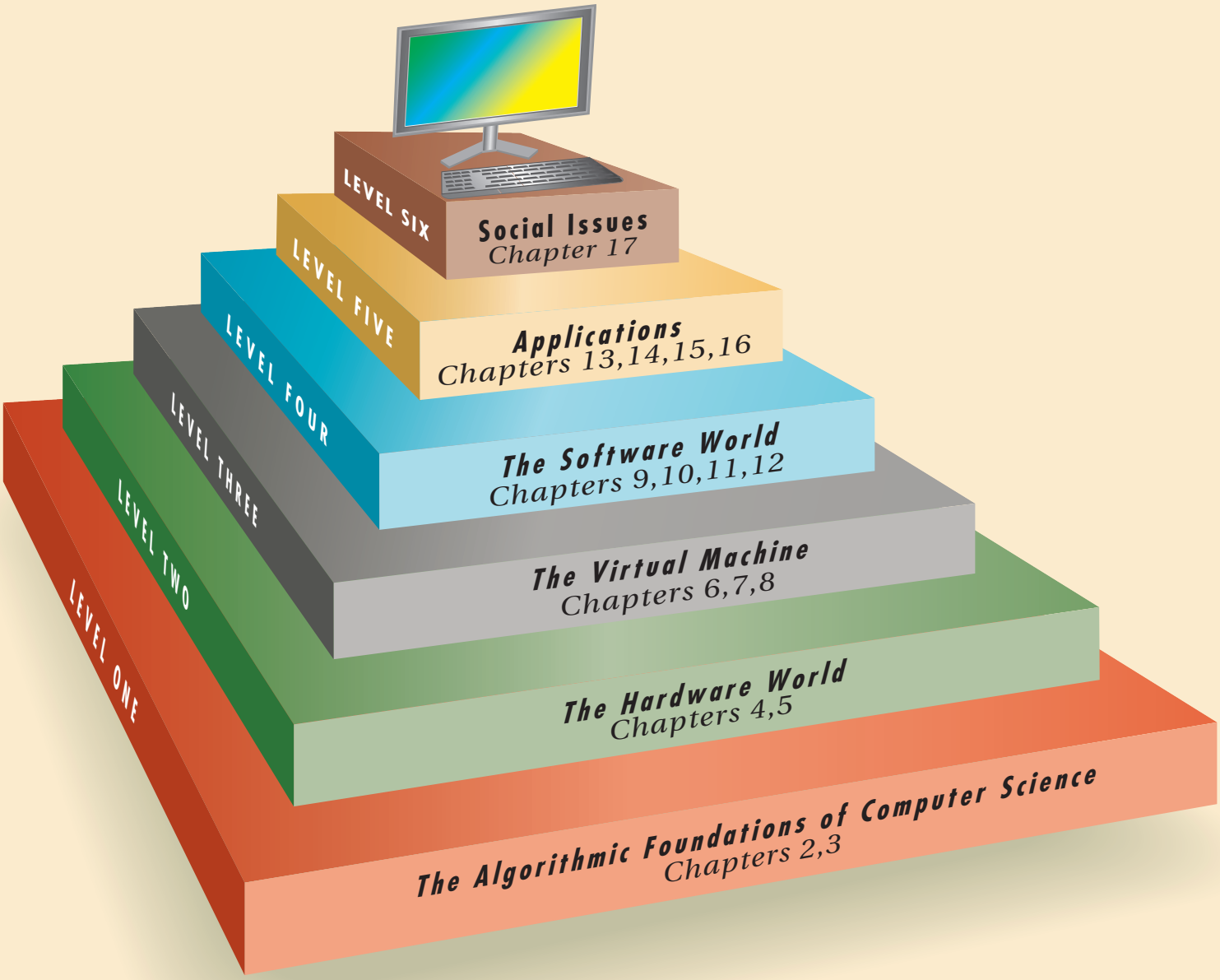

FIFTH EDITION
INVITATION TO
COMPUTER SCIENCE
G. Michael Schneider • Judith L. Gersting

5TH EDITION

Invitation to Computer Science


LEVEL SIX
Social Issues
Chapter 17
LEVEL FIVE
Applications
Chapters 13,14,15,16
LEVEL FOUR
The Software World
Chapters 9,10,11,12
LEVEL THREE
The Virtual Machine
Chapters 6,7,8
LEVEL TWO
The Hardware World
Chapters 4,5
LEVEL ONE
The Algorithmic Foundations of Computer Science
Chapters 2,3

5TH EDITION

Invitation to Computer Science

G. Michael Schneider
Macalester College

Judith L. Gersting
University of Hawaii, Hilo

Contributing author:
Keith Miller
University of Illinois, Springfield

COURSE TECHNOLOGY
CENGAGE Learning

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

Invitation to Computer Science, Fifth Edition
G. Michael Schneider and Judith L. Gersting

Executive Editor: Marie Lee

Acquisitions Editor: Amy Jollymore

Senior Product Manager: Alyssa Pratt

Development Editor: Deb Kaufmann

Editorial Assistant: Julia Leroux-Lindsey

Marketing Manager: Bryant Chrzan

Content Project Manager: Jennifer K. Feltri

Art Director: Faith Brosnan

Cover Designer: RHDG/Tim Heraldo

Cover Artwork: Fotolia.com (Royalty Free), Image # 375162

Compositor: Integra

ISBN-13: 978-0-324-78859-4
ISBN-10: 0-324-78859-2

Course Technology
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:
international.cengage.com/region

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **course.cengage.com**
Visit our corporate website at **cengage.com**.

Printed in Canada
1 2 3 4 5 6 7 13 12 11 10 09

BRIEF CONTENTS

CHAPTER 13

Simulation and Modeling

13.1 Introduction

The computational devices of the nineteenth and early twentieth centuries were used to solve important mathematical and scientific problems of the day. We saw this in the historical review of computing in Chapter 1: Charles Babbage's Difference Engine evaluated polynomial functions; Herman Hollerith's punched card machines carried out a statistical analysis of the 1890 census; ENIAC computed artillery ballistic tables; and Alan Turing's Colossus cranked away at Bletchly Park, breaking the "unbreakable" German Enigma code. The users of these early computing devices were primarily mathematicians, physicists, and engineers.

Today, there is hardly a field of study or aspect of our society—from art to zoology, business to entertainment—that has not been profoundly changed by information technology. Now we use computers in many "nonscientific" ways, such as playing games (a topic we will investigate in Chapter 16), surfing the Web, listening to music, and sending e-mail.

However, the physical, mathematical, engineering, and economic sciences are still some of the largest users of computing and information technology. In this chapter we investigate perhaps the single most important scientific use of computing—computational modeling. This application is having a major impact on a number of quantitative fields, including chemistry, biology, medicine, meteorology, ecology, geography, and economics.

13.2 Computational Modeling

13.2.1 *Introduction to Systems and Models*

The **scientific method** entails observing the behavior of a system and formulating a hypothesis that tries to explain its behavior. We then design and carry out experiments to either prove or disprove the validity of that hypothesis. This is the fundamental way to obtain new scientific knowledge and understanding.

Scientists often work with a model of a system rather than experimenting on the "real thing." A **model**, as defined in Section 12.2, is an abstraction of the system being studied, which we claim behaves much like the original. If that claim is true, then we can experiment on the model and use these results to understand the behavior of the actual system. For example, physical models (small-scale replicas) have been in use for many years, and we are all familiar with the idea of testing a model airplane in a wind tunnel to understand how the full-sized aircraft would behave.

In this chapter we are not interested in physical models but in **computational models**, also called **simulation models**. In a computer simulation, a physical system is modeled as a set of mathematical equations and/or algorithmic procedures that capture the fundamental characteristics and behavior of a system. This model is then translated into one of the high-level languages of Chapters 9 and 10 and executed on the Von Neumann computer described in Chapters 4 and 5.

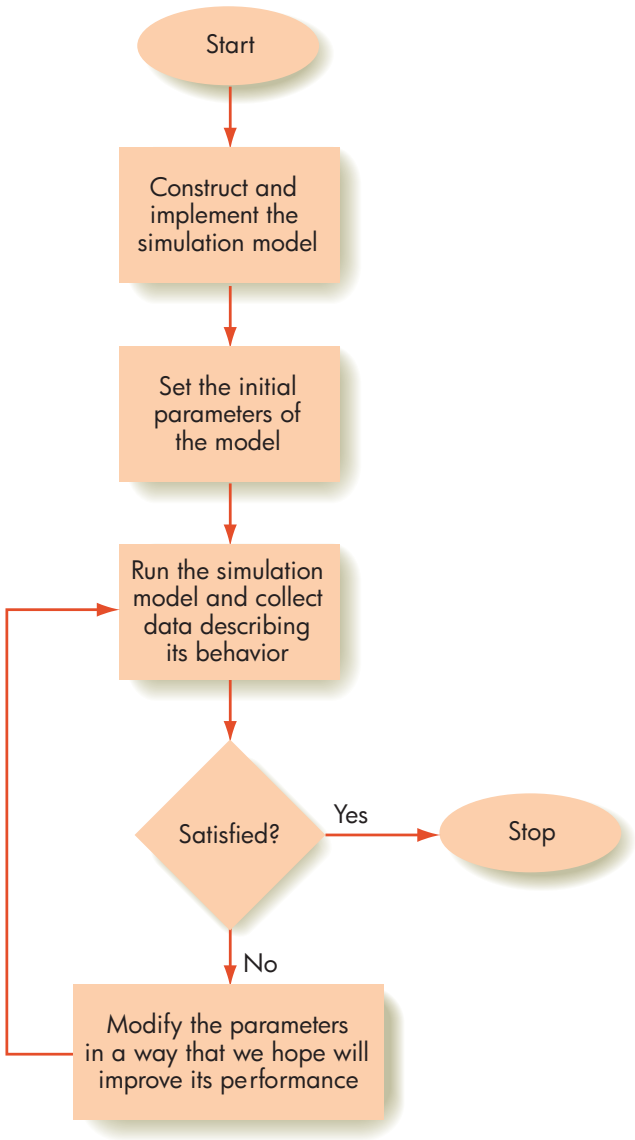
Why construct a simulation model? Why not study the system itself, or a physical replica of the system? There are many reasons:

- *Existence*. The system may not exist; therefore, it is not possible to experiment directly on the actual system.
- *Physical realization*. The system is not constructed from entities that can be represented by physical objects. For example, it may be a social system (e.g., welfare policies, labor practices) that can only be simulated on a computer.
- *Safety*. It may be dangerous to experiment on the actual system or a physical replica. For example, you would not want to try out a totally new welfare policy that could economically devastate a population or build a nuclear reactor using a new and unproven technology.
- *Speed of construction*. It may take too much time to construct a physical model. Sometimes it is faster to design and build a computer simulation.
- *Time scale*. Some physical systems change too slowly or too quickly. For example, an elementary particle in a high-speed accelerator may decompose in 10^{-15} seconds. At the other end of the time scale, some ecosystems take thousands of years to react to a modification. A simulation can easily model fractions of a second or billions of years, because time is simply a parameter in an equation.
- *Ethical behavior*. Some physical models have serious moral and ethical consequences, perhaps the best known being the use of animals for medical research. In this case, a computational model could eliminate a great deal of suffering.
- *Ease of modification*. If we are not happy with our original design, we would need to construct a brand new physical model. In a simulation, we only need to change some numerical parameters and rerun the model.

This last advantage—ease of modification—makes computational modeling a particularly attractive tool for designing totally new systems. We initialize the system, observe its response, and if we are not satisfied, modify the parameters and run the model again. We repeat this process over and over, always trying to improve performance. Only when we think we have created the best design possible would we actually build it. This "interactive" approach to design, sometimes called **computational steering**, is usually infeasible using physical models, as it would take too much time. This interactive design methodology is diagrammed in Figure 13.1.

Computational models are therefore an excellent way to design new systems and to study and improve the behavior of existing systems. Virtually every branch of science and engineering makes use of models, and it is not unusual today to see chemists, biologists, physicists, ecologists, and physicians conducting research at their computer screens rather than in the laboratory.

FIGURE 13.1
Using a Simulation in an Interactive Design Environment



Computational models often use advanced mathematical techniques far beyond the scope of this text (and solving them often requires the large-scale parallel computers mentioned in Chapter 5). Therefore, in the following pages we often rely on rather simple examples, far simpler than the models you will encounter in the real world. However, even these simple examples illustrate the enormous power and capabilities of computational modeling.

13.2.2 *Computational Models, Accuracy, and Errors*

Legend says that in the late sixteenth century the famed scientist Galileo Galilei dropped two balls from the top of the Tower of Pisa—a massive iron cannonball and a lighter wooden one—to disprove the Aristotelian Theory, which predicted that heavy objects would fall faster than light ones. When Galileo dropped the two balls they hit the ground at the same time, exactly as he had hypothesized. Whether this event actually took place (and there is considerable debate), it is an excellent example of scientific experimentation

using a physical system, in this case two balls of different weight, a high platform, and the earth below.

Today, we do not need to climb the Tower of Pisa because there is a well-known mathematical model that describes the behavior of a falling mass acted upon only by the force of gravity:

$$d = v_{init}\, t + 1/2\, gt^2$$

This equation says that if a mass in free fall has an initial velocity v_{init} meters/sec at time 0, then at time t it will have fallen a distance of d meters. (Notice that the object's mass is not part of the equation. This is exactly what Galileo was trying to demonstrate.) The factor g is the acceleration due to gravity, which is assumed to be 9.8 meters/sec^2 everywhere along the Earth's surface.

Using this model, we can reproduce aspects of Galileo's sixteenth-century experiment without having to travel to Italy. For example, we can determine the time when the two balls Galileo dropped from the 54-meter high Tower of Pisa would have hit the ground, assuming that their initial velocity was 0.0:

$$54 = (0 * t) + 1/2 * 9.8 * t^2$$
$$t^2 = 11.02$$
$$t = 3.32 \text{ seconds}$$

This simple example shows the beauty and simplicity of computational models. Such models can provide quick answers to questions without the cumbersome setup often required of physical experiments. This model is also easy to modify. For example, if we want to know how long it takes those same two balls to hit the ground when dropped from a height of 150 meters, rather than 54, we reset d to 150 and solve the same equation:

$$150 = (0 * t) + 1/2 * 9.8 * t^2$$
$$t^2 = 30.6$$
$$t = 5.53 \text{ seconds}$$

To use a physical model, Galileo would have had to scour the sixteenth-century world for a 150-meter high tower. (A mathematical model is also much safer because no one ever fell off the top of an equation!)

Unfortunately, modeling is not quite as simple as we have just described, and there are a number of issues that must be addressed and solved to make this technique workable.

The first issue is achieving the proper balance between **accuracy** and **complexity**. Our model must be an accurate representation of the physical system, but at the same time, it must be simple enough to implement as a program or set of equations and solve on a computer in a reasonable amount of time. Often this balance is not easy to achieve, as most real-world systems are acted upon by a large number of factors. We need to decide which of those factors are important enough to be included in our model and which can safely be omitted.

For example, the model of a falling body given earlier is inaccurate because it does not account for the effects of air resistance. (It is only an appropriate model if the object is falling in a vacuum.) Whereas the effect of air resistance on a cannonball may be minimal, imagine dropping a feather! The model would produce inaccurate results, and our conclusions about how

the system behaves would be wrong. It is obvious that we need to incorporate the effects of air resistance into our model if we have any hope of producing worthwhile and useful results.[1]

Our model also assumes that the Earth is a perfect sphere and that the acceleration due to gravity is the same everywhere along its surface. That assumption is not quite true. The Earth is a "slightly squashed" sphere with a radius of 6,378 km at the equator and 6,357 km at the poles. This means that the acceleration due to gravity is a tiny bit greater at the North and South Poles than at the equator, because the poles are 21 km closer to the center of the Earth. Is this something for which we should account? Is it important when constructing a model of a freely falling body? In this case probably not, because the miniscule error resulting from this approximation will almost certainly not affect our conclusions.

This is how computational models are built. We include the truly important factors that act upon our system so that our model is an accurate representation but omit the unimportant factors that make the model harder to build, understand, and solve. As you might imagine, identifying these factors and distinguishing the important from the unimportant can be a daunting task.

Another problem with building simulations is that we may not know, in a mathematical sense, exactly how to describe certain types of systems and behaviors. The gravitational model given earlier is an example of a **continuous model**. In a continuous model, we write out a set of explicit mathematical equations that describes the behavior of a system as a continuous function of time *t*. These equations are then solved on a computer system to produce the desired results. Unfortunately, there are many systems that cannot be modeled using precise mathematical equations because researchers have not discovered exactly what those equations should be. Simply put, science is not yet sufficiently knowledgeable about how some systems function to characterize their behavior using explicit mathematical formulae.

In some cases what makes these systems difficult to model is that they contain **stochastic components**. This means that there are parts of the system that display **random** behavior, much like the throw of the dice or the drawing of a card. In these cases, we cannot say with certainty what will happen to our system because it is the very essence of randomness that we can never know what event will occur next. An example of this is a model of a business in which customers walk into the store at random times. In these cases we need to build models that use **statistical approximations** rather than precise and exact equations. We will present one such example in the following section.

In summary, computational modeling is a powerful but complex technique for designing and studying systems. Building a good model can be a difficult task that requires us to capture, in computational form, all the important factors that influence the behavior of a system. If we are able to successfully build such a model, then we have at our disposal a powerful tool for studying the behavior of that system. This is how a good deal of quantitative research is being done today. Simulation is also an interesting area of study within computer science itself. Researchers in this field create new

---

1 The resistance of the air, called drag, is given by the equation $D = KrV^2A/2.0$, where K is the coefficient of drag, r is the air density, V is the velocity of the object, and A is the reference area of the object. Now you can begin to see why computational models can quickly become so complex.

techniques, both algorithms and special-purpose languages, that allow users to design and implement computer models more quickly and easily.

13.2.3 *An Example of Model Building*

As we mentioned at the end of the previous section, there are many ways to build a model, but most of them require mathematical techniques far beyond the scope of this text. In this section we will construct a model using a method that is relatively easy to understand and does not require a lot of complex mathematics. It is called **discrete event simulation**, and it is one of the most popular and widely used techniques for building computer models.

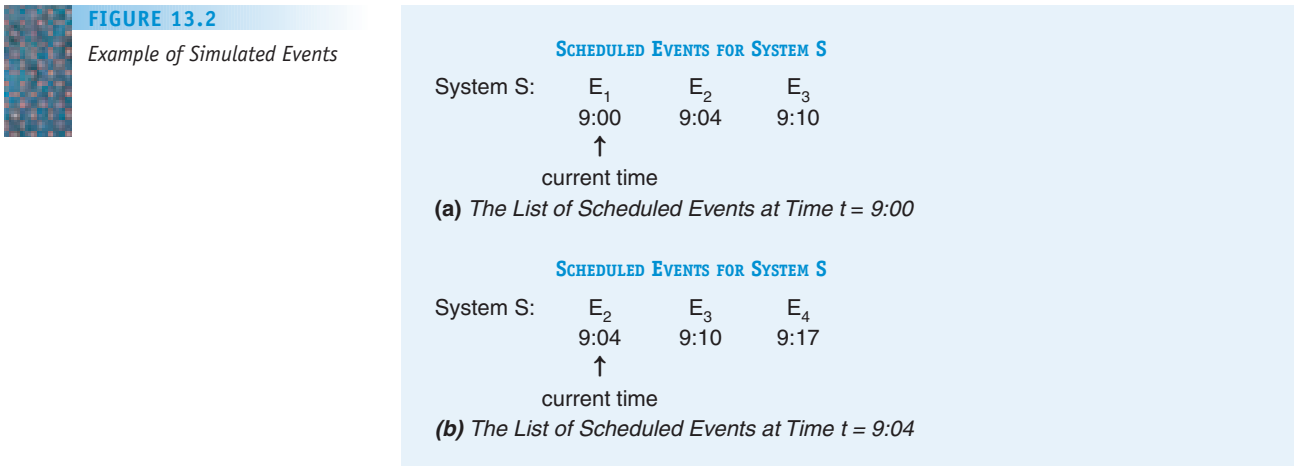
In a discrete event simulation, we do not model time as continuous, like the falling body model in the last section, but as **discrete**. That is, we model the behavior of a system only at an explicit and finite set of times. The moments we model are those times when an event takes place, an **event** being any activity that changes the state of our system. For example, if we are modeling a department store, an event might be a new customer entering the store or a customer purchasing an item.

When we process an event, we change the state of the simulated system in the same way that the actual system would change if this event had occurred in real life. In the case of a department store, this might mean that when a customer arrives we add one to the number of customers currently in the store or, if a customer buys an item, we decrease the number of these items on the shelf. Furthermore, the processing of one event can cause new events to occur some time in the future. For example, a customer coming into a store creates a later event related to that customer leaving the store. When we are finished processing one event we move on to the next, skipping those times when nothing is happening, that is, when there are no events scheduled to occur.

Figure 13.2(a) shows system S and three events scheduled to occur within system S: event E_1 at time 9:00, event E_2 at time 9:04, and event E_3 at time 9:10.

Because E_1 is the event currently being processed, the variable *current time,* which functions like a "simulation clock," has the value 9:00. Let's assume that E_1 causes a new event, E_4, to be created and scheduled for time

FIGURE 13.2
Example of Simulated Events



(a) *The List of Scheduled Events at Time t = 9:00*

(b) *The List of Scheduled Events at Time t = 9:04*

9:17. We add this new event to the list of all scheduled events. When we are finished processing event E_1, we remove it from the list and determine the next event scheduled to occur in system S, in this case E_2. We move *current time* ahead to 9:04, skipping the period 9:01–9:03, because nothing of interest happens, and begin processing E_2. The new list of events scheduled for system S is shown in Figure 13.2(b).

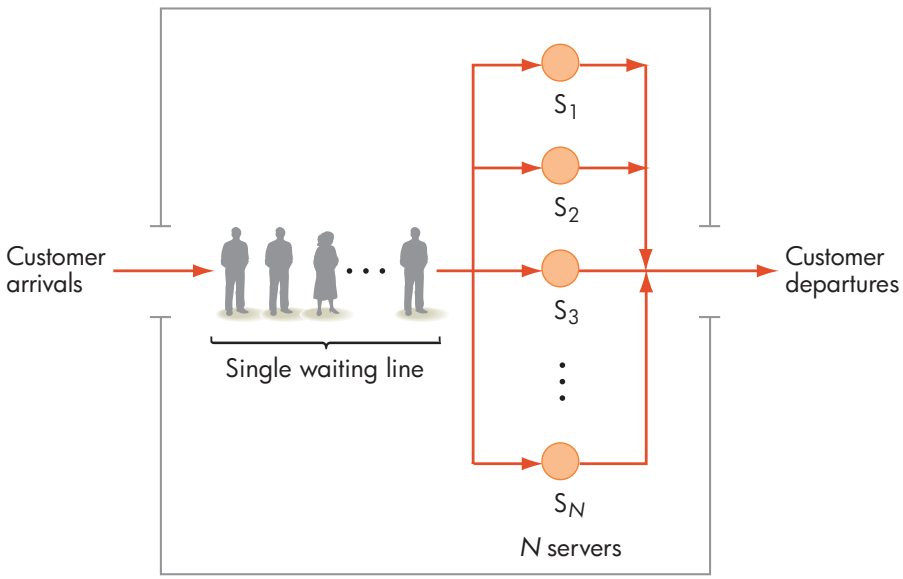
We repeat this same sequence—process an event, remove it from the list, add newly created events to the list, move on to the next event—as long as desired. The variable *current time* keeps advancing as we process the events in strict time order. Typically the simulation is terminated when *current time* reaches some upper bound. For example, in a department store we might choose to run the model until closing time. When the simulation is complete, the program displays a set of results that characterizes the system's behavior and allows the user to examine these results at their leisure.

Let's apply this modeling technique to an actual problem. Assume that you are the owner of a new take-out restaurant, McBurgers, currently under construction. You want to determine the proper number of checkout stations needed in your new store. This is an important decision because, if there are too few checkout stations, the lines will get long and customers will leave. If there are too many checkout stations, you will waste money paying for unnecessary construction costs, equipment, and personnel. Because you took a computer science class in school, you decide to build a simulation model of your new restaurant and use this model to determine the optimal number of servers.

The system being simulated is shown in Figure 13.3. Customers enter the restaurant and wait in a single line for service. If any of the *N* servers is available, where *N* is an input value provided by the user, the first customer in line goes to that station, places an order, waits until the order is processed, pays, and departs. During that time the server is busy and cannot help anyone else. When the server is finished with a customer, he or she can immediately begin serving the next person, if someone is in line. If no one is waiting, then the server waits until a new customer arrives.

To create a model, we must first identify the events that can change the state of our system and thus need to be included in the model. In this example there are two: a new customer arriving and an existing customer departing

FIGURE 13.3
System to Be Modeled



after receiving food and paying. An arrival changes the system because either the waiting line grows longer or an idle cashier becomes busy. A departure changes the system because the cashier serving that customer either begins serving a new customer or becomes idle because no one is in line.

For each of these two events we must develop an algorithm that describes exactly what happens to our system when the event occurs. Figure 13.4 shows the algorithm for the new customer arrival event.

Let's look at this algorithm in more detail. When a new customer arrives, we record the time. The arrival time of each new customer is stored in a separate variable until that customer is served and departs. As we mentioned earlier, when the simulation is finished, we want to display a set of results that allows a user to determine how well the system performed. The total time a customer spends in the restaurant (waiting time + service time) is a good example of this type of result. If this value is large, we are not doing a good job serving customers, and we need to increase the number of servers so that customers don't wait so long. A large part of any simulation model is collecting data about the system so that we can understand and analyze its performance.

The next thing in our New Customer Arrival algorithm is to determine if there is an idle server. If not, the customer goes to the end of the waiting line (no special treatment here at McBurgers), and the length of the waiting line is increased by 1. If there is an idle server then the customer goes directly to that server, who is then marked as busy. (*Note:* If more than one server is free, the customer can go to any one because our model assumes that all servers are identical. We could also construct a model in which not all servers are identical and some provide a special service.)

Now we must determine how much time is required to service this customer. This is a good example of what we termed a stochastic, or random, component of a simulation model. Exactly what a customer orders and how much time it takes to fill that order are random quantities whose exact value can never be known in advance. However, even though it behaves randomly, it is possible that this value, called T_{serve} in Figure 13.4, follows a pattern called a **statistical distribution**. If we know this pattern, then the computer can generate a sequence of random numbers that follows this pattern, and this sequence accurately models the time it takes to serve customers in real life.

How can we discover this pattern? One way is to know something about the statistical distribution of quantities that behave in a similar way. For example, if we know something about the distribution of service times for

FIGURE 13.4

Algorithm for New Customer Arrival

```
New Customer Arrival
    Record the time that this customer entered the restaurant
    Check if any one of the N servers S1, S2, ... , SN is currently idle
    If all of the servers are busy then
        Put this customer at the end of the waiting line
        Increase the length of the waiting line by 1
    Else (server Si is idle)
        Mark that server Si is now busy
        Determine how long it will take to serve this customer, call
            that value Tserve
        Schedule a customer departure event for (current time + Tserve)
        Increase the total time that server Si has worked by Tserve
End of New Customer Arrival
```

customers in a bank or a grocery store, then this information might help us understand the pattern of service times at our hamburger stand. Another way is to observe and collect data from an actual system similar to ours. For example, we could go to other take-out restaurants and measure exactly how long it takes them to service their customers. If these restaurants were similar to ours, then the McBurgers owner might be able to discover from this data the statistical distribution of the variable T_{serve}.

There are other ways to work with statistical distributions, but we will leave this topic to courses in statistics. In this example we simply assume that the statistical distribution for the customer service time, T_{serve}, has been discovered and is shown in the graph in Figure 13.5.

The graph in Figure 13.5 states that 5% of the time a customer is served in less than 1 minute; 15% of the time it takes 1–2 minutes; 40% of the time it takes 2–3 minutes; 30% of the time it takes 3–4 minutes; and finally, 10% of the time it takes 4–5 minutes. It never requires more than 5 minutes to serve a customer. We can model this distribution using the algorithm shown in Figure 13.6.

First, we generate a random integer v that takes on one of the values 1, 2, 3, . . . , 100 with equal likelihood. This is called a **uniform random number**. We now ask if v is between 1 and 5. Because there are five numbers in this range, and there were 100 numbers that could originally have been generated, the answer to this question is yes 5% of the time. This is the same percent of time that customers spend from 0 to 1 minute being served. Therefore, we generate another uniform random value, this time a real number between 0.0 and 1.0, which is the value of $T_{serve,}$ the customer service time.

FIGURE 13.5

Statistical Distribution of Customer Service Time

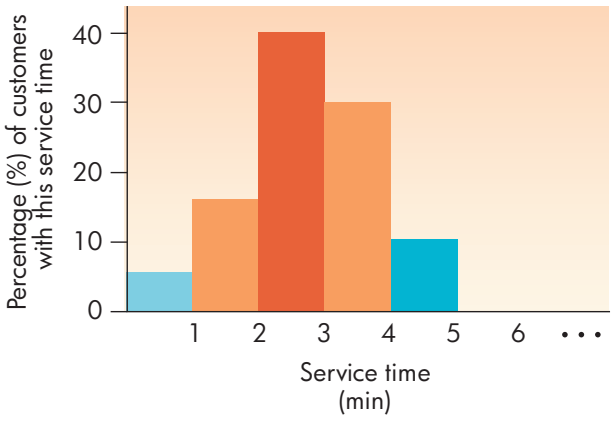


FIGURE 13.6

Algorithm for Generating Random Numbers That Follow the Distribution Given in Figure 13.5

Generate a uniform random integer value v between 1 and 100
If v is in the range 1–5, then
    Set T_{serve} to a uniform random number between 0.0 and 1.0
Else if v is in the range 6–20, then
    Set T_{serve} to a uniform random number between 1.0 and 2.0
Else if v is in the range 21–60, then
    Set T_{serve} to a uniform random number between 2.0 and 3.0
Else if v is in the range 61–90, then
    Set T_{serve} to a uniform random number between 3.0 and 4.0
Else
    Set T_{serve} to a uniform random number between 4.0 and 5.0

If the original random value v is not between 1 and 5, we ask if it is between 6 and 20. There are 15 integers in this range, so the answer to this question is yes 15% of the time, exactly the fraction of time that customers spend 1–2 minutes being served. If the answer is yes, we generate a T_{serve} value that is in the range 1.0 to 2.0. This process is repeated for all possible values of service time.

Once the value of T_{serve} has been generated, we use this value to determine exactly when this customer leaves the store (*current time* + T_{serve}) as well as to update the total amount of time the server has spent serving customers. This last computation allows us to determine the percentage of time during the day that each server was busy.

The value assigned to T_{serve} using the algorithm of Figure 13.6 exactly matches the statistical distribution shown in Figure 13.5. If this graph is an accurate representation of customer service time, then our model is an accurate depiction of what happens in the real world. However, if the graph of Figure 13.5 is not an accurate representation of the customer service time, then this model is incorrect and will produce wrong answers. This is a good example of the well-known computer science dictum **garbage in-garbage out**. The results you get out of a simulation model are only as good as the data and the assumptions put into the model.

We can now specify how to handle the second type of event contained in our model, which is customer departures. The algorithm to handle a customer leaving the restaurant is given in Figure 13.7.

When a customer is ready to leave, we determine the total time this customer spent in the restaurant. The variable *current time* represents the time now, which is the time of this customer's departure. We recorded the time this customer first arrived on line 2 of Figure 13.4, and we can retrieve the contents of the variable storing that information. The difference between these two numbers is the total time this customer spent in the restaurant. We use this result, averaged over all the customers, to determine if we are providing an adequate level of service.

If there is another customer in line, the server begins serving that customer in exactly the same way as described earlier. If no one is waiting, then the server is idle and has nothing to do until a new customer arrives. (We don't want this to happen too often as we will be paying the salary of someone with little to do.)

We have now described the two main events that change our system: someone arriving and someone leaving the restaurant. The only thing left is to

FIGURE 13.7

Algorithm for Customer Departure Event

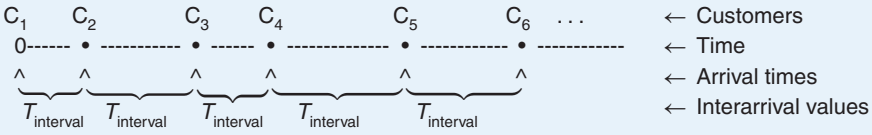
```
Customer Departure from Server Si
    Determine the total time that this customer spent in the restaurant
    If there is someone in line then
        Take the next customer out of line and decrease the waiting line size by one
        Determine how long this new customer will take to be served,
            and call that value Tserve
        Schedule a customer departure event for (current time + Tserve)
        Increase the total time that server Si has worked by Tserve
    Else
        Mark this server as idle
End Customer Departure
```

initialize our parameters and get the model started. To initialize the model we must do the following four things:

- Set the current time to 0.0 (we begin our simulation at time 0)
- Set the waiting line size to 0 (no one is in line when the doors open)
- Get a value for *N*, the number of servers, and make them all idle
- Determine the total number of customers to be served and exactly when they will arrive

The last value—customer arrival times—are like the service times discussed earlier in that they are stochastic, or random, values that cannot be known with certainty. We cannot possibly know exactly when the next customer will walk in the door. However, if we know the statistical distribution of the time interval between the arrival of any two customers, then we can generate a set of random intervals, called T_{interval}, that allows us to accurately model our customer arrivals.

Assume we have a graph like Figure 13.5 that specifies the statistical distribution of the time interval that elapses between the arrivals of two successive customers. (That is, it might say something like 10% of the time two customers arrive within 0–15 seconds of each other, 20% of the time they arrive within 15–30 seconds, etc.) We schedule our first customer to arrive at time 0.0, just as the doors open. We then use an algorithm like the one in Figure 13.6 to generate a random value that matches the distribution of interarrival times. Call this value T_{interval}. This represents the amount of time that will elapse until the next customer arrives. Because the first customer arrived at time 0.0, we schedule the next one to arrive at $(0.0 + T_{\text{interval}}) = T_{\text{interval}}$. We repeat this for as many customers as desired, scheduling each one to arrive at T_{interval} time units after the previous one. Our sequence of customer arrivals will look something like this:



The main program to run our McBurgers simulation model is given in Figure 13.8. It allows the user to provide two inputs: *M*, the number of customers they want to model, and *N*, the number of servers. Each one of the *M* customer arrivals is handled by the arrival algorithm of Figure 13.4. Each arrival event generates a customer departure event that is handled by the departure algorithm of Figure 13.7. This simulation does not terminate at a specific point in time but, instead, when there are no more events to be processed—that is, every customer who was scheduled to arrive has been served and has departed.

The last issue that we must address is how to implement the second to last line of Figure 13.8, the one that reads, "Print out a set of data that describes the behavior of our system." Looking back at Figure 13.1, we see that one of the responsibilities of a simulation is to "collect data describing its behavior." Our model must collect data that accurately measures the performance of our

FIGURE 13.8

The Main Algorithm of our Simulation Model

```
Main Part of the Simulation Model
    Set current time to 0
    Set the waiting line size to 0
    Get an input value for N, the number of servers
    Set all N servers, S1, S2, ... , SN to idle
    Get an input value for M, the total number of customers
    Schedule M customer arrivals and put them on the list of events
        Each arrival occurs Tinterval time units after the previous one
    While there is still a scheduled event on the list do
        Get the next event on the list
        Move current time to the time of this event
        If this is a customer arrival event
            Execute the arrival algorithm of Figure 13.4
        Else
            Execute the departure algorithm of Figure 13.7
        Remove this event from the list of all scheduled events
    End of the loop
    Print out a set of data that describes the behavior of our system
Stop
```

McBurgers restaurant so that we can configure it in a profitable manner *before* it is built. Therefore, we need to determine what data are required to meet this need. Often this cannot be done by the person building the model because they may not know anything at all about this application area. Instead, it is the *user* of a model who determines what data should be displayed. In our case, the user is the restaurant owner.

Let's assume that we have talked to the owner and determined that the information he or she most needs to know is the following:

- The average time that a customer spends in the restaurant, including both waiting in line and getting served
- The maximum length of the waiting line
- The percentage of time that servers are busy serving customers

From this data the owner should be able to determine whether the system is functioning well. For example, if our model determines that a server is busy only 10% of the time (about 48 minutes in an 8-hour workday), we can probably reduce the number of servers without affecting service, saving a good deal in salary costs. On the other hand, if the average time that a customer spends in the restaurant is 1 hour or there are times when there are 100 people in line, then we had better increase the number of servers if we want to avoid bankruptcy (or riots)!

This model will likely be used in the interactive design approach first diagrammed in Figure 13.1. The owner will enter his or her best estimate for the arrival time and service time distributions and then select a value for *N*, the number of servers. The computer will run the simulation, processing all *M* customers, and then print the results, perhaps something like the following:

Servers	*Average Waiting Time (min)*	*Maximum Line Length*	*Server Busy Percentage (%)*
2	*63.3*	*35*	*100.0*

With only two servers, our customers waited on average more than one hour to be served, there were dozens of people in line, and both servers were busy every second of the day—not very good performance! The owner would certainly try to improve on this performance, perhaps by having 6 servers, rather than only 2. He or she resets the parameter *N* to 6 and reruns the model, which now produces the following:

Servers	*Average Waiting Time (min)*	*Maximum Line Length*	*Server Busy Percentage (%)*
6	*2.75*	*1*	*43*

Now the owner may have erred too far in the other direction. Our customers are being well served, waiting only a couple of minutes, and the line is tiny, never having more than a single person. However, our six servers are busy only 43% of the time—meaning they are idle about 4.5 hours during an 8-hour workday. Could we provide the same high level of service to our customers with fewer servers? To answer this question, the owner might try rerunning the model with $N = 3$, 4, or 5, a compromise value between these two extremes. This is how a simulation model is used—run it, examine the results, and use these results to reconfigure the system so its performance is enhanced.

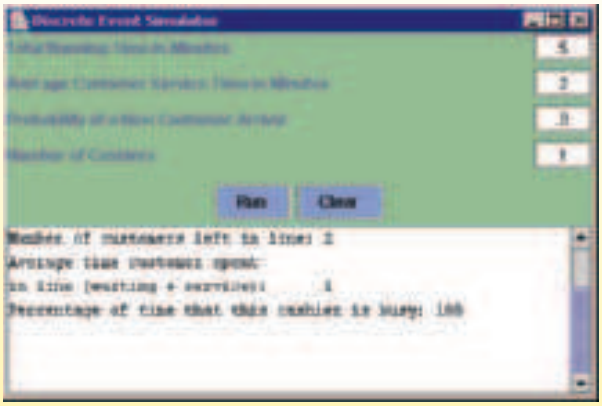
This completes the development of our McBurgers simulation, but not the end of its usefulness. In the next laboratory experience you are going to "play" with this model by selecting a range of different values for the customer arrival and service times. You then take on the role of the McBurger's owner and determine the optimal number of servers to use for the selected configuration. Working with a simulation in an interactive design environment demonstrates the enormous power and capabilities of computational models.

The restaurant modeled in this section is about as simple a system as we could present, yet it still took almost eight pages to describe. A computational model of a suspension bridge, "El Niño" Pacific Ocean currents, the human heart, or a strand of DNA would certainly be much more complex than the simulation of a hamburger joint! Real-world models are mathematically intricate, highly detailed, and difficult to build. However, if we are able to build such a model or if we have access to such a model, then we have a powerful tool that can significantly enhance our ability to do high-quality research and design.

PRACTICE PROBLEMS

1. In the McBurgers new customer arrival algorithm, describe the consequences of accidentally omitting the instruction "Mark that server S_i is now busy."
2. In the McBurgers customer departure algorithm, describe the consequences of accidentally omitting the instruction "Mark this server as idle."

LABORATORY EXPERIENCE 18



In this Lab Experience you will work with a simulation model of a McBurgers restaurant that is similar to the one presented in this section. You will play the role of the restaurant owner who is trying to determine the correct number of servers for a specific pattern of customer arrivals and service times. You will configure your restaurant, run the model, see how well you serviced your customers, and then reconfigure the restaurant to try to improve its performance and its profit. The software allows you to set parameters for (1) the total running time of the simulation, (2) the average service time of each customer, (3) the probability that a new customer will arrive, and (4) the number of servers. It will then run the model exactly as you have described and, upon completion, produce the following output: (1) the number of customers remaining in line when the simulation terminated, (2) the average time that a customer spent in the restaurant, and (3) the percentage of time that the cashier was busy over the entire simulation. The screen shot shown here is typical of what you will see when you run the lab.

Your goal in this simulation is to determine the set of parameters that optimizes behavior of the overall system.

13.3 Running the Model and Visualizing Results

The McBurgers restaurant model developed in Section 13.2.3 is much simpler than real-world models for two reasons. First, it is computationally small. Running it and producing results does not require much in the way of hardware resources. For example, assume that we model M = 1,000 customers, a reasonable value for a large restaurant. Each customer generates one arrival event (Figure 13.4) and one departure event (Figure 13.7), for a total of 2,000 events that must be processed by the computer before the simulation is completed and the results displayed. Two thousand events is a miniscule amount of work that could be handled by even the smallest desktop machine in just a few seconds or, more likely, fractions of a second. Most real-world models require much more computational work to produce their results.

For example, the U.S. Department of Energy's National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory has developed a powerful new climate system model. Using this model, simulating one year of global climatic change requires about 10^{17} computations—one hundred thousand trillion operations. A single Von Neumann machine could not handle this almost unimaginably large amount of work. A typical desktop computer executes roughly 1 billion instructions per second—1,000 MIPS. At this rate, completing one year of simulated time in the model would require three years of real time—we would not get our results until the actual time period being simulated had passed!

Massive models like this one can be executed only on the large-scale parallel machines described in Chapter 5. The NERSC climate model was executed on a massively parallel IBM-SP supercomputer containing 6,080 processors, with a peak computation rate of 6 teraflops, or 6 trillion operations per second. At this rate, one year of climatic change can be modeled in about five hours.

These numbers are much more typical of the amount of work required by real-world simulations. It is not unusual for a model to perform 10^{15}, 10^{16}, 10^{17}, or more computations to produce a single result—amounts far beyond the capabilities of individual machines. The increasing interest in building and using computational models is one of the main reasons behind the development of larger and more powerful supercomputers.

The second reason why the McBurgers model in Section 13.2.3 is so unrealistic is that it produces only a tiny amount of output. After each run is complete the model generates only three lines of output, such as those shown below and in the previous section:

Servers	*Average Waiting Time (min)*	*Maximum Line Length*	*Server Busy Percentage (%)*
6	*2.75*	*1*	*43*

Because the number of servers in a restaurant might range from one up to a couple of dozen, the total volume of output this model would ever produce is about 20–60 lines, less than a single page. With such a small amount of output, our model can display its results using a simple text format, as shown in the lines above. A user will have no difficulty reading and interpreting this output.

Unfortunately, most simulations do not produce a few dozen lines of output, but rather tens or hundreds of thousands of lines, perhaps even millions. For example, assume the NERSC climate model described earlier displayed the temperature, humidity, barometric pressure, wind velocity, and wind direction at 50-mile intervals over the surface of the Earth for every simulated day the model is run. After one year of simulated time, it will have produced roughly 500 million data values—about 10 million pages of output! If these values were displayed as text, it would overwhelm its users, who wouldn't have a clue how to deal with this mountain of paper.

Text, when it appears in such large amounts, does not lend itself to easy interpretation or understanding. The field of **scientific visualization** is concerned with the issue of how to visualize data in a way that highlights its important characteristics and simplifies its interpretation. This is an enormously important part of computational modeling, because without it we would be able to construct models and execute them, but we would not be able to interpret their results.

The term *scientific visualization* is often treated as synonymous with the related term **computer graphics**, but there is an important difference. The field of computer graphics is concerned with the technical issues involved in information display. That is, it deals with the actual rendering of an image—light sources, shadows, hidden lines and surfaces, shading, contours, and perspective. Scientific visualization, on the other hand, is concerned with how to visually display a large data set in a way that is most helpful to users and that maximizes its comprehension. It is concerned with issues such as **data extraction**, namely, determining which data values are important and should be part of the visual display and which ones can be omitted, and **data manipulation**, which consists of looking for ways to convert the data to other forms or to different units that will make the display easier to understand and interpret. Once we have decided exactly how we wish to display the data, then a scientific visualization package typically uses a computer graphics package to render an image on the screen or the printer.

For example, assume we have built a computer model of the ocean tides at some point along the coast. Our model predicts the height of the tide every 30 seconds in a 24-hour day, based on such factors as the lunar phase, wind speed, and wind direction. If this information is printed simply as text, it might look something like the following:

Time	*Height (feet)*
12:00:00 A.M.	43.78
12:00:30 A.M.	43.81
12:01:00 A.M.	43.84
12:01:30 A.M.	43.88
12:02:00 A.M.	43.92
12:02:30 A.M.	43.97
.	.
.	.
.	.
11:57:00 P.M.	45.08
11:57:30 P.M.	45.04
11:58:00 P.M.	45.01
11:58:30 P.M.	44.99
11:59:00 P.M.	44.97
11:59:30 P.M.	44.95

There are 2,880 lines of output, which at 60 lines per page would produce almost 50 printed pages. Trying to extract meaning or locate significant features from these long columns of numbers would certainly be a formidable, not to mention boring, task.

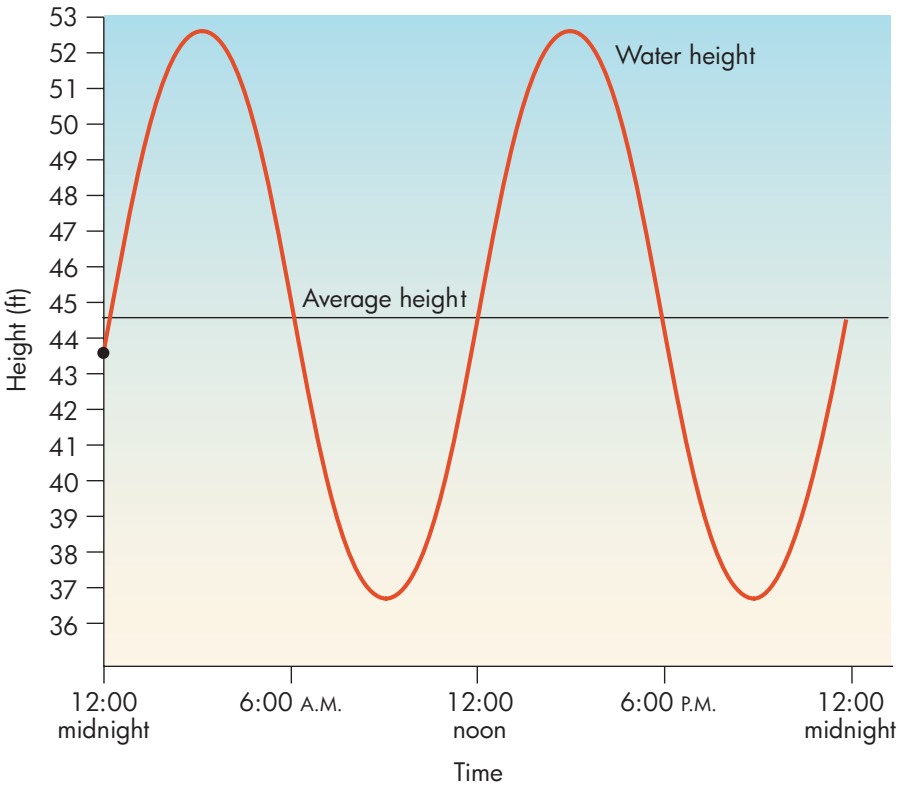
What if, instead, we displayed these two columns of values as a two-dimensional graph of time versus height? The output could also include a horizontal line showing the average water height during this 24-hour period. This latter value is not part of the original output but can easily be computed from these values and included in the output—an example of a data manipulation carried out to enhance data interpretation. Now the output of our model might look something like the graph in Figure 13.9.

Using the graph in Figure 13.9, it is a lot quicker and easier to identify the interesting features of the model's output. For example,

- There appear to be two high tides and two low tides during this 24-hour time period.
- The high tide is about 8 feet above the average water level, whereas the low tide is about 8 feet below the average water level.

It is possible to discover the same features from a textual representation of the output, but it would probably take much more time. Interpreting the graph of Figure 13.9 is a great deal easier than working directly with raw numerical data. The use of visualizations becomes even more important as the amount of output increases and grows more complex. For example, what if in addition to the tidal height our model also predicted the water temperature

FIGURE 13.9
Using a Two-Dimensional Graph to Display Output



and displayed its value every 30 seconds. Now the raw data produced by the model might look like this:

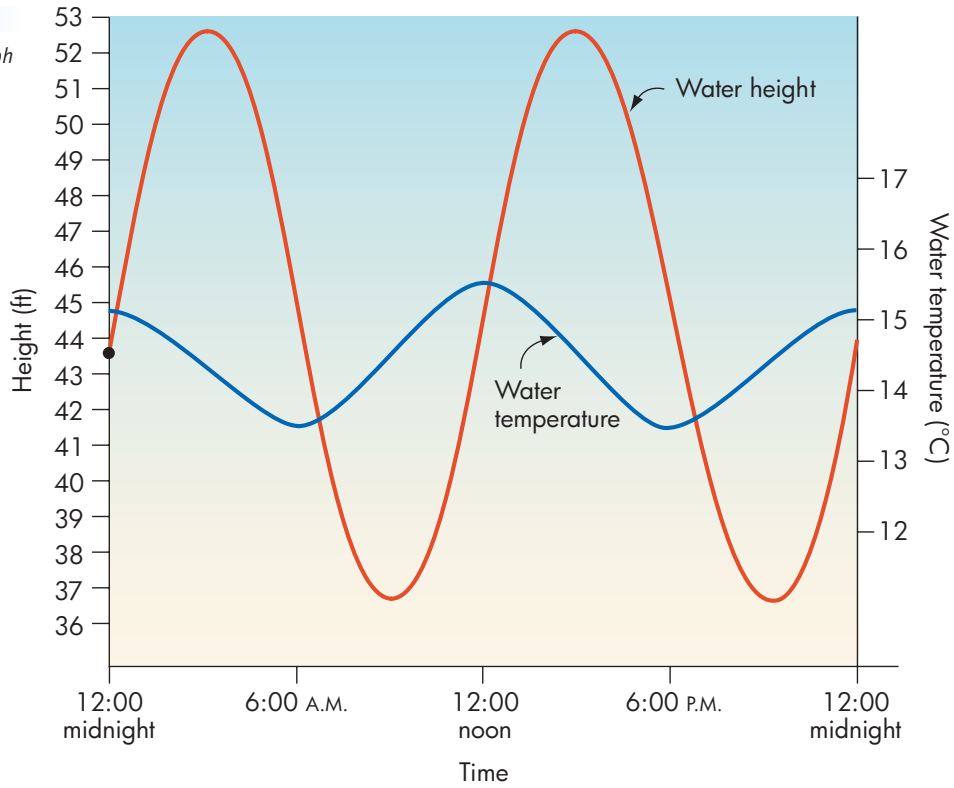
Time	*Height (feet)*	*Temperature (°C)*
12:00:00 A.M.	43.78	15.03
12:00:30 A.M.	43.81	15.02
12:01:00 A.M.	43.84	15.01
12:01:30 A.M.	43.88	14.99
12:02:00 A.M.	43.92	14.97
12:02:30 A.M.	43.97	14.94
.	.	.
.	.	.
.	.	.
11:57:00 P.M.	45.08	14.95
11:57:30 P.M.	45.04	14.98
11:58:00 P.M.	45.01	15.00
11:58:30 P.M.	44.99	15.01
11:59:00 P.M.	44.97	15.03
11:59:30 P.M.	44.95	15.05

Now there are almost 6,000 numbers, and our task has become even more difficult as we try to understand the behavior of the *two* variables, height and temperature. Working directly with the raw data generated by the model is cumbersome. However, if the value of both variables were presented on a single graph, as shown in Figure 13.10, this interpretation is much easier.

Looking at Figure 13.10, we can quickly observe that temperature seems to move in exactly the opposite direction as the tide, but delayed by a few minutes. That is, water temperature reaches its minimum value shortly after the tidal height has reached its maximum value, and vice versa. This is exactly the

FIGURE 13.10

Using a Two-Dimensional Graph to Display and Compare Two Data Values

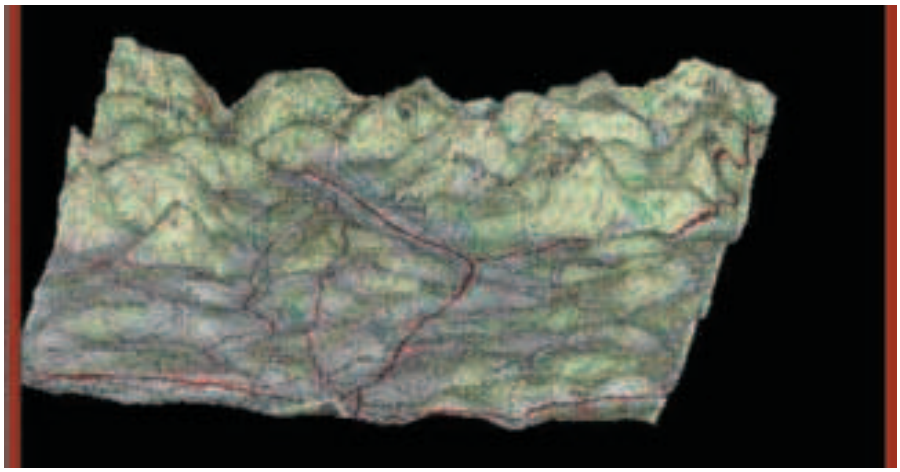


type of information that could be of help to a researcher. Without the graphical visualization in Figure 13.10, we may have overlooked this important fact.

The graphs in Figures 13.9 and 13.10 are both two-dimensional, but many real-world models study the behavior of three-dimensional objects, for example, an airplane wing, a gas cloud, or the Earth's surface. The results produced by these models are also three-dimensional, such as the spatial coordinates of a point on that airplane wing or on a gas molecule. Therefore, it is common for the output of a computational model to be displayed as a three-dimensional image rather than the two-dimensional graphs shown earlier. For example, Figure 13.11 shows a computer model of a portion of the Earth's surface. Using this three-dimensional image, it is easy to locate important topographical features, such as mountains, valleys, and rivers. This type of output would be extremely useful when, for example, planning the location of roads and bridges.

FIGURE 13.11

Three-Dimensional Image of a Region of the Earth's Surface



As a second example, suppose that medical researchers are using a simulation model to study the behavior of the chemical compound methyl nitrite, CH_3NO_2, a potential carcinogen found in our air and drinking water. Assume that their molecular model produces the following textual output:

		Location			
Molecule Number	*Element*	*x*	*y*	*z*	*Bonded To*
1	O	1.7	1.0	0.0	3, 4
2	O	3.0	0.0	0.0	3
3	N	2.6	0.3	1.0	1, 2
4	C	0.0	0.0	0.0	1, 5, 6, 7
5	H	–0.5	0.5	0.5	4
6	H	0.5	0.5	0.5	4
7	H	–0.5	-0.5	0.5	4

This is an accurate textual description of a methyl nitrite molecule. The output specifies the seven atoms in the molecule, the spatial (*x, y, z*) coordinates of the center of each atom, and the identity of all other atoms to which this one has a chemical bond. This is all the information required to understand the structure of this molecule. However, most of us would find it hard to form a mental image of what this molecule actually looks like using this table.

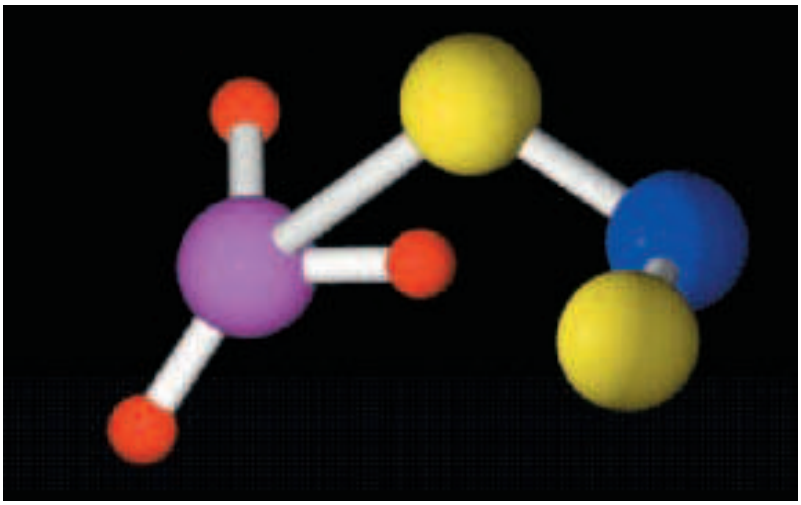
What if, instead, our model took this textual description of methyl nitrite and used it to create and display the three-dimensional image of Figure 13.12?

It is certainly a lot easier to work with the visualization in Figure 13.12 than with the numerical description. For example, if the simulation model changed the shape or structure of this molecule, say by modeling a chemical reaction or stretching a chemical bond, we would be able to observe this change on our computer screen, significantly increasing our understanding of exactly what is happening.

The image in Figure 13.12 makes use of two other features found in many visualizations—color and scale. These characteristics allow us to display information in a way that makes the image more understandable by someone looking at the diagram. In this example, color represents the element type—red for hydrogen, purple for carbon, yellow for oxygen, and blue for nitrogen. The relative size of each sphere represents the relative size of each of the atoms.

The clever use of visual enhancements such as color and size can make an enormous difference in how easy or hard it is to interpret the output of a

FIGURE 13.12
Three-Dimensional Model of a Methyl Nitrite Molecule



computer model. For example, the image displayed in Figure 13.13 was generated by simulating the dispersion of a toxic gas cloud in the downtown area of a major city. Based on wind speed and direction, the location of buildings, and the molecular structure of the gas, this model determines the gas concentration throughout the downtown area at discrete points in time. (Figure 13.13 shows one of these time points.)

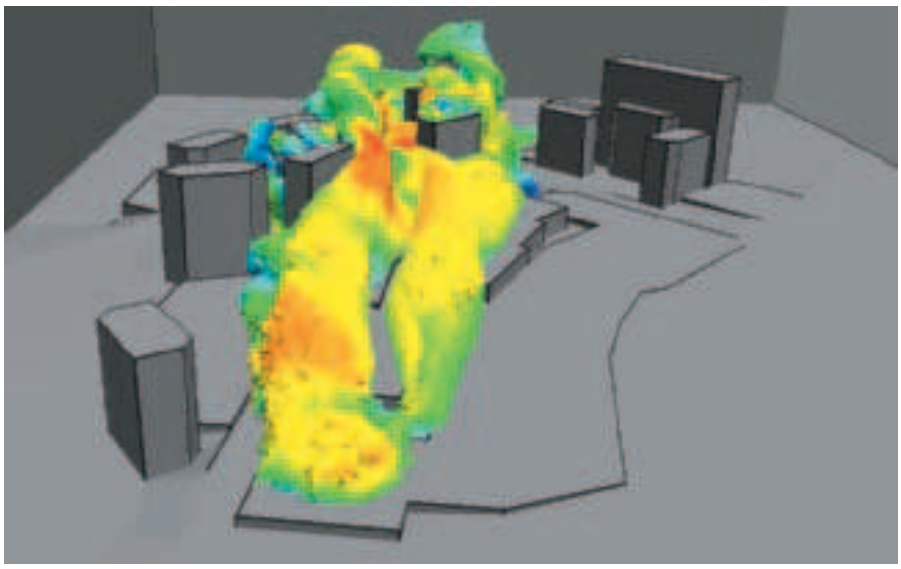
In this example, color indicates the concentration of toxic gas in the atmosphere. Blue and green represent the lowest two levels of concentrations, yellow represents a moderate level, whereas orange and red represent the highest and most deadly concentrations of gas. Using images like Figure 13.13, an emergency crew, knowing the current wind speed and direction, could quickly determine where to direct their rescue efforts in the event of a gas leak. If, instead of these color-coded, three-dimensional images, the crew was given only page after page of numerical values, it would take much longer to extract this vital information. Here is an example where enhancing comprehension is not just for convenience but for saving lives!

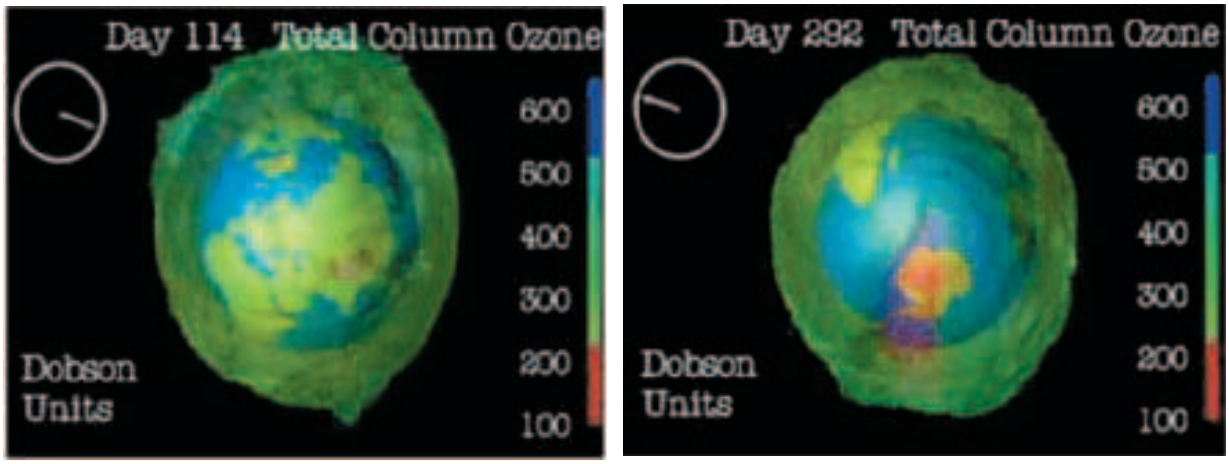
Finally, we mention one of the most powerful and useful forms of visualization—**image animation**. In many models, time (whether continuous or discrete) is one of the key variables, and we want to observe how the model's output changes over time. This could be the case, for example, with the gas dispersion model discussed in the previous paragraphs. The image in Figure 13.13 is a picture of a gas cloud at one discrete instant in time. That may be of some value, but what might be of even greater interest is how the cloud moves and disperses as a function of time. Some questions we could answer using this time-varying model are: How long does it take for the highest levels of gas (red and orange) to dissipate completely? What is the maximum distance from the site of the leak where the highest levels of gas were found?

To answer these and similar questions we need to generate not one image like Figure 13.13, but many, with each image showing the state of the system at a slightly later point in time. If we generate a sufficient number of these images, then we can display them rapidly in sequence, producing a visual animation of the model's output.

Obviously we cannot show an animation in this book, but Figure 13.14 shows two images (out of 365) from a program that models the total amount of ozone

FIGURE 13.13
Visualization of Gas Dispersion





(a) On Day 114

(b) On Day 292

FIGURE 13.14
Use of Animation to Model Ozone Layers in the Atmosphere

present in the Earth's atmosphere over a one-year period. The model computes the ozone levels for each day of the year and displays the results graphically, with green and blue representing acceptable ozone levels and red representing a dangerously low level. These 365 images can be displayed in sequence to produce a "movie" showing how the ozone level changes throughout the year.

The amount of output needed to produce these 365 images was probably in the hundreds of millions of data values, perhaps more. If this enormous volume of data were displayed as text, a user would be overwhelmed, and the truly important characteristics of the data would be buried deep within this mass of numbers, much like the proverbial "needle in a haystack." However, using the visualization techniques highlighted in this section—two- and three-dimensional graphics, color, scale, and animation—key features of the data, such as the presence of a significant ozone hole (the red area) over the Antarctic on day 292, can be quickly and easily located.

This is precisely the reason for the existence of these scientific visualization techniques. It is not merely a desire to produce "pretty pictures," although, indeed, many of the images are artistically interesting. Instead, the goal is to take a massive data set and present it in a way that is more informative and more understandable for the user of that data. Without this understanding, there would be no reason to build computational models in the first place.

13.4 Conclusion

Computational modeling is a fascinating and highly complex subject and one that will become even more important in the coming years as computers increase in power and researchers gain experience in designing and building these models.

Constructing models of complex systems requires a deep understanding of both mathematics and statistics so, as we have mentioned a number of times, they can be rather difficult to build. However, even if you are not directly

involved in building models, it is quite likely that you will *use* these types of models in your research, development, or design work. Simulation is affecting many fields of study. For example, in this chapter we looked at models drawn from physics (the falling body equations), economics (the McBurgers simulation), chemistry (the molecular model of methyl nitrite), cartography (a map of the Earth's surface), meteorology (tides, climatic changes), and ecology (toxic gas dispersion). We could just as easily have selected our examples from medicine, geology, biology, geography, or pharmacology. For those who work in scientific or quantitative fields like these, computational modeling is rapidly becoming one of the most important tools available to the researcher. It is also a vehicle for amusing and entertaining us through the creation of simulated fantasy worlds and alien planets where we can explore and play. We discuss this exciting new role of simulation in Chapter 16.

Even though simulation is an important application of computers, you are probably more familiar with the many uses of computers in the commercial sector—paying bills online, remotely accessing corporate databases, and buying and selling products on the Web. These commercial applications, often grouped together under the generic term *electronic commerce,* or *e-commerce,* will be discussed at length in Chapter 14.

The Mother of All Computations!

Climatic changes occur slowly, often taking hundreds or thousands of years to complete. For example, the ice ages were periods when large areas of the Earth's surface were covered by glaciers. These individual ice ages were separated by intervals of thousands of years during which the Earth became warmer and the glaciers receded. To study global climate change, a researcher cannot look at data for only a few years. Instead, he or she must examine changes taking place over long periods of time.

To provide this type of data, scientists at the National Center for Atmospheric Research (NCAR) in Boulder, Colorado, used the NERSC global climate model described earlier to carry out a 1,000-year simulation of climatic changes on the surface of the Earth. NCAR used a 6,000+ processor IBM-SP supercomputer and started it running in late January 2002. This massive machine worked on the problem 24 hours a day, 7 days a week, modeling decade after decade, century after century of changes to the Earth's climate. Finally, on September 4, 2002, it finished its task. It had taken more than 200 days of uninterrupted computing and the execution of about a hundred billion billion (10^{20}) computations on a multimillion-dollar machine to obtain the results!

Data from this simulation are being made available to the research community to further the study of changes to our climate and investigate such weather-related phenomena as global warming and "El Niño" ocean currents.