

| | | |
|----------------------------|---------------------|----------------------------------|
| Grupa: IO gr.1 | Ćwiczenie: Lab 7 | Imię Nazwisko: Malwina Cieśla |
| Wizualizacja Danych | | |

Cel ćwiczenia:

Zapoznanie z funkcjami realizującymi oświetlenie sceny. Ustawianie właściwości materiału oraz właściwości źródeł światła.

Przebieg ćwiczenia:

W celu określenia sceny wyglądającej maksymalnie realistycznie konieczne jest zastosowanie oświetlenia. Światła oraz materiały posiadają właściwości odpowiedzialne za wygląd oświetlanych obiektów. Rozróżniamy 3 rodzaje światel:

1. światła kierunkowe – ich promienie są równoległe, ich natężenie jest jednakowe.
2. światła punktowe – nie posiadają kierunku, oświetlają całą scenę, a natężenie zależy od odległości,
3. reflektory – świecą w danym kierunku z określonej pozycji. Snop światła mieści się w ściśle określonym stożku

W kodzie należało zaimportować obiekt do VBO wraz z informacjami o wektorach normalnych oraz przekazać informacje o wektorach normalnych określając specyfikację formatu danych wierzchołkowych:

```
GLint NorAttrib = glGetAttribLocation(shaderProgram, "aNormal"); glEnableVertexAttribArray(NorAttrib);
glVertexAttribPointer(NorAttrib, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void*)(3 * sizeof(GLfloat)));
```

Ilustracja 1: Dodane zmiany

Następnie należało ustalić położenie światła:

```
glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
GLint uniLightPos = glGetUniformLocation(shaderProgram, "lightPos");
glUniform3fv(uniLightPos, 1, &lightPos[0]);
```

Ilustracja 2: Położenie światła

Później należało w kodzie vertex shadera odebrać informację o wektorach normalnych i przekazać do fragment shadera oraz zadeklarować zmienną do przekazania pozycji fragmentów. W kodzie fragment shadera należało odebrać informację o wektorach normalnych z vertex shadera, dodać moc i kolor światła otoczenia poprzez przemnożenie aktualnego koloru fragmentu przez składową światła oraz dodać składową koloru światła rozproszonego. Dzięki temu wyznaczamy wektor kierunku między źródłem światła, a pozycją fragmentu i normalizujemy zarówno normalny, jak i wynikowy wektor kierunkowy, ustalamy wpływ składowej rozproszonej światła na bieżący fragment, biorąc iloczyn skalarny normy i wektora lightDir (dla kąta między wektorami >90 stopni wynik jest ujemny dlatego zastosowano funkcję max), a ostatecznie mając dwie składowe otoczenia i światła rozproszonego dodajemy je do siebie i mnożymy przez bieżący kolor fragmentu:

```

in vec3 aNormal;
in vec3 lightPos;
out vec3 Normal;
out vec3 FragPos;
in vec2 aTexCoord;
out vec2 TexCoord;
uniform mat4 model;
uniform mat4 view;
uniform mat4 proj;

void main(){
    TexCoord = aTexCoord;
    Color = color;
    Normal=aNormal;

```

Ilustracja 4: Vertex Shader

```

in vec3 Normal;
in vec3 FragPos;
in vec3 lightPos;
uniform sampler2D texture1;

void main(){
    outColor = vec4(Color, 1.0);
    outColor=texture(texture1, TexCoord);
    float ambientStrength = 0.1;// ambient
    vec3 ambientlightColor = vec3(1.0,1.0,1.0);
    vec4 ambient = ambientStrength * vec4(ambientlightColor,1.0);
    vec3 diffflightColor = vec3(1.0,1.0,1.0);
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos -FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * diffflightColor;
    outColor = (ambient+vec4(diffuse,1.0)) * texture(texture1, TexCoord);

```

Ilustracja 3: Fragment Shader

Dodatkowo na klawiszach „Z”, „W” oraz „S” dodałam możliwości zmieniania natężenia oświetlenia:

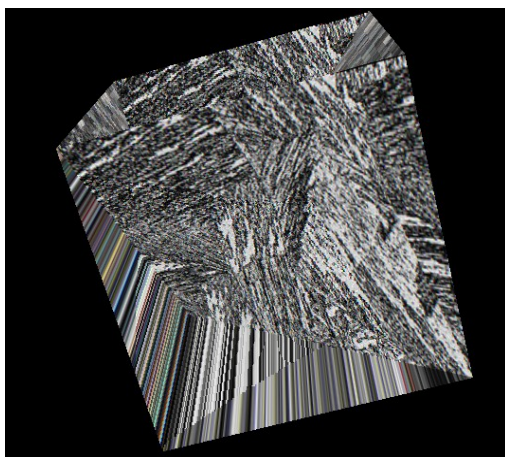
```

case sf::Keyboard::Z:
    if (turnOn == 0)
        turnOn = 1;
    else if (turnOn == 1)
        turnOn = 0;
    glUniform1f(uniTurnOn, turnOn);
    break;
case sf::Keyboard::W:
    ambientStrength += 0.1;
    glUniform1f(uniAmbientStrength, ambientStrength);
    break;
case sf::Keyboard::S:
    ambientStrength -= 0.1;
    glUniform1f(uniAmbientStrength, ambientStrength);
    break;

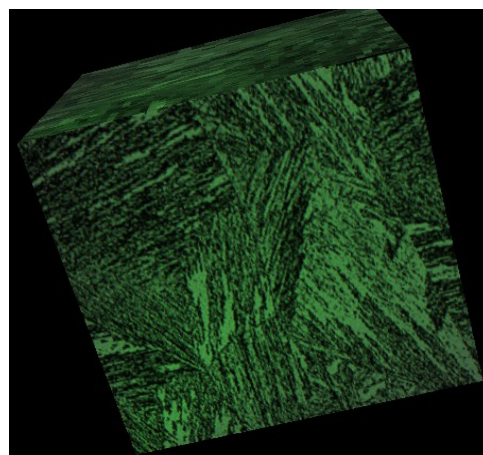
```

Ilustracja 5: Zmiany w oświetleniu

Dzięki temu po uruchomieniu programu mogłam uzyskać poniższe wyniki:



Ilustracja 6: Rozjaśniony



Ilustracja 7: Przyciemniony

Wnioski:

Oświetlenie sześcianu jest pomocne w zobrazowaniu figury w przestrzeni. Jednak należy pamiętać o tym jaki rodzaj oświetlenia stosujemy, ponieważ od rodzaju użytego oświetlenia zależy implementacja naszego kodu. Dodatkowo obraz zmienia się poprzez dodanie oświetlenia lub jego usunięcie, dzięki czemu możemy manipulować kolorem naszej wizualizacji,

KOD

```
#include <iostream>
#include <windows.h>
#include <GL/glew.h>
#include <SFML/Window.hpp>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SFML/System/Time.hpp>
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

```
const GLchar* vertexSource = R"glsl(
#version 150 core
    in vec3 position;
    in vec3 color;
    out vec3 Color;
    uniform mat4 uniformModel;
    uniform mat4 uniformView;
    uniform mat4 uniformProj;
    in vec2 aTexCoord;
    out vec2 TexCoord;
    in vec3 aNormal;
    out vec3 Normal;
    out vec3 FragPos;

    void main(){
        Color = color;
        TexCoord= aTexCoord;
        Normal=aNormal;
        gl_Position = uniformProj * uniformView * uniformModel * vec4(position,
1);
        FragPos = vec3( uniformModel* vec4(position, 1.0));

    }
)glsl";
```

```
const GLchar* fragmentSource = R"glsl(
#version 150 core
    in vec3 Color;
    out vec4 outColor;
    in vec2 TexCoord;
    uniform sampler2D texture1;
    in vec3 Normal;
    in vec3 FragPos;
    in vec3 diffuse;
    uniform vec3 lightPos;
    uniform float turnOn;
    uniform float ambientStrength;

    void main() {
        vec3 ambientlightColor = vec3(1.0,1.0,1.0);
```

```

        vec4 ambient = ambientStrength * vec4(ambientlightColor,1.0);
        vec3 diffLightColor = vec3(0.0,0.50,0.0);
        vec3 norm = normalize(Normal);
        vec3 lightDir = normalize(lightPos - FragPos);
        float diff = max(dot(norm, lightDir), 0.0);
        vec3 diffuse = diff * diffLightColor;
        if(turnOn==0)
            outColor = (ambient+vec4(diffuse,1.0)) * texture(texture1, TexCoord);
        else if(turnOn==1)
            outColor = texture(texture1, TexCoord);
        }
    }glsl";
    GLboolean isShaderCompiled(GLuint shader);

double obrot = 5;
glm::vec3 cameraPos = glm::vec3(0.0f, 1.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
float ambientStrength = 0.1;
float turnOn = 1;
bool fMouse = true;
int lastX, lastY;
double yaw = -90;
double pitch = 0;
void ustawKamereKlawisze(GLint view, float time) {

    float cameraSpeed = 0.000002f * time;

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
        cameraPos += cameraSpeed * cameraFront;
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
        cameraPos -= cameraSpeed * cameraFront;
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;

    glm::mat4 thisView;
    thisView = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

    glUniformMatrix4fv(view, 1, GL_FALSE, glm::value_ptr(thisView));
}
void ustawKamereMysz(GLint uniformView, float time, const sf::Window& window) {

    sf::Vector2i localPosition = sf::Mouse::getPosition(window);
    bool reloc = false;
    sf::Vector2i position;
    if (localPosition.x <= 0) {
        position.x = window.getSize().x;
        position.y = localPosition.y;
        reloc = true;
    }
}

```

```

    }
    if (localPosition.x >= window.getSize().x - 1) {
        position.x = 0;
        position.y = localPosition.y;
        reloc = true;
    }
    if (localPosition.y <= 0) {
        position.y = window.getSize().y;
        position.x = localPosition.x;
        reloc = true;
    }
    if (localPosition.y >= window.getSize().y - 1) {
        position.y = 0;
        position.x = localPosition.x;
        reloc = true;
    }
    if (reloc) {
        sf::Mouse::setPosition(position, window);
        fMouse = true;
    }

    localPosition = sf::Mouse::getPosition(window);

    if (fMouse) {
        lastX = localPosition.x;
        lastY = localPosition.y;
        fMouse = false;
    }

    float xoffset = localPosition.x - lastX;
    float yoffset = localPosition.y - lastY;
    lastX = localPosition.x;
    lastY = localPosition.y;

    double sensitivity = 0.3f;
    xoffset *= sensitivity;
    yoffset *= sensitivity;
    yaw += xoffset;
    pitch -= yoffset;

    if (pitch > 89.0f) pitch = 89.0f;
    if (pitch < -89.0f) pitch = -89.0f;

    glm::vec3 front;

    front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));

    cameraFront = glm::normalize(front);

    glm::mat4 view;

```

```

view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
glUniformMatrix4fv(uniformView, 1, GL_FALSE, glm::value_ptr(view));
}
int main(){
    sf::ContextSettings settings;
    settings.depthBits = 24;
    settings.stencilBits = 8;

    sf::Window window(sf::VideoMode(800, 800, 32), "OpenGL", sf::Style::Titlebar |
sf::Style::Close, settings);

    window.setMouseCursorGrabbed(true);
    window.setMouseCursorVisible(false);
    glewExperimental = GL_TRUE;
    glewInit();
    unsigned int texture1;
    glGenTextures(1, &texture1);
    glBindTexture(GL_TEXTURE_2D, texture1);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    int width, height, nrChannels;
    stbi_set_flip_vertically_on_load(true);
    unsigned char* data = stbi_load("met.bmp", &width, &height, &nrChannels, 0);
    if (data){
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else
        std::cout << "Failed to load texture" << std::endl;
    stbi_image_free(data);

    GLuint vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    GLuint vbo;
    glGenBuffers(1, &vbo);

    float vertices[] = {
-0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f,
0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f,
-0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
-0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
-0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.5f, -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
0.5f, 0.5f, 0.5f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f,
0.5f, 0.5f, 0.5f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f,
-0.5f, 0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
-0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,

```

```

-0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f,1.0f,
-0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,1.0f,1.0f,
-0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,1.0f,0.0f,
-0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,1.0f,0.0f,
-0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f,0.0f,0.0f,
-0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f,1.0f,
0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f,1.0f,
0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,1.0f,1.0f,
0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,1.0f,0.0f,
0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,1.0f,0.0f,
0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f,0.0f,0.0f,
0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f,1.0f,
-0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,1.0f,0.0f,
0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f,1.0f,1.0f,
0.5f, -0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f,1.0f,
0.5f, -0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f,1.0f,
-0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f,0.0f,0.0f,
-0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,1.0f,0.0f,
-0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 0.0f,1.0f,0.0f,
0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,1.0f,1.0f,
0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f,1.0f,
0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f,1.0f,
-0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 0.0f,0.0f,0.0f,
-0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 0.0f,1.0f,0.0f
};
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glBindTexture(GL_TEXTURE_2D, texture1);

GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);
if (isShaderCompiled(vertexShader) == GL_FALSE)
    std::cout << "Vertex shader compilation error" << std::endl;
else
    std::cout << "Vertex shader compilation OK" << std::endl;

GLint status;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &status);

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
glCompileShader(fragmentShader);

if (isShaderCompiled(fragmentShader) == GL_FALSE)
    std::cout << "Fragment shader compilation error" << std::endl;
else
    std::cout << "Fragment shader compilation OK" << std::endl;

GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);

```

```

glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);

glm::mat4 glmModel = glm::mat4(1.0f);
glmModel = glm::rotate(glmModel, glm::radians(15.0f), glm::vec3(0.0f, 0.0f, 1.0f));

GLint modelTransition = glGetUniformLocation(shaderProgram, "uniformModel");
glUniformMatrix4fv(modelTransition, 1, GL_FALSE, glm::value_ptr(glmModel));

GLint uniformView = glGetUniformLocation(shaderProgram, "uniformView");

glm::mat4 glmProj = glm::perspective(glm::radians(45.0f), (800.0f / 800.0f), 0.06f, 100.0f);
GLint uniformProj = glGetUniformLocation(shaderProgram, "uniformProj");
glUniformMatrix4fv(uniformProj, 1, GL_FALSE, glm::value_ptr(glmProj));

GLint primitive = GL_TRIANGLES;
GLint mouseX = 0, mouseY = 0;
sf::Clock clock;
sf::Time time;
window.setFramerateLimit(20);

int counter = 0;
glEnable(GL_DEPTH_TEST);

GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void*)(3
* sizeof(GLfloat)));

GLint texCoord = glGetAttribLocation(shaderProgram, "aTexCoord");
glEnableVertexAttribArray(texCoord);
glVertexAttribPointer(texCoord, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void*)(6
* sizeof(GLfloat)));

GLint NorAttrib = glGetAttribLocation(shaderProgram, "aNormal");
glEnableVertexAttribArray(NorAttrib);
glVertexAttribPointer(NorAttrib, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void*)(3
* sizeof(GLfloat)));

glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
GLint uniLightPos = glGetUniformLocation(shaderProgram, "lightPos");
glUniform3fv(uniLightPos, 1, &lightPos[0]);

GLint uniTurnOn = glGetUniformLocation(shaderProgram, "turnOn");
glUniform1f(uniTurnOn, turnOn);

GLint uniAmbientStrength = glGetUniformLocation(shaderProgram, "ambientStrength");

```



```

glUniform1f(uniAmbientStrength, ambientStrength);

bool isRunning = true;
while (isRunning) {
    time = clock.getElapsedTime();
    clock.restart();
    counter++;
    float fps = 1000000 / time.asMicroseconds();
    if (counter > fps) {
        window.setTitle(std::to_string(fps));
        counter = 0;
    }
    sf::Event winEvent;
    while (window.pollEvent(winEvent)) {
        switch (winEvent.type) {
            case sf::Event::Closed:
                isRunning = false;
                break;
            case sf::Event::KeyPressed:
                switch (winEvent.key.code) {
                    case sf::Keyboard::Escape:
                        isRunning = false;
                        break;
                    case sf::Keyboard::Num1:
                        primitive = GL_POINTS;
                        std::cout << "POINTS" << std::endl; break;
                    case sf::Keyboard::Num2:
                        primitive = GL_LINES;
                        std::cout << "LINES" << std::endl; break;
                    case sf::Keyboard::Num3:
                        primitive = GL_LINE_STRIP;
                        std::cout << "LINE STRIP" << std::endl; break;
                    case sf::Keyboard::Num4:
                        primitive = GL_LINE_LOOP;
                        std::cout << "LINE LOOP" << std::endl; break;
                    case sf::Keyboard::Num5:
                        primitive = GL_TRIANGLES;
                        std::cout << "TRIANGLES" << std::endl; break;
                    case sf::Keyboard::Num6:
                        primitive = GL_TRIANGLE_STRIP;
                        std::cout << "TRIANGLES STRIP" << std::endl; break;
                    case sf::Keyboard::Num7:
                        primitive = GL_TRIANGLE_FAN;
                        std::cout << "TRIANGLE FAN" << std::endl; break;
                    case sf::Keyboard::Num8:
                        primitive = GL_QUADS;
                        std::cout << "QUADS" << std::endl; break;
                    case sf::Keyboard::Num9:
                        primitive = GL_QUAD_STRIP;
                        std::cout << "QUAD STRIP" << std::endl; break;
                    case sf::Keyboard::Num0:
                        primitive = GL_POLYGON;

```

```

        std::cout << "POLYGON" << std::endl; break;
    case sf::Keyboard::Z:
        if (turnOn == 0)
            turnOn = 1;
        else if (turnOn == 1)
            turnOn = 0;
        glUniform1f(uniTurnOn, turnOn);
        break;
    case sf::Keyboard::W:
        ambientStrength += 0.1;
        glUniform1f(uniAmbientStrength, ambientStrength);
        break;
    case sf::Keyboard::S:
        ambientStrength -= 0.1;
        glUniform1f(uniAmbientStrength, ambientStrength);
        break;
    }
    break;
case sf::Event::MouseMoved:
    ustawKamereMysz(uniformView, time.asMicroseconds(), window);
    break;
}
}
ustawKamereKlawisze(uniformView, time.asMicroseconds());

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glDrawArrays(primitive, 0, 36);

window.display();
}
glDeleteProgram(shaderProgram);
glDeleteShader(fragmentShader);
glDeleteShader(vertexShader);
glDeleteBuffers(1, &vbo);
glDeleteVertexArrays(1, &vao);
window.close();
return 0;
}

GLboolean isShaderCompiled(GLuint shader) {
    GLint isCompiled = 0;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &isCompiled);
    if (isCompiled == GL_FALSE){
        GLint maxLength = 0;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &maxLength);
        std::string log(maxLength, ' ');
        glGetShaderInfoLog(shader, maxLength, &maxLength, &log[0]);
        std::cout << "Error log: " << log << std::endl;
        glDeleteShader(shader);
        return GL_FALSE;
    }
}

```

```
    else
        return GL_TRUE;
}
```