Realtime 3D Computer Graphics & Virtual Reality

Geometry and OpenGL

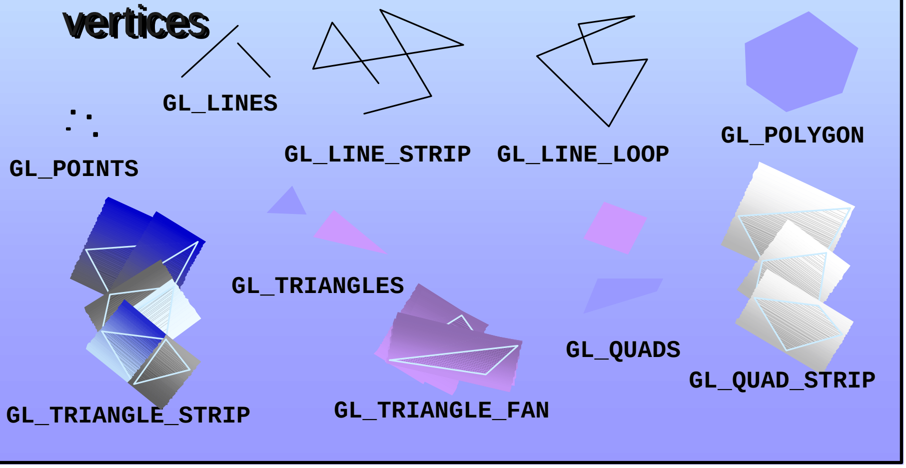
Basic OpenGL template

```
/* simple program template for
   OpenGL progs */

#include <GL/glut.h>

void myDisplay()
{
   /* clear the window */
   glClear(GL_COLOR_BUFFER_BIT);
   /* draw something */
   glBegin(GL_LINES);
        glVertex2f(-0.5, -0.5);
        glVertex2f(0.5, 0.5);
   glEnd();
   glFlush();
}
```

```
int main (int argc,
           char** argv)
{
   glutInit(&argc, argv);
   glutCreateWindow("basic
   template 1");
   glutDisplayFunc(myDisplay);
   glutMainLoop();
}
```

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices



Specifying Geometric Primitives

- Primitives are specified using

  `glBegin( primType );`

  `glEnd();`

  - *primType* determines how vertices are combined

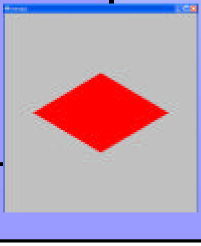
```
GLfloat red, greed, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
  glColor3f( red, green, blue );
  glVertex3fv( coords );
}
glEnd();
```

Simple Example I

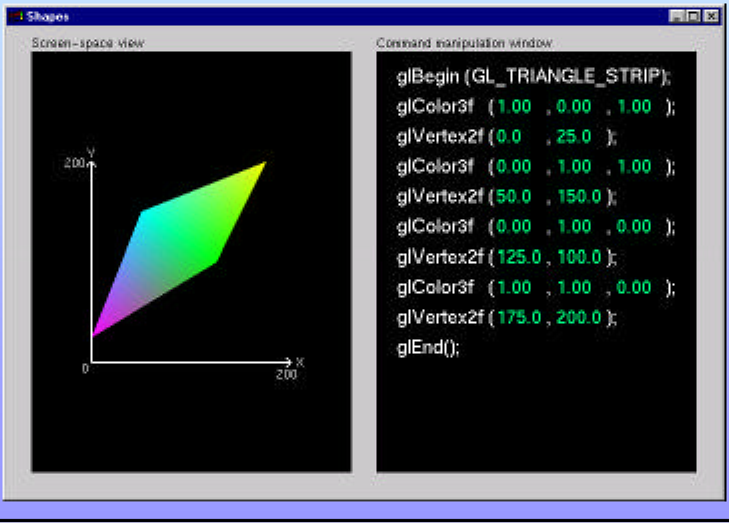
```
void drawLines()
{
    glBegin( GL_LINES );
    glColor3fv( 1.00, 1.00, 1.00 );
    glVertex2f( 50.0, 50.0 );
    glVertex2f( 100.0, 100.0 );
    glColor3fv( 1.00, 1.00, 1.00 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}
```

Simple Example II

```
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f(  0.7,  0.0 );
    glVertex2f(  0.0,  0.4 );
    glVertex2f( -0.7,  0.0 );
    glVertex2f(  0.0, -0.4 );
    glEnd();
}
```



Shapes Tutorial



```
glBegin (GL_TRIANGLE_STRIP);
glColor3f  ( 1.00   , 0.00   , 1.00   );
glVertex2f ( 0.0     , 25.0   );
glColor3f  ( 0.00   , 1.00   , 1.00   );
glVertex2f ( 50.0   , 150.0 );
glColor3f  ( 0.00   , 1.00   , 0.00   );
glVertex2f ( 125.0 , 100.0 );
glColor3f  ( 1.00   , 1.00   , 0.00   );
glVertex2f ( 175.0 , 200.0 );
glEnd();
```

Processing Polygons

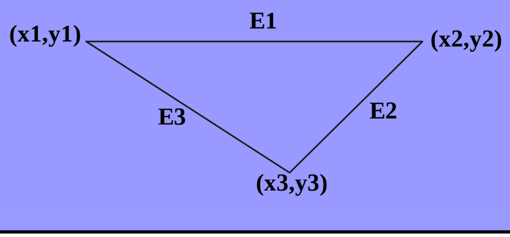
Polygons

- In interactive graphics, polygons rule the world
- Two main reasons:
  - Lowest common denominator for surfaces
    - Can represent any surface *with arbitrary accuracy*
    - Splines, mathematical functions, volumetric isosurfaces…
  - Mathematical simplicity lends itself to simple, regular rendering algorithms
    - Like those we're about to discuss...
    - Such algorithms embed well in hardware

Polygons

- A **polygon** is a many-sided **planar** figure composed of **vertices** and **edges**.
- **Vertices** are represented by points (x,y).
- **Edges** are represented as line segments which connect two points, (x1,y1) and (x2,y2).

$$P = \{ (x_i , y_i) \} \quad i=1,n$$

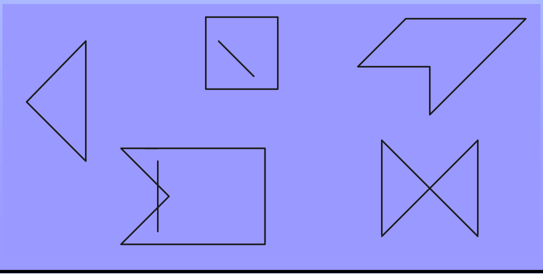


Convex and Concave Polygons

- **Convex Polygon** - For <u>any</u> two points P_1, P_2 inside the polygon, all points on the line segment which connects P_1 and P_2 are inside the polygon.

  - All points $P = uP_1 + (1-u)P_2$, u in [0,1] are inside the polygon provided that P_1 and P_2 are inside the polygon.

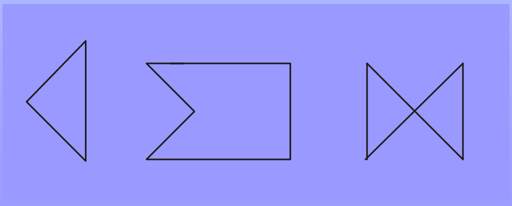
- **Concave Polygon** - A polygon which is not convex.



Simple and non simple Polygons

- **Simple Polygons** – Polygons whose edges do not cross.

- **Non simple Polygons** – Polygons whose edges cross.
  - Two different OpenGL implementations may render non simple polygons differently. OpenGL does not check if polygons are simple.



OpenGL and polygons

- standard primitive – optimized for #polygons/second
- GL_POLYGON, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP (filled)
- GL_LINE_LOOP (unfilled)
- expects planar, convex, non-self-intersecting polygons
- strips and fans are compact, efficient ways to specify lots of simple triangles

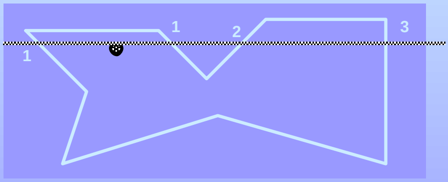
---

Rendering unfilled polygons

- trivial
- simple sequence of line renderings
- requires proper termination of lines at endpoints

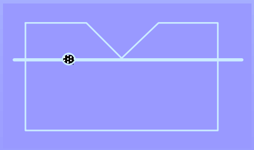
Inside Polygon Test

Inside test: A point P is inside a polygon if and only if a scanline intersects the polygon edges an odd number of times moving from P in either direction.

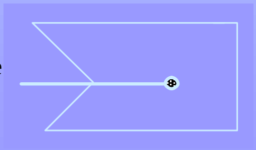


Problem when scan line crosses a vertex:

Does the vertex count as two points?

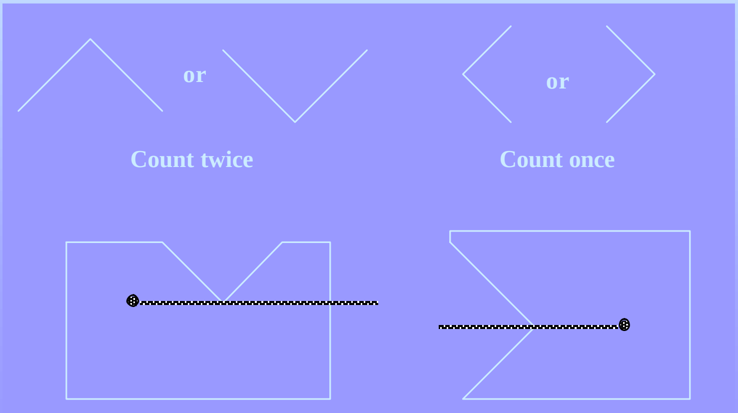


Or should it count as one point?



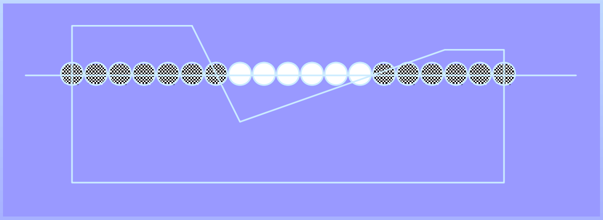
Max-Min Test

When crossing a vertex, if the vertex is a local maximum or minimum then count it twice, else count it once.



Filling Polygons

- **Fill the polygon 1 scanline at a time**



- **Determine which pixels on each scanline are inside the polygon and set those pixels to the appropriate value.**
- **Key idea: Don’t check each pixel for “inside-ness”. Instead, look only for those pixels at which changes occur.**

Scan-Line Algorithm

For each scan-line:

1. Find the intersections of the scan line with all edges of the polygon.
2. Sort the intersections by increasing x-coordinate.
3. Fill in all pixels between pairs of intersections.

Problem:

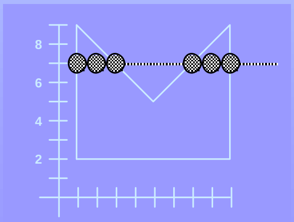
Calculating intersections is slow.

Solution:

Incremental computation / coherence

For scan-line number 7 the sorted list of x-coordinates is (1,3,7,9)

Therefore fill pixels with x-coordinates 1-3 and 7-9.



Edge Coherence

- Observation: Not all edges intersect each scanline.
- Many edges intersected by scanline i will also be intersected by scanline i+1

- Formula for scanline s is y = s, for an edge is y = mx + b
- Their intersection is

  $s = mx_s + b \dashrightarrow x_s = (s-b)/m$

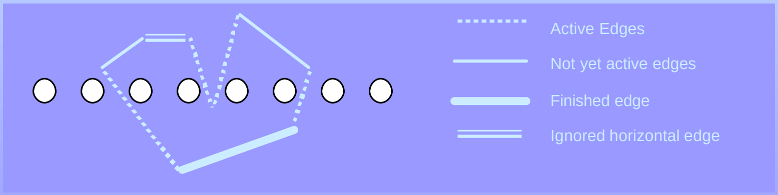
- For scanline s + 1,

  $x_{s+1} = (s+1 - b)/m = x_s + 1/m$

Incremental calculation: $x_{s+1} = x_s + 1/m$

Processing Polygons

- Polygon edges are sorted according to their minimum Y. Scan lines are processed in increasing (upward) Y order. When the current scan line reaches the lower endpoint of an edge it becomes active. When the current scan line moves above the upper endpoint, the edge becomes inactive.

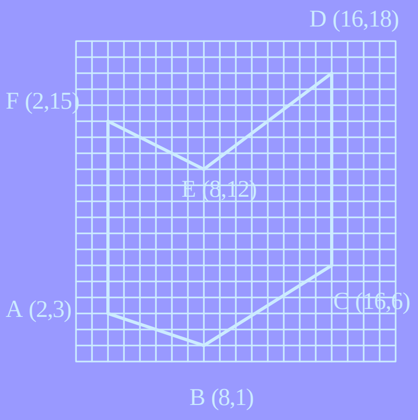


- Active edges are sorted according to increasing X. Filling the scan line starts at the leftmost edge intersection and stops at the second. It restarts at the third intersection and stops at the fourth. . . (<u>spans</u>)

Polygon fill rules (to ensure consistency)

1. Horizontal edges: Do not include in edge table
2. Horizontal edges: Drawn on the bottom, not on the top.
3. Vertices: If local max or min, then count twice, else count once.
4. Vertices at local minima are drawn, vertices at local maxima are not.
5. Only turn on pixels whose centers are *interior* to the polygon: round up values on the left edge of a span, round down on the right edge

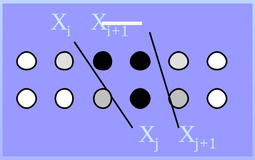
Polygon fill example



19	NULL								
18	NULL								
17	NULL								
16	NULL								
15	NULL								
14	NULL								
13	NULL								
12	->	15	8	-2	->	18	8	4/3	NULL
11	NULL								
10	NULL								
9	NULL								
8	NULL								
7	NULL								
6	->	18	16	0	NULL				
5	NULL								
4	NULL								
3	->	15	2	0	NULL				
2	NULL								
1	->	3	8	-3	->	6	8	8/5	NULL

Antialiasing Polygons

- Polygon edges suffer from aliasing just as lines do. If an edge passes between two pixels, they share the intensity. The same method can be used on the scan line fill.

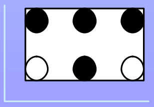


- The fill begins at the leftmost edge intersection. If the intersection is between two pixels $X_i < X < X_{i+1}$ then pixel X_i is assigned the intensity $(X_{i+1} - X)$. Pixel X_i is assigned intensity 1.0 (unless the polygon is very narrow).
- At the second intersection, where filling stops, the reverse is true. $X_j < X < X_{j+1}$ Pixel X_j is assigned intensity 1.0 and X_{j+1} is assigned $(X - X_j)$.

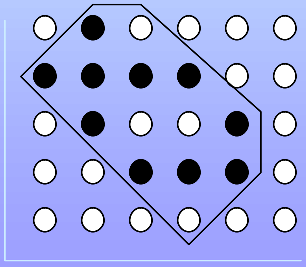
Fill Patterns

Fill patterns can be used to put a noticeable texture inside a polygon. A fill pattern can be defined in a 0-based, *m* x *n* array. A pixel (*x*,*y*) is assigned the value found in:

pattern((x mod m), (y mod n))



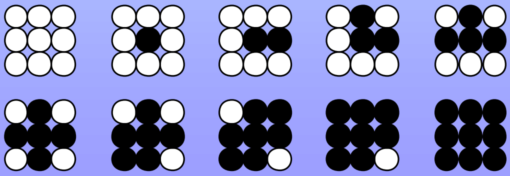
Pattern



Pattern filled polygon

Halftoning

- For bitmapped displays, fill patterns with different fill densities can be used to vary the range of intensities of a polygon. The result is a tradeoff of resolution (addressability) for a greater range of intensities and is called *halftoning*. The pattern in this case should be designed to *avoid* being noticed.
- These fill patterns are chosen to minimize banding.



Polygons in OpenGL

- Colors of polygons, shading
- Sides of polygons
- Styles of Drawing
- How to structure geometry (e.g. polygons)
- An alternative way for “packing” OGL commands

Simple shading

- We can specify color for each vertex

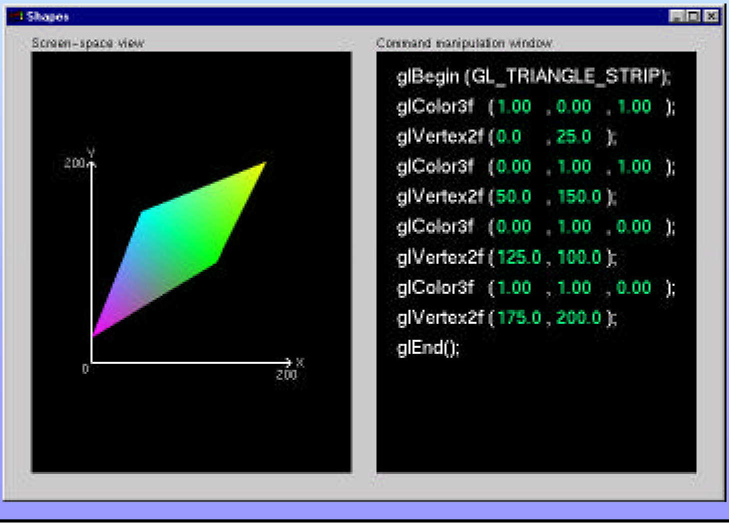
What happens if the colors are different?

- OpenGL interpolates between two points and between the lines (->*bilinear interpolation*) of different color if shading is smooth (default)!

- **Shademodel**

```
      glShadeModel( GLenum mode );
mode = GL_SMOOTH, GL_FLAT
```

Shapes Tutorial



```
glBegin (GL_TRIANGLE_STRIP);
glColor3f  (1.00  , 0.00  , 1.00  );
glVertex2f (0.0   , 25.0  );
glColor3f  (0.00  , 1.00  , 1.00  );
glVertex2f (50.0  , 150.0 );
glColor3f  (0.00  , 1.00  , 0.00  );
glVertex2f (125.0 , 100.0 );
glColor3f  (1.00  , 1.00  , 0.00  );
glVertex2f (175.0 , 200.0 );
glEnd();
```

Polygons in OpenGL

- Polygons can be drawn in three different ways:
  - (1) points (vertices, see glVertex2f(10.0,10.0)), (2) edges, (3) filled
- The two dimensional examples are just special cases of three dimensional polygons with `z=0`. Therefore polygons have two faces:
  - front face: order of vertices is counterclockwise
  - back face: order of vertices is clockwise

  that can be changed by:

```
glFrontFace( GLenum mode );
mode = GL_CCW, GL_CW
```

Polygons in OpenGL

- **Which faces are to be rendered can be controlled by OpenGL states:**

```
glPolygonMode( GLenum face, Glenum mode );
face = GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
mode = GL_POINT, GL_LINE, GL_FILL
```

```
glCullFace( Glenum mode );
mode = GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
```

```
glEnable( GL_CULL_FACE );
```

Polygons in OpenGL

- How can we render a polygon in different styles simultaneously?

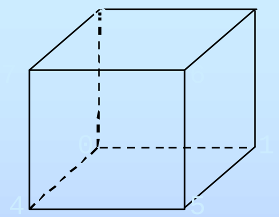
  Just draw it multiple times:

```
glPolygonMode(GL_FILL);
glColor3fv(yellow);
drawGeometry();
glPolygonMode(GL_LINE);
glColor3fv(red);
drawGeometry();
```

Structuring of geometry I

```
/* simple drawing vertex for vertex */
drawCube1()
{
   /* draw the first side of the cube */
   glColor3f(1.0, 0.0, 0.0);
   glBegin(GL_POLYGON);
        glVertex3f(-1.0, -1.0, -1.0);
        glVertex3f(-1.0, 1.0, -1.0);
        glVertex3f(-1.0, 1.0, 1.0);
        glVertex3f(-1.0, -1.0, 1.0);
   glEnd();
   /* draw the second side of the cube */
   glColor3f(0.0, 1.0, 0.0);
   glBegin(GL_POLYGON);
        ...
   glEnd();
```

Structuring of geometry II



```
/* put data in structs */
   GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
   {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
   {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

   GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
   {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
   {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

   GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
   {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
   {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

Structuring of geometry II

```
void polygon(int a, int b, int c , int d)
{
/* draw a polygon via list of vertices */
   glBegin(GL_POLYGON);
       glColor3fv(colors[a]);
       glNormal3fv(normals[a]);
       glVertex3fv(vertices[a]);
       glColor3fv(colors[b]);
       glNormal3fv(normals[b]);
       glVertex3fv(vertices[b]);
       …
   glEnd();
}
```

Structuring of geometry II

```
void drawCube2(void)
{
/* map vertices to faces */
  polygon(0,3,2,1);
  polygon(2,3,7,6);
  polygon(0,4,7,3);
  polygon(1,2,6,5);
  polygon(4,5,6,7);
  polygon(0,1,5,4);
}
```

Structuring of geometry III

- Vertex arrays
  - Avoid most of the calls to draw the cube
  - store the data in the application program
  - Access data by single function call
- OpenGL supports *six* types of arrays (not only for vertex data)
- Must be enabled (e.g., using init())

Structuring of geometry III

- Vertex arrays must be enabled

```
glEnableClientState( Glenum array );
glDisableClientState( Glenum array );
```

```
array =

     GL_VERTEX_ARRAY, GL_COLOR_ARRAY,
     GL_INDEX_ARRAY, GL_NORMAL_ARRAY,
     GL_TEXTURE_ARRAY, GL_EDGE_FLAG_ARRAY
```

Structuring of geometry III

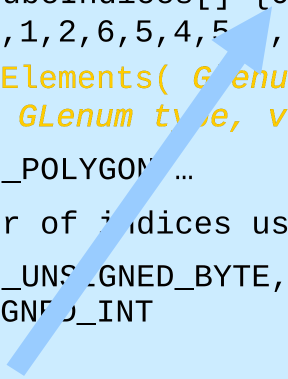
- Vertex arrays must be initialized to tell OpenGL the array structure

```
glVertexPointer( Glint dim, GLenum type,
     GLsizei stride, GLvoid* array );
glColorPointer( s.o. );
…
```

```
dim = 1, 2, 3

type = GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE

stride = number of bytes between consecutive
  data values

array = pointer to data
```

Structuring of geometry III

- Ordering and displaying of data in vertex arrays



```
GLubyte cubeIndices[]={0,3,2,1,2,3,7,6,
  0,4,7,3,1,2,6,5,4,5,[illegible],7,0,1,5,4};
```

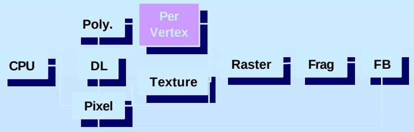
```
   glDrawElements( GLenum mode, GLisizei n,
          GLenum type, void*indices );
```

```
mode = GL_POLYGON …

n = number of indices used

type = GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT,
  GL_UNSIGNED_INT

indices =
```

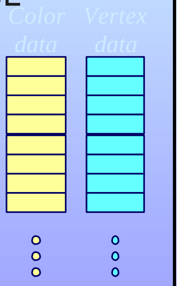
Structuring of geometry III

```
init()
{
   /* do what ever */
   glEnableClientState(GL_COLOR_ARRAY);
   glEnableClientState(GL_VERTEX_ARRAY);
   glVertexPointer(3, GL_FLOAT, 0, vertices);
   glColorPointer(3,GL_FLOAT, 0, colors);
   /* fini for the vertex array stuff */
}
drawCube()
{
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE,
   cubeIndices);
}
```

Vertex Arrays



- Pass arrays of vertices, colors, etc. to OpenGL in a large chunk



```
glVertexPointer( 3, GL_FLOAT, 0, coords )
glColorPointer( 4, GL_FLOAT, 0, colors )
glEnableClientState( GL_VERTEX_ARRAY )
glEnableClientState( GL_COLOR_ARRAY )
glDrawArrays( GL_TRIANGLE_STRIP, 0, numVerts
 );
```

- All active arrays are used in rendering

Structuring of commands (geometry IV)

- Two rendering modes in OpenGL
  - Immediate mode
  - Retained mode
- Retained mode is due to the client/server architecture of OpenGL
- data can be *compiled* into **display lists** and *stored* on the server

This feature can be used for fast preprocessing of data

Structuring of commands (display lists)

```
glNewList( GLuint name, GLenum mode );
[glCommands]
glEndList();
```

name = unique integer

mode = GL_COMPILE, GL_COMPILE_AND_EXECUTE

```
glCallList( GLuint name )
glDeleteLists( GLuint first, GLsizei number )
```

Structuring of commands (multiple display lists)

```
glListBase( GLuint offset )
glCallLists( GLiseizei num, GLenum type,
GLvoid *lists );
```

offset = number where to start

num = number of lists to execute

type = type of lists

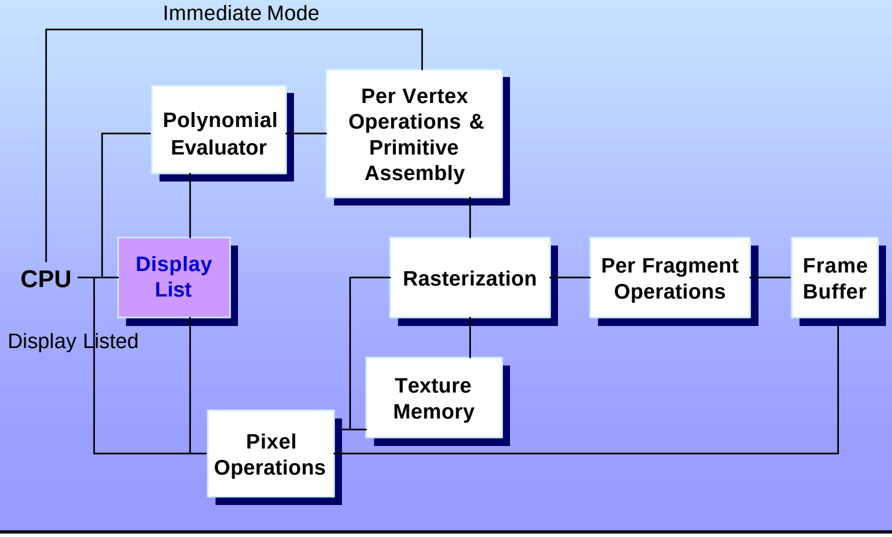
```
GLunint glGenLists( GLsizei n )
```

returns the first of n unused consecutive integers

Immediate Mode versus Display Listed Rendering

- Immediate Mode Graphics
  - Primitives are sent to pipeline and display right away
  - No memory of graphical entities
- Display Listed Graphics
  - Primitives placed in display lists
  - Display lists kept on graphics server
  - Can be redisplayed with different state
  - Can be shared among OpenGL graphics contexts

Immediate Mode versus Display Lists



Immediate Mode

Polynomial Evaluator

Per Vertex Operations & Primitive Assembly

CPU

Display List

Rasterization

Per Fragment Operations

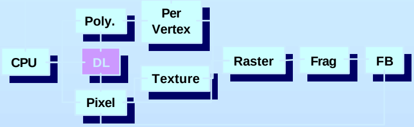
Frame Buffer

Display Listed

Texture Memory

Pixel Operations

Display Lists



- Creating a display list

```
GLuint id;
void init( void )
{
   id = glGenLists( 1 );
   glNewList( id, GL_COMPILE );
   /* other OpenGL routines */
   glEndList();
}
```

- Call a created list

```
void display( void )
{
   glCallList( id );
}
```

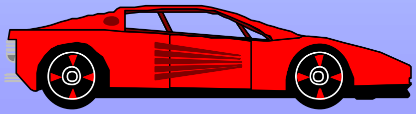
Display Lists

- Not all OpenGL routines can be stored in display lists
- State changes persist, even after a display list is finished
- Display lists can call other display lists
- Display lists are not editable, but you can fake it
  - make a list (A) which calls other lists (B, C, and D)
  - delete and replace B, C, and D, as needed

Display Lists and Hierarchy

- Consider model of a car
  - Create display list for chassis
  - Create display list for wheel

```
glNewList( CAR, GL_COMPILE );
  glCallList( CHASSIS );
  glTranslatef( … );
  glCallList( WHEEL );
  glTranslatef( … );
  glCallList( WHEEL );
      …
glEndList();
```



Why use Display Lists or Vertex Arrays?

- May provide better performance than immediate mode rendering
- Display lists can be shared between multiple OpenGL context
  - reduce memory usage for multi-context applications
- Vertex arrays may format data for better memory access

Structuring data

- **Example and outlook:**
  - The CUBE example

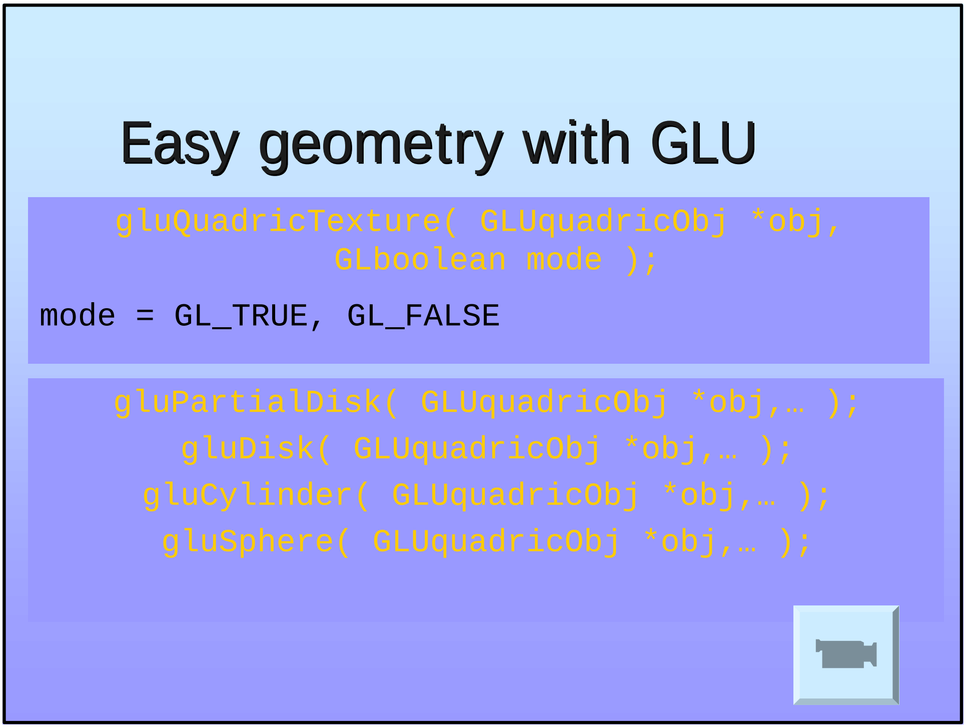
Easy geometry with GLU

```
GLUquadricObj* gluNewQuadric();
gluDeleteQuadric( GLUquadricObj *obj );

gluQuadricDrawStyle( GLUquadricObj *obj,
    GLenum style );
```

```
style = GLU_POINT, GLU_LINE, GLU_FILL,
  GLU_SILHOUETTE
```

```
gluQuadricNormals( GLUquadricObj *obj, GLenum
    mode );
```

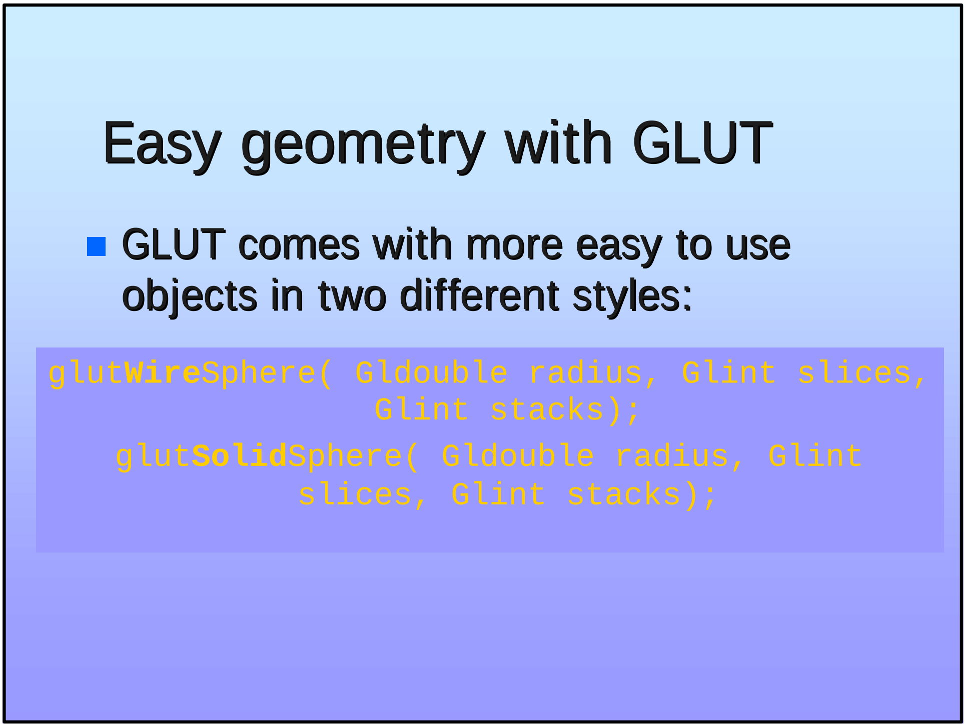
```
mode = GLU_NONE, GLU_FLAT, GLU_SMOOTH
```



Easy geometry with GLU

```
gluQuadricTexture( GLUquadricObj *obj,
                   GLboolean mode );

mode = GL_TRUE, GL_FALSE
```

```
gluPartialDisk( GLUquadricObj *obj,… );
gluDisk( GLUquadricObj *obj,… );
gluCylinder( GLUquadricObj *obj,… );
gluSphere( GLUquadricObj *obj,… );
```



Easy geometry with GLUT

- GLUT comes with more easy to use objects in two different styles:

```
glutWireSphere( Gldouble radius, Glint slices,
                Glint stacks);
glutSolidSphere( Gldouble radius, Glint
                 slices, Glint stacks);
```

Easy geometry with GLUT

```
glutXXXCone(…);
glutXXXTorus(…);
glutXXXTetrahedron();
glutXXXOctahedron();
glutXXXDodecahedron();
glutXXXIcosahedron();
glutXXXTeapot( GLdouble size );
```