6 Decisions and User-defined Questions

This chapter introduces the concept of *conditional execution* of a segment of code in a program. A *condition* is the answer to a *question* about a current situation in the world. Our program checks a current condition in the world and makes a *decision* either **to execute** the code or **not to execute** the code. (Shakespeare considered a similar question!) Conditional execution is a key concept in computer programming that allows us to run some part of the program code only if some condition is true.

In section 6-1, we look at how to write the code (encode) decision statements in Alice. In the real world, we make decisions all the time. In our virtual world, we use decisions to make animations execute in different ways depending on a condition such as where an object is located, what color it is, or whether or not an object is visible.

In sections 6-2 and 6-3, we delve into questions as a way of determining the current conditions in the world and as a way of getting information about objects. Built-in questions were previously discussed in Tips & Techniques 4. (Perhaps you will want to review that section.) In this chapter, we look at writing our own questions.

From a programming perspective, questions are pure functions. In other words, questions do some computation, and return a value (leaving the state of the world unchanged). Questions naturally follow decisions. We often wish to invoke a question, and depending on the value the question returns, either perform some action or not. In computer science terminology, decisions and conditions allow us to ***control the flow of execution***.

6-1 Decisions and Logical Questions

This section begins with an explanation of the process of making decisions in a program. Decisions are useful when writing programs where a method or some instructions are expected to run only under certain conditions.

Decisions

Sometimes life is "one decision after another." We only go out to mow the lawn if the grass isn't wet. We only run the dishwasher if it's full of dirty dishes. We only put a leash on the dog when taking the dog outside for a walk.

Programming, too, is full of the decisions. In Alice an *If/Else* statement (we will often refer to *If/Else* as an *If* statement) is an instruction that makes a decision based on the answer to a *logical question*. A logical question is one that can be answered *true* or *false*. In computer science, the term *Boolean question* is often used.[5] In Alice, an *If* statement looks like the following:

The green color of this block is a visual clue that an *If* statement is being used in the program. The *If/Else* statement has two parts (an *If* part and an *Else* part). If the answer to the question is *true*, the *If* part is executed and the *Else* part is skipped. But, if the answer to the question is "false" then the *If* part is skipped and the *Else* part is executed. The word *true* appears to the right of the word *If*. By default, Alice places a tile containing the word **true** in the statement and then allows the programmer to drop a question on top of the tile that will (at runtime) evaluate to either true or false. The following example demonstrates how the *If* statement is used in practice.

If/Else Example

In section 5-2, we revisited the world of Greek mythology with a Zeus animation. In the storyline for this world, a user can choose the next target of the god's anger by clicking on a philosopher. Then, Zeus shoots his thunderbolt at the selected philosopher. (For convenience, the code for the *shootBolt* method is reproduced in Figure 6-1-1.) We had intended that Zeus only shoot the thunderbolt at a philosopher. Unfortunately, we found that if the user clicks on a cloud, Zeus also shoots his thunderbolt at the cloud. We had the same problem for the ground and the sky – and any other object that happened to be in the world. The problem is quite clear: **we need some way to control** whether or not Zeus shoots the thunderbolt at an object.

[5] Boolean questions are named after the 19th century English mathematician, George Boole, who developed symbolic logic and was the first (as far as we know) to be interested in expressions that can only evaluate to either *true* or *false*.

Figure 6-1-1. The *shootBolt* method (from Zeus world of section 5-2)

In the shootBolt method (as illustrated in Figure 6-1-1), any object that is mouse-clicked is automatically passed in to the parameter, *who*. We want to change the *shootBolt* method so that the bolt is shot at the object only if the object (represented by the parameter *who*) is one of the philosophers. The mechanism for making this decision is the *If* statement. We drag the *If*

statement into the shootBolt method and *true* is initially selected for the condition. Then, the *Do in order* block is dragged inside the if statement block. The idea here is that the bolt will be shot at the *who* object only if some condition is true. The last step is to drop a question on top of the **true** tile to indicate what question is to be asked. As shown in Figure 6-1-2, we ask the question "**who = = Homer**". In Alice, "==" means "*is equal to.*" We call *is equal to* a *relational operator* because it checks the relationship between two values.

Figure 6-1-2. Shooting the bolt only at Homer

To create the question "who = = Homer," the *who* tile is dragged on top of the *true* tile and *Homer* is selected from the cascading menu, as shown in Figure 6-1-3. Now, when the program is run, if Homer is mouse-clicked, Zeus shoots a bolt at Homer.

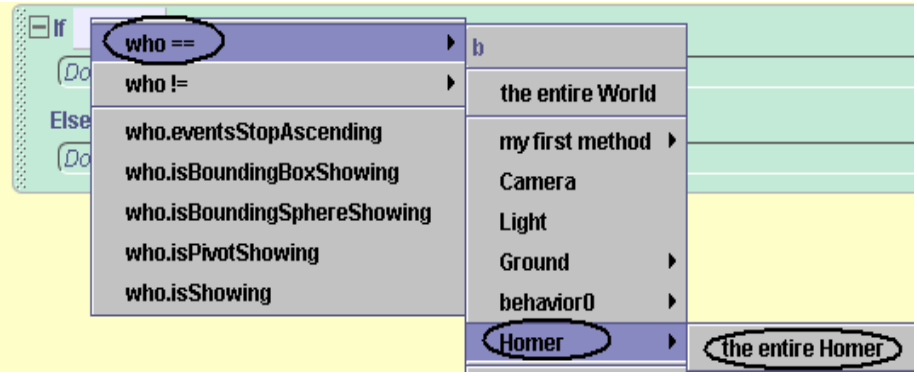


Figure 6-1-3. Selecting who == question

Logical Operators

Of course, the "who = = Homer" question only checks for the condition that the object clicked is Homer. What about the other philosophers? This is an example where more than one condition is possible. We need a question that will also have Zeus shoot a bolt if the parameter *who* is Plato, Socrates, or Euripides. One way to make this happen is to use the logical operator *or*. The *or* operator is available in the questions tab for the *World,* as shown in Figure 6-1-4. The *or* operator is a logical operator in the *boolean logic* category of world-level questions. The *or* operation means exactly what it sounds like: "either this or that or possibly both." For example, I will have a cone of vanilla ice cream **or** a cone of strawberry ice cream **or** I'll take a swirl of both.

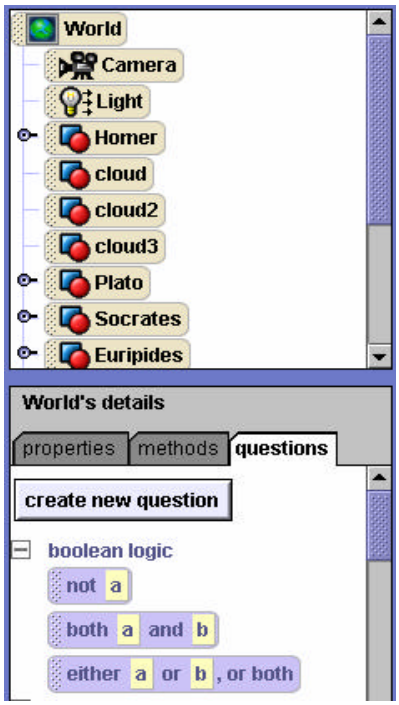


Figure 6-1-4. Logical operators

To use the *or* operator, the "*either a or b or both*" tile is dragged over the condition tile in the *If* statement. In this example, we need to drag and drop the operator three times to account for all of the philosophers. Figure 6-1-5 illustrates the modified *If* statement. (The statement is broken into

two lines to make it easily fit on the printed page – but is all on one line in Alice.) Now, when the program is run, clicking on any philosopher results in Zeus shooting a thunderbolt at that philosopher, but clicking on something else in the world causes no action.

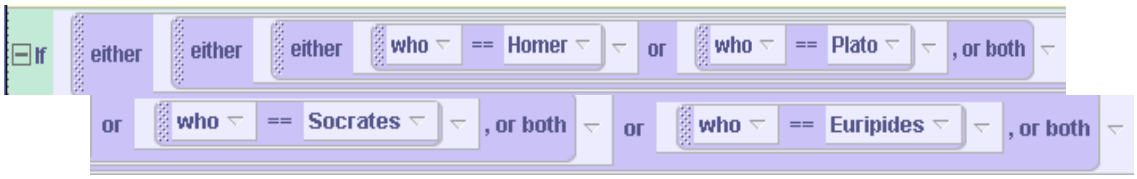


Figure 6-1-5. Multiple conditions in an *If* statement

The *or* operator is only one of three logical operators available in Alice. Another logical operator (as can be seen in Figure 6-1-3) is the *not* logical operator. The *not* operator behaves just as it sounds – if the Boolean expression is *true*, *not* of the Boolean expression is *false*. And if the Boolean expression is *false*, *not* of the Boolean expression is true. The example,

evaluates to true only when the object clicked (represented by the parameter *who*) is not Homer. If the object clicked is Homer, the above expression evaluates to *false*.

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

evaluates to *true* only if both the object clicked on is Homer, and the object clicked on has its color 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 the object clicked on is Homer **and** Homer's color is black **or** blue.

But, the expression shown below evaluates to *true* if the clicked object is black **or** it will evaluate to true if the clicked object is Homer **and** Homer's color is blue

These examples point out that levels 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 instead, as described in the next section.

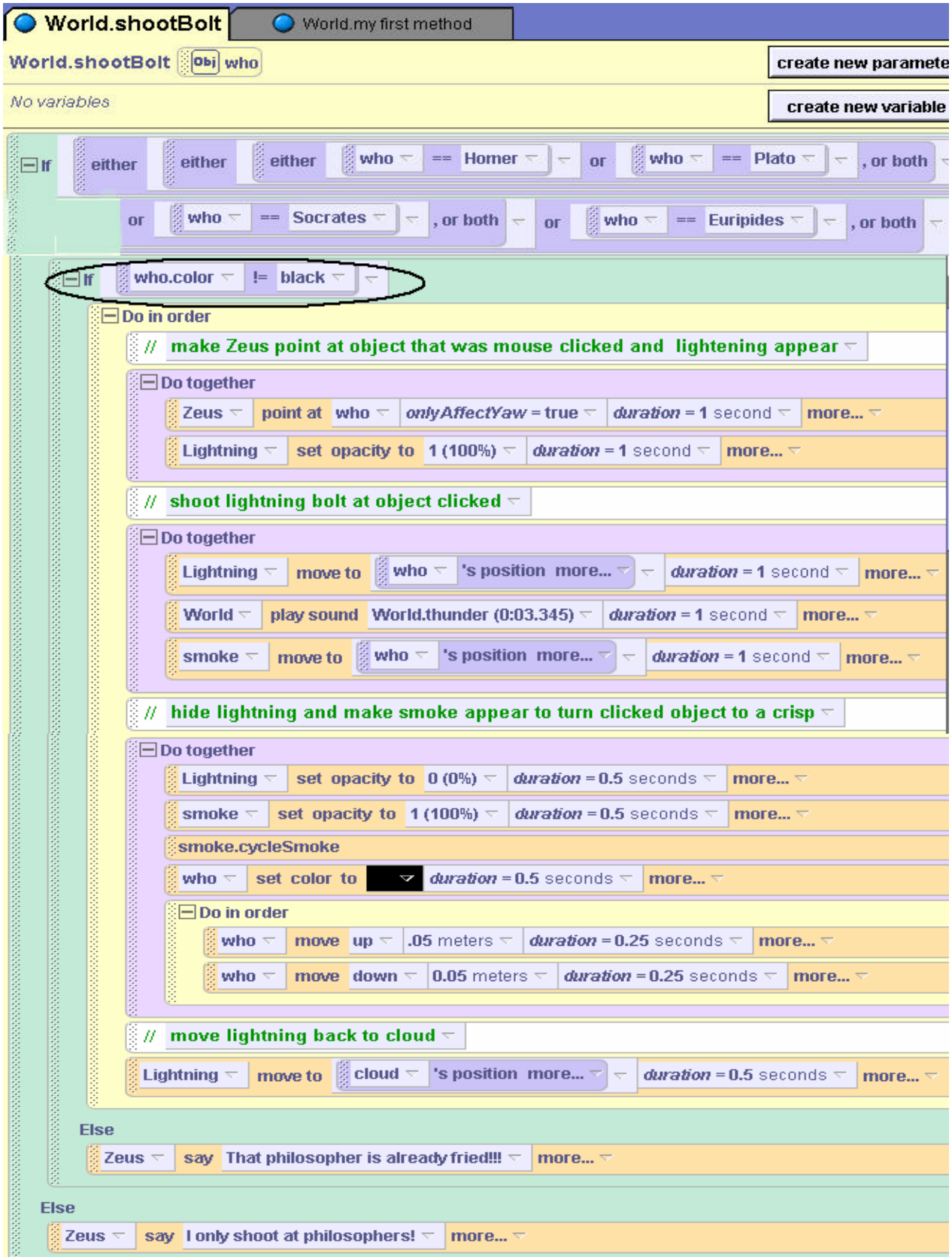
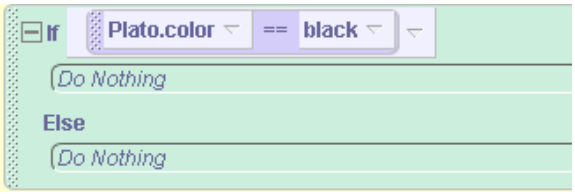


Figure 6-1-6. The complete code for the *shootBolt* method

Nesting *If* statements

One problem in our Zeus world still exists! In testing the animation, we discovered that clicking on a philosopher who has already been shot by a thunderbolt results in Zeus shooting another thunderbolt at the "already-fried" philosopher. (That seems like a waste of energy.) How can we prevent this from happening? One solution is to use another *If* statement, allowing Zeus to only shoot a thunderbolt at a philosopher who isn't already frizzled. Because each philosopher object is turned a black color (to show the effect of being hit by lightening), we can use the color property to determine whether the object has already been struck by lightening. The process of creating the question to test the color of the *who* parameter is a three step process. First, the If statement is added to the code. Then, one of the object's color property (we arbitrarily chose Plato) is dragged into the *If* statement and an = = question is asked to determine if the color is *black*.



Finally, the *who* parameter is dropped on top of *Plato* to allow the color of any philosopher object to be checked.

The completed code is illustrated in Figure 6-1-6.

Else

Code has also been added to the Else parts of the If statements. In case the user clicks on an object other than one of the four philosophers, Zeus says, "I only shoot at philosophers," and if the user clicks on a philosopher who has already been hit by a thunderbolt, Zeus now says "The philosopher is already fried!!!"

Relational operators

In the Zeus world, we made use of the = = *relational operator*. *If* statements often depend on relational operators so this is a topic we should explore a bit further. It is often the case that we would like to compare two numbers, and execute code if a certain relationship exists between those numbers. For example, if a boy's height is at least 4 feet (approximately 1.3 meters), then he should be allowed to ride a roller coaster. Alice provides six world-level relational operators grouped together in the math category of world-level questions, as illustrated in Figure 6-1-7.

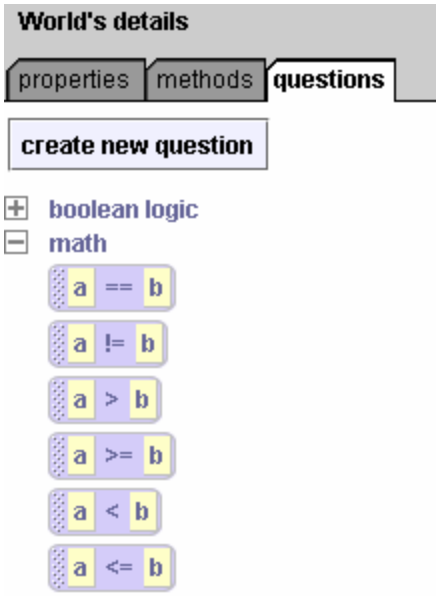
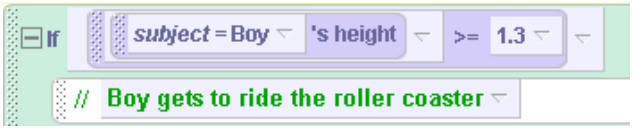


Figure 6-1-7. Relational operators

These operators allow us to compare two numbers in six different ways! While "= =" means "**is equal to**," the "**!=**" operator means "**is not equal to**." To create a logical expression (one that returns true or false) that asks the question whether the "boy's height is at least 1.3 meters" is a two-step process.

1) Drag the "*a* >= *b*" tile into an *If* statement, and enter a value of 1 for *a* and 1.3 for *b*.

2) Then, drag the boy's **height** question over the leftmost number.



6-1 Exercises

1. Modifications to the Zeus world

Modify the Zeus world to make each philosopher say something different when clicked.

a) Euripides says "Come on guys, I wanna to take a bath."
b) Plato says, "I call it… Play Doe" and then extends his right hand to show the other philosophers his Play Doe.
c) Homer says, "By my calculations, pretzels go extremely well with beer."
d) Socrates says "Like sands in the hour glass, so are the days of our lives."

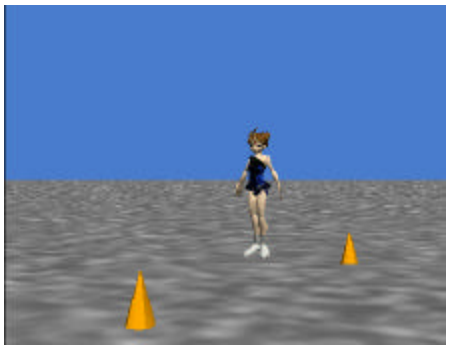
Use an If statement to determine which philosopher was clicked and have the appropriate philosopher philosophize.

2. More modifications to the Zeus world

Modify the Zeus world so that if Homer gets clicked and zapped by the thunderbolt, he he falls over, says "d'oh", and then stands back up again (instead of turning black). Allow repeated clicking on Homer, which should result in his repeated falling down and getting back up.

3. PracticeTurns

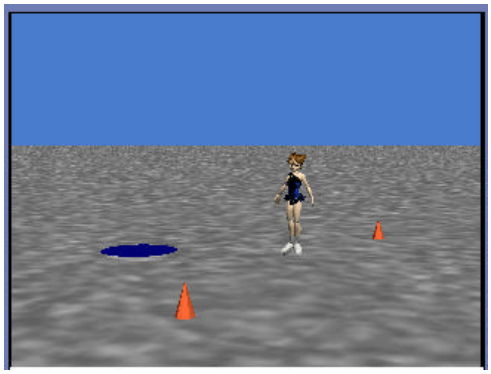
Create a skater world, as illustrated below. Import a **CleverSkater** object, as designed and created in Chapter 4. (If you have not created the CleverSkater character, an iceSkater object can be used from the gallery but you will have to write your own methods to make her skate forward and skate around an object.)



In this world, the skater is practicing turns on the ice. The skater will point at the cone and then skate forward toward the cone (a sliding step on one leg and then a sliding step with the other leg). When she gets close to a cone, she skates half way around the cone and ends up facing the other way to skate back towards the other cone. Then, she skates toward the other cone and when she gets close enough makes a turn around it. To find out whether the skater has gotten close enough to a cone to do a half circle turn around the cone, you can use the "*is within threshold of object*" question for the IceSkater object. Another possibility is to use the "distance to" question and the relational operator $a < b$ (available as a world-level question) to build the logical expression "is the skater's distance to the cone is less than 2 meters?" Write your program to make the skater complete a path around the two cones.

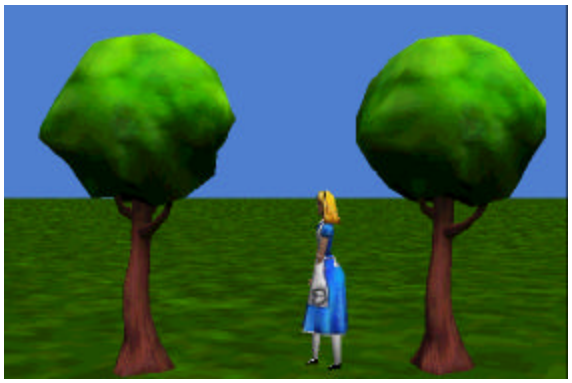
5. FigureEight This exercise is an extension of exercise 3 above.
Modify the world to have the skater complete a figure 8 around the cones.

6. IceDanger
For this exercise, you can begin with a world constructed in either exercise 4 or 5 above – 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 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 the hole. If you have sound on your computer, you may want to add a splash sound that plays when the skater falls through the ice.

7. TallTrees
Create a world with an Alice object (or some other object positioned between two tall trees). Animate Alice walking back and forth between the two trees. Make the world interactive so Alice takes a step each time the user presses the enter key. Alice should walk until she reaches a tree, then turn around to walk back toward the other tree. When she gets to the second tree, she should turn around to walk back towards the first tree. Be sure to avoid Alice colliding with one of the trees.



6- 2 User-defined Questions I (Boolean)

Introduction to Questions

Alice uses the term *question* to refer to a program construct known in many other programming languages as a *function*. A question (function) may receive values sent in to parameters (input), perform some computation on the values, and return (send back) a value as output. **In some cases, no input is needed – but, generally values are sent in.** The diagram in Figure 6-2-1 outlines the overall mechanism. One way of thinking about a question or function is that it is something like an old fashioned jukebox. You put coins in the machine and select a song. The machine loads the recording and sends the music out through speakers for you to enjoy. Actually, you have been using functions all your life, in many cases never thinking about it. For example, a cash register at a supermarket acts as a form of function. The cashier enters (as input) the prices of each of the items you are going to purchase, the cash register computes a sum and adds on the tax, and the cash register returns (as output) the total cost of all the items.

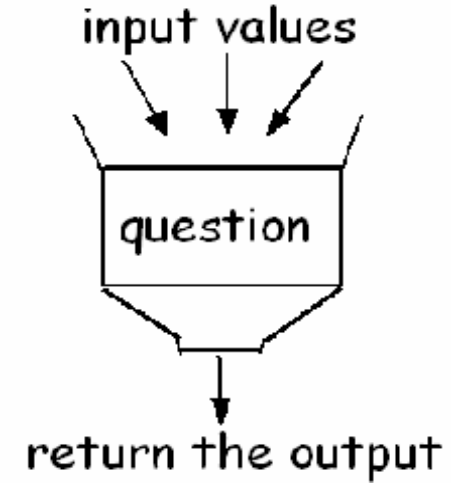


Figure 6-2-1. The functionality of a question

Abstraction

As with character-level and world-level methods, one of the important benefits of a question is it allows us to think about the overall process rather than all the nitty-gritty little details. When we use a cash register, for example, we think about finding the cost of a purchase – not about all the additions that are going on inside the machine. In the same way, we can call a question in our program to perform all the small actions. But, we just think about what we are going to get when the question returns the answer. Like methods, questions are an example of *abstraction* – collecting lots of small steps into one meaningful idea to allow us to think on a higher plane.

User-defined questions

We have already used some of Alice's built-in questions. In the Zeus world, we used a question to get the cloud's position so the thunderbolt could move to a precise location. Also, the == question was used in the Zeus world to test whether the object clicked by the user was a philosopher or not. But, sometimes we would like to use a question that does not already exist in Alice. This is when we want to write our own questions. We will call the questions we create **user-defined questions** because we are *using* the Alice system to *define questions*.

Creating a new Question

To create a user-defined question, select the World (for a world-level question) or an object (for a character-level question) in the object tree. In the questions tab of the details pane, click on the "*create new question*" tile, as in Figure 6-2-2.

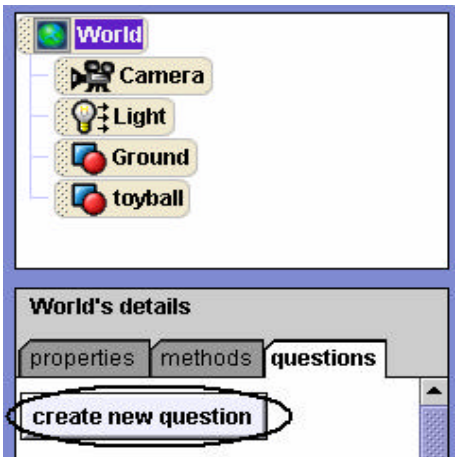


Figure 6-2-2. Create new question tile

A popup **New Question** box, see Figure 6-2-3, allows you enter the name of the new question and select its type. As with built-in questions, a user-defined question is categorized by the type of information it returns.

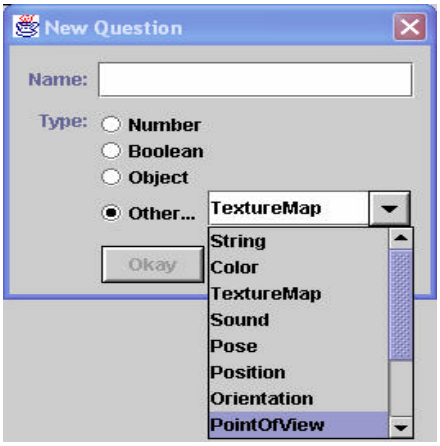


Figure 6-2-3. Types of user-defined questions

In this section, we look at writing questions that return a Boolean value (*true* or *false*). Boolean questions are logical expressions often used in *If* statements. In the next section, we will examine questions that return a value that is not Boolean.

Writing a simple Boolean question

Consider the pond scene in Figure 6-2-4. A worker bee is scouting for new sources of pollen for the hive. He is checking out the flowers surrounding the pond. In our animation, we want to write a method that will have the bee fly over to the nearest flower. There are two flowers, a red flower and a pink flower, that are possible choices. But there is no built-in question in Alice that

returns whether one object is closer than another object. So, a user-defined, world-level question is needed.

Figure 6-2-4. An initial pond scene

This question will use three parameters for input values: the comparing object (the bee), and two objects to test which one is closer to the comparing object (the red flower and the pink flower). A possible storyboard for the user-defined question is:

> *Parameters:* comparingObject, FirstObject, secondObject
>
> **If** comparingObject's distance to firstObject is less than comparingObject's distance to secondObject
>
> return true
>
> **Else**
>
> return false

The question will return *true* if the first object is closer to the bee than it is to the second object, and false otherwise. A new question named *isFirstCloser* is created in the World questions tab of the details pane and the code is entered in the editor. The code is shown in Figure 6-2-5.

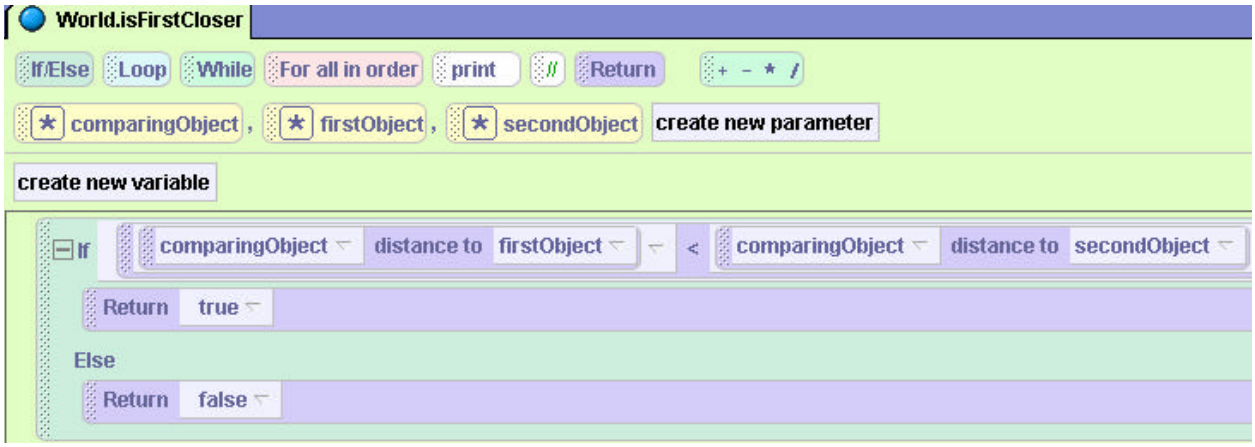


Figure 6-2-5. A user-defined question to return true if the first object is closer

Notice that the big difference between a method and a question is that the question must end with a *Return* statement. It is the Return statement that provides an answer each time the question is asked. If the Return statement is not written as part of the question, Alice will not be able to send back an answer the question.

Calling the question

Once the question has been written, it may be invoked from another method as illustrated in Figure 6-2-6. Depending on which flower is closer to the bee, the bee will turn and point at the appropriate flower, and then fly over to it.

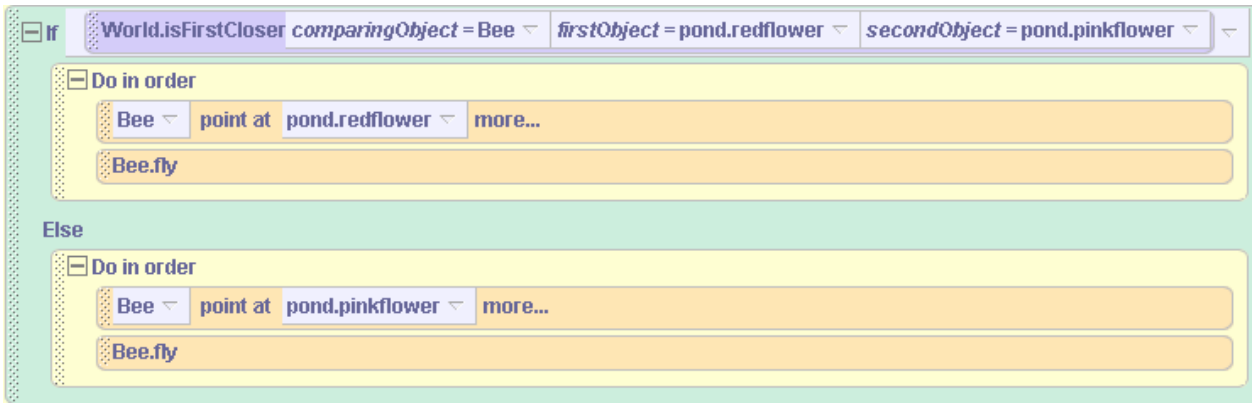


Figure 6-2-6. Calling the user-defined question

When user-defined questions are designed, it is often the case that we want to make the question somewhat generic. That is, we might want to use the same question with different objects. In this example, the *isFirstCloser* question can be used to test which of any two objects is closer to any comparing object. For example, we could use the question to find the nearest of two horses so a cowboy could stride over and hop aboard the nearest horse.

A more complex Boolean question

A biplane and a helicopter are flying in the same flyspace at approximately the same altitude, as in Figure 6-2-7. When two vehicles are in the same flyspace, a collision is possible. We want to write a question that can be used to find out whether the objects are in danger of collision. If the objects are too close to one another, the biplane can invoke a method to avoid collision.

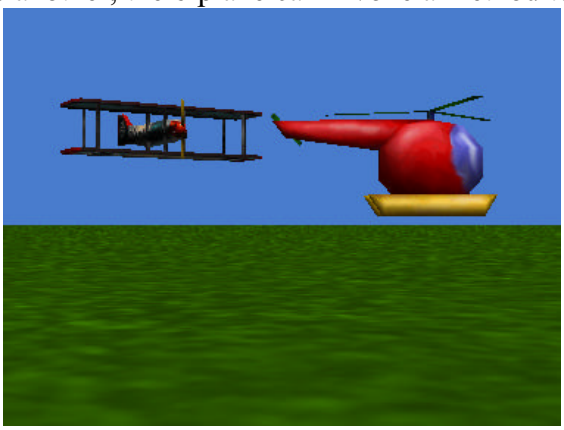


Figure 6-2-7. Flyspace collision danger

One factor in determining whether two aircraft are in danger of collision is to use a height differential (the relative heights of the two objects above the ground). In this example, the question will be designed to return *true* if the height differential is less than 10 meters (an arbitrary value). Otherwise, the question will return *false*. A possible storyboard is:

Parameters: firstObject, secondObject

If firstObject height is equal to secondObject height
  return true
Else do nothing
If firstObject is above the secondObject and its height above the secondObject < 10
  return true
Else do nothing
If secondObject is above the firstObject and its height above the firstObject < 10
  return true
Else do nothing
return false

In this storyboard, the two objects (whose heights are to be compared) are passed to the question as parameters. Three possible conditions spell danger: the two aircraft can be at the same height, the biplane can be above the helicopter and within 10 meters of it, or the helicopter is above the biplane and within 10 meters of it. If any one of these conditions is true, the question will return true. If none of these cases are true, the question will return a default value of false. The code for the user-defined Boolean question is presented in Figure 6-2-8.

Figure 6-2-8. A question with multiple If statements for multiple conditions

The code for this question consists of three consecutive *If* statements. The first statement for which the condition is true will return the value true and the question will be done. To be more explicit, we think about the code executing like this:

- *If* the first condition is *true*, the question will return the value *true* and the remaining two *If* statements will not be executed. If the first condition *is* false, the second *If* statement will execute next.
- *If* the second condition is true, the question will return the value true and the remaining *If* statement will not be executed. If the first and second conditions are both false, the third *If* statement will execute next.
- *If* the third condition is true, the question will return the value true and the question will end.
- Finally, if all three conditions are false the question will fall through to the very last return statement and the question will return false.

Some programmers prefer a cascading style for writing successive *If* statements, as illustrated in Figure 6-2-9. The code works exactly the same, but the *If* statements are nested one within the other. In the cascading style for writing the code, we think about the code executing like this:

If the first condition is true, return true.
Else, if the second condition is true, return true.
Else, if the third condition is true, return true.
Else return true.

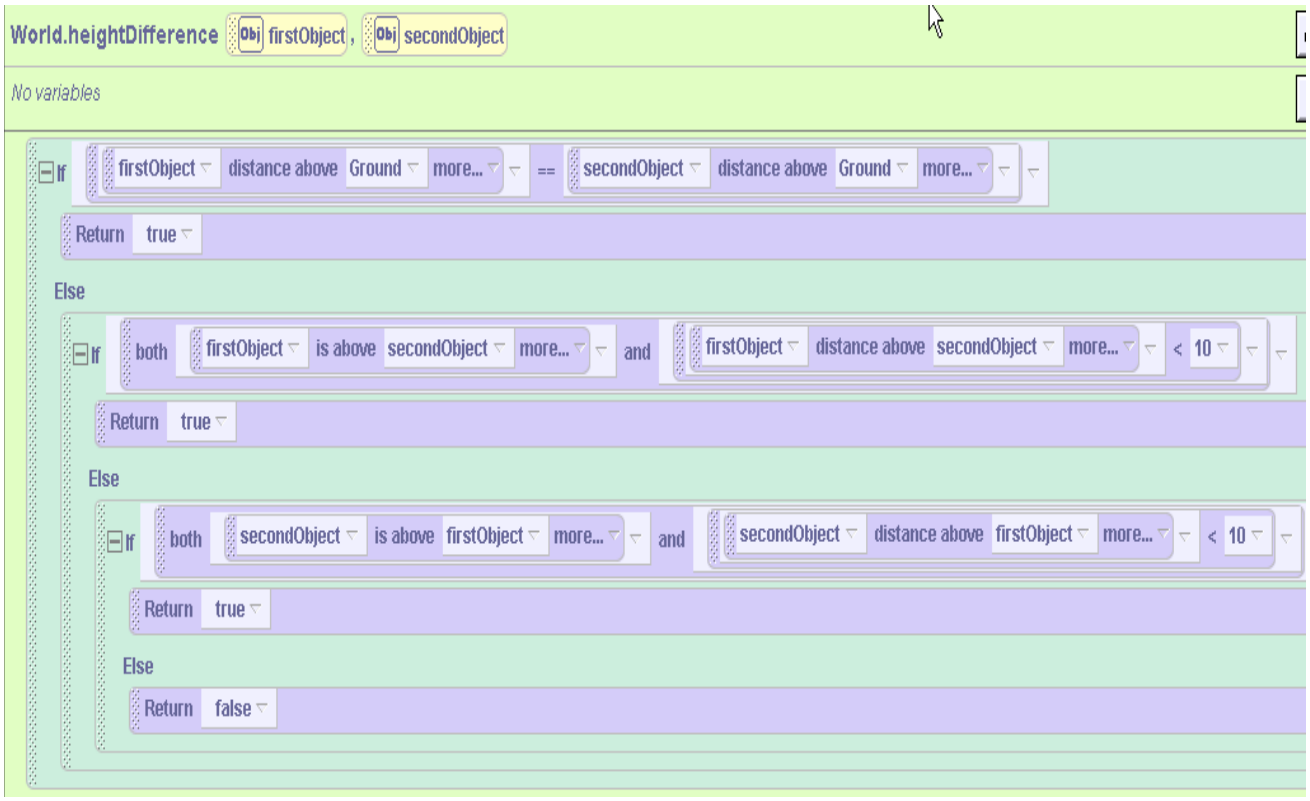


Figure 6-2-9. Cascading style – nested *If* statements

6-2 Exercises

1. Creating a question for the Zeus world

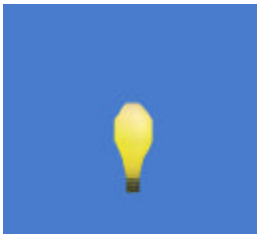
This exercise is a modification of the Zeus world from section 6-1. Create a Boolean question, *isPhilosopher*, that receives the clicked object as a parameter, and returns true if the object is one of the four Greek philosophers, and false otherwise. Then, modify the *shootBolt* method to use the *isPhilosopher* question to determine whether Zeus should shoot a thunderbolt at the clicked object.

2. Switch

Create a world using a Switch object (Controls folder of the gallery). Write a method called *FlipSwitch* and an event/behavior so that when the Switch is clicked, its handle will flip from up to down or down to up. Also, write a user-defined Boolean question *IsHandleUp which* returns true if the handle is up and false if it is down. (*FlipSwitch* will call *IsHandleUp* to deicide whether to turn the handle forward ½ revolution or backwards ½ revolution.) *Hint:* To write *IsHandleUp*, some reference point is needed to test the handle's position. One way to do this is to put an invisible sphere underneath the switch and if the handle is moved down, the sphere should move up and vice versa. (See Tips & Techniques 5 for details on moving an object relative to an invisible object.)

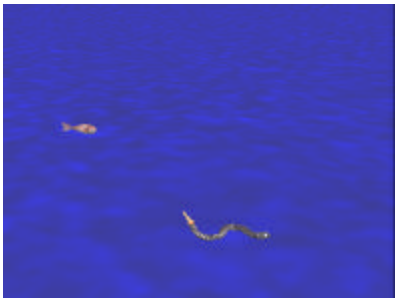
3. LightBulb

Create a world with a *lightbulb* 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 light bulb is off, its emmisive color is black. Write a Boolean user-defined question *IsLightOn* that returns *true* if the light bulb is on and *false* if it is off. When clicked, the *lightbulb* should turn on/off.



4. SnakeOnTheMenu

Create a world that contains a goldfish and a snake in the water. Your initial scene should look something like the image below. The fish is hungry and the snake looks like a good menu item. The goldfish is to move forward a random distance when the user hits the space bar. If the fish is within 1 meter of the snake, the goldfish will eat the snake. Otherwise, the snake will randomly move forward.

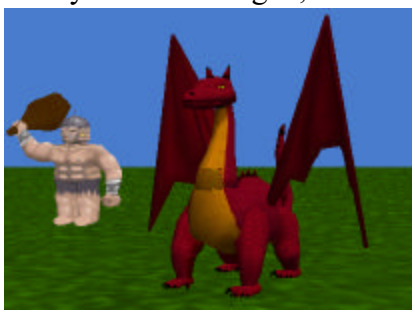


Hints: To make it look like the fish has eaten the snake, make the snake disappear by setting its *isShowing* property to *false*. Try setting the camera's vehicle to the snake to keep the characters from moving out of sight.

5. FrightenAwayTheDragon

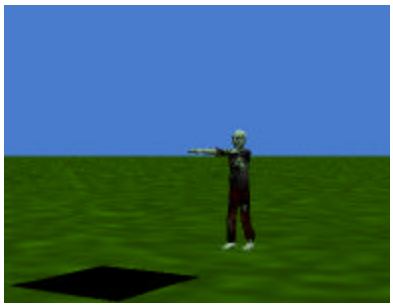
Create an initial scene of a troll and a dragon as shown below. The troll is trying to frighten away the dragon from his favorite hunting grounds. But, the troll is smart enough to not get too close to the dragon. The troll is to rant and rave while moving toward the dragon if the two are more than 5 meters apart. The troll should move toward the dragon every time the space bar is

pressed. Use a user-defined question 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.



6. ZombieWorld

Create a world with a zombie 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 question named *aboveGrave* that returns true if the zombie is within ½ meter of the grave. When the question returns true, make the zombie fall in.



6-3 User-defined Questions II (Number)

Other types of **questions**

As you know, the *type* of a user-defined question is based on the kind of value it returns. Types of questions include number, Boolean, object, and others such as string, color, and sound. In sections 6-1 and 6-2, we wrote user-defined Boolean questions that return logical (true or false) values. Boolean questions (logical expressions) are quite useful in If/Else statements. Let's take a look at how to write user-defined questions that return other kinds of values.

User-defined number question

We have used built-in questions that return a number value. For example, we used the *distance to* question to find the distance between two objects. Questions that return a number value will be useful in many situations. For example, we might want to write a question the returns the number of visible objects on the screen in an arcade-style game. The question would be called to keep any eye on how many objects the user has eliminated (made visible). When only a few objects are left, we might decide to speed-up the game.

It's always helpful to begin with a simple example. To illustrate a number type of question, let's consider a toy ball, as seen in Figure 6-3-2. To simulate rolling the ball sounds like a simple enough idea. (Don't be deceived … this is more challenging than it looks.) Think about how the ball can be made to **roll to its right**. The important part of this action is that the ball should appear to roll along the ground, not just glide forward. (A gliding motion is fine for an ice skater, but a ball should look like it is rolling.) An obvious instruction to try is a *roll* instruction. Surprise – the *roll* instruction simply rotates the ball in place. It spins the ball around, but the ball does not move to its right along the ground!

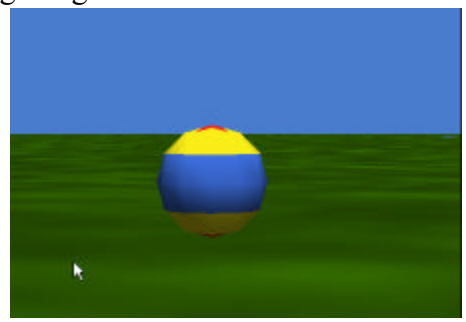
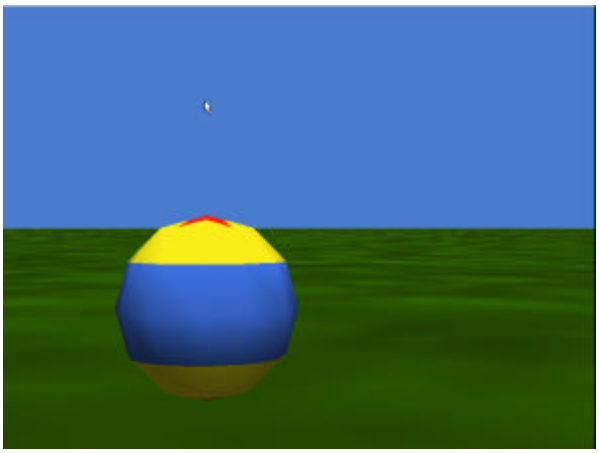


Figure 6-3-2. A toy ball

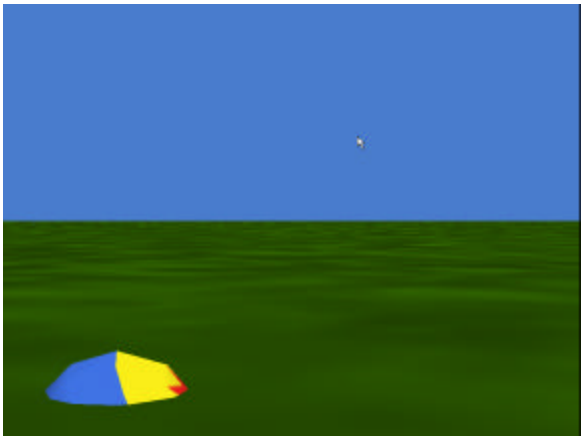
To make the ball actually roll, two actions are needed: the ball must move right and also turn in the same direction. With this in mind, *turn* and *move* instructions are placed in a *Do together* block in Figure 6-3-3.

Figure 6-3-3. Move and turn together instructions

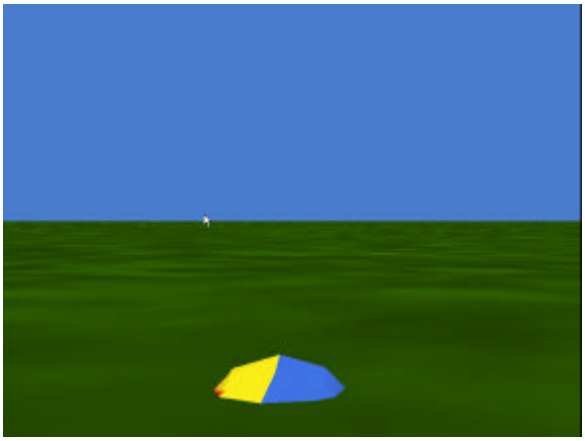
But, testing this code is also disappointing. The effect of the two instructions enclosed in a Do together block is that the ball rolls forward and turns at the same time – but no progress is made in the forward direction. Why is this? Well, when two actions are combined in a *Do together* block, they do not necessarily act in the same way as when the actions occur sequentially. In this example, the forward movement combined with the turn causes the turn to occur in a larger spin area but the ball does not end up moving to its right at all, as can be seen in the sequence of snapshots in Figure 6-3-4.



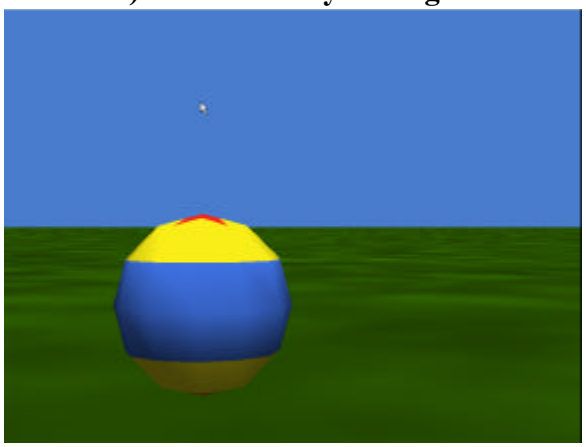
a) Starting position



b) 1/3 of the way through



c) 2/3 the way through



d) Ending position

Figure 6-3-4. Move and turn together sequence of snapshots

One solution to this problem is "*asSeenBy*". The desired effect is to have the ball move along the ground. So, the ground seems a likely object to use as a reference for "*asSeenBy*." An example of the code is shown in Figure 6-3-5. This code moves and turns the ball forward 1 meter in a pleasing simulation of a real-life ball rolling on the ground.

Figure 6-3-5. Ball moves *asSeenBy* the Ground

The ball was made to move forward only one meter. Suppose the ball is to move forward 10 meters. We must now think about how many revolutions the ball needs to turn so as to cover a distance of 10 meters in a forward direction. Creating a realistic rolling motion that covers a given distance is challenging because the number of times the ball needs to turn is proportional to the diameter of the ball. To cover the same forward distance along the ground, a small ball turns forward many more times than a larger ball. In Figure 6-3-6, the larger ball covers the same distance in one revolution as the smaller ball covers in four revolutions.

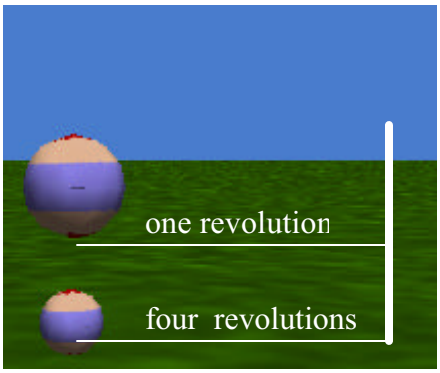


Figure 6-3-6. Distance covered by a revolution is proportional to diameter

Of course, the number of revolutions needed for the ball to roll 1 meter to its right could be found by trial and error. Or, the number of revolutions could be hand calculated, using the formula:

number of revolutions = distance / (diameter * Π)

But, every time the ball is resized or the distance the ball is to roll changes, this computation would have to be done again and the code would have to be modified. **This is where a number type question would be helpful.** Since we are only concerned with the toy ball rolling and no other objects are involved, a character-level question is appropriate. (A character-level question has the advantage that if we wish to calculate the amount the ball needs to roll in some other world, we can save out the toy ball character and then reuse our toy ball in future worlds.) Begin by selecting the *toyball* object from the object tree, and selecting the "*create new question*" tile in the questions tab. In the New Question box, the name *howManyRotations* is entered and the type *Number* is selected, as shown in Figure 6-3-7.

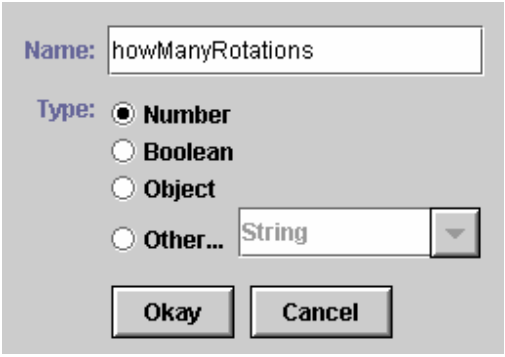


Figure 6-3-7. Naming and selecting the type for a number question

The code shown in Figure 6-3-8 illustrates the question *toyball.howManyRotations*. The code implements the question: "How many rotations does the Ball have to make to move a given distance along the ground?" The parameter is the *distance* the ball is to move. The number of rotations is computed by dividing the *distance* the ball is to move by the product of the ball's diameter (*toyball*'s width, a built-in question) and pi (3.14). The computed value is the answer to the question. The *Return* statement tells Alice to send back the computed answer.

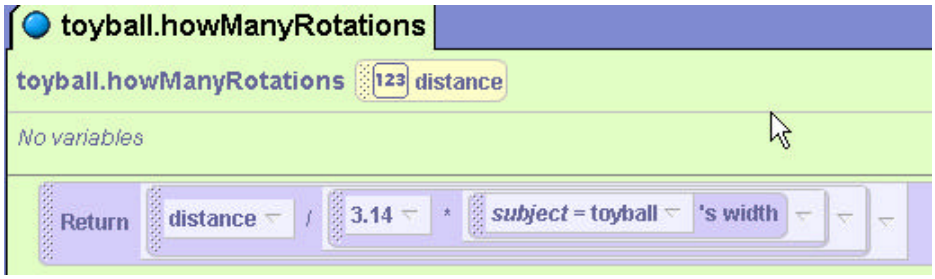


Figure 6-3-8. *toyball.howManyRotations* question

In number type questions, the order of evaluation of the values in the question must be carefully arranged. Alice uses nested tiles in the same way we would write a mathematical computation using parentheses. The expression on the most deeply nested tile, "3.14 * subject = toyball's width," will be computed first. Then the distance will be divided by that computed result.

Testing

Now that the *toyball* has a question named *howManyRotations*, the question can be used in a program. In Figure 6-3-9, a sample test is shown. An (arbitrary) distance of 3 meters is used for the move forward instruction and to provide the distance parameter for the *toyball.howManyRotations* question. This test should be repeated with low distance values (for example, -2 and 0) and also with high distance values (for example, 100). Using a range of values will reassure you that your question works on many different values.

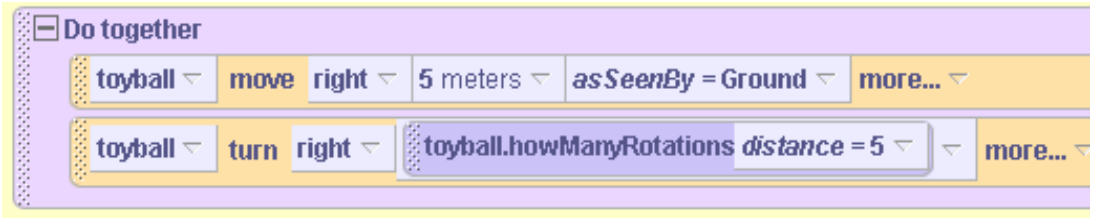


Figure 6-3-9. Testing *howManyRotations* Question

Using a question with a relational operator

If the distance is small and the ball is large, the ball may not turn a full revolution – and then **it would not look like it was rolling at all**. How can we force the ball to turn right at least one revolution? One solution is to use an *If* statement to check the value of *howManyRotations*. If *howManyRotations* is less than one, make the ball turn a complete revolution. Otherwise, the ball will turn *howManyRotations*, as above. Figure 6-3-10 shows the modified code using a call to toyball.howManyRotations as part of a logical expression. The *less than* relational operator is

used to compare the value returned by *howManyRotations* to one. The result of this comparision is true or false and determines whether the toyball makes 1 revolution or several revolutions.

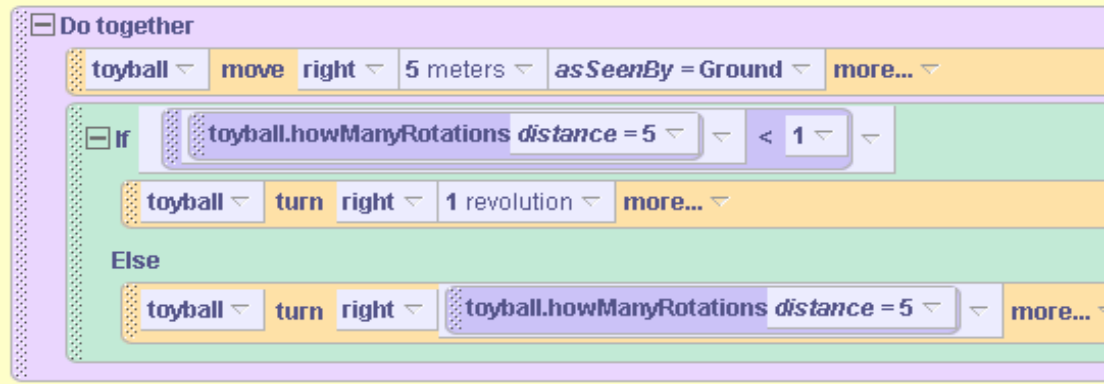


Figure 6-3-10. Calling a question as part of a logical expression

Abstracting a character-level method

The discussion above has led to the development of a very realistic rolling ball action. Of course, the rolling action is composed of several small steps and the calculation of an answer to a question. In building a larger world where the rolling ball is only a small part of many actions allows you to think about these actions as one overall action: "a realistic roll of the ball." We can abstract this action by putting the code into a method and giving it the name *realisticRoll*. Only the ball is involved in *realisticRoll*, so this will be a character-level method. The code for *realisticRoll* is shown in Figure 6-3-11. Note that *toyball.realisticRoll* has a parameter *howFarToRoll* and calls the question *toyballlhowManyRotations* in the same way as in the testing statements used in Figures 6-3-9 and 6-3-10.

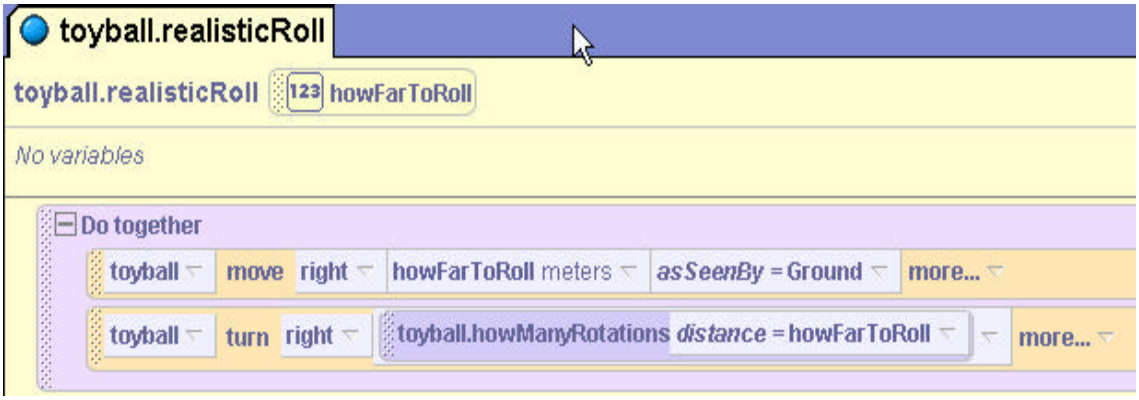


Figure 6-3-10. Character-level method toyball.realisticRoll

World-level question for generic use

In the toyball example, a character-level question and a character-level method were developed. But, it is clear that the question to compute the number of rotations for a ball might be used on any kind of spherically-shaped object. This brings up one very good reason for writing a world-level question: a question that could be asked of any object having similar properties. To write a generic kind of question similar to *howManyRotations* for any spherically-shaped object, two parameters are needed: the *distance to roll* the spherical object and the *diameter of the sphere*. The reason for distance to roll is the same as it was for the *toyball.howManyRotations* method. But, the reason for the diameter parameter is that this method is intended to be generic (work for

any spherical object). So, the diameter of the spherical object must be sent in for the computation of the number of times the sphere is to rotate. Otherwise, Alice would not know which spherical object's diameter to use for the computation.

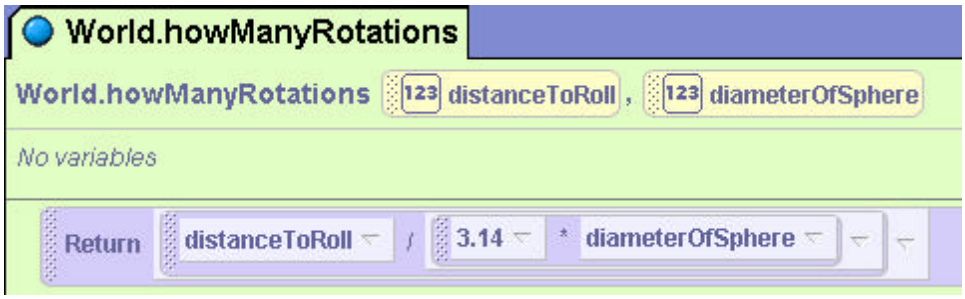
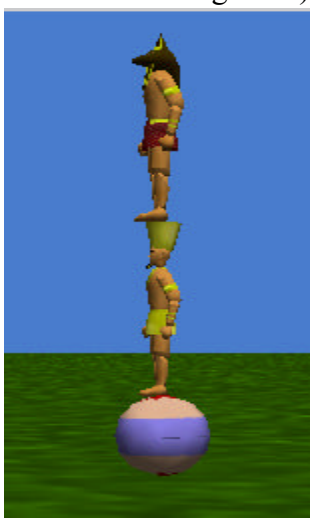


Figure 6-3-11. Generic world-level question

6-3 Exercises

1. AcrobatsWithRollingBall

Begin by creating a world containing the *toyball* (resized to twice its original size) and two characters/acrobats of your own choosing. Position the acrobats on top of the ball. **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 a one-shot instruction on each acrobat and on the ball to orient each object to the center of the world.** (See Tips & Techniques 3. Using *orient to* will ensure the objects are synchronized for movement together.)



Create an animated 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 to balance!

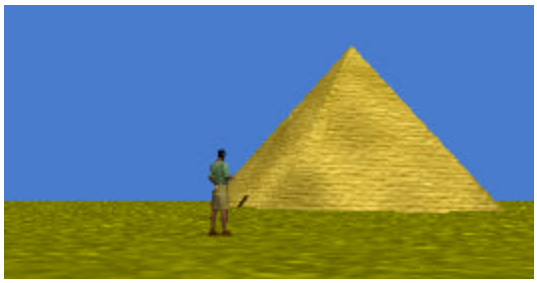
2. BeeScout

This exercise is a variation on the bee scout animation presented in section 6-2. It has been a hot, dry summer and a hive of bees is in desperate need of a new supply of pollen. One bee 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 flat on the ground and colored blue to look like a pond. Add plants, trees, and other natural scenery including some flowers. Be sure the bee is located somewhere around the edge of the pond, as shown in the screen shot below.

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 make 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 * the diameter of the circle. PI is 3.14 and the diameter is the object's width. Write a question 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 question while turning left one revolution.

3. PyramidClimb

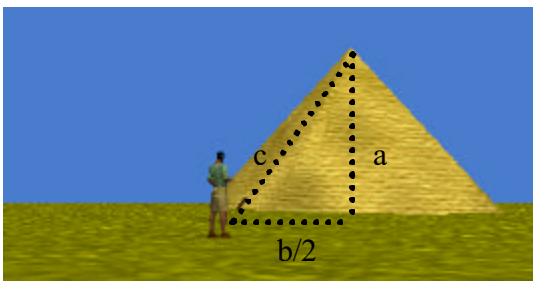
On spring break, a student is visiting the land of the Pharoahs. The student decides to climb one of the pyramids. He/She will start at the bottom and move straight up the side. Set up an initial scene consisting of a person and a pyramid, as shown in the screen shot below. Write a method to animate the *climb* of the adventuresome student up the side of the pyramid so the person's feet are always on the side of the pyramid.



Prepare the person for climbing the pyramid 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.) While the person is climbing the pyramid have the person in a leaning position. After reaching the top, the person should stand up straight.

To determine how far the person must move to climb up the side of the pyramid, the *climb* method must call a question. The question 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 ($a^2 + b^2 = c^2$). Actually, the value that is needed is the value of c, which 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:

length of the pyramid's side = v((pyramid's height) ^{2} + (pyramid's width/2) ^{2})



6 Summary

This chapter introduced the fundamental concepts of *If* statements and user-defined questions (functions). The *If* statement plays a major role in most programming languages as it allows for the conditional execution of a segment of code. The **key component** of an *If* statement is a Boolean condition that returns a true or false value. *Boolean conditions* are also referred to as *logical expressions*. A Boolean condition is used in an *If* statement 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 segment of our program code.

To demonstrate the flexibility of Boolean conditions, we started with a simple condition that called a built-in question. Then, we used logical operators to build much more complicated Boolean conditions.

Built-in questions do not always meet the particular needs of a program. In this chapter, we looked at how to write our own user-defined question that returns a Boolean value (true or false). Then, other types of user-defined questions were introduced. The benefit of writing our own user-defined questions is that we can think about the task on a higher plane – a form of abstraction. User-defined questions that compute and return a number value make writing code much cleaner because the computation is hidden in a question, rather than cluttering up the method where the result of the calculation is needed. By using parameters in user-defined questions, we can make the questions generic to use the questions with different kinds of objects. Character-level questions can be defined and saved with the object to allow us to reuse the question for that object 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 segment of code.
- An *If* statement contains a Boolean condition (logical expression) 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 of the *If* statement gets executed.
- Boolean conditions may call built-in questions that return a true or false value.
- Logical operators (and, or, not) can be used to combine simple Boolean conditions into more complex expressions.
- Relational operators (< , > , >= , <= , = =) can be used to compare values in a Boolean expression.
- User-defined questions can be written to return a Boolean value and used in If statements.
- User-defined questions can also be written to compute and return other types of values.

6 Projects

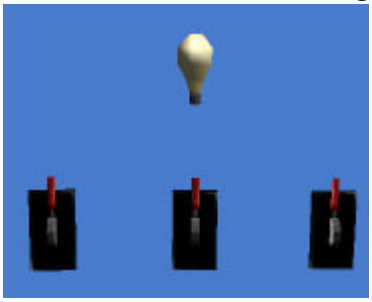
1. Gatekeeper

Build a world with any four different characters of your choice (people, shapes, vehicles, etc.). Position the four objects in a lineup. The characters are facing the player and are spaced equally apart from one another.

In this game, one of the characters 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 in the position on the far right of the lineup. (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 character 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 programmer-defined Boolean question that returns true when the objects in the lineup 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. BinaryCodeGame

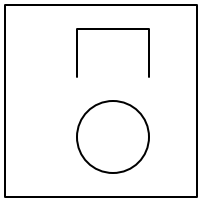
Build a world with three *switches* and a *lightbulb*, as seen below. Beneath the lever on each switch put an invisible sphere. Set the emissive color of the *lightbulb* to *black* (turned-off).



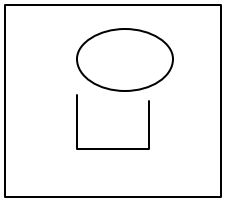
In this game, the positions of the levers on the switches represent a binary code. When a lever is up, the lever represents 1 (electric current in the switch is high) and when a lever is down, the lever represents 0 (electric current in the switch is low). In the above world, all three levers are up so the binary code would be `111`. The correct binary code is randomly chosen at the beginning of the game. (Use the world-level *random number* question.) The idea of this game is to have the user try to guess the correct binary code that will light up the lightbulb (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 the handle of the lever 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 *lightbulb* will turn on.

Each switch should respond to a mouse-click on a lever. If the lever is down, flip it up. If the lever is up, flip it down. To track the current position of the lever on a switch, an invisible sphere can be placed on the switch and moved in the opposite direction as the lever each time the lever is moved. When the sphere is below the lever, the lever is in an up position. When the sphere is above the lever, the lever is in a down position, as shown below. At the same time the lever changes position, the sphere should also move. That is, as the lever moves up the sphere moves down and vice-versa.



Lever up, Sphere below Lever



Lever down, Sphere above Lever

Your project code must include a Boolean question that determines whether a switch lever is in the up position. (Use an object parameter that specifies which switch is to be checked.) Also, include a Boolean question that determines whether the Boolean code is correct. *Hint:* use the color of the spheres (even though they are invisible) as a flag that indicates the correct position of the lever.

3. DrivingTest

Create a world that simulates a driving test. The world should have a car, 5 cones, and a gate. 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 object 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 car forward, left, or right to swerve around each of the five cones so as to avoid hitting a cone. 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 made visible. Write a user-defined question named *tooClose* that checks the car's distance to a cone. If the car is within 2 meters of the cone, the question returns *true*. It returns *false* otherwise. Also, write a question named *passedTest* which evaluates if the user has passed the test. This is recognized by the fact that the car has been driven past the gate.

Hints: 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. PhishyMove

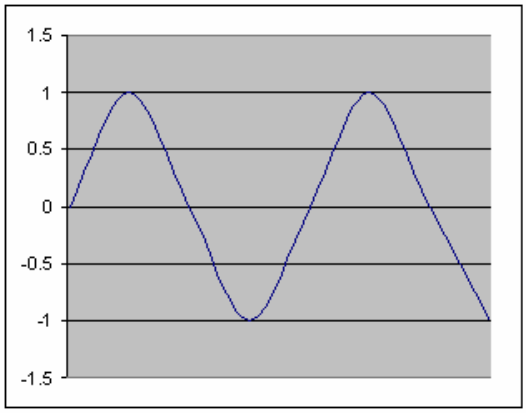
Phishy fish has just signed up at swim school to learn the latest motion, the Sine Wave. Your task is to write a method to teach her the *sineWave* motion. The initial scene with a fish and the ground modified to look like water is seen below.

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 one-shot instructions to move the fish to the world origin (0,0,0) and then turn the fish right ¼ revolution. Because Phishy is partially submerged in the water, set the opacity of the water (ground) to 30% so the object can be seen in the water. Now, use camera controls to re-position 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 Phishy 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* question that can be used to teach Phishy the sineWave motion. (The sine question/function is often used to determine the relationship between the lengths of the sides of a right triangle. For the purposes of this animation, the relationship of the lengths of the sides of a right triangle is not really important.) If the sine function is computed over all angles in a full circle, the sine value starts at 0, and 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, like so:



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. Of course this up-and-down motion is occurring at the same time as she moves to the right in the water. The Alice sine question expects to receive an angle expressed in radians (rather than degrees). Write a question, named *degreesToRadians*, that will convert the angle in degrees to the angle in radians. To convert from degrees to radians, multiply the angle degrees by PI and divide by 180. The *degreesToRadians* question 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 *moveTo* instructions (see the Tips and Techniques section of this chapter). A *moveTo* 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)*. Use *moveTo* instructions for angles: 45, 90, 135, 180, 225, 270, 315, and 360. For a smoother animation, make each *moveTo* instruction have *style = abruptly*.

5. CosineWave

Teach Phishy how to move in a cosine wave pattern, instead of the sine wave pattern as described in project 4 above.

Tips & Techniques 6

If/Else and visibility as a condition

In game-type programs, it is frequently the case that objects are made invisible. Look at the futuristic space scene below where a spaceship will "cloak" to hide from an alien spacecraft. Cloak is a science fiction term describing the ability of a spaceship to camouflage itself into the celestial sky so it is not visible or detectable on an enemy radar screen. Figures T-6-1(a) and (b) illustrate the space scene before and after the spaceship cloaks.

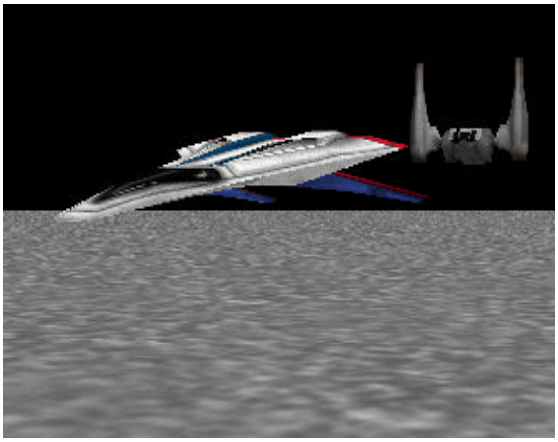
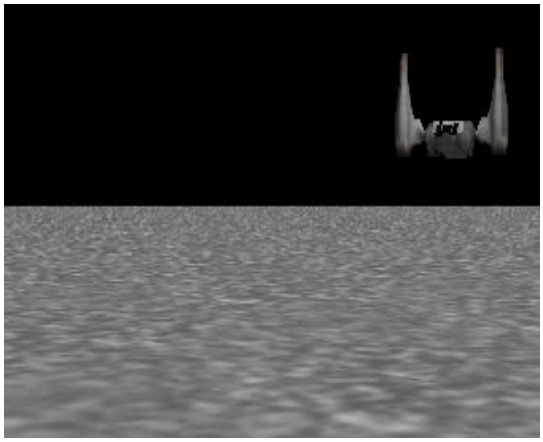


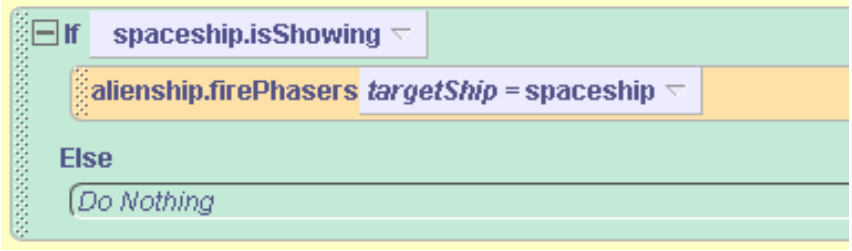
Figure T-6-1(a) Space scene before Cloaking



(b) Space scene after Cloaking

As was explained in Tips & Techniques 4, Alice offers two ways to make an object invisible: (1) **set *isShowing* to false**, and (2) **set *opacity* to 0 %**. In this space world, the code to cloak the space ship might look like this:

In worlds where visibility is used as part of the animation, a convenient programming technique is to use the visibility of an object as a Boolean condition in an *If* statement. That is, we might write something like:



What is not so obvious about the use of visibility in a Boolean expression is that your code must be consistent in its use of either *isShowing* or *opacity*. What we mean by this is: if you use

isShowing to change the visibility of an object, then use i*sShowing* in the Boolean condition. Or, if you use *opacity* to change the visibility of an object, then use opacity to change the visibility of an object. Why is this? The answer is that *isShowing* and *opacity* are two different properties that track different (though related) states of the object. *isShowing* is strictly true or false – kind of like a light switch that can be either on or off. But, opacity is a more sliding scale kind of property expressed in percentages – kind of like a dimmer switch that can adjust the brightness of a light. Though it is true that when opacity is 0% the object is invisible, when you make an object have an *opacity* of 0%, Alice does not automatically make *isShowing* false. Likewise, when you make *isShowing* false, Alice does not automatically make *opacity* 0%.

The moral of this story is: If you use isShowing to make it invisible, use isShowing to check its visibility. And, if you use opacity to make an object invisible, use opacity to check its visibility. "And never the twain shall meet."

Camera: View from the back

Figure T-6-2 illustrates a fantasy world where objects have been added to the scene and rearranged somewhat to create an initial scene for an animation. From our perspective (as the person viewing the scene), the camera is allowing us to look at the scene "from the front."

Figure T-6-2. Fantasy scene

In some worlds, you may want to move the camera around in the initial scene so it is viewed "from the back." The problem is: how do we get the camera to "turn around" so the back of the scene is in view? One technique that seems to work well is to position the mouse cursor over the camera's *forward control*, hold down the mouse and drag it forward. (The forward control is circled in Figure T-6-3). Continue to hold down the forward control and allow the camera and move straight forward until the camera seems to move right straight forward through the scene.

Figure T-6-3. Forward camera control

When the camera seems to have moved to the other side of the scene, let go of the forward control. Select the Camera in the Object tree and use a one-shot instruction to turn the camera ½ revolution, as in Figure T-6-4.

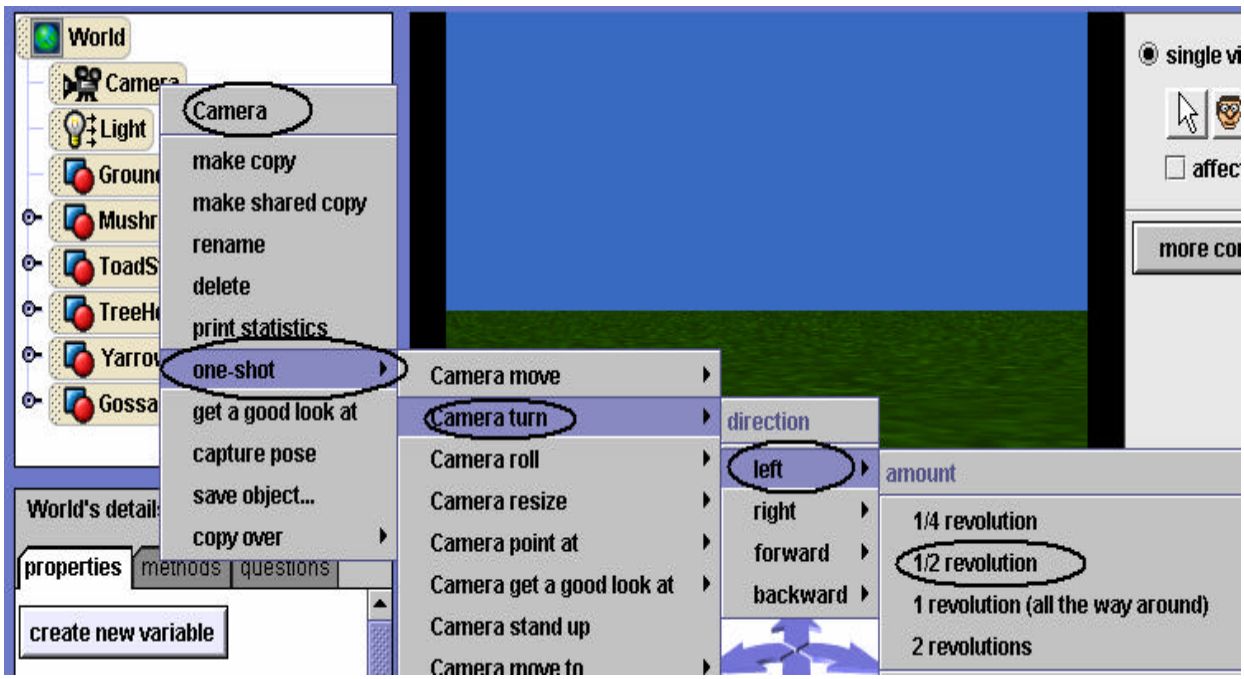


Figure T-6-4. A one-shot instruction to turn the camera around

The camera should now be facing in the opposite direction and you should be able to see the scene from the back, as illustrated in Figure T-6-4.

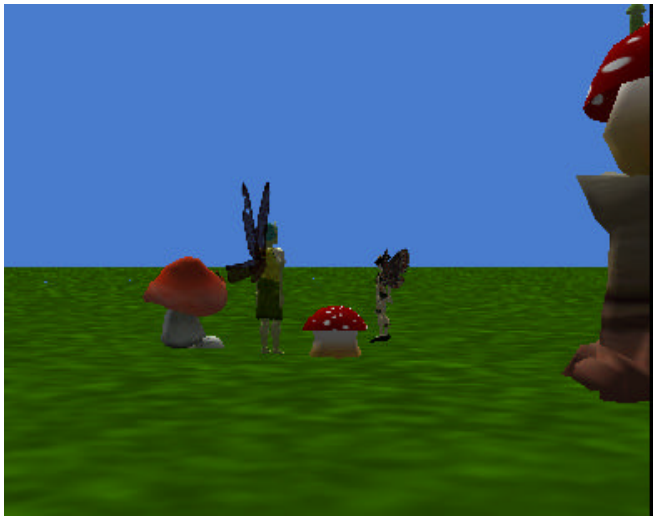


Figure T-6-5. View from the back of the scene

Lighting up the rear view

As you can see in Figure T-6-5, the rear view of the scene is somewhat disappointing. The scene looks rather dull because the built-in light object is shining so as to light up the scene from the initial camera point of view. To improve the lighting for a rear-view screen capture, you can add a light bulb to the scene – drag it in from the Lights folder in the web gallery, as shown in Figure T-6-6. Wow – what a difference!

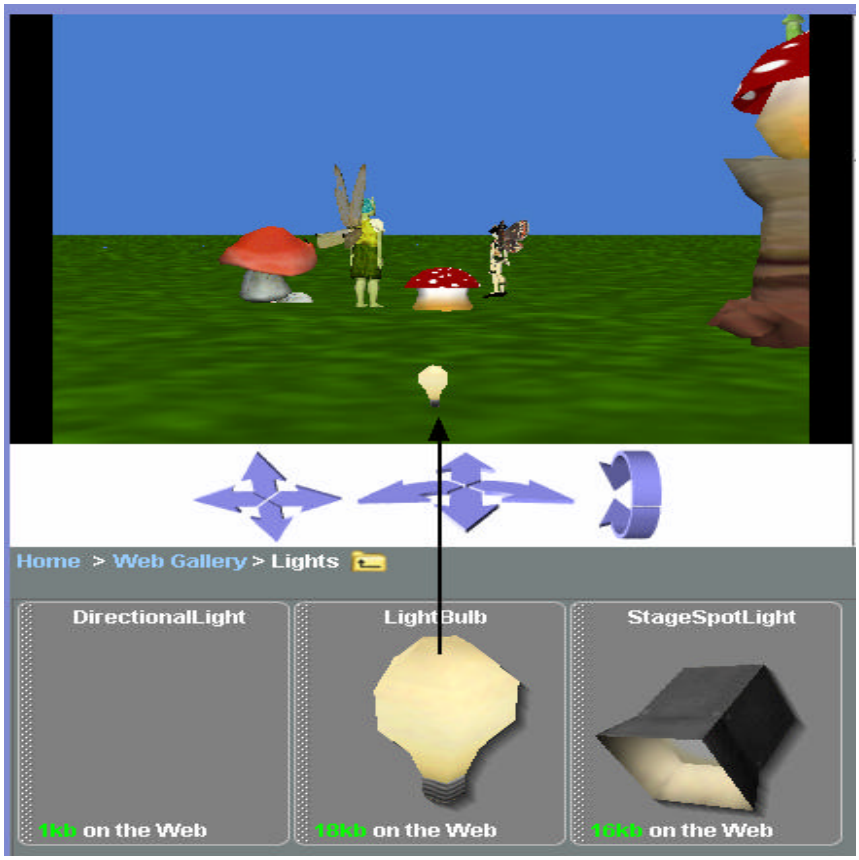


Figure T-6-6. Lighting added to rear view

You probably don't want a light bulb in the middle of your scene. Use a one-shot instruction to move the light bulb up 10 meters. The light bulb will be out of sight, but the scene will now have ambient light from both front and back views.