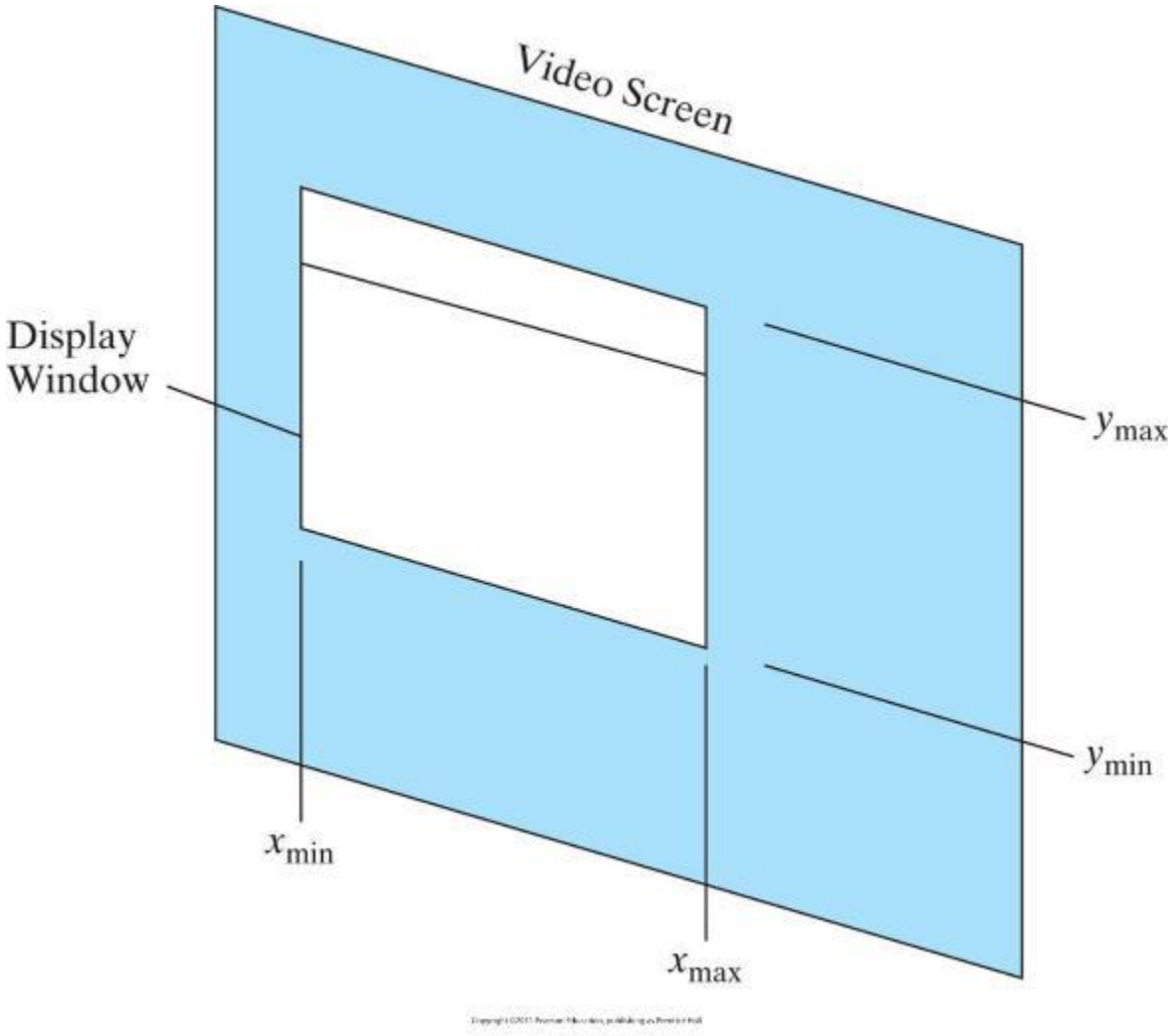
CSC 470 COMPUTER GRAPHICS

OpenGL primitives

The drawing window



```
#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, 0.0, 1.0);      // Set line segment color to red.
                glBegin (GL_LINES);
                        glVertex2i (180, 15);       // Specify line-segment
                        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.}
```

Skeleton Event-driven Program

```
// include OpenGL libraries
void main()
{
  glutDisplayFunc(myDisplay);    // register the redraw function
  glutReshapeFunc(myReshape);    // register the reshape function
  glutMouseFunc(myMouse);  // register the mouse action function
  glutMotionFunc(myMotionFunc); // register the mouse motion
  function
    glutKeyboardFunc(myKeyboard); // register the keyboard action
  function
  …perhaps initialize other things…
  glutMainLoop();                  // enter the unending main loop
}
…all of the callback functions are defined here
```

A GL Program to Open a Window

```
// appropriate #includes go here
void main(int argc, char** argv)
{
  glutInit(&argc, argv);      // initialize the toolkit
  glutInitDisplayMode(GLUT_SINGLE |
  GLUT_RGB);                  // set the display mode
  glutInitWindowSize(640,480); // set window size
  glutInitWindowPosition(100, 150);
// set window upper left corner position on screen
  glutCreateWindow("my first attempt");
// open the screen window (Title: my first attempt)
  // continued next slide
```

Part 2 of Window Program

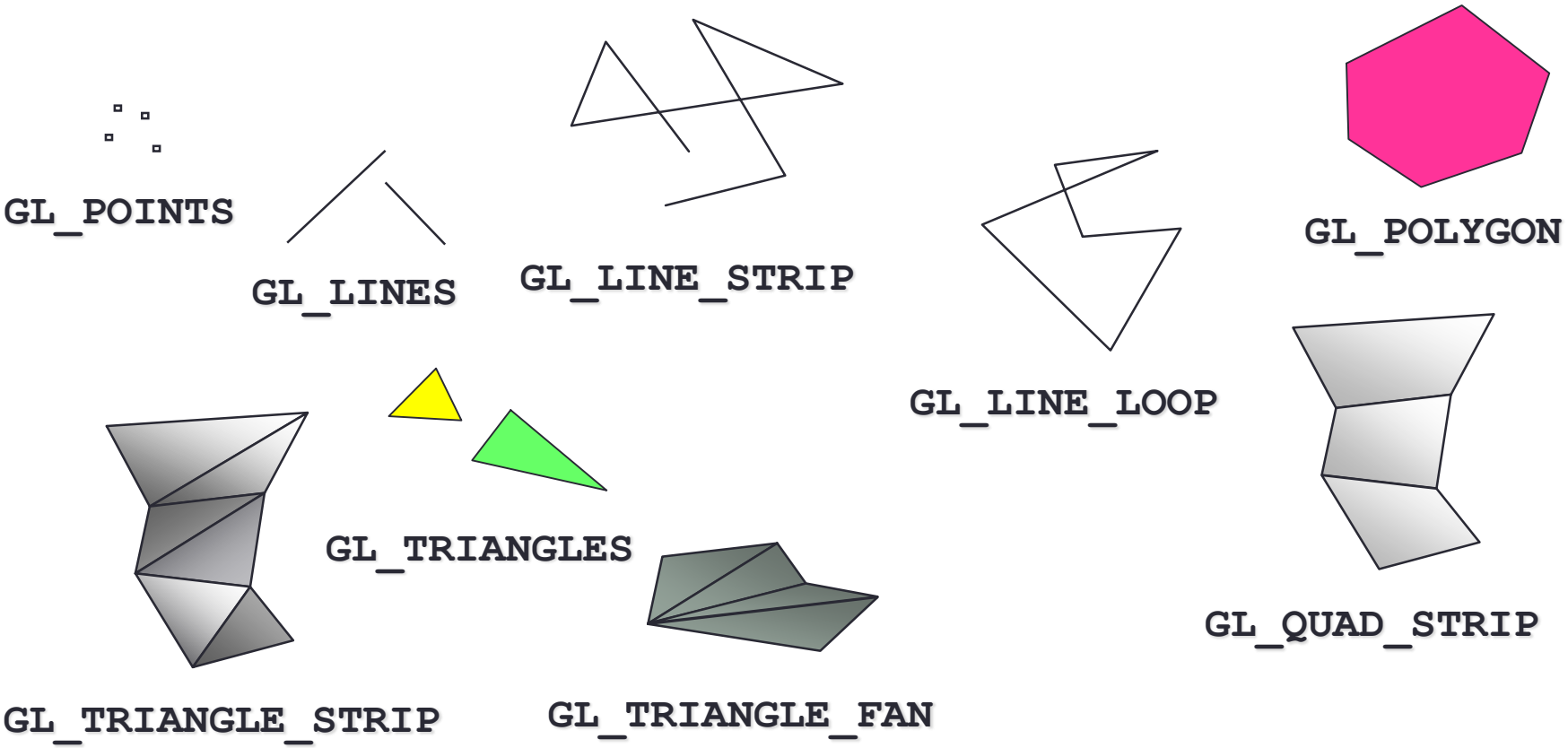
```
        // register the callback functions
   glutDisplayFunc(myDisplay);
   glutReshapeFunc(myReshape);
   glutMouseFunc(myMouse);
   glutKeyboardFunc(myKeyboard);
   myInit(); // additional initializations as necessary
   glutMainLoop();        // go into a perpetual loop
}
```

- Terminate program by closing window(s) it is using.

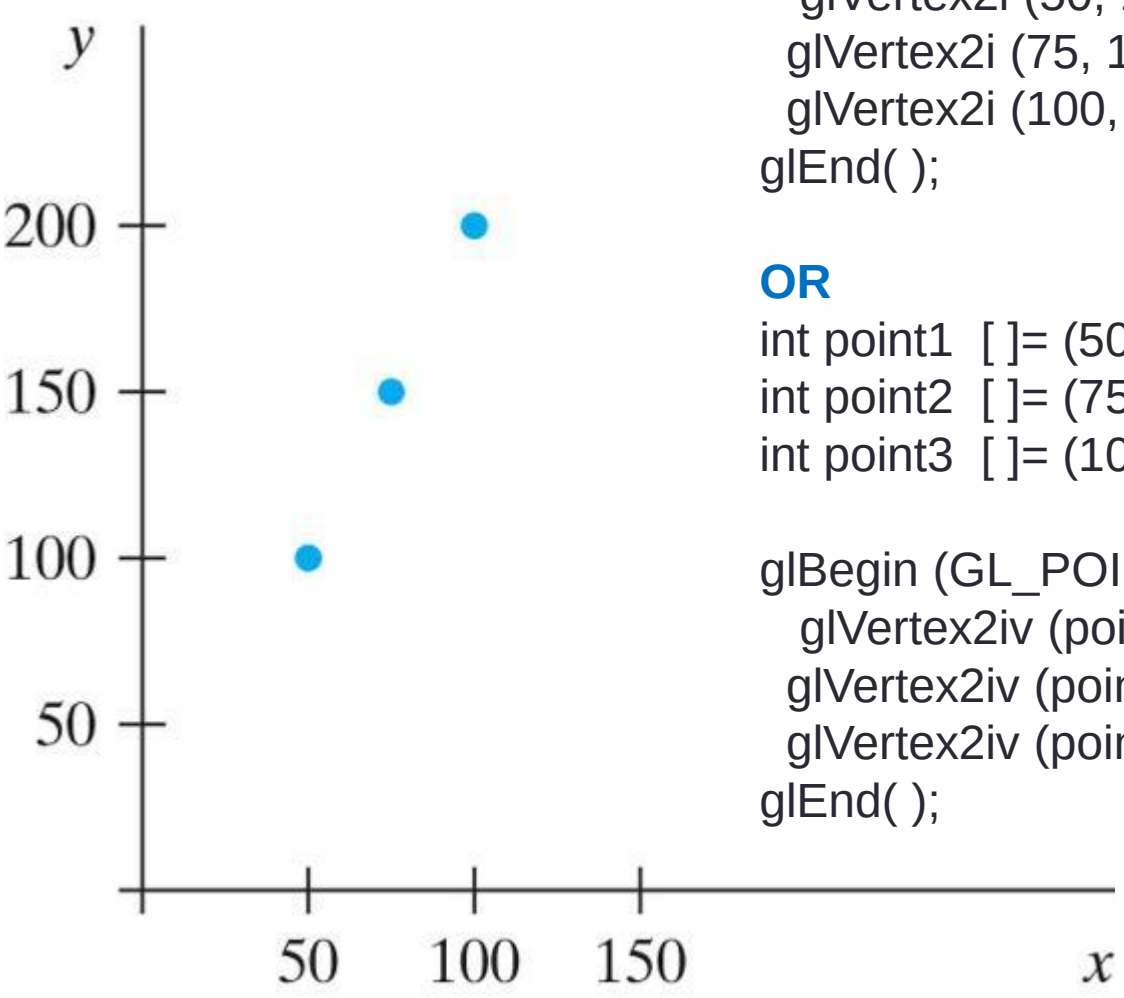
Setting Up a Coordinate System

```
void myInit(void)
{
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0, 640.0, 0, 480.0);
}
// sets up coordinate system for window from (0,0) to (639,
  479)
```

OpenGL Primitives



Display of three point positions generated with glBegin (GL_POINTS)

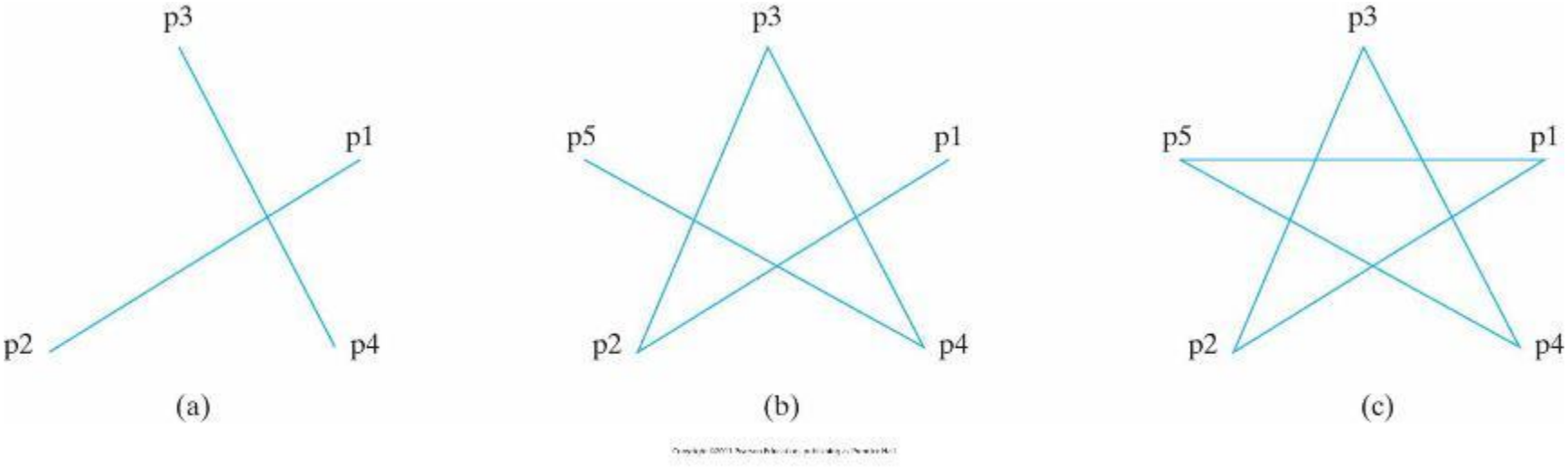



```
glBegin (GL_POINTS);
   glVertex2i (50, 100);
  glVertex2i (75, 150);
  glVertex2i (100, 200);
glEnd( );
```

OR

```
int point1  [ ]= (50, 100);
int point2  [ ]= (75, 150);
int point3  [ ]= (100, 200);

glBegin (GL_POINTS);
   glVertex2iv (point1);
  glVertex2iv (point2);
  glVertex2iv (point3);
glEnd( );
```

Line segments that can be displayed in OpenGL using a list of 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.

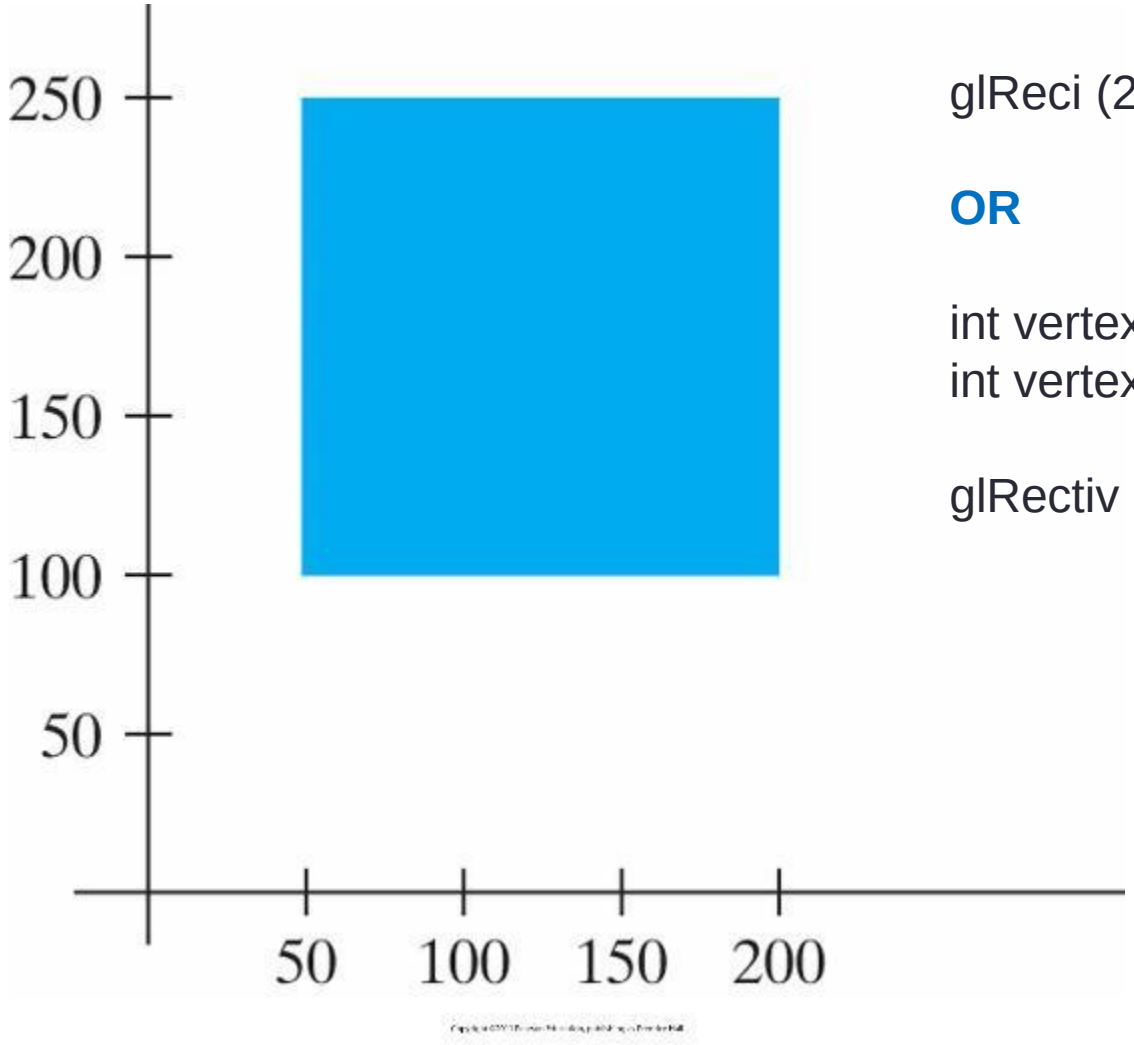



```
glBegin (GL_LINES);
   glVertex2iv (point1);
   glVertex2iv (point2);
   glVertex2iv (point3);
   glVertex2iv (point4);
   glVertex2iv (point5);
glEnd( );
```

```
glBegin (GL_LINE_STRIP);
   glVertex2iv (point1);
   glVertex2iv (point2);
   glVertex2iv (point3);
   glVertex2iv (point4);
   glVertex2iv (point5);
glEnd( );
```

```
glBegin (GL_LINE_LOOP);
   glVertex2iv (point1);
   glVertex2iv (point2);
   glVertex2iv (point3);
   glVertex2iv (point4);
   glVertex2iv (point5);
glEnd( );
```

The display of a square fill area using the glRect function.



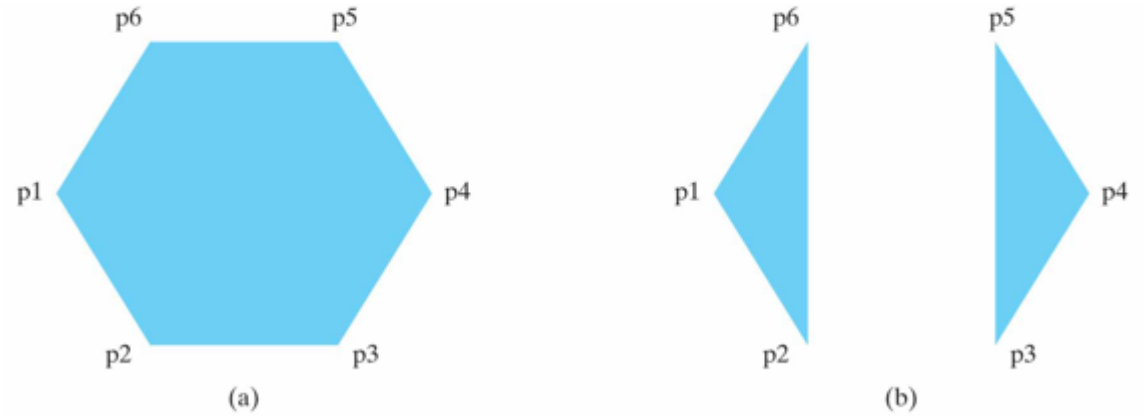
glReci (200, 100, 50, 250);

OR

int vertex1 [ ] = (200, 100);
int vertex2 [ ] = (50, 250);

glRectiv (vertex1, vertex2);

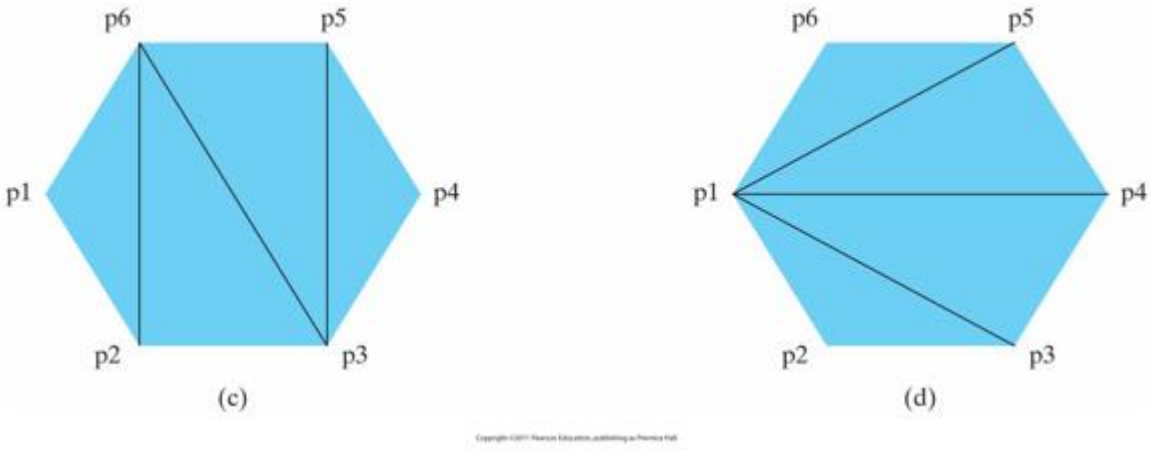
Displaying polygon fill areas using a list of six vertex positions. (a) A single convex polygon fill area generated with the primitive constant GL_POLYGON. (b) Two unconnected triangles generated with GL_ TRIANGLES.




```
glBegin (GL_POLYGON);
  glVertex2iv (point1);
  glVertex2iv (point2);
  glVertex2iv (point3);
  glVertex2iv (point4);
  glVertex2iv (point5);
  glVertex2iv (point6);
glEnd ( );
```

```
glBegin (GL_TRIANGLES);
  glVertex2iv (point1);
  glVertex2iv (point2);
  glVertex2iv (point3);
  glVertex2iv (point4);
  glVertex2iv (point5);
  glVertex2iv (point6);
glEnd ( );
```

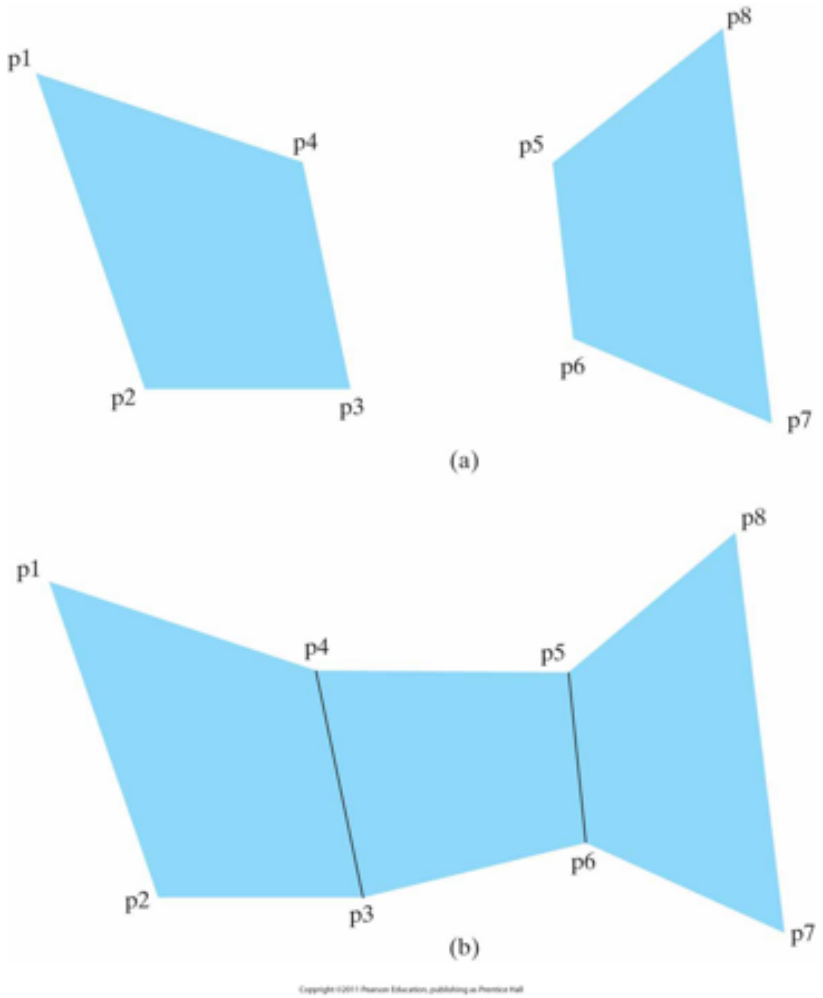
Displaying polygon fill areas using a list of six vertex positions.(c) Four connected triangles generated with GL_TRIANGLE_STRIP. (d) Four connected triangles generated with GL_TRIANGLE_FAN.




```
glBegin (GL_TRIANGLE_STRIP);
  glVertex2iv (point1);
  glVertex2iv (point2);
  glVertex2iv (point6);
  glVertex2iv (point3);
  glVertex2iv (point5);
  glVertex2iv (point4);
glEnd ( );
```

```
glBegin (GL_TRIANGLE_FAN);
  glVertex2iv (point1);
  glVertex2iv (point2);
  glVertex2iv (point3);
  glVertex2iv (point4);
  glVertex2iv (point5);
  glVertex2iv (point6);
glEnd ( );
```

Displaying 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.




```
glBegin (GL_QUADS);
  glVertex2iv (point1);
  glVertex2iv (point2);
  glVertex2iv (point3);
  glVertex2iv (point4);
  glVertex2iv (point5);
  glVertex2iv (point6);
  glVertex2iv (point7);
  glVertex2iv (point8);
glEnd ( );
```

```
glBegin (GL_QUAD_STRIP);
  glVertex2iv (point1);
  glVertex2iv (point2);
  glVertex2iv (point4);
  glVertex2iv (point3);
  glVertex2iv (point5);
  glVertex2iv (point6);
  glVertex2iv (point8);
  glVertex2iv (point7);
glEnd ( );
```

List of the Primitives

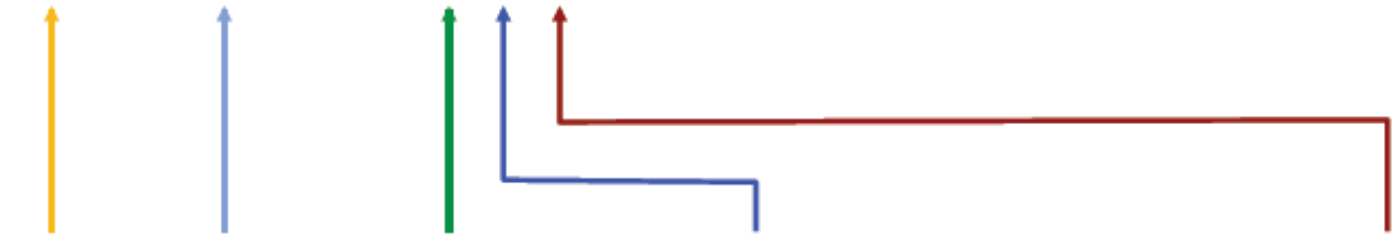
GL_POINTS	Used to draw individual points
GL_LINES	Used to draw lines
GL_LINE_STRIP	Drawing lines from the first vertex to the last without picking up the pen
GL_LINE_LOOP	Same as line strip but the first and last vertices are connected
GL_TRIANGLES	Connects a solid triangle
GL_TRIANGLE_STRIP	Every set of three vertices to form a triangle
GL_TRIANGLE_FAN	Triangles all forming a common point at the first vertex
GL_QUADS	Forms four sided polygons
GL_QUAD_STRIP	Four sided polygons in a strip
GL_POLYGON	Set of vertices that from an object

Open-GL Data Types

suffix	data type	C/C++ type	OpenGL type name
b	**8-bit integer**	**signed char**	**GLbyte**
s	**16-bit integer**	**Short**	**GLshort**
i	**32-bit integer**	**int or long**	**GLint, GLsizei**
f	**32-bit float**	**Float**	**GLfloat, GLclampf**
d	**64-bit float**	**Double**	**GLdouble,GLclampd**
ub	**8-bit unsigned number**	**unsigned char**	**GLubyte,GLboolean**
us	**16-bit unsigned number**	**unsigned short**	**GLushort**
ui	**32-bit unsigned number**	**unsigned int or unsigned long**	**GLuint,Glenum,GLbitfield**

What Code Does: GL Function Construction

```
glVertex3fv( . . . )
```



API	base call	argument count	argument data type			vector
gl	Color	2 – (x, y)	b	8 bit	GLbyte	omit 'v' for scalar form
glu	Normal	3 – (x, y, z)	ub	8 bit unsigned	GLubyte, GLboolean	`glVertex2f(x,y)`
glut	Flush	4 – (x, y, z, w) or	s	16 bit	GLshort	
glX	Vertex	4 – (r, g, b, a)	us	16 bit unsigned	GLushort	
agl	...		i	32 bit	GLint, GLsizei	
wgl			ui	32 bit unsigned	GLuint, GLenum, GLbitfield	
			f	32 bit	GLfloat, GLclampf	
			d	64 bit	GLdouble, GLclampd	

Colors in OpenGL

- OpenGL uses RGB color model
- The values of red, green, and blue are numbers from 0.0 to 1.0.

Composite Color	Red	Green	Blue
Black	0.00	0.00	0.00
Red	1.00	0.00	0.00
Green	0.00	1.00	0.00
Yellow	1.00	1.00	0.00
Blue	0.00	0.00	1.00
Magenta	1.00	0.00	1.00
Cyan	0.00	1.00	1.00
Dark Gray	0.25	0.25	0.25
Light Gray	0.75	0.75	0.75
Brown	0.60	0.40	0.12
Pumpkin Orange	0.98	0.62	0.12
Pastel pink	0.98	0.04	0.70
Barney Purple	0.60	0.40	0.70
White	1.00	1.00	1.00

Colors in OpenGL

- The instruction
  glColor3f(**float** *red*, **float** *green*, **float** *blue*)
  sets a color.

- Syntax of OpenGL instructions:

  FunctionName2i or FunctionName3f

- 2 or 3 means number of parameters: 2 or 3
- i - integer values, f- float values, d-double values.

- All the following instructions will give the same color - red:
- `glColor1f( 1.0 );`
- `glColor1d( 1.0 );`
- `glColor4i( 1, 0, 0, 0 );`.

Setting Drawing Colors in GL

- glColor3f(red, green, blue);
  // set drawing color
  - glColor3f(1.0, 0.0, 0.0); // red
  - glColor3f(0.0, 1.0, 0.0); // green
  - glColor3f(0.0, 0.0, 1.0); // blue
  - glColor3f(0.0, 0.0, 0.0); // black
  - glColor3f(1.0, 1.0, 1.0); // bright white
  - glColor3f(1.0, 1.0, 0.0); // bright yellow
  - glColor3f(1.0, 0.0, 1.0); // magenta

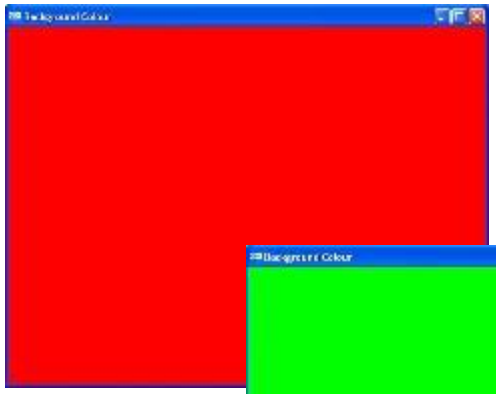
Setting Background Color in GL

- glClearColor (red, green, blue, alpha);
  - Sets background color.
  - Alpha is degree of transparency; use 0.0 for now.
- glClear(GL_COLOR_BUFFER_BIT);
  - clears window to background color

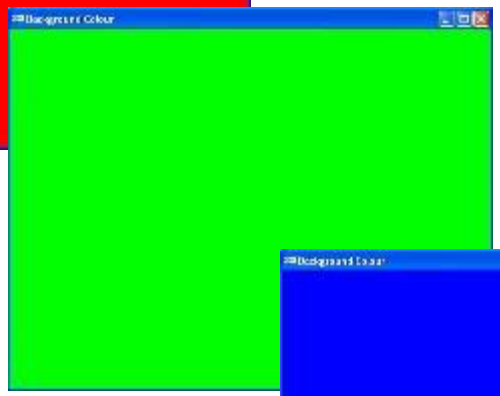
Setting the Background Color

```
glClearColor(R, G, B, α);
```

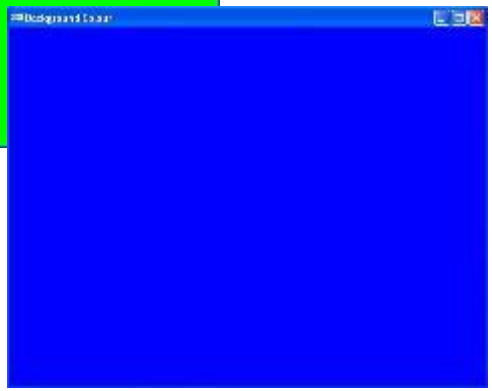
```
glClearColor(1,0,0,0);
```



```
glClearColor(0,1,0,0);
```

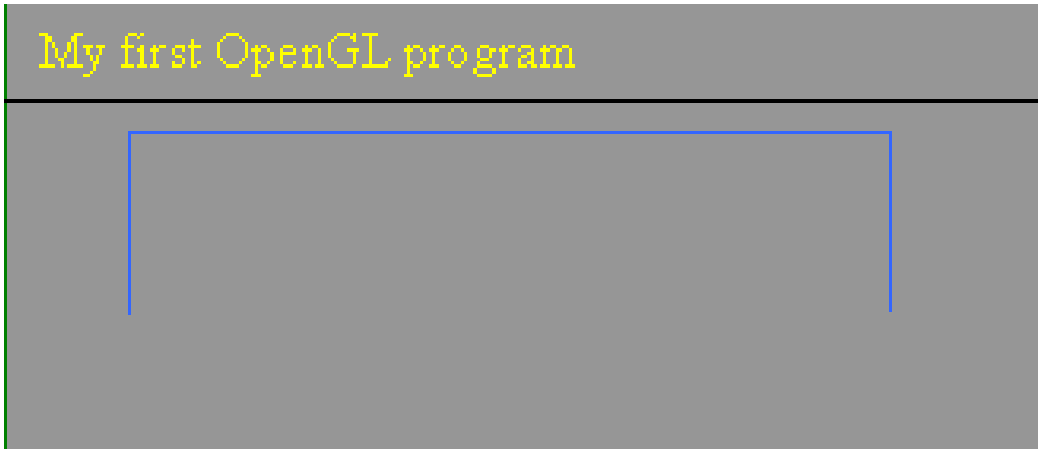


```
glClearColor(0,0,1,0);
```



GL_LINE_STRIP: draws lines without lifting the pen

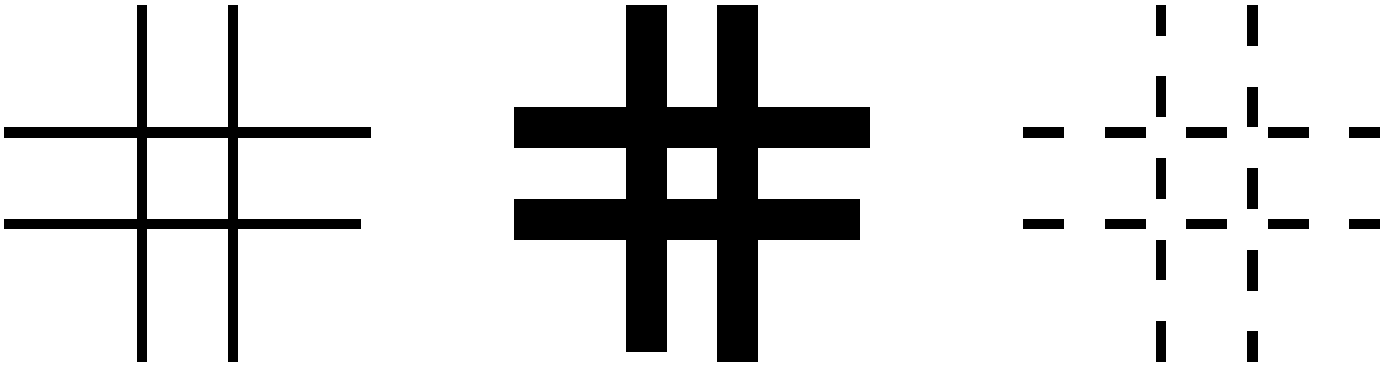
```
glColor3f( 0.0, 0.0, 1.0 );
glBegin(GL_LINE_STRIP);
      glVertex3f( 0.1, 0.2, 0 );
      glVertex3f( 0.1, 0.9, 0 );
      glVertex3f( 0.9, 0.9, 0 );
      glVertex3f( 0.9, 0.2, 0 );
glEnd();
```



Line Attributes

- Color, thickness, stippling.
- glColor3f() sets color.
- glLineWidth(4.0) sets thickness. The default thickness is 1.0.

a). thin lines b). thick lines c). stippled lines



Stippling

- Stippling means to add a pattern to a simple line or the filling of a polygon.
- OpenGL allows stippling to be performed using bit patterns.
- Turn stippling on with:
  - glEnable(GL_LINE_STIPPLE);
  - glEnable(GL_POLY_STIPPLE);
- Turn off with:
  - glDisable(GL_LINE_STIPPLE);
  - glDisable(GL_POLY_STIPPLE);

Line Stippling

- Defining a line stippling pattern:
  - `glLineStipple(GLint factor, GLushort pattern);`
  - The pattern is a 16 bit sequence of 1s and 0s
    - e.g. 1110111011101110
  - The factor is a bit multiplier for the pattern (it enlarges it)
    - e.g. factor = 2 turns the above pattern into:
    - 11111100111111001111110011111100
  - The pattern can be expressed in hexadecimal notation
    - e.g. 0xEECC = 1110111011001100
- e.g. `glLineStipple(2, 0x7733);`

Wait a minute…..

- How do I convert binary to hexadecimal?
- Hexadecimal character

  equivalents to 4 bit binary expressions

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

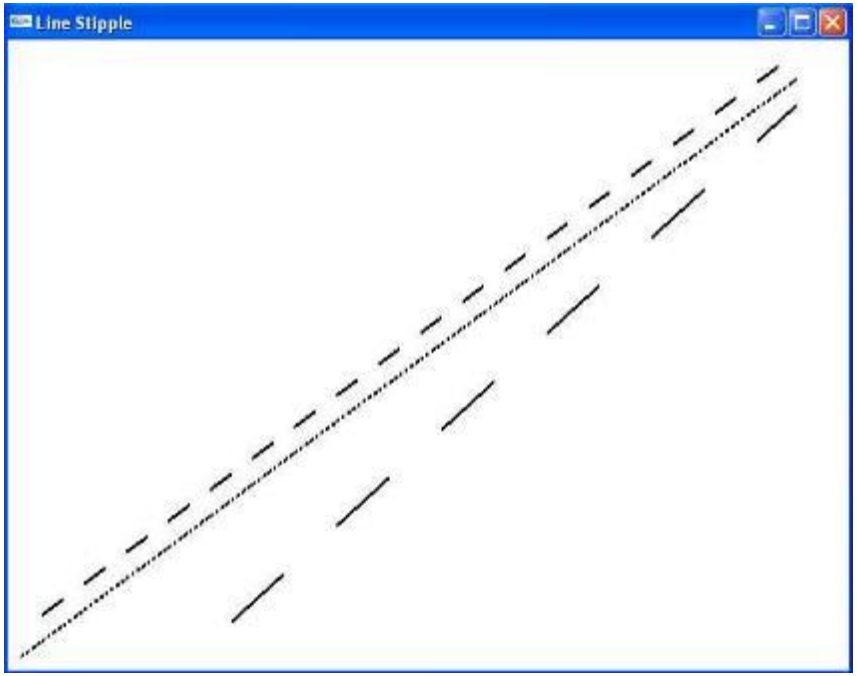
2. Split the binary into groups of **four** digits and assign hex values

0100 0010 = 42
1110 1001 = E9
0001 1100 0011 = 1C3

3. Then put 0x in front of the number in your code, e.g. 0xE9

Line Stippling

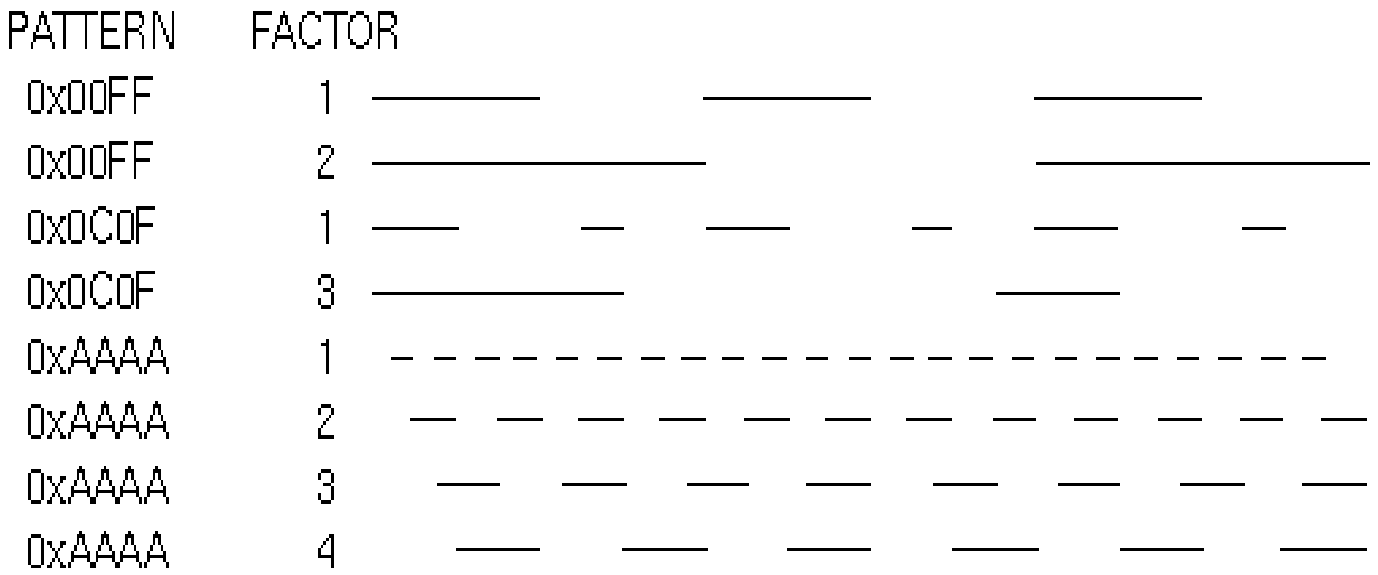
- Lets see it in action…




```
glEnable(GL_LINE_STIPPLE);
glLineStipple(1, 0x7733);
glBegin(GL_LINE_STRIP);
    glVertex2i(10,10);
    glVertex2i(600,450);
glEnd();

glLineStipple(2, 0xFF00);
glBegin(GL_LINE_STRIP);
     glVertex2i(10,30);
     glVertex2i(600,470);
glEnd();

glLineStipple(5, 0xFF00);
glBegin(GL_LINE_STRIP);
     glVertex2i(130,30);
     glVertex2i(600,430);
glEnd();

glFlush();
glDisable(GL_LINE_STIPPLE);
```

Example glLineStipple(1, 0x3F07);

- The above example and the pattern 0x3F07 (which translates to 0011111100000111 in binary), a line would be drawn with 3 pixels on, then 5 off, 6 on, and 2 off. (If this seems backward, remember that the low-order bits are used first.) If *factor* had been 2, the pattern would have been elongated: 6 pixels on, 10 off, 12 on, and 4 off. Figure 4.1 shows lines drawn with different patterns and repeat factors. If you don't enable line stippling, drawing proceeds as if *pattern* were 0xFFFF and *factor* 1. (Use **glDisable()** with GL_LINE_STIPPLE to disable stippling.) Note that stippling can be used in combination with wide lines to produce wide stippled lines.

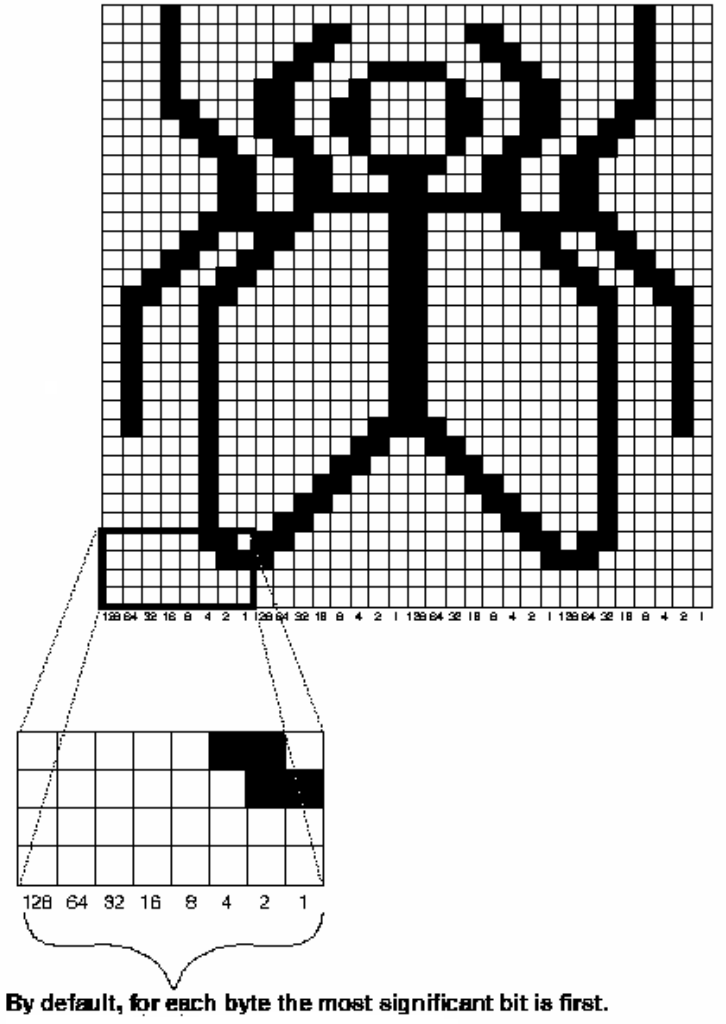


Polygon Stippling

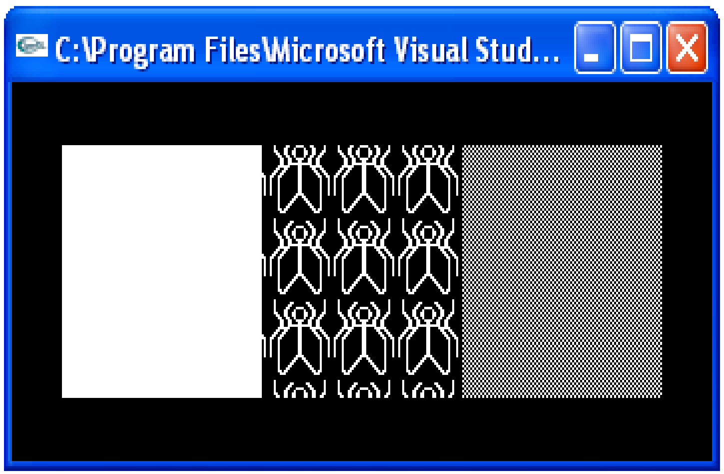
- Defining a polygon stippling pattern:
  - **`glPolygonStipple(GLubyte mask);`**
  - The pattern is a 128 byte array of 1s and 0s (32 bits across and 32 bits down)
    - e.g. GLubyte mask[ ] = {0xff, 0xfe, 0x34, ..};
  - The pattern is tiled inside the polygon.
- e.g. **`glPolygonStipple(mask);`**
- Polygon stippling is enabled and disabled by using **glEnable( )** and **glDisable( )** with GL_POLYGON_STIPPLE as an argument.

Polygon Stippling

- The argument *mask* is a pointer to a 32 x 32 bitmap that's interpreted as a mask of 0s and 1s. Where a 1 appears, the corresponding pixel in the polygon is drawn, and where a 0 appears, nothing is drawn. Figure 4.2 shows how a stipple pattern is constructed from the characters in *mask*.

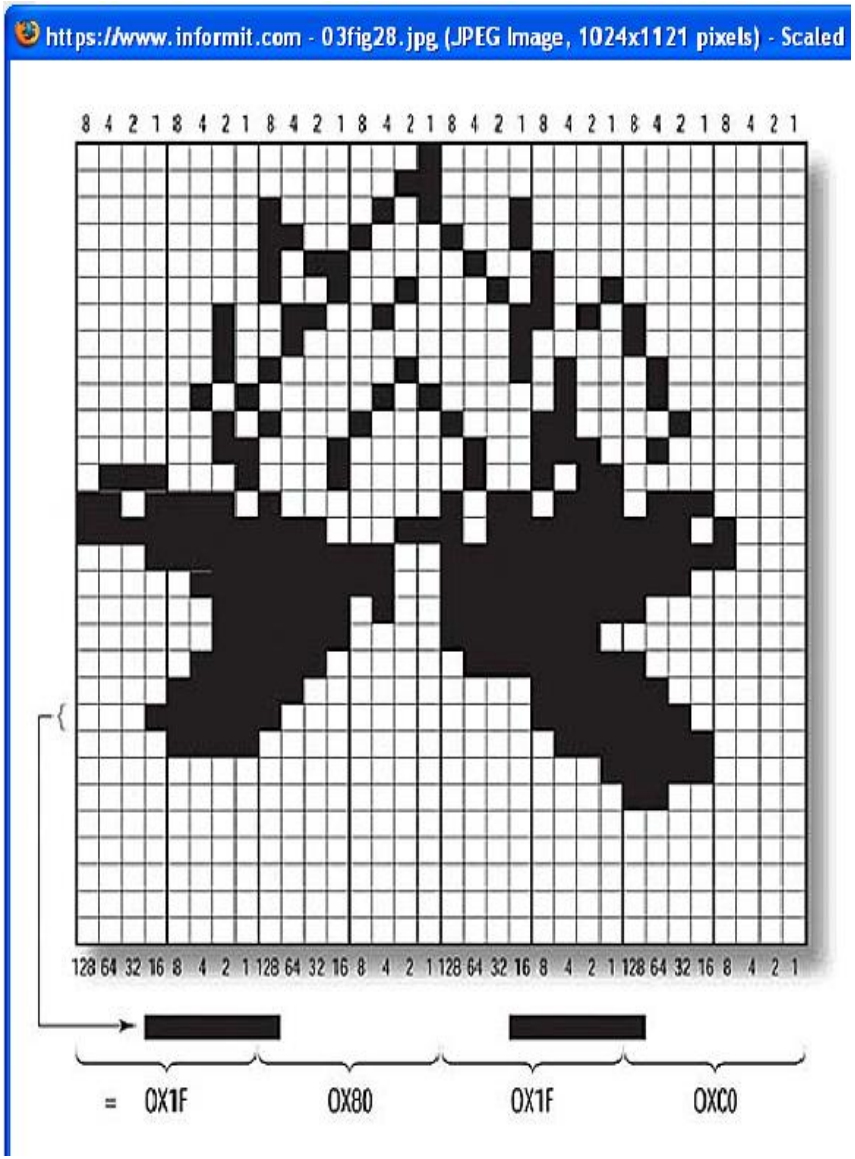


Polygon Stippling cont’d cg7



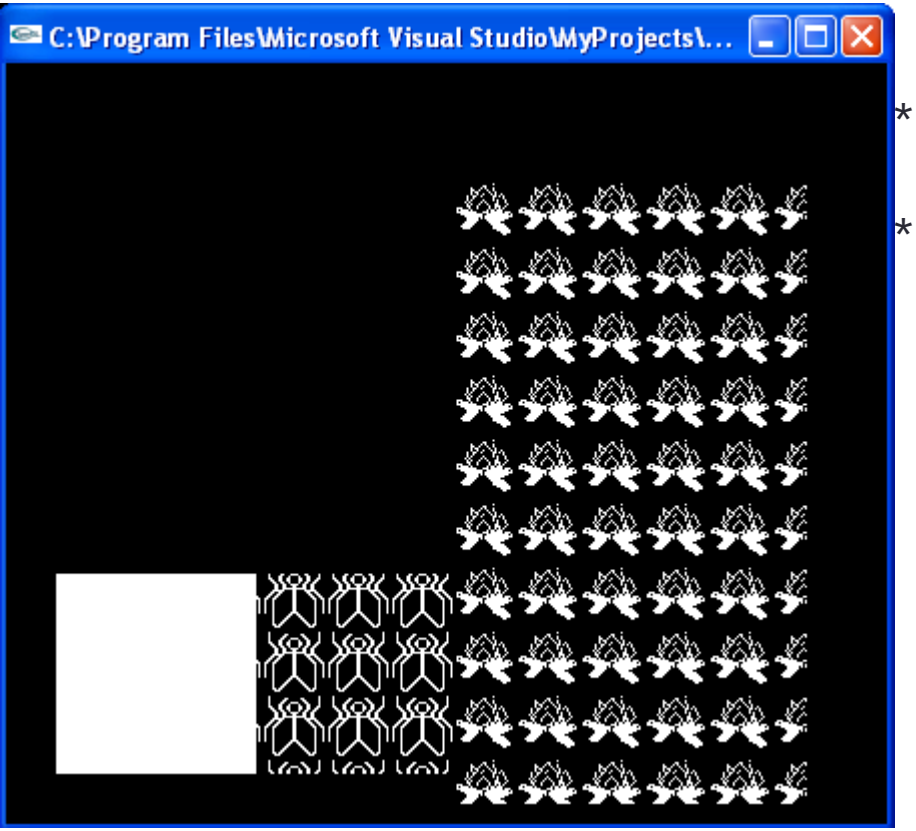

```
GLubyte fly[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
    0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
    0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
    0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
    0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
    0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
    0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
    0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
    0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
    0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
    0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};

GLubyte halftone[] = {
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55};
```

Example 2 - Polygon Stippling




```
// Bitmap of campfire
GLubyte fire[] = { 0x00, 0x00, 0x00, 0x00,
           0x00, 0x00, 0x00, 0x00,
           0x00, 0x00, 0x00, 0x00,
           0x00, 0x00, 0x00, 0x00,
           0x00, 0x00, 0x00, 0x00,
           0x00, 0x00, 0x00, 0xc0,
           0x00, 0x00, 0x01, 0xf0,
           0x00, 0x00, 0x07, 0xf0,
           0x0f, 0x00, 0x1f, 0xe0,
           0x1f, 0x80, 0x1f, 0xc0,
           0x0f, 0xc0, 0x3f, 0x80,
           0x07, 0xe0, 0x7e, 0x00,
           0x03, 0xf0, 0xff, 0x80,
           0x03, 0xf5, 0xff, 0xe0,
           0x07, 0xfd, 0xff, 0xf8,
           0x1f, 0xfc, 0xff, 0xe8,
           0xff, 0xe3, 0xbf, 0x70,
           0xde, 0x80, 0xb7, 0x00,
           0x71, 0x10, 0x4a, 0x80,
           0x03, 0x10, 0x4e, 0x40,
           0x02, 0x88, 0x8c, 0x20,
           0x05, 0x05, 0x04, 0x40,
           0x02, 0x82, 0x14, 0x40,
           0x02, 0x40, 0x10, 0x80,
           0x02, 0x64, 0x1a, 0x80,
           0x00, 0x92, 0x29, 0x00,
           0x00, 0xb0, 0x48, 0x00,
           0x00, 0xc8, 0x90, 0x00,
           0x00, 0x85, 0x10, 0x00,
           0x00, 0x03, 0x00, 0x00,
           0x00, 0x00, 0x10, 0x00 };
```

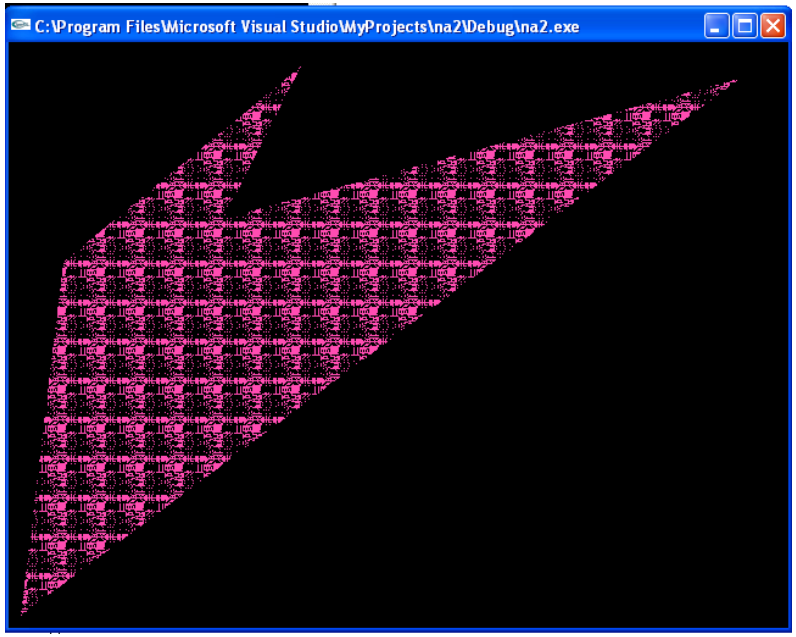
Example 2 - output




```
*  draw one solid, unstippled rectangle,          */

*  then two stippled rectangles                   */
glRectf (25.0, 25.0, 125.0, 125.0);
glEnable (GL_POLYGON_STIPPLE);
glPolygonStipple (fly);
glRectf (125.0, 25.0, 225.0, 125.0);
glPolygonStipple (fire);
glRectf (225.0, 10.0, 400.0, 325.0);
glDisable (GL_POLYGON_STIPPLE);
```

Polygon Stippling

- Lets see it in action…




```
glEnable(GL_POLYGON_STIPPLE);

GLubyte mask[] = {0x31, 0xfe, 0x34, 0x12,
                  0xff, 0xfc, 0x00, 0x12,
                  0xaa, 0xfe, 0x00, 0x12,
                  0xaa, 0xfe, 0x00, 0x12,
                  0xfc, 0xfe, 0x00, 0x12,
                  0xff, 0xfe, 0x00, 0x12,
                   …};
glPolygonStipple(mask);
glBegin(GL_POLYGON);
      glVertex2i(10,10);
      glVertex2i(600,450);
      glVertex2i(45,300);
      glVertex2i(240,460);
glEnd();

glFlush();
glDisable(GL_POLYGON_STIPPLE);
```

Typical OpenGL program structure

1. Create a window and bind OpenGL to this window
   - OpenGL API calls draw/render within this window
     - what to use to create the window?
     - what to use to interact with the graphics?

2. Register event handlers (call-back functions)

3. Set up drawing canvas and coordinate system

4. Prepare the canvas: set up OpenGL states

5. Loop:
   - clear framebuffer
   - perhaps change the screen mapping
   - or change the coordinate system or projection matrix
   - set up lights, camera
   - draw primitives
   - complete drawing

Drawing primitives options

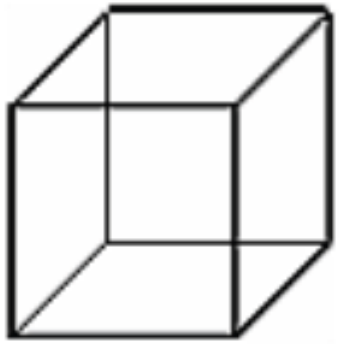
```
// Sample drawing function
void display_triangle() {
   glBegin( GL_TRIANGLES );
      glColor3f( 0.0f, 0.0f, 1.0f ); // sets color to blue
      glVertex2f( 0.0f, 0.0f ); // draw a vertex at 0,0
      glColor3f( 1.0f, 0.0f, 0.0f ); // sets color to red
      glVertex2f( 0.0f, 1.0f ); // draw a vertex at 0,1
      // this will use same color as previous vertex
      glVertex2f( 1.0f, 0.0f );
   glEnd();
}
```

glBegin/glEnd

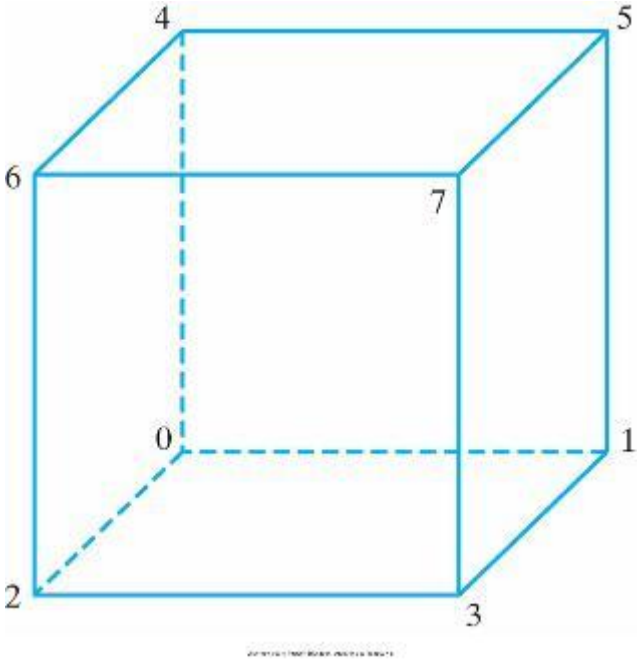
- Pros:
  - Very simple to use for simple objects.
  - Can easily specify attributes.
- Cons:
  - Very cumbersome to specify many objects
  - Code is verbose.
  - Extremely slow.
    - Several function calls per-vertex, which add up really fast.
    - Have to send all vertex data from CPU to GPU every frame.

Vertex arrays (drawing primitives options)

- Major downside of glBegin/End: for anything but simple models, we need a ton of function calls.
  - Consider a cube. Each vertex needs to be declared three times, once for each face it is a part of, resulting in 24 `glVertex` calls.
  - This gets even worse for models with thousands of vertices!



Subscript values for array pt corresponding to the vertex coordinates for the cube




```
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) ;
}
```

```
Glint points [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) } ;
```

OR

```
typedef  Glint vertex3 [3] ;
vertex3 pt [8] = { 0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 0),
((0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 1) } ;

void quad (Glint n1, Glint n2, Glint n3, Glint n4)
{
    glBegin (GL_QUADS);
      glVertex3iv ( pt [n1] );
      glVertex3iv ( pt [n2] );
      glVertex3iv ( pt [n3] );
      glVertex3iv ( pt [n4] );
    glEnd ( );
}
```

Reducing the number of function calls

- 1. Invoke the function **glEnableClientState (GL_VERTEX_ARRAY)** to activate the vertex array feature of OpenGL.
- 2. Use the function **glVertexPointer** to specify the location and data format for the vertex coordinates.
- 3. Display the scene using a routine such as **glDrawElements**, which can process multiple primitives with very few function calls.

```
glEnableClientState (GL_VERTEX_ARRAY) ;
glVertexPointer (3, GL_INT, 0, pt) ;

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) ;
glDrawElements (GL_QUADS, 24, GL_UNSIGNED_BYTE, vertIndex) ;

glDisableClientState (GL_VERTEX_ARRAY );
```

Vertex arrays

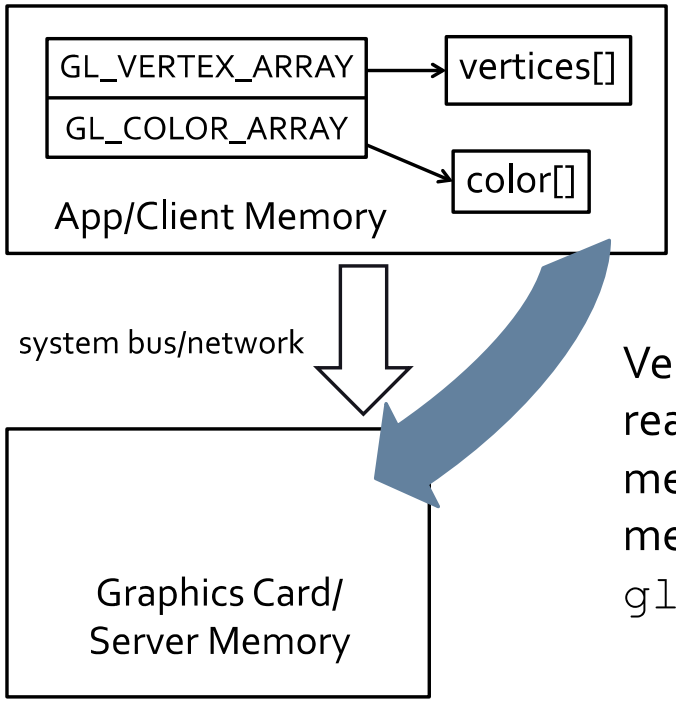
- OpenGL vertex arrays allow you to specify vertex data using arrays and few function calls.
  - Place all attribute data into an array.
  - Render the entire set of primitives at once.

Vertex arrays

Even with triangle strips, passing each vertex to OpenGL requires a separate function call

Vertex arrays allow for passing an array of vertices to OpenGL with a constant number of function calls

- store vertex data in triangle strip sequentially in application/client-side memory
- pass pointer to this memory to the API
- the API copies the data from memory to GPU/server



Vertex attribute data read from client memory to server memory whenever `glDraw*()` is called

Vertex arrays

- Several ways to draw primitives:
  - `glArrayElement`
    - Draws a single vertex
  - `glDrawArrays`
    - Draws a sequence of vertices
  - `glDrawElements`
    - Draws a sequence of vertices based on an indexed array.
    - Generally you want to use this one.

Vertex arrays

```
// Sample code using vertex arrays
void display_triangle() {
   float vertices[] = { 1.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 0.0f
                        0.0f, 0.0f, 1.0 };
   glEnableClientState( GL_VERTEX_ARRAY );
   // specify where to get vertex data
   glVertexPointer( 3, GL_FLOAT, 0, vertices );

   unsigned int indices[] = { 0, 1, 2 };
   // this is the actual draw call
   glDrawElements( GL_TRIANGLES, 3,
                   GL_UNSIGNED_INT, indices );
   glDisableClientState( GL_VERTEX_ARRAY );
}
```

Another example

```
float vertices[]= { 1.0, 0.0, 0.0,
                    0.0, 2.0, 1.0,
                    0.0, 1.0, 0.0,
                    0.0, 0.0, 1.0 };
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// void glVertexPointer(Glint size, GLenum type,
//                      GLsizei stride, const GLvoid *pointer);
// size is the number of coordinates per vertex; type specifies the
// data type; stride is the byte offset between vertices (in bytes, e.g.,
// (3*sizeof(float) for 1-vertex stride), stride=0 if the vertices
// are packed back-to-back
// pointer points to array
```

Another example – using the array

With random access (in display list):

```
glBegin(GL_TRIANGLES);
  glArrayElement(1);
  glArrayElement(0);
  glArrayElement(2);
glEnd();
```

With indexed access (not between `glBegin/glEnd`):

```
unsigned char indices[]= { 1, 0, 2 };
glDrawElements(GL_TRIANGLES, 3,
                  GL_UNSIGNED_BYTE, indices);
// void glDrawElements(GLenum mode, GLsizei count,
//                     GLenum type, void *indices);
```

// `mode` is connection type, `count` size of index array,

// `type` the data type of the index array, choose

// smallest representation necessary, `indices` the array of indices

Vertex arrays

- Pros:
  - Still quite easy to use.
  - Requires only a bit of setup.
  - Much faster than Begin/End.
- Cons:
  - For client-side vertex arrays, still pretty slow.
    - Still have to send all the vertex data from CPU to GPU every frame.

Display lists

- Display lists provide a way for OpenGL to redraw arbitrary primitives with a single call.
- Display lists compile a set of commands that draw a particular object.
- Only work with completely static geometry.

Display lists

```
int list;

// Some method called during initialization
void initialize_triangle (){
  list = glGenLists( 1 );
  glNewList( list, GL_COMPILE ); // starts the list
     glBegin( GL_TRIANGLE );
           glVertex2f( 0.0, 0.0 );
           glVertex2f( 0.0, 1.0 );
           glVertex2f( 1.0, 1.0 );
     glEnd();
  glEndList(); // finishes the list
}

// Sample drawing function
void display_callback(){
  glCallList( list ); // renders the compiled list
}
```

Display lists

- Pros:
  - Very fast, sometimes fastest of any method.
- Cons:
  - Only works with unchanging geometry.
  - Can consume a lot of GPU memory.

Depth sorting

- OpenGL uses a depth test to determine which primitives go in front of others.
  - By default, all primitives are drawn on top.
- Must explicitly enable `GL_DEPTH_TEST` to get depth testing.