Radiosity and Non-photorealistic Rendering

REYNALD AFFELDT*

February 15, 2001

Introduction

For this report, I wrote two Java applications that implement basic algorithms for radiosity and edges detection, as proposed at:
`http://nis-lab.is.s.u-tokyo.ac.jp/~ nis/junk/VisRep01.html`.

For lack of time, they are very simple and unoptimized programs. However, I do believe they are good-starting points for more serious implementations. I personally plan to use them as test programs for performance measurements in the context of my own research [1].

The source codes are respectively:

- `http://web.yl.is.s.u-tokyo.ac.jp/~ affeldt/study/RadiosityApp.java`, and
- `http://web.yl.is.s.u-tokyo.ac.jp/~ affeldt/study/MapApp.java`,

the second file depending on the first one. They are also printed in Sections 3 and 4.

1 Radiosity Test Program

The first program implements the radiosity method of volume rendering. A scene is input as an array of squares which are further decomposed into small square patches. Using the geometric properties of the scene, a matrix of form factors is computer by means of the hemicube algorithm. This is the slowest part of the application. The result is used to write down the radiosity equation:

$$\begin{bmatrix} 1-\rho_0 F_{0,0} & -\rho_0 F_{0,1} & \cdots & -\rho_0 F_{0,n} \\ -\rho_1 F_{1,0} & 1-\rho_1 F_{1,1} & \cdots & -\rho_1 F_{1,n} \\ \vdots & & & \vdots \\ -\rho_{n-1} F_{n-1,0} & -\rho_{n-1} Fn-1,1 & \cdots & -\rho_{n-1} F_{n-1,n-1} \end{bmatrix} \begin{bmatrix} B_0 \\ \vdots \\ \vdots \\ B_{n-1} \end{bmatrix} = \begin{bmatrix} E_0 \\ \vdots \\ \vdots \\ E_{n-1} \end{bmatrix}$$

where ρ, B, and E are respectively the reflectivities, the radiosities, and the emittances of the objects in the scene. The problem then amounts to solve the previous system by the Gauss-Seidel iterative method. The color of each patch is eventually updated with the final results. Finally, each patch is decomposed into two triangles and the whole scene is then rendered by a zbuffer algorithm to obtain results like in Figure 1. This example shows a room with red, blue, and grey walls, with a green ground, and a white ceiling. There is one light patch in the ceiling, and different objects inside the room (a small wall and a small cube).

Though it seems to render correctly the scene, the application suffers from some minor problems. The zbuffer algorithm should be made smoother to cope more elegantly with jagged edges and holes. The form factor calculation also takes most of the execution time and should be alleviated.

2 Edges Detection Test Program

The input of both programs is a cross-like shape (Figure 2).

*96393, `affeldt@yl.is.s.u-tokyo.ac.jp`

[1] Which is about partial evaluation of Java bytecodes

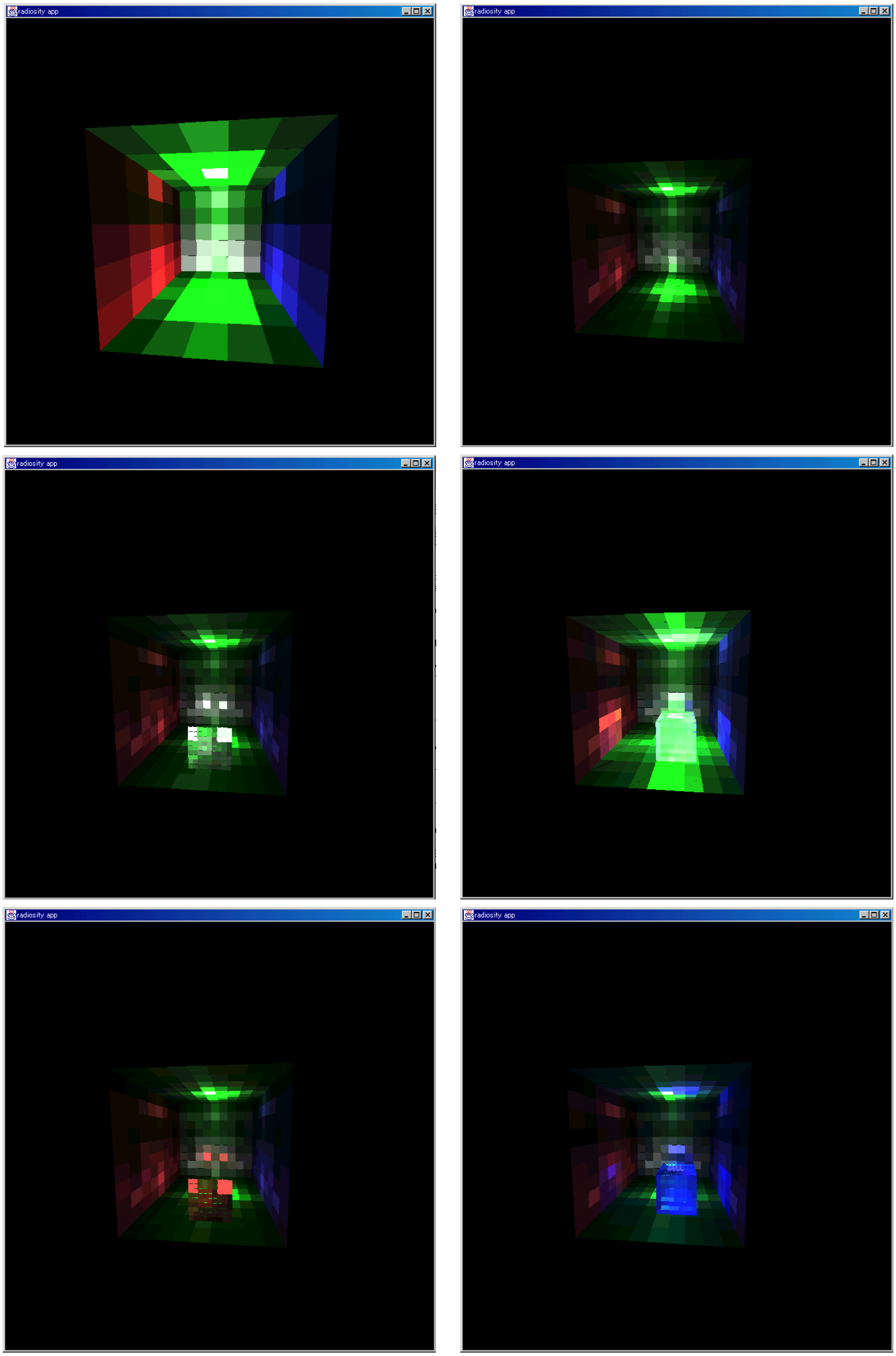

Figure 1: Radiosity Test Program

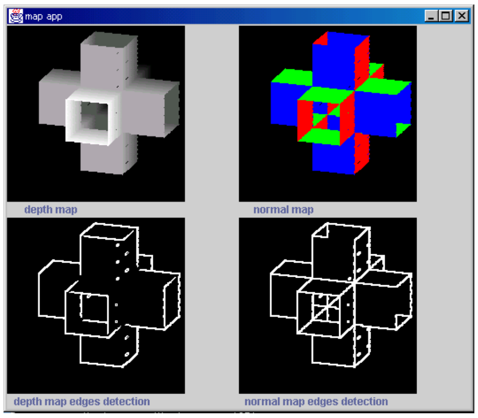

Figure 2: Edges Detection Test Program

2.1 Depth Map Edges Detection

We apply a zbuffer algorithm to the input scene and color it so as to output its depth map (a grey scale view of the scene where the level of grey of a point is proportional to its depth). Let $I(x, y)$ be the function that returns the level of grey of a point in the view plane. The procedure that decides whether or not an edge is detected ($\text{Edge}(x, y)$) compares with a threshold (T) the magnitude of the derivative (I_{mag}) which merely is the square root of two convolutions with the Sobel kernels (S_x and S_y). See equations below:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$I_x(x, y) = I(x, y) \otimes S_x \quad I_y(x, y) = I(x, y) \otimes S_y$$

$$I_{\text{mag}}(x, y) = \sqrt{I_x^2(x, y) + I_y^2(x, y)}$$

$$\text{Edge}(x, y) = \begin{cases} 1 & I_{\text{mag}}(x, y) \geq T \\ 0 & I_{\text{mag}}(x, y) < T \end{cases}$$

This is simply implemented as a double loop over the depth map. The problem with that method is that it doesn't enable detection of edges between objects that are at the same depth.

2.2 Normal Map Edges Detection

Prior to further processing, the input scene is colored so as to make its normal map appear. It consists in coloring each object in proportion to the dot product of the surface normal with the light's axis (red for the x axis, green the y axis, and blue for the z axis). Edges detection is then implemented as a double loop over the normal map that colors a point whenever adjacent point are of a different color.

As this application reuse the radiosity application zbuffer, it suffers from the same problems.

References

[1] Aaron Hertzmann, *Introduction to 3D Non-photorealistic Rendering: Silhouettes and Outlines.*

[2] Tomoyuki Nishita's Homepage, `http://nis-lab.is.s.u-tokyo.ac.jp/~ nis/`.

3 Radiosity Application Source Code

Radiosity.java

```
import javax.swing.JFrame;
import javax.swing.JPanel;

import java.lang.ArrayIndexOutOfBoundsException;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

class MyHemiCube {
    MyPoint center;
    MyVector axe_x;
    MyVector axe_y;
    MyVector axe_z;
    double edge;
    double incr;
    MyHemiCube (MyPatch from, MyVector ax, MyVector ay,
              MyPatch dest) {
        center = new MyPoint (from.center);
        axe_x = new MyVector (ax);
        axe_x.normalize ();
        double cos = from.plane.normal.dot
            (new MyVector (from.center, dest.center));
        if (cos > 0) {
            axe_z = new MyVector (from.plane.normal);
        } else if (cos < 0) {
            axe_z = new MyVector (-from.plane.normal.dx,
                                  -from.plane.normal.dy,
                                  -from.plane.normal.dz);
        } else {
            System.out.println ("Hemicube construction: shouldn't occur!");
            System.exit (1);
        }
        axe_z.normalize ();
        axe_y = axe_x.cross (axe_z);
        //axe_y = new MyVector (ay);
        //axe_y.normalize ();
        //axe_z = axe_x.cross (axe_y);
        edge = from.edge * 2;
        incr = 2;
    }
    void print () {
        System.out.println ("HemiCube:");
        center.print ();
        axe_x.print ();
        axe_y.print ();
    }
    double formFactor (MyPatch seen_from) {
        double accu = 0;
        // we check all the hemicube's pixels to find the projected area
        // by using the point cur
        MyPoint cur = new MyPoint (0, 0, 0);
        // top
        double demi_edge = edge / 2.0;
        double demi_incr = incr / 2.0;
        double pixel_area = incr * incr;
        cur.z = demi_edge;
```

```
for (cur.x = - demi_edge; cur.x <= demi_edge; cur.x += incr) {
    for (cur.y = - demi_edge; cur.y <= demi_edge; cur.y += incr) {
        MyRay ray = new MyRay (center, center
                                .plus (axe_x.scale (cur.x + demi_incr))
                                .plus (axe_y.scale (cur.y + demi_incr))
                                .plus (axe_z.scale (cur.z)));
        if (seen_from.intersect (ray)) {
            accu += 1 / (Math.PI *
                        (cur.x * cur.x + cur.y * cur.y + 1) *
                        (cur.x * cur.x + cur.y * cur.y + 1) *
                        pixel_area);
        }
    }
}
cur.x = demi_edge;
// face perpendiculaire a x (x > 0)
for (cur.y = - demi_edge; cur.y <= demi_edge; cur.y += incr) {
    for (cur.z = 0; cur.z <= demi_edge; cur.z += incr) {
        MyRay ray = new MyRay (center, center
                                .plus (axe_x.scale (cur.x))
                                .plus (axe_y.scale (cur.y + demi_incr))
                                .plus (axe_z.scale (cur.z + demi_incr)));
        if (seen_from.intersect (ray)) {
            accu += cur.z / (Math.PI *
                            (cur.z * cur.z + cur.y * cur.y + 1) *
                            (cur.z * cur.z + cur.y * cur.y + 1) *
                            pixel_area);
        }
    }
}
// face perpendiculaire a x (x < 0)
cur.x = - demi_edge;
for (cur.y = - demi_edge; cur.y <= demi_edge; cur.y += incr) {
    for (cur.z = 0; cur.z <= demi_edge; cur.z += incr) {
        MyRay ray = new MyRay (center, center
                                .plus (axe_x.scale (cur.x))
                                .plus (axe_y.scale (cur.y + demi_incr))
                                .plus (axe_z.scale (cur.z + demi_incr)));
        if (seen_from.intersect (ray)) {
            accu += cur.z / (Math.PI *
                            (cur.z * cur.z + cur.y * cur.y + 1) *
                            (cur.z * cur.z + cur.y * cur.y + 1) *
                            pixel_area);
        }
    }
}
// face perpendiculaire a y (y > 0)
cur.y = demi_edge;
for (cur.x = - demi_edge; cur.x <= demi_edge; cur.x += incr) {
    for (cur.z = 0; cur.z <= demi_edge; cur.z += incr) {
        MyRay ray = new MyRay (center, center
                                .plus (axe_x.scale (cur.x + demi_incr))
                                .plus (axe_y.scale (cur.y))
                                .plus (axe_z.scale (cur.z + demi_incr)));
        if (seen_from.intersect (ray)) {
            accu += cur.z / (Math.PI *
                            (cur.x * cur.x + cur.z * cur.z + 1) *
                            (cur.x * cur.x + cur.z * cur.z + 1) *
                            pixel_area);

        }
    }
}
// face perpendiculaire a y (y < 0)
```

```
        cur.y = - demi_edge;
        for (cur.x = - demi_edge; cur.x <= demi_edge; cur.x += incr) {
            for (cur.z = 0; cur.z <= demi_edge; cur.z += incr) {
                MyRay ray = new MyRay (center, center
                                        .plus (axe_x.scale (cur.x + demi_incr))
                                        .plus (axe_y.scale (cur.y))
                                        .plus (axe_z.scale (cur.z + demi_incr)));
                if (seen_from.intersect (ray)) {
                    accu += cur.z / (Math.PI *
                                     (cur.x * cur.x + cur.z * cur.z + 1) *
                                     (cur.x * cur.x + cur.z * cur.z + 1) *
                                     pixel_area);
                }
            }
        }
        return accu;
    }
}

class RadiosityApp {
    static double[][] buffer;

    RadiosityApp (MyImage img) {
        buffer = new double[img.width][img.height];
        for (int i = 0; i < img.width; i++) {
            for (int j = 0; j < img.height; j++) {
                buffer[i][j] = Double.NEGATIVE_INFINITY;
            }
        }
    }

    private MyMatrix gaussSeidel (MyMatrix form_factors,
                                  double[][] refl,
                                  double[][] emit,
                                  int numb_objs,    // number of objects
                                  int numb_patches, // patches per object
                                  int code_color) { // red=0, green=1, blue=2
        // we build the k matrix with form factors and  reflectivities
        double[][] k =
            new double [numb_objs * numb_patches][numb_objs * numb_patches];
        for (int i = 0; i < k.length; i++) {
            for (int j = 0; j < k.length; j++) {
                if (i != j)
                    k[i][j] = - refl[i][code_color] * form_factors.matrix[i][j];
                else
                    k[i][j] = 1 - refl[i][code_color] * form_factors.matrix[i][j];
            }
        }
        MyMatrix k_mat = new MyMatrix (k);

        // k must be diagonal dominant for Gauss-Seidel to converge
        if (!k_mat.isDiagonalDominant ()) {
            System.out.println ("The k matrix is not diagonal dominant!");
            System.exit (1);
        }

        // a better  initial  guess is  the emittance
        double[][] b = new double [numb_objs * numb_patches][1];
        for (int i = 0; i < b.length; i++)
            b[i][0] = emit[i][0];
        MyMatrix b_mat = new MyMatrix (b);
        MyMatrix emit_mat = new MyMatrix (emit);

        // first  residual
```

```
    double res = ((k_mat.times (b_mat)).minus (emit_mat)).normInf ();
    System.out.println ("first res = " + res);

    // system of linear equations' resolution
    int count = 0;
    while (res > 1E-15) {
        for (int i = 0; i < b.length; i++) {
            double sum = 0;
            for (int j = 0; j < numb_objs * numb_patches; j++) {
                if (i != j) {
                    sum += b_mat.matrix[j][0] *
                        k_mat.matrix[i][j] / k_mat.matrix[i][i];
                }
            }
            b_mat.matrix[i][0] = emit_mat.matrix[i][0] - sum;
        }
        res = ((k_mat.times (b_mat)).minus (emit_mat)).normInf ();
        count++;
    }
    System.out.println ("number of loops = " + count);
    System.out.println ("final res = " + res);

    return b_mat;
}

void radiosities (MyScene scene) {
    int numb_objs = scene.objects.length;
    int numb_patches = scene.objects[0].patches.length;

    // some variables to compute the calculation state
    int count = 0;
    int ffs = numb_objs * numb_patches * numb_objs * numb_patches;
    int previous_percentage = 0;
    int current_percentage;

    System.out.println ("compute form factors...");
    double[][] form_factors = new double
        [numb_objs * numb_patches]
        [numb_objs * numb_patches];
    for (int i = 0; i < numb_objs; i++) {
        for (int j = 0; j < numb_patches; j++) {
            // we calculate ff from patch i.j to all the other patches
            for (int m = 0; m < numb_objs; m++) {
                for (int n = 0; n < numb_patches; n++) {
                    // if same support, ff is null because the two
                    // patches can't see each other
                    /*if (i == m) {
                      form_factors
                      [i * numb_patches + j]
                      [m * numb_patches + n] = 0;
                      } else {*/
                    if (scene.objects[i].patches[j].plane.normal.dot
                        (new MyVector
                            (scene.objects[i].patches[j].center,
                             scene.objects[m].patches[n].center))
                        != 0) {
                        MyHemiCube hc = new MyHemiCube
                            (scene.objects[i].patches[j],
                             scene.objects[i].direct0,
                             scene.objects[i].direct1,
                             scene.objects[m].patches[n]);
                        double ff_tmp = 0;
                        ff_tmp = hc.formFactor
                            (scene.objects[m].patches[n]);
```

```
                    form_factors
                        [i * numb_patches + j]
                        [m * numb_patches + n] = ff_tmp;
                } else {
                    form_factors
                        [i * numb_patches + j]
                        [m * numb_patches + n] = 0;
                }
                //}
                current_percentage = count * 100 / ffs;
                if (current_percentage != previous_percentage) {
                    if ((current_percentage % 10) == 0) {
                        System.out.println
                            (current_percentage + "% of ffs done");
                    }
                }
                previous_percentage = current_percentage;
                count++;
            }
        }
    }
}
// we normlize the form factors matrix so that the sum of form
// factors on each row is 1
for (int i = 0; i < form_factors.length; i++) {
    double sum = 0;
    for (int j = 0; j < form_factors.length; j++) {
        sum += form_factors[i][j];
    }
    for (int j = 0; j < form_factors.length; j++) {
        form_factors[i][j] /= (sum + 0.001);
    }
}

for (int i = 0; i < form_factors.length; i++) {
    if (form_factors[i][i] != 0) {
        System.out.println ("Sefl form factors must be 1!");
        System.exit (1);
    }
}

MyMatrix ff = new MyMatrix (form_factors);

System.out.println ("Gauss-Seidel for red wavelength...");
MyMatrix bred_mat = gaussSeidel (ff,
                                 scene.refl,
                                 scene.emit,
                                 numb_objs,
                                 numb_patches,
                                 0);
System.out.println ("Gauss-Seidel for green wavelength...");
MyMatrix bgreen_mat = gaussSeidel (ff,
                                   scene.refl,
                                   scene.emit,
                                   numb_objs,
                                   numb_patches,
                                   1);
System.out.println ("Gauss-Seidel for blue wavelength...");
MyMatrix bblue_mat = gaussSeidel (ff,
                                  scene.refl,
                                  scene.emit,
                                  numb_objs,
                                  numb_patches,
                                  2);
```

```
        // the radiosity is the energy leaving a unit area in a unit time
        // we update the color of each patch with the computed radiosities
        double patch_area = scene.objects[0].patches[0].edge *
            scene.objects[0].patches[0].edge;
        for (int i = 0; i < numb_objs; i++) {
            for (int j = 0; j < numb_patches; j++) {
                scene.objects[i].patches[j].setColor
                    ((patch_area *
                      bred_mat.matrix [i * numb_patches + j][0]),
                     (patch_area *
                      bgreen_mat.matrix [i * numb_patches + j][0]),
                     (patch_area *
                      bblue_mat.matrix [i * numb_patches + j][0]));
            }
        }
    }
    public static void main (String[] args) {
        MyCamera camera = new MyCamera
            (new MyPoint (120, 120, 300), new MyPoint (100, 100, 0),
             true, 700, 700);

        MyScene scene = new MyScene (3);

        // initilize the zbuffer
        RadiosityApp radiosity_app = new RadiosityApp (camera.image);

        // compute the radiosities for each wavelength
        radiosity_app.radiosities (scene);

        // decompose each square patch into two triangles
        scene.changePointOfView (camera);

        System.out.println ("zbuffer the scene..." );
        scene.zBuffer (buffer, camera.image.grid);

        JFrame frame = new JFrame ("radiosity app");
        frame.getContentPane ().add (camera.image);
        frame.addWindowListener (new WindowAdapter() {
                public void windowClosing (WindowEvent e) {
                    System.exit(0);
                }
            });
        frame.pack();
        frame.setVisible (true);
    }
}

// the camera embeds a MyImage image
class MyImage extends JPanel {
    int width;
    int height;
    Color [][] grid;
    Dimension preferred_size;
    MyImage (int width, int height) {
        this.width = width;
        this.height = height;
        grid = new Color[width][height];
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < height; j++) {
                grid[i][j] = new Color (0, 0, 0);
            }
        }
        preferred_size = new Dimension (width, height);
```

```
    }
    public Dimension getPreferredSize () {
        return preferred_size;
    }
    public void paintComponent (Graphics g) {
        super.paintComponent (g); //paint background
        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                g.setColor (grid[x][y]);
                g.drawRect (x, -y + height, 1, 1);
            }
        }
    }
}

class MyCamera {
    boolean perspective;
    MyImage image;
    MyMatrix transAndRot;
    //MyMatrix trans;
    //MyMatrix rotx;
    //MyMatrix roty;
    //MyMatrix pers;
    MyCamera (MyPoint from, MyPoint to, boolean perspective,
              int width, int height) {
        this.perspective = perspective;
        this.image = new MyImage (width, height);

        MyVector n = new MyVector (from, to);
        n.normalize ();

        // change of coordinates w.r.t. the from->to vector
        // verticality is conserved
        double a = 1.0 / Math.sqrt (1 - n.dy * n.dy);
        double[][] t = new double[4][4];
        t [0][0] = - a * n.dz; t [0][1] = 0; t [0][2] = a * n.dx;
        t [0][3] = a * (from.x * n.dz - from.z * n.dx);
        t [1][0] = - a * n.dx * n.dy; t [1][1] = 1 / a;
        t [1][2] = - a * n.dy * n.dz;
        t [1][3] = a * n.dy * (from.x * n.dx + from.z * n.dz) - from.y / a;
        t [2][0] = - n.dx; t [2][1] = - n.dy; t [2][2] = - n.dz;
        t [2][3] = from.x * n.dx + from.y * n.dy + from.z * n.dz;
        if (perspective) {
            double dist = 200;
            t [3][0] = -n.dx / dist;
            t [3][1] = -n.dy / dist;
            t [3][2] = -n.dz /dist;
            t [3][3] = ( from.x * n.dx + from.y * n.dy + from.z * n.dz) / dist;
        } else {
            t [3][0] = 0; t [3][1] = 0; t [3][2] = 0; t [3][3] = 1;
        }
        this.transAndRot = new MyMatrix (t);

        /*double theta_x = Math.PI/16;
          double theta_y = Math.PI/16;

          double [][] tr = new double [4][4];
          tr [0][0] = 1; tr [0][1] = 0; tr [0][2] = 0; tr [0][3] = -from.x;
          tr [1][0] = 0; tr [1][1] = 1; tr [1][2] = 0; tr [1][3] = -from.y;
          tr [2][0] = 0; tr [2][1] = 0; tr [2][2] = 1; tr [2][3] = -from.z;
          tr [3][0] = 0; tr [3][1] = 0; tr [3][2] = 0; tr [3][3] = 1;
          this . trans = new MyMatrix (tr);

          double [][] rx = new double [4][4];
```

```
            rx [0][0] = 1;  rx [0][1] = 0;  rx [0][2] = 0;  rx [0][3] = 0;
            rx [1][0] = 0;  rx [1][1] = Math.cos (theta_x);
            rx[1][2] = −Math.sin (theta_x); rx [1][3] = 0;
            rx [2][0] = 0;  rx [2][1] = Math.sin (theta_x);
            rx [2][2] = Math.cos (theta_x); rx [2][3] = 0;
            rx [3][0] = 0;  rx [3][1] = 0;  rx [3][2] = 0;  rx [3][3] = 1;
            this.rotx = new MyMatrix (rx);

            double [][] ry = new double[4][4];
            ry [0][0] = Math.cos (theta_y); ry [0][1] = 0;
            ry [0][2] = Math.sin (theta_y); ry [0][3] = 0;
            ry [1][0] = 0;  ry [1][1] = 1;  ry [1][2] = 0;  ry [1][3] = 0;
            ry[2][0] = −Math.sin (theta_y); ry [2][1] = 0;
            ry [2][2] = Math.cos (theta_y); ry [2][3] = 0;
            ry [3][0] = 0;  ry [3][1] = 0;  ry [3][2] = 0;  ry [3][3] = 1;
            this.roty = new MyMatrix (ry);

            // perspective transformation
            double [][] p = new double[4][4];
            p [0][0] = 1;  p [0][1] = 0;  p [0][2] = 0;  p [0][3] = 0;
            p [1][0] = 0;  p [1][1] = 1;  p [1][2] = 0;  p [1][3] = 0;
            p [2][0] = 0;  p [2][1] = 0;  p [2][2] = 1;  p [2][3] = 0;
            p [3][0] = 0;  p [3][1] = 0;  p [3][2] = 1.0 /  dist ; p [3][3] = 0;
            this.pers = new MyMatrix (p);*/

        //transAndRot = /*pers.times (*/rotx.times (roty.times (trans))/*)*/;
    }
}

// classes that extend the MyObject class
// scene are made of squares, that are made of patches, that are made of
// triangles , that are supported by planes

class MyPlane {
    MyPoint point;
    MyVector normal;
    double d;
    MyPlane (MyPoint p0, MyPoint p1, MyPoint p2) {
        MyVector v0 = new MyVector (p0, p1);
        MyVector v1 = new MyVector (p0, p2);
        MyVector n = v0.cross (v1);
        n.normalize ();
        this.normal = n;
        this.point = new MyPoint (p0);
        this.d = − normal.dx * point.x
             − normal.dy * point.y
             − normal.dz * point.z;
    }
    MyPlane (MyPlane p) {
        point = new MyPoint (p.point);
        normal = new MyVector (p.normal);
        this.d = − normal.dx * point.x
             − normal.dy * point.y
             − normal.dz * point.z;
    }
    public MyPoint intersect (MyRay ray) {
        double cosinus = ray.vector.dot (normal);
        if (cosinus == 0) {
            return null;
        } else {
            double k = (− normal.dx * ray.origin.x − normal.dy * ray.origin.y
                      − normal.dz * ray.origin.z − d) / cosinus;
            if (k >= 0.001) {
                return new MyPoint (ray.origin.x + k * ray.vector.dx,
```

```
                                ray.origin.y + k * ray.vector.dy,
                                ray.origin.z + k * ray.vector.dz);
            } else {
                return null;
            }
        }
    }
}

abstract class MyObject {
    abstract void zBuffer (double[][] zbuff, Color[][] imag);
}

class MyTriangle extends MyObject {
    boolean visible;
    private MyPoint[] pts_old;
    MyPoint[] pts;
    MyColor color;
    MyPlane zbuffer_plane;
    private int top;
    private int left;
    private int right;
    MyTriangle (MyPoint p0, MyPoint p1, MyPoint p2,
                MyColor color, MyCamera camera) {
        pts_old = new MyPoint[3];
        pts_old[0] = new MyPoint (p0);
        pts_old[1] = new MyPoint (p1);
        pts_old[2] = new MyPoint (p2);

        pts_old[0].changeCoor (camera);
        pts_old[1].changeCoor (camera);
        pts_old[2].changeCoor (camera);

        top = top ();
        if (top != -1) {
            this.visible = true;
            this.color = new MyColor (color);
            pts = new MyPoint[3];
            if (top == 0) {
                pts[0] = pts_old[0];
                pts[1] = pts_old[1];
                pts[2] = pts_old[2];
            } else if (top == 1) {
                pts[0] = pts_old[1];
                pts[1] = pts_old[0];
                pts[2] = pts_old[2];
            } else if (top == 2) {
                pts[0] = pts_old[2];
                pts[1] = pts_old[1];
                pts[2] = pts_old[0];
            }
            top = 0;
            left = left ();
            if (left == 1)
                right = 2;
            else if (left == 2)
                right = 1;
            zbuffer_plane = new MyPlane (pts_old[0], pts_old[1], pts_old[2]);
        } else {
            visible = false;
        }
    }
    private int top () {
        if (pts_old[0].y > pts_old[1].y && pts_old[0].y > pts_old[2].y)
```

```
            return 0;
        if (pts_old[1].y > pts_old[0].y && pts_old[1].y > pts_old[2].y)
            return 1;
        if (pts_old[2].y > pts_old[1].y && pts_old[2].y > pts_old[0].y)
            return 2;
        if (pts_old[0].y == pts_old[1].y && pts_old[2].y < pts_old[0].y)
            if (pts_old[0].x > pts_old[1].x)
                return 1;
            else
                return 0;
        if (pts_old[0].y == pts_old[2].y && pts_old[1].y < pts_old[2].y)
            if (pts_old[0].x > pts_old[2].x)
                return 2;
            else
                return 0;
        if (pts_old[2].y == pts_old[1].y && pts_old[0].y < pts_old[1].y)
            if (pts_old[1].x > pts_old[2].x)
                return 2;
            else
                return 1;
        if (pts_old[0].y == pts_old[1].y && pts_old[0].y == pts_old[2].y) {
            System.out.println ("An invisible triangle ..." );
            return -1;
        }
        System.err.println ("Top method, shouldn't occur!");
        System.exit (1);
        return -1;
    }
    private int left () {
        if (pts[0].x != pts[1].x) {
            if (pts[0].y != pts[1].y) {
                double slope = (pts[1].y - pts[0].y) / (pts[1].x - pts[0].x);
                double b = - pts[1].x * slope + pts[1].y;
                if (slope < 0) {
                    if (pts[2].y > (pts[2].x * slope + b))
                        return 1;
                    else
                        return 2;
                } else {
                    if (pts[2].y < (pts[2].x * slope + b))
                        return 1;
                    else
                        return 2;
                }
            } else {
                if (pts[2].y < pts[1].y)
                    return 2;
                else
                    return 1;
            }
        } else {
            if (pts[2].x < pts[1].x)
                return 2;
            else
                return 1;
        }
    }
    // helper function to smooth jagged edges due to poor zbuffering
    // doesn't work...
    private double epsilon (double s) {
        double ret = 0;

        if (Math.abs (s) < 0.0001) {
            ret = 0;
```

```
    } else if (Math.abs (s) < 0.001) {
        ret = 0.01;
    } else if (Math.abs (s) < 0.1) {
        ret = 0.05;
    } else {
        ret = 0.1;
    }

    return ret;
}
void zBuffer (double[][] zbuff, Color[][] imag) {
    MyPoint top = pts[this.top];
    MyPoint left = pts[this.left];
    MyPoint right = pts[this.right];
    MyPoint bot = null;

    double slope_left = (top.y - left.y) / (top.x - left.x);
    double b_left = top.y - slope_left * top.x;
    double slope_right = (top.y - right.y) / (top.x - right.x);
    double b_right = top.y - slope_right * top.x;

    double epsi;

    epsi = epsilon (slope_right);

    MyPoint pt =
        new MyPoint (top.x, top.y, top.z);
    double buff_zz = pt.z;
    double buff_xx = pt.x;
    double zz_cst_obj = zbuffer_plane.normal.dx / zbuffer_plane.normal.dz;
    double xx_cst_left = 1.0 / slope_left;
    double zz_cst_obj_left =
        (zbuffer_plane.normal.dx / slope_left + zbuffer_plane.normal.dy) /
        zbuffer_plane.normal.dz;

    if (Double.isInfinite (slope_right)) {
        for (; ( pt.y >= right.y) && (pt.y >= left.y); pt.y--) {
            buff_zz = pt.z;
            buff_xx = pt.x;
            for (; pt.x <= right.x; pt.x++) {
                try {
                    if (pt.z > zbuff[(int)pt.x][(int)pt.y]) {
                        zbuff[(int)pt.x][( int)pt.y] = pt.z;
                        imag[(int)pt.x][( int)pt.y] = color.javaColor ();
                    }
                } catch (ArrayIndexOutOfBoundsException e) {
                    System.err.println
                        ("Caught ArrayIndexOutOfBoundsException: " +
                         e.getMessage ());
                }
                pt.z -= zz_cst_obj;
            }
            pt.x = buff_xx - xx_cst_left;
            pt.z = buff_zz + zz_cst_obj_left;
        }
    } else if (slope_right < 0) {
        for (; ( pt.y >= right.y) && (pt.y >= left.y); pt.y--) {
            buff_zz = pt.z;
            buff_xx = pt.x;
            for (; pt.y <= slope_right * pt.x + b_right + epsi; pt.x++) {
                try {
                    if (pt.z > zbuff[(int)pt.x][(int)pt.y]) {
                        zbuff[(int)pt.x][( int)pt.y] = pt.z;
                        imag[(int)pt.x][( int)pt.y] = color.javaColor ();
```

```
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println
                ("Caught ArrayIndexOutOfBoundsException: " +
                 e.getMessage ());
        }
        pt.z -= zz_cst_obj;
    }
    pt.x = buff_xx - xx_cst_left;
    pt.z = buff_zz + zz_cst_obj_left;
  }
} else if (slope_right > 0) {
  for (; ( pt.y >= right.y) && (pt.y >= left.y); pt.y--) {
    buff_zz = pt.z;
    buff_xx = pt.x;
    for (; pt.y >= slope_right * pt.x + b_right - epsi; pt.x++) {
        try {
            if (pt.z > zbuff[(int)pt.x][(int)pt.y]) {
                zbuff[(int)pt.x][(int)pt.y] = pt.z;
                imag[(int)pt.x][(int)pt.y] = color.javaColor ();
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println
                ("Caught ArrayIndexOutOfBoundsException: " +
                 e.getMessage ());
        }
        pt.z -= zz_cst_obj;
    }
    pt.x = buff_xx - xx_cst_left;
    pt.z = buff_zz + zz_cst_obj_left;
  }
}
if (pt.y <= right.y) {
  slope_right = (right.y - left.y) / (right.x - left.x);
  b_right = - slope_right * left.x + left.y;
  bot = left;
} else if (pt.y <= left.y) {
  slope_left = (left.y - right.y) / (left.x - right.x);
  b_left = - slope_left * right.x + right.y;
  xx_cst_left = 1 / slope_left;
  zz_cst_obj_left =
    (zbuffer_plane.normal.dx / slope_left +
     zbuffer_plane.normal.dy) /
    zbuffer_plane.normal.dz;
  pt = left;
  bot = right;
}

epsi = epsilon (slope_right);

if (Double.isInfinite (slope_right)) {
  for (; pt.y >= bot.y; pt.y--) {
    buff_zz = pt.z;
    buff_xx = pt.x;
    for (; pt.x <= right.x; pt.x++) {
        try {
            if (pt.z > zbuff[(int)pt.x][(int)pt.y]) {
                zbuff[(int)pt.x][(int)pt.y] = pt.z;
                imag[(int)pt.x][(int)pt.y] = color.javaColor ();
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println
                ("Caught ArrayIndexOutOfBoundsException: " +
                 e.getMessage ());
```

```
                }
                pt.z -= zz_cst_obj;
            }
            pt.x = buff_xx - xx_cst_left;
            pt.z = buff_zz + zz_cst_obj_left;
        }
    } else if (slope_right < 0) {
        for (; pt.y >= bot.y; pt.y--) {
            buff_zz = pt.z;
            buff_xx = pt.x;
            for (; pt.y <= slope_right * pt.x + b_right + epsi; pt.x++) {
                try {
                    if (pt.z > zbuff[(int)pt.x][(int)pt.y]) {
                        zbuff[(int)pt.x][(int)pt.y] = pt.z;
                        imag[(int)pt.x][(int)pt.y] = color.javaColor ();
                    }
                } catch (ArrayIndexOutOfBoundsException e) {
                    System.err.println
                        ("Caught ArrayIndexOutOfBoundsException: " +
                         e.getMessage ());
                }
                pt.z -= zz_cst_obj;
            }
            pt.x = buff_xx - xx_cst_left;
            pt.z = buff_zz + zz_cst_obj_left;
        }
    } else if (slope_right > 0) {
        for (; pt.y >= bot.y; pt.y--) {
            buff_zz = pt.z;
            buff_xx = pt.x;
            for (; pt.y >= slope_right * pt.x + b_right - epsi; pt.x++) {
                try {
                    if (pt.z > zbuff[(int)pt.x][(int)pt.y]) {
                        zbuff[(int)pt.x][(int)pt.y] = pt.z;
                        imag[(int)pt.x][(int)pt.y] = color.javaColor ();
                    }
                } catch (ArrayIndexOutOfBoundsException e) {
                    System.err.println
                        ("Caught ArrayIndexOutOfBoundsException: " +
                         e.getMessage ());
                }
                pt.z -= zz_cst_obj;
            }
            pt.x = buff_xx - xx_cst_left;
            pt.z = buff_zz + zz_cst_obj_left;
        }
    }

  }
}

class MyPatch extends MyObject {
  MyPoint top_left;
  MyPoint top_right;
  MyPoint bot_left;
  MyPoint bot_right;
  MyPoint center;
  MyColor color;
  double edge;
  // before the zBuffer, each patch is decomposed in two triangles
  MyTriangle triangle0;
  MyTriangle triangle1;
  // supporting plane
  MyPlane plane;
```

```
void setColor (double r, double g, double b) {
    color.red = r;
    color.green = g;
    color.blue = b;
}
MyPatch (MyPoint tl, MyPoint tr, MyPoint bl, MyPoint br, MyPoint c,
        MyColor col, double edge) {
    top_left = new MyPoint (tl);
    top_right = new MyPoint (tr);
    bot_left = new MyPoint (bl);
    bot_right = new MyPoint (br);
    center = new MyPoint (c);
    color = new MyColor (col);
    triangle0 = null;
    triangle1 = null;
    this.edge = edge;
    plane = new MyPlane (top_left, top_right, bot_left);
}
void changePointOfView (MyCamera camera) {
    triangle0 = new MyTriangle (top_left, top_right, bot_left,
                               color, camera);
    triangle1 = new MyTriangle (bot_right, top_right, bot_left,
                               color, camera);
}
void zBuffer (double[][] zbuff, Color[][] imag) {
    if (triangle0.visible == true) {
        triangle0.zBuffer (zbuff, imag);
    } else {
        System.out.println ("Invisible triangle t0 ..." );
    }
    if (triangle1.visible == true) {
        triangle1.zBuffer (zbuff, imag);
    } else {
        System.out.println ("Invisible triangle t1 ..." );
    }
}
boolean intersect (MyRay ray) {
    double cosinus = ray.vector.dot (plane.normal);
    if (cosinus == 0) {
        return false;
    } else {
        double k = (- plane.normal.dx * ray.origin.x
                    - plane.normal.dy * ray.origin.y
                    - plane.normal.dz * ray.origin.z - plane.d) / cosinus;
        if (k >= 0.001) {
            MyPoint inter = new MyPoint (ray.origin.x + k * ray.vector.dx,
                                         ray.origin.y + k * ray.vector.dy,
                                         ray.origin.z + k * ray.vector.dz);
            MyVector from_tl_to_inter = new MyVector (top_left, inter);
            MyVector ax = new MyVector (top_left, top_right);
            ax.normalize ();
            double x_coor = from_tl_to_inter.dot (ax);
            MyVector ay =new MyVector (top_left, bot_left);
            ay.normalize ();
            double y_coor = from_tl_to_inter.dot (ay);
            if (x_coor >= 0 && x_coor <= edge)
                if (y_coor >= 0 && y_coor <= edge)
                    return true;
            return false;
        } else {
            return false;
        }
    }
}
```

```
}

class MySquare {
    MyVector direct0;
    MyVector direct1;
    MyPoint top_left;
    MyPoint top_right;
    MyPoint bot_left;
    MyPoint bot_right;
    MyPoint center;
    MyPatch[] patches;
    MySquare (MyPoint start, MyVector dir0, MyVector dir1,
              double edge, int div, MyColor color) {
        dir0.normalize ();
        dir1.normalize ();
        direct0 = new MyVector (dir0);
        direct1 = new MyVector (dir1);
        top_left = new MyPoint (start);
        top_right = top_left.plus (direct0.scale (edge));
        bot_left = top_left.plus (direct1.scale (edge));
        bot_right = bot_left.plus (direct0.scale (edge));
        center = top_left
            .plus (direct0.scale (edge / 2.0))
            .plus (direct1.scale (edge / 2.0));
        patches = new MyPatch[div * div];

        double patch_edge = edge / (double)div;

        for (int i = 0; i < div; i++) {
            for (int j = 0; j < div; j++) {
                MyPoint patch_top_left = start
                    .plus (direct0.scale (i * patch_edge))
                    .plus (direct1.scale (j * patch_edge));
                MyPoint patch_bot_right = patch_top_left
                    .plus (direct0.scale (patch_edge))
                    .plus (direct1.scale (patch_edge));
                MyPoint patch_center = patch_top_left
                    .plus (direct0.scale (patch_edge / 2.0))
                    .plus (direct1.scale (patch_edge / 2.0));
                MyPoint patch_top_right = patch_top_left
                    .plus (direct0.scale (patch_edge));
                MyPoint patch_bot_left = patch_top_left
                    .plus (direct1.scale (patch_edge));
                patches[i * div + j] = new MyPatch (patch_top_left,
                                                    patch_top_right,
                                                    patch_bot_left,
                                                    patch_bot_right,
                                                    patch_center,
                                                    color,
                                                    patch_edge);
            }
        }
    }
}

class MyScene extends MyObject {
    MySquare objects[];
    double[][] refl;
    double[][] emit;
    // light sources have zero-valued reflectivities
    // emittance is non zero for light source only
    private void positionLight(int obj, int pat_num, int pat) {
        emit[obj * pat_num + pat][0] = 1;
```

```
    refl[obj * pat_num + pat][0] = 0;
    refl[obj * pat_num + pat][1] = 0;
    refl[obj * pat_num + pat][2] = 0;
}
MyScene (int scene_number) {
    int light_obj = 0;
    int light_pat = 0;
    // the number of patches per square must divide the length of the
    // square edge
    MyVector plus_x = new MyVector (1, 0, 0);
    MyVector minus_x = new MyVector (-1, 0, 0);
    MyVector plus_y = new MyVector (0, 1, 0);
    MyVector minus_y = new MyVector (0, -1, 0);
    MyVector plus_z = new MyVector (0, 0, 1);
    MyVector minus_z = new MyVector (0, 0, -1);
    if (scene_number == 0) { // a room with a white cube inside
        objects = new MySquare[11];
        // a square is define by a point, two vectors, and the length
        // of its edges
        // back
        objects[0] = new MySquare (new MyPoint (0, 0, 0),
                                   plus_x, plus_y,
                                   180,
                                   9,
                                   new MyColor (0.5, 0.5, 0.5));
        // left
        objects[1] = new MySquare (new MyPoint (0, 0, 0),
                                   plus_y, plus_z,
                                   180,
                                   9,
                                   new MyColor (0.8, 0.2, 0.2));
        // ground
        objects[2] = new MySquare (new MyPoint (0, 0, 0),
                                   plus_z, plus_x,
                                   180,
                                   9,
                                   new MyColor (0.2, 0.8, 0.2));
        // right
        objects[3] = new MySquare (new MyPoint (180, 180, 180),
                                   minus_z, minus_y,
                                   180,
                                   9,
                                   new MyColor (0.2, 0.2, 0.8));
        // ceiling
        objects[4] = new MySquare (new MyPoint (180, 180, 180),
                                   minus_x, minus_z,
                                   180,
                                   9,
                                   new MyColor (0.9, 0.9, 0.9));
        // obstacle
        // back
        objects[5] = new MySquare (new MyPoint (80, 0, 80),
                                   plus_x, plus_y,
                                   56,
                                   9,
                                   new MyColor (0.25, 0.25, 0.75));
        // left
        objects[6] = new MySquare (new MyPoint (80, 0, 80),
                                   plus_y, plus_z,
                                   56,
                                   9,
                                   new MyColor (0.25, 0.25, 0.75));
        // ground
        objects[7] = new MySquare (new MyPoint (80, 0, 80),
```

```
                               plus_z, plus_x,
                               56,
                               9,
                               new MyColor (0.25, 0.25, 0.75));
    // right
    objects[8] = new MySquare (new MyPoint (136, 56, 136),
                               minus_z, minus_y,
                               56,
                               9,
                               new MyColor (0.2, 0.2, 0.8));
    // ceiling
    objects[9] = new MySquare (new MyPoint (136, 56, 136),
                               minus_x, minus_z,
                               56,
                               9,
                               new MyColor (0.25, 0.25, 0.75));
    // front
    objects[10] = new MySquare (new MyPoint (136, 56, 136),
                               minus_x, minus_y,
                               56,
                               9,
                               new MyColor (0.25, 0.25, 0.75));
    light_obj = 4;
    light_pat = 40;
} else if (scene_number == 1) { // two walls
    objects = new MySquare[2];
    // back
    objects[0] = new MySquare (new MyPoint (0, 0, 0),
                               plus_x, plus_y,
                               180,
                               9,
                               new MyColor (0.5, 0.5, 0.5));
    // right
    objects[1] = new MySquare (new MyPoint (180, 180, 180),
                               minus_y, minus_z,
                               180,
                               9,
                               new MyColor (0.25, 0.25, 0.75));
    light_obj = 0;
    light_pat = 15;
} else if (scene_number == 2) { // an empty room
    objects = new MySquare[5];
    // back
    objects[0] = new MySquare (new MyPoint (0, 0, 0),
                               plus_x, plus_y,
                               200,
                               5,
                               new MyColor (0.5, 0.5, 0.5));
    // left
    objects[1] = new MySquare (new MyPoint (0, 0, 0),
                               plus_y, plus_z,
                               200,
                               5,
                               new MyColor (0.8, 0.2, 0.2));
    // ground
    objects[2] = new MySquare (new MyPoint (0, 0, 0),
                               plus_z, plus_x,
                               200,
                               5,
                               new MyColor (0.2, 0.8, 0.2));
    // right
    objects[3] = new MySquare (new MyPoint (200, 200, 200),
                               minus_z, minus_y,
                               200,
```

```
                                  5,
                                  new MyColor (0.2, 0.2, 0.8));
    // ceiling
    objects[4] = new MySquare (new MyPoint (200, 200, 200),
                                  minus_x, minus_z,
                                  200,
                                  5,
                                  new MyColor (0.9, 0.9, 0.9));
    light_obj = 4;
    light_pat = 12;
} else if (scene_number == 3) { // a room with a small white wall inside
    objects = new MySquare[6];
    // a square is define by a point, two vectors, and the length
    // of its edges
    // back
    objects[0] = new MySquare (new MyPoint (0, 0, 0),
                                  plus_x, plus_y,
                                  180,
                                  9,
                                  new MyColor (0.5, 0.5, 0.5));
    // left
    objects[1] = new MySquare (new MyPoint (0, 0, 0),
                                  plus_y, plus_z,
                                  180,
                                  9,
                                  new MyColor (0.8, 0.2, 0.2));
    // ground
    objects[2] = new MySquare (new MyPoint (0, 0, 0),
                                  plus_z, plus_x,
                                  180,
                                  9,
                                  new MyColor (0.2, 0.8, 0.2));
    // right
    objects[3] = new MySquare (new MyPoint (180, 180, 180),
                                  minus_z, minus_y,
                                  180,
                                  9,
                                  new MyColor (0.2, 0.2, 0.8));
    // ceiling
    objects[4] = new MySquare (new MyPoint (180, 180, 180),
                                  minus_x, minus_z,
                                  180,
                                  9,
                                  new MyColor (0.9, 0.9, 0.9));
    // obstacle
    objects[5] = new MySquare (new MyPoint (70, 0, 150),
                                  plus_x, plus_y,
                                  56,
                                  9,
                                  new MyColor (0.9, 0.9, 0.9));
    light_obj = 4;
    light_pat = 40;
} else if (scene_number == 4) { // for the edges detection app
    objects = new MySquare[24];
    // top
    objects[0] = new MySquare (new MyPoint (50, 150, 100),
                                  minus_y, plus_x,
                                  50,
                                  5,
                                  new MyColor (1, 1, 1));
    objects[1] = new MySquare (new MyPoint (50, 150, 100),
                                  minus_z, minus_y,
                                  50,
                                  5,
```

```
                             new MyColor (1, 1, 1));
objects[2] = new MySquare (new MyPoint (100, 150, 50),
                             minus_y, minus_x,
                             50,
                             5,
                             new MyColor (1, 1, 1));
objects[3] = new MySquare (new MyPoint (100, 150, 50),
                             plus_z, minus_y,
                             50,
                             5,
                             new MyColor (1, 1, 1));
// front
objects[4] = new MySquare (new MyPoint (50, 100, 150),
                             plus_x, minus_z,
                             50,
                             5,
                             new MyColor (1, 1, 1));
objects[5] = new MySquare (new MyPoint (50, 100, 150),
                             minus_z, minus_y,
                             50,
                             5,
                             new MyColor (1, 1, 1));
objects[6] = new MySquare (new MyPoint (100, 50, 150),
                             minus_x, minus_z,
                             50,
                             5,
                             new MyColor (1, 1, 1));
objects[7] = new MySquare (new MyPoint (100, 50, 150),
                             minus_z, plus_y,
                             50,
                             5,
                             new MyColor (1, 1, 1));
// right
objects[8] = new MySquare (new MyPoint (150, 100, 100),
                             minus_y, minus_x,
                             50,
                             5,
                             new MyColor (1, 1, 1));
objects[9] = new MySquare (new MyPoint (150, 100, 100),
                             minus_z, minus_x,
                             50,
                             5,
                             new MyColor (1, 1, 1));
objects[10] = new MySquare (new MyPoint (150, 50, 50),
                              minus_x, plus_y,
                              50,
                              5,
                              new MyColor (1, 1, 1));
objects[11] = new MySquare (new MyPoint (150, 50, 50),
                              plus_z, minus_x,
                              50,
                              5,
                              new MyColor (1, 1, 1));
// left
objects[12] = new MySquare (new MyPoint (50, 100, 100),
                              minus_y, minus_x,
                              50,
                              5,
                              new MyColor (1, 1, 1));
objects[13] = new MySquare (new MyPoint (50, 100, 100),
                              minus_z, minus_x,
                              50,
                              5,
                              new MyColor (1, 1, 1));
```

```
    objects[14] = new MySquare (new MyPoint (50, 50, 50),
                                minus_x, plus_y,
                                50,
                                5,
                                new MyColor (1, 1, 1));
    objects[15] = new MySquare (new MyPoint (50, 50, 50),
                                plus_z, minus_x,
                                50,
                                5,
                                new MyColor (1, 1, 1));
    // bottom
    objects[16] = new MySquare (new MyPoint (50, 50, 100),
                               minus_y, plus_x,
                               50,
                               5,
                               new MyColor (1, 1, 1));
    objects[17] = new MySquare (new MyPoint (50, 50, 100),
                               minus_z, minus_y,
                               50,
                               5,
                               new MyColor (1, 1, 1));
    objects[18] = new MySquare (new MyPoint (100, 50, 50),
                               minus_y, minus_x,
                               50,
                               5,
                               new MyColor (1, 1, 1));
    objects[19] = new MySquare (new MyPoint (100, 50, 50),
                               plus_z, minus_y,
                               50,
                               5,
                               new MyColor (1, 1, 1));
    // back
    objects[20] = new MySquare (new MyPoint (50, 100, 50),
                               plus_x, minus_z,
                               50,
                               5,
                               new MyColor (1, 1, 1));
    objects[21] = new MySquare (new MyPoint (50, 100, 50),
                               minus_z, minus_y,
                               50,
                               5,
                               new MyColor (1, 1, 1));
    objects[22] = new MySquare (new MyPoint (100, 50, 50),
                               minus_x, minus_z,
                               50,
                               5,
                               new MyColor (1, 1, 1));
    objects[23] = new MySquare (new MyPoint (100, 50, 50),
                               minus_z, plus_y,
                               50,
                               5,
                               new MyColor (1, 1, 1));

}
// MyScene embeds an array of objects of length numb_objs
// each MySquare object embeds an arrat of patches
// reflectivities , emittances, and radiosities are stored in column
// matrices, such that the jth patch of the ith object is at the
// index i * numb_patches + j

int numb_objs = objects.length; // number of squares in the scene
int numb_patches = objects[0].patches.length; // patches per squares

// the reflectivity must be specified for each of the wavelengths
```

```
        // light sources have zero-valued reflectivities
        // otherwise, zero-valued reflectivities mean black surfaces
        refl = new double [numb_objs * numb_patches][3];
        for (int i = 0; i < numb_objs; i++) {
            for (int j = 0; j < numb_patches; j++) {
                refl[i * numb_patches + j][0] =
                    objects[i].patches[0].color.red;
                refl[i * numb_patches + j][1] =
                    objects[i].patches[0].color.green;
                refl[i * numb_patches + j][2] =
                    objects[i].patches[0].color.blue;
            }
        }
        // emittance is non zero for light source only
        // non-zero emittance patches must have zero-values reflectivities
        emit = new double [numb_objs * numb_patches][1];
        for (int i = 0; i < emit.length; i++) {
            emit[i][0] = 0;
        }
        // position the light
        positionLight (light_obj, numb_patches, light_pat);
    }
    void changePointOfView (MyCamera camera) {
        for (int i = 0; i < objects.length; i++) {
            for (int j = 0; j < objects[0].patches.length; j++) {
                objects[i].patches[j].changePointOfView (camera);
            }
        }
    }
    void zBuffer (double[][] buff, Color [][] grid) {
        for (int i = 0; i < objects.length; i++) {
            for (int j = 0; j < objects[0].patches.length; j++) {
                objects[i].patches[j].zBuffer (buff, grid);
            }
        }
    }
}

// classes that are not application specific
// (point, vector, color, matrix, ray)

class MyPoint {
    double x;
    double y;
    double z;
    MyPoint (double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    MyPoint (MyPoint pt) {
        this.x = pt.x;
        this.y = pt.y;
        this.z = pt.z;
    }
    MyPoint plus (MyVector v) { // P2 = P1 + V
        return new MyPoint (x + v.dx, y + v.dy, z + v.dz);
    }
    MyPoint minus (MyVector v) { // P2 = P1 - V
        return new MyPoint (x - v.dx, y - v.dy, z - v.dz);
    }
    MyVector minus (MyPoint p) { // V = P2 - P1
        return new MyVector (p, this);
    }
```

```java
    void changeCoor (MyCamera camera) {
        double[][] m = new double[4][1];
        m[0][0] = x; m[1][0] = y; m[2][0] = z; m[3][0] = 1;
        MyMatrix pt = new MyMatrix (m);
        MyMatrix new_pt = camera.transAndRot.times (pt);
        if (camera.perspective) {
            x = -new_pt.matrix[0][0] / new_pt.matrix[3][0];
            y = -new_pt.matrix[1][0] / new_pt.matrix[3][0];
            z = new_pt.matrix[2][0];
        } else {
            x = new_pt.matrix[0][0];
            y = new_pt.matrix[1][0];
        }

        translate (camera.image.width / 2,
                   camera.image.height / 2,
                   0);
    }
    void translate (double dx, double dy, double dz) {
        x += dx;
        y += dy;
        z += dz;
    }
    void print () {
        System.out.println ("(" + x + "; " + y + "; " + z + ")");
    }
}

class MyVector {
    double dx;
    double dy;
    double dz;
    MyVector (double dx, double dy, double dz) {
        this.dx = dx;
        this.dy = dy;
        this.dz = dz;
    }
    MyVector (MyPoint p0, MyPoint p1) {
        this.dx = p1.x - p0.x;
        this.dy = p1.y - p0.y;
        this.dz = p1.z - p0.z;
    }
    MyVector (MyVector v) {
        this.dx = v.dx;
        this.dy = v.dy;
        this.dz = v.dz;
    }
    MyVector cross (MyVector v) {
        return new MyVector (dy * v.dz - dz * v.dy,
                             dz * v.dx - dx * v.dz,
                             dx * v.dy - dy * v.dx);
    }
    public MyVector scale (double k) {
        return new MyVector (dx * k, dy * k, dz * k);
    }
    public double dot (MyVector v) {
        return dx * v.dx + dy * v.dy + dz * v.dz;
    }
    public double length () {
        return Math.sqrt (dx * dx + dy * dy + dz * dz);
    }
    public void normalize () {
        double len = this.length ();
        if (len != 1 && len !=0) {
```

```
            dx /= len;
            dy /= len;
            dz /= len;
        }
    }
    MyVector plus (MyVector v) {
        return new MyVector (dx + v.dx, dy + v.dy, dz + v.dz);
    }
    void print () {
        System.out.println ("dx = " + dx +
                            ", dy = " + dy +
                            ", dz = " + dz);
    }
}

class MyColor {
    double red;
    double green;
    double blue;
    MyColor (double red, double green, double blue) {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }
    MyColor (MyColor c) {
        this.red = c.red;
        this.green = c.green;
        this.blue = c.blue;
    }
    public Color javaColor () {
        if (red < 0.0) red = 0.0;
        if (green < 0.0) green = 0.0;
        if (blue < 0.0) blue = 0.0;
        //if (red > 1.0) red = 1.0;
        //if (green > 1.0) green = 1.0;
        //if (blue > 1.0) blue = 1.0;
        if ((red > 1.0) || ( green > 1.0) || ( blue > 1.0)) {
            double div = Math.max (red, Math.max (green, blue));
            red /= div;
            green /= div;
            blue /= div;
        }
        int i_red = (int)Math.floor (red * 255);
        int i_green = (int)Math.floor (green * 255);
        int i_blue = (int)Math.floor (blue * 255);
        return new Color (i_red, i_green, i_blue);
    }
    void print () {
        System.out.println (" red = " + red + ", green = " + green + ", blue = " + blue);
    }
}

class MyMatrix {
    double[][] matrix;
    int lines;
    int columns;
    MyMatrix (double[][] matrix) {
        this.matrix = matrix;
        this.lines = matrix.length; // number of (horizontal) lines
        int col = matrix[0].length;
        for (int i = 1; i < matrix.length; i++) {
            if (matrix[i].length != col) {
                System.err.println ("a matrix must be a rectangle!");
                System.exit (1);
```

```
        }
    }
    this.columns = matrix[0].length;
}
MyMatrix times (MyMatrix arg) {
    if (this.columns != arg.lines) {
        System.err.println ("can't multiply those matrices!");
        System.exit (1);
    }
    int new_lines = this.lines;
    int new_columns = arg.columns;
    double[][] new_matrix = new double[new_lines][new_columns];
    for (int i = 0; i < new_lines; i++) {
        for (int j = 0; j < new_columns; j++) {
            double sum = 0;
            for (int k = 0; k < this.columns; k++) {
                sum += this.matrix[i][k] * arg.matrix[k][j];
            }
            new_matrix[i][j] = sum;
        }
    }
    return new MyMatrix (new_matrix);
}
MyMatrix minus (MyMatrix arg) {
    double[][] new_matrix = new double[this.lines][this.columns];
    for (int i = 0; i < lines; i++) {
        for (int j = 0; j < columns; j++) {
            new_matrix[i][j] = this.matrix[i][j] - arg.matrix[i][j];
        }
    }
    return new MyMatrix (new_matrix);
}
boolean isNull () {
    for (int i = 0; i < lines; i++) {
        for (int j = 0; j < columns; j++) {
            if (this.matrix[i][j] != 0)
                return false;
        }
    }
    return true;
}
void print () {
    for (int i = 0; i < lines; i++) {
        for (int j = 0 ; j < columns; j++) {
            System.out.print (matrix[i][j] + " ");
        }
        System.out.print ("\n");
    }
}
double norm () {
    if (columns == 1) {
        double sum = 0;
        for (int i = 0; i < lines; i++) {
            sum += matrix[i][0] * matrix[i][0];
        }
        sum = Math.sqrt (sum);
        return sum;
    } else {
        System.err.println ("Norm only for column matrices");
        System.exit (1);
        return -1; // dummy
    }
}
double normInf () {
```

```
        if (columns == 1) {
            double max = 0;
            double abs_val;
            for (int i = 0; i < lines; i++) {
                abs_val = Math.abs (matrix[i][0]);
                if (abs_val > max)
                    max = abs_val;
            }
            return max;
        } else {
            System.err.println ("Norm only for column matrices");
            System.exit (1);
            return -1; // dummy
        }
    }
    boolean isDiagonalDominant () {
        for (int i = 0; i < lines; i++) {
            double sum = 0;
            for (int j = 0; j < columns; j++) {
                if (i != j)
                    sum += Math.abs (matrix[i][j]);
            }
            if (sum > (Math.abs (matrix[i][i])))
                return false;
        }
        return true;
    }
}

class MyRay {
    MyPoint origin;
    MyVector vector;
    MyRay (MyPoint origin, MyVector vector) {
        this.origin = new MyPoint (origin);
        this.vector = new MyVector (vector);
    }
    MyRay (MyPoint origin, MyPoint towards) {
        this.origin = origin;
        MyVector tmp = new MyVector (origin, towards);
        tmp.normalize ();
        this.vector = tmp;
    }
}
```

4 Edges Detection Source Code

MapApp.java

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.BoxLayout;

import java.awt.Color;
import java.awt.GridLayout;
import java.awt.BorderLayout;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

class MapApp {
    static double[][] zbuffer;

    static MyImage depth_map;
```

```
static MyImage depth_map_edges;

static MyImage normal_map_edges;

MapApp (MyImage img) {
    zbuffer =
        new double[img.width][img.height];
    for (int i = 0; i < img.width; i++) {
        for (int j = 0; j < img.height; j++) {
            zbuffer[i][j] = Double.NEGATIVE_INFINITY;
        }
    }
    depth_map = new MyImage (img.width, img.height);
    depth_map_edges = new MyImage (img.width, img.height);
    normal_map_edges = new MyImage (img.width, img.height);
}
// determine the depth map
void depthMap (double[][] buff, Color [][] depths) {
    double min = Double.POSITIVE_INFINITY;
    double max = Double.NEGATIVE_INFINITY;
    for (int x = 0; x < buff.length; x++) {
        for (int y = 0; y < buff.length; y++) {
            if (buff[x][y] < min && !Double.isInfinite (buff[x][y]))
                min = buff[x][y];
            if (buff[x][y] > max)
                max = buff[x][y];
        }
    }
    for (int x = 0; x < buff.length; x++) {
        for (int y = 0; y < buff.length; y++) {
            if (! Double.isInfinite (buff[x][y])) {
                double tmp = (buff[x][y] - min) / (max - min);
                depths[x][y] = new Color ((int)(255 * tmp),
                                          (int)(255 * tmp),
                                          (int)(255 * tmp));
            } else {
                buff[x][y] = -10000;
            }
        }
    }
}
// determine the normal map
void normalMap (MyScene scene) {
    MyVector axe_x = new MyVector (1, 0, 0);
    MyVector axe_y = new MyVector (0, 1, 0);
    MyVector axe_z = new MyVector (0, 0, 1);
    for (int i = 0; i < scene.objects.length; i++) {
        for (int j = 0; j < scene.objects[i].patches.length; j++) {
            double cos_x = axe_x
                .dot (scene.objects[i].patches[j].plane.normal);
            double cos_y = axe_y
                .dot (scene.objects[i].patches[j].plane.normal);
            double cos_z = axe_z
                .dot (scene.objects[i].patches[j].plane.normal);
            double sum = cos_x + cos_y + cos_z;
            scene.objects[i].patches[j].setColor (cos_x/sum,
                                                  cos_y/sum,
                                                  cos_z/sum);
        }
    }
}
private double convolution (double[][] b, int x, int y, double[][] s) {
    double accu = 0;
```

```
    for (int xx = -1; xx <= 1; xx++) {
            for (int yy = -1; yy <= 1; yy++) {
                try {
                    accu += b[x + xx][y + yy] * s[xx + 1][yy + 1];
                } catch (ArrayIndexOutOfBoundsException e) {
                    System.err.println
                        ("Caught ArrayIndexOutOfBoundsException: " +
                         e.getMessage ());
                }
            }
    }
    return accu;
}

void depthEdgeDetection (double[][] buff, Color[][] edges) {
    double[][] s_x = new double[3][3];
    s_x [0][0] = -1; s_x [0][1] = 0;  s_x [0][2] = 1;
    s_x [1][0] = -2; s_x [1][1] = 0;  s_x [1][2] = 2;
    s_x [2][0] = -1; s_x [2][1] = 0;  s_x [2][2] = 1;
    double[][] s_y = new double[3][3];
    s_y [0][0] = -1; s_y [0][1] = -2; s_y [0][2] = -1;
    s_y [1][0] = 0;  s_y [1][1] = 0;  s_y [1][2] = 0;
    s_y [2][0] = 1;  s_y [2][1] = 2;  s_y [2][2] = 1;
    for (int x = 1; x < buff.length - 1; x++) {
        for (int y = 1; y < buff.length - 1; y++) {
            double i_x = convolution (buff, x, y, s_x);
            double i_y = convolution (buff, x, y, s_y);
            double i_mag = Math.sqrt (i_x * i_x + i_y * i_y);
            if (i_mag > 40) {
                edges[x][y] = new Color (255, 255, 255);
            } else {
                edges[x][y] = new Color (0, 0, 0);
            }
        }
    }
}

void normalEdgeDetection (Color[][] buff, Color[][] edges) {
    for (int x = 1; x < buff.length - 1; x++) {
        for (int y = 1; y < buff.length - 1; y++) {
            Color col = buff[x][y];
            if (buff[x-1][y].equals (col) &&
                buff[x+1][y].equals (col) &&
                buff[x][y-1].equals (col) &&
                buff[x][y+1].equals (col) &&
                buff[x+1][y+1].equals (col) &&
                buff[x+1][y-1].equals (col) &&
                buff[x-1][y+1].equals (col) &&
                buff[x-1][y-1].equals (col)) {
                edges[x][y] = new Color (0, 0, 0);
            } else {
                edges[x][y] = new Color (255, 255, 255);
            }
        }
    }
}

public static void main (String[] args) {
    MyCamera camera = new MyCamera
        (new MyPoint (120, 120, 250), new MyPoint (50, 50, 50),
         false/*no perspective*/, 200, 200);

    MyScene scene = new MyScene (4);
```

```
        MapApp map_app = new MapApp (camera.image);

        // color the scene elements so as to make the normal appear
        map_app.normalMap (scene);

        scene.changePointOfView (camera);

        scene.zBuffer (zbuffer, camera.image.grid);

        // compute the depth map of the scene
        map_app.depthMap (zbuffer, depth_map.grid);

        // detect edges thanks to the depth map
        map_app.depthEdgeDetection (zbuffer, depth_map_edges.grid);

        // detect edges thanks to the normal map
        map_app.normalEdgeDetection (camera.image.grid, normal_map_edges.grid);

        // panel for depth map
        JPanel depth_map_pane = new JPanel ();
        depth_map_pane.setLayout
            (new BoxLayout (depth_map_pane, BoxLayout.Y_AXIS));
        depth_map_pane.add (depth_map);
        depth_map_pane.add (new JLabel ("depth_map"));
        // panel for normal map
        JPanel normal_map_pane = new JPanel ();
        normal_map_pane.setLayout
            (new BoxLayout (normal_map_pane, BoxLayout.Y_AXIS));
        normal_map_pane.add (camera.image);
        normal_map_pane.add (new JLabel ("normal_map"));
        // panel for depth map edges
        JPanel depth_map_edges_pane = new JPanel ();
        depth_map_edges_pane.setLayout
            (new BoxLayout (depth_map_edges_pane, BoxLayout.Y_AXIS));
        depth_map_edges_pane.add (depth_map_edges);
        depth_map_edges_pane.add (new JLabel ("depth_map_edges_detection"));
        // panel for normal map edges
        JPanel normal_map_edges_pane = new JPanel ();
        normal_map_edges_pane.setLayout
            (new BoxLayout (normal_map_edges_pane, BoxLayout.Y_AXIS));
        normal_map_edges_pane.add (normal_map_edges);
        normal_map_edges_pane.add (new JLabel ("normal_map_edges_detection"));

        // with GridLayout, all components are given equal size
        JPanel pane = new JPanel ();
        pane.setLayout (new GridLayout (2/*rows*/, 2/*columns*/));
        pane.add (depth_map_pane);
        pane.add (normal_map_pane);
        pane.add (depth_map_edges_pane);
        pane.add (normal_map_edges_pane);

        // add the main panel to the content pane of the frame
        JFrame frame = new JFrame ("map_app");
        frame.getContentPane ().add (pane);
        frame.addWindowListener (new WindowAdapter() {
                public void windowClosing (WindowEvent e) {
                    System.exit(0);
                }
            });
        frame.pack();
        frame.setVisible (true);
    }
}
```