

# Etapa 3 – Motoare de grafica 3D in timp real

*Task-uri:*

1. *2D GUI System*
2. *Gestiune limbaje si scripting*
  - a. *Limbaj de scripting: Lua*
  - b. *Declansare si procesare evenimente in scena 3D*
3. *Detectia coliziunilor*
4. *Bucla principala a motorului grafic – integrarea tuturor subsistemele motorului grafic*
5. *Sistem de particule*

## 1. 2D GUI System

Pentru implementarea acestui feature am folosit Dear ImGui. Clasa pe care am folosit-o la aceasta etapa am numit-o `GuiDrawer`.

Pentru ca am privit sistemul GUI ca un tot unitar prin intermediul caruia sa se faca toate operatiile de desenare ale interfetei grafice, am considerat important ca acesta sa fie un Singleton. Una dintre modalitatile cele mai simple pe care eu le cunosc pentru a implementa acest design pattern este prin folosirea datelor de tip `static`. In acest mod, am declarat ca `static` toate metodele si variabilele de clasa din `GuiDrawer`.

Avantajul Dear ImGui si motivul pentru care am ales acest framework nu este numai faptul ca este o foarte puternic in lucrul cu interfețele grafice, ci si faptul ca este foarte bine documentat. Din aceasta cauza, mi-a fost relativ simplu sa implementez si eu o fereastră in interfata grafica si sa o personalizez dupa nevoile mele.

Structura clasei `GuiDrawer` este similara si ca la celelalte clase implementate de mine – se folosesc metodele `Init()`, `End()` si `Draw()`.

Dupa cum presupune si numele lor, `Init()` se foloseste la abstractizarea tuturor functiilor de pregatire a framework-ului Dear ImGui, iar `End()` abstractizeaza distrugerea structurilor de date folosite de acesta in memorie. Nu am folosit constructori si destructori din motive de clean-coding – am preferat sa fie cate o linie in cod unde se intampla aceste lucruri.

`Draw()` il folosesc ca si in celelalte clase la desenare, dar aici la desenarea interfetelor grafice, bineinteles.

Cum este cazul si in multe alte framework-uri din C++, Dear ImGui foloseste un memory management pattern foarte simplu – se foloseste un context care trebuie sa fie *in viata* atat timp cat se acceseaza orice date din librariile Dear ImGui. Bineinteles, Dear ImGui abstractizeaza si acest concept pe baza metodelor `ImGui::CreateContext()` si `ImGui::DestroyContext()`.

Fiind un framework de OpenGL care face desenari, este obligatoriu ca undeva sa se defineasca un shader care va fi folosit la desenari. Si aceasta functie este abstractizata de dezvoltatorii Dear ImGui, tot ce este necesar sa precizez eu fiind fereastră in care se deseneaza si versiunea GLSL. Metodele corespondente acestor operatiuni de initializare GLFW si OpenGL sunt `ImGui_ImplGlfw_InitForOpenGL()` si `ImGui_ImplOpenGL3_Init()`, urmate de distrugatoarele lor `ImGui_ImplGlfw_Shutdown()` si `ImGui_ImplOpenGL3_Shutdown()`.

Optional, se poate folosi si un stil de afisaj al interfetei grafice (light sau dark) cu metodele `ImGui::StyleColorsClassic()` si `ImGui::StyleColorsDark()`. Eu am folosit dark, pentru ca este mai usor de suportat pentru ochi.

In acest moment, tot ce mai este necesar pentru ca Dear ImGui sa poata desena o fereastră e sa desemnam inceputul si sfarsitul unui frame, si sa desemnam datele propriu-zise dintr-o fereastră grafica.

Desemnarea inceputului unui frame se face pe baza a 3 metode cu GLFW:

`ImGui_ImplOpenGL3_NewFrame()`, `ImGui_ImplGlfw_NewFrame()` si `ImGui::NewFrame()`. Desemnarea sfarsitului unui frame se face cu `ImGui::Render()` si

`ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData())`.

Desemnarea unei ferestre grafice se face cu metodele `ImGui::Begin("Nume Fereastră")` si `ImGui::End()`. Intre apelarea acestor 2 metode se pot crea o multitudine de elemente grafice cum ar fi slidere, selectoare de culoare, tickbox-uri, etc.

Eu am creat o fereastră care sa manipuleze datele pentru un sistem de particule:



## 2. Gestiune limbaje si scripting

Sistemul de scripting construit in clasa `ScriptingSystem` este bazat pe lua 5.4.4.

Logica este simpla, permitand in prezent returnarea de variabile din lua in C++, insa nu si invers – nu se pot importa variabile din C++ in lua.

Metoda implementata de mine in sistemul de scripting este `CallFunctionByName()`, care parseaza un fisier de tip `.lua` si permite executarea acestuia cu sau fara parametri.

Pe langa acestea, am adaugat 2 linii de cod in clasa `World`, care executa in metoda `Init()` din C++ orice functie din lua care poarta numele `Init`. Similar, in metoda `Update()` din C++ se executa orice functie din lua care poarta numele `Update`.

## 3. Detectia coliziunilor

Sistemul de detectie a coliziunilor construit de mine, pe langa detectia de coliziuni propriu-zisa mai face si o *decoliziune* intre obiecte, adica forteaza in exterior orice obiect care intersecteaza sau se afla in interiorul altui obiect.

Detectia de coliziuni este simpla, pentru ca am refolosit codul de la Frustum Culling. Pentru a recapitula, detectarea intersectiei unui obiect cu frustul, sau daca preferati *coliziunea* se facea prin cautarea celei mai lungi laturi ale unui obiect dintre axele oX, oY si oZ. Aceasta lungime se lua ca diametrul unei sfere imaginare care se folosea la verificari de intersectie.

Aici intervine diferenta intre cei doi algoritmi, pentru ca in loc sa verificam daca frustul se intersecteaza cu o lista de obiecte, verificam daca fiecare obiect dintr-o lista de obiect intersecteaza alt obiect din aceeași lista.

Pentru a face verificarea intersectiilor a doua obiecte, este suficient sa stim ancora lor, care se trateaza ca si centru de sfera. Intr-o astfel de situatie, ar rezulta o simpla verifica de obiect de la obiect, pentru fiecare obiect din lista, cu celelalte obiecte cu care nu s-a verificat deja.

Readucaminte ca obiectele mele din scena sunt definite ca instante ale claselor [Mesh](#) si [Model](#). In consecinta, nu voi avea un singur caz de tratat – **obiect la obiect**, ci 4 cazuri – **mesh la mesh**, **mesh la model**, **model la mesh** si **model la model**.

Verificarea intersectiei celor doua obiecte se face prin verificarea distantei dintre cele doua puncte – daca aceasta este mai mica sau egala decat suma razelor celor sfera imaginare ale obiectelor, concluzionam ca obiectele fac contact.

*Decolizionarea* celor doua obiecte se face prin mutarea primului obiect la o pozitie hardcodata in exteriorul celui de-al doilea obiect.

O imbunatatire ar fi mutarea primului obiect la cel mai apropiat punct din exteriorul sferei imaginare al celui de-al doilea obiect.

**Nota:** Dezavantajul acestei metode este ca trateaza ancora ca si centrul obiectului de fiecare data, deci pot apare bug-uri de coliziune pentru obiectele ale caror ancore sunt spre exemplu la baza.

## 4. Bucla principala a motorului grafic

Pentru usurarea dezvoltarii acestui proiect am adoptat cateva practici utile, dintre care versionarea (cu ajutorul [git](#)) si integrarea de feature-uri noi care au fost mai apoi testate si merge-uite cu branch-ul master.

## 5. Sistem de particule

Sistemul de particule din clasa [ParticleSystem](#) propus de mine este controlat pe baza a 4 metode: `Init()`, `CreateParticle()`, `UpdateParticleData()` si `DrawParticle()`.

Fiecare particula va avea proprietati diferite (pozitie, rotatie, scalare, viteza). Din aceasta cauza, bufferii particulelor care se vor trimite la shader nu vor mai putea fi definiti la initializare din simplul fapt ca ele nu vor mai fi constante, ci vor varia. Din acest motiv, crearea bufferilor pentru aceste mesh-uri se va face in bucla principala.

Acest lucru creeaza o problema: modul in care am creat eu arhitectura clasei [Mesh](#) nu favorizeaza crearea de obiecte in bucla principala. Definirea proprietatilor clasei se face la crearea bufferilor, dar momentul in care aceste proprietati sunt folosite in realitate este la desenare.

Continuarea pe baza acestei arhitecturi a adus mai multe bug-uri care se rezuma la o problema principala – matricea de modelare trebuia sa fie creata direct cu proprietatile de transformare, altfel ar fi suprascris orice alta matrice de modelare creata inainte metodei `Draw()` sau ar fi fost

actualizata abia la urmatoarea iteratie de obiecte (daca spre exemplu am 1000 de particule si sunt la indexul 500, matricea noua de modelarea s-ar fi citit abia dupa desenarea a 1000 de particule incorect).

Cea mai simpla solutie la aceasta probleme, ar fi fost sa revin la factorizarea de acum cateva etape, cand cream matricea de modelarea pe baza parametrilor metodei `Draw()`, dar acest lucru ar fi creat probleme de incompatibilitate cu clasa `Model`, intrucat in interior aceasta functioneaza pe baza clasei `Mesh`.

O alternativa a fost sa fac niste supraincari pentru metodele `CreateBuffers()` si `Draw()`, diferenta fiind ca la noul `CreateBuffers()` nu mai scriam pozitia, rotatia, scalarea si culoarea instantei de obiect, iar la `Draw()` nu mai cream matricea de modelarea pe baza variabilelor de clasa, ci pe baza parametrilor metodei. Un alt avantaj al acestei alternative este ca se creeaza o singura data bufferii (in general fiind vorba de obiecte de acelasi fel), si se omit instructiuni de creare de bufferi inutile (dupa vechea arhitectura), daca este vorba de acelasi tip de verteci si indici.

Logica clasei `ParticleSystem` se creeaza prin 4 metode: `Init()` care initializeaza lista de particule si seteaza indexul particulei curente. Index-ul porneste de la ultimul item din lista si itereaza catre elementul 0, dupa care se repeta in bucla. Acest lucru se intampla pentru ca desenarea se face din spate in fata, iar astfel se evita niste artefacte de desenare.

`CreateParticle()`, o alta metoda din aceasta clasa care in realitate nu creeaza nicio particula, ci initializeaza niste proprietati de comportament ale particulei – pozitie, rotatie, culoare la nastere si la moarte, viteza, scalare la nastere si la moarte, durata de viata si durata de viata ramasa.

`UpdateParticleData()`, care modifica durata de viata a particulei curente, pozitia (dupa viteza) si rotatia. Tot aici, se verifica daca o particula nu mai are viata, si daca este cazul, o marcheaza ca fiind dezactivata.

`DrawParticle()` care face crearea de bufferi si desenarea. De asemenea, aici se calculeaza viata ramasa a particulei si se interpoleaza culoarea si scalarea particulei dupa viata.