Chapter 5

Transformation of Objects

Drawing of objects before and after they are transformed

Composing a picture from many instances of a simple form

Dr M A BERBAR
Composing a 3D scene from primitives.

We can design a single "motif" and then fashion the whole shape by appropriate reflections, rotations, and transformation of the motif.

A designer may want to view an object from vantage points and make a picture from each one.

The scene can be rotated and viewed with the same camera, but as suggested in the figure, it is more natural to leave the scene alone and move the camera to different orientations and positions for each snapshot.

Positioning and reorienting a camera can be carried out through the use of 3D affine transformations.
In a computer animation, several objects must move relative to one another from frame to frame. This effect can be achieved by shifting and rotating the separate local coordinate system of each object as the animation proceeds. Figure 5.6 shows an example.

Transformations

• One example of a transformation is the window to viewport transformation.
• Here we have seen an image in the world window scaled and translated (moved) into a viewport window.
• We will build on this transformation to allow us to move objects to more complex locations.
Transformations

- A Transformation consists of:
  - a Rotation
  - a Scaling and
  - a Translation
- They occur in 2D and 3D

Transformations

- Transformations allow for:
  1. scene composition
Transformations

• Transformations allow for:
  2. easily create symmetrical objects

Transformations

The OpenGL Pipeline

• OpenGL makes transformations easy.
• But that doesn’t excuse you from learning about them…. in detail!!
Affine Transformations

- Every affine transformation can be represented as a composition of translations, rotations, and scalings (in some order)

Translation

- Translation displaces points by a fixed distance in a given direction
- Only need to specify a displacement vector $d$
- Transformed points are given by $P' = P + d$

\[
\begin{pmatrix}
  d_x \\
  d_y
\end{pmatrix} + \begin{pmatrix}
  x \\
  y
\end{pmatrix} = \begin{pmatrix}
  x + d_x \\
  y + d_y
\end{pmatrix}
\]
Are We Headed? Using Transformations with OpenGL

- We will present the basic concepts of affine transformation and show how they produce certain geometric effects,
- A sequence of operations that are applied to all points such as scaling, rotation, translations, both in 2D and 3D space. A drawing is produced by processing each point.
As shown in the figure, these points first encounter a transformation called the "current transformation" (CT), which alters their values into a different set of points, say, Q₁, Q₂, Q₃, ..., just as the original points P describe some geometric object, the points Qᵢ describe the transformed version of the same object.

These points are then sent through additional steps and ultimately are used to draw the final image on the display.

Object Transformations versus Coordinate Transformations

- An **object transformation** alters the coordinates of each point on the object according to some rule, leaving the underlying coordinate system fixed.

- A **coordinate transformation** defines a new coordinate system in terms of the old one and then represents all of the object's points in this new system.
Transforming Points

- A transformation simply takes a point and maps it to another location. A transformation alters each point $P$ in space (2D or 3D) into a new point $Q$ by means of a specific formula or algorithm.

Exercise:
- if $P = (3,4)$ and $Q = (5, 7)$
  - what is $M$?
- $(5,7,1)^T = M (3,4,1)^T$
- We want to increase $P_x$ by 2 and increase $P_y$ by 3
  - what is $M$??

\[
\begin{bmatrix}
Q_x \\
Q_y \\
1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 2 \\
0 & 1 & 3 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
P_x \\
P_y \\
1
\end{bmatrix}
\]
Translation

- This means that values are being added or subtracted to the existing coordinates.

\[
\begin{bmatrix}
Q_x \\
Q_y \\
1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & dP_x \\
0 & 1 & dP_y \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
P_x \\
P_y \\
1
\end{bmatrix}
\]

Example: What is the translation matrix to move \(P=(4,6)\) to \(Q=(10,3)\)?

\[
\begin{bmatrix}
10 \\
3 \\
1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 6 \\
0 & 1 & -3 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
P_x \\
P_y \\
1
\end{bmatrix}
\]

Change in X

Change in Y
Translation

- Example: What is the translation matrix to move \(P=(4,6,2)\) to \(Q=(10,3,5)\)?

\[
\begin{pmatrix}
10 \\
3 \\
5 \\
1
\end{pmatrix} =
\begin{pmatrix}
1 & 0 & 0 & 6 \\
0 & 1 & 0 & -3 \\
0 & 0 & 1 & 3 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
4 \\
6 \\
2 \\
1
\end{pmatrix}
\]

Change in X
Change in Y
Change in Z

Scaling

- This means that the x, y and or z coordinates are being multiplied by a scalar.
Scaling

• Example: What is the matrix that will scale a point $P = (6,2)$ to $Q = (3,4)$

\[
\begin{bmatrix}
3 \\
4 \\
1
\end{bmatrix}
= \begin{bmatrix}
\frac{1}{2} & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
6 \\
2 \\
1
\end{bmatrix}
\]

Scaling

• Example: What is the matrix that will scale a point $P = (6,2,9)$ to $Q = (3,4,3)$

\[
\begin{bmatrix}
3 \\
4 \\
3 \\
1
\end{bmatrix}
= \begin{bmatrix}
\frac{1}{2} & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & \frac{1}{3} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
6 \\
2 \\
9 \\
1
\end{bmatrix}
\]
Scaling

Changing the size of an object. We scale an object by scaling the x and y coordinates of each vertex in the object.

\[ s_x = \frac{w'}{w}, \quad s_y = \frac{h'}{h}, \quad x' = s_x x, \quad y' = s_y y \]

\[
\begin{bmatrix}
  s_x & 0 & x \\
  0 & s_y & y
\end{bmatrix}
= \begin{bmatrix}
  s_x & 0 & x' \\
  0 & s_y & y'
\end{bmatrix}
\]

Fixed Point Scaling

Scale by 2 with fixed point = (2,1)
- Translate the point (2,1) to the origin
- Scale by 2
- Translate origin to point (2,1)

\[
\begin{bmatrix}
  1 & 0 & 2 \\
  0 & 1 & 1 \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  2 & 0 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  1 & 0 & -2 \\
  0 & 1 & 1 \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  2 & 0 & -2 \\
  0 & 1 & -1 \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  4 & 2 \\
  0 & 1 \\
  0 & 0
\end{bmatrix}
\]
Rotation

• This means that the x, y and or z coordinates are rotated around a point.

How do we calculate P rotating to Q?

Rotation

• Use right-angles and trig.

We know:
P(x, y) = (R \cos(\Phi), R \sin(\Phi) )
and
Q(x, y) = (R \cos(\Theta + \Phi), R \sin(\Theta + \Phi) )

From trigonometry we also know:
\cos(\Theta + \Phi) = \cos(\Theta)\cos(\Phi) - \sin(\Theta)\sin(\Phi)
\sin(\Theta + \Phi) = \sin(\Theta)\cos(\Phi) + \cos(\Theta)\sin(\Phi)
Rotation

• Use right-angles and trig.

\[ Q_x = R \cos(\Theta + \Phi) = R \cos(\Theta)\cos(\Phi) - R \sin(\Theta)\sin(\Phi) \]
\[ Q_y = R \sin(\Theta + \Phi) = R \sin(\Theta)\cos(\Phi) + R \cos(\Theta)\sin(\Phi) \]

using \( P(x, y) = (R \cos(\Phi), R \sin(\Phi)) \)

we get

\[ Q_x = P_x \cos(\Theta) - P_y \sin(\Theta) \]
\[ Q_y = P_x \sin(\Theta) + P_y \cos(\Theta) \]

Rotation

• This gives us the rotation matrix:

\[
\begin{bmatrix}
Q_x \\
Q_y \\
1
\end{bmatrix} =
\begin{bmatrix}
\cos(\Theta) & -\sin(\Theta) & 0 \\
\sin(\Theta) & \cos(\Theta) & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
P_x \\
P_y \\
1
\end{bmatrix}
\]
Matrix Representation of 2D Rotation around the origin

We want to find the representation of the transformation that rotates at angle $\theta$ about the origin.

Given a point with coordinates $(x,y)$, what are the coordinates $(x',y')$ of the transformed point?

2D Rotation around the Origin

\[
\begin{align*}
  x &= r \cos \varphi \\
  y &= r \sin \varphi \\
  x' &= r \cos(\theta + \varphi) = r \cos \varphi \cos \theta - r \sin \varphi \sin \theta = x \cos \theta - y \sin \theta \\
  y' &= r \sin(\theta + \varphi) = r \cos \varphi \sin \theta + r \sin \varphi \cos \theta = x \sin \theta + y \cos \theta
\end{align*}
\]

The matrix representing the rotation is:

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix} =
\begin{bmatrix}
  \cos \theta & -\sin \theta \\
  \sin \theta & \cos \theta
\end{bmatrix}
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
\]
Homogeneous Coordinates

In order to represent a transformation as a matrix multiplication operation we use 3 x 3 matrices and pad our points to become 3 x 1 matrices. This coordinate system (using three values to represent a 2D point) is called homogeneous coordinates.

\[ P_{(x,y)} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \]

\[ R_{\theta} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

\[ S_{s_{x},s_{y}} = \begin{bmatrix} s_{x} & 0 & 0 \\ 0 & s_{y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

\[ T_{x,y} = \begin{bmatrix} 1 & 0 & d_{x} \\ 0 & 1 & d_{y} \\ 0 & 0 & 1 \end{bmatrix} \]

Rotation about a Fixed Point

Rotation of \( \theta \) degrees about Point \((x,y)\)
- Translate \((x,y)\) to origin
- Rotate
- Translate origin to \((x,y)\)

\[ C = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} \]

\[ R_{\theta} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

\[ T_{x,y} = \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix} \]
Shearing

- Shearing means that a point is dragged in a particular direction.
- This means that some coordinates are affected while others are not.

![Shearing Diagram](image)

Shearing occurs along a line.
- In this example the shear occurs along the x axis.
- This gives us:
  - $Q_x = P_x + hP_y$
  - $Q_y = P_y$

![Shearing Equation](image)
Translation

-Scaling

-Rotation

-Reflection

Shear

Reflections

Reflection about the y-axis

\[
\begin{bmatrix}
-1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

Reflection about the x-axis

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]
Combining Transformations

- **Rotation, Scaling, Translation** and **Shearing** can be combined into the one matrix.

- For example, if you want to **translate** a shape, **rotate** it and then **scale** it, the transformation, $T$, would be:

$$
\begin{bmatrix}
SP_x & 0 & 0 \\
0 & SP_y & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 \\
\sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & dP_x \\
0 & 1 & dP_y \\
0 & 0 & 1
\end{bmatrix}
$$
Combining Transformations

- Transformation matrices are listed in reverse order and....
- Matrices are multiplied backwards.

| \[ sP_x \cos(\theta) & -sP_x \sin(\theta) & sP_y \sin(\theta) & sP_y \cos(\theta) \\
| \[ sP_x \sin(\theta) & sP_x \cos(\theta) & sP_y \cos(\theta) & sP_y \sin(\theta) \\
| \[ 0 & 0 & 1 & 0 |

Affine Transformations

- What we have been looking at are affine transformation.
- They have the following properties:
  - Affine Transformations Preserve Affine Combinations of Points
  - Affine Transformations Preserve Lines and Planes
  - Affine Transformations Preserve Parallelism of Lines and Planes
  - Relative Ratios are preserved
  - The effect on areas can be predetermined.
PRACTICE EXERCISE

5.2.1 Apply the transformation

An affine transformation is specified by the matrix

\[
\begin{pmatrix}
3 & 0 & 5 \\
-2 & 1 & 2 \\
0 & 0 & 1
\end{pmatrix}.
\]

Find the image \( Q \) of point \( P = (1, 2) \).

Solution: \[
\begin{pmatrix}
8 \\
2 \\
1
\end{pmatrix} =
\begin{pmatrix}
3 & 0 & 5 \\
-2 & 1 & 2 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
1 \\
2 \\
1
\end{pmatrix}.
Transformation of a map:
(a) Translation
(b) Scaling
(c) Rotation
(d) Shear.

Translation:
new x or x' = x + t_x
New y or y' = y + t_y

Scaling:
new x or x' = S_x * x
New y or y' = S_y * y

The textbook uses
Q_x and Q_y \leftrightarrow x' and y'
P_x and P_y \leftrightarrow x and y
Scaling:

\[ \text{new } x \quad \text{or} \quad x' = S_x \cdot x \]
\[ \text{New } y \quad \text{or} \quad y' = S_y \cdot y \]

The scaling in this fashion is called scaling about origin, because each point \( P \) is moved \( S_x \) times farther from the origin in the \( x \)-direction and \( S_y \) times farther from the origin in the \( y \)-direction.

Figure 5.11 shows an example in which the scaling \((S_x, S_y) = (-1, 2)\) is applied to a collection of points.

If the two scale factors are the same \( S_x = S_y = S \), the transformation is a uniform scaling, or a magnification about the origin, with magnification factor \(|S|\).

If the scale factors are not the same, the scaling is called a differential scaling.

PRACTICE EXERCISE

- **5.2.2 Sketch the effect**
- A pure-scaling affine transformation uses scale factors \( S_x = 3 \) and \( S_y = -2 \). and sketch the image of each of the three objects in following Figure 5.12 under this transformation.
- (Make use of the facts—to be verified later—that an affine transformations maps straight lines to straight lines and ellipses to ellipses.)
Example 5.2.1

- Find the transformed point Q caused by rotating P = (3,5) about the origin through an angle of 60°.

Solution

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} = \begin{bmatrix}
  \cos(\theta) & -\sin(\theta) & 0 \\
  \sin(\theta) & \cos(\theta) & 0 \\
  0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix}
\]

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} = \begin{bmatrix}
  0.5 & -0.866 & 0 \\
  0.866 & 0.5 & 0 \\
  0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
  3 \\
  5 \\
  1
\end{bmatrix}
\]
Practice Exercise

• Rotate a point using equation (3.9)
  – (a) (2, 3) through an angle of -45
  – (b) (1, 1) through an angle of -180
  – (c) (60, 61) through an angle of 4

Solution   Page 220

• Repeat the above exercise using the transformation matrix to rotate a point.

Solution

\[
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix} =
\begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 \\
\sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]

Complete the solution by yourself
Shearing

In x direction

- \( x' = x + k_x \cdot y \)
- \( y' = y \)

\[ Q_x = P_x + h \cdot P_y \]
\[ Q_y = P_y \]

In y direction

- \( x' = x \)
- \( y' = y + k_y \cdot x \)

\[ Q_x = P_x \]
\[ Q_y = g \cdot P_x + P_y \]

Example:

Into which point does (3, 4) shear when sharing in x direction parameter equal 0.3.

Solution:

\[
\begin{bmatrix}
1 & \text{SH}_x & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix} = \begin{bmatrix}
1 & k_x & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]
Example 5.2.3:
Let the sharing in y direction with shearing parameter 0.2, to what point (6, -2) map?

\[
\begin{bmatrix}
    x' \\
    y'
\end{bmatrix} =
\begin{bmatrix}
    1 & 0.2 & 0 \\
    0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
    x \\
    y
\end{bmatrix} =
\begin{bmatrix}
    4.2 \\
    4
\end{bmatrix}
\]

PRACTICE EXERCISE
Shearing lines

Consider the shear for which \( g = K_y = 0.4 \) and \( h = K_x = 0 \).

- Experiment with various sets of three collinear points to build some assurance that the sheared points are still collinear.
- Then, assuming that lines do shear into lines, determine into what objects the following line segments shear:
  - [a]. the horizontal segment between (x3, 4) and (2, 4).
  - [b]. the horizontal segment between (x3, x4) and (2, x4).
  - [c]. the vertical segment between (x2, 5) and (x2, x1).
  - [d]. the vertical segment between (2, 5) and (2, x1).
  - [e]. the segment between (x1, x2) and (3, 2);
The Inverse of an Affine Transformation

- Most affine transformations of interest are **nonsingular**, which means that the determinant of the transformation matrix $m$ evaluates to

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

**Transformation matrix**

$$\det M = m_{11} m_{22} - m_{12} m_{21} \text{ is non zero}$$

It is reassuring to be able to undo the effect of a transformation. This is particularly easy to do with nonsingular affine transformations. If point $P$ is mapped into point $Q$ according to $Q = MP$, we simply

$$P = M^{-1} Q$$
PRACTICE EXERCISES

5.2.5 What is the inverse of a rotation?
Show that the inverse of a rotation through $\theta$ is a rotation through $-\theta$. Is this reasonable geometrically? Why?

5.2.6 Inverting a shear
Is the inverse of a shear also a shear? Show why or why not.

5.2.7 An Inverse matrix
Compute the inverse of the matrix

\[
\begin{bmatrix}
3 & 2 & 1 \\
-1 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Composing Affine Transformations

• is rare that we want to perform just one elementary transformation; usually, an application requires that we build a compound transformation out of several elementary ones.

For example, we may want to

1. translate by (3, -4), then
2. rotate through 30°, then
3. scale by (2, -1), then
4. translate by (0, 1.5), and,
5. finally, rotate through -30°.

How do these individual transformations combine into one overall transformation?
EXAMPLE 5.2.4 Build one

- Build a transformation that
  [a]. rotates through 45 degrees,
  [b]. then scales in \( x \) by 1.5 and in \( y \) by -2,
  [c]. and, finally, translates through (3, 5).

Find the image under this transformation of the point (1,2).

**SOLUTION**

- Construct the three matrices and multiply them in the proper order (first one last, etc.) as follows:

\[
\begin{pmatrix}
1 & 0 & 3 \\
0 & 1 & 5 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
1.5 & 0 & 0 \\
0 & -2 & 0 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
.707 & -.707 & 0 \\
.707 & .707 & 0 \\
0 & 0 & 1
\end{pmatrix}
= \begin{pmatrix}
1.06 & -1.06 & 3 \\
-1.414 & -1.414 & 5 \\
0 & 0 & 1
\end{pmatrix}
\]

The point (1, 2) will map to \((1.94, 0.758)\)
EXAMPLE 5.2.5 Rotating about an arbitrary point

all of the rotations we have considered have been about the origin.

But suppose we wish instead to rotate points about some other point in the plane. A suggested in Figure 5.17

Example

• Rotate points through 30° about (-2, 3) and determine into which point the point (1, 2) maps. $\cos(30) = 0.866$ and $\sin(30) = 0.5$

Result :
(1.098, 3.634)

What the point (-2, 3) maps to?
EXAMPLE 5.2.6 Scaling and shearing about arbitrary "pivot" points

- In a manner similar to that of Example 5.2.5, we often want to scale all points about some pivot point other than the origin. Because the elementary scaling operation of Equation (5.7) scales points about the origin,

- we do the same

"shift-transform-unshift"
sequence as for rotations.

EXAMPLE 5.2.7 Reflections about a tilted line

Consider the line through the origin that makes an angle of $\beta$ with the x-axis, as shown in Figure 5.18. Point $A$ reflects into point $B$, and each house shown reflects into the other. We want to develop the transformation that reflects any point $P$ about the line, called the *axis of reflection*, to produce point $Q$. Is this an affine transformation?
1. Rotate so line matches an axis
2. Reflect about that axis
3. Rotate so line back to original orientation

The composite matrix $M$ equal

$$R(\beta) \ast S(sx, sy) \ast R(-\beta)$$

Where: $sx = 1$ $sy = -1$

$$M = \begin{bmatrix}
\cos(2\beta) & \sin(2\beta) & 0 \\
\sin(2\beta) & -\cos(2\beta) & 0 \\
0 & 0 & 1
\end{bmatrix}$$

Combining Affine Transformations

Example:
Suppose we want to reflect an L-shaped object defined by the joins of vertices $(0, 0)$, $(0, 3)$, $(1, 3)$, $(1, 1)$, $(2, 1)$, $(2, 0)$, $(0, 0)$ about a line at $45^\circ$ to the x-axis passing through the point $(2, 1)$.

Solution:
This can be achieved by the sequence of transformations illustrated in Figure. The sequence is not unique. From 1st principles, reflect about any line...

1. Translate so line passes through origin
2. Rotate so line matches an axis
3. Reflect about that axis
4. Rotate so line back to original orientation
5. Translate so line back to original position
1- Translate so line passes through origin

2- Rotate so line matches an axis

3- Reflect about that axis

The steps 2, 3, and 4 could be replaced by

\[ M = \begin{bmatrix} \cos(2\beta) & \sin(2\beta) & 0 \\ \sin(2\beta) & -\cos(2\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

4- Rotate so line back to original orientation

5- Translate so line back to original position
Examples of 2D Transformations:

About origin

About y = -x

Dr M A BERBAR

PRACTICE EXERCISES

• 5.2.8 The classic: The window-to-viewport transformation
• We developed this transformation in Chapter 3. Rewriting Equation 3.2 in the current notation, we have

where the components A, B, C, and D depend on the window and viewport and are given in Equation (3.3).

Show that this transformation is composed of
• a translation through (-W.l, -W.b) to place the lower left corner of the window at the origin,
• a scaling by (A, B) to size things,
• and a translation through (V.l, V.b) to move the viewport to the desired position.

Dr M A BERBAR
Show that this transformation is composed of:

1. A translation through \((-W_l, -W_b)\) to place the lower left corner of the window at the origin,

2. A scaling by \((A, B)\) to size things,

3. and a translation through \((V_l, V_b)\) to move the viewport to the desired position.

Summarizing, we have, for the window-to-viewport transformation,

\[
\begin{bmatrix}
    sx \\
    sy \\
    1
\end{bmatrix}
= \begin{bmatrix}
    A & 0 & -AW_l + V_l \\
    0 & B & -BW_b + V_b \\
    0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    1
\end{bmatrix}
\]

where

\[
sx = Ax + C
\]

and

\[
sy = By + D, \quad (3.3)
\]

with

\[
A = \frac{V_r - V_l}{W_r - W_l}, \quad C = V_l - AW_l
\]

and

\[
B = \frac{V_r - V_b}{W_r - W_b}, \quad D = V_b - BW_b.
\]
5.2.10 Where does it end up?
Where is the point (8, 9) after it is rotated through (3,1)? Find the $M$ matrix.

Object Transformations in 3D

the operations of translation and scaling are obvious extensions of the 2D case.

- **Translation in 3D**
  
  $x' = x + tx,$
  
  $y' = y + ty$
  
  $z' = z + tz.$

- **Scaling in 3D**
  
  $x' = s_x \times x$
  
  $y' = s_y \times y$
  
  $z' = s_z \times z.$
Rotation in 3D

[ $\alpha$ is the rotation angle ]

- **Rotation about x:**
  
  \[ x' = x \]
  \[ y' = y \cos(\alpha) - z \sin(\alpha) \]
  \[ z' = y \sin(\alpha) + z \cos(\alpha) \]

- **Rotation about y:**
  
  \[ y' = y \]
  \[ z' = z \cos(\alpha) - x \sin(\alpha) \]
  \[ x' = z \sin(\alpha) + x \cos(\alpha) \]

- **Rotation about z:**
  
  \[ z' = z \]
  \[ x' = x \cos(\alpha) - y \sin(\alpha) \]
  \[ y' = x \sin(\alpha) + y \cos(\alpha) \]

Matrices, Transformations in 3D

- point \((x, y, z)\) in Cartesian coordinates as when the 'weight' \(w \neq 0\).

we take \(w = 1\) for simplicity, representing the point \((x, y, z)\) as.

\[
\begin{bmatrix}
  x \\
y \\
  z \\
1
\end{bmatrix}
\]
\[
M = \begin{pmatrix}
    m_{11} & m_{12} & m_{13} & m_{14} \\
    m_{21} & m_{22} & m_{23} & m_{24} \\
    m_{31} & m_{32} & m_{33} & m_{34} \\
    0 & 0 & 0 & 1
\end{pmatrix},
\]

\[
\begin{pmatrix}
    Q_x \\
    Q_y \\
    Q_z \\
    1
\end{pmatrix} = \begin{pmatrix}
    P_x \\
    P_y \\
    P_z \\
    1
\end{pmatrix}.
\]

**Translation**

\[
\begin{aligned}
x' &= x + t_x, \\
y' &= y + t_y, \\
z' &= z + t_z.
\end{aligned}
\]

\[
\begin{pmatrix}
    1 & 0 & 0 & m_{14} \\
    0 & 1 & 0 & m_{24} \\
    0 & 0 & 1 & m_{34} \\
    0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
T(t_x, t_y, t_z) = \begin{pmatrix}
    1 & 0 & 0 & t_x \\
    0 & 1 & 0 & t_y \\
    0 & 0 & 1 & t_z \\
    0 & 0 & 0 & 1
\end{pmatrix}.
\]

**Scaling**

\[
\begin{aligned}
x' &= s_x x, \\
y' &= s_y y, \\
z' &= s_z z.
\end{aligned}
\]

\[
\begin{pmatrix}
    S_x & 0 & 0 & 0 \\
    0 & S_y & 0 & 0 \\
    0 & 0 & S_z & 0 \\
    0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
S(s_x, s_y, s_z) = \begin{pmatrix}
    s_x & 0 & 0 & 0 \\
    0 & s_y & 0 & 0 \\
    0 & 0 & s_z & 0 \\
    0 & 0 & 0 & 1
\end{pmatrix}.
\]

**Rotation about the x-axis**

\[
\begin{aligned}
x' &= x, \\
y' &= y \cos(\theta) - z \sin(\theta), \\
z' &= y \sin(\theta) + z \cos(\theta).
\end{aligned}
\]

\[
R_x(\theta) = \begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & \cos(\theta) & -\sin(\theta) & 0 \\
    0 & \sin(\theta) & \cos(\theta) & 0 \\
    0 & 0 & 0 & 1
\end{pmatrix}.
\]

\[c = \cos(\theta), \quad s = \sin(\theta)\]
Rotation about the y-axis
\[ x' = x \cos(\theta) + z \sin(\theta), \]
\[ y' = y, \]
\[ z' = -x \sin(\theta) + z \cos(\theta). \]

Rotation about the z-axis
\[ x' = x \cos(\theta) - y \sin(\theta), \]
\[ y' = x \sin(\theta) + y \cos(\theta), \]
\[ z' = z. \]

Shear with z unchanged
\[ x' = x + k_x z, \]
\[ y' = y + k_y z, \]
\[ z' = z. \]

The shear matrices when the x coordinate unchanged and the y coordinate unchanged.

\[ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & k_x & 0 & 0 \\ 0 & 1 & k_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & k_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Rotation

• What about 3D rotations?
• There are 3 types of 3D rotations:
  • an x roll
  • a y roll
Rotation

• What about 3D rotations?
• There are 3 types of 3D rotations:
  • an x roll
  • a y roll
  • a z roll

Rotation

• A Z roll is the same as rolling in 2D as the object rolls between the x and y axis.

\[
\begin{bmatrix}
Q_x \\
Q_y \\
Q_z \\
1
\end{bmatrix} =
\begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 & 0 \\
\sin(\theta) & \cos(\theta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
P_x \\
P_y \\
P_z \\
1
\end{bmatrix}
\]

Positive angle rotation occurs according to the right-hand rule!!!.
Rotation

• An X roll is a rotation between the y and z axes.

\[
\begin{bmatrix}
Q_x \\
Q_y \\
Q_z \\
1
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(\theta) & -\sin(\theta) & 0 \\
0 & \sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
P_x \\
P_y \\
P_z \\
1
\end{bmatrix}
\]

Rotation

• A Y roll is a rotation between the y and z axes.

\[
\begin{bmatrix}
Q_x \\
Q_y \\
Q_z \\
1
\end{bmatrix}
= 
\begin{bmatrix}
\cos(\theta) & 0 & \sin(\theta) & 0 \\
0 & 1 & 0 & 0 \\
-\sin(\theta) & 0 & \cos(\theta) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
P_x \\
P_y \\
P_z \\
1
\end{bmatrix}
\]
Scaling in the z-direction by 0.5 and in the x-direction by a factor of 2.
Example 5.3.2

- Rotate the point P = (3,1,4) through 30° about y-axis.

Solution

Rotation about the y-axis

\[
\begin{align*}
x' &= x \cos(\theta) + z \sin(\theta), \\
y' &= y, \\
z' &= -x \sin(\theta) + z \cos(\theta).
\end{align*}
\]

\[
R_y(\theta) = \begin{bmatrix}
\cos(\theta) & 0 & \sin(\theta) \\
0 & 1 & 0 \\
-\sin(\theta) & 0 & \cos(\theta)
\end{bmatrix}.
\]

Complete your answer!

Composing 3D Affine Transformations

- The matrix that represents the overall transformation is the product of the individual matrices \( M_1, M_2, M_3, \ldots \), and \( M_n \), as in the two-dimensional case.
A note about Rotation in 3D

- Unlike the other transformations, the order in which rotations are carried out is significant.
- In other words, a rotation of $45^\circ$ around the $z$ axis, followed by a rotation of $-30^\circ$ around the $y$ axis will give entirely different transformed coordinates compared with a rotation of $-30^\circ$ around the $y$ axis, followed by a rotation of $45^\circ$ around the $z$ axis.

EXAMPLE 5.3.3

- What is the matrix associated with an $x$-roll of $45^\circ$, followed by a $y$-roll of $30^\circ$, followed by a $z$-roll of $60^\circ$?

Direct multiplication of the three component matrices (in the proper "reverse" order) yields

\[
\begin{bmatrix}
5 & -0.866 & 0 & 0 \\
0.866 & 5 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \times \begin{bmatrix}
0.866 & 0 & 0.5 & 0 \\
0 & 1 & 0 & 0 \\
-0.5 & 0 & 0.866 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \times \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0.707 & -0.707 & 0 \\
0 & 0.707 & 0.707 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
0.433 & -0.436 & 0.789 & 0 \\
0.75 & 0.66 & -0.047 & 0 \\
-0.5 & 0.612 & 0.612 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.
\]
Finding the Axis and Angle of Rotation

- Euler’s theorem guarantees that any rotation is equivalent to a rotation about some axis.

\[
R_u(\beta) = \begin{pmatrix}
m_{11} & m_{12} & m_{13} & 0 \\
m_{21} & m_{22} & m_{23} & 0 \\
m_{31} & m_{32} & m_{33} & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

Rotation About Axis Parallel to z-axis

Move the cube to the origin

Apply \( R_z(\theta) \)

Move back to original position
Rotations about an Arbitrary Axis

- Euler’s Theorem: Any rotation (or sequence of rotations) about a point is equivalent to a single rotation about some axis through that point.

The Classic Way

Rotation about an axis through the origin.

\[
R_u(\beta) = R_y(-\theta)R_z(\phi)R_x(\beta)R_z(-\phi)R_y(\theta).
\]
Rotation About An Arbitrary Axis Rotation

**Situation:** Given an object whose vertices are defined in the world coordinate system, rotate it by R about an axis defined by two points, P1 and P2.

**Strategy:** transform this arbitrary situation into something specific we know how to handle.

1. Translate one end of the axis to the origin

\[
[P_2 - P_1] = [ u_1, u_2, u_3 ]
\]

\[
a = \sqrt{u_1^2 + u_3^2} \\
b = \sqrt{u_1^2 + u_2^2} \\
c = \sqrt{u_2^2 + u_3^2}
\]
2. Rotate about the y-axis an angle $-\beta$

\[ \cos \beta = \frac{u_3}{a} \]
\[ \sin \beta = \frac{u_1}{a} \]

After $R_y(-\beta)$, $U$ lies in the y-z plane.

3. Rotate about the x-axis through an angle $\mu$

\[ \cos \mu = \frac{a}{\|U\|} \]
\[ \sin \mu = \frac{u_2}{\|U\|} \]

This step aligns $U$ with the z-axis.

4. When $U$ is aligned with the z-axis, apply the original rotation, $R$, about the z-axis.

5. Apply the inverses of the transformations in reverse order.
5.3.8 Classic approach to rotation about an axis

\[ \Theta = 30 \text{ and } \Phi = 45 \text{ and } \beta = 35 \]

\[
\begin{pmatrix}
0.877 & -0.366 & 0.281 & 0 \\
0.445 & 0.842 & -0.306 & 0 \\
-0.124 & 0.396 & 0.910 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

PRACTICE EXERCISES

- 5.3.6 Which ones commute? Consider two affine transformations \( T_1 \) and \( T_2 \). Is \( T_1 T_2 \), the same as \( T_2 T_1 \) when
  - a. They are both pure translations?
  - b. They are both scalings?
  - c. They are both shears?
  - d. One is a rotation and one is a translation?
  - e. One is a rotation and one is a scaling?
  - f. One is a scaling and one is a shear?
Example

- Suppose we want to rescale an object by a factor of 2 in the x direction about the origin, and subsequently rotate it by 45° positively about the x-axis, keeping the point (1, 1, 1) fixed. Remembering that \( \cos(45°) = \sin(45°) = \frac{1}{\sqrt{2}} \),

\[
\begin{align*}
\text{• Scale by 2 in the x direction:} & \quad S(2, 1, 1) = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\text{• Place (1, 1, 1) at the origin:} & \quad T(-1, -1, -1) = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{align*}
\]

Composite matrix:
\[
\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

- Rotate about x-axis by 45°:

\[
R_x(45°) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

Composite matrix:
\[
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & -\sqrt{2} \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & -\sqrt{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

(in this stage, we note that \( 2/\sqrt{2} = \sqrt{2} \), as \( 2 = \sqrt{2} \times \sqrt{2} \)).

- Replace the origin to (1, 1, 1):

\[
T(1, 1, 1) = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]
Affine Transformations

- What we have been looking at are affine transformation.
- They have the following properties:
  - Affine Transformations Preserve Affine Combinations of Points
  - Affine Transformations Preserve Lines and Planes
  - Affine Transformations Preserve Parallelism of Lines and Planes
  - Relative Ratios are preserved
  - The effect on areas can be predetermined.

\[
M = \begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 1/\sqrt{2} & -1/\sqrt{2} & 1 \\
0 & 1/\sqrt{2} & 1/\sqrt{2} & 1 - \sqrt{2} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Affine Transformations

- Affine Transformations Preserve Lines and Planes.
  - preserves linearity and flatness
    - lines stay as lines
    - planes stay flat
Affine Transformations

- The effect on areas can be predetermined.

\[
\text{area after transformation/area before transformation} = |\det T|
\]

- Rotations and Translations do not affect areas or volumes.

However, scaling does affect the area or volume.
Affine Transformations

- Example: If the volume of A is 100 units what will the change in volume of A be if the object have been transformed using the following transformation matrix T

\[
\begin{bmatrix}
0.5 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 0.5
\end{bmatrix}
\]

The new volume of A = \(|\det T| \cdot A = 0.5 \cdot 100 = 50\)

Affine Transformations

- How does this help us in computer graphics.
  - An object will not lose its relative shape when transformed together.
  - Objects will not lose their relative distances when transformed together.
Programming Affine Transformations

• OpenGL works with 4 dimensional matrices.
• However, you don’t have to deal with them directly.
• OpenGL does all the hard work for you.

Programming Affine Transformations

• Take a green square produced with
  • glRecti (x1, y1, x2, y2);
  • glRecti(-50,-50,50,50);
Programming Affine Transformations

• Now we can translate it with:
  \texttt{glTranslated(50,50,0);}

  • \texttt{glTranslated(x,y,z);}

• or we can rotate it with:
  \texttt{glRotate(30,0,0,1);}

  • \texttt{glRotate(degrees, x, y, z);}
Programming Affine Transformations

• or we can scale it with:
  
  glScaled(0.5,0.5,1);

• glScaled(x, y, z);

Programming Affine Transformations

• or we can combine them:

  glRotated(30,0,0,1);
  glScaled(0.5,0.5,1);
  glTranslated(50,50,0);
Programming Affine Transformations

• And as we have to multiply the transformation matrices in reverse order we also have to specify them in reverse in OpenGL.
• E.g. To draw the rectangle which we translate, scale and then rotate the code is:
  
  ```
  glRotated(30,0,0,1);
  glScaled(0.5,0.5,1);
  glTranslated(50,50,0);
  glRecti(-50,-50,50,50);
  ```

if we use the Canvas class developed in Chapter 3 (and the global canvas object cvs), there would be a number of calls to `moveTo()` and `lineTo()`, as in the code

```
cvs.moveTo(V[0]);
cvs.lineTo(V[1]);
cvs.lineTo(V[2]);
... // the remaining points
```

In either case, we would set up a world window and a viewport with calls like

```
cvs.setWindow(...);
cvs.setViewport(...);
```

and we would be assured that all vertex positions V[i] are "quietly" converted from world coordinates to screen window coordinates by the underlying window-to-viewport transformation.

**But how do we arrange matters so that house #2 is drawn instead?**

There is the hard way, and there is the easy way.
We have version 1 of the house defined (vertices set), but what we really want to draw is version 2. We could write routines to transform the coordinates – this is the hard way.

- The easy way lets GL do the transforming.

---

The Hard Way

- With this approach, we construct the matrix for the desired transformation, say, M, and build a routine, say, transform2D(), that transforms one point into another, such that
  - Q = transform2D(M, P);
- The routine produces $Q = MP$. To apply the transformation to each point $V[i]$ in house(), we must adjust the earlier source code, as in

```java
cvs.moveTo(transform2D(M, V[0])); // move to the transformed point
cvs.lineTo(transform2D(M, V[1]));
cvs.lineTo(transform2D(M, V[2]));
...```

- so that the transformed points are sent to moveTo() and lineTo(). This adjustment is workable if the source code for house() is at hand. But if the source code for house() is not available. Also, tools are required to create the matrix $M$ in the first place.
```cpp
#define HSIZE 5
Canvas cvs(500, 500, "Hard Way House"); // global canvas object
Point2 h[HSIZE]; // house points

Point2 transform2D( float M[3][3], Point2 P )
{
    Point2 p;
    p.set( P.getX()*M[0][0] + P.getY()*M[0][1] + M[0][2],
           P.getX()*M[1][0] + P.getY()*M[1][1] + M[1][2]);
    return p;
}

void myInit()
{
    h[0].set(0,0);
    h[1].set(0,50);
    h[2].set(25,75);
    h[3].set(50,50);
    h[4].set(50,0);
}

void drawHouse()
{
    float M[3][3] = { {0.866, -0.5, 70},
                      {0.5, 0.866, 60},
                      {0, 0, 1} };

    // draw original house in yellow
    cvs.setColor(1.0, 1.0, 0.0);
    cvs.moveTo(h[0]); // start point of drawing
    for(int i = 1; i < HSIZE; i++)
        cvs.lineTo(h[i]);
    cvs.lineTo(h[0]);

    // draw transformed house in green
    cvs.setColor(0.0, 1.0, 0.0);
    cvs.moveTo(transform2D(M, h[0])); // start point of drawing
    for(i = 1; i < HSIZE; i++)
        cvs.lineTo(transform2D(M, h[i]));
    cvs.lineTo(transform2D(M, h[0]));
}
```
void myDisplay(void)
{
    cvs.clearScreen();
    cvs.drawAxis(1.0, 0, 0);
    glLineWidth(2.0);
    drawHouse();
    glFlush(); // send all output to display
}

void main()
{
    cvs.setWindow(-200.0, 200.0, -200.0, 200.0);
    cvs.setViewport(0, 400, 0, 400);
    cvs.setBackgroundColor(0.0, 0.0, 0.0);
    glutDisplayFunc(myDisplay);
    myInit();
    glutMainLoop();
}

The Easy Way

• When glVertex2d () is called with the argument V, the vertex V is first transformed by the CT to form point Q, which is then passed through the window-to-viewport mapping to form point S in the screen window.
• This additional mapping is done automatically! OpenGL maintains a so-called **modelview** matrix, and every vertex that is passed down the graphics pipeline is multiplied by this matrix.

• We need only set up the **modelview** matrix to embody the desired transformation.

  \begin{align*}
  \text{The principal routines for altering the \textit{modelview} matrix are } & \textit{glRotated()}, \textit{glScaled()}, \text{and } \textit{glTranslated}(). \\
  \text{These do not set the CT directly; instead, each \textit{postmultiplies} the CT (the \textit{modelview} matrix) by a particular matrix, say, } & M, \text{ and puts the result back into the CT.}\\
  \text{That is, each of the routines creates a matrix } & M \text{ required for the new transformation and performs the operation } \\
  \textbf{CT} = \textbf{CT} \times M
  \end{align*}
The following are OpenGL routines for applying transformations in the 2D case:

- `glScaled(sx, sy, 1.0);`
- `glTranslated(dx, dy, 0);`
- `glRotated(angle, 0, 0, 1);`

Put the result back into CT.

1. Get started:
   initialize the CT to the identity transformation. For that purpose OpenGL provides the routine `glLoadIdentity()`.

2. And because the functions listed can be set to work on any of the matrices that OpenGL supports, we must inform OpenGL which matrix we are altering.

This is accomplished using `glMatrixMode(GL_MODELVIEW).`

```c
void setWindow(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(left, right, bottom, top);
}
```

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

This GL_PROJECTION will be explained later

This GL_MODELVIEW is our concern now
void drawHouse()
{
    cvs.moveTo(h[0]);
    for(int i = 1; i < HSIZE; i++)
        cvs.lineTo(h[i]);
    cvs.lineTo(h[0]);
}

void myDisplay(void)
{
    cvs.clearScreen();
    cvs.drawAxis(1.0,0,0);
    glLineWidth(2.0);
    cvs.setColor(1.0,1.0,0.0);
    drawHouse();
    cvs.setColor(0.0,1.0,0.0);
    glPushMatrix();
    glRotated(30,0,0,1);
    glTranslated(70,60,0);
    drawHouse();
    glPopMatrix();
    glFlush();
}
glPushMatrix();
glRotated(80, 0, 0, 1);
glTranslated(70, 60, 0);
drawHouse();
glPopMatrix();

glPushMatrix();
glRotated(50, 0, 0, 1);
glTranslated(70, 60, 0);
drawHouse();
glPopMatrix();

glPushMatrix();
glRotated(-30, 0, 0, 1);
glTranslated(80, 100, 0);
drawHouse();
glPopMatrix();

glPushMatrix();
glRotated(-80, 0, 0, 1);
glTranslated(80, 100, 0);
drawHouse();
glPopMatrix();
glPushMatrix() & glPopMatrix(),

- Since the transformations are stored as matrices, a matrix stack provides an ideal mechanism for doing this sort of successive remembering, translating, and throwing away. All the matrix operations that have been described so far (glLoadMatrix(), glMultMatrix(), glLoadIdentity(), and the commands that create specific transformation matrices) deal with the current matrix, or the top matrix on the stack. You can control which matrix is on top with the commands that perform stack operations:
  - glPushMatrix(), which copies the current matrix and adds the copy to the top of the stack, and
  - glPopMatrix(), which discards the top matrix on the stack, as shown in the following. (Remember that the current matrix is always the matrix on the top.) In effect, glPushMatrix() means "remember where you are" and glPopMatrix() means "go back to where you were."

Affine Transformations Stack

- It is also possible to push/pop the current transformation from a stack in OpenGL, using the commands
  
  glMatrixMode (GL_MODELVIEW);
  glPushMatrix(); //or glPopMatrix();

a). before  b). after pushCT()  c). after rotate2D()  d). after popCT()
The implementation of `pushCT()` and `popCT()` uses OpenGL routines `glPushMatrix()` and `glPopMatrix()`.

**Caution:** Note that each routine must inform OpenGL which matrix stack is being affected.

In OpenGL, popping a stack that contains only one matrix is an error; test the number of matrices using OpenGL’s query function `glGet(GL_MODELVIEW_STACK_DEPTH)`.

---

```c
void pushCT()
{
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix(); // push a copy of the top matrix
}

void checkStack()
{
    if (glGet (GL_MODELVIEW_STACK_DEPTH) ≤ 1)
    {
        // do something
    }
    else
    {
        popCT();
    }
}

void popCT()
{
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix(); // pop the top matrix from the stack
}
```
Example 5: Motif

- **Tilings** are based on the repetition of a basic motif both horizontally and vertically.
- Consider tiling the window with some motif, drawn centered in its own coordinate system by routine `motif()`.
- Copies of the motif are drawn $L$ units apart in the $x$-direction, and $D$ units apart in the $y$-direction, as shown in part b).

Example 5 (2)

- The motif is translated horizontally and vertically to achieve the tiling.
Transformations

- Transformations allow for:
  1. scene composition
  2. easily create symmetrical objects
A single snowflake spoke would look like that below, where the origin of the window is in the very centre. The OpenGL code to draw this spoke is:

```cpp
void flakeMotif()
{
    cva.moveTo(0,5);
    cva.lineTo(20,5);
    cva.lineTo(30,25);
    cva.lineTo(35,18);
    cva.lineTo(25,5);
    cva.lineTo(15,5);
    cva.lineTo(10,15);
    cva.lineTo(50,13);
    cva.lineTo(25,5);
    cva.lineTo(55,5);
    cva.lineTo(60,0);
}
```

To create the other half of the spoke, the above function can be used again, however this time the coordinates of the spoke need to be mirrored to produce the image below.

This part of the snowflake is created by calling `flakeMotif()` once, flipping the coordinate system and then calling `flakeMotif` again, thus:

```cpp
flakeMotif();
glScaled(1.0,-1.0,1.0);
flakeMotif();
```

The `glScaled` is used to scale the y coordinates by –1. This simply multiplies the y coordinates by –1, thus flipping them!
• The next spoke of the snowflake can be created by rotating the coordinate system by 60 degrees and redrawing the spoke using the code above. The OpenGL function to rotate the coordinate system 60 degrees will be:
  • glRotated(60,0,0,1);
  • This rotates the coordinates 60 degrees around the vector (0,0,1) which is the z axis.
  • Therefore, the snowflake is drawn rotating from x to y around the origin.

• Your Turn
• Write a complete OpenGL program to draw the snowflake as shown at the beginning of this. When complete, modify the program to draw a flurry of snowflakes in random colours.

```c
#include <stdio.h>
#include <GL/glut.h>
#include "canvas.h"
#include <stdlib.h>
#include <math.h>

#define getrandom(min, max) ((rand()%(int)(((max) + 1)?(min)))+ (min))

Canvas cvs(500, 500, "A Flurry of Flakes"); // global canvas object

void flakeMotif()
{
  cvs.moveTo(0,5);
  cvs.lineTo(20,5);
  cvs.lineTo(30,25);
  cvs.lineTo(35,18);
  cvs.lineTo(25,5);
  cvs.lineTo(30,5);
  cvs.lineTo(45,15);
  cvs.lineTo(50,13);
  cvs.lineTo(35,5);
  cvs.lineTo(55,5);
  cvs.lineTo(60,0);
}

void drawFlake()
{
  for(int i = 0; i < 6; i++)
  {
    flakeMotif();
    glScaled(1.0,-1.0,1);
    flakeMotif();
    glScaled(1.0,-1.0,1);
    glRotated(60.0,0,0,1);
  }
}
```

Dr. M. A. BERBAR
void myDisplay(void)
{
    cvs.clearScreen();
    glLineWidth(1.0);

    for(int i = 0; i < 50; i++)
    {
        cvs.initCT();
        cvs.setColor(getRandom(0,100)/100.0,getRandom(0,100)/100.0,getRandom(0,100)/100.0);
        glTranslated(getRandom(x200,200),getRandom(x200,200),0);
        float scale = getRandom(0,100)/100.0;
        glScaled(scale,scale,1);
        drawFlake();
    }

    glFlush(); // send all output to display
}

void main()
{
    cvs.setWindow(-200.0, 200.0, -200.0, 200.0);
    cvs.setViewport(-200, 200, -200, 200);
    cvs.setBackgroundColor(0.0, 0.0, 0.0);
    glutDisplayFunc(myDisplay);
    glutMainLoop();
}
The Camera in OpenGL

- The camera is created with a matrix.
  - We will study the details of how this is done in Chapter 7.
- For now, we just use an OpenGL tool to set up a reasonable camera so that we may pay attention primarily to transforming objects.

- World Coordinate System (Object Space) - The space in which the application model is defined. The representation of an object is measured in some physical or abstract units
- Screen Coordinate System (Image Space) - The space in which the image is displayed. Usually measured in pixels, but could use any units
- Window - The rectangle defining the part of the world we wish to display
Contents

- What is Rendering?
- Rendering Techniques
  - Pipeline-Based Rendering
  - Ray-Tracking

Graphics & Rendering

What is Computer Graphics?

- Mathematical or geometric description of objects
- Rendering
- 2D images simulating the appearance of a real object
- Image
Rendering

- 3d to 2d projections, known as "rendering"
  - World and camera views
  - Mapping 3D data to 2D
  - Rotating objects to acquire a better viewing angle

Rendering Objects

- We know how to model mesh objects, manipulate a jib camera, view objects, and make pictures.
- Now we want to make these objects look visually interesting, realistic, or both.
- We want to develop methods of rendering a picture of the objects of interest: computing how each pixel of a picture should look.
Rendering Objects (2)

- Much of rendering is based on different shading models, which describe how light from light sources interacts with objects in a scene.
  - It is impractical to simulate all of the physical principles of light scattering and reflection.
  - A number of approximate models have been invented that do a good job and produce various levels of realism.

World and Camera Views

- In a 3D world, 3 axes denoting the X, Y and Z axes of three dimensional world,
- and a very basic interpretation of what a camera looks like. The grayish cube is the body of the camera, the black cube on top of it is the camera's lens, and the line coming out of the camera's bottom is supposed to depict a leg at the bottom of the camera, so we can understand what's up and down from the camera's viewpoint.
- In the right panel of the picture you can see what the camera sees, it's a camera view of our world,
Mapping 3D data to 2D

• The main goal of a renderer is to transform and project the 3D coordinates of the objects to the 2D screen.
• **Assumes**: (For demonstration)
  – the camera at a fixed position.
  – the camera always looks up the Z direction, and always looks perpendicular to the XY plane.
  – The camera is placed in such a position that it sees the axes on the XY plane similar to the X and Y axes on a computer screen.
  – the X direction on a computer screen goes from left to right, where the Y direction goes from the top of your screen to the bottom of your screen.

As we’re looking along the Z axis, all objects that should be rendered on our screen appear smaller as their coordinates have bigger Z values.

Similarly, objects appear bigger as their Z values are smaller, since their coordinates are closer to the camera.

In order to transform 3D coordinates to 2D coordinates, we can thus simply divide the X and Y values in the coordinates by the Z value in order to display them as X and Y coordinates in a 2 dimensional view.

\[ x_{2d} = \left( x_{3d} / z_{3d} \right); \]
\[ y_{2d} = \left( y_{3d} / z_{3d} \right); \]
Mapping 3D data to 2D

• Some Z values will be negative, and dividing a X, Y coordinate by this negative Z value will not make it bigger, but it will reverse the sign of our X and Y coordinates and we would be scaling these coordinates the wrong way.

• In order to overcome this problem, we add a number to the Z value so all Z values are strictly positive before we divide the X and Y values by this Z. This extra number is called a distance.

Mapping 3D data to 2D

• After computed the 2 dimensional X and Y values of the projected coordinates, it needs to scale the new coordinates up or down to make sure that the objects fit your viewport.
  – To use a scaling factor of your 2D coordinates so it fits your screen

• 2D image would be appear around the top left corner of the computer screen, since the origin of a computer screen sits at the top left of the screen
  – To add an X and Y offset to your computed 2D coordinates in order to put the projected objects in the middle of the screen

• The resulting code for projecting a 3D coordinate to a 2D coordinate using a camera following those assumption will look like,
  \[
  x_{2d} = x_{Offset} + scale \cdot \frac{x_{3d}}{z_{3d} + distance}; \\
  y_{2d} = y_{Offset} + scale \cdot \frac{y_{3d}}{z_{3d} + distance};
  \]
More About Rendering

Scene Definitions
Rendering: Transformations

- So far, discussion has been in *screen space*
- But model is stored in *model space* (a.k.a. object space or world space)
- Three sets of geometric transformations:
  - *Modeling transforms*
  - *Viewing transforms*
  - *Projection transforms*
The Rendering Pipeline: 3-D

Result:
- All vertices of scene in shared 3-D "world" coordinate system
Rendering: Transformations

- Modeling transforms
  - Size, place, scale, and rotate objects and parts of the model w.r.t. each other

The Rendering Pipeline: 3-D

Result:
- Scene vertices in 3-D "view" or "camera" coordinate system
Rendering: Transformations

• Viewing transform
  – Rotate & translate the world to lie directly in front of the camera
    • Typically place camera at origin
    • Typically looking down -Z axis
  – World coordinates \( \rightarrow \) view coordinates

The Rendering Pipeline: 3-D

Result:
• 2-D screen coordinates of clipped vertices
3D Programming

• At first it will seem complex, but it’s really quite easy.

The 2D drawing so far is a special case of 3D viewing, based on a simple parallel projection. The eye is looking along the z-axis at the world window, a rectangle in the xy-plane.
The Viewing Process and the Graphics Pipeline (2)

- *Eye* is simply a point in 3D space.
- The “orientation” of the eye ensures that the view volume is in front of the eye.
- Objects closer than *near* or farther than *far* are too blurred to see.

The Viewing Process and the Graphics Pipeline (3)

- The **view volume** of the camera is a rectangular parallelepiped.
- Its side walls are fixed by the window edges; its other two walls are fixed by a *near plane* and a *far plane*.
The Viewing Process and the Graphics Pipeline (4)

- Points inside the view volume are projected onto the window along lines parallel to the z-axis.
- We ignore their z-component, so that the 3D point \((x_1, y_1, z_1)\) projects to \((x_1, y_1, 0)\).
- Points lying outside the view volume are clipped off.
- A separate **viewport transformation** maps the projected points from the window to the viewport on the display device.

The Viewing Process and the Graphics Pipeline (5)

- In 3D, the only change we make is to allow the camera (eye) to have a more general position and orientation in the scene in order to produce better views of the scene.
The Viewing Process and the Graphics Pipeline

(6)

- The z axis points toward the eye. X and y point to the viewer’s right and up, respectively.
- Everything outside the view volume is clipped.
- Everything inside it is projected along lines parallel to the axes onto the window plane (parallel projection).

The Viewing Process and the Graphics Pipeline

(7)

- OpenGL provides functions for defining the view volume and its position in the scene, using matrices in the graphics pipeline.
The Viewing Process and the Graphics Pipeline (8)

- The OpenGL pipeline: modelview matrix, projection matrix, viewport matrix.

Each vertex of an object is passed through this pipeline using `glVertex3d(x, y, z)`. The vertex is multiplied by the various matrices shown, it is clipped if necessary, and if it survives clipping, it is ultimately mapped onto the viewport.

- Each vertex encounters three matrices:
  - The modelview matrix;
  - The projection matrix;
  - The viewport matrix;
The Modelview Matrix

- The **modelview matrix** is the $CT$ (current transformation).
- It combines **modeling transformations** on objects and the transformation that orients and positions the camera in space (hence **modelview**).
- It is a single matrix in the actual pipeline.
  - For ease of use, we will think of it as the product of two matrices: a modeling matrix $M$, and a viewing matrix $V$.
  - The modeling matrix is applied first, and then the viewing matrix, so the modelview matrix is in fact the product $VM$.

The Modelview Matrix (M)

- A **modeling** transformation $M$ scales, rotates, and translates **the cube** into the block.

The $V$ matrix is now used to rotate and translate the block into a new position.

- Scene's vertices into the camera's coordinate system.

- The camera moves from its position in the scene to its generic position (eye at the origin and the view volume aligned with the $z$-axis).
The Modelview Matrix (V)

- The *V* matrix rotates and translates the block into a new position.

- The camera moves from its position in the scene to its generic position (eye at the origin and the view volume aligned with the *z*-axis).

Summary (1)

- The modelview matrix basically provides what we have been calling the CT. It combines two effects: the sequence of modeling transformations applied to objects and the transformation that orients and positions the camera in space.

- Although the modelview matrix is the product of two matrices: a modeling matrix *M* and a viewing matrix *V*. First the modeling matrix is applied and then the viewing matrix, so the modelview matrix is in fact the product VM.

FIGURE 5.53 Effect of the modelview matrix in the graphics pipeline, (a) Before the transformations, (b) After the modeling transformation. (b) (c) After the modelview transformation.
Summary (2)

• The $V$ matrix is now used to rotate and translate the block into a new position. The specific transformation is that which would carry the camera from its position in the scene to its "generic" position, with the eye at the origin and the view volume aligned with the z-axis, as shown in Part (c) of the figure.

• The matrix $V$ in fact effects a change of coordinates of the scene’s vertices into the camera's coordinate system. (Camera coordinates are sometimes also called eye coordinates.)

• In the camera's coordinate system, the edges of the view volume are parallel to the x-, y-, and z-axes.
• The view volume extends
  – from left to right in x,
  – from bottom to top in y, and
  – from near to far in z.

Setting Up the Camera

• We shall use a jib camera.
• The photographer rides at the top of the tripod.
• The camera moves through the scene bobbing up and down to get the desired shots.
Setting Up the Scene (2)

```c
glMatrixMode (GL_MODELVIEW);
    // set up the modelview matrix
glLoadIdentity ();    // initialize modelview matrix

    // set up the view part of the matrix

    // do any modeling transformations on the scene
```

Setting Up the Camera (View Matrix)

```c
glMatrixMode (GL_MODELVIEW);
    // make the modelview matrix current
glLoadIdentity();      // start with identity matrix

    // position and aim the camera
    gluLookAt (eye.x, eye.y, eye.z,  // eye position
               look.x, look.y, look.z,   // the “look at” point
               0, 1, 0)     // approximation to true up direction

    // Now do the modeling transformations
```
[1] 3D Programming

- **Modelview Matrix**
  - besides glTranslated, glScaled and glRotated…

  Position and aim the camera with

  ```
  gluLookAt(eye.x, eye.y, eye.z, lookat.x, lookat.y, lookat.z, up.x, up.y, up.z);
  ```

  creates the view matrix. (x, y, z)

  ![Diagram of camera position and up direction](image)

  - eye position
  - up direction (which way is up?)

---

**OpenGL Viewing**

- In OpenGL model and camera transformation are not separate, but **combined to single model-view** matrix

- To set camera position/orientations in OpenGL:
  - Model-view matrix: `glMatrixMode (GL_MODELVIEW)`
  - Load Identity: `glLoadIdentity ()`
  - Specify "look-at" position (multiply on the right):
    ```
    gluLookAt (ex, ey, ez, ax ay az, ux uy uz)
    ```
  - Camera eye position: `(ex, ey, ez)`
  - Look-at reference point position: `(ax, ay, az)`
  - Up direction: `(ux, uy, uz)`
• **gluLookAt** function creates the view matrix and postmultiplies the current matrix by it.
• The function takes as parameters the **eye position**, eye, of the camera and the **look-at point**, look. It also takes an **approximate upwards direction**, up.
• It is usually straightforward to choose reasonable values for eye and look for a good first view. And up is most often set to \((0, 1, 0)\) to suggest an upwards direction parallel to the \(y\)-axis.

**We want this function to set the \(V\) part of the modelview matrix \(VM\).**
• So it is invoked before any modeling transformations are added, since subsequent such transformations will postmultiply the modelview matrix. So to use **gluLookAt()**, we employ the following sequence:

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
//gluLookAt(eye.x, eye.y, eye.z, lookat.x, lookat.y, lookat.z, up.x, up.y, up.z);
gluLookAt(4.0, 4.0, 4.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0);
```

When eye = \((4, 4, 4)\), looking at the origin with look = \((0, 1, 0)\). The upwards direction is set to up = \((0, 1, 0)\), near = 1 and far = 50.

---

**Parallel Projection View Volume**

- Parallelepiped View Volume
- Front Plane
- Back Plane
**[2] 3D Programming**

**The Projection Matrix (1)**

- The *projection matrix* scales and translates each vertex so that those inside the view volume will be inside a *standard cube* that extends from -1 to 1 in each dimension (Normalized Device Coordinates).

- Setting the Projection Matrix:
  - `glMatrixMode(GL_PROJECTION);`
  - `glLoadIdentity();` // initialize projection matrix
  - `glOrtho(left, right, bottom, top, near, far);` // sets the view volume parallelepiped. (All arguments are `glDouble ≥ 0.0`.)
Setting Up the Projection

```c
glMatrixMode(GL_PROJECTION); // make the projection matrix current
glLoadIdentity(); // set it to the identity matrix
glOrtho(left, right, bottom, top, near, far); // multiply it by the new matrix
```

- Using 2 for `near` places the near plane at \( z = -2 \), that is, 2 units in front of the eye.
- Using 20 for `far` places the far plane at \(-20, 20\) units in front of the eye.

(camera at the origin looking along \(-z\)).

Projection Matrix (3)

set with `glOrtho(left, right, bottom, top, near, far)`

- Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. The near clipping plane is a rectangle with the lower left corner at \((left, bottom, -near)\) and the upper right corner at \((right, top, -near)\). The far clipping plane is a rectangle with corners at \((left, bottom, -far)\) and \((right, top, -far)\). Both `near` and `far` can be positive or negative.
- Notice the minus signs before near and far: Because the default camera is located at the origin looking down the negative z-axis.
OpenGL Projection

- Actual projection set by projection matrix
  - Select projection matrix mode:
    - `glMatrixMode (GL_PROJECTION)`
  - Same commands to set matrix as for GL_MODELVIEW
- Projection matrix specifies parallel or perspective projection parameters
- Projection matrix essentially defined by selecting a viewing volume

OpenGL Orthographic Viewing

```
glOrtho (xmin, xmax, ymin, ymax, near, far);
```

*near and far measured from camera*
The Modelview Matrix Stack

Note: As you've seen earlier in this chapter, the modelview matrix contains the cumulative product of multiplying viewing and modeling transformation matrices. Each viewing or modeling transformation creates a new matrix that multiplies the current modelview matrix; the result, which becomes the new current matrix, represents the composite transformation. The modelview matrix stack contains at least thirty-two 4 × 4 matrices; initially, the topmost matrix is the identity matrix. Some implementations of OpenGL may support more than thirty-two matrices on the stack.

The Projection Matrix Stack

The projection matrix contains a matrix for the projection transformation, which describes the viewing volume. Generally, you don't want to compose projection matrices, so you issue glLoadIdentity() before performing a projection transformation. Also for this reason, the projection matrix stack needs to be only two levels deep; some OpenGL implementations may allow more than two 4 × 4 matrices. (You can use glGetIntegerv() with GL_MAX_PROJECTION_STACK_DEPTH as the argument to find the stack depth.)
• Note that these steps correspond to the order in which you specify the desired transformations in your program, not necessarily the order in which the relevant mathematical operations are performed on an object's vertices.

• The viewing transformations must precede the modeling transformations in your code, but you can specify the projection and viewport transformations at any point before drawing occurs. The following Figure shows the order in which these operations occur on your computer.
3D Programming

• Elementary Shapes
  – the following are drawn with the myInit() function set up thus:

```c
myInit()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-100,100,-100,100,0,200);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 150.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

Example

```c
glMatrixMode (GL_PROJECTION);
    // set the view volume (world coordinates)
glLoadIdentity();
glOrtho (-3.2, 3.2, -2.4, 2.4, 1, 50);
glMatrixMode (GL_MODELVIEW);
    // place and aim the camera
glLoadIdentity ();
gluLookAt (4, 4, 4, 0, 1, 0, 0, 1, 0);
    // modeling transformations go here
```
Viewport Matrix

- Viewport Matrix
  - set with glViewport()

```c
void glViewport(GLint x, GLint y, GLint width, GLint height);
```
which sets the viewport to have a lower left corner of \((x, y)\) and an upper right corner of \((x + width, y + height)\).

And finally, the **viewport matrix** maps the surviving portion after clipping of the block into a "3D viewport."

This matrix maps the standard cube into a block shape whose \(x\) and \(y\) values extend across the viewport (in screen coordinates) and whose \(z\)-component extends from \(0\) to \(1\) and retains a measure of the depth of point (the distance between the point and the eye of the camera), as shown in Figure 5.55.
void myInit()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // glOrtho(left, right, bottom, top, near, far)
    glOrtho(-100, 100, -100, 100, 0, 200);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //gluLookAt(eye.x, eye.y, eye.z, lookat.x, lookat.y, lookat.z, up.x, up.y, up.z);
    gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

void main(int argc, char **argv)
{
    glutInitWindowSize(640,480);
    glutInitWindowPosition(100,100);
    glutCreateWindow("3D Window");
    glutDisplayFunc(displayWire);
    //glViewport(GLint x, GLint y, GLint width, GLint height);
    glViewport(0, 0, 640, 480);
    myInit();
    glutMainLoop();
}

Note: The OpenGL pipeline: modelview matrix, projection matrix, viewport matrix.

Chapter 5
Transformation of Objects
Part 7
3D Programming

• Elementary Shapes
  – Wireframe Solids
    • Cubes, Spheres etc…

```c
glutWireCube(GLdouble size);
size is the length of a side
```
```c
#include <windows.h>
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>

//<<<<<<<<<<< myinit >>>>>>>>>>>>>
void myInit()
{
    glMatrixMode(GL_PROJECTION); // set the view volume shape
    glLoadIdentity();
    // glOrtho(left, right, bottom, top, near, far)
    glOrtho(-5.0*64/48.0, 5.0*64/48.0, -5.0, 5.0, -0.1, 100);

    glMatrixMode(GL_MODELVIEW); // position and aim the camera
    glLoadIdentity();
    gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    glClear(GL_COLOR_BUFFER_BIT); // clear the screen
    glColor3d(0, 0, 0); // draw black lines
}

//<<<<<<<<<<< displayWire cube >>>>>>>>>>>>>>>>>>>
void displayWire(void)
{
    glPushMatrix();
    glTranslated(0.5, 0.5, 0.5); // big cube at (0.5, 0.5, 0.5)
    glutWireCube(3.0);
    glPopMatrix();
    glFlush();
}

//<<<<<<<<<<< main >>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Transformation testbed - wireframes");
    glViewport(0, 0, 640, 480);
    myInit();
    glutDisplayFunc(displayWire);
    glClearColor(1.0f, 1.0f, 1.0f, 0.0f); // background is white
    glutMainLoop();
}
```
glutWireSphere(GLdouble radius, GLint slices, GLint stacks);

slices are the vertical cuts (slices of cake).

stacks are the horizontal cuts (stacks of pancakes)

---

void displayWire(void)
{
    glPushMatrix();
    glTranslated(1.0, 1.0, 0); // sphere at (1,1,0)
    glutWireSphere(2.5, 20, 18);
    glPopMatrix();
}
3D Programming

```c
void displayWire(void)
{
    glPushMatrix();
    glTranslated(0, 1.0, 0); // torus at (0,1,0)
    glRotated(90.0, 1, 0, 0);
    glutWireTorus(0.5, 3, 20, 30);
    glPopMatrix();
}
```

In this code snippet, `glutWireTorus` is used to create a wireframe torus. The parameters are as follows:

- `inRad`: the centre radius
- `outRad`: the outer radius

The function is called with the centres as 

- `inRad`: 0.5
- `outRad`: 3
- `slices`: 20
- `stacks`: 30

These parameters determine the shape and appearance of the torus.
glPushMatrix();
    glutWireCone(2.0, 5, 40, 18);
glPopMatrix();

3D Programming

glutWireTetrahedron();

unit in size (use glScaled to make bigger) – 4 planes
3D Programming

`glutWireOctahedron();
unit in size (use glScaled to make bigger) – 8 planes`

Dr M A BERBAR

3D Programming

`glutWireDodecahedron();
unit in size (use glScaled to make bigger) – 10 planes`

Dr M A BERBAR
3D Programming

```c
glutWireIcosahedron();
unit in size (use glScaled to make bigger) – 20 planes
```

Dr M A BERBAR
3D Programming

And the best shape of them all…. 

```c
 glutWireTeapot(GLfloat size);
```
Programming Affine Transformations

• We can also transform 3D objects such as the glutWireTeapot(), thus:

  glRotated(30,0,1,1);
  • **This rotates the coordinates 60 degrees around the vector (0,1,1) which lies on y-z plane.**

  glScaled(0.5,1.5,0.2);

  glTranslated(10,50,0);
3D Programming

- What about solid shapes?
  - glutSolidSphere
  - glutSolidCube
  - etc....
  - and you guessed it...
  - glutSolidTeapot
- However, drawing these isn’t as straight forward as the 2D shapes and the wireframes.....
- Let's look at the result of changing from glutWireTeapot to glutSolidTeapot in the previous program.

What happened?
Why can’t we see a solid teapot?
3D Programming

• For many reasons, but in short… LIGHTING!!!
• Without lighting, there are no:
  – shadows or highlights
• These provide us with the illusion of depth and 3 dimensions.

3D Programming

• Lets turn on the lights.
• glEnable() is used to turn many OpenGL features on..
• One of these is lighting, thus:
  • glEnable(GL_LIGHTING);
    – this can be called in the main or the myInit when the window is being initialized.
3D Programming

- Now, what does the teapot look like?
- Not exactly the effect we were hoping for!
- So far we have enabled lighting, but we haven’t actually turned any lights on!!

We have to call glEnable() again to turn on a light, thus:

```c
    glEnable(GL_LIGHT0);
```

There are a maximum of 8 lights you can use, GL_LIGHT0, GL_LIGHT1, .., GL_LIGHT7
- And the result....
3D Programming

Better.
But not great if you know what is possible with OpenGL.

3D Programming

- How can we improve this?
- First, we select a Shade Model.
- This is set using `glShadeModel();`
  The `glShadeModel()` determines how the model will be shaded.

- There are two types of shading:
  - `GL_FLAT`
  - `GL_SMOOTH`
3D Programming

glShadeModel(GL_FLAT);

3D Programming

glShadeModel(GL_SMOOTH);
3D Programming

- This improves the teapot’s surface, but it still isn’t quite right.
- **There are parts of the teapot that shouldn’t be displaying.**
  - e.g. the hidden parts.
  - We need a depth test!
- **1) The glShadeModel()** determines how the model will be shaded.
- **2) GL_DEPTH_TEST** allows OpenGL to check for the depth of parts of the image being drawn and thus will make sure that any shapes behind other shapes are not drawn. GL_NORMALIZE, while not that necessary with this teapot, helps us when drawing 3D shapes as it automatically normalizes normal vectors for us.
- **glEnable(GL_DEPTH_TEST); // “for removal of hidden surfaces”**
3D Programming

- Initialization of the drawing environment for 3D.

```c
main( int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize(SCREENWIDTH, SCREENHEIGHT);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Solid");
    glutDisplayFunc(displaySolid);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_NORMALIZE);
    glClearColor(0.1f,0.1f,0.1f,0.0f);
    glViewport(0,0, SCREENWIDTH, SCREENHEIGHT);
    glutMainLoop();
    return 1;
}
```

Dr M A BERBAR
3D Programming

- To display the teapot. This function is registered with myDisplayFunc().

```c
void displaySolid(void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1, 1, -1, 1, 0.1, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(2.3,1.3,2,0,0.25,0,0,0,1,0,0,0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(0,1,0);
    glPushMatrix();
    glTranslated(0.4,0.5, 0.4);
    glRotated(30,0,1,0);
    glutSolidTeapot(0.5);
    glPopMatrix();
}
```

Light Position

- Each light has a position. You specify it using an array of values.
- Positioning the Light   - lightPosition = x, y, z, w

The position is defined thus:

```
GLfloat lightPosition[]={2.0f,3.0f,3.0f,0.0f};
```

where the values are x, y, z and w (homogeneous coordinates - 4D).

Once you have declared the position, you will need to register it with OpenGL so that it can take affect, thus:

```
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
```
3D Programming

lightPosition[]={2.0f,6.0f,3.0f, 0.0f};

lightPosition[]={1.0f,2.0f,4.0f, 0.0f};

Light Intensity

Each light has an intensity. You specify it using an array of values.

The intensity of the light is defined thus:
GLfloat lightIntensity[] = {0.9f, 0.9f, 0.9f, 1.0f};
where the values are red, green, blue and alpha (just set alpha to 1 for now).

Once you have declared the intensity, you will need to register it with OpenGL so that it can take affect, thus:

gLightfv(GL_LIGHT0, GL_DIFFUSE, lightIntensity);

The previous lines of code can be placed as the first lines in the myDisplay function for example, although you could also put them in a myInit() function or anywhere else in the program when you want the lights turned on.

GL_DIFFUSE in the last line of code above refers to the color of the light coming directly from the light source. You can also add other gLightfv() colors for the light as

GL_AMBIENT (light that doesn’t seem to come from one direction but rather is bounced around by the environment) and GL_SPECULAR (light coming from one direction, but bouncing off the object). You can set up as many of these as you like for any color you like.
3D Programming

• Turning up the Lights
• lightIntensity = red, green, blue, alpha

```c
GLfloat lightIntensity[] = {0.7f, 0.7f, 0.7f, 1.0f};
glLightfv(GL_LIGHT0, GL_DIFFUSE, lightIntensity);
```

3D Programming

```c
lightIntensity[] = {1.0f, 1.0f, 1.0f, 1.0f};
```

```c
lightIntensity[] = {0.2f, 0.2f, 0.2f, 1.0f};
```
lightIntensity[] = {0.2f, 1.0f, 0.2f, 1.0f};

lightIntensity[] = {0.2f, 0.0f, 0.8f, 1.0f};

### Types of Light

- **Types of Light**
  - **GL_AMBIENT**
    - scattered light, difficult to determine origin
  - **GL_DIFFUSE**
    - direct light with an origin
  - **GL_SPECULAR**
    - same as shininess, light comes from one direction and bounces off the surface
/// The outline code for a simple teapot drawing program is given below:

#include <GL/glut.h>
#include <math.h>

#define W 600
#define H 600

void displaySolid(void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-W, W, -H, H, -W, W);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,10,0,0,0,0.0,1.0,0.0);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glutSolidTeapot(300);
    glFlush();
    glutSwapBuffers();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);

    glutInitWindowSize(W,H);
    glutInitWindowPosition(100, 100);

    glutCreateWindow("My Teapot");
    glutDisplayFunc(displaySolid);

    glutClearColor(0.0f,0.1f,0.0f,0.0f);
    glViewport(0,0, W, H);

    glutMainLoop();
    return 1;
}
Type this code in and save it as myTeapot.cpp. Compile and run to test. You should get this:

Step Two: Turning on the Lights

- Not much to look at is it? In order to view a 3D shape, you need to enable the OpenGL lighting. The following code is used to do this. Place it in the main function just before glutMainLoop().

```cpp
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glShadeModel(GL_SMOOTH);
glEnable(GL_DEPTH_TEST);
glEnable(GL_NORMALIZE);
```

What do each of them do? GL_LIGHTING enables OpenGL lighting. GL_LIGHT0 is light number 1. This refers to the actual light (e.g. the light bulb). OpenGL had 8 lights in all, GL_LIGHT0 .. GL_LIGHT7.
The glShadeModel() determines how the model will be shaded. If it is smooth shaded (GL_SMOOTH) it uses Gourand shading. If it is set with GL_FLAT, it will shade each polygon in the shape a uniform shade.

GL_DEPTH_TEST allows OpenGL to check for the depth of parts of the image being drawn and thus will make sure that any shapes behind other shapes are not drawn.

GL_NORMALIZE, while not that necessary with this teapot, helps us when drawing 3D shapes as it automatically normalises normal vectors for us.

Your Turn

Add the lines of code for lighting as above.

Save, compile and test your program.

Determine the effect of using GL_FLAT instead of GL_SMOOTH for the shade model.

Each light has a position and intensity. You specify each using an array of values.

The position is defined thus: GLfloat lightPosition[] = {0.0f, 0.0f, 100.0f, 0.0f}; where the values are x, y, z and w (homogeneous coordinates - 4D).

The intensity of the light is defined thus:
GLfloat lightIntensity[] = {0.9f, 0.9f, 0.9f, 1.0f}; where the values are red, green, blue and alpha (just set alpha to 1 for now).

Once you have declared the position and the intensity, you will need to register it with OpenGL so that it can take affect, thus:
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
glLightfv(GL_LIGHT0, GL_DIFFUSE, lightIntensity);

The previous four lines of code can be placed as the first lines in the myDisplay function for this example, although you could also put them in a myInit() function or anywhere else in the program when you want the lights turned on.

GL_DIFFUSE in the last line of code above refers to the color of the light coming directly from the light source. You can also add other glLightfv() colors for the light as GL_AMBIENT (light that doesn’t seem to come from one direction but rather is bounced around by the environment) and GL_SPECULAR (light coming from one direction, but bouncing off the object). You can set up as many of these as you like for any color you like.

Your Turn

Add the lighting lines to your program. What is the effect of moving the light around in the x direction?
Surface Colours

- Surface Colours
  - Light can make an object appear in a different colour, but the **actual colour of the object needs to be set differently**.
  - For this we use `glMaterialfv()`
  - The material of an object can be **ambient, diffuse and specular, just as for light**. It also has a **factor of shininess**.

```c
GLfloat mat_ambient[] = {0.5f, 0.8f, 0.6f, 1.0f};
GLfloat mat_diffuse[] = {0.6f, 0.8f, 0.6f, 1.0f};
GLfloat mat_specular[] = {1.0f, 0.8f, 1.0f, 1.0f};
GLfloat mat_shininess[] = {10.0f};
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
```

3D Programming

- Surface Colours
  - For example:
```c
GLfloat mat_ambient[] = {0.5f, 0.8f, 0.6f, 1.0f};
GLfloat mat_diffuse[] = {0.6f, 0.8f, 0.6f, 1.0f};
GLfloat mat_specular[] = {1.0f, 0.8f, 1.0f, 1.0f};
GLfloat mat_shininess[] = {10.0f};
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
```
3D Programming

• Surface Colours

The End

• Next Week
  – Creating complex 3D shapes using polygon meshes.