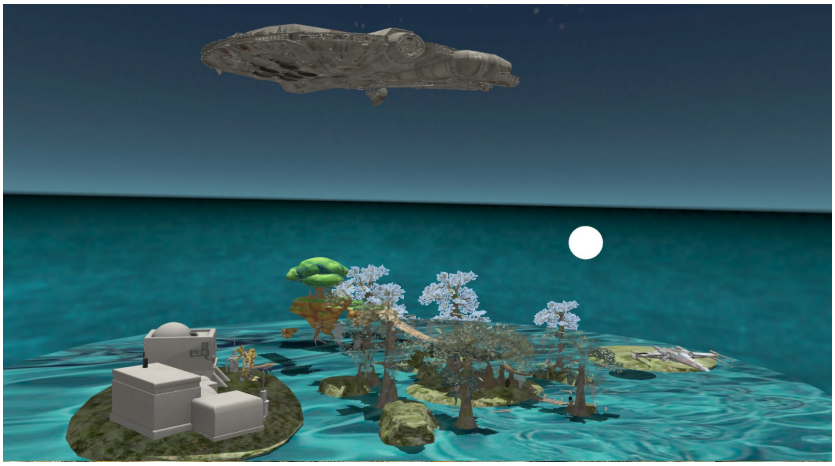
Documentație Proiect Grafică OpenGL

Student: Padawan Muresan Davide-Andrei
Profesor: Master Jedi Adrian Sabou

Data predării: 13.01.2025



Cuprins

1 Prezentarea temei

Proiectul presupune realizarea unei aplicații 3D utilizând biblioteca OpenGL. Acesta include încărcarea unor modele complexe, generarea umbrelor prin *Shadow Mapping*, și implementarea unor efecte precum ceață și iluminare. Scenele sunt interactive și permit navigarea utilizatorului prin intermediul tastaturii și al mouse-ului. Fiecare obiect este texturat, iar lumina provine de la mai multe surse de lumina avand 2 una directionala si alta pozitionala.Scena este plasata in universul studiourilor DreamWorks si Disney, un mix unic intre StarWars si Kung Fu Panda aducand personajele copilariei la viata.

2 Scenariul

Initial, utilizatorul se afla chiar in centrul scenei, dupa care se poate deplasa folosind tastele W,A, S, D si privi in jur miscand cursorul mouse-ului si rotind camera. Daca se apasa tasta "P" de oriunde din scena, va incepe un tur automat al scenei, si deplasandu-se cativa pasi in stanga si inainte pana se ajunge intr-o alta zona. Daca se apasa repetat tasta "P", utilizatorul i se opreste prezentarea si poate sa o reia apasand din nou.

2.1 Implementarea camerei cu quaternioni

Camera proiectului este implementată utilizând quaternioni, o metodă eficientă și stabilă pentru a reprezenta rotațiile 3D. Acest lucru elimină problemele asociate cu utilizarea matricilor Euler, cum ar fi efectul de "gimbal lock". În cadrul proiectului, quaternionii sunt folosiți pentru a calcula orientarea camerei și pentru a genera matricea de vizualizare.

2.1.1 Construcția camerei

Camera este inițializată prin constructorul din clasa `Camera`, care primește următorii parametri:

- `cameraPosition`: poziția camerei în spațiu.
- `cameraTarget`: punctul către care privește camera.
- `cameraUp`: vectorul "up" al camerei, care determină orientarea verticală.

Quaternionul `orientation` este inițializat ca identitate:

$$\texttt{orientation} = \text{glm::quat}(1.0f, 0.0f, 0.0f, 0.0f)$$

Astfel, camera este orientată inițial fără nicio rotație.

2.1.2 Generarea matricei de vizualizare

Matricea de vizualizare este generată folosind funcția `glm::lookAt`. Direcțiile `forward` și `up` sunt calculate aplicând quaternionul asupra vectorilor de bază:

$$\texttt{forward} = \texttt{orientation} \cdot \text{glm::vec3}(0.0f, 0.0f, -1.0f)$$

$$\texttt{up} = \texttt{orientation} \cdot \text{glm::vec3}(0.0f, 1.0f, 0.0f)$$

Punctul spre care privește camera este determinat ca suma dintre poziția camerei și direcția `forward`:

$$\texttt{target} = \texttt{cameraPosition} + \texttt{forward}$$

```
glm::mat4 Camera::getViewMatrix() {
    glm::vec3 forward = this->orientation * glm::vec3
       (0.0f, 0.0f, -1.0f);
    glm::vec3 up = this->orientation * glm::vec3(0.0f,
       1.0f, 0.0f);
    return glm::lookAt(this->cameraPosition,
                       this->cameraPosition + forward,
                       up);
}
```

2.1.3 Mișcarea camerei

Camera permite mișcarea în patru direcții: înainte, înapoi, la stânga și la dreapta. Direcțiile `forward` și `right` sunt calculate aplicând quaternionul asupra vectorilor de bază:

$$\texttt{forward} = \texttt{orientation} \cdot \text{glm::vec3}(0.0f, 0.0f, -1.0f)$$

$$\texttt{right} = \texttt{orientation} \cdot \text{glm::vec3}(1.0f, 0.0f, 0.0f)$$

Poziția camerei este actualizată în funcție de direcția mișcării și de viteza specificată:

```
void Camera::move(MOVE_DIRECTION direction, float speed)
    {
    glm::vec3 forward = this->orientation * glm::vec3
       (0.0f, 0.0f, -1.0f);
    glm::vec3 right = this->orientation * glm::vec3(1.0f
       , 0.0f, 0.0f);

    switch (direction) {
        case MOVE_FORWARD:
            this->cameraPosition += speed * forward;
            break;
        case MOVE_BACKWARD:
            this->cameraPosition -= speed * forward;
            break;
        case MOVE_LEFT:
            this->cameraPosition -= speed * right;
            break;
```

```
        case MOVE_RIGHT:
            this->cameraPosition += speed * right;
            break;
        default:
            break;
    }
}
```

2.1.4 Rotația camerei

Rotația este realizată prin compunerea quaternionilor generați pentru fiecare axă. Pentru fiecare rotație (roll, pitch, yaw), se construiește un quaternion folosind funcția `glm::angleAxis`:

$$\texttt{qPitch} = \texttt{glm::angleAxis}(\text{pitchRad}, \text{glm::vec3}(1.0f, 0.0f, 0.0f))$$

$$\texttt{qYaw} = \texttt{glm::angleAxis}(\text{yawRad}, \text{glm::vec3}(0.0f, 1.0f, 0.0f))$$

$$\texttt{qRoll} = \texttt{glm::angleAxis}(\text{rollRad}, \text{glm::vec3}(0.0f, 0.0f, 1.0f))$$

Noua orientare este calculată compunând rotațiile:

$$\texttt{newOrientation} = \texttt{qYaw} \cdot \texttt{qPitch} \cdot \texttt{qRoll}$$

Aceasta este aplicată asupra orientării existente:

$$\texttt{orientation} = \texttt{newOrientation} \cdot \texttt{orientation}$$

În final, quaternionul este normalizat pentru a evita acumularea erorilor numerice:

```
void Camera::rotate(float roll, float pitch, float yaw)
   {
    float rollRad = glm::radians(roll);
    float pitchRad = glm::radians(pitch);
    float yawRad = glm::radians(yaw);

    glm::quat qPitch = glm::angleAxis(pitchRad, glm::
       vec3(1.0f, 0.0f, 0.0f));
    glm::quat qYaw = glm::angleAxis(yawRad, glm::vec3
       (0.0f, 1.0f, 0.0f));
```

```
    glm::quat qRoll = glm::angleAxis(rollRad, glm::vec3
        (0.0f, 0.0f, 1.0f));

    glm::quat newOrientation = qYaw * qPitch * qRoll;
    this->orientation = newOrientation * this->
        orientation;
    this->orientation = glm::normalize(this->orientation
        );
}
```

Această implementare asigură o rotație fluidă și precisă a camerei, evitând problemele comune ale altor metode de reprezentare.

2.2 Descrierea scenei și a obiectelor

Scena proiectului reprezintă un mediu fantastic, unde utilizatorul poate vedea renumitele nave a rezistentei **Millennium Falcon** și **X-Wing**. Acestea sunt plasate într-un cadru animat, cu un *Skybox* detaliat. Mai la dreapta se observa niste insule plutitoare pe care se afla Razboinicul Dragon PO impreuna cu legendarul maestru Oogway. In spatele mlastinii se afla o mica bucatica din taramurile Tatooine in care s-a spawnat si Bumblebee care asteapta Decepticonii sa apara si sa ii nimiceasca!

2.3 Funcționalități

- Navigarea în scenă utilizând tastele `W`, `A`, `S`, `D`.
- Activarea/dezactivarea ceții prin tasta `F`. Scena de obiecte este implementata cu un efect de ceata de densitate 0.05, astfel incat lucrurile indepartate de pozitia de vizualizare apar mai neclare decat cele apropiate.
- Modificarea luminii globale cu tastele `J` și `L`.
- Vizualizarea hărții de adâncime prin tasta `M`.
- Schimbare pentru a vedea harta sub influenta luminii directionale sau a celei pozitionale apasand `Z`.
- Activează modul de afișare `GL_POINT`, în care geometria obiectelor este reprezentată sub forma punctelor care alcătuiesc mesh-ul. Acest mod oferă o vizualizare minimalistă, utilă pentru analiza punctelor din structura obiectelor apasand `V`.
- Activează modul de afișare `GL_LINE`, în care toate suprafețele obiectelor sunt afișate sub forma unor contururi (wireframe). E bun pentru a observa structura obiectelor din scenă apasand `B`.
- Activează modul de afișare `GL_FILL`, care afișează obiectele complet umplute cu materialele și texturile lor. Acesta este modul standard de afișare utilizat pentru vizualizarea finală apasand `N`.

3 Detalii de implementare

3.1 Funcții și algoritmi

-Unul dintre algoritmii esențiali implementați este cel pentru *Shadow Mapping*. Acest proces implică generarea unei hărți de umbre, pe baza căreia este calculată iluminarea fragmentelor.Acest algoritm presupune trasarea scenei de 2 ori, prima data efectuandu-se calculul umbrelor, iar la a doua iterare, randarea completa a scenei, cu tot cu umbre.
Mai mult, se seteaza ca punct de vizualizare pozitia luminii, dupa care se salveaza o harta de umbre pe baza informatiilor de adancime ale fiecarui fragment din scena. Aceasta harta este transformata si stocata intr-o textura.
In continuare, se plaseaza camera in pozitia originala a observatorului pentru rasterizarea finala a scenei. Pentru a decide daca un fragment e umbrit sau nu se compara adancimea fiecarui fragment cu cea stocata in harta de adancime. O valoare mai mare inseamna umbra, iar una mai mica - lumina directa.

-Un alt algoritm este cel utilizat la calculul cetii exponentiale. Acesta presupune reducerea intensitatii luminii in functie de distanta dintre fragment si lumina. Atenuarea este reprezentata de densitatea de ceata, constanta in toata scena. De asemenea, efectul de ceață este realizat prin intermediul unei formule exponențiale:

$$\text{fogFactor} = \exp\left(-\text{distance}^2 \cdot \text{density}\right)$$

3.2 Implementarea eliminării fragmentelor pentru frunze

Pentru a reprezenta frunzele din scenă în mod realist, s-a implementat o tehnică de eliminare a fragmentelor (*fragment discarding*), bazată pe valorile de transparență (*alpha*) ale texturii aplicate. Această tehnică permite eliminarea completă a fragmentelor care nu contribuie vizual la scenă, reducând astfel costurile de calcul și îmbunătățind realismul.

Principiul eliminării fragmentelor

Texturile utilizate pentru frunze includ un canal *alpha*, care determină transparența fiecărui fragment. Fragmentul este eliminat (`discard`) atunci când valoarea sa de *alpha* este sub un prag specific, în cazul nostru, 0.1. Această abordare este esențială pentru obiecte precum frunzele, care au margini neregulate și un fundal transparent.

Codul relevant din shader este următorul:

```
// Cite te pixelul din textura RGBA
vec4 texColor = texture(diffuseTexture, fTexCoords);

// Elimin  fragmentele cu alpha sub 0.1
if (texColor.a < 0.1) {
    discard;
}
```

3.3 Modelul Grafic

Proiectul are drept scop realizarea unei prezentari fotorealiste a unor scene de obiecte 3D utilizand librariile OpenGl, GLFW si GLM, astfel incat aplicatia respecta pipeline-ul grafic clasic pentru a rasteriza scena. Obiectele definite in coordonate globale trec prin vertex shader, apoi dupa operatii intermediare de transformare (de vizualizare si proiectie) prin fragment shader, ca in final sa poata fi rasterizati pixelii de pe ecran cu culorile calculate pentru fiecare fragment.

Pentru implementarea modelului grafic, s-a utilizat modelul de iluminare Phong. Modelul Phong permite calculul iluminării ambientale, difuze și speculare pentru fiecare fragment, simulând astfel efectele naturale ale luminii asupra suprafețelor.

Modelul Phong constă în trei componente principale:

- **Lumina ambientală:** Reprezintă lumina uniformă din scenă, simulând reflexia difuză a luminii pe suprafețe.
- **Lumina difuză:** Calculată pe baza unghiului dintre direcția luminii și normala fragmentului, aceasta creează efectul de iluminare dependent de poziția luminii.

- **Lumina specular**: Reprezintă reflexiile concentrate ale luminii, calculând un punct strălucitor bazat pe direcția privitorului.

Aceste componente sunt combinate pentru a obține culoarea finală a fiecărui fragment:

```
// Calculul componentei ambientale
ambient = ambientStrength * lightColor;

// Calculul componentei difuze
diffuse = max(dot(normalEye, lightDirN), 0.0f) *
   lightColor;

// Calculul componentei speculare
vec3 reflectDir = reflect(-lightDirN, normalEye);
float specCoeff = pow(max(dot(viewDir, reflectDir), 0.0f
   ), 32);
specular = specularStrength * specCoeff * lightColor;
```

Iluminarea direcțională și pozițională

Implementarea suportă atât iluminarea direcțională, cât și pozițională. Iluminarea direcțională folosește un vector de direcție fix, în timp ce iluminarea pozițională depinde de distanța și direcția față de o sursă de lumină specifică.

Pentru iluminarea pozițională, se aplică o funcție de atenuare bazată pe distanța fragmentului de la sursa de lumină:

```
float attenuation = 1.0 / (constant + linear * distance
   + quadratic * distance * distance);
ambientPos *= attenuation;
diffusePos *= attenuation;
specularPos *= attenuation;
```

Rezultatul final

Culoarea finală a fiecărui fragment este calculată combinând componentele de iluminare și textura:

```
vec3 color = (ambient + (1.0f - shadow) * diffuse) *
   texColor.rgb
            + (1.0f - shadow) * specular * specularTex;
```

Prin această implementare, modelul Phong oferă o iluminare realistă, incluzând efecte precum umbre, reflexii etc.

3.4 Structuri de date și ierarhia de clase

Codul proiectului este organizat în clase precum:

- `Camera`: Responsabilă pentru mișcarea și orientarea camerei.
- `Model3D`: Încarcă și afișează modelele 3D.
- `Shader`: Gestionează compilarea și utilizarea shader-elor.

4 Manual de utilizare

La subsectiunea **Functionalitati** este explicat in detaliu ce face fiecare tasta. Astfel, ultilizatorul putand sa interactioneze cu scena.

5 Concluzii și dezvoltări ulterioare

Acest proiect demonstrează eficiența utilizării GPU-ului pentru redarea scenelor 3D complexe. Dezvoltările ulterioare includ:

- Adăugarea mai multor animații pentru obiectele 3D.
- Îmbunătățirea detaliilor Skybox-ului.
- Extinderea efectelor de iluminare și umbre.

Referințe

1. *Cubemaps Skyboxes by Victor Gordon* `https://www.youtube.com/watch?v=8sVvxeKI9Pk`
2. *Laboratorul 6/PG - Texturi și animații simple. + Putin ajutor de la Master Jedi Nandra* `https://www.youtube.com/playlist?list=PLrgcDEgRZ_kndoWmRkAK4Y7ToJdOf-OSM`
3. *Laboratorul 78/PG - Lumini si Modelul de iluminare Phong.*
4. *Laboratorul 9/PG - Shadow Mapping.*
5. *Laboratorul 12/PG - Fog and fragment discarding.*
6. *Multiple lights in OpenGL*, URL: `https://opentk.net/learn/chapter2/6-multiple-lights.html`.