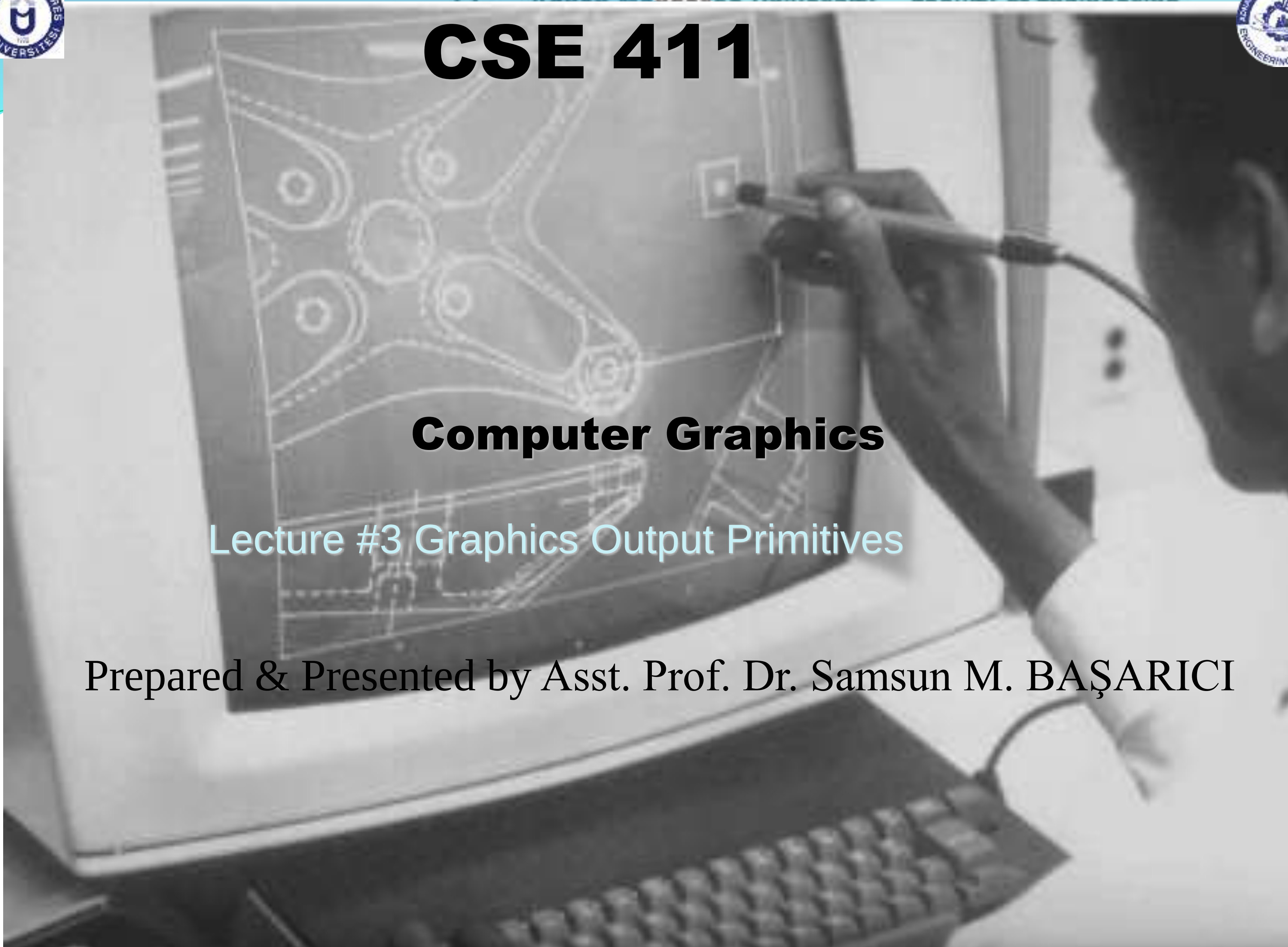


CSE 411

Computer Graphics

Lecture #3 Graphics Output Primitives

Prepared & Presented by Asst. Prof. Dr. Samsun M. BAŞARICI

Objectives

- HB Ch. 4 & GVS Ch. 7 (partly)
- Coordinate reference frames
- Two-dimensional world reference
- OpenGL Point Functions
- OpenGL Line Functions
- Polygon Fill Areas & OpenGL functions
- OpenGL Vertex Arrays
- Character Primitives & OpenGL functions

Graphics Output Primitives

- Graphics output primitives
  - Functions used to describe the various picture components
  - Examples: car, house, flower, …
- Geometric primitives
  - Functions used to describe points, lines, triangles, circles, …

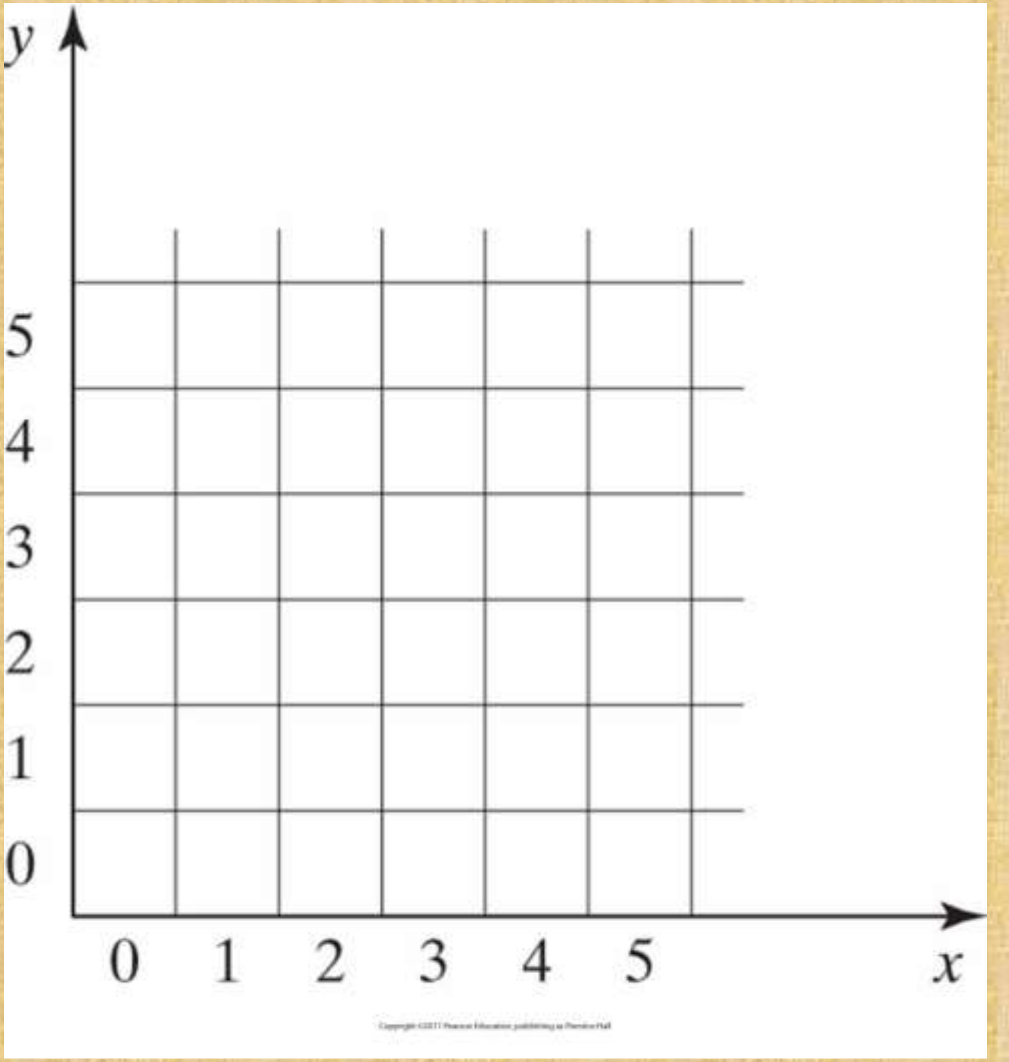
Coordinate Reference Frames

- Cartesian coordinate system
  - Can be 2D or 3D
  - Objects are associated to a set of coordinates
  - World coordinates are associated to a scene
- Object description
  - Coordinates of vertices
  - Color
  - Coordinate extents (min and max for each (x,y,z) in object – also called the bounding box
    - In 2D – bounding rectangle

Coordinate Reference Frames (cont.)

- Screen coordinates
  - Location of object on a monitor
  - Start from upper left corner (origin (0,0))
  - Pixel coordinates
    - Scan line number (y)
    - Column number (x)
  - Other origin → lower left corner (0,0)
  - Pixel coordinate references the center of the pixel
    - setPixel (x, y)
    - getPixel (x, y, color)
    - Depth value is 0 in 2D

Pixel positions



Referenced with respect to the lower-left corner of a screen area.

Coordinate Specifications

- Absolute coordinate values

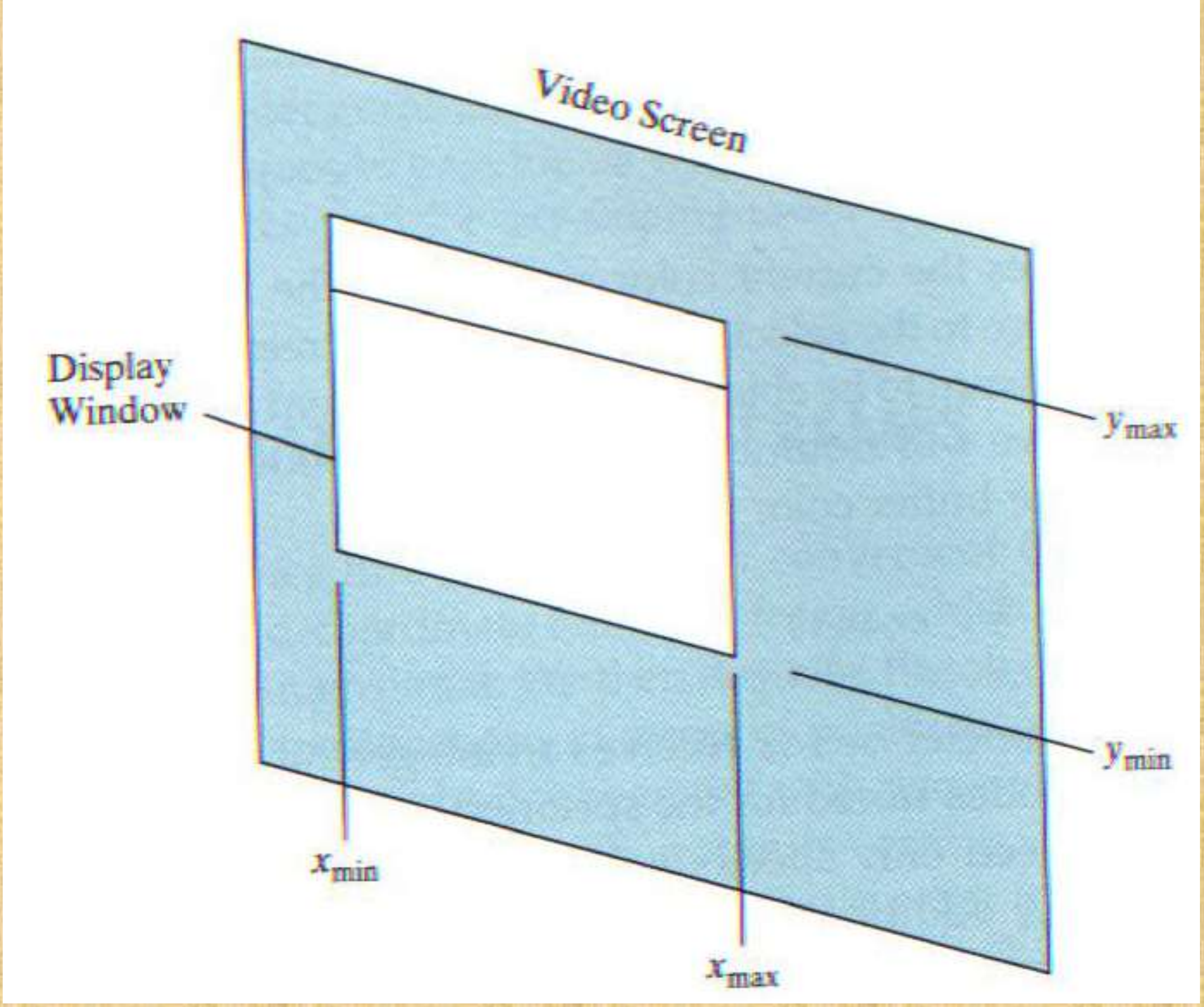
- Relative coordinate values:
  - Current position + offset

2D World Reference

- gluOrtho2D (xMin, xMax, yMin, yMax)
  - References display window as a rectangle with the minimum and maximum values listed
  - Absolute coordinates within these ranges will be displayed

```
glMatrixMode(GL_PROJECTION);
                    // set projection parameters to 2D
glLoadIdentity();   // sets projection matrix to identity
gluOrtho2D(0.0, 200.0, 0.0, 150.0);
                    // set coordinate values
                    // with vertices (0,0) for lower left
corner
                    // and (200, 150) for upper right corner
```

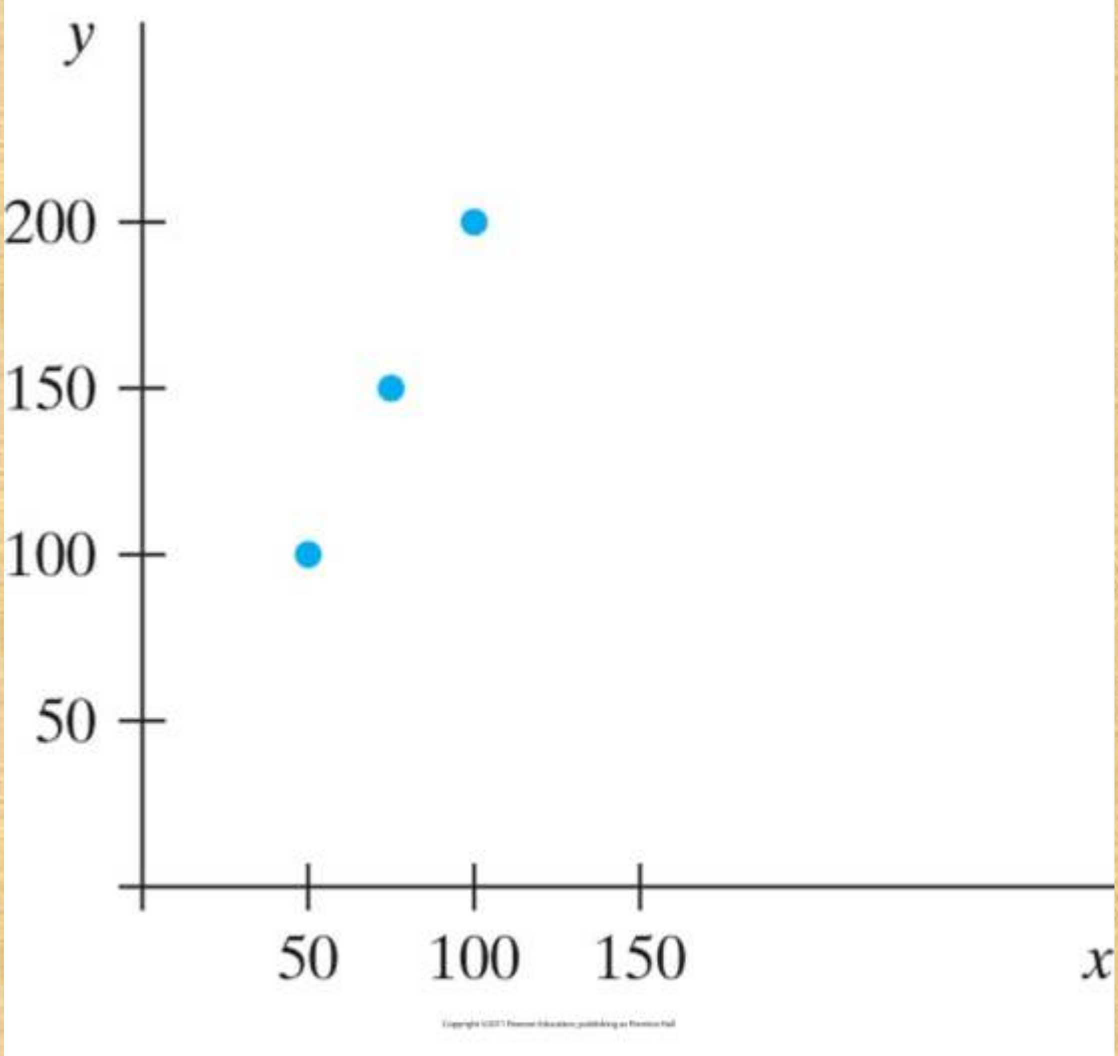
glOrtho2D Function



Point Functions

- Point
  - Coordinates
  - Color – default color is white
  - Size – one screen pixel by default (`glPointSize`)
  - ```
    glBegin (GL_POINTS)
        glVertex2i (50, 100);
        glVertex2i (75, 150);
        glVertex2i (100, 200);
    glEnd();
    ```
  - Coordinates can also be set in an int []:
    ```
    int point1 [] = {50, 100};
    …
    glVertex2iv (point1);
    ```
    ```

Example: Three Point Positions



Generated with glBegin (GL_POINTS).

OpenGL Line Functions

- Line
  - Defined by two endpoint coordinates (one line segment)

```
glBegin(GL_LINES );
      glVertex2i( 180, 15 );
      glVertex2i( 10, 145 );
glEnd();
```

  - If several vertices, a line is drawn between the first and second, then a separate one between the third and the fourth, etc. (isolated vertices are not drawn).

OpenGL Line Functions (cont.)

- Polyline
  - Defined by line connecting all the points

```
glBegin(GL_LINE_STRIP );
        glVertex2i( 180, 15 );
        glVertex2i( 10, 145 );
         glVertex2i( 100, 20 );
        glVertex2i( 30, 150 );
glEnd();
```

  - Draws a line between vertex 1 and vertex 2
    then between vertex 2 and vertex 3
    then between vertex 3 and vertex 4.

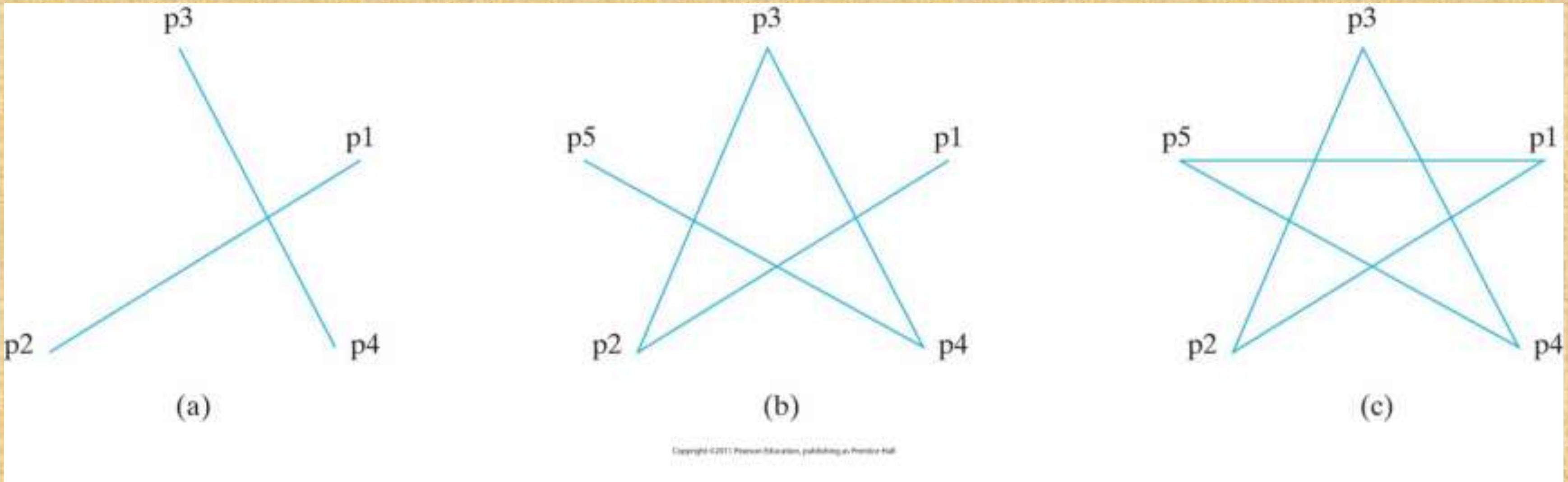
OpenGL Line Functions (cont.)

- Polyline
  - In addition to GL_LINE_STRIP, adds a line between the last vertex and the first one

```
glBegin(GL_LINE_LOOP );
        glVertex2i( 180, 15 );
        glVertex2i( 10, 145 );
         glVertex2i( 100, 20 );
        glVertex2i( 30, 150 );
glEnd();
```

  - Draws a line between vertex 1 and vertex 2
    then between vertex 2 and vertex 3
    then between vertex 3 and vertex 4
    then between vertex 4 and vertex 1.

Example: Line segments



With five endpoint coordinates

(a) An unconnected set of lines generated with the primitive line constant GL_LINES.

(b) A polyline generated with GL_LINE_STRIP.

(c) A closed polyline generated with GL_LINE_LOOP.

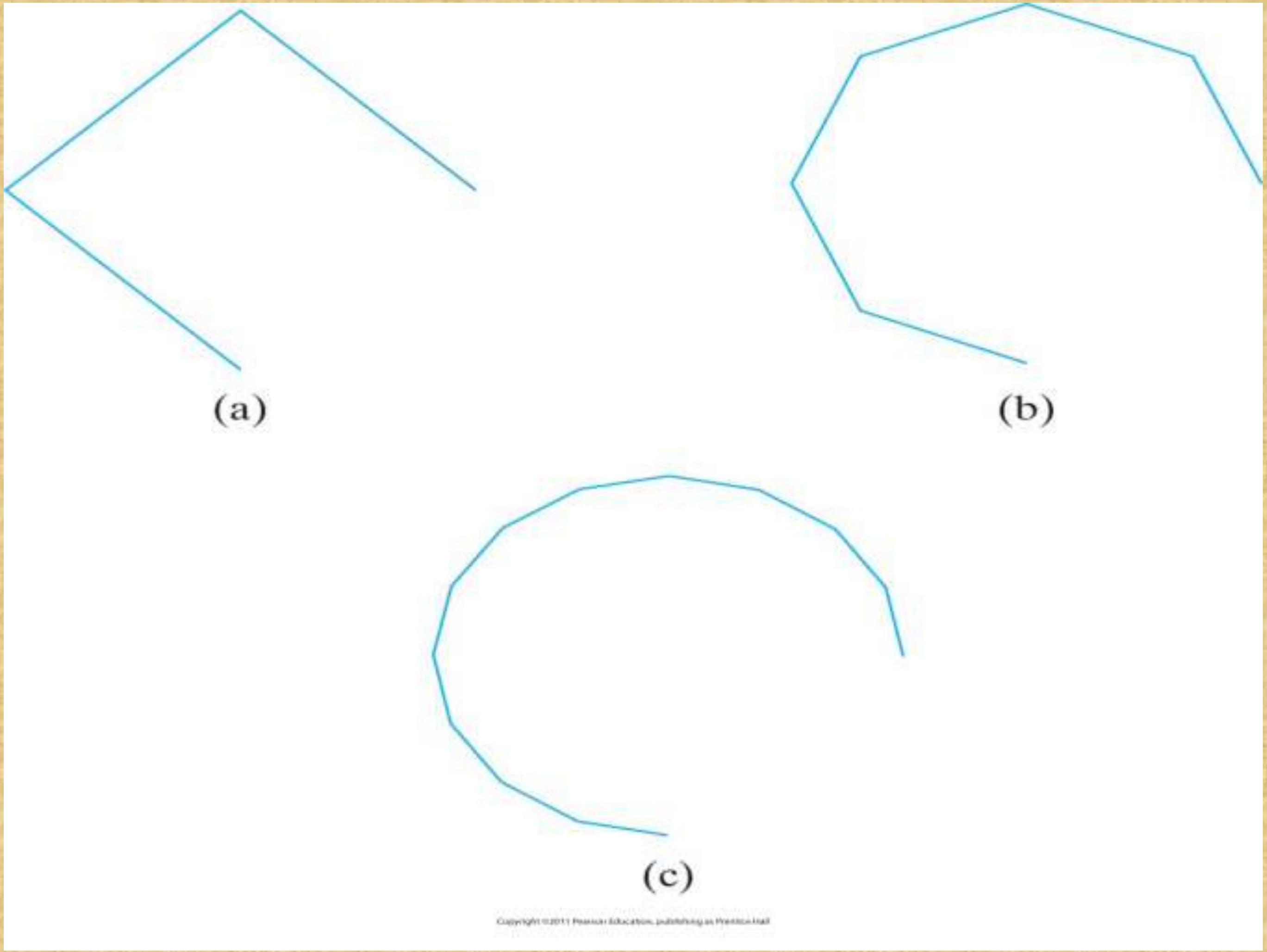
OpenGL Curve Functions

- Not included in OpenGL core library (only Bézier splines: polynomials defined with a discrete point set)
- GLU has routines for 3D quadrics like spheres, cylinders and also rational B-splines
- GLUT has routines for 3D quadrics like spheres, cones and others

OpenGL Curve Functions (cont.)

- How to draw curves?
- Solution: Approximating using polyline

Curve Approximation

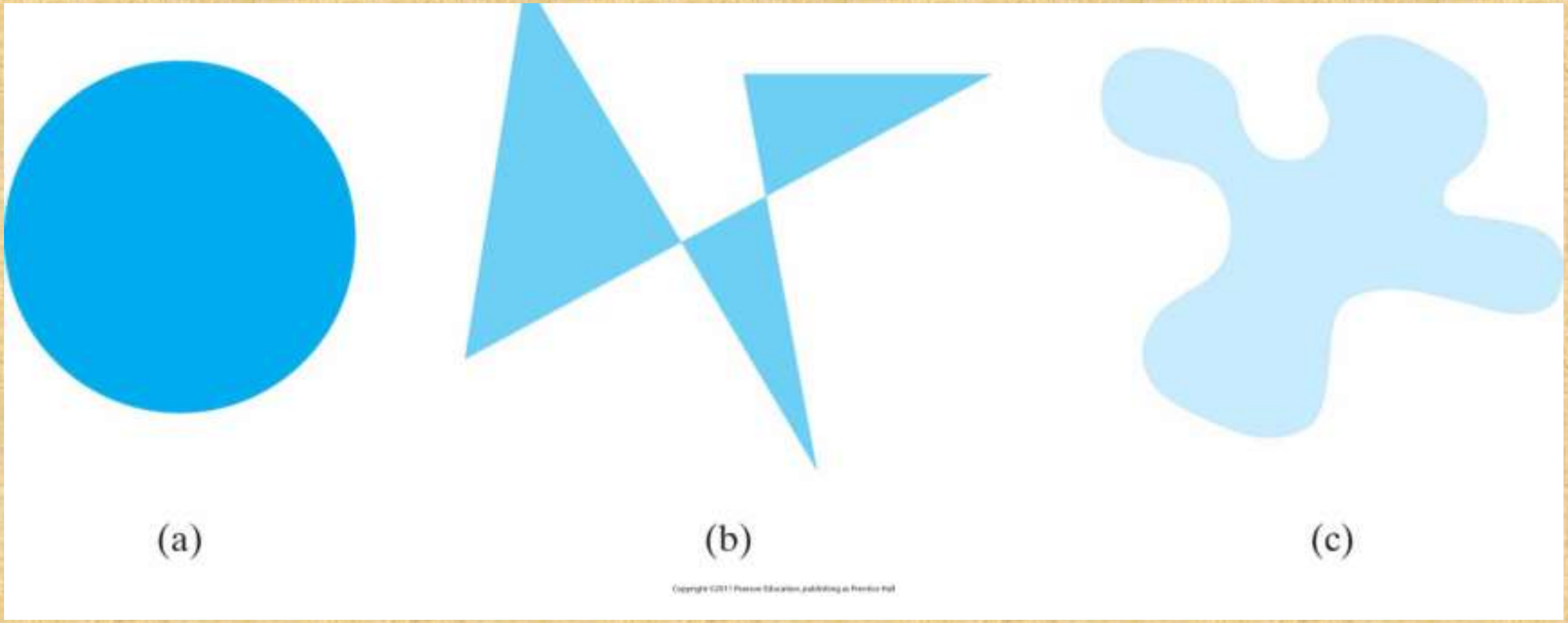


A circular arc approximated with (a) three straight-line segments, (b) six line segments, and (c) twelve line segments.

Fill-Area Primitives

- Fill-areas
  - Area filled with a certain color
  - Most often the shape is that of a polygon
  - Boundaries are linear
  - Most curved surfaces can be approximated with polygon facets (surface fitting with polygon mesh)
  - Standard graphics objects are objects made of a set of polygon surface patches.

Solid-color fill areas curved boundary



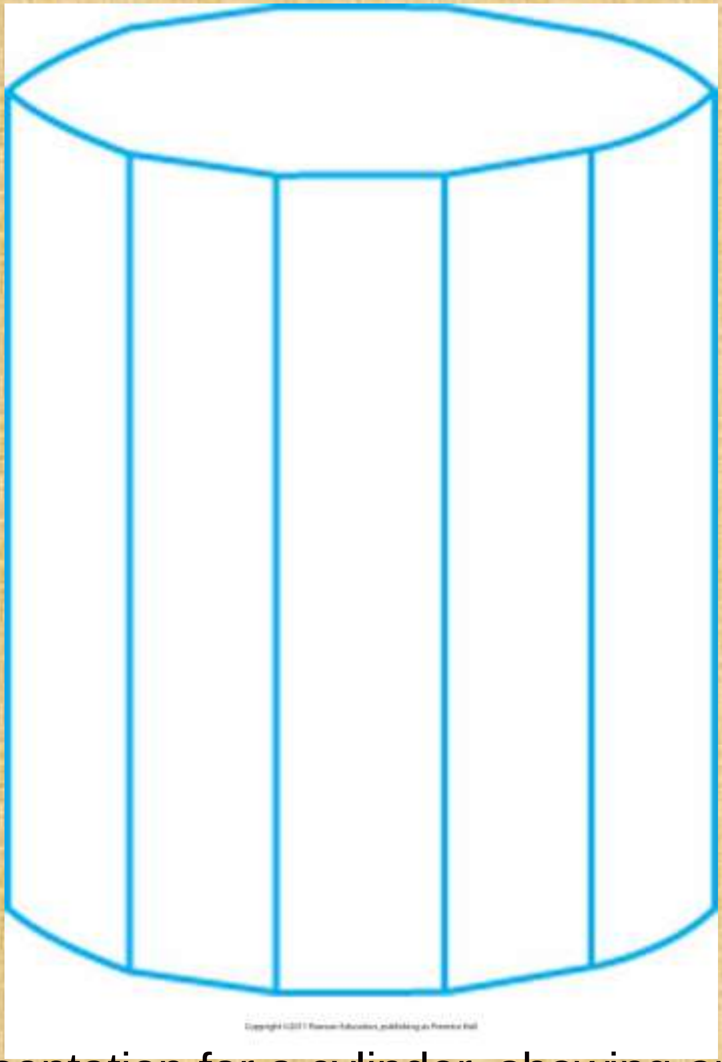
Specified with various boundaries.

(a) A circular fill region

(b) A fill area bounded by a closed polyline

(c) A filled area specified with an irregular curved boundary

Approximating a curved surface



Wire-frame representation for a cylinder, showing only the front (visible) faces of the polygon mesh used to approximate the surfaces.

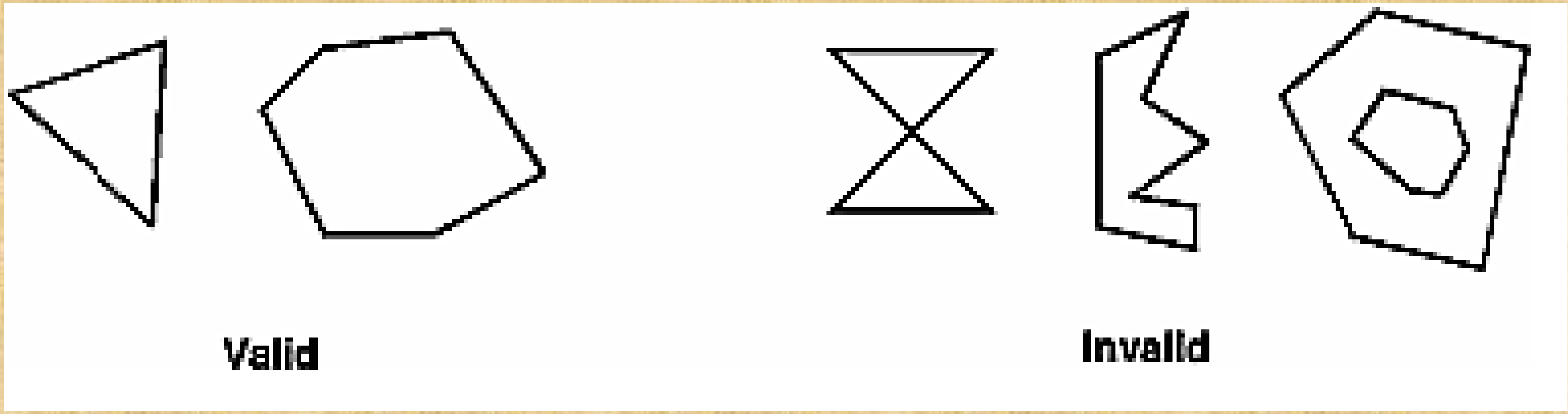
Polygon Fill-Areas

- Polygon classification
  - Polygon is a figure with three or more *vertices* and vertices are connected by a sequence of straight line called *edges* or *sides*
  - A polygon should be closed and with no edges crossing
  - Convex polygon has all interior angles less than or equal to 180º, line joining two interior points is also interior to the polygon
  - Concave polygon otherwise

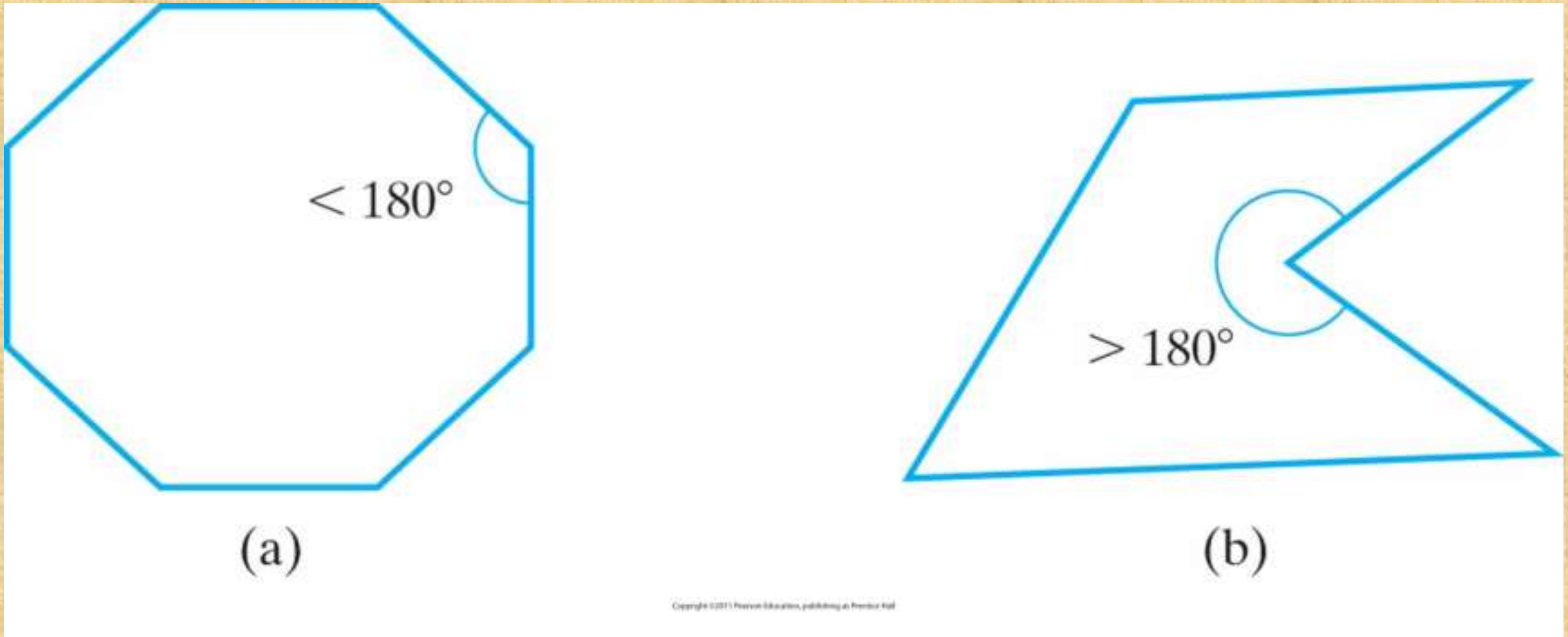
OpenGL Fill Area Functions

- OpenGL requires all polygons to be convex
- If need to draw concave polygons, then split them into convex polygons
- GLU library contains routines to convert concave polygons into a set of triangles, triangle mashes, triangle fans and straight line segments

Valid and Invalid Polygons

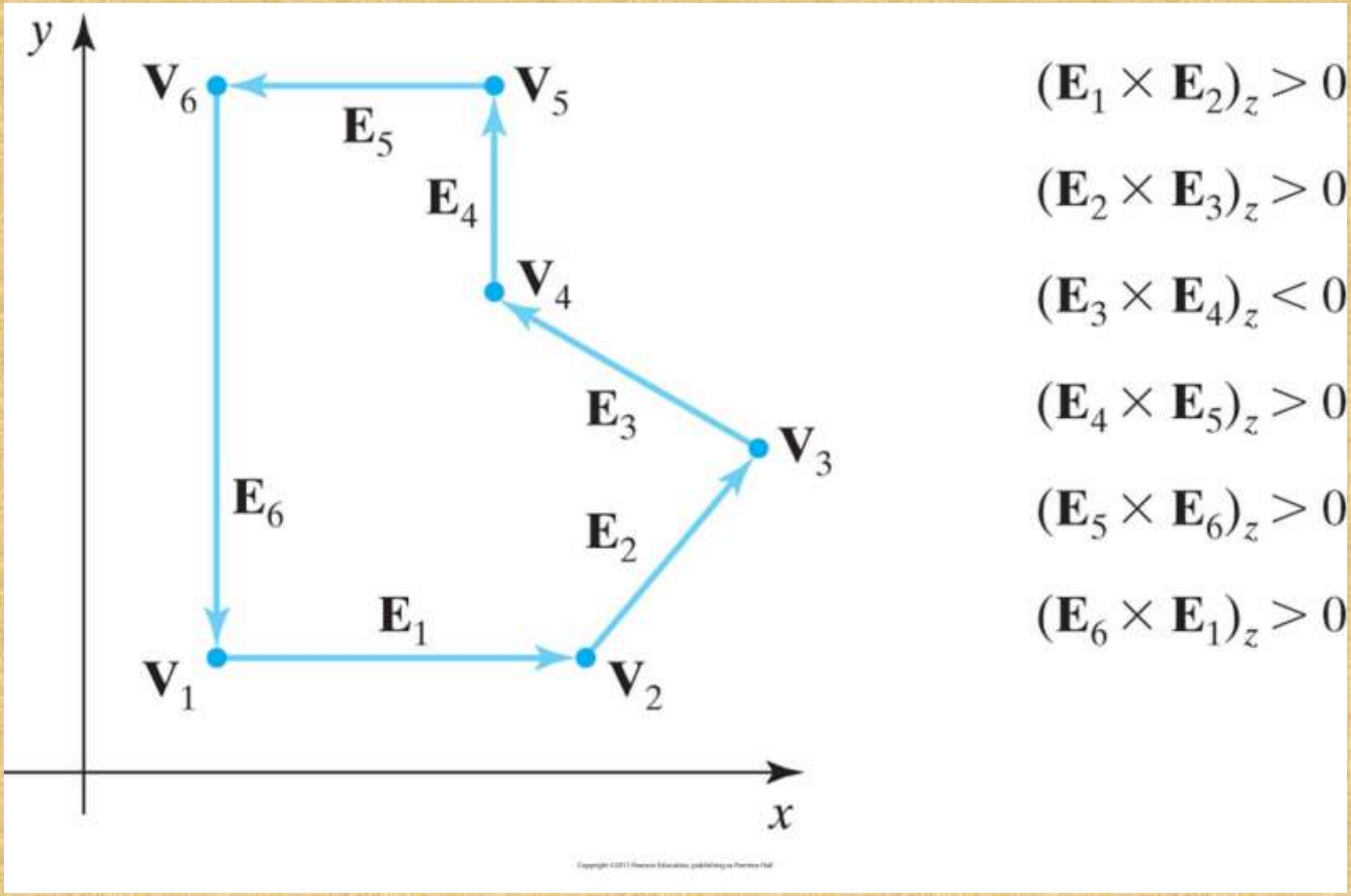


Convex and Concave Polygons



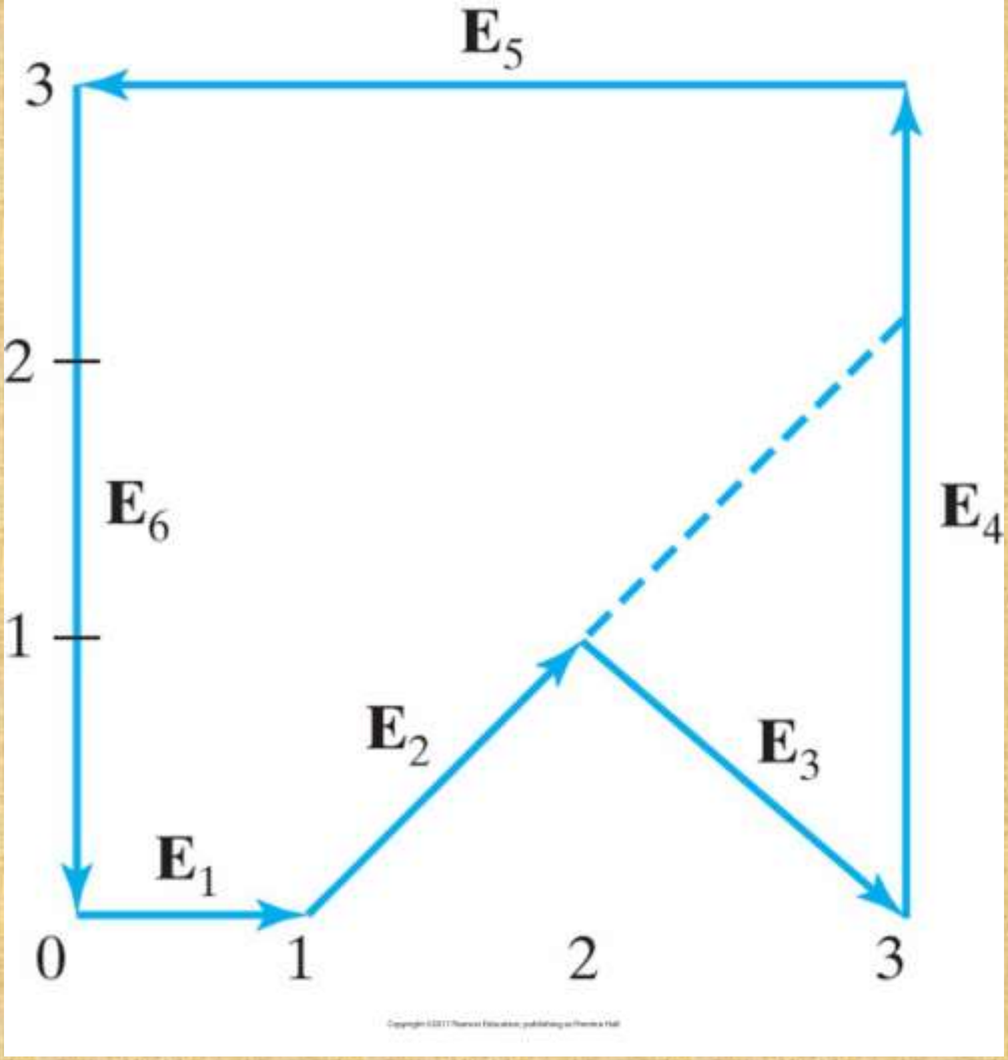
A convex polygon (a), and a concave polygon (b).

Identifying a concave polygon



By calculating cross-products of successive pairs of edge vectors

Splitting a concave polygon



Using the vector method.

Example: Splitting a concave polygon

- Polygon from Slide 27:
  - Edge vectors:
  - *z* component is 0 because all edges are in *xy* plane
    - $E_1 = (1, 0, 0)$ $E_2 = (1, 1, 0)$ $E_3 = (1, -1, 0)$
    - $E_4 = (0, 2, 0)$ $E_5 = (-3, 0, 0)$ $E_6 = (0, -2, 0)$

- (Remember) The cross-product $E_j \times E_k$ for two successive edge vectors is a vector perpendicular the *xy* plane with *z* component equal to $E_{jx} E_{ky} - E_{kx} E_{jy}$

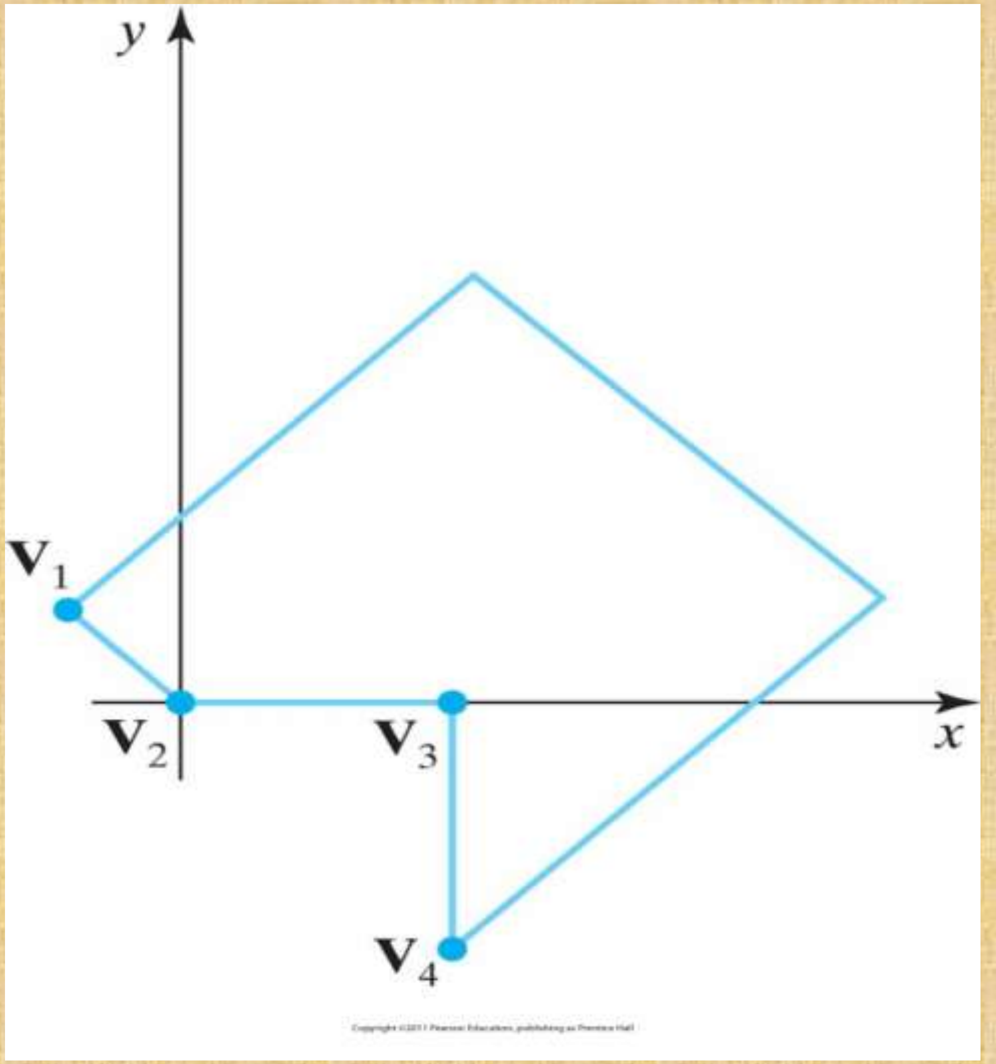
Example: Splitting a concave polygon (cont.)

- So:
  - $E_1 \times E_2 = (0, 0, 1)$ $E_2 \times E_3 = (0, 0, -2)$
  - $E_3 \times E_4 = (0, 0, 2)$ $E_4 \times E_5 = (0, 0, 6)$
  - $E_5 \times E_6 = (0, 0, 6)$ $E_6 \times E_1 = (0, 0, 2)$
- $E_2 \times E_3$ negative, split along the line of vector E_2

Example: Splitting a concave polygon (cont.)

- Line equation for E_2:
  - Slope 1
  - *y* intercept -1
    - (Remember: $y = mx + b$, $m = \frac{y_{end} - y_0}{x_{end} - x_0}$, $b = y_0 - m\, x_0$)

Splitting a concave polygon



Using the rotational method

Example: Splitting a concave polygon

- The algorithm:
  1. Shift each vertex V_k to origin
  2. Rotate so that next vertex V_{k+1} is on the x-axis
  3. If next vertex V_{k+2} is below x-axis split

- Example: Polygon from Slide 31:
  - After moving V_2 to the coordinate origin and rotating V_3 onto the x axis, we find that V_4 is below the x axis. So we split the polygon along the line of $\overline{V_2V_3}$, which is the x axis.

Inside-Outside Tests

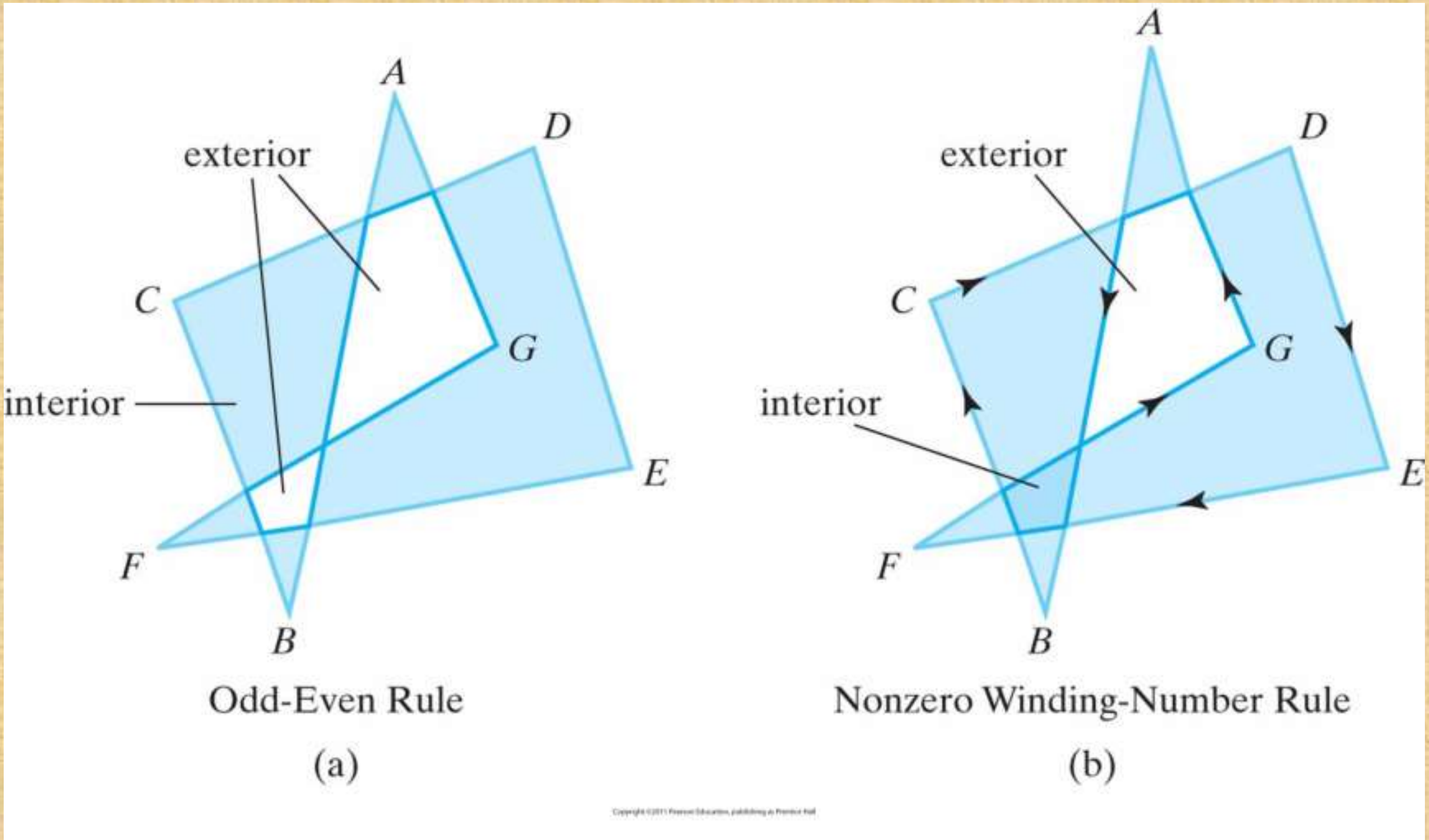
- In CG applications often interior regions of objects have to be identified.

- Approaches:
  - Odd-even rule:
    1. Draw a line from a point to outside of coordinate extents
    2. Count line segments of the object crossing this line
    3. If the number is odd then the point is interior, else exterior

Inside-Outside Tests (cont.)

- Nonzero winding-number rule:
  1. Init winding-number to 0
  2. Draw a line from a point
  3. Move along the line
  4. Count line segments of object crossing this line
  5. If crossing line is from right-to-left; winding-number + 1, otherwise winding-number – 1
  6. If winding-number $\neq 0$ then point interior, else exterior
- But: How to determine directional boundary crossings?
  - (Hint: Using vectors)

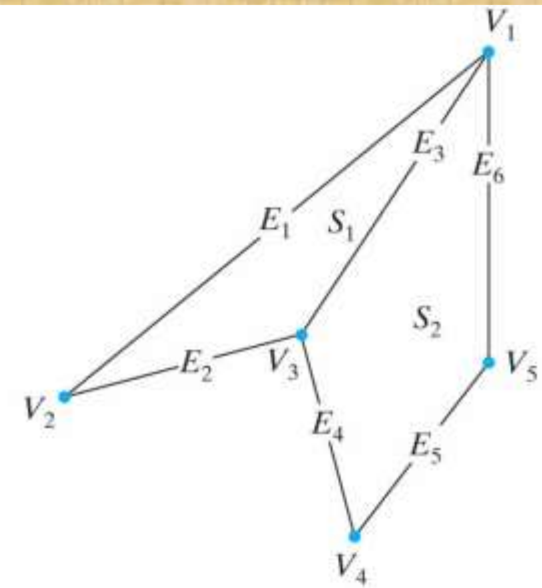
Example: Inside-Outside Tests



Polygon Tables

- Objects in a scene are described as sets of polygon surface facets.
- Data is organized in polygon data tables
  - Geometric data tables
    - Vertex table: Coordinate values for each vertex
    - Edge table: Pointers to vertex table defining each edge in polygon
    - Surface-facet table: Pointers to edge tabel defining each edge for given surface
  - Attribute data tables: Degree of transparency, surface reflectivity, texture characteristics

Geometric data-table



VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

SURFACE-FACET TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

Representation for two adjacent polygon surface facets, formed with six edges and five vertices.

Expanded Edge Table

$$
\begin{aligned}
E_1&: \quad V_1, V_2, S_1\\
E_2&: \quad V_2, V_3, S_1\\
E_3&: \quad V_3, V_1, S_1, S_2\\
E_4&: \quad V_3, V_4, S_2\\
E_5&: \quad V_4, V_5, S_2\\
E_6&: \quad V_5, V_1, S_2
\end{aligned}
$$

For the surfaces of figure in Slide 37 expanded to include pointers into the surface-facet table.

Polygon Tables

- Error checking is easier when using three data tables.
- Error checking includes:
  1. Is every vertex listed as an endpoint for at least two edges?
  2. Is every edge a part of at least one polygon?
  3. Is every polygon closed?
  4. Has each polygon at least one shared edge?
  5. If the edge table contains pointers to polygons, has every edge referenced by a polygon pointer a reciprocal pointer back to the polygon?

Plane Equations

- For many CG applications the spatial orientation of the surface components of objects is needed.
- This information is obtained from vertex coordinate values and the equations that describe the polygon surface.
- General equation for a plane is:
  - $Ax + By + Cz + D = 0$
    - (x, y, z) any point on the plane
    - A, B, C, D plane parameters

Plane Equations: The Parameters

- To find the plane parameters:
  1. Select three successive convex polygon vertices (counterclockwise)
  2. Solve $\left(\frac{A}{D}\right)x_k + \left(\frac{B}{D}\right)y_k + \left(\frac{C}{D}\right)z_k = -1$ (Hint: Using Cramer's rule)
  3. Solve:

$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$
$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$
$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$
$$D = -x_1(y_2 z_3 - y_3 z_2) - x_2(y_3 z_1 - y_1 z_3) - x_3(y_1 z_2 - y_1 z_2)$$

Front and Back Polygon Faces

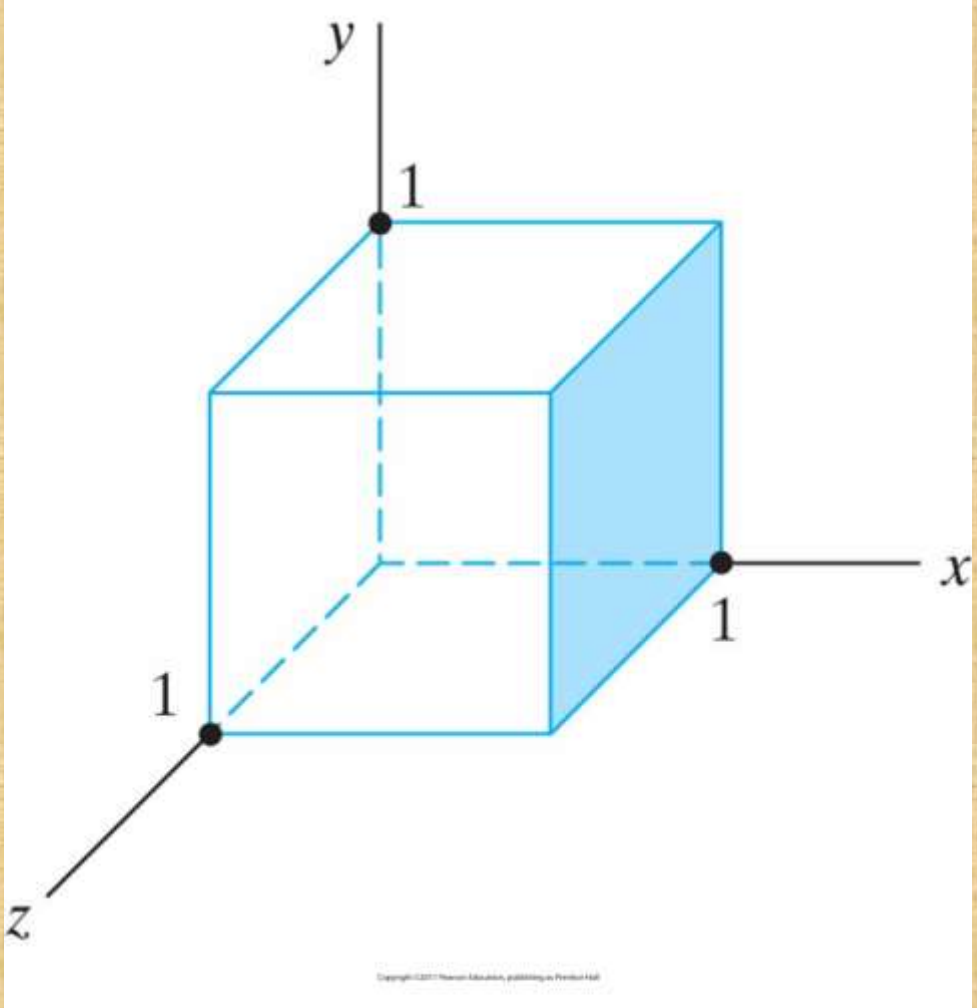
- The sides of a polygon surface have to be distinguished.
- The side of a polygon surface facing into the interior of an object is called **back face**.
- The visible/outward side of a polygon surface is called **front face**.
- Every polygon on a plane partitions the space into two regions.
- Any point that is not on the plane and is visible to the front face of a polygon surface is called **in front of/outside** the plane (and also outside the object).
- Otherwise **behind/inside**.

Where is the Point?

- For any point (x, y, z) not on a plane:
  - $Ax + By + Cz + D \neq 0$

- So if:
  - $Ax + By + Cz + D < 0$, point is **behind** the plane
  - $Ax + By + Cz + D > 0$, point is **in front** of the plane

Example: Point in Relation to Unit Cube



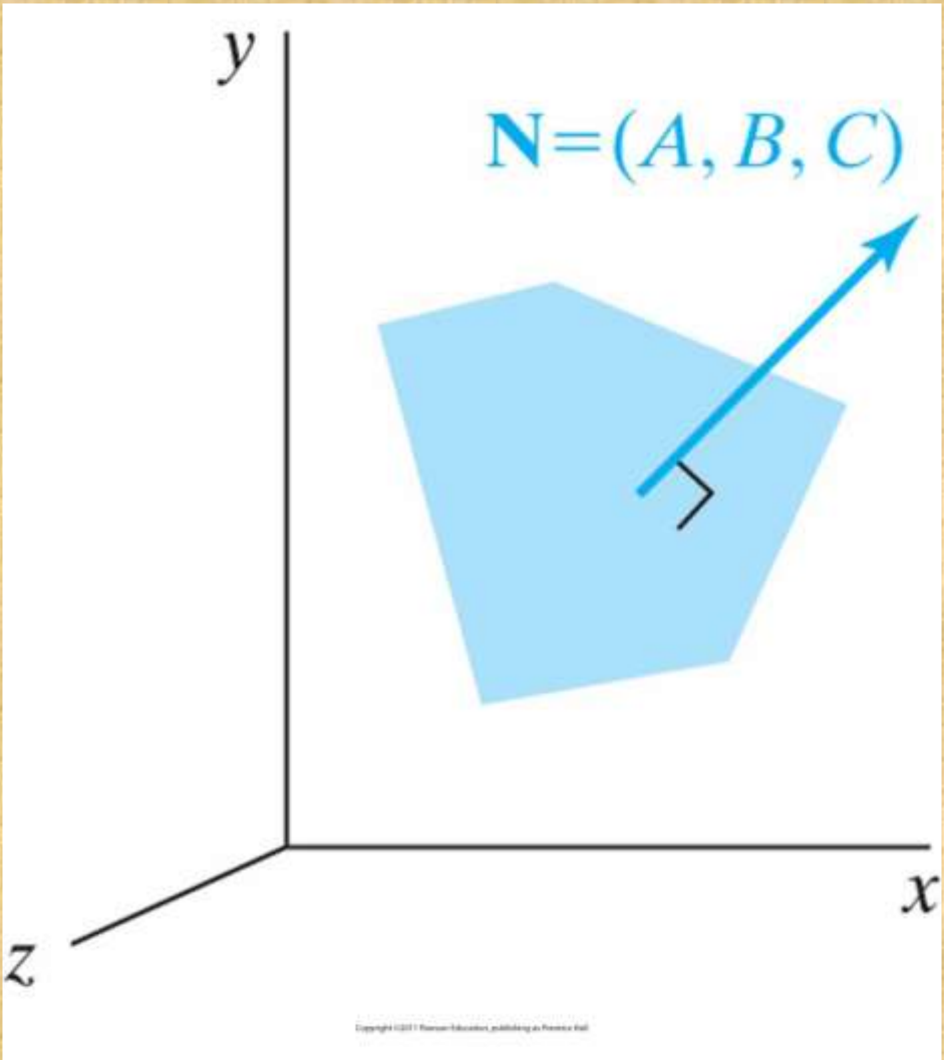
The shaded polygon surface of the unit cube has the plane equation
$x - 1 = 0$

Orientation of a Polygon Surface

- (Surface) Normal vector always points from back face to front face and is perpendicular to the surface, i.e. from inside to outside.

- When using normal vector, the plane equation can be expressed as: $N \cdot P = -D$ (coming soon)

The normal vector N



For a plane described with the equation *Ax* + *By* +*Cz* + *D* = 0 is perpendicular to the plane and has Cartesian components (*A, B, C*)

Calculating the Normal Vector

- Assumption: Convex polygon facet and right-handed Cartesian coordinates
  1. Select three vertex positions V_1, V_2 and V_3 (counterclockwise) from outside the object to inside
  2. Form two vectors from V_1 to V_2 and from V_1 to V_3
  3. Calculate N as vector cross product:

     $N = (V_2 - V_1) \text{ x } (V_3 - V_1)$ (gives plane parameters *A*, *B*, *C*)
  4. Substitute for D (in equations above) and solve

Calculating the Normal Vector

- Assumption: Convex polygon facet and right-handed Cartesian coordinates
    1. Select three vertex positions V_1, V_2 and V_3 (counterclockwise) from outside the object to inside
    2. Form two vectors from V_1 to V_2 and from V_1 to V_3
    3. Calculate N as vector cross product:

       $N = (V_2 - V_1) \text{ x } (V_3 - V_1)$ (gives plane parameters *A, B, C*)
    4. Substitute for D (in equations above) and solve

OpenGL Fill Area Functions

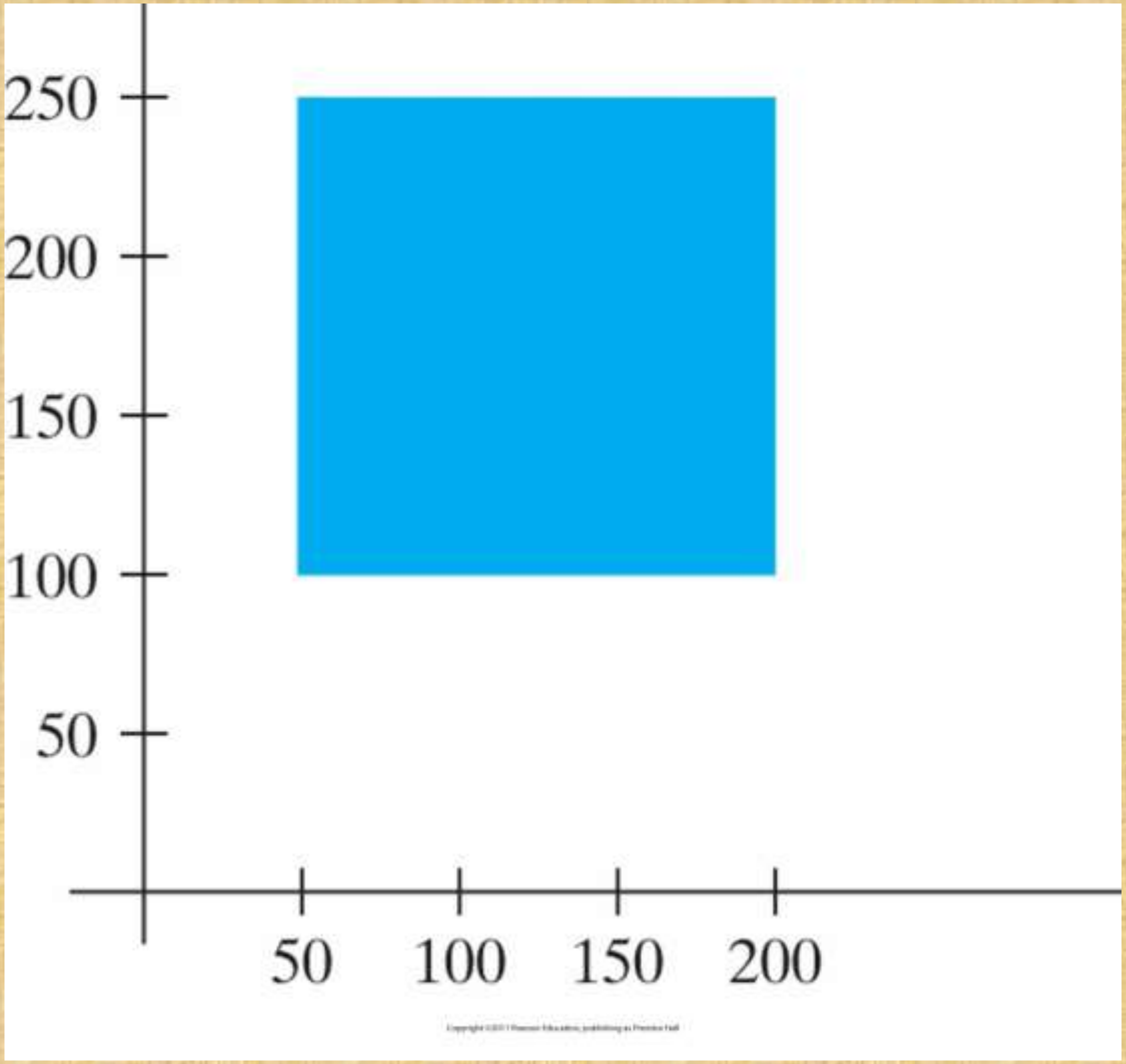
- By default, a polygon interior is displayed in a solid color, determined by the current color settings
- Alternatively, we can fill a polygon with a pattern and we can display polygon edges as line borders around the interior fill
- Polygon vertices are specified counterclockwise.

OpenGL Fill Area Functions (cont.)

- Rectangle
  - glRect*(x1,y1,x2,y2) where * means d, f, i, s, v)

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

Example: Square Fill Area



using the glRect function.

Counterclockwise? Clockwise?

- What happened in previous example?
- Why clockwise?
- Answer: In OpenGL normally always counterclockwise but in general counterclockwise is necessary if back face/front face distinction is important.

OpenGL Fill Area Functions

- GL_POLYGON
  - glBegin(GL_POLYGON);

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

OpenGL Fill Area Functions (cont.)

- Triangle (GL_TRIANGLES or GL_TRIANGLE_STRIP or GL_TRIANGLE_FAN)
- GL_TRIANGLE_STRIP
  - glBegin(GL_TRIANGLES);

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

OpenGL Fill Area Functions (cont.)

- GL_TRIANGLE_STRIP
  - glBegin(GL_TRIANGLE_STRIP);

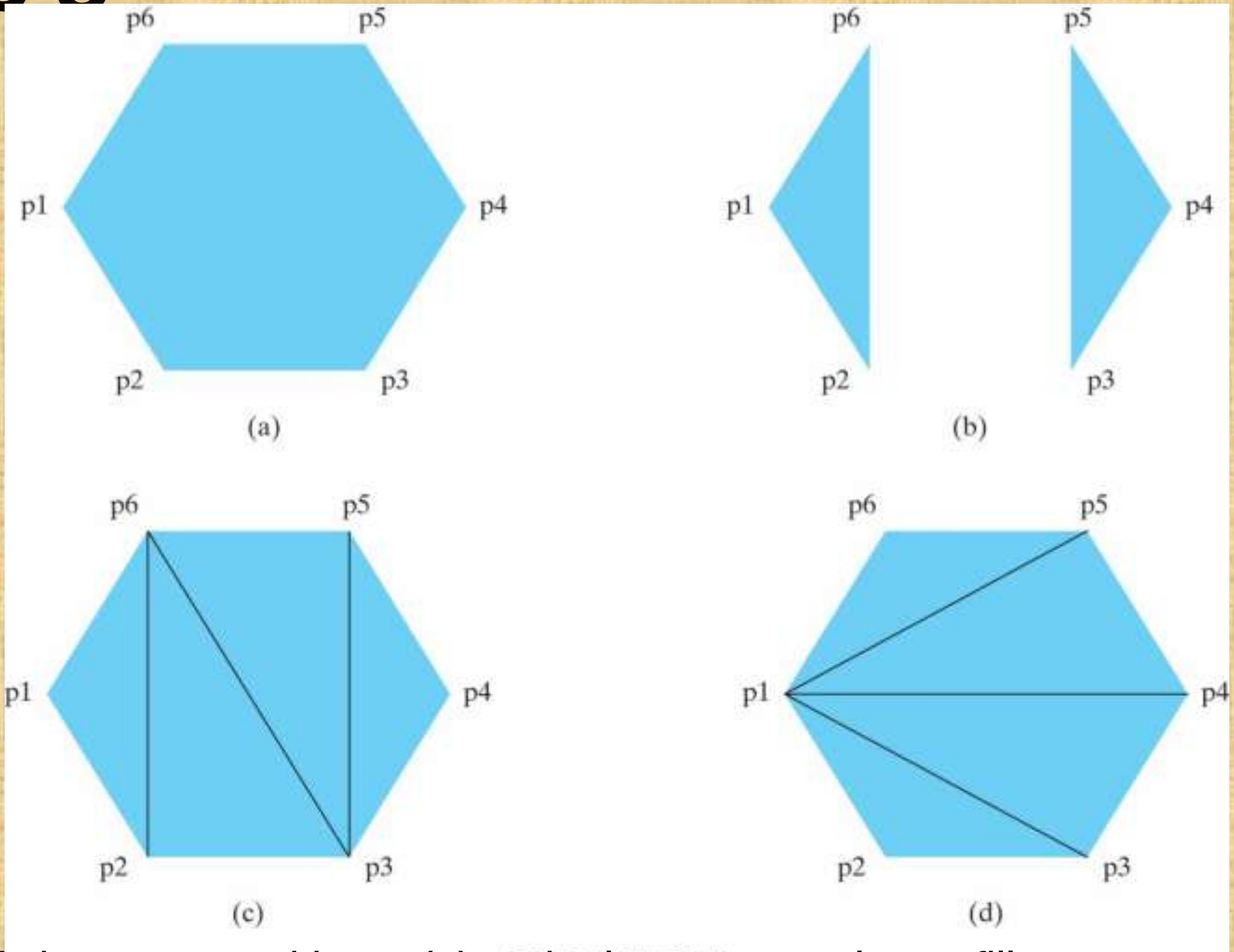
```
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p6);
        glVertex2iv(p3);
        glVertex2iv(p5);
        glVertex2iv(p4);
    glEnd( );
```

OpenGL Fill Area Functions (cont.)

- GL_TRIANGLE_FAN
  - glBegin(GL_TRIANGLE_FAN);

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

Polygon Fill Areas



Using a list of six vertex positions. (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.

OpenGL Fill Area Functions

- Quads (GL_QUADS or GL_QUAD_STRIP)
- GL_QUADS
  - glBegin(GL_QUADS);

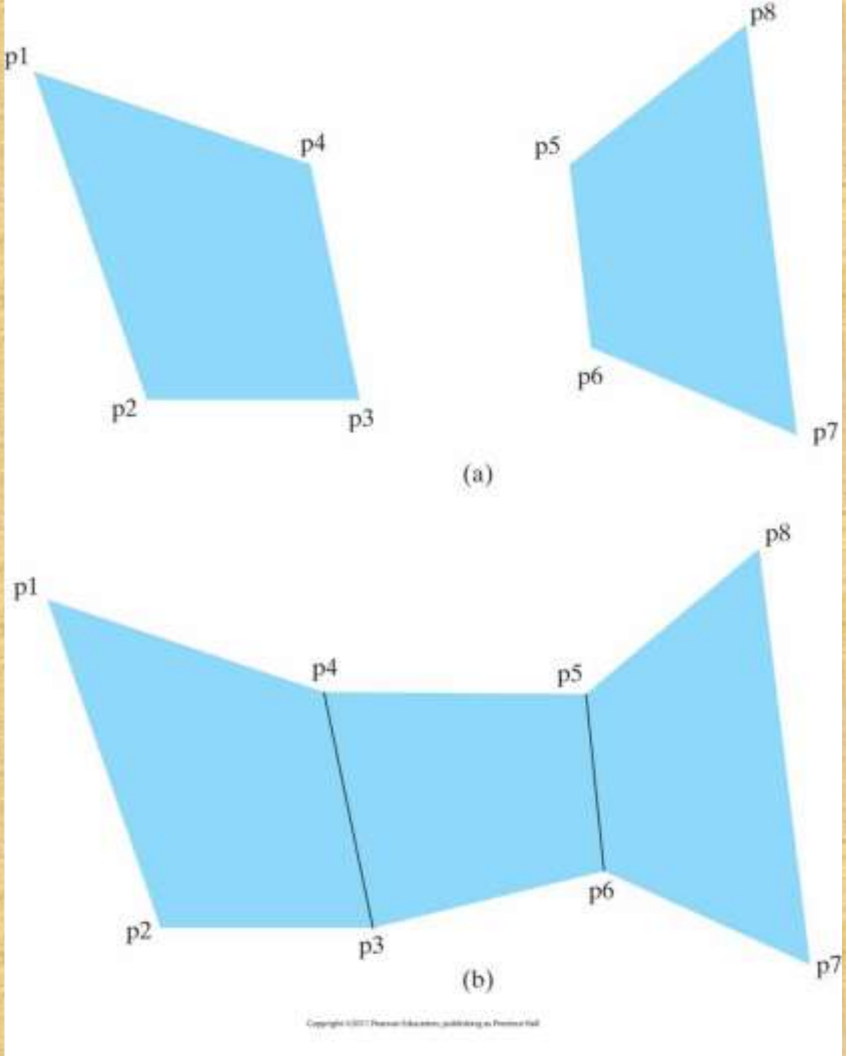
```
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p3);
        glVertex2iv(p4);
        glVertex2iv(p5);
        glVertex2iv(p6);
        glVertex2iv(p7);
        glVertex2iv(p8);
    glEnd( );
```

OpenGL Fill Area Functions (cont.)

- GL_QUAD_STRIP
  - glBegin(GL_QUADS);

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

Quadrilateral Fill Areas



Using a list of eight vertex positions. (a) Two unconnected quadrilaterals generated with GL_QUADS. (b) Three connected quadrilaterals generated with GL_QUAD_STRIP.

How many objects?

- Assumption: Number of vertices = N

- Triangles: int (N / 3) (N ≥ 3)
- Triangles in strip: N-2 (N ≥ 3)
- Triangles in fan: N-2 (N ≥ 3)

- Quads: int (N / 4) (N ≥ 4)

Processing Order

- Assumption: Position in vertex list = n
  - $(n = 1, n = 2, \ldots, n = N-2)$

- Triangles: Nothing special
- Triangles in strip:
  - If n odd: $n, n + 1, n + 2$
  - If n even: $n + 1, n, n + 2$
- Triangles in fan: $1, n + 1, n + 2$

- Quads in strip: $2n - 1, 2n, 2n + 2, 2n + 1$

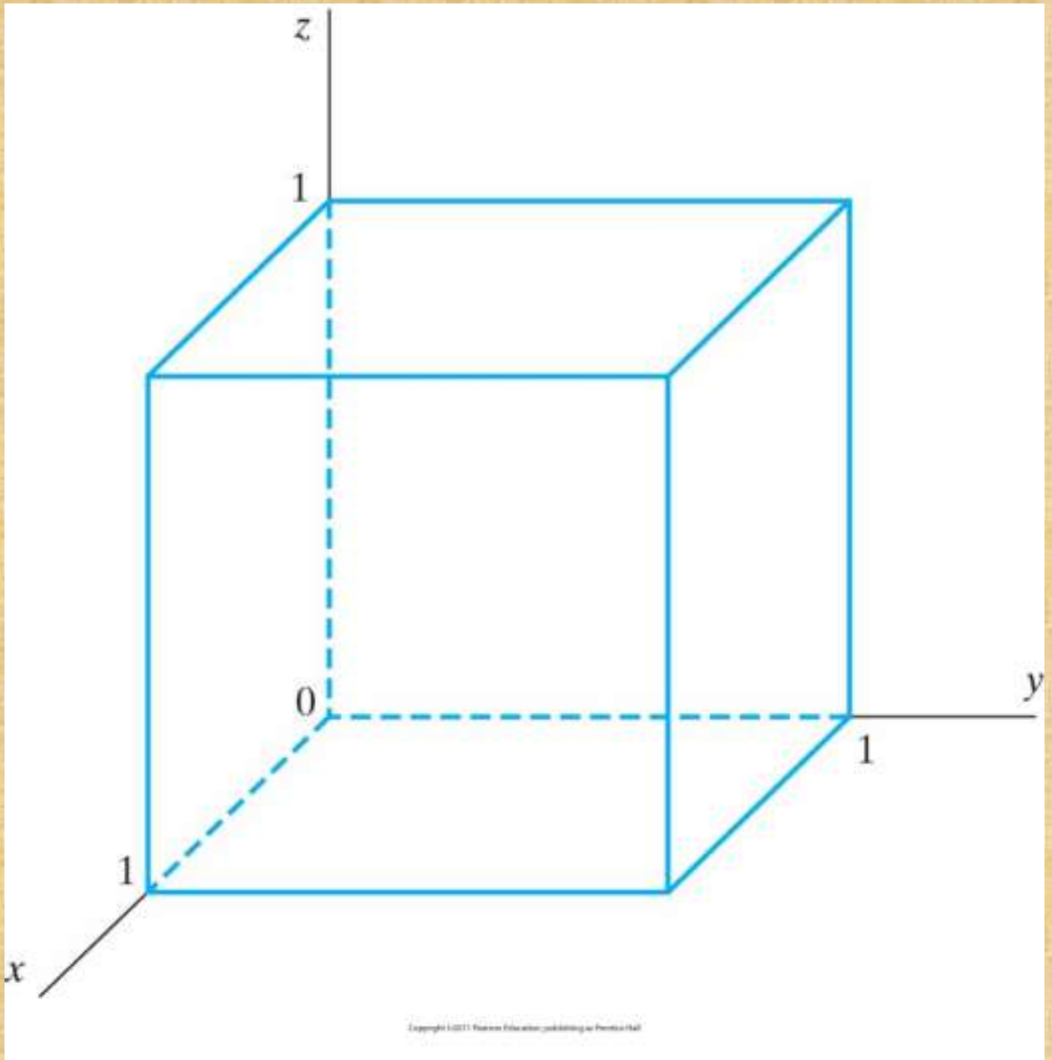
Vertex Arrays

- We can store a list of points:

```
int pt[8][3] = {{0,0,0},{0,1,0},{1,0,0},{1,1,0},
                {0,0,1},{0,1,1},{1,0,1},{1,1,1}};
```

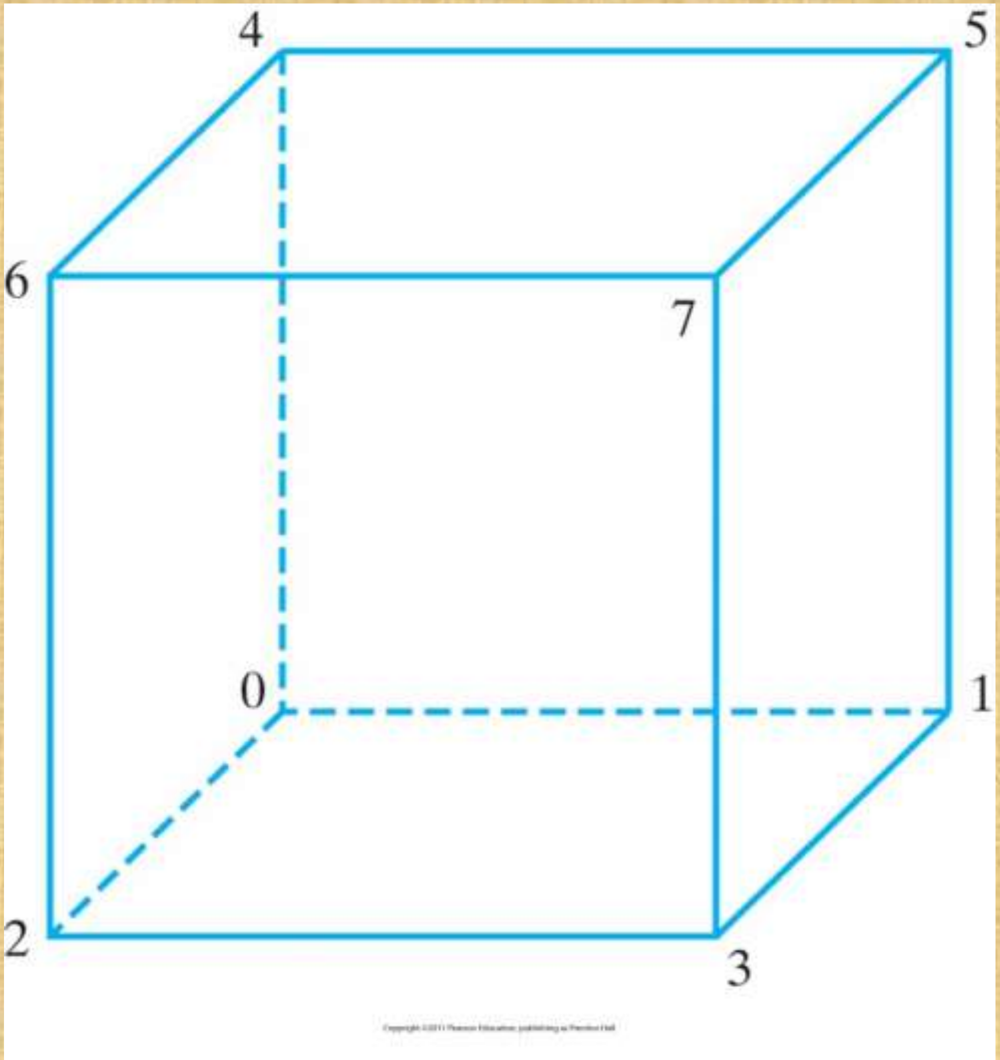
- Above could be used for a cube.
- To plot faces can make calls beginning with either glBegin(GL_POLYGON) or glBegin(GL_QUADS)

Example: Cube



with an edge length of 1

Example: Cube (cont.)



Subscript values for array pt corresponding to the vertex coordinates for the cube shown in Slide 64.

Vertex Arrays (cont.)

```
void quad(int p1, int p2, int p3, int p4) {
    glBegin(GL_QUADS);
     glVertex3i( pt[p1][0], pt[p1][1], pt[p1][2] );
     glVertex3i( pt[p2][0], pt[p2][1], pt[p2][2] );
     glVertex3i( pt[p3][0], pt[p3][1], pt[p3][2] );
     glVertex3i( pt[p4][0], pt[p4][1], pt[p4][2] );
    glEnd();
}
```

```
void cube() {
    quad(6,2,3,7);
    quad(5,1,0,4);
    quad(7,3,1,5);
    quad(4,0,2,6);
    quad(2,0,1,3);
    quad(7,5,4,6);
}
```

Too many function calls!

Vertex Arrays (cont.)

- Use vertex arrays!

- General procedure:

1. Activate vertex array feature
2. Specify location and data for vertex coordinates
3. Process multiple primitives with few calls

Vertex Arrays (cont.)

```
glEnableClientState(GL_VERTEX_ARRAY);  (1)
glVertexPointer(3,GL_INT,0,pt);  (2)
GLubyte vertIndex[] = {6,2,3,7, 5,1,0,4, 7,3,1,5, 4,0,2,6,
   2,0,1,3, 7,5,4,6};  (vertices for cube)
glDrawElements(GL_QUADS, 24,
   GL_UNSIGNED_BYTE,vertIndex);
```

- Vertex arrays can be disabled with

```
glDisableClientState(GL_VERTEX_ARRAY);  (3)
```

OpenGL Output Primitives

- Next slides give a summary of OpenGL output primitive functions and related routines (incl. Pixel-array primitives and Character primitives)
- (See also HB p. 102-117)

Table 4.1

TABLE 4-1

Summary of OpenGL Output Primitive Functions and Related Routines

Function	Description
gluOrtho2D	Specifies a two-dimensional world-coordinate reference.
glVertex*	Selects a coordinate position. This function must be placed within a glBegin/glEnd pair.
glBegin (GL_POINTS);	Plots one or more point positions, each specified in a glVertex function. The list of positions is then closed with a glEnd statement.
glBegin (GL_LINES);	Displays a set of straight-line segments, whose endpoint coordinates are specified in glVertex functions. The list of endpoints is then closed with a glEnd statement.
glBegin (GL_LINE_STRIP);	Displays a polyline, specified using the same structure as GL_LINES.
glBegin (GL_LINE_LOOP);	Displays a closed polyline, specified using the same structure as GL_LINES.
glRect*	Displays a fill rectangle in the *xy* plane.

Table 4-1 (cont.)

`glBegin (GL_POLYGON);`	Displays a fill polygon, whose vertices are given in `glVertex` functions and terminated with a `glEnd` statement.
`glBegin (GL_TRIANGLES);`	Displays a set of fill triangles using the same structure as `GL_POLYGON`.
`glBegin (GL_TRIANGLE_STRIP);`	Displays a fill-triangle mesh, specified using the same structure as `GL_POLYGON`.
`glBegin (GL_TRIANGLE_FAN);`	Displays a fill-triangle mesh in a fan shape with all triangles connected to the first vertex, specified with same structure as `GL_POLYGON`.
`glBegin (GL_QUADS);`	Displays a set of fill quadrilaterals, specified with the same structure as `GL_POLYGON`.
`glBegin (GL_QUAD_STRIP);`	Displays a fill-quadrilateral mesh, specified with the same structure as `GL_POLYGON`.
`glEnableClientState (GL_VERTEX_ARRAY);`	Activates vertex-array features of OpenGL.
`glVertexPointer (size, type, stride, array);`	Specifies an array of coordinate values.
`glDrawElements (prim, num, type, array);`	Displays a specified primitive type from array data.

Table 4-1 (cont.)

TABLE 4-1

(continued)

Function	Description
`glNewList (listID, listMode)`	Defines a set of commands as a display list, terminated with a `glEndList` statement.
`glGenLists`	Generates one or more display-list identifiers.
`glIsList`	Queries OpenGL to determine whether a display-list identifier is in use.
`glCallList`	Executes a single display list.
`glListBase`	Specifies an offset value for an array of display-list identifiers.
`glCallLists`	Executes multiple display lists.
`glDeleteLists`	Eliminates a specified sequence of display lists.
`glRasterPos*`	Specifies a two-dimensional or three-dimensional current position for the frame buffer. This position is used as a reference for bitmap and pixmap patterns.

Table 4-1 (cont.)

`glBitmap (w, h, x0, y0, xShift, yShift, pattern);`	Specifies a binary pattern that is to be mapped to pixel positions relative to the current position.
`glDrawPixels (w, h, type, format, pattern);`	Specifies a color pattern that is to be mapped to pixel positions relative to the current position.
`glDrawBuffer`	Selects one or more buffers for storing a pixmap.
`glReadPixels`	Saves a block of pixels in a selected array.
`glCopyPixels`	Copies a block of pixels from one buffer position to another.
`glLogicOp`	Selects a logical operation for combining two pixel arrays, after enabling with the constant `GL_COLOR_LOGIC_OP`.
`glutBitmapCharacter (font, char);`	Specifies a font and a bitmap character for display.
`glutStrokeCharacter (font, char);`	Specifies a font and an outline character for display.
`glutReshapeFunc`	Specifies actions to be taken when display-window dimensions are changed.

Next Lecture

Attributes of Graphics Primitives

References

- Donald Hearn, M. Pauline Baker, Warren R. Carithers, “Computer Graphics with OpenGL, 4th Edition”; Pearson, 2011
- Sumanta Guha, “Computer Graphics Through OpenGL: From Theory to Experiments”, CRC Press, 2010
- Richard S. Wright, Nicholas Haemel, Graham Sellers, Benjamin Lipchak, “OpenGL SuperBible: Comprehensive Tutorial and Reference”, Addison-Wesley, 2010
- Edward Angel, “Interactive Computer Graphics. A Top-Down Approach Using OpenGL”, Addison-Wesley, 2005