Computação Gráfica: Trabalho Complementario N 1

A simple OpenGL program to render a triangle using shaders

Objetivo: O objetivo do presente trabalho é garantir que a configuração do entorno de desenvolvimento sejá o correto e que esta pronto para o desenvolvimento de programas com OpenGL and GLSL.

A ideia principal é criar um programa que permita criar uma janela do sistema operacional onde um triangulo seja renderizado. Os vertices de aquele triangulo precisam ser das cores vermelho, azul e verde respectivamente. Para conseguir alcançar esse objetivo os seguintes passos foram feitos:

- **Instalação de pacotes:** Alguns pacotes precisam ser instalados no sistema operacional pra fazer funcionar o API (Application Programming Interface) do OpenGL.
- **Criação do programa en C++:** É preciso criar um programa em C++ que usa as funções e métodos fornecidos pelo OpenGL e o GLSL para conseguir renderizar o triangulo de acordo com o objetivo do trabalho.
- **Geração do Makefile:** Criação do script Makefile para simplificar a compilação e execução do código compilado.

Nos itens a seguir, uma explicação com maior nível de detalhe sobre os passos mencionados acima sera fornecida:

Instalação de pacotes

No caso do presente trabalho, ele foi implementado no sistema operacional ubuntu 18.04. com a seguinte placa de vídeo: **GPU: NVIDIA Corporation, GeForce GT 630M/PCIe/SSE2**. A instalação do driver da mencionada placa de vídeo não vai ser abordado no presente trabalho, mas sim a instalação de pacotes adicionais no sistema operacional.

O seguinte comando deve ser executado no terminal do sistema operacional:

```
sudo apt-get install build-essential make libx11-dev libxrandr-dev

libxinerama-dev libxcursor-dev libxcb1-dev libxext-dev

libxrender-dev libxfixes-dev libxau-dev libxdmcp-dev mesa-common-dev

libxxf86vm-dev cmake
```

Uma vez instalados os pacotes adicionais, já é possível começar a escrever o código. Além disso alguns pacotes e arquivos de bibliotecas como o GLAD[1] e o GLFW [2] necessárias no devenvolvimento do trabalho, são fornecidas junto com o código fonte neste trabalho. A estrutura e jerarquia de arquivos é a seguinte:

Figure 1: Estrutura de Arquivos do código implementado.

Criação do Programa em C++

O código implementado no presente trabalho pode ser dividido em três partes principais, a seguir:

1. **Inicialização do OpenGL:** Nesta parte do código o GLFW é inicializado, estabelecendo adicionalmente a versão de OpenGL a ser usada. O GLAD também é carregado nesta parte e uma janela do sistema operacional é criada, se fazem as validações necessárias de que tudo foi feito corretamente e se imprime informação importante no terminal sobre a placa de vídeo e as versões de OpenGL e GLSL (codigo para os shaders) sendo usadas. O código que implementa o antes mencionado se encontra desde a linha 30 até a linha 66 do arquivo triangles.cpp e é mostrado na seguinte porção de codigo:

[1] https://github.com/Dav1dde/glad
[2] https://www.glfw.org/

Computação Gráfica: Trabalho Complementario N 1

Code Snippet 1: Creating composite image from the provided files

```
//Initiating glfw
glfwInit();

//Defining glfw properties for opengl Version
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

//Specify the size of the window to be created
int window_width = 500;
int window_height = 500;

//Creating window with width and height specified before.
GLFWwindow* window = glfwCreateWindow(window_width,
window_height, "CMP143 - 00312086 - Felix Eduardo Huaroto Pachas", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }

//Indicating that current context will be window recently created
    glfwMakeContextCurrent(window);

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

// Printing to terminal opengl and glsl version
const GLubyte *vendor      = glGetString(GL_VENDOR);
const GLubyte *renderer    = glGetString(GL_RENDERER);
const GLubyte *glversion   = glGetString(GL_VERSION);
const GLubyte *glslversion = glGetString(GL_SHADING_LANGUAGE_VERSION);
printf("GPU: %s, %s, OpenGL %s, GLSL %s\n",
        vendor, renderer, glversion, glslversion);
```

2. **Implementação dos shaders:** Nesta parte do código, dois shader foram criados: o **vertex shader** e o **fragment shader** de acordo com as suas definições as quais foram definidas no mismo arquivo triangles.cpp em formato de string. Como pode ser visto no code snippet 2 os vertices recebem basicamente dois parâmetros ou informações: a posição dos vertices e as cores de cada um deles. No code snippet 3 pode ser visto que a cor que vai ser renderizada é aquela que é fornecida na declaração do shader. A seguir as duas definições mencionadas anteriormente são mostradas:

Code Snippet 2: Definição do vertex shader

```
const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec4 NDC_coefficients;\n"
    "layout (location = 1) in vec4 color_coefficients;\n"
    "out vec4 rasterizer_color;\n"
    "void main()\n"
    "{\n"
    "    gl_Position = NDC_coefficients;\n"
    "    rasterizer_color = color_coefficients;\n"
    "}\0";
```

Code Snippet 3: Definição do fragment shader

```
const char *fragmentShaderSource = "#version 330 core\n"
    "in vec4 rasterizer_color;\n"
    "out vec4 FragColor;\n"
    "void main()\n"
    "{\n"
    "    FragColor = rasterizer_color;\n"
    "}\n\0";
```

O resto de código para essa parte consiste no carregamento dos dois shader e a sua correspondente validação de erros de carregamento. Adicionalmente, é preciso criar um programa de GPU e vincular a ele os dois shaders carregados, da mesma forma fazer a validação de erros desse procedimento e remover os shaders, já que não são ainda necessários. A seguir no code snippet4 só a criação do programa de GPU é mostrado:

Code Snippet 4: Criação de um Programa GPU

```
// Creating GPU program
int shaderProgram = glCreateProgram();
//Attaching/Linking shaders
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
//Linking program.
glLinkProgram(shaderProgram);

// check for linking errors
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}

//Removing shaders after used (not needed anymore)
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

Após a criação do programa do GPU, o seguinte a fazer é criar os VBO (Virtual Buffer Object) e os VAO (Vertex Array Object). O VBO contém a informação sobre os vertices que desejamos passar para os Shaders enquanto que o VAO describes como essa informação é armazenada no VBO. O seguinte code snippet mostra como é que o VAO e inicalizado ao igual que o VBO, e que temos dos VBOs: um deles com a informação sobre a posição de fato dos vertices enquanto que o outro contém informação sobre as cores de cada um dos vertices. É precisso indicar também os valores dos arrays tanto para posição como para as cores estão expressas no NDC (Normalize Device Coordinates). No caso do array con informação das posições nós estamos usando os primeiros três valores de cada linha para representar as três dimensões enquanto que o ultimo valor é 1.0 representando a coordenada homogênea. No caso do array de cores, os tres primeiros valores de cada linha representan o RGB enquanto que o ultimo valor representa a transparência. No seguinte code snippet (5) a definição do VAO e dos VBOs assim como a definições das matrizes antes mencionadas é mostrado:

Code Snippet 5: Definição de VAO e VBOs

```
//Setting vertex ndc_coefficients (Normal device coordinates)
GLfloat NDC_coefficients[] = {
//    X      Y      Z      W
    -0.5f, -0.5f, 0.0f, 1.0f,
     0.5f, -0.5f, 0.0f, 1.0f,
     0.0f,  0.5f, 0.0f, 1.0f,
     0.5f,  0.5f, 0.0f, 1.0f
};

//Declaring Virtual Buffer Objects (VBO) and Vertex Array Object(VAO)
unsigned int VBO, VAO;
glGenBuffers(1, &VBO);
glGenVertexArrays(1, &VAO);

// binding the Vertex Array Object.
glBindVertexArray(VAO);
// Binding buffer for vertex VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO);

//Indicating data size and passing vertex information to shader
glBufferData(GL_ARRAY_BUFFER, sizeof(NDC_coefficients), NDC_coefficients,
GL_STATIC_DRAW);
//Setting 0 in position (according to the specification on the shader)
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
//Unbinding the VBO buffer
glBindBuffer(GL_ARRAY_BUFFER, 0);

//Setting variable vertices colors (Red, Green and Blue) A for opacity.
GLfloat color_coefficients[] = {
//  R     G     B     A
    1.0f, 0.0f, 0.0f, 1.0f,
```

```
    0.0f, 1.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f, 1.0f,
    0.0f, 1.0f, 1.0f, 1.0f
};

//Registering a new VBO for color coefficients
unsigned int VBO_color;
glGenBuffers(1, &VBO_color);

//Binding the new VBO to the VAO
glBindBuffer(GL_ARRAY_BUFFER, VBO_color);

//Indicating data size and passing vertices color information to shader
glBufferData(GL_ARRAY_BUFFER, sizeof(color_coefficients), color_coefficients,
GL_STATIC_DRAW);
//Setting 1 in position (according to the specification on the shader)
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);

//Unbinding VBO for color coefficiente
glBindBuffer(GL_ARRAY_BUFFER, 0);

//Unbinding VAO
glBindVertexArray(0);
```

3. **Renderização de objetos:** Finalmente nós precisamos renderizar os elementos que foram criados com anterioridade dentrom de um loop infinito. O loop faz basicamente 4 coisas em cada iteração:

   - Estabelece a cor do fundo da janela para branco.
   - Carrega ou usa o programa de GPU que nós tinhamos criado com anterioridade.
   - Anexa o VAO criado também com anterioridade.
   - Desenha o triangulo com base no VAO indicado.

   O seguinte code snippet 6 mostra o código do loop que faz o que já foi mencionado nos items acima:

Code Snippet 6: Loop principal de renderização

```
//Rendering into the created window
while(!glfwWindowShouldClose(window))
{
        //Processing input to close window
        processInput(window);

        // Clearing the window background
        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
```

```
        // Using GPU Program created
        glUseProgram(shaderProgram);

        //Binding VAO, maybe no need to binding it
        //every time because have only one VAO
        glBindVertexArray(VAO);

        //Drawing actual triangles
        glDrawArrays(GL_TRIANGLES, 0, 4);

            //Swapping buffers
        glfwSwapBuffers(window);

        //Keyboard and mouse events
        glfwPollEvents();
}
```

Execução

Para fazer a execução mais comoda e intuitiva, um arquivo Makefile foi desenvolvido para compilar o codigo e rodar ele com facilidade. Para compilar o código, na pasta principal (aquela que têm as pastas de src, bin e include) é precisso executar o comando **make**, se tudo dar certo ele não mostra messagem de erro nenhum. Para executar o código compilado, só precisa executar o comando **make run** e de novo, se tudo dar certo, vai poder observar a seguinte imagem:

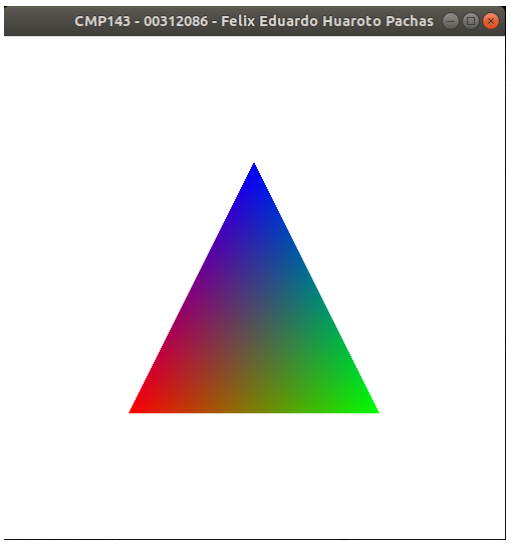

Figure 2: Execução do codigo compilado.

Conclusões

1. O objetivo esperado do trabalho foi alcançado conseguindo renderizar um triangulo com as cores vermelho, amarelo e azul nos vertices usando shaders.

2. Essa primeira experiência com o OpenGL forneceu para mim uma ideia sobre a composição geral de um programa em OpenGL assim como suas diferentes componentes.

3. Desenvolver esse programa forneceu também conhecimento sobre alguns tipos de objetos que o API de OpenGL fornece para realizar o que é desejado.