ALICE 3

How To Guide

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

© 1st Edition Copyright: May, 2012, 2nd Edition Copyright: September, 2014

This material may not be copied, duplicated, or reproduced in print, photo, electronic, or any other media without express written permission of the authors and publisher.

Cover artwork by Laura Paoletti, 2012.

Welcome to Alice 3. Alice 3 has been under development since late 2007. A Beta version was made available for adventuresome souls in 2009. This guide has been prepared for release in-sync with the first official (non-beta) release in 2012. This publication also marks the 5th anniversary of the Last Lecture presented by Dr. Randy Pausch, the founder of the Alice Project at Carnegie Mellon University.

The Alice Team

The Alice team consists of a group of software engineers, character artists, professors, and authors. A proud distinction of this team is the devotion each team member has for Alice. The life and breath of Alice software is dependent on the members of our creative and energetic development team:

Dennis Cosgrove, Lead architect and Senior Software Engineer
Dave Culyba, Software Engineer
Matthew May, Junior Software Engineer
Laura Paoletti, Character Artist
Pei Tang, Character Artist

The instructional support materials, including this How-To guide are prepared and tested by members of our authoring and curriculum team:

Wanda Dann, Carnegie Mellon University, wpdann@cs.cmu.edu
Don Slater, Carnegie Mellon University, dslater@cmu.edu

Acknowledgements

The Oracle Foundation, the Sun Microsystems Foundation, the Hearst Foundation, and Electronic Arts have contributed support for the development of the Alice 3 system, for which we are deeply grateful.

The content in this guide is based upon work partially supported by the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Our deep gratitude goes to early testers and users of Alice 3 for their helpful comments and suggestions: Daniel Green (Oracle), Caron Newman (Oracle Academy), Susan Rodger (Duke University), Pam Lawhead (University of Mississippi), Leslie Spivey (Edison College), William McKenzie (Roger Williams University), Bill Taylor, Anita Wright, and Rose Mary Boiano (Camden County College), Tebring Daly (Collin College), Eileen Wrigley and Don Smith (Community College of Allegheny County).

COMMUNITY

We are proud to recommend the Alice Educator's listserv as a community for sharing questions and answers. The listserv is monitored and restricted to instructors. A link for subscribing to the Educator's listserv is available at: www.alice.org

As always, we welcome your comments and suggestions.

The Alice Team

CHAPTER 1

EXPLORING ALICE 3

The goal of this chapter is to provide information and instructions for downloading, installing, and starting Alice 3. Alice (all versions) is free and available for download at www.alice.org.

SECTION 1

∏

How To Install and Start Alice 3

The goal of this section is to provide an overview of system requirements and installation of the Alice 3 sstem, through brief descriptions and pointing out available resources.

MINIMUM SYSTEM REQUIREMENTS

- Desktop or laptop computer. Alice runs okay on some netbooks. However, many netbook models are not powerful enough to support 3D graphics animation. We suggest a trial run of a sample Alice 3 program on any netbook being considered for purchase.
- Windows XP, Vista, Windows 7, Mac OSX (Leopard, Snow Leopard, Lion, or Mountain Lion), or Linux
- 1 GB RAM (2 GB or more is recommended)
- VGA graphics card capable of high (32 bit) color and at least 1024x768 resolution (3D video card gives faster performance)
- Two- or three-button mouse is recommended. The touchpad on a laptop may be used. Please note, however, that arranging 3D objects in a virtual world is easier to control with a mouse than with a touchpad.

JAVA JDK:

The Alice installer makes use of the Java JDK (Java SE Development Kit). If the Alice installer indicates the Java JDK has not been installed, then see the instructions at http://help.alice.org for downloading and installing the JDK prior to downloading and installing Alice. If working on a networked system, ask the system administrator to install the JDK.

DOWNLOADING AND INSTALLING ALICE 3

The www.alice.org homepage includes a Downloads menu. Click on Downloads on the menu bar, as shown in Figure 1. Select "Get Alice 3.x". The Alice 3.x webpage should be displayed, as shown in Figure 2.

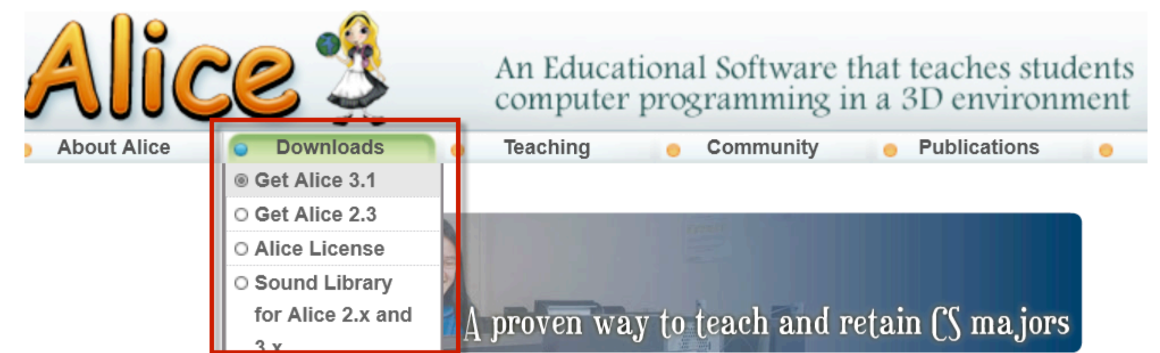


Figure 1 Downloads menu on www.alice.org

Figure 2 The Alice 3.x Download page

UNIVERSAL INSTALLER

The Universal installer works on **multiple platforms and/or on networked machines**. To download the Universal zip, click on the Universal zip Installer link. The Universal zip works on Windows, Mac,

and Linux platforms. The Universal zip automatically activates a download of the entire Alice 3.x system. On dial-up connections, this process typically takes 1 ½ - 2 hours, depending on the speed of the connection. After the download has completed, install by unzipping the downloaded file using a compression software application such as WinZip or 7-zip. The Universal zip file should extract to a folder named Alice 3. NOTE: For Windows 7 or 8, unzip to the desktop and then drag to the Program Files folder on the C:\ drive. This will avoid pop-up messages regarding administrator permissions.

STARTING ALICE 3

Because the downloaded file has just been unzipped, no shortcut icon has been created. Open the unzipped Alice 3 folder to view a list of folders and files, as shown in Figure 4. Four start files are highlighted in the red box in Figure 4.

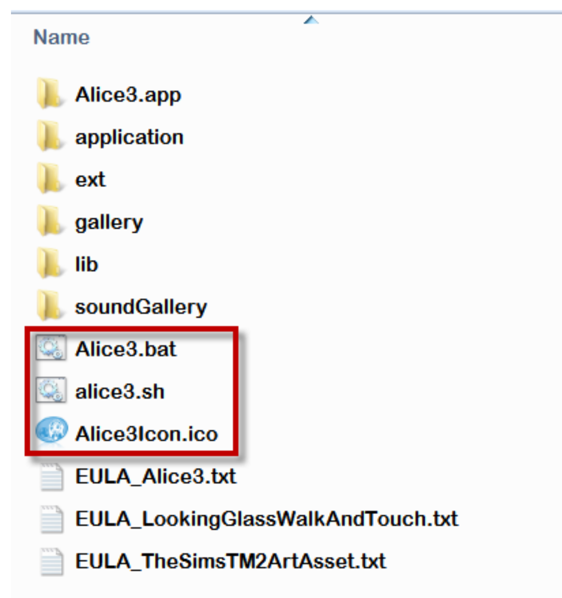


Figure 4 Start files in the Alice 3.x Universal version

These start files are designed to start Alice 3.x on a specific operating system (OS), as is appropriate. To start Alice 3.x, click on the appropriate start file for the OS installed on the computer system, as designated here:

Alice3.bat	Starts Alice 3 for a PC system (either 32 or 64 bit)
alice3.sh	Starts Alice 3 for a Linux system, 32 bit
alice364bit.sh	Starts Alice 3 for a Linux system, 64 bit
Alice3Icon.ico	Drag this icon to a Mac OSX dock. Then click the icon to start Alice 3 on a Mac system (Leopard, Snow Leopard, Lion, or Mountain Lion).

SELECT PROJECT DIALOG BOX

When Alice starts, a Select Project dialog box is automatically displayed, as shown in Figure 5. The Select Project dialog box has four tabs and the Templates tab is automatically selected. Choose any one of the templates or click on one of the other tabs to select a previously written Alice project.

The red X message in the bottom left of the window is a warning message that indicates a template or a previous project must be selected in this window in order for Alice to display the Code Editor and a current scene. If the Cancel button is clicked without selecting a template or a previously written project, Alice will close the Select Project dialog box but the Code editor will not be opened. To reopen the Select Project dialog box, click File in the menu bar at the top left of the Alice window and select New from the menu.

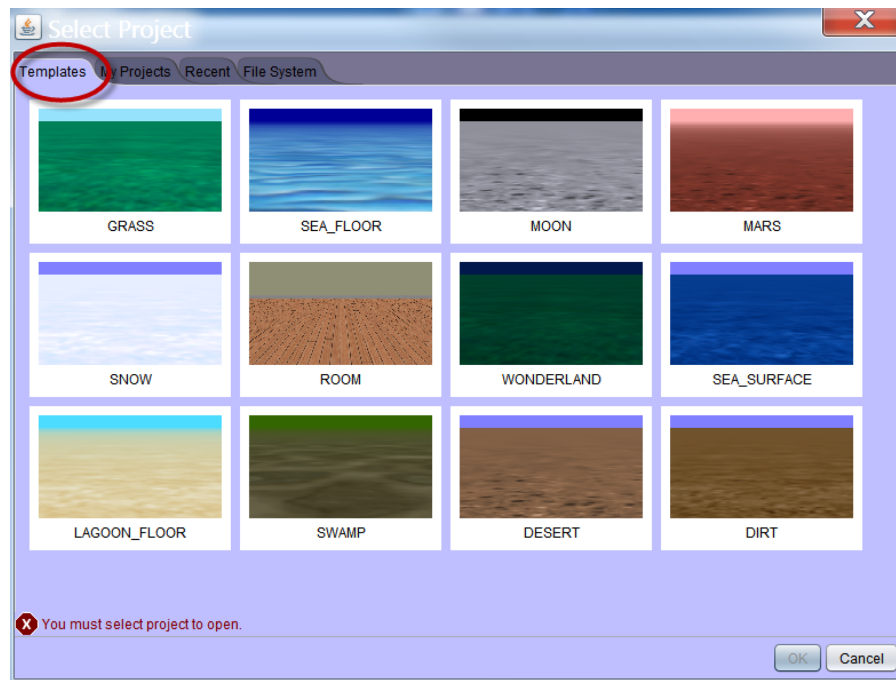


Figure 5 Select Project dialog box

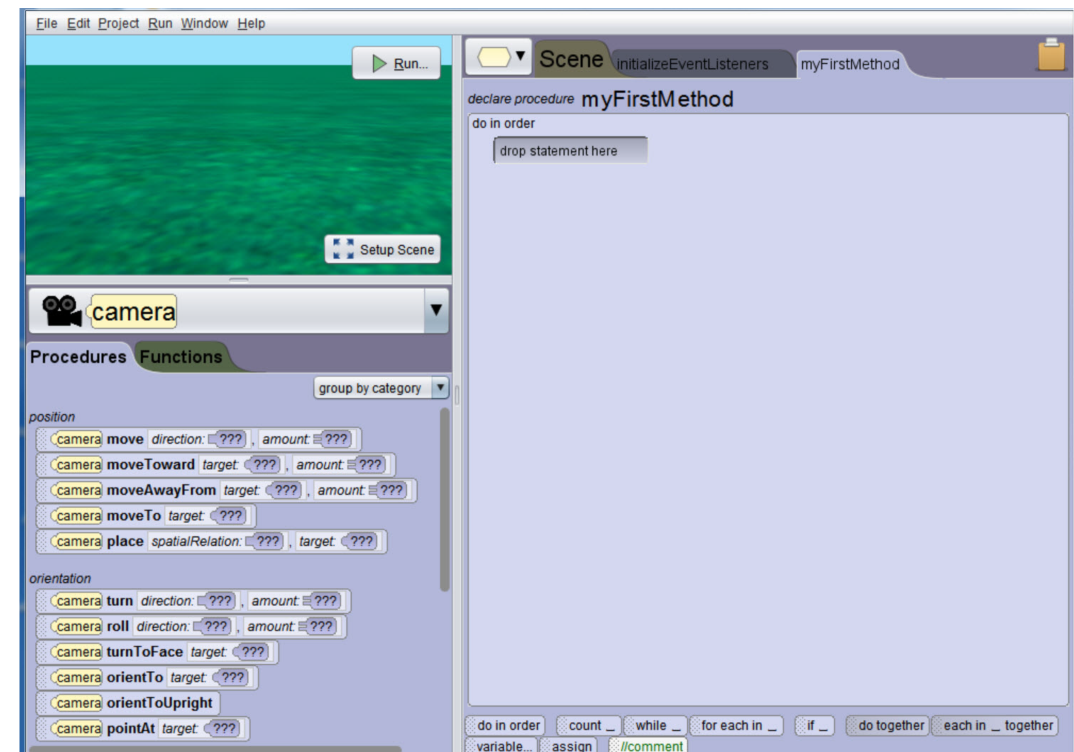


Figure 6 Start Screen: Selected template in the Code editor

Upon successful selection of a template, Alice will display the selected template scene in the upper left corner of the Code Editor, as shown in Figure 6. (The display may vary somewhat, but the basic organization should be the same as shown here.)

TROUBLESHOOTING: PC DISPLAY DRIVER UPDATES

If Alice does not start or if the templates are not properly displayed, the display driver may need to be updated. For Windows PC users, we advise updating the display driver for the computer system directly from the video display card's manufacturer website (rather than Windows Update). Instructions for updating the display driver for a computer system may be found at http://help.alice.org, in the section labeled "Updating video drivers for Windows machines."

SECTION 2

∏

A brief tour of the Alice 3 IDE

The goal of this chapter is to provide an overview of the components in the Alice IDE (Interactive Development Environment). The components are briefly described and screen shots identify the individual components.

[Video: A Brief Tour of the Alice IDE]

Later chapters will provide greater details and demonstration examples. The IDE components include:

Select Project dialog box: Select a scene template or existing project
Code editor: Camera view, Editor tabs, Control tiles, Methods panel
Scene editor: Camera view, Handles palette, Setup Panel, Gallery

SELECT PROJECT DIALOG BOX

The Select Project dialog box has four tabs, which allow selection of a scene template or an existing project. Figure 1 shows the templates each of which contains a surface (for example, grass, moon dust, snow, dirt, or water) and an atmosphere (for example, blue sky, greenish fog, or black outer-space). You can either single-click on the template image and then click the OK button or you can just double-click on the template image.

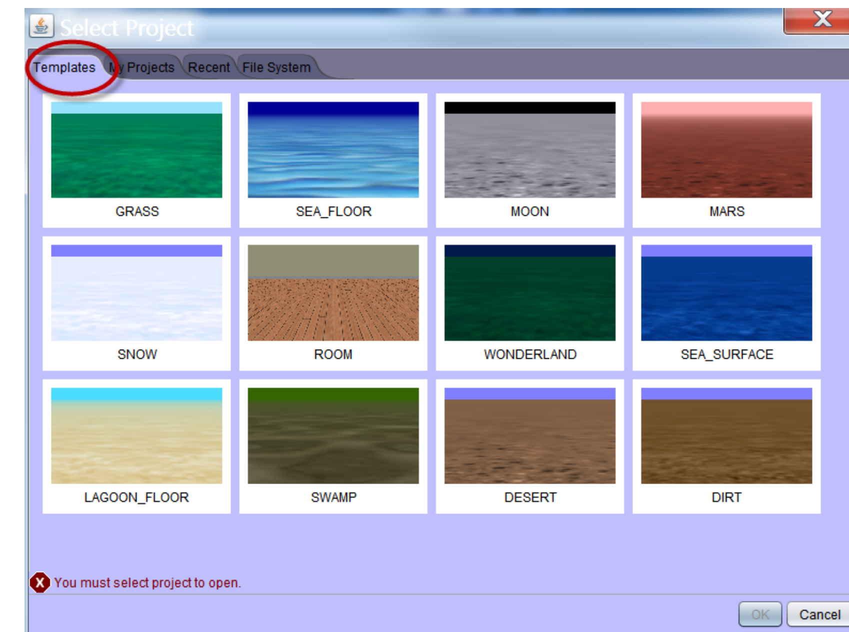


Figure 1 Select a template

Other tabs (**My Projects, Recent**, and **File System, as seen in Figure 2**) in the Select Project dialog box are for the purpose of opening an existing project. **My Projects** provides a list of existing projects stored in Alice's Projects folder, **Recent** provides a list of recently opened projects, and **File System** provides a directory browser for finding a file in other locations on your computer or a storage device (e.g., thumb drive, CD, or DVD). The **File System** browser is shown in Figure 2.

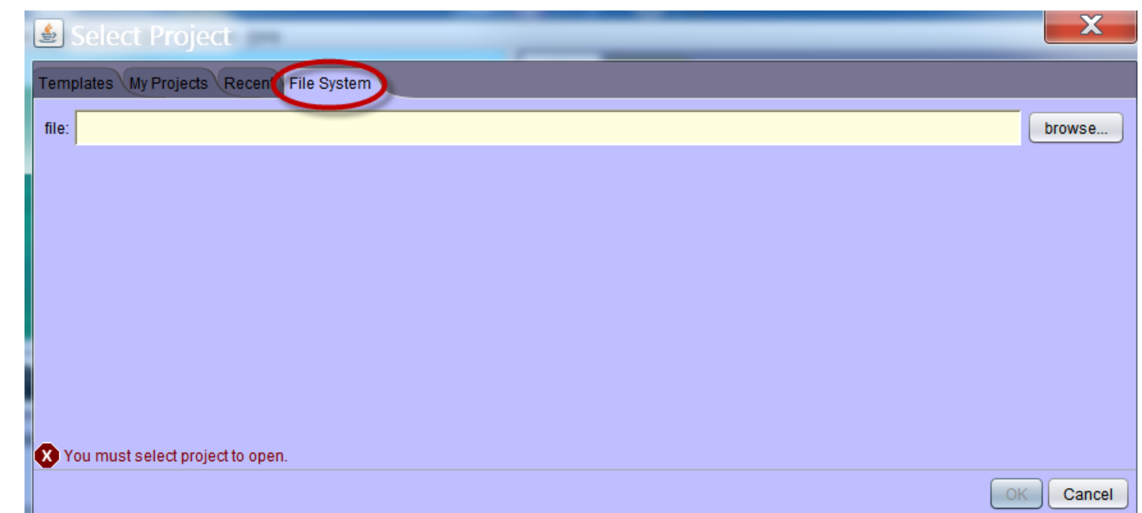


Figure 2 Select an existing project from the file system

INITIAL DISPLAY WINDOW AND MENU BAR

Upon selection of a template or starter for the scene, Alice will display the scene in the upper left corner of the window, as shown in Figure 3. In Alice, the interface is a programming environment where a virtual world (a scene with actors and props) and a program (a script that gives instructions to the actors) can be created to enable interaction and communication between Alice and a programmer (user).

CODE EDITOR

In addition to displaying the Camera View of the scene (upper left), the opening interface displays Edit panel (right) with tabbed panes where different parts of a program are created. The Code editor also has a Methods panel (lower left) and a Controls panel (lower right), as labeled in Figure 3. When Alice is first started with a new template, the camera is the selected object, the Camera view displays the scene you would see if you were looking through the camera, and *myFirstMethod* (the main method defined for a scene) is the default open tab in the Edit panel.

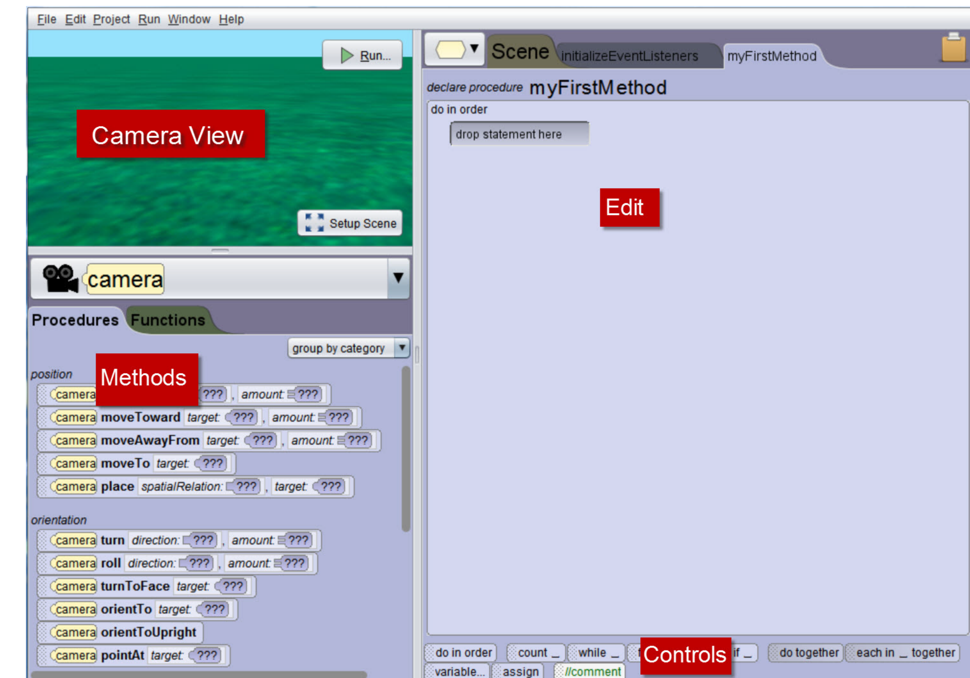


Figure 3 Code editor panels

METHODS: PROCEDURES AND FUNCTIONS

In the Methods panel, each tile represents a method. A method is an action performed on or by an object (animal, person, prop, fish, or some other entity). As shown in Figure 4, the Methods panel categorizes methods for display on two tabs: Procedures (methods that perform an action), and Functions (methods that ask a question or compute a value). In the screen capture of this example, the camera object's Procedures tab displays method tiles such as *move*, *moveToward*, …, *turn*, *roll*, and others.

Figure 4 Methods: Procedures and Functions

CONTROL PANEL

In the Control panel, each tile represents a statement for managing instructions and data in program code. Figure 5 highlights the control tiles.

Most control tiles manage the order in which instructions (method statements) are performed. As an example, the **do in order** tile is used to specify which instructions should be performed in the order in which they are listed. However, the **do together** tile is used to specify which instructions should be performed simultaneously. The *//comment* tile is used to create a statement that is NOT performed.

Some tiles in the Control panel are for managing information (data). As an example, *variable* is used to set aside some memory space for holding data. The memory space is labeled with a name (that is, a variable has a name). The *assign* tile is used to create an instruction that stores data in a variable's memory space.

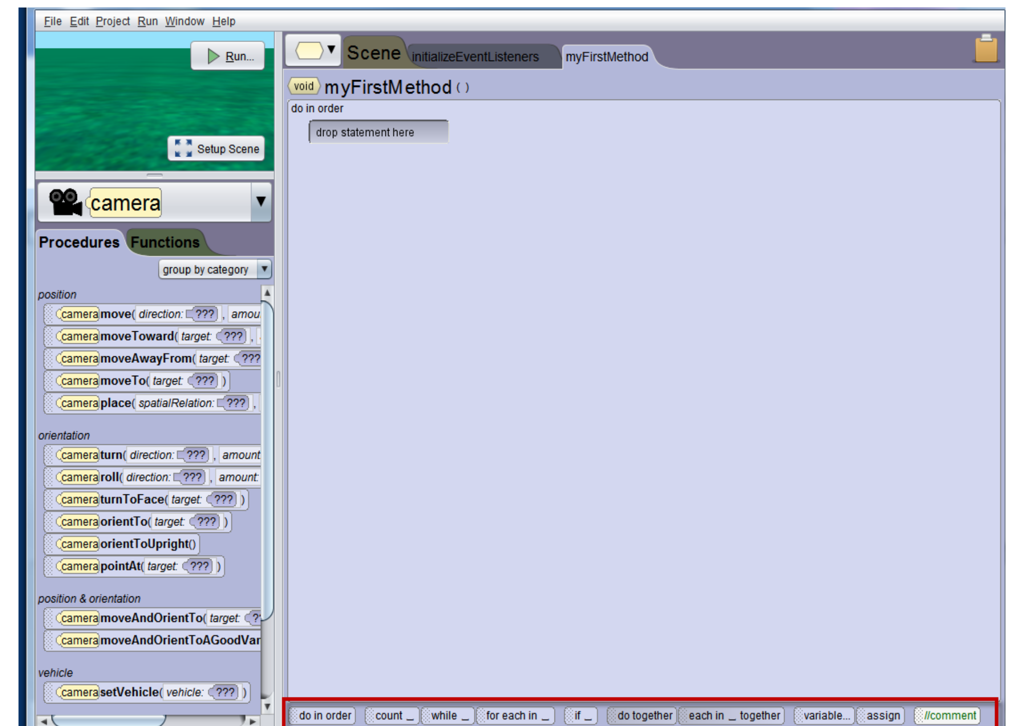


Figure 5 Control tiles

In summary, the Code editor provides a drag-and-drop environment where method and control tiles are dragged into the edit space to create **instructions** (method and control statements) that compose a program. In Alice, a program animates objects in a scene.

SCENE EDITOR

To view the Scene editor, click the Setup Scene button in the lower right corner of the scene, as shown in Figure 6.

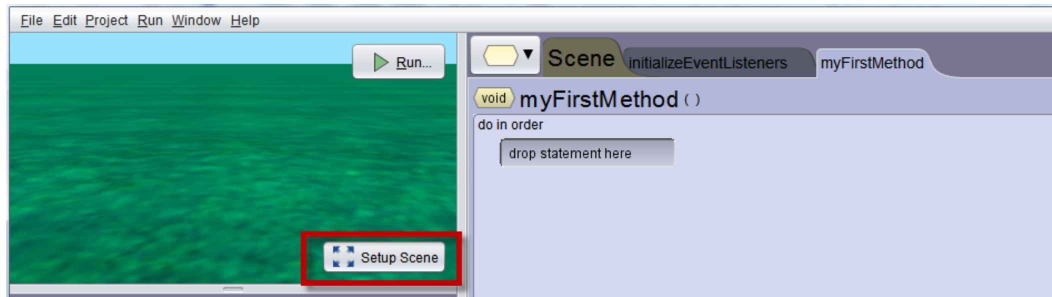


Figure 6 Setup Scene button toggles to the Scene editor

As labeled in Figure 7, the Scene editor has two panels: Scene Setup and Gallery. The purpose of this editor is to create a virtual world by adding and arranging the objects in a scene. The Gallery contains 3D models that are used to create objects in the scene. The SetUp panel provides mouse control handles for positioning objects in the scene and menus for changing size, color, vehicle, position, and other properties of objects in the scene.

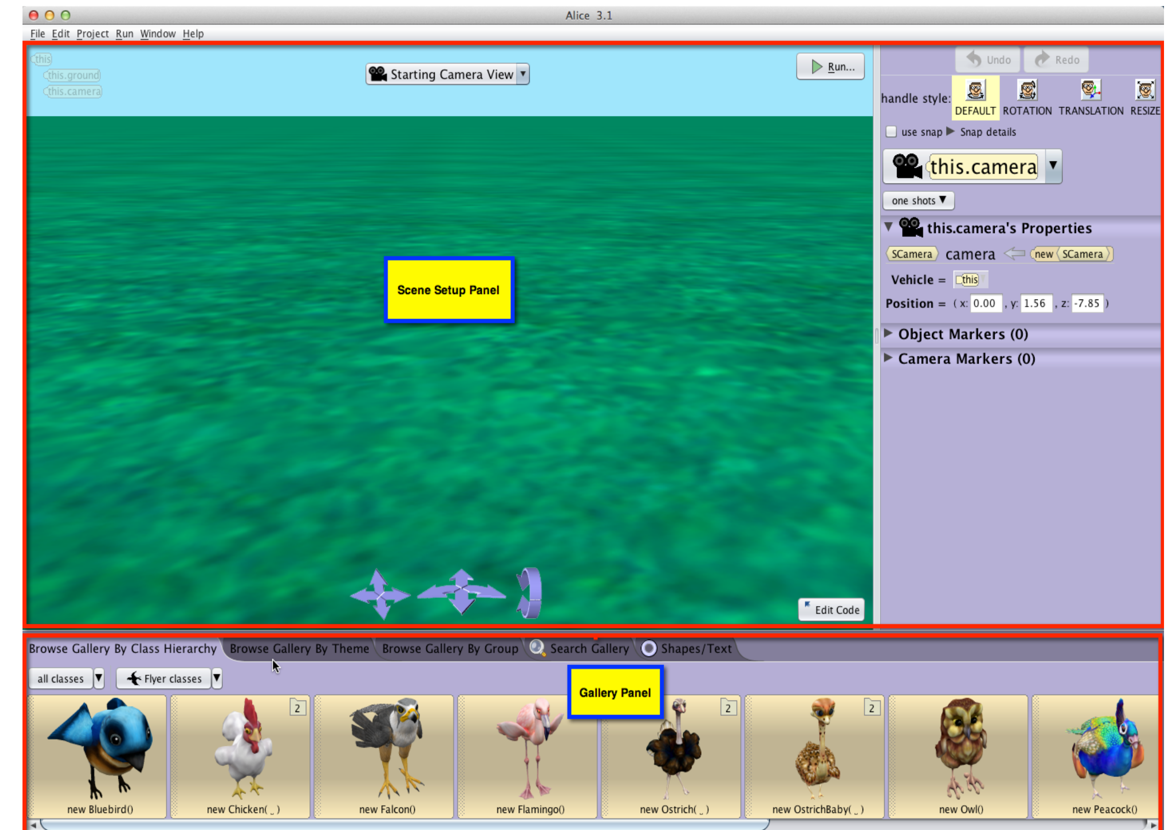


Figure 7 Alice 3: Scene editor panels

TOGGLE BETWEEN TWO EDITORS

The creation of an animation often involves frequent switching back and forth between the Code and Scene editors. To toggle between the two editors, click the Setup Scene button in the Code Editor or click the Edit Code button in the Scene editor, as shown in Figure 8.

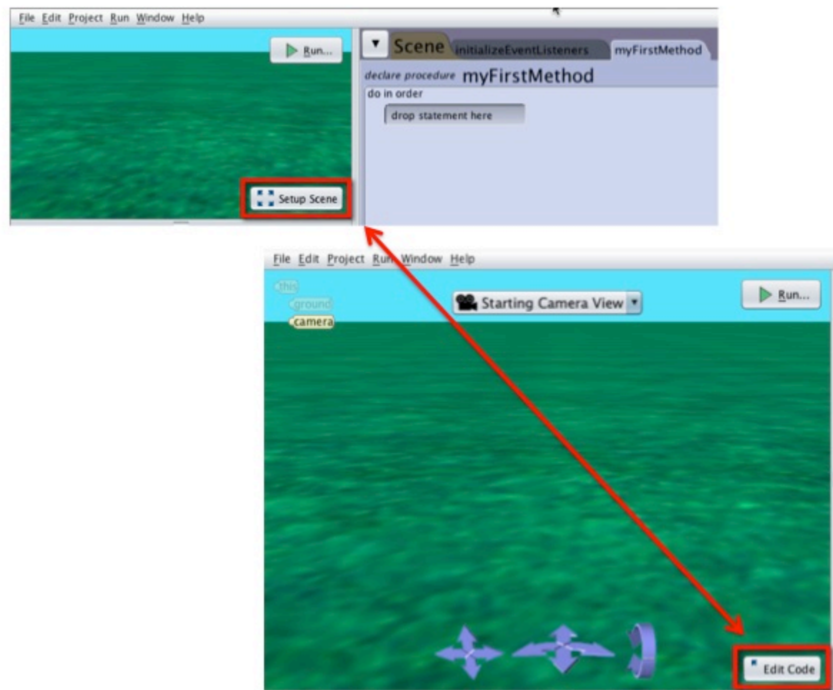


Figure 8 Toggling between Code and Scene editors

SECTION 3

∏

A brief tour of the Menu Bar

The purpose of this section is to introduce the menus of the Alice 3 menu bar.

Many menu items are typical of commonly used software applications and their operations are well-known. We will assume the reader is familiar with these items and no illustration will be provided. Some items are specific to the operation of Alice, in which case we provide a brief description and an illustration of the items.

In Alice 3, a menu bar is displayed in the upper left corner of the window, as shown in Figure 1. The menus include: File, Edit, Project, Run, Window, and Help.

Figure 1 Menu bar

FILE MENU

The File menu contains options for managing and editing files in a project, as shown in Figure 2. The items in the File menu are: **New, Open, Recent Projects, Save, Save As, Revert, Upload to YouTube, Print,** and **Screen Capture.** On Windows machines, you may also see **Exit** (at the bottom of the menu).

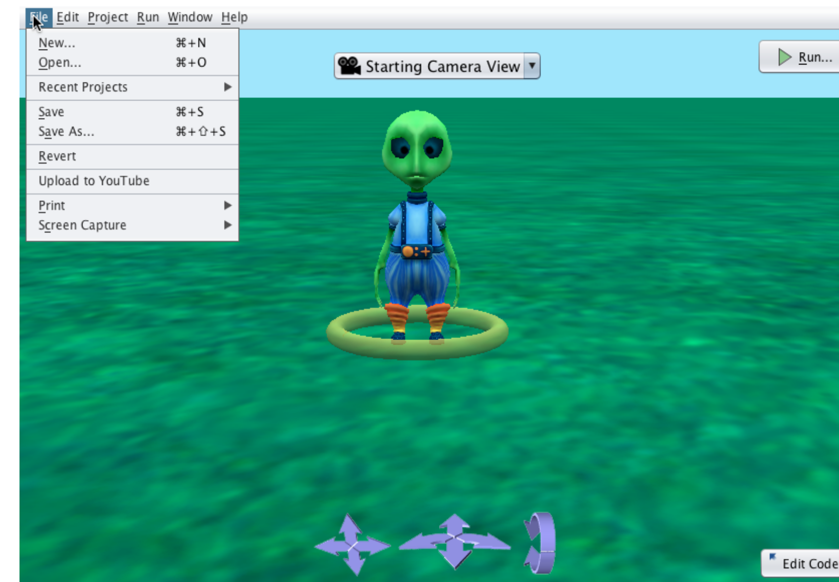


Figure 2 Menu for managing files

New, Open, Save, Save As, and **Exit** are typical of many software applications and their usage is well-known. Our only suggestion is that the first time an Alice project is saved, use Save As instead of Save. Save As guarantees that the file will be saved in a user-selected directory rather than a default directory. Files can be saved on the C drive (hard drive), a networked drive, a USB drive, or a read-write CD/DVD.

- **Recent Projects** provides links to recently saved Alice 3 projects.
- **Revert** restores a scene to its initial state when the world was first opened in the current session. In other words, all actions in the current editing session are backtracked and removed.
- **Upload to YouTube** (at the time of this writing, this feature is not yet fully implemented) will allow you to capture a video of the currently running animation and export (upload) to a YouTube account. Alternately, the video can be saved to your computer for later playback.
- The **Screen Capture** menu item is available for capturing images in either Code or Scene editor mode. As shown in Figure 3, the **Screen Capture** menu item cascades to three choices for selecting an area of the screen to be captured. The ***Capture Entire Window*** option will copy a screen shot of the entire Alice 3 interface to your system clipboard, which can then be pasted into another document. ***Capture Rectangle*** cross-hatches the entire Alice IDE. You then click and drag over the portion of the Alice interface that you wish to copy to the clipboard. ***Screen Capture...*** brings up a dialog box that allows you to choose to capture the Entire Window, the Content Pane, or a Rectangle region. It will also allow you to set the resolution dpi for the image, as illustrated in Figure 4.

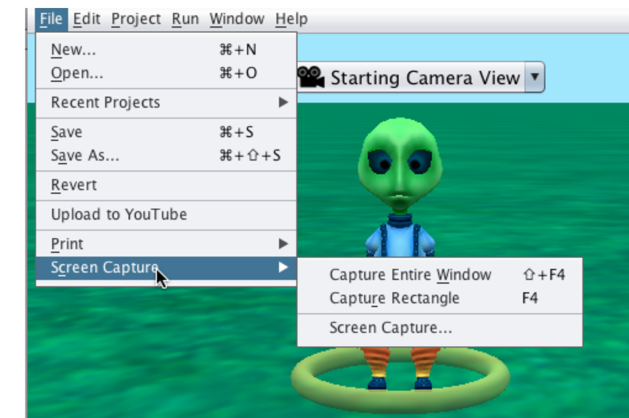


Figure 3 Screen Capture options in the File menu

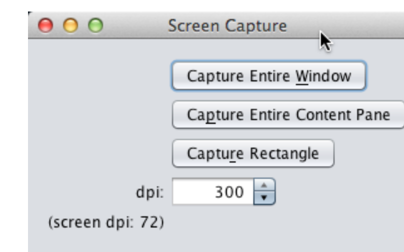


Figure 4 Screen Capture... submenu dialog box

NOTE: The built-in Screen Capture in Alice 3 does NOT capture images from the runtime window while an animation is playing. However, this can be accomplished on a Windows PC system by using the Alt/PrintScreen keyboard shortcut. The captured image is automatically saved to the system clipboard. Just paste it into Paint or a Word doc, using CTL/V. On a Mac OS X system, the keyboard shortcut for capturing the runtime window is Command-Shift-4. A cross-hair cursor will appear and you can click and drag to select the area you wish to capture. When you release the mouse button, the screen shot will be automatically saved as a PNG file on your desktop.

- The **Print** menu item is available for printing program code. As shown in Figure 5, the **Print** menu item cascades to three choices for selecting how much of the program code to print. The ***Print All*** option will print all the code created in the program. (Of course, this does not include pre-written code which is part of the Alice system.) ***Print Current Code*** will print only the code in the currently active method tab in the code editor. P***rint Scene Editor*** will print a screen capture of the Scene Editor, including a screen capture of the Camera View and the Setup panel but not including the Gallery panel.

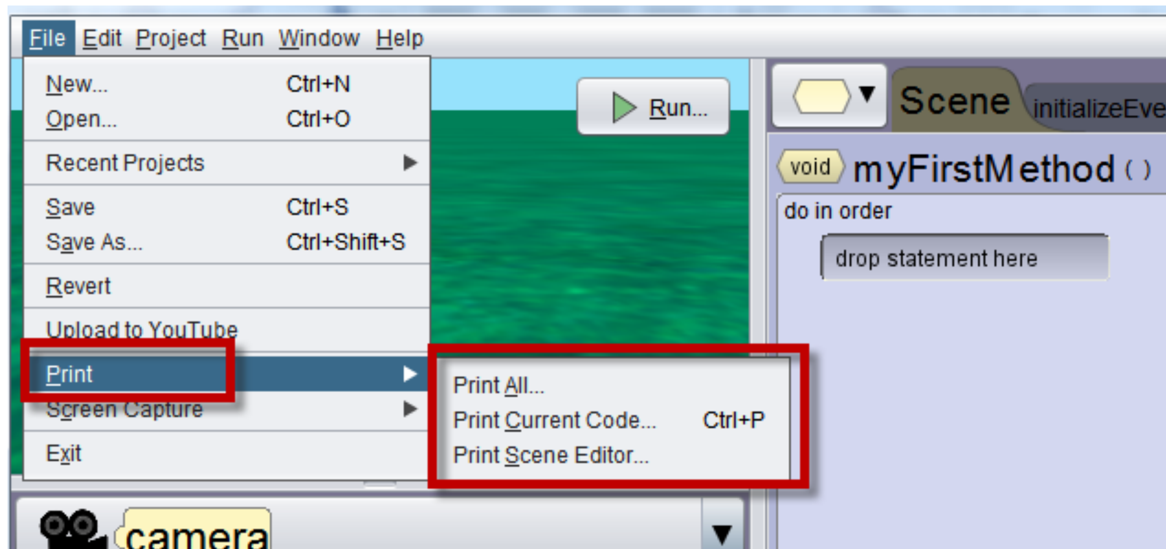


Figure 5 Print options in the File menu

EDIT MENU

The **Edit** menu contains ***Undo***, ***Redo***, ***Cut***, ***Copy***, and ***Paste***, as shown in Figure 6. These are all standard editing actions. As of this writing, ***Cut***, ***Copy***, and ***Paste*** are not implemented but are listed in the menu to allow for future modifications. Truthfully, although the traditional cut, copy, and paste actions work well in a text editor, these actions are of limited usefulness in a drag-and-drop editor. Section 16 of this guide provides detailed instructions on using the clipboard for ***Cut***, ***Copy***, and ***Paste*** in a drag-and-drop programming environment.

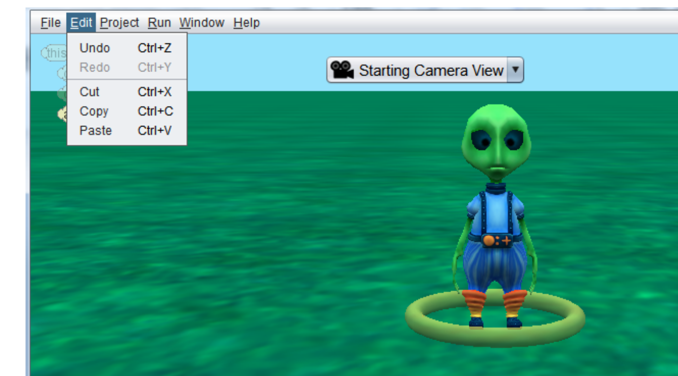


Figure 6 Menu for Edit options

PROJECT MENU

The project menu contains ***Resource Manager***, ***Find***, and ***Statistics***. The ***Resource Manager*** item opens a dialog box for importing (or removing) resource files as shown in Figure 7. A resource file may be either an audio or image file. Alice does not provide sound or image editing capabilities.

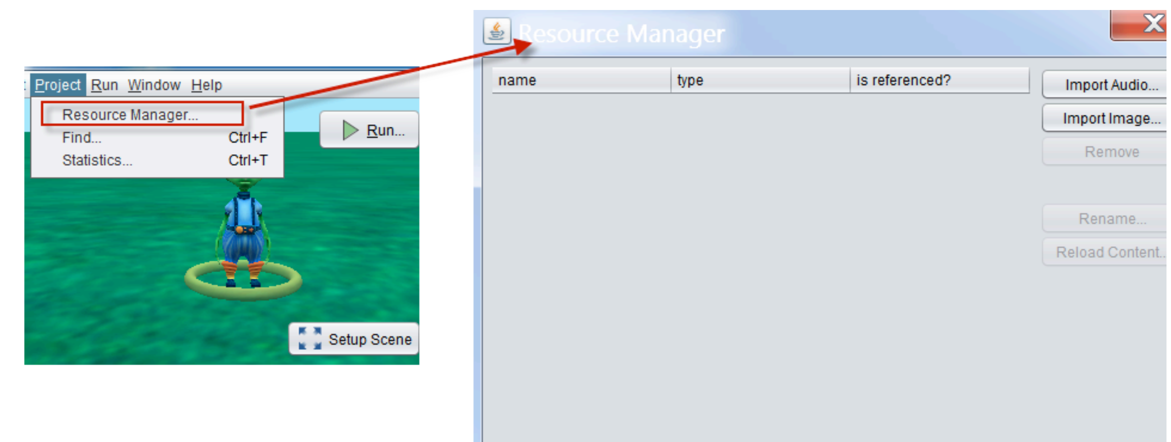


Figure 7 Project menu and Resources Manager

The ***Find*** item pops up a dialog box for searching the program code to find where a method is called. In Figure 8, the scene's method named ***setAtmosphereColor*** has been selected in the Find box. A message

appears in the column on the right to show that the method has been called (one time) in the program's *performGeneratedSetUp* method.

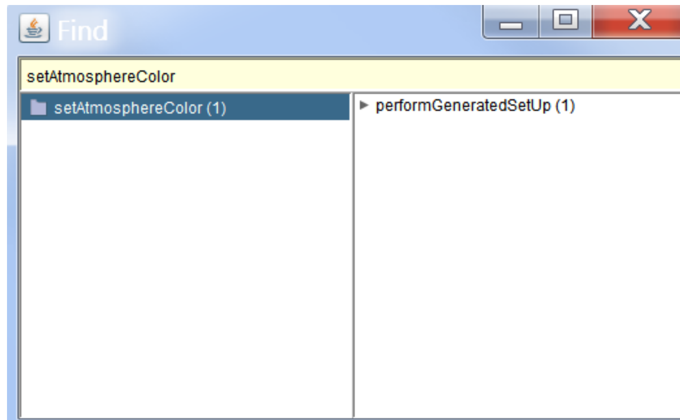


Figure 8 Find where a method is called in the program

The ***Statistics*** item pops up a window that displays a frequency analysis of constructs and method calls within the current project, as shown in Figure 9.

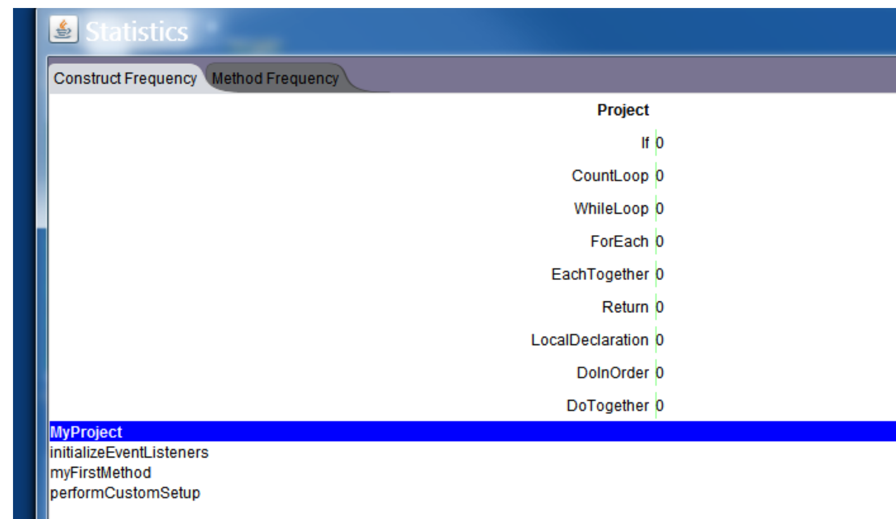


Figure 9 Statistics popup window

Window Menu

The Window menu contains ***Perspectives***, ***Project History***, ***Memory Usage***, and ***Preferences***, as shown in Figure 10. These items control the display of the Alice 3 environment in terms of the number of open windows and their content.

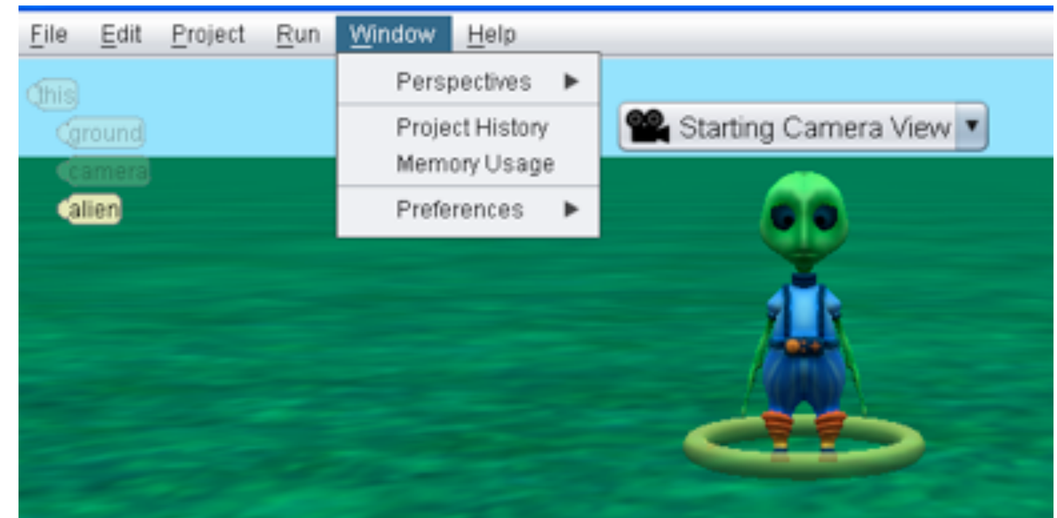


Figure 10 Window menu

The ***Perspectives*** menu provides an alternate means of toggling between the Code Editor or Scene Editor display, as shown in Figure 11.

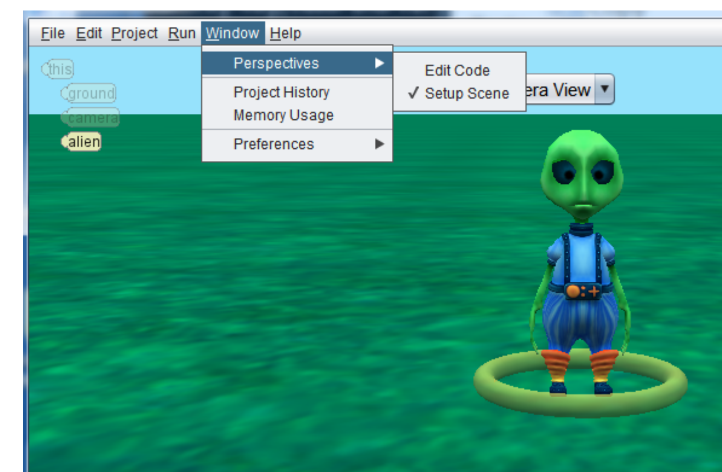


Figure 11 Perspectives

The ***Project History*** item opens a new window containing a list of all actions performed (thus far) in the current editing session. Figure 12 shows a ***Project History*** window in which the actions listed include the declaration of an alien object (was added to the scene) and then the object was moved. The actions in the history are listed in the order they were performed. The history does not extend over the life of the project, only having a record of actions in the current editing session.

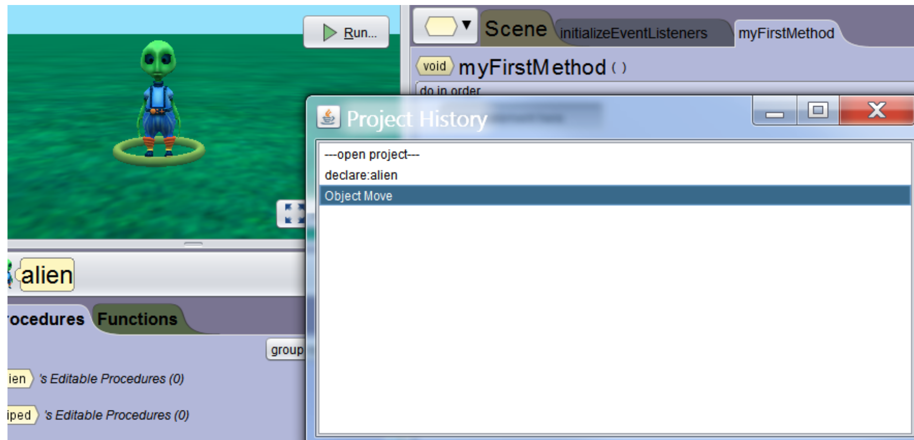


Figure 12 View the history of actions in this session

Backtrack In History

It is possible to backtrack to a previous state (objects in the scene, their locations, and their properties) of the world by clicking an earlier item in the Project History. Selecting an item (other than the last one) causes all later actions in the list to be "played backwards." To illustrate, we clicked **declare alien** in the Project History and the Object Move action was played backwards, moving the alien back to its initial location when it was added to the world. The state of the world is now displayed, as shown in Figure 13.

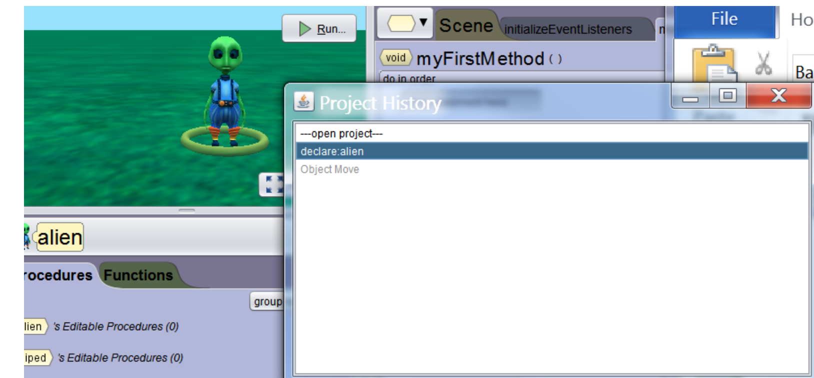


Figure 13 Backtracking to a previous state

Selecting the Memory Usage item in the Window menu opens a popup window in which memory usage is tracked, as shown in Figure 14. An alert is displayed in the window's task bar when Java's garbage collection is in progress.

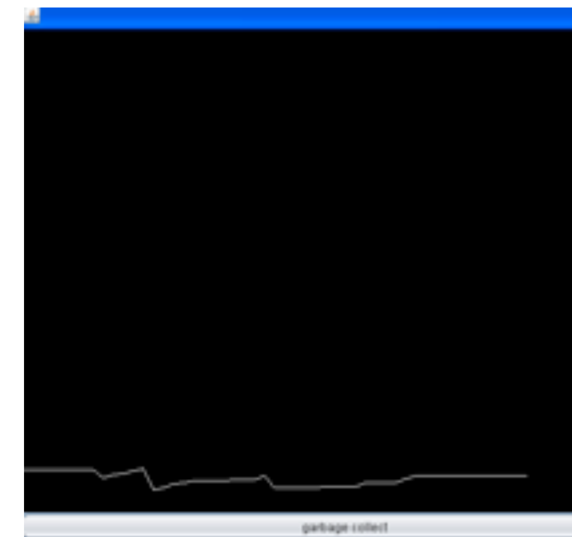


Figure 14 Memory usage window

Selecting the Preferences menu item opens a cascading menu for setting preferences in the Alice 3 environment, as shown in Figure 15.

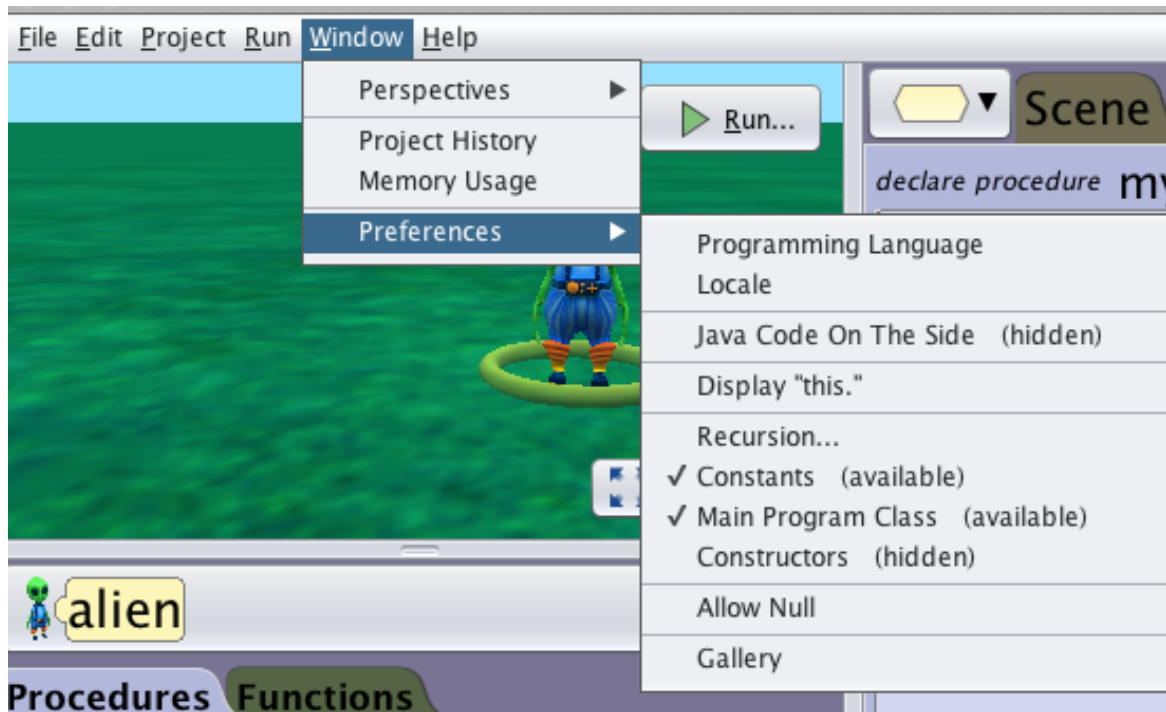


Figure 15 Preferences cascading menu

A quick overview of the Preferences menu items is provided here. Details for setting preferences are provided in the next section of this How-To guide.

- **Programming Language:** Display code using Alice or Java syntax
- **Locale**: Display code in the natural language selected (English, Spanish, Chinese, Portuguese, Russian, and others).
- **Java Code On The Side**: Display Alice code side-by-side with its Java syntax form. The Alice code can be edited and the Java code will automatically update. However, the Java code cannot be edited directly in this view.
- **Display "this.":** Display "this." in program code to represent any object of the currently selected class. An option is available to disable "this." when writing code in the Scene class.
- **Recursion**: Enable the use of recursion in Alice programs.
- **Constants**: Allows the user to declare constant fields in the program.
- **Main Program Class**: Includes the Program class in the Class menu list. Program contains the main method, which is the first method executed when the Run button is clicked.
- **Constructors**: Add a constructor option to the Class tab for each class used in a scene.
- **Allow Null**
- Allow Null for field initializers: Allow a class variable to be declared without an initial value
- Allow Null for local initializers: Allow a local variable to be declared without an initial value.
- **Gallery**: Preference settings for the dialog box are displayed when adding an object to a scene. Gallery preference settings include enabling or disabling a preview of the declaration for creating an object and an option for auto-naming. Figure 16 shows the dialog box with (left) and without (right) a preview of the declaration statement.

Figure 16 Preview on (left) and off (right) in the preference settings

Help Menu

The Help menu contains: **Help…, Help with Graphics Problems…, Report a Bug…, Suggest improvement…, Request a New Feature…,**

Show Warning…, **Show System Properties…**, and **Browse Release Notes[web]**, as shown in Figure 17.

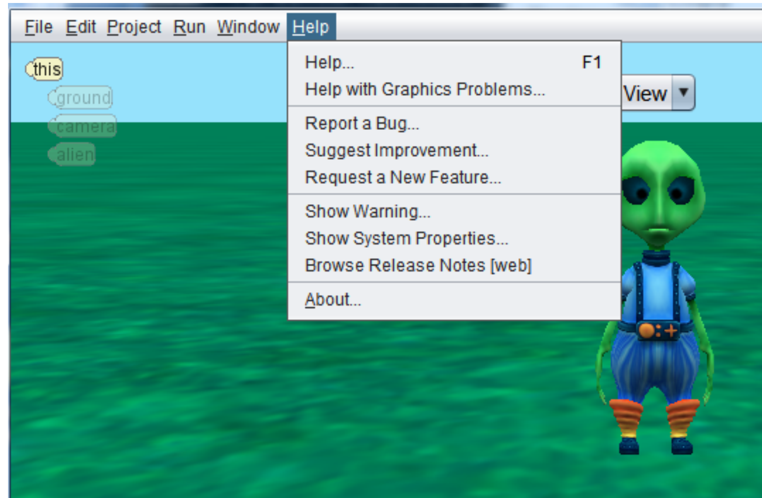


Figure 17 Help menu

The **Help…** item opens a window containing a link to the help page at http://help.alice.org, as shown in Figure 18.

Figure 18 Help link

The **Report a Bug…**, **Suggest Improvement…**, and **Request a new Feature**… items each open a window containing a form for the specified action. This feature allows Alice users to submit a bug report, suggest improvements, and provide ideas for new features. A copy of the bug report form is shown in Figure 19.

For any of these forms, submitting the form requires the computer be actively connected to the internet. If not connected to the internet, the report will simply be deleted when Alice is closed on your computer.

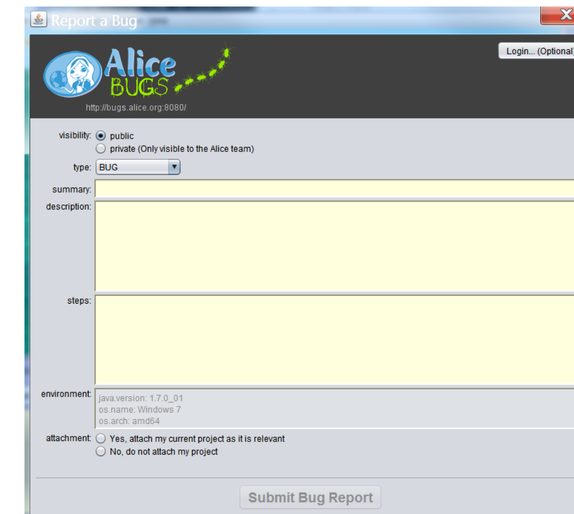


Figure 19 Bug report form

SECTION 4

∏

How to Set Preferences

Setting preferences changes the "look and feel" of the Alice 3 IDE. The purpose of this section is to demonstrate how to set a preference. Any combination of preference settings is possible, as selected by the user.

DEFAULT PREFERENCES

The Alice installer has a pre-defined set of preferences for the "look and feel" of the Alice environment. The default settings are shown in Figure 1, where only one item (Constants) is selected in the menu. Enabling the Constants preference turns on the ability to create a named value that cannot be modified at runtime.

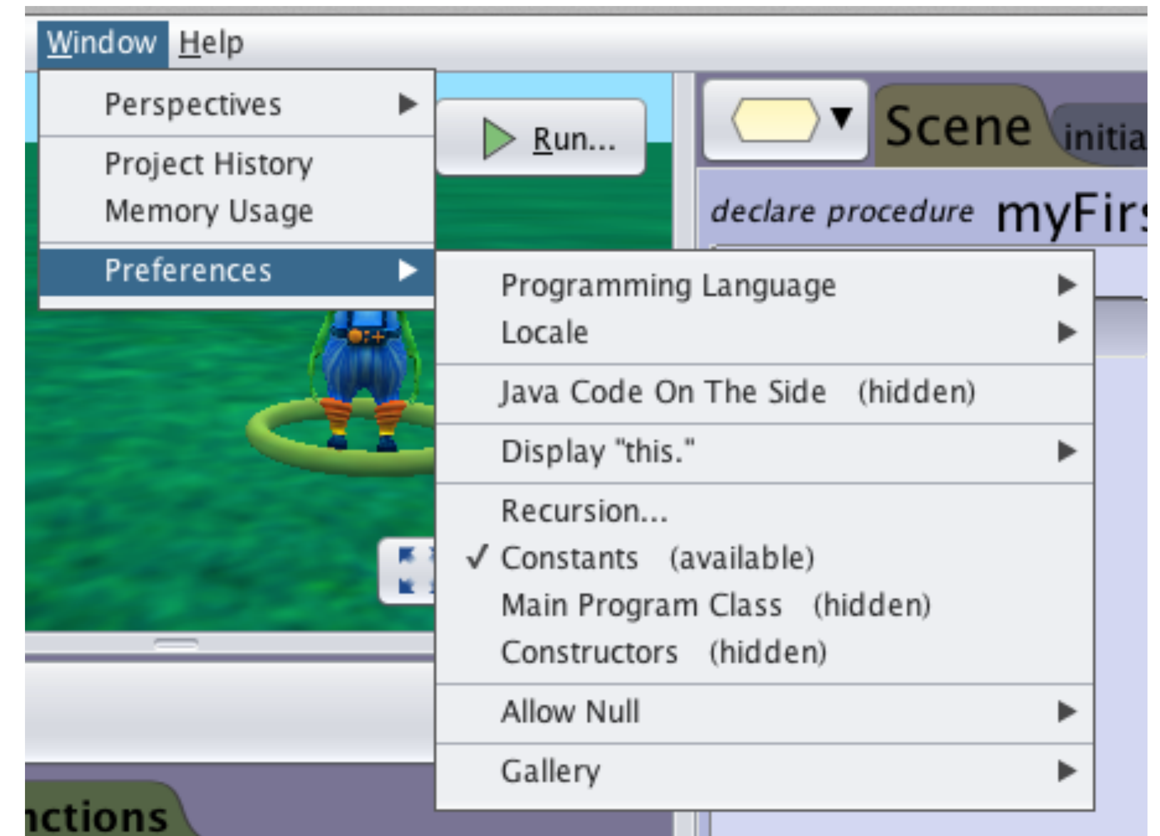


Figure 1 Default preference settings

With the default preference settings, Alice starts with Scene as the currently active class, as shown in Figure 2. The active editor tab is *myFirstMethod*, a method belonging to the Scene class. After code has been created and the user clicks on the Run button, the scene will be displayed in a popup window (runtime window) and then the code in *myFirstMethod* will be executed (run).

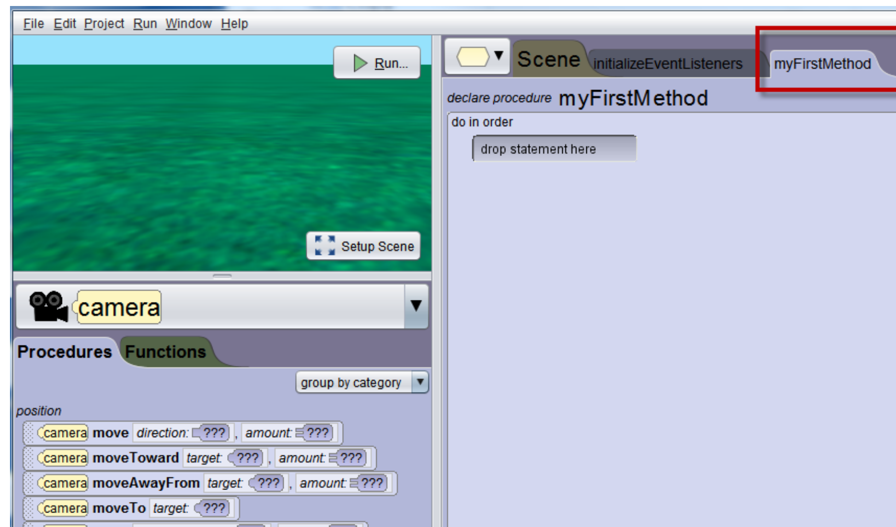


Figure 2 myFirstMethod tab

Another default preference setting is for the keyword, *'this'*. In both Alice and Java, the keyword *'this'* refers to the current object of this class. As an example, in the Scene class *'this'* is the current scene, in the Alien class *'this'* is the current alien, and in the Penguin class *'this'* is the current penguin.

In an Alice world, a scene often contains objects of other classes, as shown in Figure 3. This scene contains ground, camera, alien, and penguin objects. In the pull-down menu, 'this' is the scene and the objects belonging to the scene are labeled *this.ground, this.camera, this.alien,* and *this.penguin.*

Figure 3 'this' is the scene

Displaying the keyword *'this'* is enabled by default, as shown in Figure 4. If you find this practice to be confusing or distracting, you may elect to turn it off in the preferences menu, as shown in Figure 5. (**Note:** You may need to save the world, close, and reopen to refresh the menu display.)

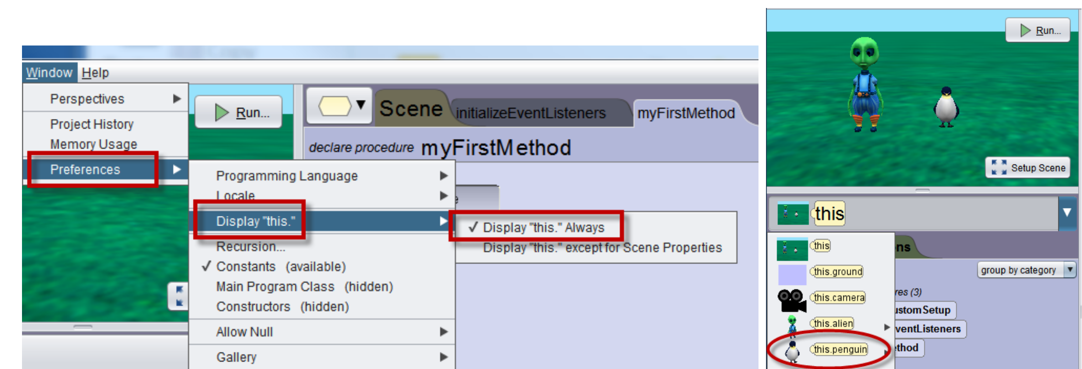


Figure 4 'this' is displayed for a scene and each object within it

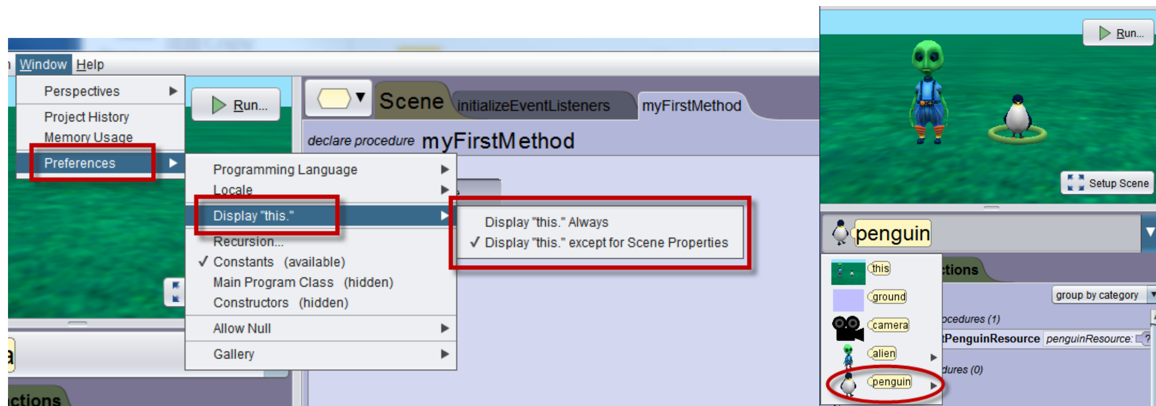


Figure 5 'this' is displayed for the scene object but not for objects within

SETTING MULTIPLE PREFERENCES

Any combination of preferences may be set. Figure 6 shows two recommended preferences (Constants and Constructors) for those who wish to focus on object-oriented programming concepts with an intention to prepare for learning a production level language, such as Java.

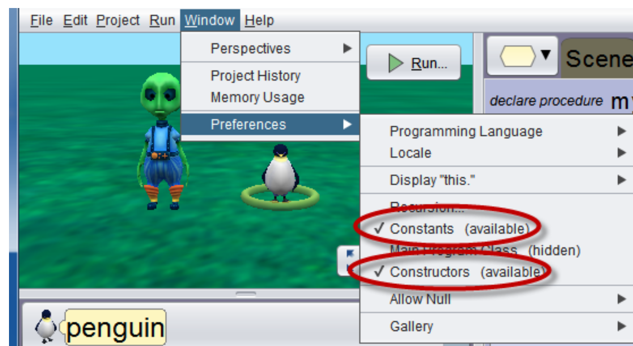
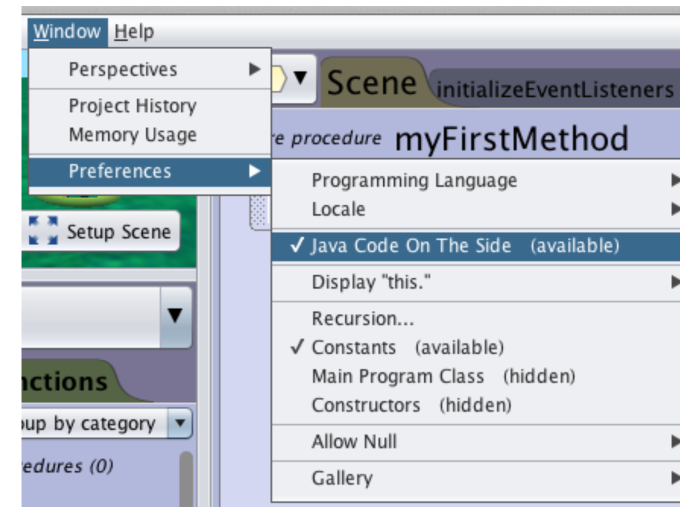


Figure 6 A selection of two preferences

SETTING A PREFERENCE FOR ALICE AND JAVA SIDE-BY-SIDE

For those using Alice as preparation for learning Java, setting the Java Code on the side preference enables a dual display of Alice and Java code in side-by-side panels, as shown in Figure 7.



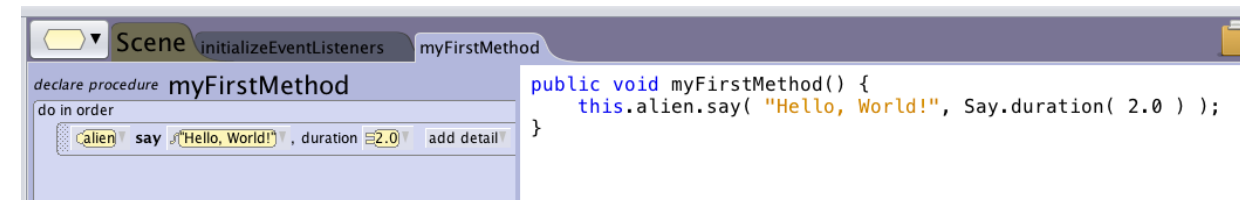


Figure 7 Alice and Java code, side-by-side display preference

Section 5

∏

Classes and the Gallery of 3D Models

The purpose of this section is to explore the concept of classes and objects as well as illustrate the relationship of the 3D models (as provided in the Gallery) to classes and objects in an Alice 3 project.

Models

In our daily lives, we think of a model in many different ways. We think of a model as a product when we say, "This car is the latest model." We might think of a model as someone to be imitated when we say, "She is a model student." To an architect, a model is a blueprint (a design for construction). Figure 1 illustrates a blueprint for house. This blueprint is a model that provides a design. The blueprint model tells a home-builder how to build the house but is not an actual physical instance of a house.

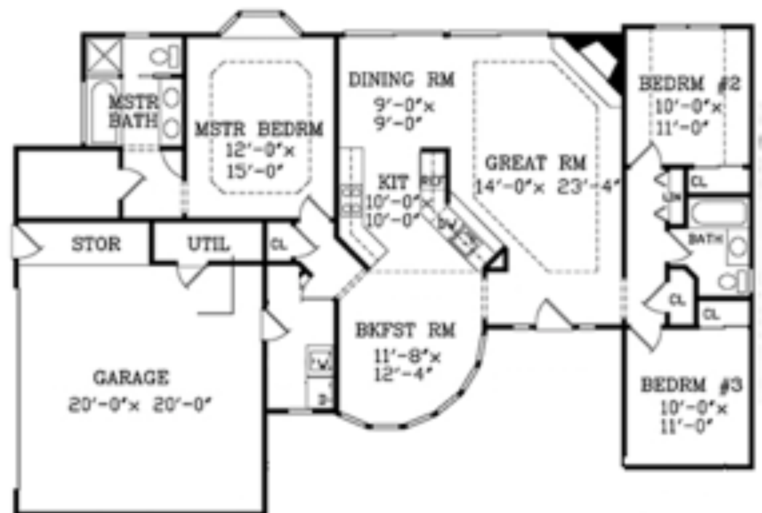


Figure 1 A blueprint for constructing a house

3D Models and Classes

In animation film studios such as Disney, Pixar, and DreamWorks, a 3D model is a digital representation of an entity (someone or something) in three dimensions (height, width, and depth). Animation adds motion to a model. A 3D model contains instructions for building the digital object.

In Alice, a class puts together a digital representation of an entity, a plan for constructing it, and instructions for animating, all in the same jar fire. A more general definition is: A class defines a type of object (a modeled entity) and actions that can be performed by that object.

Gallery

The Gallery (in the Scene Editor) contains classes for creating and animating objects in an Alice virtual world. Figure 2 shows a collection of classes in the Gallery's Biped collection. Each class is a 3D model for building an object of a specific type (for example, an alien, a cat, or a curupira).

Figure 2 Classes (3D Models) in the Alice 3 Biped

Using the Gallery

Figure 3 shows a newly created Alice world. An Object tree is displayed in the upper left corner of the scene. The Object tree contains a list of all the objects in this scene. A new scene automatically has a ground (or water) surface and a camera. The scene is an object of the Scene class, the ground an object of the Ground class, and the camera an object of the Camera class.

Figure 3 A new Alice world with an object tree (upper left of Scene editor)

A new object can be added to a scene by creating a new instance of a 3D model class. For example, in Figure 4 a new alien object is created from the Alien class, which is a 3D model in the Alice Gallery. When an object is added to a scene, the name of the object is automatically added to the **object tree** (upper left of the Scene editor), as shown in Figure. 5.

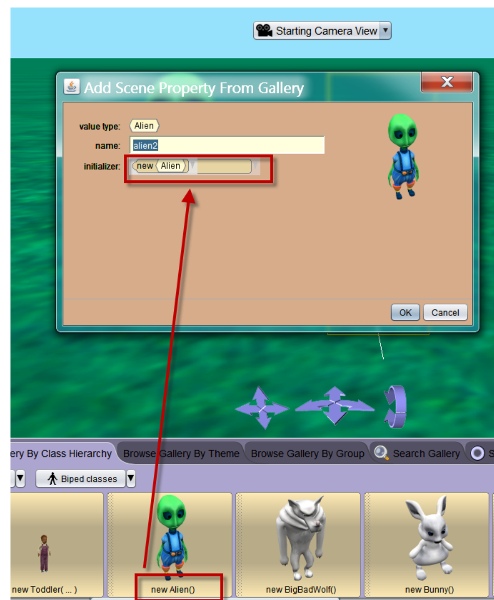


Figure 4 Creating a new Alien object from the Alien 3D model class

Figure 5 New Alien object has been added to the scene and the object tree

CLASS TREE

In addition to the object tree (shown above in Figure 5) in the Scene editor, Alice 3 also maintains a class tree in the Code editor. The class tree can be viewed in a pull-down menu, as shown in Figure 6. In this example, the list of classes includes: Scene, Biped, and Alien. Alice projects always have the Scene class. Other classes in the list will vary depending on which objects are added to the scene and which preferences have been selected. In Figure 6, you may notice that the Alien class tile is indented beneath the Biped tile. This is because the Alien class is a specific type of Biped.

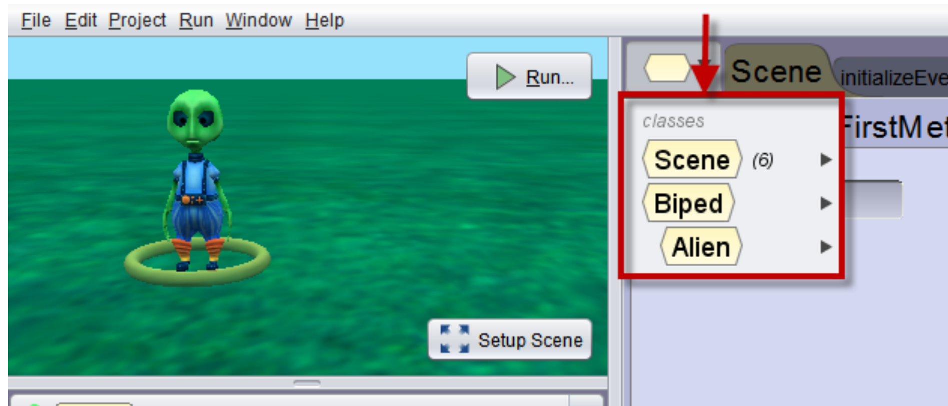


Figure 6 The class tree in the Class menu

VIEWING A CLASS FILE IN THE CODE EDITOR

Selecting one of the classes in the class tree opens a class tab in the Code editor. A class tab displays an overview of the methods defined in that class. For example, Figure 7 illustrates a class tab for the Alien class. The class tab contains three components (procedures, functions, and properties).

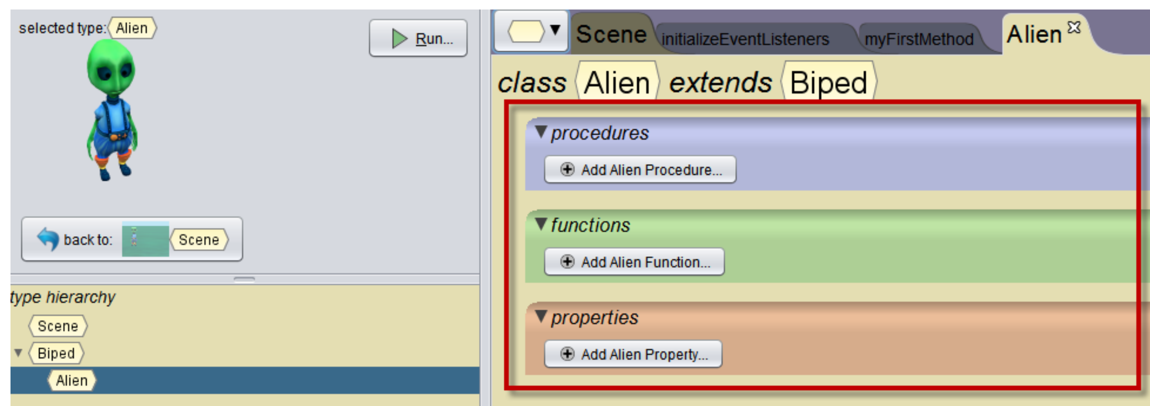


Figure 7 Alien class tab in the Code editor

If Constructors have been enabled in the preferences menu, a fourth component named constructors, is also displayed, as shown in Figure 8. A constructor is a special kind of method that contains instructions for creating a new object as defined by this class.

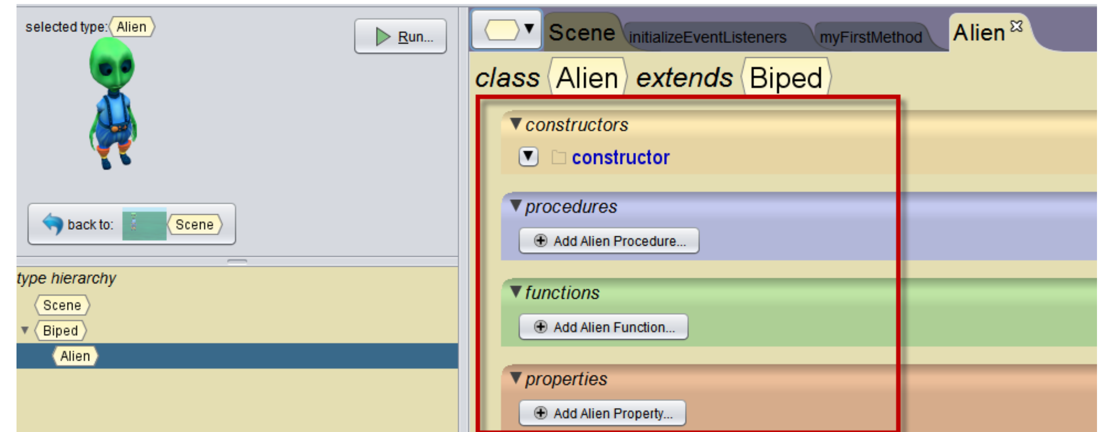


Figure 8 Method categories on a class panel

Although the Methods panel is normally displayed in the lower left corner of the Code editor window, when a class tab is opened in the Code editor, the Methods panel is replaced with a class hierarchy diagram, as illustrated in Figure 9.

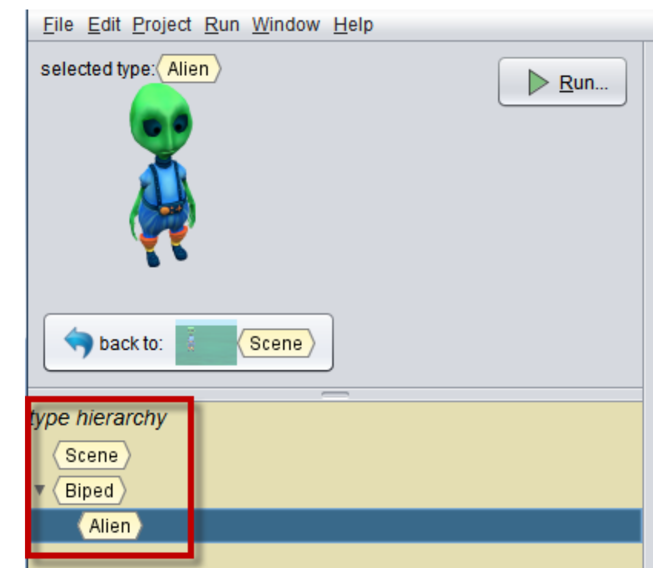


Figure 9 Hierarchy of classes in this project

GALLERY ORGANIZATION

The 3D model classes in the Gallery are organized into collections for the purpose of making it easy to find a specific model or type of model. (**Note:** New models are still being developed by members of the Alice

team. Each update of Alice 3 will likely include new models. For this reason, screen captures in this How-To guide may occasionally vary from what is displayed on your computer.)

The Gallery has five tabs: three for browsing, one for searching, and one for shapes/text. Each of the three browsing tabs organizes the 3D models into collections:

Class Hierarchy – organized by mode of mobility, how an object "gets around" in a scene (for example, Biped, Flyer, Quadruped), as illustrated in Figure 10.

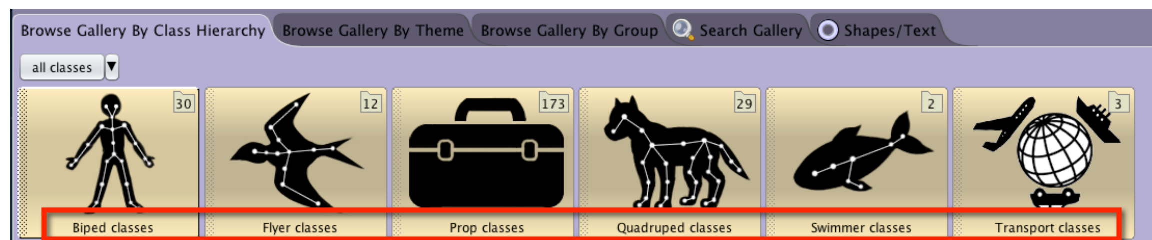


Figure 10 Browsing by Class Hierarchy

Theme – organized by region (for example, Amazon, Far East, Southwest) and by folklore context (for example, Fantasy, Wonderland), as illustrated in Figure 11.

Figure 11 Browsing by Theme

Group – organized in common storytelling categories (for example, Animals, Characters, Scenery), as illustrated in Figure 12.

Figure 12 Browsing by Group

One way to think about browsing the gallery is that each collection is like a drawer in a file cabinet, as shown in Figure 13. A collection contains classes that share some common feature. For example, in the Class Hierarchy tab, the common feature is the mode of mobility -- how an object "gets around" in a scene. Bipeds walk on two legs, Quadrupeds walk on four legs, Flyers use wings, Swimmers use fins, and Vehicles move on wheels. (Props, not depicted in Figure 13 are stationary – do not move around on their own.)

Figure 13 Common features used in Class Hierarchy tab

To view the classes in a collection, click on the icon for that collection. In the example shown in Figure 14, we clicked on the Flyer collection. A scroll bar at the bottom edge of the Gallery panel can be used to view the complete list of classes in this collection. These classes are in the Flyer folder because they each represent an entity that has two wings for flying and moving around the scene. Notice, however, that each has its own unique properties. For example, the ostrich has black and white feathers, the chicken has a comb, and the flamingo has long legs.

Figure 14 3D models in the Flyer folder

How to find a model in the Gallery

One way to find a specific 3D model in the Gallery is to take advantage of the organization system. In the Class Hierarchy tab, one would first think about how the desired object moves around…does it walk on two legs or four legs, or fly, or swim, or roll on wheels? Then, click that class folder and use the scroll bar to find the specific model. For example, to look for a Falcon, select the Flyer folder because a Falcon is likely to fly. Then, click the Falcon thumbnail sketch, as shown in Figure 15. Falcon objects belong to the Falcon class in the Flyer folder.

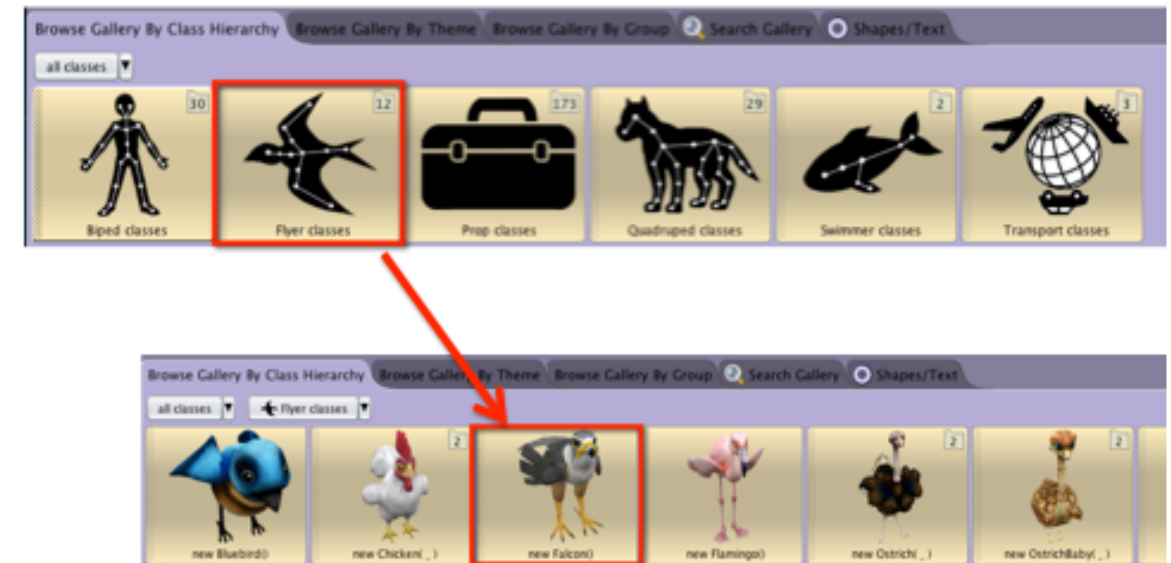


Figure 15 Finding a Falcon using Class Hierarchy organization

An alternate way to find a specific type of model is to use the Gallery's Search tab, as shown in Figure 16. To activate the search box, click the textbox on the tab. The mouse cursor should begin to blink in the box. Enter a descriptive word for an object. For example, in Figure 16, we started typing "cat" and Alice displayed models where "cat" is a significant part of the name. The more characters typed, the more Alice narrows down the possible matches.

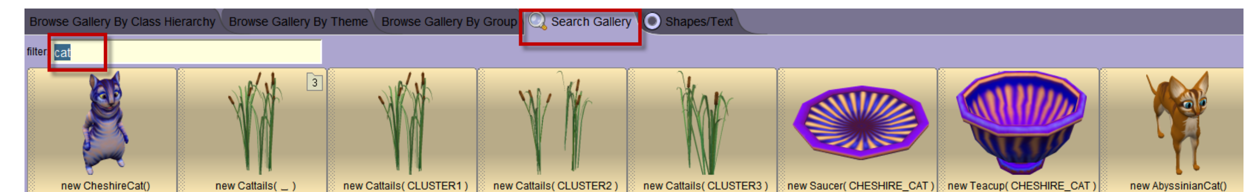


Figure 16 Using the search box

Shapes / Text

The last tab in the Alice Gallery provides 3D models for adding geometric shapes, 3D text, and billboards (importing 2D images) to the scene, as shown in Figure 17.

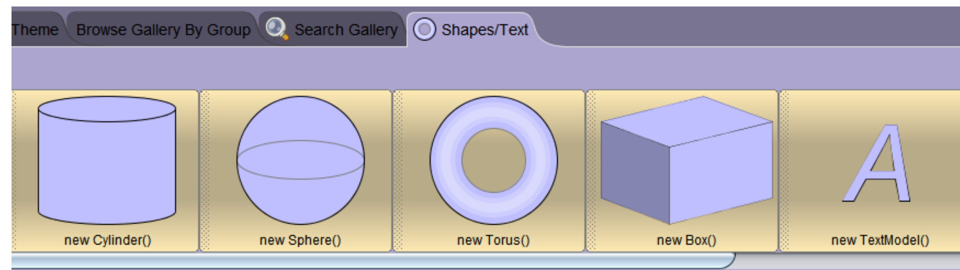


Figure 17 Shapes / Text in the Alice Gallery

CHAPTER 2

SETTING UP A SCENE

The goal of this chapter is to provide information and instructions for setting up a project scene by adding and manipulating characters, props, the camera, and other properties of the Scene class in Alice 3.

SECTION 1

∏

How to position, orient, and resize an object

The purpose of this section is to illustrate the use of handles (ring and arrow mouse controls) and also to introduce the Scene editor's Undo and Redo buttons.

Video: Using Handles to Position Objects

Ring and arrow handles are controls used to interactively position and orient an object in a scene. An additional arrow handle is used for resizing an object.

Note: We recommend using a mouse for working with ring and arrow handles in the Scene editor. A touchpad on a laptop is usable, but takes much more patience.

POSITION AND ORIENTATION

An object's position in a virtual world is tracked as an (x, y, z) location, relative to the center of the virtual world (0, 0, 0). Position, however, is not the only important factor needed to describe an object's location in a 3D world. Each object also has orientation. That is, an object lives in 3D space and thereby has a sense of direction in three dimensions. An object's senses of up and forward are used to define its **orientation**.

In Figure 1, an axes object has been embedded in the hare to illustrate the hare's sense of direction. The green arrow points upward, the white arrow is forward, the blue arrow backward, and the red arrow right, from the point of view of the hare. Although we described the orientation as though there were four separate arrows, this is not really true. The forward and backward arrows are actually just one continuous arrow but the two portions of the arrow are painted different colors to provide a better visual perspective.

Figure 1 Orientation is defined by an object's sense of up and forward directions

Orientation is important for an Alice object because motions such as move, turn, and roll are specified in terms of direction. For example, the hare may be told to move forward or move up. When an object performs a motion statement, it does so relative to its own orientation. In this example, if harry is told to move left he will move to his left. To be clear, in the scene shown in Figure 1 above, harry would move to his left which is to the right of the scene as seen by the camera. As a rule of thumb, an object's motion is generally performed in a self-centric manner.

Undo and Redo buttons

A sense of "freedom to play" when setting up a scene is provided by two buttons, Undo and Redo, in the upper right corner of the Scene editor, as shown in Figure 2.

Undo provides the ability to “make a mistake” and fix it. A click on the Undo button backtracks the most recent action and the state of the scene backs up one step, removing it. It is possible to click Undo repeatedly, backtracking all the way to the initial state of the project when it was first opened in this session (but not into previous sessions that were saved and later reopened). Redo provides the ability to "change your mind." Click the Redo button to reverse the action of an Undo. Redo also provides the ability to repeat an action.

Figure 2 Undo and Redo buttons in the Scene editor

Hint: Use the Undo and Redo buttons make it easier to set up a scene … without tension or fear of breaking something.

Handles: Rings and arrows

By default, the mouse can be used to click and drag an object forward/ backward and left/right on the horizontal plane in a scene. Handle style controls create rings and arrows that can be used to modify the mouse's drag action in the Scene editor. Each handle action is summarized in Figure 3.

Figure 3 Handles change the drag action of a mouse on an object

Single ring

When an object is first added to a scene, the Handle style displayed is usually a single rotation ring around the pivot point of the object, as shown in Figure 4. Using the mouse to click on the ring and drag the ring in a clockwise or counterclockwise direction causes the object to mimic the mouse action, rotating in the same direction as the ring is being turned.

Rotating an object with the single ring handle changes the orientation of the object by changing the forward and backward directions. (It is possible, however, for the object to end up facing in the same direction it was originally facing. In this case, the orientation is returned to its original value.)

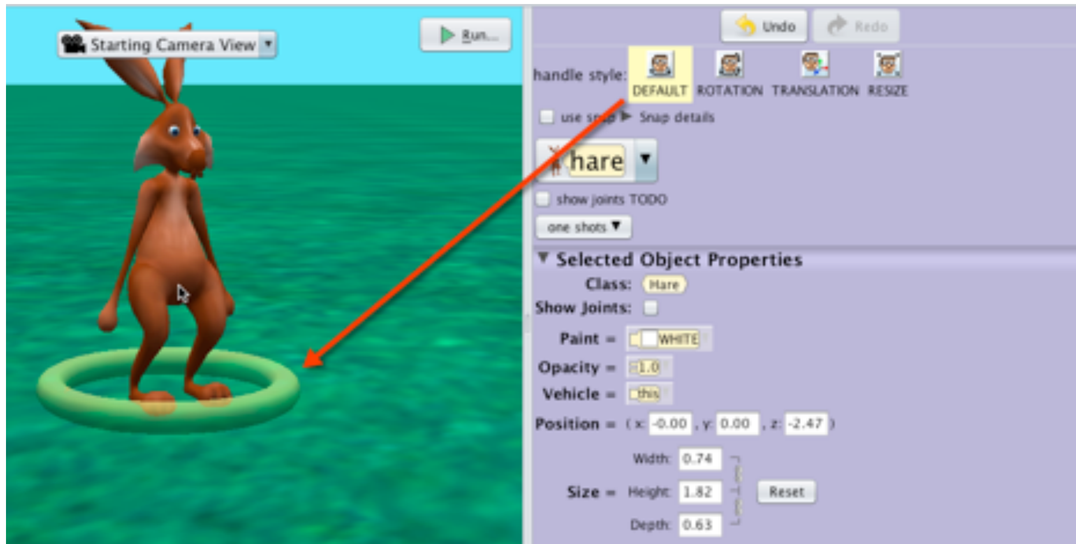


Figure 4 One ring to rotate an object left/right

THREE RINGS

The three rings handle is used to turn an object left/right (turn around), turn an object forward/backward (tilt), or roll an object left/right (similar to a door knob), as shown in Figure 5. Rotating an object with any of the rings changes the orientation of an object. The turn ring changes the forward direction. The tilt ring changes the forward and up directions. The roll ring changes the up direction. (Once again, it is possible to rotate in such a way that the orientation returns to its original value.)

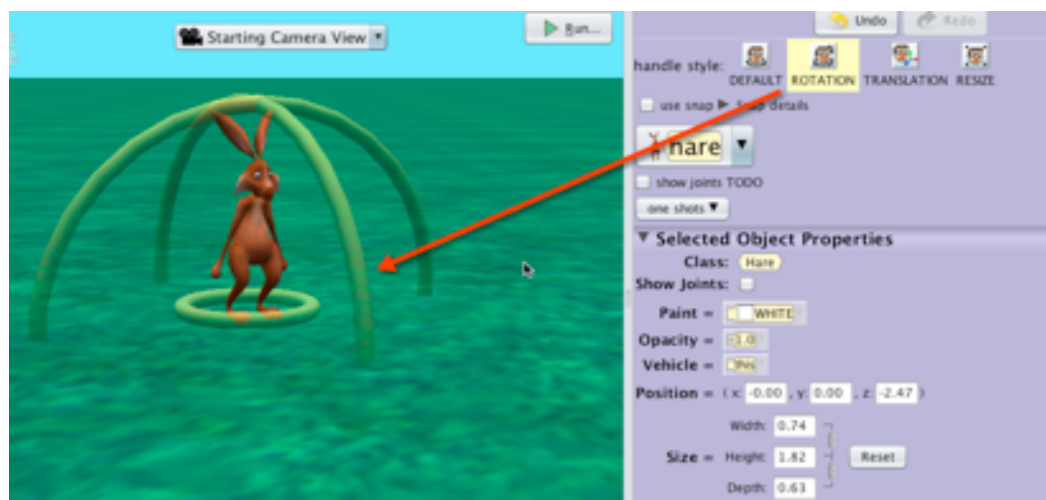


Figure 5 Three rings to turn or roll

As a short example of the usefulness of the ring handles, in Figure 6 we added a 3D text object to the scene. Note that the text is somewhat dark. The lighting in a scene is directly overhead. To get better lighting on the text, the text can be tilted slightly backward.

Figure 6 A Text object, "Hello. World!" ld

Look closely at the text object and the rotation handle button shown in Figure 7. When the rotation button is clicked, three rings are displayed around the pivot point of the text object.

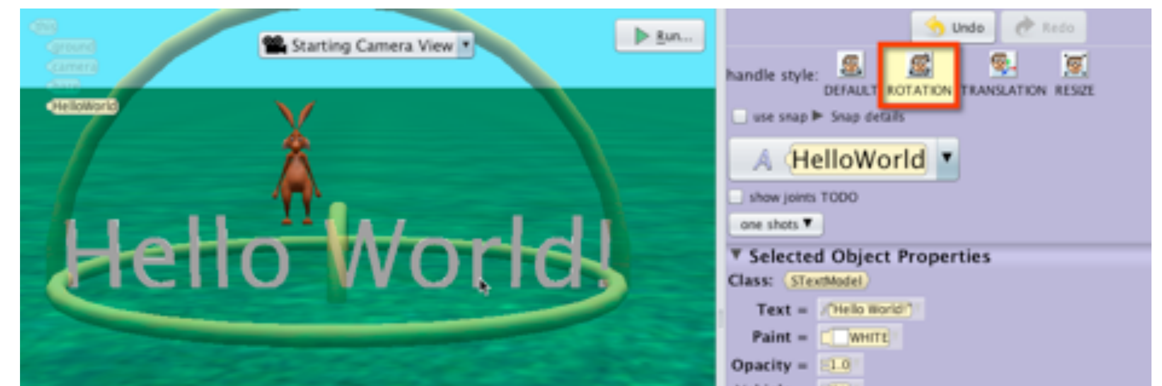


Figure 7 Three-ringed handle for rotating an object

The forward/back ring was used to tilt the text string slightly backward (toward the back of the scene). The text object appears brighter, as shown in Figure 8.

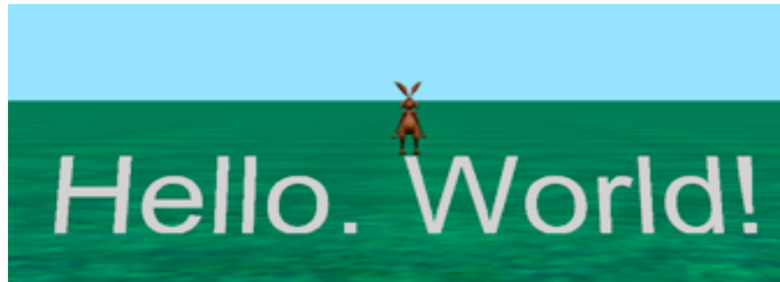


Figure 8 3D Text has better lighting

THREE ARROWS

The third handle button displays translation arrows (rather than rotation rings), as shown in Figure 9. The translation arrows can be used to move an object in any of six directions (up, down, left, right, forward, or backward. The three translation arrows change an object's (x, y, z) coordinate location in the virtual world. However, the orientation of the object remains the same. (As with orientation, it is possible to move an object in such a way that it returns to its original location.)

Note: The direction of motion for the translation arrow handles perform "as seen by the camera" instead of "as seen by the object."

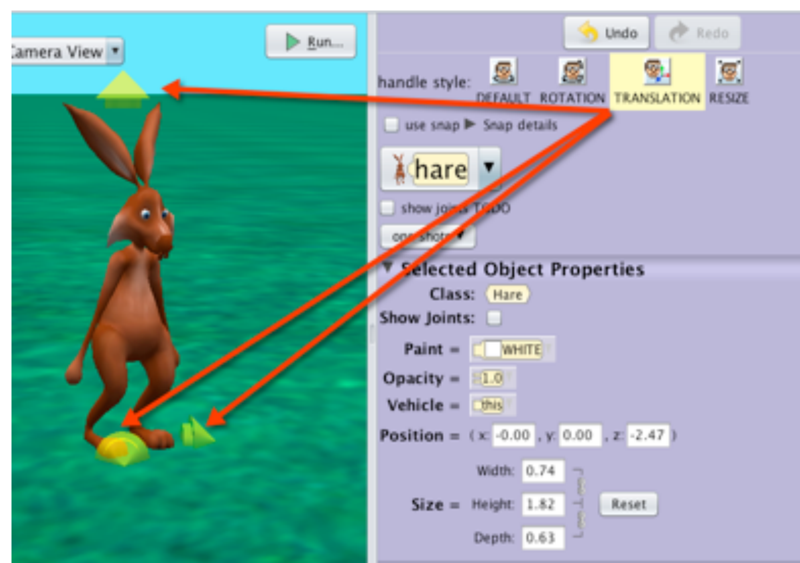


Figure 9 **Three arrows** to move an object as seen by the camera

The fourth handle style button displays a single arrow handle that can be used to resize an object, as shown in Figure 10. The single arrow changes the object's size in all directions, proportionately. The single arrow handle offers a more free-styling control for resizing as compared to the specific accuracy of the Position (Width, Height, and Depth) property boxes in the Setup. The single resize arrow does NOT change the orientation of the object.

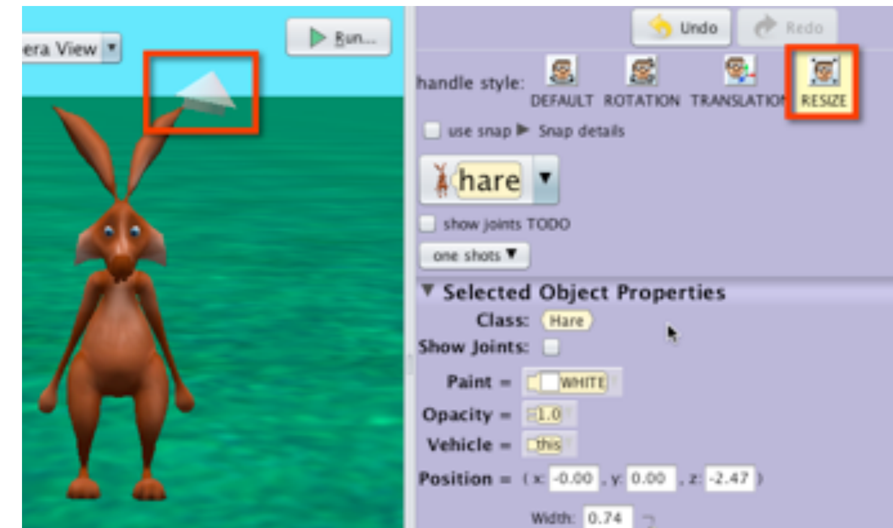


Figure 10 Single arrow resizes proportionately in all dimensions

A geometric shape has the additional capability of resizing in a single dimension. For example, the box shape has four resize arrows, as shown in Figure 11. The upward arrow resizes the cone's height without affecting its width or depth. The magenta arrow at the base resizes the cone's width without affecting its height and depth. The aqua arrow resizes depth without affecting it height or width. The pink arrow (diagonally off to the upper right side) resizes proportionately in all directions.

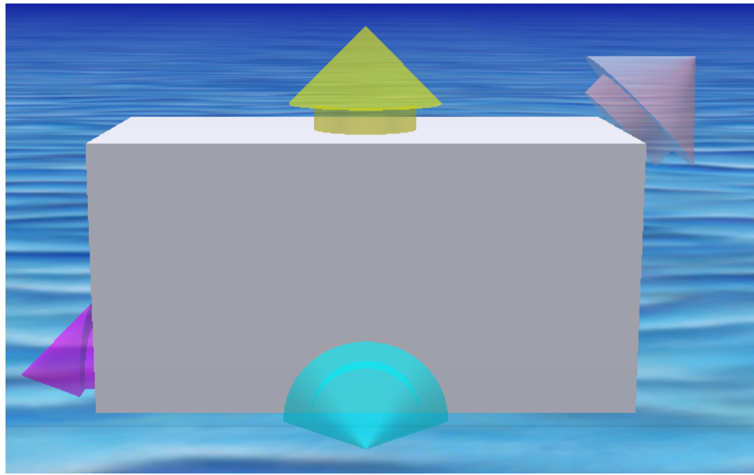


Figure 11 Four resize buttons for a geometric shape

SECTION 2

∏

Adding an object to a scene

A brief introduction for adding an object to a scene was previously presented in Part 1, Section 5 of this How-To guide and also in the video, entitled Adding Objects to a Scene. In this section of the How-To guide, more detailed illustrations are provided, with special attention devoted to different kinds of objects.

Video: Adding Objects to a Scene

The illustrations will typically make use of the **Browse by Class Hierarchy** tab in the Gallery, although other Gallery tabs are equally useful. In addition, we will illustrate how to add an object that is a Sims 2 person as well as objects from the Shapes/Text tab.

ADD AN OBJECT – TWO TECHNIQUES

We will illustrate two different techniques for adding an object to a scene. Which technique you use is a matter of comfort and style. One way to add an object is to single-click the thumbnail sketch of the desired object in the gallery. A dialog box is displayed where a name for the object can be entered (or a default name can be accepted), as shown in Figure 1. The name should be all one word (no spaces) and should begin with a lowercase letter of the alphabet. To use two or more words, use camelCase which avoids spaces by starting with a lowercase letter for the first word and then uses a capital letter for each additional word. For example, the *alien* might be named *greenAlien*. Click OK when done.

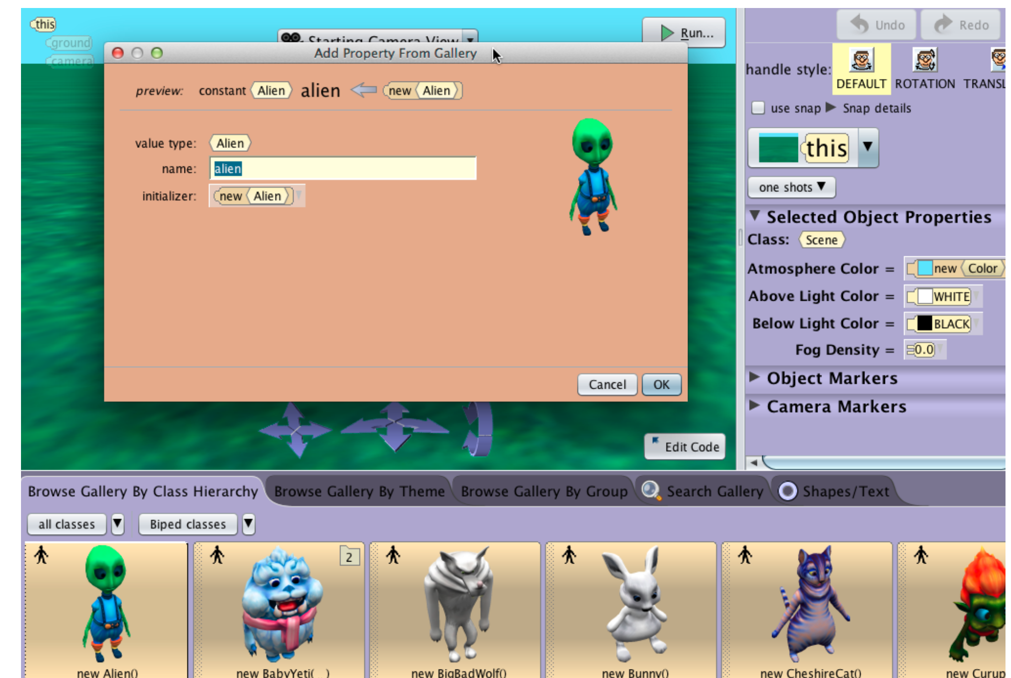


Figure 1 Adding an object to a scene

The second technique for adding an object to a scene is to click and hold the left mouse button on the thumbnail sketch and drag it into the scene. The display of the mouse cursor will change to a box-like outline, as shown in Figure 2. This is a bounding box that shows where the object will be located when the mouse button is released. When the mouse cursor is released, a dialog box pops up where a name for the object can be entered (or accept the default name) in exactly the same way as described above.

This technique of adding an object to a scene allows the user to control where the object will be positioned in a scene.

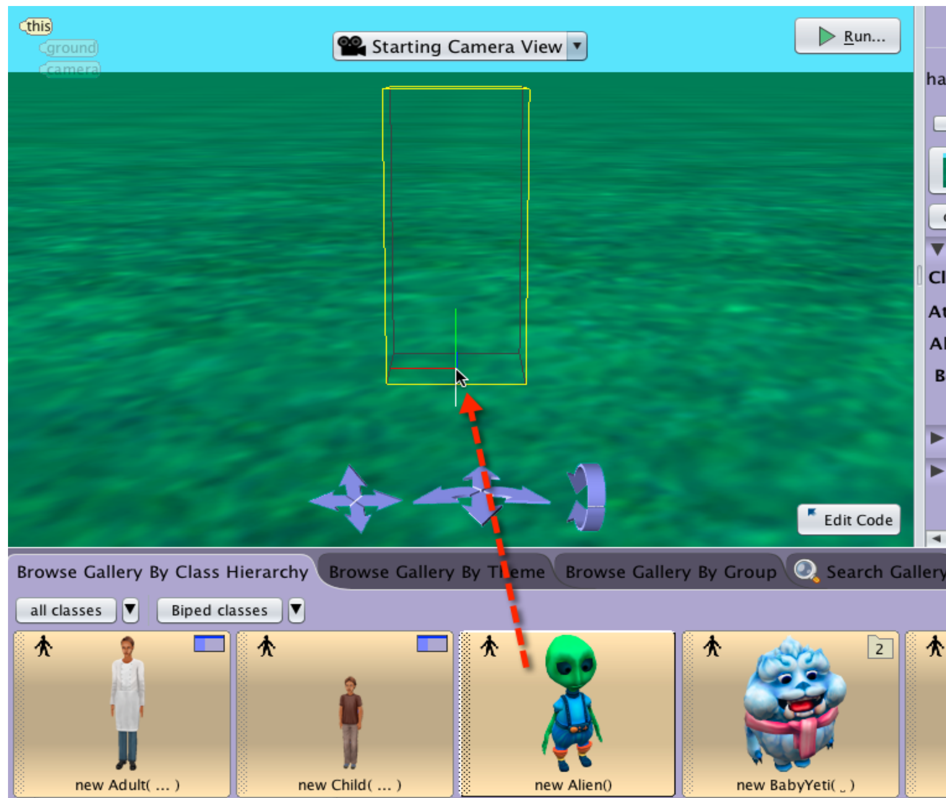


Figure 2 Click and drag thumbnail sketch into the scene

Regardless of which technique is used to add an object to the scene, the new object is displayed in the scene and the name of the object is automatically added to the Object tree in the Scene Editor, as shown in Figure 3.

Figure 3 A new object's name is automatically added to the Object tree

Multiple objects

It is possible to add more than one object of the same class. It is also possible to construct different objects from different classes in the same scene. Figure 4 shows four different objects in an Alice scene, each constructed from a different class.

Figure 4 Objects of different classes in an Alice scene

Add a Sims 2 person

In the Gallery's Class Hierarchy tab, select the Biped classes tab, as shown in Figure 5. The first five 3D Models (left-hand side) are Sims2 people classes. The Sims2 models represent people at various stages of life (elder, adult, teen, child, and toddler). To add a Sims2 person to a scene, click on a sketch for one of the life-stages. In the example shown here, we clicked on the Adult life-stage image, but you may choose any of the models.

Figure 5 The Person class defines Sims 2 people objects

When one of the life-stage models is selected, a Person Builder window is displayed to provide options for: life stage, gender, skin tone, outfit (full or top/bottom), waistline slider, hair and hat style, and facial features.

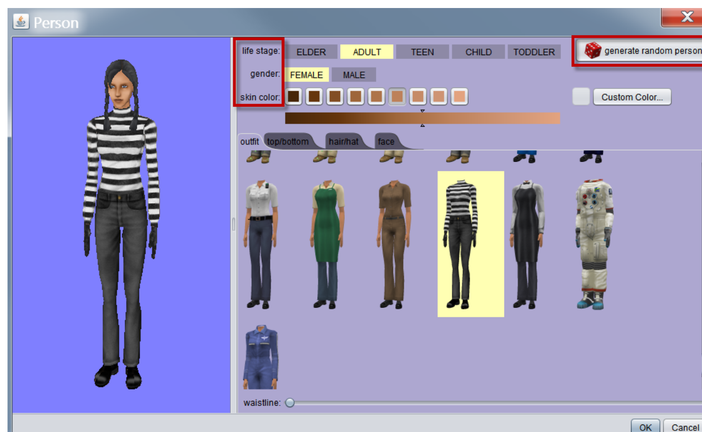


Figure 6 Sims 2 people-builder

Select features for each option and then click OK. A naming dialog box will pop up, as shown in Figure 7. Note that the features selected in the people-builder options are listed as "resources" to be used in constructing the person object. In this context, a resource is a painted image that is used to create the object's appearance. In the naming dialog box, enter a name for the person and click OK. The person object will then be added to the scene.

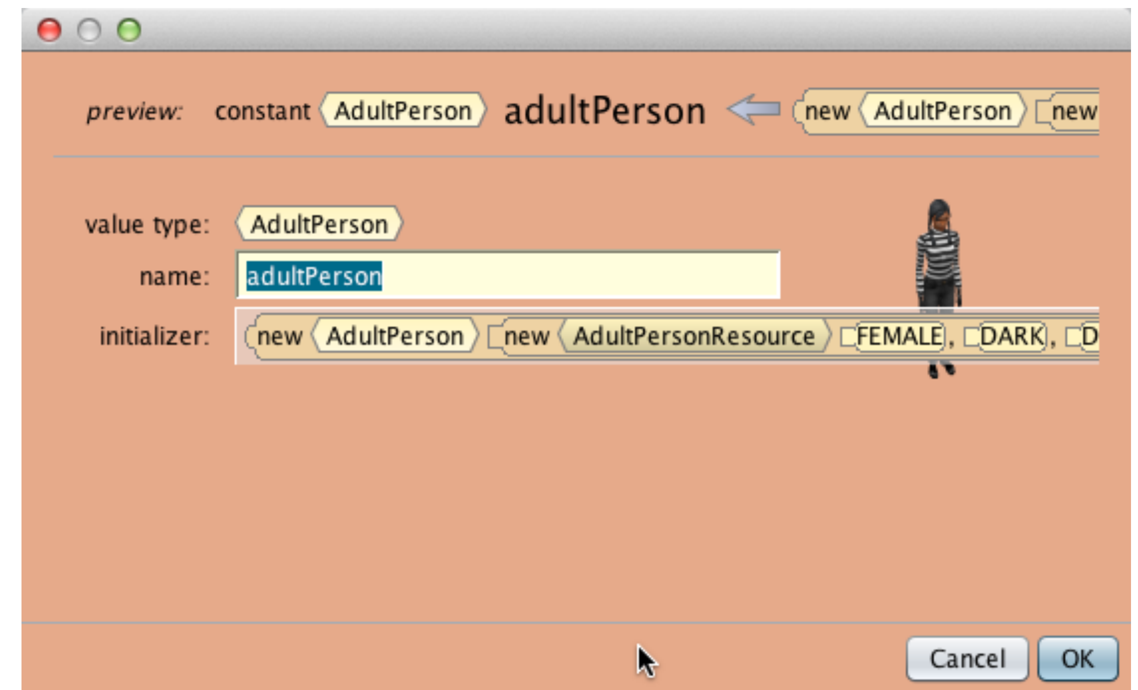


Figure 7 Naming a Sims2 person

Add a Geometric Shape object

In addition to the models in various collections, the Gallery also has a few basic, geometric shapes (disc, cone, cylinder, and sphere) and 3D Text models. To create a geometric shape, click one of the thumbnail sketches, as shown in Figure 8.

Figure 8 Geometric shapes in the Gallery

When a shape model is selected, a dialog box is displayed for entering a name for the new object, as illustrated in Figure 9.

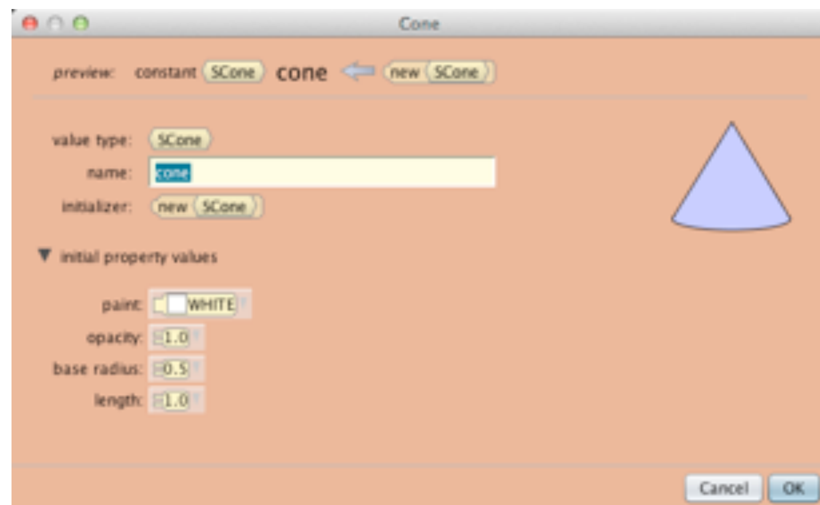


Figure 9 Naming a geometric shape object

The new object can be positioned in the scene and properties can be set, as shown in Figure 10. Details for setting the properties of an object are provided in Section 7 of this How-To guide.

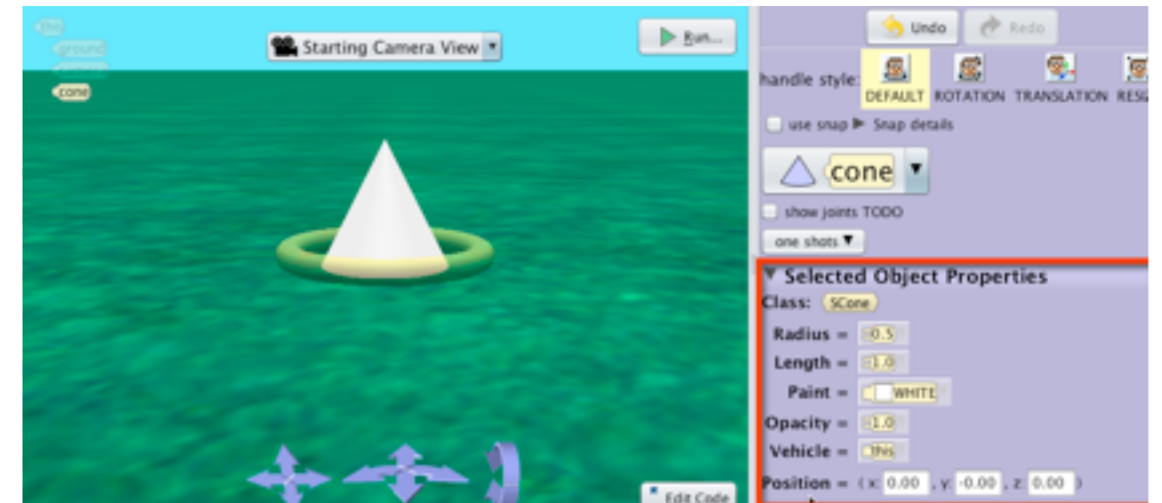


Figure 10 Properties can be set for painting, resizing, and other modifications

ADD 3D TEXT

To create an instance of the TextModel class, click the TextModel thumbnail sketch in the Gallery, as shown in Figure 11. 3D text is useful for displaying screen credits, a timer, or a scoreboard for a story or game.

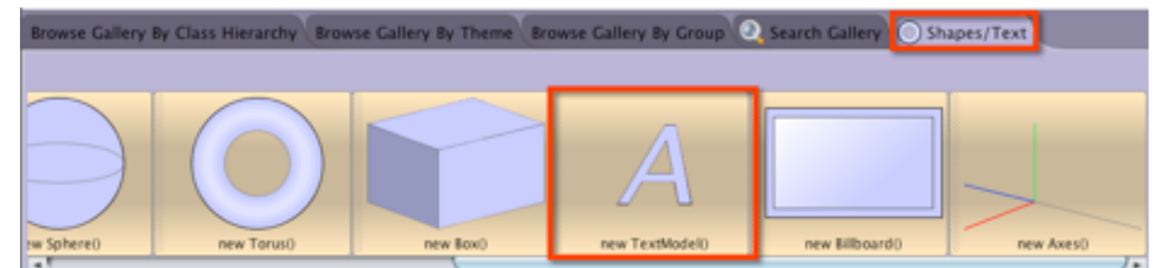


Figure 11 TextModel button to create 3D text object

When the TextModel sketch is clicked, a dialog box is displayed where two items of information must be entered, as shown in Figure 12. The first item is a name for the 3D text object. The second item is a string of text characters that will be displayed by the text object. The drop down menu for a string of text characters (a TextModel object) allows a string to be empty (“”), the default value “hello”, or a Custom TextString entered using the keyboard.

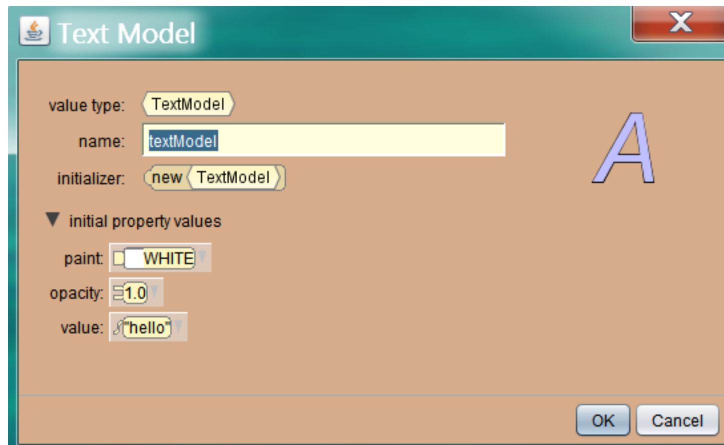


Figure 12 Text Model dialog box

An example is shown in Figure 13, where we entered "Scoreboard" as the name and selected Custom TextString. Then, in the pop-up Custom TextString box, we entered "0" as the value of the text to be displayed. Just to be clear, note that the name of the object is Scoreboard and the text string it displays is "0". That is to say, the name of a text object and the text string it displays are not necessarily the same.

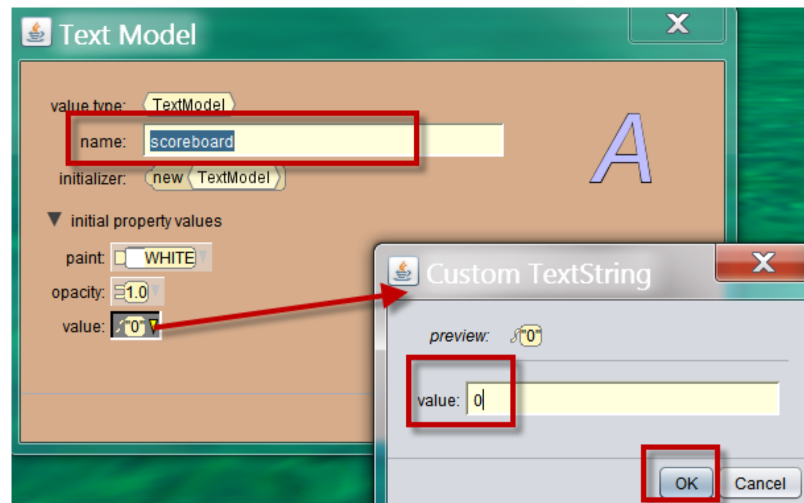


Figure 13 Naming a text object and initializing the text string (alphanumeric)

The new 3D text object will be displayed in the scene, as illustrated in Figure 14. The text object can be positioned in the scene, and its properties can be modified in the Setup panel on the right.

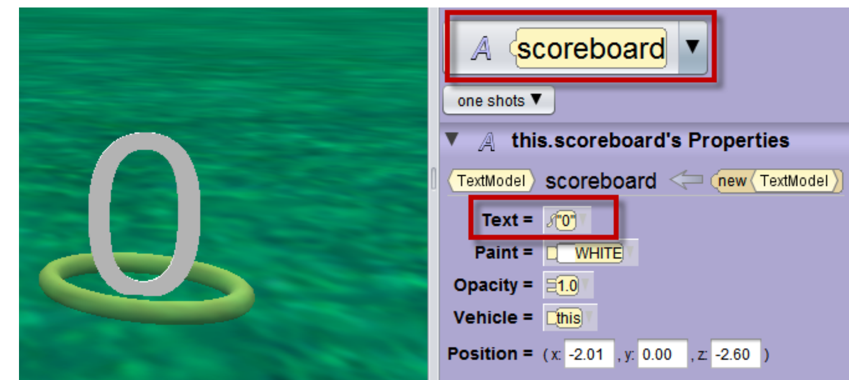


Figure 14 A 3D text object

ADD A BILLBOARD (2D IMAGE RESOURCE)

A 2D image may be added to a scene as a billboard. Billboards are useful as backdrops, a narrative element in a story, and for presenting instructions on how to play a game. To create a billboard from a 2D image, click the Billboard thumbnail sketch, as shown in Figure 15.

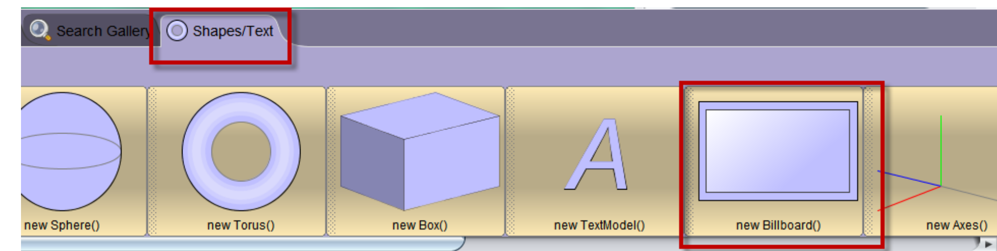


Figure 15 Billboard in the Shapes/Text Gallery

When the Billboard is selected, a dialog box is displayed, as shown in Figure 16, where three items of information must be entered.

- The first entry item is a name for the object.
- The second item is a drop-down menu for finding and importing a 2D image or to create a billboard of a solid color. The file format of a 2D image must be .jpg, .png, .bmp, or .gif (must be all lower-case).

- The third item is a drop-down menu to select an image or color for the back of the billboard. (As a 2D object, a billboard has two sides: front and back.)

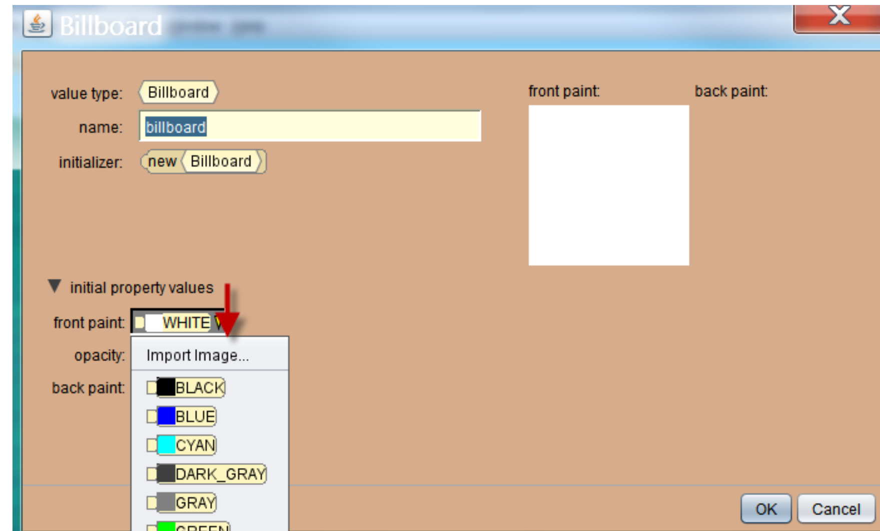


Figure 16 Billboard dialog box requires a name and image or color

An example is shown in Figure 17, where we accepted the default name "billboard" as the name for the new billboard object, selected an image source (an Alice Team photo) and then, in the pull-down menu for the back of the billboard, selected a solid black color.

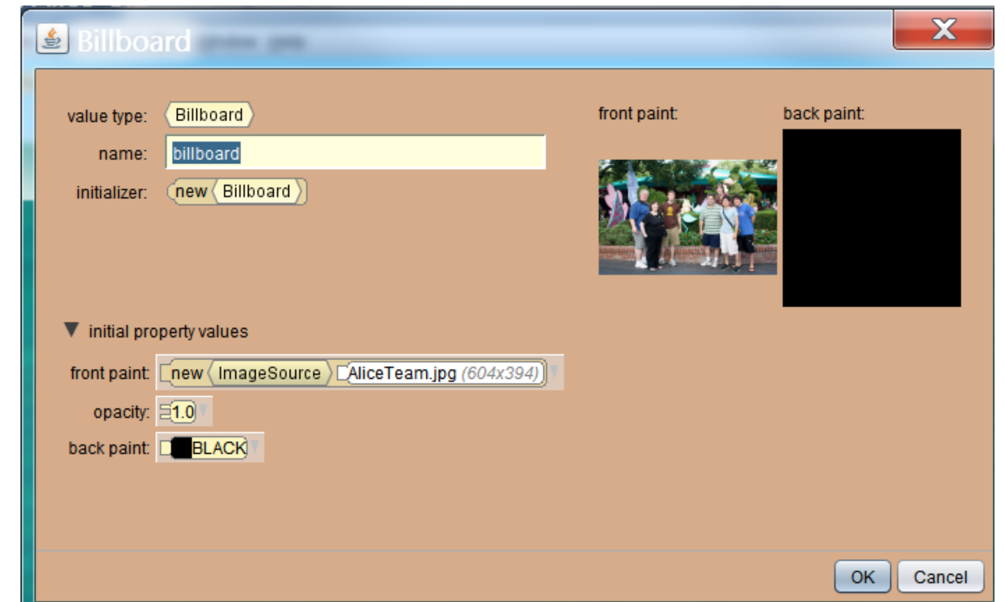


Figure 17. Example Billboard entries

When OK is clicked, the object is displayed in the scene, as shown in Figure 18. The image can be positioned in the scene and its properties can be set in Setup.

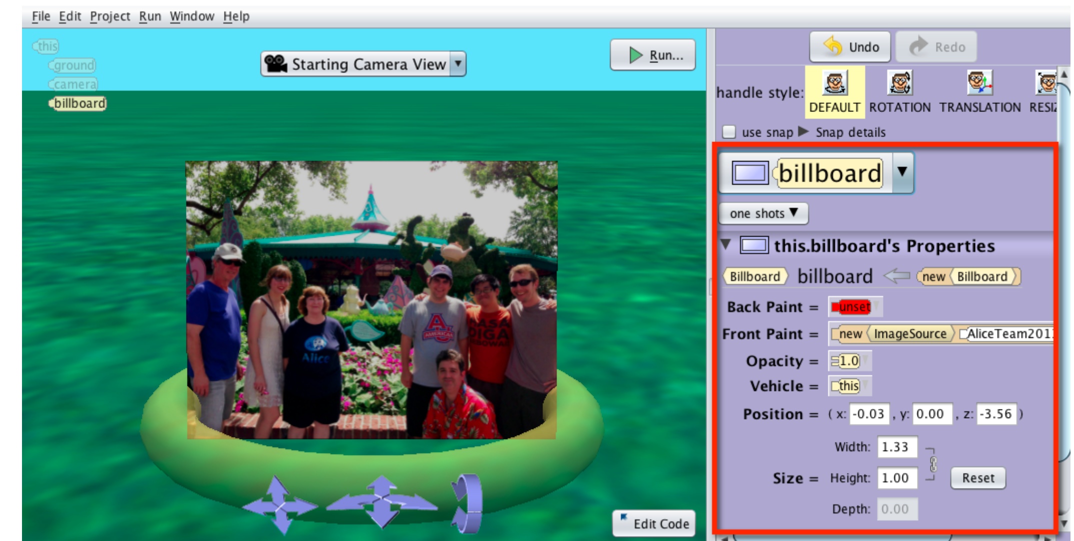


Figure 18 Alice team photo as a Billboard object in a scene

Once a 2D image is added to a scene as a billboard, the image will show up in the list of resources found in the Project menu under Manage Resources, as shown in Figure 19.

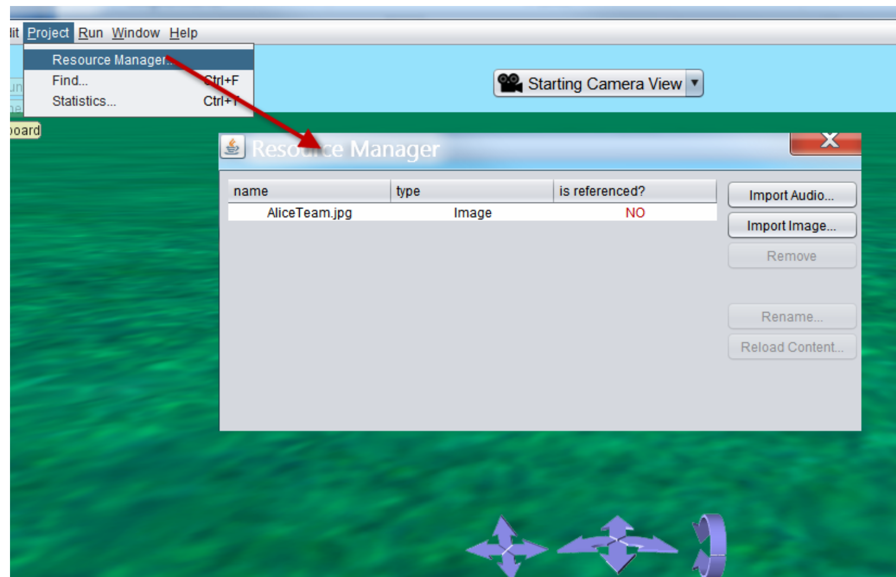


Figure 19 Billboard image is listed in Manage Resources

ADD AN AXES OBJECT

To create an Axes object, click the Axes thumbnail sketch, as shown in Figure 20.

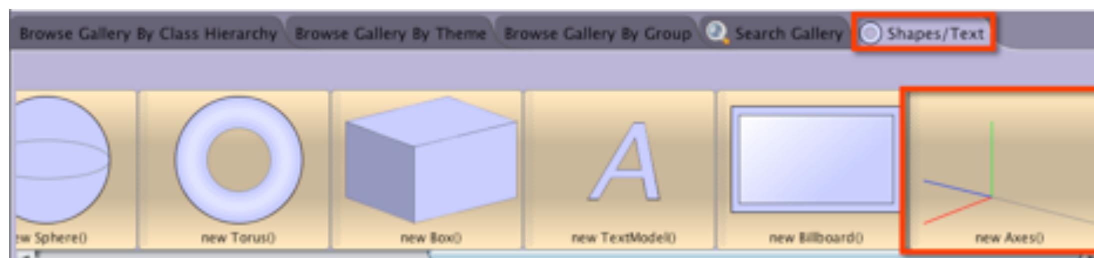


Figure 20 Axes button in the Gallery

When the Axes is selected, a dialog box is displayed, as shown in Figure 21, a name is entered for the object. Click OK. An Axes object will be added to the scene, as shown in Figure 22

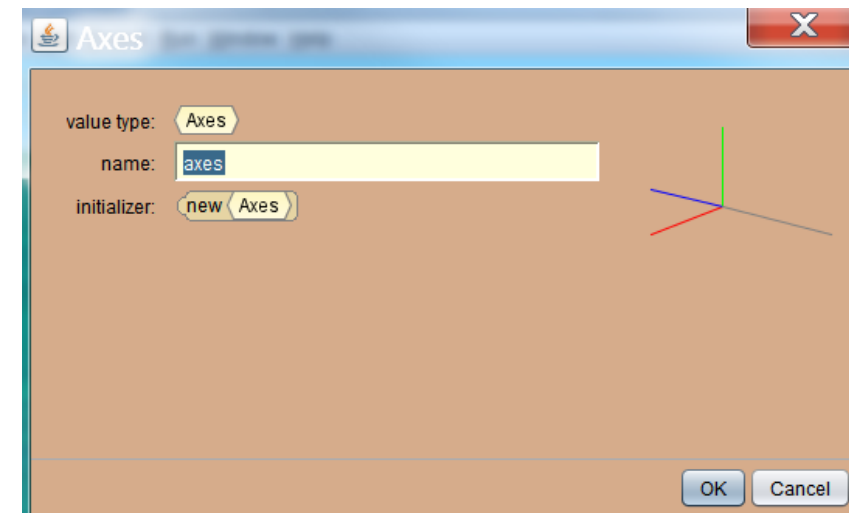


Figure 21 Naming a new Axes object

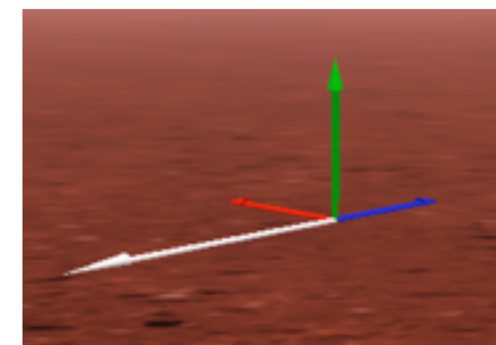


Figure 22 A new Axes object in a scene

USING AN AXES FOR ORIENTATION

The orientation of an object and its skeletal joints is very important when working with 3D objects in a virtual world. One way to determine the orientation of an object (or a skeletal joint within an object) is to create an Axes object and align it to the orientation of the target object or a joint within the object.

To illustrate, we created a new Mars scene with an asteroid, as shown in Figure 23. Just looking at the asteroid, we have no way of knowing its forward direction. To determine the orientation of the asteroid, we selected the axes and used a *moveAndOrientTo* method to move and orient the axes to the asteroid. Then, as shown on the right in Figure 23, the axes arrows are aligned with the orientation of the asteroid. The white arrow of the axes shows the forward direction for the asteroid, the red arrow shows the asteroid's right, and the green arrow shows the asteroid's up direction.

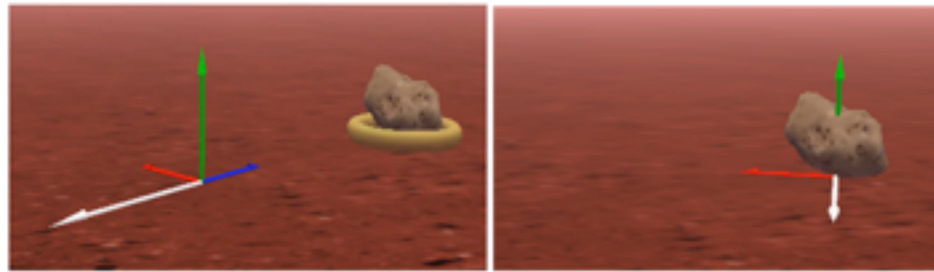


Figure 23 Original axes location (left) and aligned with android (right)

SECTION 3

∏

Setting object properties in the Scene editor

The purpose of this section is to demonstrate how to set a property of an object in the Scene editor. An object's properties are items of data that identify that object as an individual.

Figure 1 Selected object and its properties

For example, a driver's license is a form of identification that typically includes a person's with a photo and their first, middle, and last name, hair color, eye color, skin tone, height, and weight. In a similar way, identifying data about an Alice object include its **Name, Paint, Opacity, Vehicle, Position,** and **Size** (composed of Width, Height, and Depth) properties.

SETUP PANEL

To view property data about an object, first select the object in the scene. When clicked, the selected object will be surrounded by a ring-shaped handle, as shown in Figure 1. (The ring handle is a mouse control that can be used to turn the object left and right.) The selected object's properties are displayed in the Setup Panel just to the right of the scene. In this example, the selected object is named seaweed2.

SET A PROPERTY

The phrase "set a property" means that a new value is specified for that property. The following paragraphs illustrate how to set properties (paint, opacity, vehicle, position, and size).

SET PAINT

Paint includes both the texture map and the color of an object. An object has a wire mesh of polygons that creates the external appearance of the object. A texture map is applied to the mesh surface to create a "skin" coating that encloses the object. For people objects, the "skin" includes hair and eyes and for animal objects the color of fur, eyes, nose, ears, paws, and tail (if appropriate). An object's color is a coating that covers the texture map. By default, a WHITE color coating is actually just a clear coating that does not change the colors on the skin (somewhat like a clear sugar-glaze on a doughnut).

To change the color coating, click the Paint's pull-down menu in Setup and select a color in the menu. In Figure 2, MAGENTA has been selected

and the seaweed is painted with this color. The visual effect, as seen in Figure 2, is a darker color overall.

Figure 2 Setting the color for painting an object

SET OPACITY

One way to think about opacity is as the opposite of transparency. By default, the opacity of an object is 1.0, which means the object is totally opaque (it looks solid). Setting the opacity to 0.0 would mean that the object is totally transparent (it is invisible). The range of values for opacity, therefore, is from a low of 0.0 to a high of 1.0.

To set the opacity of an object, select the object in Setup and then click on the Opacity button. A drop-down menu allows the selection of opacity in a scale of 0.0 to 1.0, as shown in Figure 3. In this example, the seaweed2 object was selected and the opacity was set to 0.4. As can be seen by comparing the seaweed2 object with the other seaweed object beside it, the seaweed2 object has faded and is now partially transparent.

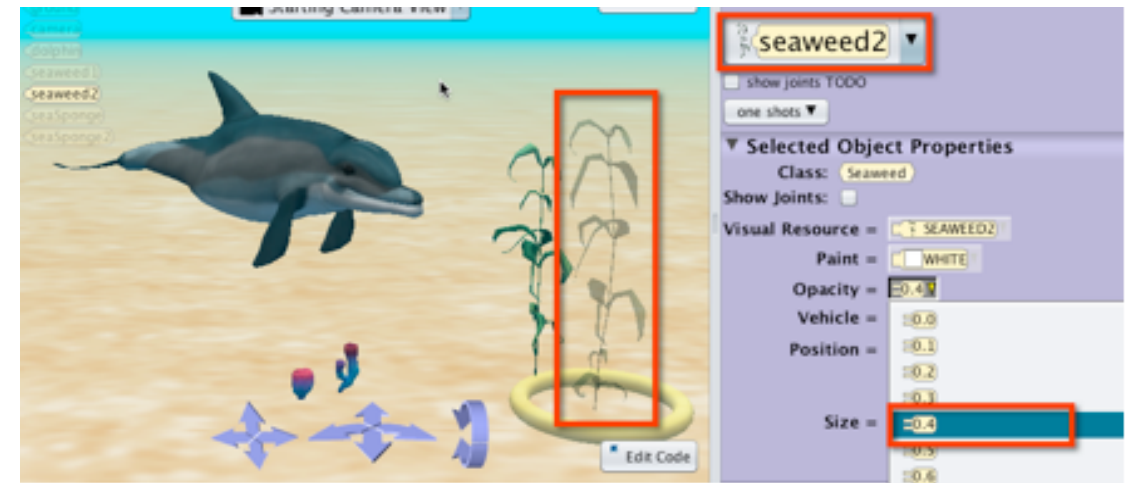


Figure 3 Setting opacity

SET VEHICLE

In Alice, a vehicle is an object whose motions affect the motions of another object in the virtual world. As an analogy, consider a car as a vehicle. When a person is riding in a car and the car moves forward, the person moves forward with the car. In Alice, the current scene is, by default, the vehicle for all objects within it. So, if the scene moves left all objects within the scene would move left with it.

To set the vehicle of an object, first select the object for which the vehicle is to be changed. Then click on the Vehicle button. A drop-down menu allows selection of another object to be the vehicle, as shown in Figure 4. In this example, we added a pajamaFish to the scene. The pajamaFish has been positioned on top of the dolphin's tail, where he wants to hitch a ride with the dolphin. To make this happen, first select the pajamaFish in the Properties Panel. Then, click the down arrow for the Vehicle property and select dolphin from the pull-down menu. Now, if the dolphin moves the pajamaFish will move with it, in the same direction and distance or if the dolphin turns, the pajamaFish will turn with it in the same angle of rotation.

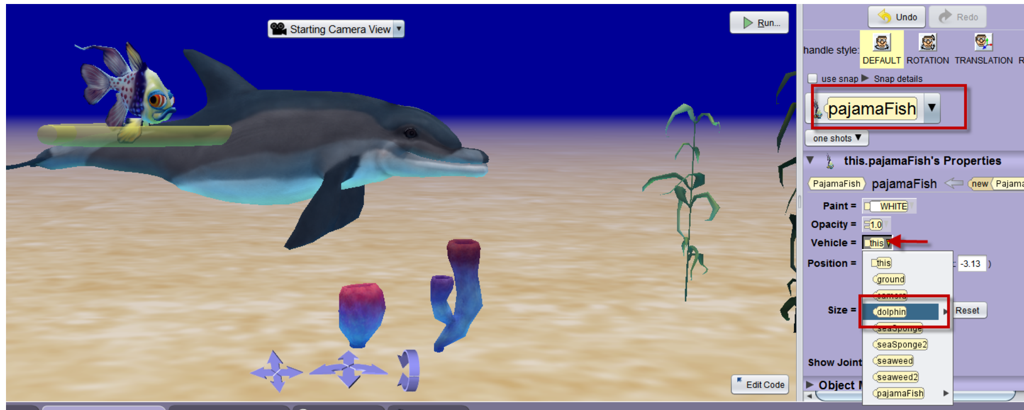


Figure 4 Set the vehicle of pajamaFish to be the dolphin

SET PRECISE POSITION

The position of an object in a scene is relative to the center point of the scene. Using the mouse to drag an object around in the scene is most common method of setting the position of an object in a scene. However, there may be some worlds in which it is important to position an object in an exact location in the scene.

The Setup panel of the Scene editor allows precise positioning of an object by setting its position coordinates. To set the position, click the mouse in one of three coordinate boxes and use the keyboard to enter a numeric value. As an example, in Figure 5 we added a blue cone and positioned it precisely at the center point of the scene (0,0,0). Then, the dolphin was positioned by entering numbers in the position boxes for x (-1.36), y (-0.04), and z (1.35). After the new values were entered and the Enter key was pressed the dolphin was immediately repositioned at that location in the scene.

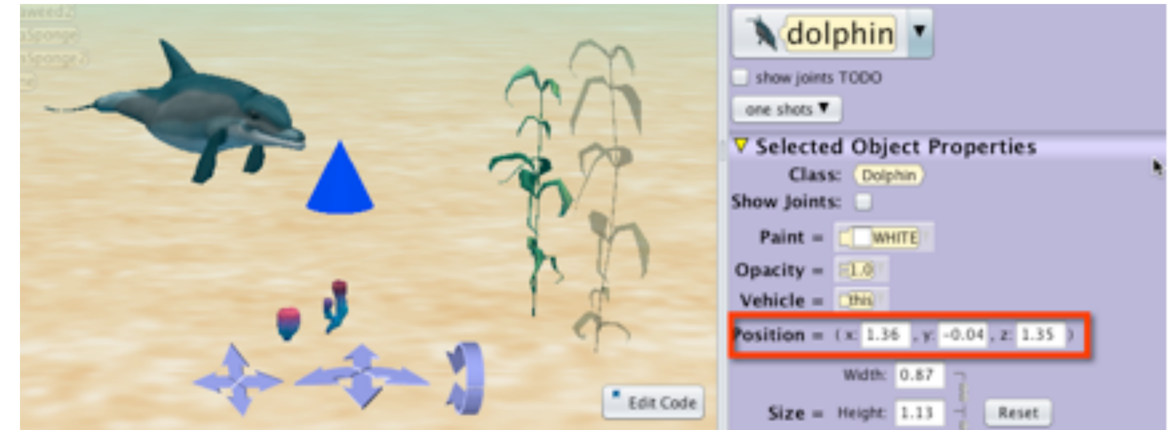


Figure 5 Setting the precise position coordinates of an object

SET SIZE

An object's size has three dimensions: width, height, and depth. The size of an object in a 3D world often needs some adjustment when added to a scene and it appears to be out of proportion with the size of other objects currently in the scene. For example, in Figure 6 the seaSponge object looks very small when compared to the size of the seaweed objects. To change the size of the seaSponge object, first click on the seaSponge object. Then, click the mouse in one of three dimension boxes (width, height, or depth) and use the keyboard to enter a numeric value. By default, changing the size of one dimension automatically updates the other two dimensions, proportionately. If you change your mind about the size change, you can use the Reset key to set the size back to its previous dimensions.

Figure 6 Setting the size dimensions of an object

As illustrated in Figure 6 above, size dimensions are locked to a proportional resize. Therefore, a change in one dimension results in all dimensions changing proportionately. A major exception to proportionate resizing is that it is possible to resize geometric shapes in one dimension only. For example, in Figure 7 a box object has been positioned in the scene. Note that it is a perfect cube, having a width, height, and depth of 1 meter each.

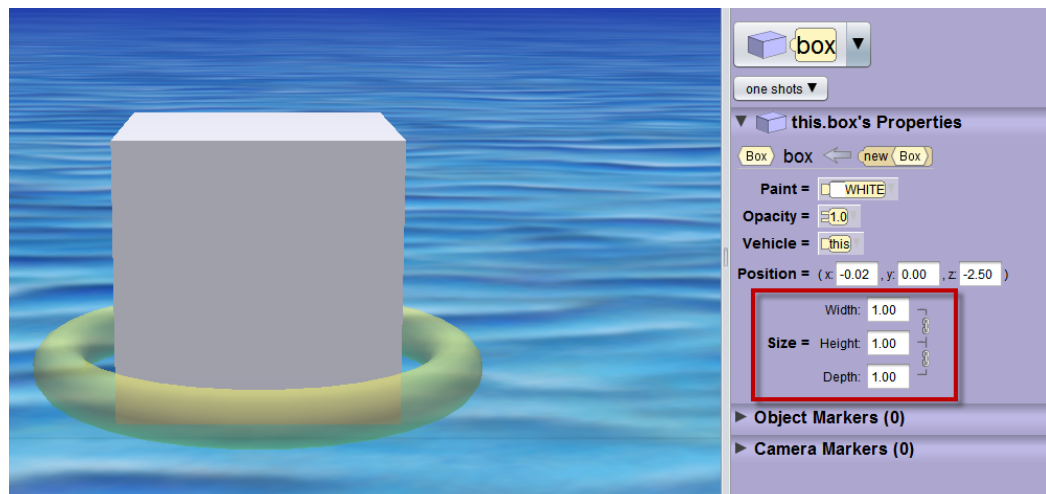


Figure 7 Dimensions of a new box object

As shown in Figure 8, we clicked on the lock icons at the right of the cube to disable proportionate resizing. Then, enter a value in one of the dimensions. That dimension will change and the other two will remain as before. Figure 9 shows the result of changing the box width to 2 meters.

Note: After disabling proportionate resizing, be sure to re-enable it by clicking again on the lock icons at the right. The icon will not automatically revert, because the Scene editor adheres to a principle of maintaining state.

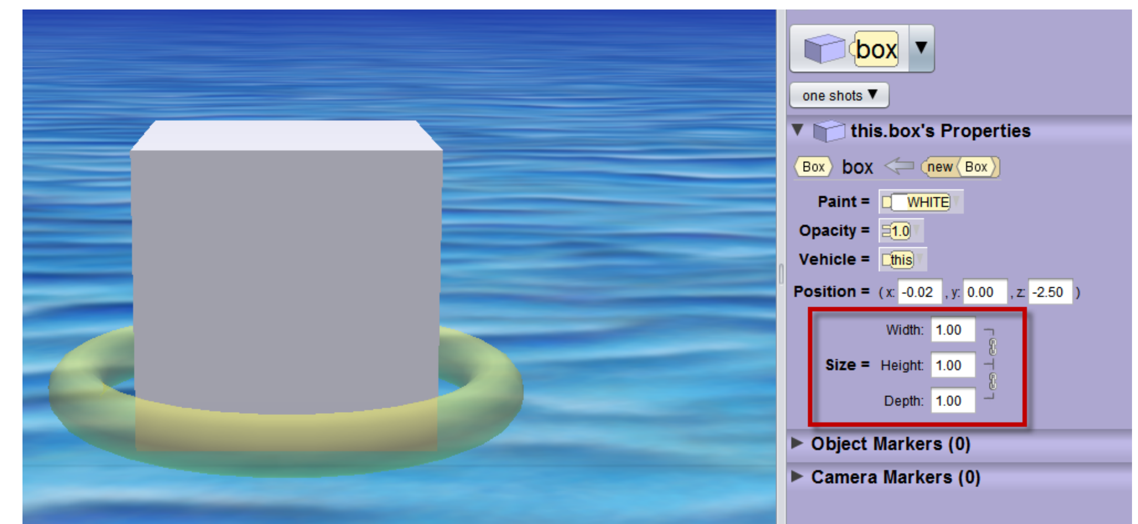


Figure 8 Dimensions of a new box object

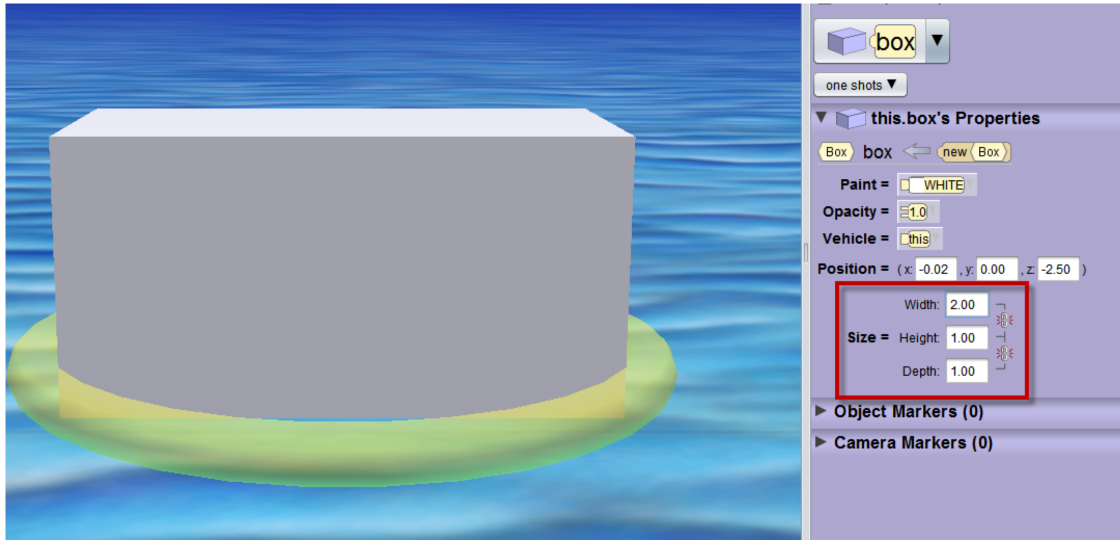


Figure 9 Resized in one dimension, only

SECTION 4

∏

How to set atmospheric properties in a scene

The purpose of this section is to illustrate how to modify a scene using the scene's atmospheric properties: atmosphere color, lighting, and fog.

EXAMPLE

To illustrate , light and fog property changes, we created a world with a brown ogre (of the Ogre class) in a green grass, blue sky scene, as shown in Figure 1. The rock and hedge objects are from the Props collection in the Gallery. We selected this scene in the Object tree, as shown in Figure 2.

Figure 1 Example scene to illustrate special effects

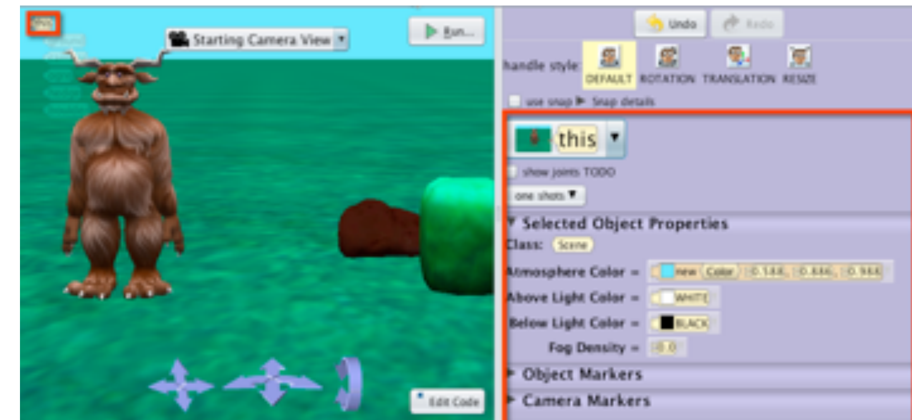


Figure 2 Select 'this' scene in the Object tree

The default settings of Atmosphere, Above Light Color, Below Light Color, and Fog Density in a green-grass, blue-sky template world are shown in the Setup Panel in Figure 3. Note that the Atmosphere color sets the color of the sky, above and below colors provide for lighting effects, and fog density simulates mist in the air.

NOTE: Examples of resetting these values are shown below in a progressive manner below. That is, each change carries over to the next so that the final result is cumulative.

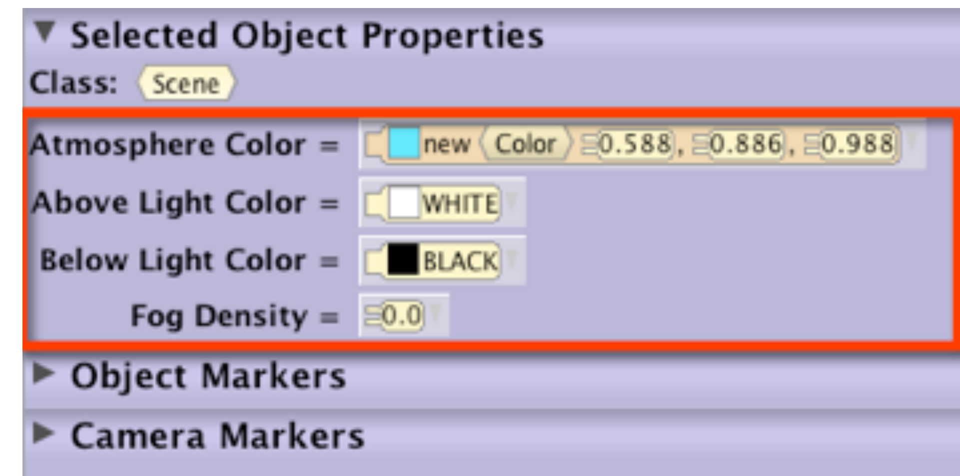


Figure 3 Default settings for lighting and fog in this scene

Set atmosphere color

To set the sky to a different color, click the button for Atmosphere Color and select a color from the drop-down menu, as shown in Figure 4. We selected dark blue, and the result is shown in the screen capture on the left of Figure 4.

The pull-down menu for color also provides a Custom Color option. When Custom Color is selected, a dialog box is displayed where you can select from a grid of color swatches or use HSB or RGB color codes, as shown in Figure 5.

Figure 4 Set the color of the sky (atmosphere)to a bright blue

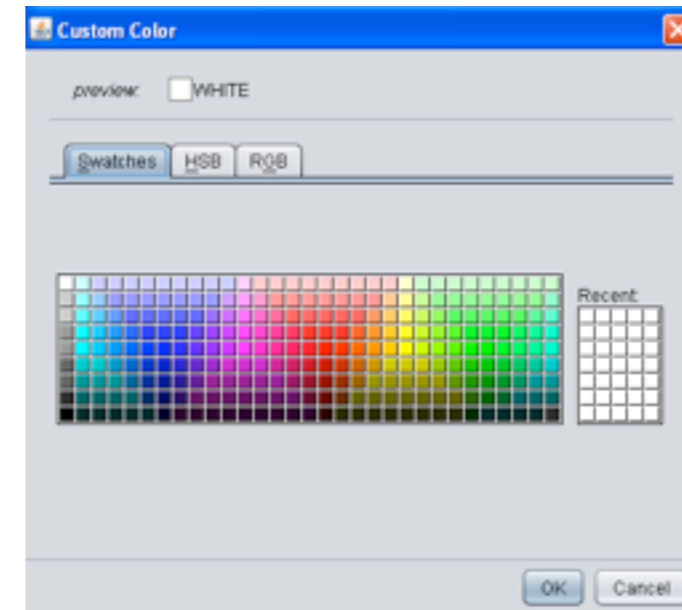


Figure 5 Custom color options

Set Above Light Color

The lighting in a scene is projected from above the scene. By default, the lighting above the scene has a setting of WHITE, which looks like a clear day when the sun is shining. To change the lighting to other settings, click the Above Light Color button and select a color, as shown in Figure 6. In this example, we selected GRAY and the result (similar to a cloudy day) is shown in the screen capture at the left.

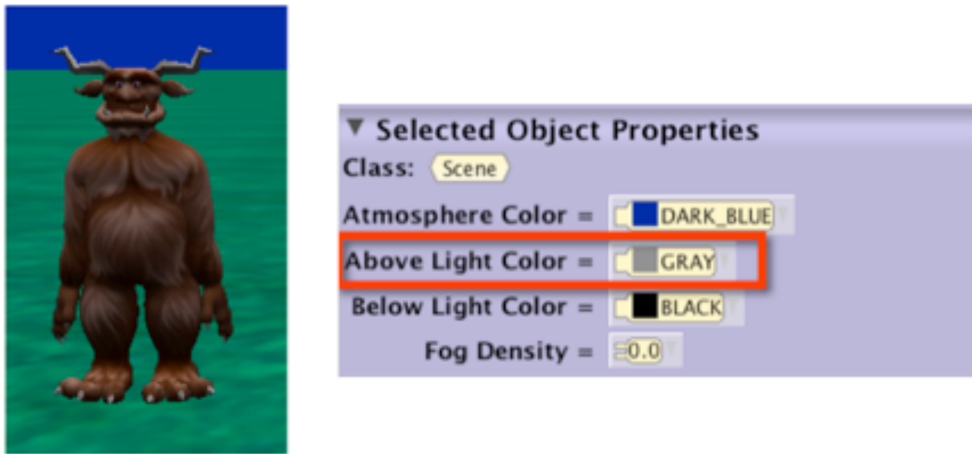


Figure 6 Above Light Color is set to GRAY

Set Below Light Color

In addition to light being projected from above the scene, it is also possible to project light from below. By default, the Below Light Color is set to BLACK, which is the equivalent of no lighting from below. To turn on lighting from below, click the Below Light Color button and select a color, as shown in Figure 7. In this example, we selected RED for a fiery effect and the result is shown in the screen capture at the left.

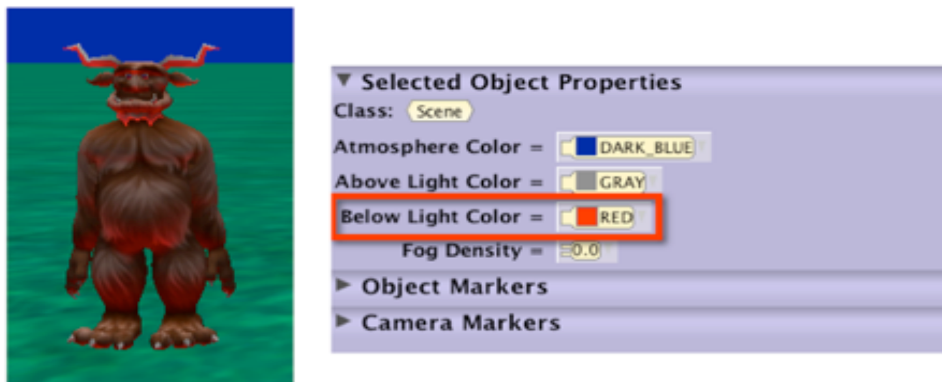


Figure 7 Below Light Color set to RED

Set Fog Density

Fog is used to create a misty effect in the scene, as shown in Figure 8. Fog can be used to allow objects to move into a scene from the back, gradually becoming more and more visible. By default, the fog density is set at 0.0, meaning that there is no fog. A fog density setting of 1 is the most fog that is possible for the scene (only the atmosphere is visible). To set the fog, click the Fog Density button and select a density value from the pull-down menu of values (in the range of 0.0 – 1.0). In this example, we selected 0.3 (approximately 30%) for a mild fog effect and the result is shown at the left in Figure 8.

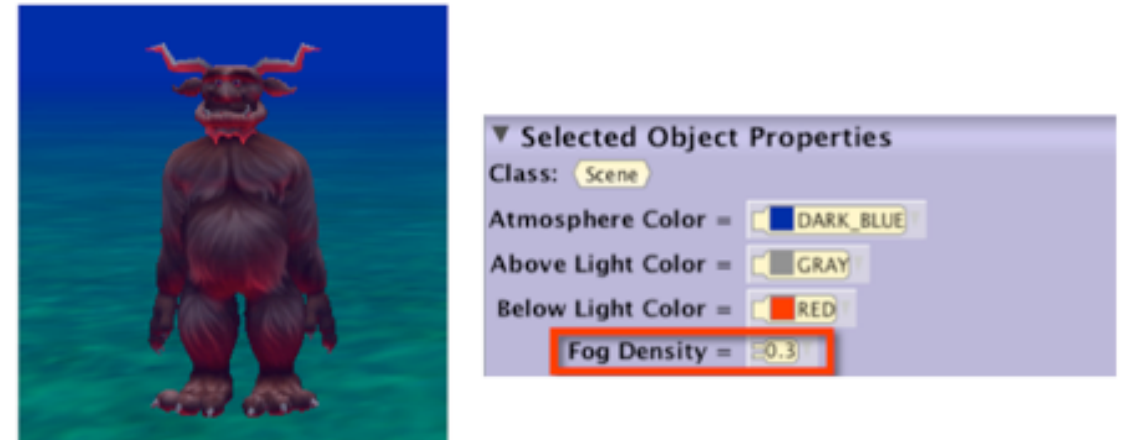


Figure 8 Fog Density set at 0.3

SECTION 5

Marking & changing the camera's position

The purpose of this section is to illustrate how to change the camera's position in a scene using marked viewpoints. A viewpoint is the camera's position and orientation in a virtual world. Alice also allows you to create your own viewpoints with the use of markers.

Note: Alice has five pre-set camera viewpoints, as described in Section 13 of this How-To guide.

MARKING THE CAMERA'S POSITION

Alice has only one camera in a scene. The camera is moved around and repositioned for different viewpoints. This is similar to the use of a camera in a Hollywood studio, where a single camera film-style production technique is often followed. Each scene and camera angle is setup and rehearsed until the director is happy with the arrangements. The camera viewpoints (location and orientation) are marked before any actual filming begins.

In Alice, a camera marker is an object that remembers the position and orientation of the camera at the time the marker was created. The camera can then be moved or rotated to a different location and orientation, but the marker stays where it was originally created. A marker object is visible in the Scene editor. There is no need to worry, however, about camera markers cluttering up a scene at runtime. When the user clicks the Run button to play an animation, markers are not visible in the scene. (The markers are still remembered but are just not made visible to the viewer of the animation.)

MARKERS SECTION OF THE SETUP PANEL

The Setup panel in the Scene editor has a section for markers, located immediately beneath the list of properties for an object, as shown in Figure 0. To view the Camera Markers section, click on the arrow next to the Camera Markers label at the bottom of the panel, as shown in Figure 1. The Camera Markers section should expand to show buttons for creating camera markers. Notice that the Camera Markers section has three buttons -- two small buttons having a dark gray camera icon and a question symbol and one button labeled Add Camera Marker ….

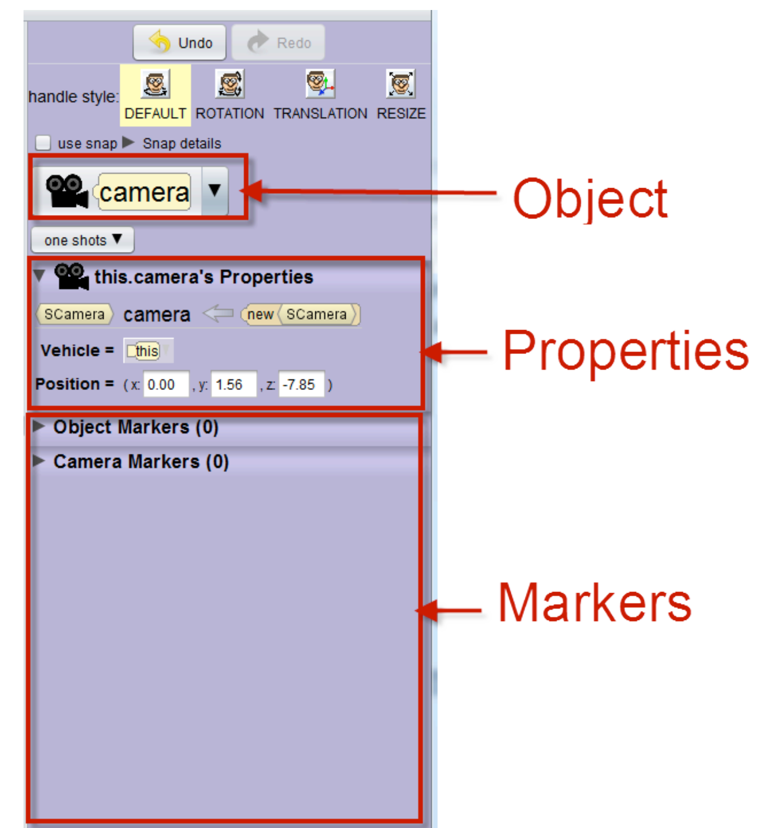


Figure 0 Setup panel in the Scene editor

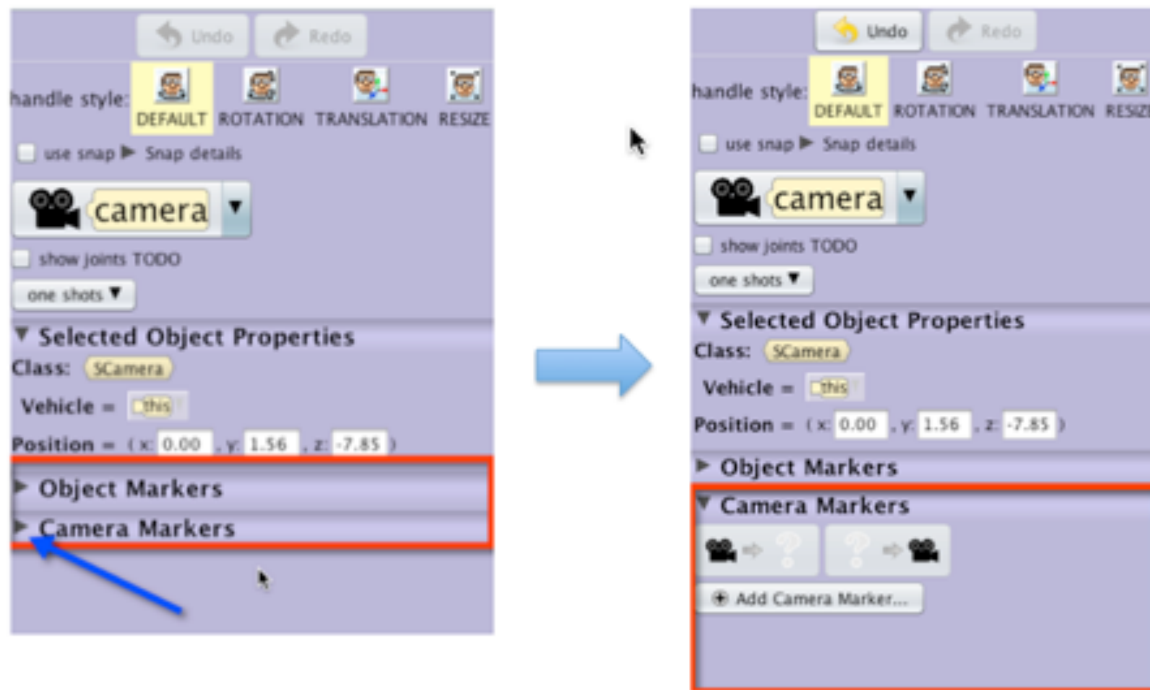


Figure 1 Collapsed (left) and Expanded (right) Camera Marker section

CREATING A CAMERA MARKER AT THE STARTING POSITION

We recommend marking the starting location of the camera before moving the camera around in the scene. The camera can then be moved freely around the scene and can always be returned to its original position, using the marker.

To create a starting location camera marker click on the Add Camera Marker… button, as shown in Figure 2. A dialog box will pop up, as shown in Figure 3. Enter a meaningful name for the marker, for example startPosition. When a name is entered, press the Enter key and Alice will automatically create a camera marker object at the current location of the camera. The marker remembers not only the location but also the camera's orientation (the direction and angle at which it is pointed). This information is commonly known as the camera's viewpoint.

If more than one camera marker is created, each successive marker is automatically assigned a different color (red, green, blue, etc.). As shown in Figure 4, we created two camera markers. One is red (startPosition) and the other is green (overheadPosition). In addition, the name of each camera marker is displayed in a matching color in the Markers section of the Setup panel.

Note: We pulled the camera way back in this scene in order to obtain a screenshot showing both markers for Figure 4. So, it may not look exactly the same on your monitor if you are following along with the instructions given here.

Figure 2 Click Add Camera Marker …

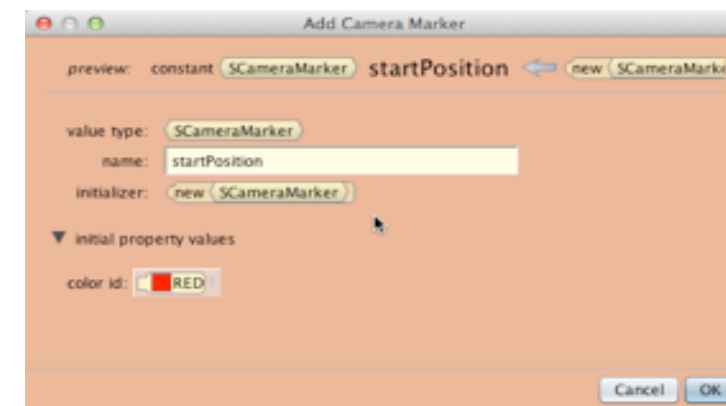


Figure 3 Enter a meaningful name for the camera marker

Figure 4 Different markers have different colors

Moving and turning the camera using navigation controls

At the bottom edge of the scene view are three sets of camera navigation controls, as shown in Figure 5. The most common use of navigation controls is to set the camera's initial point of view for best effect in animation.

A click-and-hold on an arrow will move or turn the camera as implied by the arrow-icon. Clicking and dragging in the direction of the arrow will speed up move or turn action. You can also click and drag in a direction between two arrows, which will combine the actions of the two navigation arrows. The actions of specific navigation arrows are described below.

Figure 5 Three sets of camera navigation controls

Move camera up / down / left / right

The set of four arrows on the left, as shown in Figure 6, move the camera up or down (vertically), or left or right (horizontally), from the camera's point of view. As with any move action in Alice, these arrows change the location of the camera in the scene, but not its orientation (the direction the camera is facing). Professional videographers refer to these actions as the camera being ped (up and down) or tracked (side to side).

Figure 6 Move the camera up, down, left, or right

Move camera forward / backward

For purpose of clarity, the set of four arrows in the center are described here in two subsets. The two arrows pointing forward and backward (horizontally), as outlined in yellow in Figure 7, move the camera forward or backward (as seen by the camera). We refer to this action as the camera is zooming in or out relative to an object in a scene. Although professional videographers often use the term 'zoom' for changing the focal length of the camera's lens to give the illusion of moving the camera, in Alice the camera is actually moved (no lens change occurs).

Figure 7 Move the camera forward or backward

Turn camera left / right

The other two arrows in the center set, as outlined in yellow in Figure 8, turn the camera to the left or right, as seen by the camera. As with any turn action in Alice, a turn changes the orientation of the camera in the scene, but not the location of the camera. Professional videographers refer to this action as panning the scene.

Figure 8 Turn the camera left or right

TURN THE CAMERA FORWARD / BACKWARD

The set of arrows on the right, as outlined in red in Figure 9, turn the camera forward or backward (a tilting action) in the scene. As with any turn action in Alice, a turn changes the orientation of the camera in the scene, but not the location of the camera. Professional videographers refer to this action as tilting.

Figure 9 Turn the camera forward or backward

HOW TO POSITION THE CAMERA AT A MARKER

Let's assume that we have used the camera navigation controls to move the camera around the scene and it is no longer at the starting position. Now, we can take advantage of the camera markers we created earlier.

First, select (from the list of camera markers) a marker to which the camera will be moved. In Figure 10, the overheadPosition marker has been selected. Notice that the question symbols in the two camera marker buttons have been replaced with green camera icons because the overheadPosition camera marker in this example is green.

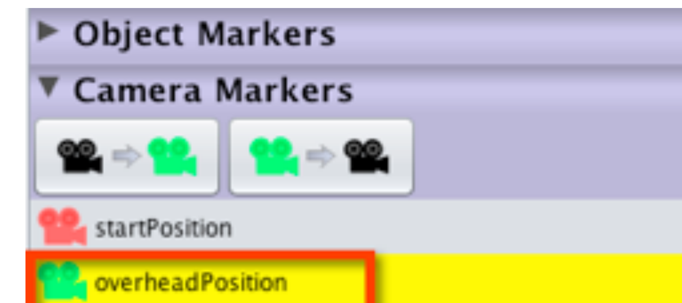


Figure 10 Step 1: Select the targeted marker

Secondly, click the camera => marker button (left of the two small buttons), as shown in Figure 11 to move the camera to the selected marker. The camera will immediately move and orient to the targeted marker.

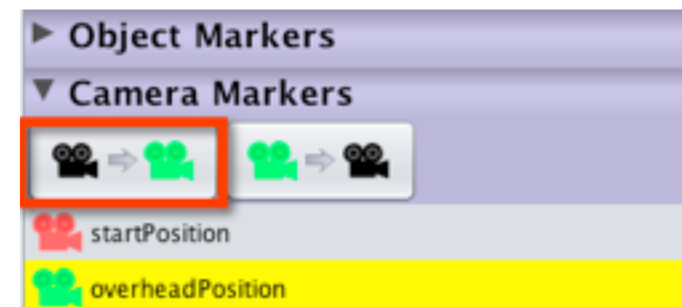


Figure 11 Step 2: Click camera => marker button (left)

REPOSITIONING A CAMERA MARKER

Once in a while, a marker may have been created in the wrong place. Rather than deleting the marker and creating a new one, the existing marker can be repositioned. To reposition a marker, first position the camera in the desired new location and orientation. Then, select the marker to be repositioned in the list of camera markers. In the example shown in Figure 20.12, we selected startPosition (a red camera marker).

Notice that the two small buttons now show the dark camera icon (current camera position) and a red camera icon (the selected marker).

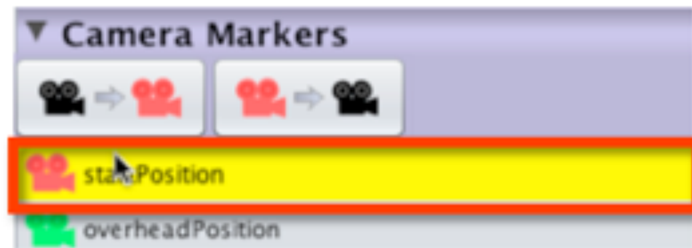


Figure 12 Select the marker to be repositioned

Now, click on the marker => camera button (outlined on the right in Figure 13). Alice repositions the selected marker to the current camera position.

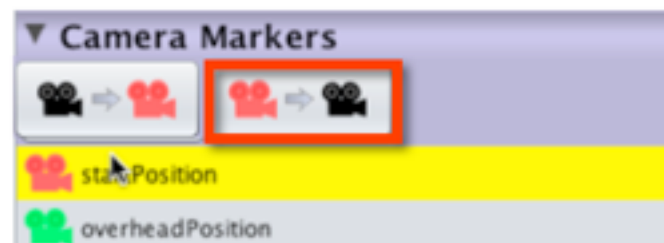


Figure 13 Click the marker => camera button (right)

SECTION 6

∏

Positioning objects with markers

In the previous section of this guide, we introduced camera markers. . The purpose of this section is to introduce object markers that remember the position and orientation of other kinds of objects.

MARKERS

To better understand object markers, consider an analogy: a bookmark in a web browser (e.g., Firefox, Safari, IE, Chrome, or some other). To make it easy to find a favorite web site, a bookmark is created. Later, to return to that favorite web site, the bookmark in the browser is used to return to that website on the Internet. Object markers in Alice do a similar kind of thing. Markers are used to remember the position and orientation of an object at the time the marker was created. Then, later, after the object has moved or rotated to a different position, the object can be repositioned at the marker.

EXAMPLE

To illustrate object markers in this section, we have created a scene with the alien and an asteroid boulder in the Mars scene, as shown in Figure 1.

Figure 1 Example scene

OPEN MARKERS IN SETUP

To open the Object Markers section of Setup in the Scene editor, click on the arrow next to the label Object Markers at the bottom of the panel, as illustrated in Figure 2 (left). The Object Markers section should expand to show buttons for creating object markers, as illustrated in Figure 2 (right).

Figure 2 Collapsed (left) and Expanded (right) Object Marker section in Setup

Create an object marker

To create an object marker, first position the object in the desired location and orientation in the scene. Next, click on the Add Object Marker … button. In the example shown in Figure 3, the object is the alien.

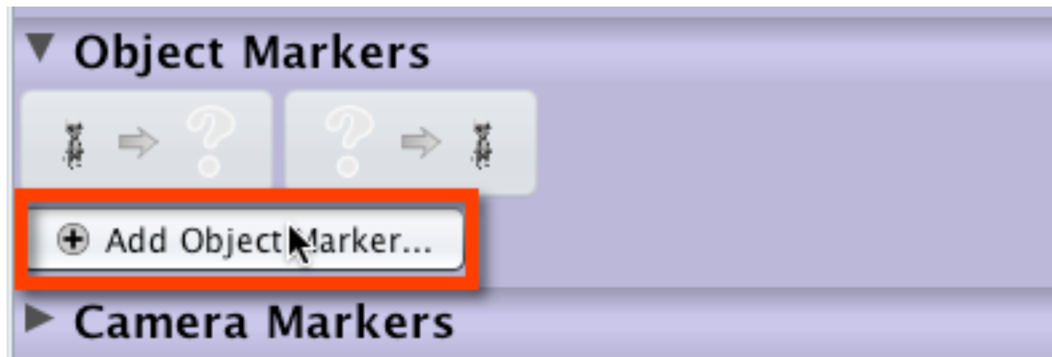


Figure 3 Add Object Marker …

A pop-up dialog box provides an opportunity to give the marker a meaningful name, such as firstPosition as shown in Figure 4.

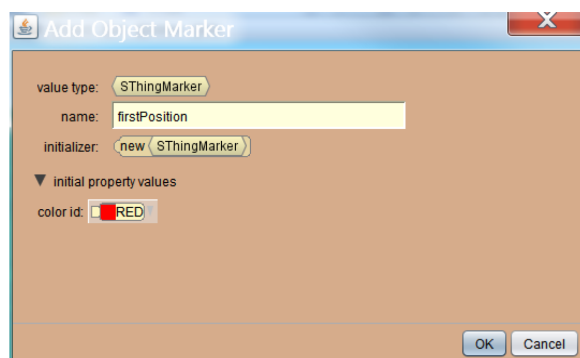


Figure 4 Enter a meaningful name for the object marker

When the name is entered, press the Enter key. Alice creates a set of axes to represent the object marker. The axes marker is automatically positioned at the pivot point of the object, as shown in Figure 5. The object marker automatically has the same orientation as the object.

Figure 5 An axes object represents an object marker

Moving an object to a marker in the Scene editor

In this example, we created a second marker at the top of the asteroid, as shown in Figure 6. To move an object from its current position to a marked position, first select (from the list of object markers) the marker to which the object will move. In Figure 6, the topOfAsteroid marker has been selected. Notice that the question symbols in the two object marker buttons have been replaced with an object (in this example, an alien) and an axes icons.

Figure 6 Step 1: Select the targeted marker

Secondly, click the object => marker button (left of the two small buttons), as shown in Figure 7, to move the object to the selected marker.

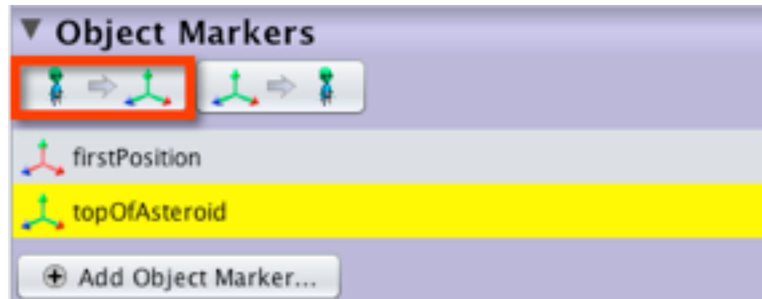


Figure 7 Step 2: Click object => marker button (left)

The object will immediately move and orient to the targeted marker. In this example, the alien moved to the top of the asteroid, as shown in Figure 8.

Figure 8 Result of moving an object to an object marker

NOTE: The Undo button can be used to reverse an action with a marker, if necessary.

REPOSITIONING AN OBJECT MARKER

To reposition an object marker from its current position to the current location of an object, first select (from the list of object markers) the marker to be repositioned. In Figure 9, the firstPosition marker in the list has been selected. Then, click on the marker => object button to move the marker to the selected object, as shown in Figure 10.

Figure 9 Select marker to be repositioned

Figure 10 Click marker => object button (right)

SECTION 7

Positioning sub-parts in Scene editor

The purpose of this section is to illustrate how to position sub-parts of an object while setting up a scene in the Scene editor. In Alice 3, the 3D model classes define objects having an internal skeletal system consisting of joints.

Sub-parts (for example an object's head, arms, legs, tail, and other parts) are connected to one another and to the body by these joints. Therefore, a sub-part of an object is positioned by rotating the joints of the skeletal system.

HOW TO VIEW THE SKELETAL JOINTS

In the real world, joints in an entity's skeletal system are usually hidden within the body. For example, a human has shoulder joints and elbow joints but these skeletal joints are enclosed within the body's skin and muscular tissue. The joints can only be seen by taking an X-ray or by some other medical procedure.

Similarly, Alice object joints can only be seen by using an X-ray-like view. To view the joint positions of an object, select an object in the Object tree and then check the box for the Show Joints option in the Setup. Next, reduce the object's opacity property to a low value such as 0.5. Figure 1 illustrates an example X-ray-like view of the skeletal system.

In Figure 1.21, notice that the location of each joint is marked with a small axes object. The axes object is for the purpose of showing the location and orientation of each joint. White is forward, red is right, and green is up (as seen by the joint at that position).

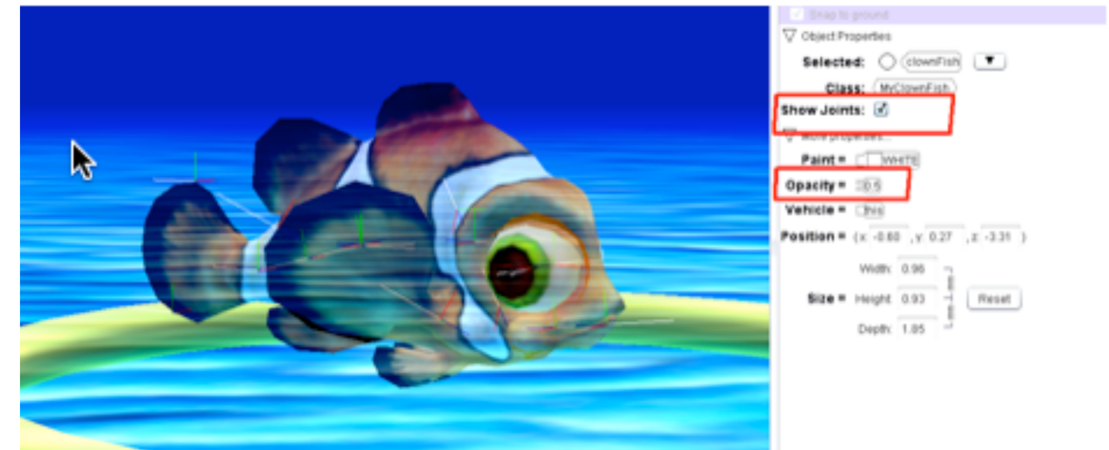


Figure 1 An X-ray-like view of the skeletal joint system

For each skeletal joint, its orientation is usually consistent with the functioning of an attached sub-part. For example, Figure 2 shows a close-up view of the fish's right eye. The important thing to understand is that this fish's eyes face outward (to the side of the fish). The eye's joint axes object has a white axis pointing in the direction the eye is "facing" (which is forward as seen by the eye), the green axis is the upright position of the eye within the fish's body (up), and the red axis is to the right of the eye (as seen by the eye).

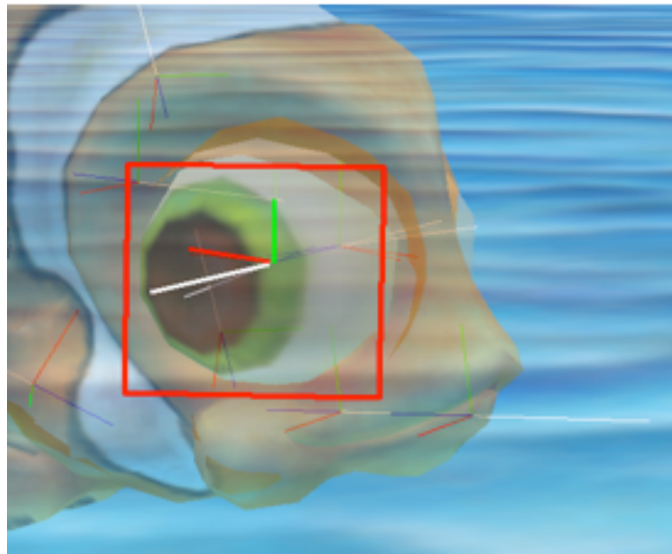


Figure 2 Orientation is consistent with the action of an attached sub-part

Some skeletal joints are located in an extended limb (for example, an arm, leg, wing, fin, or flipper). A limb often contains numerous joints that must share the same orientation. For example, Figure 3 shows a close-up view of the fish's tail. The tail is a limb that is "facing" outward (similar to the fish's eye). The tail is one sub-part but has three joints to provide some flexibility for animation. The three joints share the same orientation, as seen in the axes at each joint. The white axis of each joint is facing outward (forward for the tail), the green axis is the upright position of the tail sub-part as attached to the fish's body (up), and the red axis is the right of the tail, as seen by the tail.

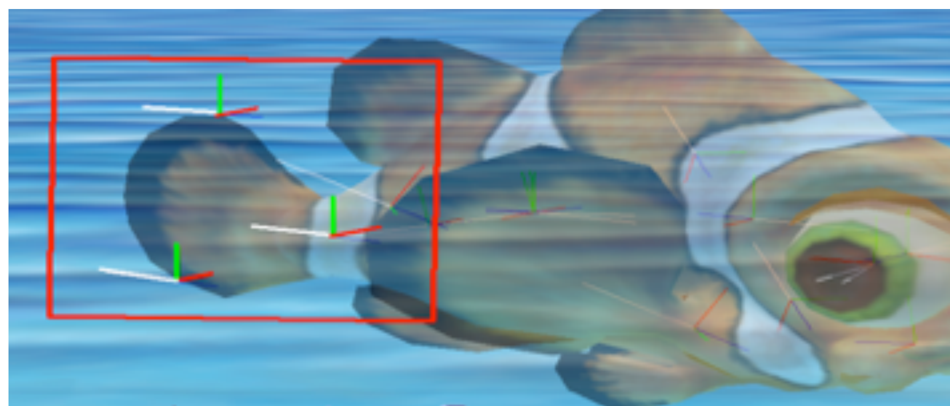


Figure 3 Multiple joints in a limb have consistent orientation, facing away from the body

HOW TO SELECT A SKELETAL JOINT

Sub-parts of an object can be positioned by selecting the appropriate joint from the Object Parts menu and then rotating the joint. To view the Object Parts menu, click the selected tile in the Setup panel, select the object in the list of objects and pull the mouse cursor over the right arrow to open a cascading menu of joints, as shown in Figure 4. In this example, the tail was selected for a clownFish object.

Figure 4 Selecting a joint/part of an object

When a joint is selected, Alice automatically displays three rotation ring handles around the selected joint. In this example, the rings are displayed with the fish's tail joint as the pivot point of the tail, as shown in Figure 5.

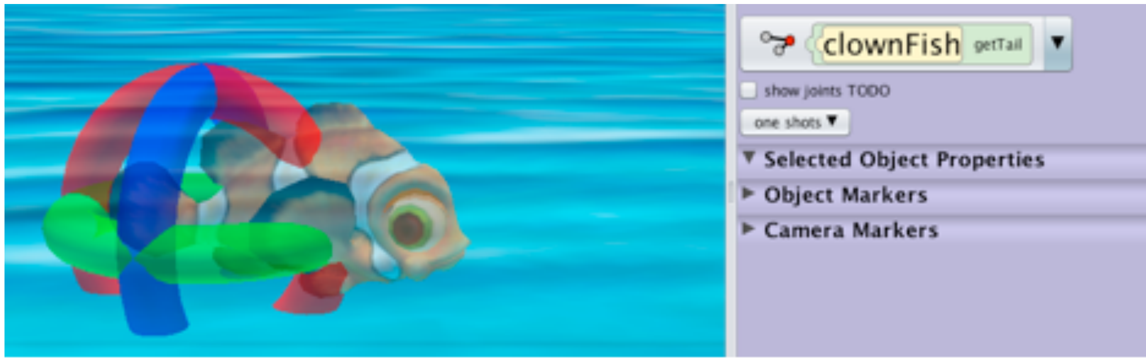


Figure 5 Three rings for rotating the tail joint

Now the rings can be used to rotate the tail into the desired location, as shown in Figure 6. The same process can be used to position other joints (and associated sub-parts) in the object.

Figure 6 Using a ring handle to rotate the tail

SECTION 8

∏

Relative positioning with camera viewpoints

The purpose of this section is to demonstrate how to position two or more objects at locations relative to one another in a scene. Alice provides five pre-set multiple camera viewpoints for relative positioning.

Video: Using Camera Views

EXAMPLE

To illustrate, we added a hare (harry), a Cheshire cat (chessy), a tiger (tiggerrr), a tea tray, and a teapot to the example scene shown in Figure 1. In this example, the goal is to put the teapot on the center of the tea tray.

Figure 1 Tea tray and teapot, in original positions

CAMERA VIEWPOINTS

Positioning the teapot on the center of the tray looks simple. Just drag the teapot onto the center of the tray, as shown in Figure 2. However, the actual position of one object relative to another object can be deceptive because our view of the scene is only what we see through the camera's lens (the camera viewpoint). In this example, the camera viewpoint is from the front of the scene and it is difficult to see whether the teapot is actually at the center of the tray.

Figure 2 Teapot is on the tray, but is it in the center?

A Camera viewpoints menu is located at the top center of the scene view. To open the Camera viewpoints menu, click on the down-arrow at the right edge of the button. The menu should drop down to show a list of pre-set camera viewpoints, as illustrated in Figure 3.

Figure 3 Camera viewpoint pull-down menu

When an item in the menu is selected, Alice automatically takes care of positioning the camera at the selected viewpoint. The Layout Scene View positions the camera upward and at an angle, as shown in Figure 4. From this viewpoint, it is easy to see that the teapot is not quite on the center of the tea tray.

Figure 4 Layout Scene View

As shown in Figure 5, all the handle style tools and the camera navigation controls are available in this view and can be used to reposition objects in the scene.

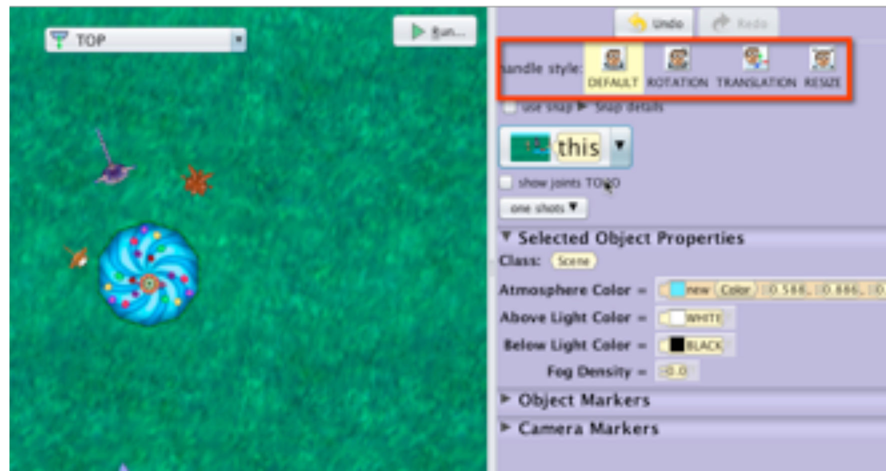


Figure 5 Camera navigation tools and handles can be used to reposition objects

Figure 6 shows the result of using the mouse to carefully position the teapot on the center of the tray. Use the Camera Viewpoints menu to put the camera back to the Starting viewpoint.

Figure 6 The teapot is now on top and at center of the tea tray

Other Camera viewpoints

In the example above, the Layout Scene view is all that was needed. However, the camera viewpoints menu offers other options:

TOP view

The TOP view presents an overhead view of a scene, as shown in Figure 7. The camera is hovering over the scene and is pointing straight toward the ground in the scene.

Figure 7 TOP view

In the TOP view, the camera navigation arrows are limited to those used for moving the camera (forward, backward, left, right, up and down), as shown in Figure 8. The four navigation arrows to the left in Figure 8 allow the camera to be moved left, right, up, and down, as seen by the camera. The two arrows on the right move the camera forward and backward, as seen by the camera. The SIDE and FRONT views, as described below, also have this limitation on available camera navigation arrows.

Figure 8 Camera navigation arrows in TOP, SIDE, and FRONT views

Note: The motion of the camera that results from using camera navigation arrows is always "as seen by the camera." This can be surprising when the camera is in TOP, SIDE, or FRONT view. For example, in TOP view, the camera's "forward" orientation is looking straight down toward the ground. So, moving the camera forward in TOP view actually zooms in closer to the ground.

SIDE view

The SIDE view presents a camera viewpoint that faces the center point of the ground, from the ground's right side, as shown in Figure All the handles are available. The camera navigation arrows allow moving the camera forward, backward, left, and right, up and down. But it is not possible to turn the camera to the left or right, forward or backward.

Figure 9 SIDE view

FRONT view

In the FRONT view the camera viewpoint faces the center point of the ground, as shown in Figure 10. The camera navigation arrows allow moving the camera forward, backward, left, and right, up and down. But it is not possible to turn the camera.

Figure 10 FRONT view

The black shape (highlighted in a red box) in Figure 10 above, is a starting camera marker (used internally by Alice to remember the starting position for the camera). The camera marker can often be seen in the other camera viewpoints as well, and the camera marker can be moved, turned, rolled and oriented in the same way as any other Alice object in a scene. Remember, however, that changing Alice's internal camera marker will change the Starting Camera View.

Note: It must be emphasized that the Camera viewpoints menu is only available in the Scene editor, for convenience in setting up a scene. The Camera viewpoints listed in the viewpoints menu are

not available in the Code editor and cannot be used for creating program code.

SECTION 9

∏

How to align objects using a Snap grid

The purpose of this section is to demonstrate how to align two or more objects. Alice provides a grid and one shot methods for alignment and positioning.

Figure 1 Current state of example scene

EXAMPLE

To illustrate, we will continue with the scene created in the previous section where the teapot was positioned on the center of the tea tray. The current state of this example scene is shown in Figure 1. In this continuing example, the goal is to position tiggerrr, chessy, and harry all in a straight line behind the tray.

To align objects in a scene, activate the Snap grid in the Setup Panel, as shown in Figure 2. The Snap grid option displays a grid on the ground or water surface in a scene. By default, the grid is set to display grid blocks that are 0.5 meters on a side. In addition, using the mouse to drag-and-drop an object will cause the object to snap into position at the nearest grid point. Rotating an object will cause the object to snap into position at the nearest 30 degree angle. The grid and angle snap values may be set to other values.

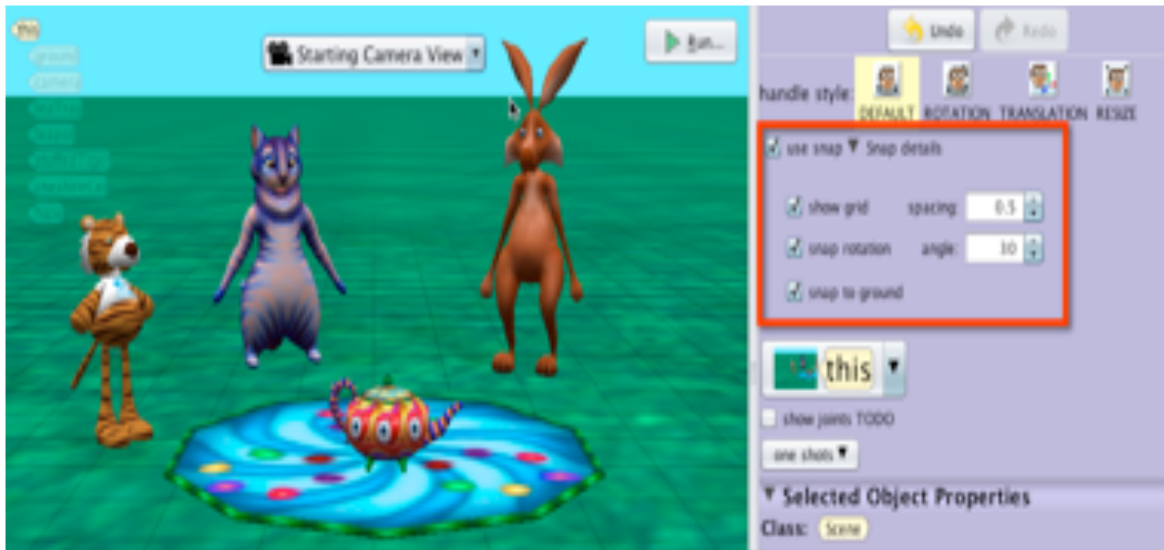


Figure 2 Grid is displayed and snap is active

To use the grid for positioning an object, click and drag the object with the mouse. Alice automatically creates extended, highlighted grid lines for the clicked object, as shown in Figure 3.

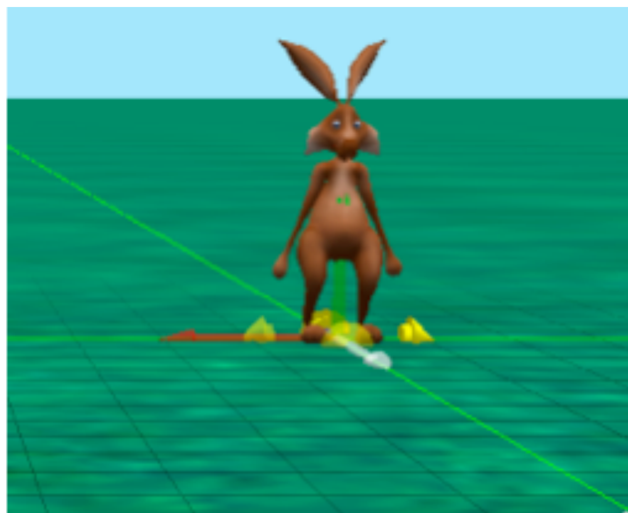


Figure 3 Highlighted grid lines for the selected object

To align the three objects along one line of the grid, click and drag each object so as to snap to a grid point along the same line, as shown in Figure 4.

Figure 4 Using Snap grid lines for alignment

SECTION 10

∏

Precise positioning with one-shots

The purpose of this section is to illustrate how to use a one-shots menu for alignment and precise positioning of objects and object sub-parts in the Scene editor.

[Video: Using One-Shot Procedures]

EXAMPLE

To illustrate precise positioning of objects and sub-parts, we will use the scene shown in Figure 1. In this example scene, the alien is on a moon surface with his pet robot.

Figure 1 An alien and his pet robot

ONE-SHOT PROCEDURES

Procedures are methods that perform an action. One-shot procedures are listed in a drop-down menu in the Scene editor. A one-shot procedure is an action performed "right now" and only once (a "one-shot") by an object in the scene. There are three techniques for opening a one-shot menu. One technique is to right-click on the name of an object in the Object tree, as shown in F in Figure 2, and then select the word "procedures" from the drop-down menu. ocedures.

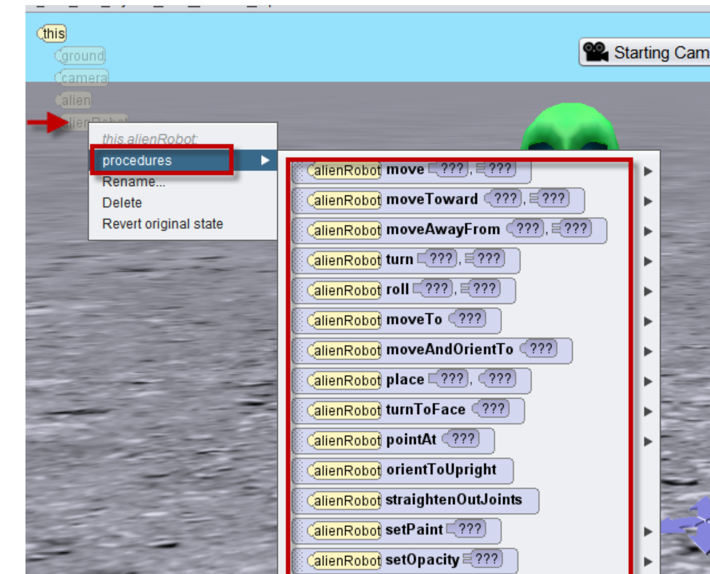


Figure 2 Opening one-shots menu by a right-click in the Objects tree

A second technique is to right-click on the object itself, as shown in Figure 3.

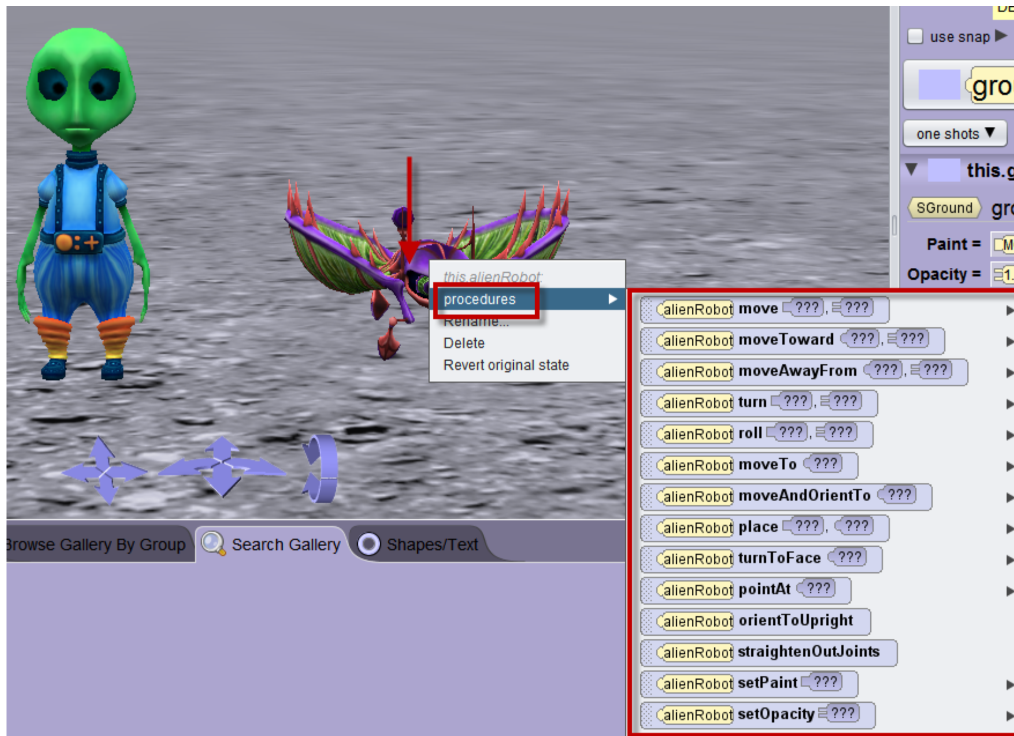


Figure 3 Opening the one-shots menu by right-clicking on the object

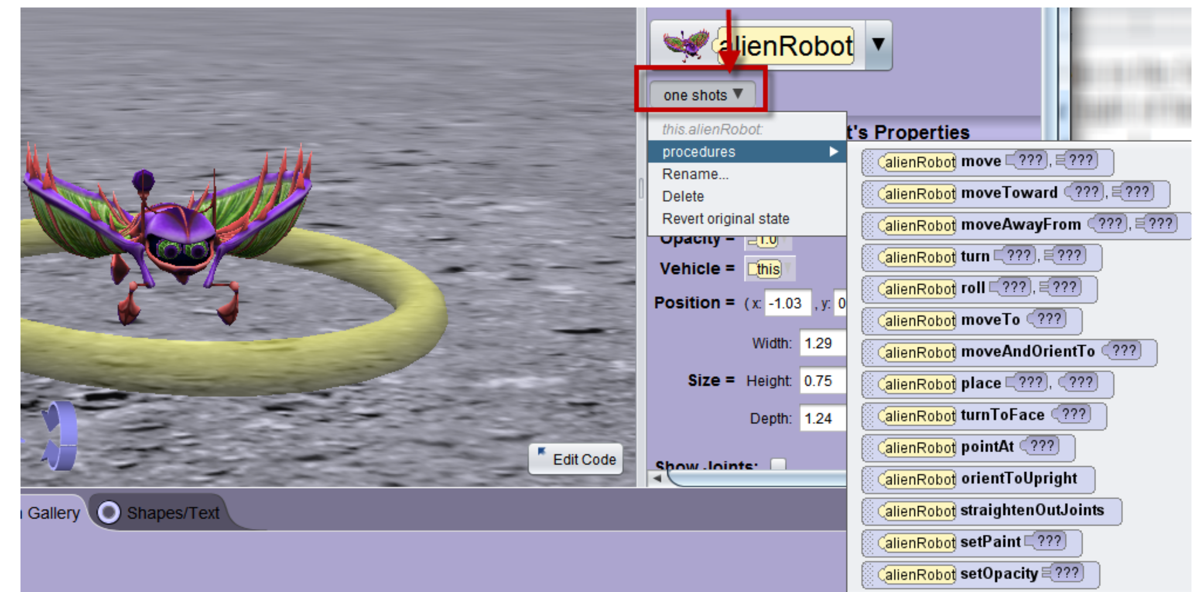


Figure 4 Opening the one-shots menu by clicking a button in Setup

The third technique is to left-click on the pull-down menu button in the Setup, as shown in Figure 4. Notice that the pull-down menu cascades beyond the right of the Alice 3 window (the monitor's wallpaper can be seen in the background). On computer systems where the monitor is not wide enough, the menu will wrap to the left instead.

As an illustration of using one-shots, let's walk through the steps of precisely positioning alien and the alienRobot exactly 2 meters apart. The first step is to position the two objects in the exact same location and orientation. Right click on the alien in the Object tree, select procedures, and then select the moveAndOrientTo tile as shown in Figure 5. In this example, we selected buddy as the target object.

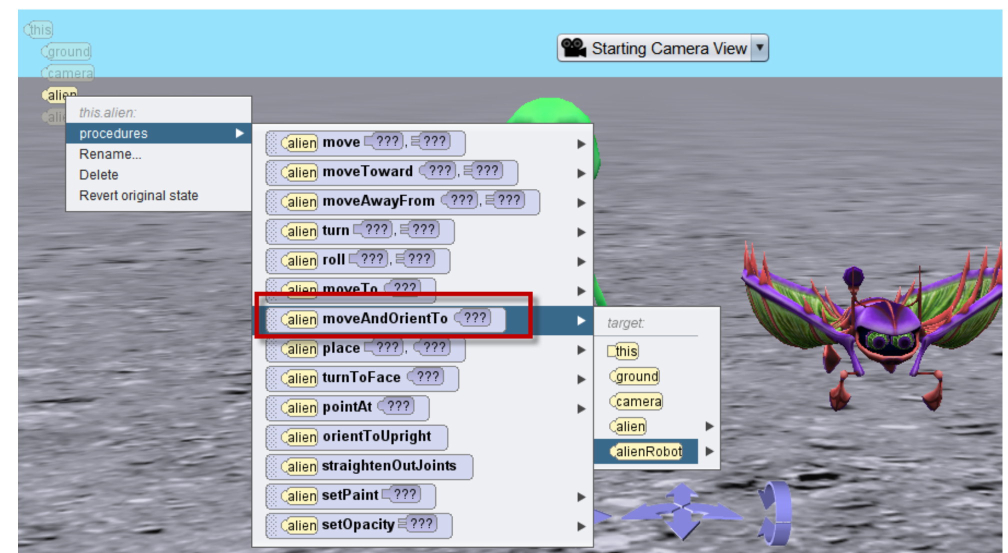


Figure 5 Select moveAndOrientTo

The alien will immediately move to the exact same location and orientation as the robot, as shown in Figure 6.

Figure 6 Two objects in same location and orientation

The second step is to select a one-shot to move the alien 2 meters to its right. In Figure 7, we right clicked on the alien tile in the Object tree, selected procedures, the *alien.move* tile, RIGHT as the direction, and 2.0 meters as the amount.

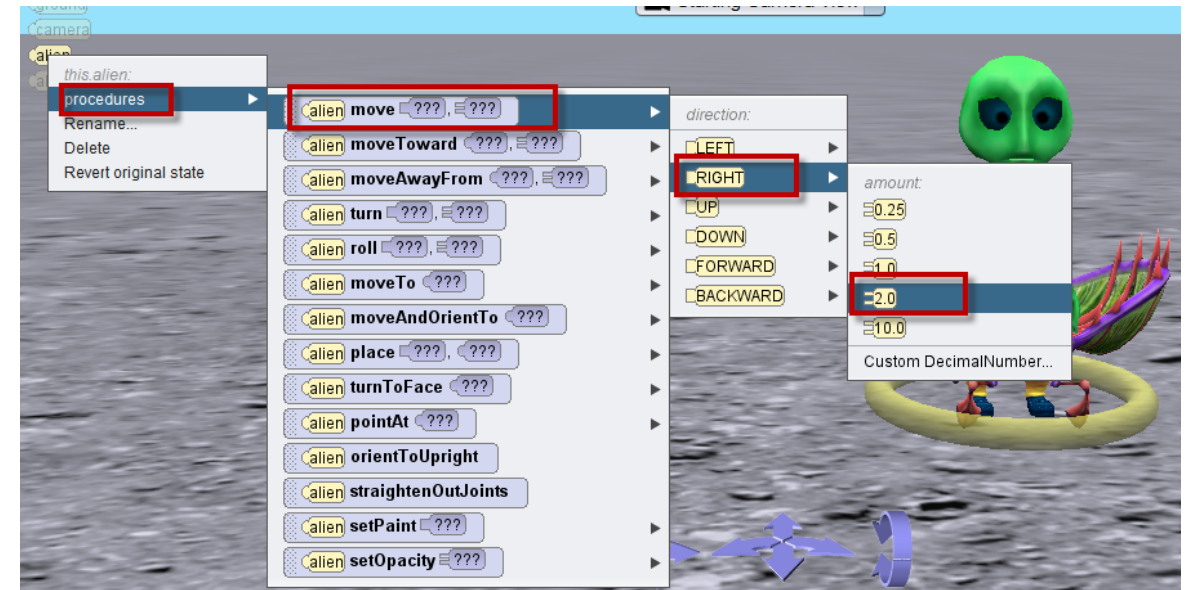


Figure 7 Positioning the alien exactly 2 meters from the robot

As seen in Figure 8, the alien and robot are now precisely 2.0 meters apart. It is important to note that the distance is measured as the shortest distance from the center of one object to the center of the other object. The center of an object is its pivot point as it moves, turns, and rolls in animations.

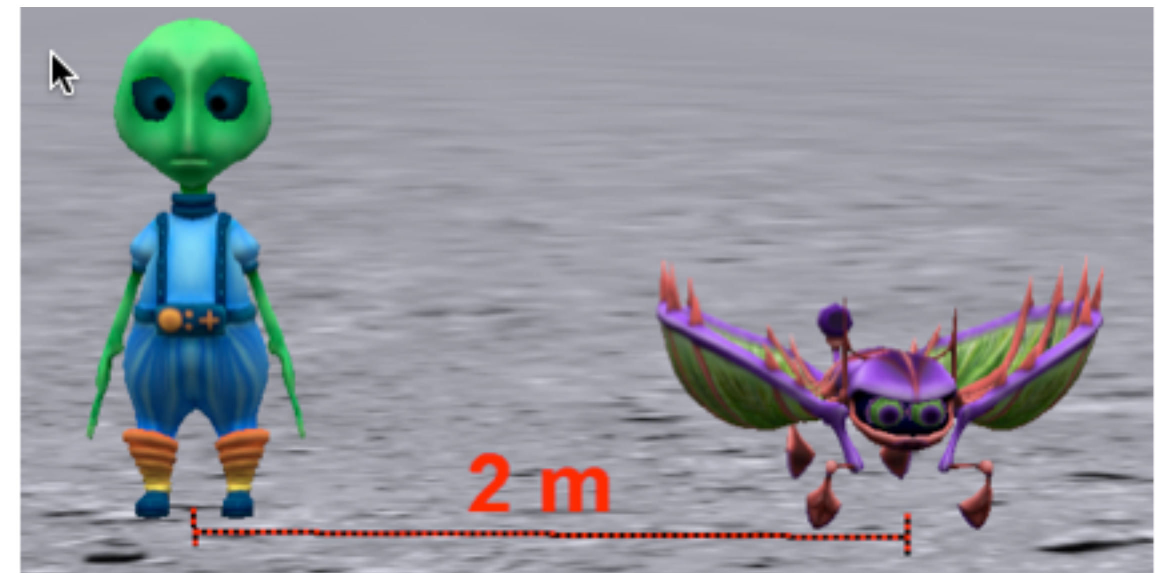


Figure 8 Distance is measured center to center

USING A ONE-SHOT FOR POSITIONING AN INDIVIDUAL JOINT

To illustrate using one-shots for positioning an individual joint, we selected the alien's right shoulder joint in the Setup, as shown in Figure 9.

Figure 9 Select the alien's right shoulder joint

With an object's joint selected, the one-shots menu can be displayed by clicking the down-arrow immediately to the right of the selected joint name, as shown in Figure 10. Note that the one-shots menu for a joint has only five procedures. This is because an object's skeletal joints can be turned and rolled but cannot be "moved."

Figure 10 Opening one-shots menu for a skeletal joint

In this example, we selected a one-shot to turn the alien's right shoulder backward 0.125 revolutions, as shown in Figure 11.

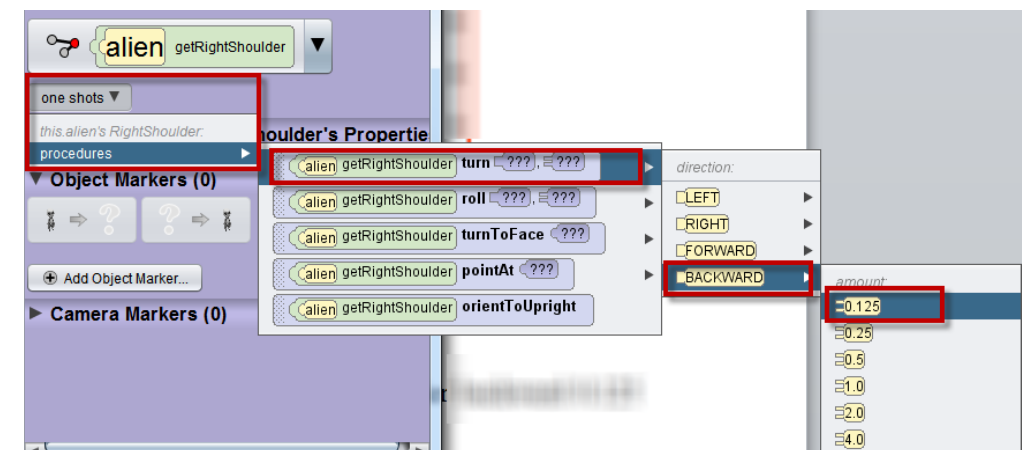


Figure 11 Turn the alien's right shoulder 0.125 revolutions backward

Turning, rolling, and orienting a skeletal joint has an effect on associated subparts of an object. Figure 12 shows the result of turning the right shoulder 0.125 revolutions (45 degrees).

Figure 12 Position of the alien's right arm after turning the right shoulder

CHAPTER 3

Writing Alice Code

The goal of this chapter is to provide information and instructions for creating animations using the Alice 3 Code Editor. This will include descriptions of the object menu, the methods panel, the class menu and the drag and drop editor of Alice 3.

Section 1

∏

Code editor tabs – Scene class

The purpose of this section is to demonstrate the purpose of the default Code editor tabs, which display the Scene class and two of its procedural methods (initializeEventListeners and myFirstMethod).

For illustration, we will use an African themed scene, as shown in Figure 1. The scene includes a variety of props (grasses, rocks, trees, termite mound, and a pond) as well as elephant, ostrich, and baby ostrich objects.

Figure 1 African scene with elephant, ostrich, and baby ostrich

When a project is first opened in Alice, the Code editor automatically displays three default tabs, as shown in Figure 2. The first tab is the Scene class document tab, which is accompanied by two procedural method tabs: initializeEventListeners and myFirstMethod. The Scene's myFirstMethod tab is automatically selected as the active editor tab. Each of these tabs is briefly described below.

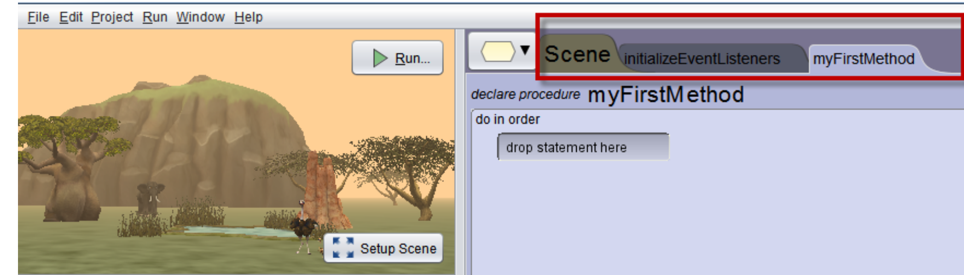


Figure 2 Code editor tabs for the Scene class

Scene class document tab

A **class** is a document file that defines how to create an instance of a specific type of object. The Scene class defines how to create a scene for a virtual world. A class document may also contain definitions for procedures (action methods), functions (computational methods), and properties (fields – objects or items of data, such as color or opacity). Figure 3 shows the Scene class document tab for the example world used in this section.

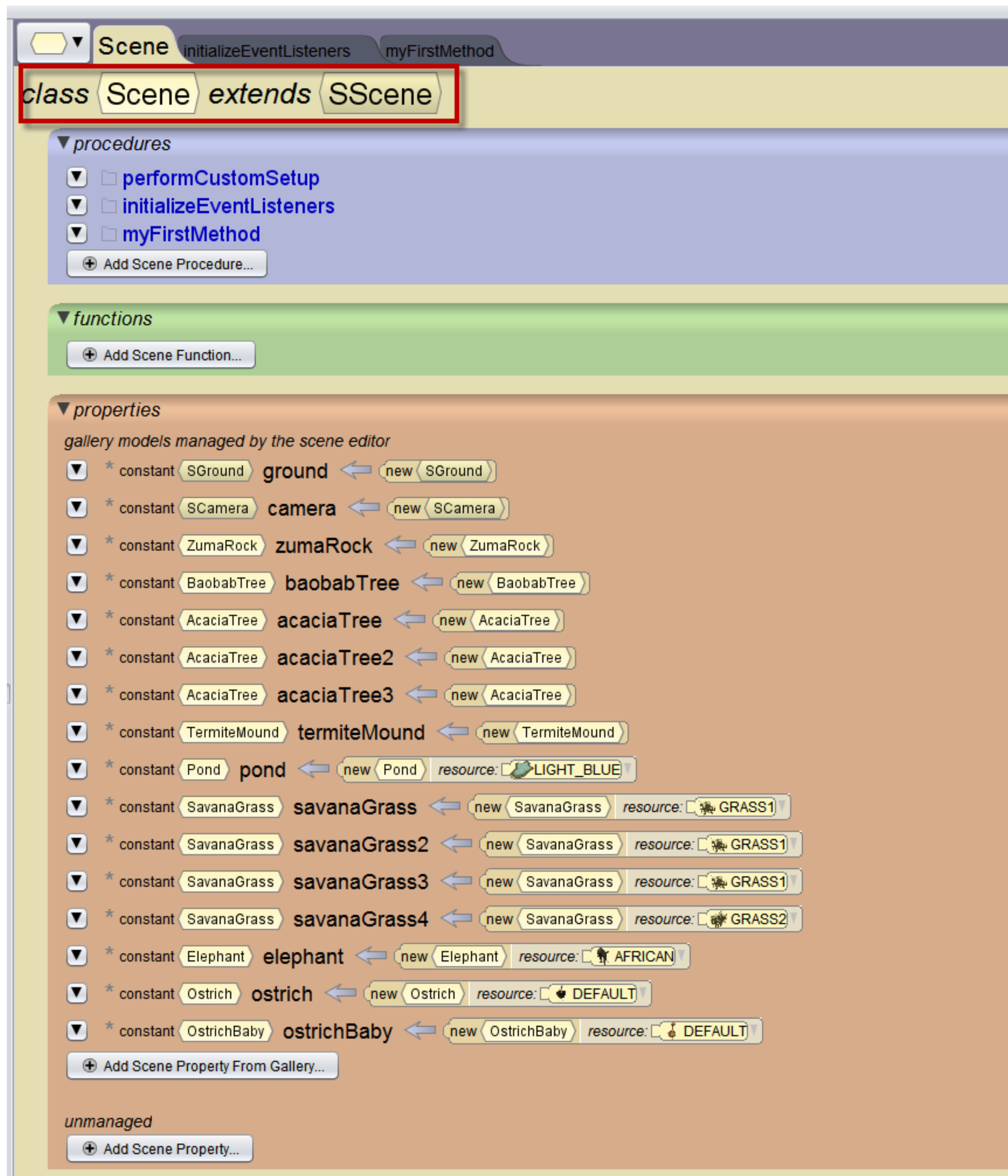


Figure 3 Scene class tab in this example world

Code Editor Tabs – Scene Class

The initializeEventListeners tab displays the current content of the Scene class' initializeEventListeners procedure, as shown in Figure 4. The Scene's initializeEventListeners procedure is similar to an Events editor, where listeners for specific events may be created. The *initializeEventListeners* tab displays the current content of the Scene class' initializeEventListeners procedure, as shown in Figure Figure 4. The Scene's initializeEventListeners procedure is similar to an Events editor, where listeners for specific events may be created.

To explain listeners and events, let's use real world analogies. An event is something that happens. It may be a huge event such as a football game held in an arena or a very small event such as the blink of an eye or a mouse-click. A listener is an alert that tells Alice to listen for an event. A listener acts like an alarm clock. When an alarm clock is set, the clock keeps watch (listens) for a specific time to occur. When the time event occurs, the alarm clock starts a buzzer or turns on the radio to a selected channel.

Similarly, in interactive computer programs (for example, tutorials and games), we make use of events and listeners. An event can be things like a mouse click on a button, a key press on a keyboard, or a touch on the monitor. An event listener keeps watch for the event to occur and then responds to the event in some way.

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

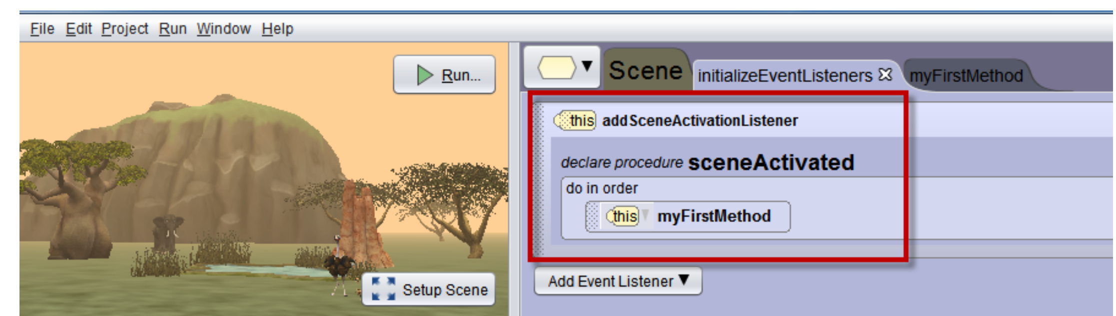


Figure 4 The Scene's initializeEventListeners tab

Additional event listeners may be created in the initializeEventListeners tab. To add a new event, click the AddEventListener button. A popup menu is displayed where four different kinds of event categories are displayed, as shown in Figure Figure 5. The event categories are Scene Activation / Time, Keyboard, Mouse, and Position / Orientation. Each category has several options of listeners that may be selected. Figures Figure 6-Figure 9 show the event listener options for each category, one at a time.

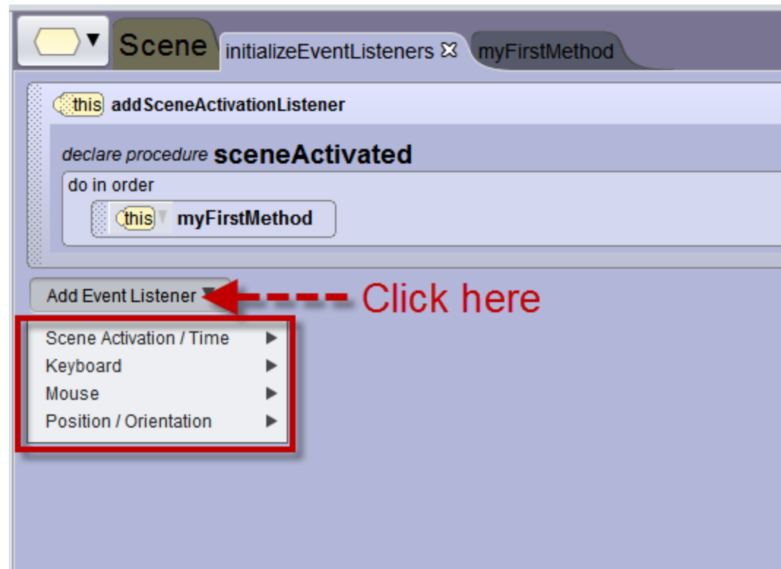


Figure 5 Event listener categories menu

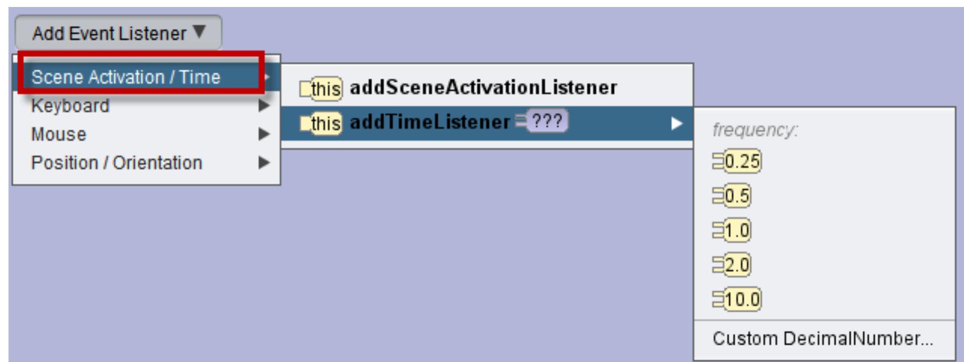


Figure 6 Scene Activation and Time event listeners

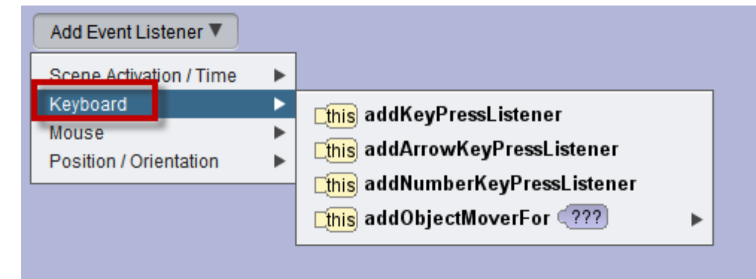


Figure 7 Keyboard event listeners

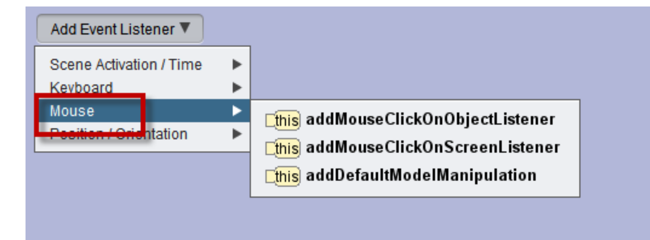


Figure 8 Mouse click event listeners

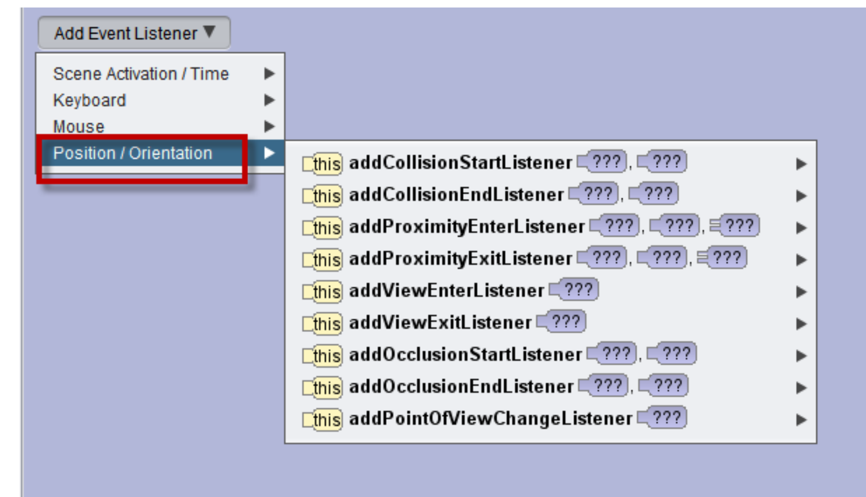


Figure 9 Position and Orientation event listeners

As an example of adding an event listener, suppose we want to allow the user to move objects around in the scene. As shown in Figure 10, select the Mouse event listener category and then *addDefaultModelManipulation* in the cascading menu.

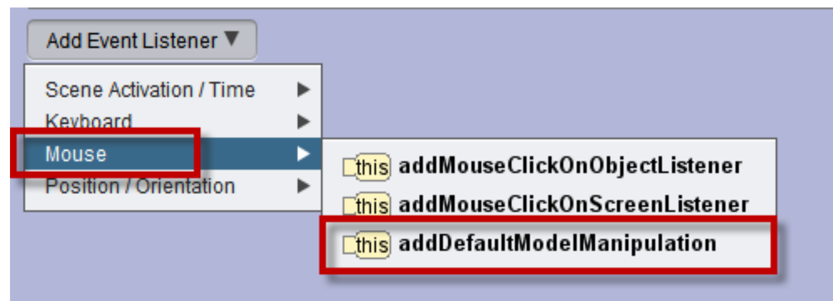


Figure 10 Example--Select Mouse/addDefaultModelManipulation listener

The resulting event listener can be seen in Figure 11. This event listener watches for a mouse click-and drag action. 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 11.

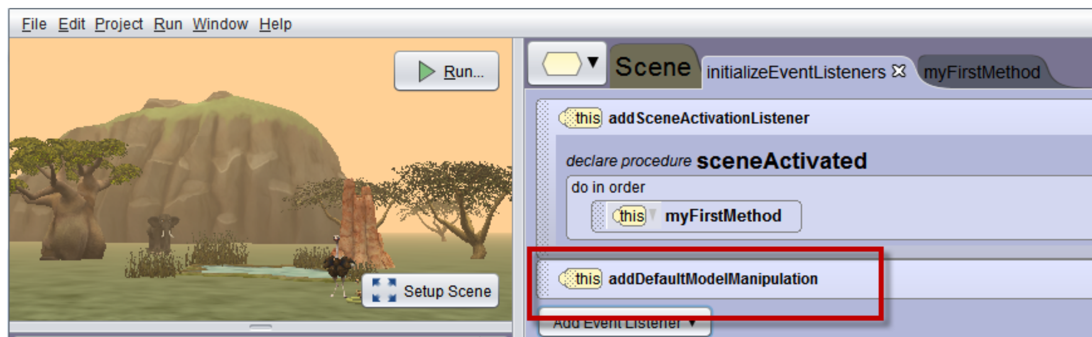


Figure 10 addDefaultModelManipulation event listener code

MYFIRSTMETHOD TAB

The myFirstMethod tab is where program you will likely create program statements that you expect to be performed when the Run button is clicked. Built into myFirstMethod is a first line of code (do in order), as shown in Figure 11. Do in order is a control statement that tells Alice to perform any statements in myFirstMethod in the order in which they listed (one at a time, one after the other). In the example shown in Figure 11 no further program statements have yet been added to myFirstMethod.

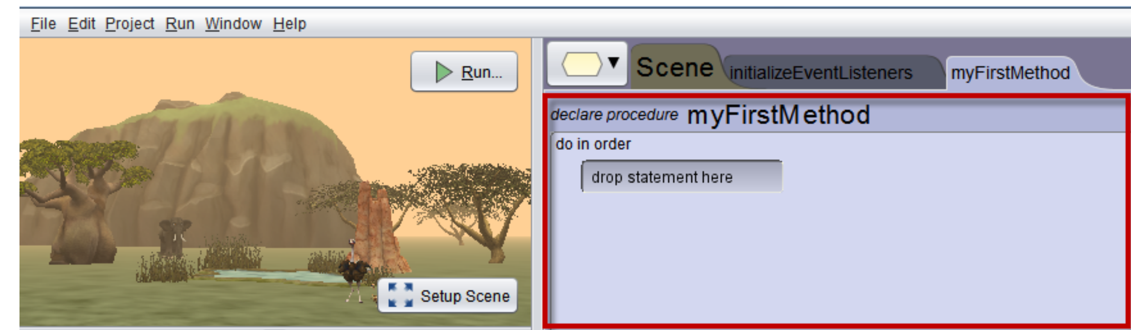


Figure 11 The Scene's myFirstMethod tab

SECTION 2

∏

How to work with a class (other than Scene)

The default class in the Code editor is the Scene class. What if you want to create program code for a different class? For example, suppose you want to write code for the Elephant class to teach an elephant object how to flap its ears.

OPEN A CLASS TAB (IN ADDITION TO THE DEFAULT SCENE TAB)

As previously described in Part 1, Section 5 of this How-To guide, a Class tree menu button is provided (just to the left of the Scene class tab). For your convenience, selecting a class is illustrated again in Figure 1. Click on the class name in the tree and also in the cascading menu. In this example, the Elephant class is selected and Alice responds by opening the Elephant class tab in the editor, as shown in Figure 2. You may notice that the Scene class and associated procedure tabs are still available in the editor, but the Elephant class tab is also now displayed as an editor tab.

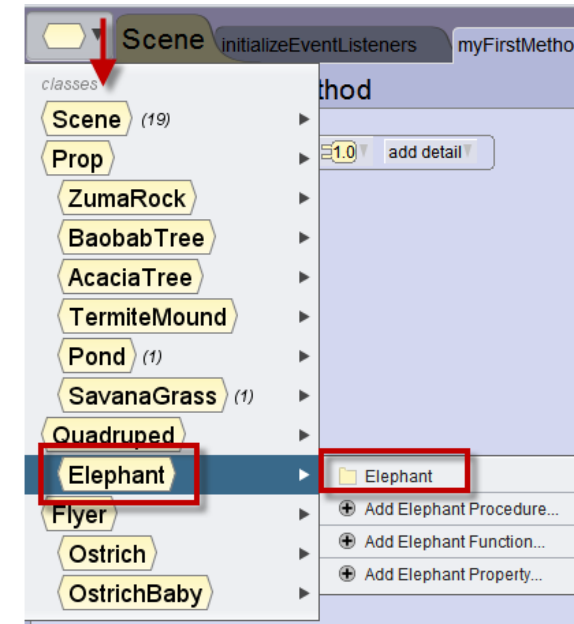


Figure 1 Selecting a class from the class tree menu

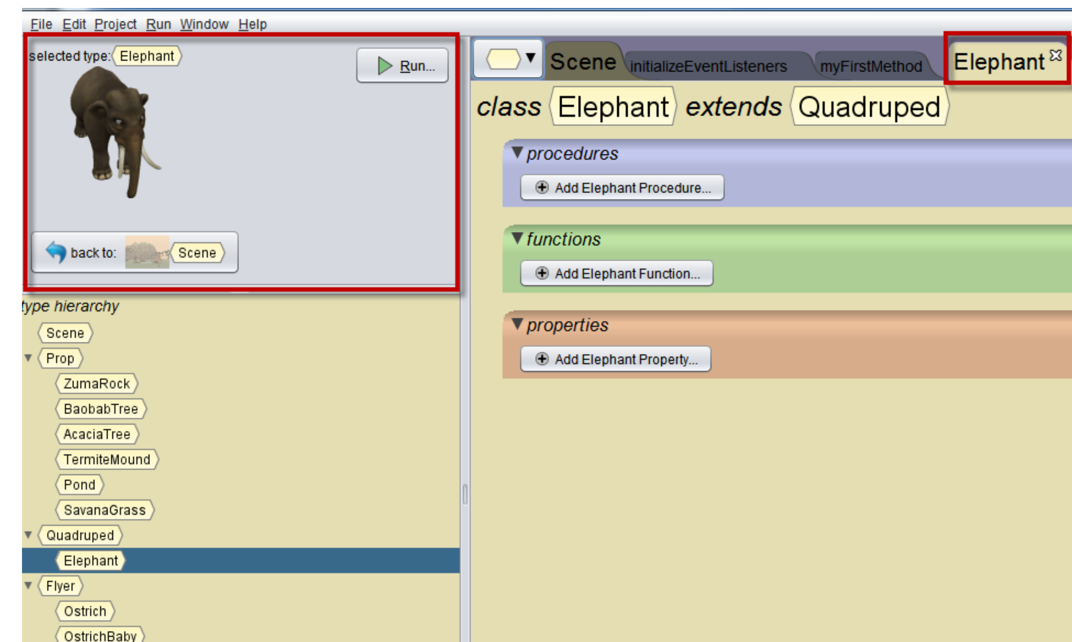


Figure 2 Elephant class is now actively displayed

OPEN A NEW METHOD (PROCEDURE OR FUNCTION) TAB

Now that the Elephant class tab is displayed, tabs can be opened for an existing Elephant class method (procedures and functions) or new methods may be created. Also, existing properties can be edited or new

properties created. As an example, we will illustrate creating a new procedure. As shown in Figure 3, we clicked the AddElephantProcedure button. When the add procedure button is clicked, a dialog box pops up where the name of new procedure may be entered. In Figure 3, we entered the name flapEars.

Figure 3 Elephant class is now actively displayed

After a name for the new procedure has been entered and the OK button is clicked, Alice opens a new tab for the new procedure, as shown in Figure 4. Code statements can be created in this tab to provide animation instructions for flapping an elephant's ears.

Figure 4 flapEars procedure tab is now active

Ordering of tabs in the Code editor

Look closely at the positioning of the tabs for classes and procedures in Figure 5, above. Tabs in the Code editor are always arranged such that a Class tab is immediately followed by method tab(s) for procedures and functions defined by that Class, as illustrated in Figure 18.4. The Scene class tabs are open by default. A newly opened class is always opened to the right of the Scene class's tabs.

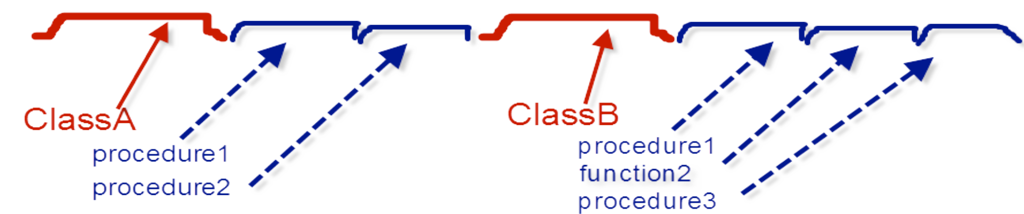


Figure 5 Order: Class tab is followed by its procedure & function tabs

Closing a tab

To close a tab, click on the X in the upper right of the tab, as shown in Figure 6. A Class document tab cannot be closed until all its procedure & function tabs have been closed.

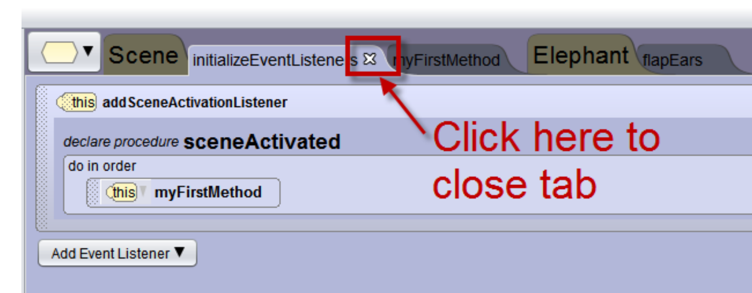


Figure 6 How to close an editor tab

SECTION 3

∏

How to create program statements

The important thing to know about the Code editor is: this is where you create your program statements. Also, when the Run button is clicked, the program statements in this scene's myFirstMethod will be performed.

So, to illustrate how to create a program statement, we will create statements in myFirstMethod.

DRAG AND DROP A PROGRAM STATEMENT

In our example scene, the baby ostrich is only a few hours old and is just getting her first look at the world around her. Let's create a statement to have the baby ostrich spin all the way around (one complete revolution). First, select the babyOstrich object in the object menu, as shown in Figure 6.

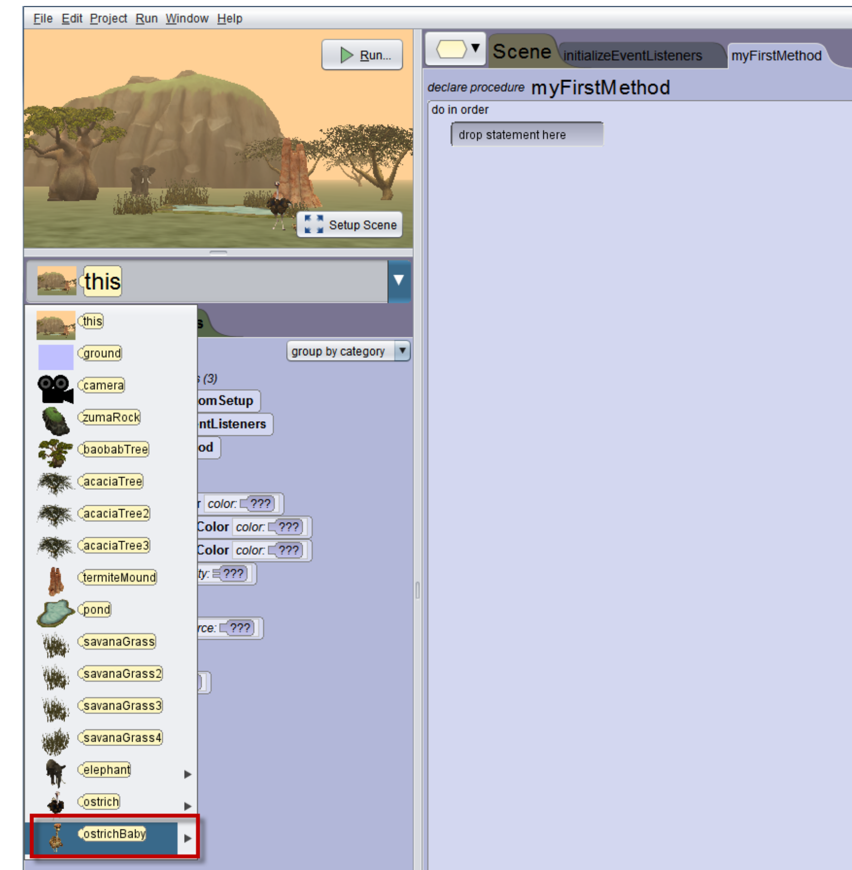


Figure 6 Select ostrichBaby object

Make sure that ostrichBaby is the selected object and then select the turn tile in the Procedures panel. Use the mouse to drag the turn tile into the editor tab, as shown in Figure 7

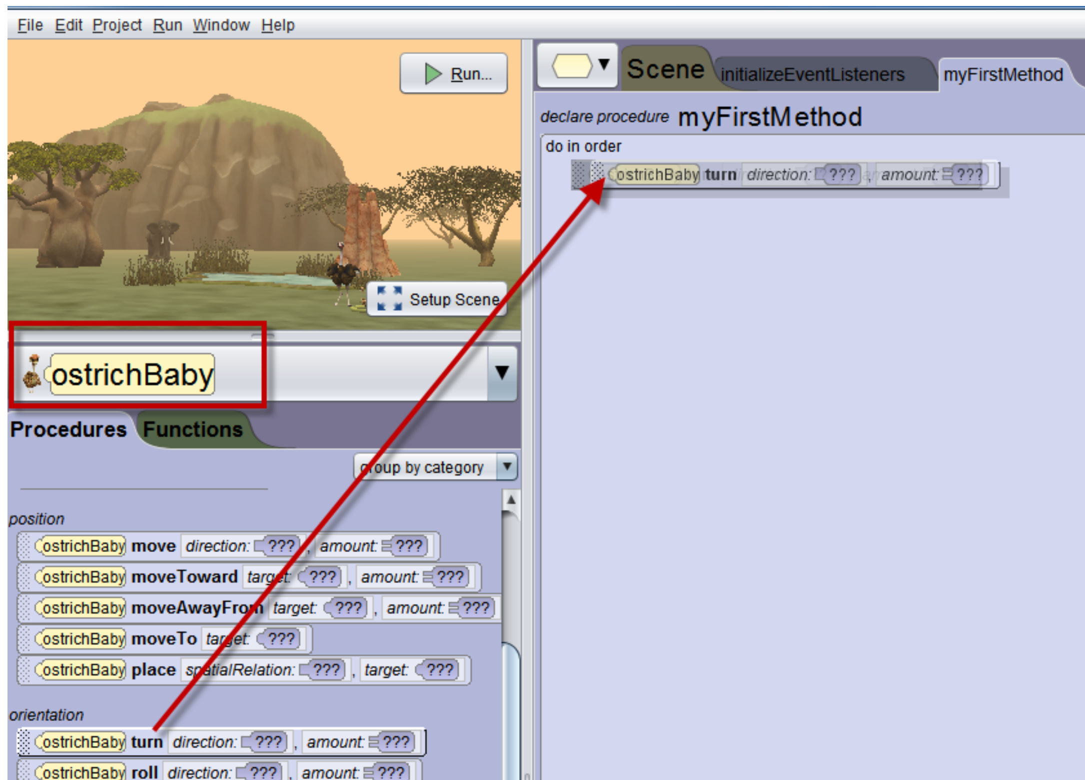


Figure 7 Select ostrichBaby object

When the tile is dropped into the editor, a menu pops up where the direction and the amount of the turn can be specified. In this example, we chose turn left 1.0 revolution, as shown in Figure 8. The resulting statement is shown in Figure 9.

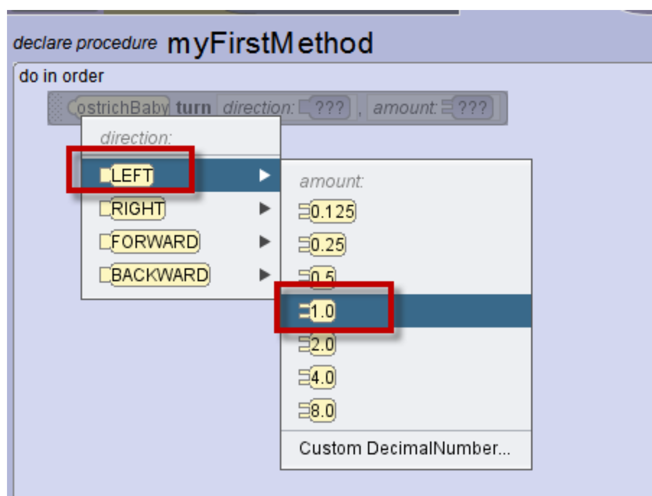


Figure 8 Select direction and amount

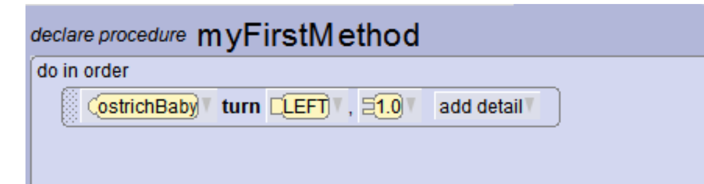


Figure 9 Completed statement

RUN

One of the features of Alice is the program code can be run and tested without first having to write dozens of lines of code and compiling a complete project. Just click on the Run button to view the animation created when your program. A runtime window is displayed where the animation can be viewed, as shown in Figure 10. To view the animation again, just click the Restart button in the runtime window.

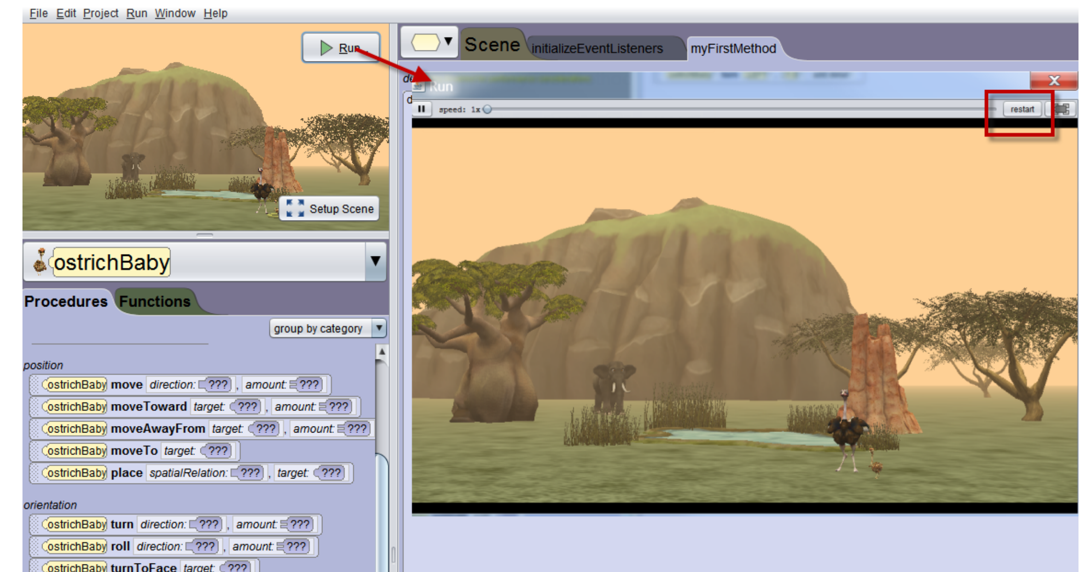


Figure 10 Click on Run to view runtime window

Section 4

∏

How to Cut, Copy, and Paste using the Clipboard

The purpose of this section is to demonstrate using the clipboard to perform cut, copy, and paste in a drag-and-drop Code editor.

Cut, Copy, and Paste items appear in the Edit menu, as shown in Figure 1. However, these items are placeholders…allowing for possible future implementation.

Figure 1 Edit options in the Menu bar

Actually, the traditional cut, copy, and paste actions are useful in a text-based editor but are of limited use in a drag-and-drop editor. In Alice, a clipboard is far more useful as a way to store a single graphic tile (one statement) or a block of graphic tiles (multiple statements) for cut, copy, and paste actions.

Cut

To cut, use the mouse to drag a single graphic tile or a block of graphic tiles into the clipboard, as shown in Figure 2. In the example shown here, an entire do together block is dragged to the clipboard.

Figure 2 Drag code to cut to the clipboard

By default, when the mouse is released, the clipboard turns white and the block of tiles is erased from the editor, as shown in Figure 3. In other words, the default clipboard action is cut.

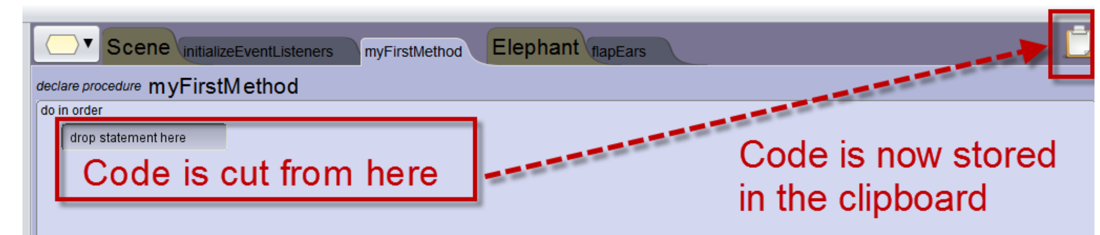


Figure 3 Cut removes selected code tiles

Paste (and remove)

Once code has been stored on the clipboard, it can be pasted into any open tab in the Code editor. In this example shown in Figure 4, we dragged the code from the clipboard back into myFirstMethod, but it could have been pasted onto any tab in the editor. Note, in Figure 4, the color of the clipboard has returned to its usual brown. By default,

dragging a graphic tile out of the clipboard removes the tile from the clipboard. In this example, the clipboard is now empty.

Figure 4 Paste the code from the clipboard into the editor

Copy

To copy code (instead of cutting), press and hold the Ctrl key (the Option key on Mac) while using the mouse to drag the code into the clipboard, as shown in Figure 5.

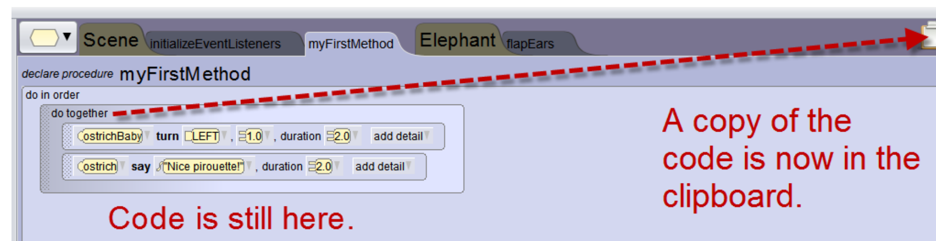


Figure 5 Drag with Ctrl key (Option key on Mac) held down to copy to clipboard

Paste (no remove)

To paste without removing the code from the clipboard, press and held the Ctrl (Option on the Mac) key while dragging from the clipboard into the editor, as shown in Figure 6. Note that the color of the clipboard has remained white. This means the clipboard still holds a copy of the tile, allowing it to be pasted more than once.

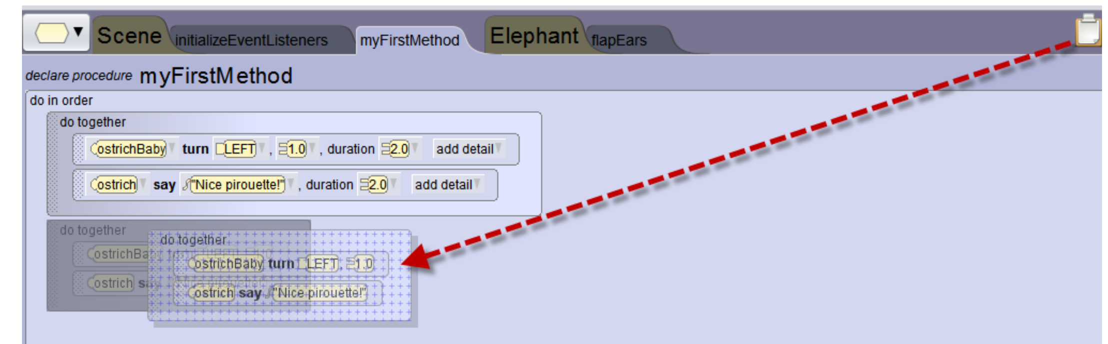


Figure 6 Paste with Ctrl (Option on Mac) to copy from clipboard to the code editor

Note: Cut, copy, and paste actions can result in scope errors. In the examples used here, we worked with myFirstMethod and encounter methods -- both of which belong to the Scene class. Because this scene contains all other objects in the virtual world (in this example, the dolphin and seaPlant1), we had no scope errors.

We wish to caution the reader, however, that if code is cut or copied from a method belonging to one class and then pasted into a method belonging to a different class, a scope error may occur. This is not unique to Alice. This is standard protocol for scope-enabled programming languages, whether working in a text editor or a drag-and-drop editor.

Section 5

How to import and play a sound (audio file)

The purpose of this section is to demonstrate how to import and use audio files for creating sound effects in an Alice 3 animation.

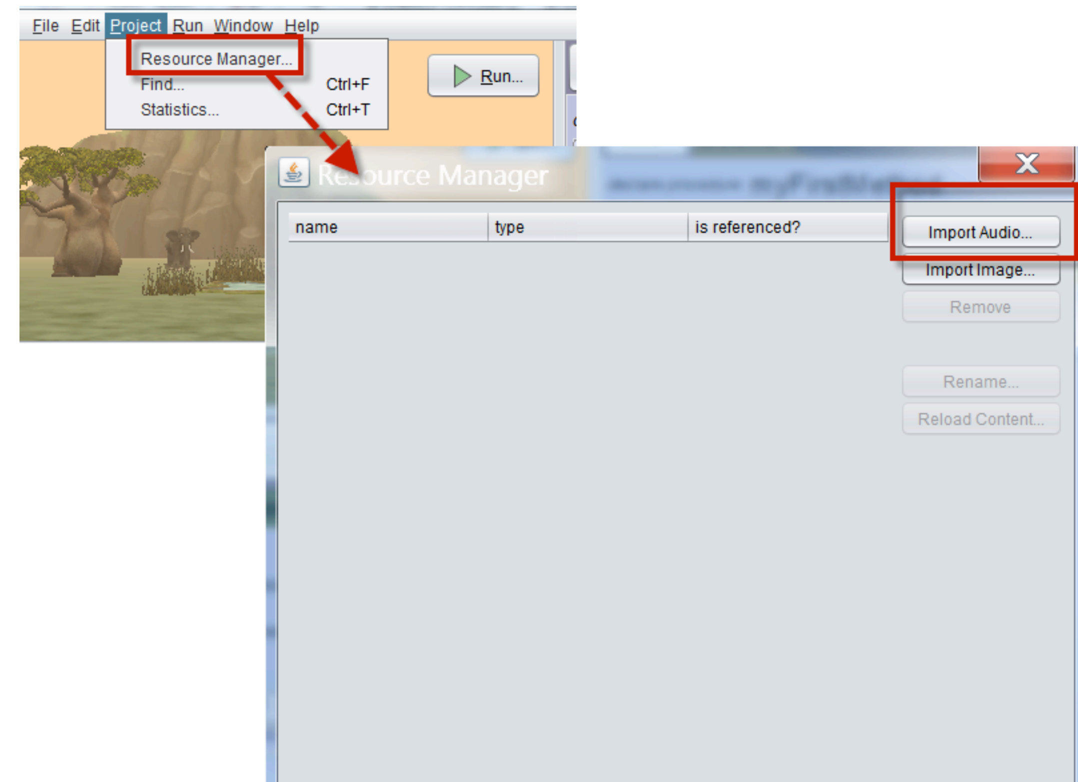


Figure 1. Paste with Ctrl (Option on Mac) to copy from clipboard to the code editor

Import with Resource Manager

One way to import a sound is to use the Resource Manager. The Resource Manager was previously introduced in Part 1, Section 3 of this How-To guide. Also, an example of using the Resource Manager to import a 2D image as a billboard was provided in Part 2, Section 6. In this section, the Resource Manager will be used to import an audio file which can then be used to play sounds in an animation program. In the Project menu, select Resource Manager, as shown in Figure 1. A Resource Manager dialog box is displayed, where you can select the Import Audio button.

When the Import Audio button is clicked, a navigation window is displayed containing a Sound Gallery. The Sound Gallery is a collection of audio files, especially constructed for use in Alice projects. The audio files in the Alice Sound Gallery are freely provided for use in non-commercial, educational projects. Please note, however, that the audio files are copyrighted and may not be used for commercial purposes without prior written permission from Carnegie Mellon University.

You may browse the audio files in the Sound Gallery and select an appropriate audio file for import. In the example shown in Figure 2, we selected drumroll_finish.mp3.

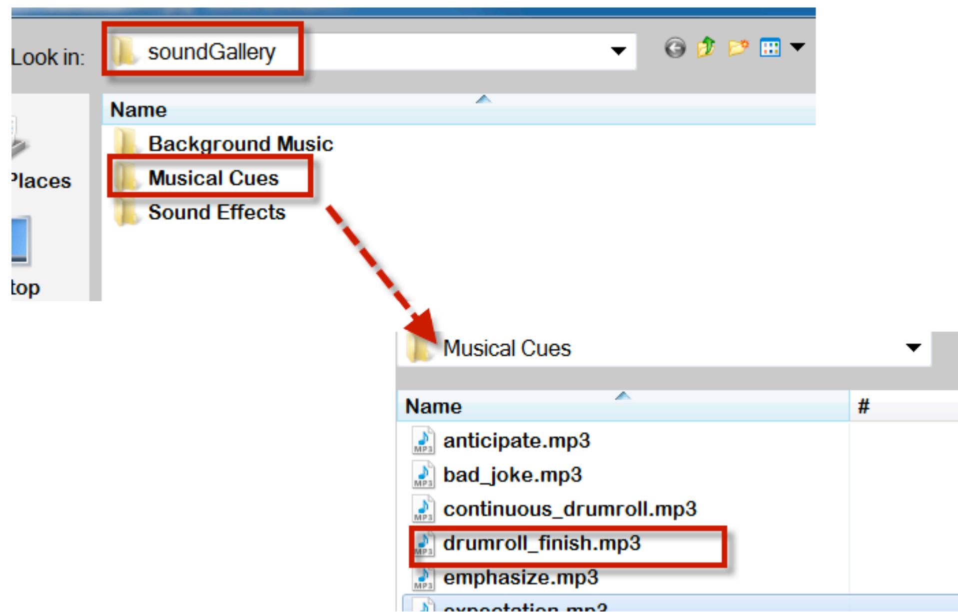


Figure 2 Select an audio file for import

The imported audio file is then listed in the Resource Manager, as shown in Figure 3. Because the file in this example has just now been imported, it is not yet being used in the program code and the Resource Manager indicates that "is referenced?" is NO.

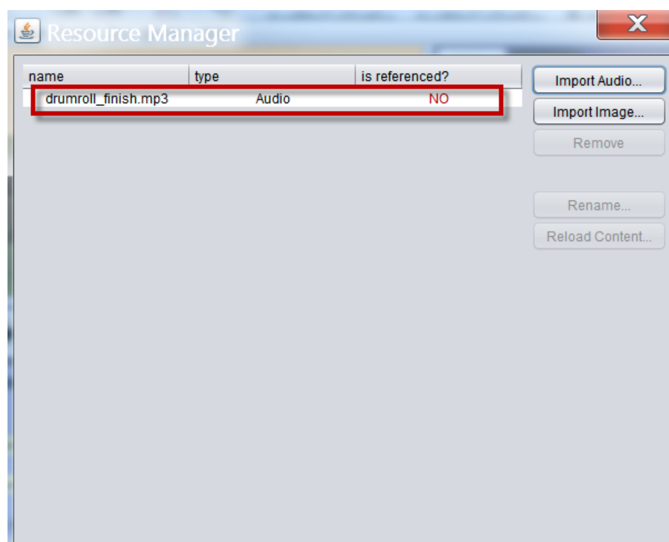


Figure 3 Imported file is listed in the Resource Manager

Play an Audio File

Sound effects in Alice animations are created by playing an audio file. To play an audio file, drag a playAudio tile into the editor, as shown in Figure 4. In this example, we dragged this (the current scene's) playAudio tile into the editor. When the tile is released in the editor, a popup menu offers the option of selecting an audio file that has already been imported into the Resource Manager or to import a different audio file.

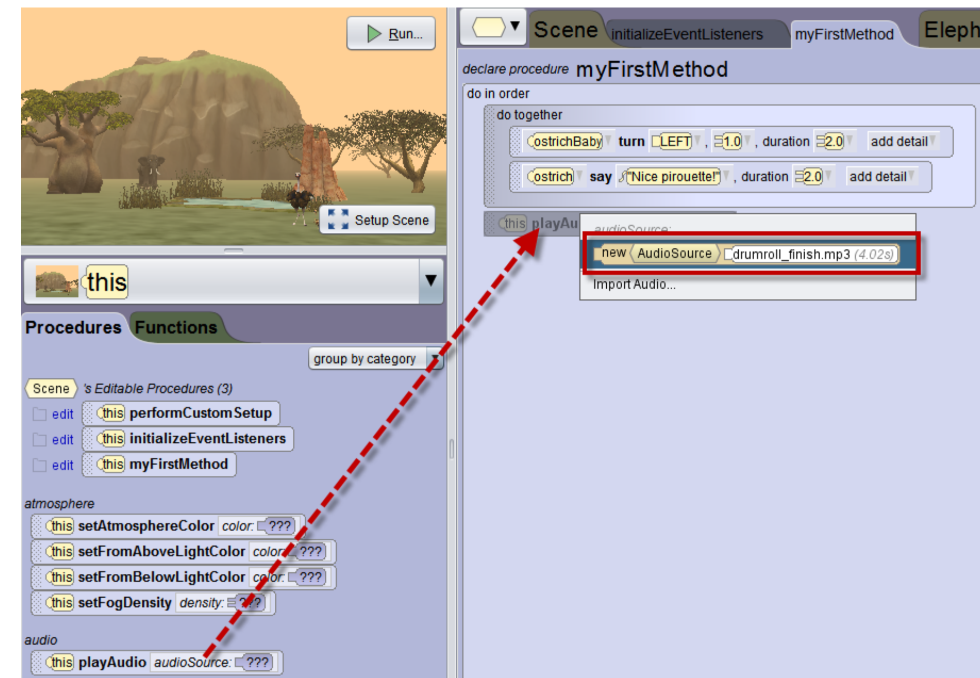


Figure 4 Creating a playAudio statement

As shown in Figure 5, the playAudio statement in this example was positioned immediately after the do together code block where the baby ostrich turns one revolution and the mother ostrich says "Nice pirouette!"

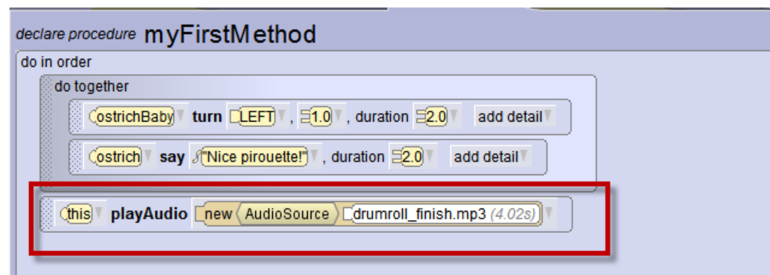


Figure 5 Complete playAudio statement

MODIFYING SOUND EFFECTS

In the example shown above, the sound is playing in a spot that isn't really appropriate for the context and sequence. It would be much better if the drumroll were played prior to the do together code block, or perhaps even within the do together. To modify where a sound is played during the animation sequence, just use the mouse to click-and-drag the statement to a different place in the code sequence, as shown in Figure 6.

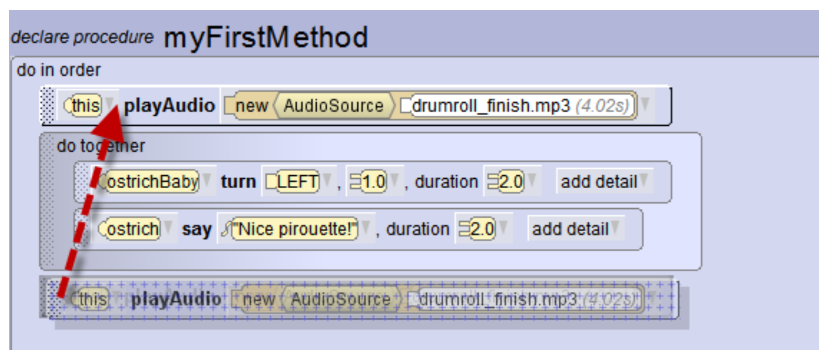


Figure 6 Changing sequence for playing a sound

When playing a sound, we often want to shorten the length of time it plays. For instance, in this example the drumroll is 4.02 seconds. This may seem to be a short time, but is actually much too long for this animation. To play only a portion of the sound, we can customize the start and stop points for playing the sound. Click the arrow at the end of the playAudio statement and then use the slides to select a start time and a stop time for a shorter length of time, as shown in Figure 7. In this example, the start was set at 1.0 and the stop at 0.0703 on the audio timer. As a result, the sound will now play for 1.5 seconds.

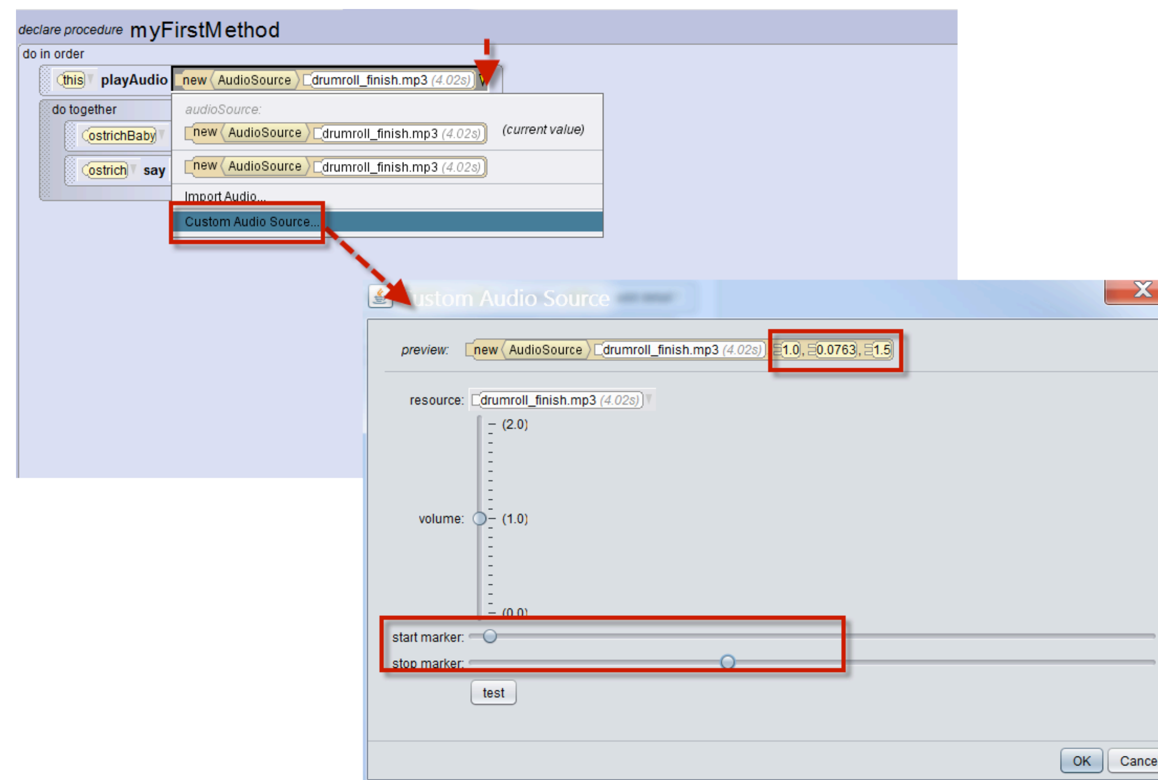


Figure 7 Customizing to play a shorter segment of the audio file

SUGGESTIONS FOR USING AND EDITING AUDIO FILES

Alice is not sound/audio recording studio software. Other applications are available online that performs these actions far better than our resources can suuport. For creating your own recordings, you might consider software such as GarageBand (Apple, Inc.) or Mixcraft 6 (Acoustica) which are not free but are reasonably priced and have user's guides.

For purposes of editing existing audio files, we use and recommend Audacity, free software from Carnegie Mellon University, see:

http://www.cmu.edu/computing/software/all/audacity/

Audacity is highly effective as a tool for extracting and exporting a short audio clip for use in Alice 3. Use of a shorter audio clip can dramatically decrease the size of an Alice project and also helps adhering to the guidelines for educational “fair use.” In any case, we strongly recommend that you observe copyright laws. Of particular importance is the need to guard against redistribution of any copyrighted media.

SECTION 6

HOW TO EXPORT AND IMPORT A CLASS FILE

The purpose of this section is to demonstrate how to reuse Class code by exporting code written for a Class in an Alice project and then (later) importing that file in a different Alice project.

Export

To export code written for a Class in an Alice project, follow these steps:

Step 1: Open the class you wish to export. In the example shown in Figure 1, the Ostrich class is selected in the Class menu.

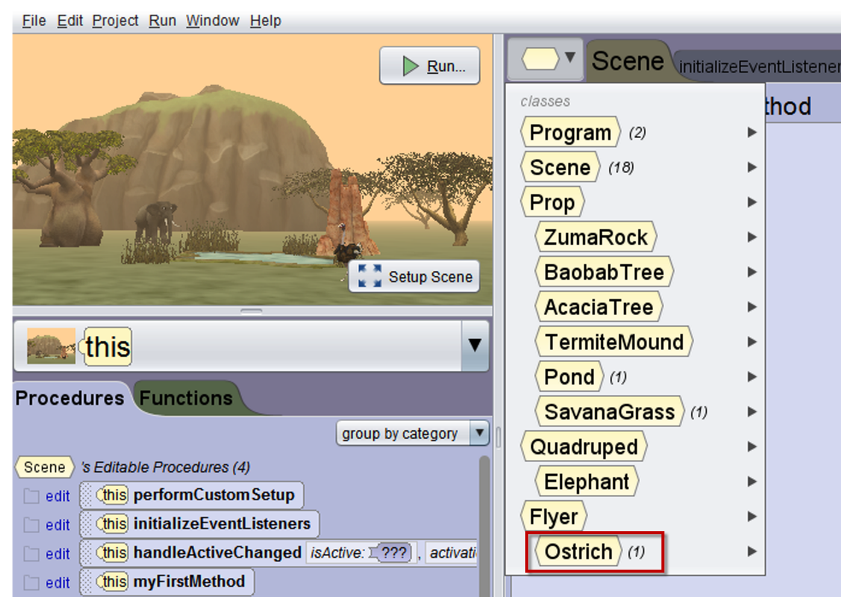


Figure 1 Select a class in Class tab

When a class is selected, the tab for that class should become the active tab in the Edit panel, as shown in Figure 2.

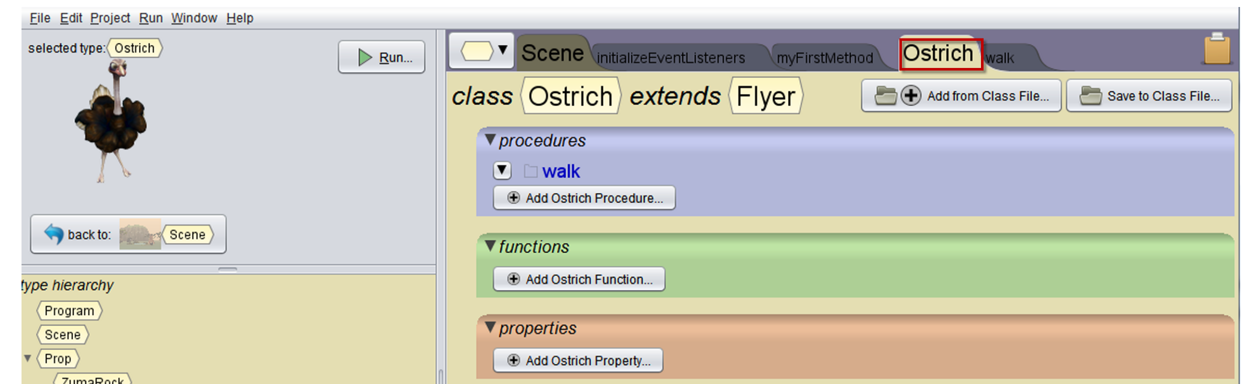


Figure 2 The Ostrich class is the active tab in the Edit panel

Step 2: In this example, we have defined a walk procedure for the Ostrich class. To save this code for use with Ostrich objects in another project, click the Save to Class File button. A save file dialog box is displayed, as shown in Figure 3.

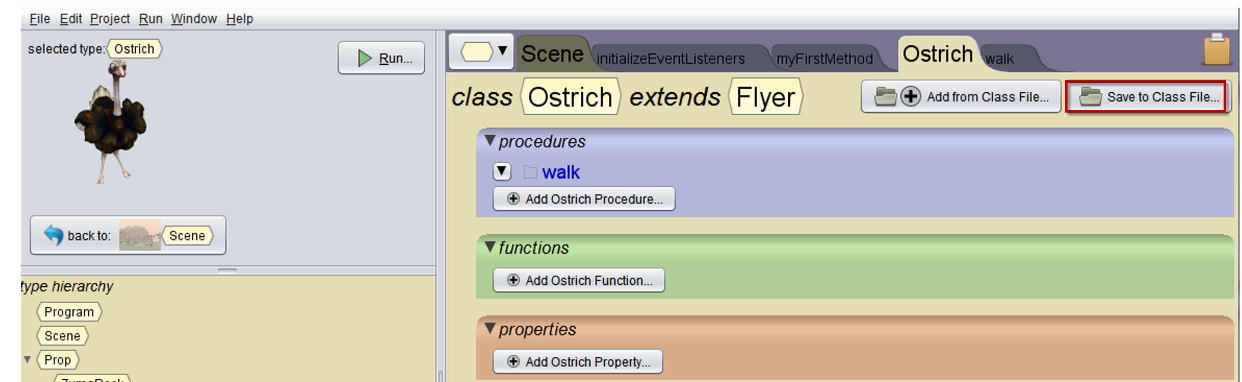


Figure 3 Click the Save to Class File button

Alice automatically creates a MyClasses folder where class files can be saved. Save the file.

Note: We recommend that you save the file in MyClasses, but you may select another location on your computer for storing the file.

You do not need to enter the filename extension; Alice automatically adds .a3c as the file format.

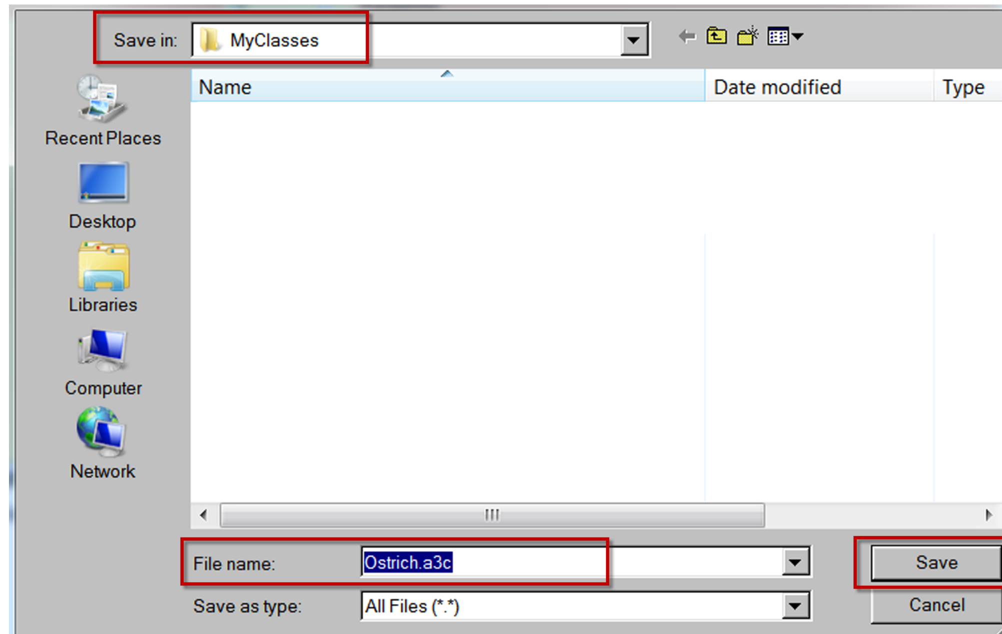


Figure 4 Save the class file

If the class file has been successfully saved in the MyClasses directory, it should appear in the Gallery tab labeled My Classes, as shown in Figure 5.

Figure 5 Exported classes in the My Classes directory

Import

To illustrate how to import previously exported code, we will build upon the Ostrich export example described above. We are assuming the Ostrich class, with the walk procedure, has been exported. Sometime later, we create a new Alice project that has an Ostrich object, as shown in Figure 6.

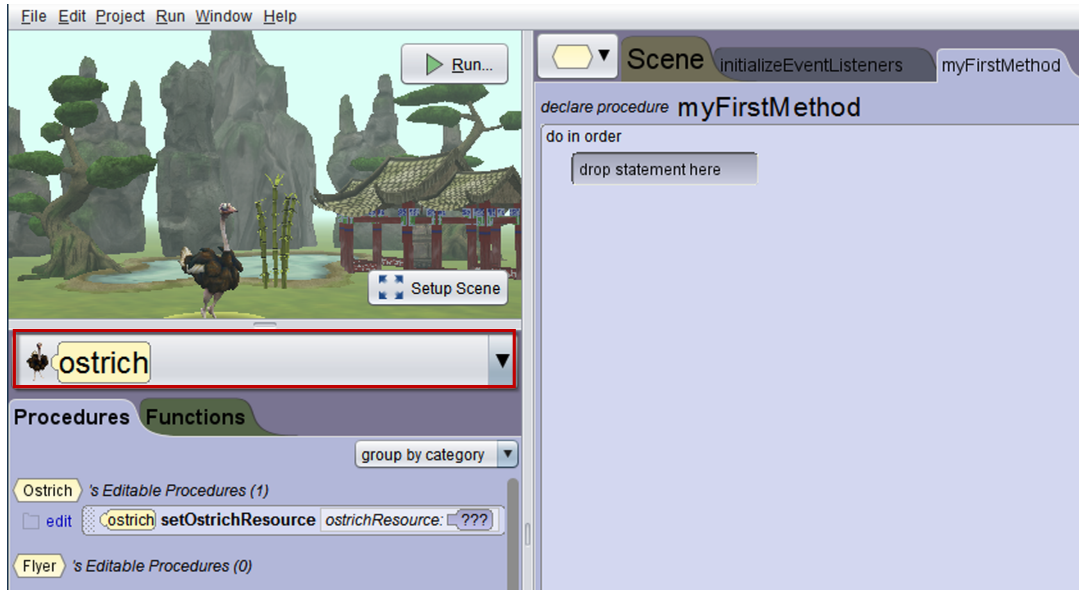


Figure 6 A different project with an ostrich object

Step 1. Select the class in the Class menu. You should see the tab for that class as the active tab in the Edit panel, as shown in Figure 7. In this example, the ostrich object was added to the scene using the Ostrich class in the Flyer Gallery. The Ostrich class in the Flyer Gallery does not have a walk procedure. (This is not unique -- none of the Flyer classes have a pre-defined walk procedure.)

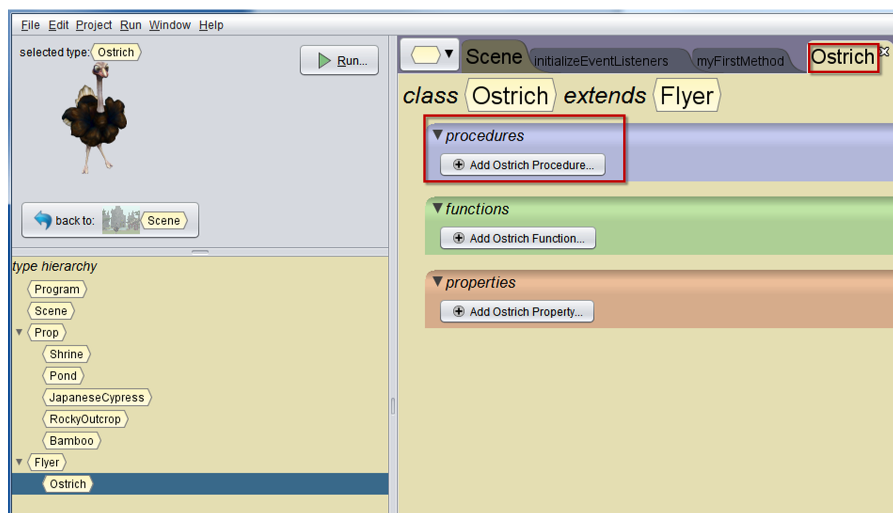


Figure 7 No walk procedure

Step 2. Click the Add from Class File button in the Class tab, as shown in Figure 8.

Figure 8 Click the Add from Class File button

A Save File dialog box is displayed, as illustrated in Figure 9. In this example, we clicked on Ostrich.a3c and then the Open button.

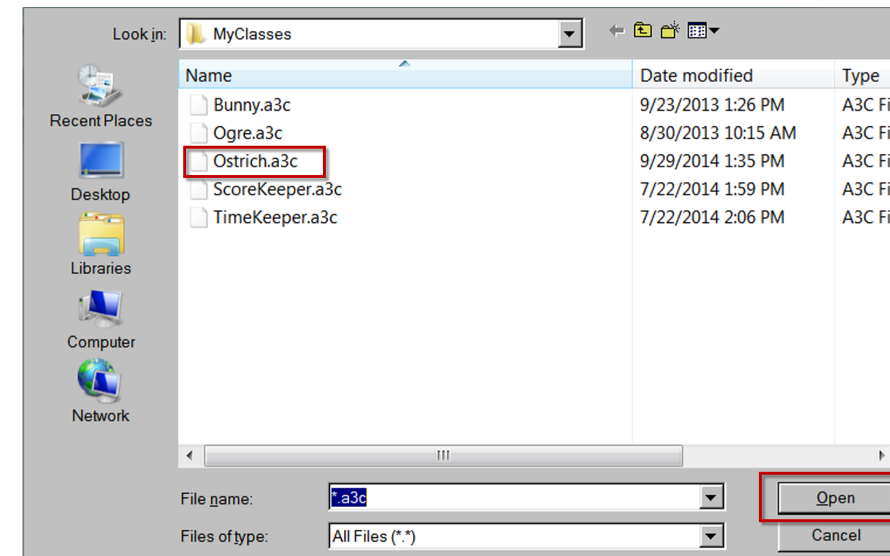


Figure 9 Select the desired class file and open it

A Class file content dialog box is displayed, where you can select the procedures, functions, or properties you wish to import. In the example shown in Figure 10, we selected the walk procedure and then clicked the Next button.

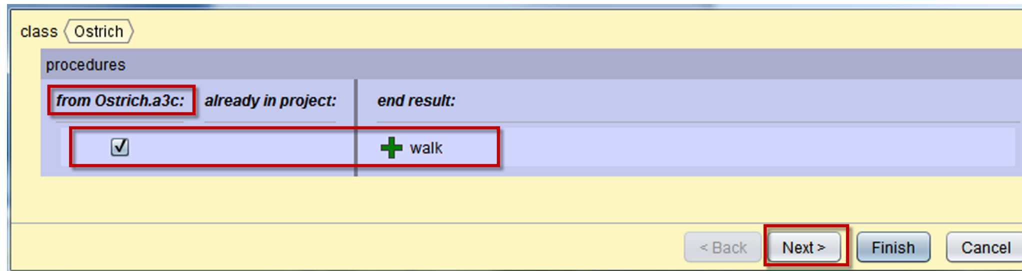


Figure 10 Select the class content to be imported

A Merge dialog box is displayed, as shown in Figure 11. If any conflicts exist between code already in the project and the code to be imported, this dialog box will provide options for selecting which version you wish to keep. If you wish to keep more than one version, the versions can be renamed to avoid name conflicts. When conflicts, if any, have been resolved, click the Finish button.

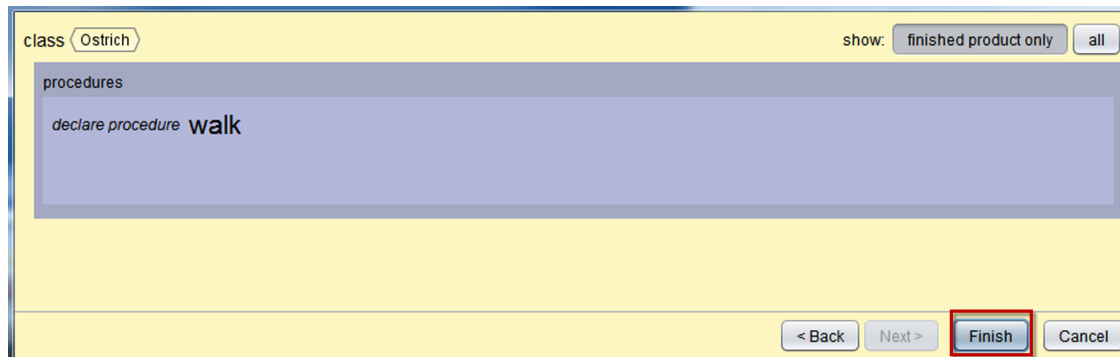


Figure 11 Click Finish

Imported class file content will now be listed in the Class tab, as shown in Figure 12.

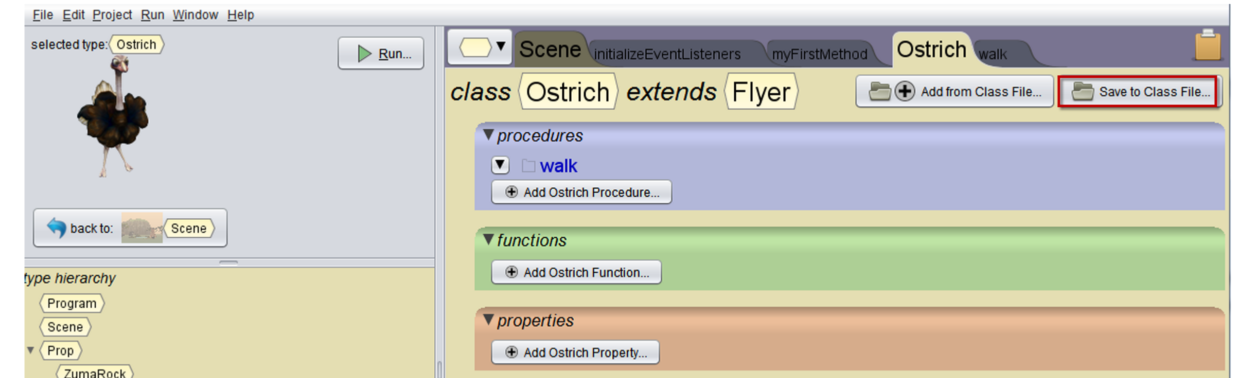


Figure 12 Import is complete

A detailed tutorial for Export and Import is available as a Video at

materials_videos.php

CHAPTER 4

EVENTS FOR INTERACTIVE & GAME PROGRAMMING

The Scene class defines a procedure named *initializeEventListeners*, similar to an Events editor, where **listeners for specific events** may be created.

Chapter 3 provided a “surface” description of listeners and events, appropriate for interactive programming. This section explores events and listener options from the perspective of game programming.

SECTION 1

∏

SCENEACTIVATED LISTENER

The Scene class defines a procedure named *initializeEventListeners*, as shown in Figure 1. The initializeEventListeners is similar to an Events editor, where listeners for specific events may be created.

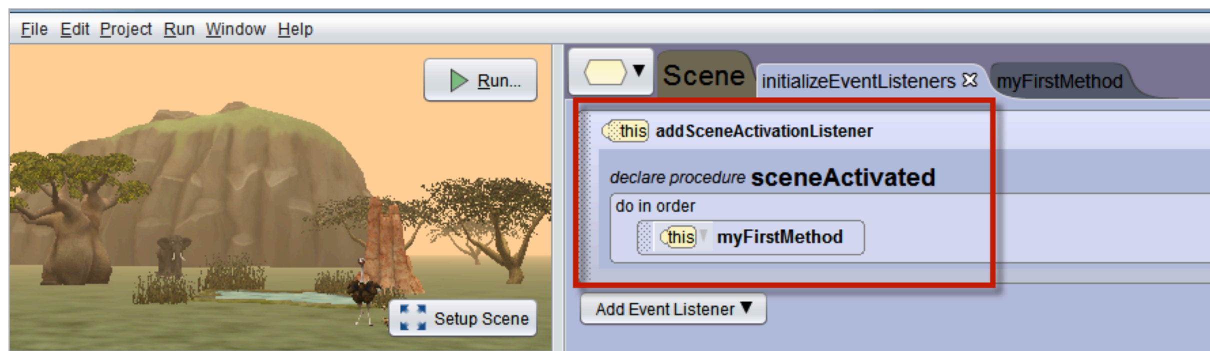


Figure 1 The Scene's initializeEventListeners tab

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).

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

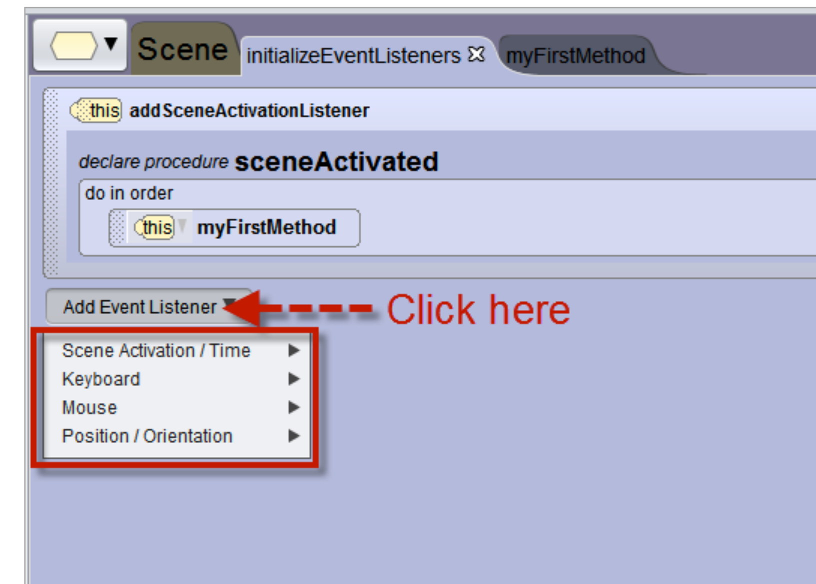


Figure 2 Event categories menu

Section 2

∏

Scene Activation / Time Events

The Scene Activation / Time category has two menu options, as shown in Figure 1.

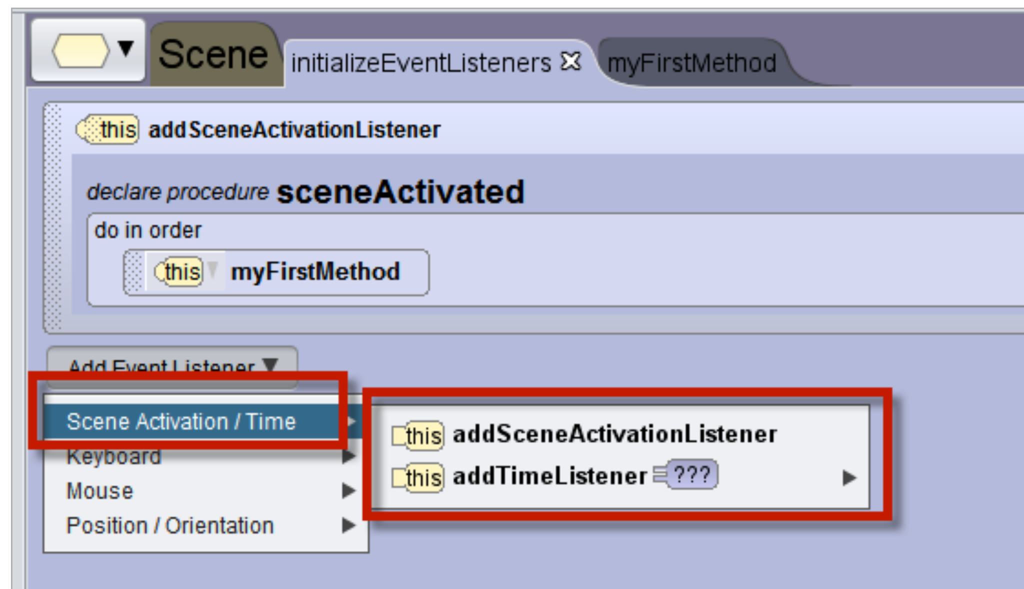


Figure 1 Scene Activation/Time listener menu options

Scene Activation

If *addSceneActivationListener* is selected, another **sceneActivated** listener is added to the editor, as shown in Figure 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.

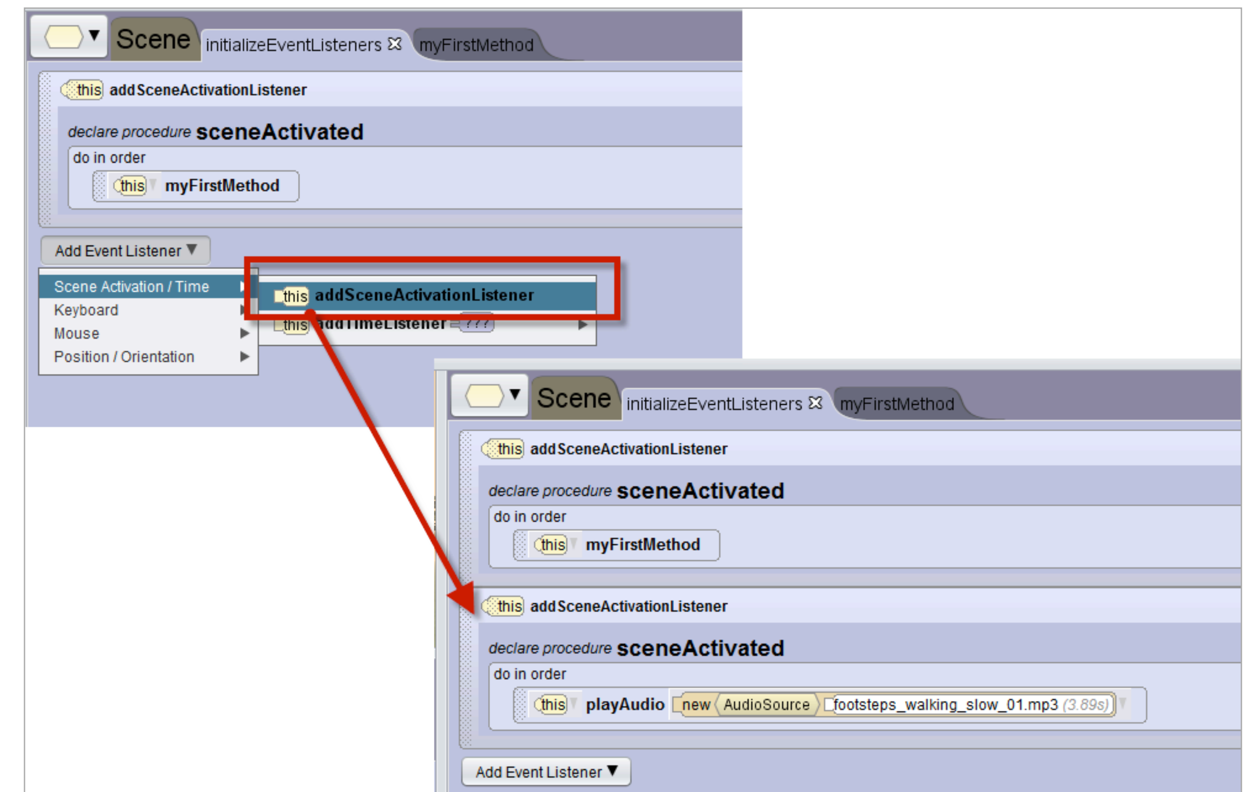


Figure 2 Two sceneActivated listeners, each with their own action

Time

If *addTimeListener* is selected, a menu cascades to select a time interval, as shown in Figure 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.

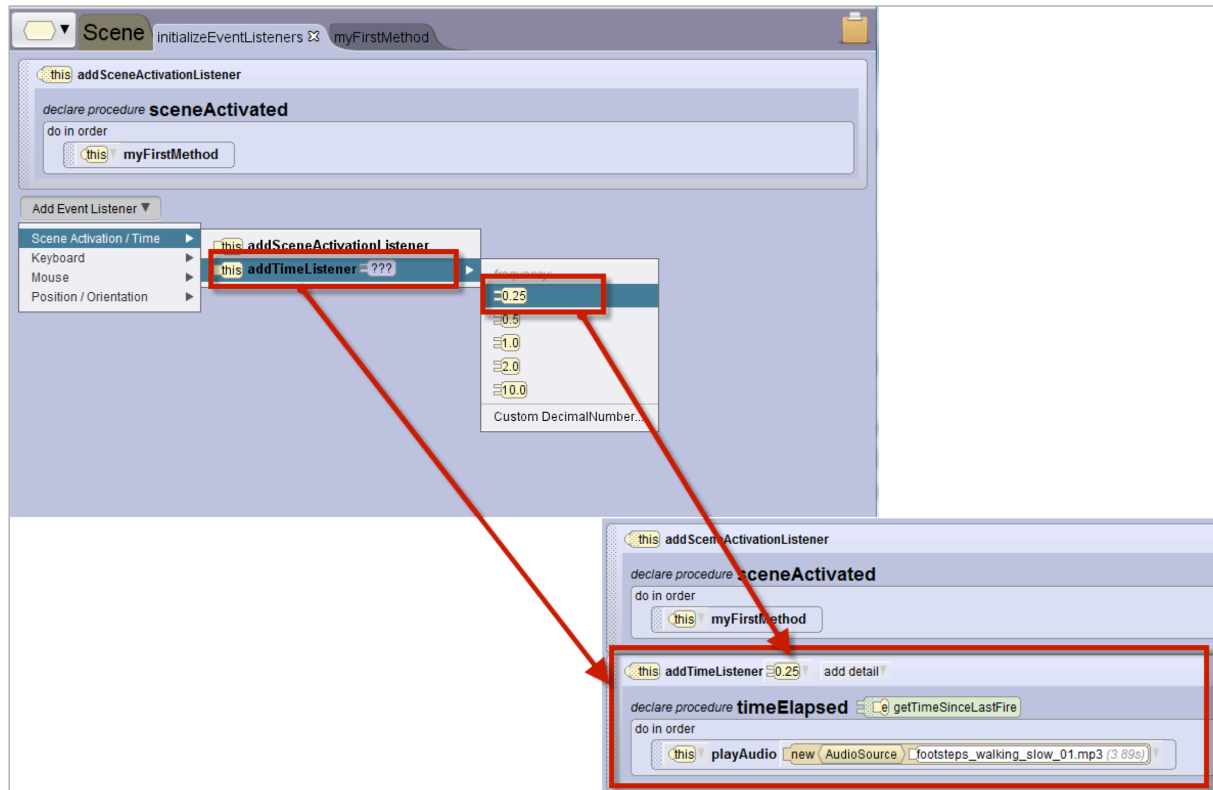


Figure 3 Time event listener

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 4.

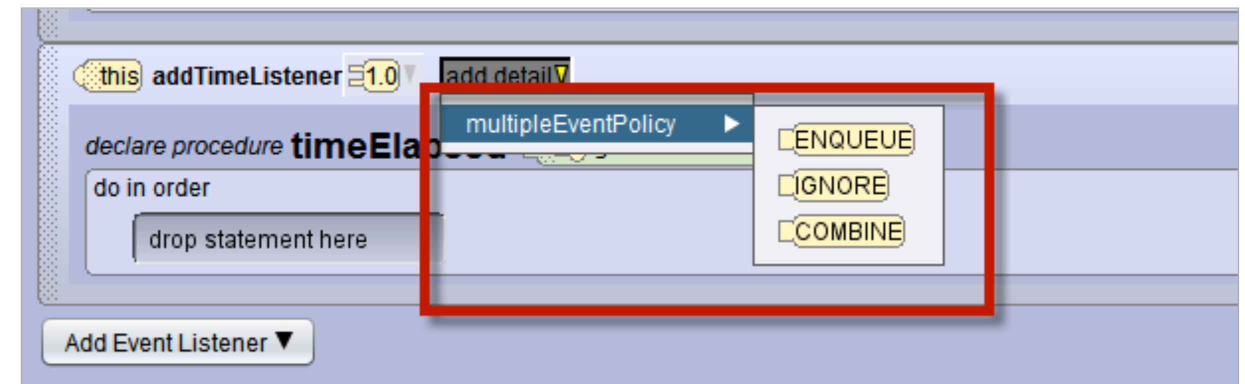


Figure 4 Multiple event policy menu

- 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 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 5 ENQUEUE option for timeElapsed listener

SECTION 3

∏

KEYBOARD EVENTS

The Keyboard category has four menu options, as shown in Figure 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.

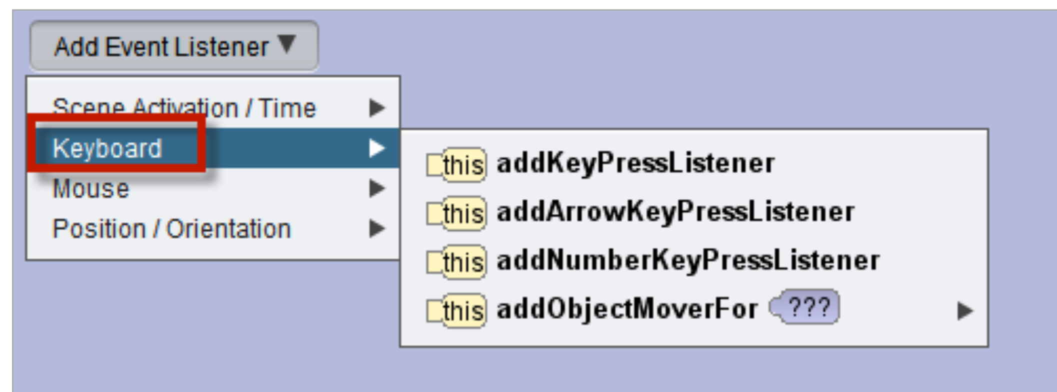


Figure 1 Keyboard event listeners

KEYPRESS

When *addKeyPressListener* is selected, a **keyPressed** listener is added to the editor, as shown in Figure 3.

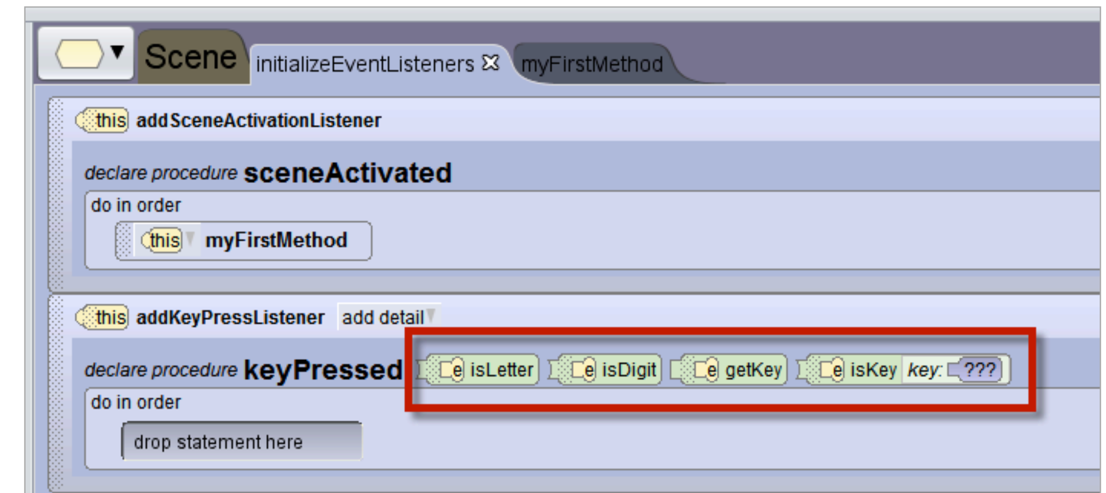


Figure 2Keyboard event listener

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 3 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.

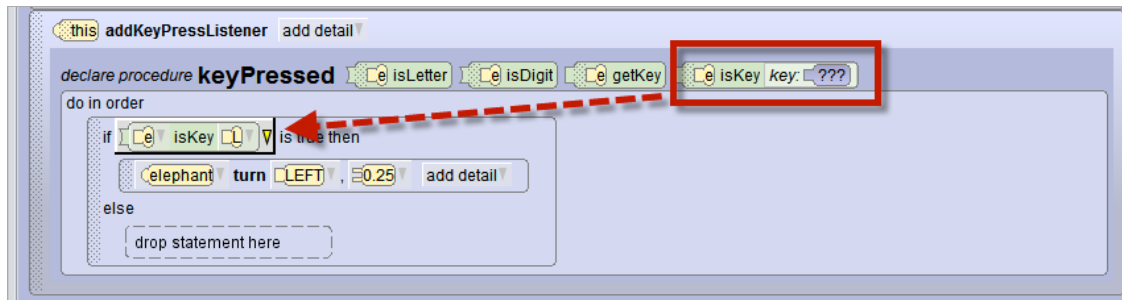


Figure 3 Using the isKey function within a keyPressed listener

As a more generic example, in Figure 4 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.

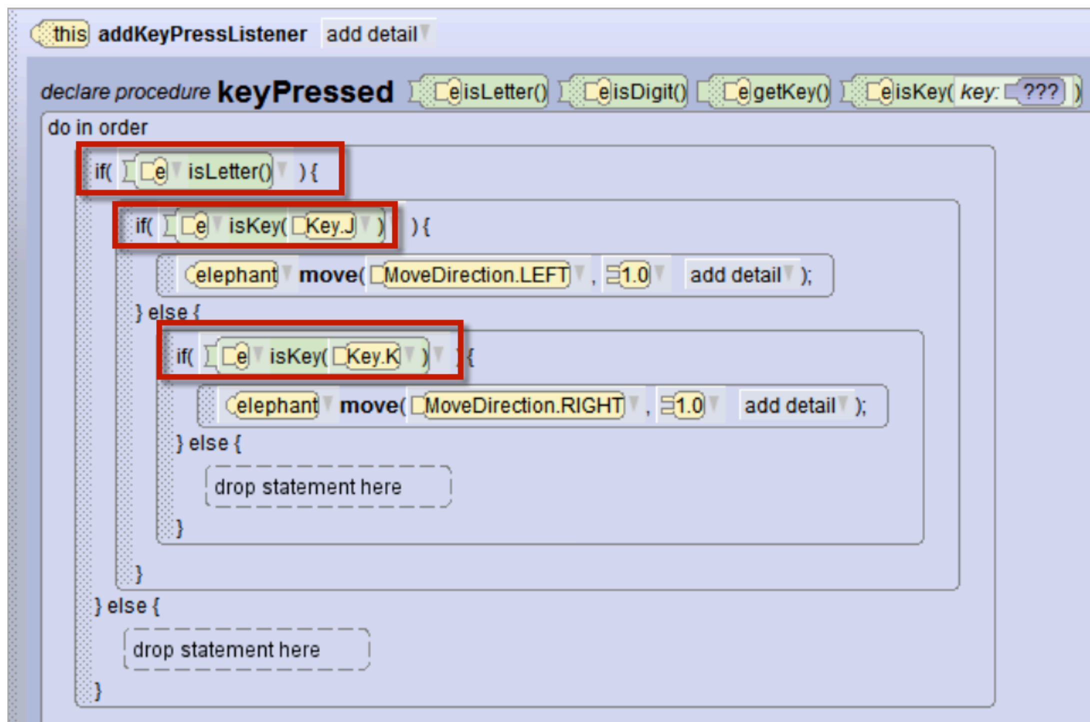


Figure 4 Using multiple functions within a keyPressed listener

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 5.

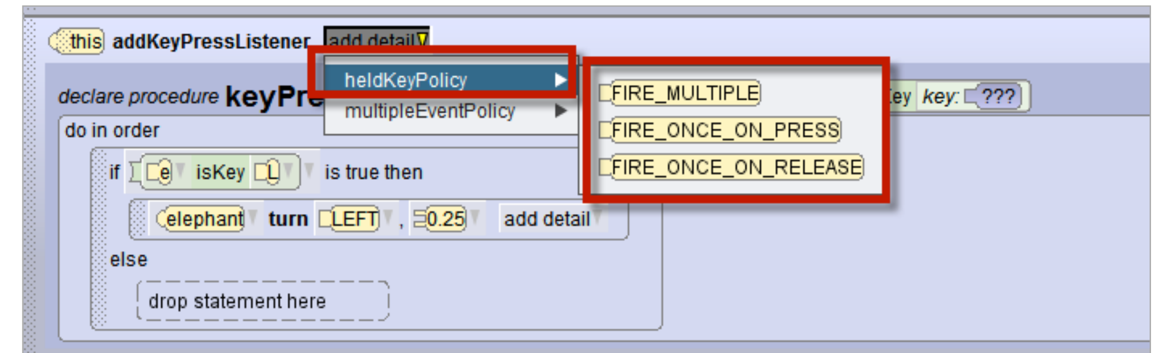


FIGURE 5 SELECTING A HELDKEYPOLICY

- 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 6. 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.

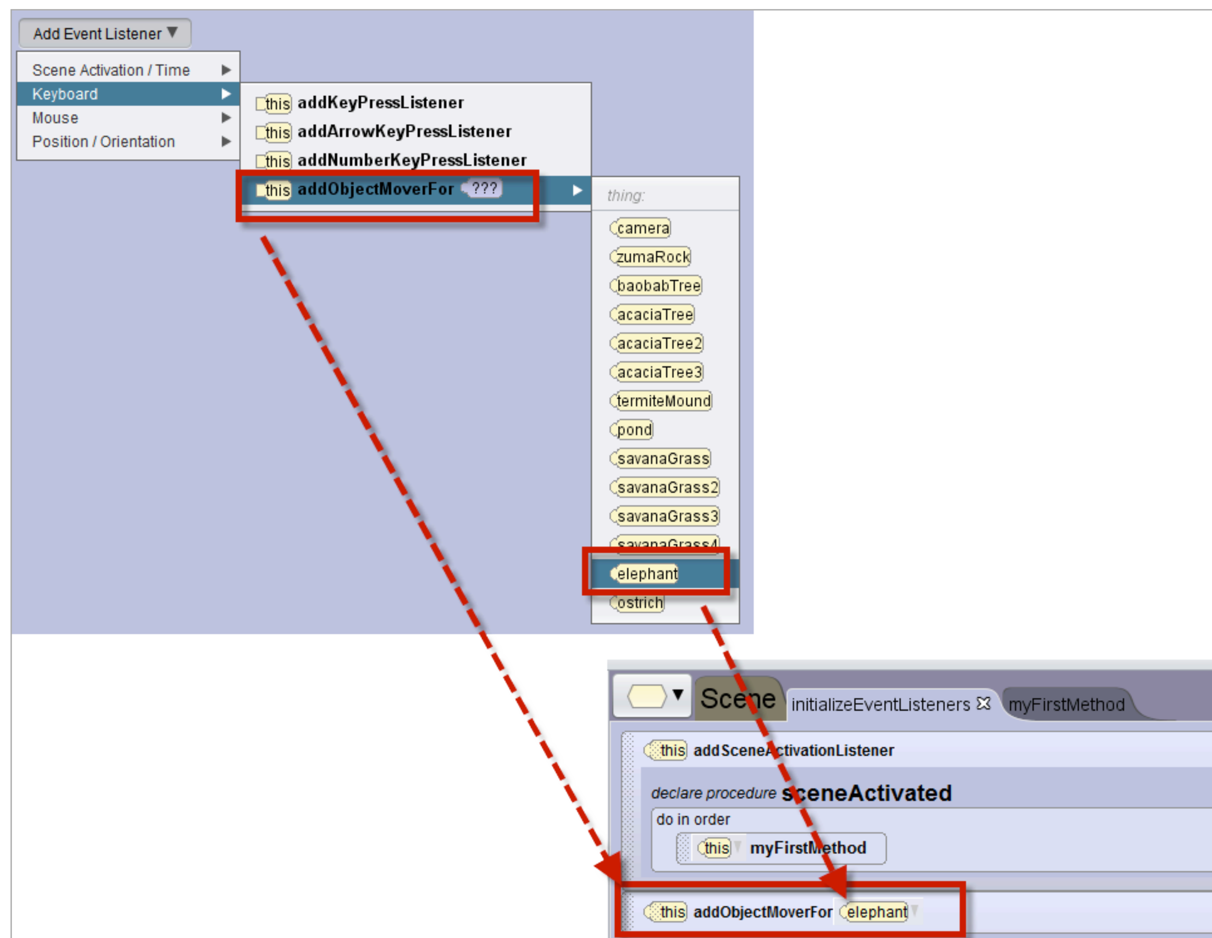


Figure 6 Keyboard event listener

SECTION 4

∏

MOUSE EVENTS

The Mouse events category has three menu options, as shown in Figure 1.

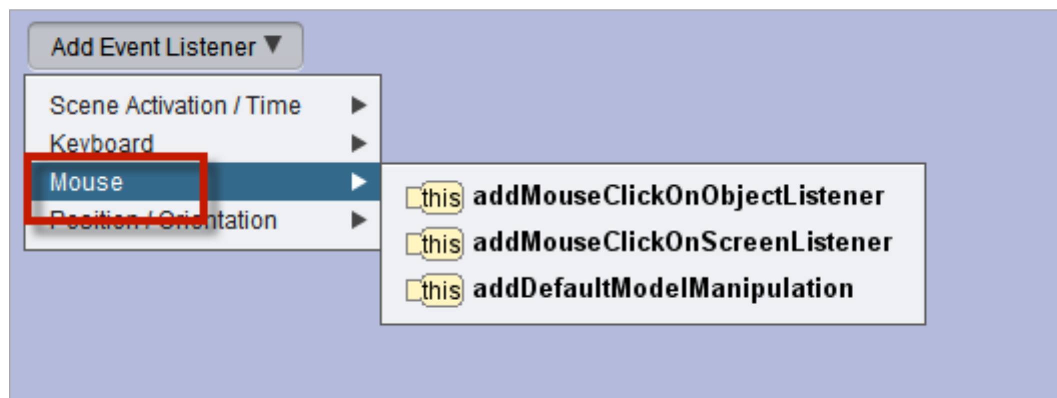


Figure 1 Mouse click event listeners

MOUSE CLICK ON OBJECT

When *addMouseClickOnObjectListener* is selected, a **mouseClicked** event listener is created, as shown in Figure 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 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.getModelAtMouseLocatio**n 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 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.

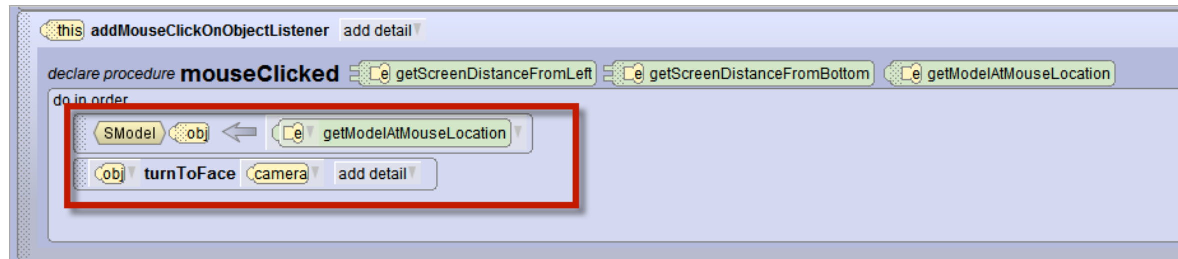


Figure 3 Calling a function to determine which object was clicked

The mouseClicked event listener's add detail button has two options (multipleEventPolicy and setOfVisuals), as shown in Figure 4.

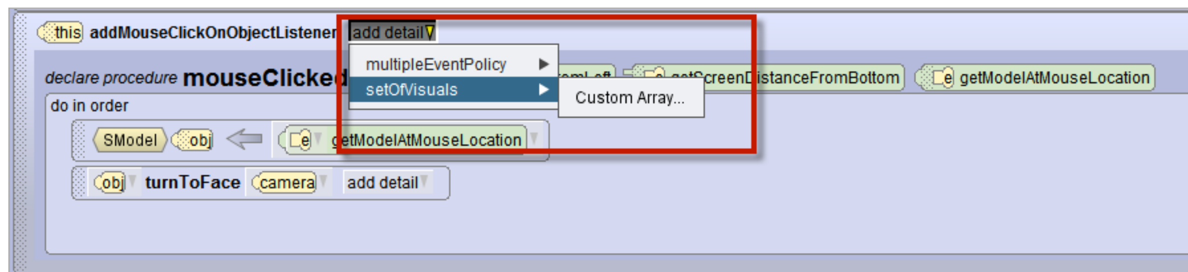


Figure 4 Add detail options

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 5.

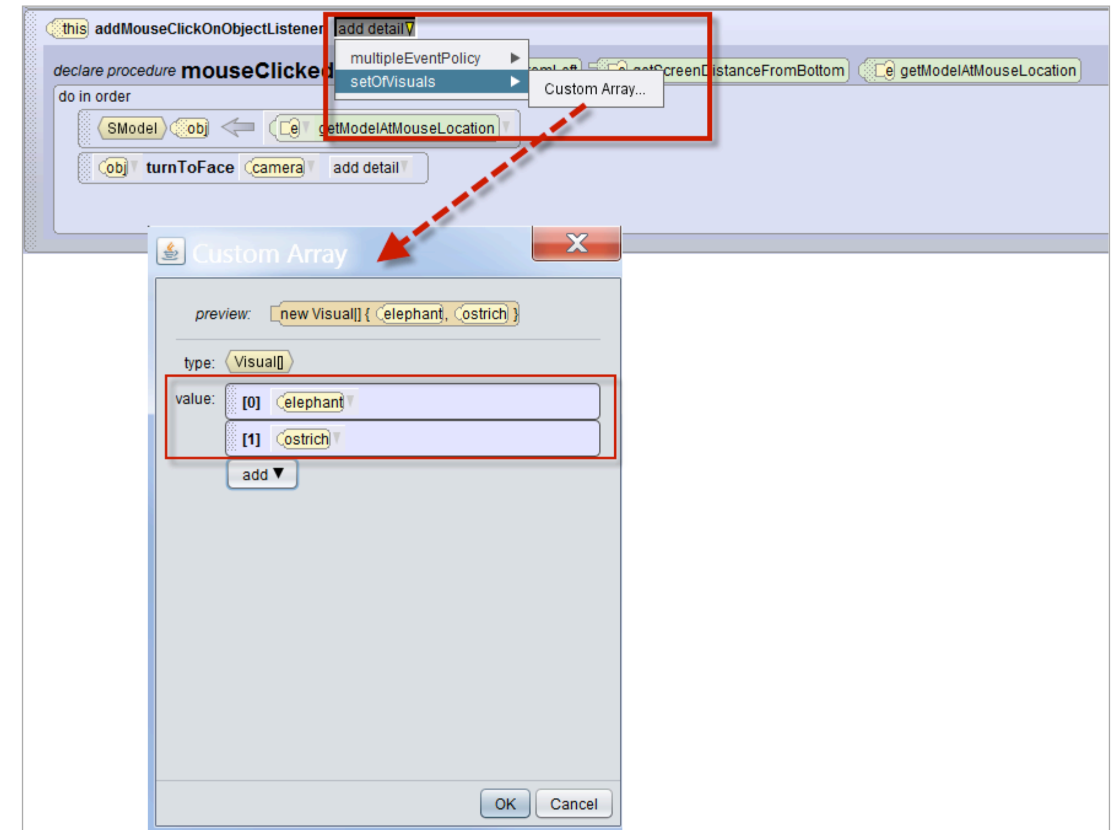


Figure 5 Creating a custom array of objects for a listener

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 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.

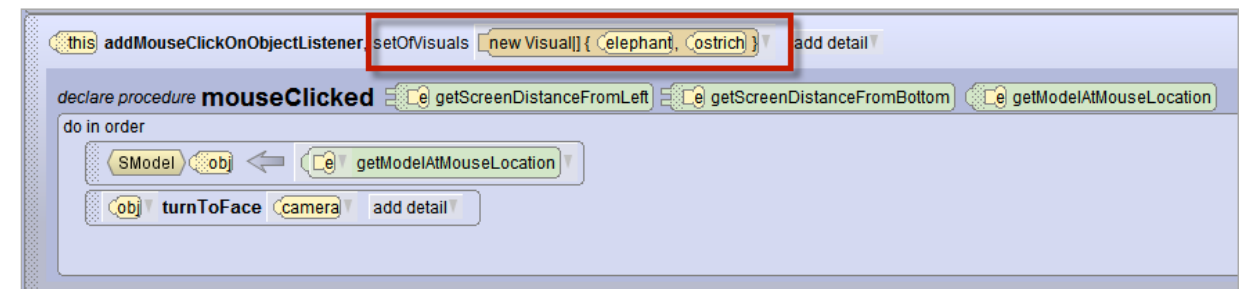


Figure 24.6 The setOfVisuals lists click-able objects

MOUSE CLICK ON SCREEN

When *addMouseClickOnScreenListener* is selected, a **mouseClicked** event listener is created, as shown in Figure 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.

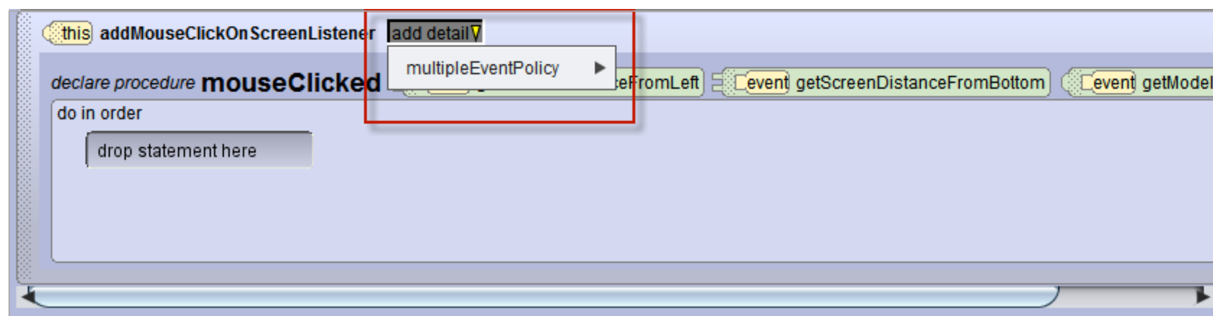


Figure 7 A mouseClicked event for anywhere on the scene

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 8.

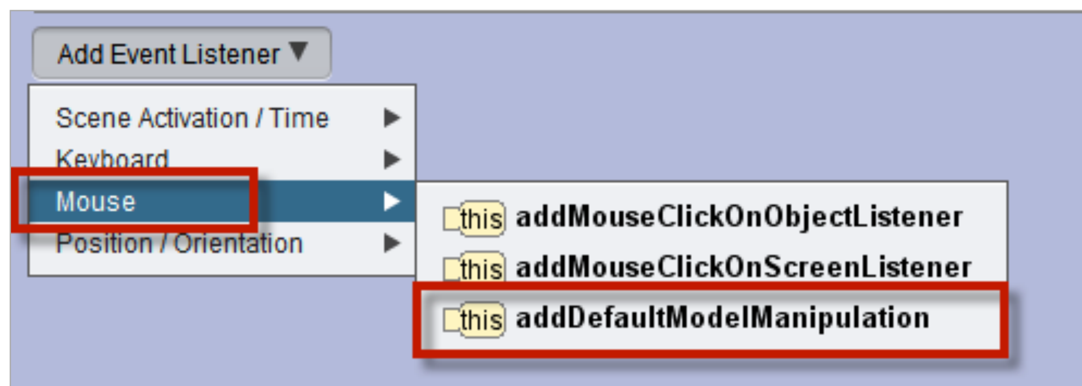


Figure 8 Select Mouse/addDefaultModelManipulation listener

The resulting event listener can be seen in Figure 9.

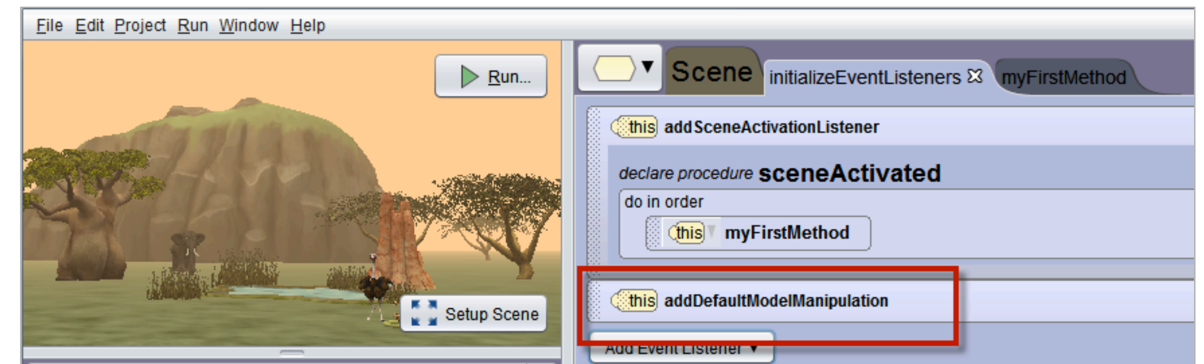


Figure 9 addDefaultModelManipulation event listener code

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 10.

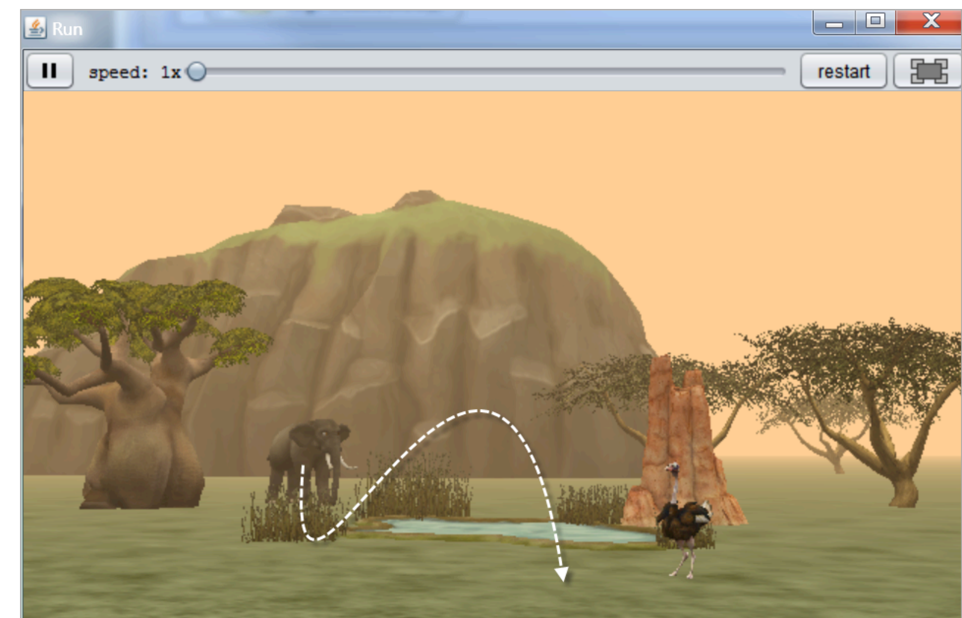


Figure 10 The mouse can move objects around the scene at runtime

SECTION 5

∏

POSITION / ORIENTATION EVENTS

The Position/Orientation events category has nine menu options, as shown in Figure 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.

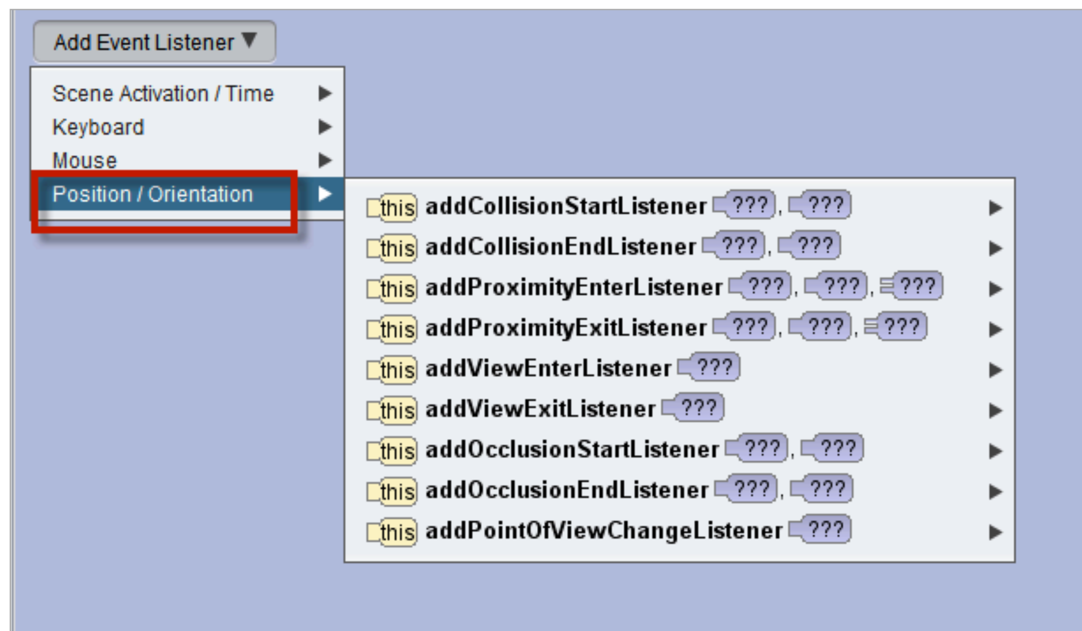


Figure 1 Position and Orientation event listeners

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 2.

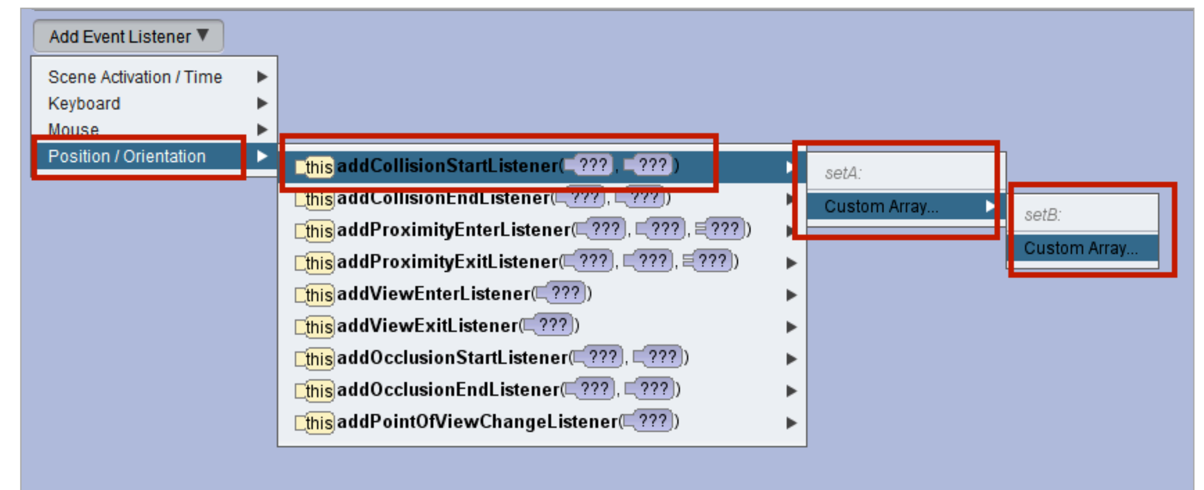


Figure 2 Create a CollisionStartListener

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 2 **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.

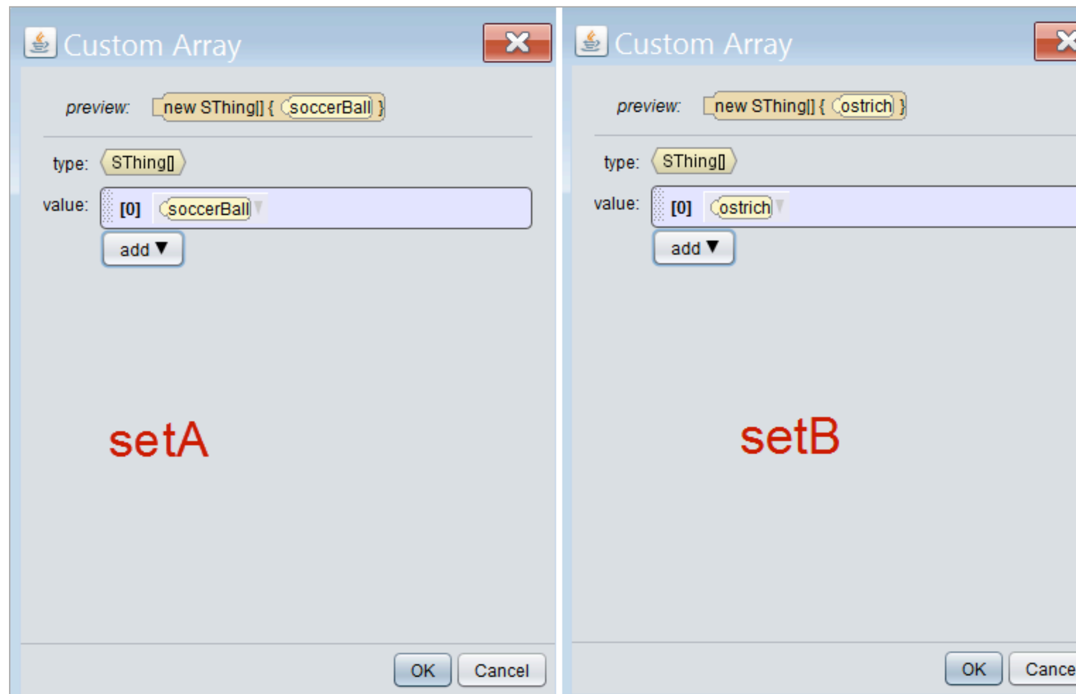


Figure 2 One object in each set, only one possible collision

More complex collision detections often have more than one object in a set, which increases the number of possible collisions. For example, in Figure 3, **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.

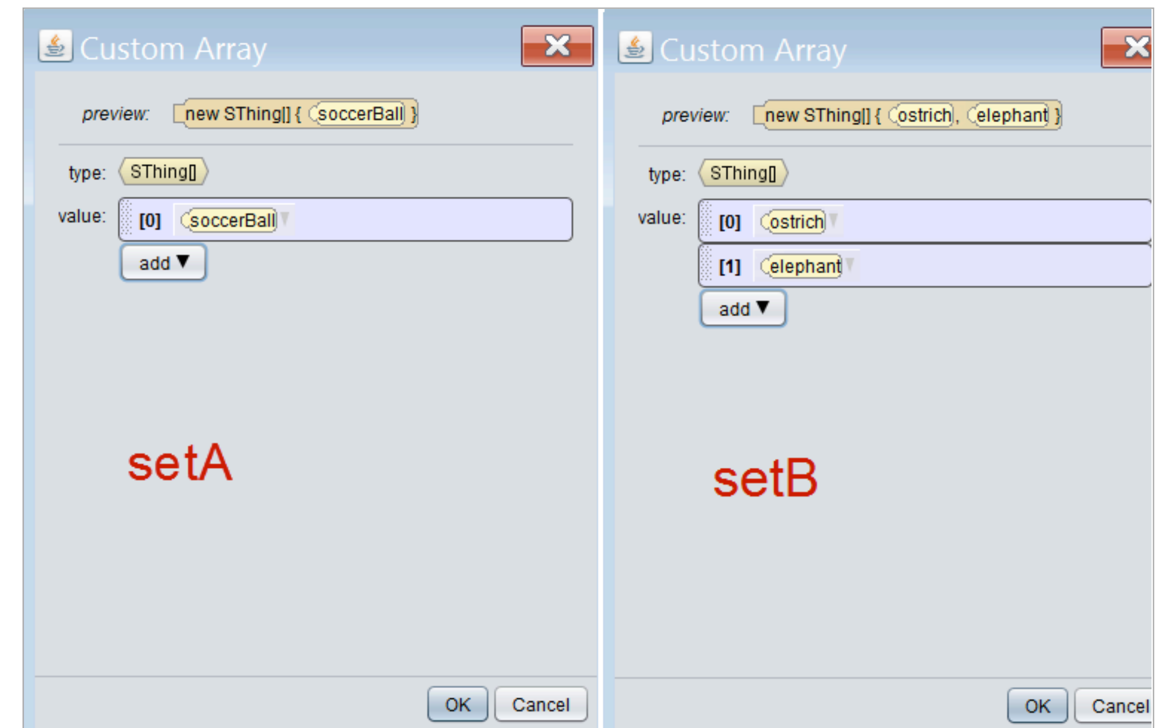


Figure 3 Two possible collisions

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 4 a computed bounding box can be seen surrounding an ostrich.

Figure 4 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 5 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.

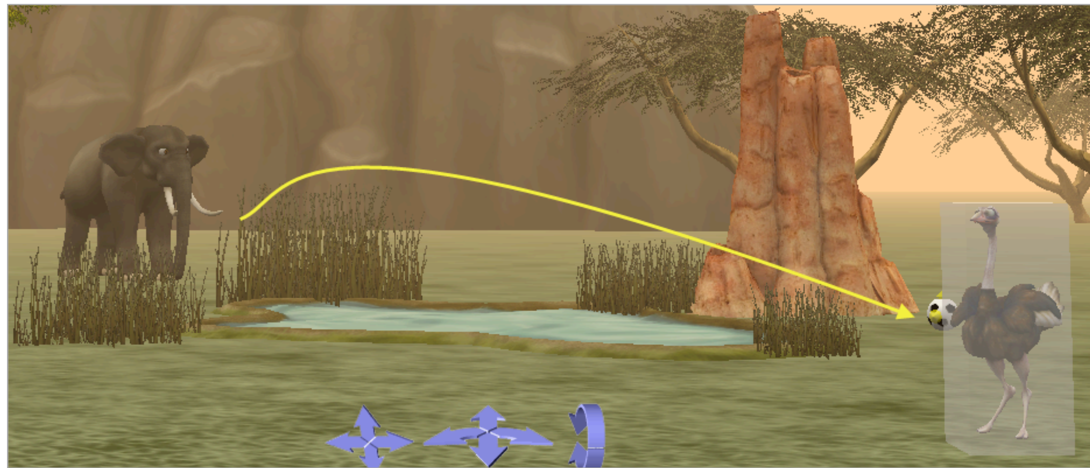


Figure 5 Ball collides with ostrich

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 6. 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.

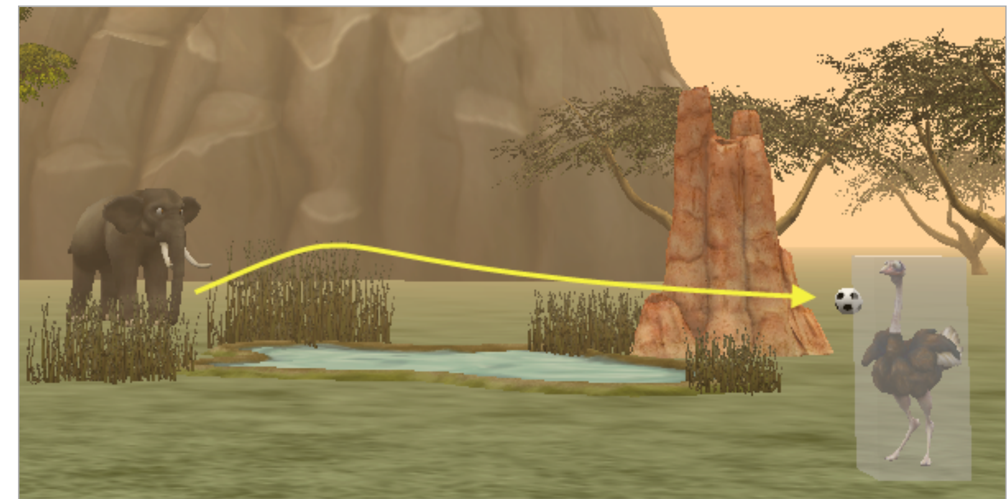


Figure 6 Collision detected, but a little too soon

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 7.

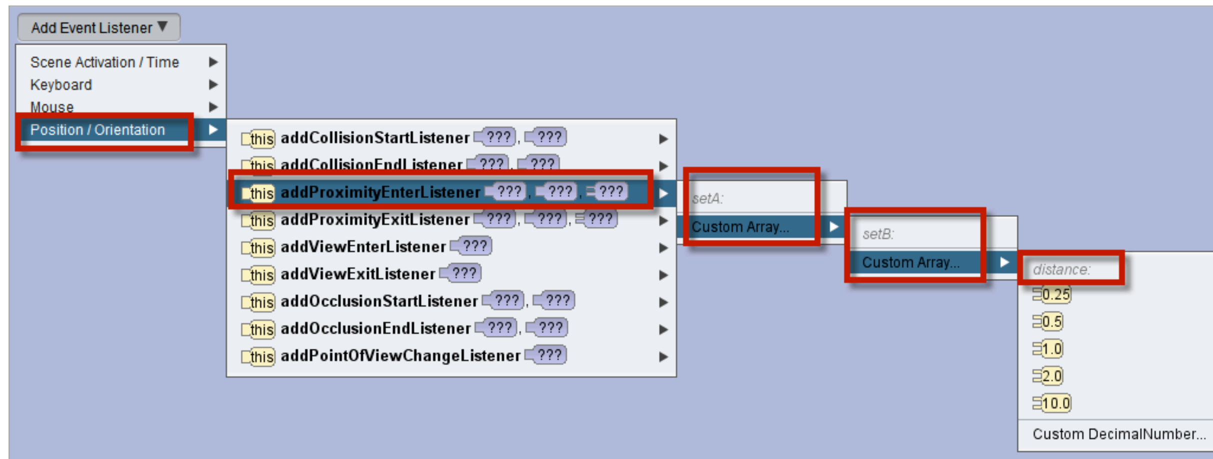


Figure 7 Create a ProximityEnterListener

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 8 **setA** contains elephant and the ostrich and **setB** contains only the pond.

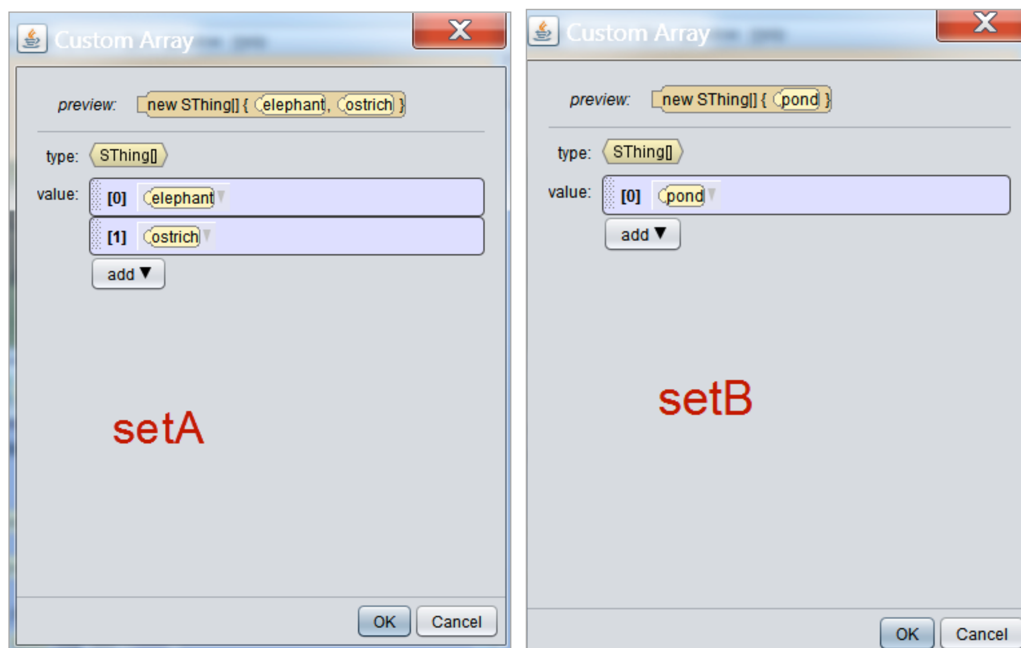


Figure 8 Two sets of objects for proximity enter listener

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 9, 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.

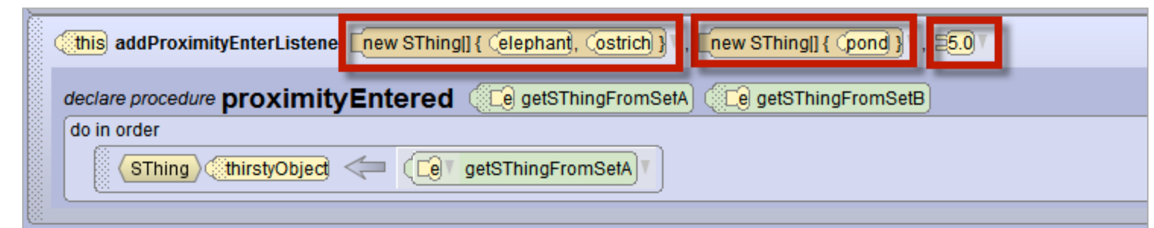


Figure 9 Example: Event listener for proximityEntered

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 10.

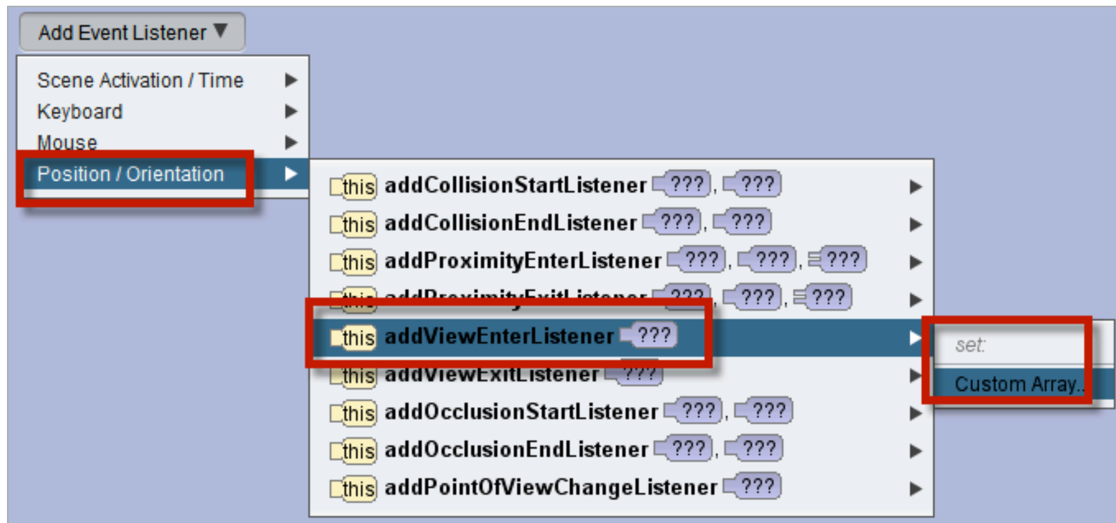


Figure 10 Create a ViewEnterListener

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 11 the set of objects contains elephant and the ostrich.

Figure 11 An array of objects that might enter and leave the scene view

As shown in Figure 12, 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.

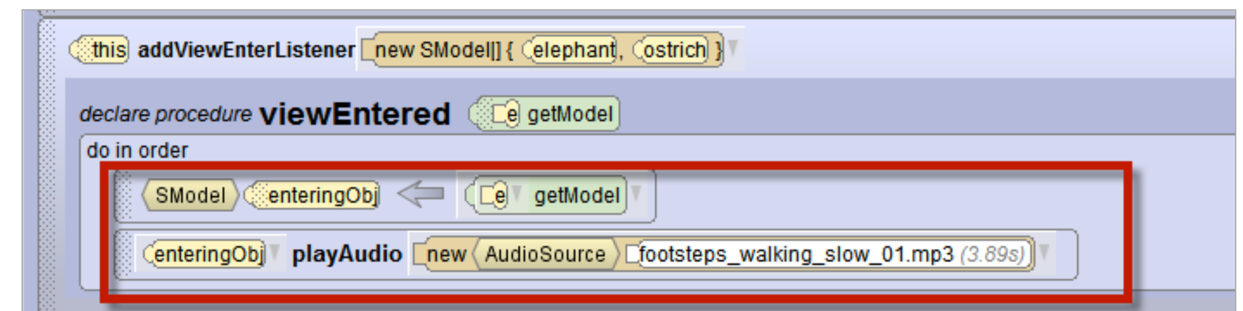


Figure 12 Example: Event listener for viewEntered

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 13.

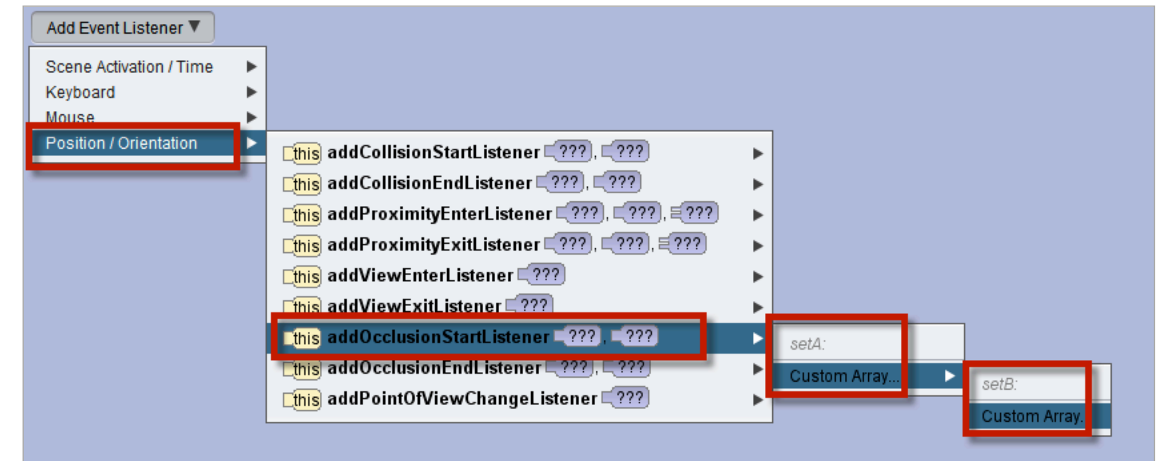


Figure 13 Create an OcclusionStartListener

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 14, **setA** contains the baobabTree and the termiteMound while **setB** contains the elephant and the ostrich.

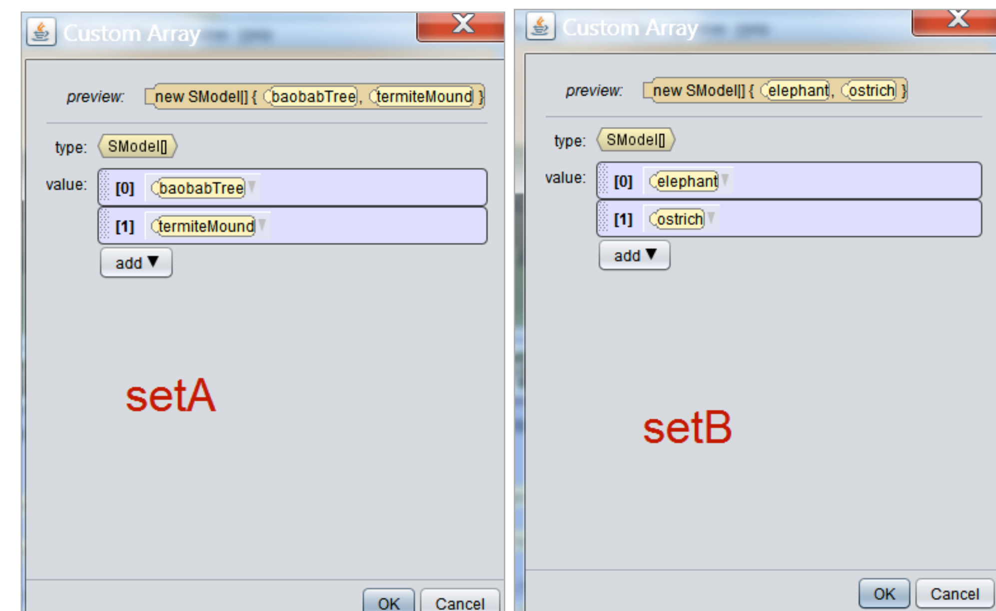


Figure 14 Potential foreground (setA) and background (setB) objects

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 15, 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).

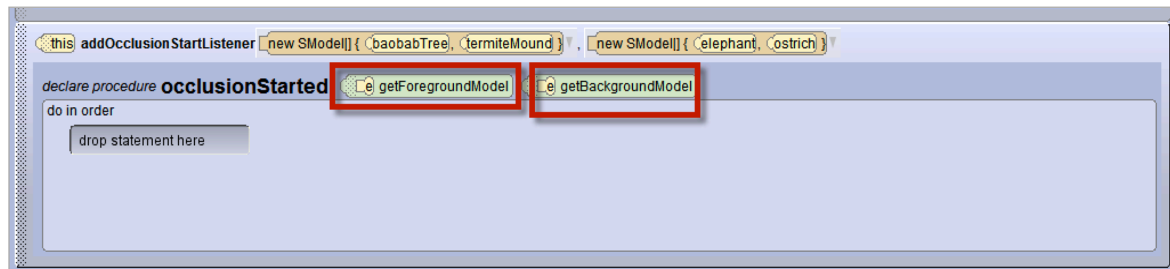


Figure 15 Functions to get foreground and background, when listener fires

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 16.

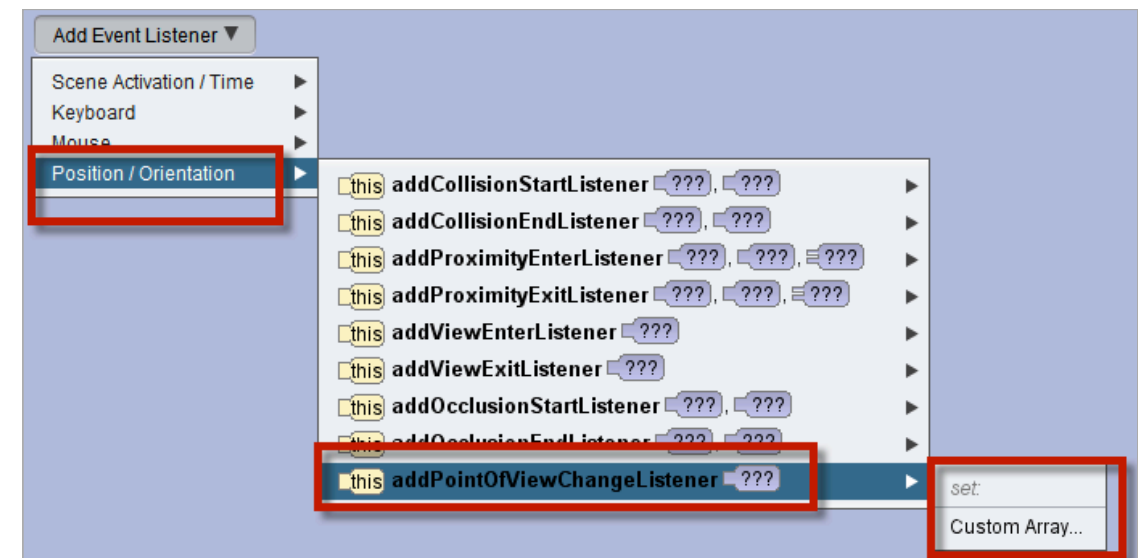


Figure 16 Create PointOfViewChangeListener

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 17, 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.

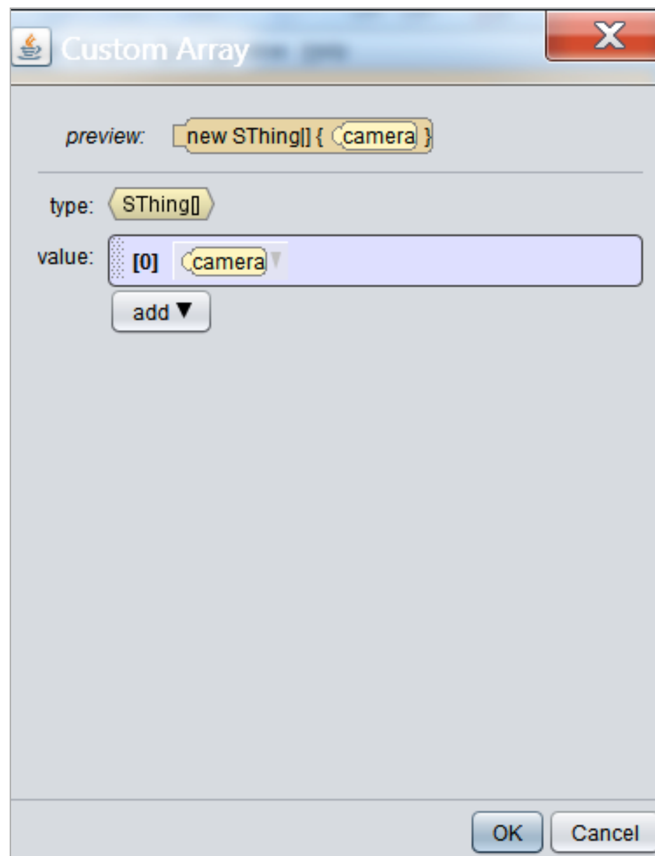


Figure 17 An array containing one element, the camera

In this example, as shown in Figure 18, 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).

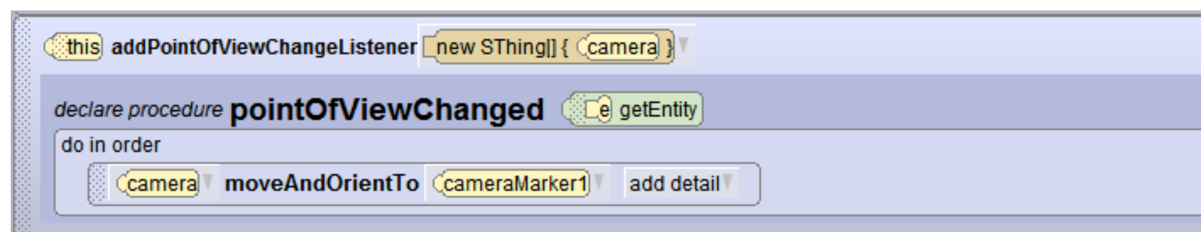


Figure 18 Example: Event listener for point of view change

CHAPTER 5

∏

TRANSFERRING ALICE CODE TO JAVA

To support first time programming students who are learning Java, Alice has several built in features and tools to helpd students move from Alice to a Java programming environment (NetBeans).

These include a Java language option for the Alice IDE, a Java side-by-side window to display the Java representation of the Alice code, and a plugin for the NetBeans development environment that will allow students to import their Alice projects into NetBeans and complete them writing Java.

SECTION 1

∏

VIEW ALICE CODE WITH JAVA SYNTAX

SIDE BY SIDE DISPLAY PREFERENCE

Alice provides a preference setting for a side-by-side panel display of Alice and Java code. To enable the side-by-side display, select **Preferences/Java Code on the Side** from the **Window** menu, as shown in Figure 26.1. An example of side-by-side Alice and Java code display is shown in Figure 26.2.

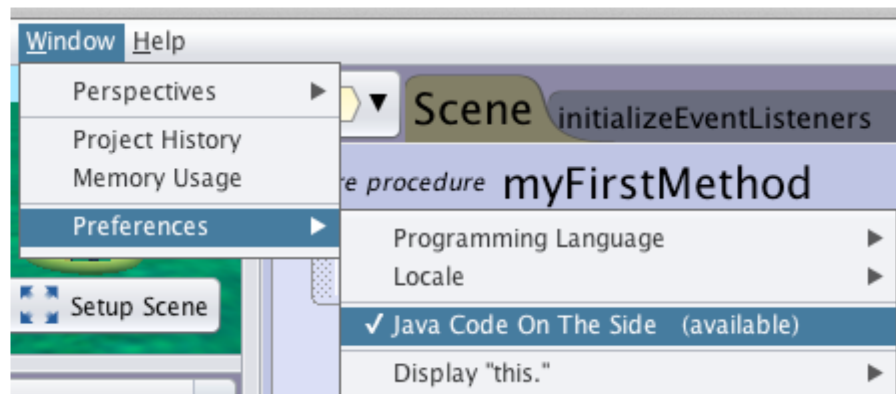


Figure 26.1 Preference setting for Java Code on the Side

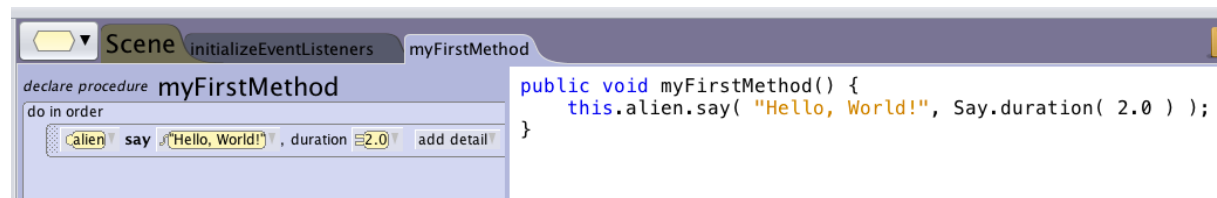


Figure 26.2 Alice and Java side-by-side code panels

In the side-by-side display mode, Alice's drag-and-drop Code editor is fully functional, allowing you to create and modify program code. As changes are made in the Alice code, the Java code display is dynamically updated. The Java code appears very much the same as in any Java IDE, including all the syntactic features (e.g., curly braces for code blocks). In this mode, the Java code **CANNOT** be directly edited using either drag-and-drop or keyboard entry.

JAVA LANGUAGE PREFERENCE

By default, program statements in Alice are displayed using Alice **syntax**. The term syntax refers to the rules of grammar that govern how statements are written. That is, syntax defines the expected ordering of words and punctuation marks for program code. The graphic tiles used in the Alice IDE are used to create program statements with a simple syntax, having a minimal number of quote marks, parentheses, and semicolons.

For those who prefer a "real world" language look and feel, however, Alice provides a preference for changing the IDE display to Java. To change to Java syntax, select **Preferences/Programming Language** in the **Window menu**. Then, in the cascading menu, select **Java**, as shown in Figure 26.3. Program statements in the Code editor will now be displayed with greater fidelity to Java syntax than in the default Alice display.

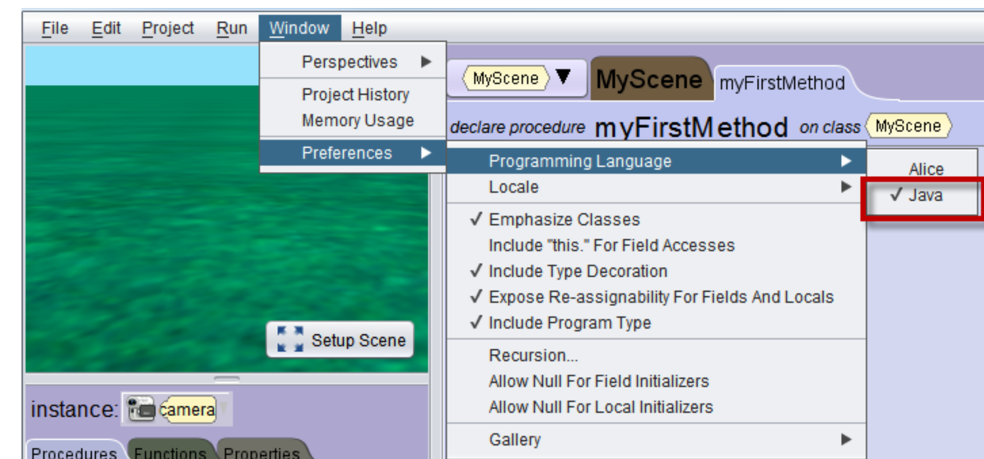


Figure 26.3 Setting the Programming Language Preference to Java

As a comparison, Figures 26.4 and 26.5 show the exact same code in Alice (Figure 26.4) and Java (Figure 26.5). Differences in corresponding statements are highlighted by the green ovals and red boxes in the two figures. The green ovals highlight differences in punctuation marks and the red boxes highlight other differences.

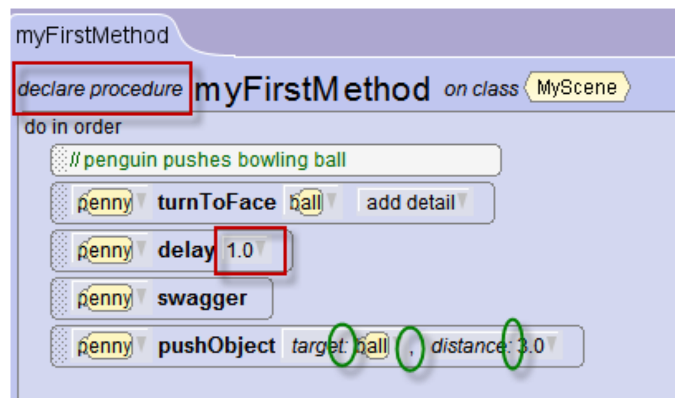


Figure 26.4 Alice code displayed with Alice Programming Language setting

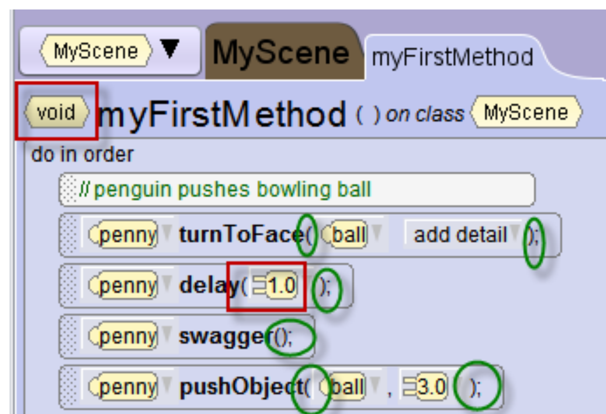


Figure 26.5 Alice code displayed with Java Programming Language setting

In the Java mode, Alice's drag-and-drop Code editor is fully functional, allowing you to create and modify program code. The resulting code is Java code but it **CANNOT** be directly edited using keyboard entry.

SECTION 2

∏

TRANSFER AN ALICE 3 PROJECT TO NETBEANS (JAVA IDE)

For those who want to create an Alice animation project using a text-based editor, Alice provides a plugin for NetBeans, a Java IDE. Using the Alice plugin for NetBeans, you can create a a virtual world in Alice, transfer it to NetBeans, and then create code using keyboard entry.

NOTE: In this section, we assume that Java 8, NetBeans, and the Alice Plugin for NetBeans have been installed on your computer. If you have not already done so, please find instructions for download and installation using the following URLs:

- The Alice Plugin Download
  - http://www.alice.org/index.php?page=downloads/download_alice3.1
- Java 8 and NetBeans Download and Installation
  - http://alice3.pbworks.com/w/page/76386062/java download and installation
- Alice Plugin Installation
  - http://alice3.pbworks.com/w/page/57586346/Download and Install Plugin

STEP 1: SAVE AN ALICE 3 PROJECT

The first step in transferring an Alice project to Java is to save the Alice 3 project before closing Alice. When saving a project, Alice 3 tries to save the file to a default **Alice3/MyProjects** directory, which was automatically created when Alice 3 was installed. It is possible, however, to save the project elsewhere (e.g., on a USB drive or CD). In any case, remember where the project is saved on the computer, as NetBeans will need to locate and open the Alice project during the transfer process.

STEP 2: CREATE A NEW PROJECT IN NETBEANS

Start NetBeans. (If working on a computer that has limited RAM, we recommend closing Alice before opening NetBeans. Alice does not have to remain open during the transfer process.) In NetBeans, select the File menu and New Project, as shown in Figure 27.1.

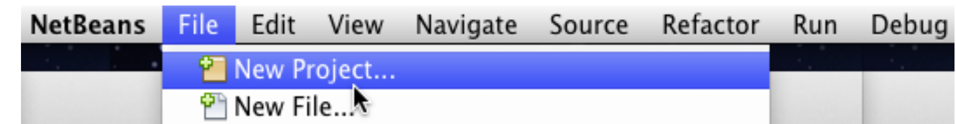


Figure 27.1 Select New Project… from the File menu in NetBeans

Alternatively, on the NetBeans toolbar, click the New Project icon, as shown in Figure 21.5.

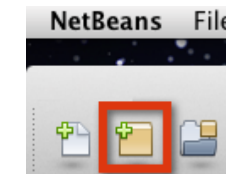


Figure 27.2 New Project icon on the NetBeans toolbar

A New Project dialog box will open, as shown in Figure 27.3. In the **Categories** box, select ***Java***. Then, in the **Projects** box, select ***Java Project from Existing Alice Project***. Then, click on **Next**.

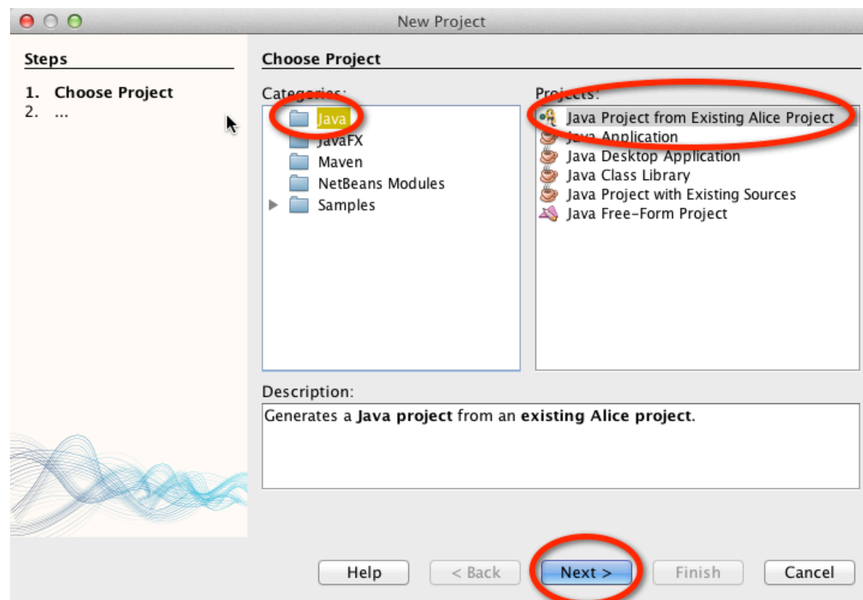


Figure 27.3 New *Project* dialog box

STEP 3: SELECT AN ALICE 3 PROJECT TO IMPORT

A selection box will pop up with a prompt to select an Alice project, as shown in Figure 27.4. Click on the **Browse** button.

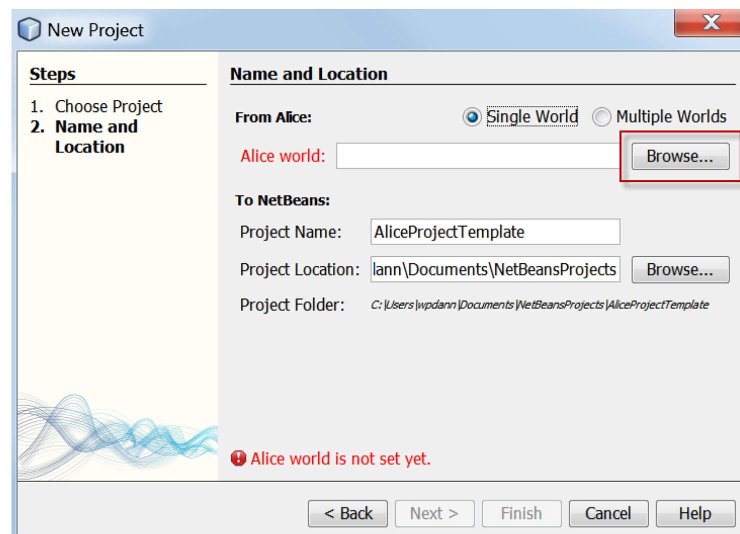


Figure 27.4 Selection box for selecting an Alice world

A navigation box, named ***Select Alice World to Import,*** will open for browsing to the location where the Alice project has been saved. As shown in Figure 27.5, NetBeans assumes that the Alice world will be in the default directory (**Projects** in the **Alice3** directory). If the desired project is in the Projects folder, select the world by a single-click on it and then click the Choose button. In Figure 27.5, we selected an example Alice 3 world named *PenguinBowling.a3p* in the Projects folder. If the desired project is not in the Projects directory but is saved elsewhere on the computer, then use the navigation box to locate the directory in which it was saved.

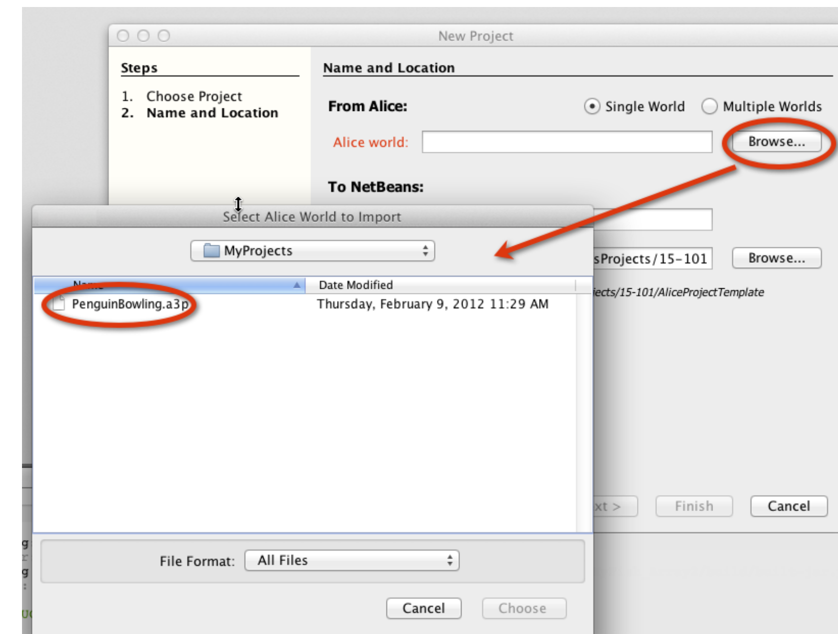


FIGURE 27.5 NAVIGATION BOX TO LOCATE AND CHOOSE THE ALICE PROJECT

Once the desired Alice world has been located and the Choose button clicked, the navigation box closes and focus returns to the selection box.

STEP 4: TRANSFER

When an Alice 3 project has been successfully selected in the New Project dialog box, click the **Finish** button, as shown in Figure 27.6. Patiently wait for the transfer to occur. (Transfer may take as few as 10 seconds or more than 30 seconds, depending on the size of the Alice 3 world.)

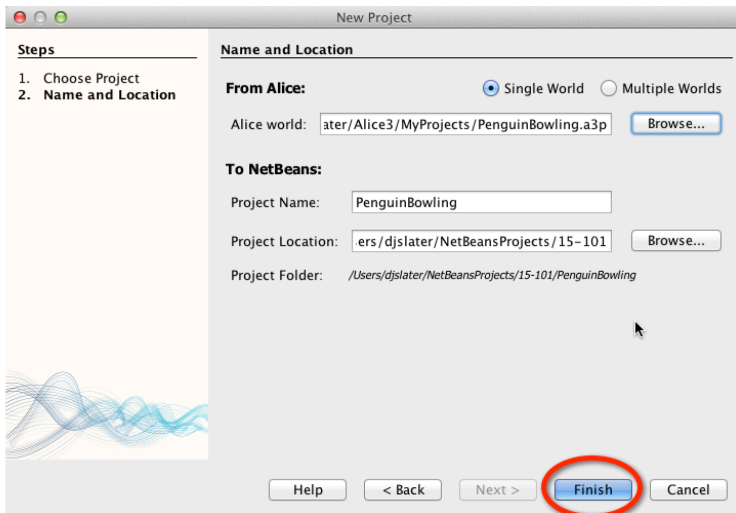


Figure 27.6 Click Finish to begin the transfer

After a successful transfer of an Alice 3 project to a NetBeans project, the new project will be listed in the Projects window in the upper left corner of the NetBeans window, as shown in Figure 27.8. In this example, the **PenguinBowling** project was transferred. Notice, however, that other projects (**SharkAndClownFinshA3Soln** and **PhilosopherSoln**) are also listed in Figure 27.8. As new projects are created in NetBeans, each new project is listed in the NetBeans Projects window along with any previously created projects.

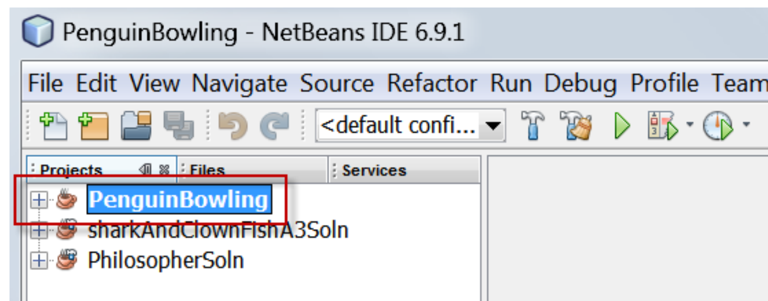


Figure 27.8 A list of projects in NetBeans' Projects window

An Alice project is transferred to NetBeans as a **copy of the original Alice 3 project** and is modified as needed to create a new NetBeans project. We refer to the new project as a **Java-Alice3 project**. The original Alice project file still exists but will *not be affected by any changes made in NetBeans* because the changes are only made to the Java-Alice3 project. Also, a Java-Alice3 project cannot be exported back to Alice.

Once an Alice project is transferred to NetBeans, you can view the list of classes in the project by expanding the Project's Source Package, as shown in Figure 27.9.

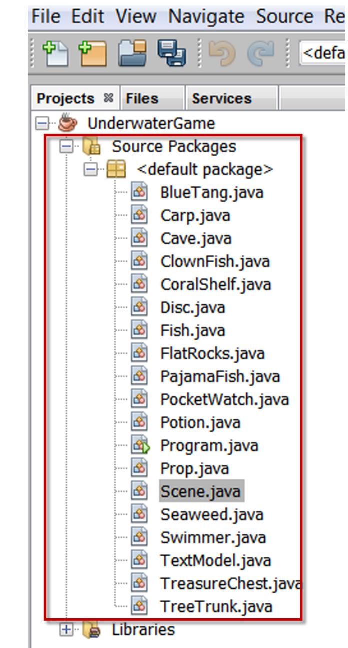


Figure 27.9 Classes in the default Project Source Package

To view the Scene class, double click on Scene.java in the list of classes, as shown in Figure 27.10.

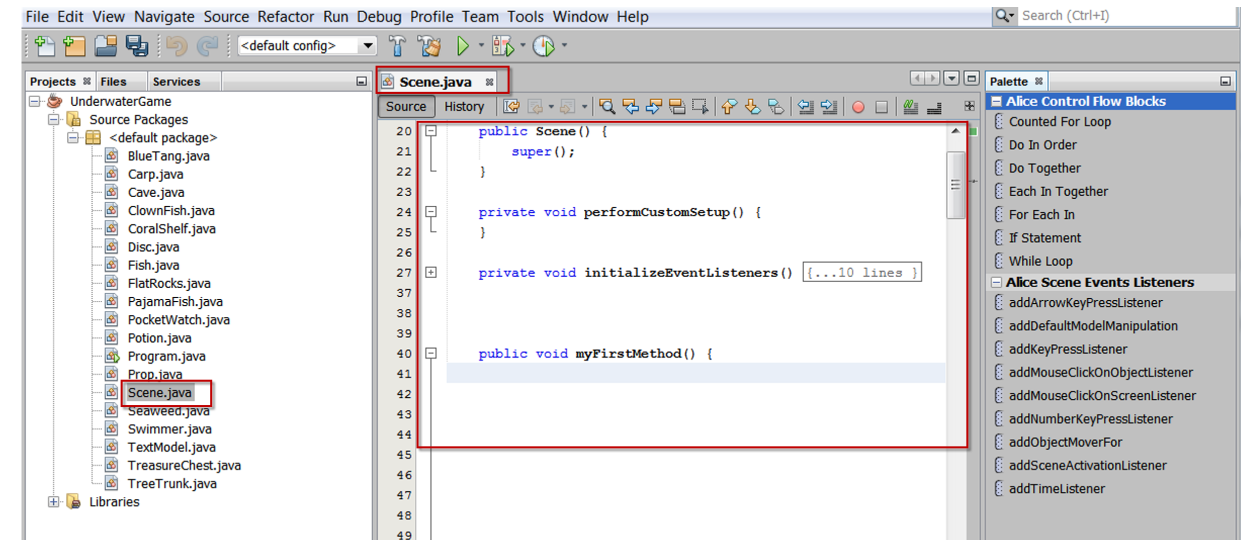


Figure 27.10 View the Scene.java class

On the far right of the IDE is a Palette menu that provides a set of buttons that can be pulled into the editor to create code blocks, as shown in Figure 27.11. The code block contains the appropriate curly braces and a comment,

//TODO: Code goes here.

The intention is to prompt the programmer to replace the TODO comment with appropriate code.

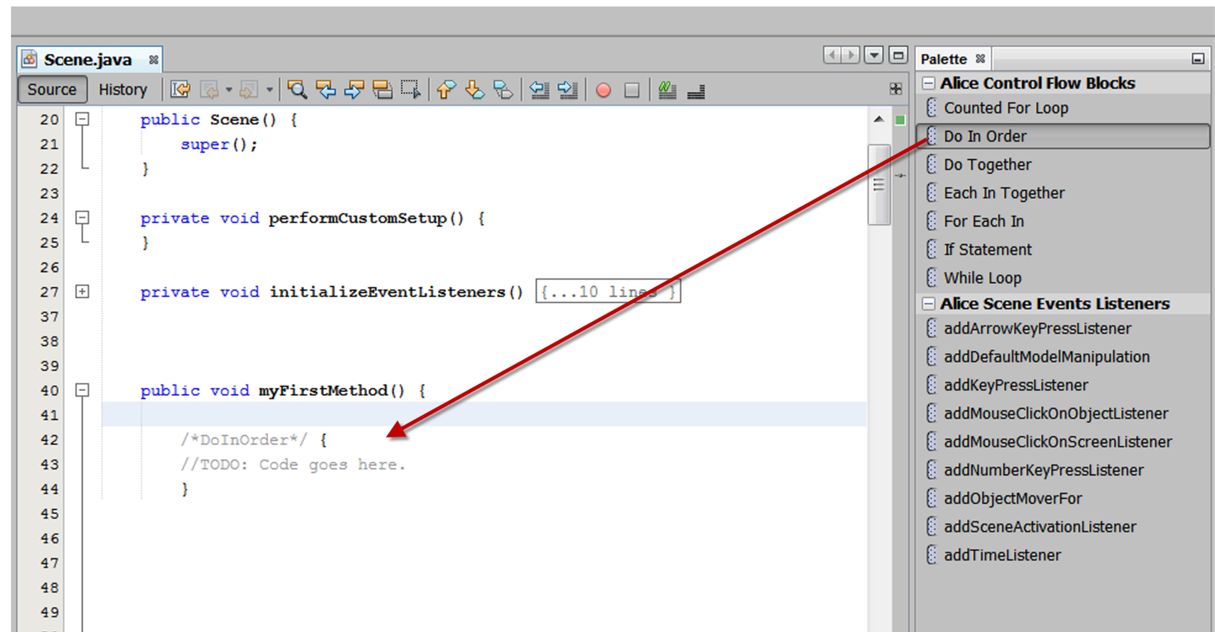


Figure 27.11 Using the Palette to create a code block in the text editor

CHAPTER 6

Appendix

This chapter resents an overview of the methods, functions and properties found in the Alice IDE.

SECTION 1

∏

BUILT-IN METHODS

The Methods panel has two tabs so as to distinguish between procedural methods and functional methods of an object in our animation.

Figure A.1 illustrates the two tabs in a side-by-side listing, using penny (a Penguin object) as an example.

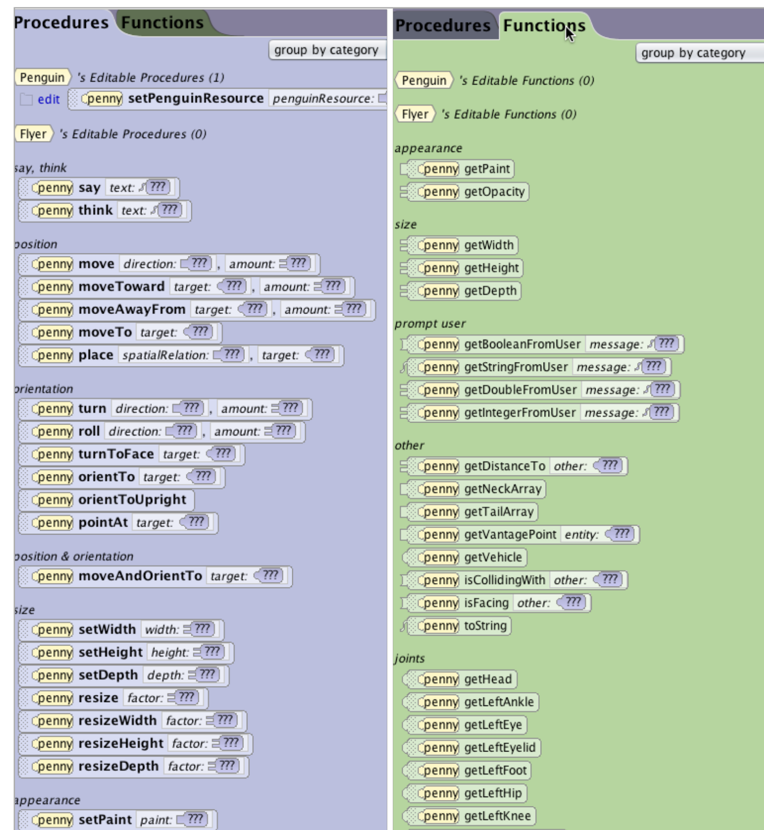


Figure A.1 Side-by-side listing of built-in procedural and functional methods

Important concepts:

Procedural methods describe actions that may be performed by an object, such as move, turn, or roll. These actions often change the location and/or orientation of an object. The important thing to know about procedural methods is that they each perform an action but do not compute and return an answer to a question.

Functional methods are expressions that compute and answer a question about an object such as what is its width or height, or what is its distance from another object.

Properties methods are methods for retrieving (get) and changing (set) specific properties of an object of this class. These specific properties, such as paint, opacity, name, and vehicle, are used in animation rendering.

As a convenient reference, the remainder of this Chapter describes the method tiles commonly found in the Procedures, Functions, and Properties tabs for an object in a scene. The Figures and examples use the alien object, as seen in the screenshot in Figure A.2.

Figure A.2

SECTION 2

∏

PROCEDURAL METHODS

*The Methods panel has two tabs so as to distinguish between procedural and fucntional methods. Procedures are methods that **do** something, in other words, the procedure call results in some action taking place in our animation.*

CHANGE THE SIZE OF AN OBJECT

Every object in Alice has three dimensions, all having a height, width, and depth (even if the value of that dimension is 0.0; e.g., a disc may have a height of 0.0). These procedures change the size of an Alice object, by changing all the dimensions at the same time, proportionately. Procedures that change the value of height, width, or depth are shown in Figure A.3 and summarized in Table A.1.

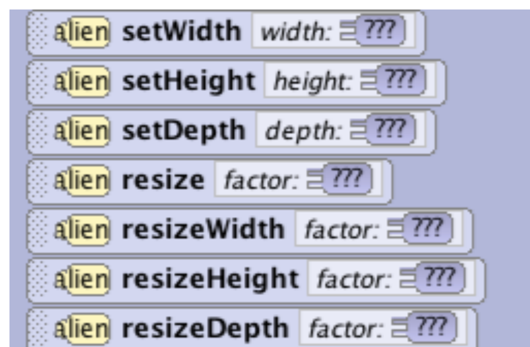


Figure A.3 Procedures that change the size of an object

The set procedures change that dimension to the absolute size provided in the statement. For example if the alien has a height of 1.5 meters, the statement

alien.setHeight height: 2.0

will animate the alien growing to a height of 2.0 meters. The value 2.0 is an argument to the method, to be used as the targeted height.

The resize procedures change a dimension by the factor of the argument value provided in the statement. For example if the same alien has a height of 1.5 meters, the statement

alien.resizeHeight factor: 2.0

will animate the alien growing to the height of 3.0 meters, as the height of the alien is increased by a factor of 2.

Table A.1 Procedures that change the size of an object

Procedure	Argument(s)	Description
setWidth	*DecimalNumber*	Changes the value of the object's width to the value of the argument *width*, with width and depth changed proportionately.
setHeight	*DecimalNumber*	Changes the value of the object's height to the value of the argument *height*, with height and depth changed proportionately.
setDepth	*DecimalNumber*	Changes the value of the object's depth to the value of the argument *depth,* with height and width changed proportionately.
resize	*DecimalNumber*	Changes all the dimensions of the object by the value of the argument *factor*, proportionately
resizeWidth	*DecimalNumber*	Changes the width dimension of the object by the value of the argument *factor*, with height and depth changed proportionately.
resizeHeight	*DecimalNumber*	Changes the height dimension of the object by the value of the argument *factor*, with width and depth changed proportionately.
resizeDepth	*DecimalNumber*	Changes the depth dimension of the object by the value of the argument *factor*, with height and width changed proportionately.

CHANGE THE POSITION OF AN OBJECT IN THE SCENE

Every object in Alice has a specific position and orientation in the scene. Each object can move to its left or right, forward or backward, up or down. Procedures that change an object's position are shown in Figure A. 4 and summarized in Table A.2.

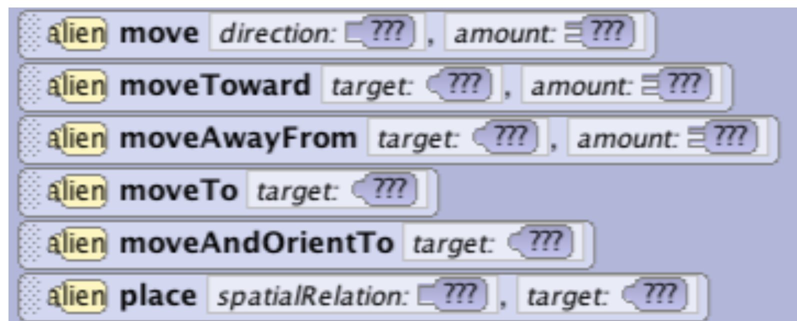


Figure A.4 Procedures that change the position of an object in the scene

Table A.2 Procedures that move an object to a different position in the scene

Procedure	Argument(s)	Description
move	*Direction, DecimalNumber*	Animates movement of the object in the specified *direction* according to its own orientation, by the specified *amount*
moveToward	*Model, DecimalNumber*	Animates movement of the object, by the specified *amount*, in the direction of the *target* object (a 3D Model)
moveAwayFrom	*Model, DecimalNumber*	Animates movement of the object, by the specified *amount*, directly away from the position of the *target* object (a 3D Model)
moveTo	*Model*	Animates movement of the object, in the direction of the *target* object (a 3D Model) until the pivot point of the object and the pivot point of the target are exactly the same; the original orientation of the object is unchanged.
moveAndOrientTo	*Model*	Animates movement in the direction of the *target* object (a 3D Model) until the pivot point of the object and the pivot point of the target are in exactly the same position and the orientation of the object is the same as the orientation of the target object.
place	*spatialRelation:* ABOVE, BELOW, RIGHT_OF, LEFT_OF, IN_FRONT_OF, BEHIND; *Model*	Animates movement of the object, so that it ends up 1 meter from the *target* object (a 3D Model) along the specified *spatialRelation*

CHANGE THE ORIENTATION OF AN OBJECT IN THE SCENE

Every object in Alice has a specific orientation in the scene, with its own sense of forward and backward, left and right, up and down. Importantly, each object has a pivot or center point, around which these

rotations occur. Procedures that change an object's position are shown in Figure A.5 and summarized in Table A.3.

Figure A.5 Procedures that rotate an object

Turn rotations can be LEFT, RIGHT, FORWARD, or BACKWARD. Roll rotations can only be LEFT or RIGHT. The rotations occur in the direction of an object's own orientation, not the camera's point of view and not as seen by the viewer of the animation. For example, if an object is given an instruction to turn LEFT, the object will turn to its own left (which may or may not be the same as left for the person viewing the animation).

The amount of a rotation is always described as a fractional part of a full rotation, expressed as a decimal value. For example, the statement

alien.turn direction: RIGHT, amount: 0.25

will animate the alien turning to its right ¼ of a full rotation, expressed as 0.25. Although a full rotation is 360 degrees and ¼ rotation is 90 degrees, Alice does not use degrees to specify the rotation amount. So, always convert any amount in degrees to a fractional part of a rotation, expressed as a decimal value.

Generally a turn will result in an object's sense of forward changing as the animation occurs, although it may come back to its original orientation if it turns all the way around. A roll will result in an object's sense of up changing as the animation occurs, although it may come back to its original orientation if it rolls all the way around. It may be helpful to note that an object's sense of forward stays the same during a roll.

Table A.3 Procedural methods that rotate an object

Procedure	Argument(s)	Description
turn	*Direction, DecimalNumber*	Animates a turn of an object around its pivot point, in the specified *direction* according to its own orientation, by the specified *amount*, given in fractional parts of a rotation. The object's sense of **forward** will be changing during the animation
roll	*Direction, DecimalNumber*	Animates a roll of the object around its pivot point, in the specified *direction* according to its own orientation, by the specified *amount*, given in fractional parts of a rotation. The object's sense of **forward** will remain unchanged during the animation
turnToFace	*Model*	Animates a turn of the object around its pivot point, so that its sense of forward will be in the direction of the *target* (a 3D Model object)
orientToUpright		Animates a rotation of the object around its pivot point, so that its sense of up will be perpendicular to the *ground*
pointAt	*Model*	Animates a rotation of the object around its pivot point, so that its sense of forward will be in the direction of the *target's* (a 3D Model object) pivot point
orientTo	*Model*	Animates a rotation of the object around its pivot point, so that its orientation will be exactly the same as the orientation of the target (a 3D Model object). The object's position will be unchanged.

Other procedures

Some procedures do not neatly fit into the descriptive categories of the preceding paragraphs. We have collected these procedures into a category called "Other." These procedures provide program output (say, think, playAudio), manage timing in an animation (delay), simplify returning an object to its original position after an animation (straightenOutJoints), and allow one object to be the vehicle for another

object as it moves around the scene (setVehicle). The Other procedures are shown in Figure A.6 and summarized in Table A.4.

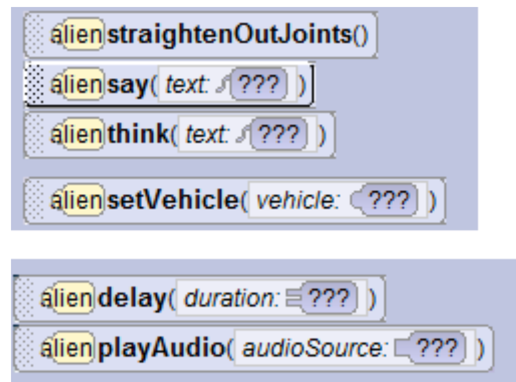


Figure A.6 Other procedures

Table A.4 Other procedures

Procedure	Argument(s)	Description
straightenOutJoints		Restores all the joints of *this* object to their original position, when *this* object was first constructed in the scene editor
say	*textString*	A speech bubble appears in the scene, containing the value of the *text* argument, representing something said by *this* object
think	*textString*	A thought bubble appears in the scene, containing the value of the *text* argument, representing something thought by *this* object
setVehicle	*Model*	Any movement or rotation of the *target* (a 3D Model object) will produce a corresponding movement by *this* object. *This* object cannot be a vehicle for itself, and two objects may not have a reciprocal vehicle relationship (in other words, *this* object cannot be the vehicle of the *target* object, if the *target* object is already the vehicle for *this* object)
delay	*DecimalNumber*	The animation pauses for the length of the *duration* in seconds
playAudio	*??? (sound file)*	The entire imported sound file (either .mp3 or .wav format) will be played in the animation. The length of sound clip that is actually played can be modified in **AudioSource** drop-down menu and selecting ***Custom Audio Source…*** See **Chapter 5: How to…**

Important concepts:

Do in order

When a *delay* action is performed within a *Do in order,* Alice waits the specified number of seconds before proceeding to the next statement. Calling a *delay* on the scene will suspend the animation until the *delay* is complete.

When a *playAudio* action is performed within a *Do in order,* Alice plays the sound for the specified amount of time before proceeding to the next statement.

Do together

When a *delay* action is performed within a *Do together*, other statements within the *Do together* are not affected. However, the *delay* does set a minimum duration for execution of the code block within the *Do together*. For example, in the code block shown below, the alien will move and turn at the same time (duration of 1 second), but Alice will not proceed to the statement following the *Do together* until the *delay* is completed (2 seconds).

```
Do together
     alien.turn direction: RIGHT, amount: 0.25
     alien.delay duration: 2.0
     alien.move direction: FORWARD, amount: 1.00

bunny.turn direction: LEFT, amount: 1.0
```

When a *playAudio* action is performed within a *Do together,* Alice starts plays the sound at the same time as other statements within the *Do together* are executing (for example, as background music).

Use detail parameter options

Most procedures in Alice have a set of parameters with default argument values. These are known as detail parameters. The detail parameters enhance or fine tune the animation action performed when a statement is executed.

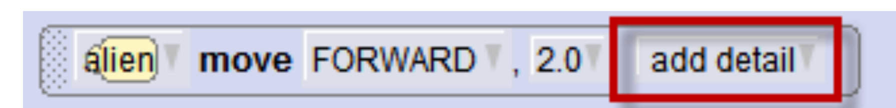


Figure A.7

The three most common detail parameters are asSeenBy, duration, and animationStyle. There are a few procedures that may not use all of these details, or they may have a different set of details, appropriate for that particular animation. Table A.5 summarizes the detail parameter options.

Table A.5 Details

Detail	Values	Description
asSeenBy	*Model*	The movement or rotational animation of this object will be as if *this* object had the pivot point position and orientation of the *target* object
duration	*DecimalNumber*	By default, Alice animation methods execute in 1 second. This modifier changes the duration value to a specified length of time.
animationStyle	*BEGIN_AND_END_ABRUPTLY* *BEGIN_GENTLY_AND_END_ABRUPTLY* *BEGIN_ABRUPTLY_AND_END_GENTLY* *BEGIN_AND_END_GENTLY*	The default animation style is *BEGIN_AND_END_GENTLY,* which begins with a reasonable period of acceleration, then constant movement at some top speed, followed by a reasonable period of deceleration. Other animation styles: *BEGIN_GENTLY_AND_END_ABRUPTLY* begins with a *gradual* acceleration to top speed and ends with a sudden

SECTION 3

∏

FUNCTIONAL METHODS

The Methods panel has two tabs so as to distinguish between procedural and functional methods. Functions are methods that ***answer questions*** *about an object and its relatonship to itself, to other objects in the animation, or to the animation itself. The function call will not produce an action taking in the animation.*

FUNCTIONS THAT ACCESS AN INTERNAL JOINT OF AN OBJECT

The internal joints of an object are part of a skeletal system. For this reason, a function is called to access an individual joint within the skeletal system. These functions return a link to the joint (similar to a link that holds the address of a web page on the web).

As an example, some of the functions to access the individual joints of an alien object are illustrated in Figure A.8 accompanied by an X-ray view of the alien's internal joints.

NOTE: Due to page space limitations, not all the alien's joint access functions are listed here.

Figure A.8 Functions that link to an internal joint of an object

The link returned by calling one of these functions provides access to the specified joint of the object, for example, if in an animation we wanted a ball to move to the alien's right hand in a game of catch with another alien, we could write the instruction statement:

ball.moveTo target: alien.getRightHand

It should be noted that these functions are dependent upon the design in the 3D Model for which the object is constructed. For example, all Bipeds have the same basic set of joints, as shown in the X-ray view of the alien and hare in Figure A.9.

Figure A.9 X-ray view of alien and hare internal joints

Although the alien and the hare have the same basic set of Biped joints, the alien also has a set of finger joints that are particular to the Alien class and the hare has a set of joints in its ears that are specific to the Hare class. These commonalities and differences are reflected in the functional methods that get access to an internal joint, as shown in Figure A.10.

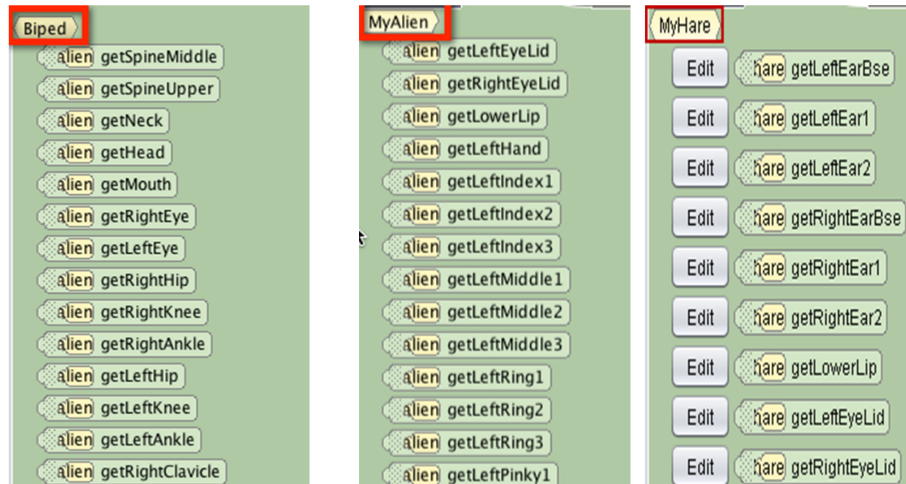


Figure A.10 Common and specific functional methods for joint access

GETTERS: FUNCTIONS THAT RETURN THE SIZE VALUES OF AN OBJECT

The term "getter" is used to describe a function that returns the current value of a property. In Alice the three dimension (width, height, and depth) properties are of special importance and have their own getter functions. These getter functions for the alien are shown in Figure A.11. Table A.6 summarizes these functions.

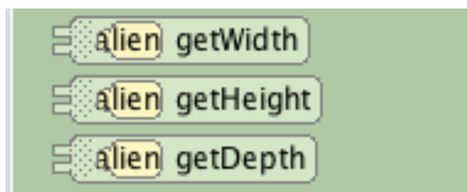


Figure A.11 Functions that return dimension property values

Table A.6 Functions that return dimension values

Function	Return type	Description
getWidth	*DecimalNumber*	Returns the width (left to right dimension) of *this* object
getHeight	*DecimalNumber*	Returns the height (bottom to top) dimension of *this* object
getDepth	*DecimalNumber*	Returns the depth (front to back) dimension of *this* object

OTHER FUNCTIONS

Some functions do not neatly fit into the descriptive categories of the preceding paragraphs. We have collected these functions into a category called "Other." Other functions are shown in Figure A.12 and summarized in Table A.7.

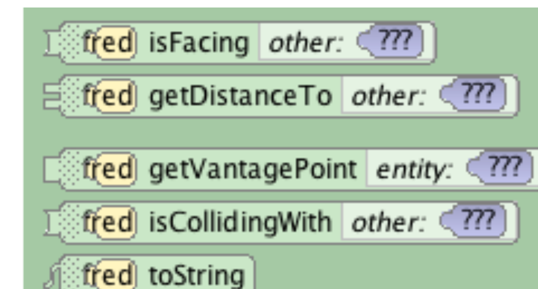


Figure A.12 Other functional methods

Table A.7 Other functional methods

Function	Return type	Arguments	Description
isFacing	*Boolean*	*Model*	Returns **true** if *this* object is facing the *other* (a 3D Model object) or else returns **false**
getDistanceTo	*DecimalNumber*	*Model*	Returns the distance from the center point of *this* object to the center point of the *other* (a 3D Model object)
getVantagePoint	*???*	*entity*	TO BE IMPLEMENTED. Returns the point of view of *this* object
isCollidingWith	*Boolean*	*Model*	returns true if the bounding box of *this* object intersects in any with the bounding box of the other (3D Model object), false otherwise
toString	*TextString*		NOTE: THIS DOES NOT RETURN THE IDENTIFIER NAME OF THIS OBJECT IN PROGRAM CODE, but the internal identifier used by Alice in the virtual machine

Functions for User Input

Functions that ask the user to use the keyboard or mouse to enter a value (of a specific type) are provided for all objects in an Alice scene. The value entered by the user is returned by the function, to be stored in a variable or used as an argument in a call to another procedure or function. Functions for User Input are shown in Figure A.13.

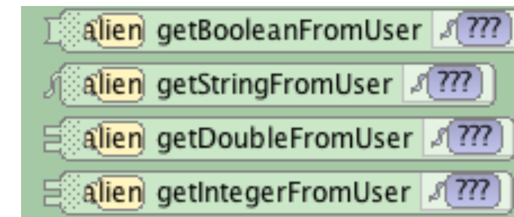


Figure A.13 Functions for User Input

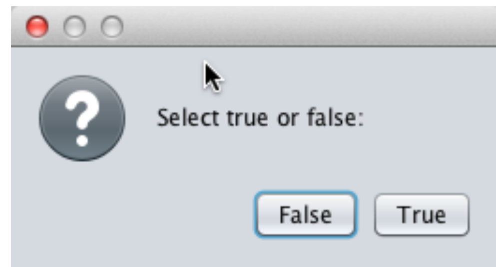
When a user input function is called, at runtime, a dialog box is displayed containing a prompt to ask the user to enter a value of a specific type. Of course, Alice does not automatically know what prompt to use in the dialog box. The programmer supplies a prompt that will be displayed. The prompt is the argument to the function within an instruction statement.

An important part of calling a function to get user input is that the value the user enters is expected to be of a specific type. For example, the value the user enters when the getIntegerFromUser function is called must be a whole number, not a number containing a decimal or a fraction. Likewise, a variable or a parameter that receives the returned value must be a compatible type with the type of value being returned by the function. For example, if the user enters a String of alphabetic characters, the String cannot be stored in an Integer variable. For this reason, Alice will continue to display the user input dialog box until the user enters a value of the right type.

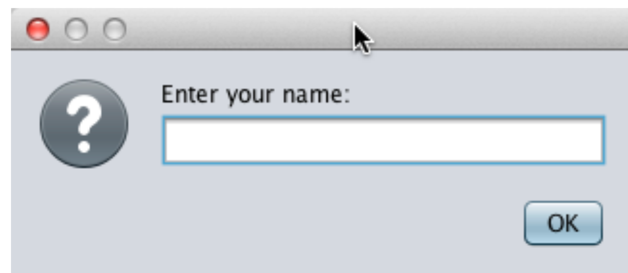
To each row of Table A.8, we have attached an image depicting a sample dialog box containing a prompt appropriate as an argument for calling that function.

Table A.8 User input Table

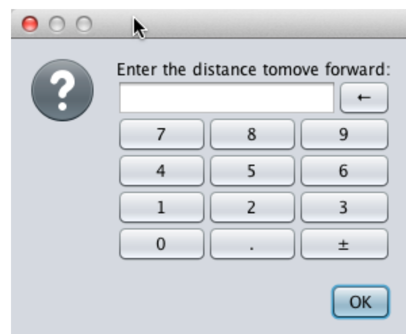
Function	Return type	Argument	Description
getBooleanFromUser	*Boolean*	*TextString*	Displays the dialog box with the TextString argument displayed as the prompt and True and False buttons for user input.



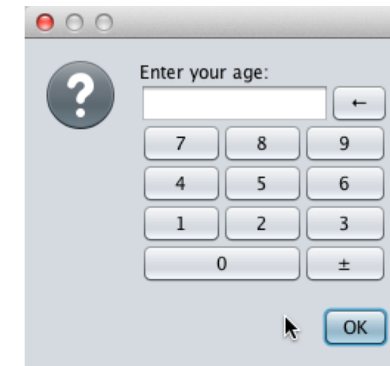
getStringFromUser	*TextString*	*TextString*	Displays the dialog box with the TextString argument displayed as the prompt and a textbox for user input.



getDoubleFromUser	*DecimalNumber*	*TextString*	Displays the dialog box with the TextString argument displayed as the prompt and a keypad (with a decimal point) for user input.



getIntegerFromUser	*WholeNumber*	*TextString*	Displays the dialog box with the TextString argument displayed as the prompt and a keypad for user input.



SECTION 4

∏

PROPERTY METHODS

There are a collection of methods that manipulate and access the given properties of a class. The procedural methods (found under the Procedures tab) change or **set** *the given properties , and the functional methods (found under the Functions tab) will* **get** *the current values of those properties.*

Setter is a specialized term used to describe a procedure that changes the value of an object's property. **Getter** is a specialized term used to describe a function that returns the current value of an object's property. Currently in Alice 3, most setters and getters can be found in the Procedures and Functions tabs of the Methods Panel. (For example, setVehicle is in the Procedures tab, and getWidth is in the Functions tab.)

Some properties, however, are general purpose in that they are defined for the purpose of rendering an object in the scene. Getters and setters for these properties are conveniently listed in the Properties tab of the Methods panel. For example, the alien's setters and getters are shown in Figure A.14 and summarized in Table A.9.

Figure A.14 Getters and setters for specialized properties

Table A.9 Setters and Getters for specialized properties

Procedure	Argument(s)	Description
setPaint	*paint*	Sets the paint value of *this* object to the *paint* argument
setOpacity	*opacity*	Used to set the transparency of *this* object by setting the opacity value of *this* object using a range of values from **0.0** (invisible) to **1.0** (fully opaque).
setName	*name*	NOTE: THIS DOES NOT CHANGE THE IDENTIFIER NAME OF THIS OBJECT IN PROGRAM CODE, but does change the internal identifier used by Alice for debugging purposes.

Function	Return type	Description
getPaint	*paint*	Returns the paint value of *this* object
getOpacity	*DecimalNumber*	Returns the opacity value in the range of **0.0** (invisible) to **1.0** (fully opaque).of *this* object
getName	*TextString*	NOTE: THIS DOES NOT RETURN THE IDENTIFIER NAME OF THIS OBJECT IN PROGRAM CODE, but the internal identifier used by Alice in the virtual machine.
getVehicle	*Model*	Returns a link to another object in the scene that is serving as the vehicle for *this* object

METHODS THAT CAN BE CALLED ON AN OBJECT'S INTERNAL JOINTS

Procedures

As described previously in Chapter 3, almost all 3D model classes in the Gallery have a system of internal joints. The joints can be thought of as the pivot points of sub-parts of the object and can be used in the Scene editor to position sub-parts during scene setup. An object's joints are also objects, and program statements can be written to animate an object's sub-parts by rotating and orienting an object's internal joints. Procedures that can be used to animate joints are shown in Figure A.15.

Figure A.15 Procedural methods for an object's internal joints

These procedures perform the same actions that were described for the entire object, but the pivot point is at the joint. For example, a statement can be created to tell the alien to turn its right shoulder joint backward, as shown in Figure A.16. As the right shoulder joint turns, the right upper arm, lower arm, and hand also turn. That is, the arm parts are attached to the body through the shoulder joint. For this reason, the arms parts turn when the joint turns.

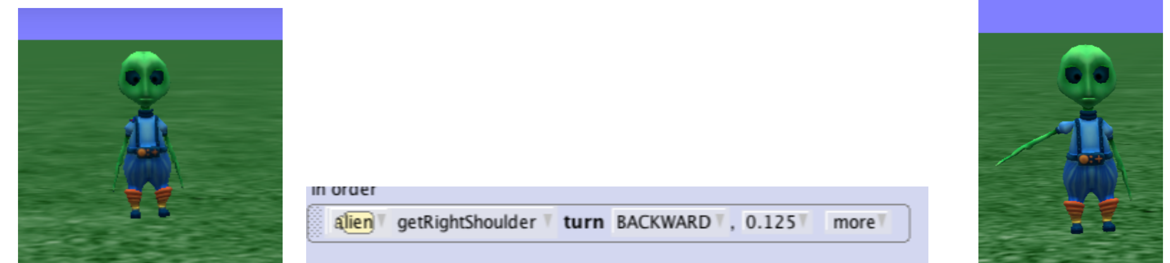


Figure A.16 A statement to turn the alien's right shoulder joint

Notice that the procedures in Figure A.15 do not include methods that move the joint. In Alice 3, a joint cannot be moved out of its normal position within the skeletal structure of the object's body. In other words, a joint and its attached sub-part(s) cannot be separated from the body.

The only unique procedure for joints is setPivotVisible, as described in Table A.10.

Table A.10 Procedure specific to internal joints

Procedure	Argument(s)	Description
setPivotVisible	**true** or **false**	Displays the pivot position and orientation of *this* joint in the animation if the argument is true, hides the pivot position and orientation of *this* joint in the animation if the argument is false

Functions

Almost all functional methods for an entire object are functions that access (return a link to) one of the joints belonging to that object. However, there are only a few functions that can be called on an individual joint, as shown in Figure A.17.

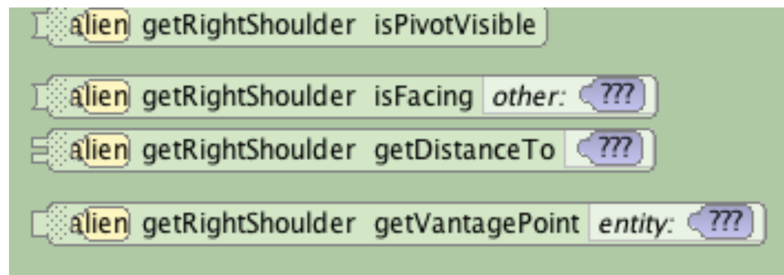


Figure A.17 Functional methods for a joint

The available functional methods have the same name and perform the same actions as the functions of the same name for the entire object. Refer back to Table A.7 for the descriptions of these methods. The only function that is unique to joints is the isPivotVisible function, as summarized in Table A.11.

Table A.11 A unique function for internal joints

Function	Return Type	Description
isPivotVisible	*Boolean*	Returns **true** if the pivot position and orientation of *this* joint in the animation is being displayed, or else returns **false** if the pivot position and orientation of *this* joint in the animation is not being displayed

Properties

All of the available getters and setters on the Properties tab/Methods panel of an object's internal joints are the same as the getters and setters of the same name for the entire object, as shown in Figure A.18. Refer to Table A.9 for descriptions of these specialized methods.

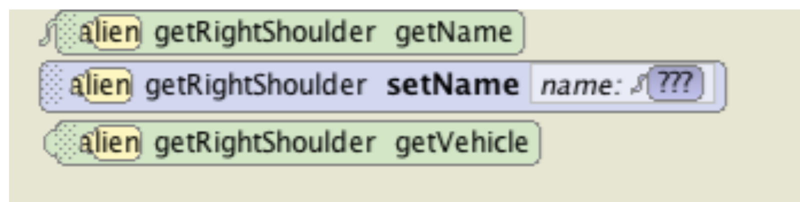


Figure A.18 Properties methods for internal joints

METHODS FOR STANDARD OBJECTS

Every Alice project has a scene (this) that is an instance of the Scene class and contains two other standard objects: the ground or water surface (an instance of the Ground class), and the camera (an instance of the Camera class), as shown in Figure A.19. Each of these objects has their own procedures, functions, and properties, as defined in their respective classes.

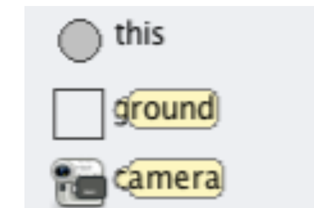


Figure A.19 The standard components of every Alice project

The Scene class has a few procedures, functions and property methods that are exactly the same as in other classes, as shown in Figures A.20 A. 21, and A.22. See previous descriptions of these procedures (Table A-4), functions (Table A-7), and properties (Table A-9) earlier in this Appendix.

Figure A.20 Procedural methods in common with other classes

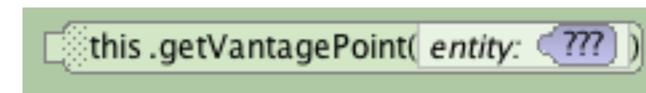


Figure A.21 Functional methods in common with other classes

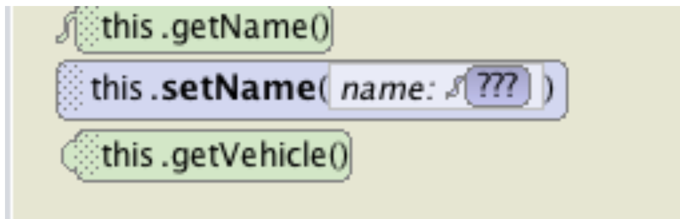


Figure A.22 Properties methods in common with other classes

The scene is truly the "universe" of an Alice 3 project because it provides the stage, the actors, and the scenery for animation. For this reason, a scene object has need of many special methods that perform unique operations for creating the scene and animating the characters in the story or game. Unique procedures that are used for setting up a scene and managing the animation are shown in Figure A.23.

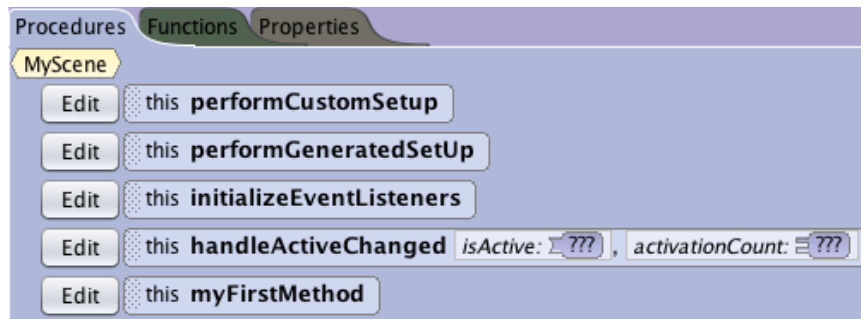


Figure A.23 Unique procedural methods defined in Scene

The Alice environment automatically calls the performGeneratedSetUp, performCustomSetup, and initializeEventListeners procedures (in order) when the user clicks on the Run button. The performGeneratedSetUp procedure contains instructions that were automatically "recorded" as objects were created and arranged in the Scene editor. When performGeneratedSetUp is executed, these instructions are used by the Alice system to re-create the scene in the runtime window. The performCustomSetup procedure contains instructions that may have been written to adjust the scene in a way not available in the Scene editor. The initializeEventListeners procedure contains instructions to start listeners for events such as key presses and mouse clicks while the animation is running. (Specific events and listeners are described below in the Scene Listeners section.)

After these three procedures are executed, the scene's myFirstMethod is called and the animation code in the project is executed. Table A.12 provides further information regarding these unique procedural methods.

Table A.12 Procedures for this scene

Procedure	Argument(s)	Description
performCustomSetup		Allows the programmer to make adjustments to the starting scene; adjustments that could not be easily made in the Scene Editor. Add program statements to this procedure as is done in any method in Alice. However, all statements here will be executed after the ***Run…*** button is clicked, but before the runtime window is displayed
performGeneratedSetup		When the ***Run…*** button is clicked, Alice inspects the scene built in the Scene Editor and generates the appropriate code necessary to display the scene created by the user in the runtime window. **NOTE: the programmer should not attempt to add or modify code in this procedure, as it is always rewritten whenever the *Run…* button is clicked.**
initializeEventListeners		This procedure of the **Scene** class is the preferred location in an Alice project for the implementation of event listeners. When the ***Run…*** button is clicked, Alice inspects this procedure and generates the appropriate code necessary to implement the listeners for the project. See section below on listener procedures
handleActiveChanged	*isActive, activationCount*	TO BE IMPLEMENTED
myFirstMethod		This is where an Alice animation starts, once the runtime window is displayed. Normally this is the method where the programmer creates program statements that control the overall execution of the animation. (A possible exception is **performCustomSetUp,** as described above).

This (scene's) unique properties are shown in Figure A.24.

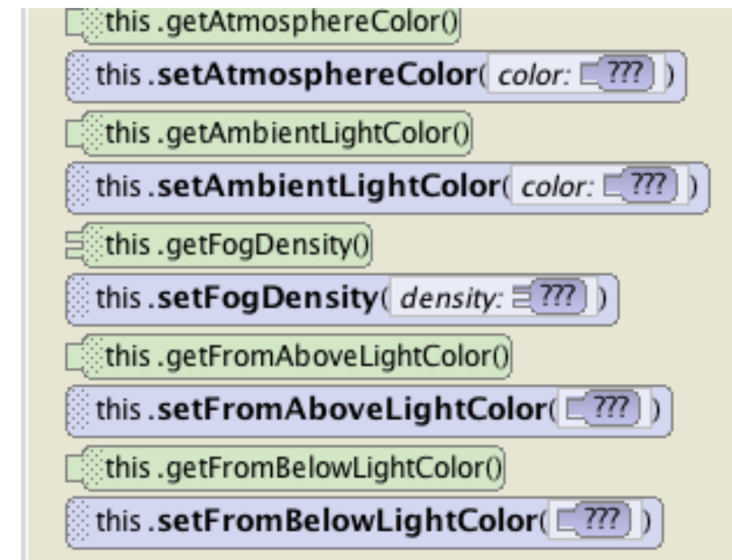


Figure A.24 Properties methods for Scene class

The setters and getters of the Scene class are used to adjust the sky color, the lighting, and the amount of fog in a scene as an animation program is running, as summarized in Table A.13. These methods are useful for changing the appearance of the scene while the animation is being performed (not for setting up the scene in the Scene editor). For example, to change the scene from a daytime to a nighttime setting, the color of the sky could be made darker and the light in the scene could be decreased.

Table A.13 Properties setters and getters for Scene class

Procedure	Argument(s)	Description
setAtmosphereColor	*color*	Sets the *color* of the sky in *this* scene
setAmbientLightColor	*color*	Sets the *color* of the primary light source in *this* scene. Think of it as the color of sunlight in an outdoor scene
setFogDensity	*DecimalNumber*	Used to set the *density* of the fog in *this* scene by setting the *density* value in the range of values from **0.0** (no fog) to **1.0** (no visibility of objects within the fog).
setFromAboveLightColor	*color*	Sets the *color* of a secondary light source from above in *this* scene
setFromBelowLightColor	*color*	Sets the *color* of a secondary light source from below in *this* scene

Function	Return Type	Description
getAtmosphereColor	*color*	Returns the *color* of the sky in *this* scene
getAmbientLightColor	*color*	Returns the *color* of the primary light source in *this* scene; think of it as the color of sunlight in an outdoor scene
getFogDensity	*DecimalNumber*	Returns the value of the density of the fog in *this* scene by getting the *density* value with a range of values from **0.0** (no fog) to **1.0** (no visibility of objects within the fog).
getFromAboveLightColor	*color*	Returns the *color* of a secondary light source from above in *this* scene
getFromBelowLightColor	*color*	Returns the *color* of a secondary light source from below in *this* scene

ADDLISTENER PROCEDURES

Listeners are used for creating interactive programs, especially games. Interactive means that the user is expected to use the keyboard, mouse, or some other input device to control the actions that occur as the program is running.

A listener is an object that, as a program is running, "listens" for a targeted event and responds to that event when it occurs. For example, a mouse-click on object listener will listen for a user to mouse-click on an object in the scene. When the mouse-click on an object occurs, we say the "targeted event has been triggered." When the event is triggered, the listener executes specified instruction statements in response.

In Alice, to create an interactive program, a Listener object must be added to the scene. A listener object is added to the scene by calling an addListener procedure, where Listener is a targeted event. For example, addDefaultModeManipulation creates a listener object that targets a mouse-click on any object in the scene and responds by allowing the user to drag that object around the scene while the animation is running.

Figure A.25 shows a list of addListener procedural methods. Table A.14 summarizes details about the addListener methods, in terms of what event is targeted and how the listener responds.

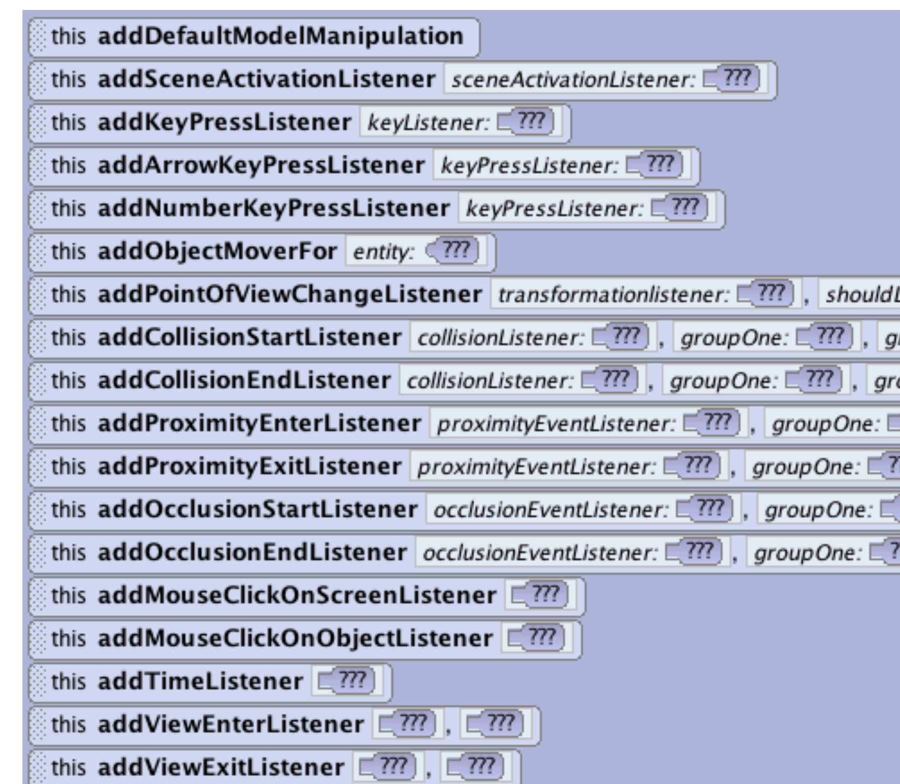


Figure A.25 addListener procedural methods

Table A.14 addListener target and response

Procedure	Argument(s)	Description
adddefaultModelManipulation		Allows the use the mouse to reposition an object in the virtual world as a program is executing. Ctrl-click turns the object, shift-click raises and lowers the object
addSceneActivationListener	*Scene*	UNDER DEVELOPMENT
addKeyPressListener	*Key*	responds to keyboard input from the user. Able to differentiate between Letter, Number, and Arrow keys
addArrowKeyPressListener	*Key*	responds to keyboard input from the user, specifically for Arrow keys (UP, DOWN, LEFT, RIGHT)
addNumberKeyPressListener	*Key*	responds to keyboard input from the user, specifically for Number keys (0..9)
addObjectMoverFor	*Entity*	The parameter object will be moved FORWARD, BACKWARD, LEFT, and RIGHT, based on its own orientation, when the user presses the UP, DOWN, LEFT, and RIGHT arrow keys respectively
addPointOfViewChangeListener	*transformationListener, shouldListenTo*	UNDER DEVELOPMENT
addCollisionStartListener	*collisionListener, Group1, Group2*	UNDER DEVELOPMENT
addCollisionEndListener	*collisionListener, Group1, Group2*	UNDER DEVELOPMENT

Ground

The Ground class has only a limited number of procedural, functional, and property methods, all of which behave exactly the same as those defined by other classes. Figures A.26 (procedures), A.27(functions), and A.28 (specialized property methods) show the methods for the Ground class. These methods were summarized previously in Tables A.4, A.7, and A9.

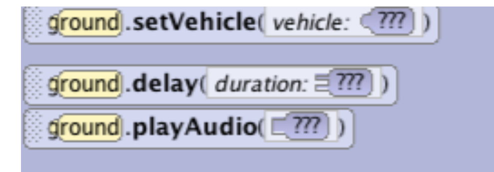


Figure A.26 Procedural methods for Ground class

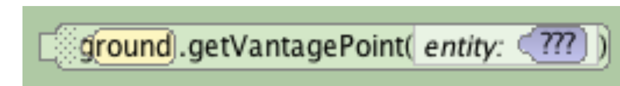


Figure A.27 Functional method for Ground class

Figure A.28 Specialized property methods for Ground class

Camera

The camera has many procedural methods that behave exactly the same as those defined by other classes, as shown in Figure A.29 and summarized previously in Table A.4. camera procedures

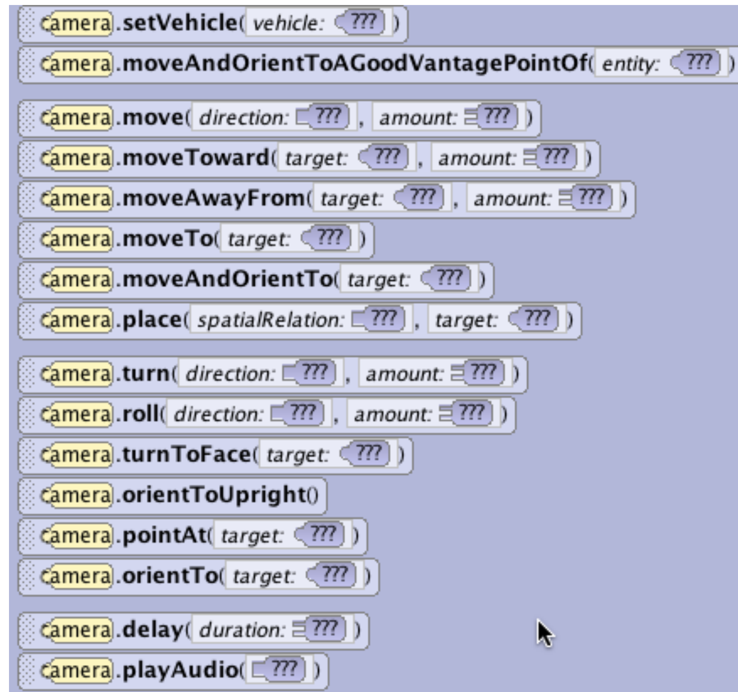


Figure A.29 Camera's procedural methods in common with other classes

One of the procedural methods shown above in Figure A.29, is defined only for the camera: moveAndOrientToAGoodVantagePointOf, as described in Table A.15, below.

Table A.15 Unique Procedural method for Camera class

Procedure	Argument(s)	Description
moveAndOrientToAGoodVantagePointOf	*entity*	Animates the reposition and reorientation of the *camera* from its current position to the vantage point of the entity

The Camera class also has only a limited number of functional, and property methods, all of which behave exactly the same as those defined by other classes. Figures A.30 and A.31 show the functional and property methods for the Camera class. These methods were summarized previously in Tables A.7 and A.9.

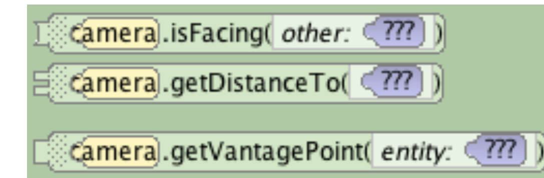


Figure A.30 Camera functional methods

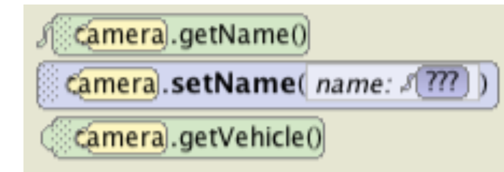


Figure A.31 Camera property methods

Class

A document file that defines how to create an instance of a specific type of object. A class document may also contain definitions for procedures (action methods), functions (computational methods), and properties (fields – objects or items of data, such as color or opacity).

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 3 - Code editor tabs – Scene class

Event

Something that happens. It may be a huge event such as a football game held in an arena or a very small event such as the blink of an eye or a mouse-click

---

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 3 - Code editor tabs – Scene class

Listener

An alert that tells a computer program to listen for an event. A listener acts like an alarm clock. When an alarm clock is set, the clock keeps watch (listens) for a specific time to occur. When the time event occurs, the alarm clock starts a buzzer or turns on the radio to a selected channel.

Related Glossary Terms

Drag related terms here

Index Find Term

Chapter 3 - Code editor tabs – Scene class