Computer Graphics

by Ruen-Rone Lee
ICL/ITRI

Wrap up from last Class

- 3D Graphics Pipeline
- Shaders
  - Vertex Shader
  - Pixel Shader
  - Geometry Shader

OpenGL at a Glance

What is OpenGL

OpenGL Processing Pipeline

OpenGL Shaders

OpenGL Shading Language

OpenGL Initialization and Toolkits

A Simple OpenGL Framework with GLUT

OpenGL Reference Links

- Official OpenGL web page
  - http://www.opengl.org/
- Khronos OpenGL web page
  - http://www.khronos.org/opengl/
- Khronos OpenGL wiki page
  - https://www.khronos.org/opengl/wiki/Main_Page

What is OpenGL

Standard Graphics APIs

What is OpenGL

- OpenGL (Open Graphics Library) is an open standard for cross-language, cross-platform API specification
- OpenGL is not a programming language
- OpenGL is a set of APIs (Application Programming Interface) that is used to write 2D/3D graphics applications
- OpenGL defines the function specification of each API and leaves the implementation to the vendors themselves

OpenGL Evolution

Source: Wikipedia

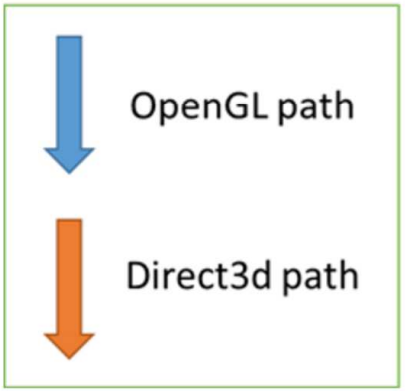
- **Fixed Function Pipeline (1992~2003)**
  - OpenGL 1.1 - Texture objects
  - OpenGL 1.2 - 3D textures, BGRA and packed pixel formats
  - OpenGL 1.3 - Multitexturing, multisampling, texture compression
  - OpenGL 1.4 - Depth textures
  - OpenGL 1.5 - Vertex Buffer Object (VBO), Occlusion Queries
- **Programmable Pipeline (2004~present)**
  - OpenGL 2.0 - GLSL 1.1, MRT, Non Power of Two textures, Point Sprites, Two-sided stencil
  - OpenGL 2.1 - GLSL 1.2, Pixel Buffer Object (PBO), sRGB Textures
  - OpenGL 3.0 - GLSL 1.3, Texture Arrays, Conditional rendering, Frame Buffer Object (FBO)
  - OpenGL 3.1 - GLSL 1.4, Instancing, Texture Buffer Object, Uniform Buffer Object, Primitive restart
  - OpenGL 3.2 - GLSL 1.5, Geometry Shader, Multi-sampled textures
  - OpenGL 3.3 - GLSL 3.30 Backports as much function as possible from the OpenGL 4.0 specification
  - OpenGL 4.0 - GLSL 4.00 Tessellation on GPU, shaders with 64-bit precision,
  - OpenGL 4.1 - GLSL 4.10 Developer-friendly debug outputs, compatibility with OpenGL ES 2.0,
  - OpenGL 4.2 - GLSL 4.20 Shaders with atomic counters, draw transform feedback instanced, shader packing, performance improvements
  - OpenGL 4.3 - GLSL 4.30 Compute shaders leveraging GPU parallelism, shader storage buffer objects, high-quality ETC2/EAC texture compression, increased memory security, a multi-application robustness extension, compatibility with OpenGL ES 3.0,
  - OpenGL 4.4 - GLSL 4.40 Buffer Placement Control, Efficient Asynchronous Queries, Shader Variable Layout, Efficient Multiple Object Binding, Streamlined Porting of Direct3D applications, Bindless Texture Extension, Sparse Texture Extension,
  - OpenGL 4.5 - GLSL 4.50 Direct State Access (DSA), Flush Control, Robustness, OpenGL ES 3.1 API and shader compatibility, DX11 emulation features
  - OpenGL 4.6 - GLSL 4.60 More efficient geometry processing and shader execution, more information, no error context, polygon offset clamp, SPIR-V, anisotropic filtering

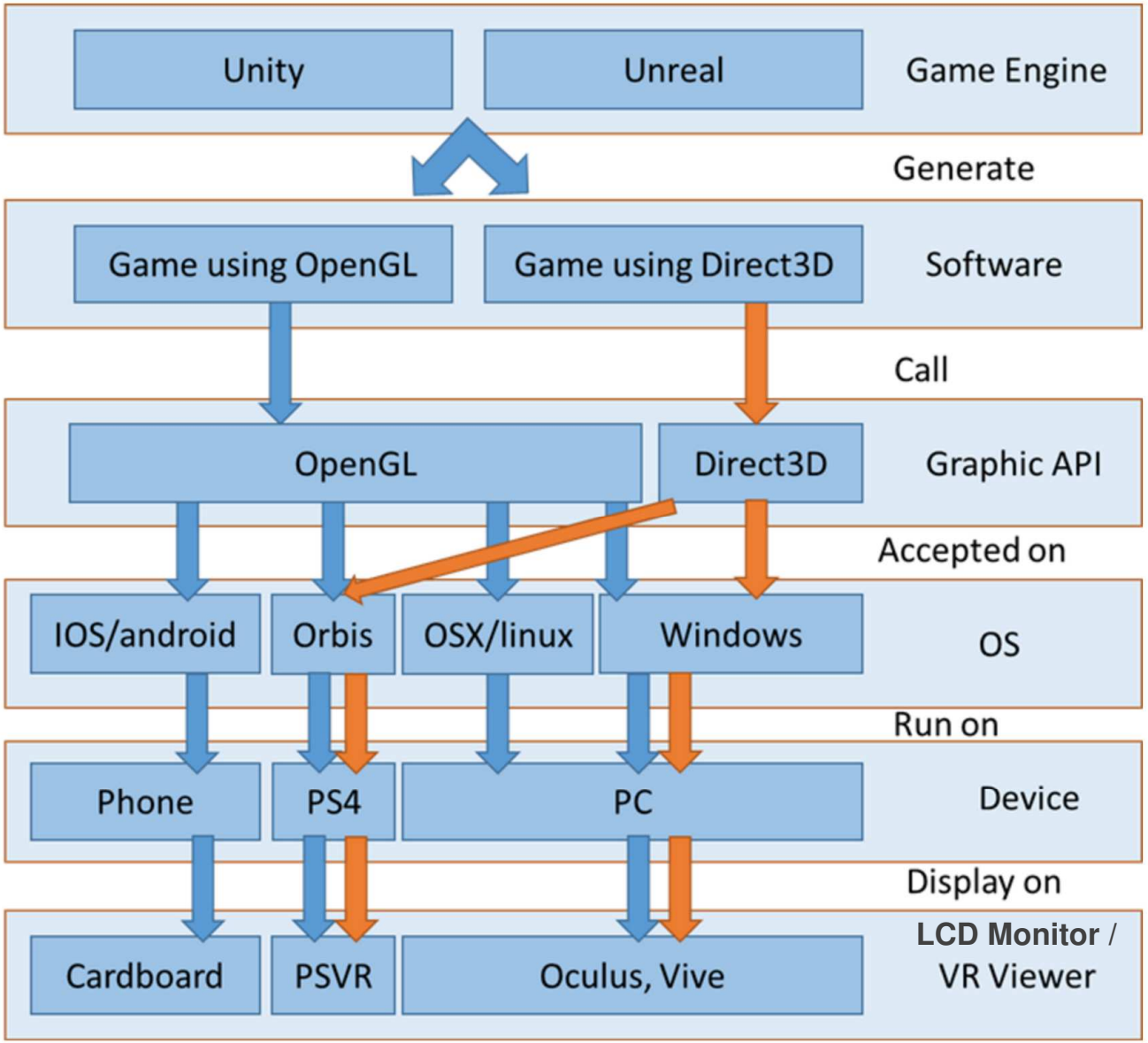
Released on July 31, 2017

Other Graphics APIs

- Direct3D
  - Proprietary Microsoft Windows 3D graphics API
- Vulkan
  - New cross-platform 3D graphics and compute API by Khronos group
- OpenGL ES
  - OpenGL for Embedded Systems
- Web-based OpenGL
  - JavaScript interface for OpenGL-ES-2.x API
- Metal, Mantle, …

OpenGL vs. Direct3D





Source: “OpenGL vs. Direct3D – Who is the Winner of Graphics API” by Hunter Lin, 2016.

Capability of OpenGL

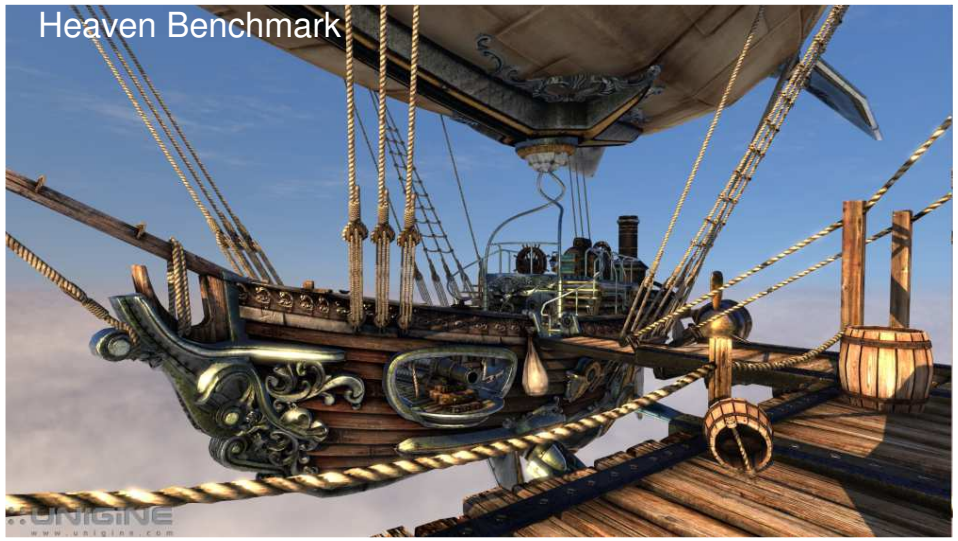
- **Flight Simulator**

Flight Gear

X-Plane

Capability of OpenGL

- **Benchmarks**
  - **Unigine's Benchmarks**



OpenGL Extension

- A mechanism to provide additional features which are not yet adopted by the OpenGL specification officially
  - New functions
  - New constants
  - Relax or remove restrictions on existing OpenGL functions

OpenGL Extension

- Advantages
  - Develop new functionality before new API spec is released
  - Hardware vendors can expose their new hardware features via extension first
  - Extension becomes core function (or extension) after being approved by the ARB (Architecture Review Board)

OpenGL Extension

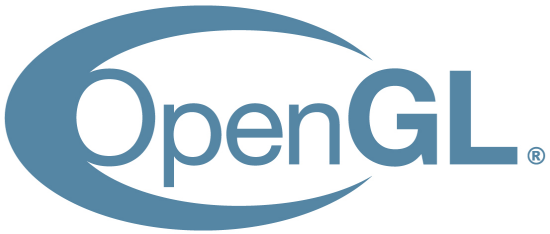
- Disadvantages
  - It is vendor specific before the extension becomes an ARB extension or core API
  - You are recommended to query the existence of a specific extension before you use it
  - Compatibility might be an issue if an application was using a vendor specific extension

- GLEW/GLAD can help in querying and loading OpenGL extensions

Why OpenGL

- Cross-platform
  - Windows, Mac OSX, Linux, …
- Better backward compatibility
- Run on various hardware platforms (OpenGL/OpenGL ES/WebGL)
  - OpenGL for desktop PC/NB
  - OpenGL ES for embedded systems such as tablet, phone, or game console (iOS/Android/…)
  - WebGL for various browsers (PC/NB/Mobile)

OpenGL-based Graphics API



Desktop PC, Laptop NB
(Windows/OSX/Linux)

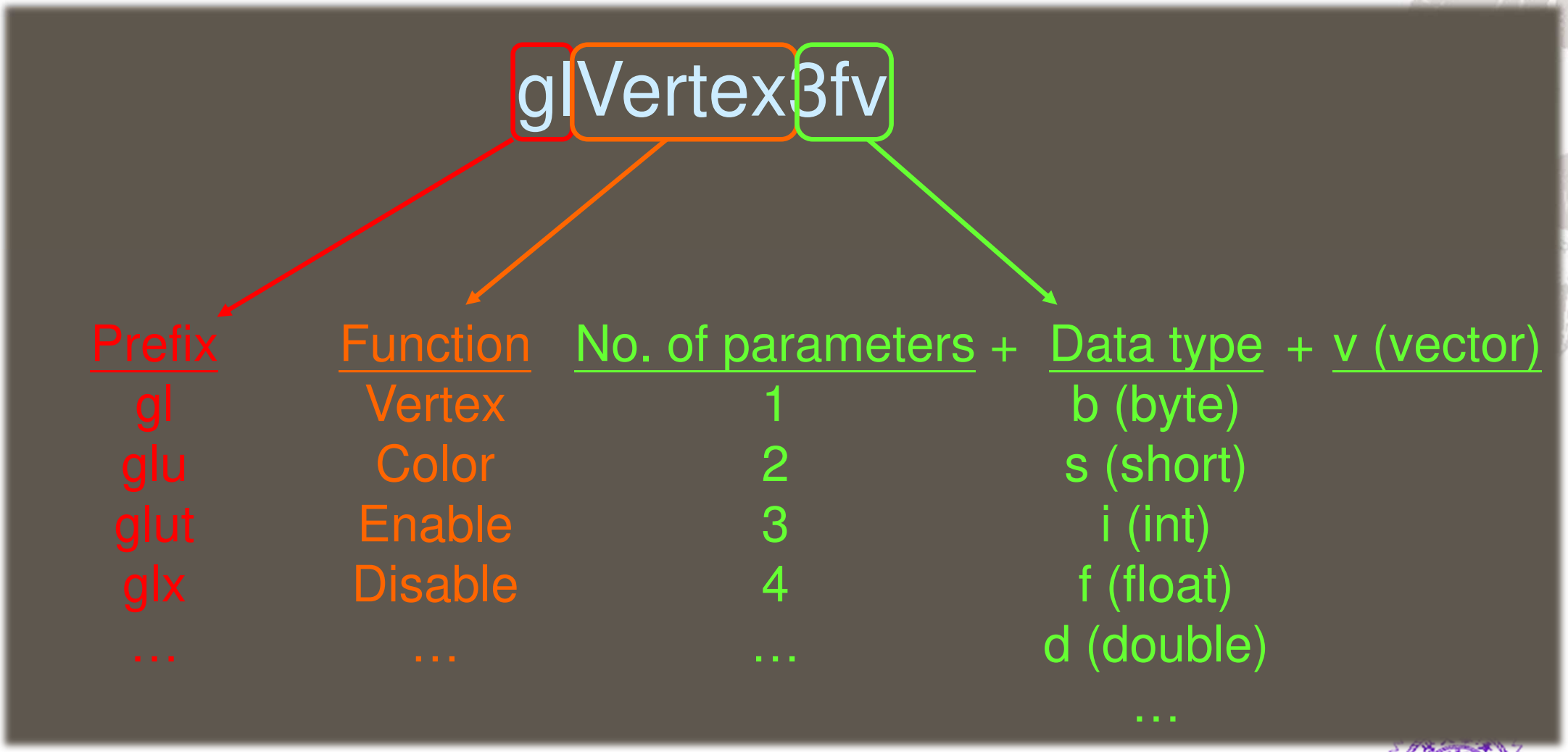
OpenGL 1.3 ⇨ OpenGL ES 1.0
OpenGL 2.0 ⇨ OpenGL ES 2.0
OpenGL 4.3 ⇨ OpenGL ES 3.0

Tablet, Smart Phone, Wearable Device, Game Console
(Android/iOS/Orbis)

OpenGL ES 2.0 ⇨ WebGL 1.0
OpenGL ES 3.0 ⇨ WebGL 2.0

Browsers on PC/Mobile Device/Embedded Device
(Google Chrome/Internet Explorer/Safari/Mozilla Firefox/Opera)

Convention of an OpenGL API



OpenGL Data Types

Suffix	Data type	Typical C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	Int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

OpenGL Data Types

- **In consideration of portability, OpenGL defined data types should be used throughout the application**
  - **Be careful when you mixed the usage of C++ data types and OpenGL data types**

Data Model	short	int	long	long long	pointer
LP64	16	32	64	64	64
LLP64	16	32	32	64	64

Many 64-bit compilers today use the LP64 model (including Solaris, AIX, HP, Linux, Mac OS X, and IBM z/OS native compilers). Microsoft's VC++ compiler uses the LLP64 model.

long ≠ GLint in Linux and Mac OS!!
Also, "long long = GLint64" and "pointer = GLintptr" in OpenGL data types

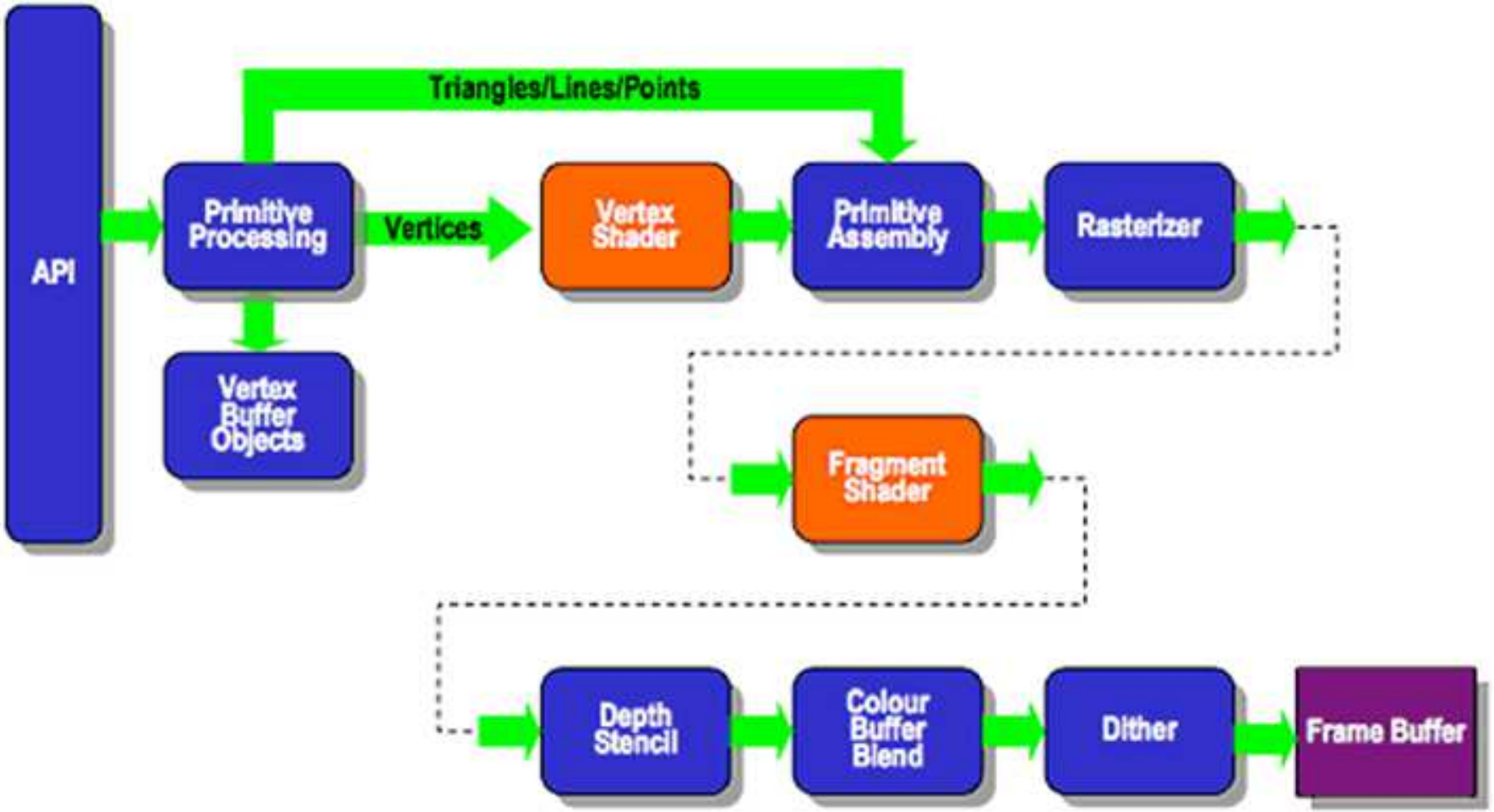
OpenGL Pipeline

Fixed Function Pipeline

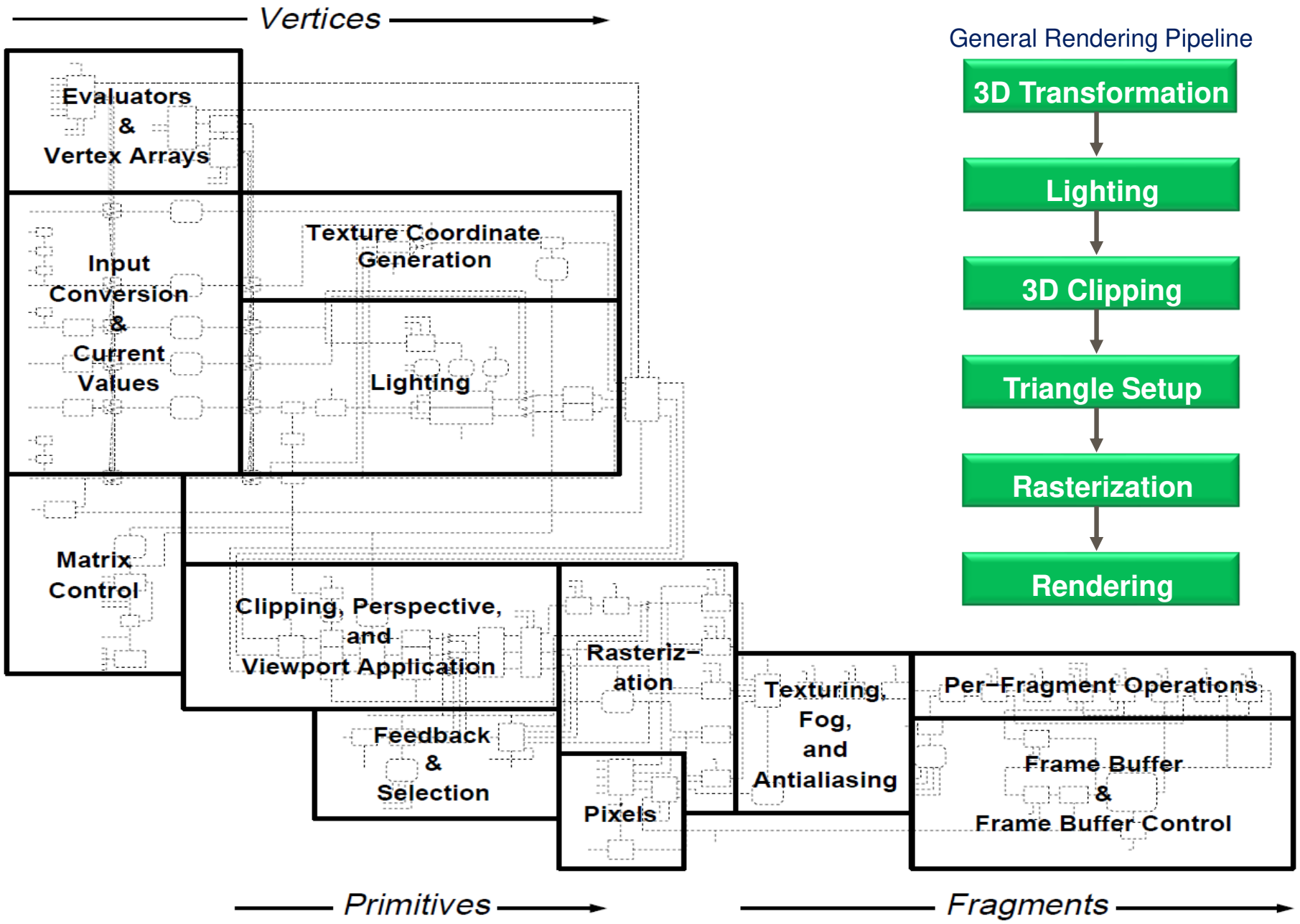
Programmable Pipeline

OpenGL Pipeline

- Difference between fixed and programmable



OpenGL Pipeline (Fixed Function)

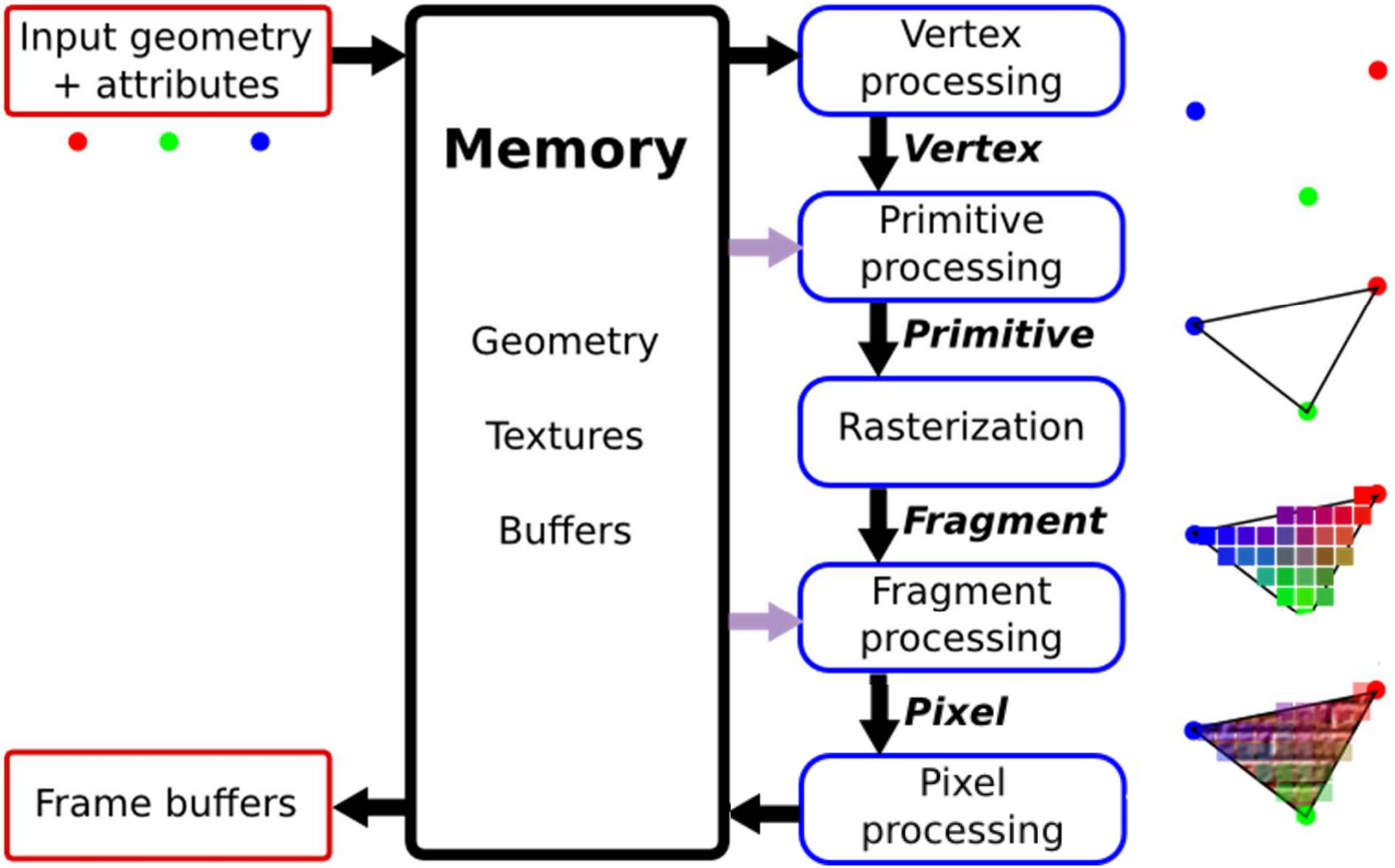


General Rendering Pipeline

- 3D Transformation
- Lighting
- 3D Clipping
- Triangle Setup
- Rasterization
- Rendering

OpenGL Pipeline (Fixed Function)

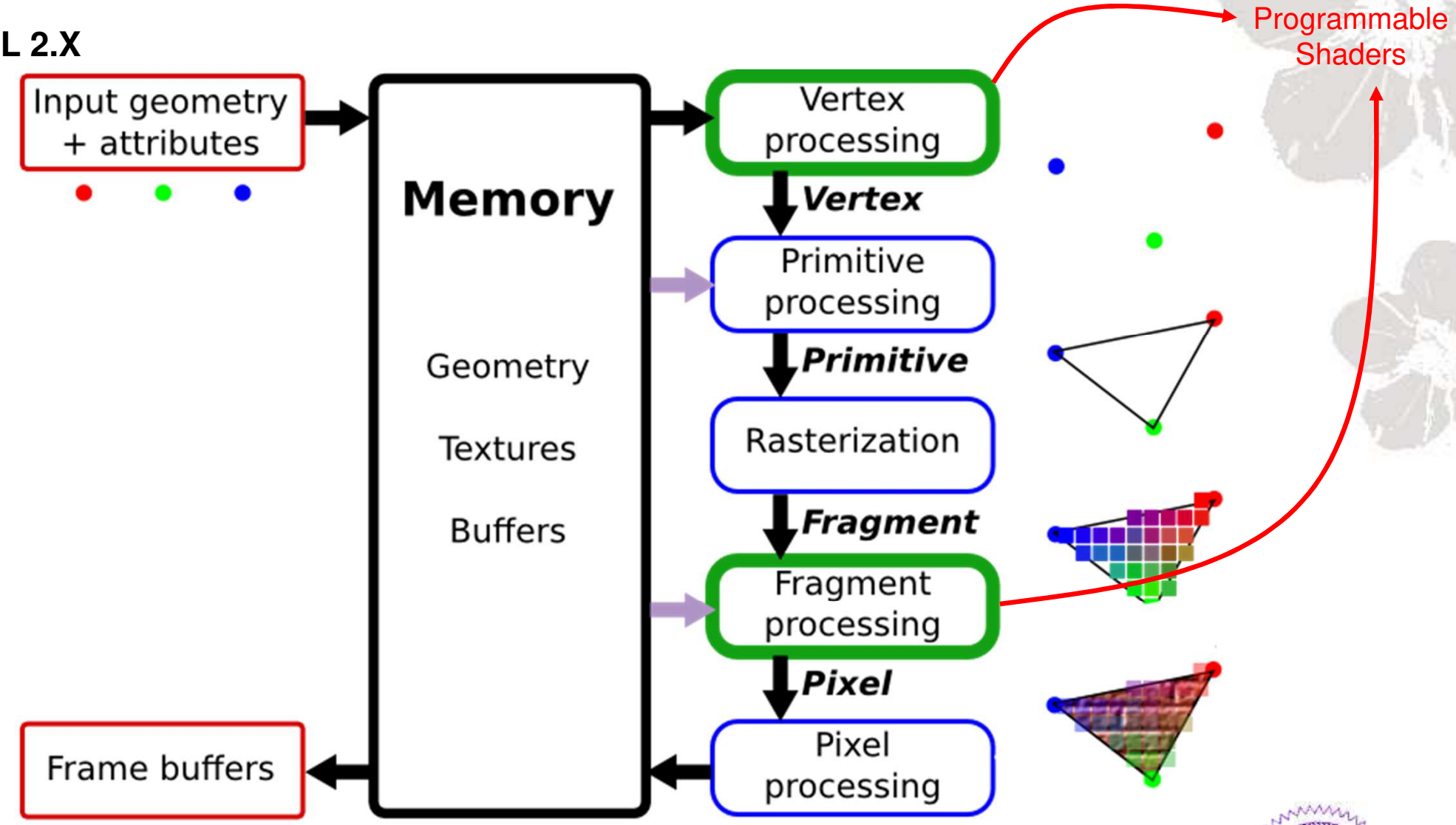
OpenGL 1.X



Source: by Romain Vergne, http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

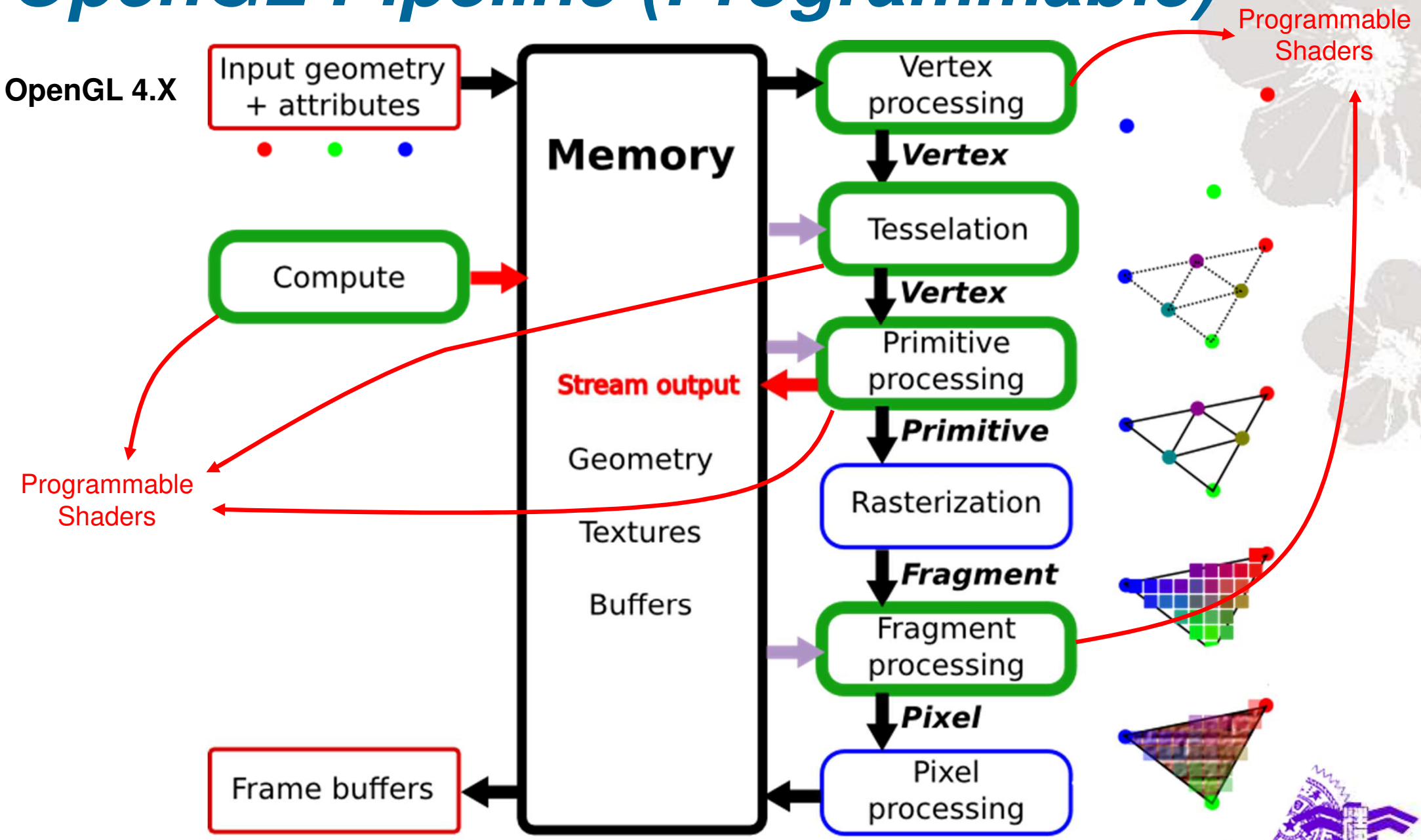
OpenGL Pipeline (Programmable)

OpenGL 2.X



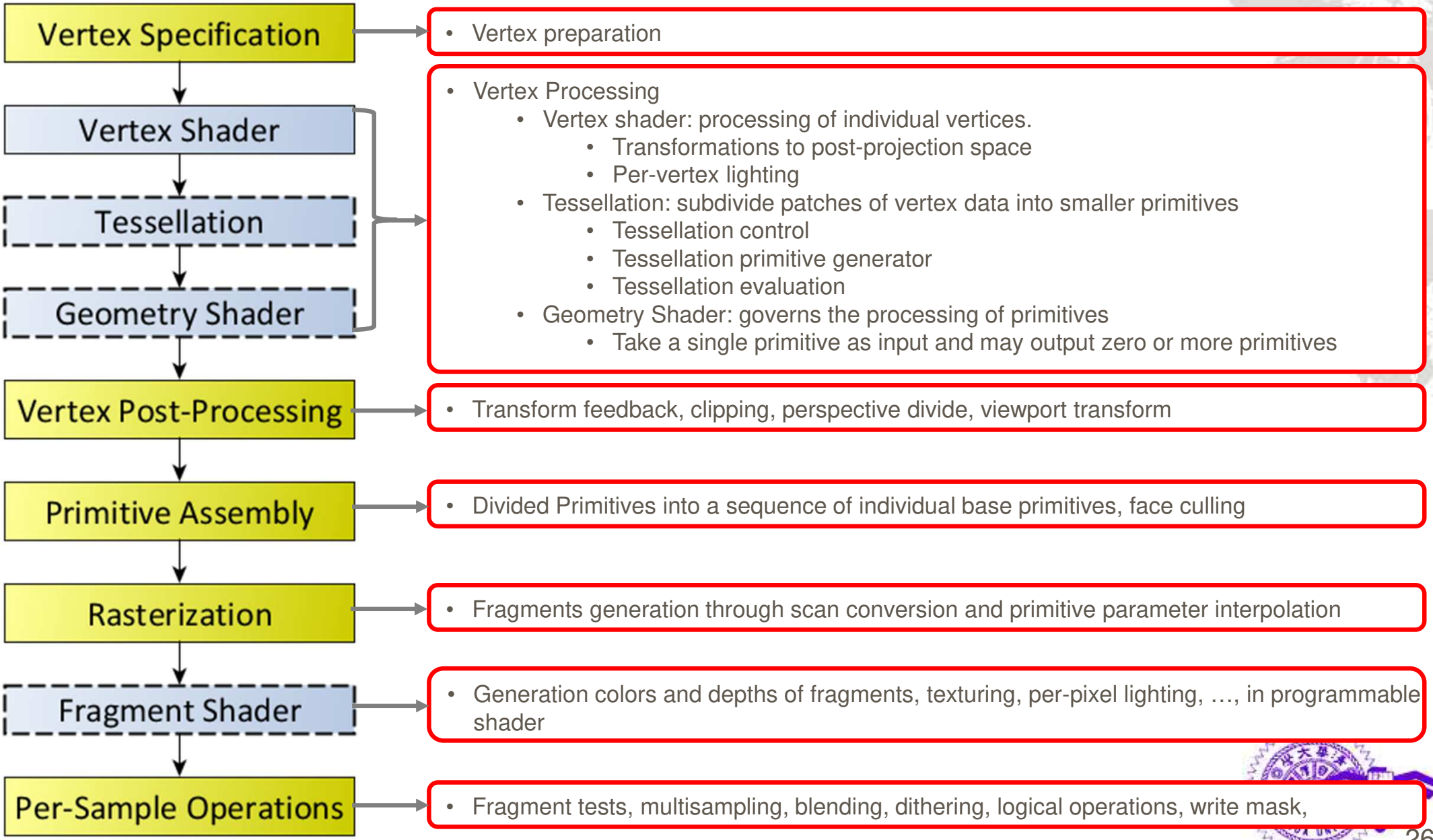
Source: by Romain Vergne, http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

OpenGL Pipeline (Programmable)



Source: by Romain Vergne, http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

OpenGL Pipeline (Programmable)



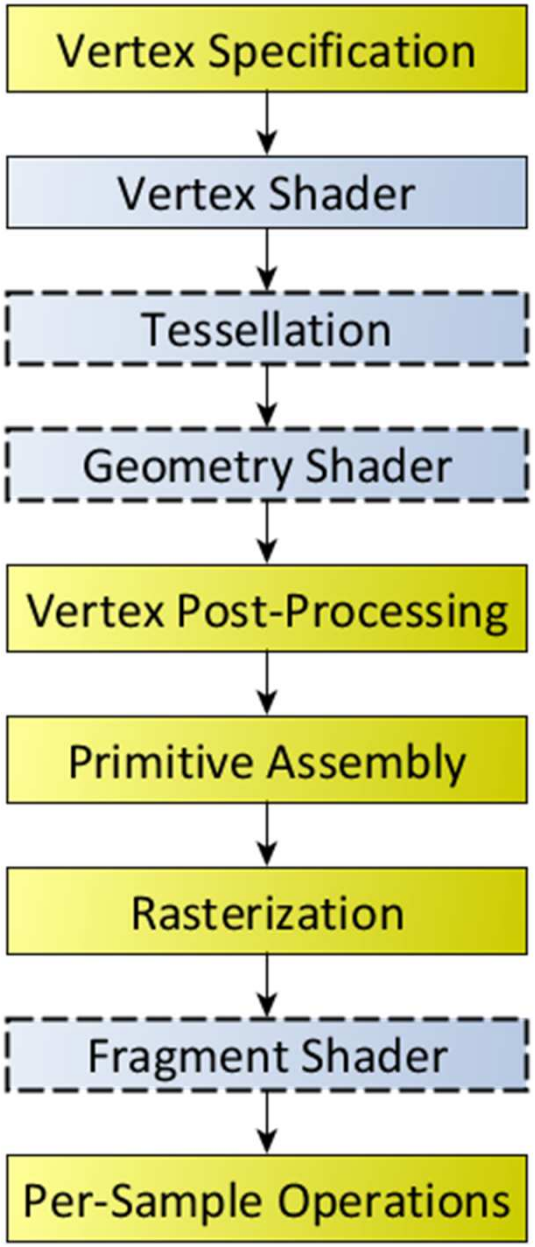
OpenGL Shaders and GLSL

Various Shaders

OpenGL Shading Language

OpenGL Shaders

- A user-defined program designed to run on some stages of a graphics piepline
  - Vertex Shader
  - Tessellation
    - Tessellation Control Shader
    - Tessellation Evaluation Shader
  - Geometry Shader
  - Fragment Shader



OpenGL Shading Language

- High-Level shading language based on C programming language
- Similar to NVIDIA's Cg (no longer support) and Microsoft's HLSL (programming shaders in DirectX)
  - DirectX + HLSL vs. OpenGL + GLSL
- Hardware vendors will provide shader compiler to optimize the shader codes for deriving best performance running on their hardware architecture

OpenGL Shading Language

GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59	2.0	April 2004	#version 110
1.20.8	2.1	September 2006	#version 120
1.30.10	3.0	August 2008	#version 130
1.40.08	3.1	March 2009	#version 140
1.50.11	3.2	August 2009	#version 150
3.30.6	3.3	February 2010	#version 330
4.00.9	4.0	March 2010	#version 400
4.10.6	4.1	July 2010	#version 410
4.20.11	4.2	August 2011	#version 420
4.30.8	4.3	August 2012	#version 430
4.40	4.4	July 2013	#version 440
4.50	4.5	August 2014	#version 450
4.60	4.6	July 2017	#version 460

A GLSL Shader Example

```
1  #version 330
2
3  layout (std140) uniform Materials {
4      vec4 diffuse;
5      vec4 ambient;
6      vec4 specular;
7      float shininess;
8  };
9
10 layout (std140) uniform Lights {
11     vec3 l_dir;    // camera space
12 };
13
14 in Data {
15     vec3 normal;
16     vec4 eye;
17 } DataIn;
18
19 out vec4 colorOut;
20
21 void main() {
22
23     // set the specular term to black
24     vec4 spec = vec4(0.0);
25
26     // normalize both input vectors
27     vec3 n = normalize(DataIn.normal);
28     vec3 e = normalize(vec3(DataIn.eye));
29
30     float intensity = max(dot(n,l_dir), 0.0);
31
32     // if the vertex is lit compute the specular color
33     if (intensity > 0.0) {
34         // compute the half vector
35         vec3 h = normalize(l_dir + e);
36         // compute the specular term into spec
37         float intSpec = max(dot(h,n), 0.0);
38         spec = specular * pow(intSpec,shininess);
39     }
40     colorOut = max(intensity * diffuse + spec, ambient);
41 }
```

Version no.

Constant variables

Input Variables

Output Variable

Main()

C-like language

- Data types
- Structure
- Assignment
- Function
- Conditional branch
- Loop
- Augmented operators
  - Vector
  - Matrix
- …

OpenGL Initialization and Toolkits

OpenGL Context/Window Creation

OpenGL Function Loader

Other OpenGL Useful Libraries

OpenGL Context/Window Creation

Platform specific

- Creation of an OpenGL **context** and an application **window** are required before rendering (i.e., before any OpenGL calls)
  - An OpenGL context stores all the states associated with the instance of OpenGL rendering
  - An application window is the window where your rendering result displayed
- Some **event handlings** also required to process different kinds of input such as keyboard and mouse

OpenGL Context/Window Creation

- A cross-platform (window system independent) toolkit for writing OpenGL programs
  - Support application frameworks to control the platform's windowing system and event handling
- Popular toolkits
  - GLUT: pretty old and no longer maintained
  - freeglut: an alternative to GLUT. Freeglut 3.2.1 released on Sep. 21, 2019 (http://freeglut.sourceforge.net/)
  - GLFW: GLFW 3.3.3 released on Feb. 23, 2021 (https://www.glfw.org/)

OpenGL Loading Libraries

Platform and vendor specific

- Default OpenGL version supported by OS might not be the right version that your graphics vendor can support
- A library to load all the pointers to OpenGL functions (core and extensions) at runtime is required in order to get the right OpenGL support
- OpenGL loader checks the graphics driver for which OpenGL version profile is supported and gets all the function pointers as well as the supported extensions

OpenGL Loading Libraries

- Some OpenGL loading libraries
  - GLEW: The OpenGL Extension Wrangler Library
    - A cross-platform open-source C/C++ extension loading library
    - Supports OpenGL 4.6 core function and conventional OpenGL, WGL and GLX extensions
    - Provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform (vendor specific)
    - Latest release: glew 2.1.0 on Jul. 31, 2017
    - http://glew.sourceforge.net/

OpenGL Loading Libraries

- Some OpenGL loading libraries
  - GLAD: Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs
    - An tool (with a web-service) for generating OpenGL, OpenGL ES, EGL, GLX and WGL headers (and loaders) based on the official XML specifications
    - You can customized to the version you like without including those deprecated or legacy functions
    - Include the header files and the loader source code to your project to use GLAD in loading the OpenGL functions
    - https://glad.dav1d.de/

Other OpenGL Useful Utilities

- Math
  - GLM: OpenGL mathematics libraries
    - A header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications
    - (https://glm.g-truc.net/0.9.9/index.html)
- Image and texture
  - stb_image.h
    - An image-loading library that supports several popular formats
    - (https://github.com/nothings/stb/blob/master/stb_image.h)

Other OpenGL Useful Utilities

- **Asset/Model loader**
  - **Open Asset Import**
    - A loader with support of variety of 3D file formats
    - (http://www.assimp.org/)
  - **TinyOBJ loader**
    - A simple wavefront obj file loader
    - (https://github.com/tinyobjloader/tinyobjloader)
  - **glm**
    - A simple wavefront obj file loader
    - (http://devernay.free.fr/hacks/glm/)
    - Not the OpenGL Mathematics Library

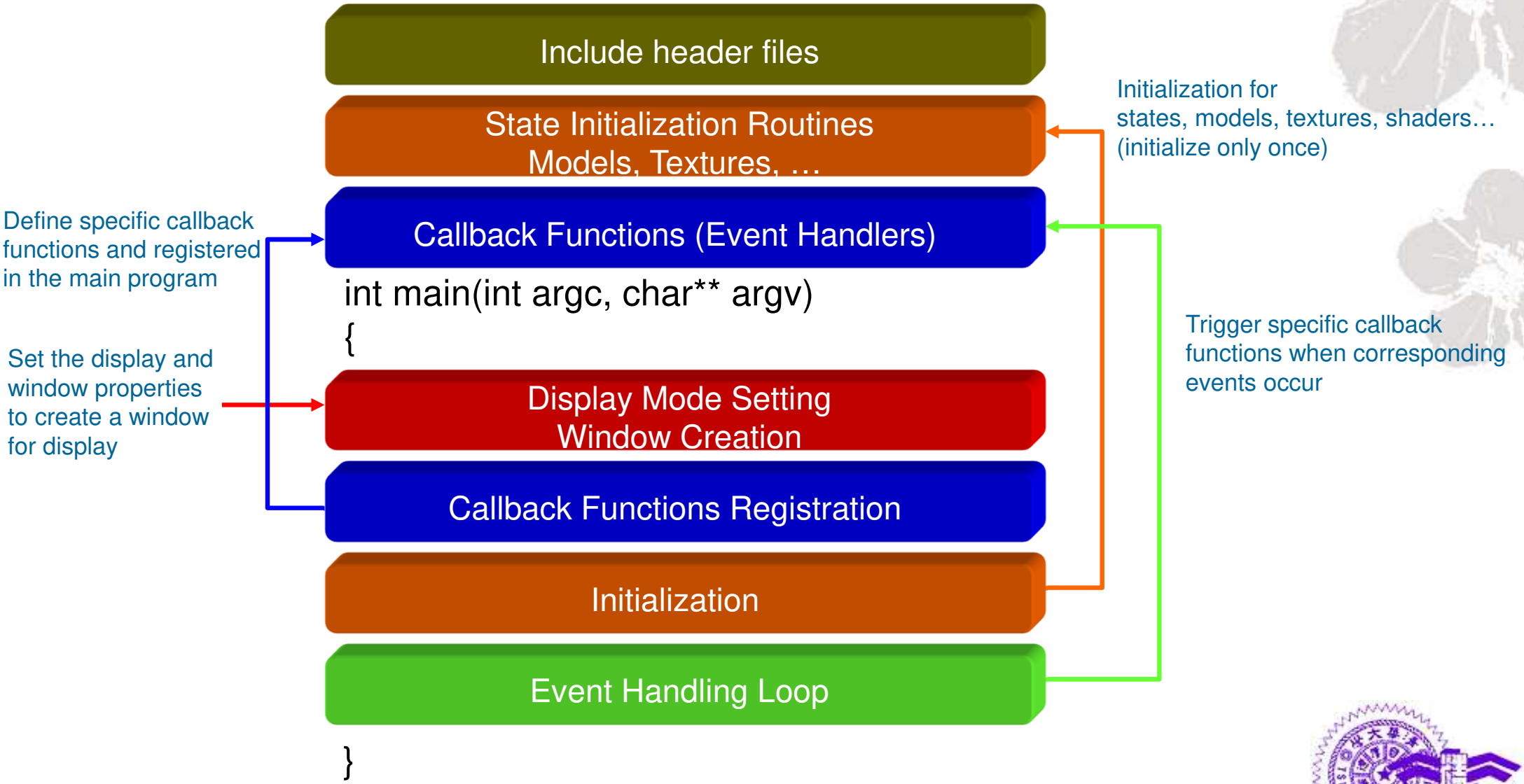
OpenGL Framework

A Simple OpenGL Framework with freeglut and glew

Illustration of an OpenGL Sample Program

Few OpenGL Programming Practices

OpenGL Application Framework



An OpenGL Example *(Using freeglut and glew)*

```
#include <GL/glew.h>
#include <freeglut/glut.h>

int main(int argc, char* argv[])
 {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(200, 50);
    glutCreateWindow("Hello Triangle");

    glewInit();
    glPrintContextInfo(FALSE);

    glutDisplayFunc(RenderScene);
    glutReshapeFunc(ChangeSize);

    SetupRC();

    glutMainLoop();

   return 0;
 }
```

Include header file

Display mode setting
Window creation

Check extensions and version support

Callback functions registration

- glutDisplayFunc
- glutReshapeFunc
- glutKeyboardFunc
- glutMouseFunc
- glutMotionFunc
- glutSpecialFunc
- glutIdleFunc
- glutTimerFunc
- ...

State initialization

Event handling loop

GLUT functions

- `glutInit` **is used to initialize the GLUT library**
- `glutInitDisplayMode` **set the initial display mode with color model and various buffer modes**
- `glutWindowSize` **set the window size in pixels**
- `glutCreateWindow` **create window**
- `glutDisplayFunc` **set the display callback**
- `glutReshpeFunc` **set the reshape callback**
- `glutMainLoop` **enter the GLUT event processing loop**

Extension and Version Support

```
void glPrintContextInfo(bool printExtension)
{
    cout << "GL_VENDOR = "
         << (const char*)glGetString(GL_VENDOR) << endl;
    cout << "GL_RENDERER = "
         << (const char*)glGetString(GL_RENDERER) << endl;
    cout << "GL_VERSION = "
         << (const char*)glGetString(GL_VERSION) << endl;
    cout << "GL_SHADING_LANGUAGE_VERSION = "
         << (const char*)glGetString(GL_SHADING_LANGUAGE_VERSION) << endl;

    if (printExtension)
    {
        GLint numExt;
        glGetIntegerv(GL_NUM_EXTENSIONS, &numExt);
        cout << "GL_EXTENSIONS =" << endl;
        for (GLint i = 0; i < numExt; i++)
        {
            cout << "\t"
                 << (const char*)glGetStringi(GL_EXTENSIONS, i) << endl;
        }
    }
}
```

Print vendor, renderer, supported OpenGL version and GLSL version

Print supported extensions if printExtension is TRUE

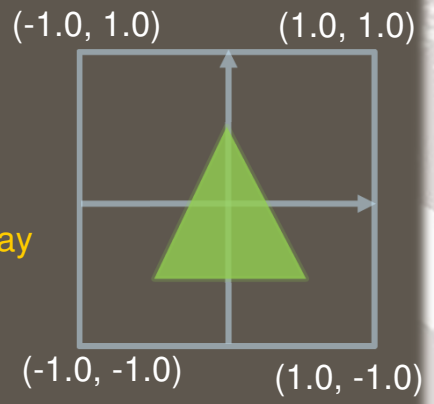
Setup Render Context

```
void setupRC()
{
    setShaders();

    glClearColor(0.2, 0.2, 0.2, 1.0);

    float vertices[] = {
           -0.5f, -0.5f, 0.0f, // left
            0.5f, -0.5f, 0.0f,  // right
            0.0f,  0.5f, 0.0f   // top    };

    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                                    3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}
```

Setup shaders (vertex shader and fragment shader)

Set frame buffer clear color to light gray

Vertices attribute definition
(coordinates for a triangle)



Setup vertex buffer
(from system memory to Graphics memory)

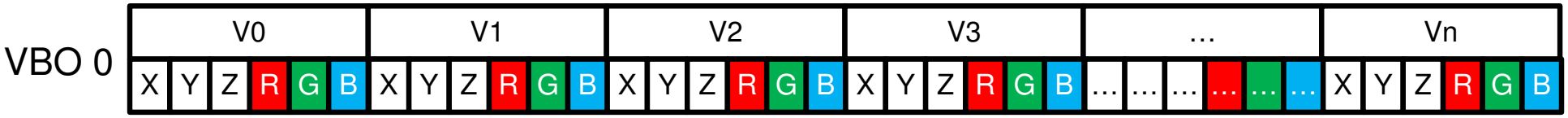
Link the vertex buffer with
the vertex shader input

Unbind VBO and VAO

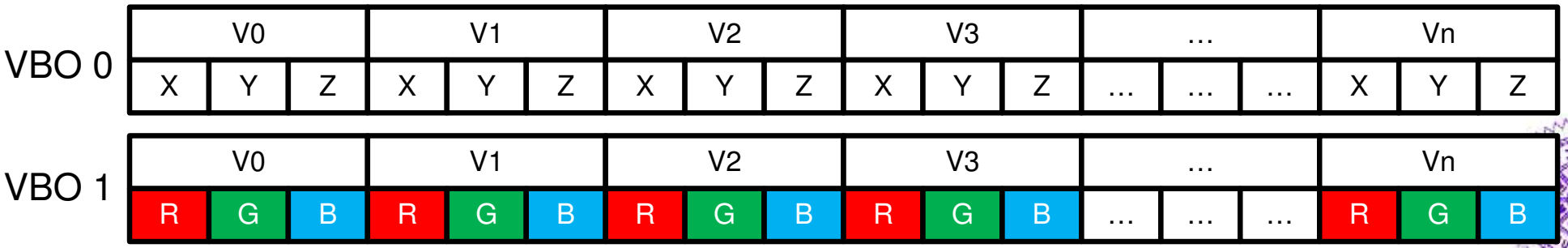
Vertex Buffer Object (VBO)

- **Memory storage for GPU to access vertex data**
  - **Used to store vertex attributes such as coordinate, normal, color, texture coordinate, etc.**
  - **Use `glDrawArray()` to draw primitives.**
    - `glDrawArrays(GL_TRIANGLES, 0, 9); // Three triangles`

Vertex buffer with compact vertex attributes

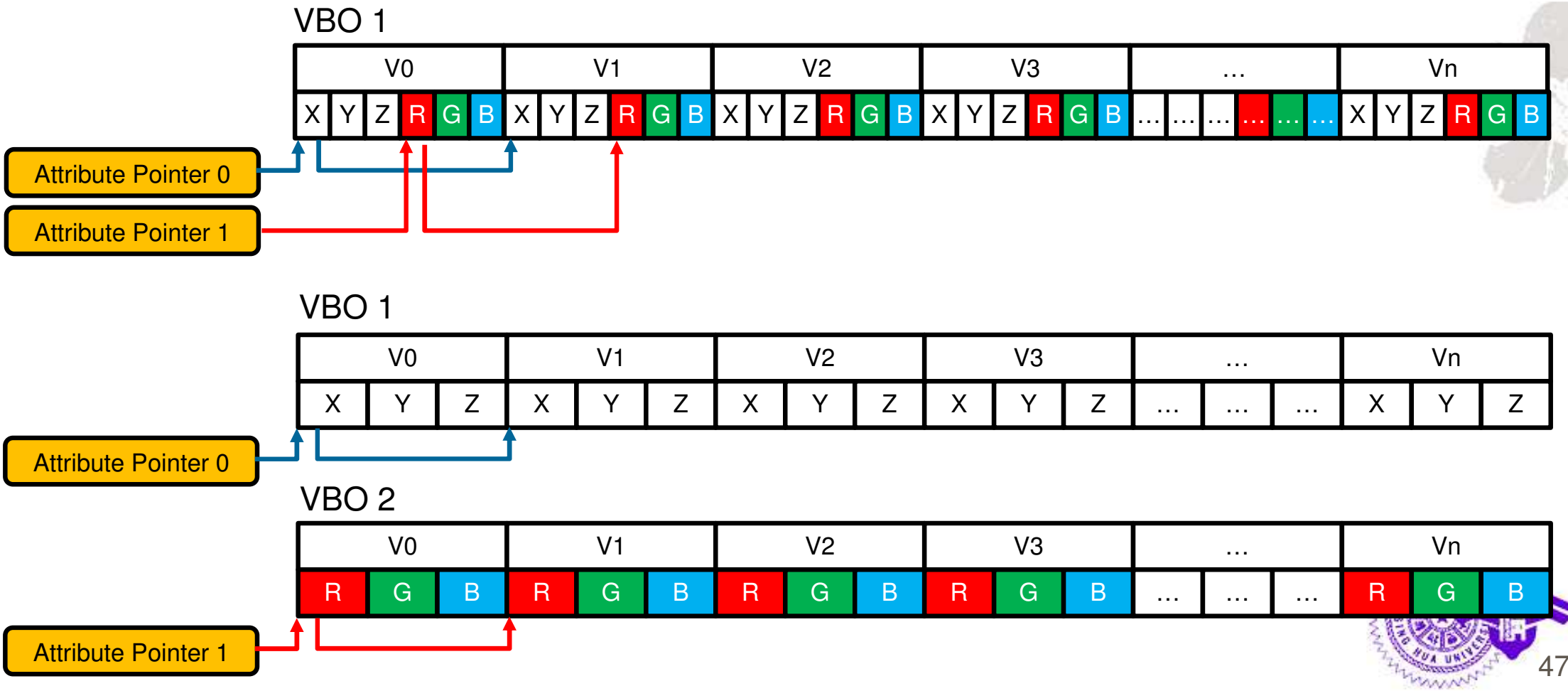


Vertex buffers with separated vertex attributes



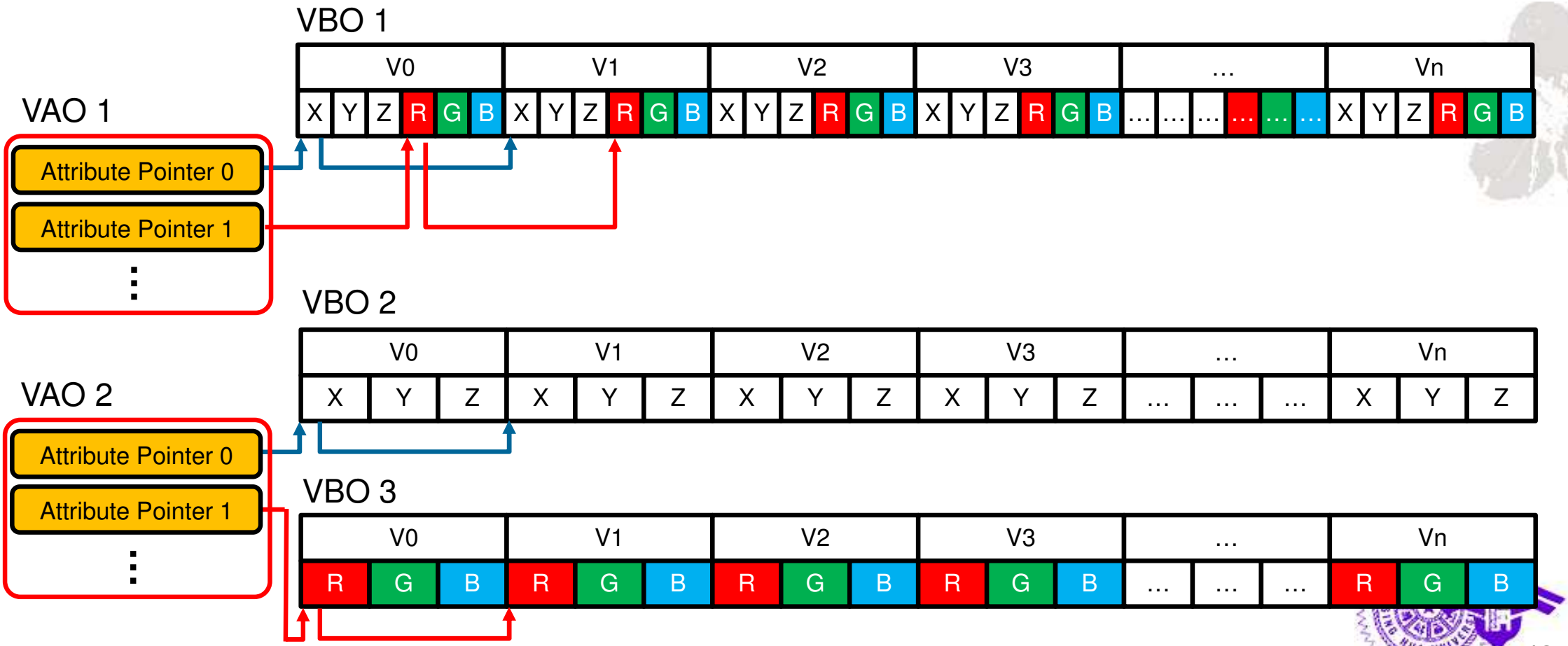
Vertex Attribute Pointer

- **Define an array of generic vertex attribute data through** `glVertexAttribPointer()`



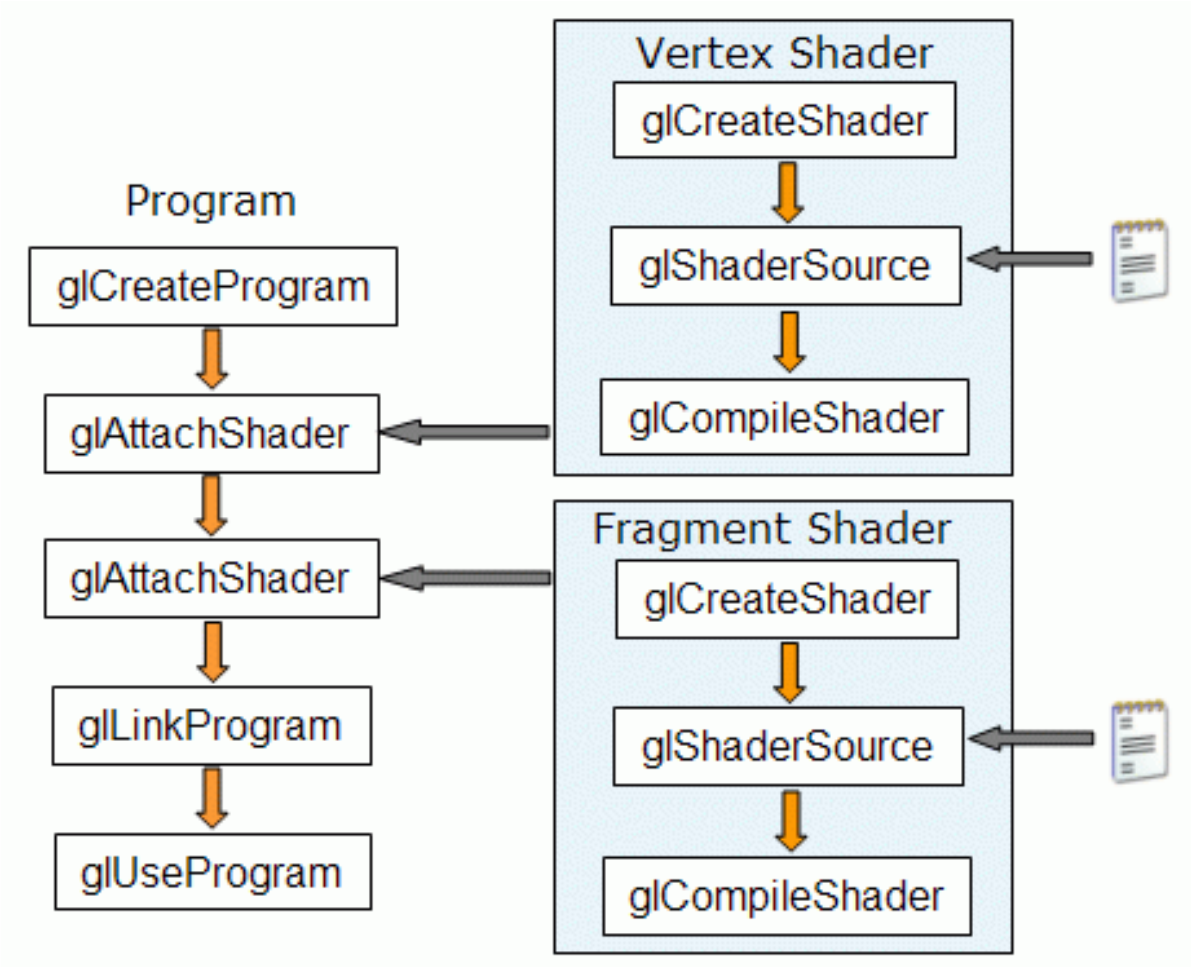
Vertex Array Object (VAO)

- An OpenGL Object that stores all the states needed by the supplied vertex data



Shader Creation Flow

- Compilation – Check for syntax error
- Link – Check for resource availability



Source: Lighthouse3d.com

Vertex Shader and Fragment Shader

```
// Vertex shader codes: shader.vs
#version 330 core

layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

Vertex shader source codes

```
// Fragment shader codes: shader.fs
#version 330 core

out vec4 FragColor;

void main() {
    FragColor = vec4(0.7f, 1.0f, 0.2f, 1.0f);
}
```

Fragment shader source codes

Setup Shader Program

```
void setShaders()
{
    GLuint v, f, p;
    char *vs = NULL;
    char *fs = NULL;

    v = glCreateShader(GL_VERTEX_SHADER);
    f = glCreateShader(GL_FRAGMENT_SHADER);

    vs = textFileRead("src/shader.vs");
    fs = textFileRead("src/shader.fs");

    glShaderSource(v, 1, (const GLchar**)&vs, NULL);
    glShaderSource(f, 1, (const GLchar**)&fs, NULL);

    free(vs);
    free(fs);
```

Create vertex shader

Create fragment shader

Shader source codes

Define shader sources

Check Shader Compile Status

```
GLint success;
char infoLog[1000];

glCompileShader(v);
```

Compile vertex shader

Check for vertex shader compile errors

```
glGetShaderiv(v, GL_COMPILE_STATUS, &success);
if (!success)
{
        glGetShaderInfoLog(v, 1000, NULL, infoLog);
        std::cout << "ERROR: VERTEX SHADER COMPILATION FAILED\n"
                  << infoLog << std::endl;
}
```

```
glCompileShader(f);
```

Compile fragment shader

Check for fragment shader compile errors

```
glGetShaderiv(f, GL_COMPILE_STATUS, &success);
if (!success)
{
        glGetShaderInfoLog(f, 1000, NULL, infoLog);
        std::cout << "ERROR: FRAGMENT SHADER COMPILATION FAILED\n"
                  << infoLog << std::endl;
}
```

Create Shader Program and Link

```
p = glCreateProgram();
```
Create shader program

```
glAttachShader(p,f);
glAttachShader(p,v);
```
Attach vertex shader and fragment shader into shader program

```
glLinkProgram(p);
```
Link shader program

Check for linking errors

```
glGetProgramiv(p, GL_LINK_STATUS, &success);
if (!success) {
        glGetProgramInfoLog(p, 1000, NULL, infoLog);
        std::cout << "ERROR: SHADER PROGRAM LINKING FAILED\n"
                  << infoLog << std::endl;
}
```

```
glDeleteShader(v);
glDeleteShader(f);
```
Clean up shader objects

```
if (success)
        glUseProgram(p);
else
        system("pause"), exit(123);
}
```
Use shader program if there is no error in both compiling and linking stages

Render Scene Callback Function

```
void RenderScene(void) {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
            GL_STENCIL_BUFFER_BIT);

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

    glutSwapBuffers();
}
```

Clear color buffer, depth buffer, and stencil buffer

Draw object (a triangle in vertex buffer)

Swap buffer from back to front for display

The display callback

- The display callback is executed when
  - Window is first opened
  - Window is reshaped
  - Window is exposed
  - Post a redisplay message (`glutPostRedisplay()`)
- Every GLUT program must have a display callback

Change Size Callback Function

```
void ChangeSize(int w, int h) {
    glViewport(0, 0, w, h);
}
```

Set the viewport to the size of entire window whenever it is resized

Note that the setting will cause the rendered objects resized as well

The codes shown above will cause object distortion if the aspect ratio is not the same with the original window size

OpenGL Programming Practice #1

- Follow TA's instruction to setup the programming environment for OpenGL in Microsoft Visual Studio / MacOS Xcode
  - Run the OpenGL sample code
  - Play with the vertex data
  - Modified the vertex/fragment shaders and see if the results meet with your expectation
  - Use KeyPress callback function to toggle wireframe/fill mode with

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)
```

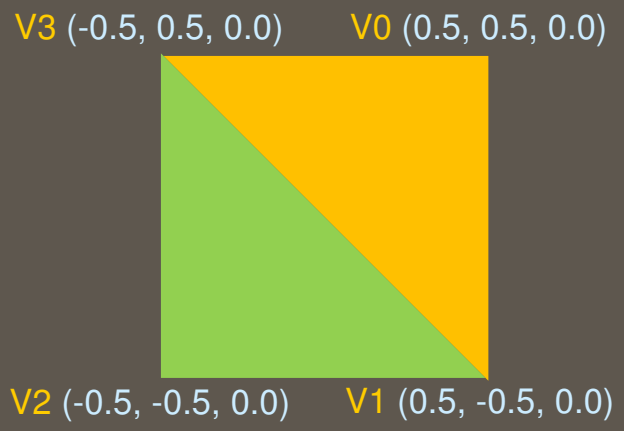
Variation of Vertex Data Formations

```
float vertices[] = {
    // first triangle
     0.5f,  0.5f, 0.0f,   // top right
     0.5f, -0.5f, 0.0f,   // bottom right
    -0.5f,  0.5f, 0.0f,   // top left
    // second triangle
     0.5f, -0.5f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   // bottom left
    -0.5f,  0.5f, 0.0f    // top left
};
```

Vertex data for defining a rectangle using two triangles

Duplicated

Duplicated



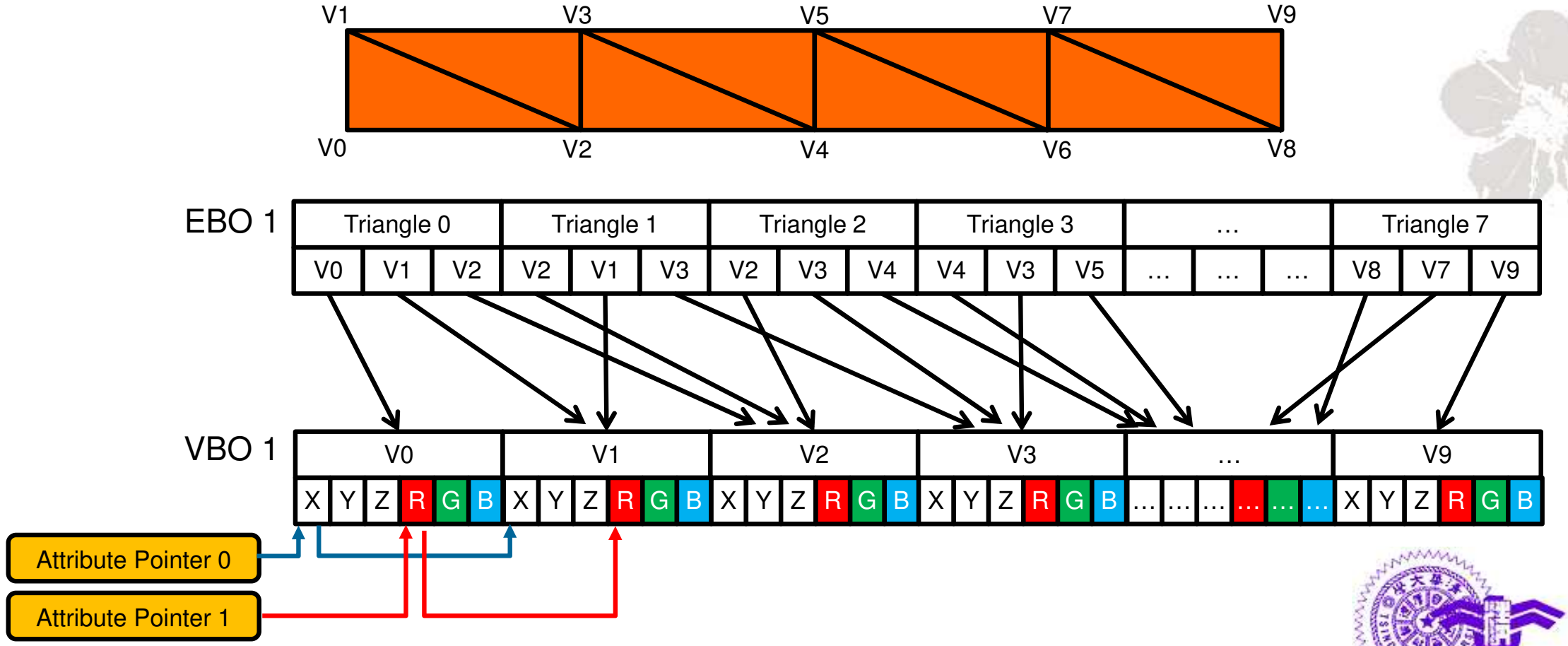
Vertex data for defining a rectangle using two triangles without vertex data duplication

```
float vertices[] = {
     0.5f,  0.5f, 0.0f,   // top right
     0.5f, -0.5f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   // bottom left
    -0.5f,  0.5f, 0.0f    // top left
};
unsigned int indices[] = {  // note that the indices start from 0!
    0, 1, 3,    // first triangle
    1, 2, 3     // second triangle
};
```

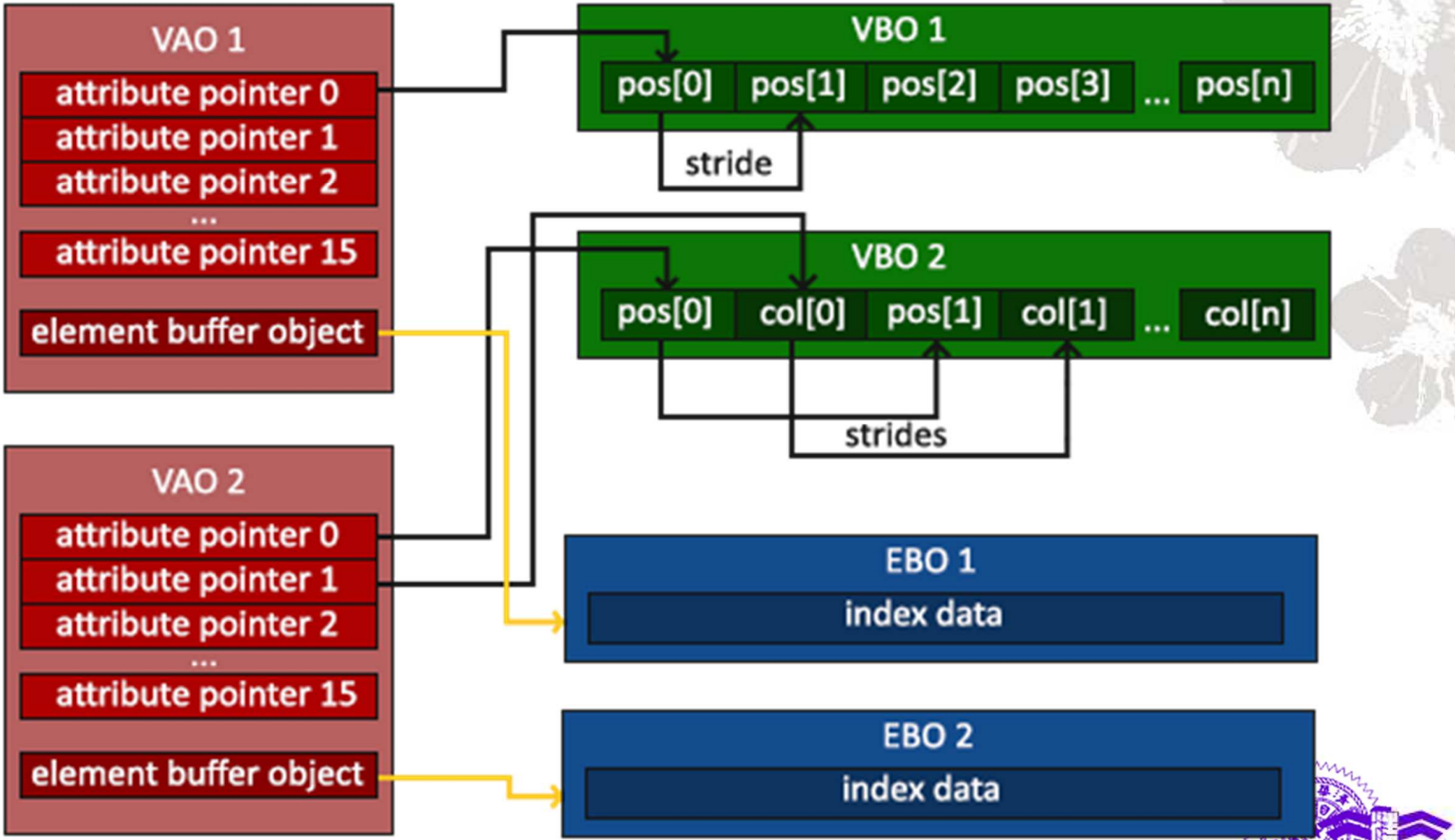
Using vertex indices

Element Buffer Object (EBO)

- An OpenGL Object that define vertex data through indices and VBO



Keep EBO in VAO



Source: Learn OpenGL by Joey de Vries

OpenGL Programming Practice #2

- Revise `setupRC` to define the same rectangle using indices as in page 58
  - Use the following APIs to setup EBO and bind the EBO

```
unsigned int EBO;
glGenBuffers(1, &EBO);
```

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
             GL_STATIC_DRAW);
```

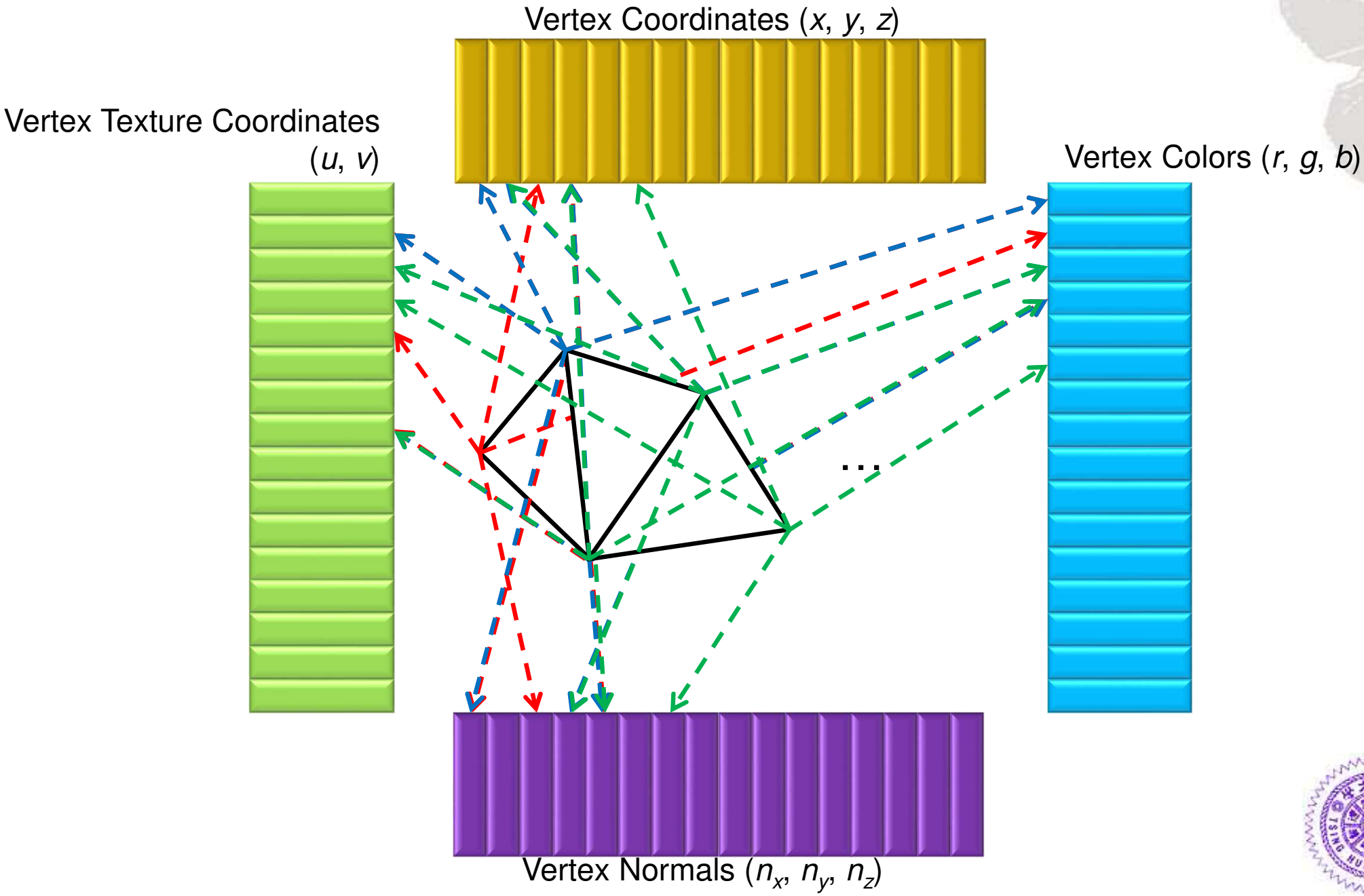
- Revise RenderScene to draw the indexed triangles using `glDrawElements()` as follows

```
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

3D File Formats

- Define 3D models/scene
- Data include
  - Vertex coordinates, vertex attributes
  - Meshes (triangles, polygons), groups
  - Textures, materials, light sources
  - Rigging control
- Various 3D file formats
  - STL, OBJ, FBX, COLLADA, 3DS…
  - We will used Wavefront OBJ 3D model format
    - Sample 3D model files (color, normal, texture)

Define a 3D Model through Indices



OpenGL Programming Practice #3

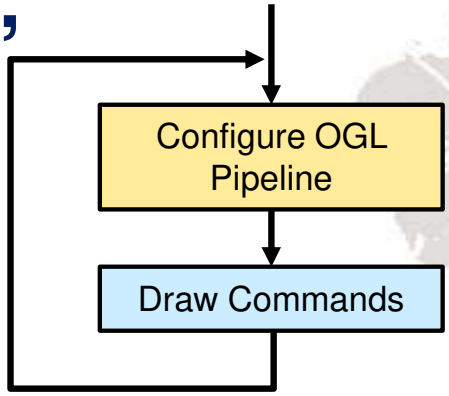
- Revise `setupRC` to define a 3D cube using the vertex and index data as in boxC.obj
  - Use the following API to enable depth test for rendering a 3D cube with correct hidden surface removed

```
glEnable(GL_DEPTH_TEST);
```

- Revise the vertex coordinates and colors to change/deform the shape and colors
  - Try to revise the number of vertices from 24 to 8 by removing duplicated vertices

Change Pipeline Configuration

- When is the right time to change pipeline configuration?
  - Before draw codes (`glDrawArrays()`, `glDrawElements()`, …) initiate
- What can be changed?
  - Vertex data (different models)
  - Fixed function states/functions (e.g., `glEnable()`)
  - Shaders (switch to other shader program)
  - Shader inputs/constants



Shader Inputs -- *in*

- Pass vertex attributes to vertex shader
  - Vertex coordinates, colors, normal, texture coordinates, etc.

```
layout (location = 0) in vec3 aPos;    // vertex coordinate (x, y, z)
Layout (location = 1) in vec3 aColor;  // vertex color (r, g, b)
```

- Pass interpolated attributes to fragment shader
  - Fragment color, normal, texture coordinates, etc.
  - Must have been defined in vertex shader as outputs (*out*)

```
in vec3 Color;     // fragment color interpolated from vertex colors
in vec3 Normal;    // fragment normal interpolated from vertex normals
```

Shader Outputs – gl_Position/out

- gl_Position
  - Build-in vertex position output variable
  - Output final vertex position after vertex shader processing

```
gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
```

- out
  - Output vertex attributes after vertex processing
  - Output fragment color after fragment processing

```
... // vertex shader
out vec4 Color;
...
main(){
   ...
   Color = vec4(r, g, b, 1.0);
}
```

Vertex color

```
... // Fragment shader
out vec4 FragColor;
...
main(){
   ...
   FragColor = vec4(r, g, b, 1.0);
}
```

Fragment color

Constant Values in Shaders

- Use ***uniform*** to represent "**constant**" values in shaders
  - Constant means it can not be altered during a draw command, such as matrices, light source position, model material coefficients, etc.

```
uniform mat4 mvp;                // model-view-projection matrix
uniform vec3 light;              // light source position (x, y, z)
uniform float specular_power; // material specular exponent
```

  - Use ***glGetUniformLocation()*** to get the uniform variable location in a shader

```
// Get the uniform locations in shaders
MVP = glGetUniformLocation(p, "mvp");      <- Uniform "mvp" in program p
LightPos = glGetUniformLocation(p, "light"); <- Uniform "light" in program p
```

Constant Values in Shaders

- Use *glUniform{1|2|3|4}{f|u|ui}()* or *glUniform{1|2|3|4}{f|u|ui}v()* or *glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv()* to pass "constant" values into shaders

```
glUniform1f(Spec_Pow, 30.0f);   // Specify a floating uniform
```

```
glUniform3f(LightPos, 1.0f, 2.0f, 2.5f);  // Specify 3 floats
--- or ---
GLfloat light[3] = {1.0f, 2.0f, 2.5f};
glUniform3fv(LightPos, 1, light};   // Specify a vec3 float
```

```
GLfloat matrix[16] = {1.0f, 0.5f, ..., 1.0f};  // 16 elements for
                                               // a 4x4 matrix
glUniformMatrix4fv(MVP, 1, GL_FLASE, matrix};  // Specify a 4x4
                                               // matrix
```

OpenGL Programming Practice #4

- Learn from this practice for
  - Defining inputs (using `in` or `uniform`) and outputs (using `out`) to shaders
  - Data linkage between vertex and fragment shaders
- Revise vertex and fragment shaders to accept vertex color attribute and render the 3D cube using the vertex colors defined as in boxC.obj
  - You can store the vertex data in a VBO or in separated VBOs (a position VBO and a color VBO)
- Revise the vertex data (positions and colors) defined as in boxC2.obj

Additional Notes

Problems in MacOS using glut and glew

A Simple OpenGL Framework with glfw and glew

A Simple OpenGL Framework with glfw and glad

Problems in MacOS using glut + glew

- No freeglut build for MacOS
  - Use GLUT.framework instead
- Default supported versions for OpenGL is 2.1 and for GLSL is 1.2
  - Workaround method
    - *glutInitDisplayMode*(GLUT_3_2_CORE_PROFILE | ...)
      can support version over 3.2+ (*hardware dependent)
- Incorrect in retina display
  - Scale down to the left
    - Non-retina display seems fine (e.g., external LCD)

OpenGL using glfw and glew

- Put the related header files and libraries of glfw and glew into the project include and lib directories
- Compare to the glut+glew framework at page 42, only 4 places need to be revised in the sample codes
  - The included header files
  - The context and window creation
  - The callback functions
  - The event handling mechanism

Revised OpenGL Example (glfw + glew)

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
```

Include header file

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

Glfw initialization

```
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
//glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

Uncomment it if using Mac

Window and Context creation

```
GLFWwindow* window = glfwCreateWindow(WINDOW_WIDTH,
                                      WINDOW_HEIGHT, "Hello Triangle", NULL, NULL);
glfwMakeContextCurrent(window);
```

Check extensions and version support

```
glewInit();
glPrintContextInfo(FALSE);
```

Revised OpenGL Example (glfw + glew)

```
glfwSetWindowSizeCallback(window, ChangeSize);
//glfwSetCharCallback(window, KeyPress);

SetupRC();

while (!glfwWindowShouldClose(window))
{
    RenderScene();

    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();

return 0;
}
```

Callback functions registration
glfwSetCharCallback
glfwSetKeyCallback
…

Event handling loop

Rendering tasks

Need an event to set it true if terminated
glfwSetWindowShouldClose(window, true);
e.g., in the callback function for KeyPress

swap buffers and
poll IO events (keys pressed/released, mouse moved etc.)

Clear all previously allocated GLFW resources

OpenGL using glfw and glad

- Put the related header files of glfw and glad into the project include directory and the library of glfw into the project lib directory
- Remember to include glad.c into your project to build together with your OpenGL application program
- Compare to the glfw+glew framework at pages 75 and 76, we only need to replace glewInit() with the corresponding glad codes as shown in page 78

Revised OpenGL Example (glfw + glad)

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

Include header file

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

Glfw initialization

```
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    //glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

Uncomment it if using Mac

```
    GLFWwindow* window = glfwCreateWindow(WINDOW_WIDTH,
                                          WINDOW_HEIGHT, "Hello Triangle", NULL, NULL);
    glfwMakeContextCurrent(window);
```

Window and Context creation

```
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    glPrintContextInfo(FALSE);
```

Load OpenGL functions and extension support
(Just replace glewinit() with these codes)

Revised OpenGL Example (glfw + glad)

```
glfwSetWindowSizeCallback(window, ChangeSize);
//glfwSetCharCallback(window, KeyPress);

SetupRC();

while (!glfwWindowShouldClose(window))
{
    RenderScene();

    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();

return 0;
}
```

Callback functions registration
glfwSetCharCallback
glfwSetKeyCallback
…

Event handling loop

Rendering tasks

Need an event to set it true if terminated
glfwSetWindowShouldClose(window, true);
e.g., in the callback function for KeyPress

swap buffers and
poll IO events (keys pressed/released, mouse moved etc.)

Clear all previously allocated GLFW resources

The remaining part is similar to the glfw+glew framework and is no need to change

Q&A