Intro to OpenGL

Immersive Games Technologies

Dr Ji-Jian Chin

University of Plymouth 2024

COMP3016

Topics for this lecture

- Part 1: OpenGL – shaders, evolution, pipelines and first program
- Part 2: OpenGL - extensions, vertices and textures

What is OpenGL

OpenGL is a computer graphics rendering *application programming interface (*API)

- can generate high-quality color images by rendering with geometric and image primitives
- What OpenGL can do:
  - Display/Generate primitives
  - coordinate transformations (transformation matrix manipulation)
  - lighting calculations
  - antialiasing
  - Pixel update operations
- forms the basis of many interactive applications that include 2D/3D graphics
- By using OpenGL, the graphics part of your application can be
  - operating system independent
  - window system independent (you need to handle windows yourself or use GLFW)
- when using OpenGL 4.x, you must use *shaders* for the graphics processing
  - we only introduce a subset of OpenGL's shader capabilities (Vertex, Fragment, Geometry)

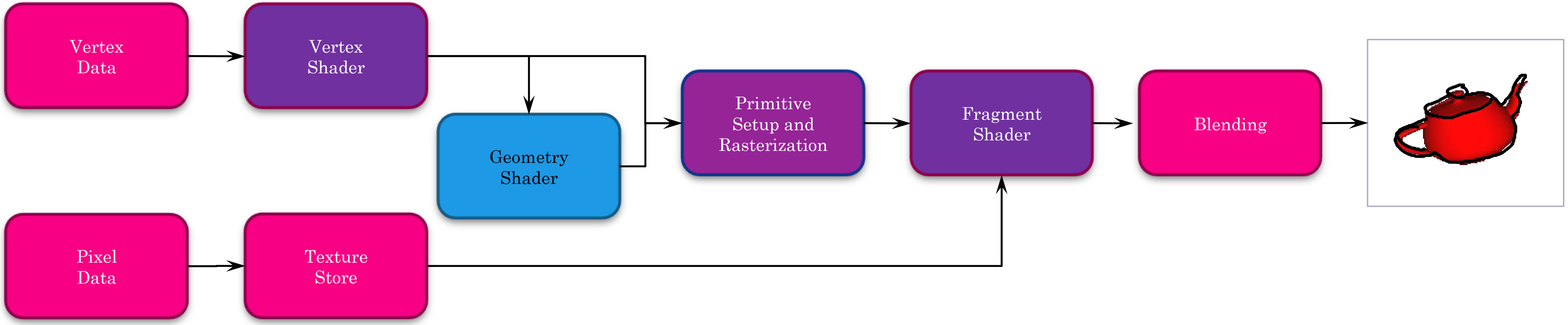
What are Shaders?

- Translate C++ code and data
- 2nd compile stage
- …more

Evolution of OpenGL

OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – geometry shaders

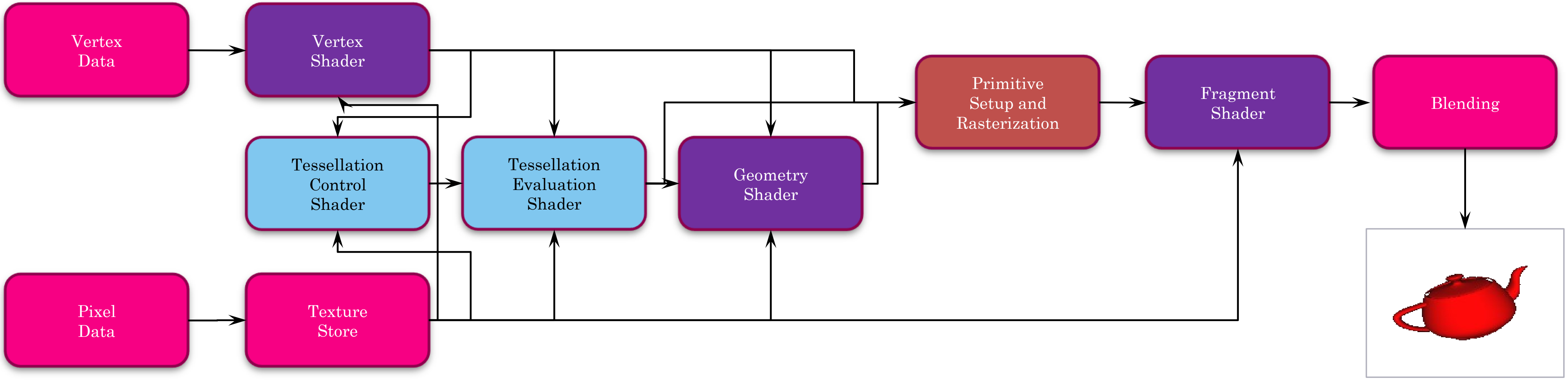
- modify geometric primitives within the graphics pipeline
- OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
  - currently two types of profiles: *core* and *compatible*



Evolution of OpenGL

OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders

- Latest version is 4.6 in 2017


Vertex Data
Vertex Shader
Tessellation Control Shader
Tessellation Evaluation Shader
Geometry Shader
Primitive Setup and Rasterization
Fragment Shader
Blending
Pixel Data
Texture Store

Evolution of OpenGL

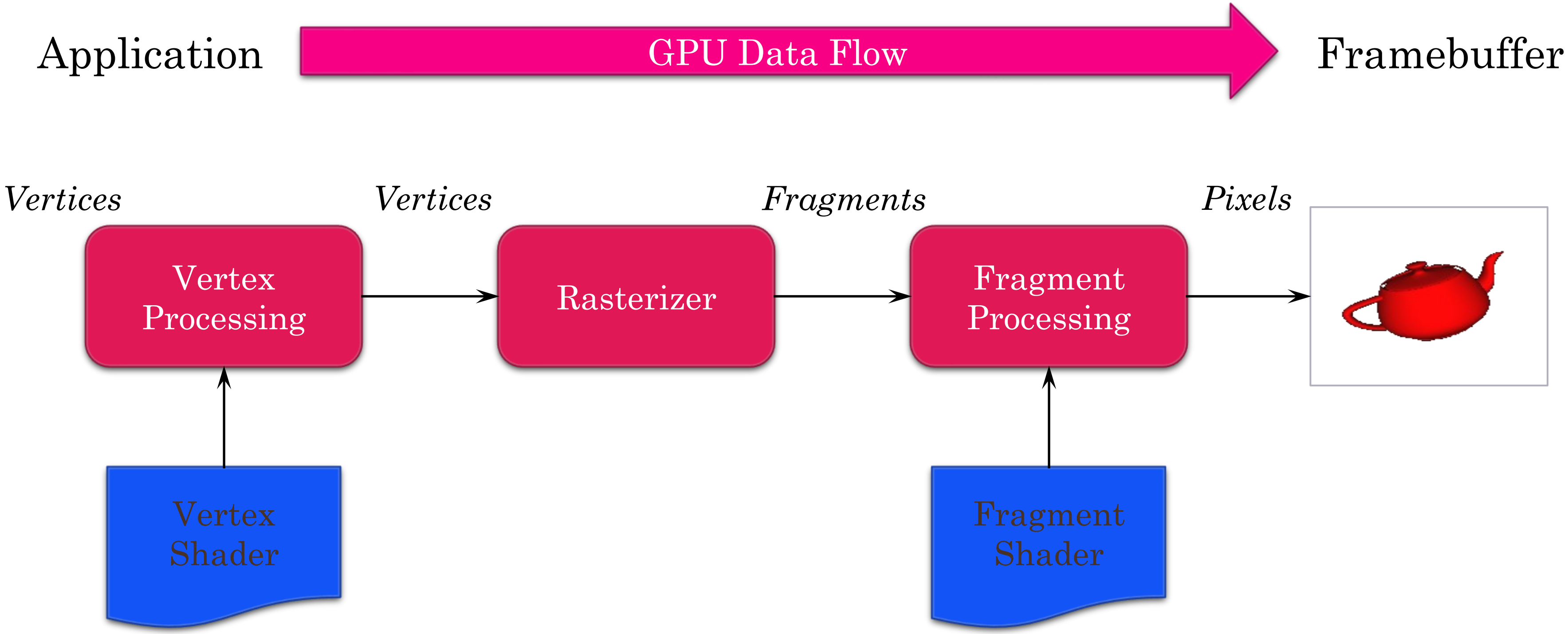
OpenGL ES 2.0

- Designed for embedded and hand-held devices such as cell phones
- Based on OpenGL 3.1
- Shader based

WebGL 1.0 (2.0)

- JavaScript implementation of ES 2.0 (3.0)
- Runs on most recent browsers

OpenGL Pipeline *(Simplification)*



Application

GPU Data Flow

Framebuffer

Vertices

Vertex Processing

Vertices

Rasterizer

Fragments

Fragment Processing

Pixels

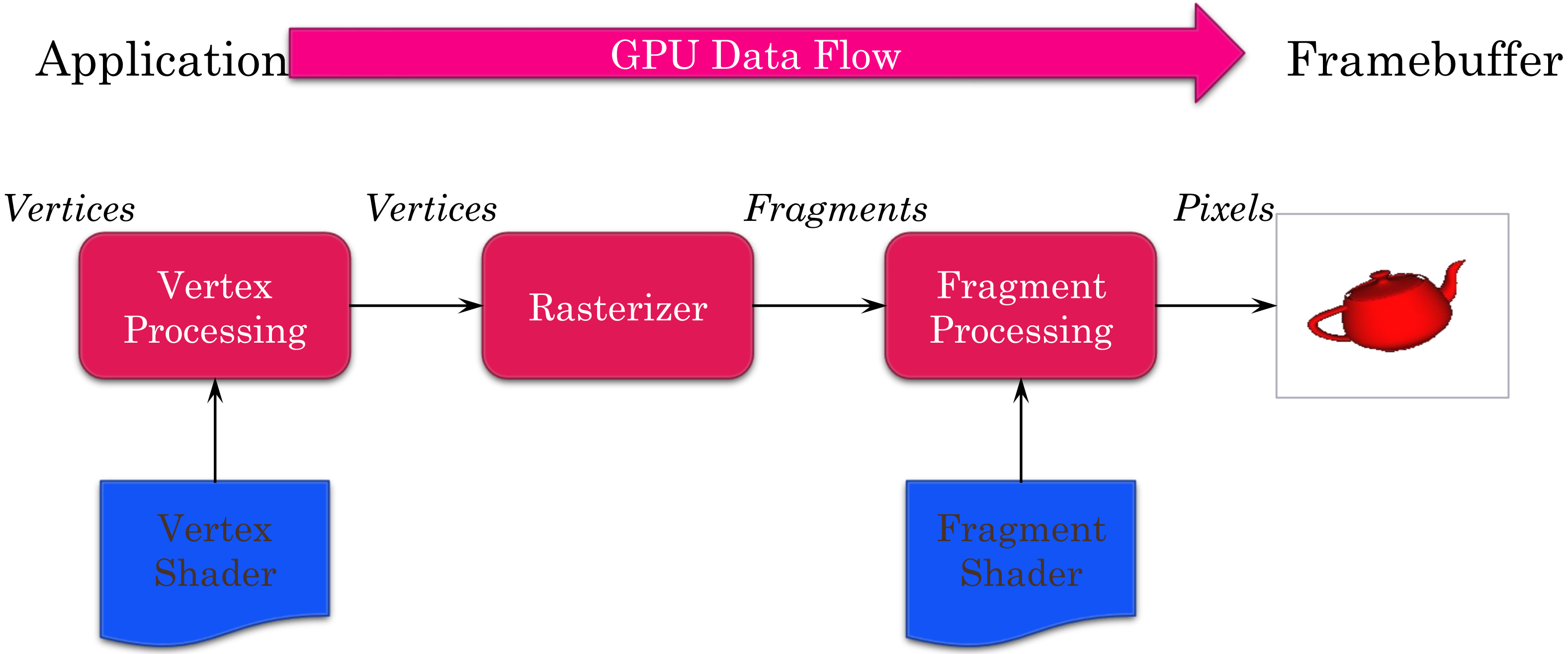
Vertex Shader

Fragment Shader

OpenGL Pipeline *(Simplification)*

OpenGL within current programs mostly follows those steps:

- Create shader programs
- Create buffer objects and load data into them
- “Connect” data locations with shader variables
- Render



OpenGL Requirements/Setup

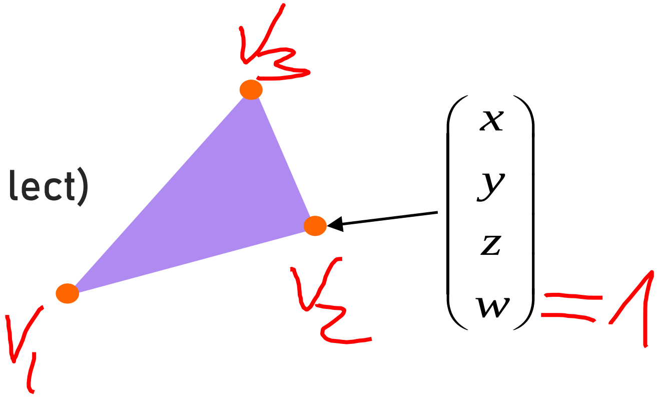
- OpenGL applications need a place to render into
    - usually an on-screen window
- Need to communicate with native windowing system
- We use GLFW
    - simple, cross platform
    - handles all windowing operations:
        - opening windows
        - input processing (mouse, keyboard, joystick, …)

OpenGL Requirements/Setup

- Operating systems deal with library functions differently
  - compiler linkage and runtime libraries may expose different functions
- OpenGL has many versions and profiles which expose different sets of functions
  - managing function access is cumbersome, and window-system dependent
- open-source library, GLEW, hides those details (less dealing with dependencies)

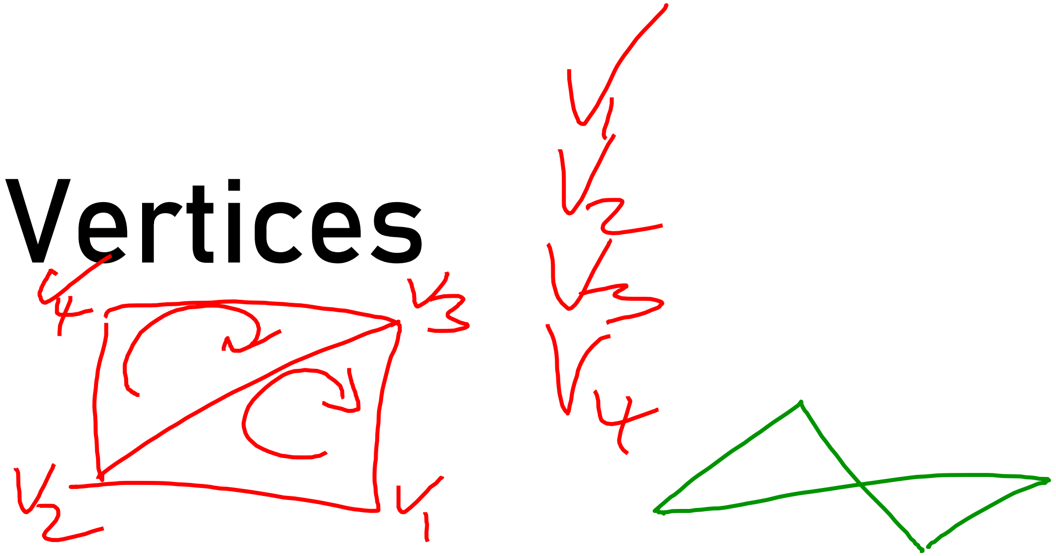
OpenGL Intro the Vertices

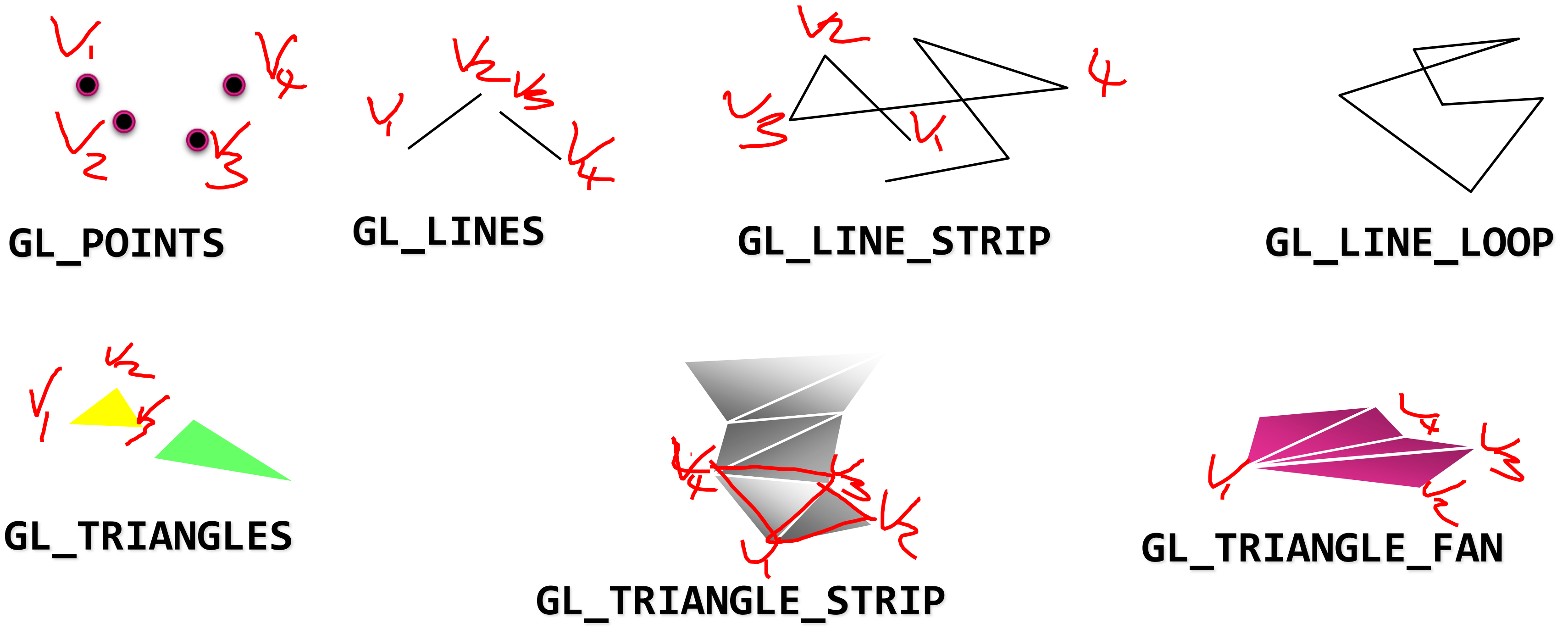
- Geometric objects are represented using *vertices*
- A vertex is a collection of generic attributes
  - positional coordinates (homogeneous cords, more next lect)
  - colors
  - texture coordinates
  - any other data associated with that point in space
- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in vertex buffer objects (VBOs)
- VBOs must be stored in vertex array objects (VAOs)



OpenGL Intro the Vertices

- 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 A First Program (Project1)

- Triangles.cpp

```
#include "GL/glew.h"
#include "GL/freeglut.h"
#include "GLFW/glfw3.h"
#include "LoadShaders.h"
```

OpenGL A First Program (Project1)

- Triangles.cpp

```
#include "GL/glew.h"
#include "GL/freeglut.h"
#include "GLFW/glfw3.h"
#include "LoadShaders.h"
```

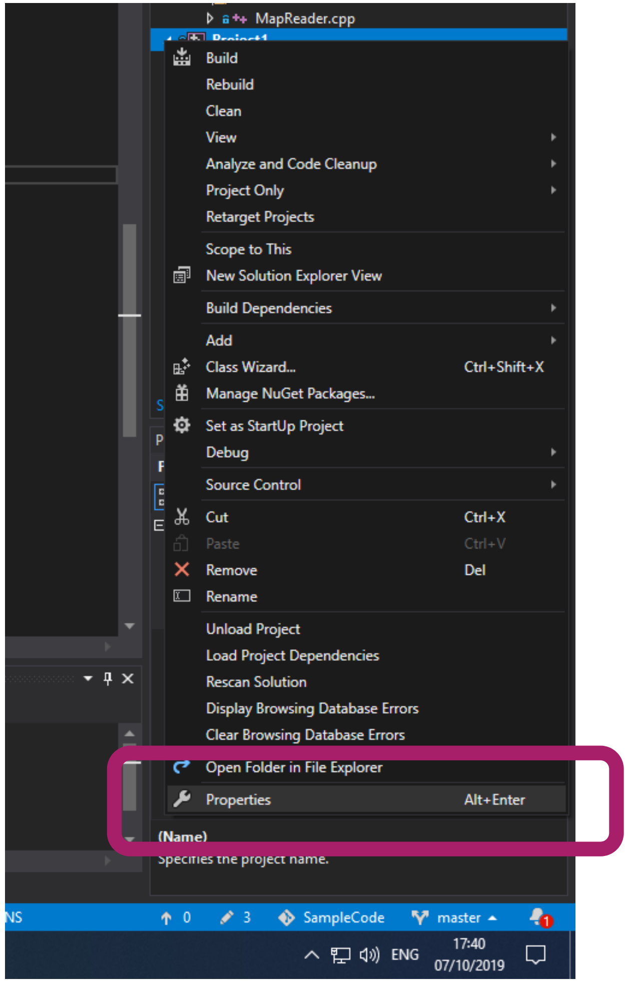
Your Own or additional libraries?

OpenGL A First Program (Project1)

- Triangles.cpp

```
#include "GL/glew.h"
#include "GL/freeglut.h"
#include "GLFW/glfw3.h"
#include "LoadShaders.h"
```

Your Own or additional libraries?

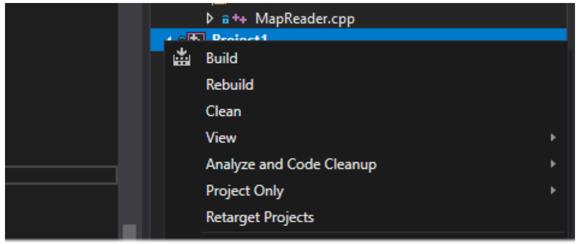


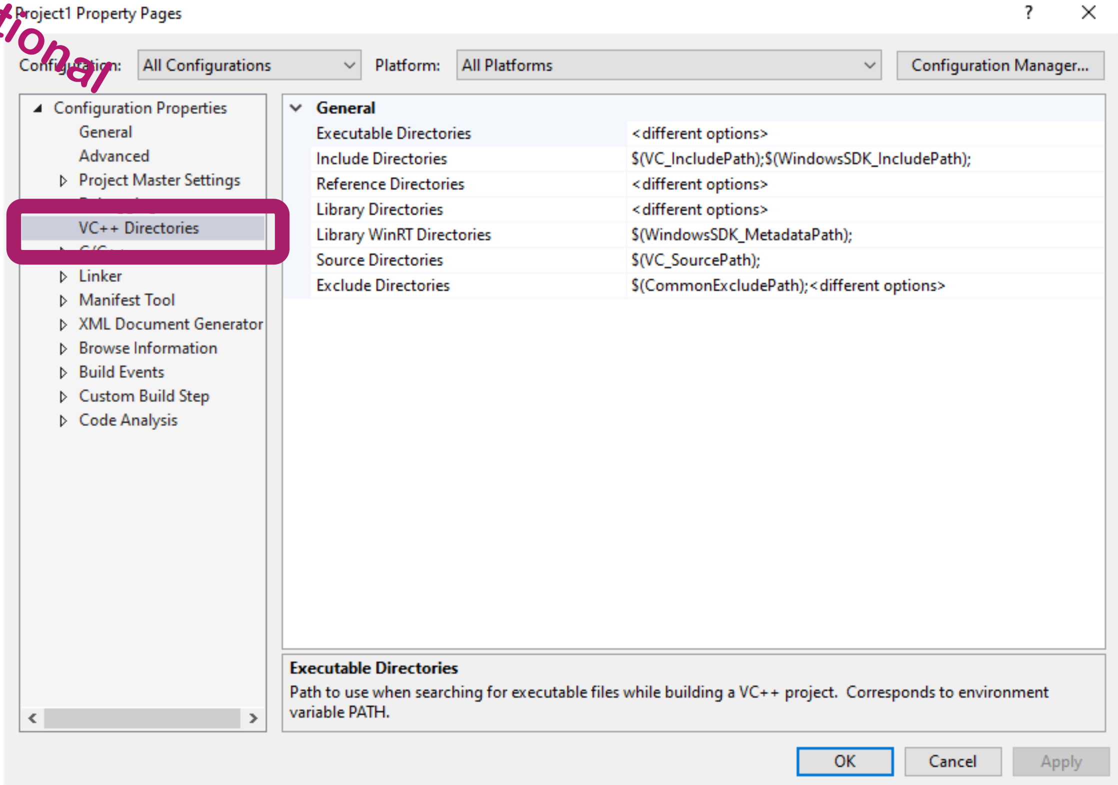
OpenGL A First Program (Project1)

- Triangles.cpp

```
#include "GL/glew.h"
#include "GL/freeglut.h"
#include "GLFW/glfw3.h"
#include "LoadShaders.h"
```

Your Own or additional libraries?





Configuration: All Configurations Platform: All Platforms

General

Executable Directories	<different options>
Include Directories	$(VC_IncludePath);$(WindowsSDK_IncludePath);
Reference Directories	<different options>
Library Directories	<different options>
Library WinRT Directories	$(WindowsSDK_MetadataPath);
Source Directories	$(VC_SourcePath);
Exclude Directories	$(CommonExcludePath);<different options>

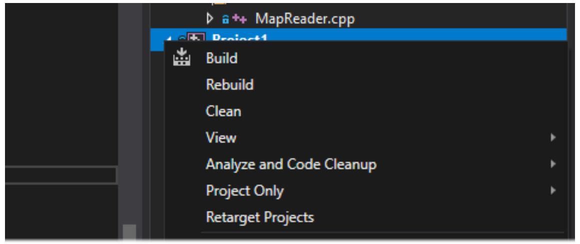
Executable Directories
Path to use when searching for executable files while building a VC++ project. Corresponds to environment variable PATH.

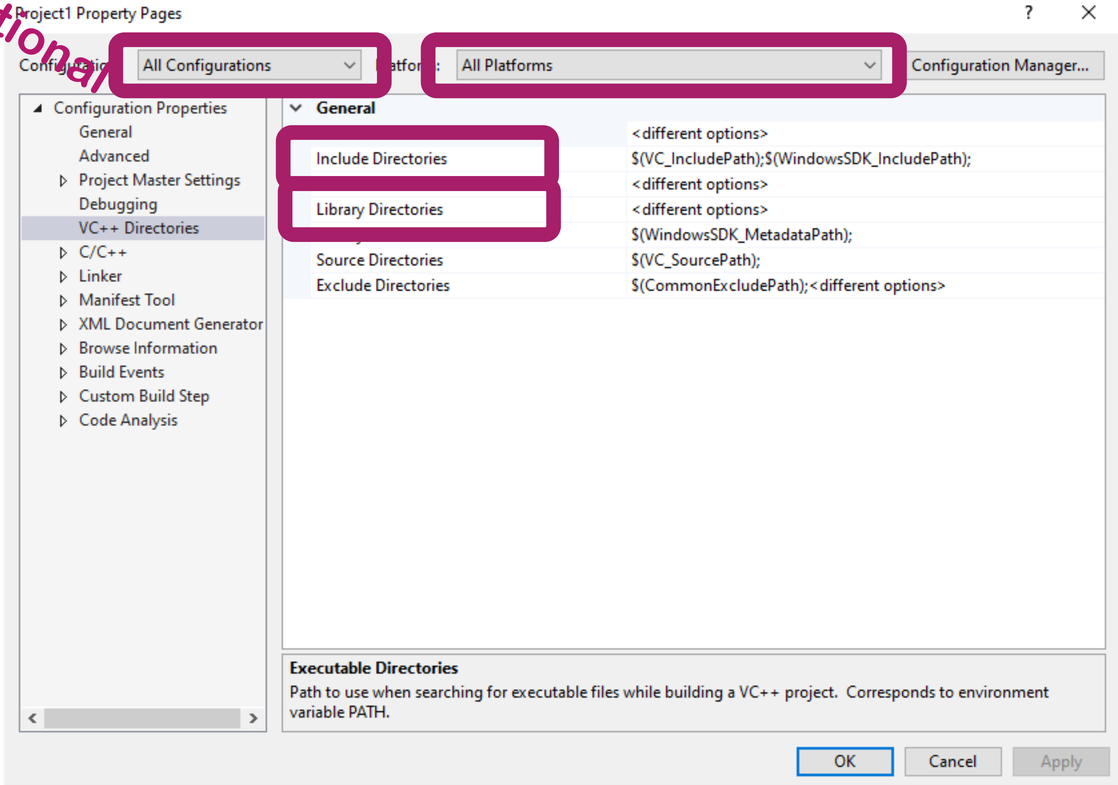
OpenGL A First Program (Project1)

- Triangles.cpp

```
#include "GL/glew.h"
#include "GL/freeglut.h"
#include "GLFW/glfw3.h"
#include "LoadShaders.h"
```

Your Own or additional libraries?





Configuration: All Configurations

Platform: All Platforms

	<different options>
Include Directories	$(VC_IncludePath);$(WindowsSDK_IncludePath);
	<different options>
Library Directories	<different options>
	$(WindowsSDK_MetadataPath);
Source Directories	$(VC_SourcePath);
Exclude Directories	$(CommonExcludePath);<different options>

Executable Directories

Path to use when searching for executable files while building a VC++ project. Corresponds to environment variable PATH.

OpenGL A First Program (Project1)

- Triangles.cpp

```
#include "GL/glew.h"
#include "GL/freeglut.h"
#include "GLFW/glfw3.h"
#include "LoadShaders.h”

enum VAO_IDs { Triangles, NumVAOs };
enum Buffer_IDs { ArrayBuffer, NumBuffers };
enum Attrib_IDs { vPosition = 0 };

GLuint  VAOs[NumVAOs];
GLuint  Buffers[NumBuffers];

const GLuint  NumVertices = 6;
```

OpenGL A First Program (Project1)

- Triangles.cpp

```
#include "GL/glew.h"
#include "GL/freeglut.h"
#include "GLFW/glfw3.h"
#include "LoadShaders.h”

enum VAO_IDs { Triangles, NumVAOs };
enum Buffer_IDs { ArrayBuffer, NumBuffers };
enum Attrib_IDs { vPosition = 0 };

GLuint  VAOs[NumVAOs];
GLuint  Buffers[NumBuffers];

const GLuint  NumVertices = 6; // how many vertices are in the scene
```

- We shouldn’t use global variables but for this example

OpenGL A First Program (Project1)

- VAOs store the data of an 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

OpenGL A First Program (Project1)

- Triangles.cpp

```
 const GLuint  NumVertices = 6; // vertices in the scene

void
init(void)
{
        glGenVertexArrays(NumVAOs, VAOs);
        glBindVertexArray(VAOs[Triangles]);

        GLfloat  vertices[NumVertices][2] = {
                { -0.90f, -0.90f }, {  0.85f, -0.90f }, { -0.90f,  0.85f },  // Triangle 1
                {  0.90f, -0.85f }, {  0.90f,  0.90f }, { -0.85f,  0.90f }   // Triangle 2
        };
```

1 (GLuint) Pointer

OpenGL A First Program (Project1)

- Triangles.cpp (init)

```
glGenVertexArrays(NumVAOs, VAOs);
glBindVertexArray(VAOs[Triangles]);

GLfloat  vertices[NumVertices][2] = {
        { -0.90f, -0.90f }, {  0.85f, -0.90f }, { -0.90f,  0.85f },  // Triangle 1
        {  0.90f, -0.85f }, {  0.90f,  0.90f }, { -0.85f,  0.90f }   // Triangle 2
};
glGenBuffers(NumBuffers, Buffers);
glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
glBufferStorage(GL_ARRAY_BUFFER, sizeof(vertices), vertices, 0);
```

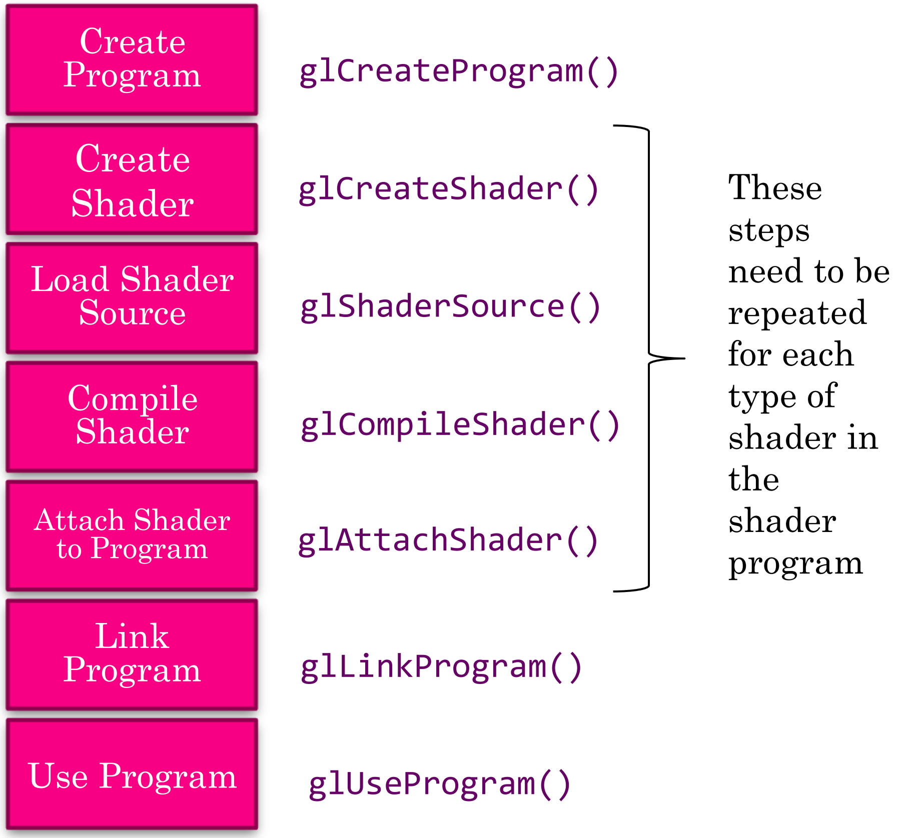
OpenGL A First Program (Project1)

- Triangles.cpp (init)

```
    ShaderInfo  shaders[] =
    {
        { GL_VERTEX_SHADER, "media/triangles.vert" },
        { GL_FRAGMENT_SHADER, "media/triangles.frag" },
        { GL_NONE, NULL }
    };

    GLuint program = LoadShaders(shaders);
    glUseProgram(program);

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

OpenGL A First Program (Project1)

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



Create Program	glCreateProgram()
Create Shader	glCreateShader()
Load Shader Source	glShaderSource()
Compile Shader	glCompileShader()
Attach Shader to Program	glAttachShader()
Link Program	glLinkProgram()
Use Program	glUseProgram()

These steps need to be repeated for each type of shader in the shader program

OpenGL A First Program (Project1)

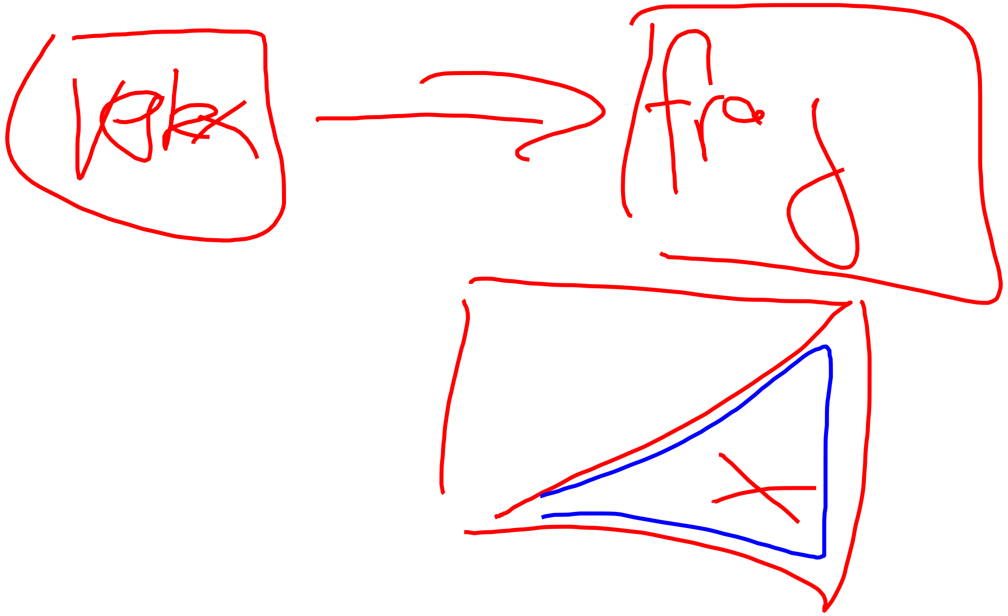
- **Vertex Shader**

```
#version 450 core

layout( location = 0 ) in vec4 vPosition;

void
main()
{
    gl_Position = vPosition;
}
```

OpenGL A First Program (Project1)

- **Fragment Shader**

```
#version 450 core

out vec4 fColor;

void main()
{
    fColor = vec4(0.5, 0.4, 0.8, 1.0);
}
```



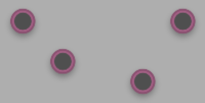
OpenGL A First Program (Project1)

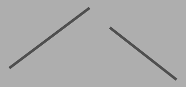
- Triangles.cpp (display)

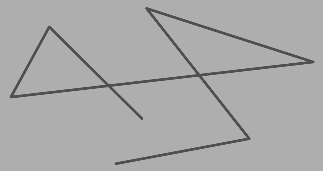
```
void
display(void)
{
    static const float black[] = { 0.0f, 0.0f, 0.0f, 0.0f };

    glClearBufferfv(GL_COLOR, 0, black);

    glBindVertexArray(VAOs[Triangles]);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
}
```

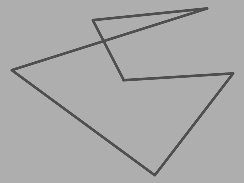
OpenGL Intro the Vertices

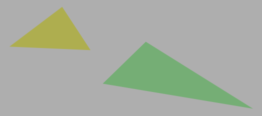
- All primitives are specified by vertices

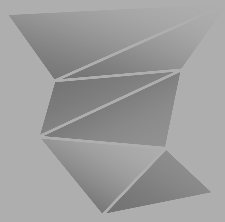

GL_POINTS

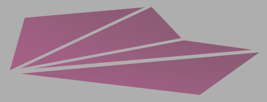

GL_LINES


GL_LINE_STRIP


GL_LINE_LOOP


GL_TRIANGLES


GL_TRIANGLE_STRIP


GL_TRIANGLE_FAN

OpenGL A First Program (Project1)

- Triangles.cpp (main)

```
int main(int argc, char** argv)
{
        …
}
```

OpenGL A First Program (Project1)

- Triangles.cpp (main)

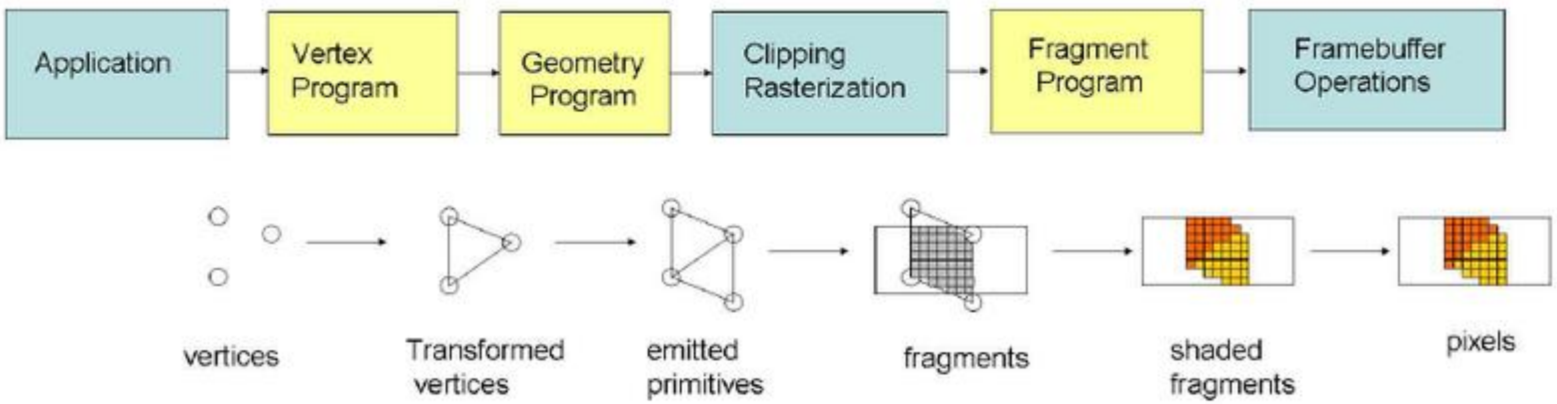
```
glfwInit();
 GLFWwindow* window = glfwCreateWindow(800, 600, "Triangles", NULL, NULL);
 glfwMakeContextCurrent(window);

glewInit();
init();
while (!glfwWindowShouldClose(window))
{
        display();
        glfwSwapBuffers(window);
        glfwPollEvents();
}

glfwDestroyWindow(window);
 glfwTerminate();
```

Break

Next: Textures

Graphic pipeline recap

- Graphic pipeline recap



Core vs Compatibility

Core Profile:

1. The core profile represents a more modern and strict version of OpenGL.
2. It is designed to encourage the use of more recent OpenGL features and deprecates older features and functionality that are considered outdated or inefficient.
3. In the core profile, deprecated features from older versions of OpenGL are no longer available. This forces developers to write code using modern OpenGL techniques, which can be more efficient and maintainable.
4. Core profiles are typically preferred for new OpenGL applications or when updating existing code to take advantage of modern OpenGL features.

Compatibility Profile:

1. The compatibility profile is designed to be more permissive and provides compatibility with older OpenGL versions.
2. It includes deprecated features from earlier versions of OpenGL, which can make it easier to port older OpenGL applications to newer systems without extensive code changes.
3. The compatibility profile is not recommended for new applications or modern OpenGL development because it encourages the use of older, less efficient techniques.

OpenGL Extensions

GL_ARB and GL_EXT Extensions:

Some extensions are promoted to become part of the core OpenGL specification, and they are prefixed with "GL_ARB" (for "Architecture Review Board") or "GL_EXT" (for "Extension"). These extensions are widely adopted and available on many systems.

Extension Registry:

The OpenGL Extension Registry, maintained by the Khronos Group (the organization responsible for OpenGL), provides a comprehensive list of all OpenGL extensions. Developers can refer to this registry to find information about specific extensions, including their specifications and usage.

State and Objects

- OpenGL is by itself a large state machine: a collection of variables that define how OpenGL should currently operate. The state of OpenGL is commonly referred to as the OpenGL **context**. When using OpenGL, we often change its state by setting some options, manipulating some buffers and then render using the current context.

- The OpenGL libraries are written in C and allows for many derivations in other languages, but in its core it remains a C-library. Since many of C's language-constructs do not translate that well to other higher-level languages, OpenGL was developed with several abstractions in mind. One of those abstractions are objects in OpenGL.
  An object in OpenGL is a collection of options that represents a subset of OpenGL's state.

Hello Triangle in SDL

- GLFW version in Lab 5. We take a look at the SDL alternative today.
- What we will still need:

- glew

- SDL2

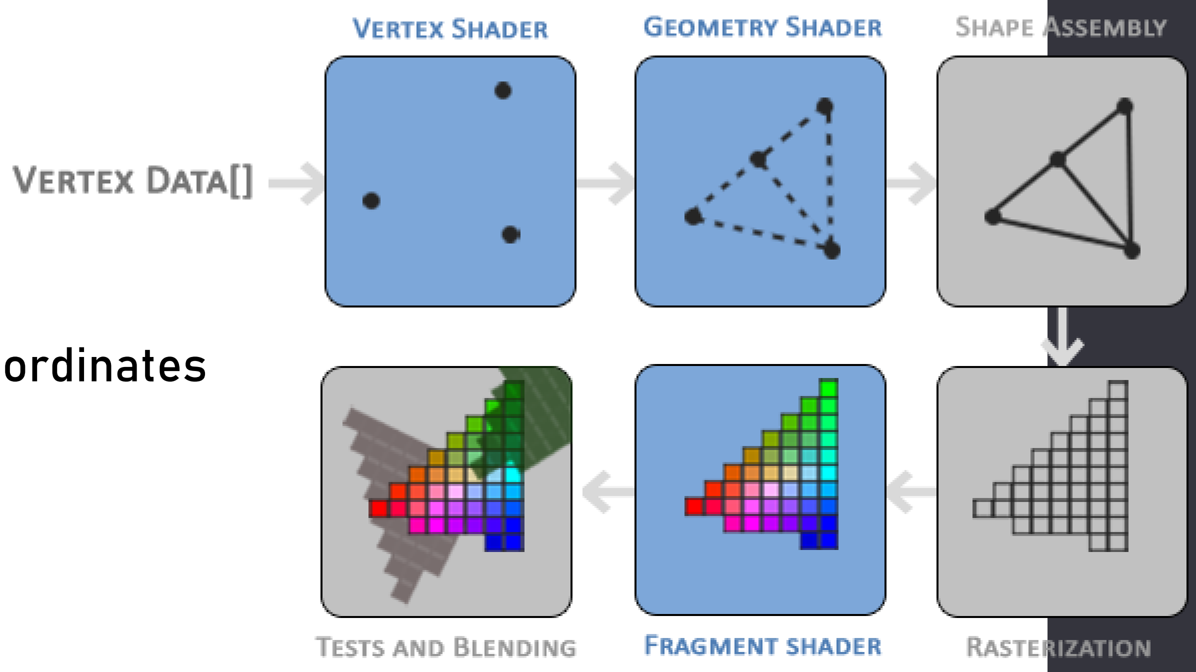
- loadshaders.cpp and loadshaders.h

- vertexShader.vert and fragmentShader.frag

Hello Window Main Processes

- Initialize GLFW/SDL
- Create a window object
- Use GLEW to appropriate the proper extensions for Windows object by loading addresses of OpenGL function pointers
- Determine size of rendering window (can be different from window size) – this is the viewport
- Coordinates in viewport are in normalized coordinates (more on that later)
- Register a callback function for resizing the viewport if window resizes
- Render Loop – similar to game loop
- Termination
- Optionally input processing?

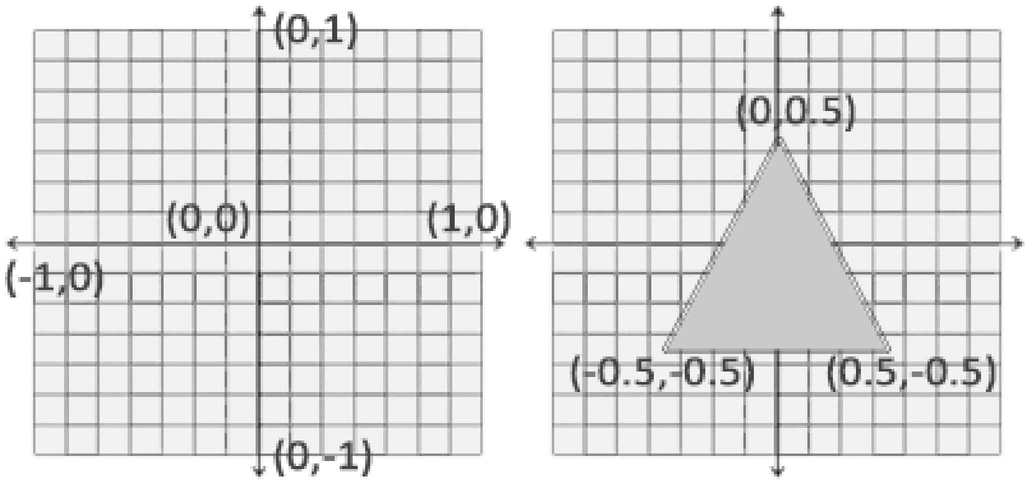
Hello Triangle



- Large part of work for OpenGL is converting 3D coordinates to 2D pixels on screen.
- Mandatory shaders – vertex and fragment
- Vertex is a collection of data per 3D coordinate
- Vertex shader processes vertices
- Rasterization maps the primitive to final screen, performs clipping
- Fragment shader calculates final color of a pixel
- There are more shaders, but those covered in next module.
- Final object goes through alpha test and blending stage.

Normalized Device Coordinates

```
float vertices[] = {
-0.5f, -0.5f, 0.0f,
0.5f, -0.5f, 0.0f,
0.0f, 0.5f, 0.0f };
```



Vertex Buffer Object

- To send input to the vertex shader from our cpp program:

- create memory on GPU for vertex data

- configure how OpenGL should interpret the memory

- specify how to send the data to the graphics card

This is done via vertex buffer objects (VBO) to store vertices in the GPU's memory. It is used to send large batches of data in simultaneously to the graphics card.

Vertex Buffer Object

- Each buffer has a unique ID. Generate this ID using glGenBuffers.
- For vertex buffer object, it is called the GL_ARRAY_BUFFER, so we have to bind the VBO to the GL_ARRAY_BUFFER using glBindBuffer
- Think of it as declaring a variable ID and binding it to a memory space buffer.
- For assignment, use glBufferData which copies the defined vertex data into the buffer's memory with the following parameters:

1) First parameter is the VBO bound to the GL_ARRAY_BUFFER
2) Second parameter is the size of the data in bytes, so we use sizeof()
3) Third parameter is the actual data (from variable or from file) we want to send
4) Last parameter is how the graphics card manages the data:
    1) GL_STREAM_DRAW: the data is set only once and used by the GPU at most a few times.
    2) GL_STATIC_DRAW: the data is set only once and used many times.
    3) GL_DYNAMIC_DRAW: the data is changed a lot and used many times

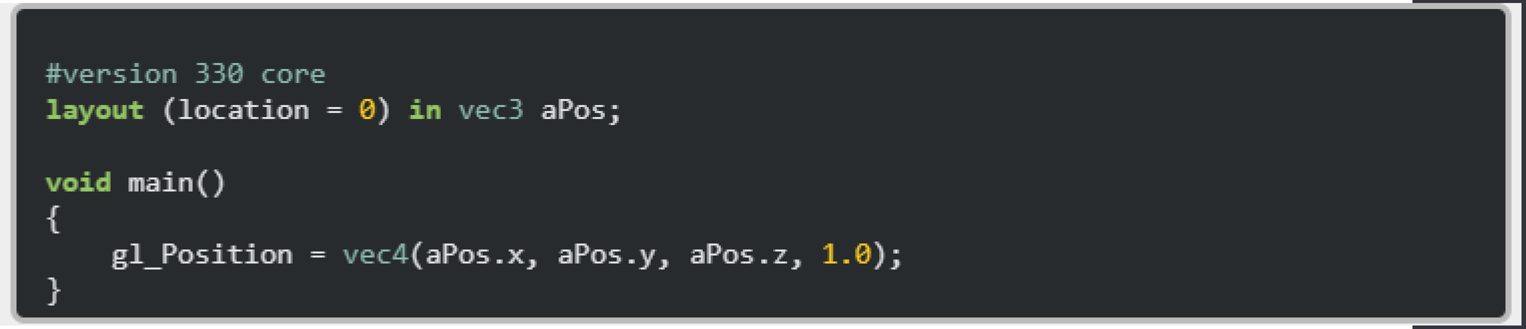
For a static triangle, we use GL_STATIC_DRAW as the triangle vertices do not change but is used a lot in every loop

Vertex and Fragment Shaders

- Written in GLSL
- Begin with version declaration
- Data type is either vec3 or vec4

Vertex Shader

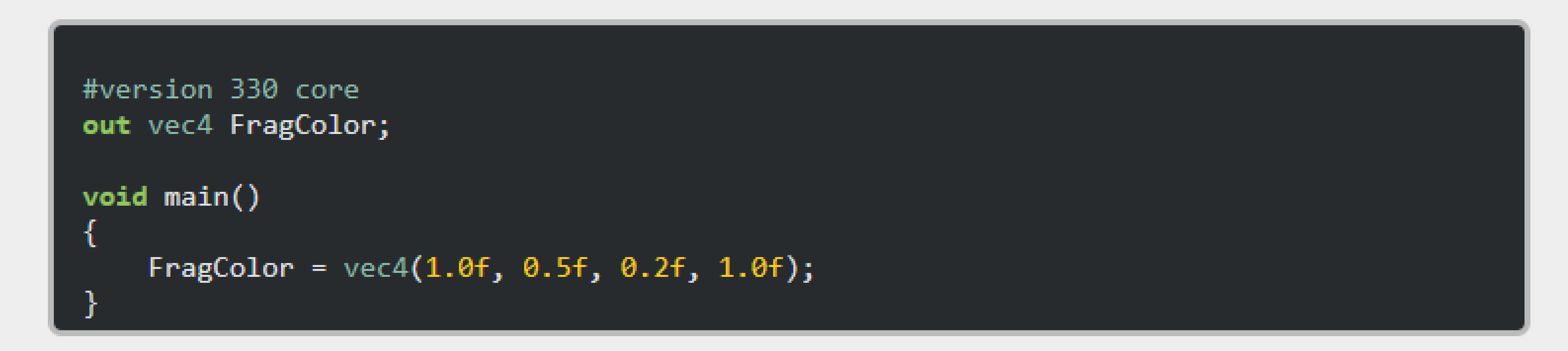
- Set the location of the input variable via layout (location=0)
- Assign the position data to the gl_Position variable which is a vec4.



```
#version 330 core
layout (location = 0) in vec3 aPos;

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

Fragment Shader

- Calculates colour output of the pixels after rasterization
- The output variable is called FragColor which is a vec4 variable.



```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

Since position coordinates are 3D, and colour RGB is also 3 pieces of data, the final param of vec4 is set as 1.0.

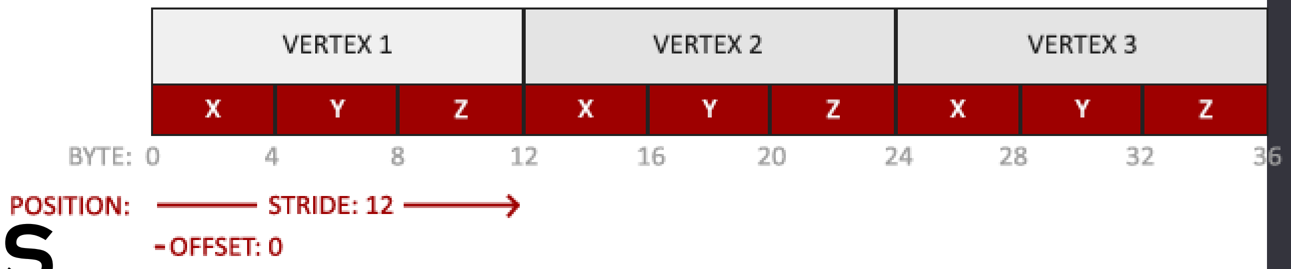
You can follow learnopengl's primitive of loading shaders, but for labs it is given to you in loadshaders.cpp which loads more than 2 shaders.

Vertex Attributes

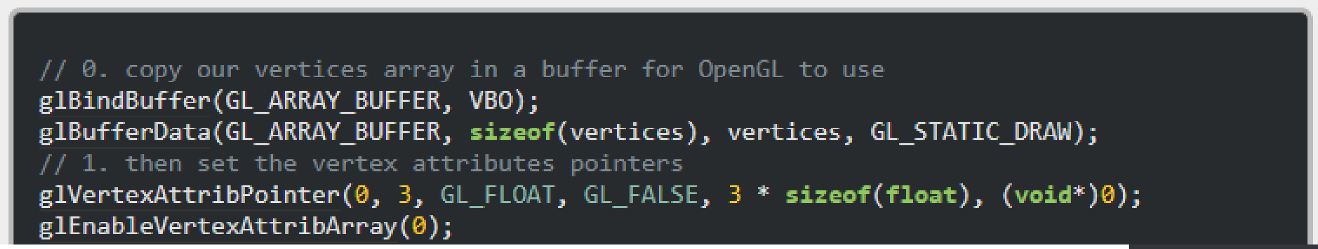
- Back to our CPP
- Link vertex attributes using glVertexAttribPointer – allows us to tell OpenGL how to interpret vertex data per vertex attribute. It has 6 params

1) Vertex attribute to configure – linked to layout location in vertex shader, 0 in this case.
2) Size of vertex attribute. Vec3 has value 3.
3) Type of data, here we use GL_Float for vec
4) Normalize data or not, either GL_FALSE or GL_TRUE. Not relevant so keep at false.
5) Stride – space between consecutive vertex attributes. We have 3 coordinates, so it's exactly 3 times the size of a float. See diagram on top left.
6) Offset of the position data's beginning in the buffer. Since we start at the beginning we set at 0. Requiress a void* cast.

- Finally use glEnableVertexAttribArray to enable vertex attributes as it is disabled by default. Input parameter is the location.
- Drawing an object in OpenGL should look like the code snippet to the right.



- The position data is stored as 32-bit (4 byte) floating point values.
- Each position is composed of 3 of those values.
- There is no space (or other values) between each set of 3 values. The values are tightly packed in the array.
- The first value in the data is at the beginning of the buffer.

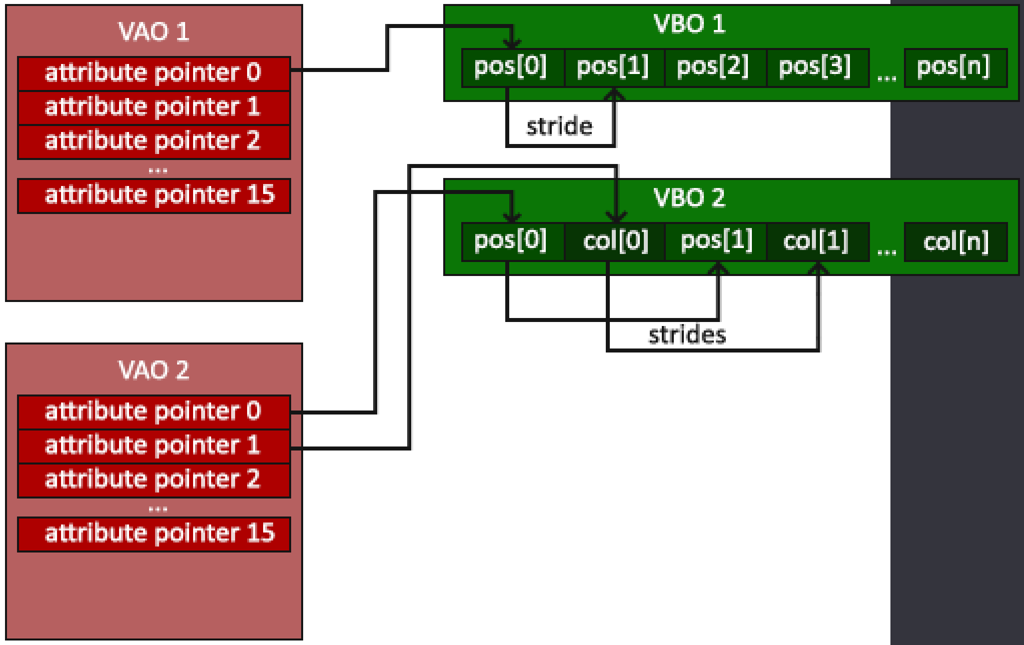


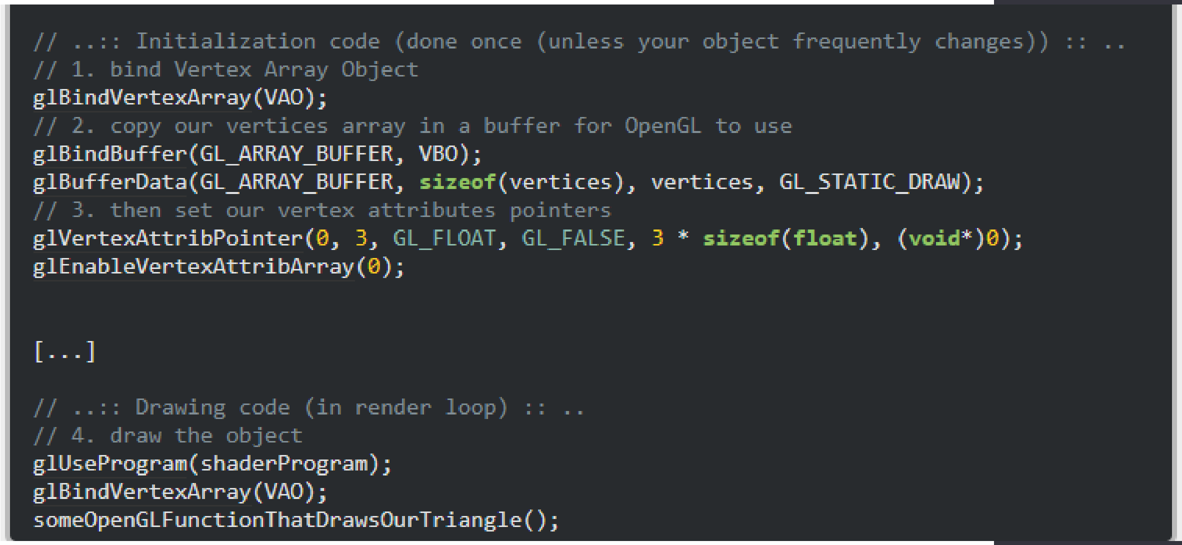
```
// 0. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Each vertex attribute takes its data from memory managed by a VBO and which VBO it takes its data from (you can have multiple VBOs) is determined by the VBO currently bound to `GL_ARRAY_BUFFER` when calling glVertexAttribPointer. Since the previously defined `VBO` is still bound before calling glVertexAttribPointer vertex attribute 0 is now associated with its vertex data.

Vertex Array Object

- Previous slide's step is to draw a single object. If you have 5 vertex attributes for 100s of different objects it will get very repetitive very quickly.
- Enter the vertex array object (VAO). Think of it as an array of vertex buffers. Each index within a VAO is a VBO.
- VAOs can also reuse VBOs, meaning a VBO can be used in more than one VBO. See example diagram above.
- The process to generate a VAO is similar to VBOs. Generate the ID using glGenVertexArrays and then bind it usin glBindVertexArray.
- Then bind/configure the VBO to the attribute pointer and unbind the VAO for later use. Rebind it again when we want to draw the object.

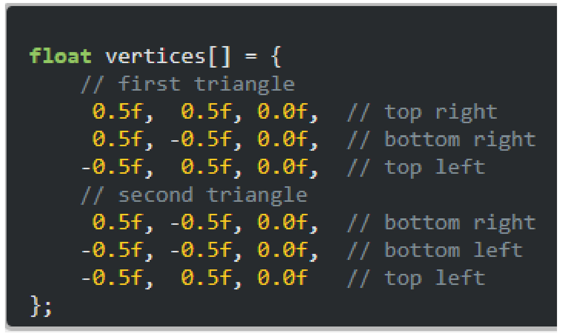




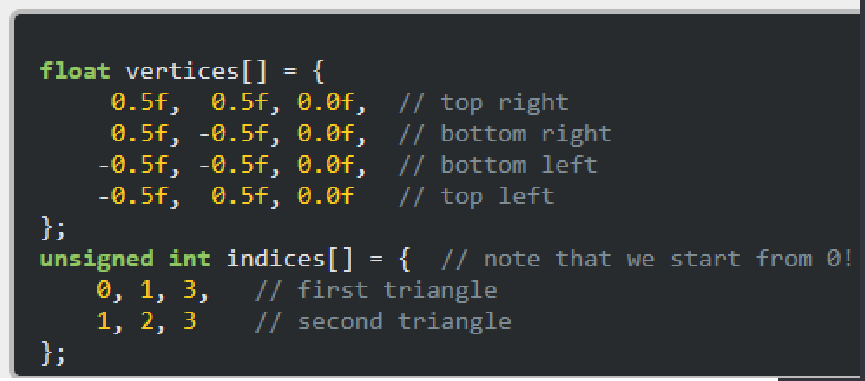
```
// ..:: Initialization code (done once (unless your object frequently changes)) :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. then set our vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);


[...]

// ..:: Drawing code (in render loop) :: ..
// 4. draw the object
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
```

Element Buffer Objects (EBO)

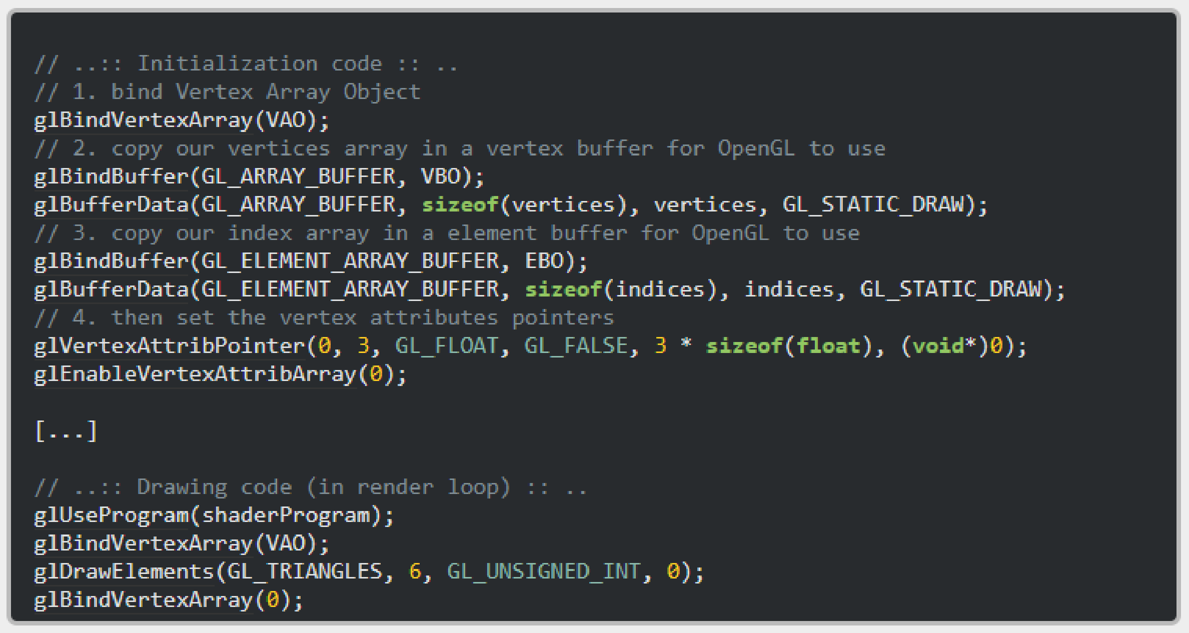
- To draw a quad, it consists of 2 triangles.
- Initialize the vertices of 2 triangles as the right.
- There is overlap!
- Use EBOs as a buffer to store indices once, then let OpenGL decide which ones to draw.
- Also known as indexed drawing.
- The bottom code utilizes EBO to draw a quad.
- Initialization and drawing should look like below:



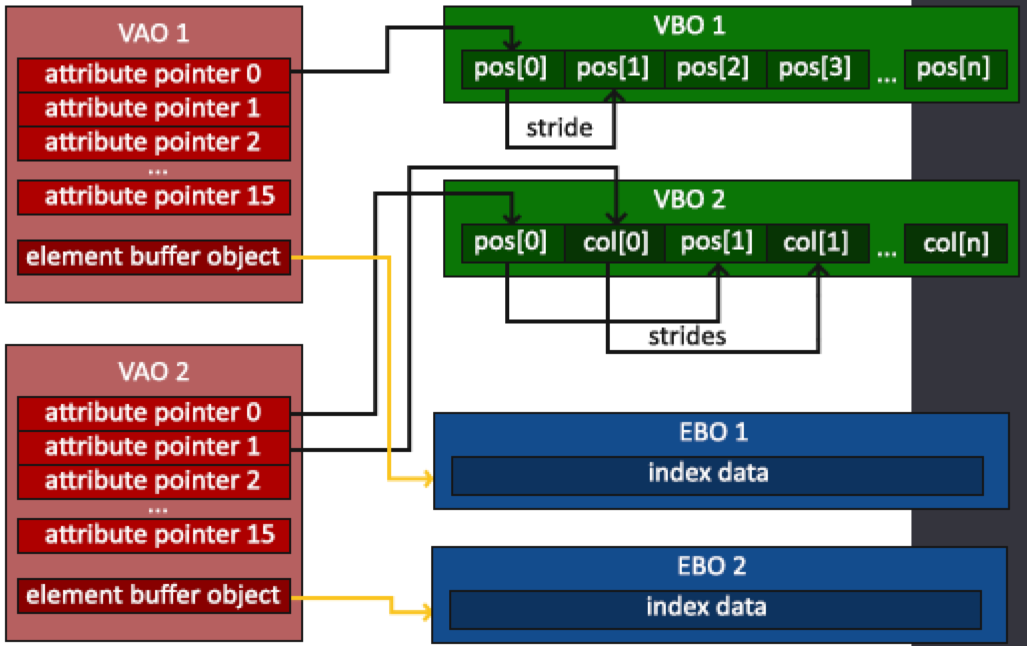
```
float vertices[] = {
    // first triangle
     0.5f,  0.5f, 0.0f,  // top right
     0.5f, -0.5f, 0.0f,  // bottom right
    -0.5f,  0.5f, 0.0f,  // top left
    // second triangle
     0.5f, -0.5f, 0.0f,  // bottom right
    -0.5f, -0.5f, 0.0f,  // bottom left
    -0.5f,  0.5f, 0.0f   // top left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



```
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 we start from 0!
    0, 1, 3,   // first triangle
    1, 2, 3    // second triangle
};
```

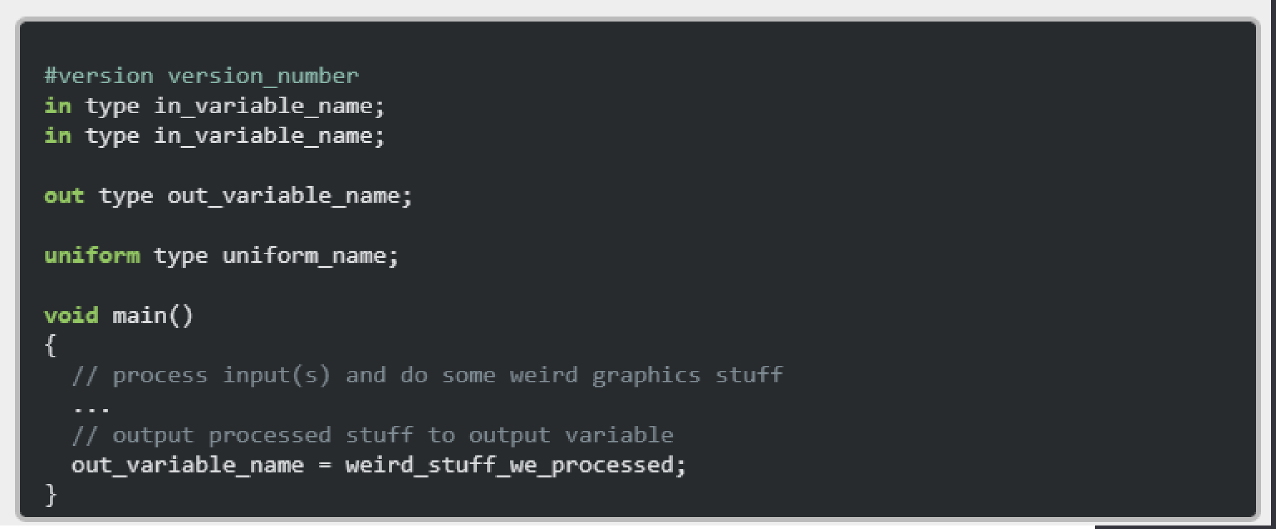


```
// ..:: Initialization code :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a vertex buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. copy our index array in a element buffer for OpenGL to use
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
// 4. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

[...]

// ..:: Drawing code (in render loop) :: ..
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
```



Shaders

- Written in GLSL – C-like language
- Always begin with version declaration, list of input output variables, uniforms and then main function which serves as main entry point.
- Uniform are like global variables – unique per shader program object and can be accessed from any shader at any stage



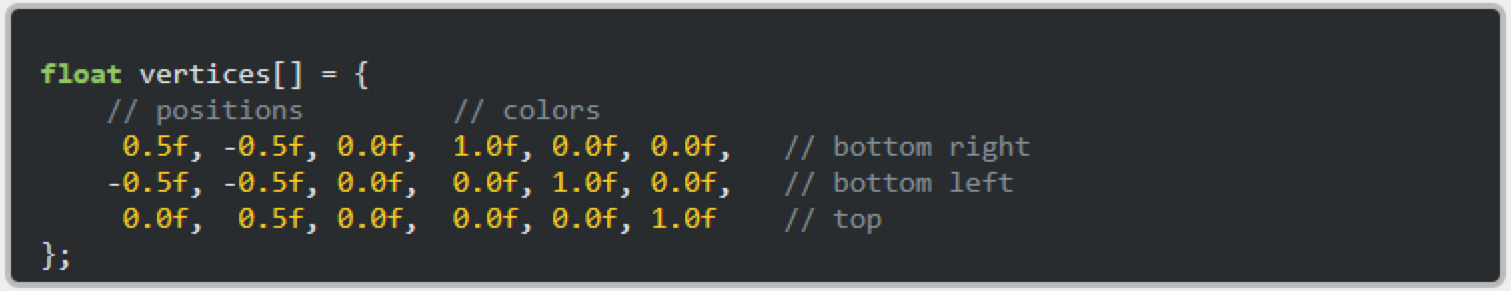
```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
  // process input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

Shader Data Types

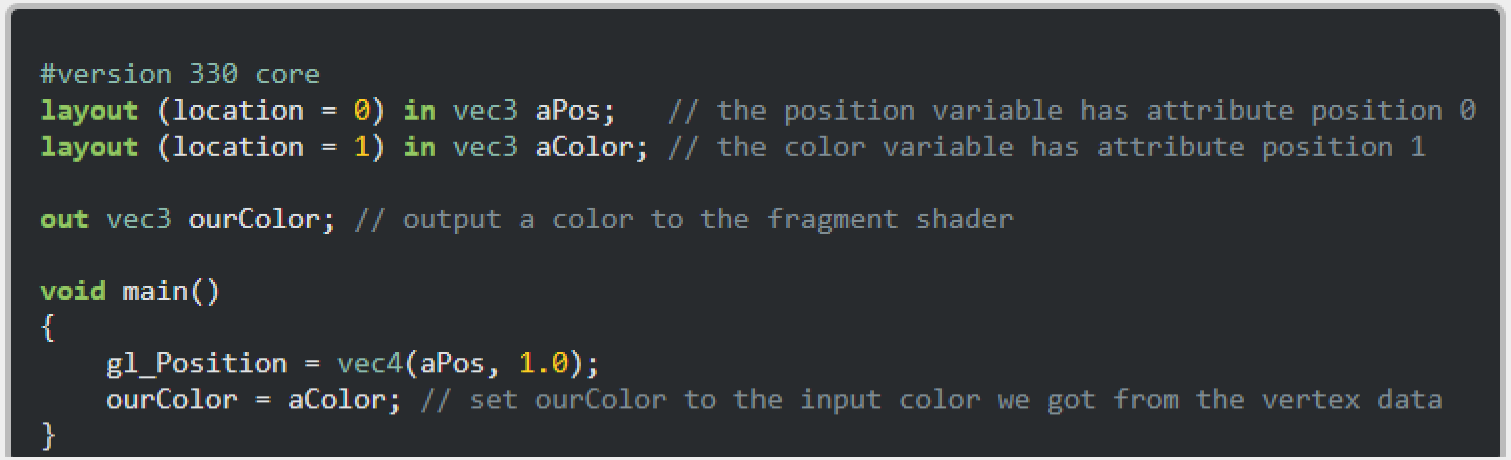
- Vectors – 2,3 or 4 component containers
- Access components via dot operator, e.g. vec.x,vec.y,vec.z and vec.w for a vec4
- GLSL allows xyzw for 3D coordinates, rgba for colours and stpq for texture coordinates using the same components
- Swizzling is also allowed – any combination of 4 letters to create a new vector of the same type as long as the original vector has those components, i.e. vec.z for vec2 is not allowed.

Adding Colours to Vertices

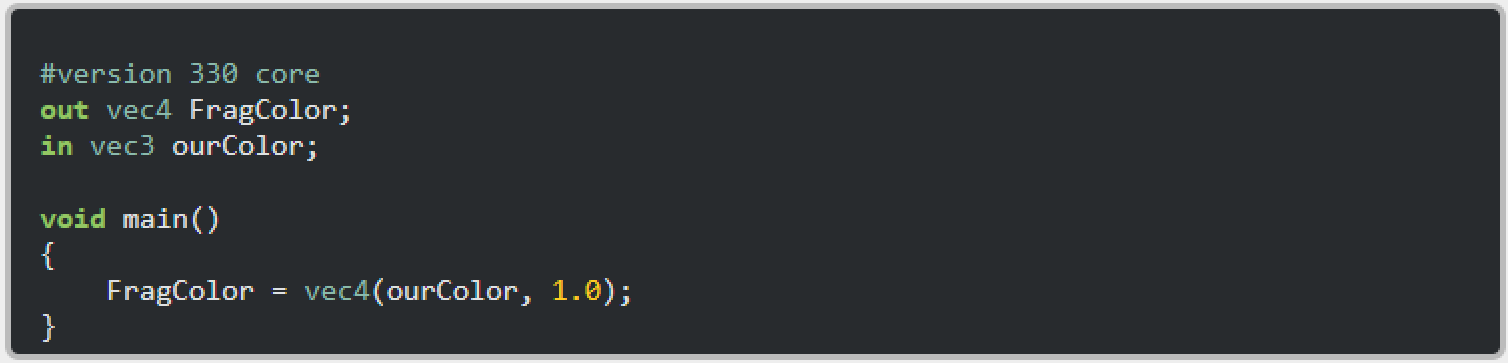
- First 3 floats for xyz coordinates, next 3 floats for RGB data.
- In our vertex shader, we pass in through 2 locations 0 and 1 – see glVertexAttribPointer on next slide.
- Pass from aColor in vertex shader to ourColor in fragment shader to determine colours based on vertex position.



```
float vertices[] = {
    // positions         // colors
     0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,   // bottom left
     0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f    // top
};
```



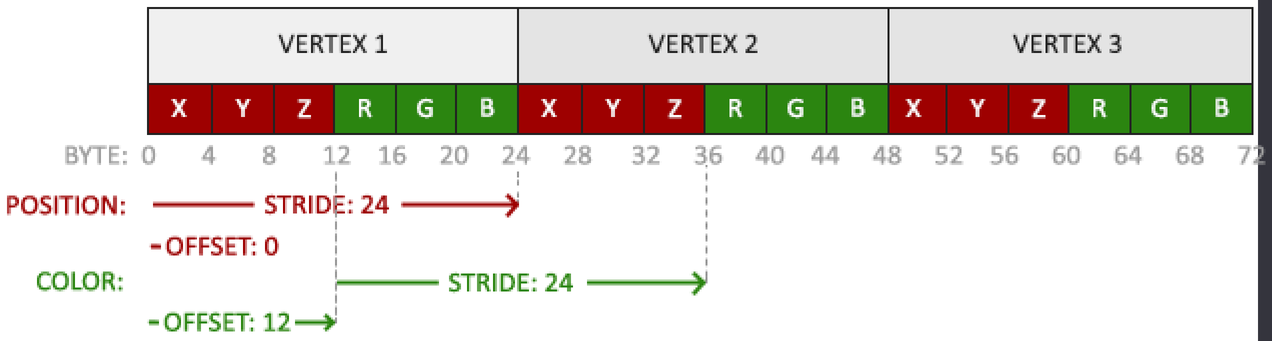
```
#version 330 core
layout (location = 0) in vec3 aPos;   // the position variable has attribute position 0
layout (location = 1) in vec3 aColor; // the color variable has attribute position 1

out vec3 ourColor; // output a color to the fragment shader

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor; // set ourColor to the input color we got from the vertex data
}
```

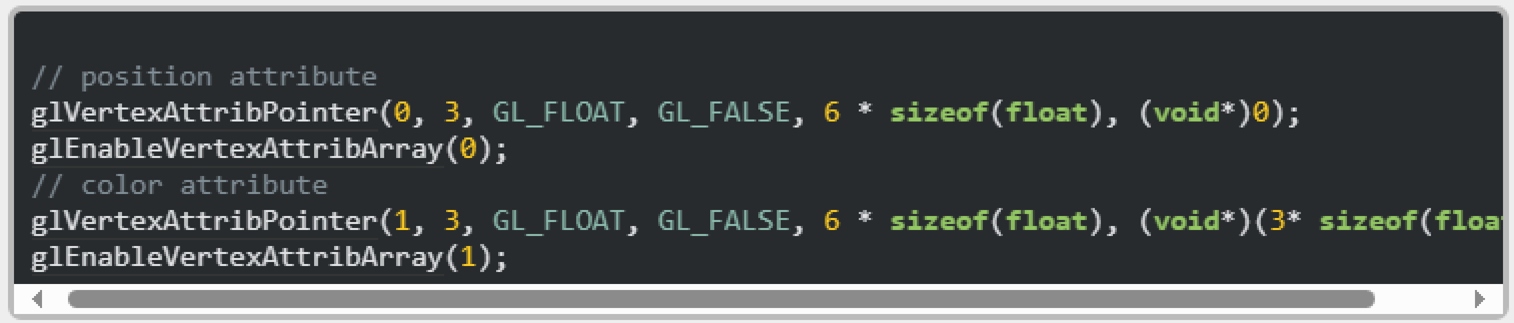


```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;

void main()
{
    FragColor = vec4(ourColor, 1.0);
}
```

Adding Colours to Vertices (2)

- First position attribute remains the same.
- The second colour attribute has to recalculate the stride value – 6 floats to the right, starting offset at 3 floats.

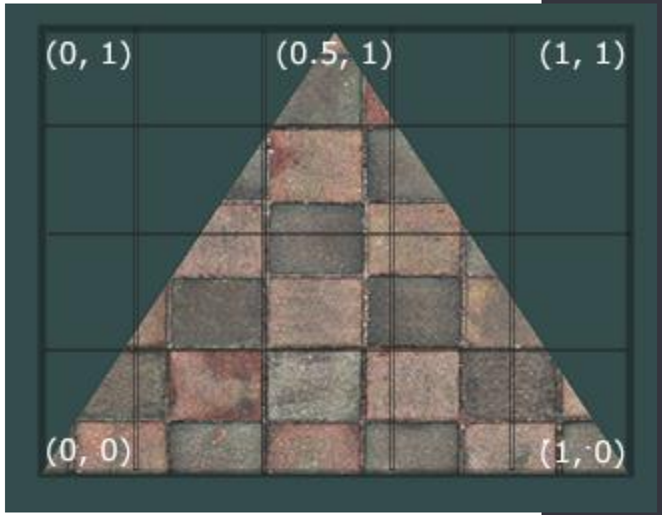

VERTEX 1
VERTEX 2
VERTEX 3
X Y Z R G B X Y Z R G B X Y Z R G B
BYTE: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72
POSITION:
STRIDE: 24
OFFSET: 0
COLOR:
STRIDE: 24
OFFSET: 12

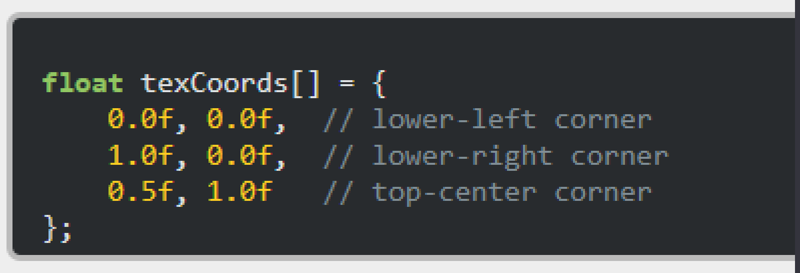


```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(floa
glEnableVertexAttribArray(1);
```

Textures

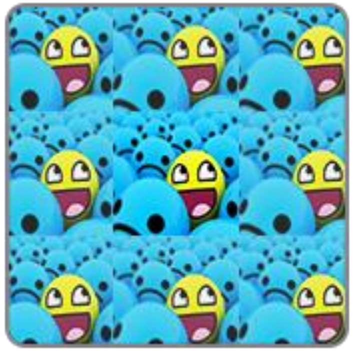
- A 2D image used to add detail to an object – a wrapper around a 3D polygon for exterior
- Textures can also be used to store a large collection of arbitrary data for shaders to process
- To map a texture to a polygon, we tell each vertex of the polygon which part of the texture it corresponds to. In the case of the triangle, see above.
- Each vertex will hav a texture coordinate associated to sample the texture image from.
- Fragment interpolation then does the rest for the other fragments.
- In this example, we pass 3 texture coordinates to the vertex shader, which passes to the fragment shader to interpolate all texture coordinates for each fragment.
- A programmer's job is to tell OpenGL how to sample the textures.



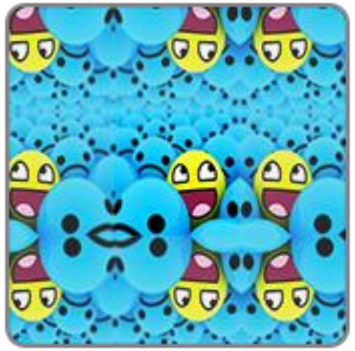


```
float texCoords[] = {
    0.0f, 0.0f,  // lower-left corner
    1.0f, 0.0f,  // lower-right corner
    0.5f, 1.0f   // top-center corner
};
```

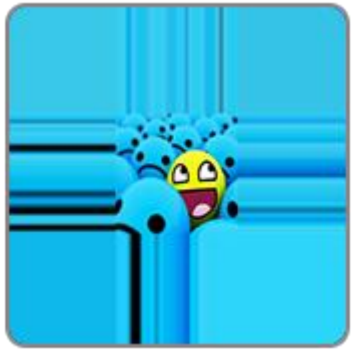
Texture Wrapping



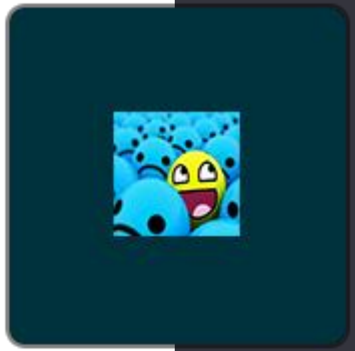
GL_REPEAT



GL_MIRRORED_REPEAT



GL_CLAMP_TO_EDGE

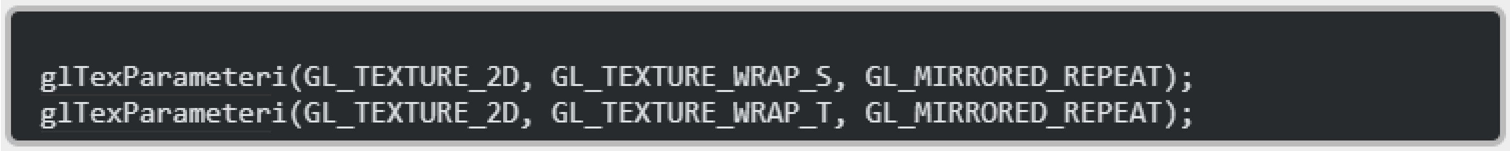


GL_CLAMP_TO_BORDER

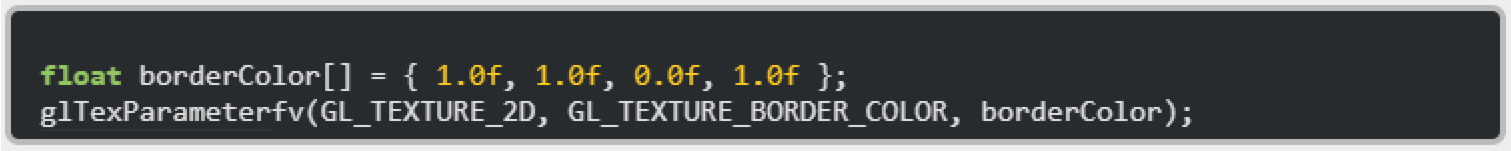
- Sometimes coordinates and texture size does not coincide.
- The default behavior is to repeat texture images.
- Other options include:

- GL_REPEAT: The default behavior for textures. Repeats the texture image.
- GL_MIRRORED_REPEAT: Same as GL_REPEAT but mirrors the image with each repeat.
- GL_CLAMP_TO_EDGE: Clamps the coordinates between 0 and 1. The result is that higher coordinates become clamped to the edge, resulting in a stretched edge pattern.
- GL_CLAMP_TO_BORDER: Coordinates outside the range are now given a user-specified border color.

- Update using glTexParameteri function
- For GL_CLAMP_TO_BORDER we then also need a border colour, which is done using glTexParameterfv as in the example below.

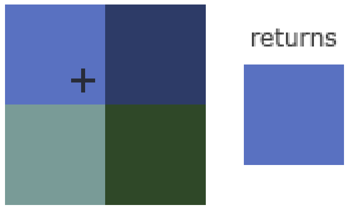


```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

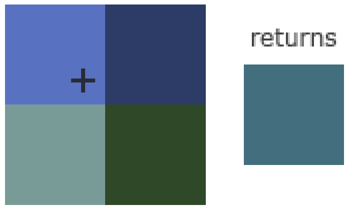


```
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

Texture Filtering

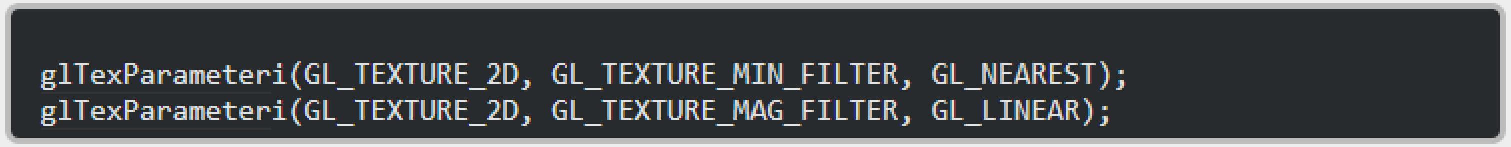


GL_NEAREST

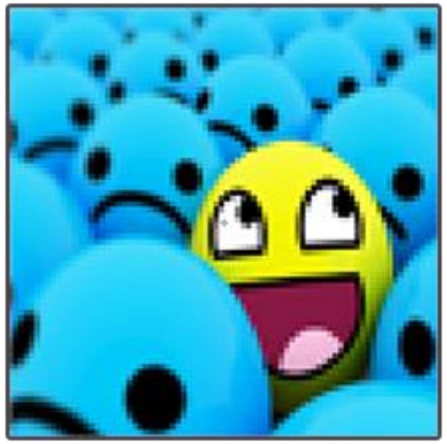


GL_LINEAR

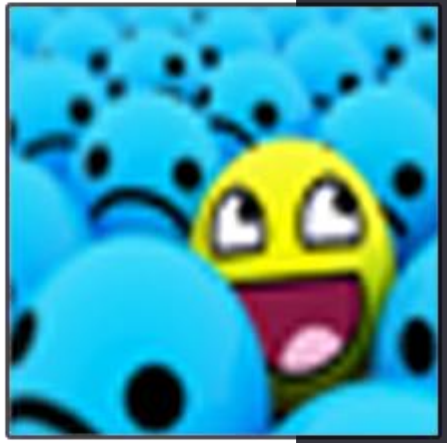
- For large objects with low resolution, use texture filtering.
- 2 options: GL_NEAREST and GL_LINEAR
- GL_NEAREST hits the texel closest to the texture coordinate
- GL_LINEAR (also known as bilinear filtering) takes an interpolated value from neighbouring pixels
- Also use glTexParameteri to set magnifying or minifying operations similar to mapping method.



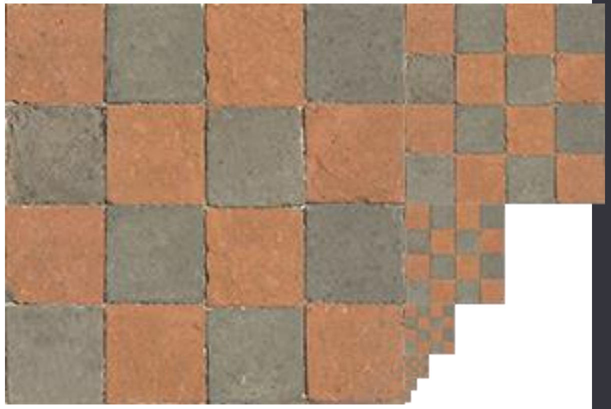
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```



GL_NEAREST



GL_LINEAR

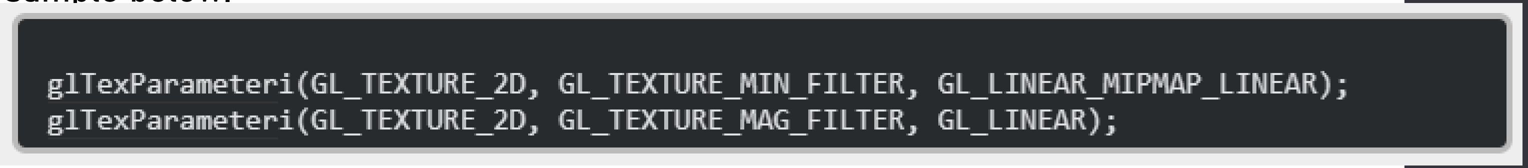


Mipmaps

- For objects far from viewer, texture resolutions will not need to be as high as closer objects.
- Mipmaps are used as a collection of texture images where each subsequent texture is twice as small compared to the previous ones.
- After a different threshold from the viewer, OpenGL will use a different mipmap texture.
- More on this when we talk about transformation matrices to determine distance next week.
- Four options:

1) GL_NEAREST_MIPMAP_NEAREST: takes the nearest mipmap to match the pixel size and uses nearest neighbor interpolation for texture sampling.
2) GL_LINEAR_MIPMAP_NEAREST: takes the nearest mipmap level and samples that level using linear interpolation.
3) GL_NEAREST_MIPMAP_LINEAR: linearly interpolates between the two mipmaps that most closely match the size of a pixel and samples the interpolated level via nearest neighbor interpolation.
4) GL_LINEAR_MIPMAP_LINEAR: linearly interpolates between the two closest mipmaps and samples the interpolated level via linear interpolation.

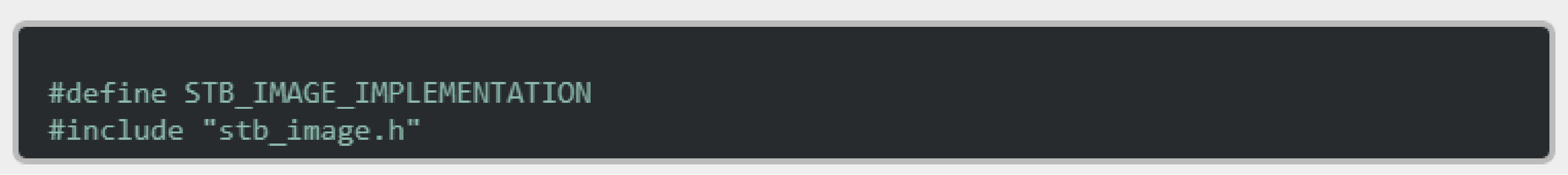
- Similarly use glTexParameteri for mipmaps as the sample below.



```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Loading and Creating Textures

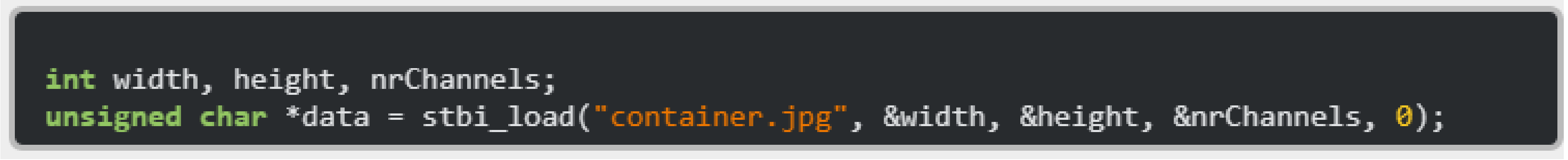
- stb_image.h standardizes an image-loading library for popular graphic formats.
- Library by Sean Barrett.
- Download and place in project folder, then include stb_image.h with the following code:



```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

STB_IMAGE_IMPLEMENTATION turns the header file into a .cpp file. This is required to use STB_Image – linking issue if you don’t place this.

To load an image, use the stbi_load function.



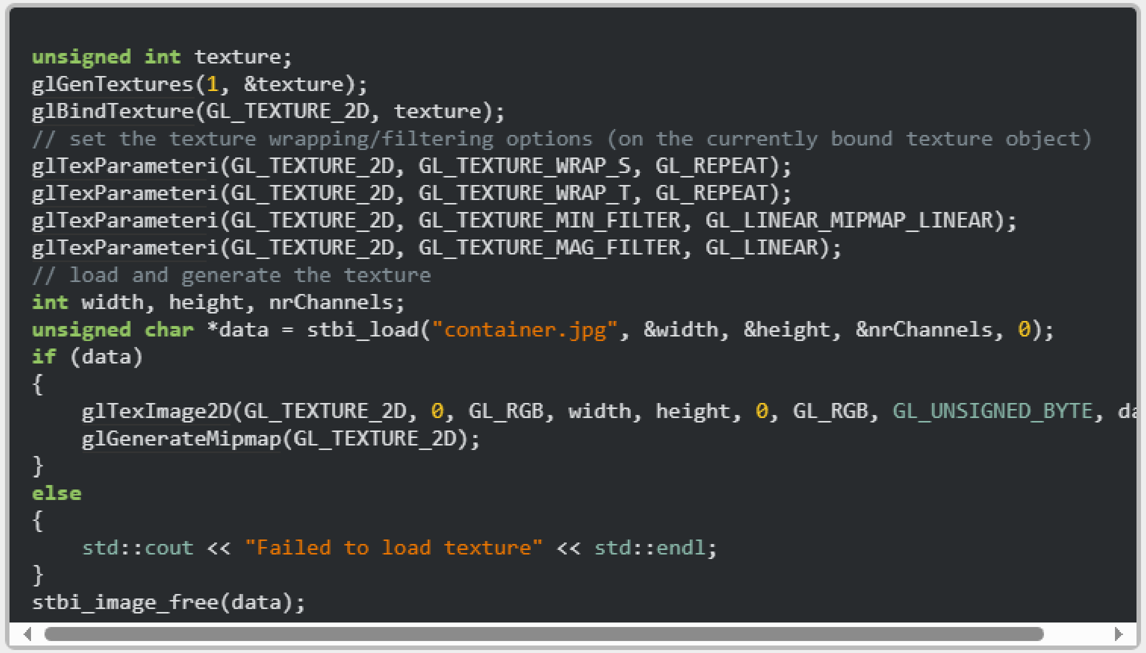
```
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
```

Generating the Texture

- Create an ID.
- Bind the texture.
- Generate the texture using loaded image data with glTexImage2D. The 9 parameters are:

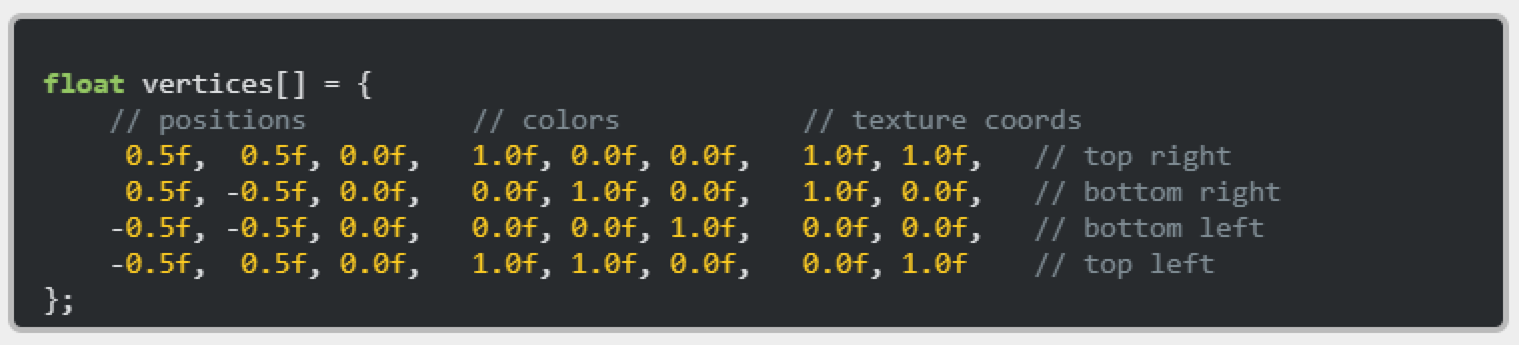
1) GL_Texture_2D generates a texture on the currently bound texture object of the same target
2) Specifies mipmap level, 0 is for base level.
3) Format to store the texture – so only RGB now.
4) Sets the width of resulting texture
5) Sets the height of resulting texture
6) 6th param is always 0 due to legacy issue
7) Format of the source image so RGB values
8) Datatype of the source image, which is GL_UNSIGNED_BYTE as the RGB is stored as charS (bytes)
9) The actual image data

- After glTexImage2D, the currently bound texture has the image attached, but it is only the base-level. So mipmaps need to be added so glGenerateMipmap comes after.
- Lastly free the image using stbi_image_free(data) to free memory.



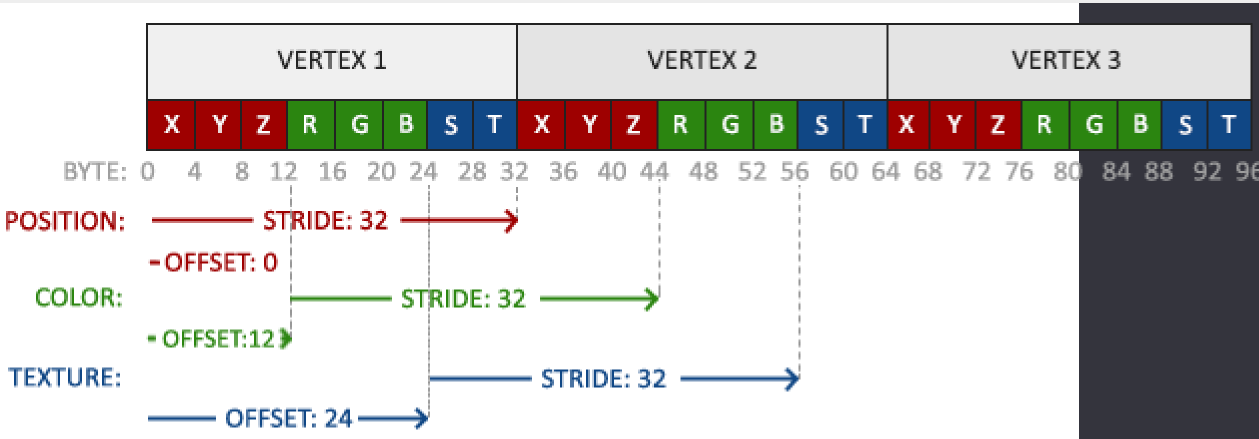
```
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
// set the texture wrapping/filtering options (on the currently bound texture object)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// load and generate the texture
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
if (data)
{
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, da
    glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
    std::cout << "Failed to load texture" << std::endl;
}
stbi_image_free(data);
```

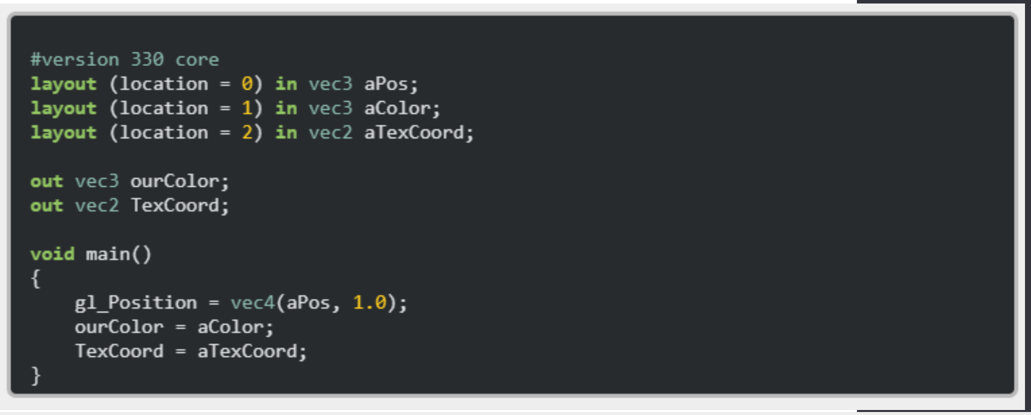
Applying Textures



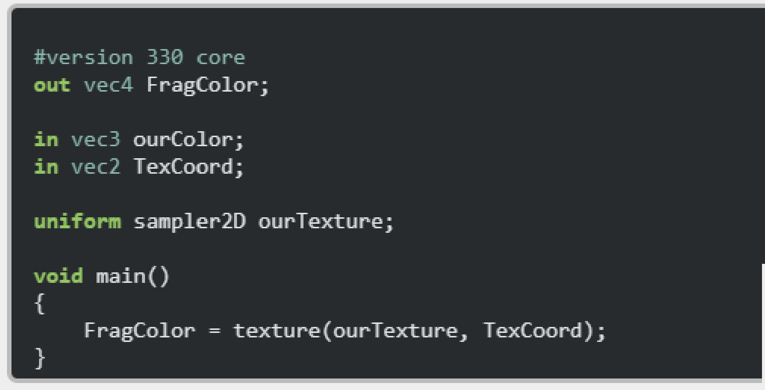
```
float vertices[] = {
    // positions          // colors           // texture coords
     0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f,   // top right
     0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   1.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,   0.0f, 0.0f,   // bottom left
    -0.5f,  0.5f, 0.0f,   1.0f, 1.0f, 0.0f,   0.0f, 1.0f    // top left
};
```

- Set aside 2 more floats for texture coordinates for your vertices array.
- The stride is now 32 bytes – 8 floats at 4 bytes each
- The vertex attribute pointer now becomes glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
- glEnableVertexAttribArray(2);
- Add this to the vertex shader and fragment shader as well.
- Last step is to bind the texture and call glDrawElements.

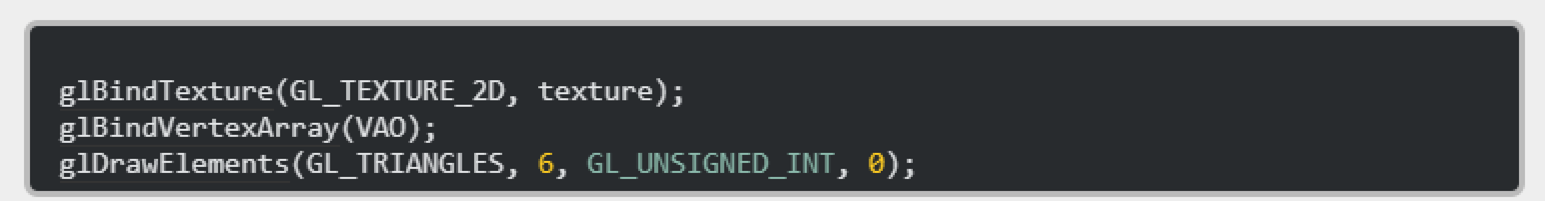

VERTEX 1
VERTEX 2
VERTEX 3
X Y Z R G B S T X Y Z R G B S T X Y Z R G B S T
BYTE: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80 84 88 92 96
POSITION:
STRIDE: 32
OFFSET: 0
COLOR:
STRIDE: 32
OFFSET:12
TEXTURE:
STRIDE: 32
OFFSET: 24



```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;

out vec3 ourColor;
out vec2 TexCoord;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;
    TexCoord = aTexCoord;
}
```



```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture;

void main()
{
    FragColor = texture(ourTexture, TexCoord);
}
```



```
glBindTexture(GL_TEXTURE_2D, texture);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Thanks so far!

Any Questions?

For this Thursday's Lab:

Sign up for Project1 on the DLE!

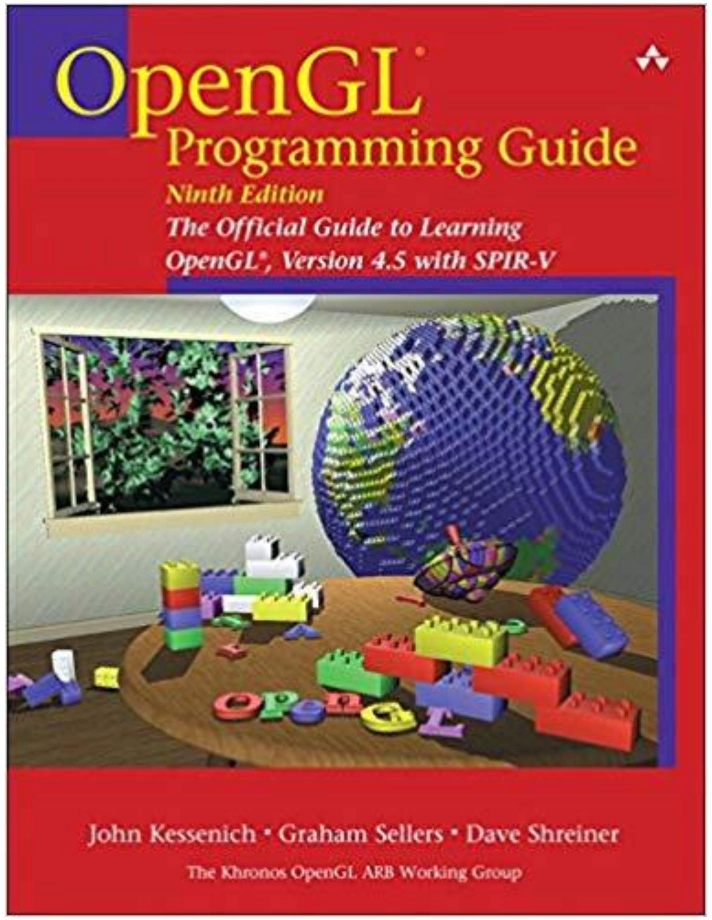
- Update your graphics driver!
- Build the Project!
- Add triangles in places to form a knots & crosses board
- Change the Colour of some of the triangles to green

Reading List

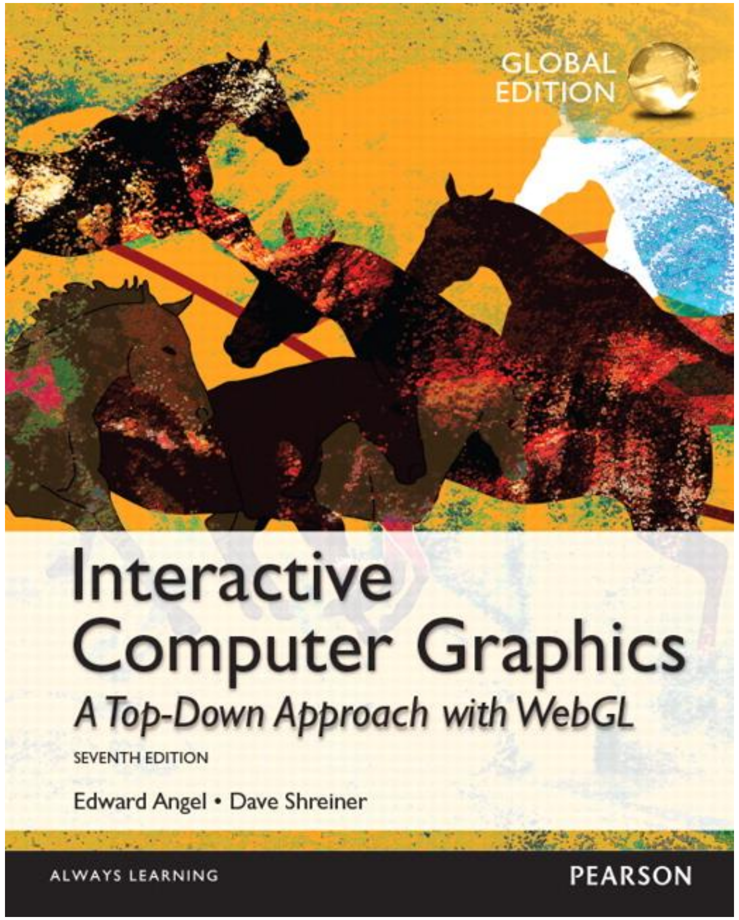
Don't pick older versions of this book!

Older versions of this book work as well

LearnOpenGL.com



OpenGL Programming Guide
Ninth Edition
The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V
John Kessenich · Graham Sellers · Dave Shreiner
The Khronos OpenGL ARB Working Group



GLOBAL EDITION
Interactive Computer Graphics
A Top-Down Approach with WebGL
SEVENTH EDITION
Edward Angel • Dave Shreiner
ALWAYS LEARNING
PEARSON

Some Exercises:

- https://coderbyte.com
- https://www.codewars.com/
- https://leetcode.com