Computer Graphics

2 - Lab - Hello Triangle!

Yoonsang Lee
Hanyang University

Spring 2023

Outline

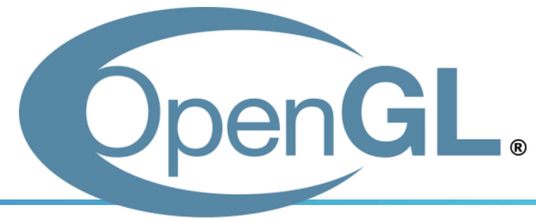
- Introduction to OpenGL and other modules
  - What is OpenGL?
  - OpenGL Versions
  - GLFW input handling
  - PyGLM, NumPy
- Shaders
  - OpenGL Rendering Pipeline
  - Vertex Shader
  - Fragment Shader
  - GLSL
- Hello Triangle!
  - Preparing Vertex Data and Drawing Using It
  - Preparing and Using Shader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

Note for CSE4020 Code Repository

- https://github.com/yssl/CSE4020
- You can download the source code used in labs, and freely modify, run, test, and play!

Introduction to OpenGL and other modules

What is OpenGL?



- **OpenGL: Open G**raphics **L**ibrary
  - Unlike its name, OpenGL is not a library.

- OpenGL is the **specification** that describes the behavior of a **rasterization-based** rendering system **for real-time 3D graphics**.
  - OpenGL is an "open" specification. Anyone can download the specfication for free.

- OpenGL describes the **interface** the programmer uses and **expected behavior**.
  - It defines the **API** (Application Programming Interface) through which a client application can control this system.

- The OpenGL rendering system is carefully specified to make hardware implementations allowable.

Who Develops / Maintains OpenGL?

- OpenGL specification:
  - Maintained by the *Khronos Group* committee called the *OpenGL Architectural Review Board (ARB)*.

- OpenGL implementation:
  - Hardware vendors (such as Nvidia) implement OpenGL to work on their GPUs.
  - Their implementations, called "drivers", translate OpenGL API commands into their GPU commands.

Characteristics of OpenGL

- Cross platform
  - OpenGL is designed to be platform-independent.
  - You can use OpenGL on Windows, Linux, MacOS, Android, and iOS.
    - Although Apple has officially deprecated support for OpenGL in favor of its own graphics API, Metal, many existing applications and games that use OpenGL still run on MacOS and iOS.

- Language independent
  - OpenGL has many language bindings (C, Python, Java, Javascript, ...)
  - We'll use Python binding for OpenGL, PyOpenGL

OpenGL Versions

- Legacy OpenGL (OpenGL 1.x~)
  - Invented when "fixed-function" hardware was standard
  - No shaders (before 2.0)
  - Easier to learn & good for rapid prototyping
  - Deprecated since OpenGL 3.0

- Modern OpenGL (OpenGL 3.x~)
  - Now programmable hardware is the common industry practice
  - Use of programmable shaders
  - More difficult to program 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 assignments and projects must use this version.

- Ensure your laptop supports **OpenGL 3.3 or higher.**
  - For how to check your OpenGL version, see "Checking OpenGL Version" in the first lecture slide, "1 - Course Intro.pdf".

OpenGL Core Profile

- In OpenGL Core Profile, using the following deprecated features will result in an error:

- glBegin(), glEnd() and functions that can be located between them:
  - glBegin(), glEnd(), glVertex*, glNormal*, …

- Functions for color, light, material properties
  - glColor*, glLight*, glMaterial*, …

- All matrix operations:
  - glRotate*, glTranslate*, glScale*, glMatrixMode(), glLoadIdentity(), glPushMatrix(), glPopMatrix(), glFrustum(), gluPerspective(…), gluLookAt(..)

- In core profile, OpenGL forces us to use modern practices with *shaders*.
  - You cannot use the old *immediate mode* or *fixed function pipeline* listed above.

So, what can we do with OpenGL?

- **Just only drawing things**
  - Provides small, but powerful set of low-level drawing operations.
  - No functions for creating windows, handling events, even for OpenGL contexts.
    - OpenGL context stores all the necessary information and state required to use the OpenGL API. It consists of resources - driver resources in RAM, texture IDs assigned, VBO IDs assigned, enabled states (GL_BLEND, GL_DEPTH_TEST) and many other things.

- So, we need additional utility libraries to use OpenGL
  - GLFW, FreeGLUT : Simple utility libraries for OpenGL
  - Fltk, wxWigets, Qt, Gtk : General purpose GUI framework

Utility Libraries for Learning OpenGL

- General GUI frameworks(e.g. Qt) are powerful, but too heavy for just learning OpenGL.

- GLUT “was” most popular for this purpose.
  - But it’s outdated and unmaintained.
  - Its open-source clone FreeGLUT is mostly concerned with providing a stable clone of GLUT.

- Now, GLFW is getting more popular.
  - Provides much fine control for managing windows and events.
  - We'll use Python binding for GLFW, glfw.

[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 that have already been received.
  - Calls a user-registered callback function for each type of events.

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

[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 key callback for escape key
    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()
```


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

Documentation for glfw

- If you import glfw like '`from glfw.GLFW import *`', you can use the same function and constant names as in the GLFW C API.

- Refer to the following site which provides GLFW C API documentation:
  – http://www.glfw.org/documentation.html

- Note that functions like glfwGetMonitors() **return a list instead of a pointer and an object count.**

PyGLM

- OpenGL Mathematics (GLM) is a C++ mathematics library based on the OpenGL Shading Language (GLSL) specification.
  - GLM emulates GLSL's approach to vector/matrix operations whenever possible.

- PyGLM is a Python extension based on GLM.

- PyGLM 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. 2021123456: 4.0**

- Note that your quiz answer must be submitted **in the above format** to receive a quiz score!

NumPy

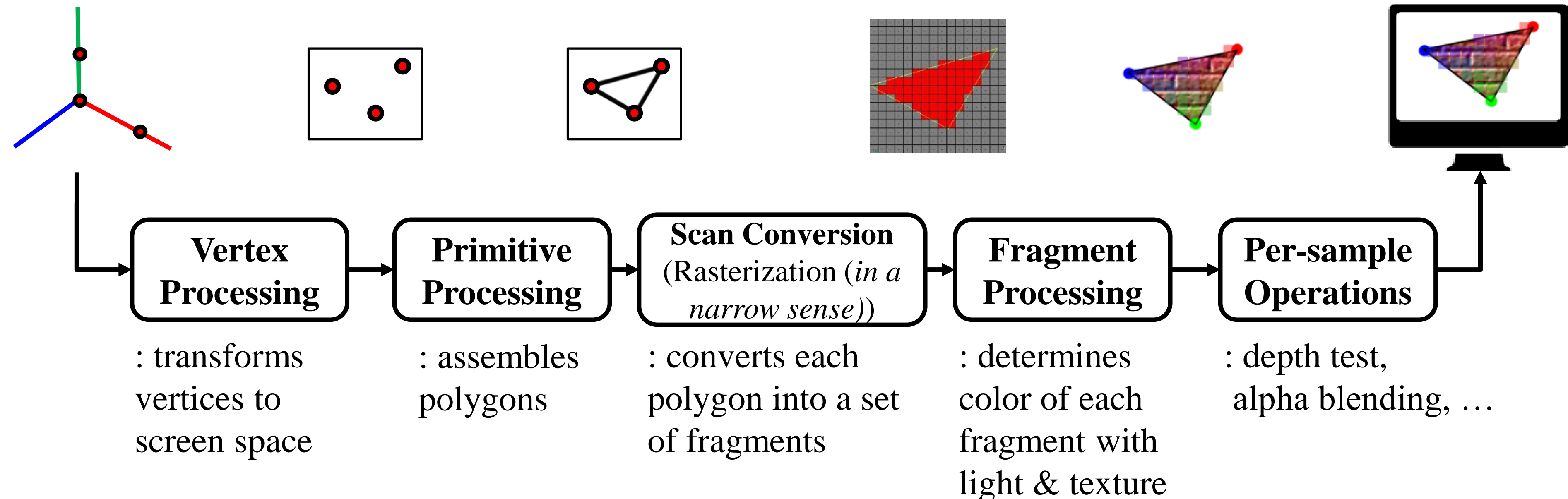
- NumPy is a general-purpose numerical computing library for Python.
  – Provides high-performance matrix & vector operations
  – Provide linear algebra functions

- NumPy & PyGLM
  – PyGLM is specifically designed to work well with PyOpenGL, with comprehensive set of functions for this purpose.
  – NumPy is for more general purposes, such as general numerical computations, including linear algebra.

- We'll probably be using PyGLM primarily, but we'll also be using NumPy in some cases, including some early labs.

Shaders

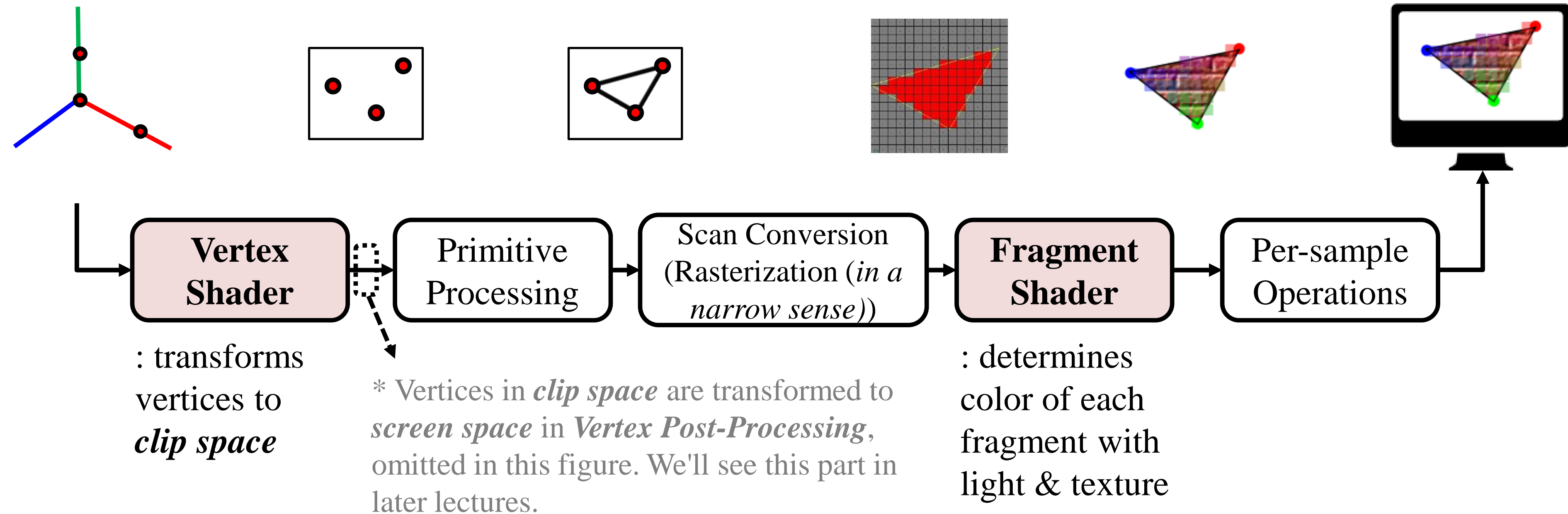
Recall: Rasterization Pipeline



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

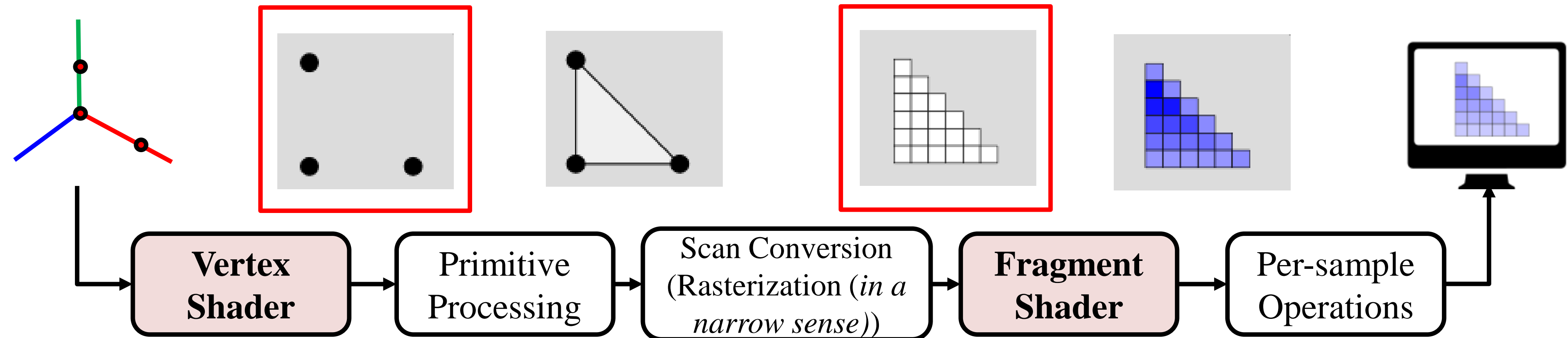
* The image is from https://graphicscompendium.com/intro/01-graphics-pipeline

OpenGL Rendering Pipeline (simplified version)



- *Shader*: a small program that runs on GPU.
- The red blocks are the only *programmable* steps.
    - Actually there are more optional programmable shaders in the OpenGL rendering pipeline but they are not covered in this course.
- Modern OpenGL requires at least a *vertex shader* and a *fragment shader* to do some rendering.

OpenGL Shaders



- Vertex shader is executed for each vertex **in parallel**.
- Fragment shader is executed for each fragment **in parallel**.

- In the example figured above,
  - The vertex shader is executed 3 times…
  - The fragment shader is executed 21 times…
  - in parallel. (actually at the same time – note that most GPUs today have thousands of small cores)

GLSL – OpenGL Shading Language

- OpenGL shaders are written in GLSL.

- GLSL is a C-style language, so it covers most of the features you would expect with C language.

- We use GLSL 3.30 core profile by specifying the following directive in the beginning of the shader code:

```
#version 330 core
```

Vertex Shader

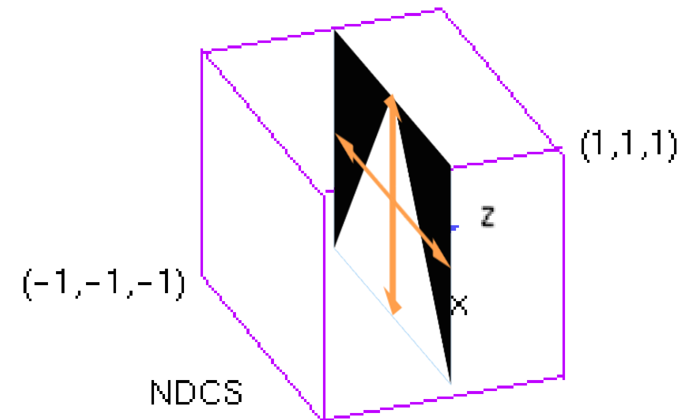
- Input: Per-vertex attributes (from CPU)
  - **Position**
  - Possibly any other per-vertex attribute such as color, normal, texture coordinates, …

- Output:
  - **Vertex position *in clip space*** (to vertex post-processing)
  - Possibly any other per-vertex output such as vertex color, vertex normal, … (to fragment shader)

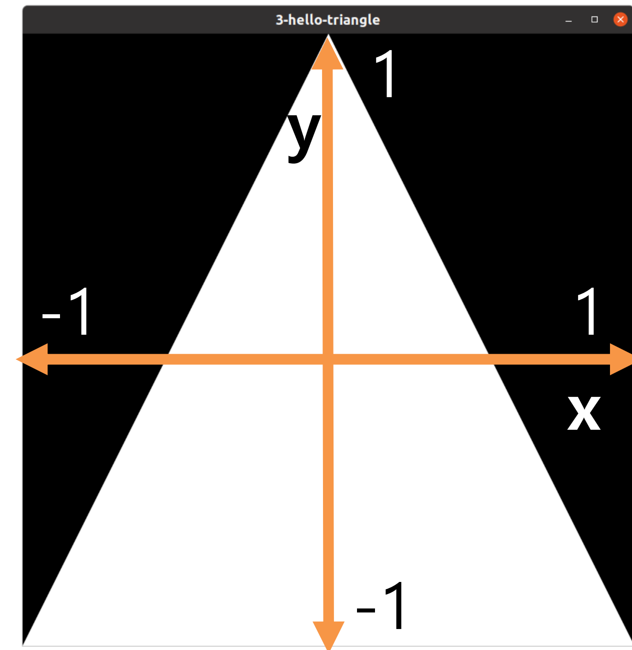
- (We will look at the gray part later.)

Clip Space

- By default (with no transformation), you can draw an object anywhere in cube space with x, y, z coordinates ranging from -1 to 1.



- This space is called *clip space* (or *NDC space*).
  - Its **xy** plane is a 2D "viewport".
  - Its coordinate system is *normalized device coordinate (NDC)*.



- We will take a closer look at their meaning in later lectures.

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 position input & output in this example.
- vec4 (x, y, z, w): w component is used for "perspective division" in perspective projection which is covered in later lectures.
- At this moment, you can think w is just 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: Only one 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(1.0f, 1.0f, 1.0f, 1.0f);
}
```

- Only fragment color output in this example. 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 (*)
  - `vec * vec` – component-wise multiplication
  - `scalar*vec, scalar*mat` – component-wise multiplication
  - **`mat * mat` – matrix-matrix multiplication**
  - **`mat * vec` – matrix-vector multiplication**

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"

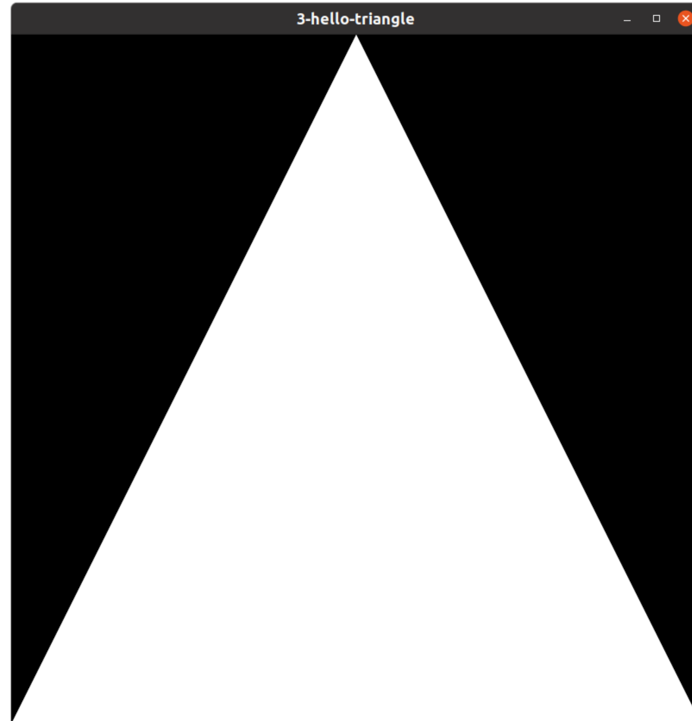
- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2021123456: 4.0**

- Note that your quiz answer must be submitted **in the above format** to receive a quiz score!

Hello Triangle!

Goal

- Draw the following white triangle:



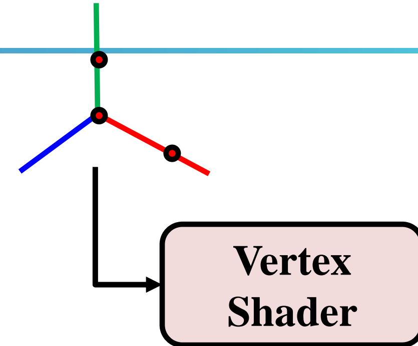
- You can download and run "2-lab/3-hello-triangle.py" from https://github.com/yssl/CSE4020.

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. Copy vertex data to VBO
  - 5. Configure vertex attributes

- Drawing (at every rendering frame):
  - 1. Activate VAO
  - 2. Call glDraw*()

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
    )
```

- This example …
  - Specifies 3 vertices with their x, y, z coordinates.
  - All vertices are in xy plane (z=0).
  - Uses `glm`, but you can use `numpy` if you want.

P2. Create and activate VAO

- Vertex Array Object (VAO):
  - OpenGL object that stores the **"state" for working with vertex data.**

- A VAO stores:
  - Vertex attribute configuration via `glVertexAttribPointer()`
  - Pointers to VBOs (vertex buffer objects) associated with vertex attributes by `glVertexAttribPointer()`
  - Whether a vertex attribute is enabled or not by `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 ID and store it to VAO variable
  glBindVertexArray(VAO) # activate VAO
  ```

- `glGenVertexArrays(n)`
  - Generate `n` vertex array object IDs and return them

- `glBindVertexArray(vao)`
  - Make `vao` vertex array object active (and other VAOs are not activated)
  - That means, any subsequent vertex attribute calls from that point on will be stored inside `vao` vertex array object.

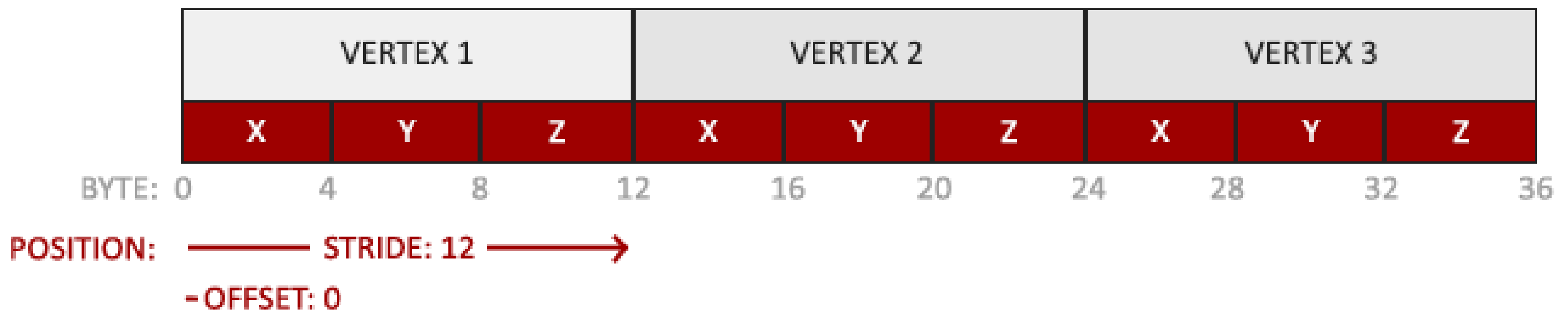
- Meaning of "**Bind**" in OpenGL
  - "Bind" basically means associating two things.
  - `glBind*()` functions associate specified objects to current OpenGL context.
  - → **Make the specified object active** or **Activate the specified object**
  ```

P3. Create and activate VBO

- Vertex Buffer Object (VBO):
  - A type of buffer that is used to store vertex data, such as vertex position, color, normal, and etc., for rendering.

- Why use VBO?
  - OpenGL cannot directly use vertex data in main memory.
    - We need to allocate some memory that OpenGL can see and copy our vertex data to it. This can be done with VBO.
  - Faster rendering time.
    - VBOs are usually allocated on extremely fast graphics card's memory, so vertex shaders can have instance access to vertex data.
    - By storing vertex data in a VBO, you can reduce the amount of data that needs to be transferred between the CPU and the GPU.

P3. Create and activate VBO

-
```
VBO = glGenBuffers(1)    # create a buffer object ID and store it to VBO variable
glBindBuffer(GL_ARRAY_BUFFER, VBO)  # activate VBO as a vertex buffer object
```

- `glGenBuffers(n)`
  - Generate `n` buffer object IDs and return them

- `glBindBuffer(target, buffer)`
  - Activate `buffer` object as `target` type buffer
  - `target`: Use `GL_ARRAY_BUFFER` for vertex buffer object

P4. Copy vertex data to VBO

- ```
  glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, 
  vertices.ptr, GL_STATIC_DRAW)
  ```
- `glBufferData(target, size, data, usage)`
  - Allocate GPU memory for and copy vertex data to the currently bound VBO.
  - `target`: Use `GL_ARRAY_BUFFER` for vertex buffer object
  - `size`: Size of the data (in bytes) that will be copied
  - `data`: A pointer to data that will be copied
  - `usage`:
    - GL_STATIC_DRAW: the data will most likely not change at all or very rarely.
    - GL_DYNAMIC_DRAW: the data is likely to change a lot.
    - GL_STREAM_DRAW: the data will change every time it is drawn.
  - Note that `usage` is just a hint which enables GPU driver to make more intelligent decision. But it does not constrain the actual usage of the buffer.
  ```

P5. Configure vertex attributes

- OpenGL does not yet know
  - how it should interpret the vertex data in VBOs
  - how it should connect the vertex data to the vertex shader's attributes

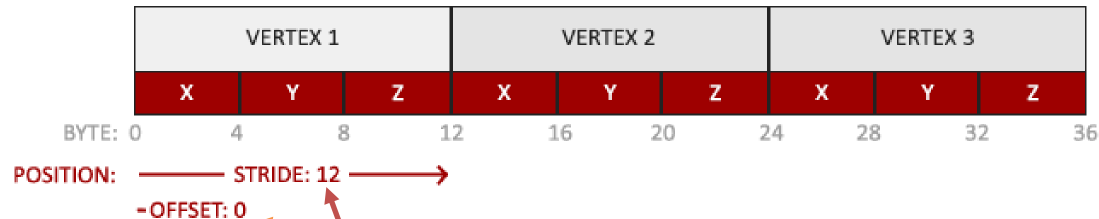
- In our example, `vertices` data (in `VBO`) is formatted as follows:

```
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
    )
```



* The image is from https://learnopengl.com/Getting-started/Hello-Triangle

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 connect the vertex data to the vertex shader's attributes.
  - `index`: Vertex attribute index to configure
  - `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 vertex attributes
  - **`pointer`**: Byte offset of the beginning component ('None' for 0)

- `glEnableVertexAttribArray(index)`: Enable the vertex attribute specified by `index`

P5. Configure vertex attributes

- Specifying vertex attribute index
  - If the attribute index of vertex positions in currently bound VBO is 0, then

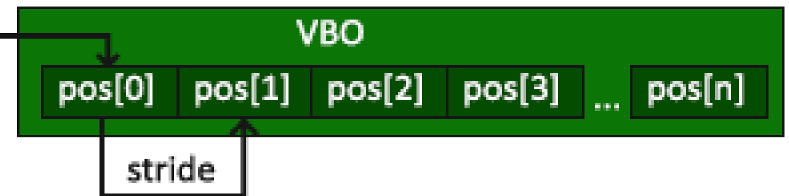
```
#version 330 core
layout (location = 0) in vec3 vin_pos;

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)
```

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 VAO associated with VBO containing vertex data to draw".

- Just call the same function:

```
glBindVertexArray(VAO) # activate VAO
```

- If only one VAO is used, `glBindVertexArray()` can be called only once in the initialization stage.

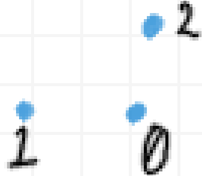
D2. Call glDraw*()

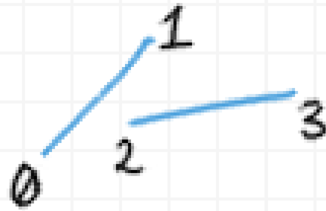
- In our example, we use `glDrawArrays()` to draw a triangle.

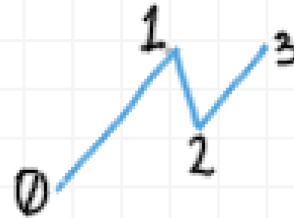
```
glDrawArrays(GL_TRIANGLES, 0, 3)
```

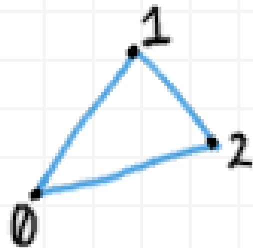
- `glDrawArrays(mode, first, count)`
  - Render primitives from vertex array in currently bounded VAO.
  - `mode`: Primitive type to render.
  - `first`: Starting index in the vertex array
  - `count`: Number of vertices to be rendered

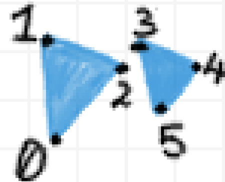
Primitive Types

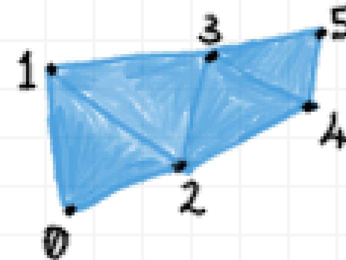


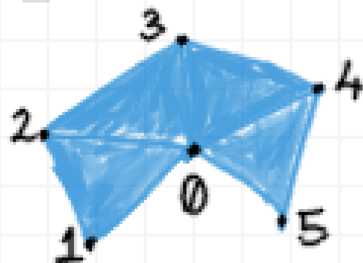












* The image is from https://antongerdelan.net/opengl/vertexbuffers.html

Preparing and Using Shader Program Object

- A *shader program object* is the final linked version of shaders.
  - The vertex and fragment shader source are *dynamically* compiled and linked to the program object.

- Preparation (at initialization):
  - 1. Create an empty shader object
  - 2. Provide shader source code
  - 3. Compile the shader object
  - 4. Create an empty program object
  - 5. Attach the shader objects to the program object
  - 6. Link the program object
  - You can just use `load_shaders()` function in "3-hello-triangle.py" for these steps.

- Using (at every rendering frame):
  - Activate the program object by `glUseProgram()`. Every rendering call after this will use this program object.
  - If you use only one program, `glUseProgram()` can be called only once in the initialization stage.

```
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 programam
```

[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)

    # copy 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. 2021123456: 4.0**

- Note that your quiz answer must be submitted **in the above format** to receive a quiz score!

Time for Assignment

- Let's start today's assignment.
- TA will guide you.