Computer Graphics Through OpenGL: From Theory to Experiments

3rd Edition

by Sumanta Guha

Chapman & Hall/CRC

Experimenter Software

(Prepared by Chansophea Chuon and Sumanta Guha)

This file is to help you run the book experiments. It's in pdf format listing all the book experiments, with clickable hyperlinks on top of each, except for those in Chapter 1. One link brings up the source program file in text form, while another brings up its Visual C++ Windows project. Source code is not available for the four experiments in the first chapter, the reader being pointed instead to a folder with a Windows executable.

For experiments in the book asking simply to run a program, *Experimenter* brings up the source code and corresponding project, while for those asking that a program be modified significantly, the modified program – the modification being made either in the actual code or commented-out code – and its project come up. For example, an experiment often asks you to replace a part of a book program with a block of code listed in the text. In this case, *Experimenter* will throw up the modified program and project either ready to run or needing only a block to be uncommented, saving you the trouble of typing or pasting. For trivial modifications of a program, though, *Experimenter* links to the original program, leaving you to make changes.

Note: The names of the folders in `ExperimenterSource` – e.g., `ExperimentRotate-Teapot` for Experiment 4.7 of the text – mostly do not appear in the text itself and are of little relevance.

Experimenter is meant as a convenience; it certainly is not mandatory in order to run the programs or experiments, which all work perfectly well stand-alone. However, because of the way it is organized, *Experimenter* makes it easy to run the programs as one follows the text so we strongly recommend that readers use it. Instructors will find it particularly useful in synchronizing the presentation of theory with the running of code.

Installing and using Experimenter: Download the directory `ExperimenterSource` from the book's website `www.sumantaguha.com` and install it as a subfolder of the same folder where you have `Experimenter.pdf` (the file you are reading, also to be found at the book's site). It's best to use Adobe Reader to open `Experimenter.pdf` as other pdf readers might not be able to resolve the hyperlinks. Note that *Experimenter* will be slow in bringing up a Windows project the first time, as various config files have to be generated locally; it should be fast after that.

If you need to learn how to set up an environment in which to run OpenGL code, then you will find a guide at the book's website `www.sumantaguha.com` to installing OpenGL and running the book's programs.

Adding your own experiments to Experimenter: Presuming you are using Latex, first include the `hyperref` package in your document. In our case, we did so with the line

```
\usepackage[pdftex]{hyperref}
```

Subsequently, add hyperlinks as follows (a sample from *Experimenter*'s Latex file):

```
Click for \ic{square.cpp}~~~
\href{run:ExperimenterSource/Chapter2/Square/square.cpp}
```

```
    {{\color{red}\ic{Program}}}~~~
\href{run:ExperimenterSource/Chapter2/Square/Square.vcxproj}
    {{\color{red}\ic{Windows Project}}}
```

Once you have created your own *Experimenter*-like document with clickable hyperlinks, you can mix and match pages of it with *Experimenter* by using a pdf document manipulating tool.

We hope you find *Experimenter* of help as you read the book. All feedback is welcome: send mail to `sg@sumantaguha.com`.

Part I

Hello World

Chapter 1

An Invitation to Computer Graphics

Executable for Windows is in the folder
`ExperimenterSource/Chapter1/Ellipsoid`.

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

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

Drawing an ellipsoid with many triangles may seem a hard way to do things. Interestingly, and often surprisingly for the beginner, OpenGL offers the programmer only a tiny set of low-level geometric primitives with which to make objects – in fact, points, lines and triangles are, basically, it. So, a curved 3D object like an ellipsoid has to be made, or, more accurately, *approximated*, using triangles. But, as we shall see as we go along, the process really is not difficult.

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

Executable for Windows is in the folder
`ExperimenterSource/Chapter1/AnimatedGarden`.

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

As you can see from the wireframe, there's again a lot of triangles (in fact, the flowers might remind you of the ellipsoid from the previous program). The plant stems

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

are thick lines and, if you look carefully, you'll spot points as well. The one special effect this program has that `Ellipsoid` did not is blending, as is not hard to see. **End**

Executable for Windows is in the folder
`ExperimenterSource/Chapter1/BezierShoe`.

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

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

Executable for Windows is in the folder
`ExperimenterSource/Chapter1/Dominos`.

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

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

Chapter 2

On to OpenGL and 3D Computer Graphics

Click for `square.cpp` Program Windows Project

Experiment 2.1. Run `square.cpp`.

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

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

Click for `square.cpp` Program Windows Project

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

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

Click for `square.cpp` Program Windows Project

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

Click for `square.cpp` Program Windows Project

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**

Click for `square.cpp` modified Program Windows Project

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 seem. Clipping is a stage in the graphics pipeline. We'll not worry about its implementation at this time, only the effect. **End**

Click for `square.cpp` modified Program Windows Project

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**

Click for `square.cpp` Program Windows Project

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

Click for `square.cpp` modified Program Windows Project

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

Click for `square.cpp` modified Program Windows Project

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();
```

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

Click for `square.cpp` modified Program Windows Project

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();
```

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

Click for `square.cpp` modified Program Windows Project

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

Click for `square.cpp` modified Program Windows Project

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

Click for `square.cpp` modified Program Windows Project

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

Click for `square.cpp` modified Program Windows Project

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

Click for `square.cpp` modified Program Windows Project

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

Click for `square.cpp` modified Program Windows Project

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

Click for `square.cpp` modified Program Windows Project

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

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**

Click for `circle.cpp` Program Windows Project

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 * 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. **End**

Click for `parabola.cpp` Program Windows Project

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**

Click for `circularAnnuluses.cpp` Program Windows Project

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:

```
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

Click for `helix.cpp` Program Windows Project

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

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**

Click for `moveSphere.cpp` Program Windows Project

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**

Click for `strangeK.cpp` Program Windows Project

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**

Click for `hemisphere.cpp` Program Windows Project

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 * PI/2.0) *
                        cos(2.0 * (float)i/p * PI),
                    R * sin((float)(j+1)/q * PI/2.0),
                    -R * cos((float)(j+1)/q * PI/2.0) *
                        sin(2.0 * (float)i/p * PI));
         glVertex3f(R * cos((float)j/q * PI/2.0) *
                        cos(2.0 * (float)i/p * PI),
                    R * sin((float)j/q * PI/2.0),
                    -R * cos((float)j/q * PI/2.0) *
                        sin(2.0 *(float)i/p * 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**

Click for `hemisphere.cpp` Program Windows Project

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++)
      ...
```

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

Click for `squareGLFW.cpp` Program Windows Project

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

Part II

Tricks of the Trade

Chapter 3

An OpenGL Toolbox

Click for `squareAnnulus1.cpp` Program Windows Project

Experiment 3.1. Run `squareAnnulus1.cpp`. A screenshot is seen in Figure 3.1(a). Press the space bar to see the wireframe in Figure 3.1(b).

It is a plain-vanilla program which draws the square annulus diagrammed in Figure 3.2 with a single giant triangle strip containing 10 vertices and their color attributes (the last two vertices being identical to the first two in order to close the strip):

```
glBegin(GL_TRIANGLE_STRIP);
   glColor3f(0.0, 0.0, 0.0);
   glVertex3f(30.0, 30.0, 0.0); // Vertex 0
   glColor3f(1.0, 0.0, 0.0);
   glVertex3f(10.0, 10.0, 0.0); // Vertex 1
   glColor3f(0.0, 1.0, 0.0);
   glVertex3f(70.0, 30.0, 0.0); // Vertex 2
   ...
   glColor3f(0.0, 0.0, 0.0);
   glVertex3f(30.0, 30.0, 0.0); // Vertex 8 = Vertex 0
   glColor3f(1.0, 0.0, 0.0);
   glVertex3f(10.0, 10.0, 0.0); // Vertex 9 = Vertex 1
glEnd();
```

End

Click for `squareAnnulus2.cpp` Program Windows Project

Experiment 3.2. Run `squareAnnulus2.cpp`.

It draws the same annulus as `squareAnnulus1.cpp`, except that the vertex coordinates and color data are now separately stored in two-dimensional global arrays, `vertices` and `colors`, respectively. Moreover, in each iteration, the loop

```
glBegin(GL_TRIANGLE_STRIP);
   for(int i = 0; i < 10; ++i)
   {
      glColor3fv(colors[i%8]);
      glVertex3fv(vertices[i%8]);
   }
glEnd();
```

retrieves a vector of coordinate values by the *pointer form* (also called *vector* form) of vertex declaration, namely, `glVertex3fv(*pointer)`, and as well a vector of color values with the pointer form `glColor3fv(*pointer)`. End

Click for squareAnnulus3.cpp Program Windows Project

Experiment 3.3. Run squareAnnulus3.cpp.

It again draws the same colorful annulus as before. The coordinates and color data of the vertices are stored in global vertex arrays, vertices and colors, respectively, as in squareAnnulus2.cpp, except, now, the arrays are flat and not 2D (OpenGL assumes 1D arrays but, because of the way C++ stores arrays, we could, in fact, have specified vertices and colors as 2D arrays exactly as in squareAnnulus2.cpp if we had so chosen). End

Click for squareAnnulus4.cpp Program Windows Project

Experiment 3.4. Run squareAnnulus4.cpp.

The code is even more concise with the single draw call

```
glDrawElements(GL_TRIANGLE_STRIP, 10, GL_UNSIGNED_INT, stripIndices)
```

replacing the entire glBegin(GL_TRIANGLE_STRIP)-glEnd() block. End

Click for squareAnnulusAndTriangle.cpp Program Windows Project

Experiment 3.5. Run squareAnnulusAndTriangle.cpp, which adds a triangle inside the annulus of the squareAnnulus*.cpp programs. See Figure 3.3 for a screenshot. End

Click for hemisphereMultidraw.cpp Program Windows Project

Experiment 3.6. Run hemisphereMultidraw.cpp, whose sole purpose is to draw the loop

```
for(j = 0; j < q; j++)
{
   // One latitudinal triangle strip.
   glBegin(GL_TRIANGLE_STRIP);
   ...
}
```

of triangle strips of hemisphere.cpp using instead the single

```
glMultiDrawElements(GL_TRIANGLE_STRIP, countIndices,
                    GL_UNSIGNED_BYTE, (const void **)indices, q)
```

command. Figure 3.6 is a screenshot. End

Click for squareAnnulusVBO.cpp Program Windows Project

Experiment 3.7. Fire up squareAnnulusVBO.cpp, which modifies squareAnnulus4.cpp to store vertex-related data in VBOs. There is a simple animation, too, through periodically changing color values in a VBO. Figure 3.8 is a screenshot, colors having already changed. End

Click for hemisphereMultidrawVBO.cpp Program Windows Project

Experiment 3.8. Run hemisphereMultidrawVBO.cpp. A screenshot is in Figure 3.9. End

Click for `squareAnnulusAndTriangleVAO.cpp` Program Windows Project

Experiment 3.9. Run `squareAnnulusAndTriangleVAO.cpp`. This program builds on `squareAnnulusVBO.cpp`. We add to it the triangle from `squareAnnulusAndTriangle.cpp` in order to have two VAOs. Note, however, that we separate out the vertex coordinates and color arrays of the triangle, because interleaved, as in `squareAnnulusAndTriangle.cpp`, they are a bit trick to manage in a VBO. Otherwise, the outputs of the current program and `squareAnnulusAndTriangle.cpp` are identical – see Figure 3.11. End

Click for `helixList.cpp` Program Windows Project

Experiment 3.10. Run `helixList.cpp`, which shows six copies of the same helix, variously transformed and colored. Figure 3.12 is a screenshot. End

Click for `multipleLists.cpp` Program Windows Project

Experiment 3.11. Run `multipleLists.cpp`. See Figure 3.13 for a screenshot. Three display lists are defined in the program: to draw a red triangle, a green rectangle and a blue pentagon, respectively. End

Click for `fonts.cpp` Program Windows Project

Experiment 3.12. Run `fonts.cpp`. Displayed are the various fonts available through the FreeGLUT library. See Figure 3.15. End

Click for `mouse.cpp` Program Windows Project

Experiment 3.13. Run `mouse.cpp`. Click the left mouse button to draw points on the canvas and the right one to exit. Figure 3.16 is a screenshot of "OpenGL" scrawled in points. End

Click for `mouseMotion.cpp` Program Windows Project

Experiment 3.14. Run `mouseMotion.cpp`, which enhances `mouse.cpp` by allowing the user to drag the newly created point using the mouse with the left button still pressed. End

Click for `mouseWheel.cpp` Program Windows Project

Experiment 3.15. Run `mouseWheel.cpp`, which further enhances `mouseMotion.cpp` with the capability to change the size of the last point drawn by turning the mouse wheel. End

Click for `moveSphere.cpp` Program Windows Project

Experiment 3.16. Run `moveSphere.cpp`, a program we saw earlier in Experiment 2.24. Figure 3.18 is a screenshot. Press the left, right, up and down arrow keys to move the sphere, the space bar to rotate it and 'r' to reset.

Note how the `specialKeyInput()` routine is written to enable the arrow keys to change the location of the sphere. Subsequently, this routine is registered in `main()` as the handling routine for non-ASCII entry. End

Click for `menus.cpp` Program Windows Project

Experiment 3.17. Run `menus.cpp`. Press the right mouse button for menu options which allow you to change the color of the initially red square or exit. Figure 3.19 is a screenshot. **End**

Click for `lineStipple.cpp` Program Windows Project

Experiment 3.18. Run `lineStipple.cpp`. Press the space bar to cycle through five different stipples. A screenshot is shown in Figure 3.20. **End**

Click for `canvas.cpp` Program Windows Project

Experiment 3.19. Run `canvas.cpp`, a simple program to draw on a flat canvas with menu and mouse functionality.

Left click on an icon to select it. Then left click on the drawing area to draw – click once to draw a point, twice to draw a line or rectangle. Right click for a pop-up menu. Figure 3.22 is a screenshot. **End**

Click for `glutObjects.cpp` Program Windows Project

Experiment 3.20. Run `glutObjects.cpp`. Press the arrow keys to cycle through the various FreeGLUT objects and 'x/X', 'y/Y' and 'z/Z' to turn them. Figure 3.24 shows the teapot. **End**

Click for `clippingPlanes.cpp` Program Windows Project

Experiment 3.21. Run `clippingPlanes.cpp`, which augments `circularAnnuluses.cpp` with two additional clipping planes which can be toggled on and off by pressing '0' and '1', respectively.

The first plane (`GL_CLIP_PLANE0`) clips off the half-space $-z + 0.25 < 0$, i.e., $z > 0.25$, removing the floating white disc of the annulus on the upper-right. The second one (`GL_CLIP_PLANE1`) clips off the half-space $x + 0.5y < 60.0$, which is the space below an angled plane parallel to the z-axis. Figure 3.26 is a screenshot of both clipping planes activated. **End**

Click for `hemispherePerspective.cpp` Program Windows Project

Experiment 3.22. Run `hemispherePerspective.cpp`, which replaces the original `glFrustum()` statement (still there in comments) of `hemisphere.cpp` with the `gluPerspective()` statement just computed. Figure 3.31 is a screenshot. The output of the program with either the original `glFrustum()` call or the equivalent `gluPerspective()` is same, as it should be. **End**

Click for `hemispherePerspective.cpp` Program Windows Project

Experiment 3.23. Run again `hemispherePerspective.cpp`.

The initial OpenGL window is a square 500×500 pixels. Drag a corner to change its shape, making it tall and thin. The hemisphere is distorted to become ellipsoidal (Figure 3.32(a)). Distortion is identical with either

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

or

```
gluPerspective(90.0, 1.0, 5.0, 100.0);
```

as the two statements are equivalent. Next, replace the projection statement with

```
gluPerspective(90.0, (float)w/(float)h, 5.0, 100.0);
```

which sets the aspect ratio of the viewing frustum equal to that of the OpenGL window. Resize the window – the hemisphere is no longer distorted (Figure 3.32(b))! **End**

Click for `viewports.cpp` Program Windows Project

Experiment 3.24. Run `viewports.cpp` where the screen is split into two viewports, one defined by `glViewport(0, 0, width/2.0, height)` and the other by `glViewport(width/2.0, 0, width/2.0, height)`, with contents a square and a circle, respectively. Figure 3.34 is a screenshot, where it should be clear from the parameters of the two `glViewport()` calls why the viewports are as seen.

A vertical black line is drawn by the program at the left end of the second viewport to separate the two. As the aspect ratio of both viewports differs from that of the viewing face, the square and circle are squashed laterally. **End**

Click for `multipleWindows.cpp` Program Windows Project

Experiment 3.25. Run `multipleWindows.cpp`, which creates two top-level windows (Figure 3.35). As you see in `main()`, after the size and position of the first window is specified, it is created with the call `glutCreateWindow("window 1")`, following which a block of four statements specify its initialization, display, resize and keyboard routines, respectively. The first three of these routines are unique to the first window while the second is shared by both.

The second window is then likewise created after its size and position are specified. **End**

Part III

Movers and Shapers

Chapter 4

Transformation, Animation and Viewing

Click for `box.cpp` Program Windows Project

Experiment 4.1. Run `box.cpp`, which shows an axis-aligned – i.e., with sides parallel to the coordinate axes – FreeGLUT wireframe box of dimensions $5 \times 5 \times 5$. Figure 4.1 is a screenshot. Note the foreshortening – the back of the box appears smaller than the front – because of perspective projection in the viewing frustum specified by the `glFrustum()` statement.

Comment out the statement

```
glTranslatef(0.0, 0.0, -15.0);
```

What do you see now? *Nothing*! We'll explain why momentarily. End

Click for `box.cpp` Program Windows Project

Experiment 4.2. Successively replace the translation command of `box.cpp` with the following, making sure that what you see matches your understanding of where the command places the box. Keep in mind foreshortening, as well as clipping to within the viewing frustum.

1. `glTranslatef(0.0, 0.0, -10.0)` (see Figure 4.4)
2. `glTranslatef(0.0, 0.0, -5.0)`
3. `glTranslatef(0.0, 0.0, -25.0)`
4. `glTranslatef(10.0, 10.0, -15.0)`

End

Click for `box.cpp` modified Program Windows Project

Experiment 4.3. Add a scaling command, in particular, replace the modeling transformation block of `box.cpp` with:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
glScalef(2.0, 3.0, 1.0);
```

Figure 4.5 is a screenshot – compare with the unscaled box of Figure 4.1.

End

Click for box.cpp modified Program Windows Project

Experiment 4.4. An object less symmetric than a box is more interesting to work with. Care for some teapot? Accordingly, change the modeling transformation and object definition part of box.cpp to:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
glScalef(1.0, 1.0, 1.0);

glutWireTeapot(5.0); // Teapot.
```

Of course, glScalef(1.0, 1.0, 1.0) does nothing and we see the original unscaled teapot (Figure 4.7).

Next, successively change the scaling parameters by replacing the scaling command with the ones below. In each case, make sure your understanding of the command matches the change that you see in the shape of the teapot.

1. glScalef(2.0, 1.0, 1.0)
2. glScalef(1.0, 2.0, 1.0)
3. glScalef(1.0, 1.0, 2.0)

End

Click for box.cpp modified Program Windows Project

Experiment 4.5. Replace the cube of box.cpp with a square whose sides are not parallel to the coordinate axes. In particular, replace the modeling transformation and object definition part of that program with:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
// glScalef(1.0, 3.0, 1.0);

glBegin(GL_LINE_LOOP);
   glVertex3f(4.0, 0.0, 0.0);
   glVertex3f(0.0, 4.0, 0.0);
   glVertex3f(-4.0, 0.0, 0.0);
   glVertex3f(0.0, -4.0, 0.0);
glEnd();
```

See Figure 4.10(a). Uncomment the scaling. Now, see Figure 4.10(b). The square seems skewed to a parallelogram.

End

Click for box.cpp modified Program Windows Project

Experiment 4.6. Add a rotation command by replacing the modeling transformation and object definition part – we prefer a teapot – of box.cpp with:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
glRotatef(60.0, 0.0, 0.0, 1.0);

glutWireTeapot(5.0);
```

Figure 4.11 is a screenshot.

The *rotation* command `glRotatef(A, p, q, r)` rotates an object about an axis from the origin $O = (0, 0, 0)$ to the point (p, q, r). The amount of rotation is $A°$, measured CCW (counter-clockwise) looking *from* (p, q, r) to the origin. In this experiment, then, the rotation is 60° CCW looking down the z-axis. **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.7. Continuing with Experiment 4.6, successively replace the rotation command with the ones below, in each case trying to match what you see with your understanding of how the command should turn the teapot. (It can occasionally be a bit confusing because of the perspective projection.)

1. `glRotatef(60.0, 0.0, 0.0, -1.0)`
2. `glRotatef(-60.0, 0.0, 0.0, 1.0)`
3. `glRotatef(60.0, 1.0, 0.0, 0.0)`
4. `glRotatef(60.0, 0.0, 1.0, 0.0)`
5. `glRotatef(60.0, 1.0, 0.0, 1.0)` **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.8. Appropriately modify the code of Experiment 4.6 to compare the effects of each of the following pairs of rotation commands:

1. `glRotatef(60.0, 0.0, 0.0, 1.0)` and `glRotatef(60.0, 0.0, 0.0, 5.0)`
2. `glRotatef(60.0, 0.0, 2.0, 2.0)` and `glRotatef(60.0, 0.0, 3.5, 3.5)`
3. `glRotatef(60.0, 0.0, 0.0, -1.0)` and `glRotatef(60.0, 0.0, 0.0, -7.5)`

There is no difference in each case. One concludes that the rotation command `glRotatef(`A, p, q, r`)` is equivalent to `glRotatef(`$A, \alpha p, \alpha q, \alpha r$`)`, where α is any *positive* scalar. **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.9. Apply three modeling transformations by replacing the modeling transformations block of `box.cpp` with:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);

glTranslatef(10.0, 0.0, 0.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
```

It seems the box is *first* rotated 45° about the z-axis and *then* translated right 10 units. See Figure 4.17(a). The first translation `glTranslatef(0.0, 0.0, -15.0)`, of course, serves to "kick" the box down the z-axis into the viewing frustum.

Next, interchange the last two transformations, namely, the rightward translation and the rotation, by replacing the modeling transformations block with:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);

glRotatef(45.0, 0.0, 0.0, 1.0);
glTranslatef(10.0, 0.0, 0.0);
```

It seems that the box is now *first* translated right and *then* rotated about the z-axis causing it to rise. See Figure 4.17(b). **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.10. Modify the code of Experiment 4.5 to draw a square with a short line segment above it, both translated to the right; precisely, replace the modeling transformation and object definition part of `box.cpp` with:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
glTranslatef(10.0, 0.0, 0.0);

glBegin(GL_LINE_LOOP);
glVertex3f(4.0, 0.0, 0.0);
glVertex3f(0.0, 4.0, 0.0);
glVertex3f(-4.0, 0.0, 0.0);
glVertex3f(0.0, -4.0, 0.0);
glEnd();

glBegin(GL_LINES);
glVertex3f(0.0, 8.0, 0.0);
glVertex3f(0.0, 10.0, 0.0);
glEnd();
```

See Figure 4.19. Now, move the second translation statement `glTranslatef(10.0, 0.0, 0.0)` to just after the `glBegin(GL_LINE_LOOP)` statement. Are the square and segment still translated rightward? *No*! Experiment with a few more placements of `glTranslatef(10.0, 0.0, 0.0)`.

The conclusion is that OpenGL indeed treats objects as unitary: rogue transformation statements within a `glBegin(GL_`*primitive*`)-glEnd()` pair are ignored. **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.11. Replace the entire display routine of the original `box.cpp` with:

```
void drawScene(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
   glLoadIdentity();

   // Modeling transformations.
   glTranslatef(0.0, 0.0, -15.0);
   // glRotatef(45.0, 0.0, 0.0, 1.0);
   glTranslatef(5.0, 0.0, 0.0);

   glutWireCube(5.0); // Box.

   //More modeling transformations.
   glTranslatef(0.0, 10.0, 0.0);

   glutWireSphere(2.0, 10, 8); // Sphere.

   glFlush();
}
```

See Figure 4.21(a) for a screenshot. The objects are a box and a sphere.

End

Click for `box.cpp` modified Program Windows Project

Experiment 4.12. Continuing with the previous experiment, uncomment the `glRotatef()` statement. Figure 4.21(b) is a screenshot. End

Click for `box.cpp` modified Program Windows Project

Experiment 4.13. Repeat Experiment 4.12. The modeling transformation and object definition part are as below:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
glTranslatef(5.0, 0.0, 0.0);

glutWireCube(5.0); // Box.

//More modeling transformations.
glTranslatef (0.0, 10.0, 0.0);

glutWireSphere (2.0, 10, 8); // Sphere.
```

First, comment out the last two statements of the first modeling transformations block as below (the first translation is always needed to place the entire scene in the viewing frustum):

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
// glRotatef(45.0, 0.0, 0.0, 1.0);
// glTranslatef(5.0, 0.0, 0.0);

glutWireCube(5.0); // Box.

//More modeling transformations.
glTranslatef (0.0, 10.0, 0.0);

glutWireSphere (2.0, 10, 8); // Sphere.
```

The output is as depicted in Figure 4.23(a).

Next, uncomment `glTranslatef(5.0, 0.0, 0.0)` as below:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
// glRotatef(45.0, 0.0, 0.0, 1.0);
glTranslatef(5.0, 0.0, 0.0);

glutWireCube(5.0); // Box.

//More modeling transformations.
glTranslatef (0.0, 10.0, 0.0);

glutWireSphere (2.0, 10, 8); // Sphere.
```

The output is as in Figure 4.23(b). Finally, uncomment `glRotatef(45.0, 0.0, 0.0, 1.0)` as follows:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
glTranslatef(5.0, 0.0, 0.0);

glutWireCube(5.0); // Box.

//More modeling transformations.
glTranslatef (0.0, 10.0, 0.0);

glutWireSphere (2.0, 10, 8); // Sphere.

glFlush();
```

The result is seen in Figure 4.23(c). Figure 4.23 shows the box's local coordinate system as well after each transition. Observe that the sphere is *always* 10 units vertically above the box in the latter's coordinate system, as one would expect from the `glTranslatef (0.0, 10.0, 0.0)` call between the two. **End**

Click for `relativePlacement.cpp` Program Windows Project

Experiment 4.14. Run `relativePlacement.cpp`. Pressing the up arrow key once causes the last statement, viz., `drawBlueMan`, of the following piece of code to be executed:

```
glScalef(1.5, 0.75, 1.0);
glRotatef(30.0, 0.0, 0.0, 1.0);
glTranslatef(10.0, 0.0, 0.0);
drawRedMan; // Also draw grid in his local coordinate system.
glRotatef(45.0, 0.0, 0.0, 1.0);
glTranslatef(20.0, 0.0, 0.0);
drawBlueMan;
```

With each press of the up arrow we go back a statement and successively execute that statement *and* the ones that follow it. The statements executed are in black text, the rest gray. Pressing the down arrow key goes forward a statement. Figure 4.24 is a screenshot after all transformations from the scaling on have been executed. **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.15. We want to create a human-like character. Our plan is to start by drawing the torso as an elongated cube and placing a round sphere as its head directly on top of the cube (no neck for now). To this end replace the drawing routine of `box.cpp` with:

```
void drawScene(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
   glLoadIdentity();

   glTranslatef(0.0, 0.0, -15.0);

   glScalef(1.0, 2.0, 1.0);
   glutWireCube(5.0); // Box torso.

   glTranslatef(0.0, 7.0, 0.0);
   glutWireSphere(2.0, 10, 8); // Spherical head.
```

```
    glFlush();
}
```

Our calculations are as follows: (a) the scaled box is $5 \times 10 \times 5$ and, being centered at the origin, is 5 units long in the $+y$ direction; (b) the sphere is of radius 2; (c) therefore, if the sphere is translated $5 + 2 = 7$ in the $+y$ direction, then it should sit exactly on top of the box (see Figure 4.25(a)).

It doesn't work: the sphere is no longer round and is, moreover, some ways above the box (Figure 4.25(b)). Of course, because the sphere is transformed by `glScalef(1.0, 2.0, 1.0)` as well! So, what to do? A solution is to *isolate* the scaling by placing it within a *push-pop pair* as below:

```
...
glTranslatef(0.0, 0.0, -15.0);

glPushMatrix();
glScalef(1.0, 2.0, 1.0);
glutWireCube(5.0); // Box torso.
glPopMatrix();

glTranslatef(0.0, 7.0, 0.0);
glutWireSphere(2.0, 10, 8); // Spherical head.
...
```

The resulting screenshot is Figure 4.25(c), which shows a round head on a neckless torso as desired. **End**

Click for `rotatingHelix1.cpp` Program Windows Project

Experiment 4.16. Run `rotatingHelix1.cpp` where each press of space calls the `increaseAngle()` routine to turn the helix. Note the `glutPostRedisplay()` command in `increaseAngle()` which asks the screen to be redrawn. Keeping the space bar pressed turns the helix continuously. Figure 4.27 is a screenshot. **End**

Click for `rotatingHelix2.cpp` Program Windows Project

Experiment 4.17. Run `rotatingHelix2.cpp`, a slight modification of `rotatingHelix1.cpp`, where pressing space causes the routines `increaseAngle()` and `NULL` (do nothing) to be alternately specified as idle functions.

The speed of animation is determined by the processor's speed and availability to execute the idle function – the user cannot influence it. **End**

Click for `rotatingHelix3.cpp` Program Windows Project

Experiment 4.18. Run `rotatingHelix3.cpp`, another modification of `rotatingHelix1.cpp`, where the particular timer function `animate()` calls itself recursively after `animationPeriod` number of msecs., by means of its own `glutTimerFunc(animationPeriod, animate, 1)` statement.

The parameter value 1 passed to `animate()` is not used in this program. The routine `increaseAngle()` called by `animate()` turns the helix as before. Figure 4.28 shows the animation scheme.

The user can vary the speed of animation by changing the value of `animationPeriod` by pressing the up and down arrow keys. **End**

Click for `rotatingHelixFPS.cpp` Program Windows Project

Experiment 4.19. Run `rotatingHelixFPS.cpp`, which enhances `rotatingHelix3.cpp` adding the routine `frameCounter()` to count the number of times the `drawScene()` routine is called, equivalently, the number of frames drawn, per second. The "fps" is output every second to the debug window.

Caveat: The number of frames drawn per second by the GPU to the color buffer, i.e., the number of completions of `drawScene()` per second, may not equal exactly the number of frames drawn to the screen per second, the true fps. See Remark 4.8 a little further on, for example, for what may happen if a fast GPU is up against a slower monitor.

The way the fps is output is with `drawScene()` incrementing the global `frameCount` every time it is called, and `frameCounter()` outputting the value of `frameCount` each second – note that `frameCounter()` calls itself every second because of its `glutTimerFunc(1000, frameCounter, 1)` statement – while simultaneously resetting the value of `frameCount` to 0. The `if` conditional in `frameCounter()` is so that no fps is output when it is first called from `setup()` with `value` passed being 0. **End**

Click for `rotatingHelix2.cpp` Program Windows Project

Experiment 4.20. Disable double buffering in `rotatingHelix2.cpp` by replacing `GLUT_DOUBLE` with `GLUT_SINGLE` in the `glutInitDisplayMode()` call in `main`, and replacing `glutSwapBuffers()` in the drawing routine with `glFlush()`. Ghostly is it not?! **End**

Click for `ballAndTorus.cpp` Program Windows Project

Experiment 4.21. Run `ballAndTorus.cpp`. Press space to start the ball both flying around (longitudinal rotation) and in and out (latitudinal rotation) of the torus. Press the up and down arrow keys to change the speed of the animation. Press 'x/X', 'y/Y' and 'z/Z' to change the viewpoint. Figure 4.31 is a screenshot.

The animation of the ball is interesting and we'll deconstruct it. Comment out all the modeling transformations in the ball's block, except the last translation, as follows:

```
// Begin revolving ball.
// glRotatef(longAngle, 0.0, 0.0, 1.0);

// glTranslatef(12.0, 0.0, 0.0);
// glRotatef(latAngle, 0.0, 1.0, 0.0);
// glTranslatef(-12.0, 0.0, 0.0);

glTranslatef(20.0, 0.0, 0.0);

glColor3f(0.0, 0.0, 1.0);
glutWireSphere(2.0, 10, 10);
// End revolving ball.
```

The ball is centered at $(20, 0, 0)$, its start position, by `glTranslatef(20.0, 0.0, 0.0)`. See Figure 4.32. There is no animation even if you press space.

The ball's intended latitudinal rotation is in and out of the circle C_1 through the middle of the torus. C_1's radius, called the *outer radius* of the torus, is 12.0, as specified by the second parameter of `glutWireTorus(2.0, 12.0, 20, 20)`. Moreover, C_1 is centered at the origin and lies on the xy-plane. Therefore, ignoring longitudinal motion for now, the latitudinal rotation of the ball *from its start position* is about the

line L through $(12, 0, 0)$ parallel to the y-axis (L being tangent to C_1). This rotation will cause the ball's center to travel along the circle C_2 centered at $(12, 0, 0)$, lying on the xz-plane, of radius 8.

As `glRotatef()` always rotates about a radial axis, how does one obtain the desired rotation about L, a non-radial line? Employ the Trick (see Example 4.3 if you don't remember). First, translate left so that L is aligned along the y-axis, then rotate about the y-axis and, finally, reverse the first translation to bring L back to where it was. This means uncommenting the corresponding three modeling transformations as below:

```
// Begin revolving ball.
// glRotatef(longAngle, 0.0, 0.0, 1.0);

glTranslatef(12.0, 0.0, 0.0);
glRotatef(latAngle, 0.0, 1.0, 0.0);
glTranslatef(-12.0, 0.0, 0.0);

glTranslatef(20.0, 0.0, 0.0);

glColor3f(0.0, 0.0, 1.0);
glutWireSphere(2.0, 10, 10);
// End revolving ball.
```

Press space to view only latitudinal rotation.

Note: The two consecutive translation statements could be combined into one, but then the code would be less easy to parse.

Finally, uncomment `glRotatef(longAngle, 0.0, 0.0, 1.0)` to implement longitudinal rotation about the z-axis. The angular speed of longitudinal rotation is set to be five times slower than that of latitudinal rotation – the increments to `longAngle` and `latAngle` in the `animate()` routine being 1° and 5°, respectively. This means the ball winds in and out of the torus five times before it completes one trip around.
End

Click for `ballAndTorus.cpp` modified Program Windows Project

Experiment 4.22. We want to add a satellite which tags along with the ball of `ballAndTorus.cpp`. The following piece of code added to the end of the drawing routine – just after `// End revolving ball.` – does the job:

```
glTranslatef(4.0, 0.0, 0.0);

// Satellite
glColor3f(1.0, 0.0, 0.0);
glutWireSphere(0.5, 5, 5);
```

See Figure 4.33 for a screenshot. For a revolving satellite add the following instead:

```
glRotatef(10*latAngle, 0.0, 1.0, 0.0);
glTranslatef(4.0, 0.0, 0.0);

// Satellite
glColor3f(1.0, 0.0, 0.0);
glutWireSphere(0.5, 5, 5);
```

Observe how Proposition 4.1 is being applied in both cases to determine the motion of the satellite *relative to the* ball by means of transformation statements between the two. **End**

Click for `throwBall.cpp` Program Windows Project

Experiment 4.23. Run `throwBall.cpp`, which simulates the motion of a ball thrown with a specified initial velocity subject to the force of gravity. Figure 4.34 is a screenshot partway through the ball's flight.

Press space to toggle between animation on and off. Press the right/left arrow keys to increase/decrease the horizontal component of the initial velocity, up/down arrow keys to likewise change the vertical component of the initial velocity and the page up/down keys to change the gravitational acceleration. Press 'r' to reset. The values of the initial velocity components and of gravitational acceleration are displayed on the screen.

End

Click for `ballAndTorusWithFriction.cpp` Program Windows Project

Experiment 4.24. Run `ballAndTorusWithFriction.cpp`, which modifies `ballAndTorus.cpp` to simulate an invisible viscous medium through which the ball travels.

Press space to apply force to the ball. It has to be kept pressed in order to continue applying force. The ball comes to a gradual halt after the space bar is released. Increase or decrease the level of applied force by using the up and down arrow keys. Increase or decrease the viscosity of the medium using the page up and down keys. Press 'x/X', 'y/Y' and 'z/Z' to rotate the scene. Figure 4.36 is a screenshot. The values of the applied force and viscous drag are shown at the top. **End**

Click for `flag.cpp` Program Windows Project

Experiment 4.25. Run `flag.cpp`, which simulates a flag fluttering in the wind. Press space to start the animation, the up and down arrow keys to change its speed, and 'x/X', 'y/Y' and 'z/Z' to rotate the viewpoint. Figure 4.37 is a screenshot.

End

Click for `clown3.cpp` with parts corresponding to `clown1.cpp` and `clown2.cpp` identified Program Windows Project

Experiment 4.26. We start with only a blue sphere for the head by running `clown1.cpp` (see note next) which has the drawing routine below:

Note: `clown1.cpp` and `clown2.cpp` are not separate programs but incremental stages of `clown3.cpp` which is in `ExperimenterSource/Chapter4`. Currently, `clown3.cpp` is functionally actually `clown1.cpp`, drawing just a blue sphere because additions corresponding to `clown2.cpp` and `clown3.cpp` are commented out. The program indicates clearly the parts added by `clown2.cpp` and `clown3.cpp`, so uncomment them to get those programs.

```
void drawScene(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glLoadIdentity();

  // Place scene in frustum.
   glTranslatef(0.0, 0.0, -9.0);

   // Head.
   glColor3f(0.0, 0.0, 1.0);
   glutWireSphere(2.0, 20, 20);
```

```
   glutSwapBuffers();
}
```

Figure 4.39(a) is a screenshot.

Next, we want a green conical hat. The command `glutWireCone(`*base*`,` *height*`,` *slices*`,` *stacks*`)` draws a wireframe cone of base radius *base* and height *height*. The base of the cone lies on the xy-plane with its axis along the z-axis and its apex pointing in the positive direction of the z-axis. See Figure 4.40(a). The parameters *slices* and *stacks* determine the fineness of the mesh (not shown in the figure).

Accordingly, insert the lines (as entire `clown2.cpp` is already in `clown3.cpp`, you might want to simply uncomment the selected lines):

```
   // Hat.
   glColor3f(0.0, 1.0, 0.0);
   glutWireCone(2.0, 4.0, 20, 20);
```

in `clown1.cpp` after the call that draws the sphere, so that the drawing routine becomes:

```
void drawScene(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glLoadIdentity();

   // Place scene in frustum.
   glTranslatef(0.0, 0.0, -9.0);

   // Head.
   glColor3f(0.0, 0.0, 1.0);
   glutWireSphere(2.0, 20, 20);

   // Hat.
   glColor3f(0.0, 1.0, 0.0);
   glutWireCone(2.0, 5.0, 20, 20);

   glutSwapBuffers();
}
```

Not good! Because of the way `glutWireCone()` aligns, the hat covers the clown's face. This is easily fixed. Translate the hat 2 units up the z-axis and rotate it $-90°$ about the x-axis to arrange it on top of the head. Finally, rotate it a rakish 30° about the z-axis! Here's the modified drawing routine of `clown1.cpp` at this point:

```
void drawScene(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glLoadIdentity();

   // Place scene in frustum.
   glTranslatef(0.0, 0.0, -9.0);

   // Head.
   glColor3f(0.0, 0.0, 1.0);
   glutWireSphere(2.0, 20, 20);

   // Transformations of the hat.
   glRotatef(30.0, 0.0, 0.0, 1.0);
   glRotatef(-90.0, 1.0, 0.0, 0.0);
   glTranslatef(0.0, 0.0, 2.0);
```

```
   // Hat.
   glColor3f(0.0, 1.0, 0.0);
   glutWireCone(2.0, 5.0, 20, 20);

   glutSwapBuffers();
}
```

Let's add a brim to the hat by attaching a torus to its base. The command `glutWireTorus(`*inRadius*`,` *outRadius*`,` *sides*`,` *rings*`)` draws a wireframe torus of inner radius *inRadius* (the radius of a circular section of the torus), and outer radius *outRadius* (the radius of the circle through the middle of the torus). The axis of the torus is along the z-axis and centered at the origin. See Figure 4.40(b). Insert the call `glutWireTorus(0.2, 2.2, 10, 25)` right after the call that draws the cone, so the drawing routine becomes:

```
void drawScene(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glLoadIdentity();

   // Place scene in frustum.
   glTranslatef(0.0, 0.0, -9.0);

   // Head.
   glColor3f(0.0, 0.0, 1.0);
   glutWireSphere(2.0, 20, 20);

   // Transformations of the hat and brim.
   glRotatef(30.0, 0.0, 0.0, 1.0);
   glRotatef(-90.0, 1.0, 0.0, 0.0);
   glTranslatef(0.0, 0.0, 2.0);

   // Hat.
   glColor3f(0.0, 1.0, 0.0);
   glutWireCone(2.0, 5.0, 20, 20);

   // Brim.
   glutWireTorus(0.2, 2.2, 10, 25);

   glutSwapBuffers();
}
```

Observe that the brim is drawn suitably at the bottom of the hat and stays there despite modeling transformations between head and hat – a consequence of Proposition 4.1.

To animate, let's spin the hat about the clown's head by rotating it around the y-axis. We rig the space bar to toggle between animation on and off and the up/down arrow keys to change speed. All updates so far are included in `clown2.cpp`. Figure 4.39(b) is a screenshot.

What's a clown without little red ears that pop in and out?! Spheres will do for ears. An easy way to bring about oscillatory motion is to make use of the function $\sin(angle)$ which varies between -1 and 1. Begin by translating either ear a unit distance from the head, and then repeatedly translate each a distance of $\sin(angle)$, incrementing *angle* each time.

Note: A technicality one needs to be aware of in such applications is that angle is measured in *degrees* in OpenGL syntax, e.g., in `glRotatef(`*angle*`,` p`,` q`,` r`)`, while the C++ math library assumes angles to be given in *radians*. Multiplying by $\pi/180$ converts degrees to radians.

The ears and head are physically separate, though. Let's connect them with

springs! Helixes are springs. We borrow code from `helix.cpp`, but modify it to make the length of the helix 1, its axis along the x-axis and its radius 0.25. As the ears move, either helix is scaled along the x-axis so that it spans the gap between the head and an ear. The completed program is `clown3.cpp`, of which a screenshot is seen in Figure 4.39(c). **End**

Click for `floweringPlant.cpp` Program Windows Project

Experiment 4.27. Run `floweringPlant.cpp`, an animation of a flower blooming. Press space to start and stop animation, delete to reset, and 'x/X', 'y/Y' and 'z/Z' to change the viewpoint. Figure 4.41 is a screenshot.

End

Click for `box.cpp` Program Windows Project

Experiment 4.28. Replace the translation command

```
glTranslatef(0.0, 0.0, -15.0)
```

of `box.cpp` with the viewing command

```
gluLookAt(0.0, 0.0, 15.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)
```

There is no change in what is viewed (Figure 4.45). In fact, the commands `glTranslatef(0.0, 0.0, -15.0)` and `gluLookAt(0.0, 0.0, 15.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)` are *exactly equivalent*. **End**

Click for `boxWithLookAt.cpp` Program Windows Project

Experiment 4.29. Continue the previous experiment, or run `boxWithLookAt.cpp`, successively changing only the parameters *centerx*, *centery*, *centerz* – the middle three – of the `gluLookAt()` call to the following:

1. $0.0, 0.0, 10.0$
2. $0.0, 0.0, -10.0$
3. $0.0, 0.0, 20.0$
4. $0.0, 0.0, 15.0$

End

Click for `boxWithLookAt.cpp` modified Program Windows Project

Experiment 4.30. Restore the original `boxWithLookAt.cpp` program with its `gluLookAt(0.0, 0.0, 15.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)` call and, again, first replace the box with a `glutWireTeapot(5.0)`. Run: a screenshot is shown in Figure 4.48(a). Next, successively change the parameters *upx*, *upy*, *upz* – the last three parameters of `gluLookAt()` – to the following:

1. $1.0, 0.0, 0.0$ (Figure 4.48(b))
2. $0.0, -1.0, 0.0$ (Figure 4.48(c))
3. $1.0, 1.0, 0.0$ (Figure 4.48(d))

Screenshots of the successive cases are shown in Figures 4.48(b)-(d). The camera appears to *rotate* about its line of sight, the z-axis, so that its up direction points along the *up* vector (*upx*, *upy*, *upz*) each time. **End**

Click for `boxWithLookAt.cpp` Program Windows Project

Experiment 4.31. Replace the wire cube of `boxWithLookAt.cpp` with a `glutWireTeapot(5.0)` and replace its `gluLookAt()` call instead with:

```
gluLookAt(0.0, 0.0, 15.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0)
```

The vector $los = (0.0, 0.0, 0.0) - (0.0, 0.0, 15.0) = (0.0, 0.0, -15.0)$, which is down the z-axis. The component of $up = (1.0, 1.0, 1.0)$, perpendicular to the z-axis, is (1.0, 1.0, 0.0), which, therefore, is the up direction. Is what you see the same as Figure 4.48(d), which, in fact, is a screenshot for `gluLookAt(0.0, 0.0, 15.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0)`? **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.32. Replace the one modeling transformation statement of `box.cpp` with the block:

```
gluLookAt(0.0, 0.0, 15.0, 15.0, 0.0, 0.0, 0.0, 1.0, 0.0);

// glRotatef(45.0, 0.0, 1.0, 0.0);
// glTranslatef(0.0, 0.0, -15.0);
```

Run. Next, both comment out the viewing transformation and uncomment the two modeling transformations following it. Run again. The displayed output, shown in Figure 4.58, is the same in both cases. The reason, as Figures 4.59(a)-(c) explain, is that the viewing transformation is equivalent to the modeling transformation block. In particular, the former is undone by the latter. **End**

Click for `spaceTravel.cpp` Program Windows Project

Experiment 4.33. Run `spaceTravel.cpp`. The left viewport shows a global view from a fixed camera of a conical spacecraft and 48 stationary spherical asteroids arranged in a 6×8 grid. The right viewport shows the view from a front-facing camera attached to the tip of the craft.

Press the up and down arrow keys to move the craft forward and backward and the left and right arrow keys to turn it. Approximate collision detection is implemented to prevent the craft from crashing into an asteroid. See Figure 4.62 for a screenshot after the spacecraft has traveled and turned a bit. The frame rate of the program is shown in the debug window.

The asteroid grid can be changed in size by redefining `ROWS` and `COLUMNS`. The probability that a particular row-column slot is filled is specified as a percentage by `FILL_PROBABILITY` – a value less than 100 leads to a non-uniform distribution of asteroids. **End**

Click for `spaceTravel.cpp` Program Windows Project

Experiment 4.34. Run `spaceTravel.cpp` after increasing both `ROWS` and `COLUMNS` to 100. The spacecraft now begins to respond so slowly to key input that its movement

seems clunky, unless, of course, you have a very powerful machine (in which case, increase the values of `ROWS` and `COLUMNS` even more).

In fact, on our fairly fast desktop, there was almost no lag in response to key presses with the original 6×8 grid of asteroids – for example, we got a steady frame rate of 25 fps by keeping the up arrow key pressed, sending the craft straight ahead. However, we could never exceed 3 fps through the 100×100 grid. **End**

Click for `animateMan1.cpp` Program Windows Project

Experiment 4.35. Run `animateMan1.cpp`. This is a fairly complex program to develop a sequence of key frames for a man-like figure, which can subsequently be animated. In addition to its spherical head, the figure consists of nine box-like body parts which can rotate about their joints. Figure 4.65 shows the opening screen. All parts are wireframe. **End**

Click for `animateMan2.cpp` Program Windows Project

Experiment 4.36. Run `animateMan2.cpp`. This is simply a pared-down version of `animateMan1.cpp`, whose purpose is to animate the sequence of configurations listed in the file `animateManDataIn.txt`, likely generated from the develop mode of `animateMan1.cpp`. Press 'a' to toggle between animation on/off. As in `animateMan1.cpp`, pressing the up or down arrow key speeds up or slows down the animation. The camera functionalities via the keys 'r/R' and 'z/Z' remain as well. Think of `animateMan1.cpp` as the studio and `animateMan2.cpp` as a movie theater.

The current contents of `animateManDataIn.txt` cause the man to do a handspring over the ball. Figure 4.67 is a screenshot. **End**

Click for `ballAndTorusLitOrthoShadowed.cpp` Program Windows Project

Experiment 4.37. Run `ballAndTorusLitOrthoShadowed.cpp` of Chapter 11. This program, obviously based on `ballAndTorus.cpp`, has lighting, as well as shadows drawn on a checkered floor. Press space to start the ball traveling around the torus and the up and down arrow keys to change its speed. Figure 4.68 is a screenshot. **End**

Click for `selection.cpp` Program Windows Project

Experiment 4.38. Run `selection.cpp`, which is inspired by a similar program in the red book. It uses selection mode to determine the identity of rectangles, drawn with calls to `drawRectangle()`, which intersect the viewing volume created by the projection statement `glOrtho (-5.0, 5.0, -5.0, 5.0, -5.0, 5.0)`, this being a $10 \times 10 \times 10$ axis-aligned box centered at the origin. Figure 4.70 is a screenshot. Hit records are output to the command window. In the discussion following, we parse the program carefully. **End**

Click for `ballAndTorusPicking.cpp` Program Windows Project

Experiment 4.39. Run `ballAndTorusPicking.cpp`, which preserves all the functionality of `ballAndTorus.cpp` upon which it is based and adds the capability of picking the ball or torus with a left click of the mouse. The picked object blushes. See Figure 4.73 for a screenshot. **End**

Chapter 5

Inside Animation: The Theory of Transformations

Click for `box.cpp` modified Program Windows Project

Experiment 5.1. Fire up `box.cpp` and insert a rotation command – in fact, the same one as in the previous exercise – just before the box definition so that the transformation and object definition part of the drawing routine becomes:

```
// Modeling transformations.
glTranslatef(0.0, 0.0, -15.0);

glRotatef(90.0, 0.0, 1.0, 1.0);
glutWireCube(5.0); // Box.
```

The rotation command asks to rotate 90° about the line l from the origin through $(0, 1, 1)$. See Figure 5.23(a) for the displayed output.

Let's try now, instead, to use the strategy suggested above to express the given rotation in terms of rotations about the coordinate axes. Figure 5.23(b) illustrates the following simple scheme. One can align l along the z-axis by rotating it 45° about the x-axis. Therefore, the given rotation should be equivalent to (1) a rotation of 45° about the x-axis, followed by (2) a rotation of 90° about the z-axis followed, finally, by a (3) rotation of $-45°$ about the x-axis.

Give it a whirl. Replace the single rotation command `glRotatef(90.0, 0.0, 1.0, 1.0)` with a block of three as follows:

```
glRotatef(-45.0, 1.0, 0.0, 0.0);
glRotatef(90.0, 0.0, 0.0, 1.0);
glRotatef(45.0, 1.0, 0.0, 0.0);
```

Seeing is believing, is it not?! End

Click for `manipulateModelviewMatrix.cpp` Program Windows Project

Experiment 5.2. Run `manipulateModelviewMatrix.cpp`. Figure 5.30 is a screenshot, although we are now really more interested in the transformations in the program rather than its visual output.

The `displayModelviewMatrix()` routine outputs the current modelview matrix to the C++ window in conventional rectangular form. It is invoked four times in `drawScene()`, first right after the `glLoadIdentity()` command, and then successively after the `gluLookAt()`, `glMultMatrixf()` and `glScalef()` calls. We'll see next if the four output values match our understanding of theory. End

Chapter 6

Advanced Animation Techniques

Click for `spaceTravel.cpp` Program Windows Project

Experiment 6.1. First, run the program `spaceTravel.cpp` of Chapter 4 with its current values of `ROWS` and `COLUMNS`, determining the size of the asteroid grid, as 8 and 6, respectively. Move the spacecraft with the arrow keys.

Next, increase both `ROWS` and `COLUMNS` to 100. Figure 6.1 is a screenshot. The spacecraft now begins to respond sluggishly to the arrow keys, at least on a typical desktop. You may have to pump up even more the values of `ROWS` and `COLUMNS` if yours is exceptionally fast. End

Click for `spaceTravelFrustumCulled.cpp` Program Windows Project

Experiment 6.2. Run `spaceTravelFrustumCulled.cpp`, which enhances `spaceTravel.cpp` with optional quadtree-based frustum culling. Pressing space toggles between frustum culling enabled and disabled. As before, the arrow keys maneuver the craft.

The current size of the asteroid field is 100×100. Move the craft around toggling between frustum culling off and on. Dramatic isn't it, the speed-up from frustum culling?!

Note: When the number of asteroids is large, the display may take a while to come up because of pre-processing to build the quadtree structure.

End

Click for `occlusion.cpp` Program Windows Project

Experiment 6.3. Run `occlusion.cpp`, which initially shows only a green rectangle. Use the arrow keys to move a solid red cube into view from behind the rectangle. Press the space bar, which is a toggle, to reveal a blue wire sphere contained in the box. See Figure 6.7. The sphere plays the role of a complex "occludee", while the cube is its simple bounding box, and the rectangle is the "occluder".

We'll analyze this program next. End

Click for `occlusionConditionalRendering.cpp` Program Windows Project

Experiment 6.4. Run `occlusionConditionalRendering.cpp`. The major change from `occlusion.cpp` is in the following code block in the drawing routine, where the

rendering of the sphere is made conditional upon the outcome of the query with id `query`. The parameter `GL_QUERY_WAIT` says to wait for the query to complete.

```
glBeginConditionalRender(query, GL_QUERY_WAIT);
glColor3f(0.0, 0.0, 1.0);
glutWireSphere(0.5, 16, 10);
glEndConditionalRender();
```

Programmed into the block above is what to render depending on the query outcome. There is no need, therefore, for user-instigated determination of results, so the variables `result` and `resultAvailable` are now gone. **End**

Click for `spaceTravelFrustumCulledTimerQuery` Program Windows Project

Experiment 6.5. Run `spaceTravelFrustumCulledTimerQuery.cpp`. The program is simply `spaceTravelFrustumCulled.cpp` enhanced with timer queries to count the total time per frame spent in drawing the asteroids in both left and right viewports. This is output in milliseconds to the debug window. Figure 6.8 is a screengrab of the latter where, despite the small size, the reader might still be able to make out the transition from three-digit values to single digits and back to three, as we switch from frustum culling off to on and then off again. Read on to see how timer queries have, in fact, been incorporated into this program. **End**

Click for `eulerAngles.cpp` Program Windows Project

Experiment 6.6. Run `eulerAngles.cpp`, which shows an L, similar to the one in Figure 6.10(a), whose orientation can be interactively changed.

The initial orientation (Figure 6.11) of the L is analogous to the default pose of the OpenGL camera. Pressing 'x/X', 'y/Y' and 'z/Z' changes the L's Euler angles and delete resets. The Euler angle values are displayed on-screen. **End**

Click for `interpolateEulerAngles.cpp` Program Windows Project

Experiment 6.7. Run `interpolateEulerAngles.cpp`, which is based on `eulerAngles.cpp`. It simultaneously interpolates between the tuples $(0, 0, 0)$ and $(0, 90, 0)$ and between $(0, 0, 0)$ and $(-90, 90, 90)$. Press the left and right arrow keys to step through the interpolations (delete resets). For the first interpolation (green L) the successive tuples are $(0, angle, 0)$ and for the second (red L) they are $(-angle, angle, angle)$, $angle$ changing by 5 at each step in both, between 0 and 90.

The paths are different! The green L seems to follow the intuitively straighter path by keeping its long leg always on the xz-plane as it rotates about the y-axis, while the red L arcs below the xz-plane, as diagrammed in Figure 6.12. Figure 6.13 is a screenshot of `interpolateEulerAngles.cpp` part way through the interpolation. **End**

Click for `eulerAngles.cpp` Program Windows Project

Experiment 6.8. Run `eulerAngles.cpp` again.

Press 'x' and 'X' a few times each – the L turns longitudinally. Reset by pressing delete. Press 'y' and 'Y' a few times each – the L turns latitudinally. Reset. Press 'z' and 'Z' a few times each – the L twists. There appear to be three physical *degrees*

of freedom of the L derived from rotation about the three coordinate axes which we might descriptively term, respectively, longitudinal, latitudinal and twisting.

Now, from the initial configuration of `eulerAngles.cpp` press 'y' till $\beta = 90$. Next, press 'z' or 'Z' – the L twists. Then press 'x' or 'X' – the L still twists! **End**

Click for `quaternionAnimation.cpp` Program Windows Project

Experiment 6.9. Run `quaternionAnimation.cpp`, which applies the preceding ideas to animate the orientation of our favorite rigid body, an L, with the help of quaternions. Press 'x/X', 'y/Y' and 'z/Z' to change the orientation of the blue L, whose current Euler angles are shown on the display. Its start orientation is the initially stationary red L. See Figure 6.18 for a screenshot.

Pressing enter at any time begins an animation of the red L from its initial disposition to the blue's current orientation. Press the up and down arrow keys to change the speed and delete to reset. **End**

Part IV

Geometry for the Home Office

Chapter 7

Convexity and Interpolation

Click for `square.cpp` modified Program Windows Project

Experiment 7.1. Replace the polygon declaration part of our old favorite `square.cpp` with the following:

```
glBegin(GL_TRIANGLES);
   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);
glEnd();
```

Observe how OpenGL interpolates vertex color values throughout the triangle. Figure 7.7 is a screenshot. We'll check next if it's doing this the way described above. End

Click for `interpolation.cpp` Program Windows Project

Experiment 7.2. Run `interpolation.cpp`, which shows the interpolated colors of a movable point inside a triangle with red, green and blue vertices. The triangle itself is drawn white. See Figure 7.8 for a screenshot.

As the arrow keys are used to move the large point, the height of each of the three vertical bars on the left indicates the weight of the respective triangle vertex on the point's location. The color of the large point itself is interpolated (by the program) from those of the vertices. End

Click for `convexHull.cpp` Program Windows Project

Experiment 7.3. Run `convexHull.cpp`, which shows the convex hull of 8 points on a plane. Use the space bar to select a point and the arrow keys to move it. Figure 7.14 is a screenshot.

Note: The program implements a very inefficient (but easily coded) algorithm to compute the convex hull of a set F as the union of all triangles with vertices in F. End

Chapter 8

Triangulation

Click for `invalidTriangulation.cpp` Program Windows Project

Experiment 8.1. Run `invalidTriangulation.cpp`, which implements exactly the invalid triangulation $\{ABC, DBC, DAE\}$ of the rectangle in Figure 8.3(d). Colors have been arbitrarily fixed for the five vertices A-E. Press space to interchange the order that ABC and DBC appear in the code. Figure 8.4 shows the difference. End

Click for `square.cpp` modified Program Windows Project

Experiment 8.2. 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 8.8(a)).

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 identical to the one output earlier (see Figure 8.8(b))! Because the cyclic order of the vertices is unchanged, shouldn't it be as in Figure 8.8(a) both filled and outlined? End

Chapter 9

Orientation

Click for `square.cpp` Program Windows Project

Experiment 9.1. Replace the polygon declaration part of `square.cpp` of Chapter 2 with:

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
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();
```

This simply adds the two `glPolygonMode()` statements to the original `square.cpp`. In particular, they specify that front-facing polygons are to be drawn in outline and back-facing ones filled. Now, the order of the vertices is (20.0, 20.0, 0.0), (80.0, 20.0, 0.0), (80.0, 80.0, 0.0), (20.0, 80.0, 0.0), which appears CCW from the viewing face. Therefore, the square is drawn in outline (Figure 9.8).

Next, rotate the vertices cyclically so that the declaration becomes:

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
glBegin(GL_POLYGON);
   glVertex3f(20.0, 80.0, 0.0);
   glVertex3f(20.0, 20.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
   glVertex3f(80.0, 80.0, 0.0);
glEnd();
```

As the vertex order remains equivalent to the previous one, the square is still outlined.

Reverse the vertex listing next:

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
glBegin(GL_POLYGON);
   glVertex3f(80.0, 80.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
   glVertex3f(20.0, 20.0, 0.0);
   glVertex3f(20.0, 80.0, 0.0);
glEnd();
```

The square is drawn filled as the vertex order now appears CW from the front of the viewing box (Figure 9.9). End

Click for square.cpp Program Windows Project

Experiment 9.2. Replace the polygon declaration part of square.cpp with:

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
glBegin(GL_TRIANGLES);
   // CCW
   glVertex3f(20.0, 80.0, 0.0);
   glVertex3f(20.0, 20.0, 0.0);
   glVertex3f(50.0, 80.0, 0.0);

   //CCW
   glVertex3f(50.0, 80.0, 0.0);
   glVertex3f(20.0, 20.0, 0.0);
   glVertex3f(50.0, 20.0, 0.0);

   // CW
   glVertex3f(50.0, 20.0, 0.0);
   glVertex3f(50.0, 80.0, 0.0);
   glVertex3f(80.0, 80.0, 0.0);

   // CCW
   glVertex3f(80.0, 80.0, 0.0);
   glVertex3f(50.0, 20.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
glEnd();
```

The specification is for front faces to be outlined and back faces filled, but, as the four triangles are not consistently oriented, we see both outlined and filled triangles (Figure 9.14(a)). **End**

Click for square.cpp Program Windows Project

Experiment 9.3. Continuing the previous experiment, next replace the polygon declaration part of square.cpp with:

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
glBegin(GL_TRIANGLE_STRIP);
   glVertex3f(20.0, 80.0, 0.0);
   glVertex3f(20.0, 20.0, 0.0);
   glVertex3f(50.0, 80.0, 0.0);
   glVertex3f(50.0, 20.0, 0.0);
   glVertex3f(80.0, 80.0, 0.0);
   glVertex3f(80.0, 20.0, 0.0);
glEnd();
```

The resulting triangulation is the same as before, but, as it's consistently oriented, we see only outlined front faces (Figure 9.14(b)). **End**

Click for squareOfWalls.cpp Program Windows Project

Experiment 9.4. Run squareOfWalls.cpp, which shows four rectangular walls enclosing a square space. The front faces (the outside of the walls) are filled, while the back faces (the inside) are outlined. Figure 9.15(a) is a screenshot.

The triangle strip of squareOfWalls.cpp consists of eight triangles which are consistently oriented, because triangles in a strip are *always* consistently oriented. **End**

Click for `threeQuarterSphere.cpp` Program Windows Project

Experiment 9.5. Run `threeQuarterSphere.cpp`, which adds one half of a hemisphere to the bottom of the hemisphere of `hemisphere.cpp`. The two polygon mode calls ask the front faces to be drawn filled and back ones outlined. Turn the object about the axes by pressing 'x', 'X', 'y', 'Y', 'z' and 'Z'.

Unfortunately, the ordering of the vertices is such that the outside of the hemisphere appears filled, while that of the half-hemisphere outlined. Figure 9.15(b) is a screenshot. Likely, this would not be intended in a real design application where one would, typically, expect a consistent look throughout one side.

Such mixing up of orientation is not an uncommon error when assembling an object out of multiple pieces. Fix the problem in the case of `threeQuarterSphere.cpp` in four different ways:

(a) Replace the loop statement

```
for(i = 0; i <= p/2; i++)
```

of the half-hemisphere with

```
for(i = p/2; i >= 0; i--)
```

to reverse its orientation.

(b) Interchange the two `glVertex3f()` statements of the half-hemisphere, again reversing its orientation.

(c) Place the additional polygon mode calls

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
```

before the half-hemisphere so that its back faces are drawn filled.

(d) Call

```
glFrontFace(GL_CCW)
```

before the hemisphere definition and

```
glFrontFace(GL_CW)
```

before the half-hemisphere to change the front-face default to be CW-facing for the latter.

Of the four, either (a) or (b) is to be preferred because they go to the source of the problem and repair the object, rather than hide it with the help of state variables, as do (c) and (d). End

Experiment 9.6. Make a Möbius band as follows.

Take a long and thin strip of paper and draw two equal rows of triangles on one side to make a triangulation of the strip as in the bottom of Figure 9.16. Turn the strip into a Möbius band by pasting the two end edges together after twisting one 180°. The triangles you drew on the strip now make a triangulation of the Möbius band.

Try next to orient the triangles by simply drawing a curved arrow in each, in a manner such that the entire triangulation is consistently oriented. Were you able to?! End

Click for sphereInBox1.cpp Program Windows Project

Experiment 9.7. Run sphereInBox1.cpp of Chapter11, which draws a green ball inside a red box. Press up or down arrow keys to open or close the box. Figure 9.18(a) is a screenshot of the box partly open.

Ignore the statements to do with lighting and material properties for now. See the statements before and following where the sphere is drawn with a glutSolidSphere(). Culling is enabled with a call to glEnable(GL_CULL_FACE) and disabled with glDisable(GL_CULL_FACE). The command glCullFace(*face*) where *face* can be GL_FRONT, GL_BACK or GL_FRONT_AND_BACK – it is, in fact, GL_BACK in the program – is used to specify if front-facing or back-facing or all triangles are to be culled. Therefore, back-facing triangles of the sphere, which evidently are all hidden behind front-facing ones, are indeed culled.

Comment out the glDisable(GL_CULL_FACE) call and open the box. Oops! Some sides of the box have disappeared as you can see in Figure 9.18(b). The reason, of course, is that the state variable GL_CULL_FACE is set when the drawing routine is called the first time so that all back-facing triangles, including those belonging to the box, are eliminated on subsequent calls.

End

Click for sphereInBox1.cpp Program Windows Project

Experiment 9.8. Here's a trick often used in 3D design environments like Maya and Studio Max to open up a closed space. Suppose you've finished designing the exterior of a house and now want to work inside it, e.g., to make interior walls and fittings. A good way to do this is to remove only those exterior walls which obscure your view of the interior and leave the rest; that the obscuring walls are either *all* front-facing or *all* back-facing means a cull will do the trick. Let's see this in action.

Insert the pair of statements

```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```

in the drawing routine of sphereInBox1.cpp just before the first glDrawElements() call. The top and front sides of the box are not drawn, leaving its interior visible. Figure 9.18(c) is a screenshot. **End**

Click for squareOfWallsReflected.cpp Program Windows Project

Experiment 9.9. Run squareOfWallsReflected.cpp, which is squareOfWalls.cpp with an additional block of code before the drawing of the wall, including a glScalef(-1.0, 1.0, 1.0) call, to reflect the scene about the yz-plane:

```
if (isReflected)
{
   ...
   glScalef(-1.0, 1.0, 1.0);
   // glFrontFace(GL_CW);
}
else
{
   ...
   // glFrontFace(GL_CCW);
}

// Draw walls as a single triangle strip.
glBegin(GL_TRIANGLE_STRIP);
...
```

The original walls are as in Figure 9.19(a). Press space to reflect. Keeping in mind that front faces are filled and back faces outlined, it seems that `glScalef(-1.0, 1.0, 1.0)` not only reflects, but turns the square of walls inside out as well, as you can see in Figure 9.19(b). **End**

Part V

Making Things Up

Chapter 10

Modeling in 3D Space

Click for `astroid.cpp` Program Windows Project

Experiment 10.1. Run `astroid.cpp` which modifies `circle.cpp` to implement instead the parametric equations

$$x = \cos^3 t,\ y = \sin^3 t,\ z = 0,\ 0 \le t \le 2\pi$$

for the astroid of Exercise 10.2. Figure 10.9 is a screenshot. End

Click for `cylinder.cpp` Program Windows Project

Experiment 10.2. Run `cylinder.cpp`, which shows a triangular mesh approximation of a circular cylinder, given by the parametric equations

$$x = f(u,v) = \cos u, \quad y = g(u,v) = \sin u, \quad z = h(u,v) = v,$$

for $(u,v) \in [-\pi,\pi] \times [-1,1]$. Pressing arrow keys changes the fineness of the mesh. Press 'x/X', 'y/Y' or 'z/Z' to turn the cylinder itself. Figure 10.28 is a screenshot. End

Click for `helicalPipe.cpp` Program Windows Project

Experiment 10.3. Without really knowing what to expect (honestly!) we tweaked the parametric equations of the cylinder to the following:

$$x = \cos u + \sin v,\ y = \sin u + \cos v,\ z = u, \quad (u,v) \in [-\pi,\pi] \times [-\pi,\pi]$$

It turns out the resulting shape looks like a helical pipe – run `helicalPipe.cpp`. Figure 10.31 is a screenshot.

Functionality is the same as for `cylinder.cpp`: press the arrow keys to coarsen or refine the triangulation and 'x/X', 'y/Y' or 'z/Z' to turn the pipe.

Looking at the equations again, it wasn't too hard to figure out how this particular surface came into being. See the next exercise. End

Click for `torus.cpp` Program Windows Project

Experiment 10.4. Run `torus.cpp`, which applies the parametric equations deduced above within the template of `cylinder.cpp` (simply swapping the new `f`, `g` and `h` function definitions into the latter program). The radii of the circular trajectory and the profile circle are set to 2.0 and 0.5, respectively. Figure 10.35 is a screenshot.

Functionality is the same as for `cylinder.cpp`: press the arrow keys to coarsen or refine the triangulation and 'x/X', 'y/Y' or 'z/Z' to turn the torus. **End**

Click for `torusSweep.cpp` Program Windows Project

Experiment 10.5. Run `torusSweep.cpp`, modified from `torus.cpp` to show the animation of a circle sweeping out a torus. Press space to toggle between animation on and off. Figure 10.36 is a screenshot part way through the animation. **End**

Click for `table.cpp` Program Windows Project

Experiment 10.6. The preceding equations are implemented in `table.cpp`, again using the template of `cylinder.cpp`. Press the arrow keys to coarsen or refine the triangulation and 'x/X', 'y/Y' or 'z/Z' to turn the table. See Figure 10.39 for a screenshot of the table.

Note that the artifacts at the edges of the table arise because sample points may not map exactly to corners $(0, -8), (4, -8), \ldots, (0, 8)$ of the profile drawn in Figure 10.38(a) – which can be avoided by including always t values 0, 4, 5, 8, 22, 31, 32 and 42 in the sample grid. **End**

Click for `doublyCurledCone.cpp` Program Windows Project

Experiment 10.7. The plan above is implemented in `doublyCurledCone.cpp`, again using the template of `cylinder.cpp`, with the value of A set to $\pi/4$ and a to 0.05. Press the arrow keys to coarsen or refine the triangulation and 'x/X', 'y/Y' or 'z/Z' to turn the cone. Figure 10.41 is a screenshot. **End**

Click for `extrudedHelix.cpp` Program Windows Project

Experiment 10.8. Run `extrudedHelix.cpp`, which extrudes a helix, using yet again the template of `cylinder.cpp`. The parametric equations of the extrusion are

$$x = 4cos(10\pi u),\ y = 4sin(10\pi u),\ z = 10\pi u + 4v,\quad 0 \leq u, v \leq 1$$

the constants being chosen to size the object suitably. As the equation for z indicates, the base helix is extruded parallel to the z-axis. Figure 10.42 is a screenshot. **End**

Click for `bilinearPatch.cpp` Program Windows Project

Experiment 10.9. Run `bilinearPatch.cpp`, which implements precisely Equation (10.20). Press the arrow keys to refine or coarsen the wireframe and 'x/X', 'y/Y' or 'z/Z' to turn the patch. Figure 10.47 is a screenshot. **End**

Click for `hyperboloid1sheet.cpp` Program Windows Project

Experiment 10.10. Run `hyperboloid1sheet.cpp`, which draws a triangular mesh approximation of a single-sheeted hyperboloid with the help of the parametrization

$$x = \cos u \sec v,\ y = \sin u \sec v,\ z = \tan v, \quad u \in [-\pi, \pi],\ v \in (-\pi/2, \pi/2)$$

Figure 10.51(a) is a screenshot. In the implementation we restrict v to $[-0.4\pi, 0.4\pi]$ to avoid $\pm\pi/2$ where sec is undefined. **End**

Click for `gluQuadrics.cpp` Program Windows Project

Experiment 10.11. Run `gluQuadrics.cpp` to see all four GLU quadrics. Press the left and right arrow keys to cycle through the quadrics and 'x/X', 'y,Y' and 'z/Z' to turn them. The images in Figure 10.52 were, in fact, generated by this program. **End**

Click for `glutObjects.cpp` Program Windows Project

Experiment 10.12. Run `glutObjects.cpp`, a program we originally saw in Chapter 3. Press the left and right arrow keys to cycle through the various FreeGLUT objects and 'x/X', 'y/Y' and 'z/Z' to turn them. Among other objects you see all five regular polyhedra, both in solid and wireframe. **End**

Click for `tetrahedron.cpp` Program Windows Project

Experiment 10.13. Run `tetrahedron.cpp`. The program draws a wireframe tetrahedron of edge length $2\sqrt{2}$ which can be turned using the 'x/X', 'y/Y' and 'z/Z' keys. Figure 10.56 is a screenshot. **End**

Click for `bezierCurves.cpp` Program Windows Project

Experiment 10.14. Run `bezierCurves.cpp`. Press the up and down arrow keys to select an order between 2 and 6 on the first screen. Press enter to proceed to the next screen where the control points initially lie on a straight line. Press space to select a control point and then the arrow keys to move it. Press delete to start over. Figure 10.65 is a screenshot for order 6. Obviously, the control points always lie on a plane, keeping matters simple for now.

In addition to the black Bézier curve, drawn in light gray is its *control polygon*, the polyline through successive control points. Note how the Bézier curve tries gently, without hard corners, to track its control polygon's shape. **End**

Click for `bezierCurveWithEvalCoord.cpp` Program Windows Project

Experiment 10.15. Run `bezierCurveWithEvalCoord.cpp`, which draws a fixed Bézier curve of order 6. See Figure 10.66 for a screenshot. There is no interaction. **End**

Click for `bezierCurveWithEvalMesh.cpp` Program Windows Project

Experiment 10.16. Run `bezierCurveWithEvalMesh.cpp`. This program is similar to `bezierCurveWithEvalCoord.cpp` except that, instead of calls to `glEvalCoord1f()`, the pair of statements

```
glMapGrid1f(50, 0.0, 1.0);
glEvalMesh1(GL_LINE, 0, 50);
```

are used to draw exactly the same approximating polyline.

The call `glMapGrid1f(`*n*`, `*t1*`, `*t2*`)` specifies an *evenly-spaced* grid of $n+1$ sample points in the parameter interval, starting at $t1$ and ending at $t2$. So, the sample points returned by `glMapGrid1f(50, 0.0, 1.0)` are $0.0, 0.02, 0.04, \ldots, 1.0$. The call `glEvalMesh1(`*mode*`, `*p1*`, `*p2*`)` works in tandem with the `glMapGrid1f(`*n*`, `*t1*`, `*t2*`)` call. For example, if *mode* is `GL_LINE`, then it draws a line strip through the mapped sample points, starting with the image of the $p1$th sample point and ending at the image of the $p2$th one. **End**

Click for `bezierCurveTangent.cpp` Program Windows Project

Experiment 10.17. Run `bezierCurveTangent.cpp`. The blue curve may be shaped by selecting a control point with the space key and moving it with the arrow keys. Check that the two curves meet in a visually smooth way when their control polygons meet smoothly. Figure 10.68 is a screenshot of such a configuration. **End**

Click for `bezierSurface.cpp` Program Windows Project

Experiment 10.18. Run `bezierSurface.cpp`, which allows the user herself to shape a Bézier surface by selecting and moving control points originally in a 6×4 grid. Drawn in black actually is a 20×20 quad mesh approximation of the Bézier surface. Also drawn in light gray is the *control polyhedron*, which is the polyhedral surface with vertices at control points.

Press the space and tab keys to select a control point. Use the left/right arrow keys to move the selected control point parallel to the x-axis, the up/down arrow keys to move it parallel to the y-axis, and the page up/down keys to move it parallel to the z-axis. Press 'x/X', 'y/Y' and 'z/Z' to turn the surface. Figure 10.69 is a screenshot. **End**

Click for `bezierCanoe.cpp` Program Windows Project

Experiment 10.19. Run `bezierCanoe.cpp`. Repeatedly press the right arrow key for a design process that starts with a rectangular Bézier patch, and then edits the control points in each of three successive steps until a canoe is formed. The left arrow reverses the process. Press 'x/X', 'y/Y' and 'z/Z' to turn the surface.

The initial configuration is a 6×4 array of control points placed in a rectangular grid on the xz-plane, making a rectangular Bézier patch.

The successive steps are:

(1) Lift the two end columns of control points up in the y-direction and bring them in along the x-direction to fold the rectangle into a pocket.

(2) Push the middle control points of the end columns outwards along the x-direction to plump the pocket into a "canoe" with front and back still open.

(3) Bring together the two halves of each of the two end rows of control points to stitch closed the erstwhile open front and back. Figure 10.70 is a screenshot after this step.

End

Click for `torpedo.cpp` Program Windows Project

Experiment 10.20. Run `torpedo.cpp`, which shows a torpedo composed of a few different pieces, including bicubic Bézier patch propeller blades:

(i) Body: GLU cylinder.

(ii) Nose: hemisphere.

(iii) Three fins: identical GLU partial discs.

(iv) Backside: GLU disc.

(v) Propeller stem: GLU cylinder.

(vi) Three propeller blades: identical bicubic Bézier patches (control points arranged by trial-and-error).

Press space to start the propellers turning. Press 'x/X', 'y/Y' and 'z/Z' to turn the torpedo. Figure 10.73 is a screenshot. **End**

Click for `OBJmodelViewer.cpp` Program Windows Project

Experiment 10.21. Run `OBJmodelViewer.cpp`. Press 'x'-'Z' to turn the object, currently a gourd. See Figure 10.76. Replace `gourd.obj` with `lamp.obj` as parameter to `loadOBJ()` in `setup()` to see a lamp, or with `rectangle.obj` to see a simple test object.

You can load the other OBJ model files from the site where we obtained the gourd and lamp (see `README.txt` in the program's directory), as well as those you might find in the numerous OBJ model galleries on the web, or make yourself using a 3D design package.

Note, however, that our object-loading routine `loadOBJ()` is barebones, reading, as we'll soon see, only the physical coordinates of an object's vertices, as well as its mesh structure, the latter being displayed by the program. It *ignores* additional vertex attributes such as texture coordinates and normal values so the program as now can neither display texture nor handle lighting. Nevertheless, the program is fairly robust and should draw at least the wireframe for any OBJ model. **End**

Click for `fractals.cpp` Program Windows Project

Experiment 10.22. Run `fractals.cpp`, which draws three different fractal curves – a Koch snowflake, a variant of the Koch snowflake and a tree – all within the framework above, by simply switching source-sequel specs. Press the left/right arrow keys to cycle through the fractals and the up/down arrow keys to change the level of recursion. Figure 10.81 shows all three at level 4. **End**

Part VI

Lights, Camera, Equation

Chapter 11

Color and Light

Click for `sphereInBox1.cpp` Program Windows Project

Experiment 11.1. Run `sphereInBox1.cpp`. Press the up-down arrow keys to open or close the box. Figure 11.19 is a screenshot of the box partly open. We'll use this program as a running example to explain much of the OpenGL lighting and material color syntax. End

Click for `lightAndMaterial1.cpp` Program Windows Project

Experiment 11.2. Run `lightAndMaterial1.cpp`.

The ball's current ambient and diffuse reflectances are identically set to a maximum blue of $\{0.0, 0.0, 1.0, 1.0\}$, its specular reflectance to the highest gray level $\{1.0, 1.0, 1.0, 1.0\}$ (i.e., white), shininess to 50.0 and emission to zero $\{0.0, 0.0, 0.0, 1.0\}$.

Press 'a/A' to decrease/increase the ball's blue **A**mbient reflectance and 'd/D' to change likewise its **D**iffuse reflectance. Pressing 's/S' decreases/increases the gray level of its **S**pecular reflectance. Pressing 'h/H' decreases/increases its s**H**ininess, while pressing 'e/E' decreases/increases the blue component of the ball's **E**mission. The page up and down keys move the ball while 'r' returns it to its original position. End

Click for `lightAndMaterial2.cpp` Program Windows Project

Experiment 11.3. Run `lightAndMaterial2.cpp`.

The white light's current diffuse and specular are identically set to a maximum of $\{1.0, 1.0, 1.0, 1.0\}$ and it gives off zero ambient light. The green light's attributes are fixed at a maximum diffuse and specular of $\{0.0, 1.0, 0.0, 1.0\}$, again with zero ambient. The global ambient currently is a low intensity gray at $\{0.2, 0.2, 0.2, 1.0\}$.

Press 'w' to toggle the **W**hite light off and on and 'g' to toggle likewise the **G**reen light. Press 'd/D' to decrease/increase the gray level of the white light's **D**iffuse and specular intensity (the ambient intensity never changes from zero). Pressing 'm/M' decreases/increases the gray intensity of the global a**M**bient. Move the ball with the page up and down keys. Rotate the white light about the ball's *original* position by pressing the arrow keys (it does not follow the ball if the latter is moved).

The program has added functionality which we'll need later. End

Click for `lightAndMaterial1.cpp` Program Windows Project

Experiment 11.4. Run `lightAndMaterial1.cpp`.

Reduce the specular reflectance of the ball. Both the white and green highlights begin to disappear, as it's the specular components of the reflected lights which appear as specular *highlights* (or, *glint*). See Figure 11.21. **End**

Click for `lightAndMaterial1.cpp` Program Windows Project

Experiment 11.5. Restore the original values of `lightAndMaterial1.cpp`.

Reduce the diffuse reflectance gradually to zero. The ball starts to lose its roundness and brightness until it looks like a flat glassy disc. The reason for this is that ambient intensity, which does not depend on eye or light direction, is uniform across vertices of the ball and cannot, therefore, provide the perception of depth that obtains from a contrast in color values across the surface. Diffuse light, on the other hand, which varies in intensity across the surface depending on how the normal turns, cues the viewer to depth or "3Dness".

Even though there is a specular highlight, sensitive to both eye and light direction, it's too localized to provide much depth contrast. Reducing the shininess does spread the highlight but the effect is not a realistic perception of depth.

Figure 11.24 shows the ball starting with only ambient reflectance, then successively adding in diffuse and specular. **End**

Click for `lightAndMaterial1.cpp` Program Windows Project

Experiment 11.6. Restore the original values of `lightAndMaterial1.cpp`.

Now reduce the ambient reflectance gradually to zero. The ball seems to shrink! See Figure 11.25. This is because the vertex normals rotate away from the light direction (and viewer) near the visible edges of the ball, scaling down the diffuse reflectance there (recall the $\cos\theta$ term in the diffusive reflectance equation (11.7)). The result, with no ambient light to offset the reduction in diffuse, is that the ends of the ball are dark. **End**

Click for `lightAndMaterial1.cpp` Program Windows Project

Experiment 11.7. Restore the original values of `lightAndMaterial1.cpp`.

Reduce both the ambient and diffuse reflectances to nearly zero. See Figure 11.26. It's like the cat disappearing, leaving only its grin! **End**

Click for `lightAndMaterial1.cpp` Program Windows Project

Experiment 11.8. Run `lightAndMaterial1.cpp` with its original values.

With its current high ambient, diffuse and specular reflectances the ball looks a shiny plastic (Figure 11.28(a)). Reducing the ambient, diffuse and specular reflectances somewhat makes for a heavier and less plastic appearance (Figure 11.28(b)). Restoring the ambient and diffuse to higher values, but reducing the specular reflectance makes it a less shiny plastic (Figure 11.28(c)). Low values for all three of ambient, diffuse and specular reflectances give the ball a somewhat wooden appearance (Figure 11.28(d)).

But, of course, the eyes of the beholder have final say, which means that color and light are art as much as science. **End**

Click for `lightAndMaterial2.cpp` Program Windows Project

Experiment 11.9. Run `lightAndMaterial2.cpp`.

Reduce the white light's diffuse and specular intensity to nearly 0. The ball becomes a flat dull blue disc with only the green highlight prominent (Figure 11.29). This is because the ball's diffuse (and ambient) is blue and cannot reflect the green light's diffuse component, causing the ball thereby to lose almost all diffuse light and, consequently, three-dimensionality.

Raising the white global ambient brightens the ball, but it still looks flat in the absence of diffusive light. **End**

Click for `spotlight.cpp` Program Windows Project

Experiment 11.10. Run `spotlight.cpp`. The program is primarily to demonstrate spotlighting, the topic of a forthcoming section. Nevertheless, press the page-up key to see a multi-colored array of spheres. Figure 11.32 is a screenshot.

Currently, the point of interest in the program is the invocation of the color material mode for the front-face ambient and diffuse reflectances by means of the last two statements in the initialization routine, viz.,

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
```

and subsequent coloring of the spheres in the drawing routine by `glColor4f()` statements. **End**

Click for `lightAndMaterial2.cpp` Program Windows Project

Experiment 11.11. Run `lightAndMaterial2.cpp`. Press 'l' or 'L' to toggle between the **L**ocal and the infinite viewpoint in `lightAndMaterial2.cpp`. See Figures 11.34 and 11.34 for screenshots. **End**

Click for `litTriangle.cpp` Program Windows Project

Experiment 11.12. Run `litTriangle.cpp`, which draws a single triangle, whose front face is coded to be red and back blue, initially front-facing and lit two-sided. Press the left and right arrow keys to turn the triangle and space to toggle two-sided lighting on and off. See Figures 11.36 and 11.37 for screenshots.

Notice how the back face is dark when two-sided lighting is disabled – this is because the normals are pointing oppositely of the way they should be. **End**

Click for `lightAndMaterial2.cpp` Program Windows Project

Experiment 11.13. Press 'p' or 'P' to toggle between **P**ositional and directional white light in `lightAndMaterial2.cpp`. The white wire sphere indicates the positional light, while the white arrow the incoming directional light. See Figures 11.39 and 11.40. Keep in mind that you can both move the ball and rotate the light position. **End**

Click for `lightAndMaterial2.cpp` Program Windows Project

Experiment 11.14. Run `lightAndMaterial2.cpp`. The current values of the constant, linear and quadratic attenuation parameters are 1, 0 and 0, respectively, so

there's no attenuation. Press 't/T' to decrease/increase the quadratic a**T**tenuation parameter. Linear attenuation is always zero.

Move the ball by pressing the page up/down keys to observe the effect of attenuation. Figure 11.41 shows the ball some ways off with no attenuation and then with some amount of attenuation in the case of both a positional and a directional light. **End**

Click for `spotlight.cpp` Program Windows Project

Experiment 11.15. Run `spotlight.cpp`, which shows a bright white spotlight illuminating a multi-colored array of spheres. Figure 11.43 is a screenshot.

Press the page up/down arrows to increase/decrease the angle of the light cone. Press the arrow keys to move the spotlight. Press 't/T' to change the spotlight attenuation exponent (to be explained soon). A white wire mesh is drawn separately along the light cone boundary. **End**

Click for `spotlight.cpp` Program Windows Project

Experiment 11.16. Run again `spotlight.cpp`. Observe the darkening of the balls near the cone boundary as the attenuation exponent is increased by pressing 'T'. Figure 11.44 is a screenshot with the attenuation exponent equal to 10.0. Compare this with the Figure 11.43 where the exponent was 2.0. **End**

Click for `checkeredFloor.cpp` Program Windows Project

Experiment 11.17. Run `checkeredFloor.cpp`, which creates a checkered floor drawn as an array of flat shaded triangle strips. See Figure 11.45. Flat shading causes each triangle in a strip to be painted with the color of the last of its three vertices, according to the order of the strip's vertex list.

End

Click for `sphereInBox1.cpp` Program Windows Project

Experiment 11.18. Run again `sphereInBox1.cpp`. The normal vector values at the eight box vertices of `sphereInBox1.cpp`, placed in the array `normals[]`, are

$$[\pm 1/\sqrt{3} \quad \pm 1/\sqrt{3} \quad \pm 1/\sqrt{3}]^T$$

each corresponding to one of the eight possible combinations of signs. **End**

Click for `sphereInBox2.cpp` Program Windows Project

Experiment 11.19. Run `sphereInBox2.cpp`, which modifies `sphereInBox1.cpp`. Press the arrow keys to open or close the box and space to toggle between two methods of drawing normals.

The first method is the same as that of `sphereInBox1.cpp`, specifying the normal at each vertex as an average of incident face normals. The second creates the box by first drawing one side as a square with the normal at each of its four vertices identically specified to be the unit vector perpendicular to that square, then placing that square in a display list and, finally, drawing it six times appropriately rotated. Figure 11.53(b) shows the vertex normals to the three incident faces at each vertex. Figure 11.54 shows screenshots of the box created with and without averaged normals.

End

Click for `litCylinder.cpp` Program Windows Project

Experiment 11.20. Run `litCylinder.cpp`, which builds upon `cylinder.cpp` using the normal data calculated above, together with color and a single directional light source. Press 'x/X', 'y/Y' and 'z/Z' to turn the cylinder. The functionality of being able to change the fineness of the mesh approximation has been dropped. Figure 11.58 is a screenshot. End

Click for `litDoublyCurledCone.cpp` Program Windows Project

Experiment 11.21. The program `litDoublyCurledCone.cpp`, in fact, applies the preceding equations for the normal and its length. Press 'x/X', 'y/Y', 'z/Z' to turn the cone. See Figure 11.60 for a screenshot.

As promised, `litDoublyCurledCone.cpp` is pretty much a copy of `litCylinder.cpp`, except for the new `f()`, `g()`, `h()`, `fn()`, `gn()` and `hn()` functions, as also the new `normn()` to compute the normal's length.

End

Click for `litCylinderProgrammedNormals.cpp` Program Windows Project

Experiment 11.22. Run `litCylinderProgrammedNormals.cpp`. Press 'x/X', 'y/Y', 'z/Z' to turn the cylinder. Figure 11.61 is a screenshot. End

Click for `ballAndTorusLitOrthoShadowed.cpp` Program Windows Project

Experiment 11.23. Run `ballAndTorusLitOrthoShadowed.cpp`. Press space to start the ball traveling around the torus and the up and down arrow keys to change its speed. Figure 11.64 is a screenshot.

The only user-provided normal in the program is the unit vector in the $+y$-direction for the checkered floor; normals are automatic for the ball and torus, FreeGLUT objects both. The shadows are the result of simple degenerate scaling calls, as discussed in Section 4.7.2. End

Click for `litBezierCanoe.cpp` Program Windows Project

Experiment 11.24. Run `litBezierCanoe.cpp`. Press 'x/X', 'y/Y', 'z/Z' to turn the canoe. You can see a screenshot in Figure 11.65.

This program illuminates the final shape of `bezierCanoe.cpp` of Experiment 10.19 with a single directional light source. Other than the expected command `glEnable(GL_AUTO_NORMAL)` in the initialization routine, an important point to notice about `litBezierCanoe.cpp` is the reversal of the sample grid along the u-direction. In particular, compare the statement

```
glMapGrid2f(20, 1.0, 0.0, 20, 0.0, 1.0)
```

of `litBezierCanoe.cpp` with

```
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0)
```

of `bezierCanoe.cpp`. This change reverses the directions of one of the tangent vectors evaluated at each vertex by OpenGL and, consequently, that of the normal (which is the cross-product of the two tangent vectors).

Modify `litBezierCanoe.cpp` by changing

```
glMapGrid2f(20, 1.0, 0.0, 20, 0.0, 1.0);
```

back to `bezierCanoe.cpp`'s

```
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
```

Wrong normal directions! The change from `bezierCanoe.cpp` is necessary. Another solution would be to leave `glMapGrid2f()` as it is in `bezierCanoe.cpp`, instead making a call to `glFrontFace(GL_CW)`. **End**

Click for `shipMovie.cpp` Program Windows Project

Experiment 11.25. Run `shipMovie.cpp`. Pressing space starts an animation sequence which begins with a torpedo traveling toward a moving ship and ends on its own after a few seconds. Press space again at any time to stop the animation. Figure 11.66 is a screenshot as the torpedo nears the ship. **End**

Click for `sizeNormal.cpp` Program Windows Project

Experiment 11.26. Run `sizeNormal.cpp` based on `litTriangle.cpp`.

The ambient and diffuse colors of the three triangle vertices are set to red, green and blue, respectively. The normals are specified separately as well, initially each of unit length perpendicular to the plane of the triangle.

However, pressing the up/down arrow keys changes (as you can see) the size, but not the direction, of the normal at the red vertex. Observe the corresponding change in color of the triangle. Figure 11.68 is a screenshot.

End

Click for `sizeNormal.cpp` modified Program Windows Project

Experiment 11.27. Run `sizeNormal.cpp` after placing the statement `glEnable(GL_NORMALIZE)` at the end of the initialization routine. Press the up/down arrow keys. The triangle no longer changes color (though the white arrow still changes in length, of course, because its size is that of the program-specified normal). **End**

Chapter 12

Texture

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.1. Run `texturedSquare.cpp`, which loads an external image of a shuttle launch as one texture and generates internally a chessboard image as another.

The program paints both the external and the procedural texture onto a square polygon. Figure 12.5 shows the two. Press space to toggle between them, the left and right arrow keys to turn the square and delete to reset.

As one rotates the square one sees the texture image front-facing on the front of the square, while the image is seen from behind on its backside. **End**

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.2. Replace every 1.0 in each `glTexCoord2f()` command of `texturedSquare.cpp` with 0.5 so that the polygon specification is:

```
glBegin(GL_POLYGON);
   glTexCoord2f(0.0, 0.0); glVertex3f(-10.0, -10.0, 0.0);
   glTexCoord2f(0.5, 0.0); glVertex3f(10.0, -10.0, 0.0);
   glTexCoord2f(0.5, 0.5); glVertex3f(10.0, 10.0, 0.0);
   glTexCoord2f(0.0, 0.5); glVertex3f(-10.0, 10.0, 0.0);
glEnd();
```

The lower left quarter of the texture is interpolated over the square (Figure 12.9 for a screenshot and Figure 12.13(a) for the texture map). Make sure to see both the launch and chessboard textures! **End**

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.3. Restore the original `texturedSquare.cpp` and delete the last vertex from the polygon so that the specification is that of a triangle:

```
glBegin(GL_POLYGON);
   glTexCoord2f(0.0, 0.0); glVertex3f(-10.0, -10.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(10.0, -10.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(10.0, 10.0, 0.0);
glEnd();
```

Exactly as expected, the lower-right triangular half of the texture is interpolated over the world-space triangle (Figure 12.10 and Figure 12.13(b)).

Change the coordinates of the last vertex of the world-space triangle to $(0.0, 10.0, 0.0)$:

```
glBegin(GL_POLYGON);
   glTexCoord2f(0.0, 0.0); glVertex3f(-10.0, -10.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(10.0, -10.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 10.0, 0.0);
glEnd();
```

Interpolation is clearly evident now. Parts of both launch and chessboard are skewed by texturing, as the triangle specified by texture coordinates (still the lower-right half) is not similar to its world-space counterpart (Figure 12.11 and Figure 12.13(c)).

Continuing, change the texture coordinates of the last vertex of the triangle to $(0.5, 1.0)$:

```
glBegin(GL_POLYGON);
   glTexCoord2f(0.0, 0.0); glVertex3f(-10.0, -10.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(10.0, -10.0, 0.0);
   glTexCoord2f(0.5, 1.0); glVertex3f(0.0, 10.0, 0.0);
glEnd();
```

The textures are no longer skewed as the triangle in texture space is similar to the one being textured (Figure 12.12 and Figure 12.13(d)). **End**

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.4. Restore the original `texturedSquare.cpp` and replace `launch.bmp` with `cray2.bmp`, an image of a Cray 2 supercomputer.

View the texture images in the `Textures` folder and note their sizes: the launch is 512×512 pixels while the Cray 2 is 512×256. As you can see, the Cray 2 (original texture is Figure 12.14) is now scaled by half width-wise to fit the square polygon (screenshot is Figure 12.15). **End**

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.5. Restore the original `texturedSquare.cpp` and then change the coordinates of only the third world-space vertex of the textured polygon to $(20.0, 0.0, 0.0)$:

```
glBegin(GL_POLYGON);
   glTexCoord2f(0.0, 0.0); glVertex3f(-10.0, -10.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(10.0, -10.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(20.0, 0.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(-10.0, 10.0, 0.0);
glEnd();
```

The launch looks odd. The rocket rises vertically, but the flames underneath are shooting sideways! Toggle to the chessboard and it's instantly clear what's happening Figure 12.16 shows both textures. **End**

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.6. Restore the original `texturedSquare.cpp` and change the texture coordinates of the polygon as follows:

```
glBegin(GL_POLYGON);
   glTexCoord2f(-1.0, 0.0); glVertex3f(-10.0, -10.0, 0.0);
   glTexCoord2f(2.0, 0.0); glVertex3f(10.0, -10.0, 0.0);
   glTexCoord2f(2.0, 2.0); glVertex3f(10.0, 10.0, 0.0);
   glTexCoord2f(-1.0, 2.0); glVertex3f(-10.0, 10.0, 0.0);
glEnd();
```

It seems that the texture space is *tiled* with the texture. See Figure 12.18.

In particular, the texture seems repeated in every unit square of texture space with integer vertex coordinates. As the world-space polygon is mapped to a 3×2 rectangle in texture space, it is painted with six copies of the texture, each scaled to an aspect ratio of 2:3. The scheme itself is indicated Figure 12.19. End

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.7. Change again the texture coordinates of `texturedSquare.cpp` by replacing each -1.0 with -0.5:

```
glBegin(GL_POLYGON);
   glTexCoord2f(-0.5, 0.0); glVertex3f(-10.0, -10.0, 0.0);
   glTexCoord2f(2.0, 0.0); glVertex3f(10.0, -10.0, 0.0);
   glTexCoord2f(2.0, 2.0); glVertex3f(10.0, 10.0, 0.0);
   glTexCoord2f(-0.5, 2.0); glVertex3f(-10.0, 10.0, 0.0);
glEnd();
```

Again it's apparent that the texture space is tiled with the specified texture and that the world-space polygon is painted over with its rectangular image in texture space. See Figure 12.20. End

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.8. Restore the original `texturedSquare.cpp` and then change the texture coordinates as below, which is the same as in Experiment 12.6:

```
glBegin(GL_POLYGON);
   glTexCoord2f(-1.0, 0.0); glVertex3f(-10.0, -10.0, 0.0);
   glTexCoord2f(2.0, 0.0); glVertex3f(10.0, -10.0, 0.0);
   glTexCoord2f(2.0, 2.0); glVertex3f(10.0, 10.0, 0.0);
   glTexCoord2f(-1.0, 2.0); glVertex3f(-10.0, 10.0, 0.0);
glEnd();
```

Next, replace the `GL_REPEAT` parameter in the

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

statement of both the `loadlTextures()` and `loadChessboardTexture()` routines with `GL_CLAMP_TO_EDGE` so that the statement becomes

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

This causes the wrapping mode to be clamped to the edge in the s-direction. It's probably easiest to understand what happens in this mode by observing, in particular, the chessboard texture. See Figure 12.21: points of the square with texture coordinate s-value is greater than 1 (i.e., points in the right one-third of the square) each get their color values from the point on the right edge of the texture with the same t-value. In other words, the right edge of the texture, where $s = 1$, is repeated along every column of the square whose texture s is greater than 1. Likewise, the left edge of the texture, where $s = 0$, is repeated along every column of the square whose texture s is less than 1. End

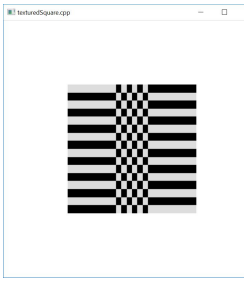


Figure 12.1: Screenshot from Experiment 12.8.

Click for `texturedSquare.cpp` Program Windows Project

Experiment 12.9. Continue the previous experiment by clamping to the edge the texture along the t-direction as well. In particular, replace the `GL_REPEAT` parameter in the

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

statements with GL_CLAMP_TO_EDGE. We leave the reader to parse the output (Figure 12.22). End

Click for fieldAndSky.cpp Program Windows Project

Experiment 12.10. Run fieldAndSky.cpp, where a grass texture is tiled over a horizontal rectangle and a sky texture clamped to a vertical rectangle. There is the added functionality of being able to transport the camera over the field by pressing the up and down arrow keys. Figure 12.23 shows a screenshot.

As the camera travels, the grass seems to *shimmer* – *flash* and *scintillate* are terms also used to describe this phenomenon. This is our first encounter with the *aliasing* problem in texturing. Any visual artifact which arises owing to the finite resolution of the display device, or, equivalently, the "large" size of individual pixels – at least to the extent that they are discernible to the human eye – is said to be caused by aliasing. End

Click for fieldAndSky.cpp Program Windows Project

Experiment 12.11. Change to linear filtering in fieldAndSky.cpp by replacing every GL_NEAREST with GL_LINEAR. The grass still shimmers though less severely. The sky seems okay with either GL_NEAREST or GL_LINEAR. End

Click for fieldAndSkyFiltered.cpp Program Windows Project

Experiment 12.12. Run fieldAndSkyFiltered.cpp, identical to fieldAndSky.cpp except for additional filtering options. Press the up/down arrow keys to move the camera and the left/right ones to cycle through filters for the grass texture. Messages at the top identify the current filters. Screenshots of the weakest and strongest filters applied are in Figure 12.28. End

Click for compareFilters.cpp Program Windows Project

Experiment 12.13. Run compareFilters.cpp, where one sees side-by-side identical images of a shuttle launch bound to a square. Press the up and down arrow keys to move the squares. Press the left arrow key to cycle through filters for the image on the left and the right arrow key to do likewise for the one on the right. Messages at the top say which filters are currently applied. Of course, if one of the four mipmap-based min filters – GL_NEAREST_MIPMAP_NEAREST through GL_LINEAR_MIPMAP_LINEAR – is applied, then the particular mipmap actually chosen by OpenGL depends on the screen space currently occupied by the square. Figure 12.29 is a screenshot of the initial configuration.

Compare, as the squares move, the quality of the textures delivered by the various min filters. For example, apply the GL_NEAREST min filter to the image on the left and GL_LINEAR_MIPMAP_LINEAR to that on the right and press the up key to move both away – the contrast between the shimmer on the left and its absence on the right is dramatic. End

Click for mipmapLevels.cpp Program Windows Project

Experiment 12.14. Run `mipmapLevels.cpp`, where the mipmaps are supplied by the program, rather than generated automatically with `glGenerateMipmap()`. The mipmaps are very simple: just seven differently colored square images, created by the routine `createMipmaps()`, starting with the blue 64×64 `mipmapRes64` down to the black 1×1 `mipmapRes1`. Subsequently, commands of the form

```
glTexImage2D(GL_TEXTURE_2D, level, GL_RGBA, width, height,
             0, GL_RGBA, GL_UNSIGNED_BYTE, image);
```

in the routine `loadMipmaps()` each binds a *width* × *height* mipmap *image* to the current texture object, starting with the highest resolution image (which, of course, would be the original texture image in the case of an imported texture) at *level* 0, and with each successive image of lower resolution having one higher *level* all the way up to 6.

Move the square using the up and down arrow keys. As it grows smaller a change in color indicates a change in the currently applied mipmap. Figure 12.30 is screenshot after the first change. As the min filter setting is `GL_NEAREST_MIPMAP_NEAREST`, a unique color, that of the closest mipmap, is applied to the square at any given time.
End

Click for `texturedTorus.cpp` Program Windows Project

Experiment 12.15. Run `texturedTorus.cpp`, which shows a high-calorie donut made from mapping a sugary texture onto a torus. Figure 12.32 is a screenshot. Press 'x'-'Z' to turn the torus. **End**

Click for `texturedTorpedo.cpp` Program Windows Project

Experiment 12.16. Run `texturedTorpedo.cpp`, which textures parts of the torpedo of `torpedo.cpp` – from Experiment 10.20 – as you can see in the screenshot in Figure 12.34. Press space to start the propeller turning. **End**

Click for `texturedSphere.cpp` Program Windows Project

Experiment 12.17. Run `texturedSphere.cpp`, which applies `earth.bmp` as texture to a sphere, the texture map being Mercator projection. Press 'x'-'Z' to turn the sphere. Figure 12.38 is a screenshot. **End**

Click for `fieldAndSkyTextureAnimated.cpp` Program Windows Project

Experiment 12.18. Run `fieldAndSkyTextureAnimated.cpp`, which animates the sky texture of `fieldAndSky.cpp` by applying translations to the current texture matrix. Press space to toggle between animation on and off. Figure 12.39 is a screenshot.
End

Click for `fieldAndSkyLit.cpp` Program Windows Project

Experiment 12.19. Run `fieldAndSkyLit.cpp`, which lights the scene of `fieldAndSky.cpp` with help of the `GL_MODULATE` option. The light source is directional – imagine the sun – its direction being controlled with the left and right arrow keys, while its brightness can be changed using the up and down arrow keys. A white line

indicates the direction and brightness of the sun. Figure 12.40(a) is an early morning screenshot.

The material colors are all white and the light is a uniform gray, which means essentially that the texture is modulated by the intensity of the light source, which can be controlled by changing its diffuse and specular $(d, d, d, 1)$ value or its direction at angle of θ to the ground. The normal to the horizontal grassy plane is vertically upwards. Strangely, we use the same normal for the sky's vertical plane, because using its "true" value toward the positive z-direction has the unpleasant, but not unexpected, consequence of a sky that doesn't darken together with land. **End**

Click for `litTexturedCylinder.cpp` Program Windows Project

Experiment 12.20. Run `litTexturedCylinder.cpp`, which adds a label texture and a can top texture to `litCylinder.cpp` from the previous chapter. Press 'x'-'Z' to turn the can. Figure 12.40(b) is a screenshot.

Most of the program is routine – the texture coordinate generation is, in fact, a near copy of that in `texturedTorus.cpp` – except for the following lighting model command in `setup()` which we're invoking in a program for the first time:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR)
```

We had briefly encountered this statement as an OpenGL lighting model option in Section 11.4. It causes a modification of OpenGL's `GL_MODULATE` procedure: the specular color components are separated and not multiplied with the corresponding texture color components, as are the ambient and diffuse, but added in after. The result is that specular highlights are preserved rather than blended with the texture.

Turning the can you do see the specular highlights, but if you care to comment out the statement above then they disappear. **End**

Click for `multitexture.cpp` Program Windows Project

Experiment 12.21. Run `multitexture.cpp`, which interpolates between night and day sky texture. Press the left/right arrow keys to transition between night and day. Figure 12.42 shows stages in the transition. **End**

Click for `renderedTexture.cpp` Program Windows Project

Experiment 12.22. Run `renderedTexture.cpp`. Press space to toggle the can label animation on and off and 'x'-'Z' to turn the can. Figure 12.47 is a screenshot. **End**

Chapter 13

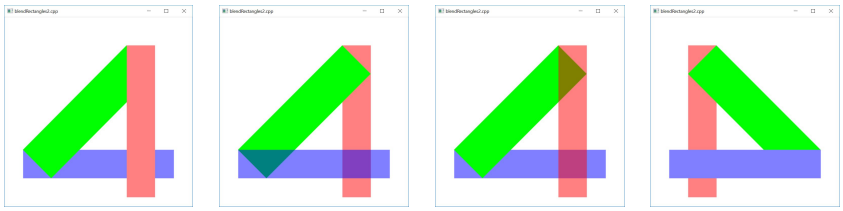
Special Visual Techniques

Click for `blendRectangles1.cpp` Program Windows Project

Experiment 13.1. Run `blendRectangles1.cpp`, which draws two translucent rectangles with their alpha values equal to 0.5 set by calls of the form `glColor4f(*, *, *, 0.5)`, the red one being closer to the viewer than the blue one. The *code* order in which the rectangles are drawn can be toggled by pressing space. Figure 13.2 shows screenshots of either order. End

Click for `blendRectangles2.cpp` Program Windows Project

Experiment 13.2. Run `blendRectangles2.cpp`, which draws three rectangles at different distances from the eye with depth testing enabled. The closest rectangle at $z = 0.5$ is vertical and a translucent red ($\alpha = 0.5$), the next one at $z = 0.3$ is angled and opaque green ($\alpha = 1$), while the farthest at $z = 0.1$ is horizontal and a translucent blue ($\alpha = 0.5$). See Figure 13.3(a).



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

Figure 13.1: Screenshots of `blendRectangles2.cpp`: (a) Initial (b) With depth testing disabled (c) With depth testing re-enabled and rectangles re-ordered to blue, green, red in the code (d) New ordering seen from the $-z$-direction.

The scene is clearly not authentic as no translucency is evident in either of the two areas where the green and blue rectangles are behind the red. The fault is not OpenGL's as it is rendering as it's supposed to with depth testing, as we see next. End

Click for `blendRectangles2.cpp` Program Windows Project

Experiment 13.3. Rearrange the rectangles and insert two `glDepthMask()` calls in the drawing routine of `blendRectangles2.cpp` as follows:

```
// Draw opaque objects, only one.
drawGreenRectangle();

glDepthMask(GL_FALSE); // Make depth buffer read-only.

// Draw translucent objects.
drawBlueRectangle();
drawRedRectangle();

glDepthMask(GL_TRUE); // Make depth buffer writable.
```

Try both `gluLookAt(..., 0.0, 0.0, -1.0, ...)` and `gluLookAt(..., 0.0, 0.0, 1.0, ...)` to see the rectangles from the front and back. Interchange the drawing order of the two translucent rectangles as well. The scene is authentic in every instance. **End**

Click for `sphereInGlassBox.cpp` Program Windows Project

Experiment 13.4. Run `sphereInGlassBox.cpp`, which makes the sides of the box of `sphereInBox2.cpp` glass-like by rendering them translucently. Only the unaveraged normals option of `sphereInBox2.cpp` is implemented. Press the up and down arrow keys to open or close the box and 'x/X', 'y/Y' and 'z/Z' to turn it.

The opaque sphere is drawn first and then the translucent box sides, after making the depth buffer read-only. A screenshot is Figure 13.5(a).

End

Click for `fieldAndSkyTexturesBlended.cpp` Program Windows Project

Experiment 13.5. Run `fieldAndSkyTexturesBlended.cpp`, which is based on `fieldAndSkyLit.cpp` of the preceding chapter. Press the arrow keys to move the sun (actually, the direction of the light).

As the sun rises the night sky morphs into a day sky. Figure 13.5(b) shows late evening. Yes, we saw this very same morph in `multitexture.cpp` as an application of multitexturing, using the interpolation combiner, in Section 12.8. **End**

Click for `ballAndTorusReflected.cpp` Program Windows Project

Experiment 13.6. Run `ballAndTorusReflected.cpp`, which builds on `ballAndTorusLitOrthoShadowed.cpp`. Press space to start the ball traveling around the torus and the up and down arrow keys to change its speed.

The reflected ball and torus are obtained by drawing them scaled by a factor of -1 in the y-direction, which creates their reflections in the xz-plane, and then blending the floor into the reflection. Figure 13.5(c) shows a screenshot. **End**

Click for `ballAndTorusMotionBlurred.cpp` Program Windows Project

Experiment 13.7. Run `ballAndTorusMotionBlurred.cpp`, which enhances the ball of `ballAndTorusLitOrthoShadowed.cpp` with motion blur. Press space to start the ball traveling around the torus and the up and down arrow keys to change its speed. See Figure 13.6 for a screenshot.

Examine the drawing routine to see that the blur, in fact, is simulated by blending four additional copies of the ball behind it with alpha values decreasing with distance from the actual ball. **End**

Click for `ballAndTorusFogged.cpp` Program Windows Project

Experiment 13.8. Run `ballAndTorusFogged.cpp`, which enhances `ballAndTorusLitOrthoShadowed.cpp` from Chapter 11 with a gray fog. Figure 13.7 is a screenshot.

Press the delete key to toggle fog on and off, press space to toggle animation on and off, and the up/down arrows to change the ball's speed. End

Click for `billboard.cpp` Program Windows Project

Experiment 13.9. Run `billboard.cpp`, where an image of two trees is textured onto an upright rectangle in a display list, and then the list called four times to arrange a staggered array of images from left to right. Press the space bar to turn billboarding on and off. See Figure 13.9 for screenshots.

As can be seen, billboarding causes a fairly convincing arc of trees to appear in the distance, while turning it off dispatches the illusion. End

Click for `antiAliasing+Multisampling.cpp` Program Windows Project

Experiment 13.10. Run `antiAliasing+Multisampling.cpp`. Ignore the multisampling controls, as well as the blue-yellow rectangle, for now.

Focus on the red line segment and the green point, which are both either antialiased or not, 'a' or 'A' toggling between the two modes. The width of the line is changed by pressing 'l/L', while the size of the point with 'p/P'. The scene can be turned by the 'x'-'Z' keys and translated by pressing the arrow and page up/down keys.

Figures 13.12(a) and (b) show screenshots of antialiasing off and on, respectively. The effect of antialiasing is especially marked when the line is just shy of horizontal or vertical. End

Click for `antiAliasing+Multisampling.cpp` Program Windows Project

Experiment 13.11. Fire up again `antiAliasing+Multisampling.cpp`. Multisampling is toggled on/off, independently of antialiasing, by pressing 'm' or 'M'. Figures 13.12(c) is a screenshot with multisampling on.

Multisampling antialiases polygons particularly effectively and its effect in our program is best observed on the boundary of the blue-yellow rectangle, as well as the edge between its two colored triangular halves, particularly, when they are nearly horizontal or vertical. When multisampling is enabled, lines and points are antialiased regardless if `GL_POINT_SMOOTH` or `GL_LINE_SMOOTH` has been enabled, which you can see as well.

The number of sample buffers, shown at the top of program's window when multisampling is enabled, should be at least one, or there is effectively no multisampling and you may need to check the settings of your graphics card. End

Click for `pointSprite.cpp` Program Windows Project

Experiment 13.12. Fire up `pointSprite.cpp`. The space bar toggles animation on and off. The particle system of six sprites of fluctuating size, imprinted with the same star texture, spinning in a circle, is simple-minded, though, hopefully, suggestive of possibilities. Figure 13.14 is a screenshot. End

Click for `sphereMapping.cpp` Program Windows Project

Experiment 13.13. Run `sphereMapping.cpp`, which shows the scene of a shuttle launch with a reflective rocket cone initially stationary in the sky above the rocket. Press the up and down arrow keys to move the cone. As the cone flies down, the reflection on its surface of the launch image changes. Figure 13.16 is a screenshot as it's about to crash to the ground. The environment texture applied obviously is `launch.bmp`. **End**

Click for `skybox.cpp` Program Windows Project

Experiment 13.14. Run `skybox.cpp`, which creates a skybox using precisely the cube map of Figure 13.18. A wire ball, the only "real" object in the scene, has been placed inside the box.

The viewer is initially halfway between the middle of the box and the back, looking toward the latter. The up/down arrow keys translate the viewpoint, while the left/right and page up/down keys rotate it. Figure 13.19 is a screenshot.

Note that translations are constrained to keep the viewer far enough always from the background to preserve the illusion that it's real. **End**

Click for `ballAndTorusStenciled.cpp` Program Windows Project

Experiment 13.15. Run `ballAndTorusStenciled.cpp`, based on `ballAndTorusReflected.cpp`. The difference is that in the earlier program the entire checkered floor was reflective, while in the current one the red floor is non-reflective except for a mirror-like disc lying on it. Pressing the arrow keys moves the disc and pressing the space key starts and stops the ball moving. As can be seen in the screenshot of Figure 13.28, the ball and torus are reflected only in the disc and nowhere else. **End**

Click for `imageManipulationFBO.cpp` Program Windows Project

Experiment 13.16. Run `imageManipulationFBO.cpp`. An image of the numeral 1 appears at the bottom left of the OpenGL window. Clicking the mouse left button anywhere on the window will move the image to that location, while you can, as well, drag it with the left button pressed. Figure 13.29 is a screenshot of the initial configuration. **End**

Click for `bumpMapping.cpp` Program Windows Project

Experiment 13.17. Run `bumpMapping.cpp`, where a plane is bump mapped to make it appear corrugated. Press space to toggle between bump mapping turned on and off. Figure 13.33 shows screenshots. **End**

Click for `ballAndTorusShadowMapped.cpp` Program Windows Project

Experiment 13.18. Before running `ballAndTorusShadowMapped.cpp` you may want to run again `ballAndTorusLitOrthoShadowed.cpp` from Chapter 11, seen as well in Experiment 4.37 in Chapter 4, which implements a simple-minded blacken-and-flatten strategy to draw orthographic shadows on the floor cast by a distant overhead light source.

The scenes of the two shadowing programs are nearly identical, except that `ballAndTorusShadowMapped.cpp` shadow maps a local light source, whose position is indicated by a red sphere. Controls are identical too: press space to start the ball

traveling around the torus and the up and down arrow keys to change its speed. As can even be seen in the screenshot of Figure 13.36, shadowing is far more authentic in `ballAndTorusShadowMapped.cpp`. **End**

Part VII

Pixels, Pixels, Everywhere

Chapter 14

Raster Algorithms

Click for `DDA.cpp` Program Windows Project

Experiment 14.1. Run `DDA.cpp`, which is pretty much a word for word implementation of the DDA algorithm above. A point of note is the *simulation* of the raster by the OpenGL window: the statement `gluOrtho2D(0.0, 500.0, 0.0, 500.0)` identifies "pixel-to-pixel" the viewing face with the 500×500 OpenGL window (a "pixel" of the viewing face being a 1×1 square with corners at integer coordinates).

There's no interaction and the endpoints of the line are fixed in the code at $(100, 100)$ and $(300, 200)$. Figure 14.11 is a screenshot. End

Part VIII

Programming Pipe Dreams

Chapter 15

OpenGL 4.3, Shaders and the Programmable Pipeline: Liftoff

Click for `squareShaderized.cpp` Program Windows Project

Experiment 15.1. Fire up the application program `squareShaderized.cpp`, which comes with its two sidekick shaders, the imaginatively named `vertexShader.glsl` and `fragmentShader.glsl`, located in the subfolder `Shaders`. Not only is the functionality of `squareShaderized.cpp` – drawing a black square over white background, see Figure 15.3 – *exactly* that of `square.cpp`, but, as we'll see, so are its internals *modulo* shaders. End

Click for `ballAndTorusShaderized.cpp` Program Windows Project

Experiment 15.2. Run `ballAndTorusShaderized.cpp`. The program's two shaders are `vertexShader.glsl` and `fragmentShader.glsl`. If you care to run `ballAndTorus.cpp` again you see that the functionality of `ballAndTorusShaderized.cpp` is exactly same: space toggles animation on and off, the up and down arrow keys change its speed, and 'x'-'Z' turn the scene. Figure 15.6 is a screenshot. End

Click for `bumpMappingShaderized.cpp` Program Windows Project

Experiment 15.3. Run `bumpMappingShaderized.cpp`. Interaction is the same as `bumpMapping.cpp`: press space to toggle between bump mapping on and off. Figure 15.7(b) is a screenshot with bump mapping enabled, evidently identical to that of `bumpMapping.cpp` in Figure 15.7(a). End

Click for `litCylinderShaderized.cpp` Program Windows Project

Experiment 15.4. Run `litCylinderShaderized.cpp`. Interaction is the same as `litCylinder.cpp`: press the 'x'-'Z' keys to turn the cylinder. Figure 15.8 is a screenshot. End

Click for `bumpMappingPerPixelLight.cpp` Program Windows Project

Experiment 15.5. Run `bumpMappingPerPixelLight.cpp`. Again, press space to toggle between bump mapping on and off. Figure 15.7(c) is a screenshot. The program, its associated C++ source and header files are all *exactly* same as for `bumpMappingShaderized.cpp` – the difference is only in the shaders! **End**

Click for `fieldAndSkyFilteredShaderized.cpp` Program Windows Project

Experiment 15.6. Run `fieldAndSkyFilteredShaderized.cpp`. As in `fieldAndSkyFiltered.cpp` press the up and down arrow keys to move the viewpoint. However, unlike the earlier program, `fieldAndSkyFilteredShaderized.cpp` implements (to keep it simple) only one fixed filter for the grass texture and no options. Figure 15.9 is a screenshot. **End**

Click for `texturedTorusShaderized.cpp` Program Windows Project

Experiment 15.7. Run `texturedTorusShaderized.cpp`. As in `texturedTorus.cpp` press 'x'-'Z' to turn the torus. See Figure 15.10 for a screenshot.

The point to note in `textureTorusShaderized.cpp` is how the associated source `torus.cpp` defines texture coordinates for the torus following exactly `texturedTorus.cpp`. Beyond that the application program and shaders should be easily understood. **End**

Click for `litTexturedCylinderShaderized.cpp` Program Windows Project

Experiment 15.8. Run `litTexturedCylinderShaderized.cpp`. The same beer can is drawn twice, per-vertex lit on the left and per-pixel lit on the right. Press 'x'-'Z' to turn the two cans simultaneously to able to compare per-vertex and per-vertex lighting on the same disposition of the can. Figure 15.11 is a screenshot. We discuss the program below. **End**

Click for `proceduralTexture.cpp` Program Windows Project

Experiment 15.9. Run `proceduralTexture.cpp`. Press 'x'-'Z' to turn the beer can and space to toggle between a procedurally textured whole can and one with a grid of rectangular holes. Screenshots are in Figure 15.13. We'll discuss the program's working next. **End**

Click for `specularMapping.cpp` Program Windows Project

Experiment 15.10. Run `specularMapping.cpp`. Press 'x'-'Z' to turn the beer can and space to toggle between specular mapping off and on. The program is obviously based on `litTexturedCylinderShaderized.cpp`, in particular, keeping only per-pixel lighting from the earlier program.

Screenshots are in Figure 15.16, specular mapping being off on the left and on to the right. Evidently, specular mapping indeed restricts the highlight to the black part of the label. We see how next. **End**

Click for `normalMapping.cpp` Program Windows Project

Experiment 15.11. Run `normalMapping.cpp`. Press space to toggle between applying a normal map to a plane and displaying the normal map as an image. Figure 15.17 shows both. **End**

Chapter 16

OpenGL 4.3, Shaders and the Programmable Pipeline: Escape Velocity

Click for `helixListShaderizedInstancedVertAttrib.cpp` Program Windows Project

Experiment 16.1. Run `helixListShaderizedInstancedVertAttrib.cpp`. The output of six different helixes is exactly the same as that of `helixList.cpp`. See Figure 16.1. End

Click for `helixListShaderizedShaderCounter.cpp` Program Windows Project

Experiment 16.2. Run `helixListShaderizedShaderCounter.cpp`. The output, just like that of `helixListShaderizedInstancedVertAttrib.cpp`, is the same as that of `helixList.cpp`. See Figure 16.2. End

Click for `litTexturedCylinderClipped.cpp` Program Windows Project

Experiment 16.3. Run `litTexturedCylinderClipped.cpp`. Pressing 'x'-'Z' turns the can while space toggles between enabling and disabling a clipping plane which slices off the can top. Figure 16.3 is a screenshot with the top gone. End

Click for `specularMapping.cpp` Program Windows Project

Experiment 16.4. Run `specularMapping.cpp`. Pressing space toggles between specular mapping off and on, the label at the top left indicating the current status. Figure 16.4 is a screenshot with specular mapping on.

We'll leave the reader to check that we have created (using a drawing program) two textures, one saying "Specular mapping on!" and the other "Specular mapping off!", and paste one or the other onto a long thin rectangle in the scene. A tedious process this certainly but, then, OpenGL was never really meant to provide a text-drawing service. End

Click for `ballAndTorusShaderSubroutines.cpp` Program Windows Project

Experiment 16.5. Run `ballAndTorusShaderSubroutines.cpp`. The controls are exactly as for `ballAndTorusShaderized.cpp`: space to toggle animation on and off, up/down arrows to change its speed, and 'x'-'Z' to rotate the scene. Figure 16.5 is a screenshot. **End**

Click for `ballAndTorusTwoProgramObjects.cpp` Program Windows Project

Experiment 16.6. Run `ballAndTorusTwoProgramObjects.cpp`. The controls are same as for `ballAndTorusShaderized.cpp`: space to toggle animation on and off, up/down arrows to change its speed, and 'x'-'Z' to rotate the scene. Figure 16.6 is a screenshot. **End**

Click for `fieldAndSkyTexturesBlendedShaderized.cpp` Program Windows Project

Experiment 16.7. Run `fieldAndSkyTexturesBlendedShaderized.cpp`. As in `fieldAndSkyTexturesBlended.cpp` press the arrow keys to change the direction of the sun, the transition between day and night happening from a blending of day and night textures. See Figure 16.7 for a screenshot. **End**

Click for `launchCameraBlurredcpp.cpp` Program Windows Project

Experiment 16.8. Run `launchCameraBlurredcpp.cpp` to see an image of a shuttle launch. Press space to blur the image as though the camera jolted rightward and space again to unblur it. Figure 16.8 has screenshots. **End**

Click for `points.cpp` Program Windows Project

Experiment 16.9. Run `points.cpp`. Drawn are three large points. Press the right and left arrow keys to cycle between four different point renderings and the up and down arrow keys to move the points parallel to the z-axis. Figure 16.9(a) is a screenshot of what is seen at first, while Figures 16.9(b)-(d) show the other three cases. **End**

Click for `ballAndTorusPickingShaderized.cpp` Program Windows Project

Experiment 16.10. Run `ballAndTorusPickingShaderized.cpp`. The controls are exactly as for `ballAndTorusPicking.cpp`: space to toggle animation on and off, up/down arrows to change its speed, 'x'-'Z' to rotate the scene, and, most importantly, left mouse click to pick either ball or torus and make it blush. See Figure 16.10 for a screenshot. **End**

Click for `ballsAndTorusTransformFeedback.cpp` Program Windows Project

Experiment 16.11. Run `ballsAndTorusTransformFeedback.cpp`. The controls are exactly as for `ballAndTorusShaderized.cpp`: space to toggle animation on and off, up/down arrows to change its speed, and 'x'-'Z' to rotate the scene.

However, instead of one ball, now there are two, initially coincident, which travel in opposite directions around the torus. When the balls intersect they are red, when they are close (closer than a particular threshold distance) they turn orange, and beyond that they are blue. Figure 16.13 is a screenshot when the balls are close. **End**

Click for `tessellatedCurve.cpp` Program Windows Project

Experiment 16.12. Run `tessellatedCurve.cpp`. You see the three input vertices and an initially 5-segment Bézier curve polyline. Press the up/down arrow keys to increase/decrease the number of segments. Figure 16.17 is a screenshot with the initial five segments.

The Bézier curve is evidently a poor approximation to the circular arc (not drawn) through the three input vertices because of the rather naive setting of control points in (16.2), but this is irrelevant to our understanding of the code which is discussed through the next few sections.

Incidentally, it's interesting of itself to see how the program goes about drawing the three input vertices and the Bézier curve together. Its initialization routine actually attaches only the vertex and fragment shaders. The drawing routine, then, first attaches the TCS and TES to draw the Bézier polyline, and, next, detaches both to draw the points – of course, one does not want any tessellation happening when drawing just points. End

Click for `tessellatedSphere.cpp` Program Windows Project

Experiment 16.13. Run `tessellatedSphere.cpp`. Press space to toggle between filled and wireframe and the up and down arrow keys to move the sphere. The tessellation primitive is `quads` and the tessellation levels – all six including the two inner and four outer – are always equal, starting at 50 and decreasing down a staircase function to a minimum of 10 as the sphere moves away. However, pressing 'm' makes all the tessellation levels equal to 50, regardless of how far the sphere has moved; pressing 'm' again restores distance-based tessellation.

Figure 16.24 shows screenshots, all wireframe. The one on the left is a wireframe of the sphere close to the eye; the other two are of the sphere moved the same distance away, the one in the middle being tessellated according to the distance, while the one on the right is maximally tessellated, all levels equal to 50.

The rightmost one is clearly over-tessellated, looking filled rather than wireframe. In fact, the reader can view the filled version of the middle sphere by pressing space to see that it is perfectly smooth, proving that its sparser tessellation is adequate. Excess tessellation is inefficient because it sends redundant triangles down the pipeline and, moreover, tends to introduce aliasing artifacts when the object is moved. End

Click for `torusSilhouette.cpp` Program Windows Project

Experiment 16.14. Run `torusSilhouette.cpp`. Press the space bar to toggle between the silhouette and mesh of a torus. Press 'x'-'Z' to turn the torus. Figure 16.30 is a screenshot of the torus in silhouette. Note the imperfections in the silhouette at certain alignments arising owing to aliasing, as well as floating point round-off errors, which can be fixed with added effort, but this is not our concern here. End

Click for `particleSystem.cpp` Program Windows Project

Experiment 16.15. Run `particleSystem.cpp`. Press space to step through the animation of a decidedly bland particle system. Figure 16.32 is a screenshot a few steps into the animation. We explain below how the program works. End

Part IX

Anatomy of Curves and Surfaces

Chapter 17

Bézier

Click for `deCasteljau3.cpp` Program Windows Project

Experiment 17.1. Run `deCasteljau3.cpp`, which shows an animation of de Casteljau's method for three control points. Press the left or right arrow keys to decrease or increase the curve parameter u. The interpolating points $a(u)$, $b(u)$ and $c(u)$ are colored red, green and blue, respectively. Figure 17.4 is a screenshot. End

Click for `bezierCurves.cpp` Program Windows Project

Experiment 17.2. Run `bezierCurves.cpp` of Chapter 10, which allows the user to choose a Bézier curve of order 2-6 and move each control point.

You can choose an order in the first screen by pressing the up and down arrow keys. Select 3. Press enter to go to the next screen to find the control points initially on a straight line. Press space to select a control point – the selected one is red – and then arrow keys to move it. Delete resets to the first screen. Figure 17.5 is a screenshot.

The polygonal line joining the control points, called the *control polygon* of the curve, is drawn in light gray. Evidently, the Bézier curve "mimics" its control polygon, but smoothly, avoiding a corner. End

Click for `bezierCurveTangent.cpp` Program Windows Project

Experiment 17.3. Run `bezierCurveTangent.cpp` of Chapter 10 which shows two cubic Bézier curves. The second curve may be shaped by selecting a control point with the space bar and moving it with the arrow keys. Visually verify Proposition 17.1(g) – the configuration of the screenshot of Figure 17.10, in fact, does this. End

Click for `sweepBezierSurface.cpp` Program Windows Project

Experiment 17.4. Run `sweepBezierSurface.cpp` to see an animation of the procedure. Press the left/right (or up/down) arrow keys to move the sweeping curve and the space bar to toggle between the two possible sweep directions. Figure 17.13 is a screenshot.

The 4×4 array of the Bézier surface's control points (drawn as small squares) consists of a blue, red, green and yellow row of four control points each. The four fixed Bézier curves of order 4 are drawn blue, red, green and yellow, respectively (the curves are in 3-space, which is a bit hard to make out because of the projection). The sweeping Bézier curve is black and its (moving) control points are drawn as larger

squares. The currently swept part of the Bézier surface is the dark mesh. The current parameter value is shown at the top left. **End**

Click for `bezierSurface.cpp` Program Windows Project

Experiment 17.5. Run `bezierSurface.cpp` from Chapter 10, which allows the user to shape a Bézier surface by selecting and moving control points. Press the space and tab keys to select a control point. Use the left/right arrow keys to move the control point parallel to the x-axis, the up/down arrow keys to move it parallel to the y-axis, and the page up/down keys to move it parallel to the z-axis.

Press 'x/X', 'y/Y' and 'z/Z' to turn the viewpoint. Figure 17.14 is a screenshot. **End**

Chapter 18

B-Spline

Click for `bSplines.cpp` Program Windows Project

Experiment 18.1. Run `bSplines.cpp`, which shows the non-zero parts of the spline functions from first order to cubic over the uniformly spaced knot vector

$$[0, 1, 2, 3, 4, 5, 6, 7, 8]$$

Press the up/down arrow keys to choose the order. Figure 18.9 is a screenshot of the first order. The knot values can be changed as well, but there's no need to now. End

Click for `bSplines.cpp` Program Windows Project

Experiment 18.2. Run again `bSplines.cpp` and select the linear B-splines over the knot vector

$$[0, 1, 2, 3, 4, 5, 6, 7, 8]$$

Figure 18.13 is a screenshot. End

Click for `bSplines.cpp` Program Windows Project

Experiment 18.3. Run again `bSplines.cpp` and select the quadratic B-splines over the knot vector

$$[0, 1, 2, 3, 4, 5, 6, 7, 8]$$

Figure 18.18 is a screenshot. Note the joints indicated as black points. End

Click for `quadraticSplineCurve.cpp` Program Windows Project

Experiment 18.4. Run `quadraticSplineCurve.cpp`, which shows the quadratic spline approximation of nine control points in 2D space over a uniformly spaced vector of 12 knots. Figure 18.21 is a screenshot.

The control points are green. Press the space bar to select a control point – the selected one turns red – and the arrow keys to move it. The knots are the green points on the black bars at the bottom. At this stage there is no need to change their values. The blue points are the joints of the curve, i.e., images of the knots. Also drawn in light gray is the control polygon.

Ignore the code itself for now. We'll be seeing how to draw spline curves and surfaces using OpenGL soon. End

Click for `bSplines.cpp` Program Windows Project

Experiment 18.5. Run `bSplines.cpp` and change the order to see a sequence of cubic B-splines. **End**

Click for `cubicSplineCurve1.cpp` Program Windows Project

Experiment 18.6. Run `cubicSplineCurve1.cpp`, which shows the cubic spline approximation of nine control points in 2D space over a uniformly-spaced vector of 13 knots. The program's functionality is similar to that of `quadraticSplineCurve.cpp`. See Figure 18.23 for a screenshot.

The control points are green. Press the space bar to select a control point – the selected one is colored red – then the arrow keys to move it. The knots are the green points on the black bars at the bottom. The blue points are the joints of the curve. The control polygon is a light gray. **End**

Click for `bSplines.cpp` Program Windows Project

Experiment 18.7. Run again `bSplines.cpp`. Change the knot values by selecting one with the space bar and then pressing the left/right arrow keys. Press delete to reset knot values. Note that the routine `Bspline()` implements the CdM formula (and its convention for 0 denominators).

In particular, observe the quadratic and cubic spline functions. Note how they lose their symmetry about a vertical axis through the center, and that no longer are they translates of one another.

Play around with making knot values equal – we'll soon be discussing the utility of multiple knots. Figures 18.27(a) and (b) are screenshots of the quadratic and cubic functions, respectively, both over the same non-uniform knot vector with a triple knot at the right end. **End**

Click for `quadraticSplineCurve.cpp` Program Windows Project

Experiment 18.8. Run again `quadraticSplineCurve.cpp`. Press 'k' to enter knots mode and alter knot values using the left/right arrow keys and 'c' to return to control points mode. Press delete in either mode to reset.

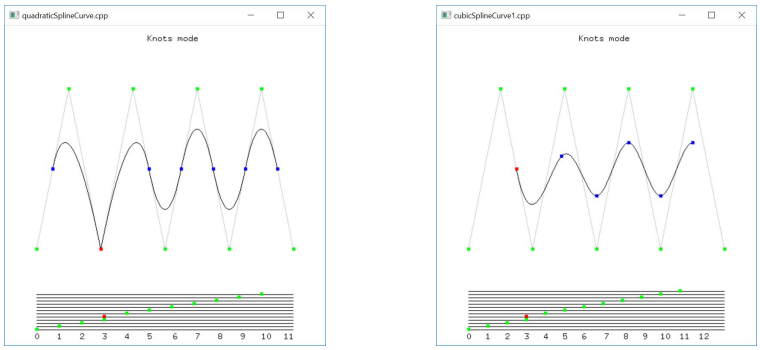
Try to understand what happens if knots are repeated. Do you notice a loss of C^1-continuity when knots in the interior of the knot vector coincide? What if knots at the ends coincide? Figure 18.28 is a screenshot of `quadraticSplineCurve.cpp` with a double knot at 5 and a triple at the end at 11. **End**

Click for `quadraticSplineCurve.cpp` Program Windows Project
Click for `cubicSplineCurve1.cpp` Program Windows Project

Experiment 18.9. Use the programs `quadraticSplineCurve.cpp` and `cubicSplineCurve1.cpp` to make the quadratic and cubic B-spline approximations over the knot vector $T = \{0, 1, 2, 3, 3, 4, 5, 6, 7, \ldots\}$ of nine control points placed as in Figure 18.31(a) (or (b)). See Figure 18.32(a) and (b) for screenshots of the quadratic and cubic curves, respectively.

The quadratic approximation loses C^1-continuity precisely at the control point P_2, which it now *interpolates* as the curve point $c(3)$. It's still C^0 everywhere.

It's not easy to discern visually, but the cubic spline drops from C^2 to C^1-continuous at $c(3)$. **End**



(a) (b)

Figure 18.1: Screenshots of (a) `quadraticSplineCurve.cpp` and (b) `cubicSplineCurve1.cpp` over the knot vector $T = \{0, 1, 2, 3, 3, 4, 5, 6, 7, \ldots\}$ and approximating nine control points arranged in two horizontal rows.

Click for `cubicSplineCurve1.cpp` Program Windows Project

Experiment 18.10. Continuing with `cubicSplineCurve1.cpp` with control points as in the preceding experiment, press delete to reset and then make equal t_4, t_5 and t_6, creating a triple knot at 4. Figure 18.33 is a screenshot of this configuration. Evidently, the control point P_3 is now interpolated at the cost of a drop in continuity there to mere C^0. Elsewhere, the curve is still C^2. End

Click for `quadraticSplineCurve.cpp` Program Windows Project
Program Windows Project

Experiment 18.11. Make the first three and last three knots separately equal in `quadraticSplineCurve.cpp` (Figure 18.34(a)). Make the first four and last four knots separately equal in `cubicSplineCurve1.cpp` (Figure 18.34(b)). Are the first and last control points interpolated in both. *Yes*. Do you notice any impairment in continuity? *No*. End

Click for `quadraticSplineCurve.cpp` Program Windows Project

Experiment 18.12. Change the last parameter of the statement

```
gluNurbsProperty(nurbsObject, GLU_SAMPLING_TOLERANCE, 10.0);
```

in the initialization routine of `quadraticSplineCurve.cpp` from 10.0 to 100.0. The fall in resolution is noticeable as one sees in Figure 18.36. End

Click for `cubicSplineCurve2.cpp` Program Windows Project

Experiment 18.13. Run `cubicSplineCurve2.cpp`, which draws the cubic spline approximation of 30 movable control points, initially laid out on a circle, over a fixed standard knot vector. Press space and backspace to cycle through the control points and the arrow keys to move the selected control point. The delete key resets the control points. Figure 18.37 is a screenshot of the initial configuration. The number of control points being much larger than the order, the user has good local control.

Incidentally, note how managing large numbers of control points has been made efficient with B-splines. Together 30 control points would have led to a 29th degree polynomial Bézier curve, a computational nightmare; alternatively we could split the control points into smaller sets, e.g., of size 4 for cubic curves, but then would come the issue of smoothly joining the successive sub-curves. End

Click for bicubicSplineSurface.cpp Program Windows Project

Experiment 18.14. Run bicubicSplineSurface.cpp, which draws a spline surface approximation to a 15×10 array of control points, each of which the user can move in 3-space. The spline is cubic in both parameter directions and a standard knot vector is specified in each as well.

Press the space, backspace, tab and enter keys to select a control point. Move the selected control point using the arrow and page up and down keys. The delete key resets the control points. Press 'x/X', 'y/Y' and 'z/Z' to turn the surface. Figure 18.39 is a screenshot. **End**

Click for bicubicSplineSurfaceLitTextured.cpp Program Windows Project

Experiment 18.15. Run bicubicSplineSurfaceLitTextured.cpp, which sugar-coats the spline surface of bicubicSplineSurface.cpp. Figure 18.40 is a screenshot. The surface is illuminated by a single positional light source whose location is indicated by a large black point. User interaction remains as in bicubicSplineSurface.cpp. Note that pressing the 'x'-'Z' keys turns only the surface, not the light source.

The bicubic B-spline surface, as well as the fake bilinear one in texture space, are created by the following statements in the drawing routine:

```
gluBeginSurface(nurbsObject);
gluNurbsSurface(nurbsObject, 19, uknots, 14, vknots,
       30, 3, controlPoints[0][0], 4, 4, GL_MAP2_VERTEX_3);
gluNurbsSurface(nurbsObject, 4, uTextureknots, 4, vTextureknots,
       4, 2, texturePoints[0][0], 2, 2, GL_MAP2_TEXTURE_COORD_2);
gluEndSurface(nurbsObject);
```

We'll leave the reader to parse in particular the third statement and verify that it creates a "pseudo-surface" – a 10×10 rectangle – in texture space on the same parameter domain $[0, 12] \times [0, 7]$ as the real one. **End**

Click for bicubicBsplineSurfaceTrimmed.cpp Program Windows Project

Experiment 18.16. Run bicubicBsplineSurfaceTrimmed.cpp, which shows the surface of bicubicBsplineSurface.cpp trimmed by multiple loops. The code is modified from the latter program, functionality remaining same. Figure 18.42(a) is a screenshot. **End**

Chapter 19

Hermite

Click for `hermiteCubic.cpp` Program Windows Project

Experiment 19.1. Run `hermiteCubic.cpp`, which implements Equation (19.10) to draw a Hermite cubic on a plane. Press space to select either a control point or tangent vector and the arrow keys to change it. Figure 19.4 is a screenshot. The actual cubic is simple to draw, but as you can see in the program we invested many lines of code to get the arrow heads right! **End**

Part X

Well Projected

Chapter 20

Applications of Projective Spaces: Projection Transformations and Rational Curves

Click for `manipulateProjectionMatrix.cpp` Program Windows Project

Experiment 20.1. Run `manipulateProjectionMatrix.cpp`, a simple modification of `manipulateModelviewMatrix.cpp` of Chapter 5. Figure 20.4 is a screenshot, though the output to the OpenGL window is of little interest. Of interest, though, are the new statements in the `resize()` routine that output the current projection matrix just before and after the call `glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0)`.

Compare the second matrix output to the command window by the program with P(`glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0)`) as computed with the help of Equation (20.4). End

Click for `rationalBezierCurve1.cpp` Program Windows Project

Experiment 20.2. Run `rationalBezierCurve1.cpp`, which draws the cubic rational Bézier curve specified by four control points on the plane at *fixed* locations, but with *changeable* weights.

The control points on the plane (light gray triangular mesh) are all red, except for the currently selected one, which is black. Press space to cycle through the control points. The control point weights are shown at the upper-left, that of the currently selected one being changed by pressing the up/down arrow keys. The rational Bézier curve on the plane is red as well. Figure 20.6 is a screenshot.

Drawn in green are all the lifted control points, except for that of the currently selected control point, which is black. The projective polynomial Bézier curve approximating the lifted control points is green too. The lifted control points are a larger size as well.

Note: The lifted control points and the projective Bézier curve are primitives in $\mathbb{P}^2$, of course, but represented in $\mathbb{R}^3$ using their homogeneous coordinates.

Also drawn is a cone of several gray lines through the projective Bézier curve which intersects the plane in its projection, the rational Bézier curve.

Observe that increasing the weight of a control point pulls the (red rational Bézier) curve toward it, while reducing it has the opposite effect. Moreover, the end control points are always interpolated regardless of assigned weights. It's sometimes hard to

discern the very gradual change in the shape of the curve as one varies the weights. A trick is to press delete for the curve to spring back to its original configuration, at which moment the difference should be clear. It seems, then, that control point weights are indeed an additional set of "dials" at the designer's disposal to fine-tune a rational Bézier curve.

The code of `rationalBezierCurve1.cpp` is instructive as well, as we'll see in the next section on drawing. **End**

Click for `rationalBezierCurve2.cpp` Program Windows Project

Experiment 20.3. Run `rationalBezierCurve2.cpp`, which draws a red quadratic rational Bézier curve on the plane specified by the three control points $[1, 0]^T$, $[1, 1]^T$ and $[0, 1]^T$. See Figure 20.7. Also drawn is the unit circle centered at the origin. Press the up/down arrow keys to change the weight of the middle control point $[1, 1]^T$. The weights of the two end control points are fixed at 1.

Decrease the weight of the control point $[1, 1]^T$ from its initial value of 1.5. It seems that at some value between 0.70 and 0.71 the curve lies exactly along a quarter of the circle (the screenshot of Figure 20.7 is at 1.13). This is no accident, as the following exercise shows. **End**

Click for `rationalBezierCurve3.cpp` Program Windows Project

Experiment 20.4. Run `rationalBezierCurve3.cpp`, which shows a rational Bézier curve on the plane specified by six control points. See Figure 20.8 for a screenshot. A control point is selected by pressing the space key, moved with the arrow keys and its weight changed by the page up/down keys. Pressing delete resets. **End**

Click for `turnFilmBezier.cpp` Program Windows Project

Experiment 20.5. Run `turnFilmBezier.cpp`, which animates the snapshot transformation of a polynomial Bézier curve described above. Three control points and their magenta approximating polynomial Bézier curve are initially drawn on the $z = 1$ plane. The locations of the control points, and so of their approximating curve as well, are *fixed* in world space and never change through the program.

Initially, the film lies along the $z = 1$ plane. Pressing the right arrow key rotates it toward the $x = 1$ plane, while pressing the left arrow key rotates it back. The film itself, of course, is never seen. As the film turns, the control points and the magenta curve *appear* to move because what is drawn actually is their *projection* (snapshot transformations, particularly) onto the film at its current position. Also drawn on the film is a blue curve, which is the polynomial Bézier curve approximating the current projections of the control points.

Note: The control points and their approximating curve, all fixed on the $z = 1$ plane, corresponding to p_0, p_1 and p_2 and the red curve in Figure 20.9, are *not* drawn by the program – only their snapshot transformations on the turning film.

Initially, when the plane of the film coincides with that on which the control points are drawn, viz., $z = 1$, the projection onto the film of the polynomial Bézier curve approximating the control points (the magenta curve) coincides with the polynomial Bézier curve approximating the projected control points (the blue curve). This is to be expected because the control points coincide with their projections. See Figure 20.10(a) for the initial configuration where the blue curve actually overwrites the magenta one as it comes later in code.

However, as the film turns away from the $z = 1$ plane, the magenta and blue curves begin to separate. Their final configuration, when the film lies along $x = 1$,

is shown in Figure 20.10(b). There is more functionality to the program that we'll discuss momentarily. **End**

Click for `turnFilmBezier.cpp` Program Windows Project

Experiment 20.6. Fire up `turnFilmBezier.cpp` once again. Pressing space at any time draws, instead of the blue curve, the green *rational* Bézier curve approximating the projected control points on the current plane of the film. The control point weights of the green curve are computed according to the strategy just described.

And, one sees: the green rational curve and the magenta projected curve are inseparable, though only the green one is visible because it overwrites the magenta. **End**

Click for `rationalBezierSurface.cpp` Program Windows Project

Experiment 20.7. Run `rationalBezierSurface.cpp`, based on `bezierSurface.cpp`, which draws a rational Bézier surface with the functionality that the location and weight of each control point can be changed. Press the space and tab keys to select a control point. Use the arrow and page up/down keys to translate the selected control point. Press '</>' to change its weight. Press delete to reset. The 'x/X', 'y/Y' and 'z/Z' keys turn the viewpoint. Figure 20.12 is a screenshot.

Mark the use of `glMap2f(GL_MAP2_VERTEX_4, ...)`, as also of `glEnable(GL_MAP2_VERTEX_4)`. The 2's in the syntax are for a surface. **End**

Part XI

Time for a Pipe

Chapter 21

Pipeline Operation

Click for `box.cpp` modified Program Windows Project

Experiment 21.1. Replace the box `glutWireCube(5.0)` of `box.cpp` of Chapter 4 with the line segment

```
glBegin(GL_LINES);
   glVertex3f(1.0, 0.0, -10.0);
   glVertex3f(1.0, 0.0, 0.0);
glEnd();
```

and delete the `glTranslatef(0.0, 0.0, -15.0)` statement.

You see a short segment, the clipped part of the defined line segment, whose first endpoint $[1\ 0\ -10]^T$ is inside the viewing frustum defined by the program's projection statement `glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0)`, while the second $[1\ 0\ 0]^T$ is outside (as is easily checked). Figure 21.3 is a screenshot. **End**

Click for `perspectiveCorrection.cpp` Program Windows Project

Experiment 21.2. Run `perspectiveCorrection.cpp`. You see a thick straight line segment which starts at a red vertex at its left and ends at a green one at its right. Also seen is a big point just above the line, which can be slid along it by pressing the left/right arrow keys. The point's color can be changed, as well, between red and green by pressing the up/down arrow keys. Figure 21.5 is a screenshot.

The color-tuple of the segment's left vertex, as you can verify in the code, is $(1.0, 0.0, 0.0)$, a pure red, while that of the right is $(0.0, 1.0, 0.0)$, a pure green. As expected by interpolation, therefore, there is a color transition from red at the left end of the segment to green at its right.

The number at the topmost right of the display indicates the fraction of the way the big movable point is from the left vertex of the segment to the right. The number below it indicates the fraction of the "way" its color is from red to green – precisely, if the value is u then the color of the point is $(1 - u, u, 0)$.

Initially, the point is at the left and a pure red; in other words, it is 0 distance from the left end, and its color 0 distance from red. Change both values to 0.5 – the color of the point does *not* match that of the segment below it any more. It seems, therefore, that the midpoint of the line is not colored $(0.5, 0.5, 0.0)$, which is the color of the point. Shouldn't it be so, though, by linear interpolation, as it is half-way between two end vertices colored $(1.0, 0.0, 0.0)$ and $(0.0, 1.0, 0.0)$, respectively? **End**

The program `sphereInBoxPOV.pov` *is in the folder* `ExperimenterSource/Chapter21/SphereInBoxPOV`.

Click for `sphereInBox1.cpp` Program Windows Project

Experiment 21.3. If you have successfully installed POV-Ray, then open `sphereInBoxPOV.pov` using that application and press the Run button at the top; if not, use any editor to at least view the code. Figure 21.17(a) is the output. Impressive, is it not, especially if you compare with the output in Figure 21.17(b) of `sphereInBox1.cpp` of Chapter 11, which, of course, is almost the exact scene captured with OpenGL's synthetic camera? The inside of the box, with the interplay of light evident in shadows and reflections, is far more realistic in the ray-traced picture (the front of the ray-traced box could do with an additional light source, though, but we wanted to keep the OpenGL and POV-Ray versions as similar as possible). **End**

The program `sphereInBoxPOV.pov` *is in the folder* `ExperimenterSource/Chapter21/ExperimentRadiosity`.

Experiment 21.4. Run again `sphereInBoxPOV.pov`. Then run again after uncommenting the line

```
global_settings{radiosity{}}
```

at the top to enable radiosity computation with default settings. The difference is significant, is it not? Figure 21.25(a) is the ray-traced output without radiosity, while Figure 21.25(b) is the output with radiosity. There clearly is much more light going around inside the box in the latter rendering. **End**

Appendix A

Projective Spaces and Transformations

Click for `turnFilm.cpp` Program Windows Project

Experiment A.1. Run `turnFilm.cpp`, which animates the setting of the preceding exercise by means of a viewing transformation. Initially, the film lies along the $z = 1$ plane. Pressing the right arrow key rotates it toward the $x = 1$ plane, while pressing the left one reverses the rotation. Figure A.11 is a screenshot midway. You cannot, of course, see the film, only the view of the lines as captured on it.

The reason that the lower part of the X-shaped image of the power lines cannot be seen is that OpenGL film doesn't capture rays hitting it from behind, as the viewing plane is a clipping plane too. Moreover, if the lines seem to actually meet to make a V after the film turns a certain finite amount, that's because they are very long and your monitor has limited resolution!

This program itself is simple with the one statement of interest being `gluLookAt()`, which we ask the reader to examine next. End