Coiculescu Veronica-Ionela
Grupa 341

Documentație Proiect

I.Conceptul Proiectului

În proiectul realizat am reprezentat o scenă 3D care cuprinde mai mulți copaci de diferite dimensiuni. Am integrat elemente precum: obiecte 3D(cilindru, con), iluminare, umbre, efect de ceață.

II.

1. **Reprezentarea obiectelor 3D**
   Pentru a reprezenta suprafețele de rotație, m-am folosit de ecuațiile parametrice ale acestora:

```
//reprezentare con
float u = U_MIN + parr * step_u; // valori pentru u si v
float v = V_MIN + merid * step_v;
float x_vf = v * cosf(u); // coordonatele varfului corespunzator lui (u,v)
float y_vf = v * sinf(u);
float z_vf = v;
```

```
// reprezentare cilindru
float u = U_MIN_cil + parr * step_u_cil;
float v = V_MIN_cil + merid * step_v_cil;
float x_vf = radius * cosf(u); // coordonatele varfului corespunzator lui (u,v)
float y_vf = radius * sinf(u);
float z_vf = v;
```

Pentru fiecare vârf am reținut coordonatele, culoarea, normala, iar in vectorul *Indices* am stocat indicii. Am desenat mai multe instante ale acestor obiecte, folosind o matrice de translatie pentru a le plasa în diverse locuri(myMatrix este o variabilă uniformă transmisă în shader și folosita la compunerea transformarilor). Am declarat doi vectori pentru a varia pozitia:
float coef_i[] = { 2.75, 5.5, -7.8,-11.6, 14.8 };
float coef_j[] = {2.5, -5.75, 7.8,-11.6,14.5 };

```
c1 = coef_i[i];
c2 = coef_j[j];
if (j % 2 != 0 || i % 2 == 0)
    m = 1;
else
    m = 2;
myMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(4.5*c1 *k* 80, 4.5*c2 *k* 80, (m-1)*150.0));
myMatrixLocation = glGetUniformLocation(ProgramId, "myMatrix");
glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
```

2. **Iluminare, umbre și efectul de ceață**

   Pentru iluminarea suprafetelor am folosit Modelul Phong, făcând calculele necesare în Shadere. În programul principal mi-am declarat poziția sursei de lumină, apoi în Shaderul de fragment am variat parametrii pentru a lua in considerare componentele ambientala, difuza.

Coiculescu Veronica-Ionela
Grupa 341

```
// Ambient
float ambientStrength = 0.2f;
vec3 ambient = ambientStrength * lightColor;

// Diffuse
vec3 normala = normalize(Normal);
vec3 lightDir = normalize(inLightPos - FragPos);
float diff = max(dot(normala, lightDir), 0.0);
vec3 diffuse = diff * lightColor;

// Specular
float specularStrength = 0.5f;
vec3 viewDir = normalize(inViewPos - FragPos);//vector catre observator normalizat (V)
vec3 reflectDir = reflect(-lightDir, normala); // reflexia razei de lumina (R)
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 1);
vec3 specular = specularStrength * spec * lightColor;
vec3 emission=vec3(0.0, 0.0, 0.0);
vec3 result = emission+(ambient + diffuse + specular) * ex_Color;
```

Pentru reprezentarea umbrei, am declarat matricea pentru umbra, proiectand umbrele obiectelor in planul de ecuatie z=-304.

```
// matricea pentru umbra
float D = 304;
//proiectie pe planul z=-304
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;
```

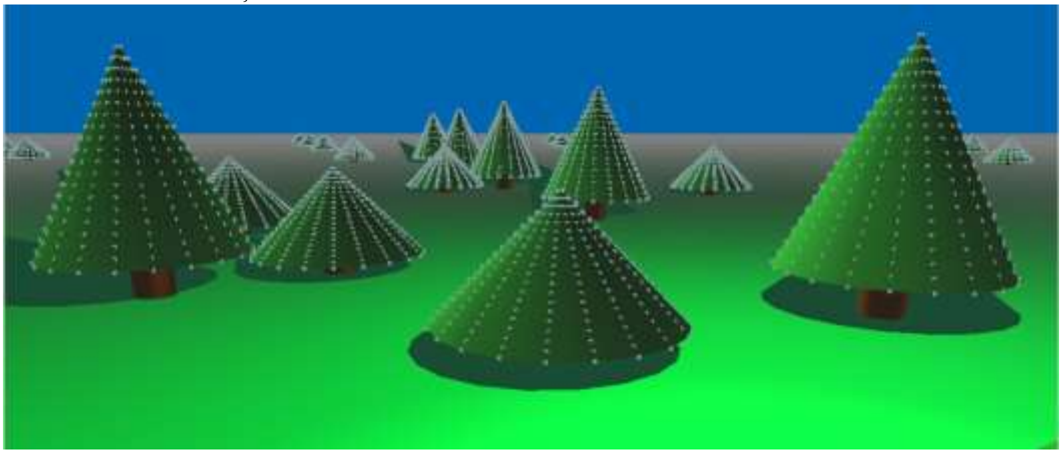
În funcție de codul de culoare trimis în Shader, dacă codCol este 1, înseamnă că se dorește reprezentarea unei umbre, deci vom lua in considerare matrUmbra. In caz contrar, nu ne vom folosi de aceasta.

```
if ( codCol==1 )
    gl_Position = projection*view*matrUmbra*myMatrixScale*myMatrix*in_Position;
    FragPos = vec3(gl_Position);
```

Iar pentru efectul de ceață am calculat in shaderul de fragment:

```
// Efect de ceata
vec3 fogColor = vec3(0.5, 0.5, 0.5);
float dist=length(inViewPos - FragPos);
float fogFactor=exp(-0.0001*dist); // intre 0 si 1; 1 corespunde aproape de obiect
out_Color = vec4(mix(fogColor,result,fogFactor), 1.0f);
```

III. Rezultatul obținut:



IV. Completări față de etapa de prezentare:

Am folosit un factor de scalare pentru a obtine copaci de diferite dimesiuni.

Coiculescu Veronica-Ionela
Grupa 341

Anexe

Proiect.cpp

```
#include <windows.h>  // biblioteci care urmeaza sa fie incluse
#include <stdlib.h> // necesare pentru citirea shader-elor
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <GL/glew.h> // glew apare inainte de freeglut
#include <GL/freeglut.h> // nu trebuie uitat freeglut.h

#include "loadShaders.h"

#include "glm/glm/glm.hpp"
#include "glm/glm/gtc/matrix_transform.hpp"
#include "glm/glm/gtx/transform.hpp"
#include "glm/glm/gtc/type_ptr.hpp"


using namespace std;


// identificatori
GLuint
VaoId,
VboId,
EboId,
ColorBufferId,
ProgramId,
myMatrixLocation,
myMatrixLocationScale,
matrUmbraLocation,
viewLocation,
projLocation,
matrRotlLocation,
codColLocation,
depthLocation;

GLuint texture;


float PI = 3.141592;
```

```
// Elemente pentru reprezentarea suprafetei
float const U_MIN = 0, U_MAX = PI * 2, V_MIN = -250, V_MAX = 10,
V_MIN_cil = -304, V_MAX_cil = -190, U_MIN_cil = 0, U_MAX_cil = 2 *
PI;
float const U_MIN_r = 0, U_MAX_r = PI * 2, V_MIN_r = 0, V_MAX_r = 90;
int const  NR_PARR = 20, NR_MERID = 20;
float step_u = (U_MAX - U_MIN) / NR_PARR, step_v = (V_MAX - V_MIN)
/ NR_MERID, step_v_cil = (V_MAX_cil - V_MIN_cil) /
NR_MERID,step_u_cil = (U_MAX_cil - U_MIN_cil) / NR_PARR;
float step_u_r = (U_MAX_r - U_MIN_r) / NR_PARR, step_v_r= (V_MAX_r -
V_MIN_r) / NR_MERID;
// alte variabile
int codCol, aux, aux2,aux3,aux4;
float radius = 50;
int index, index_aux;
// matrice utilizate
glm::mat4 myMatrix, matrRot, myMatrixScale;
// elemente pentru matricea de vizualizare
float Refx = 1500.0f, Refy = 1500.0f, Refz = -300.0f;
float alpha = PI / 8, beta = 0.0f, dist = 1000.0f;
float Obsx, Obsy, Obsz;

float Vx = 0.0, Vy = 0.0, Vz = 1.0;
glm::mat4 view;

// elemente pentru matricea de proiectie
float width = 800, height = 600, xwmin = -800.f, xwmax = 800, ywmin = -600,
ywmax = 600, znear = 0.3, zfar = 1, fov = 45;
glm::mat4 projection;

// sursa de lumina
float xL =1000.f, yL = 1000.0f, zL =  500.0f;

// 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 '+':
			dist += 5;
			break;
		case '-':
			dist -= 5;
			break;

		}
		if (key == 27)
			exit(0);
	}
	void processSpecialKeys(int key, int xx, int yy) {

		switch (key) {
		case GLUT_KEY_LEFT:
			beta -= 0.01;
			break;
		case GLUT_KEY_RIGHT:
			beta += 0.01;
			break;
		case GLUT_KEY_UP:
			alpha += 0.01;
			break;
		case GLUT_KEY_DOWN:
			alpha -= 0.01;
			break;
		}
	}

	void CreateVBO(void)
	{
		glm::vec4 Vertices[3*(NR_PARR + 1) * NR_MERID + 5];
		glm::vec3 Colors[3*(NR_PARR + 1) * NR_MERID + 5];
		glm::vec3 Normals[3*(NR_PARR + 1) * NR_MERID + 5];
		GLushort Indices[3* (2 * (NR_PARR + 1) * NR_MERID + 4 *
	(NR_PARR + 1) * NR_MERID)+14];


		for (int merid = 0; merid < NR_MERID; merid++)
		{
```

```
			for (int parr = 0; parr < NR_PARR + 1; parr++)
			{

				//reprezentare con
				float u = U_MIN + parr * step_u; // valori pentru u si v
				float v = V_MIN + merid * step_v;
				float x_vf = v * cosf(u); // coordonatele varfului
	corespunzator lui (u,v)
				float y_vf = v * sinf(u);
				float z_vf = v;

				index = merid * (NR_PARR + 1) + parr;
				Vertices[index] = glm::vec4(x_vf, y_vf, z_vf, 1.0);
				Colors[index] = glm::vec3(0.1,0.5, 0.1);

				Normals[index] = glm::vec3(x_vf, y_vf, -z_vf); // normala
	la suprafata conului
				Indices[index] = index;
				index_aux = parr * (NR_MERID)+merid;
				Indices[(NR_PARR + 1) * NR_MERID + index_aux] =
	index;

				if ((parr + 1) % (NR_PARR + 1) != 0) // varful considerat
	sa nu fie Polul Nord
				{
					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);
					}
					Indices[AUX + 4 * index] = index1;
					Indices[AUX + 4 * index + 1] = index2;
					Indices[AUX + 4 * index + 2] = index3;
					Indices[AUX + 4 * index + 3] = index4;
				}
			}
		}
```

```
aux2 = (NR_PARR + 1) * NR_MERID;
aux = 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++)
            {

                // reprezentare cilindru
                float u = U_MIN_cil + parr * step_u_cil;
                float v = V_MIN_cil + merid * step_v_cil;
                float x_vf = radius * cosf(u); // coordonatele
varfului corespunzator lui (u,v)
                float y_vf = radius * sinf(u);
                float z_vf = v;


                index = merid * (NR_PARR + 1) + parr;
                Vertices[index + aux2] = glm::vec4(x_vf,
y_vf, z_vf, 1.0);
                Colors[index + aux2] = glm::vec3(0.5,0.2,
0);
                Normals[index + aux2] = glm::vec3(x_vf,
y_vf, 0);
                Indices[index+aux] = index+aux2;
                index_aux = parr * (NR_MERID)+merid;
                Indices[(NR_PARR + 1) * NR_MERID +
index_aux+aux] = index+aux2;
                if ((parr + 1) % (NR_PARR + 1) != 0) //
varful considerat sa nu fie Polul Nord
                {
                    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);
                    }
                    Indices[AUX + 4 * index+aux] =
index1+aux2;
                    Indices[AUX + 4 * index + 1+aux] =
index2+aux2;
                    Indices[AUX + 4 * index + 2+aux] =
index3+aux2;
                    Indices[AUX + 4 * index + 3+aux] =
index4+aux2;
                }
            }
        }




    aux3 = 2* (2 * (NR_PARR + 1) * NR_MERID + 4 * (NR_PARR + 1) *
NR_MERID);
    aux4 = 2 * (NR_PARR + 1) * NR_MERID;
    int k = 100;
    Vertices[aux4] = glm::vec4(-k*1500.0f, -k*1500.0f, -305.0f, 1.0f);
    Vertices[aux4 + 1] = glm::vec4(k*1500.0f, -k*1500.0f, -305.0f, 1.0f);
    Vertices[aux4 + 2] = glm::vec4(k*1500.0f, k*1500.0f, -305.0f, 1.0f);
    Vertices[aux4 + 3] = glm::vec4(-k*1500.0f, k*1500.0f, -305.0f, 1.0f);

    Colors[aux4] = glm::vec3(0.0f, 0.9, 0.2f);
    Colors[aux4 + 1] = glm::vec3(0.0f, 1.0f, 0.0f);
    Colors[aux4 + 2] = glm::vec3(0.0f, 0.8f, 0.2f);
    Colors[aux4 + 3] = glm::vec3(0.0f, 1.0f, 0.0f);

    Normals[aux4] = glm::vec3(0.0f, 0.0f, 1.0f);
    Normals[aux4 + 1] = glm::vec3(0.0f, 0.0f, 1.0f);
    Normals[aux4 + 2] = glm::vec3(0.0f, 0.0f, 1.0f);
    Normals[aux4 + 3] = glm::vec3(0.0f, 0.0f, 1.0f);



    Indices[aux3] = aux4 + 1; Indices[aux3 + 1] = aux4 + 2; Indices[aux3 +
2] = aux4;
    Indices[aux3 + 3] = aux4 + 2; Indices[aux3 + 4] = aux4; Indices[aux3 +
5] = aux4 + 3;
```

```
	Indices[aux3+6] = aux4; Indices[aux3 + 7] = aux4 + 4; Indices[aux3 + 8] = aux4+1;
	Indices[aux3 + 9] = aux4 + 4; Indices[aux3 + 10] = aux4+2; Indices[aux3 + 11] = aux4 + 4;
	Indices[aux3 + 12] = aux4; Indices[aux3 + 13] = aux4 + 4;


	// generare VAO/buffere
	glGenBuffers(1, &VboId); // atribute
	glGenBuffers(1, &EboId); // indici

	// legare+"incarcare" buffer
	glBindBuffer(GL_ARRAY_BUFFER, VboId);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId);
	glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices) + sizeof(Colors) + sizeof(Normals), NULL, GL_STATIC_DRAW);
	glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(Vertices), Vertices);
	glBufferSubData(GL_ARRAY_BUFFER, sizeof(Vertices), sizeof(Colors), Colors);
	glBufferSubData(GL_ARRAY_BUFFER, sizeof(Vertices) + sizeof(Colors), sizeof(Normals), Normals);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices), Indices, GL_STATIC_DRAW);

	// atributele;
	glEnableVertexAttribArray(0); // atributul 0 = pozitie
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
	glEnableVertexAttribArray(1); // atributul 1 = culoare
	glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)sizeof(Vertices));
	glEnableVertexAttribArray(2); // atributul 2 = normala
	glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)(sizeof(Vertices) + sizeof(Colors)));


}
void DestroyVBO(void)
{
	glDisableVertexAttribArray(2);
	glDisableVertexAttribArray(1);
	glDisableVertexAttribArray(0);

	glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
		glDeleteBuffers(1, &VboId);
		glDeleteBuffers(1, &EboId);

		glBindVertexArray(0);
		glDeleteVertexArrays(1, &VaoId);
	}


	void CreateShaders(void)
	{
		ProgramId = LoadShaders("11_02_Shader.vert", "11_02_Shader.frag");
		glUseProgram(ProgramId);
	}

	void DestroyShaders(void)
	{
		glDeleteProgram(ProgramId);
	}

	void Initialize(void)
	{

		glClearColor(0.0f, 0.4f, 0.7f, 0.0f); // culoarea de fond a ecranului
		CreateShaders();

	}
	void RenderFunction(void)
	{

		glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
		glEnable(GL_DEPTH_TEST);

		//pozitia observatorului
		Obsx = Refx + dist * cos(alpha) * cos(beta);
		Obsy = Refy + dist * cos(alpha) * sin(beta);
		Obsz = Refz + dist * sin(alpha);

		// reperul de vizualizare
		glm::vec3 Obs = glm::vec3(Obsx, Obsy, Obsz);   // se schimba pozitia
	observatorului
		glm::vec3 PctRef = glm::vec3(Refx, Refy, Refz); // pozitia punctului de
	referinta
```

```
	glm::vec3 Vert = glm::vec3(Vx, Vy, Vz); // verticala din planul de
vizualizare
	view = glm::lookAt(Obs, PctRef, Vert);

	projection = glm::infinitePerspective(fov, GLfloat(width) /
GLfloat(height), znear);
	myMatrix = glm::mat4(1.0f);
	myMatrixScale = glm::mat4(1.0f);

	// matricea pentru umbra
	float D = 304;
	//proiectie pe planul z=-304
	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;

	CreateVBO();

	// variabile uniforme pentru shaderul de varfuri
	myMatrixLocation = glGetUniformLocation(ProgramId, "myMatrix");
	glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE,
&myMatrix[0][0]);
	myMatrixLocationScale = glGetUniformLocation(ProgramId,
"myMatrixScale");
	glUniformMatrix4fv(myMatrixLocationScale, 1, GL_FALSE,
&myMatrix[0][0]);
	matrUmbraLocation = glGetUniformLocation(ProgramId,
"matrUmbra");
	glUniformMatrix4fv(matrUmbraLocation, 1, GL_FALSE,
&matrUmbra[0][0]);
	viewLocation = glGetUniformLocation(ProgramId, "view");
	glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &view[0][0]);
	projLocation = glGetUniformLocation(ProgramId, "projection");
	glUniformMatrix4fv(projLocation, 1, GL_FALSE, &projection[0][0]);

	// Variabile uniforme pentru iluminare
	GLint lightColorLoc = glGetUniformLocation(ProgramId,
"lightColor");
```

```
	GLint lightPosLoc = glGetUniformLocation(ProgramId, "lightPos");
	GLint viewPosLoc = glGetUniformLocation(ProgramId, "viewPos");
	GLint codColLocation = glGetUniformLocation(ProgramId, "codCol");
	glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
	glUniform3f(lightPosLoc, xL, yL, zL);
	glUniform3f(viewPosLoc, Obsx, Obsy, Obsz);







	codCol = 0;
	glUniform1i(codColLocation, codCol);
	glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT,
(GLvoid*)(aux3 * sizeof(GLushort)));



	int c1, c2;
	float coef_i[] = { 2.75, 5.5, -7.8,-11.6, 14.8 };
	float coef_j[] = {2.5, -5.75, 7.8,-11.6,14.5 };
	int m;
	for (int k = 0; k <= 5; k++)
	{

		for (int j = 0; j <= 4; j++)
		{
			for (int i = 0; i <= 4; i++)
			{
				c1 = coef_i[i];
				c2 = coef_j[j];
				if (j % 2 != 0 || i % 2 == 0)
					m = 1;
				else
					m = 2;
				myMatrix = glm::translate(glm::mat4(1.0f),
glm::vec3(4.5*c1 *k* 80, 4.5*c2 *k* 80, (m-1)*150.0));
				myMatrixLocation =
glGetUniformLocation(ProgramId, "myMatrix");
				glUniformMatrix4fv(myMatrixLocation, 1,
GL_FALSE, &myMatrix[0][0]);

				myMatrixScale = glm::scale(glm::mat4(1.0f),
glm::vec3(1.0, 1.0, m));
```

```
			myMatrixLocationScale = 
glGetUniformLocation(ProgramId, "myMatrixScale");
			glUniformMatrix4fv(myMatrixLocationScale, 1, 
GL_FALSE, &myMatrixScale[0][0]);

			codCol = 0;
			glUniform1i(codColLocation, codCol);
			for (int patr = 0; patr < (NR_PARR + 1) * 
NR_MERID; patr++)
			{
				if ((patr + 1) % (NR_PARR + 1) != 0)
					glDrawElements(GL_QUADS, 4, 
GL_UNSIGNED_SHORT, (GLvoid*)((2 * (NR_PARR + 1) * 
(NR_MERID)+4 * patr) * sizeof(GLushort)));
			}

			codCol = 3;
			glUniform1i(codColLocation, codCol);
			glPointSize(5.0);
			glDrawElements(GL_POINTS, (NR_PARR + 1) * 
NR_MERID, GL_UNSIGNED_SHORT, 0);

			//-----desenare cilindru

			aux = 2 * (NR_PARR + 1) * NR_MERID + 4 * 
(NR_PARR + 1) * NR_MERID;
			codCol = 0;
			glUniform1i(codColLocation, codCol);
			for (int patr = 0; patr < (NR_PARR + 1) * 
NR_MERID; patr++)
			{
				if ((patr + 1) % (NR_PARR + 1) != 0) // nu 
sunt considerate fetele in care in stanga jos este Polul Nord
					glDrawElements(GL_QUADS, 4, 
GL_UNSIGNED_SHORT, (GLvoid*)((aux + 2 * (NR_PARR + 1) * 
(NR_MERID)+4 * patr) * sizeof(GLushort)));
			}

			// desenare umbra con
			codCol = 1;
			glUniform1i(codColLocation, codCol);
```

```
                        for (int patr = 0; patr < (NR_PARR + 1) *
NR_MERID; patr++)
                        {
                                if ((patr + 1) % (NR_PARR + 1) != 0)
                                        glDrawElements(GL_QUADS, 4,
GL_UNSIGNED_SHORT, (GLvoid*)((2 * (NR_PARR + 1) *
(NR_MERID)+4 * patr) * sizeof(GLushort)));
                        }

                        // desenare umbra cilindru
                        aux = 2 * (NR_PARR + 1) * NR_MERID + 4 *
(NR_PARR + 1) * NR_MERID;

                        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_QUADS, 4,
GL_UNSIGNED_SHORT, (GLvoid*)((aux + 2 * (NR_PARR + 1) *
(NR_MERID)+4 * patr) * sizeof(GLushort)));
                        }

                }
        }
    }

    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, 100);
        glutInitWindowSize(1200, 900);
        glutCreateWindow("Implementarea modelului de iluminare");
        glewInit();
        Initialize();
        glutIdleFunc(RenderFunction);
        glutDisplayFunc(RenderFunction);
        glutKeyboardFunc(processNormalKeys);
        glutSpecialFunc(processSpecialKeys);


        glutCloseFunc(Cleanup);
        glutMainLoop();

    }
```

11_02_Shader.vert

```
// Shader-ul de varfuri

 #version 400


layout(location=0) in vec4 in_Position;
layout(location=1) in vec3 in_Color;
layout(location=2) in vec3 in_Normal;


out vec3 FragPos;
out vec3 Normal;
out vec3 inLightPos;
out vec3 inViewPos;
out vec3 ex_Color;

uniform mat4 matrUmbra;
uniform mat4 myMatrix;
uniform mat4 view;
uniform mat4 projection;
uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;
uniform int codCol;
uniform mat4 myMatrixScale;


void main(void)
```

```
  {
    ex_Color=in_Color;
        if ( codCol==0 || codCol==2|| codCol == 3 || codCol == 4)
    {
                gl_Position = projection*view*myMatrixScale*myMatrix*in_Position;
        Normal=mat3(projection*view*myMatrix)*in_Normal;
        inLightPos= vec3(projection*view*myMatrix* vec4(lightPos, 1.0f));
        inViewPos=vec3(projection*view*myMatrix*vec4(viewPos, 1.0f));
        FragPos = vec3(gl_Position);
    }
        if ( codCol==1 )
                gl_Position =
projection*view*matrUmbra*myMatrixScale*myMatrix*in_Position;
        FragPos = vec3(gl_Position);


   }
```

11_02_Shader.frag

```
// Shader-ul de fragment / Fragment shader

 #version 400

in vec3 FragPos;
in vec3 Normal;
in vec3 inLightPos;
in vec3 inViewPos;
in vec3 ex_Color;

out vec4 out_Color;

uniform vec3 lightColor;
uniform int codCol;

void main(void)
  {
        // Ambient
    float ambientStrength = 0.2f;
    vec3 ambient = ambientStrength * lightColor;

    // Diffuse
    vec3 normala = normalize(Normal);
    vec3 lightDir = normalize(inLightPos - FragPos);
    float diff = max(dot(normala, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    // Specular
    float specularStrength = 0.5f;
```

```
    vec3 viewDir = normalize(inViewPos - FragPos);//vector catre observator normalizat (V)
    vec3 reflectDir = reflect(-lightDir, normala); // reflexia razei de lumina (R)
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 1);
    vec3 specular = specularStrength * spec * lightColor;
    vec3 emission=vec3(0.0, 0.0, 0.0);
    vec3 result = emission+(ambient + diffuse + specular) * ex_Color;


    // Efect de ceata
    vec3 fogColor = vec3(0.5, 0.5, 0.5);
    float dist=length(inViewPos - FragPos);
    float fogFactor=exp(-0.0001*dist); // intre 0 si 1; 1 corespunde aproape de obiect
    out_Color = vec4(mix(fogColor,result,fogFactor), 1.0f);

    if ( codCol==1 )
                    out_Color=vec4 (0.1, 0.3, 0.2, 0.0);
    if ( codCol==2 )
        out_Color=vec4 (0.5, 0.5, 0.0, 0.0);
    if ( codCol==3 )
        out_Color=vec4 (0.5, 0.6, 0.6, 0.0);




  }
```