

<i>Preface to the Instructor.....</i>	<i>viii</i>
<i>Introduction.....</i>	<i>13</i>
Why Learn about Programming Computers?	13
How You Will Learn to Program with This Book and Alice	14
The Basics of Computer Programming	15
Why is it called Alice?.....	17
<i>1 Alice.....</i>	<i>18</i>
What is a Virtual World?	18
How does Alice create an animation?	19
An Alice virtual world	19
3D Models of Objects	20
A Hitchhiker’s Guide to Alice	22
1 Exercises	22
<i>Tips & Techniques 1.....</i>	<i>25</i>
Window size	25
Arranging objects.....	25
Moving subparts of objects.....	26
A comparison of motions	27
Vehicle	28
3D Text	29
<i>2 Programming in Alice: Design and Implementation.....</i>	<i>32</i>
2-1 Scenarios and Storyboards (Design)	33
Visual Storyboard	34
Generic Template.....	35
2-2 A First Program.....	41
What is a Program?.....	41
2 Summary.....	53
<i>Tips & Techniques 2.....</i>	<i>54</i>
Style	54
Sound	54
point at and onlyAffectYaw.....	56
Clipboards	58
As seen by.....	59
<i>3 Methods and Parameters</i>	<i>61</i>
3-1 Methods: World-level.....	62
Methods	62
Why do you care?	62
Example	62
Creating our own method	63
Calling (Invoking) a Method	65
Comments	67
Technical Note	68
3-1 Exercises.....	69

3-2 Parameters	71
Example	71
Setting the Stage	71
Storyboard with a Parameter	72
Creating an Object Parameter	73
Using an Object Parameter	74
Importance of Parameter Type	76
Test with Arguments	77
Multiple Parameters	78
Other types of parameters	82
Example	82
Number parameter	83
Technical Note	83
3-2 Exercises	85
3 Summary	88
3 Projects	89
<i>Tips & Techniques 3.....</i>	93
Renaming an Object	93
Color Property as Identification	93
Printing your program code	95
Orient to	95
<i>4 Character-Level Methods and Inheritance.....</i>	99
4-1 Creating New Characters	100
Example	100
A Character-Level Method	100
A Second Example	103
Saving a New Character	105
Inheritance	106
Benefits: reuse and teamwork	106
Guidelines for Writing Character-level Methods	106
Character-level Method With an Object Parameter	108
4-1 Exercises	109
4 Summary	111
4 Projects	112
<i>Tips & Techniques 4.....</i>	113
Properties	113
Setting Properties at Runtime	113
Built-in Questions (Functions)	117
Collision	120
Expressions	121
The move to instruction	122
<i>5 Interactive Programs: Events and Event-Handling.....</i>	125
5-1 Interactive Programming: Events and Event-Handlers	126
Input	127
Design -- Storyboards	127

Event-handler methods	128
Link Events to Event-handlers.....	128
Testing	130
Technical Notes	130
Mouse-Control Example.....	132
Design -- Storyboards	132
Event-handler methods	132
Link Events to Event-handlers.....	133
Technical Note	134
5-1 Exercises.....	135
5-2 Passing Parameters to Event-Handlers	138
Numeric parameter example	138
Design -- storyboard	138
Three events, one event-handler	139
Parameter is used for both duration and distance	139
Link events to event-handler	140
Pass a number parameter	140
Testing	141
Object parameter example	141
Design -- Storyboard.....	142
Writing the event-handler	142
Link the event to the event-handler	145
Testing the program.....	145
Technical Notes	145
5-2 Exercises.....	147
5 Summary.....	149
5 Projects	150
<i>Tips & Techniques 5.....</i>	<i>153</i>
Let the mouse move objects	153
Billboards.....	154
Special Effect: fog	155
Rotating around an invisible object	157
<i>6 Decisions and User-defined Questions</i>	<i>159</i>
6-1 Decisions and Logical Questions	160
Decisions	160
If/Else Example	160
Logical Operators	163
Nesting <i>If</i> statements.....	166
Else	166
Relational operators	166
6-1 Exercises.....	168
6- 2 User-defined Questions I (Boolean)	171
Introduction to Questions	171
Abstraction.....	171
User-defined questions	171
Creating a new Question.....	172

Calling the question.....	174
A more complex Boolean question.....	174
6-2 Exercises.....	177
6-3 User-defined Questions II (Number).....	180
Other types of questions	180
User-defined number question.....	180
Testing	183
Using a question with a relational operator	183
Abstracting a character-level method	184
World-level question for generic use.....	184
6-3 Exercises.....	186
6 Summary.....	188
6 Projects	189
Tips & Techniques 6.....	193
If-Else and visibility as a condition	193
Camera: View from the back	198
Lighting up the rear view.....	200
7 Repetition: Definite and Indefinite Loops	202
7-1 Loops	202
Introducing repetition	202
The need for repetition.....	203
Using a Loop.....	205
Loops and <i>Do in order</i> or <i>Do together</i>	206
A More Complex Example	206
Symmetric Methods	208
Nesting of loops	210
Technical note on looping	212
Infinite Loop	212
7-2 The indefinite loop – <i>While</i>	217
Chase scene	218
Problem.....	218
Storyboard solution.....	218
Infinite while.....	222
7-2 Exercises.....	224
7 Summary.....	227
7 Projects	228
8 Repetition: Recursion.....	232
8-1 Introduction to Recursion.....	233
Revisiting the shark <i>Chase</i>	233
Terminology	235
A game-like example	236
How to determine when the race is over	237
Random selection.....	237
8-2 Another Flavor of Recursion.....	242
A second form recursion.....	242

Towers of Hanoi puzzle	242
Two requirements	243
Towers Method	244
Special <i>Moveit</i> Method	245
Conversion Question	246
Comparison of two forms of recursion.....	249
Infinite Recursion	250
8 Summary.....	255
8 Projects	256
<i>Tips & Techniques 8.....</i>	258
Texture Maps	258
<i>9 Lists and List Processing.....</i>	261
9-1 Lists	262
Creating a list.....	262
For all in order -- Iterating sequentially through a list.....	264
For all together -- Iterating simultaneously	266
9-1 Exercises.....	268
9-2 List Search.....	272
Simulation using a list search.....	272
First Subtask – Game in progress	274
A list of hamsters to allow random selection of hamster.....	274
Second Subtask – Scoring	275
9-2 Exercises.....	277
9 Summary.....	278
9 Projects	279
<i>10 Arrays</i>	282
11- 1 Arrays: Creation and use	283
Introduction to arrays.....	283
Creating an array.....	283
Accessing an element in an array	283
Iterating though an array to find an element with a certain property	284
Searching through an array to find the tallest element	286
Using a location index variable a while statement in place of an iterator	289
11 Exercises	291
11- 2 Sorting arrays	292
What is sorting?	292
Insertion sort.....	292
Walk through the array.....	293
Remove an element from the array.....	293
Insert element in its correct location.....	295
10-2 Exercises.....	300
10 Summary.....	301
10 Projects	302
<i>11 Revisiting Inheritance: Mutable variables.....</i>	303
12 -1 Inheritance	304

Character-level variables	304
Using a character-level method to track steering a car	304
Figure 11-1-1. An initial scene	304
Implementation	305
11-1 Exercises.....	312
11 Summary.....	315
12 Cool Worlds You Can Build.....	316
12-1 Cool worlds	317
A flight simulator.....	317
Initial world with billboard instructions	317
Planning the simulation	317
How do we know when the plane is flying through a ring?	318
When should the game end?	319
What happens if the plane flies into the ground?	321
How do we start the game, continue to fly the plane, and keep track of the game?.....	323
How does the user steer the plane?	324
13 Exercises	327
13 Transition to Real World Languages.....	328
13-1 Transition to Java/C++: Syntax Switch.....	329
Syntax	329
13-2: Objects and Classes	334
Objects	334
Classes	335
Creating (instantiating) objects	336
“What you see is what you get”.....	337
Creating new classes	339
Encapsulation.....	340
Data Types and Structures	341
Methods and Functions (Questions)	341
13-3: Program Constructs	342
<i>Do in order</i> and <i>Do together</i>	342
Decision-making.....	342
Repetition: loop (for)	343
Repetition: while.....	344
Repetition: recursion.....	344
13 Summary.....	345
Appendix A.....	346
A Hitchhiker’s Guide to Alice, Part 1	346
How to Start Alice:	346
World 1: Opening and Running Your First World	348
World 2: Interactive World.....	351
World 3: Creating and saving your own new world	352
Saving a World	352
Optional: Using the Web Gallery	355
Moving Objects	356

Arranging Multiple Character Objects (the quad view)	357
Moving the Camera	360
A Hitchhiker's Guide to Alice, Part 2	362
Using One-shot Instructions to position objects	362
Using One-Shot Property Animations	368

Preface to the Instructor

This book takes a radically different approach to teaching introductory programming.

There have been relatively few innovations in the teaching of programming in the last 30 years, despite the fact that introductory programming courses are often extremely frustrating to students. This text, with the associated Alice system, is intended to take a novel approach. *We strongly recommend you see a live demonstration of the Alice system, or a videotape, before evaluating this text.* The Alice system is free; the system and videos are available at www.alice.org.

What is different about our approach?

Our approach allows students to author on-screen movies and games, where the concept of an "object" is made tangible via on-screen objects that populate a three-dimensional micro world. Students create programs by dragging and dropping program elements (if/then statements, loops, variables, etc.) in a mouse-based editor that prohibits syntax errors. The Alice system is a powerful, modern programming environment that supports methods, functions, variables, parameters, recursion, arrays, and events. We use this strong visual environment to support an objects-first approach (described in the ACM and IEEE-CS Computing Curricula 2001 report) with an early introduction to events. In Alice, every object *is* an object that students can visibly see! We begin with objects in the very first chapter.

In our opinion, there are four primary obstacles to introductory programming:

1) The fragile mechanics, particularly syntax, of program creation

The Alice editing environment removes the frustration of syntax errors in program creation, and allows students to develop an intuition for syntax, because every time a program element is dragged into the editor, all valid "drop targets" are highlighted.

2) The inability to see the results of computation as the program runs

Although textual debuggers and variable watchers are better than nothing, the Alice approach makes the state of the program inherently visible. In a sense, we offload the mental effort from the student's cognitive system to his or her perceptual system. It is much easier for a student to see that an object has moved backward instead of forward, as opposed to noticing that the "sum" variable has been decremented, rather than incremented. Alice's visual nature allows students to see how their animated programs run, affording an easy relationship of the program construct to the animation action. Today's students are immersed in a world where interactive, three-dimensional graphics are commonplace; we attempt to leverage that fact without pandering to them.

3) The lack of motivation for programming

Many students take introductory programming courses only because they are required to do so. Nothing will ever be more motivating than a stellar teacher, but the right environment can go a long way. In pilot studies of classes using Alice, students do more optional exercises and are more likely to take a second class in programming than control groups of students using traditional tools. The most common request we received regarding earlier versions of Alice was the ability to share creations with peers; we have added the ability to run Alice programs in a WWW browser so students can post them on their web pages. Although we have seen increased

motivation for all students, we have seen especially encouraging results with under-represented student groups, especially female students.

4) The difficulty of understanding compound logic and learning design techniques to overcome it
The Alice environment physically encourages the creation of small methods and functions. More importantly, the analogy of making a movie allows us to utilize the concept of a storyboard, which students know is an established movie-making process. Since most students cannot draw their storyboards, we encourage them to express their high level story concept in short prose snippets, which are progressively refined, essentially designing through pseudo-code.

How to Use this Text

Of course, as the instructor, you should use this text as you best see fit! Below are four ways we imagine the book being used, but you may discover additional ways of using this textbook:

1. **As the only text in a semester-long or short-course on programming.** This would allow students to build relatively complex (say, 300-line) programs by the end of the semester or term. Such a course might be for non-majors who want to learn the concepts behind programming without a need for transitioning to a real-world language. Alternatively, this course can be used as a pre-CS1 course for students who would like to be, or are considering, a computer science major but who lack previous programming experience. In our NSF-supported study (NSF-#0126833), we found that students who jump right into a rigorous CS1 course with little or no previous programming have an extremely high attrition rate in CS1. The use of an Alice course as a pre-CS1 course has significantly reduced the attrition rate of these students in our CS1 courses.
2. **As the first portion of a traditional "Introduction to Programming" course,** such as CS1. Both Seymour Papert's Logo and Rich Pattis' Karel the Robot have been used this way, and these systems have inspired us greatly. Unlike these systems, Alice is powerful enough to support students for several semesters (for example, seniors majoring in Computer Science at Carnegie Mellon routinely write 3,000 line programs in Alice). However, many introductory programming courses must both teach concepts and also prepare students to write programs in traditional languages, such as Java. By learning Alice first, students are well acquainted with the fundamental concepts of programming, and can quickly learn the specific syntax rules of a particular "real" language as a transition. Our final chapter provides support for this transition, and the Alice environment can ease the transition by displaying programs with a Java-like syntax, as shown in the Figure Preface-1 below.
3. **As the programming component of a "Computer Literacy" course.** At many schools, computer literacy courses attempt to give a broad introduction to computers and/or "information technology" to non-majors. Many of these courses have removed their programming component and are little more than extended laboratories on "office productivity tools" such as spreadsheets and word processors. Alice has the potential to return a gentle programming component to these computer literacy courses.
4. **In a high school "Introduction to Programming" course.** A course in Alice has great potential for a high school environment, where a high-interest, highly motivating environment is a teacher's best friend. This course could be a stand-alone course as a preparation for college courses or as preparation for the College AP computing course.

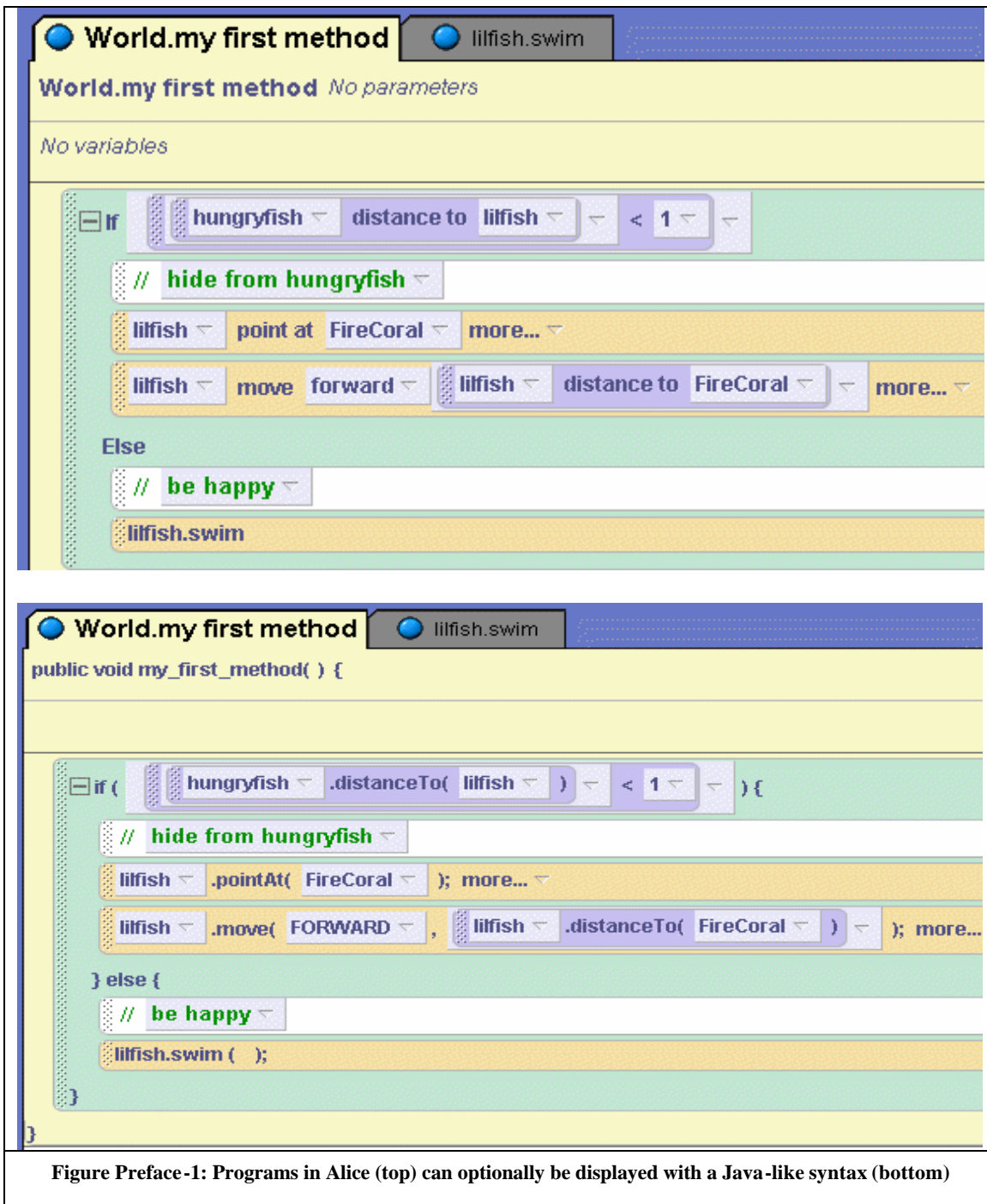


Figure Preface-1: Programs in Alice (top) can optionally be displayed with a Java-like syntax (bottom)

Structure of the Book

The text begins with an introduction motivating why students would want to write a computer program and addressing their fears about programming. We then have a brief first chapter on the basic concept of a three-dimensional virtual world. The text contains a traditional, paper-based tutorial as an appendix, and the Alice system also includes an interactive software-based tutorial that some students may prefer.

The remaining chapters begin with overview and motivation of the chapter's topic, have exercises at the end of each section, and end with a summary. For your convenience, the following is a (very) brief overview of the major concepts covered, chapter-by-chapter. Clearly, the major focus of the text material is an introduction to the fundamental concepts of programming.

Chapter 1	3D worlds, 3D models (classes) and objects in the worlds.
Chapter 2	Storyboard design and a first program.
Chapter 3	World-level (global) methods and parameters.
Chapter 4	Character-level methods and using inheritance to define new classes of objects.
Chapter 5	Interactive programs: events and event-handling
Chapter 6	Decisions and user-defined questions (functions)
Chapter 7	Repetition: a Counted Loop (for), the indefinite loop, While
Chapter 8	Repetition: Recursion
Chapter 9	Lists and List Processing
Chapter 10	Arrays
Chapter 11	Revisiting inheritance: mutable variables
Chapter 12	Cool Worlds You Can Build
Chapter 13	Transition to Real World Languages (Java/C++)

At the end of most chapters is a selection of larger, more free-form projects. We have found projects to be an enjoyable and important part of our Alice courses. The projects provided in this text are meant to inspire, not weigh down the student. In particular, where feasible, we recommend “open-ended” projects. An open-ended project asks students to design their own animation beginning with their own storyline and using objects of their choosing. We do require that a project meet certain requirements. For example: “...an interactive world, containing two or more interactions with the mouse, at least three methods, and two characters you have created by writing character-level methods and saving out the characters.” Alice lends itself particularly well to student demonstrations of their worlds to the rest of the class on the project due date.

Following most chapters is a "Tips and Techniques" section. These sections cover features of Alice, rather than the fundamental concepts of programming presented in the major chapter material. Use of the Alice interface is integrated with examples for programming concepts, where needed. The Tips and Techniques sections enrich the flavor of the book with selected “how to” topics. Many of the techniques are used in later chapters. The techniques to be used in later chapters will be marked in **green**.

Topic Selection and Sequence of Coverage

Once again, as the instructor, the topic selection and sequence of coverage is totally in your hands! As a simple guideline, Chapters 1-7 were designed to be covered in sequence and Chapters 8 – 14 were designed to be selectively covered in whatever sequence the instructor prefers.

We realize that the amount of time you have available and the goals you have for using Alice will undoubtedly affect your decision. For those pushed by very tight time schedules, a possible short introduction could be:

- Chapters 1 – 4 in about 2 weeks.

- Chapters 1 - 7 in 3-5 weeks. Those who prefer to use the Loop statement early can cover the first section of Chapter 7 immediately after Chapter 3.

A note on running Alice

The version of Alice on this disk supplied with this text is meant to run on a PC running the Windows operating system. If you are using a Macintosh or a PC with Linux, check the web site <http://www.alice.org> for a version of Alice for your operating system.

Alice's use of 3-D graphics is quite memory intensive. The Stage 3 development team (the creators of Alice) have a set of minimum and recommended machine requirements for running Alice. Please note that many older laptops do not meet these requirements.

Minimum hardware requirements:

A Pentium running at 500 MHz or better
 a VGA graphics card capable of high (16 bit) color
 128 MB of RAM

Recommended hardware requirements:

A Pentium running at 1.0 GHz or better
 16 MB 3D video card
 (TNT, i810, Rage 128, GeForce, Radeon equivalent or better)
 Video resolution of 1024x768
 256 MB of RAM
 A sound card

Acknowledgements

As noted above, Seymour Pappert's Logo and Rich Pattis' Karel the Robot were great inspirations in using a visible micro world. Alan Kay and the Squeak team inspired us to create the drag-and-drop program editor, and we were also inspired by the syntax-directed editor work done by Tim Teitelbaum. We are indebted to George Polya and Doug Cooper for our problem-solving approach.

Our deep gratitude goes to early testers and users of our text and instructional materials for their helpful comments and suggestions: Susan Rodger (Duke University), Rick Zacone (Bucknell University), Bill Taffe (Plymouth State), Angela Shifflet (Wofford College), and William Taylor (Camden County College). In addition, we are thankful for the assistance of our students: Toby Dragon (Ithaca College), Kevin Dietzler (Saint Joseph's University), Patricia Hasson (Saint Joseph's University), and Kathleen Ryan (Saint Joseph's University).

The life and breath of the Alice software is dependent on a group of creative, energetic, and dedicated graduate students, undergraduate students, and staff members group at Carnegie

Mellon University. Without these students, Alice does not live and we could not have written this textbook. The primary authors of this version of Alice include Ben Buchwald, Dennis Cosgrove, Dave Culyba, Cliff Forlines, Jason Pratt, and Caitlin Kelleher, but a more complete list is available at www.alice.org. Many artists at Carnegie Mellon have graciously placed their work into the gallery for the benefit of others; we list Sarah Hatton, Mo Mahler, Shawn Lawson and Tiffany Pomarico here, but the contributors run into the hundreds. Tommy Burnette, Kevin Christiansen, Rob Deline, Matt Conway, and Rich Gossweiler all made seminal contributions to earlier versions of Alice at the University of Virginia, who we also thank for its support and encouragement of earlier versions of Alice.

We would like to thank Alan Apt and Prentice Hall for supporting this effort.

Over the last ten years, the National Science Foundation, DARPA, NASA, Apple, Ford, Intel, Microsoft Research, and SAIC have contributed support for the development of the Alice system, for which we are very grateful.

Introduction

Why Learn about Programming Computers?

Most people reading this textbook are doing so as part of a college-or high school course. We're guessing you are taking that course because either A) you want to learn about programming computers, or B) somebody else thought you should be required to learn about programming computers. In either case, let's begin by talking about why it might be valuable for you to learn how to write computer programs.

First off, let's get one thing out of the way: **we promise to not turn you into a computer nerd.** We know there are lots of pre-formed ideas in people's heads about what computer programming is, and what kinds of people do it. But we promise that you won't suddenly develop a desire to wear a pocket protector, stop taking showers, or start speaking exclusively in obscure abbreviations. Honest. **This book uses a system called Alice, which makes it possible to write computer programs in a totally different way than ever before.** Rather than typing obscure "computer language" into a machine in the hopes of getting it to do some sort of strange calculation, you'll have the opportunity to be the director of a play, where on-screen characters act out the script you create! But...let's not get too far ahead of ourselves yet. Let's get back to why you might want to program a computer at all.

There are many reasons to learn to program a computer. For some people, computer programming is actually a great deal of fun; they enjoy programming computers as an end unto itself. But for most people, writing computer programs is satisfying more because it is a means to an end; they have something important to do, and the computer is a useful tool for them. In fact, the applications of computers are becoming so pervasive in our society that it's fair to say that you'll be interacting with a computer from your very first day to your very last day - literally! The day that a baby is born [see photo BABY], many hospitals put a small computer chip on a band around his or her ankle, to make sure they know where the baby is at all times. On your last

day, you're likely to have a computer monitoring your vital signs on your deathbed. In between, you're likely to live a lot longer, and a lot healthier, because of computer advances helping medical research that allows us all to reduce our risk of cancer, computer-controlled brakes and airbags in our cars [see photo AIRBAG], and computer modeling that allows us to design drugs to fight diseases like AIDS [see photo DRUG DESIGN]. Computer programmers help to make all of these technological advances possible.

Computers, and the software that computer programmers write for them, have revolutionized the entertainment industry. Computer gaming is becoming increasingly popular. Movies in the Star Wars series [see photo STAR WARS], and the special effects in them, are only possible because of computers. By the way, one of the undergraduate authors of the Alice system graduated and went to work at ILM (Industrial Light and Magic), who do the special effects for Star Wars films. So the next time you watch *The Phantom Menace*, look for Tommy Burnette in the credits!

Computers help us communicate with each other by maintaining complex cellular telephone networks [see photo CELLPHONE USER], aiding marine research by tracking animal migratory patterns [see photo WHALES], and allowing us to explore space [see photo SPACE STATION or FEMALE ASTRONAUT], none of which would be possible without computers.

Of course most of the people who write the software for these projects are professional programmers, who have spent years studying programming. But even people who are not going to end up as professional programmers can benefit greatly from even one course's worth of programming. Modern applications, like spreadsheets and word processors, provide end users with the opportunity to save themselves time and effort by using "macros" or other programming-like features that allow you to tell the computer to do something long and tedious, rather than having to do it yourself by hand. Also, if you have even a little experience with programming, you're much more likely to become the "go to" person in an office where computers are used, and that can help you get ahead in your career. Taking a course in computer programming will help you **overcome your fear** of doing things with computers if and when you are required to use them. While it makes for a cute joke to say that you're "computer challenged," you'll be happier when you stop viewing computers as mystical, scary things.

Most importantly, even one course in computer programming can be useful as a way of **learning a new way to think**, much as taking a drawing course is valuable as a way to learn how to **look at the world differently**. Everyone talks about how they'd like to improve their general problem solving skills. Well, computer programming may have a reputation as a nerdy activity, but the truth is that it is a pure, distilled form of problem solving. And learning to think in new ways is always extremely valuable.

How You Will Learn to Program with This Book and Alice

This textbook and the associated Alice system will teach you to program a computer, but in a fundamentally different, and more enjoyable, way than ever before. In terms of tone, we have worked very hard to make learning to program as painless and "non-geeky" as humanly possible:

we know that many people are turned off by the culture that surrounds computers and programming, and we desperately want to help you see that it doesn't have to be that way!

Most programming, especially in introductory computer courses, has the feeling of mundane calculation: add up a bunch of numbers, and print out their sum and average. Often, these courses are very frustrating because of all the obscure technical details that students must get right before anything will work at all. Students often talk about slaughtering chickens in ritual attempts to appease the computer gods. We felt there had to be a better way.

This book uses a completely different approach that is only recently possible due to the increased power of desktop computers and the development of novel software that uses that power, especially for 3D graphics. The Alice system, which is provided freely as a public service by Carnegie Mellon University¹, provides a completely new approach to learning to program. Originally developed as part of a research project in Virtual Reality², Alice lets you be the director of a movie, or the creator of a video game, where 3D objects in an on-screen virtual world move around according to the directions you give them. Rather than using obscure computer terms, you use natural English language words, like "move forward" or "turn around." Best of all, you can't make mistakes! Well, of course you can **always** make mistakes, by telling one of your objects to move forward when you **meant** to move the object backward. What we mean is that you can't make the kind of "computer mistake" that most students get frustrated by - where you type something wrong and you can't figure out why the program won't run at all.

If the term "computer programmer" makes you think of some poor slob hunched over a computer keyboard in a darkened room - don't worry! In fact, you'll almost never even touch the keyboard when you are using the Alice system. You will create programs by dragging words and objects around on the screen using the mouse. Then, when you press the "play" button, the objects in the 3D world on your screen will come to life and act out the script you have written for them to perform! [see figure INTRO-ALICE-SCREEN]. So, in a sense, **being a "computer programmer" using Alice is really like being a movie director or a puppeteer.**

After you've learned how to use Alice, you'll understand all the fundamental ideas involved in programming. Then, you will be in great shape to go use one of the "real world" languages where you have to type with the keyboard and get all the commas and semicolons in the right place. But at least you'll know how to program, and **all you'll have to learn are the particular "rules" (sometimes called "syntax"³) of languages like Java, C++, C #, or whatever.**

The Basics of Computer Programming

A computer program really is nothing more than a set of instructions that tell the computer what to do. Of course, there are a million ways of telling the computer to do something, so how you do it can matter. Believe it or not, computer programmers often use words like "elegant" to describe well-written programs. We recommend that you think of a computer program not as a way to tell

¹ We would also like to gratefully thank the University of Virginia, where an earlier version of Alice was developed.

² We would like to acknowledge the support of the National Science Foundation, DARPA, Intel, and a number of other sponsors who have supported the Alice project: a complete list is available at www.alice.org

³ Not to be confused with *sin tax*; a tax on things like cigarettes or alcohol.

the computer what to do, but instead consider that a computer program is a way to tell another human being what you want the computer to do.

A computer program is not “a way to tell the computer what to do.”

A computer program is a way to tell another human being what you want the computer to do

That makes it a lot easier to talk about whether something is “elegant” or not; it’s elegant if other human beings can easily understand and appreciate the intentions of the original programmer.

The most important thing about computer programming is to not let it scare you: at the heart, it's really a very simple thing: All computer programs are made from very simple ideas:

A list of instructions, for example "put on your left sock, then put on your right sock, then put on your left shoe, then put on your right shoe." Computer Scientists call this *sequential processing*.

IFs, for example "IF it is raining, take an umbrella". Computer Scientists call this *conditional execution*.

Repeating Behavior, for example "Stomp your foot five times" or "WHILE there are cookies on the plate, keeping eating cookies" (That last part actually sounded kind of fun!) Computer Scientists call this *looping*, or *iteration*.

Breaking things up into smaller pieces, for example "The way we're going to clean the house is to first clean the kitchen, then we'll clean the bathroom, then we'll clean each of the three bedrooms one at a time." Okay, so that doesn't sound like as much fun as eating the cookies, but it’s still a pretty easy concept. Computer Scientists call this *problem decomposition*, or *step-wise refinement*, or *top-down design*, but it’s really an ancient philosophical term called *reductionism*.

Asking questions, for example, "Look in the phone book and find the number for Rebecca Smith," or "Weigh this baby and tell me how many pounds she weighs." Computer Scientists call this *calling a function*.

Computer programming is really just these ideas, used in all sorts of combinations. Of course, what can make things hard is complexity. Then, just like putting on a stage play with 400 actors, 500 costumes, and live camels that appear in Act II, Scene IV, things can get complicated just because that’s a lot to keep track of! In this book, we will teach you some of the tricks for managing complexity, and for planning out how to write programs before you actually try to get them to work. In fact, **learning how to think as you design a program is probably the most valuable part of learning to program**. You may have heard the term *object-oriented programming*; this textbook and the Alice system are based on the use of objects. In an Alice program, the objects are things you can actually see.

Why is it called Alice?

First of all, it's not an acronym: Alice isn't A.L.I.C.E, and it doesn't stand for anything. The team named the system "Alice" in honor of Lewis Carroll, who wrote *Alice's Adventures in Wonderland* and *Through the Looking Glass*. This honors a man who was both a great author of children's literature and a great mathematician. Just like the people who built Alice, Lewis Carroll was able to do the really complex technical stuff, but he knew that the most important thing was to be able to make things simple and fascinating to a young learner. [see PHOTO ALICE STEPPING INTO LOOKING GLASS]

Alice was doubtless a little afraid when she stepped through the looking glass herself, but please take that first step, and we promise that **learning to program a computer won't be nearly as hard as you might think.**

1 Alice

As we said in the introduction, learning to program in Alice means that you will create virtual worlds on your computer and populate the worlds with some really cool characters and scenes. Then, you will write programs (sort of like a movie script or a video game controller) to direct your own production of animations in those worlds. In this chapter, we begin with an introduction to virtual worlds in Alice and a look inside some of the secrets of what Alice does to help you create an animation.

What is a Virtual World?

{Definition} Virtual: to exist in effect, though not in actual fact.

Video games and simulations can be either two- or three-dimensional (2D or 3D). You may have used a 2D graphic simulator in a driver education course. Pilots use flight simulators as part of their training. The advantage of simulations is obvious – when the fighter plane crashes under the hands of the novice pilot, neither the pilot nor the aircraft goes up in smoke. A video game or simulation implemented in 3-D is called a *virtual world*. Using a virtual world lends a sense of reality to the simulator and increases its effectiveness. To see the difference between 2D and 3D, compare the images in Figure 1-2-1 and 1-2-2. The image in Figure 1-2-1 shows a movie set mock-up front and back. Clearly the structure is 2D because it has width and height, but no depth. Figure 1-2-2 shows front and back camera shots of the Tortoise and Hare out for their daily exercise run. The Tortoise and Hare are objects in a 3D virtual world, having width, height, and depth so camera shots captured from different angles show objects that give a sense of being real.



Figure 1-2-1. 2-D mock-up, front and back view



Figure 1-2-2. Tortoise and Hare, front and back view

How does Alice create an animation?

In Alice, you will build virtual worlds and create animations by moving the objects in the world in the same way that objects are moved in a flight simulator or a video game. Many of the same techniques are used in Alice to give the illusion of motion as are used by animators to create animated cartoons for film studios such as Disney and Pixar. Animation is a fantasy of vision, an illusion. To perform this illusion, the filmmaker and artist collaborate to create a sequence of artwork frames (drawings or images) where each successive frame has a slightly different view of a scene. The scene is drawn with objects. Then the same scene is redrawn with the objects positioned in a slightly different place. The scene is drawn again and the objects moved just a bit more, over and over and over! For example, Figure 1-2-3 illustrates a sequence of frames (left-to-right) showing a ball-like figure in each one:

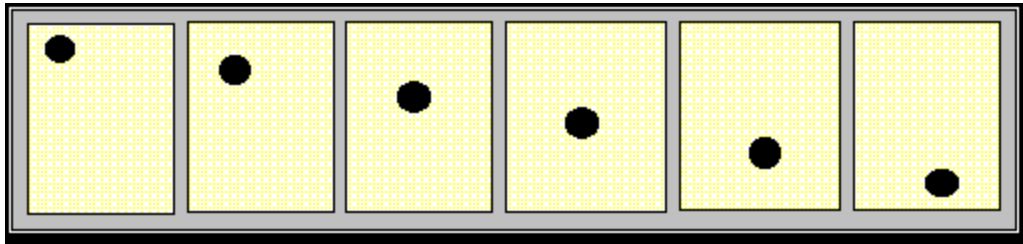


Figure 1-2-3. A sequence of frames

In animation production, frames are photographed in sequence on a reel of film or captured by a digital video camera. The film is run through a projector or viewed on a monitor at a speed that displays the sequence faster than the human eye can detect. Therefore, the brain is tricked into perceiving the ball (in the above sequence of frames) falling through the air. Instead of creating just one picture, many, many pictures are created and displayed in rapid sequence -- creating an illusion of motion. Alice creates a similar effect on your computer screen. There is no need to worry about being a great artist. Alice takes care of all the computer graphic work to create the sequence of frames. You act as the director to tell Alice what actions the objects are to perform. Alice creates (*renders*) the animation on your computer screen.

An Alice virtual world

An Alice virtual world begins with an initial scene composed of a blue sky and a green ground surface, as shown in Figure 1-2-4.

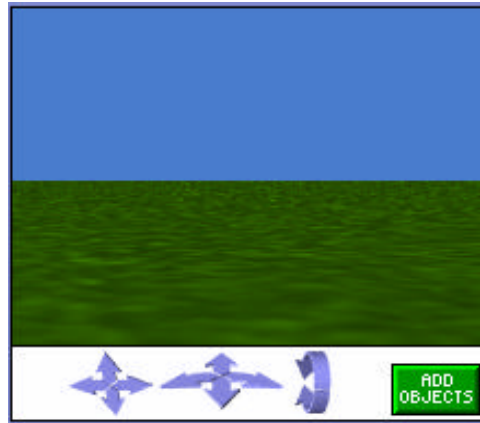



Figure 1-2-4. A simple world

Part of the fun of using Alice is to use your imagination to create new worlds with creative ideas for animations. Objects are added to the scene to provide a setting (trees, houses, starry skies, and such) and people, animals, space ships, and others that will carry out the actions. 3D models, prepared by the Alice developers, are provided as part of the Alice software. Objects are added to

a new world by first pressing on  at the lower right of the 3D world. Then, a scene editor appears, as shown in Figure 1-2-5. By clicking on the folders, you can find objects to add into your world.

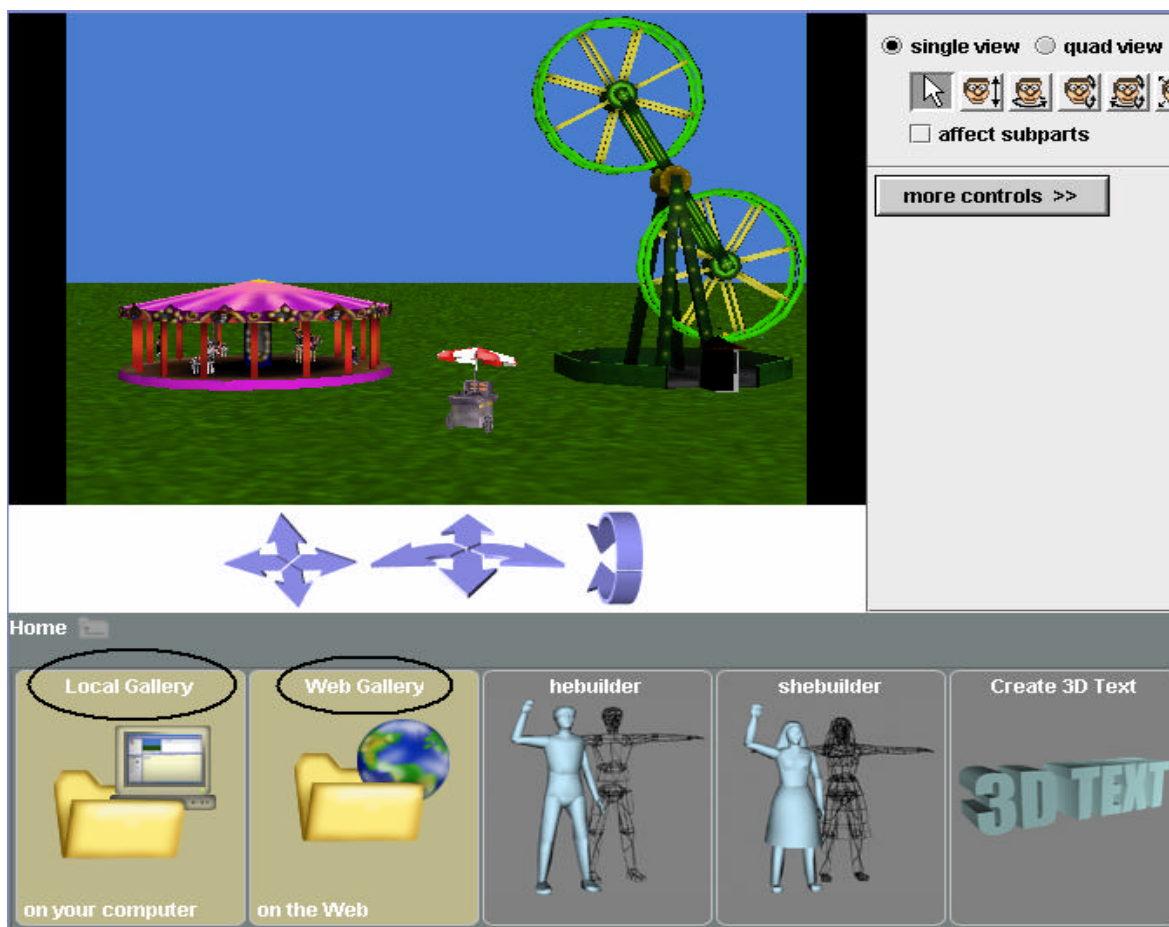


Figure 1-2-5. Web and local galleries

3D Models of Objects

A huge number of free 3D modeled objects can be used in your worlds. The installation of Alice on your computer includes a Local Gallery that contains a selection of 3D models. Additional models can be found in the Web Gallery (an online gallery) and at www.alice.org. The CD with this book also contains a gallery. To view the objects in a gallery collection, click on the gallery folder in Alice, as shown in Figure 1-2-5. Instructions for starting an Alice world and adding objects to the world are given in Hitchhiker's Guide, parts 1 and 2 in Appendix A.

Alice is not a 3D graphics-drawing program. This is why generous galleries of 3D models are provided. But, it is not possible to think of everything someone will want for his or her virtual world. To help you build objects on your own, custom builder tools are available to construct people objects and 3D text, as shown in Figure 1-2-6. The *hebuilder* and *shebuilder* tools allow you to build your own person object with various skin textures, body sizes, hairstyles and clothing.

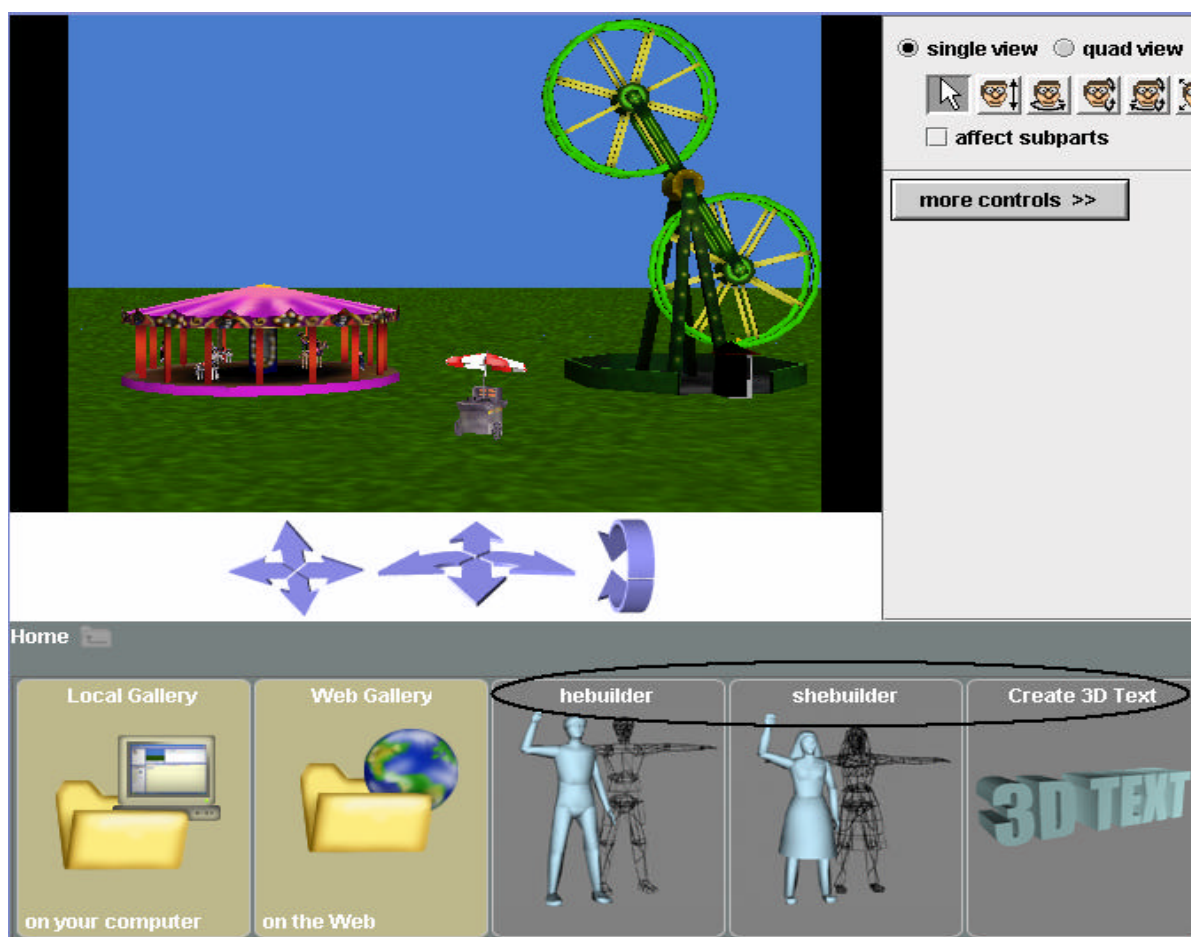


Figure 1-2-6. Construct your own objects

Objects in an Alice virtual world are automatically given a three-dimensional orientation. An object "knows" (perhaps from some internal gyroscope) which way is *up* or *down* (along the vertical axis) relative to itself. Also, the object understands the meaning of *left* and *right* (along the horizontal axis), and also which direction is *forward* and *backward* (along the depth axis), as seen in Figure 1-2-7. (In this world, the astronaut has been loaded from the People collection and the moon surface from the Sets collection of the gallery.) This amounts to six degrees of orientation within the three-dimensional space of the virtual world. It is important to notice that *left* and *right* directions are with respect to the astronaut object, not the camera's point of view.

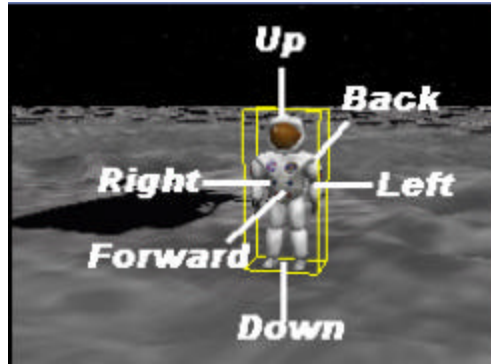


Figure 1-2-7. Object orientation

A Hitchhiker's Guide to Alice

Appendix A provides a tutorial-style “Hitchhiker’s Guide” with detailed instructions on how to start a new Alice world, where to find the galleries of 3D models, how to change the color of the ground, and how to add objects to a new world and properly position objects in a scene. This is where we have to depend on you to make maximum use of the hitchhiker’s guides to find out how to create a new virtual world of your own. Oh, go ahead – have fun!

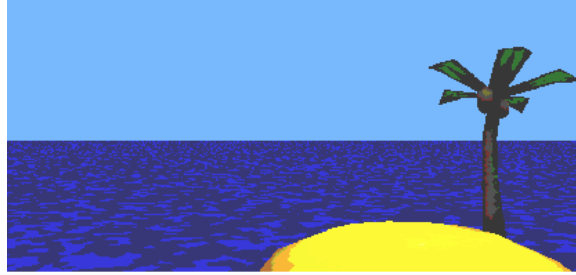
1 Exercises

The purpose of this first set of exercises is to learn to use the Alice interface to create a new virtual world. We strongly urge you to complete the Hitchhiker’s Guide, parts 1 and 2, in Appendix A. Then, you may wish to complete the exercises below for further practice on your own.

Note: It is a good idea to save your world periodically – every 15 minutes or so. Select **File | Save World As** from the menu. In the popup file save box, navigate to where the world will be stored on your computer, enter a name for the world, and finally click on the Save button. A name for a world is suggested by the name of the exercise. For example, exercise 1 could be named *IslandWorld*.

1. IslandWorld

This exercise is for practice with building a background scene and adding objects into your own world. Create an island scene. Start by changing the ground color to *blue*. Add an island object (located in the **Sets and Skies** collection of the gallery). Use the scene editor to position the island a bit to the right of the center of the scene. Adjust the camera view so you can see the coconuts in the palm tree on the island. Be sure to keep the island in view. The world should look something like the world scene shown below.



Now, add a goldfish to the scene. When the goldfish is added to the world, you may find that the goldfish is not visible because it is located behind the island or that it is not properly positioned. Use the scene editor and its quad view to arrange the goldfish so it looks like it is swimming in the water to the left of the island. Use the camera controls to zoom out so the island and the goldfish are both in the camera's view.



Note: Sometimes, you may change your mind about an object and want to get rid of it. To remove an object from the scene, position the mouse cursor over the name of the object in the object tree and right mouse-click. Select **Delete** from the listed actions. The object should disappear into the atmosphere! Another helpful way to remove an object that has just been added to a world is to click the Undo button. The Undo button can be clicked again and again.

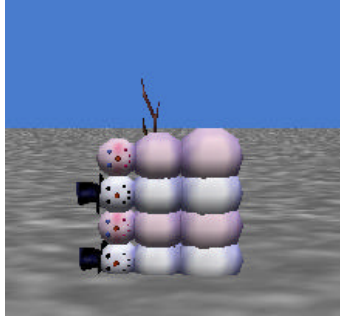
2. Winter

Open a new world in Alice and add two snowman objects to the scene (located in the Animals collection). Of course, snowmen standing on a grassy-green lawn will probably melt rather quickly. Make the scene more realistic by changing the color of the ground to white. Then, create a snowmen stack by using the scene editor's quad view to position one snowman on top of the other (vertically), as shown below. Finally, add a "cloudysky" to the background (Sets and Skies collection in the gallery).



3. SnowPile

Build a "wall" with two snowmen and two snowwomen by tipping them over on their sides, and piling them on top of one another. (Use one-shot instructions and the mouse controls.) Four snowpeople (alternating between men and women might be used to produce a wall that looks something like the one below.

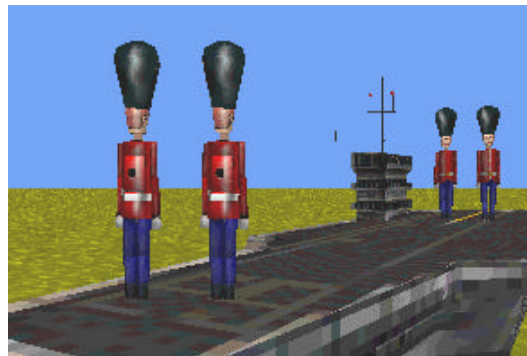


4. TeaParty

As a tribute to Lewis Carroll (the author who imaginatively wrote about the famous Alice character in wonderland), create a Tea Party for Alice and the WhiteRabbit. In addition to Alice and the WhiteRabbit, the party should include a table and two painted chairs, a teapot and creamer, a toaster, plates, and mini-wheat biscuits (make them yellow). Try to use one-shot motion instructions and the screen editor to properly position objects like the teapot and creamer on the table.

5. ToySoldiers

Add a carrier (aircraft carrier) and 4 toy soldiers to a new world. Line up the soldiers for a formal ceremony -- two on each end of the carrier as shown below.



Use one-shot move instructions to move the arms of the soldiers to make the soldiers salute each other. Or, use the mouse controls in the layout menu to move the arms into position. (Use the “affect subparts” checkbox to allow the mouse to move their arms.) Raise the left arm of each soldier up (at about a 45-degree angle with the horizontal plane). The result should be a scene where all four soldiers are saluting.

Tips & Techniques 1

Alice provides a rich animation development environment that includes many tools for rapidly developing cartoon-like animations with special effects. While it is beyond the scope of this text to cover all the special capabilities of Alice, we would like to show you some. At the end of many chapters in this book, one or more *tips and techniques* will be presented. These special tools are not essential to learning how to program per se, but they do provide fun ways to build some great animations with special effects. Please note that the Tips & Techniques sections are important. Those tips and techniques marked in green (such as the **Vehicle** tip) will be used later in the text.

Window size

When the Play button is clicked to run an animation, a World Running window appears on the screen where the animation is displayed. You may wish to make this window larger or smaller. Grab the corner of the window with the mouse cursor and drag the window to the size you want, as shown in Figure T-1-1. After you change the size of the window, Alice will remember the size. One important consideration regarding window size is a large window slows down the graphic renderer (the graphic renderer is the technical name of the software that creates the animation) so the animation runs more slowly but a small window allows a faster runtime.



Figure T-1-1. Modifying window size for animation rendering

Arranging objects

When setting up a new world, we add an object to the scene. Alice carefully positions the new object in the center of the world. Then, we use the camera controls to shift the camera position and the mouse controls to drag the object to where the object is to be positioned. This is followed by adding the next object to the scene and using the mouse to position that object in the scene.

This procedure works just fine in worlds where only a few objects are in the scene and the objects are to be placed relatively close together. But, when many objects are to be in the scene,

adding objects to the scene one at a time and arranging them around the scene can be confusing. The reason is that not only are we moving objects, we are also moving the camera around the scene. After several such moves, a new object added to the scene might not be in view of the camera when it is added. This means you may spend a lot of time with the camera controls trying to find newly added objects and put them into position.

We recommend that all objects be added to the scene before arranging the scene. Figure T-1-2(a) shows a scene where many objects have been added to a scene but not yet arranged and T-1-2(b) shows the scene after arranging the objects in the scene.



Figure T-1-2. (a) Before arranging

(b) After arranging

Moving subparts of objects

In the scene editor (see Tutorial 1-2, Appendix A), the mouse movement controls are automatically set to move an entire object. But, an option is available to allow the mouse to move subparts, as shown in Figure T-1-3. When the affect subparts option is checked, the mouse can be used to control the movement of subparts of objects, rather than the entire object.

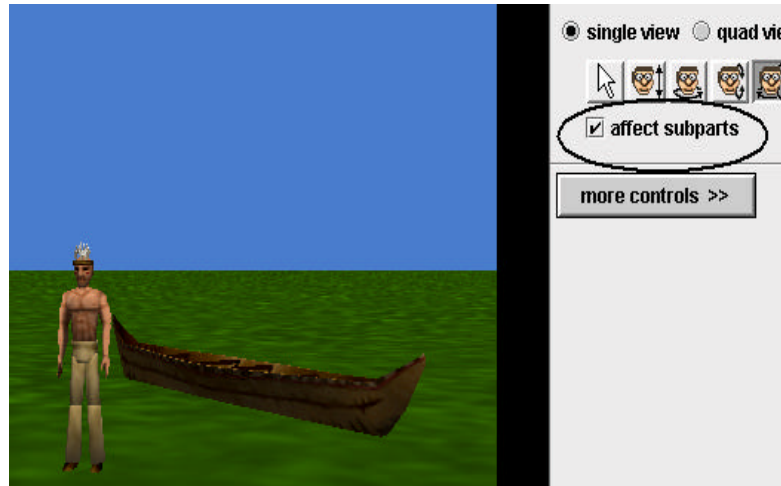


Figure T-1-3. Affect subparts option

This option is most helpful when object sub-parts must be dramatically rearranged for some purpose. In this example, the man is to sit in the canoe. One-shot instructions could be used. But, the easiest way to reposition the man's legs into a sitting position is to use the mouse to move the leg sub-parts. The result is illustrated in Figure T-1-4. Now, the man can be dragged into a sitting position within the canoe.



Figure T-1-4. Legs repositioned

A comparison of motions

We have found that mouse motion control (in the scene editor) is very good for placing an object in an approximate location but one-shot instructions are needed for exact alignment. The scene editor (especially the quad view) is great for positioning objects relative to one another. It is quite easy to add an object to a world, and then use the mouse to move and rotate it approximately to the location we would like. If we make a mistake, we can simply undo our actions (or even delete the object) and try again. While the approximate positioning of an object is easy to do with the mouse, its exact positioning (we find) is a bit more challenging. For example, trying to pile several snow people one on top of one another to build a snow wall is difficult to do with the mouse. Getting them approximately on top of one another isn't too difficult, but placing them exactly one on top of another is tough to do with the mouse. One-shot

instructions, however, will give good alignment. In setting up world scenes, the best strategy is to use a combination of one-shot instructions and mouse motion controls in the scene editor.

Vehicle

In some animations, we may want two objects to move together in such a way that one object appears to be riding on another object. For example, a person could be riding in a car or sailing in a boat. Alice provides a special property called *vehicle* that is intended to make it easy to simulate this kind of coordinated motion for two objects. As an illustration of the vehicle property, consider a circus act where a chicken rides on the back of a horse, as seen in Figure T-1-5. As part of the circus act, the horse trots around in a circular path and the chicken rides on the back of the horse.



Figure T-1-5. Circus act, chicken riding on horse

To synchronize the movement of the chicken and the horse, we will make the horse be a *vehicle* for the chicken. To create this special effect, we first select chicken in the object tree and then select the details tab (under the object tree at the lower left of the window). We then click on the image to the right of the vehicle property. A list of possible vehicles appears in a pull-down menu and we select the horse, as illustrated in Figure T-1-6. Now, when the horse moves the chicken will move with it.

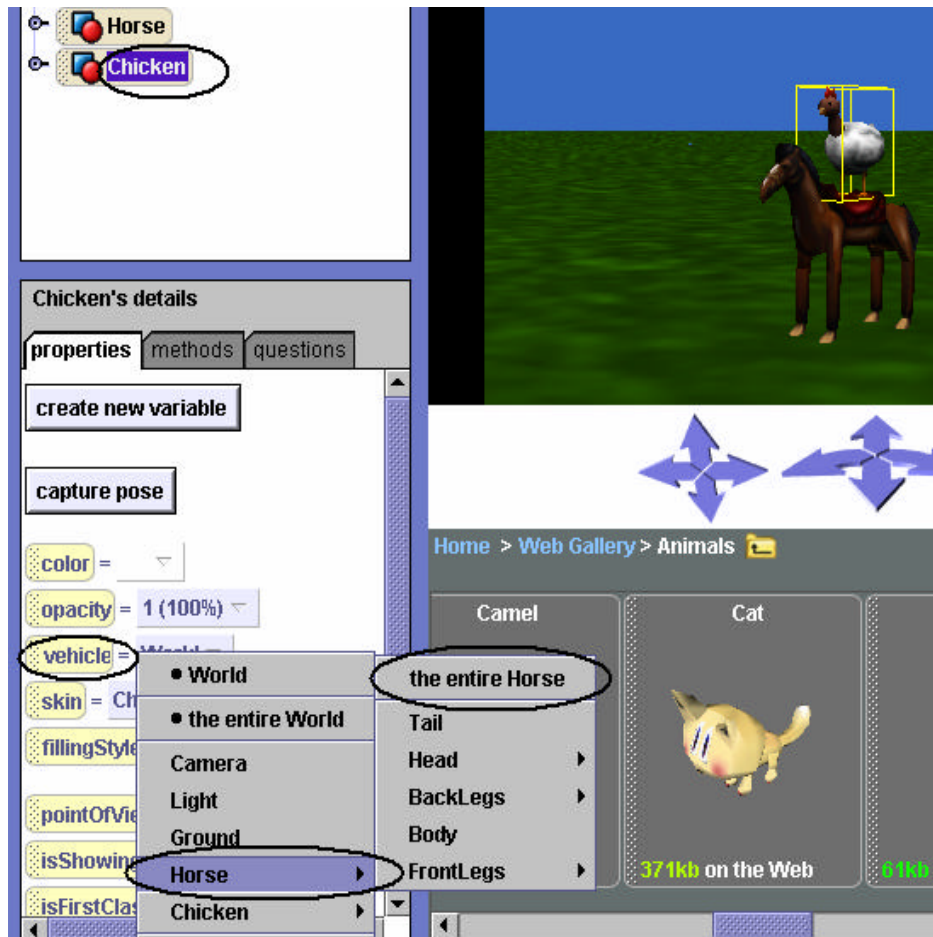


Figure T-1-6. Selecting horse as a vehicle for chicken

3D Text

The Alice gallery offers 3D text objects. To add a 3D text object to a world, click on the Create 3D text folder in the gallery, as seen in Figure T-1-7. A text dialog box pops up for entering text, as in Figure T-1-8. The dialog box allows font selection, bold and italic options, and an editor for entering the text.

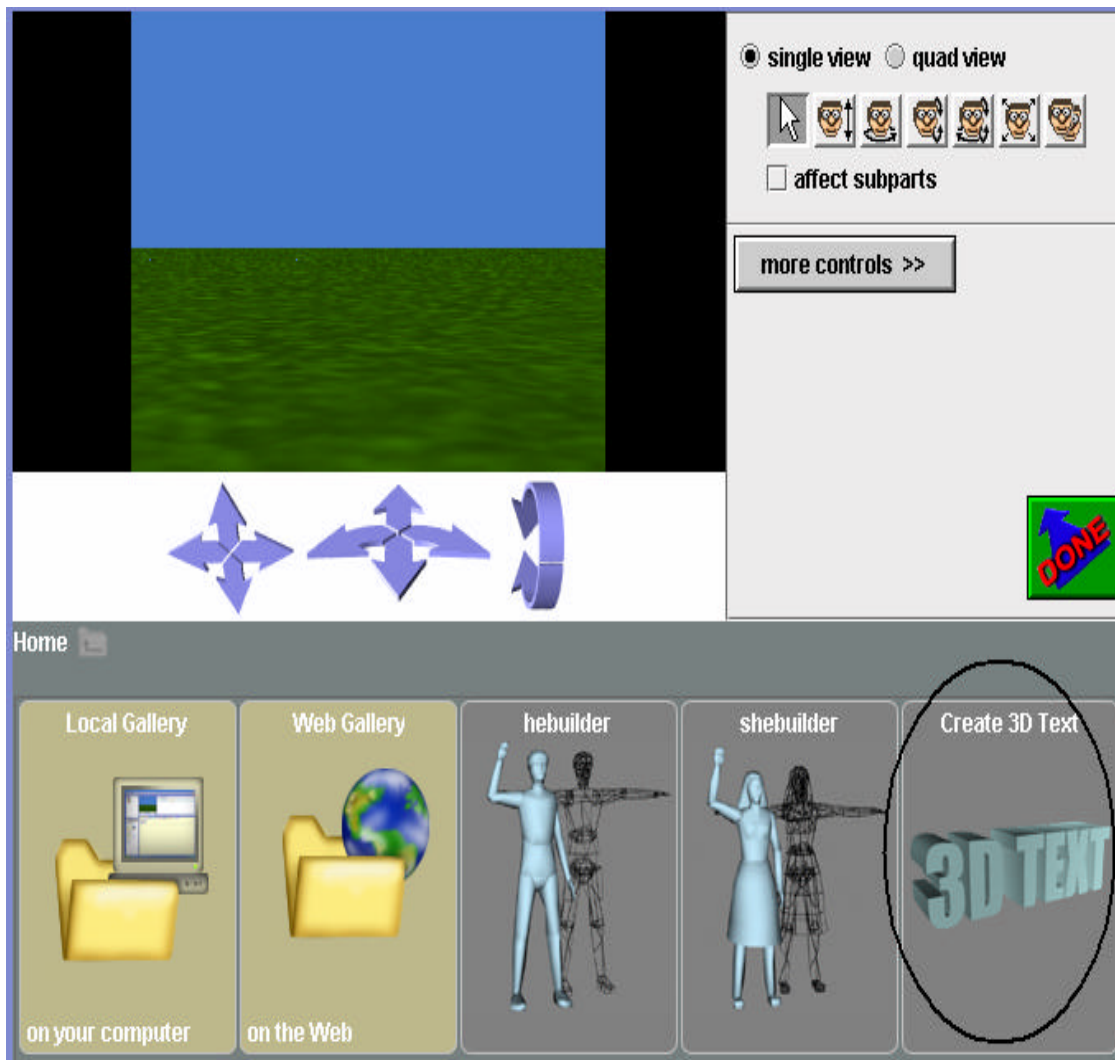


Figure T-1-7. 3D text gallery

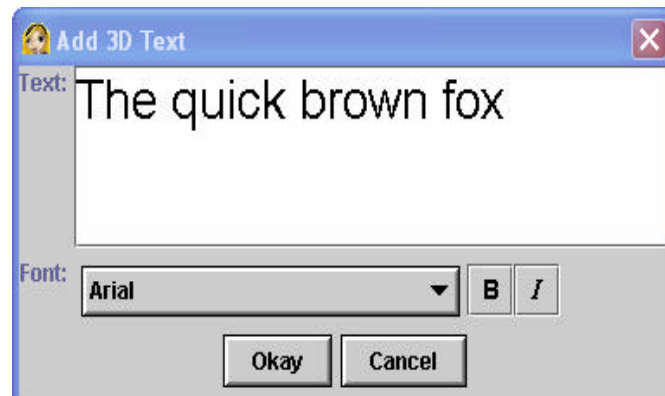


Figure T-1-8. Text Dialog Box

When the Okay button in the dialog box is clicked, Alice adds a text object to the world and an entry for the object in the object tree. The name of the object is the same as the text string displayed, as can be seen in Figure T-1-9.

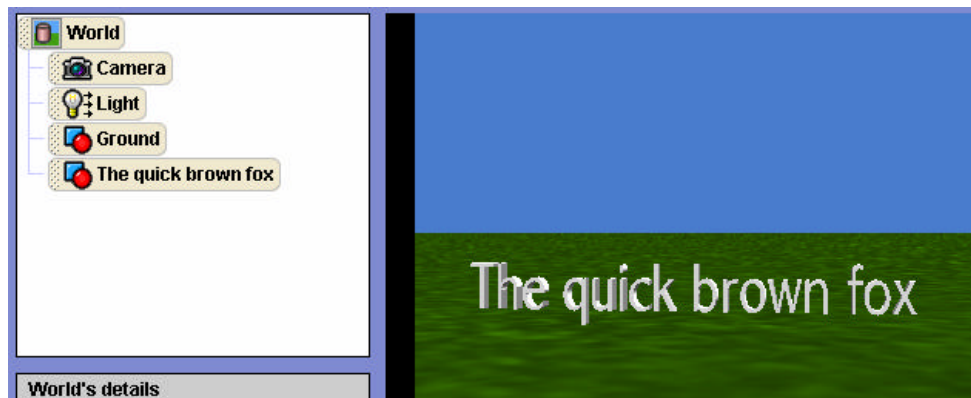


Figure T-1-9. Text object is added to scene and object tree

The text object can be positioned using mouse controls in the same ways as any other object. To modify the text in the object string, click on the text in the properties list of Details area. Then, enter a new string of text in the popup dialog box.

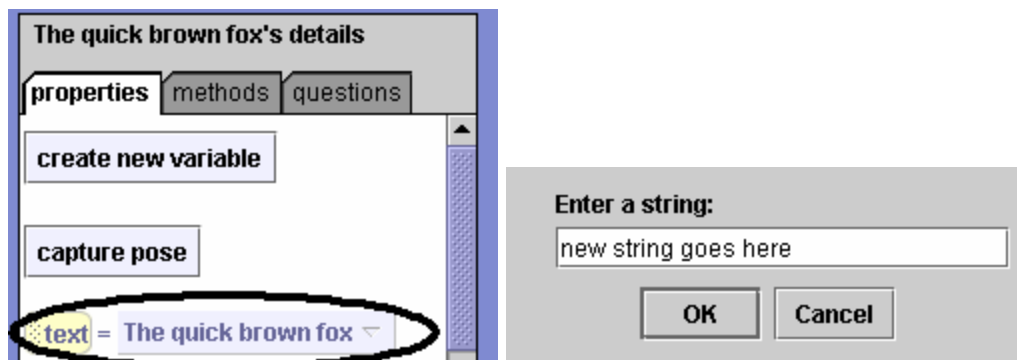


Figure T-1-10. Modifying text

Note that modifying the string in the text object does not modify the name of the object. The name is still as it was when object was originally created, as seen in T-1-11.



Figure T-1-11. Name of text object remains the same

2 Programming in Alice: Design and Implementation

In this chapter, we begin an introduction to programming using an “objects-first” approach. Every 3D object in Alice is an object! Writing a program to animate 3D objects in a virtual world is naturally all about objects and the actions objects can perform. From a practical viewpoint, writing a program is somewhat like working with word problems in a math course. In word problems, we first read the word problem (a description of the situation) and decide how to go about solving the problem (what steps need to be done) and then we solve it (write a solution). Similarly, in writing an animation program we first read a *scenario* (a description of the story, game, or simulation) and decide how to go about creating the animation (*design a storyboard*) and then we write the program code (*implementation*).

Section 1 begins with scenarios and storyboards as a methodology for designing programs. Visual storyboards were chosen because they are the design tool used by professional animators in animation film studios. Textual storyboards were chosen because they provide an *algorithmic* (step-by-step) structure. The lines of text in a textual storyboard are similar to *pseudocode* – a loose version of the instructions that will eventually become program code.

Section 2 presents the basics of creating a simple program in Alice. The idea is to use the storyboard as a guide for writing lines of code in Alice’s drag-and-drop editor. We can focus on the step-by-step solution because Alice will automatically take care of all the details of syntax (sentence structure and punctuation). In an animation, some actions must take place in sequence and other actions simultaneously. This means the program code must be structured to tell Alice which actions to *Do in order* and which actions to *Do together*.

Note to instructors: We highly recommend that chapters 2 and 3 be covered in sequence. If you are using this text to teach/learn Alice in a non-“problem-solving” approach, you may skip section 1 of this chapter. However, storyboards will be used throughout the rest of the text as providing a framework in which to discuss design from an algorithmic, problem-solving perspective. You may choose to use a different design framework (perhaps using the UML or a more traditional version of pseudocode) – this may be done safely, without impacting the content.



2-1 Scenarios and Storyboards (Design)

Creating a computer program that animates objects in a virtual world is a two-step process. The first step is **design** (planning ahead) and the second step is **implementation** (writing the animation program). This section introduces the design step.

Design is an important part of constructing programs for animation. A design is a “plan ahead” strategy and takes practice to master. While the programs presented in the first few chapters of this text are reasonably clear-cut, we think it is advisable to start building good designs early on. Then, when programs begin to get more complicated, the time invested in learning how to design good program solutions will pay great dividends.

Two Design Components

A design consists of two components, a *scenario* and a *storyboard*. A scenario is a problem statement that describes the overall animation in terms of what problem is to be solved or what lesson is to be taught. Cartoons and feature-length animated films begin with a scenario created by professional writers. Sometimes, a scenario is called “the story.” As used here, in addition to the traditional meaning of “story,” a “story” can be a lesson to teach, a game to play, or a simulation to demonstrate.

At Pixar, Disney, and other major animation studios, animators break down a long scenario into sequences of many short scenarios. For each scenario, a storyboard is created to depict the sequence of scenes. The storyboard may consist of dozens of scene sketches, drawn by animation artists or generated by computer animation specialists using computer software. Figure 2-1-1 illustrates an example of a storyboard in Pixar’s

Figure 2-1-1. Example from Pixar

Scenario – provides description and information

As presented in chapter 1, a new Alice world is a simple scene of green grass and blue sky. To create an initial scene for an animation, character objects and background scenery are added and positioned in the world. But, before objects are added and positioned in the scene, answers to the following questions must be given:

1. What story is to be told?
2. What objects are needed? The objects include characters that play leading roles in the story as well as background scenery such as islands, mountains, and lakes.
3. What actions are to take place?

A scenario provides the information needed to answer these questions. That is, a scenario gives all necessary details for setting up the initial scene and then planning the storyboard sequence for the animation.

Scenario Example

Suppose you have recently been sitting at home, having missed another day of classes because of a winter snowstorm that dropped 2 feet of snow on the ground. You see some children outdoors creating snow-people. You are daydreaming about a dance you recently attended. Being a very creative person, your imagination gets carried away and the two scenes blend together:

Several snow-people are outdoors, on a snow-covered landscape. A DJ is playing the song “Let it snow.” A snowman is trying to meet a snowwoman who is talking with a group of her friends (other snowwomen.) He “makes eyes” at her, trying to get her attention. But, alas, she is not interested in dancing with him. She gives him a cold shoulder and turns back to talk with her friends.

From this scenario, we have answers to the questions posed in the previous paragraph. This scenario tells a sad story about a snowman’s unsuccessful attempt to flirt with a snowwoman at a winter dance. The objects are snow-people and the background scenery should depict a winter scene. The actions are the snowman making eyes at the snowwoman and the snowwoman giving the snowman a cold shoulder.

Visual Storyboard⁴

A visual *storyboard* breaks down a scenario into a sequence of major scenes with transitions between scenes. Each sketch represents a *state* of the animation – sort of a *snapshot* of the scene. Each snapshot is associated with objects in certain positions, colors, sizes, and poses. When one or more transitions (changes) occur in the animation, the transition leads to the next scene (state).

The scene snapshots are numbered in sequence and labeled with necessary information. For short animations, the breakdown might be presented on one large sheet of paper. For more

⁴ “Story board” is an animation term. In other areas of computer science, the term “state-transition-diagram” is used to refer to what we will describe here as a storyboard.

complex designs, a separate sheet of drawing paper might be used for each scene snapshot to allow the animation artist to easily rearrange or discard scenes without starting over.

Generic Template

A generic scene template is shown in Figure 2-1-2 below. Each snapshot is labeled with a **Scene Number** and contains a **Sketch** showing where the objects are in the scene. The **Description** tells what action is occurring. If sound is appropriate in the animation, the **Sound** field is used to list sound clips that will be played during the scene. If a comic-book style is desired, the **Text** field is used to show the words or phrases that will be displayed. Sound and/or Text are used only if needed.

Scene Number: _____
<div style="border: 1px solid black; width: 300px; height: 100px; margin: 10px auto; text-align: center; vertical-align: middle;">(sketch)</div>
<hr/> Description _____ _____
Sound: _____
Text: _____

Figure 2-1-2. Generic storyboard template

Hand-Sketched Example

One technique we can use to create a visual storyboard is borrowed from professional animators -- a sequence of hand-drawn scenes. For our purposes, preparing storyboard sketches is not intended to be a highly artistic task. Simple circles, squares, and lines can be used to represent the objects that will appear in the scene. If necessary, shapes can be labeled with the name of the object or color-coded.

To illustrate the creation of a storyboard, the sample scenario for the snow-people party (described above) will be used. The sketch in Figure 2-1-3 shows a simple drawing of a scene where the snowman makes eyes at the snowwoman of his dreams. Circles were used to create the snowman and snowwoman. Diagonal lines were drawn to create a background of mountains in the distance. The grey squiggly lines were put in to represent the surface of the snow-covered ground. Using simple figures, hand-sketched storyboards are quick and easy to create.


Scene Number: <u>2</u>

Description <u>Snowman's eyes roll up and</u> <u>down</u>
Sound: <u>Let It Snow</u>

Figure 2-1-3. Hand-drawn sketch

Scene-editor-generated Example

A second technique, also borrowed from professional animators, is to create a visual storyboard using Alice's scene editor to add objects to a world and then patiently arrange the objects in various poses. As each successive scene is created, a screen capture is made and sent to the printer. The sketches shown in Figures 2-1-4(a) through 2-1-4(c) illustrate a scene-editor-generated storyboard for the beginning of the snow party animation. Naturally, scene-editor-generated sketches for a storyboard are more impressive than hand-drawn sketches. (This is the technique we used to prepare visual illustrations for the figures in chapters throughout this book.) But, the hand-sketched drawings are much faster and easier to put together.

Textual Storyboard

While professional animation artists use visual storyboards as part of their project development process, not everyone has the inclination to make dozens of sketches, either hand-drawn or computer-generated. A textual storyboard is a good alternative to visual storyboards. A textual storyboard looks something like a "to-do list." While sketched and computer-generated storyboards provide a visual representation of the sequence of scenes, a textual storyboard allows us to prepare a planned structure for writing program code. To take advantage of each of these strengths, both visual and textual storyboards are used in this book.

Scene Number: 1



Description

The initial scene. The snowman is interested in meeting the snow woman on the left

Sound: __Let it snow__

Text: ____none____

(a) Scene 1

Scene Number: 2



Description:

The snowman tries to get the snow woman's attention

Sound: Let it snow

Text: Ahem....

(b) Scene 2

Scene Number: 3



Description:

The snow woman notices the snowman.

Sound: Let it Snow

Text: none

(c) Scene 3

Figures 2-1-4 (a)-(c). Scene-editor-generated storyboard scenes

Textual Storyboard Example

A textual storyboard for the snow-people winter party is illustrated in Figure 2-1-5. An important point that should be mentioned about textual storyboards is that one textual storyboard may encompass more than one scene from a visual storyboard. For instance, the Figure 2-1-5 textual storyboard includes the actions shown in all three sequential scenes of the visual storyboard in Figures 2-1-4(a) through (c).

Do the following steps in order

- 1) Snowman point at the snowwoman
- 2) *Do the following steps together*
 - a) Snowman calls out to the snowwoman
 - b) Snowman makes eyes at the snowwoman
- 3) Snowwoman's head turns to see who is calling her.

...

Figure 2-1-5. Snow-people textual storyboard

The lines of text in the textual storyboard provide an ordered list of actions that will take place in the animation. Notice that the textual storyboard has two lines that are displayed in italics. These lines are for organizing the actions – some actions are to be done in order (one at a time), others are to be done together (simultaneously). Indentation is used to make the storyboard easy to read. The snowman makes eyes at the snowwoman at the same time that he says “Hey there!” But, the snowwoman's actions are done in sequence. After the snowman performs his actions, the snowwoman turns around to see who said “Hey there!”

In Computer Science terminology, a textual storyboard is called an *algorithm* – a list of actions to perform a task or solve a problem.

Evaluate and Revise

Once a storyboard has been designed, it is a good idea to take an objective look at the design to decide what might be changed. Evaluate the storyboard by answering these questions:

- Does the action flow from scene to scene, as the story unfolds?
- Do any transitions need to be added between scenes to blend one scene to the next?
- Did you overlook some essential part of the story?
- Is there something about the story that should be changed?

The important idea is that the storyboard is not written in concrete. We should be willing to modify plans to get a better effect or to fix something that doesn't seem to work as expected.

Warning!!!!!!

An alarm must be sounded about storyboards. It is quite possible to create a storyboard only to discover that the animation tool being used does not support some of the special effects that were planned. This happens to everybody – no matter how well you think you have planned and no matter how sophisticated the animation tool may be. Such situations require (1) revision of the storyboard, or (2) writing new program code to create new animated actions. Writing program code is the topic of the next section of this chapter.

2-1 Exercises

1. Create your own scenario.

In many of the exercises in this text, you will be given a scenario as part of the assigned exercise. But soon, you will likely want to be creative in a project where you dream up some animation on your own. To help you prepare for such a creative project, this assignment is to create your own scenario. A scenario is a general description of the scene and the animation that is to unfold. In this chapter, the scenario was a party scene involving snow-people.

Here are some ideas to help you in thinking about designing your own animation:

Think about things you have recently read or seen on television.

Daydream.

Talk with friends to brainstorm ideas.

Jot down quick notes, or doodle.

Consider the following:

Who is the intended audience?

What do you want to say to this audience?

What do you want this audience to experience, what story do you want to tell, what lesson do you want to teach?

Would you like to create a game-like animation or a simulation that allows the user to control the animation?

One good way to make up your own scenario is to work with a teammate. In working with another person, you can brainstorm ideas. Or, if you write the scenario by yourself, you can still exchange scenarios with another person to give each other feedback. You can combine ideas from two or three different sources and come up with something new. Or you can take something you have seen and give it a different ending or a different interpretation.

Write a scenario that is at least 4 or 5 sentences. Be sure the scenario includes information that answers the questions presented in this chapter section.

2. Creating a Storyboard

In each of the following, you are asked to create a storyboard for the scenario you created in question 1 above. While the **scenario** is a verbal description of *what will happen* in an animation, a **storyboard** is a plan or a *design for how the animation will be written*. A storyboard can be either a sequence of sketches or it may be a textual to-do-list.

(a) Create a visual storyboard for the scenario you created in exercise 1 above. The visual storyboard can be hand-sketched or scene-editor-generated. This visual storyboard should create a sketch using the generic storyboard template for each major scene. See Figure 2-1-4(a) through (c) as an example.

(b) Create a textual storyboard (to-do list) for the scenario you created in exercise 1 above.

3. Creating a Storyboard from a given scenario

Create a storyboard for *one* (your choice) of the following scenarios:

(a) A scenario for a child's game: Alice, the white rabbit, and the Cheshire cat enjoy a game of musical chairs in a tea party scene. Every so often, one of the characters yells "switch" and they all run around the table to stand beside the next chair. After each switch, a chair is tipped over and the character standing next to it is eliminated from the game. The last character left is the winner of the game.

(b) A scenario for a video game: A jet fighter plane is returning to the carrier deck after a training mission. The plane makes a half-circle around the carrier to get into position for landing and then gradually descends toward the carrier. Of course the carrier is in motion, so the plane has to continually adjust its descent to finally land on the deck. After the plane touches down on the carrier, it continues across the deck to finally come to a halt.

2-2 A First Program

In the previous section, we learned how to design an animation using a scenario and a storyboard. Now, we are ready to look at how an animation program can be written. This step in building an animation is called *implementation*.

What is a Program?

A *program* is a list of actions (instructions). We can think of an Alice program as being somewhat like a *script* for a theatrical play. A theatrical script tells a story by describing the actions to be taken and the words to be delivered by actors on stage. In a similar manner, an Alice program prescribes the actions to be taken and the sound and text to be used by objects in a virtual world.

Create an Initial Scene

An ancient Chinese proverb is "The longest journey begins with a single step." Let's begin our journey by implementing the Snow-people animation described in the previous section. Recall that a snowman is interested in meeting a snowwoman. He flirts with her (makes eyes at her), but she isn't interested in meeting him, so she turns away. The first step is to create the initial scene. To a new world, several snowmen and snowwomen (found in the People collection in the local gallery) are added. Then, a wintry backdrop (Sets and Skies collection of the local gallery) is added, and the ground is turned white. The initial scene is shown in Figure 2-2-1.



Figure 2-2-1. Snow-people initial scene

Program Code Editor

Once the initial scene has been set up, the instructions that make up the program code must be written. Alice provides a program code editor -- the large yellow rectangle at the lower right of the main Alice window, as shown in Figure 2-2-2. The instructions for a program are entered in the editor. Once a program is written, we would like to run the animation many times without having to write the program again. As such, it is necessary to save ("store") the instructions that make up the program in a file. At a later time, the file can be reopened and run again.

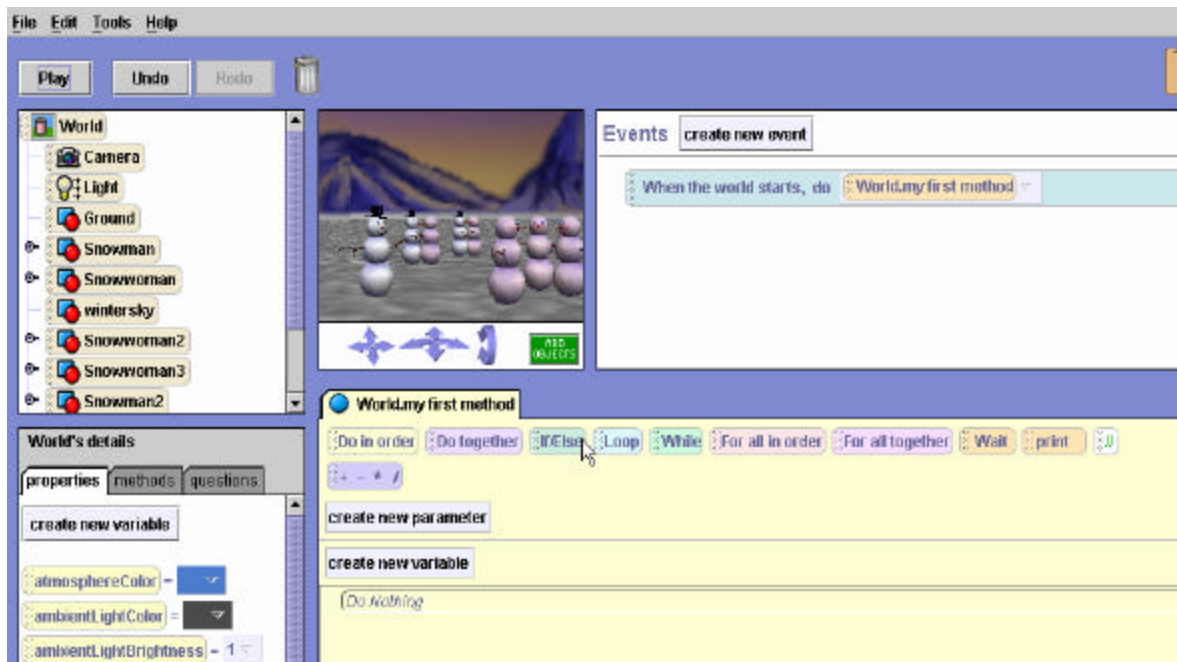


Figure 2-2-2. Program code editor at lower right of the main window

World.my first method

As seen in Figure 2-2-2, the tab for the editing area is labeled *World.my first method*. A *method* is a segment of code in a program that performs a specific task. Alice automatically uses the name *World.my first method* for the first editing pane. Actually, any name can be made-up and used for a method name. But, we will just use the name *World.my first method* for this example. The Snow-people world is a simple program and all the program code can be written in *World.my first method*. By default, when the Play button is pressed in Alice, the *World.my first method* will execute.

What instructions are needed?

The program code editor (hereafter referred to as "the editor") will be used to write the instructions for the Snow-people program. What instructions must be written and in what order? This is where the storyboard is used to help plan the instructions and the order. A complete version of the textual storyboard for the Snow-people animation is shown in Figure 2-2-3.

<i>Do in order</i>	
1)	Snow man looks at the snow woman
2)	<i>Do together</i>
	Snowman calls out to the snowwoman
	Snowman makes eyes at the snowwoman
3)	Snowwoman turns to see who is calling her.
4)	<i>Do together</i>
	Snowwoman blushes (her head turns red)
	Snowwoman turns back to her friends
5)	Snowwoman's face turns back to white

Figure 2-2-3. Snow-people storyboard

The program should probably consist of several *move* and *turn* instructions. Of course, the first step is where the program begins. It seems reasonable to begin by making the snowman look at the snowwoman. A *point at* instruction can be used to make the snowman appear to look at the snowwoman. Then, a *say* instruction can be used to make the snowman call out to the snowwoman as he raises and lowers his eyes. Clearly, these actions must occur in a specific sequence.

Sequential versus Simultaneous actions

Pause to consider that some actions occur *sequentially* (one after the other) but other actions occur *simultaneously* (at the same time). In this example, the snowman should look (point) at the snowwoman before calling out to her. This seems too obvious; but Alice must be told to *Do* these instructions *in order*. Also, the snowman is to call out to the snowwoman while raising and lowering his eyes at the same time. To make the snowman call out to the snowwoman at the same time he is raising and lowering his eyes, Alice must be told to *Do* these actions *together*.

Do in order

To tell Alice to do instructions in sequential order, a *Do in order* block is created in the editor and instructions are placed within the block. The *Do in order* tile is dragged into the editor, as seen in Figure 2-2-4.

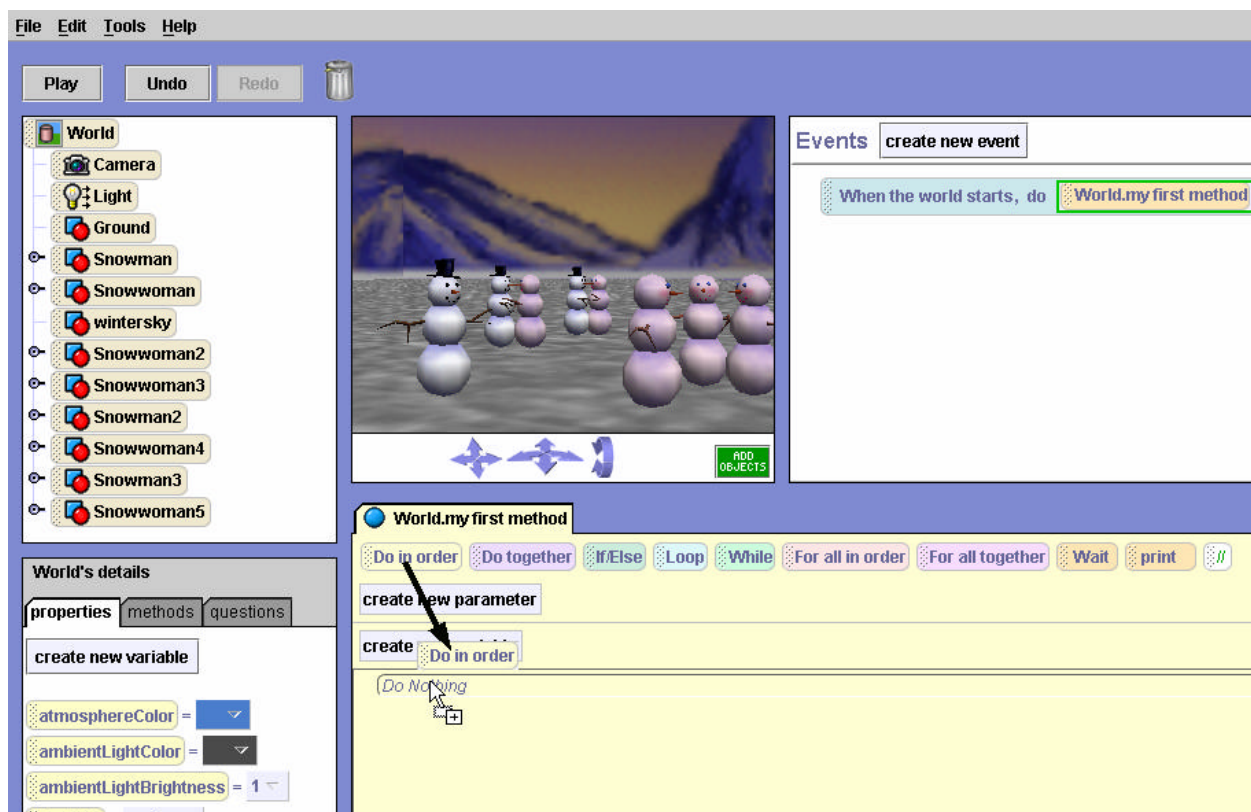


Figure 2-2-4. Placing a *Do in order* block in the editor

The instructions to perform the animation will now be placed within the Do in order block. The first instruction is to make snowman point at (look at) the snowwoman. The snowman is selected in the object tree (in the upper left hand side of the Alice window). Then, in the snowman's methods tab (located in the details pane immediately below the object tree), the *point at* instruction is selected and dragged into the *Do in order*, as shown in Figure 2-2-5.

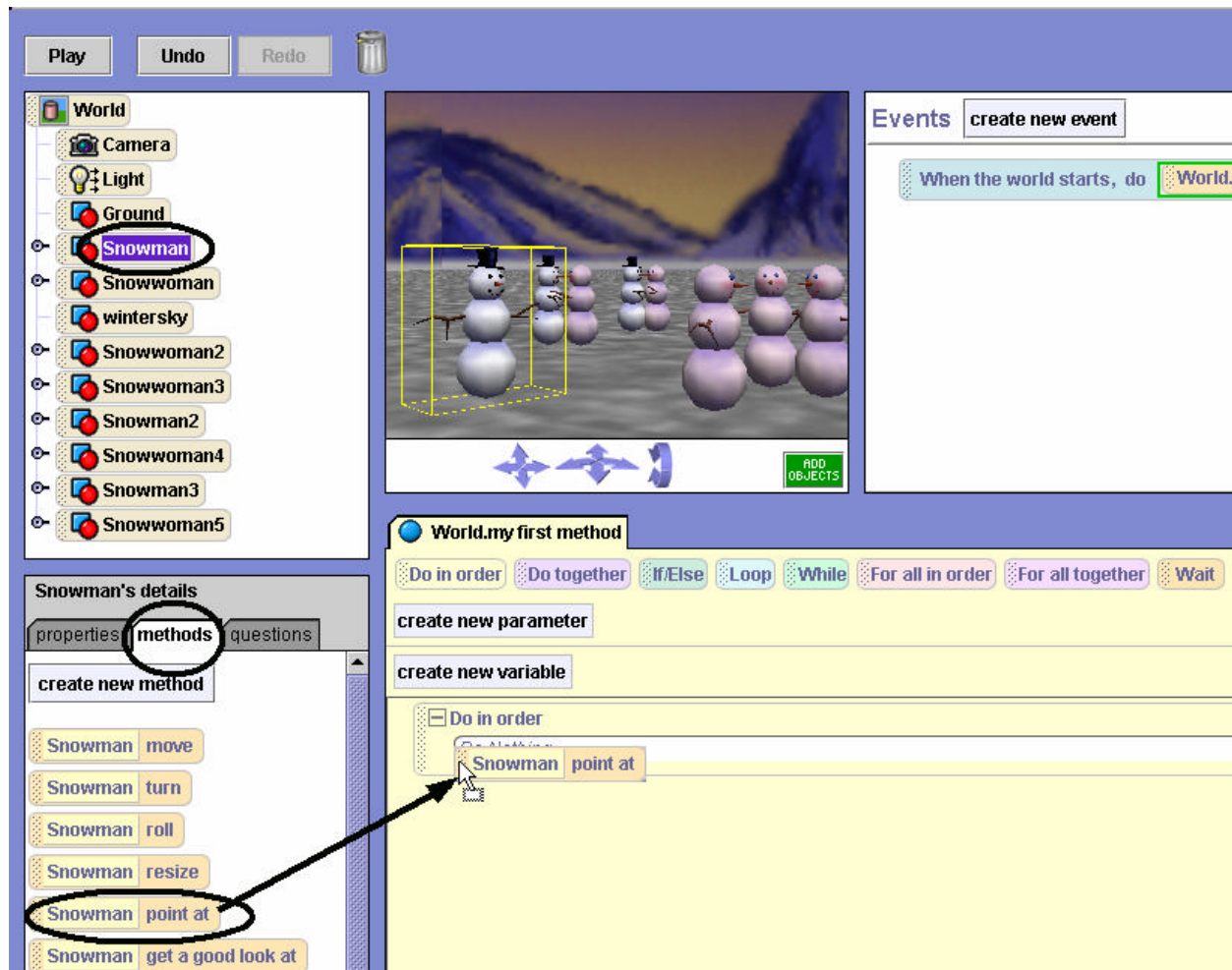


Figure 2-2-5. Adding a *point at* instruction

The *point at* instruction requires a parameter, namely which object the snowman should *point at*. (A *parameter* is an item of information that must be supplied so Alice can execute the action.) In this example, the snowwoman is selected as the object to *point at*. The result is illustrated below.



Do together

The next step in the storyboard requires two things to occur at once: the snowman saying “Hey there” to the snowwoman, and the snowman raising and lowering his eyes. A *Do together* block

is dragged into the *Do in order*, as shown in Figure 2-2-6. The result of this modification is the *Do together* block is nested within the *Do in order* block. Note that nesting the *Do together* inside the *Do in order* just happens to be the best way to animate this example. A *Do together* does not have to be inside a *Do in order*. These two coding blocks can work together or can work separately in many different combinations. It is also worthwhile to notice the horizontal black line in Figure 2-2-6. The black line indicates where the *Do together* instruction will be dropped.

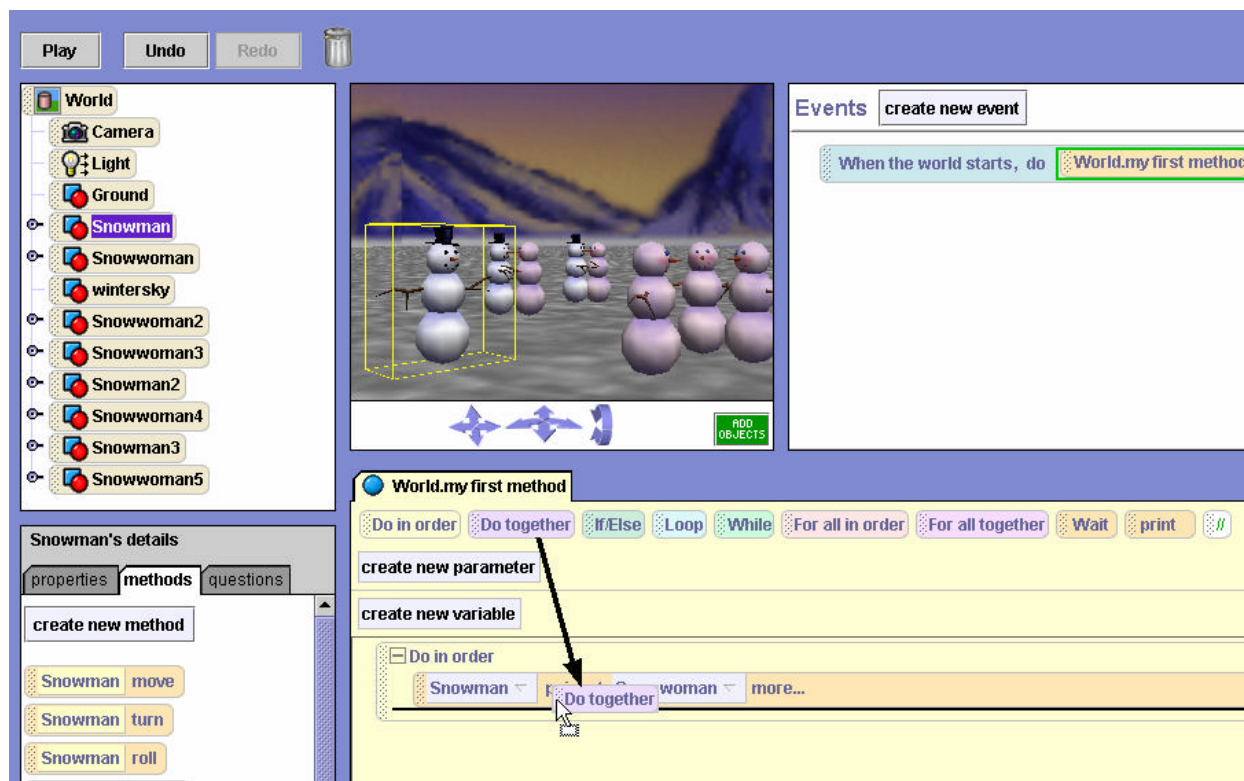


Figure 2-2-6. Adding a *Do together* (inside the *Do in order*)

The result should appear as illustrated below:



Now, instructions can be dragged into the *Do together* block to make the snowman say “Hey there” and flirt (raising and lowering his eyes). The say instruction is easy -- just drag the instruction into the *Do together* block, and select a “Hey there” string. The resulting code appears below:



Issuing instructions for the snowman's eyes is more complicated. A mouse click on the key to the left of the snowman in the object tree allows access to the snowman's parts. Then, a mouse click on the key to the left of the snowman's head, allows access to the snowman's eyes, as shown in Figure 2-2-7.

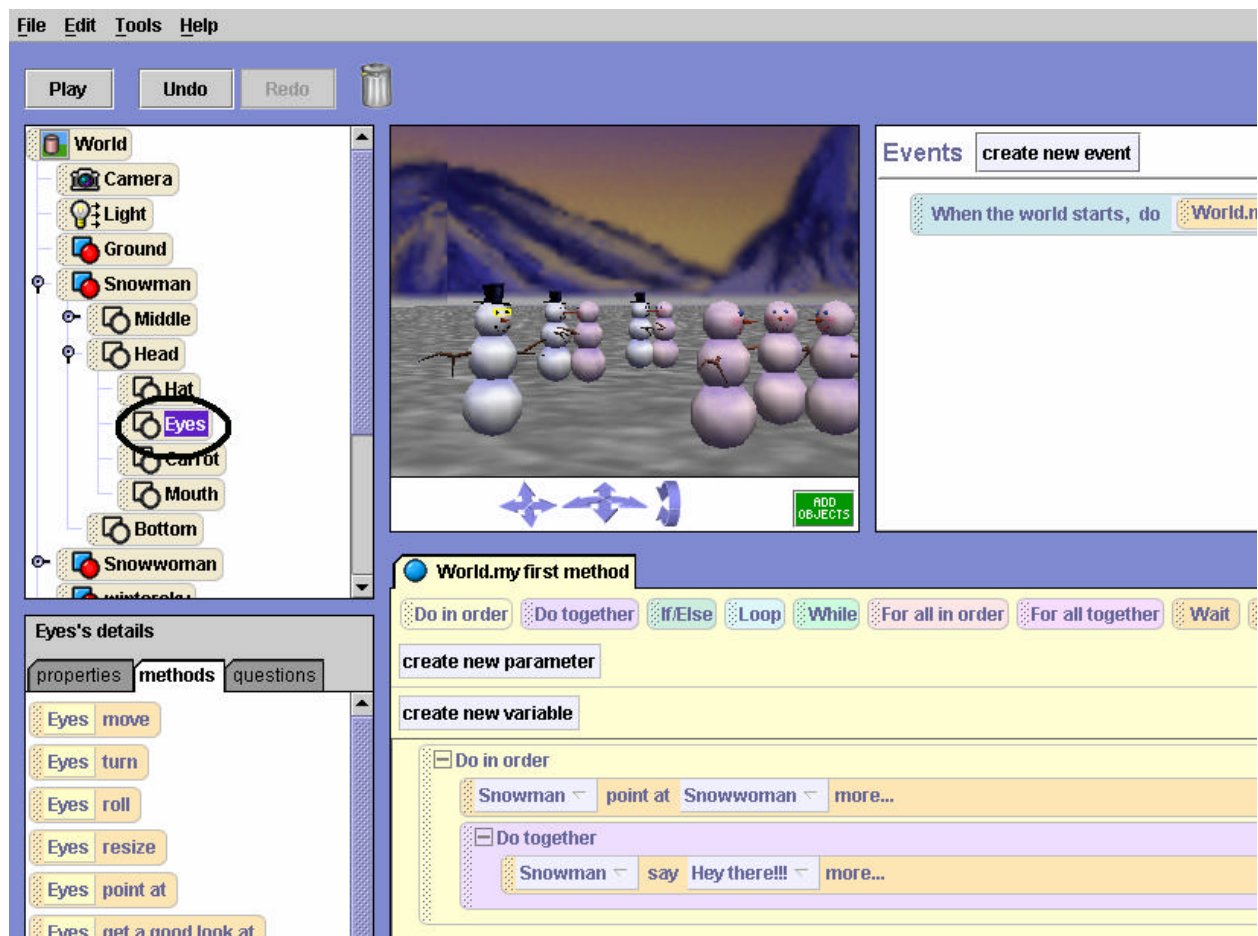


Figure 2-2-7. Accessing the snowman's eyes

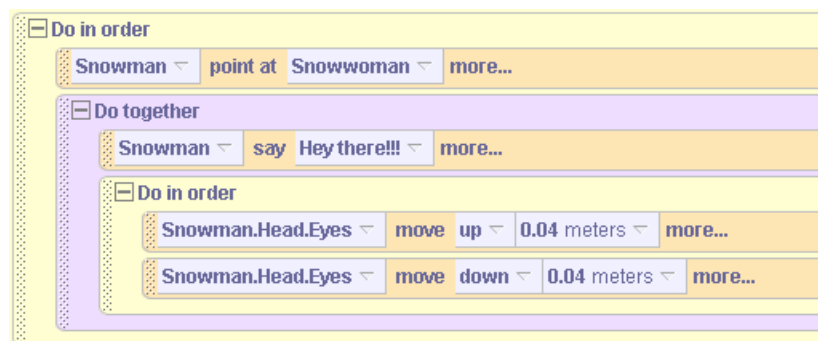
It is now possible to drag instructions into the editor to move the snowman's eyes up 0.04 meters and then down 0.04 meters, producing the code shown below. Note that the direction of movement is required, as is the distance (which is entered by selecting *Other*, and then entering a value of 0.04 on the calculator). How did we know to use 0.04 meters as the distance? Well, we didn't. We just tried several different distance values until we finally found one that worked to give us the best effect. This is an example of a *trial and error* strategy. While we always recommend good planning strategies, sometimes trial and error is useful tool.



Time out for Testing and Debugging

Whenever several lines of code have been written, it is generally a good idea to test it. So, to test the instructions, as written thus far, the **Play** button is clicked. The snowman turns to point at the snowwoman, and the snowman says “Hey there!!!” but his eyes don’t move up and down. In fact, they do not appear to move at all!

The reason they do not move is that the program has a *bug*. (Errors in computer programs are generally referred to as bugs.) The problem is that the snowman has been given instructions to move his eyes up and down *at the same time*. These two animation instructions effectively cancel each other out, and the snowman’s eyes do not move at all! To fix this problem, it is necessary to place the moving of the snowman’s eyes within a *Do in order* block, as illustrated below.



Now, when this program is run, the snowman’s eyes move up and down! There is one other useful observation to make. Animation instructions in Alice, by default, require one second to run. Normally, within a *Do together* block, it is desirable that all of the instructions take the same amount of time. Since the snowman saying “Hey there!!!” takes one second, it is desirable that the moving of his eyes up and down also requires one second for both instructions. Thus, it is desirable that moving his eyes up (and moving his eyes down) each require $\frac{1}{2}$ a second. To change the duration of an animation instruction, we click on *more...* (to the right of the instruction where the duration is to be changed), select the duration option, and change its value to $\frac{1}{2}$, as shown in Figure 2-2-8.

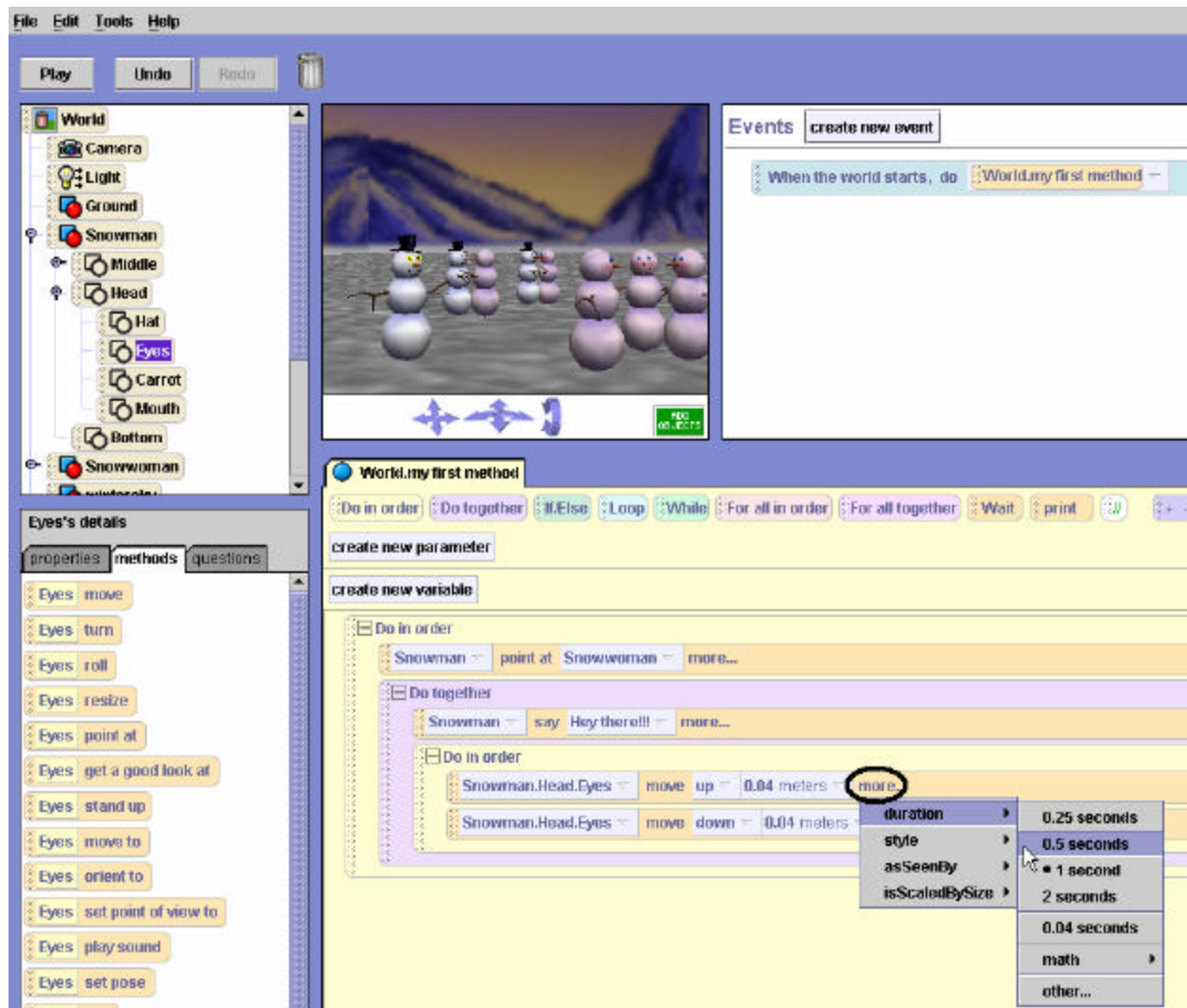


Figure 2-2-8. Changing the duration of an instruction

Completing the animation

It is still necessary to complete the final three steps described in the storyboard. The third step is for the snowwoman to turn her head to the right. An instruction to turn the snowwoman's head 0.40 of a revolution to the right is added to the code, as shown below.



The fourth step requires the snowwoman to blush (her head turns red) as she turns her head back to her snow friends. Another *Do together* block will be required for this step. Making the snowman's head change color is slightly different from other instructions we have used so far. To change the color of the snowwoman's head, the color property must be changed directly. To do so, we first select the snowwoman's head in the object tree. Then, the property tab is clicked in the details pane, and finally the color tile is dragged into the *Do together* block, where a value of Red is selected, as illustrated in Figure 2-2-9.

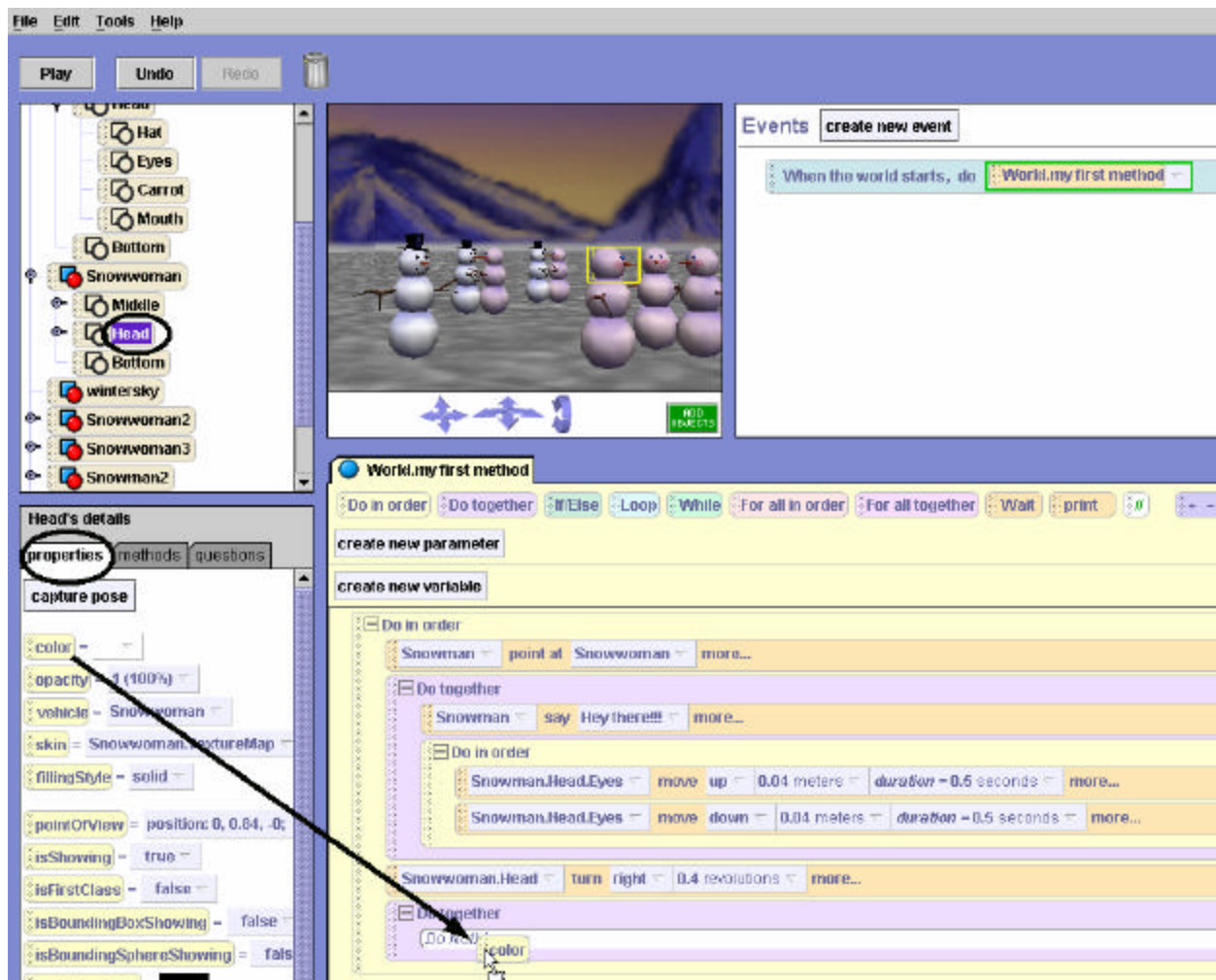
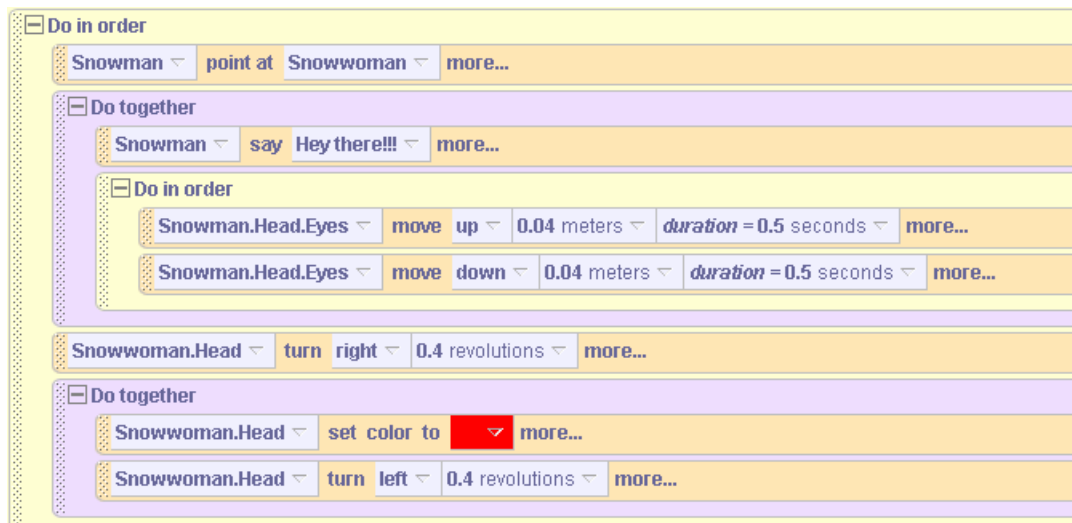
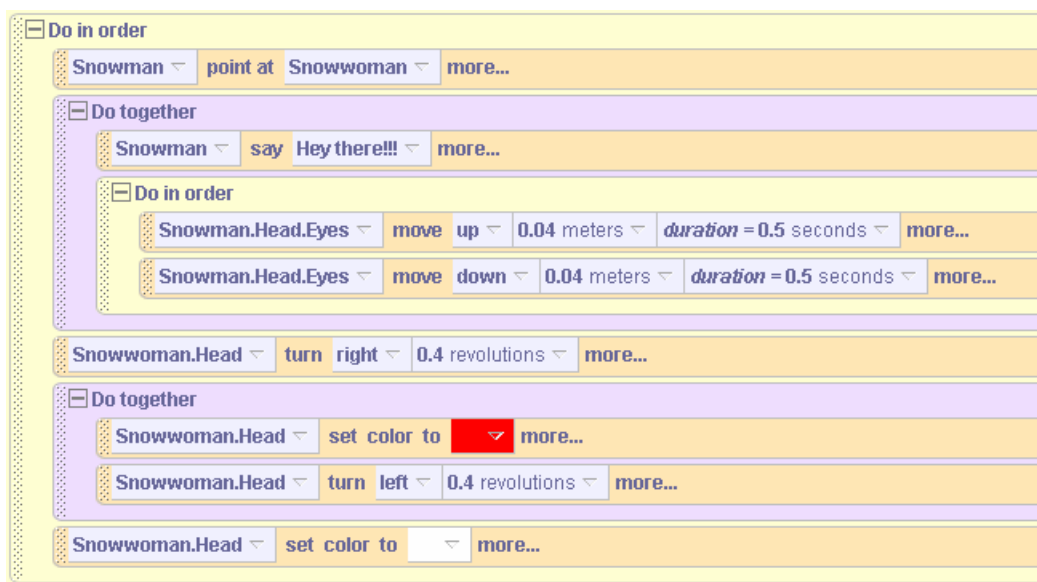


Figure 2-2-9. Changing color

The resulting code is illustrated below.



Finally, the last step of this animation may be added – where the snowwoman’s head returns to a white color. **The resulting code for the entire animation is listed below.**



Technical Note

It may seem a bit strange that the fifth animation step should be to set the color of the snowwoman’s head back to white, when her head and body originally had a somewhat pinkish appearance. The reason has to do with the original form of 3D models. In the process of preparing the 3D models for the gallery, the models are textured and painted in the 3D rendering software. By default, each model in the gallery is registered with Alice as having a white color (in other words, Alice has not painted another color over the model). Changing the snowwoman’s head to red causes red to be painted over the surface of her head. Changing the color of her head back to white returns her head to the color it was originally, as specified by the model in the gallery.

2-2 Exercises

Reminder: Each time an animation is completed, be sure to save the world. A good way to remember which file contains a particular world is to name the file using the name of exercise.

1 Circling Fish

Create an island world with a fish in the water. Position the fish and the camera point of view so the scene appears as illustrated below.



Use the program code editor to write a program that has the fish swim around in a circle in front of the island. Next, have the fish swim around the island. Finally, have the fish jump out of the water and then dive down into the water. The final scene should look somewhat like the initial scene, with the fish back in roughly the same position where it started.

2 Penguin Gets Cookie

Create a world having the initial scene as shown in the picture below. Write a program to animate the penguin so it waddles to the stool to get a cookie someone left on top of the stool. Have the penguin show its thanks by waving its wings.



3. Open Door

Create a world having Alice standing in front of a closed door. Have Alice turn to face the door, turn the knob, open it, and walk through it. Then have her face the camera and wave.



4 Magnet Fun

Create a world where Alice has a magnet held out in her right hand. Add five metal objects of your choice to the world and one by one have Alice point the magnet at each object. As Alice points the magnet toward an object, have the object move toward the magnet. Have the last object be very large (like a car) so when Alice points at it, she instead is pulled towards the object while saying "Whoah" or "Yikes". (HINT: Consult Tips & Techniques 1 for how to use the *vehicle* property. This will help in making the magnet move as Alice's hand moves.)



2 Summary

This chapter introduced the fundamental concepts of programming in Alice. We began with an examination of design. The design tools include a scenario (description of the story, game, or simulation) and a storyboard. A scenario helps us set the stage – that is, it tells us what objects will be used and what actions they will perform. A storyboard breaks down a scenario into a sequence of scenes that provide a sense of the order in which actions will take place. Some actions in a program will take place in sequence (one after the other) and some actions simultaneously. Once prepared, a storyboard is used as a guide for implementation (writing the program code).

Important concepts in this chapter

- A scenario is a problem statement that describes the overall animation in terms of what problem is to be solved, what lesson is to be taught, what game played, or what simulation demonstrated.
- A storyboard can be visual or textual.
- A visual storyboard is a sequence of hand-drawn or computer-generated sketches that break down a scenario into a sequence of major scenes with transitions between scenes.
- Each sketch represents a *state* of the animation – sort of a *snapshot* of the scene – showing the position, color, size, and other properties of objects in the scene.
- A textual storyboard is somewhat like a to-do list, providing an algorithmic list of steps that describe sequential and/or simultaneous actions.
- A program consists of lines of code that specify the actions objects are to perform.
- All 3D characters in Alice are objects. We have them move about their virtual world by dragging their action instructions (methods) into the editor.
- In Alice, program code is structured in *Do in order* and *Do together* blocks to tell Alice which instructions are to be executed in order and which are to be executed simultaneously.

Tips & Techniques 2

Style

Most animation instructions end with a list of options named **'more...'**, as seen in the example below.



When **more...** is clicked, a pop-up menu appears. An important option is **style**. This allows the programmer to specify the way in which one animation instruction blends into the next instruction. The options are *gently* (begin and end gently), *abruptly*, *begin gently* (and end abruptly), and *end gently* (and begin abruptly). To get the right degree of “smoothness” of the animation, it is often worthwhile to experiment with style.

Sound

If your computer has a sound card, it is easy to use sound files with virtual worlds. Alice uses either MP3 or WAV sound files. The use of sound is not a requirement for most animations. But, in some worlds the use of sound enhances the effect of an animation. In the world shown in Figure T-2-1, a fan dancer is performing a dance accompanied by the sounds of a koto (a traditional Japanese musical instrument).



Figure T-2-1. Fan Dance

The first step in using a sound file is to import the sound into Alice. A sound can be imported into the properties list for the entire World or for a specific object, as illustrated in Figure T-2-2. In this world, the sound is intended only to accompany the fan dancer. So, we will import the sound file for the fan dancer.

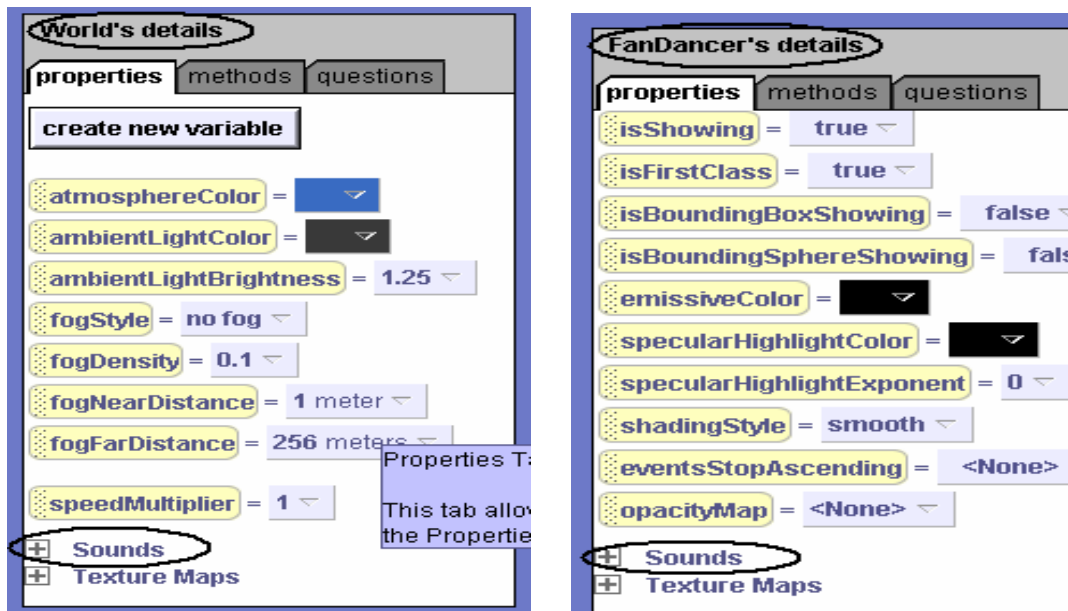


Figure T-2-2. Sounds In both World and Object details

To import the sound file, click on the *import sound* tile in the fan dancer's detail pane. A file selection box appears. Navigate to the directory where your sound files have been stored and then select the sound file to be used, as shown in Figure T-2-3. Note that it may be necessary to navigate through several folders on your computer to find the sound file to import. Once the file has been selected, click on the *Import* button. The name of the sound file will automatically appear in the list of imported files for the object, as seen in Figure T-2-4. The green arrow next to the name of the sound file is a preview button and can be clicked to test the sound file.

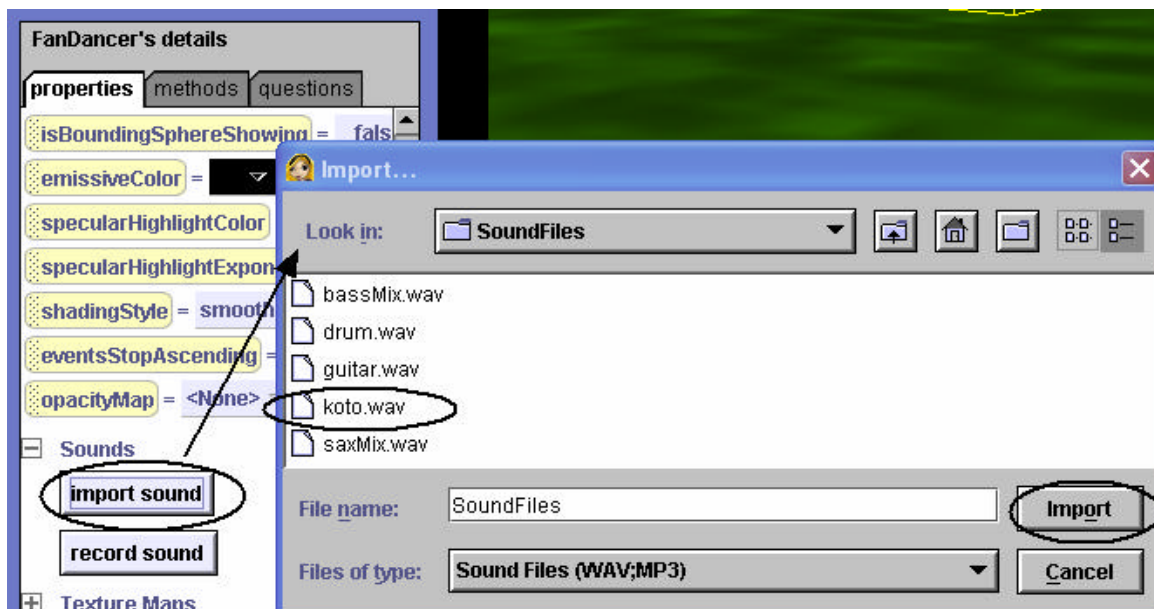


Figure T-2-3. FileSelection



Figure T-2-4. Imported Sound Files

To play the sound as part of an animation, select *playSound* from the object's methods in the details area. Then, drag it into the method editor and then select the name of the sound clip, as seen in Figure T-2-5. The time needed to play the entire sound will automatically appear in the instruction (circled in black). If the time is too long, you can clip the sound by selecting a shorter duration (circled in blue).

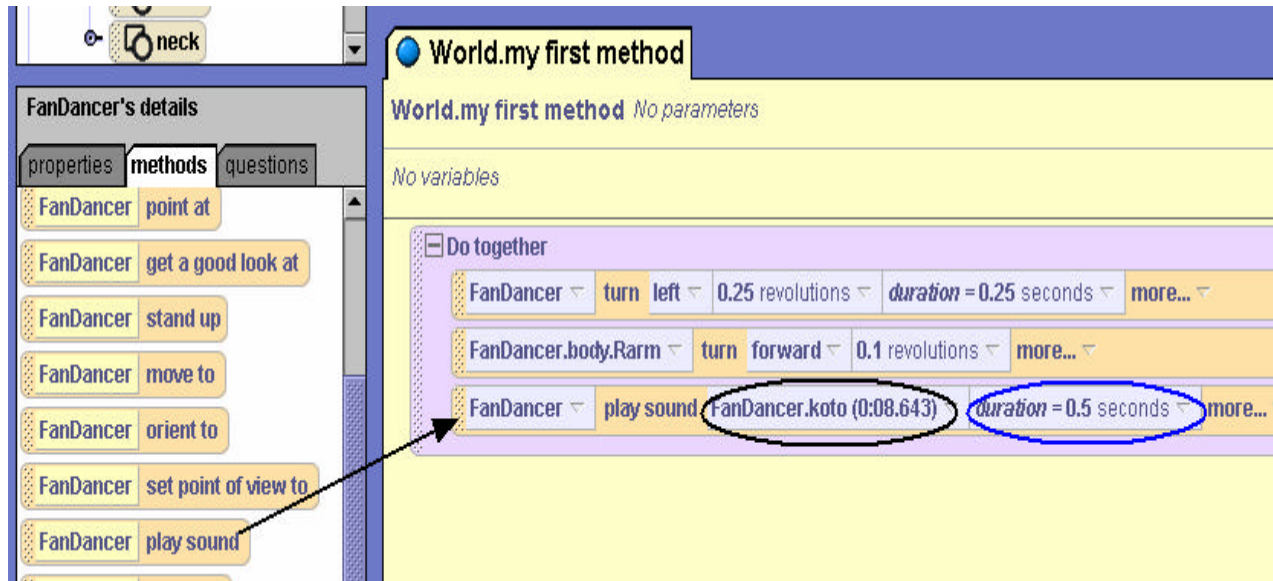


Figure T-2-5. Writing an Instruction to Play a Sound Clip

point at and onlyAffectYaw

A common action in animations is to move one object toward another object. The first step is to make the object that will be doing the moving turn and point at the object to which it will be moving. For example, in Figure T-2-6, the rowers in the lifeboat want to row towards the lighthouse on the island.



Figure T-2-6. Lighthouse and Boat Initial Scene

The point at instruction seems to be the logical statement to use to aim the boat at the lighthouse location.



While this instruction does turn the lifeboat towards the lighthouse, it also has the unfortunate effect of tipping the boat so it seems to be sinking on one end, as seen in Figure T-2-6.



Figure T-2-6. lifeBoat tips on *point at* Instruction

The tipping of the lifeboat is due the boat being aimed at the center of the lighthouse, which is higher in elevation than the boat. To avoid this effect, Alice provides *onlyAffectYaw* for the *point at* instruction. To use *onlyAffectYaw*, click on more (at the end of the instruction) and select *onlyAffectYaw* → true, as in Figure T-2-7. Selecting a true value for *onlyAffectYaw* allows an object to point at another object without affecting the pitch (forward-back orientation).

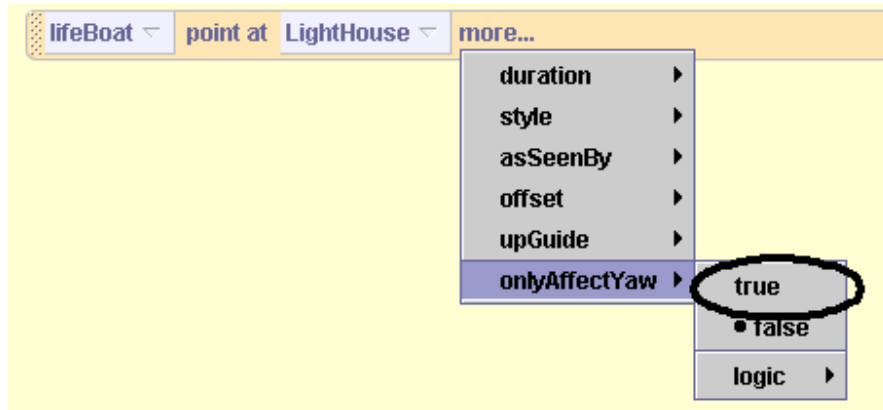


Figure T-2-7. Selecting onlyAffectYaw → true

Clipboards

This tip is not an animation tip at all. But, it makes programming in Alice much more pleasant so it is being shared as a tip. Suppose a particular animation sequence (an individual animation instruction or a collection of animation instructions) is something that you would like to copy or cut-and-paste. Control-C and Control-V (for those working on Windows computers) do not work in Alice. This is where the clipboards, illustrated in Figure T-2-8, come into play.

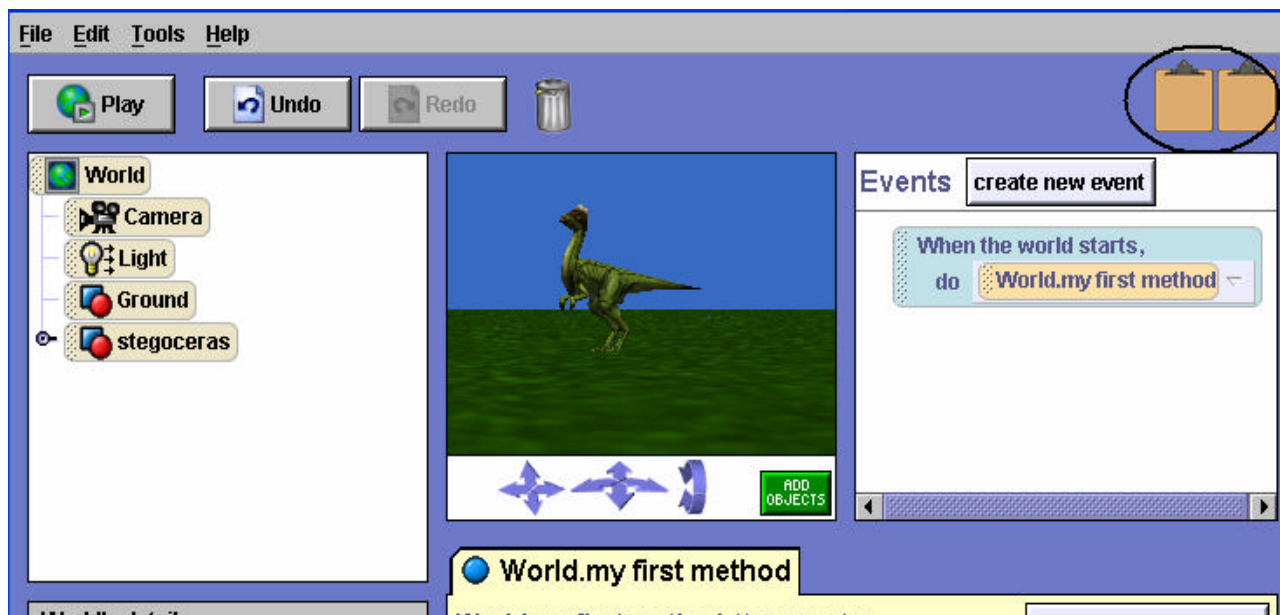


Figure T-2-8. The clipboards

To copy instructions from one location to another, left-click on the instruction or the instruction block. (Note that an instruction block is several instructions within a *Do in order* or within a *Do together*.) Then, drag the instructions to the clipboard. The copied-to clipboard changes color, as illustrated in Figure T-2-9. The color change is a visual clue indicating that copied instructions are now on the clipboard. Once instructions have been copied to the clipboard, you can use the mouse to drag the selected instructions to one of the clipboards.

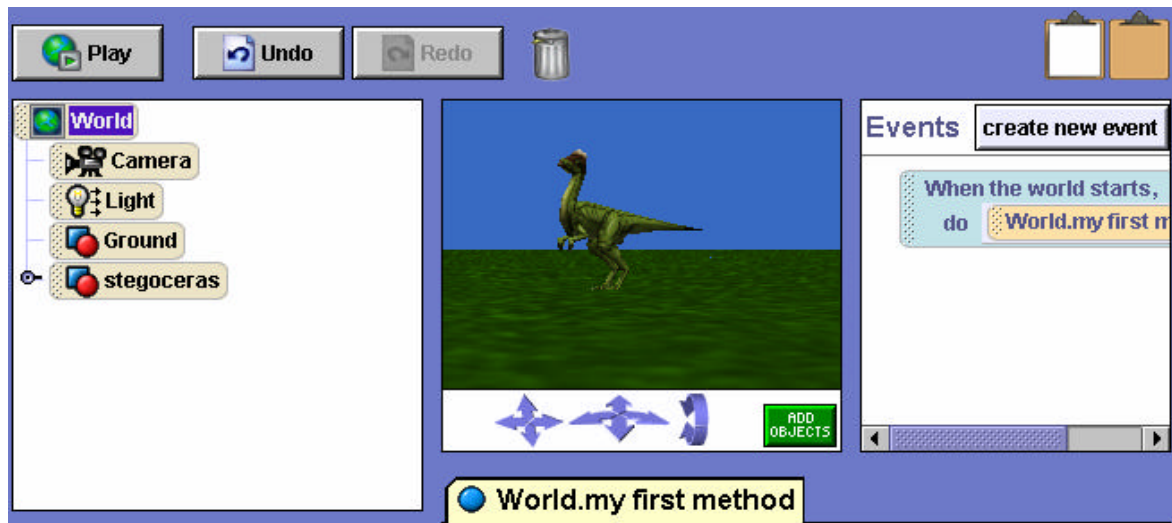


Figure T-2-9. A clipboard in use

Note: If instructions have been previously copied to that clipboard, copying a new set of instructions to the same clipboard will overwrite (destroy) what was already on the clipboard.

As seen by

As described in chapter 1, each object in Alice has its own sense of orientation. This is exactly what is expected in a 3D animation environment. But, sometimes an object's sense of orientation can lead to rather surprising results when an animation is being run. Suppose we have a helicopter on a pilot training mission, as shown in Figure T-2-10.



Figure T-2-10. Training Mission

We write code to roll the helicopter left and then move up:



Running the animation, we see that the result is not exactly what we had in mind. When the helicopter moves upward, it does so from its own sense of orientation, as seen in Figure T-2-11.

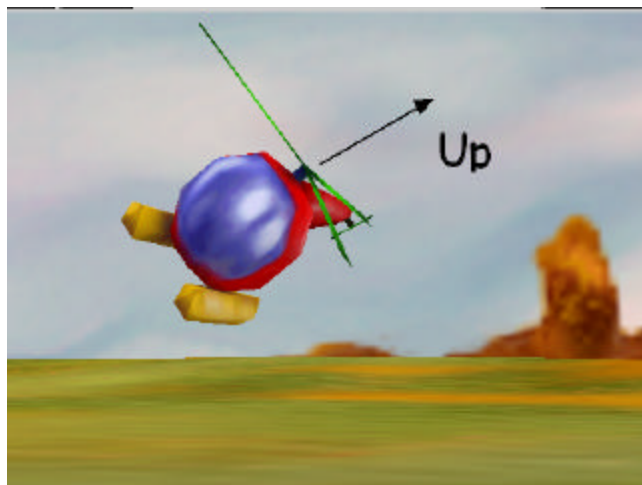
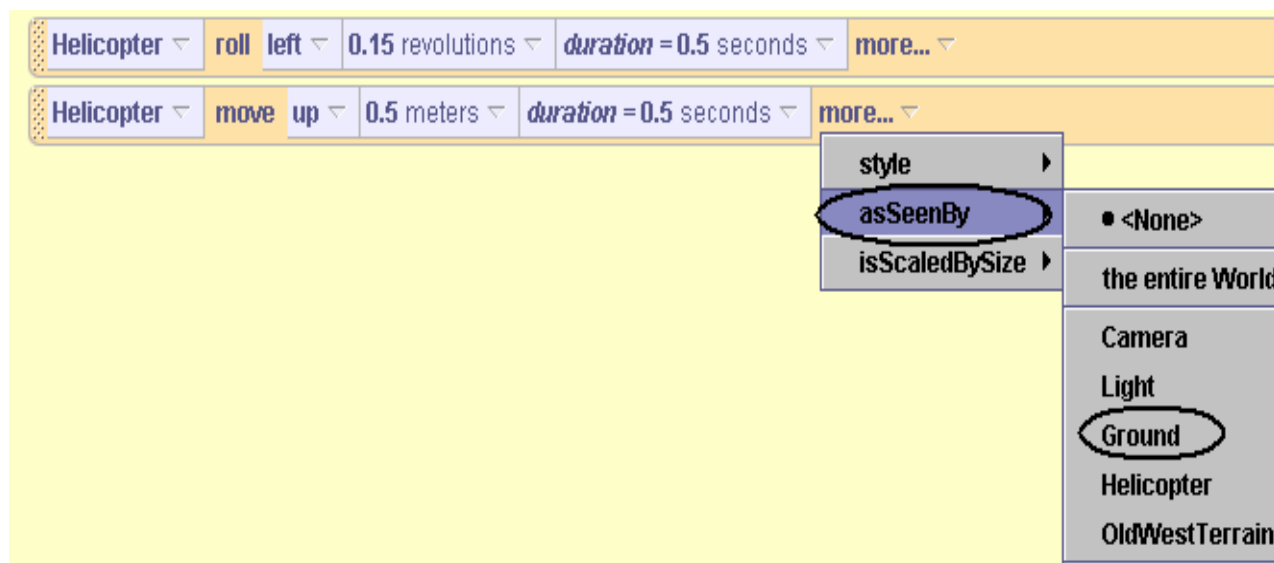


Figure T-2-11. Up – from the helicopter’s orientation

What we had expected, however, was an upward movement with respect to the ground. One way to make it easier to write code where we want to use an orientation of one object to guide the movements of another object is to use *asSeenBy*. In this example, we clicked on *more...* at the end of the instruction and then selected *asSeenBy* → *Ground*.



The resulting code, shown below, gives the desired movement.



3 Methods and Parameters

This chapter introduces the concepts of methods and parameters. Methods and parameters are key concepts in object-oriented programming. A method is a collection of actions that will be carried out by objects in the world when we request that the method be executed. A parameter acts like a basket to send information to a method.

The worlds created as examples and exercises in chapter 2 were relatively small animations constructed to introduce concepts of building and programming animations in virtual worlds. As worlds in exercises and projects grow larger, it will become increasingly important to use many, many methods. Methods provide a number of advantages to the programmer. For example, once a method is written it allows us to think about an overall task instead of all the small actions that were needed to complete the task. This is called *abstraction*.

In Alice, methods can be *character-level* (defined as an action for an object acting alone) or *world-level* (involving the actions of more than one kind of object). In Computer Science, character-level methods are called *class-level* methods. In programming languages such as Java or C++, world-level methods are often called directly from `main` – which is called when the program is run.

In section 1, our focus is on learning how to create and run our own world-level methods. To run our own method, it must be *called (invoked)*. Comments are used to document the code.

Section 2 launches a discussion of *parameters*. A parameter is used to send values and object names to a method – a form of communication between methods. The information that gets sent to a method can be of many different types (e.g., a number value, an object, or some property value such as color).

3-1 Methods: World-level

Every type of object in the Alice gallery has a repertoire of primitive actions it can perform—such as *move*, *turn*, *roll*, and *point at*. In Chapter 2, we wrote our first program as a collection of primitive instructions (for the snow man and snow woman). In this section, we will learn how to write intricate (and impressive) programs by writing and running new *methods*.

Methods

A method is a well-designed collection of instructions that will be carried out when requested. Of course we know that, in Alice, instructions are carried out by one or more objects in a virtual world. We also know that when objects carry out instructions, they may be acting alone – that is, not affecting or affected by other objects. On the other hand, when objects carry out instructions they may very well be interacting in some way with another object. When viewed this way, methods can be thought of as prescribing the behavior of objects. Methods that define behaviors for an object acting alone are considered object or *character-level*. (In object-oriented languages, these are called *class-level* methods.) Methods that involve actions by more than one object are considered *world-level* methods. In this section, we focus on world-level methods. Character-level methods are discussed in the next chapter.

Why do you care?

When you created your own animations in the exercises for the last section, no doubt you started to think about more complicated kinds of scenarios with more twists and turns in the storyline, perhaps games or simulations. Naturally, as the storyline becomes more intricate so does the program code for creating the animation. The program code can quickly increase to many, many lines of code – sort of an “explosion” in program size and complexity. Animation programs are not alone in this increase in complexity. Real world software applications can have thousands, even millions, of lines of code.

How does a programmer deal with huge amounts of program code? One technique is to divide a large program into several small modules, where each module performs a specific portion of the functionality of the overall program. In object-oriented programming, modules are used to define classes of objects and methods for those objects. Thus, methods play a major role in structuring the program code. Methods allow the programmer to think about a collection of actions as if it was just one action—this is called *abstraction*. Furthermore, each individual method can be tested to be sure it works properly. Finding a bug in a few lines of code is much easier than trying to find a bug in hundreds of lines of code. Not only do methods make a program easier to think about, they also make a program easier to read and debug.

Example

In our first program, a snow man tried to meet a snow woman at a winter dance. But, the snow man only tried once to attract the snow woman’s attention. It is perhaps more realistic that the snowman shouldn’t give up so easily. Perhaps he should try to catch the snow woman’s attention several times, speaking to her and making eyes at her repeatedly before she turns to look at him. Clearly, to make this change to the animation we will need to repeat the sequence of attention-catching instructions several times in the program.

Of course, it is possible to copy and paste instructions in the editor wherever needed. But, this becomes particularly tedious if it needs to be done many times. Wouldn't it be nice if these instructions could be grouped together to work as a method? Once the method is defined, Alice can be told to run that method several times, or from several different places in the program, without having to copy the instructions again and again into the editor.

Creating our own method

Let's write a method named *catchAttention*. This method will involve both the snow man and the snow woman, so it should be a world-level method. Although the instructions we will use are similar to those presented in the previous section, the animation will be constructed again from scratch, using a new world and a new initial scene. (The purpose of starting from scratch is to illustrate the complete process of writing your own method.)

In the object tree, the *World* object is selected and then the *methods* tab in the Details pane (located in the lower left of the screen). Then, the *create new method* tile (in the methods detail pane) is clicked. Figure 3-1-1 illustrates the *World*, *methods*, and *create new method* selections.

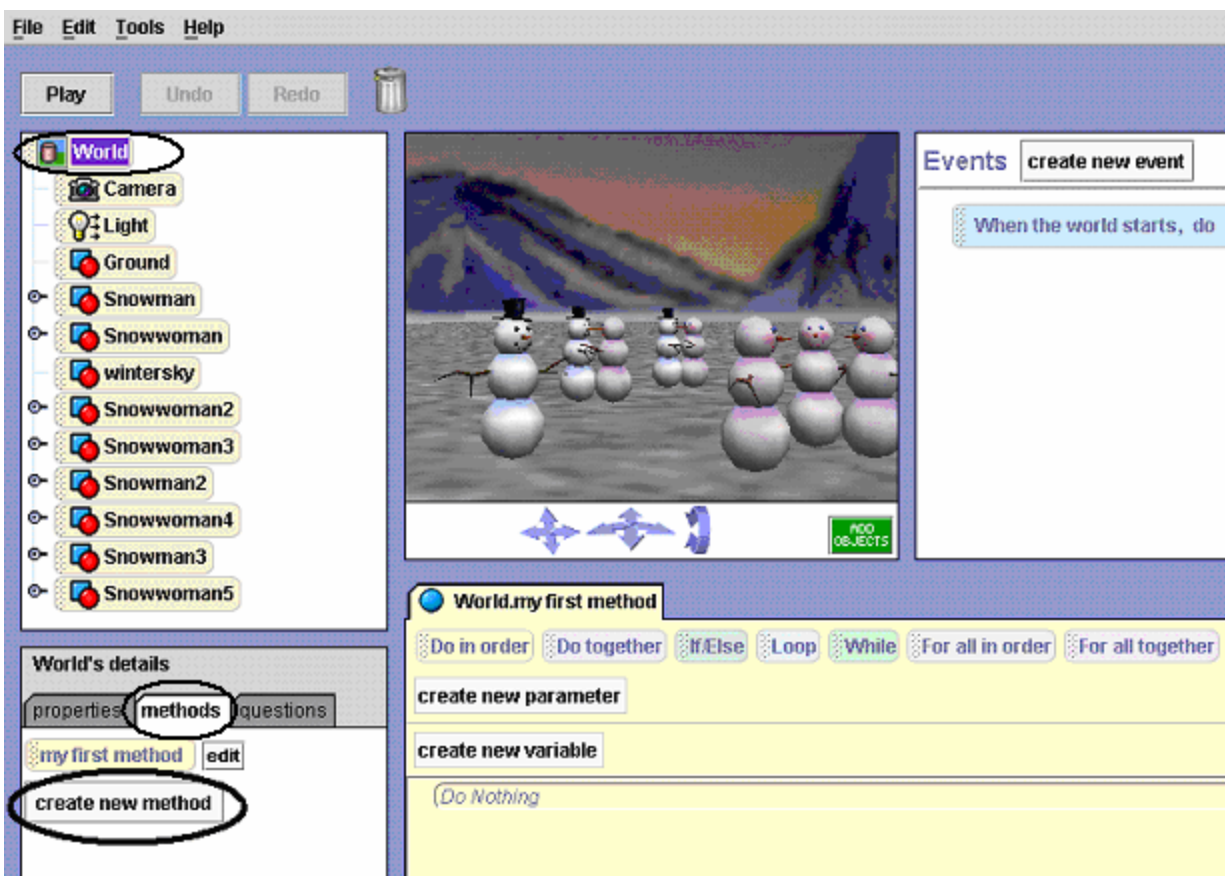


Figure 3-1-1. Selecting World's "create new method"

When the *create new method* tile is clicked, a new tile automatically appears in the method detail pane. Alice automatically names the new method with the simple term "**method**," as seen in Figure 3-1-2.

Replace the word "**method**" with a new name by highlighting the word "method" in the box and using the keyboard to enter the name "**catchAttention**", as shown in Figure3-1-3.

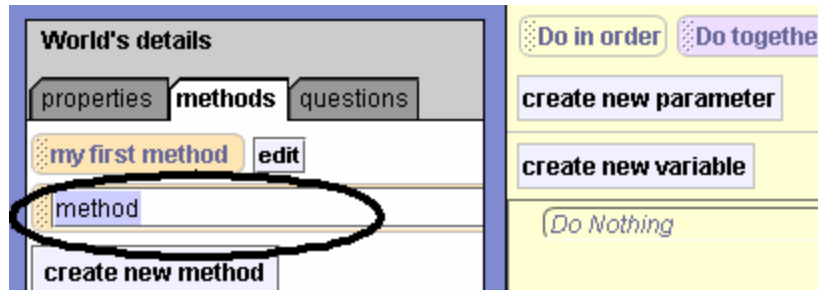


Figure 3-1-2. The Name Tile

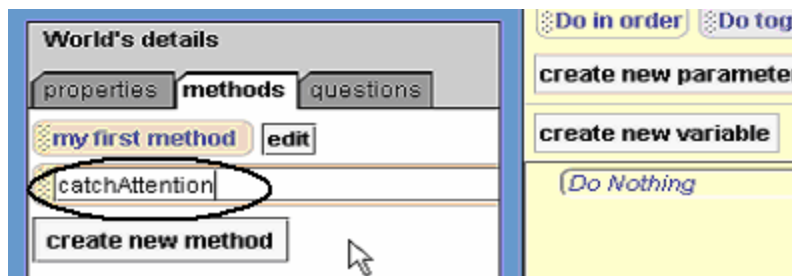


Figure 3-1-3. Renaming the New Method

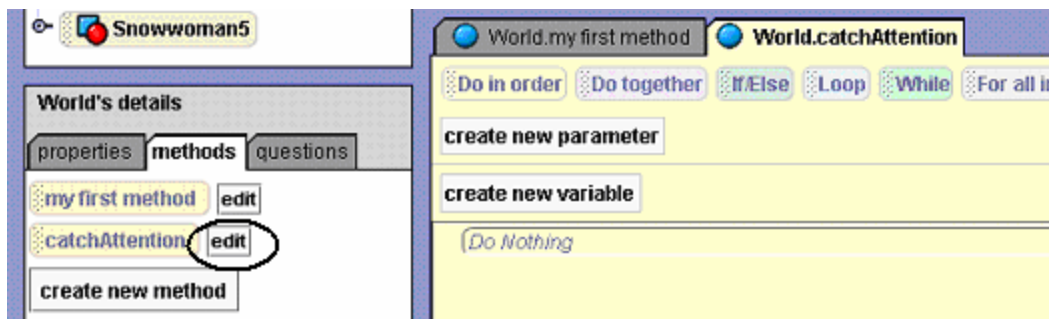


Figure 3-1-4. Selecting *World.catchAttention* for editing

Clicking on the *edit* button to the right of the *catchAttention* tile in the details opens a new pane in the editor to allow code to be written for the method. (See Figure 3-1-4.) Note that the particular method being edited has its tab colored yellow, and that all other method tabs are grayed out. The instructions for the *World.catchAttention* method are illustrated in Figure 3-1-5.

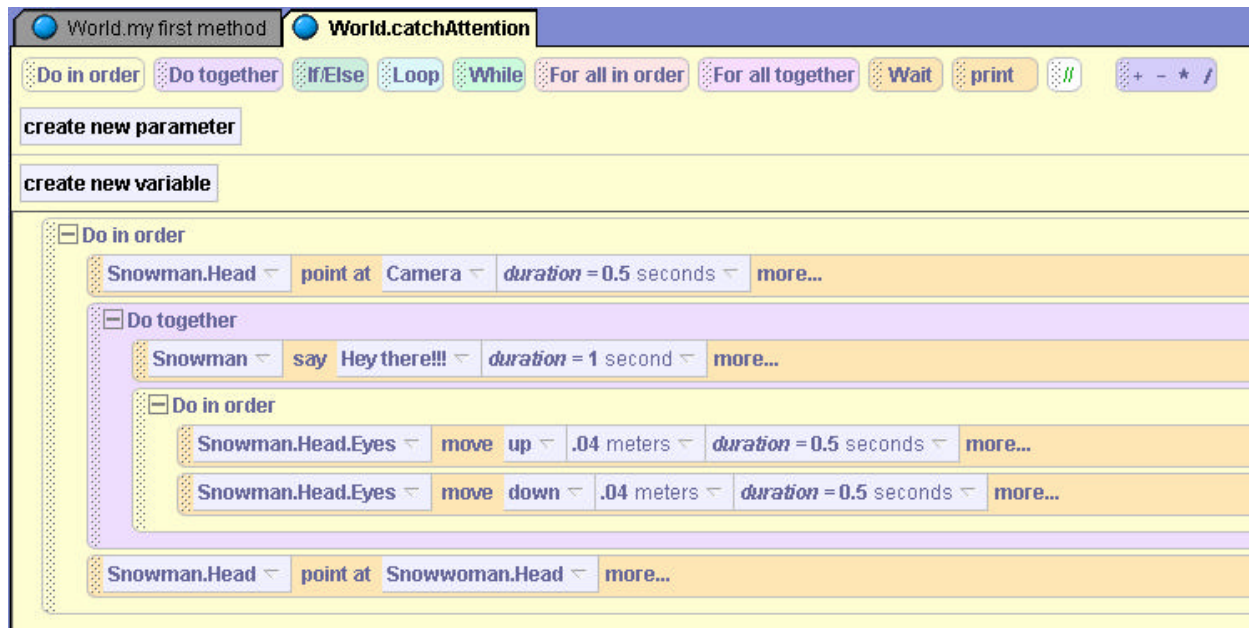


Figure 3-1-5. Defining the *catchAttention* method

Warning: If the **PLAY** button is clicked at this time, the animation will **NOT** run. This is because the *catchAttention* method has been defined but Alice has not been told to execute the method. That is, the method has not been *called* into action. Computer scientists would say the method has not been *invoked*.

Calling (Invoking) a Method

So, how do we call (invoke) our new method? Well, in the previous section of this chapter, we learned that when you (as the human “user”) click on the **Play** button, Alice automatically executes *World.my first method*. We can see why this happens by looking carefully at the Events editor, located in the top right of the Alice interface as seen in Figure 3-1-6. The instruction in this editor tells Alice *When the world starts, do World.my first method*. We didn't put this instruction here – the Alice interface is automatically programmed this way. So, when the user clicks on the Play button the world starts and *myFirstMethod* is invoked.

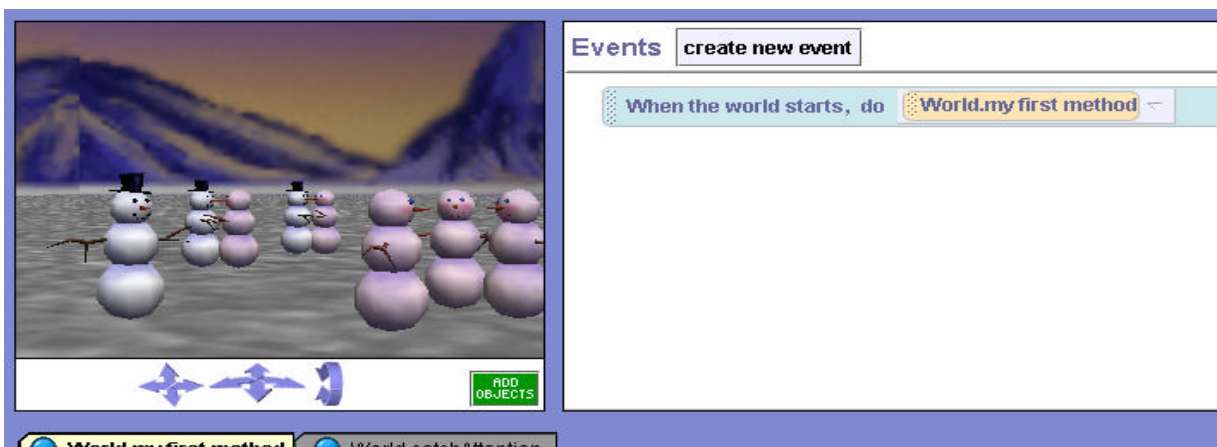


Figure 3-1-6. When the World Starts is linked to *World.my first method*

Let's take advantage of this arrangement. All we have to do is drag the *World.catchAttention* method from where it is listed in the methods tab of the details pane into *World.my first method*, as illustrated in Figure 3-1-7. Now, whenever the Play button is clicked, *my first method* will run and the *World.catchAttention* method will be invoked.

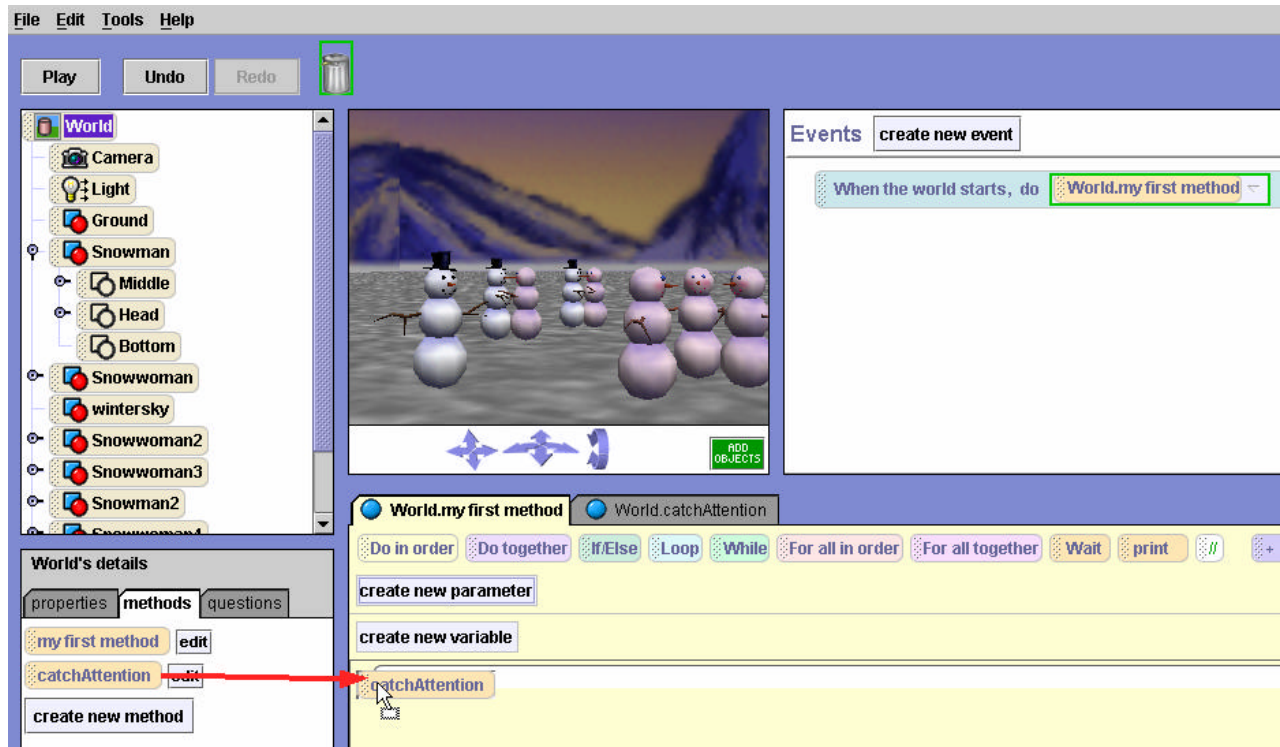


Figure 3-1-7. Dragging *catchAttention* into *my first method*

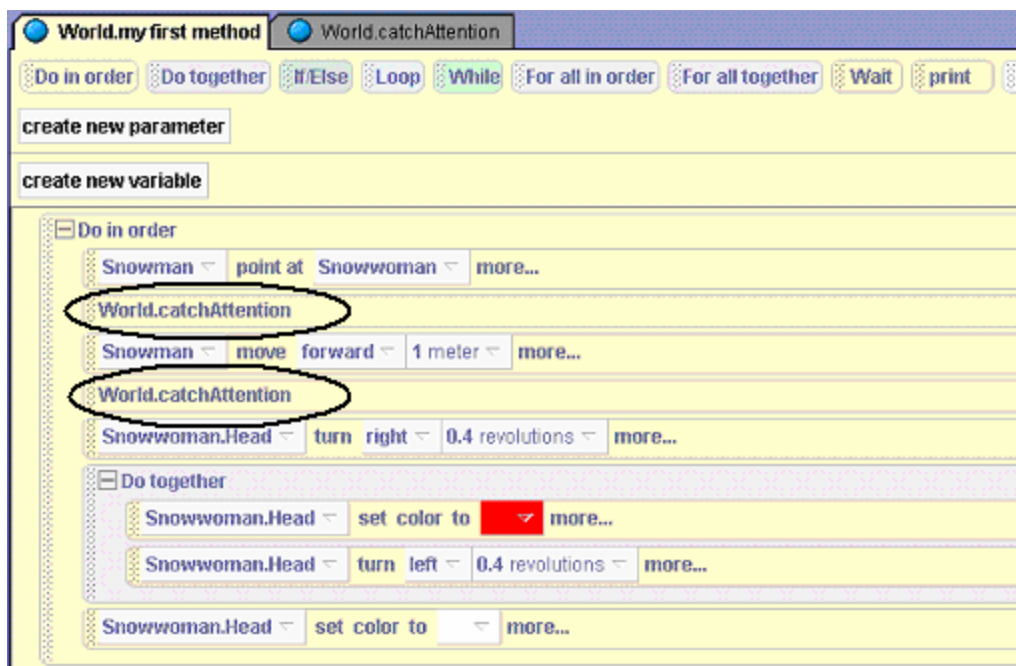


Figure 3-1-8. Calling *catchAttention* from *my first method*

Of course, we want other actions to take place as well. So, other instructions are added to *World.my first method*, as shown in Figure 3-1-8. This new program is somewhat different from the first program code written in Chapter 2. In the new program, the snowman tries to catch the snowwoman’s attention twice, moving closer to her (so she can hear him) after the first time.

Comments

Now that we have written our own method, it is time to look at a useful component to programs and methods – comments. Comments are not executable instructions. This means that Alice can ignore comments when running a program. However, comments are considered good programming “style” and are extremely useful for humans who are reading a program. Comments help the human reader to understand what a program does -- particularly helpful when someone else (for example, your instructor) has to read your program code.

Comments in Alice are created by dragging the green `//` tile into a program and then writing a description of what a sequence of code is intended to do. Figure 3-1-9 illustrates the program with comments added.

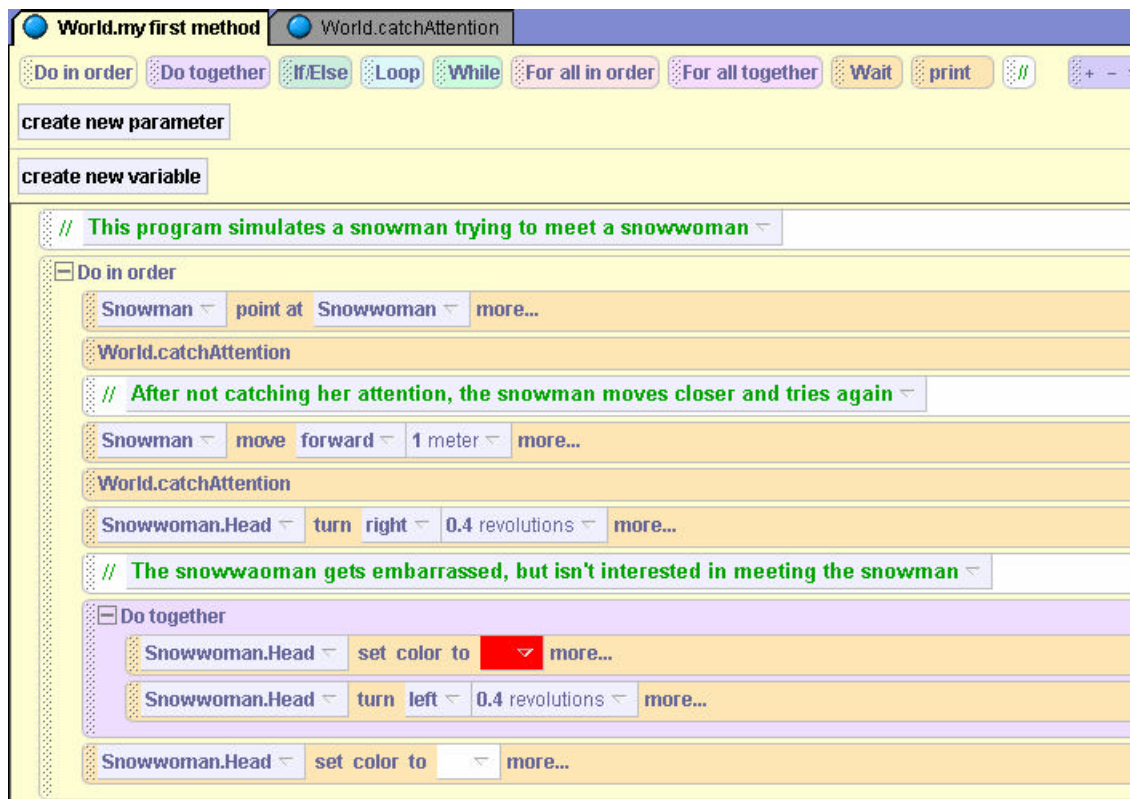


Figure 3-1-9. Comments in *World.my first method*

In methods where it is not obvious, a comment should be included at the beginning of a method to explain what the method does. In addition, small sections of several lines of code that collectively perform some action should be documented using a comment. The first comment in Figure 3-1-9 provides a general statement about what the program does: “...simulates a

snowman trying to meet a snowwoman.” The second comment describes the lines of code that animate the snowman trying to catch the snow woman’s attention and the third comment documents the snow woman’s responding actions.

Technical Note

The Events editor will be described in further detail in Chapter 4. But, it is worthwhile mentioning at this time that the Events editor can be used to modify the *When the world starts* instruction. Modifying this instruction allows you to call a new method rather than *World.my first method* when the user clicks on the **Play** button.

To modify the instruction in the Events editor, click on the image to the right of *World.my first method* in the Events editor. Then, select *catchAttention* from the drop down list that appears, as seen in Figure 3-1-10. Now, when the world starts, the *World.catchAttention* method will run instead of *World.my first method*.

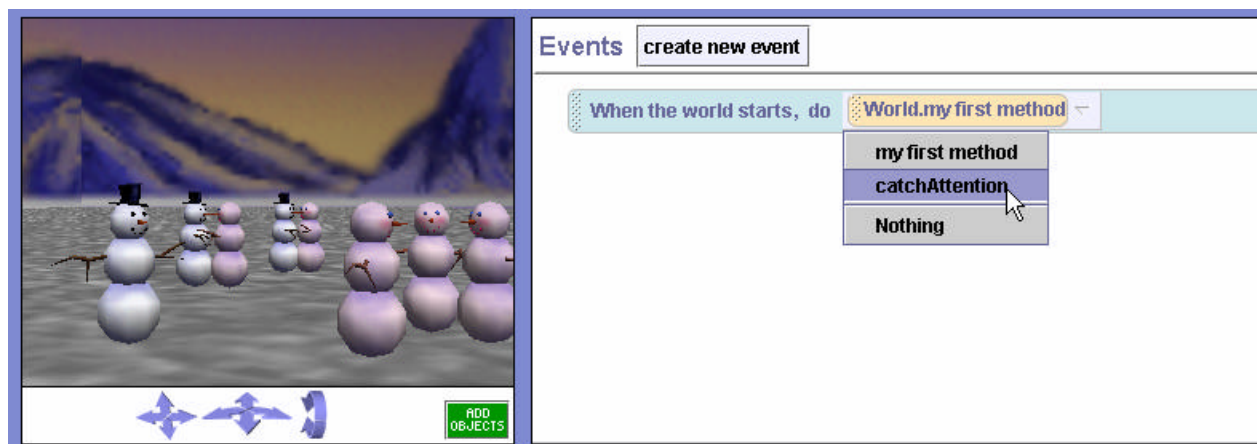


Figure 3-1-10. Modifying "When the world starts" event

3-1 Exercises

Reminder: Be sure to add comments to your methods that document what the method does and what actions are carried out by sections of code within the method.

1. Confused Kanga

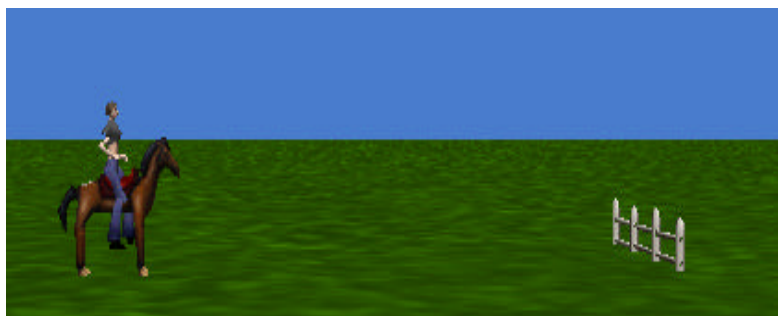
Scrounging for breakfast on the outback, Kanga encounters a rather confusing sign (Roads and Signs folder.) Kanga, stares at the sign for a few seconds and then hops left and turns toward the sign and then hops right and turns toward the sign and then left and then right....



Create a simulation that implements this comical story. Write methods *hopLeft* (Kanga turns left a small amount and hops, and then turns to face the sign) and *hopRight* (Kanga turns right a small amount and hops, then turns to face the sign). With each hop, Kanga should make some progress toward the sign. In *World.my first method*, alternately invoke the *hopLeft* and *hopRight* methods (twice) to make Kanga take a zig-zag path toward the sign.

2) Gallop and Jump.

Kelly has entered an equestrian show as an amateur jumper. She is somewhat nervous about the competition so she and the horse are practicing a jump. Create an initial scene with a horse and rider facing a fence, as shown below.



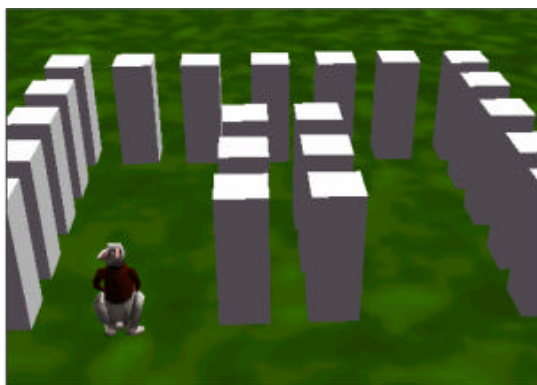
Write two world-level methods, one named *gallop* (horse and rider gallop forward one step) and another named *jump* (the horse and rider jump the fence.) In the gallop, the horse's front legs should lift and then go down as the back legs lift and the horse moves forward. Then the back legs should go back down. The jump should be similar but the horse should move up far enough to clear the fence in mid-stride. Test each method to be sure they work as expected. You will need to adjust the distance amounts to make each look somewhat realistic.

Hint: If you make the horse the vehicle for Kelly (see Tips & Techniques 1 for details on the vehicle property), you will only need to write an instruction to move the horse and Kelly will go along for the ride.

When you think the gallop and jump methods are both working properly, write instructions in *World.my first method* that call the *gallop* method as many times as needed to move the horse and rider up close to the fence and then call the *jump* method. Use trial and error to find out how many times the *gallop* method must be called to make the animation work well.

3. Rabbit Maze.

Create a world with the WhiteRabbit object (Animals folder) standing at the edge of a maze. The maze is created using blocks (Shapes folder.) Create walking methods for the rabbit to walk forward and backward. (Hint: Just reverse the forward walk for the backward walk.) Also create *TurnLeft* and *TurnRight* methods. In *my first method*, place these four methods in an order that will guide the rabbit successfully through the maze and then rewind through the maze (or just go through the maze backwards).



3-2 Parameters

It is clear from the example and exercises in the previous section that one program may be made up of several methods. Each method is its own small block of instructions, designed to perform a specific task when requested. It is possible, if not probable, that objects may be performing actions in more than one world-level method. We can appreciate that some communication might need to occur between methods. In this section, we look at *parameters*. Parameters are used for communication between methods. We arrange to transfer values (e.g., a number or a color) or names of objects from one method to another by using parameters in our methods.

Example

An example world will illustrate the creation and use of parameters. For a spring concert, our entertainment committee has hired a popular music group – the Bug Band. Our job is to create an animation to advertise the concert. In the animation, each band member wants to show off their musical skills in a short solo performance.



Figure 3-2-1. Bug Band on Concert Stage

Setting the Stage

Figure 3-2-1 shows the initial scene for the animation. The world is made up of a concert stage (Sets), queen bee, mantis, worker bee, ant (Animals), bass, saxophone, drums, and guitar (Musical Instruments). A spotlight will be used to highlight one band member at a time for their solo. To make the spotlight more dramatic, the light in the scene should be less bright. To dim the lights on stage, *Light* is selected from the Object tree and then the *brightness* property of the Light is changed from 1 to 0.5, as seen in Figure 3-2-2. A spotlight is added to the scene and a one-shot instruction is used to point the spotlight at the crowd. Figure 3-2-3 shows the dimmed scene with the spotlight shining on the crowd. (The spotlight is circled in Figure 3-2-3 for the purpose of showing the spotlight pointed at the crowd.)

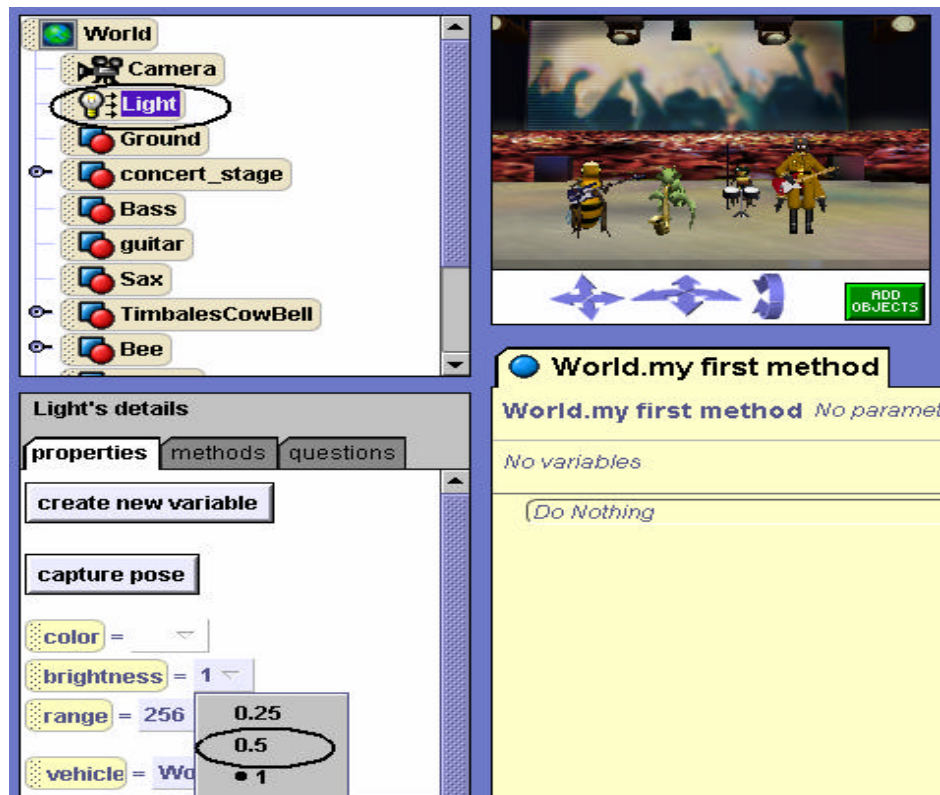


Figure 3-2-2. Dim the Light brightness

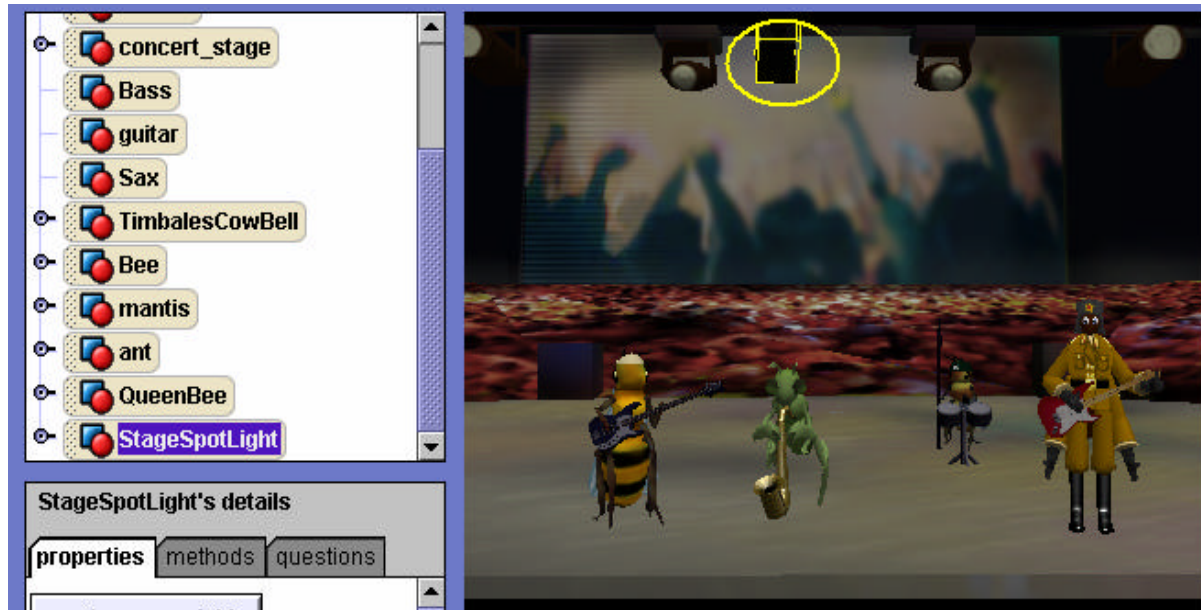


Figure 3-2-3. Dimmed Scene with Spotlight on Crowd

Storyboard with a Parameter

The storyline for this animation is that each band member will highlighted by a spotlight and will perform a short solo (play a musical instrument.) Let's create a storyboard for the spotlight focus

and movement of each band member during a solo. Because we have four band members, four textual storyboards are composed:


<i>Do in order</i> point spotlight at queen bee queen bee move and play instrument point spotlight at crowd	<i>Do in order</i> point spotlight at mantis mantis move and play instrument point spotlight at crowd
<i>Do in order</i> point spotlight at worker bee mantis move and play instrument point spotlight at crowd	<i>Do in order</i> point spotlight at ant ant move and play instrument point spotlight at crowd

Of course, we could write four methods – one for each solo storyboard. But, it is quite clear that the four storyboards are strikingly similar. The major difference is which band member will perform. This is where parameters come in. Let’s collapse the four storyboards into one storyboard and use a parameter to communicate which band member will perform the solo. The storyboard with a parameter is:

Parameter: bandMember
<i>Do in order</i> point spotlight at bandMember bandMember move and play point spotlight at crowd

The *bandMember* parameter name (an arbitrary name) is taking the place of the name of the specific object that will perform the solo. You can think of a parameter as acting like someone who stands in a cafeteria line for you until you arrive – sort of a *place-holder*. If a parameter is created, we can write just one method use the parameter to communicate which band member is to perform the solo when the method is called.

Creating an Object Parameter

To implement the animation, a new world-level method, named *solo*, is created. The editor creates a new method pane, as seen in Figure 3-2-4. A “create new parameter” tile automatically appears in the upper right of the method’s tabbed pane in the editor. When the create new parameter tile is clicked, a dialog box pops up as shown in Figure 3-2-5. The *name* of the parameter is entered and its *type* (Number, Boolean, Object, or Other) is selected. In this example, the name of the parameter is *bandMember* and its type is *Object*. When completed, the parameter name is in the upper left of the method pane, as shown in Figure 3-2-6. Note the 

type label that appears immediately before the parameter name. This means that any method that calls the *World.solo* method must supply a value that is the name of the object.



Figure 3-2-4. *World.solo* method pane

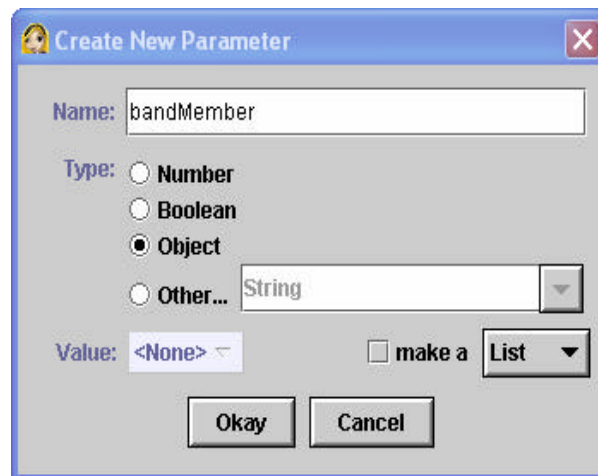


Figure 3-2-5. Name and select type for parameter declaration

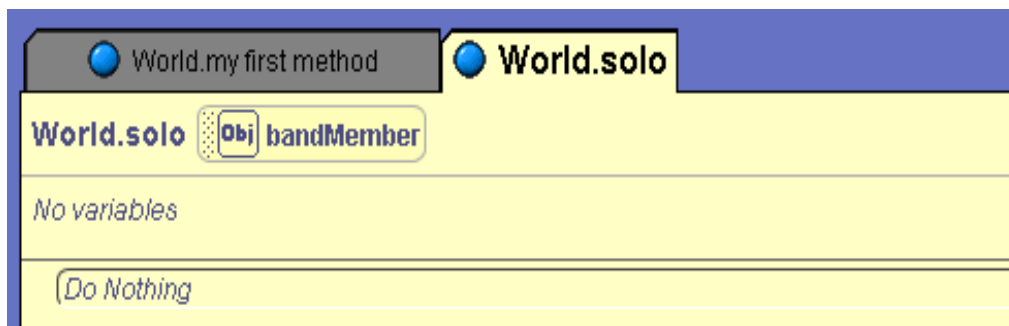


Figure3-2-6. Resulting parameter

Using an Object Parameter

In this example, two different ways to use an object parameter will be illustrated: (1) to specify a target of an action, and (2) to specify the object that performs an action.

(1) Parameter is a target

The first step in the storyboard above is “point spotlight at bandMember.” The *SpotLight* is selected in the Object tree and its *point at* method is dragged into the editor. The target of the spotlight is selected from the cascading menu, as shown in Figure 3-2-7. The list of possible targets for a *point at* instruction is composed of all the objects in the Object tree. But,

bandMember is a parameter, not an object in the Object tree. Instead, *bandMember* is considered an *expression* and is listed at the bottom of the cascading menu. The resulting code, illustrated in Figure 3-2-8, shows the object parameter *bandMember* used as a target for the point at instruction.

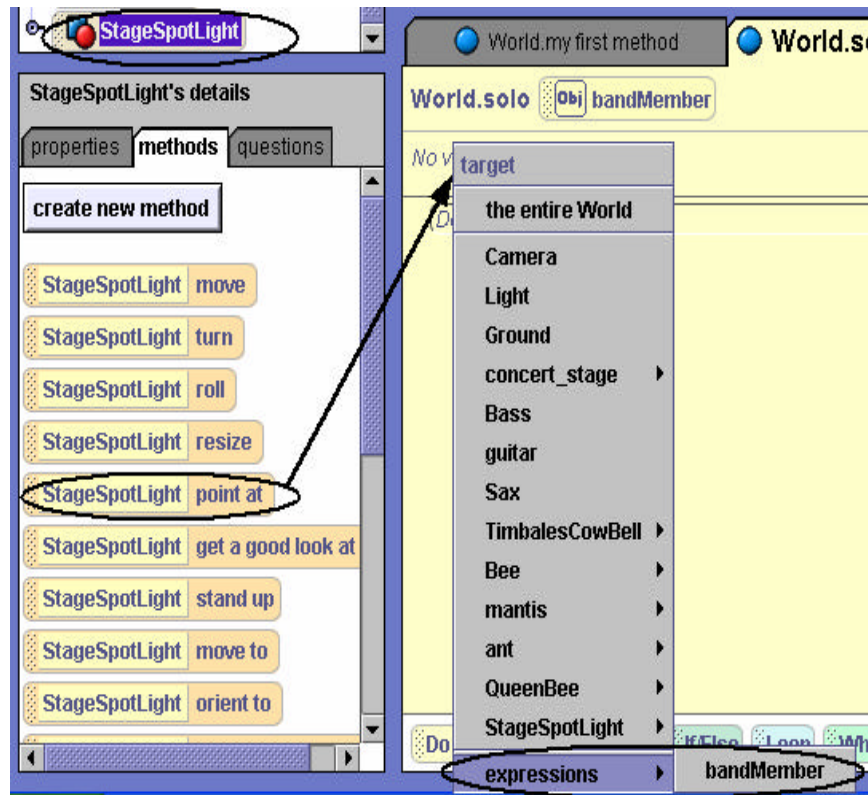


Figure 3-2-7. Select parameter as target



Figure 3-2-8. Resulting point at instruction

(2) Parameter performs an action

The second step in the storyboard is “bandMember move and play.” Clearly, in this action, the *bandMember* will perform an action, not be the target of an action. This is the second way a parameter can be used in a method. Intuitively, we look at the Object tree to find *bandMember* so its *move* instruction can be dragged into the editor. But, of course, *bandMember* is not in the Object tree. (See Figure 3-2-9) This makes sense because, as mentioned earlier, *bandMember* is not an object – it is acting as a place-holder for an object.

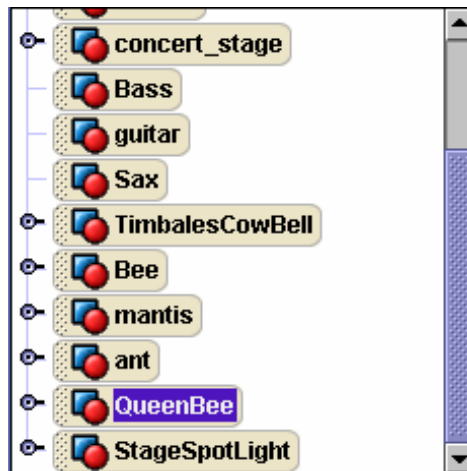


Figure 3-2-9. *bandMember* is not in Object tree

Because *bandMember* is a parameter and not an object, two steps are needed to create instructions where an object parameter is to perform an action. The first step is to select one of the objects (Bee, mantis, ant, or QueenBee) and create instructions for that object. Let's use the QueenBee (an arbitrary choice) for move instructions, as shown in Figure 3-2-10.

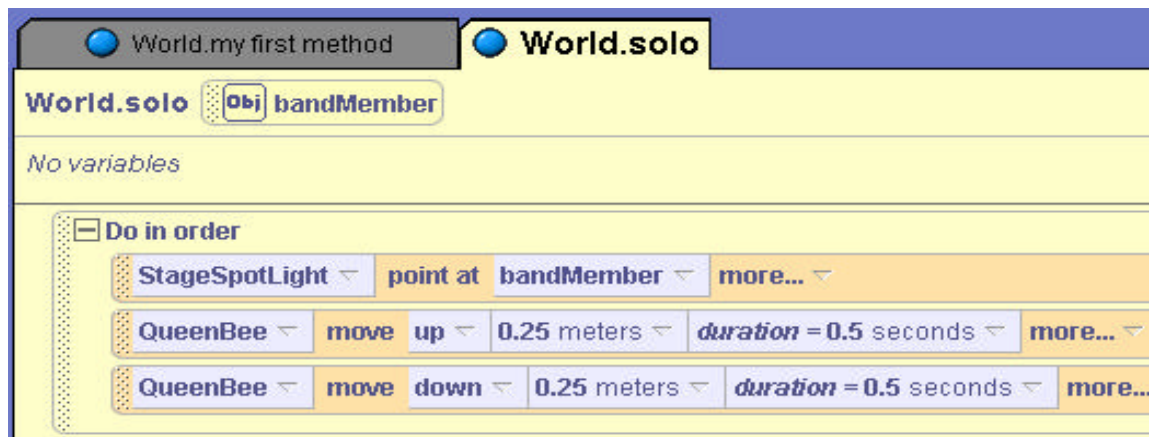


Figure 3-2-10. Instructions for an arbitrary object

The second step is to modify the move instructions to allow the parameter to act as a placeholder for the object. The *bandMember* parameter is dragged toward a move instruction as illustrated in Figure 3-2-11. In this example, *bandMember* will be used to take the place of *QueenBee* in both *move* instructions.

Importance of Parameter Type

We wish to draw your attention to the appearance of the instructions in Figure 3-2-11. When a parameter tile is dragged over instructions, all the tile locations where the parameter tile can be dropped are immediately enclosed by a green border. The color-change is a visual hint to allow you to select an appropriate location where the parameter can be used. In this example, *bandMember* is to be used as a place-holder for an **object** in a move instruction. But,

bandMember should not be used as a placeholder for distance (0.25 meters) or duration (0.5 seconds) because these are **number** values, not objects.

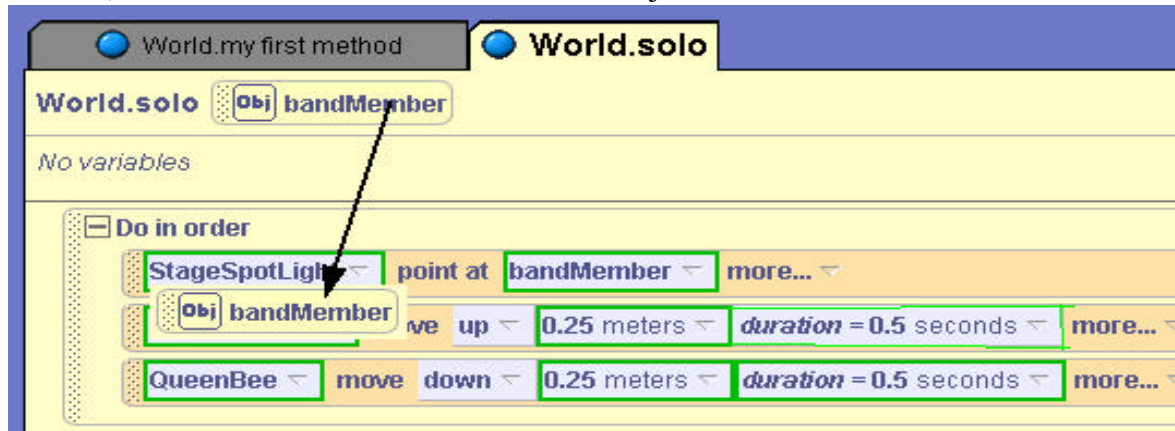


Figure 3-2-11. Drag parameter to object position

The last step in the storyboard is to turn the spotlight back on the crowd (that is, away from the band member who just performed a solo). A second spotlight *point at* instruction is added at the end of the *World.solo* method. In this instruction, *concert_stage.crowd* is the target for the *point at* instruction. The resulting code is shown in Figure 3-2-12.

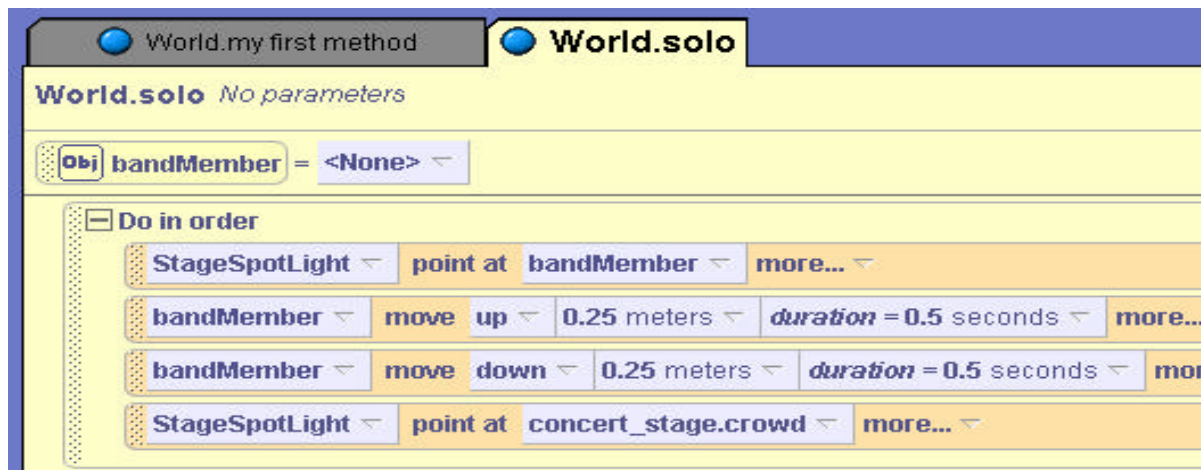


Figure 3-2-12. Resulting solo code

Test with Arguments

Let's pause to save and test the code. To test the *World.solo* method, the *solo* method is called from *my first method*. When *solo* is dragged into *my first method*, cascading menus (Figure 3-2-13) allow the selection of an object that *bandMember* will represent for that call of the method. To be certain the *solo* method works for each Bug Band musician, four statements are written, as seen in Figure 3-2-14. In the first call, the solo method will be performed with *bandMember* representing QueenBee, in the second call *bandMember* will represent mantis, in the third the worker Bee, and in the fourth the ant. In Computer Science terminology, the value communicated to a method when called is known as an *argument*. In this example, the QueenBee, mantis, worker Bee, and ant are each used as an argument in a call to the solo

method. In this way, the solo method is somewhat *generic* – it can be used with different arguments to carry out the same task with different objects.

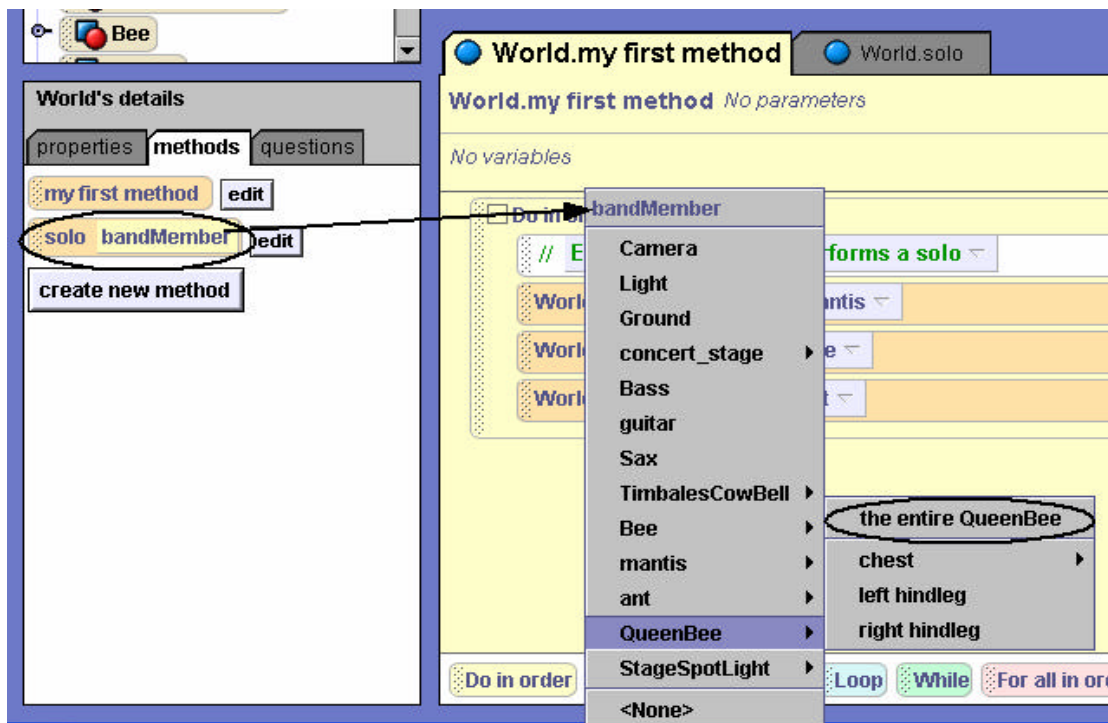


Figure 3-2-13. Selecting an object for bandMember



Figure 3-2-14. Calling *World.solo* with different arguments

Multiple Parameters

You may have noticed that the above code does not yet complete the animation. In each solo, the band member should not only move but should also play a musical instrument. (If your computer does not have a sound card, the “say” instruction can be used to display song lyrics instead of

play a sound.) An instruction is needed in the solo method to play a sound. But, each musical instrument should have a different sound. This means that the sound of the bass should be played for the QueenBee, a saxophone for the mantis, drums for the worker Bee, and guitar for the ant's performance. Evidently, more than one parameter would be useful in this situation. To illustrate the use of multiple parameters, a second parameter can be added to the solo method. The type of this parameter will be Sound, as shown in Figure 3-2-15. The necessary sound files are then imported (see Tips & Techniques 2 for details on importing and using sound files). As with object parameters used to perform an action, an instruction to play a specific sound (*bassMix*) is dragged into the editor, as illustrated in Figure 3-2-16.

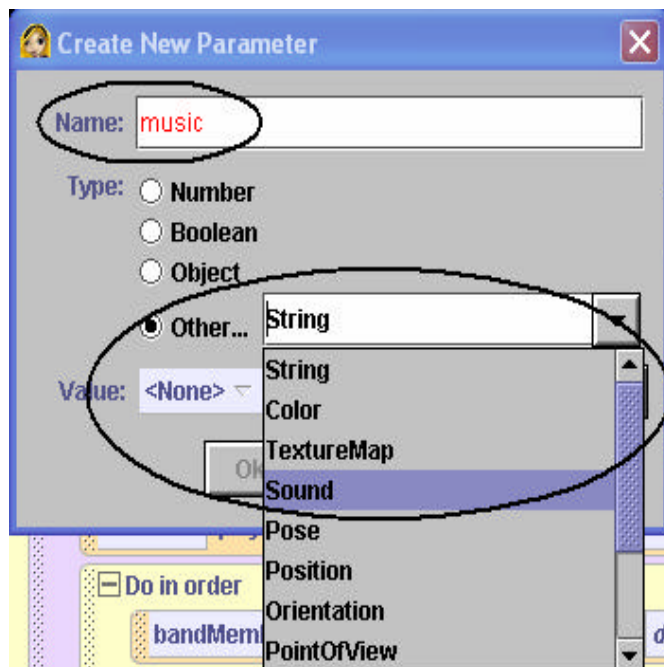


Figure 3-2-15. Sound parameter

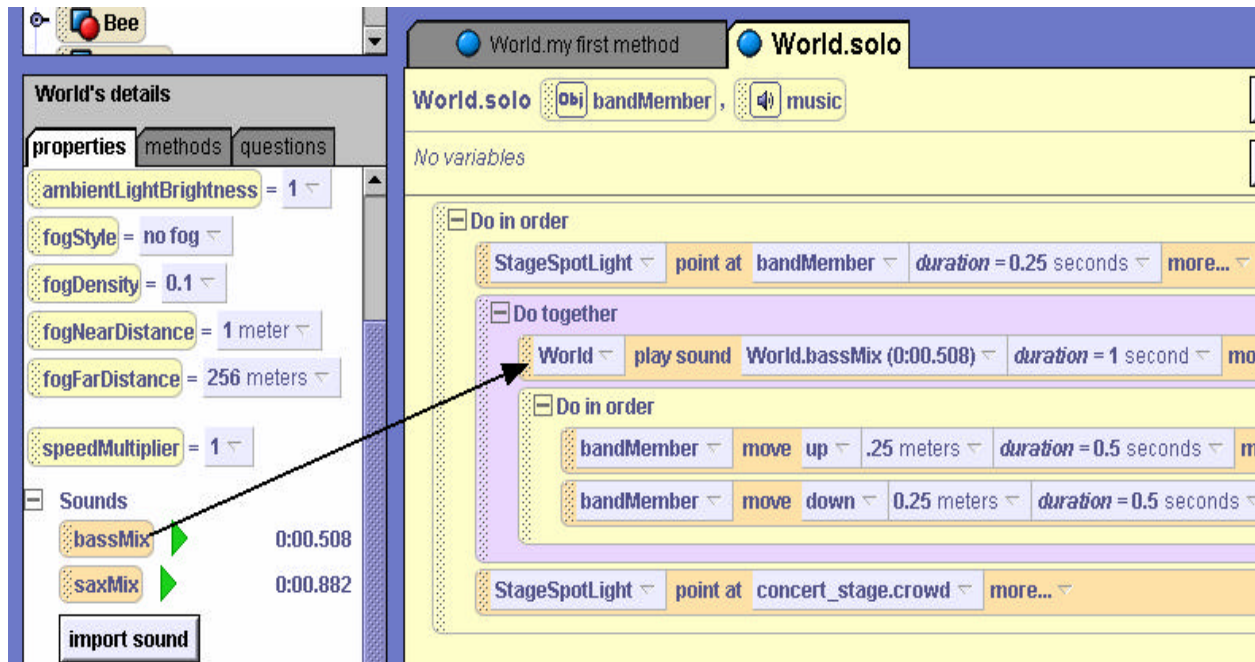


Figure 3-2-16. Drag sound instruction into method

Finally, the *music* parameter is used to act as a place-holder for a sound. The completed solo method is shown in Figure 3-2-17. Calls to the solo method are revised in *my first method* to pass in two arguments, an object and a sound. The revised code is shown in Figure 3-2-18.

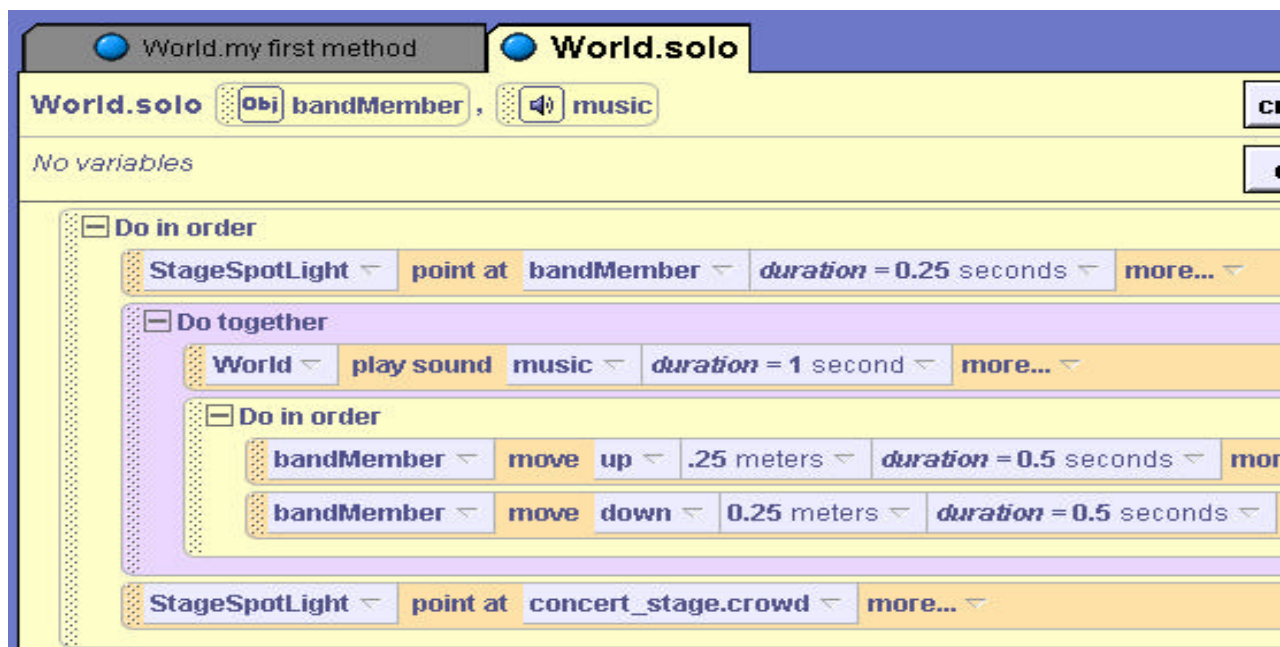


Figure 3-2-17. Completed *solo* method

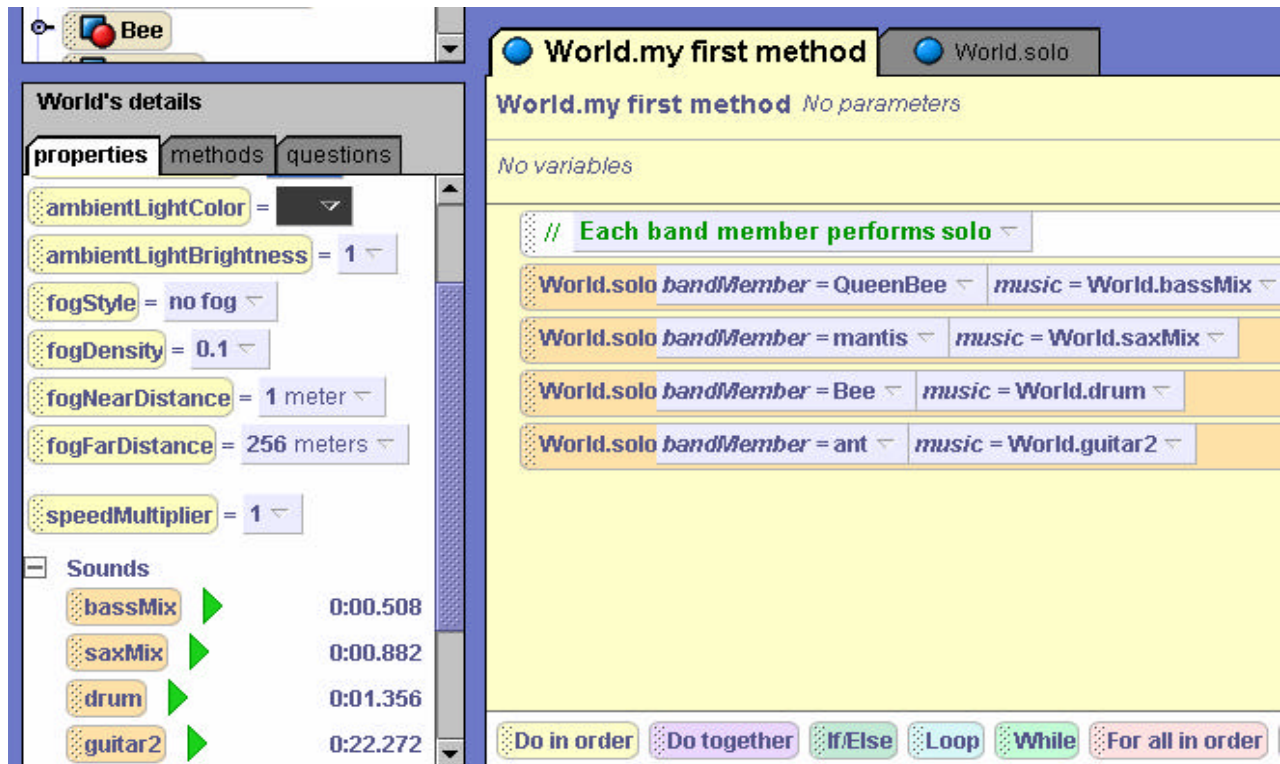


Figure 3-2-18. Completed *my first method*

As mentioned above, this animation can be completed without the use of sound. An alternate version of the solo method is shown in Figure 3-2-19. A String parameter, songLyric, is used instead of a Sound parameter. The string is displayed using a *say* instruction.

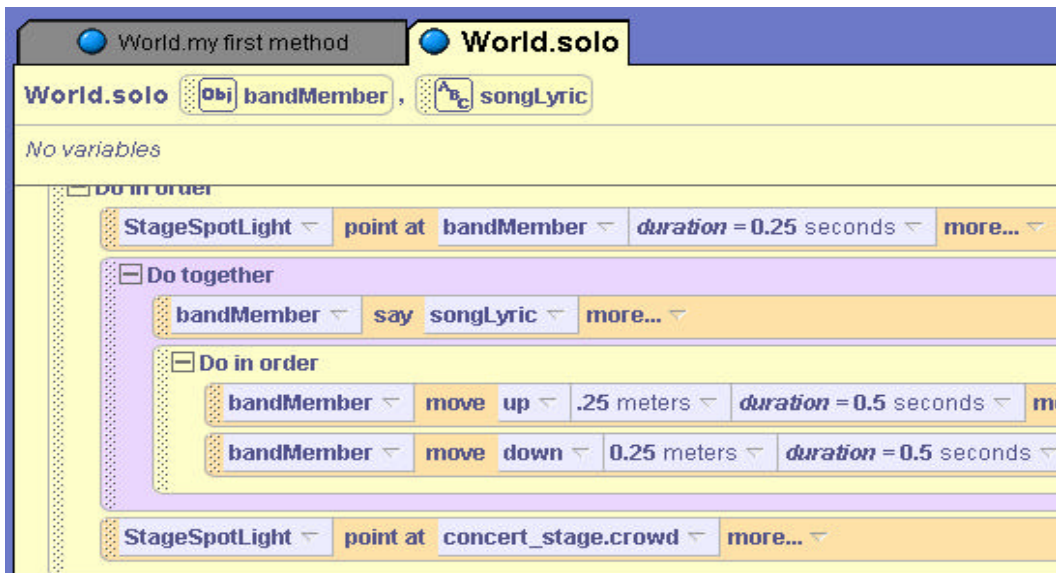


Figure 3-2-19. Lyric version of *solo*

Other types of parameters

The Bug Band example illustrated two kinds of parameters, objects and sound. A parameter can also be a number, a Boolean (*true* or *false*) value, a color (red, blue, green, etc.), or any of several other types. (See Figures 3-2-5 and 3-2-15 for examples of parameter type selection.) Each of these types of values contributes to a rich environment for programming. Number values play an important role in many programming languages. So, before leaving our discussion of parameters, let's take a look at a simple world in which a number parameter is illustrated.



Figure 3-2-19. Magic Act Initial Scene

Example

A magician is performing a levitation illusion. In a levitation illusion, objects seem to rise magically into the air. The magician points a magic wand at his assistant and she gently rises into the air and then floats back down to her original position on the table. Then, the magician performs the same trick with the rabbit. The rabbit, being a lighter object, will float into the air higher than the magician's assistant. The initial scene is illustrated in Figure 3-2-19.

Because the magician's assistant and rabbit are each to levitate in the same way, the animation can be implemented as a single method if we use a parameter to communicate which object is to float. The height the object floats could also be passed as an argument to the method. So two parameters are needed, one an object (*floatingObject*) and the second a number (*height*). A possible storyboard is shown below.

<p>Parameters: floatingObject, height</p> <p><i>Do in order</i></p> <p>Magician right arm point at floatingObject</p> <p>floatingObject move upward height meters</p> <p>floatingObject move downward height me ters</p>

Number parameter

An object parameter named *floatingObject* and a number parameter named *height* are created and then lines of code are written to float the object up and back down the given height. The resulting method is shown in Figure 3-2-20.

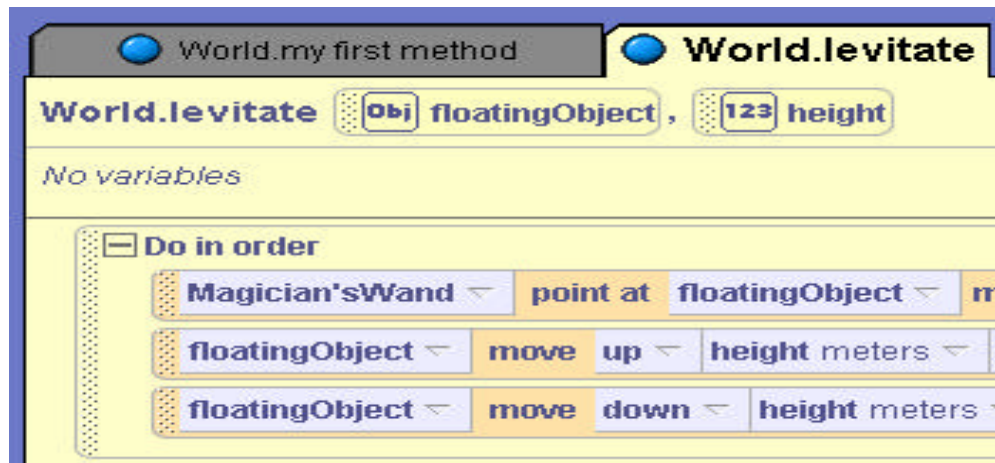


Figure 3-2-20. Object and Number parameters

Calls to the levitate method are written in my first method, as illustrated in 3-2-21 and then the program is run to test the method.

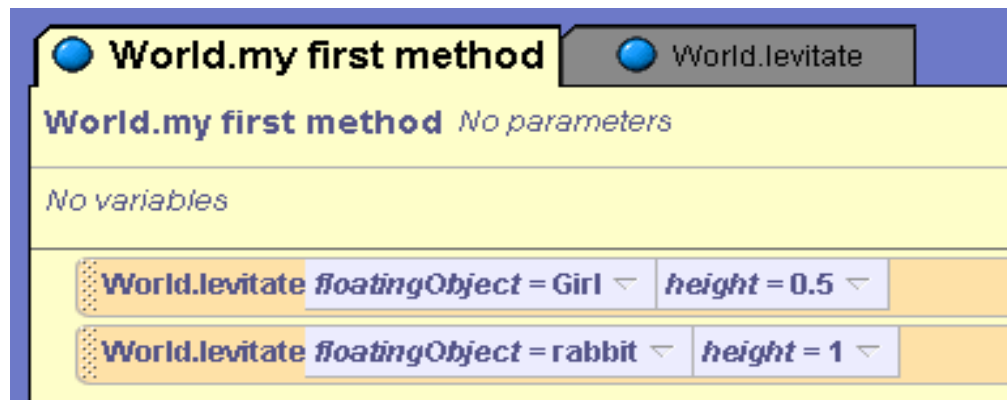


Figure 3-2-21. Calling levitate method

Technical Note

When this animation is run, the result is rather surprising. The rabbit floats up and down just as expected. But, the magician's assistant moves horizontally instead of floating upward in the air! Of course, the magician's assistant is reclining on the table and so her "up" (see Figure 3-2-22) is not exactly what we had in mind. To solve this problem of orientation, the *move* instructions in the *levitate* method are revised to use the *as seen by* option. As seen by allows us to select a direction based on the perspective of a different object. In this example, we selected the Ground as a reference, as shown in Figure 3-2-23. The revised code is presented in Figure 3-2-24. Now the objects will move upward with reference to the Ground. (See Tips & Techniques 2 for more detail on *asSeenBy*.)



Figure 3-2-22. Up –assistant’s orientation

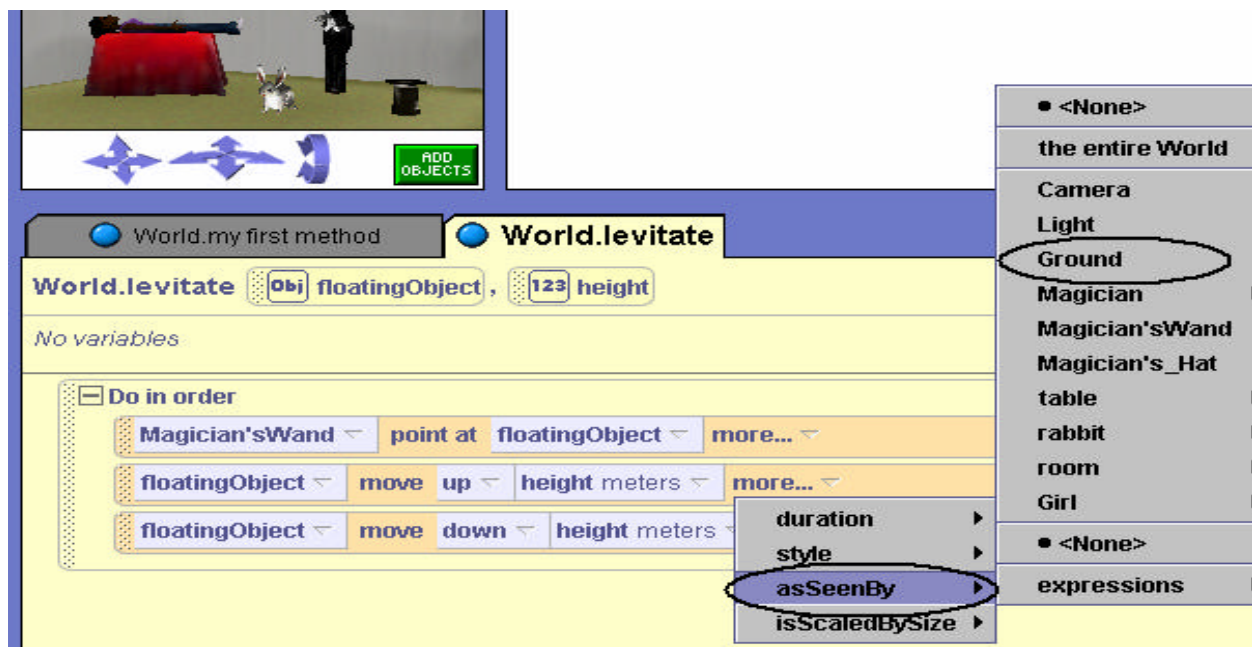
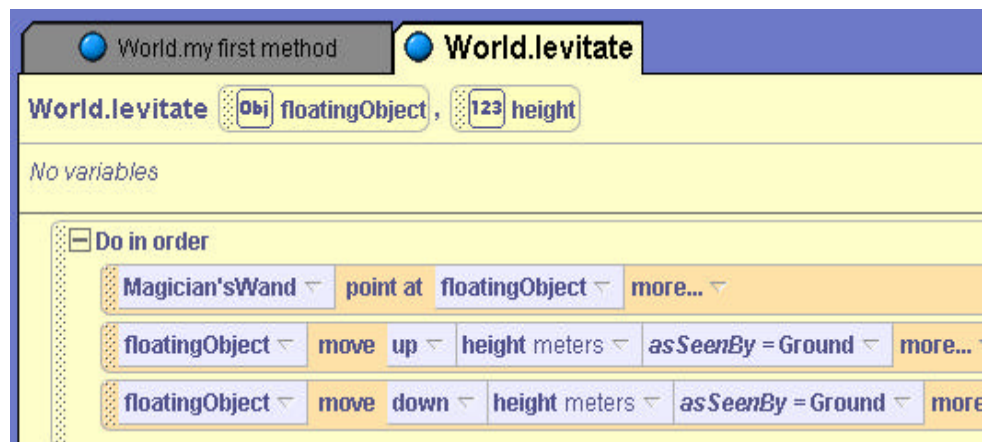


Figure 3-2-23. Selecting up asSeenBy Ground



3-2-24. Revised Levitate

3-2 Exercises

1. FrogEscape

At the local lily pond, the frogs enjoy climbing out of the water now and then to warm up in the sun. Of course, the frogs get a bit jumpy when a predator is sighted. On this fine day, a hungry snake wanders into the scene. Create a world scene similar to the one below and animate the frogs jumping into the pond when the snake approaches. Write a method that points the snake at a frog and slides the snake toward the frog. Then, have the frog turn to the pond and jump in. Your method should use a parameter to specify which frog is escaping.



2. FunnyRace

This is a funny kind of race because you will know who is going to win before the race begins. (But, this is a good exercise to gain some experience in race simulations as a preparation for more complicated worlds later in the book.) Create a world with three characters lined up as if they were at the starting line of a race. Add another object to the world that can serve as a start and finish line. (In the scene below, we used the solorail from the Skate Park collection.) Create a method that simultaneously makes each character move forward to pass the finish line. Use three parameters (one for each character) that specify the amount of time (duration) for each of the three characters to complete the race.



3 Dragons

Legend has it that dragons are distant relatives of chickens. So, we are not surprised that a favorite pastime of dragons was a game of “chicken.” The scene below shows a world with four dragons, carefully placed in a diamond-like pattern so any dragon is an equal distance from each of the other dragons (similar to baseball players at the four bases of a baseball field). Create a simulation of a game of chicken where any two dragons face each other and fly upward to a slightly different height above the ground. Then, the dragons fly towards each other, nearly missing one another. The dragons should each land in the position where the other dragon was located. That is, the two dragons trade places. Your simulation should use a method named *dragonFlight* that has four parameters – the two dragons that will face off in a game of chicken and the heights for each of the dragon’s flight.



4 WheelMotion

This exercise is to allow you to work with motor vehicles that have wheels. The challenge here is that the motion of the wheels (rotation) is different than the overall motion of the motor vehicle. The wheels must rotate while the car itself moves forward. To gain some appreciation for this kind of animation, create a simple world with a car and a dump truck. Create 2 methods *wheelRoll* and *vehicleMove*. *wheelRoll* should receive a parameter that is one wheel and turn it one revolution forward. The *vehicleMove* method should receive a parameter that is a motor vehicle and make it move forward while the wheels roll (by calling *wheelRoll* for each wheel of the vehicle).



3 Summary

In this chapter, we looked at how to write our own methods and how to use parameters to communicate between methods. In particular, we focused on world-level methods that involve two or more objects interacting some way. An advantage of using methods is that the programmer can think about a collection of actions as if it was just one action— *abstraction*. Also, methods make it easier to debug our code. Comments were used to document methods, where the purpose of the method or a segment of a method is not immediately obvious. Comments are considered good programming “style.”

We used parameters to arrange a transfer of values from one method to another. In a method, a parameter acts as a place-holder for a value of a particular type. Examples presented in this chapter included object, sound, string, and numeric parameters. Parameters allow methods to be written in a generic manner. The method can be called with different arguments to carry out the same task with different values.

Important concepts in this chapter

- A method is a well-designed collection of instructions that will be carried out when requested.
- Methods that define behaviors for an object acting alone are considered object or *character-level*.
- Methods that involve actions by more than one object have a more global perspective and are considered *world-level* methods.
- To run (or execute) a method, the method must be called (invoked).
- Parameters are used for communication between methods.
- A parameter can be declared to represent a value of a particular type.
- Types of values for parameters include object, Boolean, number, sound, color, string, and others.
- In a call to a method, a value sent in to a method parameter is an *argument*.

3 Projects

Projects are more challenging than exercises in a chapter. The projects in this chapter involve motion of human body parts. Professional animators spend many, many hours mastering the art of making these movements look realistic. But, our focus is on mastering the art (and science) of writing methods in a program.

1. Dance

Note: To assist you in learning how to animate human body parts, this first project includes some explanations and coding suggestions. The goal of this animation is to have the couple perform a dance step in a traditional box (square) figure as used in a waltz and other dances. Create a scene with a man and a woman character (People) inside a saloon (Old West), as illustrated below.



In the first step of a box figure, the man takes a step forward, leading with his left leg and (at the same time) the woman takes a step backwards, leading with her right leg. This is not as simple as it sounds. One way to make a character with legs appear to take a “step” is to have the character raise one leg some small amount and then move forward as the leg moves back down. Then, the other leg performs a similar action. Thus, to make two characters appear to dance with each other requires a coordinated leg lift, move and leg drop action for both characters. A potential storyboard for the first step of the box dance is illustrated below.

Parameters: howFar, speed

Do in order

1) *Do together*

Man's left leg move up

Woman's right leg move up

2) *Do together*

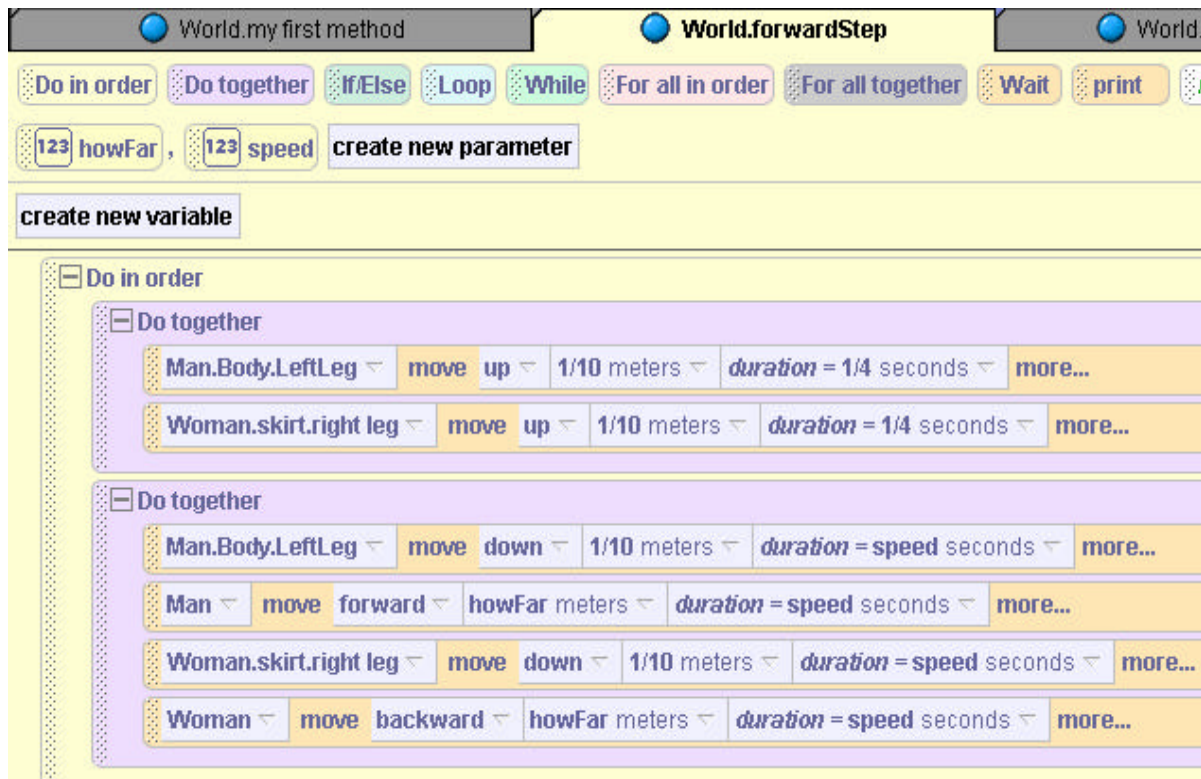
Man move forward howFar

Man's left leg move down

Woman move backward howFar

Woman's right leg move down

Write a method named *forwardStep*. The *forwardStep* method has two parameters -- *howFar* which specifies the distance of the move forward, and *speed* which specifies the duration for the forward motion. An example of a *forwardStep* method is shown below.



After the *forwardStep*, three steps remain: (1) *rightStep*, where the man and woman take a step sideways (his right, her left), (2) *backStep*, where the man takes a step backward, leading with his left leg and the woman takes a step forwards, leading with her right leg, and (3) *leftStep*, where the couple takes a step sideways (his left, her right). You will need to experiment with the amount to move the legs up (and down), and with the size of step forward, backward, and sideways the couple is to take.

Then, create a method named *spin*, which has the man spin the woman around. The code for a spin method might appear as follows:



If are four steps are properly performed in sequence, the couple moves in a square-like pattern on the dance floor. Create a method to call all the methods in order so as to animate the couple performing a box figure followed by a spin for a dance. Then, create a second method to animate the performance of a different dance figure – calling the dance steps in a different order.

2. HandBall

Create a world with a RightHand holding a ball. Have the fingers close to grasp the ball. Then, throw the ball into the air while opening the fingers of the hand. Finally, make the hand catch the ball as the hand re-closes its fingers. (*Hint*: Consult Tips & Techniques 1 for how to use the *vehicle* property. The vehicle property will help in making the ball move as the hand moves.)



3. RaRowYourBoat

Create a world with a boat, a person sitting in the boat, an island, and a pier located 25 meters away from the island. In the world shown below, Ra (Egypt) is sitting in a rowboat (Vehicles). (Note: this world takes some time to set up because getting a person object to sit down requires patient rearrangement of the body parts in the scene editor. If you wish, you can use the world provided on the CD.) Create a method to make the Ra row the boat 25 meters from an island to a pier. One suggested way to do this would be to create the methods: *RowLeft* and *RowRight* (to control the arms motions), *ControlTorsoAndHead* (to control back and head motions), and *StartRow* and *StopRow* (to put Ra's body in and out of the rowing position).



4. RearrangingRobot

Alice received a gorilla robot (Animals) as a gift. She is trying to figure out how to program the robot to help her pick up things in her room (a chore she has to do every Saturday morning). In the initial scene for this animation, the robot is standing in the middle of Alice's room near several objects scattered around the floor (for example, a barbell, a pinata, and a teddy bear). The robot should be located between all the objects so it can easily be turned during the animation – not moving away from any of the objects.



Write a program to teach the robot to pick up one object at a time and rearrange it. Write two methods, named *pickUp* and *putDown*. Each of these methods should have one parameter identifying the name of the object to be picked up or put down. The *pickUp* method should make the robot pick up an object in its hand. The *putDown* method should have the robot put the object down somewhere other than where it started. (Hint: have the robot turn 1 / 4 left or right before putting the object down.) When these methods have been written, then write instructions in *myFirstMethod* to make the robot pick up each object, and place it somewhere else.

Tips & Techniques 3

Renaming an Object

In some animations several objects of the same type of character may be in the scene at the same time. Consider the scene in Figure T-3-1. Four chickens have been added to the world. Alice automatically names the chicken objects as Chicken, Chicken2, Chicken3, and Chicken4. Names serve to identify an object as being different from other objects in the world. If the names of particular objects do not suit your purpose, the names can be changed.

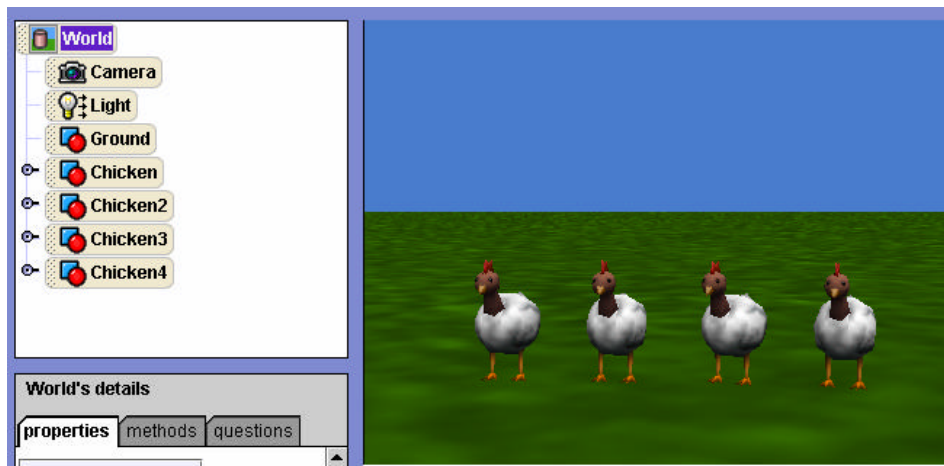


Figure T-3-1. A scene with multiple objects of the same model

To rename an object, right-click on the object name in the object tree, and select the rename option from the popup menu, as shown in Figure T-3-2. Then, enter the new name in the dialog box that pops up. Be sure to press the enter key after typing the new name.



Figure T-3-2. Renaming an object

Color Property as Identification

After several operations to arrange objects in the world, you likely will not remember which chicken is which – regardless of what the chickens are named! The simple problem is all four

chickens look identical. For example, just by looking at the chickens in the scene, it is impossible to determine which one is Chicken2.

The solution is to change the appearance of each chicken in some way, so that each chicken can easily be identified. One possible change is the color property of each chicken's Upper Lip. (Do chickens have lips?) This is a quick and easy change -- click the chicken object in the Object-tree and successively work your way down the tree to eventually select the Upper Lip, as seen in Figure T-3-3.

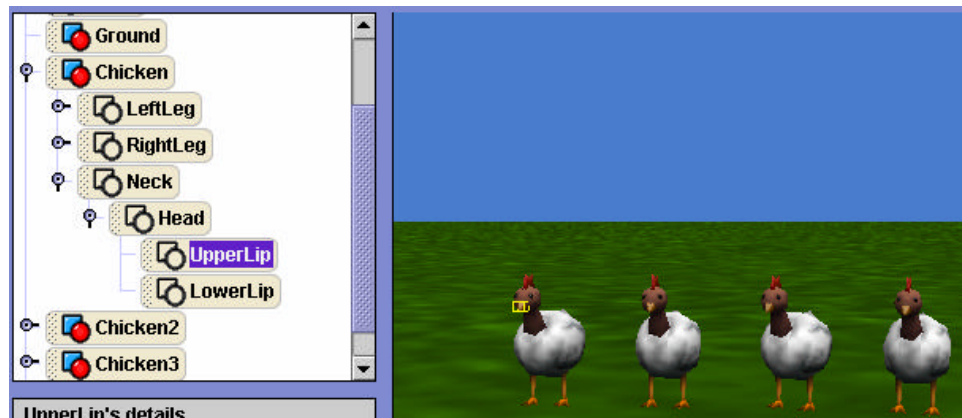


Figure T-3-3. Chicken's UpperLip

Then, change the color of the chicken's Upper Lip to a different color. This is done by right-clicking on the name of the object in the object tree, selecting *one shot*. When *one shot* is selected, cascading menus popup one at a time. In Figure T-3-4, the menu selections are one-shot → property animations → Chicken.Neck.Head.UpperLip set color to → value (where the value is a color).

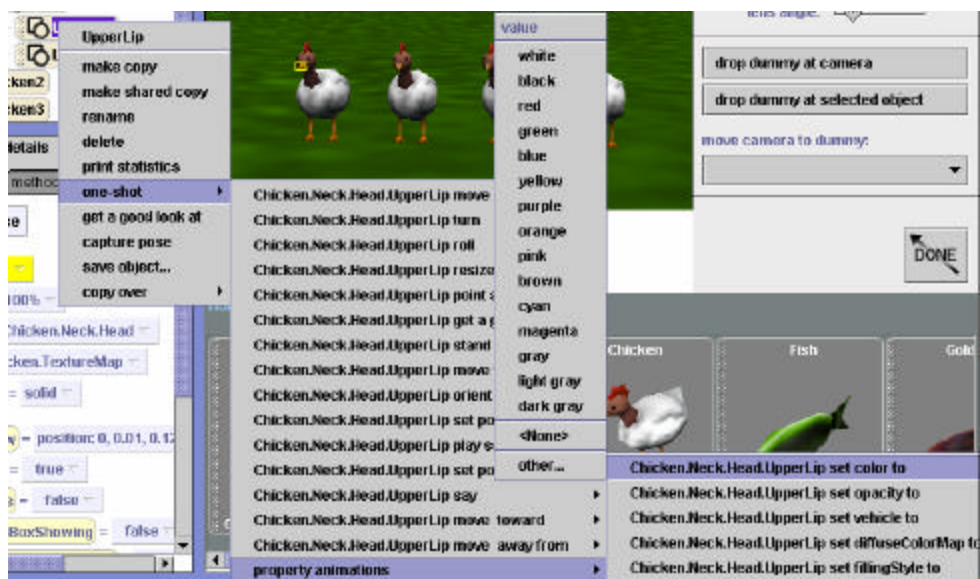


Figure T-3-4. Setting the color of chicken's UpperLip

If all four chickens have a different color upperLip, the chickens will now be easily identifiable, as seen in Figure T-3-5.



Figure T-3-5. Identifiable objects

This identification technique will be quite useful in future worlds. If we wish to know that something happened during the course of running an animation, we can just change some property of that object, such as its color!

Printing your program code

It is often useful to print the code from one of your methods. (Some instructors wish to have students hand in a printed copy of their program code.) To print a method, click on File, and then select Print. Select the name of the method to be printed, as shown in Figure T-3-6.

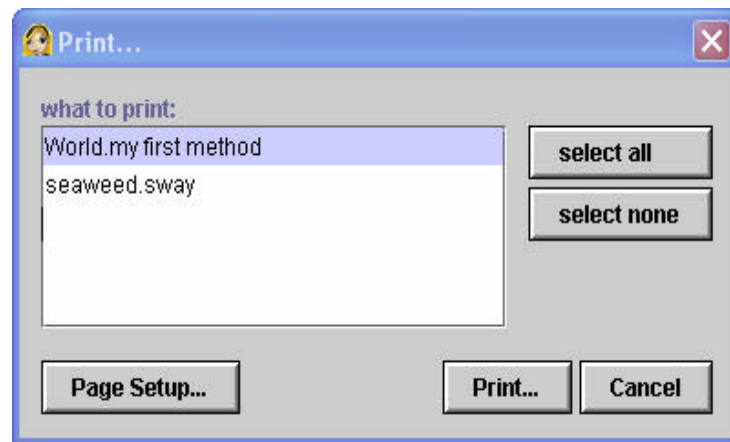


Figure T-3-6. Printing a method

Orient to

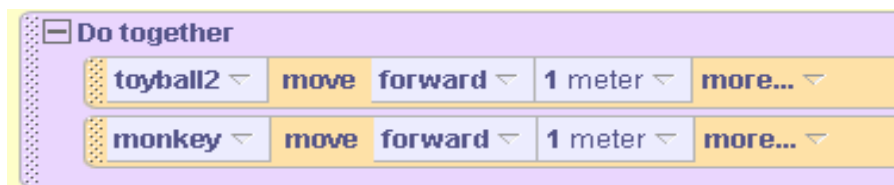
Surprising results can occur when two objects must move together but have different orientations. One technique is to make one object the vehicle of another. But, sometimes we do not want to make one object be the vehicle of another. In the world shown in Figure T-3-7, the

monkey runs and jumps on the ball. Then, both the monkey and the ball are supposed to roll forward together.

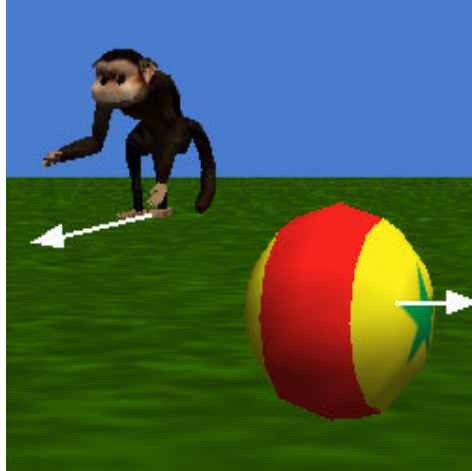


T-3-7. Monkey jumps on the ball

We wanted the monkey to stay on top of the ball, moving along with it for a short distance and then jump off and run away. The code we wrote to make the monkey and ball move forward together was:



Imagine our surprise when the ball rolled in one direction and the monkey moved in a different direction, ending up well away from the ball, sort of suspended in midair. (No doubt, the monkey was a bit startled, as well!)



T-3-8. Monkey and ball move forward in different directions

Why did this happen? Well, the ball is an example of an object for which we can't tell (just by looking at it) which direction is forward and which direction is backward. Evidently, in moving the ball and the monkey around in the scene, we positioned the ball so its forward direction was not the same as the forward direction for the monkey.

The way to solve this problem is to synchronize the orientation of the two objects when the initial world is being created. First add the objects to the world scene. Then use a one-shot *orient to (0,0,0)I* instruction on each object. The *orient to (0,0,0)I* instruction may seem a bit weird, but it simply tells Alice that the object should take on the same sense of the center of the world, as viewed by the camera. So, if we orient two objects to same orientation as the world, then the two objects are synchronized – they have the same sense of up, down, left, right, forward, and backward.

The expanded menu selection for *orient to* is illustrated in Figure T-3-9. This shows the monkey being oriented to the world center. Of course, the same must be done for the ball. Then, the two objects will move in the same direction when a move forward instruction is given to each.

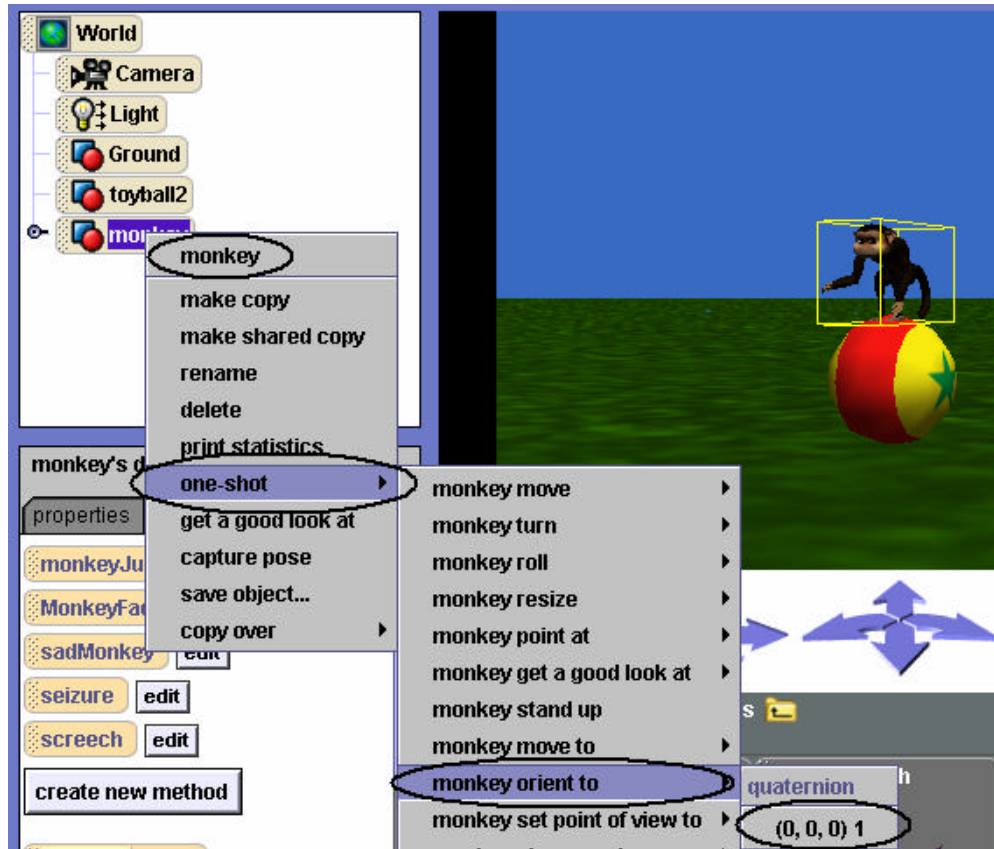


Figure T-3-9. Cascading menus for one-shot *orient to*

4 Character-Level Methods and Inheritance

The galleries of 3D models (characters) in Alice give us a choice of diverse and well-designed kinds of objects for populating and creating a scenic backdrop in a virtual world. As you know, each character object added to a world comes with a predefined set of actions it can perform – move, turn, roll, and resize (to name a few). After writing several programs, it is natural to think about expanding the actions an object “knows” how to perform.

In fact, we **can** add functionality to any character by writing a method specifically designed for that kind of object to carry out. In object-oriented programming, methods that add functionality to a particular kind of object (a *class* of objects) are called *class-level* methods. In Alice, we use the term *character-level* because we think of a specific kind of object as a 3D character model.

This chapter will focus on character-level methods as methods designed specifically for one kind of object. These methods are rather special because we can save the character along with its newly defined method(s) as a new kind of character. The new kinds of objects still know how to perform all the actions of the original character. In object-oriented languages, we would say that the new kinds of objects (a derived class) *inherit* all the properties and actions of the original objects (the base class). This concept is known as *inheritance*.

Once a new character has been saved, it can then be used later in a different program. Creating new characters in this way offers an advantage in that we can reuse code in many different programs.

Notes to instructors:

1) Creating a new character in Alice is not a complete implementation of inheritance. When a new character model is created in Alice, the new character gets a copy of the properties and methods of the base character model and this is saved in a new 3D model file. If you subsequently change the base character model, the changes in the base will not be reflected in derived character model.

2) Inheritance is accomplished in object-oriented programming languages via two mechanisms: a) adding (or overwriting) methods, and b) adding extra state information via the use of variables. This chapter will focus only on the former approach. Chapter 13 will introduce the use of variables to add functionality to an inherited character. Because variables are not visible/visual in the way the rest of the Alice environment is, variables per se are introduced much later in the text, after students have developed a mastery of several other programming concepts. See the preface for a more detailed discussion of our use of variables in Alice.

4-1 Creating New Characters

In the previous chapter, world-level methods were written to animate more than one object in a scene, where the objects were interacting with one another in some fashion. This chapter will focus on character-level methods. A major difference between world-level methods and character-level methods is that a character level method is specifically designed for one kind of object. The modified character can be saved out to create a new character.

Example

Consider the ice skater shown in the winter scene of Figure 4-1-1. We want the ice skater to perform typical figure skating actions. It would be most convenient if the skater already knew how to skate forward and backward and perform spin and axel movements. But, as with other characters from the gallery, the ice skater only knows how to perform simple move, turn, and roll actions. So, let's write some methods to teach the skater how to perform more complex movements. We begin with a method to make the skater perform a skate forward motion.



Figure 4-1-1. Ice skater

A Character-Level Method

Skating movements are complex actions that require several motion instructions involving various parts of the body. A possible storyboard for a skate forward movement is shown below. The storyboard breaks the skating action up into two segments – slide on the left leg and slide on the right leg. Then, each segment is planned (a design technique known as *stepwise refinement*). To slide on the left leg, the right leg is lifted and body rolled right. Then, the left leg is lowered and the body is made upright. Similar actions are carried out to slide on the right leg. The sliding actions are all taking place at the same time as the entire skater's body is moving forward.

```
Do together
  Do in order
    (Slide Left)
      Lift right leg and roll upper body right
      Lower right leg and return body upright
    (Slide Right)
      Lift left leg and roll upper body left
      Lower left leg and return body upright
  Move forward 2 meters
```

This method is designed specifically for the IceSkater and involves no other kinds of objects. Thus, instead of world-level, the method is written as a character-level method. IceSkater is selected in the Object tree and the “create new method” tile is clicked in the details pane. *skate* is entered as the name of the new method. (See Figure 4-1-2.)

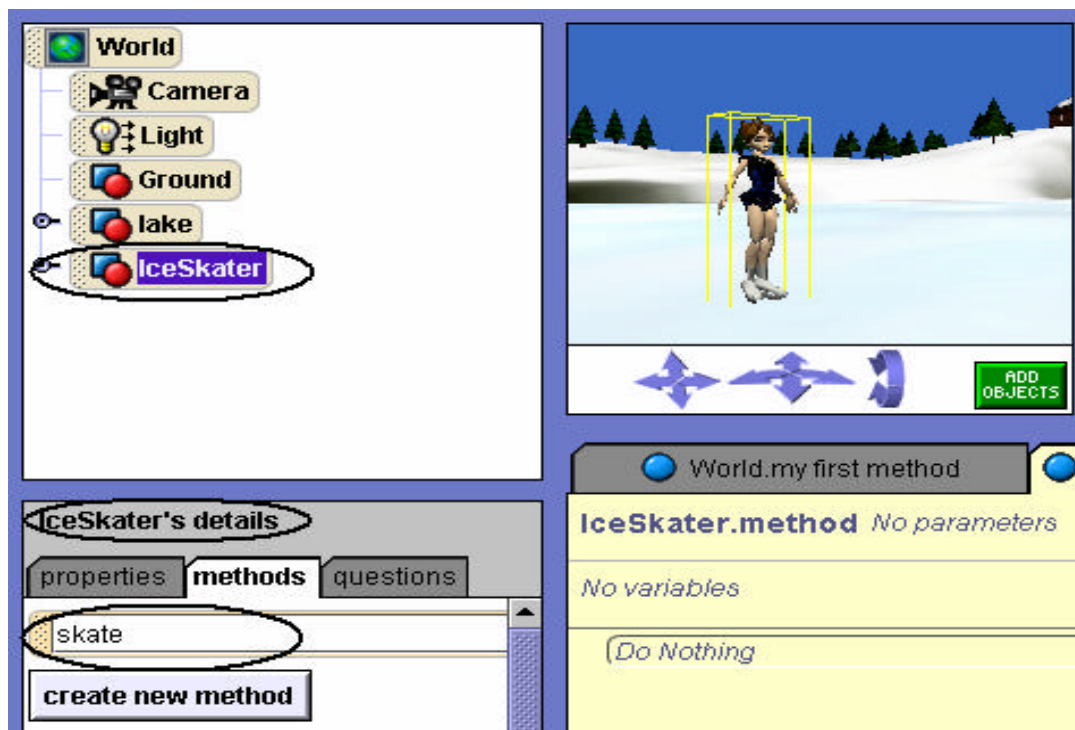


Figure 4-1-2. Creating a *skate* character-level method

To implement the method, instructions are entered in the editor. Figures 4-1-3(a) and (b) make up the definition of the *skate* method. The method name is *IceSkater.skate* (not *World.skate*) – indicating that the method is a character-level method. While the method looks somewhat complicated, you should not be intimidated by this code. It is actually quite simple – a slide left followed by a slide right and the whole body moves forward.

Figure 4-1-3(a) illustrates instructions to slide on the left leg:

The right leg is lifted (by turning the right thigh forward), the upper part of the body (abs) turned forward, and the head tilted (by rolling the neck left). A short wait is used to allow the entire skater to move forward. Then, the right leg is lowered and the abs and neck returned to their original positions (to prepare for the next leg slide).

Figure 4-1-3(b) illustrates similar instructions to slide forward on the right leg.

Note that left slide and right slide segments are each enclosed in a *Do in order* block. The *Do in order* blocks are nested within a *Do together* block along with an instruction that moves the skater forward simultaneously with the left and right sliding motion. The duration of the forward movement of the skater is the **sum of the durations of the left and right slides** so as to coordinate the sliding motions to begin and end at the same time as the forward motion of the entire body. When the *skate* method is called, the skater glides forward in a realistic motion.

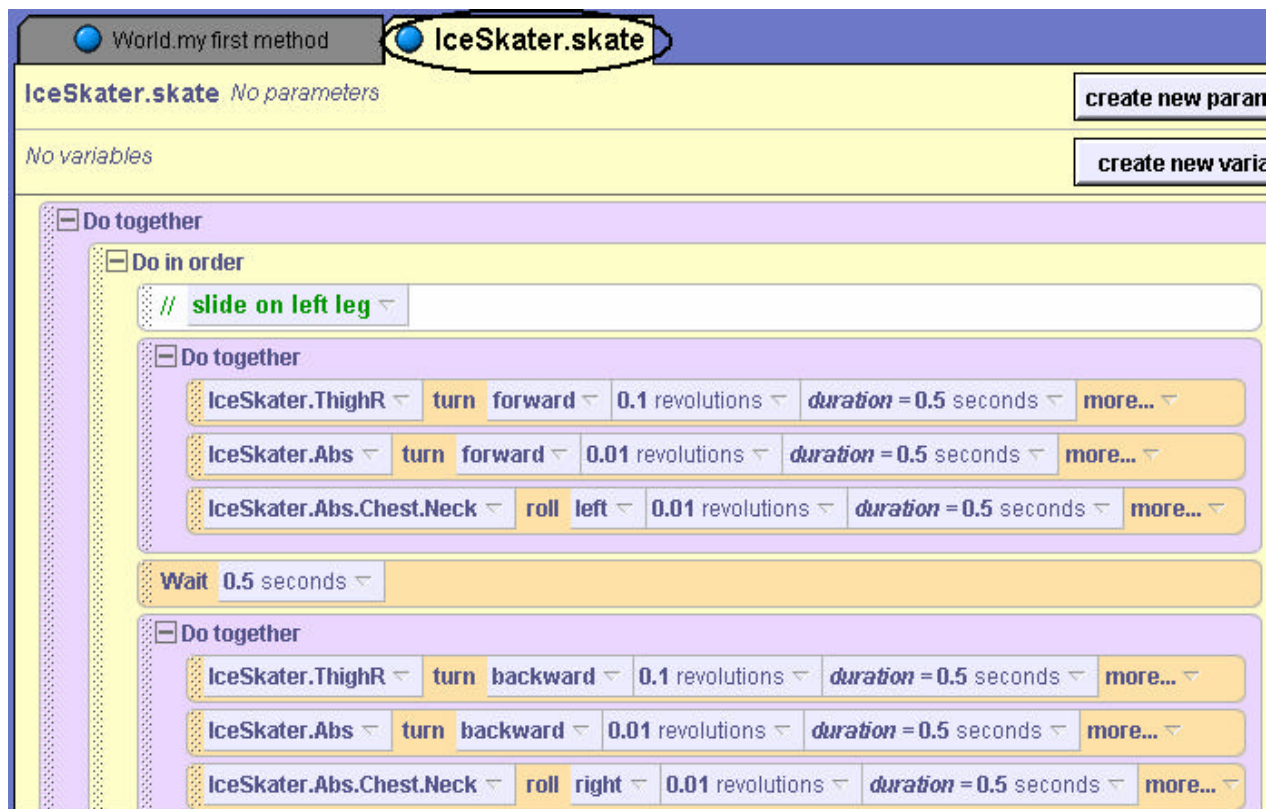


Figure 4-1-3(a) Slide left

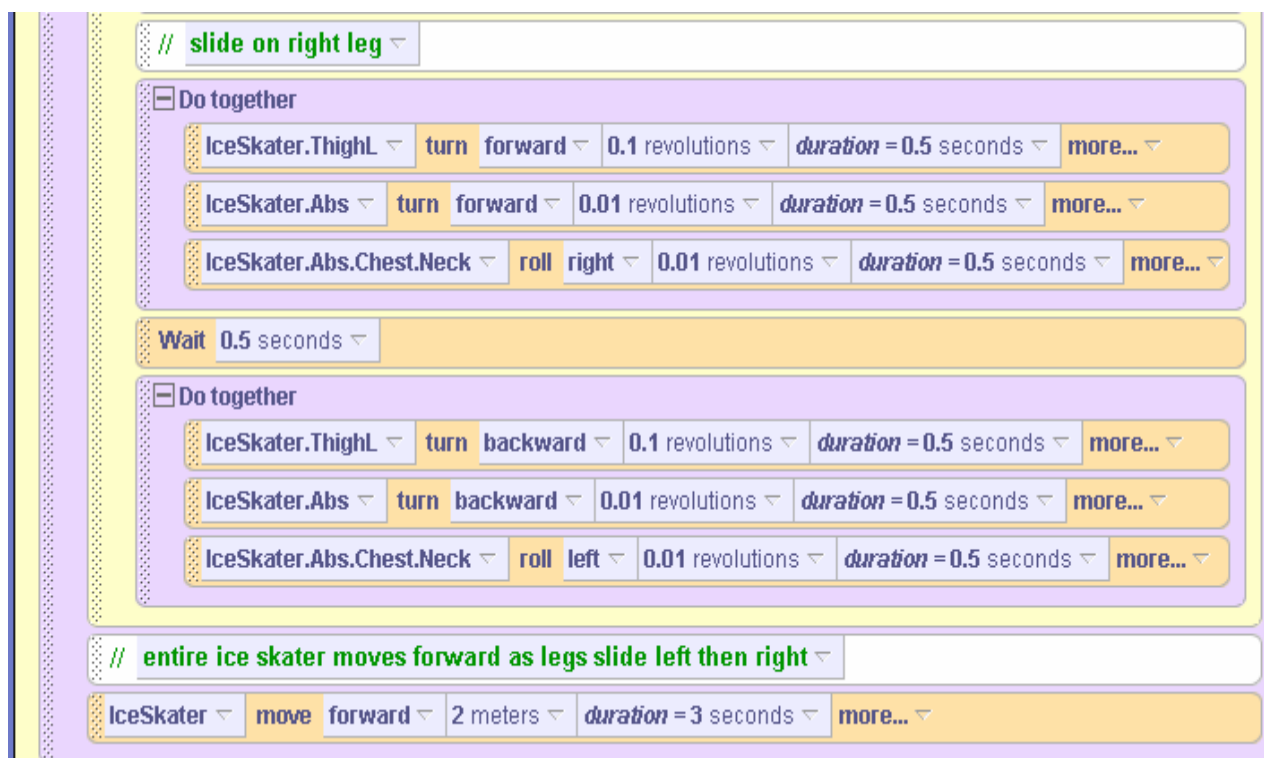


Figure 4-1-3(b) Slide right

A Second Example

The forward skate motion is truly quite impressive! Building on this success, let's write a second method to make the ice skater perform a spin. A storyboard for a spin could be as follows:

Do in order

1. Do together
 - (1) Lift left leg up in preparation for spin
 - (2) Lift arms out to the side
2. Do together
 - (1) Bend leg (at knee) in and out during spin
 - (2) Lower arms gradually as spin occurs
 - (3) Turn skater around six times
3. Lower left leg back to starting position

A character-level method can now be written with the storyboard as a guide. Figures 4-1-4 (a) – (d) make up the definition of the *IceSkater.spin* method.

Figure 4-1-4(a) illustrates a DoTogether block of instructions to prepare the skater for a spin:

The left leg is lifted to the side by turning the left thigh left and then backward. Then, the upper arms are rolled outward – away from the torso of the body.

Figure 4-1-4(b) illustrates a DoTogether block to move the left leg in and out as the body spins around:

The left leg is bent at the knee by first turning the thigh left and then the calf backward. After a short wait, the knee is unbent by reversing the thigh and calf movements.

Figure 4-1-4(c) continues the Do together loop started in Figure 4-1-4(b):

The arms are gradually returned to their starting position as the skater turns around and around.

Figure 4-1-4(d) illustrates instructions to return the skater's left leg to its starting position.

Once again, the code is a bit longer than methods we have written in previous examples. But, it is important to realize that it is easily understood because the method has been carefully broken down into small segments and each small segment of code within the method has been well-documented with comments that tell us what the code segment accomplishes as part of the method. Good design and comments sprinkled throughout the method make our code easier to understand as well as easier to write and debug.

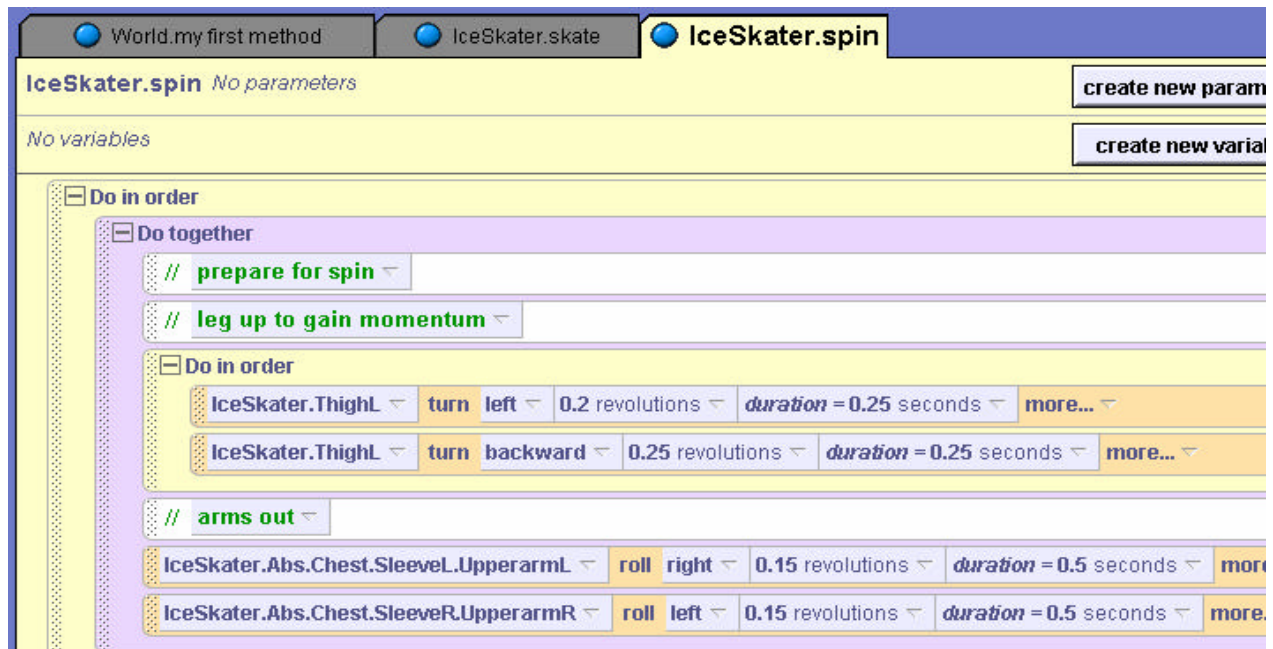


Figure 4-1-4. (a) Preparing for spin

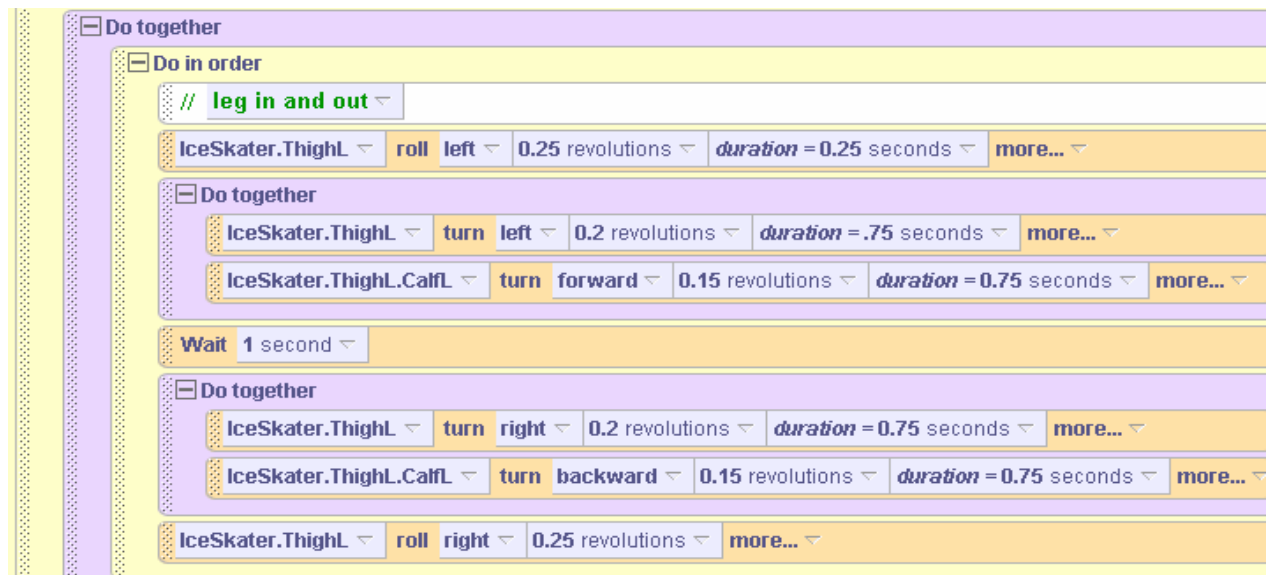


Figure 4-1-4. (b) Leg in and out as skater spins

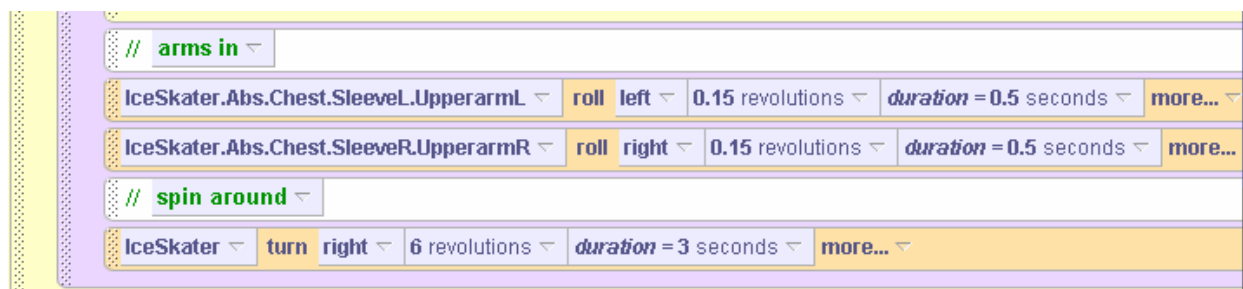


Figure 4-1-4. (c) Arms come in as skater spins

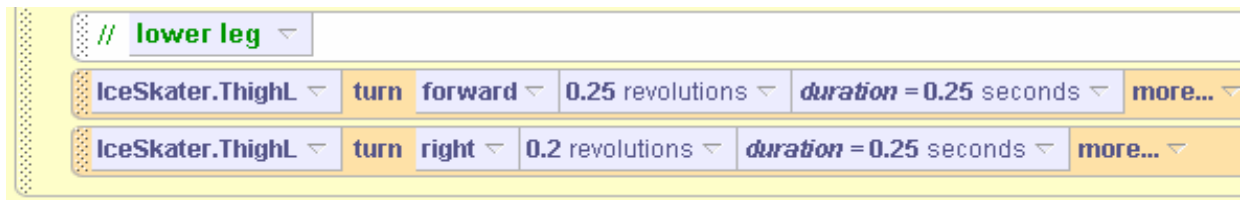


Figure 4-1-4. (d) Return left leg to starting position after spin

Saving a New Character

The IceSkater now has two character-level methods, *skate* and *spin*. Writing and testing the methods took some time and effort to achieve. It would be a shame to put all this work into one world and not be able to use it again in another animation program we might create later. This is why we want to save an object and its newly defined methods as a new character. Saving the IceSkater (with skate and spin methods) as a new kind of character is exactly what we want to do – so we can use the IceSkater in other worlds and not have to write these methods again in another animation program.

Saving an object (with its newly defined methods) as a new character is a two-step process. The first step is to rename the object. This is an IMPORTANT STEP! We want Alice to save this new character with a different 3D model filename than the original ice skater. To rename the IceSkater, we right click on the IceSkater in the Object tree, and select *rename*. Then, type in a different name. In Figure 4-1-5, we renamed the IceSkater as “CleverSkater”. (She has learned some clever moves.)



Figure 4-1-5. Renaming IceSkater as CleverSkater

The second step is to actually save the CleverSkater: right click on CleverSkater in the Object tree and this time select *save object...* In the Save Object popup box, navigate to the folder/directory where you wish to save the new character, as in Figure 4-1-6, and then click the Save button. The file is automatically named with the name of the object in the Object tree and given a filename extension **.a2c**, which stands for “Alice version **2.0** Character” (just as the **.a2w** extension in a world filename stands for “Alice version 2.0 World”).

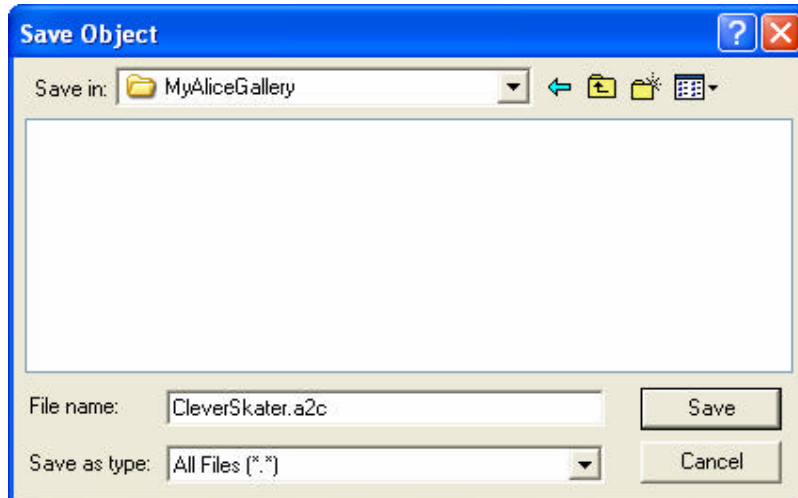


Figure 4-1-6. Save object pop up box

Once the object has been saved as a new kind of 3D model character, it can be used in any world by importing (use File→Import) the object into the scene. The CleverSkater will be just like IceSkater except that a CleverSkater knows how to skate and spin.

Inheritance

In programming languages such as C++, Java, and C#, creating a new class (kind of objects) based on a previously defined class is called *inheritance*. Inheritance in these languages is more complicated than in Alice, as they offer different forms of inheritance. But, the basic idea is the same – adding functionality to the base object by defining behaviors (methods) for a new kind of object.

Benefits: reuse and teamwork

Inheritance is considered one of the strengths of object-oriented languages because it supports reuse of code. Many programmers devote much of their time rewriting code they have written before. But, inheritance allows a programmer to write code once and use it again later in a different program.

A major benefit of creating new characters in Alice is that it allows us to share code with others in team projects. If you are working on a project as a team, each person can write character-level methods for one of the characters in the virtual world. Then, each team member can save their new character. As a team, a common world can then be created by adding all the new characters to a single team-constructed world. This is a benefit we cannot stress enough. In the “real world”, computer scientists generally work on team projects. Building animation programs as a team helps you develop skills in learning to work cooperatively with others.

Guidelines for Writing Character-level Methods

Character-level methods are a powerful feature of Alice. But, as with many powerful features in programming languages, character-level methods have the potential for misuse. Below is a list of “do's and don'ts” – guidelines to follow when creating character-level methods.

1. **Do** create many different character-level methods. They are extremely useful and helpful. Some characters in Alice already have a few character-level methods defined. For example, the lion character has methods *startStance*, *walkForward*, *completeWalk*, *roar*, and *charge*. Figure 4-1-7 shows a thumbnail image for the lion character (from the web gallery), including its character-level methods and sounds.



Figure 4-1-7. Character-level methods for lion

2. **Do not** play a sound in a character-level method unless the sound has been imported for the character (instead of the world). A sound that has been imported for a character will be saved with the character when a new character is saved, as can be seen in Figure 4-1-7.
3. **Do not** call world-level methods from within a character-level method. Figure 4-1-8 illustrates *CleverSkater.kalidoscope* -- a character-level method that calls a world-level method named *World.changeColors*. As explained earlier, an important motivation for creating character-level methods is to create a new character (saved as a new 3D model) and use it later in other programs for other worlds. If the *CleverSkater* object (with the *CleverSkater.kalidoscope* method) is saved as a new character and is then added to a later world where the *World.changeColors* method has not been defined, Alice will crash.

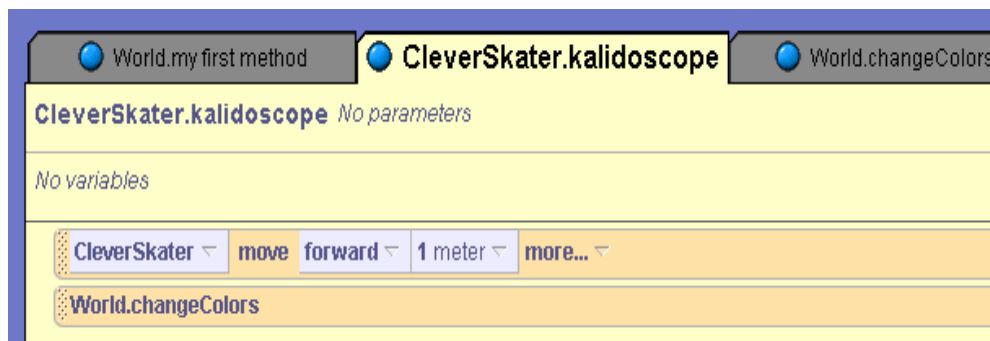


Figure 4-1-8. **Bad Example:** Calling a world-level from a character-level method

4. **Do not** use instructions for other objects from within a character-level method. Character-level methods are clearly defined for a specific character. We expect to save the object as a new kind of character and reuse it in a later world. We cannot depend on other objects being present in other programs in other worlds. For example, suppose a Penguin is added to our winter scene and we define a character-level method named *skateAround*, as in Figure 4-1-9. If we save the CleverSkater object (with the skateAround method) as a new kind of character and then add a CleverSkater object to a later world where no penguin exists, Alice will crash when we try to call the method.

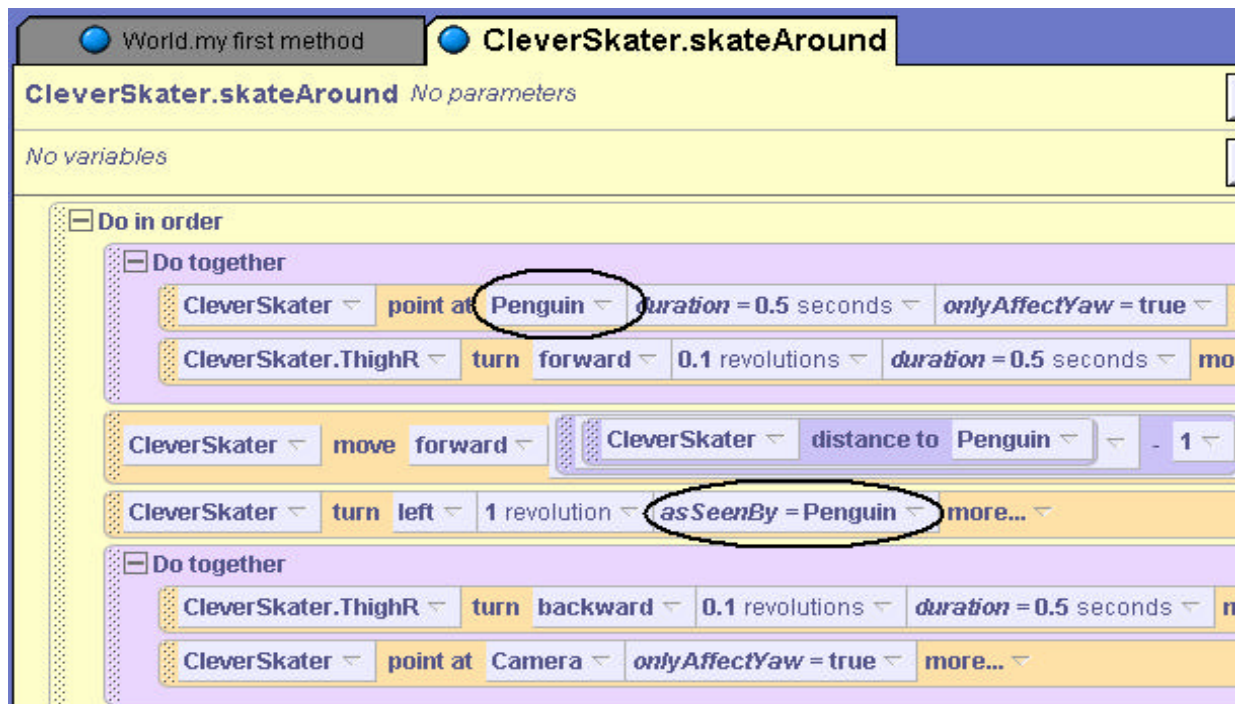


Figure 4-1-9. **Bad Example:** Instructions for another object in a character-level method

Character-level Method With an Object Parameter

What if you are **absolutely convinced** that a character level method is needed where another object is involved? One technique is to use an object parameter in the character-level method. Let's use the same example as above, where we want a CleverSkater object to be able to skate

around another object. The *skateAround* method can be modified to use a parameter, arbitrarily named *whichObject*, as shown in Figure 4-1-10. The *whichObject* parameter is only a placeholder, not an actual object, so we do not have worry about a particular object (like the penguin) having to be in another world. Alice will not allow the *skateAround* method to be invoked without passing in an object to the *whichObject* parameter. So, we can be assured that some sort of object will be there to skate around.

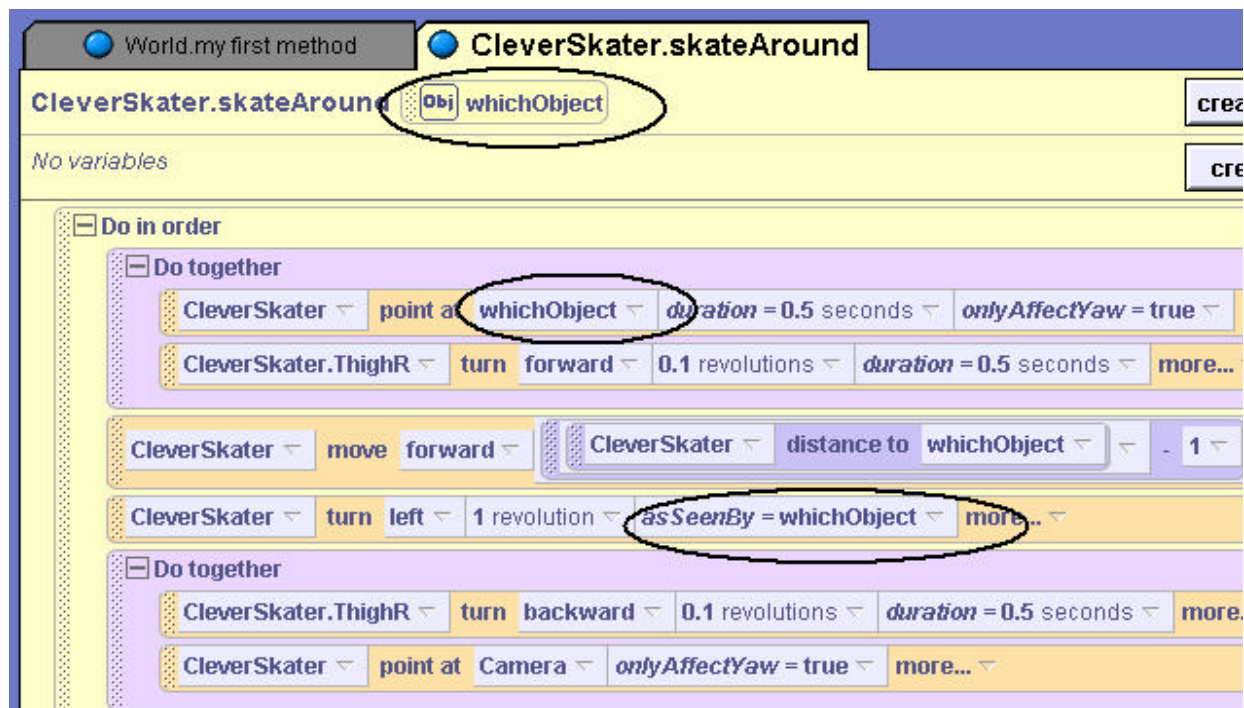


Figure 4-1-10. Using an object parameter in a character-level method

4-1 Exercises

1. LockCombination

Create a world with a ComboLock. Create four character-level methods: *leftOne*, *rightOne*, *leftRevolution*, and *rightRevolution* that turn the dial on the lock 1 number to the left, 1 number to the right, 1 revolution to the left, and 1 revolution to the right, respectively. Then, create a character-level method named *Open* that opens the lock and another named *Close* that closes the lock. (Hint: Use an *end Gently* style to make the motion more realistic.) Rename ComboLock as **TurningComboLock** and save it as a new kind of character.



Create a new world and add a TurningComboLock object to the world. Write a program that turns the dial of the TurningComboLock object so it opens to the following combination: Left 25, Right 16, Left 3. However, when going Right 16, make sure that the dial turns one full revolution by passing 16 once and stopping on it the second time. Then, pop open the latch, close the latch and return the dial to zero. (*Hint: Use wait to make the lock pause between each turn of the dial.*)

2. FunkyChicken

Starting with a basic chicken, create a character-level method *walk* that will have the chicken perform a realistic stepping motion consisting of a left step followed by a right step. Then, create a character-level method to make the chicken perform a “*funkyChicken*” dance motion. Save the chicken as a new character named **CoolChicken**. Create a new world and add a CoolChicken object to the world. In *myfirstMethod*, call the *walk* and *funkyChicken* methods. Play a sound file or use a *say* instruction to accompany the funky chicken dance animation.



3. Samurai Practice

Create a world with a Samurai and write character methods for traditional Samurai moves. For example, you can write *RightJab* and *LeftJab* (where the Samurai jabs his hand upward with the appropriate Hand), *KickLeft* and *KickRight* (where he kicks with the appropriate leg), and *LeftSpin* and *RightSpin* (where he does a spin in the appropriate direction). Save the Samurai as a new character named **TrainedSamurai**. Start a new world and add two TrainedSamurai objects. Create an animation where the two TrainedSamurai objects practice their moves, facing one another.



4 Summary

Character-level methods were introduced as a special kind of method written specifically for one type of object. An object for which character-level methods are defined may be saved as a new type of object – using a different name than the base character model. The new character inherits the properties and actions of the original character -- it is a fancy new model that knows how to do more things than the base model. Comments and stepwise refinement were used to make complex actions in character-level methods easy to understand and debug.

Some guidelines must be imposed for writing character-level methods. Only sounds imported for the character should be played, world-level methods should not be invoked, and instructions involving other objects should not be used.

A major benefit of defining new characters is we can use the characters over and over again in new worlds to take advantage of the methods we have written without having to write them again.

Important concepts in this chapter

- A character-level method is defined for a specific type of object.
- New character models can be created by defining character-level methods for an object and then saving the character object with a new name.
- Stepwise refinement is a design technique where a complex action is broken down into segments and then each segment is designed and implemented.
- Inheritance is an object-oriented concept where new kinds of objects (derived class) are defined based on an existing kind of object (base class). In Alice, the existing 3D models are like base classes and our new characters are similar to derived classes.
- Character-level methods can be written that accept object parameters. This allows a character-level method to interact with another character. Otherwise, character-level methods should avoid interaction with other characters.

4 Projects

1. CleverSkater

Create an even better CleverSkater than the one presented in this chapter. Start with the IceSkater character from the People collection, create the character-level *skateForward*, *spin*, and *skateAround* methods. In addition, create *skateBackward* and *jump* character-level methods. In *skateBackward*, the skater should perform similar actions to those in the *skateForward* method, but slide backward instead of forward. In a jump, the skater should move backward, bend and lift one leg, then move upward (in the air) and spin around twice before gracefully landing on the ice and lowering her leg back to its starting position.

Start a new world with a winter scene. Add your CleverSkater character to the world and create an animation where the skater shows off her figure skating skills. (Call each of the methods you have written.) Add a penguin and a duck to the world and use a parameter to pass the object the skater is to skate around to the *skateAround* method.

2. Your Own Creation

Choose an animal or a person from one of the galleries. The character selected must have at least two legs, arms, and/or wings that can move, turn, and roll. Write three character-level methods for the object that substantially add to what this kind of object knows how to do. Save the new character with a different name and then add it to a new world. Write an animation program to demonstrate the methods you defined for this kind of object.

Tips & Techniques 4

Properties

Objects have properties where information (about the object) is kept. At the lower left of the window in Figure T-4-1, the lilfish object properties are listed. Properties of objects include color, opacity, vehicle, skin texture, and other values that tell Alice how to display and animate the object.

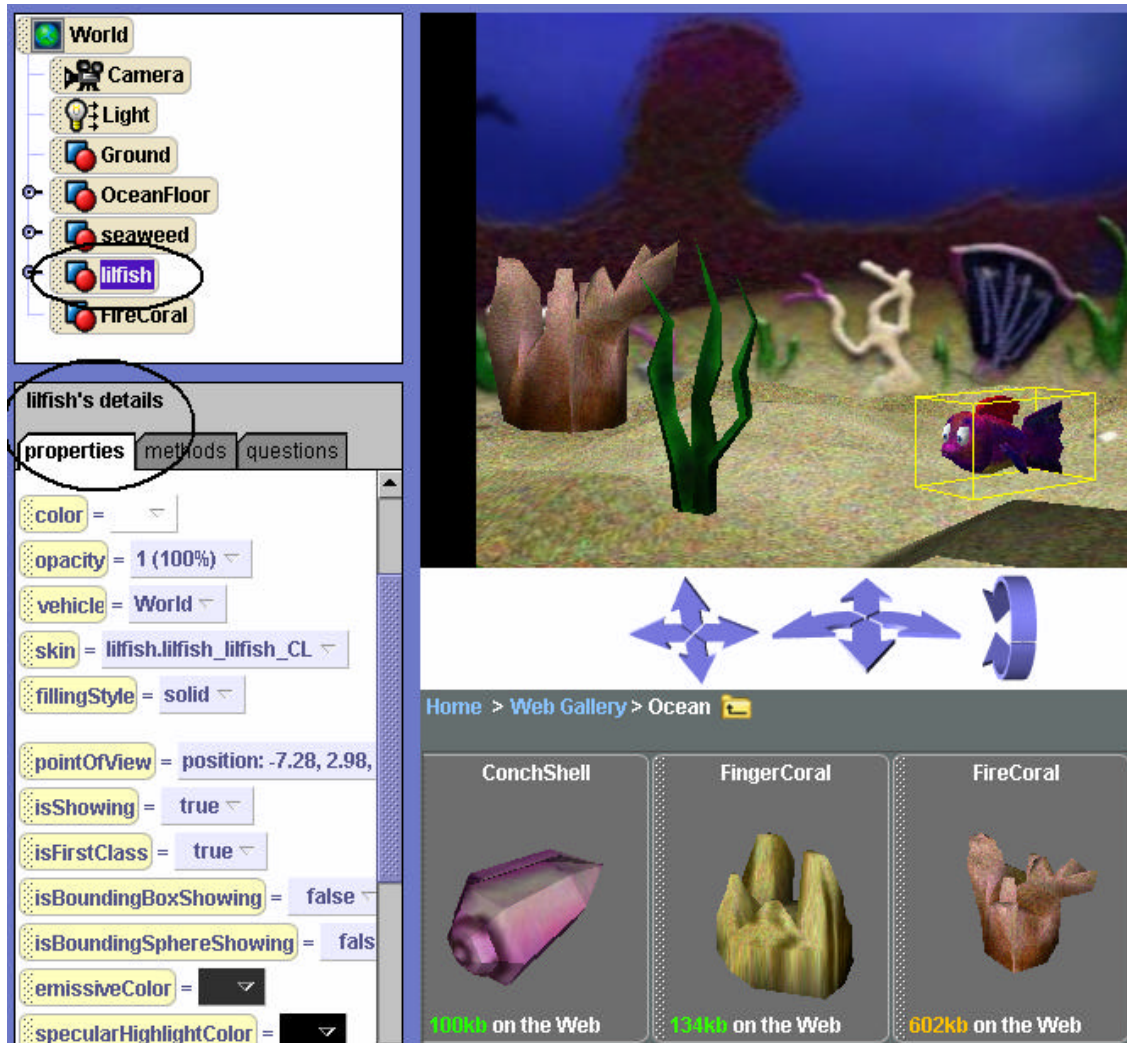


Figure T-4-1. Object properties

Setting Properties at Runtime

Throughout this book, examples use one-shot instructions to set (change) properties of an object at the time an initial scene is created. This section presents a tip & technique on how to change properties while an animation is running. The technical term for “while an animation is running” is “at runtime.”

If you buy a new cell phone, you will take it out of the box and try out all the cool features of your new phone – sort of a “poke and prod” kind of procedure. In the same way, you can learn about properties of objects by taking a “poke and prod” point of view. To set a property at runtime, drag the property tile into the editor and select the new value for the property. Alice will automatically generate a statement in the editor to set the property to the selected value. Set up a world with a couple of objects, drag some property instructions into the editor, and then run the program to see what happens! **Keep trying – on lots and lots of properties until you figure out what they all do.** To get you started, two property change examples are demonstrated here. Other property change instructions are scattered through the remainder of the book, where examples set various properties at runtime.

Setting *Opacity*. Suppose lilfish, as in Figure T-4-1, is swimming out to lunch and her favorite seafood is seaweed. Instructions to point lilfish at the seaweed and then swim toward it are shown below. The *wiggleTail* method is defined to make the fish waggle its tail in a left-right motion. The world and the swim method can be found on the CD with this book.



As the fish moves toward the seaweed, she will also be moving away from the camera. To simulate underwater conditions, we must consider that as lilfish swims away from the camera she should fade because water blurs our vision of distant objects. We can make lilfish become less visible by changing the opacity property. As opacity is decreased, an object becomes less distinct (faded). To change opacity, click on the opacity tile in the properties pane and drag it into the editor. From the popup menu, select the opacity percentage, as shown in Figure T-4-2.

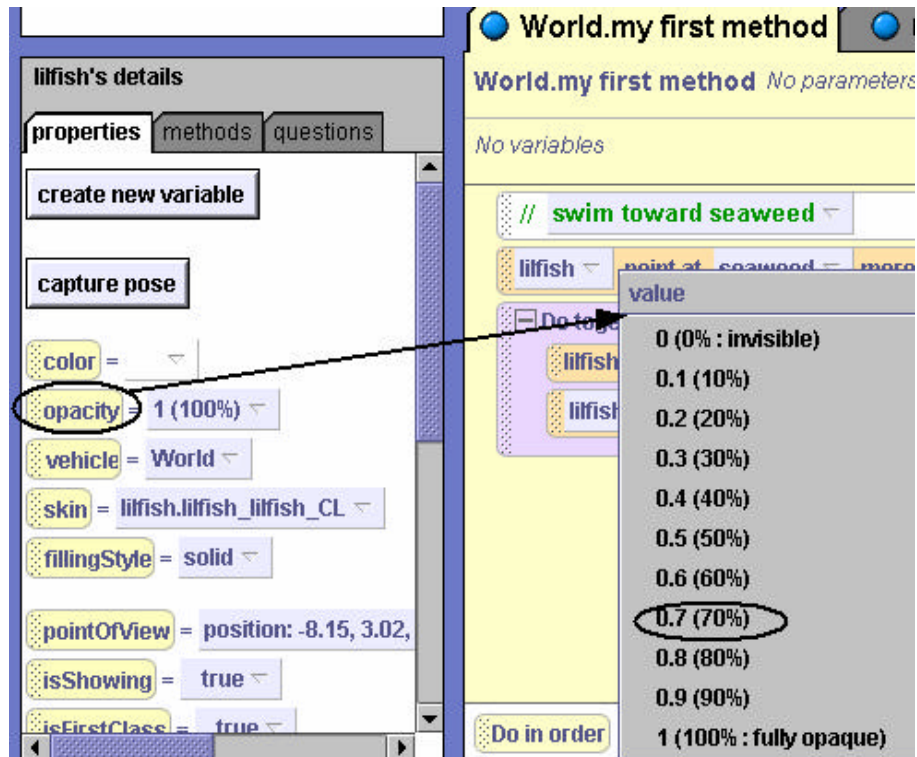
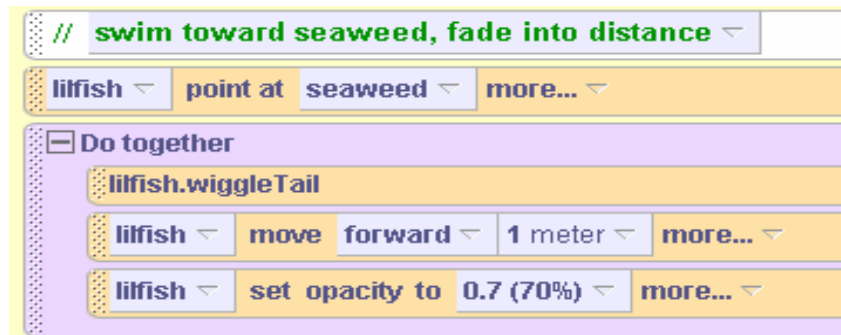


Figure T-4-2. Dragging the *opacity* property into editor

The resulting code looks like this:



When the world is run, the lilfish object will fade as shown in Figure T-4-3. Lowering opacity causes the object to become less opaque, and hence less visible. At 0%, the object will totally disappear from view. This does not mean that the object has been deleted. The object is still part of the world but is not visible on the screen.



Figure T-4-3. Fish image fades as opacity is decreased

Setting *isShowing*. Instead of gradually fading away into the distance, suppose we want an object to suddenly disappear? Alice provides a second mechanism for making an object invisible, a property called *isShowing*. At the time an object is first added to a new world, Alice automatically makes the object visible in the scene and *isShowing* is considered *true*. When we want an object to just evaporate “into thin air,” the *isShowing* property is set to *false* while the animation is running. To set the *isShowing* property to false, drag the *isShowing* property tile into the world and select false from the popup menu, as shown in Figure T-4-4.

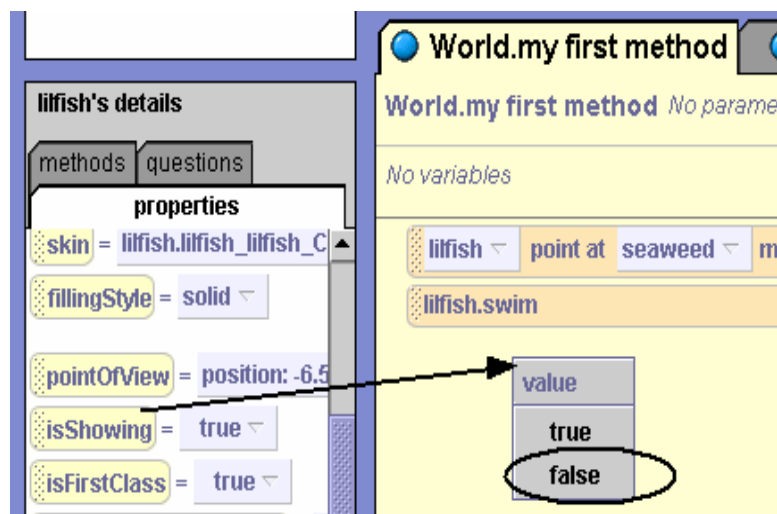
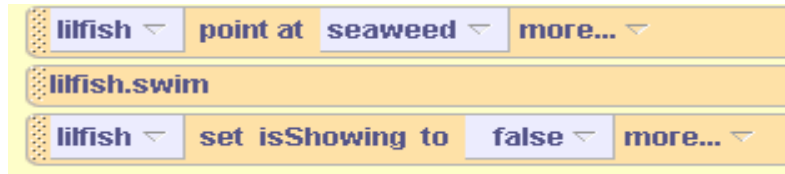


Figure T-4-4. Dragging *isShowing* property into the editor

The resulting code is shown below.



When an object's `isShowing` property is set to `false`, it is not removed from the world. Instead, the object is simply not rendered to the screen. Later in the program, the object can be made to reappear by setting its `isShowing` property back to `true`.

Built-in Questions (Functions)

The Alice system provides a set of built-in *questions* -- statements you can ask about properties of objects and relationships of objects to one another. In other computer programming languages, questions are often called *functions* and a function is said to "return a value."

What kinds of values can you expect to receive when you ask a question in Alice? Values can be any one of several different types. Five of the more common are:

- number (for example, 5)
- logic value (*true* or *false*)
- string (for example, "hello world")
- object (for example, a Robot)
- position (translational and rotational orientation)

What questions can you ask? As with methods, some questions are character-level and some are world-level. Character-level questions are about properties of a specific object in the world such as its height, width, and depth. World-level questions are more utilitarian, having to do with things like the mouse, duration times, and some math operations.

As an example of character-level questions, let's continue with the *lilfish* example used above. To view a list of questions about the *lilfish* object, select *lilfish* in the Object Tree and then the questions tab in the details pane, as in Figure T-4-5.



Figure T-4-4. Object questions

Character-level questions are divided into subcategories:

- *Proximity* – how close the object is (such as distance to, distance above) to some other object in the world
- *Size* – dimensions such as height, width, and depth, and how these compare to the dimensions of another object in the world
- *Spatial relation* – orientation compared to another object in the world (such as to left of, to right of)
- *Point of view* – position in the world
- *Other* – miscellaneous items such as the name of a subpart of the object

The questions shown in Figure T-4-4 are *proximity* questions. Some proximity questions such as “lilfish is within threshold of another object” and “lilfish is at least threshold away from another object” (*threshold* is a distance in meters) return a *true* or *false* value. Other questions such as “lilfish distance to” and “lilfish distance to the left of” return a number which is the distance in meters.

Take on that “poke and prod” attitude to explore all these different kinds of questions in the same way as you explored properties, suggested above. To show you how to get started, here is an example.

Asking about *Distance to*. Life is never easy in the ocean world. You may have noticed that a predator named *uglyfish* has entered the aquatic scene in Figure T-4-4. Of course, **lilfish is not dumb** – she is going to swim to the coral reef (FireCoral object) to hide. The program code needs to be modified to make lilfish swim to the coral. But, how far away is the coral? One way to find out is to use trial and error – that is, try different distances until we find one that works. Another technique to find the distance is to ask Alice a question: “What is the distance of lilfish to the FireCoral?” Alice will answer the question by returning the distance. Then, lilfish can be moved forward that distance. First, modify the code to point lilfish at the FireCoral object. Then, the move forward instruction should be modified to make lilfish swim the distance to the FireCoral object (instead of 1 meter, as previously written). To ask the *distance to* question, first drag the *distance to* tile into the editor and drop it to replace the 1 meter distance. From the popup menu, select the target FireCoral object, as demonstrated in Figure T-4-5.

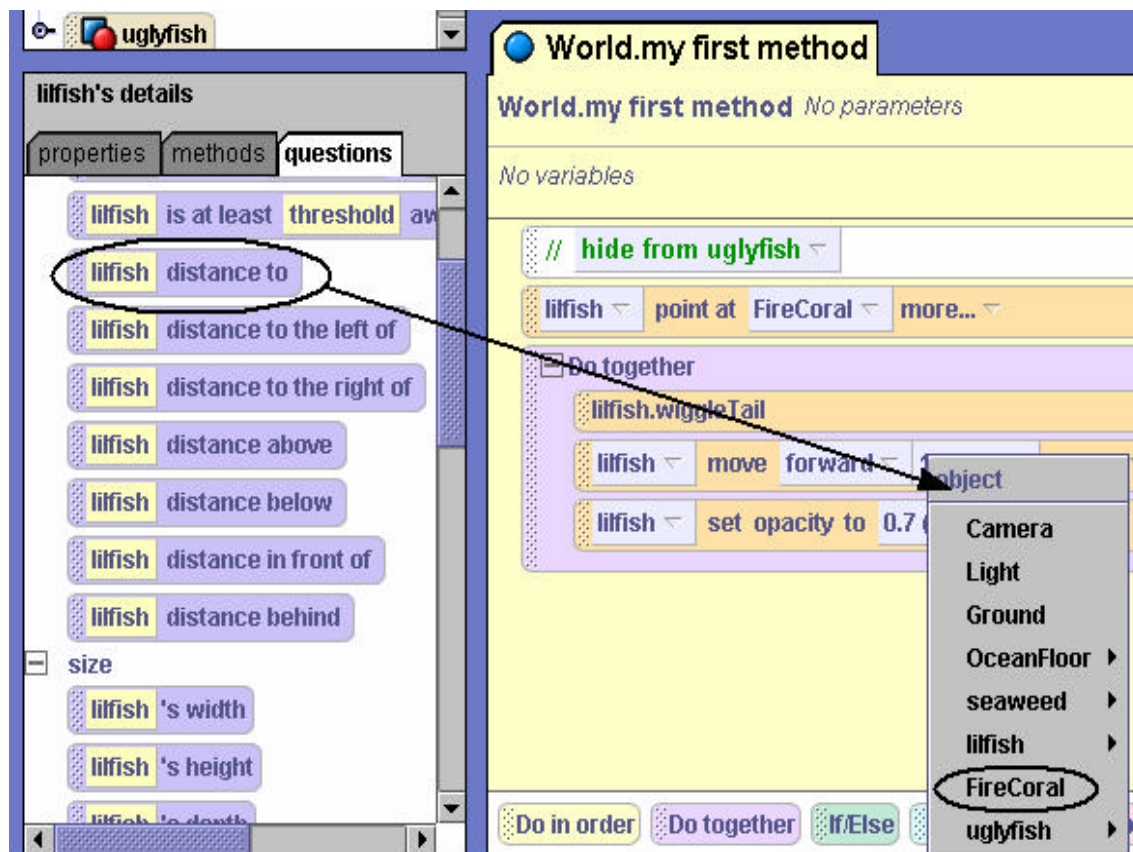


Figure T-4-5. Dragging distance to question into editor

The resulting code is:



Notice that no question mark appears at the end of the call to the question. Alice will interpret this as a question anyway and reply with the distance between the two objects.


Collision

When the program code to hide lilfish from uglyfish is executed, the animation is not exactly what was expected. Lilfish seems to run right into the middle of the FireCoral object. This is called a *collision*. In some animations, a collision is exactly what is desired. But, in this example, **we are not willing to believe that lilfish can swim right through the bony-like crust of the coral object**. The reason a collision occurs is that Alice answers the *distance to* question with a number representing the distance from the center of the fish to the center of the coral. To see how this works, look at Figure T-4-6.



Figure T-4-6. *distance to* is center-to-center

Expressions

How can a collision be avoided between these two objects? One way is to adjust the distance that lilfish moves so she doesn't swim right inside the coral. Adjusting the distance requires the use of an arithmetic expression. Alice provides arithmetic operators for common arithmetic expressions: add (+), subtract (-), multiply (*), and divide (/). To use an arithmetic expression to adjust the distance the object moves, click  to the right of the distance tile, then select math \rightarrow lilfish distance to FireCoral - \rightarrow 1.25.

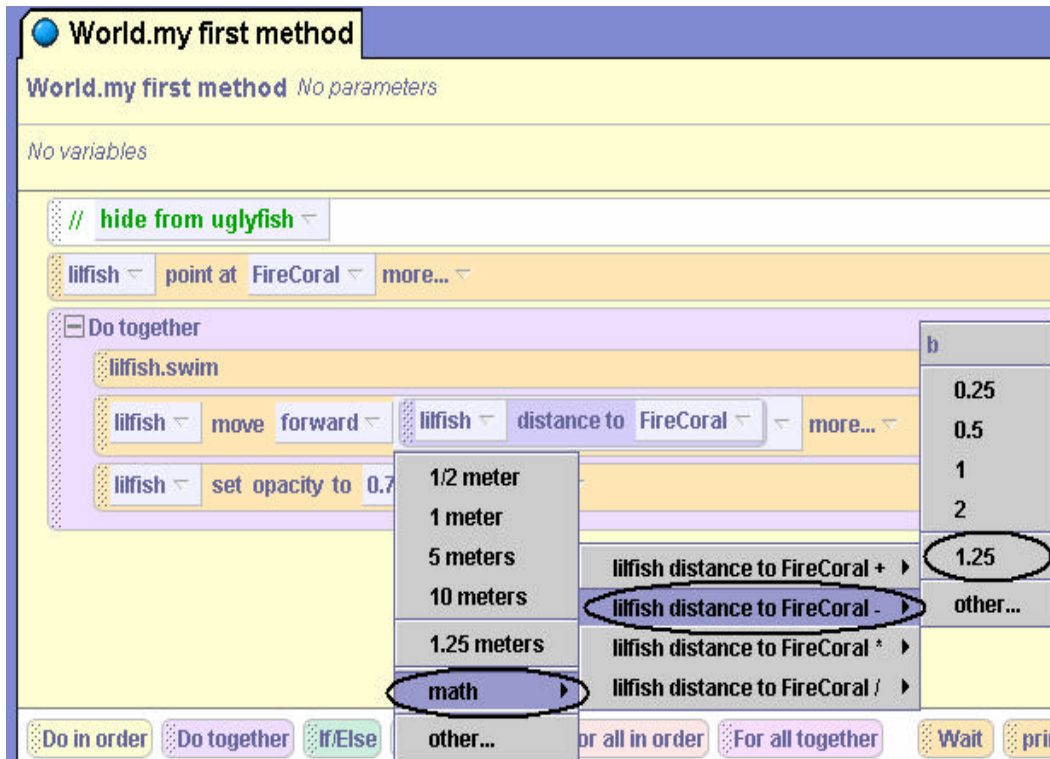
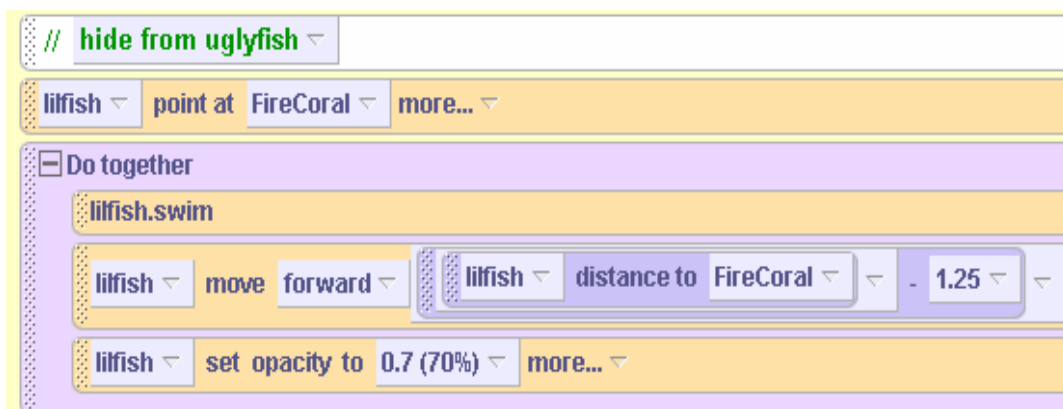


Figure T-4-7. Selecting an expression to adjust distance

The resulting instruction subtracts 1.25 meters from the distance as shown below. Now, the lilfish object will stop short of colliding with the coral.



The move to instruction

A *move to* instruction moves an object to a specific location in a world. The objects in our worlds are, of course, in 3D space. So, a location is specified using a reference to the position of the object along a 3D axis, as shown in Figure T-4-8.

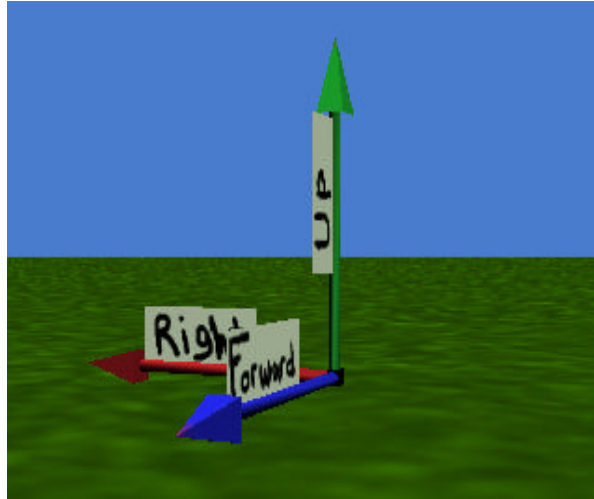


Figure T-4-8. Location reference along 3D axis

When an object is first added to the world, it is automatically positioned at location (0,0,0) which is at the center of the world. The first coordinate specifies the left/right position, the second an up/down position, and the third specifies a forward/backward position relative to the center of the world. Although we can tell Alice to move an object to a location by entering a location using number coordinates, much of the time we use the move to instruction with the position of another object as the target location. (Alice knows the 3-dimensional location of every object in the scene and can move another object to the same location.) This is easier to understand if we show you an example. Figure T-4-9 shows a coach practicing basketball in the gym.



Figure T-4-9. Basketball initial scene

The coach is going to shoot the basketball toward the hoop. We want to animate the movement of the basketball to the rim of the hoop on the backboard. This can be done using a move to

instruction with the position of the hoop rim as the target location. Creating the instruction is a two step process.

First, select the basketball in the Object tree. From the basketball's methods, drag the move to instruction into the editor, as shown in Figure T-4-10.

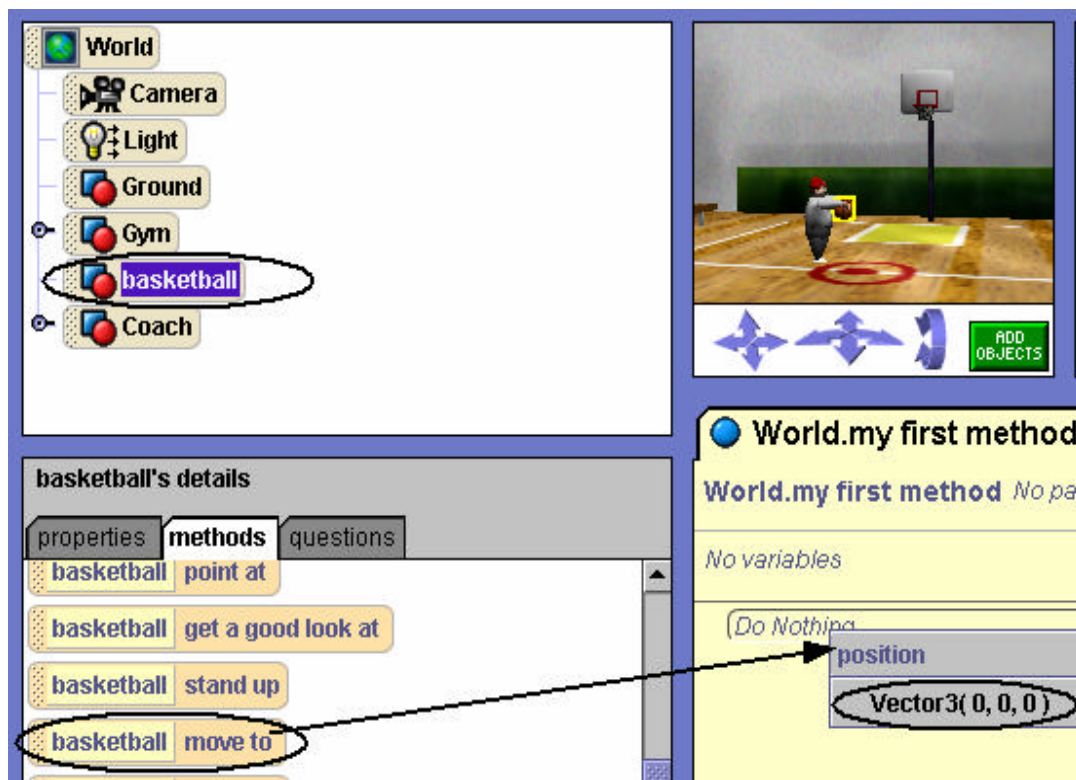


Figure T-4-10. Dragging *move to* into editor

Clearly, when the *move to* instruction is dragged into editor (see Figure T-4-10), the only choice is the default location Vector3 (0,0,0) -- the center of the world. The resulting instruction is:



Step two is to replace the default location with the position of the target object, in this case the hoop rim. To get the position of the hoop rim, drag in the built-in position question for the rim object that asks Alice to return the position of the rim, as shown in Figure T-4-11.

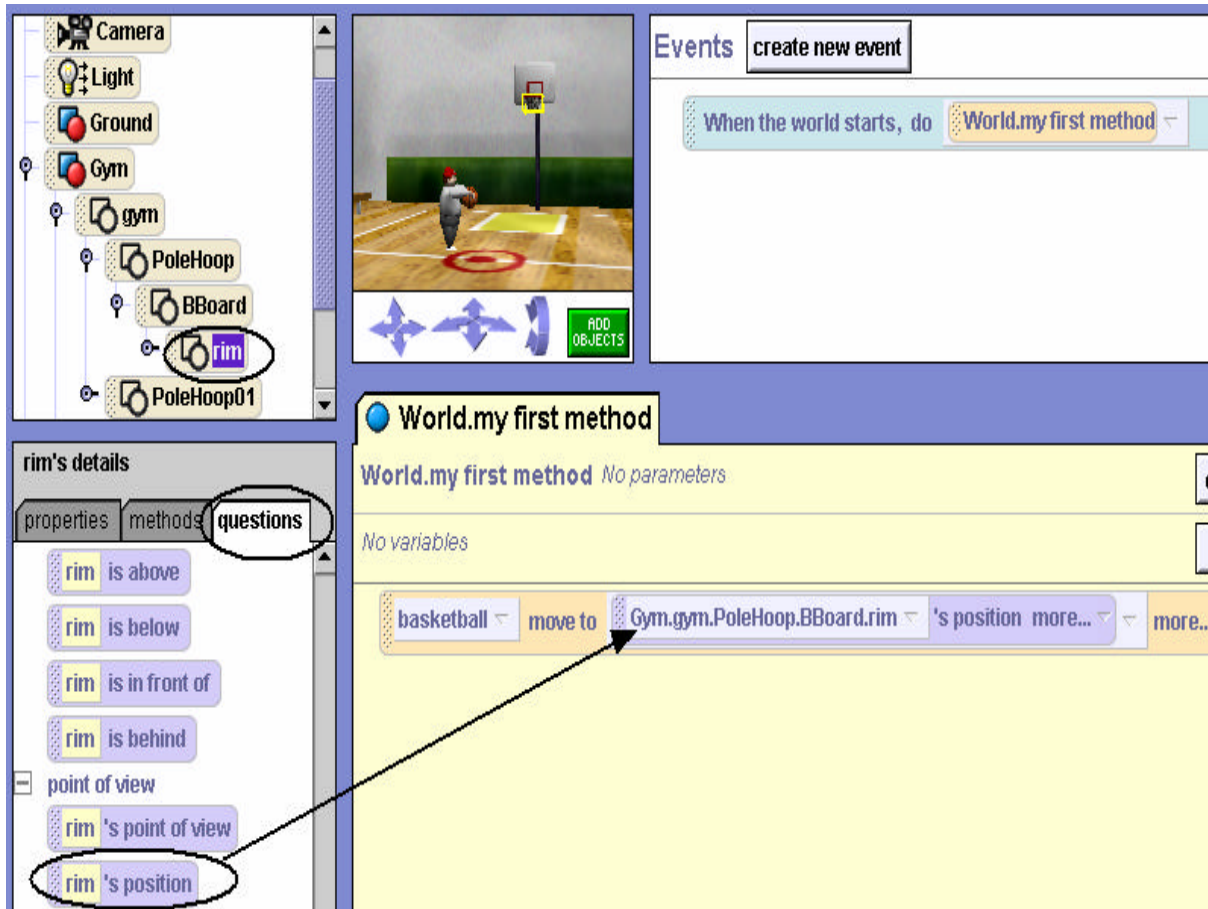


Figure T-4-11. Dragging in the position built-in question

Now, when the instruction is executed, the basketball will move to the rim of the hoop, see Figure T-4-12.



Figure T-4-12. Ball moves to rim

5 Interactive Programs: Events and Event-Handling

The real world around us is interactive. We drive cars that turn right or left when we turn the steering wheel. We change the channel on our television set by sending a signal from a remote control. We press a button on a game-controller to make a character in a video game jump out of the way of danger. It's time we looked at how to create interactive programs in Alice – where the objects in the scenes respond to mouse clicks and key presses. We have concentrated on writing programs that were non-interactive – we watched the objects perform actions in a movie-style animation. In this chapter, we will see how programs can be made interactive.

Much of computer programming (and of the Alice movie-style animations seen earlier) is computer-centric. That is, the computer program basically runs as the programmer has intended it. The programmer sets the order of actions and controls the program flow. However, many computer programs today are user-centric. In other words, it is the computer user (rather than the programmer) who determines the order of actions. The user clicks the mouse or presses a key on the keyboard to send a signal to Alice about what to do next. The mouse click or key press is an *event*. An *event* is something that happens. In response to an event, an action (or many actions) is carried out. We say the “*event triggers a response*.”

Section 5-1 focuses on the mechanics of how the user creates an event and how the program responds to the event. Naturally, all of this takes a bit of planning and arrangement. We need to tell Alice to listen for a particular kind of event and then we need to tell Alice what to do when the event happens. This means we need to write methods that describe the actions objects in the animation should take in response to an event. Such a method is called an *event-handler* method.

Section 5-2 describes how to pass parameters to *event-handler* methods. In some programming languages, arranging events and writing event-handler methods is a rather complex kind of programming. But, one of the achievements of Alice is that the event-response model is sufficiently simple to illustrate this material to novice programmers.

A special note to instructors:

We have found that interactive programs are fun and highly motivating to students. Nonetheless, this chapter may be safely skipped from a pedagogic perspective. Almost all exercises and projects in this book can be created in a non-interactive style.

5-1 Interactive Programming: Events and Event-Handlers

Control of flow

Writing an interactive program has one major difference from writing a program that is non-interactive (like the Alice movies we wrote in the previous chapter). The difference is how the sequence of actions is controlled. In a non-interactive program, the sequence of actions is pre-determined by the programmer. The programmer designs a complete storyboard and then writes the program code for the animated actions. Once the program is constructed and tested, then every time the program runs the same sequence of actions will occur. But, in an interactive program the sequence of actions is determined at runtime, when

- the user clicks the mouse, or presses a key on the keyboard.
- objects in the scene move (randomly or guided by the user) to create some condition, such as a collision.

Events

Each time the user clicks the mouse, or presses a key on the keyboard, an event is generated that triggers a response. Or, objects in the scene may move to positions that trigger a response. The point is: each time the program runs, it is possible that different user interactions or different object actions may occur such that the overall animation sequence is different from some previous execution of the program. For example, in a video game that simulates a car race, where the player is "driving" a race car, the sequence of scenes is determined by whether the player is skillful in steering the car to stay on the road through twists, turns, and hazards that suddenly appear in the scene.

Event-handlers

The question that must be answered is: how does this affect what you do as an animation programmer? As the animation programmer, you must think about all possible events and make plans for what should happen -- responses to the events. Animation methods, called *event-handlers*, are then written to carry out responses. Finally, the event must be *linked* to the event-handler. A *link* is a way to connect the event to the event-handler.

One important thing to keep in mind is that each time an event occurs and the event-handler is called, the location of objects in the scene may or may not be the same every time. This is because the user's actions may change the scene and the location of objects in the scene between calls to the event-handler.

Keyboard-Control Example

We begin with an acrobatic air show flight simulator. The initial scene, as can be seen in Figure 5-1-1, consists of the biplane in midair and some objects on the ground (house, barn, and so on.) A guidance system will allow the user to be the pilot. The biplane has controls that allow the pilot to maneuver the plane forward, left, and right. Of course, the biplane is a star in an acrobatic air show so we will want to program the biplane to perform a popular show stunt – a barrel turn. In the exercises at the end of this section, other acrobatic stunts can be added.

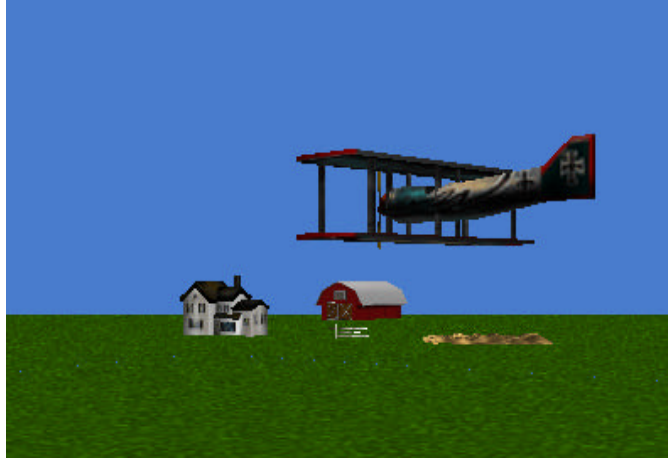


Figure 5-1-1. Initial Scene

Input

The whole idea of a flight simulator is to allow the user to interact with the biplane. The interaction consists of the user providing some input that sends a signal to Alice to animate a particular motion. Of course, a keyboard and mouse are two commonly available input devices that allow the user to provide this input. For a flight simulator, the user might press a set of keys on the keyboard. For example, arrow keys can be used, where each arrow key corresponds to a given direction of movement. Of course, input can also be obtained from mouse-clicks, the movement of a trackball, or the use of a game stick controller. In this text, we will rely on the keyboard and mouse to provide user input for interaction with the animations.

In our flight simulator, arrow-key and spacebar key presses will be used to provide input from the user. If the user presses the up arrow key, the biplane will move forward. If the user presses the left or right arrow keys, the biplane will turn left or right. For the acrobatic barrel turn, we will use the spacebar. The selection of these keys is arbitrary -- other sets of keys could easily be used.

Design -- Storyboards

Now, we are ready to design the flight simulator program – the set of instructions that tell Alice how to perform the animations. Each time the user presses an arrow key or the spacebar, an *event* is generated. The animation program consists of methods to respond to these events. Such methods are called *event-handlers*. To simplify the discussion over the next few pages, let's concentrate on two possible events: the spacebar press for the barrel turn and the up-arrow key to move the biplane forward. Two event-handler storyboards are needed, as illustrated in Figure 5-1-2 (a) and (b). Note that sound is optional and can be omitted.

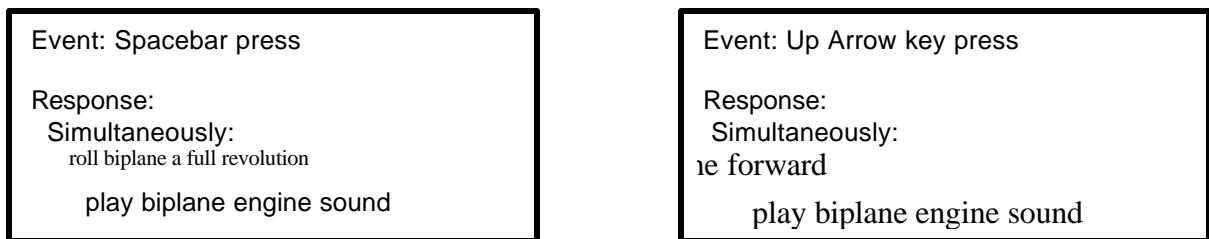


Figure 5-1-2. Storyboards: a) Spacebar—barrel turn b) Up Arrow— fly forward

Event-handler methods

The only object affected by key press events is the biplane. So, the methods can be character-level methods for the biplane object. Two methods will be written, *flyForward* and *barrel*. The *flyForward* method will handle an Up arrow key press event by moving the biplane forward as illustrated in Figure 5-1-3. The barrel method will handle a spacebar press event by rolling the biplane one complete revolution, seen in Figure 5-1-4. In the methods shown here, a sound clip is played simultaneously with the movement. (Review importing sound clips in Tips&Tricks 3.) The duration of the biplane movement and the sound clip are set to 1 second each. As noted previously, sound is a nice feature but can be omitted.

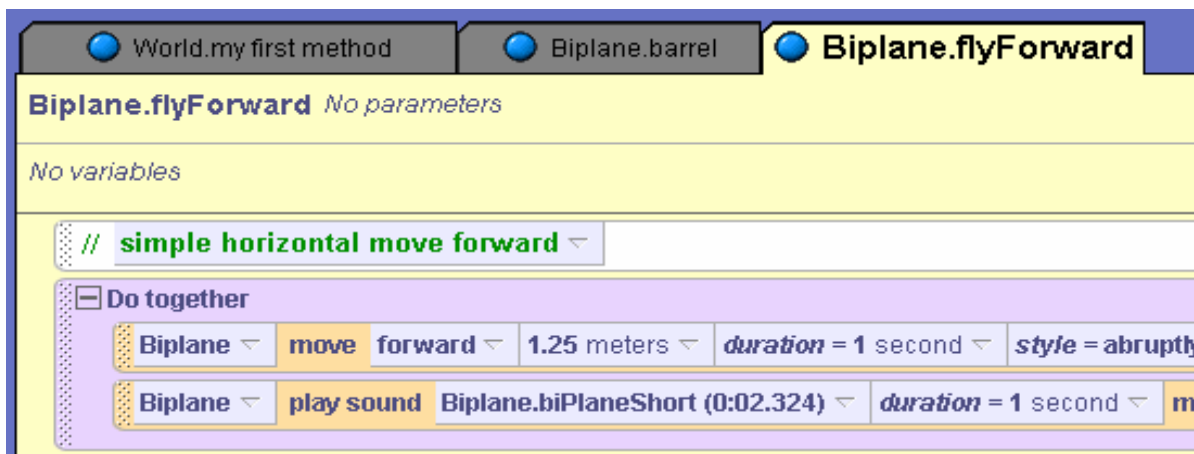


Figure 5-1-3. *flyForward* event-handler method

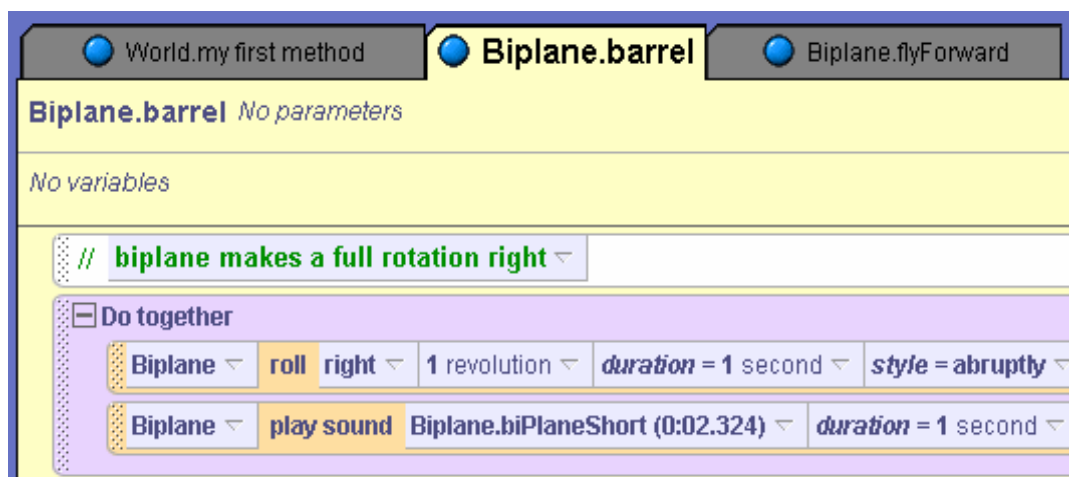


Figure 5-1-4. *barrel* event-handler method

Link Events to Event-handlers

Each event-handler method must be linked to the event that will be used to trigger the method as a response. The Events editor is where links are created. The Events editor is shown in Figure 5-1-5. By default, Alice creates a link between "When the world starts" (an event) and "World.my first method", as seen in Figure 5-1-5.

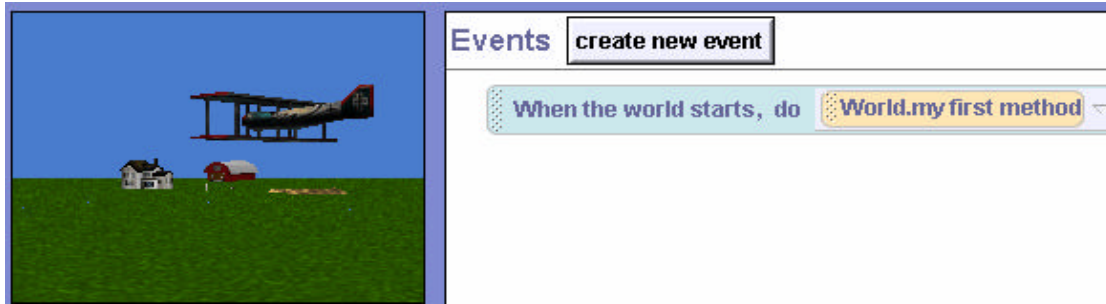


Figure 5-1-5. Event Editor

In the flight simulator, two events (Up arrow key press and Spacebar key press) are each to be linked to their corresponding event-handler method (*flyForward* and *barrel*). First, create an event by clicking the “create new event” button and then select the event from the pop-up menu of events. In Figure 5-1-6, the “When a key is typed” event is selected.

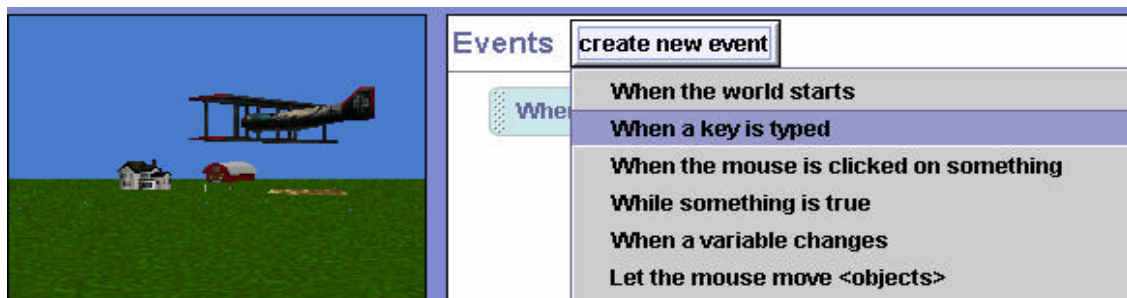


Figure 5-1-6. Creating a key press event

In Figure 5-1-7, an event for “any key” press has been added to the Events editor. Now, the Up arrow key is selected by pressing the image to the right of “any key” and selecting Up.

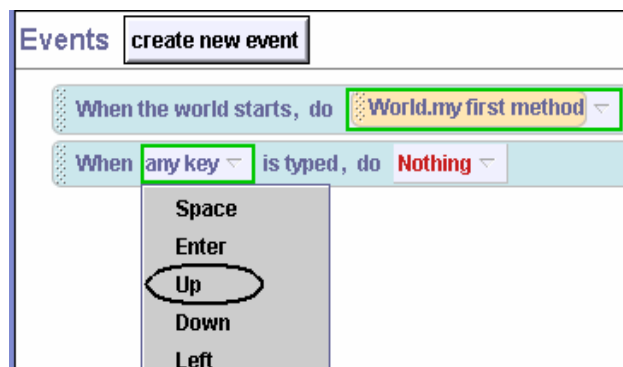


Figure 5-1-7. Specify Up Arrow key

Now that Alice has been notified that an Up arrow key event may occur, it is time to tell Alice what to do when the event occurs. As shown in Figure 5-1-8, click the image to the right of **Nothing** in the event instruction and then select *Biplane* and *flyForward* from the pop-up menus.

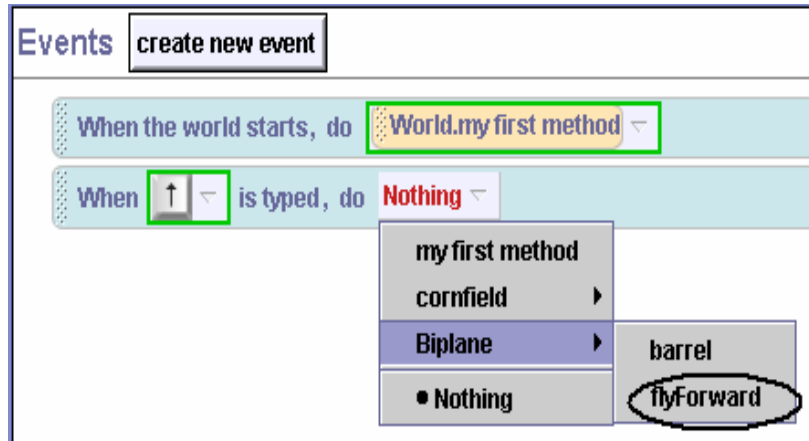


Figure 5-1-8. Link Event-handler to Event

The process is repeated to link the Space bar to the barrel method. Figure 5-1-9 shows the Events editor with both links completed.

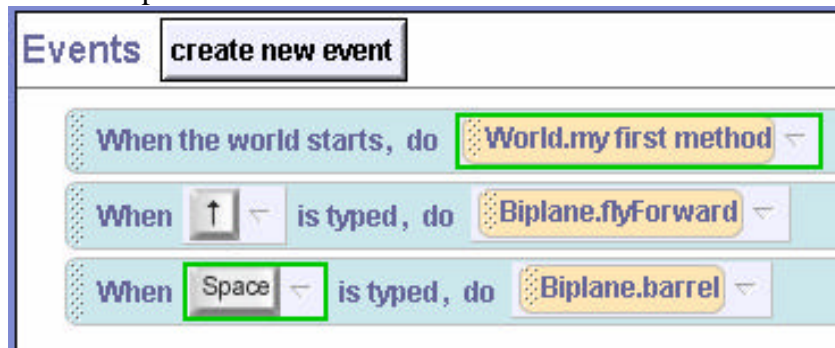


Figure 5-1-9. Links completed

Testing

Now that the links are completed, the world should be tested. To test the flight simulator, just save the world and press the play button. Note that this animation is not yet completed. Events and methods need to be created for the Left and Right arrow keys and other acrobatic stunts could be written. However, it is important to test event-handler methods as they are developed. Write a method and test it...write a method and test it... until the program is completed. This is a recommended program development strategy called *incremental implementation*. The advantage of this strategy is easier debugging. When something isn't working as expected, it can be fixed before it causes problems elsewhere.

Technical Notes

An interactive world such as a flight simulator requires that the user know what keys to press to make the simulation work properly. A startup method could be written in *World.my first method* to display 3D text for a quick explanation of the guidance system. After a few seconds, the 3D text can be made to disappear (set the *isShowing* property to *false*) and then the simulation can begin.

The biplane in the flight simulator is a great example of an object that is likely to move out of view of the camera. Of course, the disappearance of the biplane from the world view would be

frustrating for the user. One technique to keep the camera focused on the biplane is to make the biplane a vehicle for the camera, as shown in Figure 5-1-10. (Review setting vehicle property in Tips&Techniques 2.) Each time the biplane moves, the camera will then move with it.

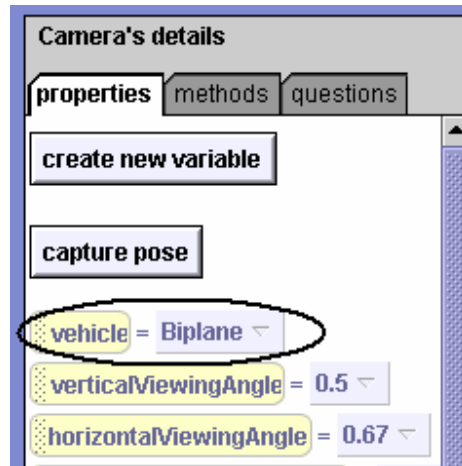


Figure 5-1-10. Make Biplane the Camera vehicle

Sometimes, when one problem is solved another problem is created. That is the case with making the biplane the camera's vehicle. Now, when a barrel turn is performed, the camera spins around with the biplane. What a wild effect this is -- the whole world seems to spin around! To prevent a stomach disorder caused by the wildly spinning world, perhaps the barrel method should be revised to temporarily set the vehicle back to the World while the roll action takes place. Then, the vehicle can be set back to the biplane just before the method ends. Figure 5-1-10 illustrates the revised method.

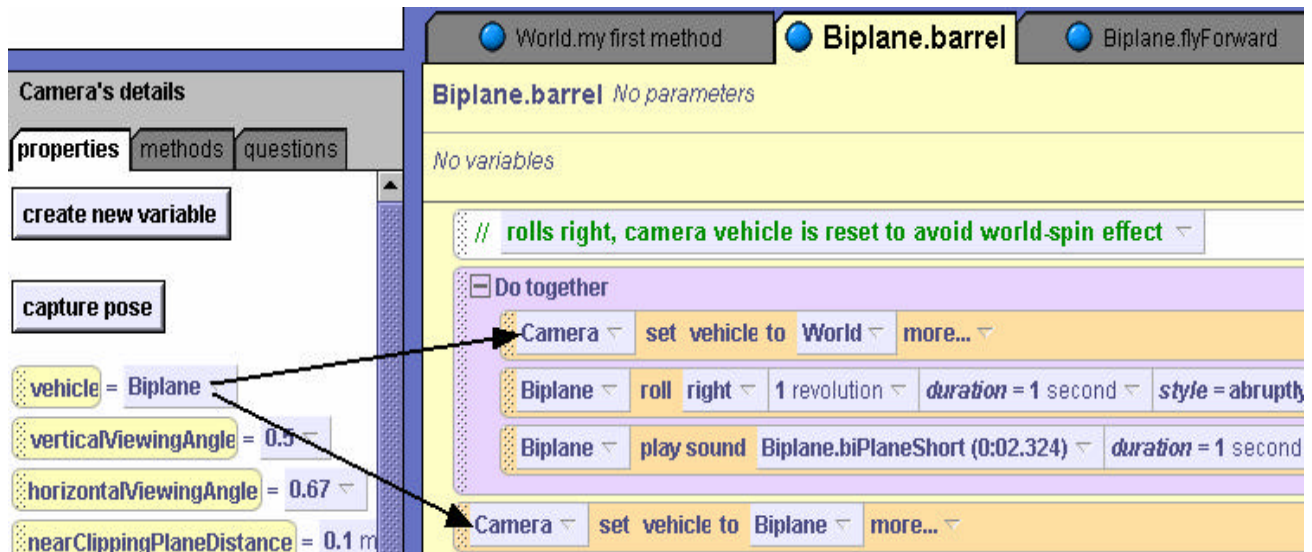


Figure 5-1-10. Revised barrel method

Mouse-Control Example

In this second example, Slappy, an adventuresome squirrel, has just gotten her own squirrel-sized Snowmobile. This animation will show Slappy's first ride on the Snowmobile. We want the user to control the forward and reverse motion of the Snowmobile. A remote control switch box will be used (see lower right of the initial scene in Figure 5-1-11). When the user clicks the green button on the switch, the Snowmobile will move forward and Slappy will ride the Snowmobile while screaming something like "wahoo." When the user clicks the red button on the switch, the Snowmobile and Slappy will move in reverse and Slappy will look at the camera and mutter something about having "the silly thing in reverse."



Figure 5-1-11. Initial Scene

Design -- Storyboards

Two events can occur: a click on the red button and a click on the green button of the control switch. Two event-handler storyboards need to be designed: one for a forward animation (rideForward) and one for a reverse animation (rideBackward), as illustrated in Figure 5-1-12 (a) and (b). Note that sound is optional and can be omitted.

<p>Event: Click on Red Button</p> <p>Responding Method: Simultaneously: Slappy and Snowmobile move forward Say "Wahoo" or Play a "forward" sound</p>	<p>Event: Click on Green Button</p> <p>Responding Method: Simultaneously: Slappy and Snowmobile move backward Say "Had silly thing in reverse" or Play a "backwards" sound</p>
--	--

Figure 5-1-12. Storyboards: a) rideForward

b) rideBackward

Event-handler methods

Of course, two objects are involved in these actions. So, the methods should be world-level methods. In both methods, Slappy should appear to be riding on the Snowmobile. With this in mind, it is obvious that the motions of Slappy and the Snowmobile must take place at the same time. As is often the case in computer programming, there are two ways to make Slappy and the

Snowmobile move together. One way to synchronize the moves is to use A *Do together* block. A second way is to make the Snowmobile be the *vehicle* for Slappy. The event-handler methods illustrated below will assume that the vehicle property is used to synchronize Slappy and Snowmobile motions. The advantage of using vehicle synchronization is that only one instruction is needed to move both objects. *Do together* blocks will still be useful for synchronizing movement with sound.

Link Events to Event-handlers

Each animation method must be linked to the mouse-click event that will be used to trigger the method as a response. The *rideForward* method should be linked to a mouse click on the green button of the TwoButtonSwitch object. And, the *rideBackward* method should be linked to a mouse click on the red button. To make a link, the "create new event" tile is clicked in the Events editor. In this example, "When the mouse is clicked on something." is selected, as shown in Figure 5-1-13 (a). The result is the creation of a new event listing in the editor, as seen in Figure 5-1-13(b).

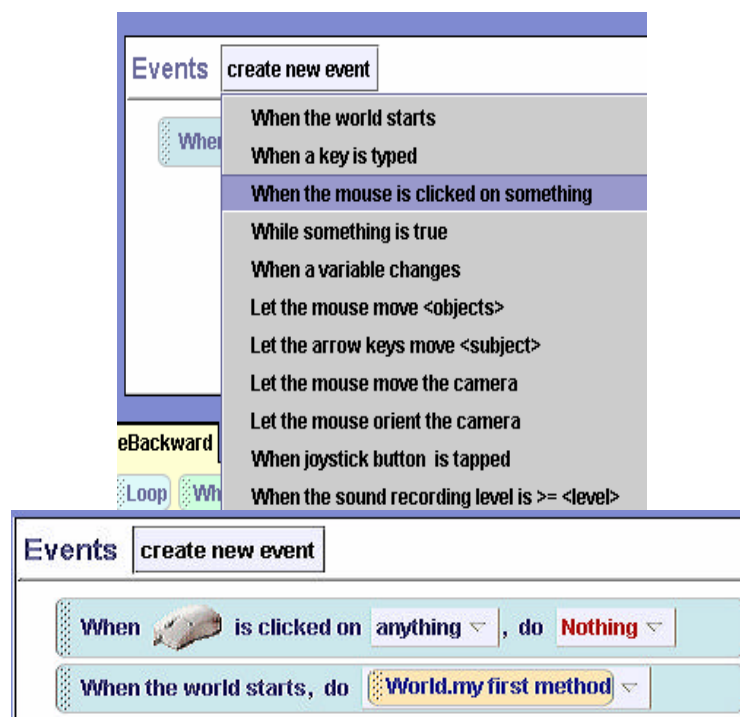


Figure 5-1-13. (a) Selecting mouse click event

(b) Result: mouse click event

Figure 5-1-14 shows the selection of the GreenButton as the target of the mouse click. Finally, a link is made to the *rideForward* event-handler method is made, as shown in Figure 5-1-15. The linking process is repeated for the RedButton event and the *rideBackward* event-handler method. The animation is not yet complete. Slappy's paw and head motions need to be added. Completion of the animation is left as an exercise.

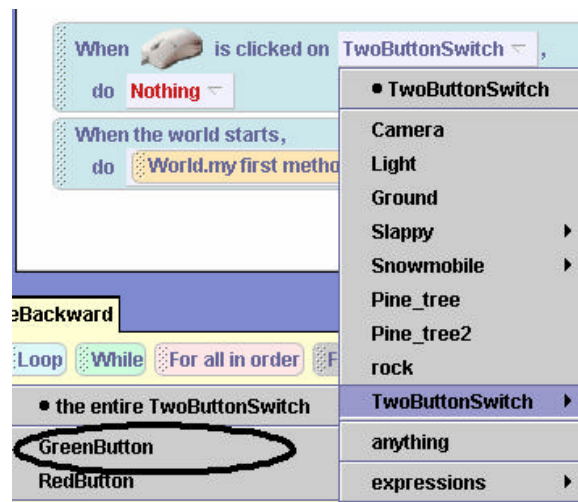


Figure 5-1-14. Select GreenButton as Target of Mouse Click

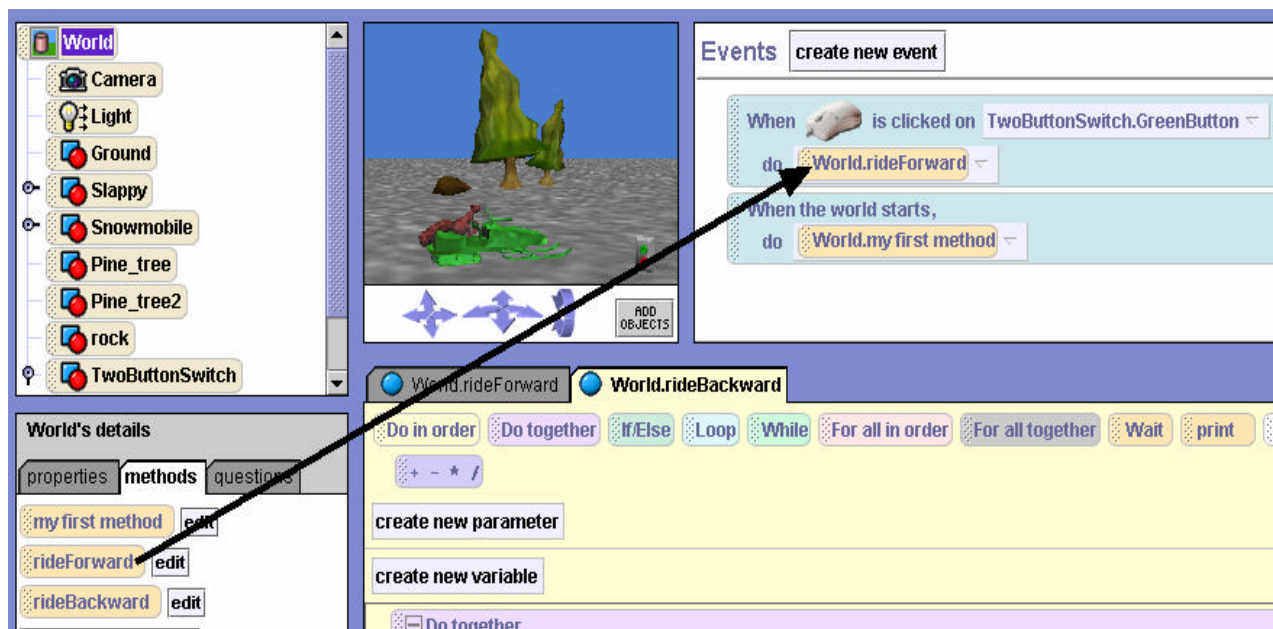


Figure 5-1-15. Drag Event-Handler Method into Link

Technical Note

We noted that Slappy and the Snowmobile must move together as if they were only one object. To guarantee that Slappy and the Snowmobile move together, not only did we make the snowmobile the vehicle for Slappy, we also used the *orient to* instruction on both objects in setting up the initial scene. See Tips & Techniques 3 for more details on *orient to*.

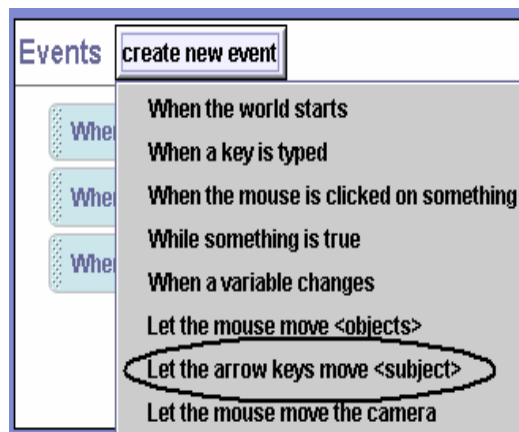
5-1 Exercises

1. Flight Simulator

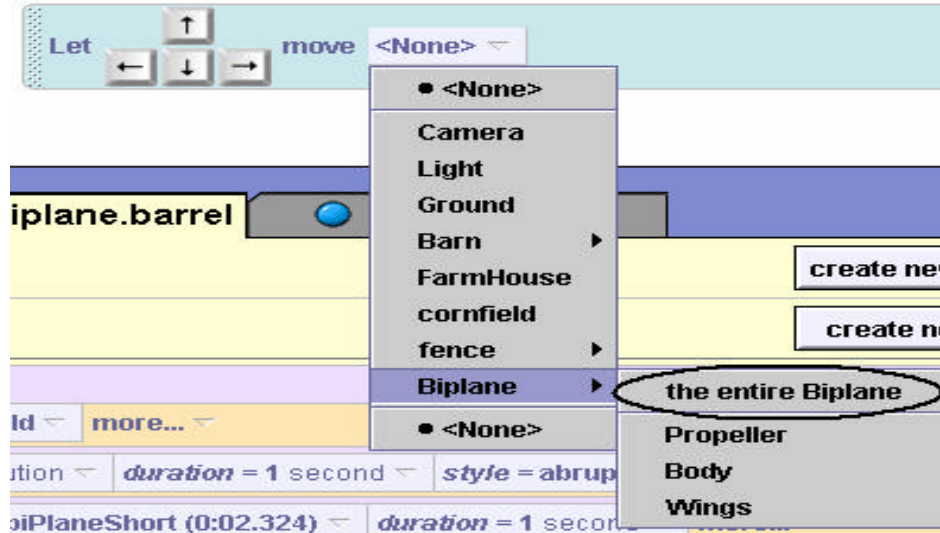
- (a) Create the world for the biplane acrobatic air show example as presented in this section. Implement the *flyForward* and *barrel* event-handling methods and link the methods to the corresponding events. *Make the move and roll actions have an abrupt style to reduce the pause in the animation between key presses.* If your computer has sound, use a biplane sound clip to make the animation more realistic.
- (b) When you have the *flyForward* and *barrel* methods working, add *flyLeft* and *flyRight* event-handlers for the left and right arrow keys to steer the biplane left or right.
- (c) Add a *forwardLoop* stunt that works when the user presses the Enter key.

2. Flight Simulator – alternate version

The arrow key press events work when the user *releases* the key. Of course, this means that multiple key press-release events are needed to keep the biplane moving. In this exercise, you can experiment with a different kind of event. Create a second version of the BiPlane Acrobat world (use **File→SaveAs** to save the world with a different name). In the second version of the world, remove the events that link the arrow keys to *flyForward*, *flyLeft* and *flyRight* event-handler methods. In the Events editor create a new event by selecting “let the arrow keys move <subject>” as shown below.



Then, link the biplane, as shown below. Run the flight simulator again to see the effect.



3. SlappyRide

(a) One example illustrated in this chapter was Slappy riding on a Snowmobile. Set up the initial scene and create the *rideForward* and *rideBackward* methods as shown above. Link the methods to mouse-clicks on the green and red buttons of the switch panel and test the program code to see if it works as described.

(b) When you have *rideForward* and *rideBackward* working properly, modify the methods: Add instructions in *rideForward* to make Slappy look forward, raise her front paws in the air (like you would on a roller coaster) and then lower her front paws back into the Snowmobile as the method ends. And, modify *rideBackward* so the squirrel will look at the camera while the Snowmobile moves in reverse.

(c) If your computer system has sound, add sound files to be played when *rideForward* and *rideBackward* methods are invoked.

4. Typing Tutor

Learning to type rapidly (without looking at the keyboard) is a skill requiring much practice. In this exercise, you are to create a typing tutor that encourages the beginning typist to type a specific set of letters. Use 3D text letters to create a word in the world. (For example, you could create the word ALICE with the letters A, L, I, C, and E) and create a method for each letter that spins the letter two times. When the user types a letter key on the keyboard that matches the letter on the screen, the letter on the screen should perform its spin method. Also include an additional method, *spinWord*, that spins the entire word in a circle when the user types the space bar. *Hint: Use As seen by.*

5. Rotational Motion

A popular topic in Physics is the study of rotational motion. Create a world with at least four objects (such as a Compass, Mailbox, MantleClock and Tire object.) Create a realistic rotation method for each object. For example, if one of your objects is a Compass, make the compass needles spin around quickly in opposite directions and then slowly point North. For each of the objects add a behavior that will invoke the rotational motion method for that object when the object is clicked.

6. Mad Scientist Magic

Who says that a Mad Scientist has no magic skills? Create a world using the Mad Scientist character (People gallery) behind the Counter object (Furniture gallery). On the countertop place three objects, the blender, the cup and the mug objects (Kitchen gallery). The point of the exercise is that when the Mouse clicks any one of the objects on the countertop the Mad Scientist will point at that object, raise his arm (like he is casting a spell of some sort) and have the object spin or turn in any way you wish. Hint: Both the mug and the cup have some kind of liquid in them so it may be interesting to have the liquid disappear (by setting the `isShowing` property to false) when the Mad Scientist performs his “spell” on it.

7. Ninja Motion

A ninja is trying out for a karate movie. But he needs a little practice. Create a world with a ninja object in a dojo. The motions the ninja needs to practice are: jump, duck, chop, and kick. Write motion methods for the ninja that include the following:

- a) `kickRightLeg`, `kickLeftLeg`: allows the ninja to kick his right/left leg, including all appropriate movements (i.e. foot turning, etc.)
- b) `chopRightArm`, `chopLeftArm`: allows the ninja to do a chopping motion with his arm

Create events and event-handlers that provide the user with controls to make the ninja jump, duck, chop and kick.

5-2 Passing Parameters to Event-Handlers

In chapters 3 and 4, parameters were shown to be powerful tools. Parameters allow us to customize methods to work with different objects and different numeric values. Parameters are useful in building either world-level or character-level methods. In this section, we will look at how to use parameters with events and event-handlers in interactive programs.

Once again, we will use examples to provide a context for presenting the concepts of interactive programming. The first example illustrates the mechanisms available for using parameters in response to events. The second example, while still using parameters in response to an event, illustrates an important concept used in game programs and simulations – namely that whatever the user clicks on can be passed as a parameter to an event-handling method.

Numeric parameter example

Jack is planning to try out for the school hockey team this fall. As a successful athlete, Jack knows that “practice is the name of the game.” Jack has set up a hockey net on the frozen lake surface and is going to practice his aim with the hockey stick to improve his chances of making the team. The initial scene is illustrated in Figure 5-2-1.



Figure 5-2-1. Hockey practice initial scene

Design -- storyboard

This animation could be the first phase of developing an interactive ice-hockey game. To design an interactive program storyboard, some thought must be given to what events will occur and what event-handler methods are needed. Let's allow the user to select the power factor behind Jack's swing of the hockey stick. The power factor will determine how fast Jack swings the stick and how far the hockey puck travels when hit by the stick. The power factor will be selected by a mouse-click on one of the power buttons in the lower-right of the scene. The yellow button will select low, green will select average, and red will select a high power factor. A storyboard, in textual form, is shown in Figure 5-2-2.

Event: Click on lowPower Responding Method: Swing with power factor 1	Event: Click on avePower Responding Method: Swing with power factor 2	Event: Click on highPower Responding Method: Swing with power factor 3
---	---	--

Figure 5-2-2. Event-handler storyboards

Three events, one event-handler

Three events are possible and three event-handler methods could be written (one to respond to each event). Somehow, writing three methods seems unnecessary for animations that perform the same simple action – just a different amount of power. A better solution might be to write just one method and send in the power factor as a parameter value.

A world-level method named *swing* will be created to manage the motion of Jack, the hockey stick and the puck objects. In the editor for this new method, a new parameter is created and given the name *power*, the Type *Number*, and an initial value of *1*, as illustrated in Figure 5-2-3 (a) and the result is shown in Figure 5-2-3(b).

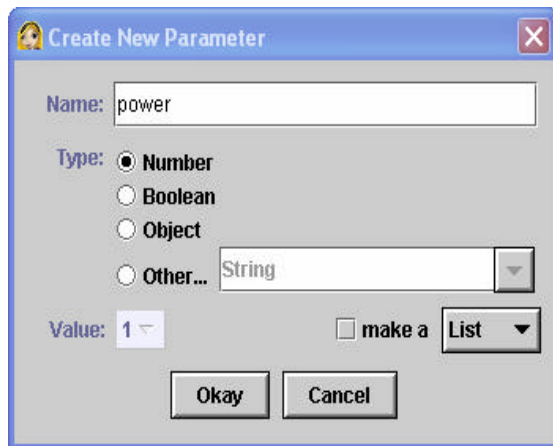
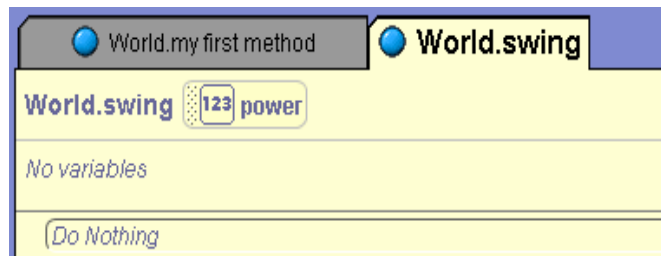


Figure 5-2-3. (a) Creating number parameter



(b) Result

Parameter is used for both duration and distance

The *swing* method is then constructed using the power parameter for duration (to control speed). The swing method is shown in Figure 5-2-4. Note that the power parameter is used as a ratio $1/\text{power}$. This way, if the power factor is a larger number, the duration will be shorter and the animation will occur faster. Also, the power parameter is used as the distance the hockey puck moves towards the net.



Figure 5-2-4. The *swing* method with *power* parameter

Link events to event-handler

Three events are possible so three events are created in the Events editor, Figure 5-2-5.

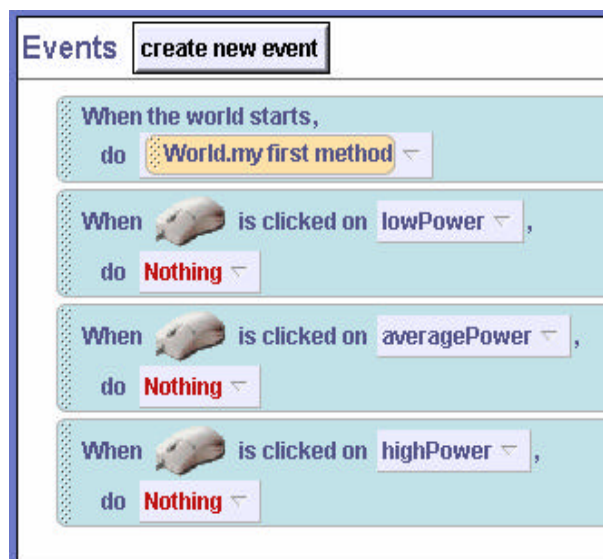


Figure 5-2-5. Three events: one for each object that can be clicked

Pass a number parameter

Now, the *swing* method can be linked as an event-handler for the mouse-click events. Following the same procedures as described in the previous section, the event-method links are created, as shown in Figure 5-2-6. The parameter is a different number in each link. When triggered by a

mouse click on the lowPower button, *power* = 1 , on averagePower *power* = 2, and on highPower *power* = 3. In each case, the passed parameter is a number value.



Figure 5-2-6. Link *swing* method and use a number parameter

Testing

When parameters are used in event-driven programming, it is especially important to run the animation several times, each time creating different events to be sure each possible parameter value works as expected. A well-known guideline for testing numeric parameters is to try a small value, a large value, and perhaps even a negative value— just to be sure the program works with a range of parameter values. In this example, no negative value is used but we could put one in just to see what would happen.

Object parameter example

Scenario: In the ancient world of Greek mythology, Zeus was an all-powerful god. If Zeus were angered, a thunderbolt would be shot out of the heavens and strike anyone who got in the way. This animation is a simulation of an ancient Greek tragedy. (The Greeks were fond of tragic dramas.) The initial scene is constructed with Zeus, overlooking a temple scene from his position on a cloud, a thunderbolt object, and some Greek philosophers named Euripides, Plato, Socrates, and Homer. The initial temple scene is illustrated in Figure 5-2-7. The thunderbolt object has been hidden within a cloud (the one immediately in back of Zeus). Also, a smoke object (a special effect) has been positioned below the ground – initially out of sight. (A one-shot instruction was used to move the smoke object down 5 meters.)



Figure 5-2-7. A Greek tragedy initial scene

Design -- Storyboard

To make this animation interactive, we will let the user choose the object that will be the next victim of Zeus's anger. When the user clicks on one of the objects, Zeus will point at that object. Then, a thunderbolt will strike the object that was clicked. The smoke object will move up to cover the object. The smoke is a special effect used to make an object appear to meet a sad fate. A textual storyboard is shown in Figure 5-2-8.

Event: An object is clicked with the mouse

Event-handler method:

The following actions occur in order:

- 1) Zeus turns to point at clicked object
- 2) A thunderbolt strikes the object
- 3) Smoke appears as the object meets a tragic ending

Figure 5-2-8. Greek tragedy storyboard

Writing the event-handler

The first step, as in the previous example, is to create a world-level method, *shootBolt*. This method should receive an object parameter, as illustrated in Figure 5-2-9. In this example, the parameter was arbitrarily named *who*.

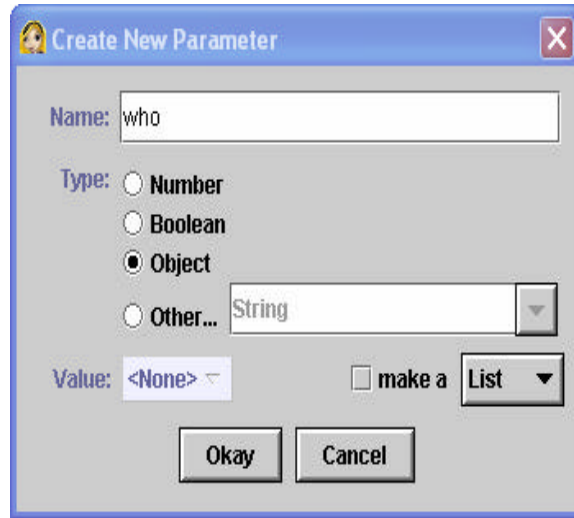


Figure 5-2-9. Creating an object parameter

The complete code for the *shootBolt* event-handler method is illustrated in Figure 5-2-10. In interactive programs, it is possible the user can repeat an action. In this world, the user may mouse click on more than one philosopher and so we must prepare the lightening bolt for another possible strike. The lightening bolt is made to disappear (by changing its opacity to 0%) after it strikes the object. Then, at the end of the method, the invisible bolt is moved back up to the cloud. Now, if the user clicks on another object, the bolt is back in position to become visible and strike again.

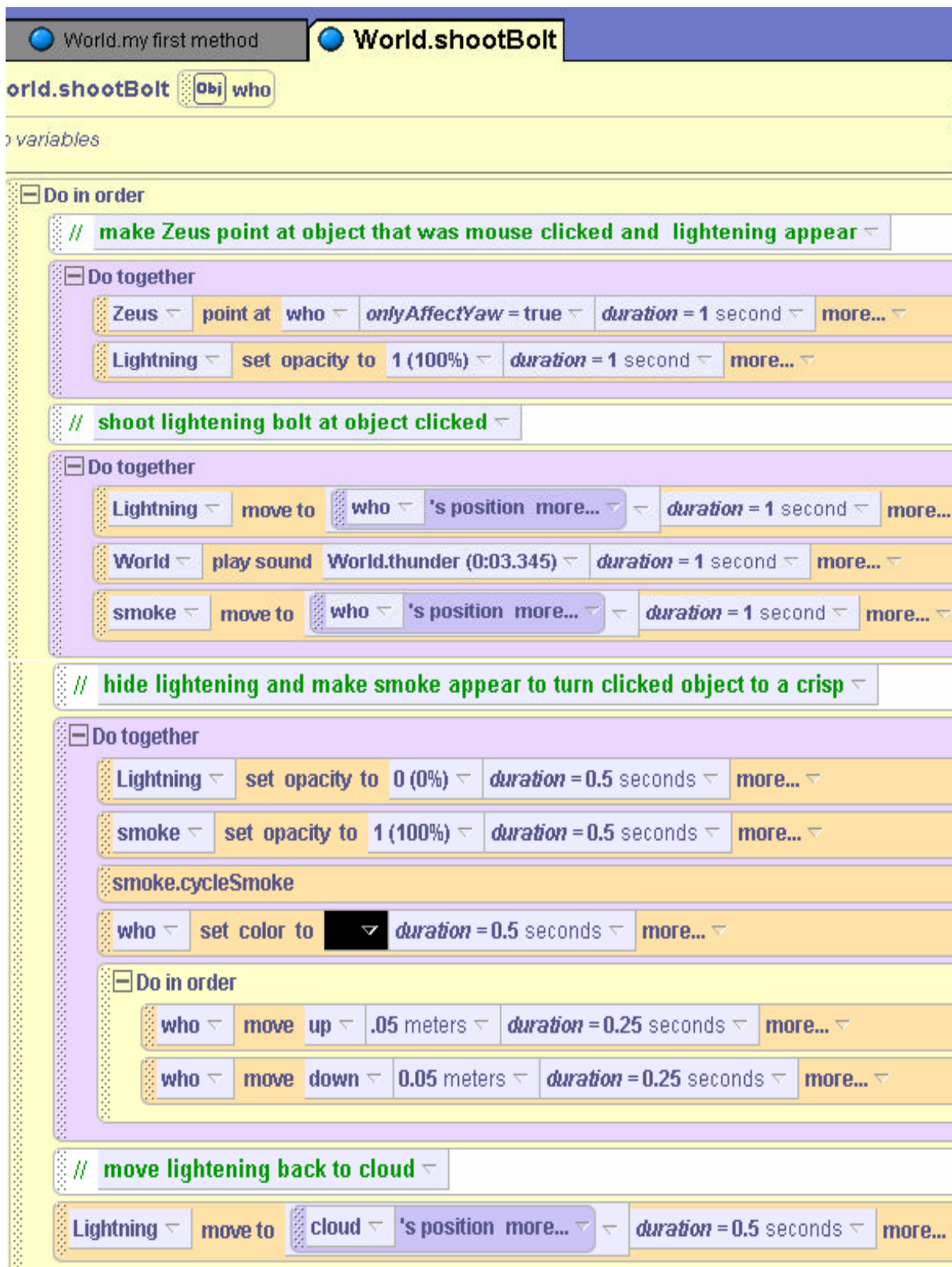


Figure 5-2-10. The *shootBolt* event-handler

Link the event to the event-handler

To link the mouse-click event to the *shootBolt* event-handler, it is only necessary to go through a similar set of steps as was done in previous interactive examples. Click on the *create new event* button in the Events editor, and select *when the mouse is clicked on something*. Then the *shootBolt* method is dragged into the link. Of course, *shootBolt* expects to be passed a parameter to identify the object that was clicked. Select *expressions* and then *object under mouse cursor*, as shown in Figure 5-2-11.

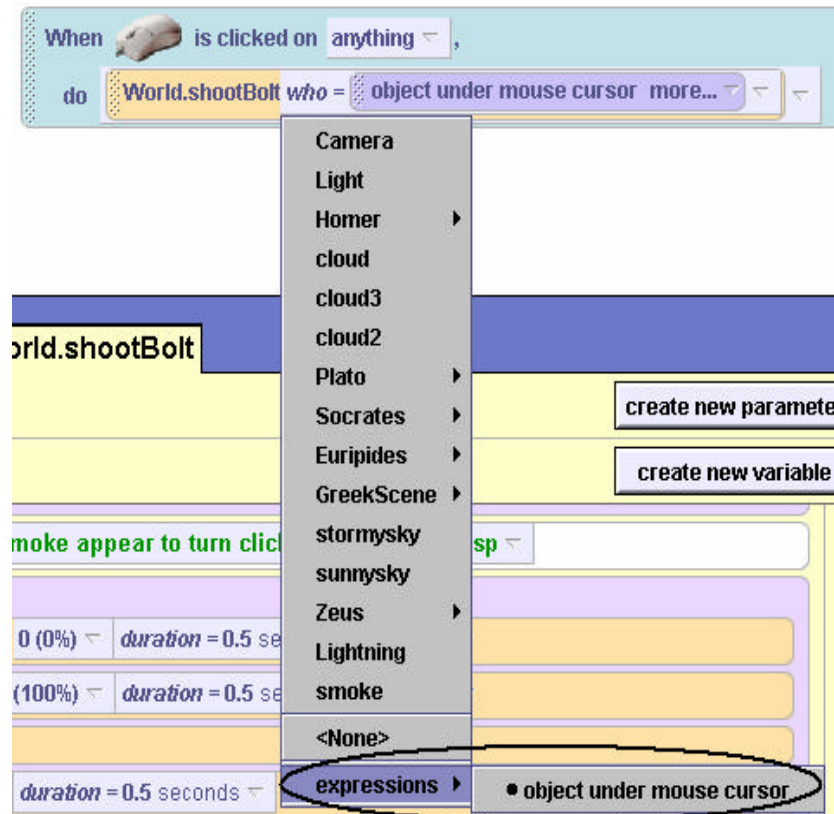


Figure 5-2-11. Pass object under mouse cursor as the parameter

Testing the program

The world is now complete. Of course, the program should be tested by running it and having Zeus shoot thunderbolts. When we tested this program, each of the philosophers was clicked on, to make sure the thunderbolt properly hit the target. But when we clicked on the clouds, the thunderbolt struck the clouds, turning them black. And when we clicked on the scene itself, the whole scene was turned black! This is not the behavior we wanted or expected. Another problem with the animation is that the user can click on an object that has already been zapped with lightning. A solution to these problems will be presented later.

Technical Notes

Several comments should be made concerning this example.

- Objects (such as the thunderbolt and the smoke) can be made to appear and disappear several different ways. One approach is to move an object below the ground and then

move it up when it is to appear. A second approach is to change the object's *opacity* to 0% (invisible) or 100%(visible). Yet another approach is to change the object's *isShowing* property. (See Tips & Techniques 3 for more details on opacity and *isShowing*.)

- The *cyclesmoke* method is a built-in method of the smoke object.
- One way to move an object (in this case, the lightening bolt and the smoke) to the position of another object is to use a *moveto* instruction. We followed the three-step approach for creating a *moveto* instruction, as was presented in Tips & Techniques 3: First, the Lightning object's *move to* instruction was dragged into the editor with the default Vector3(0,0,0) as the target.



Second, the built-in *Lightning's position* question was dragged over the Vector3(0,0,0) to replace it. (See Tips & Techniques 3 for details on built-in questions.)



Third, the *who* parameter tile was dragged into the position target to be a place-holder for any clicked object.



5-2 Exercises

1. CarouselGoRound

Create an amusement park scene with a carousel. In this animation, the carousel is to have at least four animals (horses, giraffes, or others of your choice). Add a two-way switch to the initial scene. Create two event-handler methods – one to make the carousel rotate clockwise and one to make it rotate counterclockwise. When the green button is clicked the carousel should rotate clockwise and a click on the red button should make the carousel rotate counterclockwise. Create a *storyboard* and then implement the carousel. *Optional*: add a sound clip that plays as the carousel goes around.

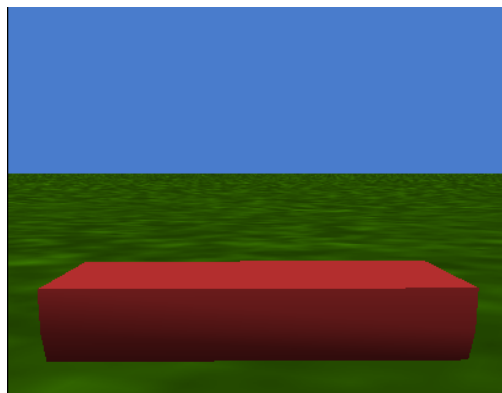
2. Snow Festival

Your team has created a snowman as the centerpiece of an entry in the Winter Snow Festival competition. To gain attraction to your snow festival display, you have set up colored spotlights that will turn the color of the snowman any one of 4 different colors. Create an initial world with 4 spotlights (spheres of 4 different colors) and a snowman, as shown below. Write *only one* method to change the color of the snowman. When the user clicks on a spotlight pass the color of the spotlight to the method and make the snowman change to be that color.

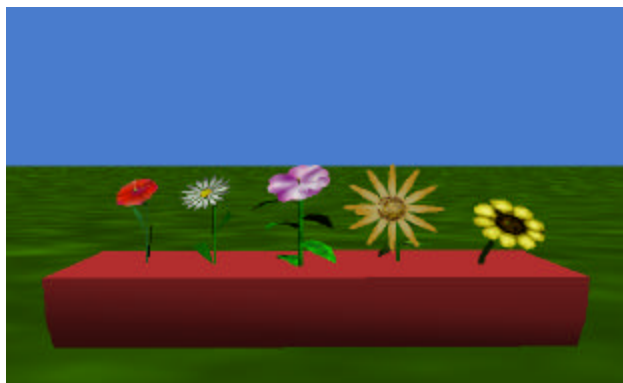


3. Flowerbox

It's spring and you are anxiously waiting for flowers to grow. You decide to give them a little help. Create an initial scene of a flowerbox (or a brown half cylinder, turned upside down) with 5 flowers (of your choosing) in it. (Use one-shot instructions to move the flowers down out of sight.)



Write *one method* to grow a flower in the box (move the flower up into view). The flower that grows in the box depends on which key the user presses on the keyboard. For example, if the user presses “S” key, the sunflower will grow but if the user presses the “D” key, the daisy will grow. To grow the flowers, create a “*When <key> is typed*” event for each key selected to represent a specific flower. Link the key-pressed event to the *growFlower* event-handler method using the particular flower represented by that key as its parameter. When all the flowers are grown, the flowerbox will look something like the following:



4. PenguinSlide. A favorite activity of penguins in the local zoo is to slide down an icy slope into a pool of water in the pond. Create a world with a frozen lake scene and three penguins on the slope, as shown below. Make the program event-driven. Allow the user to click on the next penguin to slide down the slope into the pool of water. Each penguin slides on its back and spins around as it slides. Each penguin should spin a different number of times. When the penguin reaches the pond, move the penguin down 5 meters so it disappears below the water. Write only one event-handler method. When the penguin is mouse-clicked, pass the penguin object that was clicked and the number of times the penguin is to spin around as it slides down the slope. *Optional:* add a water-splash sound as the penguin hits the water.



5 Summary

This major focus of this chapter was the creation of interactive (event-driven) worlds. Creating worlds with events allowed us to build significantly more interesting worlds such as game-like animations and simulations. In many object-oriented programming languages, event-driven programming requires knowledge of design patterns such as Observer, Callback, or Model-View-Controller. In Alice, creating events and invoking event-handler methods that respond to events is quite simple. Many of the messy details of event-driven programming are handled by Alice.

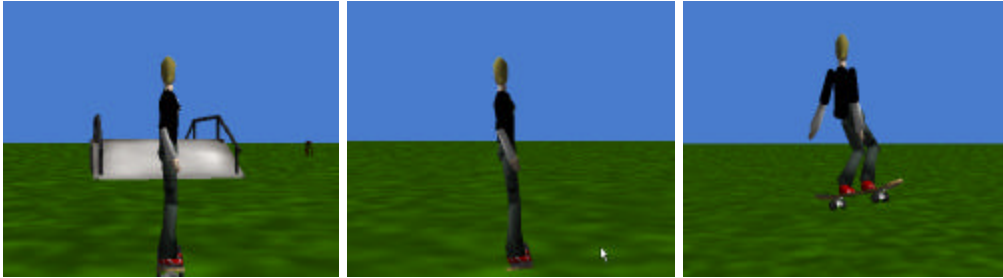
Important concepts in this chapter

- An *event* is something that happens.
- An event is created by user input (keyboard press or mouse click.)
- An event is linked to an *event-handler* method.
- Each time an event occurs, its corresponding event-handler method is invoked. This is what is meant by event-driven programming.
- The event-handler method contains instructions to carry out a response to the event.
- A parameter can be passed to an event-handler method when an event occurs.
- Parameters allow us to write one method that can handle several related events

5 Projects

1) SkaterWorld

The goal of this world is to simulate a skater girl and her moves. Create a world with a skater girl object on a skateboard. Add several some objects she can jump over. A sample scene is shown below.



The skater should have both *jump* and *spin* methods. Allow the user to press the up/down arrow keys to move the camera and skater forward/backward and the left/right arrow keys to make the skater lean left/right while the camera goes left/right. At the end of each motion, the skater should lean back to her original position. The **j** key can be used to make the skater jump and the **s** key to make her spin.

Hints:

- 1) If you are having problems with the skater girl moving where you want her to move, trying using *As seen by* camera or *As seen by* skateboard.
- 2) One way to make the camera follow the action is to position the camera behind the skater and make the skater a vehicle of the camera. A “while the world is running” event can be used to make the camera constantly move forward. This will allow the skater to skate forward on the ground while jumping and doing twists and the camera will continue to follow the action.

2. Skydiving Guidance System

Alice has taken up a new hobby: skydiving. She is on a helicopter, wearing a parachute. She is to jump to the carrier, which is a little ways in front of her. In the world shown below, we added a half cylinder inverted and connected to the helicopter to be used as a jump platform. And, a torus was used to create a harness for Alice (although this isn’t absolutely necessary, it is helpful due to Alice’s small waist compared to the parachute’s cords)



The idea of this animation is to provide a skydiving guidance system to allow the user to guide the movement of Alice as she jumps from the platform of the helicopter and glides down to the carrier. When the user thinks Alice has hit the top of the carrier, the user can press the enter key to have Alice drop her chute.

Guidance system methods (as smooth and lifelike as possible):

- 1) jump: which will allow Alice to jump from the helicopter's platform
- 2) glideUp, glideDown, glideRight, glideLeft: allows Alice to glide in the appropriate direction
- 3) swingLegs: allows Alice's legs to swing a bit when gliding or jumping
- 4) dropChute: allows Alice to get rid of her parachute (to simplify things, just have the chute rotate as if it was falling and make it disappear)

Keyboard controls:

- 1) Space bar -- Alice should jump off the platform.
- 2) Up/down/right/left arrow -- Alice should glide up/down/right/left.
- 3) When the enter key is pressed, Alice's parachute drops.

Remember that Alice should first jump off the platform prior to gliding and should not drop her chute until she hits the carrier.

3. Golfing. Alice has been taking golf lessons and she wants to demonstrate her new golfing skills to her friends. A fancy trick she has learned is a "ricochet." In a ricochet shot, the golf ball is hit toward an intermediate object and then bounces off in a different direction to go in a hole. Create a world with Alice, a sphere that will act as a golf ball, four circle objects that will be the holes in the ground, and four other small objects to act as intermediate objects for the ricochet shot. Arrange the four small objects so each one is ten meters away from Alice (at different angles from Alice). Also, arrange the holes so each small intermediate object is 10 meters away from one of the holes. The images below show a possible initial scene.



To animate the ricochet shot, create a *ricochet* method that takes two object parameters, *intermediateObject* and *whichHole*. When the ricochet method is called, Alice should swing, the sphere should travel ten meters and then bounce off the *intermediateObject*, changing direction so as to travel another ten meters into *whichHole*. (As *SeenBy* can be used to make the ball travel where it should.)

Create links in the events editor so that a user click on one of the four intermediate objects calls the ricochet method with the name of the object and the appropriate hole where the golf ball will fall in the cup. When Alice makes a ricochet shot, have her wave her hands in the air in triumph.

4. Turtle Motion Control

In this project, you are to create a turtle motion controller to help the turtle perform exercises for his upcoming race with the hare. Create a world which contains only a turtle and then create motion control methods for the turtle:

- a) headBob: allows the turtle's head to bob a little
- b) tailWag: allows the turtle's tail to wag
- c) oneStep: allows the turtle to move forward one step; his legs should move while he is taking that one step
- d) walkForward: combines the above 3 methods, to make a realistic step; all movements should take the same amount of time and should occur at the same time
- e) turnAround: turns the turtle 180 degrees; he should be walking while turning around
- f) turnLeft, turnRight: turns the turtle left/right, walking while he is turning
- g) hide: allows the turtle to hide in his shell (you may assume that the turtle is currently outside of his shell); remember to not leave the shell hanging in midair
- h) reappear: allows the turtle to reappear from his shell (you may assume that the turtle is currently hidden)
- i) talk: has the turtle look at the camera and say "hello" (or something different, if you wish) to the user

Create keyboard controls:

- a) When the up arrow key is pressed, the turtle is to walk forward.
- b) When the down arrow key is pressed, the turtle is to turn around.
- c) When the left arrow key is pressed, the turtle is to turn left.
- d) When the right arrow key is pressed, the turtle is to turn right.
- e) When the letter "H" is pressed, the turtle is to hide in his shell.
- f) When the letter "R" is pressed, the turtle is to reappear from his shell.
- g) When the letter "T" is pressed, the turtle is to talk to the user.

Test the turtle motion control system by running your world and trying all the interactions at least once. Be sure to only hide the turtle when he is already out of his shell and only have him reappear when he is hiding.

Tips & Techniques 5

Let the mouse move objects

The interactive worlds introduced up to now have used events where the user mouse-clicks on some object or uses key presses to control the motion of some object. But, some interactive worlds work best if the user is allowed to drag objects around with the mouse. When you are setting up a new world, you can move objects around the scene with the mouse. But, how do we make that option available to the user while the program is running? As an example, suppose a world where the user will rearrange the furniture in a room, as in Figure T-5-1. (We removed the walls so as to make this for this example.)



Figure T-5-1. Furniture arrangement

To allow the user to move the furniture around, create a "Let the mouse move objects" event in the event editor, as shown in Figure T-5-2. The result is shown in Figure T-5-3.

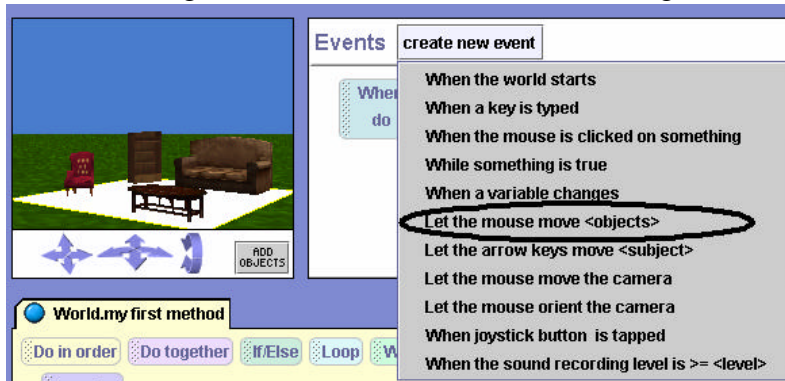


Figure T-5-2. Selecting *Let the mouse move objects* event



Figure T-5-3. Resulting event

Billboards

While Alice is a 3D programming environment, it is possible to display flat 2D images in a scene. Flat 2D images can be created in any paint tool and saved in GIF, JPG, or TIF format. To add the 2D image (Alice calls it a **billboard**) to your world, select Make Billboard from the File menu, as seen in Figure T-5-4. In the selection dialog box, navigate to the stored image and then click

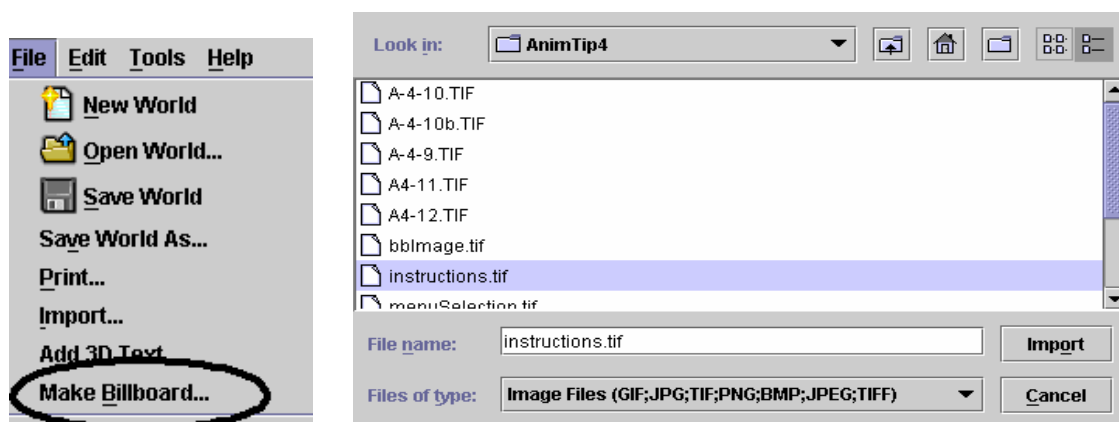


Figure T-5-4. Making a billboard

Alice will add the flat image to the world, as seen in Figure T-5-4.



Figure T-5-4. Billboard added to the scene

The billboard used in this example illustrates one of the uses of billboards -- providing information to the user about how to play a game or simulation. In the example shown in Figure T-5-5, the billboard provides instructions for using keys from the keyboard to steer a biplane. To make the billboard easy to read, we resized the billboard and used it to cover the scene, Figure T-5-5. Then, we set up an event so the user can click on the billboard and a method will be called that sets the opacity to 0%, causing the billboard to fade away to the opening scene, as seen in Figure T-5-6.

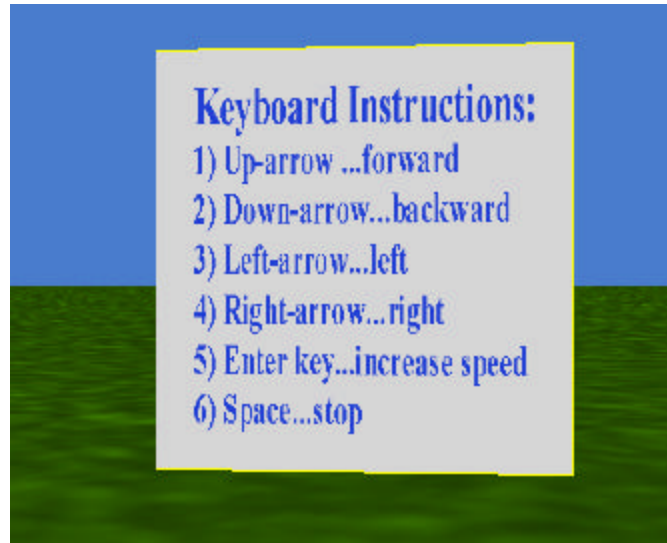


Figure T-5-5. Billboard provides instructions

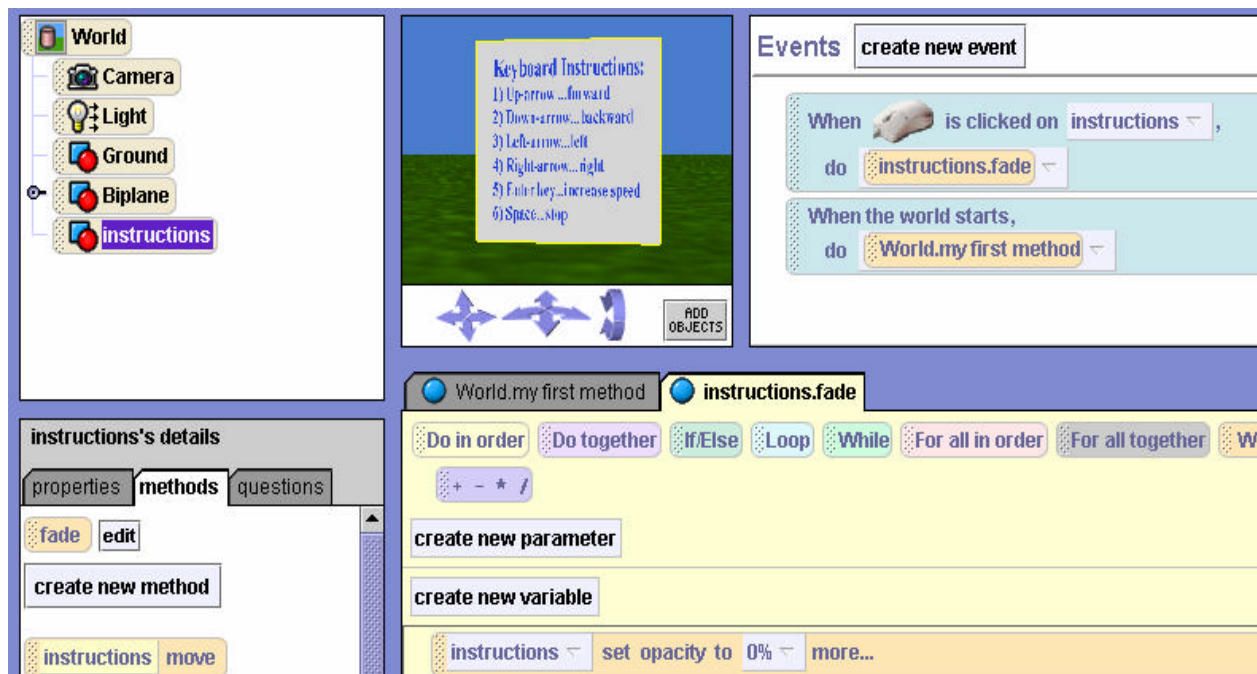


Figure T-5-6. When clicked, billboard fades away

Special Effect: fog

The examples above illustrate a number of character-level properties and built-in questions. Let's look at a world-level property. Consider the scene in Figure T-5-7. A knight is searching for a dragon in a forest. We would like to give the impression that the dragon is hiding from the knight. In most stories involving dragons, it is dreary and gray out. Some sort of fog would to make the knight's job (of finding the dragon) much harder.



Figure T-5-7. No fog

To add fog, click on the *World* object in the object-tree and select properties, as shown in Figure T-5-8. Then, click on the image to the right of *fogStyle* and select *density*. Density refers to the thickness of the fog. To adjust the fog density, click the image to the right of *fogDensity* and adjust the density value to achieve the desired effect. The larger the density value, the thicker the fog.

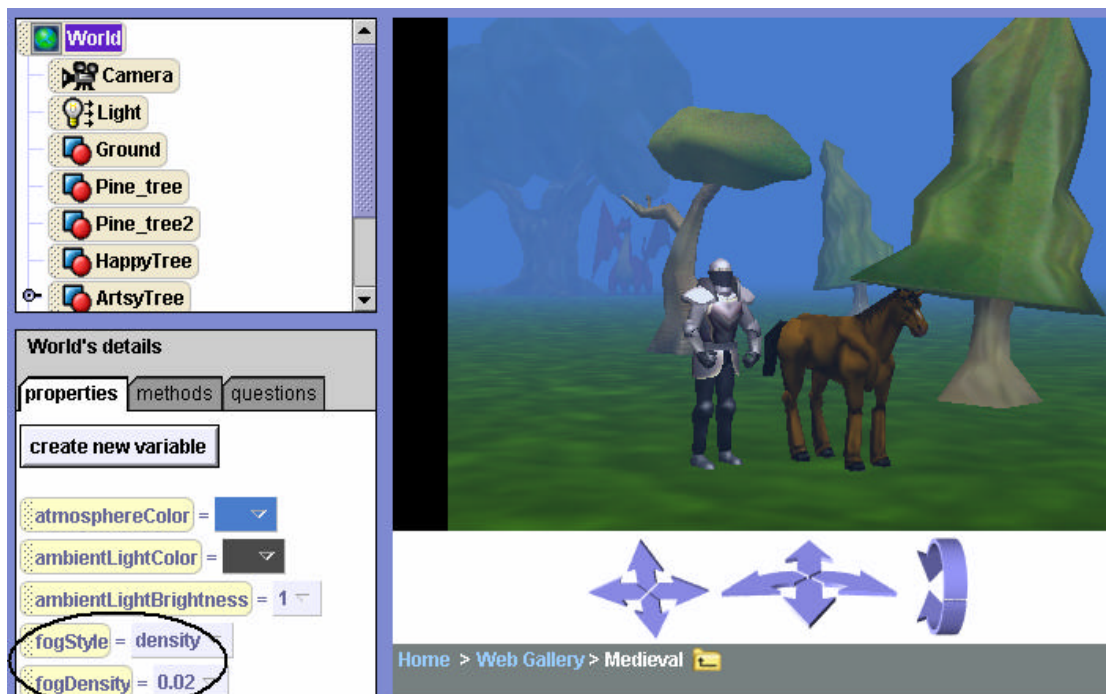


Figure T-5-8. fogStyle and fogDensity modified

Rotating around an invisible object

Consider the world illustrated in figure T-5-9. Suppose we wanted the pterodactyl to fly around the dragon.

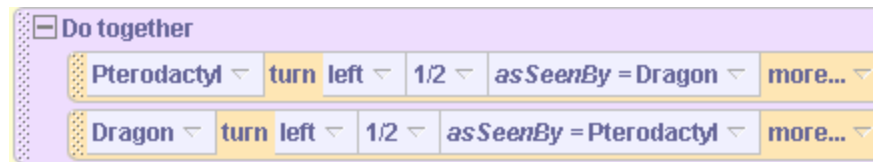


Figure T-5-9. A dragon and a pterodactyl

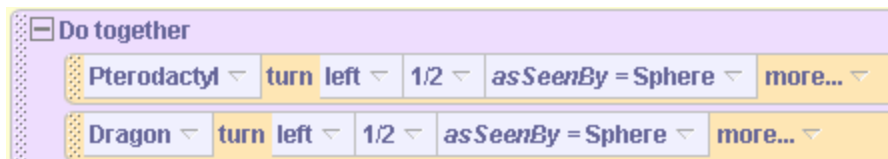
The pterodactyl will fly around the dragon if we use “as seen by = Dragon” in a turn left instruction for the pterodactyl object.



But suppose we wanted them to each make a half-turn relative to each other, so that the dragon ended up at the pterodactyl’s location (facing the opposite direction as it will have made half a circle), and the pterodactyl will end up at the dragon’s location (facing the opposite direction). A first attempt might be



But when this program is run, each animal ends up where it started, facing the opposite direction! The problem is that once each animal has begun to move, its location has changed, so that further moves relative to each other lead to unexpected results! What we need is a stationary object, located somewhere in between the dragon and the pterodactyl, which does not move. Let’s add a sphere object, placing it in between the dragon and the pterodactyl. Now, make the sphere invisible by changing its isShowing property to false. Now, we can write the following code:



When this code is run, the dragon and pterodactyl change places, as seen in Figure T-5-10!



Figure T-5-10. Pterodactyl and dragon change places

6 Decisions and User-defined Questions

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

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

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

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

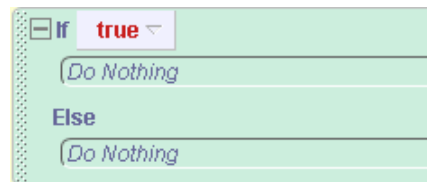
6-1 Decisions and Logical Questions

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

Decisions

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

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



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

If/Else Example

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

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

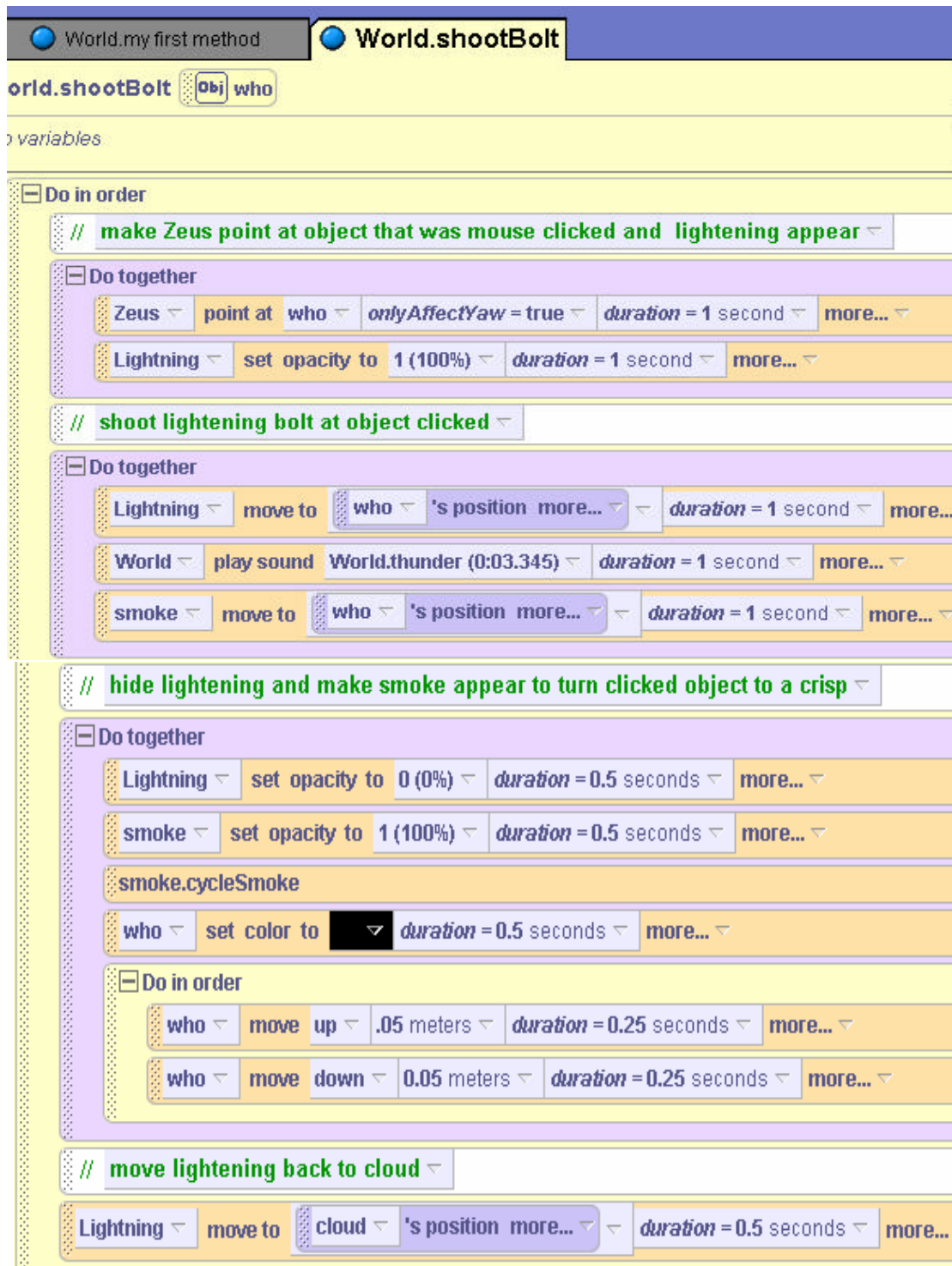


Figure 6-1-1. The *shootBolt* method (from Zeus world of section 5-2)

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

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

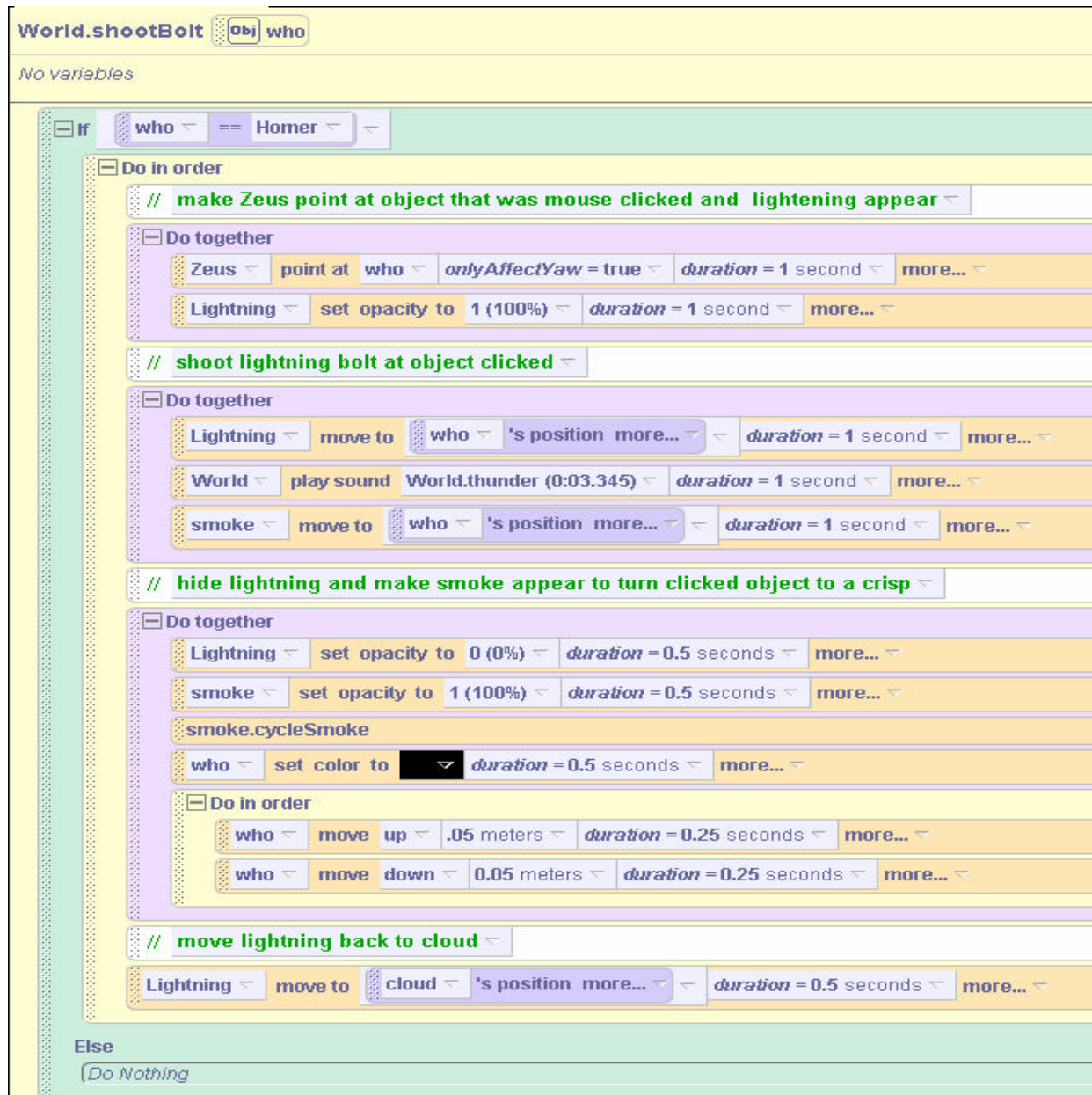


Figure 6-1-2. Shooting the bolt only at Homer

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

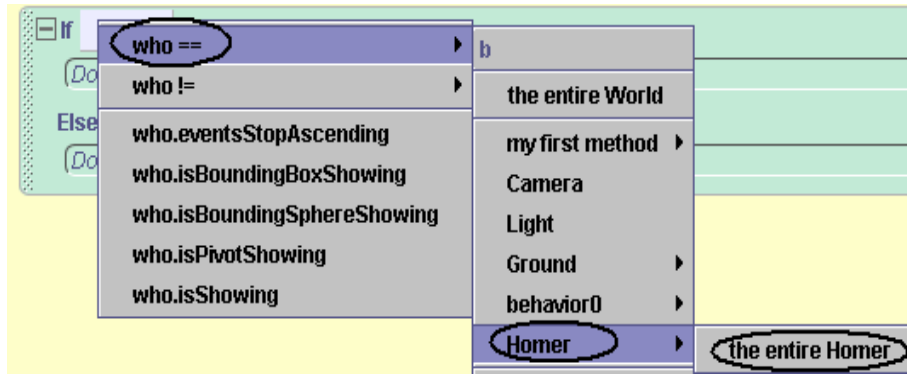


Figure 6-1-3. Selecting who == question

Logical Operators

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

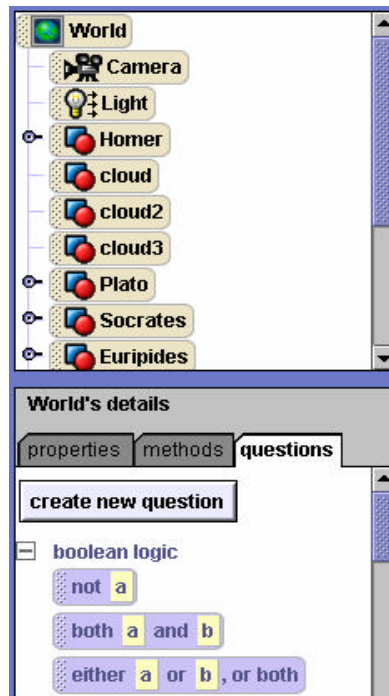


Figure 6-1-4. Logical operators

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

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

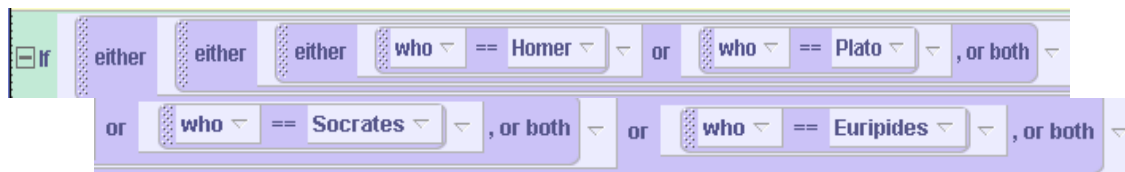


Figure 6-1-5. Multiple conditions in an *If* statement

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



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

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

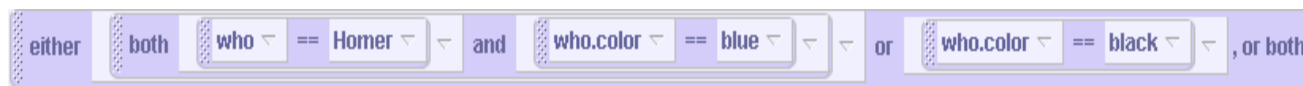


evaluates to *true* only if both the object clicked on is Homer, and the object clicked on has its color set to blue.

It is important to be very careful with expressions containing two or more logical operators. The following expression evaluates to *true* only if the object clicked on is Homer **and** Homer's color is black **or** blue.



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



These examples point out that levels of nesting in logical expressions can be tricky. In general, we recommend not including more than one logical operator in a Boolean expression. If more are needed, we recommend using nested *If* statements instead, as described in the next section.

World.shootBolt World.my first method

World.shootBolt [Obj] who create new parameter create new variable

No variables

If either either either who == Homer or who == Plato , or both or who == Socrates , or both or who == Euripides , or both

If who.color != black

Do in order

// make Zeus point at object that was mouse clicked and lightening appear

Do together

Zeus point at who onlyAffectYaw = true duration = 1 second more...

Lightning set opacity to 1 (100%) duration = 1 second more...

// shoot lightning bolt at object clicked

Do together

Lightning move to who 's position more... duration = 1 second more...

World play sound World.thunder (0:03.345) duration = 1 second more...

smoke move to who 's position more... duration = 1 second more...

// hide lightning and make smoke appear to turn clicked object to a crisp

Do together

Lightning set opacity to 0 (0%) duration = 0.5 seconds more...

smoke set opacity to 1 (100%) duration = 0.5 seconds more...

smoke.cycleSmoke

who set color to duration = 0.5 seconds more...

Do in order

who move up .05 meters duration = 0.25 seconds more...

who move down 0.05 meters duration = 0.25 seconds more...

// move lightning back to cloud

Lightning move to cloud 's position more... duration = 0.5 seconds more...

Else

Zeus say That philosopher is already fried!!! more...

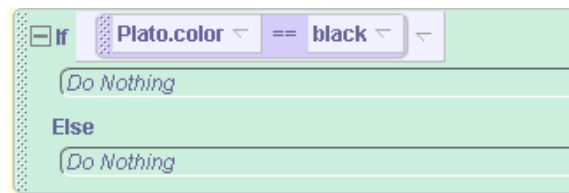
Else

Zeus say I only shoot at philosophers! more...

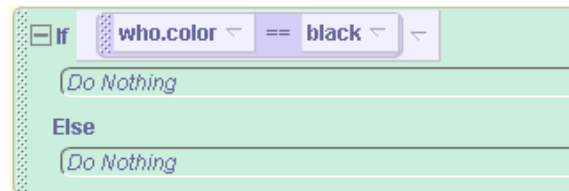
Figure 6-1-6. The complete code for the *shootBolt* method

Nesting If statements

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



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



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

Else

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

Relational operators

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

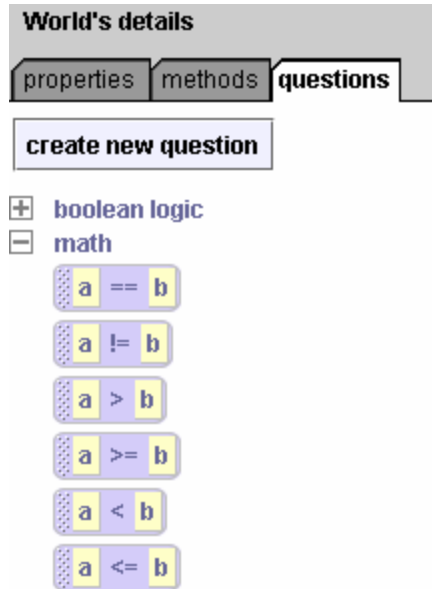


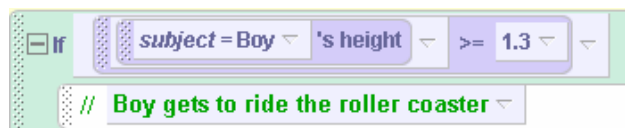
Figure 6-1-7. Relational operators

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

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



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



6-1 Exercises

1. Modifications to the Zeus world

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

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

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



2. More modifications to the Zeus world

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

3. PracticeTurns

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

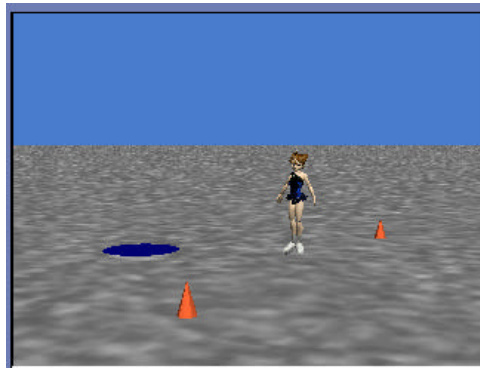


In this world, the skater is practicing turns on the ice. The skater will point at the cone and then skate forward toward the cone (a sliding step on one leg and then a sliding step with the other leg). When she gets close to a cone, she skates half way around the cone and ends up facing the other way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

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

6. IceDanger

For this exercise, you can begin with a world constructed in either exercise 4 or 5 above – or create a new skater world from scratch. Add a hole in the ice (a blue circle).



Make the world interactive to allow the user to use the mouse to move the hole around on the icy surface. (See Tips & Techniques for using the *let mouse move objects* event.) Now, as the skater is moving across the surface of the ice, the user can move the hole into the skater’s path. Modify your method that skates the skater forward to use an If statement that checks whether the skater is skating over the hole. If she is on top of the hole, she will drop through the hole. If you have sound on your computer, you may want to add a splash sound that plays when the skater falls through the ice.

7. TallTrees

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



6- 2 User-defined Questions I (Boolean)

Introduction to Questions

Alice uses the term *question* to refer to a program construct known in many other programming languages as a *function*. A question (function) may receive values sent in to parameters (input), perform some computation on the values, and return (send back) a value as output. **In some cases, no input is needed – but, generally values are sent in.** The diagram in Figure 6-2-1 outlines the overall mechanism. One way of thinking about a question or function is that it is something like an old fashioned jukebox. You put coins in the machine and select a song. The machine loads the recording and sends the music out through speakers for you to enjoy.

Actually, you have been using functions all your life, in many cases never thinking about it. For example, a cash register at a supermarket acts as a form of function. The cashier enters (as input) the prices of each of the items you are going to purchase, the cash register computes a sum and adds on the tax, and the cash register returns (as output) the total cost of all the items.

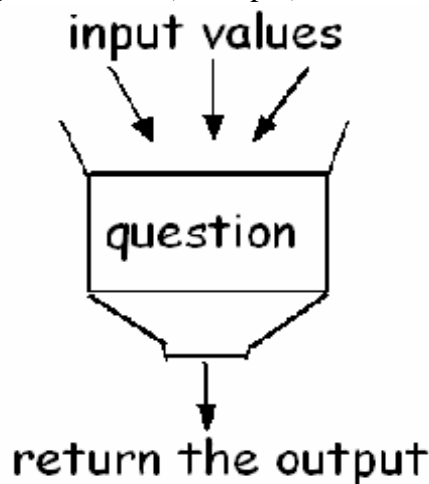


Figure 6-2-1. The functionality of a question

Abstraction

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

User-defined questions

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

Creating a new Question

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



Figure 6-2-2. Create new question tile

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

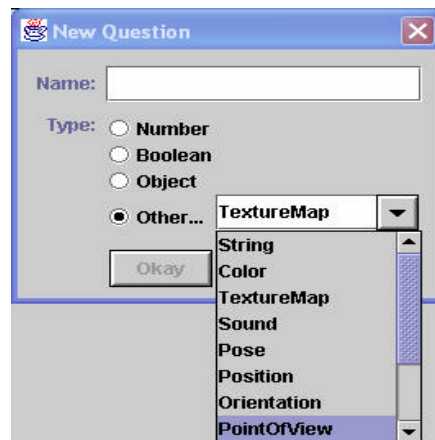


Figure 6-2-3. Types of user-defined questions

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

Writing a simple Boolean question

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

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



Figure 6-2-4. An initial pond scene

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

Parameters: comparingObject, FirstObject, secondObject

If comparingObject's distance to firstObject is less than
comparingObject's distance to secondObject
 return true

Else
 return false

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

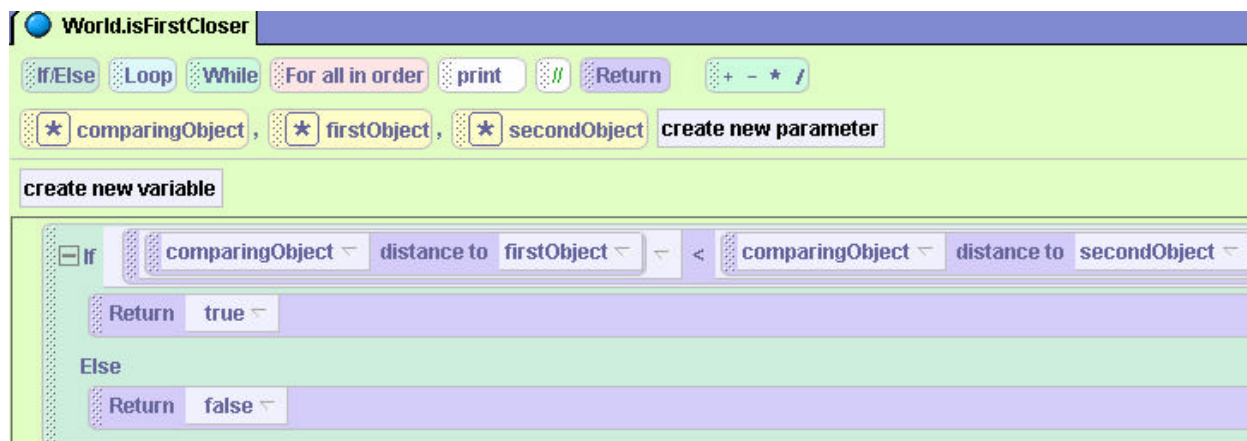


Figure 6-2-5. A user-defined question to return true if the first object is closer

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

Calling the question

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

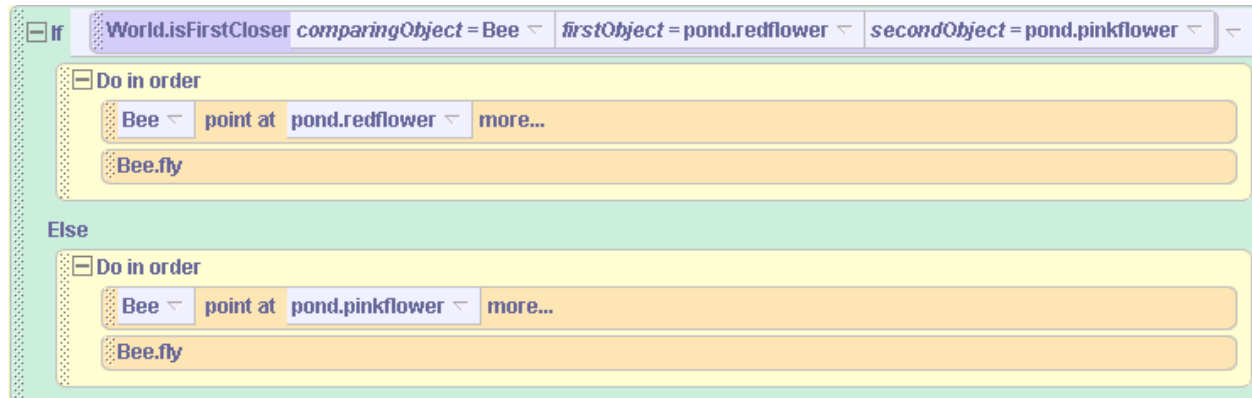


Figure 6-2-6. Calling the user-defined question

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

A more complex Boolean question

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

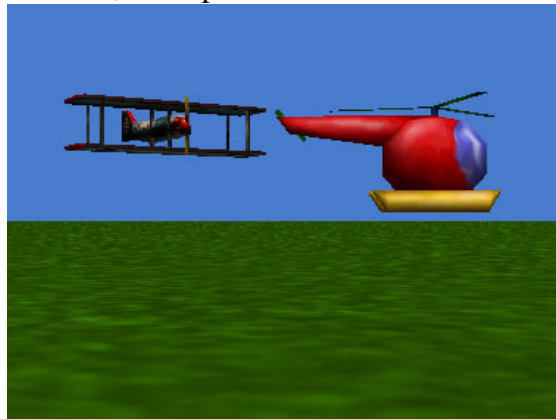


Figure 6-2-7. Flyspace collision danger

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

Parameters: firstObject, secondObject

If firstObject height is equal to secondObject height

 return true

Else do nothing

If firstObject is above the secondObject and its height above the secondObject < 10

return true

Else do nothing

If secondObject is above the firstObject and its height above the firstObject < 10

 return true

Else do nothing

return false

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

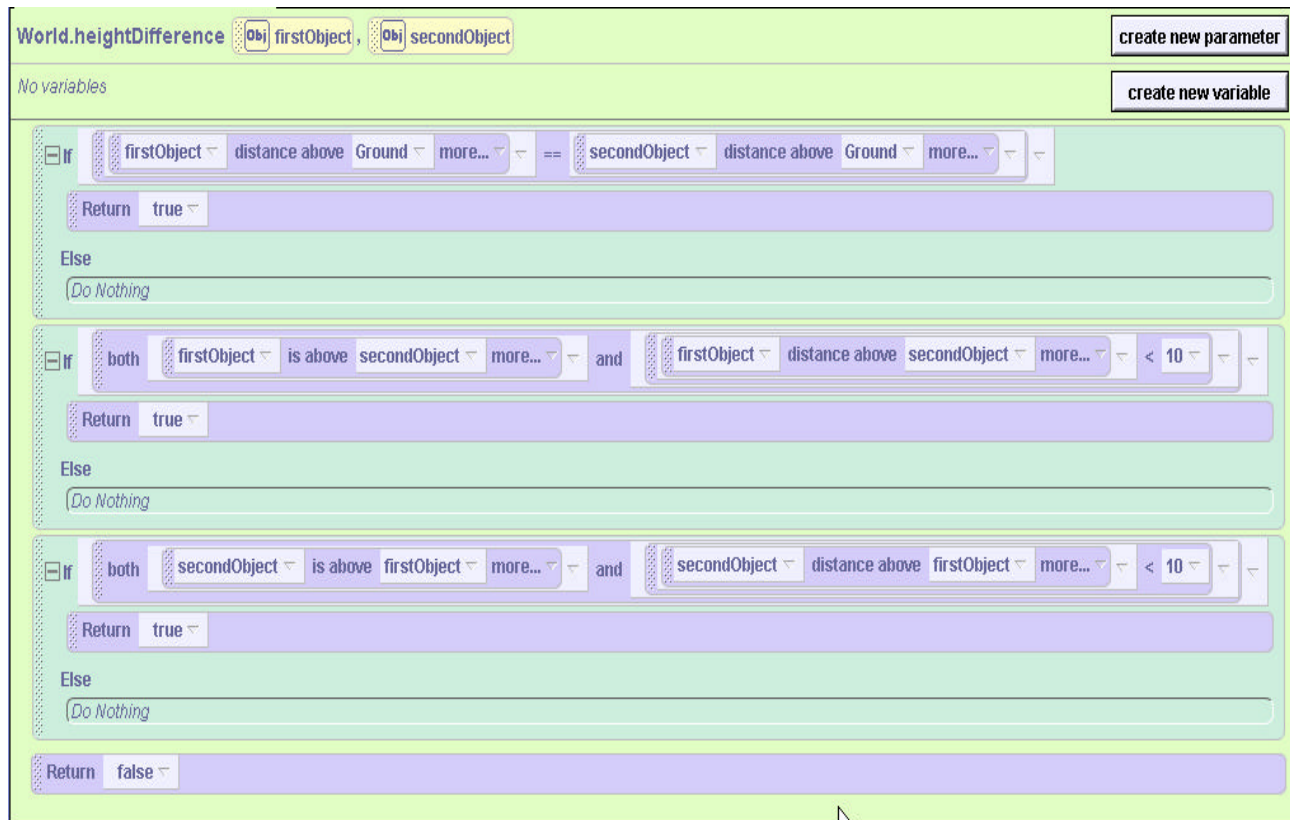


Figure 6-2-8. A question with multiple If statements for multiple conditions

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

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

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

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

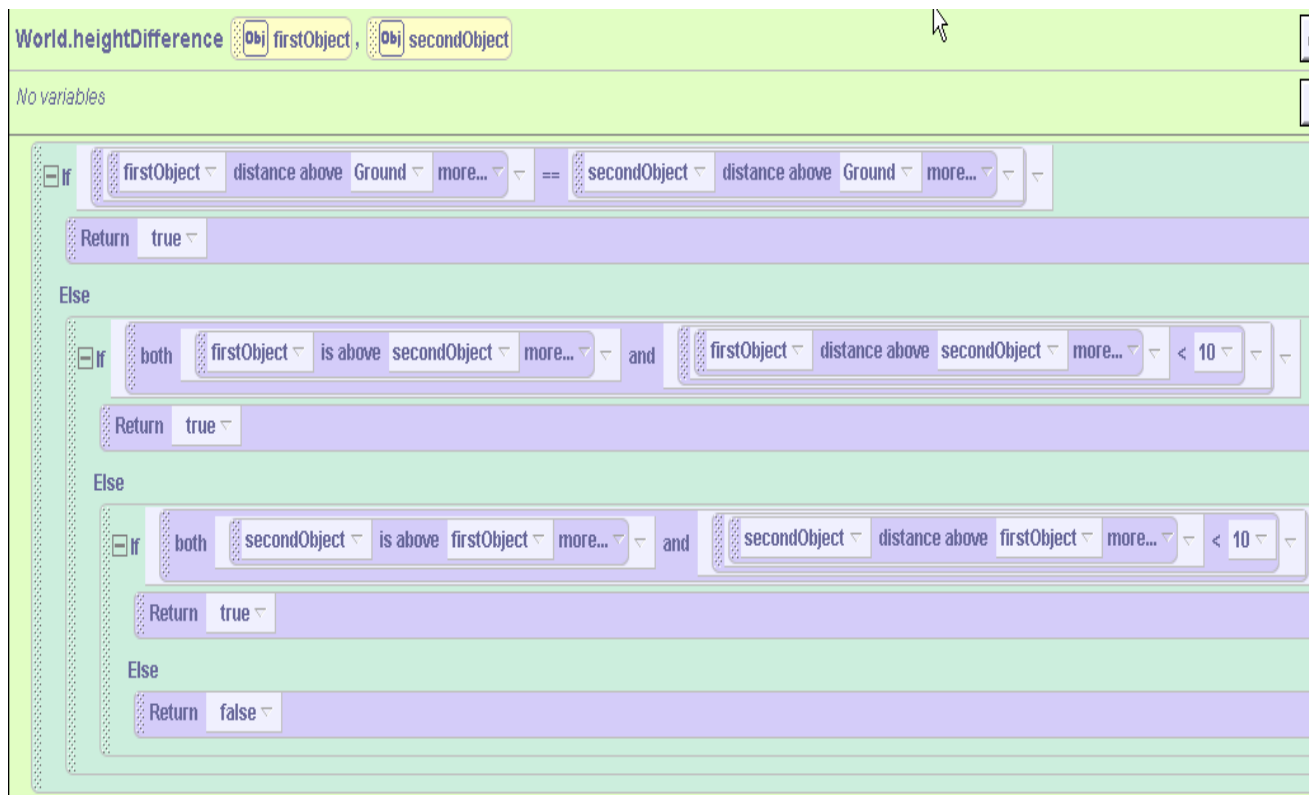


Figure 6-2-9. Cascading style – nested *If* statements

6-2 Exercises

1. Creating a question for the Zeus world

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

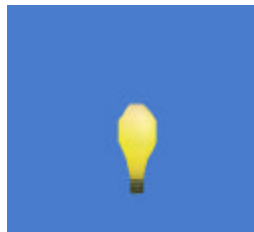
2. Switch

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



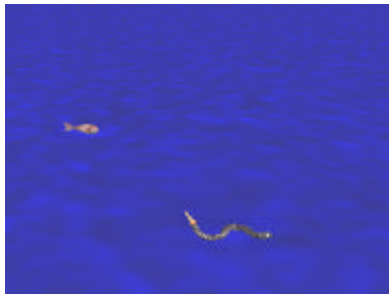
3. LightBulb

Create a world with a *lightbulb* and a method *TurnOnOff* that turns the *lightbulb* on/off depending on whether it is already on/off. When the lightbulb is on, its emissive color property has a value of yellow. When the light bulb is off, its emissive color is black. Write a Boolean user-defined question *IsLightOn* that returns *true* if the light bulb is on and *false* if it is off. When clicked, the *lightbulb* should turn on/off.



4. SnakeOnTheMenu

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



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

5. FrightenAwayTheDragon

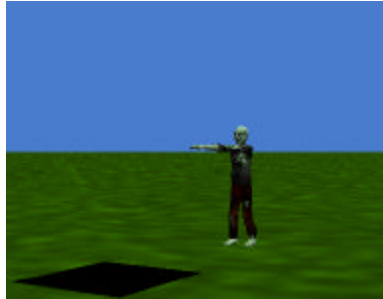
Create an initial scene of a troll and a dragon as shown below. The troll is trying to frighten away the dragon from his favorite hunting grounds. But, the troll is smart enough to not get too close to the dragon. The troll is to rant and rave while moving toward the dragon if the two are more than 5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

pressed. Use a user-defined question to find out when the troll gets too close to the dragon. When the troll is less than 5 meters away from the dragon, have the dragon fly away.



6. ZombieWorld

Create a world with a zombie and an open grave (a black square on the ground). In a scene from a scary movie, the zombie walks forward toward the grave and falls in. In this animation, every time the user presses the space bar, the zombie should walk forward. A Boolean question named *aboveGrave* that returns true if the zombie is within $\frac{1}{2}$ meter of the grave. When the question returns true, make the zombie fall in.



6-3 User-defined Questions II (Number)

Other types of questions

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

User-defined number question

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

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

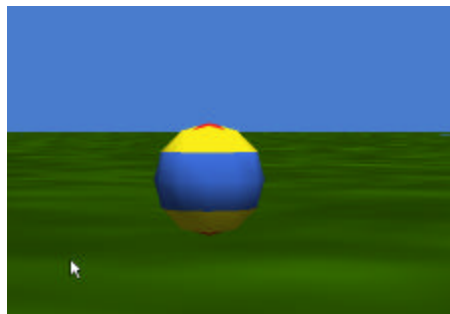


Figure 6-3-2. A toy ball

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



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

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

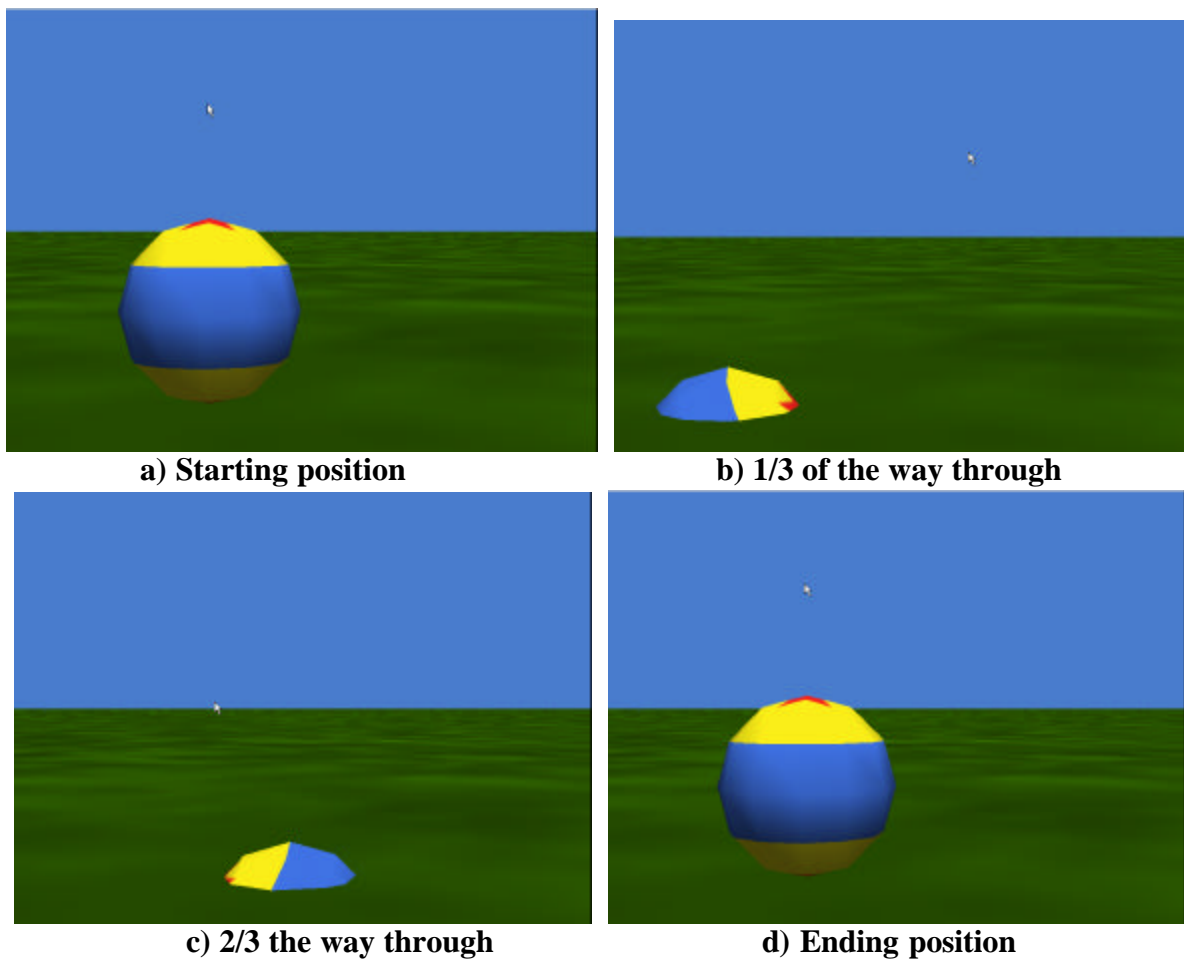


Figure 6-3-4. Move and turn together sequence of snapshots

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



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

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

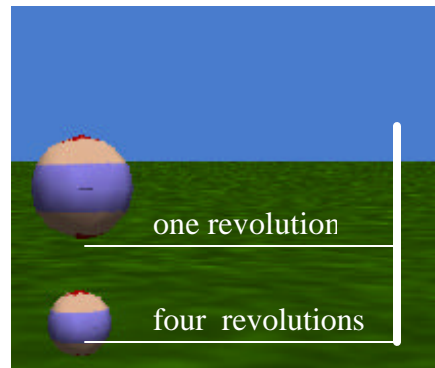


Figure 6-3-6. Distance covered by a revolution is proportional to diameter

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

$$\text{number of revolutions} = \text{distance} / (\text{diameter} * P)$$

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

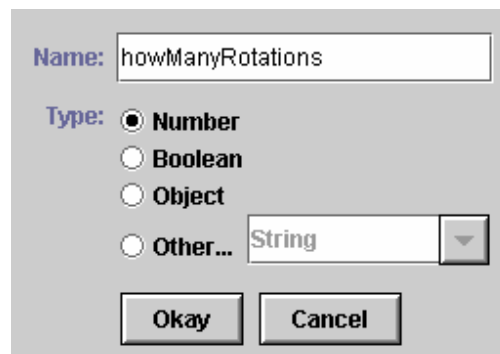


Figure 6-3-7. Naming and selecting the type for a number question

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

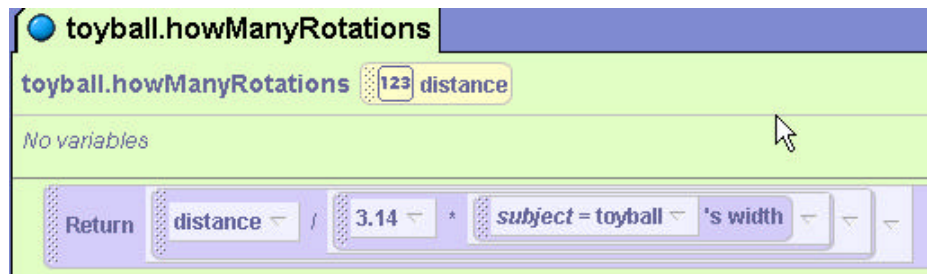


Figure 6-3-8. *toyball.howManyRotations* question

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

Testing

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

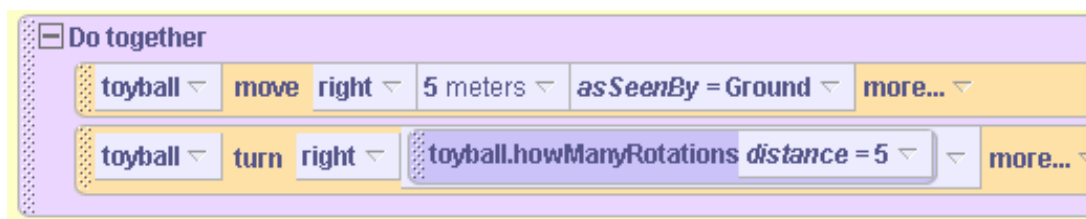


Figure 6-3-9. Testing *howManyRotations* Question

Using a question with a relational operator

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

toyball.howManyRotations as part of a logical expression. The *less than* relational operator is used to compare the value returned by *howManyRotations* to one. The result of this comparison is true or false and determines whether the toyball makes 1 revolution or several revolutions.

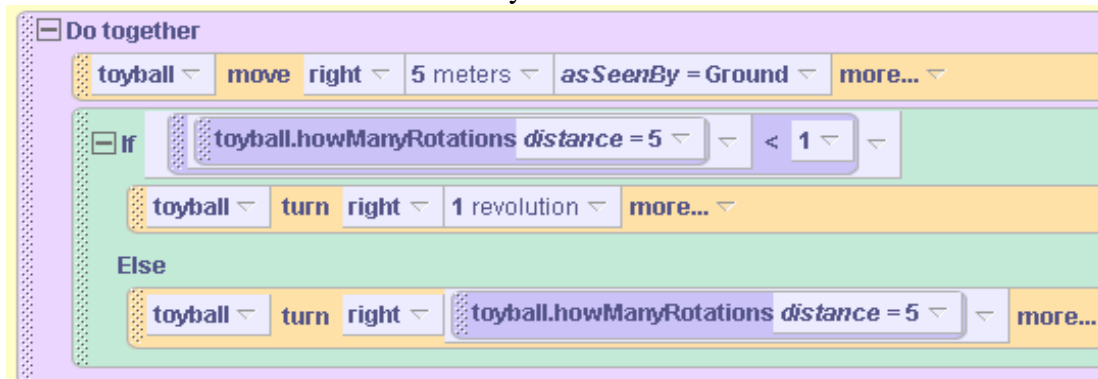


Figure 6-3-10. Calling a question as part of a logical expression

Abstracting a character-level method

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

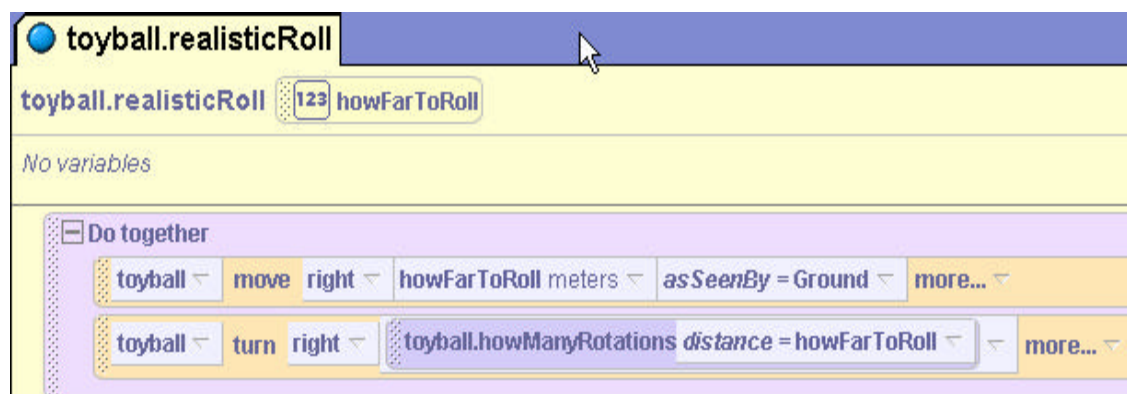


Figure 6-3-11. Character-level method *toyball.realisticRoll*

World-level question for generic use

In the toyball example, a character-level question and a character-level method were developed. But, it is clear that the question to compute the number of rotations for a ball might be used on any kind of spherically-shaped object. This brings up one very good reason for writing a world-level question: a question that could be asked of any object having similar properties. To write a generic kind of question similar to *howManyRotations* for any spherically-shaped object, two

parameters are needed: the *distance to roll* the spherical object and the *diameter of the sphere*. The reason for distance to roll is the same as it was for the *toyball.howManyRotations* method. But, the reason for the diameter parameter is that this method is intended to be generic (work for any spherical object). So, the diameter of the spherical object must be sent in for the computation of the number of times the sphere is to rotate. Otherwise, Alice would not know which spherical object's diameter to use for the computation.

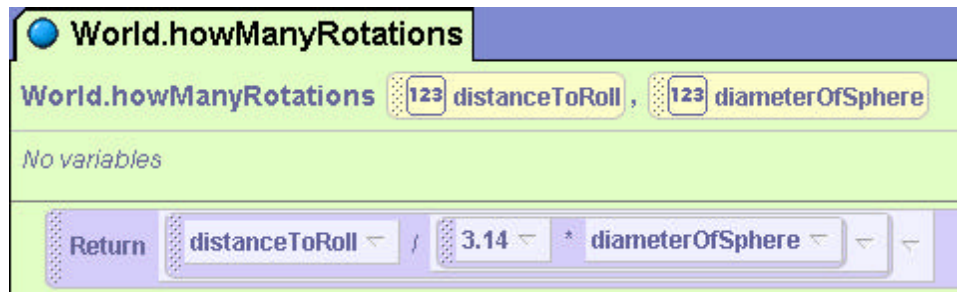
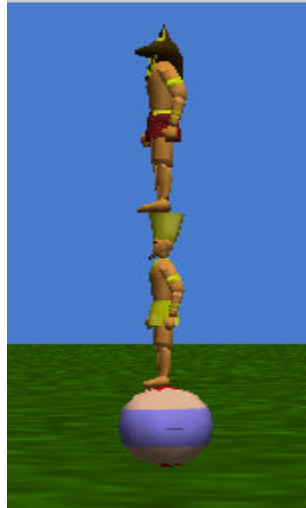


Figure 6-3-11. Generic world-level question

6-3 Exercises

1. AcrobatsWithRollingBall

Begin by creating a world containing the *toyball* (resized to twice its original size) and two characters/acrobats of your own choosing. Position the acrobats on top of the ball. **Use the scene editor quad view to be certain the acrobats are standing directly on top of one another and are centered on the ball. Also, use a one-shot instruction on each acrobat and on the ball to orient each object to the center of the world.** (See Tips & Techniques 3. Using *orient to* will ensure the objects are synchronized for movement together.)



Create an animated circus act, where the acrobats move with the ball, staying on top of it, as the ball rolls. The acrobats should put their arms up half way, to help them to balance!

2. BeeScout

This exercise is a variation on the bee scout animation presented in section 6-2. It has been a hot, dry summer and a hive of bees is in desperate need of a new supply of pollen. One bee has ventured far from the hive to scout for new pollen sources. A natural place to look is near ponds in the area. Set up the initial scene with a circle flat on the ground and colored blue to look like a pond. Add plants, trees, and other natural scenery including some flowers. Be sure the bee is located somewhere around the edge of the pond, as shown in the screen shot below.



Write a program to animate the bee scouting the edge of a pond for flowers that are in bloom. The bee is to fly around the perimeter of the pond (circle). Write a method to make the bee scout around the perimeter of the pond in which the circumference of the circle is used to guide the motion. (Yes, *asSeenBy* could be used – but that is not the point of this exercise.) The formula for computing the circumference of a circle is $\text{PI} * \text{the diameter of the circle}$. PI is 3.14 and the diameter is the object's width. Write a question that computes and returns the circumference of the circle. Then, have the bee fly around the perimeter of the pond by moving forward the amount of meters returned by the circumference question while turning left one revolution.

3. PyramidClimb

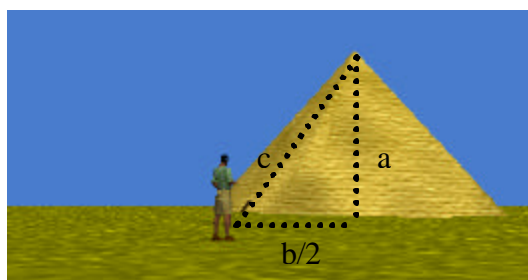
On spring break, a student is visiting the land of the Pharoahs. The student decides to climb one of the pyramids. He/She will start at the bottom and move straight up the side. Set up an initial scene consisting of a person and a pyramid, as shown in the screen shot below. Write a method to animate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



Prepare the person for climbing the pyramid by pointing the person at the pyramid and walking him/her up to edge. Then, turn the person about 1/8 of a revolution so as to lean into the climb. (Play with this leaning movement until you get a reasonable angle for the person to climb the pyramid.) While the person is climbing the pyramid have the person in a leaning position. After reaching the top, the person should stand up straight.

To determine how far the person must move to climb up the side of the pyramid, the *climb* method must call a question. The question computes the side length of the pyramid. The formula for computing the distance up the side of the pyramid is based on the Pythagorean theorem ($a^2 + b^2 = c^2$). Actually, the value that is needed is the value of c , which will provide a rough estimate of how far the person should move (in a diagonal direction) up the side of the pyramid. The formula is:

$$\text{length of the pyramid's side} = \sqrt{(\text{pyramid's height})^2 + (\text{pyramid's width}/2)^2}$$



6 Summary

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

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

Built-in questions do not always meet the particular needs of a program. In this chapter, we looked at how to write our own user-defined question that returns a Boolean value (true or false). Then, other types of user-defined questions were introduced. The benefit of writing our own user-defined questions is that we can think about the task on a higher plane – a form of abstraction. User-defined questions that compute and return a number value make writing code much cleaner because the computation is hidden in a question, rather than cluttering up the method where the result of the calculation is needed. By using parameters in user-defined questions, we can make the questions generic to use the questions with different kinds of objects. Character-level questions can be defined and saved with the object to allow us to reuse the question for that object in another program.

Important concepts in this chapter

- An *If* statement is a block of program code that allows for the conditional execution of that segment of code.
- An *If* statement contains a Boolean condition (logical expression) used to determine whether the segment of code will execute.
- If the Boolean condition evaluates to *true*, the *If* part of the statement is executed. If the expression evaluates to *false*, the *Else* part of the *If* statement gets executed.
- Boolean conditions may call built-in questions that return a true or false value.
- Logical operators (and, or, not) can be used to combine simple Boolean conditions into more complex expressions.
- Relational operators (< , > , >= , <= , =) can be used to compare values in a Boolean expression.
- User-defined questions can be written to return a Boolean value and used in *If* statements.
- User-defined questions can also be written to compute and return other types of values.

6 Projects

1. Gatekeeper

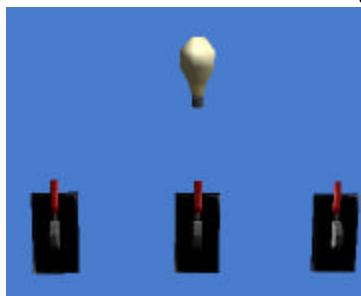
Build a world with any four different characters of your choice (people, shapes, vehicles, etc.). Position the four objects in a lineup. The characters are facing the player and are spaced equally apart from one another.



In this game, one of the characters is a gatekeeper, holding a secret password to allow the user to open a hidden door into the pyramid. To find the secret password, the user must rearrange the objects in the lineup until the gatekeeper is in the position on the far right of the lineup. (If the objects in the lineup are counted from left to right, 1 – 2 – 3 – 4, the gatekeeper must be moved into position 4.) When an object is clicked, it switches position with the character farthest from it, 1 and 4 will switch with each other if either one is clicked, 3 will switch with 1 if clicked, and 2 will switch with 4 if clicked). Make one of the objects (in position 1, 2, or 3) the gatekeeper. You must use a programmer-defined Boolean question that returns true when the objects in the lineup have been rearranged so the gatekeeper is on the far right of the lineup and false if it is not. When the gatekeeper is in position, display a 3D text object containing the password.

2. BinaryCodeGame

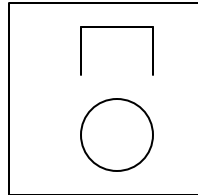
Build a world with three *switches* and a *lightbulb*, as seen below. Beneath the lever on each switch put an invisible sphere. Set the emissive color of the *lightbulb* to *black* (turned-off).



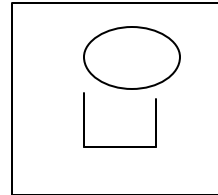
In this game, the positions of the levers on the switches represent a binary code. When a lever is up, the lever represents 1 (electric current in the switch is high) and when a lever is down, the lever represents 0 (electric current in the switch is low). In the above world, all three levers are up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

direction – up (if currently down) or down (if currently up). When all three switches are in the correct position for the binary code, the *lightbulb* will turn on.

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



Lever up, Sphere below Lever



Lever down, Sphere above Lever

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

3. DrivingTest

Create a world that simulates a driving test. The world should have a car, 5 cones, and a gate. Set up your world as shown in the image below. Also, create two 3D text phrase objects “You Pass” and “Try Again”. Set the *isShowing* property of each text phrase object to *false*, so that they are not visible in the initial scene.



In this driver test, the user will use arrow-key presses to move car forward, left, or right to swerve around each of the five cones so as to avoid hitting a cone. If the car hits one of the cones, the driver fails the test, the car stops moving and the "Try Again" text object is made visible. If the user manages to steer the car past all 5 cones, the car should drive through the gate and the "You Pass" text object made visible. Write a user-defined question named *tooClose* that checks the car's distance to a cone. If the car is within 2 meters of the cone, the question returns *true*. It returns *false* otherwise. Also, write a question named *passedTest* which evaluates if the user has passed the test. This is recognized by the fact that the car has been driven past the gate.

Hints: Work under the assumption that the user will not cheat (i.e. pass all the cones and head straight through the gate). Due to the differences in the width and depth of the car, do not be concerned if part of the front or back of the car hits a cone.

4. PhishyMove

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

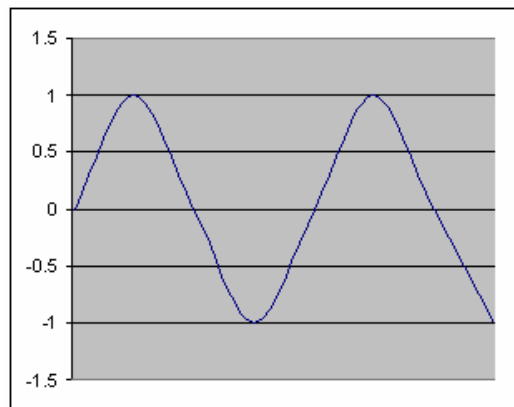


Note: This world is provided on the CD that accompanies this book. We recommend that you use the prepared world, as setting up the scene is time consuming. If you are an adventuresome soul, here are the instructions for setting up the world on your own: Use one-shot instructions to move the fish to the world origin (0,0,0) and then turn the fish right $\frac{1}{4}$ revolution. Because Phishy is partially submerged in the water, set the opacity of the water (ground) to 30% so the object can be seen in the water. Now, use camera controls to re-position the camera (this takes a bit of patience as the camera must be moved horizontally 180-degrees from its default position). Then, adjust the vertical angle to give a side view of Phishy in the water, as seen above. The fish should be located at the far left and the water should occupy the lower half of the world view, as seen in the screen shot above.

Alice has a *sine* question that can be used to teach Phishy the *sineWave* motion. (The sine question/function is often used to determine the relationship between the lengths of the sides of a right triangle. For the purposes of this animation, the relationship of the lengths of the sides of a right triangle is not really important.) If the sine function is computed over all angles in a full circle, the sine value starts at 0, and goes up to 1, back through 0 to -1 and returns to 0:

Angle	Sine of the angle
0	0
45	0.707
90	1
135	0.707
180	0
225	-0.707
270	-1
315	-0.707
360	0

This function is continuous, so if sine values are plotted some multiple of times, we will see the curve repeated over and over, like so:



Sine wave

For the *sineWave* motion, Phishy is to move in the sine wave pattern. In the world provided on the CD, Phishy has been positioned at the origin of the world. In a 2D system, we would say she is at point (0,0). To simulate the sine wave pattern, she needs to move up to height that is 1 meter above the water, then down to a depth of 1 meter below the surface of the water (-1 meter), back up to 1 meter above the water, and so on. Of course this up-and-down motion is occurring at the same time as she moves to the right in the water. The Alice sine question expects to receive an angle expressed in radians (rather than degrees). Write a question, named *degreesToRadians*, that will convert the angle in degrees to the angle in radians. To convert from degrees to radians, multiply the angle degrees by π and divide by 180. The *degreesToRadians* question should return the angle in radians.

Now, write a method to have Phishy move in the sine wave pattern. Remember that in the world provided on the CD, Phishy has already been positioned at the origin of the world. So, Phishy is already at the position for 0 degrees.

Hint: One way to create the sine wave pattern is to use *moveTo* instructions (see the Tips and Techniques section of this chapter). A *moveTo* instruction should move Phishy to a position that is (*right*, *up*, 0), where *right* is the radian-value and *up* is the $\sin(\text{radian-value})$. Use *moveTo* instructions for angles: 45, 90, 135, 180, 225, 270, 315, and 360. For a smoother animation, make each *moveTo* instruction have *style = abruptly*.

5. CosineWave

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

Tips & Techniques 6

If-Else and visibility as a condition

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

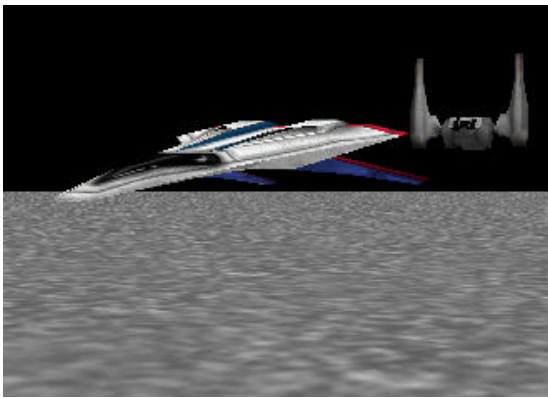
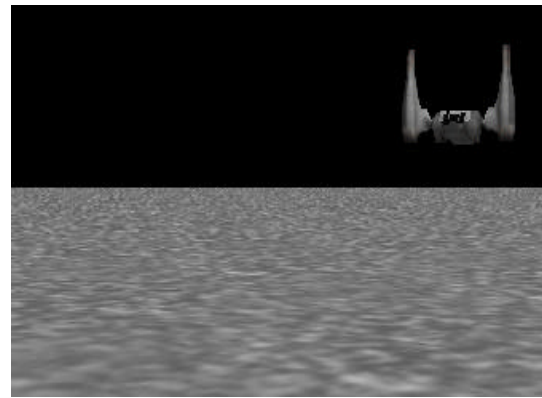


Figure T-6-1(a) Space scene before Cloaking

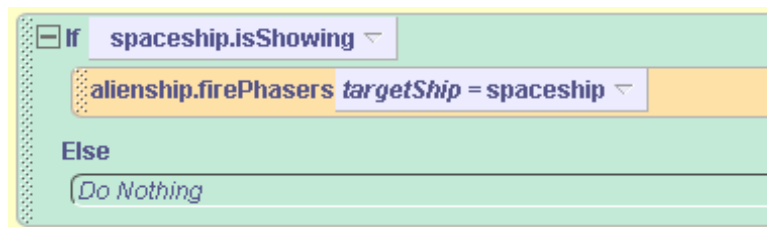


(b) Space scene after Cloaking

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



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



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

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

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

Random numbers

The use of random numbers plays a big role in certain kinds of computing applications. For example, random numbers are used for audit sampling, lottery selection, and encrypting computer files. An encrypting utility uses random numbers to create a code for scrambling the information stored in a file. To decrypt (unscramble) the information, you need to know the code that was used. Encryption has become an important part of internet-centric communications.

Various kinds of simulations also make use of random numbers. In Alice, random numbers can be used to create animations where motion is random (see next topic below). Random numbers are created by selecting the world-level question *random number*, circled in Figure T-6-2.

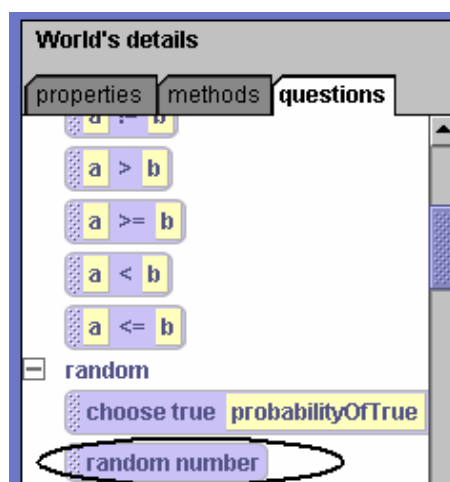


Figure T-6-2. World-level random number questions

As an example, consider the world scene in Figure T-6-3. An Eskimo child has adopted an abandoned penguin as a pet and they are playing together, sliding around on the ice.



Figure T-6-3. Pet penguin sliding on the ice

Suppose you want the penguin to slide forward a random distance. This is easily accomplished by first creating a *move* instruction and then dragging the *random number* question tile into the distance parameter. The result is shown here:



The *random number* question returns a fractional value between 0 and 1. When the above instruction is executed, the penguin will slide forward some fractional amount between 0 and 1. We don't know what that number will be. And, each time the program is executed the distance is likely to be different than the previous execution. This unpredictability is what makes it random.

But, suppose you want a random distance between 1 and 3, instead of between 0 and 1. To specify the range of numbers for the random number question, click *more* in the *random number* tile (purple) and select the *minimum* value. Then, do the same to select a *maximum* value. The instruction with the range 1 to 3 is shown here:



Now, each time the program is executed the penguin will slide forward 1.8 meters or 2.3 meters or ... some other random amount between 1 and 3. You can test this out by running the world and pressing the **Restart** button several times to observe the sliding moving.

Move to and random motion

The *move to* instruction was introduced in Tips & Techniques 4 as a way to make one object move to the location of another object, where a basketball was moved to the rim of the hoop. But, suppose we just want to move an object to a random location in the world, rather than the location of another object. If we combine the *random number* question with the *move to* instruction, we can create a form of random motion where objects move to a random location. Random motion of this sort is essential in many different kinds of games and simulations. As an

example, consider the world in Figure T-6-4. We want the goldfish to swim in a random-like movement. To do this, the goldfish should move to a random position but the new position should be fairly close to its current position, so the animation will look realistic.

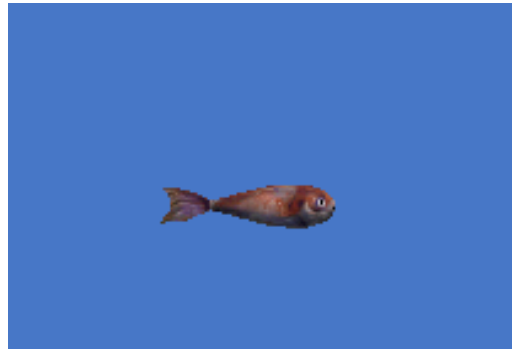


Figure T-6-4. Goldfish, for random motion animation

We can use the *move to* instruction to create random motion by following a few simple steps. First, create a *move to* instruction for the goldfish, accepting the default Vector3 position. (You may recall that Vector3(0,0,0) is the location of the center of the world.)



Then, the world-level question “*right, up, forward*” is dragged onto the “*Vector(0,0,0)*”, as in Figure T-6-5.

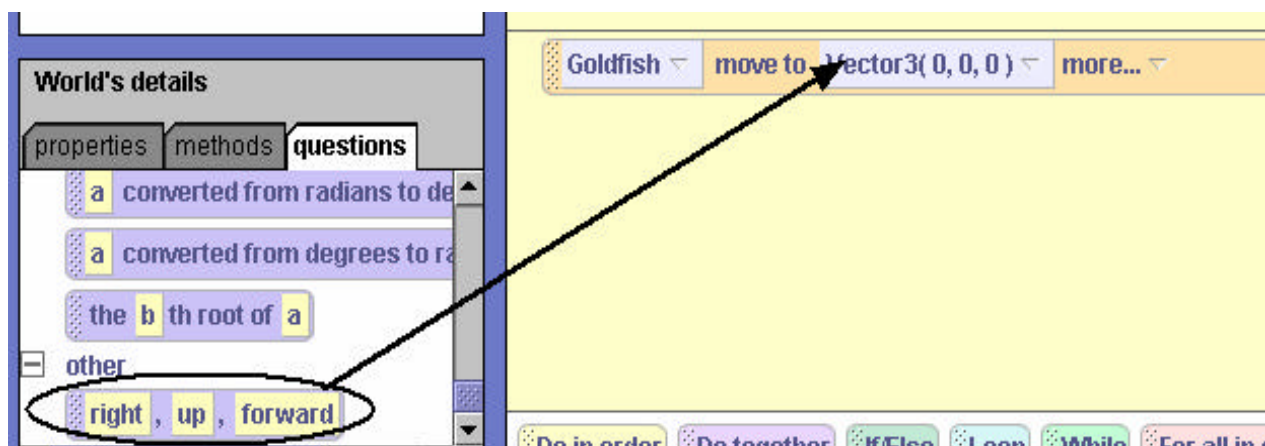


Figure T-6-5. Drag in template for *right, up, forward* position

The result is shown below, having **dummy values** of 1 for the right, up, and forward positions. We chose arbitrary values 1, 1, 1 from the popup menu. The *right, up, forward* parameters provide a template where we can specify the right, up, and forward movements to move the goldfish to a new location.



Of course, we do not really want the goldfish to move to position (1,1,1) in the world. Instead, we want the goldfish to move to a random location (but close to its current location). To get random values for the (right, up, forward) coordinates, the world random number question is used. As explained above, the random number question generates a random number between 0 and 1. But, what we really want to do is limit the random movement to a nearby location (so our animation looks more realistic). In other words, the goldfish's new location will be random – but will be adjusted to stay near to its current position. To keep the goldfish near its current position, we must find out what the goldfish's current position is. The built-in character-level question *goldfish's position* (see Figure T-6-6) returns whichever coordinate of the goldfish's location (right/left, up/down, forward/back) is desired.



Figure T-6-6. Finding the goldfish's current position

For convenience, we created a question, *adjust*, that will take a current position coordinate as a parameter, and return that value adjusted by a small, random amount, arbitrarily between $-1/5$ and $1/5$. See the *adjust* question in Figure T-6-7.

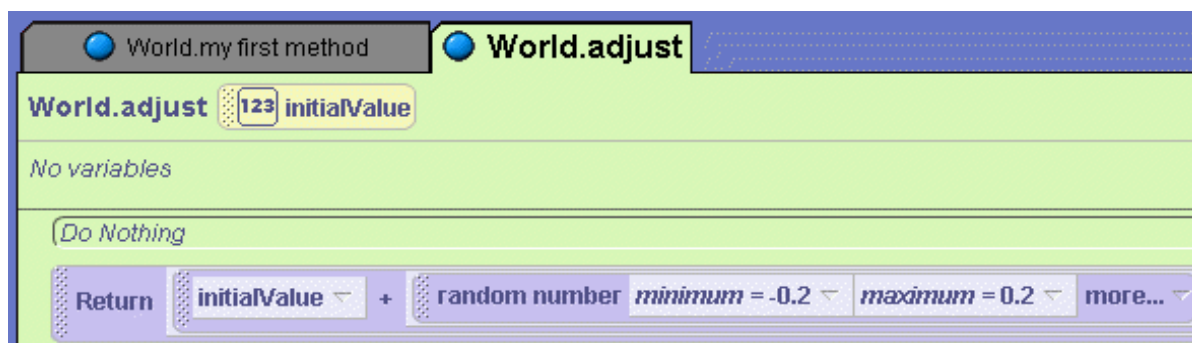


Figure T-6-7. The world-level *adjust* question

Now, we can use the *goldfish's position* together with *adjust* to move the goldfish a small random amount from its current position. The *move to* instruction is written by invoking the *adjust* method, passing in the goldfish's coordinates as parameters for each of the three coordinates (right, up, forward). Figure T-6-8 shows the complete code.

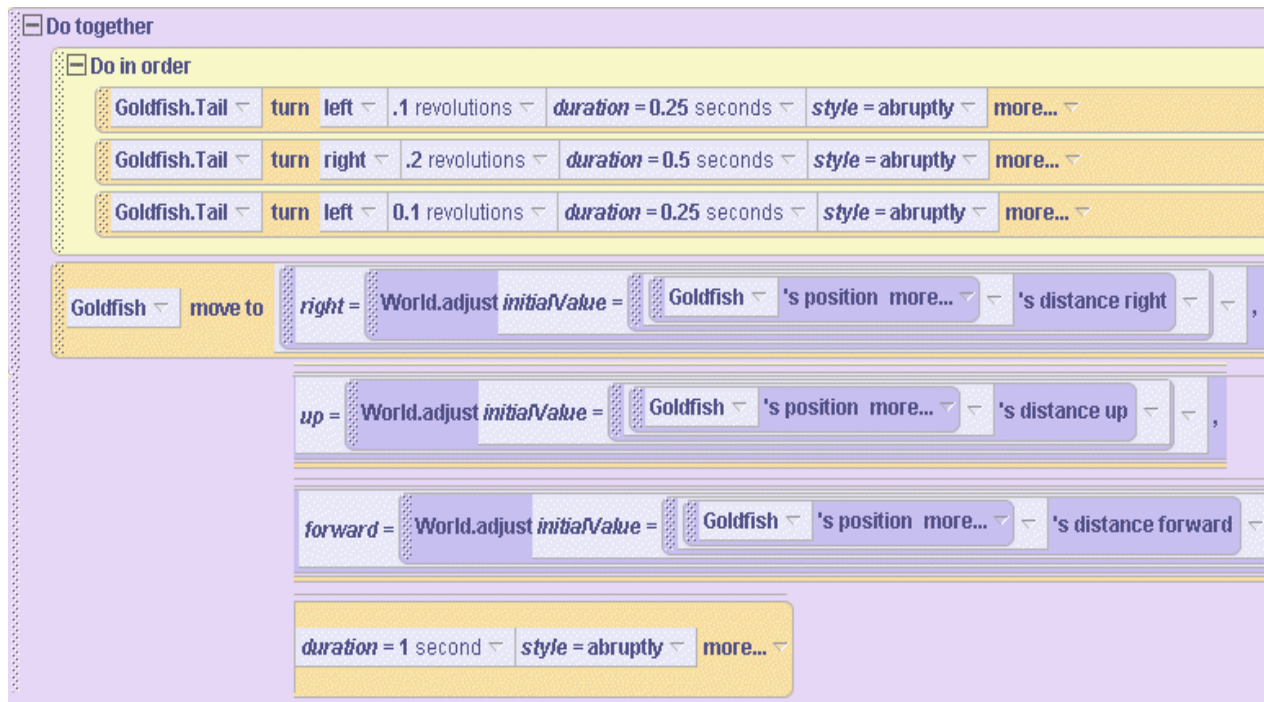


Figure T-6-8. fish swims to a random position

Camera: View from the back

Figure T-6-9 illustrates a fantasy world where objects have been added to the scene and rearranged somewhat to create an initial scene for an animation. From our perspective (as the person viewing the scene), the camera is allowing us to look at the scene “from the front.”



Figure T-6-9. Fantasy scene

In some worlds, you may want to move the camera around in the initial scene so it is viewed “from the back.” The problem is: how do we get the camera to “turn around” so the back of the scene is in view? One technique that seems to work well is to position the mouse cursor over the camera’s *forward control*, hold down the mouse and drag it forward. (The forward control is circled in Figure T-6-10). Continue to hold down the forward control and allow the camera and move straight forward until the camera seems to move right straight forward through the scene.



Figure T-6-10. Forward camera control

When the camera seems to have moved to the other side of the scene, let go of the forward control. Select the Camera in the Object tree and use a one-shot instruction to turn the camera $\frac{1}{2}$ revolution, as in Figure T-6-11.

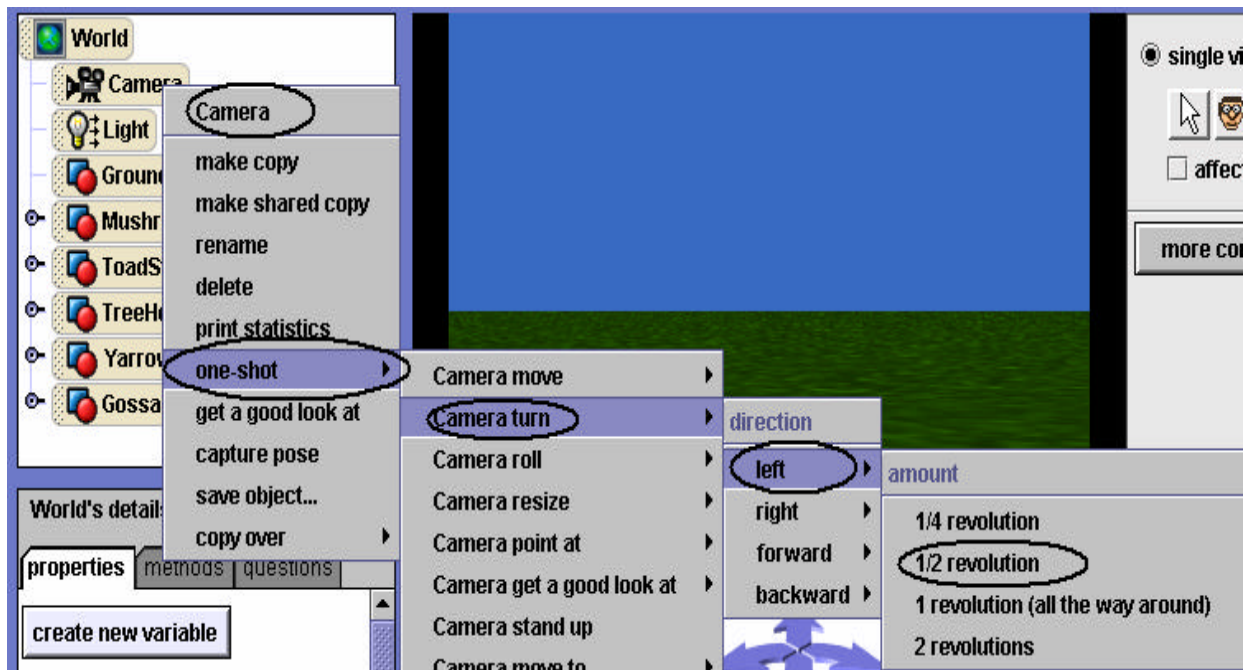


Figure T-6-11. A one-shot instruction to turn the camera around

The camera should now be facing in the opposite direction and you should be able to see the scene from the back, as illustrated in Figure T-6-12.



Figure T-6-12. View from the back of the scene

Lighting up the rear view

As you can see in Figure T-6-12, the rear view of the scene is somewhat disappointing. The scene looks rather dull because the built-in light object is shining so as to light up the scene from the initial camera point of view. To improve the lighting for a rear-view screen capture, you can add a light bulb to the scene – drag it in from the Lights folder in the web gallery, as shown in Figure T-6-13. Wow – what a difference!

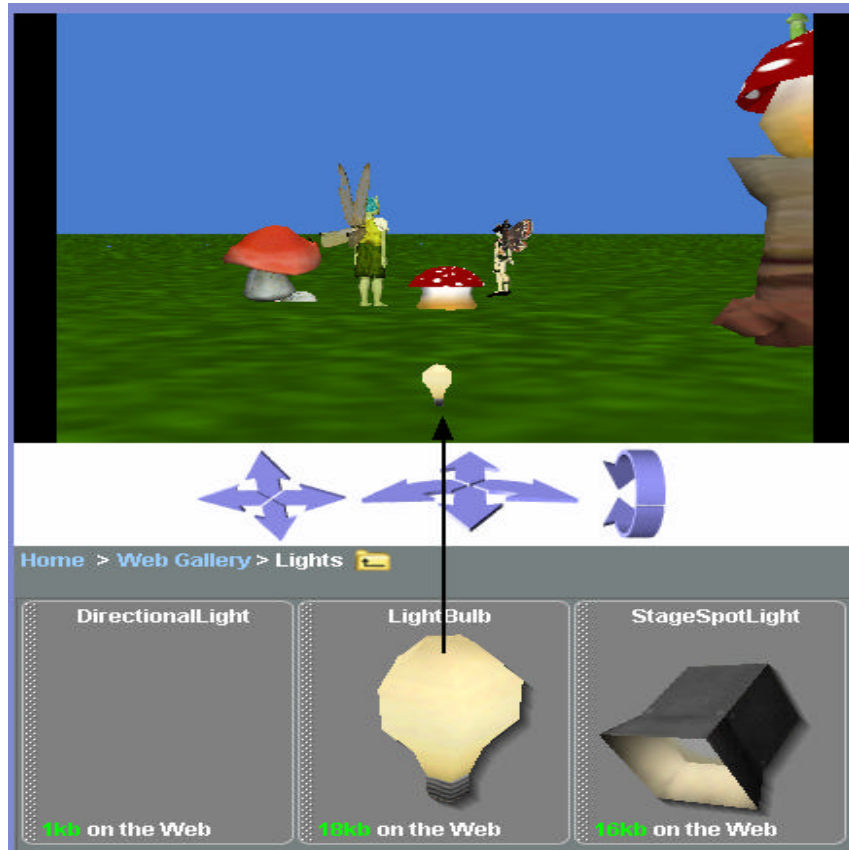


Figure T-6-13. Lighting added to rear view

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

7 Repetition: Definite and Indefinite Loops

In chapter 6, we looked at *If-else* statements for deciding whether or not an action is taken. In a way, we might think of *If-else* statements as allowing Alice to make a choice about an action -- a form of control of program execution. In this chapter, we look at a second kind of control statement, one that loops. A looping statement is one that iterates or **repeats** actions over and over again.

In section 7-1, we introduce looping statements with the *Loop* instruction. **In Alice, a Loop is a block of one or more instructions that are run again and again a given number of times.** We call the number of times the **count**. For this reason, the Loop may be referred to as a **counted loop**. The counted loop is very much like the “for loop” construct found in many other programming languages. But, Alice makes the loop easier to use by allowing the programmer to directly specify the count or to use a question to determine the count.

In section 7-2, *While* statements are introduced for repeating blocks of instructions where we do not know exactly how many times the loop should be repeated. A *While* statement corresponds naturally to the way we talk and think about certain actions in our everyday conversation. For example, we might say, “While I am driving, I will continue to keep my hands on the wheel and my eyes on the road.” Or, perhaps something like “While I am a member of the baseball team, I will practice batting.” **The *While* construct is intuitive and provides a way of thinking about actions that repeat while some condition is true.**

Note to the instructor:

Examples in chapters 7 and 8 illustrate repetition within methods rather than within questions (even though it is possible to have repetition in questions, as well). Our approach in this book is to use visualization to make programming concepts easier to teach and learn. A question is not visible. That is, a question returns a value but does not actually do any animation. So, in a question, the actual repetition is not visible and the student does not see the repetition in action. Thus, we do not see added value in trying to demonstrate the concept of repetition in examples where the repetition occurs in questions.

7-1 Loops

Introducing repetition

As our worlds become more sophisticated, the code tends to become longer. We also encounter the need to repeat an animation instruction, or several animation instructions, to make the same action occur over and over. It is quite possible that you have already built worlds where you needed to drag the same method into the editor several times. If an animation was to be repeated, the only way (until this point) was to drag the method into the editor as many times as needed. In this section, a programming construct is introduced that allows instructions to be repeated. The focus of this section is on how repetition works, and how to write repetition code in Alice.

The need for repetition

As an introduction to repetition, let's begin with a very simple world, shown in Figure 7-1-1. A bunny has snuck into his neighbor's garden and has spotted the nice broccoli shoots the gardener planted a few weeks ago. Broccoli is obviously on the bunny's menu today. Our task is to write a program to animate the bunny hopping over to munch a few bites of the broccoli.



Figure 7-1-1. Bunny in Garden

In this description of the program, let's assume we have already written a method for the bunny named *hop*. The hop method enables the bunny to hop forward by moving his legs up and down at the same time as the entire bunny moves forward. A possible implementation of the *hop* method appears in Figure 7-1-2. (Of course, the sound is optional.)



Figure 7-1-2. *Bunny.hop*

Let us suppose that the bunny is eight hops away from the broccoli. One possible way to animate eight bunny hops is given in Figure 7-1-3. A *Do in order* block is placed in the *World.my first method*. Then, the bunny points at the broccoli, and the *Bunny.hop* instruction is dragged into the editor eight times, as illustrated in Figure 7-1-3.

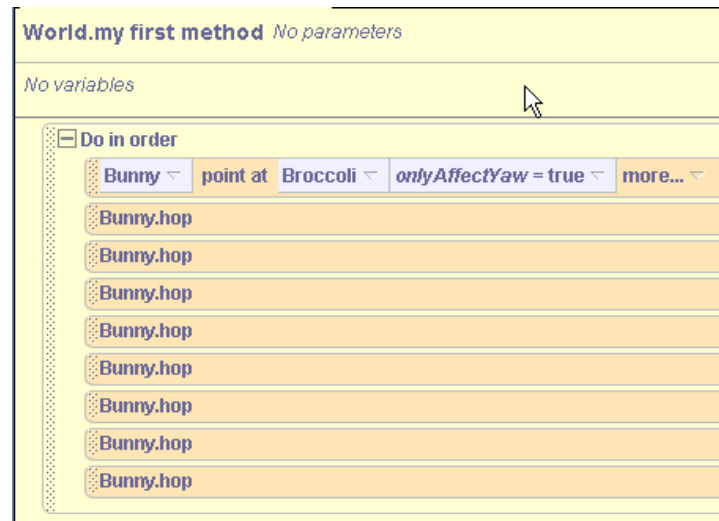


Figure 7-1-3. Eight hops

Actually, there is nothing wrong with this code. But, it was tedious to drag eight *Bunny.hop* instructions into the World script. Alice provides a special program construct, called a *Loop*, to allow repeated motion without having to do so much work. A *Loop* instruction is commonly found in many programming languages. Loops are used for the purpose of providing a simple and easy way to repeat an action a counted number of times. In programming terminology, program code that repeats (executes again and again) is said to **iterate**.

Using a Loop

To create a loop in a program, the *Loop* tile is dragged into the editor. When the *Loop* tile is dragged into the editor, a popup menu offers a choice for the **count** (the number of times the loop will execute), as shown in Figure 7-1-4. In this example, the number 8 is entered as the count.

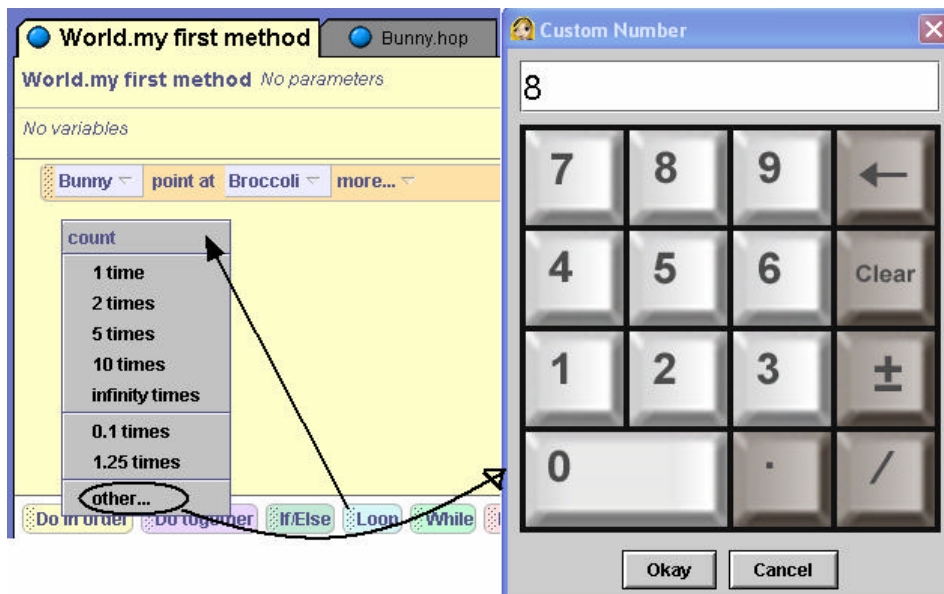


Figure 7-1-4. Selecting a count

Then, the *Bunny.hop* instruction is placed inside the *Loop* block, as shown in Figure 7-1-5. When the program is run, the Bunny will point at the Broccoli and then hop eight times. That is, the **loop will iterate eight times**. The benefit of using a *Loop* is immediately obvious – the *Loop* is quick and easy to write. And, the *Loop* is easy to understand! In this example, the loop count tells Alice to run the *Bunny.hop* method 8 times. The count must be a positive number value.

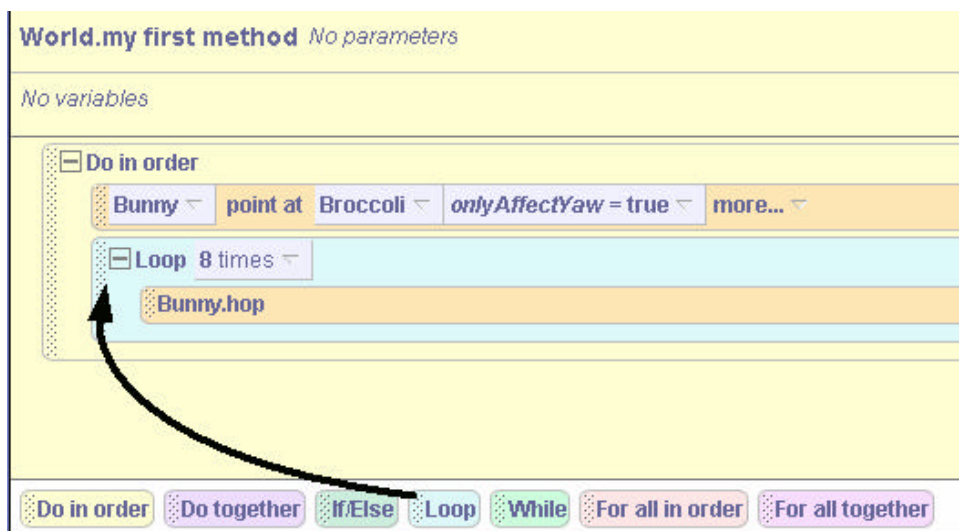


Figure 7-1-5. Using the *Loop* instruction

Loops and *Do in order* or *Do together*

We placed a single instruction (*Bunny.hop*) in the *Loop* block for this example. But, several instructions could have been dragged into the block and all the instructions would be repeated. A *Do in order* or *Do together* block can be nested inside the loop to control how the instructions are repeated. If we do not use a *Do in order* or *Do together* block inside the *Loop*, Alice will assume that the instructions are to be done in order.

A More Complex Example

As a more complex example, consider the initial scene illustrated in Figure 7-1-6. In this world, we want to write a program that will make the astronaut perform a moonwalk. (The moonwalk is the famous dance step popularized by Michael Jackson).

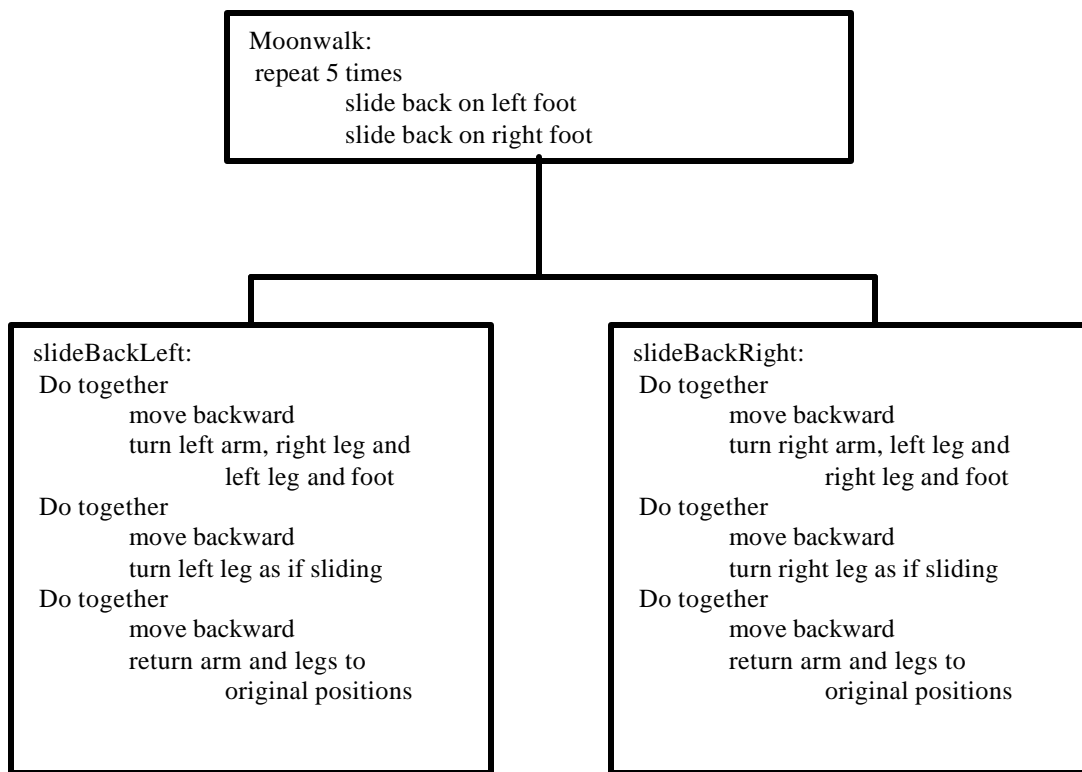


Figure 7-1-6. Astronaut on the moon initial scene

The astronaut will need to turn right and then moonwalk backward several steps. A moonwalk is performed by a sliding sort of motion where one leg slides backward and then the other leg slides backward. The sliding leg motions cause the entire body to move backward at the same time. A storyboard for the moonwalk is:

Moonwalk: repeat 5 times Do in order slide back on left foot slide back on right foot

Of course, the “slide back” motions for both the left and right feet are composed of several animation instructions. The storyboard must be developed further to show the slide back motions, as shown below. Once again, this is an example of **stepwise refinement** where an overall design is broken down in simpler steps, as shown below.



Symmetric Methods

Methods will be written to implement the *slideBackLeft* and *slideBackRight* storyboards. The actions in these two methods are a mirror image of the actions in the other. That is, what the left arm does in one, the right arm does in the other. Similarly, what the left leg or foot does in one, the right leg or foot does in the other. Due to this similarity, we have included only the code for the *slideBackLeft* method in Figure 7-1-8. The code for the *slideBackRight* method looks almost exactly the same as *slideBackLeft* (and is available on the CD).

After writing the *slideBackLeft* and *slideBackRight* methods, we can then write and test the moonwalk method. The code for the *moonwalk* method, as seen in Figure 7-1-7, is rather simple. All we have to do is call *slideBackLeft* and then *slideBackRight*.



Figure 7-1-7. The *moonwalk* method

astronaut.slideBackLeft *No parameters*

create new parameter

No variables

create new variable

Do in order

Do together

astronaut ▾ move backward ▾ 0.15 meters ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperArm.LeftLowerArm ▾ turn backward ▾ 0.1 revolutions ▾ style = abruptly ▾ more... ▾
astronaut.RightUpperLeg.RightLowerLeg ▾ turn forward ▾ 0.05 revolutions ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg.LeftLowerLeg ▾ turn backward ▾ 0.05 revolutions ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg.LeftLowerLeg.LeftFoot ▾ turn forward ▾ 0.05 revolutions ▾ style = abruptly ▾

Do together

astronaut ▾ move backward ▾ .15 meters ▾ duration = 0.5 seconds ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg ▾ turn forward ▾ .025 revolutions ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg.LeftLowerLeg ▾ turn forward ▾ 0.15 revolutions ▾ duration = 1 second ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg.LeftLowerLeg.LeftFoot ▾ turn backward ▾ 0.05 revolutions ▾ duration = 1 second ▾ style = abruptly ▾ more... ▾

Do together

astronaut ▾ move backward ▾ .15 meters ▾ duration = 0.5 seconds ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg ▾ turn forward ▾ .025 revolutions ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg.LeftLowerLeg ▾ turn forward ▾ 0.15 revolutions ▾ duration = 1 second ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg.LeftLowerLeg.LeftFoot ▾ turn backward ▾ 0.05 revolutions ▾ duration = 1 second ▾ style = abruptly ▾ more... ▾

Do together

astronaut.LeftUpperArm.LeftLowerArm ▾ turn forward ▾ 0.1 revolutions ▾ style = abruptly ▾ more... ▾
astronaut ▾ move backward ▾ .15 meters ▾ duration = 0.5 seconds ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg ▾ turn backward ▾ .025 revolutions ▾ style = abruptly ▾ more... ▾
astronaut.RightUpperLeg.RightLowerLeg ▾ turn backward ▾ 0.05 revolutions ▾ duration = 1 second ▾ style = abruptly ▾ more... ▾
astronaut.LeftUpperLeg.LeftLowerLeg ▾ turn backward ▾ 0.1 revolutions ▾ duration = 1 second ▾ style = abruptly ▾ more... ▾

Figure 7-1-8. Code for the *slideBackLeft* method

Now, we can complete the program code by using a loop in *my first method* to iterate the moonwalk step, as shown in Figure 7-1-9. We make the astronaut turn right so we can see the left side of the astronaut as the moonwalk is performed. And, we have the astronaut's helmet turn to face the user (for greater appeal).

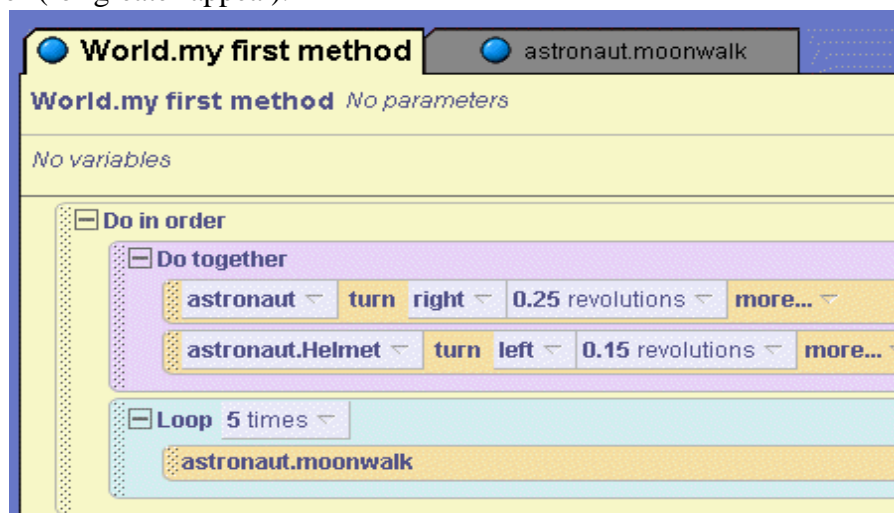


Figure 7-1-9. Using a *Loop* to iterate the moonwalk animation

Nesting of loops

Each of the examples above used only one loop instruction. Of course it is possible to have several *Loop* instructions in the same program. In fact, a *Loop* instruction can be nested inside another *Loop* instruction. In this section, we take a look at a world where this is an appropriate way to write the program code.

Consider the following situation. Suppose an initial scene with a two-wheeled Ferris wheel, as in Figure 7-1-10. An animation is to be written that simulates the running of this Ferris wheel. In this ride, to make it as jarring as possible, the double wheel will turn clockwise (rolling right in Alice terminology) while each of the individual wheels within the Ferris wheel turns counterclockwise.



Figure 7-1-10. A Ferris wheel

Figure 7-1-11 is code for a *Loop* to roll the Ferris wheel ten times. The “*style=abruptly*” parameter has been used to smooth out the rotating of the entire double wheel over each iteration. (It is worthwhile to experiment with the different *style=* options whenever an animation happens to run choppy, as this one will run if the *style=abruptly* option is not used. The reason we chose the abrupt style is the default style, *gently*, slows down the animation instruction as it begins and ends. Because we are repeating the instruction within a loop, we do not wish to have a slowdown in the action.)



Figure 7-1-11. Rotating the double wheel clockwise 10 times

With the motion of the entire Ferris wheel accomplished, instructions can be written to rotate each of the inner wheels counterclockwise (rolling left) while the outer double wheel is rotating clockwise. The code to rotate the inner wheels is presented in Figure 7-1-12.

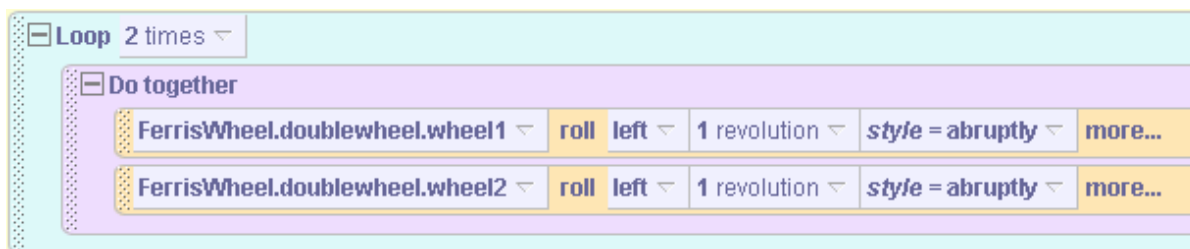


Figure 7-1-12. Rotating the inner wheels clockwise 2 times

Now, the rotations can be combined. To increase the excitement of the ride, we will have the inner wheels rotate more frequently, perhaps twice as often as the outer wheel. So, each time the double wheel rotates one revolution clockwise, the inner wheels should at the same time rotate twice counterclockwise. Figure 7-1-13 shows the modified code. Note that, since the inner wheels need to rotate twice, the duration of the double wheel is made to be twice the duration of rotation of the inner wheels.

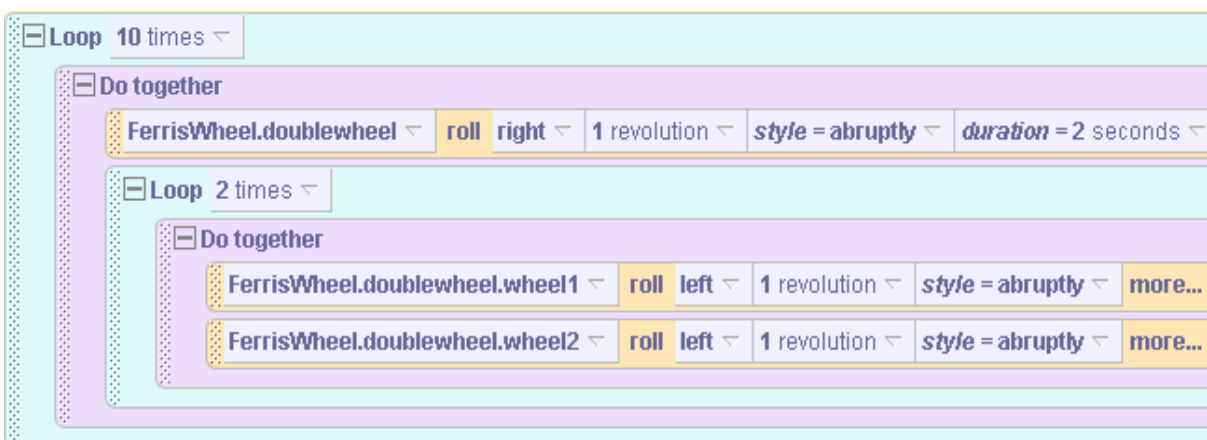


Figure 7-1-13. The complete code for the Ferris wheel

It is interesting to note the operation of the nested (inner loop). Each time the outer loop is run once, the inner loop runs twice. Notice that double wheel takes two seconds to complete its rotation, and the inner wheels will require 1 second (the default duration) to complete their rotations, but will rotate twice. In all, the inner wheels will rotate 20 times counterclockwise as the double wheel rotates 10 times clockwise.

Technical note on looping

In the examples presented here, the number of times a block of code was to loop was a specific number, 2, 8, or 10. But the count can also be a question that returns a number. For example, if a boy were 6 meters from the girl, the *Loop* in Figure 7-1-14 would make the boy turn a cartwheel 6 times.



Figure 7-1-14. Using a question to determine the loop count

Infinite Loop

One of the options in the popup menu for selecting a *Loop* count is **infinity**. (See Figure 7-1-5.) If infinity is selected, the loop will continue on and on until the program is stopped.

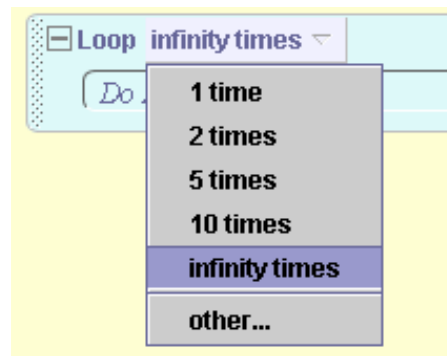


Figure 7-1-15. Selecting Infinity as the count

As an example of a program in which we would want to use an infinite loop, consider an amusement park animation where a carousel is one of the objects in the scene, as seen in Figure 7-1-16.



Figure 7-1-16. Carousel in infinite motion

To animate the carousel, two methods are used: one to animate the horses going up and down, and one to rotate the carousel at the same time as the horses go up and down. The code for each is presented in Figures 7-1-17 and 7-1-18.

World.my first method World.HorseAnimation

World.HorseAnimation * horseStartUp , * horseStartDown create n

No variables create

Do together

Do in order

horseStartDown move down 0.5 meters style = gently duration = 1 second

horseStartDown move up 0.5 meters style = gently duration = 1 second more

Do in order

horseStartUp move up 0.5 meters style = gently duration = 1 second more

horseStartUp move down 0.5 meters style = gently duration = 1 second more

Figure 7-1-17. Make two horses go and up and down on the carousel

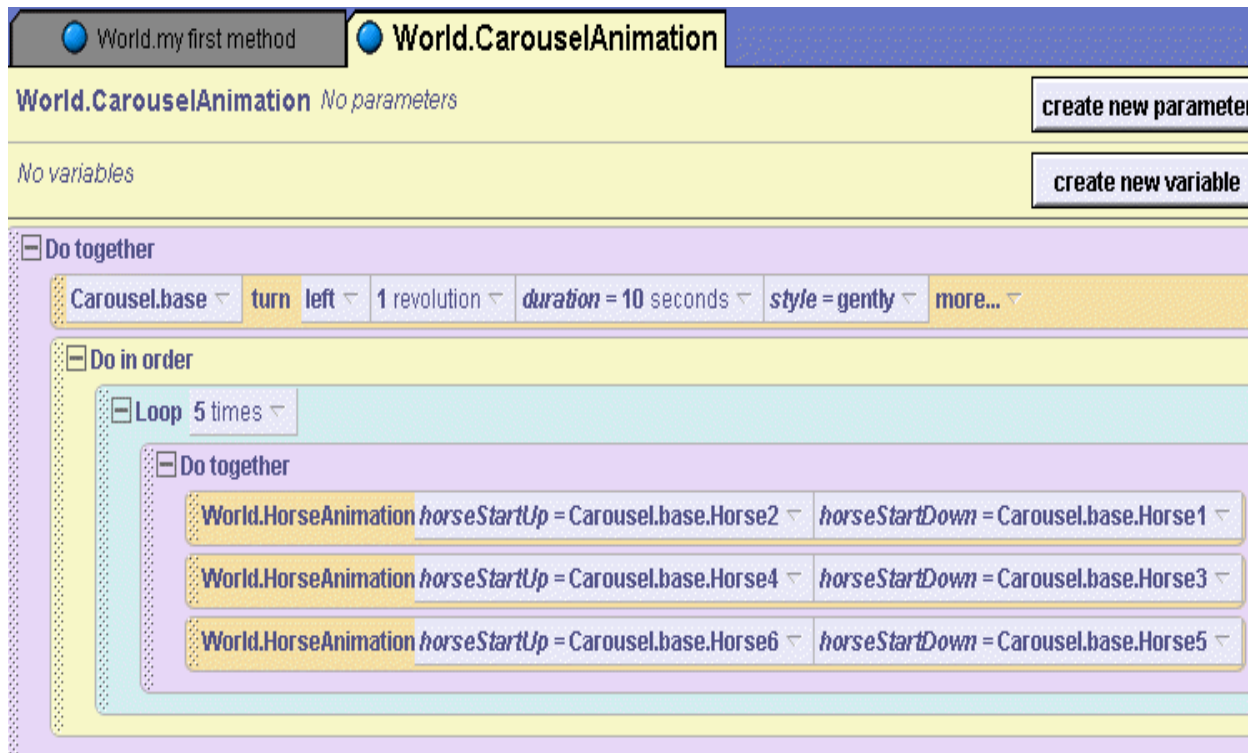


Figure 7-1-18. All horses going up and down as carousel turns around

We would like to make the carousel go round and round as long as the program is running. One way to make the carousel go around forever in an infinite loop is to use a *Loop* instruction with *infinity* as the count, as shown in the *carouselAnimationLoop* method in Figure 7-1-19.

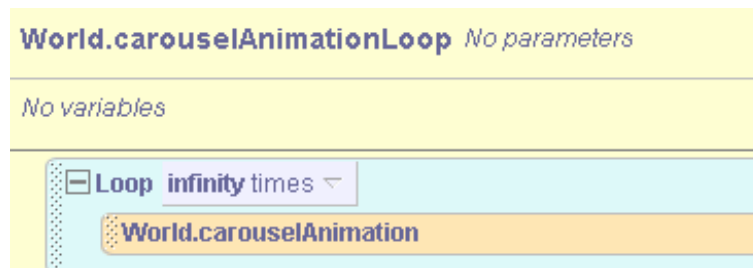


Figure 7-1-19. Loop *infinity* times

7-1 Exercises

1. CaughtInTheAct

This exercise is to complete the Bunny in the garden example in section 7-1. You will recall that in the storyline the baby bunny has snuck into his neighbor's garden and is hopping over to take a bite out of the tempting broccoli shoots. Code was presented to make the Bunny hop eight times (in a loop) over to the broccoli. Just as the bunny reaches the broccoli papa rabbit appears at the gateway to the garden and taps his foot in dismay. The bunny lowers his head sadly and hops a quick retreat out of the garden. Write a program to implement the Bunny in the garden animation. Your code should not only use a loop to make the bunny hop over to the broccoli (see

the initial scene in Figure 7-1-1) but also to hop out of the garden when papa rabbit catches him in the garden.



2. SquareBunnyHop

This exercise is to explore the use of nested loops. Papa rabbit has been teaching the Bunny some basic geometry. The lesson this morning is on the square shape. To demonstrate that the Bunny understands the idea of a square shape, the bunny is to hop in a square. Create a world with the Bunny and the hop method, as described in section 7-1. Use a loop to make the Bunny hop three times. When the loop ends, have the Bunny turn left $\frac{1}{4}$ revolution. Then add another loop to repeat the above actions as shown in the storyboard below.

```
loop 4 times
  loop 3 times
    Bunny.hop
    turn left  $\frac{1}{4}$  revolution
```

3. BlimpD

Create a scene as shown below with a blimp and a dragon. The dragon was on his way to the castle when he noticed a blimp. Dragons are rather curious creatures and this dragon can't resist going over to check it out. The dragon decides fly over and take a look around the blimp. Write a program to have the dragon move to the blimp and then fly (length-wise) around the blimp three times. Your program must include methods for *MoveToBlimp* and *FlyAroundBlimp* and a loop to repeat the *FlyAroundBlimp* method called three times (from *my first method*).



4. SnowManToStool

This exercise is to practice using a number question as the count for a *Loop*. Create a world with a snowman and a stool, as seen below. Use a *Loop* to make the snowman move to the stool, stopping just short of bumping into the stool. Use a *distance to* question to determine the count (the number of times the loop repeats).



5. SaloonSign

This old saloon is being converted into a tourist attraction. Use 3D text (see Tips & Techniques 1 for details on using 3D text) to create a neon sign to hang on the front of the balcony. Then use a loop to make the sign blink 10 times.



7-2 The indefinite loop – *While*

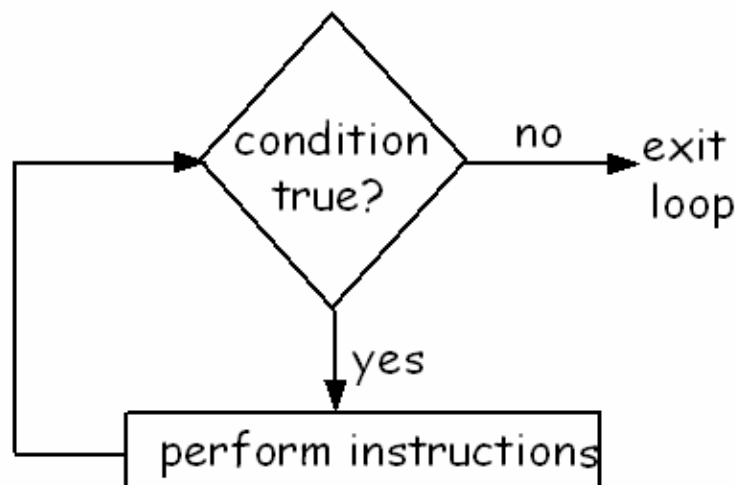
The *Loop* statement requires that the programmer specify the number of times the loop is to be repeated. This could be a numeric constant, such as 10, or a question such as *boy.distanceTo(girl)*. But, exactly how many times a loop should repeat might not be known ahead of time. This is often the case in games and simulations, where objects move in a random fashion. In this section, a form of looping known as the *while* statement will be introduced to handle situations where the programmer does not know (at the time the program is written) how many times the loop should be repeated.

The *while* statement is a **conditional loop**. One way to think about a while statement is:

```
while some condition is true
    perform instruction(s)
```

The instruction performed inside the while statement can be a single action or several actions enclosed in a *Do in order* or *Do together* block. The condition used in a while statement is the same kind of Boolean condition as used in *if-else* statements. The Boolean condition acts like a gatekeeper at a popular dance club. If the condition is true, entry is gained to the while loop and the instructions within the while statement are executed, otherwise the instructions are skipped.

But, unlike the *if/else* statement, the while statement is a loop. The diagram below illustrates how the while statement works as a loop.



If the condition is true and the instructions are performed, then (after the instructions are performed) the condition gets checked again. If the condition is still true, the instructions within the loop get repeated. But, if the condition has become false, the loop ends and Alice goes on to execute the next statement in the program. The way a while statement works as a loop makes it useful for situations where we do not know how many times the loop should be repeated. All we need to know is the condition that determines whether the loop will be repeated.

Chase scene

Let's look at an example of a situation where we do not know (ahead of time) how many times a block of code should be repeated. This animation is a "chase scene" simulation. Chase scenes are common in video games and animation films where one character is trying to catch another character as part of the game or story. In this example, a shark is hungry for dinner. The shark is going to chase after and catch a fleeing goldfish. OK, so this is not a gracious animation. But, sharks have to eat, too! And, a chase scene in an action movie is all about one individual chasing and another fleeing. Figure 7-2-1 shows a very simple initial world.



Figure 7-2-1. Chase scene

Problem

Our task is to animate the shark chasing the goldfish, trying to get close enough (say, within 0.5 meters of the goldfish) to gobble the goldfish down for dinner. Naturally, as the shark chases the goldfish, the goldfish is not standing still. Instead, the goldfish is dramatically moving away, trying to evade the shark by moving in a random direction.

Storyboard solution

The basic idea is that if the goldfish is more than 0.5 meter away from the shark, the shark is going to point at the goldfish and swim towards the goldfish. Of course, at the same time the shark is moving toward the goldfish, the goldfish is moving away to a random position nearby. If the goldfish is still more than 0.5 meters away, the shark will change course and swim towards the goldfish and the goldfish will try to swim away. Eventually, when the shark finally gets within 0.5 meter of the goldfish, the chase is over.

Let's plan a method for the chase animation using the *while* statement. Think about the situation like this:

"While the shark is more than .5 meters away from the goldfish, move the shark toward the goldfish and, at the same time, move the goldfish to a random position nearby"

The condition in our while statement will be "the goldfish is more than 0.5 meter away from the shark." If this condition is true, the chase will ensue. A storyboard is shown below.

```

While the goldfish is more than 0.5 meters away from the shark
  Do in order
    Point the shark at the goldfish
  Do together
    Shark swim towards the goldfish
    Goldfish flee away from the shark

The shark eats the goldfish

```

The condition must be true to allow entry into the loop. After running the instructions within the while statement, the condition will be evaluated again, and if the condition is still true, the instructions within the while statement will run again. This process continues until the condition finally evaluates to *false*. A *chase* method to implement the storyboard is illustrated in Figure 7-2-2.

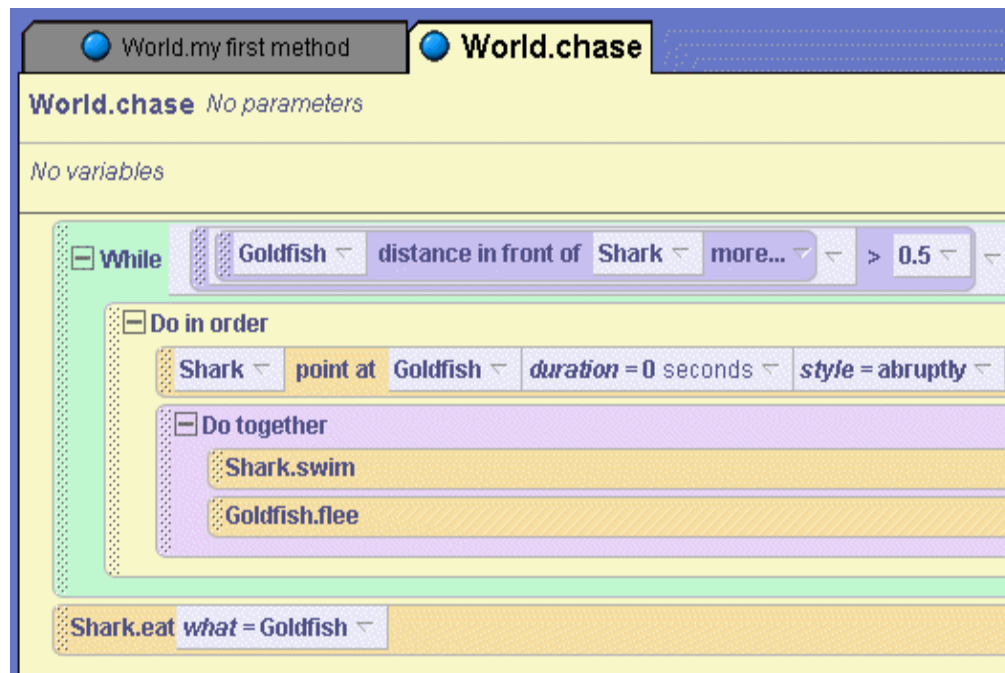


Figure 7-2-2. The *chase* method using a While loop

The *Shark.swim*, *Shark.eat*, and *Goldfish.flee* methods are called from the *chase* method. The *Shark.swim* method moves the shark forward and the torso of the shark right and left to simulate a swimming motion through water. Figure 7-2-3 illustrates *Shark.swim*. *Shark.eat* is called when the while loop ends. The method uses a parameter to specify what object in the scene is on the menu. In this example, *Shark.eat* was invoked with the Goldfish object as the parameter. The shark moves forward 1 meter to completely swallow the goldfish. The opacity of the goldfish is faded to 0% so the goldfish becomes invisible. Figure 7-2-4 shows the code for *Shark.eat*.

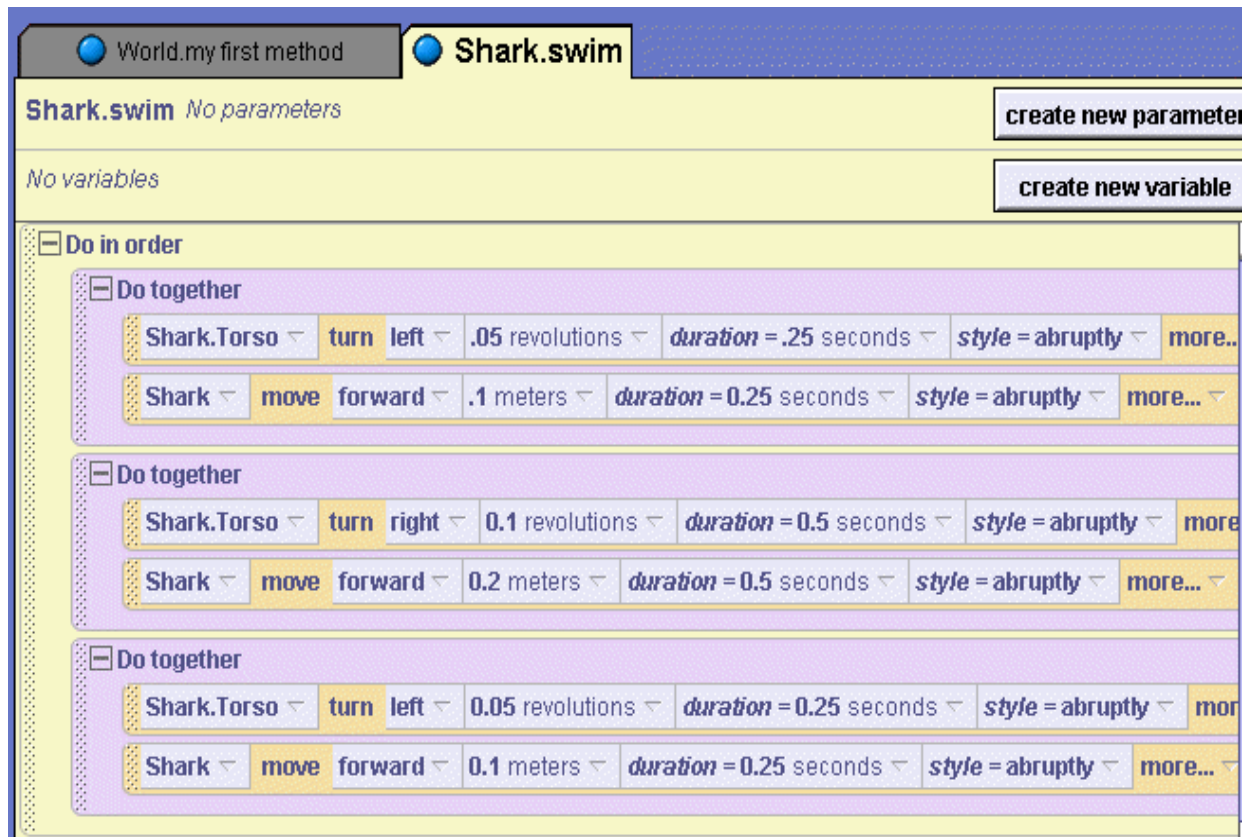


Figure 7-2-3. *Shark.swim* gives realistic shark movement

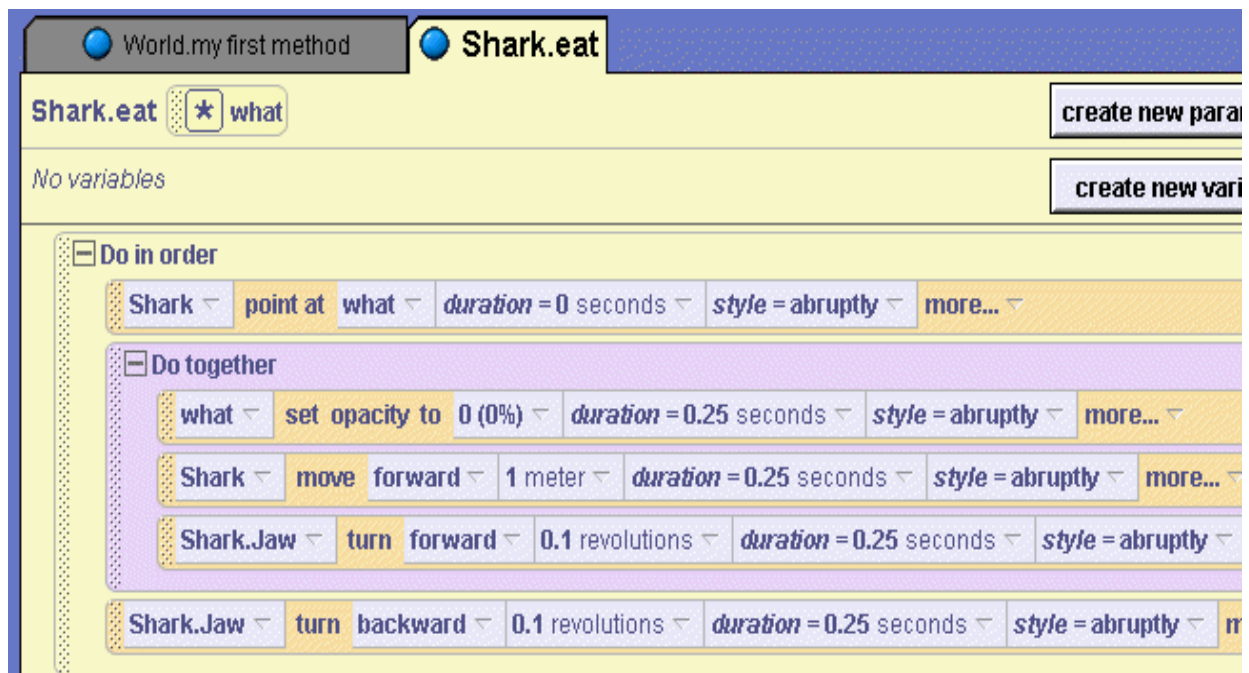


Figure 7-2-4. *Shark.eat* method

The *Goldfish.flee* method is the most interesting of the three methods because random motion is used to move the goldfish to a nearby random location. We make use of the *move to* instruction and the *World.adjust* question, shown in Figure 7-2-5. The *move to* instruction uses the current *goldfish position* together with *adjust* to move the goldfish a small random amount from its current position, as shown in Figure 7-2-6.

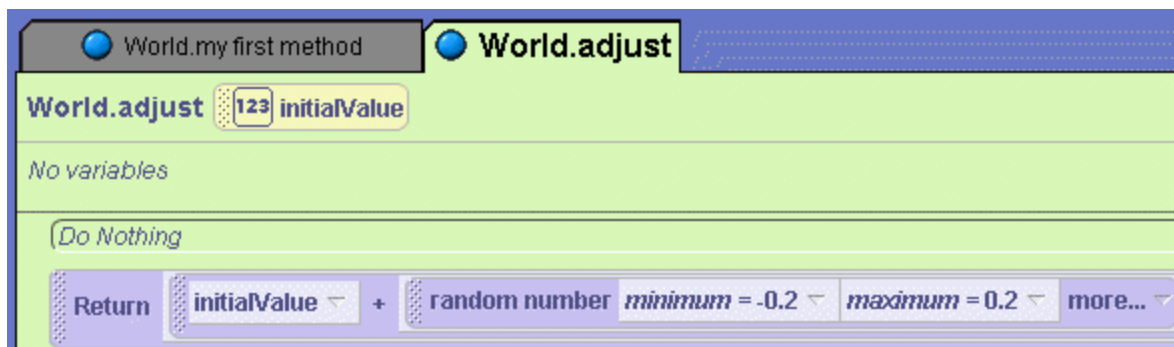


Figure 7-2-5. World-level *adjust* method

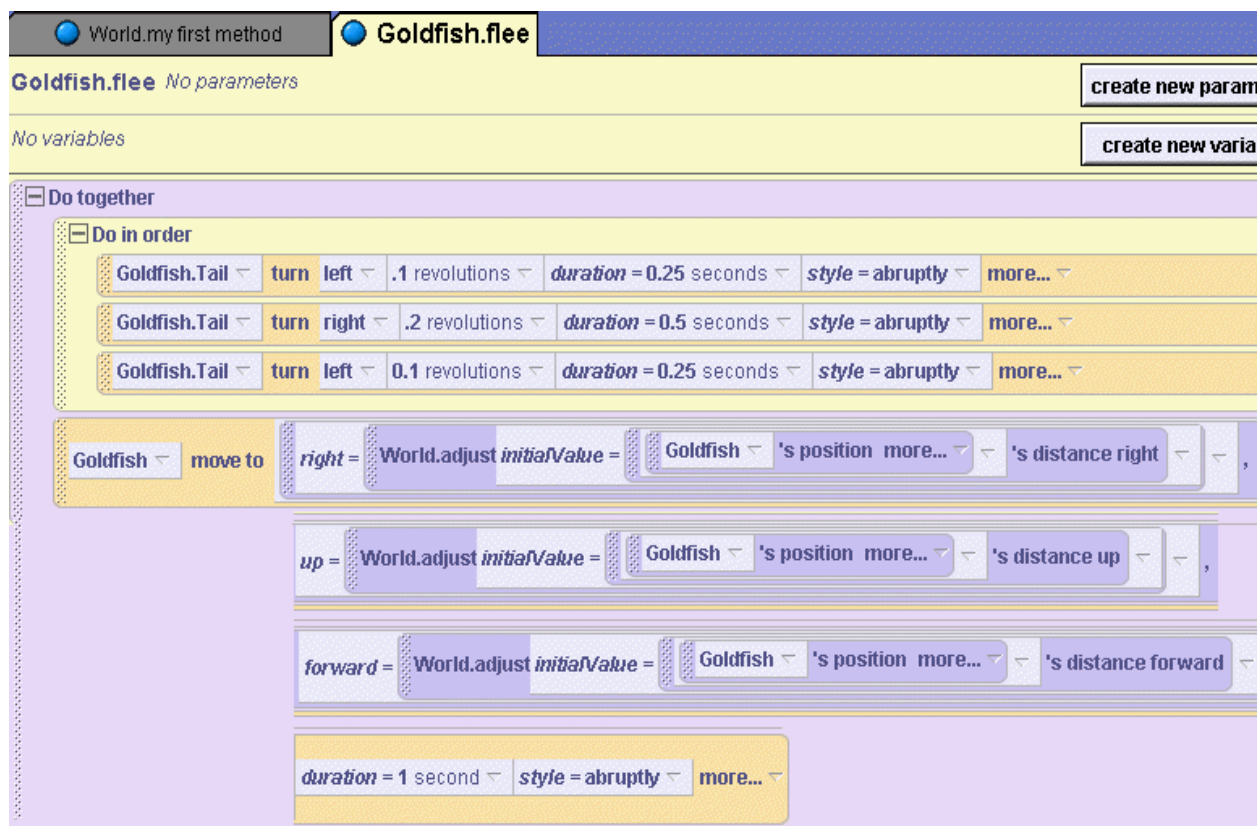


Figure 7-2-6. *Goldfish.flee*

The code in *adjust* and *Goldfish.flee* methods are exactly the same code as was explained in Tips & Techniques 6. This is a good example of the fact that we can often reuse code, saving ourselves much time and effort.

Infinite while

As a general rule, a while statement should contain instructions that change conditions in the world so the while loop will eventually end. Otherwise, the loop would continue to execute in a never-ending cycle – an **infinite loop**. An infinite loop is often a program error that makes the program seem to be “spinning its wheels.” In our shark chase example above, we avoided an infinite loop by carefully planning the distance the shark and the goldfish move with each execution of the loop. Although the goldfish is moving randomly, we set the maximum value to 0.2 meters (see *World.adjust*) in any 1 direction (forward, up, and/or backward). A bit of 3D geometry is needed to show that the total distance the goldfish travels must be less than .35 meters. So, the goldfish never moves more than 0.35 meters away from the shark. And, the shark always moves 0.4 meters (see *Shark.swim*) closer to the goldfish. The shark’s distance advantage guarantees that the shark is eventually going to catch up to the goldfish and the loop will end.

On the other hand, we should point out that there are some kinds of programs in which an infinite loop is desirable. For example, a game may have a loop that depends on some event such as hitting a target. Hitting the target allows the player to score points in the game but that is not the end of the game. The game is reset in some way and the game continues. Let’s extend this chase scene to a more game-like scenario where the shark continues to catch a goldfish as part of some sort of game. When the shark catches and eats the goldfish, the goldfish will disappear and after a short delay a new goldfish will appear in a different location on the screen, then the game will continue.

Two actions must be considered. The first action is that of creating a new goldfish. Actually, this is an illusion. We simply move the invisible goldfish to a new, random location and then, after a short delay, set the goldfish *opacity* property back to 100%(visible). When the goldfish reappears, it gives the viewer the impression that it is a different goldfish. The second action is to move the goldfish to a new location. The reason for moving the goldfish to a new location before making it visible again is so the chase can continue. If the goldfish were made visible again at the same location where it disappeared, the shark would be waiting at this location, and always immediately “catch” the goldfish again.

The goldfish can be moved to a new location using the *move to* instruction similar to the *Goldfish.flee* method. This time, however, we want to move the goldfish much further away. Still, limits must be placed on the amount of movement so as to keep the object in view of the camera and the shark. In this example, number values (-4 to 4) are used to limit the random motion of the goldfish in right/left and forward/back directions and (2 to 4) for the up/down direction. So, the new goldfish will start out somewhere in the world between 4 units forward and 4 units backward from the center of the world, between 2 and 4 units up or down, and between 4 units to the left and 4 units to the right of center. These are arbitrary values we tried with this animation. Figure 7-2-7 illustrates the *Goldfish.reappear* method.

Once a method is written to make the goldfish reappear, the while loop can be rewritten to be an infinite loop. The technique is to write the *while* statement as “*While true*” and use an *If-else* statement within the loop to control the repeated actions of the shark and goldfish, as can be seen in Figure 7-2-8. This illustrates that *If-else* statements can be combined with *while* statements. In some programs, a *while* statement is nested within an *If-else* statement. In other situations, an *If-else* statement may be nested with a *while* loop.

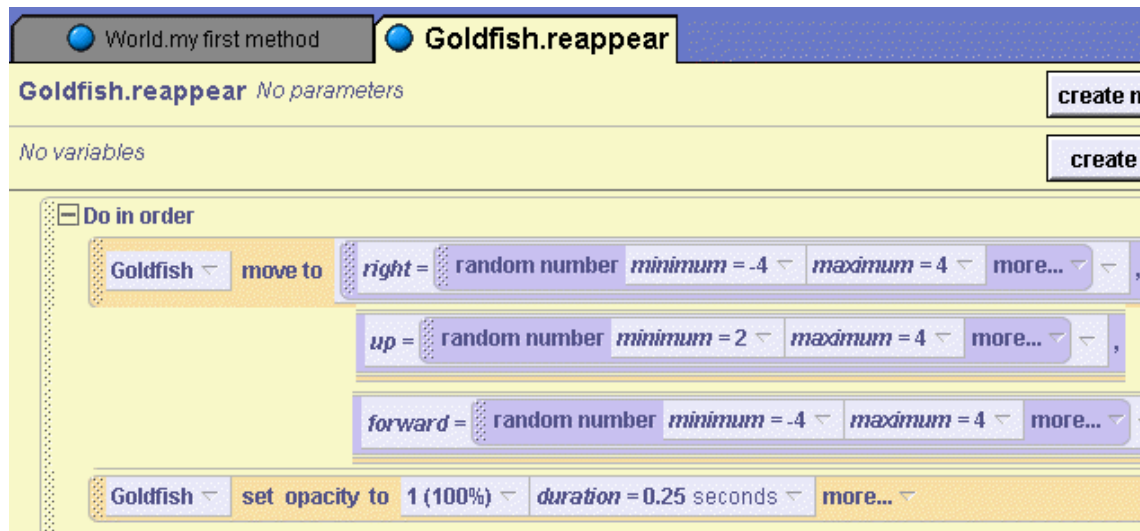


Figure 7-2-7. The goldfish reappears in a new random location

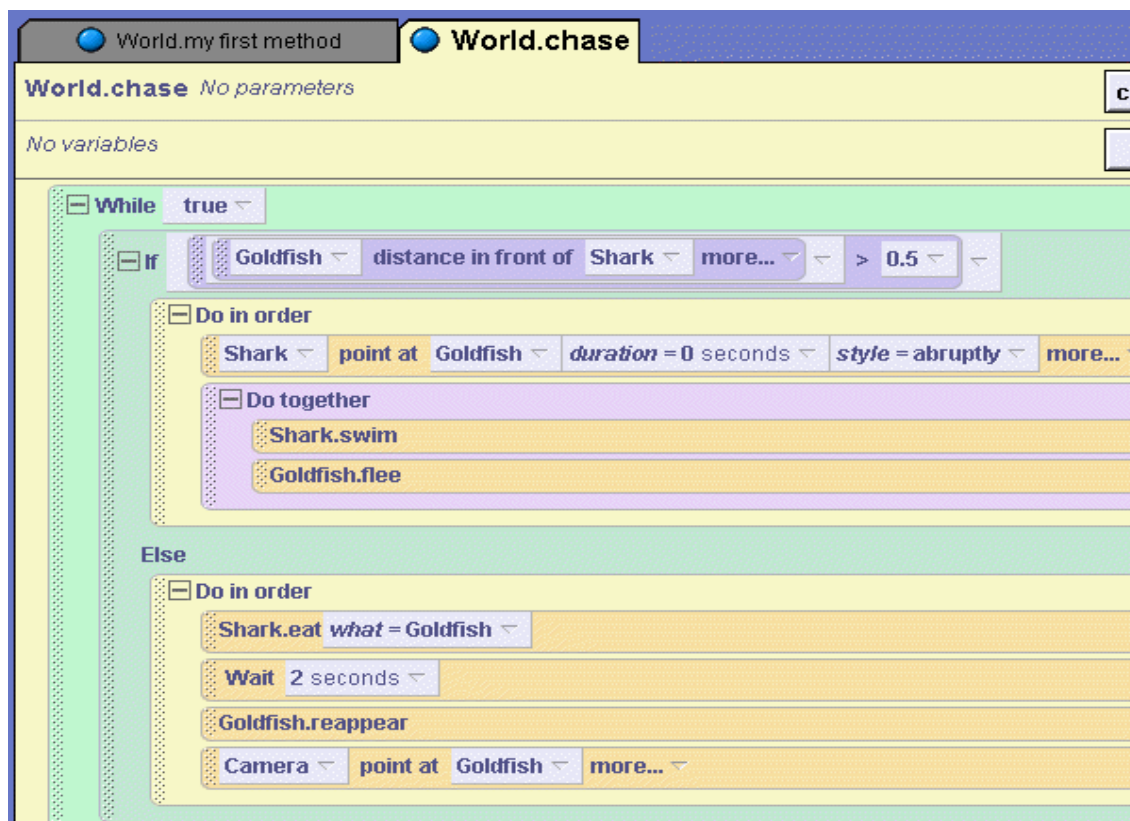


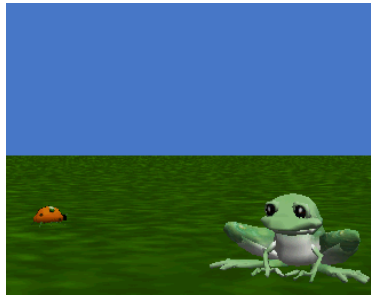
Figure 7-2-8. Infinite while loop

Now, when the shark eats the goldfish, the goldfish will reappear at a new random location. The camera will point at the goldfish, and the *while* loop will continue until the program is shut down.

7-2 Exercises

1. FrogAndLadybug

Create a world with a frog and a ladybug. Write an interactive program to allow the user to drag the ladybug around the scene. Use a *let the mouse move* event to allow the user to move the ladybug with the mouse. As the ladybug is dragged around the scene, make the frog chase after it by moving to the frog one hop at a time without colliding with the ladybug. If the user moves the ladybug within 2 meters of the frog, have the frog look at the camera and say “ribbit” then end the animation.



2. BumperCars

Create a simulation of the bumper car ride where the cars move continuously around within the bumper arena. The Bumper object is available in the Amusement Park gallery. Use the one-shot *copy* instruction to create additional bumper cars so there are 4 cars altogether inside the bumper arena. In this animation, each car should be moving forward a small amount until it gets too close to another car or to the wall. If the car gets too close to another car or to the wall, the car should turn slightly to get a different direction for moving forward. Use a switch control object (Switches folder in the web gallery) to stop and start the ride. As long as the switch is on, the ride should continue.

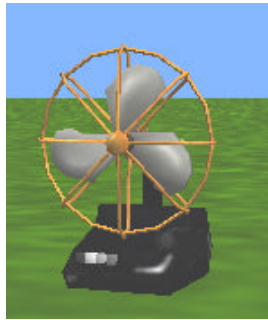


3. StopAndStart

Choose a ride object other than the carousel from the Amusement Park gallery that moves in a *circular pattern*. Circular pattern means that it moves in a round-and-round manner (e.g., the Ferris Wheel). Create a method that performs an animation appropriate for the ride object selected. Then, create a way to start and stop the ride using the *While <something> is true* event. For example, you could add a switch to your world that could be clicked to start and stop the ride.

4. SpeedControl

Place a fan (from Objects in the Gallery) in a new world. The fan has four buttons – high, medium, low, and off. Create a method that controls the speed at which the fan blades rotate, depending on whether the high, medium, or low button is clicked. (You may want to use several methods instead of just one method.) The fan should continue running until the animation stops running or the user clicks the off button.



5. OldTimeRockAndRoll

Create a world with an old fashioned phonograph in it. (See the Objects folder in the Gallery.) Create methods to turn the crank and turn the record. Then, create a *While <true> is true* event and use the BDE control mechanism (Tips & Techniques 7) to call a method that plays the record at the same time the crank is turning.



6. Windup Penguin

Create a world with a windup penguin. This is actually a penguin(from the Animals gallery) with a windup key (from the Objects gallery) positioned against its back. The key's *vehicle* property has been set to the penguin. In this world, make the penguin waddle (or walk) around the world continuously while its wind-up key turns. Use a "while..." event to make this an infinite loop.



Hint:

A simple Penguin waddle can be simulated by moving the penguin forward 0.12 meters at the same time as doing the following in order: roll the penguin to the left 0.01 revolutions, roll him to the right 0.02 revolutions and roll him to the left again 0.01 revolutions.

7 Summary

The loop construct was introduced in this chapter as a simple construct for repeating an instruction or block of instructions. Two kinds of repetition statements were presented: the counted *Loop* and the indefinite *While* loop. A counted loop allows us to specify exactly how many times a block of code will be repeated and a *While* loop allows us to repeat a block of code as long as some condition remains true. The advantage of using loops is immediately obvious. Loops are fast and easy to write and also easy to understand. Of course, it is possible to write loop instructions that are complicated. But, overall, loops are impressive programming tools.

Important concepts in this chapter

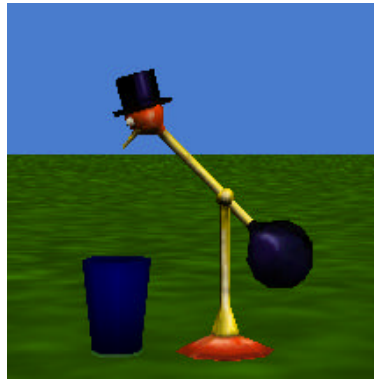
- The counted *Loop* instruction can be used to repeat an instruction, or a block of several instructions.
- A key component in a counted *Loop* is the count – the number of times the instructions within the loop will be repeated.
- The count must be a positive number or infinity.
- If the count is infinity, the loop will repeat until the program shuts down.
- A *Do in order* or *Do together* can be nested inside a loop. If neither a *Do in order* or a *Do together* is placed within the loop, Alice assumes that the instructions are to be executed in order.
- Loops can be nested within loops.
- When a loop is nested within a loop, the inner loop will execute the inner-loop-count for each execution of the outer loop. For example, if the outer loop count is 5 and the inner loop count is 10 then the inner loop executes 10 times for each of the 5 executions of the outer loop. In other words, the outer loop would execute 5 times and the inner loop will execute 50 times.
- The *While* statement is a conditional loop.
- The Boolean condition used for entry into a while statement is the same kind of Boolean condition as used in *If-else* statements. The difference is the *While* statement is a loop and the condition is checked again after the instructions within the while statement have been executed. If the condition is still true, the loop repeats.
- *If-else* statements can be combined with *While* statements. In some programs, a *While* statement is nested within an *If-else* statement. In other situations, an *If-else* statement may be nested with a *While* loop.

7 Projects

1. DrinkingParrot

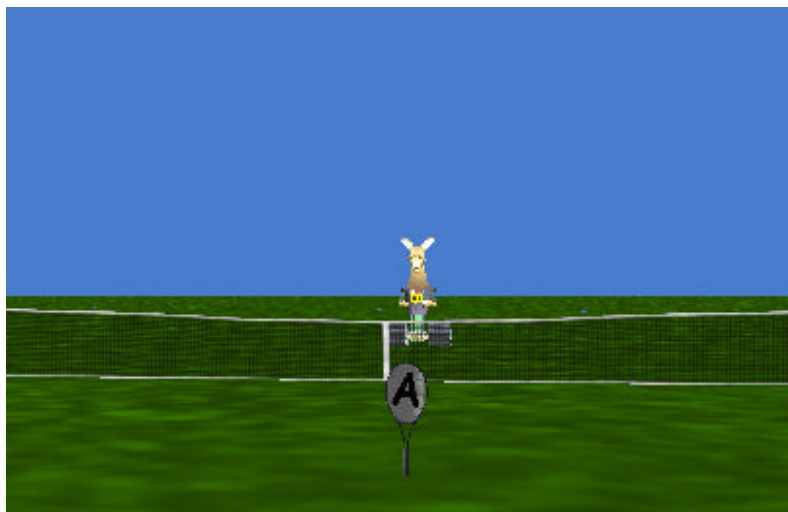
A small toy popular with children is a drinking parrot. The parrot is positioned in front of a container of water and the body of the parrot given a push. Because of the counterbalance of weights on either end of its body, the parrot repeatedly lowers its head into the water. Create a simulation of the drinking parrot (Objects folder in the gallery). Use an infinite Loop instruction to make the parrot drink.

Hint: In the world shown below, we used the blender object (Objects folder), pushed the base into the ground, and changed the color of the blender to blue to simulate a bucket of water.



2. TennisShot

Create a tennis shot game with a kangaroo_robot (SciFi folder on web gallery), a tennis ball, tennis net, and tennis racket (Sports folder on web gallery). Position the tennis ball immediately in front of the robot on the other side of the net (from the camera point of view) and the tennis racket on this side of the net (once again, from the camera point of view). The initial scene should appear as shown below.



Set up an event in the events handler to *let the mouse move* the tennis racket. (See Tips & Techniques 5.) In this game, the kangaroo_robot and the tennis ball move together left or right a

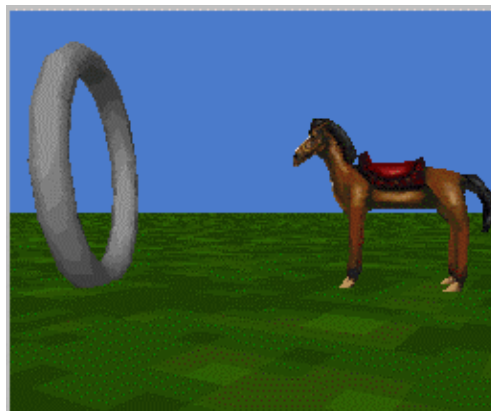
random distance (use the World random number question) between -1 and 1 meter. Then the robot “throws” the ball across the net -- the tennis ball moves up a random height and forward a given distance over a given duration. You will need to use trial and error to figure out an appropriate distance for the ball to move forward and the amount of time to allow for the move. When the ball has moved far enough towards the camera, it should be out of sight.

The player (user) will move the tennis racket to try to “hit” the tennis ball. A “hit” occurs when the tennis racket gets within 0.1 meters of the tennis ball. Actually, the tennis ball is virtual in this simulation and will go right through the racket even if the player manages to “hit” it. However, we will know if the player manages to “hit” the tennis ball, because the kangaroo_robot will wiggle his ears.

The real challenge in writing this program is to figure out whether the player is successful in moving the tennis racket to get it close enough to the tennis ball to hit it before it goes out of sight. To make this work, use a loop where each execution of the loop moves the tennis ball up and forward only a very short distance (something like 0.1 meters). Each time through the loop, check to see if the tennis racket has gotten close enough to the tennis ball to score a hit. As mentioned above, you will need to experiment with your world to figure out the appropriate count for the loop so as to eventually move the ball forward out of sight of the camera. When the loop ends, have the kangaroo_robot turn left $\frac{1}{4}$ revolution and move off the tennis court signaling that the game is over.

3. HorseThroughHoop

The horse is in training for a circus act. As part of the act, the horse is required to jump through a large hoop (a torus object from the Shapes folder of the gallery). Create a world, as shown below, with a horse facing the hoop. Write a program to use a loop to have the horse **trot** forward several times so as to move through the hoop. Use trial and error to determine the count for the loop.

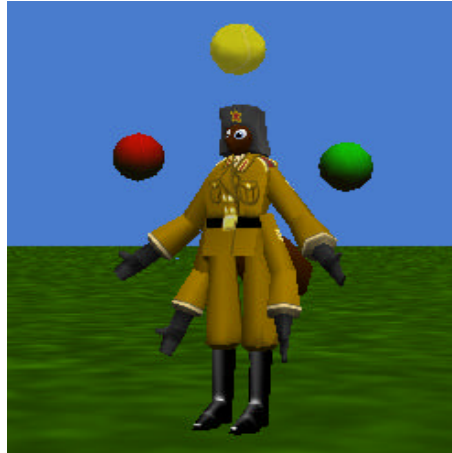


Your project must include a **trot** method that makes the horse perform a trot motion forward. A horse trots by moving the right front leg forward at the same time the left leg moves forward and then moves the left front leg forward at the same time the right leg moves forward. The leg motions can bend at knee for a more realistic simulation, if desired. Of course, the leg motions should happen simultaneously with the entire horse moving forward. The loop should call the **trot** method a four times.

4. Juggling

Create a world with a an object that has arms (the picture below shows an ant object) and three juggling balls. Of course, the character must have at least two arms and be able to move them in some way that resembles a tossing motion. Write a method to animate the object juggling the balls in the air. Use a loop to make the juggling act repeat five times and then the juggling balls should fall to the ground.

Hint: A juggling ball is easily created by adding a sphere or a tennis ball to the world, resizing it, and giving it a bright color.



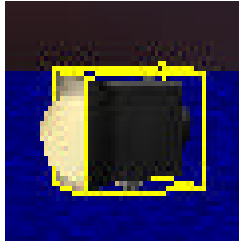
5. LighthouseWarning

Set up a world that contains a lighthouse, a spotlight, a lightbulb, a stars skydome and a number of boats, as seen in the image below. The lighthouse sits in the harbor to warn ships of shallow water. In this simulation, the light of the lighthouse is to rotate around, as in real life. You should be able to see the light on the side of the boats as the light of the lighthouse swings around.



To create a dramatic lighting effect in this animation, place a light bulb inside of the spotlight. The *lightbulb* and *spotlight* will be moved together to create a brighter beam of light. (Hint: set the *lightbulb*'s vehicle property to be the *spotlight*.) Then, place the spotlight inside the

windowed area at the top of the lighthouse. Finally, make both the *lightbulb* and the *spotlight* invisible. Since it is inside the lighthouse, a composite shot of the light is shown below.



Use repetitive world methods to continuously move each boat forward past the lighthouse and to continuously rotate the spotlight. (*Hint:* You may want to set the duration of the rotation to be 3 seconds or so.) In “my first method”, make the world dark by setting the brightness property of the world’s light to be 0. Then, invoke the methods to create the lighting effect.

8 Repetition: Recursion

This chapter continues our discussion of repetition with an introduction to a powerful form of repetition known as **recursion**. Recursion means that a method (or a question) calls itself. This is an extremely powerful technique that greatly enhances the types of problems that can be solved in Alice.

As with the *While* loop statement, recursion is often used where we not know (at the time the program is written) the number of times a block of code should be repeated. In what kinds of situations do we not know the number of repetitions?

In Alice, there are two major situations where we do not know ahead of time the count of repetitions. The first is when *random motion* is involved. Random motion means that an object is moving in some way that is unpredictable. We used random motion in the shark chase example in chapter 7 section 2. In section 8-1, we will revisit the shark chase example to explore the technique of writing methods where repetition is implemented with recursion. The technical term for the kind of looping presented in section 8-1 is **generative recursion**.

The second situation where we do not have a count of repetitions is when some complex computation is to be done that depends on an ability to break a problem down into smaller sub-problems. Once the smaller sub-problems are solved, the solutions to the smaller sub-problems are used to cooperatively solve the larger problem. In these kinds of situations, we can use a different form of recursion. In Section 8-2, we will look at a famous puzzle and present a solution using a second form of recursion, known as **structural recursion**.

8-1 Introduction to Recursion

Two powerful constructs for repeating blocks of instructions were presented in Chapter 7: the *Loop* statement for repeating a block of instructions a counted number of times and the *While* statement to repeat instructions when the count was not known. This chapter presents another form of repetition, **recursion**. Recursion, like the *While* statement, is a form of looping used to handle situations where the programmer does not know how many times the loop should be repeated. But, unlike *Loop* and *While* statements, recursion is not a program code statement with a special word that identifies it as part of the programming language. Instead, recursion is a well-known programming technique where a method (or a question) calls itself. Examples in this section will show you how to use this technique.

Revisiting the shark *Chase*

As an example of recursion, let's borrow the chase animation from chapter 7. In this world, seen in Figure 8-1-1, a shark is hungry for dinner. The shark chases after and catches a fleeing fish. Figure 8-1-2 shows the code for the *chase* method as completed in chapter 7.



Figure 8-1-1. Chase scene

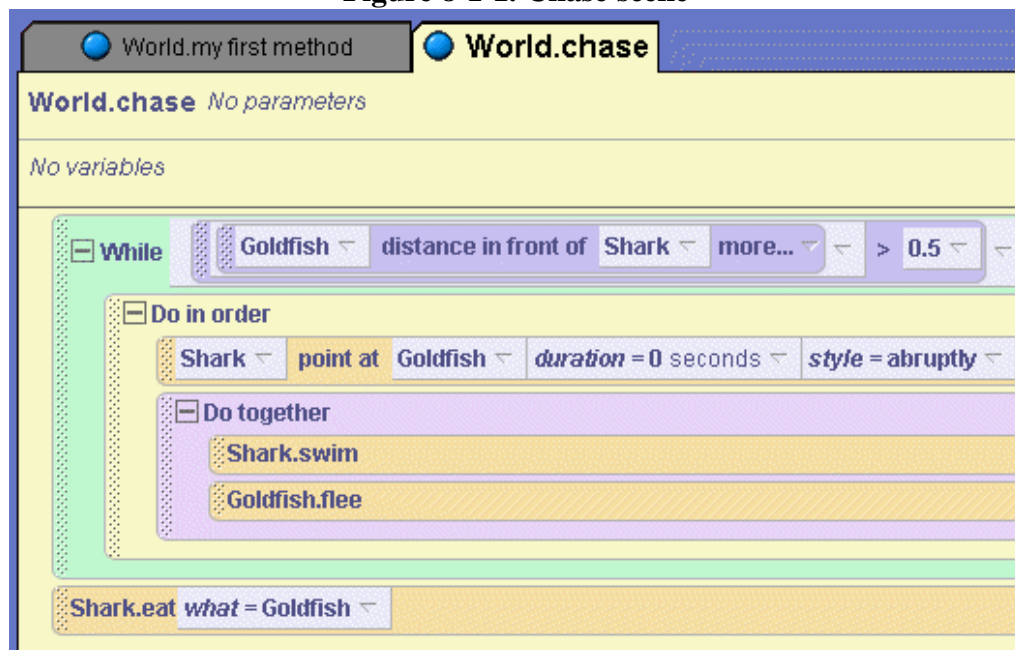
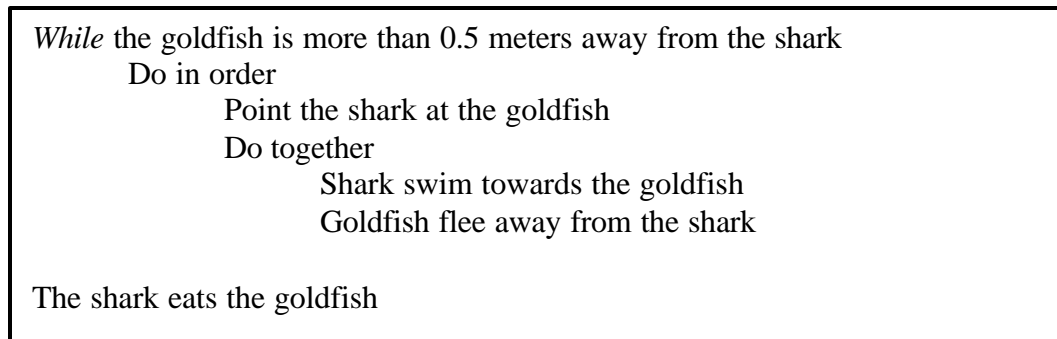


Figure 8-1-2. The *chase* method implemented using *While*

The key concept in the *chase* method was repetition implemented as a *While* loop. Code written using a *While* loop can also be written using recursion. One fundamental difference between the

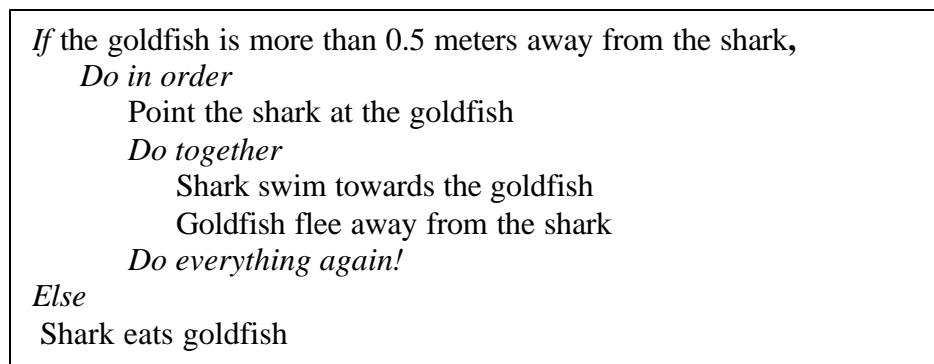
While loop and recursive looping is how we think about the solution to the problem. The following storyboard is the one we used when planning a *While* loop solution for the shark chase in chapter 7:



Now we want to think about the shark chase in terms of recursion. This means we will use a decision statement (in Alice, an *If-else* statement) to structure the solution. Think about it this way:

“If the goldfish is more than 0.5 meters away from the shark, the shark will move toward the goldfish and the goldfish will flee. Then, repeat (that is, do it again). On the other hand, if the goldfish is not more than 0.5 meters away from the shark, the shark will eat the goldfish.”

The following storyboard illustrates the recursive solution:



Comparing the two storyboards, it is easy to see that the *If-else* structure has replaced the *While* statement. Another big difference is the statement "*Do everything again.*" This means that the method is to be repeated by calling itself – this is where the recursion takes place. Clearly, an *If-else* decision statement is important to recursion because it not only decides whether the block of instructions will be executed but also decides whether the method will be repeated.

Now, the *chase* method can be written using the recursion technique. Figure 8-1-3 illustrates the *chase* method. The circled statement, *World.chase*, is the recursive call.

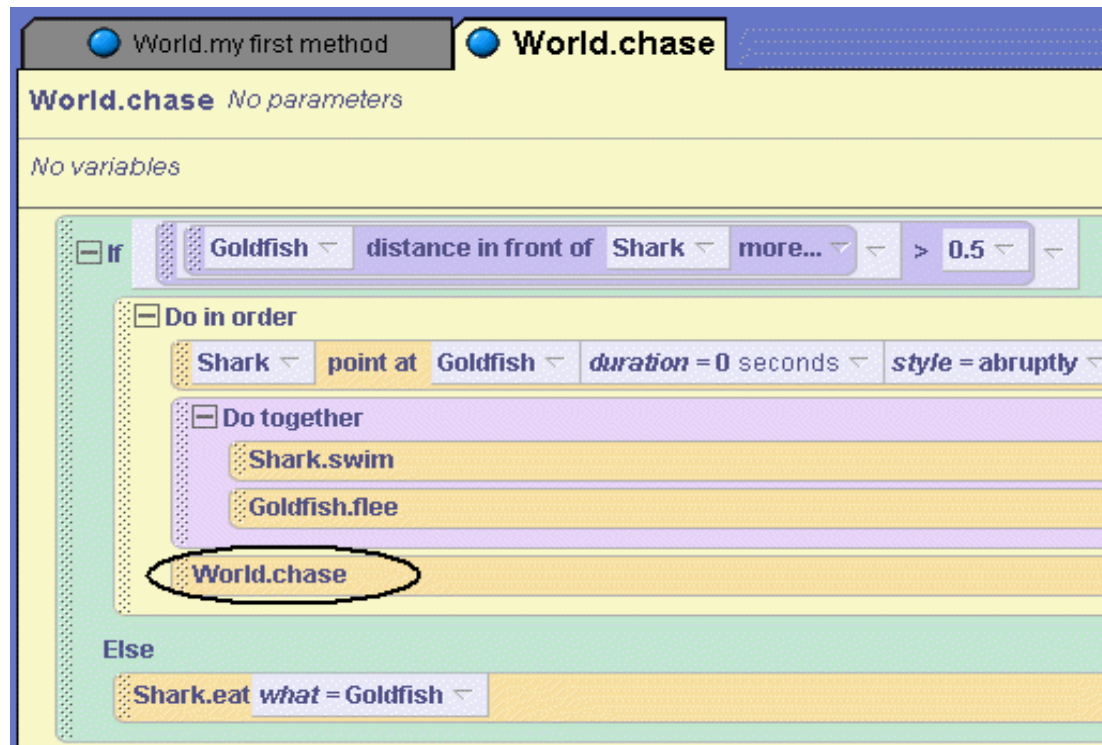


Figure 8-1-3. *World.chase* method calls itself

An important concept to understand is that the recursive call is repeated over and over as long as the condition for the *If* statement is *true*. But, when the condition for the *If* statement becomes *false* the *Else* kicks in and the recursive calls stop. In the *Else*-part of the *If-else* statement in *World.chase*, a call to *Shark.eat (what=Goldfish)* is the end of the chase. In this example, the recursive call was in the *If*-part of the *If-else* structure. We should point out that a recursive call is not restricted to the *If*-part of the *If-else* structure -- a recursive call can be made in either the *If*-part or the *Else*-part.

In terms of writing program code, the recursive version of a method is often almost exactly the same as the *While* loop version. The syntax difference is that a decision statement (*If-else*) replaces the *While* statement and the recursive call performs the looping action. Some programmers prefer using recursion and others prefer using *While*. In some other programming languages the *While* statement runs faster than recursion and so a *While* loop is preferred. However, in Alice both approaches run correctly, so it is a matter of choice as to which approach you wish to use.

Terminology

The form of recursion illustrated in this example is said to be **generative**. In generative recursion, a decision is made. Depending on the results of that decision, the method is either finished or will be executed again. If the method is executed again, the same decision is made once more. This repetition may go on and on until, eventually, the tested condition changes and the solution is finally complete. This kind of recursion is called generative recursion because additional executions are "generated" each time the result of the previous decision (in this case, "Is the goldfish more than 0.5 meters away from the shark?") is true.

The solution in our shark chase example is also said to be **tail recursive**. A method is said to be tail recursive when two conditions are met:

- (1) **Only one** recursive call appears in the method, and
- (2) The recursive call is the **last** statement in the *If-part* or the *Else-part* of an *if-else* statement.

In the code illustrated in Figure 8-1-3, there is only one call to *World.chase* and it is the last statement in the *If-part* of the *If-else* statement. So, the *chase* method is tail recursive.

A game-like example

This example is a simple version of a horse race in a carnival game. In the initial scene, shown in Figure 8-1-4, the finish line was created by adding a kayak paddle to the scene and positioning it flat on the ground. (Other objects could be used to simulate a finish line, of course.) Then, 3 colored markers (we used gumdrops) were added for the target location where each horse should run to finish the race. Unlike a traditional horse race, where horses run around an oval-shaped track and each horse breaks to the inside of the track, in a carnival game horses move straight ahead in a mechanical track and you can win a prize if you pick the right horse. We changed the color of each horse's midsection to match the color of the marker (gumdrop) to which it will run. So, the horse with a red midsection will race to the red marker, the horse with blue midsection to the blue marker, and the same for the yellow marker.

There are at least two choices as to how to run the race. One choice is to have all the horses run continually, perhaps different random amounts, until one of the horses reaches the finish line. (This is left as a lab exercise.) The approach we show here is to generate a random number and then, depending on the value of that random number, move one of the horses forward. An essential step is how to decide when the game will end. Well, as soon as one of the horses reaches the finish line the game is over.



Figure 8-1-4. Initial scene for a 3-horse race

As long as all the horses have not yet reached the finish line, the horses should move forward toward the finish line. Recursion can be used to implement the race. To write the code for the race, we have two problems:

- 1) How to determine when the race is over.
- 2) How to *randomly select* one of the horses to move forward for each execution of the loop.

How to determine when the race is over

The race is over when one of the horses gets close enough to its marker on the finish line to be declared a winner. Built-in questions in Alice offer a number of possible choices to find out how close a horse is to its marker. In this example, let's use the question *at least threshold away from*, where threshold is some minimal distance between the horse and its marker on the finish line. **(In our world, we found the winning distance between horse and finish line to be 1.1 meters. But, you will have to experiment in your world to find the best distance.)**

The race is run by executing a method where one of the horses is randomly selected and moved forward. Then, we need to check whether the race is now over. If the race is over, the method should end. Otherwise, the method should repeat. So what is needed is a test condition that will determine when the race is over. In this example, it is not enough just to test "Horse is at least 1.1 meters from gumdrop" and stop as soon as it becomes less than 1.1. This is because we have three different horse objects. So, we must test the condition for all three horses and stop the race as soon as any one horse gets within 1.1 meters. Clearly, the test condition must have three parts combined by using the world-level logical operator *and*. (We used a similar condition with the if-else statement in the Zeus world in section 6-1.) The condition for the horse race is shown in Figure 8-1-5:



Figure 8-1-5. A multiple condition to end the race

Random selection

Now, we can tackle the task of deciding which horse moves next. This is where random selection is needed. In Tips & Techniques 6, the *random number* question was introduced. But, in this problem we need to randomly select a horse to move forward in such a way as to give each horse a "fighting chance" to win the race. (After all, we don't want to "fix" the race.) So, instead of using the *random number* question, let's use the world-level question "*choose true n of the time*."

It seems reasonable to use a value of 1/3 for *n*, so that each horse will be chosen 1/3 of the time. The other 2/3 of the time, that horse will not move. As shown in Figure 8-1-6, we start the random selection with the first horse, whose name is Horse. Horse will be selected 1/3 of the time. But, what about when Horse is not selected? When Horse is not selected, the Else-part takes over and a selection must be made from Horse2 or Horse3. Once again, to decide which of

these horses should move, “choose true *n of the time*” is used. But, this time a value of $\frac{1}{2}$ is entered for **n** because we want the two horses to each have an equal chance to be selected.

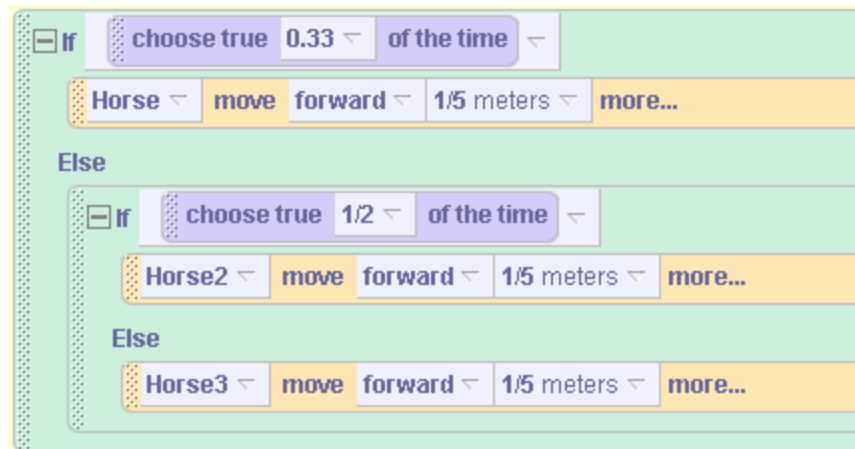


Figure 8-1-6. Random selection of a horse

Now that we have figured out how to write the Boolean condition and how to randomly select one of the horses to move forward each repetition of the loop, we can put all of the code together, as seen in Figure 8-1-7.

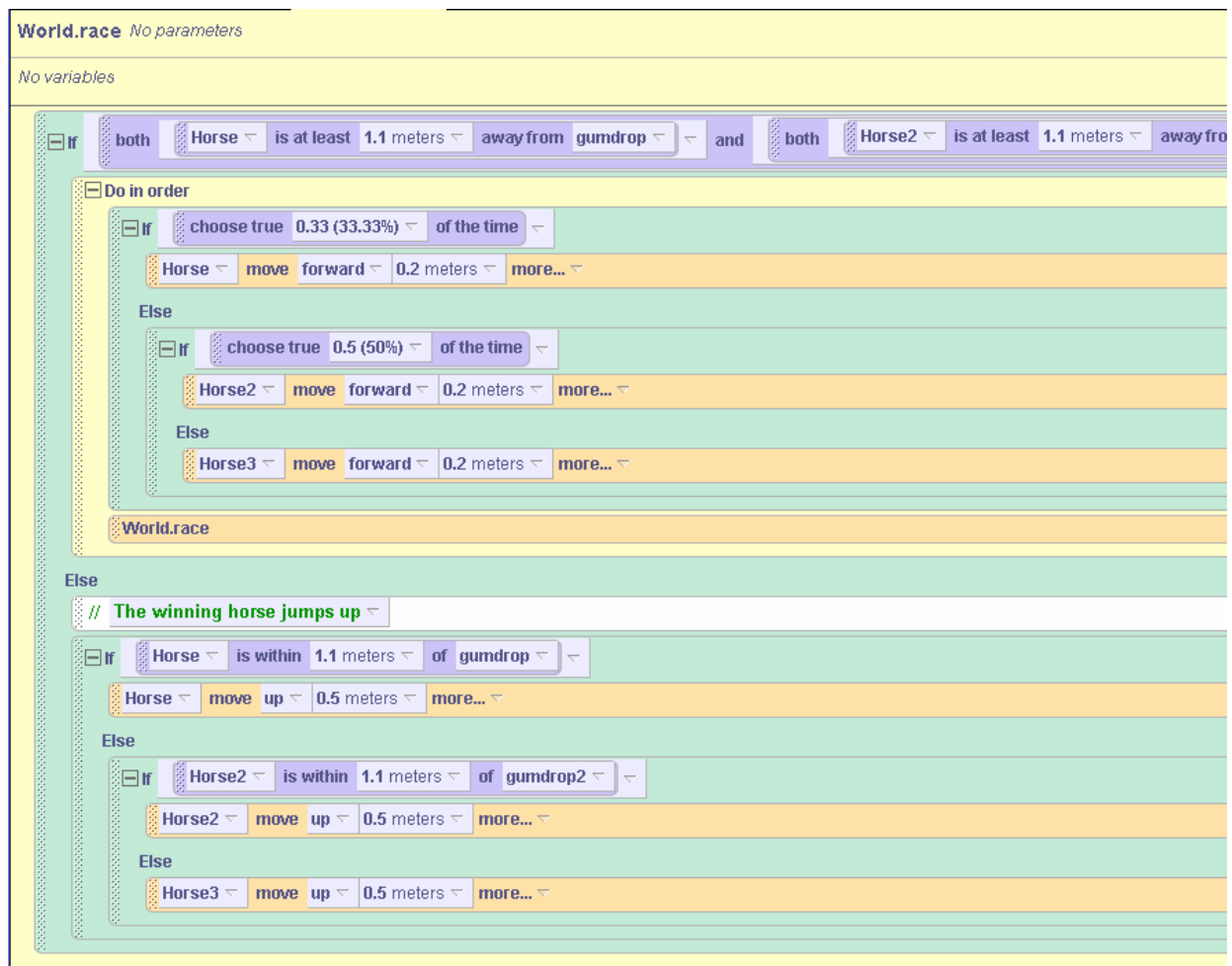


Figure 8-1-7. Horse Race Animation

Nested within the *Else-part* of the horse race code is another *If-else* statement to determine which horse has won the race. The winner simply moves up 0.50 meters into the air. This is not a very realistic end to a race, but at least it is easy to see which horse won! Because of the random selection used in this program, different horses should win the race if the program is run several times.

8-1 Exercises

1. ButterflyChase

Use recursion to create an animation for the following world:



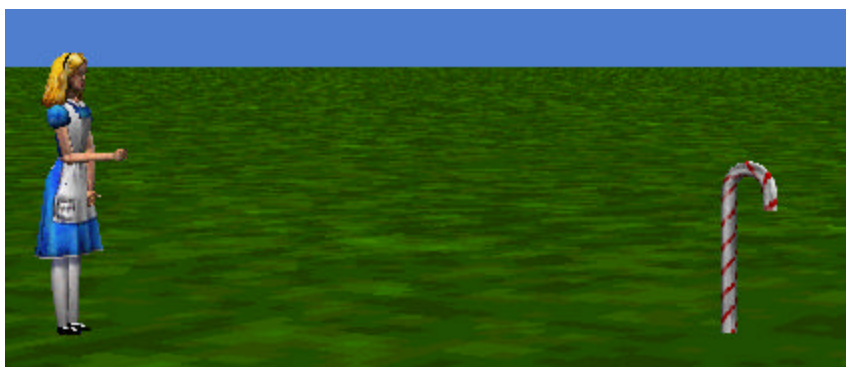
A WhiteRabbit is chasing after a butterfly. The world to build is similar to the chase scene from the text except in two important respects:

- The White Rabbit must always remain on the ground (rabbits do not fly). Use the “*only affect yaw=true*” option of the *point at* instruction.
- In order to prevent the butterfly from flying away (too high or too low), the butterfly’s up-down coordinate should be set to a random value between 0 and 1 rather than a slight change from its current up-down value.

When the WhiteRabbit gets close enough to the butterfly, have him catch the butterfly in his net.

2. MidasTouch1

A magician has cast a spell on Alice. The effect of the magician’s spell is to give Alice the Midas touch. Everything she touches turns to gold.



Alice is facing the candycane. (Use *PointAt* to make this alignment.) Write a recursive method, named *CheckCandy*, that checks whether Alice’s right hand is very close to the candycane (within 1 meter). If it is, Alice bends over to touch the candycane. After Alice touches the candycane, the candycane turns to gold (color changes to yellow). If Alice is not yet close enough to the candycane to be able to touch it, Alice moves a small distance forward (one-tenth meter) and the *CheckCandy* method is recursively called.

3. **MidasTouch2.** Create a second version of the MidasTouch1 world. This new version will be interactive. The idea is allow the user to guide the movement of Alice towards a candy cane. To make this a bit more challenging, Alice should **NOT** be pointing towards the candycane in the initial scene. This will require the *CheckCandy* method be modified so that whenever Alice is close enough to the candycane to touch it, she first turns to point towards the candycane before bending over to touch it.

Use the left and right arrow keys to turn Alice left or right. Create 2 methods: *TurnRight* and *TurnLeft*. The methods should turn Alice 0.05 meters to the right or left when the user presses the right-arrow key or the left-arrow key. These methods will allow the user to guide Alice towards a candycane.

Hint: It is possible that Alice will wander out of the range of the camera. There are two possible solutions to this problem. One is to make the camera point at Alice each time she moves. The other is to make the camera's vehicle be Alice!

4. Horse Race1

Create and modify the horserace world in this section. The modified horse race should:

- (a) The horses should move in a more realistic animation, rather than simply sliding along the ground as in the example world.
- (b) All 3 horses should move simultaneously forward, but a different random distance.

5. Horse Race2

Any code that uses tail-recursion can be written using a while loop. Create a horse race implementation that uses a *while* loop instead of tail-recursion.

8-2 Another Flavor of Recursion

In the section 8-1, a general form of recursion was introduced. To summarize what we learned in section 8-1, we can say that recursion depends on a decision statement that evaluates a condition. Depending on the results of that decision, a method is either finished or will be recursively called. If the method is recursively called, the same decision needs to be considered once more to determine whether another repetition will occur. This repetition may go on and on until, eventually, the tested condition changes and the method ends. In this section, we examine a second kind of recursion. The goal in this section is to reinforce the concept of recursion by presenting a different kind of problem to be solved recursively.

A second form recursion

The second form of recursion depends on an ability to break a problem down into smaller and smaller sub-problems. Once the smaller sub-problems are solved, the solutions to the smaller sub-problems are used to cooperatively solve the larger problem. As an analogy, suppose you have an emergency situation where you need \$60, but you have only \$10. So, you ask your best friend to lend you the rest (\$50). Your best friend has only \$10, but says he will ask another friend for the remaining amount (\$40). This story continues like this...

You need \$60 and have \$10 so you ask a friend to borrow \$50
Friend has \$10 and asks another friend to borrow \$40
Friend has \$10 and asks another friend to borrow \$30
Friend has \$10 and asks another friend to borrow \$20
Friend has \$10 and asks another friend to borrow \$10
Friend loans \$10

Collectively, each friend lends \$10 back up the stream of requests and the problem of borrowing enough money to have a total of \$50 is solved. (Of course, you now have the problem of paying back the money!) Notice that each friend has a similar problem to your problem – but the amount of money each friend needs to borrow is successively smaller. This is what we mean by breaking the problem down into smaller and smaller sub-problems. And, when the smallest problem (the **base case**) is solved, the solution is passed back up the line, and that solution is passed backup the line ...and so on. Collectively, the entire problem is solved.

To illustrate this form of recursion in terms of designing and writing a program, it is perhaps best to look at an example world.

Towers of Hanoi puzzle

The problem to be considered is the Towers of Hanoi, as illustrated in Figure 8-2-1. The Towers of Hanoi is a legendary puzzle. In this puzzle, disks of varying widths have been placed on a tower. To solve the puzzle, all the disks must be moved from one tower to one of the other towers, following certain strict rules.

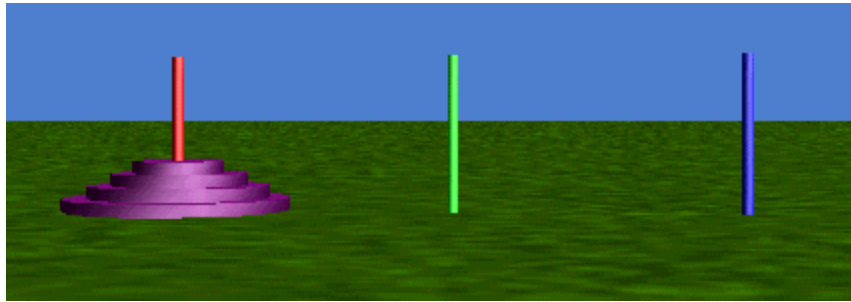


Figure 8-2-1. The Towers of Hanoi

In the world shown in Figure 8-2-1, four disks on the leftmost tower. Each tower is exactly 5 meters from its nearest neighbor. And, each disk is exactly 0.2 meters in height. To make it easier to describe this puzzle and our solution to the puzzle, let's number the purple disks 1, 2, 3, 4 ...in order of increasing size. The smallest disk is disk 1 at the top and the largest disk is disk 4 at the bottom of the stack. Also, let's name the towers "one, two, three" left to right from the camera point of view. Our goal is to move all of the disks from the leftmost tower, tower one, to another tower. In this example, we will move the disks to the rightmost tower (tower 3).

In this puzzle, the rules that must be strictly followed are:

- 1) Only one disk may be moved at a time.
- 2) A larger disk may never be placed on top of a smaller disk.

So, for example, the following attempts to solve the problem are illegal:

- 1) Simultaneously move disks 1 and 2 from tower one to tower two. (This move violates rule #1.)
- 2) Move disk 1 (the smallest disk) from tower one to tower two. Then move disk 2 from tower one to tower two. (This would result in disk 2 being on top of disk 1, thus violating rule #2.)

In the ancient story about this puzzle, there were 64 disks on the Towers in Hanoi. Solving the puzzle with 64 disks would be a huge task and would take much too long to run! (In fact, assuming that it takes 1 second to move a disk from one tower to another, it would take many, many centuries to run!) However, we can use just 4 disks to illustrate a solution to the puzzle that uses recursion. Most people can solve the 4-disk puzzle in just a few minutes, so this will allow a quick check for a correct solution.

Two requirements

We want to solve this puzzle for 4 disks using the second form of recursion, where we will break the problem down into smaller and smaller sub-problems. To use this form of recursion, two requirements must be met. **The first requirement is that we must assume we know how to solve the problem for a smaller sub-problem.** Well, let's assume that we do know a solution for solving the problem for 3 disks. If we know how to solve the problem of moving 3 disks, it would be quite easy to write a program to solve it for 1 more disk (4 disks). The following steps would work:

- 1) Move the 3 disks (imagining the solution for the puzzle with only 3 disks is already known) from tower one to tower two. See Figure 8-2-2(a).

- 2) Move the last disk, disk 4, from tower one to tower three. See Figure 8-2-2(b). (Remember this move is now safe, as all of the 3 smaller disks are now located on tower two.)
- 3) Move the 3 towers (again, imagining the solution is already known) from tower two to tower three.

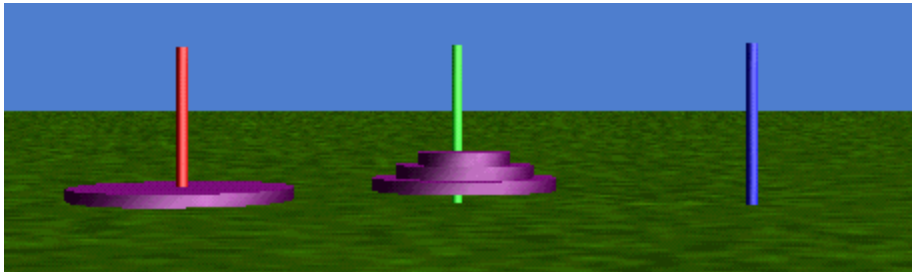


Figure 8-2-2(a). After step 1

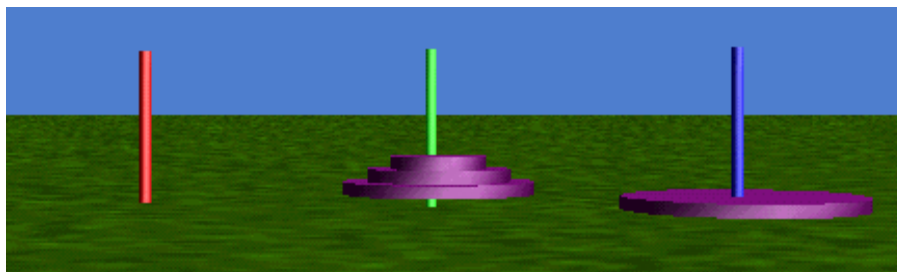


Figure 8-2-2(b). After step 2

The second requirement is that we must have a **base case**. A base case is the simplest possible situation where the solution is unquestionably obvious. Since the solution to the base case is obvious, the base case does not require a solution to a smaller sub-problem. The obvious “base case” in the Towers of Hanoi puzzle is the situation where we have only 1 disk. In order to move 1 disk, we can just move it, as there are no smaller disks than disk 1, it can always be moved from whichever tower to whichever tower without question!

In summary, two requirements must be met when using this second flavor of recursion:

- 1) Use the solution to a smaller sub-problem to help solve the larger problem.
- 2) Have a base case, which does not require a solution to a smaller sub-problem.

Towers Method

Putting these two ideas together, we can now write a method, named *Towers*, to animate a solution to the Towers of Hanoi puzzle. The method will have instructions to move some number (**n**) of disks from a **source tower** (the tower where the disks are currently located) to a **target tower** (the tower where the disks will be located when done). In the process of moving the disks from source tower to target tower, a **spare tower** (the tower that is neither source nor target) will be used as a temporary holder for towers on their journey from source to target.

What information has to be sent to parameters in the *Towers* method so it can do its work? Well, the **number of disks to be moved** must be known along with **which tower is the source**, **which tower is the target**, and **which tower is the spare**. To provide this information to the *Towers* method, four parameters will be needed: *howmany*, *source*, *target*, and *spare*. Here is the storyboard:

```
Towers (howmany, source, target, spare)
  if howmany equals 1
    move the disk (the smallest one) from the source to the target.
  otherwise
    Recursively call Towers() to move howmany-1 disks from source to
    spare (using target as spare).
    Move disk # howmany from the source to the target.
    (This is ok because all smaller disks are now located on
    the spare peg, after step 1).)
    Recursively call Towers() to move howmany-1 disks from the spare to
    the target (using the source as the spare)
```

We know this looks kind of complicated; but hang in there -- it really is not too difficult. All we are saying is: (1) Move all but one of the disks from the source to the spare tower. (2) Now that only one disk is left on the source, move it to the target tower. (3) Now, move all the disks that have been temporarily stored on the spare tower to the target tower. Based on this storyboard, the code for the *Towers* method is shown in figure 8-2-3.

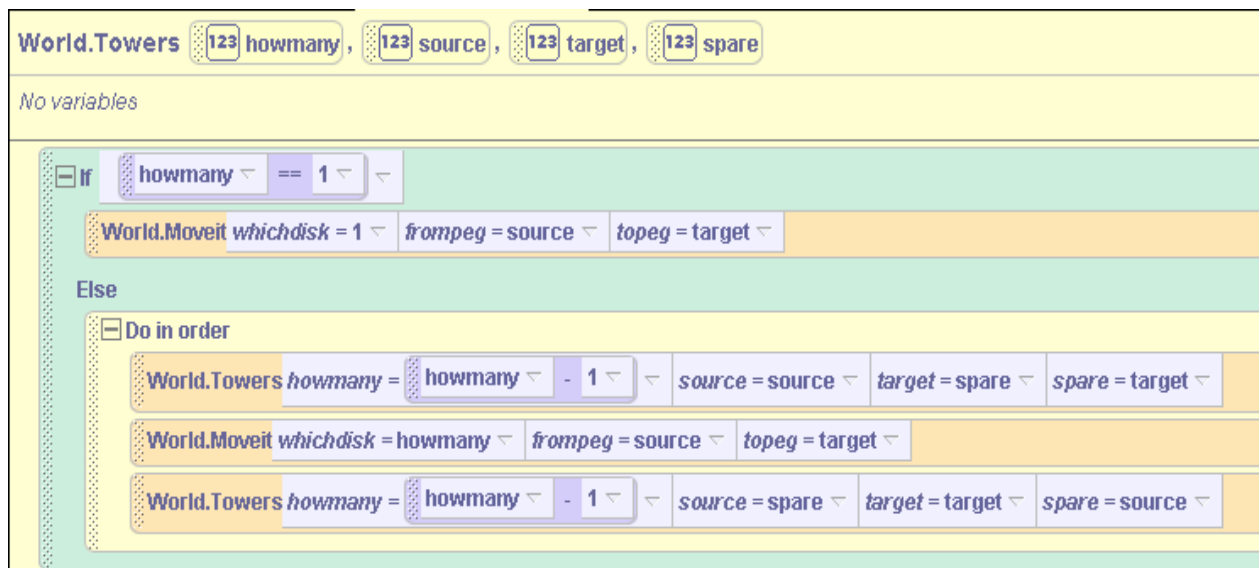


Figure 8-2-3. The Towers method

Special *Moveit* Method

No doubt as you looked at the code in Figure 8-2-3, you realized that we called a method named *Moveit*. (In fact, we called the *Moveit* method a couple of times.) So, exactly what does the *Moveit* method do? Well, *Moveit* is a special method we wrote to move a specific disk from one tower to another tower. The *Moveit* method needs 3 parameters because it needs to know: which disk is to be moved, the source tower, and the target tower. A storyboard for *Moveit* could be:

```
Do in order
```

```
    Lift the disk up above the top of the source peg  
    Move it (forward or back) to a location above the target peg.  
    Lower the disk down onto the target peg.
```

This sounds easy enough. But, how high should the disk be lifted? Each disk is at a different initial height on the tower. So, it will be necessary to raise each disk a different amount. (Note that disk1 has been placed on top, disk2 immediately below it, disk3 immediately below disk2, and disk4 immediately below disk3.) Also, in our example world, we made each disk be 0.2 meters in height. Then, we determined the height to lift each disk by trial and error. In the sample world, it was necessary to lift the first disk approximately 2.0 meters, disk2 approximately 2.2 meters, disk3 approximately 2.4 meters, and disk4 approximately 2.6 meters (to “clear” the tower).

Once the lift-height is determined for each disk, then a plan can be created for carrying out the lift to the appropriate height for the particular disk. One possibility is to pass a parameter (to the *Moveit* method) that contains the id number of the disk to be moved. If the disk id is 1, move Disk1, if the disk id is 2, move Disk2, and so on. Then, a cascading *If* statement can be used to check on the id of the disk passed in and lift the appropriate disk the appropriate amount. The storyboard (for the *If* statement) would look something like this:

```
If whichdisk is 1 then  
    Move Disk1 Up 2.0 meters  
Else  
    If whichdisk is 2 then  
        Move Disk2 Up 2.2 meters  
    Else  
        If whichdisk is 3 then  
            Move Disk3 Up 2.4 meters  
        Else  
            Move Disk4 Up 2.6 meters
```

We thought about this for a while, realizing that our code is a bit awkward. Then, we came up with a more clever approach using a nifty mathematical expression. Note that the amount any disk should move Up is $1.8 + 0.2 * \text{whichdisk}$. So, it would seem that the above storyboard could be condensed into just one step:

```
Move the appropriate disk Up  $1.8 + 0.2 * \text{whichdisk}$ 
```

Conversion Question

One question still needs to be answered: "How is Alice to be told the name of the disk object to be moved when only the disk id number is known?" The problem is there is no easy way to

convert the disk *id* number to the *disk-object* name. What is needed is a conversion method that takes the id as a parameter and then returns the appropriate name of the disk object to move. Such a conversion method can be written in Alice using a question. The *which* question is illustrated in Figure 8-2-4. Note that this question returns an object – the disk to be moved.

In this example, each *If* statement includes an instruction containing the keyword *return*. The return statement means that the function will be sending information back to the method that called it. For instance, suppose "*which(i = 2)*" is called, the information that will be returned is "*disk2*". So, the *which* question provides a way to convert from an id number (the *whichdisk* parameter) to an object name so the *Moveit* method will know which object is to be moved!

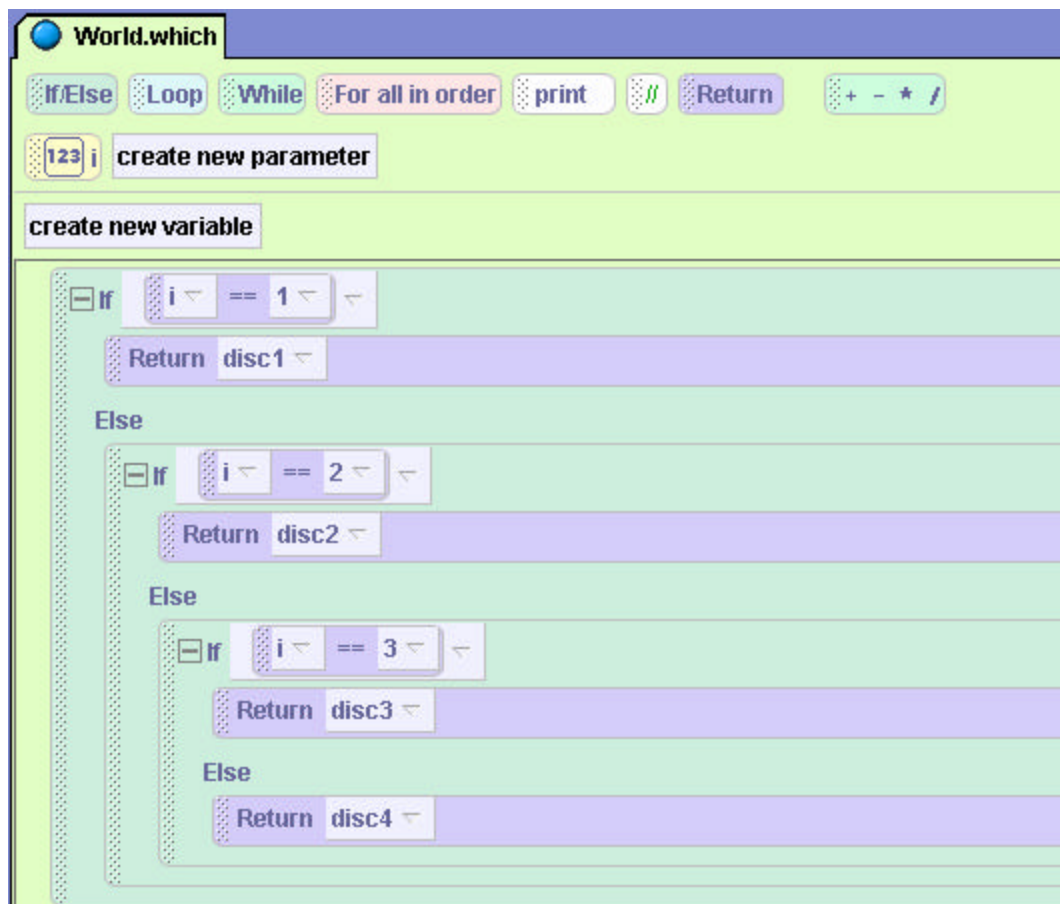


Figure 8-2-4. A World-level Conversion Question

An instruction is now written in the *Moveit* method that calls the *which* question to determine which disk to move, as illustrated in Figure 8-2-5.



Figure 8-2-5. Invoking the question to determine which disk to move

In this instruction, *which(i = id)* is a **call** to the *which* function. When executed, *which(i = id)* will select the appropriate disk and return the name of the object that will move it the appropriate amount, as explained in the earlier expression.

The next piece of information we need to know is the distance (forward or back) to move the disk (in order to move it over to the destination tower). The towers are purposely positioned exactly 5.0 meters from one another. So, one possibility is to have several if statements, covering all possible cases of source and destination towers – perhaps something like:

```

if the Source peg is 1
  if the target peg is 2
    move the appropriate disk 5 meters forward
  else
    move the appropriate disk 10 meters forward
else
  if the Source peg is 2
    if the target peg is 3
      move the appropriate disk 5 meters forward
    else
      move the appropriate disk 5 meters back
  else (the source peg is 3)
    if the target peg is 1
      move the appropriate disk 10 meters back
    else
      move the appropriate disk 5 meters back

```

While this storyboard is correct, the nested if – else structures make it quite confusing. Once again, we gave some thought to what we might do to come up with an easier way to figure out the distance to move forward or back. With a bit of thought, you can see that the forward amount to move is: $(\text{totower} - \text{fromtower}) * 5.0$. (5 is the distance between two adjacent towers) Also, notice that moving forward -5.0 meters is the same as moving back 5.0 meters. With this insight, the instruction in Figure 8-2-6 can be written:



Figure 8-2-6. Moving the disk the appropriate amount forward/back

The last step in the *Moveit* action is to move the disk back down onto the tower. This instruction should simply do the opposite of what was done in step 1. The complete *Moveit* method appears in Figure 8-2-7.

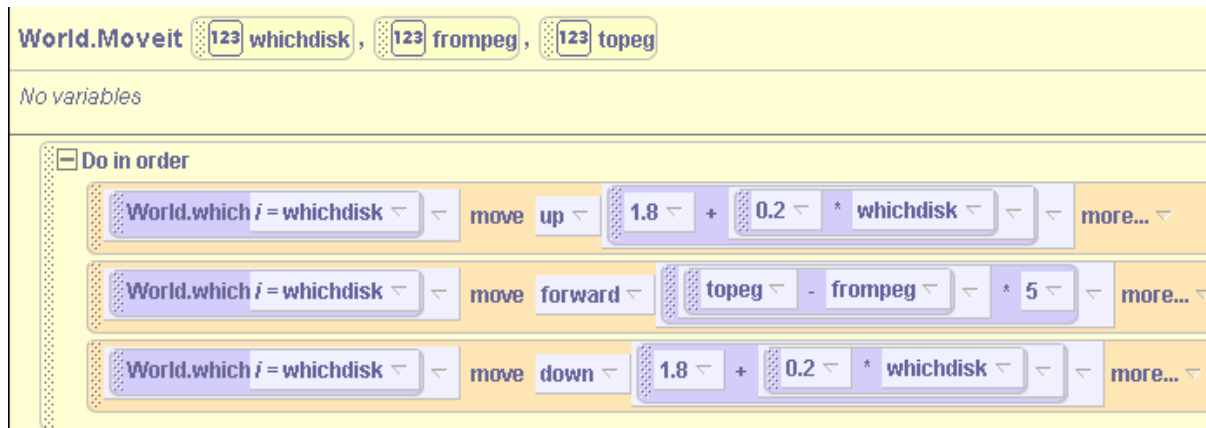


Figure 8-2-7. The code for *Moveit*

The *Towers* method was presented earlier in Figure 8-2-3. Now, with the *Moveit* method completed, all that remains is to call the *Towers* method when the world starts. The following link in the Events Editor will work:

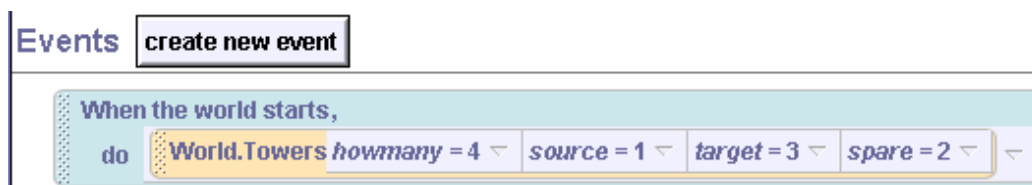


Figure 8-2-8. Invoking the *Towers* method

Comparison of two forms of recursion

The form of recursion introduced in chapter 8 section 1 is technically known as **generative** recursion. In the example presented in section 8-1, the shark is chasing the goldfish and if the goldfish is more than 0.5 meter from the shark, both the shark and the goldfish are moved. After both objects have moved, the question is asked again: Is the shark more than 0.5 meters from the goldfish? Such recursion is technically known as “generative” because another execution may be “generated” each time the result of the decision is true. We do not know whether executing the method one more time will produce an acceptable solution to the problem (getting the shark within 0.5 meter of the goldfish) until the method has run and the results can be tested again.

While this form of recursion is fairly useful and easy to implement, it has one serious flaw: ***It is often very difficult to determine that the program will ever end.*** How can we be sure that the shark will ever catch the goldfish? We are haunted by the thought that, by some random chance, the goldfish might constantly move far enough away from the shark that the shark would never catch up. (Actually, we expect in random behavior that sometimes the goldfish will move towards the shark, rather than away.) To show that the shark will catch the goldfish eventually, it is necessary to argue that the total amount of distance the shark moves (in each step) is greater than the total distance traveled by the goldfish in each step.

The second form of recursion, as presented in this section, is technically known as **structural** recursion. It is structural because it depends on the structure of a problem and its solution, breaking a problem down into smaller and smaller sub-problems. Many mathematicians (and

computer scientists interested in logic) often prefer this form of recursion because it is easier to show that the program does end – and that it ends with the correct solution. For example, with the Towers of Hanoi, we reasoned that we could:

- move 4 disks, if we can move 3 disks and then 1 disk (we know how to move 1 disk)
- move 3 disks, if we can move 2 disks and then 1 disk
- move 2 disks, if we can move 1 disk and then 1 disk
- move 1 disk (base case -- we know how to move 1 disk)

Clearly, the problem has been broken down into smaller and smaller sub-problems. And, we know that the program will end because we know that eventually the problem size will get down to the base case, which is moving one disk.

Infinite Recursion

As with *Loop* and *While* statements, it is possible to write an infinite recursive method. A recursive method can be infinite if the programmer unintentionally writes the method with an incorrect base case (or without a base case altogether) or if statements within the method do not change conditions in such a way as to ensure that the base case will eventually be reached. As a general rule, infinite recursion is to be avoided in your program code. We caution you to always plan your recursive methods to be reasonably certain that execution of the method will eventually reach a stopping point.

Naturally, there are exceptions. As seen with worlds using *Loop* and *While* statements, some animations are designed to run continuously until the program shuts down. To illustrate, let's revisit the Carousel animation world, presented in chapter 7, section 2. The method to animate the carousel is shown in Figure 8-2-9.

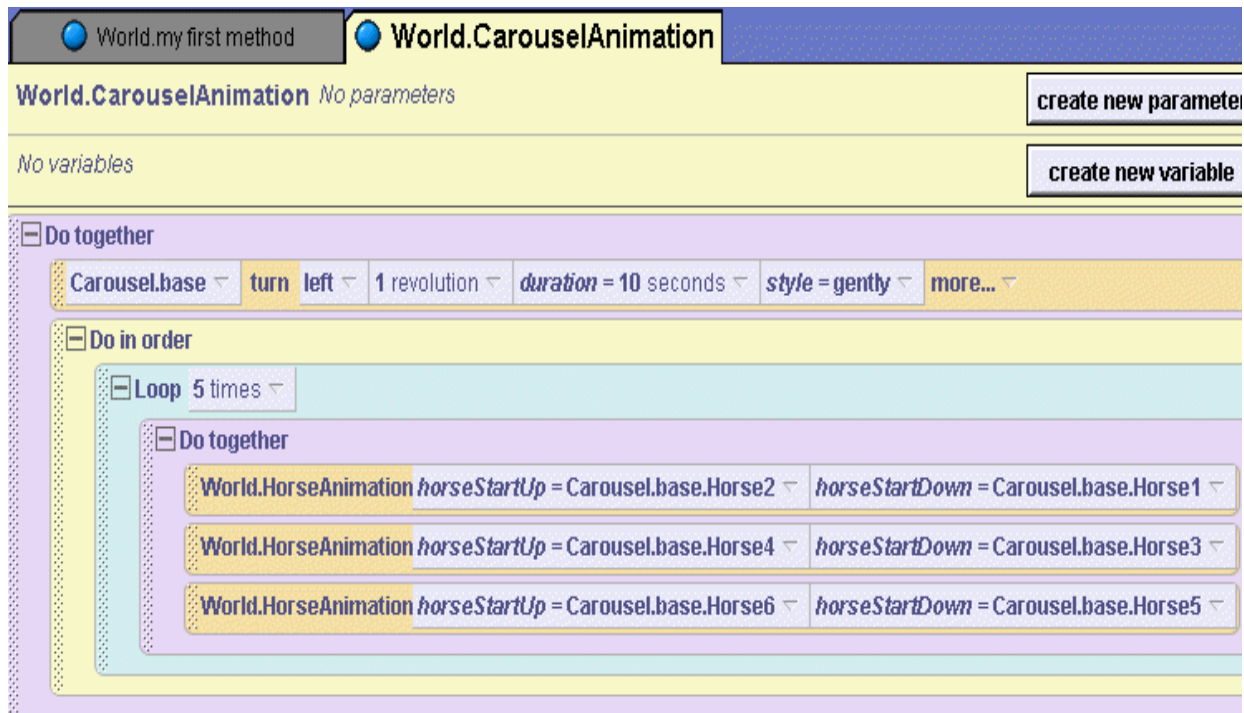


Figure 8-2-9. Carousel animation

A *Loop* statement was used to make the carousel go round forever, as shown in Figure 8-2-10.

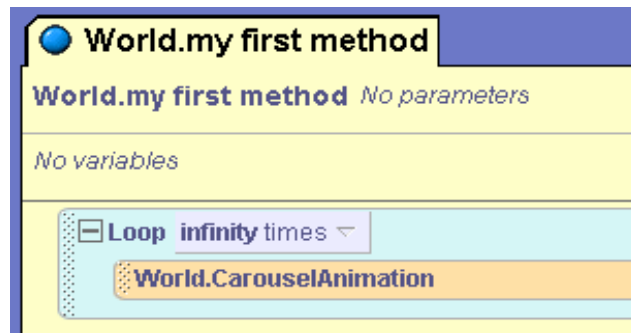


Figure 8-2-10. Infinite *Loop* statement

An alternate solution is to modify the *CarouselAnimation* method to make it infinitely recursive, as illustrated in Figure 8-2-11. Intentionally, no base case was used. With no base case, the recursive calls will occur again and again and the method will never end. When the program is run, the carousel will go round and round forever (until the program is shut down).

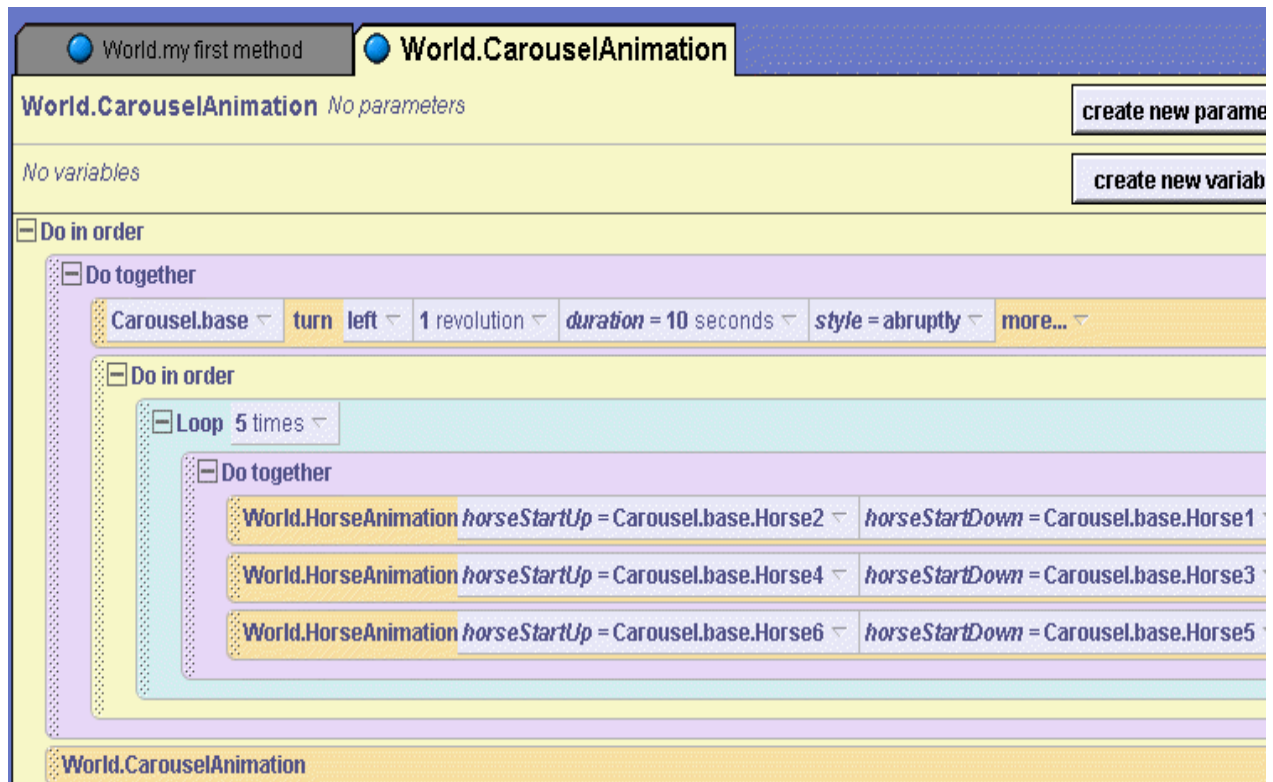


Figure 8-2-11. Infinitely recursive carousel animation

8-2 Exercises:

1. TowersOfHanoi

Create the Towers of Hanoi puzzle as described in the reading for this section. When you have it working, modify the event-trigger link that calls the Towers method so that it moves the disks from tower 1 to tower 2 (instead of tower 3).

2. Reversal

In the world below, the row of skeletons (Graveyard folder on the web gallery) are guarding the gate. Every so often in this world, the row of skeletons is to reverse order. This project is to animate the skeleton chain reversal using structural recursion. The storyboard goes something like the following:

Reverse method:

If the row of Skeletons is not yet reversed is more than one then
Reverse the row of Skeletons starting with the 2nd Skeleton (by recursively
calling Reverse
Move the Head Skeleton to the end of the list



The base case is when there is just one Skeleton in the row (that has not yet been reversed). Of course, a row of 1 Skeleton is already reversed!

The recursive case (for n Skeletons, where n is larger than 1) says to first reverse the last $n-1$ Skeletons, and then move the first Skeleton to the end of the list.

Implement the Skeleton reversal storyboard given above. The program you will write should be quite similar to the Towers of Hanoi program, including the *which* method.

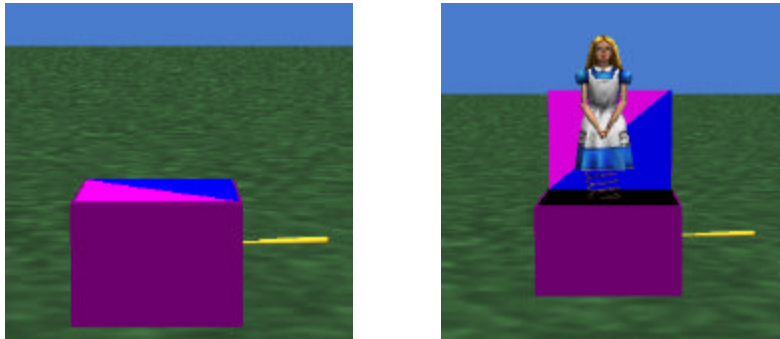
3. OnTheHour

Place a cuckoo clock in a new world. The idea of this animation is to have the cuckoo clock keep time (not in real-time, of course). The minute hand should go around on the face of the clock (perhaps 1 complete revolution should take about 30 seconds in real-time) and the pendulum should swing back and forth. When the minute hand has made 1 revolution (from 12 back to 12 on the face of the clock), then the hour hand should advance to the next hour, the doors should open, and the cuckoo bird (on the clock arm) should come out and chirp once for each hour of time that has elapsed. Then the bird should retreat inside the clock and the doors should close until the next hour has gone by. “All is well” as long as the clock is running -- which should continue until the user stops the animation. Use a recursive method to implement the animation.

4. Alice-in-the-box

Animations are popular in computer software created for children. For example, in a child’s game, a click on an object labeled with a letter of the alphabet causes the letter to be “spoken” and some animation occurs on screen. This world illustrates an example of an Alice-in-the-box animation for children’s software. As illustrated in the picture below left, the initial scene shows a closed box (composed of a cube with a square as its top). Inside the box, but hidden from

view, is Alice and a spring. Extending from the side of the box is a "crank" made from a baseball bat.



At the beginning of the animation, the box is closed. The crank continuously moves around and around and a "clinky" kind of sound could be playing throughout the running of the animation. Every 5 seconds, the box top opens and Alice on her spring pops out, bouncing around a few times (Note that a black square was used in the example above to simulate the depth of the box – although this is not necessary). When Alice stops bouncing around, she and the spring are lowered back into the box and the top closes. Use a recursive method to implement the animation.

8 Summary

In this chapter, the concept of using recursion as a mechanism for repetition was introduced. Recursion is a powerful tool for building more complex and interesting worlds. Throughout the rest of the book, example worlds will be significantly richer and more interesting than they have been in the past!

A chase scene example was used to demonstrate how to combine a *move to* instruction with the world-level *random number* question to enable *random motion*. Moving objects randomly makes it easier to build many different kinds of fun worlds. And, random motion makes it easier to understand how recursion works and why we want to use it. The chase scene example in this chapter demonstrates a kind of recursion said to be *generative*, because more executions are "generated" each time the result of the previous decision is true.

The famous Towers of Hanoi puzzle provided an example of the use of *structural* recursion, allowing us to compare generative recursion to structural recursion.

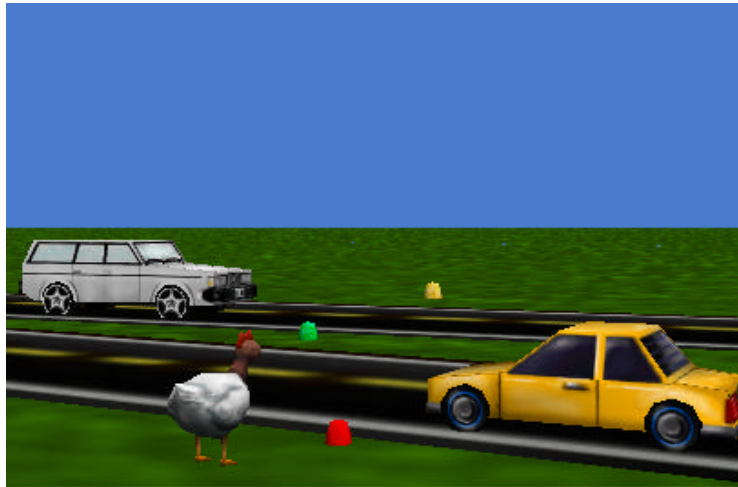
Important concepts in this chapter

- Recursion is most useful when we cannot use a Loop instruction because we do not know (at the time we are writing the program) how many loop iterations will be needed.
- Recursion is when a method calls itself.
- Recursion enables a method to be repeatedly invoked.
- Any recursive method must have at least one *base case*, where no recursive call is made. When the base case occurs, the recursion stops.
- Recursion may be *generative* or *structural*

8 Projects

1. Why did the chicken cross the road?

A popular child's riddle is "Why did the chicken cross the road?" Of course, there are many answers to this riddle. In this project, the chicken (who has a real sweet-tooth) crosses the road to eat the gumdrops along the way. Write a game animation where the player guides the chicken across the road to get to the gumdrops. Cars and other vehicles should move in both directions as the chicken tries to cross the road to where the gumdrops are located. Use arrow keys to make the chicken jump left, right, forward and back. Use the space bar to have the chicken peck at the gum drop. When the gum drop is pecked, it should disappear.



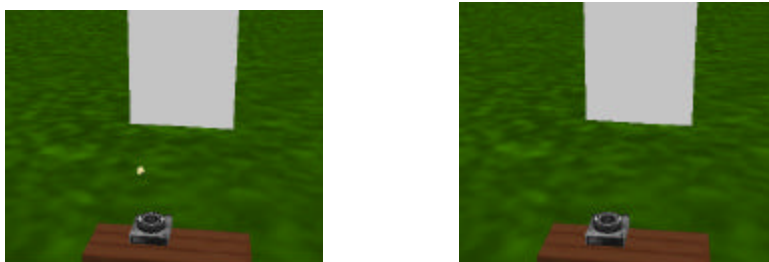
A recursive method is used to control the play of the game. If the chicken gets hit by a vehicle, the game is over (squish!). The game continues as long as the chicken has not managed to peck all the gumdrops and the chicken has not yet been squished by a vehicle. If the chicken manages to cross the road and peck at all the gumdrops along the way, the player wins the game. Signal the player's success by making 3D text "You Win" appear or by playing some triumphant sound.

Extra: Create three buttons that allow the player select the speed at which the vehicles move across the screen (slow, medium, and fast). The buttons should be visible at the beginning of the game. Once the player clicks on one of the buttons, the buttons disappear and the game begins.

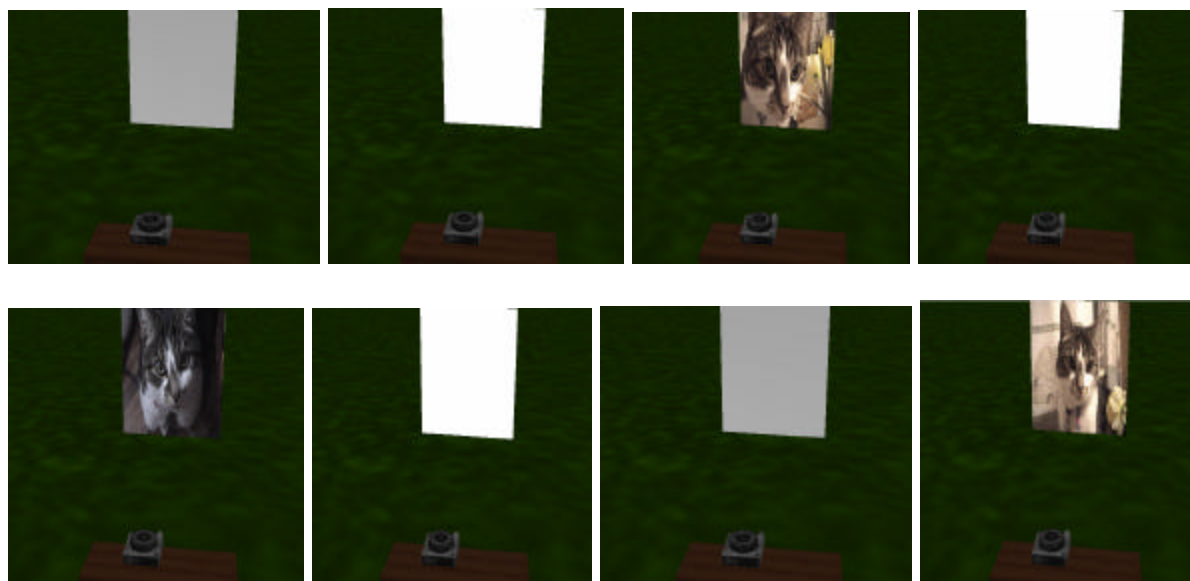
2. MidasTouchGame. An exercise in section 8-1 was named MidasTouch2. If you have already completed this exercise, begin with that world and modify it for this project. Otherwise, begin by creating the world for the MidasTouch2 exercise and then modify the animation for this project. In this project, convert the interactive MidasTouch world to work as a primitive kind of game. Add several candy canes to the scene. To win the game, the user has to guide Alice around the screen to touch each candy cane. When all have been touched and turned to gold, the game is over. Once a candy cane has turned to gold, it should become inactive. That is, Alice should not bend over to touch it again – even if she gets close enough to do so. The CheckCandy method should be modified to use a parameter that specifies which candy cane is to be checked.

3. NeverEndingSlideShow

This project makes use of the technique of changing the skin of an object. See Tips & Techniques 8. In this world, you are to create a never-ending slide show. Set up a new world with a *square* (from the Shapes gallery) as the screen, the *slide projector* on a *table*, and a *lightbulb* in front of the screen, as shown in the image below left. The lightbulb is for the purpose of lighting up the screen (simulating light from the projector). Then, make the lightbulb invisible by setting its *isShowing* property to *false*. The resulting initial scene should appear as shown in the image below right.



For an added sense of realism, start with the world's light on and then dim the light before turning on the projector. (Dimming the lights was discussed in the BugBand world presented in section 3-2.) The slide show in your project must display at least 4 different slides. Numbering the slides and using an *if* statement may help you create a method to change slides. Between each slide, the screen must go blank, the projector light must flicker (change the color of the light), and the slide projector's tray of slides must rotate. The slides are to change continuously (meaning once all 4 have been seen, the show should go back to slide 1 and start the show over). The sequence of images below show a sample slide show.



4. TowersOfHanoi (*challenging*)

Write the Towers of Hanoi animation program presented as an example section 8-2 using **while** loops instead of recursion. (Hint: You will need 2 **while** loops, one located within the other one.)

Tips & Techniques 8

Texture Maps

Objects displayed in Alice are covered with a texture maps to provide a sense of realness to the object. For example, the Ground surface is covered with a texture map named Ground.TextureMap and a Pinata object is covered with a customized texture map named Pinata.Material #21_CL as can be seen in Figure T-8-1.

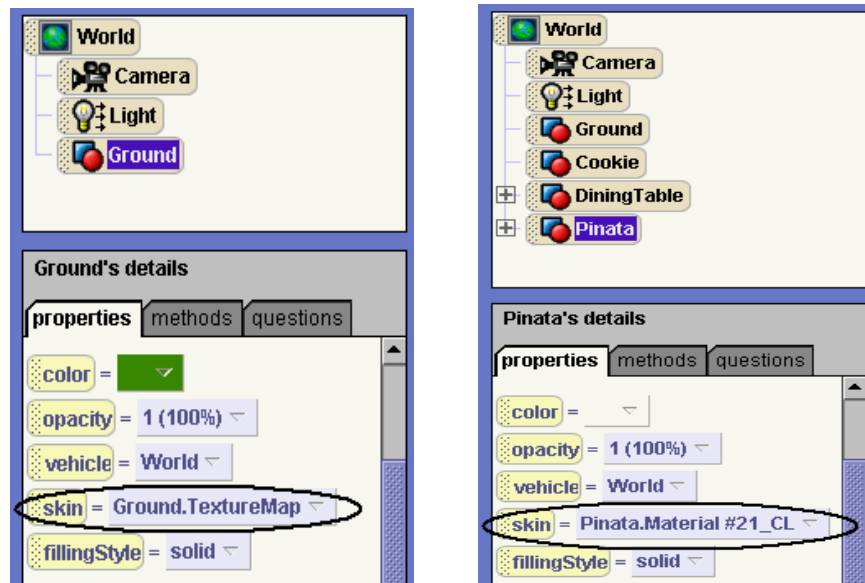


Figure T-8-1. Texture maps used as skin

A graphic file (.gif, .bmp, .jpg, .tif) can be used to give an object a different appearance. For example, in the world shown in Figure T-8-2 a plate object is positioned on the table. The plate has a rather ordinary appearance. We want the plate to look like a cookie instead.

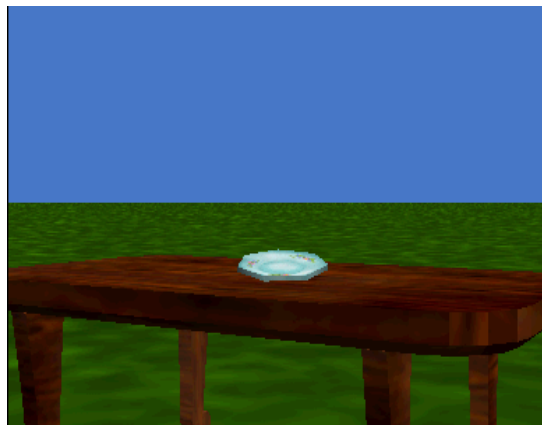


Figure T-8-2. Plate before texture change

To change the appearance of the plate, two steps are required. The first step is to import a texture map that we intend to use for that object. In this example, we selected the plate object,

clicked the import **texture map button**, and then selected *cookie.gif* to be used as the texture. Figure T-8-3 illustrates the importing step.

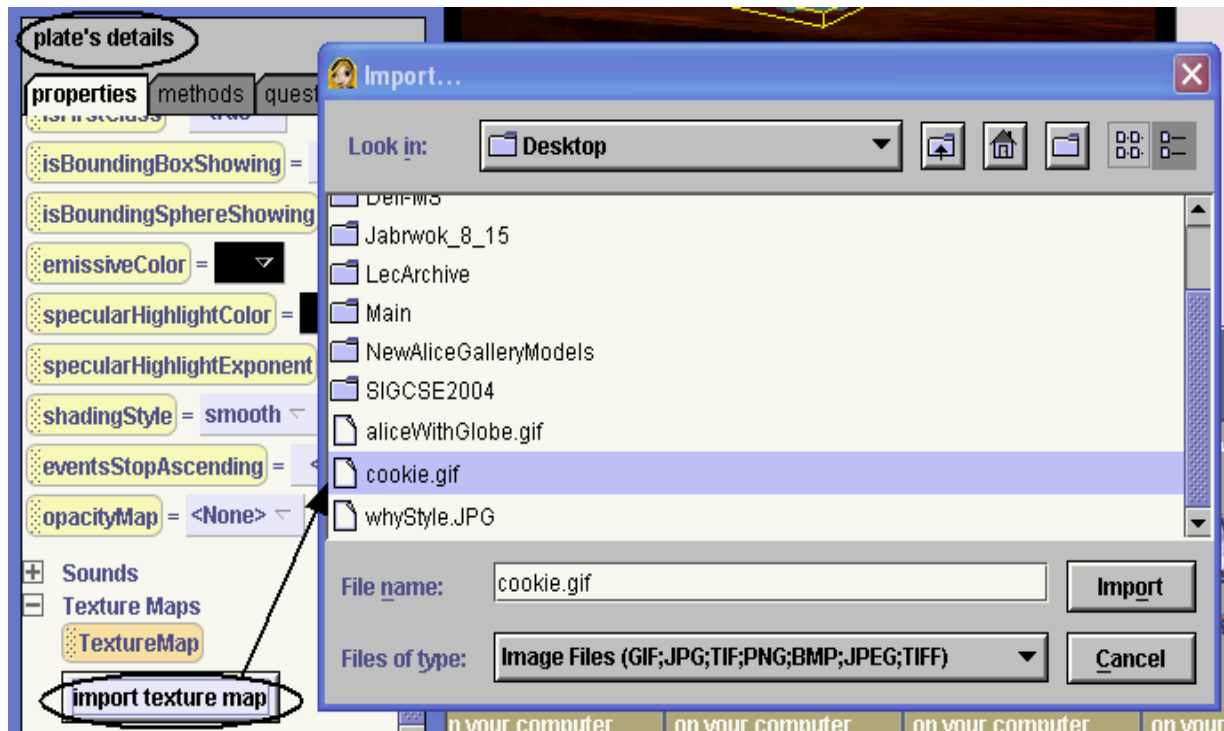


Figure T-8-3. Importing a texture map

The second step is to set the skin property to use the new texture map, as shown in Figure T-8-4.

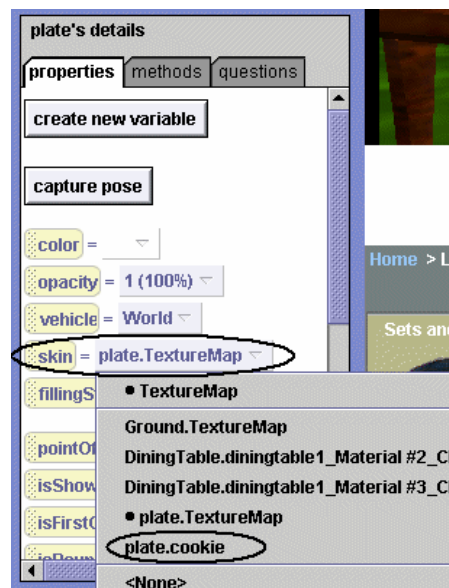


Figure T-8-4. Changing the texture map used for the skin

The result is seen in Figure T-9-5.



Figure T-8-5. Cookie plate

It is also possible to write a program that includes instructions to change the texture map for an object at run-time. An instruction to change the texture map of an object is written by dragging the skin tile from the Properties list into the editor. Then, the texture map is used as a parameter to tell Alice which texture map to apply to the object while the program is running.

For example, Figure T-8-6 shows an aqua-blue rock texture. If the rock texture has been imported as a texture map for the Ground, then we can change the appearance of the Ground at runtime. The instructions illustrate setting the skin to the blue rock texture and then changing the color. The result is a very different ground landscape, as shown on the right of Figure T-8-6.



Figure T-8-6. Setting the skin of the Ground at run-time

9 Lists and List Processing

This chapter will explore a different aspect of programming – that of working with data structures. In previous chapters, the focus has been on program instructions -- the actions used to animate objects in a virtual world. However, this chapter (as well as the next) will focus on organizing or grouping information. Up to now, worlds were typically made up of only a few objects. But what if an animation were to be created where 20 soldiers were marching? It would be necessary to create at least 20 animation instructions! One way to handle this situation is to collect all the objects into an organizing structure that allows the programmer to work either with the group as a whole or with one item at a time.

In computer science, there are many organizing structures used to create a collection of objects. Some examples of organizing structures are lists, arrays (one-dimensional or multi-dimensional), vectors, sets, and bags. Each kind of structure offers different capabilities or features. In Alice, lists and one-dimensional arrays are used as organizing structures. Lists are introduced in this chapter. A list is typically a collection of several objects of the same type.

Section 9-1 begins with a demonstration of how to create a list in Alice. We then look at how to *iterate* through the list (called the iterator design pattern). In Alice, there are two program constructs for iterating through a list. The first is *for all in order* – a sequential iteration, somewhat like “walking your fingers” down a list of items on a written page, looking at each item one at a time. (This is what is done in object-oriented languages such as C++ or Java). The second program construct is *for all together* – a simultaneous iteration where some action is taken for all items in the list simultaneously, somewhat like “putting all your eggs in one basket.”

Section 9-2 illustrates another use of lists, that of searching through a list to find an element of the list that possesses a certain property. In standard languages, this might be illustrated by searching through a list for an element that contains a certain key value. The concept is illustrated in this chapter by means of an arcade-style game.

Special notes to instructors:

- 1) We believe it is necessary to cover the topics in section 3-2 before lists. The reason is that iterating through a list of objects in Alice requires that an object from the list be passed to a method, which can then invoke animations on that object. Passing objects as parameters is discussed in section 3-2.
- 2) The iterator design pattern (and even lists themselves) is often not part of the introductory computer science curriculum. Alice’s handling of the messy details (how the list is built, and how iteration works) make the topic much easier to handle by beginning programmers.
- 3) In the examples presented in this chapter, lists were constructed before the program was run and were not changed during execution. Lists can change in Alice, but this feature has not yet been added to examples in this text.

9-1 Lists

Lists are one of the most popular ways to organize information. Examples of specialized lists such as stacks, queues, linked-lists, and doubly linked lists can be found in thousands of software applications. In Alice, a list (generally) contains objects of a similar type. In this section we will look at how to create a list and then *iterate through* the list. Iterating through a list is for the purpose of either looking for a particular item in the list or taking some action with each item in the list.

Creating a list

Before a list can be used in a program, a list must be created. As an example of a list, let's suppose we have an initial scene made up of five of the famous Rockette dancers, as shown in Figure 9-1-1.

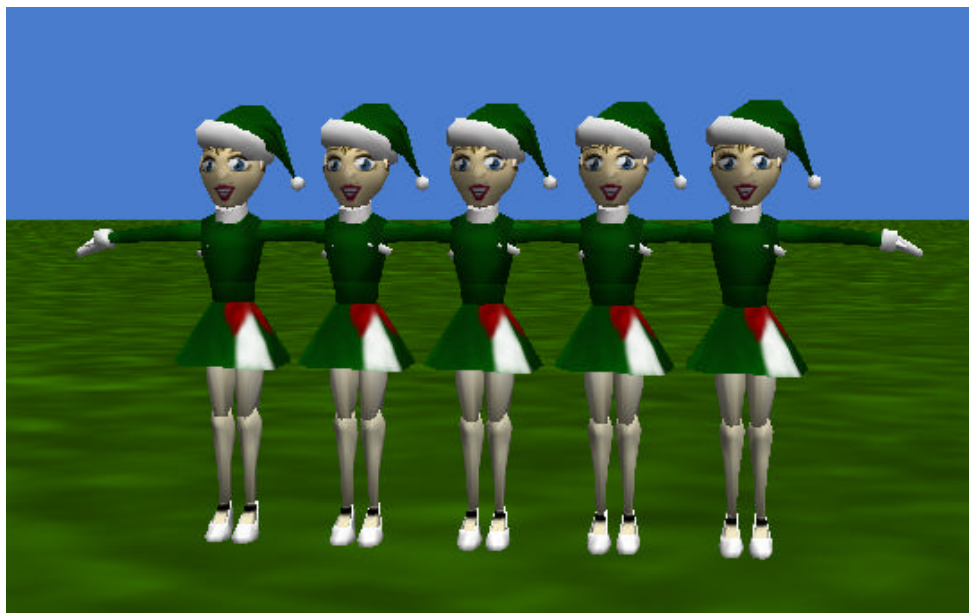


Figure 9-1-1. The Rockettes.

We would like to create a list variable that will be the name of the collection of the five Rockette objects. This list variable will be very useful when we create an animation. We can use the list to make all the Rockettes perform a dance routine. To create a list variable, select World in the Object tree and then click on its properties tab. Click on “create new variable”, as is done in Figure 9-1-2.

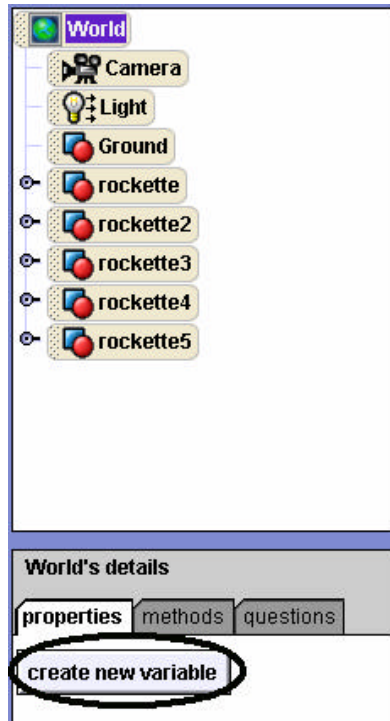


Figure 9-1-2. Creating a list variable

In the popup dialog box, we named the variable “*dancers*” and selected its type as *Object*. **The key action here is to check the box marked *make a list*.** Then, click the tile labeled “**new item**” five times to enter each rockette object, as seen in Figure 9-1-3. Clicking *Okay* creates the list.

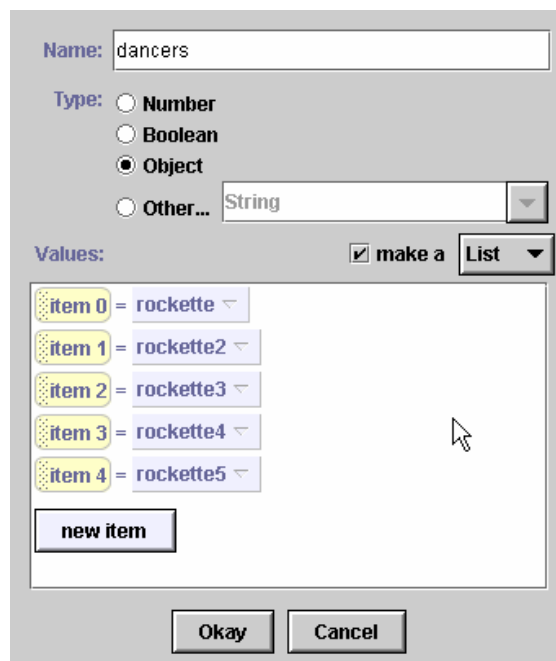


Figure 9-1-3. Creating a list of *dancers*.

For all in order -- Iterating sequentially through a list

One of the most useful operations that can be done with a list is to iterate sequentially through the list. This means that each item in the list will be instructed to perform the same instruction, one after the other. This is like a mail delivery person walking down the street and placing mail in each mailbox. This kind of iteration through a list is sometimes called “walking” or “traversing” a list. Alice provides a special *for all in order* construct that works with one item at a time from a list. As an example, let’s have each Rockette, from left to right, kick up her right leg as part of the dance routine. The storyboard would look like this:

```
For all dancers in order
    item_from_dancers kickUpRightLeg
```

The *for all in order* construct is exactly what is needed. The *for all in order* tile is dragged into the editor and the expression `World.dancers` is selected. The result is seen in Figure 9-1-4.

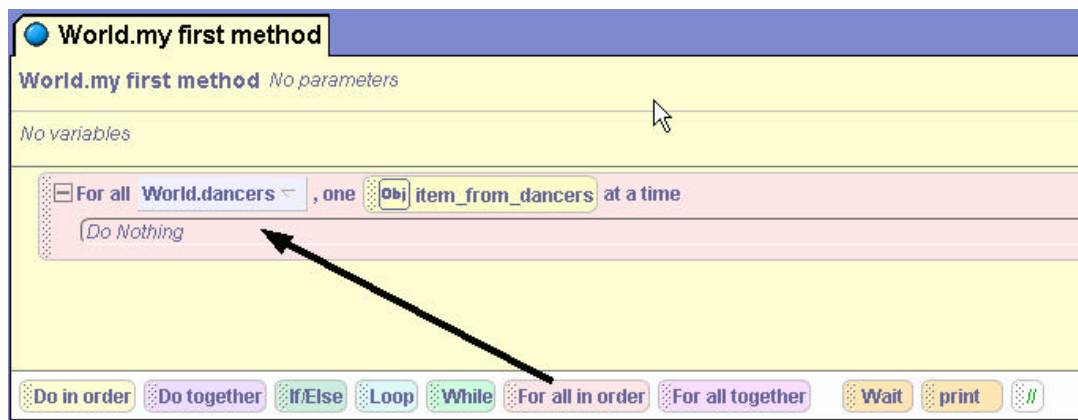


Figure 9-1-4. The *For all in order* tile

Now, we want to write instructions to make each individual Rockette kick up her right leg can be written as the character-level method `kickUpRightLeg`. Figure 9-1-5 displays the code for one making a rockette object kick up her right leg.

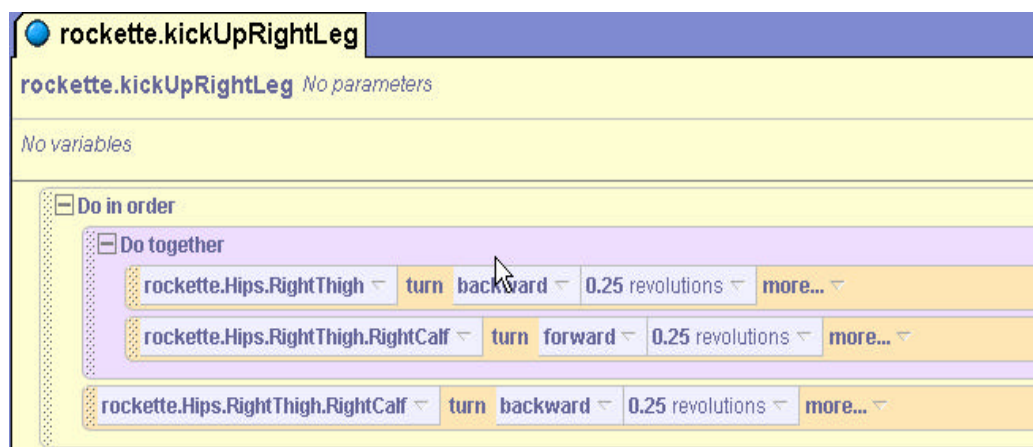


Figure 9-1-5. Rockette method to kick up her right leg

Now, following the same two-step procedure used before with character-level methods, we selected one rockette object as a prototype object and dragged her *kickUpRightLeg* method into the *for all in order* code block. The result is shown in Figure 9-1-6.



Figure 9-1-6. Prototype *kickUpRightLeg* instruction in *for all in order*

The next step is to drag the *item_from_dancers* tile over *rockette* in the *Rockette.kickUpRightLeg* instruction to make the instruction apply to any Rockette object in the list. **But, this is unsuccessful because the *rockette.kickUpRightLeg* method is not a built-in method and the smart editor will not allow the name of the object to be separated from the name of the method for a user-defined method.** So, we will make use of a technique we have used before (section 3-2) to pass the object as a parameter to a method.

Instead of the character-level *rockette.kickUpRightLeg*, let's create a world level method named *World.kickUpRightLegs*, as illustrated in Figure 9-1-7. In this world-level *kickUpRightLegs* method, the rockette object to kick up her leg is passed as a parameter named *whichRockette*.

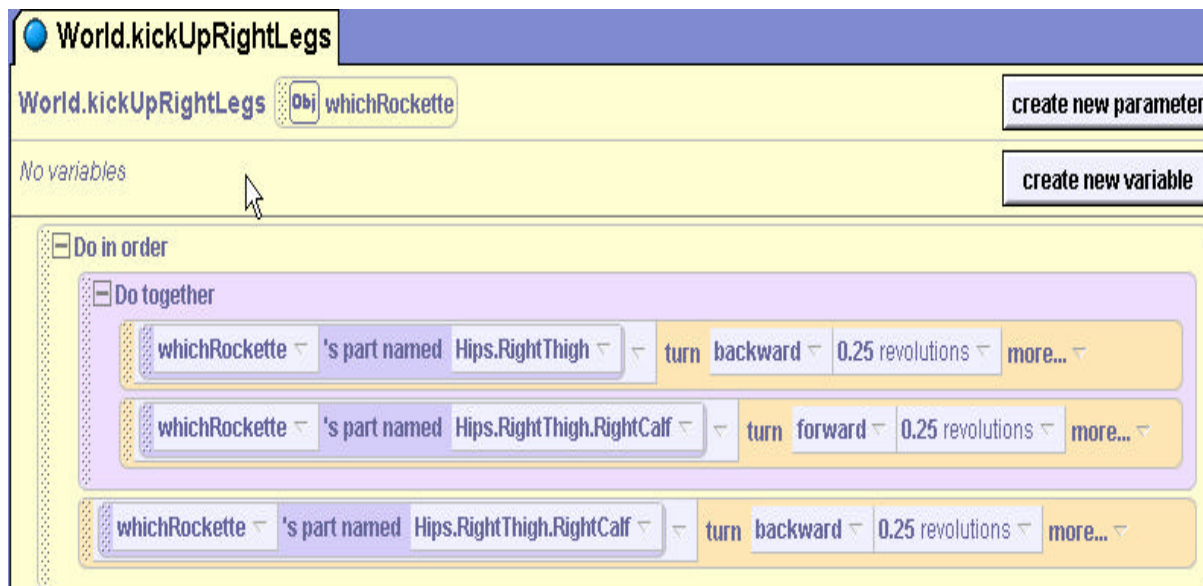


Figure 9-1-7. World-level *kickUpRightLegs* method

Now, the *World.kickUpRightLegs* method can be called from the *for all in order* construct, as illustrated in Figure 9-1-8.

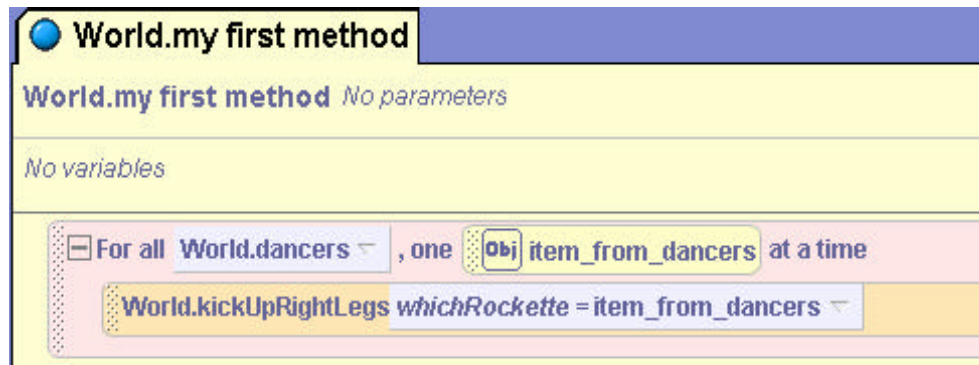


Figure 9-1-8. The modified *for in order*

When the program is run, each dancer kicks up her leg in order. Figure 9-1-9 illustrates the running of this program – the first 2 rockettes have their right leg in the air, and the third is just starting to kick her right leg upward.



Figure 9-1-9. Running the dance animation.

For all together -- Iterating simultaneously

Alice also provides a construct named *for all together* to make all the objects of a list perform the same action at the same time. This is called *iterating through a list simultaneously*. One way to think of *for all together* is that it is similar to a multi-way telephone conference call – everyone who is in connected to the conference call is on the phone line at the same time and everyone can talk and interact at the same time.

In any group dance routine, the dancers perform some steps sequentially and other steps at the same time. Let's have all the rockettes kick up their right leg at the same time. To do so, all we need to do is to replace the *for all in order* tile with a *for all together* tile. Alice makes this really

easy. A right-mouse click on the *for all in order* tile will allow you to select “coerce to *for all together*,” as shown in Figure 9-1-10. (Or, we could just drag the *For all together* tile into the world.)

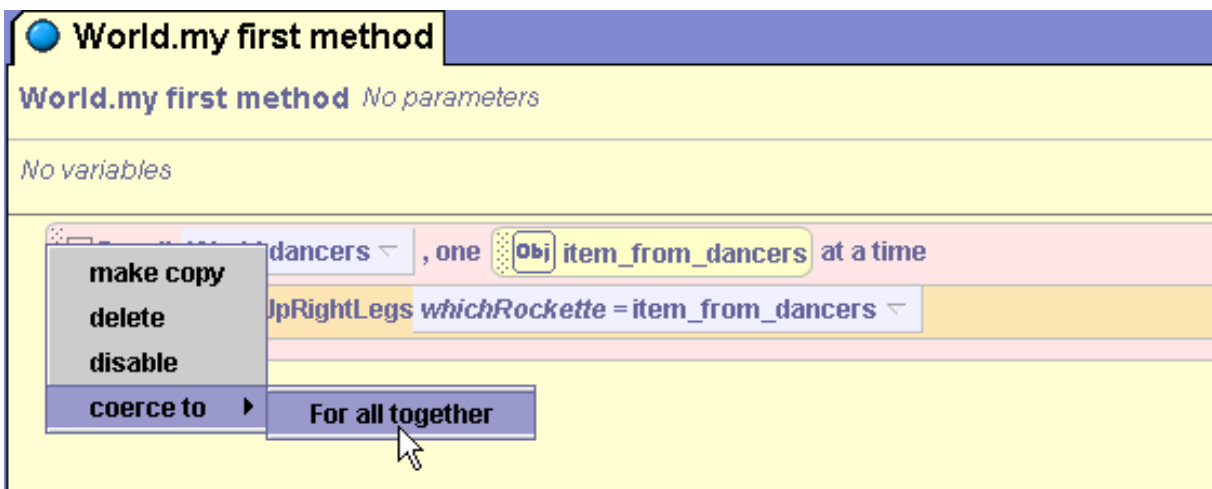


Figure 9-1-10. Changing the *For all in order* to a *For all together*

Now, when the code is run, the dancers all kick up their right legs at the same time! Figure 9-1-11 illustrates the running of the world using *for all together* -- the rockettes are all kicking up their leg at the same time.



Figure 9-1-11. Running the dance animation using *For all together*.

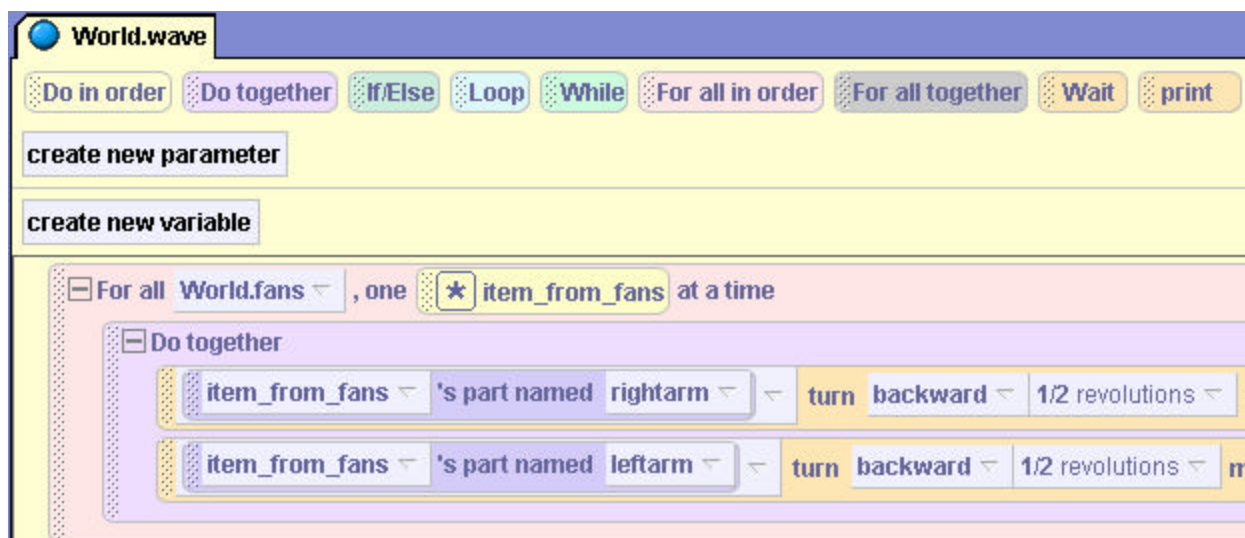
9-1 Exercises

1. The Wave

This exercise is to practice using *for all in order*. Create an animation that simulates some sports fans doing “the wave” – a popular stadium activity. Create an initial scene similar to the one illustrated in Figure 9-1-12. (Choose your own four characters.) Then, create a list of made up of the objects in the scene.

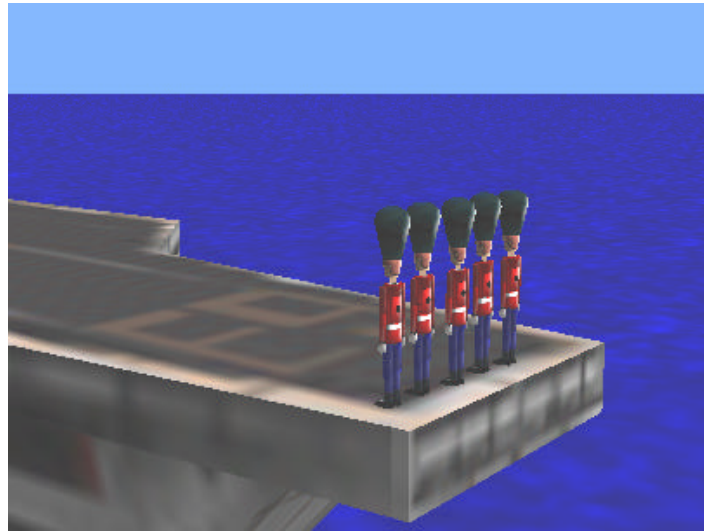


Use the *for all in order* construct to animate each of the characters raising their arms in order. One possible implementation of “the wave” is illustrated below.



2. Military Drill

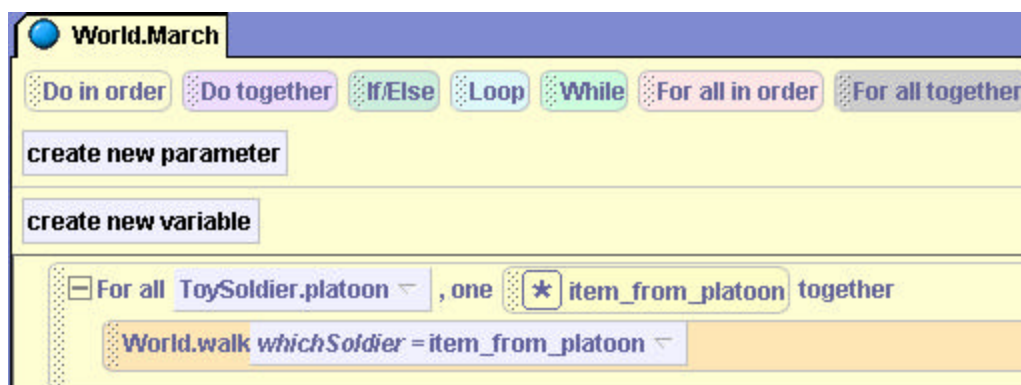
2. This exercise is to practice using *for all together*. Create a world with five soldiers on the edge of the carrier. The idea is to make the soldiers walk “in step” from one end of the carrier to the other. An initial world is shown below. Create a list variable, named **platoon**, made up of the five soldiers.



Use the *for all together* construct to conduct a military drill with the soldiers. The storyboard would look something like this:

For all toy soldiers in platoon list together
World.walk (object = item_from_platoon)

To implement this storyboard, create a world level *walk* method that receives a soldier object as a parameter an object. Then, in a *for all together* construct call the walk method. Sample code is illustrated below.



3. Landscapes.

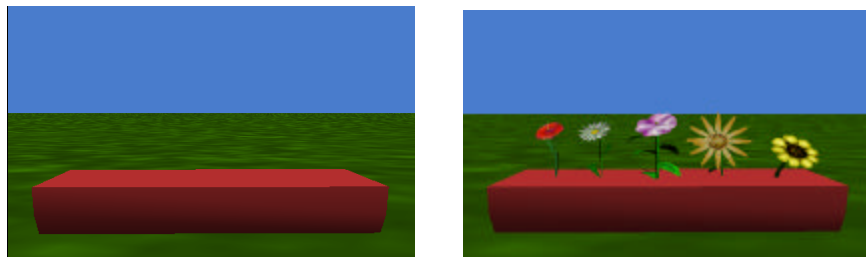
Several web sites can be found on the internet that provide free downloads of graphic files for web page development. For example, you might find a background of small blue stones. A background file can be used in Alice by importing the file as a texture map and then using commands to pass the texture map to the Ground as a skin and change the color of the ground. The result is a very different landscape, as shown below right.



Download from the internet 5 different backgrounds (not copyrighted). In a new Alice world, import the background files as texture maps for the Ground. Create a list of the five texture maps. Write an animation that cycles through the different textures, one at a time, creating a special effect where the landscape changes every 5 seconds. Each time the texture of the Ground is changed, also set the Ground to a color appropriate for the texture.

4. SpringFlowers.

Spring has sprung and you are anxiously awaiting the first flowers. You decide to give them a little help. Create an initial scene of a flowerbox with 5 flowers (your choice) hidden inside. (Hint: the flowers can be hidden inside the box by using the mouse to position the flowers out of sight.) Make a list, where each flower is one item of the list.



- Create an animation where each flower in the list “grows” upward (out of the flowerbox) one at a time.
- When you have that working well, create a second version where all flowers grow out of the flowerbox at the same time.

5. OlympicTrials

Create an animation to help the ice skater practice the standard skating figures for the Olympics. Use a list to hold the number of times a figure-eight followed by a full-pivot-turn is to be skated. The skater should skate the figures 1 time and then skate the figures 2 times, and then skate the figures 3 times...etc. Have the skater wave to the audience after each set of loops – this will signal when one loop set ends and another begins.

Also, the ice skater has been given a new set of skates that will really wow the judges: skates that change color while she's on the ice! To make the skates change color during the animation, create a list of different colors. Use nested loops through the two lists (how many figures and color of skates). The storyboard is:

```
For all (howManyLoops list) in order
  Do together
    Loop (current element from howManyLoops)
    For all (skateColor list) in order
      Change left and right skate to that color
```


9-2 List Search

Internet search engines are a great invention for helping us locate web pages. For instance, imagine you are looking for a video that tells the story of Caesar and Cleopatra. You could connect to a search engine (such as Yahoo or Google) and enter “Caesar and Cleopatra.” In a few short seconds, a list of web page links appears on your web browser. **Now, you do a *list search* – you read through the items, one item at a time, and look for a web page that has exactly what you want.** A *list search* is a common list operation used in programming. As with the real-world web-search example, the purpose of a list search in a computer program is to iterate through a list of items, one item at a time, and check each item to see if it has exactly what we want. In this section, we will explore how to do a list search in Alice.

Simulation using a list search

As an example of a list search, imagine a popular carnival game named "Whack-a-Mole." In this game, the player picks up a hammer and whacks at a mole-like puppet that pops up here and there out of a box at the carnival stand. Each time the player whacks the mole, points are scored. Our simulation of this game will be called “whack-a-hamster” (the authors of the text thought a hamster looked a lot like a mole!).

Figure 9-2-1 shows the initial world for the game. A yellow cube has been added to a scene and nine circles have been added to the top of the cube. Nine hamsters have been added to the scene and placed inside the cube, one below each circle (out of sight in Figure 9-2-1). Finally, 2 cylinders (named *totalscore* and *playerscore*) have been added in the background to act as a primitive visual scoreboard. The *totalscore* cylinder, representing the total (or target) score, is gray and is positioned on the ground. The *playerscore* cylinder is green and has been arranged in the scene so its top is just under the surface of the ground (out of sight in Figure 9-2-1). The *totalscore* cylinder represents the total (or target) score. The *playerscore* cylinder represents the current score. Of course, at the beginning of the game the player's score is zero – which is why the top of the *playerscore* cylinder is initially at ground level.

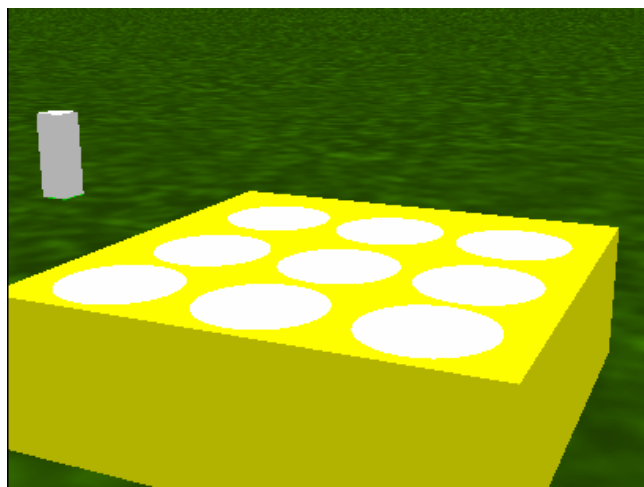


Figure 9-2-1. Initial scene for *whack-a-hamster*

The idea is for a hamster to randomly pop-up and down. Each time the player succeeds in clicking on a hamster while it is popped up, the *playerscore* cylinder will be raised, indicating an increase in the player's score. Once the entire *playerscore* cylinder has been raised above ground level the *playerscore* cylinder will then be the same height above ground as the *totalscore* cylinder. At this point, the player has won the game, and the hamsters should cease popping up and down. Figure 10–2-2 shows a screen capture of the game during execution. The top of the green *playerscore* cylinder is approaching the top of the gray *totalscore* cylinder, showing the player is doing well in the game.

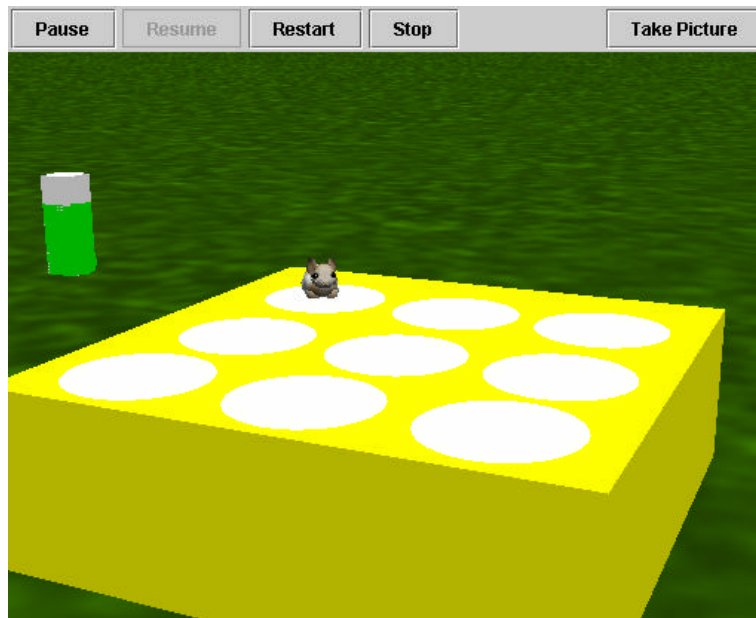


Figure 9-2-2. Scorekeeping columns at runtime

In planning a solution to this problem, it is clear there are two subtasks. The first is to set up a loop that continues to pop (raise and lower) hamsters randomly while the **game is in progress**. The second is to **increase the player's score** each time the player manages to actually click on one of the hamsters that pops up. In storyboard form,

Event: When the world starts
Response: *World.myFirstMethod*
 while the *playerscore* column is not yet above ground
 pop (and lower) a random hamster

Event: User clicks mouse
Response: *World.score*
 if mouse was clicked on one of the hamsters
 move the *playerscore* column upward
 play a pop sound (sound is optional)

First Subtask – Game in progress

When the world starts, the game should immediately begin. As you know, Alice automatically executes *World.my first method* when the world starts. Taking advantage of this automatic execution, the code for starting the game and keeping it in progress will be written in *World.my first method*.

While loop for continuous action

A while loop can be used to give us continuous action in the game. While the *playerscore* column has not yet moved above the ground (meaning that the player hasn't yet won the game) calls will be repeatedly made to a *pophamster* method. So, we need a method to pop-up a hamster. We have nine hamsters, so we will use a parameter *whichHamster* to represent one of the hamsters. The method is illustrated in Figure 9-2-3. Whichever hamster is passed to *whichHamster* will be the hamster to pop-up. The pop-up action makes the hamster move up, stay up for a half-second, and then move back down.

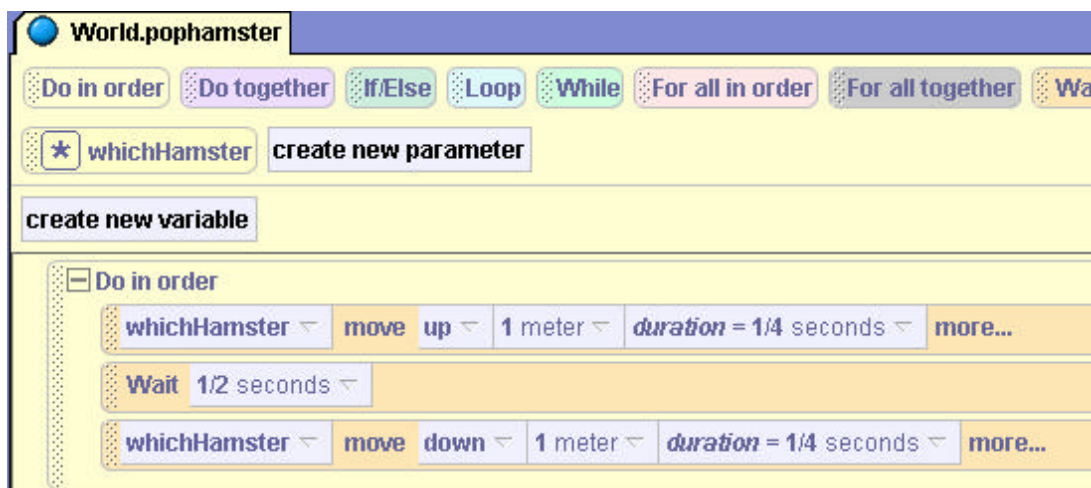


Figure 9-2-3. The pophamster method

A list of hamsters to allow random selection of hamster

Now we need to figure out how to pass a random hamster to the *pophamster* method. One way is to organize the hamsters into a list structure and then randomly select a hamster from the list. A list of hamsters is created, as previously described in section 10-1. The name of the list variable in this example is **hamsters** – not a very creative name, but it has the advantage of being obvious!

Now that the *pophamster* method has been written and the *hamsters* list created, the code for keeping the game in progress can be written in *World.my first method*, as in Figure 9-2-4. To send a random hamster from the list to the pophamster method each time it is invoked, we dragged the hamster list into the parameter tile and then selected *random item from list* from the popup menu. When this code is run, a random hamster from the list is passed as an argument to the *pophamster* method every time the while statement is repeated. When the *playerscore* column moves up above the ground, the while loop will end.

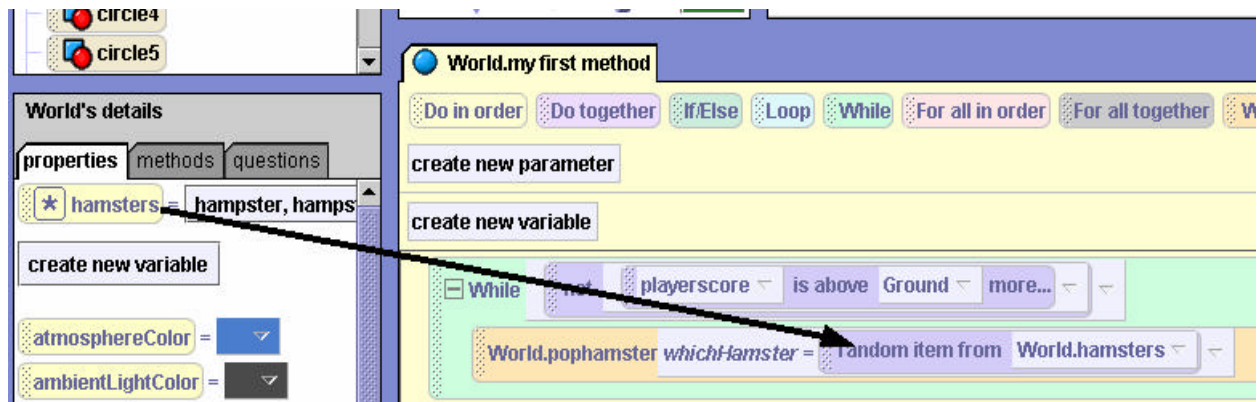


Figure 9-2-4. Repeatedly invoking *pophamster*

Second Subtask – Scoring

A score method will be written to visually display the player's success in the game. Each time the player actually clicks the mouse on a hamster that pops up. The playerscore column will be raised. Critical to this scoring subtask is the event that occurs when the player clicks the mouse. One of the events available in Alice is *When the mouse is clicked on something*, as shown in Figure 9-2-5.

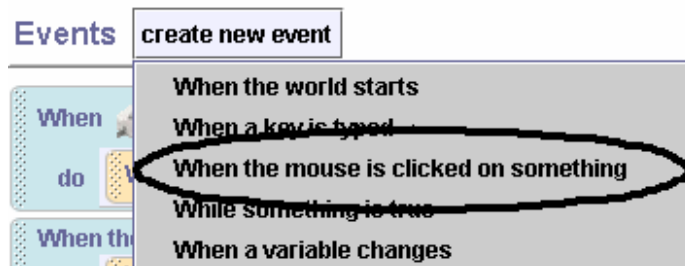


Figure 9-2-5. Mouse clicked on something event

In response to the mouse being clicked on something (anything), the *score* method will be called, as shown in Figure 9-2-6. The parameter *object under the mouse cursor* is selected for the *clicked* parameter tile. This event notifies Alice that the **mouse has been clicked** and Alice can determine **what object is under the mouse cursor**. The object under the mouse cursor will be passed to the *clicked* parameter for the *score* method.



Figure 9-2-6. Creating the event

The *score* method is shown in Figure 9-2-7. *Score* is written to make use of the knowledge of what object is under the mouse cursor.

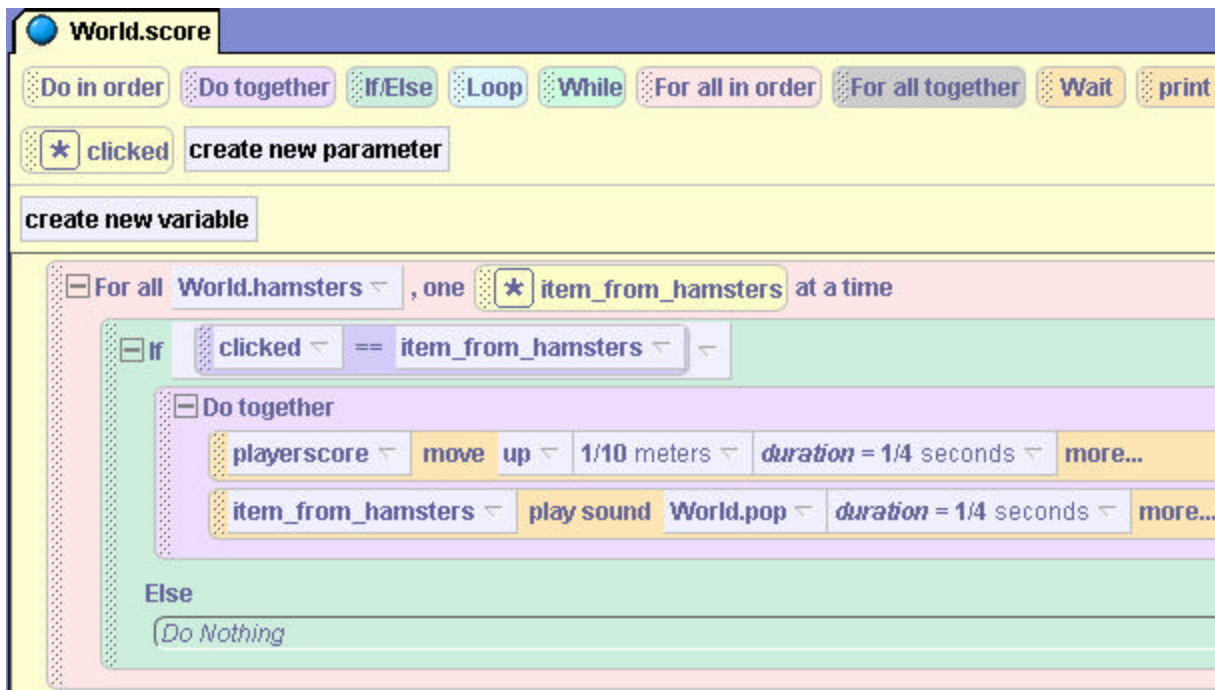


Figure 9-2-7. The score method

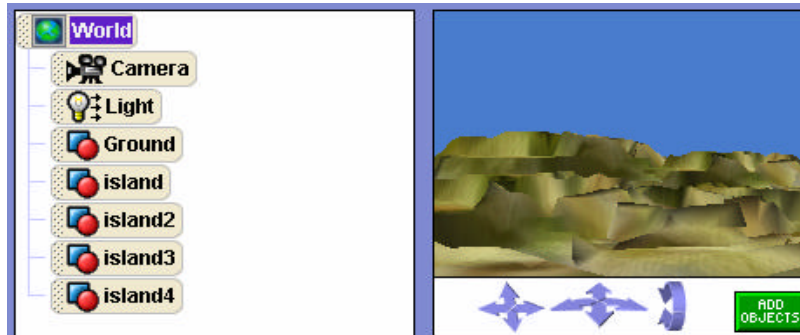
In the score method, a *for all in order* construct is used to iterate through the list, checking each hamster in the list to determine if that hamster is the object under the mouse cursor. **This is where the list search takes place!** If one of the hamsters in the list is the object clicked, the player's score is increased (by raising the *playerscore* column) and the hamster that was clicked makes a popping noise.

The important concept illustrated by the *whack-a-hamster* example is a technique of searching a collection of items to determine whether one of the items has a particular property. Other techniques for searching can be used, but the overall idea is the same. In this example, the collection of items is a list of hamsters. The search is conducted using a *For all in order* construct to iterate through the list. The *For all in order* construct goes through each hamster in the list and checks whether one of the hamsters has been clicked by the player.

Of course, it is possible that the player clicked too soon or too late and the object under the cursor was a circle or the yellow cube because the hamster had disappeared back underneath the circle. Or, the player may move the cursor too quickly and click on the grass or the sky. In any such situation, the search for a hamster that has been clicked fails. So, the player's score is not increased and no pop sound is made.

9-2 Exercises

1. Earthquake. Geologists who study earthquakes often use simulations of earthquake motions to study shifts in the earth's crust. Simulated earthquake motions can be useful when seismographic recordings are not available or are difficult to obtain. For example, earthquakes in the ocean depths are not easily measured. In this exercise, you will create a simulation of earthquake motions. The world is constructed with four mountainous structures (four islands from the *Skies And Grounds* folder in the web gallery). Arrange the structures close together, in various positions, to create a mountain range, as shown below.



Make a list consisting of the four mountain structures (islands). Create an animation where each of the islands in the list “shake.” A shake is a move in a random direction by a random amount. The animation should occur for each mountain structure in order and then for all mountain structures together. Experiment with different limits for the randomly generated amount of movement.

2. ShipMotion. To create a realistic animation of ships at sea, ships must appear to move as if responding to the motion of the waves. A response amplitude operator (RAO) ship motion approach is commonly used by animators. Create a simulation of ship motion with a world containing four ships. Make a list of the four ship objects. Then, animate all ships moving at the same time but by different amounts. Use a randomly generated RAO to determine the amount of movement in a random direction for each ship in the list. Because the ships are moving with the waves, the movements tend to be symmetrical. That is, a forward(left, up) movement is followed by a backward(right, down) movement in approximately the same amount. Use a loop to continue the motion simulation until the user stops the animation.



9 Summary

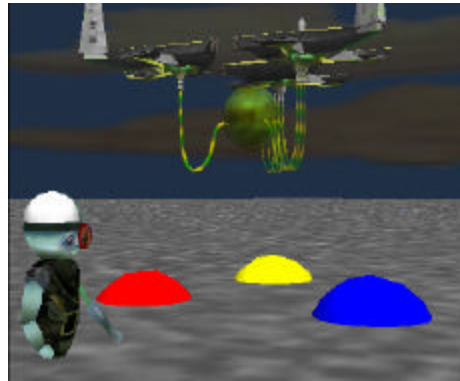
This major focus of this chapter was the list data structure. A list is an important data structure that allows us to organize and make use of objects and information about objects. We looked at how to create a list as well as iterating through a list. Alice provides two mechanisms for iterating through a list: sequentially (using *for all in order*) and simultaneously (using *for all together*). The *for all in order* iteration is similar to “walking” (or traversing) through each item in the list, one at a time. The *for all together* iteration is like setting up a multi-way telephone conference call – everyone is on the line at the same time. A common application of lists, searching for an item in a list that possesses some property, was also presented.

Important concepts in this chapter

- An *list* is a collection of values.
- The items in a list generally all have the same type (such as object).
- Lists are “*iterated through*” either sequentially or simultaneously.
- When a list contains objects and the objects are to perform some action, the list element is often passed as a parameter to a method and this method applies some action to that object in the list.

9 Projects

1. CodeTransmission. Create an interactive world to allow the user to transmit a message to a ship currently in port at the space docking station. The user should be able to click the red, yellow, and blue light modules on the ground in some particular sequence. (For example: red, yellow, blue, yellow, blue, red, blue). KidRobot will carefully memorize the sequence as a list of colors. When the user clicks on KidRobot, the robot will use her transmission cap (the white cap on her head) to transmit the color-coded sequence to a ship at the station – that is, the robot's cap will turn colors in the same sequence as the user entered. If your computer has sound, play a different sound along with each color.



2. CodeGame. Use the same scene as shown in the **CodeTransmission** project above. In this project, create a game where the player tries to remember the color sequence and reproduce it by clicking on the light modules. The game begins by generating a list of seven random number codes. Use red = 1, yellow = 2, and blue = 3. For example, the list 1 2 3 2 1 would be red, yellow, blue, yellow, red. KidRobot's transmission cap then displays the color sequence -- using the number codes from the list to determine the colors. The player then tries to click the light modules in the same sequence. If the player clicks the light modules in the exact same sequence, the player wins and the space station dock blinks. If the player does not enter the same color sequence, the player loses and 3D text appears that says "Play Game Again."

3. HalloweenTreasureHunt. In this game, Ghostlee (a local legend) is out for a treasure hunt on Halloween night. Phantom Manor (from the amusement park web gallery folder) has been littered with tombstones (from the graveyard web gallery folder). Under each tombstone is a treasure. When Ghostly gets close to a tombstone, the tombstone falls over and the treasure is revealed. Ghostlee's task is to visit all the tombstones and reveal all the treasures. Unfortunately, skeletons are patrolling the grounds – walking back and forth near the tombstones to protect the treasures. If Ghostlee gets too close to a skeleton, he will vaporize and have to wait until next Halloween night to find the treasures.

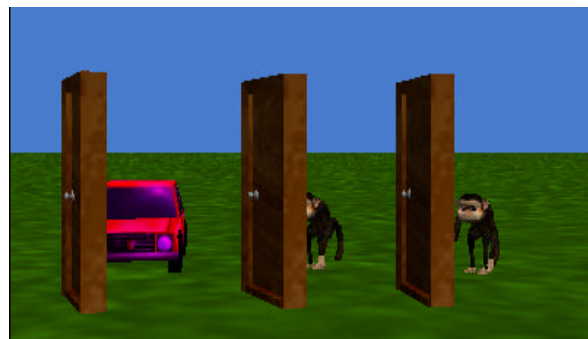
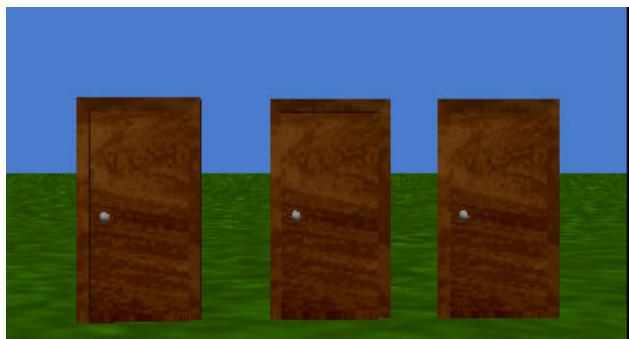


Create an animated game for the Halloween Treasure Hunt. Use a list to store 10 tombstones. The tombstones are all around the manor's grounds (screen shot shows only the front side). Allow the user to use the keyboard arrow keys to move Ghostlee around the grounds of Phantom Manor. Make the camera follow Ghostlee so he is always in sight. Do not worry about Ghostlee moving through an object -- he is a ghost! Use a second list to store whether the tombstone has been visited -- *false* if not yet visited, *true* if visited. If Ghostlee manages to visit all the tombstones, the game is over and Ghostlee has won. If Ghostlee encounters a skeleton at too close a range, the game is over and Ghostlee will vaporize into thin air.

4. Monty. A famous mathematical problem (Monty Dilemma) goes something like this:

Suppose you're on a game show, and you're given the choice of three doors. Behind one door is a car, behind the others, toy monkeys. You pick a door, say number 1, and the host, who knows what's behind the doors, opens another door, say number 3, which has a toy monkey. He says to you, "Do you want to pick door number 2?" Is it to your advantage to switch your choice of doors?

One way to find an answer to this question is to create a simulation of the game where the objects are placed randomly behind the doors. Play the game 25 times where you do not switch and 25 times where you do switch. Don't count the times where you pick the correct door on the very first try. Keep track of your success rate when you switch and your success rate when you do not switch. Then, you will know the answer to the question!



To set up the initial world, add 3 doors, a car, and 2 toy monkeys to a new world. The doors can be found in the Furniture gallery, the car in Vehicles, and the monkeys in Animals. Position the car and the two monkeys behind the three doors, as shown above. Each door should be 2.5 meters to the left (or right) of its neighbor door. Make the door positioned in front of the car the

vehicle for the car and then do the same for each monkey and its respective door. (When the door moves, the object behind will move with it.) Also create a 3D text object that asks the player to “Select A Door”. Make the text object invisible by setting its *isShowing* property to *false*.

To program the game, make a list containing the three doors. When the animation begins, randomly pick 2 doors and swap them. Use a method called *swap* that takes 2 doors as parameters. Swap the 2 doors by having them move 2.5 meters in opposite directions. **IMPORTANT:** two doors should only swap if they are different (meaning a door cannot swap places with itself). Repeat the swap ten times. (The idea is to make it difficult for the player to know what object is behind what door.)

Display the 3D text to ask the player to select a door. When the player clicks on one of the doors, determine whether that door hides the car. If the object behind the door is the car, the player has won. Open the door and declare the player a winner. Otherwise, pick a door that hides a monkey and open that door. Then, ask the player to choose again. Once again, if the object behind the door is the car, the player has won. Open the door and the player wins. Otherwise, the player has lost. Use sound or 3D text to indicate the win or loss.

10 Arrays

An array is a structure that allows us to collect elements of the same type into a group. In our examples, arrays will be made up of objects. In chapter 10, we looked at a list structure – which is also a collection of elements in a group. How is an array different from a list? An array and a list are both collections of elements and they do share some common characteristics. **But, an array uses a *location index* to say where the element is positioned in the collection, whereas a list does not. And, arrays generally do not change *size* while lists often do.** *Size* is the number of elements in the array.

Arrays are commonly used in programming languages as the location index makes it possible to pick out a specific element from the array without necessarily having to iterate through the entire array to find the element you want. Of course, we can also go through an array from beginning to end (*iterate through* the array).

Our goal in this chapter is to demonstrate visual representations of some of the classical operations on arrays. We believe that Alice’s strong visual presentation of arrays will help you gain good intuitions about how arrays work. The eventual reward for learning about arrays in Alice is that these intuitions will prepare you for learning to use arrays in a “real world” language.

Section 10-1 starts with the basics: how to create an array and populate it with objects. It is not necessary for arrays to contain objects but our examples will use objects to make the array visual. Once we have constructed an array, we will then look at how to go through an array to find an element with a given property (tallest object in the array) in a manner quite similar to what we did with lists. Several solutions to this task will be presented as a means to give reason for the introduction of a mutable variable. This will demonstrate how much easier it is to write a method to find the tallest element of an array using variables.

In section 10-2, a visualization of a famous (and classic) array sort will be presented. Objects in the array will be sorted by height, as a follow-up to the example presented in section 10-1, though other sorting criterion could certainly be used (for example, girth or volume).

11- 1 Arrays: Creation and use

Introduction to arrays

An array is a structure for organizing data and objects into a collection or group. Of course this description sounds very much like what we said about a list in chapter 9. Let's use a "real-world" analogy to explain the concept of an array. An array is somewhat like a music CD. The CD contains a collection of songs. The label on the CD lists the order of the songs and you can use a CD player to play the entire CD or you can use a location index to select a specific song (or songs) to play. If you want to play song 5 on the CD you can just select song 5 and play it. Compare this to a cassette tape containing the same collection of songs (which is something like a list). If you want to play song 5 on the tape, you put the cassette in a player and, starting at the beginning of the tape with the first song, fast forward through several songs on the tape to get to song 5 – then you can play it. Both the CD (like an array) and the cassette (like a list) contain a collection of songs, but the way you "get at" songs in the collection differs in ideology and style.

In this chapter, we will work with arrays of objects because objects are visible in our virtual worlds. Some worlds will demonstrate the action of going through the array, one object at a time. We will also gain *access* to individual elements in the array directly, using special array operations.

Creating an array

To a world with several objects, we add an *ArrayVisualization* object. At the time the array is added to the world, the array has no elements. To add elements to the array, click on the ArrayVisualization object properties tab and see that there is a variable named *iterator*. Clicking on the box to the right of the iterators allows us to add several objects to the array, as shown in Figure 10-1-2. As each object is added to the array the object moves into its position in the array. Note that arrays start in location 0 of the array. (Computer scientists are probably the only people in the world who start counting from 0!) In this example, we have five insects. The beetle is in location 0, the mantis in location 1, the Scarab in location 2, the Bee in location 3, and the ladybug in location 4.

Accessing an element in an array

If we want the bee (located in position 3 of the array) to turn around once to its left, we can write an instruction in the same way as we have done in previous programs:



But, now we can write:



This is because the object at location 3 in the array is the bee. The question *‘the value at ArrayVisualization[index]’* is one of the *ArrayVisualization* built-in questions.

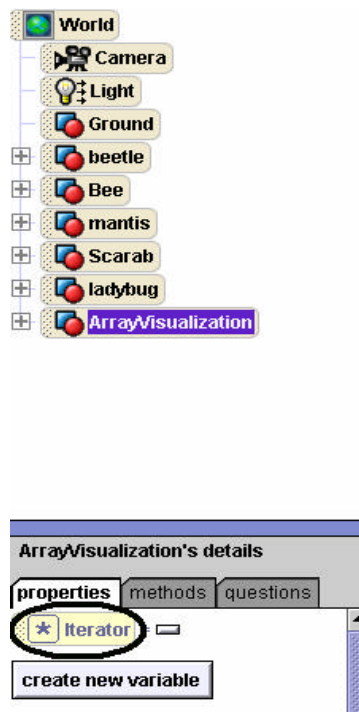


Figure 10-1-1. The ArrayVisualization iterator

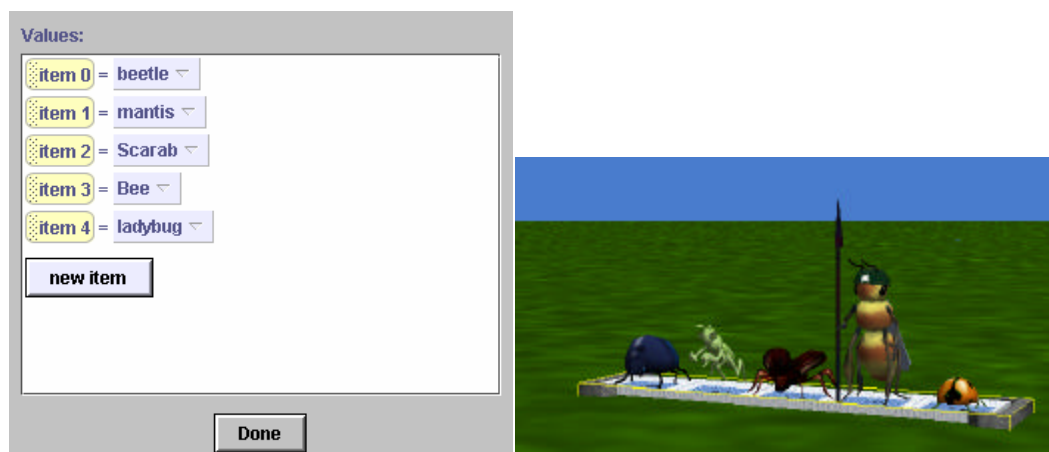
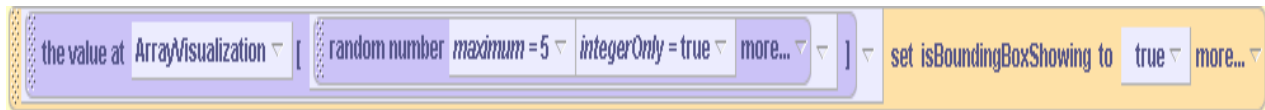


Figure 10-1-2. Adding objects to an array

Iterating though an array to find an element with a certain property

One of the ways of working with an array is to go through the array, one element at a time. This is called *iterating through* the array (an action also performed with lists). To illustrate iterating through the array, let's do something to identify one of the objects in the array as an object we are looking for. This is kind of like the FBI creating a [Wanted](#) Poster. To identify one of the objects as “**wanted**,” we randomly turn on the bounding box for one of the elements in the array.



Now, let's write a question that returns the wanted object (the one that has its bounding box on). With a list, we could use *For all in order* to check the *isBoundingBoxShowing* property of each of the items and return the object whose *isboundingBoxShowing* property is true. But, the *for all in order* construct only works with a list, not an array. One solution is to create a list of the indices, as is done in Figure 10-1-3.

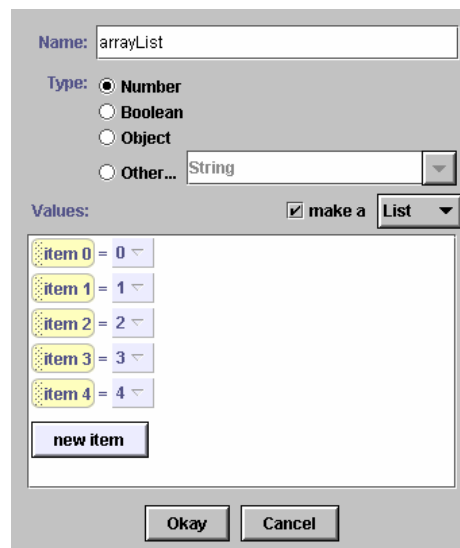


Figure 10-1-3. A list of indices of the array

The *for all in order* construct is used to iterate through this list of numbers and the number is used as a location index for each position in the array. The resulting code is in Figure 10-1-4. When *item_from_arrayList* is 0, the value at `ArrayVisualization[item_from_arrayList]` is actually `ArrayVisualization[0]` – the beetle. So, *for all in order* walks through the array one insect object at a time looking for the insect whose bounding box is turned on.

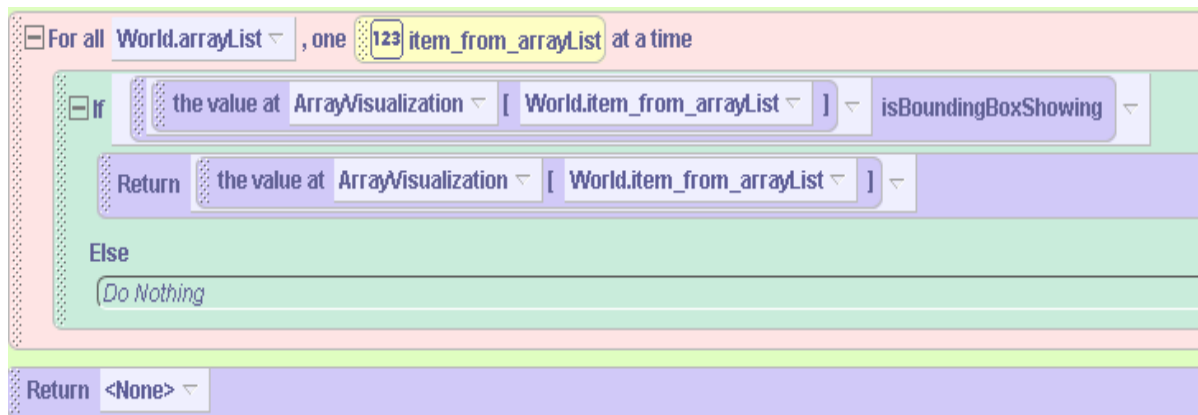


Figure 10-1-4. Determining which insect's bounding box is turned on

A sample program segment that calls this question is shown in Figure 10-1-5. The insect whose bounding box was randomly turned on is made to jump up and down, indicating that the iteration through the array has been successful. Note that we must have turned on the bounding box of one of the objects in the array before a call is made to *World.insectInBoundingBox*. Otherwise, the iteration though the list will not find an insect object where the bounding box is showing and nothing will be returned by the question. We left this in the code as an intentional illustration of the fact that **your program can crash if you don't care for possible situations**.

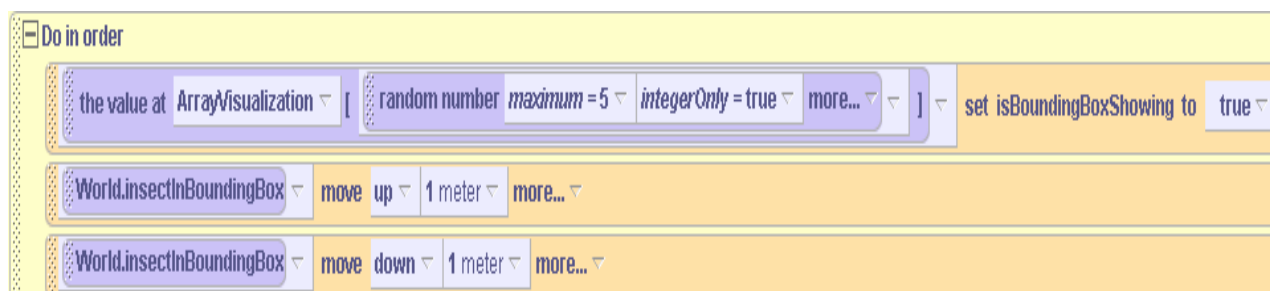


Figure 10-1-5. Calling the *InsectInBoundingBox* question

Searching through an array to find the tallest element

As a second example, let's search through the array to find the tallest object in the array. This problem is more complicated than the previous problem in that there is no built-in question to determine which item in an array is the tallest one.

One approach to this problem is to guess that the first element is the tallest. That is, it is possible that the first object could be the tallest one in the array, so we can start there and say that it is the "tallest object so far." Then, compare the tallest one so far to the second object in the array. Of course, if the second object is taller than the tallest one so far, then the second object becomes the tallest one so far. Now compare the third element against the tallest object so far. If the third object is taller, then indicate the third one is the tallest one so far. Then compare the fourth object against the tallest one so far. You get the idea... this keeps on going and going until we get to the end of the array. When we get to the end of the array, the tallest one so far is the tallest one in the

whole array. What we have done is to think through a sequence of steps where we mark one object as the wanted object. Then, if we find a better object we change our mind and say that this object is now the wanted object. We can put this sequence of steps (an *algorithm*) into a storyboard (pseudocode):

Set the bounding box to be true for the first element in the array (the bounding box is set on for the largest element)
Iterate through the array
 If the i'th element is taller than the tallest
 Indicate that it is the new tallest element by turning its bounding box on, and turn off the bounding box of the previous tallest element

The code in Figure 10-1-6 is an implementation of this solution.

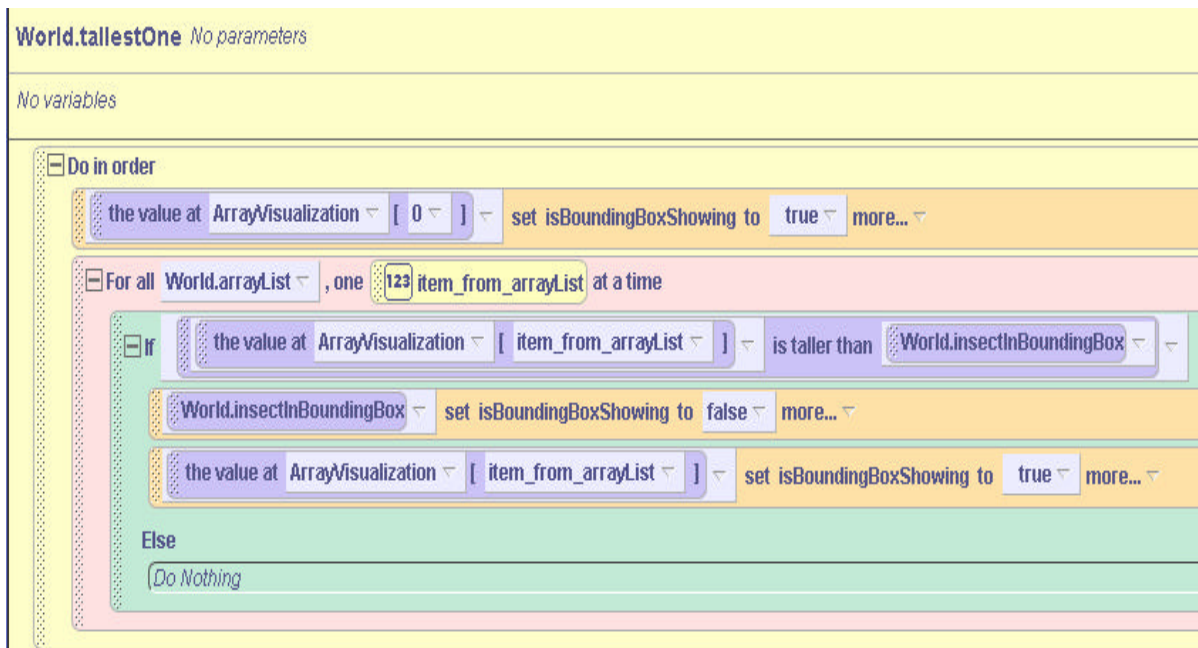


Figure 10-1-6. Searching for the tallest object in the array

This solution, while correct, is somewhat unsatisfying. The reason we think this solution is somewhat unsatisfying is that we need to call `World.insectInBoundingBox` many times. The bounding box is being used as a visual marker of the tallest one we have seen so far (a very clever way of letting you know which one is the tallest one so far). But, every time we want to check for the tallest one we have seen so far, the `World.insectInBoundingBox` method has to be called. And, of course, that means Alice goes iterating through the array again looking to find which of the insect objects in the array has its bounding box turned on. This means that Alice is doing a search through the array of insects again and again and again and Goodness, far too many times!

Adding variables and revisiting the tallest element method

If we could only **keep track** of which of the insects was the tallest one so far, we wouldn't need to continually be calling `World.insectInBoundingBox` to tell us which one it is. **This is where a variable comes into play.** A variable is a piece of storage that stores a value. Unlike what we have seen in Alice so far, such variables are not visible. We cannot see their contents displayed in our virtual worlds. But they play a very important role. We will use a variable to keep track of the location of the tallest insect we have seen so far. This will eliminate the need to call `World.insectInBoundingBox` as the variable will **keep track** of which insect is the tallest one.

To create a variable, click on the “Add variable” button to the upper right of the `tallestOne` method in the editor, as shown in Figure 10-1-7. A popup box appears where the name can be entered for the variable and its type selected. In this example, we named the variable `locationOfTallest` and selected its type to be number, as the location index is a number. We gave it an initial value of 0, guessing that the tallest insect is in location 0 of the array. Using the variable, the `tallestOne` method can now be modified, as shown in Figure 10-1-8. The value of the `locationOfTallest` variable is set by dragging it into the code and giving it the value of the `item_from_arrayList` (from a drop-down list). The bounding box is no longer actually necessary, but our example continues to turn the bounding box on and off for visual effect.

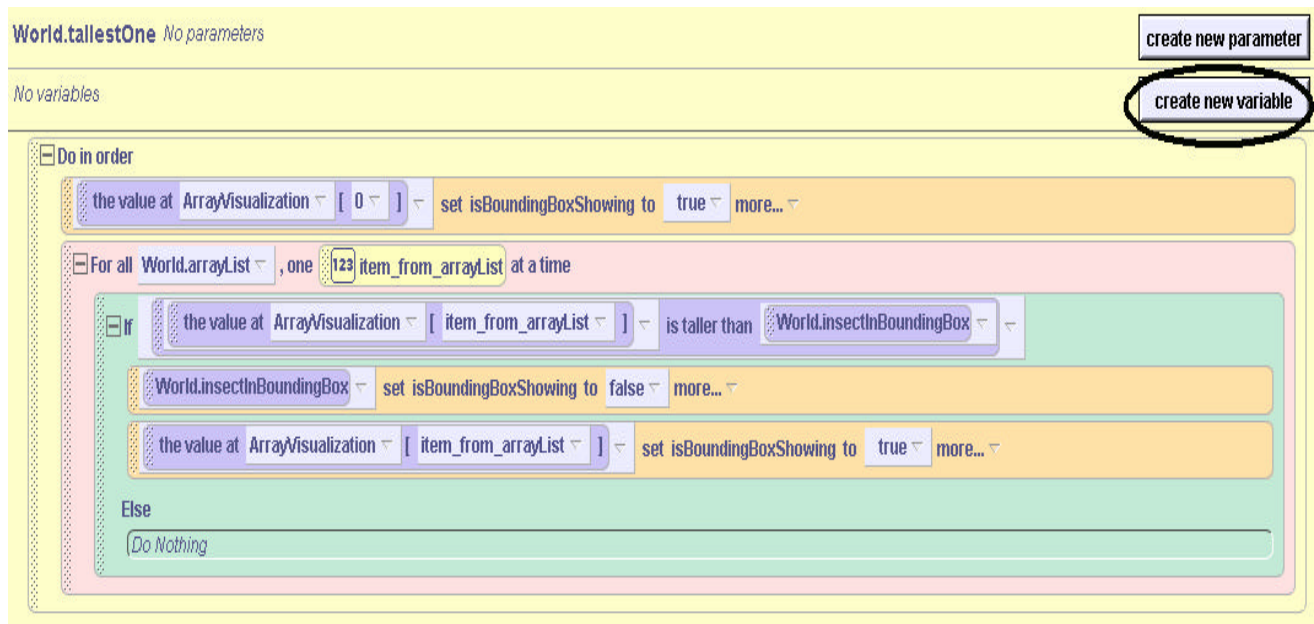


Figure 10-1-7. Creating a variable for a method



Figure 10-1-8. Using a variable to track the location of the tallest element

Using a location index variable a while statement in place of an iterator

Now that we've added variables, we no longer need to use a list of iterators to iterate through the array. Instead, a second variable (we'll call it *i*) can be used to represent the location index for the array. The location index variable *i* is used in coordination with a *while* loop to do the same task that the iterator was doing – going through the array one element at a time. The resulting code is displayed in Figure 10-1-9.

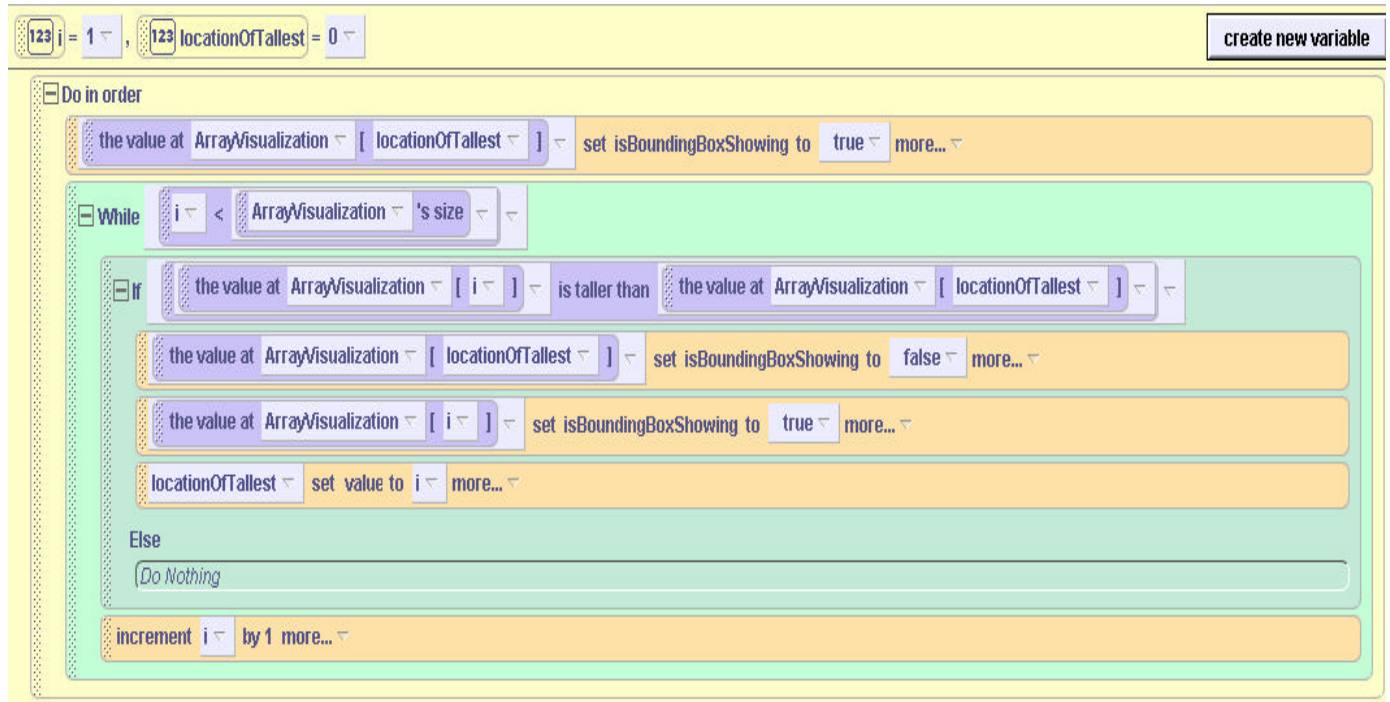


Figure 10-1-9. Using a while loop with an index variable

To begin, the variable *i* is initialized to 1 and the `locationOfTallest` is 0. At the beginning of the while loop, the Boolean condition is “*i* < *ArrayVisualization's size*”. *ArrayVisualization's size* is a built-in array question that returns the number of elements in the array. In this example, the array contains 5 insects. Since *i* is 1 and the size of the array is 5, the loop will execute and the beetle's height (at location 0 in the array) will be compared to the height of the mantis (at location 1 in the array). Of course, the mantis is the tallest so the `locationOfTallest` will be set to 1, the bounding box of the beetle will turn off and the bounding box of the mantis will turn on.

The last statement inside the while loop is “**increment *i* by 1 more**”, so *i* becomes 2. Now, after the first execution of the while loop, the condition of the while statement is checked again. The location index variable *i* is now 2 and the size is still 5, so *i* is still less than 5 and the loop will execute again. Now, the height of the mantis will be compared to the height of the Scarab...and the process will continue again and again. Each time through the *while* loop, the variable *i* is incremented by 1, so eventually *i* will no longer be less than the size of the array and the loop will stop.

Overall, the loop in this example should execute four times. When the loop ends, the **`locationOfTallest`** variable hold the location of the tallest object in the array.

11 Exercises

Create a world containing an array of 5 insects and implement the code for the *tallestOne* and *insectInBoundingBox* methods, as demonstrated in this section. Then, modify the world in each of the following exercises.

1. Modify the question *insectInBoundingBox* so it returns the index of the insect in the bounding box rather than the insect that is in the bounding box. And, write code that will have that insect jump up and down when its index is returned.
2. Modify one of the search algorithms presented in this section to instead search for the shortest character in the array.
3. Write a question, *firstBeetle*, which searches through an array and returns the location of the first beetle in the array. (Make sure you put at least one beetle in the array. You may also wish to create a list of all of the beetles.) To test your question, in a method, include the statement:



4. Write a question, *lastBeetle*, which searches through an array and returns the location of the last beetle in the array. (Make sure you put at least one beetle in the array.) Again, test your question by having the *lastBeetle* move up.

11- 2 Sorting arrays

What is sorting?

Sorting is the arrangement of some collection of items using some ordering measurement (called a *criterion*) as a reference. In a shoe store, for example, the sales person might want to arrange pairs of shoes on a sale display rack so the shoe size increases from left to right. This kind of arrangement is used to make it easier for the customer to find their size shoes on the sales rack. We often assume that sorting will be done in increasing order (ascending) but, of course, decreasing order (descending) is used for some purposes.

Sorting an array of elements is one of the classic uses of arrays in introductory programming. There are many, many sorting algorithms that can be used to sort an array. In this section, we will demonstrate a famous sorting algorithm called an *insertion sort*. A couple other sorting algorithms will be presented in the exercises at the end of this section.

Insertion sort

Let's assume we want to sort an array of Alice animal objects by increasing height (a natural follow-up to the search for the tallest object we wrote in section 11-1). First, we create a world with an array of animals, as shown in Figure 10-2-1.



Figure 10-2-1. An initial array of animal objects

The basic idea for an insertion sort is:

- 1) Start with the second object in the array. Look to see if it is shorter than the first element. If it is, rearrange the first two objects so that they are in shortest to tallest order.
- 2) Look at the third object in the array. Insert it in its correct order among the first two.
- 3) Now look at the fourth object, etc.

Notice that with each step, **the object we visit gets inserted into the correct position among the previous objects.** This algorithm can be written as the following storyboard pseudocode:

Method insertionSort:

For each array object starting with the second

 Remove the object from the array

 Call insertInPosition, passing the object to the parameter

 (insertInPosition will place the object in its correct position
 among the previous objects)

To implement the insertion sort, we need to consider three subtasks: (1) walking through the array one object at a time, (2) removing an object from the array, and (3) inserting the object in its correct position among the previous objects.

Walk through the array

To walk through the array one object at a time, we will use the same strategy we used in section 11-1. Our insertionSort method will have a location index variable (named *i*) to go through the array and gain access to one object at a time, from the second to the last object in the array. Of course, the index variable will be used in coordination with a while statement. The code for the *insertionsort* method with the location index variable *i* and the while statement is shown in Figure 10-2-2.

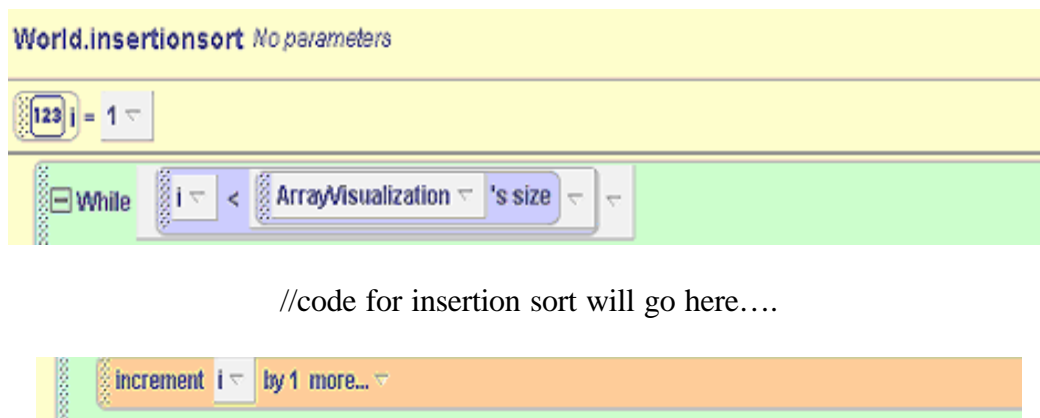


Figure 10-2-2 While statement and location index to walk through array

Remove an element from the array

The first action within the loop is to remove the current element from the array. The reason is we want to position the object somewhere among the previous objects. But, objects already occupy all the available positions. Somebody has got to move! To enable the objects to shift position, so our current object can be repositioned, we will make room by moving the current object (the *i*'th object) into a temporary placeholder location.

For the placeholder location, we drag an *ObjectVisualization* object (available in the Visualizations folder of the Alice Home gallery) into the world. Figure 10-2-3 shows the unsorted array of animals with a white square in front of the array object. The white square is the *ObjectVisualization* object. When we remove the *i*'th object from the array this is where the object will go until it can be repositioned in its correct place in the array. You can think of the *ObjectVisualization* object as like a spare bedroom in a house – a temporary place for someone to stay while away from home.



Figure 10-2-3. The initial array of animals, with an extra *ObjectVisualization* object

Now, we can write the code to remove the object from the array. The code is shown in Figure 10-2-4. The statement “*let ObjectVisualization = the value at ArrayVisualization[i]*” moves the *i*'th object from the array to the white square that represents the *ObjectVisualization* placeholder location.

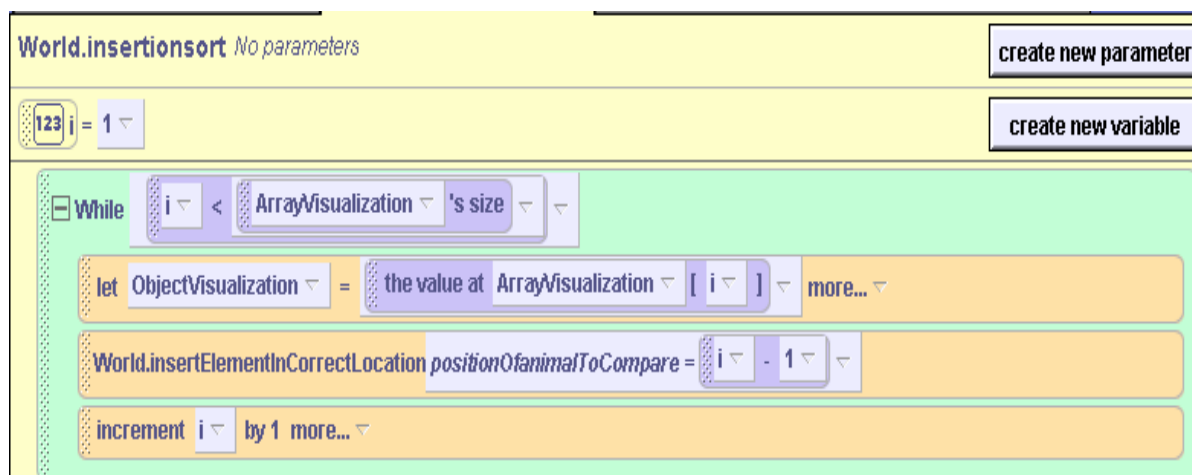


Figure 10-2-4. Insertion sort

Insert element in its correct location

By removing the i 'th element of the array (the current animal object in the array), we can now shift over to the right any of the $i-1$ remaining elements that are taller than i 'th element. As seen in the code in Figure 10-2-4, we invoke the *World.insertElementInCorrectPosition* method to perform this task.

The *insertElementInCorrectPosition* method is subtle to write. It takes as a parameter the index of the element that the current element is to be compared against. If the current element is shorter than the element being compared, the method must move the compared element over one space in the array. If we are not at the front of the array, recursively call itself with the next lower index.

The following pseudocode illustrates this procedure:

```
Method insertElementInCorrectPosition    Parameter i
If array[i] is taller than the ObjectVisualization object
    Move array[i] over into position i+1
    If i isn't 0 (we're not looking at the first element)
        Recursively call insertElementInCorrectPosition, passing as a parameter, the value i-1
    Else
        Insert the ObjectVisualization object into array[0]
Else
    Replace the ObjectVisualization object into the hole in the array (at position i+1)
```

The code to implement this pseudocode is presented in Figure 10-2-3.

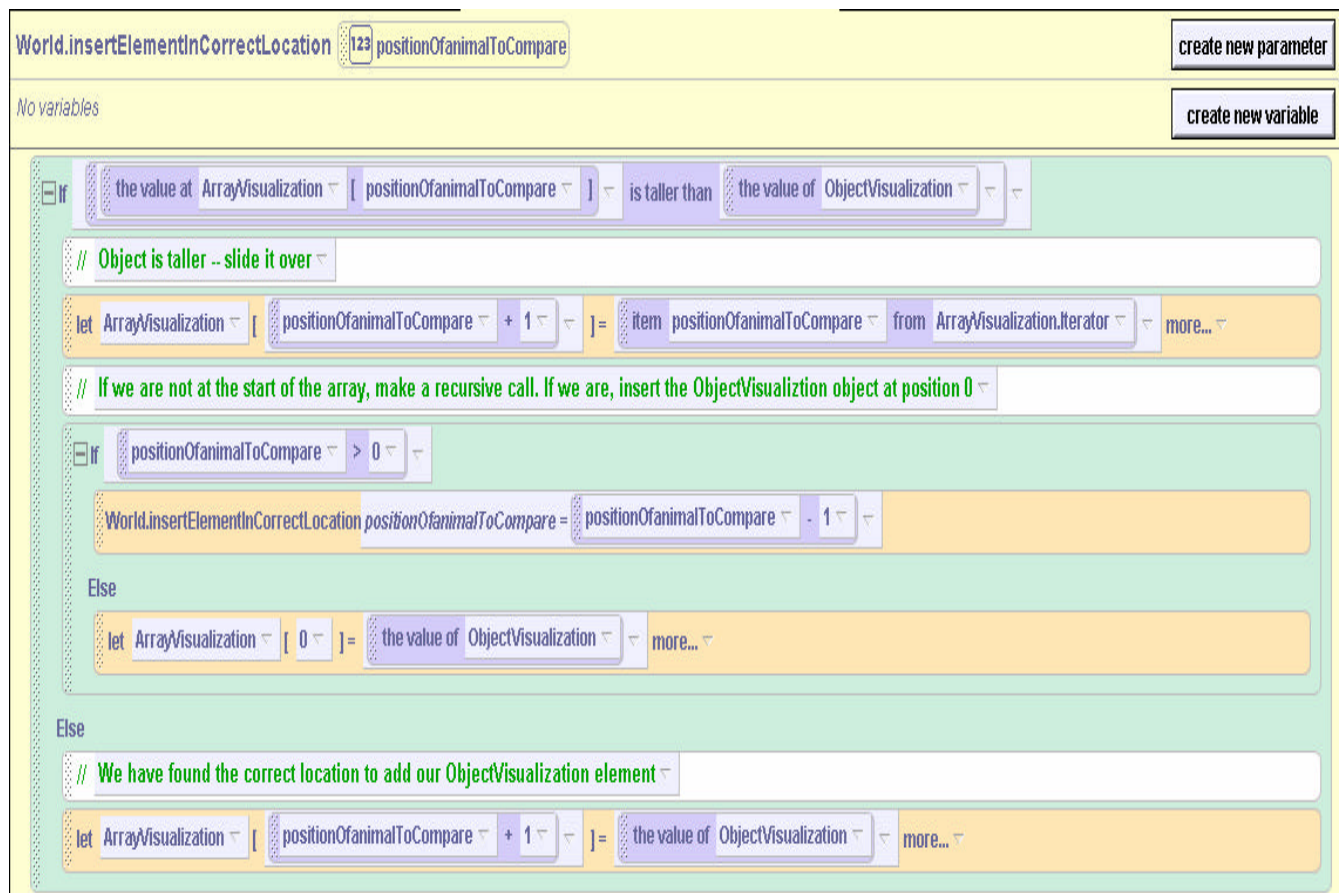


Figure 10-2-3. Inserting an element into its correct location

This code can best be seen in operation. Using our animals array example, we can see that the first two elements, the horse and the fish objects, are already in their correct locations. The goldfish is the 3rd element of the array and is moved to the placeholder location. Let's pick up the action with the call to the `insertElementInCorrectLocation` method for the goldfish.

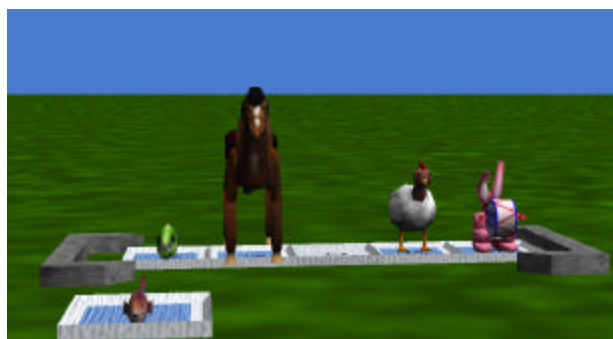


Figure 10-2-4. Inserting the goldfish into its correct location

The goldfish's height is compared to the horse's height. Since the horse is taller, it is moved over one place to the right (to location 2 in the array).



Figure 10-2-5. The horse is taller

Now, a recursive call is made to the *insertElementInCorrectPosition* method. Now, the goldfish's height is compared to the height of the green fish in position 0. Again the green fish is taller, so it is moved over one place to the right (to location 1 in the array).



Figure 10-2-6. The fish is taller

But now, we have gotten down to position 0 (the first position in the array). Rather than making another recursive call, the goldfish, being the shortest element among the first three, is placed into location 0.



Figure 10-2-7. The goldfish is placed back into location 0 of the array

Let take a second look, later in the sorting process. The first 4 animals are now in height-order, and it is time to move the fifth animal, the bunny, into its correct location by height-order.



Figure 10-2-8. Placing the bunny into its correct location

When method *insertElementInCorrectPosition* is first called, it compares the bunny's height to that of the horse. The horse is taller, so the horse is moved into location 4 of the array.



Figure 10-2-9. The horse is taller

Since the horse was not in the first position of the array, a recursive call is made to *insertElementInCorrectPosition*. Now the bunny's height is compared to the height of the chicken. The chicken is taller so it is moved over into location 3.

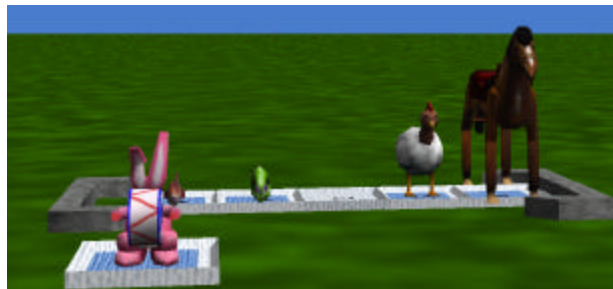


Figure 10-2-10. The chicken is taller

Again, the recursive call is made to *insertElementInCorrectPosition*. But, this time the bunny is taller than the fish, so it is moved into location 2 into the array.



Figure 10-2-11. The bunny is taller than the fish

And, since the bunny was originally the last element in the array to be inserted, the insertion sort is over, and the animals are all in height increasing order.

10-2 Exercises

- 1) Write a method named *swap* that receives 2 numbers as parameters, named *location1* and *location2*. The two numbers will be the location index for two objects in the array. The swap method should swap the contents of the array element so that after running the method *swap*, *array[location1]* should contain the object that was previously in *array[location2]* and *array[location2]* should contain the object that was previously in *array[location1]*.

Hint: You will need an *ObjectVisualization* to serve as a placeholder during the swap.

- 2) Rewrite the insertion sort using an iterator and for all in order instead of a variable and a while loop (similar to the first example in section 11-1.)

10 Summary

This major focus of this chapter was the use of arrays. Arrays, like lists, are an extremely important data structure in computer science. They allow for information and objects to be organized into collections. Arrays allow us to access specific array elements using a location index. The size of the array never changes during the running of a program. We find that we often need an extra placeholder (the *ObjectVisualization* object) to help in rearranging array elements.

We saw that we could use iterators to go through the elements of an array one element at a time. But we also found that it was often more convenient to use *variables* to keep track of information (such as the index of the largest element in the array so far). Variables were also useful as a way of walking through all of the elements of an array in conjunction with a *while* loop.

We learned how to search for an element with a given property in an array. We also saw how to find the tallest element of an array. And we looked at a visualization of a famous sorting algorithm, known as an insertion sort.

Important concepts in this chapter

- An *array* is an organized collection of values and objects.
- The items in an array all have the same type (such as object).
- An object in an array is accessed through a *location index*.
- We can iterate through arrays, one element at a time.
- A *variable* is a piece of storage that stores a value.
- It is often useful to have variables store temporary information about the elements in an array.
- Variables can be used, in conjunction with a while loop, to iterate through an array.
- The objects in an array can be sorted.

10 Projects

1. Bubble sort

Create a world that simulates the steps of a bubble sort to sort the objects by height. A bubble sort is the following algorithm for sorting an array.

Go through the array comparing all neighboring elements in order. Compare array[0] and array[1]. Then compare array[1] and array[2], etc. If any two elements are out of height-order, call the swap method to swap their contents. When you are done, repeat this process. Continue repeating this process until you compare all neighbors without having to swap any of them!

Hint: Add an extra object to the screen and make it invisible. With each walk through the array, make the object visible if you have to do any swaps. If you go through the entire array and the object is still invisible, the bubble sort is done. Otherwise, make the object invisible and go through the array again.

2. Selection sort

Create a world that simulates the steps of a selection sort to sort the objects by height. A selection sort is the following algorithm for sorting an array.

Go through all of the elements of the array, and find the smallest. Swap that element with the element in array[0]. Then, starting with the element at location 1 of the array, find the smallest element (do not consider the element at location 0). Swap that element with the element in array[1]. Continue this process until the array is sorted!

11 Revisiting Inheritance: Mutable variables

The chapter returns to characters and inheritance, as used in Alice to create new classes of objects. In chapter 4, we first introduced the concept of creating new character models (classes) where new methods were defined for the characters and the characters were saved out with a new name. Of course, the new character class inherited all the methods of the original character (the base class) and had its new methods as well. In this chapter we look at another way we can use inheritance to create a new character.

This chapter contains only one section. In this section, character-level mutable variables are introduced. A *mutable variable* is a piece of storage that holds onto a value while our program is running. **The reason we say it is *mutable* is we can change the value stored in the variable as things happen during the program execution.** In this section, we look at adding mutable variables to a character. If we then save out the character as a new character, we are (once again) creating a new character using inheritance.

Note to instructors:

At first glance, it would seem that this chapter should immediately follow chapter 4. But, a major focus of this text is to make programming concepts easier to understand by relating the statements in a program directly to animated visible actions as the program runs. Because mutable variables have no visible representation, they have been avoided until late. You may, of course, decide to cover the topics in a different order as suits your style and preferences.

12 -1 Inheritance

In programs and projects presented in previous chapters we have made use of the technique of designing and creating new characters by writing new character-level methods that extend the capability of an existing character. Then, the character with its new methods is saved out as a new kind of character. The new kind of character has inherited all the properties and behaviors of the original character, but now has additional capabilities. In computer science terminology, this is a form of *inheritance*. In this section, we will look at the idea of a mutable variable, and how it can be used to support inheritance and the creation of new characters.

Character-level variables

In the last chapter, we introduced variables in a method to make array processing easier. Such variables are considered *mutable* because the value stored in the variable can be changed at runtime. From Alice's perspective, a mutable variable is a property of a character or of the world. New mutable variables can be added to a character. Then, if the character is saved out and given a new name, a new kind of character is created (similar to the way we created new characters by defining new methods). One thing we need to remember about a mutable variable is the variable is *not* directly visible in a virtual world. Although mutable variables have no visible representation when a program is running, they are sometimes quite useful as another way to extend a character's capabilities. This concept will be illustrated by means of an example.

Using a character-level method to track steering a car

Consider an interactive world that simulates steering (driving) a car. The challenge is to create a car that can be steered, perhaps to drive along city streets, or as part of a racecar game, where the player needs to steer the car to drive along the road and avoid obstacles. Figure 11-1-1 illustrates such an initial scene, with a City Terrain and a car added to an initial world.



Figure 11-1-1. An initial scene

The world is interactive in that the player uses the left, right, up, and down arrow-keys on the keyboard to steer the car along the road. That is, the car's front wheels turn right and left

(assuming the car is front-wheel drive) in response to a right or left arrow-key press and the car moves forward and backs up in response to an up or down arrow-key press.

The problem we must solve in designing and writing the program code for this animation is not how to turn the wheels right in response to, say, the user pressing the right arrow key. Rather, the problem is how to keep track of how much right (or left) the front wheels have been *previously* turned, so that when the player later presses a key to move the car forward, the car will also turn to the right/left the appropriate amount. In other words, it is necessary to somehow keep track of how much the front wheels have been turned right or left. A character-level variable may be used to help to solve this problem.

Implementation

There are four directions of steering motions. So, four character-level methods will be created -- forward, backward, right, and left. The methods will be created as character-level methods and the character saved out as **SteerableCar**. Then, *SteerableCar* objects can be used in other worlds.

The simplest methods to write are those to move the car forward and back. (The *right* and *left* methods are discussed below.) Moving the car forward or back consists of moving the car and rotating all four of its tires. "Rotating the tires" is a phrase meaning that each wheel will turn to simulate the motion of tires as the car travels. The number of times each tire must turn around is computed using the same technique as described for rolling a ball in the world presented in Chapter 6. The *numTurns* question is illustrated in Figure 11-1-2.

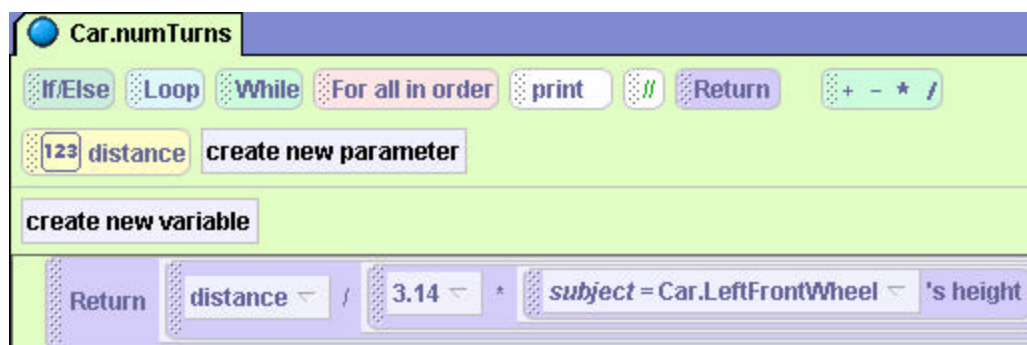


Figure 11-1-2. The *numTurns* question

A generic *moveIt* method, as presented in Figure 11-1-3, will move the car an amount specified by a parameter. Note that specifying a negative value for the *howfar* parameter will result in the car moving backwards by that amount (rather than forward).

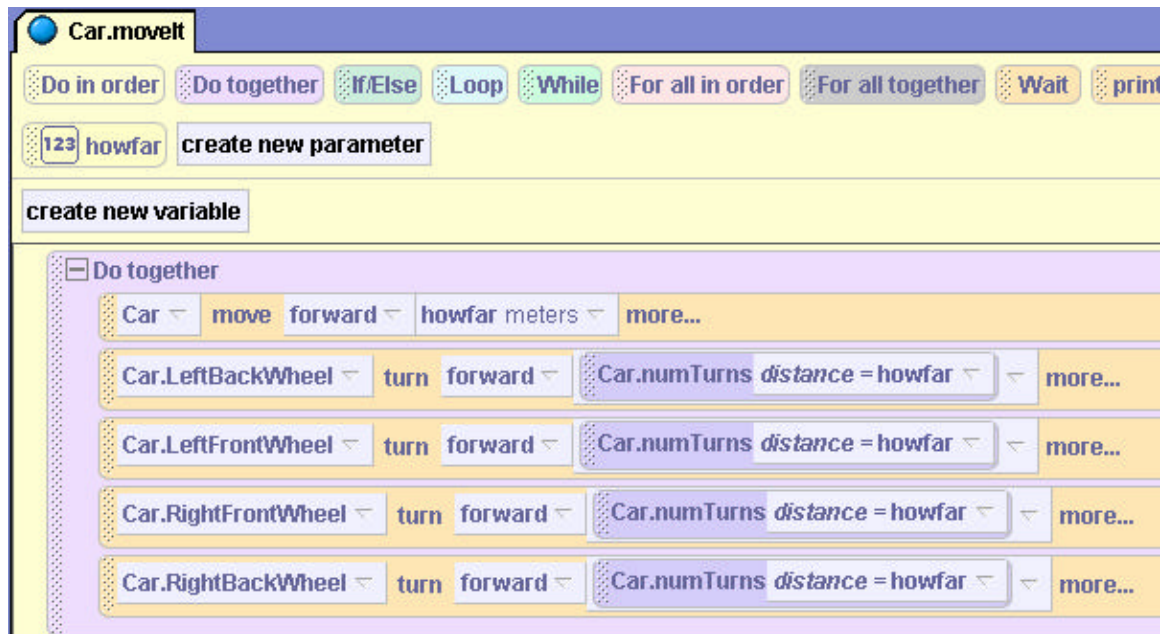


Figure 11-1-3. The *moveIt* method

With the *moveIt* method now written, the code for *forward* method (Figure 11-1-4) and the *backup* method (11-1-5) become easy to write.

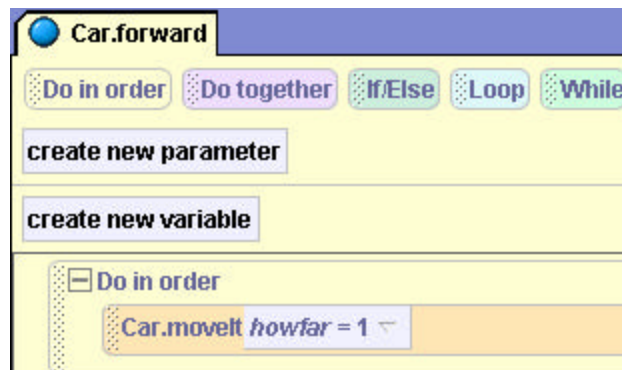


Figure 11-1-4. The code for *forward* method

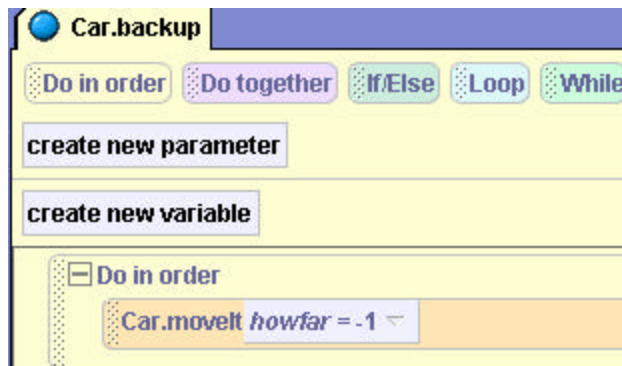


Figure 11-1-5. The code for *backup* method

The *right* and *left* methods are more challenging. The reason is that turning the wheels right or left has an effect on what happens when the car then moves forward or back. For example, if the wheels are turned left and then the car moves forward then the net result is that the car not only moves forward but also moves to the left.

So, in addition to turning the front wheels of the car right or left, the *right* and *left* methods also need to somehow keep track of the fact that the car wheels have been turned right or left, so that when the *forward* or *backup* method is then invoked the car will turn right or left in addition to moving forward or back. A first attempt at the *right* method is illustrated in Figure 11-1-6. However, while it does turn the front wheels right, it does not indicate the amount the wheels have been turned to the right.

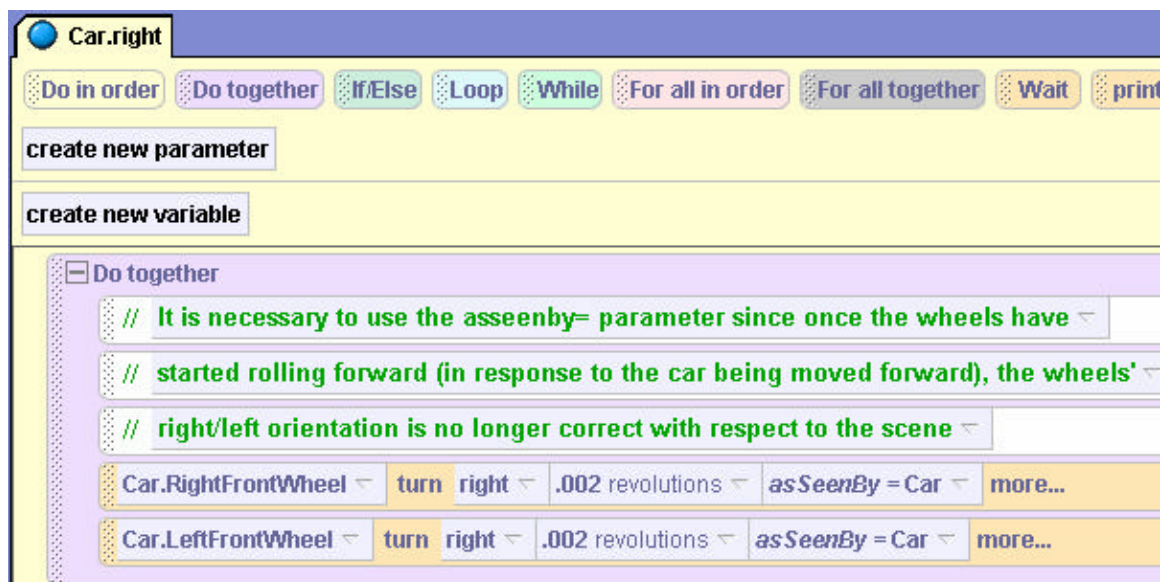


Figure 11-1-6. A first attempt at *right* method

To accomplish this goal (of tracking the amount right, or left, the wheels have been turned), a character-level variable is needed. To create a character-level variable, click on the car in the object-tree, and then select its properties. Click on the “create new variable” button, as illustrated in Figure 11-1-7. This creates a new character-level variable.

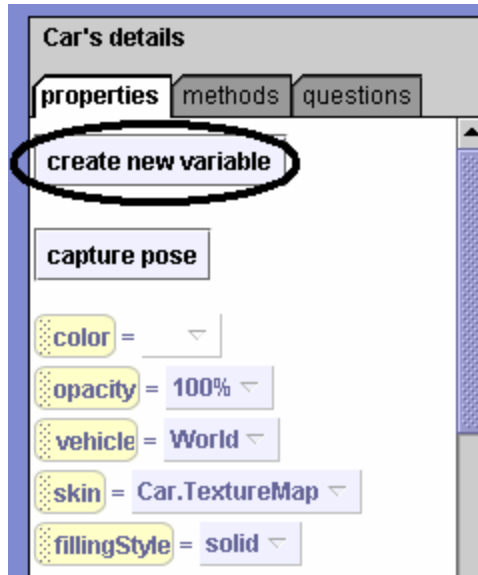


Figure 11-1-7. The car's properties

Name the variable *amountRight*, and select **Number** as its type and its value as 0, illustrated in Figure 11-1-8.

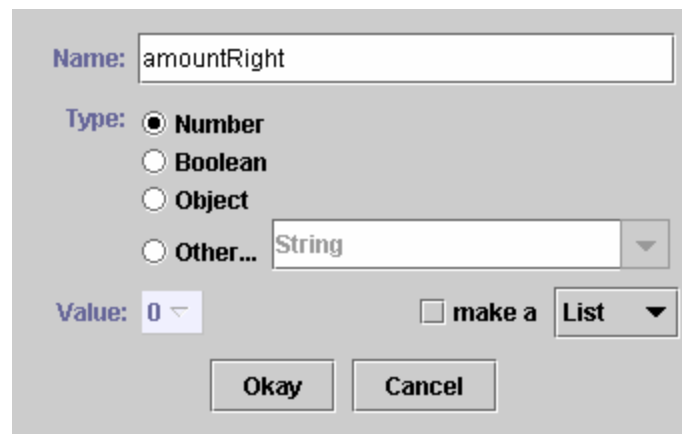


Figure 11-1-8. Creating the car's character-level variable

The idea of the *amountRight* variable is the following: Every time the user wants the car to increase the amount it should turn right (by clicking on the right arrow key), the variable *amountRight* should be incremented by 1, up to some maximum value (cars have a maximum amount the wheel can be turned). The when the car moves forward, it can also turn right by an amount specified by the variable. In other words, if the user clicks on the right arrow 5 times, the value of the variable *amountRight* should be set to 5. Then, when the user clicks on the up arrow to move the car forward, the car should also turn right as it moves forwards, five times the amount it would turn right had the user pressed on the right arrow key only once. It is possible to increment the value of the variable in the method *Car.right*, as illustrated below in Figure 11-1-9.

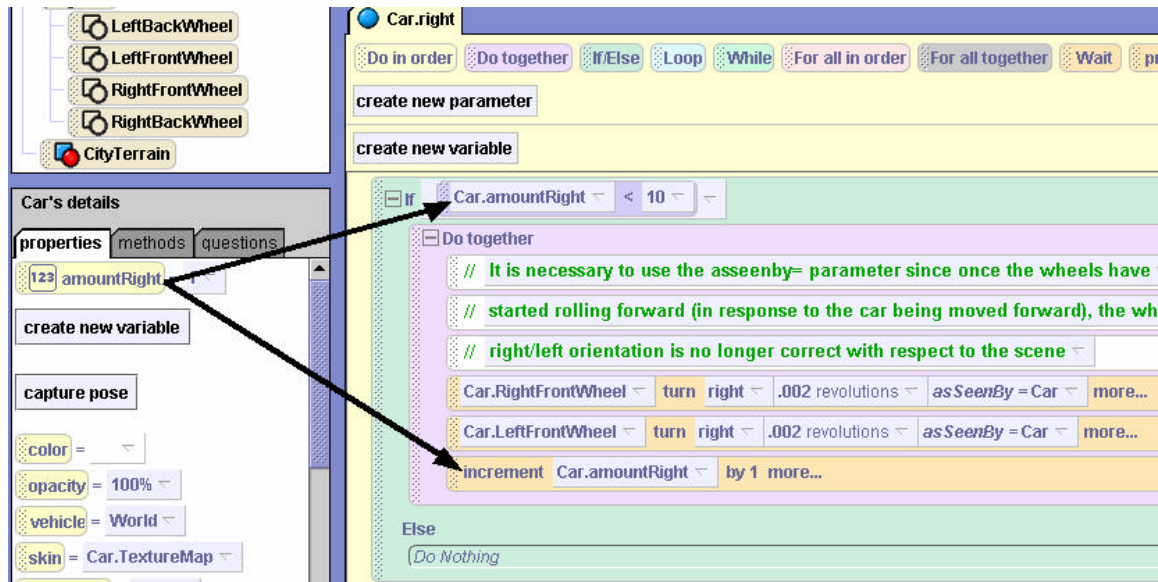


Figure 11-1-9. The *Car.right* method uses the *amountRight* character-level variable

The variable *amountRight* has been dragged into the method twice. The first occurrence is in the if statement – if the wheel has been turned right less than 10 times, it is ok to turn it further to the right. In effect, this allows the if-statement to set a limit of 10 as the maximum amount (right/left) that the steering wheel can be turned in succession. Note that 10 turns is an arbitrary amount but is reasonable. As will be seen in the final code (Figure 11-1-12), each turn of the steering wheel is two degrees. Ten turns would be 20 degrees, which is a reasonable turn of the steering wheel in a fast-moving car.

The second occurrence of the *amountRight* variable is in the increment method call (the last line in the if-part of the if-else structure).



Figure 11-1-10. Invoking increment on *amountRight*

Increment adds 1 to the value in *amountRight*. The effect is that each time this method is called, the value in the *amountRight* variable is incremented by 1.

Now it is time to look at the *left* method. It could be constructed in a similar manner with a character-level mutable variable called *amountLeft*. But, on thinking about the situation, it becomes clear that turning the steering wheel left is actually the opposite of turning the steering wheel right. In a way, turning the steering wheel left 1 time removes the action of turning the steering wheel right 1 time. This insight leads to the idea that the value in *amountRight* can be incremented when the steering wheel is turned right and decremented when the steering wheel is turned left. So, an *amountLeft* variable is not needed.



Figure 11-1-11. The *left* method

With this in mind, the *left* method is written as displayed in Figure 11-1-11. This method is quite similar to *right* method with only three small changes. The first change is the *if*-statement checks for *amountRight* being greater than -10 . This is reasonable because a turn to the left decreases the value in *amountRight*. Just as the *right* method is limited to 10 (actually $+10$) turns, the maximum amount left the car can be turned has been arbitrarily set to 10 (actually -10) as well. The second change is to turn the wheels left instead of right. The third change is to decrement *amountRight* by 1 (instead of increment).

Finally, the *moveIt* method needs to be updated to allow the car to turn right/left as it is moved. The modified *moveIt* method is illustrated in Figure 11-1-12.

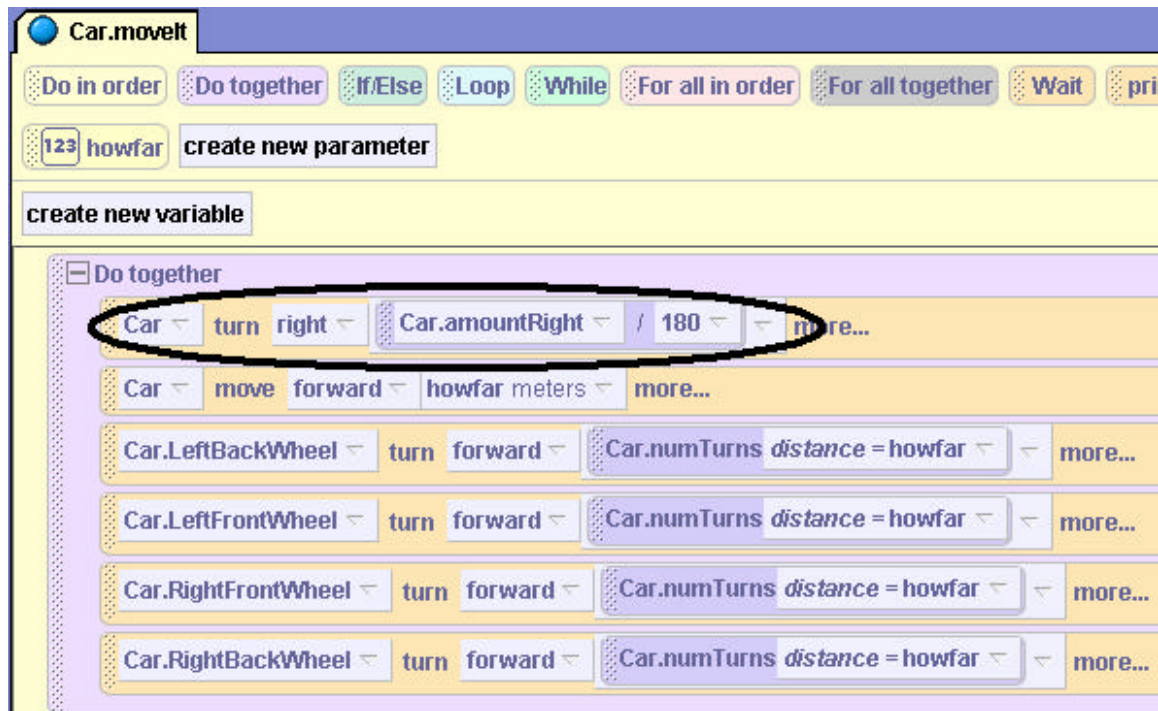


Figure 11-1-12. The modified *moveIt* method

The car turns two degrees to the right multiplied by the value of *amountRight* as it is moved forwards or backwards. Again, this value has been chosen arbitrarily but it seems to give a reasonable performance.

Note that the car must be renamed and saved out before it can be used in other worlds.

11-1 Exercises

1. SimpleSwitch

Some examples and exercises in this book have used a switch as part of an interactive world. Several switches are shown below. A switch object allows the user to use a mouse-click-and-drag to turn a lever on the switch. Of course, we intend the mouse-click-and-drag operation (to move the lever on the switch) to create an event. Then, an event-handler method can be written to respond to the event.



A mouse-click event tells us that the user has clicked the mouse on the lever, but this does not tell us whether the switch has been moved to an on or off position.

Two techniques can be used to solve this problem. One technique is to position an invisible sphere on the end of the handle and then check whether the sphere is above or below the middle of the switch. If the switch is above the invisible sphere, the switch is on – otherwise it is off. A second technique, which eliminates the need for an invisible object, is to add a Boolean variable to the switch's properties. Then, an animation is created to move the switch's lever to on and off positions.

Create a simple world containing two objects—a switch and some object that performs one action when the switch is turned on and a different action when the switch is turned off. Add a Boolean variable to the switch's properties (name it *On-Off*). Write a character-level event-handler (for the switch object) named *switchOn-Off* that responds to a mouse-click on the switch. The method should use an *if/else* block to check if the value of the *On-Off* variable is *true*. If *On-Off* is *true*, make the object perform its On-animation, rotate the switch handle $\frac{1}{2}$ revolution, and set the *On-Off* variable to *false*. If *On-Off* is *false*, do the opposite and set *On-Off* to *true*.

2. BinarySwitches

Digital electronics is based on the use of high and low currents to represent binary values. A high current flowing through a circuit represents the digit 1 and a low current represents the digit 0. Thus, a single circuit represents a single **binary digit** (a **bit**). A nibble is a combination of four circuits (4 bits) put together to represent numbers in base 2 (binary numbers). A byte is eight circuits (8 bits).

As a review, let's compare our base 10 number system to the base 2 number system.

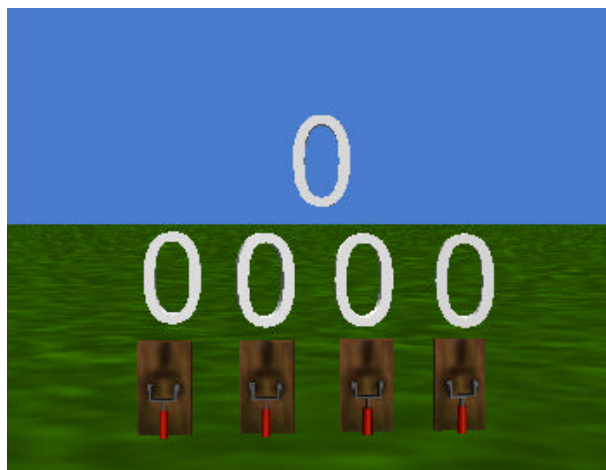
A base ten number is created using digits 0 through 9. The following example breaks down the base-ten number 5932:

$$\begin{array}{cccc}
 10^3 & 10^2 & 10^1 & 10^0 \\
 5 & 9 & 3 & 2 \\
 \\
 (5 * 10^3) + (9 * 10^2) + (3 * 10^1) + (2 * 10^0) = \\
 (5 * 1000) + (9 * 100) + (3 * 10) + (2 * 1) = \\
 5000 + 900 + 30 + 2 = \\
 5932
 \end{array}$$

Like base-ten numbers, the first column from the right represents 2^0 or 1, the second column represents 2^1 or 2, the third represents 2^2 or 4, and the fourth 2^3 or 8. Here is an example of how the base-two number 1001 is converted to its base-ten counterpart 9:

$$\begin{array}{cccc}
 2^3 & 2^2 & 2^1 & 2^0 \\
 1 & 0 & 0 & 1 \\
 \\
 (1 * 2^3) + (0 * 2^2) + (0 * 2^1) + (1 * 2^0) = \\
 (1 * 8) + (0 * 4) + (0 * 2) + (1 * 1) = \\
 8 + 0 + 0 + 1 = \\
 9
 \end{array}$$

This exercise is a simulation of a byte of computer memory. Create a world with four switches, each with an *On-Off* Boolean variable. Above each switch, insert 3D Text set to be the number 0, and centered above these four 0s add an additional 0, as shown below. Create an event-handler method to set the 3D text above a switch to 0 when the switch is off (lever down) or to 1 when the switch is on (lever up).



The top-level 3D text digit is to be the base-ten representation of the binary number represented by the zeros and ones below it. Write a method called *findResult* that converts the base-two number represented by the switches to a base-ten number and change the top level 3D text to this base-ten number each time a switch is clicked and switched. The easiest way to make this work is to create a number variable for each 3D text object that represents the digit value to which its text is set. Also, modify the on-off event-handler method so that in addition to changing the 3D text above it to 0 or 1, it also sets its value variable to the corresponding 0 or 1 and then calls the *findResult* method.

Hint: Use the following formula: $(a * 8) + (b * 4) + (c * 2) + (d * 1)$, where a, b, c, and d represent the 0s and 1s in the appropriate columns. In the World's questions under **string** you will find the option <what> **as a string**. This will convert the integer result to text that can be displayed by the base-10 3D text.

3. SteerableCar.

Create the interactive car world, as discussed in this chapter. When you have the car steering properly, modify the car object by adding the ability shift gears, thereby increasing or decreasing the speed of the car. Make sure that the speed can never go below 0, and that it is given a reasonable maximum limit. Add events and event-handlers so that when the user presses 'F', the car's speed is increased (the car goes Faster), and when the user presses 'S', the car's speed is decreased (the car goes Slower). (This may mean that the car's forward and backup methods will both need modification.) The **SteerableCar** is useful for many projects. Save the car object as **SteerableCar**.

4. CarRace

In Alice, create a world containing an obstacle course, either using the city terrain or perhaps enlarging the stadium. Add a *SteerableCar* object, using the *SteerableCar* character created in the previous exercise. If the user steering the car hits an obstacle, the speed should be reset to 0, and perhaps the car should be backed up some distance. Using the world-level "time elapsed" question, add a loop to allow the user to steer the car for 30 seconds. (Or, you can just use a Wait instruction for 30 seconds.) If the user has completed the course by the time 30 seconds is up, the user "wins" a prize (perhaps by making visible some object appear on the screen or playing a victory march tune). If the user has not completed the course by the end of the 30-second period, an appropriate 3-D message should appear, perhaps "Better luck next time!"

11 Summary

In this chapter, we looked at the use of *character-level* variables. Character-level variables are used to store information concerning the particular character. We call this *state* information. Character-level methods are often called *mutable* variables because they are used to track changes in the state of characters in the world as the program executes. Character-level methods are associated with the character, and can be used in any of the character's methods.

Important concepts in this chapter

- Mutable variables can be useful for extending the capabilities of a character.
- Like character-level methods, character-level variables are saved out with the character and reused when the character is added to other virtual worlds.

12 Cool Worlds You Can Build

This chapter has a very different flavor from the rest of the book. No new concepts are introduced. The worlds in this chapter are larger, more dramatic animations that “put it all together.” Projects found at the end of each chapter have focused on the concepts presented in that chapter. But, here we draw together several key concepts from multiple chapters. Hopefully, you will have as much fun with the worlds in this chapter as we have had!

12-1 Cool worlds

A flight simulator

We have presented simple worlds with very simplistic methods for flying a biplane as examples in a previous chapter. But, this example was far short of a flight simulator. Let's build a fancier, high-powered version of a flight simulator as a game. Our goal is to design a flight simulator where the player (user) must steer a plane through five rings to win the game. We will not be concerned with the steps involved in the biplane takeoff or landing.

Initial world with billboard instructions

Figure 12-1-1 shows a possible initial world. The screen shot looks a bit cluttered because we have added a billboard of instructions on how to play the game and the billboard overshadows the objects in the scene. A switch is positioned immediately in front, so the user can click the switch after reading the instructions and the billboard will go away. In addition to the billboard and the switch, the world has a biplane and 5 rings through which the user is supposed to steer the plane. Also, we have added a helicopter, some trees, a windmill and a gazebo as background scenery, to make the flight simulation more interesting. A six-pack of soda (not visible in the initial scene below) is in the world –as a prize for winning the game.



Figure 12-1-1. Initial Scene

Planning the simulation

In thinking about and planning this simulation, we considered several problems:

- 1) The user is supposed to fly the biplane through each of the five rings. How do we know when the plane is flying through a ring?
- 2) When should we end the game?
- 3) What happens if the plane flies into the ground?

- 4) How do we keep the plane from flying out of view of the camera?
- 5) How do we start the game and then continue to fly the plane and keep track of the game?
- 6) How does the user steer the plane?

Let's break down the overall task into each of these subtasks, using stepwise refinement to develop a solution. This way, we can create small storyboards for each individual piece of the overall simulation.

How do we know when the plane is flying through a ring?

Certainly, we can use the *distance to* question to determine when the plane gets close to a ring. An advantage of using *distance to* is that this question measures the distance from the center of the ring – this is exactly what we want! After the plane flies through the ring, we can simply make the ring invisible to indicate that the plane has flown through the ring. The storyboard pseudocode below describes a method to check whether a plane is flying through a ring. In this design, if the plane is flying through a ring, the ring is made invisible. As the plane is flying through a ring we should also check to see if this is the last ring to fly through. If all the rings have now been “flown through,” we can say the user has won.

<pre>If the plane is within 1 meter of a ring Make the ring invisible If all rings have been “flown through” Indicate that the user has won</pre>

The method that implements this design, named *CheckForCollisionWithRing*, appears in Figure 12-1-2. The particular ring being checked is passed as a parameter. The condition in the *If* statement checks if the plane is within 1 meter (of the center) of the ring and also checks that the ring hasn't already been made invisible (opacity != 0). If these conditions are true, the plane is flying through a visible ring. To reward the player for having flown the biplane through a ring, a sound is played and the ring tumbles as it disappears.

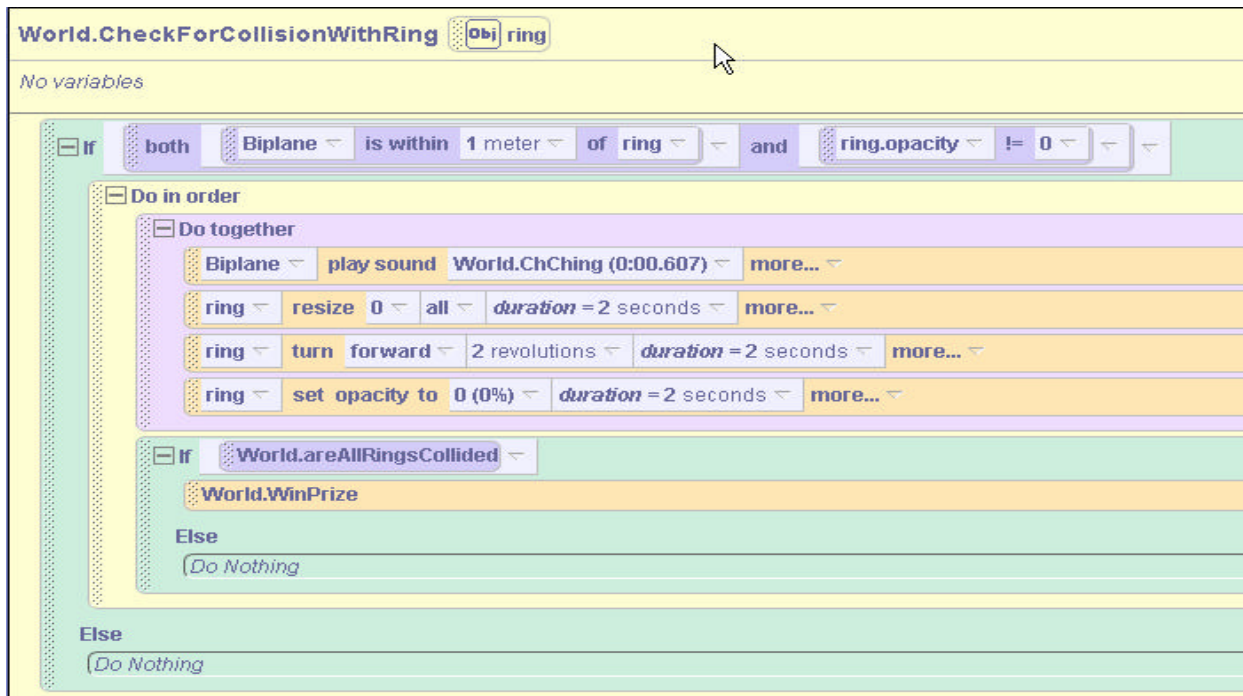


Figure 12-1-2. Checking to see if the plane is flying through a ring

Of course, this method just checks for one ring and we need to check all five rings. To do this, let's write another method named *RingCollisions* that calls *CheckForCollisionWithRing* five times, passing a different ring each time.

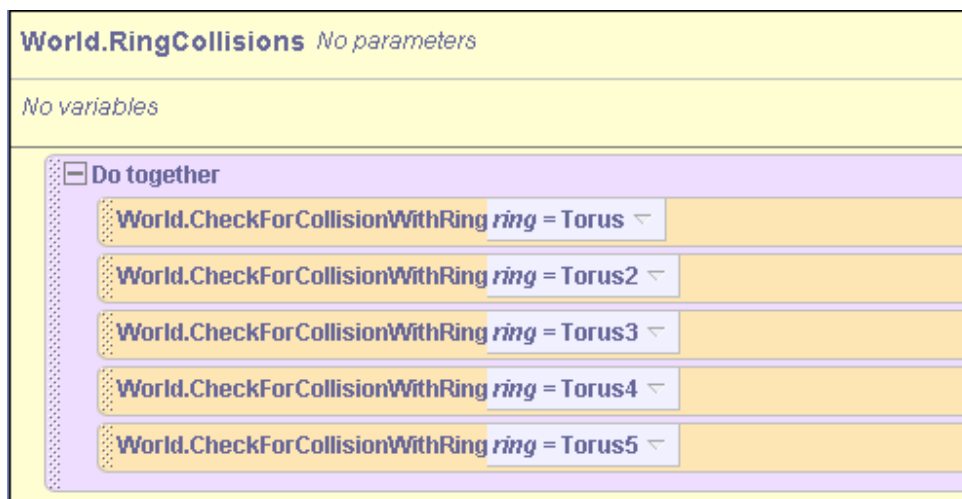


Figure 12-1-3. Check all five rings

When should the game end?

The game should end when all five rings have been made invisible. A question that answers whether all five rings have been made invisible is shown in Figure 12-1-4.

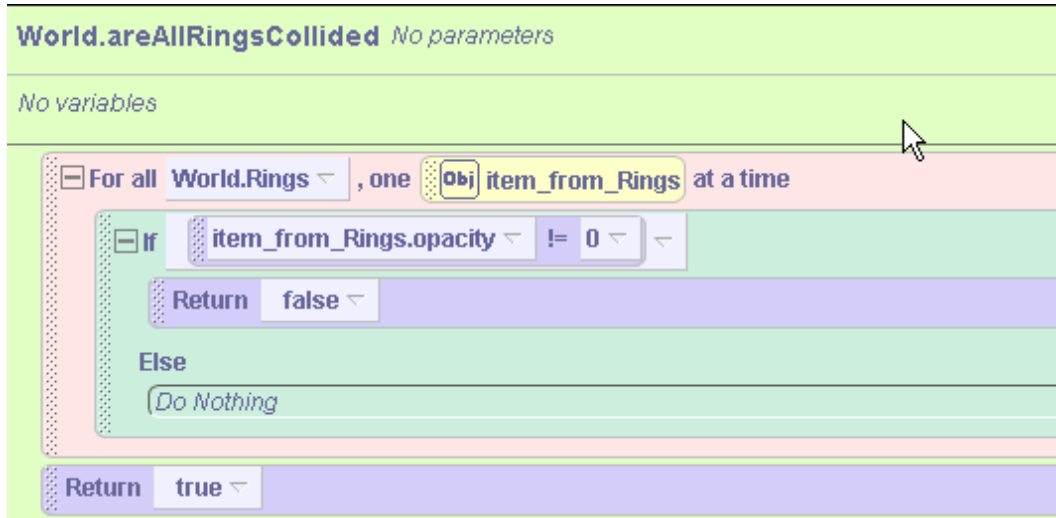


Figure 12-1-4. A question to determine if the plane has collided with all five rings

The code uses a list of all five rings, named *Rings*, and iterates through the list. If any ring is still visible (opacity != 0), the question returns *false* (meaning that not all of the rings are invisible). If all the rings are invisible the question returns *true*, indicating that the game should now end.

The code to end the game is presented in Figure 12-1-5. The prize six-pack of soda appears on the screen, and the sound of applause is played.

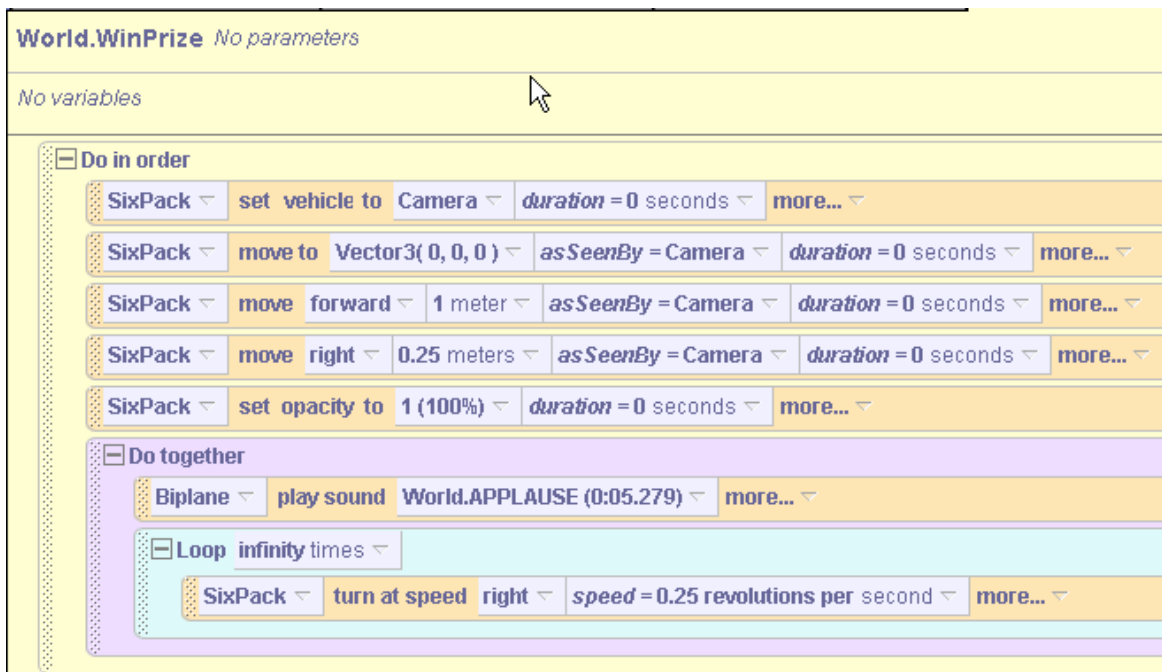


Figure 12-1-5. Ending the game

What happens if the plane flies into the ground?

Well, the plane should crash and burn! After which, we decided to put the plane back into the air and start it flying again. This is a common strategy in video games – to give the player another try at it. (Naturally, you can decide for yourself what to do about this.)

A method named *TestForCrash*, shown in Figure 12-1-6, implements this strategy. If the front of the plane gets below the ground, a crash occurs. A loud noise is made, and there are flashes of red and black everywhere! Then, the plane is moved back to its original position, 3 meters above the ground, and it is able to start flying again.

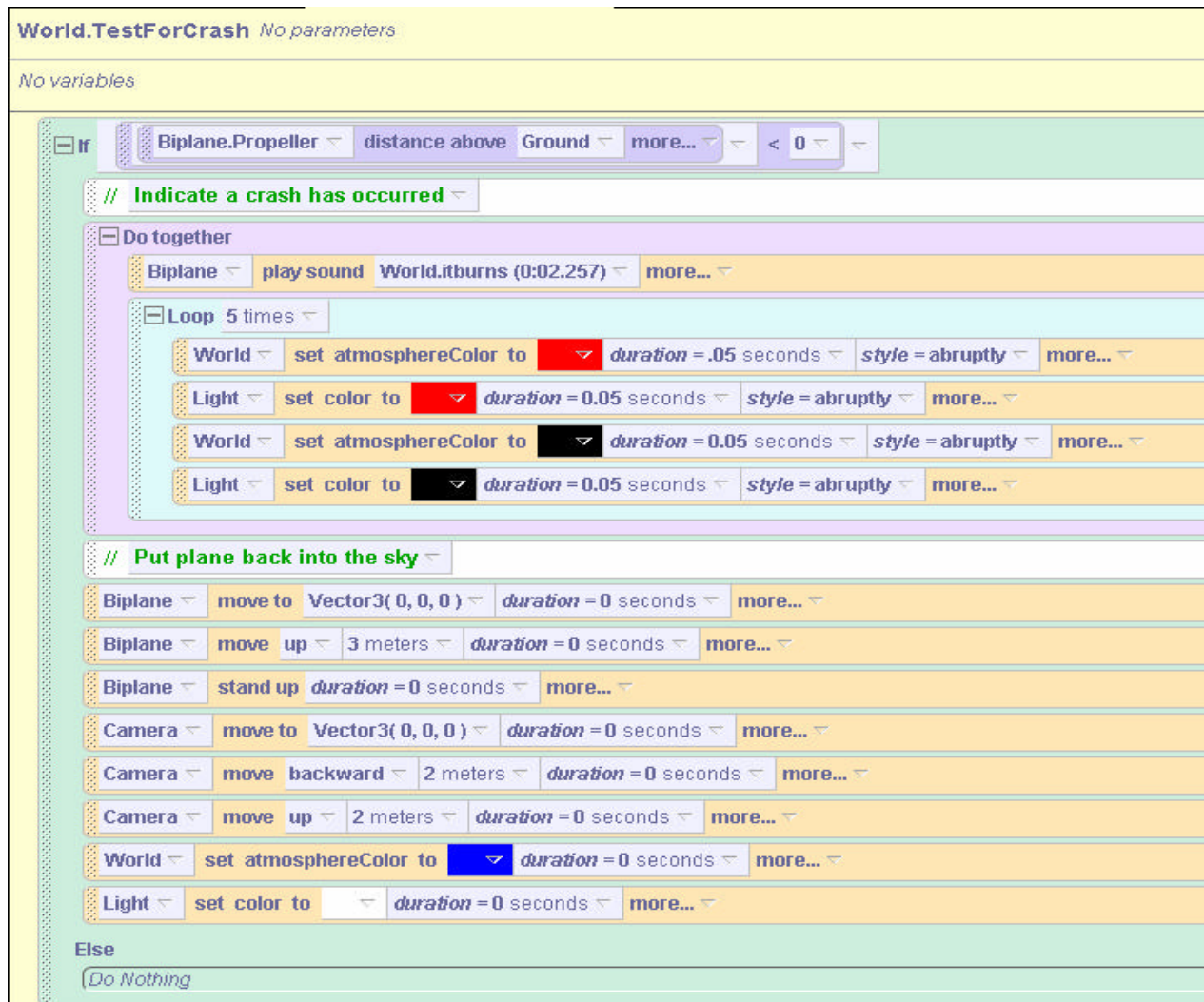


Figure 12-1-6. Crash and recover

How do we keep the plane from flying out of view of the camera?

A potential problem in a flight simulator is that the player can steer the plane out of view of the camera. This is not a new situation, we have seen this sort of thing in previous worlds. The solution presented here is to create a method that has the camera point at the plane whenever it gets out of view (or far away from the camera). The *FollowPlane* method in Figure 12-1-7 accomplishes this goal.



Figure 12-1-7. Having the camera follow the plane

How do we start the game, continue to fly the plane, and keep track of the game?

Starting the game is easy. Create an event for *when mouse is clicked on switch*. When this event occurs, the billboard of instructions and the switch should fade away and the plane should begin flying. Once the game begins, we must constantly maintain the game by continuing to fly the plane forward while checking whether the plane has crashed or perhaps flown through another ring. **The key programming strategy is to nest infinite Loops and *Do together* instructions.** The infinite loops keep everything going and the *Do together* instructions make it all happen at the same time. The storyboard pseudocode is shown below.

```

Make the instructions and switch invisible
Do together
  Loop an infinite number of times
    Move the plane forward
  Loop infinite number of times
    Check for a crash
  Loop and infinite number of times
    Keep the camera pointing at the plane
  Loop an infinite number of times
    Check to see if the plane has flown through any rings

```

The code that implements this storyboard is illustrated in Figure 12-1-8 below. To add more interest to the animation, instructions are included to move the helicopter and windmill blades in the background. The plane is made to continually move forward by using a *Biplane.move at speed* instruction inside the first infinite loop.

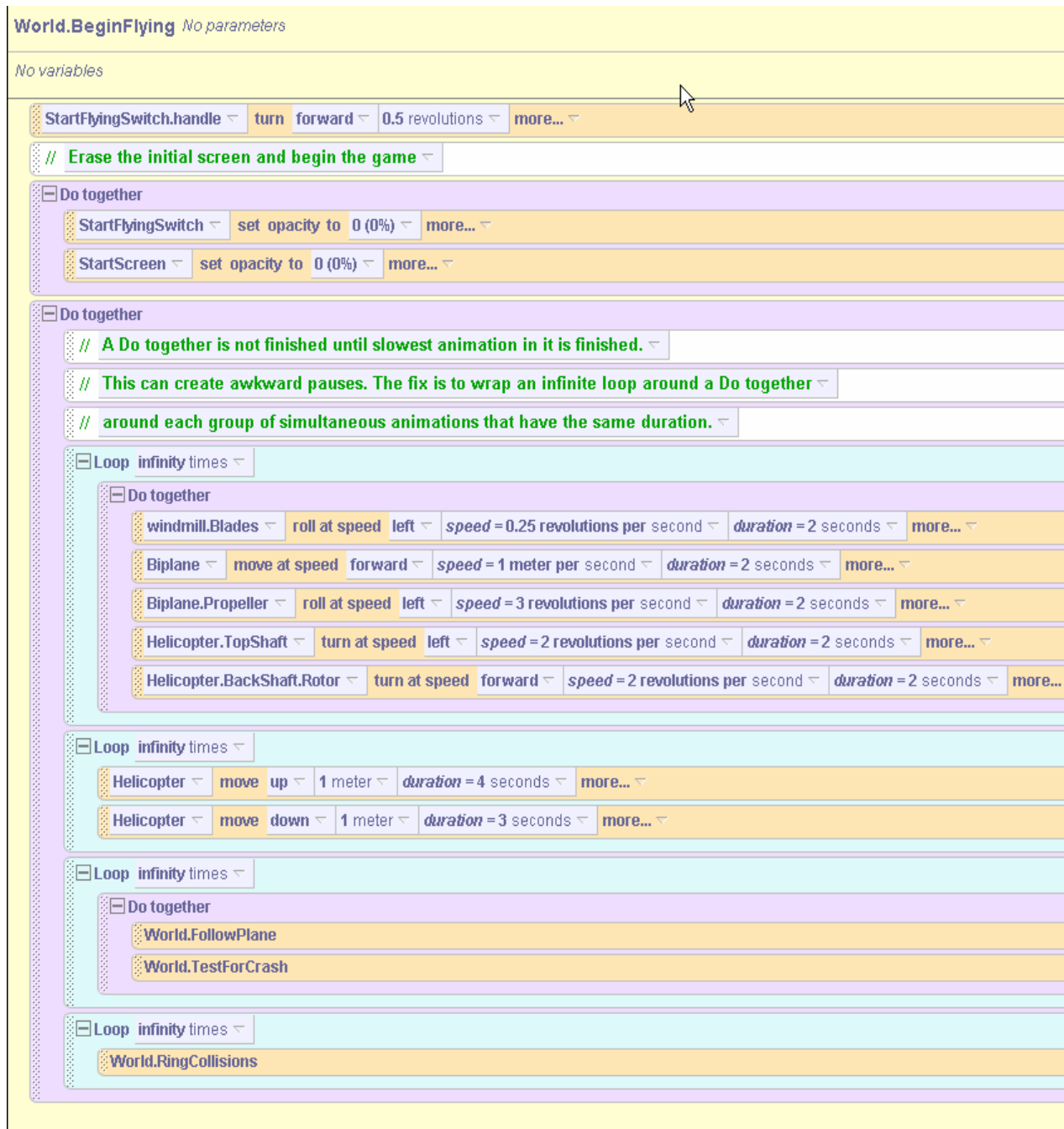


Figure 12-1-8. The main driver method for checking what has happened to the plane

How does the user steer the plane?

Five keys are used to allow the player to steer the plane through the rings (left, right, up, and down arrow keys and the space bar). For each arrow key, a *while key is pressed* event is created in the Events editor and a turn at speed instruction is created in the *During* part of the BDE construct. The *turn at speed* instruction, in conjunction with the BDE, will continue the plane turning in the appropriate direction (up, down, right, or left) as long as the appropriate key is

pressed. As soon as the key is released, the plane continues in the direction it was heading. The Events editor is shown in Figure 12-1-9 with the events that allow user interaction with this world.

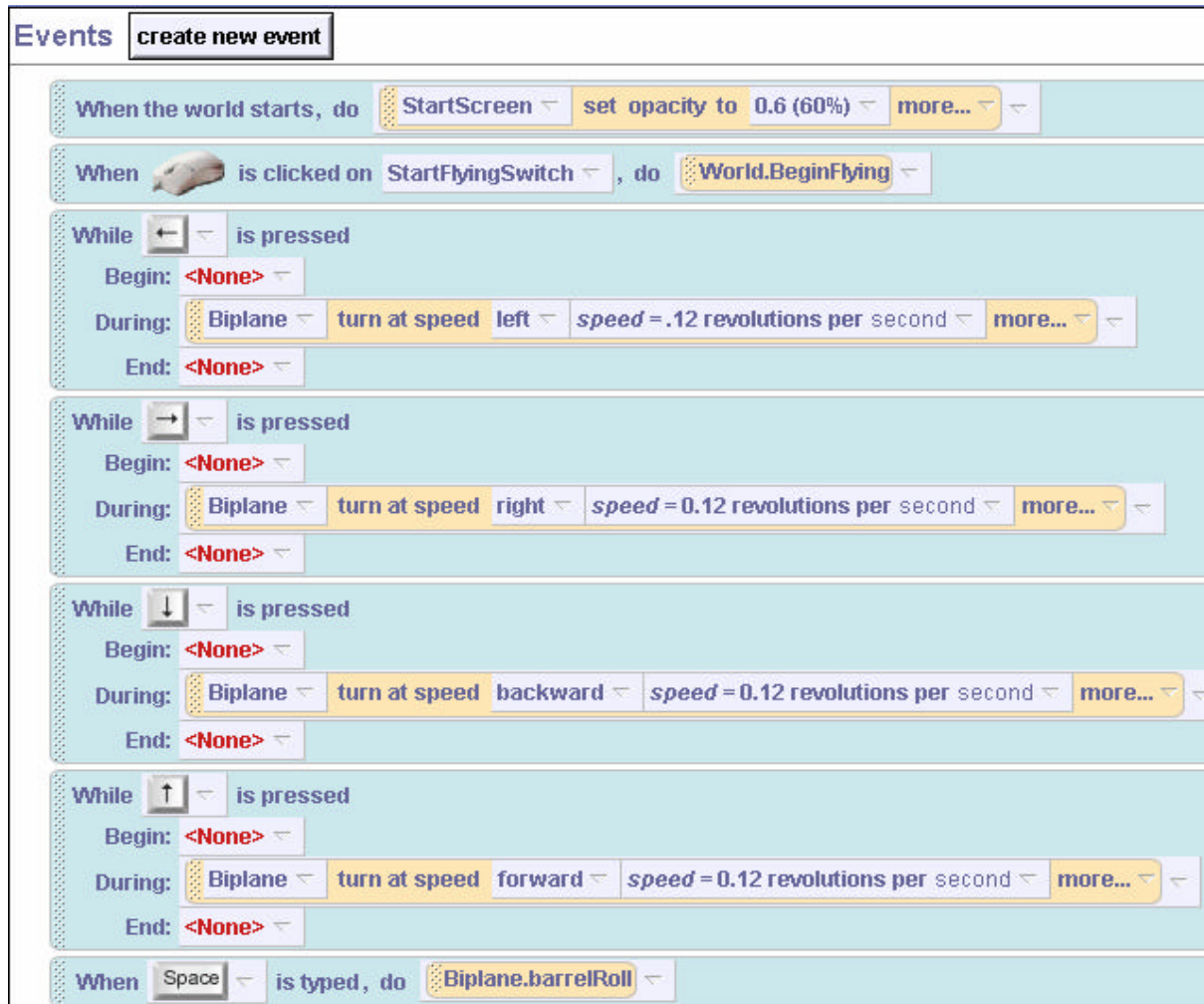


Figure 12-1-9. Events that allow user interaction with the flight simulator

An additional event is created for the space key. When the user presses the space key, a barrelRoll instruction is called. Figure 12-1-10 contains the code for the barrel roll, a fun rotation of the plane when the user presses the enter key. It does not affect the rest of the animation, but it is fun!

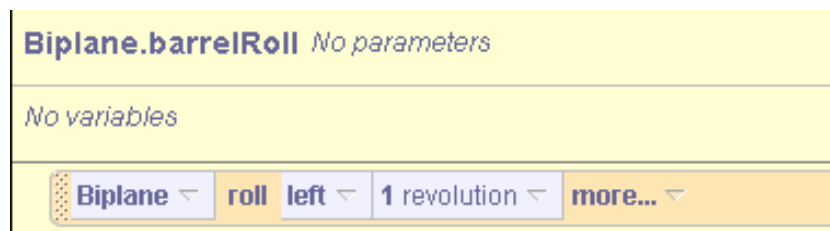


Figure 12-1-10. The code for the plane's barrel roll

Example 2: Create your own adventure game

(example to be added)

13 Exercises

- 1) Create and modify the flight simulator presented in this chapter, to have the biplane crash if it hits the helicopter, windmill, or any of the other background objects.
- 2) Modify the flight simulator to make the flying more realistic. When the plane is pointed down, towards the ground, its speed should increase. Conversely, when it is pointed up, its speed should decrease. You may wish to use a character-level variable *planeSpeed*, to keep track of the speed of the plane. Such a variable may be used in a similar fashion to the example in Chapter 12 where a variable was used to control the amount the wheels of a car turn.
- 3) Modify the flight simulator to add a timing element to the world. In other words, the user playing the game must fly through all 5 rings in a specified amount of time, or the user the game ends and the player loses the game.

13 Transition to Real World Languages

We expect that many students who are using this book as a text will decide to take another course in programming. If you are one of these people (and we sincerely hope you are), you will need to make a transition from Alice to another programming language. As we write this book, the most likely language for such a transition is Java, C++, or C#. The purpose of this chapter is to help you make that transition by connecting the concepts you have learned about programming in Alice to the concepts of programming in other object-oriented languages.

The Alice programming language has a distinct object-oriented flavor but differs in some ways from Java, C++, and C#. The reason for the differences between Alice and these real world languages is clearly the underlying goals of the respective language designs. So called “real world languages” are designed for general-purpose programming of workhorse applications in a world of commerce, Internet communication, and scientific research. Alice, on the other hand, has been designed to achieve two very different goals: (1) to provide a programming environment for 3D graphic animations, and (2) to provide a high-impact visualization tool that can enjoyably be used for learning to write computer programs. Nonetheless, the designers and implementers of Alice are aware that many who learn to program with Alice will continue on to study other programming languages. With this in mind, the Alice development team has made an effort to maintain a level of consistency with other object-oriented languages.

We begin in section 13-1 with a look at syntax. Alice provides a **syntax switch** that, by default, is *off*. You can turn the syntax switch *on* to display the program code with details of syntax (such as curly brackets, parentheses, commas, quotes, and semicolons). The syntax displayed is “Java-like.” Programs look like programs that are written in Java.

In section 13-2, we examine the use of objects and classes in Alice as compared to real world languages. In Alice, a “what you see is what you get” approach to classes, objects, and their properties is presented, while in real world languages classes are often more abstract and information presented to the user is often textual. In this section, the creation of new objects and classes in Alice is compared to class definition and object instantiation in Java code.

In section 13-3, common program constructs in Alice are compared to constructs in Java and C++. Alice supports both a sequential, traditional order of program execution as well as an interactive, event-driven flow of control. The program constructs in Alice are very similar to those in Java and C++, providing a foundation for a smooth transition to programming with real world languages.

Note to instructors: It is not the purpose of this text to explain the differences in philosophy that distinguish Java, C++, C# and other real world languages. In this chapter, classes, objects, and other aspects of Alice are detailed with transition in mind, rather than a full discussion of Java or other languages. For example, static classes and interfaces are not considered in our discussion of examples of class definitions in Java.

13-1 Transition to Java/C++: Syntax Switch

The process of learning to program in any programming language requires a mastery of the *syntax* of the language as well as the programming *concepts and constructs* supported by the language. This section will provide a bridge between the syntax of Alice and the syntax of Java and C++.

Syntax

Syntax is a technical term for the rules that specify how words and punctuation marks in a language can be used to put together statements to communicate some action or idea. For example, consider the following statement in a natural language (English):

The frog moves forward 1 meter.

In the English language, the rules of syntax say that a statement begins with a capitalized word and ends with a dot, question mark, or exclamation point. Also, the order in which words appear in the sentence must meet certain rules of order. An example of such a rule of order is:

`<noun phrase> <verb phrase> <modifier phrase>`

In this rule, the general order of words is a noun phrase ("The frog") followed by a verb phrase ("moves") followed by a modifier phrase ("forward 1 meter"). Thus, the example statement above meets the order specified by the given syntax rule.

As with natural languages, computer programming languages also have rules of syntax. Statements must begin with certain kinds of words, which are followed by other kinds of words, which are followed by other kinds of words...and so on. Furthermore, punctuation marks such as commas, quotes, semicolons, and parentheses are sprinkled liberally throughout. It is important to understand that natural languages are very flexible, allowing great variation in style and punctuation. But, programming languages have rigid rules of syntax and allow for very little flexibility. If a statement written in a program does not meet the rules of syntax, the compiler (which is used to validate the syntactic correctness of the program as well as to generate an equivalent program written in a language the computer can understand) rejects the whole program.

Syntax rules can be complicated and novice programmers are often frustrated by having to learn where all the punctuation marks go and the order in which the words must be positioned. In Alice, the editor for constructing statements is purposely designed to lessen the burden of syntax. Figure 13-1-1 illustrates the same example statement in the Alice editor.

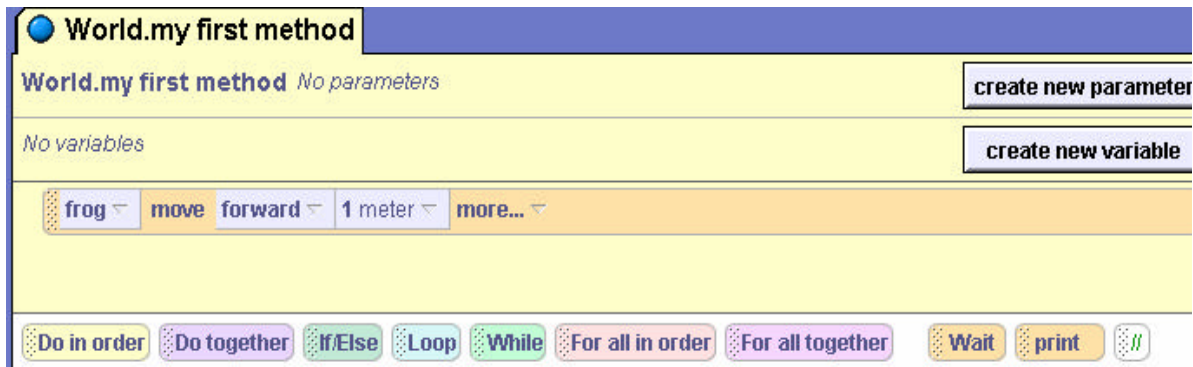


Figure 13-1-1. Alice statement syntax example

The syntax rule for the order of words in an Alice statement is:

`<object_name> <method_name> <parameters>`

The object_name is "frog", the method_name is "move" and the parameters are "forward" and "1 meter." Note that the "more..." at the end of the statement is NOT actually part of the statement. Instead, it is an Alice **editorial tag** that can be mouse-clicked to allow the selection of additional optional parameters from a pop-up menu. These parameter options are used frequently in examples throughout the text. Additional parameters of this method include items such as *duration= 2 seconds* and *style= abruptly*.

The drag-and-drop editor in Alice was designed to work with the rules of syntax to automatically put the words in acceptable syntactic order. The editor is said to be *syntax-sensitive*. Also, by default, the editor leaves out punctuation marks, relying instead on color and visual tiles to separate parts of the statement. The power of the syntax-sensitive editor allows the novice programmer to concentrate on how to solve problems and write programs without having to deal with the details of syntax, particularly all the commas, parentheses and semicolons.

In making a transition from Alice to real world languages, you may want to see all the punctuation marks. To show all the punctuation marks, Alice provides a syntax switch. The punctuated syntax can be obtained by first selecting **Edit** from the main menu bar at the top of the Alice interface and then selecting *JavaLikeSyntax.py* in the **Preferences | General** menu, as illustrated in Figure 13-1-2. The syntax resources option acts as a toggle-switch that can be turned on and off to suit the programmer's preferences. To return to the default syntax display, simply set the resources switch back to the *Default.py* setting.

(Note: On your computer, setting the syntax switch may require that Alice be shut down and then restarted for the syntax switch change to take effect.)

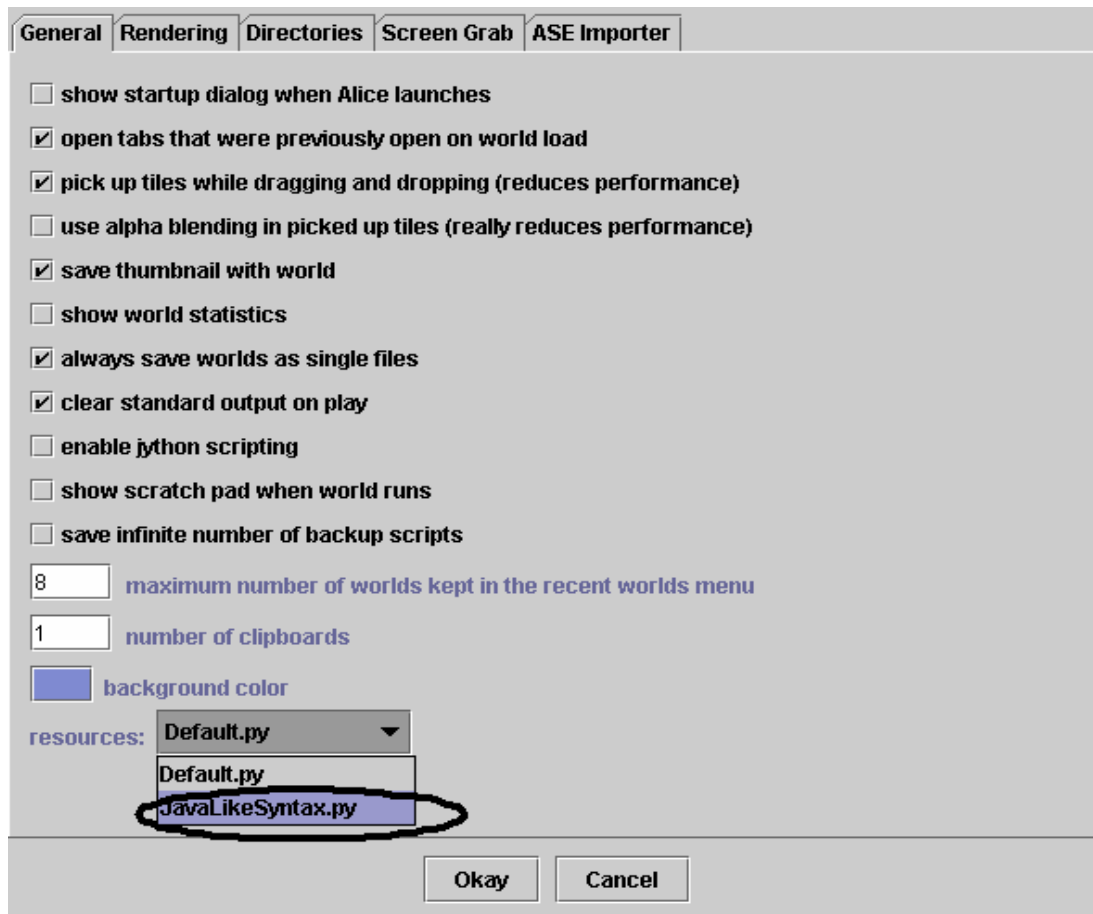


Figure 13-1-2. Setting the Java-like syntax switch

Once the syntax switch has been set, statements in the editor are more detailed, including the standard punctuation symbols as would appear in other object-oriented languages. The *frog move forward* statement with the syntax switch turned on appears as shown in Figure 13-1-3.

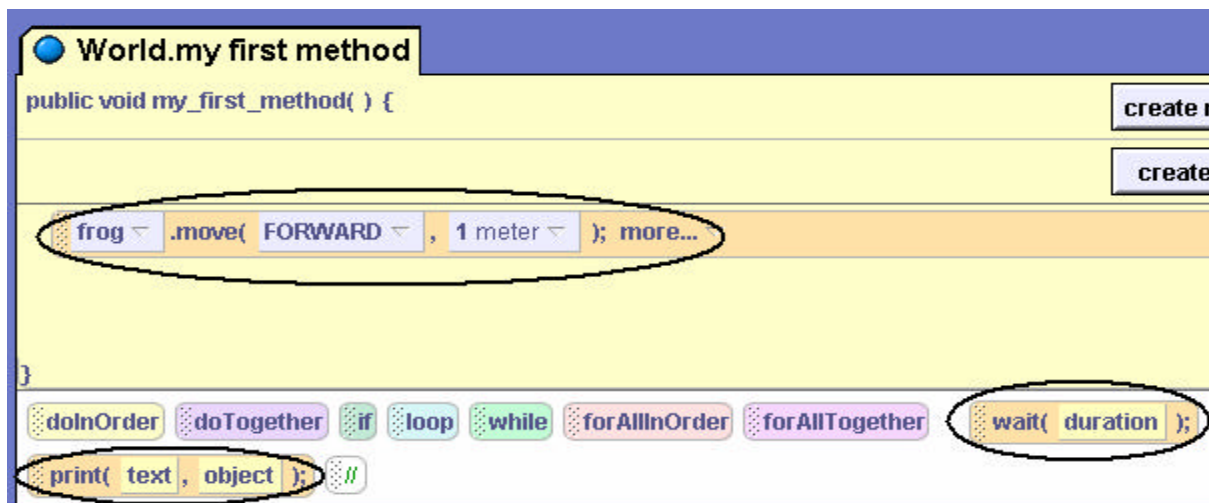


Figure 13-1-3. Syntax switch on

The overall statement is still the same as before, but now it includes all the punctuation marks. A dot "." has been placed between the object_name "frog" and the method_name "move". The parameters are now enclosed in parentheses and separated by commas, "(FORWARD, 1 meter)". And, the entire statement now ends with a semicolon ";". Once again, the "more..." at the end of the statement is an Alice **editorial tag** for selecting parameters -- not actually part of the statement. In effect, the syntax rule with punctuation included is:

```
<object_name> . <method_name> (<parameter> , <parameter> , ... );
```

Look closely at the editor box in Figure 13-1-3. Notice that some items in the interface itself have also changed. One change is the "wait" and "print" tiles are now displayed as "wait(duration);" and "print(text, object);" to show the syntax of parameters for the wait and print methods. This is because each of these operations requires a parameter.

A more striking change is the line of text at the beginning of the method. This line of text, called a *method signature*, is circled in Figure 13-1-4.

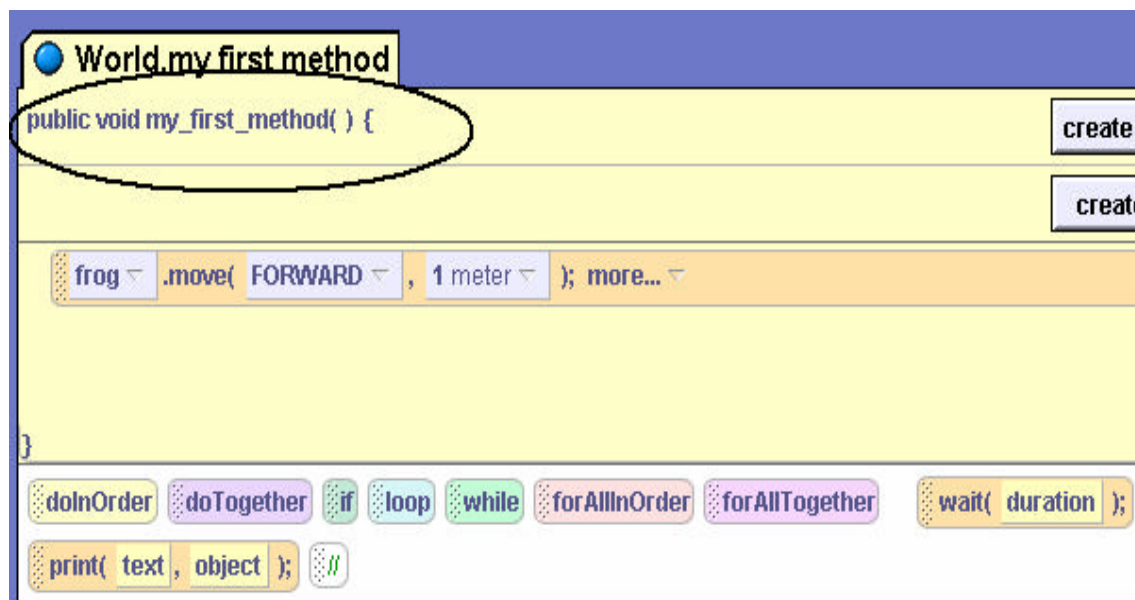


Figure 13-1-4. The method's signature

The method signature specifies information to the system about how to run the method. In particular, it specifies what type of information needs to be passed to this method as well as what type of information is returned. In many object-oriented languages, a program is understood to start running by first calling a main method. The first method (the main method) is generally expected to be *public* and *void*. *Public* means that it can be called from anywhere such as from the operating system. (In Alice, all methods are public.) *Void* means that it does not return any information. In other words, it is NOT a question (function) that sends back some information to a calling method. (All methods in Alice are void, as they do not return information.) Finally, notice that two curly brackets have been added to enclose the method statements, as circled in Figure 13-1-5. Again, this is consistent with real world object-oriented languages. A method is a

block of code (potentially several instructions) and blocks of code are marked by enclosure in matching curly brackets.

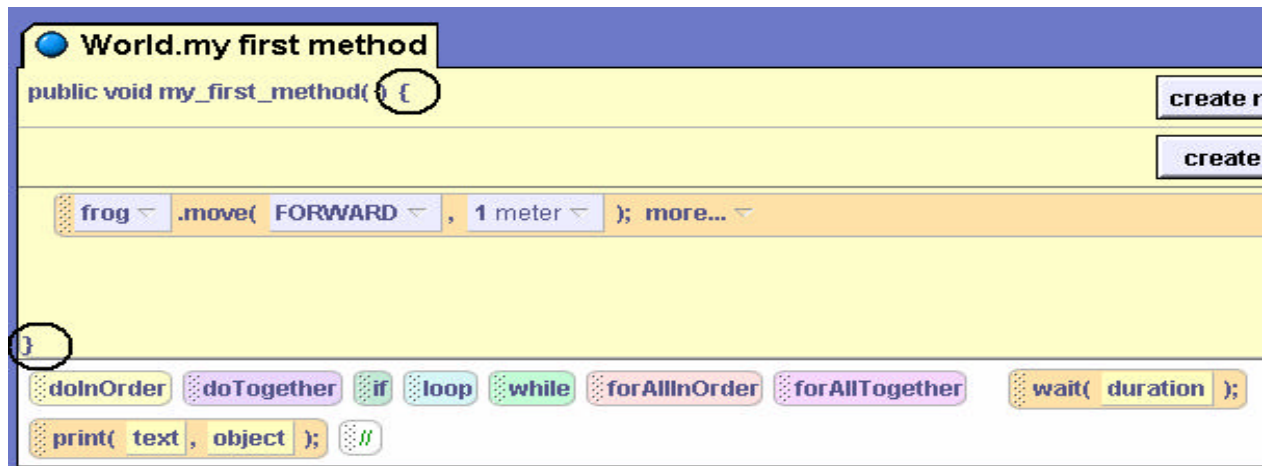


Figure 13-1-5. Curly brackets enclose a code block

13-2: Objects and Classes

Fundamental concepts in all object-oriented programming include those of *objects* and *classes*. Intuitively, an *object* is anything that can be identified as unique from other things. Objects are identified by having (1) a name, (2) some number of properties, and (3) the ability to perform certain actions or carry out specific tasks. An Alice object is rendered as a 3D image of a rabbit, a person, a car, or some other selection from the gallery. In Java, C++, and C#, an object is represented by a textual name in the program code and is dependent on an abstraction mentally envisioned by the programmer.

Objects

In Alice, as in object-oriented languages, objects are instantiations of *classes*. An as example, in Figure 13-2-1, **Person** and **Dogs** are classes. **Joe**, **Stan**, and **Cindy** are *instances* of the **Person** class while **Spike**, **Scamp**, and **Fido** are *instances* of the **Dogs** class. In Figure 13-2-2, Larry, Lila, and Louis are all instances of the Lemur class. (Note that these lemurs have been renamed.)

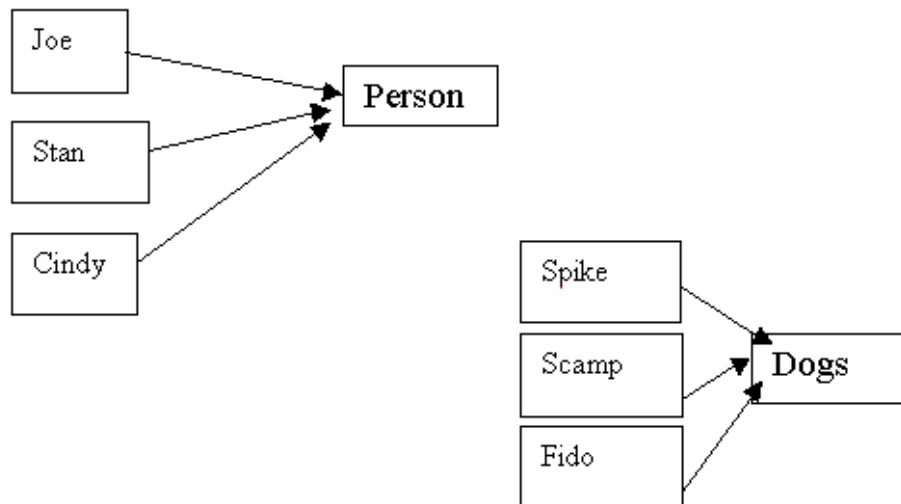


Figure 13-2-1. Organizing objects into classes



Figure 13-2-2. Objects of the same class in Alice

All objects of the same class share some commonality. All Person objects have properties such as a name, two legs, two arms, height, and eye-color. And, all Person objects can perform walking and singing actions. All Dogs objects have properties including a name, four legs, height, and fur-color. And, all Dogs objects have the ability to run and bark. But, while each object belongs to a class, it is still unique in its own way. Joe is tall and has green eyes. Cindy is short and has blue eyes. Spike has brown fur and his bark is low growl and Scamp has golden-color fur and his bark is a high-pitched yip. In the world shown in Figure 13-2-2, Larry, Lila and Louis are all objects of the same Lemur character class but the objects are different in that Larry has short arms, Lila is darker colored, and Louis is short.

Classes

In Alice, classes are predefined as 3D models provided in the gallery. The 3D models are categorized into groups such as Animals, People, Buildings, Sets and Scenes, Space, and so on. Figure 13-2-3 shows some of the model classes in the Animals collection.

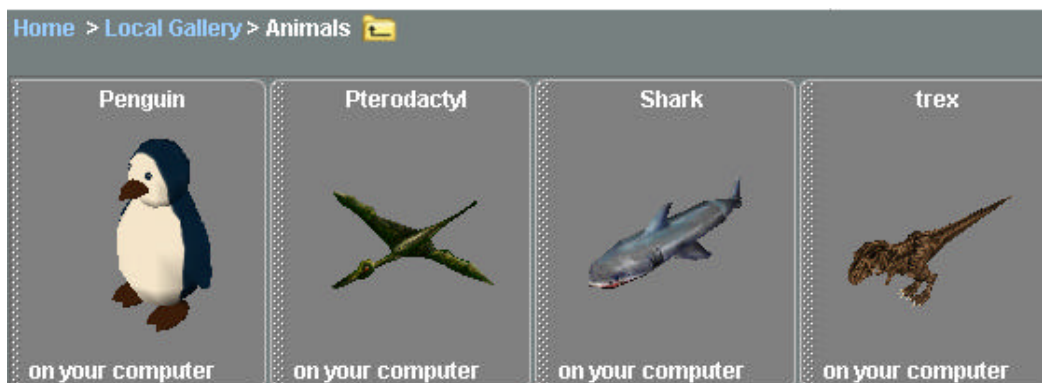


Figure 13-2-3. 3D Models in the Animals collection

As with the Alice gallery, real world languages also provide pre-defined classes, stored in *libraries*. Libraries provide textual definitions of classes as utilities for input/output operations (involving text and number values), math functions, string manipulations, drawing graphic figures, and others. Some of these classes are designed to provide support routines and others are designed to provide special features such as data structures and widgets for window applications. To use one of the library classes, the program code generally must *include* the class. For example, in C++

```
#include <iostream.h>
#include "myClass.h"
```

The angle brackets tell the C++ compiler that the library class to be included can be found in the standard library provided with the system. The quotation marks are used to designate a library class that was written by the programmer (or obtained from some other source) and can be found in the same directory as the program being compiled. In Java, the include statement is named *import*. For example, the Java statement

```
import java.applet.*;
```

imports the well-known Java Applet class for creating applications that can be run on a web page. The "." at the end of the statement indicates that all classes in the directory are to be imported.

Creating (instantiating) objects

In Alice, selecting a picture from the gallery and dragging it into the world scene constructs an object of that class. Suppose, for example, an animation is to be created which uses a Snowman. The 3D model of a Snowman is found in the gallery and dragged into the world, as shown in Figure 13-2-4. (This figure was borrowed from the Tutorials in Appendix A.)



Figure 13-2-4. Constructing an object from a 3D model class in Alice

The Snowman can then be named (actually, renamed, as the process of dragging it into a virtual world also gives it a default name) by right-clicking on the Snowman in the object tree and selecting the rename option. In C++ or Java, this process of object creation is sometimes called **new**ing an object. One important difference of Alice with respect to C++ and Java is that all objects in Alice must be constructed at the outset, before the program starts running. In other words, objects may not be constructed or destructed dynamically.

So, how are objects such as a snowman created in real world languages? Well, let's take a look at some code to define a snowman class and construct a snowman object in Java, written in Figure 13-2-5. The Java Snowman class defines a snowman as having height and width properties. Other properties are ignored in this simulation. The Snowman method (lines 4 – 7) is a constructor. A constructor method is invoked to create a Snowman object. In this example, *snowMan1* is a *Snowman* object, created by a call to the constructor in the first line of the *main* method. In Java, the *main* method is the first method run when this program is executed. So, the first thing that happens at runtime is the construction of a Snowman object.

```

public class Snowman {
    private double height;
    private double width;
    public Snowman (double h, double w){
        height = h;
        width = w;
    }
    public double getHeight() { return height; }
    public double getWidth() { return width; }

// Create snowman object and display its properties
    public static void main(String [] args) {
        Snowman snowMan1 = new Snowman(4.0, 2.0);
        System.out.println("The snowman height is "+ snowMan1.getHeight());
        System.out.println(" and width is "+ snowMan1.getWidth() );
    } //end main
} //end snowman class

```

Figure 13-2-5. Defining a class and constructing an object in Java

“What you see is what you get”

When a snowman object is created in Alice, we can see the object on the screen and view its properties in the properties tab, as shown in Figure 13-2-6.



Figure 13-2-6. Snowman object and properties

In Java, when the *snowMan1* object is created everything takes place behind the scenes. In the memory of the computer, variables are created where height and width values are stored that represent the properties of the snowman object. In the definition of the *Snowman* class in Figure 13-2-5, *getHeight* and *getWidth* methods were written. The methods can be called to display the values on the screen. When the Java code in Figure 13-2-5 is run, what you see on the screen is:

The snowman height is 4.0 and width is 2.0

Clearly, the representation of the snowman object in Alice is highly visual, while the representation of the snowman object in Java is textual.

Still, the representational distinction is not mutually exclusive. We should mention that it is possible in many object-oriented languages to create a drawing of a snowman object on the screen – **but this generally requires importing special graphics libraries and writing many, many lines of code**.

Another similar feature in Alice to C++/Java (actually to their debuggers) is the ability to watch variables. For example, we can create a *sizeFactor* variable for the snowman and set a watch on the variable for code that modifies the variable value, as in Figure 13-2-6. When the program is run, the change in the *sizeFactor* variable value is displayed as the program executes, as can be seen in Figure 13-2-7. One important difference is that Alice does not permit the setting of breakpoints in code. It is only possible to watch the variables as the program runs.

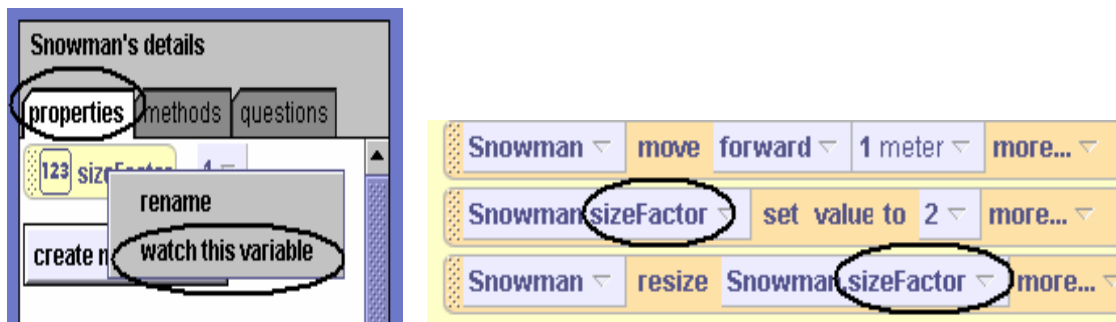


Figure 13-2-6. Setting watch on a variable used in the program code

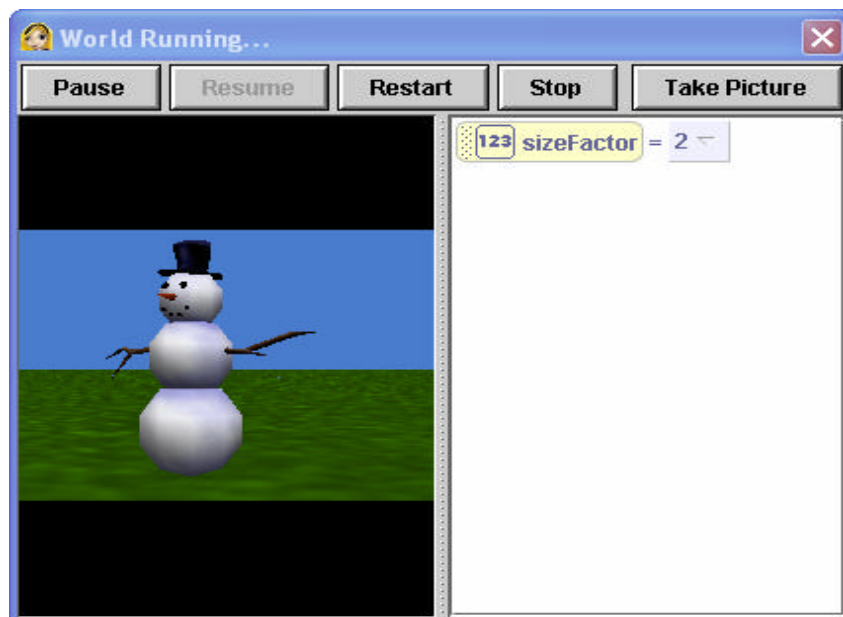


Figure 13-2-7. Watching the value of a variable as the code executes

Creating new classes

Suppose it would be desirable to create a Snowman that melts in response to an increase in the external temperature. To do so, a character-level *melt* method can be written to animate the meltdown based on some increase in temperature degrees, as seen in Figure 13-2-8. In the melt method, the snowman's dimensions are automatically adjusted by calling the *resize* method.

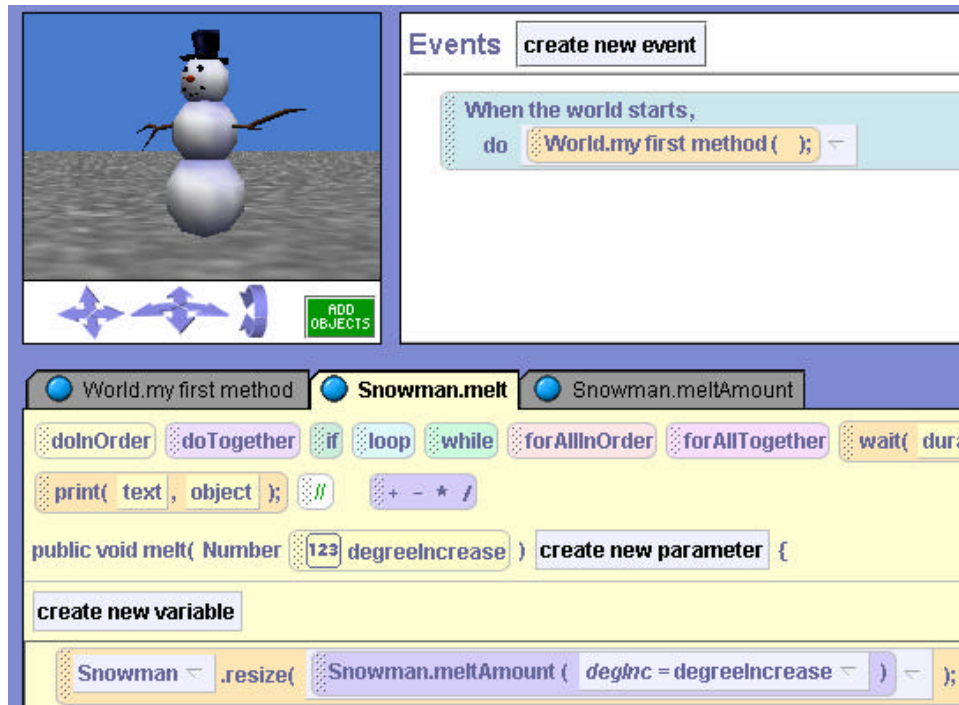


Figure 13-2-8. Snowman melt method

The built-in *resize* method expects to be sent a parameter indicating the amount by which the object will be resized. In this example, a user-defined question named *meltAmount* is called to compute the amount of melting for resizing. The *meltAmount* question, shown in Figure 13-2-9, computes the meltdown amount as a fraction of increase in degrees. When the animation is run, the user would see a greater change (melting) in the height and width of the snowman if the increase in temperature were 30° than would be seen for an increase of 10°.

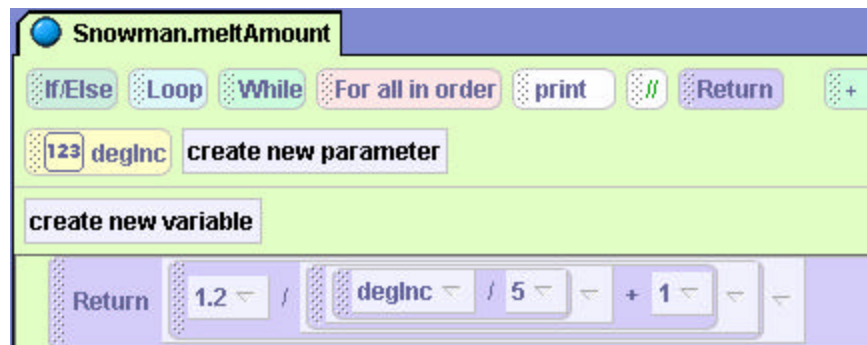


Figure 13-2-9. Question to compute amount of melting

Once new methods have been written for the snowman, we can save out the snowman and his newly-defined methods as a new character class. (A full explanation of how to write code to create new characters in Alice can be found in chapter 4.) Thus, Alice makes use of inheritance to create new classes. The reason classes are not built from scratch in Alice is the process of creating a 3D object in a 3D rendering package and giving it the basic animations of move, turn, roll, etc. is quite complex – quite beyond the intent of using the software to introduce programming to beginning programmers. Thus, modifying an existing character class to write code for creating a new class is the way to new classes are built in Alice.

How does this compare to creating a new class in real world languages? We have already shown how to create a new class of snowman objects in Figure 13-2-5. The class definition is typed into an editor and compiled (we hope). We can also create a new class in Java using inheritance. Let's create a `Snowman2` class that inherits all the properties and methods from the `Snowman` class and adds methods to *melt* the snowman and compute the *meltAmount*. The `Snowman2` class definition is illustrated in Figure 13-2-10.

```
public class Snowman2 extends Snowman {

    public void melt (double degreeIncrease) {
        height = meltAmount (height, degreeIncrease);
        width  = meltAmount (width, degreeIncrease);
    }
    private double meltAmount( double dimension, double degInc) {
        return dimension * ( 5 / degInc );
    }

    // Create snowman object and melt at 10 degree increase
    public static void main(String [] args) {
        Snowman2 snowperson = new Snowman(4.0, 2.0);
        snowperson.melt(10.0);
        System.out.println("The snowman height is "+ snowperson.getHeight());
        System.out.println(" and width is "+ snowperson.getWidth() );
    } //end main
} //end snowman class
```

Figure 13-2-10. Snowman2 inherits from Snowman class

Note that the *meltAmount* method is *private* (cannot be invoked from the *main* method). When the *melt* method is invoked, the snowman's height and width shrink. Of course, no snowman object is actually visible on the screen so it is impossible to see the change in size. Instead, the program prints a message to tell the user what the height and width of the snowman have become.

Encapsulation

In Alice as well as Java, each of the Snowman program examples makes use of a `Snowman` class. `Snowman` objects are created (constructed) as instances of the class. Although not illustrated in these particular examples, multiple snowman objects can be created and used. Each snowman object has its own properties and can perform certain actions (methods). The properties

of a snowman object are private *state* information for the individual snowman. Evidence of privacy of information is seen in the need to write *getHeight* and *getWidth* accessor methods in the Java example, to obtain the height and width of the object. (In Alice, we use built-in or user-defined questions to get private state information pertaining to objects.) One way to think about an object is to compare it to a capsule (a self-contained unit). The object is said to *encapsulate* its own properties and methods.

Data Types and Structures

Properties of objects in Alice can be any of several types: number, Boolean, string, color, texture, and location. Except for texture and location, these *data types* are common with object-oriented languages. Texture is an image map that may be painted on the surface of an object and location is the 3D coordinate position of the object within the world. And, Alice supports lists and arrays -- as do most object-oriented languages. Data structures are collections of data items, arranged in some order.

Methods and Functions (Questions)

In object-oriented terminology, the actions that can be performed on/by objects of a class are called *methods*. But, other names for these actions are commonly used to be more reflective of the operation performed. Commonly used terms are *modifiers*, *accessors*, and *functions*. *Modifiers* are methods that perform an action that changes the object in some way. In the above Alice and Java program examples, the *melt* method is a modifier method because it changes the dimensions of the object. *Accessors* are methods that retrieve information about some property of an object. In the Java program example above, *getHeight* and *getWidth* methods are accessor methods used to obtain the height and width for printing a message to the user after meltdown has occurred. These methods were not needed in the Alice program because the *resize* action is animated for the user and no text message was printed. *pure functions*, called *Questions* in Alice, are methods that perform some arithmetic or logical operation and return a value (e.g., number, Boolean, object) based on the result of that operation. (They do not change the state.) In the Alice and Java examples above, *meltAmount* is a function/question.

It is important to point out that the terminology discussed here is NOT consistently applied. In fact, in C++ all actions are considered functions -- whether a value is returned or not. Functions that do not return a value are labeled *void*. This is understandable in that C++ is an extension of the C programming language, where the fundamental program structure is a function.

13-3: Program Constructs

Program constructs are used to control the “flow of execution” – the order in which program instructions are carried out. In traditional forms of programming, the flow of execution can be sequential -- each instruction is executed one line at a time from beginning to end. Or, the flow of execution can branch (depending on conditions that exist at run time) or loop (repeat again and again). Modern programming languages are trending to yet another form of control, where the order of execution depends on interaction with the user. *Interactive* programs are said to have an *event-driven* flavor that affects the flow of execution. Event-driven programming is often used in many software applications that make use of windows, buttons, checkboxes, and other widgets for interaction with the user. In Java, AWT and Swing classes provide event-driven capabilities for building Graphical User Interfaces (GUIs). Some versions of C++ have similar libraries, for example Borland's C++ Builder.

Alice animations can be traditional. We call this kind of animation a *movie* because the animation is very similar to a cartoon or feature length animated film. But, Alice animations can also be *interactive*, allowing the flow of execution to be in the hands of the user. An interactive animation in Alice consists of methods that are linked to events (created by keyboard and mouse input).

In this section, we look at the program constructs used in Alice and how they compare to the program constructs used in Java and other real world languages. Once again, the purpose is transition rather than a full discussion of Java and C++ programming.

Do in order and Do together

The first constructs to consider are *Do in order* and *Do together*. *Do in order* is the default order of execution in Alice, corresponding to the traditional sequential flow of execution in Java and C++. A major difference is that Alice allows the programmer to explicitly specify that a block of instructions be run in sequence, while Java and C++ assume this is so. But, *Do in order* is needed in Alice to distinguish from a block of instructions that run simultaneously (*Do together*). Instructions that run simultaneously (or at least appear to do so) are said to be *concurrent*. Concurrency is a great advantage for computer animation but is not a common feature of most programming languages. The standard version of C++ does not have concurrency although many extensions add concurrency, often by extensive compiler extensions. Concurrency in Java exists in the form of *threads*, which provide a way to divide a program into separate, independently running subtasks. The *Do together* construct does not correspond exactly to Java *threads* (as Java threads can communicate with one another in ways not possible in Alice). The bottom line is that, in C++ and Java, *Do in order* is assumed and *Do together* is much more complicated.

Decision-making

The fundamental decision-making construct in Alice is *if - else*. An example of the construct, with the Java-like syntax switch turned on, is shown in Figure 13-3-1(a). A similar example for Java and C++ is shown in Figure 13-3-1(b). In the Java/C++ example, it is assumed that *iceSkater* is an object of a class that defines a *circle* method and a *distanceTo* function.

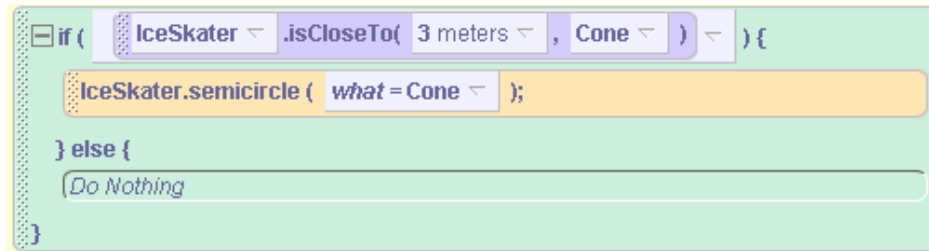


Figure 13-3-1(a). Alice If –Else construct

```
if (IceSkater.distanceTo(Cone) < 3) {
    IceSkater.semicircle(Cone);
}
```

Figure 13-3-1(b). Java and C++ If-Else construct

A difference between the *if-else* construct and the same construct in Java/C++ is that Alice always includes the *else* – even when it is not needed. (A default *Do Nothing* is used as a placeholder.) But in Java and C++, the *else* is omitted when not needed. Otherwise, the construct is consistent, making a branch in execution on the basis of the *true* or *false* value for the conditional expression. Java and C++ also support a *switch* statement, which is a form of specialized *if* statement. The *switch* statement does not add any computational power to these languages – it is merely a convenience.

Repetition: loop (for)

As with *if-else*, Alice constructs for repeating blocks of code are consistent with those in Java and C++. Only minor differences in syntax exist. The counted loop (*for* loop) construct in Alice is illustrated in Figure 13-3-2(a) where the syntax switch is turned on and in Figure 13-3-2(b) where the syntax switch has been turned off. A similar example in Java/C++ is in Figure 13-3-2(c), where we assume *bunny* is an object instance of a class that defines a *hop* method.

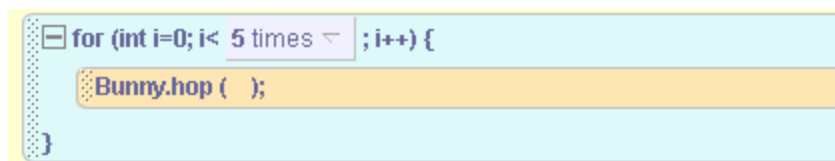


Figure 13-3-2(a). Alice *for* loop construct with syntax switch on

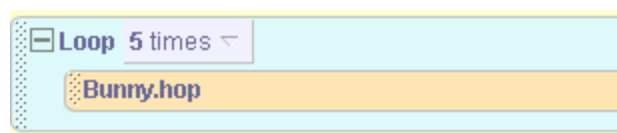


Figure 13-3-2(b). Alice *for* loop construct with syntax switch off

```
for (int i = 0; i < 5; i++) {
    bunny.hop();
}
```

Figure 13-3-2(c). Java/C++ *for* loop construct

Repetition: while

The *while* loop construct in Alice is illustrated in Figure 13-3-3(a) (syntax switch on) and a similar example in Java and C++ in Figure 13-3-2(b). The Java/C++ example assumes that bunny is an object instance of a class that defines the *hop* method and *distanceTo* function.

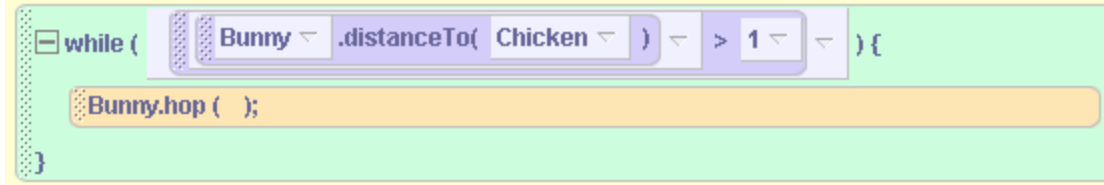


Figure 13-3-3(a). Alice *while* loop construct

```
while (bunny.distanceTo(Chicken) > 1 ) {  
    bunny.hop( );  
}
```

Figure 13-3-3b. Java/C++ *while* loop construct

C++ and Java both have a *do...while* loop. Again, while this construct is not part of Alice, no additional computational power is derived.

Repetition: recursion

A method in Alice has the ability to call itself – known as a *recursive* call. While a recursive call is not actually a control statement as are the *if-else*, *loop*, and *while* statements, recursion is a fundamental means of repetition, common to most languages.

13 Summary

Making a transition from Alice to Java, C++, or C# is a process of making connections between what you have learned in Alice and what you will need to learn in another language. The good news is that Alice has provided to you all the fundamental concepts common to object-oriented languages, hopefully with less frustration and less detail (and hopefully a lot of fun along the way).

To make your transition to real world languages easier, in this chapter we have made use of an Alice **syntax switch** that allows you to switch *on* all details of syntax (such as curly brackets, parentheses, commas, quotes, and semicolons). Once this switch has been turned on, the display of code in the program editor is distinctly “Java-like.” Also, a method signature is automatically included in the code to give a more faithful syntactic flavor. Methods are labeled *void* because they return no value and questions are labeled with the type of value returned. Looking at the lines of program code with the syntax switch turned on makes it easy to see that Alice statements follow very similar rules of syntax as Java and other object-oriented languages.

Classes of objects in Alice are defined by 3D models provided by the Alice system. In real world languages, you will define your own classes using a text editor. In Alice as well as real world languages, an *object* is an instance of a class and *encapsulates* its own properties and methods (is a self-contained unit). In Alice, you can easily see the visual object and its properties. But, in real world languages objects are often harder to visualize and the only way to “see” the properties of an object is to print out the values of these properties to the screen. We looked at how to write methods in Java that can be used to access the values stored in object variables so as to output a textual display of the values that represent an the object.

Program constructs are used to control flow of execution in a program. In Alice, we have looked at if-else constructs for deciding whether or not an instruction (or block of instructions) will be executed. Also, we have looked at counted loops and while loops as looping constructs. These constructs are very similar to those used in object-oriented languages, providing for a quick transition into learning to program in another language.

Important concepts in this chapter

- *Syntax* is a technical term for rules that specify how words and punctuation marks are put together to create a statement in a program.
- A *signature* is a header for a method or question that specifies the name of the method, the parameters that expect to receive information when the method is invoked, and the type of data that will be returned (if any).
- In Alice, a method is not expected to return a value and its signature is labeled *void*.
- A question is expected to return a value and its signature is labeled with the type of data that will be returned.
- Methods and questions are made up of blocks of code containing one or more instructions. In real-world languages, the blocks of code are often enclosed in curly brackets.

Appendix A



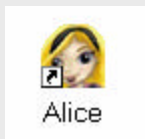
A Hitchhiker's Guide to Alice, Part 1

This tutorial-style guide will provide instruction for learning the basics of using the Alice interface. Based on our experience with many Alice users, we suggest you work through the Hitchhiker's Guide with a friend. You'll have fun together, and you'll be able to help each other work through any parts you might find confusing. If at any point you get lost or stuck, go back to the beginning of the section, reload the world and try again. You can't hurt anything and you will only lose a few minutes of work.

Whenever you see text printed like this, the Hitchhiker's Guide is giving you specific instructions about what to do.

Alice lets you create virtual worlds populated by 3D objects that have behaviors that can change over time. Objects in Alice can move, spin, change color, make sounds, react to the mouse and keyboard, and more.

How to Start Alice:



Alice can be started in one of two ways:

- 1) Click the icon on the desktop of your machine**
Alice may take a minute or two to load. If you cannot find the icon use the windows search utility find Alice.exe

"C:\Program Files\Alice\AliceW.exe"

In some installations, the Alice startup may display a choice dialog box, as illustrated below.

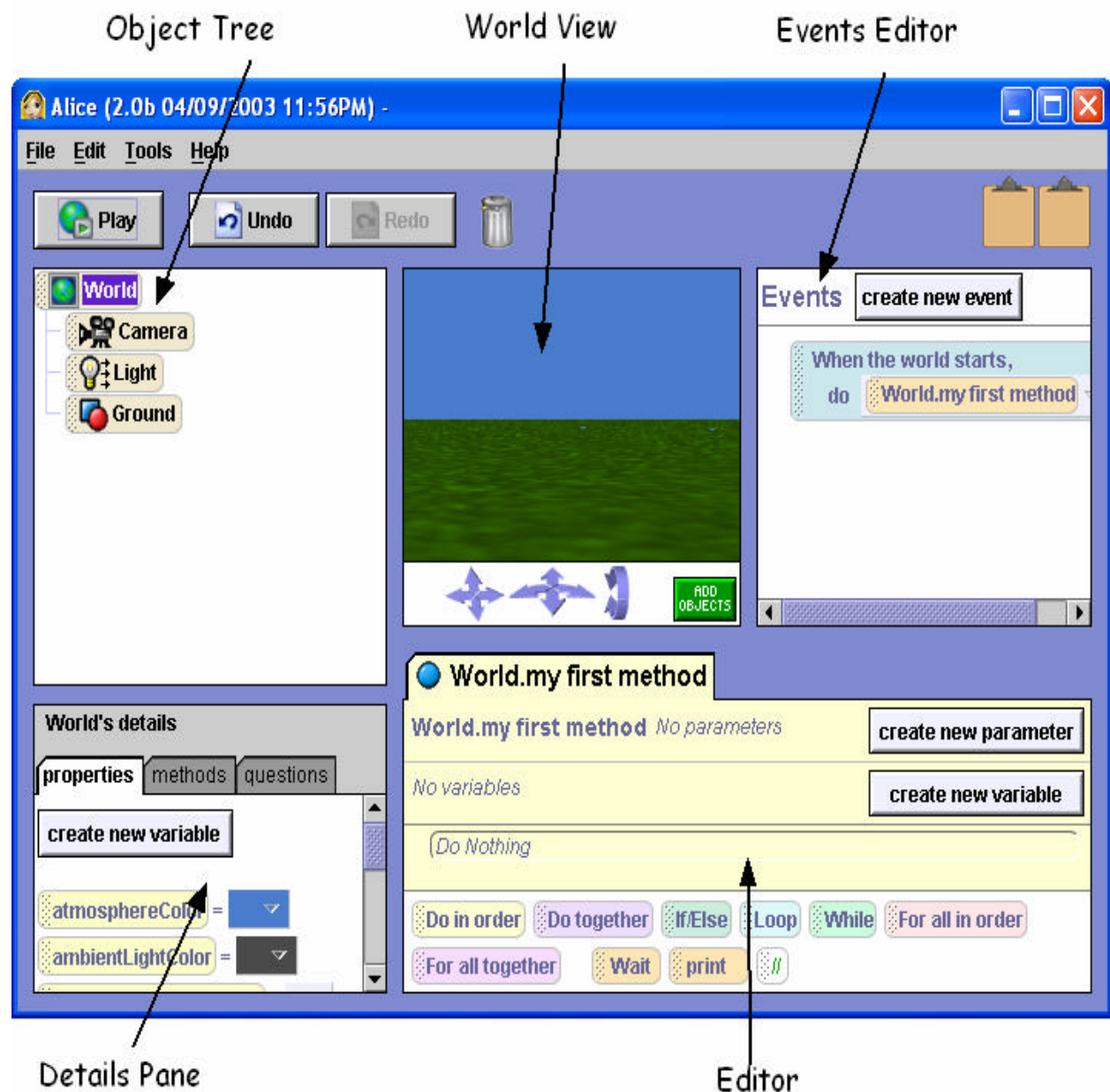


At any time that you decide you no longer wish to see this dialog box each time you start Alice, uncheck the box in the lower left corner of the dialog box, labeled "Show this dialog at start."

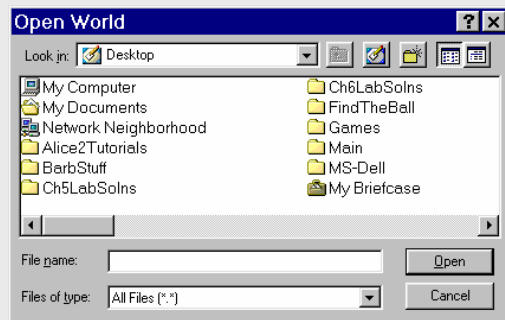
Click the Go to Alice button in the lower right of the choice dialog box.

World 1: Opening and Running Your First World

Alice starts with an empty world. In the World View window, you should see the green grass and a blue sky. If this is not the case, see Appendix (?) on Alice installation. The parts of the interface are labeled in the image below.

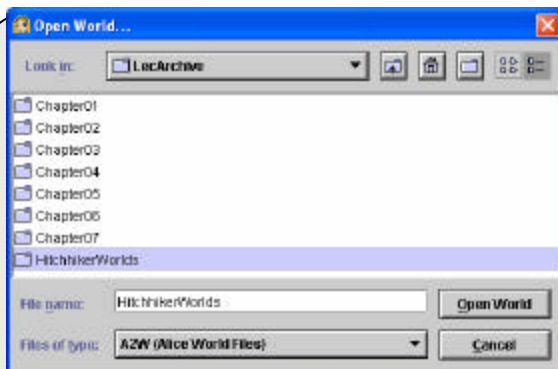


Let's start by opening a world.



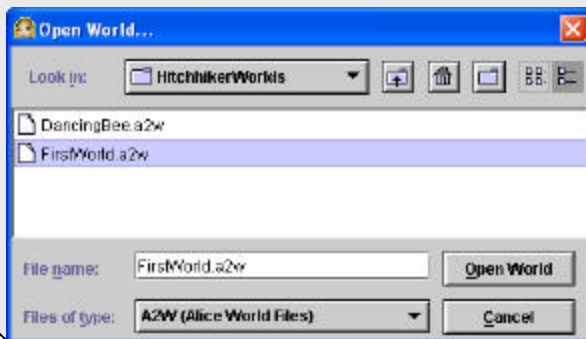
Go to File and choose Open World.

This brings up a dialog box that lets you find the world to be opened.

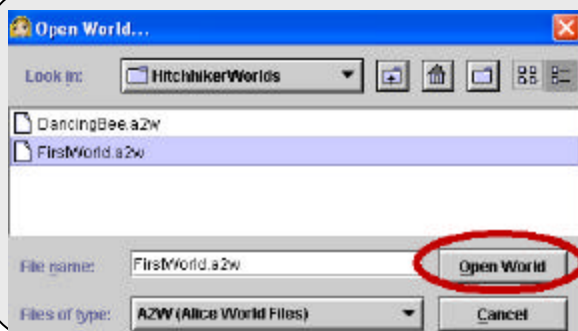


Navigate to the HitchhikerWorlds folder.

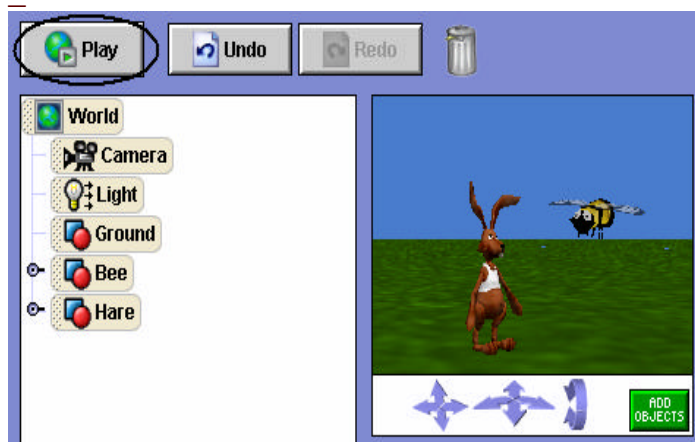
Note: The HitchhikerWorlds folder is available on the CD that accompanies this text.



**Select the file
"FirstWorld.a2w"**

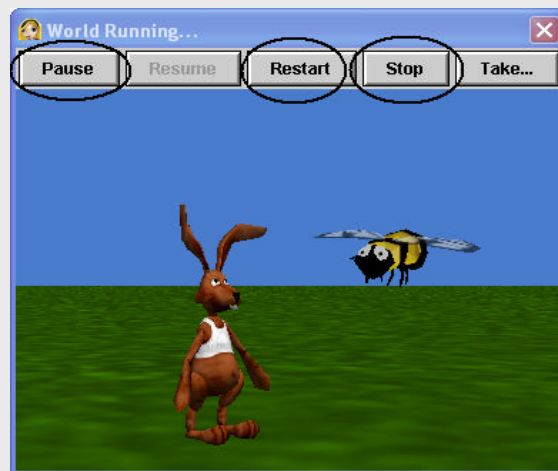


Click the Open World Button.



In this world, you should see a Bee and Hare objects.

Click on the PLAY button to run the world!



Click on the Stop button to exit the window.

Other buttons, such as **Restart** and **Pause** give you control over the window where the animation is running.

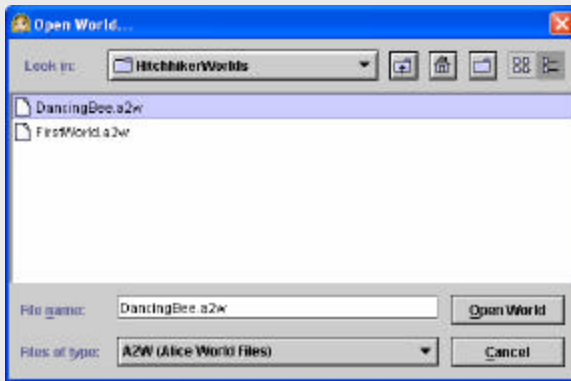
Summary

Here's a recap of what we just covered. If you're not comfortable with any of these topics, go back to the start of this section and go through it again.

- How to **Start** Alice
- How to **Open** a saved world
- How to **Play** a world
- How to **Stop** a world

World 2: Interactive World

World 1 (FirstWorld), viewed above, is a "movie." A movie runs from beginning to end while you, as the human "user", view the animation. Let's look at a world that is interactive where you can make choices as to how the animation works.



Use File | Open World to open the dialog box.

In the dialog box, navigate to the HitchhikerWorlds folder.

Select "DancingBee.a2w" and click on the Open button.



Click the Play button.

Then, try each of the two choices (keys can be pressed in any order):

- 1) Press the down arrow key
- 2) Press the space bar.

Pressing the down arrow key creates an *event*. Alice *responds* to the down-arrow event by executing the pirouette animation. Pressing the Space bar creates a different *event*. Alice *responds* to the Space-bar event by executing a jumpUp animation. This is an example of an interactive, "event-driven" program.

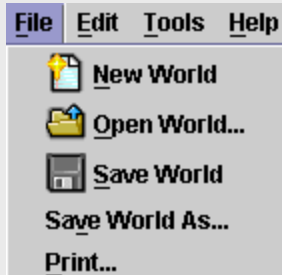
Summary

Here's a recap of what we just covered. If you're not comfortable with any of these topics, go back to the start of this section and go through it again.

- ? running interactive worlds
- ? events and responses

World 3: Creating and saving your own new world

First, let's make a new world.

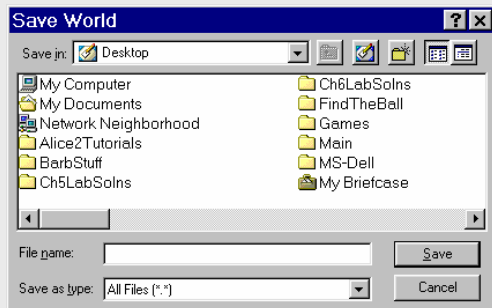


Click on the File menu in the upper left-hand corner of Alice. Select "New World"

Alice opens a new world with some green grass and a blue sky.

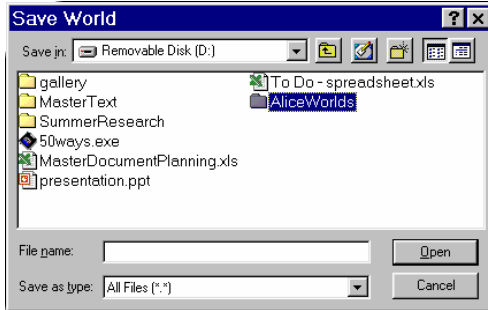
Saving a World

Each time a new world is created, it is a good idea to save the world. Then, if the computer crashes, the world will be safe to reload when the computer is rebooted. A world can be saved to any one of several different locations. For example, you can save the world to the desktop or a folder on the hard drive of your computer. Or, if you have an account on a file server, you can save your world in that space. Of course, a world can also be saved to a disk. If a disk is to be used, we recommend that a zip disk be used rather than a floppy disk (the size of the world may be more than a floppy disk can hold). The example below shows directories on a zip disk, but other storage areas should work just as well.



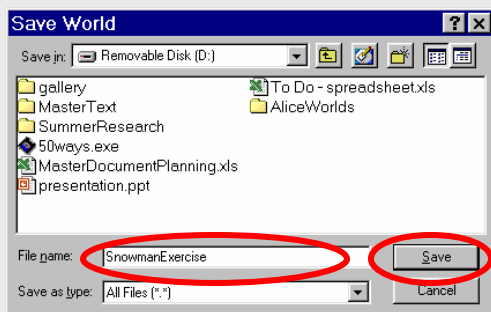
Go to *File* and choose *Save World As*.

This brings up a dialog box that lets you find the location where the world will be saved.



Navigate to the folder (directory) where you plan to save your world.

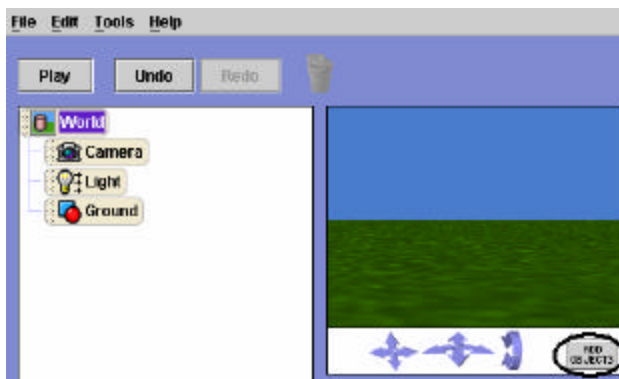
We recommend that you create a folder named *AliceWorlds* where you will save all your animations.



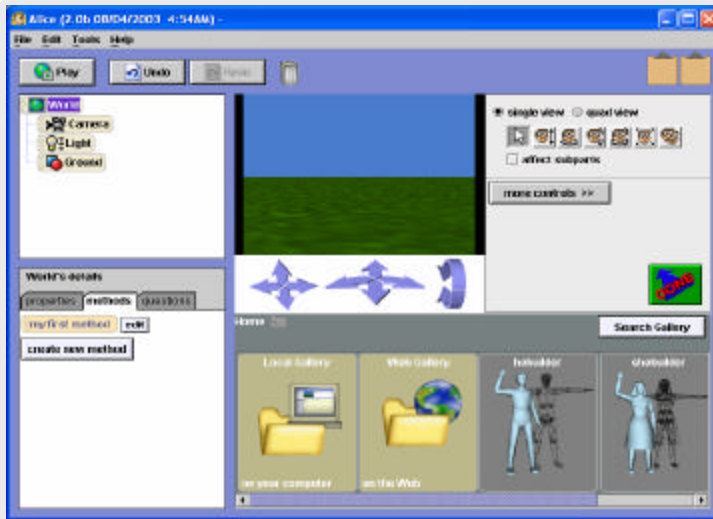
Make up a name for your world -- we recommend a single word name, using upper and lower case characters.

Enter the name for your world and then press the Save button. Your world will be saved with the .a2w extension (An Alice version 2 world)

Adding objects to the world



Click on the “Add Objects” button in the lower right of the World view window.



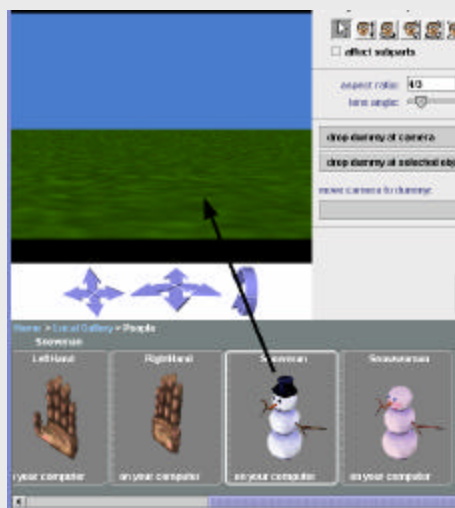
Alice opens the Scene Editor. A visual directory to the Local and Web galleries is provided for access to 3D models. A Search Gallery button allows you to search for a particular kind of object.

Click on the Local Gallery folder.
(See below for use of the Web Gallery.)



Note that the gallery is organized into collections, for example *Animals*, *Buildings*, and *People*.

Click on the *People* folder.



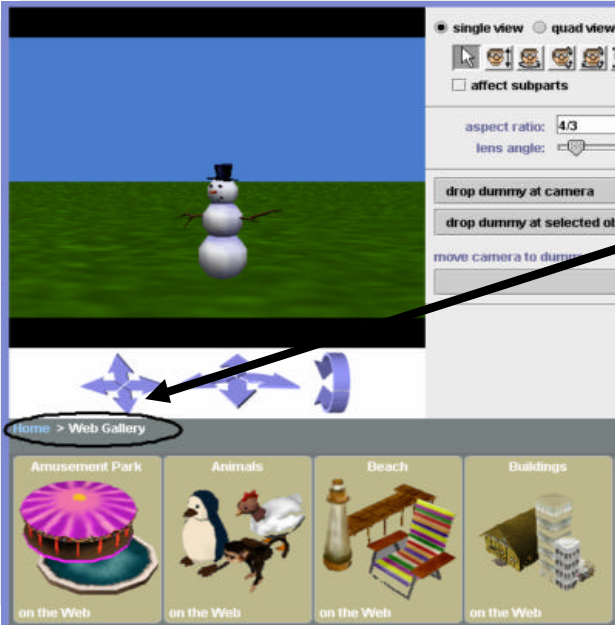
Click-and-drag the Snowman into the scene.

Alice creates a Snowman object in your world.

Note: You can also click on the picture of the 3D model. A small dialog box will appear that asks if you want to add it to the world.

Optional: Using the Web Gallery

If your computer is attached to the Internet, you may wish to use the Web Gallery. This is an index, linked to the online web gallery. The online gallery provides many more models for building worlds. Note: The models in the Web Gallery may take longer than the models on the CD-ROM to load – they are coming from farther away, after all!



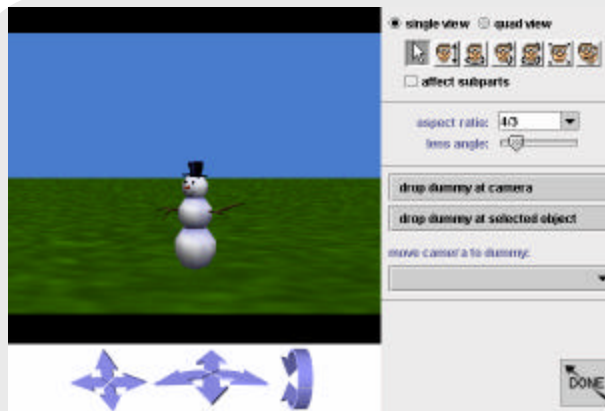
If your computer is attached to the Internet, click on Home. Then, click the Web Gallery from the selection box (shown below).

Then, click and drag in the Snowman from the People folder in the Web Gallery.

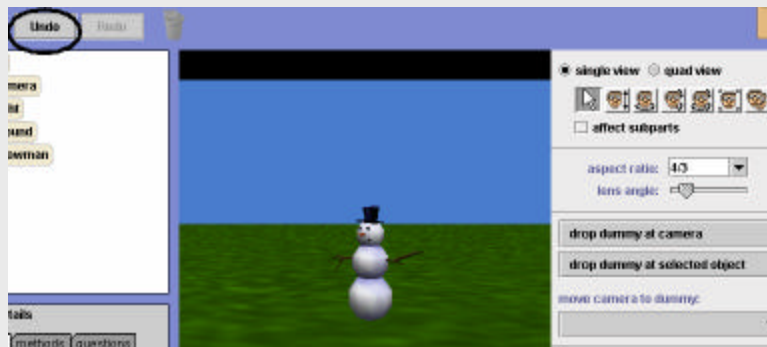


If you tried using the Web Gallery and want to return to the Local Gallery or the CD Gallery, click once again on the Home link (located just above the gallery folder display) and Alice will return to the selection box. From there, you can click on the Gallery folder of choice.

Moving Objects

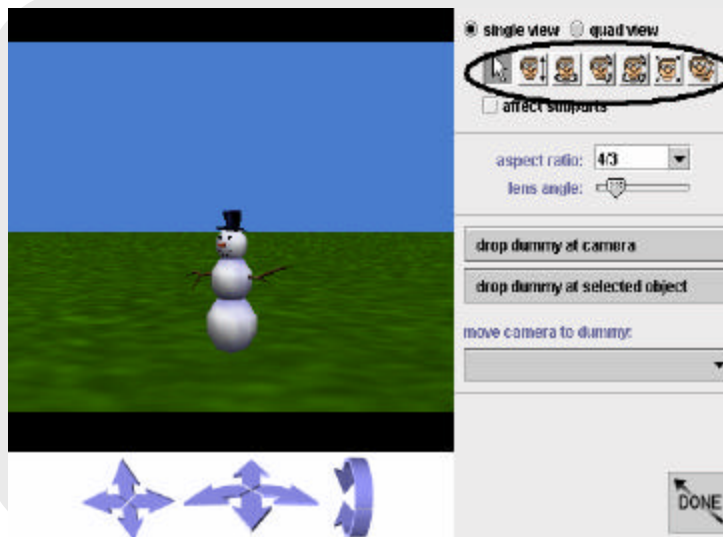


Drag the snowman around the scene with the mouse.



Click the Undo button.

The snowman will return to his previous position.



On the far right of the scene editor, is a row of buttons that select the way the mouse moves an object in 3D space. By default, the horizontal move is selected.

Select each mouse control and experiment with moving the snowman.

Don't worry about messing things up. At any time, you can use Undo.

?

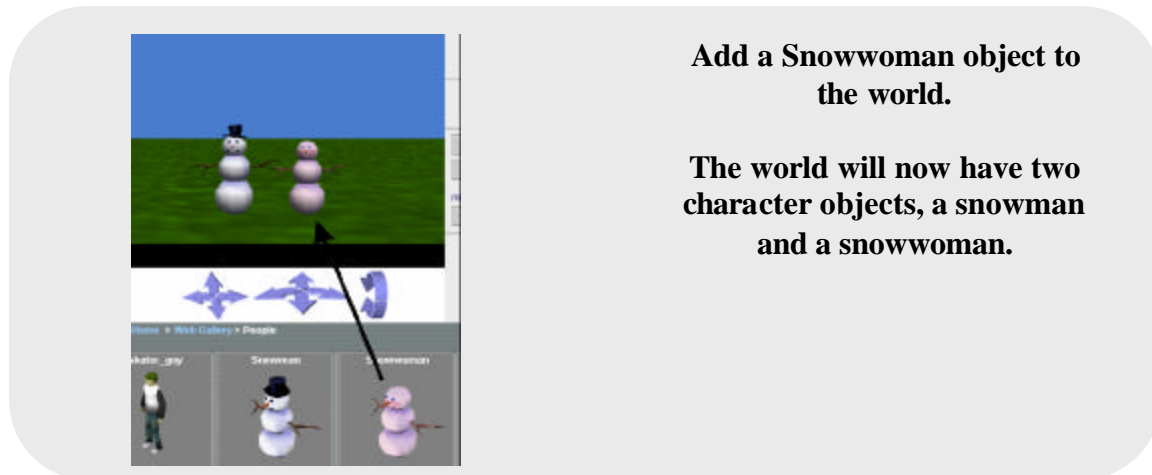
From your experimentation with the mouse controls, you should now be aware that each mouse control button gives a different kind of movement within the 3-dimensional space (forward, back, left, right, up, down) as well as a combination (tumbling). For a more detailed explanation of the movement of objects in three dimensions, read the next section of this chapter (chapter 1 section 2).

Note: The rightmost mouse control button creates a copy of the object (snowman). To remove the copy, use the Undo button. Another way to remove an object from the scene is to right click on the object and choose "Delete" from the pop-up menu.

Now that you have added an object to the world, resave it. We recommend that you periodically save your world.

?

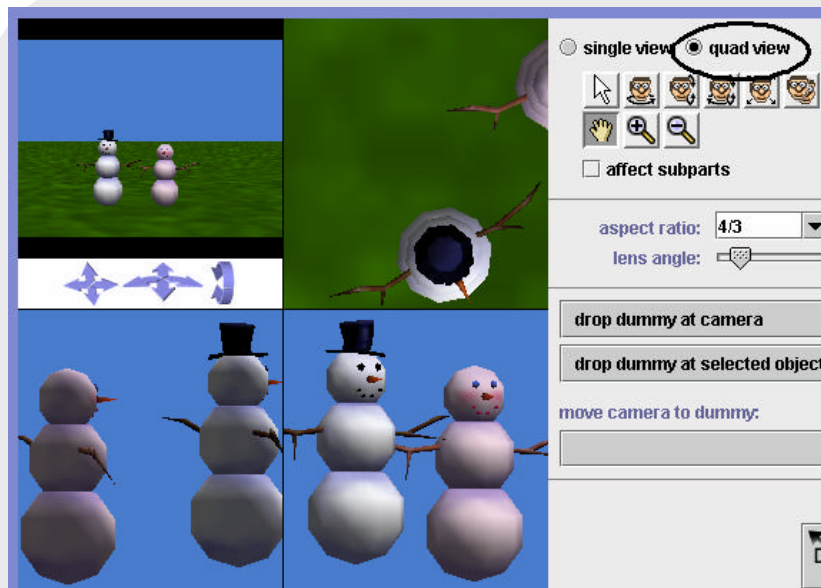
Arranging Multiple Character Objects (the quad view)



Add a Snowwoman object to the world.

The world will now have two character objects, a snowman and a snowwoman.

We would like the snowman and snowwoman to stand side-by-side and face one another. Let's use the scene editor to arrange the two characters.

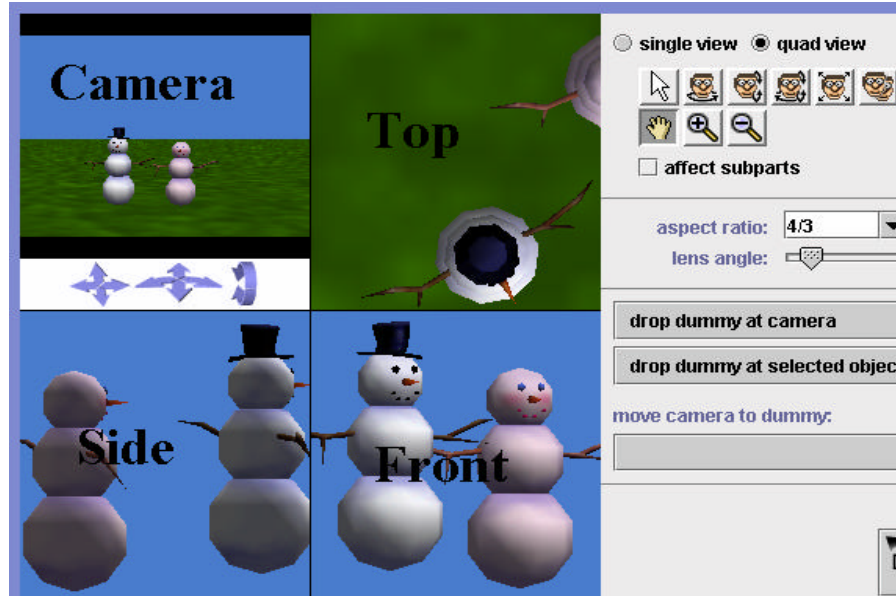


Select "quad view" in the scene editor.

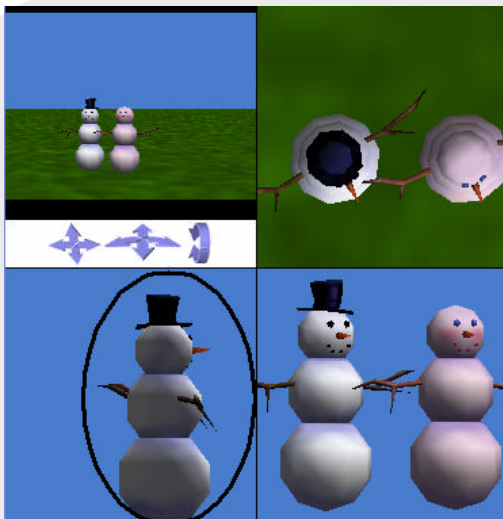
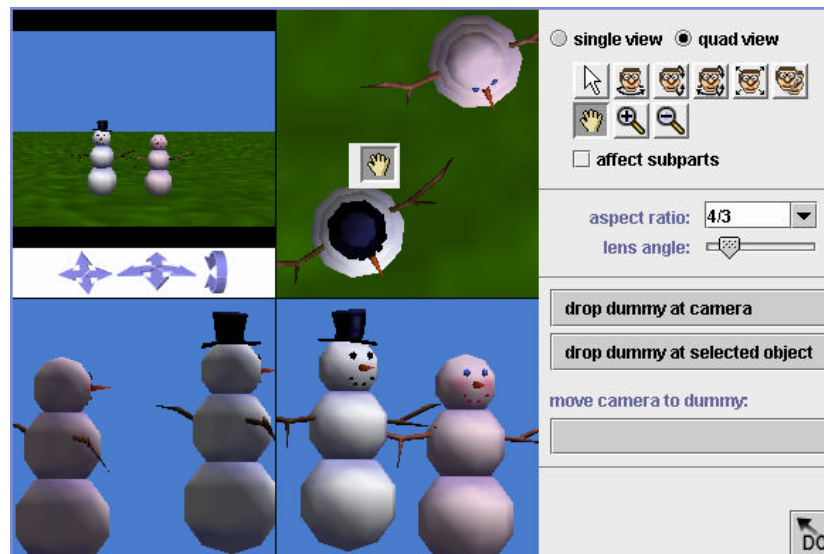
In the quad view, a second row of mouse control buttons appears.

The buttons in the second row allow you to scroll and zoom to change which portion of the world is visible in the window.

The world view window changes to a four-pane **quad view**. The four panes show Camera, Top, Side and Front viewpoints (Labeled in the image below). In each view, the mouse can be used to rearrange the objects.

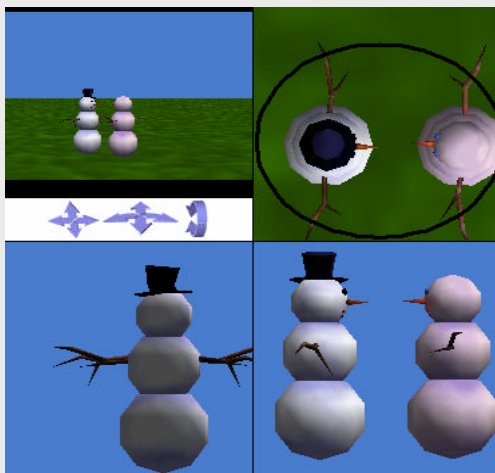


In the screen capture shown above, note that the Top view pane does not show the Snowwoman very well (she is partially out of sight). Alice provides a scroll tool to reposition the viewpoint in a pane. The scroll tool is the button with a human hand pictured on it (in the second row of mouse control buttons). As shown in the image below, we used the scroll control to reposition the Top view pane. At any time, you can use the scroll control to adjust a viewpoint to obtain better layout position.



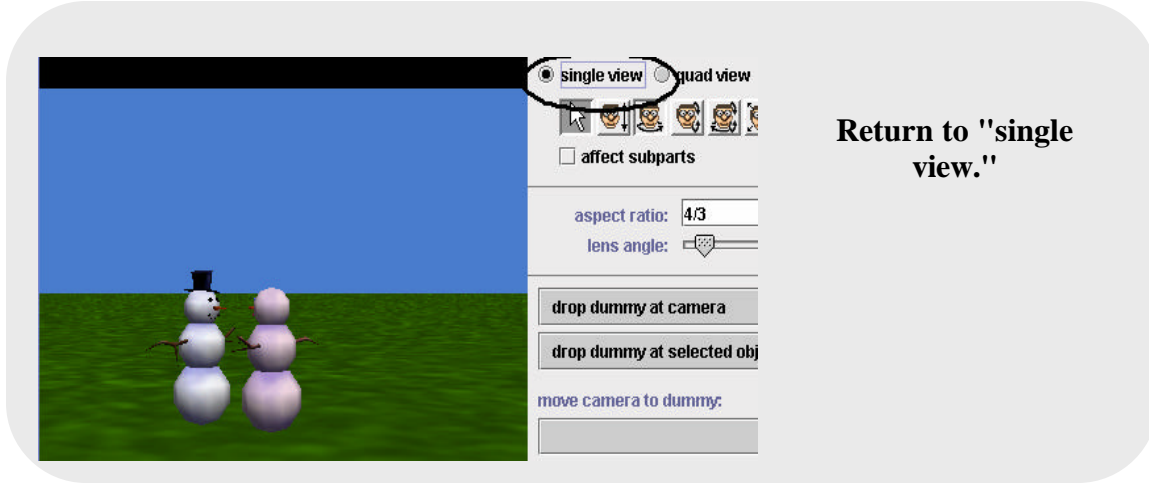
Use the mouse to arrange the two objects side by side.

This position can be recognized when one object (more or less) hides the other in the side view.



Use the mouse to arrange the objects facing one another.

This position is recognized in the top view.



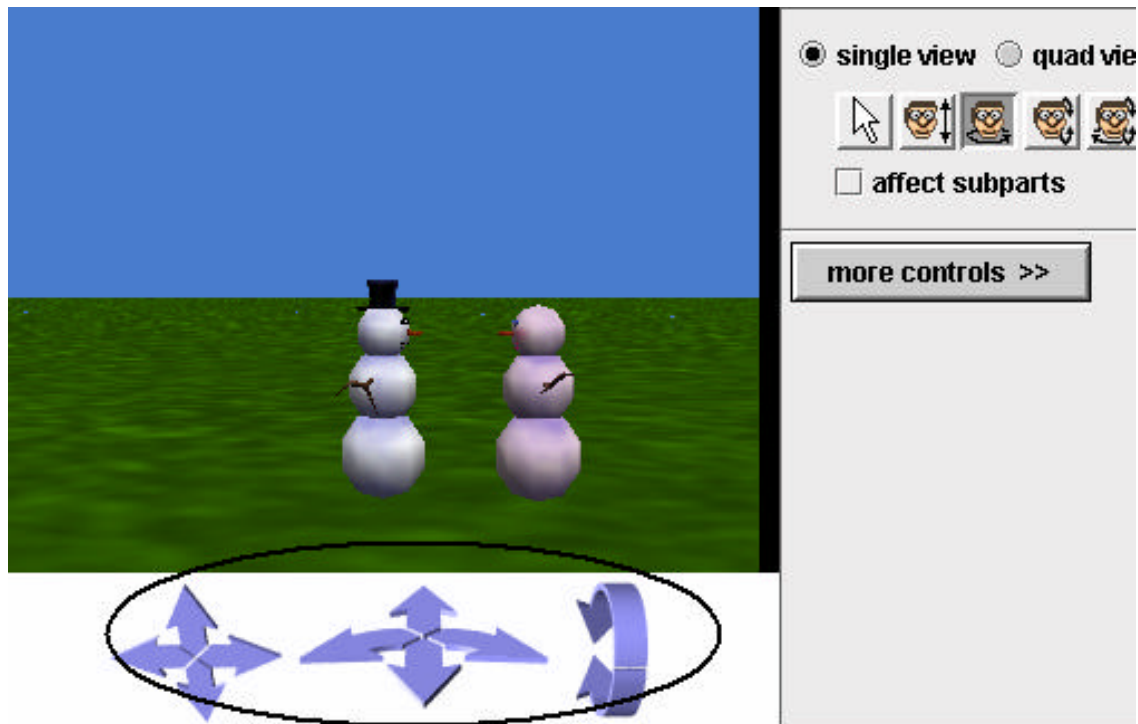
Moving the Camera

In setting up a scene, the camera viewpoint allows us to adjust what the user will be able to see in the animation. It may be helpful to think of the camera as a remote-controlled airborne device that hovers in mid-air over the scene. By moving the Camera, we change our view of the world. The Snowman is a far away, so let's move the Camera to get a better view.



The blue controls on the bottom of the 3D window are called the Camera Controls, as shown below.

Click and drag on the camera controls to get an idea of what each camera control does. Do not be concerned about messing things up. You can always use the Undo button to recover.



Summary

Here's a recap of what we just covered. If you're not comfortable with any of these topics, go back to the beginning of this section and try it again.

- How to make a new world
- Saving a world
- How to add an object to the world
- How to move objects around the world (the scene editor)
- Undo
- Arranging multiple character objects (the layout manager)
- Camera movement
- Avoiding mid-air suspensions




A Hitchhiker's Guide to Alice, Part 2

Using One-shot Instructions to position objects

File Edit Tools Help

 **New World**

 **Open World...**

 **Save World**


Save World As...

Add Object...

Add 3D Text...

To begin this section, first use File|NewWorld to start a new world.

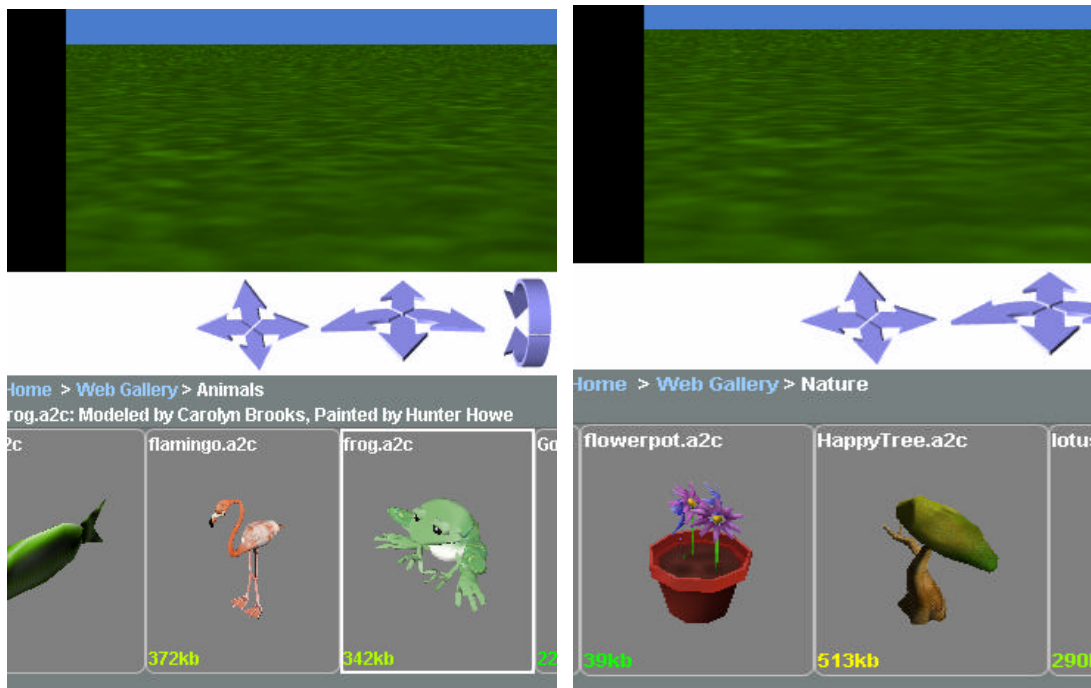


Click on the  button to set up your initial scene.



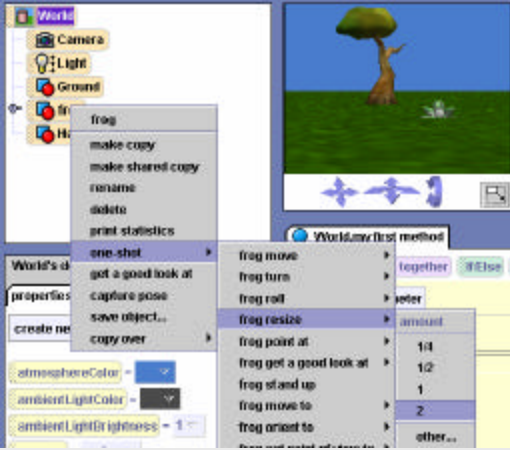
Add a tree and a frog to the new world.

See the illustrations below for selecting the frog and tree from the gallery of objects. (Local or CD galleries provide the same/similar objects.)



Locating the Frog and Tree in the Web Gallery (as seen in the Scene Editor)

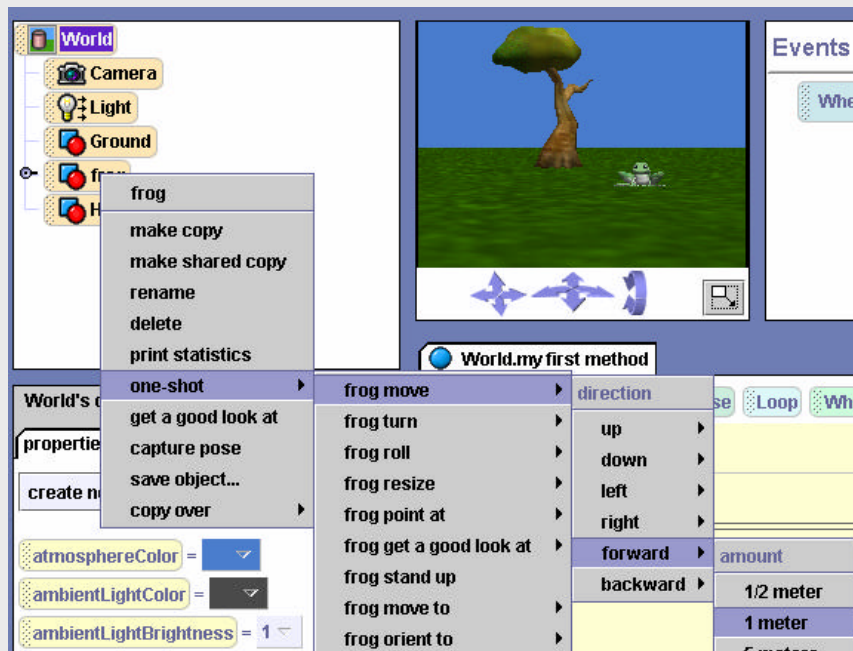
In the scene pictured below, the size of the frog is a bit small. We can use a one-shot *resize* instruction to make the frog larger.



Right-click on "frog" in the object tree.

Use the one-shot instruction named "frog resize" and select 2 as the amount. This will make the frog 2 times as big as its current size.

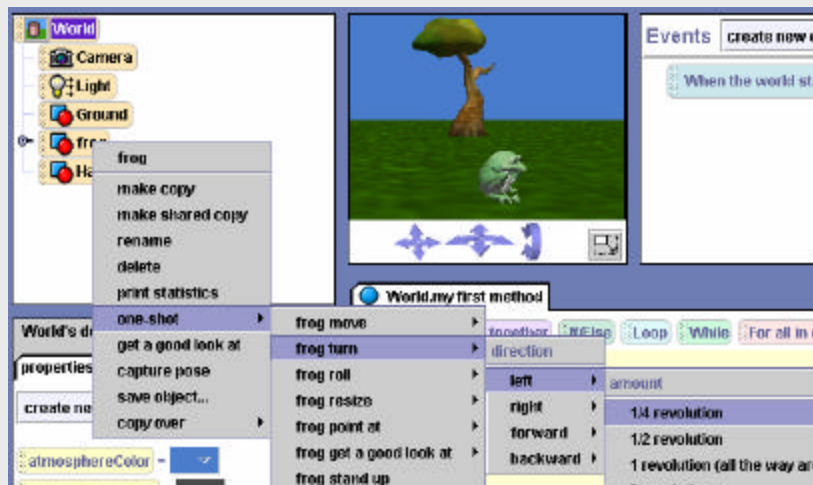
Important Note: Resizing an object may have some unexpected results. For example, suppose you have an object standing on the ground and the object is resized to twice its original size. After the resizing occurs, it is likely that some of the object may be sinking into the ground. This means that the object may need to be repositioned in the world after resizing.



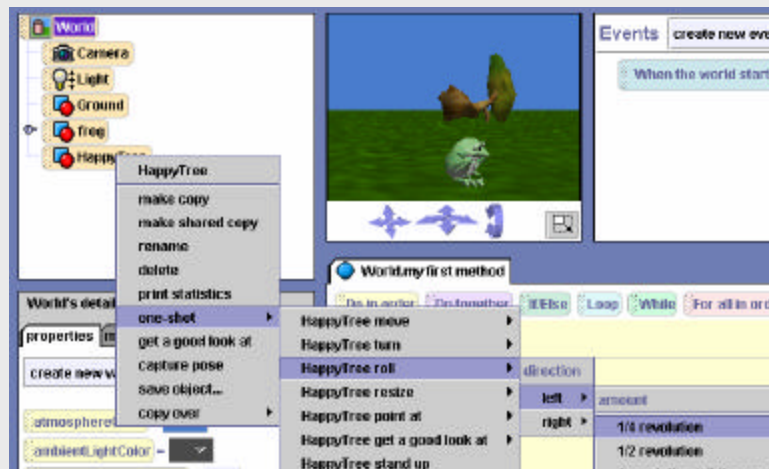
Use a one-shot instruction to move the frog forward 1 meter.
Use a one-shot instruction to move the tree backward 1 meter.
Each object moves relative to its own orientation.

The *move* instruction makes an object move in a given direction, by a given distance. Direction is based on the six degrees of motion available in 3-dimensional space: left, right, up, down, forward, back. (See description in chapter 1 section 2.)

The *turn* instruction makes an object turn in a given direction (forward, back, left, right) by a given amount (in rotations).



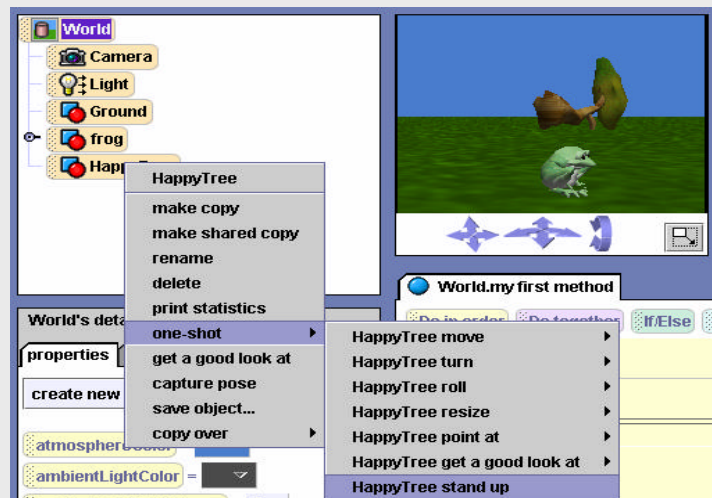
Use a one-shot instruction to turn the frog left 1 / 4 revolution.



Use a one-shot instruction to roll the tree 1 / 4 revolution (choose left or right – either direction is fine).

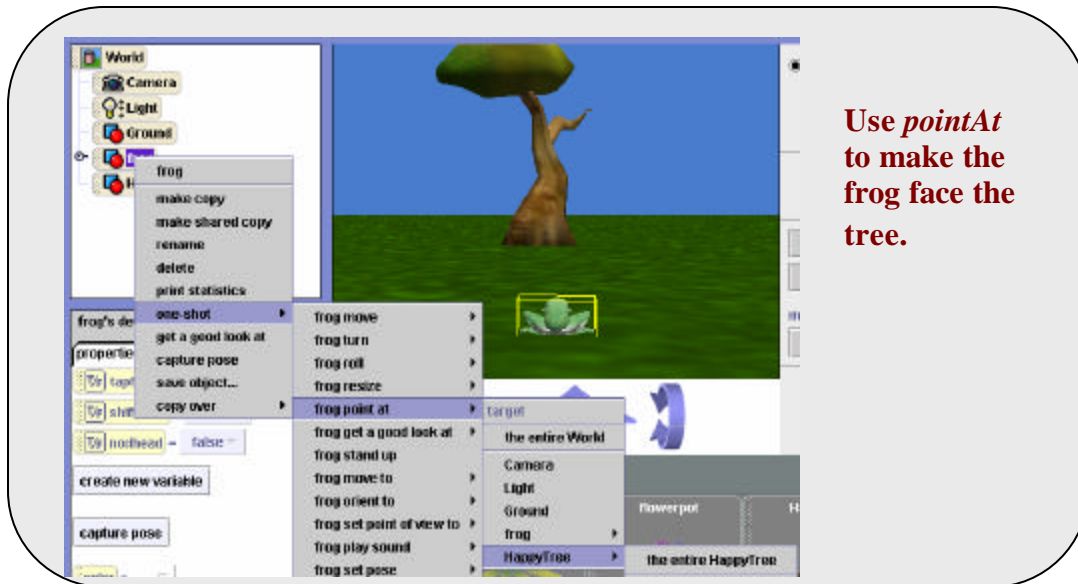
Note that the *roll* instruction causes an object to roll left or right, relative to the object's own sense of orientation.

The *standup* instruction makes an object's vertical axis line up with the vertical axis of the world. In other words, the object stands up!

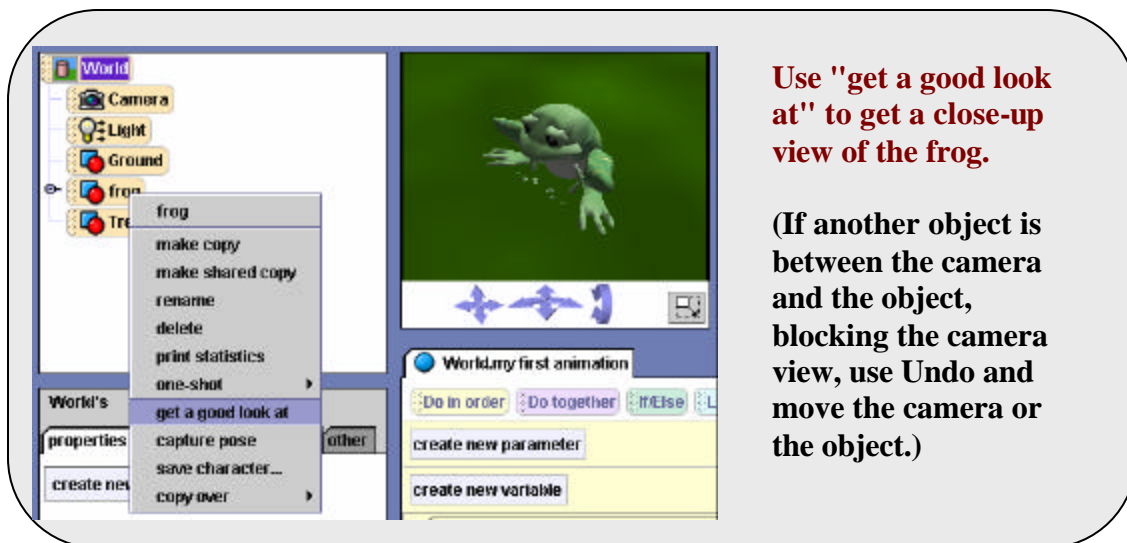


Use a stand-up instruction to put the tree back into an upright position.

The *pointAt* instruction makes an object turn to look toward another object.




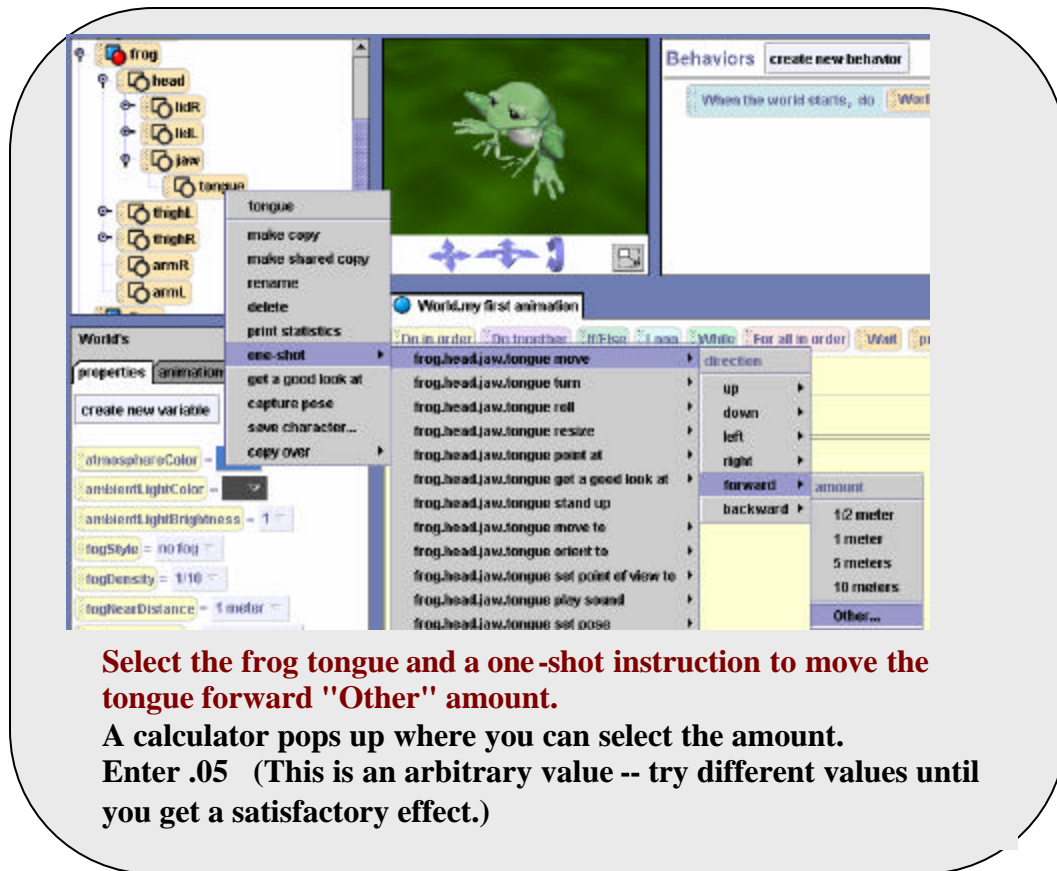
Notice that the HappyTree menu cascades to “the entire HappyTree” at the end. This is to allow you to pick subparts of objects. In the case of the HappyTree, there are no subparts so the menu looks a bit weird (though it is correct.)



Before going on, use Undo to return to the full view of the frog and the tree.

It is possible to move small parts of an object with a one-shot instruction.

✓ Remember: to view a list of the parts of an object, it is first necessary to left click on  to the left of the object in the object tree. One of the frog's parts is its jaw – and one part of the jaw is the tongue. (Parts can have parts, which can have parts, and so on....)

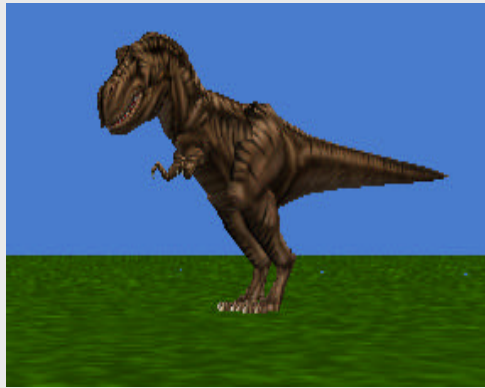


The frog's tongue sticks out of his mouth.



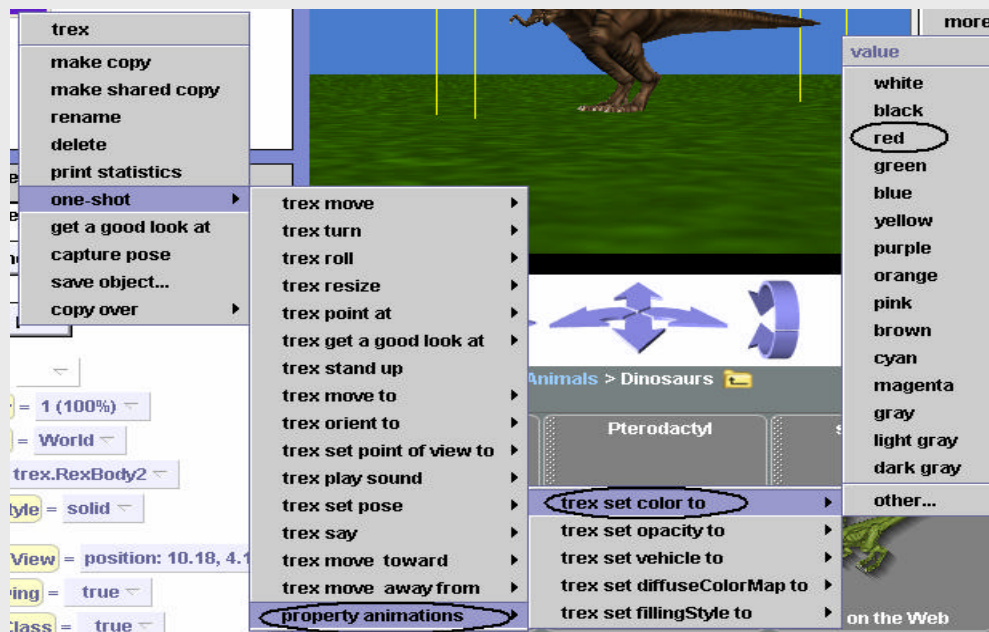
Using One-Shot Property Animations

An option in the one-shot pop-up menu is *property animations*. An example of a property animation instruction is an instruction to set the color of an object, as shown below.



Use File|New to clear the world window for a new world.

Add a trex object to the world.



From the object tree, right-click on trex. From the popup menu, select one-shot → property animations → trex set color to → red. The trex's color should turn red.

Another example is opacity, a useful property for creating ghost-like objects. Opacity refers to the degree of transparency. An opacity of 0% means the object is fully transparent, thus invisible.

