POLITECHNIKA WROCŁAWSKA
Instytut Informatyki, Automatyki i Robotyki
Zakład Systemów Komputerowych

Wprowadzenie do grafiki komputerowej

Kurs: INEK00012L

Sprawozdanie z ćwiczenia nr 7

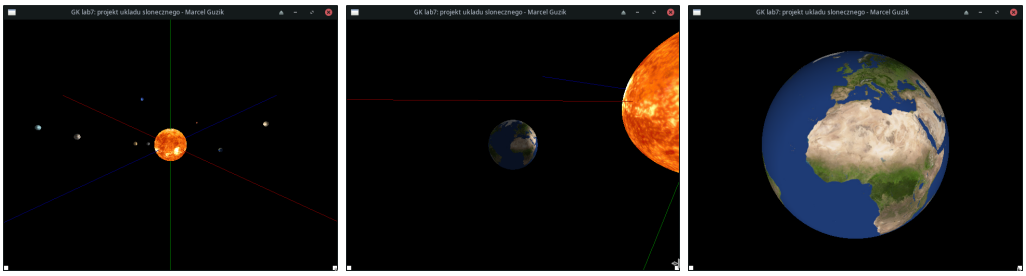
TEMAT ĆWICZENIA: OpenGL – projekt

Wykonał:	**Marcel Guzik**
Termin:	**PN/P 7:30-10:00**
Data wykonania ćwiczenia:	**17-01-2022**
Data oddania sprawozdania:	**24-01-2022**
Ocena:	

Uwagi prowadzącego:

1 Wprowadzenie

Celem ćwiczenia jest wykonanie projektu inkorporującego wszystkie techniki OpenGL poznane podczas kursu, wykonując interaktywny model systemu słonecznego.



Rysunek 1: Kopie widoku okna programu przedstawiające słońce, planety układu słonecznego oraz zbliżenie na Ziemię

2 Zakres projektu

Zaimplementowano:

- Realistyczny system oświetlenia przez oświetlanie i cieniowanie Phonga
- Intuicyjne sterowanie kamerą
- Poruszanie planetami wg. ich orbit dookoła słońca, które dla uproszczenia w programie są idealnymi okręgami
- Kontrolę czasu by przyśpieszać lub spowalniać obieg planet.

Nie zaimplementowano:

Symulowane są tylko główne planety systemu słonecznego, bez ich naturalnych satelitów. Skala wielkości planet oraz odległości między nimi nie są realistyczne i zostały wybrane arbitralnie by zagwarantować poglądowość modelu. Model symuluje rotację planet wokół słońca, wokół własnej osi Y, jednak nie mają inklinacji orbity (wszystkie poruszają się na tej samej powierzchni) ani pochylenia osiowego, rotacja wokół własnej osi wszystkich planet jest w tym samym kierunku.

3 Kod programu

```
#include <cstdio>
#include <cmath>
#include <vector>
#include <iostream>
#include <chrono>
#include <cstdint>
#include <GL/gl.h>
#include <GL/glut.h>


#include "texture.cpp"
```

```
const float EPSILON = 0.000001;

typedef float point3[3];
typedef float angles[3];

float angle = 90.0f;

const double DAYS_PER_YEAR = 365.0;
const double SECONDS_IN_MINUTE = 60.0;

// simulated days per real-time second
double timeScale = DAYS_PER_YEAR / SECONDS_IN_MINUTE;
double simCurrentDay = 0.0;

struct Planet {
    float distance;
    int orbPeriodDays;
    float color[4];
    float size;
    GLbyte* texture;
    float rotationRate;
    point3 position;
    float rotationY;
};

float lightPos[4] = {0.0, 0.0, 0.0, 1.0};

int focusedBodyIndex = 0;

std::vector<Planet> planets;

float zstep = 0.01;

// stan klawiszy myszy
// 0 - nie naciśnięto żadnego klawisza
// 1 - naciśnięty został lewy klawisz
// 2 - naciśnięty został prawy klawisz
static GLint status = 0;

// poprzednia pozycja kursora myszy
static int x_pos_old = 0;
static int y_pos_old = 0;

// różnica pomiędzy pozycją bieżącą i poprzednią kursora myszy
static int delta_x = 0;
static int delta_y = 0;
angles viewerAngles = {0.0, 0.5 * M_PI, 2.0};
```

```
int width, height, components;
GLenum format;
GLbyte* sunTex;
GLbyte* merTex;
GLbyte* venTex;
GLbyte* earTex;

GLuint textures[9];

std::chrono::_V2::steady_clock::time_point lastTime;

float clamp(float d, float min, float max) {
    const float t = d < min ? min : d;
    return t > max ? max : t;
}

float fmodfp(float a, float b) {
    if(a < 0) {
        a += b;
    } else if(a > b) {
        a -= b;
    }
    return a;
}

// calculates a planet position from its distance from the sun, orbital period,
// and current time
void positionFromTime(float distance, float orbPeriod, point3 position) {
    double planetOrbPos = (simCurrentDay / orbPeriod);

    float x = distance * sin(planetOrbPos * M_PI);
    float z = distance * cos(planetOrbPos * M_PI);

    position[0] = x;
    position[2] = z;
}

void rotationFromTime(Planet& planet) {
    if(planet.rotationRate == 0.0) return;
    planet.rotationY = fmodf((simCurrentDay / planet.rotationRate) * 360.0,
↪   360.0);
}

void angles_to_coords(float* angles, point3 coords) {
    coords[0] = angles[2] * cos(angles[1]) * cos(angles[0]);
    coords[1] = angles[2] * sin(angles[0]);
    coords[2] = angles[2] * sin(angles[1]) * cos(angles[0]);
}
```

```
// Funkcja "bada" stan myszy i ustawia wartości odpowiednich zmiennych
↪  globalnych
void mousePressed(int btn, int state, int x, int y) {
    // przypisanie aktualnie odczytanej pozycji kursora jako pozycji
↪  poprzedniej
    x_pos_old = x;
    y_pos_old = y;

    if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
        status = 1;          // wcięnięty został lewy klawisz myszy
    }
    else if(btn == GLUT_RIGHT_BUTTON && state == GLUT_DOWN) {
        status = 2;
    }
    else {
        status = 0;          // nie został wcięnięty żaden klawisz
    }
}

void mouseMoved(GLsizei x, GLsizei y) {
    // obliczenie różnicy położenia kursora myszy i podstawienie bieżącego
    // położenia jako poprzednie
    delta_x = x - x_pos_old;
    x_pos_old = x;

    delta_y = y - y_pos_old;
    y_pos_old = y;

    if(status == 1) {
        float new_elevation, new_azimuth;
        new_elevation = viewerAngles[0] + delta_y * 0.02;
        new_azimuth = viewerAngles[1] + delta_x * 0.02;

        viewerAngles[0] = fmodfp(new_elevation, 2 * M_PI);
        viewerAngles[1] = fmodfp(new_azimuth, 2 * M_PI);
        if(0.5 * M_PI < viewerAngles[0] && viewerAngles[0] < 1.0 * M_PI) {
            viewerAngles[0] = 0.5 * M_PI - EPSILON;
        } else if(1.0 * M_PI < viewerAngles[0] && viewerAngles[0] < 1.5 * M_PI)
↪  {
            viewerAngles[0] = 1.5 * M_PI + EPSILON;
        }
    }
    else if(status == 2) {
        viewerAngles[2] = clamp(viewerAngles[2] * (1.0 + delta_y * zstep), 0.01,
↪  100.0);
    }
}

// Funkcja rysująca osie układu współrzędnych
```

```
void axes() {
    point3 x_min = {-10.0, 0.0, 0.0};
    point3 x_max = { 10.0, 0.0, 0.0};
    // początek i koniec obrazu osi x

    point3 y_min = {0.0, -10.0, 0.0};
    point3 y_max = {0.0,  10.0, 0.0};
    // początek i koniec obrazu osi y

    point3 z_min = {0.0, 0.0, -10.0};
    point3 z_max = {0.0, 0.0,  10.0};
    //  początek i koniec obrazu osi y

    glBegin(GL_LINES);
        glColor4f(0.5f, 0.0f, 0.0f, 0.1f);  // kolor rysowania osi - czerwony
        glVertex3fv(x_min);
        glVertex3fv(x_max);

        glColor4f(0.0f, 0.5f, 0.0f, 0.1f);  // kolor rysowania - zielony
        glVertex3fv(y_min);
        glVertex3fv(y_max);

        glColor4f(0.0f, 0.0f, 0.5f, 0.1f);  // kolor rysowania - niebieski
        glVertex3fv(z_min);
        glVertex3fv(z_max);
    glEnd();
}

void glTranslatefv(float arr[]) {
    return glTranslatef(arr[0], arr[1], arr[2]);
}

// a gluLookAt wrapper that takes arguments as structs
void lookAtp3(point3 viewer, point3 center, point3 up) {
    gluLookAt(viewer[0], viewer[1], viewer[2], center[0], center[1], center[2],
↪   up[0], up[1], up[2]);
}

void renderScene() {
    // update simulation timestep
    auto currentTime = std::chrono::steady_clock::now();
    uint64_t timeElapsedUs =
↪   std::chrono::duration_cast<std::chrono::microseconds>(currentTime -
↪   lastTime).count();
    lastTime = currentTime;

    double timeElapsedS = (double)timeElapsedUs / 1000000.0;
    double simDaysElapsed = timeScale * timeElapsedS;
    simCurrentDay += simDaysElapsed;
```

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDisable(GL_LIGHTING);

    for(auto& planet: planets) {
        positionFromTime(planet.distance, planet.orbPeriodDays,
↪   planet.position);
        rotationFromTime(planet);
    }

    float* focusedPlanet = planets[focusedBodyIndex].position;

    float viewerPos[4] = {0.0, 0.0, 0.0, 1.0};
    point3 up = {0.0, 1.0, 0.0};

    angles_to_coords(viewerAngles, viewerPos);
    viewerPos[0] += focusedPlanet[0];
    viewerPos[1] += focusedPlanet[1];
    viewerPos[2] += focusedPlanet[2];

    glLoadIdentity();
    lookAtp3(viewerPos, focusedPlanet, up);

    glDisable(GL_TEXTURE_2D);
    axes();
    glEnable(GL_TEXTURE_2D);

    for(int i = 0; i < planets.size(); ++i) {
        auto& planet = planets.at(i);
        GLUquadric* quadric = gluNewQuadric();

        glEnable(GL_LIGHTING);
        glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
        glLoadIdentity();
        lookAtp3(viewerPos, focusedPlanet, up);
        glTranslatefv(planet.position);
        glRotatef(planet.rotationY, 0.0f, 1.0f, 0.0f);

        if(planet.distance == 0.0) {
            glDisable(GL_LIGHTING);
            glColor4fv(planet.color);
        }

        if(planet.texture) {
            glEnable(GL_TEXTURE_2D);
            glBindTexture(GL_TEXTURE_2D, textures[i]);
            glRotatef(angle, 1.0f, 0.0f, 0.0f);
```

```
            gluQuadricTexture(quadric, true);
        }

        gluSphere(quadric, planet.size, 100, 100);
        gluQuadricTexture(quadric, false);
    }


    glFlush();
    glutSwapBuffers();

    glutPostRedisplay();
}

void changeSize(int horizontal, int vertical) {
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if(vertical == 0) return;
    glViewport(0, 0, horizontal, vertical);

    // Ustawienie parametrów dla rzutu perspektywicznego:
    // 70 stopni pola widzenia w osi y
    // proporcje między osią x i y, do uzyskania pola widzenia w osi x
    // minimalna odległość od obserwatora
    // maksymalna odległość od obserwatora
    float aspectRatio = (float)horizontal / (float)vertical;
    gluPerspective(70, aspectRatio, 0.01, 100000.0);
}

void keyPressed(unsigned char key, int x, int y) {
    switch(key) {
    case 'p':
        timeScale *= 2.0;
        break;
    case 'l':
        timeScale /= 2.0;
        break;

    case 'x':
        focusedBodyIndex = (focusedBodyIndex + 1) % planets.size();
        break;
    case 'z':
        focusedBodyIndex -= 1;
        if(focusedBodyIndex < 0) focusedBodyIndex = planets.size() - 1;
        break;
    }
    glutPostRedisplay();
}
```

```
void init() {
    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);

    // współczynniki ka =[kar,kag,kab] dla światła otoczenia
    float mat_ambient[]  = {1.0, 1.0, 1.0, 1.0};

    // współczynniki kd =[kdr,kdg,kdb] światła rozproszonego
    float mat_diffuse[]  = {1.0, 1.0, 1.0, 1.0};

    // współczynniki ks =[ksr,ksg,ksb] dla światła odbitego
    float mat_specular[] = {1.0, 1.0, 1.0, 1.0};

    // współczynnik n opisujący połysk powierzchni
    float mat_shininess  = {20.0};


    float light_ambient[] = {0.1, 0.1, 0.1, 1.0};
    // składowe intensywności świecenia źródła światła otoczenia
    // Ia = [Iar,Iag,Iab]

    float light_diffuse[] = {1.0, 1.0, 1.0, 1.0};
    // składowe intensywności świecenia źródła światła powodującego
    // odbicie dyfuzyjne Id = [Idr,Idg,Idb]

    float light_specular[]= {1.0, 1.0, 1.0, 1.0};
    // składowe intensywności świecenia źródła światła powodującego
    // odbicie kierunkowe Is = [Isr,Isg,Isb]

    float att_constant  = {1.0};
    // składowa stała ds dla modelu zmian oświetlenia w funkcji
    // odległości od źródła

    float att_linear    = {0.05};
    // składowa liniowa dl dla modelu zmian oświetlenia w funkcji
    // odległości od źródła

    float att_quadratic  = {0.001};
    // składowa kwadratowa dq dla modelu zmian oświetlenia w funkcji
    // odległości od źródła

    // Ustawienie patrametrów materiału
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);

    // Ustawienie parametrów źródła
```

```
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);

    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, att_constant);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, att_linear);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, att_quadratic);

    glEnable(GL_LIGHT0);

    planets = {
        // sun
        {
            0.0,
            1,
            {1.0, 1.0, 1.0, 1.0},
            0.5,
            LoadTGAImage("textures/sun_f.tga", &width, &height, &components,
↪   &format),
            0.0
        },

        // mercury
        {
            0.7,
            115,
            {0.2, 0.2, 0.2, 1.0},
            0.048,
            LoadTGAImage("textures/mercury_f.tga", &width, &height, &components,
↪   &format),
            176.0
        },

        // venus
        {
            1.1,
            224,
            {0.8, 0.8, 0.8, 1.0},
            0.060,
            LoadTGAImage("textures/venus_f.tga", &width, &height, &components,
↪   &format),
            116.0
        },

        // earth
        {
            1.5,
```

```
            365,
            {0.0, 0.0, 1.0, 1.0},
            0.064,
            LoadTGAImage("textures/earth_f.tga", &width, &height, &components,
↪   &format),
            1.0
        },

        // mars
        {
            2.2,
            779,
            {0.6, 0.2, 0.0, 1.0},
            0.034,
            LoadTGAImage("textures/mars_f.tga", &width, &height, &components,
↪   &format),
            1.025
        },

        // jupiter
        {
            3.2,
            4332,
            {1.0, 1.0, 1.0, 1.0},
            0.1,
            LoadTGAImage("textures/jupiter_f.tga", &width, &height, &components,
↪   &format),
            0.41
        },

        // saturn
        {
            4.2,
            10759,
            {1.0, 1.0, 1.0, 1.0},
            0.1,
            LoadTGAImage("textures/saturn_f.tga", &width, &height, &components,
↪   &format),
            0.43
        },

        // uranus
        {
            5.2,
            30688,
            {1.0, 1.0, 1.0, 1.0},
            0.1,
            LoadTGAImage("textures/uranus_f.tga", &width, &height, &components,
↪   &format),
```

```
            0.718
        },

        // neptune
        {
            6.2,
            60195,
            {1.0, 1.0, 1.0, 1.0},
            0.1,
            LoadTGAImage("textures/neptune_f.tga", &width, &height, &components,
↪   &format),
            0.671
        },

    };

    glGenTextures(9, textures);

    // gen textures
    for(int i = 0; i < planets.size(); ++i) {
        glBindTexture(GL_TEXTURE_2D, textures[i]);
        glTexImage2D(GL_TEXTURE_2D, 0, components, width, height, 0, format,
↪   GL_UNSIGNED_BYTE, planets[i].texture);
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    }
}

int main(int argc, char** argv) {
    std::string title = "GK lab7: projekt ukladu slonecznego - Marcel Guzik";

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |GLUT_DEPTH);
    glutInitWindowSize(640, 480);
    glutCreateWindow(title.c_str());

    std::cout << title << std::endl;

    std::cout << "Klawisze:" << std::endl;
    std::cout << "\tz, x: wycentruj kamerę na roznych cialach niebieskich" <<
↪   std::endl;
    std::cout << "\tp, l: przyspieszaj i zwalniaj czas (startowa predkosc to
↪   365 dni symulacji na minute)" << std::endl;

    std::cout << "Mysz:" << std::endl;
    std::cout << "\tLPM: obraca kamere dookola aktualnie wycentrowanego
↪   obiektu" << std::endl;
```

```
    std::cout << "\tPPM: zmiana odleglosc kamery od aktualnie wycentrowanego
↪   obiektu" << std::endl;

    init();

    glutDisplayFunc(renderScene);
    glutReshapeFunc(changeSize);
    glutMouseFunc(mousePressed);
    glutMotionFunc(mouseMoved);
    glutKeyboardFunc(keyPressed);

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    lastTime = std::chrono::steady_clock::now();
    glutMainLoop();

    return 0;
}
```

4 Podsumowanie

Wykonany program skupił w sobie techniki poznane w trakcie kursu Grafika Komputerowa. Wykonany model systemu słonecznego zawiera jednak wiele uproszczeń.