Computer Graphics

4 - Lab – 2D Transformations, Affine Space / Frame / Matrix

Yoonsang Lee
Hanyang University

Spring 2023

Outline

- Using NumPy / PyGLM Matrices with OpenGL
- 2D Linear Transformations
- 2D Affine Transformations (using Homogeneous Coordinates)
- Animating Transformations
- Drawing Multiple Objects - Reference Frames

Using NumPy / PyGLM Matrices with OpenGL

Matrix Storage Convention

- **NumPy** uses a **row-major** storage convention.
  - Elements are stored in contiguous memory **row by row**.
  - Element access: [**row_index**, column_index]

- **PyGLM** uses a **column-major** storage conventions.
  - Elements are stored in contiguous memory **column by column**.
  - Element access: [**column_index**, row_index]

Matrix:

```
| a b |
| c d |
```

Row-major representation:

```
[a, b, c, d]
```

Column-major representation:

```
[a, c, b, d]
```

[Code] 1-numpy-pyglm-matrix

```python
import glm
import numpy as np

# numpy matrix creation
M_np = np.array([[1., 2.],
                 [0., 1.]])
print('M_np:')
print(M_np)
print()

# M_np:
# [[1. 2.]
#  [0. 1.]]

# numpy indexing: [row_index, col_index]

# first row
print('M_np[0]:', M_np[0])

# element at first row, second col
print('M_np[0,1]:', M_np[0,1])

# M_np[0]: [1. 2.]
# M_np[0,1]: 2.0
```

```python
print()

# glm matrix creation
M_glm = glm.mat2(1., 0.,
                 2., 1.)
print('M_glm:')
print(M_glm)
print()

# M_glm:
# [ 1 ][ 2 ]
# [ 0 ][ 1 ]

# glm indexing: [col_index, row_index]

# first col
print('M_glm[0]:', M_glm[0])

# element at first col, second row
print('M_glm[0,1]:', M_glm[0,1])

# M_glm[0]: mvec2( 1, 0 )
# M_glm[0,1]: 0.0
```


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

Matrix Storage Convention

- In practice, PyGLM's column-major convention does not match the convention in mathematics, which makes it counter-intuitive.
- However, **OpenGL** uses the **column-major** convention.
  - Recall that GLM faithfully emulates GLSL vector/matrix operations.
- You can **transpose** NumPy's row-major matrix to match OpenGL's column-major convention.

For this **M**:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

OpenGL expects to take:

```
[a, c, b, d]
```

But NumPy express **M** as:

```
[a, b, c, d]
```

$\mathbf{M^T}$:

$$\begin{vmatrix} a & c \\ b & d \end{vmatrix}$$

NumPy express $\mathbf{M^T}$ as:

```
[a, c, b, d]
```

2D Linear Transformations

Recall: 2D Linear Transformations

- 2x2 matrices represent 2D linear transformations such as:
  - uniform scaling, non-uniform scaling, rotation, shearing, reflection

- For example, non-uniform scaling:

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$$

[Code] 2-linear-transform-2D

- Vertex shader
  - Transformations typically vary from frame to frame, so let's pass a 2x2 matrix through a uniform variable.

```
#version 330 core
layout (location = 0) in vec3 vin_pos;
layout (location = 1) in vec3 vin_color;
out vec4 vout_color;

uniform mat2 M;

void main()
{
    // 3D point in homogeneous coordinates
    gl_Position = vec4(0, 0, 0, 1.0);

    // setting x, y coordinate values of gl_Position
    gl_Position.xy = M * vin_pos.xy;

    vout_color = vec4(vin_color, 1);
}
```

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

[Code] 2-linear-transform-2D

```
    ...
    # get uniform locations
    M_loc = glGetUniformLocation(shader_program,
'M')
    # update uniforms
    glUseProgram(shader_program)     # updating
uniform require you to first activate the shader
program

    use_numpy = True
    # use_numpy = False

    if(use_numpy):
        # numpy

        # 2x2 identity matrix
        # M = np.array([[1., 0.],
                        # [0., 1.]])    # or
        M = np.identity(2)

        # # uniform scaling
        # M = np.array([[2., 0.],
                        # [0., 2.]])


        ...
        # # shearing in x
        # M = np.array([[1., 2.],
                        # [0., 1.]])
        ...
        # print(M)

        # note that 'transpose' (3rd parameter) is
set to GL_TRUE
        # because numpy array is row-major.
        glUniformMatrix2fv(M_loc, 1, GL_TRUE, M)
```

- main()

```
else:
        # glm

        # 2x2 identity matrix
        # M = glm.mat2(1., 0.,
                       # 0., 1.)
        M = glm.mat2()

        # # uniform scaling
        # M = glm.mat2(2., 0.,
                       # 0., 2.)
        ...
        # # shearing in x
        # # # not this matrix!:
        # # M = glm.mat2(1., 2.,
                       # # 0., 1.)
        # # note that glm matrix is column-major
(numpy array is row-major)
        # # correct matrix is:
        # M = glm.mat2(1., 0.,
                       # 2., 1.)
        ...
        # print(M)

        # note that 'transpose' (3rd parameter) is
set to GL_FALSE
        # because glm matrix is column-major.
        glUniformMatrix2fv(M_loc, 1, GL_FALSE,
glm.value_ptr(M))
```

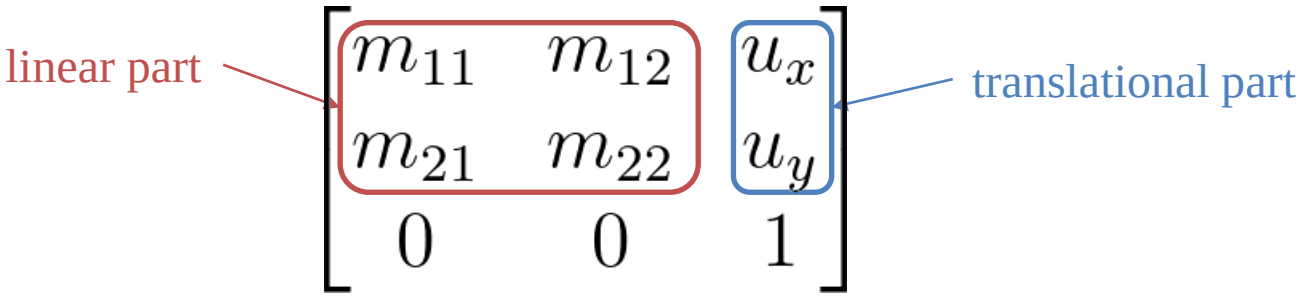
glUniformMatrix*()

- `glUniformMatrix*(location, count, transpose, value)`
  - Specify the value of a uniform matrix variable.
  - `location`: Location of a uniform variable.
  - `count`: Number of matrices. Use 1 to pass a single matrix.
  - `transpose`: Whether to transpose the matrix as the values are loaded in.
  - `value`: Pointer to an array of count values.

2D Affine Transformations (using Homogeneous Coordinates)

Recall: Affine Transformations in 2D

- In homogeneous coordinates, **2D** affine transformations can be represented as multiplication of **3x3 matrix**:



[Code] 3-affine-transform-2D-homogeneous-coord

- Vertex shader

```
#version 330 core
layout (location = 0) in vec3 vin_pos;
layout (location = 1) in vec3 vin_color;
out vec4 vout_color;
uniform mat3 M;

void main()
{
    // 3D point in homogeneous coordinates
    gl_Position = vec4(0, 0, 0, 1.0);

    // 2D points in homogeneous coordinates
    vec3 p2D_in_hcoord = vec3(vin_pos.x, vin_pos.y, 1.0);
    vec3 p2D_new_in_hcoord = M * p2D_in_hcoord;

    // setting x, y coordinate values of gl_Position
    gl_Position.xy = p2D_new_in_hcoord.xy;

    vout_color = vec4(vin_color, 1);
}
```

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

[Code] 3-affine-transform-2D-homogeneous-coord

```
def main():
    ...
    # get uniform locations & update uniforms
    M_loc = glGetUniformLocation(shader_program, 'M')
    glUseProgram(shader_program)

    # rotation 30 deg
    th = np.radians(30)
    R = np.array([[np.cos(th), -np.sin(th), 0.],
                  [np.sin(th),  np.cos(th), 0.],
                  [0.,          0.,         1.]])

    # tranlation by (.5, .2)
    T = np.array([[1., 0., .5],
                  [0., 1., .2],
                  [0., 0., 1.]])

    M = R
    # M = T
    # M = R @ T   # '@' is matrix-matrix multiplication operator
    # M = T @ R

    # print(M)

    # note that 'transpose' (3rd parameter) is set to GL_TRUE
    # because numpy array is row-major.
    glUniformMatrix3fv(M_loc, 1, GL_TRUE, M)
```

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!

Animating Transformations

Recall: For Animation

- For animation, we need to update uniforms every frame.
- Get uniform locations at initialization.
- Update uniforms in the rendering loop.

```
glGetUniformLocation

while:
    glUseProgram
    glUniform*

    glBindVertexArray(VAO)
    glDrawArrays
```

[Code] 4-animating-transform

```
def main():
    ...
    # get uniform locations & update uniforms
    M_loc = glGetUniformLocation(shader_program, 'M')
    ...
    while not glfwWindowShouldClose(window):
        ...
        glUseProgram(shader_program)

        t = glfwGetTime()
        # rotation 30 deg
        th = np.radians(t*90)
        R = np.array([[np.cos(th), -np.sin(th), 0.],
                      [np.sin(th),  np.cos(th), 0.],
                      [0.,          0.,         1.]])
        # tranlation by (.5, .2)
        T = np.array([[1., 0., np.sin(t)],
                      [0., 1., .2],
                      [0., 0., 1.]])
        M = R
        # M = T
        # M = R @ T
        # M = T @ R
        glUniformMatrix3fv(M_loc, 1, GL_TRUE, M)

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

Drawing Multiple Objects - Reference Frames

Drawing Multiple Objects

- Basically, you can use multiple VAOs to render multiple objects (meshes) – one VAO for one object.

```
vao_ship = initialize_ship_vao()
vao_enemy = initialize_enemy_vao()

while:
    # update uniform for ship
    glUniform

    # draw ship
    glBindVertexArray(vao_ship)
    glDrawArrays

    # update uniform for enemy
    glUniform

    # draw enemy
    glBindVertexArray(vao_enemy)
    glDrawArrays
```

Drawing Multiple Objects at Once

- If a group of objects is rendered with the same
  - primitive type (GL_TRIANGLES, …)
  - set of vertex attributes and their configurations
  - shader program
  - uniform states
- , they can be efficiently rendered in **a single draw call** (such as glDrawArrays) using a single VAO.
  - To do this, these objects' data should be stored in the same VBO (or the same group of VBOs - different VBOs for different attributes).

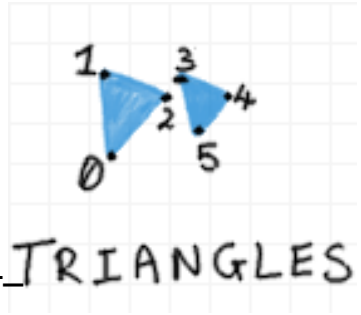
- Drawing multiple objects at once can be helpful when dealing with a large number of objects (more than hundreds or thousands).

- However, the assignments and projects in this course do not require complex scenes with that many objects, so performance is not a big concern. Please consider the above as a reference.

[Code] 5-drawing-frames

```
def prepare_vao_triangle():
    # prepare vertex data (in main memory)
    vertices = glm.array(glm.float32,
        # position          # color
         0.0, 0.0, 0.0,  1.0, 0.0, 0.0, # v0
         0.5, 0.0, 0.0,  0.0, 1.0, 0.0, # v1
         0.0, 0.5, 0.0,  0.0, 0.0, 1.0, # v2
    )
    # 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 positions
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6*glm.sizeof(glm.float32), None)
    glEnableVertexAttribArray(0)

    # configure vertex colors
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * glm.sizeof(glm.float32),
ctypes.c_void_p(3*glm.sizeof(glm.float32)))
    glEnableVertexAttribArray(1)
    return VAO
```

These vertices are rendered with GL_TRIANGLES.

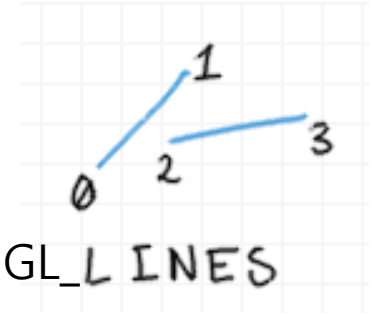


* The full source code can be found at https://github.com/yssl/CSE4020

[Code] 5-drawing-frames

```
def prepare_vao_frame():
    # prepare vertex data (in main memory)
    vertices = glm.array(glm.float32,
        # position          # color
         0.0, 0.0, 0.0,  1.0, 0.0, 0.0, # x-axis start
         1.0, 0.0, 0.0,  1.0, 0.0, 0.0, # x-axis end
         0.0, 0.0, 0.0,  0.0, 1.0, 0.0, # y-axis start
         0.0, 1.0, 0.0,  0.0, 1.0, 0.0, # y-axis end
         0.0, 0.0, 0.0,  0.0, 0.0, 1.0, # z-axis start
         0.0, 0.0, 1.0,  0.0, 0.0, 1.0, # z-axis end
    )
    ...
    return VAO
```

These vertices are rendered with GL_LINES.



[Code] 5-drawing-frames

```
# prepare vaos
vao_triangle = prepare_vao_triangle()
vao_frame = prepare_vao_frame()

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

    glUseProgram(shader_program)

    # current frame: I (world frame)
    I = np.identity(3)
    glUniformMatrix3fv(M_loc, 1, GL_TRUE, I)

    # draw current frame
    glBindVertexArray(vao_frame)
    glDrawArrays(GL_LINES, 0, 6)


    # animating
    t = glfwGetTime()

    # rotation 30 deg
    th = np.radians(t*90)
    R = np.array([[np.cos(th), -np.sin(th), 0.],
                  [np.sin(th),  np.cos(th), 0.],
                  [0.,          0.,         1.]])
```

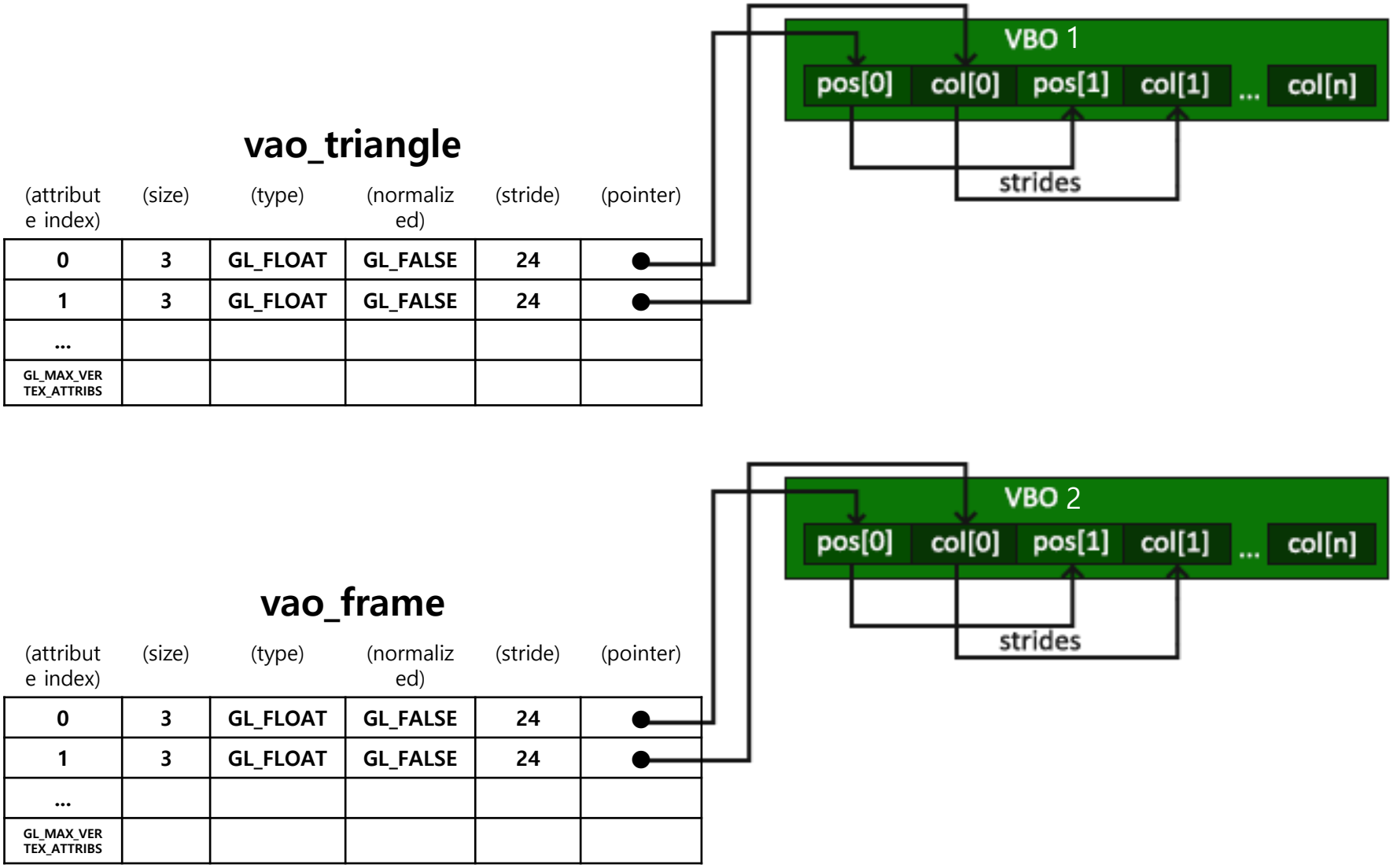
```
    # tranlation by (.5, .2)
    T = np.array([[1., 0., np.sin(t)],
                  [0., 1., .2],
                  [0., 0., 1.]])

    # M = R
    # M = T
    # M = R @ T
    M = T @ R

    # print(M)


    # current frame: M
    glUniformMatrix3fv(M_loc, 1, GL_TRUE, M)

    # draw triangle w.r.t. the current frame
    glBindVertexArray(vao_triangle)
    glDrawArrays(GL_TRIANGLES, 0, 3)

    # draw current frame
    glBindVertexArray(vao_frame)
    glDrawArrays(GL_LINES, 0, 6)

    ...
```

[Code] 5-drawing-frames

- Vertex shader (same as 3-affine-transform-2D-homogeneous-coord)

```
#version 330 core
layout (location = 0) in vec3 vin_pos;
layout (location = 1) in vec3 vin_color;
out vec4 vout_color;
uniform mat3 M;

void main()
{
    // 3D point in homogeneous coordinates
    gl_Position = vec4(0, 0, 0, 1.0);

    // 2D points in homogeneous coordinates
    vec3 p2D_in_hcoord = vec3(vin_pos.x, vin_pos.y, 1.0);
    vec3 p2D_new_in_hcoord = M * p2D_in_hcoord;

    // setting x, y coordinate values of gl_Position
    gl_Position.xy = p2D_new_in_hcoord.xy;

    vout_color = vec4(vin_color, 1);
}
```

VAO & VBO in this example



Time for Assignment

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