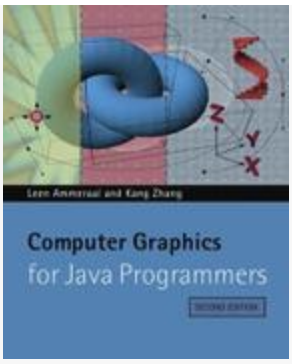

Computer Graphics for Java Programmers, Second Edition
by Leen Ammeraal; Kang Zhang*Published by John Wiley & Sons, 2007*

Chapter 7. Hidden-Face Elimination

In the previous two chapters we have discussed the wire-frame model and how to remove hidden-lines of 3D objects. For example, a realistic looking cube is displayed by simply drawing all the visible edges of the cube. Instead, we can construct a polygon for each of the six faces. If we display only the visible faces, we obtain a more realistic image of a cube. In Exercise 5.2 this was done by displaying only the top, right and front faces, taking the chosen viewpoint into account. Recall that we used $\theta = 0.3$ and $\varphi = 1.3$ (both in radians) at the beginning of class *Obj* in program *CubePers.java*. The solution of Exercise 5.2 will fail, for example, if we use $\theta = -0.3$ because then the left instead of the right face of the cube (see Figure 5.10) should be displayed as a filled polygon. In this chapter, the viewpoint will be taken care of automatically, so that the invisible faces of a cube, also known as *back faces*, are omitted regardless of the chosen viewpoint. Besides, a face may be only partly visible, which is the case, for example, if we have two cubes, with the nearer one partly hiding the farther one. The problem of displaying only the visible portions of faces will also be solved in this chapter. Finally, instead of taking just some colors, we will assume that there is a source of light, such as the sun, to determine the brightness or darkness of a face. Except for Section 7.1, we will display all faces of 3D objects in shades of yellow, against a blue background.

BACK-FACE CULLING

It is not difficult to determine which faces of a cube are invisible because they are at the back. Such faces, known as *back faces*, can be detected by investigating the

orientation of their vertices. To begin with, we specify each face counter-clockwise, when viewed from outside the object. For example, we denote the top face of the cube of Figure 5.11 as the counter-clockwise vertex sequence 4, 5, 6, 7, or, for example, 6, 7, 4, 5, but not 6, 5, 4, 7, for that would be clockwise. In contrast, we can specify the bottom face of the cube as the sequence 0, 3, 2, 1, which is counter-clockwise when this face is viewed from the outside but clockwise in our perspective image. Here we see that the orientation of the bottom face in the image is different from that in 3D space, when the object is viewed from the outside (that is, from below). This is because this bottom face is a back face. We use this principle to tell back faces from visible faces. The method *area*2, discussed in Section 2.7, will now be very useful, so that we will again use the class *Tools2D*, in which this method occurs. Recall that the complete version of class *Tools2D* can be found inSection 2.13.

Note that we should use screen coordinates in the test we have just been discussing. If we used the x_e- and y_e-coordinates instead (ignoring the *ze*-coordinate; see Figure 5.4), this test about the orientation of the vertices would be equivalent to testing whether the normal vector, perpendicular to the face in question and pointing outward, would point more or less towards us or away from us, that is, whether that vector would have a positive or a negative *ze*-component. This test would be correct with orthographic (or parallel) projection but not necessarily with central projection, which we are using. Exercise 7.1 deals with this subject in greater detail.

This time we will use the *Graphics* methods *setColor* and *fillPolygon*, to display each face, if it is visible, in a color that is unique for that face. Just before we do this (in the *paint* method) we test whether the face we are dealing with is visible; if it is not, it is a back face so that we can ignore it. To demonstrate that this really works, we will modify the viewpoint (by altering the angles θ and φ) each time the user presses a mouse button:

```
// Backface.java: A cube in perspective with back-face culling.
// Uses: Point2D (Section 1.5), Point3D (Section 3.9),
//       Tools2D (Section 2.13).

import java.awt.*;
import java.awt.event.*;

public class Backface extends Frame
{
   public static void main(String[] args){new Backface();}
```

```
   Backface()
   {  super("Press mouse button ...");
      addWindowListener(new WindowAdapter()
         {public void windowClosing(WindowEvent e){System.exit(0);}});
      add("Center", new CvBackface());
      Dimension dim = getToolkit().getScreenSize();
      setSize(dim.width/2, dim.height/2);
      setLocation(dim.width/4, dim.height/4);
      show();
   }
}

class CvBackface extends Canvas
{  int centerX, centerY;
   ObjFaces obj = new ObjFaces();
   Color[] color = {Color.blue, Color.green, Color.cyan,
                    Color.magenta, Color.red, Color.yellow};
   float dPhi = 0.1F;
   CvBackface()
   {  addMouseListener(new MouseAdapter()
      {  public void mousePressed(MouseEvent evt)
         {  obj.theta += 0.1F;
            obj.phi += dPhi;
   if (obj.phi > 2 || obj.phi < 0.3) dPhi = -dPhi;
            repaint();
         }
      });
   }

   int iX(float x){return Math.round(centerX + x);}
   int iY(float y){return Math.round(centerY - y);}

   public void paint(Graphics g)
   {  Dimension dim = getSize();
      int maxX = dim.width - 1, maxY = dim.height - 1,
          minMaxXY = Math.min(maxX, maxY);
      centerX = maxX/2;
      centerY = maxY/2;
      obj.d = obj.rho * minMaxXY / obj.objSize;
      obj.eyeAndScreen();
      Point2D[] p = new Point2D[4];
      for (int j=0; j<6; j++)
      {  Polygon pol = new Polygon();
         Square sq = obj.f[j];
         for (int i=0; i<4; i++)
```

```
         {  int vertexNr = sq.nr[i];
            p[i] = obj.vScr[vertexNr];
            pol.addPoint(iX(p[i].x), iY(p[i].y));
         }
         g.setColor(color[j]);
         if (Tools2D.area2(p[0], p[1], p[2]) > 0)
            g.fillPolygon(pol);
      }
   }
}

class ObjFaces // Contains 3D object data of cube faces
{  float rho, theta=0.3F, phi=1.3F, d;
   Point3D[] w;     // World coordinates
   Point3D[] e;     // Eye coordinates
                    // (e = wV where V is a 4 x 4 matrix)
   Point2D[] vScr; // Screen coordinates
   Square[] f;      // The six (square) faces of a cube.
   float v11, v12, v13, v21, v22, v23,
      v32, v33, v43, // Elements of viewing matrix V.
      xe, ye, ze, objSize;
   ObjFaces()
   {  w = new Point3D[8];
      e = new Point3D[8];
      vScr = new Point2D[8];
      f = new Square[6];
      // Bottom surface:
      w[0] = new Point3D( 1, -1, -1);
      w[1] = new Point3D( 1,  1, -1);
      w[2] = new Point3D(-1,  1, -1);
      w[3] = new Point3D(-1, -1, -1);
      // Top surface:
      w[4] = new Point3D( 1, -1,  1);
      w[5] = new Point3D( 1,  1,  1);
      w[6] = new Point3D(-1,  1,  1);
      w[7] = new Point3D(-1, -1,  1);
      f[0] = new Square (0, 1, 5, 4); // Front
      f[1] = new Square (1, 2, 6, 5); // Right
      f[2] = new Square (2, 3, 7, 6); // Back
      f[3] = new Square (3, 0, 4, 7); // Left
      f[4] = new Square (4, 5, 6, 7); // Top
      f[5] = new Square (0, 3, 2, 1); // Bottom
      objSize = (float)Math.sqrt (12F);
         // distance between two opposite vertices.
      rho = 3 * objSize; // For reasonable perspective effect
```

```
   }

   void initPersp()
   {  float
      costh = (float)Math.cos(theta), sinth = (float)Math.sin(theta),
      cosph = (float)Math.cos(phi), sinph = (float)Math.sin(phi);
      v11 = -sinth;   v12 = -cosph * costh;   v13 = sinph * costh;
      v21 = costh;    v22 = -cosph * sinth;   v23 = sinph * sinth;
                      v32 = sinph;            v33 = cosph;
                                              v43 = -rho;
   }

   void eyeAndScreen()
   {  initPersp();
      for (int i=0; i<8; i++)
      {  Point3D p = w[i];
         float x = v11 * p.x + v21 * p.y;
         float y = v12 * p.x + v22 * p.y + v32 * p.z;
         float z = v13 * p.x + v23 * p.y + v33 * p.z + v43;
Point3D Pe = e[i] = new Point3D(x, y, z);
         vScr[i] = new Point2D(-d * Pe.x/Pe.z, -d * Pe.y/Pe.z);
      }
   }
}

class Square
{  int nr[];
   Square(int iA, int iB, int iC, int iD)
   {  nr = new int[4];
      nr[0] = iA; nr[1] = iB; nr[2] = iC; nr[3] = iD;
   }
}
```

If you do not use the Java *classpath* variable, you should make sure that the files *Point2D.class*, *Point3D.class* and *Tools2D.class* (or the corresponding *.java* files) are in the same directory as this program. We now obtain images of a cube in bright colors, so the result on the screen looks much better than Figure 7.1. Initially, the top face of the cube is visible, but after we have clicked the mouse button several times the cube shows its bottom face, as is the case in this illustration.

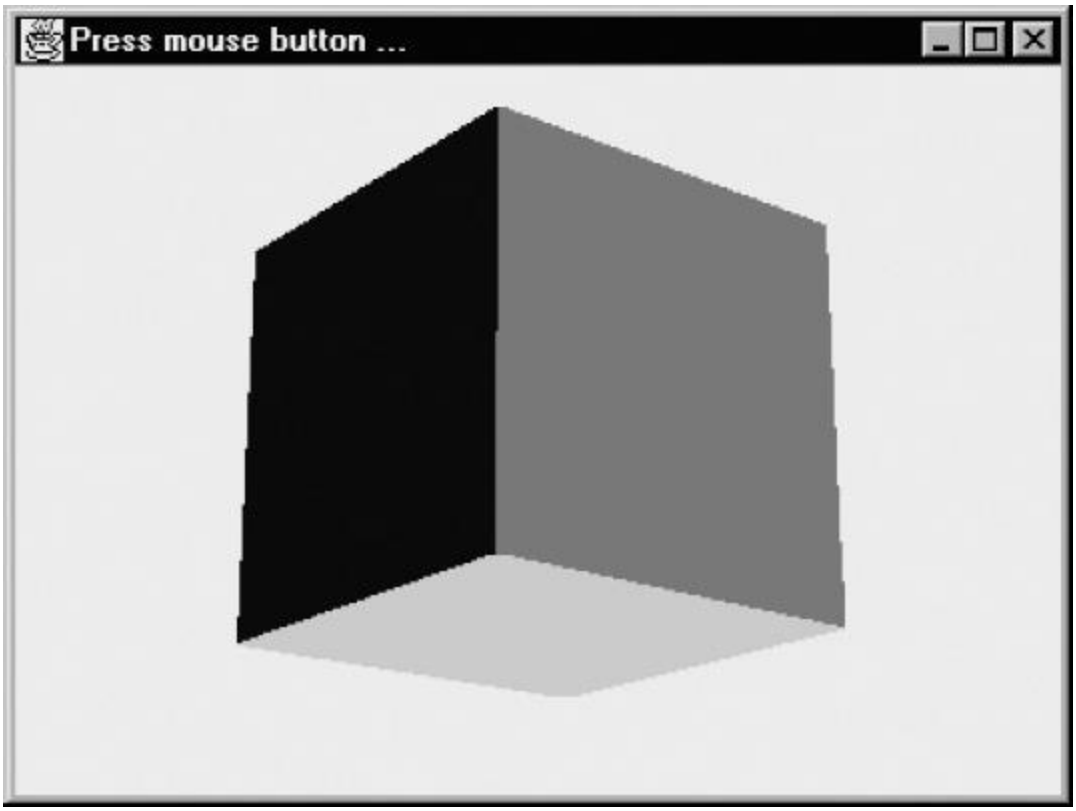

Figure 7.1. Result of back-face culling

We will use back-face culling in the programs that follow. This technique is worthwhile because it drastically reduces the number of polygons that may be visible and it is inexpensive compared with some more time-consuming algorithms to be discussed later in this chapter. However, using only back-face culling is not sufficient, since it does not work for non-convex solids. As Figure 6.12 illustrates, the solid letter *L* is not convex, since the line that connects, for example, vertices 1 and 11 does not lie completely inside the object. In that example, the rectangle 5-6-7-8 is not a back face, but it is only partly visible.

COLORING INDIVIDUAL FACES

Let us now decide on the color to be used for each polygon (and for all triangles of which it consists) based on its orientation in relation to the light source. For a polygon in a plane with equation

$ax + by + cz = h,$

we can obtain a color code by calling the method *colorCode* of the class *Obj*3*D*as follows:

```
int cCode = obj.colorCode(a, b, c);
```

(Note that we use the name *colorCode* both for an array and for a method.) This color code, stored in the variable *cCode*, will later be used as follows:

```
g.setColor(new Color(cCode, cCode, 0));
```

Using integers in the range 0–255 for *cCode*, we obtain shades of yellow in this way. For a given polygon, we will associate the values in this range with the inner product (see Section 2.2) between the following two vectors:

$\mathbf{s} = (1/\sqrt{3})(-1, 1, 1)$:	vector directed from the object to the sun; normal vector, perpendicular to the polygon and pointing away from the object (more or less towards the viewpoint E).
$\mathbf{n} = (a, b, c)$:	

Note that these vectors have length 1 and are expressed in eye coordinates x_e,y_e and z_e. The signs of the triple elements −1, 1, 1 imply that the light comes from the left, from above and from the front, as Figure 7.2 illustrates.

Since we know the range of the color codes, we can convert inner-product values to color codes, if we also know the range of all possible inner products that can occur. This range

inprodRange = inprodMax – inprodMin

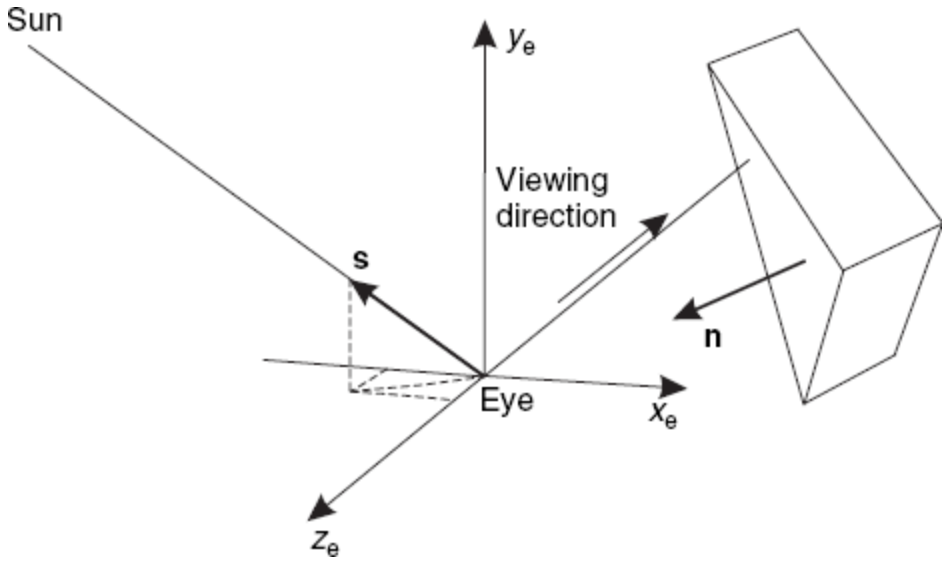

Figure 7.2. Vector s, pointing to the sun, and normal vector n of a face

is found in the method *planeCoeff* of the class *Obj*3*D* simply by computing all those inner products, as soon as *a*, *b* and *c* are computed.

The following method of *Obj*3*D.java* shows how an inner product **n** · **s**, discussed above, is computed and how the above values *inprodMin* and *inprodRange* are used to derive a color code in the range 0–255:

```
int colorCode(double a, double b, double c)
{  double inprod = a * sunX + b * sunY + c * sunZ;
   return (int)Math.round(((inprod -
inprodMin)/inprodRange) * 255);
}
```

The method *colorCode* needs to be called for every polygon, or face, to be colored. It is usually placed in the method *paint*, along with its color code and its distance, as demonstrated in the example program of the next section.

PAINTER'S ALGORITHM

It is an attractive idea to solve the hidden-face problem by displaying all polygons in a specific order, namely first the most distant one, then the second furthest, and so on, finishing with the one that is closest to the viewpoint. Painters sometimes follow the same principle, particularly when working with oil paintings. They typically start with the background and paint a new layer for objects on the

foreground later, so that the overlapped parts of objects in the background, painted previously, are covered by the current layer and thus become invisible. This algorithm is therefore known as the *painter's algorithm*. It is based on the assumption that each triangle can be assigned a *ze*-coordinate, which we can use to sort all triangles. One way of obtaining such a *ze*-coordinate for a triangle ABC is by computing the average of the three z_e-coordinates of A, B and C. However, there is a problem, which is clarified by Figure 7.3. For each of the three beams, a large rectangle is partly obscured by another, so we cannot satisfactorily place them (or the triangles in which these rectangles are divided) in the desired order, that is, from back to front (see also Figures 7.8 and 7.9). Consequently, the naïve approach just suggested will fail in this case. Surprisingly enough, it gives good results in many other cases, such as the solid letter *L* in Figure 6.2 and the sphere and cone in Figure 7.5. It is also very fast.

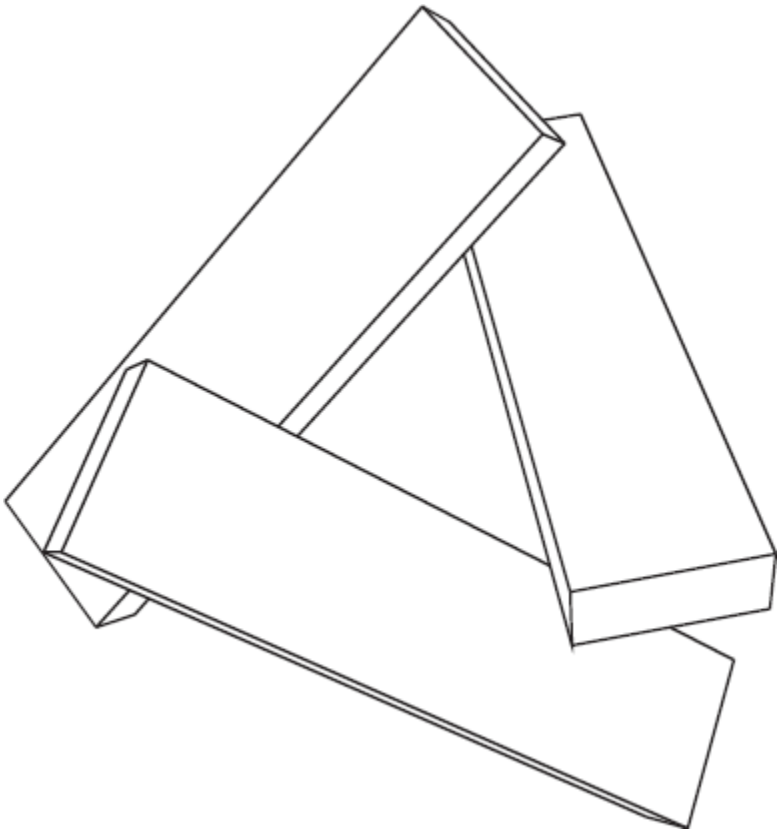

Figure 7.3. Beams, each of which partly obscures another

The program *Painter.java* is listed below. If you compile and run this program, all classes defined in the program files *Point2D.java, Point3D.java, Obj3D.java*, etc., must be available, as the comment below indicates. Recall our discussion of this subject at the end of Chapter 1.

```
// Painter.java: Perspective drawing using an input file that lists
//    vertices and faces. Based on the Painter's algorithm.

// Uses: Fr3D (Section 5.5) and CvPainter (Section 7.3),
//       Point2D (Section 1.5), Point3D (Section 3.9),
//       Obj3D, Polygon3D, Tria, Fr3D, Canvas3D (Section 5.5).
import java.awt.*;

public class Painter extends Frame
{  public static void main(String[] args)
   {  new Fr3D(args.length > 0 ? args[0] : null, new CvPainter(),
         "Painter");
   }
}
```

In the above *Fr3D* constructor call, the first argument is a conditional expression to check if the user has used the option of specifying an input file as a program argument in the command line. (This is not required, since the user can also use the *File* menu to open input files). Recall that this is similar to what we did in Sections 5.6 and 6.8. The second argument of the constructor call just mentioned creates an object of class *CvPainter*, which is listed below in the separate file *CvPainter.java*. The third argument specifies the window title *Painter* that will appear.

```
// CvPainter.java: Used in the file Painter.java.

// Copied from Section 7.3 of
//    Ammeraal, L. and K. Zhang (2007). Computer Graphics for Java
Programmers, 2nd Edition,
//       Chichester: John Wiley.

import java.awt.*;
import java.util.*;

class CvPainter extends Canvas3D
{  private int maxX, maxY, centerX, centerY;
   private Obj3D obj;
   private Point2D imgCenter;

   Obj3D getObj(){return obj;}
   void setObj(Obj3D obj){this.obj = obj;}
   int iX(float x){return Math.round(centerX + x - imgCenter.x);}
   int iY(float y){return Math.round(centerY - y + imgCenter.y);}
```

```
void sort(Tria[] tr, int[] colorCode, float[] zTr, int l, int r)
{  int i = l, j = r, wInt;
   float x = zTr[(i + j)/2], w;
   Tria wTria;
   do
   {  while (zTr[i] < x) i++;
      while (zTr[j] > x) j--;
      if (i < j)
      {  w = zTr[i]; zTr[i] = zTr[j]; zTr[j] = w;
         wTria = tr[i]; tr[i] = tr[j]; tr[j] = wTria;
         wInt = colorCode[i]; colorCode[i] = colorCode[j];
         colorCode[j] = wInt;
         i++; j--;
      }  else
      if (i == j) {i++; j--;}
   }  while (i <= j);
   if (l < j) sort(tr, colorCode, zTr, l, j);
   if (i < r) sort(tr, colorCode, zTr, i, r);
}

public void paint(Graphics g)
{  if (obj == null) return;
   Vector polyList = obj.getPolyList();
   if (polyList == null) return;
   int nFaces = polyList.size();
   if (nFaces == 0) return;

   Dimension dim = getSize();
   maxX = dim.width - 1; maxY = dim.height - 1;
   centerX = maxX/2; centerY = maxY/2;
   // ze-axis towards eye, so ze-coordinates of
   // object points are all negative.
   // obj is a java object that contains all data:
   // - Vector w        (world coordinates)
   // - Array e         (eye coordinates)
   // - Array vScr      (screen coordinates)
   // - Vector polyList (Polygon3D objects)

   // Every Polygon3D value contains:
   // - Array 'nrs' for vertex numbers
   // - Values a, b, c, h for the plane ax+by+cz=h.
   // - Array t (with nrs.length-2 elements of type Tria)

   // Every Tria value consists of the three vertex
   // numbers iA, iB and iC.
```

```
      obj.eyeAndScreen(dim);
            // Computation of eye and screen coordinates.

      imgCenter = obj.getImgCenter();
      obj.planeCoeff();    // Compute a, b, c and h.

      // Construct an array of triangles in
      // each polygon and count the total number
      // of triangles:
      int nTria = 0;
      for (int j=0; j<nFaces; j++)
      {  Polygon3D pol = (Polygon3D)(polyList.elementAt(j));
         if (pol.getNrs().length < 3 || pol.getH() >= 0)
            continue;
         pol.triangulate(obj);
         nTria += pol.getT().length;
      }
      Tria[] tr = new Tria[nTria];
      int[] colorCode = new int[nTria];
      float[] zTr = new float[nTria];
      int iTria = 0;
      Point3D[] e = obj.getE();
      Point2D[] vScr = obj.getVScr();

      for (int j=0; j<nFaces; j++)
      {  Polygon3D pol = (Polygon3D)(polyList.elementAt(j));
         if (pol.getNrs().length < 3 || pol.getH() >= 0) continue;
         int cCode =
            obj.colorCode(pol.getA(), pol.getB(), pol.getC());
         Tria[] t = pol.getT();
         for (int i=0; i< t.length; i++)
         {  Tria tri = t[i];
            tr[iTria] = tri;
            colorCode[iTria] = cCode;
            float zA = e[tri.iA].z, zB = e[tri.iB].z,
               zC = e[tri.iC].z;
            zTr[iTria++] = zA + zB + zC;
         }
      }

      sort(tr, colorCode, zTr, 0, nTria - 1);

      for (iTria=0; iTria<nTria; iTria++)
      {  Tria tri = tr[iTria];
         Point2D a = vScr[tri.iA],
```

```
                b = vScr[tri.iB],
                c = vScr[tri.iC];
        int cCode = colorCode[iTria];
        g.setColor(new Color(cCode, cCode, 0));
        int[] x = {iX(a.x), iX(b.x), iX(c.x)};
        int[] y = {iY(a.y), iY(b.y), iY(c.y)};
        g.fillPolygon(x, y, 3);
      }
   }
}
```

We use a special method, *sort*, to sort the triangles; it is based on the well-known and efficient quicksort algorithm, discussed in detail in Ammeraal (1996). Before calling *sort*, we build three arrays:

Array element	Type	Contains
tr[*iTria*]	*Tria*	the three vertex numbers of the triangle
colorCode[*iTria*]	*int*	value between 0 and 255
zTr[*iTria*]	*float*	value representing z_e-coordinate of triang

For a given subscript value *iTria*, the three array elements in the first column of this table belong to the same triangle. We may regard them as members of the same record, of which *zTr* is the key. During sorting, whenever two elements *zTr*[*i*] and *zTr*[*j*] are swapped, we swap *tr*[*i*] and *tr*[*j*] as well as*colorCode*[*i*] and *colorCode*[*j*] at the same time. It seems reasonable to use the average of the three z_e-coordinates of a triangle's vertices as the z_e-coordinate of that triangle, but we may as well simply use the sum instead of the average. Recall that the positive z_e-axis points towards us, so that all z_e values that we use are negative: the more negative it is, the further away the triangle is. It follows that we have to paint the triangles in increasing order of their z_e-coordinates, or, equivalently, in decreasing order of their absolute values.

As the class *CvPainter* in the program *Painter.java* shows, there is a call to*colorCode* in the method *paint* for every polygon just before entering the loop. Within the loop, all the triangles of that polygon, their color code, and each triangle's distance, are stored in the three arrays *tr*, *colorCode* and *zTr*, discussed

above. After this has been done for all polygons, the triangles are sorted on the basis of their distances stored in *zTr*, and then displayed in order of decreasing distance.

Next, we demonstrate the application of the painter's algorithm to some simple 3D objects, with the implementation of user operations. As Figure 7.4 shows, we will enable the user to change the viewpoint E, characterized by its spherical coordinates. Immediately after an input file has been opened, these coordinates have the following (default) values:

ρ = 3 × the distance between two opposite vertices of a box in which the objects fits

θ = 0.3 radians (≈ 17°)

φ = 1.3 radian (≈ 74°)

If the user gives the menu command *Viewpoint Down* the value of φ is increased by 0.1 radians (≈6°). Similarly, *Viewpoint Up* decreases φ by 0.1 radians, while *Viewpoint Left* and *Viewpoint Right* decrease and increase θ by 0.1 radians, respectively. The same effects can be achieved by using one of the four arrow keys, ↓, ↑, ← and →, together with the Ctrl-key, as indicated in the menu. Figure 7.4 shows the screen when the input file *letterL.dat*, discussed in Section 6.3, has been read by program *Painter.java* and the user has used the *Viewpoint Up* command four times, so that we view the object from a higher viewpoint than in the initial situation.

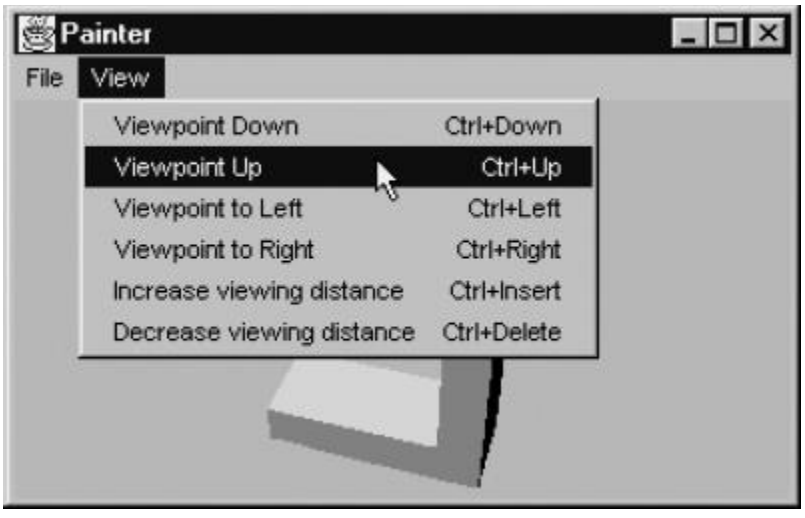

Figure 7.4. Program *Painter.java* applied to file *letterL.dat*

The *File* menu, not shown in Figure 7.4, consists of the following commands:

```
Open    Ctrl+O
Exit    Ctrl+Q
```

If several input files are available, the user can switch to another object by using the *Open* command (or Ctrl+O), as usual. The simplicity of our file format makes it easy to generate such files by other programs, as we will discuss in the next two sections. In particular, mathematically well-defined solids, such as the sphere and the cone of Figure 7.5, are very suitable for this. Incidentally, this example demonstrates that what we call a 'three-dimensional object' may consist of several solids. Note, however, that the input-file format requires that all vertices, in this example of both the sphere and the cone, must be defined in the first part of the file and all faces in the second. In other words, the line with the word *Faces* must occur only once in the file. This example also shows that curved surfaces can be approximated by a great many flat faces. In this example, there are altogether 1804 vertices and 1920 faces. Here the command *Viewpoint Down* has been used four times and the command*Viewpoint Right* twice.

An object will normally appear with a reasonable perspective effect, but the user will be able to increase or decrease the viewing distance by using menu commands or their shortcuts, Ctrl+Insert and Ctrl+Delete, as Figure 7.4 shows.

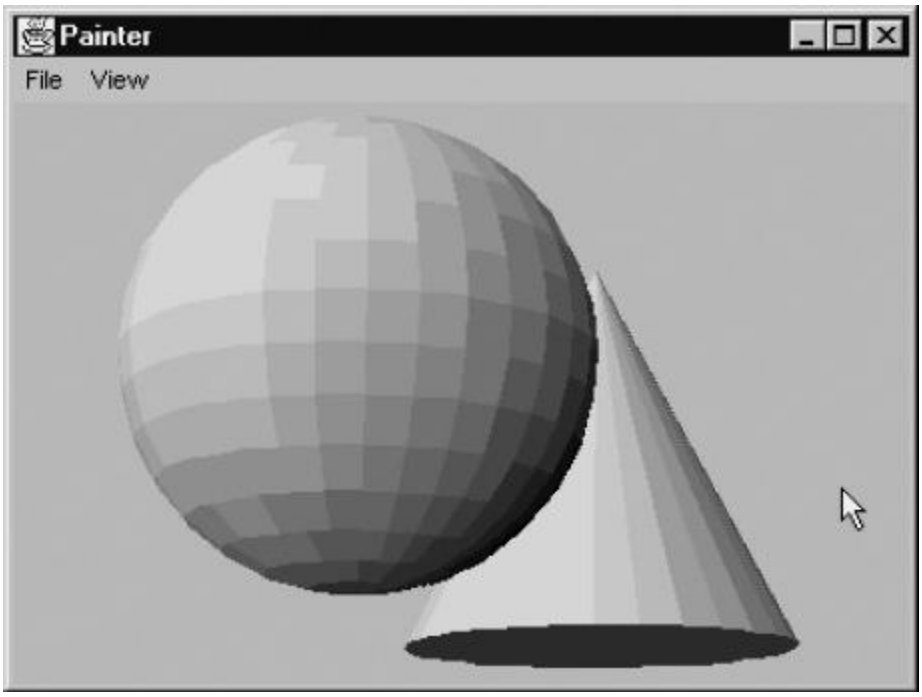

Figure 7.5. Sphere and cone

Although we can see only shades of gray in Figures 7.4 and 7.5, the situation looks much more interesting on the screen: the background is light blue like the sky and all faces of the object are yellow. The source of light is far away, at the top left, so that the upper and the left faces are bright yellow, while those on the right and on the bottom are much darker.

To start the program, we can supply the (initial) input file as a program argument, if we like. For example, if the first 3D object to be displayed is given by the file *letterL.dat*, we can start the program by entering

```
java Painter letterL.dat
```

Alternatively, we can enter

```
java Painter
```

and use the *Open* command in the *File* menu (or Ctrl+O) to specify the input file *letterL.dat*. With both ways of starting the program, we can switch to another object by using this *Open* command.

As discussed at the beginning of this section, the painter's algorithm will fail in some cases, such as the three beams of Figure 7.3. We will therefore discuss another algorithm for hidden-face elimination in the next section.

Z-BUFFER ALGORITHM

If the images of two faces F_1 and F_2 overlap, we may consider two questions with regard to the correct representation of these faces:

1. Which pixels would be used for both F_1 and F_2 if each face were completely visible?
2. For each of these pixels, which of the corresponding points in F_1 and F_2 is nearer?

The Z-buffer algorithm deals with these questions in a general and elegant way. Recall that we are using an eye-coordinate system, with *z*-coordinates denoting the distance we are interested in. We consider points P in 3D space and their corresponding projections P' in 2D space, where we use central projection with the viewpoint E as the center of projection. In other words, each line PE is a ray of

light, intersecting the screen in P'. We are especially interested in such points P' that are the centers of pixels.

The Z-buffer algorithm is based on a large two-dimensional array in which we store numbers that represent *z*-coordinates. Using the variable *dim* as the current canvas dimension, we define the following array, also known as a *Z-buffer*, for this purpose:

```
private float buf[][];
buf = new float[dim.width][dim.height];
```

We initialize array *buf* with values corresponding to points that are very far away. As before, we ignore back faces. For each of the remaining faces we compute all pixels and the *z*-values that correspond with these. For each pixel P'(*ix*, *iy*), we test whether the corresponding point P in 3D space is nearer than*buf*[*ix*][*iy*] indicates. If it is, we put this pixel on the screen, using the color for the face in question, computed as in the previous section, while, at the same time, updating *buf*[*ix*][*iy*]:

```
For each face F (and its image, consisting of a set
of pixels):
  For each pixel P′(ix, iy), corresponding with a 3D
point P of F:
    If P is nearer than the distance stored in buf
[ix][iy],
    {   set pixel P′ to the color for face F
        update buf [ix][iy] so that it refers to the
distance of P
    }
```

In this discussion, the words *distance* and *near* refer to the *z*-coordinates in the eye-coordinate system. (Since we use no other 3D coordinates here, we simply write *z* instead of z_e here.) There are two aspects that make the implementation of the above algorithm a bit tricky:

1. Since the *z*-axis points towards us, the larger *z* is, the shorter the distance.
2. It is necessary to use 1/*z* instead of *z* for linear interpolation.

Let us take a look at this rather surprising point 2. Suppose we are given the z_e-coordinates of two points A and B in 3D space and the central projections A′ and B′ of these points on the screen. Besides, some point P′ on the screen, lying on A′*B*′, is given and we have

$$x_P' = x_A' + \lambda(x_B' - x_A')$$

$$y_P' = y_A' + \lambda(y_B' - y_A')$$

where x_P', and so on, are screen coordinates. We are then interested in the point P (in 3D space) of which P′ is the central projection. Since we want to know how far away P is, our goal is to compute z_P. (After this, we can also compute the 3D coordinates $x_P = -x_P' z_P/d$ and $y_P = -y_P' z_P/d$, using Equations (5.7) and (5.8) of Section 5.3, where we wrote X and Y instead of x' and y'.) Curiously enough, to compute this eye coordinate z_P by interpolation, we need to use the inverse values of the z-coordinates:

$$\frac{1}{z_P} = \frac{1}{z_A} + \lambda\left(\frac{1}{z_B} - \frac{1}{z_A}\right)$$

We will simply use this result here; it is discussed in more detail in Appendix A. Let us write

$$z_{Pi} = \frac{1}{z_P}$$

which we write as *zPi* (equal to 1/*zP*) in the program. Using the same convention ($z_{Ai} = 1/zA$, etc.) for other variables and writing *xA*, *yA* etc. for screen coordinates, we compute the centroid D(x_D, y_D) along with its inverse z-value z_{Di} for each triangle ABC as follows:

```
xD = (xA + xB + xC)/3;
yD = (yA + yB + yC)/3;
zDi = (zAi + zBi + zCi)/3;
```

This centroid will be the basis for computing z_i-values for other points of the triangle by linear interpolation. To do this, we are interested in how much z_{Pi} increases if P moves one pixel to the right. This quantity, which we may write as $\partial z_i/\partial x$ or simply as *dzdx* in the program, is constant for the whole triangle. We will use this value, as well as its counterpart *dzdy* = $\partial z_i/\{\partial y$ indicating how much z_{Pi} increases if P moves one pixel upward, that is, if the screen coordinate y_P is increased by 1. It is useful to think of the triples x, y and z_i (where x and y are screen coordinates and $z_i = 1/z$) as points in a plane of an imaginary 3D space. We can then denote this plane as

Equation 7.1.

$$ax + by + cz_i = k$$

Writing this in the form $z_i = (k - ax - by)/c$ and applying partial differentiation to z_i, we obtain

Equation 7.2.

$$\frac{\partial z_i}{\partial x} = -\frac{a}{c}$$

Equation 7.3.

$$\frac{\partial z_i}{\partial y} = -\frac{b}{c}$$

To find *a*, *b* and *c*, remember that (*a*, *b*, *c*) in Equation (7.1) is the normal vector of the plane of triangle ABC (in the imaginary space we are dealing with). We define the vectors

u = **AB** = (*u*1, *u*2, *u*3)

v = **AC** = (*v*1, *v*2, *v*3)

where

$u_1 = x_B - x_A$	$v_1 = x_C - x_A$
$u_2 = y_B - y_A$	$v_2 = y_C - y_A$
$u_3 = z_{Bi} - z_{Ai}$	$v_3 = z_{Ci} - z_{Ai}$

Then the vector product

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix}$$

(see Section 2.4) is also perpendicular to triangle ABC, so that we can compute the desired values *a*, *b* and *c* of (7.1), (7.2) and (7.3) as the coefficients of **i**,**j** and**k**, respectively, finding

$a = u_2v_3 - u_3v_2$

$b = u_3v_1 - u_1v_3$

$c = u_1v_2 - u_2v_1$

So much for the computation of $dzdx = \partial z_i/\partial x$ and $dzdy = \partial z_i/\partial y$, which we will use to compute z_{Pi} for each point P of triangle ABC, as we will see below.

To prepare for traversing all relevant *scan lines* for triangle ABC, such as LR inFigure 7.6, we compute the *y*-coordinates *yTop* and *yBottom* of the vertices at the top and the bottom of this triangle. For all pixels that comprise triangle ABC on the screen, we have to compute the *z*-coordinate of the corresponding point in 3D space. In view of the enormous number of those pixels, we should do this as efficiently as possible. Working from the bottom of the triangle to the top, when dealing with a scan line, we traverse all its pixels from left to right.Figure 7.6 shows a situation in which all pixels of triangle ABC below the scan line LR have already been dealt with.

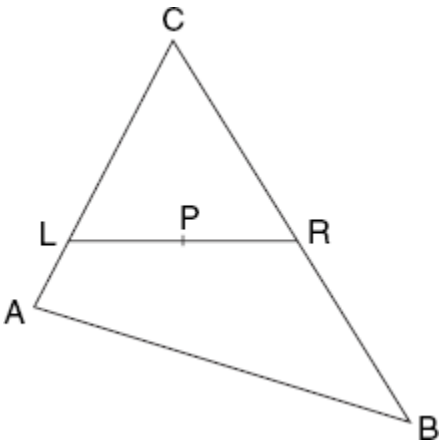

Figure 7.6. Triangle ABC and scan line LR

For each scan line at level *y* (*yBottom* ≤ *y* ≤*yTop*), we find the points of intersection L and R with triangle ABC as follows. We introduce the points I, J and K, which are associated with the triangle edges BC, CA and AB, respectively. Initially, we set the program variables *xI*, *xJ* and *xK* to 10^{30}, and*xI*1, *xJ*1 and *xK*1 to -10^{30}. Then,

if *y* lies between *yB* and *yC* or is equal to one of these values, we compute the point of intersection of the scan line with BC, and we assign the *x*-coordinate of this point to both *xI* and *xI*1. In the same way we possibly update *xJ*, *xJ*1 for CA and *xK* and *xK*1 for AB. After this, each of the variables *xI*, *xJ* and *xK* is equal either to its original value 10^{30} or to the *x*-coordinate of the scan line in question with BC, CA and AB, respectively. The same applies to the other three variables, except that these may still have a different original value, -10^{30}. We can now easily find *xL* and *xR*:

$xL = \min(xI, xJ, xK)$

$xR = \max(xI1, xJ1, xK1)$

So far, we have been using floating-point, logical coordinates *y*, *xL* and *xR*. Since we have to deal with pixels, we convert these to the (integer) device coordinates *iY*, *iXL* and *iXR* as follows:

```
int iy = iY(y), iXL = iX(xL+0.5), iXR = iX(xR-0.5);
```

By adding 0.5 to *xL* and subtracting 0.5 from *xR*, we avoid clashes between neighboring triangles of different colors: the pixel (*iXR*, *y*) belonging to triangle ABC should preferably not also occur as a pixel (*iXL*, *y*) of the right-hand neighbor of this triangle, because it would then not be clear which color to use for this pixel. Before entering the loop for all pixels *iXL*, *iXL* + 1, . . ., *iXR*for the scan line on level *y*, we compute the inverse *z* value *zi* = *zLi* for the pixel (*iXL*, *y*). Theoretically, this value is

$$z_{\mathrm{Li}} = z_{\mathrm{Di}} + (y - y_{\mathrm{D}})\frac{\partial z}{\partial y} + (x_{\mathrm{L}} - x_{\mathrm{D}})\frac{\partial z}{\partial x}$$

In the program we modify this a little, giving a little more weight to the centroid D of the triangle:

```
double zi = 1.01 * zDi + (y - yD) * dzdy + (xL - xD) *
dzdx;
```

This modification is useful in some special cases of which we will give an example at the end of this section.

Starting at the left end (*iXL*, *y*) of a scan line with the above *z* value *zi* = z_{Li}, we could now write the loop for this scan line as follows:

```
for (int x=iXL; x<=iXR; x++)
{  if (zi < buf[x][iy])    // '<' means '\ammenearer'
   {  g.drawLine(x, iy, x, iy);
      buf[x][iy] = (float)zi;
   }
   zi += dzdx;
}
```

Along a horizontal line LR, shown in Figure 7.6, we compute the inverse *z*-coordinate, *zi*. If this is less than the contents of the array element *buf*[*x*][*y*], we put a pixel on the screen and update that array element.

The above test *zi* < *buf*[*x*][*y*] may at first look confusing. Since the positive *z*-axis of the eye-coordinate system points towards us, we have:

- the *larger* the *z*-coordinate of a point, the nearer this point is to the eye.

  However, we are using inverse values *zi* = 1/*z*, so that the above is equivalent to

- the *smaller* the *zi*-value of a point, the nearer it is to the eye.

A complicating factor is that we are using negative *z*-coordinates, but the above also applies to negative numbers. The following example for two points P and Q will make the situation clear:

P nearby		Q far away
$z_P = -10$	>	$z_Q = -20$
$zi_P = 1/z_P = -0.1$	<	$zi_Q = 1/z_Q = -0.05$

Another curious aspect of the above fragment is that putting a pixel on the screen is done here by drawing a line of only one pixel. It is strange that Java does not supply a more elementary routine, say, *putPixel*, for this purpose. However, we can do much better by delaying this '*putPixel*' operation until we know for how many adjacent pixels it is to be used; in other words, we build horizontal line segments in memory, storing their leftmost *x* values and displaying these segments if we can no longer extend it on the right. This implies that even if there were a *putPixel* method, we would not use it, but rather draw horizontal line segments, consisting of some pixels we have recently been dealing with. Instead of the above

for-loop we will actually use an 'optimized' fragment which is equivalent to it, as we will discuss in a moment. The program *ZBuf.java* is listed below.

```
// ZBuf.java: Perspective drawing using an input file that
//    lists vertices and faces.
//    Z-buffer algorithm used for hidden-face elimination.

// Copied from Section 7.4 of
//    Ammeraal, L. and K. Zhang (2007). Computer Graphics for Java
Programmers, 2nd Edition,
//       Chichester: John Wiley.

// Uses: CvZBuf (Section 7.4),
//       Point2D (Section 1.5), Point3D (Section 3.9) and
//       Obj3D, Polygon3D, Tria, Fr3D, Canvas3D (Section 5.5).
import java.awt.*;

public class ZBuf extends Frame
{  public static void main(String[] args)
   {  new Fr3D(args.length > 0 ? args[0] : null, new CvZBuf(),
         "ZBuf");
   }
}
```

The class *CvZBuf* is defined in the following separate file:

```
// CvZBuf.java: Canvas class for ZBuf.java.

// Copied from Section 7.4 of
//    Ammeraal, L. and K. Zhang (2007). Computer Graphics for Java
Programmers, 2nd Edition,
//       Chichester: John Wiley.

import java.awt.*;
import java.util.*;

class CvZBuf extends Canvas3D
{  private int maxX, maxY, centerX, centerY, maxX0 = -1, maxY0 = -1;
   private float buf[][];
   private Obj3D obj;
   private Point2D imgCenter;

   int iX(float x){return Math.round(centerX + x - imgCenter.x);}
   int iY(float y){return Math.round(centerY - y + imgCenter.y);}
```

```
Obj3D getObj(){return obj;}
void setObj(Obj3D obj){this.obj = obj;}

public void paint(Graphics g)
{  if (obj == null) return;
   Vector polyList = obj.getPolyList();
   if (polyList == null) return;
   int nFaces = polyList.size();
   if (nFaces == 0) return;
   float xe, ye, ze;

   Dimension dim = getSize();
   maxX = dim.width - 1; maxY = dim.height - 1;
   centerX = maxX/2; centerY = maxY/2;
   // ze-axis towards eye, so ze-coordinates of
   // object points are all negative. Since screen
   // coordinates x and y are used to interpolate for
   // the z-direction, we have to deal with 1/z instead
   // of z. With negative z, a small value of 1/z means
   // a small value of |z| for a nearby point. We there-
   // fore begin with large buffer values 1e30:
   if (maxX != maxX0 || maxY != maxY0)
   {  buf = new float[dim.width][dim.height];
      maxX0 = maxX; maxY0 = maxY;
   }
   for (int iy=0; iy<dim.height; iy++)
      for (int ix=0; ix<dim.width; ix++)buf[ix][iy] = 1e30F;

   obj.eyeAndScreen(dim);
   imgCenter = obj.getImgCenter();
   obj.planeCoeff();    // Compute a, b, c and h.
   Point3D[] e = obj.getE();
   Point2D[] vScr = obj.getVScr();

   for (int j=0; j<nFaces; j++)
   {  Polygon3D pol = (Polygon3D)(polyList.elementAt(j));
      if (pol.getNrs().length < 3 || pol.getH() >= 0)
         continue;
      int cCode =
         obj.colorCode(pol.getA(), pol.getB(), pol.getC());
      g.setColor(new Color(cCode, cCode, 0));

      pol.triangulate(obj);

      Tria[] t = pol.getT();
```

```
        for (int i=0; i<t.length; i++)
        {  Tria tri = t[i];
           int iA = tri.iA, iB = tri.iB, iC = tri.iC;
           Point2D a = vScr[iA], b = vScr[iB], c = vScr[iC];
           double zAi = 1/e[tri.iA].z, zBi = 1/e[tri.iB].z,
               zCi = 1/e[tri.iC].z;
           // We now compute the coefficients a, b and c
           // (written here as aa, bb and cc)
           // of the imaginary plane ax + by + czi = h,
           // where zi is 1/z (and x, y and z are
           // eye coordinates. Then we compute
           // the partial derivatives dzdx and dzdy:
           double u1 = b.x - a.x, v1 = c.x - a.x,
                  u2 = b.y - a.y, v2 = c.y - a.y,
                  cc = u1 * v2 - u2 * v1;
           if (cc <= 0) continue;
           double xA = a.x, yA = a.y,
                  xB = b.x, yB = b.y,
                  xC = c.x, yC = c.y,
                  xD = (xA + xB + xC)/3,
                  yD = (yA + yB + yC)/3,
                  zDi = (zAi + zBi + zCi)/3,
                  u3 = zBi - zAi, v3 = zCi - zAi,
                  aa = u2 * v3 - u3 * v2,
                  bb = u3 * v1 - u1 * v3,
                  dzdx = -aa/cc, dzdy = -bb/cc,
                  yBottomR = Math.min(yA, Math.min(yB, yC)),
                  yTopR = Math.max(yA, Math.max(yB, yC));
           int yBottom = (int)Math.ceil(yBottomR),
                  yTop = (int)Math.floor(yTopR);

           for (int y=yBottom; y<=yTop; y++)
           {  // Compute horizontal line segment (xL, xR)
              // for coordinate y:
              double xI, xJ, xK, xI1, xJ1, xK1, xL, xR;
              xI = xJ = xK = 1e30;
              xI1 = xJ1 = xK1 = -1e30;
              if ((y - yB) * (y - yC) <= 0 && yB != yC)
                 xI = xI1 = xC + (y - yC)/(yB - yC) * (xB - xC);
              if ((y - yC) * (y - yA) <= 0 && yC != yA)
                 xJ = xJ1 = xA + (y - yA)/(yC - yA) * (xC - xA);
              if ((y - yA) * (y - yB) <= 0 && yA != yB)
                 xK = xK1 = xB + (y - yB)/(yA - yB) * (xA - xB);
              // xL = xR = xI;
              xL = Math.min(xI, Math.min(xJ, xK));
```

```
                xR = Math.max(xI1, Math.max(xJ1, xK1));
                int iy = iY((float)y), iXL = iX((float)(xL+0.5)),
                   iXR = iX((float)(xR-0.5));
                double zi =  1.01 * zDi + (y - yD) * dzdy +
                                          (xL - xD) * dzdx;
             /*
                for (int x=iXL; x<=iXR; x++)
                {  if (zi < buf[x][iy]) // < is nearer
                   {  g.drawLine(x, iy, x, iy);
                      buf[x][iy] = (float)zi;
                   }
                   zi += dzdx;
                }
             */
             // The above comment fragment is optimized below:
             // ---
                boolean leftmostValid = false;
                int xLeftmost = 0;
                for (int ix=iXL; ix<=iXR; ix++)
                {  if (zi < buf[ix][iy])  // < means nearer
                   {  if (!leftmostValid)
                      {  xLeftmost = ix;
                         leftmostValid = true;
                      }
                      buf[ix][iy] = (float)zi;
                   }
                   else
                   if (leftmostValid)
                   {  g.drawLine(xLeftmost, iy, ix-1, iy);
                      leftmostValid = false;
                   }
                   zi += dzdx;
                }
                if (leftmostValid)
                   g.drawLine(xLeftmost, iy, iXR, iy);
             // ---
             }
          }
       }
    }
}
```

Almost at the end of this program, the fragment between the two lines // --- is a more efficient version than the for-loop in the comment that precedes it. In that for-loop, each line LR is drawn pixel by pixel. In the more efficient version that

follows, however, the line LR is drawn by one or more line segments depending on how much the line LR is blocked by the triangles that are nearer. In the best case, where no triangle is in front of the line, the line is drawn by a simple call to the *drawLine* method. Scanning through each horizontal line from left to right, we use a Boolean variable *leftmostValid* to record if we have found a leftmost point that is not blocked by any other triangle. If so, we keep scanning and updating the Z-buffer until we encounter the first point that is further away than the corresponding Z-buffer value, such as *I* in Figure 7.7. We then draw the line up to the point just before that blocked point (*I*). We meanwhile set *leftmostValid* to false, entering the line segment, such as *IJ* inFigure 7.7. Continuing scanning through until the first visible point, such as *J*, is found, we record *J* as the new leftmost point and set *leftmostValid* to true. This process continues to reach R, and the last line segment, such as *JR*, is drawn if it is not blocked.

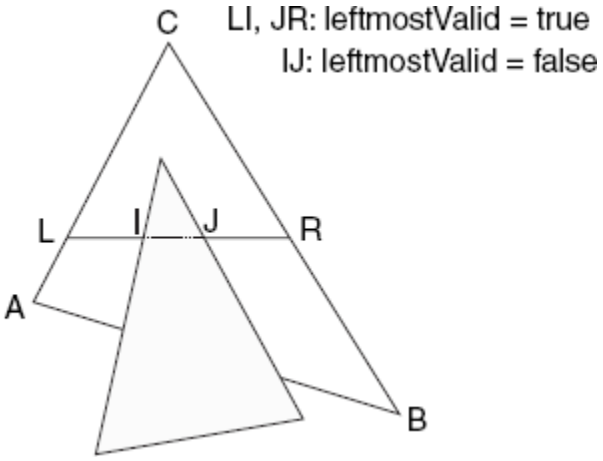

Figure 7.7. Illustration of more efficient version

Figure 7.8 demonstrates that the Z-buffer algorithm can also be used in cases in which the painter's algorithm fails. The latter is illustrated by Figure 7.9. (We have also seen these three beams in Figures 6.25 and 7.3.)

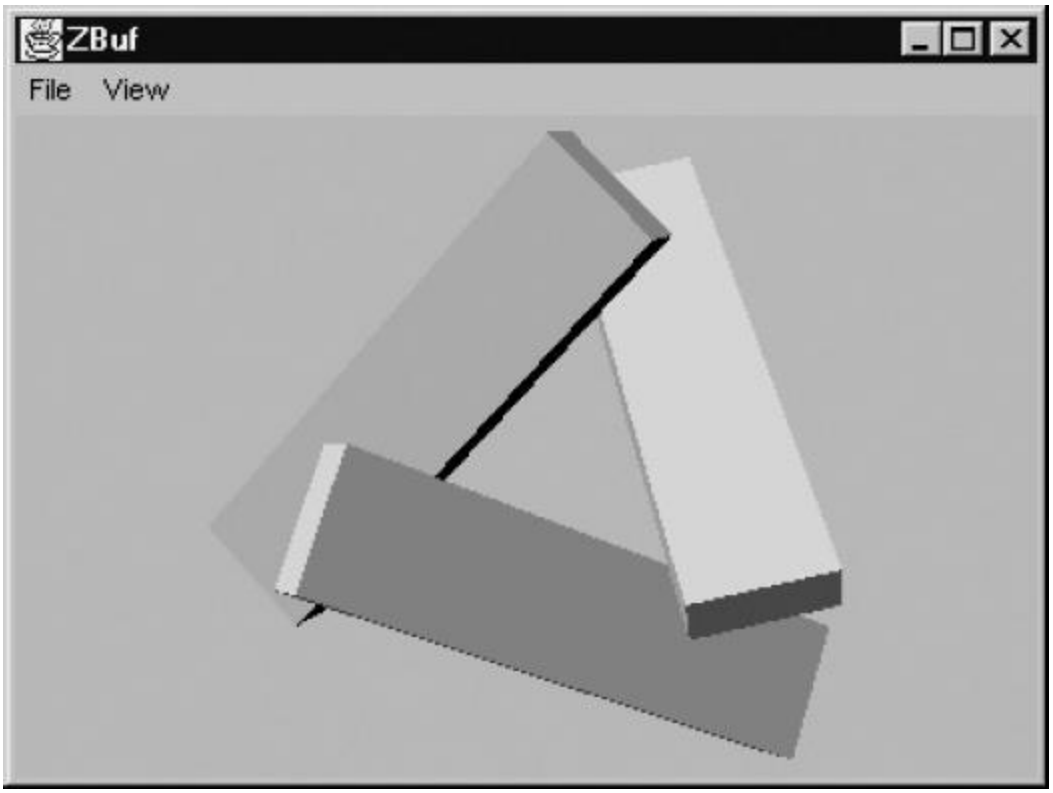

Figure 7.8. Z-buffer algorithm applied to three beams

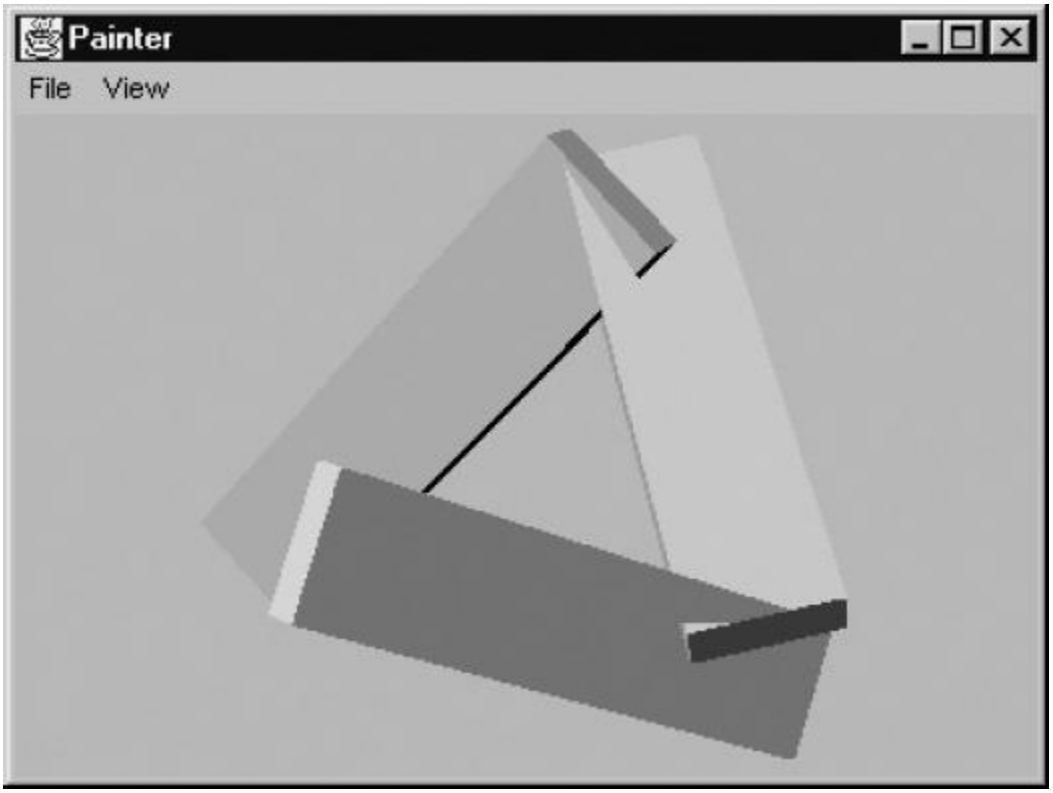

Figure 7.9. Painter's algorithm failing

In Appendix E (Section E.5) we will develop a program that enables us to display the surfaces of functions of two variables. Files generated by this program are also accepted by the program *ZBuf.java*, as Figure 7.10demonstrates.

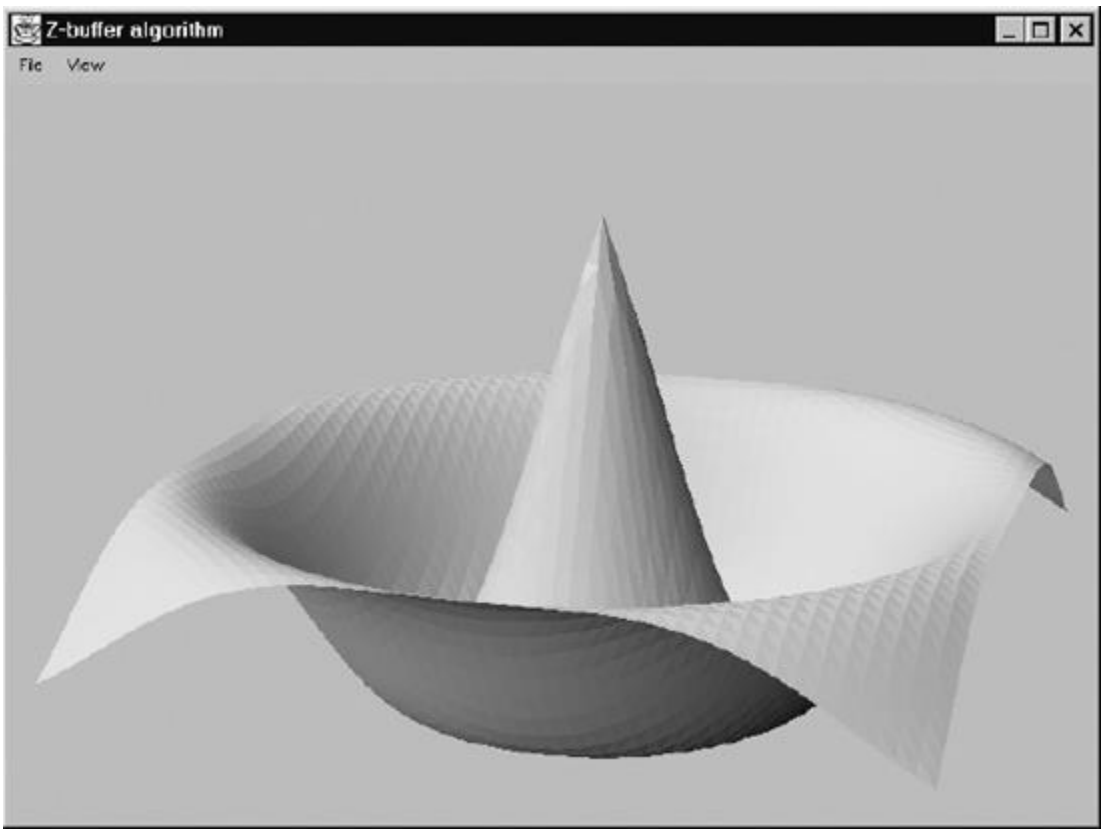

Figure 7.10. Function of two variables

Here we are using faces with two sides, as discussed in Section 6.5. The way these are used here requires a correction in the computation of *zi*. Recall that we have introduced the factor 1.01 in the following statement:

```
double zi =  1.01 * zDi + (y - yD) * dzdy + (xL - xD) *
dzdx;
```

Without this factor, some incorrect dark pixels would appear on the boundary, also referred to as the *silhouette*, of the (yellow) object and the (light blue) background. To understand this, we note that in Figure 7.11 point P lies on the boundary of the triangles T_1 and T_2.

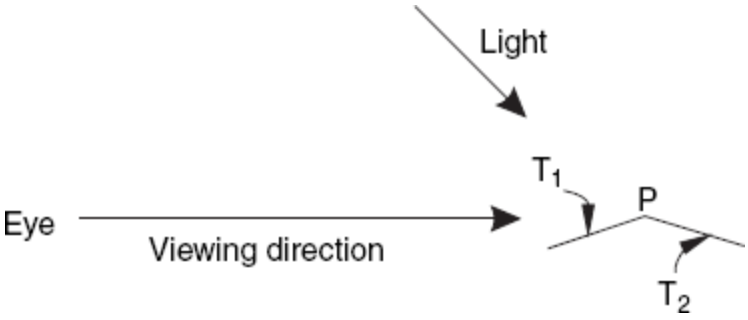

Figure 7.11. Point P belonging to triangles of very different colors

The nearer triangle, T_1, is visible and its color is bright yellow because it lies on the upper side of the surface. Triangle T_2 lies on the lower side and would appear almost black on the screen if it were not obscured by T_1. Point P, belonging to both triangles, is used twice to determine the color of the corresponding pixel on the basis of the *zi* value of P. In both cases, this value is the same or almost the same, so that it is not clear which color will be used for this pixel. To solve this problem, we use the factor 1.01 instead of 1 in the above computation of *zi*. Because of this, the *zi* value for P when taken as a point of T_1will now be slightly less than when P is regarded as a point of T_2. As a result, the light yellow color will be used for the pixel in question.

EXERCISES

7.1 The problem of back-face culling cannot generally be solved by testing whether the z_e-component of the normal vector of a face in question is positive or negative. Recall that we are using the equation

$$ax + by + cz = h$$

for the plane of each face, in which $\mathbf{n} = (a, b, c)$ is the normal vector of that face, pointing outward. Using also the vector $\mathbf{x} = (x, y, z)$, we can write the above equation as

$$\mathbf{n} \cdot \mathbf{x} = h$$

Figure 7.12 shows the geometrical interpretation of **n**, **x** and *h*. For a visible face, the inner product *h* is negative because **n** and **x**point in opposite directions. By contrast, *h* is positive for a back face.

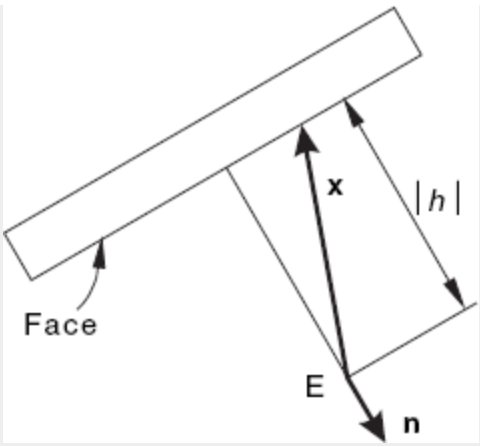

Figure 7.12. Geometrical interpretation of x, n and *h*, where *h* is negative

Although in most cases *c*, the third component of the vector **n**, is positive for a visible face and negative for a back face, there are situations in which this is not true. Give an example of such a situation, with a detailed explanation. How does back-face culling based on the signs of *h* and (incorrectly) of *c* relate to that based on the orientation of point sequences?

7.2 Apply animation with double buffering to a cube, as in Exercise 5.4, but use colored faces.

7.3 Apply animation to two colored cubes (see Exercises 5.5 and 7.2), but use colored faces. Even if they partly hide each other, this problem can be solved by back-face culling, provided the farther cube is drawn before the nearer one. This is demonstrated in Figure 7.13; here the *Graphics* methods *fillPolygon* and *drawPolygon* are applied to each visible face. By drawing the edges of such faces in black, we make sure that we can clearly distinguish the colored faces if shades of gray are used in black-and-white reproductions, such as Figure 7.13.

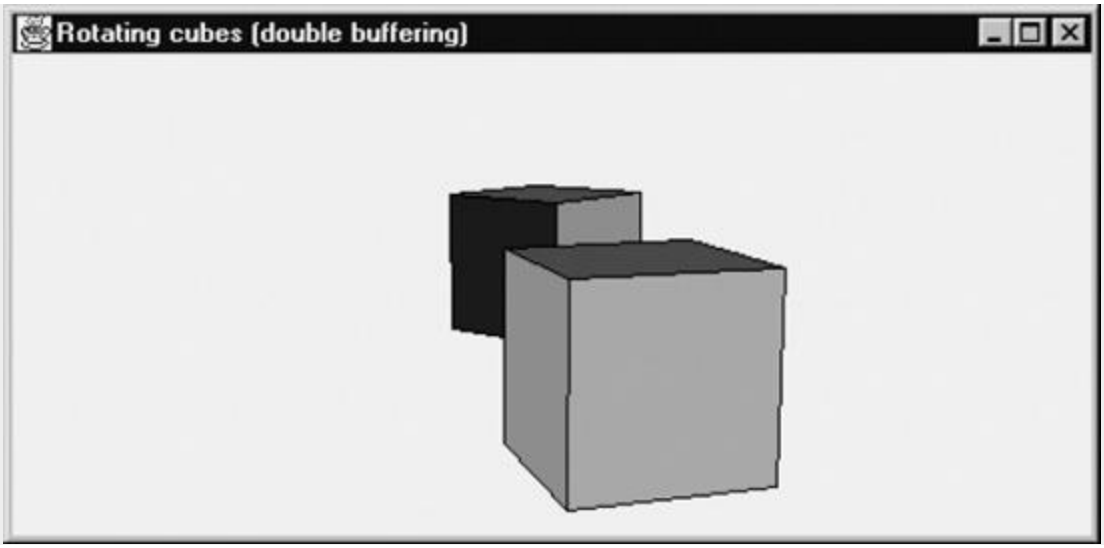

Figure 7.13. Two rotating cubes, generated by back-face culling

7.4 In the program file *Obj3D.java*, the direction of the light vector (*sunX*, *sunY*, *sunZ*) pointing to the source of light was arbitrarily chosen. Even without moving the object (by rotating it, for example), we can obtain an exciting effect by using animation to change this vector, so that the source of light rotates. Extend the program *Painter.java* to realize this. An easy way of doing this is by using spherical coordinates *sunTheta* and *sunPhi* and radius 1 for a light vector of unit length, similar to the spherical coordinates θ, $\varphi$and ρ, shown in Figure 5.1, and increasing, say, *sunTheta* by 0.02 radians every 50 ms.

7.5 Write a 3D data file (using a text editor) for the steps shown inFigure 7.14. Apply the programs *Painter.java* and *ZBuf.java* to it. Can you write a program to generate such data files? In such a program, the number of steps and the dimensions should preferably be variables.

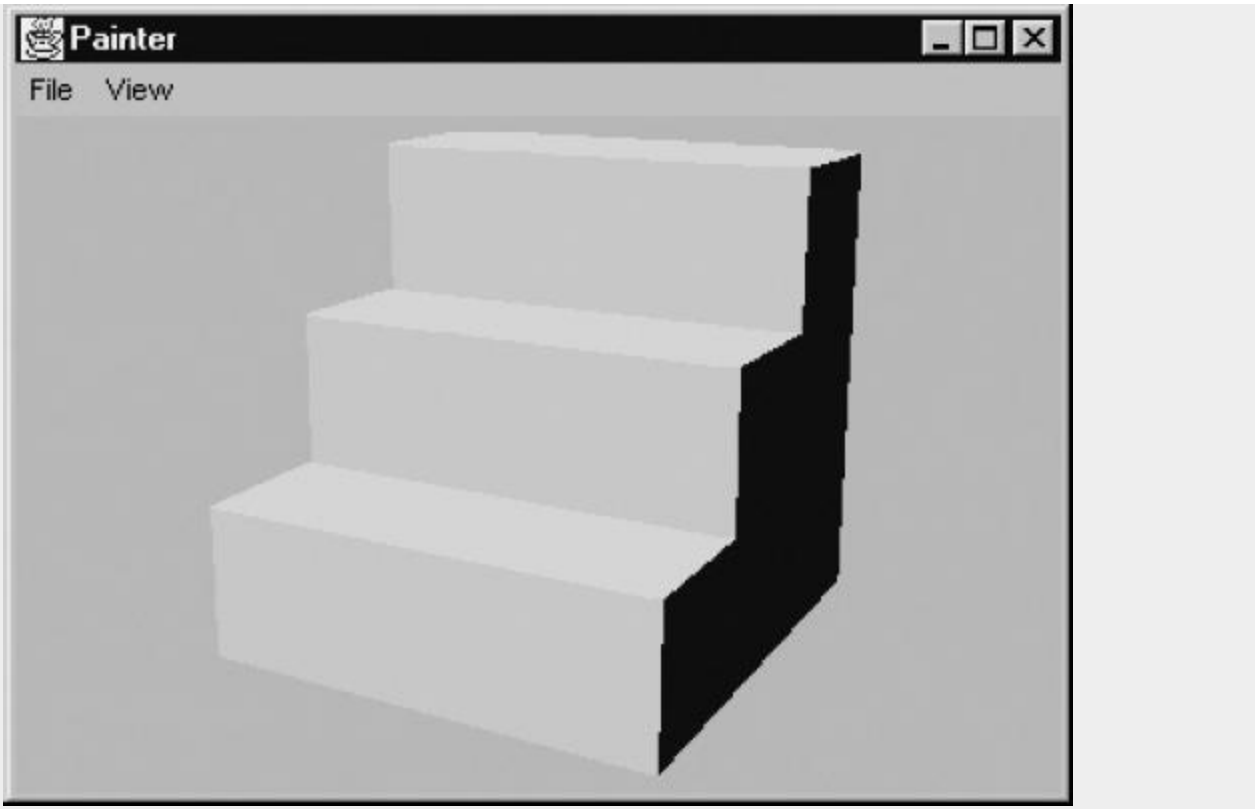

Figure 7.14. Steps