
2

Transformation and Viewing

Chapter Objectives:

- Understand basic transformation and viewing methods
- Understand 3D hidden-surface removal and collision detection
- Design and implement 3D models (cone, cylinder, and sphere) and their animations in OpenGL

2.1 Geometric Transformation

In Chapter 1, we discussed creating and scan-converting primitive models. After a computer-based model is generated, it can be moved around or even transformed into a completely different shape. To do this, we need to specify the rotation axis and angle, translation vector, scaling vector, or other manipulations to the model. The ordinary *geometric transformation* is a process of mathematical manipulations of all the vertices of the model through matrix multiplications, where the graphics system then displays the final transformed model. The transformation can be predefined, such as moving along a planned trajectory; or interactive, depending on the user input. The transformation can be permanent — the coordinates of the vertices are changed and we have a new model replacing the original one; or just temporary — the vertices return to their original coordinates. In many cases a model is transformed in order to be displayed at a different position or orientation, and the graphics system discards the transformed model after scan-conversion. Sometimes all the vertices of a model go through the same transformation, and the shape of the model is preserved; sometimes different vertices go through different transformations, and the shape is dynamic.

A model can be displayed repetitively with each frame going through a small transformation step. This causes the model to be animated on display.

2.2 2D Transformation

Translation, rotation, and scaling are the basic and essential transformations. They can be combined to achieve most transformations in many applications. To simplify the discussion, we will first introduce 2D transformation and then generalize it into 3D.

2.2.1 2D Translation

A point (x, y) is translated to (x', y') by a distance vector (d_x, d_y) :

$$x' = x + d_x, \quad \text{(EQ 12)}$$

$$y' = y + d_y. \quad \text{(EQ 13)}$$

In the homogeneous coordinates, we represent a point (x, y) by a column vector

$P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. Similarly, $P' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$. Then, translation can be achieved by matrix

multiplication:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad \text{(EQ 14)}$$

Let's assume $T(d_x, d_y) = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}$. We can denote the translation matrix equation as:

$$P' = T(d_x, d_y)P. \quad \text{(EQ 15)}$$

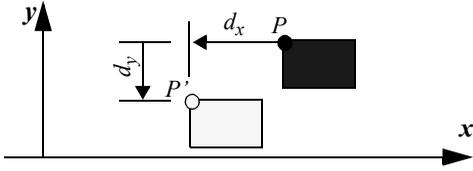


Fig. 2.1 Basic transformation: translation

If a model is a set of vertices, all vertices of the model can be translated as points by the same translation vector (Fig. 2.1). Note that translation moves a model through a distance without changing its orientation.

2.2.2 2D Rotation

A point $P(x, y)$ is rotated counter-clockwise to $P'(x', y')$ by an angle θ around the origin $(0, 0)$. Let us assume that the distance from the origin to point P is $r = OP$, and the angle between OP and x axis is α . If the rotation is clockwise, the rotation angle θ is then negative. The rotation axis is perpendicular to the 2D plane at the origin:

$$x' = r \cos(\alpha + \theta), \quad (\text{EQ 16})$$

$$y' = r \sin(\alpha + \theta), \quad (\text{EQ 17})$$

$$x' = r(\cos \alpha \cos \theta - \sin \alpha \sin \theta), \quad (\text{EQ 18})$$

$$y' = r(\sin \alpha \cos \theta + \cos \alpha \sin \theta), \quad (\text{EQ 19})$$

$$x' = x \cos \theta - y \sin \theta, \quad (\text{EQ 20})$$

$$y' = x \sin \theta + y \cos \theta. \quad (\text{EQ 21})$$

In the homogeneous coordinates, rotation can be achieved by matrix multiplication:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (\text{EQ 22})$$

Let's assume $R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$. The simplified rotation matrix equation is

$$P' = R(\theta)P. \quad (\text{EQ 23})$$

If a model is a set of vertices, all vertices of the model can be rotated as points by the same angle around the same rotation axis (Fig. 2.2). Rotation moves a model around the origin of the coordinates. The distance of each vertex to the origin is not changed during rotation.

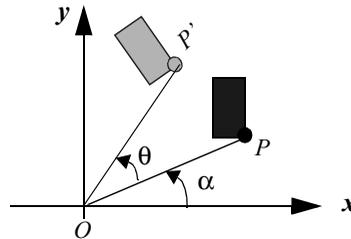


Fig. 2.2 Basic transformation: rotation

2.2.3 2D Scaling

A point $P(x, y)$ is scaled to $P(x', y')$ by a scaling vector (s_x, s_y) :

$$x' = s_x x, \quad (\text{EQ 24})$$

$$y' = s_y y. \quad (\text{EQ 25})$$

In the homogeneous coordinates, again, scaling can be achieved by matrix multiplication:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (\text{EQ 26})$$

Let's assume $S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$. We can denote the scaling matrix equation as:

$$P = S(s_x, s_y)P. \quad (\text{EQ 27})$$

If a model is a set of vertices, all vertices of the model can be scaled as points by the same scaling vector (Fig. 2.3). Scaling amplifies or shrinks a model around the origin of the coordinates. Note that a scaled vertex will move unless it is at the origin.

2.2.4 Simulating OpenGL Implementation

OpenGL actually implements 3D transformations, which we will discuss later. Here, we implement 2D transformations in our own code in `J2_0_2DTransform.java`, which corresponds to the OpenGL implementation in hardware.

OpenGL has a MODELVIEW matrix stack that saves the current matrices for transformation. Let us define a matrix stack as follows:

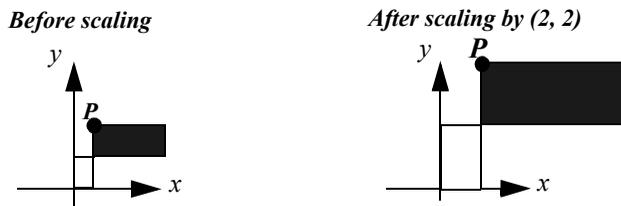


Fig. 2.3 Basic transformation: scaling

/* 2D transformation OpenGL style implementation */

```
import net.java.games.jogl.*;

public class J2_0_2DTransform extends J1_5_Circle {
    private static float my2dMatStack[][][] =
        new float[24][3][3];
    private static int stackPtr = 0;

    ...
}
```

The identity matrix for 2D homogeneous coordinates is $I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. Any matrix

multiplied with identity matrix does not change.

The `stackPtr` points to the current matrix on the matrix stack (`my2dMatrixStack[stackPtr]`) that is in use. Transformations are then achieved by the following methods: `my2dLoadIdentity()`, `my2dMultMatrix(float mat[][])`, `my2dTranslatef(float x, float y)`, `my2dRotatef(float angle)`, `my2dScalef(float x, float y)`, and `my2dTransformf(float vertex[], float vertex1[])` (or `my2dTransVertex(float vertex[], float vertex1[])`) for vertices already in homogeneous form).

1. `my2dLoadIdentity()` loads the current matrix on the matrix stack with the identity matrix:

```
// initialize a 3*3 matrix to all zeros
private void my2dClearMatrix(float mat[][]) {

    for (int i = 0; i<3; i++) {
        for (int j = 0; j<3; j++) {
            mat[i][j] = 0.0f;
        }
    }
}

// initialize a matrix to Identity matrix
private void my2dIdentity(float mat[][]) {
```

```

    my2dClearMatrix(mat);
    for (int i = 0; i<3; i++) {
        mat[i][i] = 1.0f;
    }
}

```

```

// initialize the current matrix to Identity matrix
public void my2dLoadIdentity() {
    my2dIdentity(my2dMatStack[stackPtr]);
}

```

2. *my2dMultMatrix(float mat[][])* multiplies the current matrix on the matrix stack with the matrix *mat*. $\text{CurrentMatrix} = \text{currentMatrix} * \text{Mat}$.

```

// multiply the current matrix with mat
public void my2dMultMatrix(float mat[][]) {
    float matTmp[][] = new float[3][3];

    my2dClearMatrix(matTmp);

    for (int i = 0; i<3; i++) {
        for (int j = 0; j<3; j++) {
            for (int k = 0; k<3; k++) {
                matTmp[i][j] +=
                    my2dMatStack[stackPtr][i][k] * mat[k][j];
            }
        }
    }

    // save the result on the current matrix
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<3; j++) {
            my2dMatStack[stackPtr][i][j] = matTmp[i][j];
        }
    }
}

```

3. *my2dTranslatef(float x, float y)* multiplies the current matrix on the matrix stack with the translation matrix $T(x, y)$:

```

// multiply the current matrix with a translation matrix
public void my2dTranslatef(float x, float y) {

```

```
float T[] [] = new float [3] [3];  
  
my2dIdentity(T);  
  
T[0] [2] = x;  
T[1] [2] = y;  
  
my2dMultMatrix(T);  
}
```

4. *my2dRotatef(float angle)* multiplies the current matrix on the matrix stack with the rotation matrix $R(\text{angle})$:

```
// multiply the current matrix with a rotation matrix  
public void my2dRotatef(float angle) {  
    float R[] [] = new float [3] [3];  
  
    my2dIdentity(R);  
  
    R[0] [0] = (float)Math.cos (angle);  
    R[0] [1] = (float)-Math.sin (angle);  
    R[1] [0] = (float)Math.sin (angle);  
    R[1] [1] = (float)Math.cos (angle);  
  
    my2dMultMatrix(R);  
}
```

5. *my2dScalef(float x, float y)* multiplies the current matrix on the matrix stack with the scaling matrix $S(x, y)$:

```
// multiply the current matrix with a scale matrix  
public void my2dScalef(float x, float y) {  
    float S[] [] = new float [3] [3];  
  
    my2dIdentity(S);  
  
    S[0] [0] = x;  
    S[1] [1] = y;  
  
    my2dMultMatrix(S);  
}
```

6. `my2dTransformf(float vertex[]; vertex1[])` multiplies the current matrix on the matrix stack with `vertex`, and save the result in `vertex1`. Here `vertex` is first extended to homogeneous coordinates before matrix multiplication.

```

// v1 = (the current matrix) * v
// here v and v1 are vertices in homogeneous coord.
public void my2dTransHomoVertex(float v[], float v1[]) {
    int i, j;

    for (i = 0; i<3; i++) {
        v1[i] = 0.0f;
    }
    for (i = 0; i<3; i++) {
        for (j = 0; j<3; j++) {
            v1[i] +=
                my2dMatStack[stackPtr][i][j]*v[j];
        }
    }
}

// vertex = (the current matrix) * vertex
// here vertex is in homogeneous coord.
public void my2dTransHomoVertex(float vertex[]) {
    float vertex1[] = new float[3];

    my2dTransHomoVertex(vertex, vertex1);
    for (int i = 0; i<3; i++) {
        vertex[i] = vertex1[i];
    }
}

// transform v to v1 by the current matrix
// here v and v1 are not in homogeneous coordinates
public void my2dTransformf(float v[], float v1[]) {
    float vertex[] = new float[3];

    // extend to homogenous coord
    vertex[0] = v[0];
    vertex[1] = v[1];
    vertex[2] = 1;

    // multiply the vertex by the current matrix
    my2dTransHomoVertex(vertex);
}

```

```
// return to 3D coord
v1[0] = vertex[0]/vertex[2];
v1[1] = vertex[1]/vertex[2];
}

// transform v by the current matrix
// here v is not in homogeneous coordinates
public void my2dTransformf(float[] v) {
    float vertex[] = new float[3];

    // extend to homogenous coord
    vertex[0] = v[0];
    vertex[1] = v[1];
    vertex[2] = 1;

    // multiply the vertex by the current matrix
    my2dTransHomoVertex(vertex);

    // return to 3D coord
    v[0] = vertex[0]/vertex[2];
    v[1] = vertex[1]/vertex[2];
}
```

7. In addition to the above methods, *my2dPushMatrix()* and *my2dPopMatrix()* are a powerful mechanism to change the current matrix on the matrix stack, which we will discuss in more detail later. *PushMatrix* will increase the stack pointer and make a copy of the previous matrix to the current matrix. Therefore, the matrix remains the same, but we are using a different set of memory locations on the matrix stack. *PopMatrix* will decrease the stack pointer, so we return to the previous matrix that was saved at *PushMatrix*.

```
// move the stack pointer up, and copy the previous
// matrix to the current matrix
public void my2dPushMatrix() {
    int tmp = stackPtr+1;

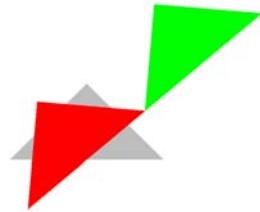
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<3; j++) {
            my2dMatStack[tmp][i][j] =
                my2dMatStack[stackPtr][i][j];
        }
    }
    stackPtr++;
}
```

```
// move the stack pointer down
public void my2dPopMatrix() {

    stackPtr--;
}

```

With the above 2D transformation methods, the following example (*J2_0_2DTransform.java*) achieves different transformations using the implemented methods, as shown in Fig. 2.4.



/* 2D transformation: OpenGL style implementation */

```
import net.java.games.jogl.*;

public class J2_0_2DTransform
extends J1_5_Circle {

    ....// the matrix stack

    static float vdata[] [] = { {1.0f, 0.0f, 0.0f}
                                , {0.0f, 1.0f, 0.0f}
                                , {-1.0f, 0.0f, 0.0f}
                                , {0.0f, -1.0f, 0.0f}
                                };
    static int cnt = 1;

    // called for OpenGL rendering every reshape
    public void display(GLDrawable drawable) {

        if (cnt<1||cnt>200) {
            flip = -flip;
        }
        cnt = cnt+flip;

        gl.glClear(GL.GL_COLOR_BUFFER_BIT);

        // white triangle is scaled
        gl.glColor3f(1, 1, 1);
        my2dLoadIdentity();
    }
}

```

Fig. 2.4 Transformations of triangles [See Color Plate 1]

```
my2dScalef(cnt, cnt);
transDrawTriangle(vdata[0], vdata[1], vdata[2]);

// red triangle is rotated and scaled
gl.glColor3f(1, 0, 0);
my2dRotatef((float)cnt/15);
transDrawTriangle(vdata[0], vdata[1], vdata[2]);

// green triangle is translated, rotated, and scaled
gl.glColor3f(0, 1, 0);
my2dTranslatef((float)cnt/100, 0.0f);
transDrawTriangle(vdata[0], vdata[1], vdata[2]);

try {
    Thread.sleep(20);
} catch (InterruptedException e) {}
}

// the vertices are transformed first then drawn
public void transDrawTriangle(float[] v1,
                             float[] v2, float[] v3) {
    float v[][] = new float[3][3];

    my2dTransformf(v1, v[0]);
    my2dTransformf(v2, v[1]);
    my2dTransformf(v3, v[2]);

    gl.glBegin(GL.GL_TRIANGLES);
    gl.glVertex3fv(v[0]);
    gl.glVertex3fv(v[1]);
    gl.glVertex3fv(v[2]);
    gl.glEnd();
}

... // the transformation methods

public static void main(String[] args) {
    J2_0_2DTransform f = new J2_0_2DTransform();

    f.setTitle("JOGL J2_0_2DTransform");
    f.setSize(500, 500);
    f.setVisible(true);
}
}
```

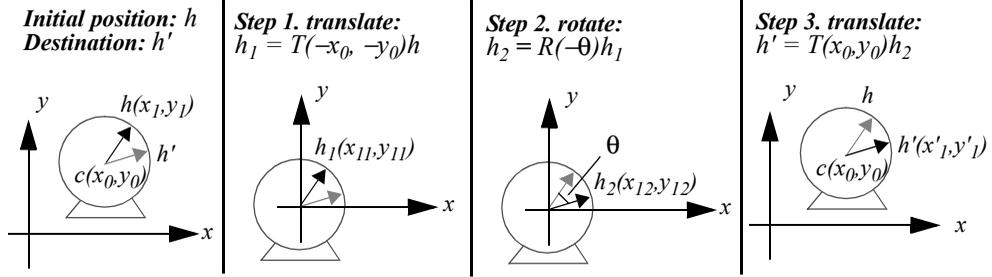


Fig. 2.5 Moving the clock hand by matrix multiplications

2.2.5 Composition of 2D Transformations

A complex transformation is often achieved by a series of simple transformation steps. The result is a composition of translations, rotations, and scalings. We will study this through the following three examples.

Example 1: Find the coordinates of a moving clock hand in 2D. Consider a single clock hand. The center of rotation is given at $c(x_0, y_0)$, and the end rotation point is at $h(x_1, y_1)$. If we know the rotation angle is θ , can we find the new end point h' after the rotation? As shown in Fig. 2.5, we can achieve this by a series of transformations.

1. Translate the hand so that the center of rotation is at the origin. Note that we only need to find the new coordinates of the end point h :

$$\begin{bmatrix} x_{11} \\ y_{11} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}. \tag{EQ 28}$$

$$\text{That is, } h_1 = T(-x_0, -y_0)h. \tag{EQ 29}$$

2. Rotate θ degrees around the origin. Note that the positive direction of rotation is counter-clockwise:

$$h_2 = R(-\theta)h_1. \quad (\text{EQ 30})$$

3. After the rotation. We translate again to move the clock back to its original position:

$$h' = T(x_0, y_0)h_2. \quad (\text{EQ 31})$$

Therefore, putting Equations 29, 30, and 31 together, the combination of transformations to achieve the clock hand movement is

$$h' = T(x_0, y_0)R(-\theta)T(-x_0, -y_0)h. \quad (\text{EQ 32})$$

$$\text{That is, } \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}. \quad (\text{EQ 33})$$

In the future, we will write matrix equations concisely using only symbol notations instead of full matrix expressions. However, we should always remember that the symbols represent the corresponding matrices.

Let's assume $M = T(x_0, y_0)R(-\theta)T(-x_0, -y_0)$. We can further simplify the equation:

$$h' = Mh. \quad (\text{EQ 34})$$

The order of the matrices in a matrix expression matters. The sequence represents the order of the transformations. For example, although matrix M in Equation 34 can be calculated by multiplying the first two matrices first $[T(x_0, y_0)R(-\theta)]T(-x_0, -y_0)$ or by multiplying the last two matrices first $T(x_0, y_0)[R(-\theta)T(-x_0, -y_0)]$, the order of the matrices cannot be changed.

When we analyze a model's transformations, we should remember that, logically speaking, the order of transformation steps are from right to left in the matrix expression. In this example, the first logical step is $T(-x_0, -y_0)h$; the second step is $R(-\theta)[T(-x_0, -y_0)h]$; and the last step is $T(x_0, y_0)[R(-\theta)[T(-x_0, -y_0)h]]$. In the actual OpenGL style implementation, the matrix multiplication is from left to right, and there

is always a final matrix on the matrix stack. The following is a segment of *J2_1_Clock2d.java* that simulates a real-time clock.

```
my2dLoadIdentity();
my2dTranslate(c[0], c[1]); // x0=c[0], y0=c[1];
my2dRotate(-a);
my2dTranslate(-c[0], -c[1]);
transDrawClock(c, h);
```

In the above code, first the current matrix on the matrix stack is loaded with the identity matrix I , then it is multiplied by a translation matrix $T(x_0, y_0)$, after that it is multiplied by a rotation matrix $R(-\theta)$, and finally it is multiplied by a translation matrix $T(-x_0, -y_0)$. Written in an expression, it is $[[[I]T(x_0, y_0)]R(-\theta)]T(-x_0, -y_0)$. In *transDrawClock()*, the clock center c and end h are both transformed by the current matrix, and then scan converted to display. In OpenGL, transformation is implied. In other words, the vertices are first transformed by the system before they are sent to the scan-conversion. The following is the complete program.

/* 2D clock hand transformation */

```
public class J2_1_Clock2d extends J2_0_2DTransform {
    static final float PI = 3.1415926f;

    public void display(GLDrawable glDrawable) {
        // homogeneous coordinates
        float c[] = {0, 0, 1};
        float h[] = {0, WIDTH/6, 1};

        long curTime;
        float ang, second, minute, hour;

        gl.glClear(GL.GL_COLOR_BUFFER_BIT);

        curTime = System.currentTimeMillis()/1000;
        // returns the current time in milliseconds
        hsecond = curTime%60;
        curTime = curTime/60;
        hminute = curTime%60+hsecond/60;
        curTime = curTime/60;
        hhour = (curTime%12)+8+hminute/60;
        // Eastern Standard Time
```

```
    ang = PI*second/30; // arc angle

    gl.glColor3f(1, 0, 0); // second hand in red
    my2dLoadIdentity();
    my2dTranslatef(c[0], c[1]);
    my2dRotatef(-ang);
    my2dTranslatef(-c[0], -c[1]);
    gl.glLineWidth(1);
    transDrawClock(c, h);

    gl.glColor3f(0, 1, 0); // minute hand in green
    my2dLoadIdentity();
    ang = PI*minute/30; // arc angle
    my2dTranslatef(c[0], c[1]);
    my2dScalef(0.8f, 0.8f); // minute hand shorter
    my2dRotatef(-ang);
    my2dTranslatef(-c[0], -c[1]);
    gl.glLineWidth(2);
    transDrawClock(c, h);

    gl.glColor3f(0, 0, 1); // hour hand in blue
    my2dLoadIdentity();
    ang = PI*hour/6; // arc angle
    my2dTranslatef(c[0], c[1]);
    my2dScalef(0.5f, 0.5f); // hour hand shortest
    my2dRotatef(-ang);
    my2dTranslatef(-c[0], -c[1]);
    gl.glLineWidth(3);
    transDrawClock(c, h);
}

public void transDrawClock(float C[], float H[]) {
    float End1[] = new float[3];
    float End2[] = new float[3];

    my2dTransHomoVertex(C, End1);
    // Transform the center by the current matrix
    my2dTransHomoVertex(H, End2);
    // Transform the end by the current matrix

    // assuming z = w = 1;
    gl.glBegin(GL.GL_LINES);
    gl.glVertex3fv(End1);
    gl.glVertex3fv(End2);
    gl.glEnd();
}
```

```

public static void main(String[] args) {
    J2_1_Clock2d f = new J2_1_Clock2d();

    f.setTitle("JOGL J2_1_Clock2d");
    f.setSize(500, 500);
    f.setVisible(true);
}
}

```

Example 2: Reshaping a rectangular area. In OpenGL, we can use the mouse to reshape the display area. In the Reshape callback function, we can use *glViewport()* to adjust the size of the drawing area accordingly. The system makes corresponding adjustments to the models through the same transformation matrix. Viewport transformation will be discussed later in the section “Viewing”.

Here, we discuss a similar problem: a transformation that allows reshaping a rectangular area. Let's assume the coordinate system of the screen is as in Fig. 2.6. After reshaping, the rectangular area and all the vertices of the model inside the rectangular area go through the following transformations: translate so that the lower-left corner of the area is at the origin, scale to the size of the new area, and then translate to the scaled area location.

$$T(P_2)S(s_x, s_y)T(-P_1). \quad (\text{EQ 35})$$

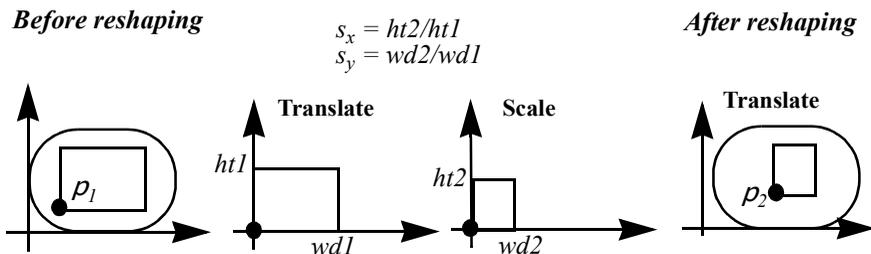


Fig. 2.6 Scaling an arbitrary rectangular area

P_1 is the starting point for scaling, and P_2 is the destination. We can use the mouse to interactively drag P_1 to P_2 in order to reshape the corresponding rectangular area. In the following example (*J2_2_Reshape.java*), we use the mouse to drag the lower-left vertex P_1 of the rectangular area to a new location. The rectangle and the clock inside are reshaped accordingly. A snapshot is shown in Fig. 2.7.

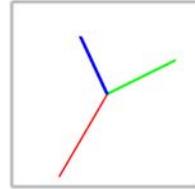


Fig. 2.7 Reshape a drawing area with a clock inside

```
/* reshape the rectangular drawing area
*/
```

```
import net.java.games.jogl.*;
import java.awt.event.*;

public class J2_2_Reshape extends J2_1_Clock2d implements
    MouseMotionListener {

    // the point to be dragged as the lower-left corner
    private static float P1[] = {-WIDTH/4, -HEIGHT/4};

    // reshape scale value
    private float sx = 1, sy = 1;

    // when mouse is dragged, a new lower-left point
    // and scale value for the rectangular area
    public void mouseDragged(MouseEvent e) {
        float wd1 = WIDTH/2;
        float ht1 = HEIGHT/2;

        // The mouse location, new lower-left corner
        P1[0] = e.getX()-WIDTH/2;
        P1[1] = HEIGHT/2-e.getY();
        float wd2 = WIDTH/4-P1[0];
        float ht2 = HEIGHT/4-P1[1];
```

```

    // scale value of the current rectangular area
    sx = wd2/wd1;
    sy = ht2/ht1;
}

public void mouseMoved(MouseEvent e) {
}

public void init(GLDrawable drawable) {

    super.init(drawable);
    // listen to mouse motion
    drawable.addMouseListener(this);
}

public void display(GLDrawable glDrawable) {
    // the rectangle lower-left and upper-right corners
    float v0[] = {-WIDTH/4, -HEIGHT/4};
    float v1[] = {WIDTH/4, HEIGHT/4};

    // reshape according to the current scale
    my2dLoadIdentity();
    my2dTranslatef(P1[0], P1[1]);
    my2dScalef(sx, sy);
    my2dTranslatef(-v0[0], -v0[1]);

    gl.glClear(GL.GL_COLOR_BUFFER_BIT);
    gl.glColor3f(1, 1, 1); // the rectangle is white

    // rectangle area
    float v00[] = new float[2], v11[] = new float[2];
    my2dTransformf(v0, v00);
    my2dTransformf(v1, v11);
    gl.glBegin(GL.GL_LINE_LOOP);
    gl.glVertex3f(v00[0], v00[1], 0);
    gl.glVertex3f(v11[0], v00[1], 0);
    gl.glVertex3f(v11[0], v11[1], 0);
    gl.glVertex3f(v00[0], v11[1], 0);
    gl.glEnd();

    // the clock hands go through the same transformation
    curTime = System.currentTimeMillis()/1000;
    hsecond = curTime%60;
    curTime = curTime/60;
    hminute = curTime%60+hsecond/60;
    curTime = curTime/60;
}

```

```
    hhour = (curTime%12)+8+hminute/60;
    // Eastern Standard Time

    hAngle = PI*hsecond/30; // arc angle

    gl.glColor3f(1, 0, 0); // second hand in red
    my2dTranslatef(c[0], c[1]);
    my2dRotatef(-hAngle);
    my2dTranslatef(-c[0], -c[1]);
    gl.glLineWidth(3);
    transDrawClock(c, h);

    gl.glColor3f(0, 1, 0); // minute hand in green
    my2dLoadIdentity();
    my2dTranslatef(P1[0], P1[1]);
    my2dScalef(sx, sy);
    my2dTranslatef(-v0[0], -v0[1]);
    hAngle = PI*hminute/30; // arc angle
    my2dTranslatef(c[0], c[1]);
    my2dScalef(0.8f, 0.8f); // minute hand shorter
    my2dRotatef(-hAngle);
    my2dTranslatef(-c[0], -c[1]);
    gl.glLineWidth(5);
    transDrawClock(c, h);

    gl.glColor3f(0, 0, 1); // hour hand in blue
    my2dLoadIdentity();
    my2dTranslatef(P1[0], P1[1]);
    my2dScalef(sx, sy);
    my2dTranslatef(-v0[0], -v0[1]);
    hAngle = PI*hhour/6; // arc angle
    my2dTranslatef(c[0], c[1]);
    my2dScalef(0.5f, 0.5f); // hour hand shortest
    my2dRotatef(-hAngle);
    my2dTranslatef(-c[0], -c[1]);
    gl.glLineWidth(7);
    transDrawClock(c, h);
}

public static void main(String[] args) {
    J2_2_Reshape f = new J2_2_Reshape();

    f.setTitle("JOGL J2_2_Reshape");
    f.setSize(500, 500);
    f.setVisible(true);
}
}
```

Example 3: Drawing a 2D robot arm with three moving segments. A 2D robot arm has 3 segments rotating at the joints in a 2D plane (Fig. 2.8). Given an arbitrary initial posture (A, B, C), let's find the transformation matrix expressions for another posture (A_f, B_f, C_f) with respective rotations (α, β, γ) around the joints. Here we specify (A, B, C) on the x axis, which is used to simplify the visualization. (A, B, C) can be initialized arbitrarily. There are many different methods to achieve the same goal. Here, we elaborate three methods for the same goal.

Method I.

1. Rotate $oABC$ around the origin by α degrees:

$$A_f = R(\alpha)A; B' = R(\alpha)B; C' = R(\alpha)C. \tag{EQ 36}$$

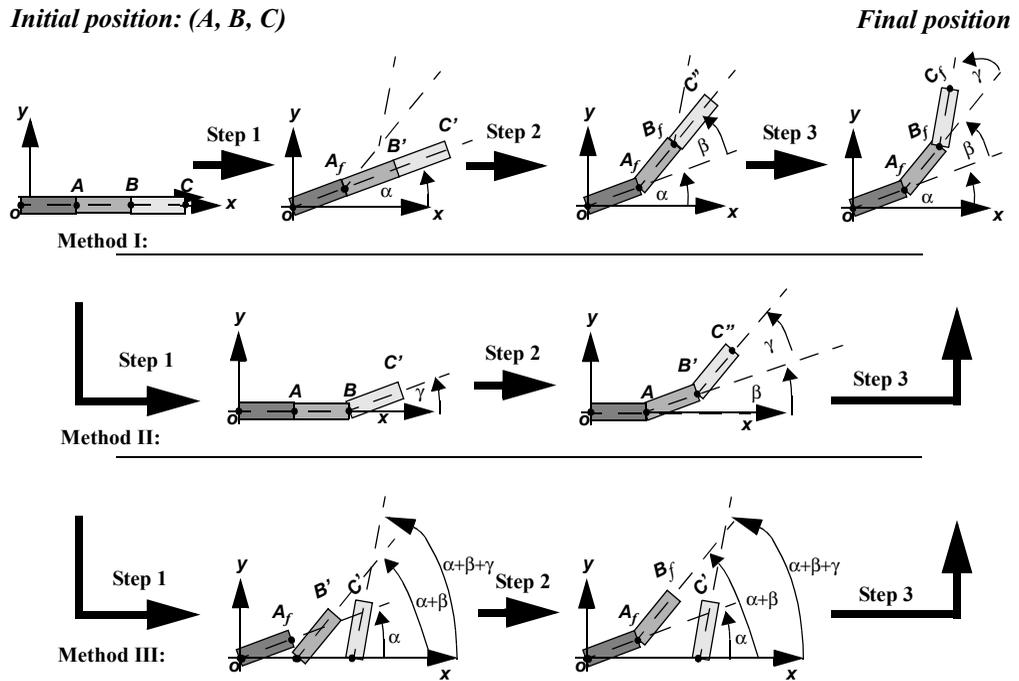


Fig. 2.8 A 2D robot arm rotates (α, β, γ) degrees at the 3 joints, respectively

2. Consider $A_f B' C'$ to be a clock hand like the example in Fig. 2.5. Rotate $A_f B' C'$ around A_f by β degrees. This is achieved by first translating the hand to the origin, rotating, then translating back:

$$B_f = T(A_f)R(\beta)T(-A_f)B'; C'' = T(A_f)R(\beta)T(-A_f)C'. \quad (\text{EQ 37})$$

3. Again, consider $B_f C''$ to be a clock hand. Rotate $B_f C''$ around B_f by γ degrees:

$$C_f = T(B_f)R(\gamma)T(-B_f)C''. \quad (\text{EQ 38})$$

The corresponding code is as follows. Here *my2dTransHomoVertex*(*v1*, *v2*) will multiply the current matrix on the matrix stack with *v1*, and save the results in *v2*. *drawArm()* is just drawing a line segment.

```
// Method I: 2D robot arm transformations
public void transDrawArm1(float a, float b, float g) {
    float Af[] = new float[3];
    float B1[] = new float[3];
    float C1[] = new float[3];
    float Bf[] = new float[3];
    float C2[] = new float[3];
    float Cf[] = new float[3];

    my2dLoadIdentity();
    my2dRotatef(a);
    my2dTransHomoVertex(A, Af);
    my2dTransHomoVertex(B, B1);
    my2dTransHomoVertex(C, C1);

    drawArm(O, Af);

    my2dLoadIdentity();
    my2dTranslatef(Af[0], Af[1]);
    my2dRotatef(b);
    my2dTranslatef(-Af[0], -Af[1]);
    my2dTransHomoVertex(B1, Bf);
    my2dTransHomoVertex(C1, C2);
    drawArm(Af, Bf);

    my2dLoadIdentity();
    my2dTranslatef(Bf[0], Bf[1]);
    my2dRotatef(g);
    my2dTranslatef(-Bf[0], -Bf[1]);
```

```

    my2dTransHomoVertex(C2, Cf);
    drawArm(Bf, Cf);
}

```

Method II.

1. Consider BC to be a clock hand. Rotate BC around B by γ degrees:

$$C' = T(B)R(\gamma)T(-B)C. \quad (\text{EQ 39})$$

2. Consider ABC' to be a clock hand. Rotate ABC' around A by β degrees:

$$B' = T(A)R(\beta)T(-A)B; C'' = T(A)R(\beta)T(-A)C'. \quad (\text{EQ 40})$$

3. Again, consider $oAB'C''$ to be a clock hand. Rotate $oAB'C''$ around the origin by α degrees:

$$A_f = R(\alpha)A; \quad (\text{EQ 41})$$

$$B_f = R(\alpha)B' = R(\alpha)T(A)R(\beta)T(-A)B; \quad (\text{EQ 42})$$

$$C_f = R(\alpha)C'' = R(\alpha)T(A)R(\beta)T(-A)T(B)R(\gamma)T(-B)C. \quad (\text{EQ 43})$$

The corresponding code is as follows. Here *transDraw()* will first transform the vertices, and then draw the transformed vertices as a line segment.

```

// Method II: 2D robot arm transformations
public void transDrawArm2(float a, float b, float g) {

    my2dLoadIdentity();
    my2dRotatef(a);
    transDrawArm(O, A);
    my2dTranslatef(A[0], A[1]);
    my2dRotatef(b);
    my2dTranslatef(-A[0], -A[1]);
}

```

```

    transDrawArm(A, B);
    my2dTranslatef(B[0], B[1]);
    my2dRotatef(g);
    my2dTranslatef(-B[0], -B[1]);
    transDrawArm(B, C);
}

```

Method III.

1. Consider oA , AB , and BC as clock hands with the rotation axes at o , A , and B , respectively. Rotate oA by α degrees, AB by $(\alpha+\beta)$ degrees, and BC by $(\alpha+\beta+\gamma)$ degrees:

$$A_f = R(\alpha)A; B' = T(A)R(\alpha+\beta)T(-A)B; C' = T(B)R(\alpha+\beta+\gamma)T(-B)C. \quad (\text{EQ 44})$$

2. Translate AB' to A_fB_f :

$$B_f = T(A_f)T(-A)B' = T(A_f)R(\alpha+\beta)T(-A)B. \quad (\text{EQ 45})$$

Note that $T(-A)T(A) = I$, which is the identity matrix: $I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. Any matrix

multiplied by the identity matrix does not change. The vertex is translated by vector A , and then reversed back to its original position by translation vector $-A$.

3. Translate BC' to B_fC_f :

$$C_f = T(B_f)T(-B)C' = T(B_f)R(\alpha+\beta+\gamma)T(-B)C. \quad (\text{EQ 46})$$

The corresponding code is as follows.

```

// Method III: 2D robot arm transformations
public void transDrawArm3(float a, float b, float g) {
    float Af[] = new float[3];
    float Bf[] = new float[3];
    float Cf[] = new float[3];
}

```

```

my2dLoadIdentity();
my2dRotatef(a);
my2dTransHomoVertex(A, Af);
drawArm(O, Af);
my2dLoadIdentity();
my2dTranslatef(Af[0], Af[1]);
my2dRotatef(a + b);
my2dTranslatef(-A[0], -A[1]);
my2dTransHomoVertex(B, Bf);
drawArm(Af, Bf);
my2dLoadIdentity();
my2dTranslatef(Bf[0], Bf[1]);
my2dRotatef(a + b + g);
my2dTranslatef(-B[0], -B[1]);
my2dTransHomoVertex(C, Cf);
drawArm(Bf, Cf);
}

```

In the above examples, we use *Draw()* and *transDraw()*, which are implemented ourselves. The difference between the two functions are that *Draw()* will draw the two vertices as a line directly, whereas *transDraw()* will first transform the two vertices by the current matrix on the matrix stack, and then draw a line according to the transformed vertices. In OpenGL implementation, as we will see, *transDraw* is implied. That is, whenever we draw a primitive, the vertices of the primitive are always transformed by the current matrix on the MODELVIEW matrix stack, even though the transformation matrix multiplication is unseen. We will discuss this in detail later. The three different transformation are demonstrated in the following sample program (*J2_3_Robot2d.java*).

/* three different methods for 2D robot arm transformations */

```

import net.java.games.jogl.*;

public class J2_3_Robot2d extends J2_0_2DTransform {
    // homogeneous coordinates
    float O[] = {0, 0, 1};
    float A[] = {100, 0, 1};
    float B[] = {160, 0, 1};
    float C[] = {200, 0, 1};
    float a, b, g;
}

```

```
public void display(GLDrawable glDrawable) {

    gl.glClear(GL.GL_COLOR_BUFFER_BIT);

    a = a + 0.01f;
    b = b - 0.02f;
    g = g + 0.03f;

    gl.glColor3f(0, 1, 1);
    transDrawArm1(a, b, g);

    gl.glColor3f(1, 1, 0);
    transDrawArm2(-b, -g, a);

    gl.glColor3f(1, 0, 1);
    transDrawArm3(g, -a, -b);

    try {
        Thread.sleep(10);
    } catch (Exception ignore) {}
}

...; // Method I: 2D robot arm transformations
...; // Method II: 2D robot arm transformations
...; // Method III: 2D robot arm transformations

// transform the coordinates and then draw
private void transDrawArm(float C[], float H[]) {

    float End1[] = new float[3];
    float End2[] = new float[3];

    my2dTransHomoVertex(C, End1);
    // multiply the point with the matrix on the stack
    my2dTransHomoVertex(H, End2);

    // assuming z = w = 1;
    drawArm(End1, End2);
}

// draw the coordinates directly
public void drawArm(float C[], float H[]) {

    gl.glLineWidth(5);

    // assuming z = w = 1;
```

```

    gl.glBegin(GL.GL_LINES);
    gl.glVertex3fv(C);
    gl.glVertex3fv(H);
    gl.glEnd();
}

public static void main(String[] args) {
    J2_3_Robot2d f = new J2_3_Robot2d();

    f.setTitle("JOGL J2_3_Robot2d");
    f.setSize(500, 500);
    f.setVisible(true);
}
}

```

2.3 3D Transformation and Hidden-Surface Removal

2D transformation is a special case of 3D transformation where $z=0$. For example, a 2D point (x, y) is $(x, y, 0)$ in 3D, and a 2D rotation around the origin $R(\theta)$ is a 3D rotation around the z axis $R_z(\theta)$ (Fig. 2.9). The z axis is perpendicular to the display with the arrow pointing toward the viewer. We can assume the display to be a view of a 3D drawing box, which is projected along the z axis direction onto the 2D drawing area at $z=0$.

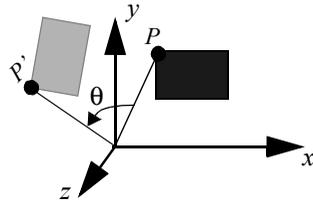


Fig. 2.9 A 3D rotation around z axis

2.3.1 3D Translation, Rotation, and Scaling

In 3D, for translation and scaling, we can translate or scale not only along the x and the y axis but also along the z axis. For rotation, in addition to rotating around the z axis, we can also rotate around the x axis and the y axis. In the homogeneous coordinates, the 3D transformation matrices for translation, rotation, and scaling are as follows:

$$\text{Translation: } T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (\text{EQ 47})$$

$$\text{Scaling: } S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (\text{EQ 48})$$

$$\text{Rotation around } x \text{ axis: } R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (\text{EQ 49})$$

$$\text{Rotation around } y \text{ axis: } R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (\text{EQ 50})$$

$$\text{Rotation around } z \text{ axis: } R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (\text{EQ 51})$$

For example, the 2D transformation Equation 41 can be replaced by the corresponding 3D matrices:

$$A_f = R_z(\alpha)A, \quad (\text{EQ 52})$$

where $A = \begin{bmatrix} A_x \\ A_y \\ A_z \\ I \end{bmatrix}$, $A_f = \begin{bmatrix} A_{fx} \\ A_{fy} \\ A_{fz} \\ I \end{bmatrix}$, and $A_z=0$. We can show that $A_{fz}=0$ as well.

2.3.2 Transformation in OpenGL

As an example, we will again implement in OpenGL the robot arm transformation MODELVIEW matrix stack to achieve the transformation. We consider the transformation to be a special case of 3D at $z=0$.

In OpenGL, all the vertices of a model are multiplied by the matrix on the top of the MODELVIEW matrix stack and then by the matrix on the top of the PROJECTION matrix stack before the model is scan-converted. Matrix multiplications are carried out on the top of the matrix stack automatically in the graphics system. The MODELVIEW matrix stack is used for geometric transformation. The PROJECTION matrix stack is used for viewing, which will be discussed later. Here, we explain how OpenGL handles the geometric transformations in the following example (*J2_4_Robot.java*, which implements *Method II* in Fig. 2.8.)

1. Specify that current matrix multiplications are carried out on the top of the MODELVIEW matrix stack:

```
gl.glMatrixMode (GL.GL_MODELVIEW);
```

2. Load the current matrix on the matrix stack with the identity matrix:

```
gl.glLoadIdentity ();
```

The identity matrix for 3D homogeneous coordinates is $I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

3. Specify the rotation matrix $R_z(\alpha)$, which will be multiplied by whatever on the current matrix stack already. The result replaces the matrix currently on the top of the stack. If the identity matrix is on the stack, then $IR_z(\alpha)=R_z(\alpha)$:

```
gl.glRotatef (alpha, 0.0, 0.0, 1.0);
```

4. Draw a robot arm — a line segment between point O and A . Before the model is scan-converted into the frame buffer, O and A will first be transformed by the matrix on the top of the MODELVIEW matrix stack, which is $R_z(\alpha)$. That is, $R_z(\alpha)O$ and $R_z(\alpha)A$ will be used to scan-convert the line (Equation 41):

```
drawArm (O, A);
```

5. In the following code section, we specify a series of transformation matrices, which in turn will be multiplied by whatever is already on the current matrix stack: I , $[I]R(\alpha)$, $[[I]R(\alpha)]T(A)$, $[[[I]R(\alpha)]T(A)]R(\beta)$, $[[[[I]R(\alpha)]T(A)]R(\beta)]T(-A)$. Before *drawArm* (A , B), we have $M = R(\alpha)T(A)R(\beta)T(-A)$ on the matrix stack, which corresponds to Equation 42:

```
gl.glPushMatrix();
gl.glLoadIdentity ();
gl.glRotatef (alpha, 0.0, 0.0, 1.0);
drawArm (O, A);

gl.glTranslatef (A[0], A[1], 0.0);
gl.glRotatef (beta, 0.0, 0.0, 1.0);
gl.glTranslatef (-A[0], -A[1], 0.0);
drawArm (A, B);
gl.glPopMatrix();
```

The matrix multiplication is always carried out on the top of the matrix stack. *glPushMatrix()* will move the stack pointer up one slot and duplicate the previous matrix so that the current matrix is the same as the matrix immediately below it on the stack. *glPopMatrix()* will move the stack pointer down one slot. The advantage of this mechanism is to separate the transformations of the current model between *glPushMatrix()* and *glPopMatrix()* from other transformations of models later.

Status of the OpenGL MODELVIEW matrix stack

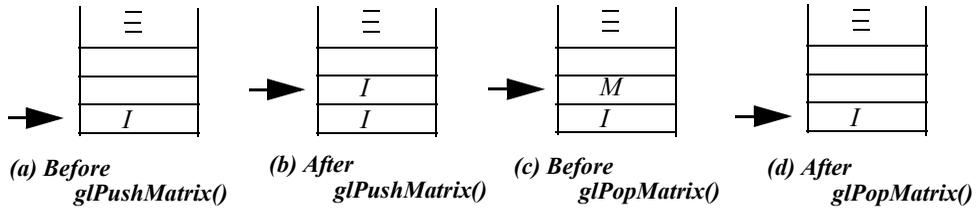


Fig. 2.10 Matrix stack operations with `glPushMatrix()` and `glPopMatrix()`

Let's look at the function `drawRobot()` in `J2_4_Robot.java` below. Fig. 2.10 shows what is on the top of the matrix stack, when `drawRobot()` is called once and then again. At `drawArm(B, C)` right before `glPopMatrix()`, the matrix on top of the stack is $M = R(\alpha)T(A)R(\beta)T(-A)T(B)R(\gamma)T(-B)$, which corresponds to Equation 43.

- Suppose we remove `glPushMatrix()` and `glPopMatrix()` from `drawRobot()`, if we call `drawRobot()` once, it appears fine. If we call it again, you will see that the matrix on the matrix stack is not an identity matrix. It is the previous matrix on the stack already (Fig. 2.11).

For beginners, it is a good idea to draw the state of the current matrix stack while you are reading the sample programs or writing your own programs. This will help you clearly understand what the transformation matrices are at different stages.

Status of the OpenGL MODELVIEW matrix stack

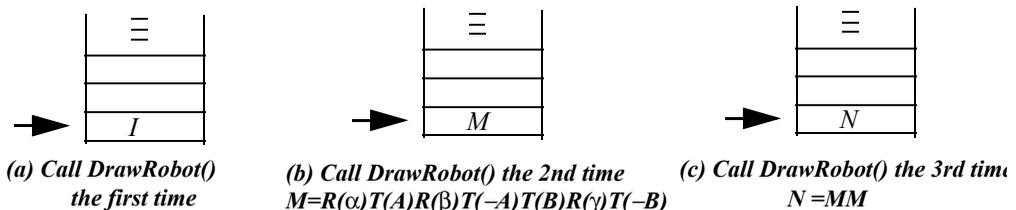


Fig. 2.11 Matrix stack operations without `glPushMatrix()` and `glPopMatrix()`

Methods I and III (Fig. 2.8) cannot be achieved using OpenGL transformations directly, because OpenGL provides matrix multiplications, but not the vertex coordinates after a vertex is transformed by the matrix. This means that all vertices are always fixed at their original locations. This method avoids floating point accumulation errors. We can use `glGetDoublev(GL_MODELVIEW_MATRIX, M[])` to get the current 16 values of the matrix on the top of the MODELVIEW stack, and multiply the coordinates by the current matrix to achieve the transformations for Methods I and III. Of course, you may implement your own matrix multiplications to achieve all the different transformation methods as well.

/* 2D robot transformation in OpenGL */

```
import net.java.games.jogl.*;

public class J2_4_Robot extends J2_3_Robot2d {
    public void display(GLDrawable glDrawable) {
        gl.glClear(GL_COLOR_BUFFER_BIT);

        a = a+0.1f;
        b = b-0.2f;
        g = g+0.3f;

        gl.glLineWidth(7f); // draw a wide line for arm
        drawRobot(A, B, C, a, b, g);

        try {
            Thread.sleep(10);
        } catch (Exception ignore) {}
    }

    void drawRobot(
        float A[],
        float B[],
        float C[],
        float alpha,
        float beta,
        float gama) {

        gl.glPushMatrix();

        gl.glColor3f(1, 1, 0);
```

```

gl.glRotatef(alpha, 0.0f, 0.0f, 1.0f);
// R_z(alpha) is on top of the matrix stack
drawArm(O, A);

gl.glColor3f(0, 1, 1);
gl.glTranslatef(A[0], A[1], 0.0f);
gl.glRotatef(beta, 0.0f, 0.0f, 1.0f);
gl.glTranslatef(-A[0], -A[1], 0.0f);
// R_z(alpha)T(A)R_z(beta)T(-A) is on top
drawArm(A, B);

gl.glColor3f(1, 0, 1);
gl.glTranslatef(B[0], B[1], 0.0f);
gl.glRotatef(gama, 0.0f, 0.0f, 1.0f);
gl.glTranslatef(-B[0], -B[1], 0.0f);
// R_z(alpha)T(A)R_z(beta)T(-A) is on top
drawArm(B, C);

gl.glPopMatrix();
}

public static void main(String[] args) {
    J2_4_Robot f = new J2_4_Robot();

    f.setTitle("JOGL J2_4 Robot");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}

```

2.3.3 Hidden-Surface Removal

Bounding volumes. We first introduce a simple method, called *bounding volume* or *minmax testing*, to determine visible 3D models without using a time-consuming hidden-surface removal algorithm. Here we assume that the viewpoint of our eye is at the origin and the models are in the negative z axis. If we render the models in the order of their distances to the viewpoint of the eye along z axis from the farthest to the closest, we will have correct overlapping of the models. We can build up a rectangular box (bounding volume) with the faces perpendicular to the x , y , or z axis to bound a 3D model and compare the minimum and maximum bounds in the z direction between boxes to decide which model should be rendered first. Using bounding volumes to decide the priority of rendering is also known as *minmax testing*. In addition to

visible-model determination, bounding volumes are also used for *collision detection*, which will be discussed later in this chapter.

The z-buffer (depth-buffer) algorithm. In OpenGL, to enable the hidden-surface removal (or visible-surface determination) mechanism, we need to enable the depth test once and then clear the depth buffer whenever we redraw a frame:

```
// enable zbuffer (depthbuffer) once
gl.glEnable(GL.GL_DEPTH_TEST);

// clear both frame buffer and zbuffer
gl.glClear(GL.GL_COLOR_BUFFER_BIT|GL.GL_DEPTH_BUFFER_BIT);
```

Corresponding to a frame buffer, the graphics system also has a z-buffer, or depth buffer, with the same number of entries. After *glClear()*, the z-buffer is initialized to the *z* value farthest from the viewpoint of our eye, and the frame buffer is initialized to the background color. When scan-converting a model (such as a polygon), before writing a pixel color into the frame buffer, the graphics system (the z-buffer algorithm) compares the pixel's *z* value to the corresponding *xy* coordinates' *z* value in the z-buffer. If the pixel is closer to the viewpoint, its *z* value is written into the z-buffer and its color is written into the frame buffer. Otherwise, the system moves on to considering the next pixel without writing into the buffers. The result is that, no matter what order the models are scan-converted, the image in the frame buffer only shows the pixels on the models that are not blocked by other pixels. In other words, the visible surfaces are saved in the frame buffer, and all the hidden surfaces are removed.

A pixel's *z* value is provided by the model at the corresponding *xy* coordinates. For example, given a polygon and the *xy* coordinates, we can calculate the *z* value according to the polygon's plane equation $z=f(x,y)$. Therefore, although scan-conversion is drawing in 2D, 3D calculations are needed to decide hidden-surface removal and others (as we will discuss in the future: lighting, texture mapping, etc.).

A plane equation in its general form is $ax + by + cz + I = 0$, where (a, b, c) corresponds to a vector perpendicular to the plane. A polygon is usually specified by a list of vertices. Given three vertices on the polygon, they all satisfy the plane equation and therefore we can find (a, b, c) and $z=-(ax + by + I)/c$. By the way, because the

cross-product of two edges of the polygon is perpendicular to the plane, it is proportional to (a, b, c) as well.

2.3.4 3D Models: Cone, Cylinder, and Sphere

Approximating a cone. In the example discussed at the end of last chapter (*J1_5_Circle.java*), we approximated a circle with subdividing triangles. If we raise the center of the circle along the z axis, we can approximate a cone, as shown in Fig. 2.12. Because the model is in 3D, we need to enable depth test to achieve hidden-surface removal. Also, we need to make sure that our model is contained within the defined coordinates (i.e., the viewing volume):

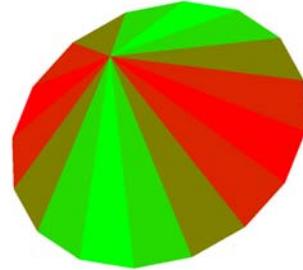


Fig. 2.12 A cone by subdivision
[See Color Plate 1]

```
gl.glOrtho(-w/2, w/2,
           -h/2, h/2, -w, w);
```

/* draw a cone by subdivision */

```
import net.java.games.jogl.*;

public class J2_5_Cone extends J1_5_Circle {
    public void reshape(GLDrawable glDrawable,
                       int x, int y, int w, int h) {
        WIDTH = w; HEIGHT = h;

        // enable depth buffer for hidden-surface removal
        gl.glEnable(GL.GL_DEPTH_TEST);

        gl.glMatrixMode(GL.GL_PROJECTION);
        gl.glLoadIdentity();

        // make sure the cone is within the viewing volume
```

```
gl.glOrtho(-w/2, w/2, -h/2, h/2, -w, w);

gl.glMatrixMode(GL.GL_MODELVIEW);
gl.glLoadIdentity();
}

public void display(GLDrawable glDrawable) {

    if ((cRadius>(WIDTH/2)) || (cRadius==1)) {
        flip = -flip;
        depth++;
        depth = depth%5;
    }

    cRadius += flip;

    // clear both frame buffer and zbuffer
    gl.glClear(GL.GL_COLOR_BUFFER_BIT|
               GL.GL_DEPTH_BUFFER_BIT);

    gl.glRotatef(1, 1, 1, 1); // accumulated on matrix
    // rotate 1 degree along vector (1, 1, 1)
    gl.glPushMatrix(); // not accumulated
    gl.glScaled(cRadius, cRadius, cRadius);
    drawCone();
    gl.glPopMatrix();

    try {
        Thread.sleep(10);
    } catch (Exception ignore) {}
}

private void subdivideCone(float v1[],
                          float v2[], int depth) {

    float v0[] = {0, 0, 0};
    float v12[] = new float[3];

    if (depth==0) {
        gl.glColor3f(v1[0]*v1[0], v1[1]*v1[1], v1[2]*v1[2]);

        drawtriangle(v1, v2, v0);
        // bottom cover of the cone

        v0[2] = 1; // height of the cone, the tip on z axis
        drawtriangle(v1, v2, v0); // side cover of the cone

        return;
    }
}
```

```

    }

    for (int i = 0; i<3; i++) {
        v12[i] = v1[i]+v2[i];
    }
    normalize(v12);

    subdivideCone(v1, v12, depth-1);
    subdivideCone(v12, v2, depth-1);
}

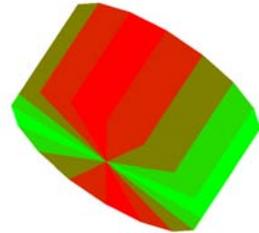
public void drawCone() {
    subdivideCone(cVdata[0], cVdata[1], depth);
    subdivideCone(cVdata[1], cVdata[2], depth);
    subdivideCone(cVdata[2], cVdata[3], depth);
    subdivideCone(cVdata[3], cVdata[0], depth);
}

public static void main(String[] args) {
    J2_5_Cone f = new J2_5_Cone();

    f.setTitle("JOGL J2_5_Cone");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}

```

Approximating a cylinder. If we can draw a circle at $z=0$, then draw another circle at $z=l$. If we connect the rectangles of the same vertices on the edges of the two circles, we have a cylinder, as shown in Fig. 2.13.



/* draw a cylinder by subdivision */

```

import net.java.games.jogl.*;

public class J2_6_Cylinder
extends J2_5_Cone {

```

Fig. 2.13 A cylinder by subdivision [See Color Plate 1]

```
public void display(GLDrawable glDrawable) {
    if ((cRadius>(WIDTH/2)) || (cRadius==1)) {
        flip = -flip;
        depth++;
        depth = depth%6;
    }
    cRadius += flip;

    // clear both frame buffer and zbuffer
    gl.glClear(GL.GL_COLOR_BUFFER_BIT|
               GL.GL_DEPTH_BUFFER_BIT);

    gl.glRotatef(1, 1, 1, 1);
    // rotate 1 degree along vector (1, 1, 1)
    gl.glPushMatrix();
    gl.glScaled(cRadius, cRadius, cRadius);
    drawCylinder();
    gl.glPopMatrix();

    try {
        Thread.sleep(20);
    } catch (Exception ignore) {}
}

private void subdivideCylinder(float v1[],
                              float v2[], int depth) {
    float v11[] = {0, 0, 0};
    float v22[] = {0, 0, 0};
    float v0[] = {0, 0, 0};
    float v12[] = new float[3];
    int i;

    if (depth==0) {
        gl.glColor3f(v1[0]*v1[0],
                    v1[1]*v1[1], v1[2]*v1[2]);

        for (i = 0; i<3; i++) {
            v22[i] = v2[i];
            v11[i] = v1[i];
        }

        drawtriangle(v1, v2, v0);
        // draw sphere at the cylinder's bottom

        v11[2] = v22[2] = v0[2] = 1.0f;
        drawtriangle(v11, v22, v0);
        // draw sphere at the cylinder's bottom
    }
}
```

```

    gl.glBegin(GL.GL_POLYGON);
    // draw the side rectangles of the cylinder
    gl.glVertex3fv(v11);
    gl.glVertex3fv(v22);
    gl.glVertex3fv(v2);
    gl.glVertex3fv(v1);
    gl.glEnd();

    return;
}

for (i = 0; i<3; i++) {
    v12[i] = v1[i]+v2[i];
}
normalize(v12);

subdivideCylinder(v1, v12, depth-1);
subdivideCylinder(v12, v2, depth-1);
}

public void drawCylinder() {
    subdivideCylinder(cVdata[0], cVdata[1], depth);
    subdivideCylinder(cVdata[1], cVdata[2], depth);
    subdivideCylinder(cVdata[2], cVdata[3], depth);
    subdivideCylinder(cVdata[3], cVdata[0], depth);
}

public static void main(String[] args) {
    J2_6_Cylinder f = new J2_6_Cylinder();

    f.setTitle("JOGL J2_6_Cylinder");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}

```

Approximating a sphere. Let's assume that we have an equilateral triangle with its three vertices (v_1, v_2, v_3) on a sphere and $|v_1|=|v_2|=|v_3|=1$. That is, the three vertices are unit vectors from the origin. We can see that $v_{12} = \text{normalize}(v_1 + v_2)$ is also on the sphere. We can further subdivide the triangle into four equilateral triangles, as shown in Fig. 2.14a. Example *J2_7_Sphere.java* uses this method to subdivide an octahedron (Fig. 2.14b) into a sphere, as shown in Fig. 2.14c.

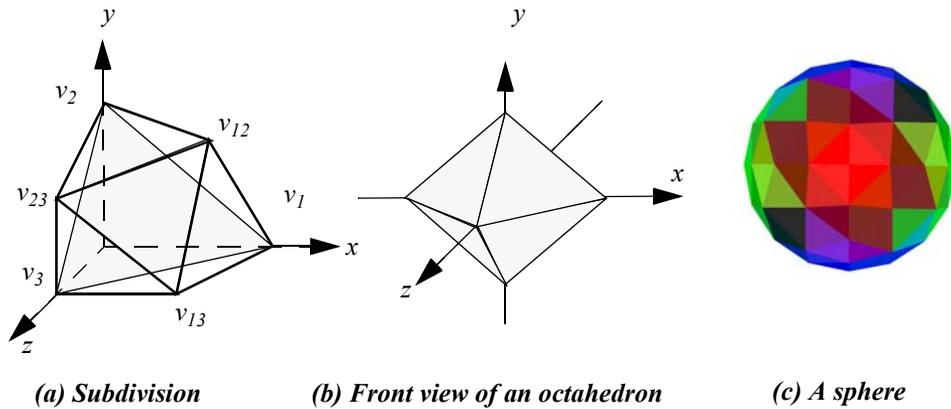


Fig. 2.14 Drawing a sphere through subdivision [See Color Plate 1]

/* draw a sphere by subdivision */

```
import net.java.games.jogl.*;

public class J2_7_Sphere extends J2_6_Cylinder {
    static float sVdata[][] = {
        { 1.0f, 0.0f, 0.0f }
        , { 0.0f, 1.0f, 0.0f }
        , { 0.0f, 0.0f, 1.0f }
        , { -1.0f, 0.0f, 0.0f }
        , { 0.0f, -1.0f, 0.0f }
        , { 0.0f, 0.0f, -1.0f }
    };
};

public void display(GLDrawable glDrawable) {
    if ((cRadius > (WIDTH / 2)) || (cRadius == 1)) {
        flip = -flip;
        depth++;
        depth = depth % 5;
    }
    cRadius += flip;
}
```

```
// clear both frame buffer and zbuffer
gl.glClearColor(GL.GL_COLOR_BUFFER_BIT |
                GL.GL_DEPTH_BUFFER_BIT);

gl.glRotatef(1, 1, 1, 1);
// rotate 1 degree along vector (1, 1, 1)
gl.glPushMatrix();
gl.glScalef(cRadius, cRadius, cRadius);
drawSphere();
gl.glPopMatrix();

try {
    Thread.sleep(20);
} catch (Exception ignore) {}
}

private void subdivideSphere(
    float v1[],
    float v2[],
    float v3[],
    long depth) {
    float v12[] = new float[3];
    float v23[] = new float[3];
    float v31[] = new float[3];
    int i;

    if (depth == 0) {
        gl.glColor3f(v1[0] * v1[0],
                    v2[1] * v2[1], v3[2] * v3[2]);
        drawtriangle(v1, v2, v3);

        return;
    }
    for (i = 0; i < 3; i++) {
        v12[i] = v1[i] + v2[i];
        v23[i] = v2[i] + v3[i];
        v31[i] = v3[i] + v1[i];
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    subdivideSphere(v1, v12, v31, depth - 1);
    subdivideSphere(v2, v23, v12, depth - 1);
    subdivideSphere(v3, v31, v23, depth - 1);
    subdivideSphere(v12, v23, v31, depth - 1);
}
```

```

public void drawSphere() {
    subdivideSphere(sVdata[0], sVdata[1], sVdata[2], depth);
    subdivideSphere(sVdata[0], sVdata[2], sVdata[4], depth);
    subdivideSphere(sVdata[0], sVdata[4], sVdata[5], depth);
    subdivideSphere(sVdata[0], sVdata[5], sVdata[1], depth);
    subdivideSphere(sVdata[3], sVdata[1], sVdata[5], depth);
    subdivideSphere(sVdata[3], sVdata[5], sVdata[4], depth);
    subdivideSphere(sVdata[3], sVdata[4], sVdata[2], depth);
    subdivideSphere(sVdata[3], sVdata[2], sVdata[1], depth);
}

public static void main(String[] args) {
    J2_7_Sphere f = new J2_7_Sphere();

    f.setTitle("JOGL J2_7_Sphere");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}

```

2.3.5 Composition of 3D Transformations

Example *J2_8_Robot3d.java* implements the robot arm in Example *J2_4_Robot.java* with 3D cylinders, as shown in Fig. 2.15. We also add one rotation around the y axis, so the robot arm moves in 3D.

/* 3D 3-segment arm transformation */

```

import net.java.games.jogl.*;

public class J2_8_Robot3d extends
    J2_7_Sphere {

    static float alpha = -30;
    static float beta = -30;
    static float gama = 60;
    static float aalpha = 1;
    static float abeta = 1;
    static float agama = -2;

```

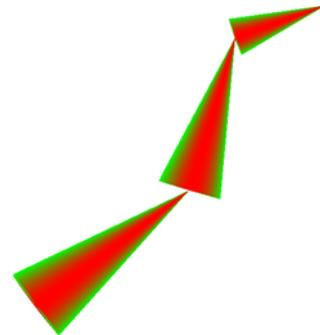


Fig. 2.15 A 3-segment robot arm [See Color Plate 2]

```
float O = 0;
float A = (float) WIDTH / 4;
float B = (float) 0.4 * WIDTH;
float C = (float) 0.5 * WIDTH;

public void display(GLDrawable glDrawable) {

    // for reshape purpose
    A = (float) WIDTH / 4;
    B = (float) 0.4 * WIDTH;
    C = (float) 0.5 * WIDTH;

    depth = 4;
    alpha += aalpha;
    beta += abeta;
    gama += agama;

    gl.glClear(GL.GL_COLOR_BUFFER_BIT |
               GL.GL_DEPTH_BUFFER_BIT);
    drawRobot(O, A, B, C, alpha, beta, gama);
}

void drawArm(float End1, float End2) {

    float scale;
    scale = End2 - End1;

    gl.glPushMatrix();

    // the cylinder lies in the z axis;
    // rotate it to lie in the x axis
    gl.glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
    gl.glScalef(scale / 5.0f, scale / 5.0f, scale);
    drawCylinder();

    gl.glPopMatrix();
}

void drawRobot(float O, float A, float B, float C,
               float alpha, float beta, float gama) {
    // the robot arm is rotating around y axis
    gl.glRotatef(1.0f, 0.0f, 1.0f, 0.0f);
    gl.glPushMatrix();

    gl.glRotatef(alpha, 0.0f, 0.0f, 1.0f);
    // R_z(alpha) is on top of the matrix stack
    drawArm(O, A);
}
```

```

gl.glTranslatef(A, 0.0f, 0.0f);
gl.glRotatef(beta, 0.0f, 0.0f, 1.0f);
// R_z(alpha)T_x(A)R_z(beta) is on top of the stack
drawArm(A, B);

gl.glTranslatef(B - A, 0.0f, 0.0f);
gl.glRotatef(gama, 0.0f, 0.0f, 1.0f);
// R_z(alpha)T_x(A)R_z(beta)T_x(B)R_z(gama) is on top
drawArm(B, C);

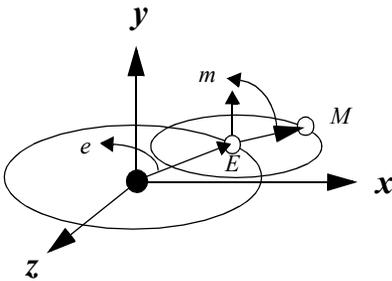
gl.glPopMatrix();
}

public static void main(String[] args) {
    J2_8_Robot3d f = new J2_8_Robot3d();

    f.setTitle("JOGL J2_8_Robot3d");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}

```

Example *J2_9_Solar.java* is a simplified solar system. The earth rotates around the sun and the moon rotates around the earth in the xz plane. Given the center of the earth at $E(x_e, y_e, z_e)$ and the center of the moon at $M(x_m, y_m, z_m)$, let's find the new centers after the earth rotates around the sun e degrees, and the moon rotates around the earth m degrees. The moon also revolves around the sun with the earth (Fig. 2.16).



The moon rotates first:

$$M' = T(E) R_y(m) T(-E) M;$$

$$E_f = R_y(e) E;$$

$$M_f = R_y(e) M';$$

The earth-moon rotates first:

$$E_f = R_y(e) E;$$

$$M' = R_y(e) M;$$

$$M_f = T(E_f) R_y(m) T(-E_f) M'$$

Fig. 2.16 Simplified solar system: a 2D problem in 3D

This problem is exactly like the clock problem in Fig. 2.5, except that the center of the clock is revolving around y axis as well. We can consider the moon rotating around the earth first, and then the moon and the earth as one object rotating around the sun.

In OpenGL, because we can draw a sphere at the center of the coordinates, the transformation would be simpler.

/* draw a simplified solar system */

```
import net.java.games.jogl.*;
import net.java.games.jogl.util.*;

public class J2_9_Solar extends J2_8_Robot3d {
    public void display(GLDrawable glDrawable) {
        depth = (cnt/100)%6;
        cnt++;

        gl.glClear(GL.GL_COLOR_BUFFER_BIT|
                  GL.GL_DEPTH_BUFFER_BIT);

        drawSolar(WIDTH/4, cnt, WIDTH/12, cnt);

        try {
            Thread.sleep(10);
        } catch (Exception ignore) {}
    }

    public void drawColorCoord(float xlen, float ylen,
                              float zlen) {
        GLUT glut = new GLUT();

        gl.glBegin(GL.GL_LINES);

        gl.glColor3f(1, 0, 0);

        gl.glVertex3f(0, 0, 0);
        gl.glVertex3f(0, 0, zlen);

        gl.glColor3f(0, 1, 0);

        gl.glVertex3f(0, 0, 0);
        gl.glVertex3f(0, ylen, 0);
    }
}
```

```
gl.glColor3f(0, 0, 1);

gl.glVertex3f(0, 0, 0);
gl.glVertex3f(xlen, 0, 0);

gl.glEnd();

// coordinate labels: X, Y, Z
gl.glPushMatrix();
gl.glTranslatef(xlen, 0, 0);
gl.glScalef(xlen/WIDTH, xlen/WIDTH, 1);
glut.glutStrokeCharacter(gl, GLUT.STROKE_ROMAN, 'X');
gl.glPopMatrix();

gl.glPushMatrix();
gl.glColor3f(0, 1, 0);
gl.glTranslatef(0, ylen, 0);
gl.glScalef(ylen/WIDTH, ylen/WIDTH, 1);
glut.glutStrokeCharacter(gl, GLUT.STROKE_ROMAN, 'Y');
gl.glPopMatrix();

gl.glPushMatrix();
gl.glColor3f(1, 0, 0);
gl.glTranslatef(0, 0, zlen);
gl.glScalef(zlen/WIDTH, zlen/WIDTH, 1);
glut.glutStrokeCharacter(gl, GLUT.STROKE_ROMAN, 'Z');
gl.glPopMatrix();
}

void drawSolar(float E, float e, float M, float m) {
    drawColorCoord(WIDTH/4, WIDTH/4, WIDTH/4);

    gl.glPushMatrix();

    gl.glRotatef(e, 0.0f, 1.0f, 0.0f);
    // rotating around the "sun"; proceed angle

    gl.glTranslatef(E, 0.0f, 0.0f);

    gl.glPushMatrix();
    gl.glScalef(WIDTH/20f, WIDTH/20f, WIDTH/20f);
    drawSphere();
    gl.glPopMatrix();

    gl.glRotatef(m, 0.0f, 1.0f, 0.0f);
```

```

    // rotating around the "earth"
    gl.glTranslatef(M, 0.0f, 0.0f);
    drawColorCoord(WIDTH/8f, WIDTH/8f, WIDTH/8f);
    gl.glScalef(WIDTH/40f, WIDTH/40f, WIDTH/40f);
    drawSphere();

    gl.glPopMatrix();
}

public static void main(String[] args) {
    J2_9_Solar f = new J2_9_Solar();

    f.setTitle("JOGL J2_9_Solar");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}

```

Next, we change the above solar system into a more complex system, which we call the *generalized solar system*. Now the earth is elevated along the y axis, and the moon is elevated along the axis from the origin toward the center of the earth, and the moon rotates around this axis as in Fig. 2.17. In other words, the moon rotates around the vector E . Given E and M and their rotation angles e and m , respectively, can we find the new coordinates of E_f and M_f ?

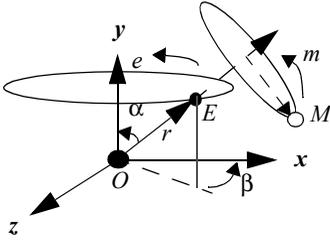
We cannot come up with the rotation matrix for the moon, M , immediately. However, we can consider E and M as one object and create the rotation matrix by several steps. Note that for M 's rotation around E , we do not really need to rotate E itself, but we use it as a reference to explain the rotation.

1. As shown in Fig. 2.17, the angle between the y axis and E is $\alpha = \text{arc cos}(y/r)$; the angle between the projection of E on the xz plane and the x axis is $\beta = \text{arc tg}(z/x)$; $r = \text{sqrt}(x^2 + y^2 + z^2)$.
2. Rotate M around the y axis by β degrees so that the new center of rotation E_l is in the xy plane:

$$M_l = R_y(\beta)M; E_l = R_y(\beta)E. \quad (\text{EQ 53})$$

$$r = \text{sqrt}(x^2 + y^2 + z^2);$$

$$\alpha = \text{arc cos } (y/r); \quad \beta = \text{arc tg } (z/x);$$



$E_f = R_y(e) E$; // the earth rotates around the y axis

$M_1 = R_y(\beta) M$; // the center of rotation OE is in the xy plane

$M_2 = R_z(\alpha) M_1$ // OE is along the y axis

$M_3 = R_y(m) M_2$; // the moon rotates along the y axis

$M_4 = R_z(-\alpha) M_3$; //OE returns to the xy plane

$M_5 = R_y(-\beta) M_4$; // OE returns to its original orientation

$M_f = R_y(e) M_5$; // the moon proceeds with the earth

$$M_f = R_y(e)R_y(-\beta) R_z(-\alpha) R_y(m) R_z(\alpha) R_y(\beta) M;$$

Fig. 2.17 Generalized solar system: a 3D problem

3. Rotate M_1 around the z axis by α degrees so that the new center of rotation E_2 is coincident with the y axis:

$$M_2 = R_z(\alpha)M_1; E_2 = R_z(\alpha)E_1. \tag{EQ 54}$$

4. Rotate M_2 around the y axis by m degree:

$$M_3 = R_y(m)M_2. \tag{EQ 55}$$

5. Rotate M_3 around the z axis by $-\alpha$ degree so that the center of rotation returns to the xz plane:

$$M_4 = R_z(-\alpha)M_3; E_1 = R_z(-\alpha)E_2. \tag{EQ 56}$$

6. Rotate M_4 around y axis by $-\beta$ degree so that the center of rotation returns to its original orientation:

$$M_5 = R_y(-\beta)M_4; E = R_y(-\beta)E_1. \tag{EQ 57}$$

7. Rotate M_5 around y axis e degree so that the moon proceeds with the earth around the y axis:

$$M_f = R_y(e)M_5; E_f = R_y(e)E. \quad (\text{EQ 58})$$

8. Putting the transformation matrices together, we have

$$M_f = R_y(e)R_y(-\beta) R_z(-\alpha) R_y(m) R_z(\alpha) R_y(\beta) M. \quad (\text{EQ 59})$$

Again, in OpenGL, we start with the sphere at the origin. The transformation is simpler. The following code demonstrates the generalized solar system. The result is shown in Fig. 2.18. Incidentally, `glRotatef(m, x, y, z)` specifies a single matrix that rotates a point along the vector (x, y, z) by m degrees. Now, we know that the matrix is equal to $R_y(-\beta) R_z(-\alpha) R_y(m) R_z(\alpha) R_y(\beta)$.

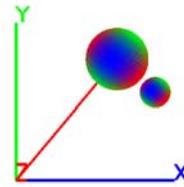


Fig. 2.18 Generalized solar system [See Color Plate 2]

/* draw a generalized solar system */

```
import net.java.games.jogl.*;

public class J2_10_GenSolar extends J2_9_Solar {
    static float tiltAngle = 40;

    void drawSolar(float earthDistance,
                  float earthAngle,
                  float moonDistance,
                  float moonAngle) {

        // Global coordinates
        gl.glLineWidth(6);
        drawColorCoord(WIDTH/4, WIDTH/4, WIDTH/4);

        gl.glPushMatrix();
```

```
gl.glRotatef(earthAngle, 0.0f, 1.0f, 0.0f);
// rotating around the "sun"; proceed angle
gl.glRotatef(tiltAngle, 0.0f, 0.0f, 1.0f);
// tilt angle, angle between the center line and y axis
gl.glBegin(GL.GL_LINES);
gl.glVertex3f(0.0f, 0.0f, 0.0f);
gl.glVertex3f(0.0f, earthDistance, 0.0f);
gl.glEnd();

gl.glTranslatef(0.0f, earthDistance, 0.0f);
gl.glLineWidth(2);

gl.glPushMatrix();
drawColorCoord(WIDTH/6, WIDTH/6, WIDTH/6);
gl.glScalef(WIDTH/20, WIDTH/20, WIDTH/20);
drawSphere();
gl.glPopMatrix();

gl.glRotatef(moonAngle, 0.0f, 1.0f, 0.0f);
// rotating around the "earth"
gl.glTranslatef(moonDistance, 0.0f, 0.0f);
gl.glLineWidth(3);
drawColorCoord(WIDTH/8, WIDTH/8, WIDTH/8);
gl.glScalef(WIDTH/40, WIDTH/40, WIDTH/40);
drawSphere();

gl.glPopMatrix();
}

public static void main(String[] args) {

    J2_10_GenSolar f = new J2_10_GenSolar();

    f.setTitle("JOGL J2_10_GenSolar");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}
```

The generalized solar system corresponds to a top that rotates and proceeds as shown in Fig. 2.19b. The rotating angle is m and the proceeding angle is e . The earth E is a point along the center of the top, and the moon M can be a point on the edge of the top. We learned to draw a cone in OpenGL. We can transform the cone to achieve the motion of a top. In the following example (*J2_11_ConeSolar.java*), we have a top that rotates and proceeds and a sphere that rotates around the top (Fig. 2.19c).

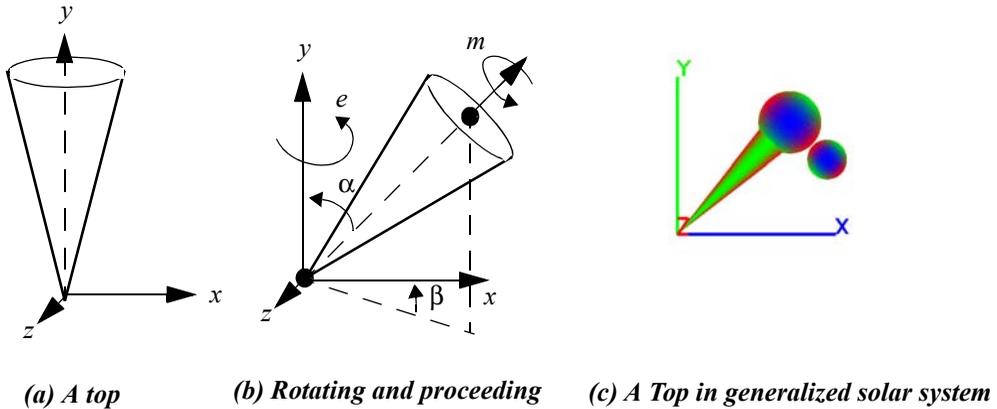


Fig. 2.19 A top rotates and proceeds [See Color Plate 2]

/* draw a cone solar system */

```
public class J2_11_ConeSolar extends J2_10_GenSolar {
    void drawSolar(float E, float e, float M, float m) {
        // Global coordinates
        gl.glLineWidth(6);
        drawColorCoord(WIDTH / 4, WIDTH / 4, WIDTH / 4);

        gl.glPushMatrix();
        gl.glRotatef(e, 0.0f, 1.0f, 0.0f);
        // rotating around the "sun"; proceed angle
        gl.glRotatef(alpha, 0.0f, 0.0f, 1.0f); // tilt angle
        gl.glTranslatef(0.0f, E, 0.0f);
        gl.glPushMatrix();
        gl.glScalef(WIDTH / 20, WIDTH / 20, WIDTH / 20);
        drawSphere();
        gl.glPopMatrix();
        gl.glPushMatrix();
        gl.glScalef(E / 8, E, E / 8);
        gl.glRotatef(90, 1.0f, 0.0f, 0.0f); // orient the cone
        drawCone();
        gl.glPopMatrix();

        gl.glRotatef(m, 0.0f, 1.0f, 0.0f);
    }
}
```

```
// rotating around the "earth"
gl.glTranslatef(M, 0.0f, 0.0f);
gl.glLineWidth(4);
drawColorCoord(WIDTH / 8, WIDTH / 8, WIDTH / 8);
gl.glScalef(E / 8, E / 8, E / 8);
drawSphere();
gl.glPopMatrix();
}

public static void main(String[] args) {

    J2_11_ConeSolar f = new J2_11_ConeSolar();

    f.setTitle("JOGL J2_11_ConeSolar");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}
```

2.3.6 Collision Detection

To avoid two models in an animation penetrating each other, we can use their bounding volumes to decide their physical distances and collision. Of course, the bounding volume can be in a different shape other than a box, such as a sphere. If the distance between the centers of the two spheres is bigger than the summation of the two radii of the spheres, we know that the two models do not collide with each other. We may use multiple spheres with different radii to more accurately bound a model, but the collision detection would be more complex. Of course, we may also detect collisions directly without using bounding volumes, which is likely much more complex and time consuming.

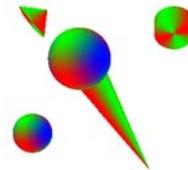


Fig. 2.20 Collision detection
[See Color Plate 2]

We can modify the above example to have three moons (a cylinder, a sphere, and a cone) that rotate around the earth in different directions and collide with one another changing the directions of rotation (Fig. 2.20). If we use a sphere as a bounding

volume, the problem becomes how to find the centers of the bounding spheres. We know that each moon is transformed from the origin. If we know the current matrix on the matrix stack at the point we draw a moon, we can multiply the matrix with the origin (0, 0, 0, 1) to find the center of the moon. Because at the origin x , y , and z are 0s, we only need to retrieve the last column in the matrix, which is shown in the following example (*J2_11_coneSolarCollision.java*). Collision detection is then decided by the distances among the moons' centers. If a distance is shorter than a predefined threshold, the two moons will change their directions of rotation around the earth.

/* draw a cone solar system with collisions of the moons */

```
import java.lang.Math;
import net.java.games.jogl.*;

public class J2_11_ConeSolarCollision extends
    J2_11_ConeSolar {
    //direction and speed of rotation
    static float coneD = WIDTH/110;
    static float sphereD = -WIDTH/64;
    static float cylinderD = WIDTH/300f;
    static float spherem = 120, cylinderm = 240;
    static float tmpD = 0, conem = 0;

    // centers of the objects
    static float[] coneC = new float[3];
    static float[] sphereC = new float[3];
    static float[] cylinderC = new float[3];

    // current matrix on the matrix stack
    static float[] currM = new float[16];

    void drawSolar(float E, float e, float M, float m) {

        // Global coordinates
        gl.glLineWidth(8);
        drawColorCoord(WIDTH/4, WIDTH/4, WIDTH/4);

        gl.glPushMatrix(); {
            gl.glRotatef(e, 0.0f, 1.0f, 0.0f);
            // rotating around the "sun"; proceed angle
            gl.glRotatef(alpha, 0.0f, 0.0f, 1.0f); // tilt angle
            gl.glTranslatef(0.0f, E, 0.0f);
```

```
gl.glPushMatrix();
gl.glScalef(WIDTH/20, WIDTH/20, WIDTH/20);
drawSphere();
gl.glPopMatrix();

gl.glPushMatrix();
gl.glScalef(E/8, E, E/8);
gl.glRotatef(90, 1.0f, 0.0f, 0.0f);

// orient the cone
drawCone();
gl.glPopMatrix();

gl.glPushMatrix();
cylinderm = cylinderm+cylinderD;
gl.glRotatef(cylinderm, 0.0f, 1.0f, 0.0f);
// rotating around the "earth"
gl.glTranslatef(M*2, 0.0f, 0.0f);
gl.glLineWidth(4);
drawColorCoord(WIDTH/8, WIDTH/8, WIDTH/8);
gl.glScalef(E/8, E/8, E/8);
drawCylinder();
// retrieve the center of the cylinder
// the matrix is stored column major left to right
gl.glGetFloatv(GL.GL_MODELVIEW_MATRIX, currM);
cylinderC[0] = currM[12];
cylinderC[1] = currM[13];
cylinderC[2] = currM[14];
gl.glPopMatrix();

gl.glPushMatrix();
spherem = spherem+sphereD;
gl.glRotatef(spherem, 0.0f, 1.0f, 0.0f);
// rotating around the "earth"
gl.glTranslatef(M*2, 0.0f, 0.0f);
drawColorCoord(WIDTH/8, WIDTH/8, WIDTH/8);
gl.glScalef(E/8, E/8, E/8);
drawSphere();
// retrieve the center of the sphere
gl.glGetFloatv(GL.GL_MODELVIEW_MATRIX, currM);
sphereC[0] = currM[12];
sphereC[1] = currM[13];
sphereC[2] = currM[14];
gl.glPopMatrix();

gl.glPushMatrix();
conem = conem+coneD;
gl.glRotatef(conem, 0.0f, 1.0f, 0.0f);
// rotating around the "earth"
```

```

    gl.glTranslatef(M*2, 0.0f, 0.0f);
    drawColorCoord(WIDTH/8, WIDTH/8, WIDTH/8);
    gl.glScalef(E/8, E/8, E/8);
    drawCone();
    // retrieve the center of the cone
    gl.glGetFloatv(GL.GL_MODELVIEW_MATRIX, currM);
    coneC[0] = currM[12];
    coneC[1] = currM[13];
    coneC[2] = currM[14];
    gl.glPopMatrix();
}
gl.glPopMatrix();

if (distance(coneC, sphereC)<E/5) {
    // collision detected, swap the rotation directions
    tmpD = coneD;
    coneD = sphereD;
    sphereD = tmpD;
}

if (distance(coneC, cylinderC)<E/5) {
    // collision detected, swap the rotation directions
    tmpD = coneD;
    coneD = cylinderD;
    cylinderD = tmpD;
}

if (distance(cylinderC, sphereC)<E/5) {
    // collision detected, swap the rotation directions
    tmpD = cylinderD;
    cylinderD = sphereD;
    sphereD = tmpD;
}
}

// distance between two points
float distance(float[] c1, float[] c2) {
    float tmp = (c2[0]-c1[0])*(c2[0]-c1[0])+
                (c2[1]-c1[1])*(c2[1]-c1[1])+
                (c2[2]-c1[2])*(c2[2]-c1[2]);

    return ((float)Math.sqrt(tmp));
}

public static void main(String[] args) {
    J2_11_ConeSolarCollision f =
        new J2_11_ConeSolarCollision();
}

```

```
f.setTitle("JOGL J2_11_ConeSolarCollision");  
f.setSize(WIDTH, HEIGHT);  
f.setVisible(true);  
}  
}
```

2.4 Viewing

The display has its device coordinate system in pixels, and our model has its (virtual) modeling coordinate system in which we specify and transform our model. We need to consider the relationship between the modeling coordinates and the device coordinates so that our virtual model will appear as an image on the display. Therefore, we need a *viewing* transformation — the mapping of an area or volume in the modeling coordinates to an area in the display device coordinates.

2.4.1 2D Viewing

In 2D viewing, we specify a rectangular area called the *modeling window* in the modeling coordinates and a display rectangular area called the *viewport* in the device coordinates (Fig. 2.21). The modeling window defines what is to be viewed; the viewport defines where the image appears. Instead of transforming a model in the modeling window to a model in the display viewport directly, we can first transform the modeling window into a square with the lower-left corner at $(-1, -1)$ and the upper-right corner at $(1, 1)$. The coordinates of the square are called the *normalized* coordinates. Clipping of the model is then calculated in the normalized coordinates against the square. After that, the normalized coordinates are scaled and translated to the device coordinates.

We should understand that the matrix that transforms the modeling window to the square will also transform the models in the modeling coordinates to the corresponding models in the normalized coordinates. Similarly, the matrix that transforms the square to the viewport will also transform the models accordingly. The process (or pipeline) in 2D viewing is shown in Fig. 2.21. Through normalization, the clipping algorithm avoids dealing with the changing sizes of the modeling window and the device viewport.

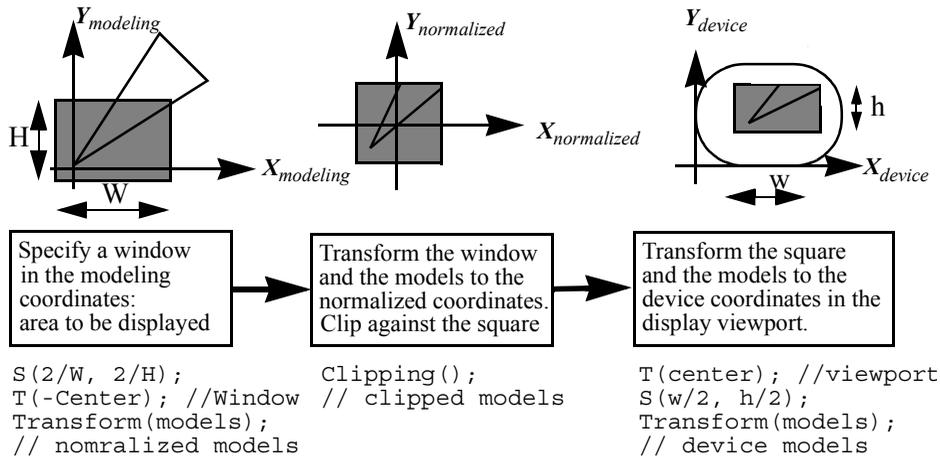


Fig. 2.21 2D viewing pipeline

2.4.2 3D Viewing

The display is a 2D viewport, and our model can be in 3D. In 3D viewing, we need to specify a viewing volume, which determines a projection method (*parallel* or *perspective*) — for how 3D models are projected into 2D. The projection lines go from the vertices in the 3D models to the projected vertices in the projection plane — a 2D *view plane* that corresponds to the viewport. A parallel projection has all the projection lines parallel. A perspective projection has all the projection lines converging to a point named the *center of projection*. The center of projection is also called the *viewpoint*. You may consider that your eye is at the viewpoint looking into the viewing volume. Viewing is analogous to taking a photograph with a camera. The object in the outside world has its own 3D coordinate system, the film in the camera has its own 2D coordinate system. We specify a viewing volume and a projection method by pointing and adjusting the zoom.

As shown in Fig. 2.22, the viewing volume for the parallel projection is like a box. The result of the parallel projection is a less realistic view but can be used for exact measurements. The viewing volume for the perspective projection is like a truncated pyramid, and the result looks more realistic in many cases, but does not preserve sizes in the display — objects further away are smaller.

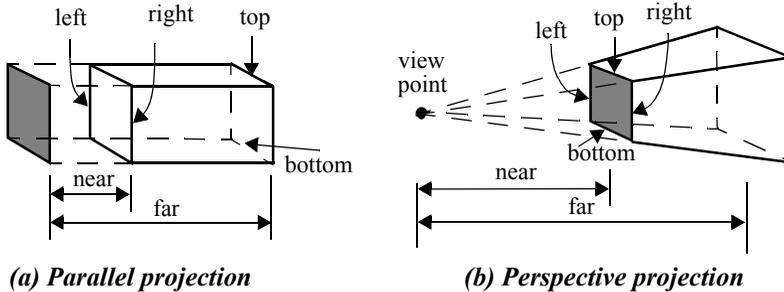


Fig. 2.22 View volumes and projection methods

In the following, we use the OpenGL system as an example to demonstrate how 3D viewing is achieved. The OpenGL viewing pipeline includes normalization, clipping, perspective division, and viewport transformation (Fig. 2.23). Except for clipping, all other transformation steps can be achieved by matrix multiplications. Therefore, viewing is mostly achieved by geometric transformation. In the OpenGL system, these transformations are achieved by matrix multiplications on the PROJECTION matrix stack.

Specifying a viewing volume. A parallel projection is called an *orthographic projection* if the projection lines are all perpendicular to the view plane. `glOrtho(left, right, bottom, top, near, far)` specifies an orthographic projection as shown in Fig. 2.22a. `glOrtho()` also defines six plane equations that cover the orthographic viewing volume: $x=\text{left}$, $x=\text{right}$, $y=\text{bottom}$, $y=\text{top}$, $z=-\text{near}$, and $z=-\text{far}$. We can see that (left, bottom, -near) and (right, top, -near) specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane. Similarly, (left, bottom, -far) and (right, top, -far) specify the (x, y, z) coordinates of the lower-left and upper-right corners of the far clipping plane.

`glFrustum(left, right, bottom, top, near, far)` specifies a perspective projection as shown in Fig. 2.22b. `glFrustum()` also defines six planes that cover the perspective viewing volume. We can see that (left, bottom, -near) and (right, top, -near) specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane. The far clipping plane is a cross section at $z=-\text{far}$ with the projection lines converging to the viewpoint, which is fixed at the origin looking down the negative z axis.

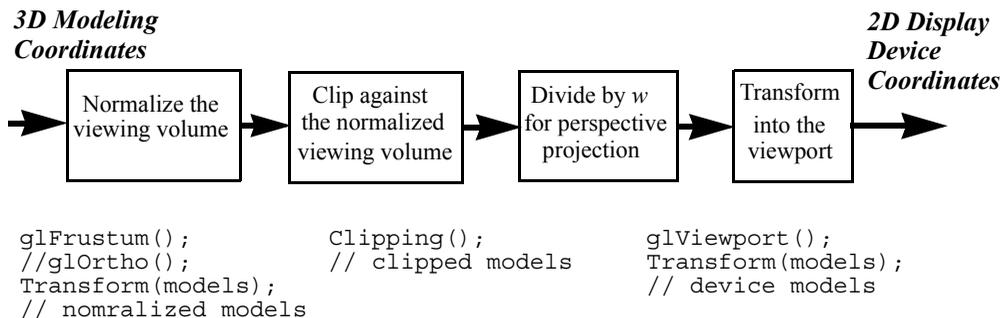


Fig. 2.23 3D viewing pipeline

As we can see, both *glOrtho()* and *glFrustum()* specify viewing volumes oriented with left and right edges on the near clipping plane parallel to *y* axis. In general, we use a vector *up* to represent the orientation of the viewing volume, which when projected on to the near clipping plane is parallel to the left and right edges.

Normalization. Normalization transformation is achieved by matrix multiplication on the PROJECTION matrix stack. In the following code section, we first load the identity matrix onto the top of the matrix stack. Then, we multiply the identity matrix by a matrix specified by *glOrtho()*.

```

// hardware set to use projection matrix stack
gl.glMatrixMode (GL.GL_PROJECTION);
gl.glLoadIdentity ();
gl.glOrtho(-Width/2,Width/2,-Height/2,Height/2,-1.0, 1.0);
  
```

In OpenGL, *glOrtho()* actually specifies a matrix that transforms the specified viewing volume into a *normalized* viewing volume, which is a cube with six clipping planes as shown in Fig. 2.24 ($x=1$, $x=-1$, $y=1$, $y=-1$, $z=1$, and $z=-1$). *glOrtho(l, r, b, t, n, f)* is equivalent to the following matrix expression:

$$S(-2/(r-l), -2/(t-b), -2/(f-n))T(-(r+l)/2, -(t+b)/2, (f+n)/2); \quad (\text{EQ 60})$$

Therefore, instead of calculating the clipping and projection directly, the normalization transformation is carried out first to simplify the clipping and the projection. Similarly, *glFrustum()* also specifies a matrix that transforms the perspective viewing volume into a normalized viewing volume as in Fig. 2.24. Here a division is needed to map the homogeneous coordinates into 3D coordinates. In OpenGL, a 3D vertex is represented by (x, y, z, w) and transformation matrices are 4×4 matrices. When $w=1$, (x, y, z) represents the 3D coordinates of the vertex. If $w=0$, (x, y, z) represents a direction. Otherwise, $(x/w, y/w, z/w)$ represents the 3D coordinates. A perspective division is needed simply because after the *glFrustum()* matrix transformation, $w \neq 1$. In OpenGL, the perspective division is carried out after clipping.

Clipping. Because *glOrtho()* and *glFrustum()* both transform their viewing volumes into a normalized viewing volume, we only need to develop one clipping algorithm. Clipping is carried out in homogeneous coordinates. Therefore, all vertices of the models are first transformed into the normalized viewing coordinates, clipped against the planes of the normalized viewing volume ($x=-w, x=w, y=-w, y=w, z=-w, z=w$), and then transformed and projected into the 2D viewport.

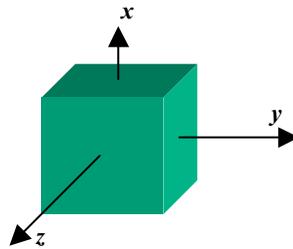


Fig. 2.24 Normalized viewing volume — a cube with $(-1$ to $1)$ along each axis

Perspective division. The perspective normalization transformation *glFrustum()* results in homogenous coordinates with $w \neq 1$. Clipping is carried out in homogeneous coordinates. However, a division for all the coordinates of the model $(x/w, y/w, z/w)$ is needed to transform homogeneous coordinates into 3D coordinates.

Viewport transformation. All vertices are kept in 3D. We need the z values to calculate hidden-surface removal. From the normalized viewing volume after dividing by w , the viewport transformation calculates each vertex's (x, y, z) corresponding to the pixels in the viewport and invokes scan-conversion algorithms to draw the model into the viewport. Projecting into 2D is nothing more than ignoring the z values when

scan-converting the model's pixels into the frame buffer. It is not necessary but we may consider that the projection plane is at $z=0$. In Fig. 2.22, the shaded projection planes are arbitrarily specified.

Summary of the viewing pipeline. Before scan-conversion, an OpenGL model will go through the following transformation and viewing processing steps:

- *Modeling*: Each vertex of the model will be transformed by the current matrix on the top of the MODELVIEW matrix stack.
- *Normalization*: After the above MODELVIEW transformation, each vertex will be transformed by the current matrix on the top of the PROJECTION matrix stack.
- *Clipping*: Each primitive (point, line, polygon, etc.) is clipped against the clipping planes in homogeneous coordinates.
- *Perspective division*: All primitives are transformed from homogeneous coordinates into Cartesian coordinates.
- *Viewport transformation*: The model is scaled and translated into the viewport for scan-conversion.

2.4.3 3D Clipping Against a Cube

Clipping a 3D point against a cube can be done in six comparisons. If we represent a point by its six comparisons in six bits, we can easily decide a 3D line clipping.

```
Bit 6 = 1 if x<left;  
Bit 5 = 1 if x>right;  
Bit 4 = 1 if y<bottom;  
Bit 3 = 1 if y>top;  
Bit 2 = 1 if z<near;  
Bit 1 = 1 if z>far;
```

If the two end points of a line's 6 bits are 000000 (the logic OR is equal to zero), then the end points of the line are inside the cube. If there is a same bit in the two end points is not equal to zero (the logic AND is not equal to zero), then the two end points are outside the viewing volume. Otherwise, we can find the lines intersections with the cube. Given two end points (x_0, y_0, z_0) and (x_1, y_1, z_1) , the parametric line equation can be represented as:

$$x = x_0 + t(x_1 - x_0) \quad (\text{EQ 61})$$

$$y = y_0 + t(y_1 - y_0) \quad (\text{EQ 62})$$

$$z = z_0 + t(z_1 - z_0) \quad (\text{EQ 63})$$

Now if any bit is not equal to zero, say Bit 2 = 1, then $z = \text{near}$, and we can find t in Equation 63. and therefore find the intersection point (x, y, z) according to Equation 61 and Equation 62.

For a polygon in 3D, we can extend the above line clipping algorithm to walk around the edges of the polygon against the cube. If a polygon's edge lies inside the clipping volume, the vertices are accepted for the new polygon. Otherwise, we can throw out all vertices outside a volume boundary plane, cut the two edges that go out of and into a boundary plane, and generate new vertices along a boundary plane between the two edges to replace the vertices that are outside a boundary plane. The clipped polygon has all vertices in the viewing volume after the six boundary planes are processed.

Clipping against the viewing volume is part of OpenGL view pipeline discussed earlier. Actually, clipping against an arbitrary plane can be calculated similarly as discussed below.

2.4.4 Clipping Against an Arbitrary Plane

A plane equation in general form can be expressed as follows:

$$ax + by + cz + d = 0 \quad (\text{EQ 64})$$

We can clip a point against the plane equation. Given a point (x_0, y_0, z_0) , if $ax_0 + by_0 + cz_0 + d \geq 0$, then the point is accepted. Otherwise it is clipped. For an edge, if the two end points are not accepted or clipped, we can find the intersection of the edge with the plane by putting Equation 61, Equation 62, and Equation 63 into Equation 64. Again, we can walk around the vertices of a polygon to clip against the plane.

OpenGL has a function *glClipPlane()* that allows specifying and clipping plane. You can enable the corresponding clipping plane so that objects below the clipping plane will be clipped.

/* clipping against an arbitrary plane.*/*

```
import java.lang.Math; // import net.java.games.jogl.*;
import javax.media.opengl.*;

public class J2_12_Clipping extends J2_11_ConeSolarCollision {

    static double[] eqn = new double[4];
    // plane equation ax+by+cz+d = 0

    public void display(GLAutoDrawable glDrawable) {

        //1. specify plane equation x = 0;
        eqn[0] = 1;
        //2. tell OpenGL system eqn is a clipping plane
        gl.glClipPlane(GL.GL_CLIP_PLANE0, eqn, 0);
        //3. Enable clipping of the plane.
        gl.glEnable(GL.GL_CLIP_PLANE0);

        super.display(glDrawable);
    }

    public static void main(String[] args) {
        J2_12_Clipping f = new J2_12_Clipping();

        f.setTitle("JOGL J2_12_Clipping");
        f.setSize(WIDTH, HEIGHT);
        f.setVisible(true);
    }
}
```

2.4.5 The Logical Orders of Transformation Steps

Modeling and viewing transformations are carried out by the OpenGL system automatically. For programmers, it is more practical to understand how to specify a

viewing volume through *glOrtho()* or *glFrustum()* on the PROJECTION matrix stack and to make sure that the model is in the viewing volume after being transformed by the current matrix on the MODELVIEW matrix stack. The PROJECTION matrix is multiplied with the MODELVIEW matrix, and the result is used to transform (normalize) the original model's vertices. The final matrix, if you view it from how it is constructed, represents an expression or queue of matrices from left-most where you specify normalization matrix to right-most where you specify a vertex in drawing.

When we analyze a model's transformation steps, logically speaking, the order of transformation steps is from right to left in the matrix expression. However, we can look at the matrix expression from left to right if our logical is transforming the projection (camera) instead of the model. We will discuss these two different logical reasoning orders here.

The following demonstrates how to specify the modelview and projection matrices on the two stacks in the example *J2_12_RobotSolar.java*, as shown in Fig. 2.25. Here the logical reasoning is from where we specify the model to where we specify the projection matrix.

1. In *display()*, a robot arm is calculated at the origin of the modeling coordinates.
2. As we discussed before, although the matrices are multiplied from the top-down transformation commands, when we analyze a model's transformations, logically speaking, the order of transformation steps are bottom-up from the closest transformation above the drawing command on the MODELVIEW matrix stack to where we specify the viewing volume on the PROJECTION matrix stack.
3. OpenGL provides PROJECTION and MODELVIEW matrix stacks to facilitate viewing and transformation separately, which is a nice separation and logical structure. Theoretically, we do not have to require two pieces of hardware, because the matrix on top of the PROJECTION matrix stack and the matrix on top of the MODELVIEW matrix stack are multiplied together to transform the models into

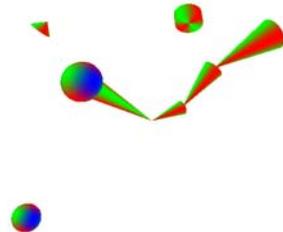


Fig. 2.25 Viewing in 3D [See Color Plate 2]

the canonical viewing volume. Therefore, we can view these two matrices as one matrix expression, and some of the transformations can be on either of the matrix stacks. The following transformation step is an example.

4. In *Reshape()*, the robot arm is translated along *z* axis $-(zNear + zFar)/2$ in order to be put in the middle of the viewing volume. This translation here can be the first matrix in the MODELVIEW matrix expression or the last matrix in the PROJECTION matrix expression.
5. *glOrtho()* or *glFrustum()* specify the viewing volume. The models in the viewing volume will appear in the viewport area on the display.
6. *glViewport()* in *Reshape()* specifies the rendering area within the display window. The viewing volume will be projected into the viewport area. When we reshape the drawing area, the viewport aspect ratio (*w/h*) changes accordingly. We may specify a different viewport using *glViewport()* and draw into that area. In other words, we may have multiple viewports with different renderings in each display, which will be discussed later.

/* 3D transformation and viewing */

```
import net.java.games.jogl.*;

public class J2_12_RobotSolar extends
    J2_11_ConeSolarCollision {

    public void reshape(
        GLDrawable glDrawable,
        int x,
        int y,
        int w,
        int h) {

        WIDTH = w;
        HEIGHT = h;

        // enable zbuffer for hidden-surface removal
        gl.glEnable(GL.GL_DEPTH_TEST);

        // specify the drawing area within the frame window
        gl.glViewport(0, 0, w, h);
    }
}
```

```
// projection is carried on the projection matrix
gl.glMatrixMode(GL.GL_PROJECTION);
gl.glLoadIdentity();
// specify perspective projection using glFrustum
gl.glFrustum(-w/4, w/4, -h/4, h/4, w/2, 4*w);

// put the models at the center of the viewing volume
gl.glTranslatef(0, 0, -2*w);

// transformations are on the modelview matrix
gl.glMatrixMode(GL.GL_MODELVIEW);
gl.glLoadIdentity();
}

public void display(GLDrawable glDrawable) {

    cnt++;
    depth = (cnt/100)%6;

    gl.glClear(GL.GL_COLOR_BUFFER_BIT|
               GL.GL_DEPTH_BUFFER_BIT);

    if (cnt%60==0) {
        aalpha = -aalpha;
        abeta = -abeta;
        agama = -agama;
    }
    alpha += aalpha;
    beta += abeta;
    gama += agama;

    drawRobot(O, A, B, C, alpha, beta, gama);

    try {
        Thread.sleep(15);
    } catch (Exception ignore) {}
}

void drawRobot (float O, float A, float B, float C,
                float alpha, float beta, float gama) {

    gl.glLineWidth(8);
    drawColorCoord(WIDTH/4, WIDTH/4, WIDTH/4);

    gl.glPushMatrix();

    gl.glRotatef(cnt, 0, 1, 0);
```

```
gl.glRotatef(alpha, 0, 0, 1);
// R_z(alpha) is on top of the matrix stack
drawArm(O, A);

gl.glTranslatef(A, 0, 0);
gl.glRotatef(beta, 0, 0, 1);
// R_z(alpha)T_x(A)R_z(beta) is on top of the stack
drawArm(A, B);

gl.glTranslatef(B-A, 0, 0);
gl.glRotatef(gama, 0, 0, 1);
// R_z(alpha)T_x(A)R_z(beta)T_x(B)R_z(gama) is on top
drawArm(B, C);

// put the solar system at the end of the robot arm
gl.glTranslatef(C-B, 0, 0);
drawSolar(WIDTH/4, 2.5f*cnt, WIDTH/6, 1.5f*cnt);

gl.glPopMatrix();
}

public static void main(String[] args) {
    J2_12_RobotSolar f = new J2_12_RobotSolar();

    f.setTitle("JOGL J2_12_RobotSolar");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}
```

Another way of looking at the modeling and viewing transformation is that the matrix expression transforms the viewing method instead of the model. Translating a model along the negative z axis is like moving the viewing volume (camera) along the positive z axis. Similarly, rotating a model along an axis by a positive angle is like rotating the viewing volume along the axis by a negative angle. When we analyze a model's transformation by thinking about transforming its viewing, the order of transformation steps are top-down from where we specify the viewing volume to where we specify the drawing command. We should remember that the signs of the transformation are logically negated in this perspective. Example *J2_12_RobotSolar.java*, specifies transformation in *myCamera()* from the top-down point of view. The result is shown in Fig. 2.26.

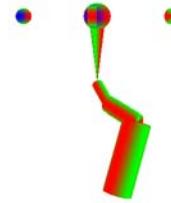


Fig. 2.26 Transform the viewing
[See Color Plate 2]

/* going backwards to the moon in generalized solar system */

```
import net.java.games.jogl.*;

public class J2_13_TravelSolar extends J2_12_RobotSolar {
    public void display(GLDrawable glDrawable) {
        cnt++;
        depth = (cnt/50)%6;

        gl.glClear(GL.GL_COLOR_BUFFER_BIT|GL.GL_DEPTH_BUFFER_BIT);

        if (cnt%60==0) {
            aalpha = -aalpha; abeta = -abeta; agama = -agama;
        }
        alpha += aalpha; beta += abeta; gama += agama;

        gl.glPushMatrix();
        if (cnt%1000<500) {
            // look at the solar system from the moon
            myCamera(A, B, C, alpha, beta, gama);
        }
    }
}
```

```
drawRobot(O, A, B, C, alpha, beta, gama);
gl.glPopMatrix();

void myCamera(float A, float B, float C,
             float alpha, float beta, float gama) {

    float E = WIDTH/4; float e = 2.5f*cnt;
    float M = WIDTH/6; float m = 1.5f*cnt;

    //1. camera faces the negative x axis
    gl.glRotatef(-90, 0, 1, 0);

    //2. camera on positive x axis
    gl.glTranslatef(-M*2, 0, 0);

    //3. camera rotates with the cylinder
    gl.glRotatef(-cylinderm, 0, 1, 0);

    // and so on reversing the solar transformation
    gl.glTranslatef(0, -E, 0);
    gl.glRotatef(-alpha, 0, 0, 1); // tilt angle
    // rotating around the "sun"; proceed angle
    gl.glRotatef(-e, 0, 1, 0);

    // and reversing the robot transformation
    gl.glTranslatef(-C+B, 0, 0);
    gl.glRotatef(-gama, 0, 0, 1);
    gl.glTranslatef(-B+A, 0, 0);
    gl.glRotatef(-beta, 0, 0, 1);
    gl.glTranslatef(-A, 0, 0);
    gl.glRotatef(-alpha, 0, 0, 1);
    gl.glRotatef(-cnt, 0, 1, 0);
}

public static void main(String[] args) {
    J2_13_TravelSolar f = new J2_13_TravelSolar();

    f.setTitle("JOGL J2_13_TravelSolar");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}
```

2.4.6 gluPerspective and gluLookAt

The OpenGL Utility (GLU) library, which is considered part of OpenGL, contains several groups of convenience functions that are built on top of OpenGL functions and complement the OpenGL library. The prefix for OpenGL Utility library functions is "glu" rather than "gl." We have only focused on the OpenGL library. For further understanding viewing, here we discuss two GLU library functions: *gluPerspective()* and *gluLookAt()*. More GLU library functions are discussed in Chapter 5.

gluPerspective() sets up a perspective projection matrix as follows:

```
void gluPerspective(  
    double fovy, // the field of view angle in y-direction  
    double aspect, // width/height of the near clipping plane  
    double zNear, // distance from the origin to the near  
    double zFar // distance from the origin to far  
);
```

The parameters of *gluPerspective()* are explained in Fig. 2.27. Compared with *glFrustum()*, *gluPerspective()* is easier to use for some programmers, but it is less powerful. The *fovy* (field of view) angle is symmetric around *z* axis in *y* direction, and its near and far clipping planes are symmetric around *z* axis as well. Therefore, *gluPerspective()* can only specify a symmetric viewing frustum around *z* axis, whereas *glFrustum()* has no such restriction. The following example *J2_14_Perspective.java* shows an implementation of *myPerspective(double fovy, double aspect, double near, double far)*:

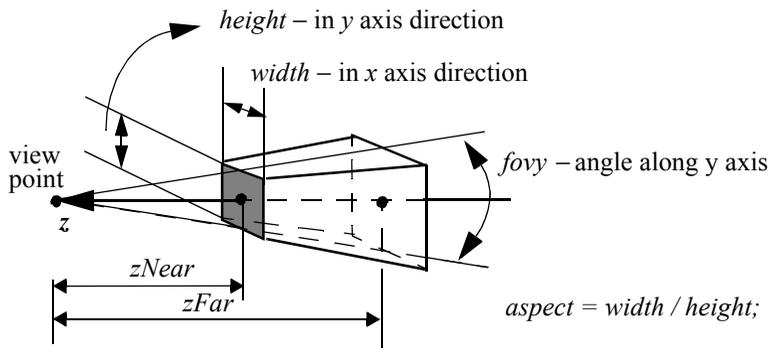


Fig. 2.27 *gluPerspective* specifies a viewing frustum symmetric around z axis

```
/* simulate gluPerspective */
```

```
import net.java.games.jogl.*;
import java.lang.Math;
```

```
public class J2_14_Perspective extends
    J2_13_TravelSolar {

    public void myPerspective(double fovy, double aspect,
                             double near, double far) {
        double left, right, bottom, top;

        fovy = fovy*Math.PI/180; // convert degree to arc

        top = near*Math.tan(fovy/2);
        bottom = -top;
        right = aspect*top;
        left = -right;

        gl.glMatrixMode(GL.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glFrustum(left, right, bottom, top, near, far);
    }

    public void reshape(GLDrawable glDrawable,
                       int x, int y, int width, int height) {
```

```
WIDTH = width;
HEIGHT = height;

// enable zbuffer for hidden-surface removal
gl.glEnable(GL.GL_DEPTH_TEST);
gl.glViewport(0, 0, width, height);

myPerspective(45, 1, width/2, 4*width);

gl.glMatrixMode(GL.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslatef(0, 0, -2*width);
}

public static void main(String[] args) {
    J2_14_Perspective f = new J2_14_Perspective();

    f.setTitle("JOGL J2_14_Perspective");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}
```

glOrtho(), *glFrustum()*, and *gluPerspective* all specify a viewing volume oriented with left and right edges on the near clipping plane parallel to y axis. As we mentioned earlier, we use an *up* vector to represent the orientation of the viewing volume. In other words, by default the projection of *up* onto the near clipping plane is always parallel to the y axis. Because we can transform a viewing volume (camera) now as discussed in the past section, if we specify an orientation vector (upX , upY , upZ), we can orient the viewing volume accordingly. Here the angle between y axis and *up*'s projection on the xy plane is $atan(upX/upY)$, we just need to rotate the viewing volume $-atan(upX/upY)$ to achieve this. This can go further. We do not necessarily have to look from the origin down to the negative z axis. Instead, we can specify the viewpoint as a point *eye* looking down to another point *center*, with *up* as the orientation of the viewing volume. This seems complex, but an equivalent transformation seems much simpler. Given a triangle in 3D (*eye*, *center*, *up*), can we build up a transformation matrix so that after the transformation *eye* will be at the origin, *center* will be in the negative z axis, and *up* in the yz plane? The answer is shown in the method *myLookAt()* in the example *J2_15_LookAt.java* in the next section. *myLookAt()* and *myGluLookAt()* in the example are equivalent simulations of *gluLookAt()*, which

defines a viewing transformation from viewpoint *eye* to another point *center* with *up* as the viewing frustum's orientation vector:

```
void gluLookAt (double eyeX
                , double eyeY
                , double eyeZ
                , double centerX
                , double centerY
                , double centerZ
                , double upX
                , double upY
                , double upZ
                );
```

Here the *eye* and *center* are points, but *up* is a vector. This is slightly different from our triangle example, where *up* is a point as well. As we can see, the *up* vector cannot be parallel to the line (*eye*, *center*).

2.4.7 Multiple Viewports

glViewport(int x, int y, int width, int height) specifies the rendering area within the frame of the display window. By default *glViewport(0, 0, w, h)* is implicitly called in the *reshape(GLDrawable glDrawable, int x, int y, int w, int h)* with the same area as the display window. The viewing volume will be projected into the viewport area accordingly.

We may specify a different viewport using *glViewport()* with lower-left corner (*x, y*) goes from (*0, 0*) to (*w, h*) and the viewport region is an area of *width* to *height* in pixels confined in the display window. All drawing functions afterwards will draw into the current viewport region. That is, the projection goes to the viewport. Also, we may specify multiple viewports at different regions in a drawing area and draw different scenes into these viewports. For example, *glViewport(0, 0, width/2, height/2)* will be the lower-left quarter of the drawing area, and *glViewport(width/2, height/2, width/2, height/2)* will be the upper-right quarter of the drawing area. In our example *J2_15_LookAt.java* below, we also specified different projection methods to demonstrate *myLookAt()*, *mygluLookAt()*, and *myPerspective()* functions. If we don't specify different projection methods in different viewports, the same projection matrix will be used for different viewports. Fig. 2.28 is a snapshot of the multiple viewports rendering.

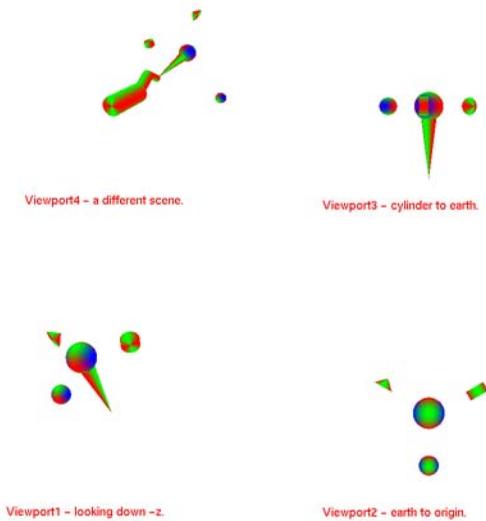


Fig. 2.28 Multiple viewports with different LookAt projections [See Color Plate 3]

/* simulate gluLookAt and display in multiple viewports */

```
import net.java.games.jogl.*;
import java.lang.Math;
import net.java.games.jogl.util.GLUT;

public class J2_15_LookAt extends J2_14_Perspective {
    GLUT glut = new GLUT();

    public void display(GLDrawable glDrawable) {
        cnt++;
        depth = 4;
        gl.glClear(GL.GL_COLOR_BUFFER_BIT |
            GL.GL_DEPTH_BUFFER_BIT);

        viewport1();
        drawSolar(WIDTH/4, cnt, WIDTH/12, cnt);
        // the objects' centers are retrieved from above call
```

```

viewPort2();
drawSolar(WIDTH/4, cnt, WIDTH/12, cnt);
viewPort3();
drawSolar(WIDTH/4, cnt, WIDTH/12, cnt);
viewPort4();
drawRobot(O, A, B, C, alpha, beta, gama);

try {
    Thread.sleep(10);
} catch (Exception ignore) {}
}

public void viewPort1() {
    int w = WIDTH, h = HEIGHT;

    gl.glViewport(0, 0, w/2, h/2);

    // use a different projection
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrtho(-w/2, w/2, -h/2, h/2, -w, w);
    gl.glRasterPos3f(-w/3, -h/3, 0); // start position
    glut.glutBitmapString(gl, GLUT_BITMAP_HELVETICA_18,
        "Viewport1 - looking down -z.");

    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glLoadIdentity();
}

public void viewPort2() {
    int w = WIDTH, h = HEIGHT;
    gl.glViewport(w/2, 0, w/2, h/2);

    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();

    // make sure the cone is within the viewing volume
    gl.glFrustum(-w/8, w/8, -h/8, h/8, w/2, 4*w);
    gl.glTranslatef(0, 0, -2*w);
    gl.glRasterPos3f(-w/3, -h/3, 0); // start position
    glut.glutBitmapString(gl, GLUT_BITMAP_HELVETICA_18,
        "Viewport2 - earth to origin.");

    // earthC retrieved in drawSolar() before viewPort2
    myLookAt(earthC[0], earthC[1], earthC[2],
        0, 0, 0, 0, 1, 0);
}

```

```
    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glLoadIdentity();
}

public void viewPort3() {
    int w = WIDTH, h = HEIGHT;

    gl.glViewport(w/2, h/2, w/2, h/2);

    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    // make sure the cone is within the viewing volume
    gl.glFrustum(-w/8, w/8, -h/8, h/8, w/2, 4*w);
    gl.glTranslatef(0, 0, -2*w);

    gl.glRasterPos3f(-w/3, -h/3, 0); // start position
    glut.glutBitmapString(gl, GLUT_BITMAP_HELVETICA_18,
        "Viewport3 - cylinder to earth.");

    // earthC retrieved in drawSolar() before viewPort3
    mygluLookAt(cylinderC[0], cylinderC[1], cylinderC[2],
        earthC[0], earthC[1], earthC[2],
        earthC[0], earthC[1], earthC[2]);

    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glLoadIdentity();
}

public void viewPort4() {
    int w = WIDTH, h = HEIGHT;

    gl.glViewport(0, h/2, w/2, h/2);

    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    // implemented in superclass J2_14_Perspective
    myPerspective(45, w/h, w/2, 4*w);
    gl.glTranslatef(0, 0, -1.5f*w);

    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glLoadIdentity();

    gl.glRasterPos3f(-w/2.5f, -h/2.1f, 0);
    glut.glutBitmapString(gl, GLUT_BITMAP_HELVETICA_18,
        "Viewport4 - a different scene.");
}
```

```

public void myLookAt(
    double eX, double eY, double eZ,
    double cX, double cY, double cZ,
    double upX, double upY, double upZ) {
    //eye and center are points, but up is a vector

    //1. change center into a vector:
    // glTranslated(-eX, -eY, -eZ);
    cX = cX-eX; cY = cY-eY; cZ = cZ-eZ;

    //2. The angle of center on xz plane and x axis
    // i.e. angle to rot so center in the neg. yz plane
    double a = Math.atan(cZ/cX);
    if (cX>=0) {
        a = a+Math.PI/2;
    } else {
        a = a-Math.PI/2;
    }

    //3. The angle between the center and y axis
    // i.e. angle to rot so center in the negative z axis
    double b = Math.acos(
        cY/Math.sqrt(cX*cX+cY*cY+cZ*cZ));
    b = b-Math.PI/2;

    //4. up rotate around y axis (a) radians
    double upx = upX*Math.cos(a)+upZ*Math.sin(a);
    double upz = -upX*Math.sin(a)+upZ*Math.cos(a);
    upX = upx; upZ = upz;

    //5. up rotate around x axis (b) radians
    double upy = upY*Math.cos(b)-upZ*Math.sin(b);
    upz = upY*Math.sin(b)+upZ*Math.cos(b);
    upY = upy; upZ = upz;

    double c = Math.atan(upX/upY);
    if (upY<0) {
        //6. the angle between up on xy plane and y axis
        c = c+Math.PI;
    }
    gl.glRotated(Math.toDegrees(c), 0, 0, 1);
    // up in yz plane
    gl.glRotated(Math.toDegrees(b), 1, 0, 0);
    // center in negative z axis
    gl.glRotated(Math.toDegrees(a), 0, 1, 0);
    //center in yz plane
    gl.glTranslated(-eX, -eY, -eZ);
    //eye at the origin
}

```

```
public void mygluLookAt(
    double eX, double eY, double eZ,
    double cX, double cY, double cZ,
    double upX, double upY, double upZ) {
    //eye and center are points, but up is a vector

    double[] F = new double[3];
    double[] UP = new double[3];
    double[] s = new double[3];
    double[] u = new double[3];
    F[0] = cX-eX; F[1] = cY-eY; F[2] = cZ-eZ;
    UP[0] = upX; UP[1] = upY; UP[2] = upZ;
    normalize(F); normalize(UP);
    crossProd(F, UP, s); crossProd(s, F, u);

    double[] M = new double[16];
    M[0] = s[0]; M[1] = u[0]; M[2] = -F[0];
    M[3] = 0; M[4] = s[1]; M[5] = u[1];
    M[6] = -F[1]; M[7] = 0; M[8] = s[2];
    M[9] = u[2]; M[10] = -F[2]; M[11] = 0;
    M[12] = 0; M[13] = 0; M[14] = 0; M[15] = 1;

    gl.glMultMatrixd(M);
    gl.glTranslated(-eX, -eY, -eZ);
}

public void normalize(double v[]) {
    double d = Math.sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);

    if (d==0) {
        System.out.println("0 length vector: normalize().");
        return;
    }
    v[0] /= d; v[1] /= d; v[2] /= d;
}

public void crossProd(double U[],
    double V[], double W[]) {
    // W = U X V
    W[0] = U[1]*V[2]-U[2]*V[1];
    W[1] = U[2]*V[0]-U[0]*V[2];
    W[2] = U[0]*V[1]-U[1]*V[0];
}
```

```

public static void main(String[] args) {
    J2_15_LookAt f = new J2_15_LookAt();

    f.setTitle("JOGL J2_15_LookAt");
    f.setSize(WIDTH, HEIGHT);
    f.setVisible(true);
}
}

```

2.5 Review Questions

1. An octahedron has $v1=(1,0,0)$, $v2=(0,1,0)$, $v3=(0,0,1)$, $v4=(-1,0,0)$, $v5=(0,-1,0)$, $v6=(0,0,-1)$. Please choose the triangles that face the outside of the octahedron.

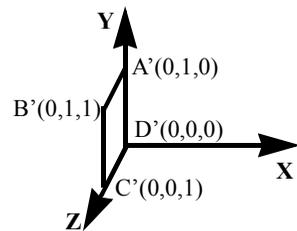
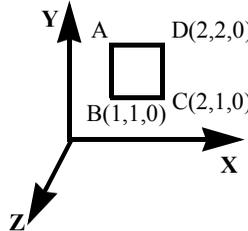
- a. $(v1v2v3, v1v3v5, v1v5v6, v1v2v6)$
- b. $(v2v3v1, v2v1v6, v2v6v4, v2v4v3)$
- c. $(v3v2v1, v3v5v1, v3v4v2, v3v4v5)$
- d. $(v4v2v1, v4v5v1, v3v4v2, v3v4v5)$

2. If we subdivide the above octahedron 8 times (depth=8), how many triangles we will have in the final sphere.

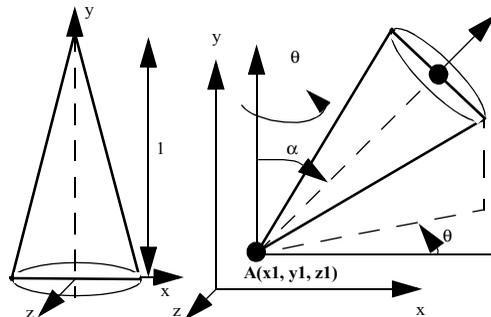
No. of triangles: _____

3. Choose the *matrix expression* that would transform square ABCD into square A'B'C'D' in 3D as shown in the figure below.

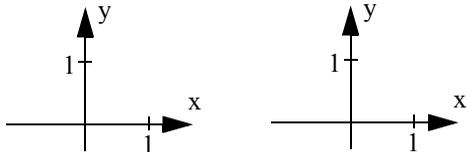
- a. $T(-1,-1, 0)R_y(-90)$
- b. $R_y(-90) T(-1,-1, 0)$
- c. $T(-2,-2, 0)R_z(-90)R_y(90)$
- d. $R_y(90)R_z(-90)T(-2,-2, 0)$



4. *myDrawTop()* will draw a top below on the left. Write a section of OpenGL code so that the top will appear as specified on the right with tip at $A(x1, y1, z1)$, tilted α , and preceeded θ around an axis parallel to y axis.



5. *myDrawTop()* will draw an object in oblique projection as in the question above with height equals 1 and radius equals 0.5. Please draw two displays in **orthographic** projection according to the program on the right (as they will appear on the screen where the z axis is perpendicular to the plane).



```

glLoadIdentity();
glRotatef (-90, 0.0, 1.0, 0.0);
myDrawTop (); // left
glRotatef (-90, 0.0, 0.0, 1.0);

glPushMatrix();
glTranslatef (0.0, 0.0, 1.0);
myDrawTop (); //right
glPopMatrix();
    
```

6. In the scan-line algorithm for filling polygons, if z-buffer is used, when should the program call the z-buffer algorithm function?

- a. at the beginning of the program
- b. at the beginning of each scan-line
- c. at the beginning of each pixel
- d. at the beginning of each polygon

7. Collision detection avoids two models in an animation penetrating each other; which of the following is FALSE:

- a. bounding boxes are used for efficiency purposes in collision detection
- b. both animated and stationary objects are covered by the bounding boxes
- c. animated objects can move whatever distance between frames of calculations
- d. collision detection can be calculated in many different ways

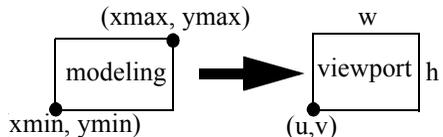
8. After following transformations, what is on top of the matrix stack at *drawObject2()*?

```

glLoadIdentity(); glPushMatrix(); glMultMatrixf(S); glRotatef(a,1,0,0); glTranslatef(t,0,0);
drawObject1 (); glGetFloatv(GL_MODELVIEW_MATRIX, &tmp); glPopMatrix();
glPushMatrix(); glMultMatrixf(S); glMultMatrixf(&tmp); drawObject2(); glPopMatrix();
    
```

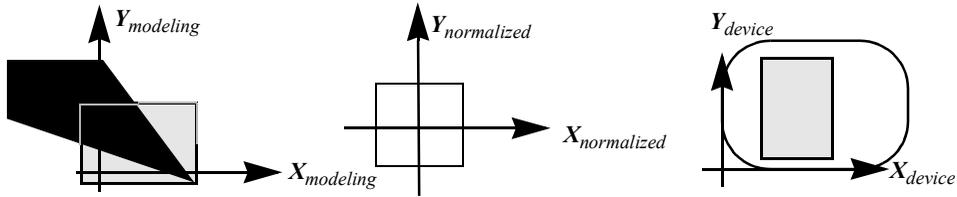
- a. $SSR_x(a)T_x(t)$
- b. $ST_x(t)R_x(a)S$
- c. $T_x(t)R_x(a)SS$
- d. $R_x(a)SST_x(t)$
- e. $SR_x(a)T_x(t)$

9. Given *glViewport (u, v, w, h)* and *gluOrtho2D(xmin, xmax, ymin, ymax)*, choose the 2D transformation **matrix expression** that maps a point in the modeling (modelview) coordinates to the device (viewport) coordinates.



- a. $S(1/(xmax - xmin), 1/(ymax - ymin))T(-xmin, -ymin)T(u, v)S(w, h)$
- b. $S(1/(xmax - xmin), 1/(ymax - ymin))S(w, h)T(-xmin, -ymin)T(u, v)$
- c. $T(u, v)S(w, h)S(1/(xmax - xmin), 1/(ymax - ymin))T(-xmin, -ymin)$
- d. $T(-xmin, -ymin)T(u, v)S(1/(xmax - xmin), 1/(ymax - ymin))S(w, h)$

10. Given a 2D model and a modeling window, please draw the object in normalized coordinates after clipping and in the device as it appears on a display.

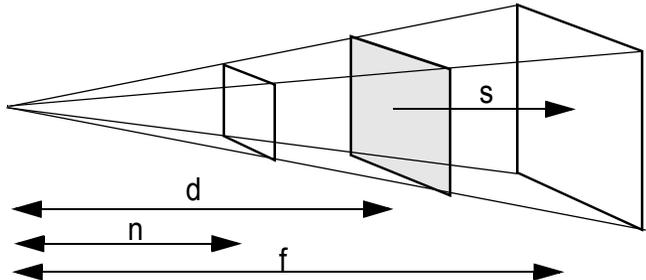


11. In the OpenGL graphics pipeline, please order the following according to their order of operations:

- | | |
|---|--|
| <input type="checkbox"/> clipping | <input type="checkbox"/> viewport transformation |
| <input type="checkbox"/> modelview transformation | <input type="checkbox"/> normalization |
| <input type="checkbox"/> perspective division | <input type="checkbox"/> scan conversion |

12. Please implement the following viewing command: `gmuPerspective(fx, fy, d, s)`, where the viewing direction is from the origin looking down the negative z axis. fx is the field of view angle in the x direction; fy is the field of view angle in the y direction; d is the distance from the viewpoint to the center of the viewing volume, which is a point on the negative z axis; s is the distance from d to the near or far clipping planes.

```
gmuPerspective(fx, fy, d, s) {
```



```
    glFrustum(l, r, b, t, n, f);
}
```

2.6 Programming Assignments

1. Implement `myLoadIdentity`, `myRotatef`, `myTranslatef`, `myScalef`, `myPushMatrix`, and `myPopMatrix` just like their corresponding OpenGL commands. Then, in the rest of the programming assignments, you can interchange them with OpenGL commands.

2. Check out online what is polarview transformation; implement your own polarview with a demonstration of the function.

3. As shown in the figure on the right, use 2D transformation to rotate the stroke font and the star.

4. The above problem can be extended into 3D: the outer circle rotates along y axis, the inner circle rotates around x axis, and the star rotates around z axis.

5. Draw a cone, a cylinder, and a sphere that bounce back and forth along a circle, as shown in the figure. When the objects meet, they change their directions of movement. The program must be in double-buffer mode and have hidden surface removal.

6. Draw two circles with the same animation as above. At the same time, one circle rotates around x axis, and the other rotates around y axis.

7. Implement a 3D robot arm animation as in the book, and put the above animation system on the palm of the robot arm. The system on the palm can change its size periodically, which is achieved through scaling.

8. Draw a cone, a cylinder, and a sphere that move and collide in the moon's trajectory in the generalized solar system. When the objects meet, they change their directions of movement.

9. Put the above system on the palm of the robot arm.

10. Implement `myPerspective` and `myLookAt` just like `gluPerspective` and `gluLookAt`. Then, use them to look from the cone to the earth or cylinder in the system above.

11. Display different perspectives or direction of viewing in multiple viewports.

