# COSC 4332 Computer Graphics

## Modern OpenGL

Dr. Khaled Rabieh

# Outline

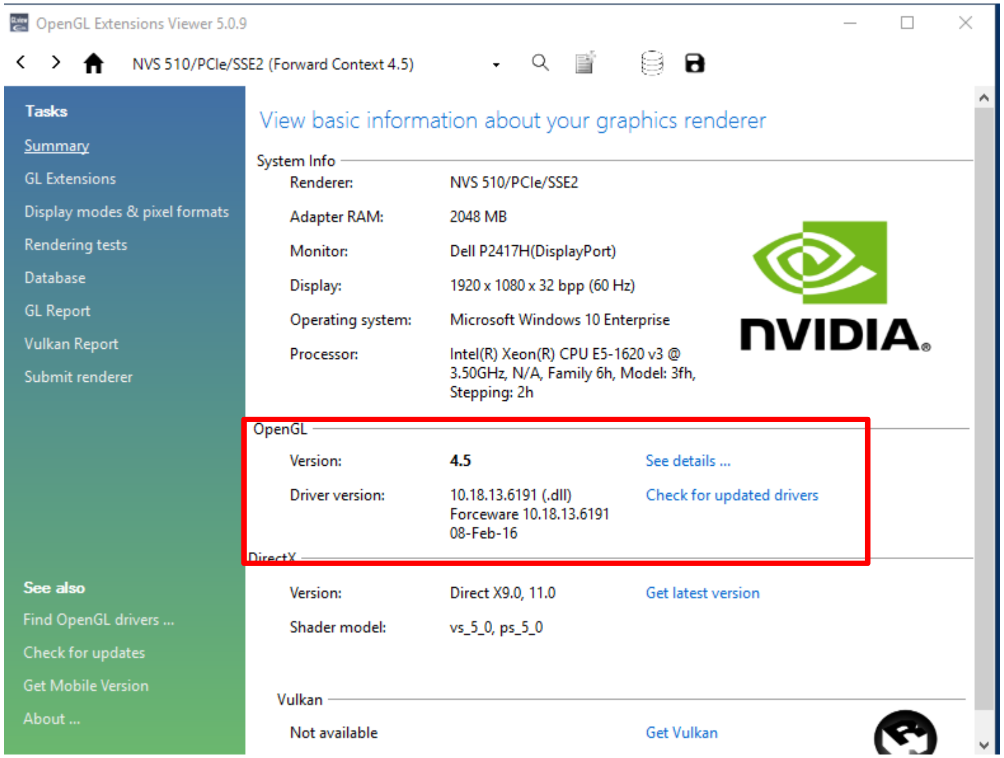
## 1. Introduction to Modern Open GL

- OpenGL rendering Pipeline
- Vertex and Fragment Shaders
- Compiling shaders

# OpenGL Version

OpenGL comes with the system.
You *will* need to **ensure** that you have downloaded and installed a **recent driver for your graphics hardware**.

To know the supported version of OpenGL by your graphics card try this program:
[http://realtech-vr.com/admin/glview](http://realtech-vr.com/admin/glview)




Below 3.1 ???
If yes, you will not be able to proceed

# Initializing OpenGL

There are two phases for initialization:

1. Creating **context**:
   It is like creating a window in an application.
   we’re going to use **GLFW** library.

2. Getting **functions**:
   For loading all available OpenGL extensions and functions automatically.
   we’re going to use **GLEW** library.

## GLFW

Open Source, multi-platform library for **creating windows** with OpenGL contexts and receiving **input** and **events.**
Other libraries: GLUT, FreeGLUT, SDL, SFML.

## GLEW

It loads pointers to OpenGL functions at runtime.
It supports Windows, MacOS X, Linux, and FreeBSD.

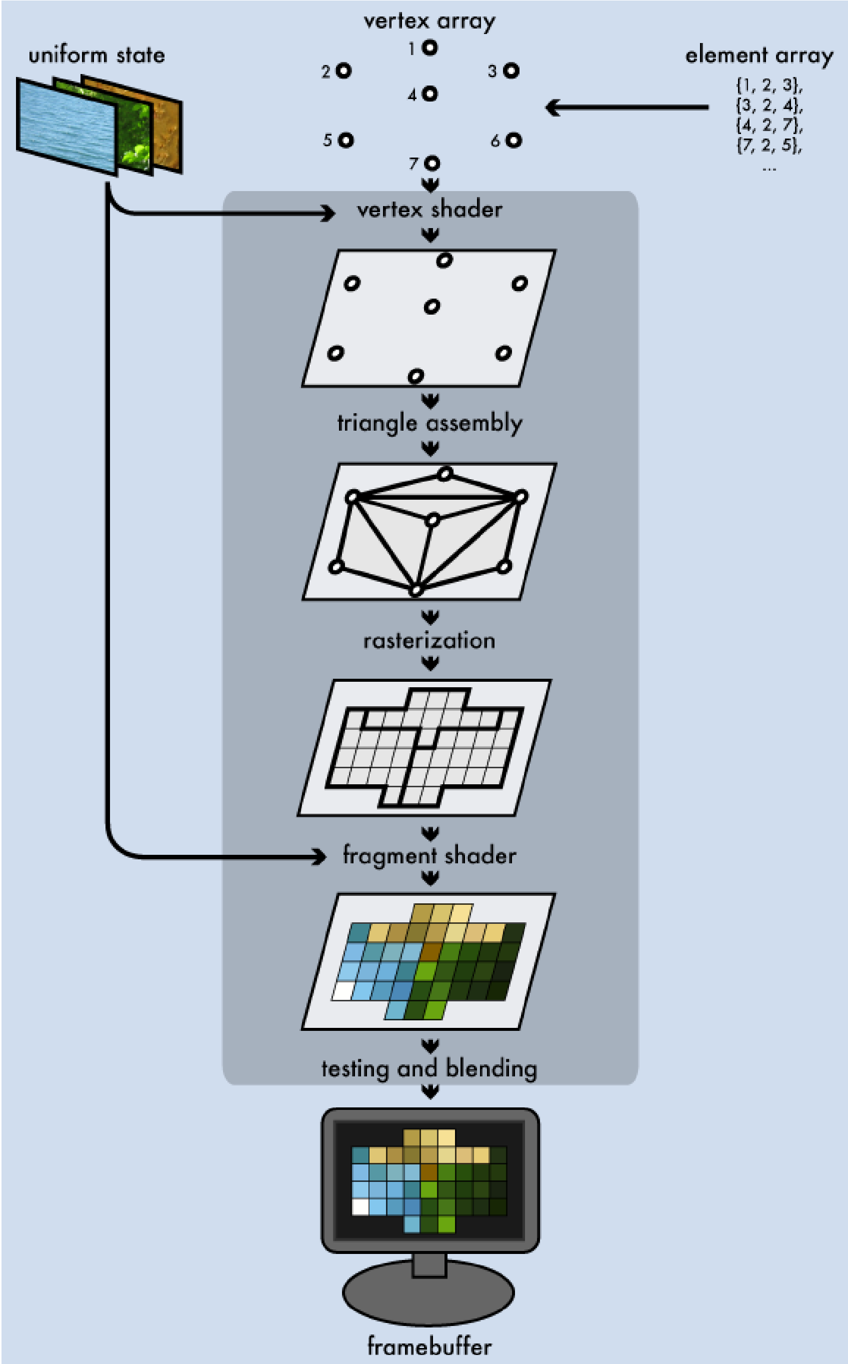
## Math Library for Transformations

There are plenty of them.
You can implement one.
We're going to use: **GLM** library.

# Modern OpenGL Pipeline

## <u>Stage1</u>

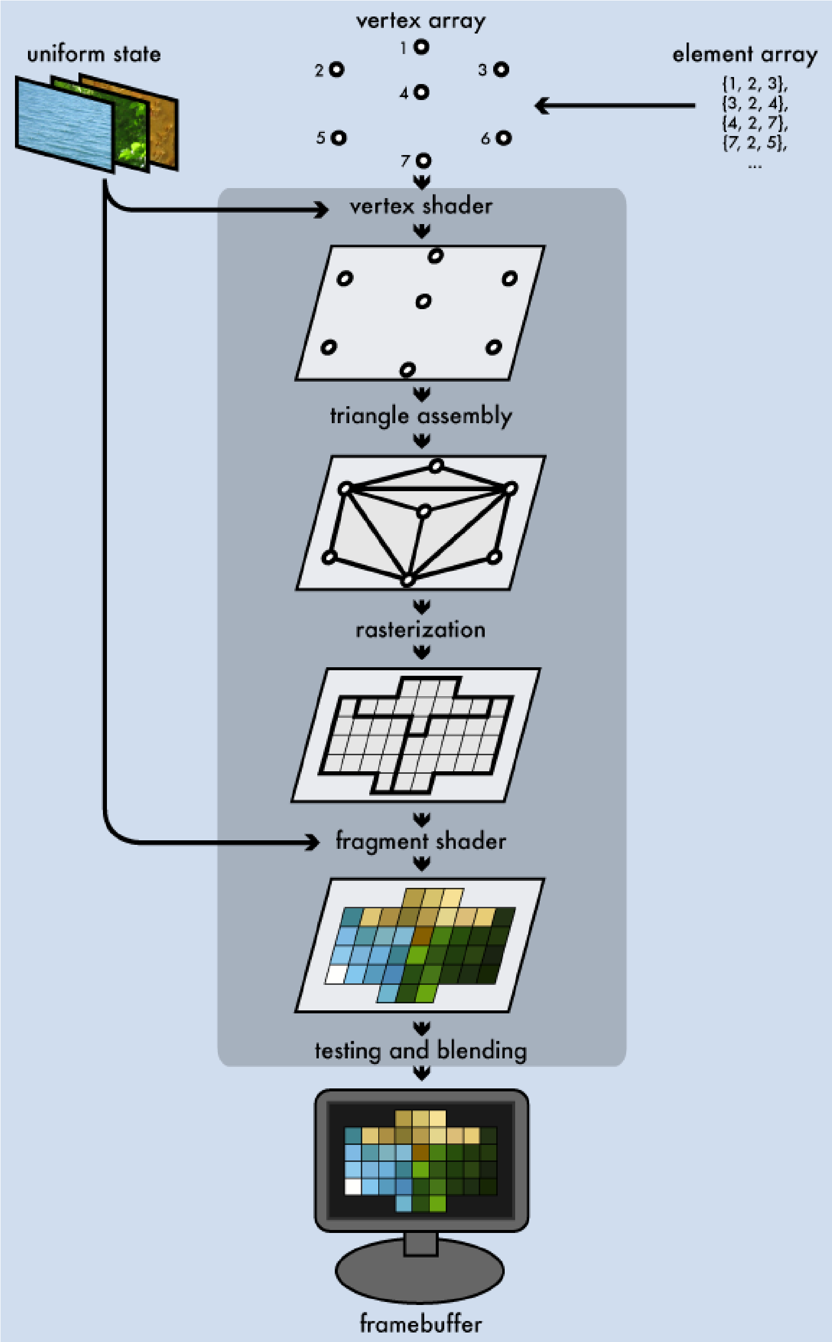
- A rendering job starts its journey through the pipeline in a set of one or more **<u>vertex buffers</u>**, which are filled with arrays of vertex attributes.
- Common vertex attributes include the location of the vertex in 3d space




http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html

# Modern OpenGL Pipeline

- The set of vertex buffers supplying data to a rendering process are collectively called the **vertex array**.
- When a render job is submitted, we can supply an additional **element array**, an array of indexes into the vertex array that select which vertices get fed into the pipeline.

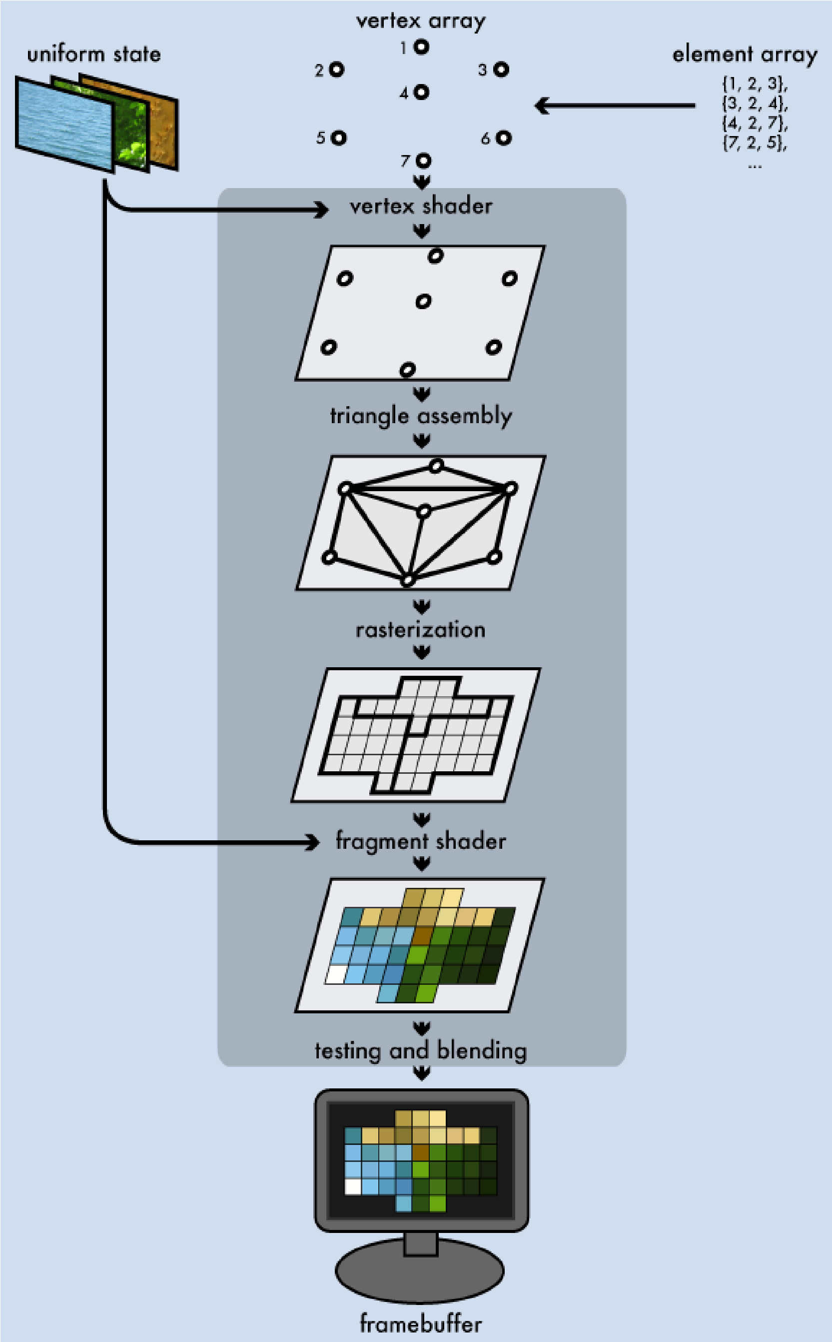



http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html

# The Vertex Shader

<u>Stage 1</u>

- Each vertex in the vertex array runs against it through the **vertex shader**,
- A vertex shader is a program that takes a set of vertex attributes as inputs and outputs a new set of attributes, referred to as **varying** values
- At a minimum, the vertex shader calculates the projected **position** of the vertex in screen space.

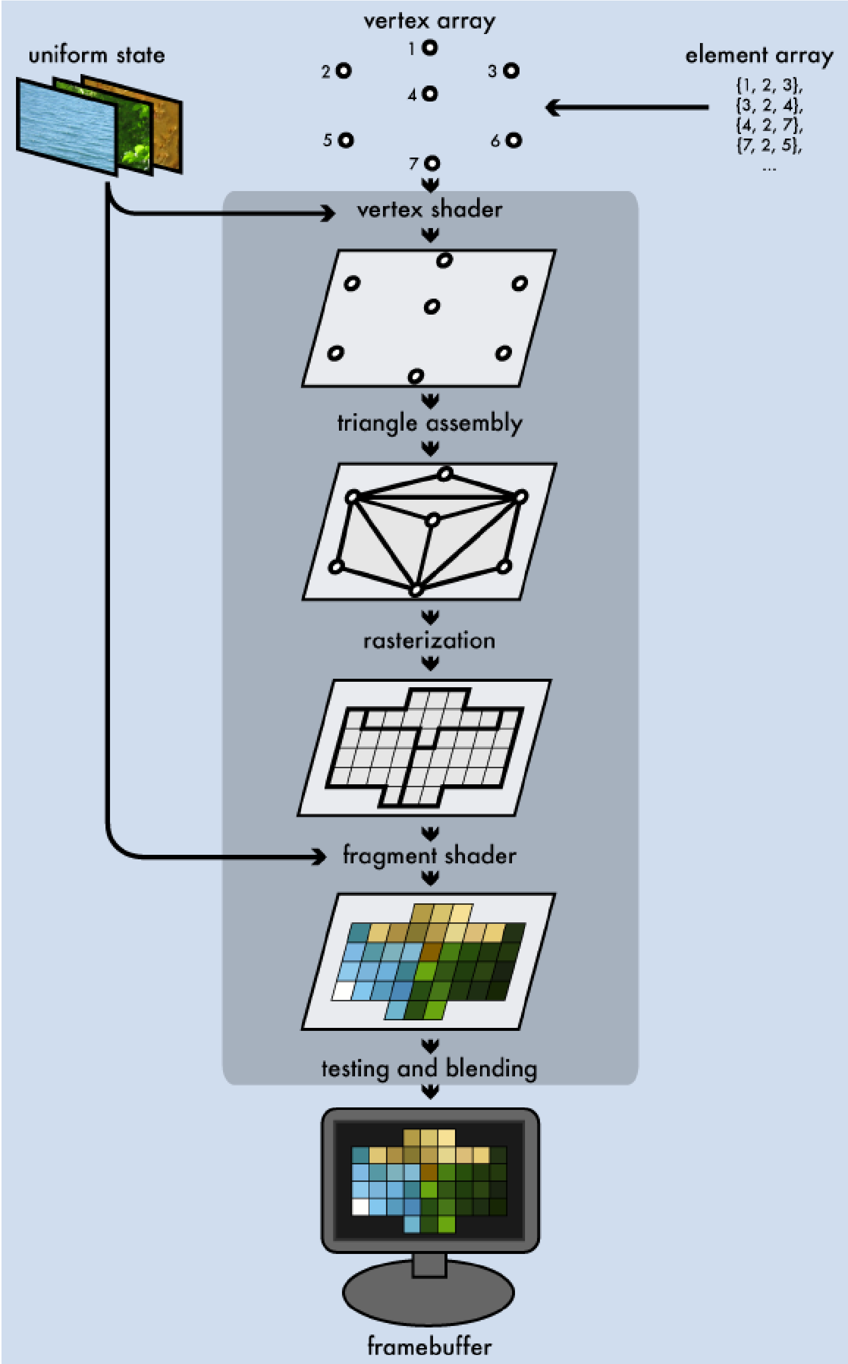


# Triangle Assembly

Stage 2

- The GPU connects the projected vertices to form triangles by the order specified by the element array and grouping them into sets of three.
- The vertices can be grouped in a few different ways:

A. Independent triangles
B. A **triangle strip**, reusing the last two vertices of each triangle as the first two vertices of the next
C. Make a **triangle fan**, connecting the first element to every subsequent pair of elements

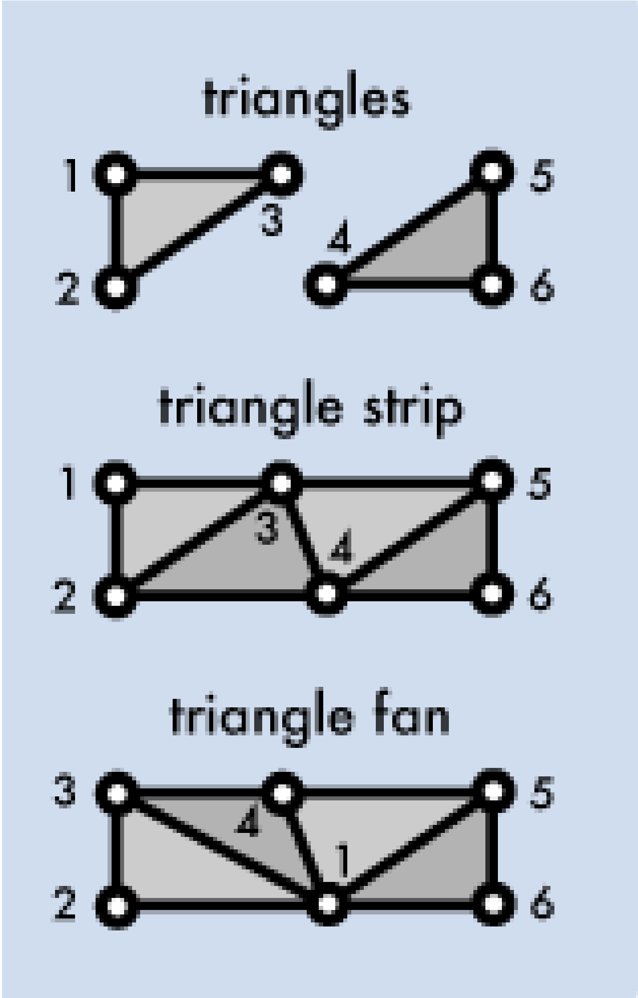


# Triangle Assembly

Stage 2

- The vertices can be grouped in a few different ways:

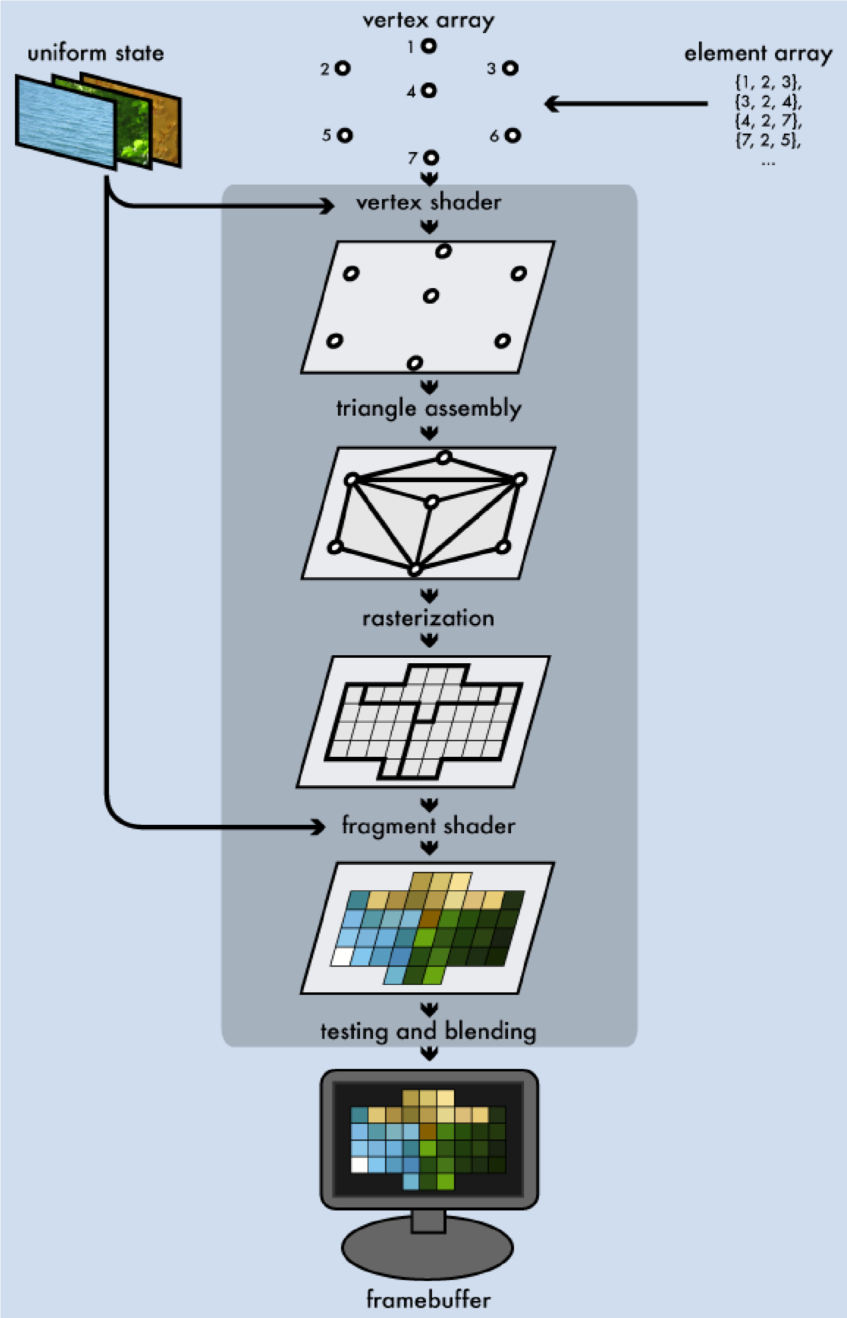
A. Independent triangles
B. A **triangle strip**, reusing the last two vertices of each triangle as the first two vertices of the next
C. Make a **triangle fan**, connecting the first element to every subsequent pair of elements



# Rasterization

## Stage 3

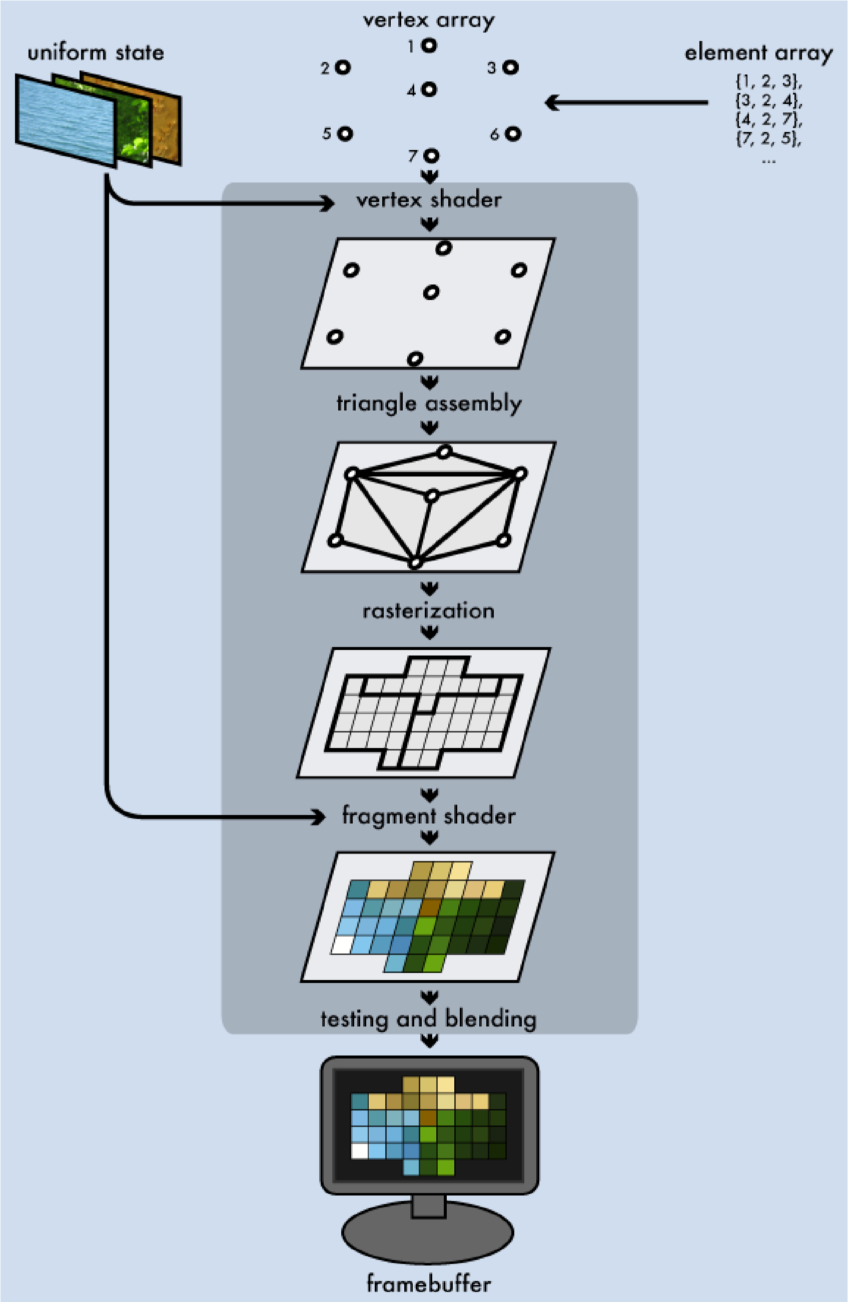
- The **rasterizer** takes each triangle, clips it and discards parts that are outside of the screen
- Breaks the remaining visible parts into pixel-sized **fragments**.



# The Fragment Shader

## Stage 4

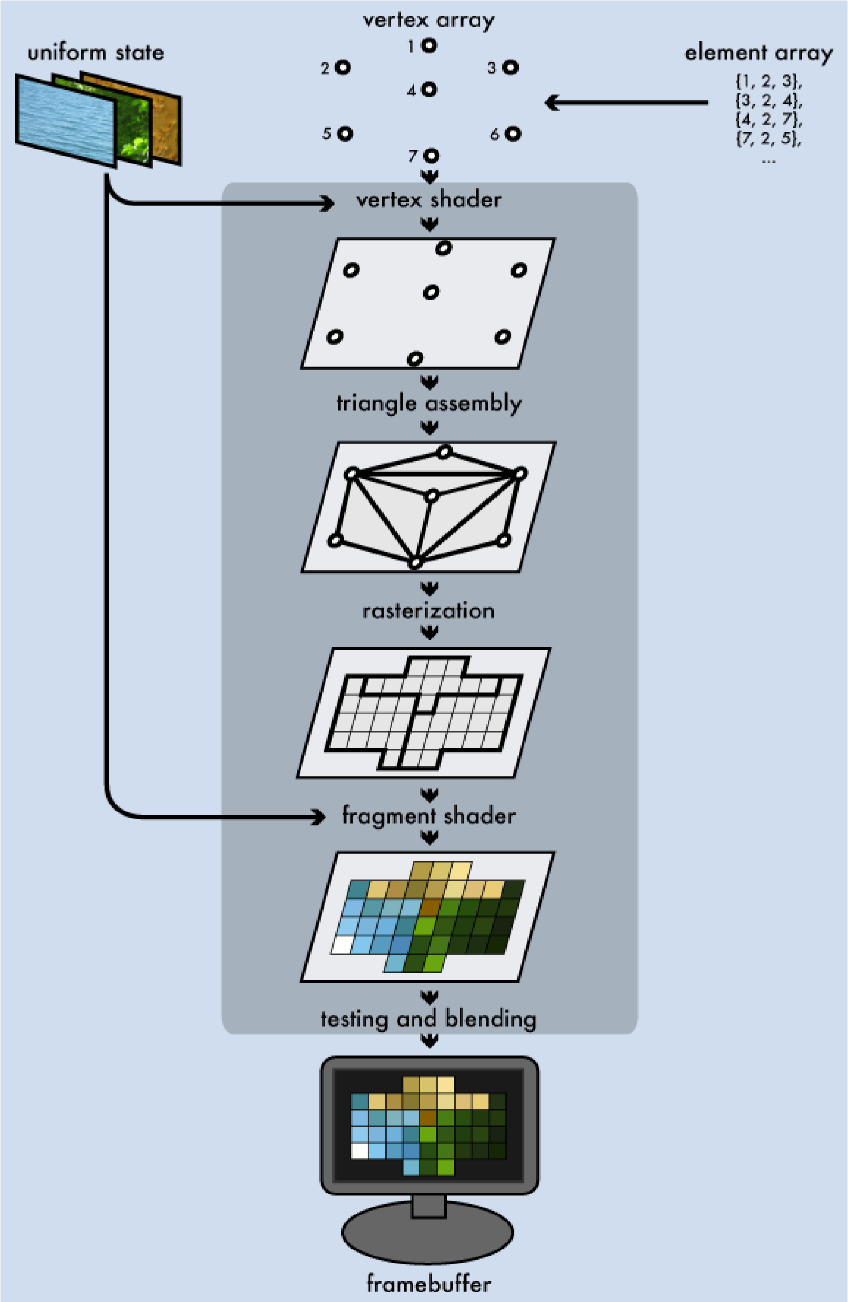
- Receives the varying values output by the vertex shader and interpolated by the rasterizer as inputs.
- Outputs color and depth values that then get drawn into the framebuffer.
- Texture mapping
- Lighting
- Performs the most sophisticated special effects; however, it is also the most performance-sensitive part of the graphics pipeline.



# The Fragment Shader

Stage 5

- A **framebuffer** is the final destination for the rendering job's output.
- Most modern OpenGL implementations let you make **framebuffer objects** that you draw into them



# Some terminology

**Rendering:** the process by which a computer creates an image from models.
**Models or Objects:** are constructed from geometric primitives (points, lines, triangles) that are specified by their vertices.
**Vertex:** the position of a point in space (note that a vertex has no size), can be 2D or 3D.
**Fragment:** for now we can loosely consider the fragment as a screen pixel. Although it's more complicated than that.
**Primitive:** it is a way that OpenGL interprets vertex streams, representing them as triangles, lines, points and so on.

**Shaders:** simply put, shaders are special functions that are executed by the graphics card. Or we can say that shaders are little programs that are specifically compiled for your GPU. The OpenGL provided by your GPU manufacturer includes the compiler tools that will take the shader's source code and create the code that your GPU needs to execute. In OpenGL there are four shader stages (we will explain them later). The most common are, **vertex shaders**, which process the vertex data (ie: vertex positions, projection), and **fragment shaders**, which operates on fragments generated by the rasterizer (ie: pixel colors) (again, we will explain shaders in depth later). The point is BOTH vertex and fragment shaders are required in every OpenGL program.
**Buffer object:** is an object that holds a chunk (array) of unformatted memory and other attributes related to it, the OpenGL context manages the buffer object, which can store the vertex data, pixel data from images and a variety of other things. (ex: frame buffer).
**Framebuffer:** is a chunk of memory that the graphics hardware manages, which contains the pixels to be displayed on your monitor, and the graphics card feeds it to your monitor.

# Required Packages

**freeglut** by freeglut contributors, Garrett Serack — v2.8.1.15
Freeglut, the Free openGL Utility Toolkit, is meant to be a free alternative to Mark Kilgard's GLUT library

**freeglut.redist** by freeglut contributors, Garrett Serack — v2.8.1.15
Redistributable components for for package 'freeglut'

**nupengl.core** by Jonathan Dickinson, Ali Badereddin — v0.1.0.1
NupenGL allows you to access OpenGL from your application.

**nupengl.core.redist** by Jonathan Dickinson, Ali Badereddin — v0.1.0.1
Redistributable components for for package 'nupengl.core'

# OpenGL Rendering Pipeline

## 1. Preparing data:

- OpenGL requires that all data be stored in buffer objects.
- Filling these buffers with data can happen in different ways.

**Creating & Binding VertexArray**

```
GLuint VertexArrayID;
glGenVertexArrays(1, &VertexArrayID);
glBindVertexArray(VertexArrayID);
```

# OpenGL Rendering Pipeline

## 1. Preparing data:

- OpenGL requires that all data be stored in buffer objects.
- Filling these buffers with data can occur in different ways.

**Creating & Binding Buffer array**

```
////////////////////////////////////////////////
// 2) create a buffer object name(ID) holder.
GLuint myBufferID;
////////////////////////////////////////////////
// 3) reserve/generate a buffer object name(ID).
// void glGenBuffers(GLsizei n, GLuint * buffers);
// n: number of names to be generated. (you can generate more than one name)
// buffers: names generated.
glGenBuffers(1, &myBufferID);
////////////////////////////////////////////////
// 4) set myBufferID as the current GL_ARRAY_BUFFER.
//   note that since this is the first time we bind myBufferID,
//   in this step OpenGL will both allocate and bind the buffer object.
// void glBindBuffer(GLenum target,GLuint buffer);
// target: Specifies the target to which the buffer object is bound.
// buffer: Specifies the name of a buffer object.
glBindBuffer(GL_ARRAY_BUFFER, myBufferID);
```

# OpenGL Rendering Pipeline

## 2- Filling the buffer array

- Can be from an array hard coded inside your program
- Can be from an external module file

```
static const GLfloat g_vertex_buffer_data[] = {
    -1.0f, -1.0f, 0.0f,
    1.0f, -1.0f, 0.0f,
    0.0f,  1.0f, 0.0f,
};
```

*the vertex buffer data name*

```
//Fill the buffer with the vertices data
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL_STATIC_DRAW);
```

*the size in bytes of the buffer object's new data store.*

# OpenGL Rendering Pipeline

## 3. Enabling the vertex attribute array and draw your scene

```
//////////////////////////////////////////////////
//6) enable the desired attributes. (Please go to this section for more information about
vertex attributes:
//   the attributes are 0 indexed, and here we have only one attribute.
// void glEnableVertexAttribArray( GLuint index);
glEnableVertexAttribArray(0);
//////////////////////////////////////////////////
//7) specify the vertex-data format.
// void glVertexAttribPointer(
// GLuint index, (the index of the attribute you are describing)
//  GLint size,          (the number of elements in that attribute)
// GLenum type,  (the type of each element in that attribute)
// GLboolean normalized, (do you want to normalize the data?)
//  GLsizei stride,      (the offset between each instance of that attribute)
// const GLvoid * pointer (the offset of the first component of the first instance of the
attribute) );
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,0);
// void glDrawArrays(GLenum mode, GLint first, GLsizei count); // note that
glDrawArrays, uses the currently bound BO in GL_ARRAY_BUFFER.
  glDrawArrays(GL_TRIANGLES, 0, 3);
```

# Vertex Shader

- Outputs the vertex position coordinates and output any data the Fragment Shader needs.
- The simplest vertex shader can define the position of vertices as follows

```
#version 330 core

// input
in vec3 position;

void main()
{
    gl_Position = vec4(position,1);
}
```

# Fragment Shader

- Output from the Vertex Shader is passed to the fragment shader
- A simple fragment shader can be as follows

```
#version 330 core

// Ouput data
out vec3 color;

void main()
{
    // we set the color of each fragment to red.
    color = vec3(1,0,0);

}
```

# Compiling Shaders

- You need to read the shaders file, compile it

```
// Read the Vertex Shader code from the file
std::string VertexShaderCode;
std::ifstream VertexShaderStream(vertex_file_path, std::ios::in);
if(VertexShaderStream.is_open()){
    std::stringstream sstr;
    sstr << VertexShaderStream.rdbuf();
    VertexShaderCode = sstr.str();
    VertexShaderStream.close();
}else{
    printf("Impossible to open %s. Are you in the right directory ? Don't forget to read the FAQ !\n", vertex_file_path);
    getchar();
    return 0;
}
```

```
// Compile Vertex Shader
printf("Compiling shader : %s\n", vertex_file_path);
char const * VertexSourcePointer = VertexShaderCode.c_str();
glShaderSource(VertexShaderID, 1, &VertexSourcePointer , NULL);
glCompileShader(VertexShaderID);

// Check Vertex Shader
glGetShaderiv(VertexShaderID, GL_COMPILE_STATUS, &Result);
glGetShaderiv(VertexShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
if ( InfoLogLength > 0 ){
    std::vector<char> VertexShaderErrorMessage(InfoLogLength+1);
    glGetShaderInfoLog(VertexShaderID, InfoLogLength, NULL, &VertexShaderErrorMessage[0]);
    printf("%s\n", &VertexShaderErrorMessage[0]);
}
```

# Compiling Shaders

For simplicity, we create a class shader to use

```
// Create and compile our GLSL program from the shaders
GLuint programID = LoadShaders("SimpleVertexShader.vs", "SimpleFragmentShader.fs");
```

Use the shader before actually drawing primitives

One statement to both compile vertex and fragment shaders

```
// Use our shader
glUseProgram(programID);
// Draw the triangle !
glDrawArrays(GL_TRIANGLES, 0, 3); // 3 indices starting at 0 -> 1 triangle
```

# Compiling Shaders

Check the console window after compiling your Project
If there are no problems, an output like below should appear

```
Compiling shader : SimpleVertexShader.vs
Compiling shader : SimpleFragmentShader.fs
Linking program
```

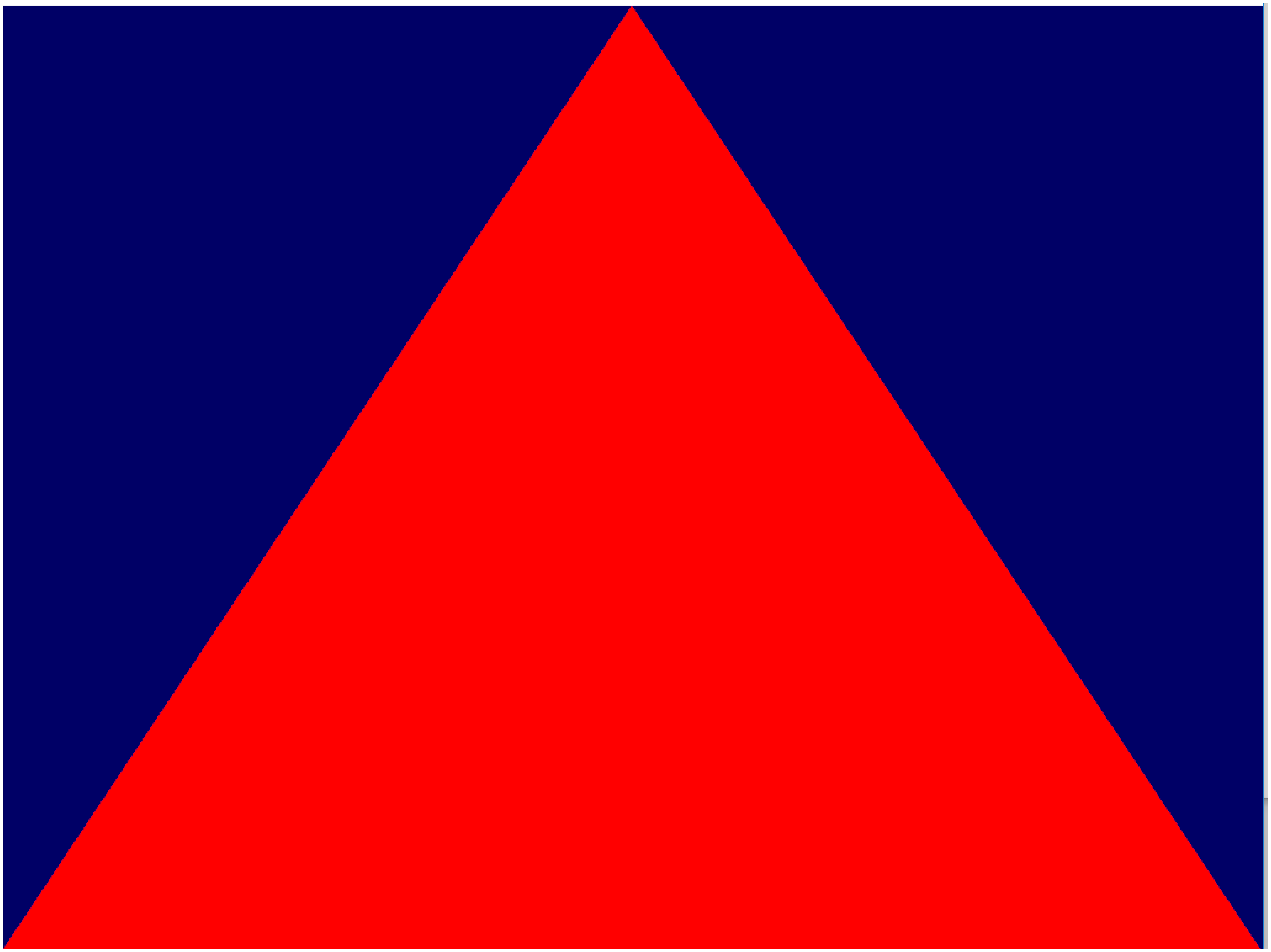
# CleanUP

Delete the both bufferarray and vertex array

```
// Cleanup VBO
glDeleteBuffers(1, &vertexbuffer);
glDeleteVertexArrays(1, &VertexArrayID);
glDeleteProgram(programID);
```

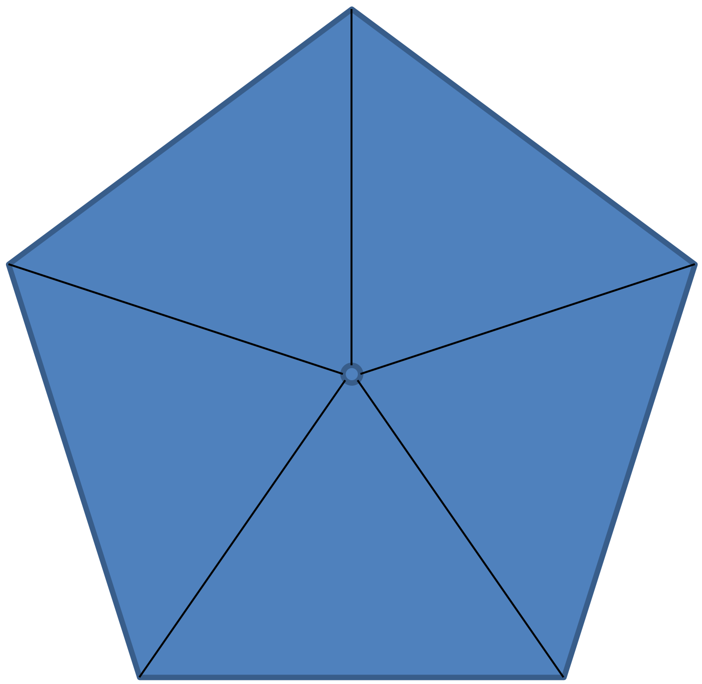
# Primitive Drawing

Run application.



# Practice

Draw a pentagon as GL_TRIANGLES using Modern OpenGL



# References


https://www.opengl.org/wiki/Getting_Started#Downloading_OpenGL
http://www.opengl-tutorial.org/miscellaneous/useful-tools-links/#Windowing___misc
http://www.glfw.org/

*Thank You*

**Questions**

*Khaled Rabieh*