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

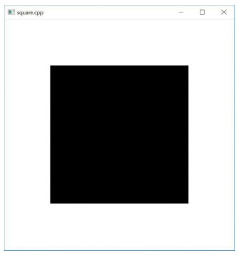

Figure 2.1: Screenshot of `square.cpp`, in particular, the OpenGL window.

Experiment 2.1. Run `square.cpp`.

Note: Visit the book's website `www.sumantaguha.com` to download all of the book's code in the folder `ExperimenterSource`, as well as a guide to how to install OpenGL and run the programs. Available for download too is `Experimenter.pdf` which lists the programs in their book order and conveniently allows the user to click to open each.

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.

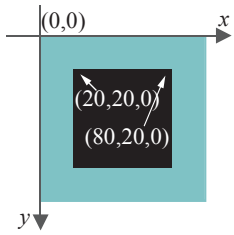

Figure 2.3: The coordinate axes on the OpenGL window of `square.cpp`? *No*.

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. We'll soon see how OpenGL converts a 3D scene to its 2D rendering.

2.2 Orthographic Projection, Viewing Box and World Coordinates

Section 2.2 Orthographic Projection, Viewing Box and World Coordinates

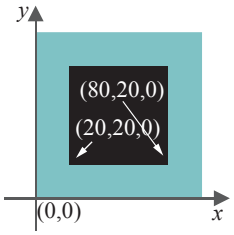

Figure 2.4: The coordinate axes on the OpenGL window of `square.cpp`? *Well, pretty much, but there's a bit more to it.*

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, its code coordinate values appear not to mean any kind of absolute screen units. End

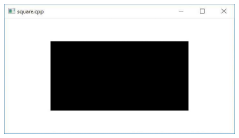

Figure 2.5: Screenshot of `square.cpp` with window size 500 × 250.

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.

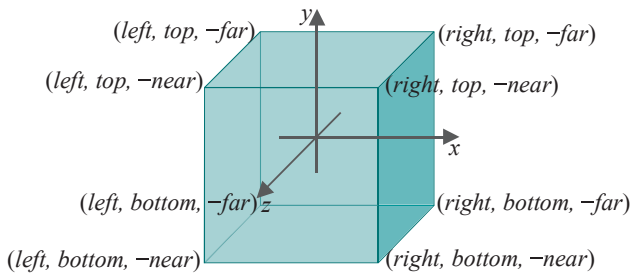

Figure 2.6: Viewing box defined by `glOrtho(`*left*`, `*right*`, `*bottom*`, `*top*`, `*near*`, `*far*`)` sitting in world space whose *xyz* coordinate axes are drawn.

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,

*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.

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

sets up a viewing box, as in Figure 2.6, with corners at the 8 points:

(*left*, *bottom*, −*near*), (*right*, *bottom*, −*near*), (*left*, *top*, −*near*), (*right*, *top*, −*near*), (*left*, *bottom*, −*far*), (*right*, *bottom*, −*far*), (*left*, *top*, −*far*), (*right*, *top*, −*far*)

It's a box with sides aligned along the axes, whose span along the x-axis is from *left* to *right*, along the y-axis from *bottom* to *top*, and along the z-axis from $-near$ to $-far$. Note the little quirk of OpenGL that the *near* and *far* values are flipped in sign. The viewing box corresponding to the projection statement `glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0)` of `square.cpp` is shown in Figure 2.7(a).

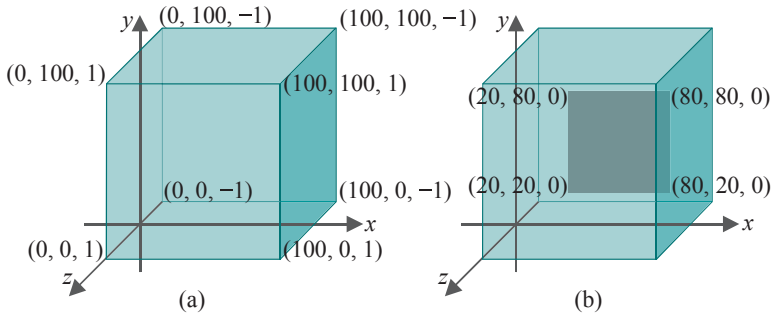

Figure 2.7: (a) Viewing box of `square.cpp` (b) With the square drawn inside.

The viewing box itself is located in a 3D space, also virtual, called *world space*, by means of the coordinate system of that space (Figure 2.6 shows the coordinate axes of world space). The reader uncomfortable with all of this "virtuality" is welcome to make matters real by taking world space to be the room she is in with coordinate axes, say, meeting somewhere in the middle.

How then are the coordinate axes of world space calibrated, for the size of the viewing box depends on this? E.g., is a unit along an axis one inch, one centimeter or something else? The answer will be evident once the rendering process is explained momentarily but, again for the sake of concreteness, the reader can for now fix a unit to be an inch. So, the picture of a "real" (say, glass) viewing box *right*−*left* inches wide, *top*−*bottom* inches high and *far*−*near* inches deep located in a room is perfectly good.

As for drawing now, the vertex declaration `glVertex3f(x, y, z)` corresponds to the point (x, y, z) in world space. For example, the corner of the square declared by `glVertex3f(20.0, 20.0, 0.0)` is at (20.0, 20.0, 0.0). The square of `square.cpp` then, as depicted in Figure 2.7(b), is located entirely inside the viewing box.

Once the programmer has drawn the entire scene, if the projection statement is `glOrtho()` as in `square.cpp`, then the rendering process is two-step:

1. *Shoot*: First, objects are *projected perpendicularly* onto the front face of the viewing box, i.e., the face on the $z = -near$ plane. For example, the square in Figure 2.8(a) (same as Figure 2.7(b)) is projected as in Figure 2.8(b). The front face of the viewing box is called the *viewing face* and the plane on which it lies the *viewing plane*. This step is like shooting the scene on film.

2. *Print*: Next, the viewing face is *proportionately scaled* to fit the rectangular OpenGL window. This step is like printing film on paper. In the case of `square.cpp`, printing takes us from Figure 2.8(b) to (c).

   If, say, the window size of `square.cpp` were changed to one of *aspect ratio* (= width/height) of 2, by replacing `glutInitWindowSize(500, 500)`

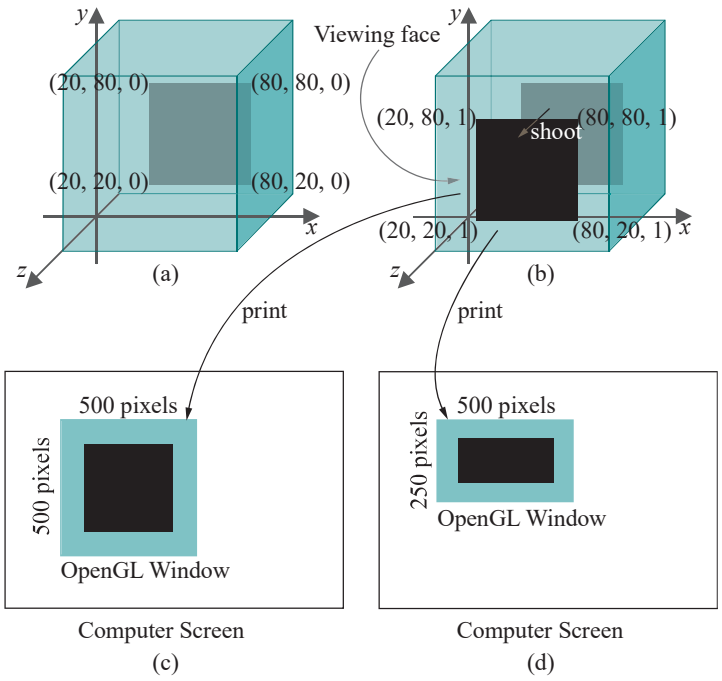

Figure 2.8: Rendering with `glOrtho()`.

with `glutInitWindowSize(500, 250)`, printing would take us from Figure 2.8(b) to (d) (which actually distorts the square into a rectangle).

The answer to the earlier question of how to calibrate the coordinate axes of world space should be clear now: *it does not matter*! For, the 2D rendering finally displayed is the same *no matter* how the axes are calibrated, because of the proportionate scaling of the viewing face of the box to match the OpenGL window. And, of course, this is why OpenGL never prompts us for information about our world space and how it's coordinatized – it doesn't need to for it to do its job. Here's a partly-solved exercise to drive home the point.

Exercise 2.1.

(a) Suppose the viewing box of `square.cpp` is set up in a world space where one unit along each axis is 1 cm. Assuming pixels to be 0.2 mm. × 0.2 mm. squares, compute the size and location of the square rendered by shoot-and-print to a 500 pixel × 500 pixel OpenGL window.

(b) Suppose next that the coordinate system of the world space is re-calibrated so that a unit along each axis is 1 meter instead of 1 cm., everything else remaining same. What then are the size and location of the rendered square in the OpenGL window?

(c) What is rendered if, additionally, the size of the OpenGL window is changed to 500 pixel × 250 pixel?

Part answer:

(a) Figure 2.9 on the left shows the square projected to the viewing face, which is 100 cm. square. The viewing face is then scaled to the OpenGL window on the right, which is a square of sides 500 pixels = 500 × 0.2 mm. = 100 mm. Scaling from face to the window, therefore, is a factor of 1/10 in both dimensions.

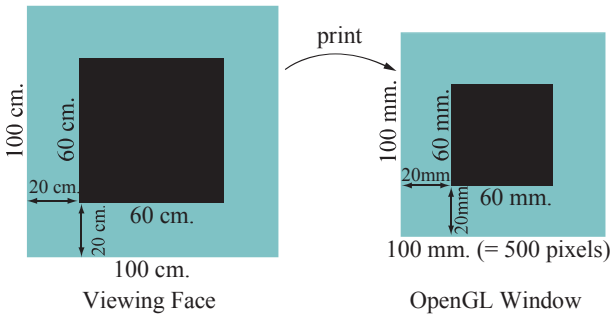

Figure 2.9: The viewing face for `square.cpp`, given that one unit along each coordinate axis is 1 cm., scaled to a 500 pixel × 500 pixel OpenGL window.

It follows that the rendered square is 60 mm. × 60 mm., with its lower-left corner located both 20 mm. above and to the right of the lower-left corner of the window.

(b) Exactly the same as in part (a) because, while the viewing box and viewing face are now 100 times larger in both the x and y dimensions, the scaling from face to window is now a factor of 1/1000, rather than 1/10.

We conclude that the size and location of the rendering in each coordinate direction are independent of how the axes are calibrated, but determined rather by the *ratio* of the original object's size to that of the viewing box in that direction.

Although the calibration of the world space axes doesn't matter, nevertheless, we'll make the sensible assumption that all three are calibrated *identically*, i.e., one unit along each axis is of equal length (yes, oddly enough, we could make them different and still the rendering would not change, which you can verify yourself by re-doing Exercise 2.1(a), after assuming that one unit along the x-axis is 1 cm. and along the other two 1 meter). The only other assumptions about the initial coordinate system which we make are conventional ones:

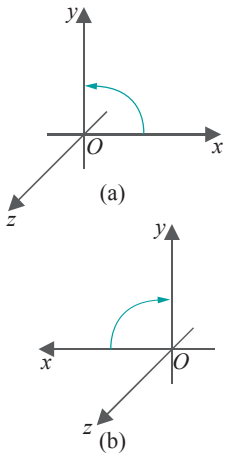

Figure 2.10: The x-, y- and z-axes are rectangular and form (a) a right-handed system (b) a left-handed system.

(a) It is *rectangular*, i.e., the three axes are mutually perpendicular.

(b) The x-, y- and z-axes in that order form a *right-handed* system in the following sense: a rotation of the x-axis 90° about the origin so that its positive direction matches with that of the y-axis appears *counter-clockwise* to a viewer located on the positive side of the z-axis (Figure 2.10).

Fixed World System

To summarize, set up an initial rectangular right-handed coordinate system with axes all calibrated identically. Call a unit along each axis just "a unit" as we know it doesn't matter what exactly this unit is. Imagine the system located wherever you like – on top of your desk maybe – and then leave it fixed.

See Figure 2.11 – it actually helps to imagine this system of axes as real and fixed forever. The system coordinatizes world space and, in fact, we shall refer to it as the *world coordinate system*. All objects, including the viewing box and those that we create ourselves, inhabit world space and are specified in world coordinates. These are all virtual objects, of course.

Remark 2.3. Even though the world coordinate system can be located wherever you like, it is most often helpful to imagine the x-axis running rightward along the bottom of your monitor, the y-axis climbing up the left side, and the z-axis coming at you.

Remark 2.4. Because it's occupied by user-defined objects, world space is sometimes called *object space*.

Figure 2.11: A dedicated 3D graphics programmer in a world all her own.

Incidentally, it's clear now that our working hypothesis after the first experiment in Section 2.1, that the OpenGL window comes with axes fixed to it, though not unreasonable, was not entirely accurate. The OpenGL window it turns out is simply an empty target rectangle on which the front face of the viewing box is printed. This rectangle is called *screen space*.

So, there are two spaces we'll be working with: world and screen. The former is a virtual 3D space in which we create our scenes, while the latter is a real 2D space where scenes are rendered in a shoot-and-print process.

Experiment 2.3. Change only the viewing box of `square.cpp` by replacing `glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0)` with `glOrtho(-100, 100.0, -100.0, 100.0, -1.0, 1.0)`. The location of the square in the new viewing box is different and, so as well, the result of shoot-and-print. Figure 2.12 is a screenshot and Figure 2.13 explains how. **End**

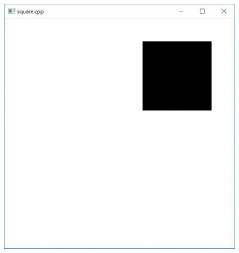

Figure 2.12: Screenshot of `square.cpp` with `glOrtho(-100, 100.0, -100.0, 100.0, -1.0, 1.0)`.

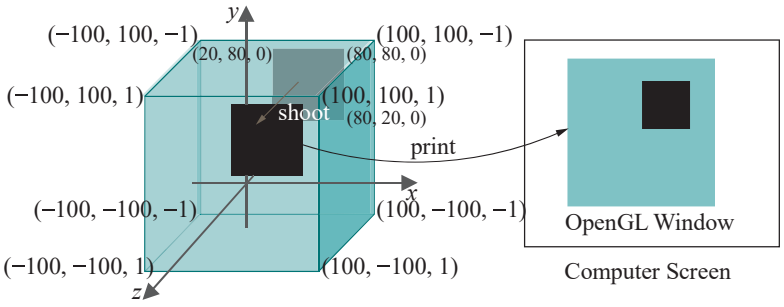

Figure 2.13: The viewing box of `square.cpp` defined by `glOrtho(-100, 100.0, -100.0, 100.0, -1.0, 1.0)`.

Exercise 2.2. (Programming) Change the viewing box of `square.cpp` by replacing `glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0)` successively with the following, in each case trying to predict the output before running:

(a) `glOrtho(0.0, 200.0, 0.0, 200.0, -1.0, 1.0)`

(b) `glOrtho(20.0, 80.0, 20.0, 80.0, -1.0, 1.0)`

(c) `glOrtho(0.0, 100.0, 0.0, 100.0, -2.0, 5.0)`

Exercise 2.3. If the viewing box of `square.cpp` is changed by replacing `glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0)` with `glOrtho(-100.0, 100.0, -100.0, 100.0, -1.0, 1.0)`, and the OpenGL window size changed replacing `glutInitWindowSize(500, 500)` with `glutInitWindowSize(500, 250)`, then calculate the area (in *number of pixels*) of the image of the square.

Exercise 2.4. (Programming) We saw earlier that, as a result of the print step, replacing `glutInitWindowSize(500, 500)` with `glutInitWindowSize(500, 250)` in `square.cpp` causes the square to be distorted into a rectangle. By changing *only one* numerical parameter elsewhere in the program, eliminate the distortion to make it appear square again.

Exercise 2.5. (Programming) Alter the z coordinates of each vertex of the "square" – we should really call it a polygon if we do this – of `square.cpp` as follows:

```
glBegin(GL_POLYGON);
   glVertex3f(20.0, 20.0, 0.5);
   glVertex3f(80.0, 20.0, -0.5);
   glVertex3f(80.0, 80.0, 0.1);
   glVertex3f(20.0, 80.0, 0.2);
glEnd();
```

The rendering does not change. Why?

Remark 2.5. Always set the parameters of `glOrtho(`*left*`, `*right*`, `*bottom*`, `*top*`, `*near*`, `*far*`)` so that *left* < *right*, *bottom* < *top*, and *near* < *far*. However, we'll revisit this edict in Problem 2.11.

Remark 2.6. The aspect ratio (= width/height) of the viewing box should be set same as that of the OpenGL window or the scene will be distorted by the print step.

Remark 2.7. The perpendicular projection onto the viewing plane corresponding to a `glOrtho()` call is also called *orthographic projection* or *orthogonal projection* (hence the name of the call). Yet another term is *parallel projection* as the lines of projection from points in the viewing box to the viewing plane are all parallel.

2.3 The OpenGL Window and Screen Coordinates

We've already had occasion to use the `glutInitWindowSize(`w`, `h`)` command which sets the size of the OpenGL window to width w and height h measured in pixels. A companion command is `glutInitWindowPosition(`x`, `y`)` to specify the location (x, y) of the upper-left corner of the OpenGL window on the computer screen.

Experiment 2.4. Change the parameters of `glutInitWindowPosition(`x`, `y`)` in `square.cpp` from the current (100, 100) to a few different values to determine the location of the origin (0, 0) of the computer screen, as well as the orientation of the screen's own x-axis and y-axis. End

The origin $(0, 0)$ of the screen it turns out is at its upper-left corner, while the increasing direction of its x-axis is horizontally rightwards and that of its y-axis vertically downwards; moreover, one unit along either axis is *absolute* and represents a pixel. See Figure 2.14, which shows as well the coordinates of the corners of the OpenGL window as initialized by `square.cpp`.

Note the inconsistency between the orientation of the screen's y-axis and the y-axis of the world coordinate system, the latter being directed *up* the OpenGL window (after being projected there). One needs to take this into account when reading data from the screen and using it in world space, or vice versa. We'll see this when programming the mouse in the next chapter.

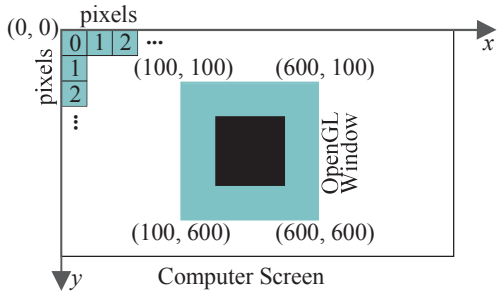

Figure 2.14: The screen's coordinate system: a unit along either axis is the pitch of a pixel.

2.4 Clipping

A question may have come to the reader's mind about objects which happen to be drawn outside the viewing box. Here are a few experiments to clarify how they are processed.

Experiment 2.5. Add another square by inserting the following right after the code for the original square in `square.cpp`:

```
glBegin(GL_POLYGON);
   glVertex3f(120.0, 120.0, 0.0);
   glVertex3f(180.0, 120.0, 0.0);
   glVertex3f(180.0, 180.0, 0.0);
   glVertex3f(120.0, 180.0, 0.0);
glEnd();
```

From the value of its vertex coordinates the second square evidently lies entirely outside the viewing box.

If you run now there's no sign of the second square in the OpenGL window. This is because OpenGL *clips* the scene to within the viewing box before rendering, so that objects or parts of objects drawn outside are not seen. Clipping is a stage in the graphics pipeline. We'll not worry about its implementation at this time, only the effect. End

Exercise 2.6. (Programming) In the preceding experiment can you redefine the viewing box by changing the parameters of `glOrtho()` so that both squares are visible?

Experiment 2.6. For a more dramatic illustration of clipping, first replace the square of the original `square.cpp` with a triangle by deleting its last vertex; in particular, replace the polygon code with the following:

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

See Figure 2.15. Next, lift the first vertex up the z-axis by changing it to `glVertex3f(20.0, 20.0, 0.5)`; lift it further by changing its z-value to 1.5 when Figure 2.16 is a screenshot, then 2.5 and, finally, 10.0. Make sure you believe that what you see in the last three cases is indeed a triangle clipped to within the viewing box – Figure 2.17 may be helpful. End

The viewing box has six faces that each lie on a different plane and, effectively, OpenGL clips the scene off on one side of each of these six planes, accordingly called

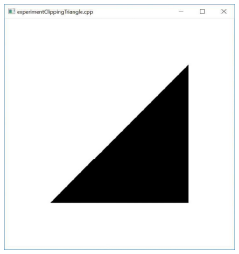

Figure 2.15: Screenshot of a triangle.

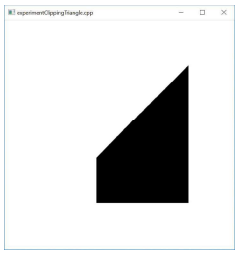

Figure 2.16: Screenshot of the triangle clipped to a quadrilateral.

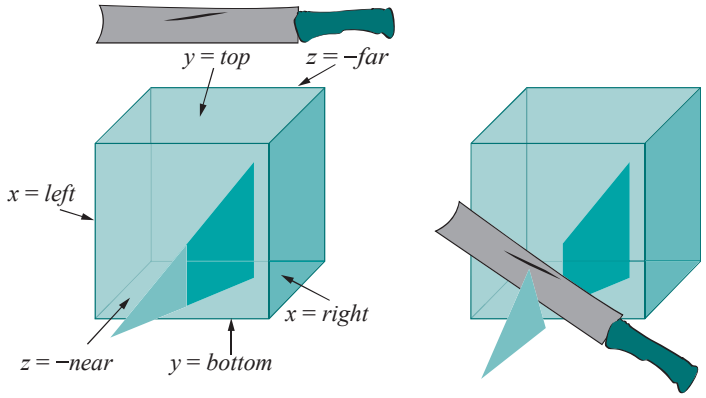

Figure 2.17: Six clipping planes of the `glOrtho(`*left*`,` *right*`,` *bottom*`,` *top*`,` *near*`,` *far*`)` viewing box and a "clipping knife".

clipping planes. One might imagine a knife slicing down each plane as in Figure 2.17. Specifically, in the case of the viewing box set up by `glOrtho(`*left*`,` *right*`,` *bottom*`,` *top*`,` *near*`,` *far*`)`, clipped off is to the left of the plane $x = left$, to the right of the plane $x = right$, and so on.

Remark 2.8. As we shall see in Chapter 3, the programmer can define clipping planes in addition to the six that bound the viewing box.

Exercise 2.7. Use pencil and paper to guess the output if the polygon declaration part of `square.cpp` is replaced with the following:

```
glBegin(GL_POLYGON);
   glVertex3f(-20.0, -20.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
   glVertex3f(120.0, 120.0, 0.0);
   glVertex3f(20.0, 80.0, 0.0);
glEnd();
```

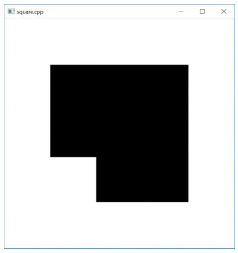

Figure 2.18: Screenshot of the first vertex of `square.cpp` raised.

Exercise 2.8. (Programming) Here's a bit of a "mathy" question. A triangle was clipped to a (4-sided) quadrilateral in the earlier experiment. Can you come up with a triangle which gets clipped to within the viewing box to a figure with more than 4 sides? What's the maximum number of sides you can make happen?

We'll leave this section with a rather curious phenomenon for the reader to resolve.

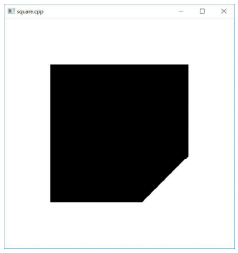

Figure 2.19: Screenshot of the second vertex of `square.cpp` raised.

Exercise 2.9. (Programming) Raising the first vertex of (the original) `square.cpp` from `glVertex3f(20.0, 20.0, 0.0)` to `glVertex3f(20.0, 20.0, 1.5)` causes it to be clipped – see Figure 2.18.

If, instead, the second vertex is raised from `glVertex3f(80.0, 20.0, 0.0)` to `glVertex3f(80.0, 20.0, 1.5)`, then the figure is clipped too, but very differently – see Figure 2.19. Why? Should not the results be similar by symmetry?
Hint: OpenGL draws a polygon after triangulating it as a so-called *triangle fan* with the *first* vertex of the polygon, in code order, the center of the fan. For example, the fan in Figure 2.20 consists of five triangles around vertex v_0.

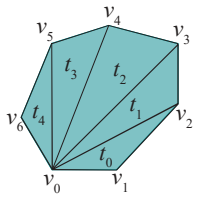

Figure 2.20: A triangle fan.

2.5 Color, OpenGL State Machine and Interpolation

Experiment 2.7. The color of the square in `square.cpp` is specified by the three parameters of the `glColor3f(0.0, 0.0, 0.0)` statement in the `drawScene()` routine,

each of which gives the value of one of the three primary components, *blue*, *green* and *red*.

Determine which of the three parameters of `glColor3f()` specifies the blue, green and red components by setting in turn each to 1.0 and the others to 0.0 (e.g., Figure 2.21 shows one case). In fact, further verify the following table for every possible combination of the values 0.0 and 1.0 for the primary components.

Call	Color
`glColor3f(0.0, 0.0, 0.0)`	Black
`glColor3f(1.0, 0.0, 0.0)`	Red
`glColor3f(0.0, 1.0, 0.0)`	Green
`glColor3f(0.0, 0.0, 1.0)`	Blue
`glColor3f(1.0, 1.0, 0.0)`	Yellow
`glColor3f(1.0, 0.0, 1.0)`	Magenta
`glColor3f(0.0, 1.0, 1.0)`	Cyan
`glColor3f(1.0, 1.0, 1.0)`	White

End

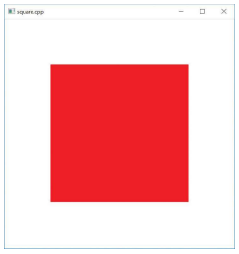

Figure 2.21: Screenshot of `square.cpp` with `glColor3f(1.0, 0.0, 0.0)`.

Generally, the `glColor3f(`*red*`,` *green*`,` *blue*`)` call specifies the *foreground color*, or *drawing color*, which is the color applied to objects being drawn. The value of each color component, which ought to be a number between 0.0 and 1.0, determines its *intensity*. For example, `glColor3f(1.0, 1.0, 0.0)` is the brightest yellow while `glColor3f(0.5, 0.5, 0.0)` is a weaker yellow.

Remark 2.9. The color values are each *clamped* to the range $[0, 1]$. This means that, if a value happens to be set greater than 1, then it's taken to be 1; if less than 0, it's taken to be 0.

Exercise 2.10. (Programming) Both `glColor3f(0.2, 0.2, 0.2)` and `glColor3f(0.8, 0.8, 0.8)` should be grays, having equal RGB intensities. Guess which is the darker of the two. Verify by changing the foreground color of `square.cpp`.

The call `glClearColor(1.0, 1.0, 1.0, 0.0)` in the `setup()` routine specifies the *background color*, or *clearing color*. Ignore for now the fourth parameter, which is the *alpha* value. The statement `glClear(GL_COLOR_BUFFER_BIT)` in `drawScene()` actually clears the window to the specified background color, which means that every pixel in the color buffer is set to that color.

Experiment 2.8. Add the additional color declaration statement `glColor3f(1.0, 0.0, 0.0)` just after the existing one `glColor3f(0.0, 0.0, 0.0)` in the drawing routine of `square.cpp` so that the foreground color block becomes

```
glColor3f(0.0, 0.0, 0.0);
glColor3f(1.0, 0.0, 0.0);
```

The square is drawn red like the one in Figure 2.21 because the *current value* or *state* of the foreground color is red when each of its vertices is specified. End

Foreground color is one of a collection of variables, called *state variables*, which determine the state of OpenGL. Among other state variables are point size, line width, line stipple, material properties, etc. We'll meet several as we go along or you can refer to the *red book** for a full list. OpenGL remains and functions in its current state until a declaration is made changing a state variable. For this reason, OpenGL is often called a *state machine*. The next couple of experiments illustrate important points about how state variables control rendering.

*The *OpenGL Programming Guide* [109] and its companion volume, the *OpenGL Reference Manual* [110], are the canonical references for the OpenGL API and affectionately referred to as the red book and blue book, respectively. Note that the on-line reference docs at `www.khronos.org` pretty much cover all that is in the blue book.

Experiment 2.9. Replace the polygon declaration part of `square.cpp` with the following to draw two squares:

```
glColor3f(1.0, 0.0, 0.0);
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();

glColor3f(0.0, 1.0, 0.0);
glBegin(GL_POLYGON);
   glVertex3f(40.0, 40.0, 0.0);
   glVertex3f(60.0, 40.0, 0.0);
   glVertex3f(60.0, 60.0, 0.0);
   glVertex3f(40.0, 60.0, 0.0);
glEnd();
```

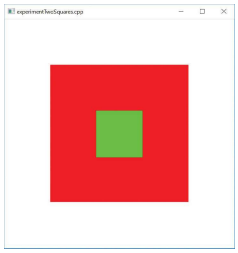

Figure 2.22: Screenshot of a green square drawn after (in the code) a red square.

A small green square appears inside a larger red one (Figure 2.22). Obviously, this is because the foreground color is red for the first square, but green for the second. One says that the color red *binds* to the first square – or, more precisely, to each of its four vertices – and green to the second square. These bound values specify the color *attribute* of either square. Generally, the values of those state variables which determine how it is rendered collectively form a primitive's attribute set.

Flip the order in which the two squares appear in the code by cutting the seven statements which specify the red square and pasting them after those to do with the green one. The green square is overwritten by the red one and no longer visible. This is because at the end of the day an OpenGL program is still a C++ program which processes code line-by-line, so objects are drawn in their *code order*. End

Experiment 2.10. Replace the polygon declaration part of `square.cpp` with:

```
glBegin(GL_POLYGON);
   glColor3f(1.0, 0.0, 0.0);
   glVertex3f(20.0, 20.0, 0.0);
   glColor3f(0.0, 1.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
   glColor3f(0.0, 0.0, 1.0);
   glVertex3f(80.0, 80.0, 0.0);
   glColor3f(1.0, 1.0, 0.0);
   glVertex3f(20.0, 80.0, 0.0);
glEnd();
```

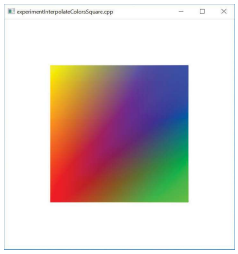

Figure 2.23: Screenshot of a square with differently colored vertices.

The different color values bound to the four vertices of the square are evidently *interpolated* over the rest of the square as you can see in Figure 2.23. In fact, this is most often the case with OpenGL: numerical attribute values specified at the vertices of a primitive are interpolated throughout its interior. In a later chapter we'll see exactly what it means to interpolate and how OpenGL goes about the task. End

Now that we have a square, as in Figure 2.23, which is not symmetric left to right or bottom to top, let's see what happens if we mess with what we were told in Remark 2.5, which was to set always the parameters of `glOrtho(`*left*`,` *right*`,` *bottom*`,` *top*`,` *near*`,` *far*`)` so that *left* < *right*, *bottom* < *top*, and *near* < *far*.

Exercise 2.11. (Programming) Return to the preceding experiment and flip the *near* and *far* values in the projection statement, precisely, change it to

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

Is there a change in what we see? Does it seem that we are viewing the square from the back so that the greenish corner is now at the lower left? *No* and *no*. The reason is that, in the shoot step, OpenGL always projects *up* the z-axis, i.e., in its positive direction. So, the viewing face is always the one with the higher z-value, which, given the `glOrtho()` statement above, effectively is now on the $z = -far = 1$ plane.

Restore the original *near* and *far* far values and flip *left* and *right*, and then *bottom* and *top*. Do you see differences? *Yes.* Can you explain them?

2.6 OpenGL Geometric Primitives

The geometric primitives – also called drawing primitives or, simply, primitives – of OpenGL are the parts that programmers use in Lego-like manner to create objects from the humblest to the incredibly complex. In fact, a thumbtack and a spacecraft would both be assembled from the same small family of OpenGL primitives. The only primitive we've seen so far though is the polygon. It's time to get acquainted with the whole family depicted in Figure 2.28.

Experiment 2.11. Replace `glBegin(GL_POLYGON)` with `glBegin(GL_POINTS)` in `square.cpp` and make the point size bigger with a call to `glPointSize(5.0)` – the default size being 1.0 – so that the part drawing an object now is

```
glPointSize(5.0); // Set point size.
glBegin(GL_POINTS);
   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();
```

See Figure 2.24. **End**

Experiment 2.12. Continue the previous experiment, replacing `GL_POINTS` with `GL_LINES`, `GL_LINE_STRIP` and, finally, `GL_LINE_LOOP`. See Figures 2.25-2.27. The thickness of lines is set by `glLineWidth(`*width*`)`. Change the parameter value of this call in the program, currently the default 1.0, to see the difference. **End**

In the explanation that follows of how OpenGL draws each of the above primitives, assume that the n vertices declared in the code between `glBegin(`*primitive*`)` and `glEnd()` are $v_0, v_1, \ldots, v_{n-1}$ in that order, i.e., the declaration of the primitive is of the form:

```
glBegin(primitive);
   glVertex3f(*, *, *); // v0
   glVertex3f(*, *, *); // v1
   ...
   glVertex3f(*, *, *); // vn-1
glEnd();
```

Refer to Figure 2.28 as you read.

GL_POINTS draws a point at each vertex

$$v_0, v_1, \ldots, v_{n-1}$$

GL_LINES draws a *disconnected* sequence of straight line segments (henceforth, we'll simply use the term "segment") between the vertices, taken two at a time. In particular, it draws the segments

$$v_0v_1,\ v_2v_3,\ \ldots,\ v_{n-2}v_{n-1}$$

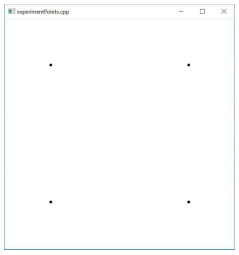

Figure 2.24: Screenshot of `square.cpp` using `GL_POINTS` instead of `GL_POLYGON`.

Figure 2.25: Screenshot of `square.cpp` using `GL_LINES`.

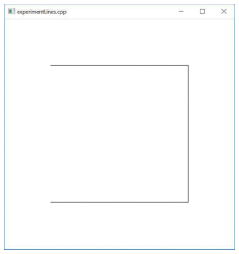

Figure 2.26: Screenshot of `square.cpp` using `GL_LINE_STRIP`.

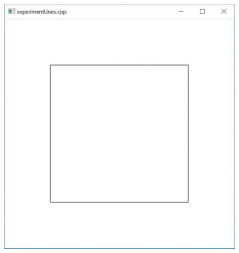

Figure 2.27: Screenshot of `square.cpp` using `GL_LINE_LOOP`.

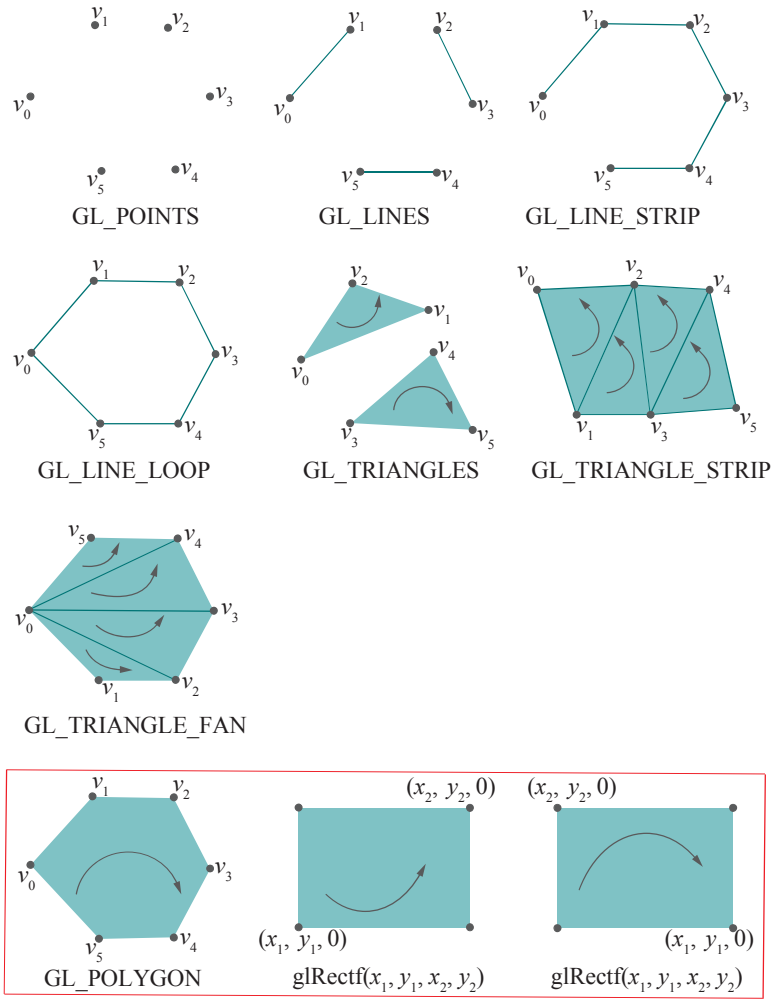

Figure 2.28: OpenGL's geometric primitives. Vertex order is indicated by a curved arrow. Primitives inside the red rectangle have been discarded from the core profile of later versions of OpenGL, e.g., 4.x; however, they are still accessible via the compatibility profile.

if n is even. If n is not even then the last vertex v_{n-1} is simply ignored.

GL_LINE_STRIP draws the *connected* sequence of segments

$$v_0v_1,\ v_1v_2,\ \ldots,\ v_{n-2}v_{n-1}$$

Such a sequence is called a *polygonal line* or *polyline*.

GL_LINE_LOOP is the same as `GL_LINE_STRIP`, *except* that an additional segment $v_{n-1}v_0$ is drawn to complete a loop:

$$v_0v_1,\ v_1v_2,\ \ldots,\ v_{n-2}v_{n-1},\ v_{n-1}v_0$$

Such a segment sequence is called a *polygonal line loop*.

Remark 2.10. In world space points actually are 0-dimensional objects having zero size, while lines are 1-dimensional objects of zero width; in fact, values specified by `glPointSize()` and `glLineWidth()` are used only for rendering, the default for both

being 1. Indeed, it would be rather hard to see a point rendered at zero size or a line at zero width!

Why does OpenGL provide separate primitives to draw polygonal lines and line loops when both can be viewed as a collection of segments and drawn using `GL_LINES`? For example,

```
glBegin(GL_LINE_STRIP);
   v0; v1; v2; v3;
glEnd();
```

is equivalent to

```
glBegin(GL_LINES);
   v0; v1; v1; v2; v2; v3;
glEnd();
```

The answer is first to avoid redundancy in vertex data. Secondly, possible rendering error is avoided as well because OpenGL does not know, for example, that the two `v1`s in the `GL_LINES` specification above are supposed to represent the same vertex, and may render them at slightly different locations because of differences in floating point round-off.

Exercise 2.12. (Programming) This relates to the brief discussion on interpolation at the end of Section 2.5. Replace the polygon declaration part of `square.cpp` with

```
glLineWidth(5.0);
glBegin(GL_LINES);
   glColor3f(1.0, 0.0, 0.0);
   glVertex3f(20.0, 20.0, 0.0);
   glColor3f(0.0, 1.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
glEnd();
```

drawing a line segment between a red and a green vertex. Can you say what the color values should be at the midpoint $(50.0, 20.0, 0.0)$ of the segment drawn? Check your answer by drawing a point with those color values just above the midpoint, say at $(50.0, 21.0, 0.0)$, with the statements

```
glPointSize(5.0);
glBegin(GL_POINTS);
   glColor3f(*, *, *);
   glVertex3f(50.0, 21.0, 0.0);
glEnd();
```

and comparing.

On to triangles next.

Experiment 2.13. Replace the polygon declaration part of `square.cpp` with:

```
glBegin(GL_TRIANGLES);
   glVertex3f(10.0, 90.0, 0.0);
   glVertex3f(10.0, 10.0, 0.0);
   glVertex3f(35.0, 75.0, 0.0);
   glVertex3f(30.0, 20.0, 0.0);
   glVertex3f(90.0, 90.0, 0.0);
   glVertex3f(80.0, 40.0, 0.0);
glEnd();
```

See Figure 2.29. End

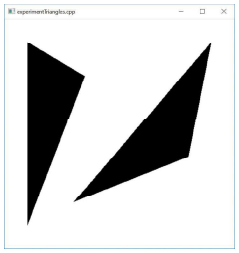

Figure 2.29: Screenshot of `square.cpp` using `GL_TRIANGLES` with new vertices.

GL_TRIANGLES draws a sequence of triangles using the vertices three at a time. In particular, the triangles are

$$v_0v_1v_2,\ v_3v_4v_5,\ \ldots,\ v_{n-3}v_{n-2}v_{n-1}$$

if n is a multiple of 3; if it isn't, the last one, or two, vertices are ignored.

The given order of the vertices for each triangle, in particular, v_0, v_1, v_2 for the first, v_3, v_4, v_5 for the second and so on, determines its *orientation* – whether clockwise (CW) or counter-clockwise (CCW) – as perceived by a viewer. Figure 2.28 indicates orientation with curved arrows (e.g., the top of the two triangles drawn for `GL_TRIANGLES` is CCW, while the bottom one CW, as perceived by a viewer at the reader's location).

The orientation of a 2D primitive, hence its vertex order, is important to specify because this enables OpenGL to decide which face, front or back, the viewer sees. We'll deal with this topic separately in Chapter 9. Till then disregard orientation when drawing.

`GL_TRIANGLES` is a *2-dimensional* primitive and, by default, triangles are drawn filled. However, one may choose a different drawing mode by applying the `glPolygonMode(`*face*`,` *mode*`)` command where *face* may be one of `GL_FRONT`, `GL_BACK` or `GL_FRONT_AND_BACK`, and *mode* one of `GL_FILL`, `GL_LINE` or `GL_POINT`. Whether a primitive is front-facing or back-facing depends, as said above, on its orientation. To keep matters simple for now, though, we'll use only `GL_FRONT_AND_BACK` in a `glPolygonMode()` call, which applies the given drawing mode to a primitive regardless of which face is visible. The `GL_FILL` option is, of course, the default for 2D primitives, while `GL_LINE` draws the primitive in *outline* (or *wireframe* as it's also called), and `GL_POINT` only the vertices.

In fact, it's often easier to decipher a 2D primitive by viewing it in outline, as we'll see in the following experiment introducing triangle strips.

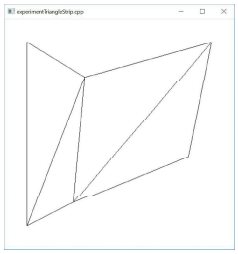

Figure 2.30: Screenshot of `square.cpp` using `GL_TRIANGLE_STRIP` with new vertices.

Experiment 2.14. Continue the preceding experiment by inserting the call `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)` in the drawing routine and, further, replacing `GL_TRIANGLES` with `GL_TRIANGLE_STRIP`. The relevant part of the display routine then is as below:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLE_STRIP);
   glVertex3f(10.0, 90.0, 0.0);
   glVertex3f(10.0, 10.0, 0.0);
   glVertex3f(35.0, 75.0, 0.0);
   glVertex3f(30.0, 20.0, 0.0);
   glVertex3f(90.0, 90.0, 0.0);
   glVertex3f(80.0, 40.0, 0.0);
glEnd();
```

See Figure 2.30. **End**

GL_TRIANGLE_STRIP draws a sequence of triangles – called a *triangle strip* – as follows: the first triangle is $v_0v_1v_2$, the second $v_1v_3v_2$ (v_0 is dropped and v_3 brought in), the third $v_2v_3v_4$ (v_1 dropped and v_4 brought in), and so on. Formally, the triangles in the strip are

$$v_0v_1v_2,\ v_1v_3v_2,\ v_2v_3v_4,\ \ldots,\ v_{n-3}v_{n-2}v_{n-1} \qquad \text{(if } n \text{ is odd)}$$

or

$$v_0v_1v_2,\ v_1v_3v_2,\ v_2v_3v_4,\ \ldots,\ v_{n-3}v_{n-1}v_{n-2} \qquad \text{(if } n \text{ is even)}$$

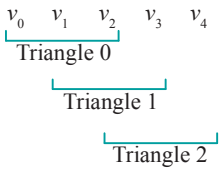

Figure 2.31: Sliding window picking the vertices of triangles in a strip.

As we know, the order of its vertices is important in determining a triangle's orientation, e.g., the second triangle $v_1v_3v_2$ in the strip is not the same as $v_1v_2v_3$. However, if we disregard order for a moment, the vertices of the strip's triangles are picked through a "sliding window" as in Figure 2.31.

Exercise 2.13. (Programming) Create a square annulus as in Figure 2.32(a) using a *single* triangle strip. You may first want to sketch the annulus on graph paper to determine the coordinates of its eight corners. The figure depicts one possible *triangulation* – division into triangles – of the annulus.
Hint: A solution is available in `squareAnnulus1.cpp` of Chapter 3.

Exercise 2.14. (Programming) Create the shape of Figure 2.32(b) using a single triangle strip. A partial triangulation is indicated.

We come to the third and final of the triangle-drawing primitives.

Experiment 2.15. Replace the polygon declaration part of `square.cpp` with:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLE_FAN);
   glVertex3f(10.0, 10.0, 0.0);
   glVertex3f(15.0, 90.0, 0.0);
   glVertex3f(55.0, 75.0, 0.0);
   glVertex3f(80.0, 30.0, 0.0);
   glVertex3f(90.0, 10.0, 0.0);
glEnd();
```

See Figure 2.33. **End**

GL_TRIANGLE_FAN draws a sequence of triangles – called a *triangle fan* – around the first vertex as follows: the first triangle is $v_0v_1v_2$, the second $v_0v_2v_3$, and so on. The full sequence is

$$v_0v_1v_2,\ v_0v_2v_3,\ \ldots,\ v_0v_{n-2}v_{n-1}$$

Exercise 2.15. (Programming) Create a square annulus using *two* triangle fans. First sketch a triangulation different from that in Figure 2.32(a).

We've already met the polygon.

GL_POLYGON draws a polygon with the vertex sequence

$$v_0\, v_1\, \ldots\, v_{n-1}$$

(n must be at least 3 for anything to be drawn).

Finally, more a macro than a true OpenGL primitive:

glRectf($x1$, $y1$, $x2$, $y2$) draws a rectangle lying on the $z = 0$ plane with sides parallel to the x- and y-axes. In particular, the rectangle has diagonally opposite corners at ($x1$, $y1$, 0) and ($x2$, $y2$, 0). The full list of four vertices is ($x1$, $y1$, 0), ($x2$, $y1$, 0), ($x2$, $y2$, 0) and ($x1$, $y2$, 0). The rectangle created is 2-dimensional and its vertex order depends on the situation of the two vertices ($x1$, $y1$, 0) and ($x2$, $y2$, 0) with respect to each other, as indicated by the two drawings at the lower right of Figure 2.28.

Note that `glRectf()` is a stand-alone call; it is not a parameter to `glBegin()` like the other primitives.

Experiment 2.16. Replace the polygon declaration part of `square.cpp` with

```
glRectf(20.0, 20.0, 80.0, 80.0);
```

to see the exact same square (Figure 2.34) of the original `square.cpp`. **End**

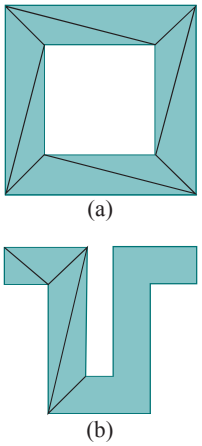

Figure 2.32: (a) Square annulus – the region between two bounding squares – and a possible triangulation (b) A partially triangulated shape.

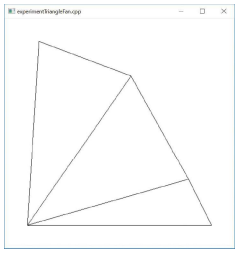

Figure 2.33: Screenshot of `square.cpp` using `GL_TRIANGLE_FAN` with new vertices.

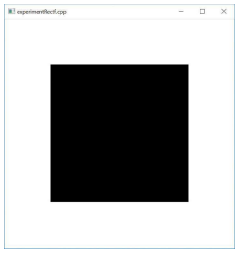

Figure 2.34: Screenshot of `square.cpp` using `glRectf()`.

Important: The preceding two primitives, `GL_POLYGON` and `glRectf()`, have both been discarded from the core profile of later versions of OpenGL, e.g., 4.x, which we are going to study ourselves later in the book; however, they are accessible via the compatibility profile.

Why polygons and rectangles have been discarded is not hard to understand: both can be made from triangles, so are really redundant. The reason we do still use them in the early part of this book is because they afford an easily understood way to make objects – e.g., the square polygon of our very first program `square.cpp` is certainly more intuitive for a beginner than a triangle strip.

Exercise 2.16. (Programming) Replace the polygon of `square.cpp` first with a triangle strip and then with a triangle fan.

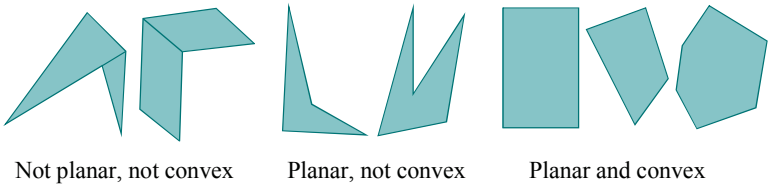

Figure 2.35: Polygons of various types.

It is important that if one does create a polygon, then one must be careful in ensuring that it is a *planar convex* figure, i.e., it lies on one plane and has no "bays" or "inlets" (see Figure 2.35); otherwise, rendering is unpredictable as we'll soon see. Therefore, even though we draw them occasionally for convenience, we recommend that the reader, in order to avoid rendering issues and to prepare for the fourth generation of OpenGL, altogether shun `glRectf`s and `GL_POLYGON`s in her own projects, and, instead, draw 2D objects using exclusively `GL_TRIANGLES`s, `GL_TRIANGLE_STRIP`s and `GL_TRIANGLE_FAN`s.

In fact, following are a couple of experiments, the second one showing how polygon rendering can behave oddly indeed if one is not careful.

Experiment 2.17. Replace the polygon declaration of `square.cpp` with:

```
glBegin(GL_POLYGON);
   glVertex3f(20.0, 20.0, 0.0);
   glVertex3f(50.0, 20.0, 0.0);
   glVertex3f(80.0, 50.0, 0.0);
   glVertex3f(80.0, 80.0, 0.0);
   glVertex3f(20.0, 80.0, 0.0);
glEnd();
```

You see a convex 5-sided polygon (like the one in Figure 2.36(a)). **End**

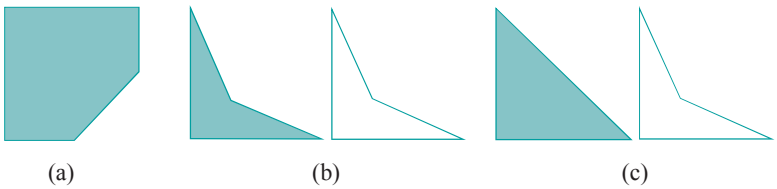

Figure 2.36: Outputs: (a) Experiment 2.17 (b) Experiment 2.18 (c) Experiment 2.18, vertices cycled.

Experiment 2.18. Replace the polygon declaration of `square.cpp` with:

```
glBegin(GL_POLYGON);
   glVertex3f(20.0, 20.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
   glVertex3f(40.0, 40.0, 0.0);
   glVertex3f(20.0, 80.0, 0.0);
glEnd();
```

Display it *both* filled and outlined using appropriate `glPolygonMode()` calls. A non-convex quadrilateral is drawn in either case (Figure 2.36(b)). Next, keeping the same *cycle* of vertices as above, list them starting with `glVertex3f(80.0, 20.0, 0.0)` instead:

```
glBegin(GL_POLYGON);
   glVertex3f(80.0, 20.0, 0.0);
   glVertex3f(40.0, 40.0, 0.0);
   glVertex3f(20.0, 80.0, 0.0);
   glVertex3f(20.0, 20.0, 0.0);
glEnd();
```

Make sure to display it both filled and outlined. When filled it's a triangle, while outlined it's a non-convex quadrilateral (Figure 2.36(c)) identical to the one output earlier! Since the cycle of the vertices around the quad is unchanged, only starting at a different point, shouldn't the output still be as in Figure 2.36(b), both filled and outlined? **End**

We'll leave the apparent anomaly* of this experiment as a mystery to be resolved in Chapter 8 on triangulation. But, if you are impatient then the hint provided with Exercise 2.9 should give the answer.

Exercise 2.17. (Programming) Verify, by cycling the vertices, that no such anomaly arises in the case of the convex polygon of Experiment 2.17.

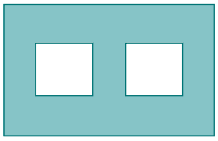

Figure 2.37: Double annulus.

Exercise 2.18. (Programming) Draw the double annulus (a figure '8') shown in Figure 2.37 using as few triangle strips as possible. Introduce extra vertices on the three boundary components, in addition to the original twelve, if you need to for a triangulation.

Note: Such additional vertices are called *Steiner vertices*. For example, Figure 2.38 shows six additional Steiner vertices allowing for a more even triangulation – desirable in certain applications – of a long thin rectangle, than just the original four.

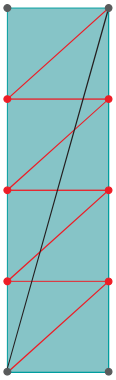

Figure 2.38: Black edge gives a triangulation, consisting of 2 long thin triangles, using only the rectangle's own 4 vertices; 6 red Steiner vertices and the red edges give a triangulation consisting of 8 nearly equal-sided triangles.

Remark 2.11. Here's an interesting semi-philosophical question. OpenGL claims to be a 3D drawing API. Yet, why does it not have a single 3D drawing primitive, e.g., cube, tetrahedron or such? All its primitives are 0-dimensional (`GL_POINTS`), 1-dimensional (`GL_LINE*`) or 2-dimensional (`GL_TRIANGLE*`).

The answer lies in how we humans (the regular ones that is and not supers with X-ray vision) perceive 3D objects such as cubes, tetrahedrons, chairs and spacecraft: *we see only the surface, which is two-dimensional*. It makes sense for a 3D API, therefore, to draw only as much as can be seen. Of course, OpenGL's "3Dness" lies in allowing us to make these drawings in 3D xyz-space.

2.7 Approximating Curved Objects

Looking back at Figure 2.28 we see that the OpenGL geometric primitives are composed of points, straight line segments and flat pieces, the latter being triangles, rectangles

*The rendering depends on the particular OpenGL implementation. However, all implementations that we are aware of show identical behavior.

and polygons. How, then, to draw curved objects such as discs, ellipses, spirals, beer cans and flying saucers? The answer is to *approximate* them with straight and flat OpenGL primitives well enough that the viewer cannot tell the difference. As a wag once put it, "Sincerity is a very important human quality. If you don't have it, you *gotta* fake it!" In the next experiment we fake a circle.

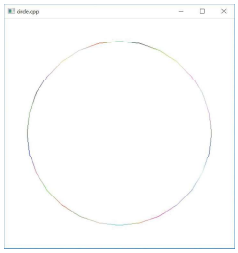

Figure 2.39: Screenshot of `circle.cpp`.

Experiment 2.19. Run `circle.cpp`. Increase the number of vertices in the line loop

```
glBegin(GL_LINE_LOOP);
   for(i = 0; i < numVertices; ++i)
   {
      glColor3f((float)rand()/(float)RAND_MAX,
                (float)rand()/(float)RAND_MAX,
                (float)rand()/(float)RAND_MAX);
      glVertex3f(X + R * cos(t), Y + R * sin(t), 0.0);
      t += 2 * M_PI / numVertices;
   }
glEnd();
```

by pressing '+' till it "becomes" a circle, as in the screenshot of Figure 2.39. Press '-' to decrease the number of vertices. The randomized colors are a bit of eye candy.

Note: `M_PI` is the constant representing π in the `<cmath>` library. **End**

The vertices of the loop of `circle.cpp`, which lie evenly spaced on the circle, are collectively called a *sample of points* or, simply, *sample* from the circle. The loop itself evidently bounds a regular polygon. See Figure 2.40(a). Clearly, the denser the sample the better the loop approximates the circle.

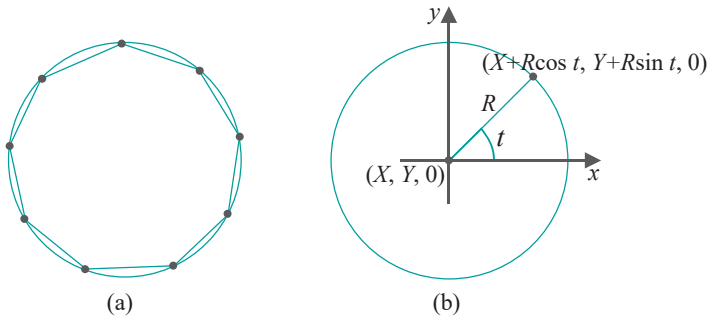

Figure 2.40: (a) A line loop joining a sample of points from a circle (b) Parametric equations for a circle.

The parametric equations of the circle implemented are

$$x = X + R\cos t,\ y = Y + R\sin t,\ z = 0, \quad 0 \le t \le 2\pi \tag{2.1}$$

where $(X, Y, 0)$ is the center and R the radius. See Figure 2.40(b). A `numVertices` number of sample points, with coordinates $(X + R\cos t, Y + R\sin t, 0)$, equally spaced apart is generated by starting with the angle $t = 0$ and then incrementing it successively by $2\pi/$`numVertices`.

Observe that the vertex specifications occur within a loop construct, which is pretty much mandatory if there is a large number of vertices.

Incidentally, the program `circle.cpp` also demonstrates output to the command window, as well as non-trivial user interaction via the keyboard. The routine `keyInput()` is registered as the key handling routine in `main()` by the `glutKeyboardFunc(keyInput)` statement. Note the calls to `glutPostRedisplay()` in `keyInput()` asking the display to be redrawn after each update of `numVertices`.

Follow these conventions when writing OpenGL code:

1. Program the "Esc" key to exit the program.
2. Describe user interaction at two places:
   (a) The command window using `cout()`.
   (b) Comments at the *top* of the source code.

Here's a parabola.

Experiment 2.20. Run `parabola.cpp`. Press '+/-' to increase/decrease the number of vertices of the approximating line strip. Figure 2.41 is a screenshot with enough vertices to make a smooth-looking parabola.

The vertices are equally spaced along the x-direction. The parametric equations implemented are

$$x = 50 + 50t,\ y = 100t^2,\ z = 0, \quad -1 \le t \le 1$$

the constants being chosen so that the parabola is centered in the window. **End**

Exercise 2.19. (Programming) Modify `circle.cpp` to draw a flat 3-turn spiral like the one in Figure 2.42.

Exercise 2.20. (Programming) Modify `circle.cpp` to draw a disc (i.e., a filled circle) by way of (a) a polygon and (b) a triangle fan.

Exercise 2.21. (Programming) Draw a flat leaf like the one in Figure 2.43.

Exercise 2.22. (Programming) Modify `circle.cpp` to draw a circular annulus, like one of those shown in Figure 2.44, using a triangle strip. Don't look at the program `circularAnnuluses.cpp`!

We'll be returning shortly to the topic of approximating curved objects, but it's on to 3D next.

2.8 Three Dimensions, the Depth Buffer and Perspective Projection

The reader by now may be impatient to move on from the plane (pun intended) and simple to full 3D. Okay then, let's get off to an easy start in 3-space by making use of the third dimension to fake a circular annulus. Don't worry, we'll be doing fancier stuff soon enough!

Experiment 2.21. Run `circularAnnuluses.cpp`. Three identical-looking red circular annuluses (Figure 2.44) are drawn in three *different* ways:

i) Upper-left: There is not a real hole. The white disc *overwrites* the red disc as it appears later in the code:

```
glColor3f(1.0, 0.0, 0.0);
drawDisc(20.0, 25.0, 75.0, 0.0);
glColor3f(1.0, 1.0, 1.0);
drawDisc(10.0, 25.0, 75.0, 0.0);
```

Note: The first parameter of the subroutine `drawDisc()` is the radius and the remaining three the coordinates of the center.

ii) Upper-right: There is not a real hole either. A white disc is drawn *closer* to the viewer than the red disc thus blocking it out:

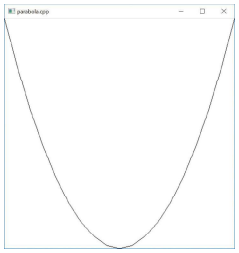

Figure 2.41: Screenshot of `parabola.cpp`.

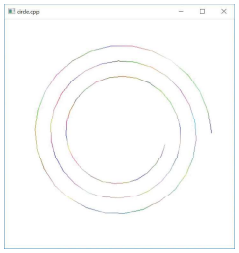

Figure 2.42: Flat spiral.

Figure 2.43: Flat leaf.

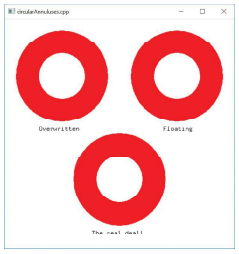

Figure 2.44: Screenshot of `circularAnnuluses.cpp`.

```
glEnable(GL_DEPTH_TEST);
glColor3f(1.0, 0.0, 0.0);
drawDisc(20.0, 75.0, 75.0, 0.0);
glColor3f(1.0, 1.0, 1.0);
drawDisc(10.0, 75.0, 75.0, 0.5);
glDisable(GL_DEPTH_TEST);
```

Observe that the z-value of the white disc's center is greater than the red disc's, bringing it closer to the viewing face. We'll discuss momentarily the mechanics of one primitive blocking out another.

iii) Lower: A true circular annulus with a real hole:

```
if (isWire) glPolygonMode(GL_FRONT, GL_LINE);
else glPolygonMode(GL_FRONT, GL_FILL);
glColor3f(1.0, 0.0, 0.0);
glBegin(GL_TRIANGLE_STRIP);
...
glEnd();
```

Press the space bar to see the wireframe of a triangle strip. End

Exercise 2.23. (Programming) Interchange in `circularAnnuluses.cpp` the drawing orders of the red and white discs – i.e., the order in which they appear in the code – in either of the top two annuluses. Which one is affected? (*Only the first!*) Why?

Remark 2.12. Note the use of a text-drawing routine in `circularAnnuluses.cpp`. OpenGL offers only rudimentary text-drawing capability but it often comes in handy, especially for annotation. We'll discuss text-drawing in fair detail in Chapter 3.

By far the most important aspect of `circularAnnuluses.cpp` is its use of the *depth buffer*, which allows objects nearer the eye to block out ones behind them, to draw the upper-right annulus. Following is an introduction to this critical 3D utility.

2.8.1 A Vital 3D Utility: The Depth Buffer

Enabling the depth buffer, also called the *z-buffer*, causes OpenGL to eliminate, prior to rendering, parts of objects that are *obscured* (or, *occluded*) by others.

Precisely, a point of an object is not drawn if its projection – think of a ray from that point – toward the viewing face is obstructed by another object. See Figure 2.45(a) for the making of the upper-right annulus of `circularAnnuluses.cpp`: the white disc obscures the part of the red disc behind it (because the projection is orthogonal, the obscured part is exactly the same shape and size as the white disc). This process is called *hidden surface removal* or *depth testing* or *visibility determination*.

Stated mathematically, the result of hidden surface removal in case of orthographic projection is as follows.

Fix an x-value X and a y-value Y within the span of the viewing box. Consider the set S of points belonging to objects in the viewing box with their x-value equal to X and y-value equal to Y. Precisely, S is the set of points where the straight line L through $(X, Y, 0)$ parallel to the z-axis intersects objects in the viewing box. Clearly, S is of the form $S = \{(X, Y, z)\}$, where z varies depending on the intersected objects (it's empty if L intersects nothing in the viewing box). Let Z be the largest of these z-values, assuming S to be not empty. In other words, of points belonging to objects in the viewing box with x-value equal to X and y-value to Y, (X, Y, Z) has the largest z-value and lies closest to the viewing face.

Next, observe that all points in S project to the same point $P = (X, Y, -near)$, on the viewing face. Here, then, is the consequence of hidden surface removal: P is rendered with the color attributes of (X, Y, Z). The implication is that only (X, Y, Z),

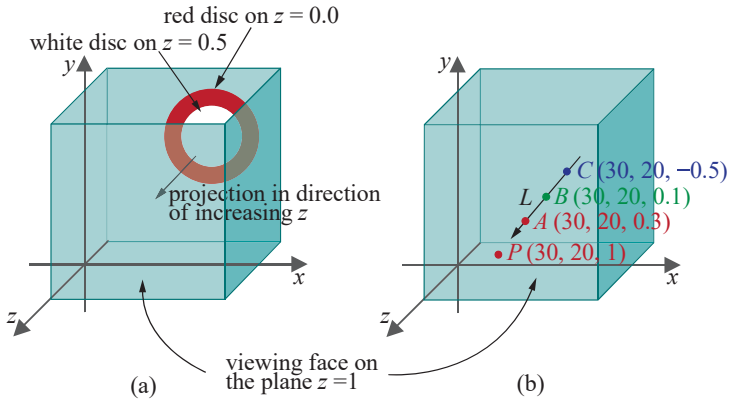

Figure 2.45: (a) The front white disc obscures part of the red one (b) The point A with largest z-value is projected onto the viewing plane so P is red.

closest to the viewing face, is drawn of the points in S, the rest, which are behind it, being obscured.

For example, in Figure 2.45(b), the three points A, B and C, colored red, green and blue, respectively, share the same first two coordinate values, namely, $X = 30$ and $Y = 20$. So, all three project along the line L to the same point P on the viewing face. As A has the largest z coordinate of the three, it obscures the other two and P, therefore, is drawn red.

The z-buffer itself is a block of memory in the GPU containing z-values, one per pixel. If depth testing is enabled, then, as a primitive is processed for rendering, the z-value of each of its points – or, more accurately, each of its pixels – is compared with that of the one with the same (x, y)-values currently resident in the z-buffer. If an incoming pixel's z-value is greater, then its RGB attributes and z-value replace those of the current one; if not, the incoming pixel's data is discarded.

For example, if the order in which the points of Figure 2.45(b) happen to appear in the code is C, A and B, here's how the color and z-buffer values at the pixel corresponding to P change:

```
draw C; // Pixel corresponding to P gets color blue
        // and z-value -0.5.
draw A; // Pixel corresponding to P gets color red
        // and z-value 0.3: A's values overwrite C's.
draw B; // Pixel corresponding to P retains color red
        // and z-value 0.3: B is discarded.
```

Remark 2.13. In actual implementation in the GPU, the value per pixel in the z-buffer is between 0 and 1, with 0 corresponding to the near face of the viewing box and 1 the far face. What happens is that, before recording them in the z-buffer, the system transforms (by a scaling, though not necessarily linear) world space z-values each to the range $[0, 1]$ with a flip in sign so that pixels farther from the viewing face have higher z-value. Consequently, following this transformation, lower values actually win the competition to be visible in the z-buffer. However, when writing OpenGL code we are operating in world space, where higher z-values in the viewing box are closer to the viewing face, and need not concern ourselves with this implementation particularity.

We ask you to note in `circularAnnuluses.cpp` the enabling syntax of hidden surface removal so that you can implement it in your own programs:

1. The `GL_DEPTH_BUFFER_BIT` parameter of `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` in the `drawScene()` routine causes the depth buffer to be cleared.

Figure 2.46: Bull's eye target.

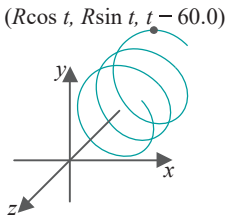

Figure 2.47: Parametric equations for a helix.

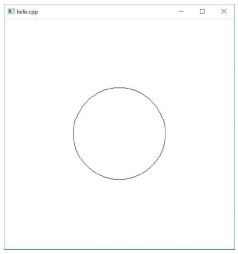

Figure 2.48: Screenshot of `helix.cpp` using orthographic projection with the helix coiling around the z-axis.

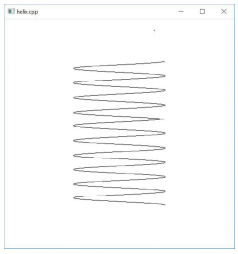

Figure 2.49: Screenshot of `helix.cpp` using orthographic projection with the helix coiling around the y-axis.

2. The command `glEnable(GL_DEPTH_TEST)` in the `drawScene()` routine turns hidden surface removal on. The complementary command is `glDisable(GL_DEPTH_TEST)`.

3. The `GLUT_DEPTH` parameter of `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)` in `main()` causes the depth buffer to be initialized.

Exercise 2.24. (Programming) Draw a bull's eye target as in Figure 2.46 by means of five discs of different colors, sizes and depths.

Exercise 2.25. (Programming) Here's a fun exercise related more to clipping than depth buffering though. The top two annuluses of `circularAnnuluses.cpp` are "fake" in that there is not a real hole in the disc. Can you make a fake annulus in yet another way by drawing first a filled disc as a triangle fan, e.g., as in Exercise 2.20(b), and then repositioning only the center of the fan outside the viewing box (so that the object is really a cone)?

Remark 2.14. The z-buffer, of course, is incredibly important in OpenGL's 3D scheme of things. However, it really will not be called upon to do much in the simple scenes with very few objects that we are going to be creating for a while. It's in busy scenes that the depth buffer comes into its own.

Remark 2.15. Before z-buffers started taking over the world with the advent of the first GPUs in the early 80s – by virtue of their simplicity and sheer speed derived from being implemented in GPU hardware – a more complex software technique based on so-called BSP (binary space partitioning) trees was used extensively for hidden surface removal. We don't cover this technique ourselves in this book but an earlier edition had a detailed discussion which the interested reader will find extracted at the Downloads page of our website.

2.8.2 A Helix and Perspective Projection

We get more seriously 3D next by drawing a spiral or, more scientifically, a helix. A helix, though itself a 1-dimensional object – drawn as a line strip actually – can be made authentically only in 3-space.

Open `helix.cpp` but don't run it as yet! The parametric equations implemented are

$$x = R\cos t,\ y = R\sin t,\ z = t - 60.0, \quad -10\pi \le t \le 10\pi \tag{2.2}$$

See Figure 2.47. Compare these with Equation (2.1) for a circle centered at (0, 0, 0), putting $X = 0$ and $Y = 0$ in that earlier equation. The difference is that the helix climbs up the z-axis *simultaneously* as it rotates circularly with increasing t (so, effectively, it coils around the z-axis). Typically, one writes simply $z = t$ for the last coordinate; however, we tack on "-60.0" to push the helix far enough down the z-axis so that it's contained entirely in the viewing box.

Exercise 2.26. Even before viewing the helix, can you say from Equation (2.2) how many times it is supposed to coil around the z-axis, i.e., how many full turns it is supposed to make?
Hint: One full turn corresponds to an interval of 2π along t.

Experiment 2.22. Okay, run `helix.cpp` now. All we see is a circle as in Figure 2.48! There's no sign of any coiling up or down. The reason, of course, is that the orthographic projection onto the viewing face flattens the helix. Let's see if it makes a difference to turn the helix upright, in particular, so that it coils around the y-axis. Accordingly, replace the statement

```
glVertex3f(R * cos(t), R * sin(t), t - 60.0);
```

in the drawing routine with

```
glVertex3f(R * cos(t), t, R * sin(t) - 60.0);
```

Hmm, not a lot better (Figure 2.49). End

Because it squashes a dimension, typically, orthographic projection is not suitable for 3D scenes. OpenGL, in fact, provides another kind of projection, called *perspective projection*, more appropriate for most 3D applications.

Perspective projection is implemented with a `glFrustum()` call replacing `glOrtho()`. Instead of a viewing box, a `glFrustum(`*left*`,` *right*`,` *bottom*`,` *top*`,` *near*`,` *far*`)` call sets up a *viewing frustum* – a frustum is a *truncated pyramid* whose top has been cut off by a plane parallel to its base – in the following manner (see Figure 2.50):

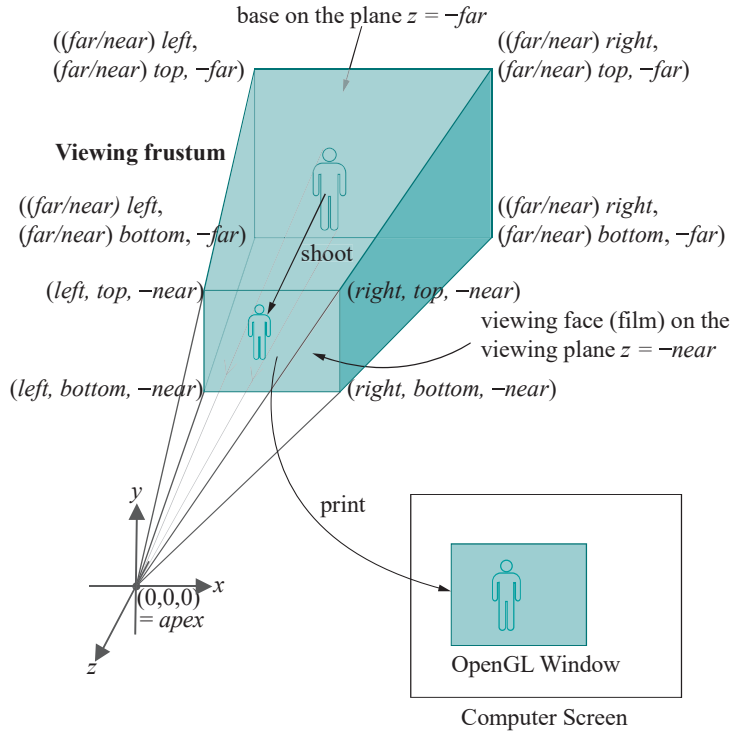

Figure 2.50: Rendering with `glFrustum()`.

The apex of the pyramid is at the origin. The front face, or *viewing face*, of the frustum is the rectangle, lying on the plane $z = -near$, whose corners are ($left$, $bottom$, $-near$), ($right$, $bottom$, $-near$), ($left$, top, $-near$), and ($right$, top, $-near$). The plane $z = -near$ is the *viewing plane* – in fact, it's the plane which truncates the pyramid.

The four edges of the pyramid emanating from the apex pass through the four corners of the viewing face. The base or back face of the frustum is the rectangle whose vertices are precisely where the pyramid's four edges intersect the $z = -far$ plane. By proportionality with the front vertices, the coordinates of the base vertices are:

$((far/near)\ left, (far/near)\ bottom, -far)$,
$((far/near)\ right, (far/near)\ bottom, -far)$,
$((far/near)\ left, (far/near)\ top, -far)$,
$((far/near)\ right, (far/near)\ top, -far)$

Values of the `glFrustum()` parameters are typically set so that the frustum lies symmetrically about the z-axis; in particular, *right* and *top* are chosen to be positive,

and *left* and *bottom* their respective negatives. The parameters *near* and *far* should both be positive and $near < far$, which means the frustum lies entirely on the negative side of the z-axis with its base behind the viewing face.

Remark 2.16. An intuitive way to relate the box of `glOrtho(`*left*`,` *right*`,` *bottom*`,` *top*`,` *near*`,` *far*`)` with the frustum of `glFrustum(`*left*`,` *right*`,` *bottom*`,` *top*`,` *near*`,` *far*`)` is to note first that their front faces are identical and that the back face of the box is simply a copy of the front face shifted back, while the back face of the frustum is stretched as it's shifted back, the corners following rays from the origin.

Example 2.1. Determine the corners of the viewing frustum created by the call `glFrustum(-15.0, 15.0, -10.0, 10.0, 5.0, 50.0)`.

Answer: By definition, the corners of the front face are $(-15.0, -10.0, -5.0)$, $(15.0, -10.0, -5.0)$, $(-15.0, 10.0, -5.0)$ and $(15.0, 10.0, -5.0)$. The x- and y-values of the vertices of the base (or back face) are scaled from those on the front by a factor of 10 (because $far/near = 50/5 = 10$). The base vertices are, therefore, $(-150.0, -100.0, -50.0)$, $(150.0, -100.0, -50.0)$, $(-150.0, 100.0, -50.0)$ and $(150.0, 100.0, -50.0)$.

Exercise 2.27. Determine the corners of the viewing frustum created by the call `glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0)`.

The rendering sequence in the case of perspective projection is a two-step shoot-and-print, similar to orthographic projection. The shooting step again consists of projecting objects within the viewing frustum onto the viewing face, *except that the projection is no longer perpendicular*. Instead, each point is projected along the line joining it to the apex, as depicted by the light black rays from the bottom and top of the man in Figure 2.50.

Perspective projection causes *foreshortening* because objects farther away from the apex appear smaller (a phenomenon also called *perspective transformation*). For example, see Figure 2.51 where A and B are of the same height, but the projection pA is shorter than the projection pB.

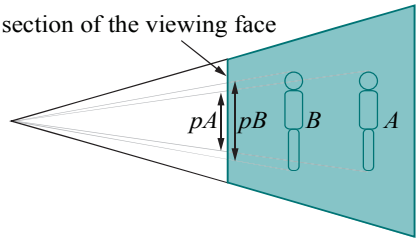

Figure 2.51: Section of the viewing frustum showing foreshortening.

The second rendering step of printing where the viewing face is proportionately scaled to fit onto the OpenGL window is exactly as for orthographic projection. Exactly as for orthographic projection as well, the scene is clipped to within the viewing frustum by the 6 planes that bound the latter.

Time now to see perspective projection work its magic!

Experiment 2.23. Fire up the original `helix.cpp` program. Replace orthographic projection with perspective projection; in particular, replace the projection statement

```
glOrtho(-50.0, 50.0, -50.0, 50.0, 0.0, 100.0);
```

with

```
glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0);
```

You can see a real spiral now (Figure 2.52). View the upright version as well (Figure 2.53), replacing

```
glVertex3f(R * cos(t), R * sin(t), t - 60.0);
```

with

```
glVertex3f(R * cos(t), t, R * sin(t) - 60.0);
```

A lot better than the orthographic version is it not?! End

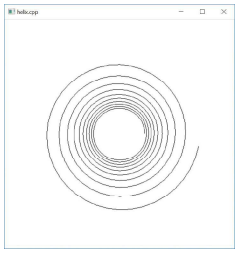

Figure 2.52: Screenshot of `helix.cpp` using perspective projection with the helix coiling up the z-axis.

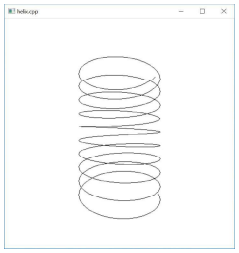

Figure 2.53: Screenshot of `helix.cpp` using perspective projection with the helix coiling up the y-axis.

Perspective projection is more realistic than orthographic projection as it mimics how images are formed on the retina of the eye by light rays traveling toward a point. And, in fact, it's precisely foreshortening which cues us humans to the distance of an object.

Remark 2.17. One can think of the apex of the frustum as the location of a *point camera* and the viewing face as its film.

Remark 2.18. One might think of orthographic and perspective projections *both* as being along lines of projection convergent to a single point, the *center of projection* (COP). In the case of perspective projection, this is a regular point with finite coordinates; however, for orthographic projection the COP is a "point at infinity" – i.e., infinitely far away – so that lines toward it are parallel.

Moreover, the sides of a viewing volume from back to front logically follow the lines of projection, so leading us from a box in the case of orthographic projection to a frustum in the case of perspective projection.

Remark 2.19. There do exist 3D applications, e.g., in architectural design, where foreshortening amounts to distortion, so, in fact, orthographic projection is preferred.

Remark 2.20. It's because it captures the image of an object by intersecting rays projected from the object – either orthographically or perspectively – with a plane, which is similar to how a real camera works, that OpenGL is said to implement the *synthetic-camera* model.

Exercise 2.28. (Programming) Continuing from where we were at the end of the preceding experiment using `helix.cpp`, successively replace the `glFrustum()` call as follows, trying in each case to predict the change in the display caused by the change in the frustum before running the code:

(a) Move the back face back: `glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 120.0)`

(b) Move the front face back: `glFrustum(-5.0, 5.0, -5.0, 5.0, 10.0, 100.0)`

(c) Move the front face forward: `glFrustum(-5.0, 5.0, -5.0, 5.0, 2.5, 100.0)`

(d) Make the front face bigger: `glFrustum(-10.0, 10.0, -10.0, 10.0, 5.0, 100.0)`

Parts (b) and (c) show, particularly, how moving the "film" causes the camera to zoom.

Exercise 2.29. Formulate mathematically how hidden surface removal should work in the case of perspective projection, as we did in Section 2.8.1 for orthographic projection.

Figure 2.54: Screenshot of `moveSphere.cpp`.

Experiment 2.24. Run `moveSphere.cpp`, which simply draws a movable sphere in the OpenGL window. Press the left, right, up and down arrow keys to move the sphere, the space bar to rotate it and 'r' to reset.

The sphere appears distorted as it nears the boundary of the window, as you can see from the screenshot in Figure 2.54. Can you guess why? Ignore the code,

especially unfamiliar commands such as `glTranslatef()` and `glRotatef()`, except for that projection is perspective.

This kind of *peripheral distortion* of a 3D object is unavoidable in any viewing system which applies perspective projection. It happens with a real camera as well, but we don't notice it as much because the field of view when snapping pictures is usually quite large with objects of interest tending to be centered, and the curved lens is designed to compensate as well. End

2.9 Drawing Projects

Here are a few exercises to stretch your drawing muscles. The objects may look rather different from what we have drawn so far, but as programming projects aren't really. In fact, you can probably cannibalize a fair amount of code from earlier programs.

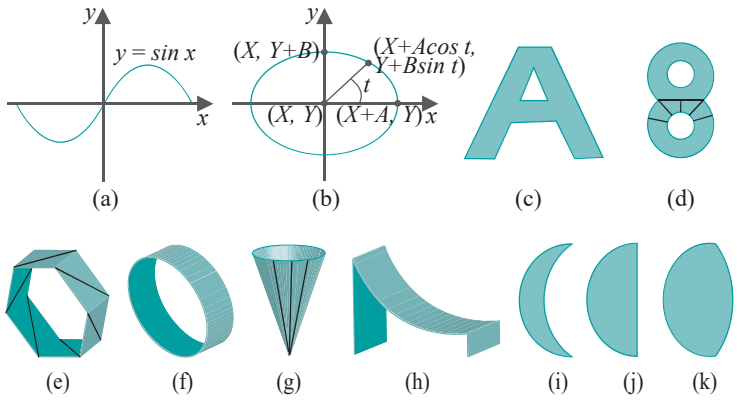

Figure 2.55: Draw these!

Exercise 2.30. (Programming) Draw a sine curve between $x = -\pi$ and $x = \pi$ (Figure 2.55(a)). Follow the strategy of `circle.cpp` to draw a polyline through a sample from the sine curve.

Exercise 2.31. (Programming) Draw an ellipse. Recall the parametric equations for an ellipse on the xy-plane, centered at (X, Y), with semi-major axis of length A and semi-minor axis of length B (Figure 2.55(b)):

$$x = X + A\cos t,\ y = Y + B\sin t,\ z = 0, \qquad 0 \le t \le 2\pi$$

Exercise 2.32. (Programming) Draw the letter 'A' as a *two-dimensional* figure like the shaded region in Figure 2.55(c). It might be helpful to triangulate it first on graph paper. Try to pack as many triangles into as few triangle strips as possible (because each drawing call costs in the GPU, but see the next experiment). Allow the user to toggle between filled and wireframe *à la* the bottom annulus of `circularAnnuluses.cpp`.

Note: Variations of this exercise may be made by asking different letters. Keep in mind that curvy letters like 'S' are harder than straight-sided ones.

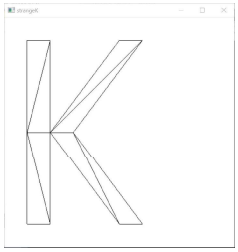

Figure 2.56: Screenshot of `strangeK.cpp`.

Experiment 2.25. Run `strangeK.cpp`, which shows how one might take the edict of minimizing the number of triangle strips a bit far. Press space to see the wireframe of a perfectly good 'K' (Figure 2.56).

Interestingly, though, the letter is drawn as a single triangle strip with 17 vertices within the one `glBegin(GL_TRIANGLE_STRIP)-glEnd()`. We ask the reader to parse

this strip as follows: sketch the K on on a piece of paper, label the vertices $v_0, v_1, \ldots, v_{16}$ according to their code order (e.g., the lower left vertex of the straight side is v_0, to its right is v_1, and so on, and, yes, some vertices are labeled multiple times), and, finally, examine each one of the strip's 15 component triangles (best done by following the sliding window scheme as in Figure 2.31).

Did you spot a few so-called *degenerate* triangles, e.g., one with all its vertices along a straight line or with coincident vertices? Such triangles, which are not really 2D, are best avoided when drawing a 2D figure. End

Exercise 2.33. (Programming) Draw the number '8' as the 2D object in Figure 2.55(d). Do this in two different ways: (i) drawing 4 discs and using the z-buffer, and (ii) as a true triangulation, allowing the user to toggle between filled and wireframe. For (ii), a method of dividing the '8' into two triangle strips is suggested in Figure 2.55(d).

Note: Variations of part (ii) may be made by asking different numbers.

Exercise 2.34. (Programming) Draw a ring with cross-section a regular (equal-sided) polygon as in Figure 2.55(e), where a scheme to triangulate the ring in one triangle strip is indicated. Allow the user to change the number of sides of the cross-section. Increasing the number of sides sufficiently should make the ring appear cylindrical as in Figure 2.55(f). Use perspective projection and draw in wireframe.

Exercise 2.35. (Programming) Draw a cone as in Figure 2.55(g) where a possible triangulation is indicated. Draw in wireframe and use perspective projection.

Exercise 2.36. (Programming) Draw a children's slide as in Figure 2.55(h). Choose an appropriate equation for the cross-section of the curved surface – part of a parabola, maybe – and then "extrude" it sideways as a triangle strip. (If you did Exercise 2.34 then you've already extruded a polygon.) Draw in wireframe and use perspective projection.

Exercise 2.37. (Programming) Draw in a single scene a crescent moon, a half-moon and a three-quarter moon (Figures 2.55(i)-(k)). Each should be a true triangulation. Label each as well using text-drawing.

Remark 2.21. Your output from Exercises 2.34-2.36 may look a bit "funny", especially viewed from certain angles. For example, the ring viewed head-on down its axis may appear as two concentric circles on a single plane. This problem can be alleviated by drawing the object with a different alignment or, equivalently, changing the viewpoint. In Experiment 2.26, coming up shortly, we'll learn code for the user to change her viewpoint in real-time.

Remark 2.22. This thought may have occurred to the reader earlier, when we were discussing the viewing box or frustum: situating the front face of, say, the frustum very close to the eye (in other words, a *near* parameter value of `glFrustum()` of nearly zero) and the back face very far (in other words, a very large *far* parameter value) will create a large viewing space and less likelihood of inadvertent clipping.

However, beware! Increasing the distance between the near and far planes of the viewing space (box or frustum) causes loss in depth resolution because world space depths are transformed all to the range $[0, 1]$ in the z-buffer (see Remark 2.13). Moreover, in the case of the frustum there'll be loss of precision too if the front face is small – as it will be if close to the eye.

In fact, the user should try to push the front face as far back and bring the back face as close in as possible while keeping the scene within the viewing volume.

2.10 Approximating Curved Objects Once More

Our next 3-space drawing project is a bit more challenging: a hemisphere, which is a 2-dimensional object. We'll get in place, as well, certain design principles which will be expanded in Chapter 10 which is dedicated to drawing (no harm starting early).

Remark 2.23. A hemisphere is a 2-dimensional object because it is a surface. Recall that a helix is 1-dimensional because it's line-like. Now, both hemisphere and helix need 3-space to "sit in"; they cannot do with less. For example, you could sketch either on a piece of paper (2-space) but it would not be the real thing. On the other hand, a circle – another 1D object – does sit happily in 2-space.

Consider a hemisphere of radius R, centered at the origin O, with its circular base lying on the xz-plane. Suppose the spherical coordinates of a point P on this hemisphere are a longitude of θ (measured counter-clockwise from the x-axis when looking from the plus side of the y-axis) and a latitude of ϕ (measured from the xz-plane toward the plus side of the y-axis). See Figure 2.57(a). The Cartesian coordinates of P are by elementary trigonometry

$$(R\cos\phi\cos\theta,\ R\sin\phi,\ -R\cos\phi\sin\theta) \tag{2.3}$$

The range of θ is $0 \le \theta \le 2\pi$ and of ϕ is $0 \le \phi \le \pi/2$.

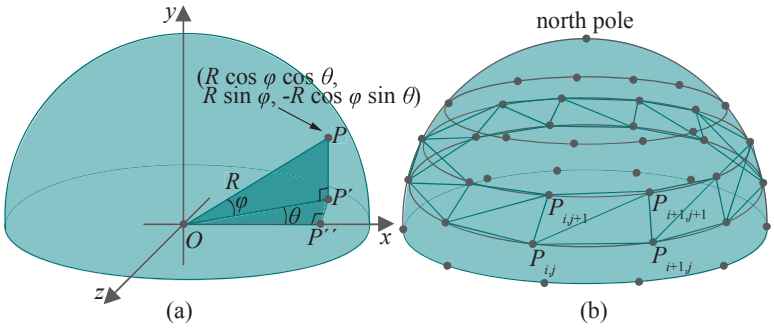

Figure 2.57: (a) Spherical and Cartesian coordinates on a hemisphere (b) Approximating a hemisphere with latitudinal triangle strips.

Exercise 2.38. Verify that the Cartesian coordinates of P are as claimed in (2.3). *Suggested approach*: From the right-angled triangle OPP' one has $|PP'| = R\sin\phi$ and $|OP'| = R\cos\phi$. $|PP'|$ is the y-value of P. Next, from right-angled triangle $OP'P''$, find in terms of $|OP'|$ and θ the values of $|OP''|$ and $|P'P''|$. The first is the x-value of P, while the latter negated the z-value.

Sample the hemisphere at a mesh of $(p+1)(q+1)$ points P_{ij}, $0 \le i \le p$, $0 \le j \le q$, where the longitude of P_{ij} is $(i/p) * 2\pi$ and its latitude $(j/q) * \pi/2$. In other words, $p+1$ longitudinally equally-spaced points are chosen along each of $q+1$ equally-spaced latitudes. See Figure 2.57(b), where $p = 10$ and $q = 4$. The sample points P_{ij} are not all distinct. In fact, $P_{0j} = P_{pj}$, for all j, as the same point has longitude both 0 and 2π; and, the point P_{iq}, for all i, is identical to the north pole, which has latitude $\pi/2$ and arbitrary longitude.

The plan now is to approximate the circular band between each pair of adjacent latitudes with a triangle strip – such a strip will take its vertices alternately from either latitude. Precisely, we'll draw one triangle strip with vertices at

$$P_{0,j+1},\ P_{0j},\ P_{1,j+1},\ P_{1j},\ \ldots,\ P_{p,j+1},\ P_{pj}$$

for each j, $0 \le j \le q-1$, for a total of q triangle strips. These q triangle strips together will approximate the hemisphere itself.

Experiment 2.26. Run `hemisphere.cpp`, which implements exactly the strategy just described. You can verify this from the snippet that draws the hemisphere:

```
for(j = 0; j < q; j++)
{
   // One latitudinal triangle strip.
   glBegin(GL_TRIANGLE_STRIP);
      for(i = 0; i <= p; i++)
      {
         glVertex3f(R * cos((float)(j+1)/q * M_PI/2.0) *
                        cos(2.0 * (float)i/p * M_PI),
                    R * sin((float)(j+1)/q * M_PI/2.0),
                    -R * cos((float)(j+1)/q * M_PI/2.0) *
                        sin(2.0 * (float)i/p * M_PI));
         glVertex3f(R * cos((float)j/q * M_PI/2.0) *
                        cos(2.0 * (float)i/p * M_PI),
                    R * sin((float)j/q * M_PI/2.0),
                    -R * cos((float)j/q * M_PI/2.0) *
                        sin(2.0 *(float)i/p * M_PI));
      }
   glEnd();
}
```

Increase/decrease the number of longitudinal slices by pressing 'P/p'. Increase/decrease the number of latitudinal slices by pressing 'Q/q'. Turn the hemisphere about the axes by pressing 'x', 'X', 'y', 'Y', 'z' and 'Z'. See Figure 2.58 for a screenshot. End

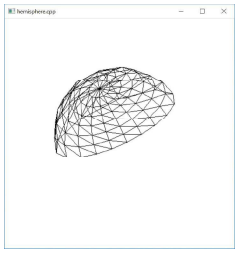

Figure 2.58: Screenshot of `hemisphere.cpp`.

Experiment 2.27. Playing around a bit with the code will help clarify the construction of the hemisphere:

(a) Change the range of the hemisphere's outer loop from

```
for(j = 0; j < q; j++)
```

to

```
for(j = 0; j < 1; j++)
```

Only the bottom strip is drawn. The keys 'P/p' and 'Q/q' still work.

(b) Change it again to

```
for(j = 0; j < 2; j++)
```

Now, the bottom two strips are drawn.

(c) Reduce the range of both loops:

```
for(j = 0; j < 1; j++)
...
      for(i = 0; i <= 1; i++)
      ...
```

The first two triangles of the bottom strip are drawn.

(d) Then, increase the range of the inner loop by 1:

```
for(j = 0; j < 1; j++)
...
      for(i = 0; i <= 2; i++)
      ...
```

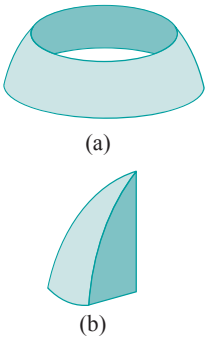

Figure 2.59: (a) Half a hemisphere (b) Slice of a hemisphere.

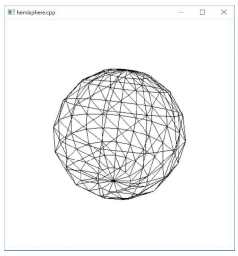

Figure 2.60: A wireframe sphere.

The first four triangles of the bottom strip are drawn. End

There's syntax in `hemisphere.cpp` – none to do with the actual making of the hemisphere – which you may be seeing for the first time. The command `glTranslatef(0.0, 0.0, -10.0)` is used to move the hemisphere, drawn initially centered at the origin, into the viewing frustum, while the `glRotatef()` commands turn it. We'll be studying these so-called *modeling transformations* in Chapter 4 but you are encouraged to experiment with them even now as the syntax is fairly intuitive. The set of three `glRotatef()`s, particularly, comes in handy to re-align a scene.

Exercise 2.39. (Programming) Modify `hemisphere.cpp` to draw:

(a) the bottom half of a hemisphere (Figure 2.59(a)).

(b) a 30° slice of a hemispherical cake (Figure 2.59(b)). Note that simply reducing the range of the inner loop of `hemisphere.cpp` produces a slice of cake without two sides and bottom, so these have to be added in separately to close up the slice.

Make sure the 'P/p/Q/q' keys still work.

Exercise 2.40. A sphere, of course, can be made of two separate back-to-back hemispheres. However, try to manufacture it as one object with a simple modification of `hemisphere.cpp` so that the latitude spans a range of π instead of $\pi/2$. Your output might be something like Figure 2.60.

Exercise 2.41. (Programming) Just to get you thinking about animation, which we'll be studying in depth soon enough, guess the effect of replacing `glTranslatef(0.0, 0.0, -10.0)` with `glTranslatef(0.0, 0.0, -20.0)` in `hemisphere.cpp`. Verify.

And, here are some more things to draw.

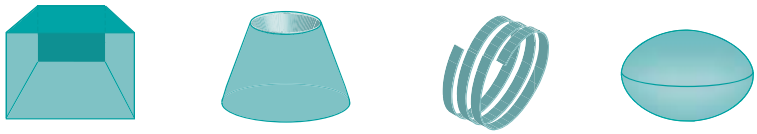

Figure 2.61: More things to draw.

Exercise 2.42. (Programming) Draw the objects shown in Figure 2.61. Give the user an option to toggle between filled and wireframe renderings. Borrow the `glRotatef()`s from `hemisphere.cpp` to allow an object to be turned.
Hint: A way to make the football, or ellipsoid, is to modify `hemisphere.cpp` to first make half an ellipsoid (a hemi-ellipsoid?). In fact, similarly to (2.3), a generic point on a hemi-ellipsoid is

$$(R_x \cos\phi\cos\theta,\ R_y \sin\phi,\ -R_z \cos\phi\sin\theta)$$

where the constants R_x, R_y and R_z are the lengths of the semi-principal axes. Two hemi-ellipsoids back-to-back would then give a whole ellipsoid. Or, you could follow the idea of Exercise 2.40 to make it as one object.

Remark 2.24. Filled renderings of 3D scenes, even with color, rarely look pleasant in the absence of lighting. See for yourself by applying color to 3D objects you have drawn so far (remember to invoke a `glPolygonMode(*, GL_FILL)` call). For this reason, we'll draw mostly wireframe till Chapter 11, which is all about lighting. You'll have to bear with this. Wireframe, however, fully exposes the geometry of an object, which is not a bad thing when one is learning object design.

2.11 An OpenGL Program End to End

We have already touched on almost every command of `square.cpp` which is functional from a graphics points of view. However, let's run over the whole program to see all that goes into making OpenGL code tick.

We start with `main()`:

1. `glutInit(&argc, argv)` initializes the FreeGLUT library. FreeGLUT [49], successor to GLUT (OpenGL Utility Toolkit), is a library of calls to manage a window and monitor mouse and keyboard input (the reason such a separate library is needed is that OpenGL itself is only a library of graphics calls).

2. ```
   glutInitContextVersion(4, 3);
   glutInitContextProfile(GLUT_COMPATIBILITY_PROFILE);
   ```

   tells FreeGLUT that the program will be wanting an OpenGL 4.3 *context* – this context being the interface between an instance of OpenGL and the rest of the system – which is backward-compatible in that legacy commands are implemented. This, for example, allows us to draw with the `glBegin()-glEnd()` operations from OpenGL 2.1, which do not belong in the core profile of OpenGL 4.3.

   Remark 2.25. If your graphics card doesn't support OpenGL 4.3 then the program may compile but not run as the system is unable to provide the context asked. What you might do in this case is thin the context by replacing the first line above with `glutInitContextVersion(3, 3)`, or even `glutInitContextVersion(2, 1)`, instead. Of course, then, programs using later-generation calls will not run, but you should be fine early on in the book.

3. `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA)` says that we'll be wanting an OpenGL context to support a single-buffered frame, each pixel having red, green, blue and alpha values.

4. ```
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   ```

   as we have already seen, set the size of the OpenGL window and the location of its top left corner on the computer screen.

5. `glutCreateWindow("square.cpp")` actually creates the OpenGL context and its associated window with the specified string parameter as title.

6. ```
   glutDisplayFunc(drawScene);
   glutReshapeFunc(resize);
   glutKeyboardFunc(keyInput);
   ```

   register the routines to call – so-called *callback routines* – when the OpenGL window is to be drawn, when it is resized, and when keyboard input is received, respectively.

7. ```
   glewExperimental = GL_TRUE;
   glewInit();
   ```

   initializes GLEW (the OpenGL Extension Wrangler Library) which handles the loading of OpenGL extensions, with the switch set so that extensions implemented even in pre-release drivers are exposed.

8. `setup()` invokes the initialization routine.

9. `glutMainLoop` begins the event-processing loop, calling registered callback routines as needed.

We have already seen that the only command in the initialization routine `setup()`, namely, `glClearColor(1.0, 1.0, 1.0, 0.0)`, specifies the clearing color of the OpenGL window.

The callback routine to draw the OpenGL window is:

```
void drawScene(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
   // Draw a polygon with specified vertices.
   glBegin(GL_POLYGON);
   ...
   glEnd();
   glFlush();
}
```

The first command clears the OpenGL window to the specified clearing color, in other words, paints in the background color. The next command `glColor3f()` sets the foreground, or drawing, color, which is used to draw the polygon specified within the `glBegin()-glEnd()` pair (we have already examined this polygon carefully). Finally, `glFlush()` forces all the commands in queue to actually execute – emptying or flushing the commands buffer as it were – which, in this case, means the polygon is drawn.

The callback routine when the OpenGL window is resized, and first created, is `void resize(int w, int h)`. The window manager supplies the width `w` and height `h` of the resized OpenGL window (or, initial window, when it is first created) as parameters to the resize routine.

The first command

```
glViewport(0, 0, w, h);
```

of `square.cpp`'s resize routine specifies the rectangular part of the OpenGL window in which actual drawing is to take place; with the given parameters it is the entire window. We'll be looking more carefully into `glViewPort()` and its applications in the next chapter.

The next three commands

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0);
```

activate the projection matrix stack, place the identity matrix at the top of this stack, and then multiply the identity matrix by the matrix corresponding to the final `glOrtho()` command. Don't worry if all this about matrices doesn't make much sense now – the takeway that the third statement above sets up the viewing box of `square.cpp` (as described in Section 2.2) is enough at this time. We'll be learning about OpenGL's matrix stacks in Chapters 4 and 5.

The final two commands

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

of the resize routine activate the modelview matrix stack and place the identity matrix at the top in readiness for modelview transformation commands in the drawing routine – commands which move objects, of which there happen to be none in `square.cpp`.

The callback routine to handle ASCII keys is `keyInput(unsigned char key, int x, int y)`. When an ASCII key is pressed it is passed in the parameter `char` to this callback routine, as is the location of the mouse in the parameters `x` and `y`. All that `keyInput` of `square.cpp` does is terminate the program when the escape key is pressed. In the next chapter we'll see callback routines to handle non-ASCII keys, as well as interaction via the mouse.

As the reader might well guess, the guts of an OpenGL program are in its drawing routine. Interestingly, the initialization routine often pulls a fair load, too, because one would want to locate there tasks that need to be done once at start-up, e.g., setting up data structures. In fact, it's a common beginner's mistake to place initialization chores in the drawing routine, as the latter is invoked repeatedly if there is animation, leading to inefficiency.

The other routines, such as `main()` and the interactivity and reshape callbacks, are simple to code and, in fact, can often be copied from one program to the next.

Remark 2.26. Another popular library for use with OpenGL in order to manage a window and handle input events is GLFW [57], the name derived from "Graphics Library Framework", which can serve as an alternate to FreeGLUT. All our programs use FreeGLUT. However, for those who have prior experience with GLFW, or might be curious, we include below a GLFW version of `square.cpp`.

Experiment 2.28. You will need to configure your environment to include GLFW before running `squareGLFW.cpp`. How to do this is described in the comments at the top of the program file. This program has the exact same functionality as `square.cpp`, only with GLFW replacing FreeGLUT. Figure 2.62 is a screenshot. **End**

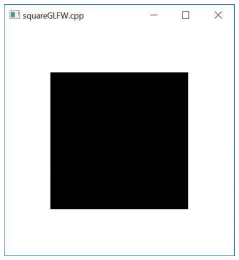

Figure 2.62: Screenshot of `squareGLFW.cpp`.

2.12 Summary, Notes and More Reading

In this chapter we began the study of 3D CG, looking at it through the "eyes" of OpenGL. OpenGL itself was presented to the extent that the reader acquires functional literacy in this particular API. The drawing primitives were probably the most important part of the API's vernacular.

We discovered as well how OpenGL functions as a state machine, attributes such as color defining the current state. Moreover, we learned that quantifiable attribute values, e.g., RGB color, are typically interpolated from the vertices of a primitive throughout its interior. We saw that OpenGL clips whatever the programmer draws to within a viewing volume, either a box or frustum.

Beyond acquaintance with the language, we were introduced as well to the synthetic-camera model of 3D graphics, which OpenGL implements via two kinds of projection: orthographic and perspective. This included insights into the world coordinate system, the viewing volume – box or frustum – which is the stage in which all drawings are made, the shoot-and-print rendering process to map a 3D scene to a 2D window, as well as hidden surface removal. We made a first acquaintance as well with another cornerstone of 3D graphics: the technique of simulating curved objects using straight and flat primitives like line segments and triangles.

Historically, OpenGL evolved from SGI's IRIS GL API, which popularized the approach to creating 3D scenes by drawing objects in actual 3-space and then rendering them to a 2D window by means of a synthetic camera. IRIS GL's efficiently pipelined architecture enabled high-speed rendering of animated 3D graphics and, consequently, made possible as well real-time interactive 3D. The ensuing demand from application developers for an open and portable (therefore, platform-independent) version of their API spurred SGI to create the first OpenGL specification in 1992, as well as a sample implementation. Soon after, the OpenGL ARB (Architecture Review Board), a consortium composed of a few leading companies in the graphics industry, was established to oversee the development of the API. Stewardship of the OpenGL specification passed in 2006 to the Khronos Group, a member-funded industry consortium dedicated to the creation of open-standard royalty-free API's. (That no *one* owns OpenGL is a good thing.) The canonical, and very useful, source for information about OpenGL is its own home page [108].

Microsoft has a non-open Windows-specific 3D API – Direct3D [93, 146] – which is popular among game programmers as it allows optimization for the pervasive Windows platform. However, outside of the games industry, where it nonetheless competes with

Direct3D, and leaving aside particular application domains with such high-quality rendering requirements that ray tracers are preferred, by far the dominant graphics API is OpenGL. The beginning graphics programmer should keep in mind though that recent versions of OpenGL and Direct3D are fairly alike in functionality – read an interesting comparison in Wikipedia [26] – so migrating from one to the other is not hard.

It's safe to say that OpenGL is the de facto standard 3D graphics API. A primary reason for this, other than the extremely well-thought-out design, is OpenGL's portability. It's worth knowing as well that, despite its intended portability, OpenGL can take advantage of platform-specific and card-specific capabilities via so-called extensions, at the cost of clumsier code.

But wait, there's more! A lighter version of OpenGL, namely, OpenGL ES (for Embedded Systems), is to be found supporting mobile 3D apps on almost every Android smartphone. And, WebGL, a derivation again of OpenGL, has become the standard for 3D graphics on the web. We'll learn WebGL ourselves in a later chapter.

Perhaps the best reason for OpenGL to be *the* API of choice for students of 3D computer graphics is – and this is a consequence of its almost universal adoption by the academic, engineering and scientific communities – the sheer volume of learning resources available. Not least among these is the number of textbooks that teach computer graphics with the help of OpenGL. Search amazon.com with the keywords "computer graphics opengl" and you'll see what we mean. Angel [2], Buss [21], de Vries [36], Govil-Pai [62], Hearn & Baker [71], Hill & Kelley [74] and McReynolds & Blythe [96] are some introductions to computer graphics via OpenGL that the author has learned much from.

Interestingly, an unofficial clone of OpenGL, Mesa 3D [97], which uses the same syntax, was originally developed by Brian Paul for the Unix/X11 platform, but there are ports now to other platforms as well.

In case the reader prefers not to be distracted by code, here are a few API-independent introductions: Akenine-Möller, Haines & Hoffman [1], Foley et al. [76, 48], Marschner & Shirley [95], Watt [152], Xiang [160] and Xiang & Plastock [161]. Keeping different books handy in the beginning is a good idea as, often, when you are having trouble grasping one author's exposition of a topic, turning to another for help with that matter may clear the way.

With regard to the code which comes with this book, we don't make much use of OpenGL-defined data types, which are prefixed with `GL`, e.g., `GLsizei`, `GLint`, etc., though the red book advocates doing so in order to avoid type mismatch issues when porting. Fortunately, we have not yet encountered a problem in any implementation of OpenGL that we've tried.

In addition to the code which comes with this book, the reader should try to acquire OpenGL programs from as many other sources as possible, as an efficient way to learn the language – any language as a matter of fact – is by modifying live code. There are numerous resources on the web for code, as well as tutorials, 3D models and texture images. The OpenGL site [108] has pointers to several coding tutorials. The book by Wright, Lipchak & Haemel [134] is specifically about programming OpenGL and has numerous example programs. The red book comes with example code as well.

Hard-earned wisdom: Write experiments of your own to clarify ideas. Even if you are sure in your mind that you understand something, do write a few lines of code in verification. As the author has repeatedly been, you too might be surprised.