Laborator 9

1 Obiective

Obiectivul acestui laborator este să descrie pe scurt un mod simplu de adăugare a umbrelor în aplicația dvs. OpenGL 4 folosind tehnica **Shadow Mapping**.

2 Shadow Mapping

Tehnicile de iluminat discutate în laboratoarele anterioare funcționează excelent pentru iluminarea obiectelor. Totuși, efectul de iluminare obținut nu ia în considerare ocluziile. În lumea reală, atunci când un obiect nu este lovit în mod direct de razele emise de o sursă de lumină din cauza obstrucțiilor (alte obiecte), atunci obiectul este în umbră. Astfel, umbrele adaugă un grad de realism scenelor noastre OpenGL, permițând, de asemenea, o percepție mai profundă a adâncimii (Figura 1 și Figura 2).

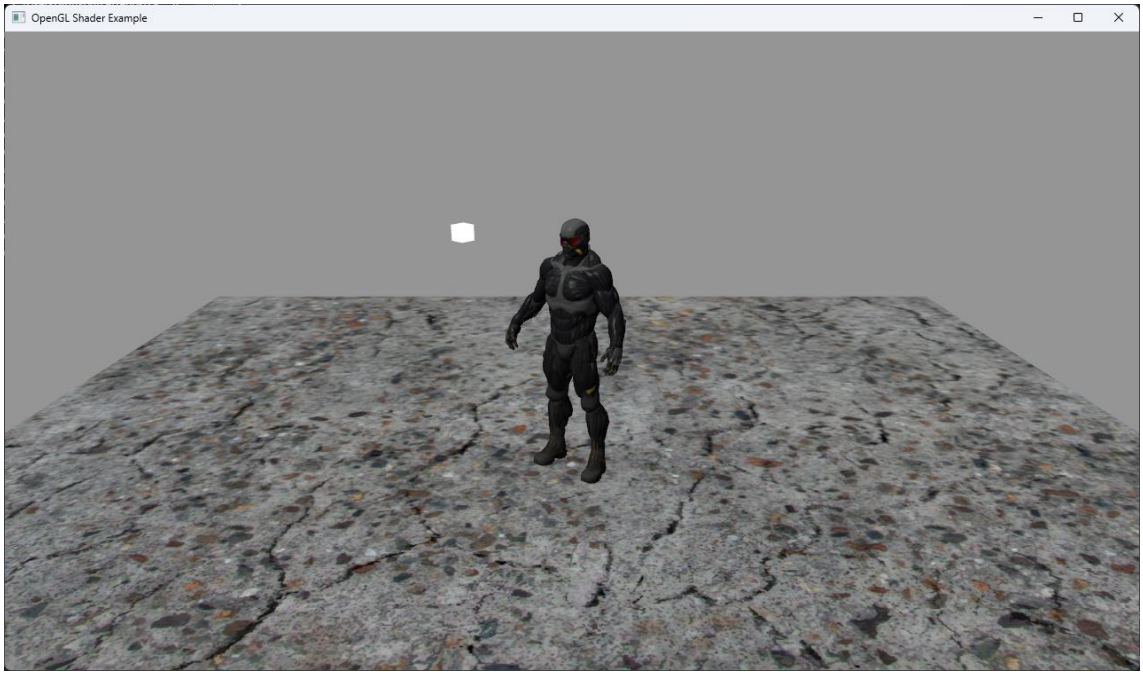

Figura 1 – Rasterizare fără umbre

Există o mulțime de tehnici moderne care pot fi folosite pentru a produce umbre (calculul umbrelor) și toate acestea pot fi implementate în OpenGL. Cu toate acestea, nu există algoritmi perfecți în timp real pentru acest lucru, fiecare având propriile sale avantaje și dezavantaje. Pentru acest laborator, vom explora o tehnică ce oferă rezultate decente și este ușor de implementat, și anume shadow mapping (“hărți de umbre”).

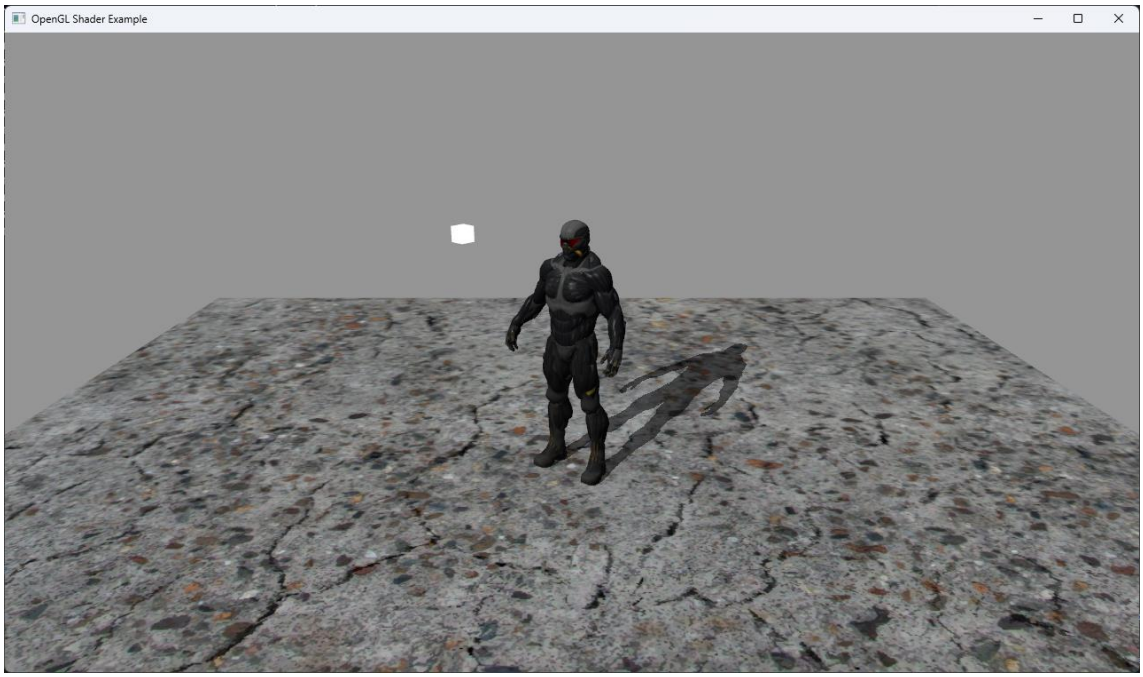

Figura 2 – Rasterizare cu umbre

Shadow mapping este o tehnică multi-trecere care utilizează texturi de adâncime pentru a decide dacă un punct se află în umbră sau nu. Cheia este aceea de a observa scena din punctul de vedere al sursei de lumină în loc de locația finală de vizionare (locația camerei). Orice parte a scenei care nu este direct observabilă din perspectiva luminii va fi în umbră.

Etapele principale ale algoritmului sunt descrise mai jos::

1. Rasterizarea scenei din punctul de vedere al luminii. Nu contează cum arată scena (informații despre culoare); singurele informații relevante în acest moment sunt valorile de adâncime. Aceste valori sunt stocate într-o hartă de umbră (sau hartă de adâncime) și pot fi obținute prin crearea unei texturi de adâncime, atașarea la un obiect framebuffer și rasterizarea întregii scene (așa cum este văzută din poziția luminii) în acest obiect. În acest fel, textura de adâncime este umplută direct cu valorile relevante ale adâncimii.
2. Rasterizarea scenei din punct de vedere al observatorului (poziția camerei). Se compară adâncimea fiecărui fragment vizibil (proiectat în cadrul de referință al luminii) cu valorile de adâncime din harta umbrelor. Fragmentele care au o adâncime mai mare decât cea care a fost stocată anterior în harta de adâncime nu sunt direct vizibile din punctul de vedere al luminii și sunt, prin urmare, în umbră.

2.1 Generarea hărţii de adâncime

Prima trecere a algoritmului de shadow mapping generează o hartă de adâncime a întregii scene privită din perspectiva luminii. Această hartă poate fi salvată direct într-o textura prin atașarea texturii la un obiect framebuffer și rasterizarea directă în acesta.

OpenGL face toată rasterizarea (informații despre culoare, adâncime și șablon) într-un framebuffer. Culorile stocate în framebuffer sunt utilizate de ecran atunci când se afișează conținutul vizual generat de aplicațiile noastre. Obiectele Framebuffer ne permit să creăm propriile framebuffere și, în loc să rasterizăm în zone speciale de memorie GPU, putem scrie informații (culori, adâncime și informații șablon) în texturi. Aceste texturi sunt atașate ca obiecte buffer de culoare, adâncime și stencil pentru obiectele noastre framebuffer.

2.1.1 Crearea unui obiect framebuffer

Un obiect framebuffer este creat după cum se arată mai jos:

```
GLuint shadowMapFBO;
//generate FBO ID
glGenFramebuffers(1, &shadowMapFBO);
```

Pentru a atașa o textură ca un buffer de adâncime pentru obiectul framebuffer, trebuie mai întâi să creăm textura:

```
GLuint depthMapTexture;
//create depth texture for FBO
glGenTextures(1, &depthMapTexture);
glBindTexture(GL_TEXTURE_2D, depthMapTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
        SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
```

Textura de adâncime este creată folosind tipul de textură GL_DEPTH_COMPONENT. De asemenea, pentru a evita artefactele vizuale nedorite atunci când generați umbre, modul de înfășurare a texturii este setat la GL_CLAMP_TO_BORDER. SHADOW_WIDTH și SHADOW_HEIGTH sunt constante și reprezintă rezoluția hărții de adâncime, care ar trebui să fie cel puțin de dimensiunea framebuffer-ului ferestrei OpenGL. Rezoluțiile prea scăzute au tendința de a conduce la rezultate mai slabe, iar rezoluțiile prea mari au tendința de a irosi memoria.

Odată ce textura de adâncime este creată, ea trebuie atașată ca buffer de adâncime pentru obiectul framebuffer:

```
//attach texture to FBO
glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMapTexture,
0);
```

Deoarece prima trecere a algoritmului de mapare a umbrelor nu necesită atașament de coloare sau șablon, dar un obiect framebuffer nu ar fi complet fără ele, putem să atribuim în mod explicit **nimic** acestor puncte de atașament:

```
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
```

Odată ce un obiect framebuffer este complet, ar trebui să îl dezactivăm până când suntem gata să îl folosim:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Cu un obiect framebuffer configurat corect, etapa completă de redare, folosind harta de adâncime, este după cum urmează:

```
// rasterizați în harta de adâncime
Schimbați portul de vizualizare pentru a acoperi dimensiunea texturării de adâncime;
Activați obiectul framebuffer;
        Curățați (inițializati) buffer-ul de adâncime;
        Configurați shader-ul (pentru completarea texturii de adâncime) și matricile de transformare (pentru sistemul de coordinate al luminii);
        Rasterizați scena;
Dezactivați obiectul framebuffer;
// rasterizați în framebuffer
Modificați portul de vizualizare pentru a acomoda obiectul framebuffer al ferestrei OpenGL;
        Curățați (inițializati) buffer-ul de adâncime și culoarea;
        Configurați shader-ul (pentru rasterizarea cu umbre) și matricile de transformare;
        Activați (legați) textura de adâncime;
        Rasterizați scena;
```

2.1.2 Transformări în spațiul luminii

Prima trecere ar trebui să rasterizeze scena din punctul de vedere al luminii. Aceasta înseamnă că toate coordonatele tuturor obiectelor din scenă trebuie transformate astfel încât să fie relative la lumină și nu la cameră. Astfel, putem folosi aceeași funcție ca și în cazul transformarea spațiului camerei, **lookAt**, dar cu poziția luminii ca prim parametru. Pentru acest laborator, vom folosi o lumină direcțională. Deoarece luminile direcționale nu au poziție (sunt amplasate la infinit), putem folosi ca poziție orice punct de-a lungul direcției luminii, inclusiv direcția însăși (interpretată ca un punct):

```
glm::mat4 lightView = glm::lookAt(lightDir, glm::vec3(0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

Deoarece toate razele provenite de la o lumină direcțională sunt paralele, vom folosi o proiecție ortografică pentru a evita orice deformare perspectivă:

```
const GLfloat near_plane = 0.1f, far_plane = 6.0f;
glm::mat4 lightProjection = glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, near_plane, far_plane);
```

Matricea finală de transformare pentru spațiul luminii este:

```
glm::mat4 lightSpaceTrMatrix = lightProjection * lightView;
```

2.1.3 Rasterizarea în harta de adâncime

Vertex Shader-ul care transformă toate vârfurile în spațiul luminii este prezentat mai jos:

```
#version 410 core
```

```
layout(location=0) in vec3 vPosition;

uniform mat4 lightSpaceTrMatrix;
uniform mat4 model;

void main()
{
    gl_Position = lightSpaceTrMatrix * model * vec4(vPosition, 1.0f);
}
```

În OpenGL 4, este permis să nu aveți un shader de fragment numai atunci când rasterizarea este oprită. Astfel, un simplu shader de fragment cu ieșire "dummy"(falsă) trebuie implementat[1]:

```
#version 410 core

out vec4 fColor;

void main()
{
        fColor = vec4(1.0f);
}
```

Rasterizarea în harta de adâncime devine:

```
//render the scene to the depth buffer

depthMapShader.useShaderProgram();

glUniformMatrix4fv(glGetUniformLocation(depthMapShader.shaderProgram, "lightSpaceTrMatrix"),
        1,
        GL_FALSE,
        glm::value_ptr(computeLightSpaceTrMatrix()));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);

RenderTheScene();

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Dacă ar fi să vizualizăm harta de adâncime în acest moment, ar arăta în figura 3.

2.2 Rasterizarea cu umbre

Odată ce hartă adâncimii a fost creată, ea poate fi utilizată pentru a rasteriza scena cu umbre. Când se compară adâncimea fragmentului curent cu adâncimea existentă în harta de adâncime, ambele trebuie să

[1] Am putea avea, de asemenea, un shader de fragment gol, deoarece ne pasă doar de valorile de adâncime, care sunt generate în spatele scenei de OpenGL oricum.

se afle în sistemul de referință al luminii. Deoarece primul nu este, trebuie transformat folosind aceeași matrice ca atunci când se generează harta. Transformările ar trebui efectuate în vertex shader:

```
…
out vec4 fragPosLightSpace;
…
uniform mat4 lightSpaceTrMatrix;
…
fragPosLightSpace = lightSpaceTrMatrix * model * vec4(vPosition, 1.0f);
…
```

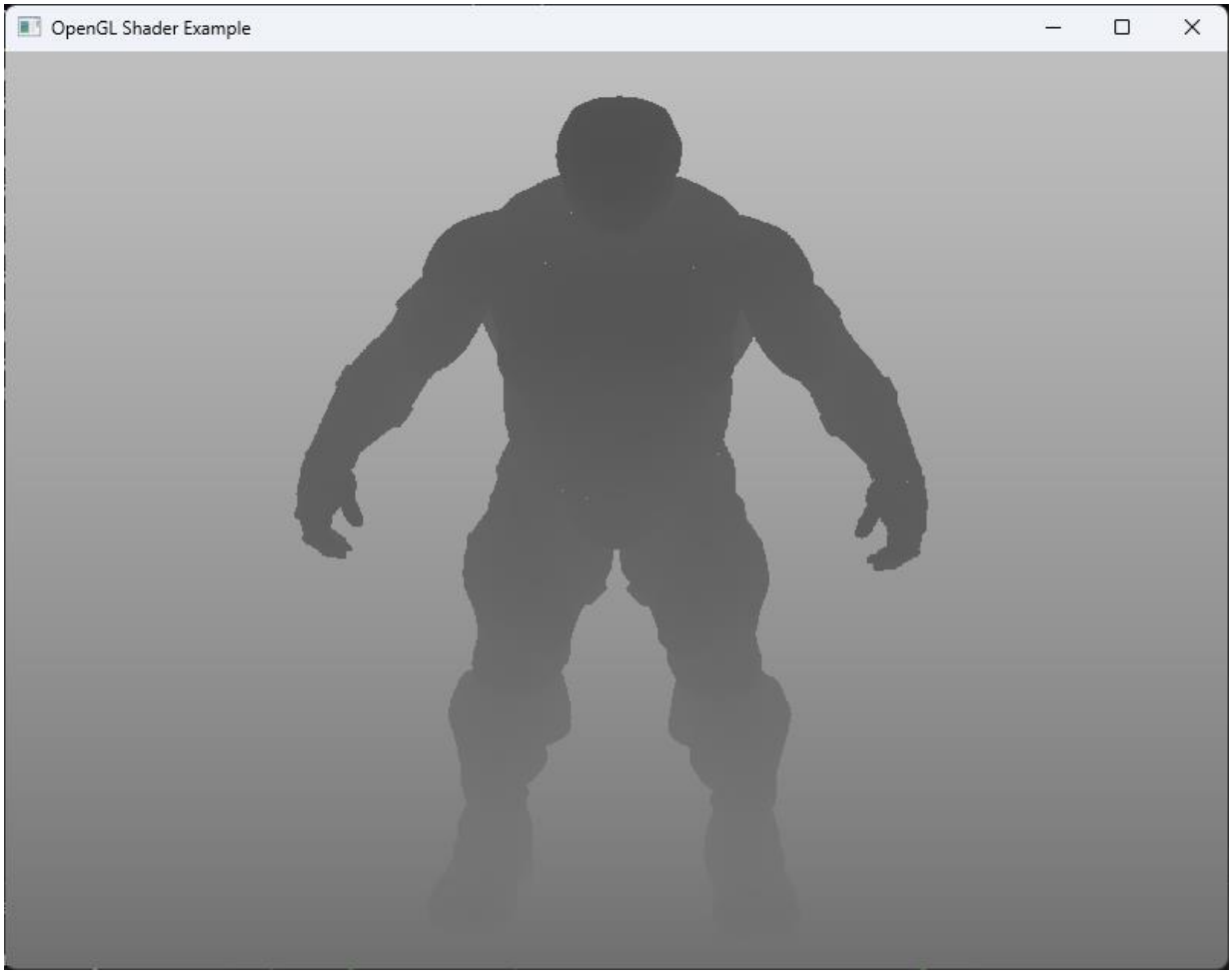

Figura 3 – Exemplu de hartă de adâncime

Calculul real al umbrelor va avea loc în shader-ul fragmentului. Culoarea finală a fiecărui fragment (pentru componente difuze și speculare) va fi modulată cu valoarea de umbră calculate (1.0 sau 0.0)

```
…
float computeShadow()
{
…
}
…
//modulate with shadow
shadow = computeShadow();
vec3 color = min((ambient + (1.0f - shadow)*diffuse) + (1.0f - shadow)*specular, 1.0f);
```

Pentru a efectua calculul efectiv al umbrei, primul lucru pe care ar trebui să-l facem este să transformăm poziția fragmentului în coordonatele normalizate ale spațiului luminii. Deoarece transformarea în spațiul luminii a fost făcută în vertex shader, rămâne să aplicăm diviziunea perspectivă pentru a obține coordonatele normalizate:

```
in vec4 fragPosLightSpace;

…
float computeShadow()
{
…
// perform perspective divide
vec3 normalizedCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
…
}
…
```

Aceasta returnează poziția fragmentului curent în intervalul [-1,1]. Deoarece valorile adâncimii fragmentului sunt în intervalul [0,1], ar trebui să transformăm coordonatele normalizate în consecință:

```
…
float computeShadow()
{
…
// Transform to [0,1] range
normalizedCoords = normalizedCoords * 0.5 + 0.5;
…
}
…
```

Acum putem eșantiona adâncimea existentă în harta de adâncime folosind aceste coordonate, deoarece acestea corespund acum cu coordonatele NDC (Normalized Device Coordinates) din prima trecere:

```
…
uniform sampler2D shadowMap;
…
float computeShadow()
{
…
// Get closest depth value from light's perspective
float closestDepth = texture(shadowMap, normalizedCoords.xy).r;
…
```

```
}
…
```

Acest lucru ne oferă cea mai apropiată valoare de adâncime din perspectiva luminii. Apoi luăm adâncimea (în sistemul de referință al luminii) fragmentului curent:

```
…
float computeShadow()
{
…
// Get depth of current fragment from light's perspective
float currentDepth = normalizedCoords.z;
…
}
…
```

Acum putem compara cele două valori. Dacă adâncimea fragmentului curent este mai mare decât valoarea din harta adâncimii, fragmentul curent este în umbră. Altfel, este iluminat:

```
…
float computeShadow()
{
…
// Check whether current frag pos is in shadow
float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;
…
}
…
```

Atunci când utilizați acest shader, cu textura de adâncime corect legată, ieșirea trebuie să fie similară celei din figura 4.

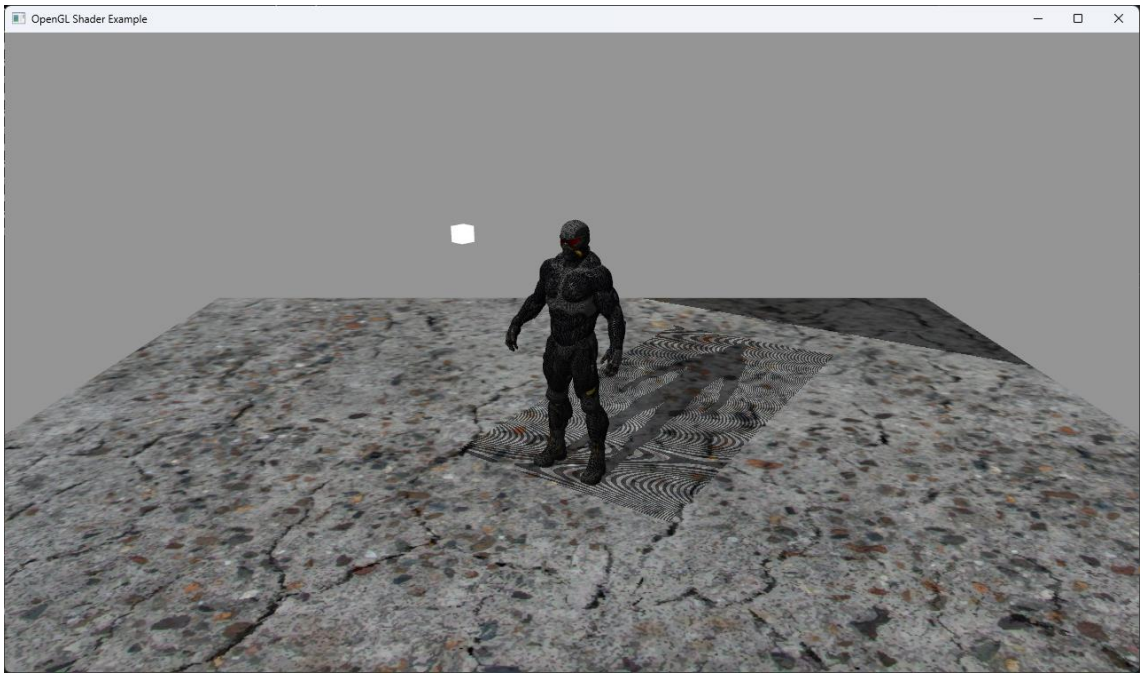

Figura 4 – Rasterizare cu umbre

2.3 Îmbunătățirea calității

2.3.1 Shadow acne

Privind mai atent, putem observa un model nedorit, așa cum este ilustrat în Figura 5.

Figura 5 – Shadow acne

Acest lucru se întâmplă din cauza rezoluției limitate a hărții de adâncime și apare atunci când mai multe fragmente eșantionează aceeași valoare din harta de adâncime. De obicei, putem rezolva această problem folosind o mică modificare la adâncimea fragmentului, fie o valoare fixă, fie o valoare bazată pe unghiul

pe care suprafața îl face cu direcția luminii (aceasta rezolvă problemele pentru suprafețe care au un unghi abrupt față de lumină):

```
…
float computeShadow()
{
…
// Check whether current frag pos is in shadow
float bias = 0.005f;
float shadow = currentDepth - bias > closestDepth  ? 1.0f : 0.0f;
…
}
…
```

sau

```
…
float computeShadow()
{
…
// Check whether current frag pos is in shadow
float bias = max(0.05f * (1.0f - dot(normal, lightDir)), 0.005f);
float shadow = currentDepth - bias > closestDepth  ? 1.0f : 0.0f;
…
}
…
```

Ieșirea vizuală ar trebui să devină acum similară cu cea din Figura 6.

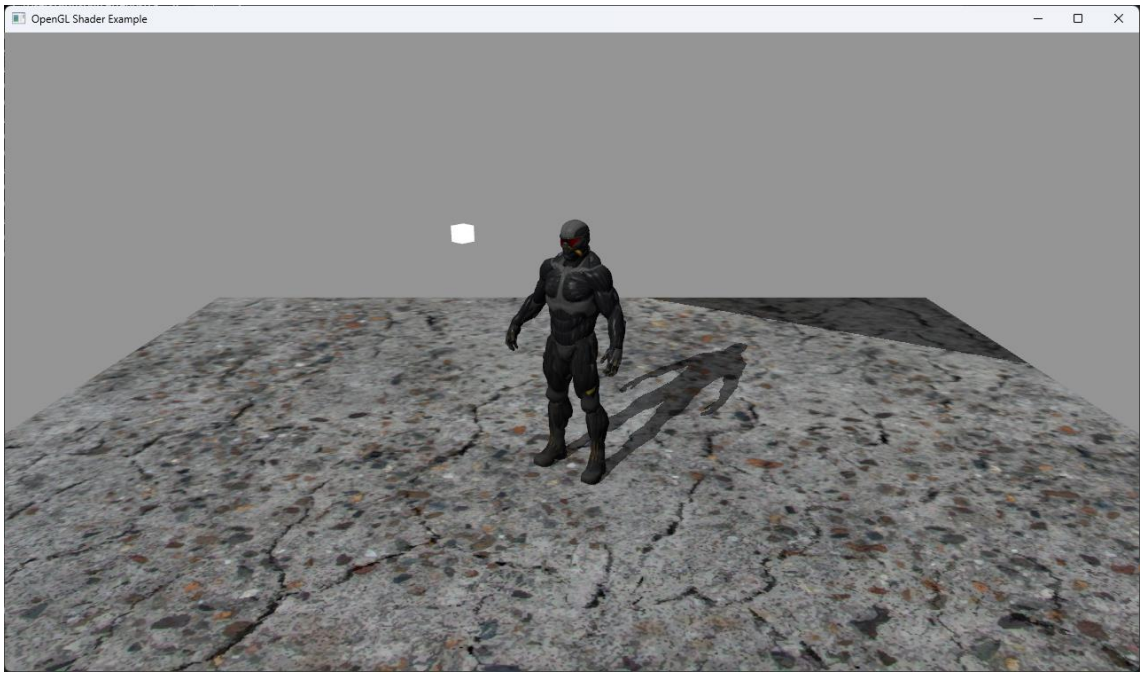

Figura 6 – Fără shadow acne

2.3.2 Supra-eșantionare

Observați că există zone în umbră care ar trebui să fie iluminate (colțul din dreapta din Figura 6). Acest lucru se întâmplă deoarece fragmentele în afara frustumului hărții de adâncime au o valoare a adâncimii mai mare de 1,0. Acest lucru poate fi ușor rectificat în fragment shader:

```
…
float computeShadow()
{
…
if (normalizedCoords.z > 1.0f)
        return 0.0f;
…
}
…
```

Ieșirea vizuală ar trebui să devină acum similară cu cea din Figura 7.

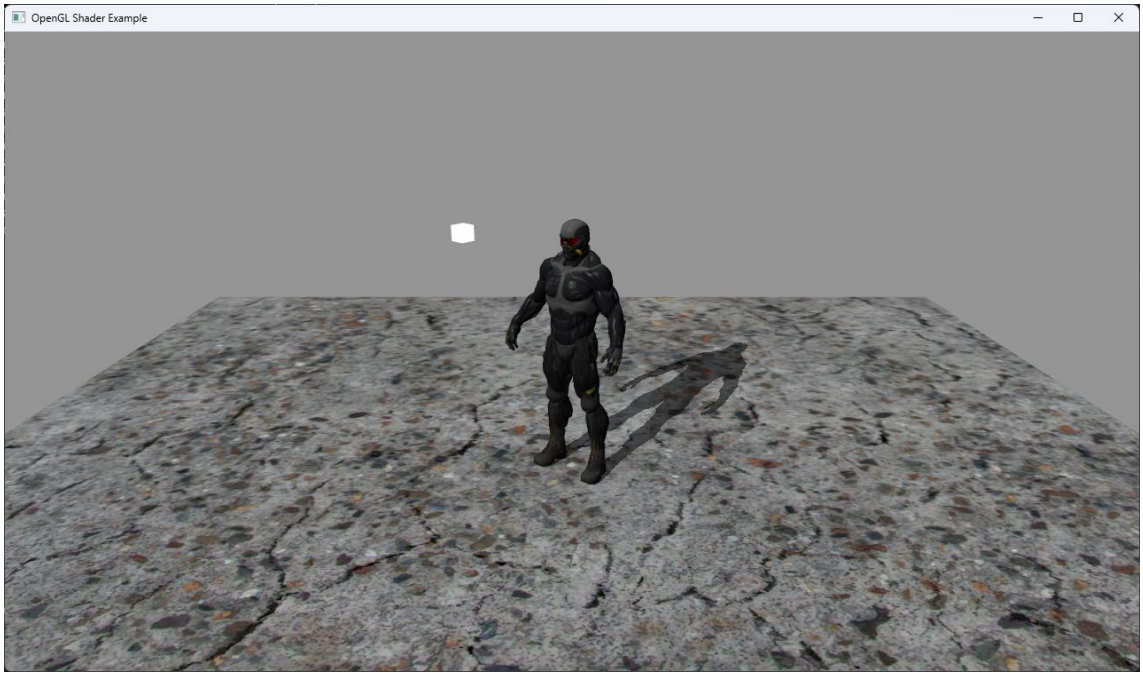

Figure 7 – Fără supra-eșantionare

3 Lectură suplimentară

- OpenGL Programming Guide 8th Edition – Capitolul 7 (Shadow Mapping)
- OpenGL tutorials – Advanced OpenGL (Depth testing, Face culling, Framebuffers), Advanced lighting (Shadows) – http://www.learnopengl.com/

4 Temă

1. Descărcați resursele de pe site

2. Adăugați obiectul sol în scenă
3. Adăugați un obiect framebuffer la aplicația existentă
4. Adăugați o textură de adâncime la obiectul framebuffer
5. Adăugați codul și shaderele necesare pentru a rasteriza harta de adâncime a luminii în textura de adâncime
6. Adăugați codul necesar pentru a rasteriza scena cu umbre
7. Efectuați ajustările necesare pentru a rezolva diferitele probleme vizuale întâlnite (shadow acne umbră, supra-eșantionare)