Alice 3

How-to Guide
(Part 4 - Events)

Wanda Dann
Don Slater
Laura Paoletti
Dennis Cosgrove
Dave Culyba
Pei Tang

Cover artwork by Laura Paoletti, 2012.

Events for Interactive and Game Programming

Video: Events

EVENTS FOR INTERACTIVE & GAME PROGRAMMING

The Scene class defines a procedure named *initializeEventListeners*, as shown in Figure IV.1. The *initializeEventListeners* is similar to an Events editor, where **listeners for specific events** may be created**.** An overview of the *initializeEventListeners* procedure was provided in Part 3, Section 16 of this guide. You may wish to review Part 3, Section 16 as it provides a "surface" description of listeners and events, appropriate for interactive programming. For those want a more in-depth look at events and listeners, this section explores events and listener options from the perspective of game programming.

By default, the Scene class' *initializeEventListeners* tab has one built-in event listener, *addSceneActivationListener*, that tells Alice to listen for a mouse-click on the Run button event. "In the event that" the Run button is clicked, a runtime window is displayed and the current scene becomes active. When this event occurs, *myFirstMethod* is called (executed).

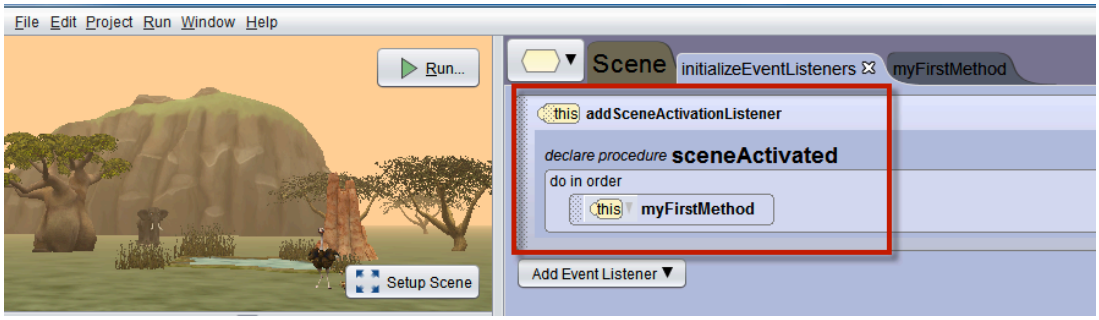

Figure IV.1 The Scene's initializeEventListeners tab

Additional event listeners may be created in the *initializeEventListeners* tab. To add a new event, click the **Add Event Listener** button. A popup menu displays four different categories of events, as shown in Figure IV.2. The event categories are **Scene Activation/Time, Keyboard, Mouse,** and **Position/Orientation**.

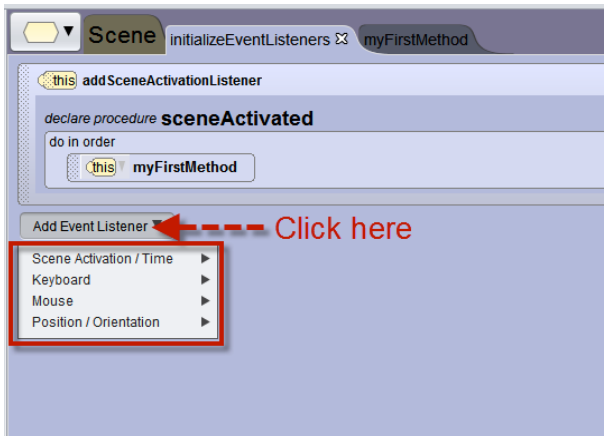

Figure IV.2 Event categories menu

Chapter 22: Scene Activation/Time Events

The **Scene Activation/Time** category has two menu options, as shown in Figure 22.1.

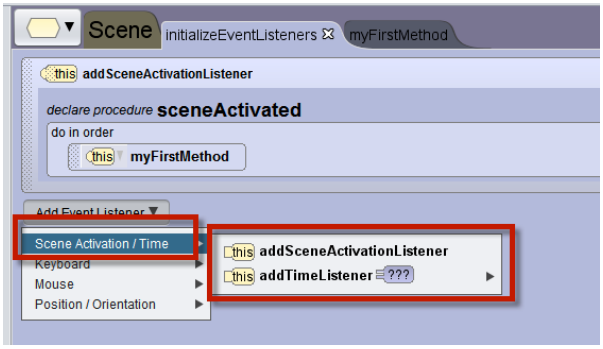

Figure 22.1 Scene Activation/Time listener menu options

Scene Activation

If **addSceneActivationListener** is selected, another **sceneActivated** listener is added to the editor, as shown in Figure 22.2. In this example, a statement was created in the second **sceneActivated** listener to play a *footsteps_walking* audio file. Now, when the Run button is clicked, both **sceneActivated** listeners will "fire" simultaneously. As a result, *myFirstMethod* will start to execute and the *footsteps_walking* audio file will start to play at the same time.

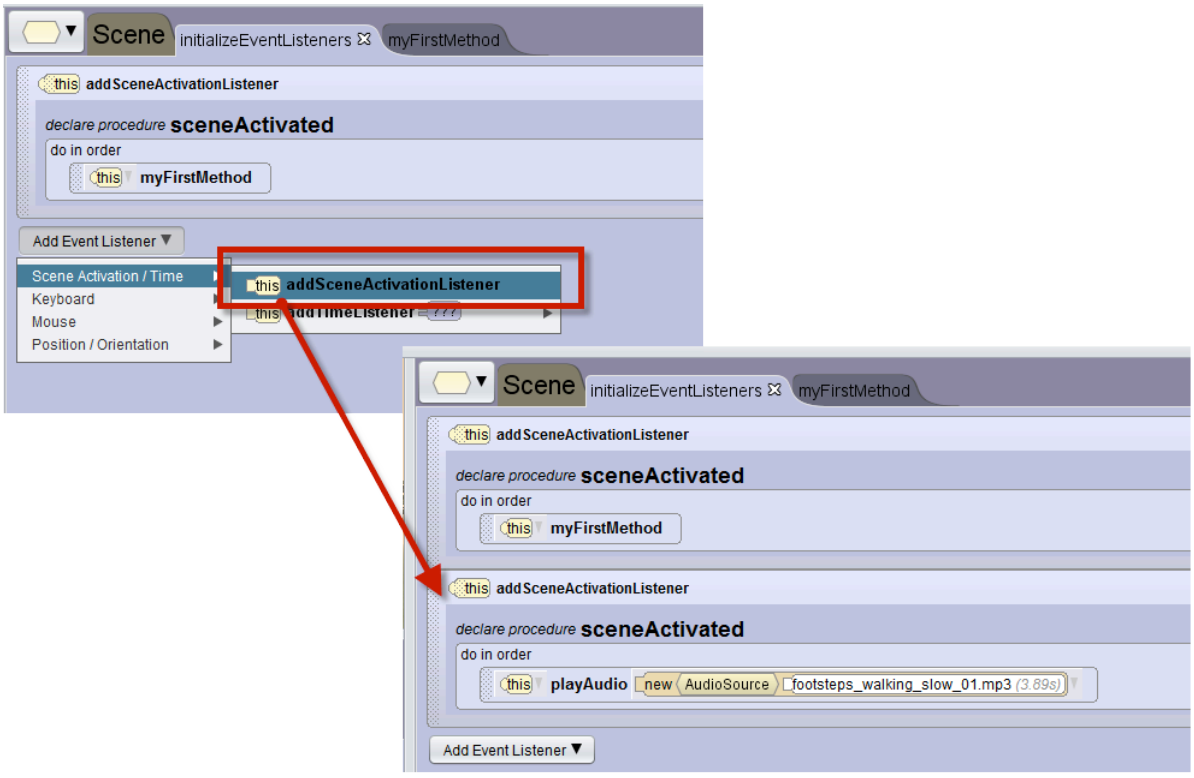

Figure 22.2 Two sceneActivated listeners, each with their own action

Time

If **addTimeListener** is selected, a menu cascades to select a time interval, as shown in Figure 22.3. In this example, a time interval of 0.25 seconds was selected and a statement was created in the **timeListener** to play a *footsteps_walking* audio file. Now, when the Run button is clicked, *myFirstMethod* will start running and after 0.25 seconds has elapsed, the *footsteps_walking* audio file will start playing.

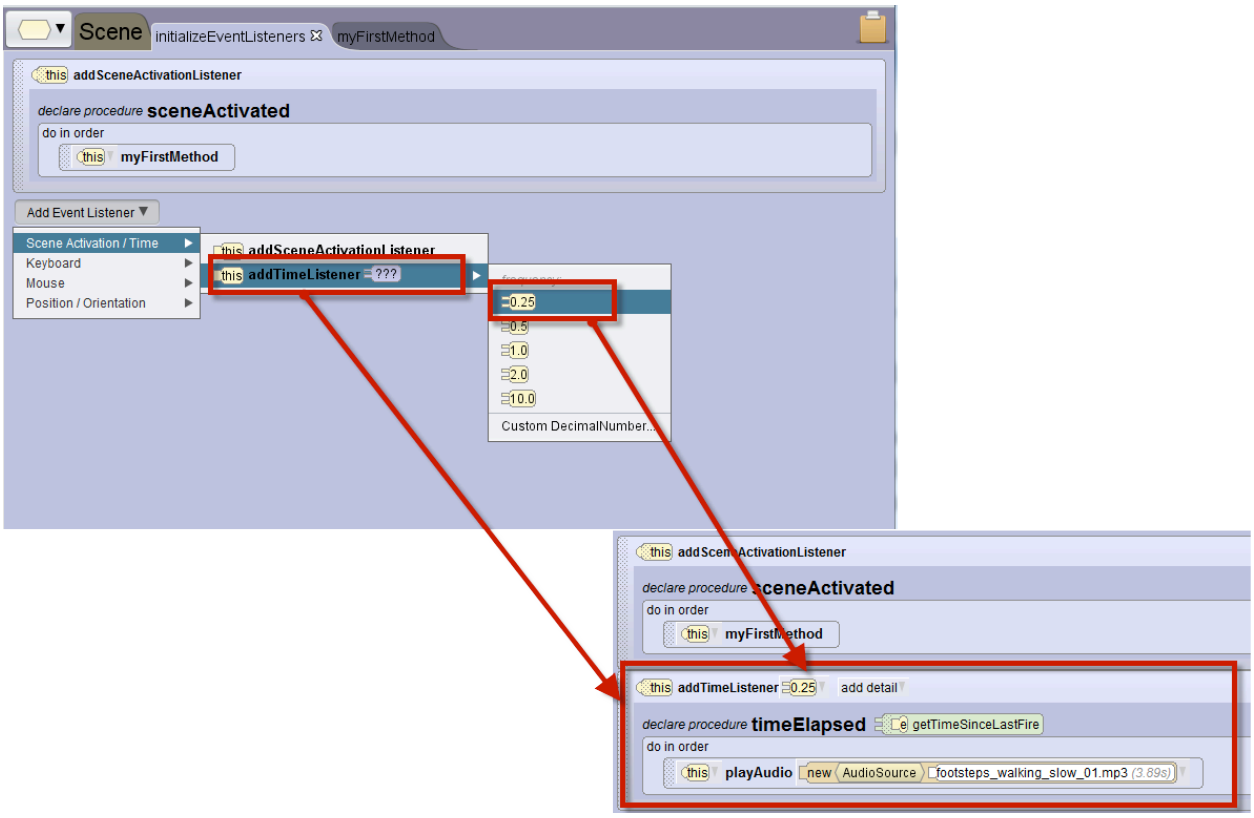

Figure 22.3 Time event listener

Single vs. Multiple Events

The **sceneActivated** and **timeElapsed** listeners expect an event to occur only once. That is, for the **sceneActivated** listener it is expected that the user will click on the Run button only once (to start the execution of the program). Likewise, for the **timeElapsed** listener it is expected the time interval will elapse and the action will occur just once.

Of course, sometimes you may expect that an event may occur several times while a program is running. To handle events that may occur multiple times, Alice provides a multiple event policy option. To set the multiple event policy, clicking on the add detail button in the listener header's signature heading and select one of the menu options, as shown in Figure 22.4.

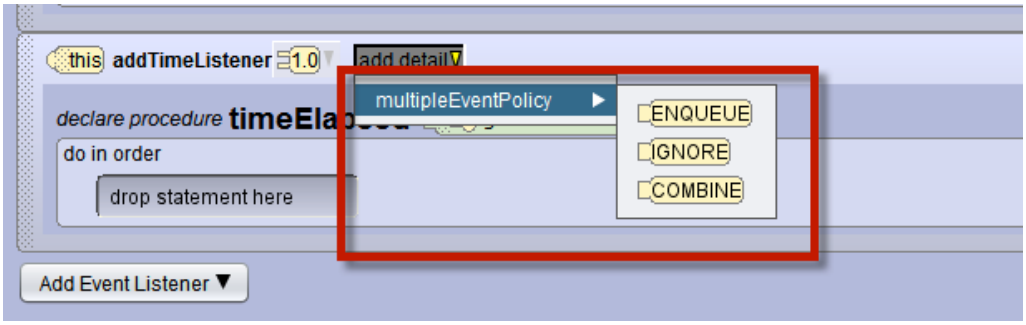

Figure 22.4 Multiple event policy menu

- IGNORE – just do the action once (default setting).
- ENQUEUE – repeat the action each time the event occurs, in succession (wait for the previous action to finish before doing it again)
- COMBINE – repeat the action each time the event occurs, concurrently (don't wait for the previous action to finish)

As an example, suppose ENQUEUE is selected for the **timeElapsed** listener, as shown in Figure 22.5. Now, when the Run button is clicked, Alice will wait 0.25 seconds and then play the *footsteps_walking* audio until the audio is finished, wait another 0.25 seconds and then play the audio again, … and repeat this action again and again until the program ends.

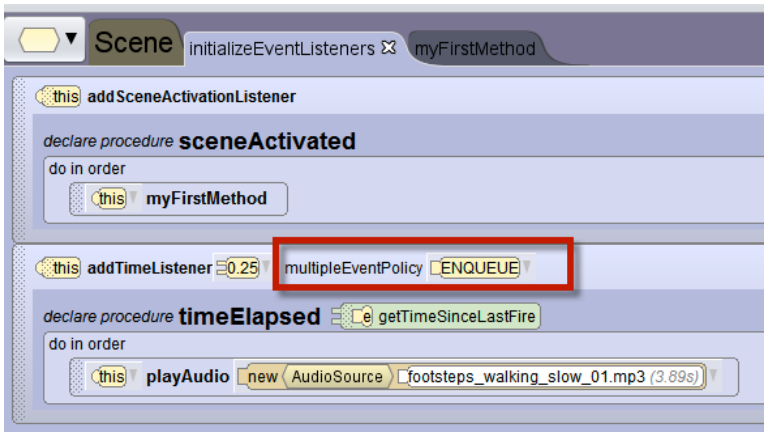

Figure 22.5 ENQUEUE option for timeElapsed listener

Chapter 23: Keyboard Events

Video: Keyboard Events

The **Keyboard** category has four menu options, as shown in Figure 23.1. The first three options (**addKeyPressListener**, **addArrowKeyPress** and **addNumberKeyPress**) listeners all work in the same way**.** The illustration shown here is for **KeyPressListener** but can be applied to any one of the three.

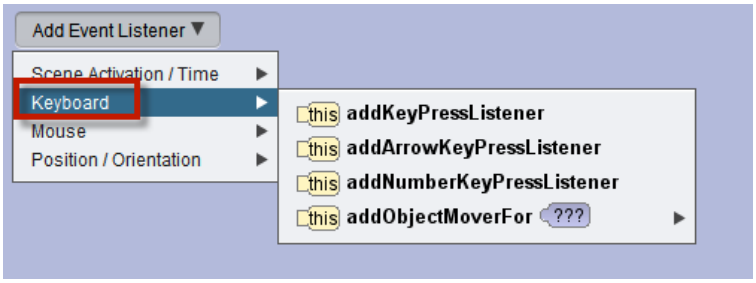

Figure 23.1Keyboard event listeners

KeyPress

When **addKeyPressListener** is selected, a **keyPressed** listener is added to the editor, as shown in Figure 23.3.

Figure 23.3Keyboard event listener

The **keyPressed** listener has four functions on the event (*e*) that may be used within its code block to retrieve information about which key was pressed:

- *e.isLetter* returns a boolean value that is true if the key pressed is a letter of the alphabet and false, otherwise.
- *e.isDigit* returns a boolean value that is true if the key pressed is a digit (0 – 9) and false, otherwise.
- *e.getKey* returns a Char value representing the key pressed

- *e.isKey* (**key**) returns a boolean value that is true if the key press is equal to the **key** argument

For example, in Figure 23.4 an If statement has been added to the keyPressed listener that calls *e.isKey* with the letter 'L' as the argument. If the user has pressed a key and the key is 'L', the elephant will turn left 0.25 revolutions.

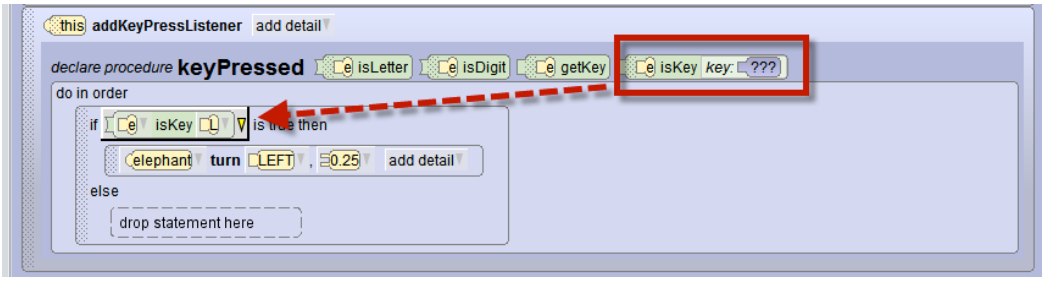

Figure 23.4 Using the isKey function within a keyPressed listener

As a more generic example, in Figure 23.5 an If statement has been added to the keyPressed listener that calls *e.isLetter*. Additional If statements are nested within, to check whether the letter is 'J' or 'K'. If the letter is J, the elephant will move left 1 meter. Else, if the letter is K, the elephant will move right 1 meter. If some other letter or key is pressed, no action is taken.

Figure 23.5 Using multiple functions within a keyPressed listener

Held key policy

Keyboard events are geared to work with a single key press, but if you expect that the user may want to hold down a key, it is possible to set the **heldKeyPolicy** to control how the event is handled, as shown in Figure 23.6.

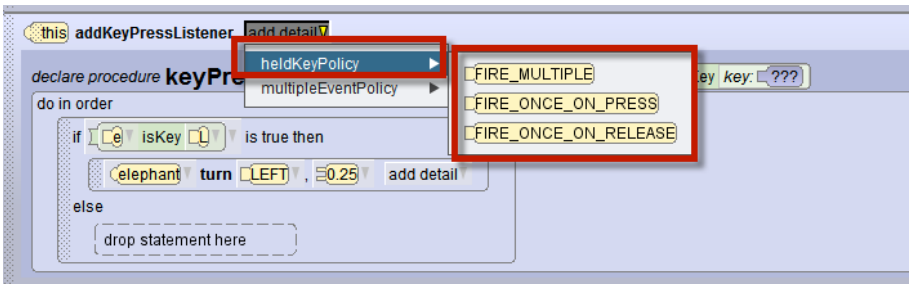

Figure 23.6 Selecting a heldKeyPolicy

- FIRE_MULTIPLE – repeatedly perform the action until the key is released
- FIRE_ONCE_ON_PRESS – when the key is first pressed down, perform the action once only (default setting)
- FIRE_ONCE_ON_RELEASE – the the key is released, perform the action once only

In practice, most programmers set either a held key policy or a multiple event policy – but not both. It is possible, however, to set choices for both policies for the same event. Predicting the behavior takes a bit of logic, but here is an example:

Let's say you have selected FIRE_MULTIPLE as the **heldKeyPolicy** and ENQUEUE as the **multipleEventPolicy.** When the user holds down a key, Alice will queue up the events and fire one after another until all the actions are eventually performed (which might cause events to fire long after the user stops holding down the key).

Object Mover

The fourth option for Keyboard events is **addObjectMoverFor**. When this option is selected, an **addObjectMoverFor** listener is added to the editor, as shown in Figure 23.7. In this example, the elephant was selected as the object to be controlled using arrow key presses. When the user clicks an arrow key, the elephant object will move in the direction the arrow points (Forward, Backward, Right, and Left). The FIRE_MULTIPLE **heldKeyPolicy** is automatically in effect for this event listener and cannot be reset.

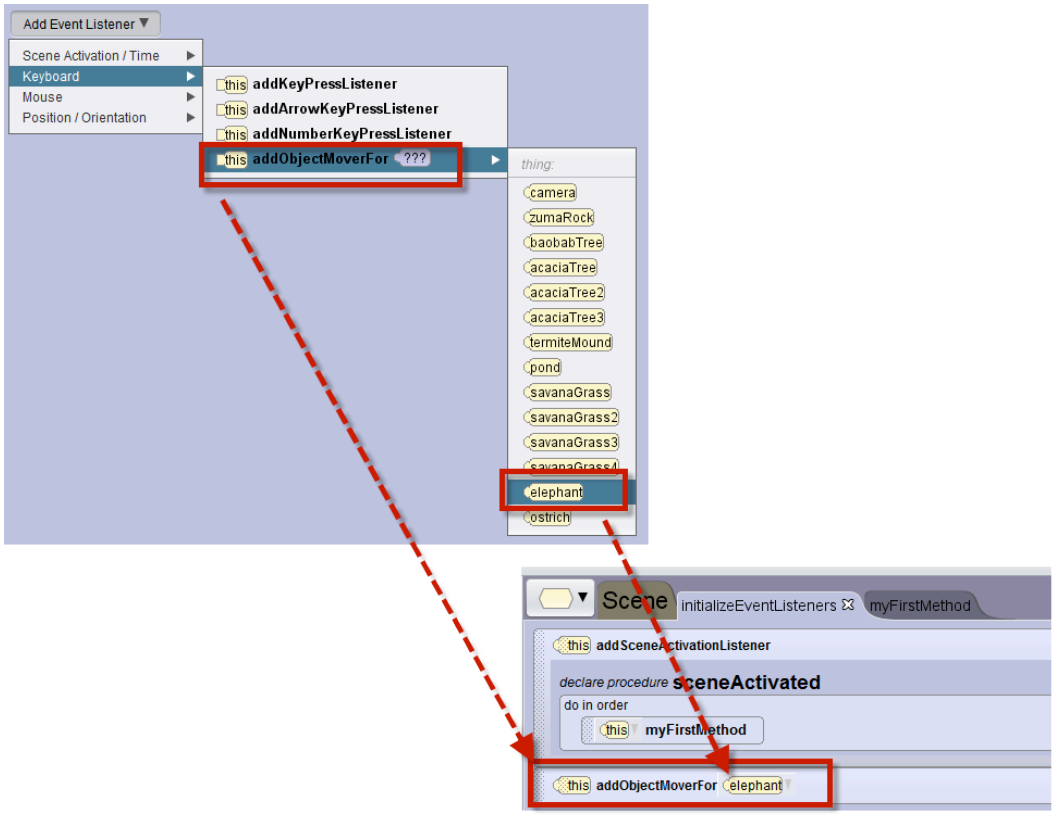

Figure 23.7 Keyboard event listener

Chapter 24: Mouse events

The **Mouse** events category has three menu options, as shown in Figure 24.1.

Figure 24.1 Mouse click event listeners

Mouse click on object

When **addMouseClickOnObjectListener** is selected, a **mouseClicked** event listener is created, as shown in Figure 24.2. The mouseClicked event listener fires when the mouse is clicked on any object in the scene. **NOTE: A mouse click on the scene's ground surface or atmosphere is ignored by this listener.**

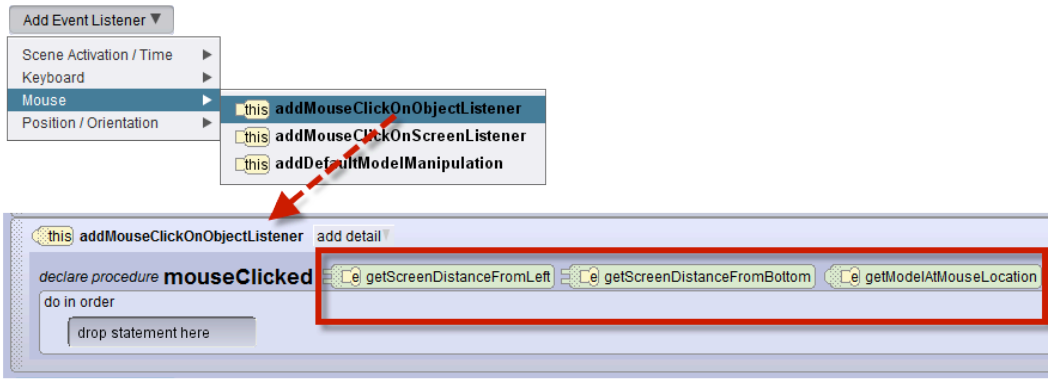

Figure 24.2 The mouseClicked event listener

The **mouseClicked** listener has three functions on the event (*e*) that may be used within its code block to retrieve information about the object that was clicked:

- *e.getScreenDistanceFromLeft* returns a decimal (double) value that is the x-coordinate for the location of the clicked object.
- *e.getScreenDistanceFromBottom* returns decimal (double) value that is the y-coordinate for the location of the clicked object.
- *e.getModelAtMouseLocation* returns a link to the clicked object, of type SModel

As an example of using the mouseClicked event listener, we added statements to the mouseClicked code block in Figure 24.3. The first statement declares an SModel variable named *obj*. (SModel provides maximum level of compatibility with all objects in the scene that might be

clicked.) The *obj* variable is assigned the clicked object as the result of a call to the *getModelAtMouseLocation* function. The second statement tells that object to turn to face the camera.

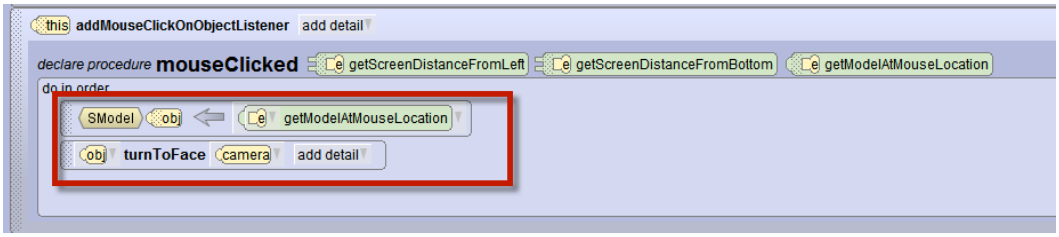

Figure 24.3 Calling a function to determine which object was clicked

The mouseClicked event listener's **add detail** button has two options (**multipleEventPolicy** and **setOfVisuals),** as shown in Figure 24.4.

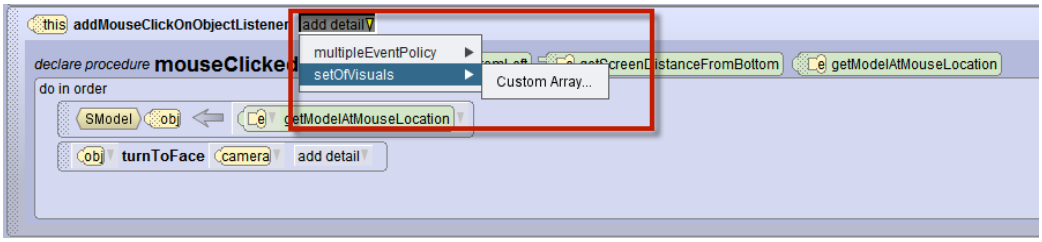

Figure 24.4 Add detail options

One of the detail options is the **multipleEventPolicy**, which works as described earlier in this section of the How-To guide. You may wish to review that section, see above.

The second option is **setOfVisuals**, which allows you to create a custom array of one or more objects for which this mouse click event listener will work. For example, if setOfVisuals → Custom Array is selected, a window pops up where you may select one or more objects from the scene, as shown in Figure 24.5.

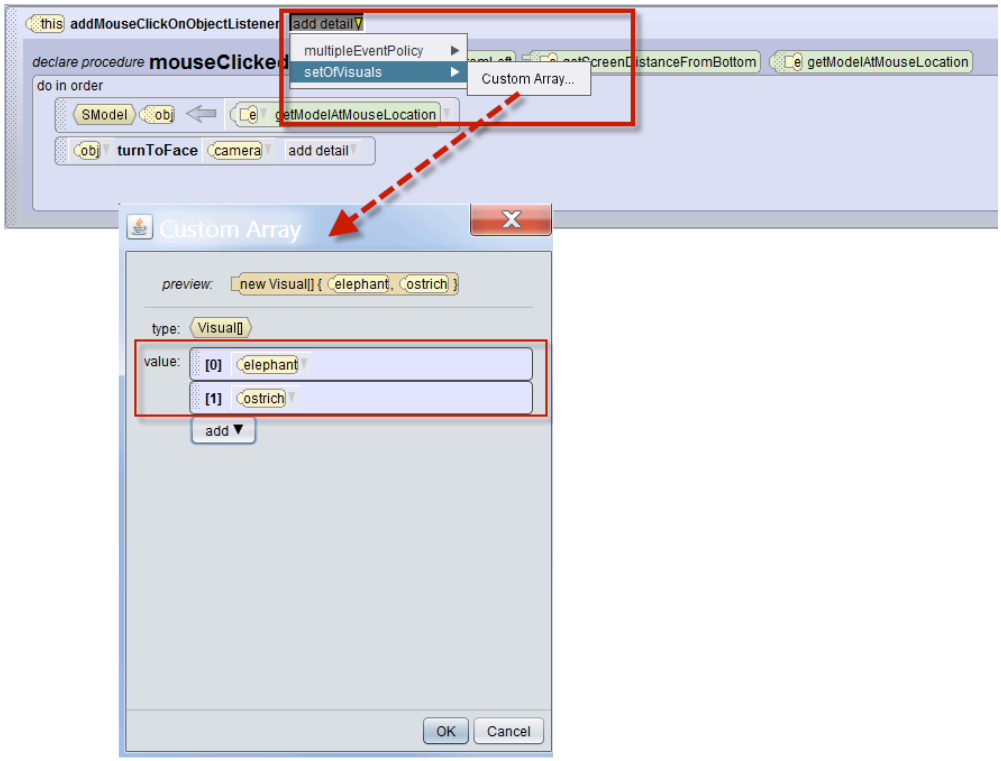

Figure 24.5 Creating a custom array of objects for a listener

After the array has been created, the list of objects in the array is displayed in the listener's code block, as shown in Figure 24.6. In this example, an elephant and an ostrich objects are selected for the *setOfVisuals* array. Alice will listen for a mouse click and will fire an event only if the elephant or ostrich is clicked. A mouse click on any other object in the scene will automatically be ignored by this listener.

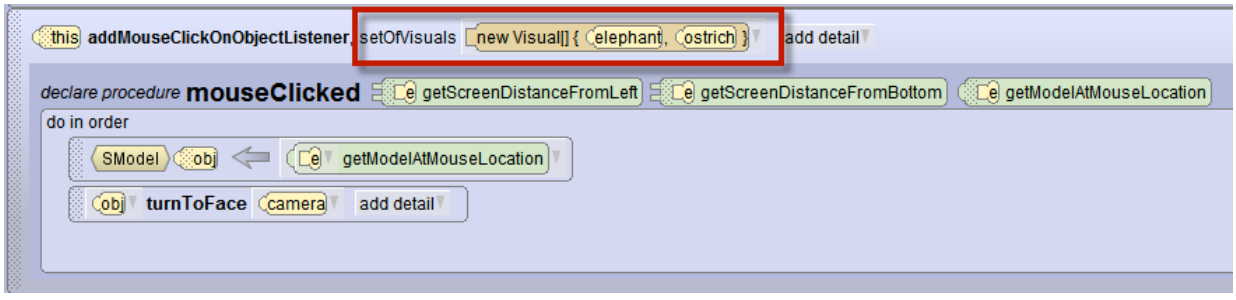

Figure 24.6 The setOfVisuals lists click-able objects

Mouse click on Screen

When **addMouseClickOnScreenListener** is selected, a **mouseClicked** event listener is created, as shown in Figure 24.7. This mouseClicked event listener fires when the mouse is clicked anywhere on the screen. No custom array option is available and so it cannot be restricted to specific objects.

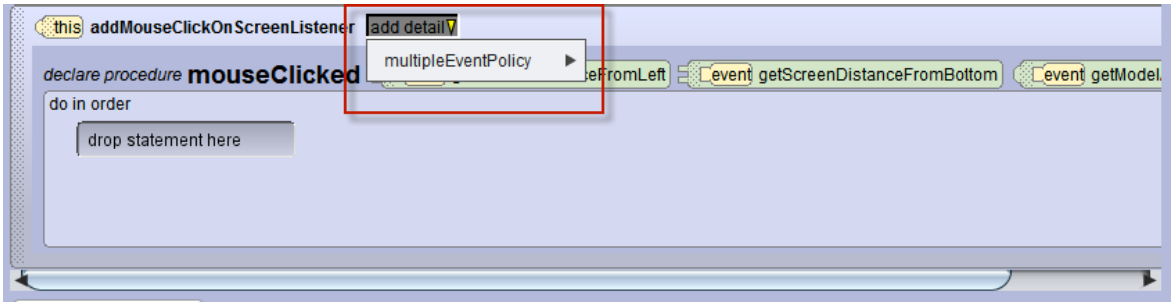

Figure 24.7 A mouseClicked event for anywhere on the scene

Use mouse to move object

The **addDefaultModelManipulation** is a mouse listener that allows the user to use the mouse to drag an object around the screen. To create this listener, select the **Mouse** event listener category and then **addDefaultModelManipulation** in the cascading menu, as shown in Figure 24.8.

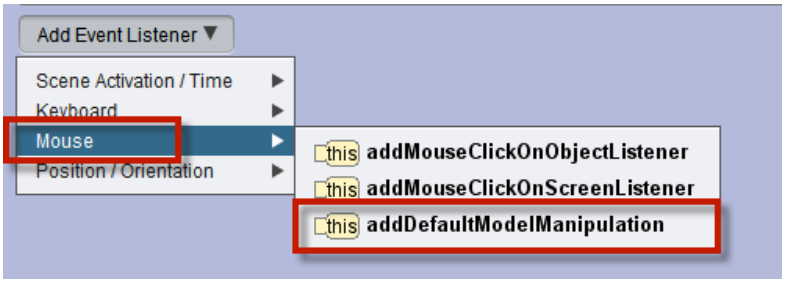

Figure 24.8 Select **Mouse/addDefaultModelManipulation** *listener*

The resulting event listener can be seen in Figure 24.9.

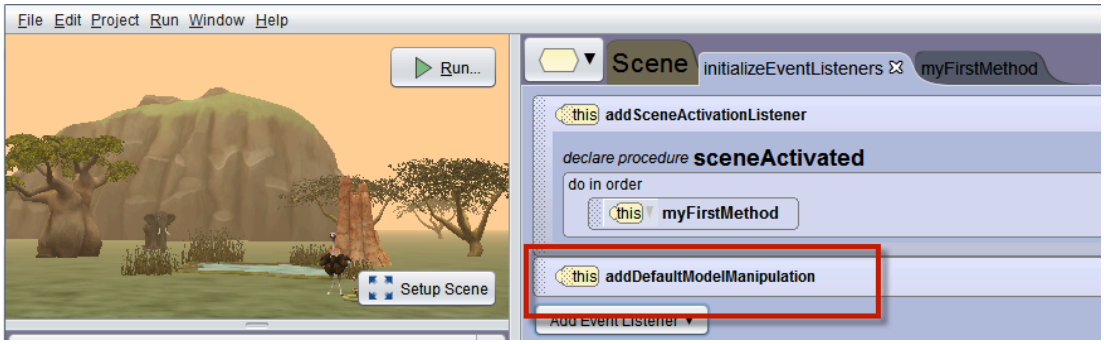

Figure 24.9 **addDefaultModelManipulation** *event listener code*

The **addDefaultModelManipulation** listener fires when the mouse is clicked and held on an object in the runtime window. The mouse can be used drag an object around in the scene. Any object within the scene can be pulled around the scene as the animation is running. For example, the mouse could be used to move the elephant around in the scene shown in Figure 24.10.

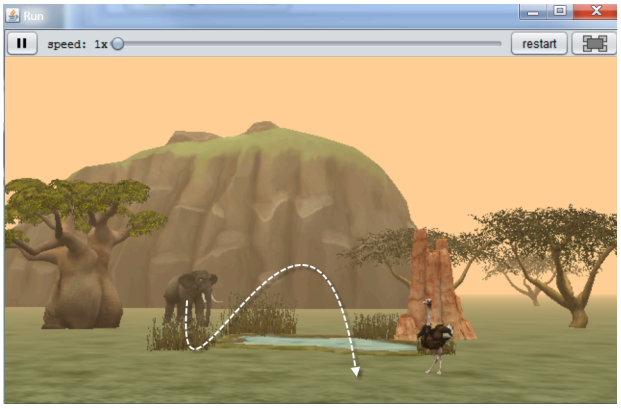

Figure 24.10 The mouse can move objects around the scene at runtime

Chapter 25: Position/Orientation events

The **Position/Orientation** events category has nine menu options, as shown in Figure 25.1. All but the last menu option are listed in pairs of listeners -- one for *start* (or *enter*) and one for *end* (or *exit)*. For example, the collision event has a pair of listeners: **addCollisionStartListener** and **addCollisionEndListener.** The following discussion explains how to use the event listeners in pairs. Because the last menu option, **addPointOfViewChangeListener**, does not have a start and end version, it will be covered as a single item.

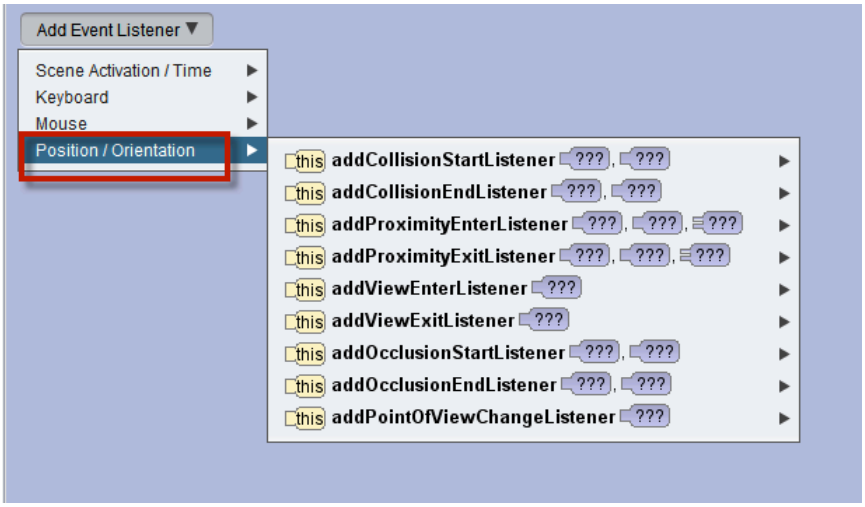

Figure 25.1 Position and Orientation event listeners

Video: Collision Detection

Collision start and end

In the real world around us, a collision occurs when one object "physically touches" another object. For example, a car being driven down the highway might veer off the road and hit a tree. We say, "The car *collided* with a tree." Of course, objects in Alice are virtual, so they don't "physically touch" one another. Instead, a collision listener detects a "virtual touch" when some portion of one object is in the same location in the world as some portion of another object. For example, a person's hand might be in the same location in the world as a dog's head. In other words, the person's hand is touching the dog's head.

To create a listener for a collision between two objects, select the **Position/Orientation** event listener category and then **addCollisionStartListener** in the cascading menu, as shown in Figure 25.5.

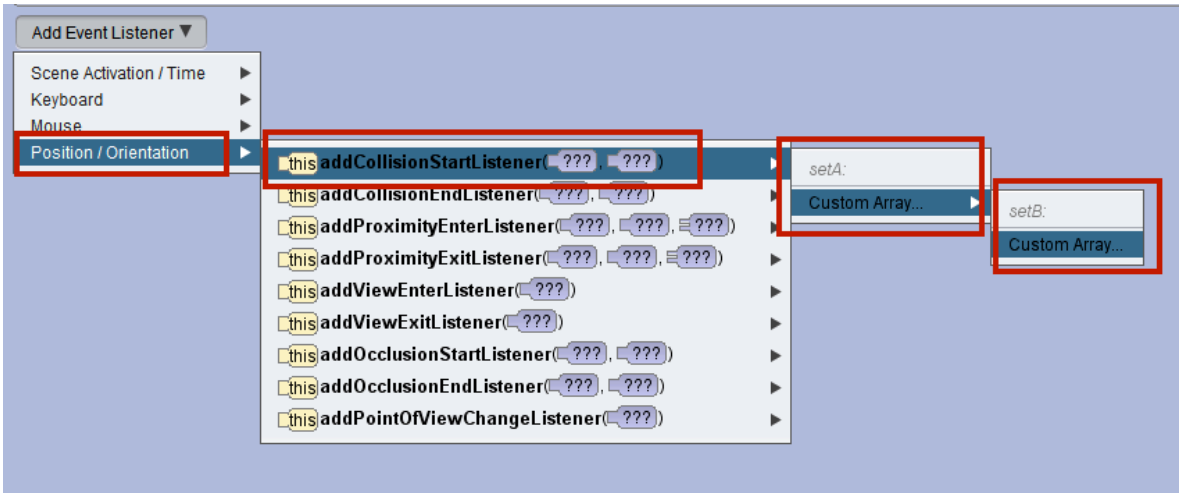

Figure 25.5 Create a CollisionStartListener

Two arguments are needed: setA (an array of one or more objects that might collide) and setB (another array of one or more objects that might collide). For simple collision detection, the arrays are most likely to contain only one object each. For example, in Figure 25.6 setA contains only the soccerBall and setB contains only the ostrich. With these settings, only a collision between the soccerBall and the ostrich will fire a collision event.

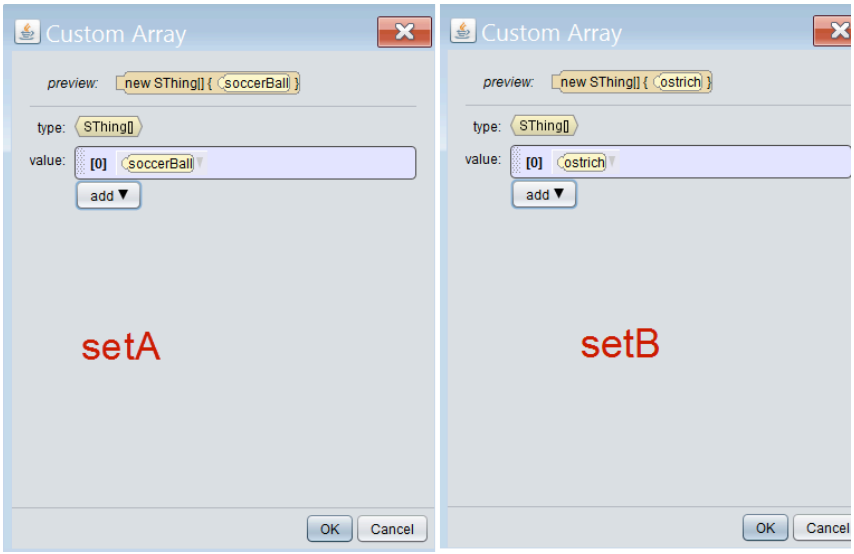

Figure 25.6 One object in each set, only one possible collision

More complex collision detections often have more than one object in a set, which increases the number of possible collisions. For example, in Figure 25.7, setA has only a soccerBall but setB has an ostrich and an elephant. In this example, two collisions are possible – a collision between the soccerBall and the ostrich and a collision between the soccerBall and the elephant.

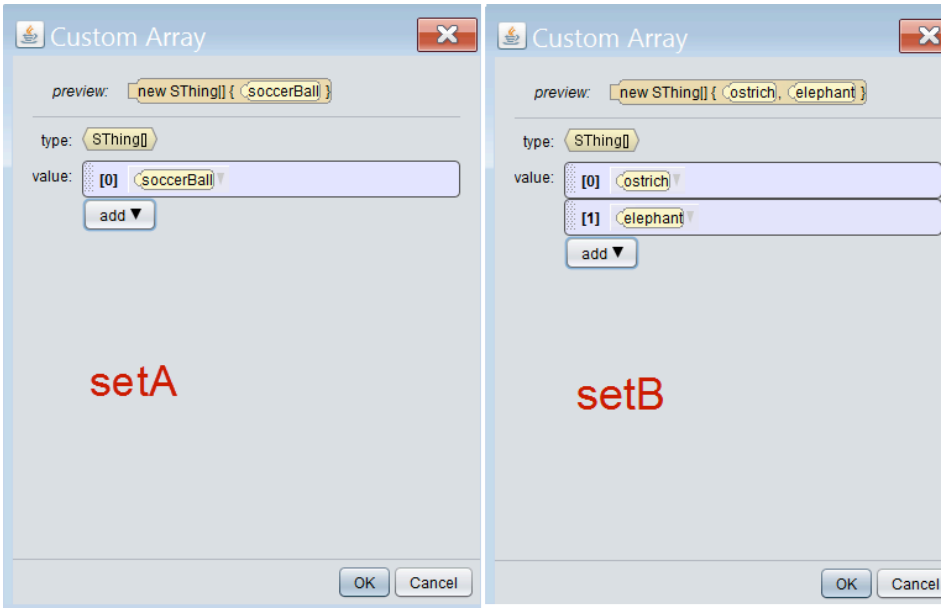

Figure 25.7 Two possible collisions

Illustration of how a collision is detected

To detect a collision, Alice computes a bounding-box shape that encloses an object. For example, in Figure 25.8 a computed bounding box can be seen surrounding an ostrich.

Figure 25.8 Computed bounding box surrounding an ostrich

Suppose the elephant uses its trunk to toss a ball to the ostrich. If the ball enters the bounding box around the ostrich, the ball starts to collide with the ostrich as shown in Figure 25.9 and the **addCollisionStartListener** will fire. It is important to note a collision end listener works similarly, except it would fire when the ball leaves the bounding box surrounding the ostrich.

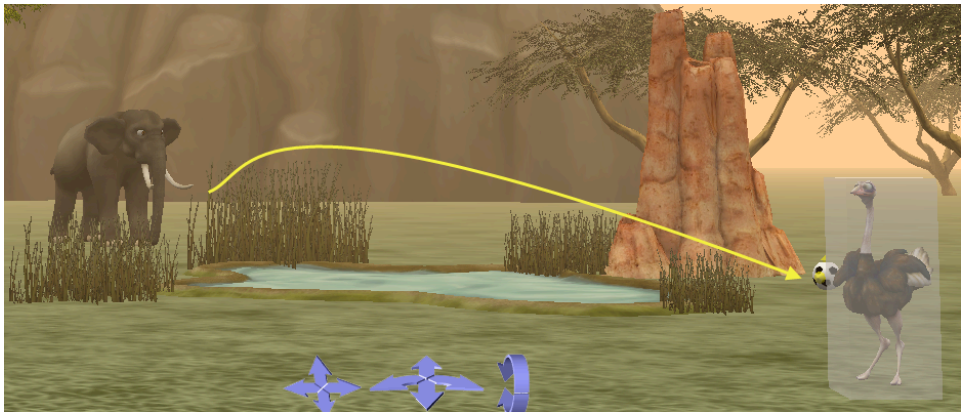

Figure 25.9 Ball collides with ostrich

It is important to note that this collision detection technique is not a perfect means of detecting collision. It works well in most situations – but it is possible to occasionally misfire. As an example, consider the situation shown in figure 25.10. In this screenshot, you can see that the soccerBall has entered the bounding box space, but it is up a little higher off the ground than in

the previous screenshot. So, it has entered the space surrounded by the bounding box but it hasn't yet collided with a part of the ostrich's body. The collision listener will fire! But, the sharp-eyed viewer might realize that the ball hasn't quite reached its target.

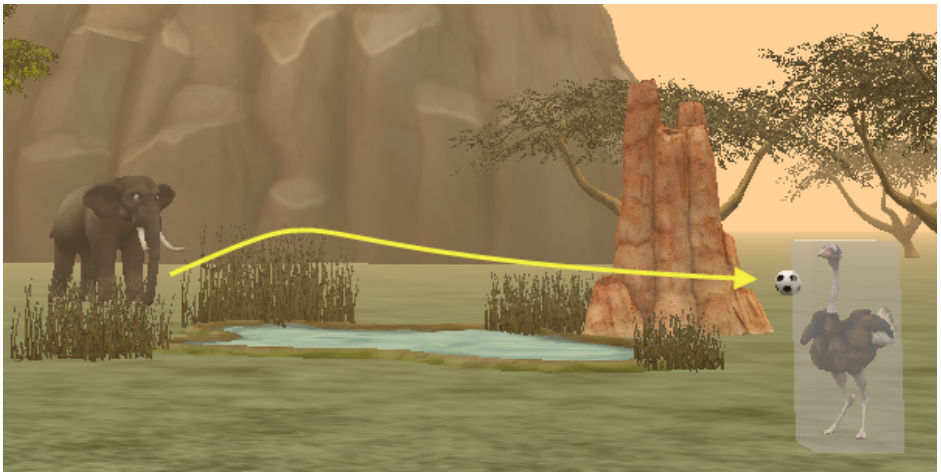

Figure 25.10 Collision detected, but a little too soon

Video: Proximity Detection

Proximity enter and exit

A proximity listener detects when an object enters or exits a boundary space around another object. Unlike collision, the two objects do not have to actually come in direct contact with one another. A proximity listener will fire when another object enters or exits a bounded space surrounding a given object. For example, an electric dog fence sets off a vibrator on a dog's collar if the dog crosses over the outer boundary of the owner's yard.

To create a proximity listener, select the **Position/Orientation** event listener category and then **addProximityEnterListener** in the cascading menu, as shown in Figure 25.11.

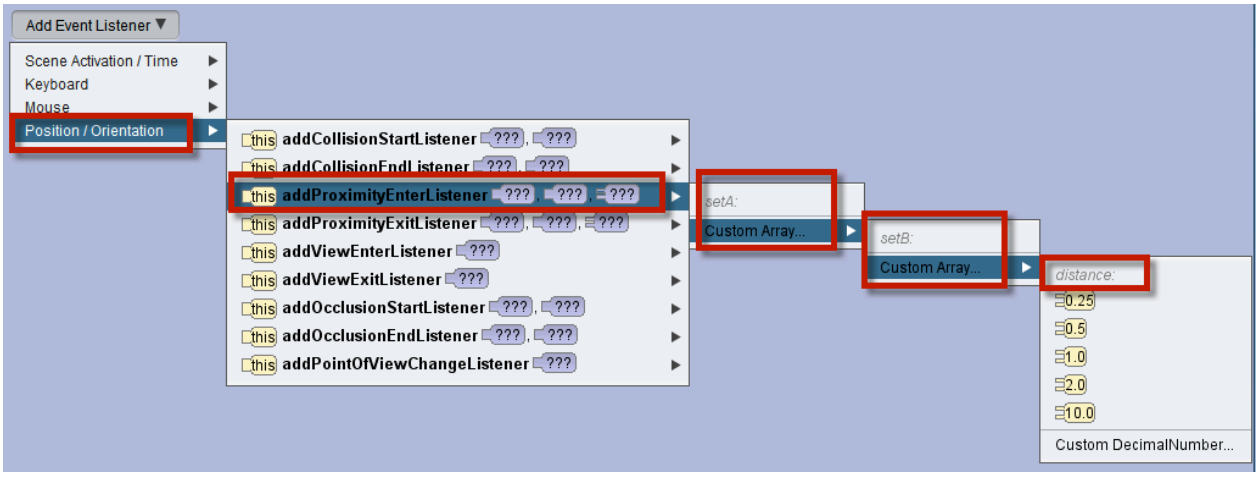

Figure 25.11 Create a ProximityEnterListener

Two arrays of objects are needed: setA (an array of one or more objects that might enter a proximity boundary) and setB (a target set of one or more objects, each having a bounded space that might be entered). For example, in Figure 25.12 setA contains elephant and the ostrich and setB contains only the pond.

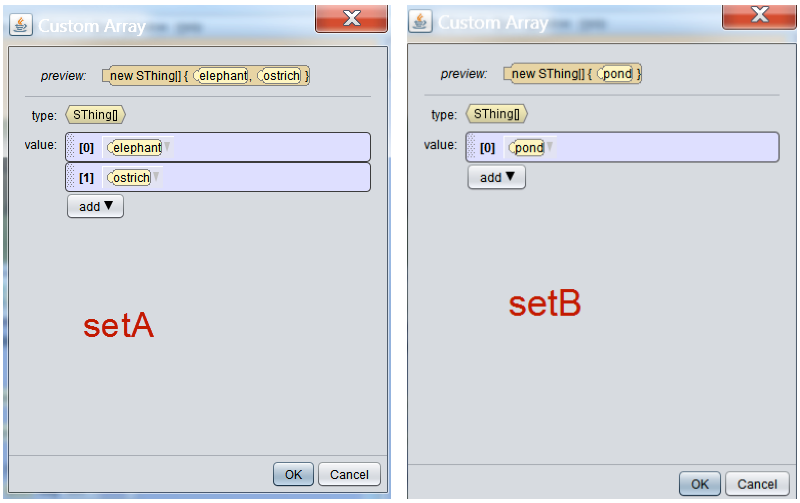

Figure 25.12 Two sets of objects for proximity enter listener

The third argument for creating a proximity listener is a distance that defines the bounded space around each object in setB. As shown in Figure 25.13, a distance of 5 meters is selected. In this example, when the elephant or ostrich wanders within 5 meters of the pond, that object is identified as a thirstyObject. (For brevity, the code is not shown here. However, the idea is to have the thirstyObject wade into the pond for a drink of water.) It is important to note a proximity exit listener works similarly, except it would fire when either the elephant or ostrich leaves the bounded area surrounding the pond.

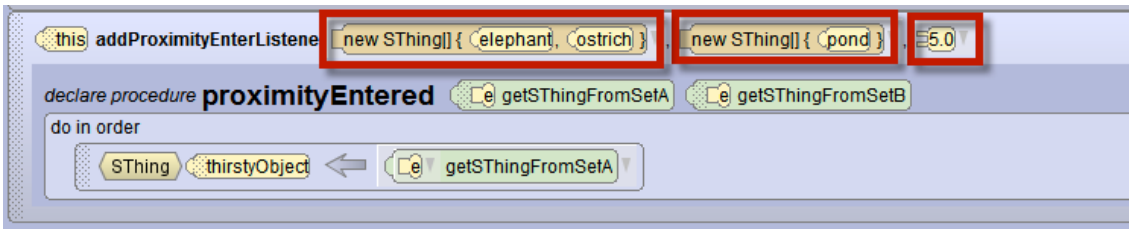

Figure 25.13 Example: Event listener for proximityEntered

Proximity Enter/Exit Multiple Events

The *proximity enter and exit* listeners work with arrays of objects. Because multiple objects are being tracked, it is possible for more than one object to enter or exit a bounded area at the same time. The multiple event policy (discussed previously for Time events), can be set to:

- IGNORE – just do the action once (default setting).

- ENQUEUE – repeat the action each time the event occurs, in succession (wait for the previous action to finish before doing it again)
- COMBINE – repeat the action each time the event occurs, concurrently (don't wait for the previous action to finish)

View enter and exit listeners

A view listener detects when an object enters-into or exits-from the view of the camera. In this discussion, we use "off-screen" to describe an object that is not currently within view of the camera. A view enter listener will fire when an off-screen object moves into the camera's view. A view exit listener will fire when an object currently in view by the camera moves off-screen.

To create a view enter listener, select the **Position/Orientation** event listener category and then **addViewEnterListener** in the cascading menu, as shown in Figure 25.14.

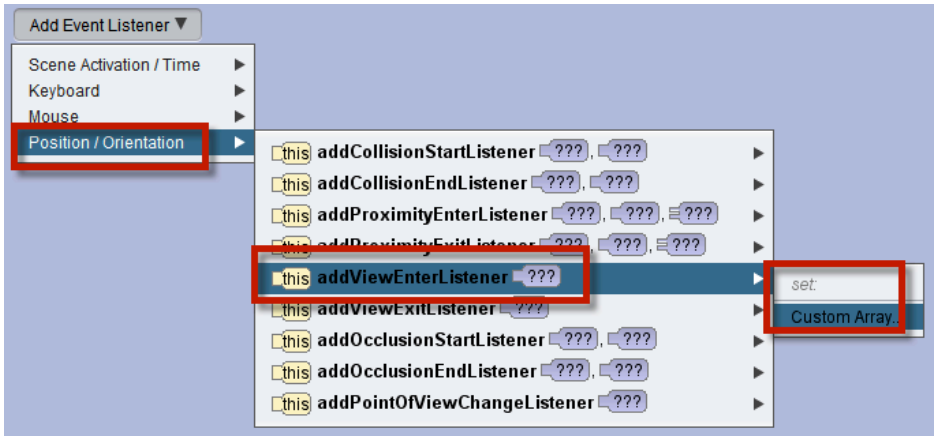

Figure 25.14 Create a ViewEnterListener

An array (set) is used to track the objects that might enter or exit the scene space currently in view of the camera. For example, in Figure 25.15 the set of objects contains elephant and the ostrich.

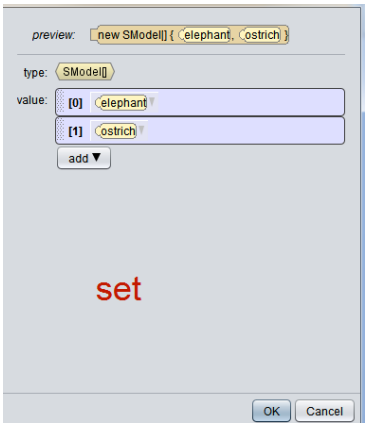

Figure 25.15 An array of objects that might enter and leave the scene view

As shown in Figure 25.16, the object entering the scene view can be selected to perform some action. In this example, the entering object plays an audio of footsteps walking slowly. It is important to note a view exit listener works similarly, except it would fire when either the elephant or ostrich leaves the area currently in view by the camera.

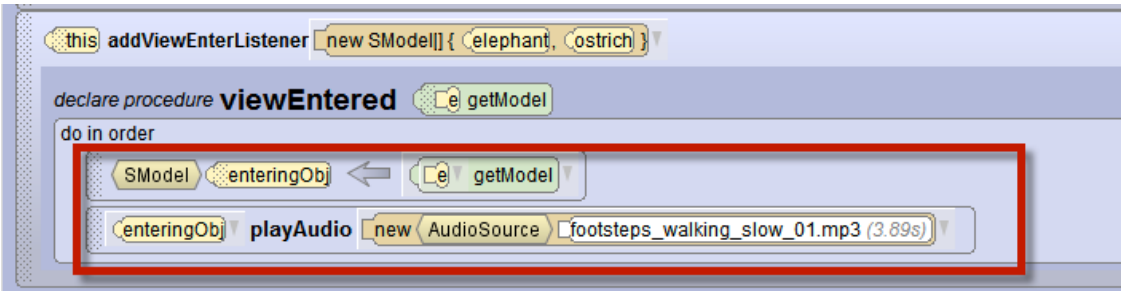

Figure 25.16 Example: Event listener for viewEntered

View Enter/Exit Multiple Events

The camera's *view enter and exit* listeners work with arrays of objects. Because multiple objects are being tracked, it is possible for more than one object to enter or exit the camaera's viewing area at the same time. The multiple event policy (discussed previously for Time events), can be set to:

- IGNORE – just do the action once (default setting).
- ENQUEUE – repeat the action each time the event occurs, in succession (wait for the previous action to finish before doing it again)
- COMBINE – repeat the action each time the event occurs, concurrently (don't wait for the previous action to finish)

Occlusion start and end listeners

An occlusion start listener detects when an object becomes (at least partially) hidden by another object. An occlusion end listener detects when an object which was (at least partially) hidden by another object becomes totally visible.

To create an occlusion start listener, select the **Position/Orientation** event listener category and then **addOcclusionStartListener** in the cascading menu, as shown in Figure 25.17.

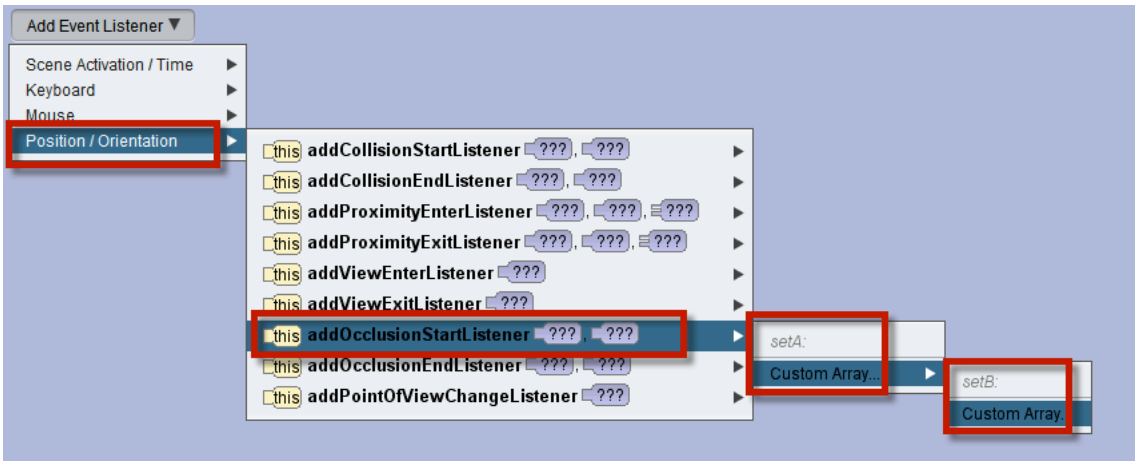

Figure 25.17 Create an OcclusionStartListener

Two arrays of one or more objects is needed: setA (an array of one or more objects that might be in the foreground) and setB (an array of one or more objects that might be hidden in the background). For example, in Figure 25.18, setA contains the baobabTree and the termiteMound while setB contains the elephant and the ostrich.

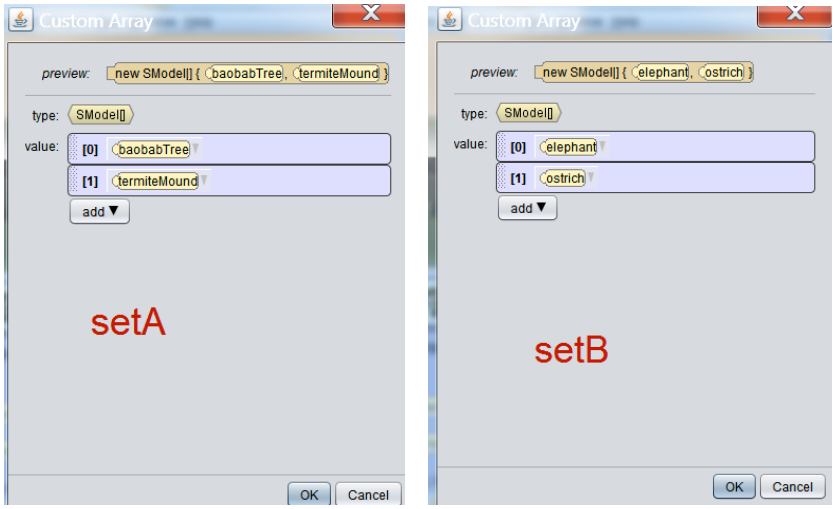

Figure 25.18 Potentional foreground (setA) and background (setB) objects

During the animation, when one object is (at least partially) hidden by another object, the occlusion start listener fires. In the listener code, as shown in Figure 25.19, two functions are available – one to determine which object is in the foreground and which is in the background. Code can then be written to move the hidden object back into view or to perform some other action as part of the story or game. It is important to note an occlusion exit listener works similarly, except it would fire when either the elephant or ostrich (which was hidden) returns to view (and is no longer hidden).

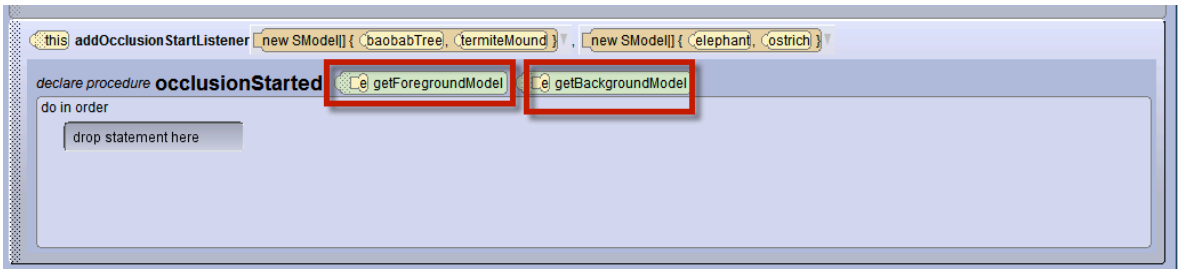

Figure 25.19 Functions to get foreground and background, when listener fires

Occlusion start and end Multiple Events

The *occulsio start and end* listeners work with arrays of objects. Because multiple objects are being tracked, it is possible for more than one object to occlude another at the same time. The multiple event policy (discussed previously for Time events), can be set to:

- IGNORE – just do the action once (default setting).
- ENQUEUE – repeat the action each time the event occurs, in succession (wait for the previous action to finish before doing it again)
- COMBINE – repeat the action each time the event occurs, concurrently (don't wait for the previous action to finish)

Point of view change listeners

A point of view change listener detects when an object changes its point of view (location and orientation). For example, in a gaming application, the camera might be moved around and you might want to reset the camera to a specific location before the next action begins.

To create a point of view change listener, select the **Position/Orientation** event listener category and then **addPointOfViewChangeListener** in the cascading menu, as shown in Figure 25.20.

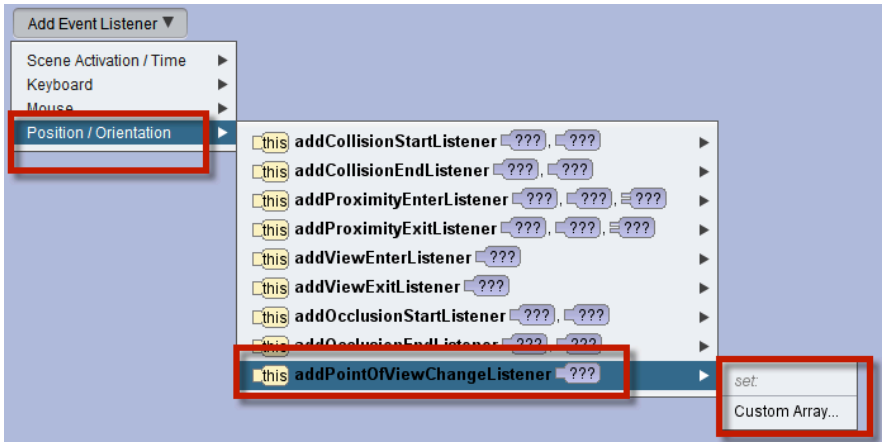

Figure 25.20 Create PointOfViewChangeListener

An array of one or more objects is needed: set (an array of one or more objects that might be have its point of view changed during game play or other interactive actions). For example, in Figure 25.21, set contains only the camera. Of course, the array could contain several objects, all with different actions to occur when their point of view changes.

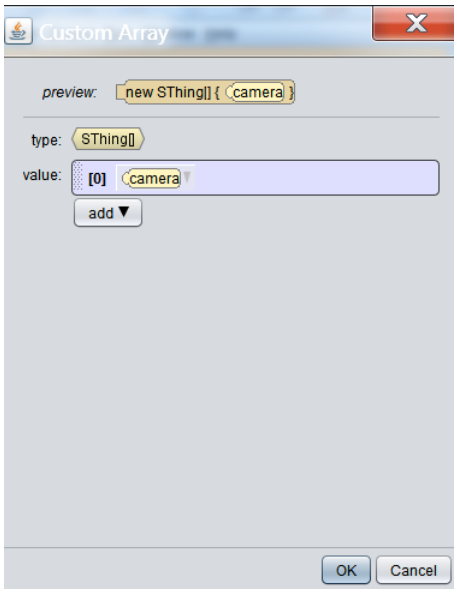

Figure 25.21 An array containing one element, the camera

In this example, as shown in Figure 25.22, each time the camera's point of view changes, it is reset (by calling the camera's *moveAndOrientTo* procedure) to the *cameraMarker1* position (which holds the original camera location and orientation).

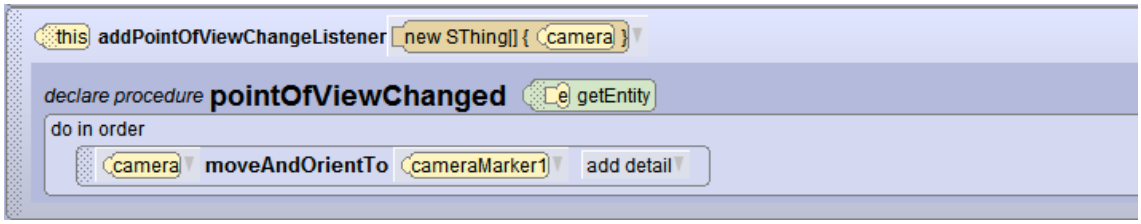

Figure 25.22 Example: Event listener for point of view change