Chapter 7: Graphical Problem Solving in Science

This chapter considers the use of computer graphics as a problem-solving tool in the sciences. It does so by example: as we discuss various graphical techniques for presenting different kinds of information, we include examples of images produced by programs that use these techniques for an actual, though usually fairly simple, scientific problem. These techniques range rather widely and it is not easy to define a natural sequence for them, so the presentations are organized in the best way we can. When you have finished with this chapter, you should have a good grasp of several techniques for graphical modeling and simulation that are used in the sciences, and have an understanding of the way these techniques create images that are useful in the various scientific fields.

In order to benefit from this chapter, you need a knowledge of computer graphics through modeling, viewing, and color, together with enough programming experience to implement the various programming techniques that will be discussed in this section to produce the images needed by the science.

Introduction

In the last 20 years, the growing complexity of scientific theory and the volume of scientific data have led to a greatly-increased use of images to represent a wide range of concepts and experiments. The general name for this representation is scientific visualization, and it is a critical part of a great deal of scientific work today. The important thing about scientific visualization is not the images that are created to describe scientific principles or processes; it is the graphical problem solving that must be done in order to create the images, the visual communication that must be done to present the images in an effective way, and the understanding of the science that is created by the images for the science community. This understanding can be used to help students learn the science, to help the public or funding sources appreciate the developments in science and support further development, or to help the researchers in the science grasp the implications of their work as they see it more fully. It is not an accident that the general expression of understanding is “Now I see it!” because our mental processes for understanding images are among the most highly developed of all our abilities.

The role of computer graphics in problem solving is to give us a reason to express a problem in visual or geometric terms, to give us a way to take that expression and make it concrete in an actual image, and finally to give us that image as a source of reflection and insight on the problem that can lead to a better understanding of the problem itself. This is described by the closed cycle in Figure 7.1. The initial problem solving task is captured in the Problem \(\rightarrow\) Geometry link below; the concrete expression in computer graphics terms is illustrated by the Geometry \(\rightarrow\) Image(s) link; and the reflection on the problem leading to better understanding is provided by good visual communication in the Image(s) \(\rightarrow\) Insight link. With this approach, supported by the use of

![Figure 7.1: The graphical problem-solving cycle](image-url)
computer graphics, we are able to reach the Insight → Problem link where the real advancement of our understanding is provided. Together these components describe a powerful way to address any kind of problem by engaging a combination of analytic and visual capabilities.

As we consider our problems and how we can view them graphically or geometrically, we find that we have some basic questions to answer. Can our problem naturally be expressed in terms of objects that have a natural image (automobiles, houses, ...) or have a familiar representation (bar, sphere, pyramid, ...)? If so, then you can start with these natural or familiar objects and see how you can represent your problem in their terms. If not, then you need to try to find or create such a representation of the problem, because you cannot display a solution without geometry. As Galileo said,

“Philosophy is written in this grand book the universe, which stands continually open to our gaze. But the book cannot be understood unless one first learns to comprehend the language and to read the alphabet in which it is composed. It is written in the language of mathematics, and its characters are triangles, circles, and other geometric figures, without which it is humanly impossible to understand a single word of it; without these, one wanders about in a dark labyrinth.” [Galileo’s Daughter, p. 16]

The overall process we describe here is what we call graphical problem solving. The key point is being able to identify ways to describe a problem in geometric terms that permit the design and creation of images that represent part of the problem. The general process or representing a problem in ways that permit better understanding is called *modeling* the problem, and modeling is a large topic that is treated in complete books of its own. However, we want to help you understand the kind of modeling that is part of graphical problem solving, and in this chapter we will describe some kinds of graphical modeling that have been used in addressing various problems in science. These will be relatively simple models because we’re looking at the kind of graphical models that can be created using your beginning graphics tools; much more complex and sophisticated kinds of models have been created and are being used, and you should look at the general literature of scientific visualization to get an understanding of the greater depth that is possible.

As we look at the examples of graphical modeling, we will examine a number of different kinds of scientific problems and for each we will describe the problem and how it can be modeled graphically, and where appropriate describe tradeoffs that we made in choosing the particular representation, and we will describe the way we build an image that is based on that model. Later in the chapter there will be a discussion of some details in the computer modeling (as distinct from the problem modeling) that would go into creating the images for the models; these will sometimes include discussions of the graphics techniques but will sometimes focus on other programming issues. The end product of this work will be a description of the process of problem solving that you should be able to take to your own projects or problems. Together with the set of examples where this process is discussed, this should give you the tools to use graphical problem solving to accomplish your own goals.

Before we move to discussing the modeling examples, we should say a word about the kinds of data that you may encounter when you’re working with scientific information. There are different kinds of data, called interval, ordinal, and nominal data, that must be treated differently when you represent them graphically. Interval data can be represented by real numbers, ordinal data are data that have a natural ordering but no meaningful numerical representation, and nominal data are data that are in different categories but have no ordering. There are few challenges in dealing with interval data, because we are familiar with numerical values and their representations as values in dimensions or even as color ramps. With ordinal data, we can use positioning, relative sizes, or colors of increasing brightness or other property. With nominal data, we can use different shapes or distinct colors, but we probably need to use some sort of legend to identify which representation
we use for each value because there will likely be no natural identification available.

When we work with interval data of high dimension and the dimension of the data, or the dimensions of the domain and range of a function together, exceeds the three dimensions we can directly plot, we are likely to need to allow the user some options in viewing the data space. Data exploration can be supported by providing various projections and controls for moving around the data to see the different projections. We are used to thinking of 2D screen projections of 3D space, but we will need to consider projections into 3D space from higher-dimension spaces as well. With the growth of genuine immersive viewing devices for 3D viewing, it may not be necessary to project below 3D space at all in the future.

**Examples**

In this chapter we will describe a number of techniques you can use to work on problems in the sciences. In a sense, presenting a set of techniques may tend to make you focus on the graphical technique rather than the problem you are examining, so you may miss the point of starting with the problem first and finding a graphical or visual representation later. However, our goal is to give you a set of arrows from which you can choose when you shoot at a problem. Learning to analyze the problem and find an appropriate technique will only come with practice.

Our techniques are taken from several years of looking at discussions of visual approaches to the sciences. They are usually not the most sophisticated kinds of images because we are considering work that you can do based on your own programming with a simple graphics API such as OpenGL, rather than with sophisticated scientific visualization tools. If you master these simpler techniques, however, you should have a very good background to understand the way the more complex tools work and to take advantage of them when they are available.

**Diffusion**

Diffusion is a process where a property that is present at one point in a material is spread throughout the material by migrating from its original point to adjacent points over time. When this is modeled for computational purposes, the material is usually divided into a grid of “points” that are usually a unit of area or volume, and the amount of the property at each grid point is given some numeric value. The property might be a quantity of salt dissolved in a unit volume of water, a quantity of heat in a unit volume of a material, or the number of events in a unit area. The process is modeled by assuming that quantities of the property transfer from a given grid point to neighboring grid points proportionately to the amount of property present, either determinately or with some randomness included in the process. This is a very general kind of process that can be applied to a wide range of problems, and in this section we will look at two models built from it.

**Temperatures in a bar**

Let us start with a rectangular bar of some material that is embedded in an insulating medium and has points at which fixed-temperature connectors may be attached. Our goal is to consider the distribution of temperatures throughout the bar over time. We assume that the bar has constant thickness and that the material in the bar is homogeneous throughout its thickness at any point, so we may treat the bar as a 2D entity. The bar may be homogeneous or heterogeneous; the material the bar is made of may have varying thermal conductivity properties; the connectors may be connected or disconnected, and may have time-varying temperatures (but their temperatures are directed by an outside agent and have no relation to the temperatures of the bar). The basic property of the distribution of heat is governed by the heat equation \( \frac{\partial F}{\partial t} = k \frac{\partial^2 F}{\partial x^2} \), the partial differential equation that describes heat transfer, with the constant \( k \) determined by the material in
the bar. This equation says, basically, that at a given point, the rate of change of heat with time is proportional to the gradient (the second derivative) of the heat with space—that is, that the change in heat with time is associated with changes in the heat transfer over space. If the distribution of heat in space, \( \frac{dF}{dx} \), is constant, whatever that constant distribution is, then there is no change in heat over time; it requires changes in the spatial distribution in order to have changes over time. Our goal is to determine approximately at any time the distribution of heat in the bar, given initial conditions and boundary conditions.

We have three basic sets of decisions to make in modeling the distribution of heat in the bar: how to represent the distribution of heat for computational purposes, how to define and model the thermal properties of the bar, and how to display our results in order to communicate the behavior of temperatures in the bar.

For the first decision, we could solve the differential equation directly, or we could model the heat transfer process by modeling the bar as a collection of cells and modeling the heat transferred from one cell to another adjacent cell as being proportional to the heat in the original cell. If two adjacent cells have the same heat, then the effect will be that they exchange the same amount of heat and end up with the same heat each started with.

For the second decision, we will choose to model the thermal behavior by a diffusion process. In the standard case of a cell that is adjacent to four other cells of similar properties, if the proportion of heat energy retained by a cell is \( \alpha \), then the proportion to be transferred to each of the adjoining cells is \((1-\alpha)/4\). We start with an initial condition and update all the cells’ heat from that condition, replace the computed values with any values that are assumed to be constant (e.g. the cells at the positions of fixed-heat points), display the values in the cells at the end of this update, and take the resulting condition as the initial condition for the next round of the simulation. We may make the value of \( \alpha \) a constant for all the cells, or we may set different values of \( \alpha \) for different cells if we want to model a heterogeneous material. For an actual implementation of this problem, we must decide whether the bar itself is homogeneous or heterogeneous, where the connectors will be attached, and what the heat properties of the connectors will be. In the simplest case we would have a homogeneous bar with connectors at fixed points and fixed temperatures, and this case will be discussed below. We could also consider the case of heterogeneous materials and will suggest how a student might address the case of varying temperatures or connections from the connectors.

![Figure 7.2: a simple representation of the temperatures in a bar with fixed-temperature connections](image)

For the third decision, we need to go back to the discussion of visual communication, where this kind of problem was discussed. There we saw that we could represent the temperature at points in the bar either through color or through height, and we discussed how each of these can affect what the viewer sees in the image. In the current case, we will use both color and height to show how the temperature varies in the bar; this seems to be the strongest visual presentation of the
information. The results are shown in Figure 7.2, and the code for this solution is given in the initial chapter of the book.

For the simplest case we saw above, the results are pretty simple to understand; the bar is hottest next to the points where the hot points are attached and is coldest next to the points where the colder points are attached. We do not show a legend that correlates the temperature with the color, and we should do that to make the figure more informative. We also do not show any change in the position or temperature of the attachment points, although there might be changes in the setup of this problem over time. These changes in temperature or position of the attachments could be controlled by user interaction to achieve various effects in the bar, and we invite the student to try some possibilities and see what happens. Specifically, we suggest that you look at a bar with the attached points at one end and with the constant temperatures at those attached points alternating periodically between hot and cold. You should be able to see waves of heat and cold moving down the bar from the attachment points, and you should then try to explain why you see the behavior the display shows.

For the somewhat more complex case of a heterogeneous material we suggested, suppose that we are concerned that some region of the bar not get too hot, at least not very quickly, no matter how much heat is input into it at some fixed hot spots. We know that we can create a composite bar that includes segments with different thermal qualities. Let us assume that the node connections are on one end and let us define a bar that contains an insulating section between the two ends. In this case we will have much less heat transmitted by the insulating material than we would in a bar that had no insulator, and we would see a much slower heating of the material on the side away from the connections. We can model this kind of heterogeneous material more generally and store information on the heat properties of the material in each cell of the bar, adjusting the way we model the spread of heat and getting a good understanding of the way heat moves in such an object.

These examples show the qualitative behavior of the temperature distributions in the bar fairly well, but how accurate are they, and how could we have made them more accurate? We have used a discrete approximation to a continuous process, and we have approximated the physical nature of the heat transfer with a diffusion model, so we know that we do not have an exact solution to the problem. We can compare the results of our model with the results of experimental measurements, however, to allow us to make better approximations of the diffusion values, and we can increase the number of mesh points to reduce the error from the discrete approximation. This will usually allow us to draw good inferences from the model that we can use in understanding the nature of the heat distribution, and in many cases could allow us to design satisfactory materials to use in heat-sensitive environments.

**Spread of disease**

As another application of a diffusion approach, let’s consider the behavior of a communicable disease in a region made up of a number of different communities. We need not try to be specific about the illness, because there are many illnesses that have this kind of behavior, nor will we try to be too specific about the behavior of the communities being modeled. However, we will find that we need to make a number of assumptions to model this in a way that allows us to create simple displays.

The general assumption on which we will build the disease spread model is that the basic mechanism of disease spread is for an infected and a susceptible (uninfected and not immune) person to come in contact with each other. When that happens, we will assume that there is a certain probability that the uninfected person will become infected. Because we cannot model every possible meeting of individuals, we will assume that the number of meetings between two populations is proportional to the product of persons in the populations. We will further assume
that the region is divided into a grid (2D array) of rectangles with one community per rectangle, and that meetings only happen between members of adjoining communities, where adjoining is assumed to mean communities in the same row or column, differing in position by only one index.

As for the disease itself, we will begin by assuming that the initial population in each community is assumed to be entirely susceptible to the disease. We will further assume that the disease is not fatal, and once an individual has recovered from the disease (s)he is permanently immune to it. We assume that an infected person recovers with a probability of $\beta$ (this value also models the duration of the illness), and that a susceptible person becomes ill with a probability of $\alpha$ times the number of possible meetings between that person and an infected person. We will assume that the number of possible meetings is defined as above.

With these assumptions, we derive a model that contains an array of susceptible, infected, and immune persons that represent the number of persons in each community in the 2D grid. For each time step in the simulation, we are able to calculate the number of meetings between infected persons and susceptible persons for each grid point by using the product of susceptible persons at that point and the numbers of infected persons at each neighboring point (including the point itself). We update the number of immune persons by calculating the number recovering and subtracting that number from the infected group, and then we add in the number of new infections from the group itself and from each of the neighboring groups. Finally, we subtract the number of newly-infected persons from the number of susceptible persons. These computations are done with a diffusion-like model for new infections:

\[
\text{infected}[i][j] = \alpha \cdot \text{pop}[i][j] \cdot (m[1][1] \cdot \text{pop}[i][j] + m[0][1] \cdot \text{pop}[i-1][j] + m[2][1] \cdot \text{pop}[i+1][j] + m[1][0] \cdot \text{pop}[i][j-1] + m[1][2] \cdot \text{pop}[i][j+1])
\]

and with appropriate updates for the immune and recovering populations in cell $[i][j]$ as needed.

Once we have the new numbers for the new time step, the display is updated and re-drawn; Figure 7.3 shows a frame from the animation produced for the simulation. The simulation introduces one additional feature by including a region for which there is no contact (we color it blue to simulate a lake); this changes the shape of the spread but does not change the fact that the infection spreads to the entire region.

Figure 7.3: the disease spread model showing the behavior around the barrier region in blue

Note that the model predicts that everyone in each community will eventually become immune to the disease, and thus the disease will spontaneously disappear from the region permanently after the disease has run its course. This is evidently not a reasonable conclusion, but if we examine the
weaknesses in the model we can see how its inaccuracies arise. We assume no births, because a newborn person has not yet acquired an immunity to the disease, and we assume no travel except between neighboring communities, so a new case of the disease would have to arise spontaneously somewhere in the region. So this model is overly simplistic, and yet it helps present the nature of communicable diseases at a simple level that can be refined by making a few additions to the model.

*Function graphing and applications*

We are certainly used to seeing many functions graphed with curves and surfaces in textbooks, so it would not be surprising to find that they are effective tools in understanding problems and finding solutions. In fact, we are so used to seeing problems phrased in terms of their graphs, curves, or surfaces that we may find it difficult to see a problem where these arise. But we will try.

Drawing a graph of a real function of a single variable is very simple. The graph of the function \( f \) is defined to be the set of points \((x, f(x))\) for values of \( x \) in the domain of the function. The graph is drawn by choosing sample values in the domain, calculating the function for each value, and drawing a straight line from one pair to the next. We are familiar with this kind of graphing from elementary mathematics and will not describe it further.

Drawing a graph of a real function of two variables is somewhat more complex, but if the function is reasonably well-behaved, this is straightforward with the aid of computer graphics. The general idea of a graph of a function of two variables presented as a surface is straightforward. For each pair \((x, y)\) in the domain we calculate a value \( z = f(x, y) \), and the set of triples \((x, y, f(x, y))\) is the graph of the function. As we saw above, however, we do not draw every point in the graph; we choose regularly-spaced values in the domain and calculate the function for each value, and join the vertices defined by these points, taken four at a time over the grid in the domain, by two triangles in the surface. Each triangle is planar and can be displayed with your choice of solid color or color determined by a lighting and shading model, depending on the results you want. The end result is described by Figure 7.4 below for a very coarse grid, with the function shown in more detail in Figure 7.5. Note that as we said, the basis for the surface is the set of rectangles in the domain, with each rectangle defining two triangles in the actual surface.

![Figure 7.4: mapping a domain rectangle to a surface rectangle](image)

An example of a straightforward surface graphed in this manner is the “ripple” function that shows rings proceeding outward from a central point, shown in Figure 7.5. We call it “ripple” because it looks like the ripples when a stone is thrown in the water. This is calculated as described above,
and the function being graphed is \( z = \cos(x^2 + y^2 + t) \). Over the square domain \(-5 \leq x \leq 5\) and \(-5 \leq y \leq 5\), a double loop iterates both \( x \) and \( y \) and calculates the value of \( z \) for each pair \((x, y)\). The values can be calculated and stored in a 2D array of \( z \)-values, or we can calculate the values in the domain as we go. We iterate over the domain and draw the triangles needed for the surface. Note that the function in this example contains a parameter \( t \) (think of time), and as this parameter is increased, the value of the argument to the cosine function is also increased. With this parameter increased linearly by adding a constant amount each time the image is re-drawn, the figure is animated and the waves move continually away from the center. The image here is drawn with a large number of triangles and using lighting and surface smoothing so it is difficult to see all the individual pieces, but they are present.

![Figure 7.5: an example of a function surface display](image)

The code for this kind of graphing is pretty straightforward. Assuming that our domain is from \( X_0 \) to \( X_1 \) and from \( Y_0 \) to \( Y_1 \), and that we are taking \( N \) steps in each, we have:

```plaintext
Xstep = (X1-X0)/N; Ystep = (Y1-Y0)/N;
for (x = X0; x += Xstep; x < X1)
    for (y = Y0; y += Ystep; y < Y1) {
        xx = x + Xstep; yy = y + Ystep;
        glBegin(GL_TRIANGLE_STRIP);
        glVertex3f( x,  y, f(x,y)  );
        glVertex3f( x, yy, f(x,yy) );
        glVertex3f(xx,  y, f(xx,y) );
        glVertex3f(xx, yy, f(xx,yy))
        glEnd();
    }
```

Of course there are many ways to make this more efficient, but this will compute and display the basic triangles that make up the surface.

Many problems can be understood in terms of a mathematical function, and this function can often be understood better with the help of its graph. For example, let’s consider the situation where there are various charges on a plane, and we want to be able to place these charges to achieve a certain electrostatic potential at a given point. We would start by realizing that the scalar electrostatic potential at a point \((x, y)\) with charges \( Q_i \) at points \((x_i, y_i)\) is given by Coulomb’s law:
For any set of fixed charges at fixed points, this defines a function of two variables that can be graphed as noted above. This function is fairly simple, but it’s not at all clear from just the equation what the nature of the electrostatic potential is. For a particular configuration with one positive charge and two negative charges at given points, the graph in Figure 7.6 describes the electrostatic potential in a rectangle, as described earlier in the chapter on visual communication. From this graph we see clearly that the potential looks like an elastic sheet with spikes directed upward or downward depending on the sign of the charge. The actual values of the potential at a point could be estimated from the 2D pseudocolor plane if we were to include a scale in the image.

Figure 7.6: the coulombic surface from three point charges (one positive, two negative) in a plane, with both a 3D surface and a planar pseudocolor presentation

If you want to achieve a given electrostatic potential at a given point, you can start with this kind of graph and define a way to select one of the point charges to manipulate it. You can then move the point around, with the mouse, for example, or you can change the amount of charge at that point. Both of these will change the graph, allowing you to see how the potential at your particular point changes. With a little experimentation, then, you will be able to get the appropriate potential at your point of interest, and you can even experiment more to see whether there are other ways that could achieve the same potential in easier ways. Thus the image provides an interactive tool to create the potentials you want at the places you want them.

Another example that can show different behaviors via functions is that of interacting waves. Here we have two (or possibly more, though we will restrict ourselves to two for this discussion) wave functions, and the overall displacement they cause is the result of a sum of the two functions.

There are two kinds of waves we may want to consider: wave trains and waves from a given point. A wave train is described by an equation such as \( f(x,y) = a \sin(bx + cy + d) \) that contains an amplitude \( a \), a frequency and direction determined by \( b \) and \( c \), and a displacement given by \( d \). By incrementing the displacement with each successive re-drawing of the image, we can animate the wave behavior. The behavior of two wave trains is thus given by a function that is the sum of two such equations. In the left-hand side of Figure 7.7 we see the effect of one wave train of relatively high frequency meeting a wave train of about the same amplitude but a lower frequency at an angle of about 120°. It is common to model water waves as a sum of many different wave trains of different angles, frequencies, and amplitudes.
In another example, we consider waves from a given point, which behave much as the example shown in Figure 7.6 above. Here each wave is given by an equation whose general form is extended from the “ripple” example earlier as \[ z = a \cos(b((x - x_0)^2 + (y - y_0)^2)) \]. Thus each wave function is defined by an initial point \((x_0, y_0)\), an amplitude \(a\), and a frequency \(b\). Thus two waves with their own initial points, amplitudes, and frequencies are given by a sum of two of these functions. When two (or more) of these wave functions are added together, they create a complex pattern of interference and reinforcement, as indicated in the right-hand side of Figure 7.7 where the two waves have slightly offset centers and the same amplitudes and frequencies.

![Figure 7.7: two wave trains intersecting at a shallow angle, left, and two circular waves whose origins are offset by \(3\pi/2\), right](image)

**Parametric curves and surfaces**

A parametric curve is given by a function from a line or line segment into 2D or 3D space. This function might be analytic (given by a formula or set of formulas) or it might be an interpolation of data values. The former is probably easier to work with for now, but an example of the latter is found in a later chapter on interpolation and splines.

To consider an example of an analytic curve, let’s consider what we would have in cylindrical coordinates if we increased one variable, the angle, at a slow rate while drawing circles with the other two variables. That is, if we name our parameter \(t\) and used the parametric equations
\[
\begin{align*}
  x &= (a \sin(c t) + b) \cos(t) \\
  y &= (a \sin(c t) + b) \sin(t) \\
  z &= a \cos(c t)
\end{align*}
\]
for real constants \(a, b,\) and \(c\). For the example shown in Figure 7.8, \(a=2.0, b=3.0, c=18.0\).

![Figure 7.8: The toroidal spiral curve](image)
Then the parametric spiral shown moves around a torus as \( t \) takes on the values between 0 and \( 2\pi \). The parametric equations are evaluated at very small steps in the parameter \( t \), giving a sequence of points that are joined by straight lines, yielding the rather smooth display of the curve. This spiral goes around the torus 18 times (\( c=18 \)) while describing a circle with radius 2 and center three units from the origin, as you can see fairly readily from the figure.

Parametric surfaces can be a little more challenging. Here we take a 2D region in the plane and map it into 3D space in various ways. This kind of modeling may need some work to lay out, but you can achieve some very interesting results. In Figure 7.9 we show an example of such a surface; here we took a planar rectangle and divided it into three parts, folding the cross-section into an equilateral triangle. Then we twisted that triangle around \( 4/3 \) times and stretched it around a torus, putting the two ends of the triangular tube back together. The resulting surface is one-sided (you can trace all the surface without crossing any of the triangle edges) and is interesting to hold and manipulate; in the later chapter on hardcopy you can see photographs of the surface that have been created with various 3D hardcopy technologies.

![Figure 7.9: a 3D parametric surface of two variables](image)

To look at this surface in more detail, the domain is the rectangle in parameters \( u \) and \( v \) defined for 
\[-2\pi \leq u \leq 2\pi \text{ and } -2\pi \leq v \leq 2\pi \text{, and given by the equations}
\[
X(u,v) = (4+2\cos(4u/3+v))\cos(u) \\
Y(u,v) = (4+2\cos(4u/3+v))\sin(u) \\
Z(u,v) = 2\sin(4u/3+v)
\]

This should look fairly familiar, because it is much the same as the toroidal spiral curve above. The difference, however, is that for the spiral we stepped the single parameter in small steps, while here we step the parameter \( v \) in small steps (e.g. 100 steps around the torus) while stepping the parameter \( u \) only three times, giving us large spaces between steps and making the cross-section of the surface triangular. This is shown in the layout of the parameter space shown in Figure 7.10; in general, it can be very helpful to lay out the parameter space in this way before going on to define the surface in more detail. In particular, you can see from the parameter space layout that the \( u \)-space is used to create the triangles and the \( v \)-space to create the steps around the torus. You might conjecture what could be produced if you used four steps instead of three in \( u \)-space (did you see a square cross-section?) or another number, but if you should do this, you must be careful to change...
the constant 4/3 in the equations so that you will still get a closed surface.

\[ u \]
\[ v \]
\[ \pi \]
\[ 0 \]
\[ \pi \]
\[ \pi \]
\[ -\pi \]
\[ v \]
\[ -\pi \]
\[ 0 \]

Figure 7.10: the parameter space for the surface shown in Figure 7.9, simplified by showing fewer steps in the \( v \) parameter

If you find that these parametric surfaces are fairly simple, you can test your geometric intuition by mapping your 2D domain into 4D space and seeing what you can produce there. Of course, you probably cannot display the 4D surface directly, so you will want to use various projections of the result into 3D space. There are some classical surfaces of this kind, such as the Klein bottle. A set of parametric equations (in 3-space only) for the Klein bottle are given by

\[
\begin{align*}
    bx &= 6 \cos(u) \times (1 + \sin(u)) \\
    by &= 16 \sin(u) \\
    rad &= 4 \times (1 - \cos(u)/2) \\
    if \ (Pi < u <= 2*Pi) \ X &= bx + rad \times \cos(v + Pi) \\
    else \ X &= bx + rad \times \cos(u) \times \cos(v) \\
    if \ (Pi < u <= 2*Pi) \ Y &= by \\
    else \ Y &= by + rad \times \sin(u) \times \cos(v) \\
    Z &= rad \times \sin(v)
\end{align*}
\]

as translated from a fairly well-known Mathematica™ function. The left-hand image in Figure 7.11 below was obtained by replacing the functions for the torus in Figure 7.9 above with the

Figure 7.11: a 4D parametric surface, the Klein bottle (left) and the structure of the parametric region that defines it (right)
functions given here, and changing the domain to $[0,2\pi]$ instead of $[-\pi,\pi]$. In fact, once you have a good program for parametric surfaces, it should be easy to adapt to different kinds of surfaces.

The actual construction for the Klein bottle is a little more complicated than shown above. The domain for the Klein bottle is a rectangle in 2D space, similar to the domain for the twisted torus above, but the function has one very different treatment for the domain: as shown in the right-hand image of Figure 7.11, with the sides identified as shown, the two sides labeled $b$ are matched as they were when we created the cylinder above, but the two sides labeled $a$ are matched in reverse order. This cannot be done in 3D space, but it can in 4D space, and the result has properties much like a cylinder, but with an embedding that can only be suggested by any 3D projection. The left-hand image of Figure 7.11 shows a standard 3D projection of the Klein bottle, but there are many others that are of interest and that illustrate other properties of the surface; for example, the Klein bottle may be created by gluing two 3D Möbius bands together in 4-space.

**Graphical objects that are the results of limit processes**

Sometimes curves and surfaces arise through processes that are different from the closed-form function graphing we described above. Some of these graphical objects are, in fact, quite surprising, such as some that arise from limit processes.

Two of these limit-process objects might be of particular interest. In the calculus, it is shown that while any function differentiable at a point must be continuous at that point, the converse is not true: you can have continuity without differentiability. There are some examples of continuous but nowhere differentiable functions given in most calculus texts, but it can be difficult to see the actual results of the process. One of these functions is called the blancmange function because its graph—the surface defined by the function—looks somewhat like the very lumpy blancmange pudding often served in England at holidays. This surface is defined recursively as the sum of an increasing number of piecewise bilinear functions that connect the points $(i/2^k,j/2^k,z)$ at level $k$, where $z$ is 0 if $i$ or $j$ is even and $1/2^k$ if $i$ and $j$ are both odd. Because the sum at any point is not larger than $\sum_k 1/2^k$, which is a converging geometric sequence, this sum converges and the surface is well defined. However, within any neighborhood of any point $(x,y)$ there will lie many points $(i/2^k,j/2^k)$ for even values of $i$ and $j$, and at each of these points there is a discontinuity of one of the summands and hence of the function. Because there are discontinuities within any neighborhood of any point, the function cannot be continuous anywhere. The graph of this function is shown in Figure 7.12. For more on the blancmange function, see [Tall]; a very similar surface, called the Takagi fractal curve, is described in Iterated Function Systems terms as an example of midpoint displacement processes in [Pietgen]; see Chapter 16 for more details.
Another surprising object is called the Sierpinski gasket. It can be defined in many different ways but is always a limit of an iterative process. One definition is to take a tetrahedron and to replace it by four tetrahedra, each half the height of the original tetrahedron and occupying one of the corners of the original. This process repeats without limit, giving tetrahedra whose volume approach zero but which occupy positions along the edges of all the possible sub-tetrahedra in the space. Another definition which is somewhat easier to compute is to take the four vertices of a tetrahedron and any collection of points in 3D space. For each point, choose one vertex at random, and move the point one-half of the way towards the vertex. Do this many times, and the limiting positions of the points will lie on the Sierpinski gasket. For more details of this process, see [Pietgen] for the 2D case; the 3D case is a very simple extension. In Figure 7.13 we see the four vertex points of a tetrahedron in red, and points in cyan that have been calculated by starting with 50,000 random points and applying this process.

![Figure 7.13: the Sierpinski attractor](image)

**Scalar fields**

A *scalar field* is a real function of a variable on a domain, which is a very general principle. If the domain is a region in a 1D real space, it is called a 1D scalar field and defines an ordinary function of one variable. A 2D scalar field is a scalar field on a 2D domain, or a real function of two real variables, but this concept is much more general than the kind of surface we discussed above. Many kinds of processes can give rise to such a function, and those processes can produce functions that are far from the continuous surface we were working with. They can even give rise to functions that cannot be expressed in a closed form (that is, in terms of equations). 3D scalar fields are discussed below when we talk about volume data, because they are usually thought of in volume terms.

An interesting example of a 2D scalar field in the digital elevation map (DEM). These maps, available from the USGS and other sources, are 2D grayscale images whose color value represents the elevation of points in a map space. They are, in fact, examples of images colored with pseudocolor data representing elevations. If you know the base and top of the elevation scale for a map, then, you can read the pixels and calculate elevations for each point, giving you a surface grid that is a good representation of the topography of the area. In addition you can often get an aerial photograph, radar scan, or other image of the same area. Figure 7.14 shows the image of a digital elevation map and of an aerial photograph of a portion of the same space, specifically of the San Diego State University campus area, which you will find toward the lower right of Figure 7.15.
When the height data is turned into polygons and they are texture-mapped with the photographs, the results can be strikingly realistic. In Figure 7.15 we see an example of the San Diego area east of downtown, courtesy of Jordan Maynard; the San Diego State University campus can be seen near the lower right of the image. Note that because the image is texture mapped from a photograph, the actual buildings are not seen as they would be in an actual photograph; only the shapes of the buildings on the ground are shown.

There are a number of other places where values are sampled on a domain, and we can think of these as samples of a scalar field. For example, laser range scanning produces a scalar field for distances from a sampling point; as in the terrain map case the sample points may be converted into geometric vertices of polygons of the space being scanned. As an example of this approach, the research into the paleolithic cave art of Cap Blanc used both photographs of the cave, as shown in Figure 7.16.
The cave’s geometry is computed from the results of a laser scan carried out in the cave. The scanner captures the vertical and horizontal offset from the original scanner position at each of many hundred systematically-scanned points. This produces a height field shown as both raw laser scans (left), calculated from the offsets and distance. This is further developed into a mesh (right), both shown in Figure 7.17.

The positions of each point on the mesh is then matched with a position on the photograph that acts as a texture map for the cave wall, allowing the researcher to create an accurate geometric and color model of the cave wall.

This model is then used to understand how the cave might have looked under different lighting conditions, so the researchers can study the effect of different kinds of firelight in various positions. An example of this is shown in Figure 7.18, and you will see how the shape of a horse stands out from the light of a lamp. Initially the lamp is shown at the center bottom of the space. As this lamp is moved from left to right the horse seems to move, showing that the original cave inhabitants may have been able to present the notion of a living horse from the carving.
Simulation of objects and behaviors

In some cases it is possible to represent an object of interest and to create a simulation of the object or its behavior so it can be studied and so the simulation can be compared with the actual object. This kind of simulation can help us understand an object or a phenomenon. In these cases our images help us to visualize the behavior as expressed in the simulation.

This is actually a very broad topic and we cannot go into much detail on simulations here. We will present two simulations of the behavior of an ideal gas and will show how we can not only visualize the behavior but also get data from the simulation to test hypotheses on the behavior. This allows us to get both numerical and visual results, allowing us to verify that the simulation has the right analytic properties while we look at its visual properties. We will also present a simulation of a scientific instrument so we can understand its operation and be better able to interpret its output.

Gas laws and diffusion principles

The behavior of ideal gases under various conditions is a standard part of chemistry and physics studies. Our first simulation will put a number of objects (points, actually, but they represent molecules) into a closed space. Each will be given a random motion by generating a random direction (a vector with random coordinates, normalized to make it a direction) and moving the object a given distance in that direction. This simulates the behavior of a gas under constant temperature. We test for collisions of objects with the walls, and when these are detected the object direction is changed to simulate the object bouncing off the wall. However, we do not detect and account for internal collisions between objects. The behavior of molecules in the space and the numerical output of the simulation are displayed in Figure 7.19, with the pressure (the number of objects hitting a wall is counted at each step) and volume (the product of the dimensions of the box) being displayed, as well as their product. The user can test the simulation by pressing a key and getting the pressure $P$, volume of the box $V$, and the product. This product should be a constant for an ideal gas.
This simulation would be of minimal interest if we left it as described, but we also allow the user to increase or decrease the volume of the box. If this is an accurate simulation of the behavior of an ideal gas, we should have the same product (within statistical limits) as we change the volume of the box, although we need to wait for the gas to expand or contract before we test the model. A student should be able to take the data from several samples at different volumes and perform a statistical test of the hypothesis that the product of the pressure and volume is a constant. The display in Figure 7.19 shows the objects in the box (with one traced through several steps to show the random walk behavior of the gas) as well as the results of several tests.

Another simulation we could create, building on the one above, examines the behavior of gas in a box that is divided by a semipermeable membrane. Such a simulation is shown in Figure 7.20. In such a membrane, if a gas molecule hits the membrane from one side, there is a different probability of it passing through the membrane than if it hits from the opposite side, concentrating that gas somewhat more on the side to which transmission is easier. If there are two different kinds of molecules in the gas and if the objects are treated differently by the membrane, we would expect to see some separation of the two kinds of molecules in the two parts of the space. We
simulate this with particles simulating the gas molecules, as above, and using the same kind of code to detect particles hitting either a wall or the membrane. If a particle hits the membrane, a random number function determines whether the particle passes through the membrane or is reflected back. We are again able to tabulate the state of the particles in the system at any time, but in this case we count the number of particles on each side and calculate the ratio of particles on the left and right sides of the membrane. In the initial state, all the particles are on one side of the membrane; as the simulation runs, we expect to see a steady state reached with different left/right ratios for the two kinds of gas. In the figure we see a snapshot of the space as one of the (red) particles has just passed through the membrane and returned, and we show the display of the tabulation at several times before the screen capture image.

Molecular display

Sometimes we will simply want to see some kind of invisible objects or processes in order to understand our problem better. This has long been the case in chemistry, where the ability to identify and display the structure of molecules has been an important part of chemistry students’ education and where molecular visualization has led to great developments in drug design and other breakthroughs. Molecular visualization is a large topic, however, and many of the visualizations are very complex and require a deep understanding of molecular-level physics. We cannot hope to create this kind of complex work in a beginning computer graphics course, but we can show the beginning of the process.

The traditional start to understanding molecular structure (at least in the author’s student days!) was the spring-and-ball display of a molecule. In this display, each atom is represented by a ball whose color represented a particular kind of atom, and each bond was represented by a spring that connected two balls. This allowed you to assemble a fairly simple molecule and manipulate it by moving it around and seeing it from various angles. We can do at least this much fairly readily, and perhaps we can suggest some ways this could be extended.

To begin, we need to know that the basic geometry of many different kinds of molecules has been determined and is readily available to the public. One of the major information sources is the protein data bank at http://www.rcsb.org (and at several mirror sites), and another is MDL Information Systems (http://www.mdli.com). Molecular descriptions are stored at these (and many more) sites in standard formats, and you can go to these sources, find and download descriptions of the molecules you want to examine, and create displays of these molecules. The descriptions are usually in one of two major formats: .pdb (protein data base) format or .mol (CT files) format. An appendix to these notes gives you the details of the formats, and your instructor can make available some very simple functions to read the basic geometry from them. Check out the “Molecule of the Month” listing from the University of Bristol for some interesting examples! (http://www.bris.ac.uk/Depts/Chemistry/MOTM/motm.htm)

Creating a display from the molecule description is fairly straightforward. Once you have decoded the description file, you will have a list of atom positions and names, and a list of atomic bonds in the molecule. You can then draw the atoms at the positions indicated and draw in the links. It is common to draw the atoms as spheres with colors and sizes that are traditional for each kind of atom; the spheres may be drawn as opaque or partially transparent, and the colors and sizes can be provided in the file reading function. Links are usually drawn as some kind of line. Figure 7.21 shows an couple of examples of simple molecules from the molecule reader provided with these notes and a display function written by the author. Note that the atoms are drawn with a fairly small alpha value so the bonds and other atoms can be seen; note also that in the example from the .mol file, double bonds are shown (these are included in the .mol file format but not the .pdb format). It is straightforward to include various kinds of interaction with these displays, as described in the section of these notes on creating interactive programs.
In more advanced work, it is common to use displays that include additional information, such as displaying the molecule as a smoothed surface around the spheres with colorings that illustrate the electrostatic forces at various places on the surface. This kind of display helps to show how molecular docking would happen by showing the surface shapes as well as the forces that would guide the docking process.

A scientific instrument

The gas chromatograph measures the quantity of different kinds of molecules in a substance as the substance is vaporized and the vapor is driven down a tube. As the molecules pass down the tube, the more massive molecules move more slowly than lighter molecules, and as molecules pass a detector at the end of the tube, the numbers of molecules are recorded in a strip chart. This process produces different profiles for different substances and so is used to identify the components of a material.

In order to understand how the gas chromatograph works, a simulation written by a student, Mike Dibley, shows a group of molecules starting at the left-hand end of a tube. The molecules are of three different types, indicated by colored particles, and they move down the tube at different (and slightly randomized) rates. When they pass the right-hand end of the tube, they are counted and the number of particles at the end is recorded in a strip chart that is presented at the bottom of the screen. Thus the presentation includes a 3D component, the tube, and a 2D component, the strip chart. Three stages of the simulation are shown in Figure 7.22: the image at left shows the simulation when the first set of particles are leaving the tube; the center image when the second set is leaving, and the third when the last set of particles is leaving.
There are a number of interesting simulations that are built from generating large numbers of random occurrences and looking at the long-term behavior of the system. Because random numbers have long been associated with gambling, and because the casino at Monte Carlo is known worldwide, such processes are often called Monte Carlo processes.

These processes can range from the simple to the complex. We have already seen an example of such a process in the gas law and diffusion simulations above, but there are many other kinds of Monte Carlo simulations. In a very simple case, consider a 2D region that is bounded by a complex curve that makes it very difficult to measure the area of the region. If it is possible to determine quickly whether any given point is inside or outside the region, then a Monte Carlo approximation of the region’s area could be made by generating a large number of random points in a known (probably rectangular) area containing the region. One then counts the proportion of the total number of points generated that lie within the region, and takes that proportion of the known area as an estimate of the region’s area. A very similar process operating in 3D space allows you to estimate volumes, and is illustrated graphically by Figure 7.23 which shows 10,000 points in a cube that is two units on a side and contains a number of randomly-placed (and overlapping) spheres. The spheres are rendered in yellow with a low blending value, points lying inside the spheres are colored red, while those lying outside are colored green, so you can get a general idea of the relative proportions of the points. This figure is not oriented towards getting an exact solution, but it will provide a reasonable estimate that could be very helpful if you were trying to explain the concept and process to a layman (or, perhaps, a judge or jury). It is presented by a program that also allows the user to rotate the volume in arbitrary directions to see how the highlighted points lie in the volume.

Figure 7.23  a Monte Carlo estimate of a complex volume

More complex and interesting kinds of Monte Carlo simulations can be found in other areas. For example, in queuing theory and transportation engineering, arrivals are defined to be random with
certain parameters (mean, standard deviation, probability distribution) and service or transit times are also defined randomly. The system being studied is driven by these events, and the nature of the system is then studied through its reaction to these simulations. These are probably beyond the scope of this discussion, but they provide interesting examples of the power of a simple technique and it can often be very enlightening to use computer graphics to display the ongoing state of the system during the simulation. For example, in a traffic simulation, you could use colors to display the traffic flow in the transportation system, with high-attention colors used to show problem areas and low-attention colors used to show smooth flow. The fact that these colors might be red and green, respectively, would be a nice link to the culture of traffic study! And, of course, statistical simulations can be made more and more complex, so that war games, business simulations, and large-scale economic simulations are all in some sense Monte Carlo models.

4D graphing

Dimensions are an interesting question in computer graphics. We now say that graphics is natively 3D and we feel good about that because the world we live in seems 3D to us. Of course, it really isn’t; it really has many more dimensions that we don’t often think about. We think about driving as a 2D process because our streets (even in San Francisco) are laid out in a 2D pattern, but along with our position we also have velocity vectors, fuel levels, temperatures, and a number of other quantities that we must constantly balance while driving. When we start doing computer graphics to solve problems, we often find that our ability to define and view things in three dimensions is too limited to express everything we need. So as we discussed when we talked about visual communication, we also find that we must create models having more than three dimensions for many of our problems.

Volume data

Volume data is data that has one dimension, one real value, at each point in a volume. Extending the notion of a two-dimensional scalar field, this can be thought of as a scalar field on a three-dimensional space, and we will think of the data as coming from a scalar field, or a real-valued function of three variables. We will take two approaches to the display of the field: we will find implicit surfaces in the volume, or we will display the values in a cross-section of the field.

The implicit surfaces we will seek are surfaces made up of the points where the function has a constant value; these are also called isosurfaces in the volume. Finding them can be a hard problem, because volume data is unstructured and it is necessary to identify a structure in the data in order to create displays of these implicit surfaces. This is done in several ways, but the most common is the marching cubes process, where the volume is divided into a number of small cubes, called voxels, and each cube is analyzed to see whether the surface passes through the cube. If it does, then a detailed analysis of the voxel is done to see for which edges the volume crosses the edge; this allows us to see what kind of structure the surface would have on that voxel, and the surface within the voxel is displayed. For a good reference on this process, see Watt & Watt.

The display is not too difficult for this beginning course, but the analysis is very detailed and its code expression would involve more coding than we want to have here. Instead of the marching cube process, we will simply identify those voxels that contain a point whose value is the one defining the implicit surface, and we will display those voxels in a way that implies the surface. The simplest such display is simply to place a small lighted sphere in the cube, as shown in the left-hand image of Figure 7.24. This gives us the general shape of the surface (though with very little detail) with a modest amount of shape from the way the spheres are lighted, and is a way to start the analysis of the scalar field in the cube. This can lead to an exploration of the space by using an interactive technique to change the value that defines the surface. By sweeping the value through a range that covers the volume being studied, we can see the overall shape of the scalar field and get a better understanding of the problem that generates the field.
Another way to understand the nature of the scalar cube is to slice the cube and see the scalar field on the slicing plane, or cross-section of the volume, as shown in the right-hand image of Figure 7.24. This allows us to think of the function as a 2D scalar field and to use whatever kind of mesh and pseudocolor we want on the slicing planes in order to see the field, so in some ways this is a more precise approach to the problem than the implicit surface. It also gives us an understanding of the relationships of the values throughout the space, in contrast to the shape of the level values, so it complements the implicit surface process well. Again, the display can be interactive so the user can explore the space at his or her leisure, and sweeping the planes through the space can give us an overall idea of the nature of the scalar field.

We should note that both the images in Figure 7.24 come from interactive programs that allow the user to sweep the space by increasing or decreasing the value defining the implicit surface, or that by moving move any of the three cutting planes parallel to the axes in order to see the entire shape of the scalar field. These interactive explorations are critical to understanding the data fully and are quite easy to implement. See the chapter on interaction for more details and examples.

The two images in this figure actually represent the same function: the three-variable hyperbolic function \( f(x,y,z) = xyz \). The implicit surface describes the geometry of the subspace where \( f(x,y,z) \) is constant, which is a set of hyperbolas in four of the eight octants of 3D space. The four octants are those in which the signs of the variables are right for the sign of the constant, and the shape in each octant is determined by the constant. On the other hand, with the cross-sections we are looking for ways to represent the values in the 2D spaces that slice through a cube in 3D space through the use of a color ramp. We have used a rapidly repeating color ramp to show the contours of the scalar field, but a single color ramp across all the values in the cube would have been better if we had wanted to be able to read off actual values from the image.

**Vector fields**

We have discussed scalar fields earlier in this chapter, but we can extend the notion of function to include vector-valued functions on 2D or 3D space. These functions are called *vector fields*, and they arise in a number of situations. We should note that not all vector fields are necessarily presented as vectors; any function whose range lies in 2D or 3D space can have its values...
considered as functions, even if that might not be the intent when the function’s context is set. For example, a complex function of a complex variable has a 2D domain and a 2D range because of the identification of complex numbers as pairs of real numbers. Let’s consider some examples of functions that give rise to vector fields.

We already mentioned complex functions of a complex variable, so let’s start there. If we consider the function $w = z^3 + 12z + 2$ described in [Braden], we see by analogy with real functions that as a cubic function, we should expect to have three “roots” of the equation, that is, three points where $w(z) = 0$. In the left-hand image of Figure 7.25, we take advantage of the alternate representation of complex numbers as $re^{i\theta}$ to display the graph of the equation by showing the magnitude of $r$ as a color (in the uniform luminance color ramp defined in the chapter on visual communication) and by showing the direction $\theta$ as a direction vector. The three black points in the image correspond to the three roots we would expect, and we see that we have a smooth vector field across the space, showing that the complex function is smooth (indeed, infinitely differentiable) across its domain. With this display we can get a better understanding of the nature of the function and its behavior.

Another kind of problem generates differential equations that describe the nature of the change in a value across a domain. For example, if we think of fluid flowing across a plane, we see the velocity at each point of the plane; the plane is a 2D domain the velocity at each point is a 2D value. The point is a position, and the fluid flow at that point can be seen as a 2D derivative of the position; to solve the differential equation would be to determine the function that describes the fluid flow. In the right-hand image of Figure 7.25, we see the pair of differential equations

$$\frac{\partial x}{\partial y} = y^2 - 1 \quad \text{and} \quad \frac{\partial y}{\partial x} = x^2 - 1$$ [Buchanan] that describe such a situation, and the image shows the nature of the flow: low speed (magnitude of the velocity) at the four points ($\pm 1, \pm 1$), vortex flow at the two of these points where the signs differ, and source/sink flow at the two points whose signs are the same. As above, we use the color value to indicate the speed of the flow and the vector to show the direction.

**Graphing in higher dimensions**

We can go farther and consider functions on 3D space whose values are also in 3D space. This can challenge our imaginations because we are actually working in six dimensions, but such problems are everywhere around us. So standard examples include electromagnetic and
Consider electromagnetic fields generated by a current moving in a wire. These fields have vector values at every point in space, and so are represented by a function from 3D space to 3D space, and it takes six dimensions to represent this function fully. The function can be shown by sampling the domain on a regular grid and showing the field vector at each point, as shown in Figure 7.26, taken from a student project by Jordan Maynard. Such vector fields are commonly used for many higher-dimensional problems, but they are difficult for a user to understand and there are various other techniques that can be used to show this information.

Another approach to a problem with 3D values in 3D space is to use a set of vectors to see the effects of the function. We saw the use of vectors in a 2D problem in 2D space earlier, but for the 3D problem let’s return to Coulomb’s Law of electrostatic force that we saw in scalar form earlier in this chapter. This law states that the electrostatic force between two particles is given by \( F = kQq/r^2 \), where \( q \) and \( Q \) are the charges on the two particles, \( r \) is the distance between the particles, and \( k \) is an appropriate constant. This force is directed along a vector from one particle to the other, and the value computed above may be taken as the length of that vector.

If we extend this to 3D, we start with a unit positive charge \( Q \) for one particle at an arbitrary point in space and calculate the force between that particle and a set of particles with charges \( q_i \) in space. We get the equation: 
\[
F(x, y, z) = \sum_i kq_iV_i/r_i(x, y, z)^2
\]
for the force vector \( F(x, y, z) \) at any point in 3D space, where \( V_i \) is the vector from the arbitrary point to the \( i^{th} \) particle and \( r_i(x, y, z) \) is the distance between the point \((x, y, z)\) and the \( i^{th} \) particle. In Figure 7.27, we show the way this force
is distributed in 3D space with a collection of cyan vectors in that space; the figure also shows a set of particles with charges of +2 (green) or -1 (red), and the path of a particle of charge +1 that is released at zero velocity within the space.

![3D Coulomb’s law simulation with a 3D vector field](image)

Figure 7.27: the 3D Coulomb’s law simulation with a 3D vector field

There are many opportunities to choose different kinds of displays in these higher-dimensional cases. For example, just above we saw an example of displays of a 2D vector field on a 2D domain that separated magnitude and direction; we could examine a 3D vector field on a 3D domain by using 2D slices of the domain and on each slice displaying the 3D direction and the 1D magnitude of each vector. The opportunity to be creative here is very large.

An example of a field that has a long history of multi-dimensional data is statistics. Here it is rare for a data set to have as few as three variables, so there has been a lot of work with adding extra information to 3D scatterplots to try to see more information and with allowing user exploration of data by selecting which variables or combinations of variables to use. One of the key features of this kind of display is that the viewer needs to be able to move around the space to see the scatterplots from various angles in order to see any structure that might be present. The emphasis on interaction in these notes will help you find ways to let your viewer choose and manipulate the data and space where viewing happens.

**Data-driven graphics**

When we develop graphics based on data rather than theory or principles, we have some new issues that we must consider. Data are not always accurate, are not always directly comparable within a data set (for example, data may be gathered at different times of day or under different conditions), may not be collected at points that make it easy to graph, or may not measure exactly what we would like to present. But we need to deal with the data as it is given to us as we develop images that represent the data.

The visual representation of the data need not be sophisticated. For data with a few dimensions, a scatterplot can be an excellent way to visualize the data. Patterns could be efficiently unveiled by simply drawing each data point as a geometric object in the space determined by one, two or three numeric variables of the data, while its size, shape, color and texture determined by other variables...
of the data. Here we must recognize the importance of making both the position and the representation of the data communicate the meaning of the data to the user. As we discussed in the chapter on visual communication, nominal data can be represented by shape or texture, ordinal data by discrete colors, and interval data by position and continuous color.

Another classical kind of graphing is the line graph. This kind of graphic is very familiar, but usually in its two-dimensional form, plotting an independent variable against a dependent variable. With the tools of computer graphics, this can be much more descriptive. Figure 7.28 illustrates the effectiveness of this kind of graph in plotting the state of the economy (the moving line) against time (the red axis with years marked). The description of this graph by its author, Bernard Pailthorpe, is “The state of the economy in any given year is plotted according to its similarity to three characteristics, or archetypes: boom (green), stagflation (blue), and depression (red). The colors help indicate the position relative to the three axes; the shadow line represents the economy's position relative to the depression and stagflation axes.”

If the data are known to come from a pattern with two independent variables and one dependent variable, we can present it as an approximation of a surface. We may not have a regular distribution of the independent variables on a 2-dimensional domain, so we may need to tesselate the domain and create a flat-shaded polygonal surface to represent the actual data distribution. On the other hand, we may have reason to know that the data come from sampling a known class of surfaces, so we may want to use the data to define the parameters of the actual surface (for example, using a technique such as least-squares parameter fitting) and then draw the computed surface along with the data, shown as points. There is ample opportunity for creative thinking about how you communicate this kind of information to the viewer.

As we noted when we talked about visual communication, it is important that you tell the truth with your images. It is very easy to create smooth surfaces that contain your data and that seem to say that the actual data varies smoothly, but sometimes the data actually contains jumps and other artifacts that simply aren’t smooth, and making a smooth surface implies a continuity of values that is not true. Simplicity and truth are more important than attracting the eye with an incorrectly defined image.

Code examples

We did not want to slow down the general discussion of the kinds of modeling and visualizations one could create to support problem solving by discussing implementation and code questions, but we believe it will be useful to include some discussion of implementation and some sample code.
that was used to implement the models that led to the images in this chapter. In general, the
discussion and the code will not focus on the graphics itself but on the modeling and the
processing to support that modeling. When we want to point out something about the graphics
being used, we may use generic API calls from which it should be relatively easy to infer the
OpenGL that would be equivalent. However, sometimes we may need to use the exact OpenGL to
make the point we want you to see in the discussion.

**Diffusion**

Diffusion processes work on a grid, so the first task is to define an appropriate grid on some 2D
rectangle. This grid needs to be mirrored by an array of real numbers that will hold the values of
the variable on the grid, so we could use declarations such as the following to implement these:

```c
#define LENGTH 50
#define WIDTH 30
float grid[LENGTH][WIDTH]
```

Once these are defined and the grid is initialized, we can think about the code that defines how the
value diffuses in the material. When we do, however, we realize that we will have to use the
values in the grid to calculate new values in the grid, which would instantly lead to errors if we
used the newly-calculated values instead of the original values. So we will need to use a mirror of
the grid to hold the values we are creating, and this must also be defined

```c
float mirror[LENGTH][WIDTH]
```

In general, the processing on the grids will look like calculating weighted sums. While the actual
working of heat depends on the difference between the temperatures of each pair of adjacent cells,
we will calculate the heat based on only the heat in the cells; the working of the processing for the
two cells will take care of the difference. The general calculation takes a sum

\[ \sum_{adjCells} heatAvail \times weight \times temp(cell) \]

where the value of heatAvail is the heat available for the
adjacent cell, which depends on the material, and weight depends on the diffusion model we are
using. At the beginning of the chapter we suggested that the heat we do not keep is shared equally
between the four nearest cells. We thus need to deal with the notion of the available heat for
specific cells. We will use the notion that for any material, there is a proportion of the heat in a cell
that is retained by that cell. We will also note that the weight in the formula is that proportion of
the heat given up that is shared with neighboring cells, or .25 in the case we are discussing. Thus
the equation of heat for any cell becomes the proportion retained by the cell plus the portions
obtained from the adjacent cells. This value is calculated and stored in the mirror array, and at the
end of the computation, the mirror array is copied back into the grid array. In terms of rough
coding, this is given by

```c
for i
  for j {
    mirror[i][j] = prop*grid[i][j];
    mirror[i][j] += .25*(1-prop)*grid[i+1][j];
    mirror[i][j] += .25*(1-prop)*grid[i][j+1];
    mirror[i][j] += .25*(1-prop)*grid[i-1][j];
    mirror[i][j] += .25*(1-prop)*grid[i][j-1];
  }
for i
  for j
    grid[i][j] = mirror[i][j]
```

An alternate, and more flexible, approach would be to define a 3x3 filter (or a different size if you
wish), or array of non-negative values that sums to 1.0, and multiply the grid[][] cells around
the [i][j] position by the values of that filter. This is a classical process for many kinds of
computation and should be in your bag of tricks, if it isn’t so already.
Of course, this does not take into account the behavior at boundary cells, where you will have to change your logic so that heat is not received from the out-of-bounds cells (and where less heat is given away, presumably.) This is left as a logic exercise for the reader. Finally, in case any of the grid points are being held at a fixed temperature (for example, if a hot source or cold sink is present), these points need to be given their fixed values to maintain the fixed situation there.

In case the material is not homogeneous, we must take into account the different coefficients of heat of two adjacent grid cells, and use an appropriate coefficient for the heat transfer. This needs to be symmetric—the value used by one cell with its neighbor must be the same, no matter which individual cell has the higher or lower heat coefficient—and we would suggest that the minimum of the two coefficients would be used. This change could be applied to both homogeneous and non-homogeneous materials without affecting the homogeneous case, of course, so we commend it to the reader. This would, of course, require the addition of an array of these proportions of retained heat (which we can view as values between 0 and 1), so we need to have declared

\[
\text{float prop[LENGTH][WIDTH]}
\]

and instead of working with a fixed value of \text{prop} in the computations, we would use the minimum of the proportions of the original point and new point, so for the first of the adjacent points we would see

\[
.25*(1-\text{min(prop[i][j], prop[i+1][j])}*\text{grid[i+1][j]})
\]

When we have the processing done, we will need to display the results of the computation. To do this, we will use a generic cube as our basis and will scale, translate, and color the cube to represent the grid cell we are drawing. Assuming that we have a cube of unit side that is defined in the first octant and has one vertex at the origin, and assuming that our grid is in the first quadrant and starts at the origin, with each grid length being \text{L}, we have

\[
\begin{align*}
\text{for } i & \\
\text{for } j & \\
\text{set color to colorRamp(grid[i][j])} & \\
\text{set translation(i*L, j*L)} & \\
\text{set scaling(L, L, L*grid[i][j])} & \\
\text{draw cube} &
\end{align*}
\]

If you have defined a reasonable view and set things such as depth buffering so it doesn’t matter in which order you draw the geometry, this should provide a good image. You can then add the user interaction you might want to allow the viewer to examine the situation more accurately. You should also use the idle event to keep updating the image, because unless you started with a completely even temperature across the entire grid, it will take some time for temperatures to even out or to observe other temperature behaviors, depending on your simulation.

Function graphing

This graphing process was introduced briefly above. It is based on building a uniform grid in the domain space, somewhat like the grid used for the diffusion simulation above, and calculating the function values at each point in the grid, then using the grid and the function value to calculate quads and triangles that make up the surface. Here we will give a little more detailed description of the coding process in the following code sketch. The additional detail is included because this is such a fundamental operation that you must be absolutely sure you understand all the details. This sketch is written assuming that all the setup and declarations have been done, and using some meta-API statements to draw triangles and the like:

\[
\begin{align*}
// \text{assume a function calcValue(x,y) that calculates a function for each} \\
// \text{point in the domain; assume further that we are using the same number}
\end{align*}
\]
// of points in each direction in the domain, and save the calculated
// values in a 2D real array values. Assume functions calcXValue and
// calcYValue that compute the x- and y- values of the grid points in the
// domain. Note that we use one more point in each of the directions than
// the number of rectangular regions we will create.
for ( i=0; i<=NPTS; i++ )
    for ( j=0; j<=NPTS; j++ ) {
      x = calcXValue(i); // calculate i-th point in x-direction
      y = calcYValue(j); // calculate j-th point in y-direction
      values[i][j] = calcValue(x,y);
    }

// with the values now calculated, create the surface mesh by creating
// two triangles for each rectangular piece of the grid. We work in the
// counterclockwise direction based on looking down on the grid.
for ( i=0; i<NPTS; i++ )
    for ( j=0; j<NPTS; j++ ) {
      // calculate the x and y coordinates of the corners of the rectangle
      x0 = calcXValue(i);
      x1 = calcXValue(i+1);
      y0 = calcYValue(j);
      y1 = calcYValue(j+1);
      // draw first triangle
      beginTriangle();
      // calculate properties of the triangle such as its normal;
      // this is omitted here
      setPoint(x0,y0,values[i][j]);
      setPoint(x1,y0,values[i+1][j]);
      setPoint(x1,y1,values[i+1][j+1]);
      endTriangle();
      beginTriangle();
      // calculate properties of the triangle
      setPoint(x0,y0,values[i][j]);
      setPoint(x1,y1,values[i+1][j+1]);
      setPoint(x0,y1,values[i][j+1]);
      endTriangle();
    }

**Parametric curves and surfaces**

For parametric curves, it’s straightforward to divide the domain, which will be an interval \([a,b]\) on the real line, by defining a number of points along it at which to evaluate the parametric functions. If you use the parametric equation \(ta+(1-t)b\) to define the interval, the domain consists of all points for all values of \(t\) in \([0,1]\). If you want to divide the interval into \(N\) equal pieces (it’s probably easiest to use equal subdivisions, but it’s not required), then for values of \(i\) between 0 and \(N\), the \(i\)th point is simply \(f(ta+(1-t)b)\) where we assume that the function \(f\) produces points instead of single values, and the curve is drawn by drawing lines between points determined by these values when they are sent to the function that defines the curve. Note that if the parametric curve lies in either 2D or 3D space, the process will be just the same. If you wanted to use unequal subdivisions along the domain interval, the computation of each point in the interval will be different, but drawing the curve by connecting the function values for each point will be the same.

Code to carry this out is quite straightforward, assuming that we have parametric functions \(fx(t)\), \(fy(t)\), and \(fz(t)\) that determine the \(x\), \(y\), and \(z\) coordinates of a point on the curve:

```c
#define START 0.0
#define END 1.0       // could be any values to start, end interval
```
For parametric surfaces the process is a little more complicated, but not much. We assume that the domain of the curve is a rectangle in \( u,v \)-space, with \( a \leq u \leq b \) and \( c \leq v \leq d \), and that we want equal spacing on each of the intervals for the surface. (The spacing may be different for each interval, of course; recall the discussion of the triangular cross-section for the torus.) We then determine the points \((u_i,v_j)\) in the rectangle that represent the \( i^{th} \) point in the \( u \) direction and the \( j^{th} \) point in the \( v \) direction, compute the surface coordinates from these points, and create quads (that will actually be drawn as pairs of triangles) from the four surface points corresponding to the four points that make up a small grid unit in the domain. These triangles may be drawn with any of the properties within the capability of your graphics API. The code for this operation is very much like the code for the triangles in the function surface above, except that all three coordinates of every point on the surface are determined by the function, not just the single coordinate.

**Limit processes**

Limit processes are something of a contradiction for computation, because computation is always finite while limit processes are almost always infinite. However, once we realize that converging limit processes will become arbitrarily close to their final state in a finite amount of time, it is clear that we can compute very good approximations to the final results and show these approximations to the audience.

For the idea of a limit curve or surface, then, we simply take the process out to as many steps as we wish, realizing that more steps can involve more time and perhaps memory. Once we have done the finite calculations, we simply have a function value (or perhaps several functions for a parametric operation) and we can use that in making our image as above.

Other kinds of limit processes, such as the Sierpinski attractor, will involve other operations that we cannot speak of generally. For the Sierpinski case, the process determines the positions for individual points and we simply draw them where they occur. The update for each step is done by an operation like the `idle()` callback discussed earlier and again in more detail in the chapter on event-driven programming. This callback simply needs to include the operations to change each point’s position, and then call a redisplay operation to put it on the screen. As a code sketch, we have the following, where `vertices` is the array of vertices of the tetrahedron:

```plaintext
float points[3][N], vertices[3][4];
// in the display function, we find
beginPoints();
for i = 0 to N
    setPoint(points[0][i], points[1][i], points[2][i]);
endPoints();
// in the idle() function, we find
for i = 0 to N {
    j = (int)random()%4;
    for k = 0 to 2 {
        points[0][k] = (points[0][k] + vertices[0][j])/2.0;
        points[1][k] = (points[1][k] + vertices[1][j])/2.0;
    }
```
Scalar fields

A 1D or 2D scalar field is essentially the same as a function graph or surface except possibly that there is another way to determine the scalar value at each point of the domain. Thus displaying a 1D or 2D scalar field is covered by our discussions above. 3D scalar fields are covered under 4D graphing below.

Representation of objects and behaviors

This section is really about displaying the behavior of simulations and the objects that they use. The examples we give here are fairly simple graphically, and our main challenge is to handle the details of the display. We choose to display a relatively small number of points in the volume because we don’t want to clutter it up and lose track of the fact that individual particles are being tracked in the simulation. We generate the random motion of the particles by adding a small random number to each of the coordinates of each particle in the system, and we pick individual particles to track through time by maintaining a trail of their positions.

Probably the most interesting feature of these two simulations is noting and responding when a particle hits the boundary of a region. In the gas law example we have the particle bounce back into the volume; in the semipermeable membrane example we do the same for the walls, but for the membrane boundary we generate a random number and let that determine whether the particle penetrates or bounces off the membrane. In this sense the simulation has some Monte Carlo aspects, and we describe those below.

We are able to detect a situation when a particle would leave the volume under its normal random motion. In that case, we register a hit by incrementing an accumulator, which is simple; we calculate the point to which the particle would bounce by generating a pure reflection, assuming that particles obey the usual “bounce” rules. The reflection is straightforward, but you probably want to look at the geometry of a bounce situation a bit before you read the brief code below, which assumes that the region is bounded by walls a constant distance from the origin and that the array p[i][j] contains each of the coordinates of each point, in turn:

```c
typedef GLfloat point3[3];
point3 p[NPTS];
if (p[i][j] > bound) {
  p[i][j] = 2.0*bound - p[i][j]; bounce++;
}
if (p[i][j] < -bound) {
  p[i][j] = 2.0*(-bound) - p[i][j]; bounce++;
}
```

Drawing the trails for individual points is straightforward; we simply maintain an array of the last N positions the particles have had and every time we generate a new display, we move each position back one in the array, put the new position at the front, and draw a set of connected line segments between points in the array. This is very helpful in showing the behavior of an individual point, and helps make the simulation display much more understandable to the viewer.

Finally, we gather the various statistics (how many particles are where, how many hit the walls of the volume, etc.) and display them either with the graphics system, as is described in the previous chapter on visual communication, or by printing them to the text console. This is triggered by an event of the programmer’s choice, though we usually use a simple keystroke event.
Molecular display

Our displays of molecules are driven from arrays created by the functions that read the .pdb and .mol file formats, as noted in the earlier discussion. These arrays are of the form

```c
typedef struct atomdata {
    float x, y, z;
    char  name[5];
    int   colindex;
} atomdata;

atomdata atoms[AMAX];
typedef struct bonddata{
    int first, second, bondtype;
} bonddata;
bonddata bonds[BMAX];
```

Here the field `colindex` in the atom structure will be found by looking up the name in the lookup tables described below. This index will then be used to find the color and size that matches the appropriate atom for the display.

The functions read the files and store the results in arrays of these structures, as indicated in the declarations. At the next step, the arrays are traversed and additional information is gotten from lookup tables that hold information such as size and color for individual atoms. The first stage is to look up the atom by its name and return the index of the atom in the tables. After this information is stored in the array, the images are created by traversing the arrays and drawing the molecules with size and color from the tables with this index; a partial sample of these lookup tables is below, with the first table being used to match the name, and the others used to get the color and size associated with the atoms.

```c
char atomNames[ATSIZE][4] = { // lookup name to get index
    "H   ", // Hydrogen
    "He  ", // Helium
    "Li  ", // Lithium
    "Be  ", // Beryllium
    "B   ", // Boron
    "C   ", // Carbon
    "N   ", // Nitrogen
    "O   ", // Oxygen
    ...
};
float atomColors[ATSIZE][4] = { // colors are arbitrary
    {1.0, 1.0, 1.0, 0.8}, // Hydrogen
    {1.0, 1.0, 1.0, 0.8}, // Helium
    {1.0, 1.0, 1.0, 0.8}, // Lithium
    {1.0, 1.0, 1.0, 0.8}, // Beryllium
    {1.0, 1.0, 1.0, 0.8}, // Boron
    {0.0, 1.0, 0.0, 0.8}, // Carbon
    {0.0, 0.0, 1.0, 0.8}, // Nitrogen
    {1.0, 0.0, 0.0, 0.8}, // Oxygen
    ...
};
float atomSizes[ATSIZE] = { // sizes are in angstroms
    0.37, // Hydrogen
    0.50, // Helium
    1.52, // Lithium
    1.11, // Beryllium
    ...
};
```
\{0.88\}, // Boron
\{0.77\}, // Carbon
\{0.70\}, // Nitrogen
\{0.66\}, // Oxygen
...
\};

From this, it is pretty straightforward to draw the atoms, and the bonds are simply drawn as wide lines between the locations of the atoms whose index is defined for each bond. In case a double bond is indicated, two lines are drawn, each slightly offset from the atom center. Because the viewer will doubtless want to examine the structure of the molecule from all viewpoints, it is useful to allow arbitrary rotations. It can also be useful to allow alternate descriptions of the atoms (more or less transparency, larger size to get the space-filling kind of representation, etc.) by providing a user selection through a control panel or a menu.

Monte Carlo modeling

We are using the term Monte Carlo rather loosely to refer to any kind of process based on random values. In that sense, the gas law and semipermeable membrane simulations were Monte Carlo models, and were so noted in their discussion. Sometimes, however, Monte Carlo simulations are taken to mean simulations where events are directly set up by random numbers, and the volume estimation example we give is of this type where the events are placing individual points. It is no trick to figure out whether a given point \((p.x, p.y, p.z)\) lies within a radius \(sphere.r\) of a point \((sphere.x, sphere.y, sphere.x)\), so generating a large number of randomly-placed points and counting those that lie within the sphere’s radius from the sphere’s center for some one or more spheres is not difficult.

Other kinds of Monte Carlo modeling might be a little more challenging. There is a famous experiment that estimates the value of \(\pi\), for example; called the Bouffon needle experiment, it consists of drawing a number of parallel lines on a sheet of paper exactly as far apart as the length of a needle, and then dropping a large number of these needles on the sheet. The proportion of needles that cross one of the lines is an approximation of \(2/\pi\). Simulating this with computer graphics is straightforward: you generate one random point as one end of the needle, generate a random angle (number between 0 and \(2\pi\)) and place a second point one unit distant from the first along that angle, and compare the values of the endpoints to see whether the “needle” crossed the “line.” And, of course, you can draw the needles and the lines as you go so that the viewer can watch the experiment proceed. It might be a pleasant diversion to code this up some rainy day!

4D graphing

A 3D scalar field is a more difficult object to display because it has a 3D domain and 1D range, so we are working at the 4D level. We saw two different ways of handling this case, and of course there would be many more besides these. The code for these two approaches is fairly straightforward. For the isosurface approach, we divide the volume into a number of voxels and evaluate the scalar field function at each of the eight vertices of the voxel. If the function passes through the fixed value that defines the isosurface, that voxel contains part of the surface and so we draw a sphere at that location. We identify whether the function passes through the fixed value by subtracting that value from the value of the function at each vertex and then multiplying these differences; if the sign of that product is negative for any of the edges, the fixed value is crossed. So the code consists of a triple-nested loop with a number of tests, and if the test is positive, we draw a sphere, as follows:
for (i=0; i<XSIZE; i++)
for (j=0; j<YSIZE; j++)
for (k=0; k<ZSIZE; k++) {
    x = XX(i); x1 = XX(i+1);
    y = YY(j); y1 = YY(j+1);
    z = ZZ(k); z1 = ZZ(k+1);
    p1 = f(x, y, z); p2 = f(x, y, z1);
    p3 = f(x1, y, z1); p4 = f(x1, y, z);
    p5 = f(x, y1, z); p6 = f(x, y1, z1);
    p7 = f(x1, y1, z1); p8 = f(x1, y1, z);
    if (((p1-C)*(p2-C)<0.0) || ((p2-C)*(p3-C)<0.0) ||
        ((p3-C)*(p4-C)<0.0) || ((p1-C)*(p4-C)<0.0) ||
        ((p1-C)*(p5-C)<0.0) || ((p2-C)*(p6-C)<0.0) ||
        ((p3-C)*(p7-C)<0.0) || ((p4-C)*(p8-C)<0.0) ||
        ((p5-C)*(p6-C)<0.0) || ((p6-C)*(p7-C)<0.0) ||
        ((p7-C)*(p8-C)<0.0) || ((p5-C)*(p8-C)<0.0)) {
        drawSphere(x, y, z, rad);
    }
}

For the cutting plane display, we simply define a plane in the space and iterate across it in the same way we would for a 2D scalar field. That is, we use the 3D grid in the space put a 2D mesh on the two remaining variables, calculate the value of the function at the midpoint of each rectangle in the mesh, and draw the mesh rectangles in 3D space in a color determined by the function value; because this is so much like the 2D scalar field case, we will not include code for it here. It is straightforward to use interaction techniques to change the coordinate axis across which we are cutting or to change the value on that axis that defines the cut. It would be possible, though a little more complicated, to define other planes to cut the space, but we have not done that.

When we consider 2D vector fields on a 2D domain, we again have four dimensions and we have a choice as to our display. We obviously cannot display an actual 2D vector at each point of the domain, because that would make it essentially impossible to find any single vector. However, if we know that the vector field is relatively smooth, we can display the result vector at selected points in the domain, giving rise to an image with vectors of various lengths and directions. Here we are asking the viewer to understand the results and integrate an image that could have (or could not have, depending on the vector field) overlapping result vectors. This is not a bad approach, but it would take some care to make it work.

We have chosen a slightly different approach in showing the magnitude and direction of the result vectors separately. The magnitude is simply a scalar field, and we have seen how we can readily display it through techniques such as pseudocolor ramps. With the magnitude removed, the vector’s direction can be displayed as a unit vector in the appropriate direction, which shows how the directions are distributed across the domain. Together we get a fairly simple display, but we need to understand that it might not be immediately obvious to a user what the display is saying, because the color and direction are disconnected. These vectors are drawn for the middle point in each 10x10 block of the grid in the scalar field after the scalar field itself has been drawn, and we draw each of the vectors in cyan. We assume that we have calculated the low and high x and y values for each of the grid rectangles, and have calculated the vectors for the midpoint of the rectangle. The code below sketches how we could create the vector portion displays shown for this section.

if ((i%10==5) && (j%10==5)) { // middle of every 10th cell
    x = 0.5*(XX(i)+XX(i+1));
    y = 0.5*(YY(j)+YY(j+1));
    len = 5.0 * sqrt(vector[0]*vector[0]+vector[1]*vector[1]);
    glBegin(GL_LINE_STRIP);
glColor4f(0.0, 1.0, 1.0, 1.0);
glVertex3f(x,y,EPSILON);  //so the vector is above the surface
glVertex3f(x+vector[0]/len, y+vector[1]/len, EPSILON);
glEnd();

Higher dimensional graphing

When we get into any higher-dimensional graphics, we must be very careful to keep the image clear and focused, because it can easily become confused. With more information in the other dimensions that we cannot readily show, you need to plan what data to present and how to present it, or you need to plan how you can allow your viewer to make these choices.

When we talk about vector fields on domains where the dimension is larger than two, we have the problems described above about showing too much information, as well as problems caused by projections hiding some of the information. It is extremely important to allow the viewer to move around the data (or alternately to allow the viewer to move the data volume around) so that it can be seen from all angles. It is also important to show only selected information so that the viewer can get the sense of the data instead of everything at once. For example, when we show vectors in 3D space as in the display of a magnetic field, we cannot show a separate vector for every point in the space because the vectors would cover each other completely. Instead, we would use a technique similar to the vector display above and show only a relatively few vectors in the space. By placing these in a regular pattern in the space, then, we would show the shape of the vector field rather than the complete field. Code for this would be straightforward, as this pseudocode shows:

```
set the color for the vectors
for i
   for j
      for k{
         calculate coordinates of the i,j,k-th point
         calculate vector from magnetic field function for point
         begin lines
         set the point
         set the offset from that point by the vector
         end lines
      }
```

We do not pursue all the possible directions in higher-dimensional graphing here, however. As we saw in the chapter on visual communication, there are ways to use color, shape, and other clues to indicate higher-dimensional information. Some of these work better for nominal data (e.g. shapes), and some for ordinal data (e.g. colors), but considerable creativity can be called for when you need to go very far in this direction.

Summary

In this chapter you have seen techniques for modeling and presenting scientific problems, ranging from surface graphing to tracing the motion of a particle in a force field. This should give you a good set of examples to extend the way you can think about scientific problem solving with computer graphics, and indeed to think about graphical problem solving in general. The ultimate goal of this book is to help you learn to use computer graphics to extend your problem solving skills, and this chapter plays a particularly key role in moving toward that goal.

Credits

A number of colleagues have helped with graphical concepts, with scientific principles, or with models of the scientific issues. We apologize in advance for those we may miss, but we want to
thank Michael J. Bailey of the San Diego Supercomputer Center (SDSC) for many fruitful discussions across all of these topics, and with Kris Stewart of San Diego State University (SDSU), Angela Shiflet of Wofford College, Rozeanne Steckler and Kim Baldridge of SDSC, and Ian Littlewood of California State University Stanislaus for their contributions in one or more of these areas. The students whose work is shown in this chapter, Mike Dibley and Jordan Maynard, are only two of many students at California State University Stanislaus and San Diego State University whose work has helped the author see new ways to understand science through computer graphics.

Questions

1. In many problems in modeling scientific phenomena, we find continuous operations such as derivatives or integrals. However, in many cases we do not have simple equations to represent these operations and must use discrete versions of the operations. Unless we are able to use complex numerical techniques, we must often use simple difference equations to model these operations, but these can introduce errors. Describe some errors that can be caused by using difference equations, and describe ways to reduce difference equation errors.

2. In the chapter on visual communication, we focused quite a bit on the creation and use of color ramps in giving a visual expression of numeric data. Are there any additional issues in the color ramps you might want to use for scientific data graphing? Are there any scientific phenomena that have expressions in colors that could help you choose a color ramp?

Exercises

In the next few exercises, you will be asked to create an example similar to one that is described in this chapter.

3. Find an example of a function of two continuous variables that comes from a problem in the sciences, and create a surface representation of that function.

4. Find an example of forces acting within a space, and create a representation of the trajectories of objects that move in the space based on these forces.

5. Find a height field image and create a surface based on the heights you are able to interpret by the grayscale values from the image.

6. The diffusion model described in the heat transfer example or the disease spread model is quite common in a number of areas of science. For example, there is a model for population growth in a region in which population diffuses outward from the completion of freeways in the region. Find a diffusion-based problem and model it, showing the growth modeled by the diffusion.

7. For the heat diffusion program, consider each cell as a point and create a surface that represents the temperatures in the bar. (To do this, create $x$ and $y$ values for the cell index and let $z$ be the temperature in the cell. Let the color of the vertex be the color of the cell.) Use smooth shading for the triangles or quads you produce. Does the surface image represent a better or worse model for the temperature in the cell?

8. Using relatively simple standard PDB or MOL molecular descriptions and the functions on the CD to read in the molecular data, create representations of the molecules and allow the user to manipulate them with simple controls.
9. Find an example of a function of three continuous variables that comes from a problem in the sciences, and create a volume representation of that function.

**Experiments**

10. For the heat diffusion example in this chapter, we encoded the temperature in both height and color form. Modify the code for this example (from the earlier chapter) to encode temperature in only height, and examine the results. Modify the code to encode temperature in only color, and examine the results. Of the three versions—height and color, height alone, color alone—which is most effective? Why might that be?

11. In the discussion of volume data we showed both an implicit surface and cross sections as ways to understand the nature of the information in the volume. See if you can find another way to represent this information and ask yourself if this gives you a better understanding of the volume.

12. In the discussion of function graphing we discussed a particular function that was everywhere smooth (had continuous derivatives of any degree). Graph a function that has a discontinuity of some sort, and examine the nature of the surface that is produced. Can you find a way to deal with the discontinuity that does not look as though the surface is continuous?

13. A good source of problems in the sciences comes from forces acting on objects. These can be modeled by choosing initial positions and velocities for the objects and then using the classical equation \( f = m{\ddot{a}} \) to calculate the accelerations at a given time, and then updating the velocities with the acceleration and the positions with the velocities. Do this for the \textit{n-body problem}, the problem of determining the behavior of a system of \( n \) objects, each of which has a gravitational attraction for each other object based on the classical gravitation equations. See if your results are realistic, or if they suffer because of being based on difference equations instead of differential equations.

**Projects**

14. (Examine a scientific problem) Find a problem in the sciences that interests you, and develop a question about that problem that should be explained or understood. Create a model for the problem that examines this problem, and write up the problem and why this model is a good representation of the it. Code up the model in OpenGL or another graphics API to create an image that communicates information about the problem, and write up how the image does so and how it provides an understanding of the problem.