Grupa: IO gr.1	Ćwiczenie: Lab 6	Imię Nazwisko: Malwina Cieśla
Wizualizacja Danych		

Cel ćwiczenia:

Zapoznanie z funkcjami realizującymi wczytywanie tekstur oraz mechanizmami nakładania tekstur na wielokąty.

Przebieg ćwiczenia:

Pierwszą rzeczą, którą trzeba było wykonać, aby używać tekstur, było załadowanie ich do aplikacji. Użytym rozwiązaniem jest zastosowanie biblioteki ładującej obraz stb_image.h. Aby tekstura mogła zostać prawidłowo nałożona na wielokąt konieczne jest podanie koordynat tekstury. Konieczna jest modyfikacja tablicy z danymi o wierzchołkach o dwie dodatkowe zmienne. Również należy poinformować OpenGL o rozmiarze generowanego bufora. Następnie, aby była możliwość przekazania tych informacji do shadera konieczne było określenie pozycji tych danych:

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

Ilustracja 1: Określenie pozycji danych

Następnie konieczna była zmiana kodu vertex shadera, aby przyjąć współrzędne tekstury jako atrybut wierzchołka, a następnie przekazać współrzędne do modułu vertex shader. Do shadera trzeba było przekazać jeszcze teksturę. Można to zrobić z użyciem tzw. samplera, a ustawienie kolorów fragmentów można uzyskać wywołaniem w funkcji main():

```
const GLchar* vertexSource
#version 150 core
in vec2 aTexCoord;
out vec2 TexCoord;
uniform mat4 model;
uniform mat4 view;
uniform mat4 proj;
in vec3 position;
in vec3 color;
out vec3 Color;
void main(){
Color = color;
gl_Position = proj * view
TexCoord = aTexCoord;
}
```

Ilustracja 2: Shader vertexów

```
const GLchar* fragmentSource = R"glsl(
#version 150 core
in vec2 TexCoord;
in vec3 Color;
out vec4 outColor;
uniform sampler2D texture1;
void main(){
outColor = vec4(Color, 1.0);
outColor=texture(texture1, TexCoord);
}
```

Ilustracja 3: Shader fragmentów

Dodatkowo należało dodać kod ładujący plik graficzny i tworzący teksturę. Fragment ten bazuje na funkcjach dostępnych w bibliotece stb_image.h:

```
unsigned int texture1; //Jak do wszystkichobiektów w OpenGL do tekstury można odwołaćsię poprz
glGenTextures(1, &texture1);// Generuje tekstury
glBindTexture(GL_TEXTURE_2D, texture1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);// set texture filtering paramete
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);// load image, create texture
int width, height, nrChannels;
stbi_set_flip_vertically_on_load(true); // tell stb_image.h to flip loaded texture's on the y-
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 {
    cout << "Failed to load texture" << endl;
}
stbi_image_free(data);
```

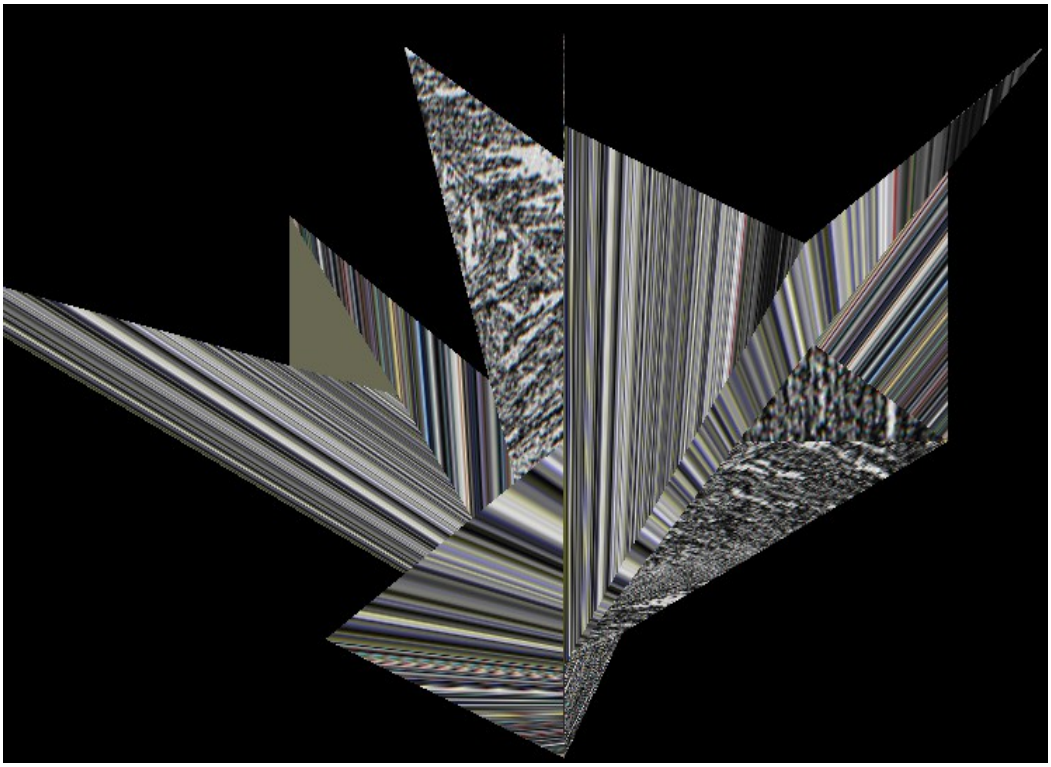
Ilustracja 4: Kod z wykorzystaniem funkcji biblioteki stb_image.h

Ostatecznie przed wywołaniem funkcji rysującej trzeba było ustawić teksturę na bieżącą:

```
glBindTexture(GL_TEXTURE_2D, texture1);
```

Ilustracja 5: Ustawienie tekstury

Dzięki temu po skompilowaniu i uruchomieniu programu mogłam uzyskać figurę wyteksturowaną przy pomocy obrazka przedstawiającego wewnętrzną budowę metalu:



Ilustracja 6: Powstała figura

Wnioski:

Tekstury są obiektami, które muszą zostać wygenerowane przez wywołanie odpowiednich funkcji. Tekstury są zazwyczaj używane jako "obrazy do dekoracji" modeli 3D, ale w rzeczywistości mogą być używane do przechowywania wielu różnych rodzajów danych. Używana w tym zadaniu biblioteka stb_image.h pozwala nam na dodanie tekstur do rysowane figury. Ponieważ współrzędne tekstur są niezależne od rozdzielczości, nie zawsze odpowiadają dokładnie położeniom pikseli. Dzieje się tak, gdy obraz tekstur jest rozciągany powyżej (lub ściskany poniżej) oryginalnego rozmiaru. W takich sytuacjach OpenGL oferuje różne metody decydowania o próbkowanych kolorach, a proces ten nazywa się filtrowaniem. W moim kodzie użyty jest GL_LINEAR, który zwraca średnią ważoną 4 tekseli otaczających podane współrzędne.

KOD

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


glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

const GLchar* vertexSource = R"glsl(
#version 150 core
in vec2 aTexCoord;
out vec2 TexCoord;
uniform mat4 model;
uniform mat4 view;
uniform mat4 proj;
in vec3 position;
in vec3 color;
out vec3 Color;
void main(){
Color = color;
gl_Position = proj * view * model * vec4(position, 1.0);
TexCoord = aTexCoord;
}
)glsl";

const GLchar* fragmentSource = R"glsl(
#version 150 core
in vec2 TexCoord;
in vec3 Color;
out vec4 outColor;
uniform sampler2D texture1;
void main(){
outColor = vec4(Color, 1.0);
outColor=texture(texture1, TexCoord);
```

```
}
)glsl";

bool firstMouse = true;
int lastX, lastY;
double yaw = -90;
double pitch=0;
double obrot = 5;
void ustawKamereKlawisz(GLint view, float _time) {
	float cameraSpeed = 0.001f*_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;
	}
	if (sf::Keyboard::isKeyPressed(sf::Keyboard::L)) {
		obrot -= cameraSpeed;
		cameraFront.x = sin(obrot);
		cameraFront.z = -cos(obrot);
	}
	glm::mat4 thisView;
	thisView = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
	glUniformMatrix4fv(view, 1, GL_FALSE, glm::value_ptr(thisView));
}
void ustawKamereMysz(GLint _uniView, float _time, const sf::Window& _window) {

	sf::Vector2i localPosition = sf::Mouse::getPosition(_window);
	sf::Vector2i position;
	bool relocation = false;

	if (localPosition.x <= 0) {
		position.x = _window.getSize().x;
		position.y = localPosition.y;
		relocation = true;
	}
	if (localPosition.x >= int(_window.getSize().x) - 1) {
		position.x = 0;
		position.y = localPosition.y;
		relocation = true;
	}
	if (localPosition.y <= 0) {
		position.y = _window.getSize().y;
		position.x = localPosition.x;
		relocation = true;
	}
	if (localPosition.y >= int(_window.getSize().y) - 1) {
		position.y = 0;
		position.x = localPosition.x;
		relocation = true;
	}
```

```
	if (relocation) {
		sf::Mouse::setPosition(position, _window);
		firstMouse = true;
	}
	localPosition = sf::Mouse::getPosition(_window);

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

	double xoffset = double(localPosition.x) - lastX;
	double yoffset = double(localPosition.y) - lastY;
	lastX = localPosition.x;
	lastY = localPosition.y;

	double sensitivity = 0.8f;
	double cameraSpeed = 0.00002f * _time;
	xoffset *= sensitivity;
	yoffset *= sensitivity;

	yaw += xoffset * cameraSpeed;
	pitch -= yoffset * cameraSpeed;

	if (pitch > 89.0f)
		pitch = 89.0f;
	if (pitch < -89.0f)
		pitch = -89.0f;
	glm::vec3 front;
	front.x = float(cos(glm::radians(yaw)) * cos(glm::radians(pitch)));
	front.y = float(sin(glm::radians(pitch)));
	front.z = float(sin(glm::radians(yaw)) * cos(glm::radians(pitch)));
	cameraFront = glm::normalize(front);

	glm::mat4 view;
	view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
	glUniformMatrix4fv(_uniView, 1, GL_FALSE, glm::value_ptr(view));

}
int main(){

	sf::ContextSettings settings;
	settings.depthBits = 24;
	settings.stencilBits = 8;
	sf::Window window(sf::VideoMode(800, 600, 32),"OpenGL", sf::Style::Titlebar | sf::Style::Close,
settings);
	window.setMouseCursorGrabbed(true);
	window.setMouseCursorVisible(false);
	glewExperimental = GL_TRUE;
	glewInit();
	window.setFramerateLimit(20);

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

```
	GLuint vbo;
	glGenBuffers(1, &vbo);
	unsigned int texture1;
	glGenTextures(1, &texture1);
	glBindTexture(GL_TEXTURE_2D, texture1);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	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 {
		cout << "Failed to load texture" << endl;
	}
	stbi_image_free(data);
	GLfloat 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, 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, 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, 0.0f,
		-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,1.0f, 0.0f,

		0.5f, -0.5f, 0.5f, 1.0f, 0.0f, 0.0f,0.0f, 0.0f,
		0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f,1.0f, 1.0f,
		-0.5f, -0.5f, -0.5f, 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,1.0f, 0.0f,

		-0.5f, 0.5f, 0.5f, 0.0f, 1.0f, 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, 1.0f, 0.0f, 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, 0.0f, 1.0f, 0.0f,0.0f, 1.0f,

		0.5f, 0.5f, 0.5f, 0.0f, 1.0f, 0.0f,0.0f, 1.0f,
		0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,1.0f, 1.0f,
		0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 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, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,};
```

```
	glBindBuffer(GL_ARRAY_BUFFER, vbo);
	glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
	cout << "Compilation vertexShader: ";
	GLuint vertexShader =
		glCreateShader(GL_VERTEX_SHADER);
	glShaderSource(vertexShader, 1, &vertexSource, NULL);
	glCompileShader(vertexShader);
	GLint status;
	glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &status);
	if (status == GL_TRUE)
		cout << "OK" << endl;
	else {
		char buffer[512];
		glGetShaderInfoLog(vertexShader, 512, NULL, buffer);
	}

	cout << "Compilation fragmentShader: ";
	GLuint fragmentShader =
		glCreateShader(GL_FRAGMENT_SHADER);
	glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
	glCompileShader(fragmentShader);
	GLint status2;
	glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &status2);
	if (status2 == GL_TRUE)
		cout << "OK" << endl;
	else {
		char buffer[512];
		glGetShaderInfoLog(fragmentShader, 512, NULL, buffer);
	}

	GLuint shaderProgram = glCreateProgram();
	glAttachShader(shaderProgram, vertexShader);
	glAttachShader(shaderProgram, fragmentShader);
	glBindFragDataLocation(shaderProgram, 0, "outColor");
	glLinkProgram(shaderProgram);
	glUseProgram(shaderProgram);

	GLint TexCoord = glGetAttribLocation(shaderProgram, "aTexCoord");
	glEnableVertexAttribArray(TexCoord);
	glVertexAttribPointer(TexCoord, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void*)(6 *
sizeof(GLfloat)));
	GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
	glEnableVertexAttribArray(posAttrib);
	glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), 0);
	GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
	glEnableVertexAttribArray(colAttrib);
	glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (void*)(2 *
sizeof(GLfloat)));
	glm::mat4 model = glm::mat4(1.0f);
	model = glm::rotate(model, glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
	GLint uniTrans = glGetUniformLocation(shaderProgram, "model");
	glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(model));

	glm::vec3 direction;
	direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
	direction.y = sin(glm::radians(pitch));
	direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
```

```
	float lastX = 400, lastY = 300;
	const float cameraSpeed = 0.05f;
	glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
	glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);

	glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
	glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
	glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
	glm::mat4 view;
	view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f,
0.0f));
	view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
	GLint uniView = glGetUniformLocation(shaderProgram, "view");
	glUniformMatrix4fv(uniView, 1, GL_FALSE, glm::value_ptr(view));

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

	glBindTexture(GL_TEXTURE_2D, texture1);
	sf::Clock clock;
	sf::Time time;
	bool running = true;
	while (running) {
		time = clock.getElapsedTime();
		clock.restart();
		sf::Event windowEvent;
		while (window.pollEvent(windowEvent)) {
			switch (windowEvent.type) {
			case sf::Event::Closed:
				running = false;
				break;
			case sf::Event::MouseMoved:
				ustawKamereMysz(uniView, time.asMicroseconds(), window);
				break;
			}
		}
		sf::Event::KeyPressed;
		switch (windowEvent.key.code) {
		case sf::Keyboard::Escape:
			window.close();
			break;
		}
		ustawKamereKlawisz(uniView, time.asMicroseconds());
		glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
		glClear(GL_COLOR_BUFFER_BIT);
		glDrawArrays(GL_POLYGON, 0, 30);
		window.display();
	}
	glDeleteProgram(shaderProgram);
	glDeleteShader(fragmentShader);
	glDeleteShader(vertexShader);
	glDeleteBuffers(1, &vbo);
	glDeleteVertexArrays(1, &vao);
	window.close();
	return 0;
}
```