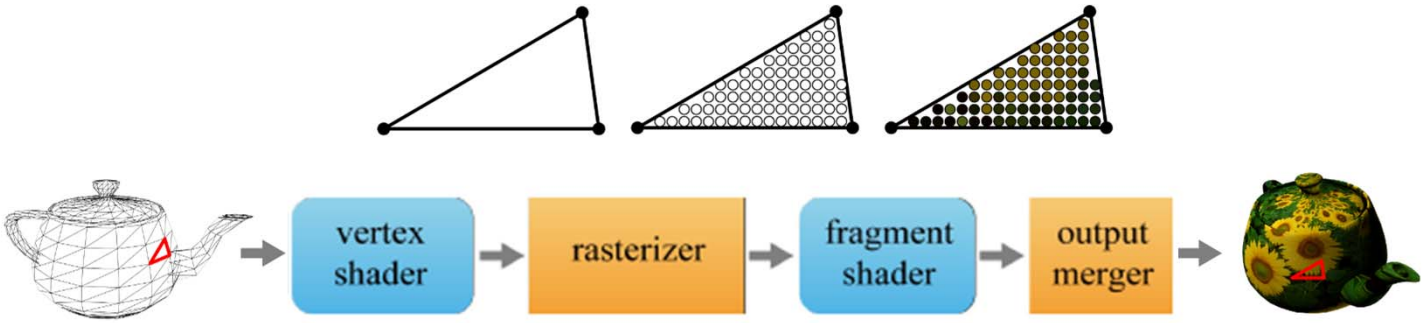
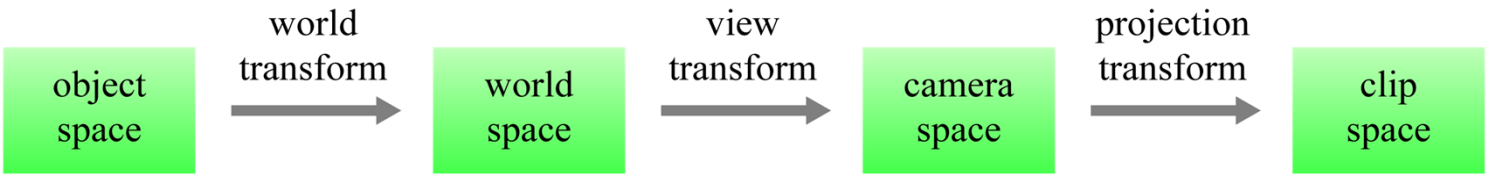
# Chapter VI
# Vertex Shader

# *GPU Rendering Pipeline (revisited)*

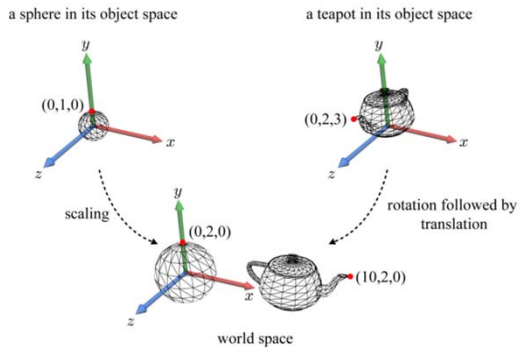
- Main stages in the *rendering pipeline*

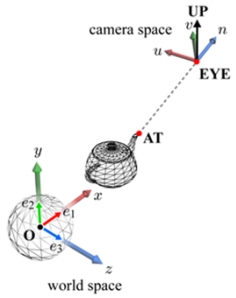


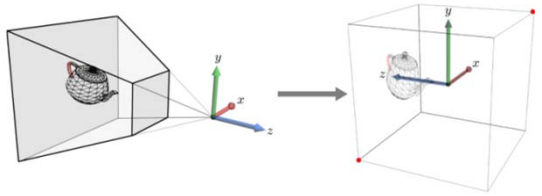

- The vertex shader (vertex program) operates on every input vertex stored in the vertex array and performs various operations.
- The essential among them is applying a series of *transforms* to the vertices.



# *Transforms (revisited)*



object 1
(object space 1)

world transform 1

object 2
(object space 2)

world transform 2

object 3
(object space 3)

world transform 3

…

world transform *n*

object *n*
(object space *n*)

scene
(world space)

view
transform

scene′
(camera space)

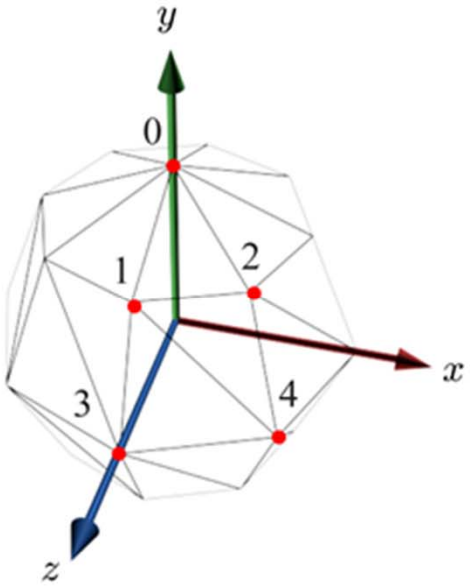
projection
transform

scene″
(clip space)

- The same world transform is applied to all vertices of an object.
- Each object has its own world transform, but all objects have common view and projection transforms.

# *Vertex and Index Arrays (revisited)*

- The indispensible components of vertex array are position, normal, and texture coordinates. Texture coordinates are 2D and used to access the 2D texture. (This is presented at CH 8.)
- Conceptually, the vertex shader processes a vertex (an element) of the vertex array at a time.




vertex array

| | | | |
|---|---|---|---|
| 0 | (0.000, 1.000, 0.000) | (0.000, 1.000, 0.000) | |
| 1 | (0.000, 0.707, 0.707) | (0.000, 0.663, 0.748) | |
| 2 | (0.500, 0.707, 0.500) | (0.529, 0.663, 0.529) | |
| 3 | (0.000, 0.000, 1.000) | (0.000, 0.000, 1.000) | |
| 4 | (0.707, 0.000, 0.707) | (0.707, 0.000, 0.707) | |
| | . . . | . . . | |
| 25 | (0.000, -1.000, 0.000) | (0.000, -0.707, -0.707) | |
| | position | normal | tex coord |

index array

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 1 |
| 4 | 3 |
| 5 | 4 |
| 6 | 4 |
| 7 | 2 |
| 8 | 1 |
| | . . . |
| 143 | 16 |

# *OpenGL ES*

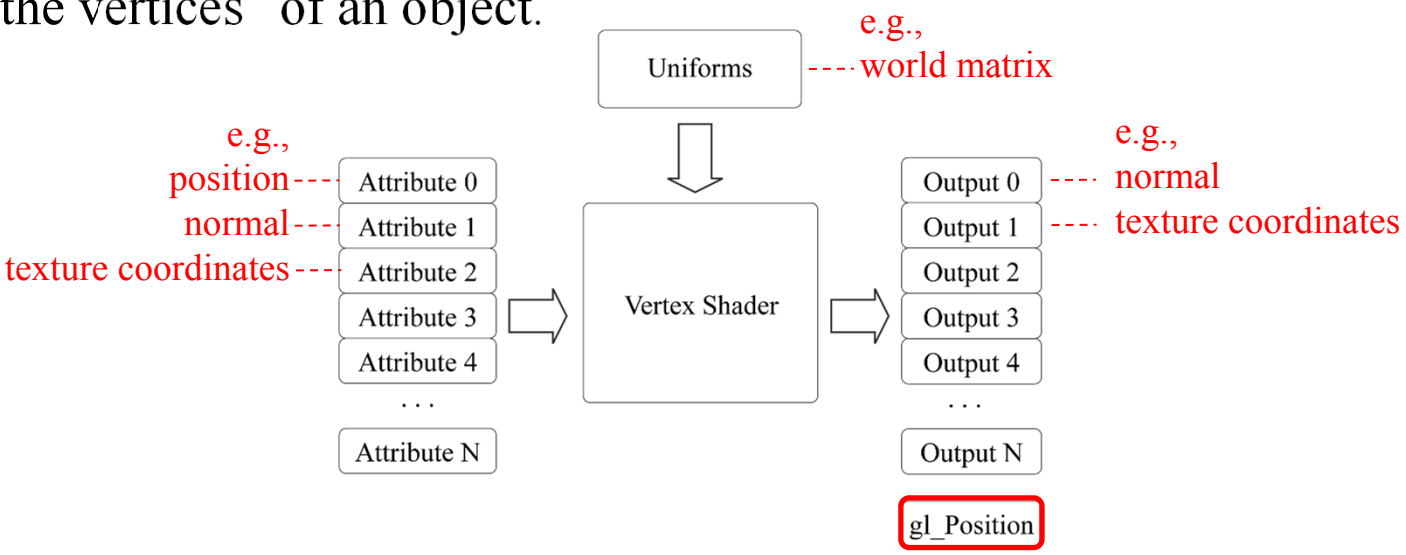
- OpenGL ES is a subset of OpenGL.
  - OpenGL ES 2.0 is derived from OpenGL 2.0 and provides vertex and fragment shaders.
  - OpenGL ES 3.0 is derived from OpenGL 3.3 and adds many enhanced features to OpenGL ES 2.0.
  - OpenGL ES 3.1 (publicly released in March 2014) includes compute shaders.
  - OpenGL ES 3.2 (August 2015) includes geometry and tessellation shaders.
- OpenGL ES 3.0 is backward compatible with OpenGL ES 2.0, i.e., applications built upon OpenGL ES 2.0 work with OpenGL ES 3.0.
- This class focuses on OpenGL ES 3.0.
  - OpenGL ES 3.0 API specification
  - OpenGL ES Shading Language Specification
- From now on, let's call OpenGL ES simply 'GL' and OpenGL ES Shading Language by 'SL.'

# *OpenGL ES Shading Language*

- SL is a C-like language. For example, `float` is supported.
- 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.
- Vector and matrix examples
  - `vec4` defines a floating-point 4D vector and `ivec3` defines an integer 3D vector.
  - `mat3` and `mat4` define 3x3 and 4x4 'square' matrices, respectively, whereas `mat3x4` is for a 3x4 matrix. The matrix elements are all `float` values.

# *Vertex Shader*

- A vertex shader per object!
- The main input
  - 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 "of an object."




- The output
  - They must include the built-in variable, `gl_Position` that stores the clip-space vertex position.
  - In addition, they usually include normal, texture coordinates, etc.

# *Vertex Shader (cont'd)*

- Our first vertex shader

```
#version 300 es

uniform mat4 worldMat, viewMat, projMat;

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
layout(location = 2) in vec2 texCoord;

out vec3 v_normal;
out vec2 v_texCoord;

void main() {
   gl_Position = projMat * viewMat * worldMat * vec4(position, 1.0);
   v_normal = normal;
   v_texCoord = texCoord;
}
```

- Attributes are read-only values, which require the keyword, `in`.
- The layout qualifiers such as `layout(location = 0)` specify the *attribute locations*. (The layout qualifiers are optional though.)

# *GL Program*

- The fragment shader is written in a similar fashion.
- While the vertex and fragment shaders are in charge of low-level details in rendering, the role of GL program itself is managing the shaders and various data needed for the shaders.

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

# GL Program - Shader Object

- Given a vertex shader stored in a file, we do the following:
  - Its source code is loaded. Assume that it is pointed by `source` that is of type `char*`.
  - A new *shader object* is created using `glCreateShader`, which takes either `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER` and returns the unsigned integer ID of 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`.

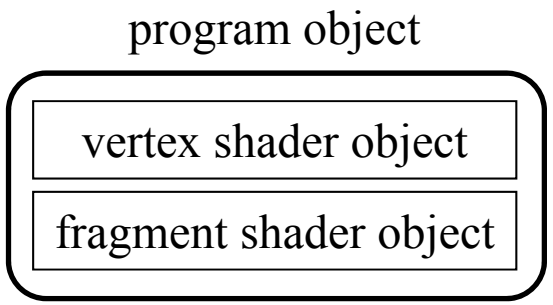
```
GLuint shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(shader, 1, &source, NULL);
glCompileShader(shader);
```

- The same process has to be done for the fragment shader using `GL_FRAGMENT_SHADER` for `glCreateShader`.
- Now you have two shader objects (for the vertex and fragment shaders) .

# GL Program - 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 the ID of the new program object.
  - The shader and program objects are given to `glAttachShader`, which attaches the shader object to the program object.
  - Then, the program object is linked by `glLinkProgram`.
  - Finally, the program object will be used for rendering! For this purpose, `glUseProgram` is invoked.

```
GLuint program = glCreateProgram();
glAttachShader(program, shader);
glLinkProgram(program);
glUseProgram(program);
```




- The above code is for vertex shader and so `glAttachShader` should be called with the fragment shader object.

# *Attributes*

- The GL program not only hands over the attributes and uniforms to the vertex shader but also informs the vertex shader of their structures.
- Like the shader source codes, 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 `vertices` and `indices` are the pointers to the arrays.

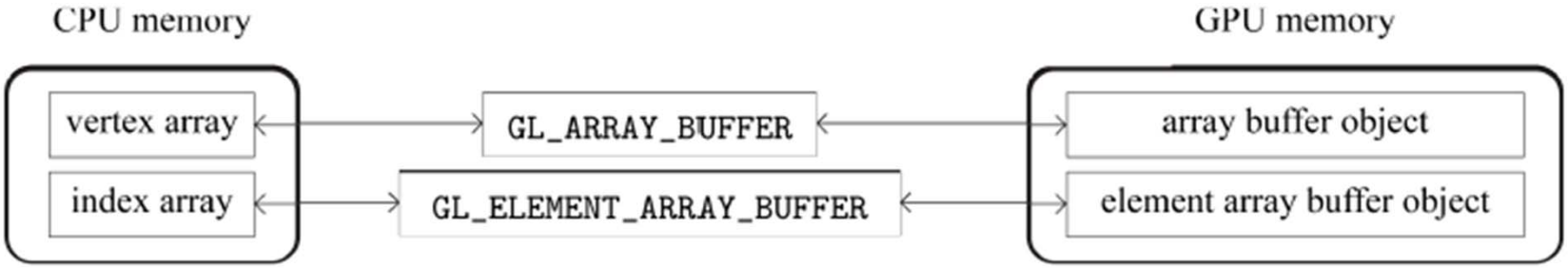
```
struct Vertex
{
    glm::vec3 pos; //position
    glm::vec3 nor; //normal
    glm::vec2 tex; //texture coordinates
};

std::vector<Vertex> vertices;
std::vector<GLushort> indices;
```

- `glm` stands for OpenGL Mathematics. It is a library that provides classes and functions with the same naming conventions and functionalities as SL.

# *Attributes (cont'd)*

- The address space where the arrays reside is called the *client memory* by convention. The arrays in the client 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:
  - Array buffer object is for the vertex array and is specified by GL_ARRAY_BUFFER.
  - Element array buffer object is for the index array and is specified by GL_ELEMENT_ARRAY_BUFFER.



# *Attributes (cont'd)*

- Creating and binding VBO
  - `glGenBuffers(Glsizei n, Gluint *buffers)` is asked for `n` buffer objects and returns them in `buffers`.
  - `glBindBuffer(GLenum target, GLuint buffer)` - `target` is either `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`.
  - The buffer object is filled with data using `glBufferData(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 `vertices` or `indices`.)

```
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex),
vertices.data(), GL_STATIC_DRAW);
```

# *Attributes (cont'd)*

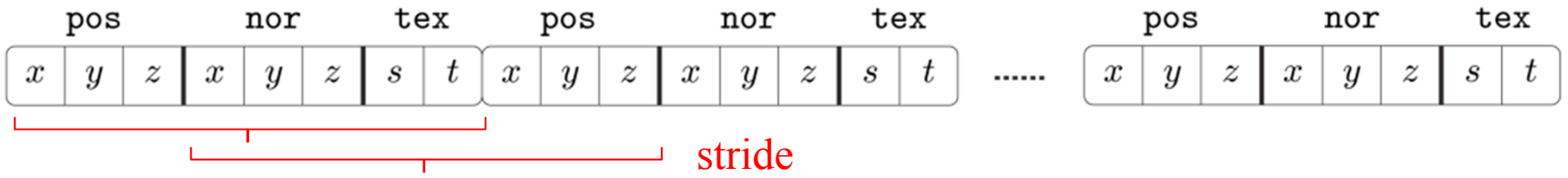
- The same process is done for the index array.

```
GLuint ibo;
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLushort),
indices.data(), GL_STATIC_DRAW);
```

- OK, now we have the VBO for vertex and index arrays.

# *Attributes (cont'd)*

- The vertex array can be described as an *array of structures*, where a structure contains all attributes (`pos`, `nor`, and `tex`) of a vertex.




- The GL program uses `glEnableVertexAttribArray` and `glVertexAttribPointer` to inform the vertex shader of such a structure.

```
int stride = sizeof(Vertex);
int offset = 0;

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

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

offset += sizeof(glm::vec3); // for accessing texture coordinates
glEnableVertexAttribArray(2); // texture coordinates = attribute 2
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, stride, (GLvoid*)offset);
```

# *Attributes (cont'd)*

```
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 |

# *Uniforms*

- Our vertex shader has three uniforms: `worldMat`, `viewMat`, and `projMat`.
- Consider a dynamic environment.
  - If the scene object moves, `worldMat` should be changed.
  - If the viewpoint moves, `viewMat` should be changed.
- The GL program updates and provides them for the vertex shader.
- For this purpose, we first have to use `glGetUniformLocation` to find the uniform's 'location' in the program object, which has been determined during the link phase.
- In order to load the uniform, `worldMat`, for example, we use `glUniformMatrix4fv`, where `4` indicates a 4x4 matrix, `f` indicates that its elements are floating-point values, and `v` implies that the values are passed in a vector, i.e., an array:

```
glm::mat4 worldMatrix;
GLint loc = glGetUniformLocation(program, "worldMat");
glUniformMatrix4fv(loc, 1, GL_FALSE, &worldMatrix);
```

# *Uniforms (cont'd)*

```
GLint   glGetUniformLocation(GLuint program,
                             const GLchar *name);
```

| | |
|---|---|
| `program` | program object |
| `name` | uniform name in a string |

```
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 |

# *Drawcalls*

- We have made all attributes and uniforms available.
- Suppose that we have a good fragment shader, whatever it is.
- 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` `GL_TRIANGLES` for polygon mesh
`first` start index in the vertex array
`count` the number of vertices to draw

# *Drawcalls (cont'd)*

- For indexed representation:

```
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 |

```
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_SHORT, 0);
```

- In the example, the index array has 144 elements and the vertex array has 26 elements. Therefore `indices.size()` returns 144.