

Module 1: Computer Graphics and OpenGL

Computer Graphics:

- 1.1 Basics of computer graphics
- 1.2 Application of Computer Graphics

OpenGL:

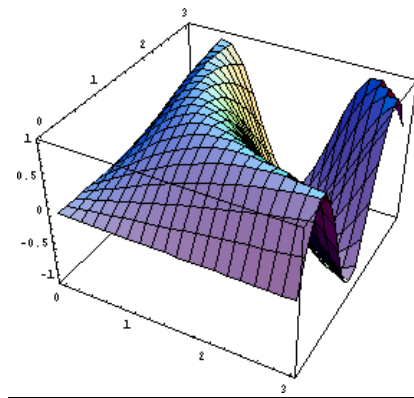
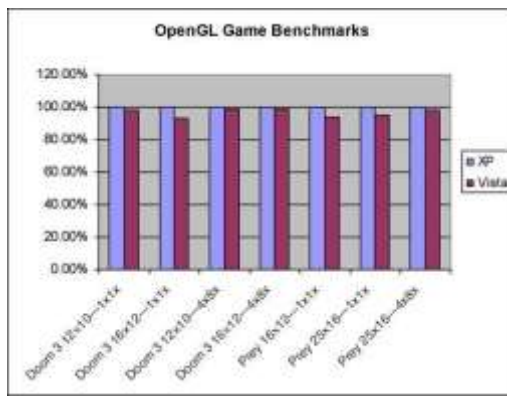
- 1.3 Introduction to OpenGL
- 1.4 Coordinate reference frames
- 1.5 Specifying two-dimensional world coordinate reference frames in OpenGL
- 1.6 OpenGL point functions
- 1.7 OpenGL line functions
- 1.8 point attributes
- 1.9 Line attributes
- 1.10 Curve attributes
- 1.11 OpenGL point attribute functions
- 1.12 OpenGL line attribute functions
- 1.13 Line drawing algorithms(Bresenham's)

Basics of Computer Graphics

Computer graphics is an art of drawing pictures, lines, charts, etc. using computers with the help of programming. Computer graphics image is made up of number of pixels. Pixel is the smallest addressable graphical unit represented on the computer screen.

Applications of Computer Graphics

a. Graphs and Charts



- ✓ An early application for computer graphics is the display of simple data graphs usually plotted on a character printer. Data plotting is still one of the most common graphics application.
- ✓ Graphs & charts are commonly used to summarize functional, statistical, mathematical, engineering and economic data for research reports, managerial summaries and other types of publications.
- ✓ Typically examples of data plots are line graphs, bar charts, pie charts, surface graphs, contour plots and other displays showing relationships between multiple parameters in two dimensions, three dimensions, or higher-dimensional spaces

b. Computer-Aided Design



- ✓ A major use of computer graphics is in design processes particularly for engineering and

architectural systems.

- ✓ CAD, computer-aided design or CADD, computer-aided drafting and design methods are now routinely used in the automobiles, aircraft, spacecraft, computers, home appliances.
- ✓ Circuits and networks for communications, water supply or other utilities are constructed with repeated placement of a few geographical shapes.
- ✓ Animations are often used in CAD applications. Real-time, computer animations using wire-frame shapes are useful for quickly testing the performance of a vehicle or system.

c. Virtual-Reality Environments



- ✓ Animations in virtual-reality environments are often used to train heavy-equipment operators or to analyze the effectiveness of various cabin configurations and control placements.
- ✓ With virtual-reality systems, designers and others can move about and interact with objects in various ways. Architectural designs can be examined by taking simulated “walk” through the rooms or around the outsides of buildings to better appreciate the overall effect of a particular design.
- ✓ With a special glove, we can even “grasp” objects in a scene and turn them over or move them from one place to another.

d. Data Visualizations

- ✓ Producing graphical representations for scientific, engineering and medical data sets and processes is another fairly new application of computer graphics, which is generally referred to as scientific visualization. And the term business visualization is used in connection with data sets related to commerce, industry and other nonscientific areas.

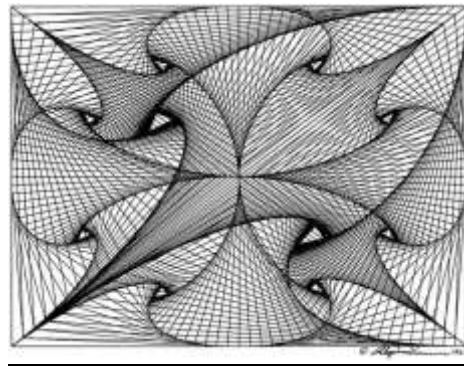


- ✓ There are many different kinds of data sets and effective visualization schemes depend on the characteristics of the data. A collection of data can contain scalar values, vectors or higher-order tensors.

e. Education and Training



- ✓ Computer generated models of physical, financial, political, social, economic & other systems are often used as educational aids.
- ✓ Models of physical processes physiological functions, equipment, such as the color coded diagram as shown in the figure, can help trainees to understand the operation of a system.
- ✓ For some training applications, special hardware systems are designed. Examples of such specialized systems are the simulators for practice sessions ,aircraft pilots, air traffic-control personnel.
- ✓ Some simulators have no video screens, for eg: flight simulator with only a control panel for instrument flying

f. Computer Art

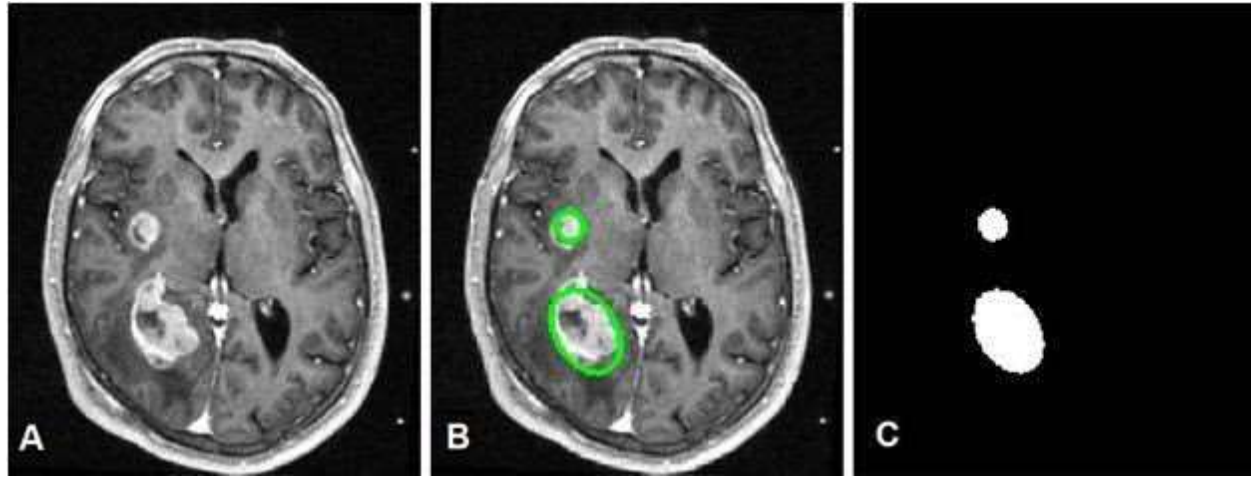
- ✓ The picture is usually painted electronically on a graphics tablet using a stylus, which can simulate different brush strokes, brush widths and colors.
- ✓ Fine artists use a variety of other computer technologies to produce images. To create pictures the artist uses a combination of 3D modeling packages, texture mapping, drawing programs and CAD software etc.
- ✓ Commercial art also uses these “painting” techniques for generating logos & other designs, page layouts combining text & graphics, TV advertising spots & other applications.
- ✓ A common graphics method employed in many television commercials is morphing, where one object is transformed into another.

g. Entertainment

- ✓ Television production, motion pictures, and music videos routinely a computer graphics methods.
- ✓ Sometimes graphics images are combined a live actors and scenes and sometimes the films are completely generated a computer rendering and animation techniques.

- ✓ Some television programs also use animation techniques to combine computer generated figures of people, animals, or cartoon characters with the actor in a scene or to transform an actor's face into another shape.

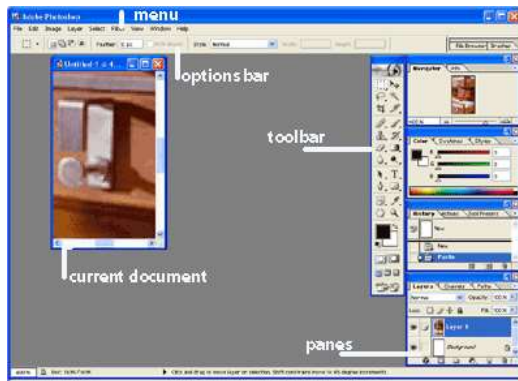
h. Image Processing



- ✓ The modification or interpretation of existing pictures, such as photographs and TV scans is called image processing.
- ✓ Methods used in computer graphics and image processing overlap, the two areas are concerned with fundamentally different operations.
- ✓ Image processing methods are used to improve picture quality, analyze images, or recognize visual patterns for robotics applications.
- ✓ Image processing methods are often used in computer graphics, and computer graphics methods are frequently applied in image processing.
- ✓ Medical applications also make extensive use of image processing techniques for picture enhancements in tomography and in simulations and surgical operations.
- ✓ It is also used in computed X-ray tomography(CT), position emission tomography(PET),and computed axial tomography(CAT).

i. Graphical User Interfaces

- ✓ It is common now for applications software to provide graphical user interface (GUI).
- ✓ A major component of graphical interface is a window manager that allows a user to display multiple, rectangular screen areas called display windows.



- ✓ Each screen display area can contain a different process, showing graphical or non-graphical information, and various methods can be used to activate a display window.
- ✓ Using an interactive pointing device, such as mouse, we can active a display window on some systems by positioning the screen cursor within the window display area and pressing the left mouse button.

Introduction To OpenGL

- ✓ OpenGL basic(core) library :-A basic library of functions is provided in OpenGL for specifying graphics primitives, attributes, geometric transformations, viewing transformations, and many other operations.

Basic OpenGL Syntax

- ➔ Function names in the OpenGL basic library (also called the OpenGL core library) are prefixed with gl. The component word first letter is capitalized.
- ➔ For eg:- glBegin, glClear, glCopyPixels, glPolygonMode
- ➔ Symbolic constants that are used with certain functions as parameters are all in capital letters, preceded by “GL”, and component are separated by underscore.
- ➔ For eg:- GL_2D, GL_RGB, GL_CCW, GL_POLYGON, GL_AMBIENT_AND_DIFFUSE.

- ➔ The OpenGL functions also expect specific data types. For example, an OpenGL function parameter might expect a value that is specified as a 32-bit integer. But the size of an integer specification can be different on different machines.
- ➔ To indicate a specific data type, OpenGL uses special built-in, data-type names, such as GLbyte, GLshort, GLint, GLfloat, GLdouble, GLboolean

Related Libraries

- ➔ In addition to OpenGL basic(core) library(prefixed with gl), there are a number of associated libraries for handling special operations:-

1) OpenGL Utility(GLU):- Prefixed with “glu”. It provides routines for setting up viewing and projection matrices, describing complex objects with line and polygon approximations, displaying quadrics and B-splines using linear approximations, processing the surface-rendering operations, and other complex tasks.

-Every OpenGL implementation includes the GLU library

2) Open Inventor:- provides routines and predefined object shapes for interactive three-dimensional applications which are written in C++.

3) Window-system libraries:- To create graphics we need display window. We cannot create the display window directly with the basic OpenGL functions since it contains only device-independent graphics functions, and window-management operations are device-dependent. However, there are several window-system libraries that supports OpenGL functions for a variety of machines.

Eg:- Apple GL(AGL), Windows-to-OpenGL(WGL), Presentation Manager to OpenGL(PGL), GLX.

4) OpenGL Utility Toolkit(GLUT):- provides a library of functions which acts as interface for interacting with any device specific screen-windowing system, thus making our program device-independent. The GLUT library functions are prefixed with “glut”.

Header Files

- ✓ In all graphics programs, we will need to include the header file for the OpenGL core library.

- ✓ In windows to include OpenGL core libraries and GLU we can use the following header files:-

#include <windows.h> //precedes other header files for including Microsoft windows ver of OpenGL libraries

#include<GL/gl.h>

#include <GL/glu.h>

- ✓ The above lines can be replaced by using GLUT header file which ensures gl.h and glu.h are included correctly,
- ✓ #include <GL/glut.h> //GL in windows
- ✓ In Apple OS X systems, the header file inclusion statement will be,
- ✓ #include <GLUT/glut.h>

Display-Window Management Using GLUT

- ✓ We can consider a simplified example, minimal number of operations for displaying a picture.

Step 1: initialization of GLUT

- ✱ We are using the OpenGL Utility Toolkit, our first step is to initialize GLUT.
- ✱ This initialization function could also process any command line arguments, but we will not need to use these parameters for our first example programs.
- ✱ We perform the GLUT initialization with the statement

glutInit (&argc, argv);

Step 2: title

- ✱ We can state that a display window is to be created on the screen with a given caption for the title bar. This is accomplished with the function

glutCreateWindow ("An Example OpenGL Program");

- ✱ where the single argument for this function can be any character string that we want to use for the display-window title.

Step 3: Specification of the display window

- ✱ Then we need to specify what the display window is to contain.
- ✱ For this, we create a picture using OpenGL functions and pass the picture definition to the GLUT routine glutDisplayFunc, which assigns our picture to the display window.

✳ Example: suppose we have the OpenGL code for describing a line segment in a procedure called `lineSegment`.

✳ Then the following function call passes the line-segment description to the display window:

`glutDisplayFunc (lineSegment);`

Step 4: one more GLUT function

✳ But the display window is not yet on the screen.

✳ We need one more GLUT function to complete the window-processing operations.

✳ After execution of the following statement, all display windows that we have created, including their graphic content, are now activated:

`glutMainLoop ();`

✳ This function must be the last one in our program. It displays the initial graphics and puts the program into an infinite loop that checks for input from devices such as a mouse or keyboard.

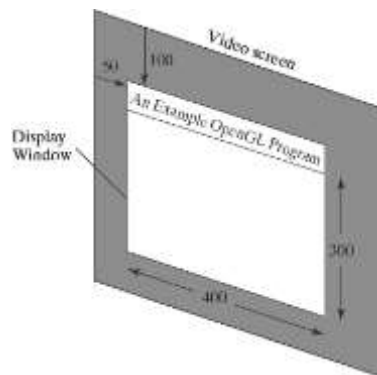
Step 5: these parameters using additional GLUT functions

✳ Although the display window that we created will be in some default location and size, we can set these parameters using additional GLUT functions.

GLUT Function 1:

➔ We use the `glutInitWindowPosition` function to give an initial location for the upper left corner of the display window.

➔ This position is specified in integer screen coordinates, whose origin is at the upper-left corner of the screen.



GLUT Function 2:

After the display window is on the screen, we can reposition and resize it.

GLUT Function 3:

- ➔ We can also set a number of other options for the display window, such as buffering and a choice of color modes, with the `glutInitDisplayMode` function.
- ➔ Arguments for this routine are assigned symbolic GLUT constants.
- ➔ Example: the following command specifies that a single refresh buffer is to be used for the display window and that we want to use the color mode which uses red, green, and blue (RGB) components to select color values:

`glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);`

- ➔ The values of the constants passed to this function are combined using a logical or operation.
- ➔ Actually, single buffering and RGB color mode are the default options.
- ➔ But we will use the function now as a reminder that these are the options that are set for our display.
- ➔ Later, we discuss color modes in more detail, as well as other display options, such as double buffering for animation applications and selecting parameters for viewing three-dimensional scenes.

A Complete OpenGL Program

- ➔ There are still a few more tasks to perform before we have all the parts that we need for a complete program.

Step 1: to set background color

- ➔ For the display window, we can choose a background color.
- ➔ Using RGB color values, we set the background color for the display window to be white, with the OpenGL function:

`glClearColor (1.0, 1.0, 1.0, 0.0);`
- ➔ The first three arguments in this function set the red, green, and blue component colors to the value 1.0, giving us a white background color for the display window.
- ➔ If, instead of 1.0, we set each of the component colors to 0.0, we would get a black background.

- ➔ The fourth parameter in the `glClearColor` function is called the alpha value for the specified color. One use for the alpha value is as a “blending” parameter
- ➔ When we activate the OpenGL blending operations, alpha values can be used to determine the resulting color for two overlapping objects.
- ➔ An alpha value of 0.0 indicates a totally transparent object, and an alpha value of 1.0 indicates an opaque object.
- ➔ For now, we will simply set alpha to 0.0.
- ➔ Although the `glClearColor` command assigns a color to the display window, it does not put the display window on the screen.

Step 2: to set window color

- ➔ To get the assigned window color displayed, we need to invoke the following OpenGL function:

`glClear (GL_COLOR_BUFFER_BIT);`

- ➔ The argument `GL_COLOR_BUFFER_BIT` is an OpenGL symbolic constant specifying that it is the bit values in the color buffer (refresh buffer) that are to be set to the values indicated in the `glClearColor` function. (OpenGL has several different kinds of buffers that can be manipulated.

Step 3: to set color to object

- ➔ In addition to setting the background color for the display window, we can choose a variety of color schemes for the objects we want to display in a scene.
- ➔ For our initial programming example, we will simply set the object color to be a dark green

`glColor3f (0.0, 0.4, 0.2);`

- ➔ The suffix `3f` on the `glColor` function indicates that we are specifying the three RGB color components using floating-point (f) values.
- ➔ This function requires that the values be in the range from 0.0 to 1.0, and we have set red = 0.0, green = 0.4, and blue = 0.2.

Example program

- ➔ For our first program, we simply display a two-dimensional line segment.
- ➔ To do this, we need to tell OpenGL how we want to “project” our picture onto the display window because generating a two-dimensional picture is treated by OpenGL as a special case of three-dimensional viewing.
- ➔ So, although we only want to produce a very simple two-dimensional line, OpenGL processes our picture through the full three-dimensional viewing operations.
- ➔ We can set the projection type (mode) and other viewing parameters that we need with the following two functions:

glMatrixMode (GL_PROJECTION);

gluOrtho2D (0.0, 200.0, 0.0, 150.0);

- ➔ This specifies that an orthogonal projection is to be used to map the contents of a twodimensional rectangular area of world coordinates to the screen, and that the x-coordinate values within this rectangle range from 0.0 to 200.0 with y-coordinate values ranging from 0.0 to 150.0.
- ➔ Whatever objects we define within this world-coordinate rectangle will be shown within the display window.
- ➔ Anything outside this coordinate range will not be displayed.
- ➔ Therefore, the GLU function gluOrtho2D defines the coordinate reference frame within the display window to be (0.0, 0.0) at the lower-left corner of the display window and (200.0, 150.0) at the upper-right window corner.
- ➔ For now, we will use a world-coordinate rectangle with the same aspect ratio as the display window, so that there is no distortion of our picture.
- ➔ Finally, we need to call the appropriate OpenGL routines to create our line segment.
- ➔ The following code defines a two-dimensional, straight-line segment with integer,
- ➔ Cartesian endpoint coordinates (180, 15) and (10, 145).

glBegin (GL_LINES);

glVertex2i (180, 15);

glVertex2i (10, 145);

glEnd ();

- ➔ Now we are ready to put all the pieces together:

The following OpenGL program is organized into three functions.

- ➔ **init:** We place all initializations and related one-time parameter settings in function `init`.
- ➔ **lineSegment:** Our geometric description of the “picture” that we want to display is in function `lineSegment`, which is the function that will be referenced by the GLUT function `glutDisplayFunc`.
- ➔ **main function** `main` function contains the GLUT functions for setting up the display window and getting our line segment onto the screen.
- ➔ **glFlush:** This is simply a routine to force execution of our OpenGL functions, which are stored by computer systems in buffers in different locations, depending on how OpenGL is implemented.
- ➔ The procedure `lineSegment` that we set up to describe our picture is referred to as a display callback function.
- ➔ And this procedure is described as being “registered” by `glutDisplayFunc` as the routine to invoke whenever the display window might need to be redisplayed.

Example: if the display window is moved.

Following program to display window and line segment generated by this program:

```
#include <GL/glut.h> // (or others, depending on the system in use)

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0);    // Set display-window color to white.
    glMatrixMode (GL_PROJECTION);    // Set projection parameters.
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void lineSegment (void)
{
    glClear (GL_COLOR_BUFFER_BIT);    // Clear display window.
    glColor3f (0.0, 0.4, 0.2);        // Set line segment color to green.
    glBegin (GL_LINES);
    glVertex2i (180, 15);              // Specify line-segment geometry.
    glVertex2i (10, 145);
    glEnd ( );
}
```



```
        glFlush ( ); // Process all OpenGL routines as quickly as possible.
    }
    void main (int argc, char** argv)
    {
        glutInit (&argc, argv);                // Initialize GLUT.
        glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB); // Set display mode.
        glutInitWindowPosition (50, 100); // Set top-left display-window position.
        glutInitWindowSize (400, 300); // Set display-window width and height.
        glutCreateWindow ("An Example OpenGL Program"); // Create display window.
        init ( ); // Execute initialization procedure.
        glutDisplayFunc (lineSegment); // Send graphics to display window.
        glutMainLoop ( ); // Display everything and wait.
    }
```

Coordinate Reference Frames

To describe a picture, we first decide upon

- ✱ A convenient Cartesian coordinate system, called the world-coordinate reference frame, which could be either 2D or 3D.
- ✱ We then describe the objects in our picture by giving their geometric specifications in terms of positions in world coordinates.
- ✱ Example: We define a straight-line segment with two endpoint positions, and a polygon is specified with a set of positions for its vertices.
- ✱ These coordinate positions are stored in the scene description along with other info about the objects, such as their color and their coordinate extents
- ✱ Co-ordinate extents :Co-ordinate extents are the minimum and maximum x, y, and z values for each object.
- ✱ A set of coordinate extents is also described as a bounding box for an object.
- ✱ Ex:For a 2D figure, the coordinate extents are sometimes called its bounding rectangle.
- ✱ Objects are then displayed by passing the scene description to the viewing routines which identify visible surfaces and map the objects to the frame buffer positions and then on the video monitor.

- ✱ The scan-conversion algorithm stores info about the scene, such as color values, at the appropriate locations in the frame buffer, and then the scene is displayed on the output device.

Screen co-ordinates:

- ✓ Locations on a video monitor are referenced in integer screen coordinates, which correspond to the integer pixel positions in the frame buffer.
- ✓ Scan-line algorithms for the graphics primitives use the coordinate descriptions to determine the locations of pixels
- ✓ Example: given the endpoint coordinates for a line segment, a display algorithm must calculate the positions for those pixels that lie along the line path between the endpoints.
- ✓ Since a pixel position occupies a finite area of the screen, the finite size of a pixel must be taken into account by the implementation algorithms.
- ✓ For the present, we assume that each integer screen position references the centre of a pixel area.
- ✓ Once pixel positions have been identified the color values must be stored in the frame buffer

Assume we have available a low-level procedure of the form

i)setPixel (x, y);

- stores the current color setting into the frame buffer at integer position(x, y), relative to the position of the screen-coordinate origin

ii)getPixel (x, y, color);

- Retrieves the current frame-buffer setting for a pixel location;
- Parameter color receives an integer value corresponding to the combined RGB bit codes stored for the specified pixel at position (x,y).
- Additional screen-coordinate information is needed for 3D scenes.
- For a two-dimensional scene, all depth values are 0.

Absolute and Relative Coordinate Specifications

Absolute coordinate:

- So far, the coordinate references that we have discussed are stated as absolute coordinate values.
- This means that the values specified are the actual positions within the coordinate system in use.

Relative coordinates:

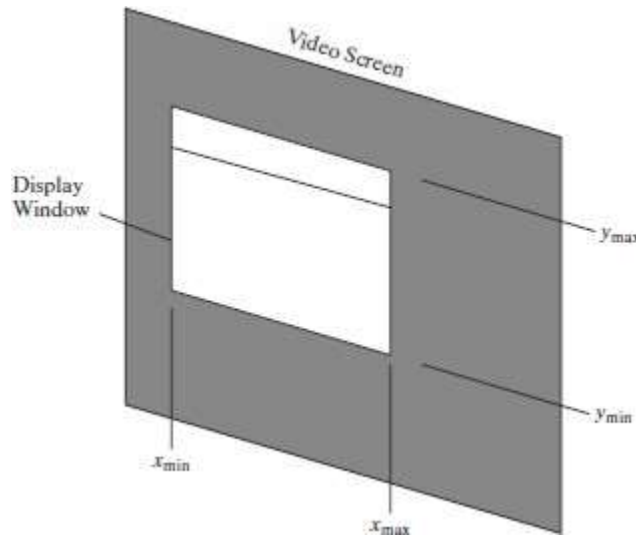
- However, some graphics packages also allow positions to be specified using relative coordinates.
- This method is useful for various graphics applications, such as producing drawings with pen plotters, artist's drawing and painting systems, and graphics packages for publishing and printing applications.
- Taking this approach, we can specify a coordinate position as an offset from the last position that was referenced (called the current position).

Specifying a Two-Dimensional World-Coordinate Reference Frame in OpenGL

- The `gluOrtho2D` command is a function we can use to set up any 2D Cartesian reference frames.
- The arguments for this function are the four values defining the x and y coordinate limits for the picture we want to display.
- Since the `gluOrtho2D` function specifies an orthogonal projection, we need also to be sure that the coordinate values are placed in the OpenGL projection matrix.
- In addition, we could assign the identity matrix as the projection matrix before defining the world-coordinate range.
- This would ensure that the coordinate values were not accumulated with any values we may have previously set for the projection matrix.
- Thus, for our initial two-dimensional examples, we can define the coordinate frame for the screen display window with the following statements

```
glMatrixMode (GL_PROJECTION);  
glLoadIdentity ( );  
gluOrtho2D (xmin, xmax, ymin, ymax);
```

- The display window will then be referenced by coordinates (x_{min} , y_{min}) at the lower-left corner and by coordinates (x_{max} , y_{max}) at the upper-right corner, as shown in Figure below



- We can then designate one or more graphics primitives for display using the coordinate reference specified in the `gluOrtho2D` statement.
- If the coordinate extents of a primitive are within the coordinate range of the display window, all of the primitive will be displayed.
- Otherwise, only those parts of the primitive within the display-window coordinate limits will be shown.
- Also, when we set up the geometry describing a picture, all positions for the OpenGL primitives must be given in absolute coordinates, with respect to the reference frame defined in the `gluOrtho2D` function.

OpenGL Functions

Geometric Primitives:

- It includes points, line segments, polygon etc.
- These primitives pass through geometric pipeline which decides whether the primitive is visible or not and also how the primitive should be visible on the screen etc.
- The geometric transformations such rotation, scaling etc can be applied on the primitives which are displayed on the screen. The programmer can create geometric primitives as shown below:

```
glBegin(type);  
  
    glVertex*();  
    glVertex*();  
    ⋮  
    glVertex*();  
  
glEnd();
```

where:

glBegin indicates the beginning of the object that has to be displayed

glEnd indicates the end of primitive

OpenGL Point Functions

- The type within glBegin() specifies the type of the object and its value can be as follows:

GL_POINTS

- Each vertex is displayed as a point.
- The size of the point would be of at least one pixel.
- Then this coordinate position, along with other geometric descriptions we may have in our scene, is passed to the viewing routines.
- Unless we specify other attribute values, OpenGL primitives are displayed with a default size and color.
- The default color for primitives is white, and the default point size is equal to the size of a single screen pixel

Syntax:

Case 1:

```
glBegin (GL_POINTS);  
glVertex2i (50, 100);  
glVertex2i (75, 150);  
glVertex2i (100, 200);
```

glEnd ();

Case 2:

- we could specify the coordinate values for the preceding points in arrays such as

int point1 [] = {50, 100};

int point2 [] = {75, 150};

int point3 [] = {100, 200};

and call the OpenGL functions for plotting the three points as

glBegin (GL_POINTS);

glVertex2iv (point1);

glVertex2iv (point2);

glVertex2iv (point3);

glEnd ();

Case 3:

- specifying two point positions in a three dimensional world reference frame. In this case, we give the coordinates as explicit floating-point values:

glBegin (GL_POINTS);

glVertex3f (-78.05, 909.72, 14.60);

glVertex3f (261.91, -5200.67, 188.33);

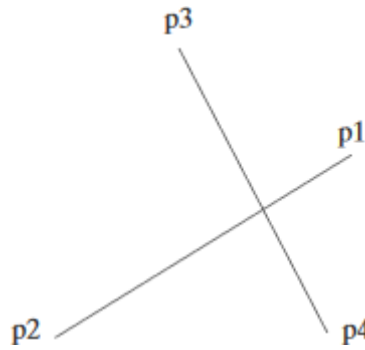
glEnd ();

OpenGL LINE FUNCTIONS

- Primitive type is GL_LINES
- Successive pairs of vertices are considered as endpoints and they are connected to form an individual line segments.
- Note that successive segments usually are disconnected because the vertices are processed on a pair-wise basis.
- we obtain one line segment between the first and second coordinate positions and another line segment between the third and fourth positions.
- if the number of specified endpoints is odd, so the last coordinate position is ignored.

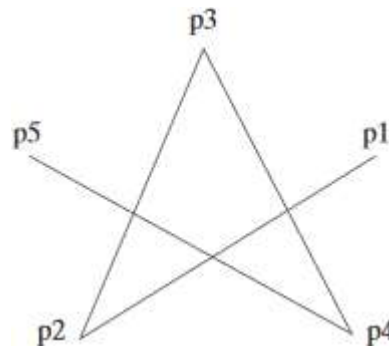
Case 1: Lines

```
glBegin (GL_LINES);  
    glVertex2iv (p1);  
    glVertex2iv (p2);  
    glVertex2iv (p3);  
    glVertex2iv (p4);  
    glVertex2iv (p5);  
glEnd ( );
```

**Case 2: GL_LINE_STRIP:**

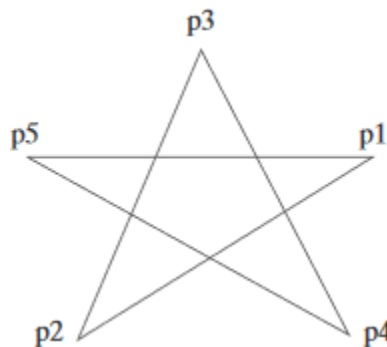
Successive vertices are connected using line segments. However, the final vertex is not connected to the initial vertex.

```
glBegin (GL_LINES_STRIP);  
    glVertex2iv (p1);  
    glVertex2iv (p2);  
    glVertex2iv (p3);  
    glVertex2iv (p4);  
    glVertex2iv (p5);  
glEnd ( );
```

**Case 3: GL_LINE_LOOP:**

Successive vertices are connected using line segments to form a closed path or loop i.e., final vertex is connected to the initial vertex.

```
glBegin (GL_LINES_LOOP);  
    glVertex2iv (p1);  
    glVertex2iv (p2);  
    glVertex2iv (p3);  
    glVertex2iv (p4);  
    glVertex2iv (p5);  
glEnd ( );
```



Point Attributes

- ➔ Basically, we can set two attributes for points: color and size.
- ➔ In a state system: The displayed color and size of a point is determined by the current values stored in the attribute list.
- ➔ Color components are set with RGB values or an index into a color table.
- ➔ For a raster system: Point size is an integer multiple of the pixel size, so that a large point is displayed as a square block of pixels

OpenGL Point-Attribute Functions

Color:

- ➔ The displayed color of a designated point position is controlled by the current color values in the state list.
- ➔ Also, a color is specified with either the glColor function or the glIndex function.

Size:

- ➔ We set the size for an OpenGL point with
glPointSize (size);
and the point is then displayed as a square block of pixels.
- ➔ Parameter size is assigned a positive floating-point value, which is rounded to an integer (unless the point is to be antialiased).
- ➔ The number of horizontal and vertical pixels in the display of the point is determined by parameter size.
- ➔ Thus, a point size of 1.0 displays a single pixel, and a point size of 2.0 displays a 2×2 pixel array.
- ➔ If we activate the antialiasing features of OpenGL, the size of a displayed block of pixels will be modified to smooth the edges.
- ➔ The default value for point size is 1.0.

Example program:

- ➔ Attribute functions may be listed inside or outside of a glBegin/glEnd pair.
- ➔ Example: the following code segment plots three points in varying colors and sizes.

- ➔ The first is a standard-size red point, the second is a double-size green point, and the third is a triple-size blue point:

Ex:

```
glColor3f (1.0, 0.0, 0.0);  
glBegin (GL_POINTS);  
glVertex2i (50, 100);  
glPointSize (2.0);  
glColor3f (0.0, 1.0, 0.0);  
glVertex2i (75, 150);  
glPointSize (3.0);  
glColor3f (0.0, 0.0, 1.0);  
glVertex2i (100, 200);  
glEnd ( );
```

Line-Attribute Functions OpenGL

- ➔ In OpenGL straight-line segment with three attribute settings: line color, line-width, and line style.
- ➔ OpenGL provides a function for setting the width of a line and another function for specifying a line style, such as a dashed or dotted line.

OpenGL Line-Width Function

- ➔ Line width is set in OpenGL with the function

Syntax: glLineWidth (width);

- ➔ We assign a floating-point value to parameter width, and this value is rounded to the nearest nonnegative integer.
- ➔ If the input value rounds to 0.0, the line is displayed with a standard width of 1.0, which is the default width.
- ➔ Some implementations of the line-width function might support only a limited number of widths, and some might not support widths other than 1.0.

- ➔ That is, the magnitude of the horizontal and vertical separations of the line endpoints, Δx and Δy , are compared to determine whether to generate a thick line using vertical pixel spans or horizontal pixel spans.

OpenGL Line-Style Function

- ➔ By default, a straight-line segment is displayed as a solid line.
- ➔ But we can also display dashed lines, dotted lines, or a line with a combination of dashes and dots.
- ➔ We can vary the length of the dashes and the spacing between dashes or dots.
- ➔ We set a current display style for lines with the OpenGL function:

Syntax: `glLineStipple (repeatFactor, pattern);`

Pattern:

- ➔ Parameter pattern is used to reference a 16-bit integer that describes how the line should be displayed.
- ➔ 1 bit in the pattern denotes an “on” pixel position, and a 0 bit indicates an “off” pixel position.
- ➔ The pattern is applied to the pixels along the line path starting with the low-order bits in the pattern.
- ➔ The default pattern is 0xFFFF (each bit position has a value of 1), which produces a solid line.

repeatFactor

- ➔ Integer parameter repeatFactor specifies how many times each bit in the pattern is to be repeated before the next bit in the pattern is applied.
- ➔ The default repeat value is 1.

Polyline:

- ➔ With a polyline, a specified line-style pattern is not restarted at the beginning of each segment.

- ➔ It is applied continuously across all the segments, starting at the first endpoint of the polyline and ending at the final endpoint for the last segment in the series.

Example:

- ➔ For line style, suppose parameter pattern is assigned the hexadecimal representation 0x00FF and the repeat factor is 1.
- ➔ This would display a dashed line with eight pixels in each dash and eight pixel positions that are “off” (an eight-pixel space) between two dashes.
- ➔ Also, since low order bits are applied first, a line begins with an eight-pixel dash starting at the first endpoint.
- ➔ This dash is followed by an eight-pixel space, then another eight-pixel dash, and so forth, until the second endpoint position is reached.

Activating line style:

- Before a line can be displayed in the current line-style pattern, we must activate the line-style feature of OpenGL.

glEnable (GL_LINE_STIPPLE);

- If we forget to include this enable function, solid lines are displayed; that is, the default pattern 0xFFFF is used to display line segments.
- At any time, we can turn off the line-pattern feature with

glDisable (GL_LINE_STIPPLE);

- This replaces the current line-style pattern with the default pattern (solid lines).

Example Code:

```
typedef struct { float x, y; } wcPt2D;  
wcPt2D dataPts [5];  
void linePlot (wcPt2D dataPts [5])  
{  
    int k;  
    glBegin (GL_LINE_STRIP);  
    for (k = 0; k < 5; k++)  
        glVertex2f (dataPts [k].x, dataPts [k].y);  
}
```

```
        glFlush ();
        glEnd ( );
    }
    /* Invoke a procedure here to draw coordinate axes. */
    glEnable (GL_LINE_STIPPLE); /* Input first set of (x, y) data values. */
    glLineStipple (1, 0x1C47); // Plot a dash-dot, standard-width polyline.
    linePlot (dataPts);
    /* Input second set of (x, y) data values. */
    glLineStipple (1, 0x00FF); // Plot a dashed, double-width polyline.
    glLineWidth (2.0);
    linePlot (dataPts);
    /* Input third set of (x, y) data values. */
    glLineStipple (1, 0x0101); // Plot a dotted, triple-width polyline.
    glLineWidth (3.0);
    linePlot (dataPts);
    glDisable (GL_LINE_STIPPLE);
```

Curve Attributes

- ➔ Parameters for curve attributes are the same as those for straight-line segments.
- ➔ We can display curves with varying colors, widths, dot-dash patterns, and available pen or brush options.
- ➔ Methods for adapting curve-drawing algorithms to accommodate attribute selections are similar to those for line drawing.
- ➔ Raster curves of various widths can be displayed using the method of horizontal or vertical pixel spans.

Case 1: Where the magnitude of the curve slope $|m| \leq 1.0$, we plot vertical spans;

Case 2: when the slope magnitude $|m| > 1.0$, we plot horizontal spans.

Different methods to draw a curve:

Method 1: Using circle symmetry property, we generate the circle path with vertical spans in the octant from $x = 0$ to $x = y$, and then reflect pixel positions about the line $y = x$ to $y=0$

Method 2: Another method for displaying thick curves is to fill in the area between two Parallel curve paths, whose separation distance is equal to the desired width. We could do this using the specified curve path as one boundary and setting up the second boundary either inside or outside the original curve path. This approach, however, shifts the original curve path either inward or outward, depending on which direction we choose for the second boundary.

Method 3: The pixel masks discussed for implementing line-style options could also be used in raster curve algorithms to generate dashed or dotted patterns

Method 4: Pen (or brush) displays of curves are generated using the same techniques discussed for straight-line segments.

Method 5: Painting and drawing programs allow pictures to be constructed interactively by using a pointing device, such as a stylus and a graphics tablet, to sketch various curve shapes.

Line Drawing Algorithm

- ✓ A straight-line segment in a scene is defined by coordinate positions for the endpoints of the segment.
- ✓ To display the line on a raster monitor, the graphics system must first project the endpoints to integer screen coordinates and determine the nearest pixel positions along the line path between the two endpoints then the line color is loaded into the frame buffer at the corresponding pixel coordinates
- ✓ The Cartesian slope-intercept equation for a straight line is

$$y = m * x + b \text{-----} (1)$$

with m as the slope of the line and b as the y intercept.

- ✓ Given that the two endpoints of a line segment are specified at positions (x0,y0) and (xend, yend), as shown in fig.

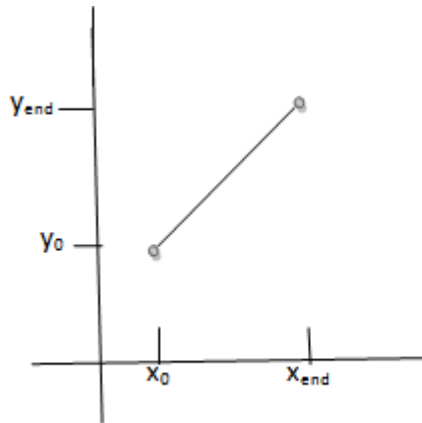


fig. Line path between endpoint positions (x_0, y_0) and (x_{end}, y_{end}) .

- ✓ We determine values for the slope m and y intercept b with the following equations:

$$m = (y_{end} - y_0) / (x_{end} - x_0) \text{ ----- } >(2)$$

$$b = y_0 - m \cdot x_0 \text{ ----- } >(3)$$

- ✓ Algorithms for displaying straight line are based on the line equation (1) and calculations given in eq(2) and (3).
- ✓ For given x interval δx along a line, we can compute the corresponding y interval δy from eq.(2) as

$$\delta y = m \cdot \delta x \text{ ----- } >(4)$$

- ✓ Similarly, we can obtain the x interval δx corresponding to a specified δy as

$$\delta x = \delta y / m \text{ ----- } >(5)$$

- ✓ These equations form the basis for determining deflection voltages in analog displays, such as vector-scan system, where arbitrarily small changes in deflection voltage are possible.
- ✓ For lines with slope magnitudes
 - ➔ $|m| < 1$, δx can be set proportional to a small horizontal deflection voltage with the corresponding vertical deflection voltage set proportional to δy from eq.(4)
 - ➔ $|m| > 1$, δy can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to δx from eq.(5)
 - ➔ $|m| = 1$, $\delta x = \delta y$ and the horizontal and vertical deflections voltages are equal

Bresenham's Algorithm:

- ➔ It is an efficient raster scan generating algorithm that uses incremental integral calculations
- ➔ To illustrate Bresenham's approach, we first consider the scan-conversion process for lines with positive slope less than 1.0.
- ➔ Pixel positions along a line path are then determined by sampling at unit x intervals. Starting from the left endpoint (x_0, y_0) of a given line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path.
- ➔ Consider the equation of a straight line $y=mx+c$ where $m=dy/dx$

Bresenham's Line-Drawing Algorithm for $|m| < 1.0$

1. Input the two line endpoints and store the left endpoint in (x_0, y_0) .
2. Set the color for frame-buffer position (x_0, y_0) ; i.e., plot the first point.
3. Calculate the constants Δx , Δy , $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

4. At each x_k along the line, starting at $k = 0$, perform the following test:
If $p_k < 0$, the next point to plot is $(x_k + 1, y_k)$ and

$$p_{k+1} = p_k + 2\Delta y$$

Otherwise, the next point to plot is $(x_k + 1, y_k + 1)$ and

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4 $\Delta x - 1$ more times.

Note:

If $|m| > 1.0$

Then

$$p_0 = 2\Delta x - \Delta y$$

and

If $p_k < 0$, the next point to plot is $(x_k, y_k + 1)$ and

$$p_{k+1} = p_k + 2\Delta x$$

Otherwise, the next point to plot is $(x_k + 1, y_k + 1)$ and

$$p_{k+1} = p_k + 2\Delta x - 2\Delta y$$

Code:

```
#include <stdlib.h>
#include <math.h>
/* Bresenham line-drawing procedure for |m| < 1.0. */
void lineBres (int x0, int y0, int xEnd, int yEnd)
{
    int dx = fabs (xEnd - x0), dy = fabs(yEnd - y0);
    int p = 2 * dy - dx;
    int twoDy = 2 * dy, twoDyMinusDx = 2 * (dy - dx);
    int x, y;
    /* Determine which endpoint to use as start position. */
    if (x0 > xEnd) {
        x = xEnd;
        y = yEnd;
        xEnd = x0;
    }
    else {
        x = x0;
        y = y0;
    }

    setPixel (x, y);
    while (x < xEnd) {
        x++;
        if (p < 0)
            p += twoDy;
```

```

        else {
            y++;
            p += twoDyMinusDx;

        }

        setPixel (x, y);
    }
}

```

OpenGL Polygon Fill-Area Functions

- ✓ A glVertex function is used to input the coordinates for a single polygon vertex, and a complete polygon is described with a list of vertices placed between a glBegin/glEnd pair.
- ✓ By default, a polygon interior is displayed in a solid color, determined by the current color settings we can fill a polygon with a pattern and we can display polygon edges as line borders around the interior fill.
- ✓ There are six different symbolic constants that we can use as the argument in the glBegin function to describe polygon fill areas
- ✓ In some implementations of OpenGL, the following routine can be more efficient than generating a fill rectangle using glVertex specifications:

```
glRect* (x1, y1, x2, y2);
```

- ✓ One corner of this rectangle is at coordinate position (x1, y1), and the opposite corner of the rectangle is at position (x2, y2).
- ✓ Suffix codes for glRect specify the coordinate data type and whether coordinates are to be expressed as array elements.
- ✓ These codes are i (for integer), s (for short), f (for float), d (for double), and v (for vector).
- ✓ Example

```
glRecti (200, 100, 50, 250);
```

If we put the coordinate values for this rectangle into arrays, we can generate the same square with the following code:

```
int vertex1 [ ] = {200, 100}; int
vertex2 [ ] = {50, 250}; glRectiv
(vertex1, vertex2);
```

Polygon

- ❖ With the OpenGL primitive constant GL POLYGON, we can display a single polygon fill area.
- ❖ Each of the points is represented as an array of (x, y) coordinate values:

```
glBegin (GL_POLYGON);
```

```
    glVertex2iv (p1);
```

```
    glVertex2iv (p2);
```

```
    glVertex2iv (p3);
```

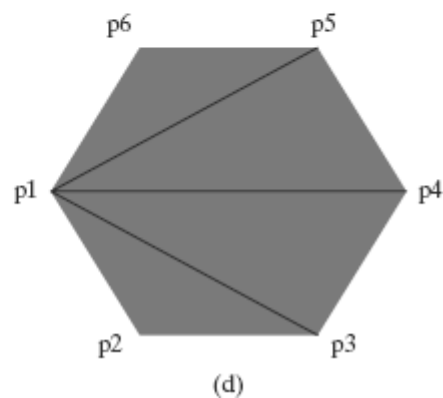
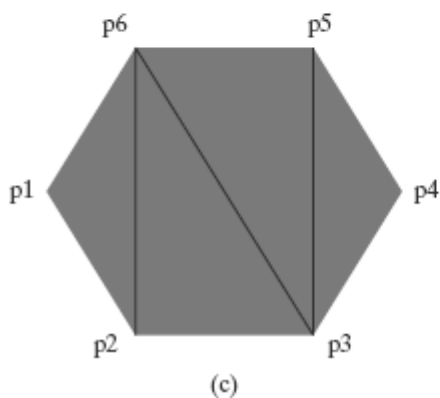
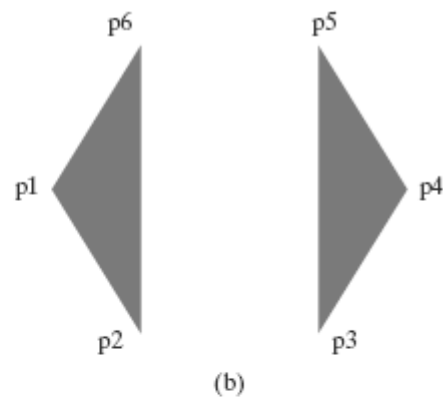
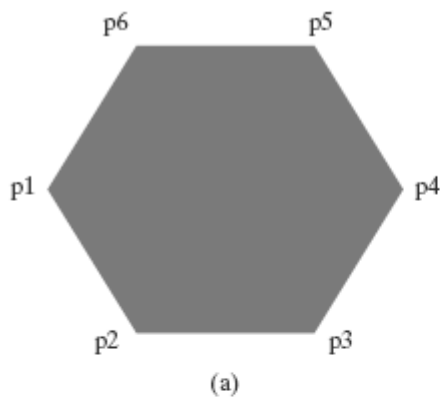
```
    glVertex2iv (p4);
```

```
    glVertex2iv (p5);
```

```
    glVertex2iv (p6);
```

```
glEnd ( );
```

- ❖ A polygon vertex list must contain at least three vertices. Otherwise, nothing is displayed.



- (a) A single convex polygon fill area generated with the primitive constant GL POLYGON. (b) Two unconnected triangles generated with GL TRIANGLES.
- (c) Four connected triangles generated with GL TRIANGLE STRIP.
- (d) Four connected triangles generated with GL TRIANGLE FAN.

Triangles

- ❖ Displays the triangles.
- ❖ Three primitives in triangles, GL_TRIANGLES, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP

```
glBegin (GL_TRIANGLES);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p6);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
glEnd ( );
```

- ❖ In this case, the first three coordinate points define the vertices for one triangle, the next three points define the next triangle, and so forth.
- ❖ For each triangle fill area, we specify the vertex positions in a counterclockwise order triangle strip

```
glBegin (GL_TRIANGLE_STRIP);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p6);
    glVertex2iv (p3);
    glVertex2iv (p5);
    glVertex2iv (p4);
glEnd ( );
```

- ❖ Assuming that no coordinate positions are repeated in a list of N vertices, we obtain $N - 2$ triangles in the strip. Clearly, we must have $N \geq 3$ or nothing is displayed.

- ❖ Each successive triangle shares an edge with the previously defined triangle, so the ordering of the vertex list must be set up to ensure a consistent display.
- ❖ Example, our first triangle ($n = 1$) would be listed as having vertices (p1, p2, p6). The second triangle ($n = 2$) would have the vertex ordering (p6, p2, p3). Vertex ordering for the third triangle ($n = 3$) would be (p6, p3, p5). And the fourth triangle ($n = 4$) would be listed in the polygon tables with vertex ordering (p5, p3, p4).

Triangle Fan

- ❖ Another way to generate a set of connected triangles is to use the “fan” Approach

```
glBegin (GL_TRIANGLE_FAN);
```

```
    glVertex2iv    (p1);
```

```
    glVertex2iv    (p2);
```

```
    glVertex2iv    (p3);
```

```
    glVertex2iv    (p4);
```

```
    glVertex2iv    (p5);
```

```
    glVertex2iv (p6);
```

```
glEnd ( );
```

- ❖ For N vertices, we again obtain $N-2$ triangles, providing no vertex positions are repeated, and we must list at least three vertices be specified in the proper order to define front and back faces for each triangle correctly.
- ❖ Therefore, triangle 1 is defined with the vertex list (p1, p2, p3); triangle 2 has the vertex ordering (p1, p3, p4); triangle 3 has its vertices specified in the order (p1, p4, p5); and triangle 4 is listed with vertices (p1, p5, p6).

Quadrilaterals

- ✓ OpenGL provides for the specifications of two types of quadrilaterals.
- ✓ With the GL QUADS primitive constant and the following list of eight vertices, specified as two-dimensional coordinate arrays, we can generate the display shown in Figure (a):

```
glBegin (GL_QUADS);
```

```
    glVertex2iv    (p1);
```

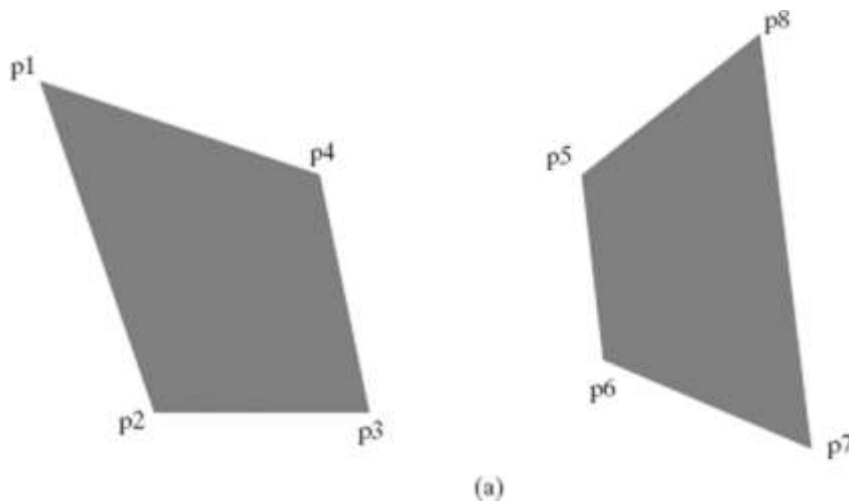
```
    glVertex2iv    (p2);
```

```
    glVertex2iv    (p3);
```

```

        glVertex2iv (p4);
        glVertex2iv (p5);
        glVertex2iv (p6);
        glVertex2iv (p7);
        glVertex2iv (p8);
    glEnd ();

```

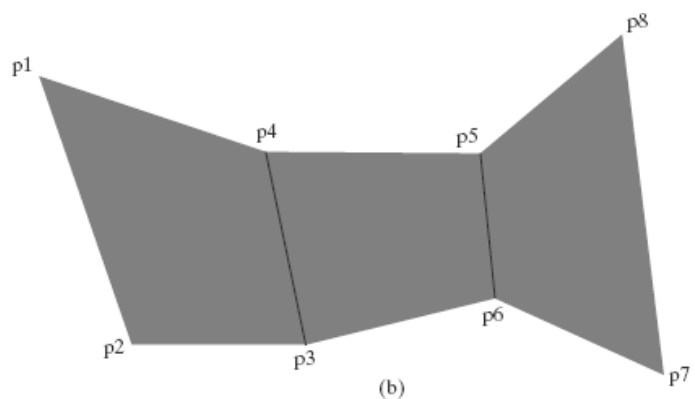


- ✓ Rearranging the vertex list in the previous quadrilateral code example and changing the primitive constant to GL_QUAD_STRIP, we can obtain the set of connected quadrilaterals shown in Figure (b):

```

glBegin (GL_QUAD_STRIP);
    glVertex2iv (p1); glVertex2iv
    (p2); glVertex2iv (p4);
    glVertex2iv (p3); glVertex2iv
    (p5); glVertex2iv (p6);
    glVertex2iv (p8); glVertex2iv
    (p7);
glEnd ();

```



- ✓ For a list of N vertices, we obtain $N/2 - 1$ quadrilaterals, providing that $N \geq 4$. Thus, our first quadrilateral ($n = 1$) is listed as having a vertex ordering of (p1, p2, p3, p4). The second quadrilateral ($n=2$) has the vertex ordering (p4, p3, p6, p5), and the vertex ordering for the third quadrilateral ($n=3$) is (p5, p6, p7, p8).