PROIECT PROCESARE GRAFICA

Toader Eric-Stefan

Realizarea unui proiect folosind libraria de procesare grafica OpenGL

2022/2023

Profesor indrumator: Cosmin Ioan Nandra

Cuprins

Prezentarea temei

Proiectul presupune dezvoltarea unei aplicatii desktop in limbajul C++ folosind libraria de procesare grafica OpenGL pentru a crea o scena complexa.

Scena ar trebui sa contina o colectie de obiecte texturate, iluminate de diferite tipuri de lumina si umbrite de alte obiecte. In plus, in cadrul programului rezultat trebuie sa existe si animatii care sa fluidizeze lumea si sa ii dea viata, precum si alte efecte.

Cerintele proiectului in intregime sunt:

- vizualizarea scenei: scalare, translație, rotație, mișcarea camerei
    - utilizând tastatura sau mausul
    - utilizând animații de prezentare
- specificarea surselor de lumina (cel puțin două surse de lumină diferite)
- vizualizare scenă în modurile solid, wireframe, poligonal și smooth
- maparea texturilor și definirea materialelor
    - calitatea texturilor și nivelul de detaliu al acestora
    - maparea texturilor pe obiecte
- exemplificarea generării umbrelor
- exemplificarea animării diferitelor componente ale obiectelor
- fotorealism, complexitatea scenei, nivelul de detaliere al modelării, dezvoltarea diferiților algoritmi și implementarea acestora (generare dinamică de obiecte, detecția coliziunilor, generarea umbrelor, ceață, ploaie, vânt), calitatea animațiilor, utilizarea diferitelor surse de lumină (globală, locală, de tip spot)

Scenariul

Descrierea scenei si a obiectelor

Am ales sa reprezint o colectie de trei insule care se afla la randul lor pe o insula plutitoare ce contine o mare de apa care se scurge prin intermediul unor cascade situate la periferiile scenei.

Prima insula contine mai multe case si felinare, pe langa alte obiecte marunte, care reprezinta un sat. A doua insula este locul unde se afla casa unui pirat, mai multe tunuri si un turn de paza. Pe tarmul celei de-a treia insule se afla mai multe barci scufundate si sulite infipte in nisip sau aruncate pe jos, ce ar putea indica faptul ca a avut loc o lupta dintre civilizatia primei insule si creaturile scheletice ce au iesit la iveala din interiorul piramidei.

De asemenea, in scena se mai afla si o corabie ce ii apartine piratului care detine si casa de pe cea de-a doua insula. Aceasta corabie poata fi folosita la navigare.

Aceasta scena beneficiaza si de efectul unui ciclu zi-noapte continuu, care influenteaza iluminarea in scena si face lumea sa prinda viata, impreuna cu celelalte efecte si animatii.

Functionalitati

Functionalitatea principala este faptul ca este posibila navigarea in totalitate a corabiei piratului (pe suprafata apei). Aceasta nu se poate naviga in exteriorul insulei plutitoare si nu se poate ciocni de alte obiecte din scena. La schimbarea traiectoriei corabiei, carma acesteia se va roti in corcondanta.

Exista un dragon scheletic care zboara deasupra scenei in continuu, in directii aleatoare, fara sa iasa din perimetrul insulei plutitoare.

De asemenea, jucatorul are puterea de a accelera ciclul zi-noapte, ceea ce este util pentru a evidentia umbrele si luminile punctiforme si directionale.

Pentru a ajuta la formarea ambientei, se vor reda sunete de ocean ale caror volum va fluctua in functie de distanta dintre jucator si marea de apa.

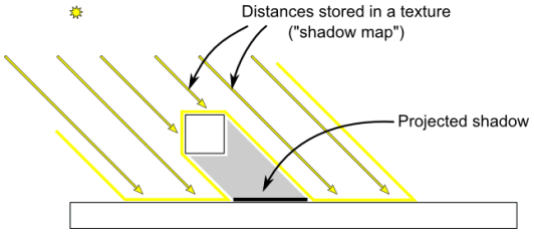
Detalii de implementare

Functii si algoritmi

Generarea umbrelor

Pentru generarea umbrelor am folosit algoritmul de shadow mapping prezentat in cadrul laboratorului de procesare grafica. Acesta are la baza randerizarea scenei de obiecte din perspectiva sursei de lumina pentru a genera o harta de adancimi, care mai apoi este citita de shader-ul folosit pentru a randeriza obiectele din scena pentru a desena umbrele.

Harta shadow map contine valori de adancime Z in relatie cu sursa de lumina. Daca un fragment are un Z mai mare decat cea mai mica valoare din acea pozitie, atunci fragmentul va fi umbrit. In caz contrar, fragmentul va fi iluminat direct de sursa de lumina, nefiind obstructionat de altele, deci se va bucura de lumina din plin.



Detectia coliziunilor

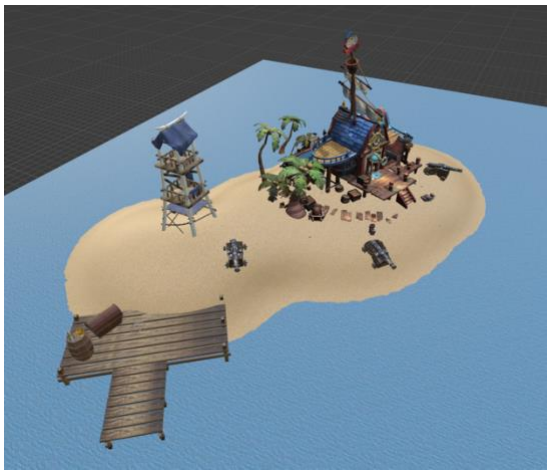
A fost necesar sa implementez un algoritm de detectie a coliziunilor pentru a ma asigura ca barca nu se va ciocni de vreo insula atunci cand este navigata de jucator.

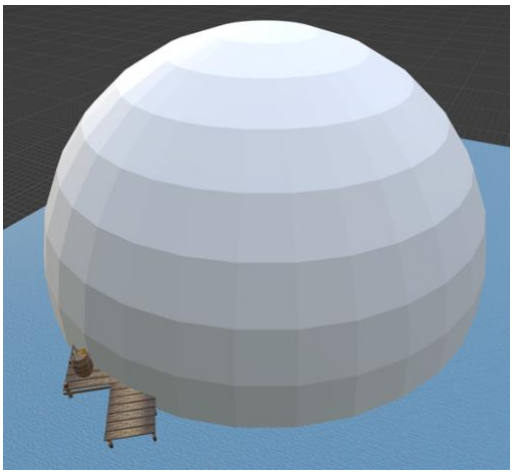
Exista o multime de algoritmi care se ocupa de detectia coliziunilor, precum:

- Verificarea coordonatelor 2D a obiectului in relatie cu alte obiecte
- Implementarea unor bounding boxes care sa delimiteze obiectele
- Implementarea unor bounding spheres care sa delimiteze obiectele

In proiect am folosit verificarea coordonatelor 2D (X si Z) pentru a ma asigura ca nava nu poate parasi scena si pentru a-l face pe dragon sa se intoarca inapoi in scena.

De asemenea, am implementat detectia de coliziuni utilizand bounding spheres care inconjoara insulele si corabia. La fiecare miscare inainte sau inapoi a corabiei, se verifica daca aceasta este sau nu o miscare legala/ posibila. Fiecare bounding sphere este caracterizat de o origine si o raza. Cand doua bounding spheres se ciocnesc, distanta de la obiecte este mai mica decat suma razelor celor doua bounding spheres asociate. In cazul corabiei, bounding sphere-ul asociat este translatat impreuna cu aceasta.





```
bool BoundingSphere::isColliding(BoundingSphere other) {
    if (glm::length(objectPosition - other.objectPosition) < radius + other.radius) return true;
    else return false;
}

void BoundingSphere::updateObjectPosition(glm::vec3 objectPosition) {
    this->objectPosition = objectPosition;
}
```

Aceste sfere nu exista in scena cu adevarat, ele sunt niste obiecte fictive care contin date despre obiecte care pot cauza coliziuni.

Animarea dragonului

Pentru a anima miscarea dragonului, l-am descompus pe acesta in trei obiecte separate (corpul si cele doua aripi). Miscarea acestuia prin scena este aleatoare si fluida, pentru a contribui la sentimentul de continuitate creat de catre ciclul zi-noapte.

In plus fata de miscarea aleatoare a dragonului prin scena, mai este prezent un mecanism care nu ii permite acestuia sa se departeze prea tare de insula plutitoare, intorcandu-l din drum cand este pe aproape sa iasa din perimetru.

Odata cu miscarea dragonului, aripile acestuia se rotesc in jurul corpului pentru a simula bataia din aripi. Acest efect a fost realizat printr-o serie de transformari de translatie si rotatie.

```cpp
// Random generator
std::random_device dev;
std::mt19937 rng(dev());

// Dragon object
gps::Model3D dragonBody;
gps::Model3D dragonWingLeft;
gps::Model3D dragonWingRight;
std::uniform_int_distribution<int> dragonAngles(-1, 1);
int lastDragonRotate;
float dragonWingsAngle = 15.f;
bool dragonWingsAscending = false;
```

Variabilele folosite

```cpp
void moveDragonWings() {
    if (dragonWingsAscending) {
        dragonWingsAngle += 0.5f;
    } else {
        dragonWingsAngle -= 0.5f;
    }

    if (dragonWingsAngle > 15.f) {
        dragonWingsAscending = false;
    }

    if (dragonWingsAngle < -15.f) {
        dragonWingsAscending = true;
    }
}
```

Functie ce se ocupa de modificarea unghiului aripilor

```cpp
void moveDragon() {
    int currentRotate;

    // turn around
    if (dragon.getPosition().x > 730) {
        currentRotate = -2;
        goto commit;

    } else if (dragon.getPosition().x < -730) {
        currentRotate = -2;
        goto commit;

    } else if (dragon.getPosition().z > 730) {
        currentRotate = -2;
        goto commit;

    } else if (dragon.getPosition().z < -730) {
        currentRotate = -2;
        goto commit;
    }

    currentRotate = dragonAngles(rng);
    goto commit;

    commit:
    dragon.move(etoader::MOVE_FORWARD);
    dragon.rotate(0.005f * currentRotate);
    lastDragonRotate = currentRotate;
}
```

Functie ce se ocupa de miscarea aripilor

```cpp
model = glm::mat4(1.0f);
model = glm::translate(model, dragon.getPosition());
model = glm::rotate(model, dragon.getYawAngle(), glm::vec3(0, 1, 0));
model = glm::rotate(model, glm::radians(90.f), glm::vec3(0, 1, 0));
model = glm::translate(model, glm::vec3(-7.f, 10.f, 40.f));
model = glm::rotate(model, glm::radians(-dragonWingsAngle), glm::vec3(0, 0, 1));
model = glm::translate(model, glm::vec3(7.f, -10.f, -40.f));

/* ... */

dragonWingLeft.Draw(shader);
```

Desenarea obiectului

In cele ce urmeaza voi explica transformarile efectuate in poza de mai sus (directia de parcurgere a liniilor este de jos in sus):

- Translatia aripii in origine (aceasta este deplasata fata de origine)
- Rotatia acesteia pe axa Z in functie de o variabila precalculata
- Translatia acesteia inapoi in pozitia initiala
- Rotirea acesteia pe axa Y cu 90 de grade (deoarece rotesc si corpul dragonului in acelasi fel)
- Rotirea aripii pe axa Y in functie de unghiul de rotatie al corpului
- Translatia la pozitia corpului (aripa este in pozitia buna deoarece ea vine ca obiect deja la un offset care reprezinta dinstanta de la originea dragonului la originea aripii

Animarea camerei de prezentare

Pentru animarea camerei am ales sa folosesc curbe Bezier pentru generarea traiectoriei. Astfel, animatia este fluida si complexa, in comparatie cu tehnica ecuatiei parametrice a dreptei.

Asadar, am extins functionalitatea camerei pentru a permite blocarea acesteia si deplasarea in functie de anumite puncte de control date, si am folosit o functie care sa interpoleze trei puncte pentru a crea urmatoarea pozitie a camerei. Pentru a ma asigura ca si camera este indreptata in sensul de miscare, am atribuit vectorului cameraFrontDirection diferenta dintre pozitia urmatoare a camerei si pozitia actuala.

```
std::vector<glm::vec3> points = {
    glm::vec3(-750.f, 20.f, 750.f),
    glm::vec3(-250.f, 20.f, 630.f),
    glm::vec3(-190.f, 30.f, 510.f),
    glm::vec3(-480.f, 50.f, 370.f),
    glm::vec3(-410.f, 50.f, 135.f),
    glm::vec3(-350.f, 50.f, -487.f),
    glm::vec3(360.f, 250.f, -300.f),
    glm::vec3(600.f, 100.f, 120.f),
    glm::vec3(430.f, 50.f, 340.f)
};

glm::vec3 Bezier3Points(float t, glm::vec3 P0, glm::vec3 P1, glm::vec3 P2) {
    return (1 - t) * (P0 * (1 - t) + P1 * t) + (P1 * (1 - t) + P2 * t) * t;
}

glm::vec3 getNextPosition(float t, int pos) {
    return Bezier3Points(t, points[pos], points[pos + 1], points[pos + 2]);
}

bool isValidIndex(int pos) {
    if (pos > points.size() - 3) {
        return false;
    }
    return true;
}

/* ... */

glm::vec3 nextPosition = getNextPosition(t, pos);
cameraFrontDirection = nextPosition - cameraPosition;
cameraPosition = nextPosition;
```

Tot acest mecanism este abstractizat in clasele si functiile de business logic, care nu sunt vizibile in fisierul main.cpp. Starea interna a camerei poate avea modul FREE_LOOK in care utilizatorul este liber sa manevreze camera in orice fel utilizand controalele, si modul PRESENTING, care respinge orice tentativa de a misca camera prin scena. Odata intrata in modul PRESENTING, camera va parasi acest mod automat atunci cand animatia de prezentare ia sfarsit, sau in orice moment in care doreste utilizatorul, apasand aceeasi tasta folosita sa declanseze prezentarea.

Pentru a creste performanta si a separa logica aplicatiei mai bine, lansez un thread aditional atunci cand camera se afla in modul PRESENTING, care se ocupa cu calcularea pozitiei urmatoare a camerei si a directiei de vizualizare si actualizarea starii interne a obiectului camera, ale carei date sunt apoi citite de catre thread-ul principal pentru a randeriza scena de obiecte.

```
void Camera::toggleMode() {

    if (mode == FREE_LOOK) {

        if (presenterThread.joinable()) {
            presenterThread.join();
        }

        mode = PRESENTING;
        presenterThread = std::thread(startPresenting, std::ref(mode), std::ref(cameraPosition),
            std::ref(cameraFrontDirection), 0.0);
    } else {
        mode = FREE_LOOK;
        presenterThread.join();
    }
}
```

```
void startPresenting(CAMERA_MODE &mode, glm::vec3 &cameraPosition,
    glm::vec3 &cameraFrontDirection, double lastGetTime) {

    float t = 0.0f;
    int pos = 0;

    while (mode == PRESENTING) {

        if (t >= 1.0f) {
            pos += 2;
            t = 0.0f;
        }

        if (!isValidIndex(pos)) break;

        double currentGetTime = glfwGetTime();
        if (currentGetTime - lastGetTime > 1.0/60.0) {

            glm::vec3 nextPosition = getNextPosition(t, pos);
            cameraFrontDirection = nextPosition - cameraPosition;
            cameraPosition = nextPosition;

            t += 0.001;

            lastGetTime = currentGetTime;
        }
    }

    mode = FREE_LOOK;
}
```

Modelul grafic

Obiectele plasate in scena, impreuna cu texturile lor au fost descarcate de pe internet, de pe diverse site-uri ce hosteaza modele 3D gratuite incarcate de catre artisti.

In cazul in care texturile nu au fost mapate corespunzator sau lipseau, m-am folosit de imagini descarcate de pe internet care sa semene cu textura pe care vreau sa o redau obiectelor.

Pentru a creste performanta aplicatiei si a timpului de incarcare a programului, am folosit tehnica Decimate Geometry pe toate obiectele din scena (pana in punctul cand erau

vizibil afectate de schimbare) pentru a micsora numarul de varfuri aflate in compozitia obiectelor, micsorand totodata si dimensiunea fisierelor OBJ.

Structuri de date si modele arhitecturale

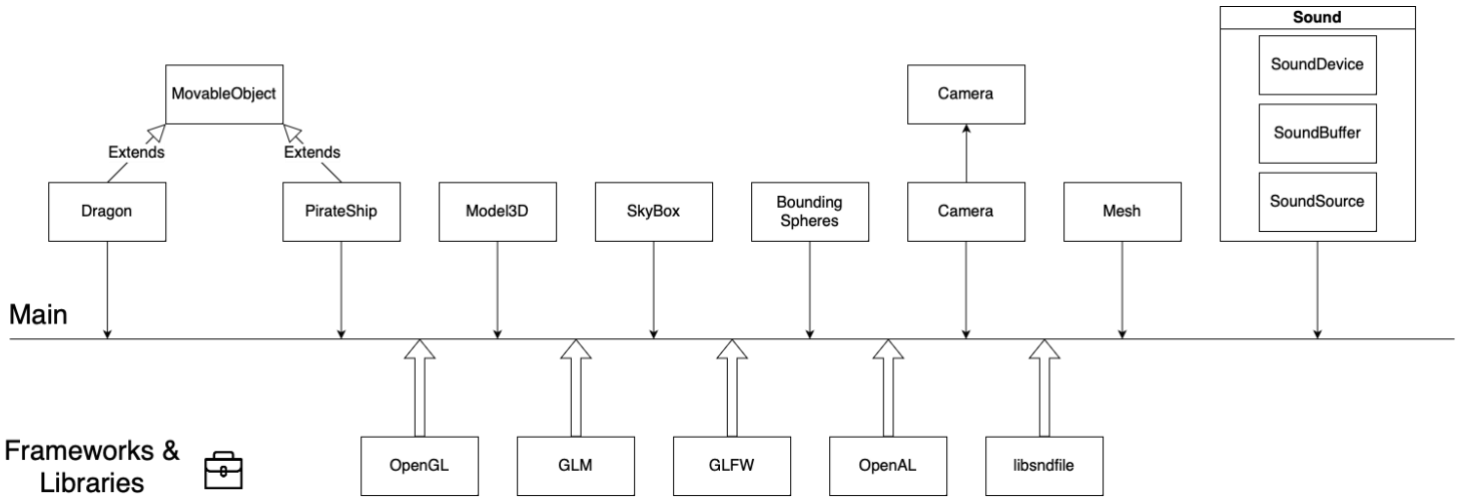
In acest proiect am folosit vectori pentru reprezentarea punctelor de control folosite in calcularea curbelor Bezier ce reprezinta traiectoria camerei de prezentare si pentru stocarea tuturor buffer-elor de sunet folosite pentru a reda sunete cu ajutorul bibliotecii OpenAL (Open Audio Library).

De asemenea, am folosit structura vectoriala pentru a incarca fetele skybox-ului intr-un obiect apartinand clasei SkyBox.

Ca modele arhitecturale, am ales sa folosesc modelul Singleton pentru a reprezenta clasa SoundDevice, deoarece trebuie sa existe un singur obiect care sa se ocupe de redarea sunetului. In viata reala, toate aplicatiile redau sunete printr-un singur device (ar fi enervant ca muzica jocului sa fie redata in casti in timp ce toate efectele de sunet sa fie redate la difuzoare).

Pentru a reduce numarul de linii de cod scris si a nu rescrie cod ce poate fi reutilizat cu usurinta, am ales sa implementez clasa MovableObject care contine toate functiile necesare pentru a misca un obiect, asa cum anticipeaza si numele. Aceasta clasa este mai apoi mostenita de PirateShip si Dragon. Miscarea acestor obiecte este astfel abstractizata.

Ierarhia de clase



Prezentarea interfetei grafice de utilizator/ manual de utilizare

Schimbarea directiei de vizualizare a camerei

Miscarea camerei sau a navei pe planul definit de axele XZ

Scaderea pozitiei camerei pe axa Y

Cresterea pozitiei camerei pe axa Y

Comutarea dintre modurile de camera (prezentare si liber)

Accelerarea ciclului zi-noapte

Comutarea dintre afisarea scenei si a shadow map-ului

Activarea si dezactivarea efectelor de sunet

Comutarea dintre controlul camerei si a navei

Comutarea dintre reprezentarile solid, wireframe si point

Iesirea din aplicatie

Concluzii si dezvoltari ulterioare

Acest proiect a fost o provocare buna pentru mine, impunand multe ziduri pe care a trebuit sa le sparg pentru a putea progresa, dar pe care, in cele din urma, am reusit sa le trec. Dificultatea proiectului la inceput este ridicata, dar odata cu trecerea timpului si dezvoltarea functionalitatilor, aceasta tinde sa scada semnificativ. As spune ca progresul pe care eram capabil sa il fac in cateva ore de lucru a crescut exponential de la o zi la alta. Acest proiect mai reprezinta si o concluzie foarte frumoasa a celor doua materii surori (ECG si PG) pe care le-am considerat interesante si ma bucur ca am reusit in final sa materializez cunostintele acumulate.

In realizarea proiectului, am pornit cu principiul ca voi implementa orice idee imi vine in cap, indiferent ca voi sti cum sau nu voi sti cum la momentul respectiv. Astfel, au luat viata functionalitatea cascadelor, a ciclului zi noapte si functionalitatea de redare a sunetelor.

Cu toate astea, au ramas niste functionalitati pe care nu am apucat sa le implementez, pe care le voi mentiona in cele ce urmeaza:

- Posibilitatea de a lansa proiectile din tunuri
- Collision resolution dintre bounding sphere-ul unui proiectil si restul obiectelor (scalarea unui obiect ce arata ca o explozie, dupa care modificarea alpha-ului pana la transparenta totala, simultan cu generarea unei lumini punctiforme puternice la coordonatele coliziunii)
- Posibilitatea dragonului de a scoate foc (particle effects)
- Afisarea unor prompturi pe ecran in apropierea navei (ex.: Press F to navigate)

Referinte

Pentru a intelge concepte referitoare la OpenGL:

- https://learnopengl.com/Lighting/Multiple-lights
- https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
- Lucrarile de laborator

Pentru a putea folosi libraria OpenAL pentru redarea sunetului:

- https://www.openal.org/documentation/
- https://www.youtube.com/watch?v=kWQM1iQ1W0E&list=PLalVdRk2RC6r7-4zciZ3LKc96ikviw6BS

Pentru a invata despre aplicarea mostenirii si diferite design patterns in C++:

- https://www.geeksforgeeks.org/inheritance-in-c/
- https://refactoring.guru/design-patterns/singleton/cpp/example
- https://www.geeksforgeeks.org/access-modifiers-in-c/

Pentru a invata despre multi-threading in C++:

- https://www.bogotobogo.com/cplusplus/C11/1_C11_creating_thread.php