump through the hoop. When the horse gets close enough to the hoop to jump through it, have the horse jump through and then the world ends.

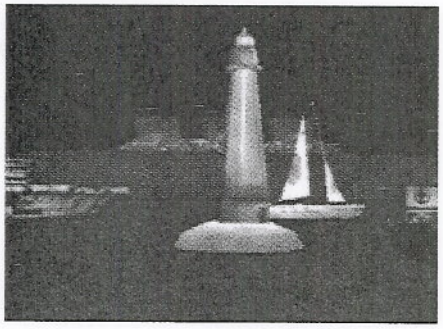


Your project must include a *trot* method that makes the horse perform a trot motion forward. A horse trots by turning the legs at the same time as the body moves forward. The leg motions can bend at the knee for a more realistic simulation, if desired. The loop calls the *trot* method.

4. *Juggling*
   Create a world with an object that has arms (the picture below shows an ant) and three juggling balls. (A juggling ball is easily created by adding a sphere or a tennis ball to the world, resizing it, and giving it a bright color.) To juggle the balls, the object must have at least two arms and be able to move them in some way that resembles a tossing motion. Write a method to animate juggling the balls in the air. Use a loop to make the juggling act repeat five times, after which the juggling balls fall to the ground.

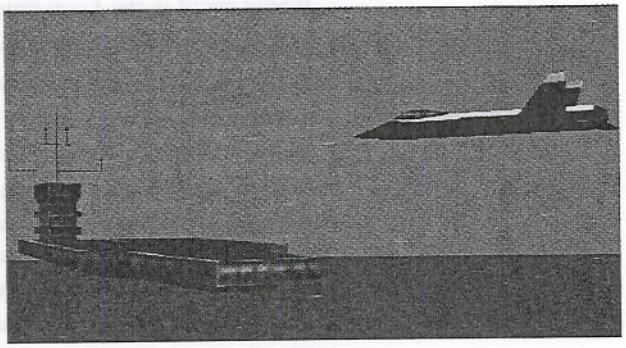


5. *Lighthouse Warning*
   Set up a world that contains a lighthouse (Beach), a spotlight (Lights), a lightbulb (Lights), a stars skydome (Environments/Skies), and a number of boats (Vehicles), as seen in the image below. The lighthouse sits in the harbor to warn ships of shallow water. In this simulation, the light of the lighthouse is to rotate, as in real life. You should be able to see the sides of the boats illuminated as the light of the lighthouse swings around. Use repetitive world methods to continuously move each boat back and forth and rotate the spotlight. In *my first method*, make the world dark by setting the brightness property of the world's light to be 0. Then invoke the methods to create the lighting effect. Set the duration of the rotation to be 3 seconds or so.

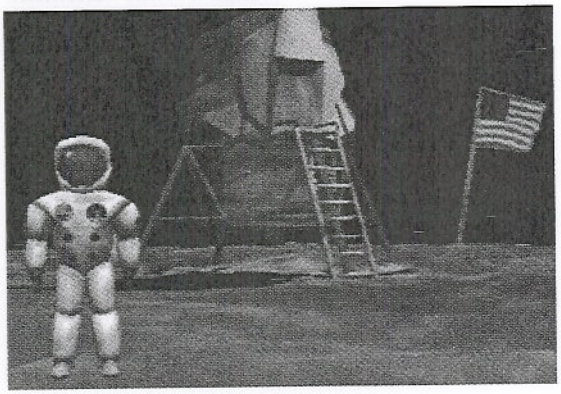


Hint: To create a dramatic lighting effect in this animation, place a light bulb inside a spotlight and put both in the windowed area at the top of the lighthouse. Make both the lightbulb and the spotlight invisible (they will still give off light). The lightbulb and spotlight should rotate together to create a bright beam of light (set the lightbulb's vehicle property to be the spotlight).

6. *Carrier Landing*
   Create a water world with a carrier in the ocean and a navy jet (Vehicles) in the air, as shown below. Write an interactive program to allow the user to use keyboard controls to land the jet on the deck of the carrier. Up, down, and forward controls are needed. Use a function to determine when the jet has landed (gotten close enough to the carrier's deck) and a *While* statement to continue the animation until the jet has landed.



7. *Moonwalk*
   Write a program that will make the astronaut (Space) perform a moonwalk. The lunar lander (SciFi) and flag (Objects) are used to decorate the Space template scene.



   To perform the moonwalk, the astronaut should turn right and then walk backward in a sliding sort of motion where one leg slides backward and then the other leg slides backward. The astronaut's entire body must move backward at the same time as the moonwalk leg motions are executed. Use a loop to make the astronaut repeat the moonwalk steps five times.

Summary

Loop and *While* were introduced in this chapter as control structures for repeating an instruction or block of instructions. A counted *Loop* allows you to specify exactly how many times a block of code will be repeated, and a *While* statement allows you to repeat a block of code as long as some condition remains true. The advantage of using loops is immediately obvious. Loops are fast and easy to write and also easy to understand. Of course, it is possible to write loop statements that are complicated. Overall, though, loops are impressive programming tools.

Important concepts in this chapter

- The counted *Loop* statement can be used to repeat an instruction, or a block of several instructions.
- A key component in a counted *Loop* is the count—the number of times the instructions within the loop will be repeated.
- The count must be a positive whole number or infinity. A negative value for the count would result in the *Loop* statement not executing at all!
- If the count is infinity, the loop will repeat until the program shuts down.
- A *Do in order* or *Do together* can be nested inside a loop; otherwise Alice assumes that the instructions are to be executed in order.
- Loops can be nested within loops.
- When a loop is nested within a loop, the inner loop will fully execute each time the outer loop executes once. For example, if the outer loop count is 5 and the inner loop count is 10, then the inner loop executes 10 times for each of the 5 executions of the outer loop. In other words, the outer loop executes 5 times and the inner loop 50 times.
- The *While* statement is a conditional loop.
- The Boolean condition used for entry into a *While* statement is the same as used in *If* statements. The difference is that the *While* statement is a loop, and the condition is checked again after the instructions within the *While* statement have been executed. If the condition is still true, the loop repeats.
- *If* statements can be combined with *While* statements. In some programs, a *While* statement is nested within an *If* statement. In other situations, an *If* statement may be nested within a *While* statement.

Chapter 8
Repetition: Recursion

. . . the kitten had been having a grand game of romps with the ball of worsted Alice had been trying to wind up, and had been rolling it up and down till it had all come undone again; and there it was spread over the hearth-rug, all knots and tangles, with the kitten running after its own tail in the middle.

This chapter introduces a third form of repetition known as recursion, in which a method (or a function) calls itself. This is an extremely powerful technique that greatly enhances the types of problems that can be solved in Alice. Recursion is often used where we do not know (at the time the program is written) the number of times a block of code should be repeated. (This is also true for the *While* statement, covered in Section 7-2.)

In Alice, there are two major situations where we do not know (even at runtime, when the program first starts to run) the count of repetitions. The first is when random motion is involved. As you recall, random motion means that an object is moving in a way that is somewhat unpredictable. In Section 8-1, we will explore the technique of writing methods where repetition is implemented with recursion.

In Section 8-2, we will look at a famous puzzle and present a solution using a second flavor of recursion. This form of recursion is useful when some complex computation is to be done that depends on an ability to break a problem down into smaller versions of the same problem (sub-problems). The solutions to the smaller sub-problems are used to cooperatively solve the larger problem.

8-1 Introduction to recursion

Recursion is a not a program statement with a special word that identifies it as part of the programming language. Instead, recursion is a well-known programming technique where a method calls itself. Recursion can be used to handle situations where the programmer does not know how many times the loop should be repeated. Examples in this section will show you how to use this technique.

A game-like example

To illustrate recursion, we will use a simple version of a horse race in a carnival game (Amusement Park). The initial scene is shown in Figure 8-1-1. Unlike a traditional horse race, where horses run around an oval-shaped track and each horse breaks to the inside of the track, horses in a carnival game move straight ahead in a mechanical track. In a carnival game, you win a prize if you pick the right horse. In this example, we won't worry about picking the right horse. We will simply have the winning horse say, "I won!!!" This is not a realistic end to the game—but it will serve the purpose of signaling the end of the race.

The problem is how to make this simulate a real carnival game, where the horses move forward again and again until one of them finally reaches the finish line. At each move, the

Figure 8-1-1. Initial scene for a three-horse race

horses move forward different amounts. To keep the game honest, each horse must have an equal chance of winning the race. This means that over several runs of the game, each horse should win about the same number of times as the other horses.

A possible solution is to randomly choose one horse to move forward a short distance. If none of the horses has won (gotten to the finish line), we will once again randomly choose one of the horses to move forward. This action (a randomly chosen horse moves forward a short distance) will be repeated again and again until one of the horses eventually gets to the finish line and wins the race.

An essential step is how to decide when the race is over. In this example, if one of the horses reaches the finish line, the horse wins the race and the game is over. An *If* statement can be used to check whether one of the horses has won. If so, the game is over. Otherwise (the *Else* part), we randomly choose another horse to move forward and do it all again. A storyboard for this animation could look something like this:

```
race

If one of the horses has won
        the winner says, "I won!!!"
Else
        randomly choose one horse and move it forward a small amount
        do everything again
```

In the storyboard above, the line "do everything again" means that the entire method should be repeated. How do we write a program instruction to "do everything again"? We simply have the method call itself! Let's modify the storyboard to show the recursive call:

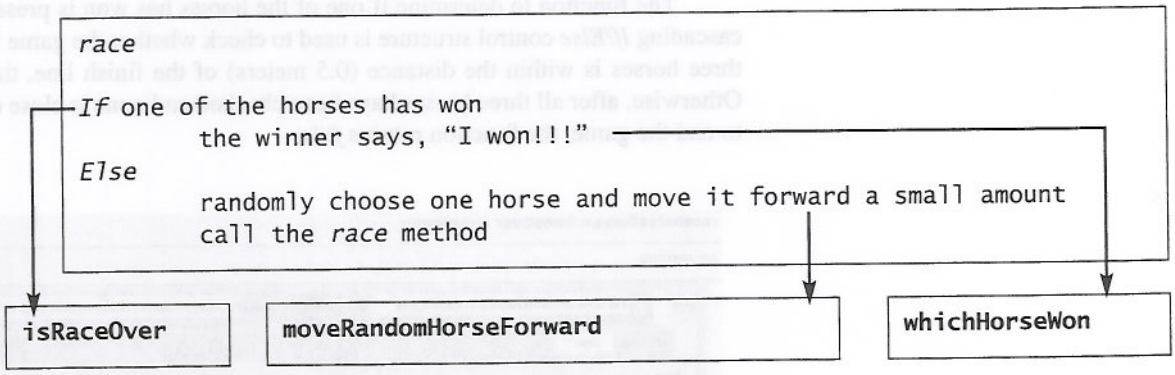
```
race

If one of the horses has won
        the winner says, "I won!!!"
Else
        randomly choose one horse and move it forward a small amount
        call the race method
```

A close look at the modified *race* storyboard shows that the method is calling itself! This is known as recursion. A method that calls itself is said to be recursive. The effect of a recursive call is that the method will repeat. Repeated execution of a method may occur again and again until some condition is reached that ends the method. In this example, the condition that ends the recursive calls is "one of the horses has won."

Now, all we have to do is translate the storyboard into program code. To write the code for the race, we have three problems:

1. How to determine when the race is over.
2. How to randomly select one of the horses to move forward for each execution of the loop.
3. How to figure out which horse won the race and have it say, "I won!!!"

These problems can be solved using stepwise refinement, creating functions and/or methods to solve each.



For now, let's pretend that the functions *isRaceOver* and *whichHorseWon* along with the *moveRandomHorseForward* method have already been written. (We will write them next.) If the functions and method were already written, we could write the *race* method as shown in Figure 8-1-2. The *If* part checks to see whether the game is over. If so, the game ends with the winning horse saying "I won!!!" Otherwise, the *Else* part kicks in and a horse is randomly selected to move forward a short distance. The last statement, *racehorseGame.race*, recursively calls the *race* method. The overall effect of the recursive call is repetition until some condition occurs that ends execution of the method.

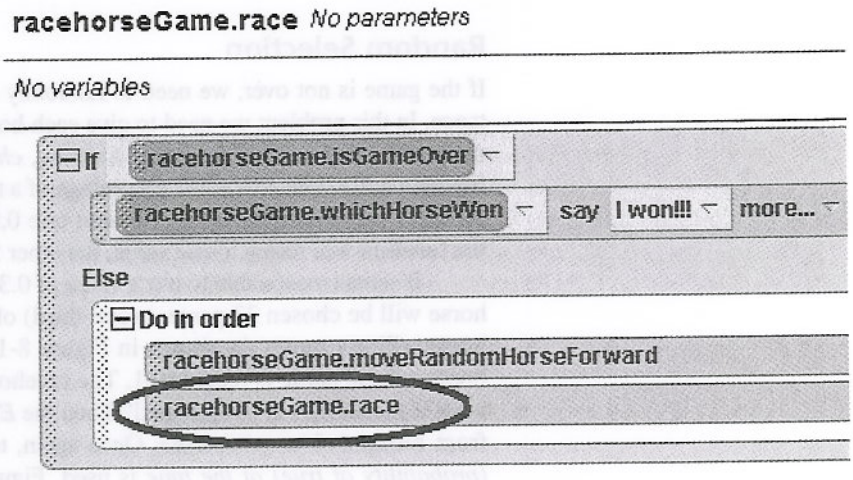


Figure 8-1-2. The *racehorseGame.race* method uses recursion (calls itself)

Now that you have the overall idea of how recursion works, we can look further at the details of implementing *isRaceOver, moveRandomHorse*, and *whichHorseWon*.

Determining when the race is over

Let's begin with the *isRaceOver* function. The race is over when one of the horses gets close enough to the finish line to be declared a winner. Either the race is over or it is not over–so the *isRaceOver* function should be a Boolean function (returns *true* or *false*). Within the *isRaceOver* function, we can make use of a built-in function. The question we will ask in the function is: "Is the finish line less than 0.5 meters in front of a horse?" The reason we have chosen 0.5 (rather than 0) is that Alice measures the distance between the horse and the finish line from the horse's center, not from the tip of its nose. In our example world, the horse's nose is approximately 0.5 meters in front of its center (a raceHorseGame has very small horses), so we use a value of 0.5 rather than 0. (If you try this world, you may resize the racehorseGame and find that this distance needs a slight adjustment.)

The function to determine if one of the horses has won is presented in Figure 8-1-3. A cascading *If/Else* control structure is used to check whether the game is over. If any one of the three horses is within the distance (0.5 meters) of the finish line, the function returns *true*. Otherwise, after all three horses have been checked and none is close enough to the finish line to end the game, the function returns *false*.

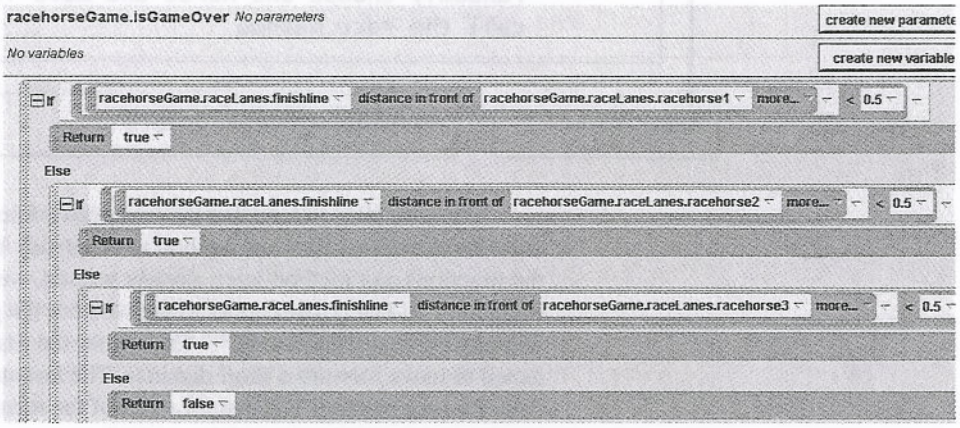


Figure 8-1-3. The *isGameOver* function

Random Selection

If the game is not over, we need to randomly choose a horse and move it forward a short distance. In this problem we need to give each horse a "fighting chance" to win the race. Let's use the world-level random selection function, *choose true (probability of true) of the time*. This function will return *true* some percentage of a total number of times (*probability of true* is a percentage). For example, saying "choose true 0.5 of the time" means that 50 percent of the time the function will return a *true* value, the other 50 percent of the time it will return a *false* value.

It seems reasonable to use a value of 0.33 (one-third) for *probability of true*, so that each horse will be chosen 33 percent (one-third) of the time. The other two-thirds of the time, that horse will not move. As shown in Figure 8-1-4, we start the random selection with the first horse, whose name is racehorse1. The racehorse1 will be selected one-third of the time. But, what if racehorse1 is not selected? Then the *Else*-part takes over and a selection must be made from racehorse2 or racehorse3. Once again, to decide which horse should move, *choose true (probability of true) of the time* is used. Figure 8-1-4 illustrates the code for the *moveRandomHorseForward* method.

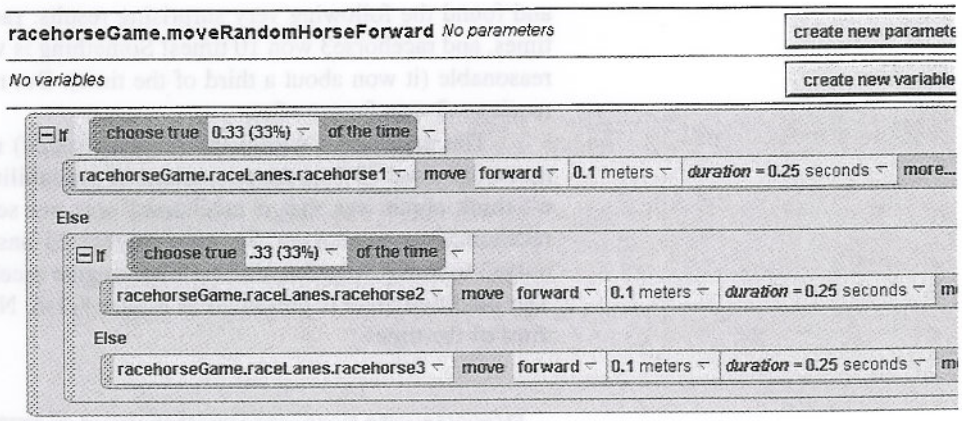


Figure 8-1-4. Randomly choose one horse and move it forward

Determining the winner

Each time the race method is executed, one of the horses moves closer to the finish line. Eventually one of the horses will get close enough to the finish line to win the race. This is when we need to determine which horse is the winner. We want to ask the question, "Did a horse get close enough to the finish line to win the race?" Once again, we use the distance (0.5 meters) of the finish line in front of the horse to check whether the horse is close enough to win. As in the *isGameOver* function, we can use a cascading (nested) *If/Else* structure to ask first about racehorse1, then racehorse2, and so on. If racehorse1 is not the winner, then we ask about racehorse2. If neither racehorse1 nor racehorse2 has won, we can assume that racehorse3 was the winner.

The *whichHorseWon* function, as illustrated in Figure 8-1-5, returns the horse object that has won the game. Note that it is impossible for two horses to cross the finish line at once as only one horse moves at a time, and we check for a winner each time one of the horses moves forward. The winning horse simply says, "I won!!!" We admit is not a very exciting end to a race, but at least it is easy to see which horse won.

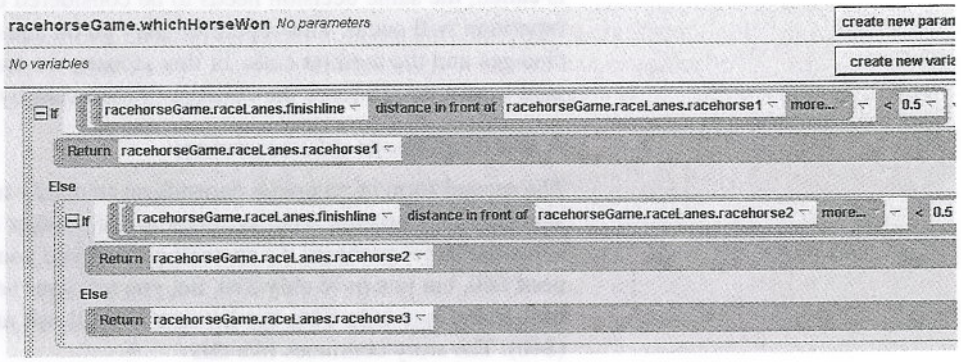


Figure 8-1-5. The *whichHorseWon* function

Testing the program

In any program where you are working with random numbers, it is especially important to do a lot of testing. Because of the random selection used in this program, different horses should win the race if the program is run several times. In this example, we ran the program 20 times

and found the following very surprising results: racehorse1 won 7 times; racehorse2 won 3 times, and racehorse3 won 10 times! Something is very wrong. The results for racehorse1 are reasonable (it won about a third of the time). But racehorse2 didn't win nearly enough, and racehorse3 won far too often.

The problem is within the second (nested) random selection of which horse should move. We used 33% as the percentage of probability for choosing each horse. What we didn't think about was that if racehorse1 was not selected, then we have only two choices: racehorse2 or racehorse3. So, after racehorse1 has been eliminated, we should select racehorse2 to move 50 percent of the time to give racehorse2 an equal chance with racehorse3. The modified code is presented in Figure 8-1-6. Now, each horse wins approximately one-third of the time!

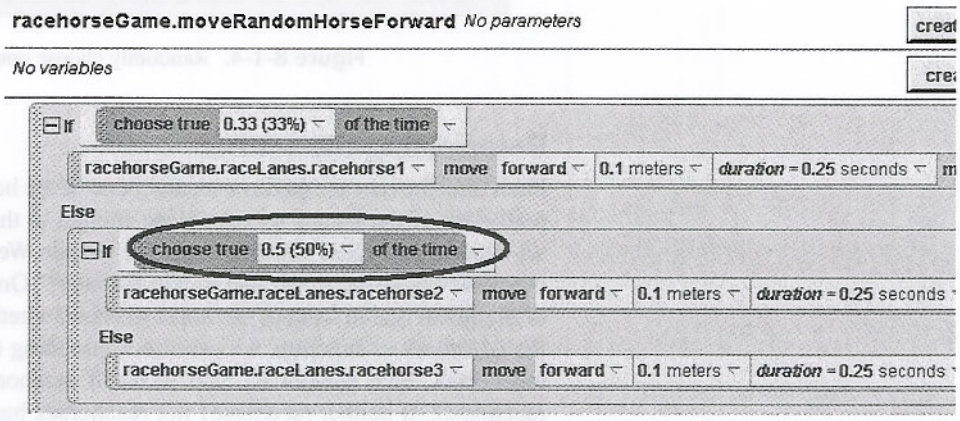


Figure 8-1-6. The corrected random-selection percentage

8-2 Another flavor of recursion

The concept of recursion, where a method calls itself, was introduced in Section 8-1. The example illustrated that recursion depends on a decision statement (*If*) that evaluates a condition. Depending on the results of that decision, a method may call itself. Each time the method is called, the same decision needs to be considered once more to determine whether another repetition will occur. This repetition may go on and on until, eventually, the tested condition changes and the method ends. In this section, we examine a second flavor of recursion. The goal here is to reinforce the concept of recursion by presenting a different kind of problem.

A second form of recursion

The second form of recursion depends on an ability to break a problem down into smaller and smaller sub-problems. The solutions to the smaller sub-problems are used to cooperatively solve the larger problem. As an analogy, suppose you have an emergency situation where you need $60, but you have only $10. So, you ask your best friend to lend you the rest ($50). Your best friend has only $10, but he says he will ask another friend for the remaining amount ($40). The story continues like this

You need $60 and have $10, so you ask a friend to borrow $50

Friend has $10 and asks another friend to borrow $40

Friend has $10 and asks another friend to borrow $30

Friend has $10 and asks another friend to borrow $20

Friend has $10 and asks another friend to borrow $10

Friend loans $10

Collectively, each friend lends $10 back up the stream of requests, and the problem of borrowing enough money is solved. (Of course, you now have the problem of paying it back!) Notice that each friend has a problem similar to your problem—but the amount of money each friend needs to borrow is successively smaller. This is what we mean by breaking the problem down into smaller and smaller sub-problems. When the smallest problem (the ***base case***) is solved, the solution is passed back up the line, and that solution is passed back up the line, and so on. Collectively, the entire problem is solved. To illustrate this form of recursion in terms of designing and writing a program, it is perhaps best to look at an example world.

Towers of Hanoi puzzle

The problem to be considered is the Towers of Hanoi, a legendary puzzle. In the ancient story about this puzzle, there are 64 disks (shaped like rings) and three tall towers. In the beginning, all 64 disks are on one of the towers. The goal is to move all the disks to one of the other towers, using the third tower as a spare (a temporary holder for disks). Two strict rules govern how the disks can be moved:

1. Only one disk may be moved at a time.
2. A larger disk may never be placed on top of a smaller disk.

Solving the puzzle with 64 disks is a huge task, and an animation showing a solution for 64 disks would take much too long to run! (In fact, assuming that it takes one second to move a disk from one tower to another, it would take many, many centuries to run!) However, we can use just four disks to illustrate a solution. Most people can solve the four-disk puzzle in just a few minutes, so this will allow you to quickly test for a correct solution.

In our solution to this puzzle, four disks (torus from Shapes folder) of varying widths have been placed on a cone (Shapes), as illustrated in Figure 8-2-1. The cones in the world will play the same role as towers in the original puzzle. Initially, four disks are on the leftmost cone.

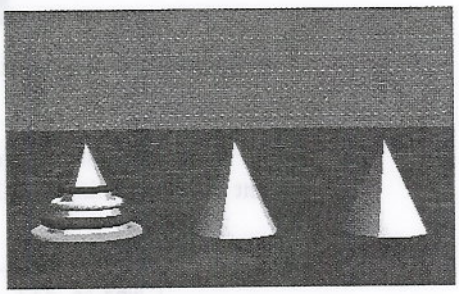


Figure 8-2-1. The Towers of Hanoi

In setting up this world, careful positioning of cones and measurement of disks will make the animation easier to program. We positioned each cone exactly 1 meter from its nearest neighbor, as labeled in Figure 8-2-2.

Each disk is placed exactly 0.1 meter in height above its neighbor, as illustrated in Figure 8-2-3.

To make it easier to describe our solution to the puzzle, let's give each disk an ID number and a name. The disk with ID number 1 is disk1, ID number 2 is disk2, ID number 3 is disk3, and ID number 4 is disk4. The smallest disk is disk1 at the top of the stack and the largest disk is disk4 at the bottom. Also, let's name the cones cone1, cone2, and cone3 (left to right from the camera point of view). The goal in the puzzle is to move all the disks from the leftmost cone, cone1, to another cone. In this example, we will move the disks to the rightmost cone (cone3).

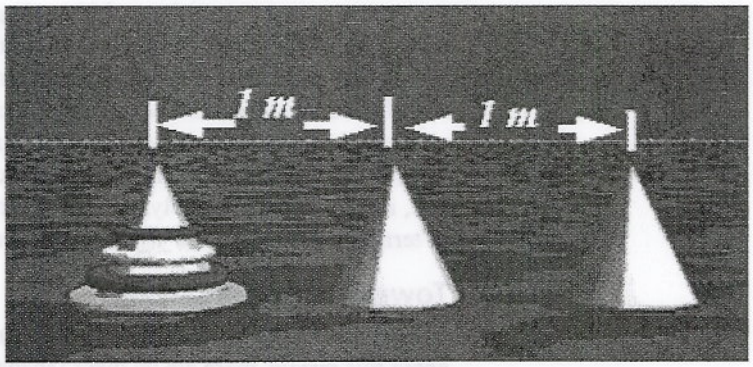


Figure 8-2-2. Each cone is 1 meter from neighboring cone

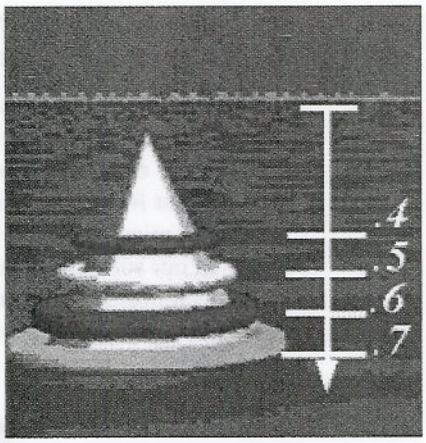


Figure 8-2-3. Each disk is 0.1 meter above its neighboring disk (measuring from top down)

A first attempt to solve the problem might be to use a while loop. The textual storyboard might look something like this:

```
While all the disks are not on the target cone
  Do
```

Unfortunately, it is not easy to figure out what needs to be done inside the "Do ." The problem can be solved using a *While* loop, but it takes much thinking and insight into the problem. Recursion makes the problem much easier to think about and solve, so we will use recursion.

Two requirements

We want to solve this puzzle for four disks using the second form of recursion, where the problem is broken down into smaller and smaller sub-problems. To use this form of recursion, two requirements must be met.

The first requirement is that we must assume we know how to solve the problem for a smaller sub-problem. Let's assume that we do know a solution for solving the problem

for three disks. If we know how to solve the problem of moving 3 disks, it would be quite easy to write a program to solve it for 1 more disk (four disks). The following steps would work:

1. Move the three disks (imagining the solution for the puzzle with only three disks is already known) from cone1 to cone2. See Figure 8-2-4(a).
2. Move the last disk, disk4, from cone1 to cone3. See Figure 8-2-4(b). (Remember that this move is now safe, as all of the three smaller disks are now located on cone2.)
3. Move the three disks (again, imagining the solution is already known) from cone2 to cone3. The final result is shown in Figure 8-2-4(c).

The second requirement is that we must have a base case. A base case is the simplest possible situation where the solution is obvious and no further sub-problems are needed. In other words, when we get down to the base case, we can stop breaking the problem down into simpler problems because we have reached the simplest problem. The obvious "base case" in the Towers of Hanoi puzzle is the situation where we have only one disk to be moved. To move one disk, we can just move it. We know that there are no smaller disks than disk 1. So, if it is the only disk to be moved, we can always move it to another cone!

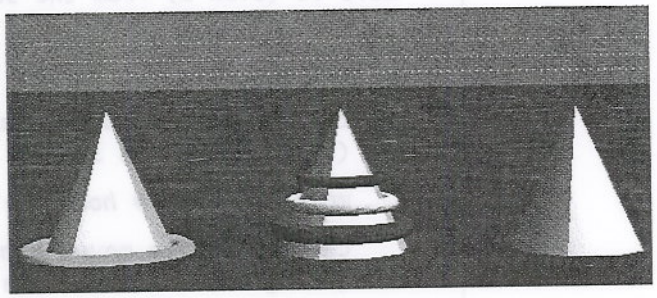


Figure 8-2-4(a). After step 1

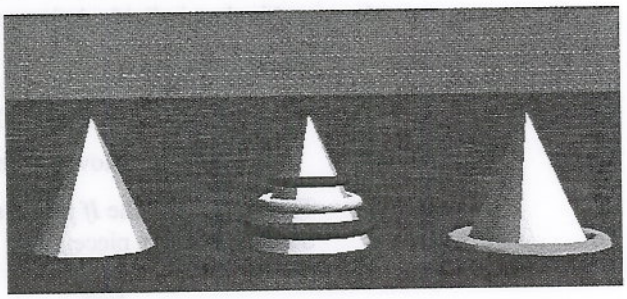


Figure 8-2-4(b). After step 2

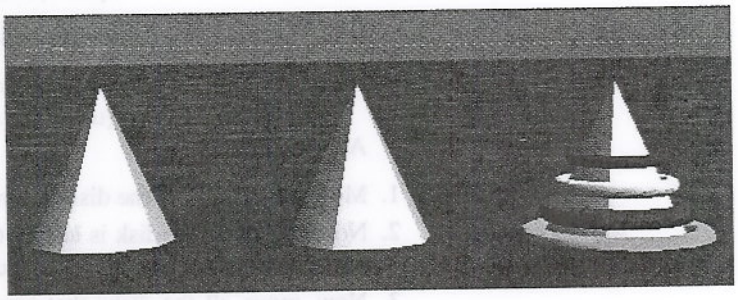


Figure 8-2-4(c). After step 3

The towers method

Now that we have determined the necessary two requirements, we can write a method to animate a solution to the Towers of Hanoi puzzle. The method, named *towers*, will have instructions to move some number *(howmany)* of disks from a *source* cone (the cone where the disks are currently located) to a *target* cone (the cone where the disks will be located when the puzzle is solved). In the process of moving the disks from the source cone to the target cone, a spare cone (the cone that is neither source nor target) will be used as a temporary holder for disks on their journey from the source to the target.

To do its work, the *towers* method needs to know how many disks are to be moved, the source cone, target cone, and spare cone. To provide this information to the *towers* method, four parameters are used: *howmany, source, target,* and *spare*. Here is the storyboard:

```
towers
Parameters: howmany, source, target, spare
If howmany is equal to 1
   move it (disk 1) from the source to the target
Else
   Do in order
      call towers to move howmany-1 disks from source to spare
      (using target as spare)
      move it (disk # howmany) from the source to the target
      call towers to move howmany-1 disks from the spare to the target
      (using the source as the spare)
```

We know this looks a bit complicated; but it really is not too difficult. Let's break it down and look at individual pieces:

First, the *If* piece:

if *how many* is equal to 1

move it (the smallest disk) from the *source* to the *target*

This is simple. The *If* piece is the base case. We have only 1 disk to move, so move it! Second, the Else piece:

Else

(1) call *towers* to move *howmany*-1 disks from *source* to *spare* (using *target* as *spare*)

(2) move it (disk # *howmany*) from the *source* to the *target*

(3) call *towers* to move *howmany*-1 disks from *spare* to *target* (using the *source* as the *spare*)

All this is saying is:

1. Move all but one of the disks from the *source* to the *spare* cone.
2. Now that only one disk is left on the *source*, move it to the *target* cone. This is ok, because all smaller disks are now located on the *spare* cone, after step 1.
3. Now, move all the disks that have been temporarily stored on the *spare* cone to the *target* cone.

Notice that the *If* piece and the *Else* piece in this storyboard each have a sub-step that says "move it" (move a disk) from the source cone to the target cone. The *moveIt* sub-steps are highlighted in the storyboard below. Moving a disk from one cone to another is actually a combination of several moves, so we will need to use stepwise refinement to break down our design into simpler steps.

```
towers
Parameters: howmany, source, target, spare

If howmany is equal to 1
    move it (the smallest disk) from the source to the target

Else
  Do in order
    call towers to move howmany-1 disks from source to spare
    (using target as spare)

    move it (disk # howmany) from the source to the target

    call towers to move howmany-1 disks from the spare to the target
                    (using the source as the spare)
```

```
moveIt:
```

Exactly what does the *moveIt* method do? *Move it* must lift the disk upward to clear the top of the cone it is currently on. Then, it must move the disk (forward or back) to a location immediately above the target cone. Finally, it must lower the disk down onto the target cone. Figure 8-2-5 illustrates a possible sequence of moves.

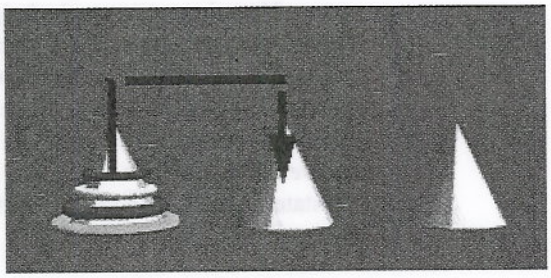


Figure 8-2-5. Moving a disk from source to target cone

To write the *moveIt* method, we need to use three parameters because it needs to know: (1) which disk is to be moved (*whichdisk*—the disk ID number), (2) the source cone (*fromcone*), and (3) the target cone (*tocone*). A storyboard for *moveIt* could be:

```
moveIt
Parameters: whichdisk, fromcone, tocone

Do in order
        Lift the disk up above the top of the fromcone
        Move it (forward or back) to a location above the tocone
        Lower the disk down onto the tocone
```

Two methods are to be written: (1) the *towers* method and (2) the *moveIt* method. Let's start by writing the *towers* method, as shown in Figure 8-2-6. The code is a straightforward translation of the storyboard.

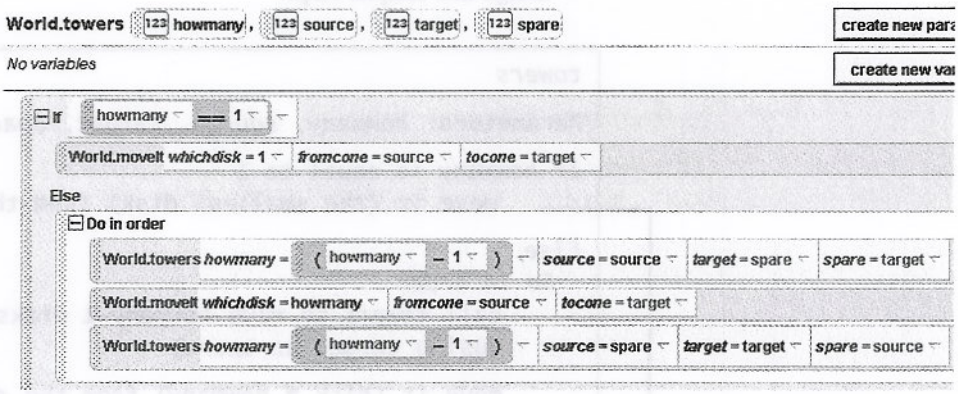


Figure 8-2-6. The *towers* method

The moveIt method

The first step in the *moveIt* method is to lift the disk up above the cone it is currently on. How high should the disk be lifted? Each disk is at a different initial height on the cone, so it will be necessary to raise each disk a different amount. In our example world, we made each disk 0.1 meters in height, so we need to lift disk1 approximately 0.4 meters, disk2 approximately 0.5 meters, disk3 approximately 0.6 meters, and disk4 approximately 0.7 meters to "clear" the cone. (Figure 8-2-3 illustrates the careful positioning of the disks on the source cone in the initial world.)

Now that we know the height to lift each disk, we can write the code to move a disk upward. Of course, the *moveIt* method needs to know which disk is to be moved. One possibility is to pass a parameter to the *moveIt* method that contains the ID number of the disk to be moved. If the disk ID is 1, move disk1, if the disk ID is 2, move disk2, and so on. We can use nested *If* statements to check on the ID number of the disk. When the correct disk is found, a *move* instruction can be used to lift it the appropriate amount. The storyboard (for the nested *If* statements) would look something like this:

```
If whichdisk is 1 then
  Move disk1 up 0.4 meters
Else
  If whichdisk is 2 then
        Move disk2 up 0.5 meters
Else
        If whichdisk is 3 then
              Move disk3 up 0.6 meters
        Else
              Move disk4 up 0.7 meters
```

The code in Figure 8-2-7 will accomplish this task.

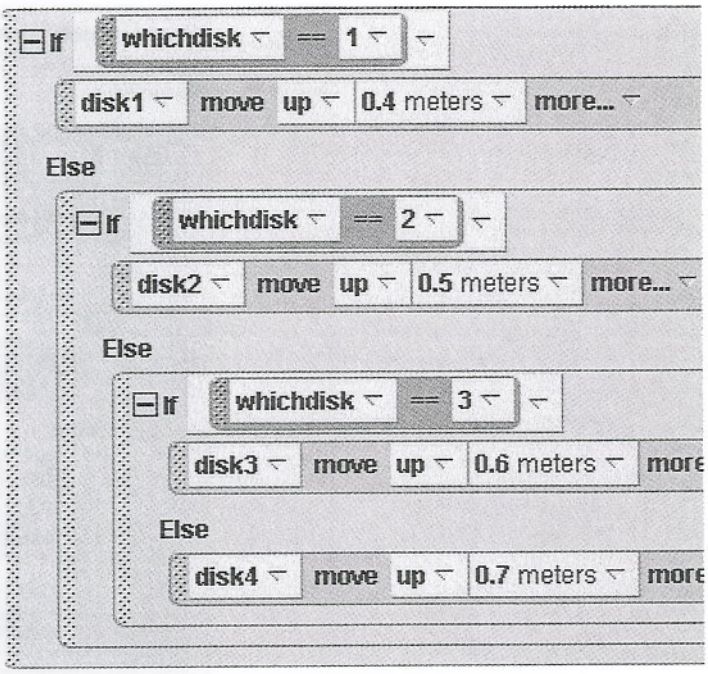


Figure 8-2-7. Nested *If* statements to select the disk to move upward

We thought about this for a while, realizing that the nested *If* statements might seem a bit awkward. We noticed that the distance each disk has to move up is 0.3 meters more than 0.1 * the ID number (*whichdisk*). Thus, a nifty mathematical expression could be used to move a disk upward. Using an expression to compute the distance would allow the storyboard to be just one step:

```
move the appropriate disk up 0.3 + 0.1 * whichdisk
```

Sometimes it is helpful to come up with an elegant idea like this one. We want to point out, however, that condensing the code into a more compact form may lead to other problems. In this case, using an expression to compute the distance leads us to a problem of how to write the statement. Notice that the one-statement storyboard uses the phrase "appropriate disk" because we don't know exactly which disk is to be moved. All we have is *whichdisk*—an ID number, not a name. Think of it like this: you have a name and a social security number. When someone talks to you, she says your name, not your social security number. Writing a *move* instruction is a similar situation. A *move* instruction uses the name of the object, not its ID number.

Conversion function

"How is Alice to be told the name of the disk object to be moved when only the disk ID number is known?" One solution is to write a conversion method that takes the ID number as a parameter and returns the appropriate name of the disk object to move. Such a conversion method can be written using a function. The *which* function is illustrated in Figure 8-2-8. In this code, each *If* statement includes an instruction containing the keyword *Return*. As you know, the *Return* statement sends information back to the method that called it. For instance, suppose $which(i = 2)$ is called; the information that will be returned is disk2. The *which* function provides a way to convert an ID number (the *whichdisk* parameter) to an object name. (We named the function *which* to make it easy to mentally connect the *which* function to the *whichdisk* parameter.)

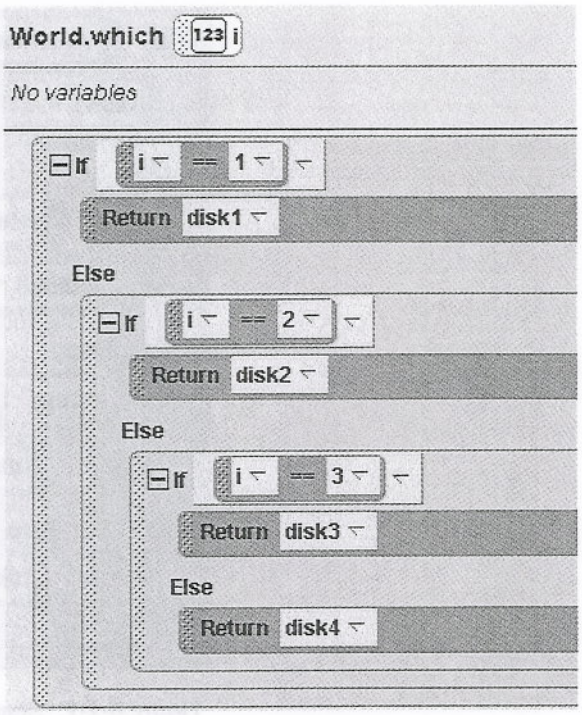


Figure 8-2-8. A world-level conversion function

An instruction can now be written in the *moveIt* method that calls the *which* function to determine which disk to move and uses a mathematical expression to compute the distance, as illustrated in Figure 8-2-9. In this instruction, *which*$(i = whichdisk)$ is a call to the *which* function. When executed, *which*$(i = whichdisk)$ will select the appropriate disk and return its name. That disk will then be moved upward the computed amount, as explained in the earlier expression.

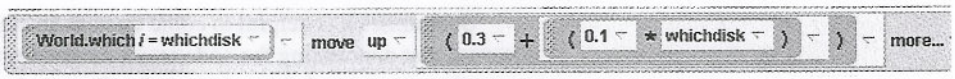


Figure 8-2-9. Calling the function to determine which disk to move

Completing the moveIt method

Recall that the *moveIt* method is composed of a sequence of three movement instructions. So far, we have completed the first *move* instruction (highlighted in blue in the storyboard, below). Let's look at how to write the two remaining instructions.

```
moveIt

Parameters: whichdisk, fromcone, tocone

Do in order
        Lift the disk up above the top of the fromcone
        Move it (forward or back) to a location above the tocone
        Lower the disk down onto the tocone
```

For the second instruction, we want to move the disk *forward* or *backward* so as to position it immediately over the target cone. How far should the disk be moved? As previously illustrated (Figure 8-2-2), the cones are purposely positioned exactly 1 meter from one another. There are six possible moves, as shown in Table 8-2-1.

Table 8-2-1. Six disk moves from source cone to target cone

Forward moves	forward distance	Backward moves	forward distance
from cone1 to cone2	1 meter	from cone2 to cone1	−1 meter
from cone1 to cone3	2 meters	from cone3 to cone1	−2 meters
from cone2 to cone3	1 meter	from cone3 to cone2	−1 meter

Notice that moving forward a negative (−1 meter) distance is the same as moving backward 1 meter. After examining the six cases in detail, we see that the forward distance can be computed using an expression (*tocone* − *fromcone*). For example, to move a disk from cone1 to cone3, move it 2 meters (3 − 1) forward. With this insight, an instruction can be written to move the disk to the target cone, as shown in Figure 8-2-10.

Figure 8-2-10. Moving the disk the appropriate amount forward/back

The last step in the *moveIt* method is to move the disk downward onto the target cone. This instruction should simply do the opposite of what was done in step 1. The complete *moveIt* method appears in Figure 8-2-11.

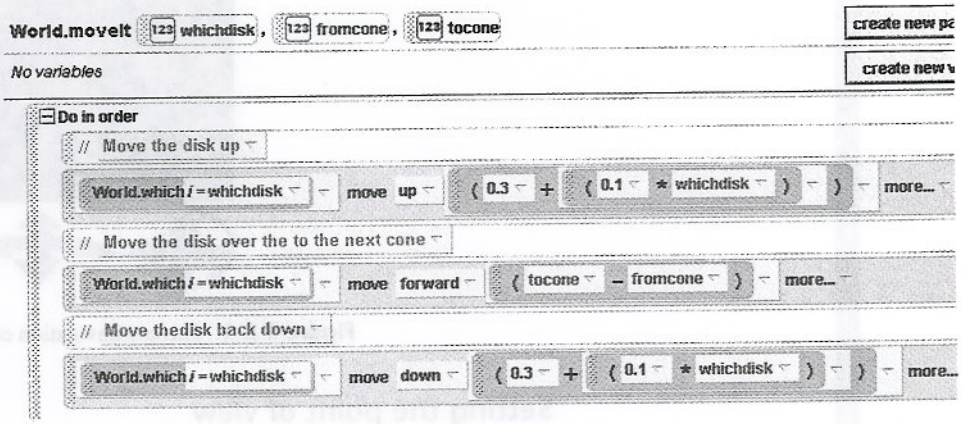


Figure 8-2-11. The code for *moveIt*

Now, with the *towers* and the *moveIt* methods completed, all that remains is to call the *towers* method when the world starts. A link in the Events editor can be used, as seen in Figure 8-2-12.

The second form of recursion (presented in this section) depends on the structure of a problem and its solution, breaking a problem down into smaller and smaller sub-problems. Many mathematicians (and computer scientists interested in logic) often prefer this form of

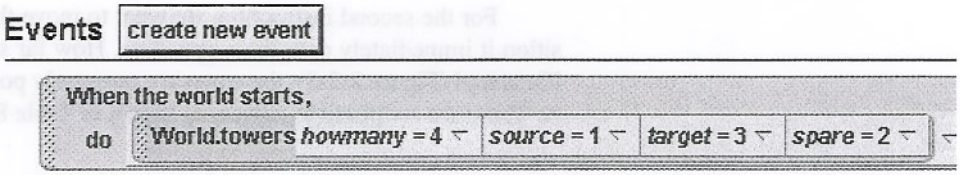


Figure 8-2-12. Calling the *towers* method

recursion because it is easy to show that the program does end—and that it ends with the correct solution. For example, with the Towers of Hanoi, we reasoned that we could:

move 4 disks, if we can move 3 disks and then 1 disk (we know how to move 1 disk)
move 3 disks, if we can move 2 disks and then 1 disk
move 2 disks, if we can move 1 disk and then 1 disk
move 1 disk (base case—we know how to move 1 disk)

Clearly, the problem has been broken down into smaller and smaller sub-problems. We know that the program will end, because we know that eventually the problem size will get down to the base case, which is moving one disk.

Tips & Techniques 8
Camera and Animation Controls

The camera controls in the Alice scene editor, as in Figure T-8-1, allow you to move the camera position at the time an initial scene is being created. Of course, the camera controls are not available while the animation is running. In this section we look at a technique for controlling the movement of the camera at runtime. Also, we offer a tip on how to use the speed multiplier to speed up (or slow down) an animation at runtime.

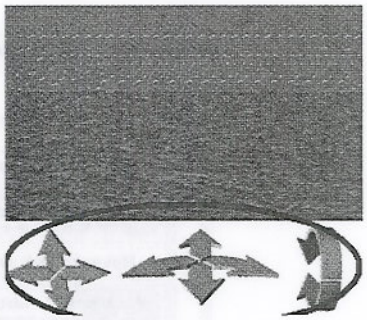


Figure T-8-1. Camera navigation controls in the scene editor

Setting the point of view

One way to move the camera around a scene is to create one or more dummy objects that mark locations where the camera will be used during the animation. A dummy object is invisible and can be used as a goalpost—a target object to which the camera will move during the animation. Once a dummy object is in place, an instruction can be created to set the camera *point of view* to the *point of view* of the dummy object. At runtime, the *set point of view* method effectively moves the camera to the dummy object (similar to a *move to* instruction), and the camera viewpoint is now from that location.

This technique is best illustrated with an example. You may find it helpful to sit at a computer and try this out as you read the description. Figure T-8-2 shows an initial scene with the

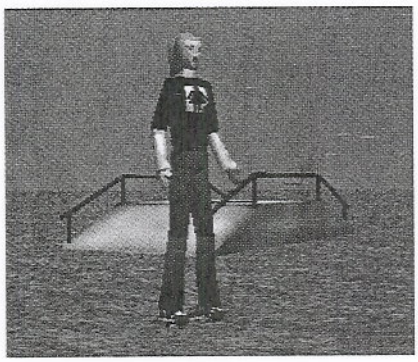


Figure T-8-2. Camera front view

camera pointed at a skateboarder (People). The skateboard and the rail-ramp can be found in the SkatePark folder on the CD or Web gallery. From our perspective (as the person viewing the scene), the camera is allowing us to look at the scene "from the front."

We want to be able to move the camera around to view the skater's actions "from the back" of the scene while the animation is running. To prepare for this action, let's create a couple of dummy objects for the front and back viewpoints. First, drop a dummy object at the current location of the camera. To do this, click the **more controls** button in the scene editor, as in Figure T-8-3.

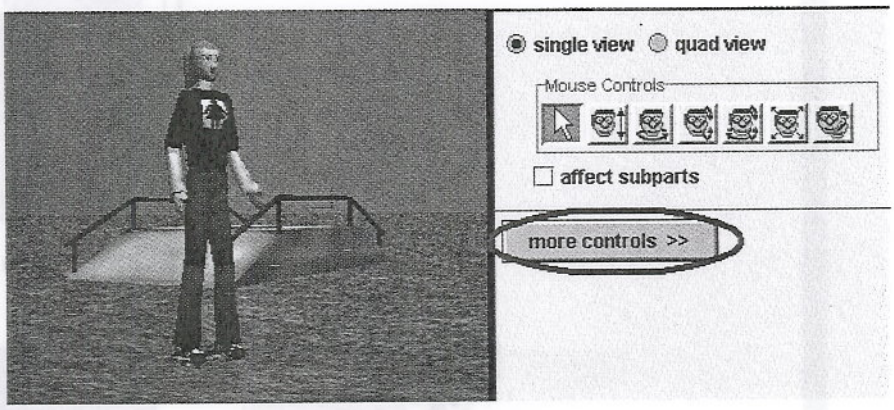


Figure T-8-3. The more controls button

The interface expands to show additional controls, including a **drop dummy at camera** button, seen in Figure T-8-4.

A click on the drop dummy at camera button creates a dummy object as a goalpost at the current location of the camera. The dummy object is added to the Object tree, as shown in Figure T-8-5. Alice automatically names dummy objects as dummy, dummy2, etc. In this example, we renamed the dummy object as frontView (a meaningful name makes it easy to remember).

The next step is to reposition the camera to view the scene from the back. One technique that seems to work well is to drag the camera's forward control (the center camera navigation arrow) toward the back of the scene. Continue to drag the forward control and allow the camera to move straight forward until the camera seems to move through the scene. When the camera seems to have moved far enough to be on the other side of the scene, release the mouse. Select the camera in the Object tree and select a method to turn the camera one-half revolution, as in Figure T-8-6.

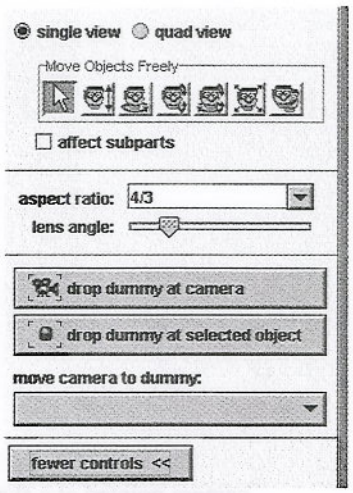


Figure T-8-4. The buttons for camera control with dummy object

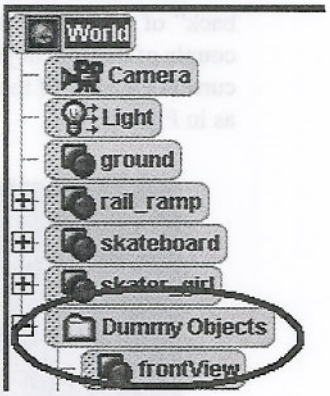


Figure T-8-5. Dummy object is added to the Object tree

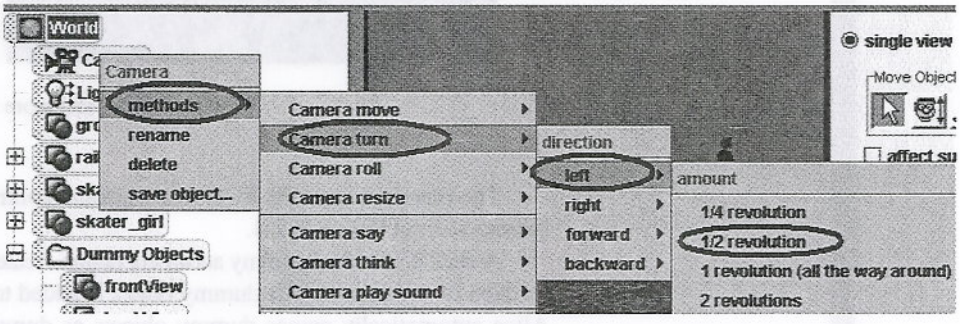


Figure T-8-6. Move camera to back of scene and then turn it around one-half revolution

The camera should now be facing in the opposite direction and you should be able to see the scene from the back. Click once more on the drop dummy at camera button. We renamed the dummy marker as backView. Two dummy objects should now be in the Object tree, frontView and backView. To return the camera to the front view, select frontView from the dropdown menu of the **move camera to dummy** button (see Figure T-8-4).

Now instructions can be written to move the camera at runtime. Select camera in the Object tree and then drag the *set point of view* instruction into the editor. Select the particular dummy object for the point of view. Figure T-8-7 illustrates how to create the instruction.

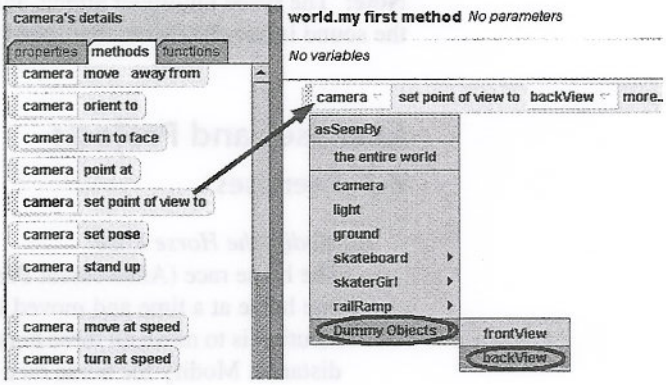


Figure T-8-7. Creating a *set point of view* instruction

A sample instruction is shown below. At runtime, this instruction will move the camera to backView and the camera's *point of view* will be from that location.

Controlling the runtime speed

As you gain experience in programming with Alice, your animations will become more and more complex and the runtime longer and longer. Naturally, this means that testing your programs takes longer because the animation takes a while to run through the sequence of actions to get to the most recent additions.

A great way to "fast forward" through the actions you have already tested is to use the Speed Multiplier that appears in the Play window, as seen in Figure T-8-8.

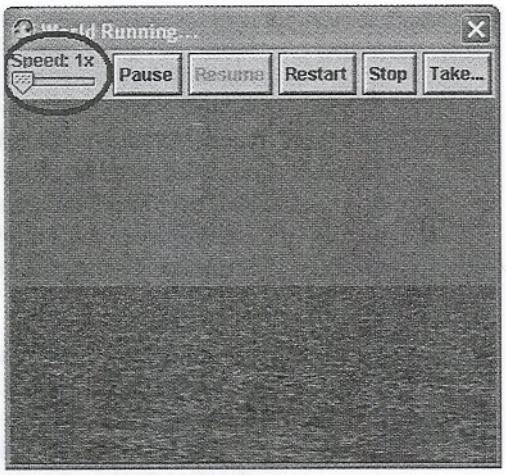


Figure T-8-8. The speed multiplier

The Speed Multiplier is a slide bar with a thumb. You can grab the thumb with the mouse and slide it left or right while the animation is running to control the speed of the animation. (Sliding the thumb has no effect if the animation has stopped.) To return the animation to normal speed, simply drag the thumb all the way to the left.

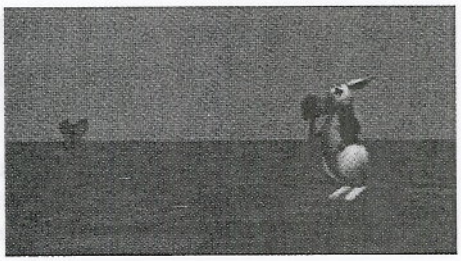
Note: The speed multiplier affects only the speed of the animation. If a sound is being played, the sound is not affected by the speed multiplier (that is, the sound does not play faster).

Exercises and Projects

8-1 Exercises

1. *Modify the Horse Race*
   The horse race (Amusement Park) example presented in Section 8-1 randomly selected one horse at a time and moved it forward until one of the horses won the race. Another solution is to move all three horses forward at the same time, but to move each a random distance. Modify the horse race program to use this solution to the problem.

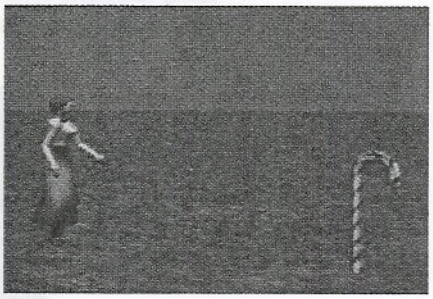
2. *Butterfly Chase*
   Use recursion to make the white rabbit (Animal) chase a butterfly (Animals/Bugs). The butterfly should fly to a random, nearby location (within 0.2 meters in any direction). The white rabbit, however, must always remain on the ground (rabbits do not fly). Each time the butterfly moves, use *turn to face* to make the rabbit turn toward the butterfly and then move the rabbit toward the butterfly 0.5 meters. When the rabbit gets close enough, have him catch the butterfly in his net (Objects).



   Hint: To move the butterfly to a random nearby location, you can adapt the *randomMotion* method presented in Tips & Techniques 6. It is necessary to keep the butterfly from flying too high or too low. (The butterfly could disappear below the ground or fly too high, so that the rabbit would never get close enough to catch it.) To limit the up-down direction of the butterfly's movement, use an *If* statement to check the butterfly's *distance above* the ground and then move the butterfly up or down accordingly. The butterfly's distance above the ground should always be within the range of 0 (ground level) and 1 (meter above the ground).

3. *Midas Touch 1*
   In the story of the Midas Touch, a greedy king was given the gift of having everything he touched turn to gold. This exercise is to create a simulation of the Midas Touch. In the initial scene shown below, a woman is facing a candy cane (Holidays). (Use *turn to face* to make this alignment.) Write a recursive method, named *checkCandy*, that checks whether the woman's right hand is very close to the candy cane. If it is, the woman should touch the candy cane. After she touches the candy cane, the candy

cane turns to gold (color changes to yellow). If the woman is not yet close enough to the candy cane to be able to touch it, she moves a small distance forward and the *checkCandy* method is recursively called.



4. *Midas Touch 2*
Create a second version of the MidasTouch1 world. This new version will be interactive. The idea is to allow the user to guide the movement of the woman toward a candy cane. To make this a bit more challenging, the woman should NOT be facing the candy cane in the initial scene. This will require the *checkCandy* method be modified so that whenever the woman is close enough to the candy cane to touch it, she first turns to point toward the candy cane before bending over to touch it.

Use the left and right arrow keys to turn the woman left or right. Create two methods: *turnRight* and *turnLeft*. The methods should turn the woman 0.02 revolutions to the right or left when the user presses the right or the left arrow key. These methods will allow the user to guide the woman toward a candy cane.

Hint: It is possible that the woman will wander out of the range of the camera. There are two possible solutions. One is to make the camera point at the woman each time she moves. The other is to make the camera's vehicle be the woman.

8-2 Exercises

5. *Towers of Hanoi*
Create the Towers of Hanoi puzzle as described Section 8-2. When you have it working, modify the event-trigger link that calls the *towers* method so that it moves the disks from cone1 to cone2 (instead of cone3).

6. *Towers of Hanoi Modified*
We do not need to pass around the spare cone as a parameter. The spare cone is always

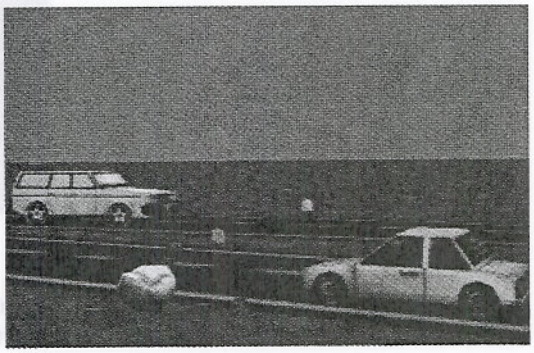
$$6 - fromcone - tocone.$$

Modify the towers method to use this new approach.

7. *On the Hour*
Place a cuckoo clock (Objects) in a new world. The idea of this animation is to have the cuckoo clock keep time (not in real time, of course). The minute hand should go around on the face of the clock (perhaps one complete revolution should take about 30 seconds in real time) and the pendulum should swing back and forth. When the minute hand has made one revolution (from 12 back to 12 on the face of the clock), then the hour hand should advance to the next hour, the doors should open, and the cuckoo bird (on the clock arm) should come out and chirp once. Then the bird should retreat inside the clock and the doors should close until the next hour has gone by. "All is well" as long as the clock is running—which should continue until the user stops the animation. Use a recursive method to implement the animation.

8-2 Projects

1. *Why Did the Chicken Cross the Road?*
A popular child's riddle is, "Why did the chicken cross the road?" Of course, there are many answers. In this project, the chicken (Animals) has a real sweet-tooth and crosses the road to eat the gumdrops (Kitchen/Food) along the way.



Write a game animation where the player guides the chicken across the road to get to the gumdrops. Cars and other vehicles should move in both directions as the chicken tries to cross to where the gumdrops are located. Use arrow keys to make the chicken jump left, right, forward and back. Use the space bar to have the chicken peck at the gumdrop. When the chicken is close enough to the gumdrop and pecks, the gumdrop should disappear.

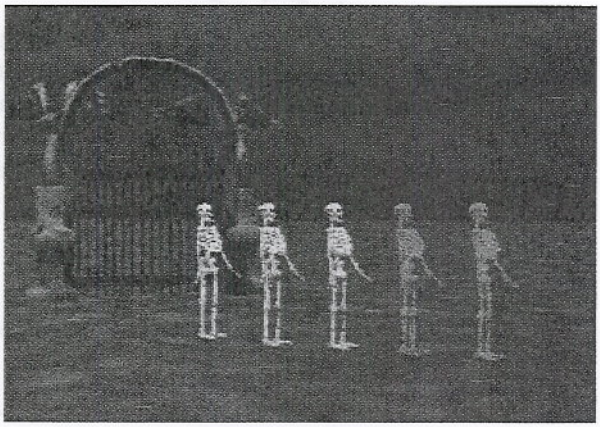
A recursive method is used to control the play of the game. If the chicken gets hit by a vehicle, the game is over (squish!). The game continues as long as the chicken has not managed to peck all the gumdrops and has not yet been squished by a vehicle. If the chicken manages to cross the road and peck at all the gumdrops along the way, the player wins the game. Signal the player's success by making 3D text "You Win" appear or by playing some triumphant sound.

2. *Reversal*
In the world below, the row of skeletons (Spooky) is guarding the gate (Spooky). Every so often in this world, the row of skeletons is to reverse order. This project is to animate the reversal using the second form of recursion. The storyboard goes something like the following:

```
reverse

If the row of skeletons not yet reversed is more than one then
   reverse the row of skeletons starting with the second skeleton (by recursively
                calling reverse)
   move the head skeleton to the end of the row
```

The base case is when there is just one skeleton in the row (that has not yet been reversed). Of course, a row of 1 skeleton is already reversed! The recursive case (for n skeletons, where n is larger than 1) says to first reverse the last $n - 1$ skeletons and then move the first skeleton to the end of the row.

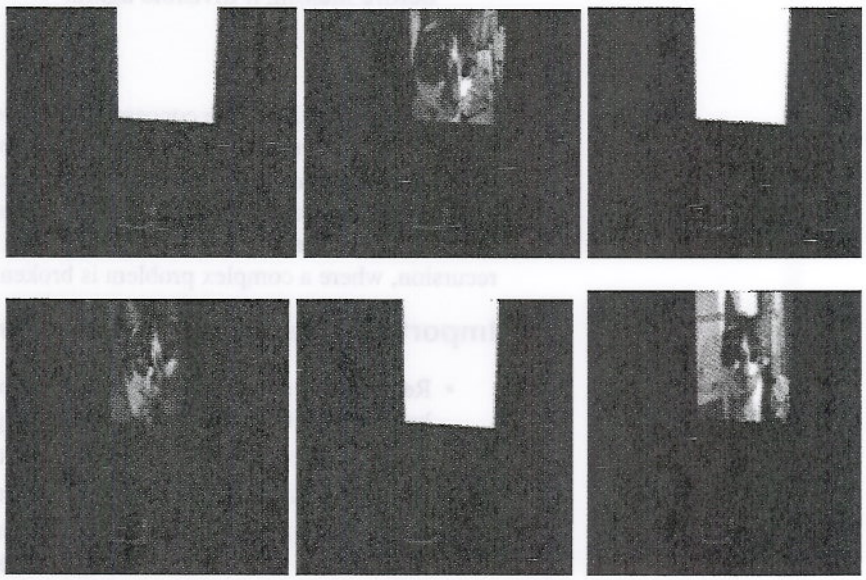
Implement the Skeleton reversal storyboard given above. The program you write should be quite similar to the Towers of Hanoi program, including the *which* function.



3. *Never Ending Slide Show*

This project makes use of the technique of changing the skin of an object (Tips & Techniques 3). In this world, you are to create a never-ending slide show. Set up a new world with a square (Shapes) as the screen, the slide projector on a table (Furniture on CD or Web), and a lightbulb (Lights) in front of the screen, as shown in the image below left. The lightbulb is for the purpose of lighting up the screen (simulating light from the projector). Then, make the lightbulb invisible by setting its *isShowing* property to *false*.

For an added sense of realism, start with the world's light on and them dim the light before turning on the projector. The slide show in your project must display at least three different slides. Numbering the slides and using an *If* statement may help you create a method to change slides. Between each slide, the screen must go blank, the projector light must flicker (change the color of the light), and the slide projector's tray of slides must rotate. (The blank slides below show the screen going blank between one slide and the next.) The slides are to change continuously (meaning once all three have been seen, the show should go back to slide 1 and start over). The sequence of images is a sample slide show. (The cat pictures are of Muffin, one of our graduate student's cats.)



4. *Click-a-Cow*

   Let us design a game where the goal is to click on an object that is appearing and then disappearing on the screen. If the user successfully clicks on the object while it is visible, some visual action should happen so the user will know he/she has managed to "click" the object with the mouse. In our sketch of a storyboard below, a cow (Animals) is the object and a windup key (Objects) is used to signal success. (Of course, you can creatively choose a different object and a different way to signal the fact that the user has clicked on the object.)

   We assume that the cow and wind-up key have been added to the world and both have been made invisible by setting their *isShowing* property to *false*. A storyboard for a method, *check*, will look something like this:

```
If the wind-up key is not visible
        Move the cow to a random location on the screen
        Make the cow visible
        Wait a short period of time, say 0.25 seconds
        Make the cow invisible
        Move the cow off the screen
          (perhaps by moving it down 10 meters so the user can no longer click on it)
        Recursively call check
Else
        Have the cow say, "Thanks for playing"
```

When the user finally manages to click the mouse on the object, a method should be called that signals success. In our example, the wind-up key would become visible. Once the key is visible, the game ends.

Hint: The most challenging part of this project is to move the object to a random location on the ground (perhaps between −3 and 3 in the forward-back and right-left directions). Also, you will find it helpful to experiment with a *Wait* instruction to find out how long to wait while the object is visible (not too fast, not too slow) before making it invisible again.

Summary

This chapter introduced the concept of using recursion as a mechanism for repetition. Recursion is a powerful tool for building more complex and interesting worlds.

The horse race example in this chapter demonstrates a kind of recursion where more executions of a section of program code are "generated" each time the result of the previous decision is true. The famous Towers of Hanoi puzzle is an example of a second flavor of recursion, where a complex problem is broken down into simpler and simpler versions.

Important concepts in this chapter

- Recursion is most useful when we cannot use a *Loop* statement because we do not know (at the time we are writing the program) how many loop iterations will be needed.
- Recursion occurs when a method calls itself.
- Recursion enables a method to be repeatedly called.
- Any recursive method must have at least one base case, where no recursive call is made. When the base case occurs, the recursion stops.

- When you use recursion, you generally do not also use a loop statement (*Loop* or *While*). Instead, an *If* statement is used to check a condition, and the decision to recursively call the method depends on the value of the condition.

A comparison of repetitions

Chapters 7 and 8 presented three ways for accomplishing repetition in a program: *Loop*, *While*, and recursion. You might ask, "Which one should I use, and when?"

The choice depends on the task being completed (the problem being solved) and why you need the repetition. If you know how many times the repetition needs to be performed (either directly or via a function), the *Loop* statement is generally easiest. In most programming that you do, you will want to avoid infinite loops. In animation programming, however, an infinite *Loop* statement can be used to create some background action that continues to run until the program is shut down.

Otherwise, use the *While* loop or recursion. Some programmers prefer *While* and others prefer recursion. Put two programmers in the same room and they are likely to disagree over this issue. The underlying difference between the two is how you think about solving a problem. We say, "Try both." Some problems are easier to solve with a *While* loop and others are easier to solve with recursion. With experience, you will discover which works best for you. The important thing is that you learn to write code that uses repetition.