
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:

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

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.

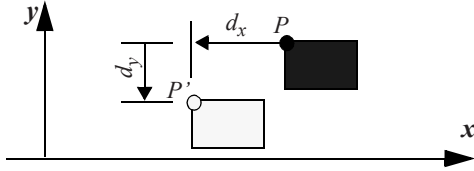


Fig. 2.1 Basic transformation: translation

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 $(0, 0)$. If the rotation is clockwise, the rotation angle θ is then negative. The rotation axis is perpendicular to the 2D plane at the origin:

$$x' = x \cos \theta - y \sin \theta, \quad (\text{EQ 16})$$

$$y' = x \sin \theta + y \cos \theta. \quad (\text{EQ 17})$$

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

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (\text{EQ 18})$$

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:

$$P' = R(\theta)P. \quad (\text{EQ 19})$$

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.

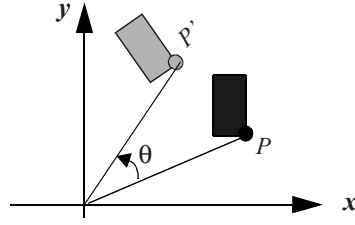


Fig. 2.2 Basic transformation: rotation

2.2.3 2D Scaling

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

$$x' = s_x x, \quad (\text{EQ 20})$$

$$y' = s_y y. \quad (\text{EQ 21})$$

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

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (\text{EQ 22})$$

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:

$$P' = S(s_x, s_y)P. \quad (\text{EQ 23})$$

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.



Fig. 2.3 Basic transformation: scaling

2.2.4 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 2.1: finding 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 $h(x_l, y_l)$. If we know the rotation angle is θ , can we find the new end point h' after the rotation? As shown in Fig. 2.4, we can achieve this by a series of transformations.

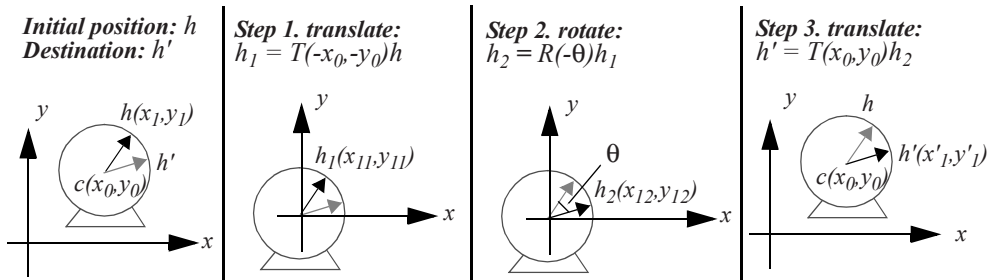


Fig. 2.4 Moving the clock hand by matrix multiplications

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 :

$$\begin{bmatrix} x_{II} \\ y_{II} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ 1 \end{bmatrix}. \quad (\text{EQ 24})$$

$$\text{That is, } h_I = T(-x_0, -y_0)h. \quad (\text{EQ 25})$$

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

$$h_2 = R(-\theta)h_I. \quad (\text{EQ 26})$$

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

$$h' = T(x_0, y_0)h_2. \quad (\text{EQ 27})$$

Therefore, putting Equations 19 to 21 together, the combination of transformations to achieve the clock hand movement is:

$$h' = T(x_0, y_0)R(-\theta)T(-x_0, -y_0)h. \quad (\text{EQ 28})$$

$$\text{That is: } \begin{bmatrix} x'_I \\ y'_I \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ 1 \end{bmatrix}. \quad (\text{EQ 29})$$

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:

$$h' = Mh. \quad (\text{EQ 30})$$

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 30 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: $T(x_0, y_0)[R(-\theta)[T(-x_0, -y_0)h]]$.

Example 2.2: 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 Viewing.

Here, we discuss a similar problem: a transformation that allows reshaping a rectangular area directly. Let's assume the coordinate system of the screen is as in Fig. 2.5. After reshaping, the rectangular area (and all the vertices of the models) 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. The corresponding matrix expression is:

$$T(P_2)S(s_x, s_y)T(-P_1). \quad (\text{EQ 31})$$

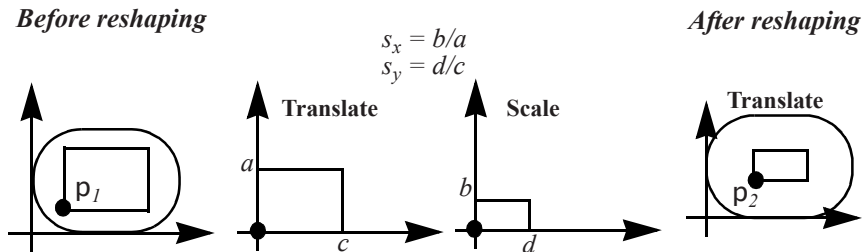


Fig. 2.5 Scaling an arbitrary rectangular area

Example 2.3: 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. 2.6). 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 to achieve the same goal.

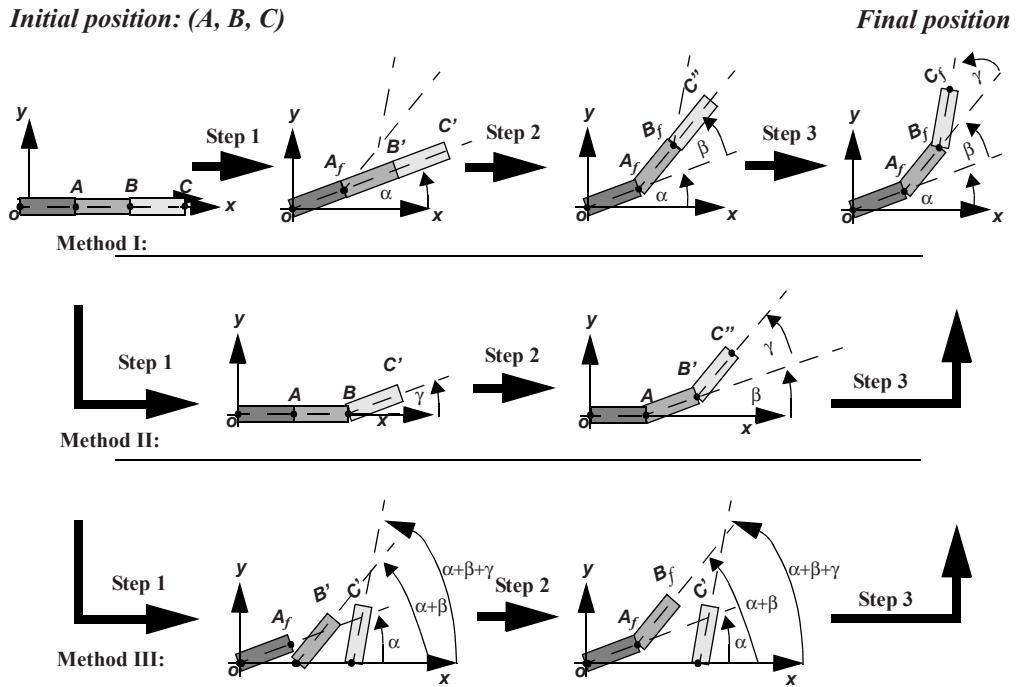


Fig. 2.6 A 2D robot arm rotates (α, β, γ) degrees at the 3 joints, respectively

Method I.

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

$$A_f = R(\alpha)A; B' = R(\alpha)B; C' = R(\alpha)C. \quad (\text{EQ 32})$$

2. Consider $A_f B' C'$ to be a clock hand like the example in Fig. 2.4. 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:

$$B_f = T(A_f)R(\beta)T(-A_f)B'; C'' = T(A_f)R(\beta)T(-A_f)C'. \quad (\text{EQ 33})$$

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

$$C_f = T(B_f)R(\gamma)T(-B_f)C''. \quad (\text{EQ 34})$$

Method II.

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

$$C' = T(B)R(\gamma)T(-B)C. \quad (\text{EQ 35})$$

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

$$B' = T(A)R(\beta)T(-A)B; C'' = T(A)R(\beta)T(-A)C'. \quad (\text{EQ 36})$$

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

$$A_f = R(\alpha)A; \quad (\text{EQ 37})$$

$$B_f = R(\alpha)B' = R(\alpha)T(A)R(\beta)T(-A)B; \quad (\text{EQ 38})$$

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

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:

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

2. Translate AB' to A_fB_f :

$$B_f = T(A_f)T(-A)B' = T(A_f)R(\alpha+\beta)T(-A)B. \quad (\text{EQ 41})$$

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 :

$$C_f = T(B_f)T(-B)C' = T(B_f)R(\alpha+\beta+\gamma)T(-B)C. \quad (\text{EQ 42})$$

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. 2.7). The z axis is perpendicular to the display with the arrow pointing towards 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 $z=0$.

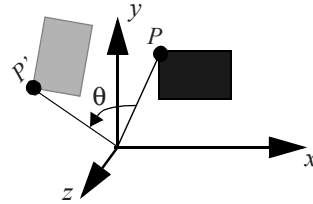


Fig. 2.7 A 3D rotation around z axis

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:

$$\text{Translation: } T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (\text{EQ 43})$$

$$\text{Scaling: } S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (\text{EQ 44})$$

$$\text{Rotation around } x \text{ axis: } R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (\text{EQ 45})$$

$$\text{Rotation around } y \text{ axis: } R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (\text{EQ 46})$$

$$\text{Rotation around } z \text{ axis: } R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (\text{EQ 47})$$

For example, the 2D transformation Equation 37 can be replaced by the corresponding 3D matrices:

$$A_f = R_z(\alpha)A, \quad (\text{EQ 48})$$

where $A = \begin{bmatrix} A_x \\ A_y \\ A_z \\ 1 \end{bmatrix}$, $A_f = \begin{bmatrix} A_{fx} \\ A_{fy} \\ A_{fz} \\ 1 \end{bmatrix}$, and $A_z=0$. We can show that $A_{fz}=0$ as well.

2.3.2 Transformation in OpenGL

As an example, we will implement in OpenGL the robot arm transformation *Method II* in Fig. 2.6. 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 (Example 2.4, which implements *Method II* in Fig. 2.6.)

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

```
glMatrixMode (GL_MODELVIEW);
```

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

```
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 matrix is on the current matrix stack already. The result of the multiplication replaces the matrix currently on the top of the stack. If the identity matrix is on the stack, then $IR_z(\alpha) = R_z(\alpha)$:

```
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 37):

```
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 38:

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

  glTranslatef (A[0], A[1], 0.0);
  glRotatef (beta, 0.0, 0.0, 1.0);
  glTranslatef (-A[0], -A[1], 0.0);
  drawArm (A, B);
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 on the top of the stack is the same as the matrix immediately below it. *glPopMatrix()* will move the stack pointer down one slot. The obvious advantage of this mechanism is to separate the transformations of the current model between *glPushMatrix()* and *glPopMatrix()* from the transformations of models later.

Let's look at the function *drawRobot()* in Example 2.4 below. Fig. 2.8 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 39.

Status of the OpenGL MODELVIEW matrix stack

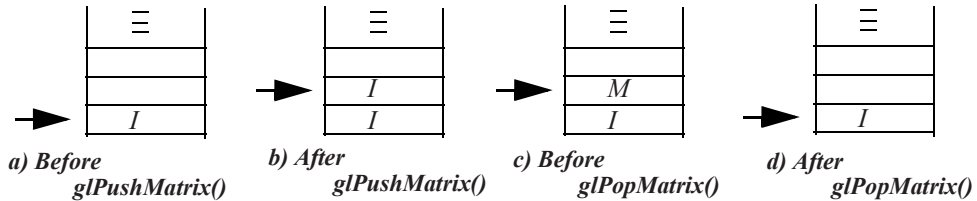


Fig. 2.8 Matrix stack manipulations with *glPushMatrix()* and *glPopMatrix()*

6. 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. 2.9).

Status of the OpenGL MODELVIEW matrix stack

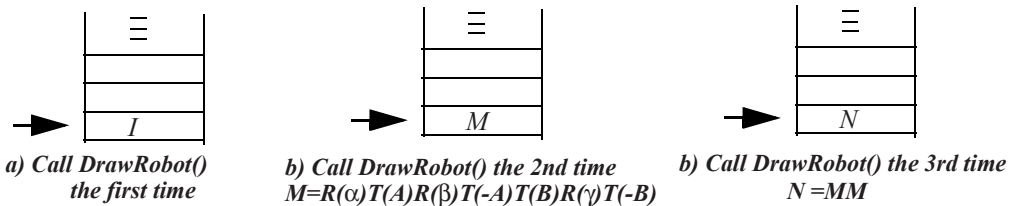


Fig. 2.9 Matrix stack manipulations without using *glPushMatrix()* and *glPopMatrix()*

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.

Methods I and III (Fig. 2.6) cannot be achieved using OpenGL transformations directly, since 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 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.

/* Example 2.4.robot2d.c: 2D three segments arm transformation */

```
float O[3] = {0.0, 0.0, 0.0}, A[3] = {0.0, 0.0, 0.0},
      B[3] = {0.0, 0.0, 0.0}, C[3] = {0.0, 0.0, 0.0};

float alpha, beta, gama, aalpha=.1, abeta=.3, agama=0.7;

void drawArm(float *End1, float *End2)
{
    glBegin(GL_LINES);
        glVertex3fv(End1);
        glVertex3fv(End2);
    glEnd();
}

void drawRobot(float *A, float *B, float *C,
              float alpha, float beta, float gama)
{
    glPushMatrix();

    glColor3f(1, 0, 0);
    glRotatef(alpha, 0.0, 0.0, 1.0);

    // R_z(alpha) is on top of the matrix stack
    drawArm(O, A);

    glColor3f(0, 1, 0);
    glTranslatef(A[0], A[1], 0.0);
    glRotatef(beta, 0.0, 0.0, 1.0);
    glTranslatef(-A[0], -A[1], 0.0);

    // R_z(alpha)T(A)R_z(beta)T(-A) is on top of the stack
    drawArm(A, B);
}
```

```
        glColor3f(0, 0, 1);
        glTranslatef (B[0], B[1], 0.0);
        glRotatef (gama, 0.0, 0.0, 1.0);
        glTranslatef (-B[0], -B[1], 0.0);

        // R_z(alpha)T(A)R_z(beta)T(-A)T(B)R_z(gama)T(-B)
        drawArm (B, C);

    glPopMatrix();
}

void display(void)
{
    if (rand() % 10000 == 0) aalpha = -aalpha;

    // arm rotation angles
    alpha+= aalpha; beta+= abeta; gama+= agama;

    glClear(GL_COLOR_BUFFER_BIT);
    drawRobot(A, B, C, alpha, beta, gama);

    glutSwapBuffers();
}

void Reshape(int w, int h)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);

    //initialize robot arm end positions
    A[0] = (float) w/7;
    B[0] = (float) w/5;
    C[0] = (float) w/4;

    Width = w; Height = h;
    glViewport (0, 0, Width, Height);

    // hardware set to use PROJECTION matrix stack
    glMatrixMode (GL_PROJECTION);
    // initialize the current top of matrix stack to identity
    glLoadIdentity ();
    glOrtho(-Width/2, Width/2, -Height/2, Height/2, -1.0, 1.0);

    // hardware set to use model transformation matrix stack
    glMatrixMode (GL_MODELVIEW);
    // initialize the current top of matrix stack to identity
    glLoadIdentity ();
}
```


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*.

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
glEnable(GL_DEPTH_TEST);

// clear both framebuffer and zbuffer
glClear(GL_COLOR_BUFFER_BIT | 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 view point 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 view point, 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 Collision Detection

In addition to visible-model determination, bounding volumes are also used for *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.

2.3.5 3D Models: Cone, Cylinder, and Sphere

Approximating a cone. In Example 1.5, we approximated a circle with subdividing triangles. If we raise the center of the circle along the z axis, we approximate a cone, as shown in Fig. 2.10. We need to make sure that our model is contained within the defined coordinates (i.e., the viewing volume):

```
glOrtho(-Width/2, Width/2, -Height/2,
        Height/2, -Width/2, Width/2);
```

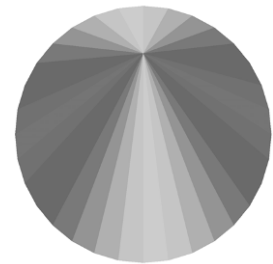


Fig. 2.10 A cone

/* Example 2.5.cone: draw a cone by subdivision */

```
int depth=5, circleRadius=200, cnt=1;
```

```
static float vdata[4][3] = {
    {1.0, 0.0, 0.0}, {0.0, 1.0, 0.0},
    {-1.0, 0.0, 0.0}, {0.0, -1.0, 0.0}
};

void subdivideCone(float *v1, float *v2, int depth)
{
    float v0[3] = {0, 0, 0}, v12[3];
    int i;

    if (depth == 0) {
        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 (i=0; i<3; i++) v12[i] = v1[i]+v2[i];
    normalize(v12);
    subdivideCone(v1, v12, depth - 1);
    subdivideCone(v12, v2, depth - 1);
}

void drawCone(void)
// draw a unit cone: center at origin and bottom in xy plane
{
    subdivideCone(vdata[0], vdata[1], depth);
    subdivideCone(vdata[1], vdata[2], depth);
    subdivideCone(vdata[2], vdata[3], depth);
    subdivideCone(vdata[3], vdata[0], depth);
}

void display(void)
{
    // clear both framebuffer and zbuffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glRotatef(1.1, 1., 0., 0.); // rotate 1.1 deg. alone x axis
    glPushMatrix();
        glScaled(circleRadius, circleRadius, circleRadius);
        drawCone();
    glPopMatrix();
    glutSwapBuffers();
}
```

```
static void Reshape(int w, int h)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);

    // enable zbuffer (depthbuffer) for hidden-surface removal
    glEnable(GL_DEPTH_TEST);

    Width = w; Height = h;
    glViewport (0, 0, Width, Height);

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    // make sure the cone is within the viewing volume
    glOrtho(-Width/2, Width/2 -Height/2,
            Height/2, -Width/2, Width/2);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}
```

Approximating a cylinder. If we can draw a circle at $z=0$, then we can draw another circle at $z=1$. If we connect the rectangles of the same vertices on the edges of the two circles, we have a cylinder, as shown in Fig. 2.11.

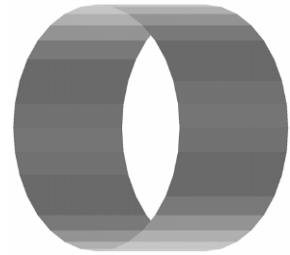


Fig. 2.11 A cylinder

/* Example 2.6.cylinder.c: draw a cylinder by subdivision */

```
void subdivideCylinder(float *v1, float *v2, int depth)
{
    float v11[3], v22[3], v00[3] = {0, 0, 0}, v12[3];
    float v01[3], v02[3];
    int i;

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

        for (i=0; i<3; i++) {
            v01[i] = v11[i] = v1[i];
            v02[i] = v22[i] = v2[i];
        }

        // the height of the cone along z axis
        v01[2] = v02[2] = 1;
    }
```

```

    // draw the side rectangles of the cylinder
    glBegin(GL_POLYGON);
        glVertex3fv(v11);
        glVertex3fv(v22);
        glVertex3fv(v02);
        glVertex3fv(v01);
    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);
}

```

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. 2.12(a). Example 2.7 uses this method to subdivide an octahedron (Fig. 2.12(b)) into a sphere.

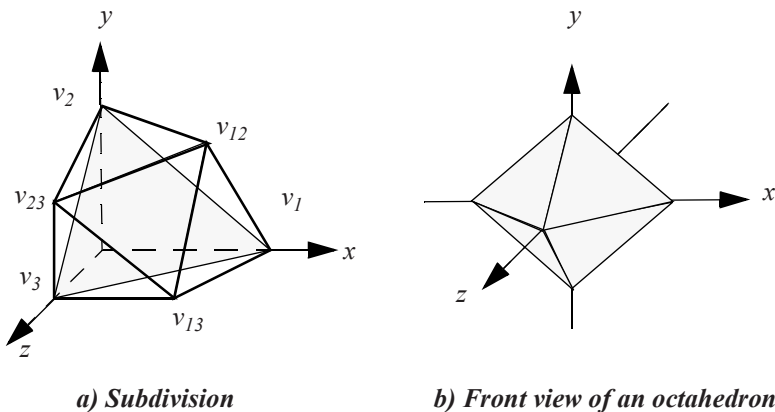


Fig. 2.12 Drawing a sphere through subdivision

/* Example 2.7.sphere.c: draw a sphere by subdivision */

```

static float vdata[6][3] = {
    {1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0},
    {-1.0, 0.0, 0.0}, {0.0, -1.0, 0.0}, {0.0, 0.0, -1.0}
};

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

    if (depth == 0) {
        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);
}

void drawSphere(void)
{
    // draw eight triangles to cover the octahedron
    subdivideSphere(vdata[0], vdata[1], vdata[2], depth);
    subdivideSphere(vdata[0], vdata[2], vdata[4], depth);
    subdivideSphere(vdata[0], vdata[4], vdata[5], depth);
    subdivideSphere(vdata[0], vdata[5], vdata[1], depth);

    subdivideSphere(vdata[3], vdata[1], vdata[5], depth);
    subdivideSphere(vdata[3], vdata[5], vdata[4], depth);
    subdivideSphere(vdata[3], vdata[4], vdata[2], depth);
    subdivideSphere(vdata[3], vdata[2], vdata[1], depth);
}

```

2.3.6 Composition of 3D Transformations

Example 2.8 implements the robot arm in Example 2.4 with 3D cylinders, as shown in Fig. 2.13. We also add one rotation around the y axis, so the robot arm moves in 3D.



Fig. 2.13 A 3-segment robot arm

/* Example 2.8.robot3d.c: 3D 3-segment arm transformation */

```
drawArm(float End1, float End2) {
    float scale;
    scale = End2-End1;

    glPushMatrix();

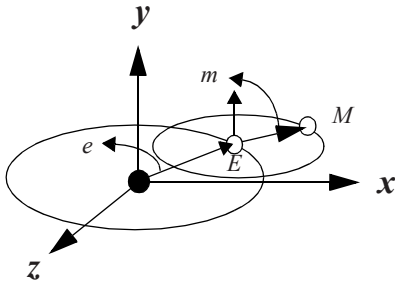
        // the cylinder lies in the z axis;
        // rotate it to lie in the x axis
        glRotatef(90.0, 0.0, 1.0, 0.0);
        glScalef(scale/5.0, scale/5.0, scale);

        drawCylinder();
    glPopMatrix();
}

static void drawRobot(float alpha, float beta, float gama)
{
    ...
    // the robot arm is rotating around the y axis
    glRotatef (1.0, 0.0, 1.0, 0.0);
    ...
}
```

Example 2.9 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. 2.14).

This problem is exactly like the clock problem in Fig. 2.4, 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.



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. 2.14 Simplified solar system: a 2D problem in 3D

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

/* Example 2.9.solar.c: draw a simplified solar system */

```
void drawSolar(float E, float e, float M, float m)
{
    glPushMatrix();
        glRotatef(e, 0.0, 1.0, 0.0); // rotate around the "sun"
        glTranslatef(E, 0.0, 0.0);
        drawSphere(); // Ry(e)Tx(E)

        glRotatef(m, 0.0, 1.0, 0.0); // rotate around the "earth"
        glTranslatef(M, 0.0, 0.0);
        drawSphere(); // Ry(e)Tx(E)R(m)Tx(M)
    glPopMatrix();
}
```


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 towards the center of the earth, and the moon rotates around this axis as in Fig. 2.15. 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 ?

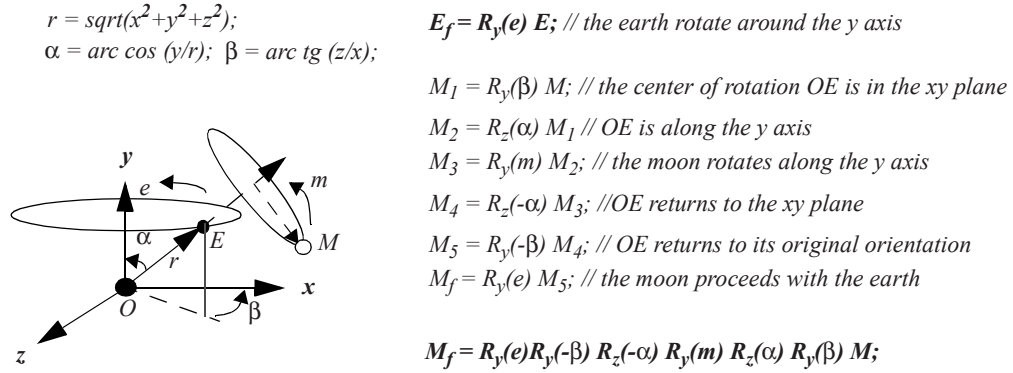


Fig. 2.15 Generalized solar system: a 3D problem

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 , but we use it as a reference to explain the rotation.

1. As shown in Fig. 2.15, 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_1 is in the xy plane:

$$M_1 = R_y(\beta)M; \quad E_1 = R_y(\beta)E. \quad (\text{EQ 49})$$

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:

$$M_2 = R_z(\alpha)M_1; E_2 = R_z(\alpha)E_1. \quad (\text{EQ 50})$$

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

$$M_3 = R_y(m)M_2. \quad (\text{EQ 51})$$

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

$$M_4 = R_z(-\alpha)M_3; E_1 = R_z(-\alpha)E_2. \quad (\text{EQ 52})$$

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

$$M_5 = R_y(-\beta)M_4; E = R_y(-\beta)E_1. \quad (\text{EQ 53})$$

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

$$M_f = R_y(e)M_5; E_f = R_y(e)E. \quad (\text{EQ 54})$$

8. Putting the transformation matrices together, we have:

$$M_f = R_y(e)R_y(-\beta) R_z(-\alpha) R_y(m) R_z(\alpha) R_y(\beta) M \quad (\text{EQ 55})$$

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 as shown in Fig. 2.16. 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 $R_y(-\beta) R_z(-\alpha) R_y(m) R_z(\alpha) R_y(\beta)$.

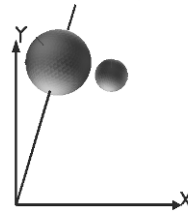


Fig. 2.16 Generalized solar system

/* Example 2.10.gensolar.c: draw a generalized solar system */

```

void drawSolar(float E, float e, float M, float m)
{
    float alpha=30;

    glPushMatrix();
        glRotatef(e, 0.0, 1.0, 0.0); // rotate around the "sun"
        glRotatef(alpha, 0.0, 0.0, 1.0); // tilt angle
        glTranslatef(0., E, 0.0);
        drawSphere(); // the earth

        glRotatef(m, 0.0, 1.0, 0.); // rotate around the "earth"
        glTranslatef(M, 0., 0.);
        drawSphere(); // the moon
    glPopMatrix();
}

```

The generalized solar system corresponds to a top that rotates and proceeds as shown in Fig. 2.17(b). 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 (Example 2.11), we have a top that rotates and proceeds, and a sphere that rotates around the top (Fig. 2.17(c)).

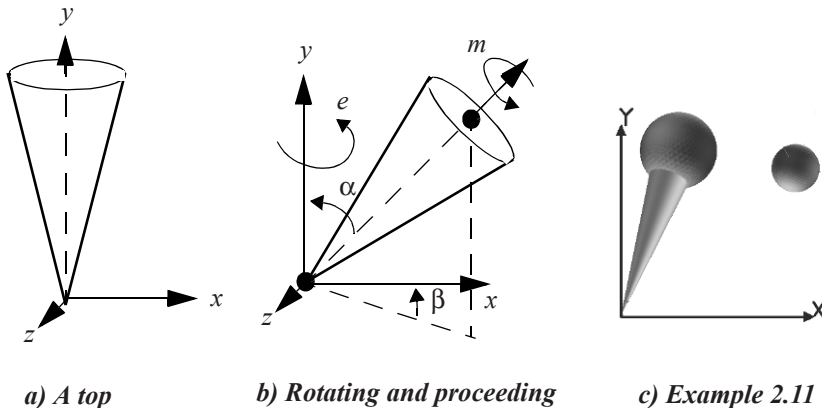


Fig. 2.17 Generalized solar system: a top rotates and proceeds

/* Example 2.11.conesolar.c: draw a cone solar system */

```
void drawSolar(float E, float e, float M, float m)
{
    float alpha=30;

    glPushMatrix();
        // rotating around the "sun"; proceed angle
        glRotatef(e, 0.0, 1.0, 0.0);
        glRotatef(alpha, 0.0, 0.0, 1.0); // tilt angle
        glTranslatef(0., E, 0.0);
        glPushMatrix();
            glScalef(E/8,E,E/8);
            glRotatef(90, 1.0, 0.0, 0.0); // orient the cone
            drawCone();
        glPopMatrix();

        glRotatef(m, 0.0, 1.0, 0.); // rotate around the "earth"
        glTranslatef(M, 0., 0.);
        glScalef(E/8,E/8,E/8);
        drawSphere();
    glPopMatrix();
}
```

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. 2.18). 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. 2.18. Through normalization, the clipping algorithm avoid dealing with the changing sizes of the modeling window and the device view port.

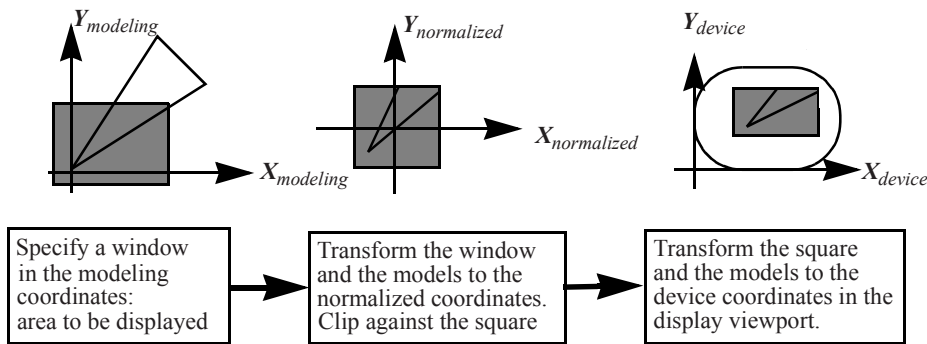


Fig. 2.18 2D viewing pipeline

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 *view point*. You may consider that your eye is at the view point 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. 2.19, 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.

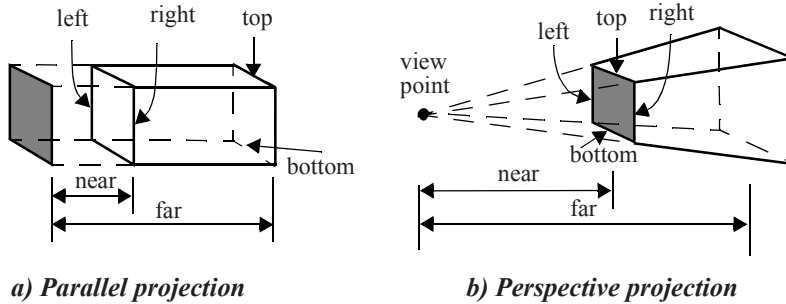


Fig. 2.19 View volumes and projection methods

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. 2.20). 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.

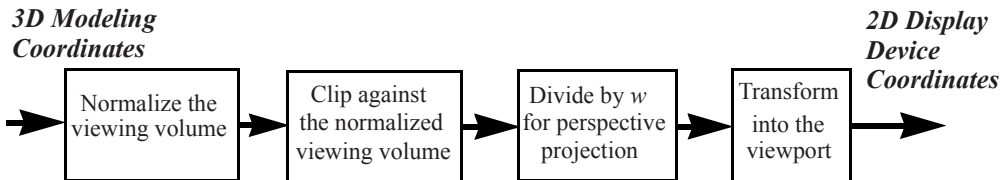


Fig. 2.20 3D viewing pipeline

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. 2.19(a). `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. 2.19(b). `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 view point, which is fixed at the origin looking down the negative z axis.

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
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
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. 2.21 ($x=1$, $x=-1$, $y=1$, $y=-1$, $z=1$, and $z=-1$). Therefore, instead of calculating the clipping and projection directly, the normalization transformation is carried out first to simplify the clipping and the projection.

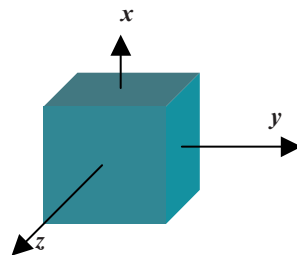


Fig. 2.21 Normalized viewing volume — a cube with $(-1$ to $1)$ along each axis

Similarly, *glFrustum()* also specifies a matrix that transforms the perspective viewing volume into a normalized viewing volume as in Fig. 2.21. 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. Since *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 to accomodate certain curves. 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.

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. 2.19, 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 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:

$$x = x_0 + t(x_1 - x_0) \quad (\text{EQ 56})$$

$$y = y_0 + t(y_1 - y_0) \quad (\text{EQ 57})$$

$$z = z_0 + t(z_1 - z_0) \quad (\text{EQ 58})$$

Now if any bit is not equal to zero, say Bit 2 = 1, then $z = \text{near}$, and we can find t in Equation 58. and therefore find the intersection point (x, y, z) according to Equation 56 and Equation 57.

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:

$$ax + by + cz + d = 0. \quad (\text{EQ 59})$$

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 56, Equation 57, and Equation 58 into Equation 59. 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.

2.4.5 An Example of Viewing in OpenGL

Viewing transformation is 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()* and to make sure that your models are in the viewing volume after being transformed by the MODELVIEW matrix. The following descriptions explain Example 2.12.

1. *glutInitWindowSize()* in *main()* specifies the display window on the screen in pixels.
2. *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.
3. *glOrtho()* or *glFrustum()* specify the viewing volume. The models in the viewing volume will appear in the viewport area on the display.
4. The first matrix we multiply on the MODELVIEW matrix stack, after loading the identity matrix, is a translation along the z axis. This translation can be viewed as the last transformation in modeling coordinates. That is, after finishing all modeling and transformation, we move the origin of the modeling coordinates (and all the models after being transformed in the modeling coordinates) along z axis into the center of the viewing volume.
5. 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 to where we specify the viewing volume.
6. In *display()*, you may think that a robot arm is calculated at the origin of the modeling coordinates. Actually, the robot arm is translated along z axis $-(zNear+zFar)/2$ in order to put the arm in the middle of the viewing volume.
7. Another way of looking at the MODELVIEW matrix is that the matrix transforms the viewing method instead of the model. Translating a model along the negative z axis is like moving the viewing volume 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.
8. When we analyze a model's transformation by thinking about transforming its viewing, the order of transformation steps are topdown 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 2.12.robotSolar.c: 3D transformation/viewing */

```
void display(void)
{
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // draw a robot arm from the origin
    drawRobot(A, B, C, alpha, beta, gama);

    glutSwapBuffers();
}

static void Reshape(int w, int h)
{
    float zNear=w, zFar=3*w;

    glClearColor (0.0, 0.0, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);

    // viewport lower left corner (0,0), aspect ratio w/h
    glViewport (0, 0, w, h);

    // hardware set to use projection matrix stack
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    //glOrtho(-w/2, w/2, -h/2, h/2, zNear, zFar);
    glFrustum(-w/2, w/2, -h/2, h/2, zNear, zFar);

    // hardware set to use model transformation matrix stack
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
    // the origin is at the center between znear and zfar
    glTranslatef (0., 0., -(zNear+zFar)/2);
}
```

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.

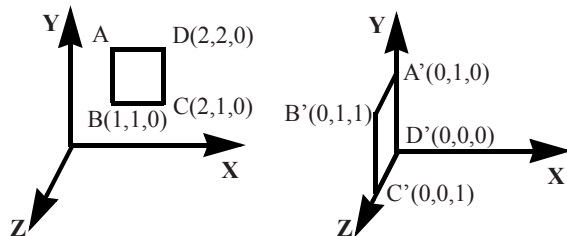
- a. $(v1v2v3, v1v3v5, v1v5v6, v1v2v6)$ b. $(v2v3v1, v2v1v6, v2v6v4, v2v4v3)$
 c. $(v3v2v1, v3v5v1, v3v4v2, v3v4v5)$ d. $(v4v2v1, v4v5v1, v3v4v2, v3v4v5)$

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

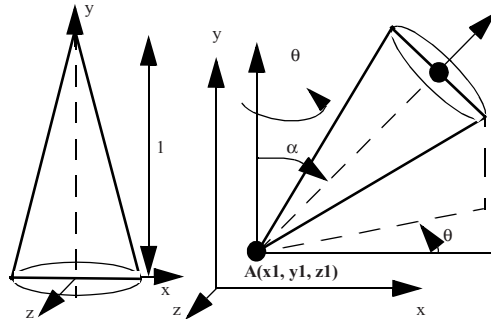
No. of triangles:

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

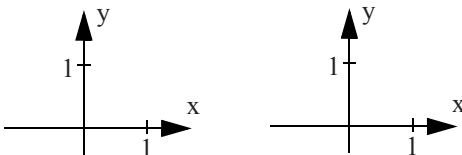
- a. $T(-1, -1, 0)R_y(-90)$
 b. $R_y(-90)T(-1, -1, 0)$
 c. $T(-2, -2, 0)R_z(-90)R_y(90)$
 d. $R_y(90)R_z(-90)T(-2, -2, 0)$



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 proceeded 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

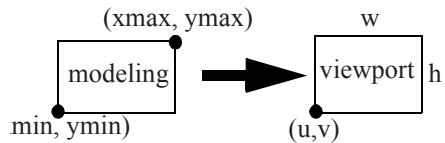
8. After following transformations, what is on top of the matrix stack at `drawObject2()`?

```
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();
```

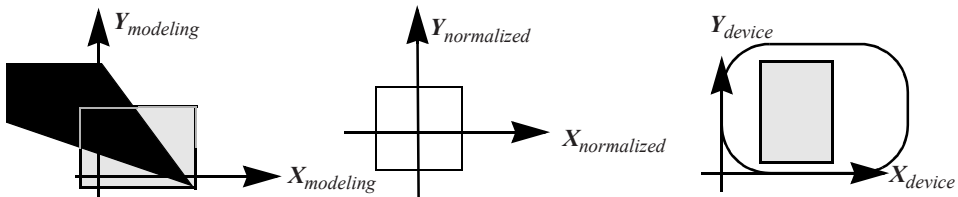
- a. $SSR_x(a)T_x(t)$
- b. $ST_x(t)R_x(a)S$
- c. $T_x(t)R_x(a)SS$
- d. $R_x(a)SST_x(t)$
- e. $SR_x(a)T_x(t)$

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:

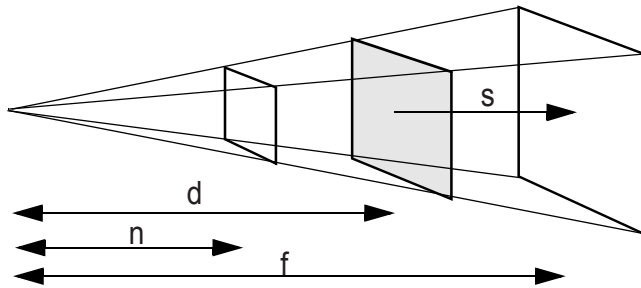
- () clipping
- () viewport transformation
- () modelview transformation
- () normalization

(_) perspective division

(_) 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

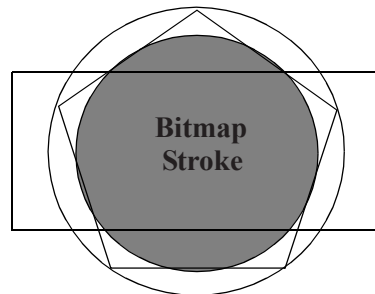
1. Implement `myLoadIdentity`, `myRotatef`, `myTranslatef`, `myScalef`, `myPushMatrix`, and `myPopMatrix` just like their corresponding OpenGL commands. Then, in the rest of the programming assignments, you can interchange them with OpenGL commands.

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.

