# Proiect 2 - Grafică pe Calculator

- Plimbare prin tunel -

Popescu Paullo Robertto Karloss
Linte Robert Ovidiu
Grupa 331

Conf. Dr. Stupariu Mihai-Sorin

- 21 Ianuarie 2023 -

# Cuprins

# Introducere

## 1.1 Modul de organizare al echipei

Echipa noastră:

- Popescu Paullo Robertto Karloss
- Linte Robert Ovidiu

Se va specifica **numele** membrului care a **contribuit** la realizarea fiecărei *etape*.

## 1.2 Obiectivele proiectului

Simularea unei "plimbări printr-un tunel":

- O cameră va avansa printr-un tunel închis
- Pereții din tunel vor fi texturați, folosind *normal* și *specular* maps pentru a putea simula căderea luminii pe aceștia mai bine
- Pe parcurs vom întâlnii mai multe obstacole (mai multi cilindrii)
- Acestea vor fi poziționate fie pe verticală fie pe orizontală prin translații
- Acesti cilindrii vor proiecta umbre pe pereții laterali, dar și pe podeaua tunelului
- Vom aplica ceața

Aprofundarea cunoștințelor în OpenGL prin:

- Utilizarea texturării
- Utilizarea umbrelor
- Utilizarea cetei
- Desenarea obiectelor
- Utilizarea culorilor

## 1.3 Vizionarea proiectului

Puteți viziona *demo-ul* proiectului ***aici***, iar *repository-ul* de pe Github ***aici***.

# Desenarea obiectelor

Această etapă a fost realizată de ***Popescu Paullo Robertto Karloss***.

## 2.1 Prezentarea tunelului

Tunelul este alcătuit din:

- Mai mulți cilindrii
- 5 pereți dintre care doi sunt pereții în sine, unul este tavanul, unul este podeaua, iar unul este capătul tunelului

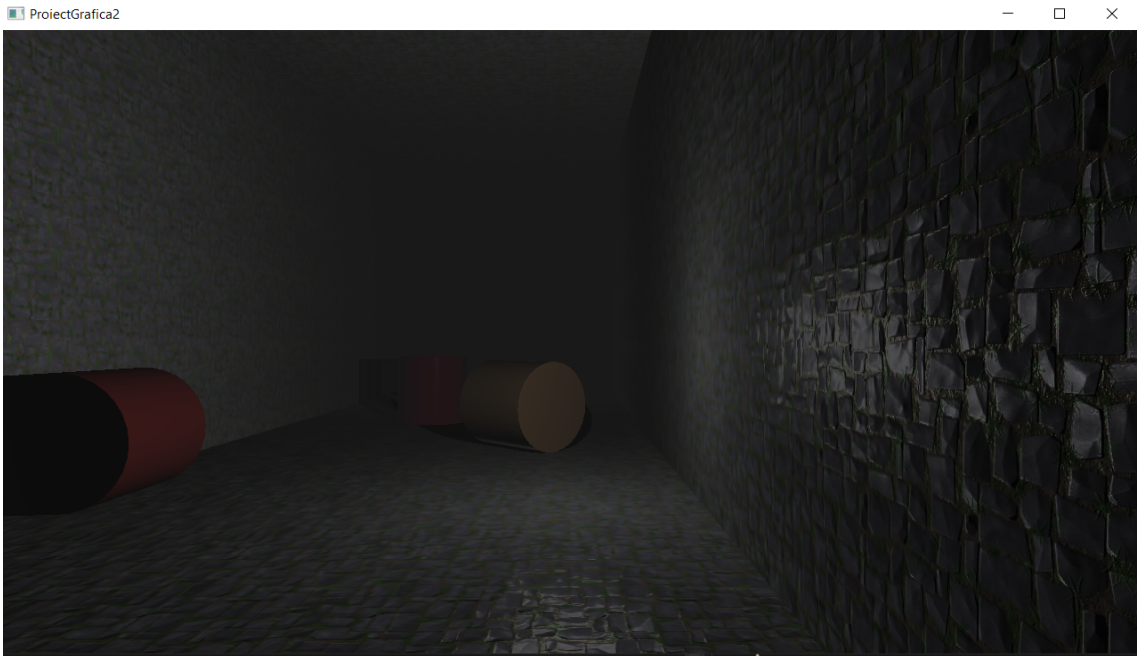


Figura 2.1.1: Interiorul tunelului

## 2.2 Prezentarea cilindrilor

Cilindrii sunt alcătuiți din:

- Două cercuri (bazele) unite între ele

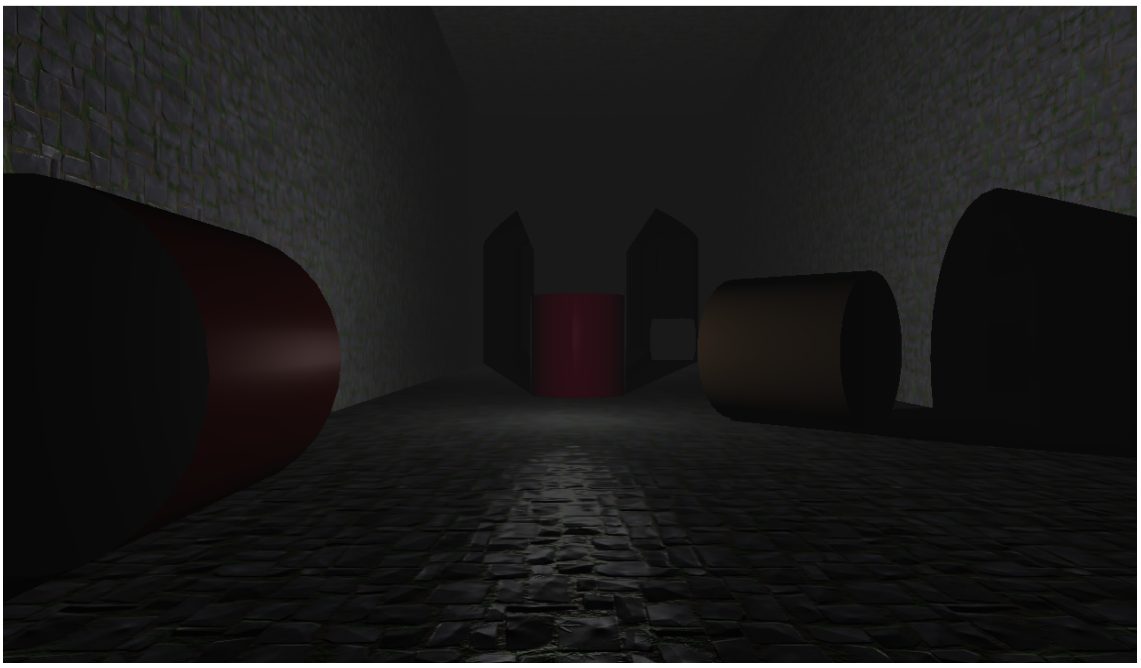

Figura 2.2: Cilindrii aflați în interiorul tunelului

## 2.3 Cod sursă pentru desenarea cilindrilor

```
1 // definim cilindrii
2 float h = -4.f;
3 pair<glm::vec3, pair<glm::vec3, glm::vec3>> cilindrii[10] = {
4     { glm::vec3(225, 193, 110), { glm::vec3(4, h, -40), glm::vec3(0,
          0, 90) } },
5     { glm::vec3(205, 127, 50), { glm::vec3(3, h, -30), glm::vec3() }
          },
6     { glm::vec3(165, 42, 42), { glm::vec3(4, h, -20), glm::vec3(90,
         0, 0) } },
7     { glm::vec3(218, 160, 109), { glm::vec3(-2, h, -15), glm::vec3
         (90, 45, 0) } },
8     { glm::vec3(128, 0, 32), { glm::vec3(1, h, -10), glm::vec3() }
         },
```

```
9       { glm::vec3(233, 116, 81), { glm::vec3(-4, h, 10), glm::vec3(0,
            135, 90) } },
10      { glm::vec3(110, 38, 14), { glm::vec3(-2, h, 30), glm::vec3(0,
            0, 90) } },
11      { glm::vec3(193, 154, 107), { glm::vec3(-1, h, 35), glm::vec3()
            } },
12      { glm::vec3(149, 69, 53), { glm::vec3(2, h, 40), glm::vec3() }
            },
13      { glm::vec3(123, 63, 0), { glm::vec3(0, h, 45), glm::vec3(90, 0,
            0) } },
14  };
15
16  #pragma region randeaza cilindrii
17          sh_cilindru.use();
18          sh_cilindru.setInt("codCol", 0);
19          for (int i = 0; i < 10; i++) {
20              glm::mat4 model;
21              model = glm::translate(model, cilindrii[i].second.first)
                    ;
22              model = glm::rotate(model, glm::radians(cilindrii[i].
                    second.second.z), glm::vec3(0, 0, 1));
23              model = glm::rotate(model, glm::radians(cilindrii[i].
                    second.second.y), glm::vec3(0, 1, 0));
24              model = glm::rotate(model, glm::radians(cilindrii[i].
                    second.second.x), glm::vec3(1, 0, 0));
25
26              sh_cilindru.setMat4("model", model);
27              sh_cilindru.setVec3("uColor", glm::vec3(cilindrii[i].
                    first.x / 255.f, cilindrii[i].first.y / 255.f,
                    cilindrii[i].first.z / 255.f));
28
29              glBindVertexArray(vao_cilindru);
30              glDrawArrays(GL_TRIANGLES, 0, N * 12);
31          }
32  #pragma endregion
```

Figura 2.3.1: Crearea cilindrilor și colorarea acestora

## 2.4 Prezentarea pereților

Pereții sunt alcătuiți din:

- Două triunghiuri

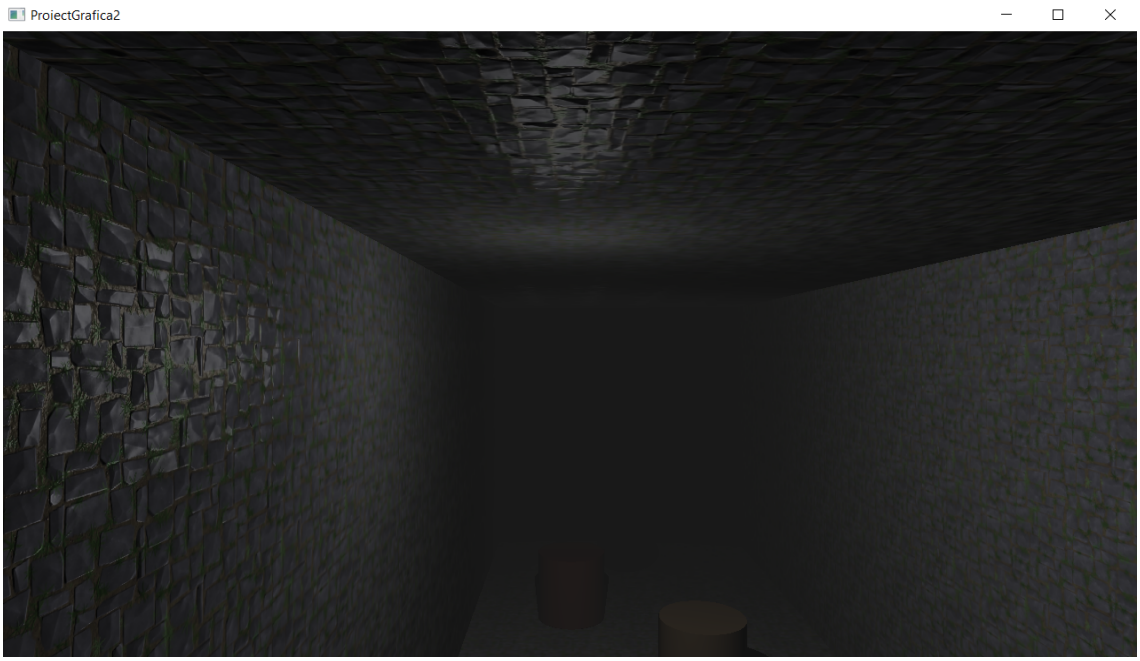


Figura 2.4.1: Pereții din interiorul tunelului


Figura 2.4.2: Pereții din interiorul tunelului

## 2.5 Cod sursă pentru desenarea pereților

```
#pragma region randeaza pereti
        static float dist = 5.f;
        static pair<glm::vec3, glm::vec3> pereti[] = {
            { glm::vec3(-dist, 0, 0), glm::vec3(0, 90, 0) },
            { glm::vec3(dist, 0, 0), glm::vec3(0, -90, 0) },
            { glm::vec3(0, dist, 0), glm::vec3(90, 90, 0) },
            { glm::vec3(0, -dist, 0), glm::vec3(-90, -90, 0) },
            { glm::vec3(0, 0, -50), glm::vec3(0, 0, 0) },
            { glm::vec3(0, 0, 50), glm::vec3(0, 180, 0) },
        };

        sh_perete.use();
        for (int i = 0; i < 6; i++) {
            glm::mat4 model;
            model = glm::translate(model, pereti[i].first);
            model = glm::rotate(model, glm::radians(pereti[i].second
               .z), glm::vec3(0,0,1));
            model = glm::rotate(model, glm::radians(pereti[i].second
               .y), glm::vec3(0,1,0));
            model = glm::rotate(model, glm::radians(pereti[i].second
               .x), glm::vec3(1,0,0));
            if (i < 4) {
                model = glm::scale(model, glm::vec3(100.f, 10.f, 1.f
                   ));
                sh_perete.setFloat("texRatio", 10.f);
            }
            else {
                model = glm::scale(model, glm::vec3(10.f, 10.f, 1.f)
                   );
                sh_perete.setFloat("texRatio", 1.f);
            }
            sh_perete.setMat4("model", model);

            glBindVertexArray(vao_perete);
            glDrawArrays(GL_TRIANGLES, 0, 6);
        }
#pragma endregion
```

Figura 2.2.1: Crearea pereților și colorarea acestora

## 2.6 Cum a fost construit tunelul

Pentru a construi jocul prima dată am generat cilindrii și pereții simpli după care am adăugat shader-ele speciale. Peste care am adăugat texturare specială, lumini și umbre.

## 2.7 Cod sursă pentru construirea tunelului

```
#pragma region cilindru
#define N 30
#define BASE_OFFSET (N * 3 * 2)
#define BASE_OFFSET2 (BASE_OFFSET * 2)
    glm::vec3 vertices[(N * 3 * 2 + N * 6) * 2];
    float deltaAngle = glm::radians(360.f) / N;

    for (int i = 0; i < N; i++) {
        float currentDelta = (i)*deltaAngle;
        float nextDelta = (i + 1) * deltaAngle;

        vertices[6 * i + 0] = glm::vec3(glm::cos(currentDelta), -1.f
           , glm::sin(currentDelta));
        vertices[6 * i + 1] = glm::vec3(0, -1.f, 0);
        vertices[6 * i + 2] = glm::vec3(glm::cos(nextDelta), -1.f,
           glm::sin(nextDelta));
        vertices[6 * i + 3] = glm::vec3(0, -1.f, 0);
        vertices[6 * i + 4] = glm::vec3(0, -1.f, 0);
        vertices[6 * i + 5] = glm::vec3(0, -1.f, 0);

        vertices[BASE_OFFSET + 6 * i + 0] = glm::vec3(glm::cos(
           currentDelta), 1.f, glm::sin(currentDelta));
        vertices[BASE_OFFSET + 6 * i + 1] = glm::vec3(0, 1.f, 0);
        vertices[BASE_OFFSET + 6 * i + 2] = glm::vec3(glm::cos(
           nextDelta), 1.f, glm::sin(nextDelta));
        vertices[BASE_OFFSET + 6 * i + 3] = glm::vec3(0, 1.f, 0);
        vertices[BASE_OFFSET + 6 * i + 4] = glm::vec3(0, 1.f, 0);
        vertices[BASE_OFFSET + 6 * i + 5] = glm::vec3(0, 1.f, 0);

        vertices[BASE_OFFSET2 + 12 * i + 0] = glm::vec3(glm::cos(
           currentDelta), -1.f, glm::sin(currentDelta));
        vertices[BASE_OFFSET2 + 12 * i + 1] = glm::vec3(glm::cos(
           currentDelta), 0, glm::sin(currentDelta));
```

```
28          vertices[BASE_OFFSET2 + 12 * i + 2] = glm::vec3(glm::cos(
               nextDelta), -1.f, glm::sin(nextDelta));
29          vertices[BASE_OFFSET2 + 12 * i + 3] = glm::vec3(glm::cos(
               nextDelta), 0, glm::sin(nextDelta));
30          vertices[BASE_OFFSET2 + 12 * i + 4] = glm::vec3(glm::cos(
               currentDelta), 1.f, glm::sin(currentDelta));
31          vertices[BASE_OFFSET2 + 12 * i + 5] = glm::vec3(glm::cos(
               currentDelta), 0, glm::sin(currentDelta));
32
33          vertices[BASE_OFFSET2 + 12 * i + 6] = glm::vec3(glm::cos(
               currentDelta), 1.f, glm::sin(currentDelta));
34          vertices[BASE_OFFSET2 + 12 * i + 7] = glm::vec3(glm::cos(
               currentDelta), 0, glm::sin(currentDelta));
35          vertices[BASE_OFFSET2 + 12 * i + 8] = glm::vec3(glm::cos(
               nextDelta), 1.f, glm::sin(nextDelta));
36          vertices[BASE_OFFSET2 + 12 * i + 9] = glm::vec3(glm::cos(
               nextDelta), 0, glm::sin(nextDelta));
37          vertices[BASE_OFFSET2 + 12 * i + 10] = glm::vec3(glm::cos(
               nextDelta), -1.f, glm::sin(nextDelta));
38          vertices[BASE_OFFSET2 + 12 * i + 11] = glm::vec3(glm::cos(
               nextDelta), 0, glm::sin(nextDelta));
39      }
40
41      unsigned int vbo_cilindru, vao_cilindru;
42      glGenBuffers(1, &vbo_cilindru);
43      glGenVertexArrays(1, &vao_cilindru);
44      glBindVertexArray(vao_cilindru);
45      glBindBuffer(GL_ARRAY_BUFFER, vbo_cilindru);
46      glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
           GL_STATIC_DRAW);
47
48      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float
           ), (void*)0);
49      glEnableVertexAttribArray(0);
50      glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float
           ), (GLvoid*)(3 * sizeof(GLfloat)));
51      glEnableVertexAttribArray(1);
52  #pragma endregion
53
54  #pragma region perete
```

```
55      unsigned int textura_perete_diffuse = loadTexture("diffuse.jpg")
            ;
56      unsigned int textura_perete_normal = loadTexture("normal.jpg");
57      unsigned int textura_perete_specular = loadTexture("specular.jpg
            ");
58
59      float dim_textura = 5.f;
60      GLfloat v[] = {
61          // pozitie              normale             tex_coords
                                                  tangenta
62          -0.5f, 0.5f, 0.0f,     0, 0, 1,          0.0f, 1.0f *
                dim_textura,                        1, 0, 0,
63          -0.5f, -0.5f, 0.0f,    0, 0, 1,          0.0f, 0.0f,
                                                    1, 0, 0,
64          0.5f, 0.5f, 0.0f,      0, 0, 1,          1.0f * dim_textura,
                1.0f * dim_textura,         1, 0, 0,
65
66          -0.5f, -0.5f, 0.0f,    0, 0, 1,          0.0f, 0.0f,
                                                    1, 0, 0,
67          0.5f, -0.5f, 0.0f,     0, 0, 1,          1.0f * dim_textura,
                0.0f,                       1, 0, 0,
68          0.5f, 0.5f, 0.0f,      0, 0, 1,          1.0f * dim_textura,
                1.0f * dim_textura,         1, 0, 0,
69      };
70
71      unsigned int vbo_perete, vao_perete;
72      glGenBuffers(1, &vbo_perete);
73      glGenVertexArrays(1, &vao_perete);
74      glBindVertexArray(vao_perete);
75      glBindBuffer(GL_ARRAY_BUFFER, vbo_perete);
76      glBufferData(GL_ARRAY_BUFFER, sizeof(v), &v, GL_STATIC_DRAW);
77
78      glEnableVertexAttribArray(0);
79      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 11 * sizeof(
            GLfloat), (void*)0);
80      glEnableVertexAttribArray(1);
81      glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 11 * sizeof(
            GLfloat), (void*)(3 * sizeof(GLfloat)));
82      glEnableVertexAttribArray(2);
83      glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 11 * sizeof(
            GLfloat), (void*)(6 * sizeof(GLfloat)));
```

```
84      glEnableVertexAttribArray(3);
85      glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 11 * sizeof(
            GLfloat), (void*)(8 * sizeof(GLfloat)));
86
87      // texturi perete
88      glActiveTexture(GL_TEXTURE0);
89      glBindTexture(GL_TEXTURE_2D, textura_perete_diffuse);
90      glActiveTexture(GL_TEXTURE1);
91      glBindTexture(GL_TEXTURE_2D, textura_perete_normal);
92      glActiveTexture(GL_TEXTURE2);
93      glBindTexture(GL_TEXTURE_2D, textura_perete_specular);
94
95      sh_perete.use();
96      sh_perete.setInt("tex_diffuse", 0);
97      sh_perete.setInt("tex_normal", 1);
98      sh_perete.setInt("tex_specular", 2);
99  #pragma endregion
```

Figura 2.1: Generarea vertecsiilor, vbou si vaou pentru tunelul nostru

# Aplicarea umbrelor, iluminării și a ceții

Această etapă a fost realizată de ***Linte Robert Ovidiu***.

## 3.1 Prezentarea umbrelor

Pe parcursul deplasării în interiorul tunelului, camera se miscă, iar în acest mod putem observa umbrele cilindrilor.

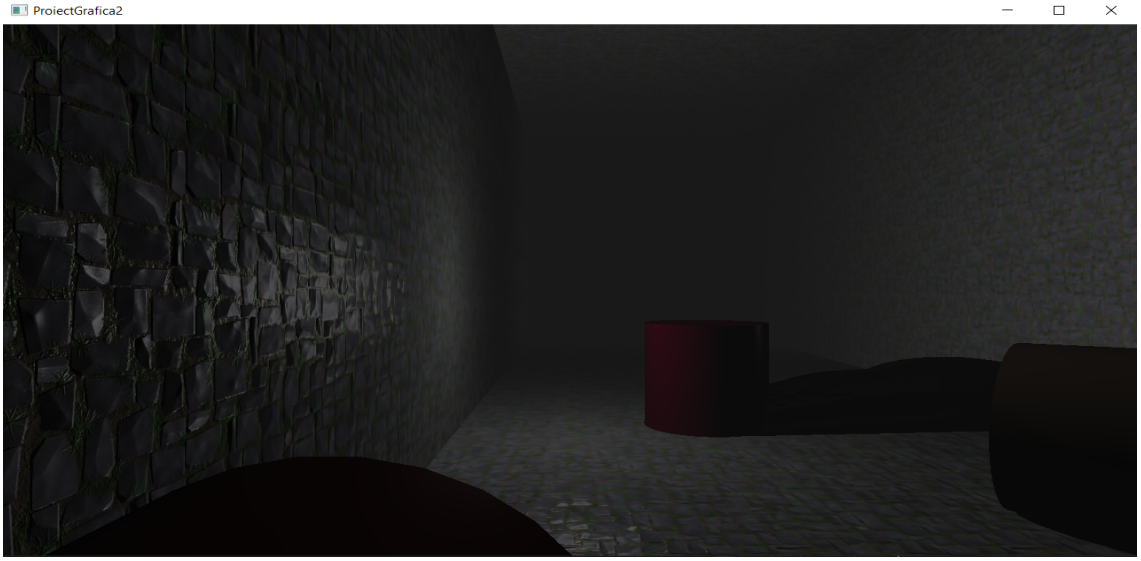


Figura 3.1.1: Umbra cilindrilor

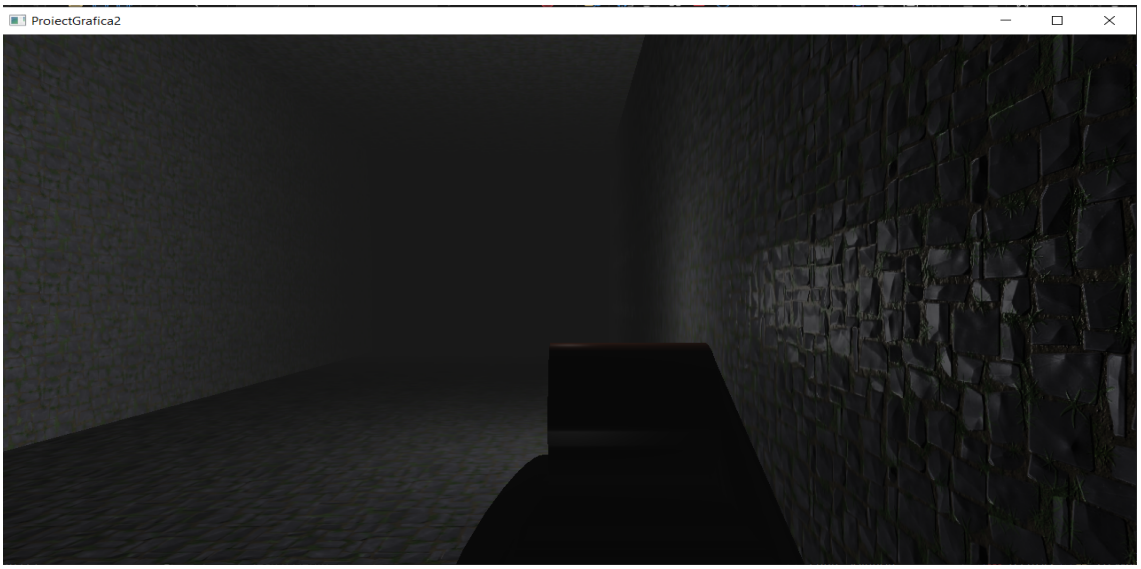


Figura 3.1.2: Umbra cilindrilor

### 3.1.1 Cod sursă

```
// planul pe care aplicam umbrele
glm::vec4 plane_umbra[3] = {
    { 0, 1, 0, 4.99f },
    { 1, 0, 0, 4.99f },
    { -1, 0, 0, 4.99f },
};
```

Figura 3.1.2: Cod OpenGL pentru planul umbrelor cilindrilor

```
float matrUmbra[4][4];

#pragma region umbre
        sh_cilindru.use();
        sh_cilindru.setInt("codCol", 1);
        for (int i = 0; i < 3; i++) {
            glm::mat4 matrUmbra;
            matrUmbra[0][0] = plane_umbra[i].y * lumina.y +
               plane_umbra[i].z * lumina.z + plane_umbra[i].w;
            matrUmbra[0][1] = -plane_umbra[i].y * lumina.x;
            matrUmbra[0][2] = -plane_umbra[i].z * lumina.x;
            matrUmbra[0][3] = -plane_umbra[i].w * lumina.x;
            matrUmbra[1][0] = -plane_umbra[i].x * lumina.y;
            matrUmbra[1][1] = plane_umbra[i].x * lumina.x +
               plane_umbra[i].z * lumina.z + plane_umbra[i].w;
            matrUmbra[1][2] = -plane_umbra[i].z * lumina.y;
            matrUmbra[1][3] = -plane_umbra[i].w * lumina.y;
            matrUmbra[2][0] = -plane_umbra[i].x * lumina.z;
            matrUmbra[2][1] = -plane_umbra[i].y * lumina.z;
            matrUmbra[2][2] = plane_umbra[i].x * lumina.x +
               plane_umbra[i].y * lumina.y + plane_umbra[i].w;
            matrUmbra[2][3] = -plane_umbra[i].w * lumina.z;
            matrUmbra[3][0] = -plane_umbra[i].x;
            matrUmbra[3][1] = -plane_umbra[i].y;
            matrUmbra[3][2] = -plane_umbra[i].z;
            matrUmbra[3][3] = plane_umbra[i].x * lumina.x +
               plane_umbra[i].y * lumina.y + plane_umbra[i].z *
               lumina.z;

            matrUmbra = glm::transpose(matrUmbra);
            sh_cilindru.setMat4("matrUmbra", &matrUmbra[0][0]);
```

```
27
28              for (int i = 0; i < 10; i++) {
29                  glm::mat4 model;
30                  model = glm::translate(model, cilindrii[i].second.
                        first);
31                  model = glm::rotate(model, glm::radians(cilindrii[i
                        ].second.second.z), glm::vec3(0, 0, 1));
32                  model = glm::rotate(model, glm::radians(cilindrii[i
                        ].second.second.y), glm::vec3(0, 1, 0));
33                  model = glm::rotate(model, glm::radians(cilindrii[i
                        ].second.second.x), glm::vec3(1, 0, 0));
34
35                  sh_cilindru.setMat4("model", model);
36                  sh_cilindru.setVec3("uColor", glm::vec3(cilindrii[i
                        ].first.x / 255.f, cilindrii[i].first.y / 255.f,
                        cilindrii[i].first.z / 255.f));
37
38                  glBindVertexArray(vao_cilindru);
39                  glDrawArrays(GL_TRIANGLES, 0, N * 12);
40              }
41          }
42
43  #pragma endregion
```

Figura 3.1.3: Cod OpenGL pentru desenarea umbrelor

## 3.2 Iluminarea cilindrilor și aplicarea ceții

În momentul în care camera se miscă putem observa cum cilindrii sunt iluminați. De asemenea dacă ne uităm drept înainte putem observa ca avem o vizibilitate redusă din cauza ceții.


Figura 3.2.1: Iluminarea cilindrului (putem observa cum cade lumina pe acest cilindru)

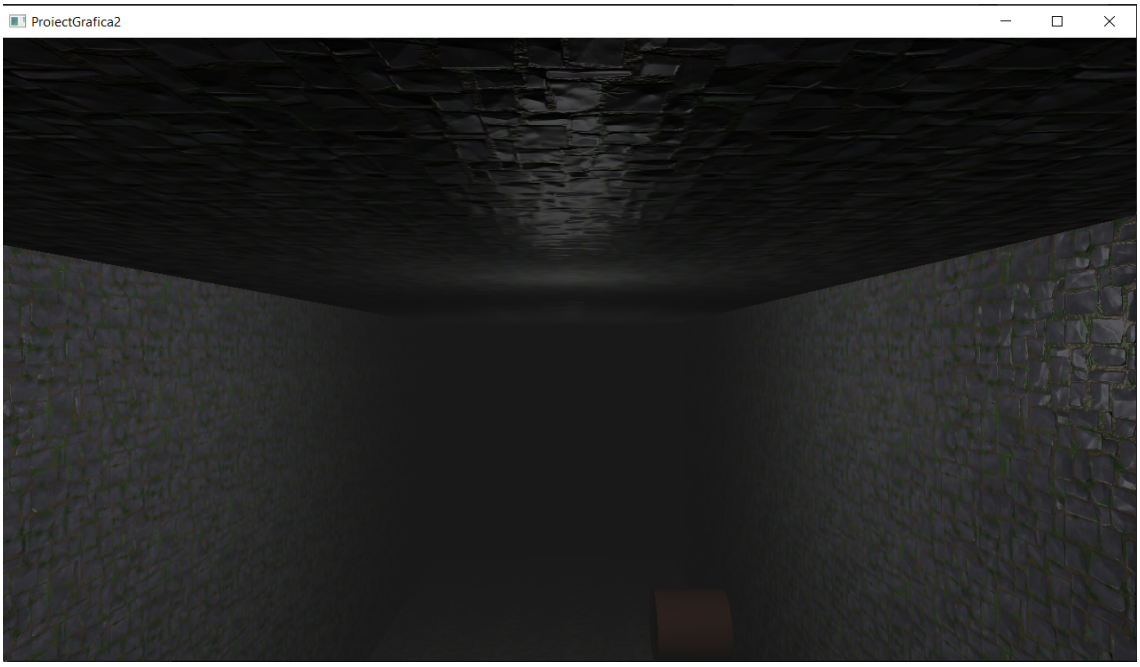


Figura 3.2.2: Ceața - vizibilitate redusă în tunel

### 3.2.1 Cod sursă

```
#version 330 core
out vec4 FragColor;

in vec3 normala;
in vec3 frag_pos;

uniform int codCol = 0;

uniform vec3 uColor = vec3(1, 0, 0);

uniform vec3 camera_pos;

uniform float fogStr = 1.f;
uniform vec3 fogColor = vec3(.1, .1, .1);
uniform vec3 ambient = vec3(1,1,1);

uniform vec3 pozitie_lumina;
uniform vec3 diffuse_lumina = vec3(.5);
uniform vec3 specular_lumina = vec3(.2);

float spec_factor = 64.f;

float ceata() {
    float dist_to_camera = distance(frag_pos, camera_pos);
    float influence = min(1.f, dist_to_camera / (35.f - 10.f));

    return influence * fogStr;
}

void main()
{
    vec3 color = vec3(0.0, 0.0, 0.0);

    if (codCol==0)
  {
        // Ambient
    float ambientStrength = 0.0f;
    vec3 ambient = ambient * ambientStrength;

```

```
40      // Diffuse
41      vec3 normala_ = normala;
42
43      vec3 lightDir = normalize(pozitie_lumina - frag_pos);
44
45      float diff = max(dot(normala_, lightDir), 0.0);
46      vec3 diffuse = diff * diffuse_lumina * uColor;
47
48      // Specular
49      vec3 lookVector = normalize(camera_pos - frag_pos);
50      vec3 lightDirReflected = reflect(-lightDir, normala_);
51      float spec = pow(dot(lookVector, lightDirReflected), spec_factor
            );
52
53      vec3 specular = specular_lumina * spec;
54
55      vec3 emission=vec3(0.0, 0.0, 0.0);
56      vec3 result = emission + ambient + diffuse + specular;
57
58      result =  mix(result, fogColor, ceata());
59
60          FragColor = vec4(result, 1.0f);
61      }
62
63      else if (codCol==1)
64    {
65      color = vec3 (0.0, 0.0, 0.0);
66
67          // ceata
68          color = mix(color, fogColor, ceata());
69
70      FragColor = vec4 (color, 1.0);
71     }
72  }
```

Figura 3.2.3: Cod OpenGL pentru a aplicare iluminării cilindrilor

```
#version 330 core
out vec4 FragColor;

in vec3 normala;
in vec3 frag_pos;

uniform int codCol = 0;

uniform vec3 uColor = vec3(1, 0, 0);

uniform vec3 camera_pos;

uniform float fogStr = 1.f;
uniform vec3 fogColor = vec3(.1, .1, .1);
uniform vec3 ambient = vec3(1,1,1);

uniform vec3 pozitie_lumina;
uniform vec3 diffuse_lumina = vec3(.5);
uniform vec3 specular_lumina = vec3(.2);

float spec_factor = 64.f;

float ceata() {
    float dist_to_camera = distance(frag_pos, camera_pos);
    float influence = min(1.f, dist_to_camera / (35.f - 10.f));

    return influence * fogStr;
}

void main()
{
    vec3 color = vec3(0.0, 0.0, 0.0);

    if (codCol==0)
  {
        // Ambient
    float ambientStrength = 0.0f;
    vec3 ambient = ambient * ambientStrength;

    // Diffuse
    vec3 normala_ = normala;
```

```
42
43      vec3 lightDir = normalize(pozitie_lumina - frag_pos);
44
45      float diff = max(dot(normala_, lightDir), 0.0);
46      vec3 diffuse = diff * diffuse_lumina * uColor;
47
48      // Specular
49      vec3 lookVector = normalize(camera_pos - frag_pos);
50      vec3 lightDirReflected = reflect(-lightDir, normala_);
51      float spec = pow(dot(lookVector, lightDirReflected), spec_factor
            );
52
53      vec3 specular = specular_lumina * spec;
54
55      vec3 emission=vec3(0.0, 0.0, 0.0);
56      vec3 result = emission + ambient + diffuse + specular;
57
58      result =  mix(result, fogColor, ceata());
59
60          FragColor = vec4(result, 1.0f);
61      }
62
63      else if (codCol==1)
64    {
65      color = vec3 (0.0, 0.0, 0.0);
66
67          // ceata
68          color = mix(color, fogColor, ceata());
69
70      FragColor = vec4 (color, 1.0);
71     }
72  }
```

Figura 3.2.3: Cod OpenGL pentru a aplicare iluminării pereților

# Codul Sursă Complet

Codul îl puteți găsi în fișierul ***proiectGrafica.cpp*** sau atașat mai jos.

```
#include <glad/glad.h>
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include "shader.h"
#include "stb_image.h"

using namespace std;

void framebuffer_size_callback(GLFWwindow* window, int width, int
    height);
void processInput(GLFWwindow* window);

unsigned int loadTexture(const char* path);

const unsigned int WIDTH = 1280;
const unsigned int HEIGHT = 720;

// camera
glm::vec3 cameraPosition;
glm::quat cameraOrientation = glm::quat(glm::lookAt(glm::vec3(), glm
    ::vec3(0, 0, 1), glm::vec3(0, 1, 0)));

float delta = 0.0f;
float lastFrame = 0.0f;

float matrUmbra[4][4];

glm::vec3 lumina;

int main()
{
    // glfw: initialize and configure
```

```
36      // ------------------------------
37      glfwInit();
38      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
39      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
40      glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
41
42      // glfw window creation
43      // --------------------
44      //glfwWindowHint(GLFW_SAMPLES, 16);
45      GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "
            ProiectGrafica2", NULL, NULL);
46      if (window == NULL)
47      {
48          std::cout << "Failed to create GLFW window" << std::endl;
49          glfwTerminate();
50          return -1;
51      }
52      glfwMakeContextCurrent(window);
53
54      // glad: load all OpenGL function pointers
55      // ---------------------------------------
56      if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
57      {
58          std::cout << "Failed to initialize GLAD" << std::endl;
59          return -1;
60      }
61
62      glm::mat4 projection = glm::infinitePerspective(glm::radians(50.
            f), (float)WIDTH / (float)HEIGHT, 0.1f);
63
64      Shader sh_perete("perete.vert", "perete.frag");
65      Shader sh_cilindru("cilindru.vert", "cilindru.frag");
66
67  #pragma region cilindru
68  #define N 30
69  #define BASE_OFFSET (N * 3 * 2)
70  #define BASE_OFFSET2 (BASE_OFFSET * 2)
71      glm::vec3 vertices[(N * 3 * 2 + N * 6) * 2];
72      float deltaAngle = glm::radians(360.f) / N;
73
74      for (int i = 0; i < N; i++) {
```

```
75          float currentDelta = (i)*deltaAngle;
76          float nextDelta = (i + 1) * deltaAngle;
77
78          vertices[6 * i + 0] = glm::vec3(glm::cos(currentDelta), -1.f
               , glm::sin(currentDelta));
79          vertices[6 * i + 1] = glm::vec3(0, -1.f, 0);
80          vertices[6 * i + 2] = glm::vec3(glm::cos(nextDelta), -1.f,
               glm::sin(nextDelta));
81          vertices[6 * i + 3] = glm::vec3(0, -1.f, 0);
82          vertices[6 * i + 4] = glm::vec3(0, -1.f, 0);
83          vertices[6 * i + 5] = glm::vec3(0, -1.f, 0);
84
85          vertices[BASE_OFFSET + 6 * i + 0] = glm::vec3(glm::cos(
               currentDelta), 1.f, glm::sin(currentDelta));
86          vertices[BASE_OFFSET + 6 * i + 1] = glm::vec3(0, 1.f, 0);
87          vertices[BASE_OFFSET + 6 * i + 2] = glm::vec3(glm::cos(
               nextDelta), 1.f, glm::sin(nextDelta));
88          vertices[BASE_OFFSET + 6 * i + 3] = glm::vec3(0, 1.f, 0);
89          vertices[BASE_OFFSET + 6 * i + 4] = glm::vec3(0, 1.f, 0);
90          vertices[BASE_OFFSET + 6 * i + 5] = glm::vec3(0, 1.f, 0);
91
92          vertices[BASE_OFFSET2 + 12 * i + 0] = glm::vec3(glm::cos(
               currentDelta), -1.f, glm::sin(currentDelta));
93          vertices[BASE_OFFSET2 + 12 * i + 1] = glm::vec3(glm::cos(
               currentDelta), 0, glm::sin(currentDelta));
94          vertices[BASE_OFFSET2 + 12 * i + 2] = glm::vec3(glm::cos(
               nextDelta), -1.f, glm::sin(nextDelta));
95          vertices[BASE_OFFSET2 + 12 * i + 3] = glm::vec3(glm::cos(
               nextDelta), 0, glm::sin(nextDelta));
96          vertices[BASE_OFFSET2 + 12 * i + 4] = glm::vec3(glm::cos(
               currentDelta), 1.f, glm::sin(currentDelta));
97          vertices[BASE_OFFSET2 + 12 * i + 5] = glm::vec3(glm::cos(
               currentDelta), 0, glm::sin(currentDelta));
98
99          vertices[BASE_OFFSET2 + 12 * i + 6] = glm::vec3(glm::cos(
               currentDelta), 1.f, glm::sin(currentDelta));
100         vertices[BASE_OFFSET2 + 12 * i + 7] = glm::vec3(glm::cos(
               currentDelta), 0, glm::sin(currentDelta));
101         vertices[BASE_OFFSET2 + 12 * i + 8] = glm::vec3(glm::cos(
               nextDelta), 1.f, glm::sin(nextDelta));
```

```
102         vertices[BASE_OFFSET2 + 12 * i + 9] = glm::vec3(glm::cos(
                nextDelta), 0, glm::sin(nextDelta));
103         vertices[BASE_OFFSET2 + 12 * i + 10] = glm::vec3(glm::cos(
                nextDelta), -1.f, glm::sin(nextDelta));
104         vertices[BASE_OFFSET2 + 12 * i + 11] = glm::vec3(glm::cos(
                nextDelta), 0, glm::sin(nextDelta));
105     }
106
107     unsigned int vbo_cilindru, vao_cilindru;
108     glGenBuffers(1, &vbo_cilindru);
109     glGenVertexArrays(1, &vao_cilindru);
110     glBindVertexArray(vao_cilindru);
111     glBindBuffer(GL_ARRAY_BUFFER, vbo_cilindru);
112     glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
            GL_STATIC_DRAW);
113
114     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float
            ), (void*)0);
115     glEnableVertexAttribArray(0);
116     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float
            ), (GLvoid*)(3 * sizeof(GLfloat)));
117     glEnableVertexAttribArray(1);
118 #pragma endregion
119
120 #pragma region perete
121     unsigned int textura_perete_diffuse = loadTexture("diffuse.jpg")
            ;
122     unsigned int textura_perete_normal = loadTexture("normal.jpg");
123     unsigned int textura_perete_specular = loadTexture("specular.jpg
            ");
124
125     float dim_textura = 5.f;
126     GLfloat v[] = {
127         // pozitie              normale             tex_coords
                                                  tangenta
128         -0.5f, 0.5f, 0.0f,     0, 0, 1,          0.0f, 1.0f *
                dim_textura,                      1, 0, 0,
129         -0.5f, -0.5f, 0.0f,    0, 0, 1,          0.0f, 0.0f,
                                                   1, 0, 0,
130         0.5f, 0.5f, 0.0f,      0, 0, 1,          1.0f * dim_textura,
             1.0f * dim_textura,        1, 0, 0,
```

```
131
132         -0.5f, -0.5f, 0.0f,    0, 0, 1,          0.0f, 0.0f,
                                                 1, 0, 0,
133         0.5f, -0.5f, 0.0f,     0, 0, 1,          1.0f * dim_textura,
                0.0f,                        1, 0, 0,
134         0.5f, 0.5f, 0.0f,      0, 0, 1,          1.0f * dim_textura,
                1.0f * dim_textura,          1, 0, 0,
135     };
136
137     unsigned int vbo_perete, vao_perete;
138     glGenBuffers(1, &vbo_perete);
139     glGenVertexArrays(1, &vao_perete);
140     glBindVertexArray(vao_perete);
141     glBindBuffer(GL_ARRAY_BUFFER, vbo_perete);
142     glBufferData(GL_ARRAY_BUFFER, sizeof(v), &v, GL_STATIC_DRAW);
143
144     glEnableVertexAttribArray(0);
145     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 11 * sizeof(
           GLfloat), (void*)0);
146     glEnableVertexAttribArray(1);
147     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 11 * sizeof(
           GLfloat), (void*)(3 * sizeof(GLfloat)));
148     glEnableVertexAttribArray(2);
149     glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 11 * sizeof(
           GLfloat), (void*)(6 * sizeof(GLfloat)));
150     glEnableVertexAttribArray(3);
151     glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 11 * sizeof(
           GLfloat), (void*)(8 * sizeof(GLfloat)));
152
153     // texturi perete
154     glActiveTexture(GL_TEXTURE0);
155     glBindTexture(GL_TEXTURE_2D, textura_perete_diffuse);
156     glActiveTexture(GL_TEXTURE1);
157     glBindTexture(GL_TEXTURE_2D, textura_perete_normal);
158     glActiveTexture(GL_TEXTURE2);
159     glBindTexture(GL_TEXTURE_2D, textura_perete_specular);
160
161     sh_perete.use();
162     sh_perete.setInt("tex_diffuse", 0);
163     sh_perete.setInt("tex_normal", 1);
164     sh_perete.setInt("tex_specular", 2);
```

```
#pragma endregion

#pragma region path
    float startZ = -50.f;
    const int nPoints = 100;
    const float deltaZ = 100.f / nPoints;
    glm::vec3 points[nPoints];
    float angle = 0.f;
    const float pi = 3.141592653589f;
    for (int i = 0; i < nPoints; i++) {
        angle += pi / 16.f;
        points[i] = glm::vec3(glm::cos(angle) * 4.f, glm::sin(angle)
            * 4.f, startZ += deltaZ);
    }
#pragma endregion

    // definim cilindrii
    float h = -4.f;
    pair<glm::vec3, pair<glm::vec3, glm::vec3>> cilindrii[10] = {
        { glm::vec3(225, 193, 110), { glm::vec3(4, h, -40), glm::
           vec3(0, 0, 90) } },
        { glm::vec3(205, 127, 50), { glm::vec3(3, h, -30), glm::vec3
           () } },
        { glm::vec3(165, 42, 42), { glm::vec3(4, h, -20), glm::vec3
           (90, 0, 0) } },
        { glm::vec3(218, 160, 109), { glm::vec3(-2, h, -15), glm::
           vec3(90, 45, 0) } },
        { glm::vec3(128, 0, 32), { glm::vec3(1, h, -10), glm::vec3()
            } },
        { glm::vec3(233, 116, 81), { glm::vec3(-4, h, 10), glm::vec3
           (0, 135, 90) } },
        { glm::vec3(110, 38, 14), { glm::vec3(-2, h, 30), glm::vec3
           (0, 0, 90) } },
        { glm::vec3(193, 154, 107), { glm::vec3(-1, h, 35), glm::
           vec3() } },
        { glm::vec3(149, 69, 53), { glm::vec3(2, h, 40), glm::vec3()
            } },
        { glm::vec3(123, 63, 0), { glm::vec3(0, h, 45), glm::vec3
           (90, 0, 0) } },
    };

```

```
195     // planul pe care aplicam umbrele
196     glm::vec4 plane_umbra[3] = {
197         { 0, 1, 0, 4.99f },
198         { 1, 0, 0, 4.99f },
199         { -1, 0, 0, 4.99f },
200     };
201 
202     int punctCurent = 0;
203     int nextPunct;
204     float progres = 0.f;
205 
206     // curatam ecranul cu negru
207     float culoare_fundal = .3f;
208     glClearColor(culoare_fundal, culoare_fundal, culoare_fundal, 1);
209     glEnable(GL_DEPTH_TEST);
210     //glEnable(GL_CULL_FACE);
211     glEnable(GL_BLEND);
212     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
213     while (!glfwWindowShouldClose(window))
214     {
215         float currentFrame = static_cast<float>(glfwGetTime());
216         delta = currentFrame - lastFrame;
217         lastFrame = currentFrame;
218         processInput(window);
219         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
220 
221         // misca camera
222         progres += delta * 5.f;
223         if (progres >= 1.f) {
224             progres -= 1.f;
225             punctCurent = min(punctCurent + 1, nPoints - 1);
226         }
227         nextPunct = min(punctCurent + 1, nPoints - 1);
228         glm::vec3 pozCurenta = glm::mix(points[punctCurent], points[
               nextPunct], progres);
229         cameraPosition = pozCurenta;
230 
231         lumina = cameraPosition + glm::vec3(0, 0, 10.f);
232         // setam matricile de vizualizare si pozitia camerei
233         glm::mat4 viewMatrix;
```

```
234         glm::mat4 rotationMatrix = glm::mat4_cast(cameraOrientation)
                ;
235         glm::mat4 translationMatrix = glm::translate(glm::mat4(1.0f)
                , -cameraPosition);
236         viewMatrix = rotationMatrix * translationMatrix;
237
238         sh_perete.use();
239         sh_perete.setMat4("view", viewMatrix);
240         sh_perete.setMat4("projection", projection);
241         sh_perete.setVec3("camera_pos", cameraPosition);
242         sh_perete.setVec3("pozitie_lumina", lumina);
243
244         sh_cilindru.use();
245         sh_cilindru.setMat4("view", viewMatrix);
246         sh_cilindru.setMat4("projection", projection);
247         sh_cilindru.setVec3("camera_pos", cameraPosition);
248         sh_cilindru.setVec3("pozitie_lumina", lumina);
249
250 #pragma region randeaza pereti
251         static float dist = 5.f;
252         static pair<glm::vec3, glm::vec3> pereti[] = {
253             { glm::vec3(-dist, 0, 0), glm::vec3(0, 90, 0) },
254             { glm::vec3(dist, 0, 0), glm::vec3(0, -90, 0) },
255             { glm::vec3(0, dist, 0), glm::vec3(90, 90, 0) },
256             { glm::vec3(0, -dist, 0), glm::vec3(-90, -90, 0) },
257             { glm::vec3(0, 0, -50), glm::vec3(0, 0, 0) },
258             { glm::vec3(0, 0, 50), glm::vec3(0, 180, 0) },
259         };
260
261         sh_perete.use();
262         for (int i = 0; i < 6; i++) {
263             glm::mat4 model;
264             model = glm::translate(model, pereti[i].first);
265             model = glm::rotate(model, glm::radians(pereti[i].second
                    .z), glm::vec3(0,0,1));
266             model = glm::rotate(model, glm::radians(pereti[i].second
                    .y), glm::vec3(0,1,0));
267             model = glm::rotate(model, glm::radians(pereti[i].second
                    .x), glm::vec3(1,0,0));
268             if (i < 4) {
```

```
269                 model = glm::scale(model, glm::vec3(100.f, 10.f, 1.f
                        ));
270                 sh_perete.setFloat("texRatio", 10.f);
271             }
272             else {
273                 model = glm::scale(model, glm::vec3(10.f, 10.f, 1.f)
                        );
274                 sh_perete.setFloat("texRatio", 1.f);
275             }
276             sh_perete.setMat4("model", model);
277 
278             glBindVertexArray(vao_perete);
279             glDrawArrays(GL_TRIANGLES, 0, 6);
280         }
281 #pragma endregion
282 
283 #pragma region randeaza cilindrii
284         sh_cilindru.use();
285         sh_cilindru.setInt("codCol", 0);
286         for (int i = 0; i < 10; i++) {
287             glm::mat4 model;
288             model = glm::translate(model, cilindrii[i].second.first)
                        ;
289             model = glm::rotate(model, glm::radians(cilindrii[i].
                        second.second.z), glm::vec3(0, 0, 1));
290             model = glm::rotate(model, glm::radians(cilindrii[i].
                        second.second.y), glm::vec3(0, 1, 0));
291             model = glm::rotate(model, glm::radians(cilindrii[i].
                        second.second.x), glm::vec3(1, 0, 0));
292 
293             sh_cilindru.setMat4("model", model);
294             sh_cilindru.setVec3("uColor", glm::vec3(cilindrii[i].
                        first.x / 255.f, cilindrii[i].first.y / 255.f,
                        cilindrii[i].first.z / 255.f));
295 
296             glBindVertexArray(vao_cilindru);
297             glDrawArrays(GL_TRIANGLES, 0, N * 12);
298         }
299 #pragma endregion
300 
301 #pragma region umbre
```

```
302         sh_cilindru.use();
303         sh_cilindru.setInt("codCol", 1);
304         for (int i = 0; i < 3; i++) {
305             glm::mat4 matrUmbra;
306             matrUmbra[0][0] = plane_umbra[i].y * lumina.y +
                    plane_umbra[i].z * lumina.z + plane_umbra[i].w;
307             matrUmbra[0][1] = -plane_umbra[i].y * lumina.x;
308             matrUmbra[0][2] = -plane_umbra[i].z * lumina.x;
309             matrUmbra[0][3] = -plane_umbra[i].w * lumina.x;
310             matrUmbra[1][0] = -plane_umbra[i].x * lumina.y;
311             matrUmbra[1][1] = plane_umbra[i].x * lumina.x +
                    plane_umbra[i].z * lumina.z + plane_umbra[i].w;
312             matrUmbra[1][2] = -plane_umbra[i].z * lumina.y;
313             matrUmbra[1][3] = -plane_umbra[i].w * lumina.y;
314             matrUmbra[2][0] = -plane_umbra[i].x * lumina.z;
315             matrUmbra[2][1] = -plane_umbra[i].y * lumina.z;
316             matrUmbra[2][2] = plane_umbra[i].x * lumina.x +
                    plane_umbra[i].y * lumina.y + plane_umbra[i].w;
317             matrUmbra[2][3] = -plane_umbra[i].w * lumina.z;
318             matrUmbra[3][0] = -plane_umbra[i].x;
319             matrUmbra[3][1] = -plane_umbra[i].y;
320             matrUmbra[3][2] = -plane_umbra[i].z;
321             matrUmbra[3][3] = plane_umbra[i].x * lumina.x +
                    plane_umbra[i].y * lumina.y + plane_umbra[i].z *
                    lumina.z;
322
323             matrUmbra = glm::transpose(matrUmbra);
324             sh_cilindru.setMat4("matrUmbra", &matrUmbra[0][0]);
325
326             for (int i = 0; i < 10; i++) {
327                 glm::mat4 model;
328                 model = glm::translate(model, cilindrii[i].second.
                        first);
329                 model = glm::rotate(model, glm::radians(cilindrii[i
                        ].second.second.z), glm::vec3(0, 0, 1));
330                 model = glm::rotate(model, glm::radians(cilindrii[i
                        ].second.second.y), glm::vec3(0, 1, 0));
331                 model = glm::rotate(model, glm::radians(cilindrii[i
                        ].second.second.x), glm::vec3(1, 0, 0));
332
333                 sh_cilindru.setMat4("model", model);
```

```
334                 sh_cilindru.setVec3("uColor", glm::vec3(cilindrii[i
                        ].first.x / 255.f, cilindrii[i].first.y / 255.f,
                        cilindrii[i].first.z / 255.f));
335
336                 glBindVertexArray(vao_cilindru);
337                 glDrawArrays(GL_TRIANGLES, 0, N * 12);
338             }
339         }
340
341 #pragma endregion
342
343
344         glfwSwapBuffers(window);
345         glfwPollEvents();
346     }
347
348     glfwTerminate();
349     return 0;
350 }
351
352 void processInput(GLFWwindow* window)
353 {
354     if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
355         glfwSetWindowShouldClose(window, true);
356 }
357
358 void framebuffer_size_callback(GLFWwindow* window, int width, int
        height)
359 {
360     glViewport(0, 0, width, height);
361 }
362
363 // functie standard de incarcat texturi
364 unsigned int loadTexture(char const* path)
365 {
366     unsigned int textureID;
367     glGenTextures(1, &textureID);
368
369     int width, height, nrComponents;
370     unsigned char* data = stbi_load(path, &width, &height, &
            nrComponents, 0);
```

```
371     if (data)
372     {
373         GLenum format;
374         if (nrComponents == 1)
375             format = GL_RED;
376         else if (nrComponents == 3)
377             format = GL_RGB;
378         else if (nrComponents == 4)
379             format = GL_RGBA;
380 
381         glBindTexture(GL_TEXTURE_2D, textureID);
382         glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0,
               format, GL_UNSIGNED_BYTE, data);
383         glGenerateMipmap(GL_TEXTURE_2D);
384 
385         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, format ==
               GL_RGBA ? GL_CLAMP_TO_EDGE : GL_REPEAT);
386         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, format ==
               GL_RGBA ? GL_CLAMP_TO_EDGE : GL_REPEAT);
387         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
               GL_LINEAR_MIPMAP_LINEAR);
388         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
               GL_LINEAR);
389 
390         stbi_image_free(data);
391     }
392     else
393     {
394         std::cout << "Texture failed to load at path: " << path <<
               std::endl;
395         stbi_image_free(data);
396     }
397 
398     return textureID;
399 }
```

# Codurile pentru Shadere

Codurile le puteți găsi atașate mai jos sau ***aici***.

```
#version 330 core
layout (location = 0) in vec3 in_pozitie;
layout (location = 1) in vec3 in_normala;
layout (location = 2) in vec2 in_tex_coords;
layout (location = 3) in vec3 in_tangenta;

out vec2 tex_coords;
out vec3 normala;
out vec3 frag_pos;
out mat3 TBN;

uniform int codCol = 0;
uniform float texRatio;

uniform mat4 matrUmbra;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(in_pozitie, 1.0);

    mat3 normalMatrix = transpose(inverse(mat3(model)));
    normala = normalMatrix * in_normala;

    frag_pos = vec3(model * vec4(in_pozitie, 1.0));
    tex_coords = vec2(in_tex_coords.x * texRatio, in_tex_coords.y);

    vec3 T = normalize(vec3(model * vec4(in_tangenta,   0.0)));
    vec3 N = normalize(vec3(model * vec4(in_normala,    0.0)));
    vec3 B = cross(N, T);
    TBN = mat3(T, B, N);

        /*
    if (codCol == 0)
    {
```

```
38          gl_Position = projection * view * model * vec4(in_pozitie,
                1.0);
39
40          // mat3 normalMatrix = transpose(inverse(mat3(model)));
41          //vec3 normal = vec3(0,0,-1);
42          //normala = normalMatrix * normal;
43
44          frag_pos = vec3(model * vec4(in_pozitie, 1.0));
45          // coord_tex = in_coord_tex;
46      }
47      else if (codCol==1)
48      {
49      gl_Position = projection*view*matrUmbra*model*vec4(in_pozitie,
            1.f);
50          frag_pos = vec3(matrUmbra * model * vec4(in_pozitie, 1.0));
51      }
52      */
53  }
```

Figura 5.1.1: Cod OpenGL pentru shader-ul vertex pentru perete

```
1   #version 330 core
2   out vec4 FragColor;
3
4   in vec2 tex_coords;
5   in vec3 normala;
6   in vec3 frag_pos;
7   in mat3 TBN;
8
9   uniform int codCol = 0;
10
11  uniform vec3 camera_pos;
12
13  uniform float fogStr = 1.f;
14  uniform vec3 fogColor = vec3(.1, .1, .1);
15  uniform vec3 ambient = vec3(1,1,1);
16
17  uniform vec3 pozitie_lumina;
18  uniform vec3 diffuse_lumina = vec3(1.f);
19  uniform vec3 specular_lumina = vec3(.3);
20
```

```
21 float spec_factor = 64.f;
22
23 uniform sampler2D tex_diffuse;
24 uniform sampler2D tex_normal;
25 uniform sampler2D tex_specular;
26
27 float ceata() {
28     float dist_to_camera = distance(frag_pos, camera_pos);
29     float influence = min(1.f, dist_to_camera / (45.f - 15.f));
30
31     return influence * fogStr;
32 }
33
34 void main()
35 {
36     vec3 color = vec3(0.0, 0.0, 0.0);
37
38     if (codCol==0)
39   {
40         // Ambient
41     float ambientStrength = 0.0f;
42     vec3 ambient = ambient * ambientStrength;
43
44     // Diffuse
45     vec3 normala_ = texture(tex_normal, tex_coords).rgb;
46     normala_ = normala_ * 2.0 - 1.0;
47     normala_ = normalize(TBN * normala_);
48
49     vec3 lightDir = normalize(pozitie_lumina - frag_pos);
50
51     float diff = max(dot(normala_, lightDir), 0.0);
52     vec3 diffuse = diff * diffuse_lumina * vec3(texture(tex_diffuse,
           tex_coords));
53
54     // Specular
55     vec3 lookVector = normalize(camera_pos - frag_pos);
56     vec3 lightDirReflected = reflect(-lightDir, normala_);
57     float spec = pow(dot(lookVector, lightDirReflected), spec_factor
          );
58
59     vec3 specular = specular_lumina * spec *  vec3(texture(
```

```
       tex_specular, tex_coords));

    vec3 emission=vec3(0.0, 0.0, 0.0);
    vec3 result = emission + ambient + diffuse + specular;

    result =  mix(result, fogColor, ceata());

        FragColor = vec4(result, 1.0f);
    }

    else if (codCol==1)
  {
    color = vec3 (0.0, 0.0, 0.0);

        // ceata
        color = mix(color, fogColor, ceata());

    FragColor = vec4 (color, 1.0);
   }
}
```

Figura 5.1.2: Cod OpenGL pentru shader-ul de fragement pentru perete

```
#version 330 core
layout (location = 0) in vec3 in_pozitie;
layout (location = 1) in vec3 in_normala;

out vec3 normala;
out vec3 frag_pos;

uniform int codCol = 0;

uniform mat4 matrUmbra;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    if (codCol == 0)
    {
```

```
        gl_Position = projection * view * model * vec4(in_pozitie,
            1.0);

        mat3 normalMatrix = transpose(inverse(mat3(model)));
        normala = normalMatrix * in_normala;

        frag_pos = vec3(model * vec4(in_pozitie, 1.0));
    } else {
        gl_Position = projection * view * matrUmbra * model * vec4(
            in_pozitie, 1.f);
        frag_pos = vec3(model * vec4(in_pozitie, 1.0));
    }
}
```

Figura 5.1.3: Cod OpenGL pentru shader-ul de vertex pentru cilindru

```
#version 330 core
out vec4 FragColor;

in vec3 normala;
in vec3 frag_pos;

uniform int codCol = 0;

uniform vec3 uColor = vec3(1, 0, 0);

uniform vec3 camera_pos;

uniform float fogStr = 1.f;
uniform vec3 fogColor = vec3(.1, .1, .1);
uniform vec3 ambient = vec3(1,1,1);

uniform vec3 pozitie_lumina;
uniform vec3 diffuse_lumina = vec3(.5);
uniform vec3 specular_lumina = vec3(.2);

float spec_factor = 64.f;

float ceata() {
    float dist_to_camera = distance(frag_pos, camera_pos);
    float influence = min(1.f, dist_to_camera / (35.f - 10.f));
```

```
26
27      return influence * fogStr;
28  }
29
30  void main()
31  {
32      vec3 color = vec3(0.0, 0.0, 0.0);
33
34      if (codCol==0)
35    {
36          // Ambient
37      float ambientStrength = 0.0f;
38      vec3 ambient = ambient * ambientStrength;
39
40      // Diffuse
41      vec3 normala_ = normala;
42
43      vec3 lightDir = normalize(pozitie_lumina - frag_pos);
44
45      float diff = max(dot(normala_, lightDir), 0.0);
46      vec3 diffuse = diff * diffuse_lumina * uColor;
47
48      // Specular
49      vec3 lookVector = normalize(camera_pos - frag_pos);
50      vec3 lightDirReflected = reflect(-lightDir, normala_);
51      float spec = pow(dot(lookVector, lightDirReflected), spec_factor
          );
52
53      vec3 specular = specular_lumina * spec;
54
55      vec3 emission=vec3(0.0, 0.0, 0.0);
56      vec3 result = emission + ambient + diffuse + specular;
57
58      result =  mix(result, fogColor, ceata());
59
60          FragColor = vec4(result, 1.0f);
61      }
62
63      else if (codCol==1)
64    {
65      color = vec3 (0.0, 0.0, 0.0);
```

```
66
67          // ceata
68          color = mix(color, fogColor, ceata());
69
70      FragColor = vec4 (color, 1.0);
71    }
72 }
```

Figura 5.1.3: Cod OpenGL pentru shader-ul de fragement pentru cilindru

# Referințe

- Generarea vertexilor pentru cilintru este preluată din laboratoarele anterioare.
- https://learnopengl.com/Advanced-Lighting/Normal-Mapping
- Materiale din Curs.