Computer Graphics

Unit 2 – Part 2

–By Manjula. S

Raster Methods for Geometric Transformations

- The characteristics of raster systems suggest an alternate method for performing certain two-dimensional transformations.
- Raster systems store picture information as color patterns in the frame buffer.
- Therefore, some simple object transformations can be carried out rapidly by manipulating an array of pixel values.
- Few arithmetic operations are needed, so the pixel transformations are particularly efficient.
- Functions that manipulate rectangular pixel arrays are called *raster operations* and moving a block of pixel values from one position to another is termed a *block transfer*, a ***bitblt***, or a ***pixblt***.
- Routines for performing some raster operations are usually available in a graphics package.

Raster Methods for Geometric Transformations

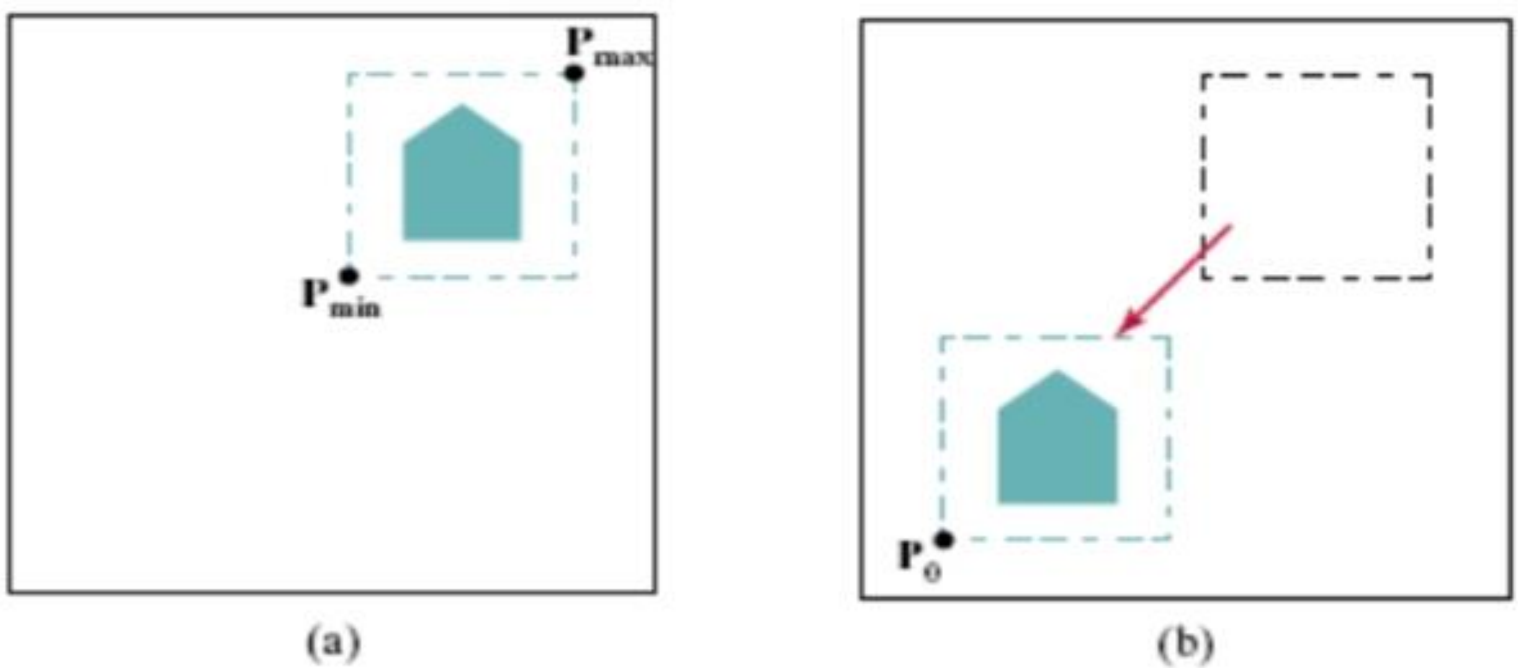


Figure to illustrate a two-dimensional translation implemented as a block transfer of a refresh-buffer area

Raster Methods for Geometric Transformations

- All bit settings in the rectangular area shown are copied as a block into another part of the frame buffer.
- We can erase the pattern at the original location by assigning the background color to all pixels within that block .

- Rotations in 90-degree increments are accomplished easily by rearranging the elements of a pixel array.
- We can rotate a two-dimensional object or pattern 90◦ counterclockwise by reversing the pixel values in each row of the array, then interchanging rows and columns.
- A 180◦ rotation is obtained by reversing the order of the elements in each row of the array, then reversing the order of the rows.
- Figure 27 demonstrates the array manipulations that can be used to rotate a pixel block by 90◦ and by 180◦.

Raster Methods for Geometric Transformations

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \qquad \begin{bmatrix} 3 & 6 & 9 & 12 \\ 2 & 5 & 8 & 11 \\ 1 & 4 & 7 & 10 \end{bmatrix} \qquad \begin{bmatrix} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

(a) (b) (c)

FIGURE 27
Rotating an array of pixel values. The original array is shown in (a), the positions of the array elements after a 90° counterclockwise rotation are shown in (b), and the positions of the array elements after a 180° rotation are shown in (c).

- For array rotations that are not multiples of 90◦, we need to do some extra processing.
- The general procedure is illustrated in Figure 28.
- Each destination pixel area is mapped onto the rotated array and the amount of overlap with the rotated pixel areas is calculated.

Raster Methods for Geometric Transformations

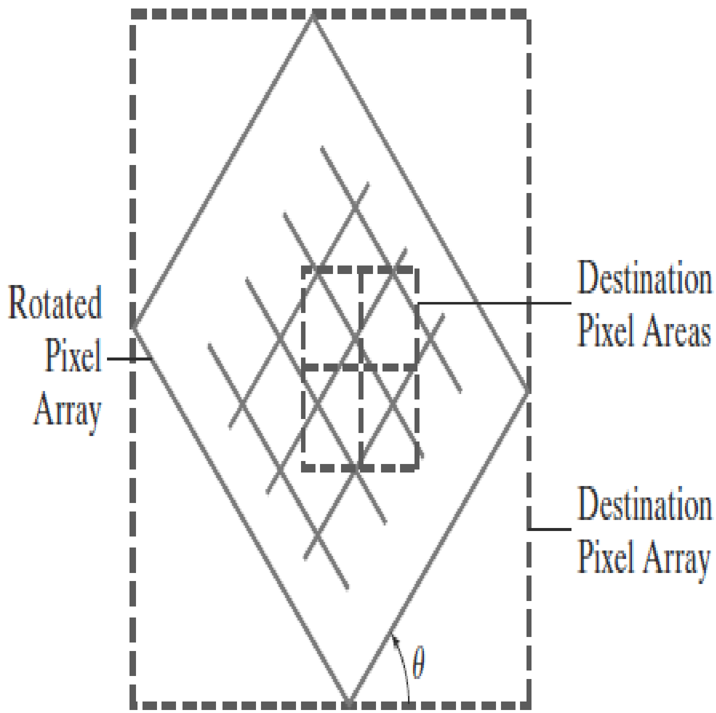


FIGURE 28

A raster rotation for a rectangular block of pixels can be accomplished by mapping the destination pixel areas onto the rotated block.

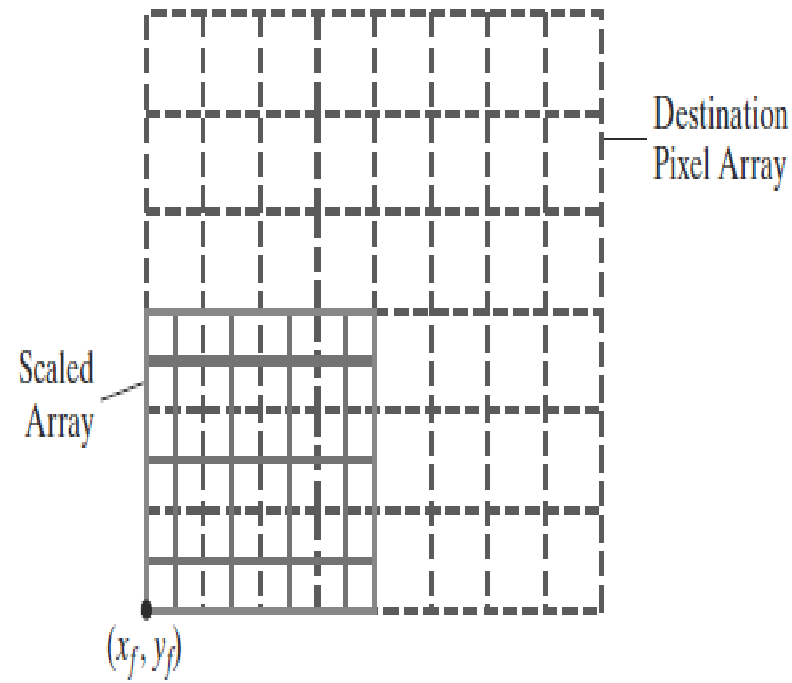


FIGURE 29

Mapping destination pixel areas onto a scaled array of pixel values. Scaling factors $s_x = s_y = 0.5$ are applied relative to fixed point (x_f, y_f).

Raster Methods for Geometric Transformations

- A color for a destination pixel can then be computed by averaging the colors of the overlapped source pixels, weighted by their percentage of area overlap.
- Alternatively, we could use an approximation method, as in antialiasing, to determine the color of the destination pixels.
- Similar methods are used to scale a block of pixels.
- Pixel areas in the original block are scaled, using specified values for *sx* and *sy*, and then mapped onto a set of destination pixels.
- The color of each destination pixel is then assigned according to its area of overlap with the scaled pixel areas (Figure 29).
- An object can be reflected using raster transformations that reverse row or column values in a pixel block, combined with translations.
- Shears are produced with shifts in the positions of array values along rows or columns.

Raster Methods for Geometric Transformations

Typical raster functions often provided in graphics packages are:

- **Copy** – Move a pixel block from one raster area to another.
- **Read** – Save a pixel block in a designated array.
- **Write** – Transfer a pixel array to a position in the frame buffer.

OpenGL Raster Transformations

- Copying pixels from one buffer area to another can be accomplished with

glCopyPixel(xmin,ymin,width,height,GL_COLOR);

– **GL_COLOR** says what is to be copied (color values)

– Copied to refresh buffer at same location.

OpenGL Raster Transformations

- To read into an array:

glReadPixels(xmin, ymin, width, height, GL_RGB, GL_UNSIGNED_BYTE, colorArray);

- To do a 90 degree rotation could rearrange rows and columns of array, then place back to refresh buffer at current raster position

glDrawPixels(width,height, GL_RGB, GL_UNSIGNED_BYTE, colorArray);

OpenGL Raster Transformations

- To scale an area use:

  glPixelZoom(sx,sy);

  where sx and sy are any nonzero floating-point values (Negative values cause reflections).

- Then use *glCopyPixels* or *glDrawPixels* to get/draw the pixels with the given scaling.

- We select the source buffer containing the original block of pixel values with *glReadBuffer*, and we designate a destination buffer with *glDrawBuffer*.

OpenGL Functions

- Transformations in OpenGL are not drawing commands. They are retained as part of the graphics state.
- When drawing commands are issued, the current transformation is applied to the points drawn.
- Transformations are cumulative.

OpenGL Geometric Transformations

- Transformation Functions enable the programmer to carryout geometric transformations such as rotation, scaling, translation.Example : glTranslatef(); glScalef(); glRotatef();
- Transformations in OpenGL
  - Translate
  - Rotate
  - Scale

OpenGL Geometric Transformations

Translation

Offset (tx, ty, tz) is applied to all subsequent coordinates. Effectively moves the origin of coordinate system.

- $x' = x + \text{tx}$, $y' = y + \text{ty}$, $z' = z + \text{tz}$
- OpenGL function is `glTranslate`
- `glTranslatef( tx, ty, tz );`

OpenGL Geometric Transformations

Rotation

Expressed as rotation through angle θ about an axis direction (x,y,z) .

- OpenGL function – `glRotatef` (θ, x,y,z). So `glRotatef(30.0, 0.0, 1.0, 0.0)` rotates by 30° about y-axis.
- Note carefully:
  - `glRotate` wants angles in degrees.
  - C math library (sin, cos etc.) wants angles in radians.
  - $degs = rads * 180/\pi$; $rads = degs * \pi / 180$
- Positive angle? Right hand rule: if the thumb points along the vector of rotation, a positive angle has the fingers curling towards the palm.

OpenGL Geometric Transformations

Rotation (cont.)

- Frequently the axis is one of the coordinate axes. Common terms:
  - rotation about *y*-axis is heading/yaw
  - rotation about *x*-axis is pitch/elevation
  - rotation about *z*-axis is roll/bank
- 3-d rotation is an extremely difficult topic! There are several different mathematical formulations. Rotations do not commute – the order that transformations are done matters.

OpenGL Geometric Transformations

Scaling

- Multiply subsequent coordinates by scale factors sx, sy, sz. (Note: these are not a point, not a vector, just 3 numbers)

  $x' = \mathrm{sx} * x\ ,\quad y' = \mathrm{sy} * y,\quad z' = \mathrm{sz} * z$
- Often sx = sy = sz for a *uniform* scaling effect. If the factors are different, the scaling is called *anamorphic*.
- OpenGL function – `glScale` For example,

  ```
  glScalef(0.5,0.5,0.5);
  ```

  would cause all objects drawn subsequently to be half as big.

OpenGL Geometric-Transformation Programming Examples

- In the following code segment, we apply each of the basic geometric transformations, one at a time, to a rectangle.

```
glMatrixMode (GL_MODELVIEW);

glColor3f (0.0, 0.0, 1.0);
glRecti (50, 100, 200, 150);       // Display blue rectangle.

glColor3f (1.0, 0.0, 0.0);
glTranslatef (-200.0, -50.0, 0.0); // Set translation parameters.
glRecti (50, 100, 200, 150);       // Display red, translated rectangle.

glLoadIdentity ( );                // Reset current matrix to identity.
glRotatef (90.0, 0.0, 0.0, 1.0);   // Set 90-deg. rotation about z axis.
glRecti (50, 100, 200, 150);       // Display red, rotated rectangle.

glLoadIdentity ( );                // Reset current matrix to identity.
glScalef (-0.5, 1.0, 1.0);         // Set scale-reflection parameters.
glRecti (50, 100, 200, 150);       // Display red, transformed rectangle.
```

Translating a Rectangle

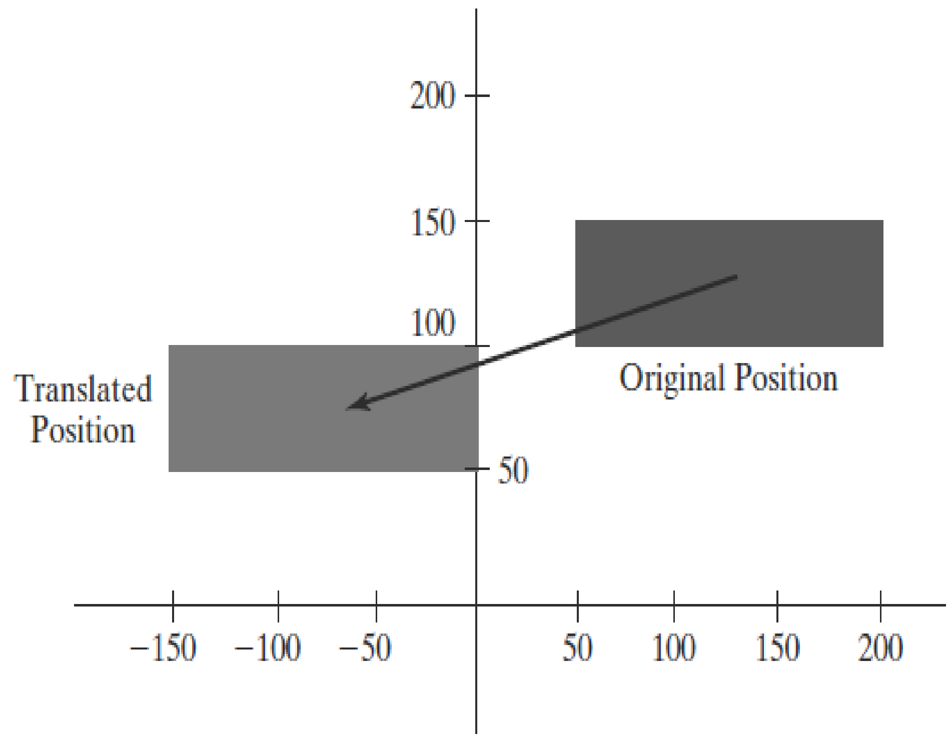


FIGURE 34
Translating a rectangle using the OpenGL function `glTranslatef (−200.0, −50.0, 0.0)`.

Rotating a Rectangle

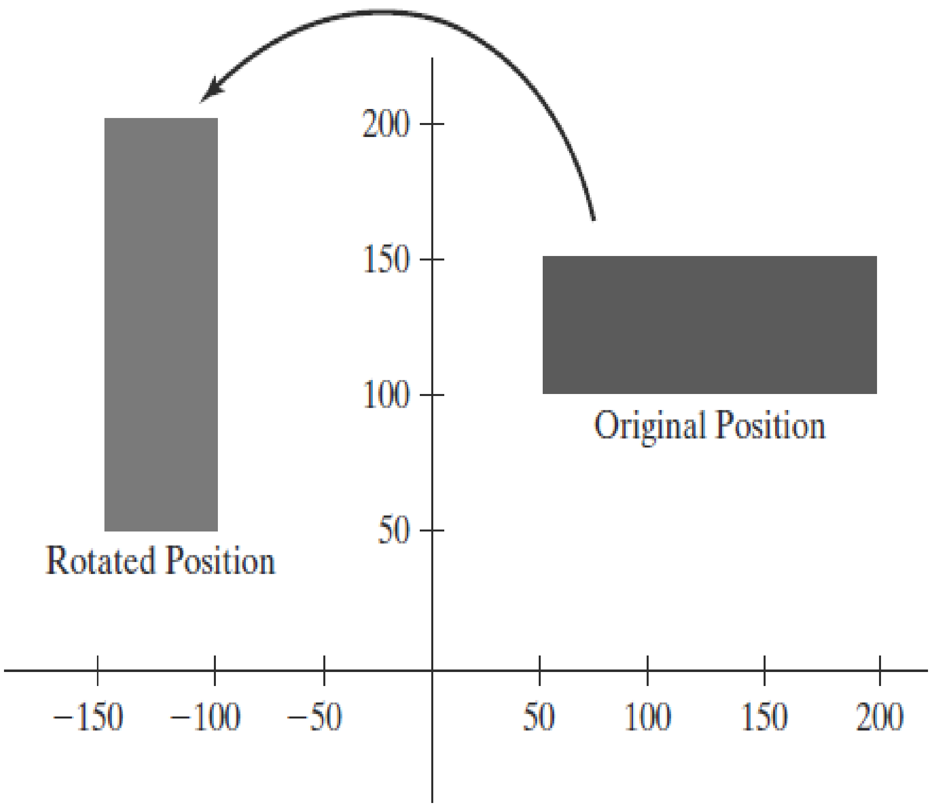


FIGURE 35
Rotating a rectangle about the *z* axis using the OpenGL function `glRotatef (90.0, 0.0, 0.0, 1.0)`.

Scaling and Reflecting a Rectangle

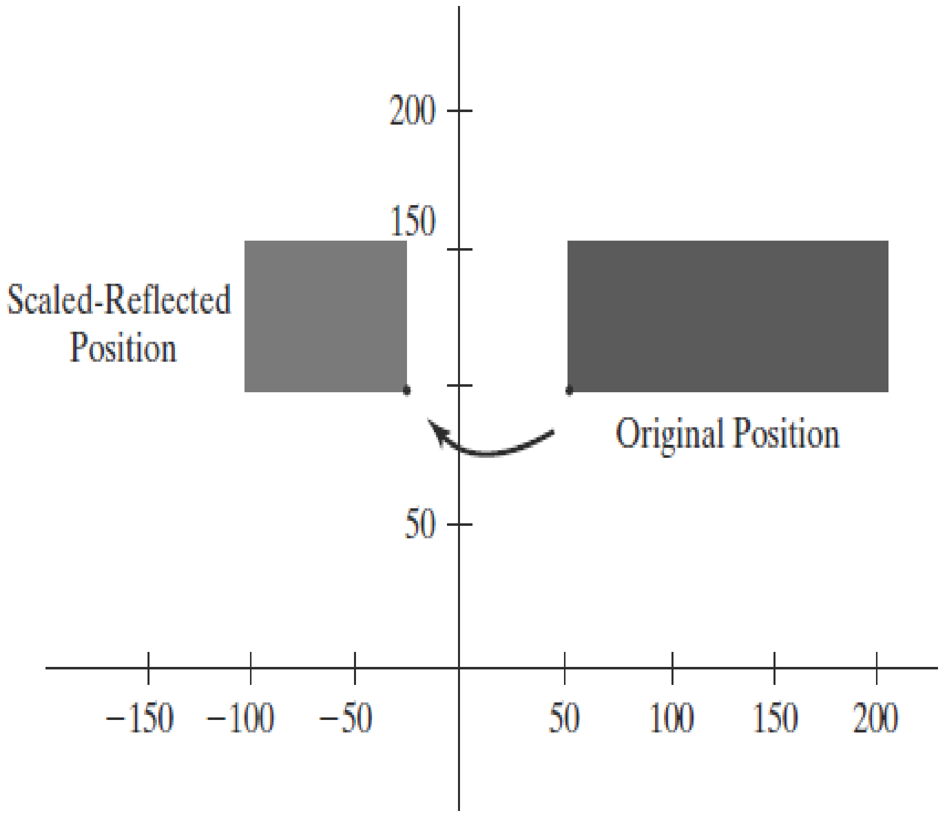


FIGURE 36
Scaling and reflecting a rectangle using the OpenGL function `glScalef (−0.5, 1.0, 1.0)`.

Order of transformations

- Transformations are cumulative and the order matters:
  - The sequence
    1. Scale 2, 2, 2
    2. Translate by (10, 0, 0)

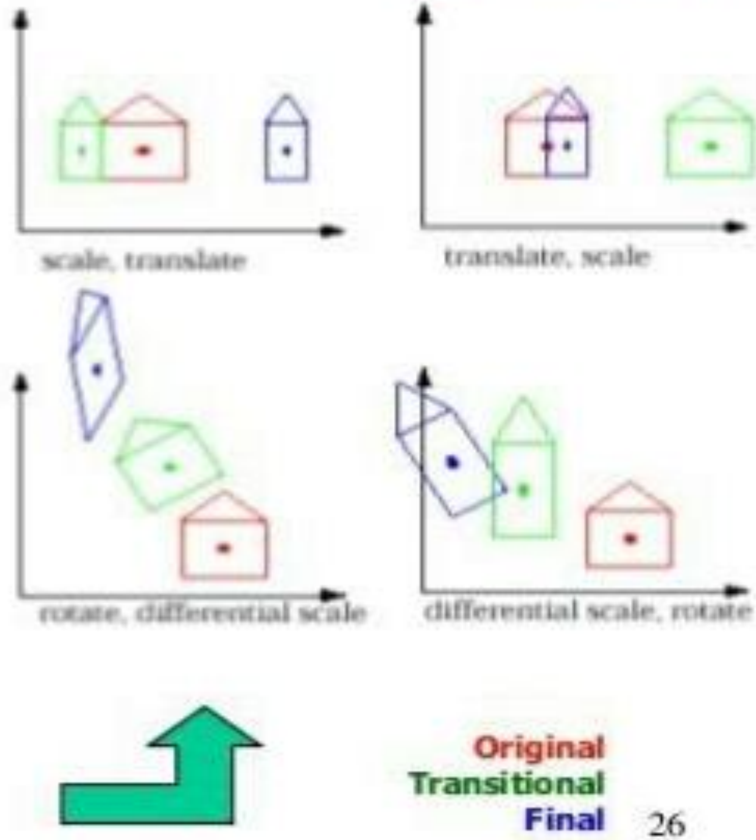
    will scale subsequent objects by factor of 2 about an origin that is 20 along the x-axis
  - The sequence
    1. Rotate 90.0 deg about (0, 1, 0)
    2. Translate by (10, 0, 0)

    will set an origin 10 along the –ve z-axis
- For each object, the usual sequence is:
  1. Translate (move the origin to the right location)
  2. Rotate (orient the coordinate axes right)
  3. Scale (get the object to the right size)

Commutative of Transformation Matrices

- In general matrix multiplication is not commutative
- For the following special cases commutativity holds i.e. $M_1.M_2 = M_2.M_1$

M_1	M_2
Translate	Translate
Scale	Scale
Rotate	Rotate
Uniform Scale	Rotate



- **Some non-commutative Compositions:**
  - Non-uniform scale, Rotate
  - Translate, Scale
  - Rotate, Translate

-By Manjula. S

Summary of OpenGL Geometric Transformation Functions

Function	Description
glTranslate*	Specifies translation parameters.
glRotate*	Specifies parameters for rotation about any axis through the origin.
glScale*	Specifies scaling parameters with respect to coordinate origin.
glMatrixMode	Specifies current matrix for geometric-viewing transformations, projection transformations, texture transformations, or color transformations.
glLoadIdentity	Sets current matrix to identity.
glLoadMatrix* (elems);	Sets elements of current matrix.
glMultMatrix* (elems);	Postmultiplies the current matrix by the specified matrix.
glPixelZoom	Specifies two-dimensional scaling parameters for raster operations.

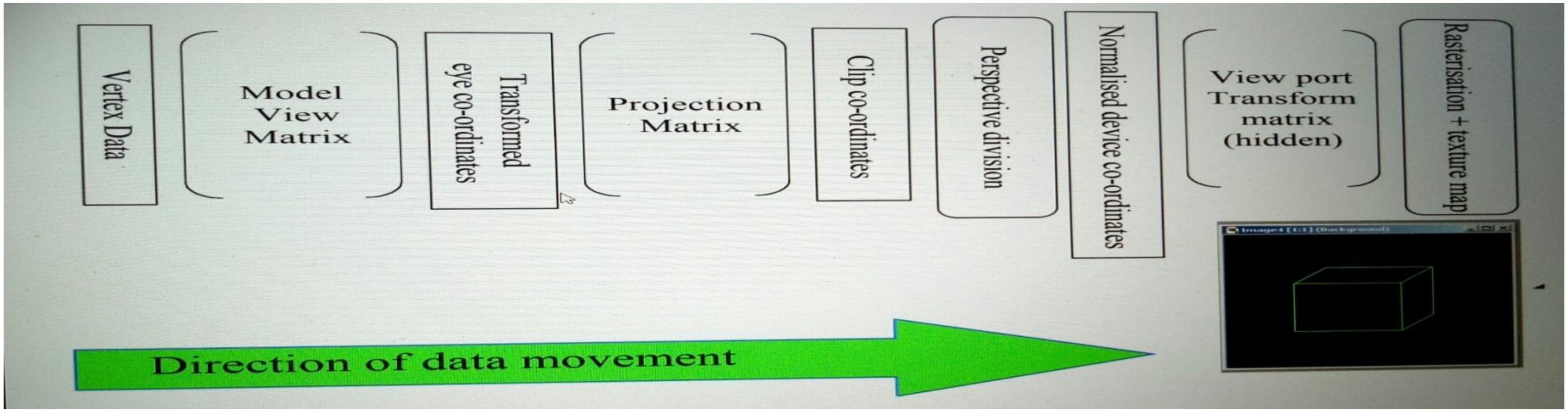
OpenGL Matrix Operations

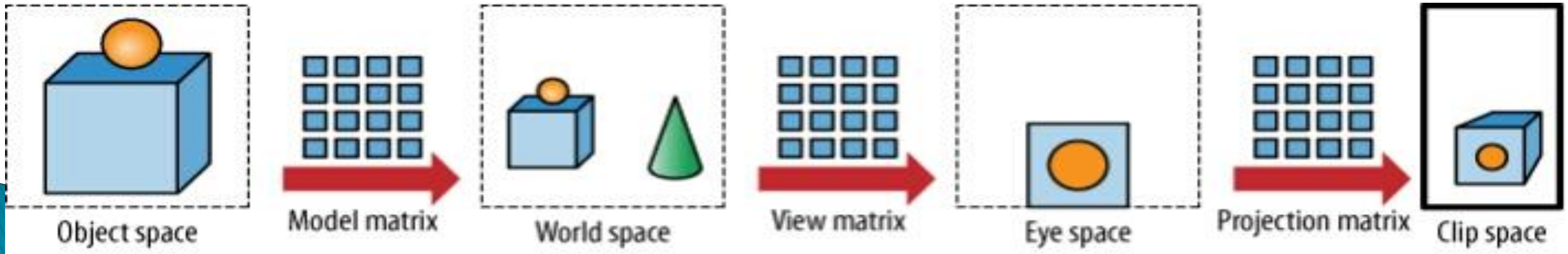
OpenGL Matrix Operations are:

- glMatrixMode(GL_MODELVIEW);
- glMatrixMode(GL_PROJECTION);
- glPushMatrix();
- glPopMatrix();
- glLoadIdentity();

OpenGL Matrix Operations

– **Modelview matrix**: combines modelling and viewing transforms
– **Projection matrix**: projects 3-D viewing coordinates onto image plane





-By Manjula. S

Matrices and Graphics State

- Each of the transformations above (Model View Matrix, Projection Matrix etc.) is maintained by OpenGL as part of the graphics state. (Current Transformation Matrix CTM)
- `glLoadIdentity` sets the CTM to the identity matrix, for a "fresh start".
- When `glRotate` or similar command is issued, the appropriate transformation matrix is updated.
- Note carefully that the rotation matrix doesn't overwrite the old CTM. It updates CTM by matrix multiplication.
- In fact the CTM is so important that OpenGL can keep several of them in a stack. By popping the stack, you can recover an old and possibly still-useful CTM.

OpenGL matrix operations

- glMatrixMode (GL_MODELVIEW)
  - Designates the matrix that is to be used for projection transformation (current matrix)
- glLoadIdentity ( )
  - Assigns the identity matrix to the current matrix
- Note: OpenGL stores matrices in column-major order
  - Reference to a matrix element m_{jk} in OpenGL is a reference to the element in column j and row k
  - glMultMatrix* ( ) post-multiplies the current matrix
- In OpenGL, the transformation specified last is the one applied first

```
glMatrixMode (GL_MODELVIEW):

glLoadIdentity ( ):        // Set current matrix to the identity.
glMultMatrixf (elemsM2):   // Postmultiply identity with matrix M2.
glMultMatrixf (elemsM1):   // Postmultiply M2 with matrix M1.
```

$$\mathbf{M} = \mathbf{M}_2 \cdot \mathbf{M}_1$$

Push and Pop

- `glMatrixMode(GL_MODELVIEW)`
- `glMatrixMode(GL_PROJECTION)`
- `glMatrixMode(GL_TEXTURE)`

- **`glPushMatrix`**`();`
  - Save the state.
  - Push a copy of the CTM onto the stack.
  - The CTM itself is unchanged.
- **`glPopMatrix`**`();`
  - Restore the state, as it was at the last Push.
  - Overwrite the CTM with the matrix at the top of the stack.
- **`glLoadIdentity`**`();`
  - Overwrite the CTM with the identity matrix.

Push and Pop

- Program module to draw a tea pot

```
glPushMatrix();              //Push operation
glTranslated(0,30,0);        //Translation
glRotatef(35,1,0.5,0);       // Rotation
glScaled(1,8,1);             // Scale
glutSolidTeapot(10);
glPopMatrix();               //Pop operation
```

Program to perform OpenGL geometric transformation function

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <GL/glut.h>

// window size
#define maxWD 640
#define maxHT 480

// rotation speed
#define thetaSpeed 0.05

// this creates delay between two actions
void delay(unsigned int mseconds)
{
    clock_t goal = mseconds + clock();
    while (goal > clock());
}

```

-By Manjula. S

```
// this is a basic init for the glut window
void myInit(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, maxWD, 0.0, maxHT);
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

// this function just draws a point
void drawPoint(int x, int y)
{
    glPointSize(7.0);
    glColor3f(0.0f, 0.0f, 1.0f);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}
```

Program Module for Rotation Transformation

```
void rotateAroundPt(int px, int py, int cx, int cy)
{   float theta = 0.0;
    while (1) {
        glClear(GL_COLOR_BUFFER_BIT);
        int xf, yf;

        // update theta anticlockwise rotation
        theta = theta + thetaSpeed;

        // check overflow
        if (theta >= (2.0 * 3.14159))
            theta = theta - (2.0 * 3.14159);

        // actual calculations..
        xf = cx + (int)((float)(px - cx) * cos(theta))
             - ((float)(py - cy) * sin(theta));
        yf = cy + (int)((float)(px - cx) * sin(theta))
             + ((float)(py - cy) * cos(theta));

        // drawing the centre point
        drawPoint(cx, cy);

        // drawing the rotating point
        drawPoint(xf, yf);
        glFlush();
        // creating a delay
        // so that the point can be noticed
        delay(10);  } }
```

-By Manjula. S

Program Module for Translation Transformation

```
// this function will translate the point
void translatePoint(int px, int py, int tx, int ty)
{
    int fx = px, fy = py;
    while (1) {
        glClear(GL_COLOR_BUFFER_BIT);

        // update
        px = px + tx;
        py = py + ty;

        // check overflow to keep point in screen
        if (px > maxWD || px < 0 || py > maxHT || py < 0) {
            px = fx;
            py = fy;
        }

        drawPoint(px, py); // drawing the point

        glFlush();
        // creating a delay
        // so that the point can be noticed
        delay(10);
    }
}
```

Program Module for Scaling Transformation

```
// this function draws
void scalePoint(int px, int py, int sx, int sy)
{
    int fx, fy;
    while (1) {
        glClear(GL_COLOR_BUFFER_BIT);

        // update
        fx = px * sx;
        fy = py * sy;

        drawPoint(fx, fy); // drawing the point

        glFlush();
        // creating a delay
        // so that the point can be noticed
        delay(500);

        glClear(GL_COLOR_BUFFER_BIT);

        // update
        fx = px;
        fy = py;

        // drawing the point
        drawPoint(fx, fy);
        glFlush();
        // creating a delay
        // so that the point can be noticed
        delay(500);    } }
```

-By Manjula. S

```c
// Actual display function
void myDisplay(void)
{
   int opt;
   printf("\nEnter\n\t<1> for translation"
         "\n\t<2> for rotation"
         "\n\t<3> for scaling\n\t:");
   scanf("%d", &opt);
   printf("\nGo to the window...");
   switch (opt) {
   case 1:
      translatePoint(100, 200, 1, 5);
      break;
   case 2:
      rotateAroundPt(200, 200, maxWD / 2, maxHT / 2);
      // point will circle around
      // the centre of the window
      break;
   case 3:
      scalePoint(10, 20, 2, 3);
      break;
   }
}

void main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(maxWD, maxHT);
   glutInitWindowPosition(100, 150);
   glutCreateWindow("Transforming point");
   glutDisplayFunc(myDisplay);
   myInit();
   glutMainLoop(); }
```

Thank You

-By Manjula. S