CSCI3260
Principles of Computer Graphics

---------Tutorial 2

XU Jiaqi

OUTLINE

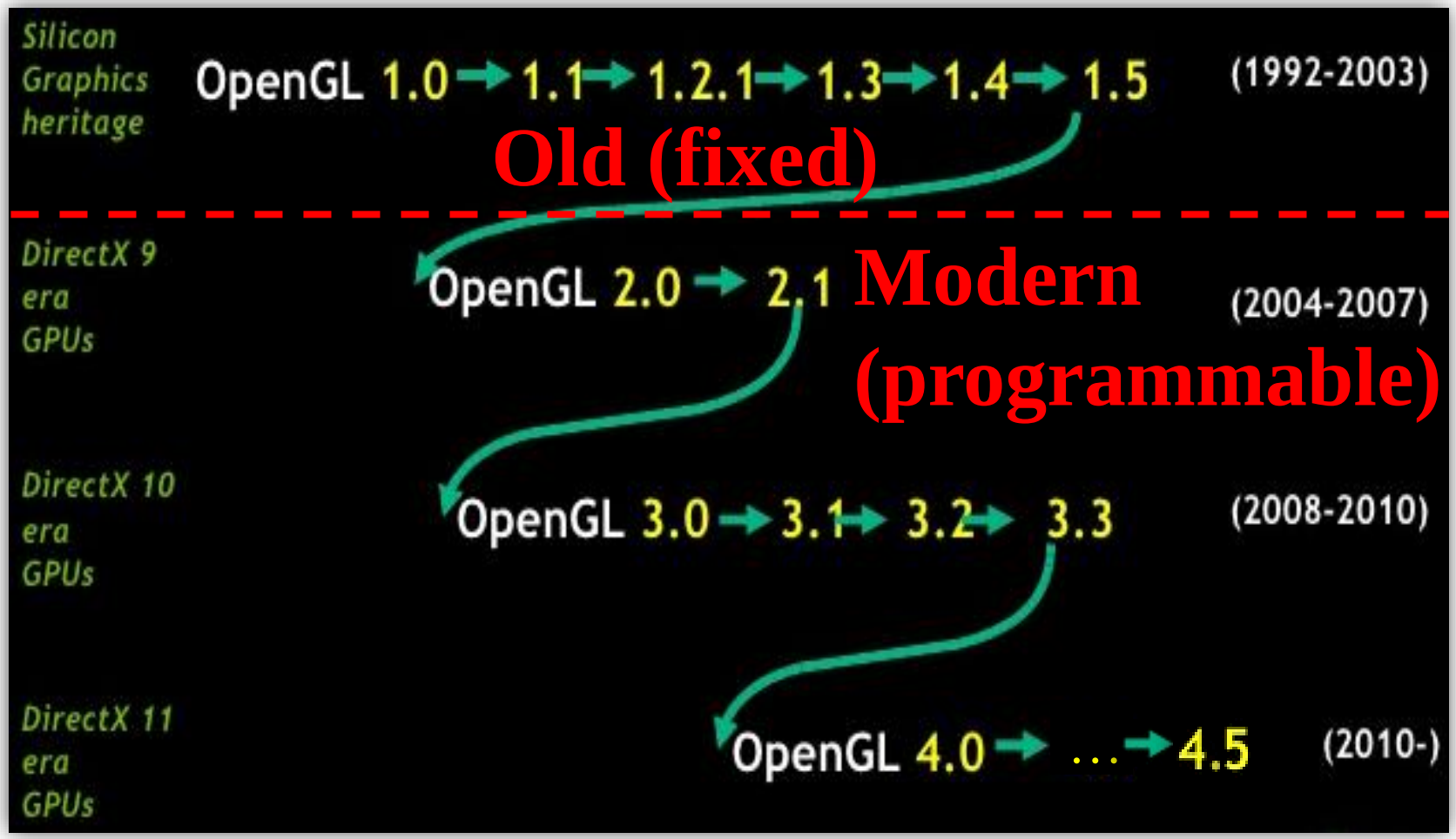
- **Old vs. Modern OpenGL**

- Identifying OpenGL version on your computer

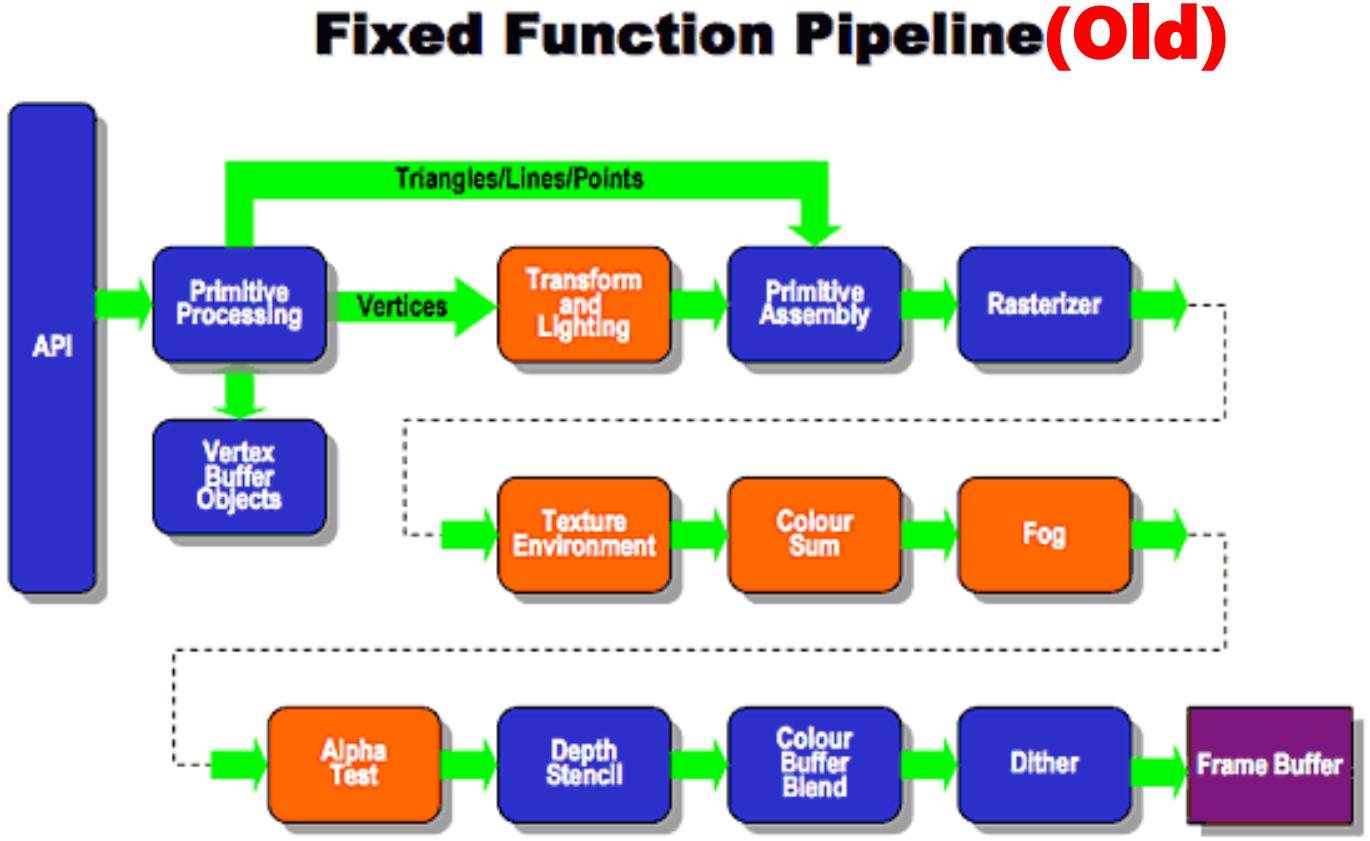
- Basic OpenGL programming

Old vs. Modern

OpenGL History:

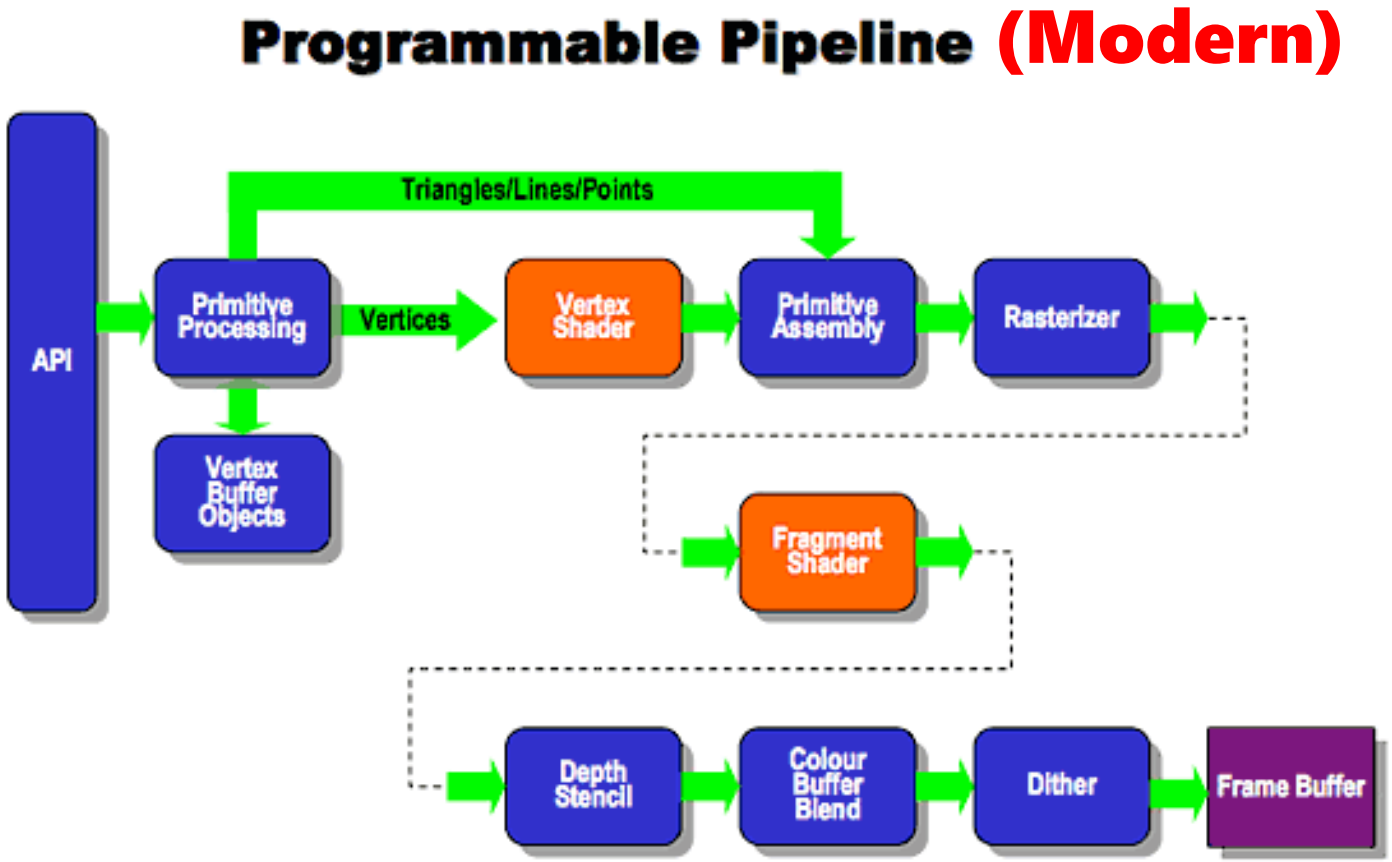


Old vs. Modern



- Older versions of OpenGL use the fixed function pipeline
- Not controllable – the exact method in which the geometry was transformed, and how fragments acquired depth and color values were build-into the hardware and cannot change

Old vs. Modern



- Modern GPUs have a programmable pipeline
- Previously build-in stages have been replaced with stages that can be controlled by coding called “shader”

Old vs. Modern

Comparison:

Fixed pipeline

Programmable pipeline

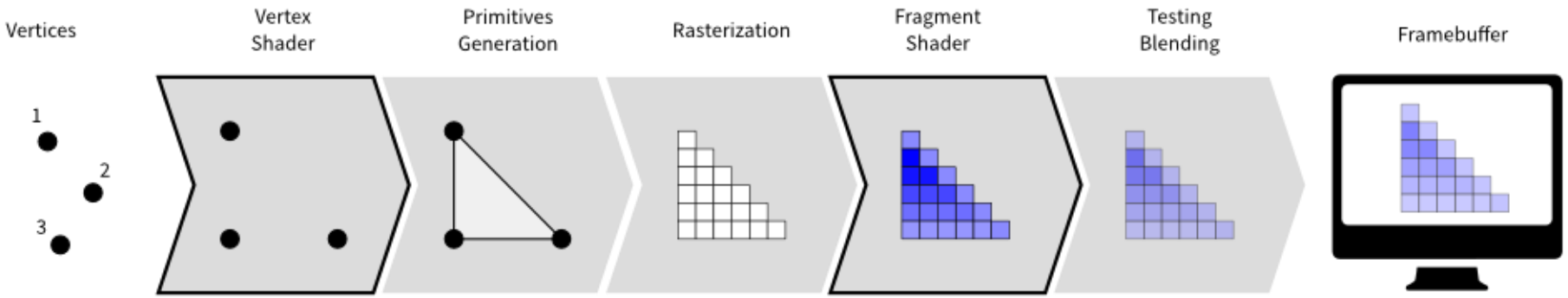


Fig. 1 Rendering pipeline

Shaders are pieces of programs (using a C-style language) that are built onto the GPU and executed during the rendering pipeline. There are many types of shaders that act at different stages of the rendering pipeline. Among them, **vertex** and **fragment** shaders are the most important.

- A vertex shader acts on vertices and is supposed to output the vertex **position**.
- A fragment shader acts at the fragment level and should output the **color**.

GLSL (OpenGL Shading Language):

- High level shading language to give developers direct control of the graphics pipeline
- In this course, we only need to do simple GLSL programming (for more info.: https://www.opengl.org/documentation/glsl/)

VertexShaderCode.glsl

```
1   #version 430
2
3   in layout(location = 0) vec3 position;
4   in layout(location = 1) vec3 vertexColor;
5
6   out vec3 theColor;
7
8   void main()
9   {
10      gl_Position = position;
11      theColor = vertexColor;
12  }
13
```

Old OpenGL codes (example):

```
glBegin ( type ) ;
    glVertex3f ( … ) ;
    glVertex3f ( … ) ;
    glVertex3f ( … ) ;
    ……
glEnd() ;
```

```
glMatrixMode ( GL_MODELVIEW ) ;
glLoadIdentity () ;
glPushMatrix() ;
  glTranslatef ( ball_X , ball_Y , ball_Z ) ;
  glRotatef ( ball_ang , ball_dirX , ball_dirY , ball_dirZ ) ;
  glScalef ( ball_Sx , ball_Sy , ball_Sz ) ;
  Draw_ball() ;
glPopMatrix() ;
glPushMatrix() ;
  glTranslatef ( cube_X , cube_Y , cube_Z ) ;
  glRotatef ( cube_ang , cube_dirX , cube_dirY , cube_dirZ ) ;
  glScalef ( cube_Sx , cube_Sy , cube_Sz ) ;
  Draw_cube() ;
glPopMatrix() ;
```

Modern OpenGL codes (example):

- Vertex Array Object (VAO): a **state-object** that stores all the state of a vertex array
- Vertex Buffer Object (VBO): a **buffer-object** that holds a vertex array

```
const GLfloat triangle[] =
{
    -0.5f, -0.5f, +0.0f,  // position, left
    +1.0f, +0.0f, +0.0f,  // color

    +0.5f, -0.5f, +0.0f,  // right
    +1.0f, +0.0f, +0.0f,

    +0.0f, +0.5f, +0.0f,  // top
    +1.0f, +0.0f, +0.0f,
};
GLuint vaoID;
```

```
GLuint vaoID;
glGenVertexArrays(1, &vaoID);
glBindVertexArray(vaoID);  // first VAO
GLuint vboID;
glGenBuffers(1, &vboID);
glBindBuffer(GL_ARRAY_BUFFER, vboID);
glBufferData(GL_ARRAY_BUFFER, sizeof(triangle), triangle, GL_STATIC_DRAW);
// 1st attribute: vertex position
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), 0);
// 2nd attribute: vertex color
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
    (char*)(3 * sizeof(float)));
```

Old vs. Modern

OpenGL function replacements

glRotate[fd]
glm::rotate
glScale[fd]
glm::scale
glTranslate[fd]
glm::translate
glLoadIdentity
The default constructor of all matrix types creates an identity matrix.
glMultMatrix[fd]
Per the GLSL specification, the multiplication operator is overloaded for all matrix types. Multiplying two matrices together will perform matrix multiplication.
glLoadTransposeMatrix[fd]
glm::transpose
glMultTransposeMatrix
Combine the last two.
glFrustum
glm::frustum
glOrtho
glm::ortho
gluLookAt
glm::lookAt

GLU function replacements

gluOrtho2D
glm::ortho
gluPerspective
glm::perspective
gluProject
glm::project
gluUnProject
glm::unProject

Try to avoid these deprecated functions!

GLM (OpenGL Mathematics) is a C++ mathematics library that provides the same functions and easy to use.

For mac:

brew install glm

OUTLINE

- Old vs. Modern OpenGL

➢ **Identifying OpenGL version on your computer**

- Basic OpenGL programming

OpenGL version Check

Check OpenGL on your computer #1:

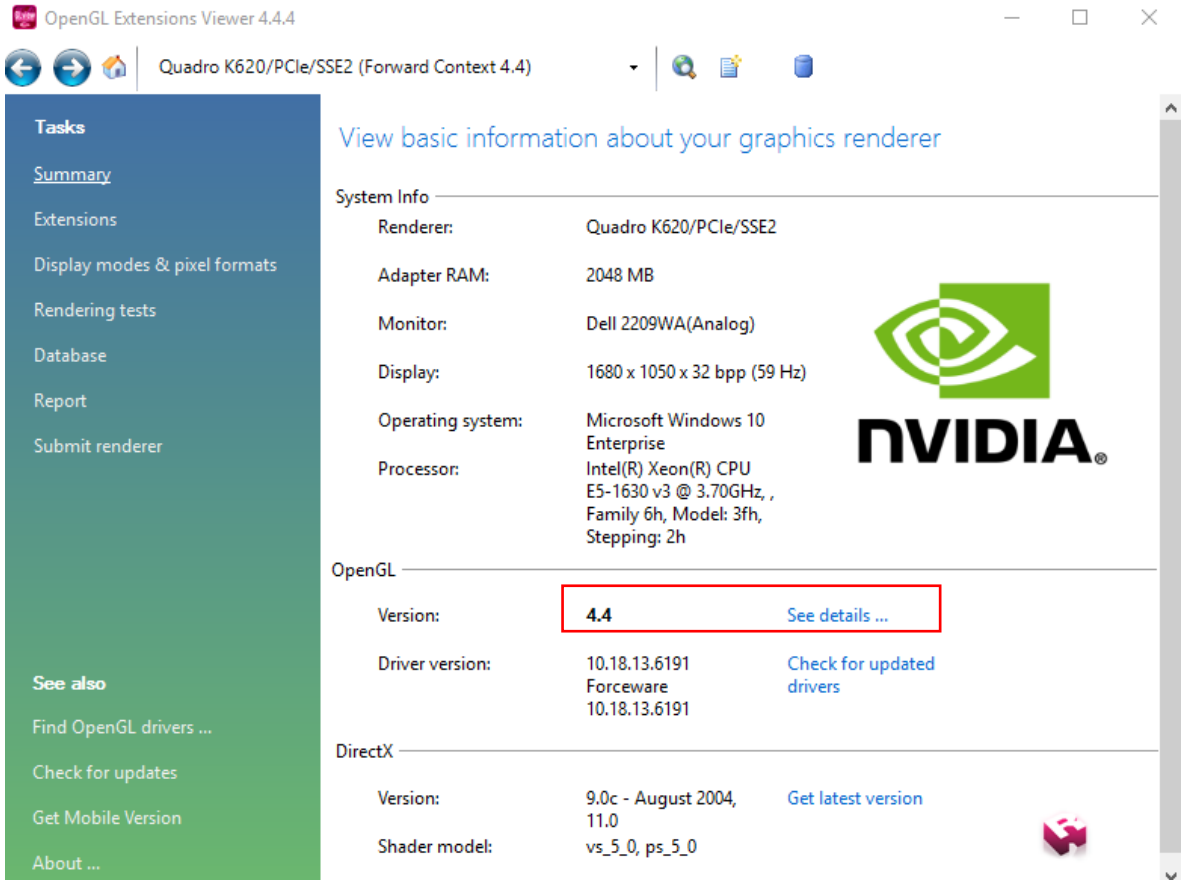
```
void get_OpenGL_info() {
    // OpenGL information
    const GLubyte* name = glGetString(GL_VENDOR);
    const GLubyte* renderer = glGetString(GL_RENDERER);
    const GLubyte* glversion = glGetString(GL_VERSION);
    std::cout << "OpenGL company: " << name << std::endl;
    std::cout << "Renderer name: " << renderer << std::endl;
    std::cout << "OpenGL version: " << glversion << std::endl;
}
```

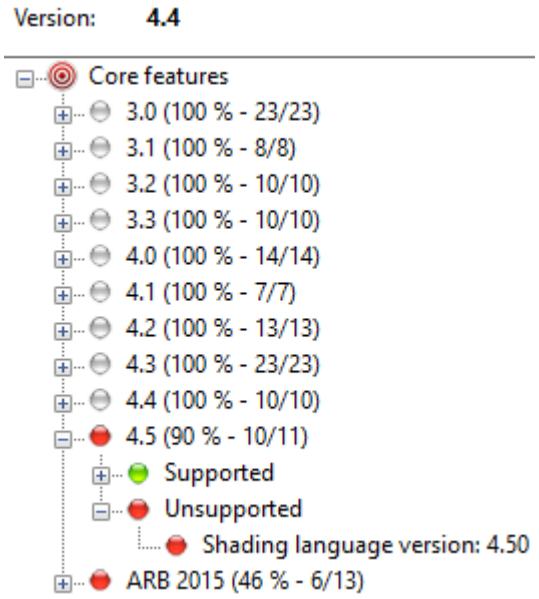
```
OpenGL company: NVIDIA Corporation
Renderer name: GeForce RTX 2060/PCIe/SSE2
OpenGL version: 3.3.0 NVIDIA 452.06
```

OpenGL version Check

Check OpenGL on your computer #2:

- OpenGL Extension Viewer: http://www.realtech-vr.com/glview/





OUTLINE

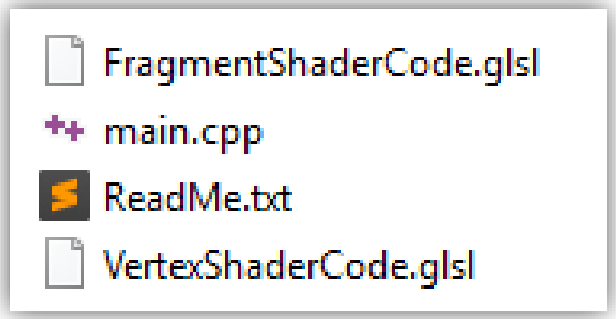
- Old vs. Modern OpenGL

- Identifying OpenGL version on your computer

➢ **Basic OpenGL programming**

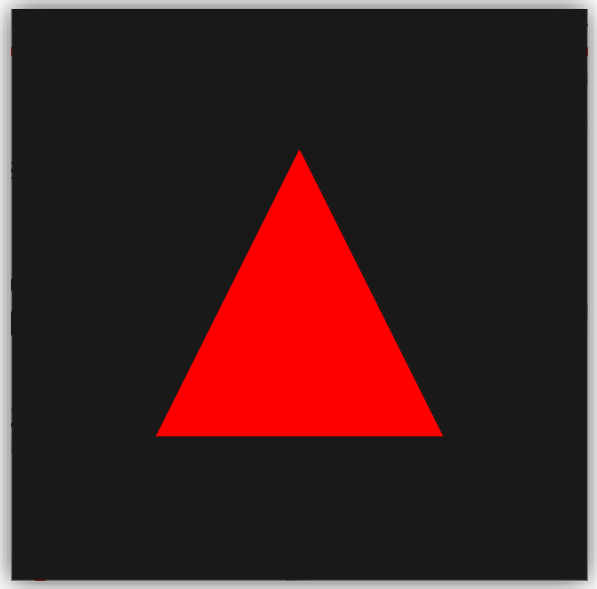
Red Triangle Demo code

(you can download from Blackboard):

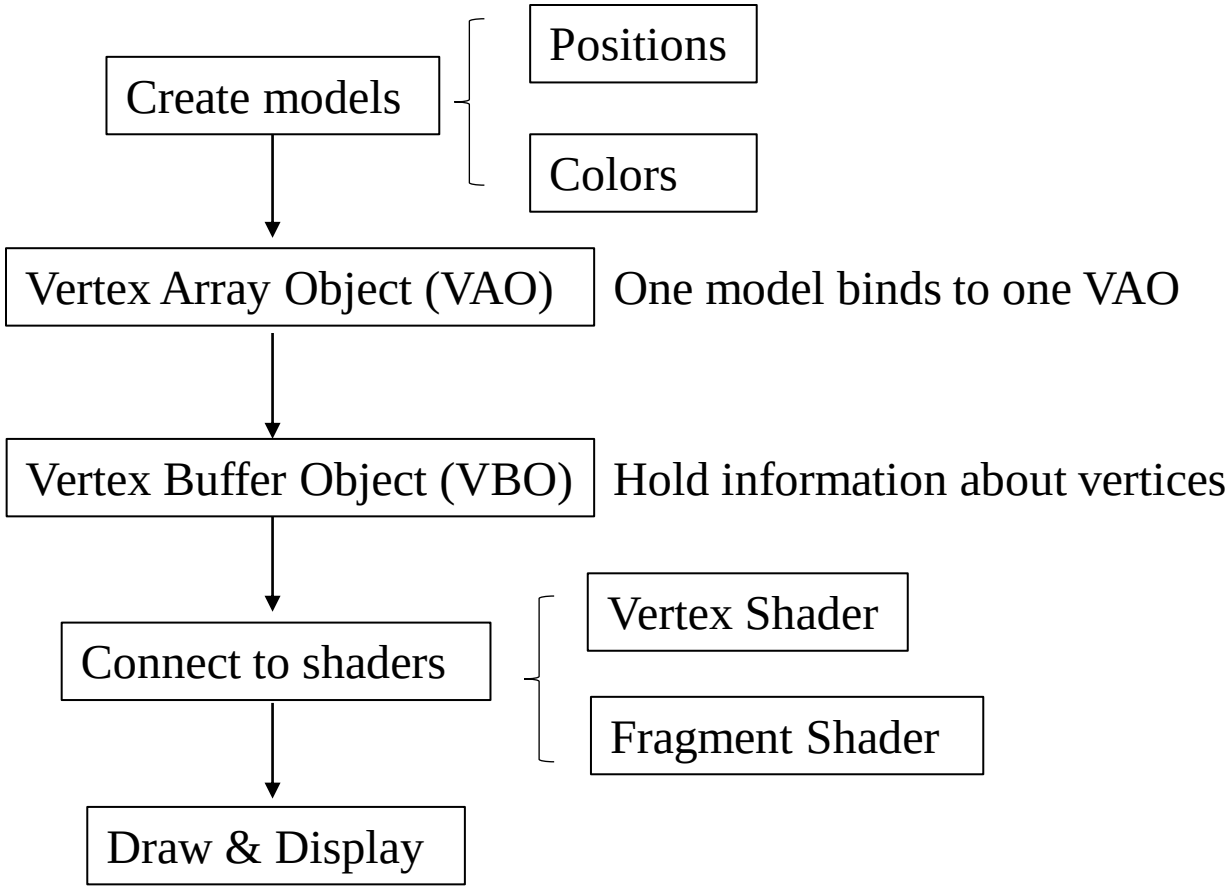


Requirements:

- Create a window with a black background color
- Draw a red triangle in the center of the window
- Press the key “a” to move the triangle left
- Press the key “d” to move the triangle right



Main steps for rendering a model:



main function (main.cpp):

```
int main(int argc, char* argv[]) {
    GLFWwindow* window;

    /* Initialize the glfw */
    if (!glfwInit()) {
        std::cout << "Failed to initialize GLFW" << std::endl;
        return -1;
    }
    /* glfw: configure; necessary for MAC */
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    /* do not allow resizing */
    glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(512, 512, argv[0], NULL, NULL);
    if (!window) {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
```

```
    /* Make the window's context current */
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
    glfwSetKeyCallback(window, key_callback);

    /* Initialize the glew */
    if (GLEW_OK != glewInit()) {
        std::cout << "Failed to initialize GLEW" << std::endl;
        return -1;
    }
    get_OpenGL_info();
    initializedGL();

    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(window)) {
        /* Render here */
        paintGL();

        /* Swap front and back buffers */
        glfwSwapBuffers(window);

        /* Poll for and process events */
        glfwPollEvents();
    }

    glfwTerminate();
    return 0;
}
```

initializedGL() function:

```
void initializedGL(void) //run only once
{
    sendDataToOpenGL();
    installShaders();
}
```

create triangle object

communicate with shaders

For the *installShaders()*, you don't need to write by yourself. We will provide this function to you when you do your assignments.

Programming

```
void sendDataToOpenGL()
{
    const GLfloat triangle[] =
    {
        -0.5f, -0.5f, +0.0f, //left
        +1.0f, +0.0f, +0.0f, //color

        +0.5f, -0.5f, +0.0f, //right
        +1.0f, +0.0f, +0.0f,

        +0.0f, +0.5f, +0.0f, //top
        +1.0f, +0.0f, +0.0f,
    };
    GLuint vaoID;
    glGenVertexArrays(1, &vaoID);
    glBindVertexArray(vaoID);  //first VAO
    GLuint vboID;
    glGenBuffers(1, &vboID);
    glBindBuffer(GL_ARRAY_BUFFER, vboID);
    glBufferData(GL_ARRAY_BUFFER, sizeof(triangle), triangle, GL_STATIC_DRAW);
    //vertex position
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), 0);
    //vertex color
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (char*)(3 * sizeof(float)));
}
```

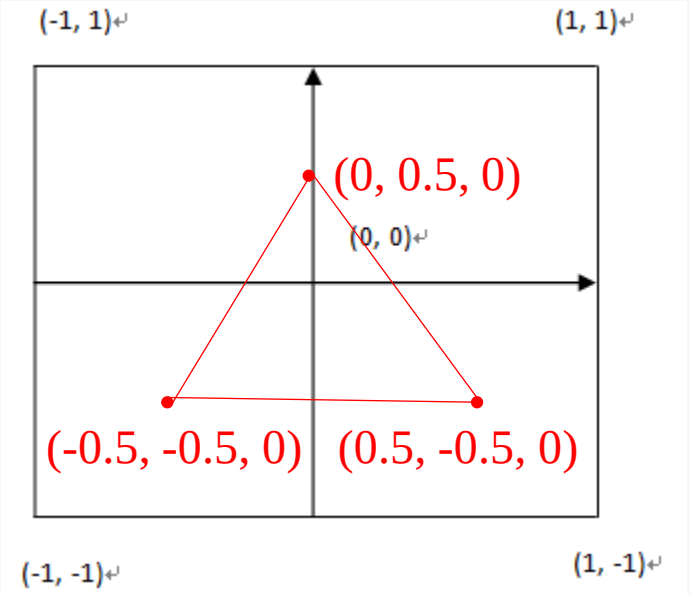
Table 1 OpenGL variable types and corresponding C data types

OpenGL Data Type	Internal Representation	Defined as C Type	C Literal Suffix
GLbyte	8-bit integer	Signed char	b
GLshort	16-bit integer	Short	s
GLint, GLsizei	32-bit integer	Long	i
GLfloat, GLclampf	32-bit floating point	Float	f
GLdouble, GLclampd	64-bit floating point	Double	d
GLubyte, GLboolean	8-bit unsigned integer	Unsigned char	ub
GLushort	16-bit unsigned integer	Unsigned short	us
GLuint, GLenum	32-bit unsigned integer	Unsigned long	ui

- *GLsizei* is an OpenGL variable denoting a size parameter that is represented by an integer.
- The *clamp* is used for color composition and stands for *color amplitude*.

Programming

```
void sendDataToOpenGL()
{
    const GLfloat triangle[] =
    {
        -0.5f, -0.5f, +0.0f, //left
        +1.0f, +0.0f, +0.0f, //color

        +0.5f, -0.5f, +0.0f, //right
        +1.0f, +0.0f, +0.0f,

        +0.0f, +0.5f, +0.0f, //top
        +1.0f, +0.0f, +0.0f,
    };
    GLuint vaoID;
    glGenVertexArrays(1, &vaoID);
    glBindVertexArray(vaoID);  //first VAO
    GLuint vboID;
    glGenBuffers(1, &vboID);
    glBindBuffer(GL_ARRAY_BUFFER, vboID);
    glBufferData(GL_ARRAY_BUFFER, sizeof(triangle), triangle, GL_STATIC_DRAW);
    //vertex position
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), 0);
    //vertex color
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (char*)(3 * sizeof(float)));
}
```



2D coordinates system

Programming

```
void sendDataToOpenGL()
{
    const GLfloat triangle[] =
    {
        -0.5f, -0.5f, +0.0f, //left
        +1.0f, +0.0f, +0.0f, //color

        +0.5f, -0.5f, +0.0f, //right
        +1.0f, +0.0f, +0.0f,

        +0.0f, +0.5f, +0.0f, //top
        +1.0f, +0.0f, +0.0f,
    };
    GLuint vaoID;
    glGenVertexArrays(1, &vaoID);
    glBindVertexArray(vaoID);  //first VAO
    GLuint vboID;
    glGenBuffers(1, &vboID);
    glBindBuffer(GL_ARRAY_BUFFER, vboID);
    glBufferData(GL_ARRAY_BUFFER, sizeof(triangle), triangle, GL_STATIC_DRAW);
    //vertex position
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), 0);
    //vertex color
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (char*)(3 * sizeof(float)));
}
```

Vertex Array Object

Vertex Buffer Object

Vertex shader & Fragment shader:

VertexShaderCode.glsl | FragmentShaderCode.glsl | main.cpp

```
1   #version 430
2
3   in layout(location=0) vec3 position;
4   in layout(location=1) vec3 vertexColor;
5
6   uniform mat4 modelTransformMatrix;
7
8   out vec3 theColor;
9
10  void main()
11  {
12      vec4 v = vec4(position, 1.0);
13      vec4 out_position = modelTransformMatrix * v;
14      gl_Position = out_position;
15      theColor = vertexColor;
16  }
```

VertexShaderCode.glsl | FragmentShaderCode.glsl | main.cpp

```
1   #version 430
2
3   out vec4 Color;
4   in vec3 theColor;
5
6   void main()
7   {
8       Color = vec4(theColor, 1.0);
9   }
10
```

Subtle difference between *win* and *mac*.
Refer to the template demo.

The first line of Shader codes:

VertexShaderCode.glsl*

```
#version 430
```

⇒ To specify which version of GLSL should be used to compile/link a shader

GLSL versions have evolved alongside specific versions of the OpenGL API. It is only with OpenGL version 3.3 and above that the GLSL and OpenGL version numbers match. These versions for GLSL and OpenGL are related in the following table.

GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59[1]	2.0	April 2004	#version 110
1.20.8[2]	2.1	September 2006	#version 120
1.30.10[3]	3.0	August 2008	#version 130
1.40.08[4]	3.1	March 2009	#version 140
1.50.11[5]	3.2	August 2009	#version 150
3.30.6[6]	3.3	February 2010	#version 330
4.00.9[7]	4.0	March 2010	#version 400
4.10.6[8]	4.1	July 2010	#version 410
4.20.11[9]	4.2	August 2011	#version 420
4.30.8[10]	4.3	August 2012	#version 430
4.40[11]	4.4	July 2013	#version 440
4.50[12]	4.5	August 2014	#version 450

Programming

In OpenGL, there is another special variable type:
Uniform Variables: used to communicate with the vertex or fragment shader from "outside". In your shader you use the **uniform** to declare the variable:

```
uniform float myVariable;
```

Vertex or Fragment Shader Source Code

Uniform variables are **read-only** in the shader codes. You can only change them within your C++ program.

Example:

vertex shader

```
# version 430
uniform float Scale;

void main (void)
{
    vec4 a = gl_Vertex;
    a.x = a.x * Scale;
    a.y = a.y * Scale;

    gl_Position = gl_ModelVeiwProjectionMatrix * a;
}
```

fragment shader

```
#version 430
uniform vec4 color;

void main (void)
{
    gl_FragColor = color;
}
```

Programming

Changing the Uniform value in C++:

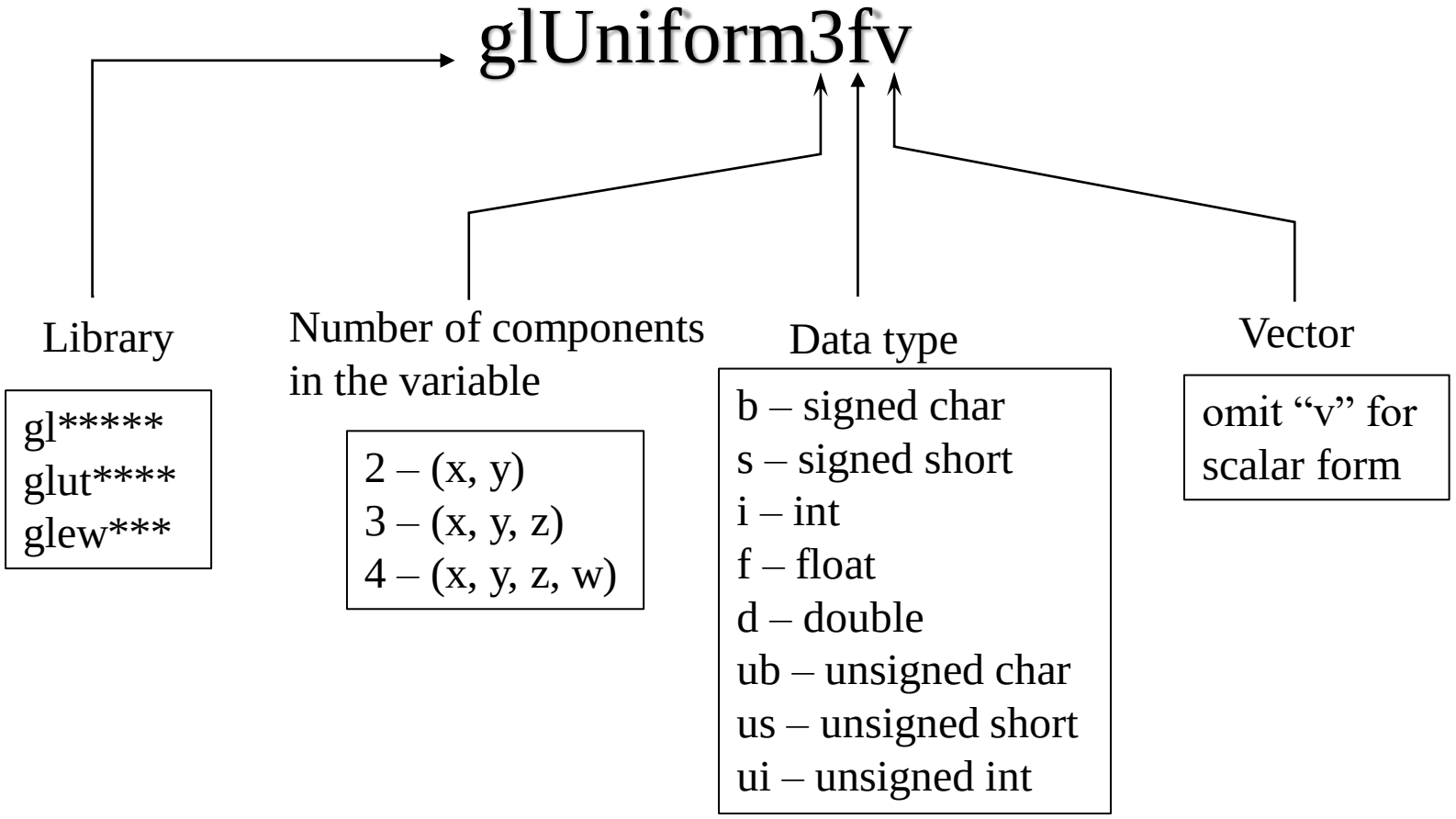
C++ source code

```
GLint loc = glGetUniformLocation (ProgramObject, “Scale”);

if (loc != -1)
{
    glUniform1f (loc, 0.432);
}
```

glGetUniformLocation: get the location of the uniform variable within the specified program object.
glUniform1f: set the value of the uniform variable.

OpenGL function naming convention:

(Specify the value of a uniform variable for the current program object)



glUniform:

void glUniform1f (GLint *location*, GLfloat *v0*);

void glUniform2f (GLint *location*, GLfloat *v0*, GLfloat *v1*);

void glUniform3f (…), void glUniform4f (…)

void glUniform1i (GLint *location*, GLint *v0*);

void glUniform2i (GLint *location*, GLint *v0*, GLint *v1*);

void glUniform3i (…), void glUniform4i (…)

void glUniform1fv (GLint *location*, GLsizei *count*, const GLfloat **value*);

void glUniform2fv (…), void glUniform3fv (…), …

void glUniform2iv (…), void glUniform3iv (…), …

paintGL() function:

```
void paintGL(void) {
    // always run
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);  //specify the background color
    glClear(GL_COLOR_BUFFER_BIT);

    glm::mat4 modelTransformMatrix = glm::mat4(1.0f);
    modelTransformMatrix = glm::translate(glm::mat4(1.0f),
        glm::vec3(x_delta * x_press_num, 0.0f, 0.0f));;
    GLint modelTransformMatrixUniformLocation =
        glGetUniformLocation(programID, "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
        GL_FALSE, &modelTransformMatrix[0][0]);

    glDrawArrays(GL_TRIANGLES, 0, 6);  //render primitives from array data
}
```

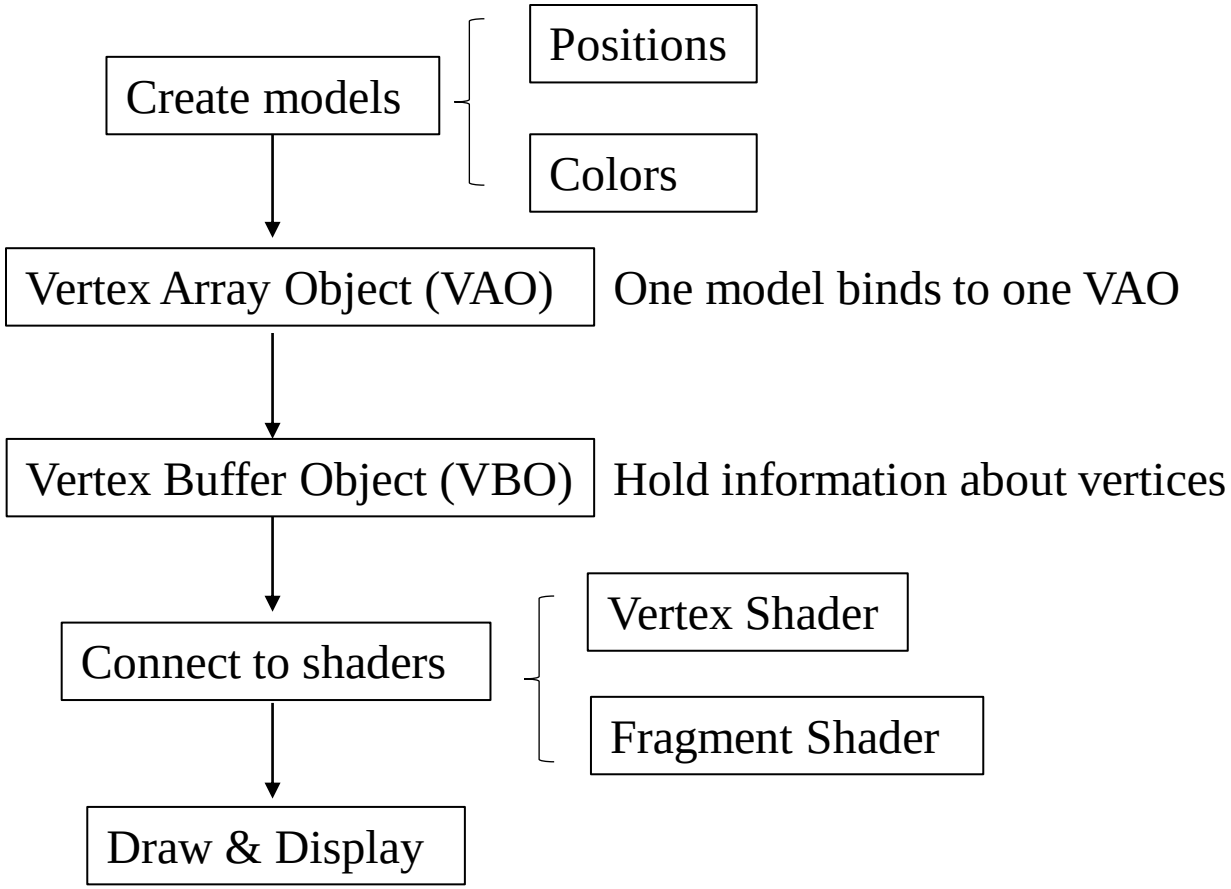
Keyboard function:

```
VertexShaderCode.glsl  FragmentShaderCode.glsl  main.cpp
1  #version 430
2
3  in layout(location=0) vec3 position;
4  in layout(location=1) vec3 vertexColor;
5
6  uniform mat4 modelTransformMatrix;
7
8  out vec3 theColor;
9
10 void main()
11 {
12     vec4 v = vec4(position, 1.0);
13     vec4 new_position = modelTransformMatrix * v;
14     gl_Position = new_position;
15     theColor = vertexColor;
16 }
```

```
void paintGL(void) {
    // always run
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);  //specify the backgr
    glClear(GL_COLOR_BUFFER_BIT);

    glm::mat4 modelTransformMatrix = glm::mat4(1.0f);
    modelTransformMatrix = glm::translate(glm::mat4(1.0f),
        glm::vec3(x_delta * x_press_num, 0.0f, 0.0f));;
    GLint modelTransformMatrixUniformLocation =
        glGetUniformLocation(programID, "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
        GL_FALSE, &modelTransformMatrix[0][0]);

    glDrawArrays(GL_TRIANGLES, 0, 6);  //render primitives from array data
}
```

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods) {
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (key == GLFW_KEY_A && action == GLFW_PRESS) {
        x_press_num -= 1;
    }
    if (key == GLFW_KEY_D && action == GLFW_PRESS) {
        x_press_num += 1;
    }
}
```

Main steps for rendering a model:



- **Next tutorial**:
  - Introduction to assignment 1
  - How to render 3D models