After finishing this chapter, you should be able to:

- Define each of the following terms: computer simulation, concatenate, interactive world, narrative world, string variable. You might also be able to define the following terms related to the subject of the simulation in this chapter: descent speed, horizontal controls, LEM, lunar acceleration due to gravity, main engine thrust
- Describe the difference between interactive and narrative Alice worlds, and the development process for interactive worlds
- Develop an initial scene for an interactive simulation based on given specifications
- Develop programming specifications for necessary events and methods based on the original specifications for an interactive Alice world
- Code the methods and events for an interactive Alice world based on given specifications
DEVELOPMENT OF AN INTERACTIVE ALICE WORLD: A LUNAR LANDING SIMULATOR

Alice worlds typically fall into two general patterns, interactive and narrative, as follows:

- **Interactive worlds**—An interactive Alice world has keyboard or mouse controls that allow the user to manipulate objects in the world while it is playing. Computer games and interactive simulations are both examples of interactive worlds. A flight simulator is a good example of an interactive world.

- **Narrative worlds**—A narrative world tells a story, from beginning to end with little or no user interaction. Many narrative worlds are not purely narrative worlds, but allow for some user interaction, such as the ability to speed up a world, select a chapter from a story, or move the Alice camera while the story is playing. An Alice version of a scene from a Hollywood film would be an example of a narrative world.

It might be safe to say that various Alice worlds have different degrees of interaction, but generally, they fit into these two main categories: interactive worlds that tend to engage the user more and are more action-oriented, and narrative worlds that tend to place more of an emphasis on a fixed story line and the characters within a story.

Throughout the rest of this chapter, you are going to develop an interactive simulation that can be played as a game. A computer simulation is a re-creation on a computer of some situation from the real world that is normally too dangerous or too expensive to physically duplicate. Simulations can be used for entertainment, such as a video game; training, such as virtual reality simulators used to train surgeons; and for research and planning, such as a simulation of a highway system to determine the effects of widening a particular road.

The user of the interactive simulation in this chapter will try to land an Apollo Lunar Excursion Module in a target zone on the surface of the Moon. The Lunar Excursion Module will be called the *LEM*, and the player will be called the *pilot*. The LEM will be pulled downward by the Moon’s gravity and will crash into the surface if it is moving too fast when it lands. The pilot can control a main engine to slow down the LEM, along with horizontal controls to steer the craft as it descends.

You may not have much experience designing and building lunar spacecraft, but such a situation is normal for programmers, who often need to build software with technical requirements outside their area of expertise. The job of programmers is to understand what the software should do, and build software that meets given specifications. Professional programmers often work with experts in the field to develop the specifications, and later these same experts will help review and test the software. Students usually don’t have the opportunity to work with outside experts, but it is still important for you to build your software according to given specifications, and to test the finished software to see that it meets the original specifications.
First you will review the specifications for the Lunar Landing Simulator and the general process for developing interactive worlds, then in Tutorial A you will develop the initial scene for the simulation. Tutorial B is not really a hands-on exercise like the other tutorials in this book, but rather a discussion of the development of programming specifications for the simulator from the original specifications. In Tutorial C, you will actually code the simulation.

This Alice world is based on a real computer simulation. The pilot of the Apollo 11 Lunar Excursion Module, Neil Armstrong, practiced with a computer simulator for months before the first flight to the Moon, as shown in Figure 1. It was a good thing that he did. During the actual landing the autopilot was steering the craft directly toward a field of large boulders, which could have damaged the LEM upon landing. Armstrong took over the controls and used the skills he had practiced to land successfully. A few hours later he and Buzz Aldrin became the first people to walk on the Moon.

The simulation you will develop won’t be as realistic as the NASA simulator Armstrong used, because you need to build your simulator using the tools and objects available in Alice, which are a little different from what NASA had in the 1960s.

Actually, to keep things from getting too complicated, the math in your Lunar Landing Simulator has been slightly simplified, and the simulator will update the velocity of the LEM once every second, so that things like the simulated effect of lunar gravity are approximately correct, but not exactly correct, as would be necessary in a real simulator like the one the Apollo astronauts used. Nevertheless, your simulator should be interesting and challenging to try, as well giving the user something of a feel for what it’s like to be pilot landing on the moon.

**SPECIFICATIONS FOR THE LUNAR LANDING SIMULATOR**

The first step in creating a simulation like this is the development of specifications for the simulation. The developers interview the client requesting the simulation, or do some
research about whatever is being simulated. In this case, we’ll skip that step, and assume you have discovered the following:

- The Moon’s gravity, which is about \( \frac{1}{6} \) that of the Earth’s gravity, is pulling the LEM toward the surface with enough force to accelerate it downward by 1.62 meters per second-squared. This means that the speed of the LEM will increase by 1.62 meters per second (mps) for each second that it falls toward the surface of the Moon. This is much more gentle than the Earth’s gravity, which accelerates an object downward by 9.8 meters per second-squared, but still enough so that an object falling freely toward the surface of the Moon will be going fast enough to crash-land unless it is somehow slowed down.

- The game will start with the LEM stalled (not moving) with the craft 100 meters above the surface. At that point the pilot must take over the controls, but remember, gravity will start accelerating the LEM downward toward the surface.

- There is a rocket engine on the bottom of the LEM, called its main engine, which can be fired to slow its descent or to fly upward away from the surface. The downward speed of the LEM will decrease by 1 mps each time the main engine is fired.

- The LEM also has small rocket engines called thrusters to steer the craft. The pilot can rotate the craft to the left or to the right, and can move the craft backward and forward with these controls.

- If the LEM is going faster than 10 mps when it lands, then the craft will be damaged, even though its legs have special shock absorbers. A speed of 10 mps is approximately 22 miles per hour. That’s about how fast you would be going when you hit the ground if you jumped off the roof of a one-story house. (Do not try this at home.)

- The pilot should be able to see the LEM’s downward speed and altitude and while flying the vehicle.

- A successful landing is in the target zone going less than 10 mps at impact. The target zone is a small circle on the surface, approximately twice the size of the LEM.

- Once the vehicle hits the ground, the simulator should tell the pilot if the landing was a success or a failure, and whether it failed because the LEM was going too fast and crashed, or because it missed the target zone.

- The client expects that someone using the simulation will fail several times before learning to successfully land in the target zone on the Moon. In other words, it should not be too easy to land gently in the target zone.

A few diagrams like those in Figure 2 might help us understand things a little better. These three images were created using Alice itself and Microsoft Visio, as the note with the drawings
describes. With a little practice, you could create similar diagrams in Visio, or even in Microsoft PowerPoint or Microsoft Word.

In a software development project, specifications, including diagrams like these, would be created with the client, or created and then submitted to the client for approval. In this stage of a project, developers are trying to make sure that they understand what the client wants. Because you did not create these diagrams, take a few minutes to look them over to be sure that you understand them.

THE DEVELOPMENT PROCESS FOR AN INTERACTIVE WORLD

The client’s specifications need to be turned into an interactive Alice world. There are really two parts to creating an interactive Alice world: first the initial scene needs to be developed; and second, the methods and events for the world need to be created.

Developing the initial scene involves choosing an Alice Template, deciding what objects will be needed, acquiring or creating those objects, and placing them in the new Alice world. Once the initial scene is ready, you can begin programming the events and methods needed for the new world. The idea of programming should immediately bring to mind the program development cycle discussed in Chapter 2: design, code, test, and debug.
Interactive software, by its nature, tends to be more event-driven than other software, so the design process starts with events, and then methods are designed as handlers for the events. The coding phase works the other way around—the methods are coded first, then the events that use those methods are coded.

Putting all of this together, we get a general development process for an interactive Alice world, as described in the organizational chart in Figure 3. Generally, the tasks at each level need to be completed from left to right across the chart. For example, the initial scene should be developed before the world is programmed. Of course, some of the tasks shown on the chart are simple, such as opening a template for an Alice world, and some of them can be quite complex, such as designing the methods for a world. The details depend on the particular world being developed.

Notice that this chart doesn’t show the software modules you will need, but shows the organization of the tasks needed to complete the project. Such charts are often used for engineering projects, including software engineering. They are somewhat related to more sophisticated charts, known as Gantt charts and PERT charts, which show a more detailed
and more sophisticated organization of the tasks involved in a development project, including the timing of tasks and the resources needed to complete each task. Gantt charts and PERT charts are not often used by novice programmers, so your development process will be based directly on this organizational chart.

The rest of this chapter has three tutorials—creating the initial scene, designing its events and methods, and then coding the methods and events. Creating the initial scene and coding the methods and events are both hands-on activities in Alice. The design of the events and methods should be completed before the coding begins, so that you as the programmer will understand what you are creating and why. After creating the initial scene, you may want to complete reading the discussion of the design in one session and then do the coding in another, because each of these will take some time. You could, for example, read the discussion of the design on the day or evening before you will actually do the coding. It is a common mistake for novice programmers to try to do too much at once, and to be too eager to jump into coding before they fully understand what the code should do. It also might help if you discussed the design with other students before beginning the coding.

**TUTORIAL A—DEVELOPING THE LUNAR LANDING SIMULATOR’S INITIAL SCENE**

For this tutorial, the following list of tasks, which is based on the approach shown in the organizational chart above, will walk you through the creation of the Lunar Landing Simulator’s initial scene. For a project of your own, you would need to develop this list based on the specifications for the project.

The simulator will need to be developed from one of the six existing Alice templates. The space template seems to be best for this project. To create the initial scene, you need to add two objects to the world: a LEM and something to mark the target zone. The objects will each need to be positioned and renamed, and the camera will also need to be positioned.

1. Open a new Alice world with a space template.
2. There is a lunarLander in the Alice SciFi folder (not the Space folder) in the Local Object Gallery. Click the ADD OBJECTS button, and then find the lunarLander in the Alice SciFi folder. Click the LunarLander class tile, and you should see the LunarLander information window, as shown in Figure 4. Click the add instance to world button. It is important to add each object to the simulation this way (rather than by dragging its class tile into the world), so that the LEM and the target zone will in the correct starting positions.
3. Right-click the **lunarLander** tile in Object tree, and rename it **LEM**. The real Lunar Excursion Module was sometimes referred to as the LEM and sometimes as the LM, although both were pronounced like the name “Lem.”

4. The simulation will need to keep track of the LEM's altitude and downward velocity. A new property is needed to keep track of the LEM's downward velocity. You need to create a new LEM variable to store the property. Select the LEM’s **Properties** tab, and then click the **create new variable** button.

5. When the create new variable window appears, type **descentSpeed** as the name and if necessary select **number** as the type. Change the initial value to **0**, and then click the **OK** button. A tile for the new variable will appear near the top of the LEM's properties tab.

The LEM and the camera need to be moved to their starting positions. The specifications call for the simulation to start with the LEM 100 meters above the surface.

1. Run primitive methods to turn the **LEM** to **face the camera**, and then **move** it **up 100 meters**. Remember, you can position an object, turn it, and so on, by right-clicking an object’s tile in the Object tree and then running methods directly. The LEM will move off camera.

2. The camera needs to be positioned. A starting point slightly above the LEM and back from it a little should work. Run the required primitive methods to **move** the camera **up 125 meters**, and then **move** it **backward 25 meters**.

A target landing zone is required. There is a doughnut-shaped object called a torus in the Shapes folder. This is the red shape that can be seen back in Figure 2.

1. Add a torus to the world by clicking its class tile in the **Shapes** folder and then clicking the **add instance to the world** button.
2. The torus is off-camera. Right-click the camera's tile in the Object tree, and then run the point at method to point the camera at the torus.

3. The torus is on the surface, but can hardly be seen. It needs to be larger and a more visible color. Set the color property on the Torus’s properties tab to red. Then, right-click the torus tile in the Object tree and run a method to resize it to be 20 times its initial size. Now it should be more visible on the surface of the moon.

4. The torus needs to be renamed. Right-click the torus tile in the Object tree again and rename it target.

5. The target needs to be positioned. Since you moved the LEM straight upward, the target is now directly under the LEM. It doesn’t matter exactly where it is on the surface, as long as it’s not too far away from its starting point and not directly under the LEM. For now, drag it to any corner of the world window.

6. Now the camera needs to be pointed back at the LEM. To point the camera at the LEM, right-click the camera tile in the Object tree and run the method to make the camera point at the LEM, the entire LEM.

7. The camera needs to be locked onto the LEM so that it will follow it as it descends. To do this, right-click the camera tile in the Object tree, and run the primitive method to set the camera’s vehicle to be the LEM, the entire LEM.

8. Now the world is set up in a good starting position with the necessary objects. Click the DONE button to leave the Scene Editor mode. Figure 5 shows what the world window should look like with the camera and LEM in their starting positions. The target zone is also in place on the surface, but that is currently off-camera.

9. At this point, you should save the world before continuing. Save the new Alice world with the name Lunar Landing Simulator and remember to pay attention to where you save it.

**FIGURE 5:** The world window with the camera and LEM in their starting positions
Next, the development process calls for the design of events, and then methods needed to handle those events. In this project, you will be given a design for both events and methods. In a project of your own, you would need to spend time creating the design before starting to create any code in the new world. The rest of this section discusses the design for the events, while the next section discusses the design of the methods.

**EVENT DESIGN FOR THE LUNAR LANDING SIMULATOR**

The simulator needs an event for lunar gravity that will pull the LEM down toward the ground. It also needs events to provide LEM controls for the pilot, and an event to respond to a successful landing or a crash. The gravity event can call a method to pull the LEM downward while it is above the surface, and then call a method to test the landing once it hits the ground. The following events will be needed:

- A gravity event
- The LEM events

The following sections will discuss each in turn.

**The Gravity Event**

According to the specifications, lunar gravity constantly pulls an object downward, like gravity on earth, but at only 1.6 meters per second squared. This constant, 1.6 mps² is called the *acceleration due to lunar gravity*. Of course, once the LEM hits the ground it will stop moving downward. So, a gravity event needs to move the LEM downward and increase its speed by 1.62 mps for each second until the LEM hits the ground. Once it hits the ground, an event handler can test to see if it crashed.

The gravity event would work like this:

```
While the LEM is above the ground
    Begin: do nothing
    During: LEM fall
    End: test for successful landing
```
The LEM Control Events

The remaining events will provide LEM flight controls for the pilot. These are pretty straightforward and are as follows:

- The spacebar will be used to fire the main engine and provide an upward thrust that will slow the falling LEM by 1 mps. The fire main engine event should work like this:

  When the spacebar is typed, decrease the LEM’s downward speed by 1 mps

- The left and right arrows will turn the LEM left and right. The up and down arrows will move the LEM forward and backward. These are known as horizontal controls.

- The horizontal control events should work like this:

  While the left arrow is pressed, turn to the left
  While the right arrow is pressed, turn to the right
  While the up arrow is pressed, move forward
  While the down arrow is pressed, move backward

It just so happens that Alice has a single built-in event type, shown in Figure 6, that lets the arrow keys move an object as specified in the pseudocode above.

![Figure 6: The built-in Alice event to let the arrow keys move an object](image)

The left and right arrow keys turn the object, while the up and down arrow keys move the object forward and backward—just what’s needed here. So, the horizontal controls can be implemented with just this one event, instead of four separate events.

METHOD DESIGN FOR THE LUNAR LANDING SIMULATOR

The gravity event will need some user-designed methods to serve as its event handlers. One method is needed to drop the LEM, and one method is needed to test for a successful landing. The specifications also call for the simulator to “tell the pilot if the landing was a success or a
failure, and whether it failed because the LEM was going too fast and crashed, or because it missed the target zone.” So, it looks like you will need the following methods for the LEM:

- The fall method
- The landing methods

**The Fall Method**

In each second, the fall method needs to increase the LEM’s downward speed by 1.62 mps and then drop the LEM the appropriate distance. The `descentSpeed` property can be used to keep track of the downward speed of the LEM. The method also needs to tell the pilot the LEM’s downward velocity, which is the value stored in the `descentSpeed` property.

The new velocity after each second is simply the old velocity plus 1.62. During each second, the average speed of the LEM, because it is accelerating constantly, is the average of the velocity at the start of the second and the velocity at the end of the second. Reducing the equation and applying this concept, you would find that a simple calculation for the new altitude is the old altitude minus the starting velocity plus .81.

You are not expected to be able to figure this out on your own, nor is it necessary to do so to create the software. The outside experts or clients working with the software developers on a professional project, such as NASA physicists and engineers working on a project like this, would provide the necessary equations. However, a student who has finished a good high school physics course should be able to figure out the equations needed here or find them in a physics book. This shows us two things, first that good software developers need to be able to work with people who have other expertise to create quality software, and second, if you are interested in things like lunar landing simulators, you should probably take a few courses in physics, higher mathematics, or engineering. College students majoring in computer programming or software engineering often choose elective courses, such as science, engineering, business, graphic arts, or entertainment technology, that will better prepare them to understand the business or profession in which they would like to work as a software professional.

Written in pseudocode, the fall method would look like this:

```
Move LEM down [descentSpeed], with a duration of 1 sec, and an abrupt style
Set new descentSpeed to the old descentSpeed + 1.62 mps
Set new altitude to the old altitude - (descentSpeed + .81)
Display the LEM's new altitude and descentSpeed.
```

The last item in the preceding list of instructions involves creating a message and then displaying the message on the screen. The user will see the term “velocity” instead of
descentSpeed, because the descentSpeed property stores the craft’s downward velocity. We’ll make this a separate event, called display, which would look like this in pseudocode:

```
Build a string to display "velocity = " descentSpeed
Build a string to display "altitude = " altitude
Display a message that combines the descentSpeed and altitude strings
```

Building the strings is a bit tedious, so this will probably be one of the more time-consuming methods to develop for this project, and demands a bit of patience on the part of the programmer, especially if the programmer’s most common experience with a mouse is playing a video game, because every drop and drag interface does not require the quick reactions of a video game.

The Landing Methods

The test landing method will be activated when the LEM hits the ground. It will check to see whether or not the landing was a success, and will call the appropriate method in response to the quality of the landing. Figure 7 shows the logic in the test landing method, and how this method calls the other landing methods.

**FIGURE 7:** The algorithmic logic inside the test landing method
Consider whether the touchdown, crash, and off target methods really need to be separate methods. If they are very short, their code could be placed inside the test landing method, but creating them as separate methods is good modular design. It will make it much easier to expand these methods later, and has all the benefits of modular design discussed in Chapter 2.

For now, the touchdown, crash, and off target methods are each simple. They will each display the appropriate messages: “Touchdown!”, “Crash!”, or “Off target. Safe landing, but out of the target zone.” In addition, the off target method should show the distance from the center of the target zone.

Figure 8 shows the events and methods that need to be created. As indicated earlier, you’ll create the methods first, then create the events that call those methods.

**FIGURE 8:** Events and methods needed for the Lunar Landing Simulator

---

**TUTORIAL C—CODING THE LUNAR LANDING SIMULATOR**

Now that your design seems complete, you can start creating the necessary methods and events. As discussed in Tutorial B, the methods should be created before the events that will use those methods.

**CODING THE SIMULATOR’S METHODS**

The LEM.fall method calls the LEM.display method, so you need to have a LEM.display method before you can use it in the LEM.fall method. It makes sense to create the LEM.display method first. This will be the toughest method to create because of the commands needed to create the messages as text strings from numeric data.

**Coding the LEM.display Method**

The LEM.display method will assemble a text string to display the LEM’s altitude and descentSpeed. You can use the print instruction to display the information in a text window at the bottom of the running world window, as shown in Figure 9. This method needs to build a text string showing the altitude, build a text string showing the descentSpeed, and then display the two together.
1. If the Lunar Landing Simulator world is not open, then open it now.
2. Click the LEM tile in the Object tree and then the methods tab in the details area.
3. Click the create new method button, type the name display in the dialog box that appears, and then click the OK button. The new method will appear in the Editor area.
4. The method needs two string variables to hold the two parts of the message: a descentSpeed string and an altitude string. Start with the descentSpeed string. Click the create new variable button on the right side of the display method in the Editor area.
5. When a Create New Local Variable window appears, enter descentSpeedString as the name and choose other and string, as shown in Figure 10. Click OK.
6. Now you can add the altitude string variable. Click the create new variable button on the right side of the method in the Editor area. When the Create New Local Variable dialog box appears this time, enter altitudeString as the name, choose other and string, and then click OK. You should be able to see tiles for the two new variables in the method, as shown in Figure 11.

The method now has the variables it needs, but three instructions are needed. The first assembles the descentSpeedString, the second assembles the altitudeString, and the third displays them together. These instructions will need to concatenate strings. To concatenate two strings means to join them together to form one longer string. For example, the strings “Rumple” and “stiltskin” can be concatenated to form “Rumplestiltskin”. Alice has a string function of the form \([a \text{ joined with } b]\) on the world’s functions tab that can do this.

You will also need to convert a numeric value to a string. Alice also has a string function on the world’s functions tab to do this. Both of these string functions are shown in Figure 12.
You will use these in creating this method. This is one of the more detailed parts of this tutorial, so take your time and be careful. If you run into trouble, back up and try again.

1. There is a descentSpeedString variable tile near the top of the method. Drag and drop a copy of it in place of do nothing further down in the method. When the value menu appears, choose other and then type “velocity = ” as the value. Be sure to include the blank spaces before and after the equal sign, but do not type the quotes. Click the OK button. The instruction should look Figure 13.

2. Now you need to add the concatenation function to the instruction. Click the world tile in the Object tree, then the functions tab, and drag and drop a joined with b into the descentSpeed set value to [velocity = ] tile in place of [velocity = ]. Choose default string as the value for b. The instruction should now look like Figure 14.

3. Next you need to put a string with descentSpeed in place of the default string. However, descentSpeed is a number variable that needs to be converted to a string, so start by dragging a copy of the world-level function [what] as a string into the method in place of [the default string].

4. When the menu appears, select expressions and then LEM.descentSpeed as the value. Your instruction should now look like Figure 15.
5. Finally, it would be good to round off the value of \textit{descentSpeed}; otherwise, the computer will display too many digits on the screen. Find the \texttt{round} function on the world’s functions tab and drag and drop it into the instruction in place of \texttt{[LEM.descentSpeed]}. Choose \texttt{expressions} and \texttt{LEM.descentSpeed} as the value to be rounded. Your completed expression should now look like Figure 16.

6. If your new instruction matches Figure 16, then this would be a good time to save the method with the same name as before. If your new instruction does not match Figure 16, then you can delete it and try the set of steps again before proceeding.

The method also needs an instruction to build the altitude string. The easiest way to create this will be to copy the instruction you just created, and modify it, as shown in Figure 17. You will create this instruction in the following step sequence.

1. Right-click the \texttt{descentSpeedString set value} tile and select \texttt{make copy} from the short menu that appears. Make sure that you click the tile and not one of the parameters in the tile. A copy of the instruction will appear just below the original.

2. Now you can modify the parameters in the copy to turn it into an \texttt{altitudeString set value} instruction. Click the first parameter, \texttt{descentSpeedString}, and select \texttt{LEM.display.altitudeString} from the menu that appears. This will replace \texttt{descentSpeedString} with \texttt{altitudeString}.
3. Next, click the second parameter, \texttt{[velocity = ]}, select \texttt{other} from the menu that appears, and then enter “ \texttt{altitude = }”. Be sure to type two blank spaces before the word altitude, as well as blank spaces before and after the equals sign.

4. The last parameter will be a little harder to change because there is no variable for altitude like there is for \texttt{descentSpeed}. You need to put the \textit{LEM distance above ground} function in place of \texttt{descentSpeed} inside the \texttt{round} function. Click the \texttt{LEM} tile in the Object tree, and then the \texttt{functions} tab in the details area.

5. Find the \textit{LEM distance above} function and drag and drop it in place of \texttt{LEM.descentSpeed} inside the \texttt{round} function. Choose \texttt{ground} from the object menu that appears. Your instruction should now look like Figure 17.

The last instruction needed for this method is the instruction to display the two strings together. You will use the print instruction to do this.

1. Drag and drop a copy of the \texttt{print} instruction tile from the bottom of the Editor area to a spot just below the last instruction in the method. When the menu appears, select \texttt{text string …}, and then type \texttt{XXX}. This is just a placeholder until the next step.

2. Next, you need to put a copy of the \texttt{[a] joined with [b]} string function in the print instruction tile in place of \texttt{XXX}. Click the \texttt{world} tile in the Object tree, and then the \texttt{functions} tab in the details area. Find the \texttt{[a] joined with [b]} function and drag and drop a copy of it into the print instruction in place of \texttt{XXX}. When the menu appears, choose \texttt{altitudeString} as the \texttt{b} parameter.

3. The instruction still has \texttt{XXX} as the first parameter for the join function. Drag a copy of the \texttt{descentSpeedString} tile and drop it in place of \texttt{XXX}. The print instruction and the method are now complete. The method should look like Figure 18, and the print instruction should look like the last instruction in the method.
Coding the LEM.fall Method

Now that the LEM.display method exists, you can use it to create the LEM.fall method. Based on the design that has been established in this chapter, the LEM.fall method should work like this:

Move LEM down \([\text{descentSpeed}]\), duration = 1 sec, style = abruptly
Set new \(\text{descentSpeed}\) to \(\text{descentSpeed} + 1.62\)
LEM.display

1. Click the LEM tile in the Object tree and then the methods tab in the details area.
2. Click the create new method button, type the name fall in the dialog box that appears, and then click the OK button. The new method will appear in the Editor area.
3. You need to add the instruction to move the LEM downward. Drag and drop the LEM move tile into the new method in place of do nothing. For the direction, select down; for the amount, select expressions, and then LEM.descentSpeed.
4. To complete coding the instruction, click more and make sure the duration is 1 second, then click more again, and make sure the style is abruptly.
5. Next, you need to add the instruction to increase the descentSpeed. Click the properties tab in the details area, then drag and drop the descentSpeed tile into the LEM.fall method below the first instruction. A set value menu will appear. Select expressions, and then LEM.descentSpeed itself as the value.
6. The instruction now says LEM.DescentSpeed set value to LEM.DescentSpeed, but it should be LEM.descentSpeed set value to LEM.descentSpeed + 1.62. Click the second occurrence of LEM.descentSpeed. Select math from the menu that appears, then LEM.descentSpeed +, then other, enter the value
1.62, and then click Okay. Figure 19 shows what this sequence of menu selections should look like. When you are done, the instruction should say \( \text{LEM.descentSpeed set value to LEM.descentSpeed + 1.62} \).

7. Finally, you can add the display instruction to the method. Click the LEM tile in the Object tree, and then the methods tab in the details area. Drag and drop a copy of the LEM.display method tile into the fall method below the other instructions. Your method should now look like Figure 20.

**FIGURE 19:** The series of menus used to set the instruction

**FIGURE 20:** The LEM's fall method

**Coding the Landing Methods**

The last methods that need to be coded are the landing methods. The LEM.testlanding method will call each of the other three methods and will contain logic like that shown in the flowchart back in Figure 7. For now, the LEM.touchdown, LEM.crash, and LEM.offTarget
methods will just display a message and some information. These methods will be easier to create than the string handling method you just finished. Because the LEM.testlanding method uses the other three, it makes sense to create the LEM.touchdown, LEM.crash, and LEM.offTarget methods first.

1. Click the LEM tile in the Object tree and the methods tab in the details area.
2. Click the create new method button, type the name touchdown in the dialog box that appears, and then click the OK button. The new method will appear in the Editor area.
3. Drag a copy of the print instruction tile from the bottom of the Editor area and drop it into the new method in place of do nothing. Select text string ... from the short menu that appears, and enter Touchdown!
4. Drag another copy of the print instruction tile from the bottom of the Editor area and drop it into the new method below the [print Touchdown!] tile. Again, select text string ... from the short menu that appears, and this time enter Congratulations! Successful landing in the target zone, and then click Okay.

The LEM.touchdown method is now complete. Now you can create the LEM.crash method.

1. Click the LEM tile in the Object tree and the methods tab in the details area.
2. Click the create new method button, type the name crash in the dialog box that appears, and then click the OK button. The new method will appear in the Editor area.
3. Drag a copy of the print instruction tile from the bottom of the Editor area and drop it into the new method in place of do nothing. Select text string ... from the short menu that appears, enter Crash!, and then click Okay.

That’s all this method will do for now. Once the simulation works, you might want to come back to this method to add some action to the crash landing. Next, you can create the LEM.offTarget method.

1. Click the LEM tile in the Object tree and the methods tab in the details area.
2. Click the create new method button, type the name offTarget in the dialog box that appears, and then click the OK button. The new method will appear in the Editor area.
3. Drag a copy of the print instruction tile from the bottom of the Editor area and drop it into the new method in place of do nothing. Select text string ... from the short menu that appears, enter Off target!, and then click Okay.
4. The method needs an instruction to tell the pilot how far off target. This will be done with two separate print instructions. First, right-click the [print Off target!] instruction tile and select make a copy. Then click [Off Target!] in the bottom copy and change the text string to Distance to center of target zone:

5. Now make a copy of the print [Distance to center of target zone:] tile. This will be modified to print the actual distance, which is a number that needs to be converted into a string. Click the world tile in the object tree, and then the functions tab in the editor area. Drag a copy of the [what] as a string function and drop it in the bottom print instruction tile in the LEM.offTarget method in place of [Distance to center of target zone:]. Select target from the menu that appears.

6. Now you can modify this instruction to display the distance from the LEM to the target. Click the LEM tile in the Object tree and then the functions tab in the details area. Drag a copy of the LEM distance to function tile and drop it in the print [target] as a string instruction tile in place of the parameter target. Select target from the menu that appears, and the instruction is complete.

The last method that needs to be created is the LEM.testLanding method, which will include the logic to determine which message should be displayed when the LEM hits the ground. The logic, shown back in Figure 7, calls for a nested If/Else structure.

1. Click the LEM tile in the Object tree and the methods tab in the details area.
2. Click the create new method button, type the name testLanding in the dialog box that appears, and then click the OK button. The new method will appear in the Editor area.
3. Drag a copy of the IF/Else tile from the bottom of the Editor area and drop it in the method in place of do nothing. Select true from the menu that appears.
4. Click the LEM tile in the Object tree and then the properties tab. Drag and drop a copy of the LEM’s descentSpeed property into the IF/Else command in place of true. When the menu appears, select descentSpeed <, then other, enter the number 10, and then click Okay.
5. Next, click the methods tab to see the LEM’s methods. Drag a copy of the crash method tile and drop it into the If/Else instruction tile in place of Do Nothing below Else.
6. Now drag a second copy of the If/Else tile from the bottom of the Editor area and drop it the existing If/Else instruction in place of Do Nothing
between *If* and *Else*. Select **true** from the menu that appears. This will create a set of nested if commands like those in Figure 21.

7. You need to create the condition for the inner if command. The LEM needs to be within 3 meters of the center of the target zone. Click the **functions** tab in the details area and then drag a copy of the **LEM is within threshold of object** function tile into the instruction in place of *true*. When the menu appears, select **1 meter** as the *threshold* and **target** as the *object*.

8. You must replace **1 meter** with 3 meters. Click the **[1 meter]** parameter and change it to **3 meters**. The LEM will be off target if it is more than 3 meters away from the center of the target zone.

9. Now click the **methods** tab, and drag and drop a copy of the LEM’s **touchdown** method into the inner *If/Else* tile in place of *Do Nothing* between *If* and *Else*.

10. Finally, drag and drop a copy of the LEM’s **offTarget** method into the inner *If/Else* tile in place of *Do Nothing* below *Else*. The method is now complete, and should look like Figure 21.

**FIGURE 21:** The LEM’s `testLanding` method

CODING THE SIMULATOR’S EVENTS

Now that all of the required methods are in place, we can create the events for which they will be event handlers. According to the revised specifications, as summarized back in Figure 8, the following three methods are needed:
- The gravity event, as follows:

  ```
  While LEM is above the ground
  Begin: Nothing
  During: LEM.fall
  End: LEM.testLanding
  ```

- The fire main engine event, as follows:

  ```
  When the spacebar is typed, descentSpeed = descentSpeed - 1
  ```

- The horizontal controls event, as follows:

  ```
  Let the arrow keys move the LEM
  ```

To create the gravity event:

1. Click the **create new event** button in the Events area, and then select **While something is true** from the menu that appears. You should see a new blank method similar to Figure 22.

![FIGURE 22: A blank While something is true method](image)

2. You need to replace `<None>` with the condition **LEM is above the ground**. Click the **LEM** tile in the Object tree and then the **functions** tab. Scroll down through the functions and drag and drop the **LEM is above** function to replace the word `<None>`. Select **ground** from the menu that appears.

3. Next, you need to put copies of three different method tiles in the new event following **Begin:**, **During:**, and **End:**. Select the **LEM** tile in the **object tree** and the **methods** tab in the **details area**. Drag a copy of the **fall** method tile from the details area and drop it after **During:**.

4. Finally, drag a copy of the **testLanding** method and drop it after **End:**. Your new event is complete and should look like Figure 23.
Next, you need to create the fire main engine event.

1. Click the **create new event** button in the Events area, and select **When a key is typed** from the menu that appears.
2. Click the **any key** parameter in the new event tile and select **Space** from the menu that appears.
3. Select the **LEM** tile in the **object tree** and the **properties** tab in the **details area**. Drag a copy of the **descentSpeed** tile and drop it in the new event in place of **Nothing**. When the menu appears, select **decrement LEM.descentSpeed by 1**. This will perform the same instruction as \( \text{descentSpeed} = \text{descentSpeed} - 1 \).

The event should now look like Figure 24.

Now the horizontal controls event needs to be created.

1. Click the **create new event** button in the Events area, then select **Let the arrow keys move <subject>** from the menu that appears.
2. Click the **<camera>** parameter in the new event tile and select **LEM**, then the entire **LEM**, from the menu that appears.

The code for the Lunar Landing Simulator is now complete. Make sure to save your world and remember where you save it and what name you use. You may want to save a backup copy in another location. It is estimated that the finished world should be between 1 and 2 Mb in length.

**TESTING THE SIMULATOR**

A programmer who finishes coding the software is like an author who has finished writing a first draft of a document. It’s now time to test the software and find out if any modifications are necessary. The primary purpose of testing is to make sure that the software functions according to its specifications, but you also need to watch for unexpected problems, such as the side effects mentioned in Chapter 2, and for awkward things that could be slightly improved, such as a camera angle in a particular scene, or the way in which a particular button works.
Figure 25 shows the software components contained in the Lunar Landing Simulator. There are three events and four methods that need to be tested. You saw in Chapter 2 that this involves unit tests, in which each software module is tested by itself to see if it performs its function properly, and integration tests to see if a module works when included in a software package with other modules.

In a professional software development project, the client with outside expertise would help develop a testing plan to see if the software is functioning in a technically correct manner. For example, the client would determine if the gravity event was properly simulating lunar gravity. Further, engineers would check the math in the simulation software and time the activities of the simulation to see that everything is working correctly. Of course, each of the controls, such as the spacebar to fire the main engine, and the arrow keys to fire the horizontal control thrusters would be checked to see if they are working properly.

To simulate such a testing environment, you will try the simulator to see if it works.

1. Play the world and first look to see if the velocity is increasing and the altitude is decreasing while the LEM falls. When the LEM hits the ground, the Crash! message should be displayed.

2. Next, play the world again and try to slow the LEM’s descent by using the spacebar to fire the main engine. Do not use the arrow keys. See if you can land with a velocity of less than 10 mps. You might need to try several times. If you thrust the engine too much, the LEM will start to move upward, with a negative descent velocity. When you land softly enough, the Off target! message should be displayed rather than the Crash! method.

3. Finally, play the world again and try to steer the LEM and move it forward and backward while it is falling by using the arrow keys. The camera’s position is locked to the LEM, so as you move and turn, the LEM will appear stationary and the world and target area will appear to move. If you can use the arrows to control the horizontal thrusters so that you land in the target zone, and use the main engine so that you are going less than 10 mps when you hit the ground, then you will get the Congratulations! Successful landing . . . message.
CHAPTER SUMMARY

This chapter centered on a single long project that involved building an interactive Lunar Landing Simulator as an Alice world. It included a discussion of the simulator’s specifications, along with a general discussion about the process of building interactive worlds, followed by three tutorials.

The first part of the chapter included the following:

- Alice worlds typically fall into two general patterns—interactive and narrative.
- An interactive Alice world has keyboard or mouse controls that allow the user to manipulate objects in the world while it is playing. Computer games and interactive simulations are both examples of interactive worlds.
- A narrative world tells a story, from beginning to end with little or no user interaction. Many narrative worlds are not purely narrative worlds, but allow for some user interaction, such as the ability to speed up a world, select a chapter from a story, or move the Alice camera while the story is playing. An Alice version of a scene from a Hollywood film would be an example of a narrative world.
- Programmers often need to build software with technical requirements outside their area of expertise. The job of programmers is to understand what the software should do, and build software that meets specifications. They work with experts in the field to develop the specifications, and later these same experts will help review and test the software.
- The first step in creating a simulation like the Lunar Landing Simulator is the development of specifications for the simulation.
- Drawing a few diagrams often helps programmers understand specifications. With a little practice, you could create simple diagrams in Microsoft PowerPoint or Word by using the items on the Drawing toolbar. Microsoft Visio can also be helpful.
- There are two parts to creating an interactive Alice world: first, the initial scene needs to be developed; and second, the methods and events for the world need to be created.
- Interactive software tends to be event-driven, so the design process starts with events, and then methods are designed as handlers for the events. The coding phase works the other way around—the methods are coded first, then the events that use those methods are coded.

In Tutorial A, you developed the initial Scene for the Alice world.

Tutorial B, which was not a hands-on tutorial, discussed converting the given specifications into programming specifications for the events and methods needed for the simulator.

You coded the methods and events for the new world in Tutorial C. Part of this tutorial included instructions to handle string variables. Testing the simulator was left as an exercise for the reader.
REVIEW QUESTIONS

1. Define the following terms: computer simulation, concatenate, interactive world, narrative world, and string variable.

2. This chapter contained terms related to the subject of the simulation. Define each of the following: descent speed, horizontal controls, LEM, lunar acceleration due to gravity, and main engine thrust.

3. Describe the difference between interactive and narrative Alice worlds.

4. List and describe the steps in the development process for interactive worlds.

5. Which should be designed first for an interactive world, events or methods? Why? Which should be coded first, and why?

6. Simple text strings often need to be concatenated when composing messages for the screen in interactive software. Give three examples of how individual strings of text on a standardized form, such as a job application, can be concatenated to form useful longer strings. For example, on a student course registration form, the simple strings for the department name and the course number, “Computer Science”, and “101” can be concatenated to print “Computer Science 101”.

7. Alice has built-in events to let the arrow keys control the camera and to let the mouse control the camera. Describe code for your own Alice events to provide the following camera controls without using the built-in camera events:
   a. To pan the camera left and right
   b. To tilt the camera up and down
   c. To zoom in and zoom out

8. Tell which of the nine event types you would use for each of the following:
   a. In a narrative world, a windmill’s blades must turn continuously.
   b. A character should walk across the screen when a world begins playing.
   c. The user must click an ice skater to make her start her routine.
   d. In an interactive world that is an imitation of Donkey Kong, the user must make a Mario-like character jump.
   e. A seaplane will crash when it hits the water.

9. Write a short introduction to the Lunar Landing Simulator contained in this chapter that describes it to a new user, explains how it works, and contains some hints on successfully landing in the target zone.

EXERCISES

1. The Objects folder in the Alice local gallery has an American flag as a class of objects called Flagpole. Modify the touchdown method for the lunar landing simulator so that an American flag pops up if the pilot lands the LEM successfully. There are several different ways to do this, but one approach involves manipulating an object’s opacity property so that it is always present, but invisible until its opacity is changed.
2. The Lunar Landing Simulator resulting from the tutorials in this chapter is still very primitive. 
   Describe:
   a. Several modifications to the existing methods in the world to make it more interesting.
   b. Additional events or methods that could be added to the world to make it more interesting.
   Which of these modifications are you able to implement?

3. Working as part of a team with at least one other student, develop a narrative Alice world that is an 
advertisement for a product or service of your choice. Look through the Alice object galleries for 
ideas. Try to use a development process similar to the development process for interactive worlds 
shown in Figure 3.

4. Alice has a method to allow an object to play a sound from a sound file. The sound files must be 
imported or recorded using the button on an object's property tab. NASA's Kennedy Space Center 
has archives of recordings from the Apollo project on the Web at: http://science.ksc.nasa.gov/ 
history/apollo/apollo-11/sounds/.
   a. Add a few appropriate sounds to your Lunar Landing Simulator from the NASA archive.
   b. Record a few sounds of your own and add those to the world. This requires a computer with a microphone 
      attached.

5. The people folder in the Alice local gallery has hebuilder and shebuilder classes that will let you 
create your own characters. These characters come with methods to walk, stand, wave hello, nod 
yes or no, and show several emotions. Experiment with the hebuilder and shebuilder, and then:
   a. Build a character of your own using the hebuilder or shebuilder.
   b. Create a simple Alice world to allow the user to experiment with the character’s built-in methods by pressing 
      various keys on the keyboard.

6. Simple things, like making a character walk, involve a bit of work in Alice, yet certain Alice objects, 
such as the Penguin, come with built-in walk methods. Try each of the following:
   a. Create an Alice world with a penguin and at least one other object. Create a method to make the penguin 
      walk over to the object.
   b. Add several additional objects to the world. Create the methods and events needed to make the penguin 
      walk to whatever object the user clicks with the mouse. Can you do this with one generic “walk to” method?
   (Note: The same thing can be done with Cow class of objects, which also has a built-in walk method.)

7. The local Alice gallery’s Animals folder has a tortoise and a hare in it. Assume that you want to 
develop an Alice world to tell the story of the tortoise and the hare based on Aesop's fables. If you are 
not familiar with the story, see http://www.eastoftheweb.com/short-stories/UBooks/TorHar.shtml.
Develop the following for a Tortoise and Hare world:
   a. A description of the initial scene for the world.
   b. A list of the events and methods needed to implement the world.
   c. Programming specifications for each method and event.

Remember the importance of modularity and reusability, exhibited in such things as methods to 
make the tortoise walk or run.
8. Create specifications for an initial scene and programming specifications for the necessary events and methods for a interactive world that will allow the user to parallel park a car in a parking space between two other cars or trucks. The vehicles folder in the local object gallery contains several cars and trucks. The City gallery contains objects that could make your world seem a little more realistic.

9. Develop specifications for a helicopter game with horizontal controls and a control to move the helicopter up and down. The Alice Vehicles folder contains a helicopter.

10. Exercises 7, 8, and 9 involve developing specifications for Alice worlds. Working from finished specifications, build one of the three worlds.

11. Extra Challenge: Alice doesn’t have a built-in feature for collision detection; in other words, there is no way to directly tell if one object bumps into another object. Can you develop specifications for a scheme for collision detection in Alice?