
Computer Graphics

2 - Lab - Hello Triangle!

Yoonsang Lee
Hanyang University

Spring 2026

Outline

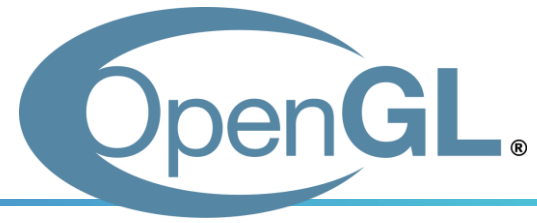
- Introduction to OpenGL and other modules
 - What is OpenGL?
 - OpenGL Versions
 - GLFW input handling
 - PyGLM, NumPy
- Shaders
 - OpenGL Rendering Pipeline
 - Vertex Shader
 - Fragment Shader
 - GLSL
- Hello Triangle!
 - Preparing Vertex Data and Drawing Using It
 - Preparing and Using Shader Program Object
- Time for Assignment

Note for CSE4020 Code Repository

- <https://github.com/yssl/CSE4020>
- Download the source code used in labs
- Modify, run, test — and play!

Introduction to OpenGL and other modules

What is OpenGL?



- **OpenGL: Open Graphics Library**
 - Unlike its name, OpenGL is not a library.
- OpenGL is a **specification** for a **rasterization-based real-time 3D rendering system**.
 - OpenGL is an "open" specification. Anyone can download it for free.
- OpenGL defines an **API** that allows applications to control the rendering system.
 - API: a set of functions and rules that define how applications use the system and its expected behavior.
- OpenGL rendering system is carefully specified to make hardware implementations allowable.

Who Develops / Maintains OpenGL?

- OpenGL specification:
 - Maintained by the Khronos Group (OpenGL ARB)
- OpenGL implementation:
 - Hardware vendors (e.g., NVIDIA, AMD, Intel) implement OpenGL to work on their GPUs.
 - Their implementations, called "drivers", translate OpenGL API commands into their GPU commands.
- OpenGL defines *what should happen*, drivers implement *how it happens*.

Characteristics of OpenGL

- Cross-platform
 - OpenGL is designed to be platform-independent.
 - You can use OpenGL on Windows, Linux, MacOS, Android, and iOS (*OpenGL deprecated, replaced by Metal*).
- Language independent
 - OpenGL is defined as a C API, but bindings exist for many languages (C, Python, Java, Javascript, ...)
 - We'll use a Python binding for OpenGL: PyOpenGL

OpenGL Versions

- Legacy OpenGL (OpenGL 1.x, 2.x)
 - Invented when “fixed-function” hardware was standard
 - No programmable shaders (until OpenGL 2.0)
 - Easier to learn & good for rapid prototyping
 - Deprecated since OpenGL 3.0
- Modern OpenGL (OpenGL 3.x~)
 - Modern GPUs are fully programmable
 - Use of programmable shaders extensively
 - More difficult but far more flexible & powerful
 - Latest version: OpenGL 4.6 (2017)

OpenGL Version in This Course

- This course uses **OpenGL 3.3 Core Profile**.
 - All labs are based on this version.
 - All lab checks and projects must use this version.
- Ensure your laptop supports **OpenGL 3.3 or higher**.
 - Refer to "Checking OpenGL Version" in the first lecture slide, 1 - Course Intro.

(FYI) Legacy OpenGL Features Deprecated in OpenGL Core Profile

- In the OpenGL Core Profile, the old immediate mode and fixed-function pipeline are not available.
- Immediate mode
 - glBegin(), glEnd(), glVertex*, glNormal*, ...
- Fixed-function lighting / materials
 - glColor*, glLight*, glMaterial*, ...
- Fixed-function transformations & matrix stack operations:
 - glRotate*, glTranslate*, glScale*
 - glMatrixMode(), glLoadIdentity(), glPushMatrix(), glPopMatrix(),...

Utility Libraries for Learning OpenGL

- What can we do with OpenGL?
 - Only drawing primitives
 - Provides low-level drawing operations
 - No functions for creating windows, handling events, or managing OpenGL contexts.
 - OpenGL context stores the current OpenGL state (textures, buffers, enabled states such as GL_BLEND, GL_DEPTH_TEST, etc.)
- So, we need additional utility libraries to use OpenGL.
 - GLFW, FreeGLUT: lightweight OpenGL helpers
 - FLTK, wxWidgets, Qt, GTK: general GUI frameworks
- We'll use Python binding for GLFW: glfw.
 - General GUI frameworks may be too heavy, while GLFW provides fine control.

[Code] 1-first-gl-program

```
from OpenGL.GL import *
from glfw.glfw import *

def key_callback(window, key, scancode, action, mods):
    if key==GLFW_KEY_ESCAPE and action==GLFW_PRESS:
        glfwSetWindowShouldClose(window, GLFW_TRUE);

def main():
    # initialize glfw
    if not glfwInit():
        return

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3)    # OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3)
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE) # Do not
allow legacy OpenGL API calls
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE) # for macOS

    # create a window and OpenGL context
    window = glfwCreateWindow(800, 800, '1-first-gl-program', None, None)
    if not window:
        glfwTerminate()
        return

    glfwMakeContextCurrent(window)

    # register key callback for escape key
    glfwSetKeyCallback(window, key_callback);
```

[Code] 1-first-gl-program

```
# loop until the user closes the window
while not glfwWindowShouldClose(window):
    # render

    # swap front and back buffers (double buffering)
    glfwSwapBuffers(window)

    # poll events
    glfwPollEvents()

# terminate glfw
glfwTerminate()

if __name__ == "__main__":
    main()
```

GLFW Input Handling

- `glfwPollEvents()`
 - Processes events received by the application.
 - Calls a user-registered callback function for for each event type.

Event type	Set a callback using...
Key input	<code>glfwSetKeyCallback()</code>
Mouse cursor position	<code>glfwSetCursorPosCallback()</code> or poll the position using <code>glfwGetCursorPos()</code>
Mouse button	<code>glfwSetMouseButtonCallback()</code>
Mouse scroll	<code>glfwSetScrollCallback()</code>

[Code] 2-glfw-input-handling

```
from OpenGL.GL import *
from glfw.GLFW import *

def key_callback(window, key, scancode, action, mods):
    if key==GLFW_KEY_ESCAPE and action==GLFW_PRESS:
        glfwSetWindowShouldClose(window, GLFW_TRUE);
    elif key==GLFW_KEY_A:
        if action==GLFW_PRESS:
            print('press a')
        elif action==GLFW_RELEASE:
            print('release a')
        elif action==GLFW_REPEAT:
            print('repeat a')
    elif key==GLFW_KEY_SPACE and action==GLFW_PRESS:
        print ('press space: (%d, %d)'%glfwGetCursorPos(window))

def cursor_callback(window, xpos, ypos):
    print('mouse cursor moving: (%d, %d)'%(xpos, ypos))

def button_callback(window, button, action, mod):
    if button==GLFW_MOUSE_BUTTON_LEFT:
        if action==GLFW_PRESS:
            print('press left btn: (%d, %d)'%glfwGetCursorPos(window))
        elif action==GLFW_RELEASE:
            print('release left btn: (%d, %d)'%glfwGetCursorPos(window))
```

[Code] 2-glfw-input-handling

```
def scroll_callback(window, xoffset, yoffset):
    print('mouse wheel scroll: %d, %d'%(xoffset, yoffset))

def main():
    ...

    # register event callbacks
    glfwSetKeyCallback(window, key_callback);
    glfwSetCursorPosCallback(window, cursor_callback)
    glfwSetMouseButtonCallback(window, button_callback)
    glfwSetScrollCallback(window, scroll_callback)

    # loop until the user closes the window
    while not glfwWindowShouldClose(window):
        # swap front and back buffers
        glfwSwapBuffers(window)

        # poll events
        glfwPollEvents()

    # terminate glfw
    glfwTerminate()

if __name__ == "__main__":
    main()
```

Documentation for glfw

- GLFW C API documentation:
 - <http://www.glfw.org/documentation.html>
- If you import glfw by
`'from glfw.GLFW import *'`
you can use the same function and constant names as in the GLFW C API.
- Some return values differ in Python.
e.g., `glfwGetMonitors()` returns a **list** instead of (pointer + count).

PyGLM

- GLM (OpenGL Mathematics)
 - C++ mathematics library based on GLSL (OpenGL Shading Language)
 - Follows GLSL-style vector and matrix operations
- PyGLM
 - Python extension of GLM
- Documentation
 - Wiki: <https://github.com/Zuzu-Typ/PyGLM/wiki>
 - Function reference: <https://github.com/Zuzu-Typ/PyGLM/blob/master/wiki/function-reference/README.md>

Quiz 1

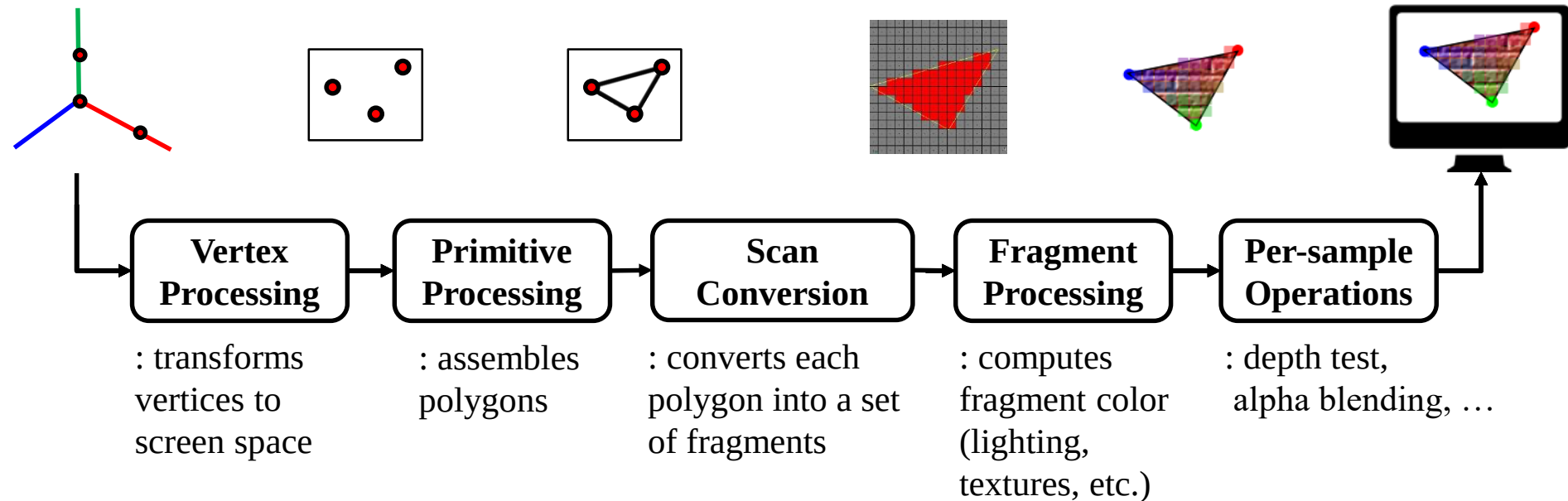
- Go to <https://www.slido.com/>
- Join #cg-ys
- Click "Polls"
- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. **2024123456: 4.0**
- Note that your quiz answer must be submitted **in the above format** to be counted for attendance.

NumPy

- General-purpose numerical computing library for Python
- NumPy vs PyGLM
 - PyGLM: math library designed for PyOpenGL
 - NumPy: general numerical computing (e.g., linear algebra)
- In this course
 - Mainly use PyGLM
 - NumPy used in some cases (especially early labs)

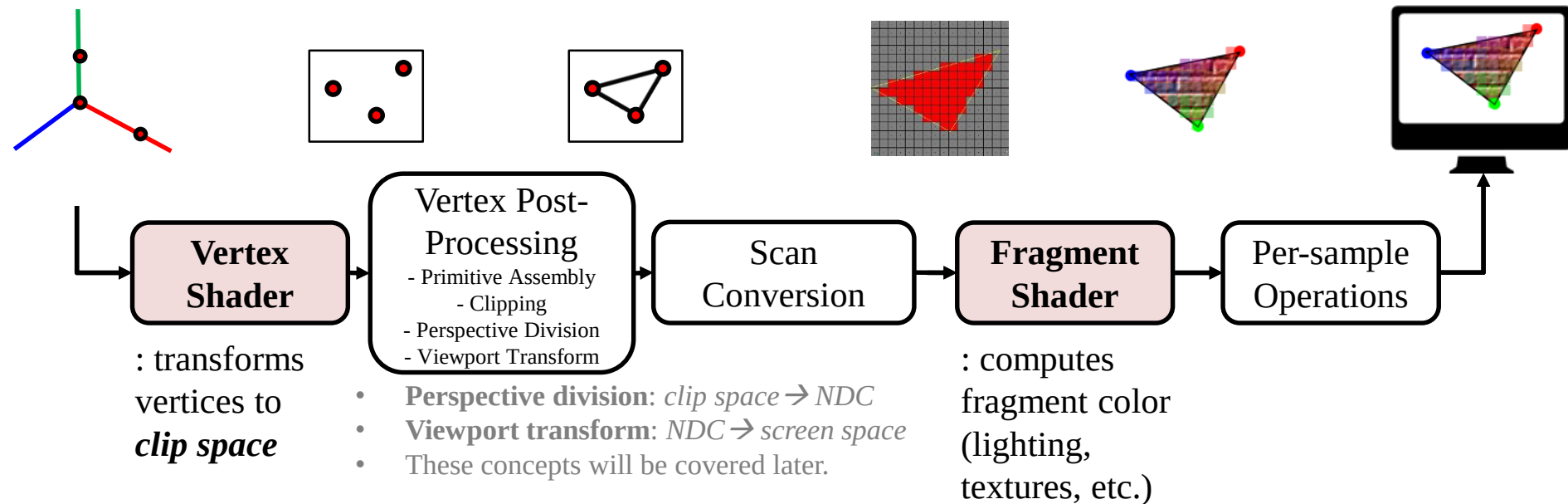
Shaders

Recall: Rasterization Pipeline



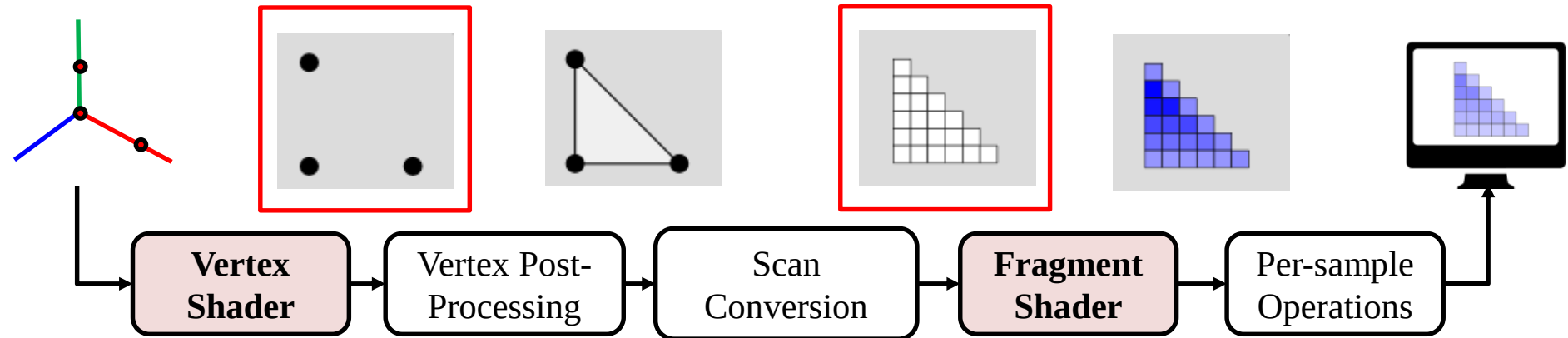
- *A.k.a. rendering pipeline or graphics pipeline.*

OpenGL Rendering Pipeline (simplified version)



- *Shader*: a small program that runs on GPU.
- The red blocks are *programmable* steps.
 - There are additional optional shader stages in the OpenGL pipeline (not covered in this course).
- Modern OpenGL requires at least a *vertex shader* and a *fragment shader*.

OpenGL Shaders



- Vertex shader runs once per vertex (**in parallel**).
- Fragment shader runs once per fragment (**in parallel**).
- In the example above:
 - The vertex shader runs 3 times
 - The fragment shader runs 21 times
 - All executions happen **in parallel** (modern GPUs have thousands of cores)

GLSL – OpenGL Shading Language

- OpenGL shaders are written in GLSL.
- GLSL is a C-style language with syntax similar to C.
- We use GLSL 3.30 core profile by specifying the following directive at the top of the shader code:

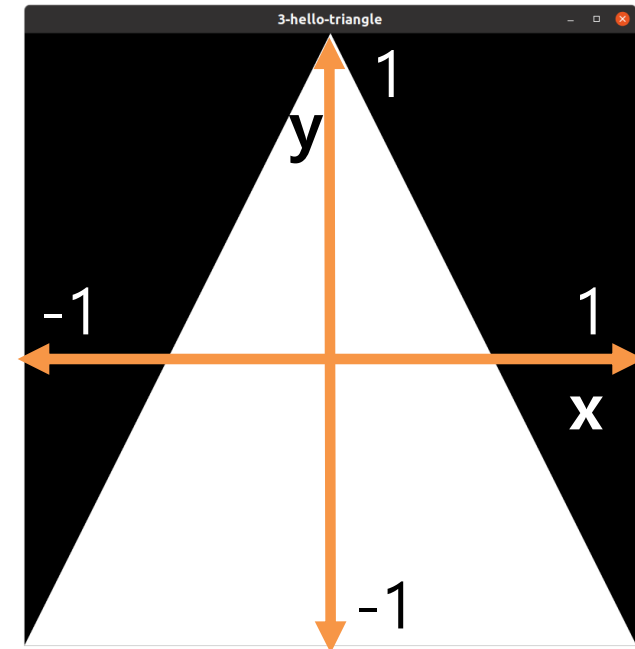
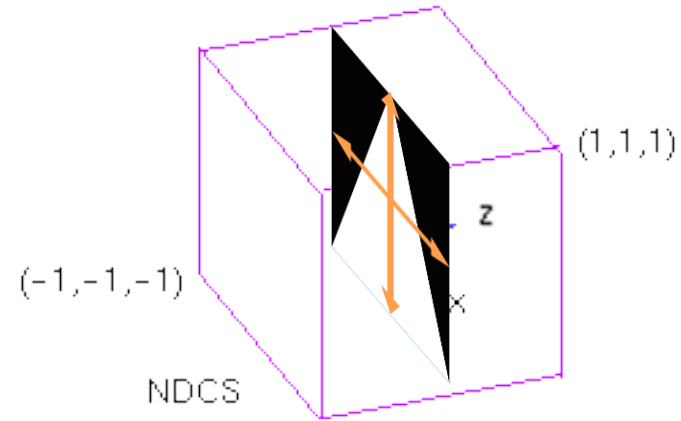
```
#version 330 core
```

Vertex Shader

- Input: *Per-vertex* attributes (provided by the application)
 - **Position** (vertex position)
 - Other attributes such as color, normal, texture coordinates
- Output: *Per-vertex* outputs
 - **Position in clip space** (`gl_Position`)
 - vertex post-processing
 - Other outputs such as color, normal, texture coordinates
 - fragment shader
- (We will look at the gray parts later.)

OpenGL Clip Space / NDC

- Objects are mapped into a cube where x, y, z range from -1 to 1 .
→ Clip Space / NDC
- Its xy plane corresponds to the 2D screen.
- We will look at this in more detail later.



Basic Vertex Shader Example

```
#version 330 core

// input vertex position. its attribute index is 0.
layout (location = 0) in vec3 vin_pos;

void main()
{
    // gl_Position: built-in output variable of type vec4 to
    // which vertex position in clip space is assigned.
    gl_Position = vec4(vin_pos.x, vin_pos.y, vin_pos.z, 1.0);
}
```

- Only vertex position is used as input and output in this example.
- `vec4 (x, y, z, w)`: `w` component is used for "perspective division" in perspective projection, which will be covered in later.
- For now, you can think of `w` as simply `1.0`.

Fragment Shader

- Input: *Interpolated* vertex shader outputs
 - To "**link**" vertex shader output and fragment shader input, they must have the same type and name.
 - Fragment shader input is **interpolated** vertex shader output at the fragment location.
- Output: **Fragment color**
- (We will look at the gray part later.)

Basic Fragment Shader Example

```
#version 330 core

// output fragment color of type vec4.
out vec4 FragColor;

void main()
{
    // set the fragment color to white.
    FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

- This example only outputs a fragment color. There is no input.
- vec4 (**r**ed, **g**reen, **b**lue, **a**lpha): alpha means opacity. (1.0 being completely opaque)

GLSL Data Types

- Scalar
 - `float`, `int`, `bool`, ...
- Vector
 - `vec2`, `vec3`, `vec4` - float vector
 - `ivec2`, `ivec3`, `ivec4` - int vector
 - `bvec2`, `bvec3`, `bvec4` - bool vector
- Matrix
 - `mat2`, `mat3`, `mat4` - 2x2, 3x3, 4x4 float matrix

GLSL Component Access

Components of vectors are accessed by array indexing with the `[]`-operator (indexing starts with 0) or with the `.`-operator and the element names `x, y, z, w` or `r, g, b, a` or `s, t, p, q`:

```
vec4 v = vec4(1.1, 2.2, 3.3, 4.4);
float a = v[3]; // = 4.4
float b = v.w; // = 4.4
float c = v.a; // = 4.4
float d = v.q; // = 4.4
```

It is also possible to construct new vectors by extending the `.`-notation ("swizzling"):

```
vec4 v = vec4(1.1, 2.2, 3.3, 4.4);
vec3 a = v.xyz; // = vec3(1.1, 2.2, 3.3)
vec3 b = v.bgr; // = vec3(3.3, 2.2, 1.1)
vec2 c = v.tt; // = vec2(2.2, 2.2)
```

Matrices are considered to consist of column vectors, which are accessed by array indexing with the `[]`-operator. Elements of the resulting (column) vector can be accessed as discussed above:

```
mat3 m = mat3(
    1.1, 2.1, 3.1, // first column
    1.2, 2.2, 3.2, // second column
    1.3, 2.3, 3.3 // third column
);
vec3 column3 = m[2]; // = vec3(1.3, 2.3, 3.3)
float m20 = m[2][0]; // = 1.3
float m21 = m[2].y; // = 2.3
```

* This image is from https://en.wikibooks.org/wiki/GLSL_Programming/Vector_and_Matrix_Operations

GLSL Operators

- Operators behave like in C.
- Most operators are component-wise, except...
- Multiplication operator (*)
 - **mat * mat** – matrix-matrix multiplication
 - **mat * vec** – matrix-vector multiplication
 - **vec * vec** – component-wise multiplication (not dot product)
 - `dot(a, b)` – dot product
 - **scalar*vec, scalar*mat** – scaling

GLSL Reference

- GLSL Programming/Vector and Matrix Operations
 - https://en.wikibooks.org/wiki/GLSL_Programming/Vector_and_Matrix_Operations

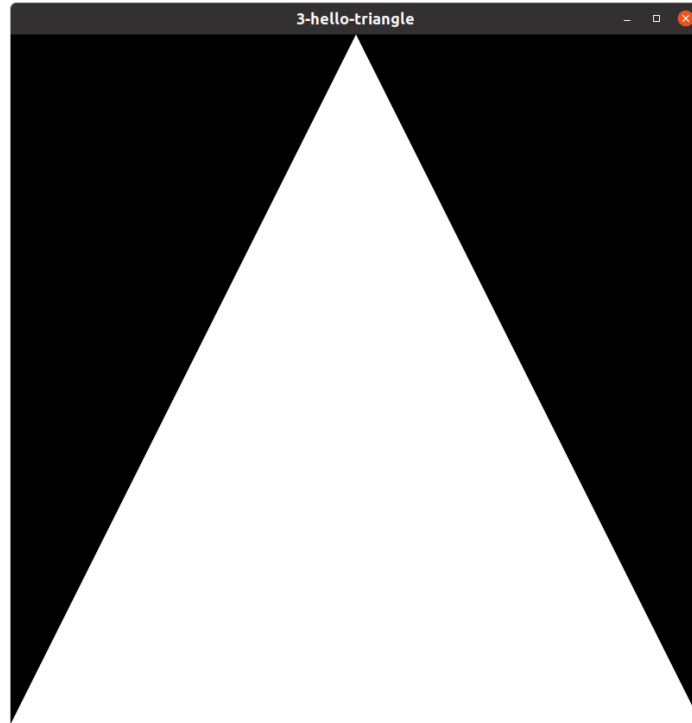
Quiz 2

- Go to <https://www.slido.com/>
- Join #cg-ys
- Click "Polls"
- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. 2024123456: 4.0
- Note that your quiz answer must be submitted **in the above format** to be counted for attendance.

Hello Triangle!

Goal

- Our goal is to draw the following white triangle.



- You can download and run "2-Lab-HelloTriangle/3-hello-triangle.py" from <https://github.com/yssl/CSE4020>.

Preparing Vertex Data and Drawing Using It

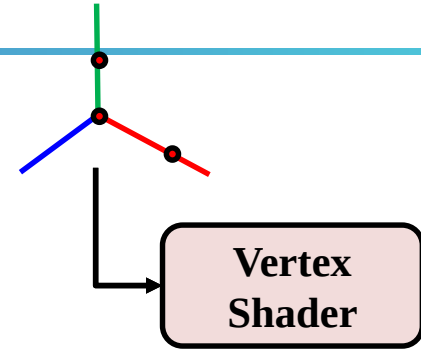
- Preparation (at initialization):
 - 1. Create vertex data (in main memory)
 - 2. Create and activate VAO
 - 3. Create and activate VBO
 - 4. Upload vertex data to VBO
 - 5. Configure vertex attributes
- Drawing (at every rendering frame):
 - 1. Activate VAO
 - 2. Call `glDrawArrays()`

P1. Create vertex data (in main memory)

- Vertex shaders require vertex input data.

```
import glm
# ...

vertices = glm.array(glm.float32,
    -1.0, -1.0, 0.0, // vertex 0
    1.0, -1.0, 0.0, // vertex 1
    0.0, 1.0, 0.0  // vertex 2
)
```



- In this example:
 - We specify three vertices with their x, y, z coordinates.
 - All vertices lie on the xy plane ($z=0$).

P2. Create and activate VAO

- Vertex Array Object (VAO):
 - An OpenGL object that stores the **"state" related to vertex data.**
- A VAO stores:
 - **Vertex attribute configuration** specified by `glVertexAttribPointer()`
 - **Pointers to VBOs** that store vertex attribute data via `glVertexAttribPointer()`
 - **Whether each vertex attribute is enabled or not** via `glEnableVertexAttribArray()`
- (We will look at VBOs and vertex attribute configuration in the following slides.)

P2. Create and activate VAO

- ```
VAO = glGenVertexArrays(1) # create a vertex array
object and store its ID in the VAO variable
glBindVertexArray(VAO) # activate VAO
```
- `glGenVertexArrays(n)`
  - Generate n vertex array object IDs and return them
- `glBindVertexArray(vao)`
  - Activate the specified VAO (and other VAOs are not activated)
  - That means, **any subsequent vertex attribute configuration will be stored in this VAO.**
- Meaning of "**Bind**" in OpenGL
  - "Bind" basically means associating two things.
  - `glBind*()` functions associate specified objects to current OpenGL context.
  - **Activate the specified object**

# P3. Create and activate VBO

- Vertex Buffer Object (VBO):
  - A buffer object used to **store vertex data (vertex attributes)**, such as vertex positions, colors, and normals for rendering.
- Why use VBO?
  - OpenGL cannot efficiently access vertex data in CPU memory.
    - We allocate memory on the GPU and upload vertex data to it using a VBO.
  - Faster rendering.
    - VBOs are usually allocated on very fast GPU memory, so vertex shaders can have instant access to vertex data.
    - By storing vertex data in a VBO, we reduce the amount of data transferred between the CPU and the GPU.

# P3. Create and activate VBO

- ```
VBO = glGenBuffers(1)    # create a buffer object
                             and store its ID in the VBO variable
glBindBuffer(GL_ARRAY_BUFFER, VBO)  # bind VBO as
a vertex buffer (GL_ARRAY_BUFFER)
```
- `glGenBuffers(n)`
 - Generate `n` buffer object IDs and return them
- `glBindBuffer(target, buffer)`
 - Activate buffer object for the specified target
 - target: `GL_ARRAY_BUFFER` is used for vertex buffer objects

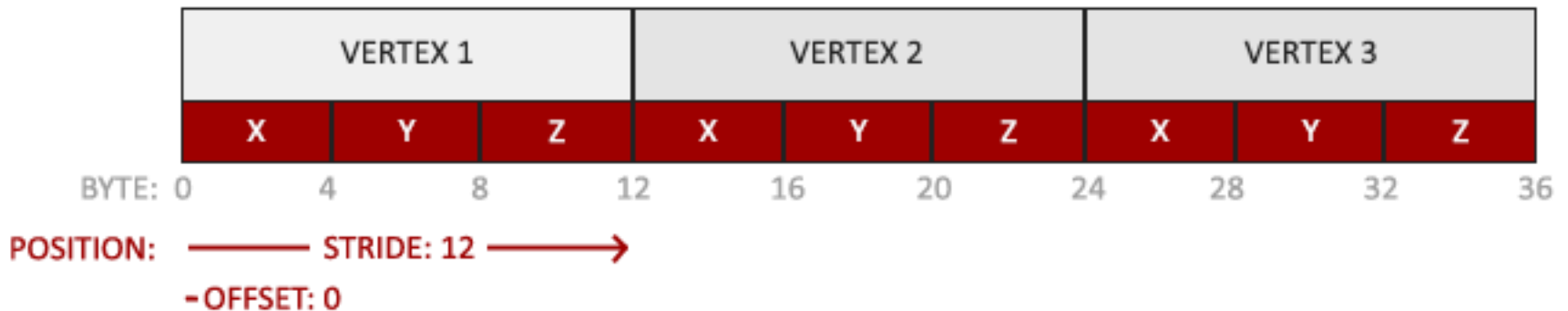
P4. Upload vertex data to VBO

- `glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices.ptr, GL_STATIC_DRAW)`
- `glBufferData(target, size, data, usage)`
 - Allocate GPU memory for the currently bound buffer (VBO in our example) and upload vertex data to it.
 - `target`: `GL_ARRAY_BUFFER` for vertex buffer objects
 - `size`: Size of the data (in bytes) that will be copied
 - `data`: A pointer to the data that will be copied
 - `usage`:
 - `GL_STATIC_DRAW`: for data that rarely changes
 - `GL_DYNAMIC_DRAW`: for data that changes frequently
 - `GL_STREAM_DRAW`: for data that changes every frame
 - Note that `usage` is only a hint for the GPU driver.
 - to help the GPU driver make more intelligent decisions.

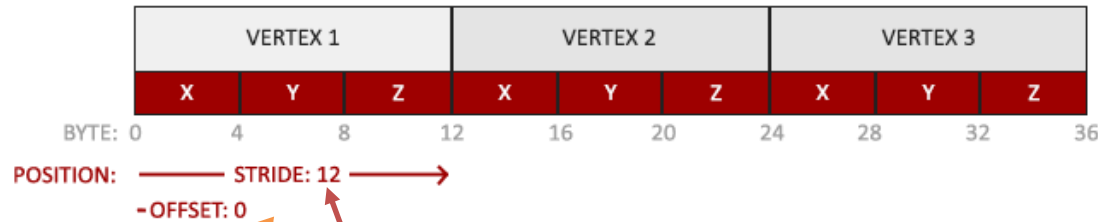
P5. Configure vertex attributes

- OpenGL does not yet know:
 - how to interpret the vertex data in VBOs
 - how to map the vertex data to the vertex shader attributes
- In our example, vertex data (in VBO) is formatted as follows:

```
vertices = glm.array(glm.float32,  
    -1.0, -1.0, 0.0, // vertex 0  
    1.0, -1.0, 0.0, // vertex 1  
    0.0, 1.0, 0.0 // vertex 2  
)
```



P5. Configure vertex attributes



- ```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
3*glm::sizeof(glm::float32), None)
glEnableVertexAttribArray(0)
```
- ```
glVertexAttribPointer(index, size, type, normalized, stride,
pointer)
```

 - Specify how OpenGL should interpret the vertex data in the currently bound VBO and map it to the vertex shader attributes.
 - index**: Vertex attribute index to configure (see the next page)
 - size**: Number of components per attribute (3 for vec3 data)
 - type**: Data type of each component
 - normalized**: Need to be normalized? Just set GL_FALSE for vertex positions.
 - stride**: Byte offset between consecutive vertices
 - pointer**: Byte offset to the first component of the attribute ('None' for 0)
- ```
glEnableVertexAttribArray(index):
```

Enable the vertex attribute specified by index

# P5. Configure vertex attributes

- Specifying the vertex attribute index
  - If the vertex position attribute index is 0, then:

```
#version 330 core
layout (location = 0) in vec3 vin_pos;

void main()
{
 gl_Position = vec4(vin_pos.x, vin_pos.y, vin_pos.z, 1.0);
}
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
3*glm::sizeof(glm::float32), None)

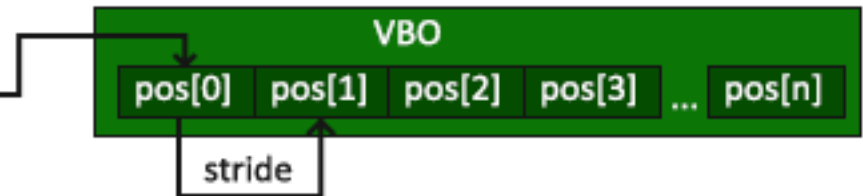
glEnableVertexAttribArray(0)
```

The index number must match between the shader and the OpenGL code.

# VAO & VBO in this example

## VAO

| (attribute index)     | (size) | (type)   | (normalized) | (stride) | (pointer) |
|-----------------------|--------|----------|--------------|----------|-----------|
| 0                     | 3      | GL_FLOAT | GL_FALSE     | 12       | ●         |
| 1                     |        |          |              |          |           |
| ...                   |        |          |              |          |           |
| GL_MAX_VERTEX_ATTRIBS |        |          |              |          |           |



# D1. Activate VAO

- More precisely:  
"Activate the VAO associated with the VBO containing vertex data to draw".
- Simply call the same function:  

```
glBindVertexArray(VAO) # activate VAO
```
- If only one VAO is used, `glBindVertexArray()` only needs to be called once during initialization.

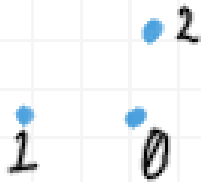
## D2. Call `glDrawArrays()`

- In this example, we use `glDrawArrays()` to draw a triangle.

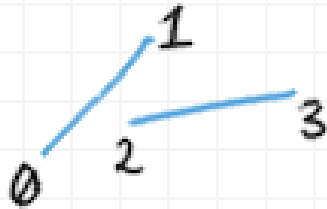
```
glDrawArrays(GL_TRIANGLES, 0, 3)
```

- `glDrawArrays(mode, first, count)`
  - Render primitives from the VBO referenced by the currently bound VAO.
  - `mode`: Primitive type to render.
  - `first`: Starting index in the vertex array
  - `count`: Number of vertices to be rendered

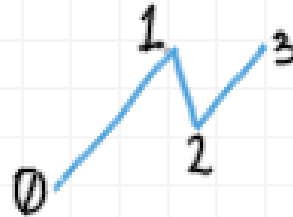
# Primitive Types



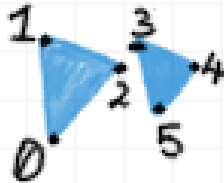
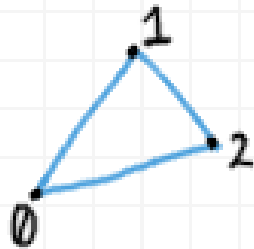
GL\_POINTS



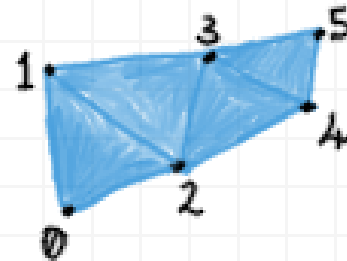
GL\_LINES



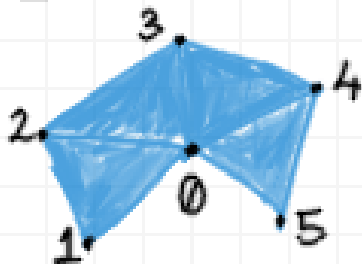
GL\_LINE\_STRIP



GL\_TRIANGLES



GL\_TRIANGLE\_STRIP



GL\_TRIANGLE\_FAN

# Preparing and Using Shader Program Object

- A *shader program object* is the final linked program from shaders.
  - The vertex and fragment shader sources are *dynamically* compiled and then linked into a program object.
- Preparation (at initialization):
  - 1. Create a shader object
  - 2. Provide the shader source code
  - 3. Compile the shader
  - 4. Create a program object
  - 5. Attach the shader objects to the program
  - 6. Link the program
  - You can simply use `load_shaders()` function in "3-hello-triangle.py" for these steps.
- Using (at every rendering frame):
  - Activate the program object using `glUseProgram()`. Every rendering call after this will use this program object.
  - If only one program is used, `glUseProgram()` only needs to be called once during initialization.

```

def load_shaders(vertex_shader_source, fragment_shader_source):
 # vertex shader
 vertex_shader = glCreateShader(GL_VERTEX_SHADER) # create an empty shader object
 glShaderSource(vertex_shader, vertex_shader_source) # provide shader source code
 glCompileShader(vertex_shader) # compile the shader object

 # check for shader compile errors
 success = glGetShaderiv(vertex_shader, GL_COMPILE_STATUS)
 if (not success):
 infoLog = glGetShaderInfoLog(vertex_shader)
 print("ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" + infoLog.decode())

 # fragment shader
 fragment_shader = glCreateShader(GL_FRAGMENT_SHADER) # create an empty shader object
 glShaderSource(fragment_shader, fragment_shader_source) # provide shader source code
 glCompileShader(fragment_shader) # compile the shader object

 # check for shader compile errors
 success = glGetShaderiv(fragment_shader, GL_COMPILE_STATUS)
 if (not success):
 infoLog = glGetShaderInfoLog(fragment_shader)
 print("ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" + infoLog.decode())

 # link shaders
 shader_program = glCreateProgram() # create an empty program object
 glAttachShader(shader_program, vertex_shader) # attach the shader objects to the program object
 glAttachShader(shader_program, fragment_shader)
 glLinkProgram(shader_program) # link the program object

 # check for linking errors
 success = glGetProgramiv(shader_program, GL_LINK_STATUS)
 if (not success):
 infoLog = glGetProgramInfoLog(shader_program)
 print("ERROR::SHADER::PROGRAM::LINKING_FAILED\n" + infoLog.decode())

 glDeleteShader(vertex_shader)
 glDeleteShader(fragment_shader)

 return shader_program # return the shader program

```

# [Code] 3-hello-triangle

```
def main():
 ...
 # load shaders
 shader_program = load_shaders(...)

 # prepare vertex data (in main memory)
 vertices = glm.array(...)

 # create and activate VAO (vertex array object)
 VAO = glGenVertexArrays(1)
 glBindVertexArray(VAO)

 # create and activate VBO (vertex buffer object)
 VBO = glGenBuffers(1)
 glBindBuffer(GL_ARRAY_BUFFER, VBO)

 # upload vertex data to VBO
 glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices.ptr, GL_STATIC_DRAW)

 # configure vertex attributes
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * glm.sizeof(glm.float32), None)
 glEnableVertexAttribArray(0)

 while not glfwWindowShouldClose(window):
 # render
 glClear(GL_COLOR_BUFFER_BIT)

 glUseProgram(shader_program)
 glBindVertexArray(VAO)
 glDrawArrays(GL_TRIANGLES, 0, 3)

 # swap front and back buffers & poll events
 glfwSwapBuffers(window)
 glfwPollEvents()

 ...
```

\* The full source code can be found at <https://github.com/yssl/CSE4020>

# Quiz 3

---

- Go to <https://www.slido.com/>
- Join #cg-ys
- Click "Polls"
- Submit your answer in the following format:
  - **Student ID: Your answer**
  - e.g. **2024123456: 4.0**
- Note that your quiz answer must be submitted **in the above format** to be counted for attendance.

# Time for Lab Check

---

- Let's start today's lab check.
- TAs will guide you.