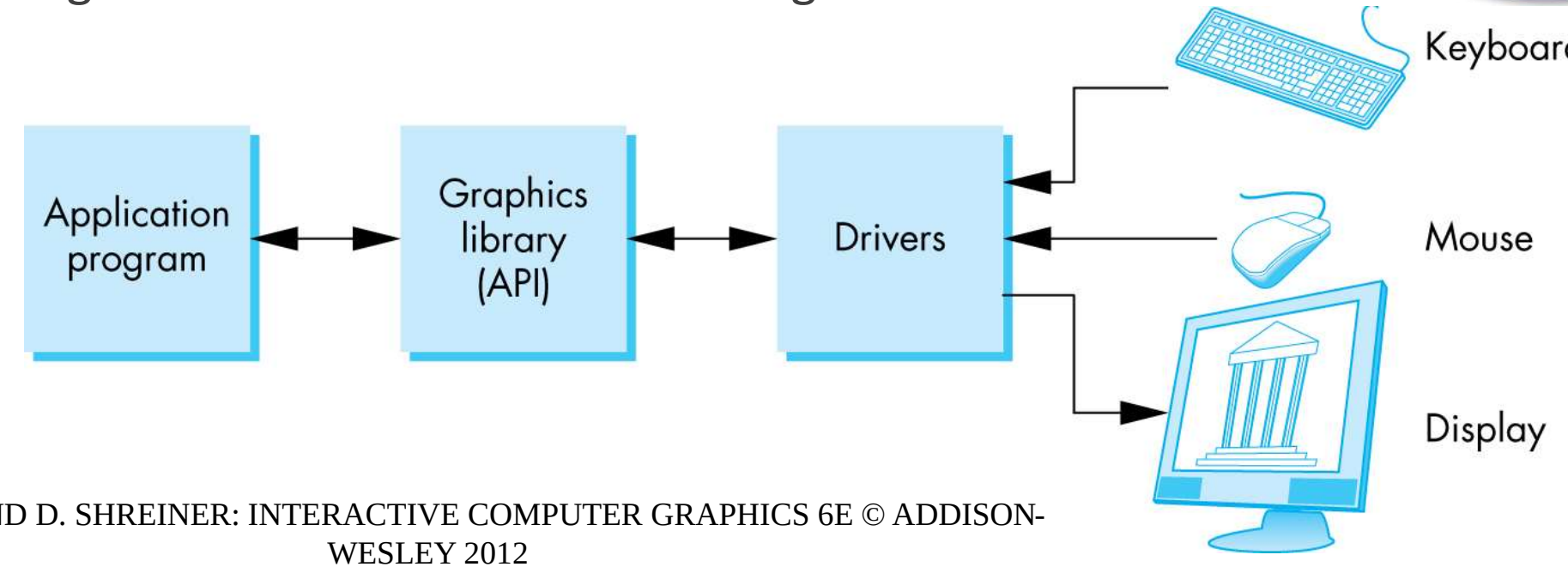
OpenGL programming: Getting Started

COMPUTER GRAPHICS, (COURSE-HY23945)

Q YOUN HONG

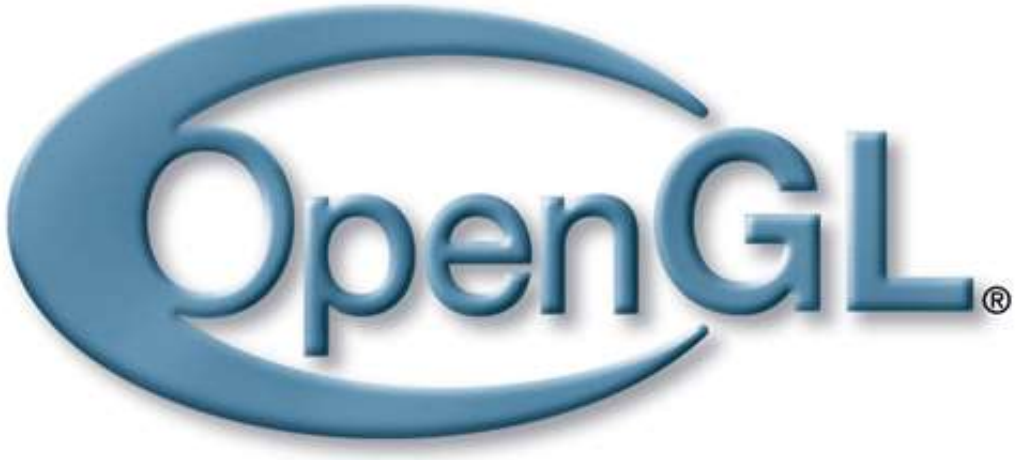
OpenGL(Open Graphics Library)

- OpenGL is a software API to graphics hardware
  - Designed as a cross-platform API (Application Programming Interface)
  - Intuitive, procedural interface with c binding
  - No windowing commands (GLUT for windowing)
  - No high-level commands for describing 3D models



OpenGL Version History

- Initiated by Silicon Graphics, Inc. (SGI) (1991)
- OpenGL 1.0 (1992):
  - “Legacy” OpenGL
- OpenGL 1.5 (2003):
  - Vertex Buffer Object (VBO)
- OpenGL 2.0 (2004):
  - GLSL 1.1
- OpenGL 3.0 (2008):
  - Frame Buffer Object (FBO)
- OpenGL 4.0 (2010)
- OpenGL 4.6 (2017):
  - Last release up to this point.



OpenGL Features

- Core-profile vs. Immediate mode
  - Immediate mode – fixed function pipeline, easy-to-draw, limited
  - Core-profile mode – more flexible, need more knowledge about graphics programming and rendering pipeline
- OpenGL (≥ Ver. 1.1) supports extensions
  - Extensions are implemented by a graphics driver (check with glewinfo)
- OpenGL is a state machine!
  - A program has OpenGL context storing global variables
  - OpenGL functions are changing or using states

OpenGL Libraries

- GL (Graphics Library):
  - OpenGL core library
  - Library of 2D, 3D drawing primitives and operations
- GLU (OpenGL Utilities):
  - Miscellaneous functions dealing with camera set-up and higher-level shape descriptions
  - Can be only used with legacy code
- GLUT(GL Utility Toolkit):
  - Window-system independent toolkit with numerous utility functions, mostly dealing with user interface

GLUT (OpenGL Utility Toolkit)

- Provides functionality common to all window systems
  - Open a window
  - Get input from mouse and keyboard
  - Menus
  - Event handlers

- Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
  - No slide bars

freeGLUT

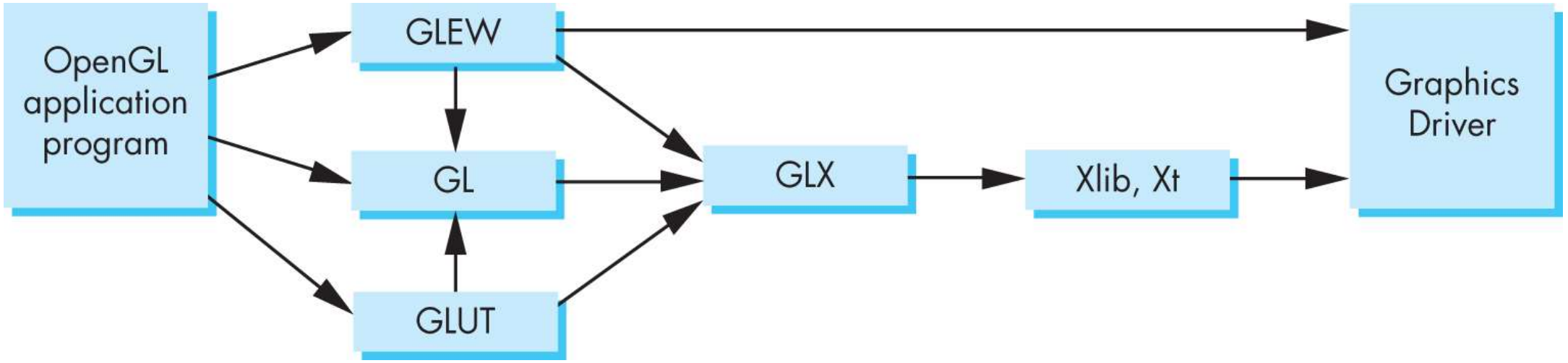
- GLUT was created long ago and has been unchanged
  - Amazing that it works with OpenGL 3.1
  - Some functionality can’t work since it requires deprecated functions

- freeGLUT updates GLUT
  - Added capabilities
  - Context checking

GLEW

- OpenGL Extension Wrangler Library
- Makes it easy to access OpenGL extensions available on a particular system
- Avoids having to have specific entry points in Windows code
- Application needs only to include glew.h and run a glewInit()

OpenGL Libraries Organization



Example: Hello OpenGL!

Preparation

- Visual Studio C/C++
- freeGLUT
  - http://freeglut.sourceforge.net/
- GLEW
  - http://glew.sourceforge.net/

✓ GLlibs.zip: freeGLUT and GLEW for x64 on visual studio 2019

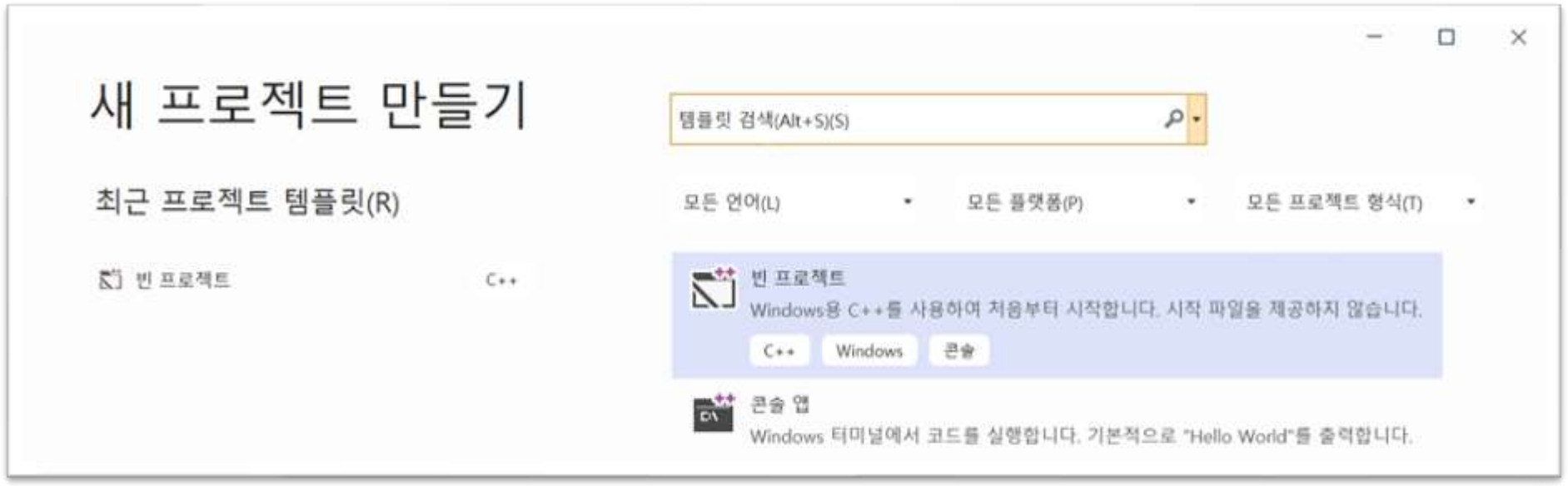
✓ Optional: need Cmake if you need to compile freeGLUT and GLEW yourself!

Preparation

- Download necessary libraries
  - Header files: Include folder
  - LIB files: lib folder
  - DLL files: bin (or system32, system64) folder
- Create a New Project
- Change the project setting
  - Directory setting

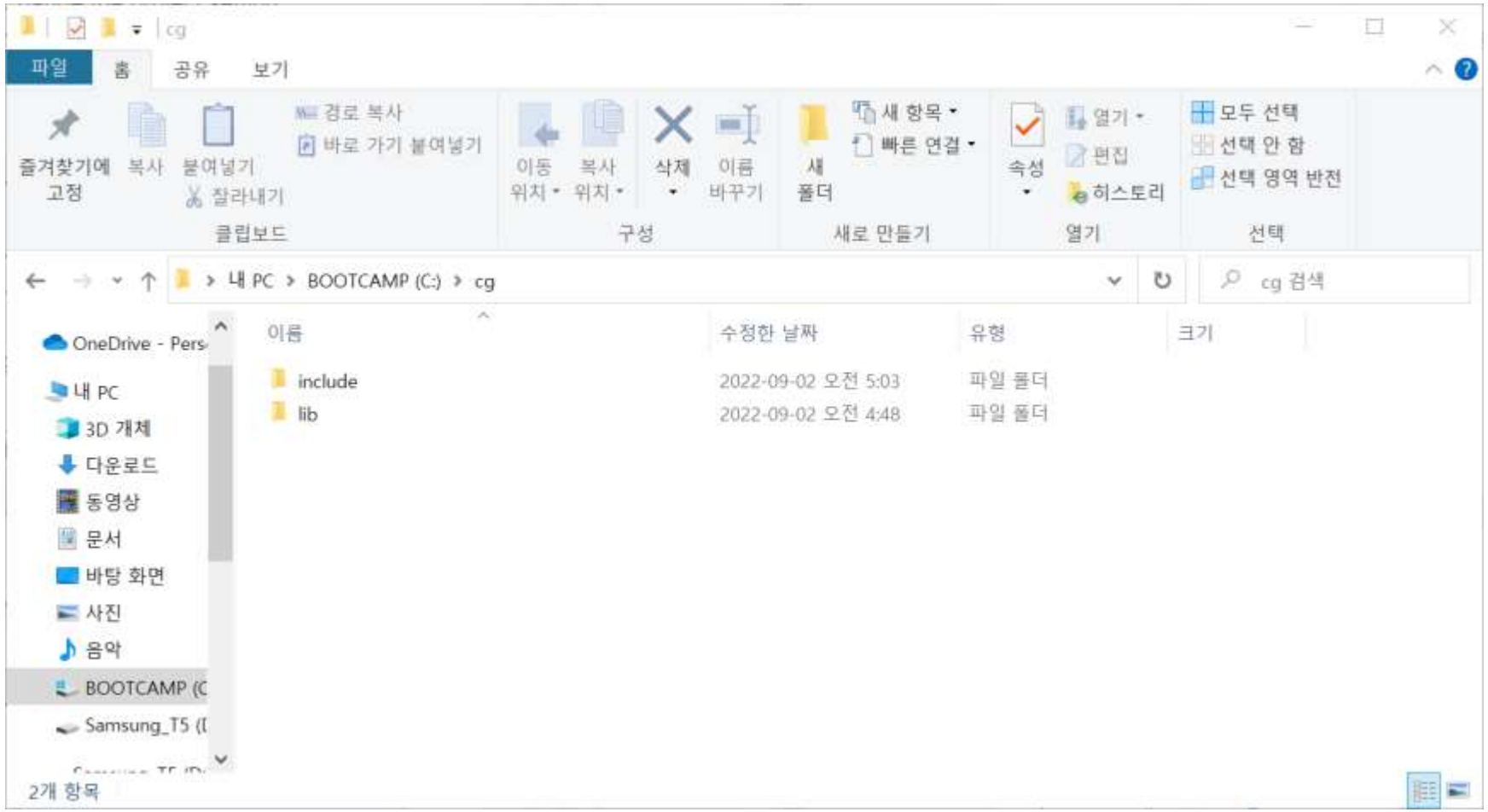
Create a New Project in VS

- Visual Studio 첫 화면에서 새 프로젝트 만들기
  - 빈 프로젝트 선택



OpenGL libraries

- Unzip Gllibs.zip to c:\cg\



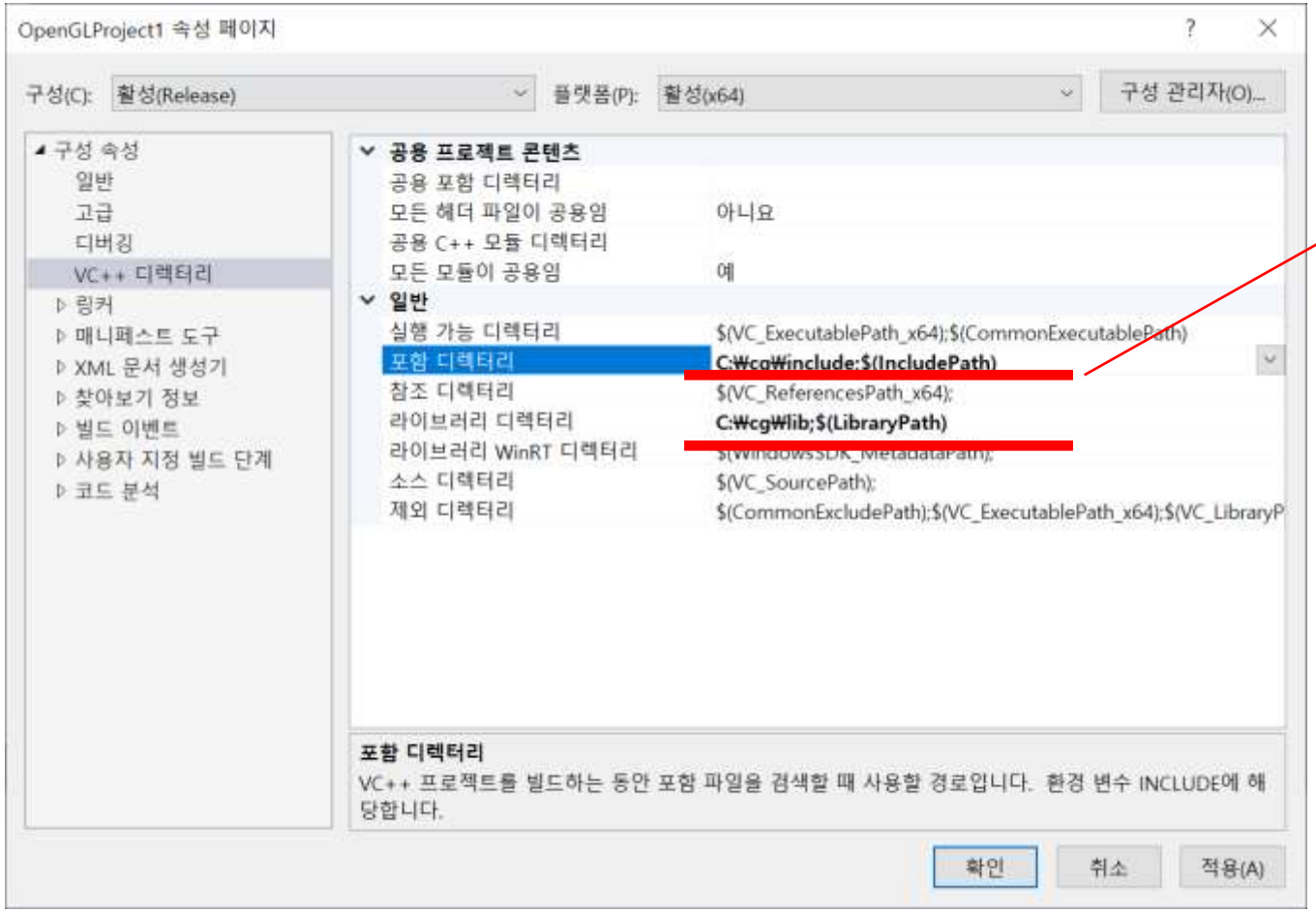
Create a New Project in VS

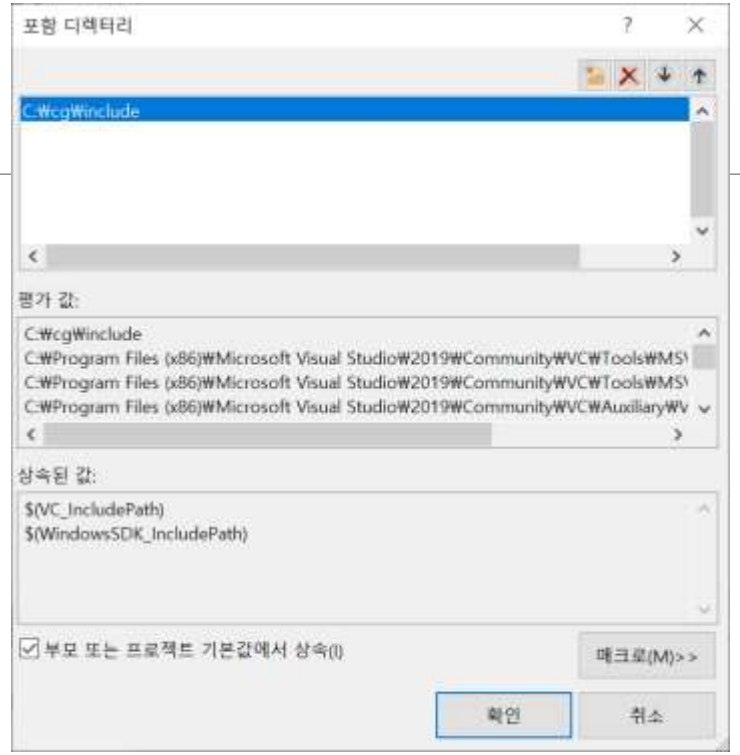
새 프로젝트 구성

- 프로젝트 이름, 위치 입력
- 솔루션 및 프로젝트를 같은 디렉토리에 배치 선택

Project Setting

Set include/lib directories

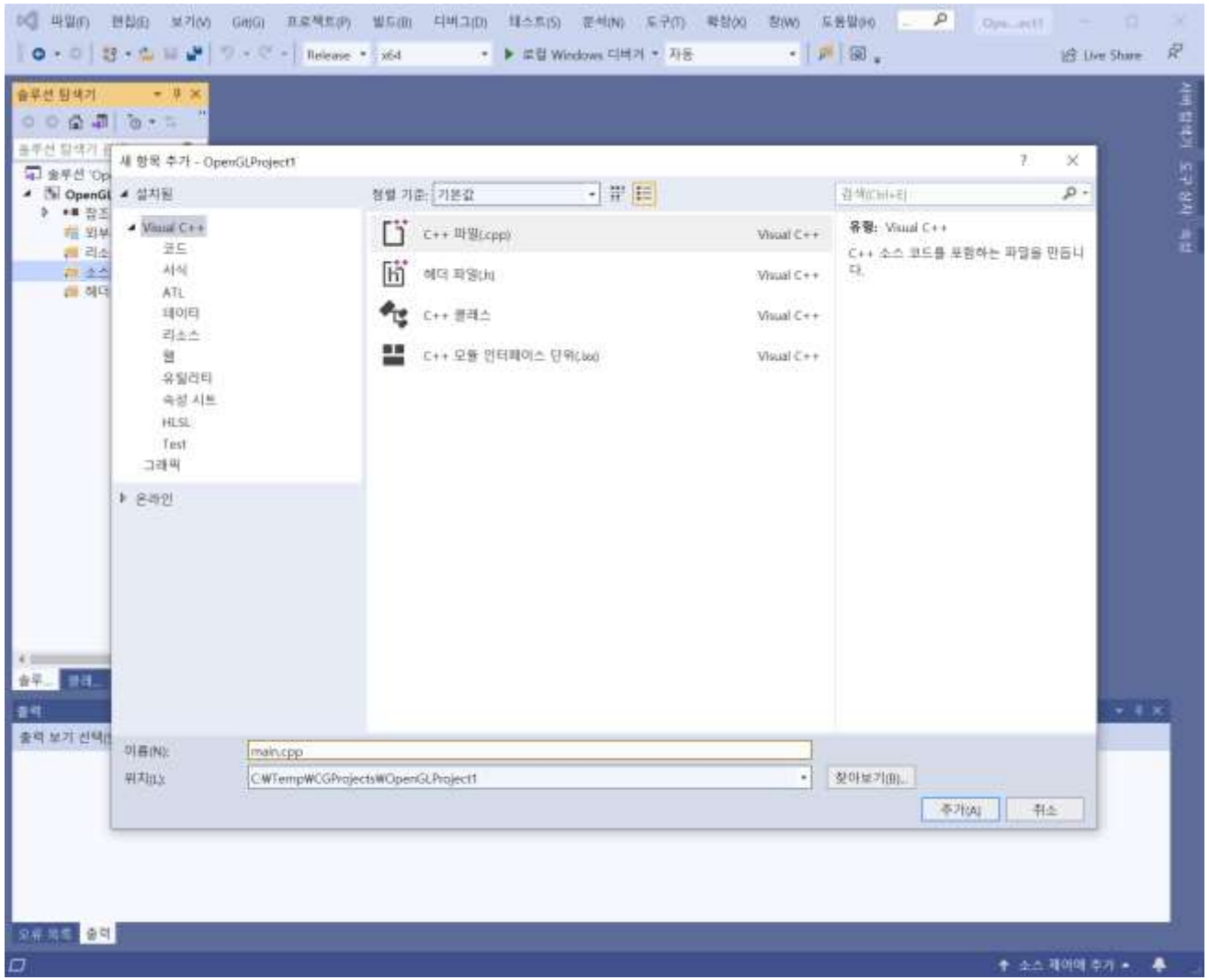




Create a new main.cpp file

소스파일 => 추가 => 새항목

- C++ 파일 추가



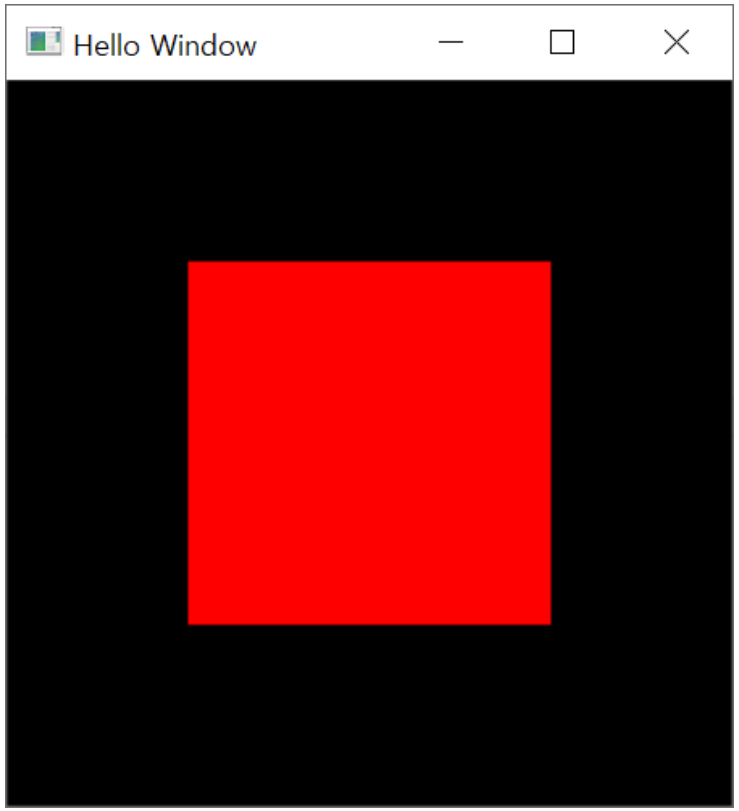
Create a new main.cpp file

And add the following line at the beginning of the code:

```
#include <vgl.h>
```

Let’s start with a legacy OpenGL 1.0 code

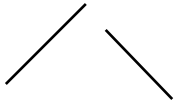
```
#include <vgl.h>

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_POLYGON);
    glVertex2f(-0.5, -0.5);
    glVertex2f(0.5, -0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(-0.5, 0.5);
    glEnd();
    glFlush();
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutCreateWindow("Hello Window");

    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```



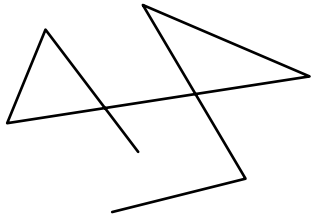
OpenGL Geometric Primitives

All primitives are specified by vertices

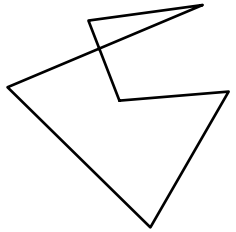
GL_POINTS



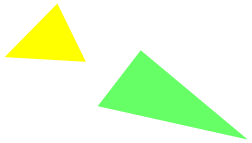
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP

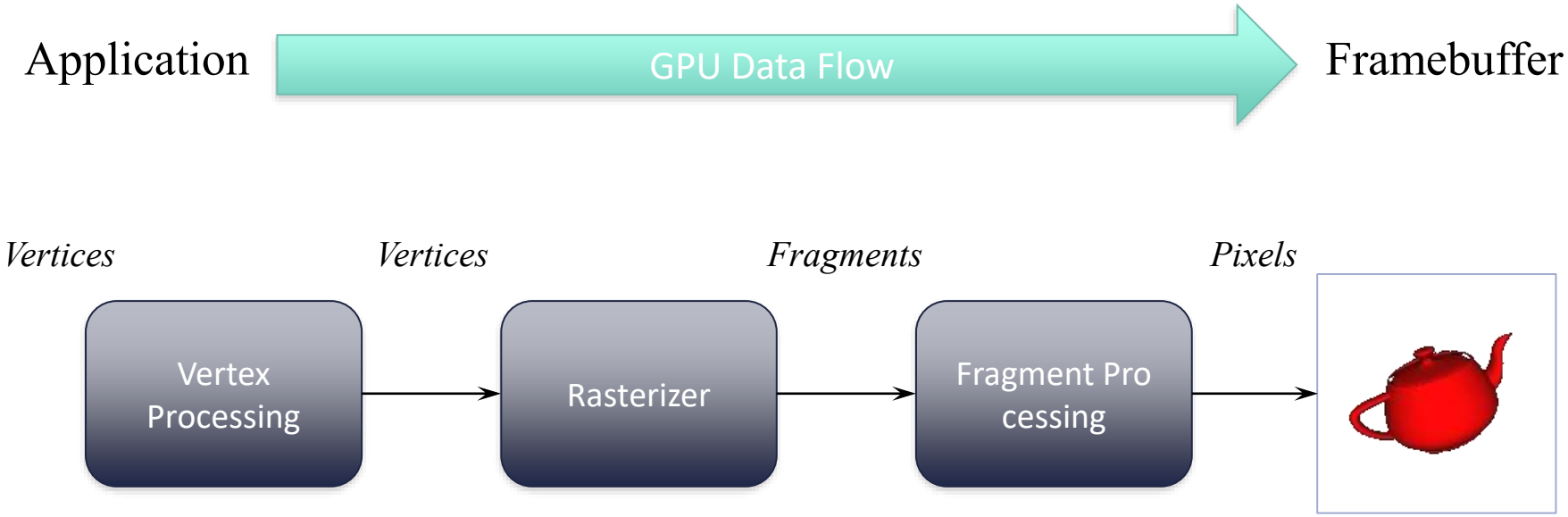


GL_TRIANGLES

GL_TRIANGLE_STRIP

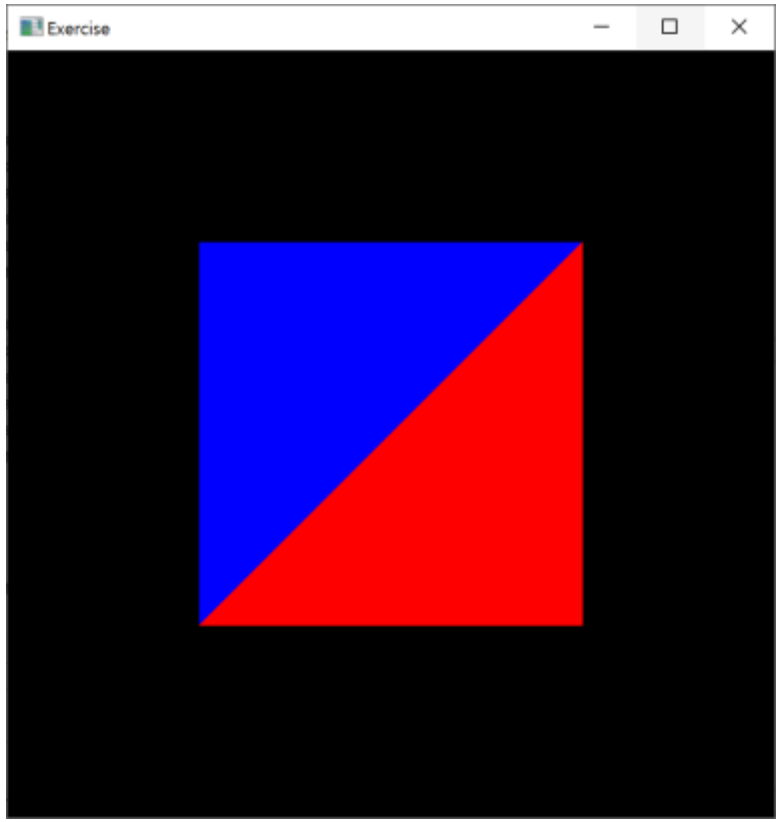
GL_TRIANGLE_FAN

OpenGL Pipeline (Simplified)



Exercise

- Write a program to display the following window. The title of the window is ‘your name_student id’



❖ Hint: You can set the initial size of the window using the following function:

```
void glutInitWindowSize(int width, int height);
```

Programming with OpenGL in a modern way

Setting OpenGL Version

```
int main(int argc, char ** argv)
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE|GLUT_RGBA);
	glutInitWindowSize(512,512);

	glutInitContextVersion(4,3);
	glutInitContextProfile(GLUT_CORE_PROFILE);

	glutCreateWindow("Many Points GPU");

	glewExperimental = true;
	glewInit();

	glutDisplayFunc(display);
	glutMainLoop();

	return 0;
}
```

Setting for the most current OpenGL version for your computer

```
int main(int argc, char ** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGBA);
        glutInitWindowSize(512,512);
        glutCreateWindow("Many Points GPU");

        glewExperimental = true;
        glewInit();

        printf("OpenGL %s, GLSL %s\n",
            glGetString(GL_VERSION),
            glGetString(GL_SHADING_LANGUAGE_VERSION));

        glutDisplayFunc(display);
        glutMainLoop();

        return 0;
}
```

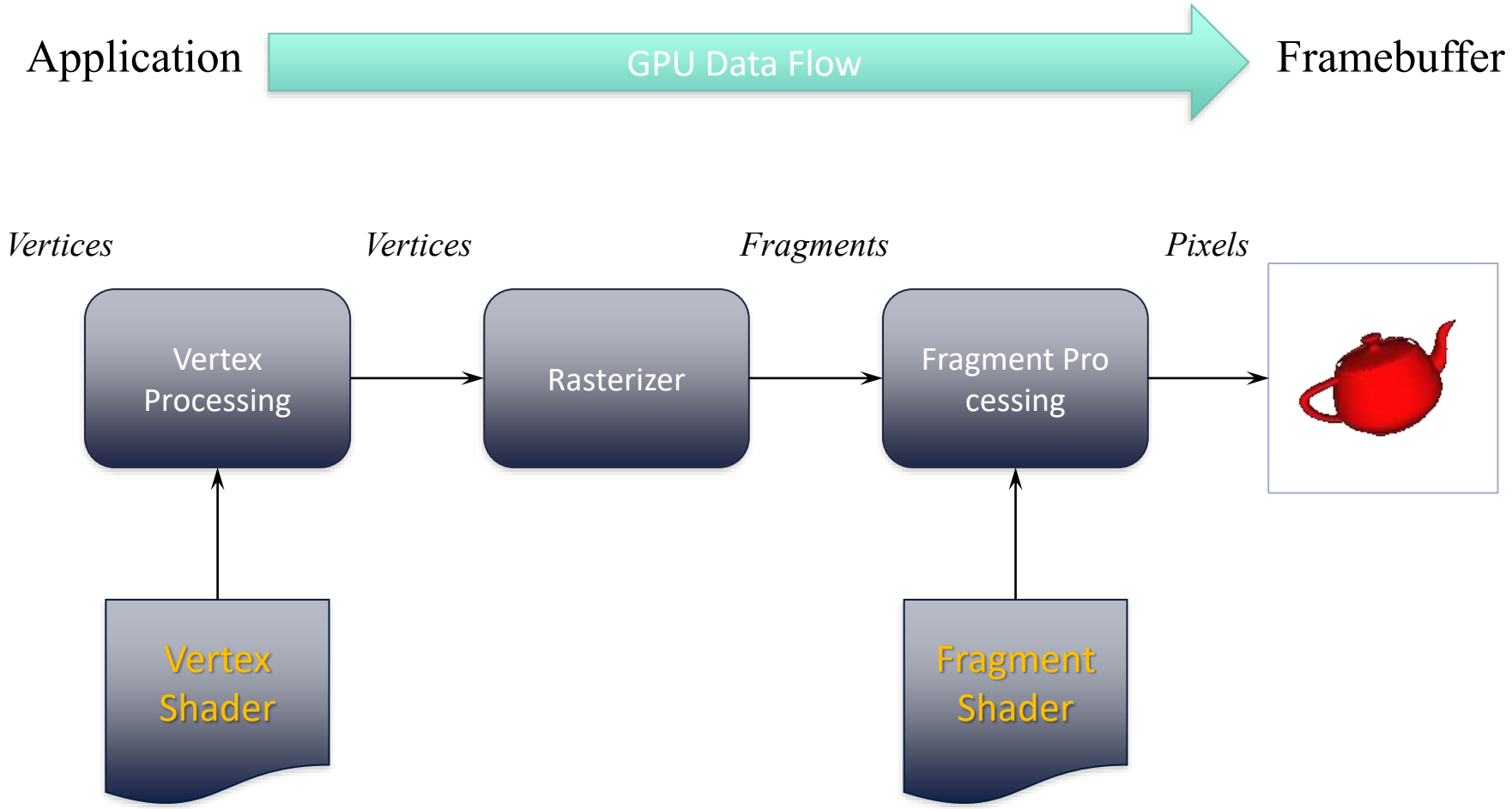
For using the modern OpenGL

To check the Current version

Modern OpenGL Programming

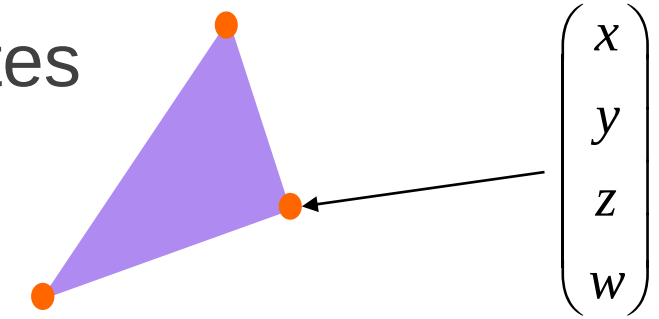
1. Create buffer objects and load data
2. Create shader programs
3. Connect data locations with shader variables
4. Render

OpenGL Pipeline (Simplified)



Representing Geometric Objects

- Geometric objects are represented using vertices
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space

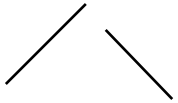


- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in *vertex buffer objects* (***VBO***s)
- VBOs must be stored in *vertex array objects* (***VAO***s)

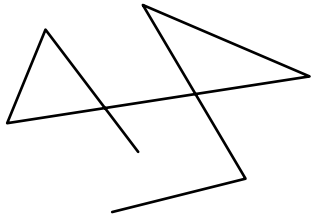
OpenGL Geometric Primitives

All primitives are specified by vertices

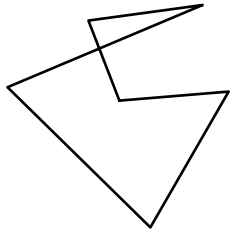
GL_POINTS



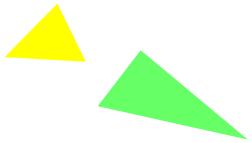
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

Create Data

Define an array for storing all points at once

```
struct vec2
{
	float x;
	float y;
};

const int NumPoints = 5000;

void init()
{
	vec2 points[NumPoints];

	for ( int i = 0; i < NumPoints; i++ )
	{
		points[i].x = (rand()%200)/100.0f-1.0f;
		points[i].y = (rand()%200)/100.0f-1.0f;
    }
}
```

Draw the array at once

Define an array for storing all the points

```
void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        glDrawArrays(GL_POINTS, 0, NumPoints);
}
```

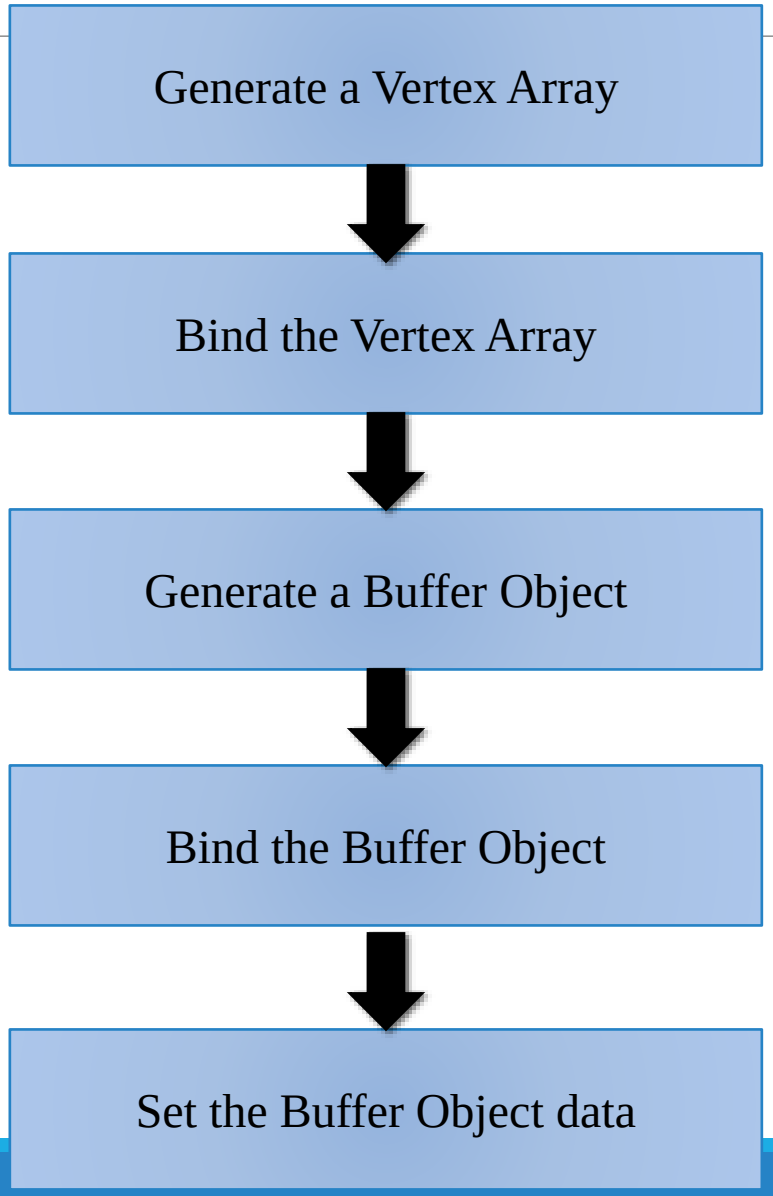
Above code draws the data in GPU.
But we didn’t send the data to GPU at all!!

How to send data

- Vertex data must be stored in *vertex buffer objects* (***VBO***s)

- VBOs must be stored in *vertex array objects* (***VAO***s)

How to send data




```
glGenVertexArray(..)
glBindVertexArray(..)
glGenBuffers(..)
glBindBuffer(..)
glBufferData(..)
```

Vertex Array Objects (VAOs)

- VAOs store the data of a geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current

VAOs in Code

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

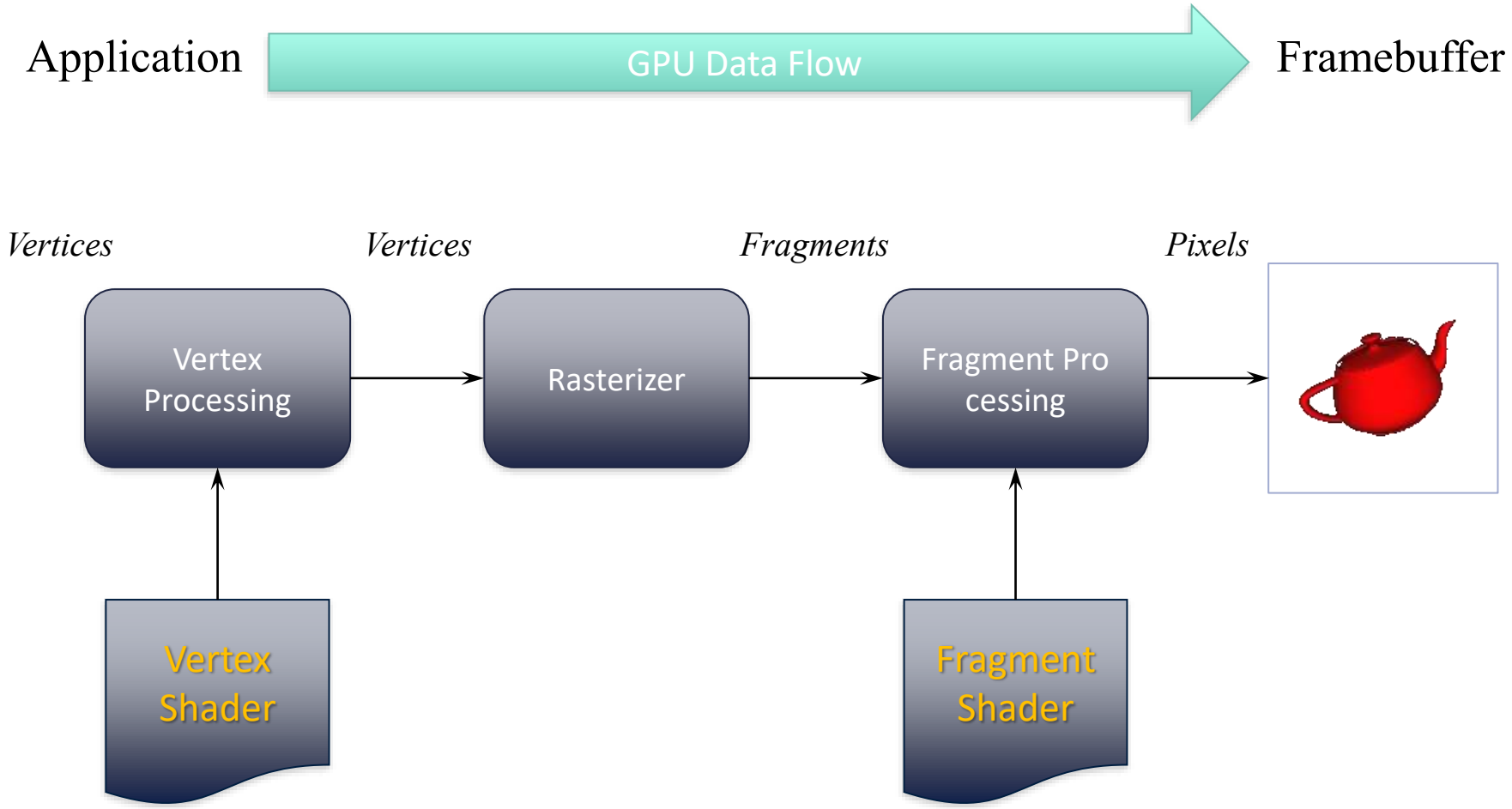
Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, …)`
  - load data into VBO using `glBufferData(GL_ARRAY_BUFFER, …)`
  - bind VAO for use in rendering later `glBindVertexArray()`

VBOs in Code

```
// Create and initialize a buffer object

GLuint buffer;

glGenBuffers(1, &buffer);

glBindBuffer(GL_ARRAY_BUFFER, buffer);

glBufferData(GL_ARRAY_BUFFER, sizeof(points),
             points, GL_STATIC_DRAW);
```

We need shaders!



Loading Shaders

```
#include <InitShader.h>
```

```
// Load and use shaders
GLuint program
      = InitShader( "vshader.glsl", "fshader.glsl" );
glUseProgram( program );
```

glsl : opengl shader language

Connecting Vertex Shaders with Geometry

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

```
// set up vertex arrays (after shaders are loaded)

int vPosition = 0;

glEnableVertexAttribArray( vPosition );

glVertexAttribPointer( vPosition, 2, GL_FLOAT,
                       GL_FALSE, 0, BUFFER_OFFSET(0) );
```

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays(GL_POINTS, 0, NumPoints);
```

- Usually invoked in display callback
- Initiates vertex shader

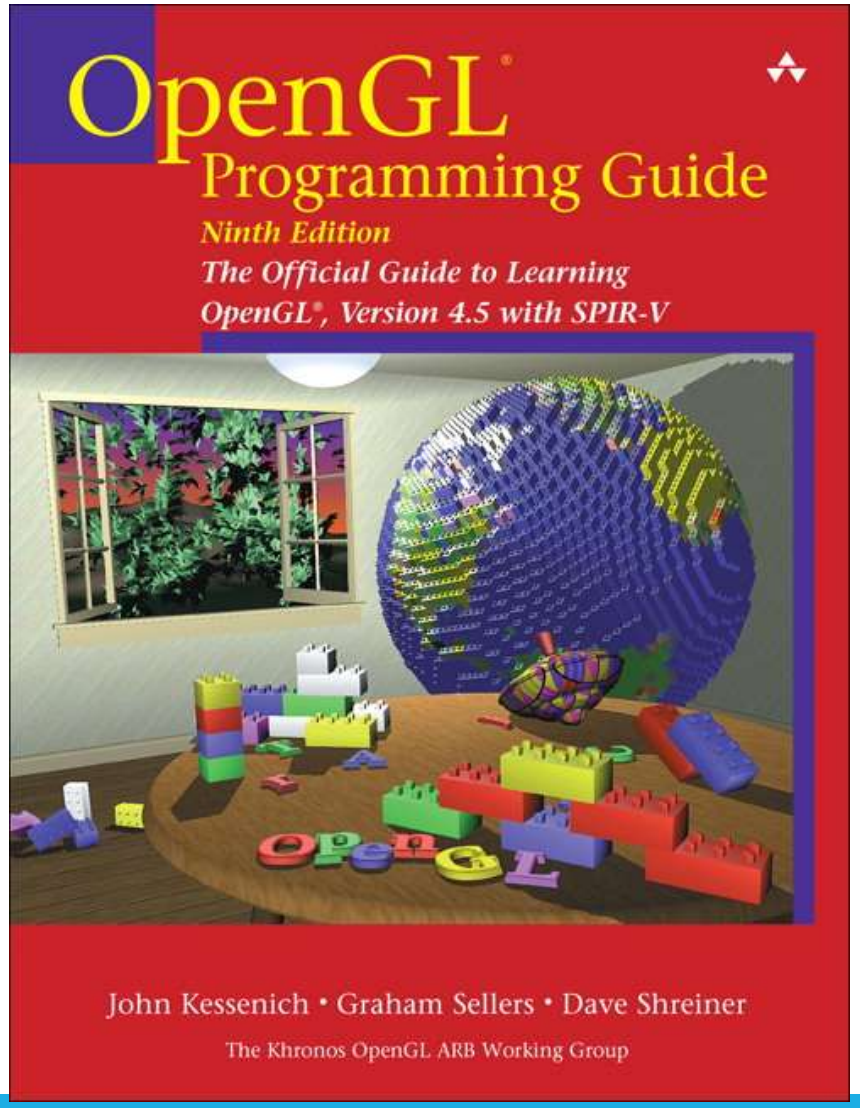
Summary

- Setting for libraries
  - Set include/lib folder
  - `#include <vgl.h>`
  - `#include <initshader.h>`
- Creating data (in an array form)
- Sending the data
  - VAO – vertex array object
  - VBO – vertex buffer object
- Loading the shaders
- Draw it with `glDrawArrays(…)`

References

OpenGL Programming Guide 9th edition ("Red Book") – everything about OpenGL

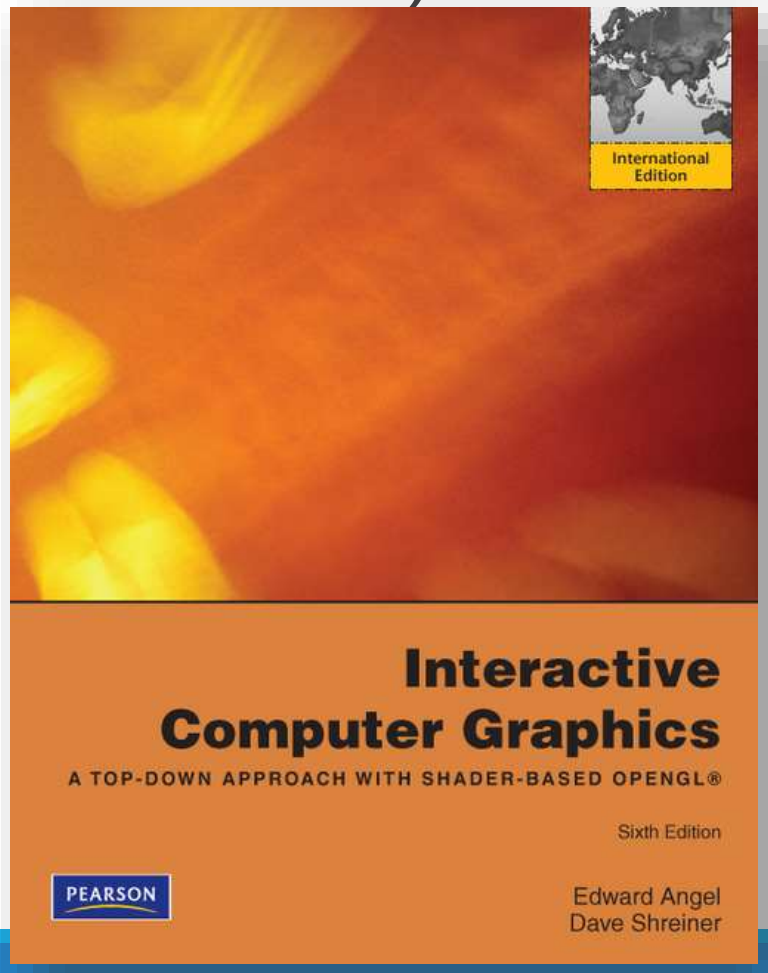
(http://www.opengl-redbook.com)



References

Interactive Computer Graphics – 6th edition
(A top-down approach with shader-based OPENGL)

by E. Angel and D. Shreiner



References

https://learnopengl.com/