Relatório do envio do trabalho 1 INF1761

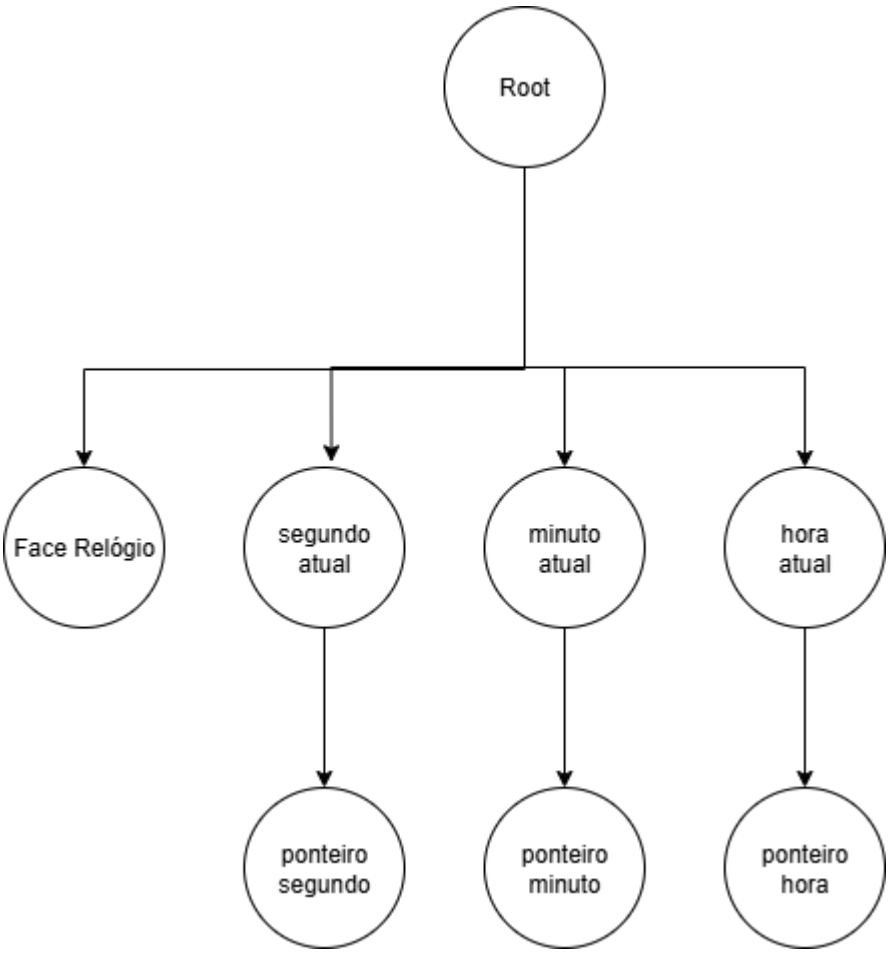
link para o vídeo com o relógio funcionando

Para este trabalho eu fiz a implementação (incompleta mas funcional) de um grafo de cena, que utiliza uma pilha de transformações para conseguir navegar o grafo.

Na main então o que eu faço é apenas definir a geometria que deve ser mostrada, que para este caso é a face do relógio e os ponteiros, e a cada frame atualizo as transformações referentes à cada ponteiro respectivamente.

Criei também uma classe Circle que herda de Polygon para facilitar a definição da face do relógio.

O grafo de cena ficou ao final com a seguinte estrutura:



Segue então, junto do envio um .zip com o projeto, e abaixo o código relevante:

trab2.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 "scene.h"
#include "circle.h"

// ShaderPtr shd;

static void initialize()
{
  // config do OpenGL
  glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
  // glFrontFace(GL_CCW);
  // glCullFace(GL_BACK);
  // glEnable(GL_CULL_FACE);
  // glPolygonMode(GL_FRONT, GL_FILL); // ERRADO

  // inicia Shader Program e SceneGraph

scene::graph()->initializeBaseShader("../shaders/vertex.glsl","../shaders/fragment.glsl");
  // scene::graph()->setView(0,1,0,1,0,1);

  // CENTERPIECE
  // inicia geometria estática
  CirclePtr face_relogio = Circle::Make(
    0.0f, // x_center
    0.0f, // y_center
    0.95f, // radius
    (float[]){1.0f, 1.0f, 1.0f}, // corRGB
    64, // pontos para discretização
    false // se tem ponto no centro ou não
  );
  scene::graph()
  ->getRoot() // não tive tempo ainda de fazer uma interface que abstrai a necessidade de
pegar a raiz
  ->addChild(scene::Node::MakeNode("face_relogio", face_relogio, nullptr,
transform::Transform::Make()));

  PolygonPtr triangulo_ponteiro = Polygon::Make(
    (float[]){ // vertices
      1.0f, 0.0f,
      -1.0f, 0.0f,
      0.0f, 1.0f,
    },
    (float[]){ // cores
      1.0f, 0.0f, 0.0f,
      1.0f, 0.0f, 0.0f,
      1.0f, 0.0f, 0.0f,
    },
    (unsigned int[]){ // incidencia
      0,1,2,
      2,3,4
    },
    3, // numero de vertices
```

```
    3 // numero de indices
  );

  transform::TransformPtr segundo_agora = transform::Transform::Make();
  scene::graph()
  ->getRoot() // adiciona a rotação antes da geometria
  ->addChild(scene::Node::MakeNode("segundo_agora", nullptr, nullptr, segundo_agora));

    // transforma o triangulo base para o formato de um ponteiro
  transform::TransformPtr ponteiro_segundos = transform::Transform::Make();
  ponteiro_segundos->translate(0.0f, 0.01f, 0.1f);
  ponteiro_segundos->scale(0.02f, 0.7f, 1.0f);

  scene::graph()
  ->getRoot()
  ->getChildByName("segundo_agora")
  ->addChild(scene::Node::MakeNode("ponteiro_horas", triangulo_ponteiro, nullptr,
ponteiro_segundos));

  // minuto agora
  transform::TransformPtr minuto_agora = transform::Transform::Make();
  scene::graph()
  ->getRoot() // adiciona a rotação antes da geometria
  ->addChild(scene::Node::MakeNode("minuto_agora", nullptr, nullptr, minuto_agora));

  // adiciona o ponteiro dos minutos
  transform::TransformPtr ponteiro_minutos = transform::Transform::Make();
  ponteiro_minutos->translate(0.0f, 0.01f, 0.1f);
  ponteiro_minutos->scale(0.04f, 0.9f, 1.0f);
  scene::graph()
  ->getRoot()
  ->getChildByName("minuto_agora")
  ->addChild(scene::Node::MakeNode("minuto_agora", triangulo_ponteiro, nullptr,
ponteiro_minutos));

  // hora agora
  transform::TransformPtr hora_agora = transform::Transform::Make();
  scene::graph()
  ->getRoot() // adiciona a rotação antes da geometria
  ->addChild(scene::Node::MakeNode("hora_agora", nullptr, nullptr, hora_agora));

  // adiciona o ponteiro das horas
  transform::TransformPtr ponteiro_horas = transform::Transform::Make();
  ponteiro_horas->translate(0.0f, 0.01f, 0.1f);
  ponteiro_horas->scale(0.04f, 0.4f, 1.0f);
  scene::graph()
  ->getRoot()
  ->getChildByName("hora_agora")
  ->addChild(scene::Node::MakeNode("hora_agora", triangulo_ponteiro, nullptr,
ponteiro_horas));
}

static void display(GLFWwindow * win)
```

```
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // CENTERPIECE
  // atualiza geometria dinâmica

  // pega a hora atual
  time_t now = time(0);
  tm *ltm = localtime(&now);
  int segundos = ltm->tm_sec;
  int minutos = ltm->tm_min;
  int horas = ltm->tm_hour;
  // calcula o ângulo dos ponteiros
  float angulo_segundos = -(360.0f * segundos / 60.0f); // 0 graus é o "topo" do relógio
  float angulo_minutos = -(360.0f * minutos / 60.0f + 6.0f * segundos / 60.0f); // cada
minuto são 6 graus, mais o avanço dos segundos
  float angulo_horas = -(360.0f * (horas % 12) / 12.0f + 30.0f * minutos / 60.0f); // cada
hora são 30 graus, mais o avanço dos minutos

  scene::graph()
  ->getRoot()
  ->getChildByName("segundo_agora")
  ->getTransform()
  ->setRotate(angulo_segundos, 0.0f, 0.0f, 1.0f);

  scene::graph()
  ->getRoot()
  ->getChildByName("minuto_agora")
  ->getTransform()
  ->setRotate(angulo_minutos, 0.0f, 0.0f, 1.0f);

  scene::graph()
  ->getRoot()
  ->getChildByName("hora_agora")
  ->getTransform()
  ->setRotate(angulo_horas, 0.0f, 0.0f, 1.0f);

  // desenha geometria
  scene::graph()->draw();

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

static GLFWwindow* WindowSetup(int width, int height);

int main(void) {

    GLFWwindow* win = WindowSetup(1000, 1000);

    setInputCallbacks(win);

    initialize();
```

```
    while (!glfwWindowShouldClose(win)) {
      display(win);
      glfwSwapBuffers(win);
      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);

  if (!gladLoadGL(glfwGetProcAddress)) {
    printf("Failed to initialize GLAD OpenGL context\n");
    exit(1);
  }
  return win;
}
```

scene.h:

```
#ifndef SCENE_H
#define SCENE_H
#pragma once
```

```cpp
#include "gl_includes.h"
#include "polygon.h"
#include "shape.h"
#include "shader.h"
#include "transform.h"
#include "error.h"
#include <vector>
#include <string>
#include <algorithm>
#include <map>
#include <memory>
#include <cmath>
#include <iostream>
#include <glm/gtc/type_ptr.hpp> // Para glm::value_ptr

// Encapsulamos toda a lógica de desenho em um namespace para organização.
namespace scene {


class Node;
using NodePtr = std::shared_ptr<Node>;

class SceneGraph;
using SceneGraphPtr = std::shared_ptr<SceneGraph>;

class Node { // BACALHAU falta fazer as subclasses de nó
    int id;
    inline static int next_id = 0;
    std::string name;
    ShapePtr shape;
    ShaderPtr shader;
    std::vector<NodePtr> children;
    transform::TransformPtr transform; // Transformação local deste nó

    Node(std::string name, ShapePtr shape, ShaderPtr shader, transform::TransformPtr
transform) :
    name(name), shape(shape), shader(shader), transform(transform)
    {
        id = next_id++;
    }

    public:

        static NodePtr MakeNode(std::string name, ShapePtr shape, ShaderPtr shader,
transform::TransformPtr transform) {
            return NodePtr(new Node(name, shape, shader, transform));
        }

        int getId() {
            return id;
        }

        const std::string& getName() const {
```

```
            return name;
        }

        transform::TransformPtr getTransform() {
            return transform;
        }

        NodePtr getChild(int index) {
            if (index < 0 || index >= children.size()) {
                std::cerr << "Index out of bounds in getChild" << std::endl;
                return nullptr;
            }
            return children[index];
        }

        NodePtr getChildByName(const std::string& child_name) {
            for (NodePtr child : children) {
                if (child->name == child_name) {
                    return child;
                }
            }
            std::cerr << "Child with name " << child_name << " not found in getChildByName"
<< std::endl;
            return nullptr;
        }

        void addChild(NodePtr child) {
            children.push_back(child);
        }

        void addChild(NodePtr child, int index) {
            if (index < 0 || index > children.size()) {
                std::cerr << "Index out of bounds in addChild" << std::endl;
                return;
            }
            children.insert(children.begin() + index, child);
        }

        void addChildFront(NodePtr child) {
            children.insert(children.begin(), child);
        }

        void addChildAfter(NodePtr child, NodePtr after) {
            auto it = std::find(children.begin(), children.end(), after);
            if (it != children.end()) {
                children.insert(it + 1, child);
            } else {
                std::cerr << "Reference child not found in addChildAfter" << std::endl;
            }
        }

        void removeChild(NodePtr child) {
            auto it = std::find(children.begin(), children.end(), child);
```

```
            if (it != children.end()) {
                children.erase(it);
            } else {
                std::cerr << "Child not found in removeChild" << std::endl;
            }
        }

        void setShader(ShaderPtr new_shader) {
            shader = new_shader;
        }

        void draw() {
            printf("Drawing node %s (id=%d)\n", name.c_str(), id);

            Error::Check("scene::Node::draw start");
            transform::TransformStack& transform_stack = transform::stack();
            // Combina a transformação do pai com a transformação local dentro do push
            if (transform) transform_stack.push(transform->getMatrix());

            if (shader) shaderStack().push(shader);

            Error::Check("scene::Node::draw before drawing shape");
            // Desenha a forma associada a este nó, se existir
            if (shape) {

                // Envia a matriz de transformação para o shader
                unsigned int shader_program = shaderStack().top()->GetShaderID();
                unsigned int transformLoc = glGetUniformLocation(shader_program, "M");
                glUniformMatrix4fv(transformLoc, 1, GL_FALSE,
glm::value_ptr(transform_stack.top()));
                shape->Draw();
            }
            Error::Check("scene::Node::draw after drawing shape");
            // Desenha os filhos
            for (NodePtr child : children) {
                child->draw();
            }
            Error::Check("scene::Node::draw after drawing children");
            if (transform) transform_stack.pop();
            if (shader) shaderStack().pop();
            Error::Check("scene::Node::draw end");
        }
};

class SceneGraph {
    private:
        NodePtr root;
        ShaderPtr base_shader;
        std::map<std::string, NodePtr> name_map; // Mapa de nomes para nós
        std::map<int, NodePtr> node_map; // Mapa de IDs para nós
        NodePtr currentNode; // Nó atualmente selecionado
        transform::TransformPtr view_transform;
```

```
        SceneGraph(ShaderPtr base) {
            root = Node::MakeNode("root", nullptr, base, transform::Transform::Make());
            base_shader = base;
            currentNode = root;
            name_map["root"] = root;
            node_map[root->getId()] = root;
            view_transform = transform::Transform::Make();
            view_transform->orthographic(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f); //
Inicializa com ortográfica padrão
        }

        friend SceneGraphPtr graph();

    public:

        // BACALHAU
        NodePtr getRoot() const {
            return root;
        }

        void setView(float left, float right, float bottom, float top, float near, float
far) {
            view_transform->orthographic(left, right, bottom, top, near, far);
        }

        void initializeBaseShader(const std::string& vertex_shader_file, const std::string&
fragment_shader_file) {
            base_shader->AttachVertexShader(vertex_shader_file);
            base_shader->AttachFragmentShader(fragment_shader_file);
            base_shader->Link();
        }

        void clearGraph() {
            root = Node::MakeNode("root", nullptr, nullptr, transform::Transform::Make());
            currentNode = root;
            name_map.clear();
            node_map.clear();
            name_map["root"] = root;
            node_map[root->getId()] = root;
        }

        void draw() {
            // Aplica a transformação de visão
            transform::stack().push(view_transform->getMatrix());
            if (root) {
                root->draw();
            }
            transform::stack().pop();
            printf("\n--------------------------------\n\n");
        }
};
```

```
inline SceneGraphPtr graph() {
    static ShaderPtr default_shader = Shader::Make();
    static SceneGraphPtr instance = SceneGraphPtr(new SceneGraph(default_shader));
    return instance;
}

}

#endif
```

transform.h:

```
#ifndef TRANSFORM_H
#define TRANSFORM_H
#pragma once

#include <memory>
#include "gl_includes.h"
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>


namespace transform {


class Transform;
using TransformPtr = std::shared_ptr<Transform>;

class TransformStack;
using TransformStackPtr = std::shared_ptr<TransformStack>;

class Transform {
    glm::mat4 matrix; // Matriz de transformação 4x4
    Transform() {
        // Inicializa a matriz como identidade
        matrix = glm::mat4(1.0f);
    }
    public:
        static TransformPtr Make() {
            return TransformPtr(new Transform());
        }

        ~Transform()=default;

        const glm::mat4& getMatrix() const {
            return matrix;
        }

        void reset() {
            matrix = glm::mat4(1.0f);
        }

        void multiply(const glm::mat4& other) {
            matrix = matrix * other;
```

```
        }

        void translate(float x, float y, float z) {
            matrix = matrix * glm::translate(matrix, glm::vec3(x, y, z));
        }

        void setTranslate(float x, float y, float z) {
            reset();
            translate(x, y, z);
        }

        void rotate(float angle_degrees, float axis_x, float axis_y, float axis_z) {
            // checks if axis is normalized
            float angle_radians = glm::radians(angle_degrees);
            glm::vec3 axis(axis_x, axis_y, axis_z);

            if (glm::length(axis) > 0.0f) {
                axis = glm::normalize(axis);
            }
            matrix = matrix * glm::rotate(matrix, angle_radians, axis);
        }

        void setRotate(float angle_degrees, float axis_x, float axis_y, float axis_z) {
            reset();
            rotate(angle_degrees, axis_x, axis_y, axis_z);
        }

        void scale(float x, float y, float z) {
            matrix = matrix * glm::scale(matrix, glm::vec3(x, y, z));
        }

        void setScale(float x, float y, float z) {
            reset();
            scale(x, y, z);
        }

        void orthographic(float left, float right, float bottom, float top, float near,
float far) {
            matrix = glm::ortho(left, right, bottom, top, near, far);
        }
};

TransformStack& stack();

class TransformStack {
private:
    std::vector<glm::mat4> stack;

    TransformStack() {
        stack.push_back(glm::mat4(1.0f));
    }

    // A amizade agora é concedida à função livre 'stack()' do namespace.
```

```
    friend TransformStack& stack();

public:
    TransformStack(const TransformStack&) = delete;
    TransformStack& operator=(const TransformStack&) = delete;
    ~TransformStack() = default;

    void push(const glm::mat4& matrix_to_apply) {
        stack.push_back(top() * matrix_to_apply);
    }

    void pop() {
        if (stack.size() > 1) {
            stack.pop_back();
        } else {
            std::cerr << "Warning: Attempt to pop the base identity matrix from the
transform stack." << std::endl;
        }
    }

    const glm::mat4& top() const {
        return stack.back();
    }
};

// Definição da função de acesso (inline para uso no header)
inline TransformStack& stack() {
    static TransformStack instance;
    return instance;
}

}
#endif
```

circle.h:

```
#ifndef CIRCLE_H
#define CIRCLE_H
#pragma once

#include <memory>
class Circle;
using CirclePtr = std::shared_ptr<Circle>;

#include <math.h>
#include "polygon.h"

class Circle : public Polygon { // BACALHAU falta extender para circulos com mais de uma
cor sólida
    float radius;
    unsigned int discretization;
    bool centered;
```

```
    float* geraDadosVertices(float x_center, float y_center, float radius, float* corRGB,
unsigned int edge_points, bool has_center_vertex) {
        int num_points = edge_points;
        if (has_center_vertex) num_points++;

        float* vertices = new float[num_points * 5]; // 2 para posição, 3 para cor
        this->radius = radius;
        this->discretization = edge_points;
        this->centered = has_center_vertex;

        float angle_step = 2.0f * M_PI / edge_points;
        if (has_center_vertex) {
            vertices[0] = x_center;
            vertices[1] = y_center;
            vertices[2] = corRGB[0];
            vertices[3] = corRGB[1];
            vertices[4] = corRGB[2];
        }
        for (unsigned int i = has_center_vertex; i < num_points; i++) {
            float angle = i * angle_step;
            float x = x_center + radius * cos(angle);
            float y = y_center + radius * sin(angle);
            vertices[i * 5 + 0] = x;
            vertices[i * 5 + 1] = y;
            vertices[i * 5 + 2] = corRGB[0];
            vertices[i * 5 + 3] = corRGB[1];
            vertices[i * 5 + 4] = corRGB[2];
        }
        return vertices;
    }

    unsigned int* geraIndices(unsigned int edge_points, bool has_center_vertex) {
        int num_points = edge_points;
        if (has_center_vertex) num_points++;

        unsigned int* indices = new unsigned int[(num_points - 1) * 3];

        for (unsigned int i = 0; i < edge_points; i++) {
            if (has_center_vertex) {
                indices[i * 3 + 0] = 0; // centro
                indices[i * 3 + 1] = i + 1;
                indices[i * 3 + 2] = (i + 1) % edge_points + 1;
            } else {
                indices[i * 3 + 0] = 0;
                indices[i * 3 + 1] = (i + 1) % edge_points;
                indices[i * 3 + 2] = (i + 2) % edge_points;
            }
        }
        return indices;
    }

    int geraNumVertices(unsigned int edge_points, bool has_center_vertex) {
        int num_points = edge_points;
```

```
        if (has_center_vertex) num_points++;
        return num_points;
    }

    int geraNumIndices(unsigned int edge_points, bool has_center_vertex) {
        return (edge_points - 1 + has_center_vertex) * 3;
    }

    Circle(float x_center, float y_center, float radius, float* corRGB, unsigned int
edge_points, bool has_center_vertex) :
        Polygon(
            geraDadosVertices(x_center, y_center, radius, corRGB, edge_points,
has_center_vertex),
            geraIndices(edge_points, has_center_vertex),
            geraNumVertices(edge_points, has_center_vertex),
            geraNumIndices(edge_points, has_center_vertex),
            {3}
        )
        {};
public:
    static CirclePtr Make(float x_center, float y_center, float radius, float* corRGB,
unsigned int edge_points, bool has_center_vertex) {
        CirclePtr circle (new Circle(x_center, y_center, radius, corRGB, edge_points,
has_center_vertex));
        glFlush();
        return circle;
    }
    virtual ~Circle () {}
    virtual void Draw () {
        Shape::Draw();
    }
};

#endif
```

polygon.h:

```
#include <memory>
class Polygon;
using PolygonPtr = std::shared_ptr<Polygon>;

#ifndef POLYGON_H
#define POLYGON_H
#pragma once
#include "gl_includes.h"
#include "shape.h"

class Polygon : public Shape {
protected:
  Polygon(float* dados_vertices, unsigned int* indices, int nverts, int
n_indices, const std::vector<int>& attr_sizes = {3}) :
```

```
    Shape(dados_vertices, indices, nverts, n_indices, attr_sizes)
    {};
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;

  }
  static PolygonPtr Make (float* dados_vertices, unsigned int* indices,
int nverts, int n_indices) {
    PolygonPtr polygon (new Polygon(dados_vertices, indices, nverts,
n_indices));
    glFlush();
    return polygon;
  }
  virtual ~Polygon () {}
  virtual void Draw () {
    Shape::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>
#include <vector>

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);
    }

    unsigned int GetShaderID() const {
        return 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);
    }
};

class ShaderStack { // singleton
    // BACALHAU falta adaptar para ter o batching dos comandos de draw pelo shader
private:
    std::vector<ShaderPtr> stack;
    ShaderPtr last_used_shader;

    ShaderStack() {
        stack.push_back(Shader::Make());
    }

    // A amizade agora é concedida à função livre 'shaderStack()' do namespace.
    friend ShaderStack& shaderStack();
public:
    ShaderStack(const ShaderStack&) = delete;
    ShaderStack& operator=(const ShaderStack&) = delete;
    ~ShaderStack() = default;
    void push(ShaderPtr shader) {
        // only push if it's different from the current top
        if (shader != stack.back()) {
            shader->UseProgram();
            last_used_shader = shader;
            stack.push_back(shader);
        }
    }
    void pop() {
        if (stack.size() > 1) {
            stack.pop_back();
        } else {
            std::cerr << "Warning: Attempt to pop the base shader from the shader stack."
<< std::endl;
        }
    }
    ShaderPtr top() {
        ShaderPtr current_shader = stack.back();
        if (current_shader != last_used_shader) {
            current_shader->UseProgram();
            last_used_shader = current_shader;
        }
        return current_shader;
    }
    unsigned int topId() {
        return top()->GetShaderID();
    }
    ShaderPtr getLastUsedShader() const {
```

```
        return last_used_shader;
    }
};

inline ShaderStack& shaderStack() {
    static ShaderStack instance;
    return instance;
}

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
```