Computer Graphics Through OpenGL: From Theory to Experiments

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. For almost all experiments there is one link, which works in Windows, Mac OS and Linux environments, to bring up the program file, and another to bring up the Windows project, which, of course, works only in a Windows environment. Since source code is not available for the three experiments in the first chapter, the reader is pointed to a folder with Windows and Mac executables.

For experiments asking simply to run a book program, *Experimenter* brings that program up, while for those asking that a program be modified significantly, the modified program – the modification being made either in the code or comments – comes 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 and, possibly, in a file in the `ExperimenterSource/CodeModifications` folder, as well, for easy copy-and-paste. In this case, *Experimenter* will throw up the modified program 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 just the program, leaving you to make changes.

Note: The names of the folders in `ExperimenterSource` – e.g., `Experiment-RotateTeapotMore` 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.

Installing and using Experimenter: Download the directory `Experimenter-Source` 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. Windows users should note that *Experimenter* will be slow in bringing up each 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 programs on various platforms.

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

Executables for Windows and the Mac are in the folder `ExperimenterSource/Chapter1/Ellipsoid`.

Experiment 1.1. Open `ExperimenterSource/Chapter1/Ellipsoid` and, hopefully, you'll be able to run at least one of the two executables there for the `Ellipsoid` program – one for Windows and one for the Mac. 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 all that difficult.

That's it. There's really not much more to this program: no lighting or blending or other effects you may have heard of as possible using OpenGL (understandably, as the program was written just a few weeks into the semester). 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**

Executables for Windows and the Mac are 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 executables 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 middle of Figure 1.16 is a screenshot a few seconds into the animation.

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

Executables for Windows and the Mac are in the folder `ExperimenterSource/Chapter1/Dominos`.

Experiment 1.3. Our final program is a movie which shows a Rube Goldberg domino effect with "real" dominos. The executables are 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, among others. 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 on how to install OpenGL and run our programs on various platforms.*

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

*If your program compiles but doesn't run, this might be because your graphics card doesn't support OpenGL 4.3. See the note following item 2 in Section 2.11 for what to do in this case.

*When we refer to `square.cpp`, or any `program.cpp`, it's always to the original version in a folder in the `ExperimenterSource` directory, so if you've modified the code for an earlier experiment you'll need to copy back the original.

Click for square.cpp modified 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 2.11 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 (Block 2):

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

Click for square.cpp modified Program Windows Project

Experiment 2.6. For a more dramatic illustration of clipping, first replace the square in the original square.cpp with a triangle; in particular, replace the polygon code with the following (Block 3):

```
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 2.13. 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 (Figure 2.14 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 2.15 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. In fact, verify the following table:

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

```
// Set foreground (or drawing) color.
glColor3f(0.0, 0.0, 0.0);
glColor3f(1.0, 0.0, 0.0);
```

The square is drawn red because the *current* value 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 (Block 5):

```
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 2.17). 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 specified 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 that 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 because OpenGL draws in *code order*.

End

Click for `square.cpp` modified `Program` `Windows Project`

Experiment 2.10. Replace the polygon declaration part of `square.cpp` with (Block 6):

```
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 2.18. 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)`, so that the part drawing the polygon is now

```
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();
```

End

Click for `square.cpp` modified   Program   Windows Project

Experiment 2.12. Continue, replacing `GL_POINTS` with `GL_LINES`, `GL_LINE_STRIP` and, finally, `GL_LINE_LOOP`. **End**

Click for `square.cpp` modified   Program   Windows Project

Experiment 2.13. Replace the polygon declaration part of `square.cpp` with (Block 8):

```
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();
```

End

Click for `square.cpp` modified   Program   Windows Project

Experiment 2.14. In fact, it's often easier to decipher a 2D primitive by viewing it in outline. Accordingly, 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:

```
// Set polygon mode.
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

// Draw a triangle strip.
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();
```

End

Click for `square.cpp` modified `Program` `Windows Project`

Experiment 2.15. Replace the polygon declaration part of `square.cpp` with (Block 9):

```
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();
```

Apply both the filled and outlined drawing modes. **End**

Click for `square.cpp` modified `Program` `Windows Project`

Experiment 2.16. Replace the polygon declaration of `square.cpp` with (Block 10):

```
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 (Figure 2.22(a)). **End**

Click for `square.cpp` modified Program Windows Project

Experiment 2.17. Replace the polygon declaration of `square.cpp` with (Block 11):

```
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 2.22(b)). Next, keeping the same *cycle* of vertices as above, list them starting with `glVertex3f(80.0, 20.0, 0.0)` instead (Block 12):

```
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 2.22(c)) identical to the one output earlier! Because the cyclic order of the vertices is unchanged, shouldn't it be as in Figure 2.22(b) both filled and outlined? **End**

Click for `circle.cpp` Program Windows Project

Experiment 2.18. 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 2.24. Press '-' to decrease the number of vertices. The randomized colors are just a bit of eye candy. **End**

Click for `parabola.cpp` Program Windows Project

Experiment 2.19. Run `parabola.cpp`. Press '+/-' to increase/decrease the number of vertices of the approximating line strip. Figure 2.26 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 OpenGL window. **End**

Click for `circularAnnuluses.cpp` Program Windows Project

Experiment 2.20. Run `circularAnnuluses.cpp`. Three identical-looking red circular annuluses (Figure 2.29) 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 `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. We'll discuss the mechanics of one primitive blocking out another momentarily.

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 2.21. Okay, run `helix.cpp` now. All we see is a circle as in Figure 2.33(a)! 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 2.33(b))! **End**

Click for `helix.cpp` modified Program Windows Project

Experiment 2.22. 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 2.36(a)). View the upright version as well (Figure 2.36(b)), 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 2.23. 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 periphery of the window, as you can see from the screenshot in Figure 2.37. Can you guess why? Ignore the code, especially unfamiliar commands such as `glTranslatef()` and `glRotatef()`, except for the fact that the projection is perspective.

This kind of *peripheral distortion* of a 3D object is unavoidable in any viewing system which implements the synthetic-camera model. 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 and objects of interest tend to be centered. **End**

Click for `hemisphere.cpp` Program Windows Project

Experiment 2.24. 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 2.40 for a screenshot. **End**

Click for `hemisphere.cpp` Program Windows Project

Experiment 2.25. 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) 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**

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 using a single triangle strip defined by a giant `glBegin(GL_TRIANGLE_STRIP)-glEnd()` block containing 10 vertices and their color attributes (the last two vertices being identical to the first two in order to close the strip).

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 one-dimensional global vertex arrays, vertices and colors, respectively, as in squareAnnulus2.cpp, except, now, the arrays are flat and not 2D (because of the way C++ stores array data, we could, in fact, have specified vertices and colors as 2D arrays exactly as in squareAnnulus2.cpp if we had so wanted). **End**

Click for squareAnnulus4.cpp   Program   Windows Project

Experiment 3.4. Run squareAnnulus4.cpp.

The code is even more concise with the single 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 in hemisphere.cpp using the single

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

command instead. **End**

Click for squareAnnulusVBO.cpp   Program   Windows Project

Experiment 3.7. Fire up `squareAnnulusVBO.cpp`, which modifies `square-Annulus4.cpp` to store vertex-related data in VBOs. There is a simple animation, too, through periodically changing color values in a VBO. Figure 3.5 is a screenshot, colors having already changed. **End**

Click for `hemisphereMultidrawVBO.cpp` Program Windows Project

Experiment 3.8. Run `hemisphereMultidrawVBO.cpp`. The code, which buffers the vertex and index data of `hemisphereMultidraw.cpp` along the lines of `vbo.cpp`, should be intelligible to one who followed our analysis of the latter program. **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, as intertwined they are difficult to manage in a VBO. The outputs of the current program and `squareAnnulusAndTriangle.cpp` are identical. **End**

Click for `helixList.cpp` Program Windows Project

Experiment 3.10. Run `helixList.cpp`, which shows many copies of the same helix, variously transformed and colored. Figure 3.6 is a screenshot. **End**

Click for `multipleLists.cpp` Program Windows Project

Experiment 3.11. Run `multipleLists.cpp`. See Figure 3.7 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 3.9. **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 3.10 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 just-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 points 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 2.23, where you can see a screenshot as well. 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 3.12 is a screenshot. **End**

Click for `lineStipple.cpp` Program Windows Project

Experiment 3.18. Run `lineStipple.cpp`. Press the space bar to cycle through stipples. A screenshot is shown in Figure 3.13. **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 3.15 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. **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 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 clips off the half-space $x + 0.5y < 60.0$, which is the space below an angled plane parallel to the z-axis. Figure 3.18 is a screenshot of both clipping planes activated. **End**

Click for `hemisphere.cpp` Program Windows Project

Experiment 3.22. Run `hemisphere.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 3.23(a)). Replace the perspective projection statement

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

with

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

As this is equivalent to the original `glFrustum()` call, there is still distortion if the window's shape is changed. 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 3.23(b))! **End**

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

Experiment 3.23. Run `viewports.cpp` where the screen is split into two viewports with contents a square and a circle, respectively. Figure 3.25 is a screenshot.

A vertical black line is drawn (in 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 `windows.cpp`   `Program`   `Windows Project`

Experiment 3.24. Run `windows.cpp`, which creates two top-level windows (Figure 3.26). **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` modified 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)`

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 (Block 1*):

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

Figure 4.4 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 a teapot? Accordingly, change the modeling transformation and object definition part of box.cpp to (Block 2):

```
// 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 4.6).

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 (Block 3):

*To cut-and-paste you can find the block in text format in the file chap4codeModifications.txt in the directory ExperimenterSource/CodeModifications.

```
// 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 4.8(a). Verify by elementary geometry that the line loop forms a square with sides of length $4\sqrt{2}$ angled at 45° to the axes.

Uncomment the scaling. See Figure 4.8(b). The square now seems skewed to a non-rectangular parallelogram. Mathematically verify that the new vertices after the transformation $(x, y, z) \mapsto (x, 3y, z)$ is applied to the square's vertices are indeed those of 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 (Block 4):

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

glutWireTeapot(5.0);
```

Figure 4.9 is a screenshot.

The *rotation* command `glRotatef(`A`, `p`, `q`, `r`)` rotates each point of 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 counter-clockwise looking *from* (p, q, r) to the origin. In this experiment, then, the rotation is 60° CCW (counter-clockwise) 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 `box.cpp` 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 (Block 5):

```
// 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 4.13(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 (Block 6):

```
// 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 4.13(b). **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.10. Replace the entire display routine of the original `box.cpp` with (Block 10):

```
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 4.15(a) for a screenshot. The objects are a box and a sphere. **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.11. Continuing with the previous experiment, uncomment the `glRotatef()` statement. Figure 4.15(b) is a screenshot.

Again, the individual placements are fairly straightforward. Working backwards from where it is created we see that, after being translated to (5.0, 10.0, 0.0), the sphere is rotated 45° counter-clockwise about the z-axis and, of course, finally pushed 15 units in the $-z$ direction. We'll not compute the exact final coordinates of its center. The individual placement of the box is simple to parse as well and left to the reader.

It's the relative placement which is particularly interesting in this case. The sphere is no longer vertically above the box, though the transformation between them is still `glTranslatef(0.0, 10.0, 0.0)`! Before trying to explain what's going on, let's return to the basics for a moment. **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.12. Repeat Experiment 4.11. The modeling transformation and object definition part are as below (Block 11):

```
// 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 4.17(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 4.17(b). Finally, uncomment `glRotatef(45.0, 0.0, 0.0, 1.0)` as follows:

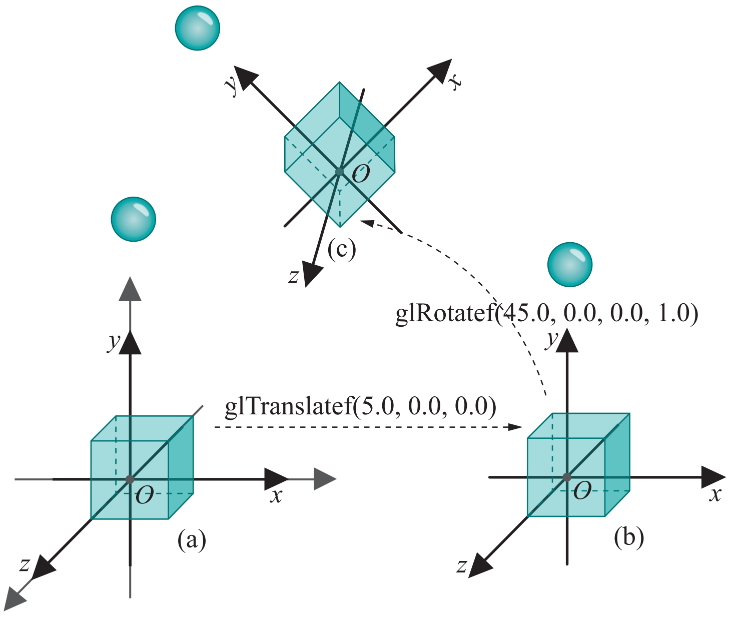

Figure 4.1: Transitions of the box, the box's local coordinates system (bold) and the sphere. The world coordinate system, which never changes, coincides with the box's initial local.

```
// 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 4.17(c). Figure 4.17 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 4.13. 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 written in black text, the rest white. Pressing the down arrow key goes forward a statement. Figure 4.18 is a screenshot after all transformations from the scaling on have been executed. **End**

Click for `box.cpp` modified `Program` `Windows Project`

Experiment 4.14. 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 (Block 12):

```
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 4.19(a)).

It doesn't work: the sphere is no longer round and is, moreover, some ways above the box (Figure 4.19(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 (Block 13):

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

   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.

   glFlush();
}
```

The resulting screenshot is Figure 4.19(c), which shows a round head on a neckless torso as desired. **End**

Click for `rotatingHelix1.cpp` `Program` `Windows Project`

Experiment 4.15. 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 4.21 is a screenshot. **End**

Click for `rotatingHelix2.cpp` `Program` `Windows Project`

Experiment 4.16. 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 speed – in particular, the speed at which frames can be redrawn – and the user cannot influence it. **End**

Click for `rotatingHelix3.cpp` `Program` `Windows Project`

Experiment 4.17. 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 4.22 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 4.18. Run `rotatingHelixFPS.cpp`, which enhances `rotatingHelix2.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.

The way this works is by `drawScene()` incrementing the global `frameCount` every time it is called, and `frameCounter()` outputting the value of `frameCount` each second – because of its final `glutTimerFunc(1000, frameCounter, 1)` statement, `frameCounter()` calls itself after a second – as well as 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 `main()` with the value passed being 0. **End**

Click for `rotatingHelix2.cpp` modified Program Windows Project

Experiment 4.19. 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 4.20. 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 4.24 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 4.25. There is no animation.

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 is 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 4.2 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 it. **End**

Click for `ballAndTorus.cpp` modified `Program` `Windows Project`

Experiment 4.21. We want to add a satellite that tags along with the ball of `ballAndTorus.cpp`. The following piece of code added to the end of the drawing routine – just before `glutSwapBuffers()` – does the job (Block 14):

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

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

See Figure 4.26 for a screenshot. For a revolving satellite add the following instead (Block 15):

```
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 4.22. Run `throwBall.cpp`, which simulates the motion of a ball thrown with a specified initial velocity subject to the force of gravity. Figure 4.27 is a screenshot.

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 increase/decrease the vertical component of the initial velocity and the page up/down keys to increase/decrease 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 4.23. 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 key 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.

End

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

Experiment 4.24. We start with simply a blue sphere for the head. See `clown1.cpp` which has the following drawing routine (note that `clown1.cpp` and `clown2.cpp` are not separate programs but incremental stages of `clown3.cpp` which is in `ExperimenterSource/Chapter4`):

```
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 4.29(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 4.30(a). The parameters *slices* and *stacks* determine the fineness of the mesh (not shown in the figure).

Accordingly, insert the 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 (Block 17):

```
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 (Block 18):

```
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 4.30(b). Insert the call `glutWireTorus(0.2, 2.2, 10, 25)` right after the call that draws the cone, so the drawing routine becomes (Block 19):

```
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 4.29(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 4.29(c). **End**

Click for `floweringPlant.cpp` Program Windows Project

Experiment 4.25. 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 4.31 is a screenshot. **End**

Click for `box.cpp` modified Program Windows Project

Experiment 4.26. 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)` so that the drawing routine is as below (Block 20):

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

   // Viewing transformation.
   gluLookAt(0.0, 0.0, 15.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

   glutWireCube(5.0); // Box.

   glFlush();
}
```

There is no change in what is viewed. 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, 1.0, 0.0)` are *exactly equivalent*.

Note: A convenient way to compare side-by-side the output of two pieces of code is to put them into the two drawing routines of `windows.cpp` of Chapter 3. **End**

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

Experiment 4.27. Continue the previous experiment, or run `boxWith-LookAt.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 4.28. 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 4.37(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 4.37(b))
2. $0.0, -1.0, 0.0$ (Figure 4.37(c))
3. $1.0, 1.0, 0.0$ (Figure 4.37(d))

Screenshots of the successive cases are shown in Figures 4.37(b)-(d). The camera indeed 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` modified `Program` `Windows Project`

Experiment 4.29. Replace the wire cube of `boxWithLookAt.cpp` with a `glutWireTeapot(5.0)` and replace the `gluLookAt()` call with:

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

The vector $los = (0.0, 0.0, 0.0) - (15.0, 0.0, 0.0) = (-15.0, 0.0, 0.0)$, 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 then is the up direction. Is what you see the same as Figure 4.37(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 4.30. Replace the display routine of `box.cpp` with (Block 21):

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

   glLoadIdentity();

   // Viewing transformation.
   gluLookAt(0.0, 0.0, 15.0, 15.0, 0.0, 0.0, 0.0, 1.0, 0.0);

   // Modeling transformation block equivalent
   // to the preceding viewing transformation.
   // glRotatef(45.0, 0.0, 1.0, 0.0);
   // glTranslatef(0.0, 0.0, -15.0);

   glutWireCube(5.0);

   glFlush();
}
```

Run. Next, both comment out the viewing transformation and uncomment the modeling transformation block following it. Run again. The displayed output, shown in Figure 4.47, is the same in both cases. The reason, as Figures 4.48(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 4.31. Run `spaceTravel.cpp`. The left viewport shows a global view from a fixed camera of a conical spacecraft and 40 stationary spherical asteroids arranged in a 5×8 grid. The right viewport shows the view from a front-facing camera attached to the tip of the craft. See Figure 4.51 for a screenshot of the program.

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.

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` modified Program Windows Project

Experiment 4.32. Run `spaceTravel.cpp` with `ROWS` and `COLUMNS` both increased to 100. The spacecraft now begins to respond so slowly to key input that its movement seems clunky, unless, of course, you have a super-fast computer (in which case, increase the values of `ROWS` and `COLUMNS` even more). **End**

Click for `animateMan1.cpp` Program Windows Project

Experiment 4.33. 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. See Figure 4.54. All parts are wireframe. We'll explain the program next. **End**

Click for `animateMan2.cpp` Program Windows Project

Experiment 4.34. 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 the movie.

The current contents of `animateManDataIn.txt` cause the man to do a handspring over the ball. Figure 4.56 is a screenshot. **End**

Click for `ballAndTorusShadowed.cpp` Program Windows Project

Experiment 4.35. Run `ballAndTorusShadowed.cpp`, based on `ballAnd-Torus.cpp`, but with additional 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 4.57 is a screenshot. **End**

Click for `selection.cpp` Program Windows Project

Experiment 4.36. 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 4.59 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 4.37. 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 4.62 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 5.24(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 5.24(b) illustrates the following simple scheme. 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:

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

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);
    glutWireCube(5.0); // Box.
```

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 in this case we are really more interested in the transformations in the program rather than its visual output.

The `gluLookAt(0.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)` statement we understand to multiply the current modelview matrix on the right by the matrix of its equivalent modeling transformation. The current modelview matrix is changed again by the `glMultMatrixf(matrixData)` call, which multiplies it on the right by the matrix corresponding to a rotation of 45° about the z-axis, equivalent to a `glRotatef(45.0, 0.0, 0.0, 1.0)` call. It's changed one last time by `glScalef(1.0, 2.0, 1.0)`.

The current modelview matrix is output to the command window initially and then after each of the three modelview transformations. We'll see next if the four output values match our understanding of the theory. **End**

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

Experiment 5.3. Run `shear.cpp`. Press the left and right arrow keys to move the ball. The box begins to shear when struck by the ball. See Figure 5.34.

The shear matrix is explicitly computed in the code and multiplied from the right into the current modelview matrix via a `glMultMatrix()` call. **End**

CHAPTER 6

Advanced Animation Techniques

Click for `spaceTravel.cpp` Program Windows Project

Experiment 6.1. First, run the program `spaceTravel.cpp` with its current values of `ROWS` and `COLUMNS`, being the size of the asteroid grid, as 8 and 5, 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. Dramatic isn't it, the speed-up from frustum culling?!

Note: Make sure the file `intersectionDetectionRoutines.cpp` is in the same directory as `spaceTravelFrustumCulled.cpp`.

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 (see Figure 6.6). Press the space bar, which is a toggle, to reveal a red wire sphere contained in the box. The sphere is our "complex" object, while the cube, of course, is its bounding box. 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);
glutWireSphere(0.5, 16, 10);
glEndConditionalRender();
```

Since OpenGL knows itself what not to render if the query outcome is zero, there is no need for user-instigated determination of results, so the variables `result` and `resultAvailable` are now gone. **End**

Click for `eulerAngles.cpp` Program Windows Project

Experiment 6.5. Run `eulerAngles.cpp`, which shows an L, similar to the one in Figure 6.8(a), whose orientation can be interactively changed.

The original orientation of the L has its long leg lying along the z-axis and its short leg pointing up parallel to the y-axis. 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. Figure 6.9 is a screenshot of the initial configuration. **End**

Click for `interpolateEulerAngles.cpp` Program Windows Project

Experiment 6.6. 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.

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 above the xz-plane, as diagrammed in Figure 6.10. Figure 6.11 is a screenshot of `interpolateEulerAngles.cpp` part way through the interpolation. **End**

Click for `eulerAngles.cpp` Program Windows Project

Experiment 6.7. 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 6.8. 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 currently fixed red L. See Figure 6.16 for a screenshot.

Pressing enter at any time begins an animation of the red L from the start 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 (Block 1*):

```
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. **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**

*To cut-and-paste you can find the block in text format in the file `chap7codeModifications.txt` in the directory `ExperimenterSource/CodeModifications`.

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 (Block 1[1]):

```
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 8.7(a)).

Next, keeping the *same* cycle of vertices as above, list them starting with `glVertex3f(80.0, 20.0, 0.0)` instead (Block 2):

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

[1]To cut-and-paste you can find the block in text format in the file `chap8codeModifications.txt` in the directory `ExperimenterSource/CodeModifications`.

```
      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 8.7(b))! Because the cyclic order of the vertices is unchanged, shouldn't it be as in Figure 8.7(a) both filled and outlined? **End**

Chapter 9

Orientation

Click for square.cpp modified   Program   Windows Project

Experiment 9.1. Replace the polygon declaration part of square.cpp with (Block 1[1]):

```
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.

Next, rotate the vertices cyclically so that the declaration becomes (Block 2):

```
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);
```

[1]To cut-and-paste you can find the block in text format in the file chap9codeModifications.txt in the directory ExperimenterSource/CodeModifications.

```
   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 listing next (Block 3):

```
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. **End**

Click for `square.cpp` modified `Program` `Windows Project`

Experiment 9.2. Replace the polygon declaration part of `square.cpp` with (Block 5):

```
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 9.11(a)). **End**

Click for `square.cpp` modified `Program` `Windows Project`

Experiment 9.3. Continuing the previous experiment, next replace the polygon declaration part of `square.cpp` with (Block 6):

```
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 9.11(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 9.12(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 9.12(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 9.13. 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`, which draws a green ball inside a red box. Press up or down arrow keys to open or close the box. Figure 9.15(a) is a screenshot of the box partly open.

Ignore the statements to do with lighting and material properties for now. The command `glCullFace(`*face*`)` where *face* can be `GL_FRONT`, `GL_BACK` or `GL_FRONT_AND_BACK`, is used to specify if front-facing or back-facing or all polygons are to be culled. Culling is enabled with a call to `glEnable(GL_CULL_FACE)` and disabled with `glDisable(GL_CULL_FACE)`.

You can see at the bottom of the drawing routine that back-facing triangles of the sphere are indeed culled, which makes the program more efficient because these triangles are hidden in any case behind the front-facing ones.

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 9.15(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` modified `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 a box-like room and now want to work on objects inside it. A good way to do this is to remove only the walls obscuring your view of the inside 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 `glDrawElements()` calls. The top and front sides of the box are not drawn, leaving its interior visible. Figure 9.15(c) is a screenshot. **End**

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

Experiment 9.9. Run `squareOfWallsReflected.cpp`, which is `square-OfWalls.cpp` with the following additional block of code, including a `glScalef(-1.0, 1.0, 1.0)` call, to reflect the scene about the yz-plane.

```
// Block to reflect the scene about the yz-plane.
if (isReflected)
{
   ...
   glScalef(-1.0, 1.0, 1.0);
   // glFrontFace(GL_CW);
}
else
{
   ...
   // glFrontFace(GL_CCW);
}
```

The original walls are as in Figure 9.16(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 9.16(b). **End**

Part V

Making Things Up

Chapter 10

Modeling in 3D Space

Click for `parabola.cpp` Program Windows Project

Experiment 10.1. Compare the outputs of `circle.cpp`, `helix.cpp` and `parabola.cpp`, all drawn in Chapter 2.

The sample is chosen uniformly from the parameter space in all three programs. The output quality is good for both the circle – after pressing '+' a sufficient number of times for a dense enough sample – and the helix. The parabola, however, shows a difference in quality between its curved bottom and straighter sides, the sides becoming smoother more quickly than the bottom. In curves such as this, one may want to sample non-uniformly, in particular, more densely from parts of greater curvature. **End**

Click for `astroid.cpp` Program Windows Project

Experiment 10.2. Run `astroid.cpp`, which was written by modifying `circle.cpp` to implement the parametric equations

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

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

Click for `cylinder.cpp` Program Windows Project

Experiment 10.3. Run `cylinder.cpp`, which shows a triangular mesh approximation of a circular cylinder, given by the parametric equations

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

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

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

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 10.5. Run `torus.cpp`, which applies the parametric equations deduced above in the template of `cylinder.cpp` (simply swapping 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 10.6. 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 10.7. These 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 10.8. The plan above is implemented in `doublyCurled-Cone.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 10.9. Run `extrudedHelix.cpp`, which extrudes a helix, using yet again the template of `cylinder.cpp`. The parametric equations of the extrusion are

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

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 10.10. 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 10.11. 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 10.12. 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 10.13. 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 10.14. Run `tetrahedron.cpp`. The program draws a wire-frame 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 10.15. 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.

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 to mimic the shape of its control polygon. **End**

Click for `bezierCurveWithEvalCoord.cpp` Program Windows Project

Experiment 10.16. Run `bezierCurveWithEvalCoord.cpp`, which draws a fixed Bézier curve of order 6. See Figure 10.66 for a screenshot. **End**

Click for `bezierCurveWithEvalMesh.cpp` Program Windows Project

Experiment 10.17. Run `bezierCurveWithEvalMesh.cpp`. This program is the same as `bezierCurveWithEval.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 the 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 $t2$. 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, which is a polyline approximation of part of the Bézier curve. **End**

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

Experiment 10.18. 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. Visually verify that the two curves meet smoothly 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 10.19. 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 10.20. 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 deep pocket.

(2) Push the middle control points of the end columns outwards along the x-direction to plump the pocket into a "canoe" with its front and back 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 10.21. 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 `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 10.78 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 again `sphereInBox1.cpp`, which we ran the first time in Section 9.4. Press the up-down arrow keys to open or close the box. Figure 11.18 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 and diffuse 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 program has further functionalities which we'll explain as they become relevant. **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 is a low intensity gray at $\{0.2, 0.2, 0.2, 1.0\}$.

Press 'w' or 'W' to toggle the **W**hite light off and on. Pressing 'g' or 'G' toggles the **G**reen light off and on. 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. Rotate the white light about the ball by pressing the arrow keys.

This program, too, 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. **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 until it looks flat as a disc. The reason for this is that the ambient intensity, which does not depend on eye or light direction, is uniform across vertices of the ball and cannot, therefore, provide the sense of depth that obtains from a contrast in color values across the surface. Diffuse light, on the other hand, which varies across the surface depending on light direction, can provide an illusion of depth.

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.

Moral: Diffusive reflectance lends three-dimensionality.

Figure 11.20 shows the ball starting with only ambient reflectance, then 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! This is because the vertex normals turn away from the viewer at

the now hidden ends of the ball, scaling down the diffuse reflectance there (recall the $\cos\theta$ term in the diffusive reflectance equation (11.7)). The result is that, with no ambient reflectance to offset the reduction in diffuse, the ends of the ball are dark.
Moral: Ambient reflectance provides a level of uniform lighting over a surface.
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. It's like the cat disappearing, leaving only its grin! Specular light is clearly for highlights and not much else. **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. Reducing the ambient and diffuse reflectances makes for a heavier and less plastic appearance. Restoring the ambient and diffuse to higher values, but reducing the specular reflectance makes it a less shiny plastic. Low values for all three of ambient, diffuse and specular reflectances give the ball a somewhat wooden appearance. **End**

Click for `lightAndMaterial2.cpp` Program Windows Project

Experiment 11.9. Run `lightAndMaterial2.cpp`.
Reduce the white light's diffuse and specular intensity to 0. The ball becomes a flat dull blue disc with a green highlight. This is because the ball's ambient (and diffuse) is blue and cannot reflect the green light's diffuse component, losing thereby three-dimensionality.
Raising the white global ambient brightens the ball, but it still looks flat in the absence of diffusive light. **End**

Click for Nate's site

Experiment 11.10. Nate Robins has a bunch of great tutorial programs at the site [100]. This is a good time to run his lightmaterial tutorial, which allows the user to control a set of parameters as well. **End**

Click for `spotlight.cpp` Program Windows Project

Experiment 11.11. 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 11.21 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 `litTriangle.cpp` Program Windows Project

Experiment 11.12. Run `litTriangle.cpp`, which draws a single triangle, whose front is coded 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 Figure 11.23 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 light in `lightAndMaterial2.cpp`.

The white wire sphere indicates the positional light, while the white arrow the incoming directional light. **End**

Click for `lightAndMaterial1.cpp` Program Windows Project

Experiment 11.14. Run `lightAndMaterial1.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. Move the ball by pressing the up/down arrow keys to observe the effect of attenuation. **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. A screenshot was shown earlier in Figure 11.21.

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 factor. A white wire mesh is drawn along the light cone boundary. **End**

Click for `spotlight.cpp` Program Windows Project

Experiment 11.16. Run again `spotlight.cpp`. Note the change in visibility of the balls near the cone boundary as the attenuation changes. **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 11.26. Flat shading causes each triangle in the 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 `sphereIn-Box1.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 specified to be the unit vector perpendicular to the square, then placing that square in a display list and, finally, drawing it six times appropriately rotated. Figure 11.34(b) shows the vertex normals to three faces. Figure 11.35 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 11.39 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 11.41 for a screenshot.

As promised, `litDoublyCurledCone.cpp` is pretty much a copy of `litCylinder.cpp`, except for the different `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 11.42 is a screenshot. **End**

Click for `litBezierCanoe.cpp` Program Windows Project

Experiment 11.23. Run `litBezierCanoe.cpp`. Press 'x/X', 'y/Y', 'z/Z' to turn the canoe. You can see a screenshot in Figure 11.44.

This program illuminates the final shape of `bezierCanoe.cpp` of Experiment 10.20 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, correspondingly, 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 is 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 11.24. Run `shipMovie.cpp`. Pressing space start an animation sequence which begins with a torpedo traveling toward a moving ship and which ends on its own after a few seconds. Figure 11.45 is a screenshot as the torpedo nears the ship. **End**

Click for `sizeNormal.cpp` Program Windows Project

Experiment 11.25. 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 11.47 is a screenshot. **End**

Click for `sizeNormal.cpp` modified Program Windows Project

Experiment 11.26. 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 `loadTextures.cpp` Program Windows Project

Experiment 12.1. Run `loadTextures.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. 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 it.

Important: Our own texture images are in the folder `ExperimenterSource/-Textures`.

Our programs all use the particular routine `getbmp()` to read in external image files, which is in the associated source file `getbmp.cpp`, and included in the application program via the header `getbmp.h`. Because `getbmp()` is written to accept input image files in *uncompressed 24-bit color RGB bmp* format, image files in other formats must first be converted, which can be done using image-editing software like Windows Paint, GIMP and Adobe Photoshop.

Note, though, that the input file is written *internally* into a 32-bit RGB**A** format, the A (alpha) field allowing for use in blending applications. **End**

Click for `loadTextures.cpp` modified Program Windows Project

Experiment 12.2. Replace every 1.0 in each `glTexCoord2f()` command of `loadTextures.cpp` with 0.5 so that the polygon specification is (Block 1[1]):

[1] To cut-and-paste you can find the block in text format in the file `chap12codeModifications.txt` in the directory `ExperimenterSource/CodeModifications`.

```
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 12.8(a)). Make sure to see both the launch and chessboard textures! **End**

Click for `loadTextures.cpp` modified Program Windows Project

Experiment 12.3. Restore the original `loadTextures.cpp` and delete the last vertex from the polygon so that the specification is that of a triangle (Block 2):

```
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 12.8(b)).

Change the coordinates of the last vertex of the world-space triangle (Block 3):

```
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 is not similar to its world-space counterpart (Figure 12.8(c)).

Continuing, change the texture coordinates of the last vertex (Block 4):

```
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 12.8(d)). **End**

Click for `loadTextures.cpp` modified   Program   Windows Project

Experiment 12.4. Restore the original `loadTextures.cpp` and replace `launch.bmp` with `cray2.bmp`, an image of a Cray 2 supercomputer. View the original 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 is scaled by half width-wise to fit the square polygon. **End**

Click for `loadTextures.cpp` modified   Program   Windows Project

Experiment 12.5. Restore the original `loadTextures.cpp` and then change the coordinates of only the third world-space vertex of the textured polygon (Block 5):

```
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 going on. Figure 12.9 shows both textures. **End**

Click for `loadTextures.cpp` modified   Program   Windows Project

Experiment 12.6. Restore the original `loadTextures.cpp` and change the texture coordinates of the polygon as follows (Block 7):

```
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* using the texture. See Figure 12.11.

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 12.12. **End**

Click for `loadTextures.cpp` modified   Program   Windows Project

Experiment 12.7. Change the texture coordinates again by replacing each -1.0 with -0.5 (Block 8):

```
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. **End**

Click for `loadTextures.cpp` modified Program Windows Project

Experiment 12.8. Restore the original `loadTextures.cpp` and then change the texture coordinates as below, which is the same as in Experiment 12.6 (Block 7):

```
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 `loadExternalTextures()` and `loadProcedural-Textures()` routines with `GL_CLAMP` so that it becomes

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

This causes the wrapping mode to be clamped in the s-direction. It's probably easiest to understand what happens in this mode by observing in particular the chessboard texture: see Figure 12.13. Texture s coordinates greater than 1 are clamped to 1, those less than 0 to 0. Precisely, instead of the texture space being tiled with the texture, points with coordinates (s, t), where $s > 1$, obtain their color values from the point $(1, t)$, while those with coordinates (s, t), where $s < 0$, obtain them from $(0, t)$. **End**

Click for `loadTextures.cpp` modified Program Windows Project

Experiment 12.9. Continue the previous experiment by clamping 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);
```

statement with `GL_CLAMP`. We leave the reader to parse the output. **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 12.14 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 and the correspondingly "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` modified 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. **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. Figure 12.19 is a screenshot of the initial configuration.

Compare, as the squares move, the quality of the textures delivered by the various min filters. 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 occupied by the square. **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 by `glGenerateMipmap()`. The mipmaps are very simple: just differently colored square images, created by the routine `createMipmaps()`, starting with the blue 64×64 `mipmapRes64` down to the black 1×1 `mipmapRes1`. Commands of the form

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

each binds a *width* × *height* mipmap image to the current texture index, starting with the highest resolution image with *level* parameter 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 12.20 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 the shuttle launch texture mapped onto a torus. Figure 12.22 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 10.21 – as you can see in the screenshot in Figure 12.24. Press space to start the propeller turning. **End**

Click for `texturedTorusAnimated.cpp` Program Windows Project

Experiment 12.17. Run `texturedTorusAnimated.cpp`, which animates the launch texture of `texturedTorus.cpp` by applying a translation to the current texture matrix. Press space to toggle between animation on and off, the up and down arrow keys to change its speed and 'x'-'Z' to turn the torus.

The modification of `texturedTorus.cpp` is simple. In particular, the following block of code in the drawing routine does the trick by entering texture matrix mode and applying a translation in the t-direction, effectively translating the texture coordinates in the t-direction:

```
glMatrixMode(GL_TEXTURE);
  glLoadIdentity();
  glTranslatef(0.0, shift, 0.0);
glMatrixMode(GL_MODELVIEW);
```

The rest of the modification of `texturedTorus.cpp` is in managing the animation. **End**

Click for `fieldAndSkyLit.cpp` Program Windows Project

Experiment 12.18. Run `fieldAndSkyLit.cpp`, which applies lighting to the scene of `fieldAndSky.cpp` with help of the `GL_MODULATE` option. The light source is directional – imagine the sun – and its direction controlled using the left and right arrow keys, while its intensity can be changed using the up and down arrow keys. A white line indicates the direction and intensity of the sun. Figure 12.26(a) is a mid-morning screenshot.

The material colors are all white, as is the light. 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 expected, consequence of a sky that doesn't darken together with land. **End**

Click for `litTexturedCylinder.cpp` Program Windows Project

Experiment 12.19. Run `litTexturedCylinder.cpp`, which adds a label texture and a can top texture to `litCylinder.cpp`. Press 'x'-'Z' to turn the can. Figure 12.26(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 statement in `setup()` which we're using 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. **End**

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

Experiment 12.20. Run `multitexture.cpp`, which interpolates between night and day sky texture. Press the left/right arrow keys to transition between night and day. Figure 12.27 shows stages in the transition. **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, 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. The closest one at depth 0.5 is vertical and a translucent red ($\alpha = 0.5$), the next one at depth 0.7 is angled and opaque green ($\alpha = 1$), while the farthest at depth 0.9 is horizontal and a translucent blue ($\alpha = 0.5$). Figure 13.3(a) is a screenshot of the output.

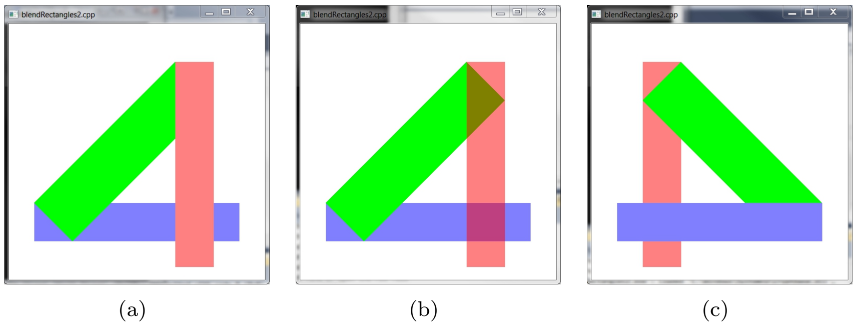

Figure 13.1: Screenshots of `blendRectangles2.cpp`: (a) Original (b) With rectangles re-ordered to blue, green, red in the code (c) New ordering seen from the $-z$-direction.

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` modified `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`. Press the arrow keys to move the sun. As the sun rises the night sky morphs into a day sky (yes, we saw this very same morph as an application of multitexturing, using the interpolation combiner, in Section 12.6). Figure 13.5(b) shows late evening. **End**

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

Experiment 13.6. Run `ballAndTorusReflected.cpp`, which builds on `ballAndTorusShadowed.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 `fieldAndSkyFogged.cpp` `Program` `Windows Project`

Experiment 13.7. Run `fieldAndSkyFogged.cpp`, which is based on our favorite workhorse program `fieldAndSky.cpp`, adding to it a movable black ball and controllable fog. Figure 13.6 is a screenshot.

Press the up/down arrow keys to move the ball. When the program starts there is no fog. We'll describe the fog controls after we discuss their implementation in the following. **End**

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

Experiment 13.8. Run `billboard.cpp`, where an image of two trees is textured onto a rectangle. Press the up and down arrow keys to move the viewpoint and the space bar to turn billboarding on and off. See Figure 13.9 for screenshots. **End**

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

Experiment 13.9. 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. A wire cube is drawn as frame of reference to view the motion.

Figure 13.11 shows screenshots of antialiasing both off and on. 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 13.10. Fire up again `antiAliasing+multisampling.cpp`. Multisampling is toggled on/off, independently of antialiasing, by pressing 'm' or 'M'.

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 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` have been enabled, which you can see as well. **End**

Click for `pointSprite.cpp` Program Windows Project

Experiment 13.11. 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, indicative of the possibilities. Figure 13.13 is a screenshot. **End**

Click for `sphereMapping.cpp` Program Windows Project

Experiment 13.12. Run `sphereMapping.cpp`, which shows the scene of a shuttle launch with a reflective rocket cone initially stationary in the sky in front of 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 13.15 is a screenshot as it's about to crash to the ground. **End**

Click for `ballAndTorusStenciled.cpp` Program Windows Project

Experiment 13.13. Run `ballAndTorusStenciled.cpp`, based on `ball-AndTorusReflected.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 you can see in the screenshot of Figure 13.24, the ball and torus are reflected only in the disc and nowhere else. **End**

Click for `imageManipulation.cpp` Program Windows Project

Experiment 13.14. Run `imageManipulation.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 the image with the left button pressed. Figure 13.25 is a screenshot of the initial configuration. **End**

Click for `imagaManipulationPBO.cpp` Program Windows Project

Experiment 13.15. Fire up `imageManipulationPBO.cpp`, which modifies `imageManipulation.cpp` to store the pixel data in a PBO. Otherwise, the functionality of both programs is identical, the image of the numeral being moved by mouse clicks and drags. **End**

Click for `bumpMapping.cpp` Program Windows Project

Experiment 13.16. 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 13.29 shows screenshots. **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 14.12 is a screenshot. **End**

Part VIII

Anatomy of Curves and Surfaces

CHAPTER 15

Bézier

Click for `deCasteljau3.cpp` Program Windows Project

Experiment 15.1. 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 15.4 is a screenshot. **End**

Click for `bezierCurves.cpp` Program Windows Project

Experiment 15.2. Run `bezierCurves.cpp`, 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 15.5 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 `bezierCurves.cpp` Program Windows Project

Experiment 15.3. Run `bezierCurves.cpp` and choose order 4 to get a feel for cubic Bézier curves. Note again how the curve mimics its control polygon. **End**

Click for `bezierCurves.cpp` Program Windows Project

Experiment 15.4. Run `bezierCurves.cpp` and choose the higher orders. It's straightforward to enhance the code for orders even greater than 6. **End**

Click for `bezierCurveTangent.cpp` Program Windows Project

Experiment 15.5. Run `bezierCurveTangent.cpp` 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. See Figure 15.10. Visually verify Proposition 15.1(g). **End**

Click for `sweepBezierSurface.cpp` Program Windows Project

Experiment 15.6. 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 15.14 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 15.7. Run `bezierSurface.cpp`, 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 15.15 is a screenshot. **End**

Chapter 16

B-Spline

Click for `bSplines.cpp` Program Windows Project

Experiment 16.1. 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 16.9 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 16.2. Run again `bSplines.cpp` and select the linear B-splines over the knot vector

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

Figure 16.13 is a screenshot. **End**

Click for `bSplines.cpp` Program Windows Project

Experiment 16.3. Run again `bSplines.cpp` and select the quadratic B-splines over the knot vector

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

Figure 16.18 is a screenshot. Note the joints indicated as black points. **End**

Click for `quadraticSplineCurve.cpp` Program Windows Project

Experiment 16.4. Run `quadraticSplineCurve.cpp`, which shows the quadratic spline approximation of nine control points over a uniformly spaced vector of 12 knots. Figure 16.21 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. **End**

Click for `bSplines.cpp` Program Windows Project

Experiment 16.5. Run `bSplines.cpp` and change the order to see a sequence of cubic B-splines. **End**

Click for `cubicSplineCurve1.cpp` Program Windows Project

Experiment 16.6. Run `cubicSplineCurve1.cpp`, which shows the cubic spline approximation of nine control points over a uniformly-spaced vector of 13 knots. The program is similar to `quadraticSplineCurve.cpp`. See Figure 16.23 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 16.7. 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 16.27(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 16.8. 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 16.28 is a screenshot of `quadraticSplineCurve.cpp` with a double knot at 5 and a triple at the end at 11. **End**

Click for `cubicSplineCurve1.cpp` Program Windows Project

Experiment 16.9. Run again `cubicSplineCurve1.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. **End**

Click for `quadraticSplineCurve.cpp` Program Windows Project

Experiment 16.10. 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 16.31(a) (or (b)). See Figure 16.32(a) and (b) for screenshots of the quadratic and cubic curves, respectively.

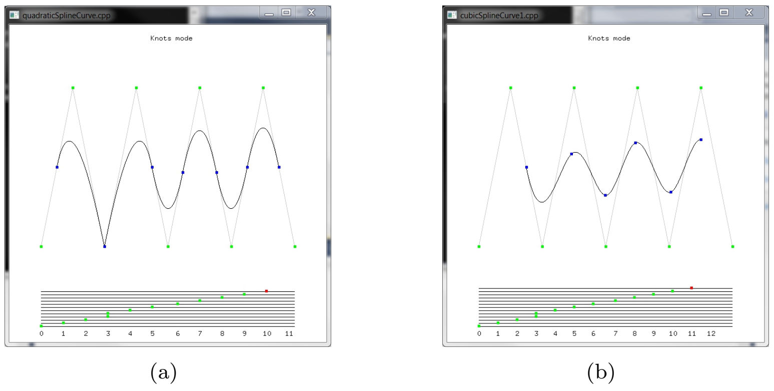

Figure 16.1: Screenshots of (a) `quadraticSplineCurve.cpp` and (b) `cubicSplineCurve1.cpp` over the knot vector $T = \{0, 1, 2, 3, 3, 4, 5, 6, 7, \ldots\}$ and approximating nine control points arranged in two horizontal rows.

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**

Click for `cubicSplineCurve1.cpp` Program Windows Project

Experiment 16.11. 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. Figure 16.33 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

Experiment 16.12. Make the first three and last three knots separately equal in `quadraticSplineCurve.cpp` (Figure 16.34(a)). Make the first four and last four knots separately equal in `cubicSplineCurve1.cpp` (Figure 16.34(b)). The first and last control points are interpolated in both. Do you notice any impairment in continuity? *No*! **End**

Click for `quadraticSplineCurve.cpp` Program Windows Project

Experiment 16.13. 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. **End**

Click for `cubicSplineCurve2.cpp` Program Windows Project

Experiment 16.14. 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 16.36 is a screenshot of the initial configuration.

The number of control points being much larger than the order, the user has good local control. **End**

Click for `bicubicSplineSurface.cpp` Program Windows Project

Experiment 16.15. Run `bicubicSplineSurface.cpp`, which draws a spline surface approximation to a 15×10 array of control points, each movable 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 16.38 is a screenshot. **End**

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

Experiment 16.16. Run `bicubicSplineSurfaceLitTextured.cpp`, which textures the spline surface of `bicubicSplineSurface.cpp` with a red-white chessboard texture. Figure 16.39 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 `trimmedBicubicBsplineSurface.cpp` `Program` `Windows Project`

Experiment 16.17. Run `trimmedBicubicBsplineSurface.cpp`, which shows the surface of `cubicBsplineSurface.cpp` trimmed by multiple loops. The code is modified from `bicubicBsplineSurface.cpp`, functionality remaining same. Figure 16.41(a) is a screenshot. **End**

Chapter 17

Hermite

Click for `hermiteCubic.cpp`   Program   Windows Project

Experiment 17.1. Run `hermiteCubic.cpp`, which implements Equation (17.10) 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 17.4 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 IX

Well Projected

Chapter 18

Applications of Projective Spaces

Click for `manipulateProjectionMatrix.cpp` Program Windows Project

Experiment 18.1. Run `manipulateProjectionMatrix.cpp`, a simple modification of `manipulateModelviewMatrix.cpp` of Chapter 5. Figure 18.4 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 with P(`glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0)`) computed with the help of Equation (18.4). **End**

Click for `ballAndTorusShadowMapped.cpp` Program Windows Project

Experiment 18.2. Before running `ballAndTorusShadowMapped.cpp` you may want to run again `ballAndTorusShadowed.cpp` from Section 4.7.2, which implements a simple-minded blacken-and-flatten strategy, with help of a scaling degenerate along the y-direction, to draw shadows on the floor.

The scenes of the two programs are almost identical, except, of course, 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. Figure 18.6 is a screenshot. **End**

Click for `rationalBezierCurve1.cpp` Program Windows Project

Experiment 18.3. 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 18.8 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 the control point weights are an additional set of "dials" at the designer's disposal for use to edit the 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 18.4. 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 18.9. 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 18.9 is at 1.13). This is no accident, as the following exercise shows. **End**

Click for `rationalBezierCurve3.cpp` Program Windows Project

Experiment 18.5. Run `rationalBezierCurve3.cpp`, which shows a rational Bézier curve on the plane specified by six control points. See Figure 18.10 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 `turnFilm2.cpp` Program Windows Project

Experiment 18.6. Run `turnFilm2.cpp`, which animates the snapshot transformation of a polynomial Bézier curve described above. Three control points and their red dashed approximating polynomial Bézier curve are initially drawn on the $z = 1$ plane. See Figure 18.12(a). The locations of the control points, and so of their approximating curve as well, are *fixed* in world space. However, they will *appear* to move as the film rotates.

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 changes position, so do the control points and the red dashed curve, these being the *projections* (snapshot transformations, particularly) onto the current film of the control points and their approximating curve (all fixed, as said, in world space). Also drawn on the film is a green dashed 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, and corresponding to the control points p_0, p_1 and p_2 and the solid red curve in Figure 18.11, 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 red dashed curve) coincides with the polynomial Bézier curve approximating the projected control points (the green dashed curve). This is to be expected because the control points coincide with their projections. However, as the film turns away from the $z = 1$ plane, the red and green dashed curves begin to separate. Their final configuration, when the film lies along $x = 1$, is shown in Figure 18.12(b).

There is more functionality to the program that we'll discuss momentarily. **End**

Click for `turnFilm2.cpp` Program Windows Project

Experiment 18.7. Fire up `turnFilm2.cpp` once again. Pressing space at any time draws, instead of the green dashed curve, a blue dashed *rational* Bézier curve approximating the projected control points on the current plane of the film. The control point weights of the blue dashed curve are computed according to the strategy just described. Voilà! The blue dashed rational curve and the red dashed projection are inseparable. **End**

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

Experiment 18.8. 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 18.14 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 X

The Time is Pipe

CHAPTER 19

Fixed-Functionality Pipelines

Click for `box.cpp` modified Program Windows Project

Experiment 19.1. Replace the box `glutWireCube(5.0)` of `box.cpp` 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 19.2 is a screenshot. **End**

Click for `perspectiveCorrection.cpp` Program Windows Project

Experiment 19.2. 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 19.4 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/Chapter19/SphereInBoxPOV`.

Experiment 19.3. If you have successfully installed POV-Ray, then open `sphereInBoxPOV.pov` from that program; if not, use any editor.

If you have installed POV-Ray, then press the Run button at the top; otherwise, open the output image file `sphereInBoxPOV.jpg` in the same folder as `sphereInBoxPOV.pov`. Figure 19.16(a) is a screenshot. Impressive, is it not, especially if you compare with the output in Figure 19.16(b) of `sphereInBox1.cpp`? The inside of the box, with the interplay of light evident in shadows and reflections, is far more realistic in the ray-traced picture. **End**

The program `sphereInBoxPOV.pov` *is in the folder* `ExperimenterSource/Chapter19/ExperimentRadiosity`.

Experiment 19.4. 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 19.24(a) is the ray-traced output without radiosity, while Figure 19.24(b) is the output with radiosity (both images are in the folder `ExperimenterSource/Chapter19/ExperimentRadiosity`). There clearly is much more light going around inside the box in the latter rendering. **End**

Part XI

Rendering Pipe Dreams

CHAPTER 20

OpenGL 4.3, Shaders and the Programmable Pipeline: Liftoff

Click for `squareShaderized.cpp` Program Windows Project

Experiment 20.1. Fire up the application program `squareShaderized.cpp`, which comes with its two sidekick shaders, the imaginatively named `vertexShader.glsl` and `fragmentShader.glsl`. Compiling and running the application program should automatically suck in the shaders provided they are all in the same folder. Not only is the functionality of `squareShaderized.cpp` – drawing a black square over white background, see Figure 20.3 – *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 20.2. 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. Figure 20.6 is a screenshot. **End**

Click for `bumpMappingShaderized.cpp` Program Windows Project

Experiment 20.3. Run `bumpMappingShaderized.cpp`. Interaction is the same as `bumpMapping.cpp`: press space to toggle between bump mapping on and off. Figure 20.7(b) is a screenshot with bump mapping enabled, evidently identical to that of `bumpMapping.cpp` in Figure 20.7(a). **End**

Click for `litCylinderShaderized.cpp` Program Windows Project

Experiment 20.4. Run `litCylinderShaderized.cpp`. Interaction is the same as `litCylinder.cpp`: press the 'x'-'Z' keys to turn the cylinder. As far as lighting is concerned, the main twist from `bumpMappingShaderized` is that now there is two-sided lighting, following `litCylinder.cpp` where it was activated by the statement

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

Material properties differ, as well, between the inside and outside of the cylinder. Figure 20.8 is a screenshot. **End**

Click for `bumpMappingPerPixelLight.cpp` Program Windows Project

Experiment 20.5. Run `bumpMappingPerPixelLight.cpp`. Again, press space to toggle between bump mapping on and off. Figure 20.7(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 20.6. 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 20.9 is a screenshot. **End**

Click for `texturedTorusShaderized.cpp` Program Windows Project

Experiment 20.7. Run `texturedTorusShaderized.cpp`. As in `texturedTorus.cpp` press 'x'-'Z' to turn the torus.

The point to note 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. See Figure 20.10 for a screenshot. **End**

Click for `litTexturedCylinderShaderized.cpp` Program Windows Project

Experiment 20.8. Run `litTexturedCylinderShaderized.cpp`. As in `litTexturedCylinder.cpp` press ‘x’-‘Z’ to turn the beer can. Figure 20.11 is a screenshot. **End**

CHAPTER 21

OpenGL 4.3, Shaders and the Programmable Pipeline: Escape Velocity

Click for `helixListShaderizedInstancedVertAttrib.cpp` Program Windows Project

Experiment 21.1. Run `helixListShaderizedInstancedVertAttrib.cpp`. The output of six different helixes is exactly the same as that of `helixList.cpp`. See Figure 21.1. **End**

Click for `helixListShaderizedShaderCounter.cpp` Program Windows Project

Experiment 21.2. Run `helixListShaderizedShaderCounter.cpp`. The output, just like that of `helixListShaderizedInstancedVertAttrib.cpp`, is the same as that of `helixList.cpp`. See Figure 21.2. **End**

Click for `ballAndTorusClipped.cpp` Program Windows Project

Experiment 21.3. Run `ballAndTorusClipped.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, there is now, as will be evident as soon as the animation is started, a clipping plane slicing through the torus's initial position. You can see this from Figure 21.3. Let's see how the clipping plane is set up. **End**

Click for `ballAndTorusShaderSubroutines.cpp` Program Windows Project

Experiment 21.4. 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 21.4 is a screenshot. **End**

Click for `ballAndTorusPickingShaderized.cpp` Program Windows Project

Experiment 21.5. 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. See Figure 21.5 for a screenshot. **End**

Click for `ballsAndTorusTransformFeedback.cpp` Program Windows Project

Experiment 21.6. 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 21.6 is a screenshot when the balls are close. **End**

Click for `fieldAndSkyTexturesBlendedShaderized.cpp` Program Windows Project

Experiment 21.7. Run `fieldAndSkyTexturesBlendedShaderized.cpp`. As in `fieldAndSkyTexturesBlended.cpp` press the arrow keys to move the sun, the transition between day and night happening from a blending of day and night textures. See Figure 21.7 for a screenshot. **End**

Click for `points.cpp` Program Windows Project

Experiment 21.8. Run `points.cpp`. Drawn are three large points. Press space to cycle between four different point renderings and the up and down arrow keys to move the points parallel to the z-axis. Figure 21.8 is a screenshot of what is seen at first, particularly, three unedited points. **End**

Click for `tessellatedCurve.cpp` Program Windows Project

Experiment 21.9. 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 21.12 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, the reason being, as we'll see in the upcoming discussion, an intentionally simple-minded choice of the control points; keep in mind, though, that the actual curve itself is of far less interest here than the programming steps to generate it.

The code is explained through the following sections. **End**

Click for `tessellatedHemisphere.cpp` Program Windows Project

Experiment 21.10. Run `tessellatedHemisphere.cpp`. You see a tessellated hemisphere. Press 'x'-'Z' to turn it, the up/down arrow keys to increase/decrease the inner tessellation levels (both equal) and the right/left arrow keys to increase/decrease the outer tessellation levels (all four equal). Figure 21.19 is a screenshot after raising both inner and outer tessellation levels a fair amount.

We ask the reader to examine the program in the following exercise. **End**

Click for `torusSilhouette.cpp` Program Windows Project

Experiment 21.11. 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 21.23 is a screenshot of the torus in silhouette. We explain the program below. **End**

Appendix A

Projective Spaces and Transformations

Click for `turnFilm1.cpp` Program Windows Project

Experiment A.1. Run `turnFilm1.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**