

# 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 functie 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

În capitolul anterior am discutat despre „obiecte”, dar nu am precizat încă la ce mă refer. Construirea de entități am făcut-o până acum în integritate în cod – am creat buffere, le-am trimis către shader și le-am desenat. La acest capitol o să discut despre un nou tip de obiecte, și anume modelele, care sunt tipuri de obiecte încărcate din exterior, dintr-un fișier în care sunt descriși datele pentru vertecși și 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 transformări individuale, însă datele de transformare ale acestora vor fi mereu relative la părinte (dacă tot modelul omului se mișcă, se vor mișca și membrele, însă poziția lor locală va fi aceeași ca și înainte, întrucât nu ele s-au mișcat, ci părintele lor).

Același raționament l-am urmat și la această etapă, alegând să folosesc o bibliotecă externă care abstractizează logica. Am hotărât să folosesc biblioteca **Assimp**. Eu am ales folosirea formatului **.OBJ** pentru modele și **.MTL** pentru materiale (fișiere care alocă anumite tipuri de texturi pentru fișiere imagine).

Logica pentru manipularea modelelor am descris-o în clasa `Model`. Aici am hotărât să folosesc o structurare a metodelor în 2 pași, similară ca și la clasa `Mesh` – **initializare** și **desenare**.

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

Logica descrisă în metoda `Import()` începe cu setarea coordonatelor de transformare relativ la poziția părintelui și apoi prin setarea culorii. Calcularea poziției și a rotației se face însumând valoarea specificată în parametri la poziția și rotația părintelui. Scalarea se face înmulțind scalarea locală la scalarea părintelui. Culoarea e individuală, nu are legătură cu cea a părintelui.

Assimp oferă o clasă numită `Importer` care permite citirea fișierelor. Modelul parsat este stocat într-un pointer de clasă `aiScene` pentru a se memora pe tot parcursul main loop-ului și evita bug-uri de tip dangling pointer.

Mai departe se apelează metoda `ProcessNode()` care parcurge structura ierarhică a mesh-urilor din model. La fiecare dintre mesh-uri se apelează metoda `ProcessMesh()` care parsează datele pentru vertecși, indici și texturi. Aceste lucruri până acum se scriau în vectori și se încărcau în buffere pe baza unui stride. Acest lucru era oarecum dificil de implementat în logica de încărcare a modelelor așa că am creat o 2 structuri: `VertexStruct` și `TextureStruct`. `VertexStruct` conține 3 `vec3`-uri pentru coordonate de poziție, normale și culori și un `vec2` pentru coordonate de textura. `TextureStruct` conține un string pentru tipul de textura și unul pentru calea către textura. Este important de menționat tipul texturii, întrucât acestea se vor desena într-un anumit fel în funcție de datele din fișierul `.MTL` care îi comunică assimp-ului ce fișiere să citească de ce tip sunt. De asemenea, acest lucru se face simplu prin folosirea enum-uri prezente în assimp care fac corelarea între numele lor și string-ul așteptat în fișierul `.MTL` (de exemplu un enum `aiTextureType_DIFFUSE` pointează către un prefix `map_Kd` în fișierul `.MTL` care reprezintă o textura difuză).

Pentru a permite adăugarea datelor a mai mulți vertecși sau texturi, am creat un vector de `VertexStruct` și unul pentru `TextureStruct` în header-ul clasei `Model`. Nu este necesară crearea unui structuri pentru indici, întrucât datele care se introduceau înainte erau doar `unsigned int`-uri. Am creat, în consecință, și un vector de `GLuint`-uri pentru indici.

În momentul când toate aceste date au fost citite se creează bufferi pentru aceste date și se returnează mesh-ul rezultat.

Fiecare mesh se salvează într-un vector de mesh-uri.

Toate lucrurile descrise în acest capitol, până acum, se fac înainte de main loop.

Desenarea mesh-urilor modelului se face în main loop și mai exact constă în parcurgerea vectorului de mesh-uri și desenarea fiecăreia dintre acestea.

Pentru obiectele care se doresc a avea texturi, singura diferență este că se apelează metoda `CreateTextures()` după importare pentru fiecare dintre mesh-uri.

### 3. Scene management

#### a. Organizare ierarhică a obiectelor / entităților scenei 3D

La capitolul anterior am precizat că mesh-urile din model sunt construite pe baza unui model ierarhic bazat pe moștenirea transformărilor. Acest lucru poate fi însă dorit și la nivelul scenei, dorindu-se a avea 2 obiecte corelate unul de altul.

Am implementat această cerință aducând un nou parametru metodelor `Import()` pentru modele și `Create()` pentru mesh-uri. Este vorba de `Model& parent`, respectiv `Mesh& parent`.

La baza scenei însă intervine o problemă, anumite obiecte neavând niciun părinte. Se consideră aici un părinte imaginar – originea. În consecință, pentru crearea primului strat de obiecte, este necesară instanțierea unor obiecte de tip `Model&` și `Mesh&` cu transformările implicite care se folosesc ca părinte pentru aceste obiecte.

#### b. Mecanism de culling

#### c. Format XML propriu pentru scena 3D

Și la această etapă am folosit o bibliotecă externă care se ocupă cu parsarea fișierelor de tip XML. Este vorba de **pugixml**.

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

Metoda `ParseScene()` este folosită ca să parșeze un fișier XML și să stocheze nodul scenei într-o variabilă `pugi : xml_node`. Această variabilă este folosită ca și parametru inițial pentru metoda `CreateModels()`. Modul de funcționare al acesteia este similar cu cel al metodei `ProcessNode()` de la capitolul 2, însă de data aceasta nu se parcurg mesh-uri recursiv, ci noduri din fișierul XML. Și aici, în mod similar, este necesară folosirea unui nod rădăcină, care nu este originea, ci scena.

Pe lângă acest lucru, metoda `ParseScene()` parșează transformările și culorile din XML pentru fiecare obiect, precum și calea către texturi și modele. Mai departe, se verifică despre ce tip de fișier este vorba – dacă este sursa de lumină sau obiect simplu, și se introduc separat în 2 vectori `modelLights`, respectiv `modelObjects`. Mai apoi, se verifică dacă nodul „children”, care este nod copil al obiectului curent conține copii, și dacă da, obiectul curent devine părintele copiilor săi, și se apelează recursiv metoda, cu nodul „children” ca parametru. În acest mod, se va începe un nou loop pentru toți copii nodului „children” care sunt noduri de obiecte.

Următoarele 2 metode din clasă sunt `DrawModelLights()` și `DrawModelObjects()`. Aceste 2 metode sunt separate pentru că tratează 2 tipuri diferite de obiecte. Luminile trebuie 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](#).