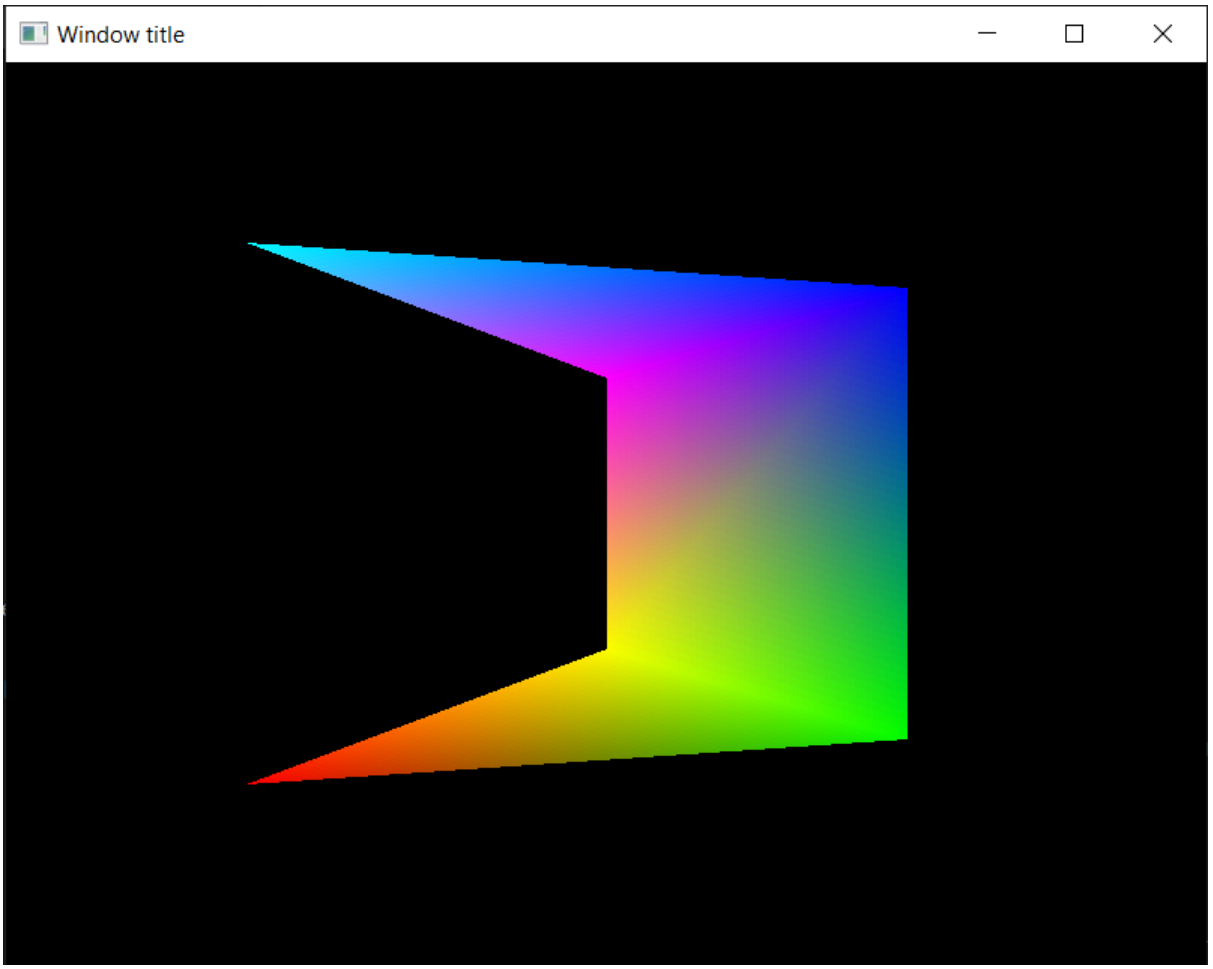
Relatório do envio do trabalho 1 INF1761

Segue em .zip uma pasta onde estão os códigos usados para gerar a janela estática da imagem a seguir.



O código em si eu decidi tentar modularizar o quanto pude, fazendo a classe de polígono herdar de uma classe de shape mais rica do que a base provida pelo enunciado, que encapsula a grande maioria das chamadas da API do OpenGL de fato. Na main é onde está a definição do polígono estático.

Decidi também separar as funções que tratam os inputs do usuário em um arquivo dedicado, para reduzir o tamanho do arquivo main o máximo possível.

Os shaders em glsl estão escritos em arquivos dedicados também com o intuito de modularização.

Os arquivos contém os códigos:

gl_includes.h:

```
#pragma once
#include <glad/gl.h>
#include <GLFW/glfw3.h>
```

generic2Dshape.h:

```
#ifndef GENERIC2DSHAPE_H
#define GENERIC2DSHAPE_H
#pragma once

#include "gl_includes.h"

#include "shape.h"

#include <memory>
class Generic2dShape;
using Generic2dShapePtr = std::shared_ptr<Generic2dShape>;


class Generic2dShape : public Shape {
    unsigned int mode = GL_TRIANGLES;
    unsigned int nverts;
    unsigned int type = GL_UNSIGNED_INT;
    unsigned int offset = 0;
    unsigned int m_vao;
    unsigned int m_vbo;
    unsigned int m_ebo;
    int n_indices;

protected:
    // Construtor agora armazena vbo e ebo
    Generic2dShape(float* dados_vertices, unsigned int* indices, int
nverts, int n_indices) :
    nverts(nverts),
    n_indices(n_indices)
    {
        // Define os parâmetros para o Draw()
        mode = GL_TRIANGLES;
        type = GL_UNSIGNED_INT;
        // Cada vértice tem 2 floats para posição e 3 floats para cor
        int amt_of_floats_per_vertex = 2 + 3;
        int vertex_stride_in_bytes = amt_of_floats_per_vertex *
sizeof(float); // 5 floats por vértice
```

```
        // 1. Geração e bind do VAO
        glGenVertexArrays(1, &m_vao);
        glBindVertexArray(m_vao);

        // 2. Geração, bind e envio de dados para o VBO
        glGenBuffers(1, &m_vbo);
        glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
        glBufferData(GL_ARRAY_BUFFER, amt_of_floats_per_vertex * nverts
* sizeof(float), dados_vertices, GL_STATIC_DRAW);

        // 3. Configuração dos atributos dos vértices
        // Diz ao OpenGL como interpretar os dados de posição do VBO
        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,
vertex_stride_in_bytes, (void*)0);
        glEnableVertexAttribArray(0);

        // Diz ao OpenGL como interpretar os dados de cor do VBO
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
vertex_stride_in_bytes, (void*)(2 * sizeof(float)));
        glEnableVertexAttribArray(1);

        // 4. Geração, bind e envio de dados para o EBO
        glGenBuffers(1, &m_ebo);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_ebo);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, n_indices *
sizeof(unsigned int), indices, GL_STATIC_DRAW);

        // 5. Unbind de tudo para evitar modificações acidentais
        // O VAO "lembra" dos binds do VBO e EBO, então podemos
desvinculá-los.
        glBindVertexArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }

public:
    static Generic2dShapePtr Make(float* dados_vertices, unsigned int*
indices, int nverts, int n_indices) {
        // é trabalho do caller interpolar as posições e cores
        return Generic2dShapePtr(new Generic2dShape(dados_vertices,
indices, nverts, n_indices));
    }
```

```
    // Destrutor que libera os buffers da GPU
    virtual ~Generic2dShape() {
        glDeleteBuffers(1, &m_vbo);
        glDeleteBuffers(1, &m_ebo);
        glDeleteVertexArrays(1, &m_vao);
    }

    // Função de desenho
    virtual void Draw() {
        glBindVertexArray(m_vao);
        // Desenha índices (3*(nverts-2))
        glDrawElements(mode, n_indices, type, (void*)0);
        glBindVertexArray(0);
    }
};

#endif
```

input_handlers.h:

```
#ifndef INPUT_HANDLERS_H
#define INPUT_HANDLERS_H
#pragma once
#include "gl_includes.h"
#include <iostream>

static void keyboard(GLFWwindow * window, int key, int scancode, int
action, int mods)
{
 if (key == GLFW_KEY_Q && action == GLFW_PRESS)
  glfwSetWindowShouldClose(window, GLFW_TRUE);
}

static void cursorpos(GLFWwindow * win, double xpos, double ypos)
{
 // convert screen pos (upside down) to framebuffer pos (e.g., retina
displays)
 int wn_w, wn_h, fb_w, fb_h;
 glfwGetWindowSize(win, &wn_w, &wn_h);
 glfwGetFramebufferSize(win, &fb_w, &fb_h);
 double x = xpos * fb_w / wn_w;
 double y = (wn_h - ypos) * fb_h / wn_h;
```

```
 std::cout << "(x,y): " << x << ", " << y << std::endl;
}

static void mousebutton(GLFWwindow * win, int button, int action, int
mods)
{
 if (action == GLFW_PRESS) {
  switch (button) {
  case GLFW_MOUSE_BUTTON_1:
   std::cout << "button 1" << std::endl;
   break;
  case GLFW_MOUSE_BUTTON_2:
   std::cout << "button 2" << std::endl;
   break;
  case GLFW_MOUSE_BUTTON_3:
   std::cout << "button 3" << std::endl;
   break;
  }
 }
 if (action == GLFW_PRESS)
  glfwSetCursorPosCallback(win, cursorpos);  // cursor position
callback
 else // GLFW_RELEASE
  glfwSetCursorPosCallback(win, nullptr);   // callback disabled
}

static void resize(GLFWwindow * win, int width, int height)
{
 glViewport(0, 0, width, height);
}

static void setInputCallbacks(GLFWwindow * win) {
    glfwSetFramebufferSizeCallback(win, resize);  // resize callback
    glfwSetKeyCallback(win, keyboard);   // keyboard callback
    glfwSetMouseButtonCallback(win, mousebutton); // mouse button
callback
}

#endif
```

polygon.h:

```
#include <memory>
class Polygon;
```

```
using PolygonPtr = std::shared_ptr<Polygon>;

#ifndef POLYGON_H
#define POLYGON_H
#pragma once
#include "generic2Dshape.h"

class Polygon : public Generic2dShape {
protected:
  Polygon(float* dados_vertices, unsigned int* indices, int nverts, int
n_indices) :
    Generic2dShape(dados_vertices, indices, nverts, n_indices)
    {};
public:
  static PolygonPtr Make (float* posicoes, float* cores, unsigned int*
indices, int nverts, int n_indices) {

    // Interpola posições e cores
    float* dados_vertices = new float[5 * nverts]; // 2 para posição, 3
para cor
    for (int i = 0; i < nverts; i++) {
      dados_vertices[5*i + 0] = posicoes[2*i + 0];
      dados_vertices[5*i + 1] = posicoes[2*i + 1];
      dados_vertices[5*i + 2] = cores[3*i + 0];
      dados_vertices[5*i + 3] = cores[3*i + 1];
      dados_vertices[5*i + 4] = cores[3*i + 2];
    }

    // Cria o Polygon
    PolygonPtr polygon (new Polygon(dados_vertices, indices, nverts,
n_indices));
    glFlush();
    delete[] dados_vertices;
    return polygon;

  }
  virtual ~Polygon () {}
  virtual void Draw () {
    Generic2dShape::Draw();
  }
};
#endif
```

shader.h:

```
#ifndef SHADER_H
#define SHADER_H
#pragma once
#include <memory>
class Shader;
using ShaderPtr = std::shared_ptr<Shader>;

#include "gl_includes.h"
#include "error.h"
#include <fstream>
#include <iostream>
#include <sstream>
#include <cstdlib>

static GLuint MakeShader(GLenum shadertype, const std::string&
filename) {

    GLuint id = glCreateShader(shadertype);
    Error::Check("create shader");

    // errorcheck
    if (id == 0) {
        std::cerr << "Could not create shader object";
        exit(1);
    }

    // open the shader file
    std::ifstream fp;
    fp.open(filename);


    // errorcheck
    if (!fp.is_open()) {
        std::cerr << "Could not open file: " << filename << std::endl;
        exit(1);
    }

    // read the shader file content
    std::stringstream strStream;
    strStream << fp.rdbuf();

    // pass the source string to OpenGL
```

```
    std::string source = strStream.str();
    const char* csource = source.c_str();
    glShaderSource(id, 1, &csource, 0);
    Error::Check("set shader source");

    // tell OpenGL to compile the shader
    GLint status;
    glCompileShader(id);
    glGetShaderiv(id, GL_COMPILE_STATUS, &status);
    Error::Check("compile shader");

    // errorcheck
    if (!status) {
        GLint len;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &len);
        char* message = new char[len];
        glGetShaderInfoLog(id, len, 0, message);
        std::cerr << filename << ":" << std::endl << message <<
std::endl;
        delete [] message;
        exit(1);
    }

    return id;
}

class Shader {
    unsigned int m_pid;
protected:
    Shader() {
        m_pid = glCreateProgram();
        if (m_pid == 0) {
            std::cerr << "Could not create program object";
            exit(1);
        }
    }
public:
    static ShaderPtr Make() {
        return ShaderPtr(new Shader());
    }

    virtual ~Shader(){
        glDeleteProgram(m_pid);
```

```
    }

    void AttachVertexShader(const std::string& filename) {
        GLuint sid = MakeShader(GL_VERTEX_SHADER, filename);
        glAttachShader(m_pid, sid);
    }

    void AttachFragmentShader(const std::string& filename) {
        GLuint sid = MakeShader(GL_FRAGMENT_SHADER, filename);
        glAttachShader(m_pid, sid);
    }

    void Link() {
        glLinkProgram(m_pid);
        GLint status;
        glGetProgramiv(m_pid, GL_LINK_STATUS, &status);
        if (status == GL_FALSE) {
            GLint len;
            glGetProgramiv(m_pid, GL_INFO_LOG_LENGTH, &len);
            char* message = new char[len];
            glGetProgramInfoLog(m_pid, len, 0, message);
            std::cerr << "Shader linking failed: " << message <<
std::endl;
            delete[] message;
            exit(1);
        }
    }
    void UseProgram() const {
        glUseProgram(m_pid);
    }
};

static GLuint educationalMakeShader(GLenum shadertype, const
std::string& filename) {

    GLuint id = glCreateShader(shadertype);

    // open the shader file
    std::ifstream fp;
    fp.open(filename);

    // read the shader file content
    std::stringstream strStream;
```

```
    strStream << fp.rdbuf();

    // pass the source string to OpenGL
    std::string source = strStream.str();
    const char* csource = source.c_str();
    glShaderSource(id, 1, &csource, 0);

    // tell OpenGL to compile the shader
    GLint status;
    glCompileShader(id);
    glGetShaderiv(id, GL_COMPILE_STATUS, &status);

    return id;
}

#endif
```

shape.h:

```
#ifndef SHAPE_H
#define SHAPE_H
#pragma once
#include <memory>
class Shape;
using ShapePtr = std::shared_ptr<Shape>;

class Shape {
protected:
  Shape () {}
public:
  virtual ~Shape () {}
  virtual void Draw () = 0;
};

#endif
```

main.cpp:

```
#define GLAD_GL_IMPLEMENTATION // Necessary for headeronly version.
#include "gl_includes.h"
#include <iostream>

#include "error.h"
#include "input_handlers.h"
```

```
#include "shader.h"

#include "polygon.h"


static GLFWwindow* WindowSetup(int width, int height);
static void error (int code, const char* msg);

ShaderPtr shd;

float posicoes[] = {
  -0.6f, -0.6f,
   0.5f, -0.5f,
   0.5f,  0.5f,
  -0.6f,  0.6f,
   0.0f,  0.3f,
   0.0f, -0.3f,
};

float cores[] = {
  1.0f, 0.0f, 0.0f, // vermelho
  0.0f, 1.0f, 0.0f, // verde
  0.0f, 0.0f, 1.0f, // azul
  0.0f, 1.0f, 1.0f, // ciano
  1.0f, 0.0f, 1.0f, // magenta
  1.0f, 1.0f, 0.0f  // amarelo
};

unsigned int indices[] = {
  0, 1, 5,  // triângulo 1
  1, 2, 5,  // triângulo 2
  2, 4, 5,  // triângulo 3
  2, 3, 4   // triângulo 4
};

PolygonPtr polygon;

static void initialize()
{
  // config do OpenGL
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
  // glFrontFace(GL_CCW);
  // glCullFace(GL_BACK);
```

```
  // glEnable(GL_CULL_FACE);
  // glPolygonMode(GL_FRONT, GL_FILL); // ERRADO

  Error::Check("setup");

  // CENTERPIECE
  // inicia Shader Program
  shd = Shader::Make();
  shd->AttachVertexShader("../shaders/vertex.glsl");
  shd->AttachFragmentShader("../shaders/fragment.glsl");
  shd->Link();

  Error::Check("shaders");
  // inicia geometria estática
  polygon = Polygon::Make(posicoes, cores, indices, 6, 12);
  Error::Check("polygon");
}

static void display(GLFWwindow * win)
{
  // CENTERPIECE
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glFlush();
  shd->UseProgram();

  // desenha geometria
  polygon->Draw();

  // errorcheck
  Error::Check("display");
}

int main(void) {

    GLFWwindow* win = WindowSetup(800, 600);

    setInputCallbacks(win);

    Error::Check("pre initialize");

    initialize();
    glFlush();
    Error::Check("initialize");
```

```
    while (!glfwWindowShouldClose(win)) {
      display(win);
      glfwSwapBuffers(win);
      glFlush();
      glfwPollEvents();
    }

    glfwTerminate();
    return 0;
}
```

```
static void error (int code, const char* msg)
{
  printf("GLFW error %d: %s\n", code, msg);
  glfwTerminate();
  exit(0);
}

static GLFWwindow* WindowSetup(int width, int height) {
  glfwSetErrorCallback(error);
  if (glfwInit() != GLFW_TRUE) {
    std::cerr << "Could not initialize GLFW" << std::endl;
    return 0;
  }
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GLFW_TRUE);

  // DIMENSOES
  GLFWwindow* win = glfwCreateWindow(width, height, "Window title",
nullptr, nullptr);
  if (!win) {
      std::cerr << "Could not create GLFW window" << std::endl;
      return 0;
    }
  glfwMakeContextCurrent(win);
```

```
  #ifdef GLAD_GL_H_
    if (!gladLoadGL(glfwGetProcAddress)) {
      printf("Failed to initialize GLAD OpenGL context\n");
      exit(1);
    }
  #endif

  return win;
}
```

shaders/fragment.glsl:

```
#version 410

in vec4 color;
out vec4 fcolor;

void main (void)
{
  fcolor = color;
}
```

shaders/vertex.glsl:

```
#version 410

layout (location=0) in vec4 vertex;
layout (location=1) in vec4 icolor;

out vec4 color;

void main (void)
{
  color = icolor;
  gl_Position = vertex;
}
```