Chapter 3
OpenGL ES and vertex shader
The GPU rendering pipeline is partitioned into *programmable* stages (vertex and fragment shaders) and *hard-wired* stages (“primitive assembler & rasterizer” and “output merger”).

The vertex shader operates on every input vertex stored in the vertex array and performs various operations.

The essential among the operations is applying a series of *transforms* to the vertices. Let us present the transform one by one.
**World Transform**

- Whereas the virtual world containing all component objects is associated with the *world space*, the coordinate system used for creating an object is named *object space*.

- The size, position, and orientation of an object in the world space are determined by what we call *world transform*. (OpenGL usually calls this *model transform*.) Its main components are scaling, translation, and rotation.

- A distinct object is associated with its own object space and consequently has its own world transform. In the example shown below, the world transform of the teapot is the combination of “rotation about the *y*-axis by 90 degrees” and “translation along the *x*-axis by seven units.”
**View Transform**

- Eye pose specification in the world space
  - **EYE**: eye’s position
  - **AT**: a reference point which the eye looks at
  - **UP**: view up vector describing where the top of the eye points
- Then, the *eye space*, \{\textbf{EYE}, \textit{u}, \textit{v}, \textit{n}\}, can be created, as presented in the box.
- We now have two valid spaces, eye space and world space, \{\textbf{O}, \textit{e}_1, \textit{e}_2, \textit{e}_3\}, where \(\textit{e}_1, \textit{e}_2,\) and \(\textit{e}_3\) make up the standard basis.

\[
\begin{align*}
\text{n} & = \frac{\text{EYE} - \text{AT}}{\|\text{EYE} - \text{AT}\|} \\
\text{u} & = \frac{\text{UP} \times \text{n}}{\|\text{UP} \times \text{n}\|} \\
\text{v} & = \text{n} \times \text{u}
\end{align*}
\]
View Transform (cont’d)

- A point is given different coordinates in distinct spaces. In the example, the red dot is on the end of the teapot’s mouth. Its coordinates are (10,2,0) and has been taken as AT. If EYE is located at (18,8,0), the distance between EYE and AT is 10 and so the red dot’s coordinate in terms of the eye space is (0,0,-10).

- If all the world-space objects can be newly defined in terms of the eye space in the manner of the teapot’s mouth end, it becomes much easier to develop the rendering algorithms. In general, it is called space change.

- The space change from the world space \( \{O, e_1, e_2, e_3\} \) to the eye space \( \{EYE, u, v, n\} \) is the view transform.
The view matrix shown below applies to the world-space object to transform it to the eye space.

\[
M_{\text{view}} = R_{\text{view}} T_{\text{view}} = \\
\begin{pmatrix}
  u_x & u_y & u_z & 0 \\
  v_x & v_y & v_z & 0 \\
  n_x & n_y & n_z & 0 \\
  0 & 0 & 1 & 0
\end{pmatrix} \begin{pmatrix}
  1 & 0 & 0 & -\text{EYE}_x \\
  0 & 1 & 0 & -\text{EYE}_y \\
  0 & 0 & 1 & -\text{EYE}_z \\
  0 & 0 & 0 & 1
\end{pmatrix}
\]

- Derivation of the matrix can be found in many computer graphics books.
**View Frustum**

- Let us simply denote the basis of the eye space by \( \{x, y, z\} \) instead of \( \{u, v, n\} \).
- Recall that, for constructing the view transform, we defined the external parameters of the camera, i.e., \( \text{EYE} \), \( \text{AT} \), and \( \text{UP} \). Now let us control the camera’s internals. It is analogous to choosing a lens for the camera and controlling zoom-in/zoom-out.
- Four parameters define a *view frustum*, which is a truncated pyramid.
  - \( \text{fovy} \) stands for the field of view along \( y \)-axis.
  - \( \text{aspect} \) for the field of view’s aspect ratio
  - \( n \) for the distance from \( \text{EYE} \) to the ‘near clipping plane’
  - \( f \) for the distance from \( \text{EYE} \) to the ‘far clipping plane’

\[
\text{aspect} = \frac{w}{h}
\]
**View Frustum (cont’d)**

- Anything out of the view frustum is considered invisible. The near and far clipping planes, which are defined by $n$ and $f$, respectively, run counter to the real-world camera or human vision system, but have been introduced for the sake of computational efficiency.

- If a polygon intersects the boundary of the view frustum, it is *clipped* with respect to the boundary, and only the portion inside the view frustum is processed for display.
**Projection Transform**

- It is not easy to clip the polygons with respect to the view frustum.
- If there is a transform that converts the view frustum to the *axis-aligned box*, and the transform is applied to the polygons of the scene, clipping the transformed polygons with respect to the box is much easier.
- It is called *projection transform* and the transformed object is said to be reside in the *clip space*.

![Projection Transform Diagram](image)
Projection Transform (cont’d)

- Projection transform matrix defined by the view frustum parameters.

\[
\begin{pmatrix}
\frac{\cot\left(\frac{\text{fovy}}{2}\right)}{\text{aspect}} & 0 & 0 & 0 \\
0 & \frac{\cot\left(\frac{\text{fovy}}{2}\right)}{0} & 0 & 0 \\
0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\
0 & 0 & -1 & 0
\end{pmatrix}
\]

- Derivation of the matrix is a little complicated and its process can be found in a few computer graphics books, such as *3D graphics for Game Programming*. 
OpenGL ES

- OpenGL ES is a subset of OpenGL.
  - OpenGL ES 2.0 is derived from OpenGL 2.0 and provides vertex and fragment shaders.
  - OpenGL ES 3.0 is derived from OpenGL 3.3 and adds many enhanced features to OpenGL ES 2.0.
  - OpenGL ES 3.1 (publicly released in March 2014) includes compute shaders.
  - OpenGL ES 3.2 (August 2015) includes geometry and tessellation shaders.
- Our lecture notes are for OpenGL ES 2.0.
  - OpenGL ES 2.0 API specification
  - OpenGL ES Shading Language Specification
- From now on, let’s call OpenGL ES simply ‘GL’ and OpenGL ES Shading Language by ‘SL.’
- A shader is an executable program running on the GPU and consists of a set of software instructions. Without vertex and fragment shaders, GL cannot draw anything.
Vertex Shader

The main inputs of vertex shader are *attributes* and *uniforms*:

- Attributes: Per-vertex data that are typically provided in *vertex array*.
- Uniforms: Read-only values such as the transform matrix to be uniformly applied to the vertices.

The outputs

- They must include the built-in variable, `gl_Position`, which stores the clip-space vertex position.
- They usually include normal, texture coordinates, etc.
### Vertex Shader (cont’d)

- **SL** is a C-like language but provides specialized types. For example, \texttt{mat4} is a 4x4 matrix and \texttt{vec3} is a 3D vector.
- The vertex shader shown below has five input values: \texttt{viewProjMat} and \texttt{worldMat} are ‘uniforms’ described by the keyword \texttt{uniform} whereas \texttt{position}, \texttt{normal}, and \texttt{texCoord} are ‘attributes’ described by the keyword \texttt{attribute}.

```cpp
uniform mat4 viewProjMat;  // 4x4 matrix for view+projection transforms
uniform mat4 worldMat;    // 4x4 matrix for world transform

attribute vec3 position;
attribute vec3 normal;
attribute vec2 texCoord;

varying vec3 v_normal;
varying vec2 v_texCoord;

void main() {
    gl_Position = viewProjMat * worldMat * vec4(position, 1.0);
    v_normal = mat3(worldMat) * normal;
    v_texCoord = texCoord;
}
```
In vertex shader, optional output variables are described with the keyword `varying`. The output usually includes vertex normal and texture coordinates.

In the example, the object-space vertex normal is transformed to the world space and output to `v_normal`. On the other hand, the texture coordinates, `texCoord`, is simply copied to `v_texCoord`.

```glsl
uniform mat4 viewProjMat;       // 4x4 matrix for view+projection transforms
uniform mat4 worldMat;         // 4x4 matrix for world transform

attribute vec3 position;
attribute vec3 normal;
attribute vec2 texCoord;

varying vec3 v_normal;
varying vec2 v_texCoord;

void main() {
    gl_Position = viewProjMat * worldMat * vec4(position, 1.0);
    v_normal = mat3(worldMat) * normal;
    v_texCoord = texCoord;
}
```
**Shader Object**

- **GL API**
  - GL commands begin with the prefix `gl`.
  - GL data types begin with the prefix `GL`.

- Given a vertex shader stored in a file, we do the following:
  - Its source code is loaded.
  - A new *shader object* is created using `glCreateShader`, which takes either `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER` and returns the handle to the shader object.
  - Taking the vertex shader’s source code and the shader object, `glShaderSource` stores the source code in the shader object.
  - The shader object is compiled using `glCompileShader`.

- The same process will be done for the fragment shader.
**Program Object**

- The shader objects (for the vertex and fragment shaders) should be attached to a *program object*, which is then linked into the final executable.
  - The *program object* is created by `glCreateProgram`, which takes no argument and simply returns a handle to a new program object.
  - The shader and program objects are given to `glAttachShader`, which attaches the shader object to the program object.

```
program object

vertex shader object
fragment shader object
```

- Then, the program object is linked by `glLinkProgram`.
- Finally, the program object will be used for rendering! For this purpose, `glUseProgram` is invoked.
Uniforms

- Uniforms are read-only constant values, which are declared at the global scope. They require the keyword, `uniform`.
- Suppose that the eye moves in the environment. Then, `viewProjMat` in our example vertex shader should be accordingly updated. On the other hand, if the scene objects continuously move, `worldMat` should be updated per frame. The GL program should update and provide them for the vertex shader.

```glsl
uniform mat4 viewProjMat;  // 4x4 matrix for view+projection transforms
uniform mat4 worldMat;     // 4x4 matrix for world transform

attribute vec3 position;
attribute vec3 normal;
attribute vec2 texCoord;

varying vec3 v_normal;
varying vec2 v_texCoord;

void main() {
    gl_Position = viewProjMat * worldMat * vec4(position, 1.0);
    v_normal = mat3(worldMat) * normal;
    v_texCoord = texCoord;
}
```
Uniforms (cont’d)

- For this purpose, we have to find the uniform locations which have been determined during the link phase.
- Given a uniform name in the shader, `glGetUniformLocation` returns its location (denoted by an integer) in the program.
  - `GLint glGetUniformLocation(GLuint program, const char* name)`
    - `program`: handle to the program object
    - `name`: the uniform name
- An example: `GLint mWHandle = glGetUniformLocation(mProgram, “worldMat”)` where `mProgram` denotes the program object and `worldMat` is the uniform name.
- Then, we use `glUniformMatrix4fv()` to load the uniform with the updated matrix. (A list of functions for loading various uniforms is available and is collectively named `glUniform*()`, which represents a set of variations including `glUniform3f()` and `glUniformMatrix4x3fv()`.)
Attributes and VBO

- The mesh data stored in a file (such as OBJ file) will be read into the vertex and index arrays of the GL program. (Suppose that `mVertexArray` and `mIndexArray` are the pointers to the arrays.)
- For rendering a mesh, we make a `drawcall` such as `glDrawArrays` or `glDrawElements`, which will be presented soon.
- The application’s address space where the vertex and index arrays reside is called the client memory or client space by convention.
- For every drawcall, the arrays would have to be ‘copied’ from the client memory to the GPU memory.
- It will be more efficient if we cache the data in the GPU memory. Instead of resending the arrays every time a mesh is drawn, they are transferred *once* and rendering will be done from the GPU memory cache.
- We use vertex buffer objects (VBO) to achieve this.
Attributes and VBO (cont’d)

- There are two types of buffer objects in GL:
  - *Array buffer object*, which is for the vertex array, is specified by `GL_ARRAY_BUFFER`.
  - *Element array buffer object*, which is for the index array, is specified by `GL_ELEMENT_ARRAY_BUFFER`. (The ‘element array’ is a misnomer, and you just take it as a synonym for the index array.)
Attributes and VBO (cont’d)

- Creating and binding VBO
  - Use `glGenBuffers(GLsizei n, GLuint *buffers)`, which is asked for `n` buffer objects and returns them in `buffers`. In general, `n` is 2: one for vertices and the other for indices.
  - Use `glBindBuffer(GLenum target, GLuint buffer)`, where `target` is either `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`.
  - The buffer object is filled with data using `glBufferBufferData(GLenum target, GLsizeiptr size, const void *data, GLenum usage)`
    - `target` is either `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`.
    - `data` points to the vertex or index array supplied by the application. (This will be either `mVertexArray` or `mIndexArray`.)
  - The vertex or index array is copied to the GPU and the client-memory storage can be released.
Attributes and VBO (cont’d)

```c
glGenBuffers(2, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, VBO[1]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW));
```
Attributes

- The vertex array is usually represented as an *array of structures*, where a structure contains all attributes, e.g., *position*, *normal*, and *texCoord*, of a vertex.

  \[
  \begin{array}{cccccccc}
  x & y & z & x & y & z & s & t \\
  x & y & z & x & y & z & s & t \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
  \end{array}
  \]

  position  normal  texCoord

- Such a structure should be presented to the shader.

- First, \texttt{glEnableVertexAttribArray(GLuint index)}

- Then, \texttt{void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *ptr)}
  
  - \texttt{index}: attribute index
  - \texttt{size} specifies the number (1, 2, 3, or 4) of components per attribute.
  - \texttt{stride} specifies the byte distance between the consecutive attributes of the same kind
  - \texttt{ptr} points to the first occurrence of the attribute in the array. It is also in bytes.
int stride = sizeof(VertexStruct); // VertexStruct with three attributes
int offset = 0;

// position attribute
glEnableVertexAttribArray(0); // position index = 0
glVertexAttribPointer(0, // index
    3, // size
    GL_FLOAT, // type
    GL_FALSE, stride, (GLvoid*)offset);

offset += sizeof(vec3); // for accessing normal
glEnableVertexAttribArray(1); // normal index = 1
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, stride, (GLvoid*)offset);

offset += sizeof(vec3); // for accessing texCoord
glEnableVertexAttribArray(2); // texCoord index = 2
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, stride, (GLvoid*)offset);
Drawcalls

- We have made all attributes and uniforms available. Suppose that we have a good fragment shader. Then, we can draw a polygon mesh.
- For rendering a polygon mesh, we can make a drawcall. We have two choices:
  - `glDrawArrays` for non-indexed mesh representation
  - `glDrawElements` for indexed mesh representation
- void `glDrawArrays(GLenum mode, GLint first, GLsizei count)`
  - `mode`: Valid values include `GL_POINTS`, `GL_LINES`, `GL_TRIANGLES`, `GL_TRIANGLE_FAN`, etc.
  - `first` specifies the starting index “in the vertex array.”
  - `count` specifies the number of vertices to be drawn.
- Assuming we have the vertex array only, we invoke the following for our low-resolution sphere, :
  - `glDrawArrays(GL_TRIANGLES, 0, 144);`
Now, consider the indexed mesh representation.

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

- `count` specifies the number of elements “in the index array”
- `type`: Valid values include `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, etc.
- `indices` points to the offset in bytes into the storage allocated by `glBufferData`.

Assuming we have both vertex array and index array, we invoke the following for our low-resolution sphere.

```c
glDrawElements(GL_TRIANGLES, 144, GL_UNSIGNED_SHORT, 0);
```
An Example in GLView

- Let us show an example code that presents how to use GL APIs in the callbacks of the GLView. In the init_glview callback, we compile and link the shaders and create VBOs.

```c
// GL Init function
static void init_glview(Evas_Object *glview) {
    // Set gl state color to black
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    appdata_s *ad = (appdata_s *) evas_object_data_get(glview, "ad");

    if (!ad->initialized) {
        init_shader_program(ad); // Compile and link shader
        create_vbo(ad); // Create vertex buffer object
        ad->texture = create_texture(ad->win, "tizen_noalpha.png");

        glEnable(GL_DEPTH_TEST);
        ad->initialized = EINA_TRUE;
    }

    // initialize application variables ...
}
```
static void init_shader_program(appdata_s *ad) {
    const char *p;
    p = vertex_tex_shader;
    ad->vtx_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(ad->vtx_shader, 1, &p, NULL);
    glCompileShader(ad->vtx_shader);
    
    p = fragment_tex_shader;
    ad->fgmt_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(ad->fgmt_shader, 1, &p, NULL);
    glCompileShader(ad->fgmt_shader);
    
    ad->program = glCreateProgram();
    glAttachShader(ad->program, ad->vtx_shader);
    glAttachShader(ad->program, ad->fgmt_shader);
    glLinkProgram(ad->program);
    
    ad->idx_a_position = glGetAttribLocation(ad->program, "a_position");
    ad->idx_a_tex = glGetAttribLocation(ad->program, "a_tex");
    
    ad->idx_wvp = glGetUniformLocation(ad->program, "u_wvpMatrix");
    ad->idx_tex = glGetUniformLocation(ad->program, "u_texSampler");
    
    glUseProgram(ad->program);
}
static void create_vbo(appdata_s *ad) {
    glGenBuffers(1, &ad->vbo_vertex);
    glGenBuffers(1, &ad->vbo_color);
    glGenBuffers(1, &ad->vbo_texture);

    glBindBuffer(GL_ARRAY_BUFFER, ad->vbo_vertex);
    glBufferData(GL_ARRAY_BUFFER, sizeof(cube_vertices), cube_vertices, GL_STATIC_DRAW);
    glVertexAttribPointer(ad->idx_a_position, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);

    glBindBuffer(GL_ARRAY_BUFFER, ad->vbo_texture);
    glBufferData(GL_ARRAY_BUFFER, sizeof(cube_texs), cube_texs, GL_STATIC_DRAW);
    glVertexAttribPointer(ad->idx_a_tex, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), 0);

    glEnableVertexAttribArray(ad->idx_a_position);
    glEnableVertexAttribArray(ad->idx_a_tex);

    glGenBuffers(1, &ad->vbo_index);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ad->vbo_index);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(cube_indices), cube_indices, GL_STATIC_DRAW);
}
In the draw_glview callback, we calculate the world, view, and projection matrices and then pass the combined matrix to the vertex shader via glUniform*().

```
// GL draw callback
static void draw_glview(Evas_Object *glview) {
    appdata_s *ad = (appdata_s *) evas_object_data_get(glview, "ad");
    float world[16], viewproj[16];
    float aspect;
    if (!ad) return;

    init_matrix(world);
    init_matrix(viewproj);

    aspect = (float) ad->glview_w / (float) ad->glview_h;
```
// view matrix is the identity matrix
view_set_perspective(viewproj, 60.0f, aspect, 1.0f, 20.0f);

translate_xyz(world, 0.0f, 0.0f, -2.5f);
rotate_xyz(world, ad->xangle, ad->yangle, 0.0f);
multiply_matrix(ad->wvp, viewproj, world);

glfwClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glfwUniformMatrix4fv(ad->idx_wvp, 1, GL_FALSE, ad->wvp);

glfwBindTexture(GL_TEXTURE_2D, ad->texture);
glfwActiveTexture(GL_TEXTURE0);
glfwUniform1i(ad->idx_tex, 0);

glfwDrawElements(GL_TRIANGLES, cube_indices_count, GL_UNSIGNED_SHORT, 0);

glfwFlush();
}