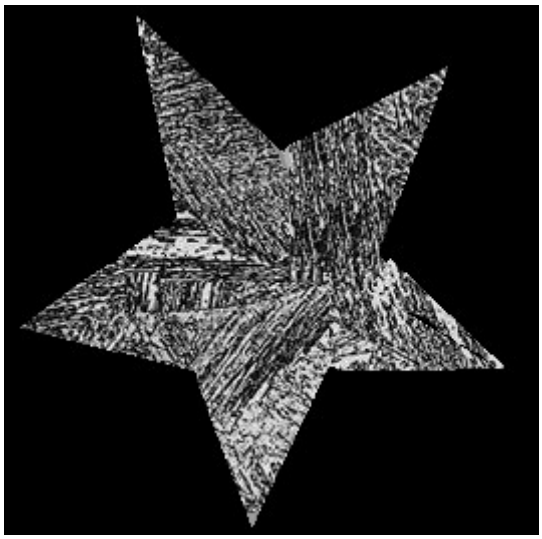
Grupa: IO gr.1	Ćwiczenie: Lab 9	Imię Nazwisko: Malwina Cieśla
Wizualizacja Danych		

Cel ćwiczenia:

Zapoznanie z zagadnieniami parsowania plików obj.

Przebieg ćwiczenia:

W celu realizacji ćwiczenia należało w programie Blender stworzyć figurę, a następnie wyexportować do pliku .obj pamiętając o wcześniejszym pogrupowaniu obiektów oraz o kliknięciu triangulacji podczas exportowania. Następnie po skompilowaniu kodu i uruchomieniu programu otrzymałam figurę 3D oteksturowaną obrazkiem używanym przeze mnie we wcześniejszych ćwiczeniach („met.bmp"). Dodatkowo wykorzystałam kody z poprzednich zajęć do utworzenia zmiany koloru figury:



Ilustracja 1: Uzyskana figura

Ilustracja 2: Przyciemniony obraz

Wnioski:

Program do budowania obiektów 3D taki jak używany przeze mnie w tym zadaniu Blender jest bardzo pomocnym narzędziem do pracy z grafiką komputerową. Jest to prostszy oraz szybszy sposób na zbudowanie wymaganej figury, która po wczytaniu do programu OpenGl działa dokładnie tak samo jak opisanie figury od początku w programie OpenGl. Dodatkowo użyte w tym zadaniu oświetlenie 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:

```
#pragma warning(disable:4996)
#include <iostream>
#include <windows.h>
#include <fstream>
#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 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 col;
	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 || col==0)
		outColor = texture(texture1, TexCoord);
		if(col==1)
		outColor = vec4(1.0,0.0,0.0,0.0)*texture(texture1, TexCoord);
		else if(col==2)
		outColor = vec4(0.0,1.0,0.0,0.0)*texture(texture1, TexCoord);
		else if(col==3)
		outColor = vec4(0.0,0.0,1.0,0.0)*texture(texture1, TexCoord);
		outColor = vec4(Color, 1.0);
		}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 150 core
in vec3 Color;
out vec4 outColor;

void main(){
outColor = vec4(Color, 1.0);
}
)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;
float col = 0;
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));
}

bool loadOBJ(const char* path, std::vector<GLfloat>& vertices, float red, float green, float blue) {
	std::vector< unsigned int > vertexIndices, uvIndices, normalIndices;
	std::vector< glm::vec3 > temp_vertices;
	std::vector< glm::vec2 > temp_uvs;
	std::vector< glm::vec3 > temp_normals;

	std::vector < glm::vec3 > out_vertices;
	std::vector < glm::vec2 > out_uvs;
	std::vector < glm::vec3 > out_normals;
```

```
	FILE* file = fopen(path, "r");
	if (file == NULL) {
		printf("Nie można otworzyć pliku !\n");
		return false;
	}
	while (true) {
		char lineHeader[128];
		int isEOF = fscanf(file, "%s", lineHeader);
		if (isEOF == EOF)
			break;
		if (strcmp(lineHeader, "v") == 0) {
			glm::vec3 vertex;
			fscanf(file, "%f %f %f\n", &vertex.x, &vertex.y, &vertex.z);
			temp_vertices.push_back(vertex);
		}
		else if (strcmp(lineHeader, "vt") == 0) {
			glm::vec2 uv;
			fscanf(file, "%f %f\n", &uv.x, &uv.y);
			temp_uvs.push_back(uv);
		}
		else if (strcmp(lineHeader, "vn") == 0) {
			glm::vec3 normal;
			fscanf(file, "%f %f %f\n", &normal.x, &normal.y, &normal.z);
			temp_normals.push_back(normal);
		}
		else if (strcmp(lineHeader, "f") == 0) {
			std::string vertex1, vertex2, vertex3;
			unsigned int vertexIndex[3], uvIndex[3], normalIndex[3];
			fscanf(file, "%d/%d/%d %d/%d/%d %d/%d/%d\n", &vertexIndex[0],
&uvIndex[0], &normalIndex[0], &vertexIndex[1], &uvIndex[1], &normalIndex[1],
&vertexIndex[2], &uvIndex[2], &normalIndex[2]);

			vertexIndices.push_back(vertexIndex[0]);
			vertexIndices.push_back(vertexIndex[1]);
			vertexIndices.push_back(vertexIndex[2]);
			uvIndices.push_back(uvIndex[0]);
			uvIndices.push_back(uvIndex[1]);
			uvIndices.push_back(uvIndex[2]);
			normalIndices.push_back(normalIndex[0]);
			normalIndices.push_back(normalIndex[1]);
			normalIndices.push_back(normalIndex[2]);
		}
	}
	for (unsigned int i = 0; i < vertexIndices.size(); i++) {
		unsigned int vertexIndex = vertexIndices[i];
		glm::vec3 vertex = temp_vertices[vertexIndex - 1];
		out_vertices.push_back(vertex);
	}
	for (unsigned int i = 0; i < uvIndices.size(); i++) {
		unsigned int uvIndex = uvIndices[i];
		glm::vec2 uv = temp_uvs[uvIndex - 1];
		out_uvs.push_back(uv);
```

```
	}
	for (unsigned int i = 0; i < normalIndices.size(); i++) {
		unsigned int normalIndex = normalIndices[i];
		glm::vec3 normal = temp_normals[normalIndex - 1];
		out_normals.push_back(normal);
	}
	for (int i = 0; i < out_vertices.size(); i++) {
		vertices.push_back(out_vertices[i].x);
		vertices.push_back(out_vertices[i].y);
		vertices.push_back(out_vertices[i].z);
	}
}
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);

	std::vector<GLfloat> vertices;
	bool res2 = loadOBJ("star.obj", vertices, 0.0, 0.0, 1.0);
	glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

```
	glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(GLfloat), &vertices[0],
GL_STATIC_DRAW);

	glBindBuffer(GL_ARRAY_BUFFER, vbo);
	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, 6 * sizeof(GLfloat), 0);

	GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
	glEnableVertexAttribArray(colAttrib);
	glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 6 * 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 uniCol = glGetUniformLocation(shaderProgram, "col");
	glUniform1f(uniCol, col);

	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:
	col = 1;
	glUniform1f(uniCol, col);
	std::cout << "Czerwony" << std::endl; break;
case::sf::Keyboard::Num2:
	col = 2;
	glUniform1f(uniCol, col);
	std::cout << "Zielony" << std::endl;  break;
case::sf::Keyboard::Num3:
	col = 3;
	glUniform1f(uniCol, col);
	std::cout << "Niebieski" << 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 << "TRAINGLES 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:
	glEnable(GL_STENCIL_TEST);
	std::cout << "Fragment" << 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, vertices.size());
                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;
}
```