Computer Graphics

Chapter 4

Graphics Output Primitives

(Part II)

Outline

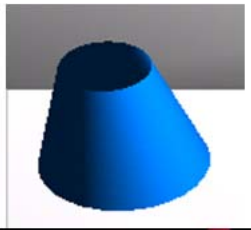
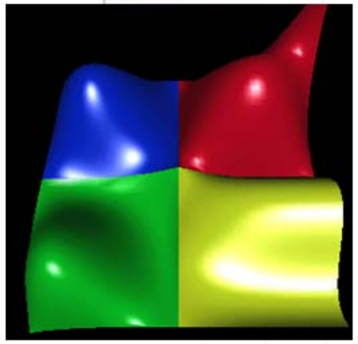
- OpenGL Curve Functions
- Fill-Area Primitives
- Polygon Fill Areas
- OpenGL Polygon Fill-area Functions
- Pixel Array Primitives
- Character Primitives
- OpenGL functions
- OpenGL Display Lists

Chapter 4
Graphics Output Primitives (Part II)

OpenGL Curve Functions

Fill-Area Primitives and Polygon Fill Areas

OpenGL Curve Functions

(By Mark Kilgard)

- **GLU** (OpenGL Utility) functions:

  3D Quadrics (Spheres, Cylinders)

  Rational B-Splines (circles, ellipse, Bezier curve)

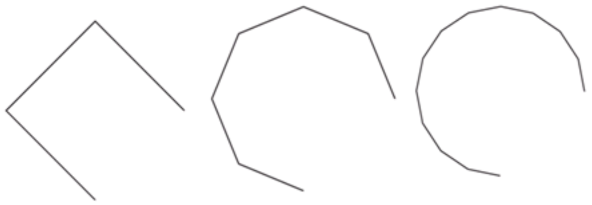
  (有理B樣條) gluSphere/gluCylinder/gluDisk…

  NURBS (non-uniform rational B-splines)

- **GLUT** (OpenGL Utility Toolkit) functions:

  3D Quadrics (Spheres, Cones etc.) glutSolidSphere/glutWireSphere, glutSolidCone, glutSolidTorus, …

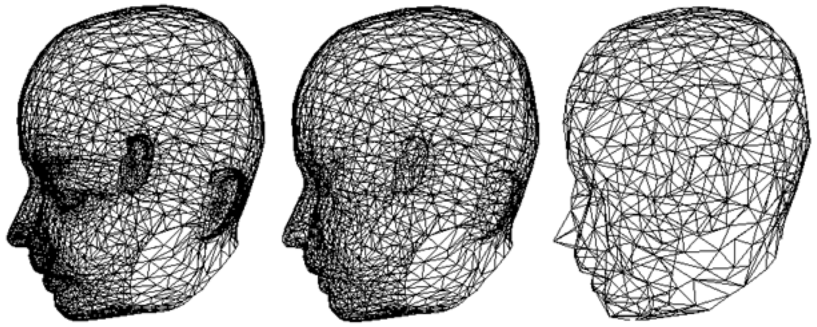
- Approximate a simple curve using a polyline
  - The more line sections, the smoother appearance of the curve.



- Write your own curve-generation algorithms

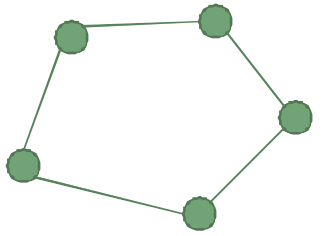
Fill-Area Primitives

- **Fill (Filled) Area**
  - An area filled with some solid color or pattern
  - To describe the surfaces of solid objects
  - Most graphics library routines don't support arbitrary fill shapes.
    - A fill area is required to be specified as a **polygon**.
- A set of polygon facets can approximate a curved surface (a polygon mesh), which can be called as **Surface Tessellation**.

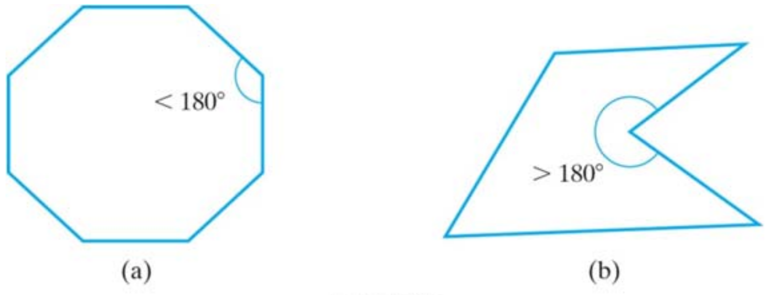


Polygon Fill Areas

- **Polygon**
  - A **plane** figure specified by a set of three or more vertices, that are connected in sequence by straight-line segments (edges). Here refer only to those without crossing edges: simple (standard) polygon
- **Polygon Classifications**
  - **Convex** Polygon (凸)
    - All interior angles are less than or equal to 180° degrees
  - **Concave** Polygon (凹)
    - Otherwise



The **interior angle** is the angle inside the polygon boundary that is formed by two adjacent edges.



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

Polygon Fill Areas

- Implementation consideration
  - Some graphics packages including OpenGL, only support **convex** polygon for the fill algorithms.

Generate a line segment

Overlapping edges, or edges with 0 length

  - **For degenerate (退化)** polygons

    -- A set of vertices that are collinear (or that have repeated vertex positions)

    -- To identify these cases, graphics systems usually leave these to the programmer

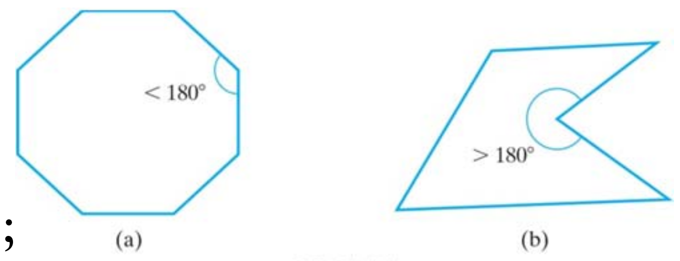
  - **For concave** polygons

    -- Implementation of fill algorithms and other graphics routines are more complicated for concave polygons

    -- To split a concave polygon into a set of convex polygons

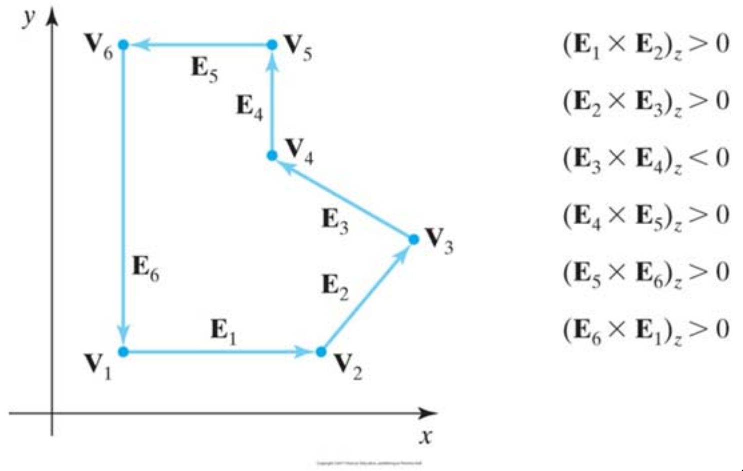
Identifying Concave Polygons

- Characteristics
  - At least one interior angle >180°;
  - Extension of some edges intersect other;
  - Line segment of some pair of interior points intersects the boundary.

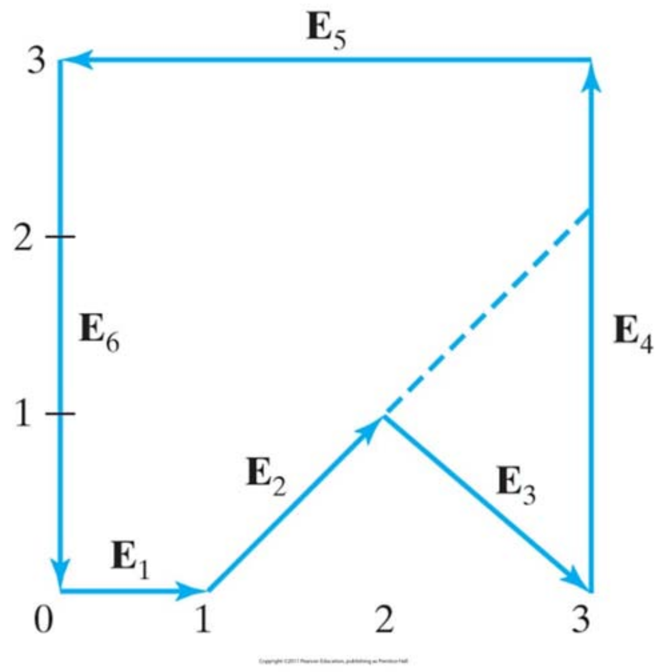


- Mathematically
  - The **cross products** of adjacent edges
  - Convex: the same sign
  - Concave: some are positive and some are negative

    (next slide for the detail)



Splitting Concave Polygons



Splitting a concave polygon using the vector method

A concave polygon with six edges.

Edge vectors for this polygon:

$$E_1 = (1,0,0) \quad E_2 = (1,1,0)$$
$$E_3 = (1,-1,0) \quad E_4 = (0,3,0)$$
$$E_5 = (-3,0,0) \quad E_6 = (0,-3,0)$$

The cross product for two adjacent edge vectors:

$$E_1 \times E_2 = (0,0,1) \quad \boxed{E_2 \times E_3 = (0,0,-2)}$$
$$E_3 \times E_4 = (0,0,3) \quad E_4 \times E_5 = (0,0,9)$$
$$E_5 \times E_6 = (0,0,9) \quad E_6 \times E_1 = (0,0,3)$$

Split the polygon along the line of vector $\mathbf{E}_2$. The two new polygons are both convex.

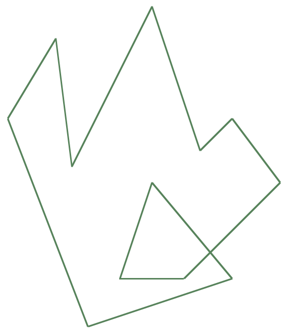
Determinant form

$$\mathbf{a} \times \mathbf{b} = \det \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix}.$$

$$\mathbf{i}a_2b_3 + \mathbf{j}a_3b_1 + \mathbf{k}a_1b_2 - \mathbf{i}a_3b_2 - \mathbf{j}a_1b_3 - \mathbf{k}a_2b_1.$$

Inside-Outside Tests

- **Inside-Outside test**
  - Area-filling algorithms and other graphics processes often need to identify interior regions of objects.
  - For simple object, it is a straightforward process.



  - For complex objects, graphics packages normally use either:
    1. Odd-Even (奇偶) rule (Odd-Parity rule or Even-Odd rule)
    2. Non-zero winding (環繞) number rule

Inside-Outside Tests

Odd-Even rule (Odd-Parity Rule, Even-Odd Rule):

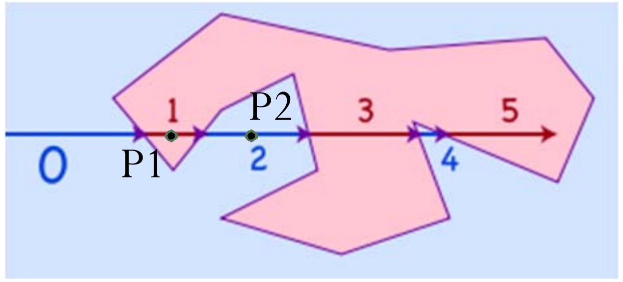
1. Draw a **line**: (the line path doesn't intersect any line-segment endpoints)

   From any position P to a distant point outside the coordinate extents of the polygon
2. Counting the number of edge crossing along the line.
3. If the number of polygon edges crossed by this line is **odd** (奇) then

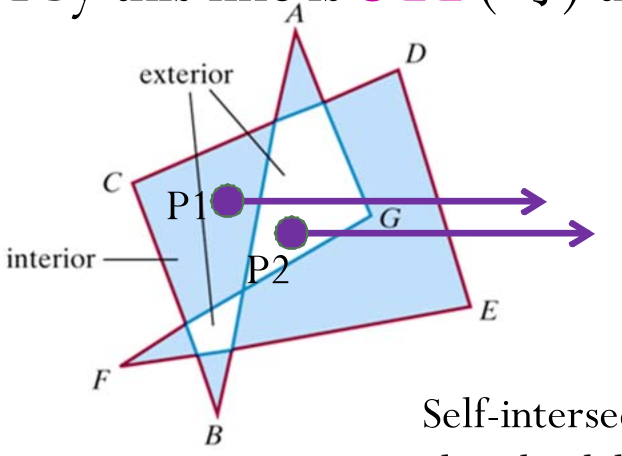
   P is an **interior** point.

   Else

   P is an **exterior** point



(From Wiki)



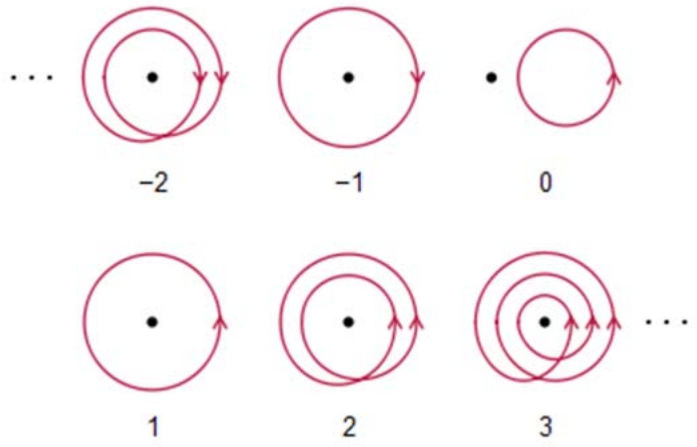
Odd-Even Rule

Self-intersecting closed polyline

Inside-Outside Tests

Non-zero Winding (環繞) Number Rule :

- Counts the **winding number** of a point
  - Non-zero: interior points
  - Zero: exterior points
- Winding number of a closed curve around a given point:
  - The number of times the curve (**polygon edges)** wind **counterclockwise** around that point.



(Pic. From Wiki)

Inside-Outside Tests

Non-zero Winding Number Rule (cont.):

1. Set up vectors along the edges and initialing the winding number to 0.
2. Imagine a line drawn from any position P to a distant point beyond the coordinate extents of the object.
3. Count the number of edges that cross the line in each direction.

   Add 1 to the winding number:

   A polygon edge crosses the line from right to left ($\longleftarrow$).

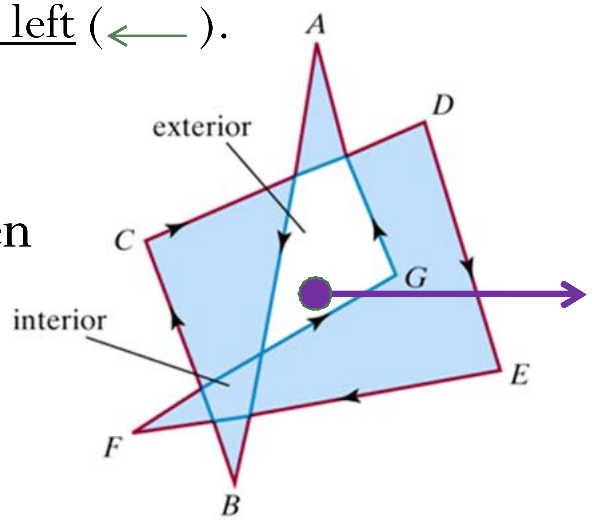
   Subtract 1:

   An edge crosses from left to right ($\longrightarrow$).
4. If the winding number is **non-zero**, then

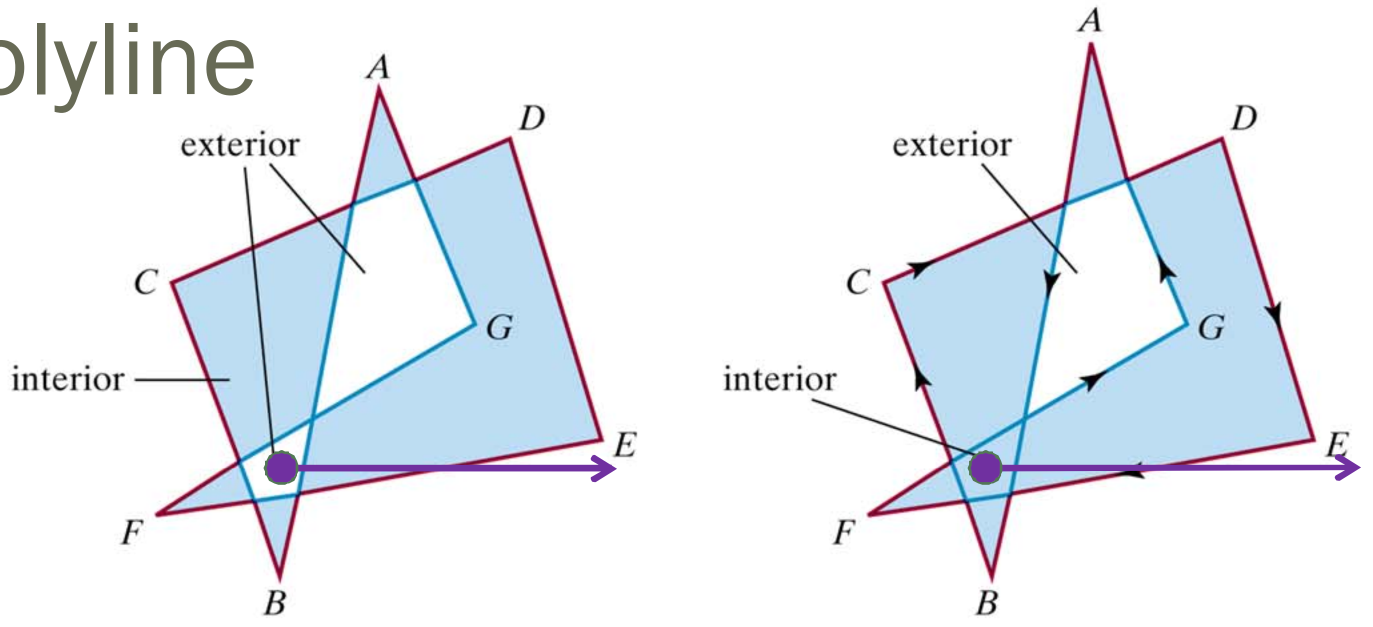
   P is defined to be an **interior** point

   Else

   P is taken to be an **exterior** point.



Example: self-intersecting closed polyline



Odd-Even Rule

(a)

Nonzero Winding-Number Rule

(b)

$\leftarrow$: +1

$\rightarrow$: -1

Figure 4-12

Identifying interior and exterior regions of a closed polyline that contains self-intersecting segments.

Odd: interior; even: exterior.

Non-zero: interior; zero: exterior.

Polygon Representation

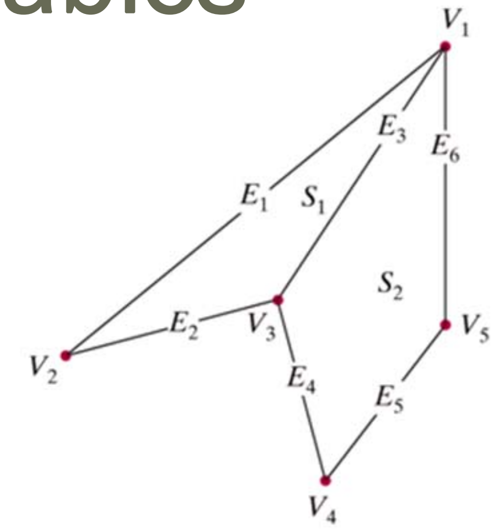
- Polygon Tables **(Fig. 4-16)**
  - Typically, the objects in a scene are described as sets of polygon surface facets with information:
    - Coordinate, color, transparency, lighting properties…
  - In systems, the information for polygons facets are represented by tables – polygon data tables:
    - Geometric Tables

      Vertex Table; Edge Table; Surface-Facet Table

    - Attribute Tables

      Normal; Transparency; Reflectance; Texture Coordinates

Polygon Tables



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

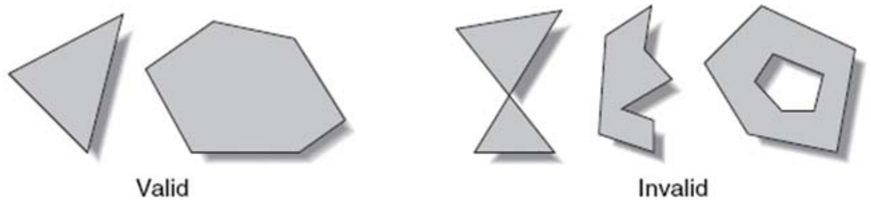
Geometric data-table representation for two adjacent polygon surface facets, formed with six edges and five vertices

Chapter 4
Graphics Output Primitives (Part II)

OpenGL Polygon Fill-Area Functions

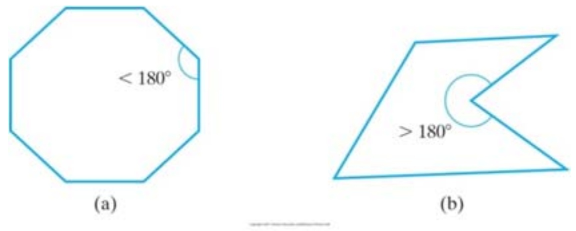
OpenGL Polygon Fill-area Functions

- In OpenGL, a fill area must be specified as a **convex** polygon
  - A vertex list for a fill polygon must contain at least **three** vertices;
  - **No crossing** edges;



(From: OpenGL Programming Guide)

  - All interior angles for the polygon must be **less than 180°**.



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

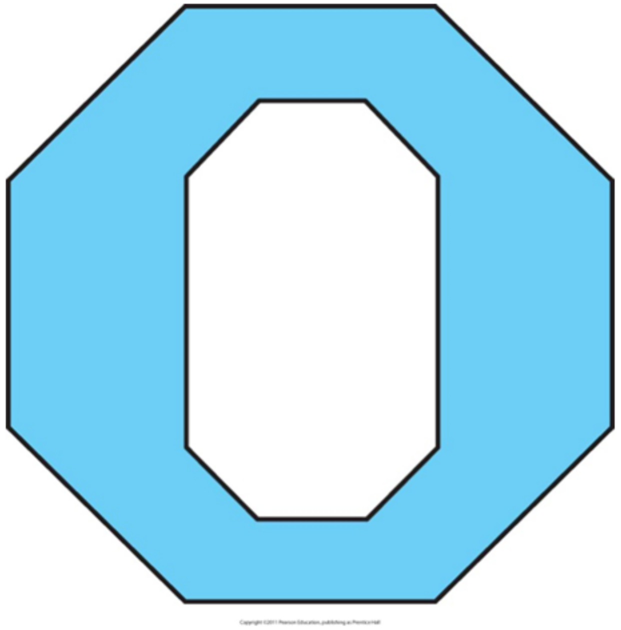
Convex:

❖ all interior angles are less than or equal to 180°.

Concave:

❖ Otherwise.

OpenGL Polygon Fill-area Functions


A polygon with a complex interior, which cannot be specified with a single vertex list.

The figure on the left has a hole in it.

How do you show such polygon?

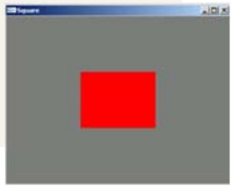
- One vertex list for only one polygon-filled area
- You can define it by two overlapping convex polygons.

OpenGL Polygon Fill-area Functions

- In OpenGL, specifying fill polygons are similar to those for describing a point or polyline.

```
glBegin(SYMBOLIC_CONSTANT)
    glVertex*(…);
    glVertex*(…);
    …
glEnd();
```

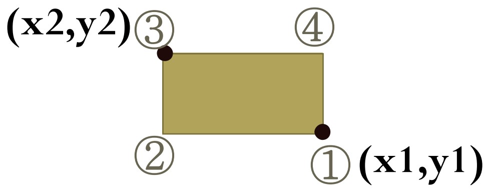
```
//set red color
glColor3f(1.0, 0.0, 0.0);
//specify 2D square
 glBegin(GL_QUADS);
    glVertex2i(-2, 2);
    glVertex2i(2, 2);
    glVertex2i(2, -2);
    glVertex2i(-2, -2);
glEnd();
```



- By default, a polygon interior is displayed in a **solid** color, determined by the current color settings

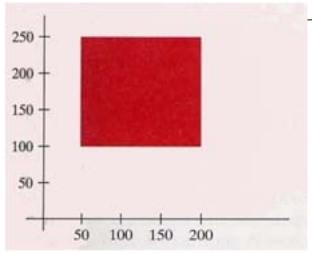
Define Rectangle in OpenGL

- **A special rectangle function of OpenGL**



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

'*': the coordinate data type: i (integer), s (short), f (float), d (double), and v (vector)



glRecti (200, 100, 50, 250);

int vertex1 [ ] = { 200, 100 };

int vertex2 [ ] = { 50, 250 };

glRectv (vertex1, vertex2);

This function is equivalent to :

```
glBegin (GL_POLYGON);
        glVertex2* (x1, y1);
        glVertex2* (x2, y1);
        glVertex2* (x2, y2);
        glVertex2* (x1, y2);
glEnd ();
```

and

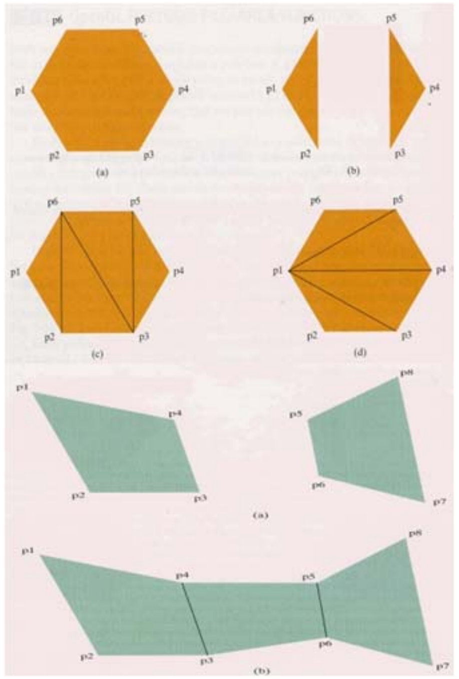
```
glBegin (GL_QUADS);
        glVertex2* (x1, y1);
        glVertex2* (x2, y1);
        glVertex2* (x2, y2);
        glVertex2* (x1, y2);
glEnd ();
```

glRect* is more efficient than using the above glVertex specifications

Six OpenGL Polygon Fill Primitives

To use the symbolic constant in the glBegin function, along with a list of glVertex commands.

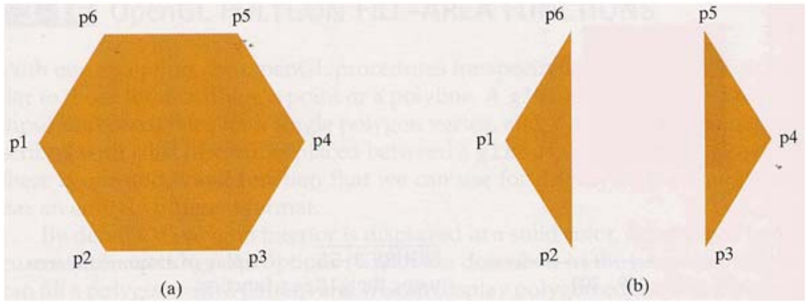
- GL_POLYGON -- closed ploygon
- GL_TRIANGLES -- disconnected triangles
- GL_TRIANGLE_STRIP -- connected triangles
- GL_TRIANGLE_FAN -- triangles sharing common point
- GL_QUADS -- disconnected quadrilaterals
- GL_QUAD_STRIP -- connected quadrilaterals



Same vertices in different order, and with different symbolic constant

OpenGL Polygon Fill-area Functions

- GL_POLYGON and GL_TRIANGLES



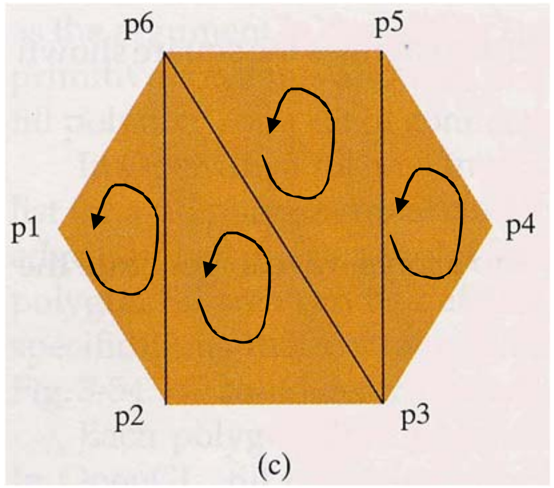
```
glBegin (GL_POLYGON);
   glVertex2iv (p1);
   glVertex2iv (p2);
   glVertex2iv (p3);
   glVertex2iv (p4);
   glVertex2iv (p5);
   glVertex2iv (p6);
glEnd ();
```

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

The orders of the vertices in (a) and (b) between the glBegin() and glEnd() pair are different.

OpenGL Polygon Fill-area Functions

- GL_TRIANGLE_STRIP

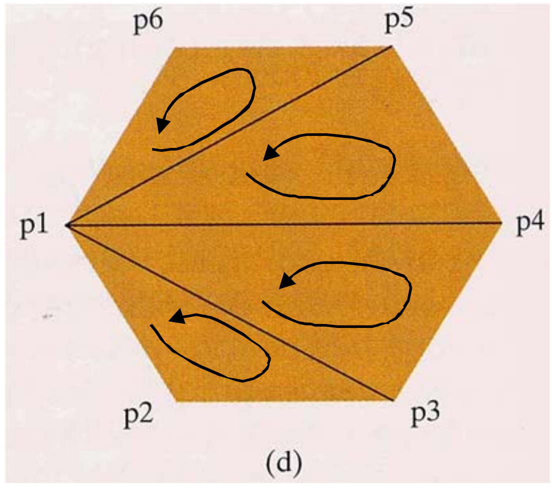


```
glBegin (GL_TRIANGLE_STRIP);
    glVertex2iv (p1);  -> n=1
    glVertex2iv (p2);  -> n=2
    glVertex2iv (p6);  -> n=3
    glVertex2iv (p3);  -> n=4
    glVertex2iv (p5);
    glVertex2iv (p4);
glEnd ();
```

- For N vertices, we obtain N-2 triangles. Each successive triangle shares an edge with the previously defined triangle.
- The **first three points** form the first triangle (**counterclockwise** viewing from the outside), points 2-4 form the second, points 3-5 form the third, and so on.
- The ordering of the vertex list is important to ensure a consistent display.
  - Define each position n in the vertex list in the order 1, 2, …, N-2.
    - If n is odd, the triangle vertices are in the order: n, n+1, n+2; -> (p1, p2, p6)
    - If n is even, the triangle vertices are in the order: n+1, n, n+2. -> (p6, p2, p3)

OpenGL Polygon Fill-area Functions

- GL_TRIANGLE_FAN

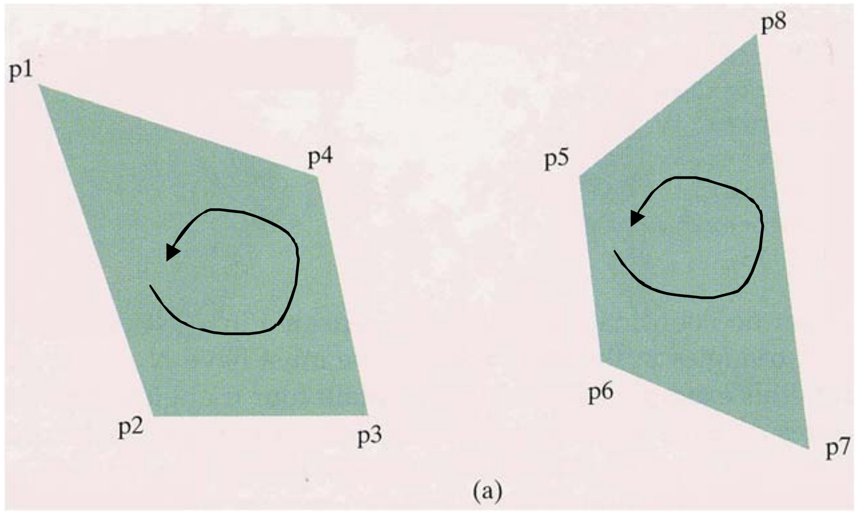


```
glBegin (GL_TRIANGLE_FAN);
    glVertex2iv (p1);  -> n=1
    glVertex2iv (p2);  -> n=2
    glVertex2iv (p3);  -> n=3
    glVertex2iv (p4);  -> n=4
    glVertex2iv (p5);
    glVertex2iv (p6);
glEnd ();
```

- ➢ For N vertices, we obtain N-2 triangles.
- ➢ The first point is shared by every triangle. Points 1&2&3 define the first one, points 1&3&4 define the second, and so on.
- ➢ Define each position n in the vertex list in the order 1, 2, …, N-2.
  - ➢ Vertices1, n+1, n+2 define nth triangle; -> (p1, p2, p3); (p1, p3, p4); …

OpenGL Polygon Fill-area Functions

- GL_QUADS

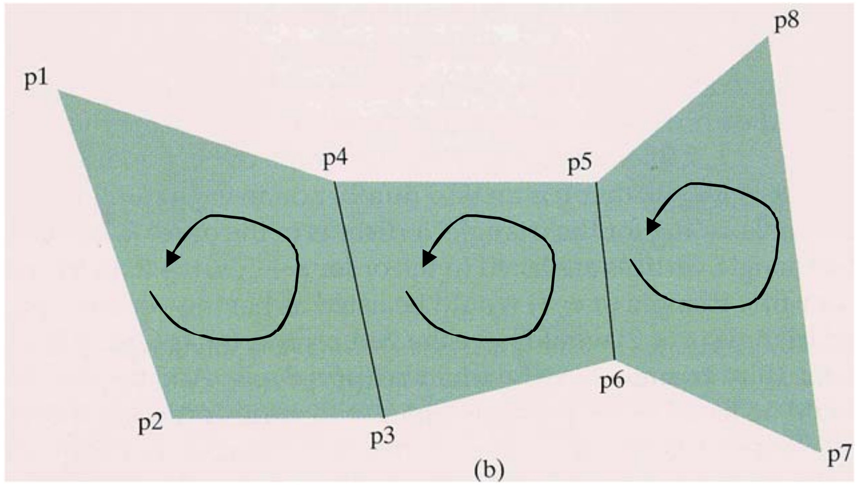


```
glBegin (GL_QUADS);
   glVertex2iv (p1);
   glVertex2iv (p2);
   glVertex2iv (p3);
   glVertex2iv (p4);
   glVertex2iv (p5);
   glVertex2iv (p6);
   glVertex2iv (p7);
   glVertex2iv (p8);
glEnd ();
```

- ➢ The first four points form a quadrilateral, the next four points form the second, and so on.
- ➢ At least four points should be listed, otherwise, nothing is displayed.

OpenGL Polygon Fill-area Functions

- GL_QUAD_STRIP



```
glBegin (GL_QUAD_STRIP);
   glVertex2iv (p1); -> n=1
   glVertex2iv (p2); -> n=2
   glVertex2iv (p4); -> n=3
   glVertex2iv (p3);
   glVertex2iv (p5);
   glVertex2iv (p6);
   glVertex2iv (p8);
   glVertex2iv (p7);
glEnd ();
```

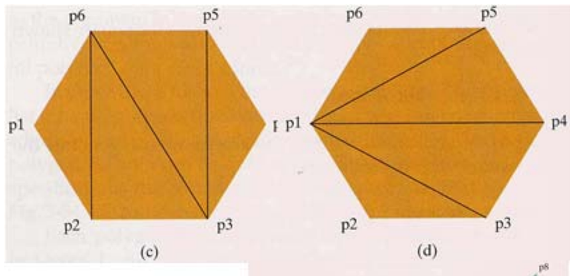
➢ Rearrange the vertex list, we can obtain the set of connected quadrilaterals.

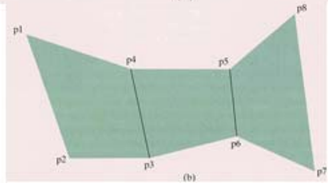
➢ Define each position n in the vertex list in the order n=1, n=2, …, n=(N/2)-1.

➢ The vertices 2n-1, 2n, 2n+2, 2n+1 define nth quadrilateral -> (p1, p2, p3, p4); (p4, p3, p6, p5)

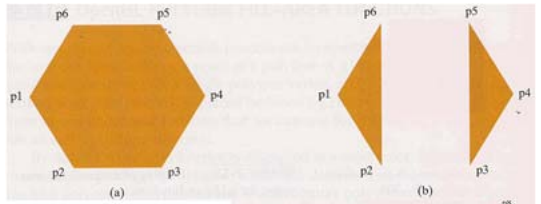
OpenGL Polygon Fill-area Functions

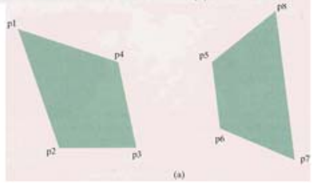
- Common use
  - GL_TRIANGLE_STRIP & GL_QUAD_STRIP
    - Easy to create solid surfaces.
  - GL_TRIANGLE_FAN
    - Conical shapes: a cone is hard with strips, easy with a triangle fan.





- Not common use
  - GL_TRIANGLES & GL_QUADS are **rare**
    - Gaps in solid objects are an easy mistake.
    - Exception: as output format of modeling programs.
  - GL_POLYGON is **fairly rare**
    - Wrongly generate non-planar/non-convex polygons.





Spatial Orientation of Polygon Faces

- Spatial Orientation of a polygon face
  - The vertex coordinates values
  - The equations of the polygon surfaces
- The general equation of a plane containing a polygon is

  $Ax + By + Cz + D = 0$ (4-1)

  (x,y, z) is any point on the plane;

  A, *B*, *C*, *D*: plane parameters giving the spatial properties of the plane.

■ A, B, C and D can be calculated by three non-collinear points in the plane, selected in a **strictly counterclockwise order**, viewing the surface along a front-to-back direction.

■ Please refer to equations **(4-3)** and **(4-4)**.

Plane Equation by 3 Points

Geometry in 3D

Find an equation of the plane (if it exists) through the points A(1,-2,0), B(2,1,0) and C(-1,1,2).

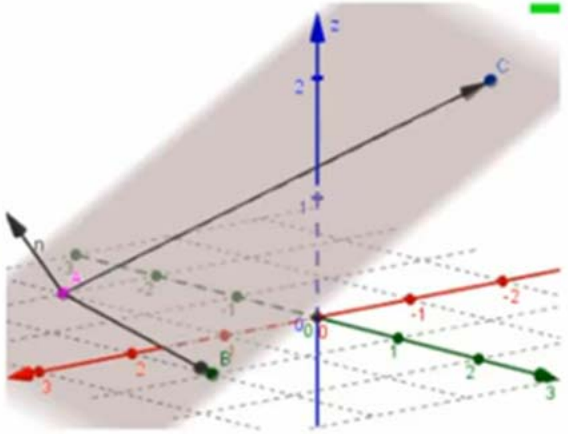
Equation of Plane with **Point** and **Normal Vector**

directional vector for line AB

$\vec{r}_{AB} = \langle B-A \rangle = \langle 2-1, 1+2, 0-0 \rangle = \langle 1,3,0 \rangle$

directional vector for line AC

$\vec{r}_{AC} = \langle C-A \rangle = \langle -1-1, 1+2, 2-0 \rangle = \langle -2,3,2 \rangle$



$$\vec{n} = \vec{r}_{AB} \times \vec{r}_{AC} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ 1 & 3 & 0 \\ -2 & 3 & 2 \end{vmatrix} = \vec{i}\cdot 6 - \vec{j}\cdot 2 + \vec{k}\cdot 9 = \langle 6,-2,9 \rangle$$

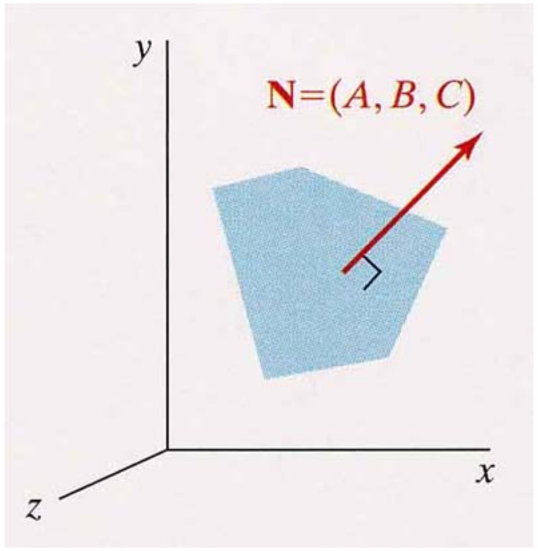
$$A(x-x_0)+B(y-y_0)+C(z-z_0)=0$$

$$6(x-1)+(-2)(y+2)+9(z-0)=0 \Rightarrow 6x-2y+9z=10$$

(By Linda Fahlberg-Stojanovska)

Orientation of Polygon Surface

- Normal vector for the plane containing that polygon



<u>The surface normal vector is:</u>

- perpendicular to the plane;
- points in the direction from inside to the outside;
- Cartesian components (A, B, C), where A, B and C are the plane coefficients

The normal vector N for a plane Ax + By + Cz + D = 0.

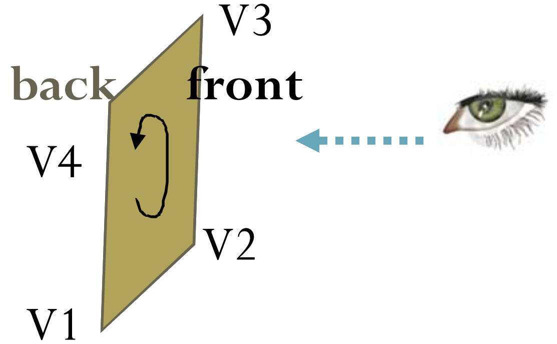
Polygon Faces

- In graphics, polygon faces are often distinguished between the two sides of each surface.
- Back face: the side that faces into the object interior.
  Front face: the side that is visible, or outward.
- Each side can have its own rendering style, such as
  - Filling
  - Wireframe
  - Vertex
  - …

Front and Back face of Polygons

- In OpenGL, each polygon we specify has two faces: a **back** face and a **front** face.
- Fill color and other attributes can be set for each face separately.

```
glPolygonMode(GL_FRONT, GL_FILL); //filled polygon
glPolygonMode(GL_BACK, GL_LINE); //wireframe
glBegin(GL_POLYGON);
            glVertex2f(…);
            ……
            glVertex2f(…);
glEnd();
```



- Define **front/back face**
  - Default: the polygon vertices in a counterclockwise order as we view the polygon from "outside".
  - User define:

```
glFrontFace(GL_CCW); //counterclockwise
glFrontFace(GL_CW);   //clockwise
```

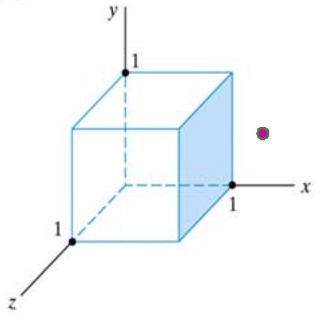
Identify Points Position Relative to Faces

- A basic task in many graphics algorithms
- How to do the test?
  - Each polygon is contained in an infinite plane that partitions the place into two regions.

■ Plane equation can be used.

    - Is the point in front of the plane?
    - Is the point inside the object?
      - Behind (inside) all polygon surface planes

If $Ax + By + Cz + D < 0$,
the point (x, y, z) is behind (inside) the plane;
If $Ax + By + Cz + D > 0$,
the point (x, y, z) is in front of (outside) the plane.



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

Any point outside it satisfies the inequality $x - 1 > 0$.

■These inequality tests are valid in a **right-handed** Cartesian system;
■ A, B, C and D were calculated using points selected in a **strictly counterclockwise order**, viewing the surface along a front-to-back direction.

Chapter 4
Graphics Output Primitives (Part II)

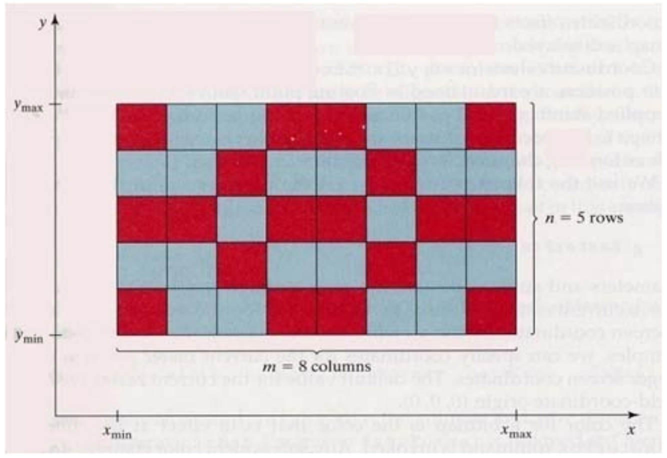
Pixel Array Primitives, Character Primitives and OpenGL Functions

Pixel-Array Primitive

- **Bitmap (mask, 2-color image)** is a pixel array with the bit value 0 or 1 to each element.

- **Pixmap** is a pixel array of color values

  How to obtain such pattern on screen?
  - Scanning a picture or generated by a graphics program.
  - So the pattern can be represented by a rectangular array of color values.
  - Then each color value in the array is mapped to one or more screen pixel positions.



Mapping an *n* by *m* color array onto a region of the screen coordinates.

OpenGL Pixel-Array Functions

- Two OpenGL functions are used to define pixel patterns: bitmap and pixmap.

- OpenGL **Bitmap** Function

glBitmap (width, height, x0, y0, xOffset, yOffset, bitShape);

width: the no. of columns for the Bitmap in the array **bitShape**;
height: the no. of rows for the Bitmap in the array **bitShape**;
Floating -point
(x0, y0): define the "origin" of the bitmap, which is positioned at the current raster position (next slide);
xOffset,yOffset: coordinate increments added to the raster position after the bitmap is displayed;
bitShape: the array of elements assigned either 1 (displayed in the previously set color) or 0 (unaffected by the bitmap); (example)

Spec.:http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/gl/bitmap.html

OpenGL Pixel-Array Functions

- OpenGL **Bitmap** Function (cont.)

Current raster position:

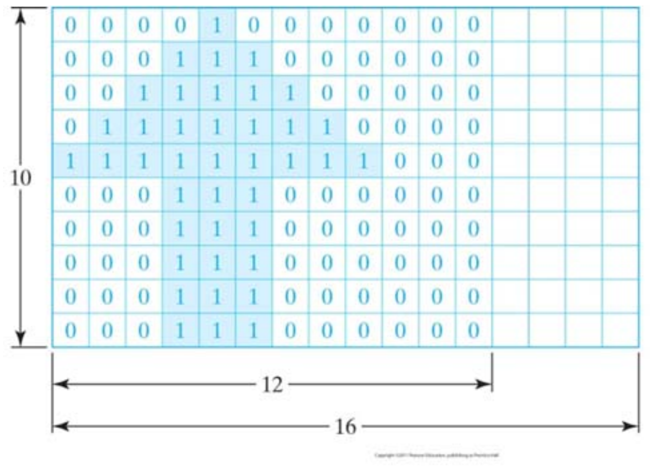
A location in the frame buffer where the pattern is to be applied.

glRasterPos*()

which sets the coordinates (floating-point) for the current raster position, and the default value is the world-coordinate origin.

OpenGL Bitmap Function

- **bitShape**: the rectangular bit array
  - Each row is stored in multiples of 8 bits, where the binary data is arranged as a set of 8-bit unsigned characters.
- But the arbitrary shape can be defined by any convenient grid size in a bit pattern.



0	0	0	0	1	0	0	0	0	0	0	0				
0	0	0	1	1	1	0	0	0	0	0	0				
0	0	1	1	1	1	1	0	0	0	0	0				
0	1	1	1	1	1	1	1	0	0	0	0				
1	1	1	1	1	1	1	1	1	0	0	0				
0	0	0	1	1	1	0	0	0	0	0	0				
0	0	0	1	1	1	0	0	0	0	0	0				
0	0	0	1	1	1	0	0	0	0	0	0				
0	0	0	1	1	1	0	0	0	0	0	0				
0	0	0	1	1	1	0	0	0	0	0	0				

0×08	0×00
0×1C	0×00
0×3E	0×00
0×7F	0×00
0×FF	0×80
0×1C	0×00
0×1C	0×00
0×1C	0×00
0×1C	0×00
0×1C	0×00

Fig.4-27

- A bit pattern defined on a 10-row by 9-colume grid.
- The binary data is specified with 16 bits for each row.
- When applied, all bit values beyond the 9th column are ignored.

OpenGL Bitmap Function

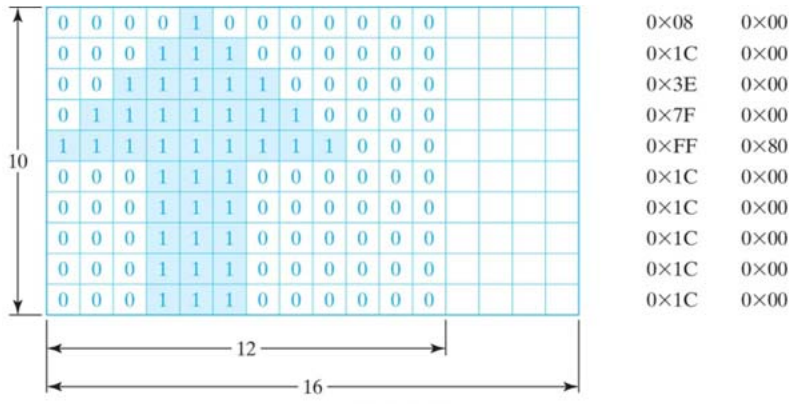
- Apply the bit pattern of Fig. 4-27.

```
 //array values defined row by row
GLubyte bitShape[20] = {
   0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00, 0xff, 0x80,
   0x7f, 0x00, 0x3e, 0x00, 0x1c, 0x00, 0x08, 0x00 };

//set the storage mode for the bitmap
glPixelStorei (GL_UNPACK_ALIGHMENT, 1);
//set the current raster position
glRasterPos2i (30, 40);
glBitmap (9, 10, 0.0, 0.0, 20.0, 15.0, bitShape);
```

The alignment requirements for the start of each pixel row in memory.

width **height**



OpenGL Image (Pixmap) Functions

- OpenGL provides three basic commands to manipulate image data
  - **glDrawPixels():** Writes [a rectangular array of pixels] from data kept in **memory** into the **framebuffer** at the current raster position specified by **glRasterPos*()**.
  - **glReadPixels():** Reads [a rectangular array of pixels] from the **framebuffer** and stores the data in **memory**.
  - **glCopyPixels():** Copies [a rectangular array of pixels] from one part of the **framebuffer** to **another**.
    - This command behaves similarly to a call to **glReadPixels()** followed by a call to **glDrawPixels()**, but the data is **never** written into the memory.

OpenGL Image Functions

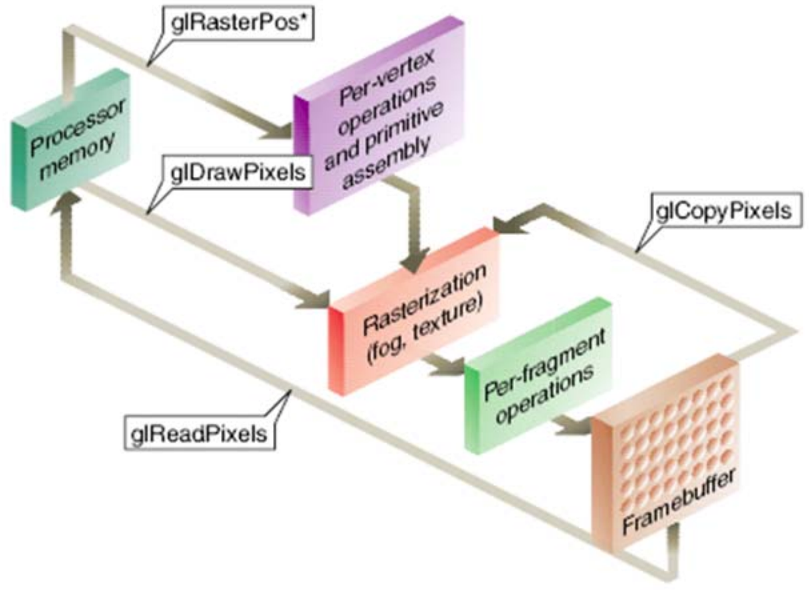


Figure 8-3 Simplistic Diagram of Pixel Data Flow

- The coordinates of **glRasterPos*()**, which specify the current raster position used by **glDrawPixels()** and **glCopyPixels()**, are transformed by the geometric processing pipeline.

(From OpenGL Programming Guide)

OpenGL Image Functions

- Write an image which is kept in processor **memory** into the **framebuffer** at the current raster position

glDrawPixels (width, height, dataFormat, dataType, pixMap);

pixMap draws to the current raster position.

width, height: the column and row dimensions of the **pixMap**;

dataFormat (OpenGL constant): to indicate how the array values are specified, such as GL_BLUE, GL_BGR; GL_DEPTH_COMPONENT, GL_STENCIL_INDEX.

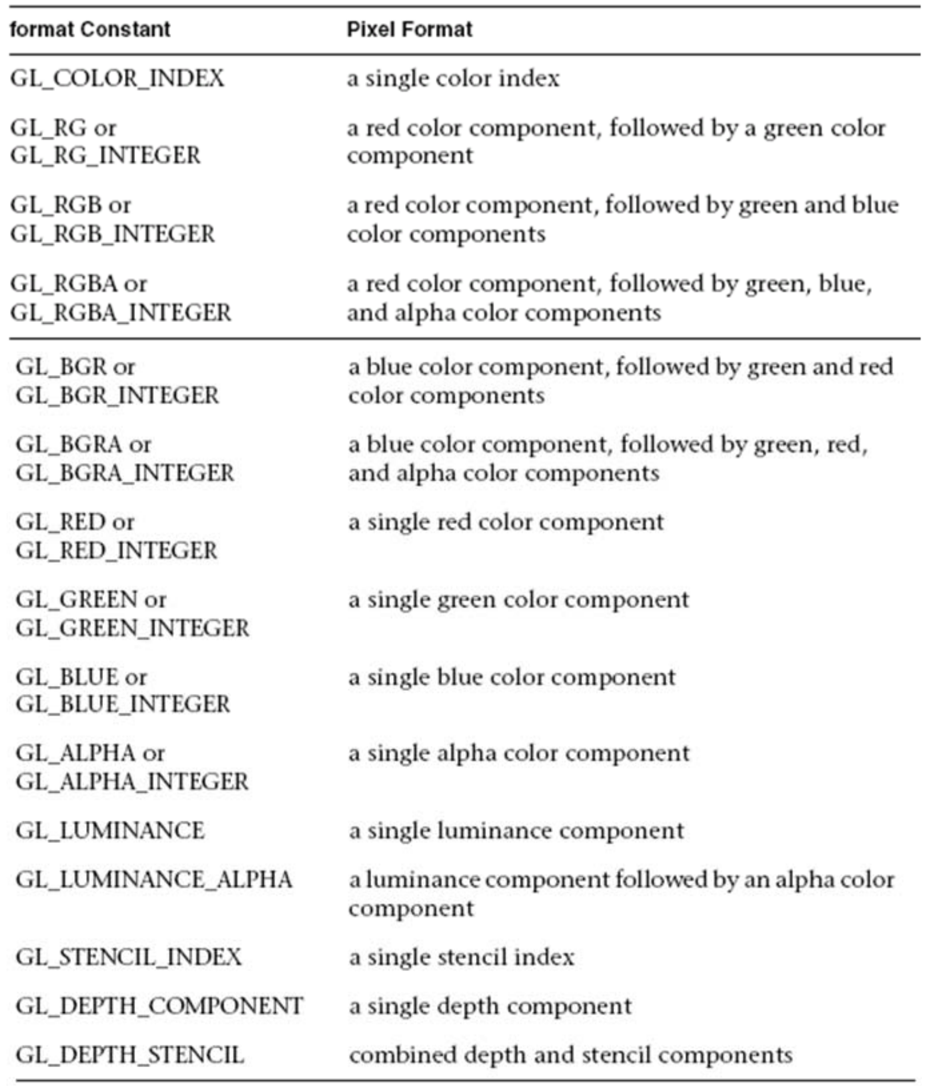
dataType (OpenGL constant): to specify the pixel format of the **pixMap**, such as GL_BYTE, GL_INT, or GL_FLOAT

Eg: a 128-by-128 array of RGB color values pixmap

glDrawPixels (128, 128, GL_RGB, GL_UNSIGNED_BYTE, colorShape);

OpenGL Image Functions

- Data format



format Constant	Pixel Format
GL_COLOR_INDEX	a single color index
GL_RG or GL_RG_INTEGER	a red color component, followed by a green color component
GL_RGB or GL_RGB_INTEGER	a red color component, followed by green and blue color components
GL_RGBA or GL_RGBA_INTEGER	a red color component, followed by green, blue, and alpha color components
GL_BGR or GL_BGR_INTEGER	a blue color component, followed by green and red color components
GL_BGRA or GL_BGRA_INTEGER	a blue color component, followed by green, red, and alpha color components
GL_RED or GL_RED_INTEGER	a single red color component
GL_GREEN or GL_GREEN_INTEGER	a single green color component
GL_BLUE or GL_BLUE_INTEGER	a single blue color component
GL_ALPHA or GL_ALPHA_INTEGER	a single alpha color component
GL_LUMINANCE	a single luminance component
GL_LUMINANCE_ALPHA	a luminance component followed by an alpha color component
GL_STENCIL_INDEX	a single stencil index
GL_DEPTH_COMPONENT	a single depth component
GL_DEPTH_STENCIL	combined depth and stencil components

(From OpenGL Programming Guide)

Store Pixel Values in Buffer

- After defining the shape or pattern by **glDrawPixels** routine, a **target buffer** should be specified.

- OpenGL buffers and their uses (store color values and other kinds of pixel data information)
  - **Color buffer**

    There are 4 available in OpenGL for screen refreshing.: A left-right pair of color buffers is for displaying stereoscopic views, a front-back pair for double-buffered animating displays;
  - **Depth buffer (*z buffer*)**

    Store object distances (depths) from the viewing position;
  - **Stencil buffer**

    Store rendering patterns for a scene;

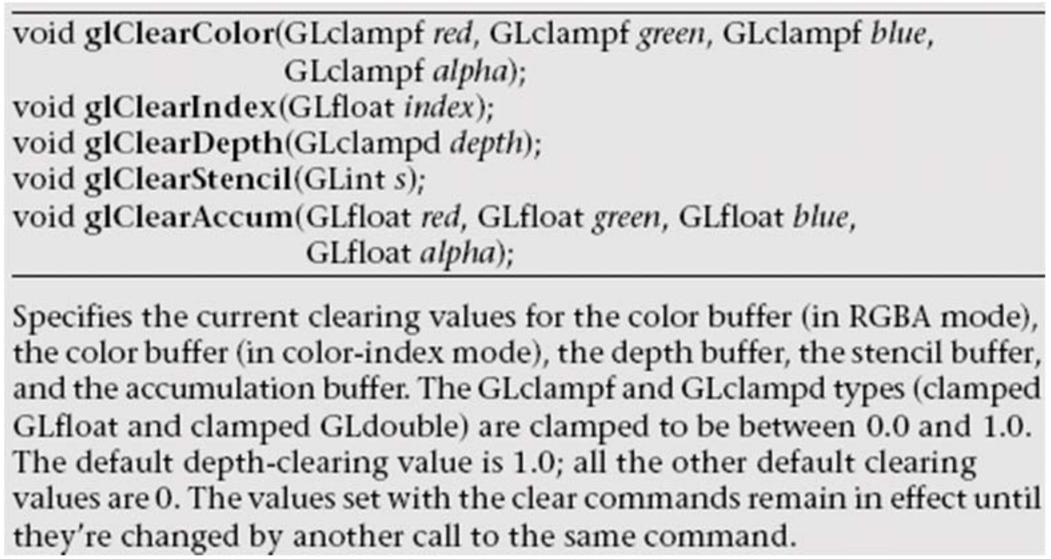
(From:Wiki)

  - **Accumulation buffer**

    Accumulating a series of images into a final, composite image.

Store Pixel Values in Buffer

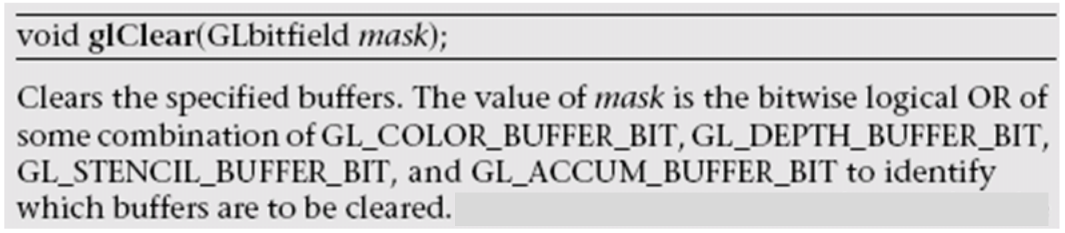
- Operations to clear buffers
  - Firstly, to set the clearing values for each buffer glClearColor(1.0, 1.0, 1.0, 1.0);



```
void glClearColor(GLclampf red, GLclampf green, GLclampf blue,
                  GLclampf alpha);
void glClearIndex(GLfloat index);
void glClearDepth(GLclampd depth);
void glClearStencil(GLint s);
void glClearAccum(GLfloat red, GLfloat green, GLfloat blue,
                  GLfloat alpha);
```

Specifies the current clearing values for the color buffer (in RGBA mode), the color buffer (in color-index mode), the depth buffer, the stencil buffer, and the accumulation buffer. The GLclampf and GLclampd types (clamped GLfloat and clamped GLdouble) are clamped to be between 0.0 and 1.0. The default depth-clearing value is 1.0; all the other default clearing values are 0. The values set with the clear commands remain in effect until they're changed by another call to the same command.

  - Then to clear the buffers glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);



```
void glClear(GLbitfield mask);
```

Clears the specified buffers. The value of *mask* is the bitwise logical OR of some combination of GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT, and GL_ACCUM_BUFFER_BIT to identify which buffers are to be cleared.

(From OpenGL Programming Guide)

Store Pixel Values in Buffer

- To select which color buffer will be drawn

```
glDrawBuffer (buffer);
```

OpenGL symbolic constants for "buffer":

A single buffer: GL_FRONT_LEFT, GL_FRONT_RIGHT,
GL_BACK_LEFT, GL_BACK_RIGHT;

Two buffers: GL_FRONT, GL_BACK; GL_LEFT, GL_RIGHT;

All available four buffers: GL_FRONT_AND_BACK

OpenGL Raster Operations

- Other operations on a pixel array by OpenGL:
  - Retrieve a block of values from a buffer into memory;
  - Copy a block of values between buffers array;
  - Logic operations to combine the two blocks of pixel values.

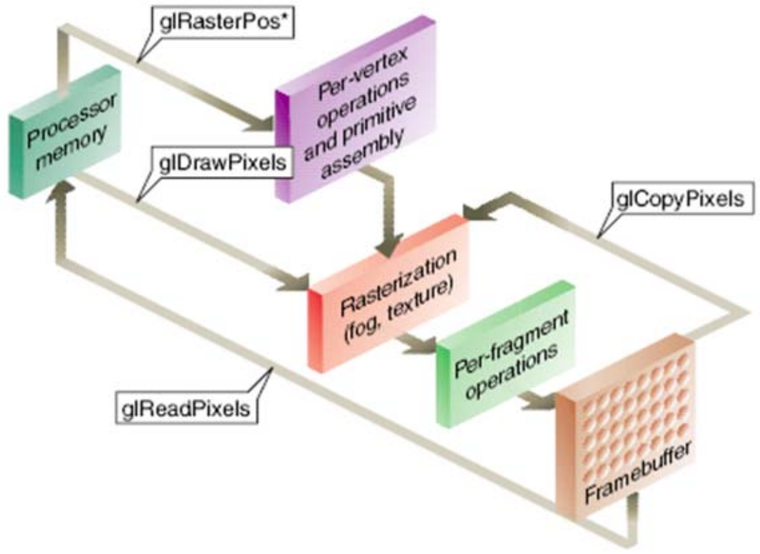


Figure 8-3 Simplistic Diagram of Pixel Data Flow

OpenGL Raster Operations

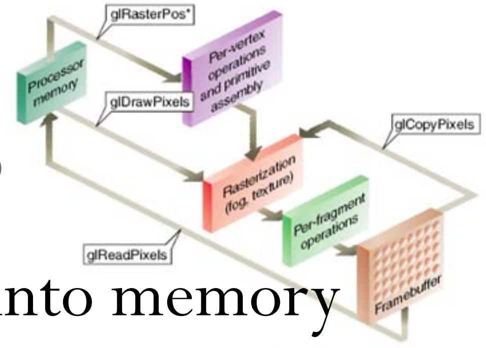


Figure 8-3 Simplistic Diagram of Pixel Data Flow

- Retrieving a block of values from a buffer into memory (array)
  - Firstly, to select a block of pixel values in a designated buffers

glReadPixels (xmin, ymin, width, height, dataFormat, dataType, array);

(xmin,ymin): the lower-left corner coordinate position of the rectangular block to be retrieved;

dataFormat: depend on the selected buffer, Eg, GL_DEPTH_COMPONENT for the depth buffer, or GL_STENCIL_INDEX for the stencil buffer;

Other parameters such as **width, height, dataType** are the same as in the **glDrawPixels** routine.

OpenGL Raster Operations

- Retrieving a block of values from a buffer into memory (array) (cont.)
  - Then, to define the source buffer where the block of pixel values is stored

```
glReadBuffer (buffer);
```

  - Symbolic constants for **buffer** are the same as in the **glDrawBuffer** routine.
  - But, we can't select all four of the buffers.
  - The default selection is the front left-right pair or the front-left buffer depending on the stereoscopic viewing.

OpenGL Raster Operations

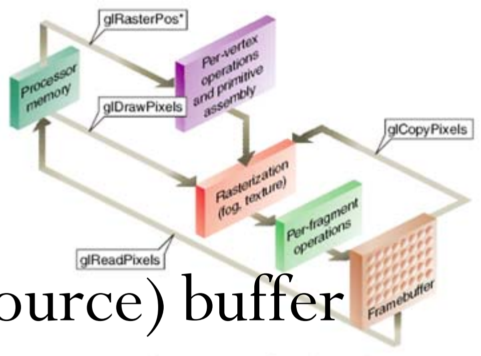


Figure 8-3 Simplistic Diagram of Pixel Data Flow

- Copy the block of pixel values from one (source) buffer into another (destination) buffer (by current raster position)

glCopyPixels (xmin, ymin, width, height, pixelValues);

roughly = glReadPixels () + glDrawPixels () without mem. array

pixelValues: GL_COLOR, GL_DEPTH, or GL_STENCIL to indicate the kind of data (the framebuffer used) to copy: color values, depth values, or stencil values.

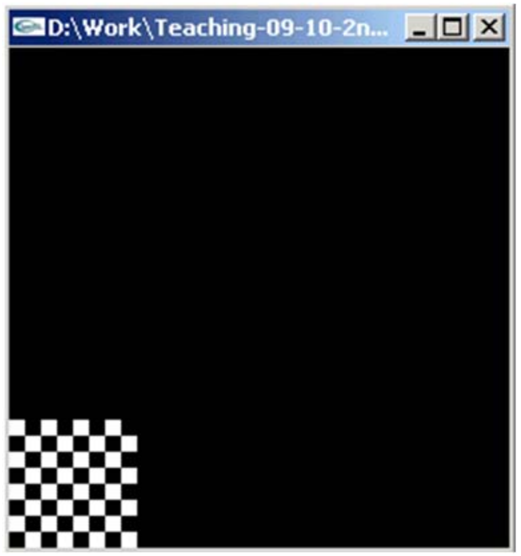
xmin, ymin, width, height : the same as the above routines;

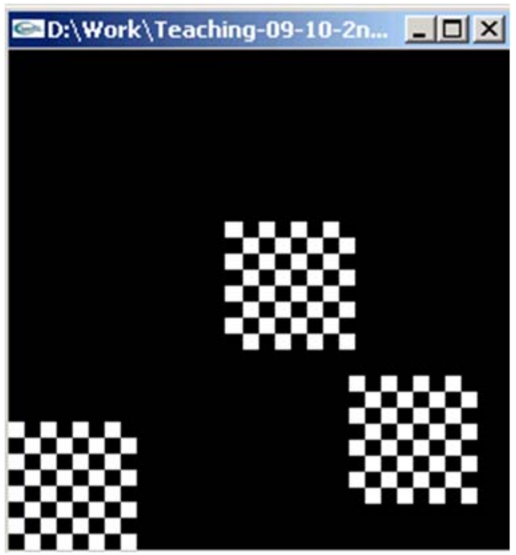
Source: by **glReadBuffer (buffer)**;

Dest. : by **glDrawBuffer (buffer)**;

Example

- To demonstrate drawing pixels and show the effect of glDrawPixels(), glCopyPixels(), and glPixelZoom().





file

Moving the mouse while pressing the mouse button will copy the image in the lower-left corner of the window to the mouse position, using the current pixel zoom factors.

(From OpenGL Programming Guide)

OpenGL Raster Operations

- Logic operations to combine the two blocks of pixel values

```
glEnable(GL_COLOR_LOGIC_OP);
glLogicOp(logicOp);
```

logicOp:

- GL_AND, CL_OR, GL_XOR, GL_COPY_INVERTED, GL_INVERT;
- GL_CLEAR (0), GL_SET (1);
- GL_COPY (by default).

Character Primitives

- Routines for generating characters are available in most graphics packages.
- Font (Typeface)
  - The overall design style for a set of characters.
  - Two kinds of representations for storing computer fonts
    - ***bitmap* (*raster*)** font
    - ***outline* (*stroke*)** font

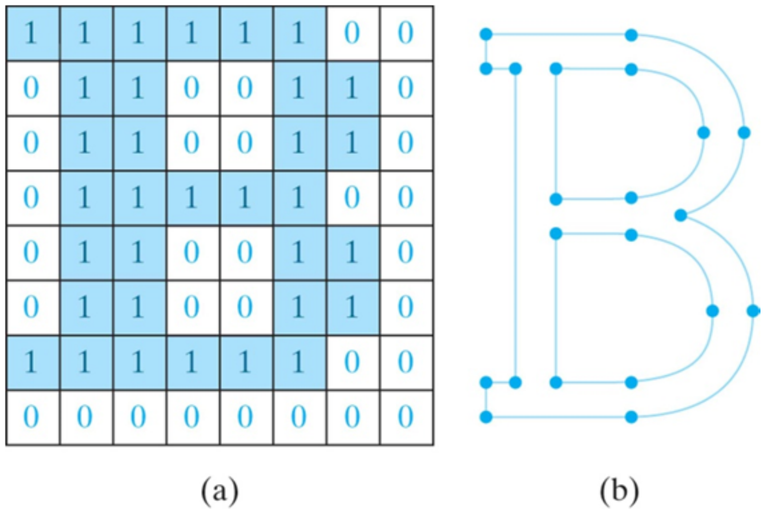


FIGURE 4-28 The letter "B" represented with an 8-by-8 *bitmap* pattern (a), and with an *outline* shape defined with straight-line and curve segments (b).

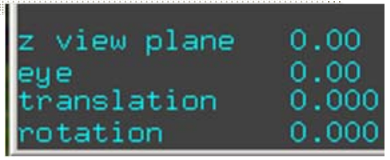
OpenGL Character Functions

- OpenGL only has low-level support for displaying characters
  - Explicitly define any character as a bitmap.
  - Displayed by mapping the bitmap in the frame buffer.

- OpenGL Utility Toolkit (GLUT) contains routines for displaying predefined character sets
  - ***bitmap*** font

glutBitmapCharacter(font, character);

glutBitmapCharacter(GLUT_BITMAP_9_BY_15, chara);



```
z view plane    0.00
eye             0.00
translation     0.000
rotation        0.000
```

  - ***outline*** font

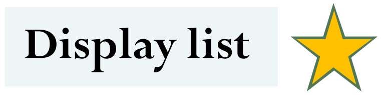
glutStrokeCharacter(font, character);

Chapter 4
Graphics Output Primitives (Part II)

OpenGL Display Lists

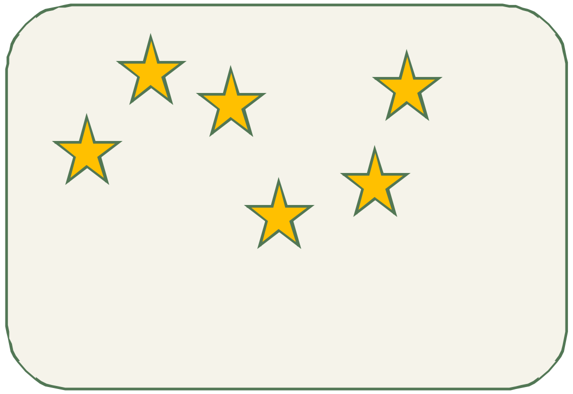
OpenGL Display Lists

- Display list (OpenGL 1.1)
  - A group of OpenGL commands that have been stored (or compiled) for later execution.
  - Once it has been created, we can reference it multiple times with different display operations.



Code:

```
{      ...
       Display list
       ...
       Display list
       ...
       Display list
       ...
}
```



  - Useful for hierarchical modeling, where a complex object can be described with a set of simpler subparts.

Creating and Naming OpenGL Display List

- To form a display list, put a set of OpenGL commands into **glNewList/glEndList** pair

```
glNewList(listID, listMode);
 … …
glEndList();
```

listMode: GL_COMPILE or GL_COMPILE_AND_EXECUTE.

```
//let OpenGL generate an identifier
listID = glGenLists(1);

//check if listID has been used
glIsList(listID);
```

return value: GL_TRUE – already used
GL_FALSE – not used

Executing and Deleting OpenGL Display List

- Executing OpenGL display lists

```
glCallList ( listID );
```

- Deleting OpenGL display lists

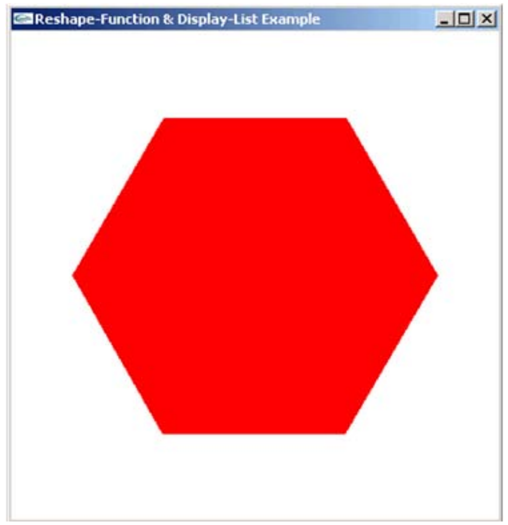
```
glDeleteList ( startID, nLists );
```

startID: the initial display-list identifier

nlists: the number of lists that are to be deleted

OpenGL Display Lists Example (segment)

```
// draw a red hexagon
const double TWO_PI = 6.281853;
GLuint        regHex;
GLdouble      theta;
GLint         x, y, k;
regHex = glGenLists(1);
glNewList (regHex, GL_COMPILE);
    glBegin (GL_POLYGON);
            for (k=0; k<6; k++) {
                    theta = TWO_PI * k / 6.0;
                    x = 200 + 150 * cos (theta);
                    y = 200 + 150 * sin (theta);
                    glVertex2i (x, y);
            }
    glEnd();
glEndList();
glCallList (regHex);
```



Summary of Chapter 4

- Line drawing algorithms
  - DDA (Digital Differential Analyzer)
  - Bresenham
- Output primitives
  - Points, straight lines, curves
  - Filled color areas
  - Polygons
  - Pixel-array primitives and characters
- OpenGL functions
- Read the example programs on P117 in the textbook