**Documentație Proiect**

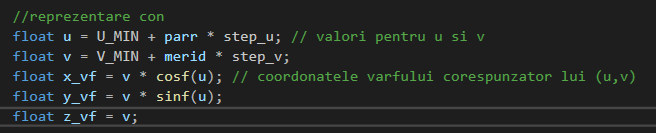
**I.Conceptul Proiectului**

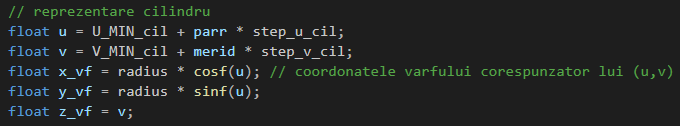
În proiectul realizat am reprezentat o scenă 3D care cuprinde mai mulți copaci de diferite dimensiuni. Am integrat elemente precum: obiecte 3D( cilindru, con), iluminare, umbre, efect de ceață.

**II.**

1. **Reprezentarea obiectelor 3D**

Pentru a reprezenta suprafețele de rotație, m-am folosit de ecuațiile parametrice ale acestora:

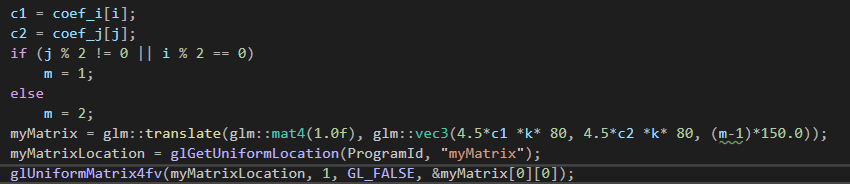




Pentru fiecare vârf am reținut coordonatele, culoarea, normala, iar in vectorul *Indices* am stocat indicii. Am desenat mai multe instante ale acestor obiecte, folosind o matrice de translatie pentru a le plasa în diverse locuri( myMatrix este o variabilă uniformă transmisă în shader și folosita la compunerea transformarilor). Am declarat doi vectori pentru a varia pozitia:

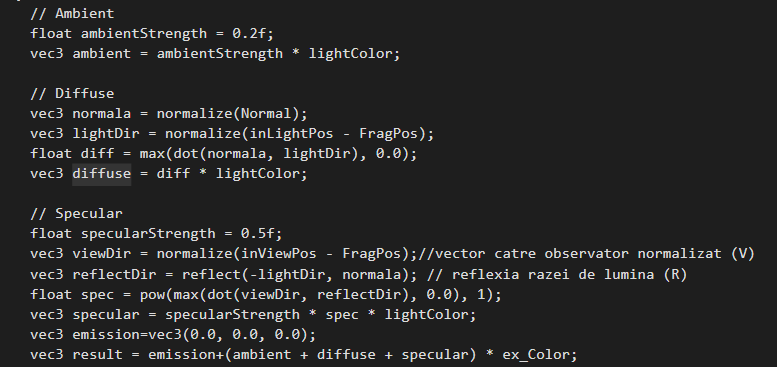
float coef\_i[] = { 2.75, 5.5, -7.8,-11.6, 14.8 };

float coef\_j[] = {2.5, -5.75, 7.8,-11.6,14.5 };

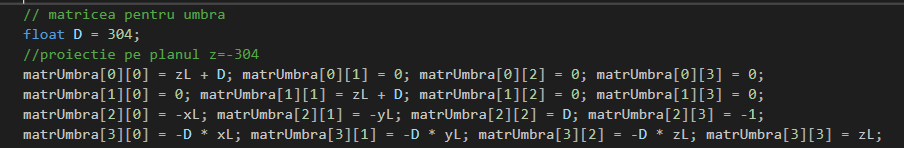


1. **Iluminare, umbre și efectul de ceață**

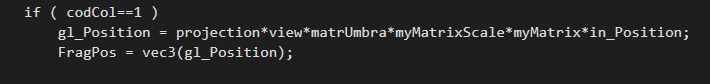
Pentru iluminarea suprafetelor am folosit Modelul Phong, făcând calculele necesare în Shadere. În programul principal mi-am declarat poziția sursei de lumină, apoi în Shaderul de fragment am variat parametrii pentru a lua in considerare componentele ambientala, difuza.



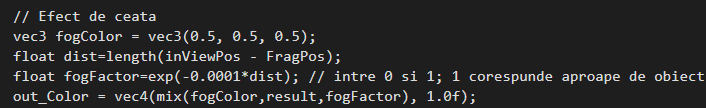
Pentru reprezentarea umbrei, am declarat matricea pentru umbra, proiectand umbrele obiectelor in planul de ecuatie z=-304.

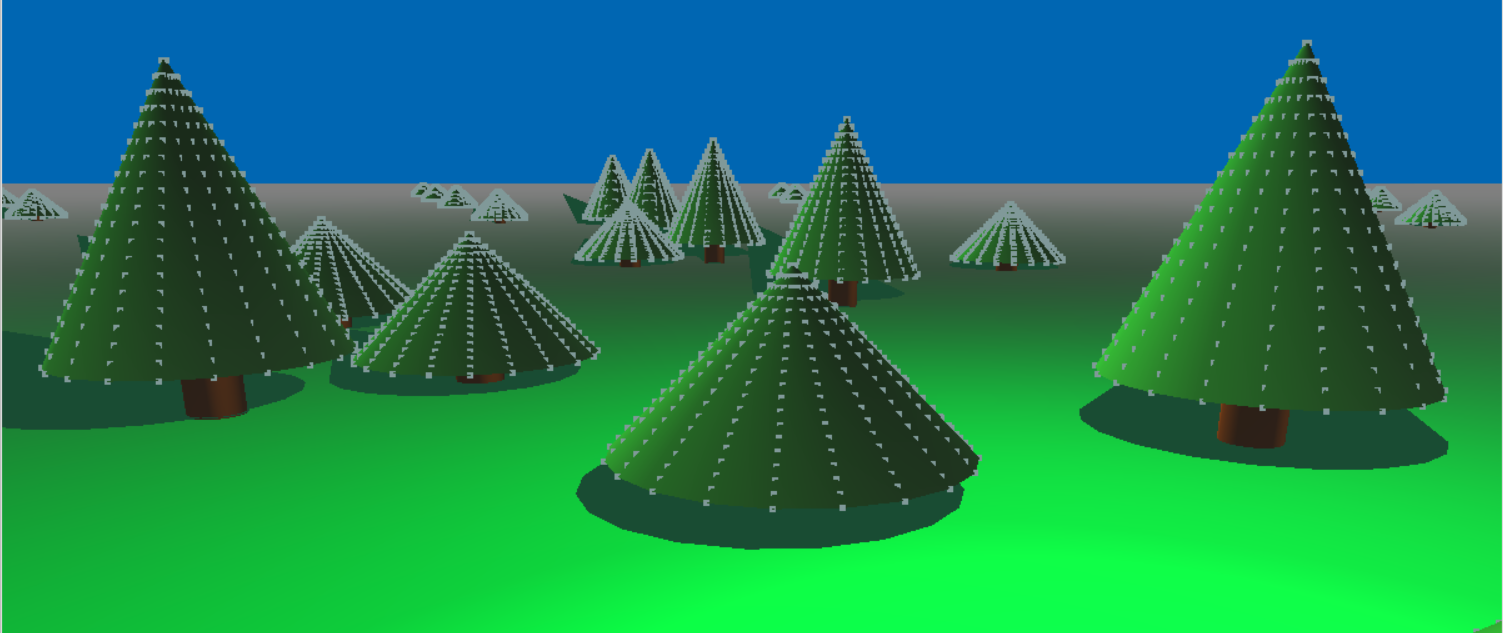


În funcție de codul de culoare trimis în Shader, dacă codCol este 1, înseamnă că se dorește reprezentarea unei umbre, deci vom lua in considerare matrUmbra. In caz contrar, nu ne vom folosi de aceasta.



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



III. Rezultatul obținut:

IV. Completări față de etapa de prezentare:

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

**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);

}