Computer Graphics

2 - Introduction to NumPy / OpenGL

Yoonsang Lee
Spring 2020

Summary of Course Intro

- Questions
  - https://www.slido.com/ - Join #cg-hyu

- Quiz & Attendance
  - https://www.slido.com/ - Join #cg-hyu - Polls
  - You must submit all quiz answers in the correct format to be checked for “attendance”.

- Language
  - I’ll “paraphrase” the explanation in Korean for most slides.

- **You MUST read 1-CourseIntro.pdf CAREFULLY.**

Notification for Quiz & Attendance

- If you cannot answer during the given quiz time (2 mins) due to the streaming problem, you can submit the quiz answer until 23:59 on the day of the lecture (I'll leave the quiz polls unclosed).

- So in this case, submit the quiz answer while listening to the recorded lecture to be checked for attendance.

- This policy is maintained until the streaming service is stabilized.

Topics Covered

- Why Python in Computer Graphics?
- Introduction to NumPy
  - What is NumPy?
  - How to use NumPy
  - Handling vectors & matrices using NumPy
- Introduction to OpenGL
  - What is OpenGL?
  - OpenGL basics
  - GLFW input handling
  - Legacy OpenGL & Modern OpenGL
  - OpenGL as a Learning Tool

Why Python in Computer Graphics?

- Productivity
  - Easy to learn and use for beginners.
  - You can write code much faster.
  - You can focus on "logic", not language-specific issues.

- Powerful modules
  - A wide range of reliable modules are available.
  - E.g.) NumPy & SciPy for scientific computing, matplotlib for data visualization, …

- **Python allows you to implement key computer graphics concepts in a short time.**

Why Python in Computer Graphics?

- Popular language in research & scientific community of other areas.
  - Most ML / DL framework provides Python API.
    - TensorFlow, PyTorch, Keras, Theano, ...
  - Most popular language in Data Science.

- **Python allows you to easily combine computer graphics and other techniques including machine learning or data science.**

Python Interpreter

- Python **3.7** or later
  - https://www.python.org/downloads/

- Note that all submissions for assignments should work in Python **3.7**.

- You can use any OS that runs Python.

Introduction to NumPy

What is NumPy?

- NumPy is a Python module for scientific computing.
  - Written in C
  - Fast vector & matrix operations

- NumPy is **de-facto standard** for numerical computing in Python.

- Very useful for computer graphics applications, which are made of vectors & matrices.

NumPy usage

- You’ve already installed NumPy in the last lab session.
  - If you haven’t, see 1-Lab-EnvSetting.pdf slides and install it.

- Now, let's launch python3 interpreter in the interactive mode and import numpy like this:

```
>>> import numpy as np
```

: use ‘np’ as the local name for the module numpy

- The following NumPy slides come from:
  - https://github.com/enthought/Numpy-Tutorial-SciPyConf-2017/blob/master/slides.pdf

Introducing NumPy Arrays

ENTHOUGHT

SIMPLE ARRAY CREATION

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

CHECKING THE TYPE

```
>>> type(a)
numpy.ndarray
```

NUMERIC "TYPE" OF ELEMENTS

```
>>> a.dtype
dtype('int32')
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
1
```

Array Operations

ENTHOUGHT

SIMPLE ARRAY MATH

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([2, 3, 4, 5])
>>> a + b
array([3, 5, 7, 9])

>>> a * b
array([ 2, 6, 12, 20])

>>> a ** b
array([  1,   8,   81,   1024])
```

NumPy defines these constants:
pi = 3.14159265359
e = 2.71828182846

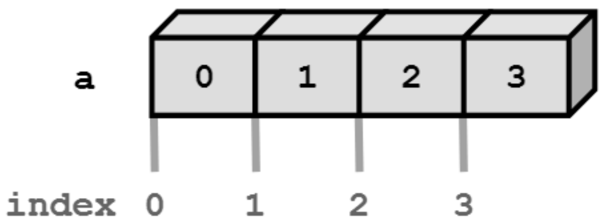
```
# multiply entire array by
# scalar value
>>> 0.1 * a
array([0.1, 0.2, 0.3, 0.4])

# in-place operations
>>> a *= 2
>>> a
array([2, 4, 6, 8])

# apply functions to array
>>> x = 0.1*a
>>> x
array([0.2, 0.4, 0.6, 0.8])
>>> y = np.sin(x)
>>> y
array([0.19866933, 0.38941834,
0.56464247, 0.71735609])
```

Setting Array Elements

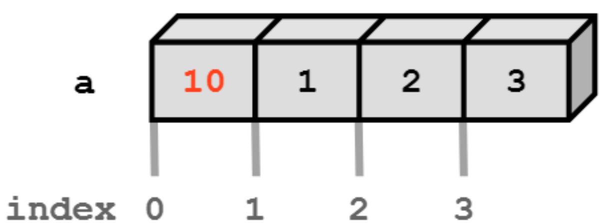
ENTHOUGHT

ARRAY INDEXING

```
>>> a[0]
0
```




```
>>> a[0] = 10
>>> a
array([10, 1, 2, 3])
```



BEWARE OF TYPE COERCION

```
>>> a.dtype
dtype('int32')

# assigning a float into
# an int32 array truncates
# the decimal part
>>> a[0] = 10.6
>>> a
array([10, 1, 2, 3])
```

Numpy array: All elements have the same type and the size.

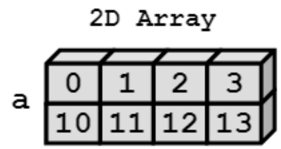
Python list: Elements can have various sizes and types.

Multi-Dimensional Arrays

ENTHOUGHT

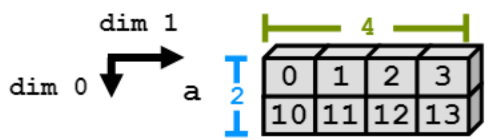
MULTI-DIMENSIONAL ARRAYS

```
>>> a = np.array([[ 0, 1, 2, 3],
...                [10,11,12,13]])
>>> a
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
```



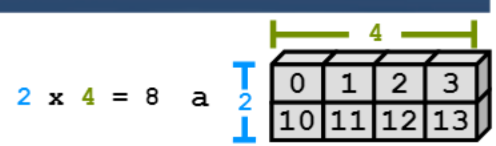
SHAPE = (ROWS, COLUMNS)

```
>>> a.shape
(2, 4)
```



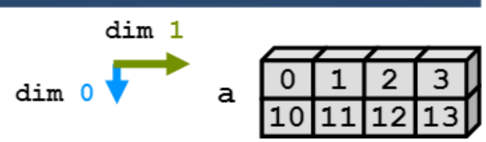
ELEMENT COUNT

```
>>> a.size
8
```



NUMBER OF DIMENSIONS

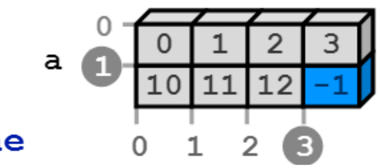
```
>>> a.ndim
2
```



GET / SET ELEMENTS

```
>>> a[1, 3]
13
```


```
>>> a[1, 3] = -1
>>> a
array([[ 0,  1,  2,  3],
       [10, 11, 12, -1]])
```




```
# Shape returns a tuple
# listing the length of the
# array along each dimension.
```

Slicing

ENTHOUGHT

```
var[lower:upper:step]
```

Extracts a portion of a sequence by specifying a lower and upper bound.
The lower-bound element is included, but the upper-bound element is **not** included.
Mathematically: [lower, upper). The step value specifies the stride between elements.

SLICING ARRAYS

```
#                      -5 -4 -3 -2 -1
# indices:              0  1  2  3  4
>>> a = np.array([10,11,12,13,14])

# [10, 11, 12, 13, 14]
>>> a[1:3]
array([11, 12])

# negative indices work also
>>> a[1:-2]
array([11, 12])
>>> a[-4:3]
array([11, 12])
```

OMITTING INDICIES

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list

# grab first three elements
>>> a[:3]
array([10, 11, 12])

# grab last two elements
>>> a[-2:]
array([13, 14])

# every other element
>>> a[::2]
array([10, 12, 14])
```

Array Slicing

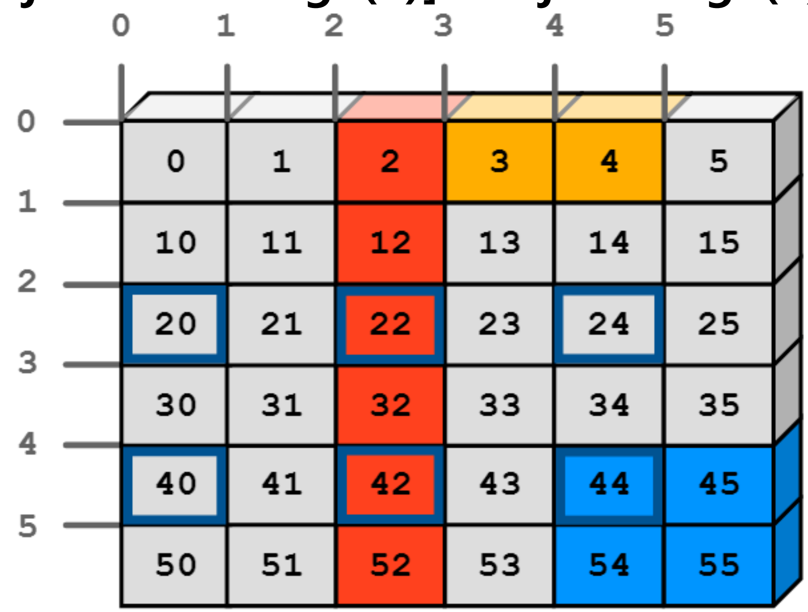
SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 45],
       [54, 55]])

>>> a[:, 2]
array([2, 12, 22, 32, 42, 52])
```

STRIDED ARE ALSO POSSIBLE

```
>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

a = np.array([[i+10*j for i in range(6)] for j in range(6)])



Array Constructor Examples

ENTHOUGHT

FLOATING POINT ARRAYS

```
# Default to double precision
>>> a = np.array([0,1.0,2,3])
>>> a.dtype
dtype('float64')
>>> a.nbytes
32
```

REDUCING PRECISION

```
>>> a = np.array([0,1.,2,3],
...              dtype='float32')
>>> a.dtype
dtype('float32')
>>> a.nbytes
16
```

Array Creation Functions

ENTHOUGHT

IDENTITY

$n \times n$ square matrix with ones on the main diagonal and zeros elsewhere.

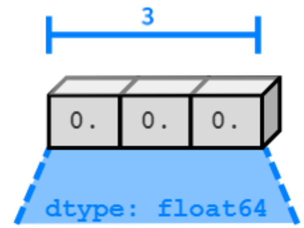
```
# Generate an n by n identity
# array. The default dtype is
# float64.
>>> a = np.identity(4)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> a.dtype
dtype('float64')
>>> np.identity(4, dtype=int)
array([[ 1, 0, 0, 0],
       [ 0, 1, 0, 0],
       [ 0, 0, 1, 0],
       [ 0, 0, 0, 1]])
```

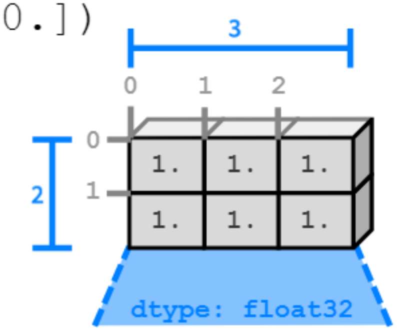
ONES, ZEROS

```
ones(shape, dtype='float64')
zeros(shape, dtype='float64')
```

shape is a number or sequence specifying the dimensions of the array. If **dtype** is not specified, it defaults to **float64.**

```
>>> np.ones((2, 3),
...         dtype='float32')
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]],
      dtype=float32)
>>> np.zeros(3)
array([ 0.,  0.,  0.])
```





zeros(3) is equivalent to **zeros((3,))**

Array Creation Functions (cont'd)

ENTHOUGHT

LINSPACE

```
# Generate N evenly spaced
# elements between (and including)
# start and stop values.
>>> np.linspace(0, 1 ,5)
array([0., 0.25., 0.5, 0.75, 1.0])
```

ARANGE

```
arange([start,] stop[, step],
                dtype=None)
```

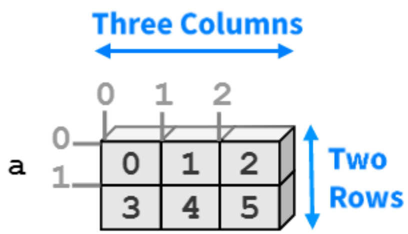
- Nearly identical to Python’s range()
- Creates an array of the interval including *start* but **excluding *stop***
- When using a **non-integer step**, the results will **often not be consistent due to finite machine precision**. It is **better to use linspace() for this case**.

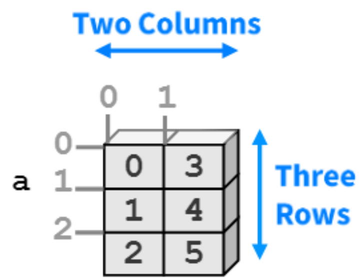
```
>>> np.arange(4)
array([0, 1, 2, 3])

>>> np.arange(1.5, 2.1, 0.3)
array([ 1.5, 1.8, 2.1])
```

Transpose

TRANSPOSE

```
>>> a = np.array([[0,1,2],
...               [3,4,5]])
>>> a.shape
(2,3)

# Transpose swaps the order
# of axes.

>>> a.T
array([[0, 3],
       [1, 4],
       [2, 5]])
>>> a.T.shape
(3,2)
```





Reshaping Arrays

RESHAPE

```
>>> a = np.array([[0,1,2],
...               [3,4,5]])

# Return a new array with a
# different shape (a view
# where possible)
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])

# Reshape cannot change the
# number of elements in an
# array
>>> a.reshape(4,2)
ValueError: total size of new
array must be unchanged
```

Quiz #1

- Go to https://www.slido.com/
- Join #cg-hyu
- Click “Polls”

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked for “attendance”.

Vector & Matrix with NumPy

- Vectors are just 1D arrays:

```
>>> v = np.arange(3)
>>> v
array([0, 1, 2])
```

- Matrices are just 2D arrays:

```
>>> M = np.arange(9).reshape(3,3)
>>> M
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Matrix & Vector Multiplication

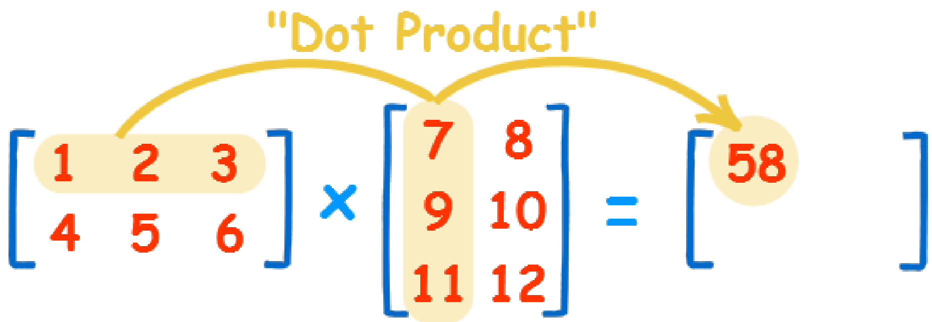
- * is an element-wise multiplication operator.

```
>>> v * v
array([0, 1, 4])
>>> M * M
array([[ 0,  1,  4],
       [ 9, 16, 25],
       [36, 49, 64]])
```

- Not so much used in computer graphics.

Matrix & Vector Multiplication

- Matrix multiplication requires "dot product" (inner product in Euclidian space)



The "Dot Product" is where we **multiply matching members**, then sum up:

$$(1, 2, 3) \bullet (7, 9, 11) = 1\times7 + 2\times9 + 3\times11$$
$$= 58$$

https://www.mathsisfun.com/algebra/matrix-multiplying.html

Matrix & Vector Multiplication

- **@** is a **matrix multiplication operator**.

```
>>> v @ v
5
>>> M @ M
array([[ 15,  18,  21],
       [ 42,  54,  66],
       [ 69,  90, 111]])
>>> M @ v
array([ 5, 14, 23])
```

- Very often used in computer graphics!

Quiz #2

- Go to https://www.slido.com/
- Join #cg-hyu
- Click “Polls”

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked for “attendance”.

Introduction to OpenGL

What is OpenGL?

- **Op**en **G**raphics **L**ibrary

- OpenGL is an **API** (Application Programming Interface) for graphics programming.
  - Unlike its name, OpenGL is not a library.

What is OpenGL?

- **API** is a **specification.**
  - API describes **interfaces** and **expected behavior.**

- As for OpenGL API,
  - OS vendors provide OpenGL interface (e.g. opengl32.dll on Windows)
  - GPU vendors provide OpenGL implementation, the graphics card driver (e.g. Nvidia drivers)

Characteristics of OpenGL

- Cross platform
  - You can use OpenGL on Windows, OS X, Linux, iOS, Android, ...

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

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 & OpenGL contexts, handling events (we'll discuss the "context" later)

- Thus, additional utility libraries are required 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.
  - So GLFW is our choice for this class.

[Practice] First OpenGL Program

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

def render():
    pass

def main():
    # Initialize the library
    if not glfw.init():
        return
    # Create a windowed mode window and its OpenGL context
    window = glfw.create_window(640,480,"Hello World", None,None)
    if not window:
        glfw.terminate()
        return

    # Make the window's context current
    glfw.make_context_current(window)

    # Loop until the user closes the window
    while not glfw.window_should_close(window):
        # Poll events
        glfw.poll_events()

        # Render here, e.g. using pyOpenGL
        render()

        # Swap front and back buffers
        glfw.swap_buffers(window)

    glfw.terminate()

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

import X
: access X's attribute or method using X.attribute, X.method()

from X import *
: access X's attribute or method just using attribute, method()

If the python interpreter is **running this source file as the main program**, it sets the special **__name__** variable to have a value **"__main__"**.

If this file **is being imported from another module**, **__name__** will be set to **the module's name**.

[Practice] Draw a Triangle

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glVertex2f(0.0, 1.0)
    glVertex2f(-1.0,-1.0)
    glVertex2f(1.0,-1.0)
    glEnd()
```

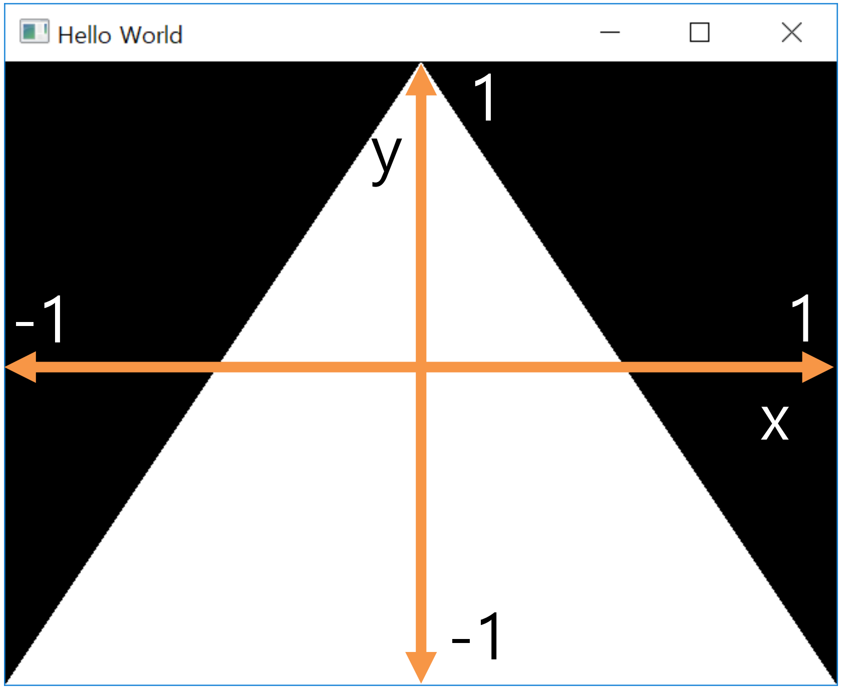
Vertex

- In OpenGL, geometry is specified by **vertices.**
- To draw something, vertices have to be listed between *glBegin(primitive_type)* and *glEnd()* calls.
- *glVertex*()* specifies the coordinate values of a vertex.

```
glBegin(GL_TRIANGLES)
glVertex2f(0.0, 1.0)
glVertex2f(-1.0,-1.0)
glVertex2f(1.0,-1.0)
glEnd()
```

Coordinate System

- You can draw the triangle anywhere in a 2D square ranging from (-1, -1) to (1, 1).
- Called “Normalized Device Coordinate” (NDC).
- We’ll see how objects are transformed to NDC in later classes.

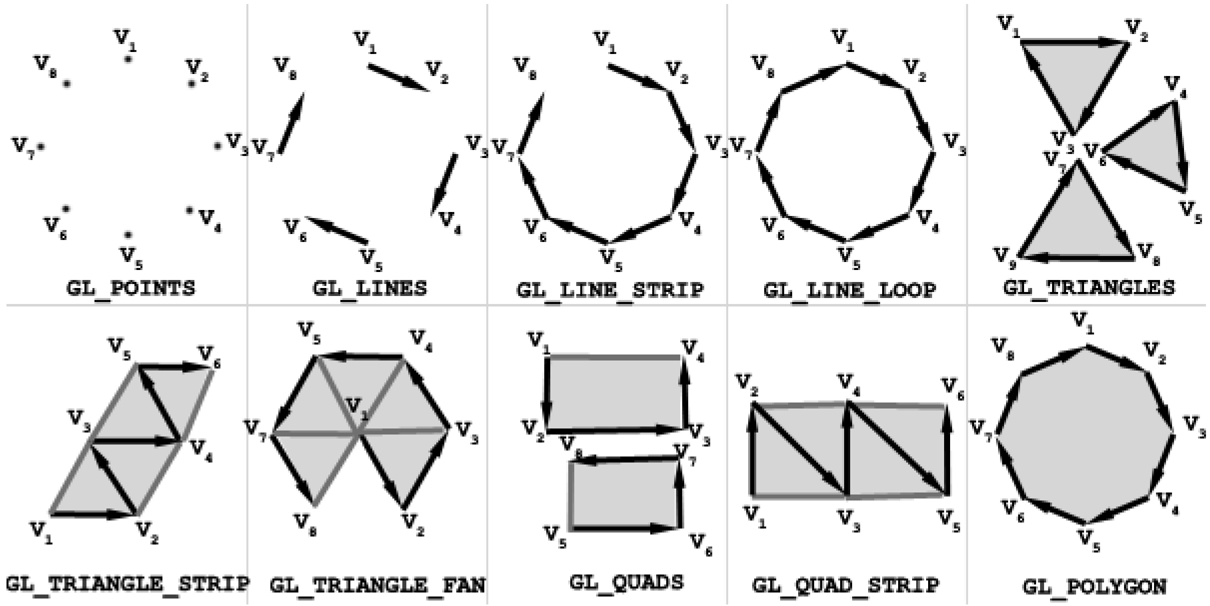


[Practice] Resize the Triangle

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glVertex2f(0.0, 0.5)
    glVertex2f(-0.5,-0.5)
    glVertex2f(0.5,-0.5)
    glEnd()
```

Primitive Types

- Primitive types in *glBegin(**primitive_type**)* :



- They represents how vertices are to be connected.

[Practice] Change the Primitive Type

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_POINTS)
    # glBegin(GL_LINES)
    # glBegin(GL_LINE_STRIP)
    # glBegin(GL_LINE_LOOP)
    # ...
    glVertex2f(0.0, 0.5)
    glVertex2f(-0.5,-0.5)
    glVertex2f(0.5,-0.5)
    glEnd()
```

Vertex Attributes

- In OpenGL, a vertex has these attributes:
  - **Vertex coordinate** : specified by glVertex*()
  - **Vertex color** : specified by glColor*()
  - **Normal vector** : specified by glNormal*()
  - **Texture coordinate** : specified by glTexCoord*()

- We'll see normal vector and texture coord. attributes in later classes.

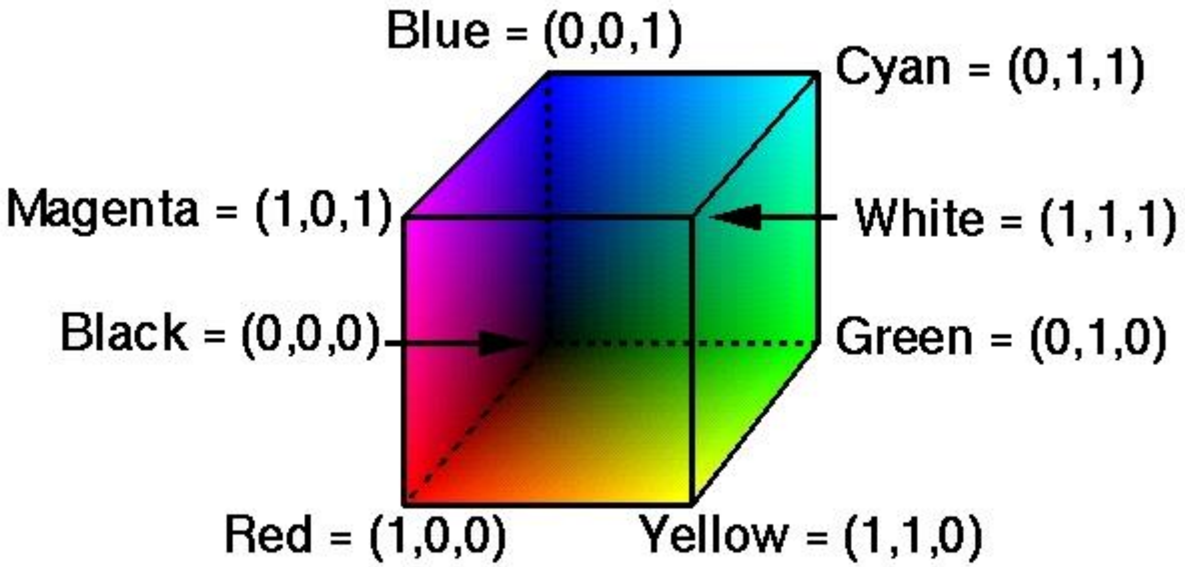
- Now, let's have a look at the **vertex color**.

[Practice] Colored Triangle

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glColor3f(1.0, 0.0, 0.0)
    glVertex2f(0.0, 1.0)
    glColor3f(0.0, 1.0, 0.0)
    glVertex2f(-1.0,-1.0)
    glColor3f(0.0, 0.0, 1.0)
    glVertex2f(1.0,-1.0)
    glEnd()
```

Color

- OpenGL uses the RGB color model.



- Colors in interior are interpolated.

Then, how to draw a just "red" triangle?

- Set red color for each vertex?

- You can do it just by:

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glColor3f(1.0, 0.0, 0.0)
    glVertex2f(0.0, 1.0)
    glVertex2f(-1.0,-1.0)
    glVertex2f(1.0,-1.0)
    glEnd()
```

OpenGL is a State Machine

- If you set a value for a state (or mode), **it remains in effect until you change it.**
  - E.g. “current” color
  - Others states:
    - “current” viewing and projection transformations
    - “current” polygon drawing modes
    - “current” positions and characteristics of lights
    - “current” material properties of the objects
    - ...
- **OpenGL context** stores all of the state associated with this instance of OpenGL.

OpenGL Functions

glVertex3fv(*v*)

Number of components

```
2 - (x,y)
3 - (x,y,z)
4 - (x,y,z,w)
```

Data Type

```
b  - byte
ub - unsigned byte
s  - short
us - unsigned short
i  - int
ui - unsigned int
f  - float
d  - double
```

Vector

omit "v" for scalar form

```
glVertex2f( x, y )
```

sglab.kaist.ac.kr/~sungeui/CG/Slides/CS380_Lab_1.pptx

[Practice] Using other forms of OpenGL Functions

```
import numpy as np

def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glColor3ub(255, 0, 0)
    glVertex2fv((0.0, 1.0))
    glVertex2fv([-1.0,-1.0])
    glVertex2fv(np.array([1.0,-1.0]))
    glEnd()
```

Quiz #3

- Go to https://www.slido.com/
- Join #cg-hyu
- Click “Polls”

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked for “attendance”.

GLFW Input Handling

- *glfw.poll_events()*
  - Processes events that have already been received and then returns immediately.
  - Calls a user-registered callback function for each type of events.

Event type	Set a callback using...
Key input	`glfw.set_key_callback()`

Mouse cursor position	`glfw.set_cursor_pos_callback()`
	or just poll the position using
	`glfw.get_cursor_pos()`

Mouse button	`glfw.set_mouse_button_callback()`
Mouse scroll	`glfw.set_scroll_callback()`

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

def render():
    pass

def key_callback(window, key, scancode, action, mods):
    if 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)'%glfw.get_cursor_pos(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)'%glfw.get_cursor_pos(window))
        elif action==glfw.RELEASE:
            print('release left btn: (%d, %d)'%glfw.get_cursor_pos(window))

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

```python
def main():
    # Initialize the library
    if not glfw.init():
        return
    # Create a windowed mode window and its OpenGL context
    window = glfw.create_window(640, 480, "Hello World", None, None)
    if not window:
        glfw.terminate()
        return

    glfw.set_key_callback(window, key_callback)
    glfw.set_cursor_pos_callback(window, cursor_callback)
    glfw.set_mouse_button_callback(window, button_callback)
    glfw.set_scroll_callback(window, scroll_callback)

    # Make the window's context current
    glfw.make_context_current(window)

    # Loop until the user closes the window
    while not glfw.window_should_close(window):
        # Poll for and process events
        glfw.poll_events()
        # Render here, e.g. using pyOpenGL
        render()
        # Swap front and back buffers
        glfw.swap_buffers(window)

    glfw.terminate()
if __name__ == "__main__":
    main()
```

Documentation for glfw

- http://www.glfw.org/documentation.html

- Note there are changes in the python binding:
  - function names use the pythonic **words_with_underscores** notation instead of camelCase
  - **GLFW_ and glfw prefixes have been removed**, as their function is replaced by the module namespace
  - functions like glfwGetMonitors **return a list instead of a pointer and an object count**
  - see https://pypi.python.org/pypi/glfw for more information

Legacy OpenGL & Modern OpenGL

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

- Modern OpenGL (OpenGL 2.x~)
  - Now programmable hardware is the common industry practice
  - Use of programmable shaders
  - More difficult to program but far more flexible & powerful

OpenGL as a Learning Tool

- My focus is on fundamental computer graphics ideas, not on concrete implementation.

- So I choose the legacy OpenGL as a basic learning tool, thanks to its simplicity.

- Note that legacy OpenGL is **just one implementation example** of fundamental computer graphics ideas we'll learn.

- Other implementations:
  - Graphics libraries: Modern OpenGL, DirectX, Vulkan, Nvidia Optix, ...
  - Game engines: Unreal, Unity, ...
  - Authoring tools: Maya, Blender, ...

Next Time

- Lab in this week:
  - How to setup Gitlab, Lab assignment 2

- Next lecture:
  - 3 - Transformation 1