centre adscrit a:

Computer Graphics

10. OpenGL Objects (VAO, VBO, etc)

Dr Joan Llobera – joanllobera@enti.cat

Outline

1. VAO, VBO, Shaders and Programs
2. Analysis of the difference between our two cubes

1. VAO? VBO?

What is a VAO?

A Vertex Array Object is:

- An object which contains one or more Vertex Buffer Objects
- An object designed to store information for a complete object

What is a VBO?

A Vertex Buffer Object is:

- A memory buffer inside the GPU
- It is designed to hold information about vertices

Analysis of the cube program

Data structures:

- An array which contains the vertices
- An array which contains the normal
- An array which contains the vertices

```
glm::vec3 cubeVerts[] = {
verts[1], verts[0], verts[2], verts[3],
verts[5], verts[6], verts[4], verts[7],
verts[1], verts[5], verts[0], verts[4],
verts[2], verts[3], verts[6], verts[7],
verts[0], verts[4], verts[3], verts[7],
verts[1], verts[2], verts[5], verts[6]
};
glm::vec3 cubeNorms[] = {
norms[0], norms[0], norms[0], norms[0],
norms[1], norms[1], norms[1], norms[1],
norms[2], norms[2], norms[2], norms[2],
norms[3], norms[3], norms[3], norms[3],
norms[4], norms[4], norms[4], norms[4],
norms[5], norms[5], norms[5], norms[5]
};

GLubyte cubeIdx[] = {
0, 1, 2, 3, UCHAR_MAX,
4, 5, 6, 7, UCHAR_MAX,
8, 9, 10, 11, UCHAR_MAX,
12, 13, 14, 15, UCHAR_MAX,
16, 17, 18, 19, UCHAR_MAX,
20, 21, 22, 23, UCHAR_MAX
};
```

(Reminder: What's in an OBJ)

v	vertex	
Vt	texture coordinates	
Vn	vertex normal	
f	a face	(indexes start at 1, not 0)

```
# Blender3D v249 OBJ File: untitled.blend
# www.blender3d.org
mtllib cube.mtl
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vt 0.748573 0.750412
vt 0.749279 0.501284
vt 0.999110 0.501077
vt 0.999455 0.750380
vt 0.250471 0.500702
vt 0.249682 0.749677
vt 0.001085 0.750380
vt 0.001517 0.499994
vt 0.499422 0.500239
vt 0.500149 0.750166
vt 0.748355 0.998230
vt 0.500193 0.998728
vt 0.498993 0.250415
vt 0.748953 0.250920
vn 0.000000 0.000000 -1.000000
vn -1.000000 -0.000000 -0.000000
vn -0.000000 -0.000000 1.000000
vn -0.000001 0.000000 1.000000
vn 1.000000 -0.000000 0.000000
vn 1.000000 0.000000 0.000001
vn 0.000000 1.000000 -0.000000
vn -0.000000 -1.000000 0.000000
usemtl Material_ray.png
s off
f 5/1/1 1/2/1 4/3/1
f 5/1/1 4/3/1 8/4/1
f 3/5/2 7/6/2 8/7/2
f 3/5/2 8/7/2 4/8/2
f 2/9/3 6/10/3 3/5/3
f 6/10/4 7/6/4 3/5/4
f 1/2/5 5/1/5 2/9/5
f 5/1/6 6/10/6 2/9/6
f 5/1/7 8/11/7 6/10/7
f 8/11/7 7/12/7 6/10/7
f 1/2/8 2/9/8 3/13/8
f 1/2/8 3/13/8 4/14/8
```

Notice the similarity with
the outcome of parsing an .obj file!

ESCOLA DE NOVES TECNOLOGIES INTERACTIVES

UNIVERSITAT DE BARCELONA

The VAO

From the perspective of the CPU, the VAO is merely an unsigned integer

```
GLuint cubeVao;
GLuint cubeVbo[3];
GLuint cubeShaders[2];
GLuint cubeProgram;
glm::mat4 objMat = glm::mat4(1.f);
```

From the perspective of the GPU, the VAO is defined with specific instructions: we generate it, and then we bind it.

```
glGenVertexArrays(1, &cubeVao);
glBindVertexArray(cubeVao);
```

The VBOs

From the CPU’s perspective, a VBO is merely an unsigned integer:

```
GLuint cubeVbo[3];
```

However, from the GPU’s perspective…

```
glGenBuffers(3, cubeVbo);

glBindBuffer(GL_ARRAY_BUFFER, cubeVbo[0]);
glBufferData(GL_ARRAY_BUFFER,
sizeof(cubeVerts), cubeVerts, GL_STATIC_DRAW);
glVertexAttribPointer((GLuint)0, 3, GL_FLOAT,
GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, cubeVbo[1]);
glBufferData(GL_ARRAY_BUFFER,
sizeof(cubeNorms), cubeNorms, GL_STATIC_DRAW);
glVertexAttribPointer((GLuint)1, 3, GL_FLOAT,
GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);

glPrimitiveRestartIndex(UCHAR_MAX);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
cubeVbo[2]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
sizeof(cubeIdx), cubeIdx, GL_STATIC_DRAW);
```

The VBOs

```
glGenBuffers(3, cubeVbo);

glBindBuffer(GL_ARRAY_BUFFER, cubeVbo[0]);
glBufferData(GL_ARRAY_BUFFER,
sizeof(cubeVerts), cubeVerts,
GL_STATIC_DRAW);
glVertexAttribPointer((GLuint)0, 3, GL_FLOAT,
GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
```

```
glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

define an array of generic vertex attribute data

index
Specifies the index of the generic vertex attribute to be modified.

size
Specifies the number of components per generic vertex attribute. Must be 1, 2, 3, 4. The initial value is 4.

type
Specifies the data type of each component in the array. The symbolic constants `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, and `GL_UNSIGNED_INT` are accepted The initial value is `GL_FLOAT`.

normalized
specifies whether fixed-point data values should be normalized (`GL_TRUE`) or converted directly as fixed-point values (`GL_FALSE`) when they are accessed.

stride
Specifies the byte offset between consecutive generic vertex attributes. If *stride* is 0, the generic vertex attributes are understood to be tightly packed in the array. The initial value is 0.

pointer
Specifies a offset of the first component of the first generic vertex attribute in the array in the data store of the buffer currently bound to the `GL_ARRAY_BUFFER` target. The initial value is 0.

```
glPrimitiveRestartIndex(UCHAR_MAX);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
cubeVbo[2]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
sizeof(cubeIdx), cubeIdx, GL_STATIC_DRAW);
```

The VBOs

```
glPrimitiveRestartIndex(UCHAR_MAX);
```

Specifies a vertex array element that is treated specially when primitive restarting is enabled. This is known as the primitive restart index.

When one of the **Draw** commands transfers a set of generic attribute array elements to the GL, if the index within the vertex arrays corresponding to that set is equal to the primitive restart index, then the GL does not process those elements as a vertex. Instead, it is as if the drawing command ended with the immediately preceding transfer, and another drawing command is immediately started with the same parameters, but only transferring the immediately following element through the end of the originally specified elements.

The Shaders and the Program

From the CPU's perspective, a Shader (or a Program) is only an unsigned integer:

```
GLuint cubeProgram;
```

From the GPU perspective, A shader:

- Contains the compiled code

From the GPU, the program:

- Contains the compiled shaders
- Has attribute location points
- It Links the 3 elements

```
cubeShaders[0] =
compileShader(cube_vertShader,
GL_VERTEX_SHADER, "cubeVert");
cubeShaders[1] =
compileShader(cube_fragShader,
GL_FRAGMENT_SHADER, "cubeFrag");

cubeProgram = glCreateProgram();
glAttachShader(cubeProgram,
cubeShaders[0]);
glAttachShader(cubeProgram,
cubeShaders[1]);
glBindAttribLocation(cubeProgram, 0,
"in_Position");
glBindAttribLocation(cubeProgram, 1,
"in_Normal");
linkProgram(cubeProgram);
```

Review of Vertex Shader

```
const char* cube_vertShader =
"#version 330\n\
in vec3 in_Position;\n\
in vec3 in_Normal;\n\
out vec4 vert_Normal;\n\
uniform mat4 objMat;\n\
uniform mat4 mv_Mat;\n\
uniform mat4 mvpMat;\n\
void main() {\n\
gl_Position = mvpMat * objMat * vec4(in_Position, 1.0);\n\
vert_Normal = mv_Mat * objMat * vec4(in_Normal, 0.0);\n\
}";
```

Review of Fragment Shader

```
const char* cube_fragShader =
"#version 330\n\
in vec4 vert_Normal;\n\
out vec4 out_Color;\n\
uniform mat4 mv_Mat;\n\
uniform vec4 color;\n\
void main() {\n\
out_Color = vec4(

color.xyz * dot(vert_Normal, mv_Mat*vec4(0.0, 1.0, 0.0, 0.0))
+ color.xyz * 0.3, 1.0 );}";
```

Review of Fragment Shader

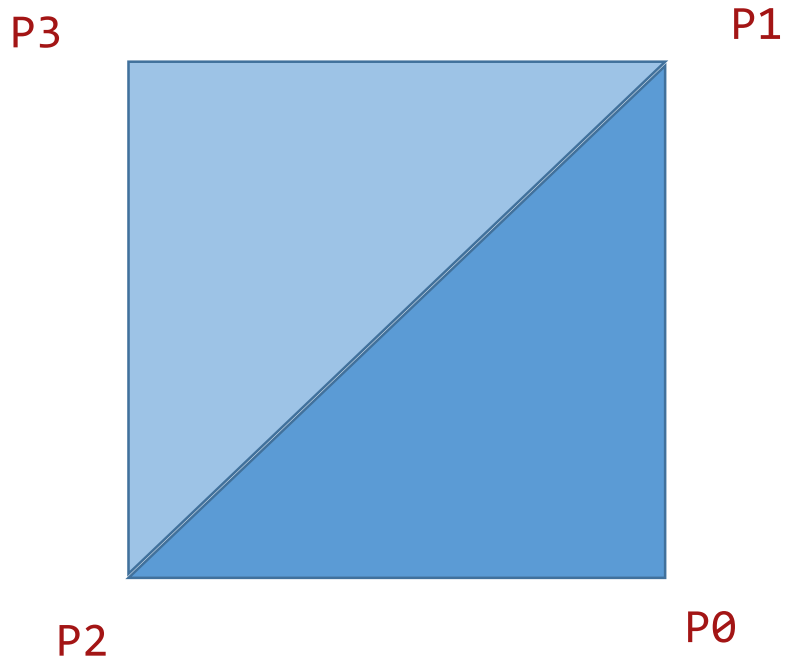
```
const char* cube_fragShader =
"#version 330\n\
in vec4 vert_Normal;\n\
out vec4 out_Color;\n\
uniform mat4 mv_Mat;\n\
uniform vec4 color;\n\
void main() {\n\
out_Color = vec4(

color.xyz * dot(vert_Normal, mv_Mat*vec4(0.0, 1.0, 0.0, 0.0))
+ color.xyz * 0.3, 1.0 );}";
```

Exercise 1: Load your own 3D model and apply the cube shader

Outcome of the model loading in the LoadObj function should be:

```
// For each vertex of each triangle
for (unsigned int i = 0; i < vertexIndices.size(); i++) {
unsigned int vertexIndex = vertexIndices[i];
glm::vec3 vertex = temp_vertices[vertexIndex - 1];
out_vertices.push_back(vertex);
unsigned int uvindex = uvIndices[i];
glm::vec2 uv = temp_uvs[uvindex - 1];
out_uvs.push_back(uv);
unsigned int normalIndex = normalIndices[i];
glm::vec3 normal = temp_normals[normalIndex-1];
out_normals.push_back(normal);
}
```

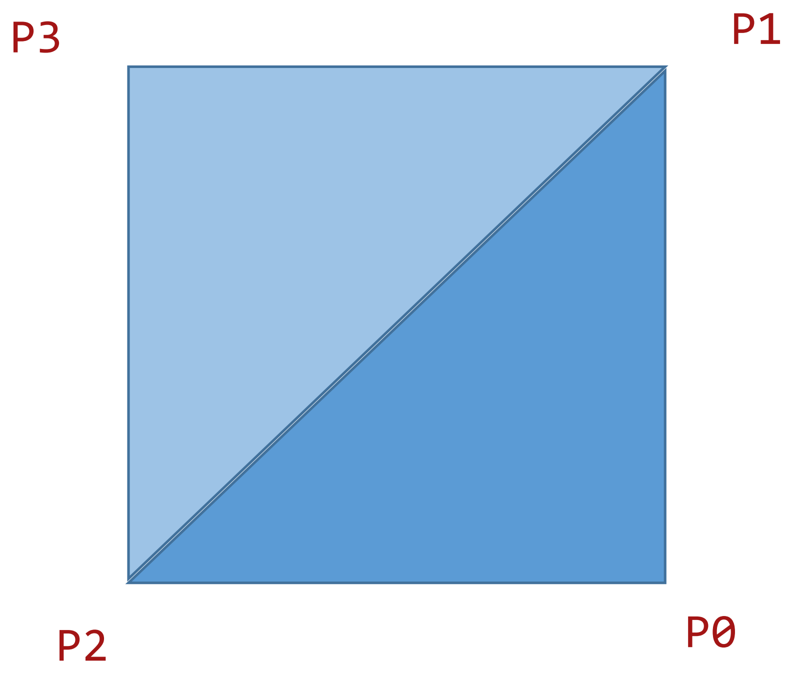
Exercise 2. Paint a cube face (1/2)



Notice how the cube given in the .obj has 3*2*6= 36 vertices. However, the primitive initially given has less. Why?

What is the difference in both rendering shaders?

Exercise 2. Paint a cube face (2/2)



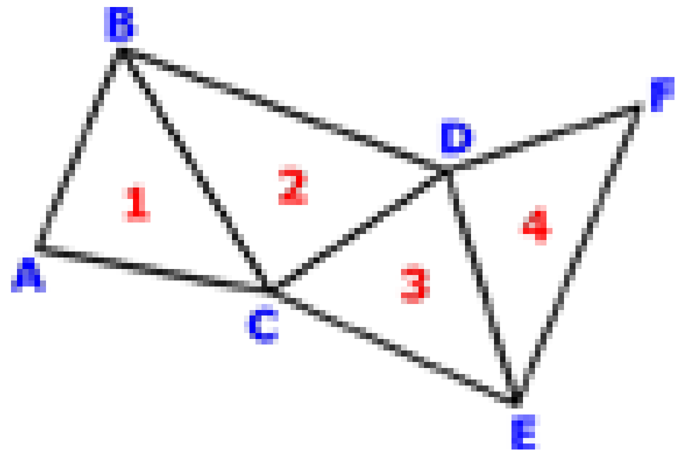
Notice the order in the primitive: the first triangle needs to be drawn anti-clockwise (right hand rule).

The second is defined from the last 2 points + a new one.

Exercise2. Details: Normals in a triangle strip

GL_TRIANGLE_STRIP

Draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v2, v1, v3 (note the order), then v2, v3, v4, and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface.



Resources

- [Kessenich] Kessenich et al. OpenGL Programming Guide. Chapter 7. Light and Shadow
- https://learnopengl.com/Lighting/Basic-Lighting
- http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/
- https://en.wikipedia.org/wiki/Phong_reflection_model