Fluency with Alice

Workbook to accompany Snyder's *Fluency with Information Technology, 4th Edition*

by Robert Seidman, Philip Funk, Jim Isaak, Lundy Lewis

Chapter 4. Game-time with Alice

"All things are ready, if our minds be so." - Henry the V. Act IV, Scene 3 - William Shakespeare.

4.1: Introduction to the Quest

In the previous chapters you learned the fundamentals of Alice programming by creating a movie of a play using a story board. There was minimal user interaction – just enough to branch the program to either save or doom our friendly penguin.

Also previously, the camera's position was in front of the stage. Even though it was able to move in for a close-up, the camera still shot from in front of the scene and did not, for example, view the scene from the point of view of one of our players (i.e., through a player's eyes).

All of this needs to change in a game environment. We want our game scene setup to be something like a theatre in the round. As an example, launch Alice and in the Welcome to Alice! window that opens, click on the Examples tab and open the **amusementPark** world. When the world loads, the camera will be facing a carousel in the amusement park. (See **Figure 4-1**) Click the Play button. Press and hold either the right or left keyboard arrow and watch as the **camera** swivels to see other parts of the amusement park. You can also press the up-arrow key and the down-arrow key to move the camera forward and backward. The point is that an Alice world is not just what you see from moment to moment in front of the camera. The world also exists above, below, to the left, to the right, and behind what the camera sees. The camera is a view into the world from a particular point of view (i.e., position + orientation).

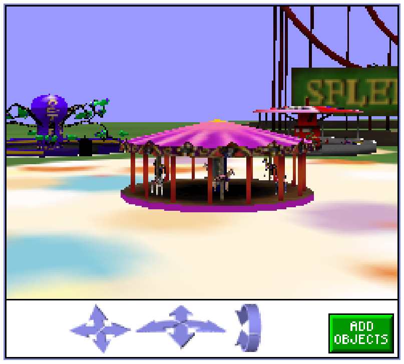


Figure 4-1 amusementPark

When you are finished exploring the **amusementPark** world, exit from Alice without saving the world.

In a game, we want to see the world from a character's point of view. This means that the camera ought to be the eyes of the character and move with the character everywhere. This is called "first person" perspective (aka, "first person shooter"). Also, the user of the program will want, indeed need, to move some of the characters around the scene (i.e., personally control character movement). We can do all of these things with Alice.

But first we need to know the goal of the game. Ours is a quest. The protagonist searches within the world in order to find and maybe keep hidden treasure and, of course, be surprised by hidden evils. We will also want to incorporate sound and pictures, known as billboards, into the game space that is our world.

We have provided many exercises for this Chapter. They are, for the most part, our suggestions for extending the game fun. You will undoubtedly think up many more. Let the games begin!

4.2: Setting the scene

You can see the results of some of what you will create in this chapter: {**Video demo** (sound on): Video 4-1}

We start out by setting the scene to look like what you see in **Figure 4-2**.

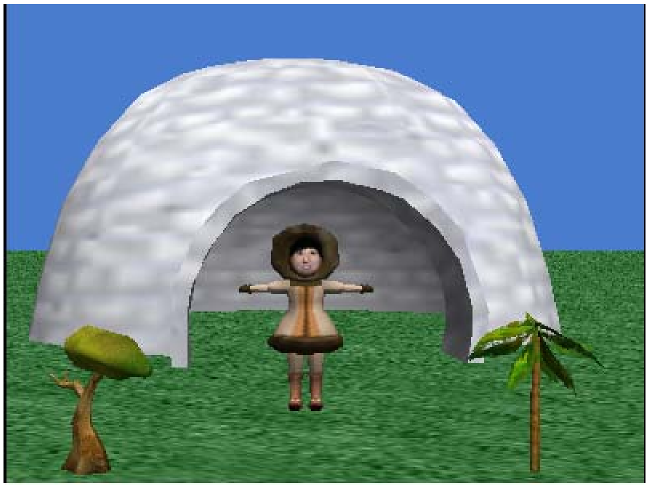


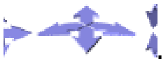
Figure 4-2 Initial quest game world setup.

The game – In this game you use your mouse and certain keys to move a character, Kirima, around until she finds a fairy princess who is initially invisible. When found, the fairy princess materializes, latches onto Kirima and goes wherever Kirima goes. There is a secret entrance to a National Park that is revealed when the fairy princess is found. You will incorporate sound and various camera angles, one of which looks out from Kirima's eyes.

Launch Alice. Close the Welcome to Alice! window. In the menu bar, click File/New World. Use the grass template. Click the ADD OBJECTS button and go to the Local Gallery and find the Buildings category tile. Open it. Click on the Igloo class and then on the Add instance to world button. Leave the igloo in its default position. Rename it to "icePalace."

Build your own character – You can build your own people in Alice. In the People category, scroll to the far right and find the hebuilder and shebuilder classes. An advantage of these classes is that objects created implement a walk method. Peter has a walk method.

Next, go up one level in the Local Gallery and find the People category tile. Open it. Click on the EskimoGirl class and then on Add Instance to World. Rename her "kirima."

The default position of **kirima** is in front of the **icePalace**. Pull the **camera** straight back a bit to show more of the scene. Use the bottom arrow in the center **camera** control. The center **camera** control looks like this: .

Now, go up one level in the Local Gallery and find the Nature category tile and place two trees, HappyTree and PalmTree, into the world. We will not bother renaming them. Resize and position the two trees well forward of **kirima** as shown as shown in **Figure 4-2**.

We realize that the scene you are setting is a bit incongruous – an igloo on grass with trees. But with Alice, we have the ability to create virtual fantasy worlds that may not exist in reality - or at least not in the reality we are used to.

The final scene setting is to alter the sky. Go up one level in the Local Gallery and find the Environments category tile. Click on the Skies folder and then the DesertSky class. Add instance to world. See **Figure 4-3**.

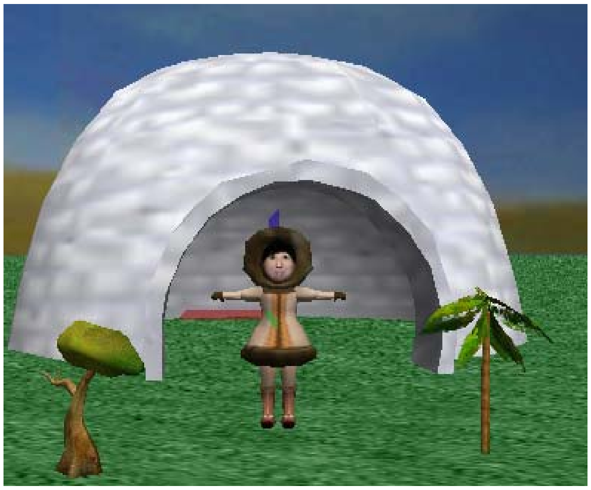


Figure 4-3 The scene with desertSky.

Experiment - If you choose ArcticLandscapeAlaskaMountains for your sky, the grass changes to snow. Go ahead and try it. Use Undo afterwards.

For reasons that will become more apparent later, place a dummy **camera** marker right where the **camera** is currently located. This is so we can relocate the **camera** back to this point of view at any time in the game.

Here's how to do it. In the upper-right-corner of the ADD OBJECTS screen, click on the more controls >> button. You will see **Figure 4-4a** (shown on the next page). Click on the drop dummy at camera button. Notice the Dummy Objects folder that appears in the **Object Tree** panel. Click on the **+** to the left of the Dummy Objects folder to open it. See the result in **Figure 4-4b**. Rename the **Dummy** object to "wideCamera." See the result in **Figure 4-4c.**

"More than one way to skin a cat" - In Chapter 1, we accomplished a similar thing using Cones. The distinction is that Dummy Objects can only be placed where an object already exists.

For similar reasons, let's mark the spot where **kirima** now stands so that we can easily return her to this spot later during the game. After all, she will wander the world and may wish to return to her starting point.

Here's how to do it. Click on **kirima** in the either the **Object Tree** panel or the **World View** panel. Then click on the drop dummy at selected object button. There will be an addition to the Dummy Objects folder in the **Object Tree** panel. Click on the **+** of Dummy Objects to open the folder. You will see a new **Dummy** object as shown in **Figure 4-5**.

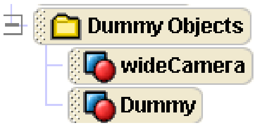


Figure 4-5 Another Dummy object.

Rename this new **Dummy** object to "kirimaOriginalPosition." We will make use of this **camera** position later.

Click the fewer controls << button to close the controls window. Click the DONE button to close the ADD OBJECTS window.

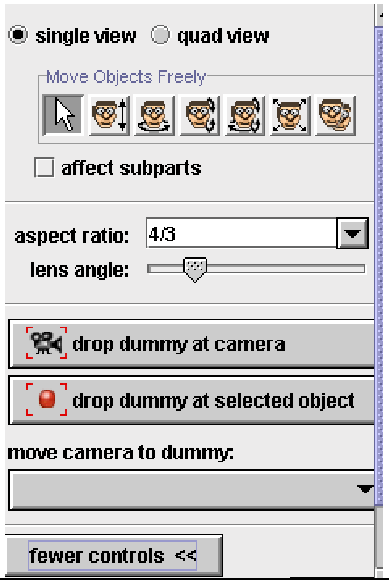


Figure 4-4a More controls.

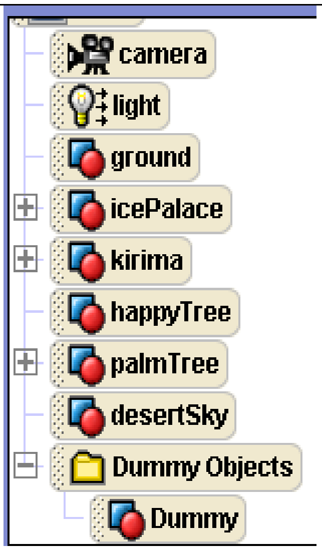


Figure 4-4b Dummy.

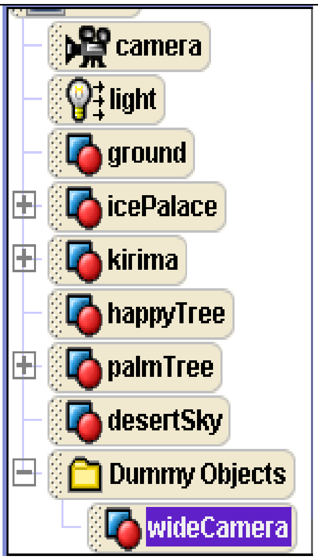
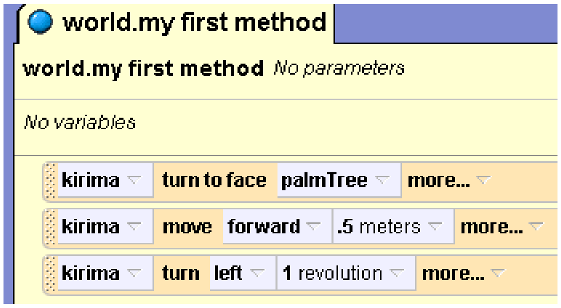


Figure 4-4c Dummy renamed to wideCamera.

4.3: User control of the actors

We want to start the game by having **kirima** take a few steps forward, turn to face the **palmTree,** and then turn (i.e., swivel) 1 complete revolution. In the **Method Editor** panel, click on the **world**.*my first method* tab to be sure it is open. In the **Object Tree** panel, click **kirima**. Then click on the method*s* tab in the **Details** panel. Drag the three instructions you see in **Figure 4-6** into **world**.*my first method Do Nothing* spot.

Play your world. How does it look?



world.my first method

world.my first method *No parameters*

No variables

kirima turn to face palmTree more...
kirima move forward .5 meters more...
kirima turn left 1 revolution more...

Figure 4-6 Opening moves for kirima.

User control of object movement using arrow keys

We can give the user control over **kirima's** movement. To do this we need to add an event. In the **Events** panel, click the create new event button. From the drop-down list select Let the arrow keys move <subject>. See **Figure 4-7a.** You can see the result in **Figure 4-7b**.

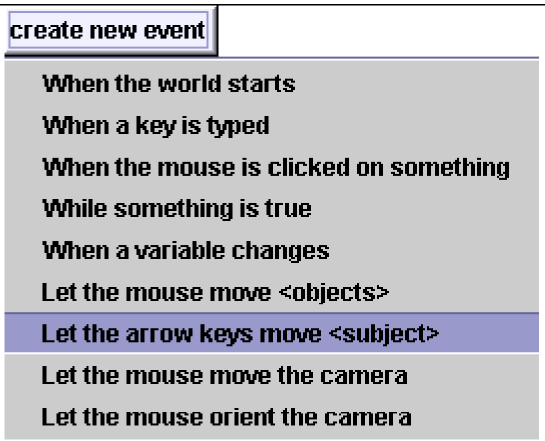

create new event
When the world starts
When a key is typed
When the mouse is clicked on something
While something is true
When a variable changes
Let the mouse move <objects>
Let the arrow keys move <subject>
Let the mouse move the camera
Let the mouse orient the camera

Figure 4-7a Create a new event.

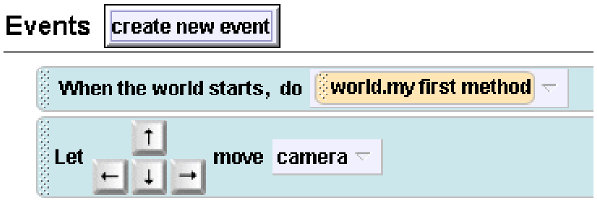


Figure 4-7b Let the arrow keys move <object>.

Note that **camera** is the default subject. Click **camera**, shown **Figure 4-7b,** and choose **kirima**/the entire kirima. See **Figure 4-8**. Save your world. Play the world and use the arrow keys to control **kirima's** movements.

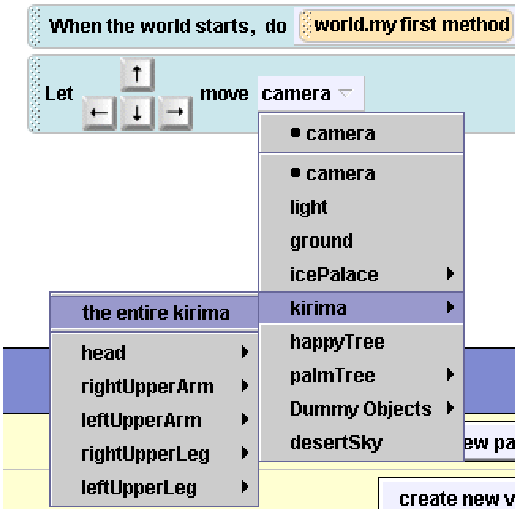
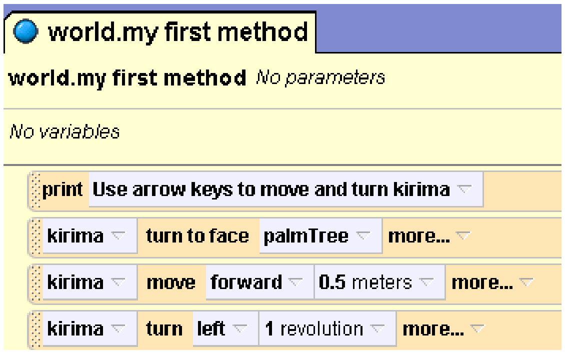


Figure 4-8 Setting arrow controls for kirima.

Events have two components:
An event trigger is a condition that causes an event handler to execute.
An event handler consists of instructions that respond to the event trigger.
For example, look at Figure 4-7b. When the world starts is the event trigger and world.my first method is its event handler.

We want to be sure that human players watching the video know which keys control which **kirima** actions. Drag the print control from the bottom of the **Method Editor** panel and place it as the first instruction in the **world**.*my first method*. Choose *text string* and type: "Use arrow keys to move and turn kirima." See this in **Figure 4-9**. Click the Play button and you should see the print text at the bottom of the animation window. {**Video demo:** Video 4-2}

Explore - Imagine a similar user arrow key option for peter in Chapter 3 when ollie is breathing fire at him. By responding to arrow key events, peter could dodge the flames. This is left as a Chapter 4 exercise.



```
print Use arrow keys to move and turn kirima
kirima turn to face palmTree more...
kirima move forward 0.5 meters more...
kirima turn left 1 revolution more...
```

Figure 4-9 Using the print control to put text at bottom of animation window.

Important note about object and viewer perspectives

After **kirima** goes through her initial movements, experiment with the four arrow keys on your keyboard. Note that the up-arrow key moves **kirima** forward with respect to her point of view. The down-arrow key moves her backward with respect to her point of view. The right-arrow key turns her clockwise with respect to where she is facing. The left-arrow key turns her counter-clockwise with respect to where she is facing. Objects are axis egocentric. They view the world and move relative to where they are placed and facing (i.e., point of view), and NOT with respect to the way YOU are facing.

Example: If **kirima** is facing you (i.e., she is looking out at you from out of the screen), you might think that pressing the up-arrow key would move her away from you (i.e., backwards with respect to where she is facing). Instead, the up-arrow key moves her toward you, which is forward relative to the way she is facing. That is an example of what object axis egocentric means.

User Mouse Control

Arrow key control of **kirima** is fine but kind of slow. What would it be like for the user to click on any object in the scene to cause **kirima** to move to that object? That might be faster.

To do this, click on the create new event button and select the When the mouse is clicked on something event trigger. See **Figure 4-10a. Figure 4-10b** shows the resulting event.

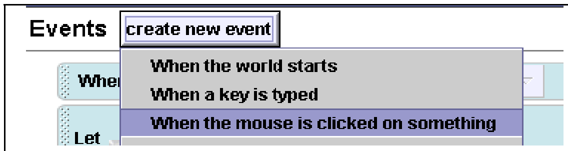


Figure 4-10a When the mouse is clicked on something event trigger.

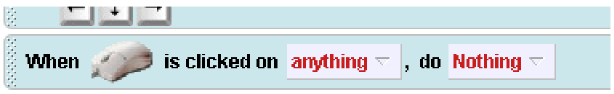


Figure 4-10b Resulting event.

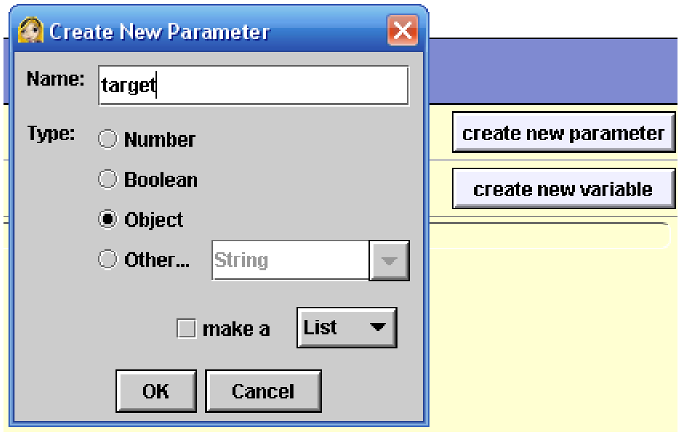
Now we need to create an event handler method to move **kirima** to any object clicked by the mouse cursor and put that method into the *Do Nothing* spot in the event we just created, shown in **Figure 4-10b.**

When the cursor is over any object and that object is clicked, we want **kirima** to turn to face the object and move forward a certain number of steps. While **kirima** has not yet reached that object, we want **kirima** to continue moving forward toward the object until she is within a certain threshold (i.e., distance) away from it. Does this sound familiar? This is what **robin** did to get closer to **peter** in Chapter 3.

Let's create a method to accomplish this task. Later, we will create a second method that works differently but accomplishes the same thing to illustrate an alternative algorithm.

Create kirima Cursor Search Method A

In the **Object Tree** panel, click on **kirima**. Be sure that the methods tab in the **Details** panel is active by clicking on it. Click on the create new method button and name the new method, *cursorSearchA*. In that method's **Method Editor** panel, click on the create new parameter button and click on the Object radio button. Name the parameter, *target,* and click on the OK button to close the window. The target parameter will allow us to pass the object name that is underneath the mouse cursor into the method itself so **kirima** will know which object to move to. See **Figure 4-11**. The result is shown in **Figure 4-12**.



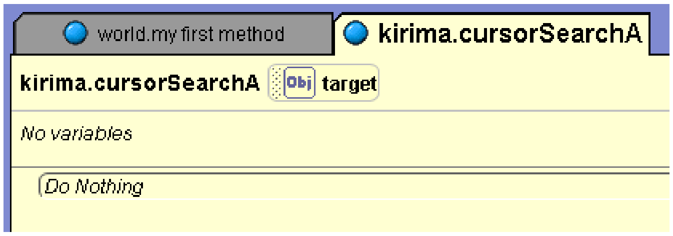
Create New Parameter

Name: target

Type: ○ Number ○ Boolean ● Object ○ Other... String

☐ make a List

Figure 4-11 Create the target parameter.



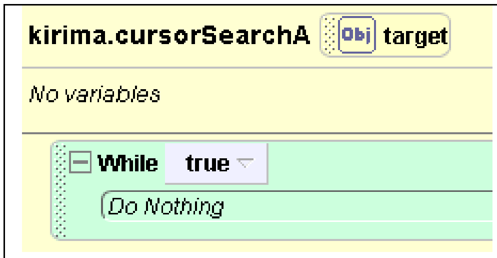
kirima.cursorSearchA Obj target

No variables

Do Nothing

Figure 4-12 target parameter shows at top of cursorSearchA method.

Be sure that the **kirima** object is selected in the **Object Tree** panel and that the function tab is active in the **Details** panel. Drag a While control to the *Do Nothing* spot in the *cursorSearchA* method as shown in **Figure 4-13a** and select *true*. Then drag in the proximity function, **kirima** *is at least threshold away from object*, and drop it on the *true* condition. Select *2 meters away from/expressions/target*. You can see the resulting instruction in **Figure 4-13b**



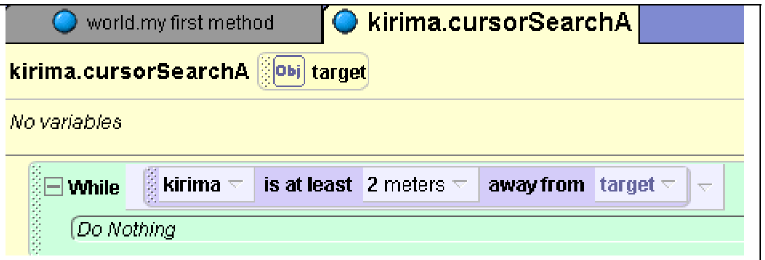
kirima.cursorSearchA Obj target

No variables

While true

Do Nothing

Figure 4-13a While control.



kirima.cursorSearchA Obj target

No variables

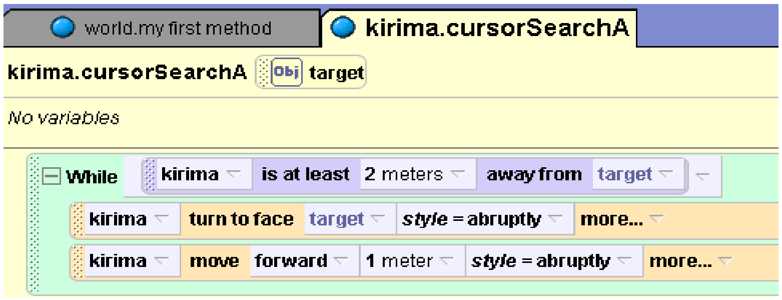
While kirima is at least 2 meters away from target

Do Nothing

Figure 4-13b Replacing the true condition with a function.

Recall that functions return values. In this case, the function shown in **Figure 4-13b** will return either a value of true or false to the While control. The partial While control shown in **Figure 4-13b** tells the computer that as long as **kirima** is at least 2 meters away from the target, the returned value is *true*, which means *Do Nothing*. Of course if **kirima** is indeed more than two meters from the selected target object, we want her to continue moving toward it instead of doing nothing. So, you can add the instructions shown in **Figure 4-14** to have **kirima** move toward the selected target object**.** Go ahead and add the instructions now.

Important: The way the While control works in this case is that as long as the While condition remains *true*, **kirima** will cycle through the two instructions shown in **Figure 4-14** again and again. But, as soon as the condition turns *false* (i.e., **kirima** is equal to or closer than 2 meters from the target object), the While control will be finished and execution moves to the next sequential instruction in the **kirima.***cursorSearchA* method. But at this time, there is no other instruction. Therefore, the **kirima.***cursorSearchA* method is finished.



```
kirima.cursorSearchA  Obj target
No variables
While  kirima  is at least  2 meters  away from  target
    kirima  turn to face  target  style = abruptly  more...
    kirima  move  forward  1 meter  style = abruptly  more...
```

Figure 4-14 While control with all its instructions.

We want **kirima's** *cursorSearchA* method to be activated whenever the mouse cursor clicks on anything. Look in **karima's** method tab in her **Details** panel. Drag **kirima's** *cursorSearchA* method and drop it into the do Nothing part of the When the mouse is clicked on event which is in the **Events** panel. In the drop-down list that opens, select *expressions* and then *object under mouse cursor* as shown in **Figure 4-15a**. The resulting event should look like **Figure 4-15b**. [If you get an error message, download an earlier PC version of Alice 2.2 from http://alice.org. Downloads/Get Alice 2.2. Click Previous Versions. Current Mac versions work fine.]

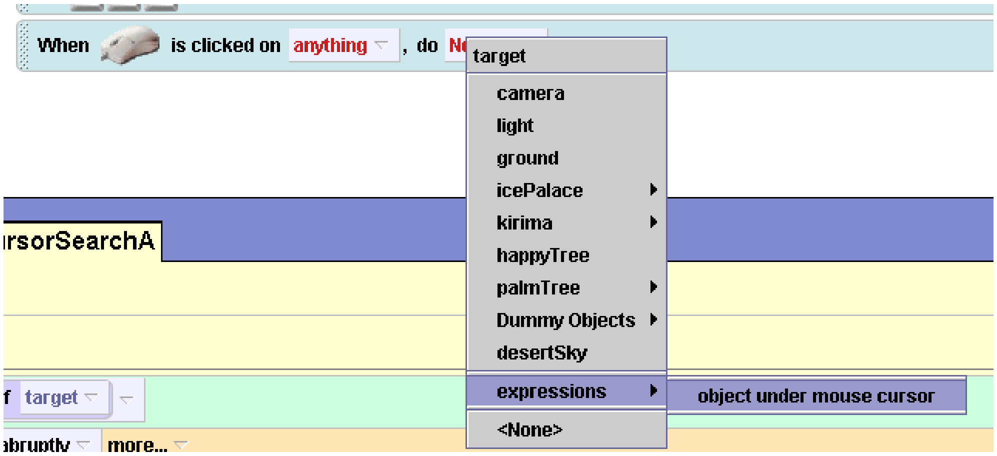
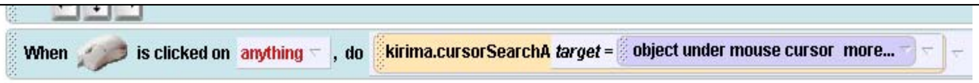


Figure 4-15a Dragging the method into the event.



```
When  is clicked on  anything , do  kirima.cursorSearchA target = object under mouse cursor  more...
```

Figure 4-15b Event.

Save your world and try it out by clicking on Play. When **kirima** has finished her initial moves, click on each of the trees, one by one. Be careful, as you may not actually be clicking what you think you are clicking on. For example, you might click the **grass**, which is an object. Practice having **kirima** move back and forth between the two trees. Finally, click on the **icePalace**. Notice, that **kirima** moves inside the **icePalace** since that is where the **icePalace's** center is. If you press the up-arrow arrow key when she stops, you can make **kirima** go through the **icePalance's** wall. Down-arrow key moves her back. Recall that we programmed the arrow keys earlier to control **kirima's** movements – the up-arrow key moves **kirima** forward. This is an example of how virtual worlds can be a playground for fanciful situations.

Tech Talk - The proximity function determines how close one object is to another by examining the distance between their centers. If you move kirima to the left or right in the World View panel and then click on the icePalace, shown in either the Object Tree panel or the World View panel, you can see that the center of the icePalace, is inside the icePalace. Because of this, it is very challenging to develop a method that would permit kirima to walk up to the inside wall of the icePalace and stop before walking through it.

Finally, add some print control text that tells the viewer to click on an object to make **kirima** move toward it. See **Figure 4-16** for the location of this text.

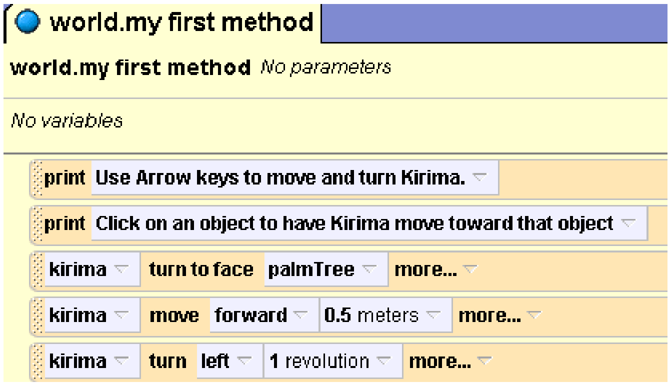
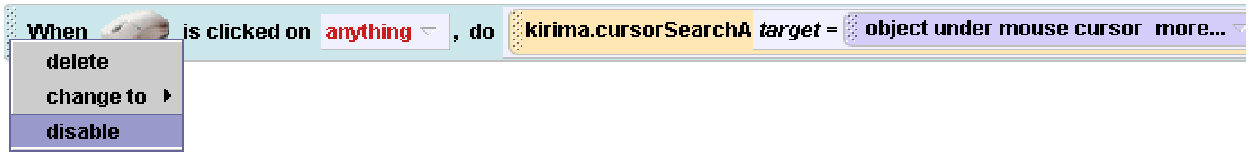

world.my first method
world.my first method No parameters
No variables
print Use Arrow keys to move and turn Kirima.
print Click on an object to have Kirima move toward that object
kirima turn to face palmTree more...
kirima move forward 0.5 meters more...
kirima turn left 1 revolution more...

Figure 4-16 User instruction added.

Create kirima Cursor Search Method B

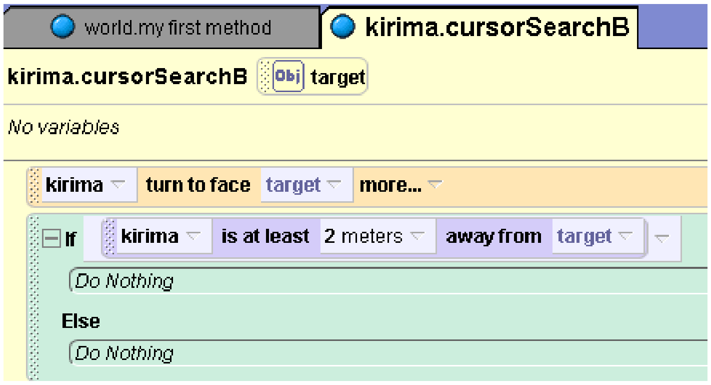
In computer programming, just like in any other activity, there are usually a number of different solutions (i.e., algorithms) for any one problem. Here is different method for **kirima** that will enable her to move toward another object. In this case, the method will explicitly refer to itself.

In the **Events** panel, right-click on the tile for the event we just added and click on disable in the drop-down list that opens, as shown below. As a result, the event will appear grayed out.


When is clicked on anything , do kirima.cursorSearchA target = object under mouse cursor more...

Go ahead and create another method for **kirima** named *cursorSearchB*. Use **Figure 4-17** as your guide.

Let's do this step-by-step. Because this is a different method from the one we just created, you will need to create a parameter for this method as well. Parameter names have meaning only within the method they are created in, so you may name this parameter **target** as well. Go ahead and do this. Notice that this method starts off with an instruction to point **kirima** toward the **target** (i.e., the target parameter). Next, instead of the While control used in cursorSearchA, we use an If/Else control. Drag it in. Notice, in **Figure 4-17**, that the If/Else control is asking the proximity function whether **kirima** is at least 2 meters away from the **target.** This is just like the While control in cursorSearchA. Go ahead and do it. Done with **Figure 4-17**.



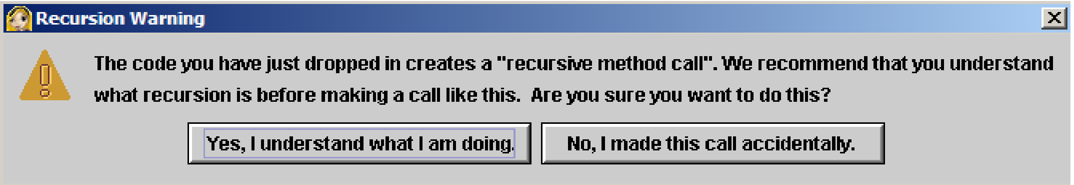
```
kirima.cursorSearchB  Obj target
No variables
kirima  turn to face  target  more...
If  kirima  is at least  2 meters  away from  target
    Do Nothing
Else
    Do Nothing
```

Figure 4-17 Beginning of cursorSearchB.

Any instruction we put in the *Do Nothing* block under the If will be executed as long as the If condition is *true* (i.e., **kirima** *is at least 2 meters away from target*). So, let's instruct **kirima** to move forward 1 meter since she is not yet close enough to the **target**. See the instruction, **kirima** *move forward 1 meter* shown in **Figure 4-18**.

So far, so good. We want **kirima** to continue to face the **target** and move toward it as long as she is as least 2 meters away from the **target** (i.e., the If condition is *true*). In other words, we want her to do what the *cursorSearchB* method does. We can accomplish that by dragging the *cursorSearchB* method tile into set of instructions following the *true* condition. You can see the result in **Figure 4-18.** Yes, you are actually creating a method that calls itself into action!

When you do drag *cursorSearchB* in, the following warning message appears.


Recursion Warning
Yes, I understand what I am doing.
No, I made this call accidentally.

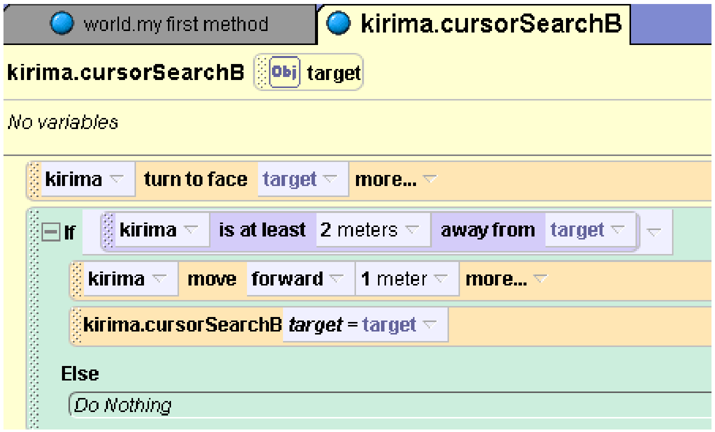
The code you have just dropped in creates a "recursive method call". We recommend that you understand what recursion is before making a call like this. Are you sure you want to do this?

{**Video demo**: Video 4-3}

The warning message wants you to be sure that you know that you are asking *cursorSearchB* to send a message to itself - telling itself to run itself - which could result in an infinite regress. Click "Yes" in the warning.

As an analogy, think of when you may have stood between two mirrors that faced each other and looked to one side. You saw an infinite number of your own images in the mirrors.

By leaving the Else spot of the If/Else control empty, you create a stopping point so that no infinite regress occurs. That is, if **kirima** is no longer at least 2 meters away from the target, the If condition turns to the value *false*, and the Else part of the If/Else construct will be activated. The *Do Nothing* will result, thus stopping the method from doing anything more. This type of method, one that calls itself, is called a “recursive” method.

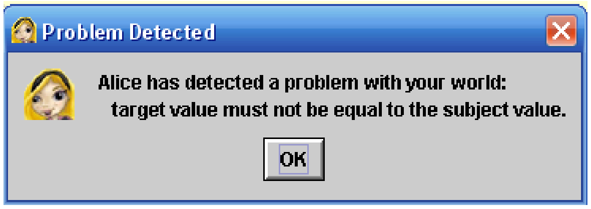


```
kirima.cursorSearchB  obj target
No variables
kirima  turn to face  target  more...
If  kirima  is at least  2 meters  away from  target
    kirima  move  forward  1 meter  more...
    kirima.cursorSearchB target = target
Else
    Do Nothing
```

Figure 4-18 Alternative cursorSearchB recursive method for mouse click control.

To try method *cursorSearchB* out, you must create an event like you did for *cursorSearchA* (the one you just disabled). See previous pages for the way to do this. Alternatively, you could simply change the current event by replacing the call to *cursorSearchA* with a call to *cursorSearchB,* thereby replacing one behavior with another. After you create (or modify) the event, click Play and try clicking the various objects in the scene.

Warning: If you try clicking on **kirima**, the animation will end and you will see a Problem Detected message like the one shown below. This simply means that you can’t have the target move to itself.



Alice has detected a problem with your world:
target value must not be equal to the subject value.

Now you have two alternative methods to make **kirima** respond to a mouse click: *cursorSearchA* and *cursorSearchB*. She is almost ready to go on her quest.

Finding Kirima

kirima may go off-camera during her quest. You (or someone) will be controlling her movements and she may be moved out of camera range. So, it is important for you, the gamer, to have an easy way to find her.

Let’s create a method that has the **camera** turn to face **kirima**, no matter where she is in the game space. Here’s how to do it. Click on **camera** in the **Object Tree** panel. Click the method tab in the **Details** panel. Create a new method and name it “findKirima.” Click the method’s edit tile. Drag in the instruction that you see in **Figure 4.20**.

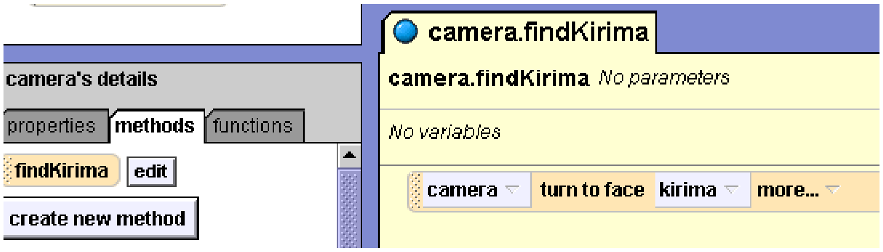


Figure 4-20 New method findKirima.

Now you must create an event to activate the method. In the **Events** panel, click on create new event. Click on the event trigger When a key is typed. Scroll down to the event you just created, click on any key and choose letters. Click on the letter F (stands for Find). Click on **camera** in the **Object Tree** panel. Drag the new method, *findKirima*, into the *Do Nothing* spot of the new event. The new event is shown in **Figure 4-21**. The method **Camera**.*findKirima* is the event handler.

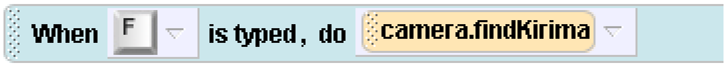


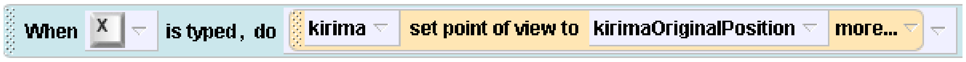
Figure 4-21 Event for camera to find kirima.

Click Play and use the arrow keys to move **kirima** off screen. Press the F key. After the **camera** finds **kirima**, move her off screen again and press the F key. As you can see, this only gets us so far. We want much more user control over the **camera** and over **kirima**.

Bringing Kirima back home

It is important that the gamer has an easy way to return **kirima** to her original position, no matter where she has moved to in the game world.

To do this, we create a new event and make when X is typed as the event trigger. The event handler will be **kirima**.*set point of view to* **kirima.originalPosition** which you can find in **kirima's** methods tab. Recall that **kirima.originalPosition** was the dummy object we set up at the start of this Chapter. {**Video demo**: Video 4-4} The final event should look like:



Don't forget to tell the user about the F key and the X key features by adding some print controls to the **world**.*my first method* as was done for other keys. Make sure everything works by clicking the Play button and moving **kirima** to the other side of the **icePalace** (i.e., out of sight). Then press the X key to return her to her original position. Note that if after you move **kirima** through the igloo to its other side, and if you press the F key before you press the X key, the **camera** will look toward **kirima** but its view of her will be obscured by the igloo.

Authors' file **AWB4-1.a2w** to this point in the chapter can be found at http://media.pearsoncmg.com/aw/aw_snyder_fluency_3/alice/World_Files/.

4.4: First person point of view

See the world through Kirima's eyes

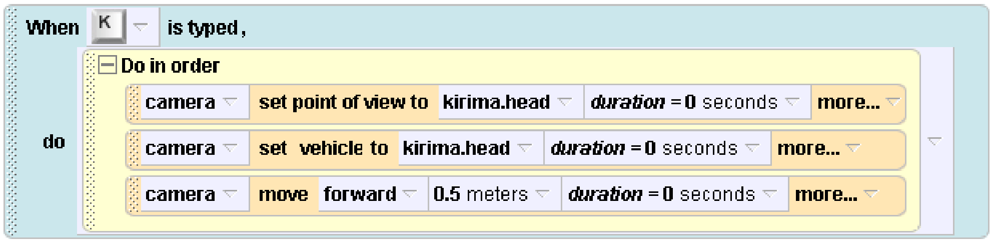
Although **kirima's** quest allows you, the viewer, to control her movements, you would also like to see things as she does – through her eyes. This means that we must move the **camera** to **kirima's** point-of-view and make it travel with her wherever she goes. Recall from Chapter 3 that point-of-view consists of position plus orientation.

We need to create a new event such that when we press the K key, the **camera** sees things through **kirima's** eyes and the action turns into a first-person game. We would also like to be able to press the Space bar on the keyboard to return the **camera** to its original position – where it was at the beginning of the game. Recall that we put a **dummy** object at that position and named it **wideCamera**.

In the **Events** panel, click on create new event and select When a key is typed. Scroll down to the event you just entered, click on *any key* and in the drop-down list, select letters and then the letter K. Drag a Do in order control into the *Do Nothing* spot. Click on **camera** in the **Object Tree** panel and drag in the items shown in **Figure 4-22**. The **camera** *set point of view to* is from the **camera** methods tab and is set to **kirima.head's** point of view. The **camera** *set vehicle* is from the **camera's** properties tab and is also set to **kirima.head**. This means that the object **kirima.head** will be the vehicle for the **camera** object.

Analogy: This is just like the real-life car you ride in. The car is a vehicle for your body and you go where it goes. Your position is its position. When the car turns a corner, you turn with it and, if you are looking out of the front window, your orientation tracks the car's orientation.

Drag in an item to create an instruction to *move* the **camera** forward from **kirima's head** 0.5 meters. Otherwise, your view will be blocked by **kirima's** brown hood. See **Figure 4-22**. Set all the *durations = 0* seconds.



```
When K is typed,
do
   Do in order
      camera  set point of view to  kirima.head  duration = 0 seconds  more...
      camera  set vehicle to  kirima.head  duration = 0 seconds  more...
      camera  move  forward  0.5 meters  duration = 0 seconds  more...
```

Figure 4-22 New event to move camera to kirima's point-of-view when K key is pressed.

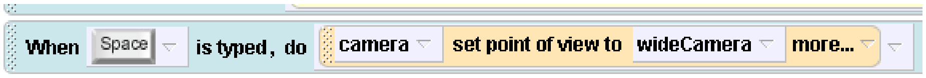
Click on Play. Press the K key during **kirima's** movements. You can still control her movements with the arrow keys and the mouse click. Try moving **kirima** toward the trees and into the **icePalace**. Use the arrow keys to have her move straight through the back wall of the igloo. Then, use the arrow keys to return her inside the igloo again and have her turn to face the opening of the igloo. Then, click on a tree to get her to move to it. Press the X key to return her to her original position. Try other key combinations to see how things look through **kirima's** eyes.

Here is what you observe. You can observe that the *set point-of-view* instruction places the **camera** at the **kirima.head** object facing the same way **kirima's** head is facing. You can observe that *set vehicle to* attaches the **camera** to **kirima's** head so that the **camera** moves with her as she moves. The combination of these two instructions gives us the first-person perspective for our game. The **camera** is now in the same location (i.e., position) as **kirima's** head and, in effect, is glued to her head, facing the same way the head is facing (i.e., orientation). Recall our Chapter 3 explanation that point-of-view = position + orientation.

Restore camera to its original point-of-view

During the game we may wish to restore the camera to its original position and orientation (aka, point of view). We can do this by creating an event that uses the Space bar (or any other key) to trigger an instruction to place the **camera** at the **wideCamera** dummy object location and face in the same direction as the **wideCamera** dummy object. Recall that we dropped this dummy object at the original **camera** location. Thus, hitting the Space bar would restore the camera to where it was before it was set to **kirima's** point of view.

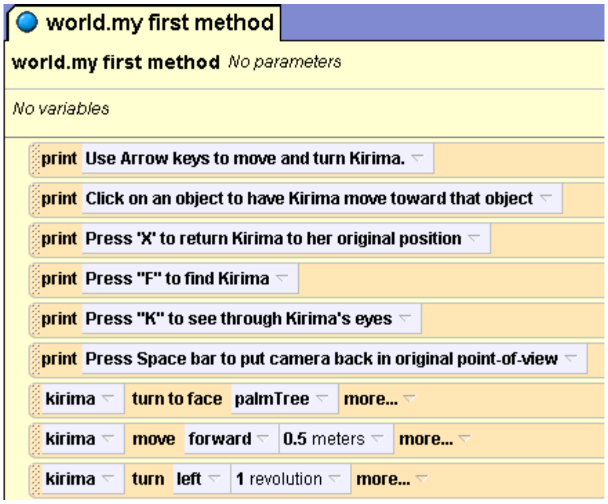
Create the event shown in **Figure 4-23**. The event trigger instruction comes from the **camera's** method tab.



When Space is typed, do camera set point of view to wideCamera more...

Figure 4-23 Restore the camera to its original point-of-view at the wideCamera dummy object.

In order to alert the game user to the K and Space bar functionalities we will want to include two new print control statements in **world**.*my first method*. See **Figure 4-24**.



```
world.my first method
world.my first method No parameters
No variables
print Use Arrow keys to move and turn Kirima.
print Click on an object to have Kirima move toward that object
print Press 'X' to return Kirima to her original position
print Press "F" to find Kirima
print Press "K" to see through Kirima's eyes
print Press Space bar to put camera back in original point-of-view
kirima turn to face palmTree more...
kirima move forward 0.5 meters more...
kirima turn left 1 revolution more...
```

Figure 4-24 print statements added for F, K and Space bar.

This event trigger, shown in **Figure 4-23**, when activated by the Space bar, simply places the camera at the **wideCamera** object location and orients the **camera** to face the same way that the **wideCamera** object is facing. Note that this event does NOT change the vehicle for the **camera**. If you Play the world and press the K key, the **camera** gets **kirima's** point-of-view and **kirima** as its vehicle (i.e., **camera** moves with her). If you later press the Space bar, the camera goes to the **wideCamera's** location but the **camera** still moves in consort with **kirima** when and where she moves. Although the **camera's** point of view has been changed, **kirima** is still the **camera's** vehicle. But now, the **camera** may be physically located at some distance from her head. This makes for some interesting camera angles and shots.

Try this sequence of actions. Take your time and observe as you move along. Play; let **kirima** do her initial thing; press K key; press the down-arrow key 20 times; press X key; press Space bar; hold left (or right) arrow key down [Notice how the lighting on **kirima's** face changes. There is a light in this world and it can cast shadows.]; press and hold the up-arrow and/or the down-arrow key; press X key. What do you think?!

Now that we are in the first-person mode, you can create some very interesting camera effects using these keys in different sequences. Before we try a different key sequence, recall what setting point-of-view with the Space bar does. It puts the camera at the location of the dummy object called **wideCamera** and has it facing the same way that the **wideCamera** object faces. But because **kirima** is still the vehicle for the **camera**, the **camera** object will move when **kirima** moves. It's as though the **camera** is on the end of a stick protruding out from **kirima's** head.

The **camera** is pointing the same way the **wideCamera** object is pointing but the **camera** moves through the **world** as if at the end of the "stick" whose other **end** is attached to **kirima's** head. This can make for a very interesting **camera** view that follows **kirima** at a distance.

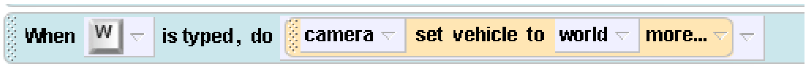
Try this sequence of actions to see what we mean. Actions are shown in **Figure 4.25**.

1. Play world.
2. After **kirima** does her first moves, use the up-arrow key to move **kirima** forward a little bit.
3. Press the K key to get first person perspective.
4. Move **kirima** forward a little using the up-arrow again.
5. Press the Space bar. [This sets the camera to the **wideCamera** object's point-of-view.]
6. Now, use the right-arrow key to turn **kirima**. Notice that the camera stays fixed looking at **kirima** and turns with her. This makes the background look like its moving. Turn **kirima** all the way around and notice how the lighting and shadows change. Very cool!
7. Move **kirima** backward and forward using the down and up-arrows. [Notice that the **camera** maintains its original orientation but its position moves with **kirima** as her position changes. **Kirima** is still the **camera's** vehicle.]
8. Now, press X to put **kirima** at her original position.
9. Then press the right or left arrow key to swivel **kirima** around. Notice that the **camera** is still on a 'stick' protruding out from **kirima** head some distance away. When she turns you get to see the effect of a 360 degree moving view of the game space (i.e., **world**).

Figure 4.25 Example key sequence controlling kirima's movements & camera point-of-view.

Release camera from kirima vehicle

We certainly don’t want the **camera** stuck with **kirima** as its vehicle forever. Releasing it from **kirima** is easy. We just create a new event and use a new key (e.g., W) to make the **world** the **camera’s** new vehicle. Then, pressing the Space bar and then the W key will not only put the **camera** back at its original position but will make the **world** the **camera’s** vehicle. See **Figure 4-26**. Put a print control statement in **world**.*myfirst method* to alert the user.



```
When W is typed, do camera set vehicle to world more...
```

Figure 4-26 W key sets the camera’s vehicle to world, thus releasing it from kirima.

As an exercise, place more objects in the scene out of sight of the **camera.** When the animation starts, move **kirima** around so that these objects come into view. This is similar to the Alice **amusementPark** example at the beginning of this chapter.

Authors’ file **AWB4-2.a2w** to this point in the chapter can be found at http://media.pearsoncmg.com/aw/aw_snyder_fluency_3/alice/World_Files/.

4.5: Finding & Picking up Quest Objects

What’s a quest without finding hidden objects and picking some of them up?

Let’s add an object to the scene. We will create it, position it and then of course, make it invisible (i.e., hide it) to be discovered by **kirima**.

Click ADD OBJECTS and in the Local Gallery find the Fantasy category. Click on it and then click on the Faeries tile. Add an instance of LichenzenSpider to your world. Change the name to **fairyPrincess**. Using your mouse cursor, move **fairyPrincess** to the **happyTree**. Then, in the **Object Tree** panel, right-click on **fairyPrincess**, select methods and resize her to the size you see in **Figure 4-27**. Make **fairyPrincess** invisible by right-clicking on **fairyPrincess** in the **Object Tree** panel and select methods and then select *set opacity to* invisible.

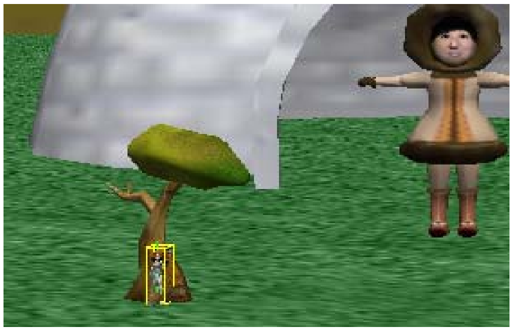


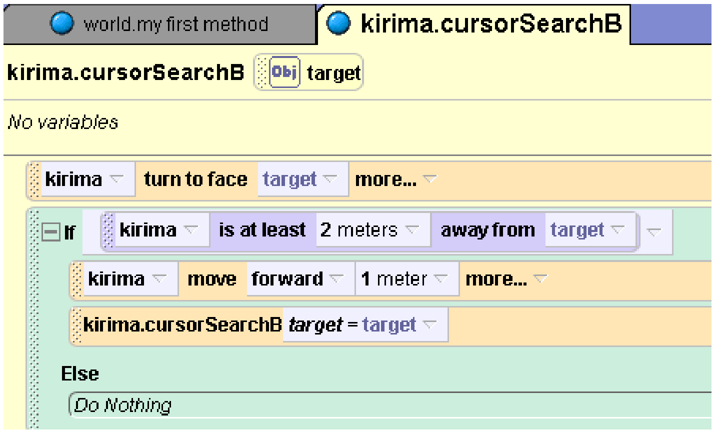
Figure 4-27 Create fairyPrincess and hide her in the happyTree.

Look ahead – In this game you will move kirima around until she finds the fairyPrincess who is initially invisible. When found, fairyPrincess materializes, latches onto kirima and goes wherever kirima goes. There is a secret entrance to a National Park that is revealed when fairyPrincess is found.

Finding an object

This idea is that when **kirima** gets close to **fairyPrincess, fairyPrincess** becomes visible. To do this requires some additional instructions in the *cursorSearchB* method, and *cursorSearchA* method depending on which method you enabled in the **Events** panel. We will use the *cursorSearchB* method for the rest of this chapter.

Look at the *cursorSearchB* method shown repeated in **Figure 4-18**, below.

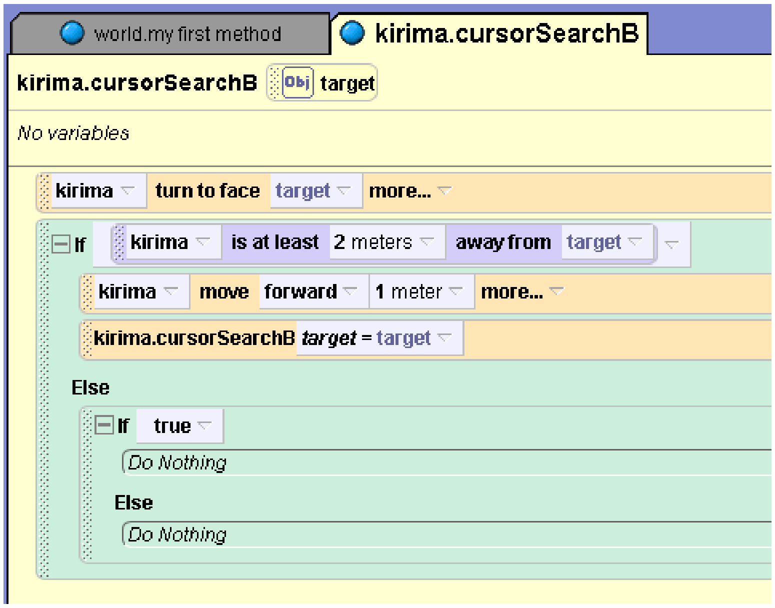


```
kirima.cursorSearchB  obj target
No variables
  kirima  turn to face  target  more...
  If  kirima  is at least  2 meters  away from  target
    kirima  move  forward  1 meter  more...
    kirima.cursorSearchB target = target
  Else
    Do Nothing
```

Figure 4-18 (repeated here). Alternative method for search.

The logic goes like this: In the *cusorSearchB* method, focus on the If/Else control. The only way the Else part is activated is when the If condition is false (i.e., **kirima** is not as least 2 meters away from the **target** object – means she is at or within 2 meters from the **target**). What we want is for the **fairyPrincess** to become visible when **kirima** is within 2 meters of the **fairyPrincess** object, but not within 2 meters of any other object. Thus, we need to put such an instruction in the Else's *Do Nothing* spot.

Drag an If/Else control into the *Do Nothing* spot of the Else part of the If/Else control and select true from the drop-down list. See **Figure 4-27**. Note we now have nested If/Else controls – one inside of another.



```
kirima.cursorSearchB   Obj target
No variables
kirima   turn to face   target   more...
If   kirima   is at least   2 meters   away from   target
    kirima   move   forward   1 meter   more...
    kirima.cursorSearchB target = target
Else
    If   true
        Do Nothing
    Else
        Do Nothing
```

Figure 4-27 cursorSearchB with beginning of proximity test.

In the true part of this If/Else control, drag in **kirima's** proximity function kirima is within threshold of object. See **Figure 4-28**. See result in **Figure 4-29**. {**Video demo**: [Video 4-5]}

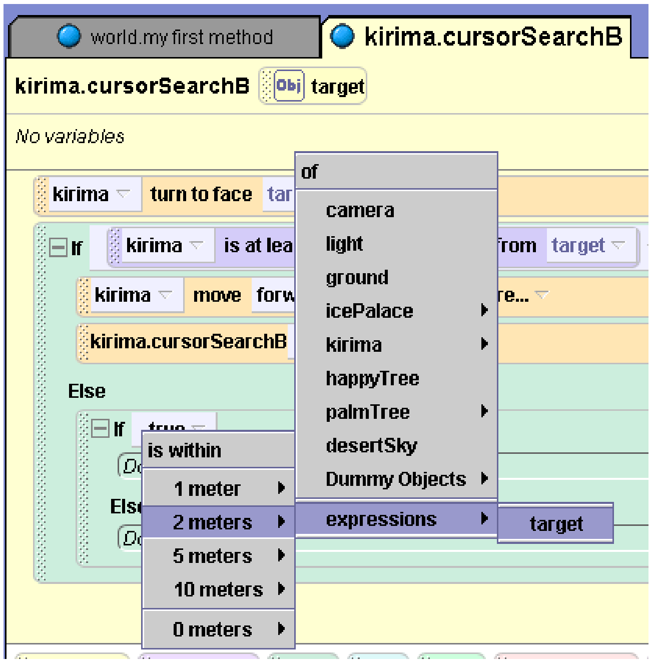
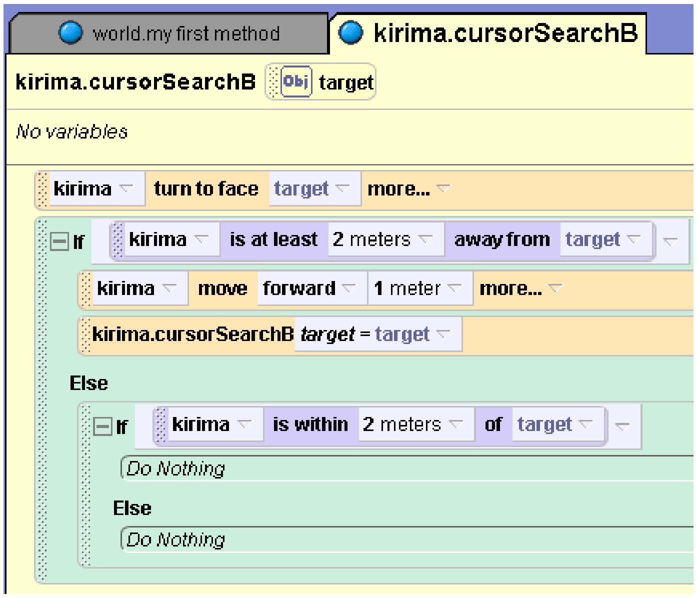


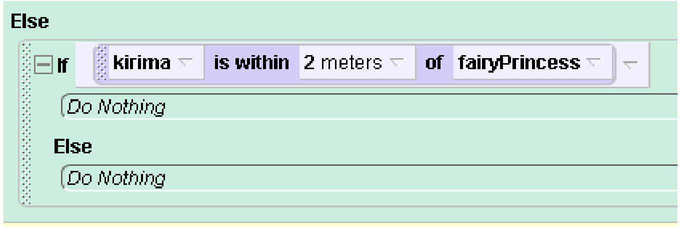
Figure 4-28 Place kirima's proximity function in true part of the nested If/Else control.



```
kirima.cursorSearchB  obj target
No variables
kirima turn to face target more...
If kirima is at least 2 meters away from target
    kirima move forward 1 meter more...
    kirima.cursorSearchB target = target
Else
    If kirima is within 2 meters of target
        Do Nothing
    Else
        Do Nothing
```

Figure 4-29 kirima's proximity function in the nested If part.

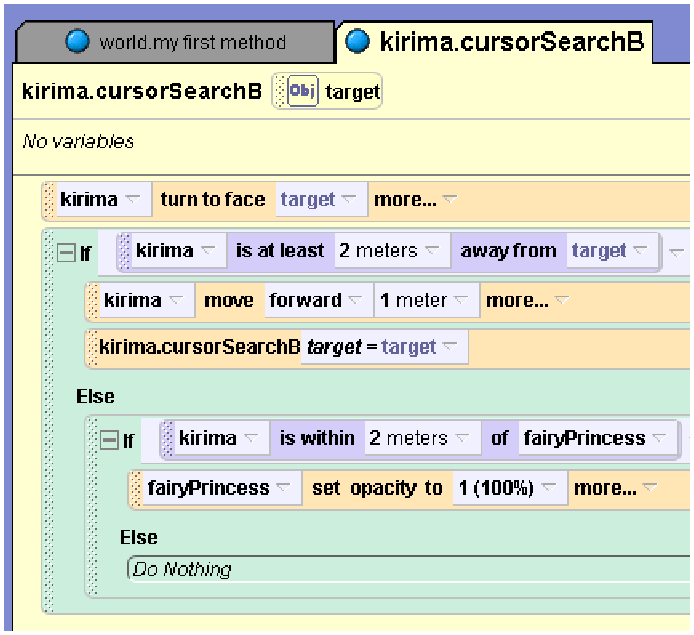
Now, click on the **target** of the If part and select **fairyPrincess** since we only want the **fairyPrincess** to become visible if she, and she alone, is the **target** the cursor clicks on and at the same time, **kirima** is within 2 meters. See **Figure 4-30**.



```
Else
    If kirima is within 2 meters of fairyPrincess
        Do Nothing
    Else
        Do Nothing
```

Figure 4-30 Test to see if fairyPrincess is the target.

Finish the method by instructing the **fairlyPrincess** to become visible when **kirima** is within 2 meters of the **fairyPrincess**. Place that instruction as shown in **Figure 4-31**. Leave the Else part blank since the method needs to stop somehow and will do so whenever **kirima** is not within 2 meters of **fairyPrincess** but is within 2 meters of some other target.



```
kirima.cursorSearchB  Obj target
No variables
kirima  turn to face  target  more...
If  kirima  is at least  2 meters  away from  target
    kirima  move  forward  1 meter  more...
    kirima.cursorSearchB target = target
Else
    If  kirima  is within  2 meters  of  fairyPrincess
        fairyPrincess  set  opacity to  1 (100%)  more...
    Else
        Do Nothing
```

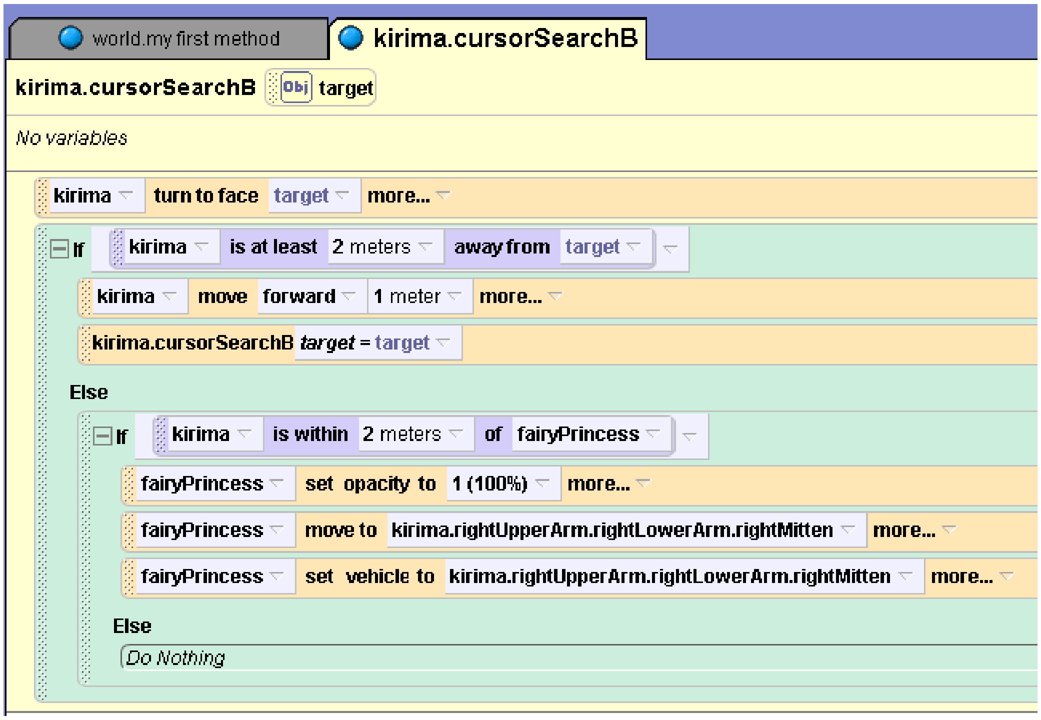
Figure 4-31 cursorSearchB method.

Play, and then click on the **happyTree** to cause **kirima** to move to it. The **fairyPrincess** should appear. Play again clicking on **palmTree**. **fairyPrincess** should not become visible. Play again and click first on **palmTree** and then on **happyTree**. **fairyPrincess** should become visible only when **kirima** approaches **happyTree**.

Picking up an object

It is one thing to find an object, but quite another thing to pick it up and carry it with you.

When **kirima** finds **fairyPrincess** we want **fairyPrincess** to latch onto **kirima** and move with her wherever **kirima** goes. To make this happen, drag **fairyPrincess** *move to* **kirima's rightMitten** into the *cursorSearchB* method. Then, make **kirima's rightMitten** the *vehicle* for the **fairyPrincess.**
See **Figure 4-32**. {**Video demo**: Video 4-6}

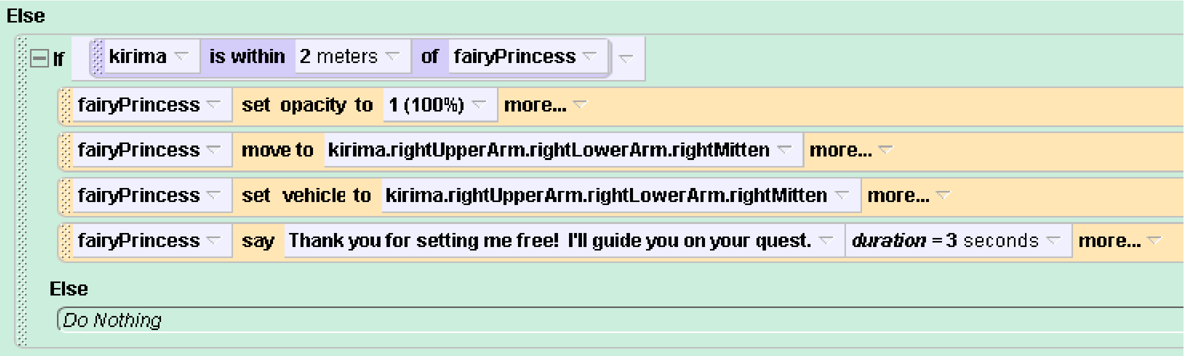


```
world.my first method | kirima.cursorSearchB
kirima.cursorSearchB  obj target
No variables
kirima turn to face target more...
If kirima is at least 2 meters away from target
    kirima move forward 1 meter more...
    kirima.cursorSearchB target = target
Else
    If kirima is within 2 meters of fairyPrincess
        fairyPrincess set opacity to 1 (100%) more...
        fairyPrincess move to kirima.rightUpperArm.rightLowerArm.rightMitten more...
        fairyPrincess set vehicle to kirima.rightUpperArm.rightLowerArm.rightMitten more...
    Else
        Do Nothing
```

Figure 4-32 fairyPrincess moves to kirima's rightMitten which is vehicle for fairyPrincess.

Play to see if it works correctly. After **fairyPrincess** is on **kirima's rightMitten**, use the cursor keys to turn **kirima** so you can see **fairyPrincess** stay on the **mitten**. Click on the **palmTree** to see **kirima** and her **fairyPrincess** move together to it. Use the keys you set for various functions (e.g., K, X, Space bar) and the arrow keys to move **kirima** and her fairy guide to various places in the game world.

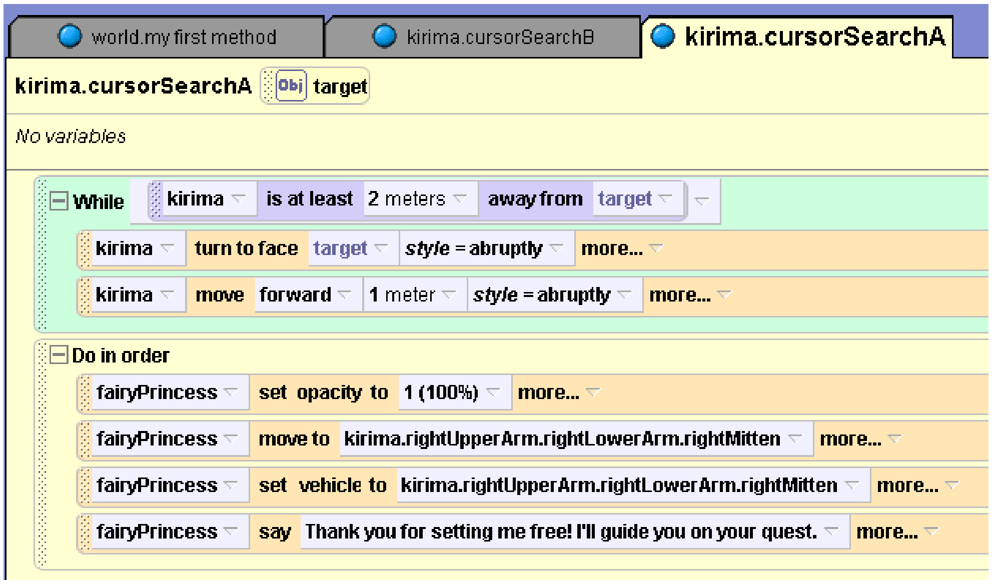
It is good fortune that **kirima** discovered the **fairyPrincess**. So, let's have the **fairyPrincess** say to **kirima**: "Thank you for setting me free! I'll guide you on your quest." To do this, we drag the **fairyPrincess** *say* method into the *cursorSearchB* method. See **Figure 4-33**.



```
Else
    If kirima is within 2 meters of fairyPrincess
        fairyPrincess set opacity to 1 (100%) more...
        fairyPrincess move to kirima.rightUpperArm.rightLowerArm.rightMitten more...
        fairyPrincess set vehicle to kirima.rightUpperArm.rightLowerArm.rightMitten more...
        fairyPrincess say Thank you for setting me free! I'll guide you on your quest. duration = 3 seconds more...
    Else
        Do Nothing
```

Figure 4-33 Completed cursorSearchB method.

Copy instructions to cursorSearchA method. If you want, you can now place a Do in order control in the *cursorSearchB* method and drag the **fairyPrincess** instructions into the Do in order control. This is so you can easily copy the Do in order control and its instructions to the clipboard and later paste them into the *cursorSearchA* method in case you decide to use *cursorSearchA* instead of *cursorSearchB*. See **Figure 4-34**.



```
kirima.cursorSearchA  Obj target
No variables
While  kirima is at least 2 meters away from target
    kirima turn to face target style = abruptly more...
    kirima move forward 1 meter style = abruptly more...
Do in order
    fairyPrincess set opacity to 1 (100%) more...
    fairyPrincess move to kirima.rightUpperArm.rightLowerArm.rightMitten more...
    fairyPrincess set vehicle to kirima.rightUpperArm.rightLowerArm.rightMitten more...
    fairyPrincess say Thank you for setting me free! I'll guide you on your quest. more...
```

See Figure 4-34 cursorSearchA method with fairyPrincess instructions copied into it.

Optional: You can add instructions to make the **fairyPrincess** move her rightArm. See Chapter 4 exercises.

Authors' file **AWB4-3.a2w** to this point in the chapter can be found at http://media.pearsoncmg.com/aw/aw_snyder_fluency_3/alice/World_Files/.

4.6: Advertising billboards, pictures and sound

What's a game without pictures and sound? Let's make those enhancements.

Adding pictures

With Alice, we can add images in from the outside. These are called "Billboards." A Make Billboard… option exists in the File menu of Alice. This will allow you to select a JPG, GIF, TIFF or BMP image to incorporate in your world. Another way to create a Billboard is to drag an image from a folder on your computer system directly into the **Word View** panel. Tip: Alice does best with JPG and GIF files.

For our quest, we will add an image using the File/Make Billboard… approach. You can do this with any image file already on your computer. Or, you can find images on the web, but be sure that you are not violating copyright laws. In addition, you can create your own image with a digital camera or cell phone, or even draw your own image with the Windows Paint facility, for example.

For our quest game, we will add an image as an object that **kirima** finds with the help of her new found fairy princess guide. We will alert **kirima** by using a sound file that we will create and we will notify the user via a print control.

Games should be entertaining. But also they can be educational - promoting some specific message. To give a non-commercial example of this, we have selected a picture taken at Denali National Park in Alaska. You can find this image, Dena2452.jpg at ftp://ftp.aw.com/cseng/authors/snyder/fluency3e/Alice/WorldFiles/. This picture is in the public domain courtesy of the National Park Service and thus there are no copyright issues we need to worry about. Use your own image if you wish. You could think of this as a end point of **kirima's** quest where the goal is to find this specific objects associated this particular National Park. Maybe that's what **kirima** is supposed to do – move around the game space until she finds this picture. It's your call – it's your game!

Know the folder where you have the image you wish to bring into your game. Click File/Make Billboard.... Browse to the image and import it. You will see the image name in the **Object Tree** panel. The imported picture is an Alice object but is not 3D. You will find the mirror image of the picture on the backside of the billboard.

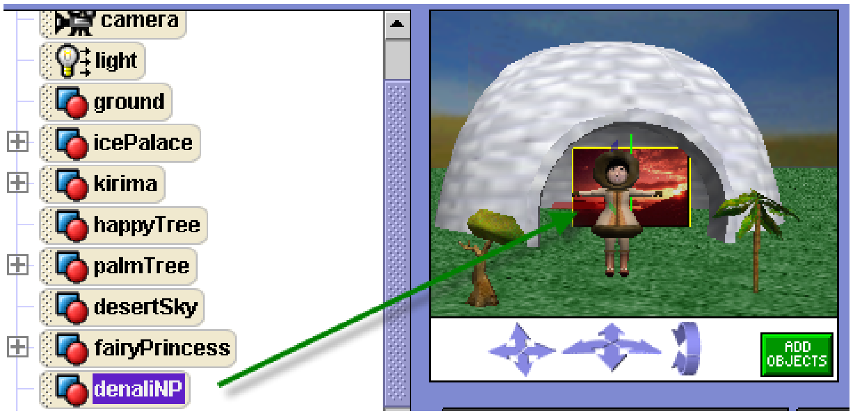


Figure 4-35 Billboard in the game space.

This is a good time to rename the image to a useful name in the context of your game. In this case, we use **denaliNP** which is the designation for Denali National Park. The picture is a bit small, so we can resize the image by a factor of two to get it into the right scale compared to other objects. To do this, right-click on **denaliNP** in the **Object Tree** panel and select methods and then choose *resize*/amount/2 (twice as big).

Let's also have the billboard face **kirima**. Right-click on **denaliNP** and select methods/*turn to face*/**kirima**/the entire **kirima**. To confirm that the **denaliNP** billboard has its mirror image is on the back, choose the method that turns the billboard around 1/2 revolution. Then, Undo.

Fun feature - You can use an imported picture as the ground image. Try this out. Click **ground** in the **Object Tree** panel. Click on the properties tab in the **Details** panel. To the right of the skin texture tile, click on **ground**.*GrassTexture* and choose **denaliNP** and then choose Denali_Texture. The grass in **World View** panel turns into the Danali picture. Click the Undo button if you want.

Let's move the picture back a bit so it fits inside the igloo. Use your cursor to move the billboard into the igloo. It will get smaller as it moves backwards. If necessary, you can resize it larger when you are done moving it.

We want to hide **denaliNP** in the game space. We could make it invisible, as we have done before with **fairyPrincess**. However, another option is to have it pop-up from underground when **kirima** finds the **fairyPrincess**. This provides a different kind of 'reveal' which can be fun and surprising in some situations.

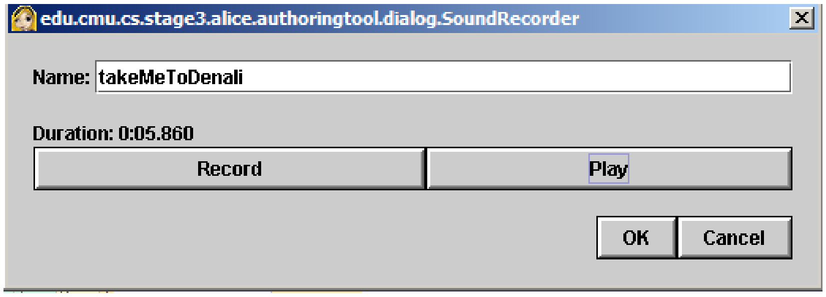
To do this, move the picture down 10 meters by right-clicking on **denaliNP** and, selecting methods/*move*/down/other/10. We could move it down less, but mainly we want it all underground.

Now let's work all of this into the game. Click on **kirima** in the **Object Tree** panel and select the methods tab in the **Details** panel. Click on the edit button next to the *cursorSearchB* method. This is because *cursorSearchB*, rather than *cursorSearchA*, is active in the **Events** panel. Take note of the instruction that has the **fairyPrincess** thanking **kirima**. Drag a Do together control under it. Click on **denaliNP** in the **Object Tree** panel and click on the methods tab in the **Details** panel. Drag the *move*/up/10 into the *Do nothing* part of the Do together control. Click Play and click on the **happyTree** to try this out. The picture pops up inside the igloo. But, let's have it move to the front of the igloo just a bit. Click on **denaliNP** in the **Object Tree** panel and click on the methods tab in the **Details** panel. Drag the *move/*forward/5 meters (or whatever distance works for you) into the Do together control. Click Play and click on the **happyTree**. Copy the Do together and paste it into the *cursorSearchA* method just in case you decide to use it instead of the *cursorSearchB* method.

Adding sound

Let's add an audio clue to the game. Select the **fairyPrincess** in the **Object Tree** panel and select the properties tab in the **Details** panel. Notice the Sounds property toward the bottom. Click the + to expand it. You can import a sound file (wav or mp3 type file) from your computer's hard drive or you can record a sound using your computer's microphone. Most laptops have a microphone built into the unit. All you have to do is to speak toward your laptop.

To record your own voice, click the record sound button. A sound recorder window will appear. Give the sound a name like "takeMeToDenali" and then click the record button. Say or whisper: "Take me to Denali National Park - the entrance is in the Igloo." Select Stop once you have recorded the message. Play the sound to see if it is what you want. You can re-record it if you wish. Whispering provides a somewhat unique voice for the fairy and obscures your own voice. See **Figure 4-36**.


edu.cmu.cs.stage3.alice.authoringtool.dialog.SoundRecorder

Name: takeMeToDenali

Duration: 0:05.860

Figure 4-36 Alice sound recorder.

Click the OK button. The sound object you just created shows up in the **fairyPrincess's** properties tab under Sounds. Now you will want to drag this object into the *cursorSearchB* method following the **denaliNP** move up instruction.

To do this just click **kirima** in the **Object Tree** panel. Click the methods tab in the **Detail** panel. Click the edit button next to the *cursorSearchB* method. Click **fairyPrincess** in the **Object Tree** panel. Click the properties tab in the **Details** panel. Make sure Sounds is expanded. Drag **takeMeToDenali** and drop it under the **denaliNP** *move* up 10 meters instruction in the Do together control. Do the same for *cursorSearchA* in case you decide to use it instead of the *cursorSearchB* method.

It would be good to add a print control to **world**.*my first method* to tell the user to make sure their computer's sound is on.

Final Thoughts on Chapter 4

You have only just begun to play. There is so much more you can do with this game, but there are only so many Workbook pages. For example, you might want to have lots of different objects pop up from underground all over the game space offering clues as to where the **fairyPrincess** can be found. Or, after she is found.

By the way, there is an interesting bug that was inadvertently introduced by virtue of adding the billboard and sound in Section 4.6. The bug causes certain unintended consequences. These things often happen in the process of creating programs. See Exercises 12 and 14**.**

Take a look at the exercises where we have provided some ideas. Don't just rely on us for ideas. Try your own. Talk with your friends and teachers to help generate ideas and even make up your own exercises. You will be pleased at what you can do with Alice. Have fun!

Authors' file **AWB4-4.a2w** to this point in the chapter can be found at http://media.pearsoncmg.com/aw/aw_snyder_fluency_3/alice/World_Files/.

Note: There are four tutorials that come packaged with Alice. You can find them in the "Tutorials" tab in the Welcome to Alice! window accessed from File/New World. We recommend that you do these tutorials. They are informative and fun. Plus, you now have the background and experience to get so much more out of them in light of the Alice fluency skills, concepts and capabilities you have developed.

Exercises for Chapter 4

In this chapter we moved from animated movies to game-like programs where the user participates in the action rather than just watching it. Nonetheless, it is all programming and you have used some of the same techniques discussed in previous chapters. The exercises below are meant to stretch your imagination. Often, good programming has more to do with imagination and creativity than simply writing instructions. Have fun with these exercises.

1. In a sidebar early in the chapter, we suggested that you build your own Alice characters from scratch. In the Local Gallery, go to the People category, scroll to the far right and find the hebuilder and shebuilder classes. Add a character to your scene and demonstrate that the character can walk.

   Saving your new character for later use. Once you have built your character, you may want to save it so that you can use it in other Alice worlds you create. To do this simply right-click on your object in the **Object Tree** panel and select save object. The object will be saved with an .a2c file extension. To import the saved object into another Alice world: File/import; browse to the object file; click the import button.

2. We saw that **kirima** can go through the igloo wall. That's strange, but it's ok in a fantasy world. Try to come up with some algorithms that would prevent **kirima** from walking through the wall. For example, if **kirima** knew the position of the igloo and knew her own position relative to the igloo, is it possible to prevent her from walking through the igloo with mathematical calculations? Do you have any other brilliant ideas? (Hint: This is a challenging exercise, so just try to come up with some clever approaches to the problem.)

3. In Section 4.3, you were shown how to press X to put **kirima** at her original position and then press the right or left arrow key to rotate **kirima** about her axis. Now, place more objects in the scene, but make them out of sight of the **camera** when the animation starts. As you move **kirima** around, demonstrate that these objects will come into view. (Hint: This is very easy exercise, unlike the previous exercise.)

4. A puzzle: Press K and then F. This sequence causes **kirima** to be looking at the back of her own head. Explain why this is the case.

5. If we choose to use *When a key is typed*, we could have assigned movement to each of the four arrow keys individually (named Up, Down, Left, Right) and specified different distances and durations for each key. Try it out.

6. Add some instructions to make the **fairyPrincess** move her **rightArm**. Also, add some interesting background sound for the **fairyPrincess**. Decide when the **rightArm** should move and when the sound should be heard, and implement those behaviors accordingly.

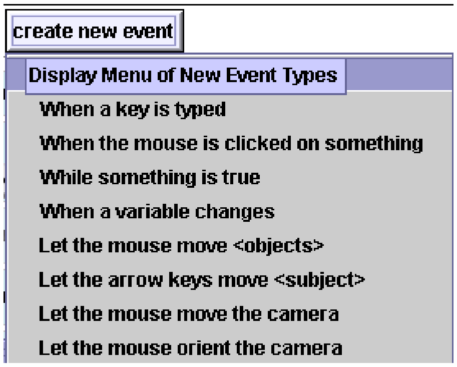
7. Think of some other interesting behavior for some special key, e.g. moving very fast or moving very slow. Create a new event using *When a key is typed* to implement your idea.

8. Try to figure out how to make the **camera** go back and forth from one character to another when the characters are having a conversation, just like you see in the movies.

9. Go back to the Chapter 3 Alice program and Implement the arrow keys for **peter** for use when **ollie** is breathing fire at him. By responding to arrow key events, **peter** could dodge the flames.

10. Want background music to play during the game? Import a wav or mp3 file and set it up to play continuously. Hint: put it in a Loop control inside **world**.*my first method*.

11. Nice that there are a number of event triggers that can be used. Try some out. For example, *Let the mouse move the camera*. This could make for some interesting camera angles.


create new event
Display Menu of New Event Types
When a key is typed
When the mouse is clicked on something
While something is true
When a variable changes
Let the mouse move <objects>
Let the arrow keys move <subject>
Let the mouse move the camera
Let the mouse orient the camera

12. When you play the end of Chapter 4 Alice file (**AWB4-4.a2w**) and click on the **happyTree** object, **fairyPrincell** shows up and says something to **kirima**. The **denaliNP** object appears in igloo. If you click on this object, **kirima** will turn to face it and move toward it. But, as she approaches **denaliNP** unexpected things happen. What and why? Look in the Alice program instructions for the answers. By the way, these same things happen for any object clicked after **fairyPrincess** appears.

13. At the end of the chapter, **kirima** with her **fairyPrincess** see the revealed entrance to Denali National Park in the igloo. Add instructions to make the Billboard come forward from the igloo and resize enough to obscure the igloo. Have **kirima** tell (via sound or *say* method) the user to use the control keys so she and the **fairyPrincess** can move right through the picture (i.e., step into the park).

14. In Exercise 13, above, alter the program instructions so that the unexpected things noted in Exercise 12 no longer occur.

Group Exercise

The following exercise is a group exercise. Depending on the size of the class, form groups of three students or so, and see what you can come up with.

Be a Thinker/Designer. We can let our imaginations run wild to think up interesting things we can do in this sort of fantasy world. Divide yourselves up into groups, where each group writes down some interesting, creative idea that expands upon what you have done in this chapter. Develop storyboards.

Think of yourself as a designer, where you hand over your idea to a 2nd group who must understand and implement the idea. Engage in discussion with the 2nd group to make sure there are no misunderstandings, and have the 2nd group have a go at implementing your idea, whereupon you simply evaluate the 2nd group's implementation by providing feedback and a grade.

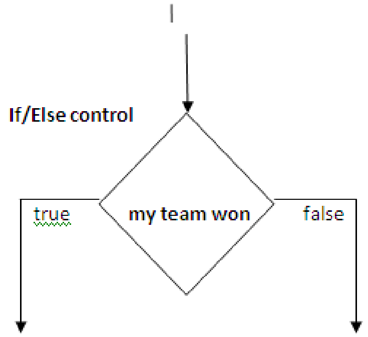
Chapter 4 Skills, Concepts, and Capabilities

In this chapter we moved from animated movies to game-like programs where the user plays a part in the action. Game-like programs are built by writing computer instructions, and thus you have used some of the same technical skills for writing computer instructions in this chapter that you used in previous chapters.

This chapter was big on events. You learned about event-based programming where you wrote computer instructions so that the typing of a keyboard character (i.e., the event trigger) would cause certain instructions to be executed (i.e., the event handler) which in turn impacted the behavior of one or more of your objects (e.g., **kirima** and **fairyPrincess**).

Event programming is all around us. For example, think of what happens when you click a virtual button on your computer screen. The event trigger is when the mouse is clicked within a certain array of pixels and the event handler is the set of instructions that make the button appear to be pressed.

In this chapter you learned about Boolean variables (a variable that has only two values – true or false, sometimes called just Booleans) and another conditional control. When used in an If/Else control, these Boolean values cause instructions in one part or another part of the control to be executed. This is one way computers can be programmed to make decisions based upon the truth or falsity of certain conditions.



Of course, we use these kinds of “controls” all the time in our everyday lives, but now you know that they are used extensively as computer instructions. This is the kind of observation that makes one more fluent in IT and less afraid of technical concepts.

Some other technical concepts that were reinforced in this chapter include modeling & abstraction, algorithmic thinking, expecting the unexpected, and how to debug a computer program. You also learned how to construct two separate programs that cause the same behavior – *cursorSearchA* and *cursorSearchB*. The latter was a recursive method – where an instruction in the *cursorSearchB* method made a call to the method itself! You learned that in general, there are several ways to implement a desired behavior.

And finally, you’ve learned, and we hope experienced, an important concept in this chapter although we never actually used the word: immersion. The concept of immersion is becoming more and more important as we witness the evolution of computers, computer simulations and computer games. An analogy is this: We’re all accustomed to watching a movie passively, but with immersion we actually become part of the movie and our decisions and behaviors influence how the movie unfolds. Sometimes this is called virtual reality and the field is rapidly expanding and changing. Perhaps you have heard of Second Life. If not, a brief web search will help you learn about being immersed in a virtual world. Maybe your university already has (or is planning) a virtual campus in Second Life.

Immersion will have a large social impact on us in the future as we program computers for serious purposes. Imagine a computer program that helps a person drive a car, or fly a plane, or teach a class, or play tennis, or repair a human heart or get over certain phobias. There are some early examples of such programs in use today, and the concepts and skills used to develop those programs are quite similar to those that you’ve learned in this and in previous chapters.