CHAPTER 9

Orientation

T he notion of orientation is vitally important in CG when drawing 3D scenes but, unfortunately, often confusing for the beginner. OpenGL itself makes critical use of orientation to determine the visible side of a surface. Note that the word "orientation" in the current context relates to handedness, e.g., clockwise or counter-clockwise, as we shall see, and has nothing to do with the orientation of a camera as discussed in Section 4.6.3, where the word meant pose or arrangement. The goal for this chapter is an understanding of orientation and its utility in CG.

The first section motivates the concept of orientation with a benign thought experiment. Section 9.2 describes how OpenGL applies orientation to determine the particular side of a 2D primitive which the viewer sees and then renders it with that side's specified material properties. If an object is specified as a collection of triangles, as in a triangulation, the question then arises of consistently orienting the collection. This is the topic of Section 9.3. Section 9.4 describes how OpenGL can make use of orientation to improve the efficiency of its rendering pipeline by culling certain triangles belonging to a closed surface, a procedure called back-face culling. In Section 9.5 we see how geometric transformations affect the perceived orientation of a primitive. We conclude in Section 9.6.

Although the three are conceptual in nature without a lot of excitement by way of programming, this chapter and the two preceding ones form a good part of the geometric core of CG.

9.1 Motivation

A thought experiment:

You and your friend, environmentally conscious types both, are headed separately toward a meeting of the Tree Huggers' Union. The meeting is out in the open in a field with, well, lots of trees and no other landmarks. There is, though, a triangle of long helium-filled balloons with the letters T, H and U at the corners floating high above the meeting site. See Figure 9.1 (ignore superman with a spray can and the sheet in the middle for now).

Now, you want to meet up with your friend before running into the crowd. So while walking you call him on his cell phone to try to figure out how he is currently situated with respect to you. How do you do this?

As both can see the balloons, a start is to determine if you are on the same side or not. Unfortunately, the letters at the corners (carefully chosen, of course) are of no help as they each look the same from either side.

What you can do, though, is ask your friend, "Does the vertex sequence THU – that's T→H→U – appear CW (clockwise) or CCW (counter-clockwise) from where

Chapter 9
ORIENTATION

Figure 9.1: Meeting of the Tree Huggers' Union.

you are?". If the orientation appears the same for both, then you are on the same side of the balloons; if not, you are on opposite sides.

OpenGL, as well, must determine for each triangle if the viewer currently sees one side or the other. And, as we'll see, it does so in an exactly similar manner. We'll understand as well the reason for this (seemingly) roundabout method.

Why does OpenGL need to distinguish sides in the first place? Because they may have properties (e.g., outlined/filled, color, etc.) specified differently by the programmer and OpenGL is obliged to display accordingly. For example, if the inside of a triangulated bowl is green and the outside red, then the two sides of every triangle composing it are colored differently as well. Given a viewpoint, OpenGL must determine the visible side of each triangle and render it with the appropriate color. See Figure 9.2. From the current (reader's) viewpoint the red side of triangle t_1 and the green side of triangle t_2 are visible. If the viewpoint travels $180°$ around the bowl, then the visible side is reversed for both.
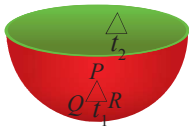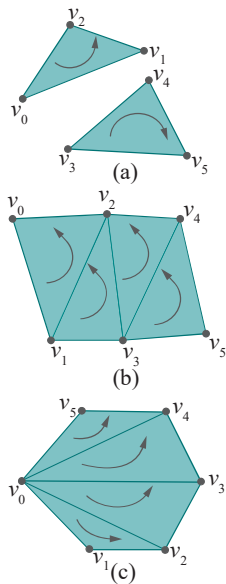

Figure 9.2: A bowl of two colors.

9.2 OpenGL Procedure to Determine Front and Back Faces

Here then is the procedure that OpenGL follows.

(1) First, it obtains the vertex orders of each 2D primitive *from the code*. For example, the declaration

```
glBegin(GL_TRIANGLES);
    v0; v1; v2; v3; v4; v5;
glEnd();
```

specifies the order of the vertices of the first triangle as `v0`, `v1`, `v2` and that of the second as `v3`, `v4`, `v5` (these orders, in fact, are part of the `GL_TRIANGLES` definition; see Section 2.6). See Figure 9.3(a). For other examples, the declaration

```
glBegin(GL_TRIANGLE_STRIP);
    v0; v1; v2; v3; v4; v5;
glEnd();
```

specifies the vertex orders of the four successive triangles in the strip as `v0`, `v1`, `v2` and `v1`, `v3`, `v2` and `v2`, `v3`, `v4` and `v3`, `v5`, `v4` (Figure 9.3(b)) and

Figure 9.3: Ordered triangles and triangle strip.

```
glBegin(GL_POLYGON);
    v0; v1; v2; v3; v4; v5;
glEnd();
```

specifies v0, v1, v2 and v0, v2, v3 and v0, v3, v4 and v0, v4, v5 (Figure 9.3(c)). And, similarly, each 2D primitive imposes an order on the vertices of its component triangles.

(2) Second, OpenGL determines for each component triangle if the order of its vertices as determined in Step (1) is *perceived* as CW or CCW by the viewer. This is said to be the *orientation* of the triangle with respect to the viewer.

OpenGL can make this determination because it knows both the location of the viewer – at the origin in case of perspective projection and at some point on the viewing face (it doesn't matter which) in case of orthographic projection – and those of the triangle's vertices. For example, in Figure 9.4, if the vertex order of the triangle is P, Q, R, then it is perceived as CCW by the viewer. We'll see later in this section an algorithm to output its orientation given an ordered input triangle.

(3) Finally, those triangles whose orientation the viewer perceives as CCW are presumed to be *front-facing*, i.e., the viewer is presumed by OpenGL to see their front faces, while those whose orientation is perceived as CW are *back-facing*. This is actually the default, which can be flipped with a `glFrontFace(GL_CW)` call. Front-facing triangles then are rendered with properties specified for their front faces, and back-facing ones with those for their back faces.

For example, if the vertex order of triangle t_1 in Figure 9.2 happens to be P, Q, R and the viewer is the reader, then OpenGL determines that this triangle is oriented CCW with respect to the viewer, who sees, therefore, the front face. Accordingly, t_1 is rendered red, assuming that the code indeed specifies that front-facing triangles are red. In Figure 9.4 we show the red rendering on the viewing face itself, pretending that it is the OpenGL window.
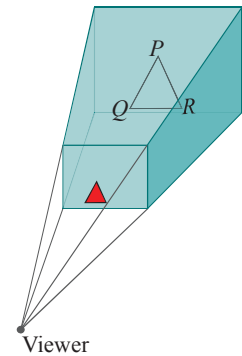

Figure 9.4: PQR is oriented CCW to the viewer, so it's rendered (as the red triangle on the viewing face) according to properties for its front face.

The reader may wonder at this point why one needs to invoke a *particular* viewpoint to distinguish sides. In real life the inside of the bowl (which is *absolute* and does not depend on the location of any viewer) is painted green and the outside (absolute as well) red. Subsequently, what a viewer sees is determined simply by the laws of nature, in particular, how light from the bowl travels to her eyes.
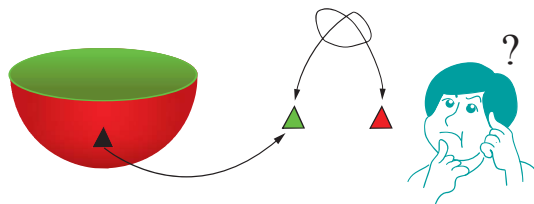

Figure 9.5: Was the inside of the bowl red or green?

Why doesn't OpenGL try to simulate this phenomenon? The answer is that, yes, it is true that the inside and outside of the bowl are absolute irrespective of the viewer, but *only after the entire bowl has been created*! If one breaks off a tiny piece of the bowl – a tiny nearly flat triangle, if you will – and shows it to someone who has never seen the whole, then it is not possible for that person to decide which side of the piece originally lay on the bowl's inside and which the outside (Figure 9.5). OpenGL has no global notion of objects either as it simply draws them triangle by triangle, and, therefore, requires direction from the programmer as to which side of each triangle is which.

The three-step procedure described above provides exactly a mechanism for such direction to OpenGL. Let's return to the thought experiment at the start of the section and assume that there is a giant triangular sheet of paper attached to the balloons (as in Figure 9.1) which you know is colored differently on either side. Then, of course, you could ask your friend, "What color do you see up there?", instead of "Does the vertex sequence THU appear CW or CCW from where you are?". The point is that the two questions are exactly equivalent in that those who perceive a particular orientation see a particular side and vice versa.

Continuing with this line of thought, suppose as you are walking that you notice a man high up about to spray-paint the sides of the triangular paper and he has a cell phone which you can call. You could then either ask him to arbitrarily paint one side green and the other red, which would at least serve the purpose of locating your friend, or you could ask him to paint *your* side red and the *other* side green, which is equivalent to you (the programmer) dictating that "CCW-seers" see red and "CW-seers" green.

There are three points worth emphasizing:

(a) A real-life 2D object (like a piece of paper) actually has two physical sides regardless of which an observer sees. This is not true of OpenGL, whose objects are all, of course, virtual. An OpenGL 2D primitive such as a triangle consists simply of data, e.g., vertex coordinates, color values, etc., residing inside the computer.

When asked to draw, OpenGL determines *if* the viewer is *supposed* to see the front or the back face according to the procedure described earlier and then *renders* the primitive with properties specified for that face.

(b) The terms "front-facing" and "back-facing" are simply used to indicate one side and the other. There is no *intrinsic* front or back of an OpenGL triangle or other virtual 2D primitive. If we didn't use these terms, we would have to say things like "the side which the viewer sees when the order `v0v1v2` appears clockwise from the origin".

(c) OpenGL draws primitives *individually*. It has no global understanding of objects formed by these primitives *together*.

Exercise 9.1. If a triangle t is specified by

    `glBegin(GL_TRIANGLES);` v_0; v_1; v_2; `glEnd();`

where the vertices are as below, in each case determine which side of t, front or back, a viewer at the origin sees, assuming the default of `glFrontFace(GL_CCW)`:

(a) $v_0 = (1, 0, 0)$, $v_1 = (0, 1, 0)$, $v_2 = (0, 0, 1)$

    Answer:

    The back face because $v_0 v_1 v_2$ appears CW from O. See Figure 9.6.

(b) $v_0 = (0, 1, 0)$, $v_1 = (1, 0, 0)$, $v_2 = (0, 0, 1)$

(c) $v_0 = (-1, 0, 0)$, $v_1 = (0, -1, 0)$, $v_2 = (0, 0, -1)$

(d) $v_0 = (1, 1, 1)$, $v_1 = (1, 1, -2)$, $v_2 = (-1, 1, -2)$

Exercise 9.2. A tacit assumption in all of the preceding discussion is that a viewer at a particular location sees, in fact, *only one* side – front or back – of a 2D primitive. For example, if a viewer could see both sides of a triangle, then is it front or back facing (or both)? So, is the assumption that only one side is visible a valid one? *Hint*: A triangle is always *flat* (planar), while a polygon should be specified to be so.

Definition 9.1. Two orders of the vertices of a polygon are said to be *equivalent* if one can be *cyclically rotated* into the other.
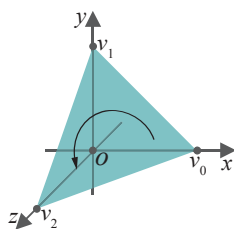

Figure 9.6: $v_0 v_1 v_2$ appears CW from O.

It follows that the sequence of vertices around any given polygon can be written in exactly *two* inequivalent orders. For example, the sequence of vertices around the quadrilateral q of Figure 9.7 can be written in eight different ways:

$$v_0v_1v_2v_3 \qquad v_1v_2v_3v_0 \qquad v_2v_3v_0v_1 \qquad v_3v_0v_1v_2$$

$$v_0v_3v_2v_1 \qquad v_3v_2v_1v_0 \qquad v_2v_1v_0v_3 \qquad v_1v_0v_3v_2$$

The orders on the top line are all equivalent to each other, while those on the second all to each other as well, and none on the first equivalent to any on the second. The notion of equivalence is important precisely because of the fact that a viewer on one side of a polygon perceives equivalent orders of vertices as either all CW or all CCW.

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

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

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

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

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

As the vertex order remains equivalent to the previous one, the square is still outlined. Reverse the vertex listing next:

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

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

Exercise 9.3. (Programming) Continuing with the preceding experiment (after the last part) add the following statements just before the polygon declaration.

```
glTranslatef(50.0, 0.0, 0.0);
glRotatef(180.0, 0.0, 1.0, 0.0);
glTranslatef(-50.0, 0.0, 0.0);
```

Section 9.2
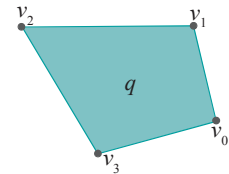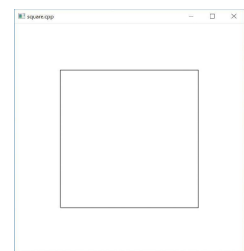**OpenGL Procedure to Determine Front and Back Faces**

Figure 9.7: A quadrilateral.

Figure 9.8: Screenshot of first part of Experiment 9.1.
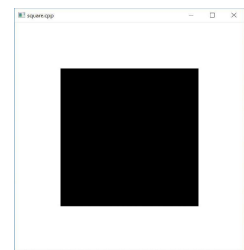

Figure 9.9: Screenshot of third part of Experiment 9.1.

Explain what you observe (Figure 9.8).

Exercise 9.4. (Programming) If the polygon declaration part of `square.cpp` is replaced with the following piece of code, then is an outlined or filled triangle seen? Try to answer first without running the program.

```
glFrontFace(GL_CW);
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
glBegin(GL_TRIANGLES);
    glVertex3f(80.0, 10.0, -1.0);
    glVertex3f(90.0, 75.0, 1.0);
    glVertex3f(15.0, 10.0, 0.5);
glEnd();
```

Remark 9.1. Before we get to Chapter 11 and learn about material properties and how to color the sides of an object differently, we'll have to do with distinguishing them by the distinctly unglamorous means of drawing one in outline and the other filled.

Algorithm to Decide the Orientation Perceived by a Viewer

An algorithmic question, that we did not address then, arose earlier in this section in Step (2) of OpenGL's procedure to determine the side of a primitive a viewer sees: given a viewpoint and a primitive with its vertices ordered, how to decide if the given order appears CW or CCW? We invite the reader to answer this for herself in the following exercise, with a fair amount of input from our end.

Exercise 9.5. Assume that the viewpoint is at the origin O and that the vertices of a triangle are $P = (x_1, y_1, z_1)$, $Q = (x_2, y_2 z_2)$ and $R = (x_3, y_3, z_3)$. See Figure 9.10. Determine if the viewer at O perceives the order PQR as CCW or CW.
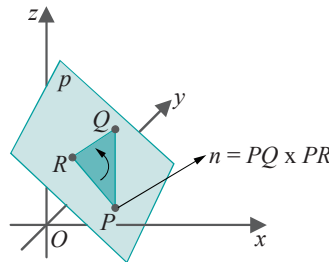

Figure 9.10: The plane p contains the triangle PQR: the orientation of PQR depends on which side of p the viewer is located.

If you don't do the exercise do at least read the conclusion below in terms of the determinant D.

Suggested approach: Supposing, first, that P, Q and R are not collinear, i.e., PQR is a non-degenerate triangle, determine the equation $ax + by + cz + d = 0$ of the unique plane p containing P, Q and R.

A point (x, y, z) lies on p if $ax + by + cz + d = 0$. A point lies in one (open) half-space of p, i.e., on one side of p, or the other, depending on whether $ax + by + cz + d < 0$ or $ax + by + cz + d > 0$.

Observe, next, that a viewer located *on* p sees triangle PQR "edge-on", in other words, as a line and not a triangle, so the question of orientation does not arise. A viewer not on p, on the other hand, perceives the orientation of PQR depending on the half-space she is in: particularly, all viewers inside one half-space perceive CCW, while those in the other CW.

Therefore, the perception at viewpoint O, in particular, depends on whether O lies on p or, if not, on which side.

Finally, conclude the following:

Let D be the determinant

$$\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{vmatrix}$$

1. If $D = 0$, then either (a) P, Q and R are collinear, in which case PQR is a degenerate triangle and the question of an orientation of PQR does not arise, or (b) O lies on the plane p containing P, Q and R, so that the viewer at O sees triangle PQR edge-on and, again, the question of orientation does not arise.

2. If $D > 0$, then the viewer at O perceives the order PQR as CW.

3. If $D < 0$, then the viewer at O perceives the order PQR as CCW.

Another approach is with the use of cross-products, by observing that $n = PQ \times PR$ is normal to the plane p and, in fact, points to the half-space where observers perceive PQR as CCW. Therefore, if the eye direction vector PO makes an angle of less than $90°$ with n – placing it in the same half-space as n – then the viewer at O perceives the order PQR as CCW as well; if greater, then as CW (in the configuration depicted in the figure the angle is, in fact, greater than $90°$). Whether the angle between the two vectors n and PO is greater or less than $90°$ can be decided from the sign of the dot product $n \cdot PO$.

Exercise 9.6. Does a viewer at the origin perceive the order PQR of the points $P = (-1, 2, 0)$, $Q = (3, 2, 2)$ and $R = (-3, -8, 6)$ as CW or CCW?

Exercise 9.7. Does a viewer at the point $O' = (1, 3, 2)$ perceive the order PQR of the points $P = (3, 7, 5)$, $Q = (4, 1, 2)$ and $R = (0, 1, 2)$ as CW or CCW?

Exercise 9.8. Relate Lemma 5.1 to the answer to Exercise 9.5.

9.3 Consistently Oriented Triangulation

The notion of orientation gets even more interesting when one considers a collection of triangles, as in a triangulation. The issue arises then of *consistency*. We have the following definition:

Definition 9.2. Suppose an order is given of the vertices of each triangle belonging to some triangulation $\mathcal{T}$ of an object X. $\mathcal{T}$ is said to be *consistently oriented* if any two triangles of $\mathcal{T}$ which share an edge order that edge oppositely; otherwise, $\mathcal{T}$ is *inconsistently oriented*.
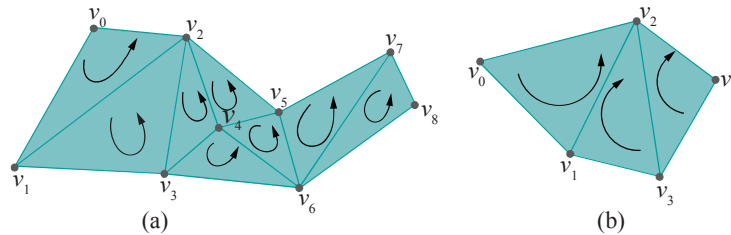

Figure 9.11: (a) Consistently oriented triangulation (b) Inconsistently oriented triangulation.

Figure 9.11(a) shows a consistently oriented triangulation. For example, the edge shared by the two triangles $v_0 v_1 v_2$ and $v_1 v_3 v_2$ is ordered $v_1 v_2$ by the first and $v_2 v_1$

by the second. The triangulation of Figure 9.11(b) is not consistently oriented as the edge shared by the two leftmost triangles is ordered v_1v_2 by both.

Intuitively, triangles in a consistently oriented triangulation of X appear oriented either all CW or all CCW "looking at one side of X". What exactly does this mean?
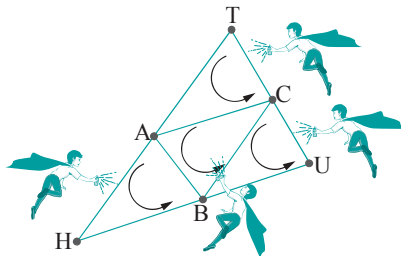

Figure 9.12: OpenGL spray-painting bots.

Let's return again to the earlier thought experiment at the point when you were about to call the painter. Looking up again you make out that the large triangular sheet is actually composed of four smaller ones and that there's a painter for each, so you'll have to call them separately (Figure 9.12). Moreover, all that you are allowed to specify to each is the order of his triangle's vertices – e.g., you can specify to the painter at the top his vertex order as either C→A→T or T→A→C – for these painters are nothing but OpenGL bots that have been programmed to do the following:

Determine if you perceive the order that you just called in as CCW or CW; if CCW then paint your side red, if not green.

Clearly, the onus then is on you to call in the four orders so that the small triangles are consistently oriented or else your side of the large triangle will be colored disparately.
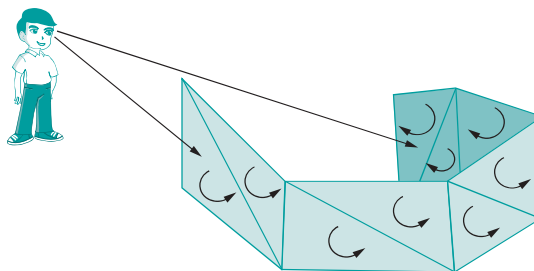

Figure 9.13: Man looking at both sides of a consistently oriented wall.

Are we saying that an observer at a given position can see only one side of a consistently oriented surface? Not at all. For example, the man in Figure 9.13 can see parts of both sides of the consistently oriented triangulated wall. However, he sees a change in side, according to the CW/CCW rule, only across boundary edges, never across an internal edge – which is physically authentic. If the wall were not consistently oriented, though, then this would not be the case. For example, the reader using the CW/CCW rule would believe herself to be seeing two different sides of the polygon of Figure 9.11(b) along the edge v_1v_2.

Recall again the bowl of Figure 9.2 with its inside green and outside red. If it's created in OpenGL as a triangulation, the programmer should then (a) specify that all front faces are of one color and back faces of the other, *and* (b) ensure consistent orientation of the triangulation so that the entire inside and entire outside appear of the desired colors, respectively.

In fact, the preceding rule should apply to all surfaces that we create. Here's what can happen if it doesn't.

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

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

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

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

    // CCW
    glVertex3f(80.0, 80.0, 0.0);
    glVertex3f(50.0, 20.0, 0.0);
    glVertex3f(80.0, 20.0, 0.0);
glEnd();
```
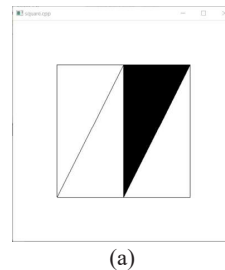
The specification is for front faces to be outlined and back faces filled, but, as the four triangles are not consistently oriented, we see both outlined and filled triangles (Figure 9.14(a)).   **End**

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

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(20.0, 80.0, 0.0);
    glVertex3f(20.0, 20.0, 0.0);
    glVertex3f(50.0, 80.0, 0.0);
    glVertex3f(50.0, 20.0, 0.0);
    glVertex3f(80.0, 80.0, 0.0);
    glVertex3f(80.0, 20.0, 0.0);
glEnd();
```
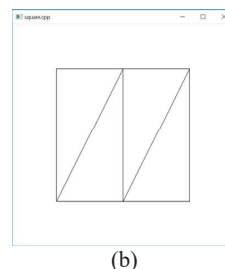
The resulting triangulation is the same as before, but, as it's consistently oriented, we see only outlined front faces (Figure 9.14(b)).   **End**

In the next experiment we see an example of a consistently oriented object, both sides of which are visible.

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

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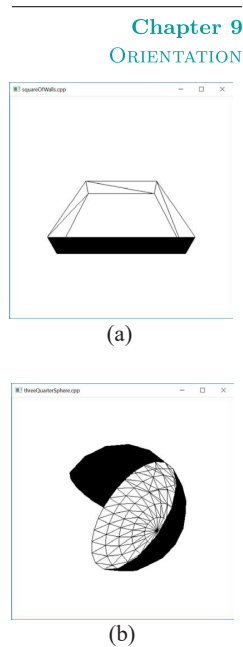

(a)

(b)

Figure 9.14: Screenshots for (a) Experiment 9.2 and (b) Experiment 9.3.

263

(a)

(b)

Figure 9.15: Screenshots of (a) `squareOfWalls.cpp` (b) `threeQuarter-Sphere.cpp`.

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

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

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

(a) Replace the loop statement

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

of the half-hemisphere with

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

to reverse its orientation.

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

(c) Place the additional polygon mode calls

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

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

(d) Call

```
glFrontFace(GL_CCW)
```

before the hemisphere definition and

```
glFrontFace(GL_CW)
```

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

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

It is not hard to orient consistently when creating objects in OpenGL because the primitives themselves tend to help. Verify from the definition of the drawing primitives in Section 2.6 that the set of triangles created by a call to `GL_TRIANGLE_STRIP` or `GL_TRIANGLE_FAN` is, in fact, consistently oriented. Therefore, a `GL_TRIANGLE_STRIP` or a `GL_TRIANGLE_FAN` call guarantees consistent orientation, at least for that particular set of triangles, so it's a good idea to use as many as possible.

Non-Orientable Surfaces

Before concluding this section, mention must be made of non-orientability. There do exist surfaces which can be triangulated but *never* consistently oriented. The most famous two, the Möbius band and Klein bottle, are depicted in Figure 9.16. Such surfaces are said to be *non-orientable*. Surfaces for which consistently oriented triangulations do exist are *orientable*.
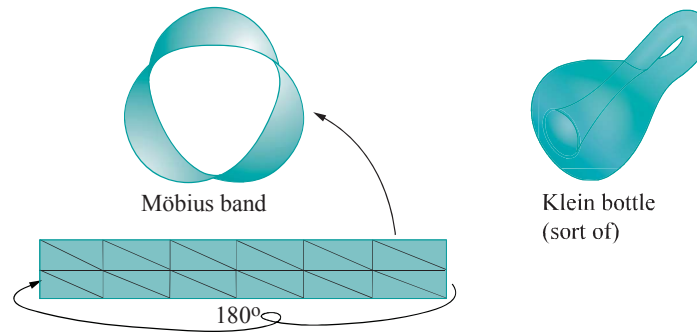

Figure 9.16: Non-orientable surfaces.

Experiment 9.6. Make a Möbius band as follows.

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

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

End

We have less to worry about with the Klein bottle, at least as far as real-world applications are concerned, because it cannot be created in 3-space. It needs at least 4D space to hold it properly. In fact, if you see Figure 9.16, the long neck of the bottle slips into the fourth dimension as it turns back to enter inside the bottle without, therefore, self-intersecting (a feat impossible in 3D space).

Further formalization of the notion of orientability requires knowledge of topology, but what we have discussed so far is ample from the point of view of first-level computer graphics. By the way, in case non-orientability looks like a potential can of worms, rest assured you will almost never encounter a non-orientable surface in the wild.

9.4 Culling Obscured Faces

Consider a *closed* surface such as a sphere, cube or torus, i.e., a surface that bounds a solid. See the first three surfaces in Figure 9.17. For example, the sphere bounds a ball, the torus bounds a solid torus and the box bounds a cube. If a closed surface is opaque, then a viewer outside of it sees only one side, no matter where she is located, while a viewer inside sees the other. Closedness is essential here, otherwise, a viewer may be able to see both sides, e.g., as the reader can see for the hemisphere or cylinder in Figure 9.17.

Such a situation where closedness blocks off a side of a surface is replicated in OpenGL by ensuring a consistently oriented triangulation of the given surface. For example, suppose the outside of the sphere of Figure 9.17 is painted red, and the inside green. Suppose, too, it's consistently oriented so that the orientation of the

Figure 9.17: First three closed surfaces, next two non-closed. The closed sphere is not shaded in order to reveal the inside; the green back face of a triangle is not visible from outside the sphere.

triangle PQR appears CCW, as shown in the figure, to a viewer outside the sphere (e.g., the reader). Then *any* viewer outside the sphere sees only front-facing (assuming the default of CCW = front-facing) triangles and never any back-facing ones (e.g., the green back face of the other triangle in the figure) because, for such a viewer, all back faces are on the inside and hidden behind front-facing triangles. The precise opposite is true for viewers inside the sphere who see only back-facing triangles.

Now, OpenGL cannot know if a surface is closed or not because this is a global decision to be made *after* the *entire* surface has been drawn (e.g., if even one triangle were missing from the sphere then it would no longer be closed). Closedness cannot, therefore, be determined by an API which simply draws one triangle after another. As a result, what happens, for example, in the case of the sphere above with the viewer outside, is that OpenGL processes *every* triangle, oblivious of the global picture, only to subsequently discard back-facing ones at the time of hidden surface removal because it's only then that OpenGL discovers back-facing triangles to be obscured by front-facing ones.

In such a situation then, knowing a viewer is outside the closed sphere, the programmer can help OpenGL be more efficient by directing it to not process any further a triangle as soon as it's been determined to be back-facing. This is called *back-face culling* or *polygon culling*.
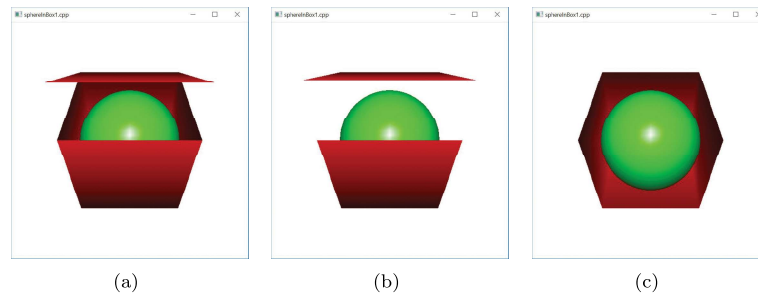

Figure 9.18: Screenshots for (a) Experiment 9.7 (b) Experiment 9.7 (disable culling commented out) (c) Experiment 9.8.

Expe**r**imen**t** **9.7.** Run `sphereInBox1.cpp` of Chapter 11, which draws a green ball inside a red box. Press up or down arrow keys to open or close the box. Figure 9.18(a) is a screenshot of the box partly open.

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

Comment out the `glDisable(GL_CULL_FACE)` call and open the box. Oops! Some sides of the box have disappeared as you can see in Figure 9.18(b). The reason, of

course, is that the state variable `GL_CULL_FACE` is set when the drawing routine is called the first time so that all back-facing triangles, including those belonging to the box, are eliminated on subsequent calls.

<div align="right">**End**</div>

Exercise 9.9. We know that back-facing triangles of a closed surface are obscured from a viewer outside. Is the converse true, in particular, are all obscured triangles necessarily back-facing?

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

Insert the pair of statements

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

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

<div align="right">**End**</div>

9.5 Transformations and the Orientation of Geometric Primitives

We know now how OpenGL uses the vertex order to determine the orientation of a primitive perceived by a viewer and, accordingly, the face seen, front or back. A reader, recollecting the theory of transformations, particularly Section 5.4.7 about orientation-preserving Euclidean transformations (i.e., rigid transformations) and orientation-reversing ones, may have already thought about and guessed the answer to the following question: how do these transformations affect the perceived orientation of a geometric primitive?

Answer: An orientation-preserving Euclidean transformation preserves the viewer's perceived orientation of the primitive, while an orientation-reversing one reverses it. An experiment will help make this clear.
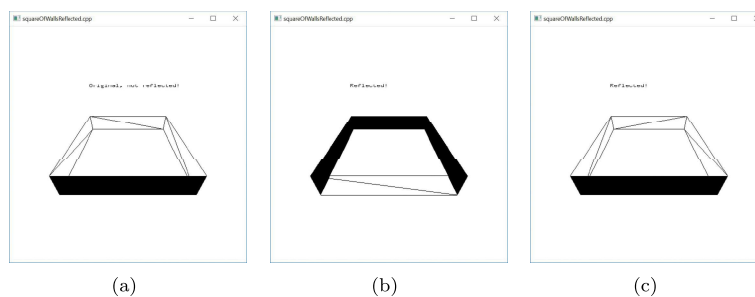

(a)   (b)   (c)

Figure 9.19: Screenshots from Experiment 9.9: (a) Original (b) Wrongly reflected (c) Correctly reflected.

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

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

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

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

Well, of course! The viewer's (default) agreement with OpenGL is that if she perceives a primitive's vertex order as CCW, then she is shown the front, if not the back. Reflection about the yz-plane, an orientation-reversing Euclidean transformation, flips all perceived orientations, so those primitives whose front the viewer used to see now have their back to her, and vice versa.

We likely want the reflection to transform the primitives but not simultaneously change their orientation. This is easily done by revising the viewer's agreement with OpenGL with a call to `glFrontFace(GL_CW)`. Accordingly, uncomment the two `glFrontFace()` statements in the reflection block. Now the reflection looks right, as shown in Figure 9.19(c). The primitives are clearly still being reflected about the yz-plane, but front and back stay the same.

9.6   Summary, Notes and More Reading

In this chapter we learned how OpenGL applies orientation to determine that side of a 2D primitive which is visible. We saw as well the importance of consistently orienting a triangulation. The technique of back-face culling to improve efficiency in rendering a closed surface was a useful addition to our repertoire. We learned as well how orientation-preserving and orientation-reversing transformations impact the orientation of a primitive.

Although our discussion of orientation at the elementary level is ample for the practical programmer, a fairly sophisticated mathematical setting is required to formalize the concept of the orientability of a surface. The interested reader is urged to look up an introductory topology text. The two by Munkres [103, 104], as well as the one by Singer & Thorpe [138], are classics. Incidentally, the mathematically-inclined student of CG will find many things of use in topology. One has only to scan the latest ACM SIGGRAPH papers [135] to see the heavy application of topological ideas in cutting-edge CG. Two introductory texts on the emerging area of computational topology are by Edelsbrunner [41] and Edelsbrunner and Harer [43].