Chapter 6

OpenGL ES and Shader

The vertex array input to the rendering pipeline contains the per-vertex data such as position and normal. The vertex shader runs once for each vertex. GPU has a massively parallel architecture with a lot of fine-grained independently working cores. The architecture is well suited for processing a large number of vertices simultaneously.

6.1 OpenGL ES

OpenGL ES is a subset of OpenGL. OpenGL ES 2.0 is derived from OpenGL 2.0. It requires two programs to be provided: vertex and fragment shaders. OpenGL ES 3.0 is derived from OpenGL 3.3 and adds many enhanced features to OpenGL ES 2.0. It is backward compatible with OpenGL ES 2.0, i.e., applications built upon OpenGL ES 2.0 work with OpenGL ES 3.0. This book focuses on OpenGL ES 3.0.

OpenGL ES 3.1 (publicly released in March 2014) supports *compute shader* for general-purpose GPU computing and OpenGL ES 3.2 (August 2015) includes *geometry shader* and *tessellation shader*. Chapter 19 will present the tessellation shader.

An OpenGL ES specification has two components: the API specification and shading language specification. From now on, let us call OpenGL ES simply 'GL.' (We do not use abbreviation for OpenGL.) On the other hand, 'SL' is short for OpenGL ES Shading Language.

This book presents the minimal syntax of GL and SL, which would be enough to understand the sample codes given in this book. Readers are referred to a manual such as [7] for serious programming in GL and SL.

6.2 Vertex Shader

SL is a C-like language, and therefore programmers with experiences in C do not have difficulties in understanding its data types, operations, and control flow. However, SL works on GPU, the goal and architecture of which are different from those of CPU. The differences shape SL in a distinct way. Vectors and matrices are good examples. In addition to the basic types such as `float`, SL supports vectors up to four elements. For example, `vec4` is for a floating-point 4D vector and `ivec3` is for an integer 3D vector. SL also supports matrices up to 4×4 elements. For example, `mat3` and `mat4` define 3×3 and 4×4 'square' matrices, respectively, whereas `mat3x4` is for a 3×4 matrix. The matrix elements are all `float` values.

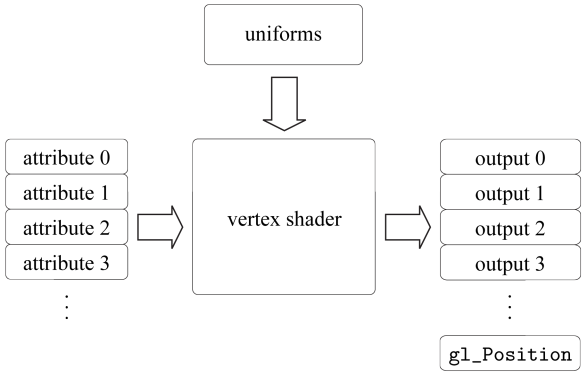

Fig. 6.1: Input and output of vertex shader.

Two major inputs to the vertex shader are named *attributes* and *uniforms*, as shown in Fig. 6.1. The per-vertex data stored in the vertex array, such as position and normal, make up the attributes. Each vertex of a polygon mesh is processed by a distinct copy of the vertex shader, and the vertex attributes are unique to each execution of the vertex shader. In contrast, the uniforms remain constant for multiple executions of a shader. A good example of the uniform input is the world transform. The same world matrix is applied to all vertices of an object.

Sample code 6-1 shows a vertex shader. The first line declares that it

Sample code 6-1 vertex shader

```
 1: #version 300 es
 2:
 3: uniform mat4 worldMat, viewMat, projMat;
 4:
 5: layout(location = 0) in vec3 position;
 6: layout(location = 1) in vec3 normal;
 7: layout(location = 2) in vec2 texCoord;
 8:
 9: out vec3 v_normal;
10: out vec2 v_texCoord;
11:
12: void main() {
13:    gl_Position = projMat * viewMat * worldMat * vec4(position, 1.0);
14:    v_normal = transpose(inverse(mat3(worldMat))) * normal;
15:    v_texCoord = texCoord;
16: }
```

is written in SL 3.0. Three 4×4 matrices for world, view, and projection transforms are preceded by the keyword `uniform`. The vertex shader takes three attributes, `position`, `normal`, and `texCoord` preceded by the keyword `in`. (Here, `texCoord` is used to access 2D texture. Chapter 8 presents how it works.) A GL 3.0 implementation must support at least 16 attributes. When we have m attributes, their locations are indexed by $0, 1, 2, \ldots, m-1$. An attribute variable is bound to a location using `layout` qualifier. In Sample code 6-1, `position` is bound to location 0, `normal` is 1, and `texCoord` is 2. Section 6.4.1 presents more on attributes.

Two output variables, `v_normal` and `v_texCoord`, are defined using the keyword `out`. The main function first transforms the object-space vertex into the clip space. Computing the clip-space vertex position is the required task for every vertex shader, and the result is stored in the built-in variable `gl_Position`. Observe that all transforms are in 4×4 matrices but `position` is a 3D vector. In order to multiply them, `position` is converted into a 4D vector by invoking `vec4` as *constructor*.

The second statement of the main function extracts the upper-left 3×3 sub-matrix from the world matrix, i.e., $\boldsymbol{L}$ from $[\boldsymbol{L}|\boldsymbol{t}]$, as discussed in Section 5.1, applies its *inverse transpose* ($\boldsymbol{L}^{-T}$) to `normal`, and assigns the result to the output variable, `v_normal`. On the other hand, the vertex shader simply copies the input attribute, `texCoord`, to the output variable, `v_texCoord`.

6.3 OpenGL ES for Shaders

The previous section presents a vertex shader. The fragment shader is more complicated in general but is written in a similar fashion. (Fragment shaders will be presented starting from Chapter 8.) While the vertex and fragment shaders are in charge of low-level details in rendering, the GL program manages the shaders and provides them with various data needed for rendering. The GL commands begin with the prefix `gl`, and GL data types begin with `GL`.

Sample code 6-2 GL program for shader object

```
1: GLuint shader = glCreateShader(GL_VERTEX_SHADER);
2: glShaderSource(shader, 1, &source, NULL);
3: glCompileShader(shader);
```

Sample code 6-2 shows a fraction of GL program, which handles the vertex shader.

• A *shader object* is a container for storing the shader source code and is created using `glCreateShader`. It takes a single argument, which is either `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER` and then returns the ID of the new shader object.

• Suppose that the shader source code file is loaded into the GL program and is pointed by `source` that is of type `char*`. The shader source code is then stored into the shader object by `glShaderSource`. (This book focuses on key arguments of GL commands, thus passing over the second and fourth arguments of `glShaderSource`, for which readers are referred to GL programming manuals.)

• The shader object is compiled using `glCompileShader`.

With `GL_FRAGMENT_SHADER` for `glCreateShader`, the same procedure has to be done for the fragment shader.

Sample code 6-3 GL program for program object

```
1: GLuint program = glCreateProgram();
2: glAttachShader(program, shader);
3: glLinkProgram(program);
4: glUseProgram(program);
```

The vertex and fragment shader objects should be 'attached' to a *program object*, which is then 'linked' into the final executable. See Sample code 6-3.

- The program object is created by `glCreateProgram`.
- The shader object is attached to the program object by `glAttachShader`. Here, `shader` denotes the vertex shader object created in Sample code 6-2. (The fragment shader object should also be attached to the program object.)
- The program object is linked by `glLinkProgram`.
- So as to use the program object for rendering, `glUseProgram` is invoked.

6.4 Attributes and Uniforms

An important mission of the GL program is to make the attributes and uniforms ready to be consumed by the vertex shader. The GL program not only hands over them to the vertex shader but also informs the vertex shader of their structures.

6.4.1 Attributes and Buffer Objects

Sample code 6-4 GL program for vertex and index arrays

```
1: struct Vertex
2: {
3:    glm::vec3 pos; //position
4:    glm::vec3 nor; //normal
5:    glm::vec2 tex; //texture coordinates
6: };
7:
8: std::vector<Vertex> vertices;
9: std::vector<GLushort> indices;
```

For rendering a polygon mesh, the mesh data stored in a file (such as .obj file) are loaded into the vertex and index arrays of the GL program. Suppose that the arrays are pointed by `vertices` and `indices`, respectively, as shown in Sample code 6-4, where `glm` stands for OpenGL Mathematics and is a library that provides classes and functions with the same naming conventions and functionalities as SL.

The arrays in the CPU memory will then be transferred into what are called *buffer objects* in the GPU memory. For the indexed representation of a mesh, GL supports two types of buffer objects: (1) *Array buffer object* is for the vertex array and is specified by `GL_ARRAY_BUFFER`. (2) *Element array buffer object* is for the index array and is specified by `GL_ELEMENT_ARRAY_BUFFER`. Fig. 6.2 shows their relation.

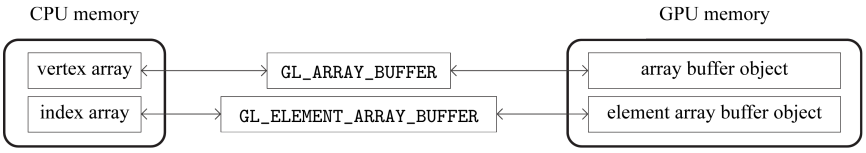

Fig. 6.2: Vertex and index arrays in the CPU memory vs. buffer objects in the GPU memory.

Sample code 6-5 GL program for array buffer object

```
1: GLuint abo;
2: glGenBuffers(1, &abo);
3: glBindBuffer(GL_ARRAY_BUFFER, abo);
4: glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex),
   vertices.data(), GL_STATIC_DRAW);
```

Sample code 19-1 shows how to create and bind a buffer object for the vertex array:

• Invoke `glGenBuffers(Glsizei n, Gluint *buffers)`, which returns `n` buffer objects in `buffers`.

• Invoke `glBindBuffer` to bind the buffer object to vertex array. The first argument of `glBindBuffer` can be `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, or many others[1].

• The buffer object is filled with data using `glBufferData`. The second argument specifies the size of the vertex array, and the third argument is the pointer to the array.

Sample code 6-6 GL program for element array buffer object

```
1: GLuint eabo;
2: glGenBuffers(1, &eabo);
3: glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, eabo);
4: glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLushort),
   indices.data(), GL_STATIC_DRAW);
```

[1]GL supports many types of buffer objects such as transform feedback buffers to be presented in Chapter 19. When such buffer objects are used, the first argument of `glBindBuffer` takes on other symbolic constants such as `GL_TRANSFORM_FEEDBACK_BUFFER`.

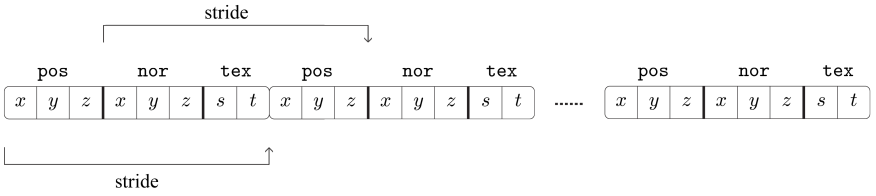

Fig. 6.3: Vertex attributes in the buffer object. The size of `Vertex` in Sample code 6-4 is 32 bytes and so is the stride.

Sample code 6-6 shows the same process for the index array. Then, both vertex and index arrays are copied to the GPU memory. Fig. 6.3 shows how the vertex attributes are put together in the buffer object. The first occurrence of `pos` is located at the very beginning of the buffer object and the next is 32-bytes away from the first. The byte distance between the consecutive attributes of the same kind is called 'stride.' By repeating the same stride, all position attributes can be retrieved. The vertex shader should be informed of this structure.

Sample code 6-7 GL program for specifying vertex attributes

```
 1: int stride = sizeof(Vertex);
 2: int offset = 0;
 3:
 4: glEnableVertexAttribArray(0); // position = attribute 0
 5: glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, stride, (GLvoid*)offset);
 6:
 7: offset += sizeof(glm::vec3); // for accessing normal
 8: glEnableVertexAttribArray(1); // normal = attribute 1
 9: glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, stride, (GLvoid*)offset);
10:
11: offset += sizeof(glm::vec3); // for accessing texture coordinates
12: glEnableVertexAttribArray(2); // texture coordinates = attribute 2
13: glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, stride, (GLvoid*)offset);
```

Recall that, in the vertex shader presented in Sample code 6-1, the vertex position is bound to location 0. Sample code 6-7 first enables attribute 0 using `glEnableVertexAttribArray` (4th line) and then details it using `glVertexAttribPointer` (5th line). The first argument of `glVertexAttribPointer` denotes attribute 0, the second and third arguments specify that it is composed of three float elements, the fifth argument is the stride, and the last

argument is the offset (in bytes) into the first occurrence of attribute 0 in the buffer object.

In the vertex shader presented in Sample code 6-1, the vertex normal and texture coordinates are bound to locations 1 and 2, respectively. They are enabled and detailed in the same manner as the vertex position. See lines 7 through 13 in Sample code 6-7. We use the same stride but the offset should be updated for each attribute. See the shaded box presented below for `glVertexAttribPointer`.

`void`	`glVertexAttribPointer(GLuint index,`
	`GLint size,`
	`GLenum type,`
	`GLboolean normalized,`
	`GLsizei stride,`
	`const GLvoid * pointer);`

`index`	the index of the vertex attribute
`size`	the number of components of the attribute, which must be 1, 2, 3, or 4
`type`	the data type such as `GL_INT` and `GL_FLOAT`
`normalized`	If `GL_TRUE`, integer data are mapped to [-1,1] or [0,1] before being used in the vertex shader.
`stride`	the byte distance between the consecutive attributes of the same kind
`pointer`	offset (in bytes) into the first occurrence of the attribute in the buffer object

6.4.2 Uniforms

The vertex shader in Sample code 6-1 takes three uniforms, `worldMat`, `viewMat`, and `projMat`. Consider a dynamic environment where a scene object keeps moving. Then, its `worldMat` should be updated for every frame. In addition, if the viewpoint moves, `viewMat` should be also updated. The GL program updates and provides them for the shader.

For this purpose, we first have to find the uniform's *location* in the program object, which has been determined during the link phase. Given the program object and a uniform's name in a string, e.g., `"worldMat"`, `glGetUniformLocation` returns an integer ID that represents the uniform's location.

`GLint`	`glGetUniformLocation(GLuint program,` `const GLchar *name);`
`program`	program object
`name`	uniform name in a string

Using the ID, we can load the uniform with specific data. A list of functions

which we collectively name `glUniform*` is available for loading uniforms. In order to load the uniform, `worldMat`, for example, we use `glUniformMatrix4fv`, where `4` indicates a 4×4 matrix.

`void`	`glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);`
`location`	uniform location
`count`	the number of matrices to be modified
`transpose`	This must be `GL_FALSE`.
`value`	pointer to an array of 16 `GLfloat` values

Sample code 6-8 shows how the world matrix is provided for vertex shader: The GL program updates the 4×4 matrix (`worldMatrix`), obtains the location of the shader uniform (`worldMat`), and provides `worldMatrix` for `worldMat`.

Sample code 6-8 GL program for specifying a uniform, world matrix

```
1: glm::mat4 worldMatrix; // repeatedly updated for a dynamic object
2:
3: GLint loc = glGetUniformLocation(program, "worldMat");
4: glUniformMatrix4fv(loc, 1, GL_FALSE, &worldMatrix);
```

6.5 Drawcalls

Using the GL commands presented so far, we have provided all attributes and uniforms for the vertex shader. Then, we can draw a polygon mesh by making a *drawcall*[2]. GL supports five kinds of drawcalls. Let us consider two among them. For non-indexed mesh representations, `glDrawArrays` is used.

[2]It can be made only when we have also attached a fragment shader to the GL program. GL cannot draw anything without vertex and fragment shaders.

```
void  glDrawArrays(GLenum mode,
                   GLint first,
                   GLsizei count);
```

`mode`	`GL_TRIANGLES` for polygon mesh
`first`	start index in the vertex array
`count`	the number of vertices to draw

Suppose that the mesh shown in Fig. 3.11 is represented in the non-indexed fashion. Then, the vertex array would have 144 elements. When we call `glDrawArrays(GL_TRIANGLES, 0, 144)`, three vertices will be repeatedly read in linear order to make up a triangle and 48 triangles in total will be drawn.

```
void    glDrawElements(GLenum mode,
                       GLsizei count,
                       GLenum type,
                       const GLvoid * indices);
```

`mode`	`GL_TRIANGLES` for polygon mesh
`count`	the number of indices to draw
`type`	index type
`indices`	byte offset into the buffer bound to `GL_ELEMENT_ARRAY_BUFFER`

For the indexed representation of a mesh, `glDrawElements` is used. For the same mesh in Fig. 3.11, we would call `glDrawElements(GL_TRIANGLES, 144, GL_UNSIGNED_SHORT, 0)`.