CHAPTER 3

INTERACTION AND ANIMATION

We now turn to the development of interactive graphics programs. Interactive computer graphics opens up a myriad of applications, ranging from interactive design of buildings, to control of large systems through graphical interfaces, to virtual reality systems, to computer games.

Our examples in Chapter 2 were all static. We described a scene, sent data to the GPU, and then rendered these data. However, in most real applications we need a dynamic display because the objects we want to display change their positions, colors, or shapes. Our first task will be to introduce a simple method for creating animated applications; that is, applications in which the display changes with time, even without input from the user.

Next, we turn our focus to adding interactivity to WebGL, a task that requires us to look in more detail at how the graphics interact with the browser environment. As part of the development, we will have a preliminary discussion of buffers, a topic we will return to in Chapter 7. We then introduce the variety of devices available for interaction. We consider input devices from two different perspectives: the way that the physical devices can be described by their physical properties, and the way that these devices appear to the application program. We then consider client–server networks and client–server graphics. Finally, we use these ideas to develop event-driven input for our graphics programs.

3.1 ANIMATION

Our examples in Chapter 2 were all static; we rendered the scene once and did nothing else. Now suppose that we want to change what we see. For example, suppose that we display a simple object such as a square and we want to rotate this square at a constant rate. A simple but slow and inelegant approach would be to have our application generate new vertex data periodically, send these data to the GPU, and do another render each time that we send new data. This approach would negate many of the advances using shaders, because it would have a potential bottleneck due to the repetitive sending of data from the CPU to the GPU. We can do much better if we start thinking in terms of a recursive rendering process where the render function can

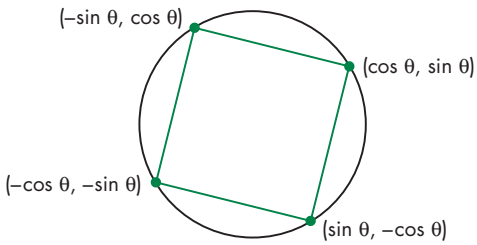

FIGURE 3.1 Square constructed from four points on a circle.

call itself. We will illustrate various options using a simple program that produces a rotating square.

3.1.1 The Rotating Square

Consider the two-dimensional point

$$x = \cos\theta \qquad y = \sin\theta.$$

This point lies on a unit circle regardless of the value of θ. The three points $(-\sin\theta, \cos\theta)$, $(-\cos\theta, -\sin\theta)$, and $(\sin\theta, -\cos\theta)$ also lie on the unit circle. These four points are equidistant along the circumference of the circle, as shown in Figure 3.1. Thus, if we connect the points to form a polygon, we will have a square centered at the origin whose sides are of length $\sqrt{2}$.

We can start with $\theta = 0$, which gives us the four vertices $(0, 1)$, $(1, 0)$, $(-1, 0)$ and $(0, -1)$. We can send these vertices to the GPU by first setting up an array

```
var vertices = [
  vec2(0, 1),
  vec2(1, 0),
  vec2(-1, 0),
  vec2(0, -1)
];
```

and then sending the array

```
var bufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);
```

We can render these data using a render function as in Chapter 2

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

and the simple pass-through shaders we used in Chapter 2.

Now suppose that we want to display the square with a different value of θ. We could compute new vertices with positions determined by the two equations we started with for a general θ and then send these vertices to the GPU, followed by another rendering. If we want to see the square rotating, we could put the code in a loop that increments θ by a fixed amount each time. But such a strategy would be extremely inefficient. Not only would we be sending vertices to the GPU repeatedly, something that would be far more problematic if we were to replace the square by an object with hundreds or thousands of vertices, but we would also be doing the trigonometric calculations in the CPU rather than the GPU where these calculations could be done much faster.

A better solution is to send the original data to the GPU as we did initially and then alter θ in the render function and send the new θ to the GPU. In order to transfer data from the CPU to variables in the shader, we must introduce a new type of shader variable. In a given application, a variable may change in a variety of ways. When we send vertex attributes to a shader, these attributes can be different for each vertex in a primitive. We may also want parameters that will remain the same for all vertices in a primitive or equivalently for all the vertices that are displayed when we execute a function such as `gl.drawArrays`. Such variables are called **uniform qualified variables**.

Consider the vertex shader

```
attribute vec4 vPosition;
uniform float theta;

void main()
{
  gl_Position.x = -sin(theta) * vPosition.x + cos(theta) * vPosition.y;
  gl_Position.y =  sin(theta) * vPosition.y + cos(theta) * vPosition.x;
  gl_Position.z = 0.0;
  gl_Position.w = 1.0;
}
```

The variable `theta` has the qualifier `uniform` so the shader expects its value to be provided by the application. With this value, the shader outputs a vertex position that is rotated by θ.

In order to get a value of θ to the shader, we must perform two steps. First, we must provide a link between `theta` in the shader and a variable in the application. Second, we must send the value from the application to the shader.

Suppose that we use a variable also named `theta` in the application[1]

```
var theta = 0.0;
```

1. We could use any valid name, but we will use the same name in our examples to make it easier to recognize the tie between a particular shader variable and the corresponding application variable.

When the shaders and application are compiled and linked by `initShaders`, tables are created that we can query to make the necessary correspondence using the function `gl.getUniformLocation`, as in the code

```
var thetaLoc = gl.getUniformLocation(program, "theta");
```

Note that this function is analogous to the way we linked attributes in the vertex shader with variables in the application. We can then send the value of `theta` from the application to the shader by

```
gl.uniform1f(thetaLoc, theta);
```

There are multiple forms of `gl.uniform` corresponding to the types of values we are sending—scalars, vectors, or matrices—and to whether we are sending the values or pointers to the values. Here, the `1f` indicates that we are sending the value of a floating-point variable. We will see other forms starting in Chapter 4, where we will be sending vectors and matrices to shaders. Although our example sends data to a vertex shader, we can also use uniforms to send data to fragment shaders.

Returning to our example, we send new values of `theta` to the vertex shader in the `render` function

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  theta += 0.1;
  gl.uniform1f(thetaLoc, theta);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);

  render();
}
```

which increases θ, renders, and calls itself. Unfortunately, this approach will not quite work. In fact, we will see only the initial display of the square. To fix the problem, we need to examine how and when the display is changed.

3.1.2 The Display Process

Before proceeding, it would be helpful to overview how the browser and the window system interact with a physical display device. Consider a typical flat-panel display and a static image that comprises multiple windows. This image is stored as pixels in a buffer maintained by the window system and is periodically and automatically redrawn on the display. The **refresh** or **repaint** operation draws a new image (or frame), at a rate of approximately 60 frames per second (fps) (or 60 hertz or 60 Hz). The actual rate is set by the window system, and you may be able to adjust it.

Historically, the frame rate was tied to the alternating current (AC) power transmission frequency of 60 Hz in North America and 50 Hz in Europe. The refresh was required by the short persistence of the phosphors in CRTs. Although now CRT monitors are no longer the dominant technology and the electronics are no longer coupled to the line frequency, there is still a minimum rate for each display technology that

must be high enough to avoid visual artifacts (most noticeably flicker, if the rate is too low). From our perspective, this process is not synchronized with the execution of our program and usually we do not need to worry about it. As we noted in Chapter 1, the redraw can be **progressive** and redraw the entire display each time, or it can be interlaced, redrawing odd and even lines on alternate frames.

Consider a browser window on the display. Although this window is being redrawn by the display process, its contents are unchanged unless some action takes place that changes pixels in the display buffer. The action (or event) can be caused by some action on the part of the user, such as clicking a mouse button or pressing a key, or the action can be caused by an event such as a new frame in a video needing to be displayed. We will discuss events and event processing in detail later in this chapter, but for now we should note that the browser runs asynchronously, executing one piece of code until some event interrupts the flow or the code runs to completion, in which case the browser waits for another event.

Now suppose that the browser window is a WebGL window. As we saw from our examples, we input a series of JavaScript files including the file with the application. The **onload** event starts the execution of our application with the `init` function. In our applications so far, the execution ended in the `render` function. That invokes the `gl.drawArrays` function. At this point, execution of our code is complete and the results are displayed. However, the code for rendering the square,

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  theta += 0.1;
  gl.uniform1f(thetaLoc, theta);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);

  render();
}
```

has a fundamental difference; namely, that it calls itself, and this recursion puts the drawing inside an infinite loop. Thus, we cannot reach the end of our code and we never see changes to the display. Nevertheless, the executions of `gl.drawArrays` cause changes in another buffer.

3.1.3 Double Buffering

Suppose that the color buffer in the framebuffer is a single buffer that holds the colored pixels produced by our application. Then each time the browser repaints the display, we would see its present contents. If the objects we are rendering are not changing, then, other than possibly seeing a little flicker if the redraw rate is slow or noticing the slight shifting of the image in an interlaced display, we would see an unchanging image in the window.

However, if we change the contents of the framebuffer during a refresh, we may see undesirable artifacts of how we generate the display. In addition, if we are rendering more geometry than can be rendered in a single refresh cycle, we will see

different parts of objects on successive refreshes. If an object is moving, its image may be distorted on the display.

Consider what might happen with the repetitive clearing and redrawing of an area of the screen, as we are attempting in our rotating-square program. Even though the square is a simple object and is easily rendered in a single refresh cycle, there is no coupling between when new squares are drawn into the framebuffer and when the framebuffer is redisplayed by the hardware. Thus, depending on exactly when the framebuffer is displayed, only part of the square may be in the buffer. This model is known as **single buffering** because there is only one buffer—the color buffer in the framebuffer—for rendering and display.

Double buffering provides a solution to these problems. Suppose that we have two color buffers at our disposal, conventionally called the front and back buffers. The **front buffer** is always the one displayed, whereas the **back buffer** is the one into which we draw. WebGL requires double buffering. A typical rendering starts with a clearing of the back buffer, rendering into the back buffer, and finishing with a buffer swap. The remaining issue is how and when the buffer swap is triggered. In the example we started with, in which we tried to use a recursive call to the render function, we failed to provide a buffer swap so we would not see a change in the display. We will examine two approaches: using timers and using the function `requestAnimFrame`.

Note that double buffering does not solve all the problems that we encounter with animated displays. If the display is complex, we still may need multiple frames to draw the image into the framebuffer. Double buffering does not speed up this process; it only ensures that we never see a partial display. However, we are often able to have visibly acceptable displays at rates as low as 10 to 20 frames per second if we use double buffering.

3.1.4 Using a Timer

One way to generate a buffer swap and to control the rate at which the display is repainted is to use the `setInterval` method to call the render function repeatedly after a specified number of milliseconds. Thus, if we replace the execution of `render` at the end of `init` with

```
setInterval(render, 16);
```

`render` will be called after 16 milliseconds (when the timer interval has elapsed), or about 60 times per second. An interval of 0 milliseconds will cause the render function to be executed as fast as possible. Each time the time-out function completes, it forces the buffers to be swapped, and thus we get an updated display.

Suppose that you want to check if indeed the interval between renderings is the interval specified in `setInterval`, or perhaps to check how long it takes to execute some code. We can measure times in milliseconds by adding a timer to our code using the `Date` object. The `getTime` method returns the number of milliseconds since midnight GMT on January 1, 1970. Here is a simple use of a timer to output the time between renderings. Before we render, we save an initial time

```
var t1, t2;

var date = new Date;
t1 = date.getTime();
```

and then in `render`

```
t2 = date.getTime();
console.log(t2 - t1);
t1 = t2;
```

3.1.5 Using `requestAnimFrame`

Because `setInterval` and related JavaScript functions, such as `setTimeout`, are independent of the browser, it can be difficult to get a smooth animation. One solution to this problem is to use the function `requestAnimFrame`, which is supported by most browsers. Consider the simple rendering function

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  theta += 0.1;
  gl.uniform1f(thetaLoc, theta);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
  requestAnimFrame(render);
}
```

The function `requestAnimFrame` requests the browser to display the rendering the next time it wants to refresh the display and then call the render function recursively. For a simple display, such as our square, we will get a smooth display at about 60 fps.

Finally, we can use `requestAnimFrame` within `setInterval` to get a different frame rate, as in the render function, of about 10 fps:

```
function render()
{
  setTimeout(function() {
    requestAnimFrame(render);
    gl.clear(gl.COLOR_BUFFER_BIT);
    theta += 0.1;
    gl.uniform1f(thetaLoc, theta);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
  }, 100);
}
```

Although this solution should work pretty well for most simple animations, let's take a somewhat different look at the problem by focusing on the framebuffer and, at least initially, not worrying about the browser, the window system, or anything else going on.

3.2 INTERACTION

One of the most important advances in computer technology was enabling users to interact with computer displays. More than any other event, Ivan Sutherland's Sketchpad project launched the present era of *interactive* computer graphics. The basic paradigm that he introduced is deceptively simple. The user sees an image on the display. She reacts to this image by means of an interactive device, such as a mouse. The image changes in response to her input. She reacts to this change, and so on. Whether we are writing programs using the tools available in a modern window system or using the human–computer interface in an interactive museum exhibit, we are making use of this paradigm.

In the 50 years since Sutherland's work, there have been many advances in both hardware and software, but the viewpoint and ideas that he introduced still dominate interactive computer graphics. These influences range from how we conceptualize the human–computer interface to how we can employ graphical data structures that allow for efficient implementations.

In this chapter, we take an approach slightly different from that in the rest of the book. Although rendering is the prime concern of most modern APIs, interactivity is an important component of most applications. OpenGL, and thus WebGL, do not support interaction directly. The major reason for this omission is that the system architects who designed OpenGL wanted to increase its portability by allowing the system to work in a variety of environments. Consequently, window and input functions were left out of the API. Although this decision makes renderers portable, it makes discussions of interaction that do not include specifics of the window system more difficult. In addition, because any application program must have at least a minimal interface to the window environment, we cannot entirely avoid such issues if we want to write complete, nontrivial programs. If interaction is omitted from the API, the application programmer is forced to worry about the often arcane details of her particular environment.

Nevertheless, it is hard to imagine any application that does not involve some sort of interaction, whether it is as simple as entering data or something more complex that uses gestures with multiple fingers on a touch screen. For desktop OpenGL, there are some toolkits that provide an API that supports operations common to all window systems, such as opening windows and getting data from a mouse or keyboard. Such toolkits require recompilation of an application to run with a different window system, but the application source code need not be changed.

There is much greater compatibility across platforms with WebGL. Because WebGL is contained within HTML5, we can employ a variety of tools and make use of a variety of packages, all of which will run on any system that supports WebGL. We will follow the same approach as in Chapter 2 and focus on creating interactive applications with JavaScript rather than using a higher-level package. We believe that this approach will make it clearer how interactivity works and will be more robust because it will not make use of software packages that are under continual change.

Before proceeding, we want to lessen some of the confusion with regard to multiple ways the terms *window* and *window system* are used. We use the term *window system*, as we did in Chapter 2, to include the total environment provided by systems

such as Linux using the X Window System, all the versions of Microsoft Windows, and Mac OS X. When you open a window in your browser, this window is also a window within the larger operating system, although one with special properties that allow it to display information specified, for example, by files written in HTML. When we run a WebGL program through our browser, the window it runs in is under control of the browser, which itself is under control of the local windowing system. Matters can get more complex since our interaction with the browser window might cause more windows to appear, such as pop-ups, or windows might be destroyed. All these actions require complex interactions among operating systems, browsers, application programs, and various buffers. Any detailed discussion will take us away from our desire to stay close to graphics.

Without getting into detail about the interaction among the various entities, we can work with the following high-level model: The WebGL application programs that we develop will render into a window opened by a browser that supports WebGL.

We start by describing several interactive devices and the variety of ways that we can interact with them. Then we put these devices in the setting of a client–server network and introduce an API for minimal interaction. Finally, we generate sample programs.

3.3 INPUT DEVICES

We can think about input devices in two distinct ways. The obvious one is to look at them as physical devices, such as a keyboard or a mouse, and to discuss how they work. Certainly, we need to know something about the physical properties of our input devices, so such a discussion is necessary if we are to obtain a full understanding of input. However, from the perspective of an application programmer, we should not need to know the details of a particular physical device to write an application program. Rather, we prefer to treat input devices as *logical* devices whose properties are specified in terms of what they do from the perspective of the application program. A **logical device** is characterized by its high-level interface with the application program rather than by its physical characteristics. Logical devices are familiar to all writers of high-level programs.

Consider this fragment of C++ code,

```
int x;
cin >> x;
cout << x;
```

in which we read and then write an integer. Obviously, we could have just as easily written this fragment in C or Java. Although we would typically run the program on a workstation, entering the data from a keyboard and seeing the output on the display, we could also use mechanisms such as redirection to have the input come from the output from another program or have the output placed in a disk file. Even if we use the keyboard and display, the use of the default input and output streams `cin` and `cout` requires no knowledge of the properties of the physical devices, such as the keyboard codes or the resolution of the display. Rather, `cin` and `cout` are logical

functions that are defined by how they handle input or output character strings from the perspective of the C++ program. More generally, data input and output in C are done through functions such as `printf`, `scanf`, `getchar`, and `putchar`, whose arguments use the standard C data types, and through input (`cin`) and output (`cout`) streams in C++.

When we output a string of characters using `printf` or `cout`, the physical device on which the output appears could be a printer, a terminal, or a disk file. The output could even be the input to another program. The details of the format required by the destination device are of minor concern to the writer of the application program.

In computer graphics, the use of logical devices is somewhat more complex because the forms that input can take are more varied than the strings of bits or characters to which we are usually restricted in nongraphical applications. For example, we can use the mouse—a physical device—to select a location on the screen or to indicate which item in a menu we wish to select. These are two different logical functions. In the first case, an (x, y) pair (in some coordinate system) is returned to the user program; in the second, the application program may receive an integer identifier that corresponds to an entry in the menu. The separation of physical from logical devices not only allows us to use the same physical device in multiple, markedly different, logical ways but also allows the same program to work without modification if the mouse is replaced by another physical device, such as a data tablet or trackball.

There is a second issue that arises from our code fragment; namely, how and when does the input data get to a program variable and, likewise, how does the data from a program variable get to a display device. These issues relate to interaction between the physical devices and the operating system and will also be examined later.

3.4 PHYSICAL INPUT DEVICES

From the physical perspective, each input device has properties that make it more suitable for certain tasks than for others. We take the view used in most of the workstation literature that there are two primary types of physical devices: pointing devices and keyboard devices. The **pointing device** allows the user to indicate a position on a display and almost always incorporates one or more buttons to allow the user to send signals or interrupts to the computer. The **keyboard device** is almost always a physical keyboard but can be generalized to include any device that returns character codes. For example, a tablet computer uses recognition software to decode the user's writing with a stylus but in the end produces character codes identical to those of the standard keyboard.

3.4.1 Keyboard Codes

For many years, the standard code for representing characters was the American Standard Code for Information Interchange (ASCII), which assigns a single byte to each character. ASCII used only the first 127 codes and was later expanded to a larger 8-bit character set known as Latin 1 that can represent most European languages. However, 8 bits is insufficient to support other languages that are needed for a true worldwide Internet. Consequently, Unicode was developed as 16-bit (2-byte) code

that is rich enough to support virtually all languages and is the standard for all modern browsers. Because Latin 1 is a superset of ASCII and Unicode is a superset of Latin 1, code developed based on earlier byte-oriented character sets should work in any browser.

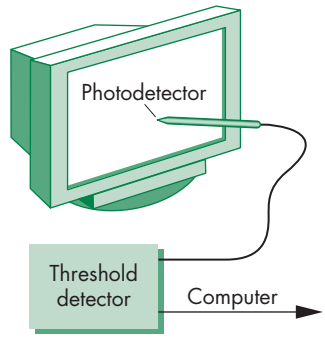

FIGURE 3.2 Light pen.

3.4.2 The Light Pen

The **light pen** has a long history in computer graphics. It was the device used in Sutherland's original Sketchpad. The light pen contains a light-sensing device, such as a photocell (Figure 3.2). If the light pen is positioned on the face of the CRT at a location opposite where the electron beam strikes the phosphor, the light emitted exceeds a threshold in the photodetector and a signal is sent to the computer. Because each redisplay of the framebuffer starts at a precise time, we can use the time at which this signal occurs to determine a position on the CRT screen (see Exercise 3.14). The light pen was originally used on random scan devices so the time of the interrupt could easily be matched to a piece of code in the display list, thus making the light pen ideal for selecting application-defined objects. With raster scan devices, the position on the display can be determined by the time the scan begins and the time it takes to scan each line. Hence, we have a direct-positioning device.

The light pen has some deficiencies, including its cost and the difficulty of obtaining a position that corresponds to a dark area of the screen. For all practical purposes, the light pen has been superseded by the mouse and track pad. However, tablet PCs are used in a manner that mimics how the light pen was used originally: The user has a stylus with which she can move randomly about the tablet (display) surface.

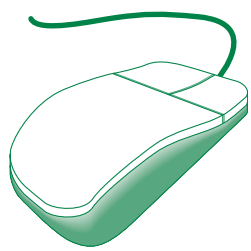

FIGURE 3.3 Mouse.

3.4.3 The Mouse and the Trackball

The mouse (Figure 3.3) and trackball (Figure 3.4) are similar in use and often in construction as well. When turned over, a typical mechanical mouse looks like a trackball. In both devices, the motion of the ball is converted to signals sent back to the computer by pairs of encoders inside the device that are turned by the motion of the ball. The encoders measure motion in two orthogonal directions. Note that the wheel found on many recent mice acts as an independent one-dimensional mouse-like device.

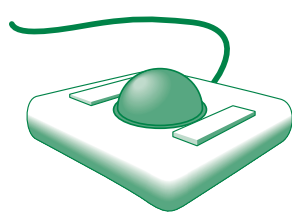

FIGURE 3.4 Trackball.

There are many variants of these devices. Some use optical detectors rather than mechanical detectors to measure motion. Small trackballs are popular with portable computers because they can be incorporated directly into the keyboard. There are also various pressure-sensitive devices used in keyboards that perform similar functions to the mouse and trackball but that do not move; their encoders measure the pressure exerted on a small knob that often is located between two keys in the middle of the keyboard.

We can view the output of the mouse or trackball as two independent values provided by the device. These values can be considered as positions and converted—either within the graphics system or by the user program—to a two-dimensional location in either screen or world coordinates. If it is configured in this manner,

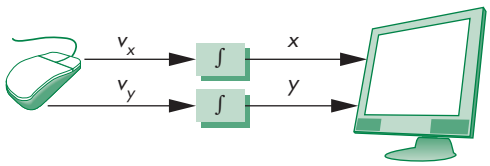

FIGURE 3.5 Cursor positioning.

we can use the device to position a marker (cursor) automatically on the display; however, we rarely use these devices in this direct manner.

It is not necessary that the outputs of the mouse or trackball encoders be interpreted as positions. Instead, either the device driver or a user program can interpret the information from the encoder as two independent velocities (see Exercise 3.4). The computer can then integrate these values to obtain a two-dimensional position. Thus, as a mouse moves across a surface, the integrals of the velocities yield x, y values that can be converted to indicate the position for a cursor on the screen, as shown in Figure 3.5. By interpreting the distance traveled by the ball as a velocity, we can use the device as a variable-sensitivity input device. Small deviations from rest cause slow or small changes; large deviations cause rapid or large changes. With either device, if the ball does not rotate, then there is no change in the integrals and a cursor tracking the position of the mouse will not move. In this mode, these devices are **relative-positioning** devices because changes in the position of the ball yield a position in the user program; the absolute location of the ball (or the mouse) is not used by the application program.

Relative positioning, as provided by a mouse or trackball, is not always desirable. In particular, these devices are not suitable for an operation such as tracing a diagram. If, while the user is attempting to follow a curve on the screen with a mouse, she lifts and moves the mouse, the absolute position on the curve being traced is lost.

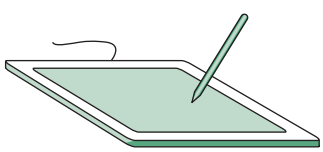

FIGURE 3.6 Data tablet.

3.4.4 Data Tablets, Touch Pads, and Touch Screens

Data tablets (or just **tablets**) provide absolute positioning. A typical data tablet (Figure 3.6) has rows and columns of wires embedded under its surface. The position of the stylus is determined through electromagnetic interactions between signals traveling through the wires and sensors in the stylus. Touch-sensitive transparent screens that can be placed over the face of a CRT have many of the same properties as the data tablet. Small, rectangular, pressure-sensitive **touch pads** are embedded in the keyboards of most portable computers. These touch pads can be configured as either relative- or absolute-positioning devices. Some are capable of detecting simultaneous motion input from multiple fingers touching different spots on the pad and can use this information to enable more complex behaviors.

Tablet devices and smart phones are characterized by **touch screens**. A touch screen is both a display and an input device. Positions are detected by changes in pressure on the display's surface, which might be initiated by a single or multiple pressure points, engaged by fingers or perhaps a stylus. Since the input device is overlaid on the displayed image, the user can interact directly with the displayed

image. Thus, if we display a button, the user can push it. If we display the image of an object, the user can move it around the screen by touching the object and then moving her finger or stylus.

Additionally, most touch devices are capable of tracking multiple pressure points that allows **gestural input**. For example, a common gesture is a "pinch," using two fingers to initially make contact with the touch device at two separate locations and drawing the two fingers together to the same point on the touch device.

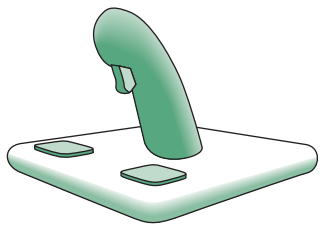

FIGURE 3.7 Joystick.

3.4.5 The Joystick

One other device, the **joystick** (Figure 3.7), is particularly worthy of mention. The motion of the stick in two orthogonal directions is encoded, interpreted as two velocities, and integrated to identify a screen location. The integration implies that if the stick is left in its resting position, there is no change in the cursor position, and that the farther the stick is moved from its resting position, the faster the screen location changes. Thus, the joystick is a variable-sensitivity device. The other advantage of the joystick is that the device can be constructed with mechanical elements, such as springs and dampers, that give resistance to a user who is pushing the stick. Such a mechanical feel, which is not possible with the other devices, makes the joystick well suited for applications such as flight simulators and game controllers.

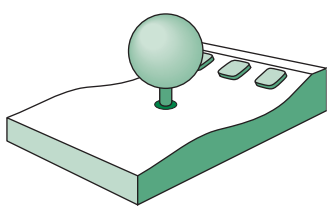

FIGURE 3.8 Spaceball.

3.4.6 Multidimensional Input Devices

For three-dimensional graphics, we might prefer to use three-dimensional input devices. Although various such devices are available, none have yet won the widespread acceptance of the popular two-dimensional input devices. A **spaceball** looks like a joystick with a ball on the end of the stick (Figure 3.8); however, the stick does not move. Rather, pressure sensors in the ball measure the forces applied by the user. The spaceball can measure not only the three direct forces (up–down, front–back, left–right) but also three independent twists. The device measures six independent values and thus has six **degrees of freedom**. Such an input device could be used, for example, both to position and to orient a camera.

Other three-dimensional devices, such as laser scanners, measure three-dimensional positions directly. Numerous tracking systems used in virtual reality applications sense the position of the user. Virtual reality and robotics applications often need more degrees of freedom than the two to six provided by the devices that we have described. Devices such as data gloves can sense motion of various parts of the human body, thus providing many additional input signals. Recently, in addition to being wireless, input devices such as Nintendo's Wii incorporate gyroscopic sensing of position and orientation.

Many of the devices becoming available can take advantage of the enormous amount of computing power that can be used to drive them. For example, motion-capture (**mocap**) systems use arrays of standard digital cameras placed around an environment to capture reflected lights from small spherical dots that can be placed on humans at crucial locations, such as the arm and leg joints. In a typical system, eight cameras will see the environment and capture the location of the dots at high frame rates, producing large volumes of data. The computer, often just a standard PC,

can process the two-dimensional pictures of the dots to determine, in each frame, where in three-dimensional space each dot must be located to have produced the captured data.

3.4.7 Logical Devices

We can now return to looking at input from inside the application program; that is, from the logical point of view. Two major characteristics describe the logical behavior of an input device: (1) the measurements that the device returns to the user program, and (2) the time when the device returns those measurements.

Some earlier APIs defined six classes of logical input devices. Because input in a modern window system cannot always be disassociated completely from the properties of the physical devices, modern systems no longer take this approach. Nevertheless, we describe the six classes briefly because they illustrate the variety of input forms available to a developer of graphical applications. We will see how WebGL can provide the functionality of each of these classes.

1. **String** A string device is a logical device that provides ASCII strings to the user program. This logical device is usually implemented by means of a physical keyboard. In this case, the terminology is consistent with that used in most window systems, which usually do not distinguish between the logical string device and a physical keyboard.
2. **Locator** A locator device provides a position in world coordinates to the user program. It is usually implemented by means of a pointing device, such as a mouse or a trackball. In WebGL, we usually use the pointing device in this manner, although we have to do the conversion from screen coordinates to world coordinates within our own programs.
3. **Pick** A pick device returns the identifier of an object on the display to the user program. It is usually implemented with the same physical device as a locator, but has a separate software interface to the user program. We will examine a few alternatives for implementing picking in WebGL through the use of a pointing device.
4. **Choice** Choice devices allow the user to select one of a discrete number of options. In WebGL, we can use various widgets provided by the window system. A **widget** is a graphical interactive device, provided by either the window system or a toolkit. Typical widgets include menus, slide bars, and graphical buttons. Most widgets are implemented as special types of windows. For example, a menu with n selections acts as a choice device, allowing us to select one of n alternatives. Widget sets are the key element defining a **graphical user interface**, or **GUI**. We will be able to use simple buttons, menus, and scroll bars in our examples without using a separate GUI package.
5. **Valuator** Valuators provide analog input to the user program. On older graphics systems, there were boxes or dials to provide valuator input. Here again, widgets within various toolkits usually provide this facility through graphical devices such as slide bars and radio boxes.

6. **Stroke** A stroke device returns an array of locations. Although we can think of a stroke device as similar to multiple uses of a locator, it is often implemented such that an action—say, pushing down a mouse button—starts the transfer of data into the specified array, and a second action, such as releasing the button, ends this transfer.

We can augment this list to include types of input from modern devices like touch pads and panels that accept multiple, simultaneous physical interactions and aggregate those actions into logical inputs representing gestures. Considering a "two-finger pinch" gesture, the device would register two initial "touches" at distinct locations. As the user completes the pinch gesture, the device would need to aggregrate the motion of the user's fingers moving toward the same location on the touch device. If an application were provided this "raw" input, it would be challenging to intrepret the input data to determine the actual user gesture. Fortunately, most windowing systems for these types of devices do the processing either in the touch device's hardware or the device driver.

3.4.8 Input Modes

The manner by which input devices provide input to an application program can be described in terms of two entities: a measure process and a device trigger. The **measure** of a device is what the device returns to the user program. The **trigger** of a device is a physical input on the device with which the user can signal the computer. For example, the measure of a keyboard should include a single character or a string of characters, and the trigger can be the Return or Enter key. For a locator, the measure includes the position of the locator, and the associated trigger can be a button on the physical device. The measure processes are initiated by the browser when the application code has been loaded.

The application program can obtain the measure of a device in three distinct modes. Each mode is defined by the relationship between the measure process and the trigger. Once a measure process is started, the measure is taken and placed in a buffer, even though the contents of the buffer may not yet be available to the program. For example, the position of a mouse is tracked continuously by the underlying window system and a cursor is displayed regardless of whether the application program needs mouse input.

In **request mode**, the measure of the device is not returned to the program until the device is triggered. This input mode is standard in nongraphical applications. For example, if a typical C program requires character input, we use a function such as `scanf` in C or `cin` in C++. When the program needs the input, it halts when it encounters the `scanf` or `cin` statement and waits while we type characters at our terminal. We can backspace to correct our typing, and we can take as long as we like. The data are placed in a keyboard buffer whose contents are returned to our program only after a particular key, such as the Enter key (the trigger), is pressed. For a logical device, such as a locator, we can move our pointing device to the desired location and then trigger the device with its button; the trigger will cause the location to be returned to the application program. The relationship between measure and trigger for request mode is shown in Figure 3.9.

FIGURE 3.9 Request mode.

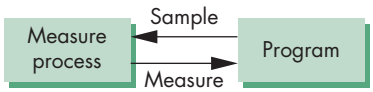

FIGURE 3.10 Sample mode.

Sample-mode input is immediate. As soon as the function call in the user program is encountered, the measure is returned. Hence, no trigger is needed (Figure 3.10). In sample mode, the user must have positioned the pointing device or entered data using the keyboard before the function call, because the measure is extracted immediately from the buffer.

One characteristic of both request- and sample-mode input in APIs that support them is that the user must identify which device is to provide the input. Consequently, we ignore any other information that becomes available from any input device other than the one specified. Both request and sample modes are useful for situations where the program guides the user, but are not useful in applications where the user controls the flow of the program. For example, a flight simulator or computer game might have multiple input devices—such as a joystick, dials, buttons, and switches—most of which can be used at any time. Writing programs to control the simulator with only sample- and request-mode input is nearly impossible because we do not know what devices the pilot will use at any point in the simulation. More generally, sample- and request-mode input are not sufficient for handling the variety of possible human–computer interactions that arise in a modern computing environment.

Our third mode, **event mode**, can handle these other interactions. We introduce it in three steps. First, we show how event mode can be described as another mode within our measure–trigger paradigm. Second, we discuss the basics of clients and servers where event mode is the preferred interaction mode. Third, we show the event-mode interface to WebGL through event handlers.

Suppose that we are in an environment with multiple input devices, each with its own trigger and each running a measure process. Each time that a device is triggered, an **event** is generated. The device measure, including the identifier for the device, is placed in an **event queue**. This process of placing events in the event queue is completely independent of what the application program does with these events. One way that the application program can work with events is shown in Figure 3.11. The application program can examine the front event in the queue or, if the queue is empty, can wait for an event to occur. If there is an event in the queue, the program can look at the first event's type and then decide what to do. If, for example, the first event is from the keyboard but the application program is not interested in keyboard input, the event can be discarded and the next event in the queue can be examined.

FIGURE 3.11 Event–mode model.

Another approach is to associate a function called a **callback** with a specific type of event. Event types can be subdivided into a few categories. **Mouse events** include moving the mouse (or other pointing device) and depressing or releasing one or more mouse buttons. **Window events** include opening or closing a window, replacing a window with an icon, and resizing a window with the pointing device. **Keyboard events** include pressing or releasing a key. Other event types are associated with the operating system and the browser, such as idle time-outs and indicators as to when a page has been loaded.

From the perspective of the window system, the operating system queries or polls the event queue regularly and executes the callbacks corresponding to events in the queue. We take this approach because it is the one currently used with the major window systems and has proved effective in client–server environments.

3.5 CLIENTS AND SERVERS

So far, our approach to input has been isolated from all other activities that might be happening in our computing environment. We have looked at our graphics system as a monolithic box with limited connections to the outside world, rather than through our carefully controlled input devices and a display. Networks and multiuser computing have changed this picture dramatically and to such an extent that, even if we had a single-user isolated system, its software probably would be configured as a simple client–server network.

If computer graphics is to be useful for a variety of real applications, it must function well in a world of distributed computing and networks. In this world, our building blocks are entities called **servers** that can perform tasks for **clients**. Clients and servers can be distributed over a network (Figure 3.12) or contained entirely within a single computational unit. Familiar examples of servers include print servers, which can allow sharing of a high-speed printer among users; compute servers, such as remotely located high-performance computers, accessible from user programs; file servers that allow users to share files and programs, regardless of the machine they are logged into; and terminal servers that handle dial-in access. Users and user programs that make use of these services are clients or client programs. Servers can also exist at a lower level of granularity within a single operating system. For example, the operating system might provide a clock service that multiple client programs can use.

It is less obvious what we should call a workstation connected to the network: It can be both a client and a server, or, perhaps more to the point, a workstation may run client programs and server programs concurrently.

The model that we use here was popularized by the X Window System. We use much of that system's terminology, which is now common to most window

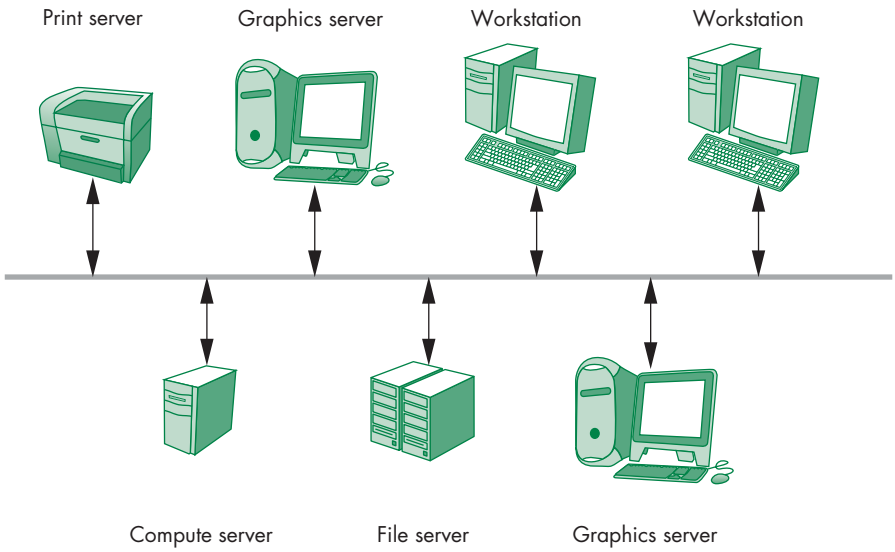

FIGURE 3.12 Network.

systems and fits well with graphical applications. A workstation with a raster display, a keyboard, and a pointing device, such as a mouse, is a **graphics server**. The server can provide output services on its display and input services through the keyboard and pointing device. These services are potentially available to clients anywhere on the network.

Application programs written in C or C++ that use desktop OpenGL for graphics applications are clients that use the graphics server. Within an isolated system, this distinction may not be apparent as we write, compile, and run the software on a single machine. However, we also can run the same application program using other graphics servers on the network. Note that in a modern system, the GPU acts as the graphics server for display, whereas the CPU is the client for those services.

As we saw in Chapter 2, WebGL works within a browser. The browser accesses applications on web servers and the browser is a **web client**. We can regard the World Wide Web as a vast storehouse of information stored as web pages in web servers using standard encodings, such as HTML for documents or JPEG for images.

3.6 PROGRAMMING EVENT-DRIVEN INPUT

In this section, we develop event-driven input through a set of simple examples that use the callback mechanism that we introduced in Section 3.4. We examine various events that are recognized by WebGL through HTML5 and, for those of interest to our application, we write callback functions that govern how the application program responds to the events. Note that because input is not part of WebGL, we will obtain

input through callbacks that are supported by the browser and thus are not restricted to use by graphics applications.

3.6.1 Events and Event Listeners

Because WebGL is concerned with rendering and not input, we use JavaScript and HTML for the interactive part of our application. An event is classified by its type and **target**. The target is an object, such as a button, that we create through the HTML part of our code and appears on the display. A target can also be a physical object such as a mouse. Thus, a "click" is an event type whose target could be a button object or a mouse object. The measure of the device is associated with the particular object.

The notion of event types works well not only with WebGL but also within the HTML environment. Event types can be thought of as members of a higher-level classification of events onto **event categories**. Our primary concern will be with the category of device-dependent input events, which includes all the types associated with devices such as a mouse and a keyboard. Within this category, event types include **mousedown**, **keydown**, and **mouse click**. Each event has a name that is recognized by JavaScript and usually begins with the prefix `on`, such as `onload` and `onclick`. For a device-independent type such as `onclick`, the target might be a physical mouse or a button on the display that was created as part of our HTML document. The target of the `onload` event that we have seen in our examples is our canvas.

We can respond to events in a number of ways. In our examples, we invoked our initialization function by

```
window.onload = init;
```

Here the event type is `load` with a target of our window. The callback function is `init`.[2] Callbacks that we associate with events are called **event listeners** or **event handlers**.

3.6.2 Adding a Button

Suppose we want to alter our rotating cube so we can rotate it either clockwise or counterclockwise and switch between these modes through a graphical button that can be clicked using the mouse. We can add a button element in the HTML file with the single line of code

```
<button id="DirectionButton">Change Rotation Direction</button>
```

which gives an identifier to the new element and puts a label ("Change Rotation Direction") on the display inside the button. In the JavaScript file, we define a boolean variable for the direction that gets used in the render function to select a positive or negative rotation:

2. Note that one reason for using an event here is the asynchronous nature of code running in the browser. By forcing our program to wait for the entire program to be loaded by the browser, we gain control over when our application can proceed.

```
var direction = true;

theta += (direction ? 0.1 : -0.1);
```

Finally, we need to couple the button element with a variable in our program and add an event listener:

```
var myButton = document.getElementById("DirectionButton");
myButton.addEventListener("click", function() { direction = !direction;
});
```

We can also use the alternate form

```
document.getElementById("DirectionButton").onclick =
  function() { direction = !direction; };
```

The click event is not the only one that can be coupled to our button. We could also use the mousedown event

```
myButton.addEventListener("mousedown",
                          function() { direction = !direction; });
```

assuming that during initialization we define

```
var direction = true;
```

Although in this example we can use either event, in a more complex application we might prefer to use the mousedown event to be specific as to which device can cause the event. We can achieve even more specificity using additional information in the measure returned by the event. For example, if we are using a multibutton mouse, we can restrict the change to a particular button, as in

```
myButton.addEventListener("click", function() {
  if (event.button == 0) { direction = !direction; }
});
```

where on a three-button mouse button 0 is the left mouse button, button 1 is the middle button, and button 2 is the right button.

If we have a single-button mouse, we can use the meta keys on the keyboard to give us more flexibility. For example, if we want to use the Shift key with our click event, our code might look like

```
myButton.addEventListener("click", function() {
  if (event.shiftKey == 0) { direction = !direction; }
});
```

We can also put all the button code in the HTML file instead of dividing it between the HTML file and the JavaScript file. For our button, we could simply have the code

```
<button onclick="direction = !direction"></button>
```

in the HTML file. However, we prefer to separate the description of the object on the page from the action associated with the object, with the latter being in the JavaScript file.

3.6.3 Menus

Menus are specified by **select elements** in HTML that we can define in our HTML file. A menu can have an arbitrary number of entries, each of which has two parts: the text that is visible on the display and a number that we can use on our application to couple that entry to a callback. We can demonstrate menus with our rotating square by adding a menu with three entries to our HTML file:

```
<select id="mymenu" size="3">
<option value="0">Toggle Rotation Direction X</option>
<option value="1">Spin Faster</option>
<option value="2">Spin Slower</option>
</select>
```

As with a button, we create an identifier that we can refer to in our application. Each line in the menu has a `value` that is returned when that row is clicked with the mouse. Let's first modify our render function slightly so we can alter the speed of rotation by having a variable `delay` that we use with a timer to control the rate of animation:

```
var delay = 100;

function render()
{
  setTimeout(function() {
    requestAnimFrame(render);
    gl.clear(gl.COLOR_BUFFER_BIT);
    theta += (direction ? 0.1 : -0.1);
    gl.uniform1f(thetaLoc, theta);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
  }, delay);
}
```

The click event returns the line of the menu that was pointed to through the `selectedIndex` member of `m`:

```
var m = document.getElementById("mymenu");

m.addEventListener("click", function() {
  switch (m.selectedIndex) {
    case 0:
      direction = !direction;
      break;
    case 1:
      delay /= 2.0;
      break;
```

```
    case 2:
      delay *= 2.0;
      break;
  }
});
```

3.6.4 Using Keycodes

We can also use key press events to control our rotating square. Suppose that we want to use the numeric keys 1, 2, and 3 rather than the menu. These keys have codes 49, 50, and 51 in Unicode (and ASCII). Now we can use the keydown event and a simple listener.

In the following code, you will see that we are responding to a window event that occurs on the page, not in the WebGL window. Hence, we use the global objects `window` and `event`, which are defined by the browser and available to all JavaScript programs.

```
window.addEventListener("keydown", function() {
  switch (event.keyCode) {
    case 49:  // '1' key
      direction = !direction;
      break;
    case 50:  // '2' key
      delay /= 2.0;
      break;
    case 51:  // '3' key
      delay *= 2.0;
      break;
    }
});
```

This listener requires us to know the Unicode mapping of keycodes to characters. Instead, we could do this mapping with a listener of the form

```
window.onkeydown = function(event) {
  var key = String.fromCharCode(event.keyCode);
  switch (key) {
    case '1':
      direction = !direction;
      break;
    case '2':
      delay /= 2.0;
      break;
    case '3':
      delay *= 2.0;
      break;
  }
};
```

3.6.5 Sliders

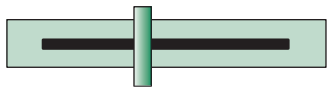

FIGURE 3.13 Slide bar.

Rather than incrementing or decrementing a value in our program through repetitive uses of a button or key, we can add a **slider element** to our display, as in Figure 3.13. We move the slider with our mouse, and the movements generate events whose measure includes a value dependent on the position of the slider. Thus, when the slider is at the left end, the value will be at its minimum and when the slider is on the right, the value will be at its maximum.

In a similar manner to buttons and menus, we can create a visual element on our web page in the HTML file and handle the input from the events in the JavaScript file. For sliders, we specify the minimum and maximum values corresponding to the left and right extents of the slider and the initial value of the slider. We also specify the minimum change necessary to generate an event. We usually also want to display some text on either side of the slider that specifies the minimum and maximum values.

Suppose that we want to create a slider to set the delay between 0 and 100 milliseconds. A basic slider can be created in the HTML file by using the HTML **range element**, as shown below:

```
<input id="slide" type="range"
  min="0" max="100" step="10" value="50" />
```

We use `id` to identify this element to the application. The `type` parameter identifies which HTML input element we are creating. The `min`, `max`, and `value` parameters give the minimum, maximum, and initial values associated with the slider. Finally, `step` gives the amount of change needed to generate an event. We can display the minimum and maximum values on the sides and put the slider below the element that precedes it on the page by

```
<div>
speed 0 <input id="slide" type="range"
  min="0" max="100" step="10" value="50" />
100 </div>
```

In the application, we can get the value of speed from the slider with the two lines

```
document.getElementById("slide").onchange =
  function() { delay = event.srcElement.value; };
```

As with buttons and menus, we use `getElementById` to couple the slider with the application. The value of the slider is returned in `event.srcElement.value` each time an event is generated. All the input elements we have described so far—menus, buttons, and sliders—are displayed using defaults for the visual appearance. We can beautify the visual display in many ways, ranging from using HTML and CSS to using various packages.

3.7 POSITION INPUT

In our examples so far when we used the mouse, all we made use of was the fact that the event occurred and perhaps which button generated the event. There is more information available when we create a click event or mousedown event. In particular, we can access the location of the mouse when the event occurred.

When a click or mouse event occurs, the returned event object includes the values `event.ClientX` and `event.ClientY`, which give the position of the mouse in window coordinates. Recall that positions in the window have dimensions `canvas.width` × `canvas.height`. Positions are measured in pixels with the origin at the upper-left corner so that positive y values are down. For this position to be useful in our application, we must transform these values to the same units as the application.

In this chapter, we are using clip coordinates for our applications, which range from $(-1, 1)$ in both directions and the positive y direction is up. If (x_w, y_w) is the position in the window with width w and height h, then the position in clip coordinates is obtained by flipping the y value and rescaling, yielding the equations

$$x = -1 + \frac{2 * x_w}{w}, \qquad y = -1 + \frac{2 * (w - y_w)}{w}.$$

We can use the mouse position in many ways. Let's start by simply converting each position to clip coordinates and placing it on the GPU. We use a variable `index` to keep track of how many points we have placed on the GPU and initialize our arrays as in previous examples. Consider the event listener

```
canvas.addEventListener("click", function() {
  gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
  var t = vec2(-1 + 2*event.clientX/canvas.width,
               -1 + 2*(canvas.height-event.clientY)/canvas.height);
  gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2']*index, t);
  index++;
});
```

The click event returns the object `event` that has members `event.clientX` and `event.clientY`, which is the position in the WebGL window with the y value measured from the top of the window.

If all we want to do is display the locations, we can use the render function

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.drawArrays(gl.POINTS, 0, index);
  window.requestAnimFrame(render, canvas);
}
```

Note that our listener can use the mousedown event instead of the click event, and the display will be the same.[3]

We can also demonstrate many of the elements of painting applications by making a few minor changes. For example, if we use

```
gl.drawArrays(gl.TRIANGLE_STRIP, 0, index);
```

in our render function, the first three points define the first triangle and each successive mouse click will add another triangle. We can also add color. For example, suppose that we specify the seven colors

```
var colors = [
  vec4(0.0, 0.0, 0.0, 1.0),  // black
  vec4(1.0, 0.0, 0.0, 1.0),  // red
  vec4(1.0, 1.0, 0.0, 1.0),  // yellow
  vec4(0.0, 1.0, 0.0, 1.0),  // green
  vec4(0.0, 0.0, 1.0, 1.0),  // blue
  vec4(1.0, 0.0, 1.0, 1.0),  // magenta
  vec4(0.0, 1.0, 1.0, 1.0)   // cyan
];
```

Then each time we add a point, we also add a color, chosen either randomly or by cycling through the seven colors, as in the code

```
gl.bindBuffer(gl.ARRAY_BUFFER, cBufferId);
var t = vec4(colors[index%7]);
gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec4']*index, t);
```

The website contains a number of sample programs that illustrate interaction. There are three versions of the rotating square application. The first, `rotatingSquare1`, only displays the square without interaction. The second, `rotatingSquare2`, adds buttons and menus. The third, `rotatingSquare3`, adds a slider. The application `square` places a colored square at each location where the mouse is clicked. The application `triangle` draws a triangle strip using the first three mouse clicks to specify the first triangle; then successive mouse clicks each add another triangle.

3.8 WINDOW EVENTS

Most window systems allow the user to resize the window interactively, usually by using the mouse to drag a corner of the window to a new location. This event is an example of a **window event**. Other examples include exposing an element that was hidden under another element and minimizing or restoring a window. In each case, we can use an event listener to alter the display.

3. If we want to display points at a larger size, we can set the value of `gl_PointSize` in the vertex shader.

Let's consider the **resize** or **reshape** event. If such an event occurs, the application program can decide what to do. If the window size changes, we have to consider three questions:

1. Do we redraw all the objects that were on the canvas before it was resized?
2. What do we do if the aspect ratio of the new window is different from that of the old window?
3. Do we change the sizes or attributes of new primitives if the size of the new window is different from that of the old?

There is no single answer to any of these questions. If we are displaying the image of a real-world scene, our resize function probably should make sure that no shape distortions occur. But this choice may mean that part of the resized window is unused or that part of the scene cannot be displayed in the window. If we want to redraw the objects that were in the window before it was resized, we need a mechanism for storing and recalling them.

Suppose that when we resize the window, we want to display the same contents as before and also maintain proportions on the canvas. Resizing refers to the entire browser window, which includes the canvas to which we are rendering and other elements, such as menus or buttons specified in our HTML file. The resize event returns the height and width of the resized window (`innerHeight` and `innerWidth`). The original canvas was specified with height and width given by `canvas.height` and `canvas.width`. As long as the smaller of the new window height and width is greater than the larger of the original canvas height and width, we need not modify our rendering. Once this condition is violated, we change the viewport to be small enough to fit in the resized window but maintain proportions. The following code assumes a square canvas must be maintained:

```
window.onresize = function() {
  var min = innerWidth;

  if (innerHeight < min) {
    min = innerHeight;
  }
  if (min < canvas.width || min < canvas.height) {
    gl.viewport(0, canvas.height-min, min, min);
  }
};
```

We could use our graphics primitives and our mouse callbacks to construct various graphical input devices. For example, we could construct a more visually pleasing slide bar using filled rectangles for the device, text for any labels, and the mouse to get the position. However, much of the code would be tedious to develop. There are many JavaScript packages that provide sets of widgets, but because our philosophy is not to restrict our discussion to any particular package, we will not discuss the specifics of such widget sets.

3.9 PICKING

Picking is the logical input operation that allows the user to identify an object on the display. Although the action of picking uses the pointing device, the information that the user wants returned to the application program is not a position. A pick device is considerably more difficult to implement on a modern system than is a locator.

Such was not always the case. Old display processors could accomplish picking easily by means of a light pen. Each redisplay of the screen would start at a precise time. The light pen would generate an interrupt when the redisplay passed its sensor. By comparing the time of the interrupt with the time that the redisplay began, the processor could identify an exact place in the display list and subsequently could determine which object was being displayed.

One reason for the difficulty of picking in modern systems is the forward nature of the rendering pipeline. Primitives are defined in an application program and move forward through a sequence of geometric operations, rasterization, and fragment operations on their way to the framebuffer. Although much of this process is reversible in a mathematical sense, the hardware is not reversible. Hence, converting from a location on the display to the corresponding primitive is not a direct calculation. There are also potential uniqueness problems (see Exercises 3.7 and 3.8).

There are at least four ways to deal with this difficulty. One process, known as **selection**, involves adjusting the clipping region and viewport such that we can keep track of which primitives in a small clipping region are rendered into a region near the cursor. The names assigned to these primitives go into a **hit list** that can be examined later by the user program. Older versions of OpenGL supported this approach, but it has been deprecated in shader-based versions, and most application programmers prefer one of the other approaches.

If we start with our synthetic-camera model, we can build an approach based on the idea that if we generate a ray from the center of projection through the location of the mouse on the projection plane, we can, at least in principle, check for which objects this ray intersects. The closest object we intersect is the one selected. This approach is best suited to a ray-tracing renderer but can be implemented with a pipeline architecture, although with a performance penalty.

A simple approach is to use (**axis-aligned**) **bounding boxes**, or **extents**, for objects of interest. The extent of an object is the smallest rectangle, aligned with the coordinates axes, that contains the object. For two-dimensional applications, it is relatively easy to determine the rectangle in screen coordinates that corresponds to a rectangle point in object or world coordinates. For three-dimensional applications, the bounding box is a right parallelepiped. If the application program maintains a simple data structure to relate objects and bounding boxes, approximate picking can be done within the application program.

Another simple approach involves using an extra color buffer, which is not displayed, and an extra rendering. Suppose that we render our objects into this second color buffer, each in a distinct color. The application programmer is free to determine an object's contents by simply changing colors wherever a new object definition appears in the program.

We can perform picking in four steps that are initiated by a user-defined pick function in the application. First, we draw the objects into the second buffer with the pick colors. Second, we get the position of the mouse using the mouse callback. Third, we use the function `gl.readPixels` to find the color at the position in the second buffer corresponding to the mouse position. Finally, we search a table of colors to find which object corresponds to the color read. We must follow this process by a normal rendering into the framebuffer. We will develop this approach in Chapter 7.

3.10 BUILDING MODELS INTERACTIVELY

One example of computer-aided design (CAD) is building geometric structures interactively. In Chapter 4, we will look at ways in which we can model geometric objects comprised of polygons. Here, we want to examine the interactive part.

Let's start by writing an application that will let the user specify a series of axis-aligned rectangles interactively. Each rectangle can be defined by two mouse positions at diagonally opposite corners. Consider the event listener

```
canvas.addEventListener("mousedown", function() {
  gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
  if (first) {
    first = false;
    var t1 = vec2(-1 + 2*event.clientX/canvas.width,
                  -1 + 2*(canvas.height-event.clientY)/canvas.height);
  }
  else {
    first = true;
    var t2 = vec2(-1 + 2*event.clientX/canvas.width-1,
                  -1 + 2*(canvas.height-event.clientY)/canvas.height);
    var t3 = vec2(t1[0], t2[1]);
    var t4 = vec2(t2[0], t1[1]);

    gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2']*(index+0),
                     flatten(t1));
    gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2']*(index+1),
                     flatten(t3));
    gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2']*(index+2),
                     flatten(t2));
    gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2']*(index+3),
                     flatten(t4));

    index += 4;
  }
});
```

and the render function

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  for (var i = 0; i < index; i += 4) {
```

```
    gl.drawArrays(gl.TRIANGLE_FAN, i, 4);
  }

  window.requestAnimFrame(render);
}
```

We use the boolean variable `first` to keep track of whether the mouse click is generating a new rectangle or generating the diagonally opposite corner of a rectangle from the previous mouse click. The position of the mouse from the first click is stored. When the second click occurs, the two positions are used to compute the positions of the two other vertices of the rectangle, and then all four positions are put on the GPU. The order in which they are put on the GPU is determined by the render function's use of a triangle fan, rather than the triangle strip we used in previous examples. We will see the advantage of this form when we extend our example to polygons with more than four vertices. The rest of the program is similar to our previous examples.

We add a color selection menu to the HTML files:

```
<select id="mymenu" size="7">
<option value="0">Black</option>
<option value="1">Red</option>
<option value="2">Yellow</option>
<option value="3">Green</option>
<option value="4">Blue</option>
<option value="5">Magenta</option>
<option value="6">Cyan</option>
</select>
```

The event listener for this menu simply stores the index of the color, thus making it the current color that will be used until another color is selected. Here is the code for color selection:

```
var cIndex = 0;
var colors = [
  vec4(0.0, 0.0, 0.0, 1.0),  // black
  vec4(1.0, 0.0, 0.0, 1.0),  // red
  vec4(1.0, 1.0, 0.0, 1.0),  // yellow
  vec4(0.0, 1.0, 0.0, 1.0),  // green
  vec4(0.0, 0.0, 1.0, 1.0),  // blue
  vec4(1.0, 0.0, 1.0, 1.0),  // magenta
  vec4(0.0, 1.0, 1.0, 1.0)   // cyan
];

var m = document.getElementById("mymenu");
m.addEventListener("click", function() { cIndex = m.selectedIndex; });
```

The color index can be used in multiple ways. If we want each rectangle to be a solid color, we can set up a vertex array for the colors and then augment the event listener for the vertex positions by adding the code

```
gl.bindBuffer(gl.ARRAY_BUFFER, cBufferId);
var t = vec4(colors[cIndex]);
```

```
gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec4']*(index-4), flatten(t));
gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec4']*(index-3), flatten(t));
gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec4']*(index-2), flatten(t));
gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec4']*(index-1), flatten(t));
```

Note that this code is coming after we have already incremented `index` when we put the four vertex positions on the GPU. If we want each rectangle to be displayed in a solid color, we could store these colors in an array; then in the render function we could send each polygon's color to the shaders as a uniform variable. Using a vertex array does, however, let us assign a different color to each vertex and then have the rasterizer interpolate these colors over the rectangle.

Now let's consider what changes are needed to allow the user to work with a richer set of objects. Suppose that we want to be able to design a polygon with an arbitrary number of vertices. Although we can easily store as many vertices as we like in our event listener, there are some issues. For example,

1. How do we indicate the beginning and end of a polygon when the number of vertices is arbitrary?
2. How do we render when each polygon can have a different number of vertices?

We can solve the first problem by adding a button that will end the present polygon and start a new one. Solving the second problem involves adding some additional structure to our code. The difficulty is that, while it is easy to keep adding vertices to our vertex array, the shaders do not have the information as to where one polygon ends and the next begins. We can, however, store such information in an array in our program.

We will make use of `gl.TRIANGLE_FAN` in the render function because it will render a list of successive vertices into a polygon without having to reorder the vertices as we did with `gl.TRIANGLE_STRIP`. However, having all the triangles that comprise a polygon share the first vertex does not lead to a particularly good triangulation of the set of vertices. In Chapter 12, we will consider better triangulation algorithms.

First, we consider a single polygon with an arbitrary number of vertices. The event listener for the mouse can add colors and positions to vertex arrays each time the mouse is clicked in the canvas.

```
canvas.addEventListener("mousedown", function() {
  var t = vec2(2*event.clientX/canvas.width-1,
               2*(canvas.height-event.clientY)/canvas.height-1);
  gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
  gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2']*index, flatten(t));

  t = vec4(colors[cIndex]);
  gl.bindBuffer(gl.ARRAY_BUFFER, cBuffer);
  gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec4']*index, flatten(t));

  index++;
});
```

We add a button

```
<button id="Button1">End Polygon</button>
```

in the HTML file and the corresponding event listener

```
getElementById("Button1").onclick = function() {
  render();
  index = 0;
});
```

and the render function

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.drawArrays(gl.TRIANGLE_FAN, 0, index);
}
```

to the application file.

As simple as this code is, there are a couple of interesting points. Note that the render function is called from the button listener, which then resets the index that counts vertices. We can change the color from the color menu between adding vertices. The rasterizer will blend the colors for the next couple of vertices when a color is changed.

To get multiple polygons, we need to keep track of the beginning of each polygon and how many vertices are in each one. We add the three variables

```
var numPolygons = 0;
var numIndices = [ 0 ];
var start = [ 0 ];
```

The variable `numPolygons` stores the number of polygons we have entered so far. The array `numIndices` stores the number of vertices for each polygon, and the array `start` stores the index of the first vertex in each polygon. The only change we have to make to the mouse listener is to increase the number of vertices in the present polygon

```
numIndices[numPolygons]++;
```

The button callback

```
getElementById("Button1") = function() {
  numPolygons++;
  numIndices[numPolygons] = 0;
  start[numPolygons] = index;
  render();
});
```

starts a new polygon before rendering. Finally, the rendering function is

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  for (var i = 0; i < numPolygons; ++i) {
    gl.drawArrays(gl.TRIANGLE_FAN, start[i], numIndices[i]);
  }
}
```

The programs `cad1` and `cad2` on the website show some of the elements that go into a simple painting program; `cad1` draws a new rectangle specified by each pair of mouse clicks, and `cad2` allows the user to draw polygons with an arbitrary number of vertices.

3.11 DESIGN OF INTERACTIVE PROGRAMS

Defining what characterizes a good interactive program is difficult, but recognizing and appreciating a good interactive program is easy. A good program includes features such as these:

- A smooth display, showing neither flicker nor any artifacts of the refresh process
- A variety of interactive devices on the display
- A variety of methods for entering and displaying information
- An easy-to-use interface that does not require substantial effort to learn
- Feedback to the user
- Tolerance for user errors
- A design that incorporates consideration of both the visual and motor properties of the human

The importance of these features and the difficulty of designing a good interactive program should never be underestimated. The field of human–computer interaction (HCI) is an active one and we will not shortchange you by condensing it into a few pages. Our concern in this book is computer graphics; within this topic, our primary interest is rendering. However, there are a few topics common to computer graphics and HCI that we can pursue to improve our interactive programs.

SUMMARY AND NOTES

In this chapter, we have touched on a number of topics related to interactive computer graphics. These interactive aspects make the field of computer graphics exciting and fun.

Our discussion of animation showed both the ease with which we can animate a scene with WebGL and some of the limitations on how we can control the animation in a web environment.

We have been heavily influenced by the client–server perspective. Not only does it allow us to develop programs within a networked environment but it also makes it possible to design programs that are portable yet can still take advantage of special features that might be available in the hardware. These concepts are crucial for object-oriented graphics and graphics for the Internet.

From the application programmer's perspective, various characteristics of interactive graphics are shared by most systems. We see the graphics part of the system as a server, consisting of a raster display, a keyboard, and a pointing device. In almost all workstations, we have to work within a multiprocessing, windowed environment. Most likely, many other processes are executing concurrently with the execution of your graphics program. However, the window system allows us to write programs for a specific window that act as though that window were the display device of a single-user system.

The overhead of setting up a program to run in this environment is small. Each application program contains a set of function calls that is virtually the same in every program. The use of logical devices within the application program frees the programmer from worrying about the details of particular hardware.

Within the environment that we have described, event-mode input is the norm. Although the other forms are available—request mode is the normal method used for keyboard input—event-mode input gives us far more flexibility in the design of interactive programs.

The speed of the latest generation of graphics processors not only allows us to carry out interactive applications that were not possible even a few years ago but also makes us rethink (as we should periodically) whether the techniques we are using are still the best ones. For example, whereas hardware features such as logical operations and overlay planes made possible many interactive techniques, now with a fast GPU we can often simply draw the entire display fast enough that these features are no longer necessary.

Because our API, WebGL, is independent of any operating or window system, we were able to use the simple event-handling capabilities in JavaScript to get input from a mouse and the keyboard. Because we want to keep our focus on computer graphics, this approach was justified but nevertheless led to applications with a limited and inelegant interface. To get a better graphical user interface (GUI), the best approach would be to use some combination of HTML, CSS, and available GUI packages. Some references are in the Suggested Readings section that follows.

Interactive computer graphics is a powerful tool with unlimited applications. At this point, you should be able to write fairly sophisticated interactive programs. Probably the most helpful exercise that you can do now is to write one. The exercises at the end of the chapter provide suggestions.

SUGGESTED READINGS

Many of the conceptual foundations for the windows-icons-menus-pointing interfaces that we now consider routine were developed at the Xerox Palo Alto Research Center (PARC) during the 1970s (see [Sch97]). The mouse also was developed there

[Eng68]. The familiar interfaces of today—such as the Macintosh Operating System, the X Window System, and Microsoft Windows—all have their basis in this work.

The volume by Foley and associates [Fol94] contains a thorough description of the development of user interfaces with an emphasis on the graphical aspects. The books by Schneiderman [Sch97] and Nielson [Nie94] provide an introduction to HCI.

The X Window System [Sch88] was developed at the Massachusetts Institute of Technology and is the de facto standard in the UNIX workstation community. The development of the Linux version for PCs has allowed the X Window System to run on these platforms too.

The input and interaction modes that we discussed in this chapter grew out of the standards that led to GKS [ANSI85] and PHIGS [ANSI88]. These standards were developed for both calligraphic and raster displays; thus, they do not take advantage of the possibilities available on raster-only systems (see [Pik84] and [Gol83]).

Using desktop OpenGL requires the application developer to choose between a platform-dependent interfacing method that gives access to the full capabilities of the local system or a simplified toolkit that supports the functionality common to all systems. Previous editions of this text [Ang10] used the GLUT toolkit [Kil94b] exclusively. Additional details on GLUT are found in *OpenGL: A Primer* [Ang08]. See [Kil94a, OSF89] for details on interfacing directly with the X Window System and various X Window toolkits. Tookits including freeglut and GLEW are available for extending GLUT to recent versions of OpenGL; see [Shr13] and the OpenGL website, *www.opengl.org*.

The approach we have taken here is to use the event-handling functionality built into JavaScript [Fla11]. We avoid use of HTML, CSS, or any of the many GUI packages available. Consequently, our code is simple, portable, and limited. The most popular package for interfacing is jQuery [McF12], which provides more widgets and a better interface to the capabilities of HTML and CSS. For an introduction to HTML and CSS, see [Duk11].

To end where we began, Sutherland's Sketchpad is described in [Sut63].

EXERCISES

3.1 Rewrite the Sierpinski gasket program from Chapter 2 such that the left mouse button will start the generation of points on the screen, the right mouse button will halt the generation of new points, and the middle mouse button will terminate the program. Include a resize callback.

3.2 Construct slide bars to allow users to define colors in the CAD program. Your interface should let the user see a color before that color is used.

3.3 Add an elapsed-time indicator in the CAD program (Section 3.10) using a clock of your own design.

3.4 Creating simple games is a good way to become familiar with interactive graphics programming. Program the game of checkers. You can look at each

square as an object that can be picked by the user. You can start with a program in which the user plays both sides.

3.5 Write a program that allows a user to play a simple version of solitaire. First, design a simple set of cards using only our basic primitives. Your program can be written in terms of picking rectangular objects.

3.6 Simulating a pool or billiards game presents interesting problems. As in Exercise 2.17, you must compute trajectories and detect collisions. The interactive aspects include initiating movement of the balls via a graphical cue stick, ensuring that the display is smooth, and creating a two-person game.

3.7 The mapping from a point in object or world coordinates to one in screen coordinates is well defined. It is not invertible because we go from three dimensions to two dimensions. Suppose, however, that we are working with a two-dimensional application. Is the mapping invertible? What problem can arise if you use a two-dimensional mapping to return to a position in object or world coordinates by a locator device?

3.8 How do the results of Exercise 3.7 apply to picking?

3.9 In a typical application program, the programmer must decide whether or not to use display lists. Consider at least two applications. For each, list at least two factors in favor of display lists and two against.

3.10 Write an interactive program that will allow you to guide a graphical rat through the maze you generated in Exercise 2.7. You can use the left and right buttons to turn the rat and the middle button to move him forward.

3.11 Inexpensive joysticks, such as those used in toys and games, often lack encoders and contain only a pair of three-position switches. How might such devices function?

3.12 The orientation of an airplane is described by a coordinate system as shown in Figure 3.14. The forward–backward motion of the joystick controls the up–down rotation with respect to the axis running along the length of the airplane, called the **pitch**. The right–left motion of the joystick controls the rotation about this axis, called the **roll**. Write a program that uses the mouse to control pitch and roll for the view seen by a pilot. You can do this exercise in two dimensions by considering a set of objects to be located far from the airplane, then having the mouse control the two-dimensional viewing of these objects.

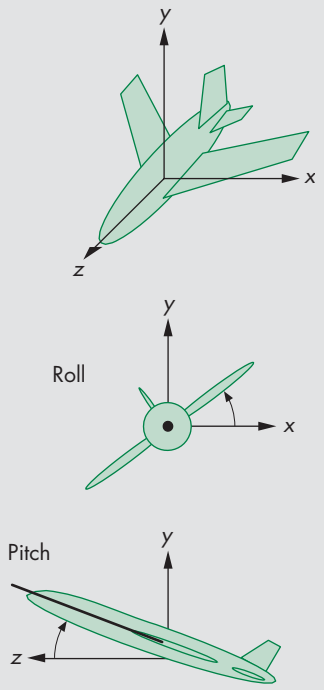

FIGURE 3.14 Airplane coordinate system.

3.13 Consider a table with a two-dimensional sensing device located at the end of two linked arms, as shown in Figure 3.15. Suppose that the lengths of the two arms are fixed and the arms are connected by simple (one degree of freedom) pivot joints. Determine the relationship between the joint angles θ and ϕ and the position of the sensor.

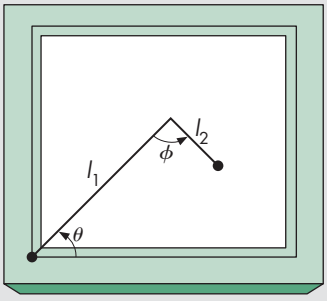

FIGURE 3.15 Two-dimensional sensing arm.

3.14 Suppose that a CRT has a square face of 40 × 40 centimeters and is refreshed in a noninterlaced manner at a rate of 60 Hz. Ten percent of the time that the system takes to draw each scan line is used to return the CRT beam from the right edge to the left edge of the screen (the horizontal retrace time), and 10 percent of the total drawing time is allocated for the beam to return from

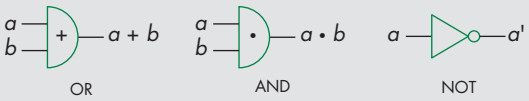

FIGURE 3.16 Symbols for logical circuits.

the lower-right corner of the screen to the upper-left corner after each refresh is complete (the vertical retrace time). Assume that the resolution of the display is 1024 × 1024 pixels. Find a relationship between the time at which a light pen detects the beam and the light pen's position. Give the result using both centimeters and screen coordinates for the location on the screen.

3.15 Circuit-layout programs are variants of paint programs. Consider the design of logical circuits using the boolean AND, OR, and NOT functions. Each of these functions is provided by one of the three types of integrated circuits (gates), the symbols for which are shown in Figure 3.16. Write a program that allows the user to design a logical circuit by selecting gates from a menu and positioning them on the screen. Consider methods for connecting the outputs of one gate to the inputs of others.

3.16 Extend Exercise 3.15 to allow the user to specify a sequence of input signals. Have the program display the resulting values at selected points in the circuit.

3.17 Extend Exercise 3.15 to have the user enter a logical expression. Have the program generate a logical diagram from that expression.

3.18 Use the methods of Exercise 3.15 to form flowcharts for programs or images of graphs that you have studied in a data structures class.

3.19 Plotting packages offer a variety of methods for displaying data. Write an interactive plotting application for two-dimensional curves. Your application should allow the user to choose the mode (polyline display of the data, bar chart, or pie chart), colors, and line styles.

3.20 The required refresh rate for CRT displays of 50 to 85 Hz is based on the use of short-persistence phosphors that emit light for extremely short intervals when excited. Long-persistence phosphors are available. Why are long-persistence phosphors not used in most workstation displays? In what types of applications might such phosphors be useful?

3.21 Modify the polygon program in Section 3.10 using a linked list rather than an array to store the objects. Your program should allow the user to both add and delete objects interactively.

3.22 Another CAD application that can be developed in WebGL is a paint program. You can display the various objects that can be painted—lines, rectangles, circles, and triangles, for example—and use picking to select which to draw. The mouse can then enter vertex data and select attributes such as colors from a menu. Write such an application.