Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Atividade 5: Fundamentos de Programação Gráfica

BCC327 - Computação Gráfica

Matheus Peixoto Ribeiro Vieira - 22.1.4104
Professor: Rafael Alves Bonfim de Queiroz

Ouro Preto
20 de dezembro de 2024

Sumário

Lista de Figuras

Lista de Códigos Fonte

1 Introdução

Para esta atividade, dois exercícios foram propostos, sendo o primeiro para gerar um triângulo com vértices de diferentes cores e o seu corpo com cores interpoladas e, no segundo exercício, era necessário criar um quadrado que se move de acordo com o pressionamento das teclas do teclado.

1.1 Especificações da máquina

A máquina onde o desenvolvimento foi realizado possui a seguinte configuração:

- Processador: Intel Core i5-9300H
- Memória RAM: 16Gb.
- Sistema Operacional: WSL 2.0 com Ubuntu 22.04.5 LTS

2 Exercício 1

Neste exercício, era solicitado a criação de um triângulo com cada vértice de uma cor e uma interpolação das cores no corpo corpo do triângulo, gerando um gradiente suave da mudanças de cores.

Para realizar a atividade, foi utilizado o OpenGl com a biblioteca GLFW e Glad, as mesmas utilizadas para a atividade 1 da disciplina.

Dessa forma, foi criado dois vetores para armazenar as informações dos vértices. Um dos vetores armazena as posições e, o outro, as cores, como pode ser visto no código 1

```
    // Posicoes dos vertices
    float positions[] = {
         0.0f,  0.5f, 0.0f,  // Topo
        -0.5f, -0.5f, 0.0f,  // Inferior esquerdo
         0.5f, -0.5f, 0.0f   // Inferior direito
    };

    // Cores dos vertices
    float colors[] = {
        1.0f, 0.0f, 0.0f,   // Vermelho
        0.0f, 1.0f, 0.0f,   // Verde
        0.0f, 0.0f, 1.0f    // Azul
    };
```

Código 1: Posições e cores

Dessa forma, da posição 0 até a 2 em ambos correspondem ao primeiro vértice, da 3 a 5, o segundo e da 6 até a 8, o último.

Assim, como o OpenGL requisita e, como foi discutido na primeira atividade, foi gerado um VAO, para armazenar os objetos, e um VBO, sendo um para cada vetor de informações do vértice.

Tanto as posições quanto as cores são passadas para o buffer a partir do `glVertexAttribPointer` [4], que recebe como parâmetros a posição do atributo no vetor de dados, a quantidade de itens que serão utilizadas para aquele vértice (ex: posição de X, Y e Z), uma explicitação do tipo de dados, marcando sendo do tipo float,um valor booleano para normalização de dados, a quantidade de itens que, de fato, serão utilizadas do vetor, e o ponteiro para o vetor com as informações.

Dessa forma, essas informações vão para o vertex shader e para o fragment shader, que são definidos no código 2

```
// Vertex Shader source code
const char* vertexShaderSource = R"(
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
out vec3 vertexColor;
void main()
{
```

```
    gl_Position = vec4(aPos, 1.0);
    vertexColor = aColor;
}
)";

// Fragment Shader source code
const char* fragmentShaderSource = R"(
#version 330 core
in vec3 vertexColor;
out vec4 FragColor;
void main()
{
    FragColor = vec4(vertexColor, 1.0);
}
)";
```

Código 2: Vertex e Fragment shader

Com os shaders definidos, as cores e posições de cada vértice também, resta agora realizar a interpolação. Para isso o próprio OpenGL e os shaders fazem tal processo automaticamente, sendo isso de responsabilidade do OpenGL Smooth shading model [1].

Para a realização do cálculo da interpolação, o OpenGL irá utilizar coordenadas baricêntricas [5], um sistema onde um ponto no espaço é representado em função de outros pontos [6].

Dessa forma, fazendo a compilação e execução dos shaders, obteve-se o resultado final como pode ser observado na imagem 1.

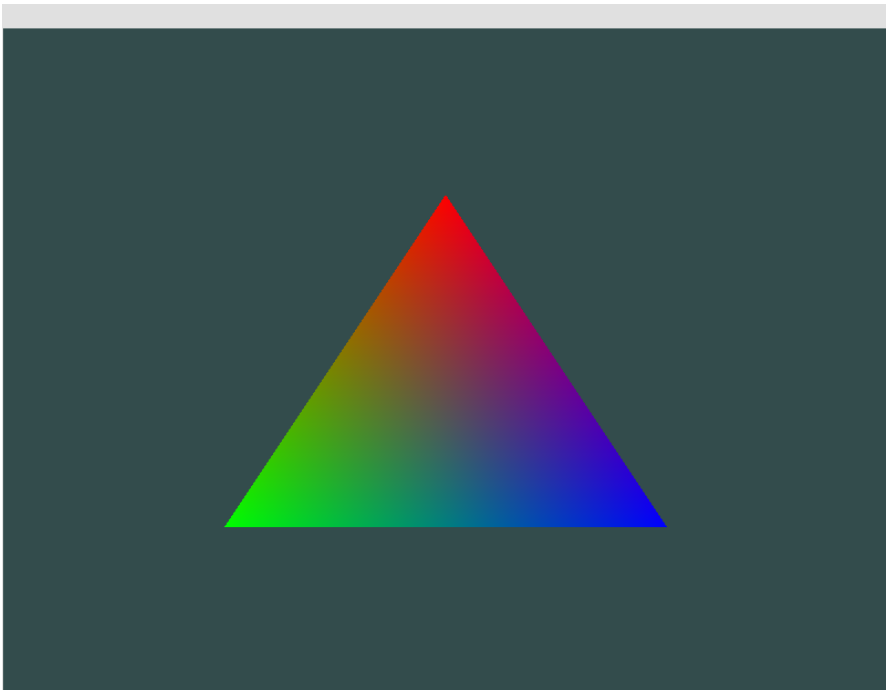

Figura 1: Triângulo com cores interpoladas

Para compilar e executar o código, é necessário utilizar os seguinte comandos:

```
g++ exercicio1.cpp glad.c -o exec -lGL -lGLU -lglfw3 -lX11 -lXxf86vm -lXrandr -lpthread -lXi
./exec
```

3 Exercício 2

para a segunda atividade, foi solicitada a implementação de uma aplicação onde seja possível movimentar um quadrado pela tela ao utilizar as setas do teclado. Dessa forma, conceitos da computação interativa foram utilizados.

No código, foi definido como uma variável global, um valor de offset para representar as posições X e Y do quadrado, sendo utilizado no shader para definir a posição do item.

Dessa forma, o OpenGL irá, no loop principal, utilizar o shader do quadrado para exibi-lo na tela a partir do `glUseProgram`. Em seguida, o `glGetUniformLocation` recupera a localização do offset em um shader determinado [2]. Com esse valor, então é possível modificar essa variável, que é do tipo 1 (float), usando os valores do offset correspondente para determinar a nova posição do item, isso tudo com a função `glUniform3fv` [3], finalizando com o bind do VAO para permitir a exibição e o desenho do quadrado.

No loop principal, é chamada uma função para processamento de inputs, e nela é verificado qual tecla foi pressionada, aumentando o valor de X e Y do offset quando as setas para cima e para a direita forem pressionadas, respectivamente, e decrementando os valores quando a seta para baixo e a para esquerda forem pressionadas. Todavia, para impedir que o quadrado saisse da tela, limites foram colocados para uma movimentação de, no máximo, 0.5, tanto positivo quanto negativo, como pode ser visto no código 3.

```
1  glm::vec3 offset = glm::vec3(0.0f, 0.0f, 0.0f);
2
3  void processInput(GLFWwindow *window){
4      if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
5          glfwSetWindowShouldClose(window, true);
6
7      if (glfwGetKey(window, GLFW_KEY_UP)    == GLFW_PRESS && offset.y <= .5)
           offset.y += 0.01f;
8      if (glfwGetKey(window, GLFW_KEY_DOWN)  == GLFW_PRESS && offset.y >= -.5)
           offset.y -= 0.01f;
9      if (glfwGetKey(window, GLFW_KEY_LEFT)  == GLFW_PRESS && offset.x >= -.5)
           offset.x -= 0.01f;
10     if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS && offset.x <= .5)
           offset.x += 0.01f;
11 }
```

Código 3: processamento de input

Por fim, a figura 2 mostra o estado inicial da aplicação, enquanto a figura 3 mostra o final da movimentação do quadrado após o seu deslocamento e atingindo o limite da tela.

Para a compilação e execução do código o seguinte comando deve ser executado:

```
g++ exercicio2.cpp glad.c -o exec -lGL -lGLU -lglfw3 -lX11 -lXxf86vm -lXrandr -lpthread -lXi
./exec
```

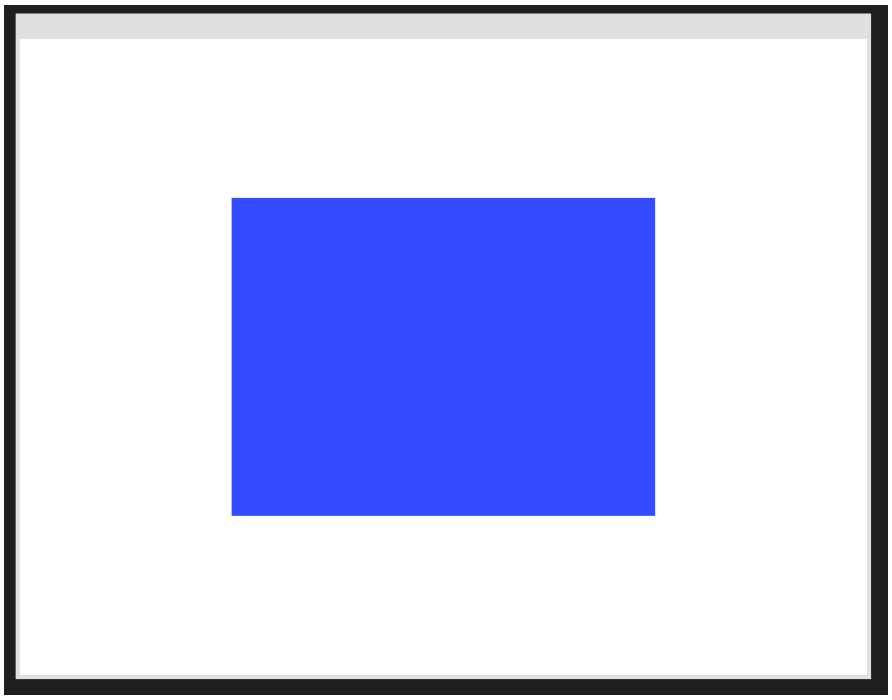

Figura 2: Posição inicial

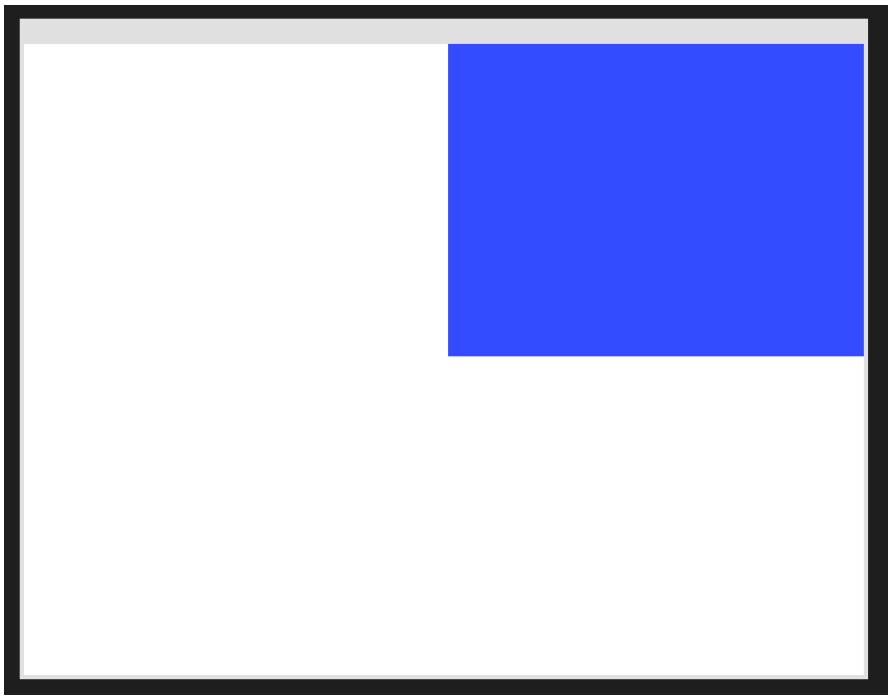

Figura 3: Posição final

Referências

[1] Ali BaderEddin. Opengl color interpolation. https://open-gl.com/2010/05/16/interpolation/. [Online; acessado em 19-Dezembro-2024].

[2] Khronos. glgetuniformlocation. https://registry.khronos.org/OpenGL-Refpages/gl4/html/glGetUniformLocation.xhtml. [Online; acessado em 19-Dezembro-2024].

[3] Khronos. gluniform3fv. https://registry.khronos.org/OpenGL-Refpages/gl4/html/glUniform.xhtml. [Online; acessado em 19-Dezembro-2024].

[4] Khronos. glvertexattribpointer. https://registry.khronos.org/OpenGL-Refpages/gl4/html/glVertexAttribPointer.xhtml. [Online; acessado em 19-Dezembro-2024].

[5] Reddit. Fragment interpolation. https://www.reddit.com/r/opengl/comments/188g9tu/fragment_interpolation/. [Online; acessado em 19-Dezembro-2024].

[6] Wikipedia. Coordenadas baricêntricas. https://pt.wikipedia.org/wiki/Coordenadas_baric%C3%AAntricas#:~:text=As%20coordenadas%20baric%C3%AAntricas%20definem%20uma,ponto%20seja%20igual%20a%20um. [Online; acessado em 19-Dezembro-2024].