UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Proiect Grafică

3D CAR SCENE

Student

Mincu Adrian-Lucian

Rezumat

Acest proiect are ca subiect principal o mașină amplasată în centrul unei priveliști de munte. Întreaga scenă are un efect stilizat de tip desen animat, creând o experiență vizuală vibrantă. Un efect de parallax sporește percepția adâncimii, făcând scena mai dinamică.

Inspirația mea a venit din: Firewatch Fan Art [1].

Cuprins

Capitolul 1

Documentatie

1.1 Originalitate

Acest proiect aduce originalitate prin stilul său. Combinația dintre modele low-poly pentru obiectele de fundal și modele mai detaliate pentru obiectele din prim-plan, împreună cu estetica de tip desen animat și efectul de parallax, creează o scenă vizual dinamică.

Utilizarea specială a nivelurilor variate de detaliu ghidează privirea spectatorului, atrăgând atenția asupra elementelor cheie, în timp ce menține un aspect captivant.

Această combinație creativă de efecte conferă unicitate scenei.

1.2 Elemente incluse

1.2.1 Focalizare centrală

Focalizare Centrală: Mașina servește drept punct focal al scenei, fiind înconjurată de un peisaj montan pitoresc.

1.2.2 Estetică de tip desen animat

Stil de Desen Animat: Scena adoptă o estetică de tip desen animat, cu culori vii și puternice, precum și forme simplificate. Acest efect a fost obținut prin setarea unei valori ridicate pentru intensitatea ambianței.

Listing 1.1: main.cpp

```
// ambient strength
glUniform1f(ambientStrengthLocation, 1.0f);
```

Listing 1.2: shader.frag

```
uniform float ambientStrength;

void main(void)
{
    switch (codCol) {
        case 0:
            // Ambient
            float _ambientStrength = ambientStrength;
            vec3 ambient = _ambientStrength * lightColor;
        // ...
    }
}
```

1.2.3 Efect parallax

Efectul de Parallax: Adaugă un sentiment de profunzime scenei prin simularea diferitelor viteze de mișcare pentru elementele din prim-plan și fundal. Acest efect poate fi obținut prin stratificarea scenei pe baza elementelor de prim-plan și fundal.

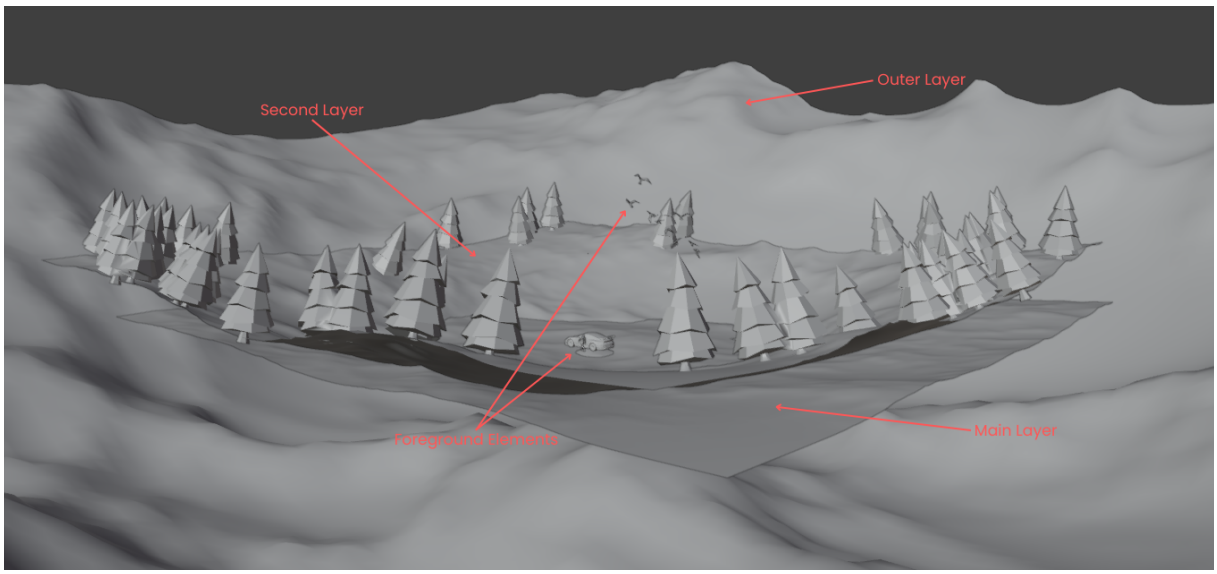

Figura 1.1: Ilustrarea metodei pentru obținerea efectului de parallax: stratificarea scenei în elemente de prim-plan și fundal pentru a simula profunzimea prin viteze diferite de mișcare.

1.2.4 Proiectarea umbrei

Proiecția Umbrelor: Umbrele mașinii și ale pilotului sunt proiectate pe sol utilizând matricea de proiecție a umbrelor specificată.

Listing 1.3: main.cpp

```
float D = -0.5f;
shadowMatrix[0][0] = zL + D;
shadowMatrix[0][1] = 0;
shadowMatrix[0][2] = 0;
shadowMatrix[0][3] = 0;

shadowMatrix[1][0] = 0;
shadowMatrix[1][1] = zL + D;
shadowMatrix[1][2] = 0;
shadowMatrix[1][3] = 0;

shadowMatrix[2][0] = -xL;
shadowMatrix[2][1] = -yL;
shadowMatrix[2][2] = D;
shadowMatrix[2][3] = -1;

shadowMatrix[3][0] = -D * xL;
shadowMatrix[3][1] = -D * yL;
shadowMatrix[3][2] = -D * zL;
shadowMatrix[3][3] = zL;
```

1.2.5 Iluminare

Sursă de Lumină: O sursă de lumină a fost adăugată scenei pentru a îmbunătăți vizibilitatea efectelor de umbră și pentru a evidenția detaliile fine ale elementelor din prim-plan.

Figura 1.2: Ilustrarea efectului de iluminare: evidențierea detaliilor fine și a umbrelor proiectate prin adăugarea unei surse de lumină în scenă.

Listing 1.4: shader.frag

```
#version 330 core

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

out vec4 out_Color;

uniform vec3 lightColor;
uniform int codCol;
uniform float ambientStrength;
```

```
void main(void)
{
    switch (codCol) {
        case 0:
            // Ambient
            float _ambientStrength = ambientStrength;
            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);
            vec3 reflectDir = reflect(-lightDir, normala);
            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;
            out_Color = vec4(result, 1.0f);

            break;

        case 1:
            vec3 black = vec3 (0.0, 0.0, 0.0);
            out_Color = vec4 (black, 1.0);
    }
}
```

1.3 Blender

Aplicația 3D Blender a fost utilizată pentru a crea întreaga scenă. Ulterior, fiecare obiect a fost exportat ca fișier .obj și redat folosind OpenGL.

1.3.1 Peisaje parametrice

Peisajele au fost generate cu ajutorul addon-ului **A.N.T Landscape** din Blender. Ulterior, un modificator de tip wave a fost aplicat pentru a obține forma sferică.

1.3.2 OBJs

- Mașină [1]
- Copac [2]
- Păsări [3]
- Pilot [4]

1.3.3 Sistem de particule

Sistemul de particule din Blender a fost utilizat pentru a genera mai mulți copaci pe stratul mijlociu din peisaj.

Demonstrație Sistem de Particule

O textură de tip weighted paint a fost aplicată pentru a crea o mască ce definește locațiile unde sunt generați copacii. Această metodă asigură că arborii sunt plasați exclusiv pe vârfurile marcate cu roșu în mască.

Demonstrație Weighted Paint

[1]Resursa preluată de pe Sketchfab: 2018 Porsche 911 GT2 RS Weissach Package
[2]Resursa preluată de pe Sketchfab: Low Poly Tree Concept
[3]Resursa preluată de pe Sketchfab: Birds
[4]Resursa preluată de pe Mixamo: Racer

Capitolul 2

Cod

Listing 2.1: Tot codul

```
// main.cpp
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <vector>

#include <GL/glew.h>
#include <GL/freeglut.h>
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtx/transform.hpp"
#include "glm/gtc/type_ptr.hpp"

#include "loadShaders.h"
#include "objloader.hpp"


GLuint
        VaoId,
        VboId,
        EboId,
        ProgramId,
        modelMatrixLocation,
        shadowMatrixLocation,
        viewLocation,
        projLocation,
        rotationMatrixLocation,
        lightColorLocation,
        lightPosLocation,
        viewPosLocation,
        codColLocation,
        colorLocation,
        ambientStrengthLocation;


GLuint VaoIdGround, VaoIdMiddle, VaoIdOuter, VaoIdBirds, VaoIdCar, VaoIdRacer;
GLuint VboIdGround, VboIdMiddle, VboIdOuter, VboIdBirds, VboIdCar, VboIdRacer;
```

```
int codCol;
float PI = 3.141592;

// matrices
glm::mat4 modelMatrix, rotationMatrix;
glm::vec3 color;

// elements for view matrix
float Refx = 0.0f, Refy = 0.0f, Refz = 0.0f;
float alpha = PI / 16, beta = 0.0f, dist = 470.0f;
float Obsx, Obsy, Obsz;
float Vx = 0.0, Vy = 0.0, Vz = 500.0;
glm::mat4 view;

// elements for projection matrix
float width = 800, height = 600, xwmin = -800.f, xwmax = 800, ywmin = -600,
      ywmax = 600, znear = 0.1, zfar = 1, fov = 45, deltaY = ywmax - ywmin;
glm::mat4 projection;
// light source
float xL = -400.f, yL = -400.f, zL = 400.f;
// shadow matrix
float shadowMatrix[4][4];


std::vector<glm::vec3> verticesGround, verticesMiddle, verticesOuter,
                       verticesBirds, verticesRacer, verticesCar;
std::vector<glm::vec2> uvsGround, uvsMiddle, uvsOuter, uvsBirds,
                       uvsRacer, uvsCar;
std::vector<glm::vec3> normalsGround, normalsMiddle, normalsOuter,
                       normalsBirds, normalsRacer, normalsCar;


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;
        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)
{
        GLfloat Vertices[] =
        {
                // ground vertices
           -1500.0f,  -1500.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
                1500.0f,  -1500.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
                1500.0f,  1500.0f,  0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
           -1500.0f,  1500.0f,  0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
        };

        // vertices index
        GLubyte Indices[] =
        {
                 // ground faces
                 1, 2, 0,   2, 0, 3,
        };

        glGenVertexArrays(1, &VaoId);
        glGenBuffers(1, &VboId);
        glGenBuffers(1, &EboId);
        glBindVertexArray(VaoId);

        glBindBuffer(GL_ARRAY_BUFFER, VboId);
        glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices), Indices, GL_STATIC_DRAW);

        // 0: positions
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 7 * sizeof(GLfloat), (GLvoid*)0);
        // 1: normals
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(
        1,
        3,
        GL_FLOAT,
        GL_FALSE,
        7 * sizeof(GLfloat),
        (GLvoid*)(4 * sizeof(GLfloat))
    );
```

```
}


void CreateVboObj(
    GLuint &VaoIdFunction,
    GLuint &VboIdFunction,
    std::vector<glm::vec3> &verticesFunction,
    std::vector<glm::vec3> &normalsFunction
)
{
        glGenVertexArrays(1, &VaoIdFunction);
        glBindVertexArray(VaoIdFunction);

        glGenBuffers(1, &VboIdFunction);
        glBindBuffer(GL_ARRAY_BUFFER, VboIdFunction);
        glBufferData(
        GL_ARRAY_BUFFER,
        verticesFunction.size() * sizeof(glm::vec3) + normalsFunction.size() * sizeof(glm::vec3),
        NULL, GL_STATIC_DRAW
    );
        glBufferSubData(
        GL_ARRAY_BUFFER,
        0,
        verticesFunction.size() * sizeof(glm::vec3), &verticesFunction[0]
    );
        glBufferSubData(
        GL_ARRAY_BUFFER,
        verticesFunction.size() * sizeof(glm::vec3),
        normalsFunction.size() * sizeof(glm::vec3), &normalsFunction[0]
    );

        glEnableVertexAttribArray(0);
        glVertexAttribPointer(
        0,
        3,
        GL_FLOAT,
        GL_FALSE,
        0,
        (GLvoid*)0
    );
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(
        1,
        3,
        GL_FLOAT,
        GL_FALSE,
        3 * sizeof(GLfloat),
        (GLvoid*)(verticesFunction.size() * sizeof(glm::vec3))
    );
}

void CreateVboObjs(void)
{
        // ground.obj
        CreateVboObj(VaoIdGround, VboIdGround, verticesGround, normalsGround);

        // middle.obj
        CreateVboObj(VaoIdMiddle, VboIdMiddle, verticesMiddle, normalsMiddle);
```

```
        // outer.obj
        CreateVboObj(VaoIdOuter, VboIdOuter, verticesOuter, normalsOuter);

        // birds.obj
        CreateVboObj(VaoIdBirds, VboIdBirds, verticesBirds, normalsBirds);

        // car.obj
        CreateVboObj(VaoIdCar, VboIdCar, verticesCar, normalsCar);

        // racer.obj
        CreateVboObj(VaoIdRacer, VboIdRacer, verticesRacer, normalsRacer);
}


void DestroyVBO(void)
{
        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("shader.vert", "shader.frag");
        glUseProgram(ProgramId);
}


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


void Initialize(void)
{
        modelMatrix = glm::mat4(1.0f);
        rotationMatrix = glm::rotate(glm::mat4(1.0f), PI / 8, glm::vec3(0.0, 0.0, 1.0));

        loadOBJ("objs/ground.obj", verticesGround, uvsGround, normalsGround);
        loadOBJ("objs/middle.obj", verticesMiddle, uvsMiddle, normalsMiddle);
        loadOBJ("objs/outer.obj", verticesOuter, uvsOuter, normalsOuter);
        loadOBJ("objs/birds.obj", verticesBirds, uvsBirds, normalsBirds);
        loadOBJ("objs/racer.obj", verticesRacer, uvsRacer, normalsRacer);
        loadOBJ("objs/car.obj", verticesCar, uvsCar, normalsCar);

        glClearColor(1.0f, 0.85f, 0.75f, 0.0f);

        CreateVBO();
        CreateVboObjs();
        CreateShaders();

        // shader locations
        modelMatrixLocation = glGetUniformLocation(ProgramId, "myMatrix");
```

```
        shadowMatrixLocation = glGetUniformLocation(ProgramId, "matrUmbra");
        viewLocation = glGetUniformLocation(ProgramId, "view");
        projLocation = glGetUniformLocation(ProgramId, "projection");
        lightColorLocation = glGetUniformLocation(ProgramId, "lightColor");
        lightPosLocation = glGetUniformLocation(ProgramId, "lightPos");
        viewPosLocation = glGetUniformLocation(ProgramId, "viewPos");
        codColLocation = glGetUniformLocation(ProgramId, "codCol");
        colorLocation = glGetUniformLocation(ProgramId, "color");
        ambientStrengthLocation = glGetUniformLocation(ProgramId, "ambientStrength");
}


void RenderFunction(void)
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glEnable(GL_DEPTH_TEST);

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

        // view + projection
        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]);

        // shadow matrix
        float D = -0.5f;
        shadowMatrix[0][0] = zL + D;
    shadowMatrix[0][1] = 0;
    shadowMatrix[0][2] = 0;
    shadowMatrix[0][3] = 0;

        shadowMatrix[1][0] = 0;
    shadowMatrix[1][1] = zL + D;
    shadowMatrix[1][2] = 0;
    shadowMatrix[1][3] = 0;

        shadowMatrix[2][0] = -xL;
    shadowMatrix[2][1] = -yL;
    shadowMatrix[2][2] = D;
    shadowMatrix[2][3] = -1;

        shadowMatrix[3][0] = -D * xL;
    shadowMatrix[3][1] = -D * yL;
    shadowMatrix[3][2] = -D * zL;
    shadowMatrix[3][3] = zL;
        glUniformMatrix4fv(shadowMatrixLocation, 1, GL_FALSE, &shadowMatrix[0][0]);

        // light variables
        glUniform3f(lightColorLocation, 1.0f, 1.0f, 1.0f);
        glUniform3f(lightPosLocation, xL, yL, zL);
```

```
glUniform3f(viewPosLocation, Obsx, Obsy, Obsz);

// ground
glBindVertexArray(VaoId);

codCol = 0;
glUniform1i(codColLocation, codCol);

// ambient strength
glUniform1f(ambientStrengthLocation, 1.0f);

modelMatrix = glm::mat4(1.0f);
glUniformMatrix4fv(modelMatrixLocation, 1, GL_FALSE, &modelMatrix[0][0]);

color = glm::vec3(0.205f, 0.041f, 0.103f);
glUniform3fv(colorLocation, 1, &color[0]);

glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);

// scale matrix
modelMatrix = glm::mat4(1.0f) * glm::scale(glm::mat4(1.0f), glm::vec3(deltaY, deltaY, deltaY));
glUniformMatrix4fv(modelMatrixLocation, 1, GL_FALSE, &modelMatrix[0][0]);

// ground.obj
glBindVertexArray(VaoIdGround);

color = glm::vec3(0.165f, 0.001f, 0.063f);
glUniform3fv(colorLocation, 1, &color[0]);

glDrawArrays(GL_TRIANGLES, 0, verticesGround.size());

// middle.obj
glBindVertexArray(VaoIdMiddle);

color = glm::vec3(1.0f, 0.037f, 0.091f);
glUniform3fv(colorLocation, 1, &color[0]);

glDrawArrays(GL_TRIANGLES, 0, verticesMiddle.size());

// ambient strength
glUniform1f(ambientStrengthLocation, 1.25f);

// outer.obj
glBindVertexArray(VaoIdOuter);

color = glm::vec3(1.0f, 0.301f, 0.105f);
glUniform3fv(colorLocation, 1, &color[0]);

glDrawArrays(GL_TRIANGLES, 0, verticesOuter.size());

// ambient strength
glUniform1f(ambientStrengthLocation, 1.0f);

// racer.obj
glBindVertexArray(VaoIdRacer);

glDrawArrays(GL_TRIANGLES, 0, verticesRacer.size());

// racer shadow
```

```
        codCol = 1;
        glUniform1i(codColLocation, codCol);
        glDrawArrays(GL_TRIANGLES, 0, verticesRacer.size());

        codCol = 0;
        glUniform1i(codColLocation, codCol);

        // ambient strength
        glUniform1f(ambientStrengthLocation, 1.0f);

        // birds.obj
        glBindVertexArray(VaoIdBirds);

        color = glm::vec3(0.018f, 0.000392f, 0.008531f);
        glUniform3fv(colorLocation, 1, &color[0]);

        glDrawArrays(GL_TRIANGLES, 0, verticesBirds.size());

        // ambient strength
        glUniform1f(ambientStrengthLocation, 0.25f);

        // car.obj
        glBindVertexArray(VaoIdCar);

        color = glm::vec3(1.0f, 0.037f, 0.091f);
        glUniform3fv(colorLocation, 1, &color[0]);

        glDrawArrays(GL_TRIANGLES, 0, verticesCar.size());

        // car shadow
        codCol = 1;
        glUniform1i(codColLocation, codCol);
        glDrawArrays(GL_TRIANGLES, 0, verticesCar.size());

        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("3D Car Scene");

        glewInit();
        Initialize();

        glutIdleFunc(RenderFunction);
        glutDisplayFunc(RenderFunction);
```

```
        glutKeyboardFunc(processNormalKeys);
        glutSpecialFunc(processSpecialKeys);
        glutCloseFunc(Cleanup);

        glutMainLoop();
}

// shader.frag
#version 330 core

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

out vec4 out_Color;

uniform vec3 lightColor;
uniform int codCol;
uniform float ambientStrength;

void main(void)
{
    switch (codCol) {
        case 0:
            // Ambient
            float _ambientStrength = ambientStrength;
            vec3 ambient = _ambientStrength * lightColor;

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

            // Specular
            float specularStrength = 0.5f;
            vec3 viewDir = normalize(inViewPos - FragPos);
            vec3 reflectDir = reflect(-lightDir, normala);
            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;
            out_Color = vec4(result, 1.0f);

            break;

        case 1:
            vec3 black = vec3 (0.0, 0.0, 0.0);
            out_Color = vec4 (black, 1.0);
    }
}

// shader.vert
#version 330 core
```

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

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

uniform mat4 matrUmbra;

uniform mat4 myMatrix;
uniform mat4 view;
uniform vec3 viewPos;

uniform mat4 projection;

uniform vec3 lightPos;
uniform vec3 lightColor;

uniform int codCol;
uniform vec3 color;

void main(void)
{
    ex_Color = color;

    switch (codCol) {
        case 0:
            gl_Position = projection * view * 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));
            dir = mat3(projection * view * myMatrix) * vec3(0.0,100.0,200.0);
            FragPos = vec3(gl_Position);

            break;
        case 1:
            gl_Position = projection * view * matrUmbra * myMatrix * in_Position;
            FragPos = vec3(gl_Position);

            break;
    }
}
```

Bibliografie

[1] tzeshi, „Firewatch Fan Art”, în *Sketchfab* (2015), URL: https://sketchfab.com/3d-models/firewatch-fan-art-8609caf1cd8c452eb7b6d4ca4228fcd0.