

CAPITULO 2

OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

2.2 Transformaciones Geométricas en 2D

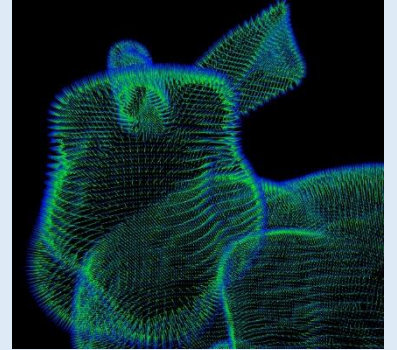
2.2.2 Shaders

Shaders – GLSL- Shader Class

- Objective → building a shader class that reads shaders from disk, compiles and links them, checks for errors and is easy to use
- We will create the shader class entirely in a **header file**, mainly for learning purposes and portability. Let's start by adding the required includes and by defining the class structure:

We used several preprocessor directives at the top of the header file. Using these little lines of code informs your compiler to only include and compile this header file if it hasn't been included yet, even if multiple files include the shader header. This prevents linking conflicts.

```
#ifndef SHADER_H
#define SHADER_H
// include glad to get all the required OpenGL headers
#include <glad/glad.h>
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
```



```
class Shader
{
public:
    unsigned int ID; // the program ID
    // constructor reads and builds the shader
    Shader(const char* vertexPath, const char* fragmentPath);

    void use(); // use/activate the shader
    // utility uniform functions
    void setBool(const std::string &name, bool value) const;
    void setInt(const std::string &name, int value) const;
    void setFloat(const std::string &name, float value) const;
};

#endif
```

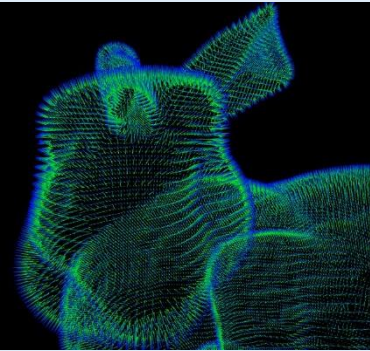
Shaders – GLSL- Shader Class

- The shader class holds the **ID** of the shader program.
- Its constructor requires the **file paths** of the source code of the vertex and fragment shader respectively that we can store on disk as simple text files.
- To add a little extra we also add several utility functions to ease our lives a little: **use** activates the shader program, and all **set...** functions query a uniform location and set its value.

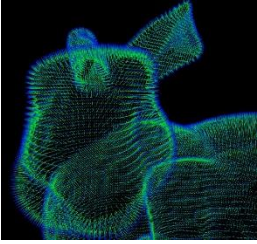
```
#ifndef SHADER_H
#define SHADER_H
// include glad to get all the required OpenGL headers
#include <glad/glad.h>
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

class Shader
{
public:
    unsigned int ID; // the program ID
    // constructor reads and builds the shader
    Shader(const char* vertexPath, const char* fragmentPath);

    void use(); // use/activate the shader
    // utility uniform functions
    void setBool(const std::string &name, bool value) const;
    void setInt(const std::string &name, int value) const;
    void setFloat(const std::string &name, float value) const;
};
#endif
```



Shaders – GLSL- Shader Class

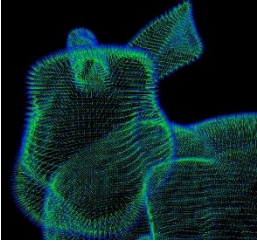


We're using C++ filestreams to read the content from the file into several string objects:

```
Shader(const char* vertexPath, const char* fragmentPath)
{
    // 1. retrieve the vertex/fragment source code from filePath
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    // ensure ifstream objects can throw exceptions:
    vShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    fShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    try
    {
        // open files
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
```

```
        // read file's buffer contents into streams
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        // close file handlers
        vShaderFile.close();
        fShaderFile.close();
        // convert stream into string
        vertexCode  = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch(std::ifstream::failure e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
    }
    const char* vShaderCode = vertexCode.c_str();
    const char* fShaderCode = fragmentCode.c_str();
    [...]
```

Shaders – GLSL- Shader Class



Next we need to **compile and link the shaders**. Note that we're also reviewing if compilation/linking failed and if so, print the compile-time errors. This is extremely useful when debugging (you are going to need those error logs eventually):

```
// 2. compile shaders
unsigned int vertex, fragment;
int success;
char infoLog[512];
// vertex Shader
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
// print compile errors if any
glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
if(!success)
{
    glGetShaderInfoLog(vertex, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n"
<< infoLog << std::endl;
};
// similiar for Fragment Shader
[...]
```

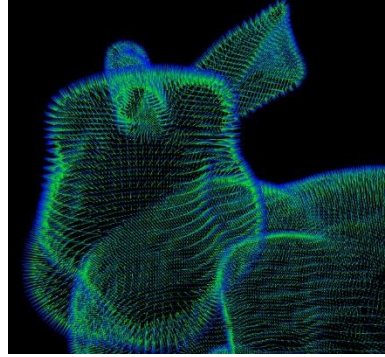
```
// shader Program
ID = glCreateProgram();
glAttachShader(ID, vertex);
glAttachShader(ID, fragment);
glLinkProgram(ID);
// print linking errors if any
glGetProgramiv(ID, GL_LINK_STATUS, &success);
if(!success)
{
    glGetProgramInfoLog(ID, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" <<
infoLog << std::endl;
}

// delete the shaders as they're linked into our program now and no
longer necessary
glDeleteShader(vertex);
glDeleteShader(fragment);
```

Shaders – GLSL- Shader Class

The use function is straightforward:

```
void use()
{
    glUseProgram(ID);
}
```



Similarly for any of the uniform setter functions:

```
void setBool(const std::string &name, bool value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
}
void setInt(const std::string &name, int value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
}
void setFloat(const std::string &name, float value) const
{
    glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
}
```

Using the shader class is fairly easy; we **create a shader object** once and from that point on simply start using it:

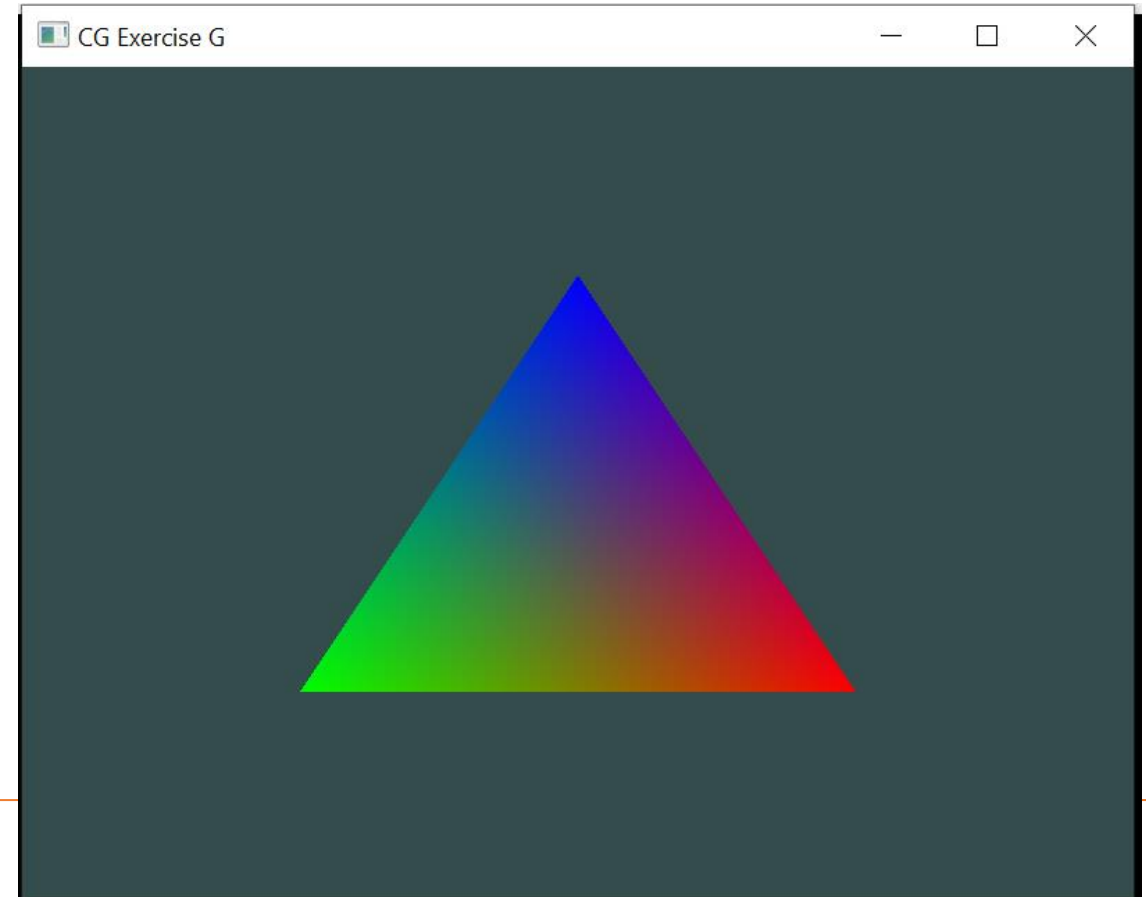
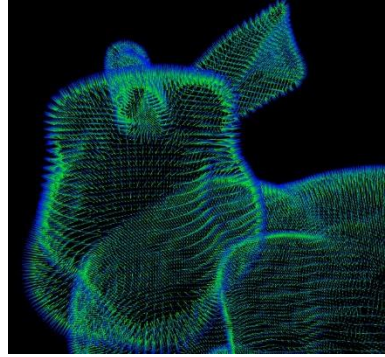
```
Shader ourShader("path/to/shaders/shader.vs",
"path/to/shaders/shader.fs");
...
while(...)
{
    ourShader.use();
    ourShader.setFloat("someUniform", 1.0f);
    DrawStuff();
}
```


Shaders – GLSL- Shader Class

Exercise 7:

Task 1:

- Create the `shader_s.h` (save in `OpenGL\OpenGL_Stuff\include\learnopengl`) and include the shader class code.
- Store the vertex and fragment shader source code in two files called `shader.vs` and `shader.fs`. The extensions `.vs` and `.fs` are quite intuitive.
- Test the Triangle Example using the Shader Class

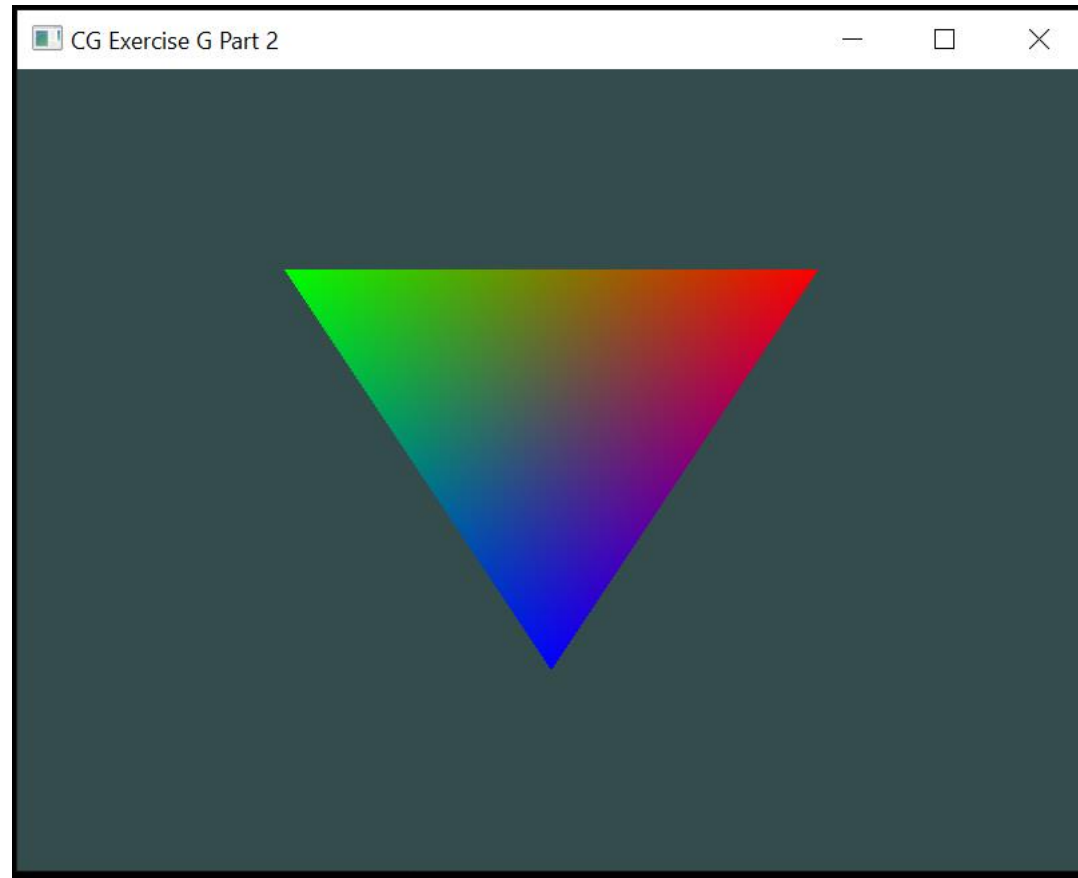
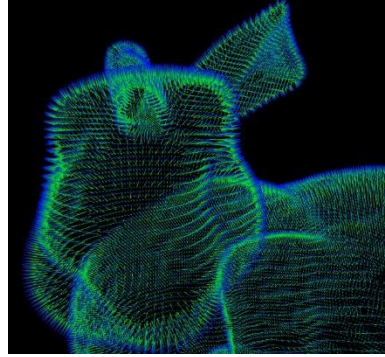


Shaders – GLSL- Shader Class

Exercise 7:

Task 2:

Adjust the vertex shader so that the triangle is upside down:



Shaders – GLSL- Shader Class

Exercise 7:

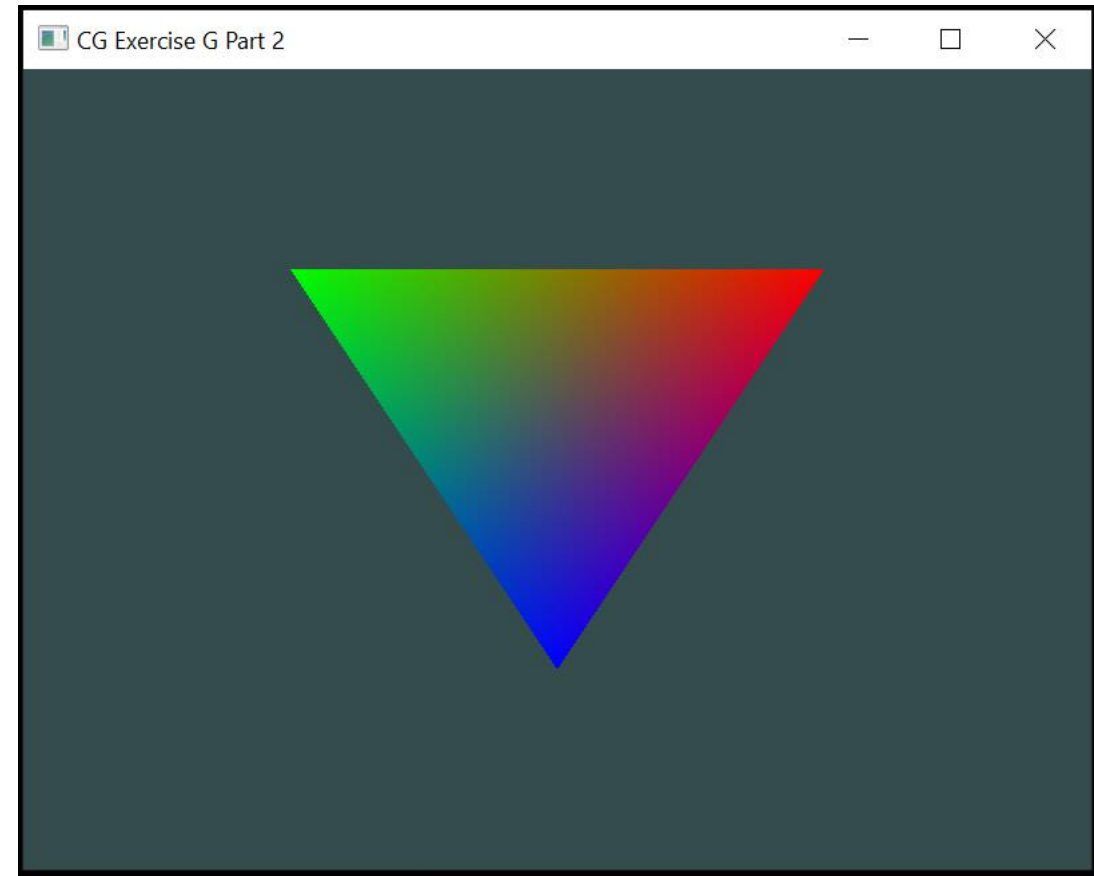
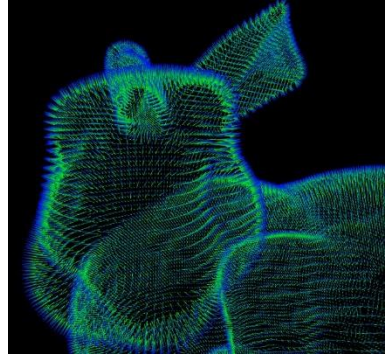
Task 2:

Adjust the vertex shader so that the triangle is upside down:

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

out vec3 ourColor;

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

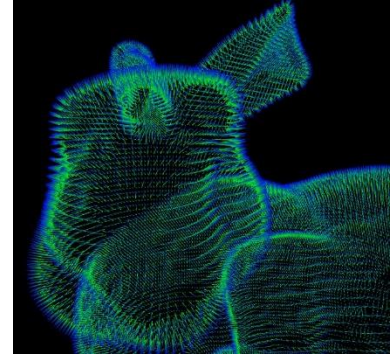
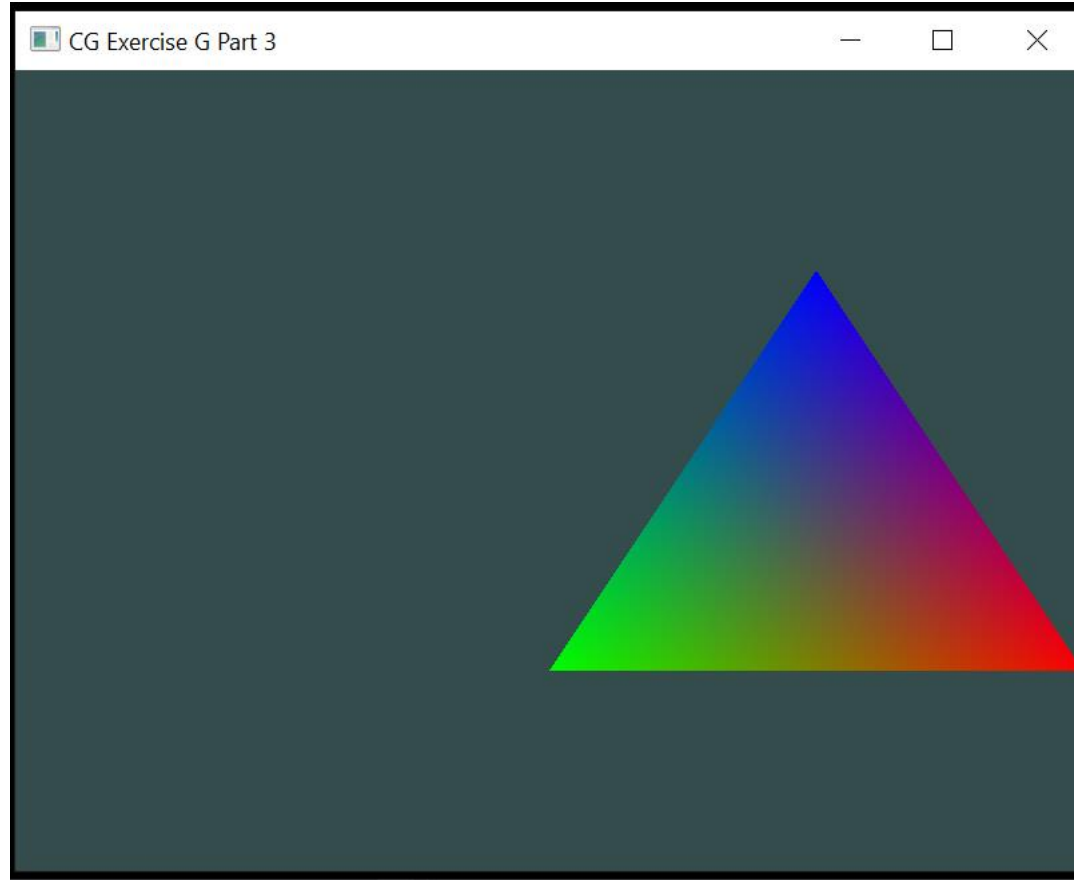


Shaders – GLSL- Shader Class

Exercise 7:

Task 3:

Specify a horizontal offset via a uniform and move the triangle to the right side of the screen in the vertex shader using this offset value: **xOffset = 0.5 f**



Shaders – GLSL- Shader Class

Exercise 7:

Task 3:

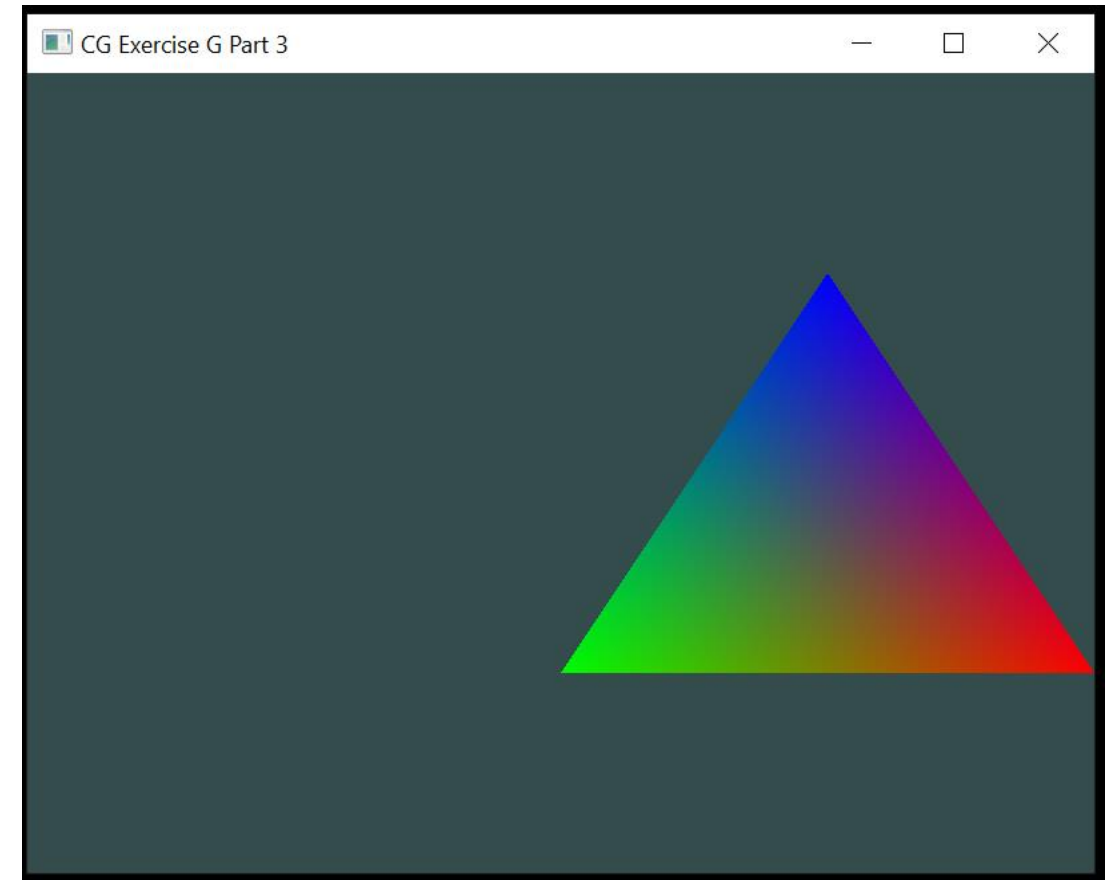
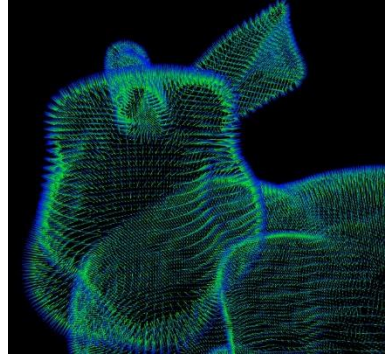
Specify a horizontal offset via a uniform and move the triangle to the right side of the screen in the vertex shader using this offset value:

In **C2_Exercise_7_SC_Task3.cpp**

```
float offset = 0.5f;  
[...]  
ourShader.use();  
ourShader.setFloat("xOffset", offset);
```

In **shader_exercise7t3.vs**

```
#version 330 core  
layout (location = 0) in vec3 aPos;  
layout (location = 1) in vec3 aColor;  
  
out vec3 ourColor;  
uniform float xOffset;  
void main()  
{  
    gl_Position = vec4(aPos.x + xOffset, aPos.y, aPos.z, 1.0);  
    ourColor = aColor;  
}
```

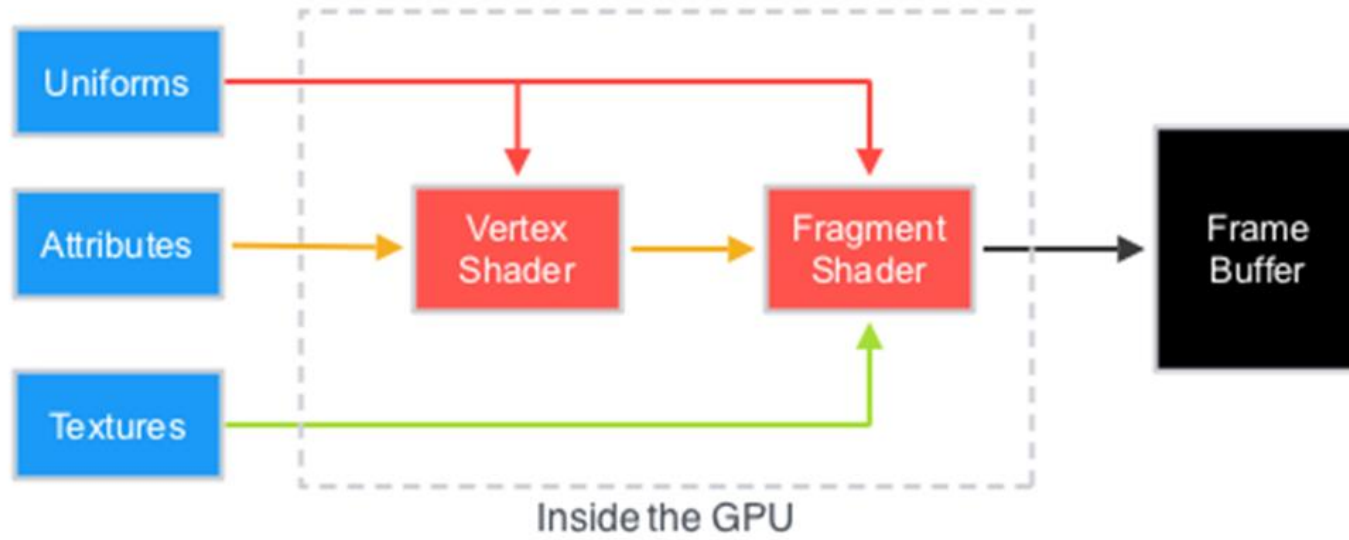
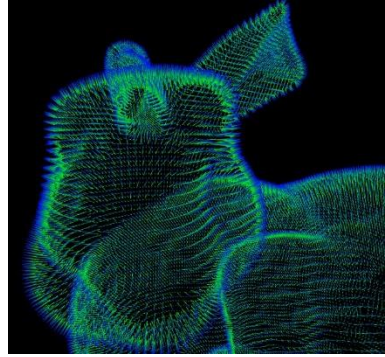


Shaders – GLSL- Shader Class

Exercise 7:

Task 4:

Output the **vertex position** to the **fragment shader** using the **out** keyword and set the fragment's color equal to this vertex position (see how even the vertex position values are interpolated across the triangle).

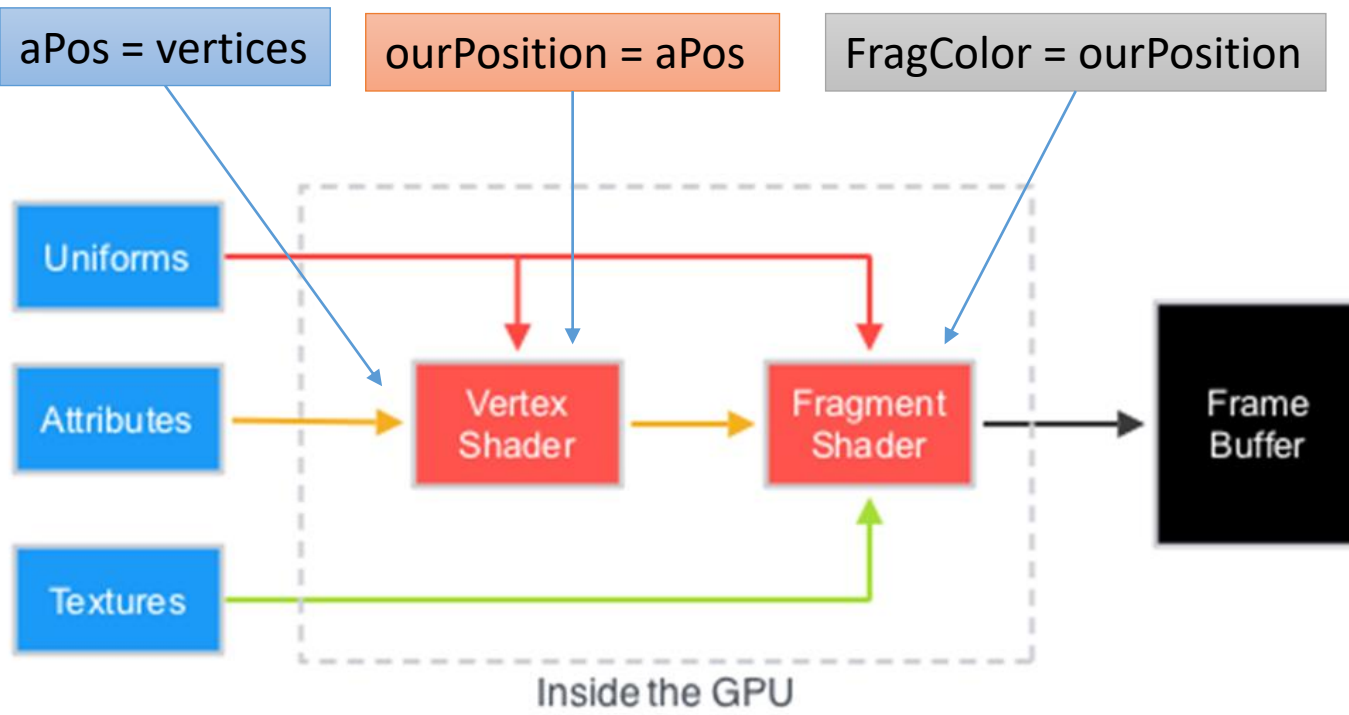
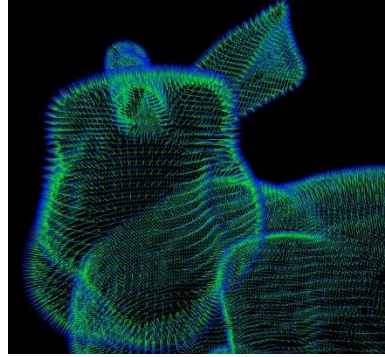


Shaders – GLSL- Shader Class

Exercise 7:

Task 4:

Output the **vertex position** to the **fragment shader** using the **out** keyword and set the fragment's color equal to this vertex position (see how even the vertex position values are interpolated across the triangle).

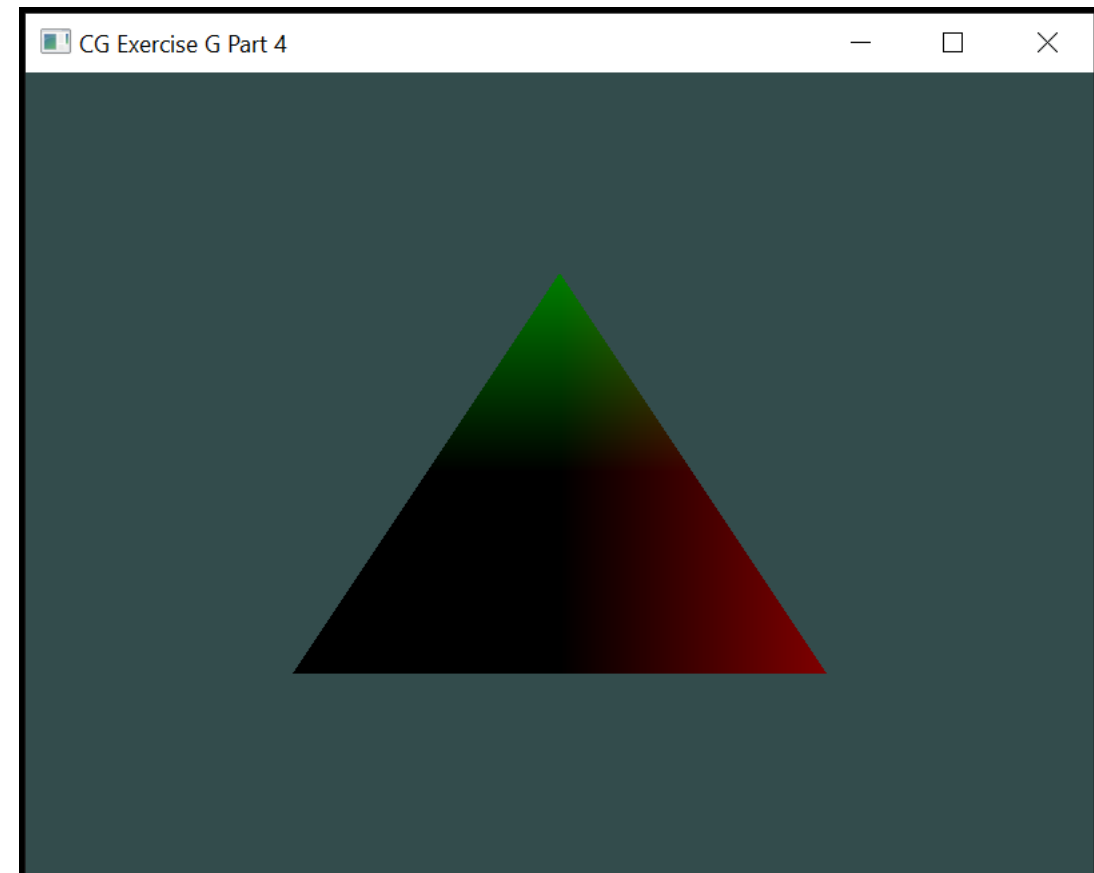
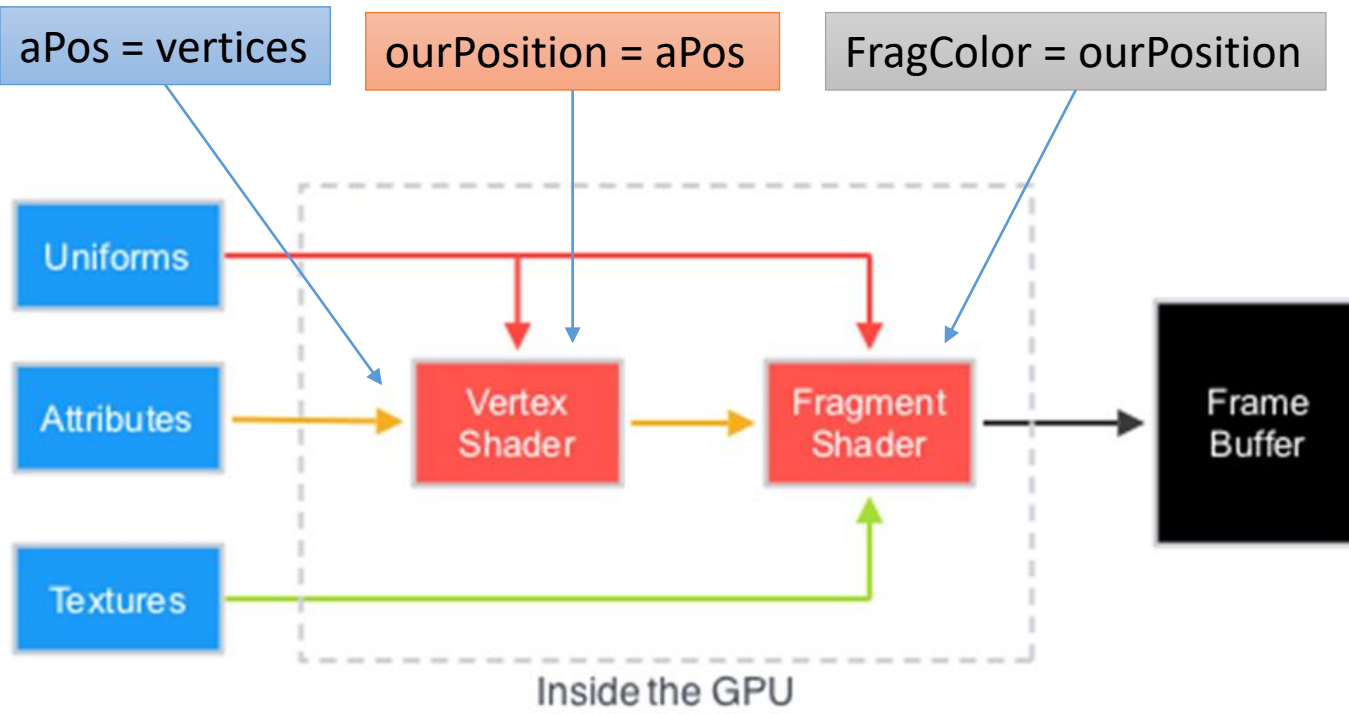
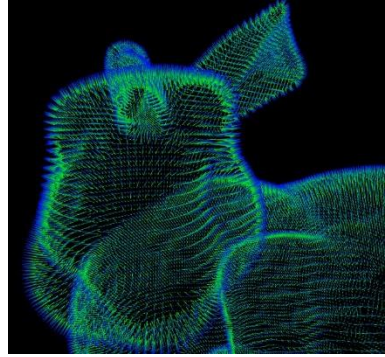


Shaders – GLSL- Shader Class

Exercise 7:

Task 4:

Output the **vertex position** to the **fragment shader** using the **out** keyword and set the fragment's color equal to this vertex position (see how even the vertex position values are interpolated across the triangle).



Shaders – GLSL- Shader Class

Exercise 7:

Task 4: Output the **vertex position** to the **fragment shader** using the **out keyword** and set the fragment's color equal to this vertex position (see how even the vertex position values are interpolated across the triangle).

aPos = vertices

ourPosition = aPos

FragColor = ourPosition

shader_exercise7t4.vs

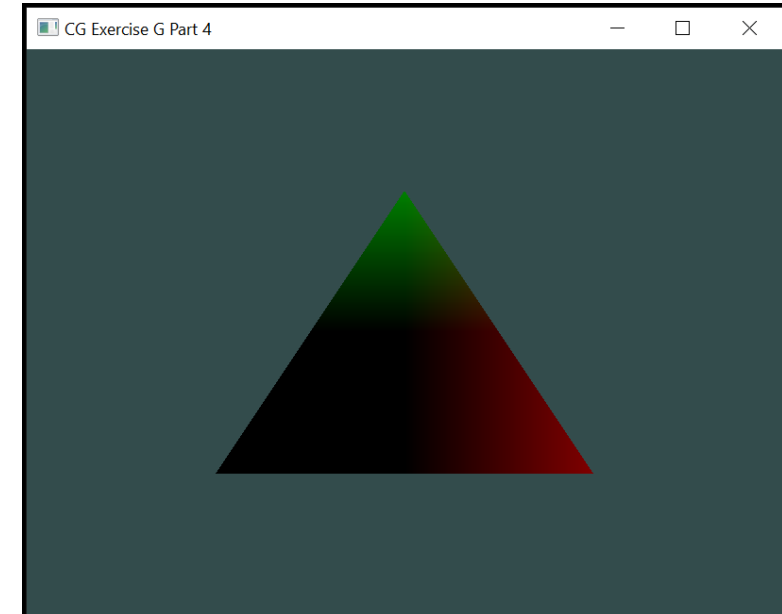
```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

// out vec3 ourColor;
out vec3 ourPosition;
void main()
{
    gl_Position = vec4(aPos, 1.0);
    // ourColor = aColor;
    ourPosition = aPos;
}
```

shader_exercise7t4.fs

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

void main()
{
    FragColor = vec4(ourPosition, 1.0);
    // note how the position value is linearly
    interpolated to get all the different colors
}
```

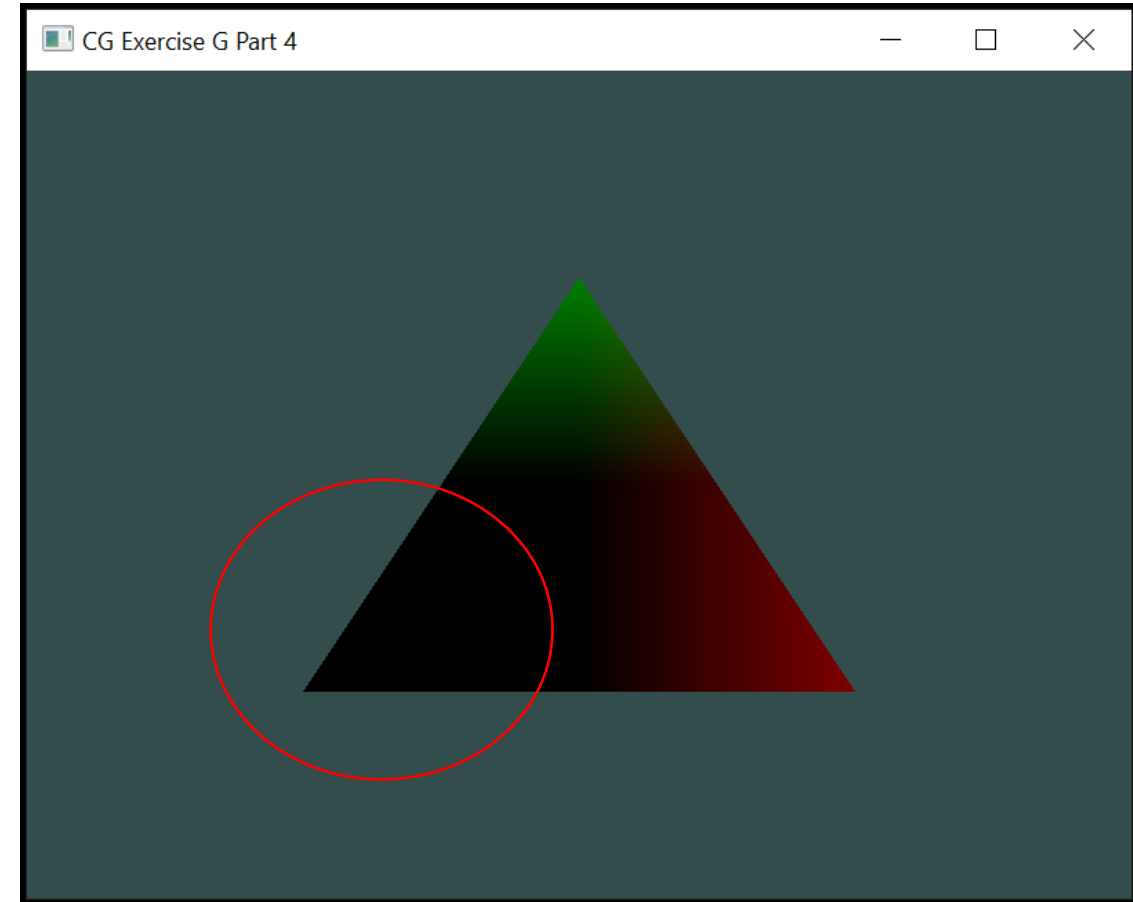


Shaders – GLSL- Shader Class

Exercise 7:

Task 4: Output the **vertex position** to the **fragment shader** using the **out keyword** and set the fragment's color equal to this vertex position (see how even the vertex position values are interpolated across the triangle).

why is the bottom-left side of our triangle black?



Shaders – GLSL- Shader Class

Exercise 7:

Task 4: Output the **vertex position** to the **fragment shader** using the **out keyword** and set the fragment's color equal to this vertex position (see how even the vertex position values are interpolated across the triangle).

why is the bottom-left side of our triangle black?

- What is the coordinate of the bottom-left point of our triangle? This is $(-0.5f, -0.5f, 0.0f)$.
- Since the xy values are **negative** they are clamped to a **value of 0.0f**.
- This happens all the way to the center sides of the triangle since from that point on the values will be interpolated positively again.
- Values of **0.0f** are of course **black** and that explains the black side of the triangle.

