CS 432 – Interactive Computer Graphics

Lecture 4 – Part 1

3D Graphics

Topics

- From 2D to 3D
- Geometry
- Change of Coordinate Systems

Reading

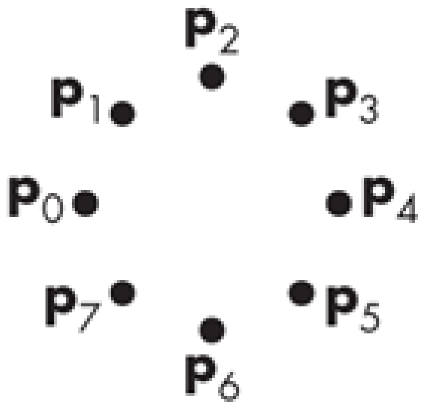
- Angel: Chapter 3
- Red Book: Chapter 5, Appendix E

From 2D to 3D

- 2D Graphics are a special case of 3D graphics
  - With the 3rd dimension $z = 0$
- Going to 3D: Not many changes
  - Use vec4 (for homogenous coordinates)
  - Need to worry more about the order in which primitives are rendered and/or use hidden-surface removal
    - Some will *occlude* the others

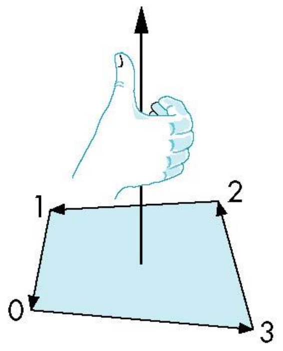
Inward and Outward Facing Polygons

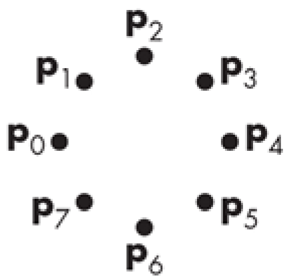
- Below we have 8 vertices.
- The order $\{p_1, p_6, p_7\}$ and $\{p_6, p_7, p_1\}$ are equivalent in that the same polygon will be rendered by OpenGL
- But the order $\{\mathrm{p}_1, \mathrm{p}_7, \mathrm{p}_6\}$ is different!



Inward and Outward Facing Polygons

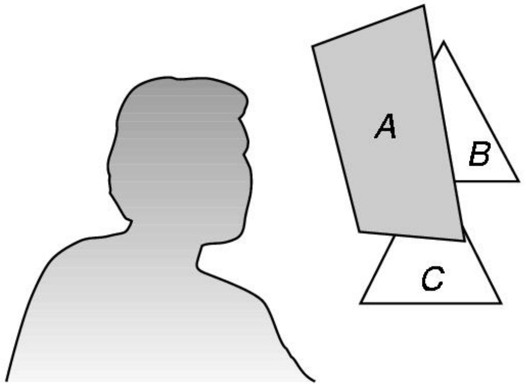
- The first two describe *inward facing* polygons
- Use the *right-hand rule:*
  - Do an encirclement of the vertices (in the order listed) with your right hand
  - Direction of your thumb is the direction the polygon is facing
    - The direction of its *normal*
- OpenGL can treat inward and outward facing polygons differently





Hidden Surface Removal

- We only want to see surface that are in front of other surfaces
- OpenGL uses a *hidden-surface* method called the *z-buffer* algorithm that saves depth information as objects are rendered so that only the front objects appear in the image



Using the z-buffer

- We must allow for usage of another buffer, the z-buffer, to store depth information as geometry travels down the pipeline.
- To do this we must:
  - Request the depth buffer in the main function
    - `glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);`
  - Enable its usage in the initialization
    - `glEnable(GL_DEPTH_TEST)`
  - Clear it in the display callback
    - `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)`

3D Cube Example

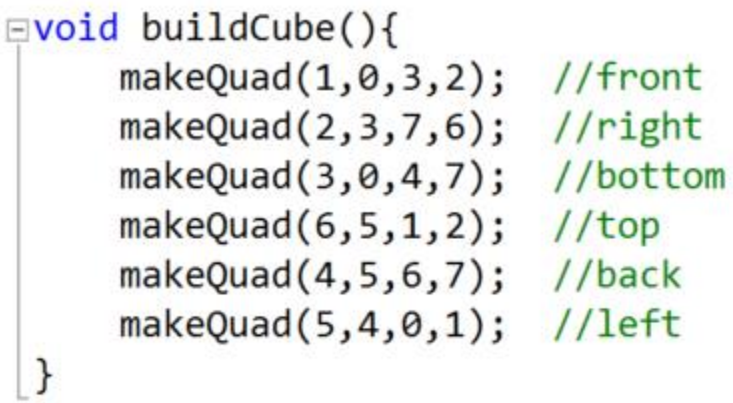
- A cube has 6 faces defined using 8 vertices
- If we decide to render the cube using GL_TRIANGLES we need 6*3*2=36 vertices
  - Each of which are from our set of 8 vertices
- vec4 points[36];
- We could "hard code" each of these 36 points, or more intelligently select them from our set of 8 *unique* vertices

```
vec4 vertices[] = {vec4(-0.5,-0.5,0.5,1),vec4(-0.5,0.5,0.5,1),vec4(0.5,0.5,0.5,1),vec4(0.5,-0.5,0.5,1),
        vec4(-0.5,-0.5,-0.5,1),vec4(-0.5,0.5,-0.5,1),vec4(0.5,0.5,-0.5,1),vec4(0.5,-0.5,-0.5,1)};
vec4 potentialColors[] = {vec4(0,0,0,1), vec4(1,0,0,1), vec4(0,1,0,1), vec4(0,0,1,1),
        vec4(1,1,0,1), vec4(1,0,1,1),vec4(0,1,1,1),vec4(0.5f,0.2f,0.4f,1)};
vec4 points[6*2*3]; //6 faces, 2 triangles/face, 3 vertices per triangle
vec4 colors[6*2*3];
```

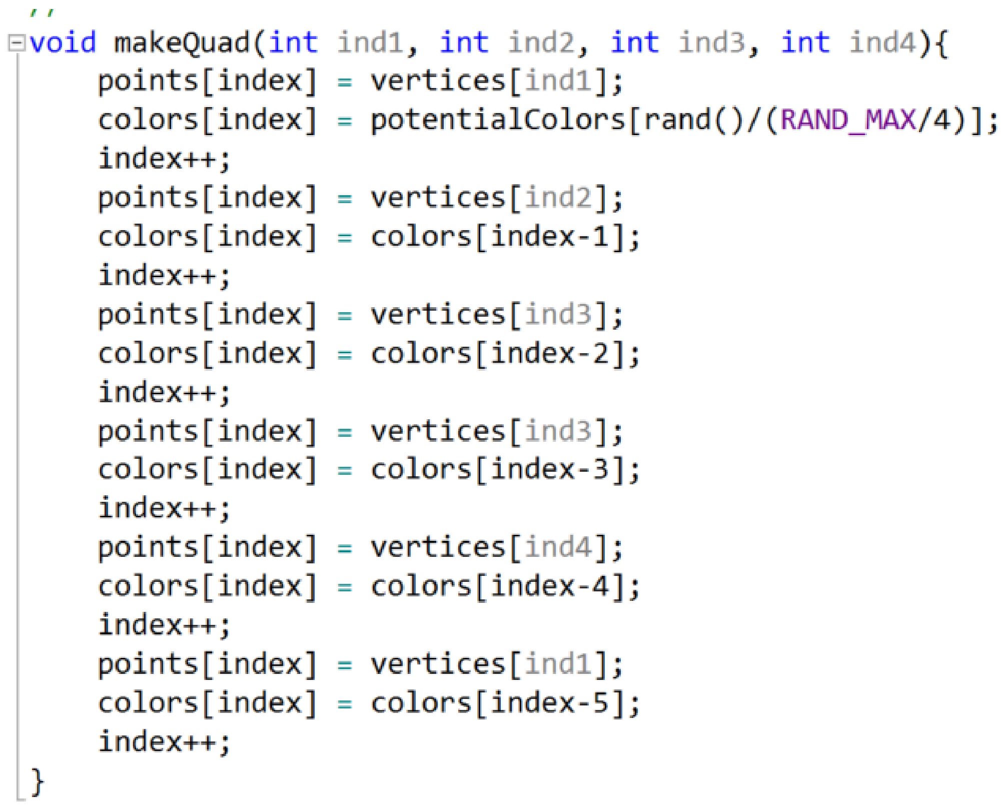
3D Cube Example

3D Cube Example

- Ok. A cube is made up of 6 faces (quads), each of which are made up of 2 triangles, each of which need 3 vertices.
- Let's try to build this!

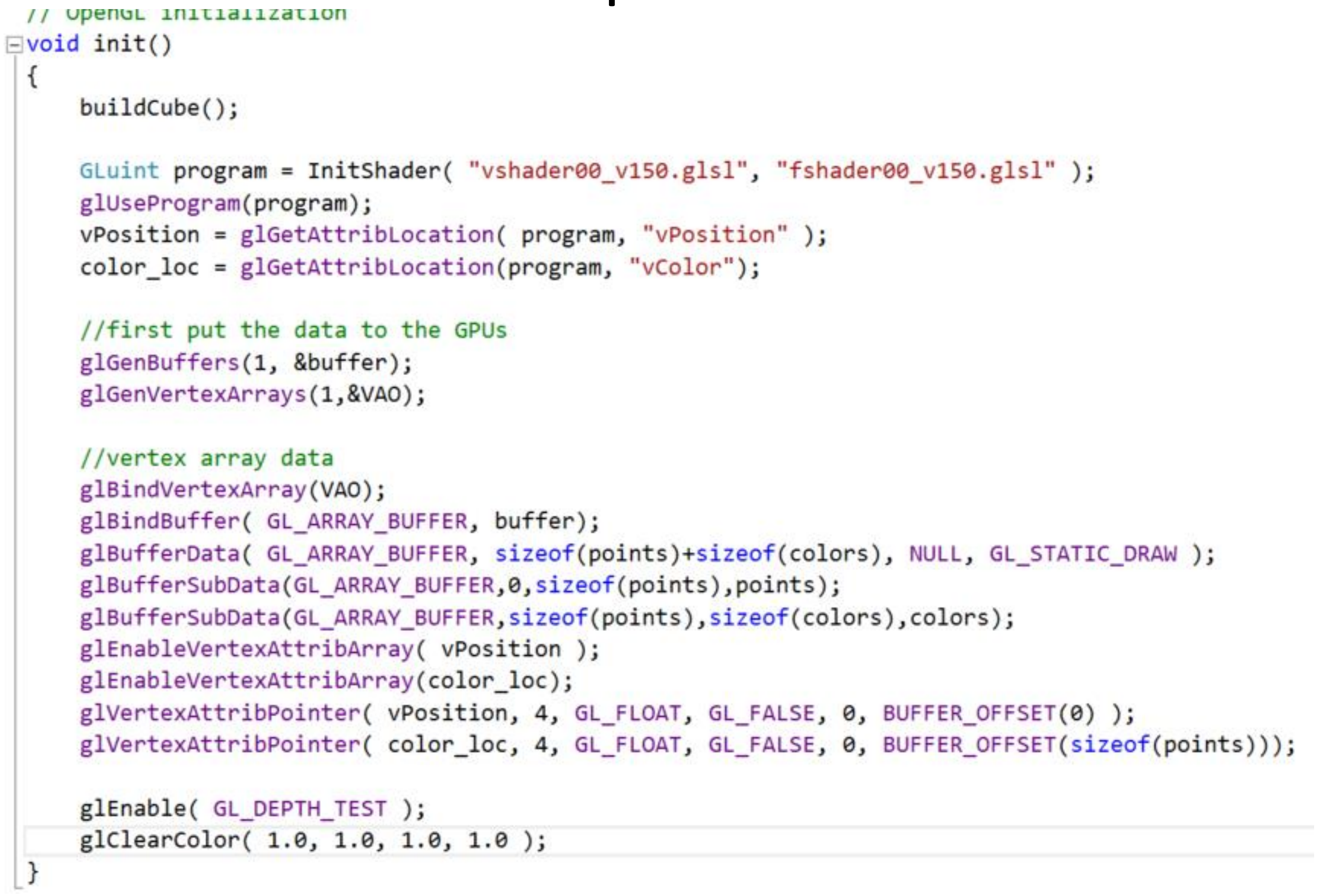


```
void buildCube(){
    makeQuad(1,0,3,2);  //front
    makeQuad(2,3,7,6);  //right
    makeQuad(3,0,4,7);  //bottom
    makeQuad(6,5,1,2);  //top
    makeQuad(4,5,6,7);  //back
    makeQuad(5,4,0,1);  //left
}
```



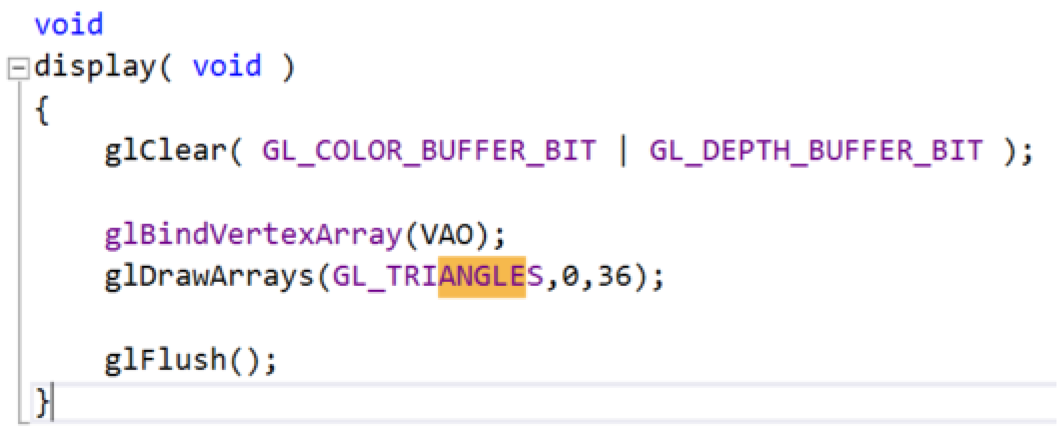
```
void makeQuad(int ind1, int ind2, int ind3, int ind4){
    points[index] = vertices[ind1];
    colors[index] = potentialColors[rand()/(RAND_MAX/4)];
    index++;
    points[index] = vertices[ind2];
    colors[index] = colors[index-1];
    index++;
    points[index] = vertices[ind3];
    colors[index] = colors[index-2];
    index++;
    points[index] = vertices[ind3];
    colors[index] = colors[index-3];
    index++;
    points[index] = vertices[ind4];
    colors[index] = colors[index-4];
    index++;
    points[index] = vertices[ind1];
    colors[index] = colors[index-5];
    index++;
}
```

3D Cube Example



```
// OpenGL initialization
void init()
{
    buildCube();

    GLuint program = InitShader( "vshader00_v150.glsl", "fshader00_v150.glsl" );
    glUseProgram(program);
    vPosition = glGetAttribLocation( program, "vPosition" );
    color_loc = glGetAttribLocation(program, "vColor");

    //first put the data to the GPUs
    glGenBuffers(1, &buffer);
    glGenVertexArrays(1,&VAO);

    //vertex array data
    glBindVertexArray(VAO);
    glBindBuffer( GL_ARRAY_BUFFER, buffer);
    glBufferData( GL_ARRAY_BUFFER, sizeof(points)+sizeof(colors), NULL, GL_STATIC_DRAW );
    glBufferSubData(GL_ARRAY_BUFFER,0,sizeof(points),points);
    glBufferSubData(GL_ARRAY_BUFFER,sizeof(points),sizeof(colors),colors);
    glEnableVertexAttribArray( vPosition );
    glEnableVertexAttribArray(color_loc);
    glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
    glVertexAttribPointer( color_loc, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(points)));

    glEnable( GL_DEPTH_TEST );
    glClearColor( 1.0, 1.0, 1.0, 1.0 );
}
```

3D Cube Example



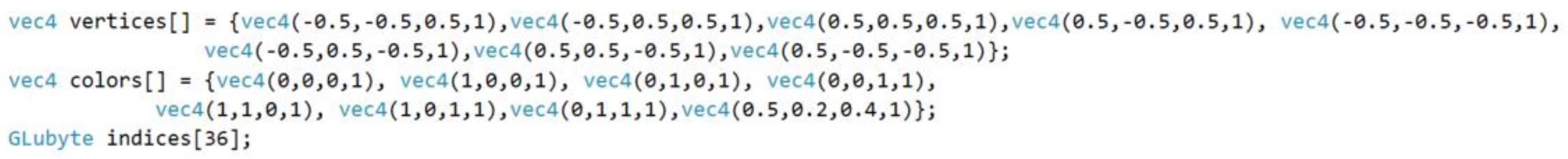
```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES,0,36);

    glFlush();
}
```

Using Index Arrays

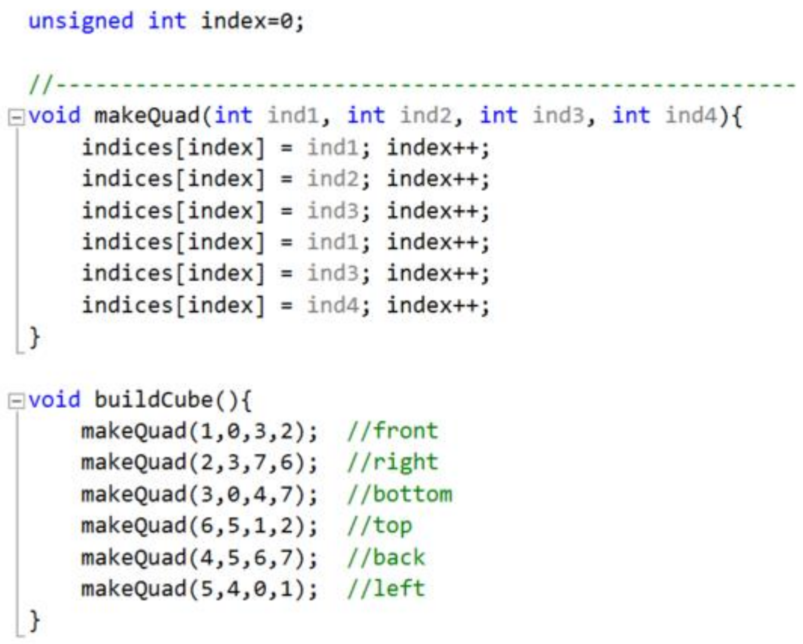
- The problem with the proposed method is that we’re wasting a lot of space
  - Push several copies of the same vertex to the GPU buffer
- What if we could just push the unique set of vertices and then *index* them

Indexing Vertex Arrays

- So now we need
  - Indices – if we have < 256 indices these can just be unsigned bytes
  - Unique vertices – these should still be vec4 objects.



```
vec4 vertices[] = {vec4(-0.5,-0.5,0.5,1),vec4(-0.5,0.5,0.5,1),vec4(0.5,0.5,0.5,1),vec4(0.5,-0.5,0.5,1), vec4(-0.5,-0.5,-0.5,1),
                vec4(-0.5,0.5,-0.5,1),vec4(0.5,0.5,-0.5,1),vec4(0.5,-0.5,-0.5,1)};
vec4 colors[] = {vec4(0,0,0,1), vec4(1,0,0,1), vec4(0,1,0,1), vec4(0,0,1,1),
            vec4(1,1,0,1), vec4(1,0,1,1),vec4(0,1,1,1),vec4(0.5,0.2,0.4,1)};
GLubyte indices[36];
```



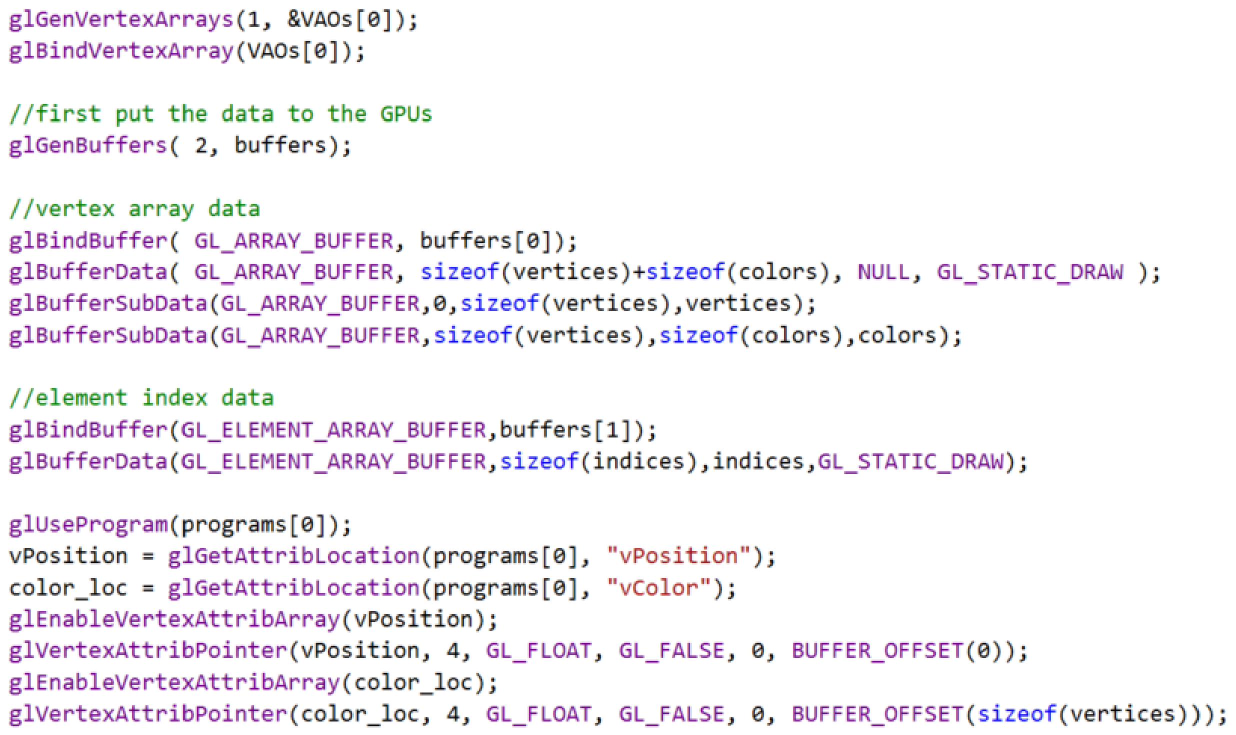
```
unsigned int index=0;

//----------------------------------------------------------
void makeQuad(int ind1, int ind2, int ind3, int ind4){
    indices[index] = ind1; index++;
    indices[index] = ind2; index++;
    indices[index] = ind3; index++;
    indices[index] = ind1; index++;
    indices[index] = ind3; index++;
    indices[index] = ind4; index++;
}

void buildCube(){
    makeQuad(1,0,3,2);  //front
    makeQuad(2,3,7,6);  //right
    makeQuad(3,0,4,7);  //bottom
    makeQuad(6,5,1,2);  //top
    makeQuad(4,5,6,7);  //back
    makeQuad(5,4,0,1);  //left
}
```

Indexing Vertex Arrays

- Now the vertex array object is supposed to store information about vertices (location, color, etc..)
- Where do we put the indices on the GPU?
- We'll they'll go in a buffer as well, but this will be an *element array buffer*
- And the last caveat is that we'll have to instruct the GPU to draw using the element array.

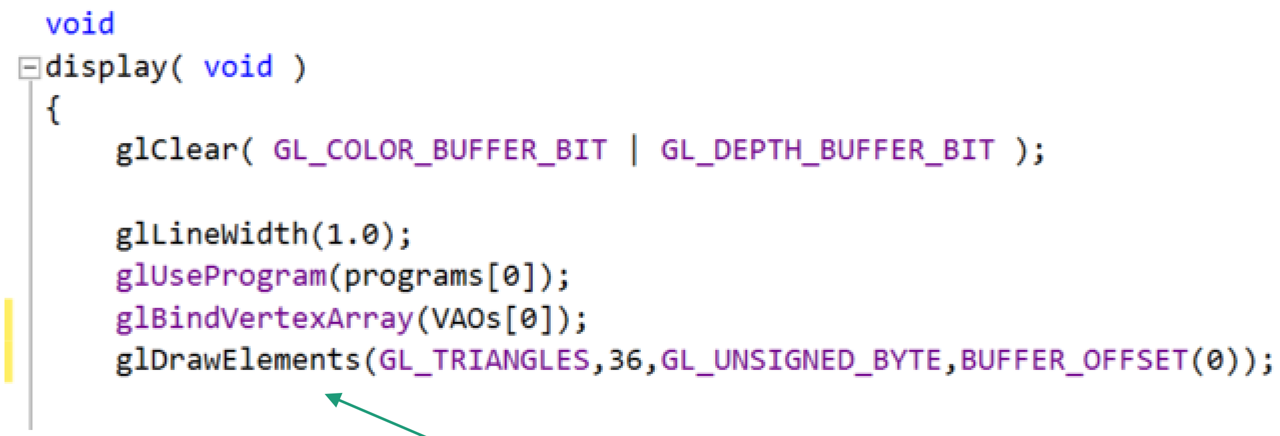
Initialization

- Copy the vertex attribute data into an **array buffer**
- Copy the indices into an **element array buffer**



```
glGenVertexArrays(1, &VAOs[0]);
glBindVertexArray(VAOs[0]);

//first put the data to the GPUs
glGenBuffers( 2, buffers);

//vertex array data
glBindBuffer( GL_ARRAY_BUFFER, buffers[0]);
glBufferData( GL_ARRAY_BUFFER, sizeof(vertices)+sizeof(colors), NULL, GL_STATIC_DRAW );
glBufferSubData(GL_ARRAY_BUFFER,0,sizeof(vertices),vertices);
glBufferSubData(GL_ARRAY_BUFFER,sizeof(vertices),sizeof(colors),colors);

//element index data
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,buffers[1]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(indices),indices,GL_STATIC_DRAW);

glUseProgram(programs[0]);
vPosition = glGetAttribLocation(programs[0], "vPosition");
color_loc = glGetAttribLocation(programs[0], "vColor");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
glEnableVertexAttribArray(color_loc);
glVertexAttribPointer(color_loc, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(vertices)));
```

Drawing



```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glLineWidth(1.0);
    glUseProgram(programs[0]);
    glBindVertexArray(VAOs[0]);
    glDrawElements(GL_TRIANGLES,36,GL_UNSIGNED_BYTE,BUFFER_OFFSET(0));
```

Note the use of `glDrawElements` instead of `glDrawArrays`

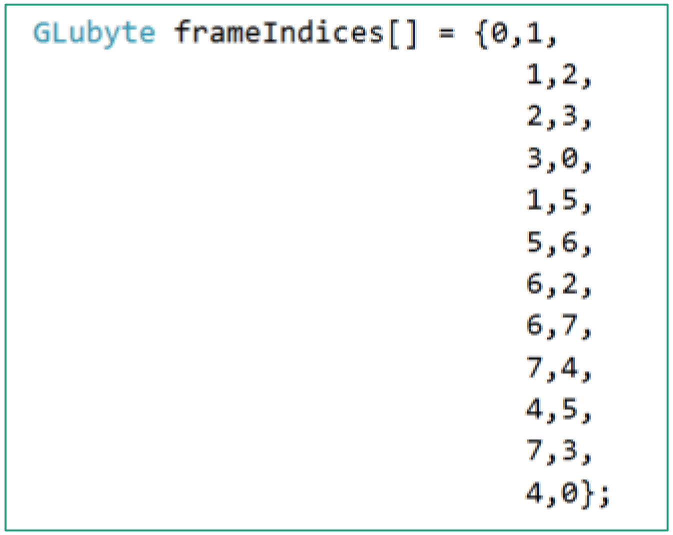
Shaders

- Fortunately our shader programs stays the same!

```
#version 150
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main(){
	gl_Position = vPosition;
	color = vColor;
}
```

Cube With Frame

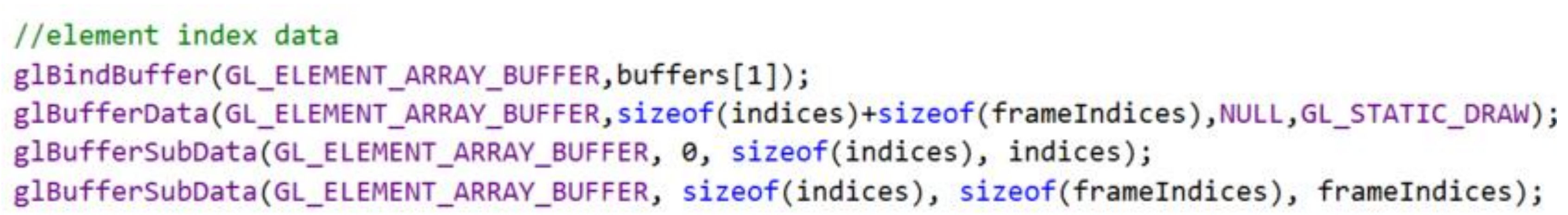
- To see how this is really useful let’s draw a cube with a frame!
- It will use the same vertices as the cube.
- But we’ll just draw lines whose vertices index the cube’s vertices



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

Cube With Frame

- We can store the indices write in the same buffer that the cube indices are in…



```
//element index data
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,buffers[1]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(indices)+sizeof(frameIndices),NULL,GL_STATIC_DRAW);
glBufferSubData(GL_ELEMENT_ARRAY_BUFFER, 0, sizeof(indices), indices);
glBufferSubData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), sizeof(frameIndices), frameIndices);
```

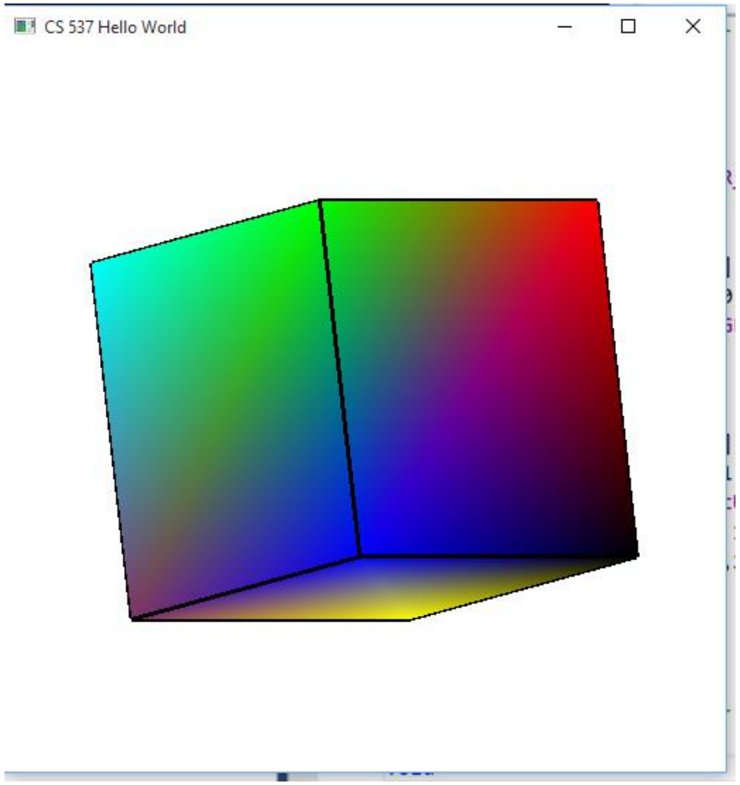
- However, since we’re going to draw the two differently (the cube has vertex colors, the frame will just be black) we’ll need different shaders.
  - And therefore the attribute locations will be different.
- Therefore we need two different VAOs so the attributes locations are correct

Cube with Frame

```
//Set up the cube VAO
glBindVertexArray(VAOs[0]);
glBindBuffer(GL_ARRAY_BUFFER,buffers[0]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,buffers[1]);
glUseProgram(programs[0]);
vPosition = glGetAttribLocation( programs[0], "vPosition" );
color_loc = glGetAttribLocation(programs[0], "vColor");
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
glEnableVertexAttribArray(color_loc);
glVertexAttribPointer( color_loc, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(vertices)));

//Set up the frame VAO
glBindVertexArray(VAOs[1]);
glBindBuffer(GL_ARRAY_BUFFER,buffers[0]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,buffers[1]);
glUseProgram(programs[1]);
vPosition = glGetAttribLocation( programs[1], "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
```

Cube with Frame

```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glLineWidth(1.0);
    glUseProgram(programs[0]);
    glBindVertexArray(VAOs[0]);
    glDrawElements(GL_TRIANGLES,36,GL_UNSIGNED_BYTE,BUFFER_OFFSET(0));


    glLineWidth(3.0);
    glUseProgram(programs[1]);
    glBindVertexArray(VAOs[1]);
    GLuint color_loc = glGetUniformLocation(programs[1], "color");
    glUniform4fv(color_loc, 1, vec4(0.0, 0.0, 0.0, 1.0));
    glDrawElements(GL_LINES,24,GL_UNSIGNED_BYTE,BUFFER_OFFSET(sizeof(indices)));

    glFlush();
}
```



Shaders

Cube Vertex Shader

```
#version 150
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main(){
		gl_Position = vPosition;
		color = vColor;
}
```

Frame Vertex Shader

```
#version 150
in vec4 vPosition;
uniform vec4 vColor;
out vec4 color;

void main(){
		gl_Position = vPosition;
		color = vColor;
}
```

Transformations

- Of course this cube is all rotated and stuff!!!!
- Let’s check out how to do 3D transformations