Trackball: This is essentially an upside-down mouse, with a socket containing a ball which the user manipulates with her hand to make the cursor move.

Spaceball: This is a pointing device with six degrees of freedom versus the two of an ordinary mouse. It is used in special applications such as manipulating a camera in a 3D scene, where the camera not only moves, but also rotates. The spaceball itself consists of a pressure-sensitive ball which can distinguish different kinds of forces – including forward and backward, lateral and twist – the applied force manipulating the selected object.

Haptic device: This is a pointing device which gives physical feedback to the user based on the location of the cursor or, possibly, that of an object being moved along with the cursor. The easiest way to understand the functioning of a haptic device, if you have never used one, is to imagine a mouse with a mechanical ball which is (somehow) programmed to lock and stop rolling when the cursor reaches the side of the screen. The reaction the user then has is of that of the cursor running into a physical obstacle at the edge of the screen, though evidently it is moving in virtual space. The device depicted in Figure 1.12 is not a haptic mouse, of course, but one commonly seen in HCI (human-computer interaction) labs. The three-link arm swiveling on a ball gives it six degrees of freedom.

Haptics has numerous applications, a couple of noteworthy ones being the tele-operation of robots (where the operator gets haptic feedback as she manipulates a robot in either a virtual or a remote real environment) and simulated surgery training in medicine (which is similar to training pilots on a flight simulator, except that surgery has the added component of tactile feedback, mostly absent in flying).

Joystick: This is an input device popular in video games and applications such as flight simulators. It originated from its namesake found in real aircraft cockpits. A joystick pivots around a fixed base, gaining thus two degrees of freedom, and usually has buttons which can be depressed to provide additional input. In a game or simulator setting a joystick is typically used to control an object traveling through space. Nowadays, high-end joysticks have embedded motors to provide haptic feedback to user motion, e.g., resistance as a plane is banked.

Wheel: This again is a specialized input device for games and simulators, obviously derived from the car steering wheel, and provides rotational input in an exactly similar manner, most often to a virtual automobile. Again, haptic feedback to give the user a sense of the vehicle's response, and even of the terrain over which it is traveling, is becoming increasingly popular.

Gamepad: This device is the standard controller for many modern game consoles. Usual features include action buttons operated usually with the right thumb and a cross-shaped directional controller with the left.

Camera: Although this input device needs no introduction, it's worth noting the increasingly sophisticated uses a peripheral camera is being put to with the help, e.g., of software to recognize faces, gestures and expressions.

Touchscreen: Increasingly popular as the interface of handheld devices such as smartphones, a touchscreen is a display which can accept input via touch. It is similar to touchpads and tablets in that it senses the location of a finger or stylus – one or the other is usually preferable based on the particular technology used to make the screen – on the display area. A common application of touchscreens is to eliminate the need for a physical keyboard by displaying a virtual one responding to taps on the screen.

Touchscreens often respond not only to the location of the touch, but also the

motion of the touching object. For example, a flicking motion with a finger may cause a window to scroll. Multi-touch capability, now increasingly common, makes possible for the device to respond to gestures with more than one finger, e.g., pinching and spreading with two fingers.

Data gloves: This device is used particularly in virtual reality environments which are programmed to react to the position of the gloves, the direction in which fingers are pointing, as well as to hand motion and gestures. The gloves themselves are wired to transmit not only their location, but also their configuration and orientation to the processor, so that the latter can display the environment accordingly. For example, an index finger pointing at a particular atom in a virtual-reality display of a molecule may cause this atom to zoom up to the viewer.

1.2.2 Output Devices

Again, the following list is not meant to be comprehensive, but, rather, representative of the most common output devices. We go clockwise around the outer ring of devices pictured in Figure 1.14 beginning with the rightmost.

CRT (cathode-ray tube) monitor: Though now nearly obsolete CRT monitors are worth a review because they were the first visual output devices, for which reason their configuration greatly influenced the development of foundational graphics algorithms.

A CRT monitor has phosphors of the three primary colors – R(ed), G(reen) and B(lue), the basis of CG color application – located at each one of a rectangular array of pixels, called the raster. Additionally, it has three electron guns inside, causing its infamous bulk, that each fires a beam at phosphors of one color. A mechanism to aim and control their intensities causes the beams to travel together, striking one pixel after another, row after row, exciting the RGB phosphors at each pixel to the values specified for it in the color buffer. Figure 1.13(a) shows the electron beams striking one pixel on a dog.

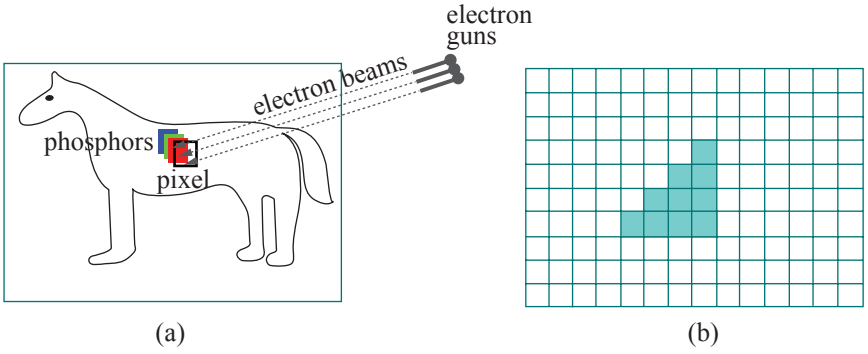

Figure 1.13: (a) Color CRT monitor with electron beams aimed at a pixel with phosphors of the 3 primaries (b) A raster of pixels showing a rasterized triangle.

From the point of view of OpenGL and most CG theory, what matters is that the pixels in a monitor are arranged in a rectangular raster (as depicted in Figure 1.13(b)). For, this layout is the basis of the lowest-level CG algorithms, the so-called raster algorithms, which actually select and color the pixels to represent user-specified shapes such as lines and triangles on the monitor. Figure 1.13(b), for example, shows the rasterization of a right-angled triangle (with terrible jaggies because of the low resolution).

The number of rows and columns of pixels in the raster determines the monitor's resolution. Typical for a CRT monitor is a resolution in the range of 1024×768 (which means 1024 columns and 768 rows). High-definition monitors (as needed,

say, for high-definition TV, or HDTV as it's acronymed) have higher resolution, e.g., 1920×1080 is common.

Figure 1.14: Output devices clockwise from the rightmost (surrounding processing devices in the middle): CRT monitor, LCD monitor, notebook, mobile phone, 3D displays including a VR headset and stereoscopic glasses tethered to a PC.

Moreover, a memory location called the color buffer, either in the CPU or graphics card, contains, typically, 32 bits of data per raster pixel – 8 bits for each of RGB, and 8 for the alpha value (used in blending). It is the RGB values in the color buffer which determine the corresponding raster pixel's color intensities. The values in the color buffer are read by the raster – in other words, the raster is refreshed – at a rate called the monitor's refresh rate. Beyond this, the technology underlying the particular display device, no matter how primitive or fancy, really matters little to the CG programmer.

For decades a bulky CRT monitor, or two, was a fixture atop work tables. Now, of course, they have been nearly totally supplanted by a sleeker successor which we discuss next.

LCD (liquid crystal display) monitor: Pixels in an LCD monitor each consist of three subpixels made of liquid crystal molecules, which separately filter lights of the primary colors. The amount of light, coming from a backlight source, emerging through a subpixel is controlled by an electric charge whose intensity is determined by subpixel's corresponding value in the color buffer. The absence of electron guns allows LCD monitors to be made flat and thin – unlike CRT monitors – so they are one of the class of flat panel displays.

Technologies other than LCD, e.g., plasma and OLED (organic light emitting diode), are used as well in flat panel displays, though LCD is by far the most common one found with computers. Keep in mind that what's called an LED monitor is simply a kind of LCD monitor where the backlighting comes from array of light-emitting diodes (LEDs) - in fact, almost all LCD monitors nowadays have this kind of backlighting, so would be classified as LED monitors.

Again, for a CG programmer the view to keep in mind of an LCD monitor, as of any other flat panel display, is as a rectangular raster of pixels whose RGB intensities are individually set by values in the computer's color buffer.

Portable computer display: This display is again a raster of pixels whose RGB values are read from a color buffer. The technology employed, typically, is TFT-LCD, a variant of LCD which uses thin film transistors to improve image quality.

Handheld display: Handheld displays, such as those on devices like mobile phones, commonly use the same TFT-LCD technology as portable computers. The resolution, though, is necessarily smaller, e.g., 480×640 would be in the ballpark for low-end mobiles.

3D display: Almost all 3D displays are based on the principle of stereoscopy, in which an illusion of depth is created by showing either eye of the viewer images of the scene captured by one of two cameras slightly offset from one another (just like a pair of eyes as in Figure 1.15). Once the scene has been recorded with two cameras, it is in ensuring that each eye of the viewer sees frames only from one of them, called stereoscopic viewing, that there are primarily two competing technologies.

Figure 1.15: An early model stereo camera filming a newish bike.

In the first, frames alternately from either camera are displayed on the monitor, a process called alternate frame sequencing. The viewer herself wears stereoscopic glasses, each lens embedded with a layer of liquid crystals which can be darkened with an electrical signal (such glasses are also called LC shutter glasses). The glasses are synchronized with the monitor's refresh rate, either lens being alternately darkened with successive frames. Consequently, each eye sees images from only one of the two cameras, resulting in a stereoscopic effect. Typically, the frame rate is increased to 48 per second as well, so that both eyes experience a smooth-seeming 24 frames each second. The great advantage of LC shutter glasses is that they can be used with any computer which has a monitor with a refresh rate fast enough to support alternate frame sequencing, as well as a graphics card with enough buffer space for two video streams. So with these glasses even a high-end home system would qualify to play 3D movies and games.

In the second, polarized 3D glasses are used to view two images, from either camera, projected simultaneously on the same screen through orthogonal polarizing filters. The lenses too contain orthogonal polarizing filters, each allowing through only light of like polarization. Consequently, either lens sees images from only one or other camera, engendering a stereoscopic view. Polarized 3D glasses are significantly less expensive than LC shutter glasses and, moreover, require no synchronization with the monitor. However, the projection system is complicated and expensive and primarily used to equip theaters for 3D viewing.

VR headsets, increasing popular nowadays for an immersive 3D experience, are each a two-stream projection system with twin-lens viewing all in one, which can use either of the above technologies.

OpenGL, the API we'll be using, is well-suited to making scenes and movies for 3D viewing because it allows multiple (virtual) cameras to be positioned arbitrarily.

1.3 Quick Preview of the Adventures Ahead

To round out this invitation to CG we want to show you four programs written by students in their first college 3D CG course, taught by the author at the Asian Institute of Technology using a draft of this book. They were written in C++ with calls to OpenGL.

But, first, what exactly is OpenGL? You may have been wondering this awhile. We said earlier in the section on CG history that OpenGL is a cross-platform 3D

graphics API. It consists of a library of over 500 commands to perform 3D tasks, which can be accessed from programs written in various languages. Well, here's a glimpse of something concrete – an example snippet from a C++ environment to draw 10 red points:

```
glColor3f(1.0, 0.0, 0.0);
glBegin(GL_POINTS);
   for(int i = 0; i < 10; i++)
   {
      glVertex3i(i, 2*i, 0);
   }
glEnd();
```

The first function call `glColor3f(1.0, 0.0, 0.0)` declares the red drawing color, while the loop bracketed between the `glBegin(GL_POINTS)` and `glEnd()` calls draws a point at $(i, 2i, 0)$ in each of ten iterations. There are calls in the OpenGL library to draw straight lines, triangles, create light sources, apply textures, move and rotate objects, maneuver the camera, and so on – in fact, not surprisingly, pretty much all one needs to create and animate 3D scenes.

In the early days, coding CG meant pretty much addressing individual pixels of the raster to turn on and off. Compared with using OpenGL today this is like programming in assembly compared with programming in a high-level language like C++. OpenGL gives an environment where we can focus on imagining and creating, leaving low-level work all to the underlying engine.

Isn't that old OpenGL, though, the code you show above? Yes, it is. Precisely, it's pre-shader OpenGL.

But, the fact you are asking this question probably means you are not familiar yet with our pedagogical approach, which is described in the book's preface. We explain there why we believe in setting the reader's foundations in the classical version of OpenGL before proceeding to the new (namely, fourth generation or 4.x) of which there is complete coverage later in the book. We urge you to read at least that part of the preface in order to be comfortable with how we plan on doing things.

Getting back to the student programs, code itself is not of importance and would actually be a distraction at this time. Instead, just running the programs and viewing the output will give an idea of what can be accomplished even in a fairly short time (ranging from 3 weeks to 3 months for the different programs) by persons coming to CG with little more than a good grasp of C++ and some basic math. Of course, we'll get a feel as well for what goes into making 3D scenes.

Experiment 1.1. Open `ExperimenterSource/Chapter1/Ellipsoid` and run the (Windows) executable there for the `Ellipsoid` program.* The program draws an ellipsoid (an egg shape). The left of Figure 1.16 shows the initial screen. There's plenty of interactivity to try as well. Press any of the four arrow keys, as well as the page up and down keys, to change the shape of the ellipsoid, and 'x', 'X', 'y', 'Y', 'z' and 'Z' to turn it.

It's a simple object, but the three-dimensionality of it comes across rather nicely does it not? As with almost all surfaces that we'll be drawing ourselves, the ellipsoid is made up of triangles. To see these press the space bar to enter wireframe mode. Pressing space again restores the filled mode. Wireframe reveals the ellipsoid to be a mesh of triangles decorated with large points. A color gradient has apparently been applied toward the poles as well.

Drawing an ellipsoid with many triangles may seem a hard way to do things. Interestingly, and often surprisingly for the beginner, OpenGL offers the programmer

*`Experimenter.pdf` does not have clickable links to run the executables for this chapter. Clickable links to bring up source code and project files start from the next chapter.

only a tiny set of low-level geometric primitives with which to make objects – in fact, points, lines and triangles are, basically, it. So, a curved 3D object like an ellipsoid has to be made, or, more accurately, *approximated*, using triangles. But, as we shall see as we go along, the process really is not difficult.

That's it, there's really not much more to this program. It's just a bunch of colored triangles and points laid out in 3D space. The magic is in those last two words: *3D space*. 3D modeling is all about making things in 3D – not a flat plane – to create an illusion of depth, even when viewing on a flat plane (the screen). **End**

(a) (b) (c) (d)

Figure 1.16: Screenshots of (a) `Ellipsoid` (b) `AnimatedGarden` (c) `BezierShoe` (d) `Dominos`.

Experiment 1.2. Our next program is animated. It creates a garden which grows and grows and grows. You will find the executable in `ExperimenterSource/-Chapter1/AnimatedGarden`. Press enter to start the animation; enter again to stop it. The delete key restarts the animation, while the period key toggles between the camera rotating and not. Again, the space key toggles between wireframe and filled. The second image of Figure 1.16 is a screenshot a few seconds into the animation.

As you can see from the wireframe, there's again a lot of triangles (in fact, the flowers might remind you of the ellipsoid from the previous program). The plant stems are thick lines and, if you look carefully, you'll spot points as well. The one special effect this program has that `Ellipsoid` did not is blending, as is not hard to see. **End**

Experiment 1.3. The third program shows fairly fancy object modeling. It makes a lady's shoe with the help of so-called Bézier patches. The executable is in `ExperimenterSource/Chapter1/BezierShoe`. Press 'x'-'Z' to turn the shoe, '+' and '-' to zoom in and out, and the space bar to toggle between wireframe and filled. The third image of Figure 1.16 is a screenshot.

Keep in mind that, as far as CG goes, the *techniques* involved in designing a shoe are the same as for designing a spaceship! **End**

Experiment 1.4. Our final program is a movie which shows a Rube Goldberg domino effect with "real" dominos. The executable is in `ExperimenterSource/-Chapter1/Dominos`. Simply press enter to start and stop the movie. The screenshot on the right of Figure 1.16 is from part way through.

This program has a bit of everything – textures, lighting, camera movement and, of course, a nicely choreographed animation sequence. Neat, is it not? **End**

All fired up now and ready to rumble? Great! Let's go.

Acknowledgments: Kumpee Teeravech wrote `Ellipsoid` and `AnimatedGarden`, Pongpon Nilaphruek wrote `BezierShoe`, while Kanit Tangkathach and Thanapoom Veeranitinunt wrote `Dominos`.

Chapter 2

On to OpenGL and 3D Computer Graphics

The primary goal for this chapter is to be acquainted with OpenGL and begin our journey into computer graphics using OpenGL as the API (Application Programming Interface) of choice. We shall apply an experiment-discuss-repeat approach where we run code and ask questions of what is seen, acquiring thereby an understanding not only of the way the API functions, but underlying CG concepts as well. Particularly, we want to gain insight into:

(a) The synthetic-camera model to record 3D scenes, which OpenGL implements.

(b) The approach of approximating curved objects, such as circles and spheres, with the help of straight and flat geometric primitives, such as line segments and triangles, which is fundamental to object design in computer graphics.

We begin in Section 2.1 with our first OpenGL program to draw a square, the computer graphics equivalent of "Hello World". Simple though it is, with a few careful experiments and their analysis, `square.cpp` yields a surprising amount of information through Sections 2.1-2.3 about orthographic projection, the fixed world coordinate system OpenGL sets up and how the so-called viewing box in which the programmer draws is specified in this system. We gain insight as well into the 3D-to-2D rendering process.

Adding code to `square.cpp` we see in Section 2.4 how parts of objects outside the viewing box are clipped off. Section 2.5 discusses OpenGL as a state machine. We have in this section as well our first glimpse of property values, such as color, initially specified at the vertices of a primitive, being interpolated throughout its interior.

Next is the very important Section 2.6 where all the drawing primitives of OpenGL are introduced. These are the parts at the application programmer's disposal with which to assemble objects from thumbtacks to spacecrafts.

The first use of straight primitives to approximate a curved object comes in Section 2.7: a curve (a circle) is drawn using straight line segments. To create more interesting and complex objects one must invoke OpenGL's famous three-dimensionality. This means learning first in Section 2.8 about perspective projection as also hidden surface removal using the depth buffer.

After a bunch of drawing exercises in Section 2.9 for the reader to practice her newly-acquired skills, the topic of approximating curved objects is broached again in Section 2.10, this time to approximate a surface with triangles, rather than a curve with straight segments as in Section 2.7. Section 2.11 is a review of all the syntax that goes into making a complete OpenGL program.

We conclude with a summary, brief notes and suggestions for further reading in Section 2.12.

2.1 First Program

Figure 2.1: Screenshot of `square.cpp`, in particular, the OpenGL window.

Experiment 2.1. Run `square.cpp`.

Note: Visit the book's website `www.sumantaguha.com` for a guide how to install OpenGL and run our programs. *Importantly*, we have changed our development environment significantly for this edition so programs for the second edition will not run off the bat in the environment for this one and vice versa. The needed change in code itself is fairly trivial though. The install guide should make everything clear.

In the OpenGL window appears a black square over a white background. Figure 2.1 is an actual screenshot, but we'll draw it as in Figure 2.2, bluish green standing in for white in order to distinguish it from the paper. We are going to understand next how the square is drawn, and gain some insight as well into the workings behind the scene.

End

Figure 2.2: OpenGL window of `square.cpp` (bluish green pretending to be white).

The following six statements in `square.cpp` create the square:

```
glBegin(GL_POLYGON);
   glVertex3f(20.0, 20.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
   glVertex3f(80.0, 80.0, 0.0);
   glVertex3f(20.0, 80.0, 0.0);
glEnd();
```

Remark 2.1. *Important*! If, from what you may have read elsewhere, you have the notion that `glBegin()-glEnd()`, and even `GL_POLYGON`, specifications are classical and don't belong in the newest version of OpenGL, then you are right insofar as they are not in the core profile of the latter. They are, though, accessible via the compatibility profile which allows for backward compatibility. Moreover, we explain carefully in the book's preface why we don't subscribe to the *toss-everything-classical* school of thought as far as *teaching* OpenGL is concerned. Of course, we shall cover thoroughly the most modern – in fact, fourth generation – OpenGL later in the book. If you have not done so yet, we strongly urge you to read about our pedagogical approach in the preface in order to be comfortable with what follows.

The corners of the square evidently are specified by the four vertex declaration statements between `glBegin(GL_POLYGON)` and `glEnd()`. Let's determine how exactly these `glVertex3f()` statements correspond to the corners.

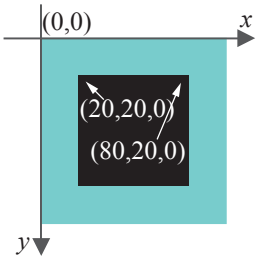

Figure 2.3: The coordinate axes on the OpenGL window of `square.cpp`? *No*.

If, suppose, the vertices are specified in some coordinate system that is embedded in the OpenGL window – which certainly is plausible – and if we knew the axes of this system, the matter would be simple. For example, *if* the x-axis increased horizontally rightwards and the y-axis vertically downwards, as in Figure 2.3, then `glVertex3f(20.0, 20.0, 0.0)` would correspond to the upper-left corner of the square, `glVertex3f(80.0, 20.0, 0.0)` to the upper-right corner, and so on.

However, even assuming that there do exist these invisible axes attached to the OpenGL window, how do we tell where they are or how they are oriented? One way is to "wiggle" the corners of the square! For example, change the first vertex declaration from `glVertex3f(20.0, 20.0, 0.0)` to `glVertex3f(30.0, 20.0, 0.0)` and observe which corner moves. Having determined in this way the correspondence of the corners with the vertex statements, we ask the reader to deduce the orientation of the hypothetical coordinate axes. Decide where the origin is located too.

Well, it seems then that `square.cpp` sets up coordinates in the OpenGL window so that the increasing direction of the x-axis is horizontally rightwards, that of the y-axis vertically upwards and, moreover, the origin seems to correspond to the lower-left

corner of the window, as in Figure 2.4. We're making progress but there's more to the story, so read on.

The last of the three parameters of a `glVertex3f(*, *, *)` declaration is evidently the z coordinate. Vertices are specified in *3-dimensional* space (simply called 3-space or, mathematically, $\mathbb{R}^3$). Indeed, OpenGL allows us to draw in 3-space and create truly 3D scenes, which is its major claim to fame. However, we *perceive* the 3-dimensional scene as a picture *rendered* to a 2-dimensional part of the computer's screen, in particular, the rectangular OpenGL window. Shortly we'll see how OpenGL converts a 3D scene to its 2D rendering.

2.2 Orthographic Projection, Viewing Box and World Coordinates

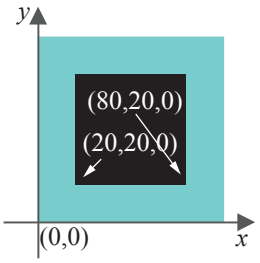

Figure 2.4: The coordinate axes on the OpenGL window of `square.cpp`? *Well, pretty much, but there's a bit more to it.*

What exactly do the vertex coordinate values mean? For example, is the vertex at (20.0, 20.0, 0.0) of `square.cpp` 20 mm., 20 cm. or 20 pixels away from the origin along both the x-axis and y-axis, or is there some other absolute unit of distance native to OpenGL? Let's do the following experiment.*

Experiment 2.2. The main routine's `glutInitWindowSize()` parameter values determine the shape of the OpenGL window; in fact, generally, `glutInitWindowSize(`w`, `h`)` creates a window w pixels wide and h pixels high.

Change `square.cpp`'s initial `glutInitWindowSize(500, 500)` to `glutInitWindowSize(300, 300)` and then `glutInitWindowSize(500, 250)` (Figure 2.5). The drawn square changes in size, and even shape, with the OpenGL window. Therefore, coordinate values of the square appear not to be in any kind of absolute units on the screen. **End**

Figure 2.5: Screenshot of `square.cpp` with window size 500 × 250.

Remark 2.2. Of course, you could have reshaped the OpenGL window directly by dragging one of its corners with the mouse, rather than resetting `glutInitWindowSize()` in the program.

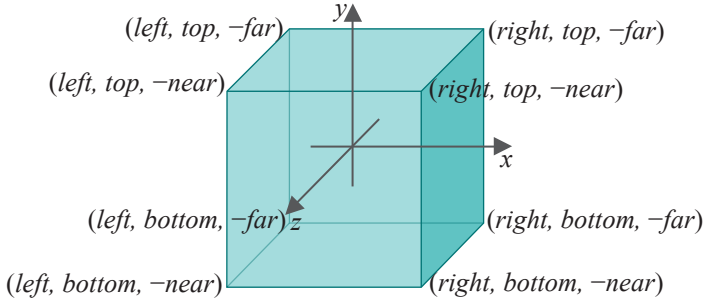

Figure 2.6: Viewing box defined by `glOrtho(`*left*`, `*right*`, `*bottom*`, `*top*`, `*near*`, `*far*`)`.

Understanding what the coordinates actually represent involves understanding first OpenGL's rendering mechanism, which itself begins with the program's *projection statement*. In the case of `square.cpp` this statement is

```
glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0)
```

in the `resize()` routine, which determines an imaginary (or, virtual, if you like) *viewing box* inside which the programmer draws scenes. Generally,

`glOrtho(`*left*`, `*right*`, `*bottom*`, `*top*`, `*near*`, `*far*`)`

*Experiments are an integral part of our teaching method so we urge you to run them as you read. The file *Experimenter.pdf* at the book's website makes this easy by allowing you to open the project file for successive experiments with a single click each. However, even if you don't run an experiment, make sure always to read its discussion in the text.