Lab 02 - Open GL Primitives (Chapter 4)

I. Open GL Immediates (Deprecated open GL feature) - Most inefficient

A. Essentially commands OpenGL to send a small data set through pipeline immediately. Requires collection of data on GPU and single vertex copies over to GPU. No bulk copy.

B. Form:

```
glBegin(GL_*); // Specify Primitive
    glVertex*(*); // Specify vertices + attributes, eg. normals or colors
    :
glEnd();
```

C. Primitives:

i. GL_POINTS specify separate points to be drawn

ii. GL_LINES vertex pairs i and i+1 define lines

iii. GL_LINE_STRIP vertices i and i+1 define lines

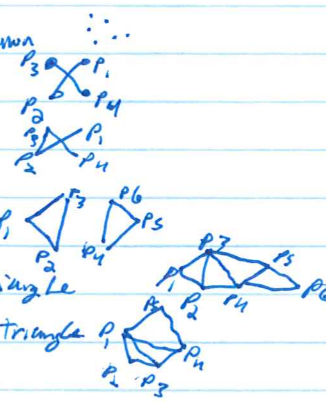
iv. GL_LINE_LOOP closed line strip

v. GL_TRIANGLES Each 3 vertices define triangle

vi. GL_TRIANGLE_STRIP Each new vertex adds another triangle

vii. GL_TRIANGLE_FAN Each new vertex defines another triangle

viii. GL_QUADS, GL_QUAD_STRIP, GL_POLYGON



D. Each Begin + End sequence is expensive. Do not do one per point/polygon. Wrap as many as you can into fewer Begin/end blocks.

II. GL Display Lists

A. Alleviating like CPU calls to GPU with Open GL Display Lists, similar to mini program. Stores calls on GPU, then calls are activated once on CPU site. Doesn't help inefficiency of immediates much.

B. Form: in preprocessing

```
GLuint list = glGenLists(1);
glNewList(list, GL_COMPILE);
: // Series of GL commands that all can be used in lists*
glEndList();
```

C. Form: in use

```
glCallList(list); // one CPU command to GPU compiled program.
```

D. Identifier 0 of GL objects, eg. a Display List, is invalid. can invoke glIs*(x) to check validity of objects, eg.. glIsList(list);

E. Cleanup. - you must deallocate GPU memory explicitly (similar to delete keyword)

```
glDeleteLists(list, 1);
```

F. Support exists for multiple lists at same time when generating, displaying, and deleting.

III. Vertex Array Objects

A. Reduces copying of data (or bulk copies data) to GPU. Asynchronous. Extremely efficient. Calls to GL tell GPU how to interpret raw arrays of memory/attributes.

B. Approximate code: *can be w/out vertex array* *can be inside of display list*

i. Vertex Array Object (VAO): All of the state needed to supply vertex data/attributes to GPU

ii. Vertex Buffer Object (VBO): An array mapping (buffer) to vertex data

iii. Index Buffer Object (IBO): An array mapping (buffer) to indices, e.g. faces/triangles

iv. Preprocessing:

```
GLuint vao;
glGenVertexArrays(1, &vao);   //create on GPU
glBindVertexArray(vao);       // make active

//vertex buffer object
GLuint vbo;
glGenBuffers(1, &vbo);  //create Buffer on GPU
glBindBuffers(GL_ARRAY_BUFFER, vbo);  //make active
                                                        usage
glBufferData(GL_ARRAY_BUFFER, numBytes, dataPtr, GL_STATIC_DRAW);
      //define mapping
//Index buffer object
GLuint ibo;
/*Same buffer process w/ GL_ELEMENT_ARRAY_BUFFER */
//Attributes can also be defined w/ glEnableVertexAttribArray(x) or glEnableClientState(GL_*)  e.g. GL_NORMAL_ARRAY
// and glVertexAttribPointer
```

v. To unbind: glBind*(GL_ARRAY_BUFFER, 0); for example

vi. To draw: Bind VAO w/ glBindVertexArray(vao);

```
glDrawElements(GL_*, number, type, startPoint of IBO)
               primitive
e.g.
glDrawElements(GL_TRIANGLES, 10, GL_UNSIGNED_INT, ((char*) NULL+0));
```

vii. To delete: glDelete*(*)

viii. A bit more complexity exists, lots of options. Bonus on first assignment.