Etapa 2 – Motoare de grafica 3D in timp real

Task-uri:

1. *Model iluminare, material si texturi*
    a. *Model de iluminare functional*
    b. *Texturare*
    c. *Suport pentru cel putin 1 format de textura clasic*
2. *Importer pentru un format cunoscut de obiecte 3D*
3. *Scene management*
    a. *Organizare ierarhica a obiectelor / entitatilor scenei 3D*
    b. *Mecanism de culling*
    c. *Format XML propriu pentru scena 3D*

1. Model iluminare, material si texturi

a. Model de iluminare functional

Pentru acest task am ales implementarea modelului Phong. Avand 2 perechi de shadere – unul pentru obiecte si unul pentru surse de lumina, implementarea modelului de iluminare se realizeaza in fragment shader-ul obiectelor. Voi preciza in capitolul urmator ce reprezinta mai exact obiectele, insa pe scurt, acestea sunt instante ale clasei Mesh sau Model, adica obiecte create pe baza buffer-elor sau importate din exterior.

In fragment shader-ul obiectelor se creeaza o structura **Material** care are componentele **ambient**(vec3), **diffuse**(vec3), **specular**(vec3) si **shininess**(float). De asemenea, este important de luat in calcul si culoarea luminii – **lightColor**(vec3), pozitia luminii – **lightPos**(vec3), pozitia camerei – **camPos**(vec3). Aceste variabile se trimit ca parametru la o functie definita in clasa Shader numita `InitMaterial()`. Scopul acestei functii este ca aceste variabile sa se trimita la shader, pentru calcule de iluminare ulterioare. Functia se poate apela de fiecare data inaintea desenarii unui nou obiect, daca vrem efecte diverse, sau se poate initializa o singura data pentru toate obiectele, cum am facut eu.

Prima componenta a luminii este *lumina ambientala*, definita prin variabila **ambient** pe CPU si **material.ambient** in shader.

Aceasta este cea mai simpla componenta, si se aplica in modelul phong cu aceeasi valoare cu care este primita in shader.

A doua componenta a luminii este *lumina difuza*, definita prin variabila **diffuse** pe CPU si **material.diffuse** in shader.

Pentru a o calcula avem nevoie in primul rand sa calculam distanta dintre pozitia luminii si pozitia obiectului. Pozitia obiectului era definita si la etapa anterioara pentru calcularea pozitiei fragmentelor prin variabila **objPos**, insa de aceasta data intervine si noua variabila **lightPos** pentru pozitia luminii, intrucat avem nevoie sa vedem de unde provine lumina. In acest mod se calculeaza un vector al traiectoriei luminii prin `lightPos - objPos`.

Mai departe, intervin si normalele care s-au trimis prin buffere la etapa precedenta, intrucat imprastierea luminii difuze tine foarte mult de suprafata pe care aterizeaza lumina. Pentru a determina unghiul sau cosinusul dintre lumina si normala, se aplica formula $\theta = arccos\frac{a \cdot b}{\|a\|\|b\|}$. Normalele (dupa cum presupune si numele lor) sunt de regula normalizate, insa eu am aplicat in shadere o normalizare atat asupra acestora, cat si asupra directiei luminii. In concluzie, numitorul va fi acum 1, deci prin produs scalar dintre normala fragmentului care se deseneaza si directia luminii

obtinem inversul unghiului dintre acestea. Se tine seama ca este vorba de arccosinus, deci cu cat rezultatul de dinainte este mai mare, cu atat unghiul este mai mic. Pentru a obtine puterea luminii difuze, calculam maximul dintre acest rezultat si 0, si deci cu cat unghiul este mai mare, cu atat valoarea rezultata va fi mai mica. Aceasta valoare o voi inmulti cu variabila material.diffuse rezultand valoare luminii difuze pentru fragmentul respectiv.

A treia componenta a luminii este lumina speculara, definita prin variabila **specular** pe CPU si **material.specular** in shader.

Acest tip de iluminare tine cont de directia de vizualizare si vectorul de traiectorie al luminii calculat si anterior.

Lumina speculara este definita prin reflexia catre observator in functia de pozitia lui de observare. In consecinta, trebuie sa calculam vectorul de reflexie al luminii in functie de traiectoria acesteia. Acest lucru este usor de implementat cu functia `reflect()` din GLSL. Ce se intampla in spate este urmatorul lucru: GLSL aplica o proiectie a vectorului directiei luminii de cealalta parte a normalei, astfel incat unghiul dintre proiectie si normala sa fie egal cu unghiul dintre vectorul directiei luminii si normala. `reflect()` returneaza un vector de reflexie care pleaca din acelasi punct ca si primul vector. Din aceasta cauza, directia luminii trebuie inmultita cu -1, altfel s-ar face o reflexie incident la suprafata sursei de lumina.

Avand acum vectorul de reflexie al luminii trebuie sa calculam unghiul dintre acesta si vectorul de la observator catre fragment. Se aplica aceeasi formula ca la lumina difuza, si se obtine $cos\theta$. Aceasta valoare se ridica la puterea specificata de **material.shininess**. Rezultatul se inmulteste cu material.specular si se obtine lumina speculara.

Valoare de shininess, dupa cum sugereaza si numele, implica cat de glossy este obiectul. Cu alte cuvinte, cu cat shininess este mai mare, cu atat obiectul este mai reflectiv, iar lumina reflectata catre observator este mai puternica, dar mai focalizata pe suprafata obiectului. Daca shininess este mai mic, lumina este mai slaba, dar ocupa o zona mai mare pe suprafata obiectului.

Adunand lumina ambientala cu lumina difuza si cu lumina speculara se obtine modelul phong de iluminare, care se inmulteste cu culoarea obiectului si cu culoarea sursei de lumina.

Bineinteles, se poate crea un model phong mai complex decat cel implementat de mine, cu mai multe surse de lumina. Acest lucru ar implica sa se trimita la shader un vector de pozitii si un blend intre culori. Blend-ul este usor de realizat, inmultind valorile intre ele. Procesarea pozitiilor este insa mai complexa din punct de vedere computational, intrucat lumina difuza si speculara se calculeaza pe baza directiei luminii. In acest mod, trebuie calculate mai multe lumini difuze si speculare pentru fiecare sursa de lumina in parte.

b. Texturare

In etapa anterioara am explicat cum se incarca un fisier imagine ca textura in buffere si cum se trimite mai apoi catre shadere. Am mai explicat cum se creeaza coordonatele de textura si cum se trimit catre shadere. La final, am explicat cum pe baza acestor 2 tipuri de informatii se deseneaza fragmentele corespunzator, cum se poate da unei texturi o tenta de imagine sau cum se poate interpola o culoare cu o textura folosind functia `mix()`.

c. Suport pentru cel putin 1 format de textura clasic

Avantajul folosirii unei librarii de incarcare a texturilor, cum a fost **stb_image** la mine este ca aceasta este abstractizata in asa fel incat manipularea texturilor se face in cateva linii de cod. De asemenea, un alt avantaj important e ca aceasta suporta foarte multe tipuri de imagini. In acest proiect eu am folosit fisiere de tip **.JPG** si **.PNG**.

2. Importer pentru un format cunoscut de obiecte 3D

In capitolul anterior am discutat despre „obiecte”, dar nu am precizat inca la ce ma refer. Construirea de entitati am facut-o pana acum in integritate in cod – am creat buffere, le-am trimis catre shadere si le-am desenat. La acest capitol o sa discut despre un nou tip de obiecte, si anume modelele, care sunt tipuri de obiecte incarcate din exterior, dintr-un fisier in care sunt descrisi datele pentru vertecsi si ordinea de desenare a indicilor. Modelele pot fi construite dintr-un singur mesh, sau pot avea mai multe mesh-uri legate una de alta prin mostenire. De exemplu, un model pentru un om este construit din mai multe mesh-uri pentru membre, trunchi, cap, degete etc. la care se pot aplica transformari individual, insa datele de transformare ale acestora vor fi mereu relative la parinte (daca tot modelul omului se misca, se vor misca si membrele, insa pozitia lor locala va fi aceiasi ca si inainte, intrucat nu ele s-au miscat, ci parintele lor).

Acelasi rationament l-am urmat si la aceasta etapa, alegand sa folosesc o librarie externa care abstractizeaza logica. Am hotarat sa folosesc libraria **Assimp**. Eu am ales folosirea formatului **.OBJ** pentru modele si **.MTL** pentru materiale (fisiere care aloca anumite tipuri de texturi pentru fisiere imagine).

Logica pentru manipularea modelelor am descris-o in clasa Model. Aici am hotarat sa folosesc o structurare a metodelor in 2 pasi, similar ca si la clasa Mesh – **initializare** si **desenare**.

Metoda de initializare am numit-o import. Aceasta are 2 overload-uri pentru ca programatorul poate dori sa importe un obiect caruia sa ii aplice textura sau nu. Acest lucru ramane la atitudinea programatorului.

Logica descrisa in metoda `Import()` incepe cu setarea coordonatelor de transformare relativ la pozitia parintelui si apoi prin setarea culorii. Calcularea pozitiei si a rotatiei se face insumand valoarea specificata in parametri la pozitia si rotatia parintelui. Scalarea se face inmultind scalarea locala la scalarea parintelui. Culoarea e individuala, nu are legatura ce cea a parintelui.

Assimp ofera o clasa numita `Importer` care permite citirea fisierelor. Modelul parsat este stocat intr-un pointer de clasa `aiScene` pentru a se memora pe tot parcursul main loop-ului si evita bug-uri de tip dangling pointer.

Mai departe se apeleaza metoda `ProcessNode()` care parcurge structura ierarhica a mesh-urilor din model. La fiecare dintre mesh-uri se apeleaza metoda `ProcessMesh()` care parseaza datele pentru vertecsi, indici si texturi. Aceste lucruri pana acum se scriau in vectori si se incarcau in buffere pe baza unui stride. Acest lucru era oarecum dificil de implementat in logica de incarcare a modelelor asa ca am creat o 2 structuri: `VertexStruct` si `TextureStruct`. VertexStruct contine 3 vec3-uri pentru coordonate de pozitie, normale si culori si un vec2 pentru coordonate de textura. TextureStruct contine un string pentru tipul de textura si unul pentru calea catre textura. Este important de mentionat tipul texturii, intrucat acestea se vor desena intr-un anumit fel in functie de datele din fisierul .MTL care ii comunica assimp-ului ce fisiere sa citeasca de ce tip sunt. De asemenea, acest lucru se face simplu prin folosirea enum-uri prezente in assimp care fac corelarea intre numele lor si string-ul asteptat in fisierul .MTL (de exemplu un enum `aiTextureType_DIFFUSE` pointeaza catre un prefix map_Kd in fisierul .MTL care reprezinta o textura difuza).

Pentru a permite adaugarea datelor a mai multi vertecsi sau texturi, am creat un vector de `VertexStruct` si unul pentru `TextureStruct` in header-ul clasei Model. Nu este necesara crearea unui structuri pentru indici, intrucat datele care se introducea inainte erau doar `unsigned int`-uri. Am creat, in consecinta, si un vector de `GLuint`-uri pentru indici.

In momentul cand toate aceste date au fost citite se creeaza bufferi pentru aceste date si se returneaza mesh-ul rezultat.

Fiecare mesh se salveaza intr-un vector de mesh-uri.

Toate lucrurile descrise in acest capitol, pana acum, se fac inainte de main loop.

Desenarea mesh-urilor modelului se face in main loop si mai exact consta in parcurgerea vectorului de mesh-uri si desenarea fiecareia dintre acestea.

Pentru obiectele care se doresc a avea texturi, singura diferenta este ca se apeleaza metoda `CreateTextures()` dupa importare pentru fiecare dintre mesh-uri.

3. Scene management

a. Organizare ierarhica a obiectelor / entitatilor scenei 3D

La capitolul anterior am precizat ca mesh-urile din model sunt construite pe baza unui model ierarhic bazat pe mostenirea transformarilor. Acest lucru poate fi insa dorit si la nivelul scenei, dorindu-se a avea 2 obiecte corelate unul de altul.

Am implementat aceasta cerinte aducand un nou parametru metodelor `Import()` pentru modele si `Create()` pentru mesh-uri. Este vorba de `Model& parent`, respectiv `Mesh& parent`.

La baza scenei insa intervine o problema, anumite obiecte neavand niciun parinte. Se considera aici un parinte imaginar – originea. In consecinta, pentru crearea primului strat de obiecte, este necesara instantierea unor obiecte de tip `Model&` si `Mesh&` cu transformarile implicite care se folosesc ca parinte pentru aceste obiecte.

b. Mecanism de culling

In primul rand, pentru imbunatatirea performantei am modificat indicii care specifica ordinea de desenarea in asa fel incat parcurgerea lor sa se faca mereu in sens contrar acelor de ceasornic. OpenGL este inteligent, si considera in mod implicit o astfel de desenare ca fiind orientata catre directia din care se priveste. Daca un triunghi este desenat in sensul acelor de ceasornic, acesta este orientat catre partea opusa. OpenGL defineste aceste orientari ca `GL_FRONT` si `GL_BACK`. Prin simpla apelare a metodei `glEnable(GL_CULL_FACE)` se omit de la randarea fetele specificate ca parametru metodei `glCullFace()` si se randeaza in acest mod doar jumatate din triunghiurile totale, cele orientate catre observator.

Acesta este o optimizare bine-venita, insa insuficienta, de aceea am implementat si algoritmul *Frustum Culling*.

Pentru a implementa algoritmul Frustum Culling se pot aborda 3 modalitati:

Geometric Approach care presupune extragerea celor 6 planuri care formeaza frustul (planul din stanga, din dreapta, de sus, de jos, planul apropiat si planul indepartat). Planurile care se vor extrage vor fi definite in coordonate globale. Pe baza acestor planuri, se poate determina daca un corp se afla in interiorul volumului format de aceste planuri, sau daca suprafata acestui corp intersecteaza unul dintre planuri (cazuri in care corpul se va desena).

Clip Space Approach care presupune de aceasta data extragerea obiectului pe baza coordonatelor lui din *clip space*. Clip space este spatiul dinainte de rasterizare (trecerea din clip space la screen space) care descrie cum se va afisa un obiect pe camera (tinand cont si de proiectiile lui tridimensionale). Pentru a se ajunge la clip space, se inmulteste pozitia locala a obiectului cu

matricea globala (de modelare), matricea de vizualizare si matricea de proiectie. Pentru a desena in shadere un obiect, trebuie bineinteles sa se tina cont de matricea de vizualizare, intrucat aceasta descrie pozitia camerei raportata la pozitia obiectului. In cazul Clip Space Approach, se porneste de la pozitia camerei ca si functie de referinta, deci matricea de proiectie este matricea identitate. Din aceasta cauza, unele implementari de pe internet tin cont doar de matricea globala si de proiectie. Clip space este normalizat si deci se considera ca planurile sunt descrise dupa urmatoarele coordonate: stanga **x = -1**, dreapta **x = 1**, jos **y = -1**, sus **y = 1**, apropiat **z = -1**, indepartat **z = 1**. Pe baza coordonatelor de pozitie rezultate in urma transformarii obiectului in clip space se poate determina daca acesta se afla in interiorul frustului sau daca intersecteaza una dintre planele marginase.

Radar Approach este si modalitatea pe care am ales-o eu, intrucat este cea mai eficienta din punct de vedere computational. Aceasta presupune mai intai calcularea pozitiei planurilor in functie de pozitia camerei si de datele care o descriu. Calcularea planurilor indepartat si apropiat se face insumand la pozitia camerei valorile de clipping declarate la initializarea camerei. Calcularea planurilor de jos si sus se face pe baza inaltimii camerei. La fel se procedeaza si pentru calcularea planurilor din stanga si dreapta, dar folosind latimea camerei. Avand aceste date, se verifica separat, pentru axa OX, OY si OZ, daca coordonatele X, Y si Z se afla in intervalele corespunzatoare. La final datele se pun laolalta, si daca intr-una dintre cele 3 comparatii corpul se afla in afara intervalului, se considera ca este in afara frustului.

Pentru a determina daca coordonata Z a obiectului este intre planul apropiat si indepartat, se calculeaza un **vector de la camera catre obiect**, apoi se face produs scalar intre vectorul catre obiect si **orientarea camerei**. Daca acesti 2 vectori ar fi fost normalizati, produsul scalar dintre ei ar fi rezulat cu cosinusul unghiului dintre ei sau raportul dintre lungimea catetei alaturate si ipotenuza, adica raportul dintre Z si lungimea vectorului de la camera la obiect. Pentru ca nu vrem asta, putem fie sa facem exact acest lucru, dar adaugam o impartire la fiecare calculare, sau sa nu mai normalizam vectorul de la camera la obiect, si deci $a \cdot b = \frac{\cos\theta}{\|a\| b}$. Coordonata Z este egala cu lungimea vectorului rezultat, pentru ca originea in acest este camera, cu coordonate (0, 0, 0) pentru ca suntem in spatiu local, raportat la aceasta.

Pentru a calcula coordonata Y, vom avea nevoie sa calculam **inaltimea** camerei. Fie un unghi θ egal cu **FOV**-ul **camerei**. Daca privim planul de vizualizare ZoY, putem deduce inaltimea aplicand formula tangentei (cateta opusa / cateta alaturata). Aceasta se poate aplica doar intr-un unghi mai mic de 90 grade, deci vom imparti campul de vizualizare in doua. Daca calculam tangenta intre vectorul de la camera catre obiect si vectorul descris de planul de sus, obtinem ca $tg\theta$ este egala cu raportul dintre lungimea unei jumatati de inaltime si lungimea vectorului de la camera catre obiect. De aici rezulta ca inaltimea este egala cu produsul dintre $\boldsymbol{tg\theta}$ si *de doua ori* lungimea **vectorului de la camera catre obiect**.

Pentru a calcula coordonata X, trebuie sa calculam latimea camerei. Putem sa aplicam acelasi procedeu ca mai sus pentru planul ZoX, insa cel mai simplu mod de a obtine latimea este sa inmultim **inaltimea** cu **aspect ratio**-ul ecranului. Atat planul apropiat cat si planul indepartat sunt proiectii ale ecranului. Din aceasta cauza, ele au aspect ratio-ul constant. In consecinta, care se afla intre acestea va avea si el acelasi aspect ratio.

Pentru a implementa oricare dintre cele 3 metode descrise mai sus pentru un corp tridimensional trebuie insa sa se tina cont si de volumul acestuia la comparatii. Exista mai multe solutii, insa solutia folosita de mine a fost sa calculez o sfera in jurul obiectului (cu lungimea razei egala cu cea mai multa latura a corpului OX, OY sau OZ).

In codul sursa, functia `SetCamInternals()` are scopul de de a calcula niste factori, care se vor folosi la calcularea dimensiunilor reale ale sferei din jurul unui corp. Trebuie luat in calcul ca sfera se va afla la o distanta fata de camera, iar dimensiunea care se vede nu este reala, intrucat cu cat un obiect este mai departe in perspectiva, cu atat este mai mic. Functia `SetCamDef()` calculeaza coordonatele X, Y si Z. Nu in ultimul rand, functia `SphereInFrustum()` determina daca sfera se afla in frust. Aceasta poate returna una dintre enum-urile `INSIDE`, `INTERSECT` si `OUTSIDE`. Pe baza acestei functii am folosit in final 2 functii `MeshInFrustum()` si `ModelInFrustum()` care calculeaza sfera si pe baza enum-ului returnat, returneaza la randul lor o valoare booleana `true` sau `false` pe baza carora se decide daca obiectul se va desena sau nu.

c. Format XML propriu pentru scena 3D

Si la aceasta etapa am folosit o librarie externa care se ocupa cu parsarea fiserelor de tip XML. Este vorba de **pugixml**.

Logica am creat-o in clasa `XMLParser`, unde sunt 4 metode.

Metoda `ParseScene()` este folosita ca sa parseze un fisier XML si sa stocheze nodul scena intr-o variabila `pugi::xml_node`. Aceasta variabila este folosita ca si parametru initial pentru metoda `CreateModels()`. Modul de functionare al acesteia este similar ca cel al metodie `ProcessNode()` de la capitolul 2, insa de data aceasta nu se parcurg mesh-uri recursiv, ci noduri din fisierul XML. Si aici, in mod similar, este necesara folosirea unui nod radacina, care nu este originea, ci scena.

Pe langa acest lucruri, metoda `ParseScene()` parseaza transformarile si culorile din XML pentru fiecare obiect, precum si calea catre texturi si modele. Mai departe, se verifica despre ce tip de fisier este vorba – daca este sursa de lumina sau obiect simplu, si se introduc separat in 2 vectori `modelLights`, respectiv `modelObjects`. Mai apoi, se verifica daca nodul „children”, care este nod copil al obiectului curent contine copii, si daca da, obiectul curent devine parintele copiilor sai, si se apeleaza recursiv metoda, cu nodul „children” ca parametru. In acest mod, se va incepe un nou loop pentru toti copii nodului „children” care sunt noduri de obiecte.

Urmatoarele 2 metode din clasa sunt `DrawModelLights()` si `DrawModelObjects()`. Aceste 2 metode sunt separate pentru ca trateaza 2 tipuri diferite de obiecte. Luminile trebuiesc separate pentru ca ele contribuie la aspectul obiectelor normale. Daca retineti, se folosesc pozitiile si culorile lor in shaderele obiectelor normale, pentru a determina culoarea fragmentelor. In concluzie, luminile trebuiesc separate de obiectele normale, iar luminile trebuiesc desenate primele, ca mai apoi obiectele sa aiba ce date sa foloseasca in shadere.

Logica acestor metode consta intr-un simplu loop prin toate modelele din vectorii surselor de lumina si al obiectelor simple. Pentru fiecare dintre acestea se apeleaza metoda `Draw()` corespunzatoare clasei `Model`.