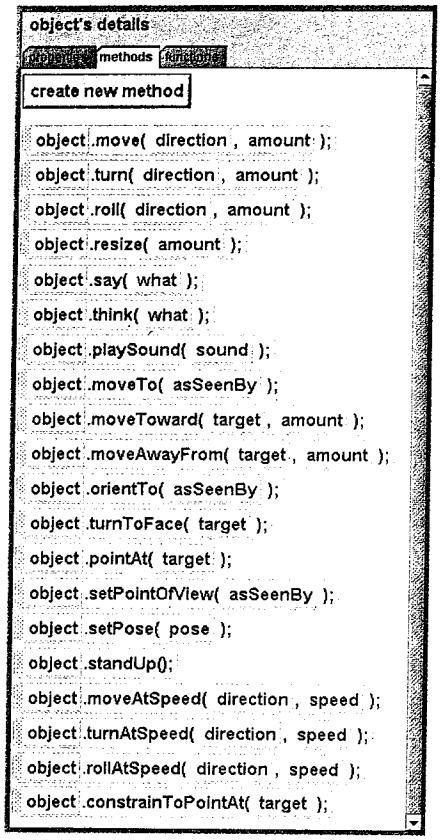
JOEL ADAMS

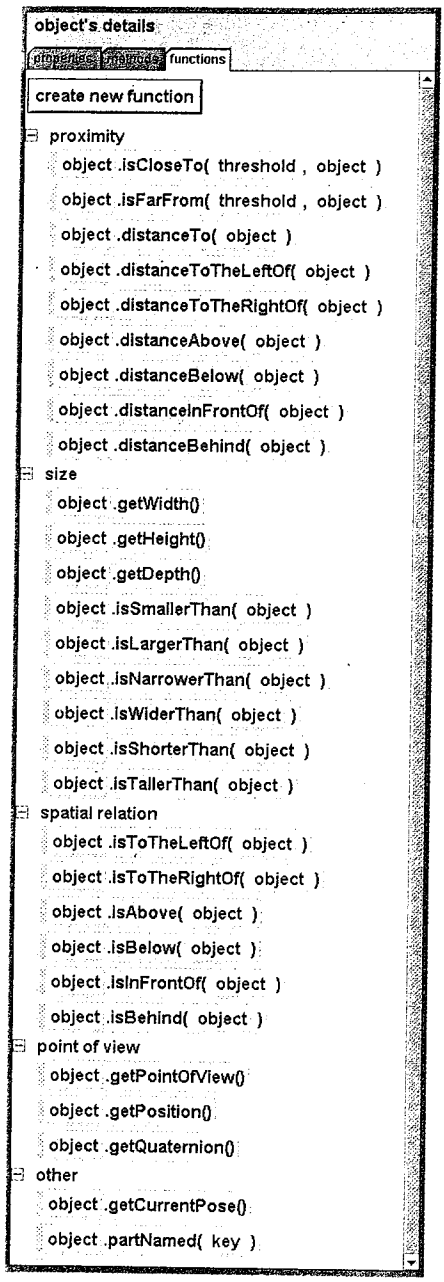
in Action

Computing Through Animation

Object Methods



Object Functions



Alice in Action

Computing Through Animation

Joel Adams

Calvin College

Contents

Preface

I wrote this book to remedy some of the problems in today's introductory computer programming (CS1) courses. To put it politely, most CS1 books are less than engaging, and simply fail to capture the imaginations of most of today's students. No matter how often I say it, many of my students never bother to "read the book." Now, these students aren't blameless, but it isn't entirely their fault. Many CS1 books present computer programming in a dry, abstract, mind-numbing way that's great if you're trying to fall asleep, but not so good if you want to learn.

This is a tragedy, because writing software is one of the best opportunities to exercise creativity in today's world. Traditional engineers and scientists are limited in what they can do by the physical laws that govern our world. But if a software engineer can imagine something, he or she can usually make it happen in the virtual world of the computer. In its 2006 "Best Jobs in America" study, *Money Magazine* listed software engineer #1 on its list of best jobs, because of its *creativity*, pay, and prestige. According to the U.S. Bureau of Labor Statistics, software engineering is also expected to be one of the fastest-growing job markets in the next decade.

This growing demand for software engineers poses a problem, because ever since the dot-com bust in 2001-2, fewer and fewer students have been enrolling in CS1 courses. Of those who do enroll, many drop out, at least in part because the subject matter fails to engage them. CS1 courses are the starting point for software engineers; every student who drops out of CS1 is one less prospective software engineer. Those of us who are CS1 instructors need to do everything we can to (1) attract students to CS1, and (2) retain as many of those students as possible. I wrote this book to try to help attract students to and retain students in CS1.

The Advantages of Alice

At the 2003 ACM SIGCSE conference, I saw Carnegie Mellon's Randy Pausch demonstrate 3D animation software he called Alice. Using Alice, he built a sophisticated 3D animation (like *Shrek* or *Toy Story*, but much simpler) in just a few minutes. To do so, he used the traditional computer programming tools: variables, if statements, loops, subprograms, and so on. But his Alice software offered some startling advantages over traditional programming, including the following:

- *The allure of 3D graphics.* It is difficult to overstate the visual appeal of 3D animations, especially to today's visually-oriented students. When your program works, you feel

euphoric! But even when you make a mistake (a logic error), the results are often comical, producing laughter instead of frustration.

- *The Alice IDE.* Alice includes a drag-and-drop integrated development environment (IDE) that eliminates syntax errors. This IDE eliminates all of the missing semicolons, curly braces, quotation marks, misspelled keywords or identifiers, and other syntax problems that bedevil CS1 students.
- *Object-based programming.* Alice includes a huge library of off-the-shelf 3D objects — ranging from astronauts to ants, cowboys to castles, fairies to farms, mummies to motorboats, ponds to pagodas, robots to rowboats, skyscrapers to space shuttles, turtles to T-rexes, wizards to waterfalls, and zombies to Zambonis — each of which can be animated through a variety of predefined methods. Alice makes it easy to build 3D worlds from these objects. Those objects can then be animated using object-based programming.

By using 3D animation to motivate students, eliminating syntax errors, and turning logic errors into comedy, Alice transforms the CS1 experience from frustration to joy. In short, Alice makes it *fun* to learn object-based programming!

As I watched Professor Pausch's demonstration, it became apparent to me that Alice could solve many of the problems afflicting CS1 courses. If instructors would use Alice to initially *introduce* each programming topic, Alice's engaging environment would help motivate students to master that topic. Then, with that mastery to build upon, the instructor could *review* that topic in a traditional programming language like Java or C++, reinforcing its importance.

Imaginary Worlds

In the summer of 2003, I decided to put some of these ideas to the test, by offering a summer "computer camp" in which we would use Alice to teach some middle school students how to program. Our pilot group of 6th, 7th, and 8th graders learned object-based programming, and had a lot of fun doing so!

Our 2003 results were very encouraging, so in the summer of 2004, we began offering *The Imaginary Worlds Camps*, with 28 middle school boys and 25 middle school girls signing up. The results were amazing. Alice captured their imaginations and wouldn't let them go. Some students wanted to stay at the end of the day to keep working on their programs — we had to force them to leave! Others wanted to skip the snack break. (My college students have never passed up food to keep working on a project.) At the end of the camp, the feedback was loud and uniformly positive: these students had loved learning how to program with Alice.

The *Imaginary Worlds Camps* gave me the chance to experiment with Alice, trying out different examples, and honing them to teach a concept in the simplest way possible. Many of those examples have made their way into this book, and I owe a debt of gratitude to all of the young boys and girls whose creativity, energy, and enthusiasm made these camps so much fun.

Why This Book?

Despite what I said at the beginning of this Preface, many instructors are content with the textbooks they use in their CS1 courses. In order for such instructors to use Alice, someone needed to write a *concise* Alice book to supplement their CS1 texts. I decided to write a "short and sweet" book, that would present just what you need to know to use Alice well, and skip over its more specialized features.

I spent the Fall 2004 semester on sabbatical at Carnegie Mellon. Each week, I spent three days writing parts of this book and three days working as a member of the Alice team, helping them find errors in Alice. Working with these people was invaluable, as they helped me better understand Alice's strengths and weaknesses. This in turn helped me decide which Alice features to include in the book, and which features to exclude.

Pedagogical Features

To help students master the concepts of object-based programming, this book uses a number of pedagogical features, including the following:

- *Movie Metaphors.* Movies are pervasive in our culture. Since Alice programs are similar to movies, this book uses the language of movies to introduce software design. Using this approach, the book builds a conceptual bridge from a student's existing knowledge of movies to the new ideas of software design.
- *Detailed Diagrams.* This book contains approximately 75 color screen captures. Many of these demonstrate the exact drag-and-drop steps needed to use Alice effectively.
- *Engaging Examples.* Using Alice's rich library of 3D objects, this book includes examples that keep students captivated, such as:
  - a dragon flapping its wings
  - a scarecrow singing "Old MacDonald Had a Farm"
  - a fish jumping out of the water
  - three trolls facing off against a wizard
  - a girl walking in a spiral to follow a treasure map
  - and many more!
- *Integrated Software Design.* Beginning in Chapter 1 and continuing throughout, this book emphasizes software design. Each chapter shows how that chapter's concepts fit into the overall software design methodology. Students following this methodology can never say, "I don't know where to start."
- *Alice Tips.* Most chapters include one or more special "Alice Tip" sections that cover critical details students need to know to use Alice effectively.
- *Chapter Summaries.* The final section of each chapter includes a bulleted list of the key concepts covered in that chapter, plus a separate list of that chapter's key vocabulary terms.
- *Programming Projects.* Each chapter concludes with 10-12 programming projects, of varying levels of difficulty.

Using This Book

This book is intended as a supplement for CS1 courses, but it can be used in any course where an instructor wishes to teach the ideas of object-based programming. The book covers these ideas in six chapters, arranged as follows:

1. Getting started: using objects and methods
2. Building methods: using abstraction to hide details
3. Variables, parameters, and functions: computing and storing data for later use
4. Control structures: controlling flow via `if`, `while`, and `for` statements
5. Data structures: using and processing arrays and lists
6. Events: handling mouse and keyboard input

These six chapters can be used in a variety of ways, including:

- *The Spiral Approach*: Spend 4-6 weeks introducing all of the programming concepts using Alice (the first spiral). Then spend the remainder of the semester revisiting those same concepts in Java or a different language (the second spiral). In this approach, the programming concepts are covered in two distinct "batches": an Alice batch, followed by a Java batch.
- *The Interleaved Approach*: For each concept (for example, parameters), introduce that concept using Alice. After the students have had hands-on experience with that concept in Alice, immediately revisit that same concept in Java or a different language. In this approach, the programming concepts are covered sequentially, with the Alice and Java coverage interleaved.

If an instructor does not normally cover event-driven programming, Chapter 6 may be omitted, or deferred until the end of the course. However, most students find this material to be *very* engaging, because it allows them to start building games! If an instructor wishes to do so, events may be introduced at any point after Chapter 3.

As students work through the examples in this book, they should make sure to save their Alice worlds regularly. We will begin some worlds in one chapter and add to those same worlds in a later chapter, so students should save at the end of each example. Each world should be saved with a unique, descriptive name, so that it can be easily identified later.

[illegible]

The appendices provide resources and material supplementing what is covered in the chapters. Appendix A presents an exhaustive list of Alice's standard methods and functions, including detailed behavioral descriptions. Appendix B provides a "mini-chapter" on recursion, with examples that help students visualize recursion.

The inside covers contain two useful Alice "Quick Reference" pages. Inside the front cover is a complete list of the standard methods and functions that can be applied to an Alice object. Inside the back cover is a complete list of the standard functions that can be applied to an Alice world. Unlike the lists in Appendix A, these "Quick References" display each method, function, and parameter exactly as they appear in Alice. By presenting all of these methods and functions together, a student can see all of the methods and functions at once, and quickly locate a particular method or function.

[illegible]

Copies of the example programs from this book are available online, at

- the Student Downloads section of the Course Technology Web site (**www.course.com**)
- the author's Alice Web site (**http://alice.calvin.edu**)

A feedback link and errata list are also available at the author's Web site. If you find a mistake, or want to point out a feature that works especially well, please use that feedback link. Such feedback will help me improve future editions of the book.

The Alice 2.0 software can be freely downloaded from **http://alice.org**.

Chapter 1
Getting Started With Alice

The computer programmer ... is a creator of universes for which he [or she] alone is the lawgiver ... universes of virtually unlimited complexity can be created in the form of computer programs. Moreover ... systems so formulated and elaborated act out their programmed scripts. They compliantly obey their laws and vividly exhibit their obedient behavior. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or a field of battle and to command such unswervingly dutiful actors or troops.

JOSEPH WEIZENBAUM

If you don't know where you're going, you're liable to wind up somewhere else.

YOGI BERRA

Louis, I think this is the beginning of a beautiful friendship.

RICK (HUMPHREY BOGART) TO CAPTAIN RENAULT (CLAUDE RAINS), IN *CASABLANCA*

Objectives

Upon completion of this chapter, you will be able to:

- Design a simple Alice program
- Build a simple Alice program
- Animate Alice objects by sending them messages
- Use the Alice **`doInOrder`** and **`doTogether`** controls
- Change an object's properties from within a program
- Use Alice's **`quad view`** to position objects near one another

Welcome to the fun and exciting world of computer programming! In this chapter, we are going to build our first computer program using **Alice**, a free software tool for creating virtual worlds.

1.1 Getting and Running Alice

1.1.1 Downloading Alice

Alice can be freely downloaded from the Alice website at **`http://alice.org`**. For the Windows version, clicking the download link begins the transfer of a compressed archive file named **`Alice.zip`** to your computer. (For MacOS, the file is named **`Alice.dmg`**.) Save this file to your computer's desktop.

1.1.2 Installing Alice

Alice does not have a special installer like other programs you might have used. When the download has finished, double-click the **`Alice.zip`** (or **`Alice.dmg`**) file to open the archive file. Your computer will open a window containing a folder named **`Alice`**. Drag that **`Alice`** folder from the window onto your computer's desktop.

If you'd rather not have the **`Alice`** folder on your desktop, open a window to the folder in which you wish to store **`Alice`** (for example, **`C:\Program Files\`**). Then drag the **`Alice`** folder from your desktop into that window.

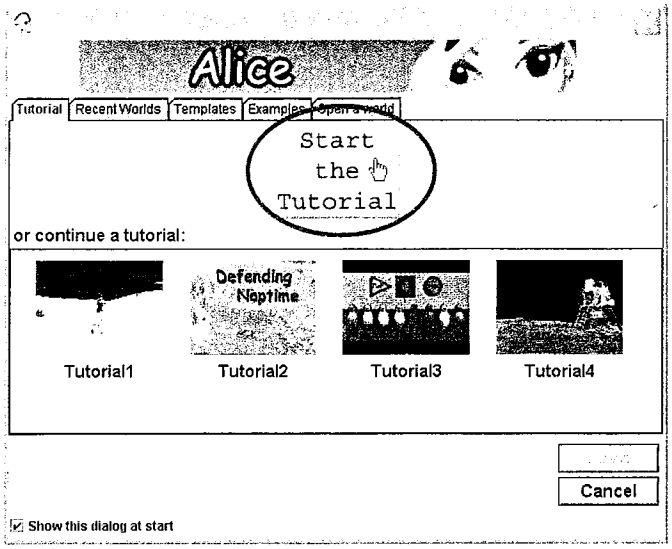
Once the **`Alice`** folder is where you want it, open the **`Alice`** folder, and locate the file named **`Alice.exe`** (or just **`Alice`** in MacOS). In Windows, right-click the file, and from the menu that appears, choose **`Create Shortcut`** to create a shortcut (alias) to **`Alice.exe`**. (MacOS users, select the file and choose **`File->Make Alias`**.) Drag the resulting shortcut to your desktop and rename it **`Alice`**, so that you can launch Alice conveniently.

1.1.3 Running Alice

To start Alice, just double-click the **`Alice`** icon on your desktop. Congratulations!

1.2 The Alice Tutorials

As shown in Figure 1-1, when you start Alice for the first time, Alice gives you the option of working through a set of interactive tutorials. These excellent tutorials cover the basics of using Alice while giving you hands-on practice working in the Alice environment. Because they are such effective learning tools, we are going to let these tutorials teach you the basics of Alice. This chapter will concentrate on aspects of Alice *not* covered in the tutorials.



(If this window does not appear, you can make it appear by clicking on Alice's **Help** menu and then selecting the **Tutorial** choice.) To activate the tutorials, click the **Start the Tutorial** button, and then work your way through the four tutorials. Remember, the point of these tutorials is to learn how to use Alice, not to see how fast you can finish them. Read carefully, taking special note of *what* you are doing at each step, *how* you are doing it, and *why* you are doing it. Close Alice when you are finished with the tutorials.

The rest of this chapter assumes you have completed the tutorials, so if you have not yet done so, you should *complete the tutorials now, before proceeding further.* If for some reason you cannot complete the tutorials right away, feel free to keep reading, but I strongly encourage you to complete the tutorials as soon as possible, and then *re-read this chapter.*

Developing programs to solve problems is a complex process that is both an art and a science. It is an art in that it requires a good deal of imagination, creativity, and ingenuity. But it is also a science in that it uses certain techniques and methodologies. In this chapter, we're going to work through the *thought process* that goes into creating computer software.

If you can manage it, the very best way to read this book is at a computer, doing each step or action as we describe it. By doing so, you will be engaging in *active learning*, which is a much better way to learn than by trying to absorb the ideas through passive reading.

1.3 Program Design

Now that you have finished the tutorials, we are ready to build our first computer program and put into practice several of the skills you learned in the tutorials. Programming in Alice is similar to *filmmaking*, so let's begin with how a film is put together.

When filmmakers begin a film project, they do not begin filming right away. Instead, they begin by *writing*. Sometimes they start with a short prose version of the film called a **treatment**; eventually they write out the film's dialog in a **screenplay**, but they always begin by *writing*, to define the basic structure of the *story* their film is telling.

A screenplay is usually organized as a series of **scenes**. A scene is one piece of the story the film is telling, usually set in the same location. A scene is usually made up of multiple **shots**. A shot is a piece of the story that is told with the **camera** in the same position. Each change of the camera's **viewpoint** in a scene requires a different shot. For example, if a scene has two characters talking in a restaurant, followed by a closeup of one of the character's faces, the viewpoint showing the two characters is one shot; the viewpoint of the closeup is a different shot.

Once the screen play is complete, the filmmaker develops **storyboards**, which are drawings that show the position and motion of each character in a shot. Each storyboard provides a sort of blueprint for a shot, indicating where the actors stand, where the camera should be placed with respect to them, and so on. (You may have seen storyboards on the extras that come with the DVD version of a film.)

Creating an Alice program is much like creating a film, and modern computer software projects are often managed in a way that is quite similar to film projects.

1.3.1 User Stories

A modern software designer begins by writing a prose description of what the software is to do, from the perspective of a person using the software. This is called a **user story**. For example, here is a user story for the first program we are going to build:

> When the program begins, Alice and the White Rabbit are facing each other, Alice on the left and the White Rabbit on the right. Alice turns her head and then greets us. The White Rabbit also turns and greets us. Alice and the White Rabbit introduce themselves. Simultaneously, Alice and the White Rabbit say "Welcome to our world."

A user story provides several important pieces of information, including:

- A basic description of what happens when the user runs the program
- The *nouns* in the story (for example, Alice, the White Rabbit) correspond to the **objects** we need to place in the Alice world. Objects include the characters in the story — background items like plants, buildings, or vehicles, and so on.
- The *verbs* in the story (for example, turns, says) correspond to the *actions* we want the objects to perform in the story.
- The chronological *flow* of actions in the story tells us what has to happen *first*, what happens *next*, what happens *after that*, and so on. The *flow* thus describes the *sequence of actions* that take place in the story.

By providing the objects, behaviors, sequence of actions, and description of what the program will do, a user story provides an important first step in the software design process, upon which the other steps are based. The user story is to a good software product as the screenplay is to a good film.

It is often useful to write out the flow of the story as a numbered sequence of objects and actions. For example, we can write out the flow in the user story as shown in Figure 1-2:

Scene: Alice is on the left, the White Rabbit is on the right.

Alice turns her head toward the user.
Alice greets the user.
The White Rabbit turns to face the user.
The White Rabbit greets the user.
Alice introduces herself.
The White Rabbit introduces himself.
Simultaneously, Alice and the White Rabbit say "Welcome to our world".

A **flow** is thus a series of steps that precisely specify (in order) the behavior of each object in the story. In programming terminology, a flow — a sequence of steps that solve a problem — is called an **algorithm**.

1.3.2 Storyboard-Sketches

When they have a completed screenplay, filmmakers often hire an artist to sketch each shot in the film. For each different shot in each scene, the artist creates a drawing (in consultation with the filmmaker) of that shot, with arrows to show movements of the characters or camera within the shot. These drawings are called **storyboards**. When completed, the collection of storyboards provides a graphical version of the story that the filmmaker can use to help the actors visualize what is going to happen in the shot, before filming begins.

To illustrate, Figure 1-3 shows a pair of storyboards for a scene we will develop in Section 2.4.1. The first storyboard frames the scene, showing three trolls menacing a wizard, with the wizard's castle in the background. In the second storyboard, we zoom in on the wizard to get a better view of his reaction. The progression of storyboards thus serves as a kind of cartoon version of the story, which the filmmaker uses to decide how the film will look, before the actual filming begins. By first trying out his or her ideas on paper, a filmmaker can identify and discard bad ideas early in the process, before time, effort, and money are wasted filming (or in our case, programming) them.

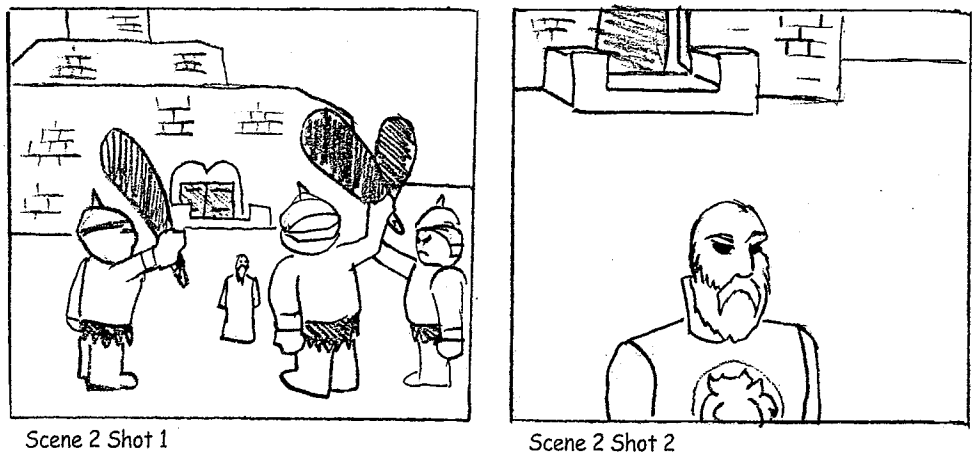


FIGURE 1-3 A storyboard and corresponding scene

In a similar fashion, the designers of modern computer software draw sketches of what the screen will look like as their software runs, showing any changes that occur. Just as each distinct shot in a film scene requires its own storyboard, each distinct screen in a computer application requires a different sketch, so we will call these **storyboard-sketches**. Since our first program has just one scene, it has just one storyboard-sketch, as shown in Figure 1-4.

1.4

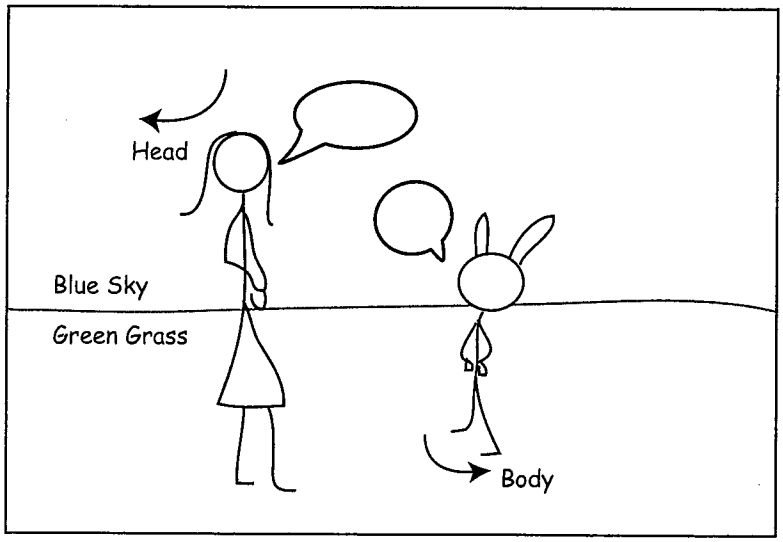


FIGURE 1-4 Storyboard-sketches

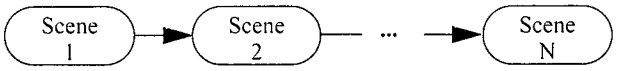
In Alice programming, the storyboard-sketches provide important information for the programmer about each object visible on the screen, including:

Its **position** (where it is with respect to the other objects)
Its **pose** (what are the positions of its limbs, if it has any)
Its **orientation** (what direction it is facing)

Storyboard-sketches also indicate where Alice's **camera** object should be positioned, whether it is stationary or moving during the shot, and so on.

1.3.3 Transition Diagrams

When a program has multiple scenes, it has multiple storyboards. When all the storyboard-sketches are completed, they are linked together in a **transition diagram** that shows any special events that are required to make the transition from one sketch to the next. In a movie, there are no special events, so the transition diagram is a simple linear sequence, as shown in Figure 1-5.



With a user story, storyboard-sketches, and transition diagram in hand, the program's **design** is done, and we are ready to build it in Alice. We begin by starting Alice. Alice displays a **`Welcome to Alice`** window that allows us to choose a **`Template`** (background and sky) for the world, as shown in Figure 1-6.

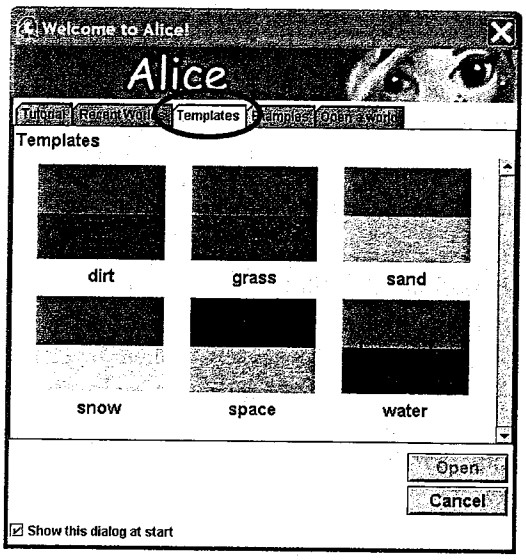


FIGURE 1-6 Alice's template worlds

(You can also get this window to appear by clicking Alice's **File** menu, and then choosing **New World**.) Double-click the template you want to use (we will choose the **grass** template here), and Alice will create a pristine[1] three-dimensional world for you, using that template, as shown in Figure 1-7. (Your screen may look slightly different.) For consistency with the tutorials, we have added the names for the various areas in Alice to Figure 1-7.

1. Here and elsewhere, we will use the word *pristine* to describe an Alice world in its beginning state — before any objects have been added to it.

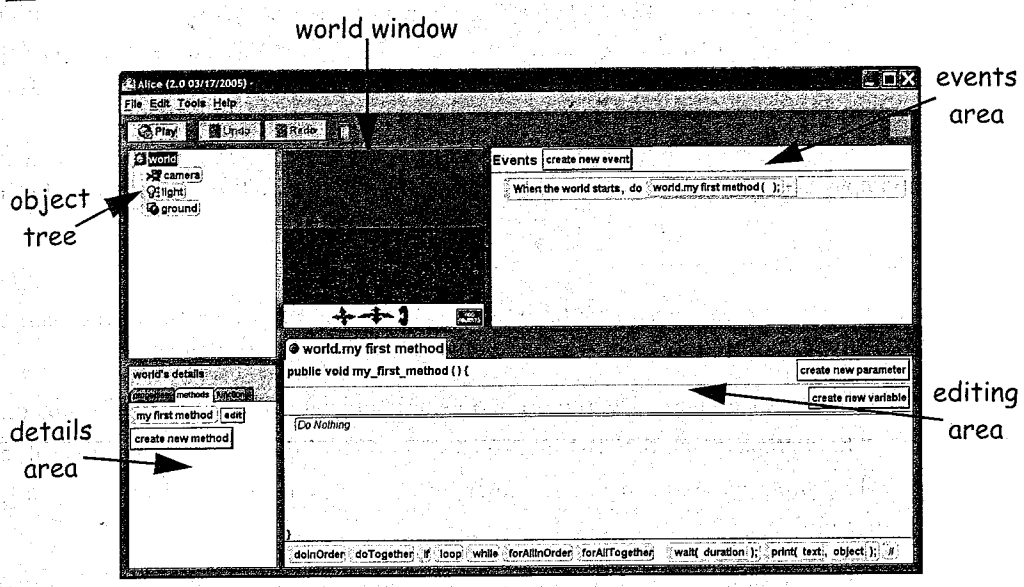


FIGURE 1-7 A pristine Alice world

Menus

At the top of the Alice window are four menus:

- **File** lets you load and save your Alice programs/worlds (and other things).
- **Edit** lets you change your preferences.
- **Tools** lets you examine your world's statistics, error console, and so on.
- **Help** lets you access the Alice tutorials, some example worlds, and so on.

Buttons

Below Alice's menus are three buttons:

- **Play** runs the program associated with the current world.
- **Undo** undoes your most recent action (this is very handy!).
- **Redo** redoes the most recently undone action.

If you are like me, you will find yourself using the **Play** button frequently (every time you want to run your program); the **Undo** buton when using trial-and-error to find just the right effect, and the **Redo** button very rarely.

The Object Tree

The *object tree* is where the objects in your world are listed. Even in a pristine world, the *object tree* contains several objects, namely the **camera**, the **light**, and the **ground**. Like other objects in Alice, the **camera** can be moved within the world. Its position determines

what is seen in the *world window*. As you saw in the tutorials, the blue arrow-controls at the bottom of the *world window* can be used to modify the **camera**'s position and orientation.

The **light** can also be moved, though we won't be doing much of that. If you are working on a shot and find that you need more light, you can change the **light**'s position, orientation, color, and brightness.

It doesn't make much sense to move the ground, though we may wish to change it (for example, from grass to snow, or sand, or ...). We'll see how to do this in Section 2.3 in Chapter 2.

The Details Area

In the *details area*, there are three *tabbed panes*: the *properties* pane, the *methods* pane, and the *functions* pane. For whatever object is selected in the *object tree*:

- The *properties* pane lists the properties or changeable attributes of that object;
- The *methods* pane lists the messages we can send that object to animate it; and
- The *functions* pane lists the messages we can send that object to get information from it.

Take a moment to click through these panes, to get a feel for the things they contain. We'll present an overview of them in Section 1.5.

The Editing Area

The *editing area* is where we will edit or build the program that controls the animation. As can be seen in Figure 1-7, the *editing area* of a pristine world contains a method named **World.my first method** that is empty, meaning it contains no *statements*. Very shortly, we will see how to build our first program by adding statements to this method.

At the bottom of the *editing area* are *controls* (**doInOrder**, **doTogether**, **if**, **loop**, and so on) that can be used to build Alice statements. We will introduce these controls one by one, as we need them, throughout the next few chapters.

The Events Area

The *events area* is where we can tell Alice what to do when special actions called **events** occur. A pristine world contains just one event, as can be seen in Figure 1-7. This event tells Alice to send the **my first method** message to the **World** object when the world starts (that is, when the user clicks the **Play** button). Clicking the **Play** button thus causes **my first method** to **run**, meaning any statements within it are performed. Programmers often use the phrases ***run* a program** and ***execute* a program** interchangeably.

1.4.1 Program Style

Before we begin programming, you may want to alter the *style* in which Alice displays the program. Click the **Edit** menu, followed by the **Preferences** choice, and Alice will display the *Preferences* window shown in Figure 1-8.

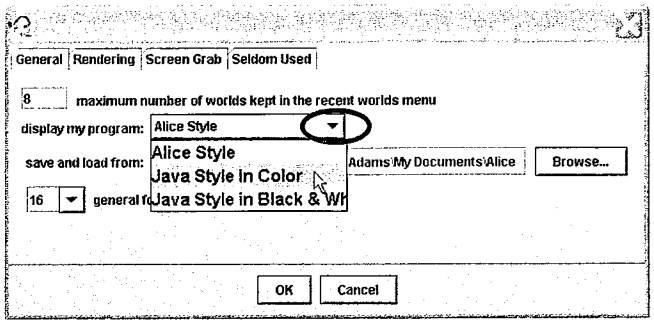


FIGURE 1-8 Using Java style

Since Java is a popular programming language, we will be displaying our programs using Alice's *Java style in Color*. By doing so, Alice will provide us with an introduction to the Java we will learn later in the course. If you want your programs to look consistent with those in the text, please make this change on your copy of Alice. You will then need to restart Alice for the changes to take effect.

1.4.2 Adding Objects to Alice

Once we have a pristine world, the next step is to populate it with objects using the skills you learned in the Alice tutorials. By clicking the **ADD OBJECTS** button below the *world window*, locating **Class AliceLiddell** and **Class WhiteRabbit** in the Alice Gallery (under **People** and **Animals**, respectively), adding them to the world, and repositioning and rotating them, we can build the scene from our first storyboard-sketch, as shown in Figure 1-9.

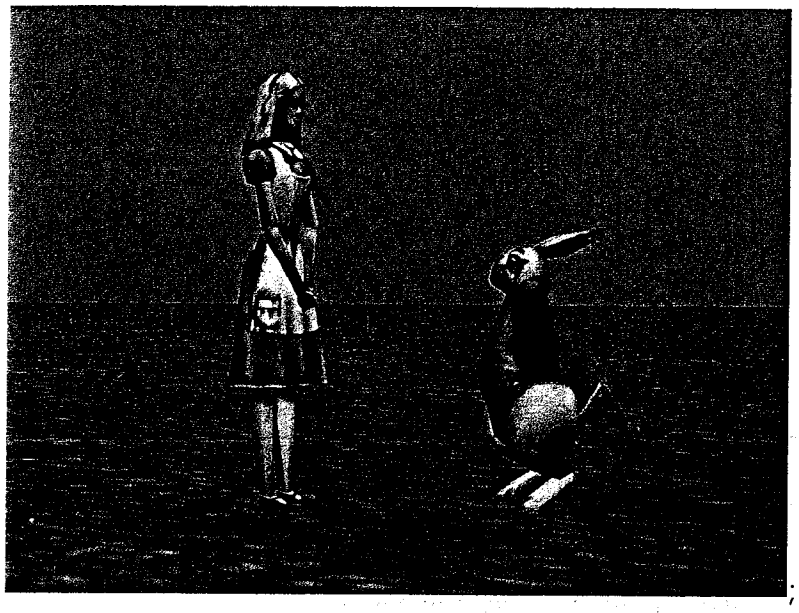


FIGURE 1-9 Alice Liddell and the White Rabbit

The items in the Alice Gallery are not objects but are like blueprints that Alice uses to build objects. Such blueprints are called **classes**. Whenever we drag a class from the Gallery into the *world window*, Alice uses the class to build an object for the world.

For example, when we drag the **`AliceLiddell`** and **`WhiteRabbit`** classes to the world, Alice adds two new objects to the world, and lists them in the *object tree*: **`aliceLiddell`** and **`whiteRabbit`**. If we were to drag **`Class WhiteRabbit`** into the world again, Alice would again use the class to create an object for the world, but this object would be named **`whiteRabbit2`**. Feel free to try this; you can always delete **`whiteRabbit2`** (or any object in the *object tree*), either by dragging it to the **`Trash`**, or by right-clicking it and selecting **`delete`** from the menu that appears.

The key idea is that each object is made from a class. Even though the world might contain ten **`whiteRabbit`** objects, there would still be just one **`WhiteRabbit`** class from which all of the **`whiteRabbit`** objects were made.

To distinguish objects from classes, Alice follows this convention: each word in the name of a *class* is capitalized (for example, **`AliceLiddell`**, **`WhiteRabbit`**); but for an *object*, each word in the name *except the first* is capitalized (for example, **`aliceLiddell`**, **`whiteRabbit`**).

If you don't like the name Alice gives an object, you can always rename it by (1) right-clicking the object's name in the *object tree*, and (2) choosing **`rename`** from the menu that appears, and (3) typing your new name for the object. Alice will then update all statements that refer to the object to use the new name.

With the objects **aliceLiddel** and **whiteRabbit** in place, we are almost ready to begin programming! In Alice, programming is accomplished mainly in the *object tree* (to select the object being animated), the *details area* (the **properties** or characteristics of an object are listed under the *properties* tab, and the **messages** we can send an object are listed under the *methods* and *functions* tabs), and the *editing area* (to add statements to the program that animate the selected object).

1.4.3 Accessing Object Subparts

In the user story, the first action is that Alice should seem to see us (the user) and turn her head toward us. To make this happen, we will use skills from the Alice tutorials.

If we click on **aliceLiddell** in the *object tree*, then we select all of **aliceLiddell**. However, the user story says that Alice is to turn her head, so we just want to select that part of her. To do so, we click the + sign next to **aliceLiddell** in the *object tree* to view her subparts, and then do the same on her **neck**, exposing her **head**, which we then select as shown in Figure 1-10.

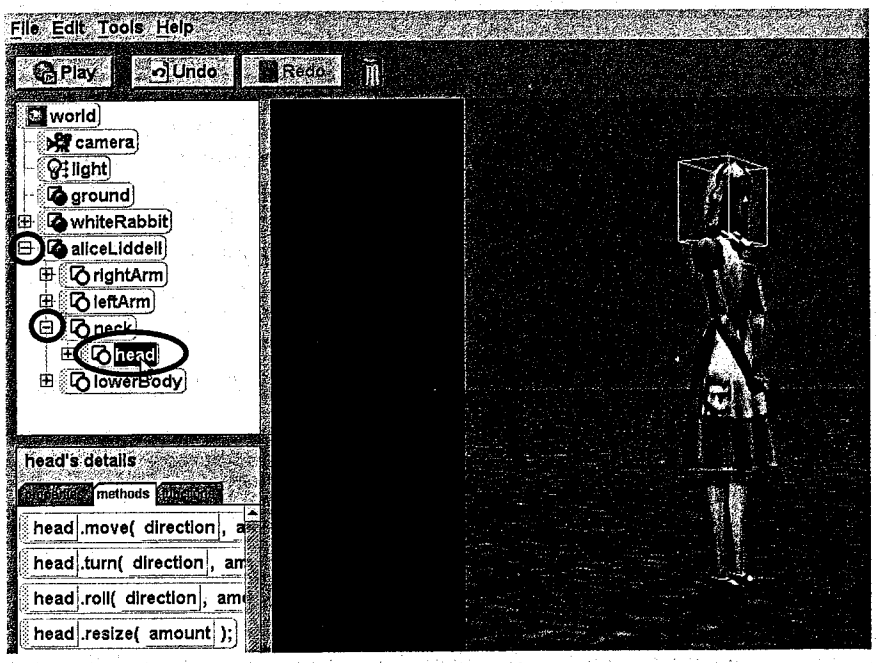


FIGURE 1-10 Accessing an object's subparts

As can be seen in Figure 1-10, when we click on an object in the *object tree* (for example, Alice Liddell's **head**), Alice draws a box around that object in the *world window*, to highlight it and show its boundaries. This box is called an object's **bounding box**, and every Alice object has one.

Selecting an object's subpart in the *object tree* also changes the *details area* to indicate the *properties*, *methods*, and *functions* for that subpart.

Since the steps in a flow or algorithm need to be performed in a specified order, we begin programming by dragging a **doInOrder** control from the bottom of the *editing area*, as shown in Figure 1-11.

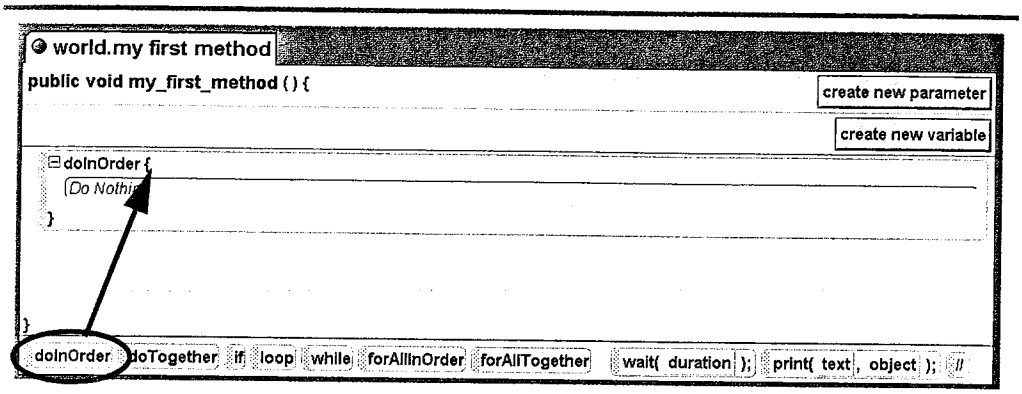


FIGURE 1-11 Dragging the **doInOrder** control

The **doInOrder** control is a structure within which we can place program statements (see Section 1.4.7 below). As its name suggests, any statements we place within the **doInOrder** will be performed in the order they appear, top-to-bottom. The **doInOrder** control also has additional convenient features that we will see in later chapters.

1.4.4 Sending Messages

Alice programming consists largely of sending messages to objects.

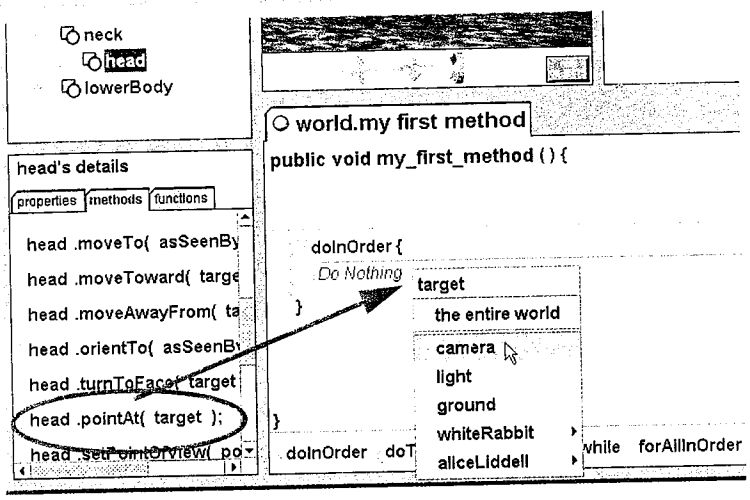
> You can get an object to perform a desired behavior by sending the object a **message** that asks the object to produce that behavior.

In Alice, behavior-producing messages are called **methods**, and are listed under the *methods pane* of the *details area*.

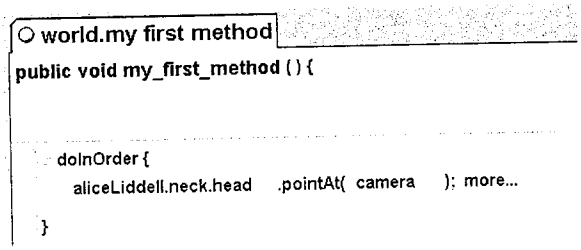
To illustrate, step 1 of the algorithm is to make Alice Liddell's head turn to look at the user. To accomplish this, we can send **aliceLiddell.neck.head** the **pointAt()** message, and specify the **camera** as the thing her head is to face. (Similarly, to make the White Rabbit say "Hello", we can send **whiteRabbit** the **say()** message, and specify **Hello** as the thing we want him to say.) With Alice Liddell's **head** selected in the *object tree*, we scan through the methods in the *details area* until we see **pointAt()**. We then click on **pointAt()**, drag it into the *editing area*, and drop it.

As we drag a method, Alice surrounds it with a red border so long as the mouse is in a place where dropping it has no benefit. When the mouse moves into an area where we can drop the method beneficially, Alice changes the method's border's color from red to green. Alice consistently uses the color red to warn you that you *should not* do something (for example, drop the method), and uses the color green to indicate when you *may* do something.

The **pointAt()** message requires that we specify a **target** — the thing at which we want Alice Liddell's head to point. When you drop the **pointAt()** method in the *editing area*, Alice displays a menu of the objects in your world, from which you can choose the **target**, as shown in Figure 1-12.



When we select the **camera**, Alice redraws the *editing area* as shown in Figure 1-13.



1.4.5 Testing and Debugging

If we now click the **Play** button, you will see **aliceLiddell**'s head turn and seem to look at you (the user)! Figure 1-14 shows the end result.

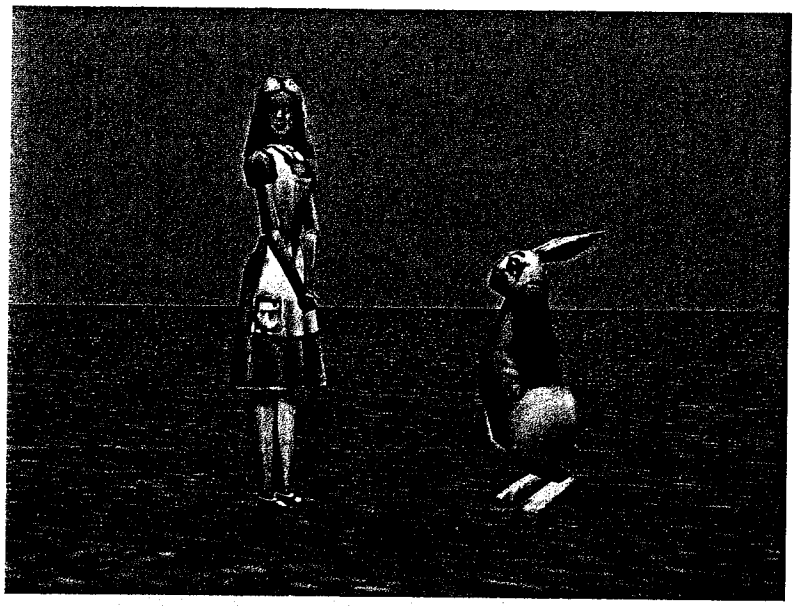


FIGURE 1-14 Alice Liddell looks at the user

(If you are not at a computer doing this interactively, compare Figure 1-14 to Figure 1-9 to see the effect of sending Alice's **head** the **pointAt(camera)** message.)

By clicking the **Play** button, we are **testing** the program, to see if it produces the desired result. If the program does something other than what we wanted, then it contains an **error** or **bug**. Finding and fixing the error is called **debugging** the program. If you have followed the steps carefully so far, your program should have no bugs, so let's continue. (If your program does have a bug, compare your editing area against that shown in Figure 1-13 to see where you went wrong.)

1.4.6 Coding the Other Actions

We can use similar steps to accomplish actions 2, 3, 4, 5, and 6 of the algorithm in Figure 1-2 by sending **pointAt()** or **say()** messages to **aliceLiddell** or the **whiteRabbit**. When we send an object the **say()** message, Alice displays a menu from which we can select what we want the object to say. To customize the greetings, select the **other...** menu-choice; then in the dialog box that appears, type what you want the object to say. After a few minutes of clicking, dragging, and dropping, we can have the partial program shown in Figure 1-15.

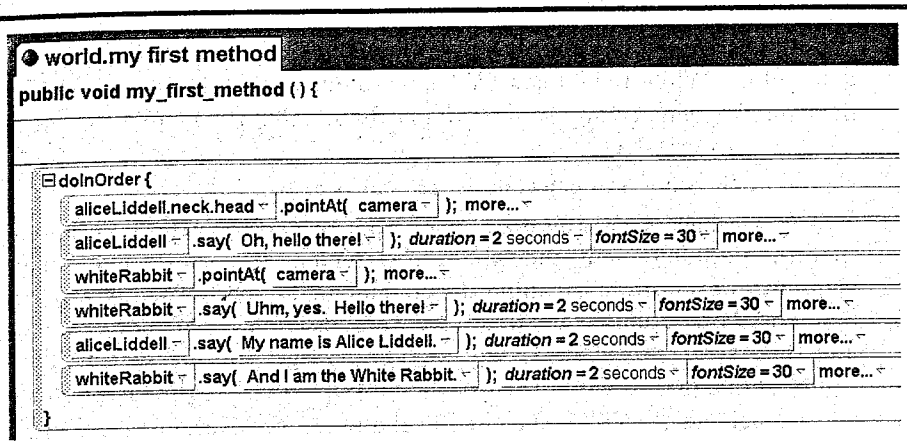


FIGURE 1-15 A partial program

By clicking on the **more...** to the right of a message in the *editing area*, we can customize various attributes of that message. For example, in Figure 1-15, we have increased the **duration** attribute of each **say()** message (depending on the length of what is being said), to give the user sufficient time to read.

For **say()** *messages, set the duration to 2–3 seconds per line of text being displayed, to give the user time to read what is being said.*

You can also adjust the **fontSize** (and other attributes) to specify the appearance of a **say()** message's letters. We will always use a **fontSize** of at least **30**, to ensure that the letters display well on high-resolution computer screens (see Figure 1-15).

1.4.7 Statements

Most of the lines in the program have the same basic structure:

```
object.message(value); more...
```

In programming terminology, such a line is sometimes called a **statement**. A computer program consists of a collection of statements, the combination of which produce some desirable behavior. The basic structure shown above is quite common, and is what we will use most often.

The **doInOrder** control is also a statement; however it is a statement that controls *how other statements are performed* (that is, one at a time, top-to-bottom).

1.4.8 The Final Action

We are nearly done! All that is left is the final step in the algorithm, in which Alice Liddell and the White Rabbit say "Welcome to our world" simultaneously. It should be evident that we can accomplish this in part by sending **`say()`** messages to **`aliceLiddell`** and the **`whiteRabbit`**. For both objects, the value accompanying the **`say()`** message should be the same value: **`Welcome to our world`**.

As we have seen, the **`doInOrder`** control performs the first statement within it, then the next statement, then the next statement, and so on. This is sometimes called **sequential execution**, meaning the statements are performed in order or *in sequence*. Sequential execution means that if we were to send **`aliceLiddell`** the **`say()`** message, and then send **`whiteRabbit`** the **`say()`** message, the message to the White Rabbit would not be performed until after the message to Alice Liddell had been completed.

To achieve the effect specified in the user story, we must send **`say()`** messages to **`aliceLiddell`** and the **`whiteRabbit`** *simultaneously*. We can accomplish this using the **`doTogether`** control, located at the bottom of the *editing area*. To use this control, we click on **`doTogether`**, drag it upwards into the *editing area*, and drop it when the green bar appears below the last statement in the program, producing the program shown in Figure 1-16.

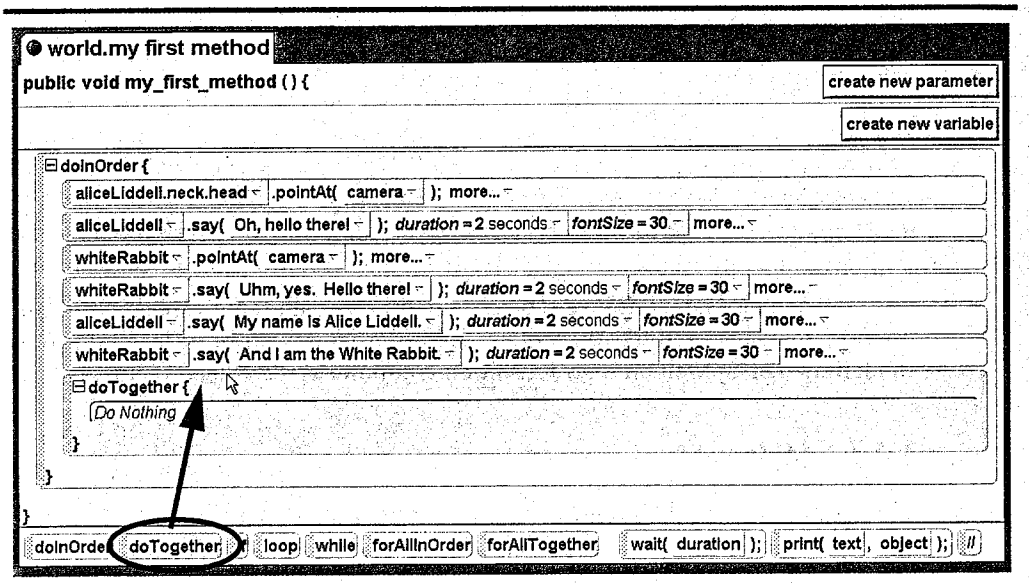


FIGURE 1-16 Dragging the **`doTogether`** control

The **`doTogether`** control is another Alice statement. Like the **`doInOrder`**, it has a form different from the **`object.message()`** structure we saw previously. When the program performs a **`doTogether`** statement, all statements within it are performed simultaneously, so it should provide the behavior we need to finish the program.

Using the same skills we used earlier, we can send **`say()`** messages to **`aliceLiddell`** and to the **`whiteRabbit`**. However, now we drop these messages inside the **`doTogether`** statement, yielding the final program, shown in Figure 1-17.

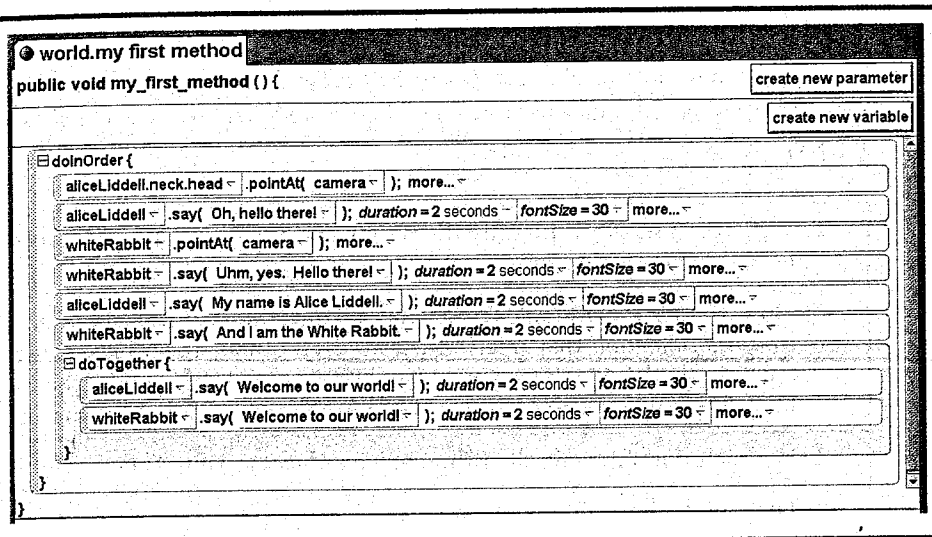


FIGURE 1-17 Our first program

1.4.9 Final Testing

When we run the program, the final scene appears as shown in Figure 1-18.

FIGURE 1-18 Alice Liddell and the White Rabbit speaking together

We saw earlier that the Alice **doInOrder** statement performs the statements within it *sequentially*. By contrast, the **doTogether** statement performs the statements it contains *simultaneously* or *concurrently*.

1.4.10 The Software Engineering Process

The approach we just used to create our first program is an example of a methodical, disciplined way that computer software can be created. The process consists of the following steps:

1. Write the *user story*. Identify the nouns and verbs within it. Organize the nouns and verbs into a *flow* or *algorithm*.
2. Draw the *storyboard-sketches*, one per distinct shot in your program, and create a *transition diagram* that relates them to each other. If you have some users available, have them review your sketches for feedback, and take seriously any improvements they suggest. Update your user story and algorithm, if necessary.
3. For each noun in your algorithm: add an *object* to your Alice world.
4. For each verb in your algorithm:
   a. Find a *message* that performs that verb's action, and send it to the verb's object. (If the object has no message that provides that verb's action, we'll see how to build our own methods in Chapter 2.)
   b. Test the message sent in Step 4a, to check that it produces the desired action. If not, either alter how the message is being sent (with its **more...** attributes), or find a different message (and if you cannot find one, build your own).

Steps 1 and 2 of this process are called **software design**. Steps 3 and 4 — in which we build the program and then verify that it does what it is supposed to do — are called **software implementation and testing**. Together, software design, implementation, and testing are important parts of **software engineering** — a methodical way to build computer programs.

We will use this same basic process to create most of the programs in this book. You should go through each of these steps for each program you write, because the result will be better-crafted programs.

1.5 Alice's Details Area

As mentioned earlier, Alice's *details area* provides three tabbed panes. Whenever an object is selected in the *object tree*, these three panes list the properties or characteristics of that object, the methods for that object, and the functions, or questions that we can ask that object. In this section, we provide an overview of this *details area*.

1.5.1 The *properties* Pane

To see the properties of an object, first click on that object in the *object tree*, and then click the *properties* tab in the *details area*, as shown in Figure 1-19.

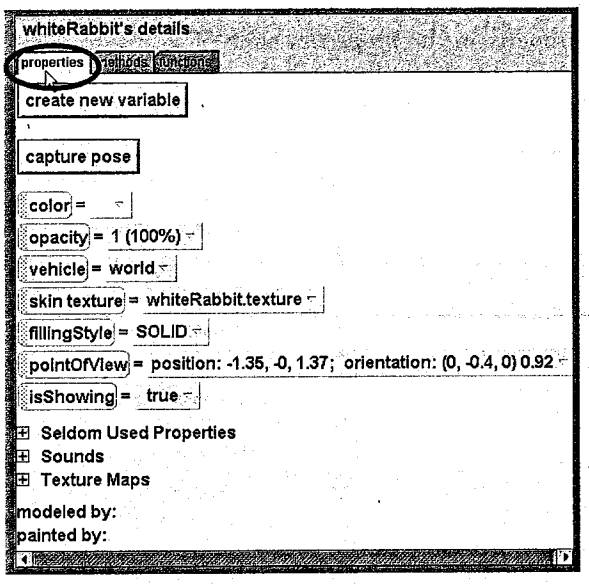


FIGURE 1-19 The *properties* pane

Here, we can see an object's properties, including its color, opacity, vehicle, skin texture, fill style, point of view (position + orientation), and whether or not it is showing.

The values of an object's properties determine the object's appearance and behavior when you run your program. Feel free to experiment with these settings, to see what they do. (You can always use Alice's **Undo** button if you make a mistake.) For example, if the White Rabbit's ghost were an object in the story, we might add a **whiteRabbit** to the world, and change its opacity to 30%, so that 70% of the light in the world passes through him. The result would be a ghostly translucent **whiteRabbit** in the program.

Changing A Property From Within A Program

When you set an object's property to a value within the *properties* pane, that property has that value when your program begins running, and will keep that value unless your program causes it to change. For example, suppose that we wanted the White Rabbit to magically disappear after he and Alice have greeted us, and Alice to then say, "Now where did he go this time?" We can easily elicit the required behavior from **aliceLiddel** by sending her a **say()** message; but how do we get the **whiteRabbit** to disappear before she says it?

There are actually two ways to accomplish this special effect. If we desire the White Rabbit to disappear instantly, we can do this by setting his **isShowing** property to **false** at the right place in the program. If we want him to disappear slowly (say,

over the course of a few seconds), we can do this by setting his **opacity** property to **0** at the right place in the program, and then modifying the statement's **duration** attribute to the required length of time. Either approach requires that we learn how to set one of the **whiteRabbit**'s properties, so we will use the latter approach, and leave the use of the first approach as an exercise.

To set an object's property to a different value at a specific point in the program, we click that property in the *properties* pane, drag it into the *editing area* until a green bar appears at the right spot in the program, and drop it. Alice will then display a drop-down menu of the options for the property's new value, as can be seen in Figure 1-20.

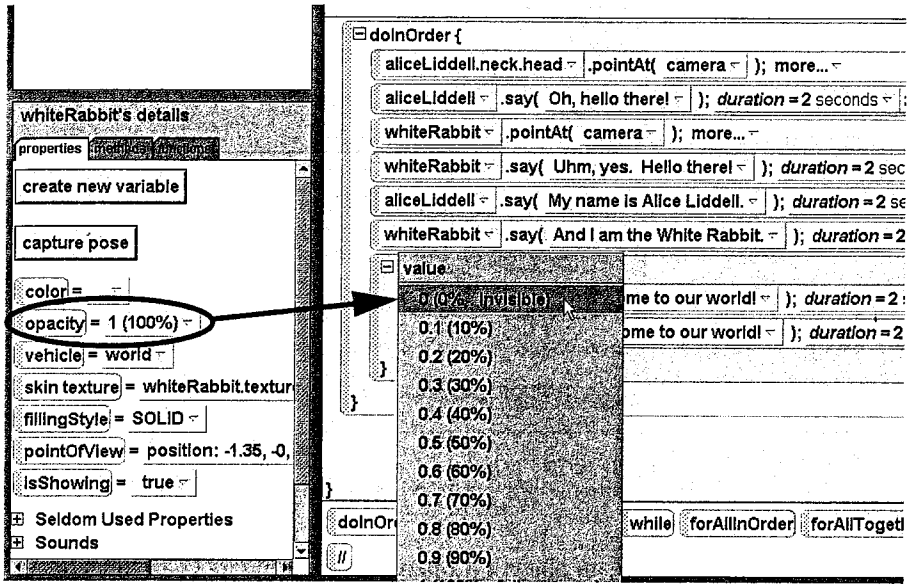


FIGURE 1-20 Setting a property by dragging it into the editing area

When we select a value from that menu, Alice inserts a new statement into the *editing area*. This statement sends a special **set()** message to the **whiteRabbit**, telling it to set its **opacity** property to the value we selected from the menu. By default, the **duration** of this **set()** message is one second, so to make the White Rabbit disappear more slowly, we set it to two seconds, yielding the statement shown in Figure 1-21.

```
    doTogether {
      aliceLiddell    .say( Welcome to our world!    ); duration = 2 seconds    fontSize = 30    more...
      whiteRabbit     .say( Welcome to our world!    ); duration = 2 seconds    fontSize = 30    more...
    }
    whiteRabbit    .set( opacity, 0 (0%)    ); duration = 2 seconds    more...
}
```

Fig. 21 The resulting world() method Alice generates to set a property

Adding the statement to make Alice say "Now where did he go this time?" is straightforward, and is left as an exercise.

1.5.2 The *methods* Pane

Click the *methods* tab of the *details area* and you will see the behavior-generating messages that you can send to the object selected in the *object tree*. Figure 1-22 shows some of the behavior-generating messages that are common to all Alice objects; a complete list is given in Appendix A.

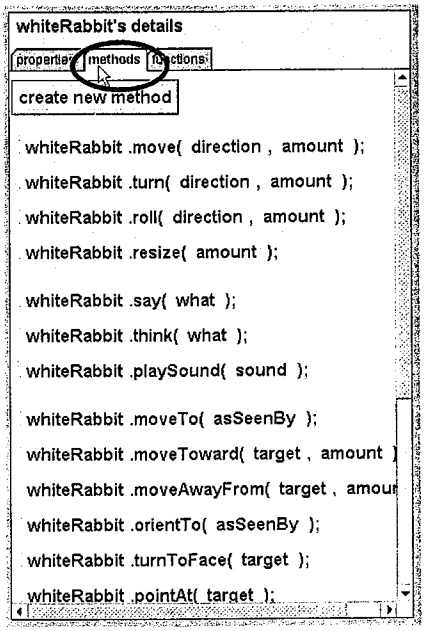


Fig. 22 The methods pane

These messages provide a rich set of operations that, together with the **doTogether** and **doInOrder** controls, let us build complex animations. Since we can send these messages to any Alice object, they allow us to build worlds containing talking animals, dancing trees, singing buildings, and just about anything else we can imagine!

The **resize()** message is especially fun, as it lets you make an object change size (for example, **resize(2)** to grow twice as big, or **resize(0.5)** to shrink to half size) as your program runs. The **resize()** message's **more...** menu includes a **dimension** choice that you can use to change an object's *width* (**LEFT_TO_RIGHT**), *height* (**TOP_TO_BOTTOM**), or *depth* (**FRONT_TO_BACK**), letting you create some interesting visual effects as your program runs.

In addition to these basic messages, some Alice objects respond to additional (non-basic) messages. For example, in the **People** folder of the Alice Gallery are tools called the **heBuilder** and **sheBuilder** that allow you to build custom male and female characters for your world. Each "person" built using one of these tools will respond to the additional messages. Figure 1-23 shows a person built using the **heBuilder**, whom we have renamed **bob**, and the non-basic messages that can be sent to such a person.

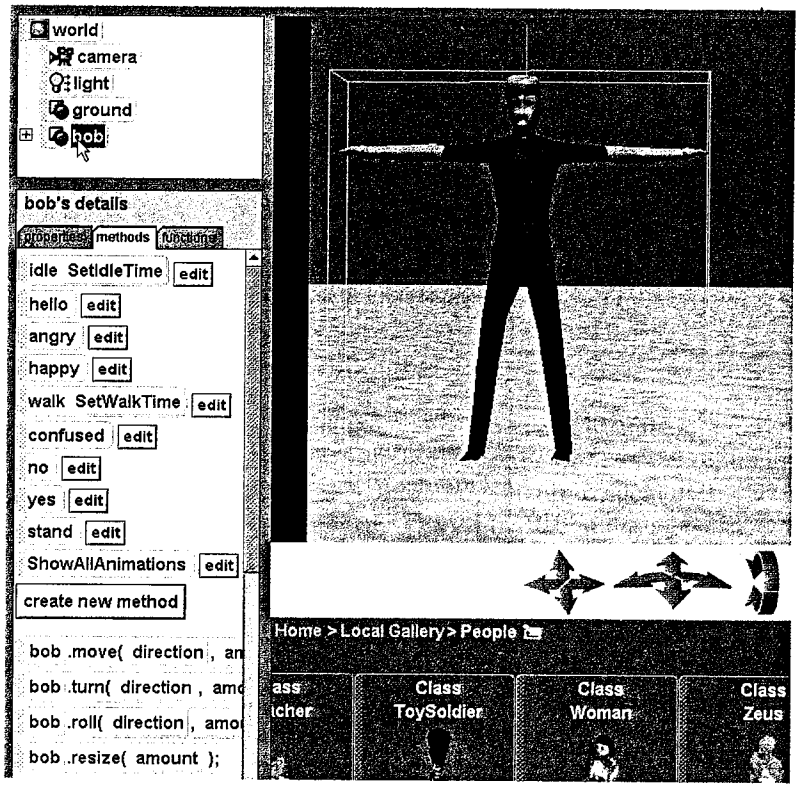


FIGURE 1-23 Non-basic methods

Other Alice classes (for example, **Frog**, **Monkey**, **Penquin**) provide different non-basic methods. To discover them, just add an object to your world and see what methods appear in the *details area*.

1.5.3 The *functions* Pane

If we click the *functions* tab in the *details area* as shown in Figure 1-24, we will see the list of functions or question messages that we can send to the object selected in the *object tree*.

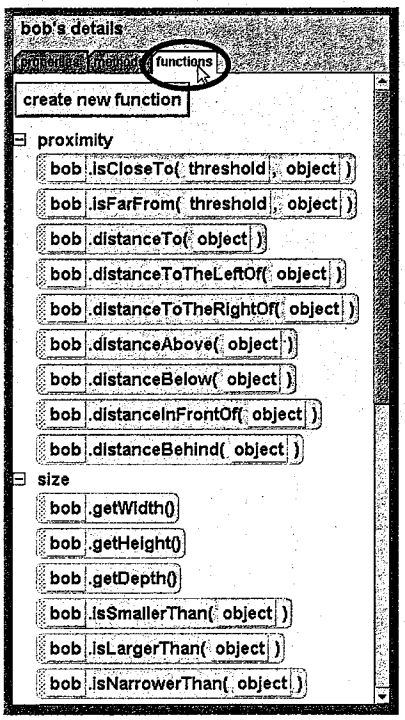


FIGURE 1-24 The *functions* pane

Functions are messages that we can send to an object to retrieve information from it. Where the *methods* tab provides standard behavior-generating messages, the *functions* tab provides a set of standard messages that we can send to an object to "ask it a question." The standard Alice functions let us ask an object about its:

- **proximity** to another object (that is, how close or how far the other object is)
- **size** (its *height*, *width*, or *depth*, and how these compare to another object)
- **spatial relation** to another object (*position* or *orientation* with respect to the other object)
- **point of view** (*position* and *orientation* within the world)
- **subparts**

Many of these standard functions refer to an object's bounding box (or one of its edges) that we saw in Section 1.4.3.

Alice also provides a different group of function messages we can send to the world. That is, if we select the **world** object and then the *functions* tab, Alice displays a group of world functions, some of which are shown in Figure 1-25.

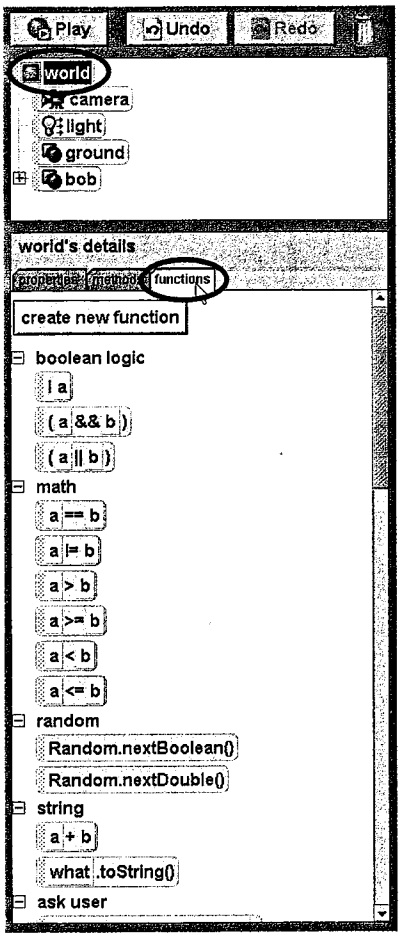


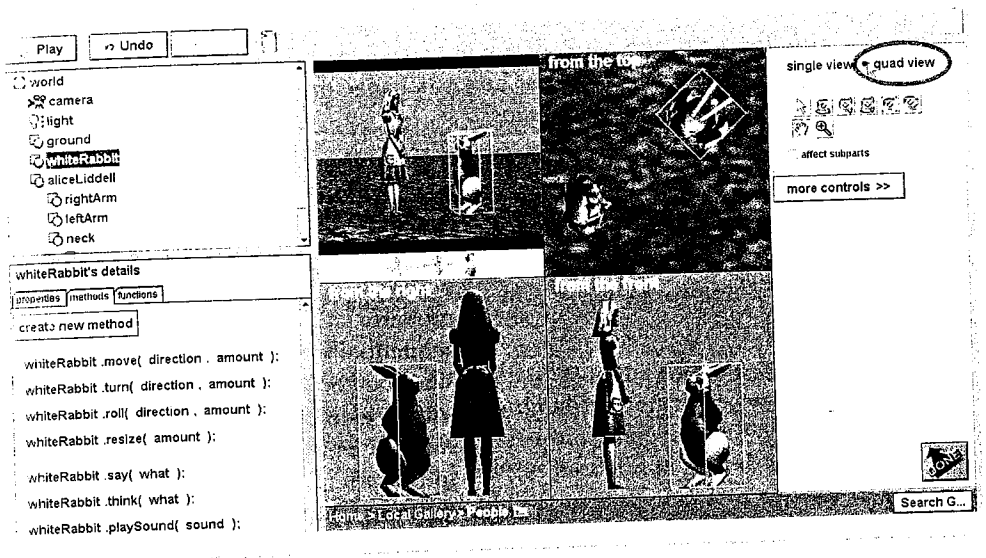
FIGURE 1-25 The *world functions* pane

We will see how to use these different kinds of messages in the coming chapters.

1.6 Alice Tip: Positioning Objects Using Quad View

In the Alice tutorials, we saw how the **ADD OBJECTS** button in the *world window* lets us navigate the Alice Gallery, locate classes, and use them to add objects to the world.

By default, **ADD OBJECTS** displays just the *world window*. However, trying to position two objects in close proximity to one another (for example, trying to position a person on the back of a horse) can be difficult using this single window, since it offers just one view. For such situations, Alice has the **quad view** that provides the *world window*, plus views from the top, right, and front of the scene. To use it, click the **quad view** radio button near the top of the window, as shown in Figure 1-26.



As can be seen above, the quad view provides two additional controls:

a "hand" control that lets you (within any of the views) drag the mouse to move the camera left, right, up, or down to view a different part of the scene

a "magnifying glass" control that lets you drag the mouse *down* to zoom the camera *in* on some detail of the scene, or drag *up* to zoom the camera *out* to see more of the scene

These additional controls are very useful when you shift to the quad view and the characters you wanted to see are nowhere to be seen. When this happens, just click the magnifying glass and then drag *up* within the view to zoom out until the characters become visible (probably very small), switch to the hand control and move the camera until the characters are centered, then switch back to the magnifying glass and drag *down* within the view to zoom back in.

1.7 Chapter Summary

- The *user story* describes the behavior of a computer program.
- *Storyboard-sketches* indicate the appearance of each of the program's scenes.
- *Transition diagrams* relate the storyboard-sketches to one another.
- A *flow* or *algorithm* provides a concise summary of the user story.
- The basics of using Alice include: how to add an object to a world; how to set its initial position, orientation, and pose; how to animate an object by sending it a message; how to select an object's subparts; how to change an object's properties; and how to send multiple messages simultaneously.

1.7.1 Key Terms

algorithm
bounding box
bug
class
concurrent execution
debugging
flow
function
message
method
object
orientation
point of view
position
pose
property
sequential execution
simultaneous execution
software design
software engineering
software implementation
software testing
statement
storyboard-sketches
user story

Programming Projects

1.1 Modify the world we created in Section 1.4 so that, after Alice and the White Rabbit introduce themselves, Alice tells the user she and the White Rabbit would like to sing a duet, after which they sing a simple song, such as *Mary Had A Little Lamb*. Have Alice and the White Rabbit sing alternate lines of the song.

Mary had a little lamb, *little lamb,* *little lamb.* *Mary had a little lamb* *it's fleece was white as snow.*	*And everywhere that Mary went,* *Mary went,* *Mary went.* *And everywhere that Mary went* *the lamb was sure to go.*
It followed her to school one day, *school one day,* *school one day.* *It followed her to school one day* *which was against the rules.*	*It made the children laugh and play,* *laugh and play,* *laugh and play.* *It made the children laugh and play* *to see a lamb at school.*

1.2 Finish the world we modified in Section 1.5.1, so that the movie ends with **aliceLiddell** saying, *Now where did he go this time?* Modify the world so that the **whiteRabbit** disappears instantly. Modify the world to make a **pop** sound when the **whiteRabbit** disappears.

1.3 If your computer has a microphone, modify the world we created in Section 1.4, using **doTogether** controls and **playSound()** messages to record voices for Alice and the White Rabbit, so that the user can hear what each character says instead of having to read it. Alter your voice for each character.

1.4 Using any two characters from the Alice Gallery, design and build a world in which one tells the other a knock-knock joke. (If you don't know any knock-knock jokes, see **www.knock-knock-joke.com**). Make your story end with both characters laughing.

1.5 Using the **heBuilder** or **sheBuilder** (under **People** in the **Local Gallery**), build a superhero named **Resizer**, who can alter his or her size at will. Build a world in which **Resizer** demonstrates his or her powers to the user by growing and shrinking. Make sure that **Resizer** tells the user what he or she is going to do before doing it.

1.6 Build a world containing one of the hopping animals (for example, a bunny or a frog). Write a program that makes the animal hop once, as realistically as possible (that is, legs extending and retracting, head bobbing, and so on). Bonus: Send your animal **playSound()** messages, so that the predefined sound **thud1** is played as it leaves the ground, **whoosh2** is played while it is in the air, and **thud2** is played when it lands.

1.7 Using the **heBuilder** or **sheBuilder** (under **People** in the **Local Gallery**), build a person. Place the person in a world containing a building. Using the **walk()**, **move()**, and **turn()** messages, write a program that makes him or her walk around the building.

1.8 Using the **heBuilder** or **sheBuilder** (under **People** in the **Local Gallery**), build a person. Then build a world containing your person and one of the items from the **Sports** section of the Gallery (for example, a baseball or a basketball). Write a program in which your person uses the item for that sport (for example, pitches the baseball or dribbles the basketball).

1.9 Choose one of your favorite movie scenes that contains just two or three characters. Use Alice to create an animated version of that scene, substituting characters from the Alice Gallery for the characters in the movie.

1.10 Write an original short story (10-20 seconds long), and use Alice to create an animated version of it. Your story should have at least two characters, and each character should perform at least five actions that combine to make an interesting story.

1.11 *Mules* is a silly (and confusing!) song with the lyrics shown below (sung to the tune of *Auld Lang Syne*). Build a world containing a horse (the closest thing in the Alice Gallery to a mule) and a person. Build a program that animates the person and horse appropriately while the person "sings" the lyrics to the song. For example, the person should point to the different legs (front or back) as he or she sings about them, move

to the back of the horse when the song calls for it, get kicked as the sixth line is sung, and so on.

On mules we find two legs behind, *and two we find before.* *We stand behind before we find,* *what the two behind be for!*	*When we're behind the two behind,* *we find what these be for —* *so stand before the two behind,* *behind the two before!*

Great thi

Weeks of

When do
a bagpipe

Objectiv

Upon cor

- ❑ Build
- ❑ Build
- ❑ Reus
- ❑ Use c
- ❑ Unde

Chapter 2
Methods

Great things can be reduced to small things, and small things can be reduced to nothing.

CHINESE PROVERB

Weeks of programming can save you hours of planning.

ANONYMOUS

When do you show the consequences? On TV, that mouse pulled out that cat's lungs and played them like a bagpipe, but in the next scene, the cat was breathing comfortably.

MARGE SIMPSON (JULIE KAVNER), IN "ITCHY AND SCRATCHY LAND," *THE SIMPSONS*

Objectives

Upon completion of this chapter, you will be able to:

- ❑ Build world-level methods to help organize a story into scenes and shots
- ❑ Build class-level methods to elicit desirable behaviors from objects
- ❑ Reuse a class-level method in multiple worlds
- ❑ Use dummies to reposition the camera for different shots within a scene
- ❑ Understand how an object's *position*, *orientation*, and *point of view* are determined

In the last chapter, we saw how to design and build computer programs. We also saw how Alice lets us build programs consisting of *statements*, in which we often send *messages* to *objects*. Finally, we saw that Alice provides us with a rich set of predefined messages that let us create programs to generate fun and interesting animations.

Alice's predefined messages provide an excellent set of *basic operations* for animation. However, for most Alice objects, these basic operations are *all* that are predefined. (The people we can create using the **heBuilder** and **sheBuilder** tools are unusual in providing methods beyond the basic ones.) The result is that for many of the behaviors we might want Alice objects to exhibit, there are no predefined methods to elicit those behaviors. For example, a horse should be able to walk, trot, and gallop, but there are no predefined **Horse** methods for these behaviors. A dragon or pterodactyl should at least be able to flap its wings (if not fly), but the **Dragon** and **Pterodactyl** classes do not provide methods for such behavior. A wizard should be able to cast a spell, but the **Wizard** class does not contain a **castSpell** message.

When an Alice class does not provide a method for a behavior we need, Alice lets us create a new method to provide the behavior. Once we have created the method, we can send the corresponding message to the object to elicit the behavior.

There are actually two quite different reasons for building your own methods. The first reason is to divide your story into manageable pieces to help keep it more organized. The second reason is to provide an object with a behavior it should have, but does not. In this chapter, we will examine both approaches. As we shall see, the motivation, thought process, and circumstances are quite different for these two different approaches.

2.1 World Methods for Scenes and Shots

As we mentioned in Chapter 1, films (and by extension, animations) are often broken down into **scenes**, with each scene making up one piece of the story. Scenes can be further broken down into **shots**, with each shot consisting of a set and whatever characters are in the shot, filmed from a particular camera position. When a film crew has finished one shot, they begin work on the next one. When all the shots for a particular scene are finished, the shots are combined to form the scene and that scene is done. Work then begins on the next scene.

Scenes and shots thus provide a logical and convenient way to break a big film project down into smaller, more manageable pieces. We can use the same approach in Alice. By organizing your user story into a series of scenes, and organizing each complex scene into a series of shots, you can work through the story shot by shot and scene by scene, without being overwhelmed by the size of the project. This approach — in which you solve a "big" problem by (1) breaking it into "small" problems, (2) solving each "small" problem, and (3) combining the "small" problem solutions into a solution to the "big" problem — is called **divide and conquer**.

2.1.1 Methods For Scenes

To illustrate how this approach can be used in Alice, suppose that we have a user story consisting of three scenes. When we first start Alice (even before we have added any objects to the world), we can organize our Alice program to reflect the scene structure of

our user story. To create a method for our first scene, we first select **world** in the *object tree*, make certain that the *methods* tab is selected in the *details area*, and then click the **create new method** button there, as shown in Figure 2-1.

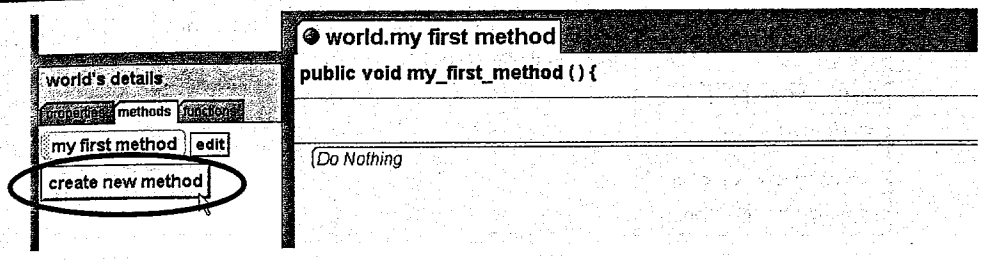


FIGURE 2-1 The **create new method** button

Clicking this box pops up a small **New Method** dialog box into which we can type the name we wish to give the new method. A method name should usually be (1) a *verb* or *verb phrase*, and (2) descriptive of what it does. Since we are creating a method to play the first scene, we will choose the name **playScene1**.

> Method names should begin with a lowercase letter and contain no spaces. If a name consists of multiple words, capitalize the first letter of each word after the first.

When we click the **New Method** dialog box's **OK** button, Alice does two things:

1. Alice creates a new pane in the *editing area*, labeled **world.playScene1**, containing an empty method definition for the **playScene1()** method.
2. Alice updates the *details area*, adding **playScene1** to the world's list of methods.

If you compare Figure 2-2 (below) to Figure 2-1, you will see both of these changes.

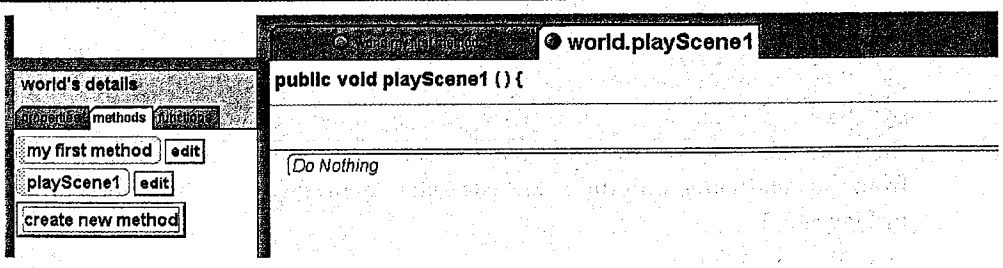


FIGURE 2-2 A new **playScene1()** method

One way to check that the method is working is to send a **say()** message to the world's **ground** object in **playScene1()**, as shown in Figure 2-3.[1]

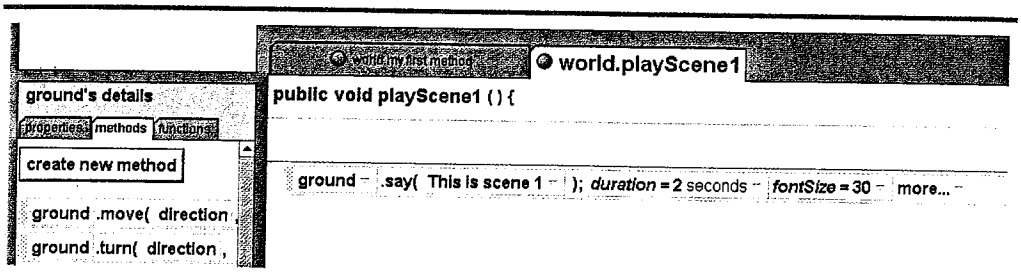


FIGURE 2-3 A simple method test

However, when we click Alice's **Play** button, the warning dialog box in Figure 2-4 appears.

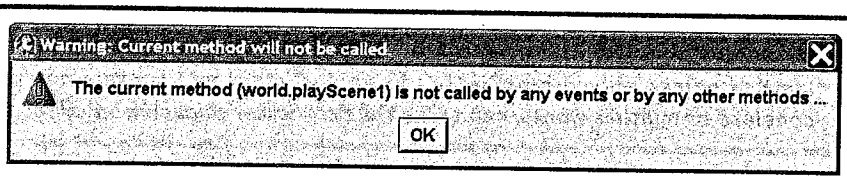


FIGURE 2-4 The "Method Not Called" warning

Alice is warning us that although we have *defined* a new method, there are no statements in the program that send the corresponding message. The problem is that **my_first_method()** is empty, and since that is where the program begins running, we need to send the **playScene1()** message from within **my_first_method()**.

After carefully reading the warning, we click the **OK** button to close that window, and then close the **World Running** window that appears. We then click on the tab for **my_first_method** in the *editing area*, drag a **doInOrder** control up from the bottom of the pane, click on **world** in the *object tree*, and then drag the **playScene1()** message from the *details area* into the **doInOrder** statement, giving us the (short) program shown in Figure 2-5.

1. Another approach is to have the method perform a **print()** statement, which is at the bottom of the *editing area*. When performed, this statement displays a message at the bottom of the **World Running** window, but it is awkward to view. (We had to resize the window and then scroll up to see the message.) The **print()** statement can also be used to view the value of a variable or parameter (see Chapter 3) when the statement is performed.

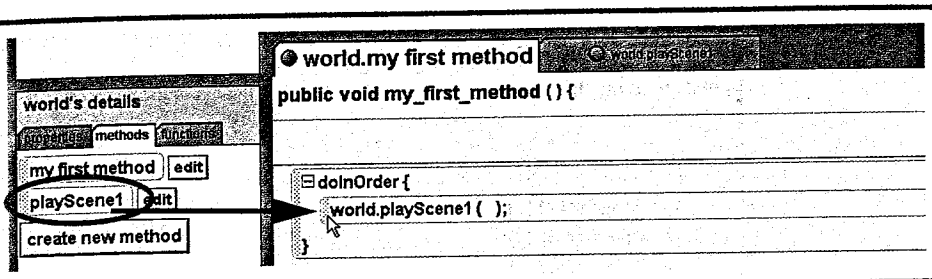


FIGURE 2-5 Sending `playScene1()` from `my_first_method()`

Now, when we click Alice's **Play** button, **`world.my_first_method()`** begins running. It sends the **`playScene1()`** message to **`world`**, which sends the **`say()`** message to the **`ground`**. If we've done everything correctly, we will see the **`ground`** "speak," as can be seen in Figure 2-6.

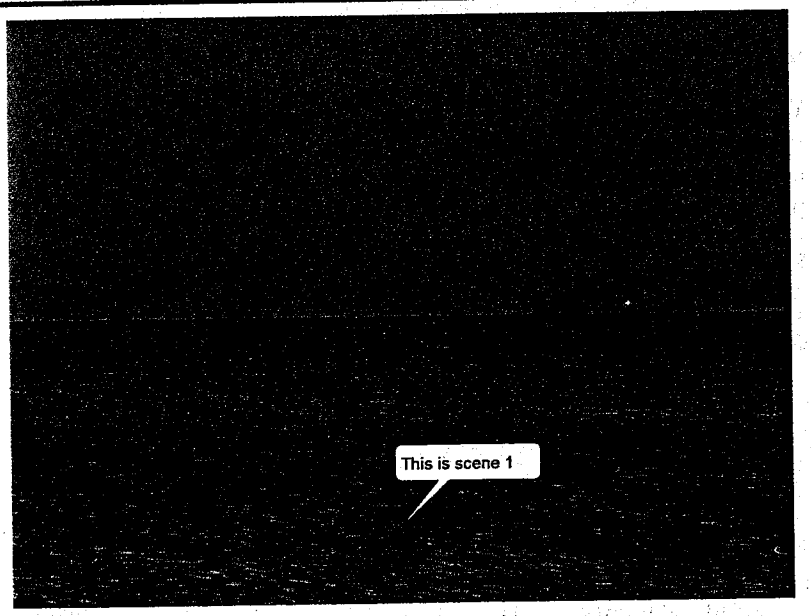


FIGURE 2-6 The **`ground`** speaks

Since the user story consists of three scenes, we can use this same approach to create new methods for the remaining two scenes, giving us the **`my_first_method()`** shown in Figure 2-7.

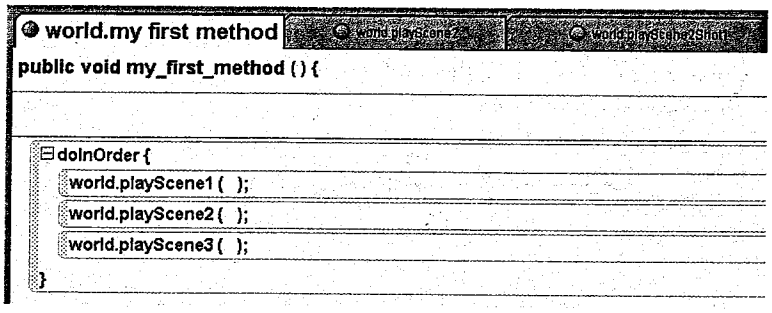


FIGURE 2-7 Three new `playScene()` methods

Inside each new **playScene** method, we can send the **ground** a distinct **say()** message (for example, naming that scene). Clicking Alice's **Play** button should then display those messages in order. This is a simple way to test that the new methods are working properly. When we are confident that all is well, we can begin adding statements to **playScene1()** to perform the first scene, adding statements to **playScene2()** to perform the second scene, and so on.

2.1.2 Methods For Shots

We have just seen how a big, complicated project can be broken down into smaller, easier-to-program scenes. However in a *very* big project, a scene itself may be overwhelmingly complicated! In such situations, complex scenes can be divided into simpler (easier-to-program) **shots**. One good rule of thumb is:

> **If you must use the scroll bar to view all the statements in a scene method, divide it into two or more shot methods.**

The idea is that long methods are complicated, and therefore more error prone. If you keep your methods short and sweet, you'll be less likely to make a mistake — and if you do make one, it will be easier to find, since you won't have to scroll back and forth through lots of statements.

To illustrate this idea, suppose that the first scene is reasonably simple, and can be implemented in a method that requires no scrolling. However, suppose that the second scene is quite complicated, and we estimate that building it would require four or more screenfuls of statements. We can use an approach similar to what we did in Section 2.1.1 to create a method for each shot. Being systematic, we might name these methods **playScene2Shot1**, **playScene2Shot2**, **playScene2Shot3**, and **playScene2Shot4**.

As before, we select **world** in the *object tree*, and then click the **create new method** button in the *details area*. When asked to name the first method, we name it **playScene2Shot1**. As before, Alice (1) updates the *editing area* with a new pane containing an empty definition for the new method, and (2) adds the new method to the list of methods in the *details area*. To test that it works, we can again send the **ground** object a **say()** message, as shown in Figure 2-8.

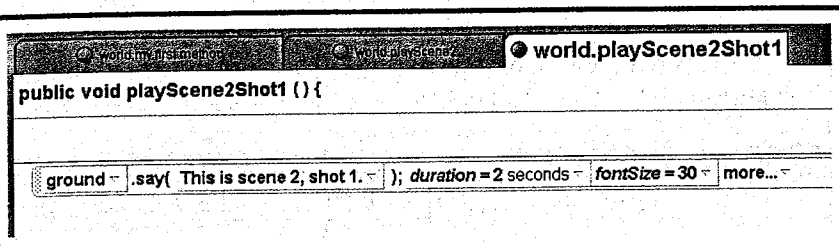


FIGURE 2-8 Testing a shot method

We can then select **world** in the *object tree*, click on the **world.playScene2** tab in the *editing area*, delete the **ground.say()** message from **playScene2()**, drag a **doInOrder** control into **playScene2()**, and finally drag the **playScene2Shot1()** message from the *details area* into the **doInOrder** statement, yielding the definition found in Figure 2-9.

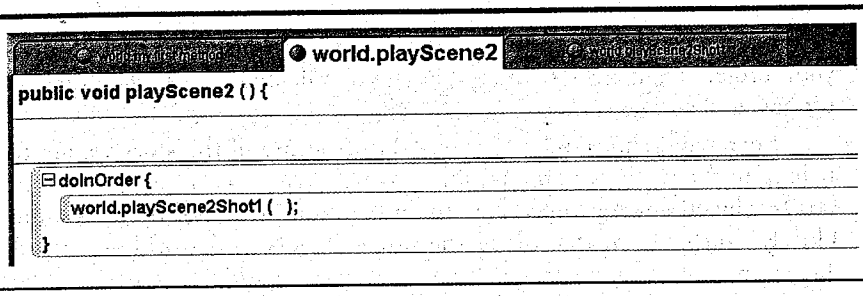


FIGURE 2-9 Calling a shot method from a scene method

If we repeat this for each of the remaining shots in the scene, we get the definition shown in Figure 2-10.

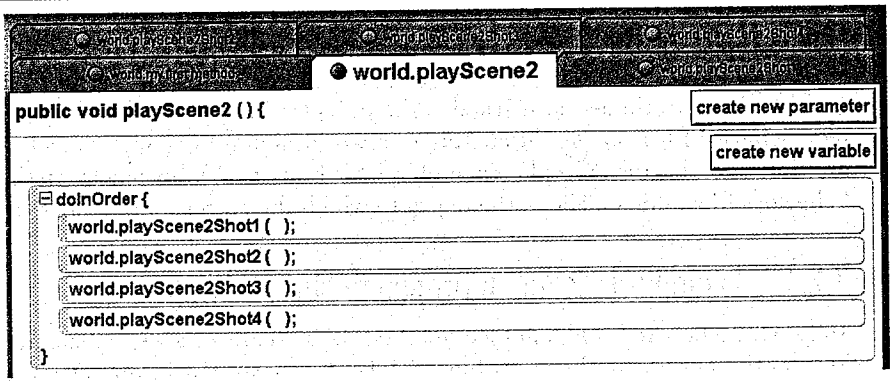


FIGURE 2-10 A scene method built from shot methods

Now we can add statements to each of the four shot methods to produce the animation required for that shot. When each is complete, we will have a complete animation for Scene 2!

If we were to draw a diagram of the structure of our program, it would be as shown in Figure 2-11.

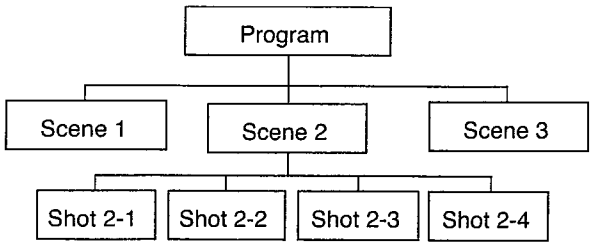


FIGURE 2-11 Structure diagram

Such a diagram or program can have as many pieces and levels as necessary to make your project manageable. If a shot is complicated, it can be further subdivided into **pieces**, and so on.

Scene and shot methods reflect the structure of the story we are telling, and hence belong to the world we are building. As such, they are properly stored in the *object tree*'s **`world`** object, since it represents the program as a whole. If you examine the *object tree* closely, you will see that all of the other objects in a world — including the **`camera`**, **`light`**, **`ground`**, and anything else we add to the world — are parts of the **`world`**. Because we store scene and shot messages in the **`world`** object, these messages must be sent to it, as we see in Figure 2-10.

In Alice, methods stored in the **`world`** are called **world methods**, because they define a message that is sent to the **`world`**. A method that affects the behavior of *multiple objects* (like a scene) should be defined as a world method.

2.2 Object Methods for Object Behaviors

An alternative to the world method is the **object method**, which is used to define a complex behavior for a *single object*. Where a world method usually controls the behavior of multiple objects (for example, each character in a scene), an object method controls the behavior of just one object — the object to which the corresponding message will be sent.

2.2.1 Example 1: Telling a Dragon to Flap Its Wings

To illustrate how to build an object method, let's create a new story starring a dragon who lives in the desert, as shown in Figure 2-12. Suppose that in one or more of this story's scenes, the dragon must flap its wings. Wing-flapping is a reasonably complex behavior, and it would be convenient if we could send a **`flapWings()`** message to a **`dragon`** object,

but class **Dragon** does not provide a **flapWings()** method. In general, the following rule of thumb should be used in defining methods:

> *Methods that control the behavior of a single object should be stored in that object.*

From another perspective, a dragon is responsible for controlling its wings, so a **flapWings()** message should also be sent to a **dragon**. To do so, the **flapWings()** method must be stored in the **dragon** object. Conversely, it makes no sense to send the **world** a **flapWings()** message (since it has no wings to flap), so **flapWings()** should *not* be defined as a world method.

Assuming that we have added a **dragon** to the **world**, we can define a **dragon** method named **flapWings()** as follows. We first select **dragon** in the *object tree*, and click the *methods* tab in the *details area*. Above the list of **dragon** methods, we see the **create new method** button, as can be seen in Figure 2-12.

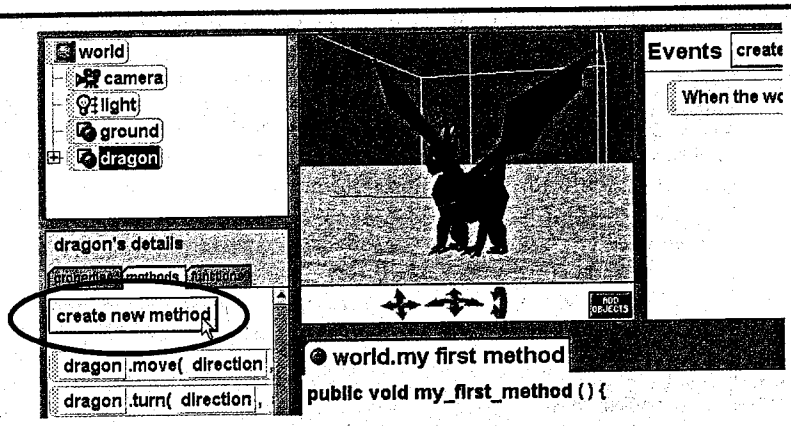


FIGURE 2-12 Creating a new class-level method

As we have seen before, clicking this button generates a dialog box asking us for the name of the new method, in which we can type **flapWings**. Alice then (1) creates a new tabbed pane in the *editing area* labeled **flapWings** containing an empty definition of a **flapWings()** method, and (2) creates an entry for the new method in the *details area* above the **create new method** button, as shown in Figure 2-13.

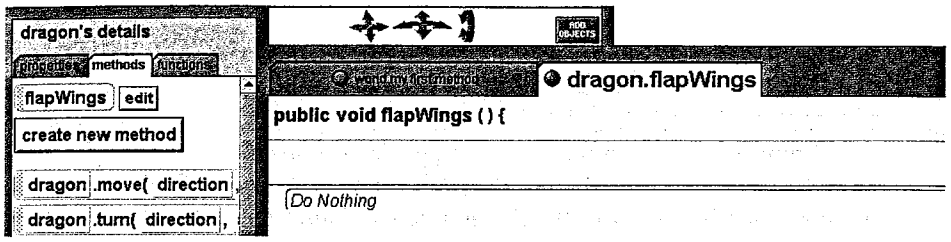


FIGURE 2-13 The empty `flapWings()` method

We can then fill this empty method definition with the statements needed to elicit the desired wing-flapping behavior. Figure 2-14 shows one way we might define such behavior, by sending **`roll()`** messages to each of the **`dragon`**'s wings.

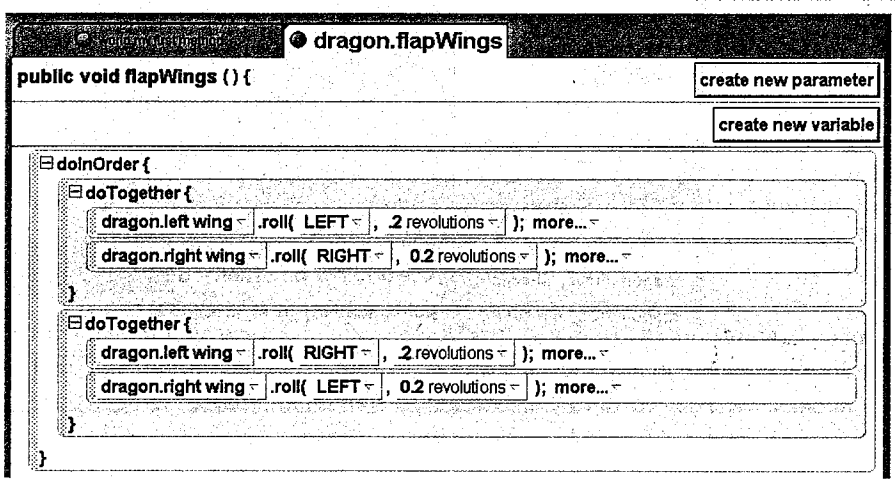


FIGURE 2-14 One way to define a `flapWings()` method

Comments

It may take you some time to figure out why each statement that appears in Figure 2-14 is there. Puzzling out the purpose of statements consumes time that could be better spent on other activities.

To help human readers understand why a method's statements are there, good programmers insert **comments** into their methods to explain the purpose of tricky statements. Comments are ignored by Alice, so you can write whatever is needed by way of explanation.

To add a comment to a method in Alice, click on the *comment control* at the bottom of the *editing area*, drag the control upwards until the green bar appears above the statements you want to explain, and then drop the control, as shown in Figure 2-15.

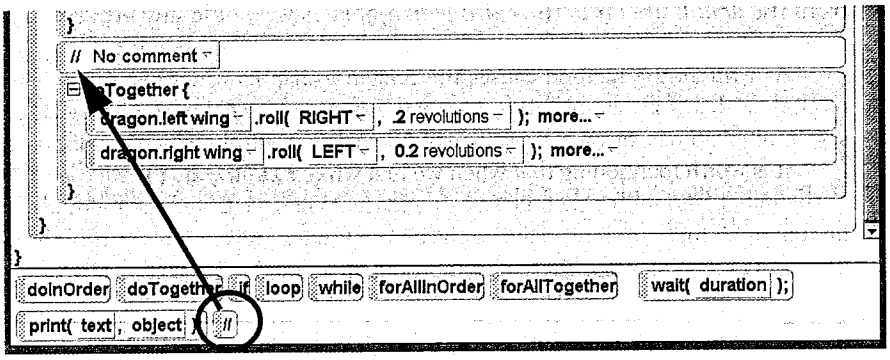


FIGURE 2-15 Dragging a comment

When you drop the comment, Alice gives it a **`No comment`** label. To edit a comment's explanation, you can either double-click its text, or click its list arrow and choose **`other`** from the menu that appears. Figure 2-16 shows a final, commented version of the **`flapWings()`** method.

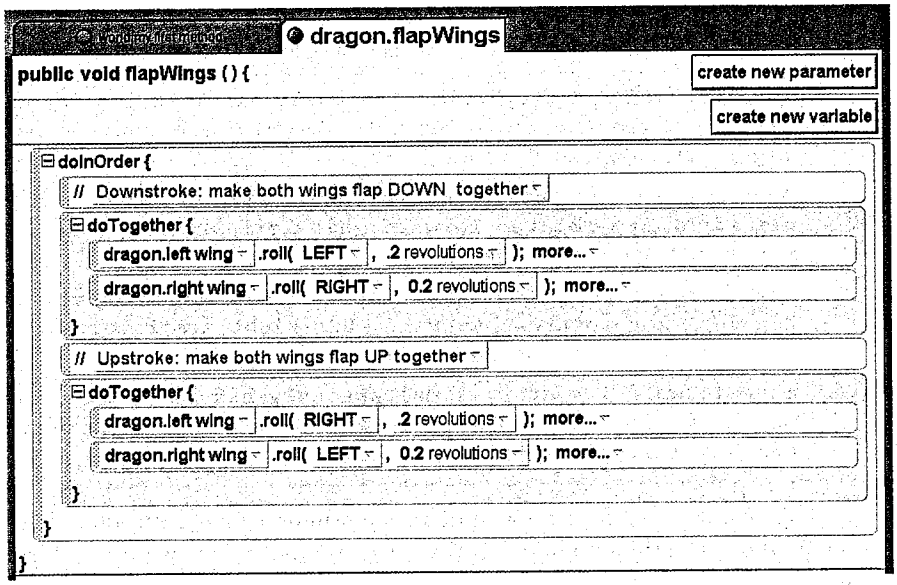


FIGURE 2-16 Final **`flapWings()`** method

Testing

To test the **flapWings()** method, we switch to **my_first_method()** — or to a world method that **my_first_method()** invokes, such as the scene method in which the dragon flaps its wings — select **dragon** in the *object tree*, and then drag **flapWings()** from the *details area* into the *editing area*, just as we would any other **dragon** method. We then click Alice's **Play** button and watch the dragon flap its wings!

As written, the method causes the dragon to flap its wings just once. If we need it to flap more than that (for example, to fly across the sky), we can either send it the **flapWings()** message multiple times, or we can use one of Alice's *loop* controls, which are discussed in Chapter 4.

It is worth mentioning that when we first wrote **flapWings()**, we tried **1/4 revolution** as the initial amount for each **roll()** message. When we tested the method, that seemed like a bit too much motion; so we reduced the amount to **0.2** revolutions. Part of the "art" of Alice programming is testing with different values until an animation is visually satisfying.

2.2.2 Example 2: Telling a Toy Soldier to March

Suppose we have a different story,[2] containing a scene in which a toy soldier is to march across the screen. There is a **ToySoldier** class in the Alice Gallery; unfortunately, this class contains no **march()** method. So let's build one! We can do so by defining an object method named **march()** in the **toySoldier**.

Design

It is always a good idea to spend time designing before we start programming, especially with a complex behavior like marching. If we think this behavior through step-by-step (Ha, ha! Get it? Step? Marching?), we might break it down into the following algorithm:

ALGORITHM 2-1 Behavior: The ToySoldier should:

1. **move forward 1/4 step; simultaneously his left leg rotates forward, his right leg rotates backward, his left arm rotates backward, his right arm rotates forward;**
2. **move forward 1/4 step; simultaneously his left leg rotates backward, his right leg rotates forward, his left arm rotates forward, his right arm rotates backward;**
3. **move forward 1/4 step; simultaneously his right leg rotates forward, his left leg rotates backward, his right arm rotates backward, and his left arm rotates forward;**
4. **move forward 1/4 step; simultaneously his right leg rotates backward, his left leg rotates forward, his right arm rotates forward, and his left arm rotates backward.**

We can figure out just how much each arm or leg needs to rotate later, when we test the method. The thing to notice is that, because the actions within each step are occuring simultaneously, *Steps 1 and 4, and Steps 2 and 3 describe exactly the same behaviors*! For

2. Whenever we begin a new story or change to a different story, you will need to save your current world (using **File -> Save World**), and then open a new world (using **File -> New World**).

lack of better names, we might call Step 1 *marchLeft* and call Step 2 *marchRight*. If we were to write methods for these two steps, then the algorithm simplifies to this:

```
1 marchLeft;
2 marchRight;
3 marchRight;
4 marchLeft.
```

To move the soldier forward, we send him the **move()** message. To make his arms and legs rotate appropriately, we send **turn()** messages to his subparts. After some trial-and-error to find good **move()** and **turn()** distances, we get a definition like the one shown in Figure 2-17.

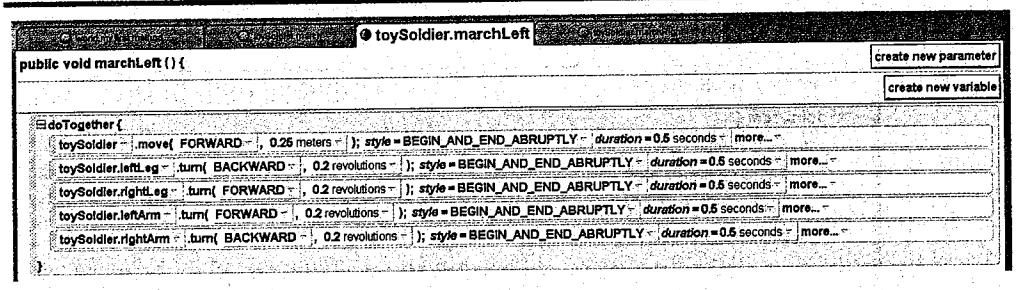


FIGURE 2-17 The **marchLeft()** method

The **marchRight()** method is similar, but with the behaviors reversed, as given in Figure 2-18.

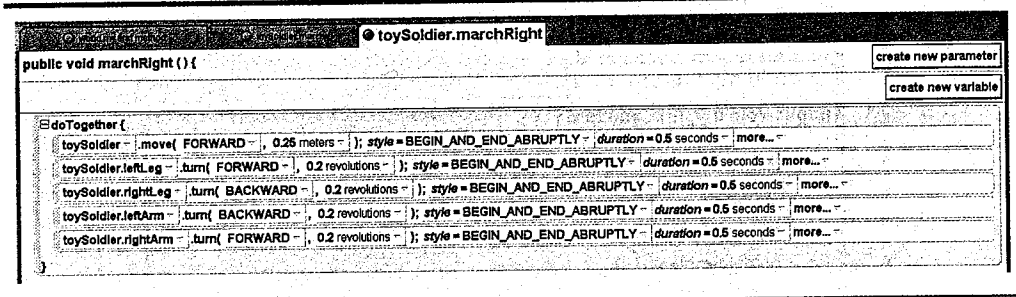


FIGURE 2-18 The **marchRight()** method

You may be wondering why in Figure 2-17 and Figure 2-18 we set each **move()** and **turn()** message's **style** attribute to **BEGIN_AND_END_ABRUPTLY**. The reason is that using this **style** smooths out the animation and makes it less "jerky." More precisely, by using this **style**, the first sending of **marchLeft()** will end abruptly, and since **marchRight()** begins abruptly, it will commence *immediately*. When it ends (abruptly), the second

sending of **marchRight()** will begin without delay. And when it ends (abruptly), the second sending of **marchLeft()** will begin with no delay.

> If you find that your animations are moving in a "jerky" fashion, try setting the **style** of the animation's messages to **BEGIN_AND_END_ABRUPTLY**.

With these two methods in place, the **march()** method is quite simple, as shown in Figure 2-19.

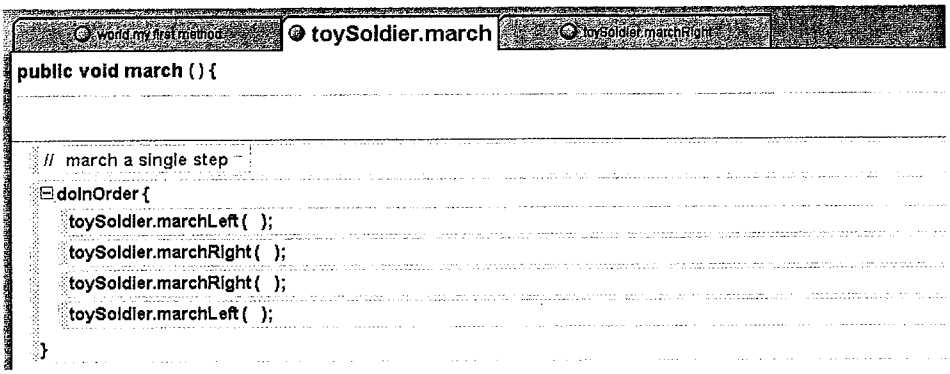


FIGURE 2-19 The **march()** method

To test the **march()** method, we can send the **toySoldier** the **march()** message, either from the scene in which it is needed or from **my_first_method()**. Figure 2-20 shows the latter.

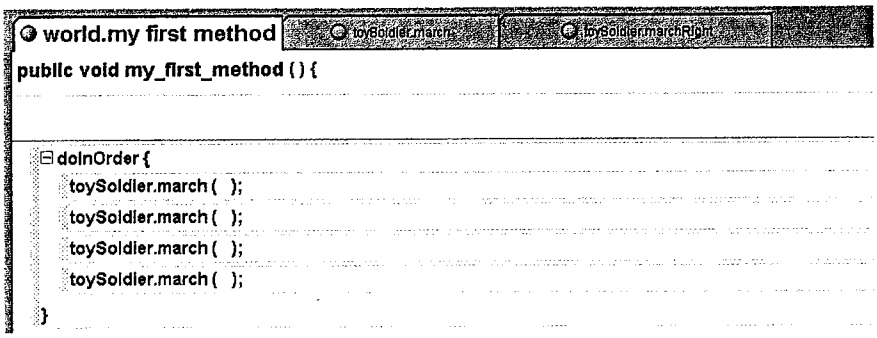


FIGURE 2-20 Testing the **march()** method

Now, when we click Alice's **Play** button, the soldier marches across the scene!

2.3 Alice Tip: Reusing Your Work

If you right-click on a statement, Alice displays a menu containing a **make copy** choice. Selecting this choice duplicates that statement. For example, in creating the program in Figure 2-20, we dragged **toySoldier.march();** into the *editing area* just once, and then used this right-click **make copy** mechanism to rapidly duplicate that statement three times.

This mechanism can also save time when you need to do similar, but not identical, things in a method. For example, to build the **flapWings()** method shown in Figure 2-16, we first built the top **doTogether** statement that makes the dragon's wings move down. We then made a copy of that statement, and in that copy, reversed the direction of the **roll()** messages, changing **LEFT** to **RIGHT** in the first message, and **RIGHT** to **LEFT** in the second message. This was much easier (and faster) than building the bottom **doTogether** statement from scratch.

In the rest of this section, we examine two other ways you can reuse existing work.

2.3.1 Using the Clipboard

The right-click **make copy** mechanism is useful when you have a statement that you want to duplicate *within* a particular method. But suppose you have written a statement in one method that you want to reuse in a different method.

For example, suppose you are programming a scene method, and producing the desired behavior takes more statements than anticipated. Viewing the method requires you to scroll back and forth, so you decide to break the scene up among two or more shot methods. How can you move the statements already in your scene method into a new (empty) shot method?

The answer is the **Alice clipboard**, located above the *events area* in the upper-right corner of the screen. From the *editing area*, you can drag any statement onto the clipboard and Alice will store a copy of it there for you, as shown in Figure 2-21.

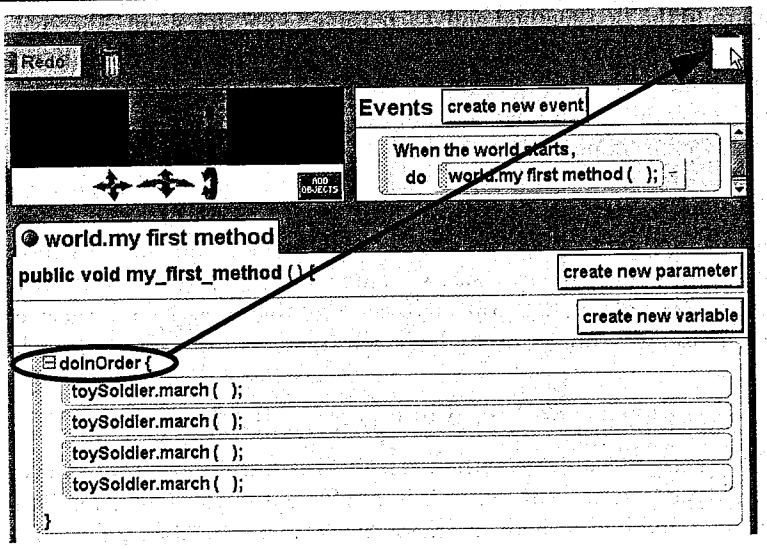


FIGURE 2-21 Dragging a statement to the clipboard

If we then create a new method (that is, for a scene or shot), we can drag the statement from the clipboard and drop it into that method, as shown in Figure 2-22.

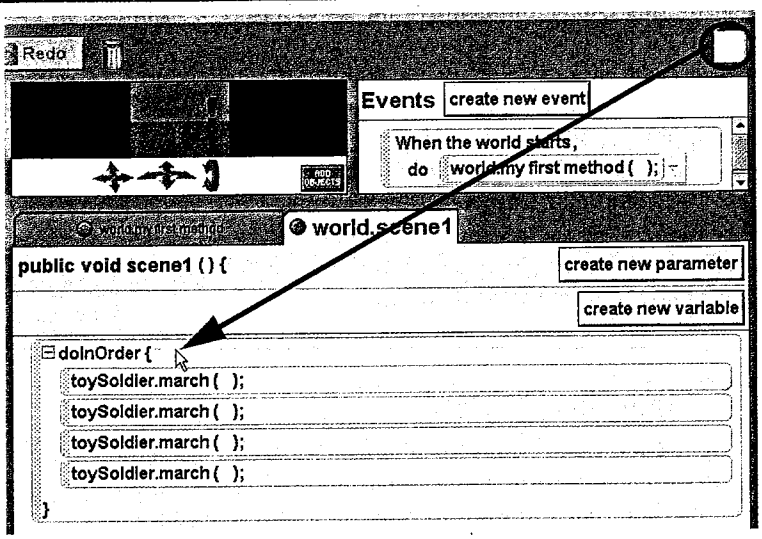


FIGURE 2-22 Dragging a statement from the clipboard

When we drag a statement from the clipboard and drop it in the *editing area*, Alice *copies* the statement from the clipboard. That is, a statement copied to the clipboard remains there until we replace it by dragging another statement onto the clipboard. In this case — where we are *moving* a statement from one method to another — we must then return to the first method and *delete* the original statement; Alice will not delete it for us.

The clipboard holds just one statement, whether it be a **`doInOrder`**, a **`doTogether`**, a message to an object, or one of the other Alice statements we will see later. If you find yourself in a situation where you need to store multiple statements, you can tell Alice to display more clipboards by selecting the **`Edit -> Preferences`** menu choice, selecting the **`Seldom Used`** tab, and then increasing the number of clipboards as necessary.

The ability to copy a statement to and from the clipboard is one advantage of placing all of a method's statements within a **`doInOrder`** statement. If for any reason we should later want to copy the method's statements into another method, we can just drag the outer **`doInOrder`** statement to the clipboard and then drag it from there into the other method. Otherwise, we would have to drag each statement to and from the clipboard individually.

2.3.2 Reusing an Object in a Different World

Writing a good object method takes time and effort. If you develop an object in one world, you may want to reuse it in a different world. For example, if we have spent time writing a method to make a dragon flap its wings — or a soldier to march, or a horse to gallop, or whatever — and we want to reuse that same character with the same behavior in a different world, we do not want to have to redo the work all over again.

Thankfully, Alice lets us reuse an object in different worlds. To do so, follow these steps:

1. In the world containing the original object, right-click it and **rename** it, choosing a new name that describes how it differs from the old object (for example, **marchingSoldier**).
2. Right-click the object again, but this time choose **save object...** Use the **Save Object** dialog box to save the object to your desktop (or anywhere you can find easily).
3. Open the world where you want to reuse the object.
4. Choose **File -> Import...** In the dialog box that appears, navigate to your saved object, select it, and click the **Import** button.

Let's go through these steps using the dragon we modified in Section 2.2.1.

1. First, we give the **dragon** a new name by right-clicking it, choosing **rename**, and then giving it a descriptive name, as shown in Figure 2-23.

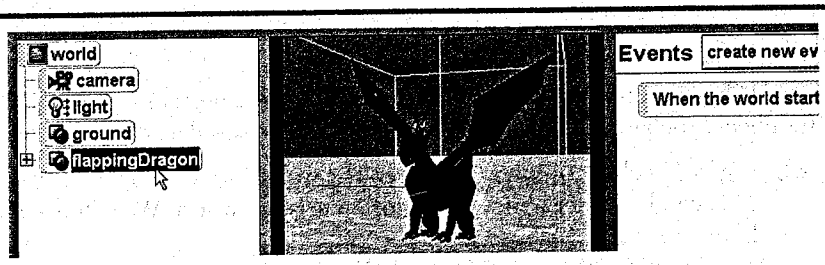


FIGURE 2-23 Renaming an object

2. We right-click again, but, as shown in Figure 2-24, this time we choose **save object...** from the menu that appears:

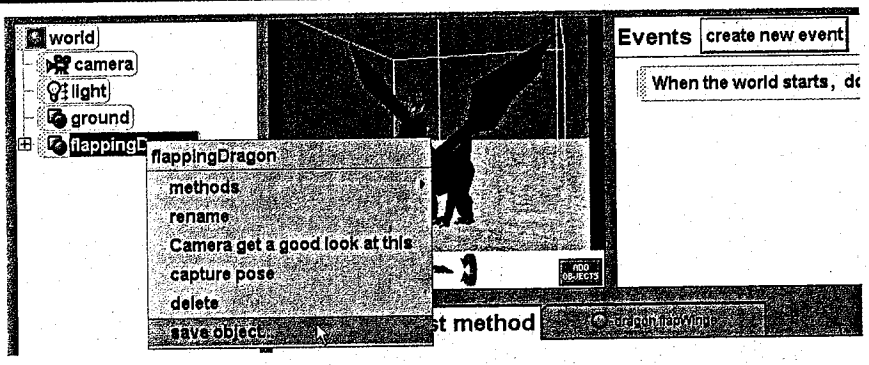


FIGURE 2-24 Saving an object

As shown in Figure 2-25, a **Save Object** dialog box appears, with which we navigate to where we want to save the object (for example, the **Desktop**).

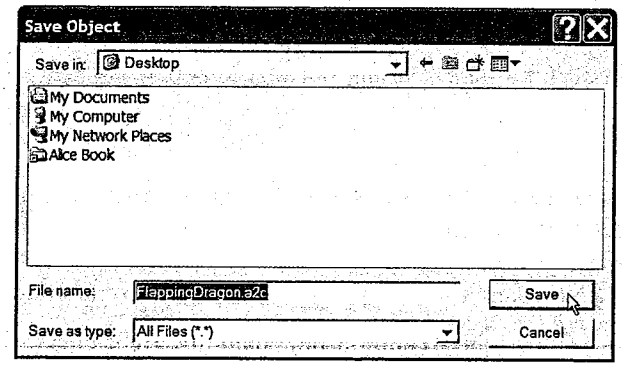


FIGURE 2-25 Saving an object

When we click the **Save** button, Alice saves the object in a special *.a2c* file (*a2c* stands for *a*lice-2.0-class). In our example, the file will be saved as **FlappingDragon.a2c**.[3]

3. Using Alice's **File** menu, we open the world into which we want to reuse the object. This can be either a new world, or an existing world. We will use a new, snowy world here.
4. With the new world open, we choose **Import...** from the **File** menu, as shown in Figure 2-26.

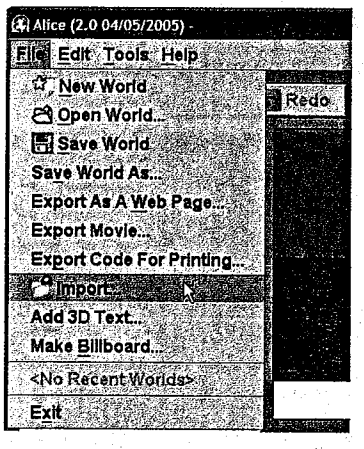


FIGURE 2-26 Importing an object

3. The first letter of a *class* is capitalized, to help distinguish it from an *object*, whose first letter is lowercase.

In the dialog box that appears, we navigate to where we saved the object (for example, the **`Desktop`**), select the *.a2c* file we saved in Step 2, and click the **`Import`** button, as shown in Figure 2-27.

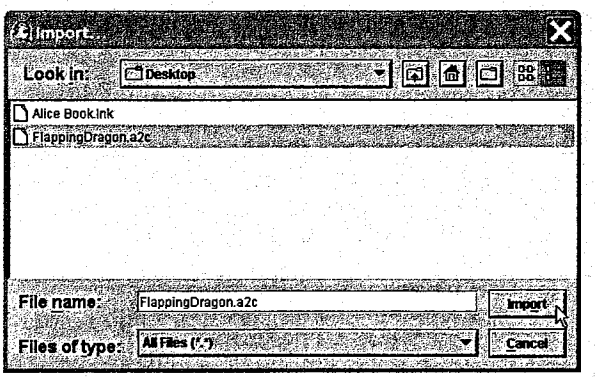


FIGURE 2-27 The import dialog box

Voila! the new world contains a copy of the **`flappingDragon`**, as shown in Figure 2-28!

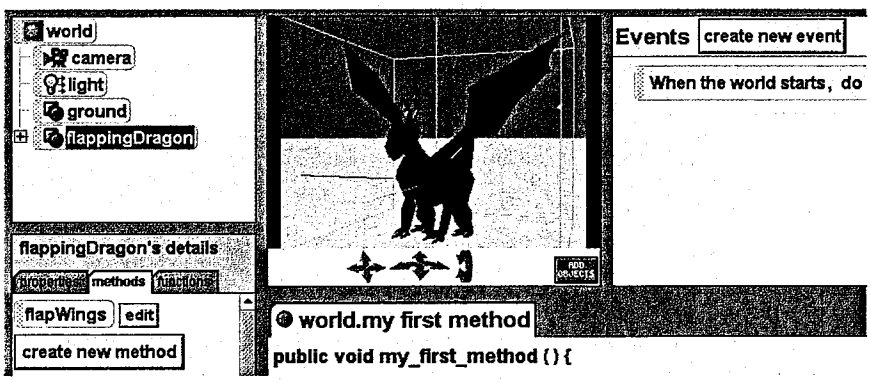


FIGURE 2-28 A reused object

As shown in Figure 2-28, the dragon in this new world includes the **`flapWings()`** method.

By saving an object from one world, and importing it into another, Alice provides us with a means of reusing the work we invest in building object methods.

2.4 Alice Tip: Using Dummies

As we mentioned earlier, scenes are often divided into shots, with each shot being a piece of a scene filmed with the camera in a different position. We have also seen that Alice places a **camera** object in every world. This raises the question: How do we move the camera from one position to another position within a scene?

Because the **camera** is an Alice object, any of the basic Alice messages from Appendix A can be sent to it. We could thus use a set of simultaneous **move()**, **turn()**, and other motion-related messages to shift the camera between shot methods. However, getting such movements right requires lots of trial and error and gets tedious. Thankfully, Alice provides a better way.

2.4.1 Dummies

The better way is to use a special Alice object called a **dummy**. A dummy is an invisible *marker* in your world that has a position and an orientation. The basic idea is as follows:

1. Manually move the camera (using the controls below the *world window*) until it is in the position and orientation where you want it for a given shot.
2. Drop a dummy at the **camera**'s position. This dummy has the **camera**'s point of view.
3. Rename the dummy something descriptive (for example, the number of the scene and shot).
4. At the beginning of the method for that shot, send the **camera** the **setPointOfView()** message, with the dummy as its target.

Let's illustrate these steps with a new example. Suppose that we have a user story whose second scene begins as follows:

> Scene 2: The Wizard and the Trolls.
>
> Shot 1: Wide-angle shot of a castle, with three trolls in the foreground. The leader of the trolls says he wants to destroy the castle. The other two trolls agree. Before they can act, a wizard materializes between them and the castle.
>
> Shot 2: Zoom in: a half-body shot of the wizard. He cries, "YOU SHALL NOT PASS!"
>
> Shot 3: Zoom out: the same wide angle shot as before. The trolls turn to the wizard ...

We can start by creating a new world, and creating empty world methods named **playScene2Shot1()**, **playScene2Shot2()**, **playScene2Shot3()**, and **playScene2()**, with this latter method invoking the first three. We then invoke **playScene2()** from **my_first_method()**, as we did in Section 2.1. We can then add the **castle**, **wizard**, and **trolls** to build the scene, as shown in Figure 2-29.

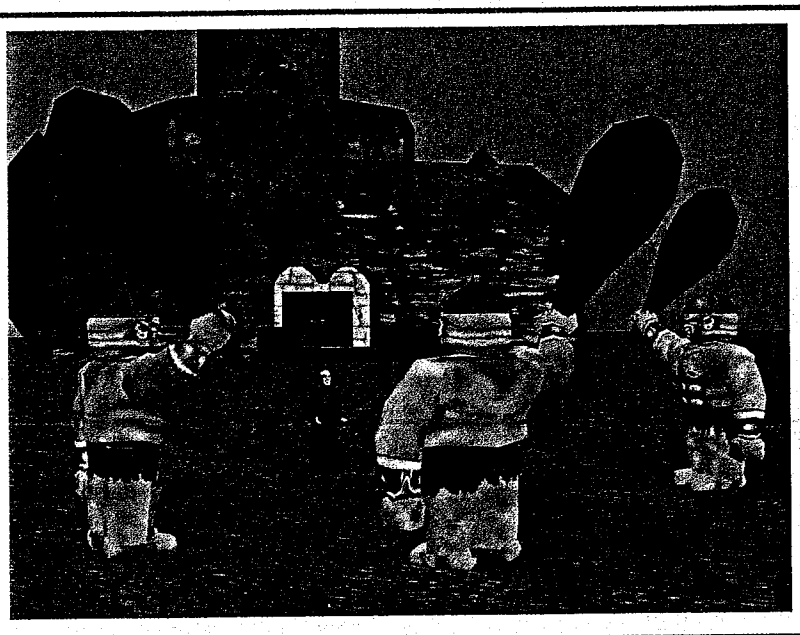


FIGURE 2-29 The set of "The Wizard and the Trolls"

With the *Add Objects* window still open, we click the **more controls** button, as shown in Figure 2-30.

FIGURE 2-30 The **more controls** button

Among the additional controls this button exposes is the **drop dummy at camera** button, as can be seen in Figure 2-31.

FIGURE 2-31 More controls

When we click this button, Alice adds a **dummy** object — an invisible marker — to the world, with the same position and orientation (point of view) as the **camera**. The first time we click this button, Alice creates a new folder named **Dummy Objects** in the *object tree*, in which all dummies are stored. If we open this folder (Figure 2-32), we can see the **Dummy** object inside it.

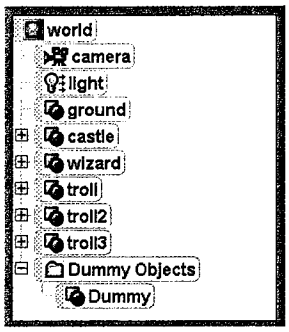


FIGURE 2-32 A dummy object

Since the name **Dummy** is not very descriptive, we can right-click on the object, select **rename** from the menu that appears, and rename the dummy **scene2Shot1**, as shown in Figure 2-33.

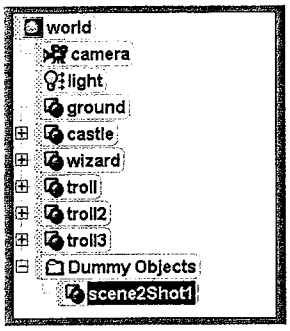


FIGURE 2-33 A renamed dummy

By doing so, we will know exactly which scene and shot this dummy is for, and not confuse it with the dummies we create for other scenes and shots.

Now that we have a dummy in place for the first shot, the next step is to manually position the **camera** where we want it for the second shot, using the controls beneath the *world window*. Using these controls, we can zoom in until we get a nice half-body shot of the wizard, leaving space above his head for his dialog-balloon to appear. See Figure 2-34.

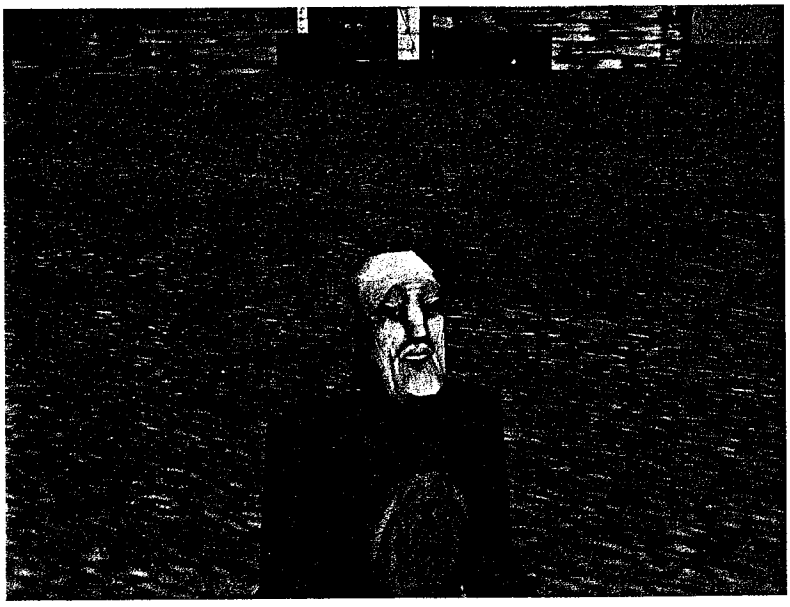


FIGURE 2-34 A half-body shot of the wizard

When we have the **camera** just where we want it, we again press the **drop dummy at camera** button to drop a second dummy at the **camera**'s current position. As before, we rename it, as shown in Figure 2-35.

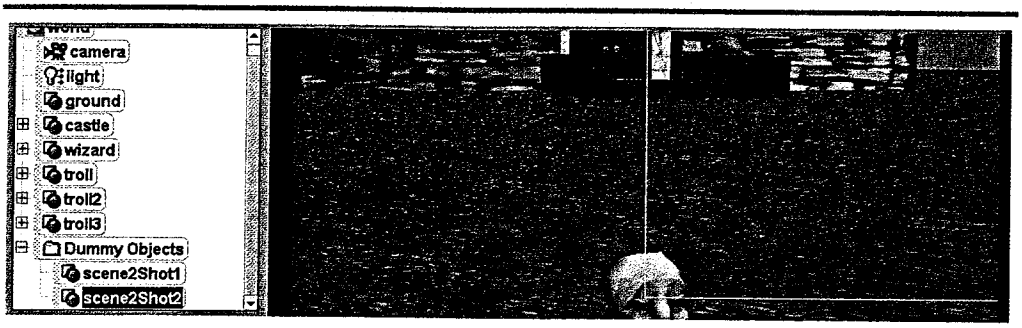


FIGURE 2-35 A second dummy

Since the third shot is back in the camera's original position, we can reuse the **scene2Shot1** dummy for the third shot and avoid creating an additional dummy.

With dummies for all three of the shots, we then click the *Add Objects* window's **DONE** button and turn our attention to programming these shots.

2.4.2 Using `setPointOfView()` to Control the Camera

Now that we have dummies for each of the shots, how do we make use of them? The key is the method ***obj*.setPointOfView(*obj2*)**, which changes the position and orientation of **obj** to that of **obj2**. If we send the message **setPointOfView(*aDummy*)** to the **camera**, then the **camera**'s position and orientation will change to that of ***aDummy***!

Back in the *editing area* with the **playScene2Shot1()** method open, we start by dragging a **doInOrder** statement into the method. We then click on the **camera** in the *object tree*, scroll down to the **setPointOfView()** method in the *details area*, and then drag **setPointOfView()** to make it the first statement in the **playScene2Shot1()** method. For its target, we select **Dummy Objects -> scene2Shot1**, as shown in Figure 2-36.

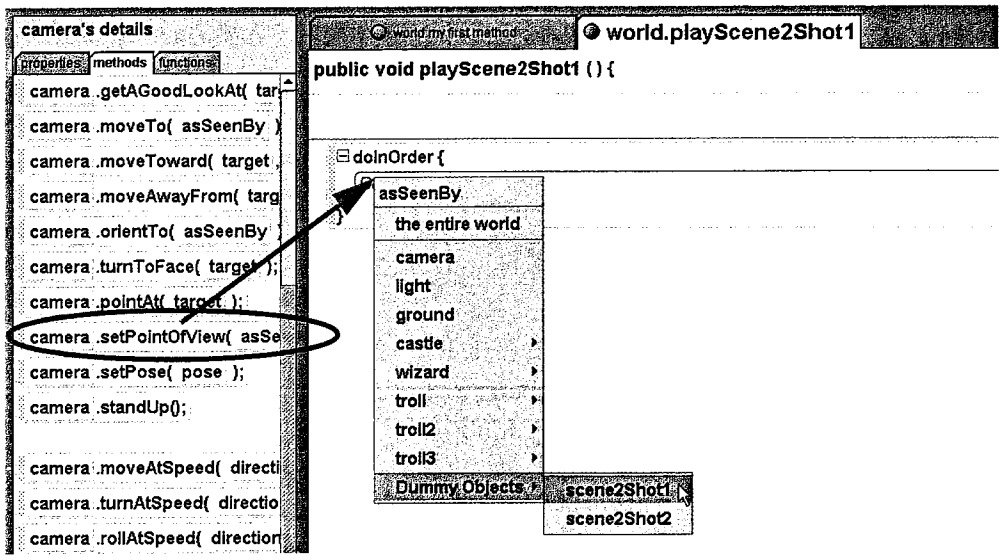


FIGURE 2-36 Setting the `camera`'s point of view to a dummy

When we have chosen **`scene2Shot1`** as its target, we then set the statement's **`duration`** to zero (so that the camera moves to this position and orientation instantly). We can then add the rest of the statements for the shot, resulting in a method definition like that shown in Figure 2-37.

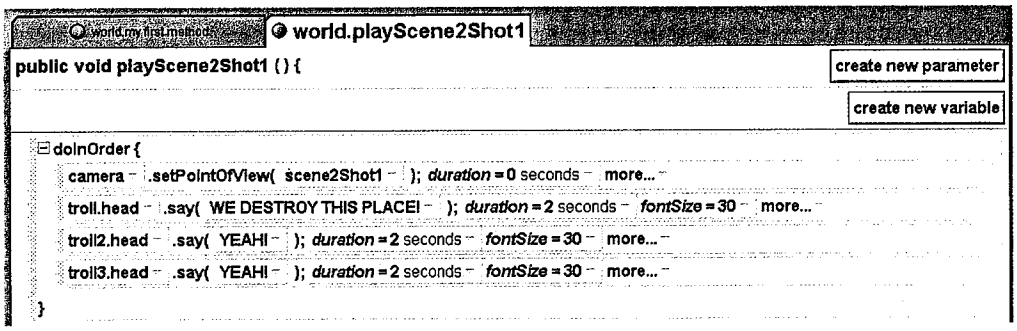


FIGURE 2-37 Using the `setPointOfView()` method with a dummy

We then use the same approach in **`playScene2Shot2()`** to move the **`camera`** to the position and orientation of the **`scene2Shot2`** dummy near the start of that method (Figure 2-38).

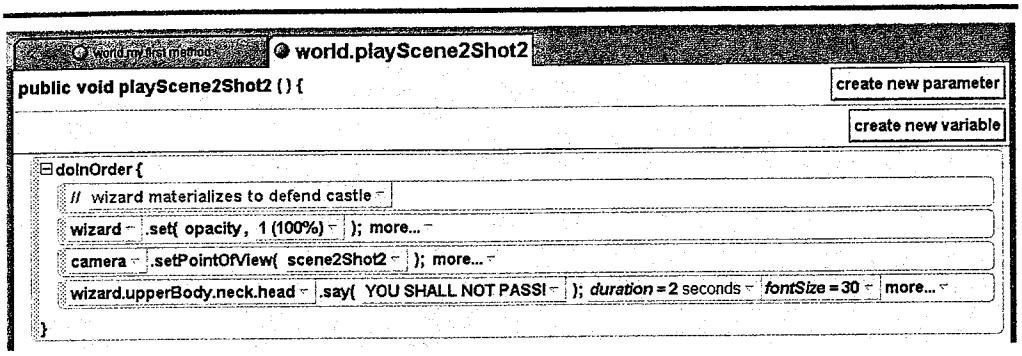


FIGURE 2-38 The `playScene2Shot2()` method

2.5

By default, the **`duration`** of the **`setPointOfView()`** method is 1 second, so the **`camera`** will take a full second to zoom in from the wide angle shot to the half-body shot of the wizard. If we want a faster zoom, we can reduce the **`duration`** (for example, 0 seconds causes an instantaneous cut). If we want a slower zoom, we can set the **`duration`** to 2 or more seconds.

Note also that to make the **`wizard`** materialize, the **`playScene2Shot2()`** method sets his **`opacity`** property to 1, using the approach described in Section 1.5.1. To make him initially invisible, we manually set his **`opacity`** to 0 in the *properties* pane.

For the third shot, we use the **`setPointOfView()`** message to reset the **`camera`**'s position and orientation back to the wide-angle shot, using the **`scene2Shot1`** dummy. Figure 2-39 shows the code at this point.

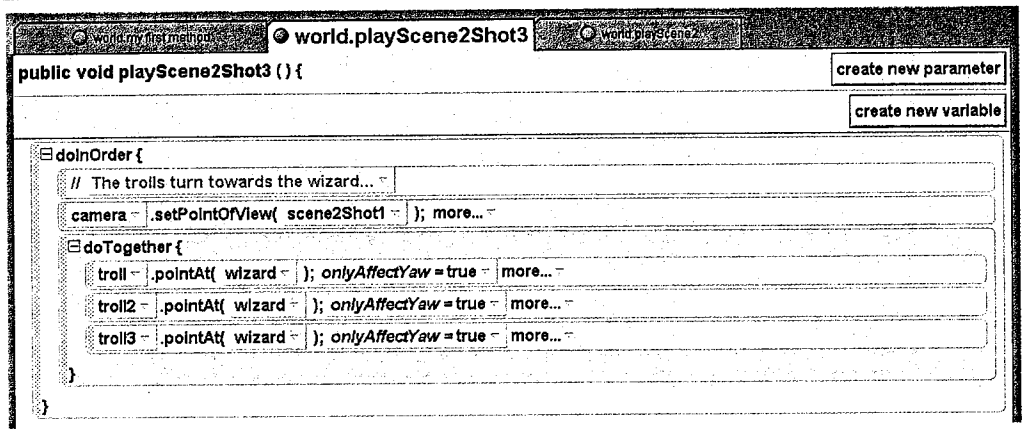


FIGURE 2-39 The `playScene2Shot3()` method

Now, when we click the **Play** button, we see the first shot from the wide angle view and see the trolls speak. The camera then zooms in to the half-body view of the wizard, and we see his dialog. The camera then zooms back out to the wide-angle view, and the trolls turn toward the wizard.... What happens next? It's up to you! (See the Chapter 3 problems for one possibility.)

You may have noticed that when we used the **pointAt()** message to make the trolls turn to the wizard, we set that message's **onlyAffectYaw** attribute to **true**. Every object in a 3D world has six attributes that determine its position and orientation in the world. Yaw is one of these six attributes, which we examine in the next section.

2.5 Thinking in 3D

Most of us are not used to thinking carefully about moving about in a three-dimensional world, any more than we think carefully about grammar rules when we speak our native language. However, to use Alice well and understand the effects of some of its methods, we need to conclude this chapter by thinking about how objects move in a 3D world.

Every object in a 3D world has the following two properties:

- An object's **position** determines its *location* within the 3D world.
- An object's **orientation** determines *the way it is facing* in the 3D world, determining what is in front of and behind the object, what is to the left and right of the object, and what is above and below the object.

In the rest of this section, we will explore these two properties in detail.

2.5.1 An Object's Position

Pretend that you are a pilot flying the seaplane in Figure 2-40.

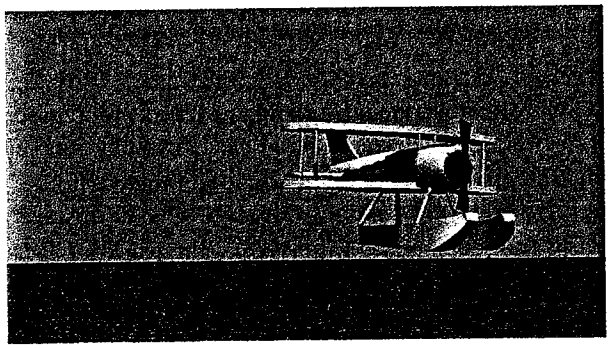


FIGURE 2-40 A seaplane

As you fly the seaplane, it can move along any of the arrows shown in Figure 2-41.

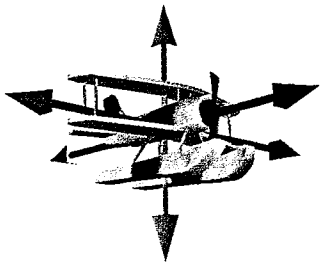


FIGURE 2-41 The seaplane and 3D axes

Each pair of opposite-facing arrows (from the pilot's perspective: *LEFT-RIGHT* [red], *UP-DOWN* [green], *FORWARD-BACKWARD* [blue]) is called an **axis**. Two or more of these arrows are called **axes**.

Every Alice object has its own three axes. For example, from a "downward-looking" angle, we might imagine the three axes of our three-dimensional world as shown in Figure 2-42.

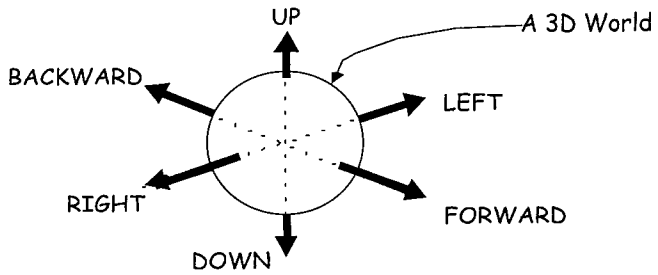


FIGURE 2-42 The three-dimensional world

Once we create a world and start adding objects to it, every object is located somewhere within that 3D world. To determine each object's exact location, we can use the world's axes.

To illustrate, the seaplane's position along the world's *LEFT-RIGHT* axis specifies its location in the world's *width* dimension. We will call this axis the **LR axis**.

Similarly, the seaplane's position along the world's *UP-DOWN* axis specifies its location in the world's *height* dimension. We will call this axis the **UD axis**.

Finally, the seaplane's position along the world's *FORWARD-BACKWARD* axis specifies its location in the world's *depth* dimension. We will call this axis the **FB axis**.

An object's **position** within a three-dimensional world thus consists of three values — *lr*, *ud*, and *fb* — that specify its location measured using the world's three axes.[4]

4. These axes are usually called the X, Y, and Z axes, but we'll use the more descriptive LR, UD, and FB.

Changing Position

To change an object's position, Alice provides a method named **`move()`** (see Appendix A). When we drop Alice's **`move()`** method into the *editing area*, Alice displays a menu of the directions the object may move, shown in Figure 2-43.

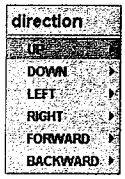


FIGURE 2-43 The directions an object may move

If you compare Figure 2-41 and Figure 2-43, you'll see that Alice's **`move()`** message allows an object to move along any of that object's three axes:

- Moving **`LEFT`** or **`RIGHT`** changes the object's location along its LR-axis.
- Moving **`UP`** or **`DOWN`** changes the object's location along its UD-axis.
- Moving **`FORWARD`** or **`BACKWARD`** changes its location along its FB-axis.

Alice's **`move()`** message thus changes the *position* of the object to which the message is sent with respect to the world's axes, but the directional values that we specify for the movement (**`LEFT`**, **`RIGHT`**, **`UP`**, **`DOWN`**, **`FORWARD`**, and **`BACKWARD`**) are given with respect to *that object's axes*, not the world's axes.

2.5.2 An Object's Orientation

When an object moves, its axes move with it. For example, if we send the seaplane of Figure 2-41 the message **`turn(RIGHT, 0.25)`**, the picture would change to that shown in Figure 2-44.

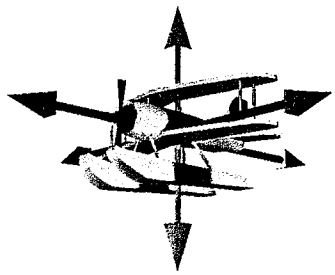


FIGURE 2-44 The seaplane turned 1/4 revolution right

If we now send the turned seaplane a message to **move(FORWARD, ...)**, the seaplane will move forward according to the new direction its FB axis points.

Yaw

If you compare the axes in Figure 2-41 and Figure 2-44 carefully, you'll see that a **turn(RIGHT, 0.25)** message causes the seaplane to *rotate* about its UD-axis. A **turn(LEFT, 0.25)** message causes a rotation about the same axis, but in the opposite direction. If we were to position ourselves "above" the plane's UD-axis and look down, we might visualize the effects of such **turn()** messages as shown in Figure 2-45.

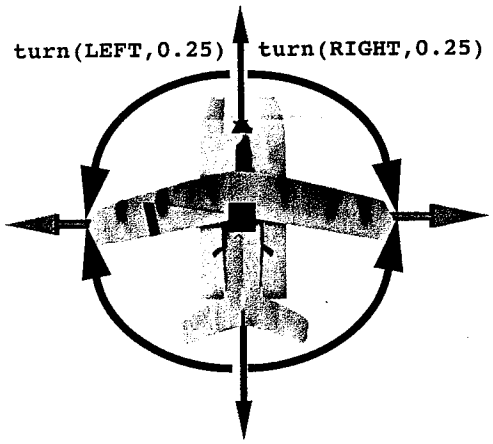


FIGURE 2-45 Changing yaw: turning left or right

In 3D terminology, an object's yaw is how much it has rotated about its UD axis from its original position. For example, when you shake your head "no," you are changing your head's yaw. Alice's **turn(LEFT, ...)** and **turn(RIGHT, ...)** messages change an object's yaw.

Pitch

We just saw that an object's yaw changes when it rotates around its UD axis. Since an object has three axes, it should be evident that we could also rotate an object around one of its other two axes. For example, if we wanted the seaplane to dive toward the sea, we could send it a **turn(FORWARD, ...)** message; if we wanted it to climb toward the sun, we could send it a **turn(BACKWARD, ...)** message. These messages cause an object to rotate about its LR-axis, as shown in Figure 2-46.

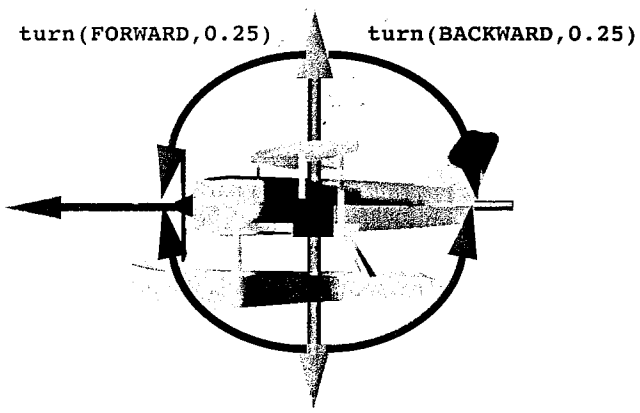


FIGURE 2-46 Turning forward or backward

An object's **pitch** is how much it has rotated about its LR axis from its original position. For example, when you shake your head "yes," you change your head's pitch. In Alice, a `turn(FORWARD, ...)` or `turn(BACKWARD, ...)` message changes an object's pitch.

Roll

An object can also rotate around its FB axis. For example, if we were to send the seaplane the `roll(LEFT, 0.25)` or `roll(RIGHT, 0.25)` message, it would rotate as shown in Figure 2-47.

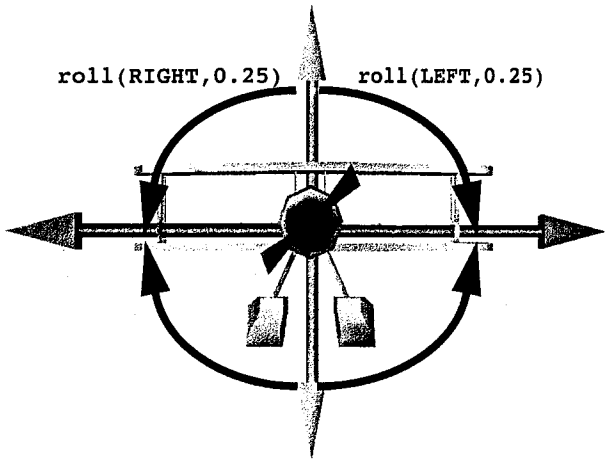


FIGURE 2-47 Rolling left or right

The amount by which an object has rotated about its FB axis (compared to its original position) is called the object's **roll**. In Alice, the `roll(LEFT, ...)` and `roll(RIGHT, ...)` messages change an object's roll.

An object's orientation is its combined yaw, pitch, and roll.

Just as an object's *position* has three parts: *lr*, *ud*, and *fb*; an object's *orientation* has three parts: yaw, pitch, and roll. An object's *position* determines where in the world that object is located; its *orientation* determines the direction the object is facing.

Back in Figure 2-39, we sent three trolls the `pointAt()` message. By default, a message `obj.pointAt(obj2);` causes `obj` to rotate so that its FB axis is pointing at the *center* of `obj2`. Unless `obj` is already pointing at `obj2`, this rotation will change the yaw of `obj`. However, if `obj` is much taller (or shorter) than `obj2`, then the center of `obj2` will be much lower (or higher) than that of `obj`, so the `pointAt()` message will also change `obj`'s pitch. This would cause the trolls to lean forward at an unnatural angle. By setting the message's `onlyAffectYaw` attribute to `true`, we ensured that each troll's pitch remained unchanged.

The message `obj.turnToFace(obj2);` is a shorthand for `obj.pointAt(obj2)` with `onlyAffectYaw=true`, and we will use it in future examples.

2.5.3 Point of View

In Alice, an object's combined position and orientation are called that object's **point of view**. An object's point of view thus consists of six values: [(*lr*, *ud*, *fb*), (yaw, pitch, roll)]. Alice's `move()`, `turn()`, and `roll()` messages let you change any of these six values for an object, giving Alice objects *six degrees of freedom*.[5] Alice's `setPointOfView()` message (see Appendix A) lets you set an object's point of view.

2.6 Chapter Summary

- ❑ World-level methods let us divide an Alice program into scenes and shots.
- ❑ The divide-and-conquer approach can simplify problem solving.
- ❑ Object-level methods let us define new behaviors for an object.
- ❑ We can reuse a class-level method in a world other than the one where we defined it.
- ❑ Control camera movement using dummies and the `setPointOfView()` message.
- ❑ In a 3D world, an object's position determines where the object is located in the world; its orientation is the object's combined pitch, roll, and yaw; and its point of view is its combined position and orientation.

5. The phrase "six degrees of separation" — which claims any two living people are connected by a chain of six or fewer acquaintances — is derived from this phrase "six degrees of freedom." The "Six Degrees of Kevin Bacon" game — that claims that the actor Kevin Bacon and any other actor are linked by a chain of six or fewer film co-stars — is further derived from "six degrees of separation." See `www.cs.virginia.edu/oracle/`.

2.6.1 Key Terms

axis
comment
divide and conquer
dummy
object method
orientation
pitch
point of view
position
reusable method
roll
scene
shot
world method
yaw

Programming Projects

2.1 Revisit the programs you wrote for Chapter 1. If any of them require scrolling to view all of their statements, rewrite them using divide-and-conquer and world-level methods whose statements can be viewed without scrolling.

2.2 The director Sergio Leone was famous for the extreme closeups he used of gunfighters' eyes in "western" movies like *For a Fistful of Dollars*; *The Good, the Bad, and the Ugly*; and *Once Upon a Time in the West*. Watch one of these films; then modify the **`playScene2()`** method we wrote in Section 2.4, using Leone's camera techniques to heighten the drama of the wizard's confrontation with the trolls.

2.3 Build an undersea world containing a **`goldfish`**. Build a **`swim()`** method for the **`goldfish`** that makes it swim forward one meter in a realistic fashion. Add a **`shark`** to your world, and build a similar **`swim()`** method for it. Build a program containing a scene in which the **`shark`** chases the **`goldfish`**, and the **`goldfish`** swims to its giant cousin goldfish that chases the **`shark`** away. (Hint: Make the giant cousin goldfish by Saving and Importing your modified **`goldfish`**.)

2.4 Choose a hopping animal from the Alice Gallery (for example, a frog, a bunny, etc.). Write a **`hop()`** method that makes it hop in a realistic fashion. Add a building to your world, then write a program that uses your **`hop()`** method to make the animal hop around the building. Write your program using divide-and-conquer so that **`my_first_method()`** contains an **`Inorder`** control and no more than four statements.

2.5 Build a world containing a flying vehicle (for example, a biplane, a helicopter, etc.). Build a class-level **`loopDeeLoop()`** method for your flying vehicle that makes it move in a vertical loop. Using the Torus class (under Shapes), build a world containing a giant arch. Then write a program in which your flying vehicle does a **`loopDeeLoop()`** through the arch.

2.6 *Boom, Boom, Ain't It Great To Be Crazy* is a silly song with the lyrics on the next page. Create an Alice program containing a character who sings this song. Use divide-and-conquer to write your program as efficiently as possible.

A horse and a flea and three blind mice *sat on a curbstone shooting dice.* *The horse he slipped and fell on the flea.* *"Whoops," said the flea, "there's a horse on me."* *Boom, boom, ain't it great to be crazy?* *Boom, boom, ain't it great to be crazy?* *Giddy and foolish, the whole day through,* *boom, boom, ain't it great to be crazy?*	*Way down south where bananas grow,* *a flea stepped on an elephant's toe.* *The elephant cried, with tears in his eyes,* *"Why don't you pick on someone your size."* *Boom, boom, ain't it great to be crazy?* *Boom, boom, ain't it great to be crazy?* *Giddy and foolish, the whole day through,* *boom, boom, ain't it great to be crazy?*
Way up north where there's ice and snow, *there lived a penguin whose name was Joe.* *He got so tired of black and white,* *he wore pink pants to the dance last night.* *Boom, boom, ain't it great to be crazy?* *Boom, boom, ain't it great to be crazy?* *Giddy and foolish, the whole day through,* *boom, boom, ain't it great to be crazy?*	

2.7 Using appropriately colored **Shapes** from the Alice Gallery, build a checker-board. Then choose an object from the Gallery to serve as a checker. Build class-level methods named **moveLeft()**, **moveRight()**, **jumpLeft()**, and **jumpRight()** for the character. Then make copies of the object for the remaining checkers. Build a program that simulates the opening moves of a game of checkers, using your board and checkers.

2.8 Using the **heBuilder** or **sheBuilder** (or any of the other persons with enough detail in the Alice Gallery), build a person and add him or her to your world. Using your person, build an aerobic exercise video in which the person leads the user through an exercise routine. Use world- and/or class-level methods in your program, as appropriate.

2.9 In Section 2.4, we developed a program consisting of Scene 2, in which a wizard faces off against three trolls. Create your own Scene 1 and Scene 3 for this program to show what happened before and after the scene we developed.

2.10 Write an original story consisting of at least two characters, three scenes, and dummies to position your characters in the different scenes. Each scene should have multiple shots. Use world- and class-level methods to create your story efficiently.