Grupa: IO gr.1	Ćwiczenie: Lab 5	Imię Nazwisko: Malwina Cieśla
Wizualizacja Danych		

Cel ćwiczenia:

Zapoznanie z programowaniem grafiki przy użyciu shader'ów, zastosowanie swobodnego poruszania kamery oraz kontroli szybkości działania pętli głównej.

Przebieg ćwiczenia:

Pierwszą rzeczą, którą powinno się wykonać jest przechwycenie kursora myszy przez okno gry oraz jego ukrycie. W bibliotece SFML można to wykonać poleceniami:

```
// Okno renderingu
sf::Window window(sf::VideoMode(800, 6
window.setMouseCursorGrabbed(true);
window.setMouseCursorVisible(false);
```

Ilustracja 1: Przechwycenie myszy

W celu skanowania ruchu myszy należy wykorzystać zdarzenie MouseMoved:

```
case sf::Event::MouseMoved:
    ustawKamereMysz(uniView, time.asMicroseconds(), window);
    break;
}
```

Ilustracja 2: Zdarzenie MouseMoved

W celu kontroli aktualnego nachylenia kamery potrzebne są dwa kąty:

```
double yaw = -90; //obrót względem osi Y
double pitch=0; //obrót względem osi X
```

Ilustracja 3: kontrolki obrotów

Następnie w funkcji UstawKamereMysz() należało wyznaczyć zmianę pozycji myszy względem ostatniej klatki, uaktualnić wartości Yaw i Pitch dla kamery, nałożyć ograniczenia co do ruchu kamery oraz wyznaczyć nowy wektor kierunku. W tym celu na początku należało go wyznaczyć. Poniżej przedstawiam opisane czynności w kodzie:

```
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;
```

Ilustracja 4: Fragment funkcji

```
if (pitch > 89.0f) {
    pitch = 89.0f;
}
if (pitch < -89.0f) {
    pitch = -89.0f;
}
```

Ilustracja 5: Fragment

```
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);
```

Ilustracja 6: Fragment funkcji

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

Ilustracja 7: Fragment funkcji

Aby zapewnić stałą szybkość wykonywania pętli programu konieczne jest obliczenie współczynnika, mnożonego przez wektory ruchu obiektów (również kamery). Potrzebny jest czas wykonywania pętli głównej. Biblioteka SFML udostępnia odpowiednie funkcje. Trzeba dodać plik nagłówkowy, a przed pętlą główną utworzyć obiekty Clock i Time oraz w pętli głównej programu pobieramy czas wykonywania pętli:

```
sf::Clock clock;
sf::Time time;
bool running = true;
while (running) {
    time = clock.getElapsedTime();
    clock.restart();
```

Ilustracja 8: obiekty Clock i Time

Należało również uniezależnić szybkość działania programu od szybkości komputera przy użyciu:

```
window.setFramerateLimit(20);
```

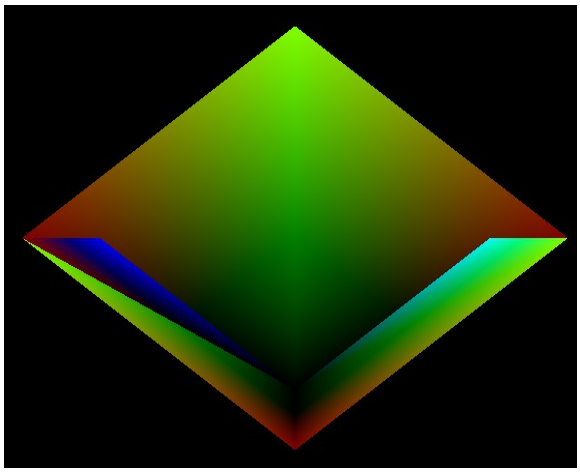
Ilustracja 9: Uniezależnienie

Dodatkowo obsługa klawiszy również się zmieni, klawisze strzałek lewo i prawo nie będą już służyły do obrotu kamery, ale do ruchu prostopadłego do wektora widoku tzw. strefę. Aby to uzyskać należało określić wektor prostopadły do wektora widoku i pionu kamery. Można wykorzystać iloczyny wektorowe, które w kodzie przedstawiają się następująco:

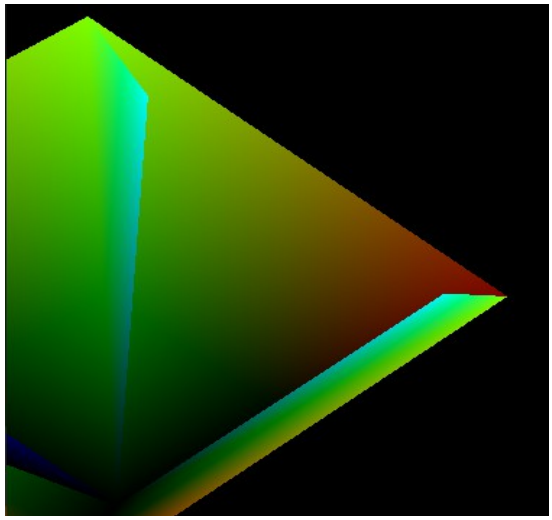
```
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)) {
    cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) {
    cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

Ilustracja 10: Iloczyny wektorowe

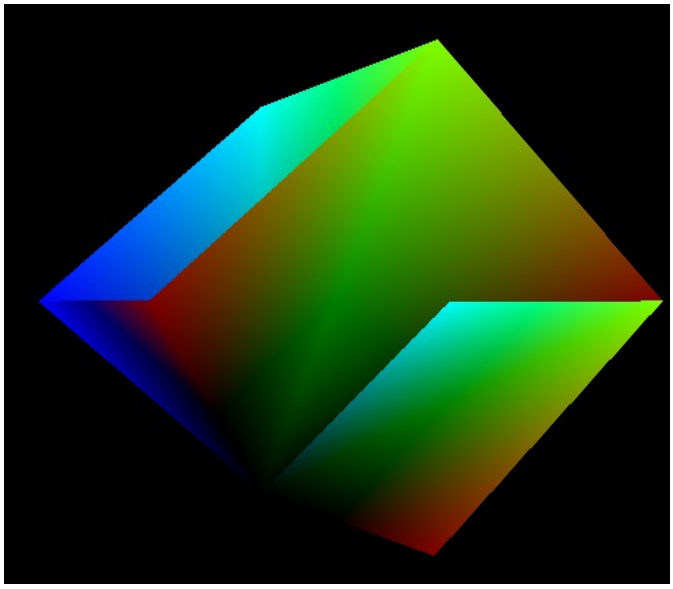
Dzięki temu po uruchomieniu kodu uzyskałam poniższe obrazy figury:



Ilustracja 12: Widok początkowy



Ilustracja 11: Przybliżenie przesuniecie i ruch myszy



Ilustracja 13: Obraz po wprowadzonych zmianach

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>
using namespace std;


//ruch ksmery
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);

// Kody shaderów
const GLchar* vertexSource = R"glsl(
#version 150 core
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);
}
)glsl";

const GLchar* fragmentSource = R"glsl(
#version 150 core
```

```
in vec3 Color;
out vec4 outColor;
void main(){
outColor = vec4(Color, 1.0);
}
)glsl";

bool firstMouse = true;
int lastX, lastY;
double yaw = -90; //obrót względem osi Y
double pitch=0; //obrót względem osi X
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;
	// Okno renderingu
	sf::Window window(sf::VideoMode(800, 600, 32),"OpenGL", sf::Style::Titlebar |
sf::Style::Close, settings);
	window.setMouseCursorGrabbed(true);
	window.setMouseCursorVisible(false);
	// Inicjalizacja GLEW
	glewExperimental = GL_TRUE;
	glewInit();
	window.setFramerateLimit(20);

	// Utworzenie VAO (Vertex Array Object)
	GLuint vao;
	glGenVertexArrays(1, &vao);
	glBindVertexArray(vao);

	// Utworzenie VBO (Vertex Buffer Object)
	// i skopiowanie do niego danych wierzchołkowych
	GLuint vbo;
	glGenBuffers(1, &vbo);

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

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

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

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

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

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

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// Utworzenie i skompilowanie shadera wierzchołków
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);
}

// Utworzenie i skompilowanie shadera fragmentów
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);
```

```
	}

	// Zlinkowanie obu shaderów w jeden wspólny program
	GLuint shaderProgram = glCreateProgram();
	glAttachShader(shaderProgram, vertexShader);
	glAttachShader(shaderProgram, fragmentShader);
	glBindFragDataLocation(shaderProgram, 0, "outColor");
	glLinkProgram(shaderProgram);
	glUseProgram(shaderProgram);

	// Specifikacja formatu danych wierzchołkowych
	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)));

	//macierz modelu wraz z wysłaniem do shader'a
	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);

	//macierz widoku przy pomocy lookAt
	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);

	// wysłanie macierzy do karty graficznej
	GLint uniView = glGetUniformLocation(shaderProgram, "view");
	glUniformMatrix4fv(uniView, 1, GL_FALSE, glm::value_ptr(view));

	//macierz projekcji
	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));
```

```
	// Rozpoczęcie pętli zdarzeń
	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: //zamknięcie okna na klawisz esc
			window.close();
			break;
		}

		ustawKamereKlawisz(uniView, time.asMicroseconds());
		// Nadanie scenie koloru czarnego
		glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
		glClear(GL_COLOR_BUFFER_BIT);

		// Narysowanie kolorowego koła
		glDrawArrays(GL_POLYGON, 0, 36);
		// Wymiana buforów tylni/przedni
		window.display();

	}
	// Kasowanie programu i czyszczenie buforów
	glDeleteProgram(shaderProgram);
	glDeleteShader(fragmentShader);
	glDeleteShader(vertexShader);
	glDeleteBuffers(1, &vbo);
	glDeleteVertexArrays(1, &vao);
	// Zamknięcie okna renderingu
	window.close();
	return 0;
}
```