08.04.2021

Grafika komputerowa

Laboratorium numer 3

„Budowa obiektu sterowanego”

Brajan Miśkowicz
Grupa L4, 163976

Podczas laboratorium należało utworzyć obiekt przypominający łazik marsjański zbudowany na prymitywach bazujących na trójkącie. Zostało utworzone 27 elementów składowych dzięki bibliotece Glut.h umożliwiającej pisanie w języku C++. Sam projekt został wykonany obiektowo dzięki klasie Lazik. Poniżej przedstawiono kod z komentarzami tworzący okno, oraz pozwalający na używanie w nim elementów OpenGL:

```
#include <stdlib.h>
#include "includes/glut.h"
#include "includes/GL.H"
#include <cmath>
// angle of rotation for the camera
direction float angle = 0.0;
// actual vector representing the camera's direction
float lx = 0.0f, lz = -1.0f, ly = 0.0f;
// XZ position of the camera
float x = 120.0f, z = 100.0f, y = 80.0f;
#define GL_PI 3.14       //PI number for round objects

void changeSize(int w, int h)
{
        // Prevent a divide by zero, when window is too short
        // (you cant make a window of zero width).
        if (h == 0)
                h = 1;
        float ratio = w * 1.0 / h;
        // Use the Projection Matrix
        glMatrixMode(GL_PROJECTION);
        // Reset Matrix
        glLoadIdentity();
        // Set the viewport to be the entire
        window glViewport(0, 0, w, h);
        // Set the correct perspective.
        gluPerspective(90.0f, ratio, 0.1f, 1000.0f);
        // Get Back to the Modelview
        glMatrixMode(GL_MODELVIEW);
}
void processSpecialKeys(int key, int xx, int yy) {

        switch (key) {
        case GLUT_KEY_LEFT:       //turn left
                angle -= 0.1f;
                lx = sin(angle);
                lz = -cos(angle);
                break;
        case GLUT_KEY_RIGHT:      //turn right
                angle += 0.1f;
                lx = sin(angle);
                lz = -cos(angle);
                break;
        case GLUT_KEY_UP:         //move forward
                y += ly;
                x += lx;
                z += lz;
                break;
        case GLUT_KEY_DOWN:       //move backward
                y -= ly;
                x -= lx;
                z -= lz;
                break;
        case GLUT_KEY_F1:         //rotation up
                ly +=0.1f;
                break;
        case GLUT_KEY_F2:         //rotation down
                ly -= 0.1f;
                break;
        }
```

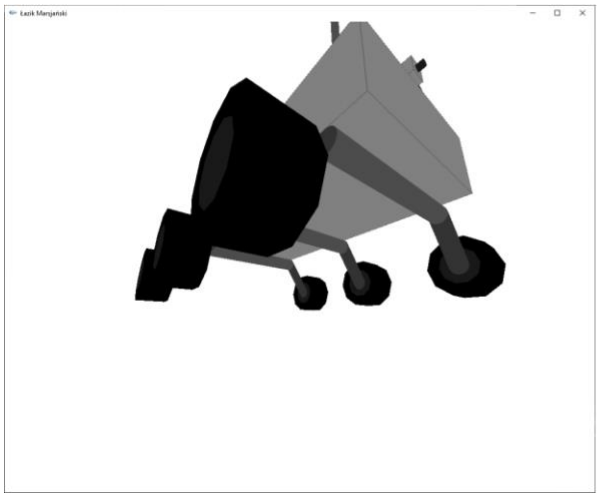
```
}

void renderScene(void)
{
        // Clear Color and Depth Buffers
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glPolygonMode(GL_BACK, GL_FILL);
        // Reset transformations
        glLoadIdentity();
        // Set the camera
        gluLookAt(x, y, z, x + lx, y + ly, z + lz, 0.0f, 1.0f, 0.0f);
        //at this point in the code there are implemented the OpenGL
        objects. glutSwapBuffers();
}
int main(int argc, char** argv) {

        // init GLUT and create window glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE |
        GLUT_RGBA); glutInitWindowPosition(100, 100);
        glutInitWindowSize(1080, 860); glutCreateWindow("Łazik
        Marsjański");

        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);      //black background colour
        // register callbacks
        glutDisplayFunc(renderScene);
        glutReshapeFunc(changeSize);
        glutIdleFunc(renderScene);
        // here are the new entries
        glutSpecialFunc(processSpecialKeys);
        // OpenGL init
        glEnable(GL_DEPTH_TEST);
        // enter GLUT event processing
        cycle glutMainLoop();
        return 0;
}
```

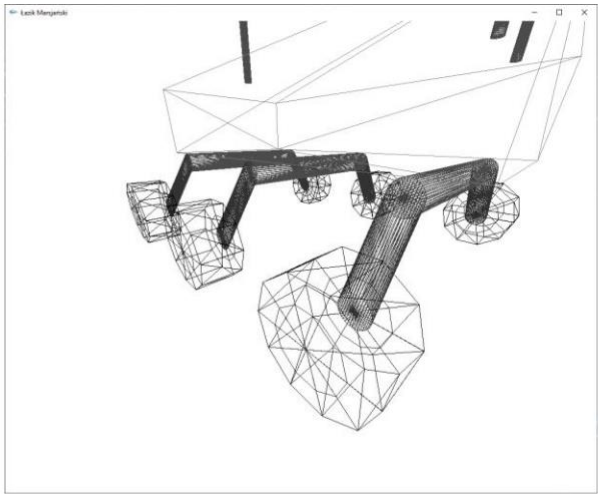
Przed ostatnią linijką funkcji renderScene, w miejscu wskazanym przez komentarz wywoływane jest stworzenie Łazika. Odpowiada za to plik *Lazik.cpp*, oraz plik nagłówkowy *Lazik.h*. Wewnątrz konstruktora klasy Lazik znajduje się wywołanie funkcji rysujących za pomocą OpenGL. Jest 6 takich funkcji, które wyglądają w następujący sposób:

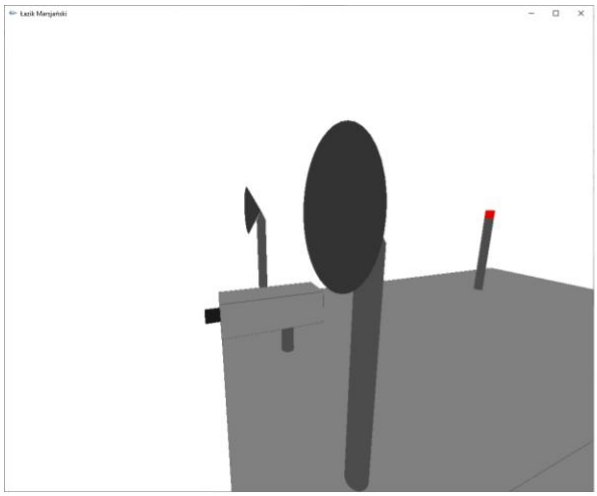
```
void cuboid(float x, float y, float z, float length, float height, float width,
float red, float green, float blue)

void cylinder(float R, float x, float y, float z, float length, float red,
float green, float blue, int base, int mode)

void axle(float R, float x, float y, float z, float length, float height, float
red, float green, float blue, int base)

void wheel(float R, float x, float y, float z, float length, float red, float
green, float blue, int base)

void cone(float R, float x, float y, float z, float height, float red, float
green, float blue, int base)

void antenna(float R, float x, float y, float z, float height, float red, float
green, float blue, int base)
```

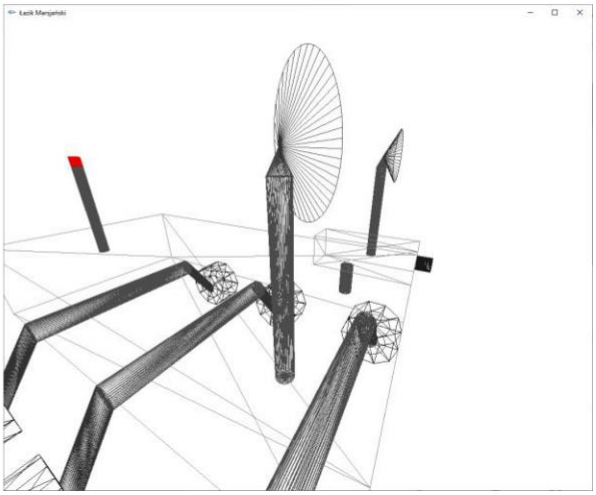
Wszystkie te funkcje pobierają parametery potrzebne do utworzenia odpowiadających im obiektów, gdzie **R** oznacza promień, **x**,**y** oraz **z** - współrzędne, **length** - długość, **height** - wysokość, **width** – szerokość, **base** – ilość części, na które podzielona zostaje liczba Pi przy okrągłych elementach, a **red**, **green** i **blue** wartości kolorów. **Mode** w funkcji *cylinder* to zmienna mówiąca w jakiej pozycji

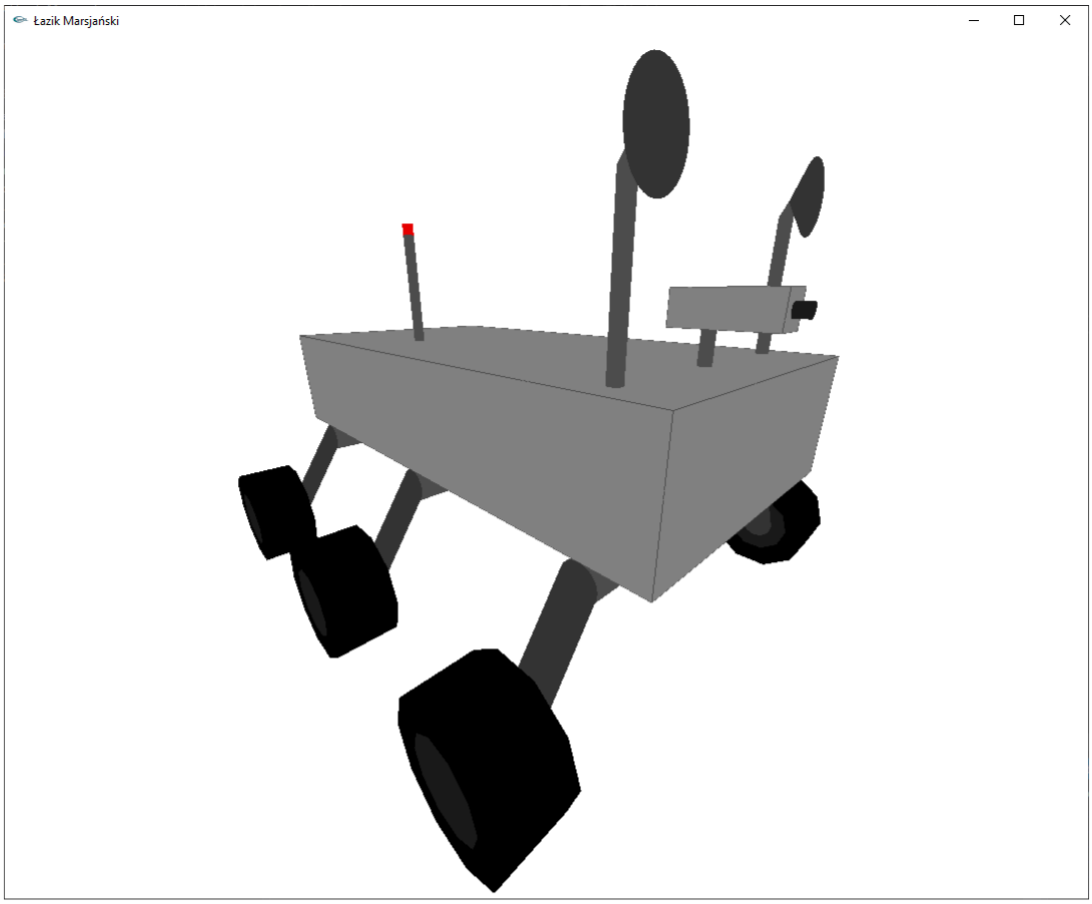
znajdował będzie się walec: dla 0 jego wysokość jest na osi Z, dla mode = 1 na osi Y, a dla mode równego 2 na osi X, kod dla tych trzech opcji różni się właśnie tymi zamienionymi parametrami. Funkcja *cuboid* tworzy prostopadłościan (wykorzystane są tutaj także funkcje `GL_LINE_STRIP` rysujące na krawędziach linie, pozwala to na tworzenie dużo wyraźniejszych prostopadłościanów), *cylinder* – walec (stworzony na podstawie kodu z poprzedniego laboratorium), *axle* walec pochyły zakończony płaskimi podstawami (na obrazku poniżej te pochyłe walce łączą koła oraz osie, tutaj kod jest niemalże identyczny, jak przy zwykłym walcu), *wheel* – koło z „kołpakiem" (walec z dodatkowymi dwoma kółkami), *cone* – stożek z podstawą, a *antenna* stożek bez podstawy przypominający radar. Idąc od dołu łazik posiada 6 kół, do każdego podłączony jest pochyły walec łączący się z osią (walec) i dalej drugim pochyłym walcem i kołem. Nad osiami znajduje się prostopadłościan będący ciałem łazika, na nim składająca się z dwóch walców i prostopadłościanu kamera, dwa radary - zbudowane każdy z walca, stożka i stożka bez podstawy, oraz antenę zbudowaną z szarego walca, oraz znajdującego się na nim niewielkiego, czerwonego walca.

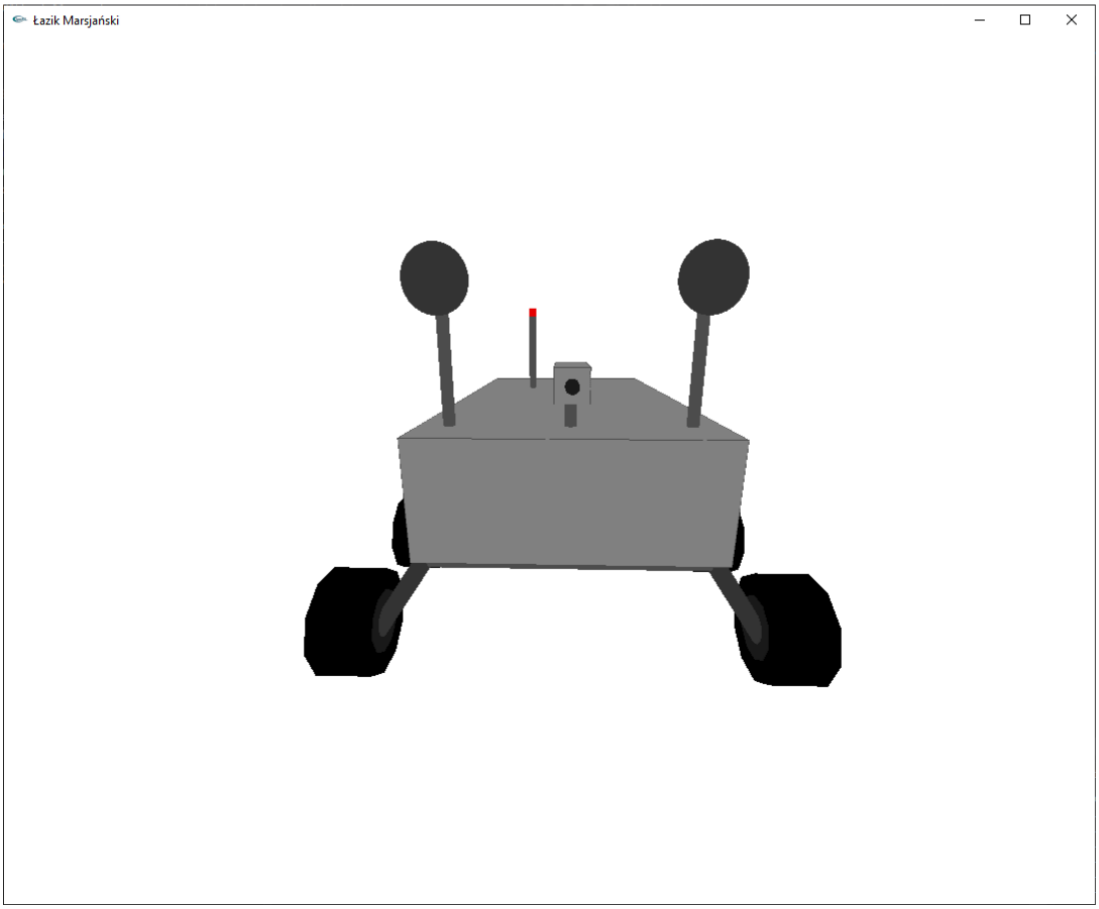









Łazik Marsjański


Łazik Marsjański

22.04.2021

Grafika komputerowa

Laboratorium numer 4

„Budowa otoczenia”

Brajan Miśkowicz
Grupa L4, 163976

Podczas laboratorium zostało utworzone otoczenie dla łazika za pomocą programu Blender. Zostało w nim stworzone nierówne podłoże, oraz dwa obiekty będące przeszkodami. Ich import został umożliwony za pomocą stworzonego interpretera (mieszczącego się w plikach *Interpreter.h* oraz *Interpreter.cpp*). Klasa Interpreter wygląda następująco:

```
class Interpreter
{
public:
        struct vertex
        {
                float x;
                float y;
                float z;
        };
        Interpreter(string filename);//Read function
        void    DrawT();//Drawing    and    texturing
        function void Draw();//Drawing function
private:
        vector<vector<GLfloat>> v;//Store vertex (x,y,z) coordinates
        vector<vector<GLint>> f;//Store the three vertex indexes of the face
};
```

W konstruktorze zczytywane są trójki liczb dla ścianek (tzw. faces) do wektora *f* z linijek rozpoczynających się od liter *f* w pliku i wierzchołków do wektora *v* z linijek *v*.

```
Interpreter::Interpreter(string filename)
{
       ifstream file(filename); //reading the
       file string line;
       while (getline(file, line)) //reading the lines of file
       {
              if (line.substr(0, 1) == "v") //if it is vertice
              {
                     vector<GLfloat> Point;
                     GLfloat x, y, z;
                     istringstream s(line.substr(2)); //substracting the line
                     s >> x; s >> y; s >> z; //reading the numbers
                     Point.push_back(x); //pushing the numbers into point
                     Point.push_back(y);
                     Point.push_back(z);
                     v.push_back(Point); //pushing points into vectors
              }
              if (line.substr(0, 1) == "f") //if it is face
              {
                     vector<GLint> vIndexSets;
                     GLint u, v, w;
                     istringstream vtns(line.substr(2)); //substracting the line
                     vtns >> u; vtns >> v; vtns >> w; //reading the numbers
                     vIndexSets.push_back(u - 1); //pushing the numbers into vectors
                     vIndexSets.push_back(v - 1);
                     vIndexSets.push_back(w - 1);
                     f.push_back(vIndexSets); //pushing points into vectors
              }
       }
       file.close();//closing the file
}
```

Funkcja *Draw()* rysuje trójkąty dzięki wyżej wspomnianym wektorom. Najpierw brane są 3 liczby z wektora *f*, te liczby oznaczają wierzchołki trójkątów, oraz indeksy w wektorze *v*, gdzie zapisane są koordynaty tych wierzchołków:

```
void Interpreter::Draw()
{

	glBegin(GL_TRIANGLES); //Start drawing
	for (int i = 0; i < f.size(); i++)
	{
		//Three vertices
		vertex a, b, c;
		GLint firstVertexIndex = (f[i])[0];//Remove the vertex index
		GLint secondVertexIndex = (f[i])[1];
		GLint thirdVertexIndex = (f[i])[2];

		a.x = (v[firstVertexIndex])[0];//The first
		vertex a.y = (v[firstVertexIndex])[1]; a.z =
		(v[firstVertexIndex])[2];

		b.x = (v[secondVertexIndex])[0]; //The second
		vertex b.y = (v[secondVertexIndex])[1]; b.z =
		(v[secondVertexIndex])[2];

		c.x = (v[thirdVertexIndex])[0]; //The third
		vertex c.y = (v[thirdVertexIndex])[1]; c.z =
		(v[thirdVertexIndex])[2];

		glVertex3f(a.x, a.y, a.z);//Draw triangle
		glVertex3f(b.x, b.y, b.z);
		glVertex3f(c.x, c.y, c.z);
	}
	glEnd();
}
```

Funkcja DrawT jest niemalże identyczna, jednak pozwala na teksturowanie trójkątów:

```
void Interpreter::DrawT()
{
	glEnable(GL_TEXTURE_2D); // turn on texture
	mode glBegin(GL_TRIANGLES);//Start drawing
	for (int i = 0; i < f.size(); i++)
	{
		vertex a, b, c; //Three vertices
		GLint firstVertexIndex = (f[i])[0];//Remove the vertex index
		GLint secondVertexIndex = (f[i])[1];
		GLint thirdVertexIndex = (f[i])[2];
		a.x = (v[firstVertexIndex])[0];//The first
		vertex a.y = (v[firstVertexIndex])[1]; a.z =
		(v[firstVertexIndex])[2];
		b.x = (v[secondVertexIndex])[0]; //The second
		vertex b.y = (v[secondVertexIndex])[1]; b.z =
		(v[secondVertexIndex])[2];
		c.x = (v[thirdVertexIndex])[0]; //The third
		vertex c.y = (v[thirdVertexIndex])[1]; c.z =
		(v[thirdVertexIndex])[2];

		glTexCoord2f(0.0, 0.0);   //texturing
		glVertex3f(a.x, a.y, a.z);//Draw triangle
		glTexCoord2f(1.0, 0.0);
		glVertex3f(b.x, b.y, b.z);
		glTexCoord2f(0.0, 1.0);
		glVertex3f(c.x, c.y, c.z);
	}
	glEnd();
	glDisable(GL_TEXTURE_2D); // turn of texture mode
}
```

Tuż przed stworzeniem wierzchołków wywoływane jest tworzenie tektur dla wierzchołków za pomocą `glTexCoord2f(i, j);`, gdzie i, j oznaczają indeksy wierzchołków na teksturze – 0 i 0 oznaczają dolny lewy róg, 1 i 0 – prawy dolny róg, 0 i 1 lewy górny róg, a 1 i 1 prawy górny róg. Z racji korzystania jedynie z trójkątów są tu wykorzystywane jedynie 3 wierzchołki.

W pliku głównym deklarowane sż zmienne globalne, oraz tworzone poniższe obiekty:

```
#define STB_IMAGE_IMPLEMENTATION
static GLuint textureName;
int width, height, nrChannels;

Interpreter textures = Interpreter("tekstury.obj");
Interpreter cube = Interpreter("cube.obj");
Interpreter icosphere = Interpreter("Icosphere.obj");
```

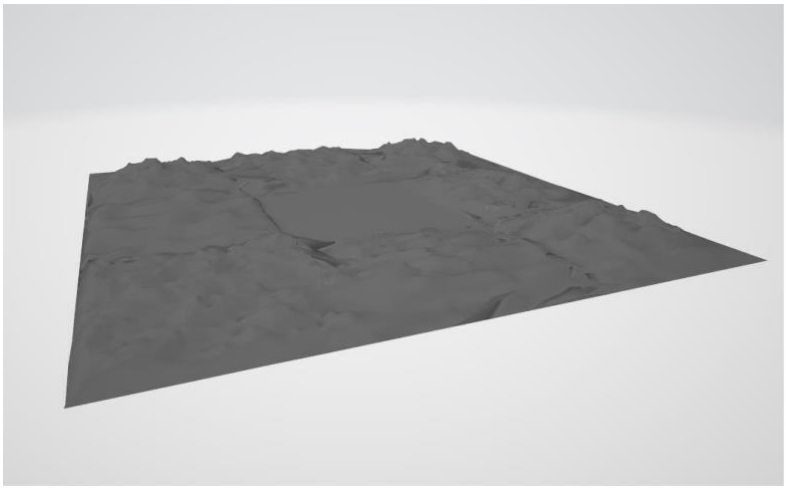
Przy teksturowaniu używana jest biblioteka `stb_image.h`, która pomaga przy imporcie obrazków w formacie jpg. Podobnie jak poprzednio realizacja zadania została wykonana wewnątrz funkcji *renderscene():*

```
init(stbi_load("textures.jpg", &width, &height, &nrChannels, 0)); //loading
image and setting up the texturing
textures.DrawT(); //drawing object and its textures
glDeleteTextures(1, &textureName); //deleting texture

init(stbi_load("stone.jpg", &width, &height, &nrChannels, 0)); //loading image and
setting up the texturing
icosphere.DrawT(); //drawing object and its textures
glDeleteTextures(1, &textureName); //deleting texture

glColor3f(0, 0, 0);
cube.Draw(); //drawing object
```

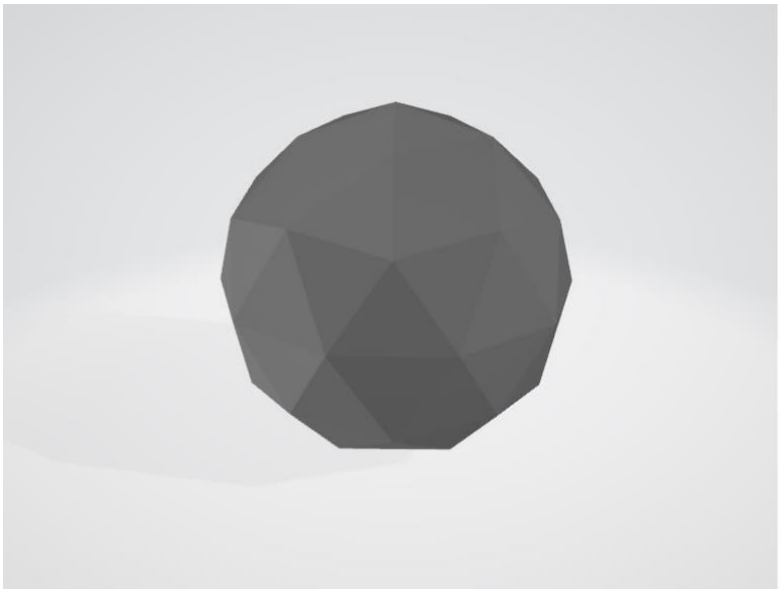
Wywoływana jest tutaj dwukrotnie funkcja *init* pozwalająca ustawić opcje dla teksturowania i rysowane są dwa obiekty teksturowane, oraz jeden nieteksturowany czarny. Funkcja *init* wygląda następująco:

```
void init(unsigned char* data)
{
	glGenTextures(1, &textureName); //creating texture name
	glBindTexture(GL_TEXTURE_2D, textureName); //Binding texture
	//Specifying texture parameters
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
GL_RGB, GL_UNSIGNED_BYTE, data); //setting the image to textures

	glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE); //Determinating
the colour of final pixel
}
```

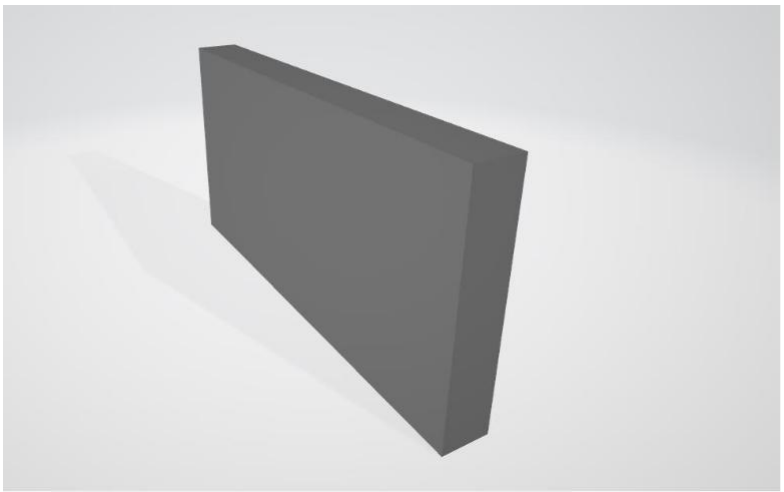
Samo podłoże i jego tekstura wyglądają w następujący sposób:



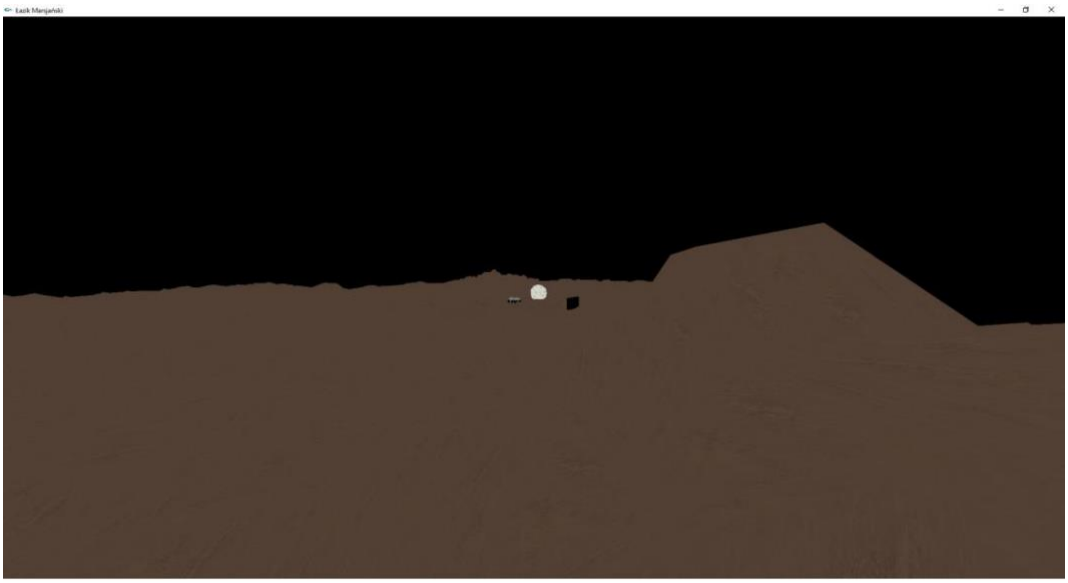
A drugi obiekt teksturowany:

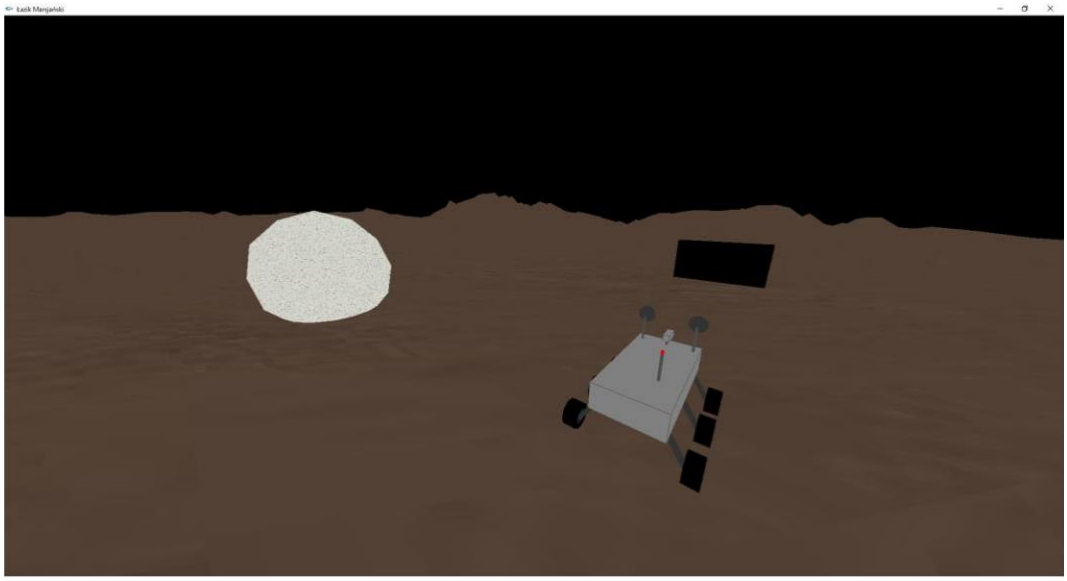


Czarny obiekt – ściana – bez tekstur:



Całość projektu prezentuje się następująco:





06.05.2021

Grafika komputerowa

Laboratorium numer 5

„Sterowanie obiektem głównym”

Brajan Miśkowicz
Grupa L4, 163976

Zaimplementowane w tym ćwiczeniu zostało sterowanie łazikiem. Działa ono podobnie, jak w typowych grach – sterowanie polega na jeździe przód-tył strzałkami **w** i **s**, których naciśnięcie powoduje wprawienie łazika w ruch, im dłużej są wciśnięte, tym szybciej porusza się łazik, aż do osiągnięcia pewnej wartości. Gdy przycisk przestaje być wciśnięty to spowalnia. Strzałki **a** i **d** powodują skręt w lewo/prawo o 8 stopni wówczas, gdy się je naciśnie. Kiedy nie są wciśnięte łazik może jechać tylko w przód/tył. Poniższy kod pokazuje obsługę strzałek:

```
void processKeyboardKeys(unsigned char key, int x, int y)
{
	switch (key) {
	case 'w':
		if (speed < 1.5)	speed += 0.03; // increasing speed till limit when moving forward, or decreasing speed when moving backwards
		if(turning_angle == 0)		//when turning angle is 0
		{
			rotation = 0;  //rover is not rotating
			angular_speed = 0;	//and agular speed is none
		}
		break;
	case 's':
		if (speed > -0.6)	speed -= 0.03; //decreasing speed till limit when moveing forward, or increasing speed when moving backwards
		if (turning_angle == 0)		//when turning angle is 0
		{
			rotation = 0;  //rover is not rotating
			angular_speed = 0;	//and agular speed is none
		}
		break;
	case 'd':
			turning_angle += 8;   //increasing turning angle by 8 degrees
			rotation = 0.8 / tan(turning_angle / (180 / GL_PI));
	//calculating the rotation of rover
			angular_speed = -speed / rotation;  //calculating the angular_speed
		break;
	case 'a':
			turning_angle -= 8;   //decreasing turning angle by 8 degrees
			rotation = 0.8 / tan(turning_angle / (180 / GL_PI));
	//calculating the rotation of rover
			angular_speed = -speed / rotation;  //calculating the angular_speed
		break;
	}
}
```

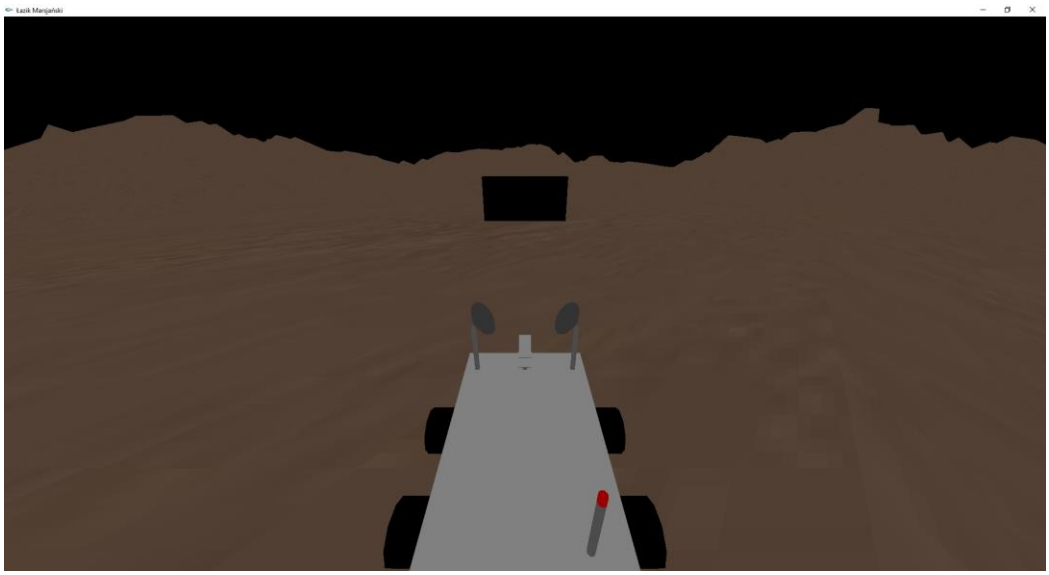
Zwalnianiem, gdy przyciski **w/s** przestają być wciśnięte zajmuje się niewielka funkcja wywoływana co 100ms, zmniejszająca prędkość o 5%. Znajduje się tutaj również iterująca się co 100ms zmienna, która po osiągnięciu wartości 15 wraca do 0 – odpowieda ona za to, żeby co 1500ms dioda znajdująca się na antence łazika migała na czerwono, czego dalsza implementacja zaprezentowana będzie w funkcji renderującej:

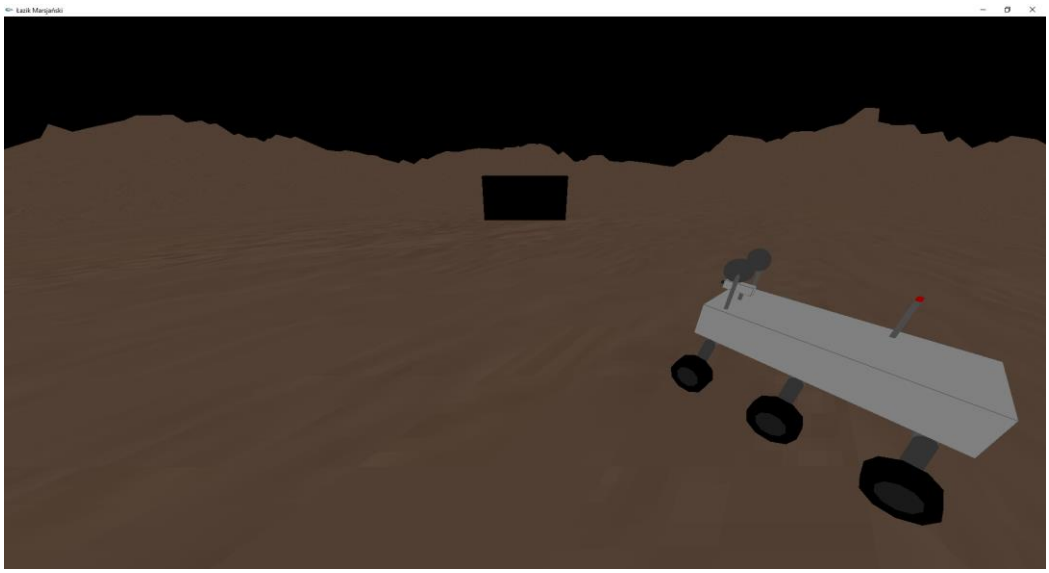
```
void timerCallback(int value)
{
	glutTimerFunc(100, timerCallback, 0);		//called every 100ms
	speed = 0.95 * speed; //decreasing the speed by 5%
	diode_time += 1;	//iteration the diode glowing variable
	if (diode_time > 14)  diode_time = 0;			//every 1500ms reseting it to 0
}
```

W funkcji renderującej wywoływanie łazika zostało zmienione zostały wyądrębnione trzy funkcje – dwie dotyczące antenn oraz jedna dotycząca diody na antence. Dzięki temu udało się zaimplementować obrót wokół własnej osi obu antenn w przeciwnych kierunkach, oraz oraz wspomniane wyżej miganie diody. Kiedy zmienna jej dotycząca osiąga wartość 0, to wyświela się kolor czerwony, w przeciwnym razie nieco ciemniejszy. Obrót każdej z anteny działa dzięki dwóm zmiennym obliczającym rotacje. Jest tutaj również obliczona prędkosc ostateczna łazika, jego kąt obrotu i nowe koordynaty X i Z po przesunięciu. Sam obrót jest oparty na osi skrętnej, a łazik obraca się wokół punktu znajdującago się w tylnej, środkowej części łazika, aby ułatwić sterowanie.

```
        //changing antennas' rotation variables
        antenna_angle += 0.01;
        antenna_rotation = 1 / tan(antenna_angle / (180 / GL_PI));

        glPushMatrix();       //stacinkg the object
        glMatrixMode(GL_MODELVIEW);

        rover_angle += angular_speed;//calculating the angle by angular speed
        pos_x += cos(rover_angle) * speed;  //calculating the new X coordinate
        pos_z += -sin(rover_angle) * speed; //calculating the new Z coordinate
        glTranslatef(pos_x, 0, pos_z);      //moving the rotation center to rover
        glRotatef(rover_angle * (180/ GL_PI), 0, 1, 0);    //rotating the rover
        glTranslatef(-pos_x, 0, -pos_z);    //moving it back
        glTranslatef(pos_x, 0, pos_z);      //moving the rover
        Lazik Marsjanski = Lazik();  //rendering the rover
        //reseting the rotating variables, so rover moves only forward and backwards, when A
and D are not clicked
        turning_angle = 0;
        rotation = 0;
        angular_speed = 0;


        glPushMatrix();       //stacinkg the object
        glTranslatef(9.0, 0, -2.0);  //moving the rotation center to wheel
        glRotatef(antenna_angle * (180 / GL_PI), 0, 1, 0); //rotating the antenne
        glTranslatef(-9.0, 0, 2.0);  //moving it back
        Marsjanski.antenna1();       //rendering the first antenne
        glPopMatrix(); //unstacinkg the object

        glPushMatrix();       //stacinkg the object
        glTranslatef(9.0, 0, 2.0);   //moving the rotation center to wheel
        glRotatef(-antenna_angle * (180 / GL_PI), 0, 1, 0);       //rotating the antenne
        glTranslatef(-9.0, 0, -2.0); //moving it back
        Marsjanski.antenna2();       //rendering the second antenne
        glPopMatrix(); //unstacinkg the object

        if (diode_time == 0)  Marsjanski.diode(0.9);       //diode red glowing when the
variable is equal 0
        else    Marsjanski.diode(0.6);       //and showing it darker, if not

        glPopMatrix(); //unstacinkg the object
```

Realizacja poruszania się wygląda następująco:





20.05.2021

Grafika komputerowa

Laboratorium numer 6

„Kolizje”

Brajan Miśkowicz
Grupa L4, 163976

Do zastosowania systemu kolizji został użyty wykorzystany wcześniej plik .obj zawierający podłoże składające się z trójkątów. Stworzona została dodatkowa klasa *Vertices,* której pierwsza część bazuje na tej, rysującej trójkąty obiektu, ponieważ są one w niej wykorzystywane. Po wczytaniu wierzchołków i trójkątków zamiast wyrysować je, ta klasa przekazuje koordynaty 3 wierzchołków do funkcji *IsInside*, w której obliczane są wszystkie wierzchołki o koordynatach, których wartości są liczbami całkowitymi, a następnie ich wysokości zapisywane są do tablicy *vertices*.
Samo obliczanie wierzchołków jest dość skomplikowane, co wynika ze zmiennoprzecinkowego charakteru wartości pozycji wierzchołków – wykluczają one proste sposoby sprawdzania, czy punkt znajduje się w trójkącie. Pierwszym krokiem tutaj jest wyznaczenie tego, który z trzech wierzchołków ma najmniejszą, który środkową, a który największą wartość koordynatu X. Następnie korzystając ze wzoru na równanie prostej przechodzącej przez dwa punkty obliczany jest punkt na prostej przechodzącej przez pierwszy i trzeci wierzchołek (pierwszy – tzn. ten o najmniejszym X, ostatni – największy X), który znajduje się na tej samej pozycji X, co środkowy wierzchołek. Dzięki temu można sprawdzić, czy wierzchołek drugi znajduje się nad, czy pod tą prostą – te dwa warunki są używane w dalszych obliczeniach.
Zakres wartości X, na których szukane są wierzchołki to wartość koordynatu X pierwszego wierzchołka oraz wartość koordynatu X ostatniego wierzchołka. Zakres Z zależy od wspomnianego wyżej warunku, tzn. jeżeli środkowa wartość Z jest nad prostą, to zakres Z zaczyna się od prostej przechodzącej przez wierzchołki pierwszy i ostatni – dolny bok, a kończy na prostej przechodzącej przez punkt pierwszy i środkowy dla X mniejszych od koordynatu X środkowego wierzchołka (tzn. do środkowego wierzchołka wartości maksymalne znajdują się na boku łączącym pierwszy i drugi wierzchołek), a następnie maksymalne Z znajduje się na prostej przechodzącej między środkowym i ostatnim wierzchołkiem. Mówiąc prościej – wartości Z znajdują się między dolnym bokiem, a górnymi dwoma.
Odwrotnie jest, gdy wierzchołek znajduje się pod prostą. Wówczas dwa boki są u dołu i to ich proste wyznaczają minimalną wartość Z dla danego X, a maksymalna znajduje się na prostej przechodzącej przez pierwszy i ostatni wierzchołek. Tutaj, podobnie jak wcześniej, wykorzystywane są równania na proste przechodzące przez dwa punkty, sam wzór w postaci kierunkowej, czyli wykorzystywanej w projekcie prezentuje się matematycznie następująco:

$$y = \frac{y_A - y_B}{x_A - x_B}x + \left(y_A - \frac{y_A - y_B}{x_A - x_B} \cdot x_A\right)$$

gdzie A to pierwszy z wykorzystywanych punktów, a B drugi. Tutaj zamiast koordynatów Y wykorzystywane są koordynaty Z.

Ostatecznie do obliczenia wysokości punktu używane są współrzędne barycentryczne, a konkretnie wzory wykorzystywane do obliczania takich wysokości. Otrzymane wysokości współrzędnych zapisywane są do tablicy o wymiarach 600x600, ponieważ taką wielkość ma podłoże.

Nie jest to rozwiązanie idealne, ponieważ aproksymacje związane z całkowitymi wartościami czasem powodują małe różnice w wysokościach, przez co porównując wysokości, na których znajduje się łazik oraz wysokości podłoża trzeba dodać niewielkie wartości dla dobrego działania.

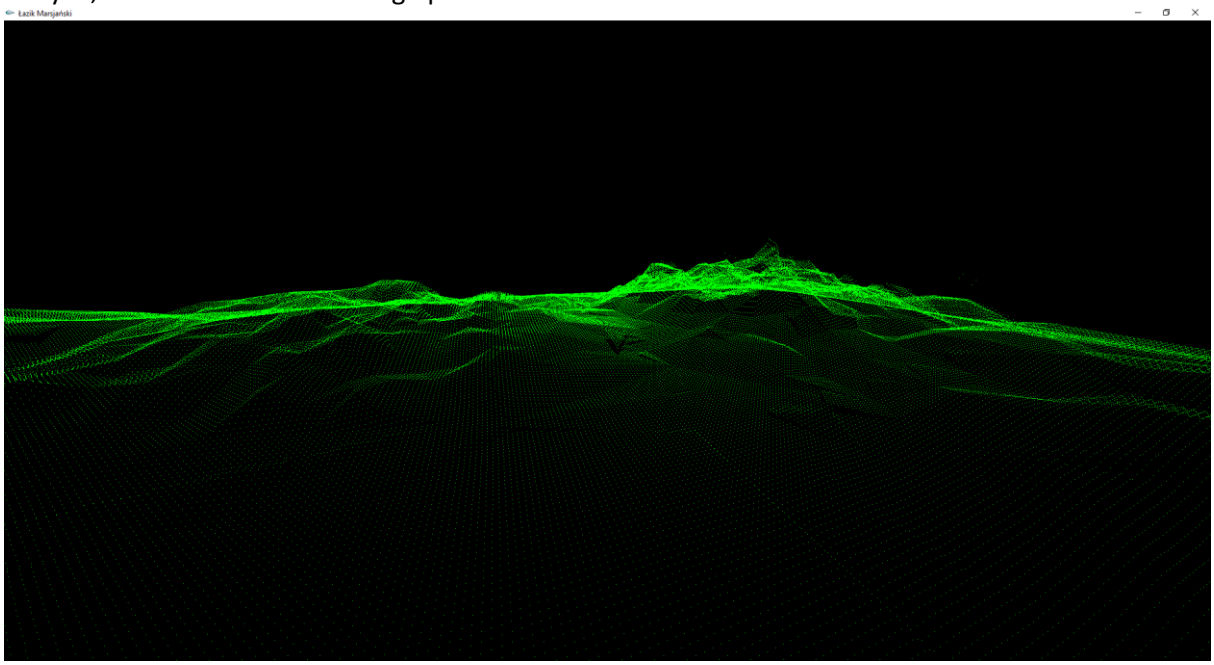
Plik nagłówkowy *Vertices.h*

```
#pragma once
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include "includes/glut.h"
#include "includes/GL.H"

using namespace std;

class Vertices
{
public:
	float vertices[600][600];	//array for all calculated points
	float minmax[3][2];	//array for triangle vertices positions
	float middleZ; //value of Z coordinate of point which is on X posisition of middle vertex and it's Z position is on straight line which is passing through first and last vertex
	float Zmin;	//maximum Z value of 3 vertices of triangle
	float Zmax;	//minimum Z value of 3 vertices of triangle
	struct vertex
	{
		float x;
		float y;
		float z;
	};
	Vertices(string filename);//Read function
	void IsInside(float x1, float x2, float x3, float z1, float z2, float z3, float y1, float y2, float y3);  //finds points inside the given triangle
	void min(float x1, float x2, float x3, float z1, float z2, float z3);	//finds for a minimum X position value from 3 vertices of triangle
	void max(float x1, float x2, float x3, float z1, float z2, float z3);	//finds maximum X position value from 3 vertices of triangle
	void middle(float x1, float x2, float x3, float z1, float z2, float z3); //finds middle X position value from 3 vertices of triangle
private:
	vector<vector<GLfloat>> v;//Store vertex (x,y,z) coordinates
	vector<vector<GLint>> f;//Store the three vertex indexes of the face
};
```

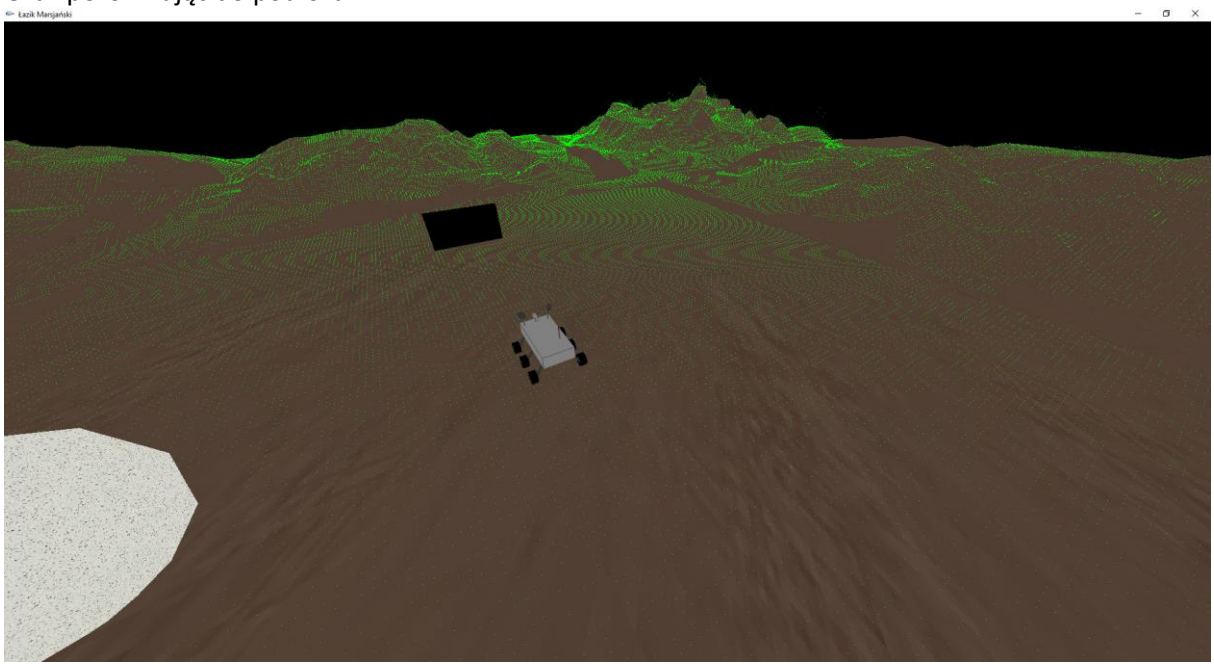
Plik *Vertices.cpp* (ze względu na widoczność i długość linijek umieszczone są zrzuty ekranu kodu):

```cpp
#include "Vertices.h"

Vertices::Vertices(string filename)
{
    ifstream file(filename);
    string line;
    while (getline(file, line))
    {
        if (line.substr(0, 1) == "v")
        {
            vector<GLfloat> Point;
            GLfloat x, y, z;
            istringstream s(line.substr(2));
            s >> x; s >> y; s >> z;
            Point.push_back(x);
            Point.push_back(y);
            Point.push_back(z);
            v.push_back(Point);
        }
        if (line.substr(0, 1) == "f")
        {
            vector<GLint> vIndexSets;
            GLint u, v, w;
            istringstream vtns(line.substr(2));
            vtns >> u; vtns >> v; vtns >> w;
            vIndexSets.push_back(u - 1);
            vIndexSets.push_back(v - 1);
            vIndexSets.push_back(w - 1);
            f.push_back(vIndexSets);
        }
    }
    file.close();
    for (int i = 0; i < f.size(); i++)
    {
        //Three vertices
        vertex a, b, c;
        GLint firstVertexIndex = (f[i])[0];//Remove the vertex index
        GLint secondVertexIndex = (f[i])[1];
        GLint thirdVertexIndex = (f[i])[2];

        a.x = (v[firstVertexIndex])[0];//The first vertex
        a.y = (v[firstVertexIndex])[1];
        a.z = (v[firstVertexIndex])[2];

        b.x = (v[secondVertexIndex])[0]; //The second vertex
        b.y = (v[secondVertexIndex])[1];
        b.z = (v[secondVertexIndex])[2];

        c.x = (v[thirdVertexIndex])[0]; //The third vertex
        c.y = (v[thirdVertexIndex])[1];
        c.z = (v[thirdVertexIndex])[2];

        //looking for points inside every triangle
        IsInside(a.x, b.x, c.x, a.z, b.z, c.z, a.y, b.y, c.y);
    }

}


void Vertices::IsInside(float x1, float x2, float x3, float z1, float z2, float z3, float y1, float y2, float y3)
{
    min(x1, x2, x3, z1, z2, z3);    //looking for the smallest X position
    max(x1, x2, x3, z1, z2, z3);    //looking for higher X position
    middle(x1, x2, x3, z1, z2, z3); //looking for middle X position

    //calculating the Z coordinate of point which is on X posisition of middle vertex and it's Z position is on straight line which is passing through first and last vertex
    middleZ = ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * minmax[1][0] + minmax[0][1] - ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * minmax[0][0];

    if (middleZ < minmax[1][1]) //if middle vertex is above straight line which is passing through first and last vertex
    {
        for (int i = minmax[0][0]; i <= minmax[1][0]; i++)//i are X positions on tringle from first vertex to middle one
        {
            //for X position there are below calculated minimum, and maximum values using equation of a line passing through two points - first and middle vertex for max, first and last for min
            Zmin = ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * i + minmax[0][1] - ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * minmax[0][0];
            Zmax = ((minmax[0][1] - minmax[1][1]) / (minmax[0][0] - minmax[1][0])) * i + minmax[0][1] - ((minmax[0][1] - minmax[1][1]) / (minmax[0][0] - minmax[1][0])) * minmax[0][0];
            for (int j = Zmin; j <= Zmax; j++)  //j are Z positions in tringle
            {
                //calculating the Y position of given X and Z using barycentric coordinate system
                float det = (z2 - z3) * (x1 - x3) + (x3 - x2) * (z1 - z3);
                float l1 = ((z2 - z3) * (i - x3) + (x3 - x2) * (j - z3)) / det;
                float l2 = ((z3 - z1) * (i - x3) + (x1 - x3) * (j - z3)) / det;
                float l3 = 1.0f - l1 - l2;
                vertices[i + 300][j + 300] = l1 * y1 + l2 * y2 + l3 * y3;   //saving Y position of Vertex in table
            }
        }
        for (int i = minmax[1][0] + 1; i <= minmax[2][0]; i++)//i are now X positions on triangle from middle vertex to lest vertex
        {
            //for X position there are below calculated minimum, and maximum values using equation of a line passing through two points - first and middle vertex for max, first and last for min
            Zmin = ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * i + minmax[0][1] - ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * minmax[0][0];
            Zmax = ((minmax[1][1] - minmax[2][1]) / (minmax[1][0] - minmax[2][0])) * i + minmax[1][1] - ((minmax[1][1] - minmax[2][1]) / (minmax[1][0] - minmax[2][0])) * minmax[1][0];
            for (int j = Zmin; j <= Zmax; j++)
            {
                //calculating the Y position of given X and Z using barycentric coordinate system
                float det = (z2 - z3) * (x1 - x3) + (x3 - x2) * (z1 - z3);
                float l1 = ((z2 - z3) * (i - x3) + (x3 - x2) * (j - z3)) / det;
                float l2 = ((z3 - z1) * (i - x3) + (x1 - x3) * (j - z3)) / det;
                float l3 = 1.0f - l1 - l2;
                vertices[i + 300][j + 300] = l1 * y1 + l2 * y2 + l3 * y3;   //saving Y position of Vertex in table
            }
        }
    }
    else
    {
        for (int i = minmax[0][0]; i <= minmax[1][0]; i++)//i are X positions on tringle from first vertex to middle one
        {
            //for X position there are below calculated minimum, and maximum values using equation of a line passing through two points
            // this time it's: first and middle vertex for min, first and last for max
            Zmax = ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * i + minmax[0][1] - ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * minmax[0][0];
            Zmin = ((minmax[0][1] - minmax[1][1]) / (minmax[0][0] - minmax[1][0])) * i + minmax[0][1] - ((minmax[0][1] - minmax[1][1]) / (minmax[0][0] - minmax[1][0])) * minmax[0][0];
            for (int j = Zmin; j <= Zmax; j++)
            {
                //calculating the Y position of given X and Z using barycentric coordinate system
                float det = (z2 - z3) * (x1 - x3) + (x3 - x2) * (z1 - z3);
                float l1 = ((z2 - z3) * (i - x3) + (x3 - x2) * (j - z3)) / det;
                float l2 = ((z3 - z1) * (i - x3) + (x1 - x3) * (j - z3)) / det;
                float l3 = 1.0f - l1 - l2;
                vertices[i + 300][j + 300] = l1 * y1 + l2 * y2 + l3 * y3;   //saving Y position of Vertex in table
            }
        }
        for (int i = minmax[1][0] + 1; i <= minmax[2][0]; i++)//i are now X positions on triangle from middle vertex to lest vertex
        {
            //for X position there are below calculated minimum, and maximum values using equation of a line passing through two points
            // this time it's: middle and last vertex for min, first and last for max
            Zmax = ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * i + minmax[0][1] - ((minmax[0][1] - minmax[2][1]) / (minmax[0][0] - minmax[2][0])) * minmax[0][0];
            Zmin = ((minmax[1][1] - minmax[2][1]) / (minmax[1][0] - minmax[2][0])) * i + minmax[1][1] - ((minmax[1][1] - minmax[2][1]) / (minmax[1][0] - minmax[2][0])) * minmax[1][0];
            for (int j = Zmin; j <= Zmax; j++)
            {
                //calculating the Y position of given X and Z using barycentric coordinate system
                float det = (z2 - z3) * (x1 - x3) + (x3 - x2) * (z1 - z3);
                float l1 = ((z2 - z3) * (i - x3) + (x3 - x2) * (j - z3)) / det;
                float l2 = ((z3 - z1) * (i - x3) + (x1 - x3) * (j - z3)) / det;
                float l3 = 1.0f - l1 - l2;
                vertices[i + 300][j + 300] = l1 * y1 + l2 * y2 + l3 * y3;   //saving Y position of Vertex in table
            }
        }
    }

}
```

Wyrysowując punkty można sprawdzić, w jaki sposób otrzymane wysokości mają się do wejściowych danych, tzn. do do renderowanego podłoża:



Oraz porównując do podłoża:



Jak widać pozycje punktów nie są idealne, ale bardzo mocno zadowalające, dodatkowo samo obliczenie ich pozycji nie obciąża symulacji. Są one podstawą do sprawdzania kolizji łazika.

W pliku głównym wywoływany jest konstruktor dla podłoża:

```
Vertices coordinates = Vertices("tekstury.obj");
```

Dodatkowo samo dodanie kolizji z innymi obiektami polega jedynie na nadpisaniu tablicy wysokości z podłoża – zwiększane do 20, tak aby były wysoko nad łazikiem, dzieje się to w funkcji *main()*, przed główną pętlą renderującą. Koordynaty do zmian brane są z plików *.obj*, tak więc nie trzeba do tego żadnych obliczeń:

```
//correcting the collisions array, so objects also have collisions
for (int i = 69; i <= 71; i++)
{
	for (int j = -10; j <= 10; j++)	coordinates.vertices[i + 300][j + 300] = 20;
}
for (int i = -10; i <= 10; i++)
{
	for (int j = -60; j <= -40; j++)	coordinates.vertices[i + 300][j + 300] = 20;
}
```

Same kolizje sprawdzane są dzięki czterem punktom określającym pozycje łazika. Znajdują się one: po jednym z przodu i z tyłu – u dołu ciała łazika, oraz po jednym między przednimi kołami i tylnymi kołami – na wysokości spodu kół. Punkty te są reprezentowane za pomocą tablic z koordynatami X, Y i Z:

```
//collision points
float backMiddle[] = {0.0, 5.0, 0.0 };
float backWheels[] = { 1.0, 2.2, 0.0 };
float frontWheels[] = { 9.0, 2.2, 0.0 };
float frontMiddle[] = { 10.0, 5.0, 0.0 };

//collision checking variable
bool canMove = 1;
```

Zadeklarowana tutaj zmienna *canMove* jest flagą, która przyjmuje wartość 1, kiedy nie ma kolizji, oraz 0, gdy taka zostanie znaleziona. Jej wartość jest modyfikowana w funkcji *collisionDetection()* tuż przed renderowaniem łazika. Sama funkcja zmienia pozycje punktów, a następnie sprawdza, czy wysokość punktów jest mniejsza, lub równa odpowiadającym im wysokościom na tych współrzędnych zapisanym w taliblicy kolizji – jeżeli jest, to flaga zmienia wartość na 0 – wówczas łazik jest renderowany, jak wcześniej, a jeżeli nie, to przypisana jest jej wartość 1 – łazik ma prędkość 0. I jest cofnięty o 2 jednostki, aby zapobiec buggowaniu w miejscu kolizji.

```
bool collisionDetection()
{
	//calculating the X position of rover
	backMiddle[0] += cos(rover_angle) * speed;
	backWheels[0] += cos(rover_angle) * speed;
	frontWheels[0] = pos_x  + cos(rover_angle) * 10;
	frontMiddle[0] = pos_x + cos(rover_angle) * 10;
	//calculating the Z position of rover
	backMiddle[2] += -sin(rover_angle) * speed;
	backWheels[2] += -sin(rover_angle) * speed;
	frontWheels[2] = pos_z  - sin(rover_angle) * 10;
	frontMiddle[2] = pos_z - sin(rover_angle) * 10;
	//checking if there is collision, and if is return 1
	if (coordinates.vertices[int(frontMiddle[0]) + 300][int(frontMiddle[2]) + 300] >=
frontMiddle[1] - 1.2) return 0;
	if (coordinates.vertices[int(backMiddle[0]) + 300][int(backMiddle[2]) + 300] >=
backMiddle[1] - 1.2)  return 0;
	if (coordinates.vertices[int(frontWheels[0]) + 300][int(frontWheels[2]) + 300] >=
frontWheels[1] + 1.5) return 0;
	if (coordinates.vertices[int(backWheels[0]) + 300][int(backWheels[2]) + 300] >=
backWheels[1] + 1.5)  return 0;
	//if there's no collision return 1
	return 1;
}
```

Działanie kolizji zostało zaprezentowane za pomocą filmiku:
https://drive.google.com/file/d/126xEfH1zpFCwMUZwqYJqBEBaezM7-nT3/view?usp=sharing

17.06.2021

Grafika komputerowa

Laboratorium numer 7

„Fabuła gry”

Brajan Miśkowicz
Grupa L4, 163976

Pierwszym elementem budowania fabuły projektu było poprawienie kolizji z podłożem – tak, aby teraz łazik wjeżdżał na wyżej i zjeżdżał na niżej położone tereny. Zrealizowane zostało to poprzez modyfikację kodu funkcji *collisionDetection()*. Zostały tutaj wprowadzone zabezpieczenia przed wyjazdem poza mapę (ma ona wymiary 300 x 300), oraz obsługa kamery tak, aby podążała ona za łazikiem. Ponadto funkcja blokuje jazdę, gdy wysokość w punkcie jest równa 600 – taka wysokość ustalana jest tak, gdzie znajdują się przeszkody, zabezpiecza także przed błędnymi wartościami wysokości, jakie mogły się pojawiać przy ci generowaniu. Położenie łazika w osi Z zmienia się w zależności od dwóch punktów znajdujących się między kołami wspomnianych we wcześniejszym sprawozdaniu.

```
bool collisionDetection()
{
	//calculating the X position of rover points
	backWheels[0] += cos(rover_angle) * speed;
	frontWheels[0] = pos_x + cos(rover_angle) * 10;
	//calculating the Z position of rover points
	backWheels[2] += -sin(rover_angle) * speed;
	frontWheels[2] = pos_z - sin(rover_angle) * 10;
	//calculating the camera position
	cameraPos[0] = pos_x + cos(rover_angle) * (-15.0) + cameraX;
	cameraPos[2] = pos_z - sin(rover_angle) * (-15.0) + cameraZ;
	cameraPos[1] = pos_y + 22.8;
	//protection against falling off the map at its end for front point
	if (frontWheels[0] < -285)
	{
		frontWheels[0] = -285;
	}
	else if (frontWheels[0] > 285)
	{
		frontWheels[0] = 285;
	}
	if (frontWheels[2] < -285)
	{
		frontWheels[2] = -285;
	}
	else if (frontWheels[2] > 285)
	{
		frontWheels[2] = 285;
	}

	//protection against falling off the map at its end for back point
	if (backWheels[0] < -285)
	{
		backWheels[0] = -285;
		pos_x = -285;
	}
	else if(backWheels[0] > 285)
	{
		backWheels[0] = 285;
		pos_x = 285;
	}

	if (backWheels[2] < -285)
	{
		backWheels[2] = -285;
		pos_z = -285;
	}
	else if (backWheels[2] > 285)
	{
		backWheels[2] = 285;
		pos_z = 285;
	}


	if (coordinates.vertices[int(frontWheels[0]) + 300][int(frontWheels[2]) + 300] == 600 ||
coordinates.vertices[int(backWheels[0]) + 300][int(backWheels[2]) + 300] == 600)  return 0;
	//checking if there is object, returning 0 if it is
	else if (coordinates.vertices[int(backWheels[0]) + 300][int(backWheels[2]) + 300] == 0 ||
coordinates.vertices[int(frontWheels[0]) + 300][int(frontWheels[2]) + 300] == 0)  return 1;
	//protection against wrong values of height
```

```
        if (frontWheels[1] > backWheels[1])      //taking the point, which is higher
        {
                //changing the height of rover, rover points, and camera position
                frontWheels[1] = coordinates.vertices[int(frontWheels[0]) + 300][int(frontWheels[2]) +
300] + 0.5;
                backWheels[1] = coordinates.vertices[int(frontWheels[0]) + 300][int(frontWheels[2]) +
300] + 0.5;
                pos_y = frontWheels[1] + 0.5;
                cameraPos[1] = pos_y + 23.2;
                return 1;       //returning 1, cause there is no colision
        }
        else    //same as above
        {
                frontWheels[1] = coordinates.vertices[int(backWheels[0]) + 300][int(backWheels[2]) +
300] + 0.5;
                backWheels[1] = coordinates.vertices[int(backWheels[0]) + 300][int(backWheels[2]) +
300] + 0.5;
                pos_y = backWheels[1] + 0.5;
                cameraPos[1] = pos_y + 23.2;
                return 1;       //returning 1, cause there is no colision
        }

        //if there's no collision return 1
        return 1;
}
```

Kolejnym etapem było dodanie dwóch elementów fabularnych – gwiazd, oraz statków UFO. Gwiazdy służą do zbierania, natomiast ufo może się pojawić w trybach agresywnych, tam poruszają się one odbijając się od krawędzi map i zatrzymują, gdy natkną się na łazik i wówczas zabierają graczowi punkty życia. Modele tych obiektów pobierane są z plików .obj, a ich działanie określone jest w ramach poniższych zmiennych:

```
float stars[600][600];   //stars collecting array
int numberOfStars = 1;
vector<vector<int> > starsPos(numberOfStars, vector<int>(3));     //positions X, Y, Z
vector<vector<int> > isStar(numberOfStars, vector<int>(2));       //index and variable responsible for
dissapearing
int starsCollected = 0;
```

dla gwiazd – oznaczają one odpowiedno tablicę z wysokościami, ilość gwiazd na mapie, wektor zawierający pozycje X, Y i Z każdej gwiazdy, wektor zawierający jej ID i zmienną mówiącą, czy gwiazda została zebrana, oraz ilość zebranych gwiazd.

```
int numberOfUfos = 1;
vector<vector<int> > ufoPos(numberOfUfos, vector<int>(3));        //positions X, Y, Z
vector<vector<int> > ufoAttacks(numberOfUfos, vector<int>(2));    //1/-1 variables for X, Z iterations
```

dla ufo podobnie pojawia się zmienna określająca ilość statków oraz wektory – jeden zawierający pozycje X, Y i Z, oraz drugi zawierający wartości kierunkowe X i Z.
Zarówno jak gwiazdy, tak i statki ufo mają generowane losowo pozycje na mapie. Dodatkowo te drugie losowo generują wartości kierunkowe -1/1 dla osi X i Z – mówią one, jak poruszać będzie się dany statek, puki nie dojdzie do ścian, bo wówczas kiedunek zostanie odwrócony. Dzięki takiemu rozwiązaniu statki pojawiają się w losowych miejscach i poruszają się w losowych, różnych miejscach, co czyni rozgrywkę trudniejszą.

Zawarty poniżej kod zawiera funkcję rysującą gwiazdy. Sprawdza ona w pętli dla każdej gwiazdy, czy nie została już zebrana przez gracza, a jeżeli nie to jest wyrysowana, a wysokość na jej pozycji i na koordynatach +/-5 na obu osiach X i Z w tablicy gwiazd zostają ustawione na 333 – dzięki temu gracz znajdując się pod gwiazdą, a niekoniecznie idealnie na jej środku jest w stanie ją zebrać.

```
void StarDrawing()
{
	for (int i = 0; i < numberOfStars; i++)
	{
		if (isStar[i][1] == 1)  //if star wasn't collected
		{
			glMatrixMode(GL_MODELVIEW);
			glPushMatrix(); //stacking the object
			glColor3f(1.0, 0.827, 0);	//yellow colour
			glTranslatef(starsPos[i][0], starsPos[i][1] + 12, starsPos[i][2]);
	//moving it back
			glRotatef(-antenna_angle * (180 / GL_PI), 0, 1, 0);	//rotating the star
			glTranslatef(-starsPos[i][0], -starsPos[i][1] + 12, -starsPos[i][2]);
	//moving the rotation center to star
			star.DrawObj(starsPos[i][0], starsPos[i][1] + 12, starsPos[i][2]);
	//rendering the star
			glPopMatrix();  //unstacinkg the object


			//filling stars collecting array
			for (int k = starsPos[i][0] - 5; k< starsPos[i][0] + 5; k++)
			{
				for (int j = starsPos[i][2] - 5; j < starsPos[i][2] + 5; j++)
				{
					stars[k + 300][j + 300] = 333;//changing Y coordinate for
stars
				}
			}
		}
	}
}
```

Druga funkcja obsługująca gwiazdy sprawdza, czy tryb gry (zostaną one omówione jeszcze później, przy menu) to ten, w którm w określonej ilości czasu zbiera się jak najwięcej gwiazd. W nim tych gwiazd może zabraknąć, dlatego w takiej sytuacji wprowadzona zostaje modyfikacja rozgrywki – zwiększa się ich ilość o jeden i ta nowa gwiazda pojawia się na mapie. Jeżeli jest to drugi tryb, w którym należy zebrać określoną ilość gwiazd w jak najniższym czasie, oraz zostały zebrane wszystkie gwiazdy, to flaga *endRound* zostaje ustawiona na 1 – oznacza to, że kończy się rozgrywana runda. Jeżeli te warunki nie zostały spełnione, to funkcja sprawdza, czy punkty łazika nie znajdują się pod gwiazdą – jeśli tak, to ona znika, a ilość zebranych gwiazd powiększa się.

```
void starsCollecting()
{
	if (timerMode == 0 && starsCollected == numberOfStars)//if it's firstmode, and stars are all
picked
	{
		numberOfStars += 1;	//increasing number of stars
		//resizing both vectors
		starsPos.resize(numberOfStars + 1, vector<int>(3));
		isStar.resize(numberOfStars + 1, vector<int>(2));
		randomX = ((250 + 250) * ((float)rand() / RAND_MAX)) - 250;	//random X
coordinate
		randomZ = ((250 + 250) * ((float)rand() / RAND_MAX)) - 250;	//random Z
coordinate
		//setting star coordinates
		starsPos[numberOfStars - 1][0] = randomX;
		starsPos[numberOfStars - 1][2] = randomZ;
		starsPos[numberOfStars - 1][1] = coordinates.vertices[int(randomX)][int(randomZ)] +
12;
		isStar[numberOfStars -1][0] = numberOfStars - 1; //setting star index
		isStar[numberOfStars -1][1] = 1; //setting the flag, so the star will be rendered
	}
```

```
        else if (timerMode == 1 && starsCollected == numberOfStars)      endRound = 1;    //if it's
second mode, and stars are all picked the round is over
        else
{
                for (int i = 0; i < numberOfStars; i++)
                {
                        if (abs(frontWheels[0] - starsPos[i][0]) < 5 && abs(frontWheels[2] -
starsPos[i][2]) < 5 && isStar[i][1] == 1)
                        {
                                isStar[i][1] = 0;       //star dissapears
                                starsCollected += 1;    //increasing number of collected stars
                        }
                }
                for (int i = 0; i < numberOfStars; i++)
                {
                        if (abs(backWheels[0] - starsPos[i][0]) < 5 && abs(backWheels[2] -
starsPos[i][2]) < 5 && isStar[i][1] == 1)
                        {
                                isStar[i][1] = 0;       //star dissapears
                                starsCollected += 1;    //increasing number of collected stars
                        }
                }
        }
}
```

Funkcja rysująca ufo sprawdza w pętli, czy statki nie wychodzą poza mapę i jeżeli tak, to zmienia ich kierunek poruszania. Następnie porusza łazikami, jeżeli zmienna *ufoTime* ma wartość 0 (Jej wartość zmienia sie w zegarze. Zwiększa się ona co 0.1 sekundy o 1, aż do 5 lub 25, wówczas zmienia wartość na 0 - tak więc statki poruszają się co pół sekundy). Dzięki temu statki poruszają się skokowo, co sprawia wrażenie, że skanują teren, a dodatkowo pozwala na wprowadzenie pewnej modyfikacji, tzn. gdy ufo znajdzie się nad graczem to wartość *ufoTime* zostaje ustawione na 6. To sprawia, że statki poruszą się dopiero po 2 sekundach po zaatakowaniu gracza, co zabezpiecza przed natychmiastową utratą całego zdrowia, pozwala na ucieczkę, oraz zmusza statki do pozostania nad graczem, jeżeli ten się nie rusza i do zabrania mu wszyskich punktów zdrowia.

```
void ufoDrawing()
{
        for (int index = 0; index < numberOfUfos; index++)
        {
                //changing X direction if ufo has reached map end
                if (ufoPos[index][0] > 283)      ufoAttacks[index][0] = -1;
                else if (ufoPos[index][0] < -283)        ufoAttacks[index][0] = 1;
                //changing Z direction if ufo has reached map end
                if (ufoPos[index][2] > 283)      ufoAttacks[index][1] = -1;
                else if (ufoPos[index][2] < -283)        ufoAttacks[index][1] = 1;

                if (ufoTime == 0)       //every 0.5s ufos are moved
                {
                        ufoPos[index][2] += ufoAttacks[index][1];
                        ufoPos[index][0] += ufoAttacks[index][0];
                        ufoPos[index][1] = coordinates.vertices[ufoPos[index][0]][ufoPos[index][2]] +
12;
                }

                glPushMatrix(); //stacking the object
                glColor3f(7.0, 0.1, 0.5);       //pink colour
                glTranslatef(ufoPos[index][0], 12, ufoPos[index][2]);    //moving it back
                glRotatef( 2 * antenna_angle * (180 / GL_PI), 0, 1, 0);  //rotating the ufo
                glTranslatef(-ufoPos[index][0], 12, -ufoPos[index][2]);  //moving the rotation center
to ufo
                ufo.DrawObj(ufoPos[index][0], 12, ufoPos[index][2]);     //rendering the ufos
                glPopMatrix();  //unstacinkg the object
        }
}
```

Kod niżej przedstawia mechanikę atakowania gracza. Sprawdzane jest, czy odległość od któregoś statku (z racji ich niewielkiej ilości takie rozwiązanie jest najprostrze) jest mniejsza od 15 na obu osiach od któregoś z punktów, lub od obo i jeżeli tak, to gracz trafi życie, a wspomniana na stronie wyżej wartość odpowiedzialna za czas poruszania się statków ustawiona zostaje na 6.

```
void ufosAttacks()
{
	for (int index = 0; index < numberOfUfos; index++)
	{
		if (abs(ufoPos[index][0] - backWheels[0]) < 15 && abs(ufoPos[index][2] -
backWheels[2]) < 15 || abs(ufoPos[index][0] - frontWheels[0]) < 15 && abs(ufoPos[index][2] -
frontWheels[2]) < 15 ||
			(abs(ufoPos[index][0] - backWheels[0]) < 15 && abs(ufoPos[index][2] -
backWheels[2]) < 15 && abs(ufoPos[index][0] - frontWheels[0]) < 15 && abs(ufoPos[index][2] -
frontWheels[2])))	//checking if any ship is above rover
		{
			hp -= 1;//taking users HP
			ufoTime = 6;	//delaying ships movement
		}
	}
}
```

Jeżeli chodzi o przyciski klawiatury, to została dodana obsługa dwóch kolejnych – przycisku „P” odpowiedzialnego za przerwanie rozgrywki bez jej kończenia, co działa dzięki flacze *pause,* oraz przycisku „Escape”, który wyłącza symulację.
Kolejnym nowym elementem jest menu, które uruchamiane jest za pomocą prawego przycisku myszy, a jego kliknięcie przerywa grę, przez co gracz nie traci niczego zmieniając opcje. Główna funkcja obsługująca menu zawiera jedynie najprościej mówiąc drzewo z jego podmenu. W samym menu znajduje się kilka ciekawych opcji – można zmienić kamerę w podmenu z trzecioosobowej na odległą, zresetować rozgrywany poziom, wejść do strony głównej, lub wejść w podmenu z trybami rozgrywek. W nim znajdują się kolejne dwa podmenu umożliwiające wybór trybu w którym zbiera się określoną ilość gwiazd na czas, lub trybu, w którym w określonym czasie należy zebrać jak najwięcej gwiazd. Każdy z trybów zawiera po trzy poziomy z jedynie gwiazdami, oraz po trzy poziomy z gwiazdami i statkami ufo, co łącznie daje 12 możliwych poziomów.

```
void createPopupMenus()
{
	//first mode menus
	easyTime = glutCreateMenu(processSpeedLevels);
	glutAddMenuEntry("Level 1", LVL1);
	glutAddMenuEntry("Level 2", LVL2);
	glutAddMenuEntry("Level 3", LVL3);
	hardTime = glutCreateMenu(processSpeedLevels);
	glutAddMenuEntry("Level 1", LVL4);
	glutAddMenuEntry("Level 2", LVL5);
	glutAddMenuEntry("Level 3", LVL6);
	//second mode menus
	easySpeed = glutCreateMenu(processTimeLevels);
	glutAddMenuEntry("Level 1", LVL7);
	glutAddMenuEntry("Level 2", LVL8);
	glutAddMenuEntry("Level 3", LVL9);
	hardSpeed = glutCreateMenu(processTimeLevels);
	glutAddMenuEntry("Level 1", LVL10);
	glutAddMenuEntry("Level 2", LVL11);
	glutAddMenuEntry("Level 3", LVL12);
	//menus for modes difficulty
	modesTime = glutCreateMenu(emptymenu);
	glutAddSubMenu("easy - normal mode", easyTime);
	glutAddSubMenu("hard - aggressive", hardTime);

	modesSpeed = glutCreateMenu(emptymenu);
	glutAddSubMenu("easy - normal mode", easySpeed);
	glutAddSubMenu("hard - aggressive", hardSpeed);
	//menu for camera view
	modesCamera = glutCreateMenu(processmodesCamera);
	glutAddMenuEntry("Third person camera view", THIRDPERSON);
```

```
        glutAddMenuEntry("Distance camera view", DISTANCE);
        //menu for modes
        gameModes = glutCreateMenu(emptymenu);
        glutAddSubMenu("Beat the time records", modesSpeed);
        glutAddSubMenu("Collecting in limited time", modesTime);
        //main menu options
        mainMenu = glutCreateMenu(processMainMenu);
        glutAddMenuEntry("Go to main menu", MAINMENU);
        glutAddMenuEntry("Restart the game round", RESTARTMENU);
        glutAddSubMenu("Change game mode", gameModes);
        glutAddSubMenu("Change camera display", modesCamera);
        glutAddMenuEntry("Disable/enable texturing", TEXTURING);
        // attach the menu to the right button
        glutAttachMenu(GLUT_RIGHT_BUTTON);
        // this will allow us to know if the menu is active
        glutMenuStatusFunc(processMenuStatus);
}
```

W samych funkcjach związanych z poziomami gier nie znajduje się jednak nic szczególnie skomplikowanego. Zmieniane sa tak w zależności od poziomu ilości gwiazd, czasu, czy statków, zmieniane rozmiary wektorów i losowane pozucje, resetowana pozycja łazika oraz ustawiane zmienne i flagi, a to wszystko po to, aby umożliwić rozpoczęcie rundy.

Zmieniła się także funkcja *rendercene.* Jedną z pierwszych rzeczy w niej jest teraz sprawdzenie, czy zakończyła się runda. Jeżeli nie, to kamera znajduje się w standardowej pozycji, a jeżeli tak, to kamera jest przesunięta na pozycje, na której wyświetlane jest główne menu, które będzie omówione później. Następnym krokiem jest sprawdzenie flagi mówiącej o tym, czy gracz chce wyświetlać tekstury na podłożu (ta zmiana możliwa jest w menu), czy nie. Zostało to wprowadzone z kilku powodów. Głównym było to, że dla użytkowników o słabszym sprzęcie renderowanie tekstur na takiej ilości trójkątów, jaka znajduje się na podłożu może być mocno obciążająca i powodować problemy z ich wyświetlaniem.

```
if (endRound == 0)
{
        // Set the camera
        gluLookAt(cameraPos[0], cameraPos[1], cameraPos[2], x + frontWheels[0], y + cameraPos[1] - 25,
z + frontWheels[2], 0.0f, 1.0f, 0.0f);
}
else
{
        // Set the camera on main page
        gluLookAt(385, 10, - 0.5, 400, 10, - 0.5, 0.0f, 1.0f, 0.0f);
}

if (texturesFlag == 1)
{
        //rendering ground with textures
        glColor3f(1.0, 1.0, 1.0);
        init(stbi_load("stone.jpg", &width, &height, &nrChannels, 0));
        textures.DrawT();
        glDeleteTextures(1, &textureName);
}
else
{
        //rendering ground without textures
        glColor3f(1.0, 1.0, 1.0);
        textures.DrawG();
}
```

Po wyrysowaniu kilku obiektów sprawdzana jest flaga *pause*, która mówi, czy rozgrywka została przerwana i jeżeli tak, to łazik jest wyświetlony w miejscu, w którym się znajduje. W przeciwnym razie sprawdzane jest, czy rozgrywka się zakończyła (flaga *endRound)* i jeżeli tak to sprawdzana jest kolejna flaga *sortOnce*. Ta ostatnia pozwala na wywołanie znajdującego się w warunku kodu jedynie raz po zakończeniu rozgrywki. Jest tam zależnie od poziomu rozgrywki zapisywany jest wynik do odpowiedniego miejsca w tablicy wyników *results*. Następnie tablica z wynikami jest sortowana i tworzone są po jednym ufo i gwiazda, które pojawiają się na ekranie głównym. Ostatecznie wywoływana jest funkcja *menuRendering*, która rysuje menu, co opisane będzie niżej. Poniżej znajduje się pętla wykonywana po zakończeniu rundy.

```
if (endRound == 1)
{
	if (sortOnce == 0)	//if it's first time the program entered here after round ended
	{
		if (level > 5 && starsCollected == numberOfStars && timer > 0)	//if user succeed in
first mode
		{
			results[level][0] = timer;	//result is saved in results array
			sortResults();	//results array is being sorted
		}
		else if (level < 6 && starsCollected != 0)	//if user succeed in second mode
		{
			results[level][3] = starsCollected;	//result is saved in results array
			sortResults();	//results array is being sorted
		}
		//drawing one star on main page
		isStar[0][0] = 0;
		isStar[0][1] = 1;
		starsPos[0][0] = 408;
		starsPos[0][2] = -20;
		starsPos[0][1] = -13;
		numberOfUfos = 1;
		//drawing one ufo on main page
		ufoPos.resize(numberOfUfos, vector<int>(3));
		ufoAttacks.resize(numberOfUfos, vector<int>(2));
		ufoPos[0][0] = 440;
		ufoPos[0][2] = 45;
		ufoPos[0][1] = 28;
		ufoAttacks[0][0] = 0;
		ufoAttacks[0][1] = 0;
	}
	menuRendering();
}
```

Jeżeli te warunki nie będą spełnione, to łazik porusza się normalnie. Pod koniec funkcji *renderScene* znajduje się jeszcze fragment kodu, w którym wywoływana jest funkcja odpowiedzialna za atakowanie przez statki ufo, o ile gracz ma życie i statek się może poruszyć. Zmieniane są tutaj zmienne odpowiedzialne za wyświetlany tekst dotyczący aktualnego życia graacza, ale o tym na koniec. Jeżeli życie gracza spadnie po atakach do 0, to runda się kończy:

```
if (ufoTime == 0 && hp > 0)
{
	ufosAttacks();
	ufostring = "HP: " + to_string(hp);	//creating string which shows current HP
	hpText = ufostring.c_str();	//saving string in character array
	if (hp == 0)	endRound = 1;	//if user has 0 HP the round is ended
}
```

Wspomniana wyżej na tej stronie funkcja *menuRendering* odpowiedzialna za wypisywanie tekstu na stronie głównej robi to korzystając z funkcji wypisujących podaną tablicę znaków:

```
void printT(float x, float y, float z, const char* text)
{
	const char* c;
	glColor3d(1.0, 1.0, 1.0);	//text colour
	glRasterPos3f(x, y, z);  //text position
	for (c = text; *(c + 4) != '\0'; c++)	//writing
	{
		glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, *c);
	}
}
```

Nieco bardziej rozbudowana, ale bardzo podobna funkcja wykorzystywana jest do wypisywania aktualnego stanu gracza – czasu, ilości pozbieranych gwiazd, oraz HP.

```
void print(float x, float y, float z, const char* text)
{
	//everything as above
	const char* c;
	glColor3d(1.0, 0.0, 0.0);
	glRasterPos3f(x, y, z-1);
	for (c = text; *(c + 4) != '\0'; c++)    //writing text for time
	{
		glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, *c);
	}
	glRasterPos3f(x, y - 2, z-1);
	for (c = starsC; *(c) != '\0'; c++)      //writing amount of stars
	{
		glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, *c);
	}
	glRasterPos3f(x, y - 4, z-1);
	for (c = hpText; *(c + 7) != '\0'; c++)  //writing HP
	{
		glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, *c);
	}
}
```

Modyfikacja zmienny związanych z wypisywaniem ilości punktów życia zostały przedstawione przy okazji wywołania funkcji atakującej łazik. Pozostałe zmieniane są w funkcjach zegarowych. W pierwszej z nich zmieniają się wspomniane już we wcześniejszych laboratoriach zmienne odpowiedzialne za diodę i przyspieszenie, oraz opisana już obsługująca poruszanie i atakowanie statków. Dodatkowo zależenie od rodzaju rozgrywki wypisywana jest nad łazikiem zebrana ilośc gwiazd, lub zebrana ilość gwiazd / ilość gwiazd na mapie.

```
void timerCallback(int value)
{
	glutTimerFunc(100, timerCallback, 0);    //called every 100ms
	speed = 0.95 * speed;    //decreasing the speed by 5%
	diode_time += 1;//iteration the diode glowing variable
	if (diode_time > 14)     diode_time = 0;          //every 1500ms reseting it to 0
	ufoTime += 1;    //itarating
	if (ufoTime == 5 || ufoTime == 25)               ufoTime = 0;     // rover is moving/attacking
every 0.5s, and ever 2s if met rover
	if (timerMode == 1)              starsS = "Collected: " + to_string(starsCollected) + "/" +
to_string(numberOfStars);        //in first mode
	else            starsS = "Collected: " + to_string(starsCollected);      //in secound mode
	starsC = starsS.c_str(); //saving string in char array
}
```

W drugiej z nich jeżeli gra trwa, to zależnie od trybu gry zwiększany, lub zmniejszany jest czas, po czym zmieniane są wartości zmiennych z nim związanych. Jeżeli czas się kończy to zmieniana jest wartość flagi tak, aby zakończyć rundę:

```
void timerCallback2(int value)
{
	glutTimerFunc(10, timerCallback2, 0);    //called every 10ms
	if (endRound == 0 && pause == 0) //if round is in progress
	{
		if (timerMode == 1)              timer += 0.01;  //increasing time in first mode
		else timer -= 0.01;      //decreasing in secound mode
		timerstring = "Time: " + to_string(timer);       //saving text with timer
		timerChars = timerstring.c_str();        //saving string in char array
		if (abs(timer - min_time) < 0.01) endRound = 1;  //end round if user is out of time
	}
}
```

Ostatnią rzeczą do omówienia w kodzie jest tablica wyników. W funkcji *main* przed wejściem do funkcji głównej wywoływana jest funkcja *loadFromFile()*, w krórej wykonywane jest wczytanie wyników z pliku:

```
void loadFromFile()
{
	ifstream in;	//input stream
	in.open("results.txt");  //opening file
	for (int i = 0; i < 13; i++)
	{
		for (int j = 0; j < 4; j++)
		{
			in >> results[i][j];	//saving results in array
		}
	}
	in.close();	//closing file
}
```

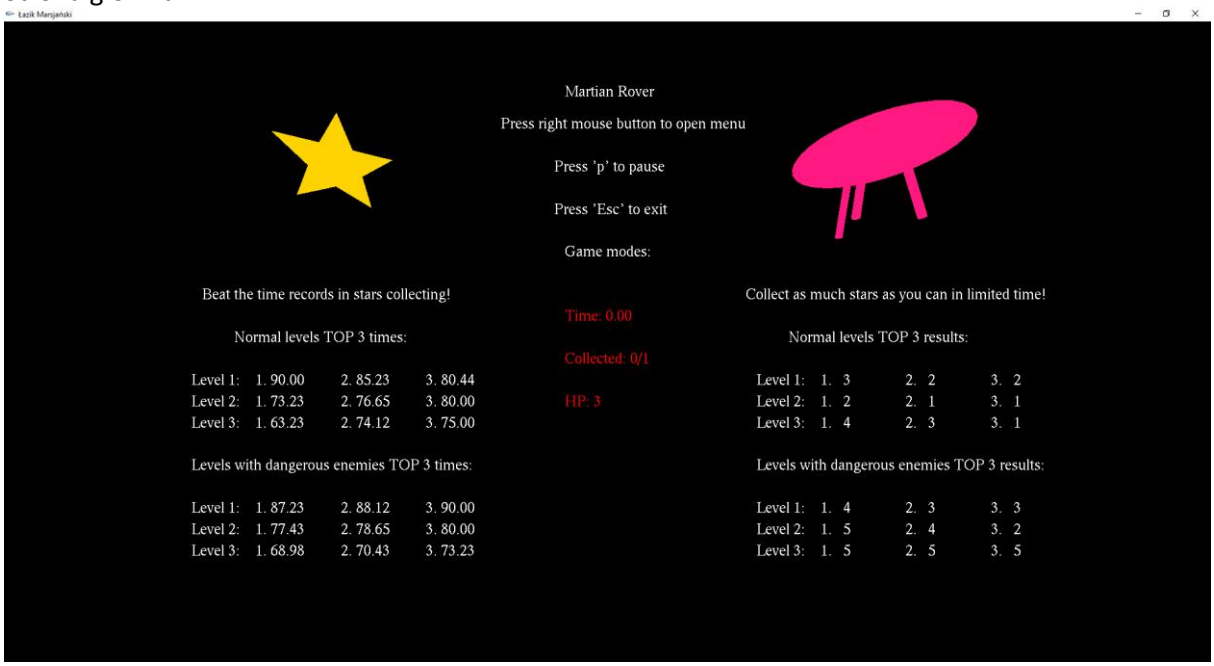
Podobnie działa funkcja zapisująca wyniki do pliku:

```
void saveToFile()
{
	ofstream out;	//output stream
	out.open("results.txt"); //opening file
	for (int i = 0; i < 13; i++)
	{
		for (int j = 0; j < 4; j++)
		{
			out << results[i][j];	//saving results in array
			out << " ";	//saving space in array, so floats doesnt stick
		}
	}
	out.close();	//closing file
}
```

Funkcja zapisująca dane wywoływana jest we wspomnianej wcześniej funkcji sortującej *sortResults*. Pozostałe zmiany w kodzie były niewielkie, raczej były dostosowaniem wartości tak, aby dało się wygodnie sterować i korzystać z programu – np.: zmiana prędkości, skrętu, itp.

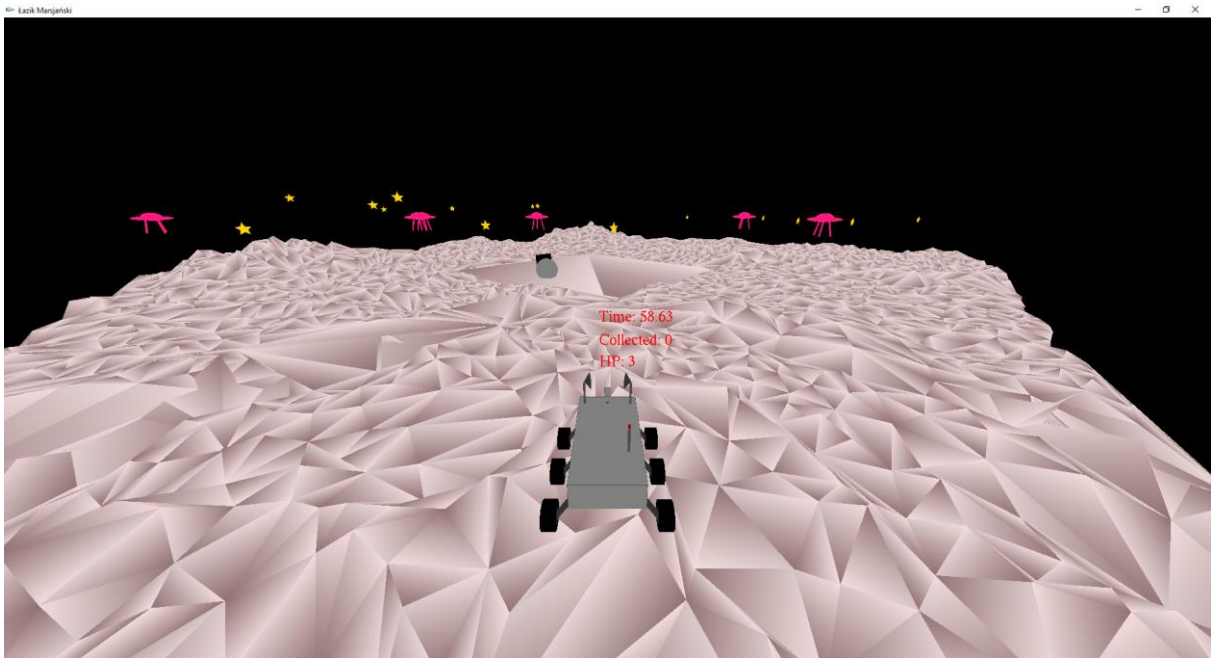
Poniżej został przedstawiony wygląd aplikacji po dodaniu wspomnianych elementów:

Strona główna:



Tryb pierwszy z włączonymi teksturami:

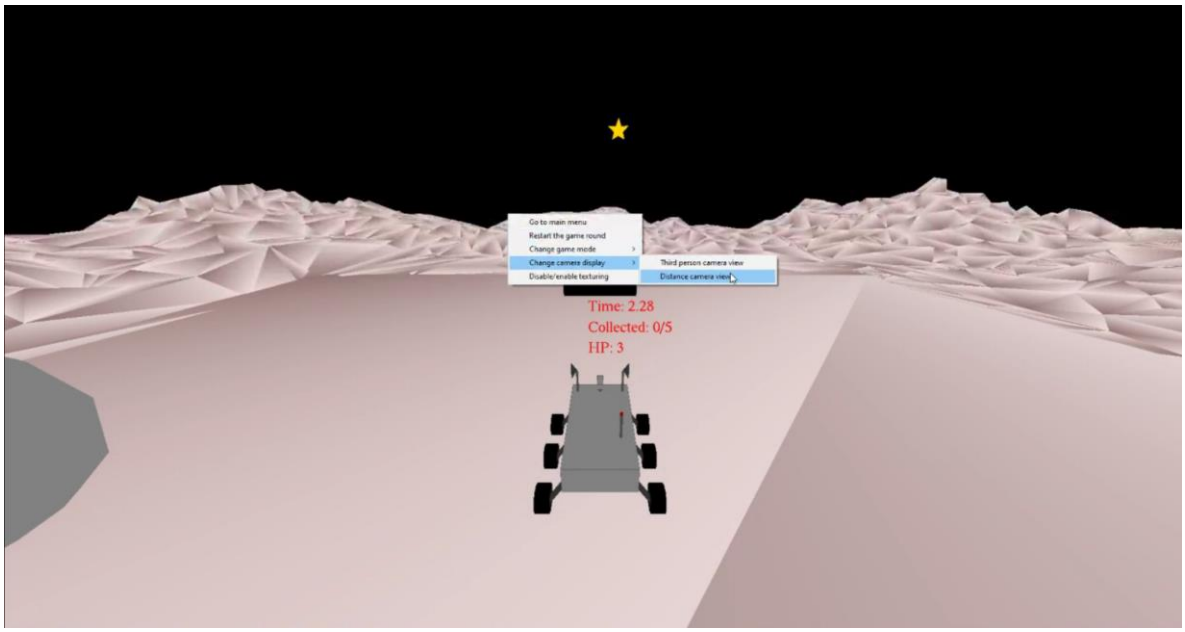
Tryb drugi z wyłączonymi teksturami:



Widok z dalekiej kamery:

Modele ufo i gwiazdy użyte w laboratorium:

Menu:



Do raportu zostaje dołączone wspomniane nagranie pokazujące, jak zmienia się działanie programu dzięki menu, widać także w nim rozgrywkę w dwóch trybach (tryb pierwszy, poziom normalny 1, oraz tryb drugi, agresywny 3) i w drugim z nich zapisanie wyniku do tablicy najlepszych 3 wyników. Zaprezentowane jest menu główne, kolizje na końcach map, kolizje z obiektami, jazda po nierównym podłożu, zbieranie gwiazd, zmiana kamery, zakończenie rozgrywki po zebraniu wszystkich gwiazd w trybie 1, oraz koniec gry spowodowany utratą wszystkich HP w trybie 2. Pokazane jest też przejście do strony głównej i kilkukrotne zresetowanie rozgrywki. Niestety na nagraniach nie widać menu, ponieważ nagrywane było okno, a menu jest tworzone, jako pop-up. Jednak łatwo zauważyć kiedy są zmieniane tryby kamery, tryby gry, resetowana gra, czy włączana strona główna. Poniżej znajduje się link do nagrania:

https://drive.google.com/file/d/1e1sAVy53wQhG-olmXcP9YyO_F6zwRbmN/view?usp=sharing