CS 4204 Computer Graphics

Meshes, Vertex Array and Displaylist

Yong Cao

Virginia Tech

Objectives

Introduce simple data structures for building polygonal models

- Vertex lists
- Edge lists

OpenGL vertex arrays

Display List in OpenGL

Wavefront OBJ and GLM library (Demo)

Mesh (Triangle meshes)

Why Triangles?

Generality

- Any planar polygon can be triangulated
- When you give OpenGL a different polygon, it may well end up getting triangulated somewhere in the pipeline

Simplicity

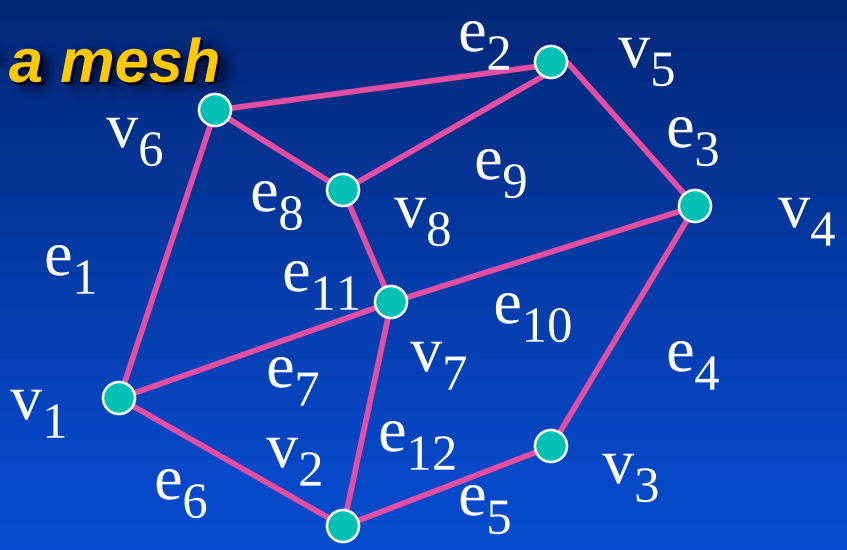
- Triangles have a nice mathematical structure we can exploit
- Impossible to specify a non-planar triangle
- Impossible to specify a non-convex triangle

Efficiency

- By picking a standard primitive, we can design custom graphics hardware to be blisteringly fast
- Convexity makes rasterization much less complex

Representing a Mesh

Consider a mesh



There are 8 nodes and 12 edges

- 5 interior polygons
- 6 interior (shared) edges

Each vertex has a location $v_i = (x_i\ y_i\ z_i)$

Simple Representation

Define each polygon by the geometric locations of its vertices

Leads to OpenGL code such as

```
glBegin(GL_POLYGON);
    glVertex3f(x1, x1, x1);
    glVertex3f(x6, x6, x6);
    glVertex3f(x7, x7, x7);
glEnd();
```

Inefficient and unstructured

- Consider moving a vertex to a new location
- Must search for all occurrences

Geometry vs Topology

Generally it is a good idea to look for data structures that separate the geometry from the topology

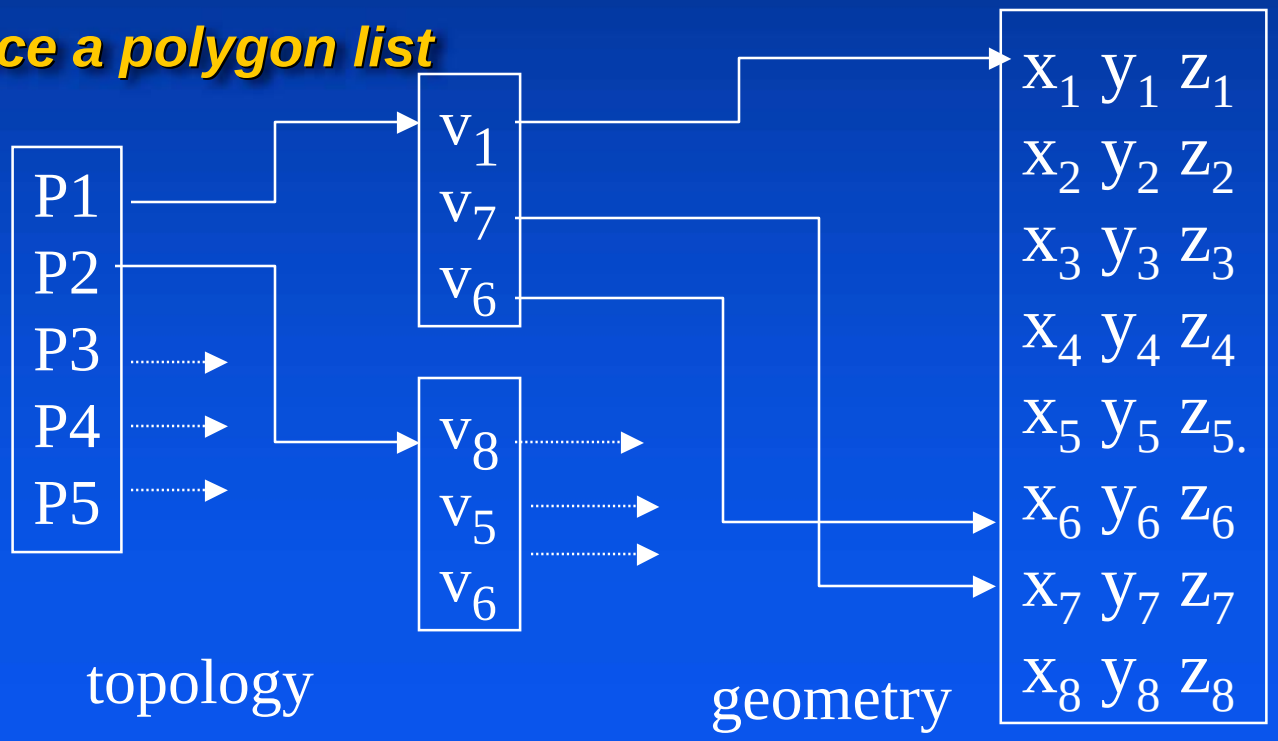
- Geometry: locations of the vertices
- Topology: organization of the vertices and edges
- Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
- Topology holds even if geometry changes

Vertex Lists

Put the geometry in an array

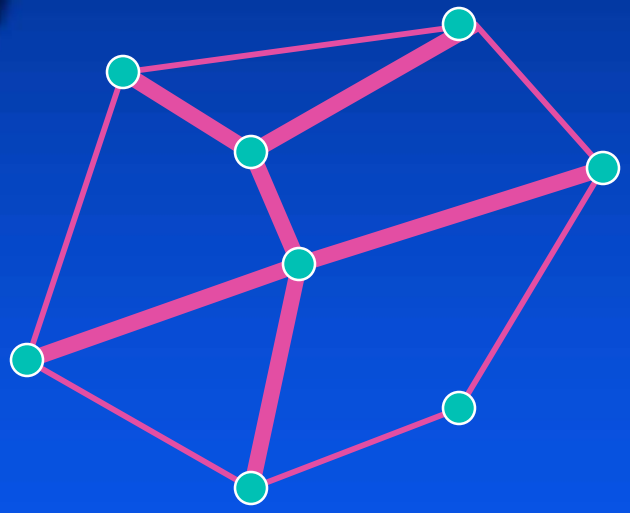
Use pointers from the vertices into this array

Introduce a polygon list



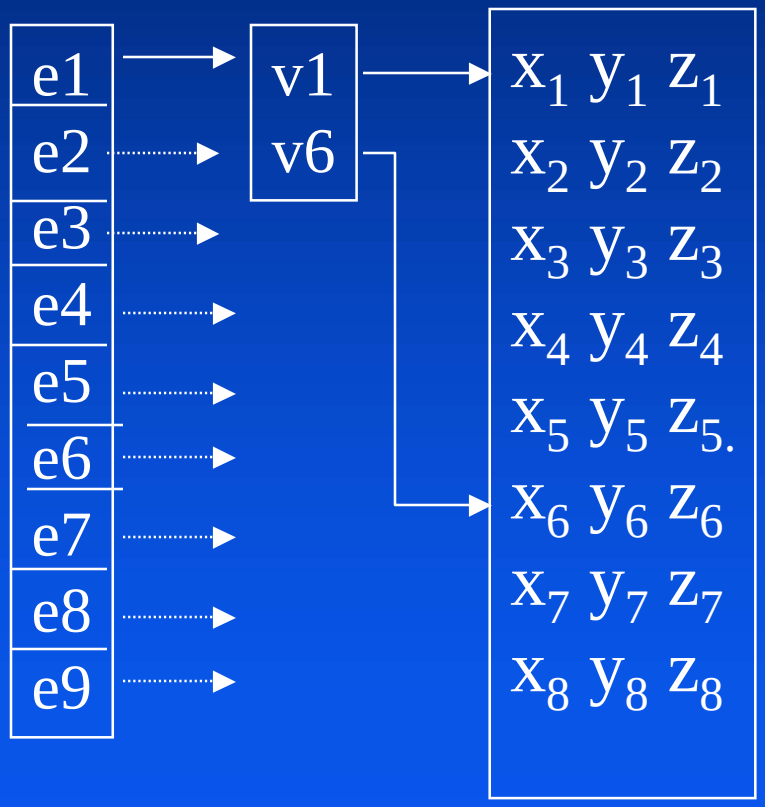
Shared Edges

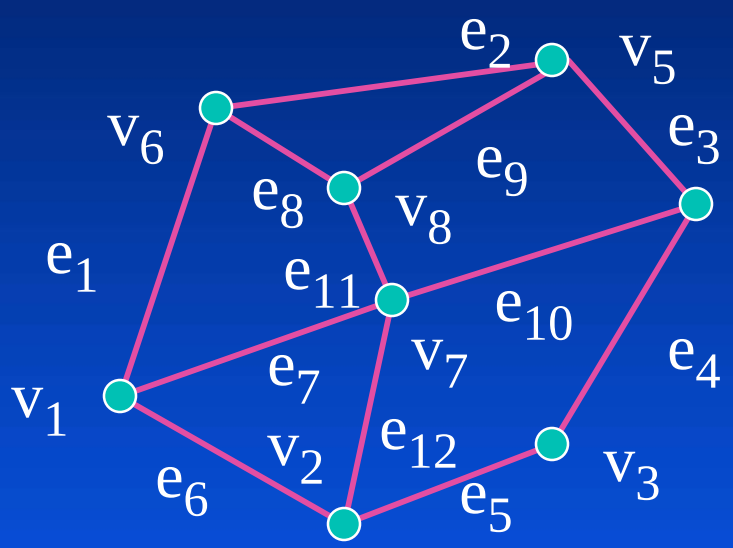
Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



***Can store mesh by* edge list**

Edge List





Note polygons are not represented

Modeling a Cube

Model a color cube for rotating cube program

Define global arrays for vertices and colors

```
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 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}};
```

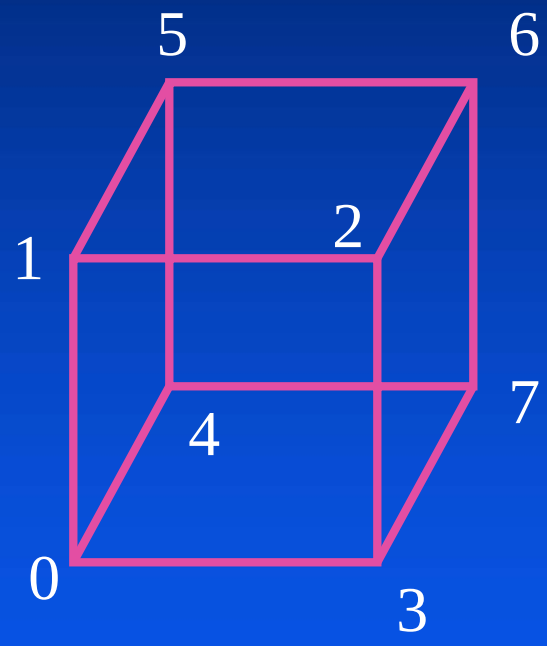
Drawing a polygon from a list of indices

Draw a quadrilateral from a list of indices into the array `vertices` and use color corresponding to first index

```
void polygon(int a, int b, int c
, int d)
{
   glBegin(GL_POLYGON);
      glColor3fv(colors[a]);
      glVertex3fv(vertices[a]);
      glVertex3fv(vertices[b]);
      glVertex3fv(vertices[c]);
      glVertex3fv(vertices[d]);
    glEnd();
 }
```

Draw cube from faces

```
void colorcube( )
{
    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);
}
```



Note that vertices are ordered so that we obtain correct outward facing normals

Efficiency

The weakness of our approach is that we are building the model in the application and must do many function calls to draw the cube

Drawing a cube by its faces in the most straight forward way requires

- 6 **`glBegin`**, 6 **`glEnd`**
- 6 **`glColor`**
- 24 **`glVertex`**
- More if we use texture and lighting

Vertex Arrays

OpenGL provides a facility called* vertex arrays *that allows us to store array data in the implementation

Six types of arrays supported

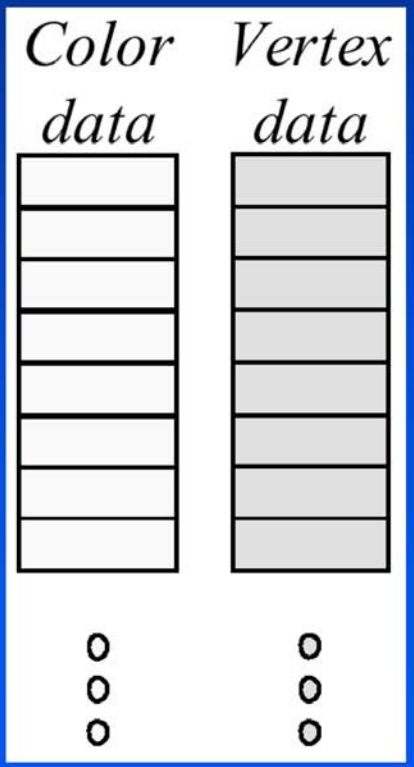
- Vertices
- Colors
- Color indices
- Normals
- Texture coordinates
- Edge flags

We will need only colors and vertices

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

- On: glEnalbleClientState()
- Off: glDisableClientState()

Vertex Arrays

Vertex Arrays allow vertices, and their attributes to be specified in chunks,

- Not sending single vertices/attributes one call at a time.

Three methods for rendering using vertex arrays:

- glDrawArrays(): render specified primitive type by processing *nV* consecutive elements from enabled arrays.
- glDrawElements(): indirect indexing of data elements in the enabled arrays. (shared data elements specified once in the arrays, but accessed numerous times)
- glArrayElement(): processes a single set of data elements from all activated arrays. As compared to the two above, must appear between a glBegin()/glEnd() pair.

Vertex Arrays

glDrawArrays(): draw a sequence

glDrawElements(): methodically hop around

glArrayElement(): randomly hop around

glInterleavedArrays(): advanced call

- can specify several vertex arrays at once.
- also enables and disables the appropriate arrays

Read Chapter 2 in Redbook for details of using vertex array

Initialization

Using the same color and vertex data, first we enable

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

Identify location of arrays

```
glVertexPointer(3, GL_FLOAT, 0, vertices);
```


```
glColorPointer(3, GL_FLOAT, 0, colors);
```

Mapping indices to faces

- ***Form an array of face indices***

```
GLubyte cubeIndices[24] = {0,3,2,1,2,3,7,6
    0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};
```

- ***Each successive four indices describe a face of the cube***
- ***Draw through `glDrawElements` which replaces all `glVertex` and `glColor` calls in the display callback***

Drawing the cube

Method 1:

what to draw

number of indices

```
for(i=0; i<6; i++) glDrawElements(GL_POLYGON, 4,
      GL_UNSIGNED_BYTE, &cubeIndices[4*i]);
```

format of index data

start of index data

Method 2:

```
glDrawElements(GL_QUADS, 24,
     GL_UNSIGNED_BYTE, cubeIndices);
```

Draws cube with 1 function call!!

Immediate Mode vs Display Lists

Immediate Mode Graphics

- Primitives are sent to pipeline and display right away
- No memory of graphical entities

Display Listed Graphics

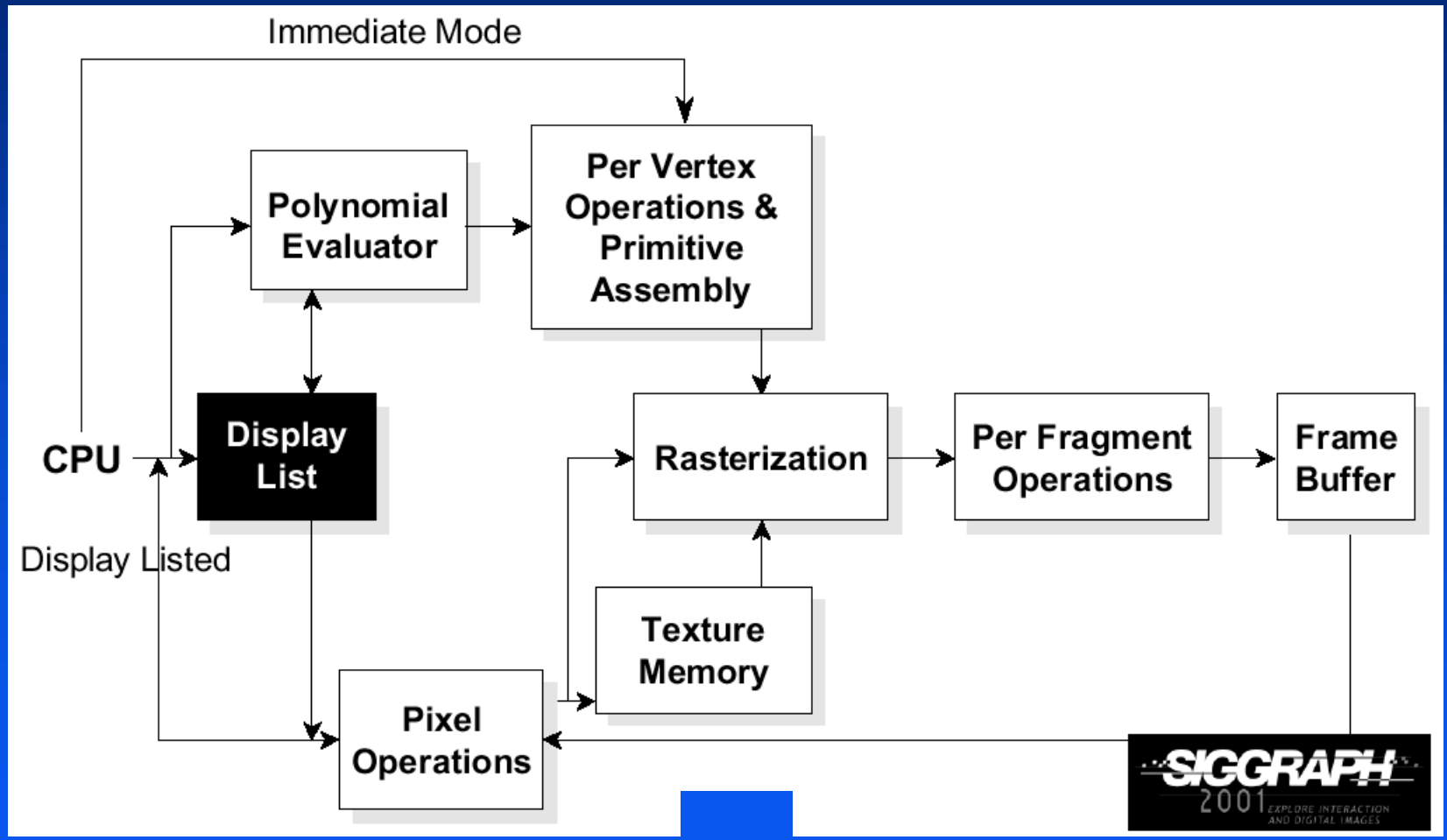
- Primitives placed in display lists
- Display lists kept on graphics server
- Can be redisplayed with different state
- Can be shared among OpenGL graphics contexts (in X windows, use the `glXCreateContext()` routine)

Immediate Mode vs Retained Mode

In immediate mode, primitives (vertices, pixels) flow through the system and produce images. These data are lost. New images are created by reexecuting the display function and regenerating the primitives.

In retained mode, the primitives are stored in a display list (in "compiled" form). Images can be recreated by "executing" the display list. Even without a network between the server and client, display lists should be more efficient than repeated executions of the display function.

Immediate Mode vs Display Lists



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 );
}
```

Instead of `GL_COMPILE`, `glNewList` also accepts constant `GL_COMPILE_AND_EXECUE`, which both creates and executes a display list.

If a new list is created with the same identifying number as an existing display list, the old list is replaced with the new calls. No error occurs.

Display Lists

Not all OpenGL routines can be stored in display lists

- If there is an attempt to store any of these routines in a display list, the routine is executed in immediate mode. No error occurs.

State changes persist, even after a display list is finished

Display lists can call other display lists

Display lists are not editable, but can fake it

- make a list (A) which calls other lists (B, C, and D)
- delete and replace B, C, and D, as needed

Some Routines That Cannot be Stored in a Display List

Some routines cannot be stored in a display list. Here are some of them:

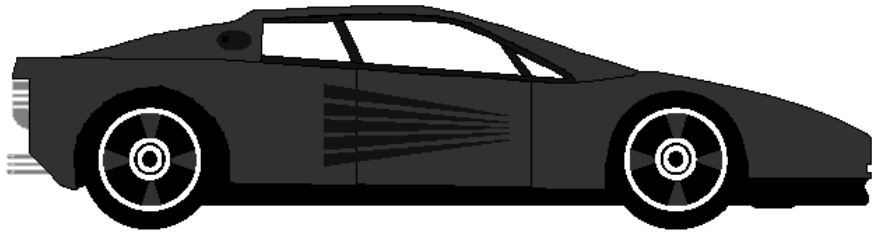
all `glGet*` routines

`glIs*` routines (e.g., `glIsEnabled`, `glIsList`, `glIsTexture`)

`glGenLists`	`glDeleteLists`	`glFeedbackBuffer`
`glSelectBuffer`	`glRenderMode`	`glVertexPointer`
`glNormalPointer`	`glColorPointer`	`glIndexPointer`
`glReadPixels`	`glPixelStore`	`glGenTextures`
`glTexCoordPointer`		`glEdgeFlagPointer`
`glEnableClientState`		`glDisableClientState`
`glDeleteTextures`		`glAreTexturesResident`
`glFlush`	`glFinish`	

An Example

```
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

- Both are principally performance enhancements. On some systems, they may provide better performance than immediate mode because of reduced function call overhead or better data organization
  - *format data for better memory access*
- Display lists can also be used to group similar sets of OpenGL commands, like multiple calls to glMaterial() to set up the parameters for a particular object

Display lists can be shared between multiple OGL contexts

- reduce memory usage or multi-context applications