Proiect 3D - Sistem Solar

Membrii Echipei
Constantin Ioana Teodora - 341
Ionescu Radu - 341
Liviu Gheorghe Ionut - 344
Flutur Angelica - 341
Ion Melania-Victorita - 341

1. Conceptul Proiectului
   Conceptul proiectului se concentrează pe dezvoltarea unei simulări grafice a sistemului nostru solar. Acesta vizează modelarea corpilor cerești, incluzând Soarele, Luna și planetele majore precum Mercur, Venus, Pamant, Marte si Jupiter, într-un mediu 3D. Scopul este de a oferi o reprezentare vizuală detaliată a mișcărilor și caracteristicilor acestor corpuri, reflectând realitățile astronomice într-un cadru interactiv.

2. Originalitatea Proiectului
   Originalitatea acestui proiect constă în modul de utilizare a texturilor și detalii vizuale pentru a conferi un grad ridicat de realism. Corpuri ceresti, precum Soarele, Luna sau Jupiter, sunt reprezentate cu o atenție deosebită asupra detaliilor vizuale, cum ar fi texturile specifice și efectele de lumină. Aceste detalii nu doar îmbunătățesc aspectul estetic al proiectului, dar contribuie și la autenticitatea sa.

   Adiția unui obiect zburător neidentificat (OZN) aduce o notă de surpriză și intrigă, amplificând experiența și diversificând elementele vizuale ale proiectului. Această integrare subliniază angajamentul față de creativitate și originalitate, oferind utilizatorilor o experiență înedită în explorarea spațiului cosmic.

3. Elemente incluse

   - Desenarea obiectelor

   Pentru desenarea corpurilor cerești în proiect, a fost folosită o abordare uniformă, bazată pe utilizarea unei sfere ca formă de bază. Această sferă a fost scalată și translata diferit pentru a reprezenta diversele dimensiuni și poziții ale corpurilor cerești, cum ar fi planetele și Soarele.
   Coordonatele varfurilor sferei:

```
float u = U_MIN + parr * step_u;
float v = V_MIN + merid * step_v;
float x_vf = radius * cosf(u) * cosf(v);
float y_vf = radius * cosf(u) * sinf(v);
float z_vf = radius * sinf(u);
```

Exemplu de desenare al unei planete, avand punct de plecare sfera mentionata:

```
//JUPITER
matrTrans = glm::mat4(1.0f) * glm::translate(glm::mat4(1.0f), glm::vec3(120.0f, -920.0f, 0.0f));
matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(0.7f, 0.7f, 0.7f));
glUniform3f(objectColorLoc, 0.64f, 0.16f, 0.16f);
glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
myMatrix = matrTrans * matrScale;
DrawPlanet(myMatrix, 3);
DrawShadow(myMatrix);
```

- Texturarea
  Procesul de texturare în acest proiect este un aspect crucial pentru a conferi realism corpurilor cerești. Texturile sunt aplicate pe suprafața sferelor pentru a simula caracteristicile vizuale ale diferitelor corpuri, precum Soarele,Luna și planetele. Aplicarea texturii si calcularea coordonate de texturare se realizeaza in shadere pentru codCol = 2.

```
void LoadTexture(const char* photoPath)
{
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    //  Desfasurarea imaginii pe orizonatala/verticala in functie de parametrii de texturare;
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);


    int width, height;
    unsigned char* image = SOIL_load_image(photoPath, &width, &height, 0, SOIL_LOAD_RGB);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
    glGenerateMipmap(GL_TEXTURE_2D);

    SOIL_free_image_data(image);
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

```
if (codCol == 2)
{
    // Culoare cu ambient
    float ambientStrength = 1.0f;
    vec3 ambient_light = ambientStrength * lightColor;
    vec3 ambient_term= ambient_light * objectColor;
    // Culoarea finala
    vec3 emission = vec3(0.0,0.0,0.0);
    vec3 result = emission + (ambient_term);
    out_Color = vec4(result, 1.0f);
    out_Color = mix(texture(myTexture, tex_Coord), out_Color, 0.2);
}
```

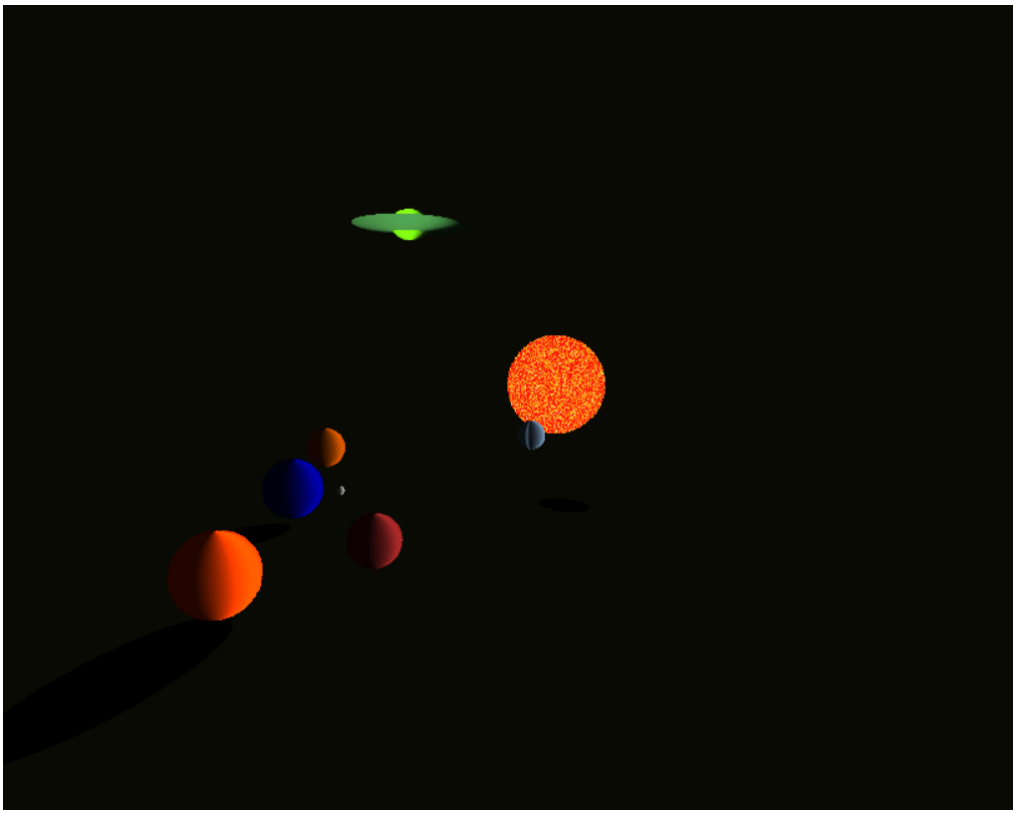
Exemplu pentru Venus:

```
// VENUS
codCol = 2;
LoadTexture("3_venus.jpg");
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
glUniform1i(glGetUniformLocation(ProgramIdf, "myTexture"), 0);
```

- Iluminare si Umbre
  În proiect, iluminarea și umbrele sunt gestionate utlizand shadere-le. Astfel, Soarele este iluminat utilizând doar lumina ambientală, indicată prin codCol = 2, iar pentru planete codCol = 0 si avem o combinație de lumini ambientale, difuze și speculare. In cele din urma, umbrele sunt desenate folosind codCol = 1 și funcția drawShadow().

- Elemente de dinamism:
  Într-un detaliu tehnic remarcabil, întregul sistem solar este proiectat să se miște într-o manieră precisă și coerentă, respectând principiile trigonometriei. Această abordare matematică aduce o autenticitate excepțională în reprezentarea dinamică a sistemului solar, având în vedere un sens trigonometric al mișcării.

- Aspect final:



- Contributii individuale:

Ionescu Radu - 341: dinamismul scenei, miscarea planetelor
Liviu Gheorghe Ionut - 344: forme, alegerea texturilor
Flutur Angelica - 341: realizarea texturarii
Ion Melania-Victorita - 341: redactarea documentatiei, iluminare
Constantin Ioana Teodora - 341: realizarea OZN-ului, adaugarea unei planetei

- Cod sursa:

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <GLFW/glfw3.h>
#include "loadShaders.h"
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtx/transform.hpp"
#include "glm/gtc/type_ptr.hpp"
#include "SOIL.h"
using namespace std;
GLuint
VaoId1, VaoId2,
VboId1, VboId2,
EboId1, EboId2,
ColorBufferId,
ProgramIdv,
ProgramIdf,
viewLocation,
projLocation,
codColLocation,
myMatrixLocation,
matrUmbraLocation,
depthLocation,
codCol;
GLint objectColorLoc, lightColorLoc, lightPosLoc, viewPosLoc;

GLuint texture;
float const PI = 3.141592f;
float const U_MIN = -PI / 2, U_MAX = PI / 2, V_MIN = 0, V_MAX = 2 * PI;
int const NR_PARR = 131, NR_MERID = 132;
float step_u = (U_MAX - U_MIN) / NR_PARR, step_v = (V_MAX - V_MIN) / NR_MERID;
float radius = 50;
int index, index_aux;
```

```
// variabile pentru matricea de vizualizare
float Obsx = 100.0, Obsy = -1500.0, Obsz = 200;
float Obsx1 = 100.0, Obsy1 = -1500.0, Obsz1 = 200;
float Refx = 0.0f, Refy = 1000.0f, Refz = 0.0f;
float Vx = 0.0, Vy = 0.0, Vz = 1.0;
float Vx1 = 0.0, Vy1 = 0.0, Vz1 = 1.0;

float sunRotationAngle = 0.0f; // Initialize the sun rotation angle

// variabile pentru matricea de proiectie
float width = 800, height = 600, znear = 0.1, fov = 45;
// matrice utilizate
glm::mat4 view, projection;
glm::mat4 myMatrix, matrTrans, matrScale, matrRotate;
// sursa de lumina
float xL = 0.0f, yL = 100.0f, zL = 250.0f;
// matricea umbrei
float matrUmbra[4][4];
void processNormalKeys(unsigned char key, int x, int y)
{
	switch (key) {
	case 'l':
		Vx -= 0.1;
		break;
	case 'r':
		Vx += 0.1;
		break;
	case '+':
		Obsy += 10;
		break;
	case '-':
		Obsy -= 10;
		break;
	}
	if (key == 27)
		exit(0);
}
void processSpecialKeys(int key, int xx, int yy)
{
	switch (key) {
	case GLUT_KEY_LEFT:
		Obsx -= 200;
		break;
	case GLUT_KEY_RIGHT:
		Obsx += 200;
		break;
	case GLUT_KEY_UP:
		Obsz += 200;
		break;
	case GLUT_KEY_DOWN:
```

```
                Obsz -= 200;
                break;
        }
}

void rotateAround() {

        Obsx -= 10;

        if (Obsx < -8000)
                Obsx = 8000;

        glutPostRedisplay();

}

void CreateVBO1(void)
{
        glm::vec4 Vertices1[(NR_PARR + 1) * NR_MERID];
        GLushort Indices1[2 * (NR_PARR + 1) * NR_MERID + 4 * (NR_PARR + 1) *
NR_MERID];
        for (int merid = 0; merid < NR_MERID; merid++)
        {
                for (int parr = 0; parr < NR_PARR + 1; parr++)
                {
                        float u = U_MIN + parr * step_u;
                        float v = V_MIN + merid * step_v;
                        float x_vf = radius * cosf(u) * cosf(v);
                        float y_vf = radius * cosf(u) * sinf(v);
                        float z_vf = radius * sinf(u);
                        index = merid * (NR_PARR + 1) + parr;
                        Vertices1[index] = glm::vec4(x_vf, y_vf, z_vf, 1.0);
                        Indices1[index] = index;
                        index_aux = parr * (NR_MERID)+merid;
                        Indices1[(NR_PARR + 1) * NR_MERID + index_aux] = index;
                        if ((parr + 1) % (NR_PARR + 1) != 0)
                        {
                                int AUX = 2 * (NR_PARR + 1) * NR_MERID;
                                int index1 = index;
                                int index2 = index + (NR_PARR + 1);
                                int index3 = index2 + 1;
                                int index4 = index + 1;
                                if (merid == NR_MERID - 1)
                                {
                                        index2 = index2 % (NR_PARR + 1);
                                        index3 = index3 % (NR_PARR + 1);
                                }
                                Indices1[AUX + 4 * index] = index1;
                                Indices1[AUX + 4 * index + 1] = index2;
                                Indices1[AUX + 4 * index + 2] = index3;
```

```
                                Indices1[AUX + 4 * index + 3] = index4;
                        }
                }
        }
        glGenVertexArrays(1, &VaoId1);
        glGenBuffers(1, &VboId1);
        glGenBuffers(1, &EboId1);
        glBindVertexArray(VaoId1);
        glBindBuffer(GL_ARRAY_BUFFER, VboId1);
        glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices1), Vertices1,
GL_STATIC_DRAW);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId1);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices1), Indices1,
GL_STATIC_DRAW);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 *
sizeof(GLfloat)));
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(3 *
sizeof(GLfloat)));
}
void DestroyVBO(void)
{
        glDisableVertexAttribArray(1);
        glDisableVertexAttribArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glDeleteBuffers(1, &VboId1);
        glDeleteBuffers(1, &EboId1);
        glBindVertexArray(0);
        glDeleteVertexArrays(1, &VaoId1);
}
void LoadTexture(void)
{
        glGenTextures(1, &texture);
        glBindTexture(GL_TEXTURE_2D, texture);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        int width, height;
        unsigned char* image = SOIL_load_image("sun.png", &width, &height, 0,
SOIL_LOAD_RGB);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, image);
        glGenerateMipmap(GL_TEXTURE_2D);
        SOIL_free_image_data(image);
        glBindTexture(GL_TEXTURE_2D, 0);
}
```

```
void CreateShadersFragment(void)
{
	ProgramIdf = LoadShaders("example.vert", "example.frag");
	glUseProgram(ProgramIdf);
}
void DestroyShaders(void)
{
	glDeleteProgram(ProgramIdf);
}
void Initialize(void)
{
	glClearColor(0.01, 0.01f, 0.01f, 0.f);
	CreateVBO1();
	CreateShadersFragment();
	objectColorLoc = glGetUniformLocation(ProgramIdf, "objectColor");
	lightColorLoc = glGetUniformLocation(ProgramIdf, "lightColor");
	lightPosLoc = glGetUniformLocation(ProgramIdf, "lightPos");
	viewPosLoc = glGetUniformLocation(ProgramIdf, "viewPos");
	viewLocation = glGetUniformLocation(ProgramIdf, "view");
	projLocation = glGetUniformLocation(ProgramIdf, "projection");
	myMatrixLocation = glGetUniformLocation(ProgramIdf, "myMatrix");
	matrUmbraLocation = glGetUniformLocation(ProgramIdf, "matrUmbra");
	codColLocation = glGetUniformLocation(ProgramIdf, "codCol");
	glUniform1i(glGetUniformLocation(ProgramIdf, "myTexture"), 0);
	LoadTexture();
	glActiveTexture(GL_TEXTURE0);
	glBindTexture(GL_TEXTURE_2D, texture);
	glUniform1i(glGetUniformLocation(ProgramIdf, "myTexture"), 0);

}
void DrawPlanet(glm::mat4 myMatrix, int codCol = 0)
{
	glBindVertexArray(VaoId1);
	glUniform1i(codColLocation, codCol);
	glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
	for (int patr = 0; patr < (NR_PARR + 1) * NR_MERID; patr++)
	{
		if ((patr + 1) % (NR_PARR + 1) != 0)
			glDrawElements(GL_TRIANGLES, 4, GL_UNSIGNED_SHORT,
(GLvoid*)((2 * (NR_PARR + 1)* (NR_MERID)+4 * patr) * sizeof(GLushort)));
	}
}
void DrawShadow(glm::mat4 myMatrix)
{
	glBindVertexArray(VaoId1);
	codCol = 1;
	glUniform1i(codColLocation, codCol);
	glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
	for (int patr = 0; patr < (NR_PARR + 1) * NR_MERID; patr++)
	{
```

```
                if ((patr + 1) % (NR_PARR + 1) != 0)
                        glDrawElements(GL_TRIANGLES, 4, GL_UNSIGNED_SHORT,
(GLvoid*)((2 * (NR_PARR + 1) * (NR_MERID)+4 * patr) * sizeof(GLushort)));
        }
}
void RenderFunction(void)
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glEnable(GL_DEPTH_TEST);
        // vizualizare + proiectie
        glm::vec3 Obs = glm::vec3(Obsx, Obsy, Obsz);
        glm::vec3 PctRef = glm::vec3(Refx, Refy, Refz);
        glm::vec3 Vert = glm::vec3(Vx, Vy, Vz);
        view = glm::lookAt(Obs, PctRef, Vert);
        glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &view[0][0]);
        projection = glm::infinitePerspective(fov, GLfloat(width) / GLfloat(height), znear);
        glUniformMatrix4fv(projLocation, 1, GL_FALSE, &projection[0][0]);
        // matricea pentru umbra
        float D = 100.0f;
        matrUmbra[0][0] = zL + D; matrUmbra[0][1] = 0; matrUmbra[0][2] = 0;
matrUmbra[0][3]= 0;
        matrUmbra[1][0] = 0; matrUmbra[1][1] = zL + D; matrUmbra[1][2] = 0;
matrUmbra[1][3]= 0;
        matrUmbra[2][0] = -xL; matrUmbra[2][1] = -yL; matrUmbra[2][2] = D; matrUmbra[2][3]
= -1;
        matrUmbra[3][0] = -D * xL; matrUmbra[3][1] = -D * yL; matrUmbra[3][2] = -D * zL;
matrUmbra[3][3] = zL;
        glUniformMatrix4fv(matrUmbraLocation, 1, GL_FALSE, &matrUmbra[0][0]);
        glUseProgram(ProgramIdf);
        // SOARELE
        matrTrans = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 1000.0f, 0.0f));
        matrRotate = glm::rotate(glm::mat4(1.0f), glm::radians(sunRotationAngle),
glm::vec3(0.0f, 1.0f, 0.0f));
        matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(3.f, 3.f, 3.f));
        glUniform3f(objectColorLoc, 1.0f, 0.4f, 0.2f);
        glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
        glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
        myMatrix = matrTrans * matrRotate * matrScale;
        DrawPlanet(myMatrix, 2);
        // MERCUR
        matrTrans = glm::translate(glm::mat4(1.0f), glm::vec3(180.f, -100.f, 0.f));
        matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f, 0.5f, 0.5f));
        glUniform3f(objectColorLoc, 0.45f, 0.57f, 0.7f);
        glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
        glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
        glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
        myMatrix = matrTrans * matrScale;
        DrawPlanet(myMatrix);
        DrawShadow(myMatrix);
        // VENUS
```

```
        matrTrans = glm::mat4(1.0f) * glm::translate(glm::mat4(1.0f), glm::vec3(-100.0f,
-300.0f, 0.0f));
        matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(0.6f, 0.6f, 0.6f));
        glUniform3f(objectColorLoc, 0.8f, 0.33f, 0.0f);
        glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
        glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
        glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
        myMatrix = matrTrans * matrScale;
        DrawPlanet(myMatrix);
        DrawShadow(myMatrix);
        // PAMANT
        matrTrans = glm::mat4(1.0f) * glm::translate(glm::mat4(1.0f), glm::vec3(10.0f, -600.0f,
0.0f));
        matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(0.7f, 0.7f, 0.7f));
        glUniform3f(objectColorLoc, 0.0f, 0.0f, 0.63f);
        glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
        glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
        glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
        myMatrix = matrTrans * matrScale;
        DrawPlanet(myMatrix);
        DrawShadow(myMatrix);
        // LUNA
        matrTrans = glm::mat4(1.0f) * glm::translate(glm::mat4(1.0f), glm::vec3(70.0f, -600.0f,
0.0f));
        matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(0.1f, 0.1f, 0.1f));
        glUniform3f(objectColorLoc, 0.7f, 0.7f, 0.7f);
        glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
        glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
        glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
        myMatrix = matrTrans * matrScale;
        DrawPlanet(myMatrix);
        // MARTE
        matrTrans = glm::mat4(1.0f) * glm::translate(glm::mat4(1.0f), glm::vec3(200.0f,
-800.0f, 0.0f));
        matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f, 0.5f, 0.5f));
        glUniform3f(objectColorLoc, 0.64f, 0.16f, 0.16f);
        glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
        glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
        glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
        myMatrix = matrTrans * matrScale;
        DrawPlanet(myMatrix);

        //JUPITER
        matrTrans = glm::mat4(1.0f) * glm::translate(glm::mat4(1.0f), glm::vec3(120.0f,
-920.0f, 0.0f));
        matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(0.7f, 0.7f, 0.7f));
        glUniform3f(objectColorLoc, 0.64f, 0.16f, 0.16f);
        glUniform3f(lightColorLoc, 2.2f, 2.2f, 0.0f);
        glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
        glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
```

```
	myMatrix = matrTrans * matrScale;
	DrawPlanet(myMatrix);
	DrawShadow(myMatrix);


	//UFO

	matrTrans = glm::mat4(1.0f) * glm::translate(glm::mat4(1.0f), glm::vec3(-500.0f,
1100.0f, 500.0f));
	matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(1.2f, 1.7f, 1.0f));
	glUniform3f(objectColorLoc, 0.5f, 1.0f, 0.5f);
	glUniform3f(lightColorLoc, 1.2f, 1.2f, 0.2f);
	glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
	glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
	myMatrix = matrTrans * matrScale;
	DrawPlanet(myMatrix);
	DrawShadow(myMatrix);

	matrTrans = glm::mat4(1.0f) * glm::translate(glm::mat4(1.0f), glm::vec3(-500.0f,
1100.0f, 500.0f));
	matrScale = glm::scale(glm::mat4(1.0f), glm::vec3(3.5f, 4.5f, 0.5f));
	glUniform3f(objectColorLoc, 0.5f, 1.0f, 0.5f);
	glUniform3f(lightColorLoc, 0.7f, 0.7f, 0.7f);
	glUniform3f(lightPosLoc, 0.f, 1000.f, 0.f);
	glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);
	myMatrix = matrTrans * matrScale;
	DrawPlanet(myMatrix);
	rotateAround();

	glutSwapBuffers();
	glFlush();
}
void Cleanup(void)
{
	DestroyShaders();
	DestroyVBO();
}
int main(int argc, char* argv[])
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
	glutInitWindowPosition(100, 10);
	glutInitWindowSize(1000, 750);
	glutCreateWindow("Sistemul Solar");
	glewInit();
	Initialize();
	glutIdleFunc(RenderFunction);
	glutDisplayFunc(RenderFunction);
	glutKeyboardFunc(processNormalKeys);
	glutSpecialFunc(processSpecialKeys);
```

```
	glutAttachMenu(GLUT_RIGHT_BUTTON);
	glutCloseFunc(Cleanup);
	glutMainLoop();
}
```

- Shader-ul de fragment:

```
#version 330 core


in vec3 FragPos;
in vec3 Normal;
in vec3 inLightPos;
in vec3 inViewPos;
in vec2 tex_Coord;
out vec4 out_Color;
uniform vec3 objectColor;
uniform vec3 lightColor;
uniform int codCol;
uniform sampler2D myTexture;
void main(void)
{
	if (codCol==0)
	{
		// Ambient
		float ambientStrength = 0.1f;
		vec3 ambient_light = ambientStrength * lightColor;
		vec3 ambient_term= ambient_light * objectColor;
		// Diffuse
		vec3 norm = normalize(Normal);
		vec3 lightDir = normalize(inLightPos - FragPos);
		float diff = max(dot(norm, lightDir), 0.0);
		vec3 diffuse_light = lightColor;
		vec3 diffuse_term = diff * diffuse_light * objectColor;
		// Specular
		float specularStrength = 0.8f;
		float shininess = 100.0f;
		vec3 viewDir = normalize(inViewPos - FragPos);
		vec3 reflectDir = normalize(reflect(-lightDir, norm));
		float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
		vec3 specular_light = specularStrength * lightColor;
		vec3 specular_term = spec * specular_light * objectColor;
		// Culoarea finala
		vec3 emission=vec3(0.0, 0.0, 0.0);
		vec3 result = emission + (ambient_term + diffuse_term + specular_term);
		out_Color = vec4(result, 1.0f);
	}
```

```
	if (codCol==1)
	{
		vec3 black = vec3(0.0, 0.0, 0.0);
		out_Color = vec4(black, 1.0);
	}
	if (codCol==2)
	{
		// Culoare cu ambient
		float ambientStrength = 1.0f;
		vec3 ambient_light = ambientStrength * lightColor;
		vec3 ambient_term= ambient_light * objectColor;
		// Culoarea finala
		vec3 emission = vec3(0.0,0.0,0.0);
		vec3 result = emission + (ambient_term);
		out_Color = vec4(result, 1.0f);
		out_Color = mix(texture(myTexture, tex_Coord), out_Color, 0.2);
	}
}
```

- Vertex shader:

```
#version 330 core

layout(location=0) in vec3 in_Position;
layout(location=1) in vec3 in_Normal;
layout(location=2) in vec2 texCoord;
out vec4 gl_Position;
out vec3 Normal;
out vec3 FragPos;
out vec3 inLightPos;
out vec3 inViewPos;
out vec2 tex_Coord;
uniform mat4 matrUmbra;
uniform mat4 myMatrix;
uniform mat4 view;
uniform mat4 projection;
uniform vec3 lightPos;
uniform vec3 viewPos;
uniform int codCol;
void main(void)
{
	 if (codCol == 0)
	 {
		gl_Position = projection*view*myMatrix*vec4(in_Position, 1.0);
		Normal=vec3(projection*view*vec4(in_Normal,0.0));
		inLightPos= vec3(projection*view*vec4(lightPos, 1.0f));
		inViewPos=vec3(projection*view*vec4(viewPos, 1.0f));
```

```
                FragPos = vec3(gl_Position);
        }
        if (codCol == 1)
        {
                gl_Position = projection*view*matrUmbra*myMatrix*vec4(in_Position, 1.0);
                FragPos = vec3(gl_Position);
        }
        if (codCol == 2)
        {
                gl_Position = projection*view*myMatrix*vec4(in_Position, 1.0);
                Normal=vec3(projection*view*vec4(in_Normal,0.0));
                tex_Coord = vec2(Normal.x, Normal.y);
        }
}
```