Universitatea Politehnica București, Romania
Facultatea de Automatica si Calculatoare

Editor de scene 3D

~ Document de Proiectare Arhitecturala ~

Membrii echipei:
Hotea Cristian
Ionescu Eugen
Tămîrjean Bogdan

Echipa: CEB Graphics
Indrumator: Florentina Oprea

Cuprins

1. Tehnologiile folosite

1.1 Motor grafic 3D

Pentru implementarea Motorul Grafic 3D(Editorul de scene 3D) vom folosi C++ ca limbaj de programare. In principiu, ne vom folosi de functiile de baza ale C++, clase simple, fara mosteniri deoarece nu este cazul. Vom folosi peste tot pointeri la diferite „obiecte/clase" si vom incerca sa nu hard-codam anumiti vectori. Totul va trebui sa fie citit din fisierele de configurare SM.cfg si RM.cfg si in functie de datele de acolo vom aloca memorie pentru pointerii nostri.Ca si framework, vom incerca sa gasim unul simplu care sa ne ajute sa cream Contextul de randare si eventual clase matematice predefinite(operatii matematice asupra vector2,vector3, matrici de rotatie, scalare, translatie,etc). Celelalte functionalitati prezentate vor fi implementate de noi. Vom folosi ca API pentru randarea imaginilor 3D, OpenGL ES 2.0.

1.2 OpenGL

OpenGL (Open Graphics Library) este o specificație standard care definește o aplicatie compatibila intre platforme (cross-platform API - application programming interface) foarte utilizat pentru programarea componentelor grafice 2D si 3D ale programelor de calculator. Interfața constă in peste 250 de apeluri diferite care pot fi folosite pentru a desenta scene 3D complexe din primitive simple. OpenGL a fost dezvoltat de Silicon Graphics Inc. (SGI) in 1992 și este foarte utilizat in softuri CAD, realitate virtuală, visualizare științifică, simulări de zboruri sau jocuri pe calculator. Acest ultim domeniu este in strânsă competiție cu tehnologia DirectX de la Microsoft. OpenGL este condus de un consortiu tehnologic non-profit, Khronos Grup.

1.3 Design

OpenGL servește două scopuri principale:

Pentru a ascunde si a nu favoriza unele componente hardware in complexitatea de interfata cu diferite acceleratoare 3D. Acest lucru este facut prin folosirea de catre programator a unui singur API uniform.

Pentru a nu se tine cont de capabilitatile, mai mari sau nu, a diferite platforme hardware in rularea de aplicatii, implementarile OpenGL suportand un intreg set de emulatoare software (acolo unde acestea sunt necesare).

Funcționarea de baza a OpenGL este aceea de a primi primitive (cum ar fi puncte, linii și poligoane) și de a le converti în pixeli. Acest lucru se face printr-un pipe (o conducta) grafic (graphics pipeline), cunoscut sub numele de mașină de stare OpenGL. Înainte de introducerea OpenGL 2.0, fiecare etapă din pipeline efectua o funcție fixă și era configurabila numai în limite restranse. OpenGL 2.0 oferă mai multe etape, care sunt pe deplin programabile folosind GLSL.

OpenGL este un API procedural de nivel mic, care necesită ca programatorul să impună măsurile exacte necesare pentru a face o scena. Acest lucru contrastează cu alte API-uri, în care un programator are nevoie doar dea descrie o scenă și poate lăsa biblioteca si gestiunea detaliilor pentru finalul scenei. OpenGL, impune ca programatorii să aibă o bună cunoaștere a pipeline-ului grafic dar, de asemenea, oferă o anumită libertatea de a pune în aplicare algoritmi noi de redare.

OpenGL are un istoric de influente de la dezvoltarea acceleratoarelor 3D, promovând un nivel de funcționalitate de bază, care este acum în comun cu nivelul hardware-ului de consum.

1.4 Shadere

În domeniul graficii de calculator, un shader este un program de calculator care poate rula pe o unitate de procesare grafică (GPU) și este folosit pentru a reda efecte speciale sau postprocesare.

Un shader calculeaza efectele de redare de pe hardware-ul grafic, cu un grad ridicat de flexibilitate. Limbajele specific shadere-lor sunt folosite pentru a programa pipeline-ul unitatii de procesare grafica (GPU). Cu ajutorul unui shader, pot fi utilizate chiar si efecte personalizate. Pozitia, nuanța, saturația, luminozitatea, contrastul tuturor pixelilor, nodurilor, sau texturilor utilizate pentru a construi o imagine finală pot fi modificate din mers, utilizând algoritmi definiti în shader și prin variabile sau texturi introduse de programul care foloseste shader-ul respectiv.

In implementarea noastra vom avea 2 tipuri de Shadere- Vertex Shader si Fragment Shader. Vertex Shader va avea ca si input coordonatele locale/globale ale primitivelor(linii,triunghiuri) care alcatuiesc obiectele din scena si va da ca si output coordonatele globale si fixe ale primitivelor. Dupa aceasta vor urma alte etape executate de GPU cum ar fi (asamblarea primitivelor, rasterizare, etc). Apoi vom folosi Fragment Shaderul care va avea ca si input fragmentele(pixelii) rezultati din etapele de dinainte.Aici vom putea sa „construim” culoarea respectivului pixel, iar outputul acestui shader va fi culoarea finala a pixelului procesat in format Vector4 (rgba).

1.5 Interfata cu utilizatorul

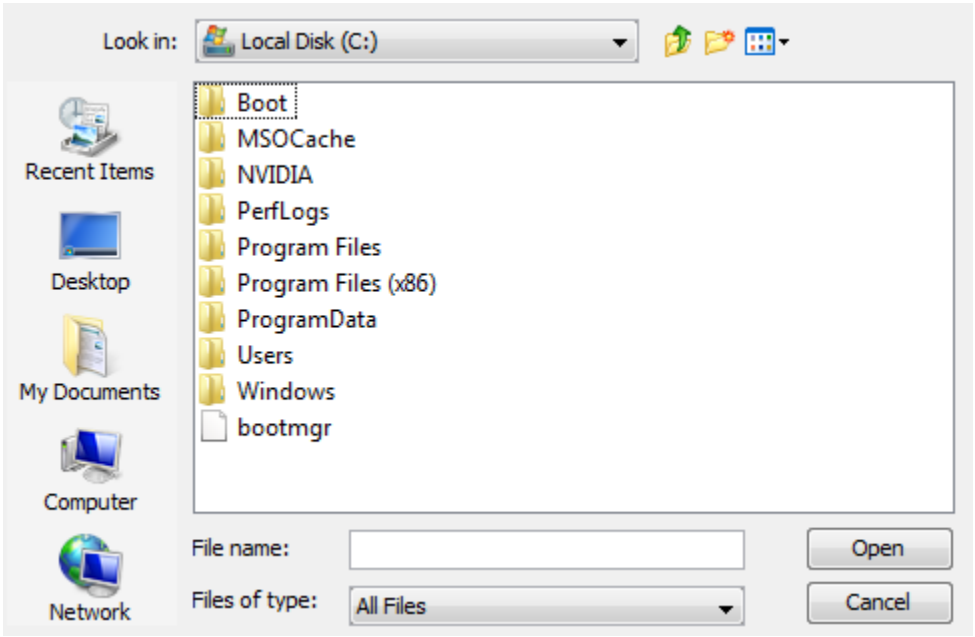
Interfata prin care utilizatorul poate edita fisierele de configurare este realizata in Java. La Initializarea aplicatiei utilizatorul are optiunea de a deschide un fisier existent sau de a crea unul nou. Selectarea fisierelor se realizeaza prin intermediul clasei JFileChooser.

Pentru realizarea interfetei grafice se folosesc obiecte din clasa *swing* . Se pun astfel la dispozitie campuri de text (JTextField) , butoane (JButton) , slider-e (JSlider) , campuri cu variante de alegere(JComboBox) etc .

Obiecte prezente in interfata :

JFileChooser :

- Se va folosi pentru a selecta fisierul de configurare ce va fi editat



JSlider & JTextField

- **Folosite pentru ajustarea valorilor parametrilor scenei**

Tratarea evenimentelor :

Schimbarea valorii unui camp din interfata de editare va fi tratata prin metode asociate evenimentelor . De exemplu , pentru tratarea unei actiuni asupra unui JTextField se va folosi metoda :

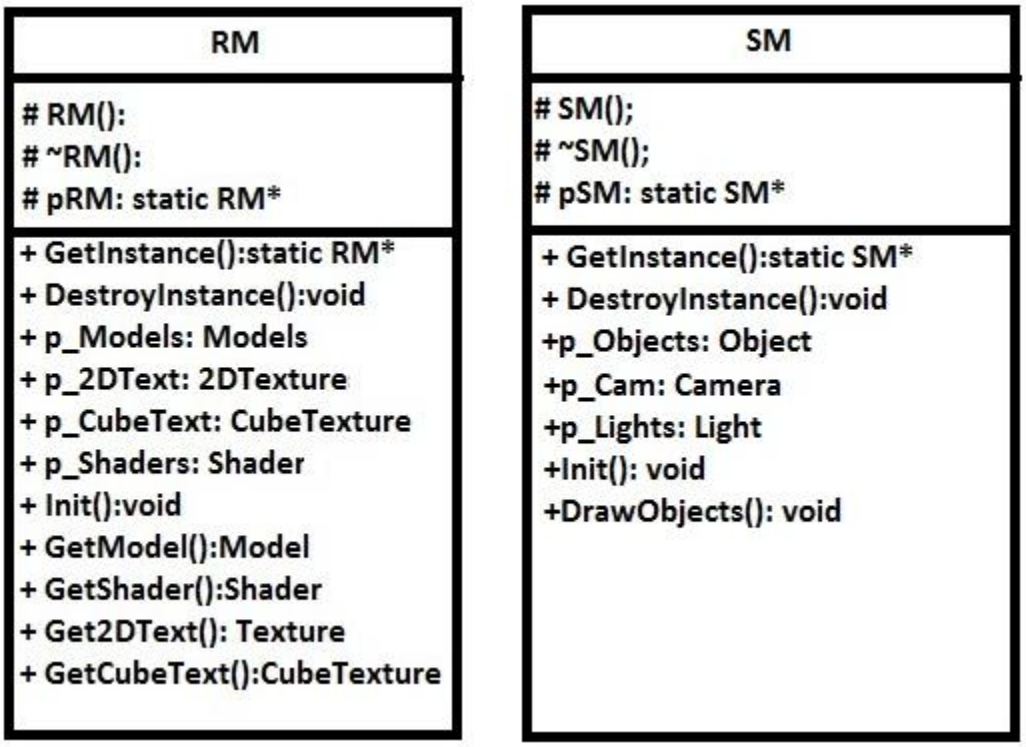
private void jTextField2ActionPerformed(java.awt.event.ActionEvent evt)

Initial , fisierele sunt incarcate in memorie . Folosind o clasa *Parser* se vor separa obiectele din Scene Manager si resursele din Resource Manager pentru a fi tratate individual in cadrul interfetei de editare . La adaugarea unui nou obiect sau a unei resurse , formularul se va ajusta pentru a include campurile de editare aferente .

La final , fisierul vechi este trunchiat si noile valori sunt inscrise , urmarind formatul specific al fisierului.

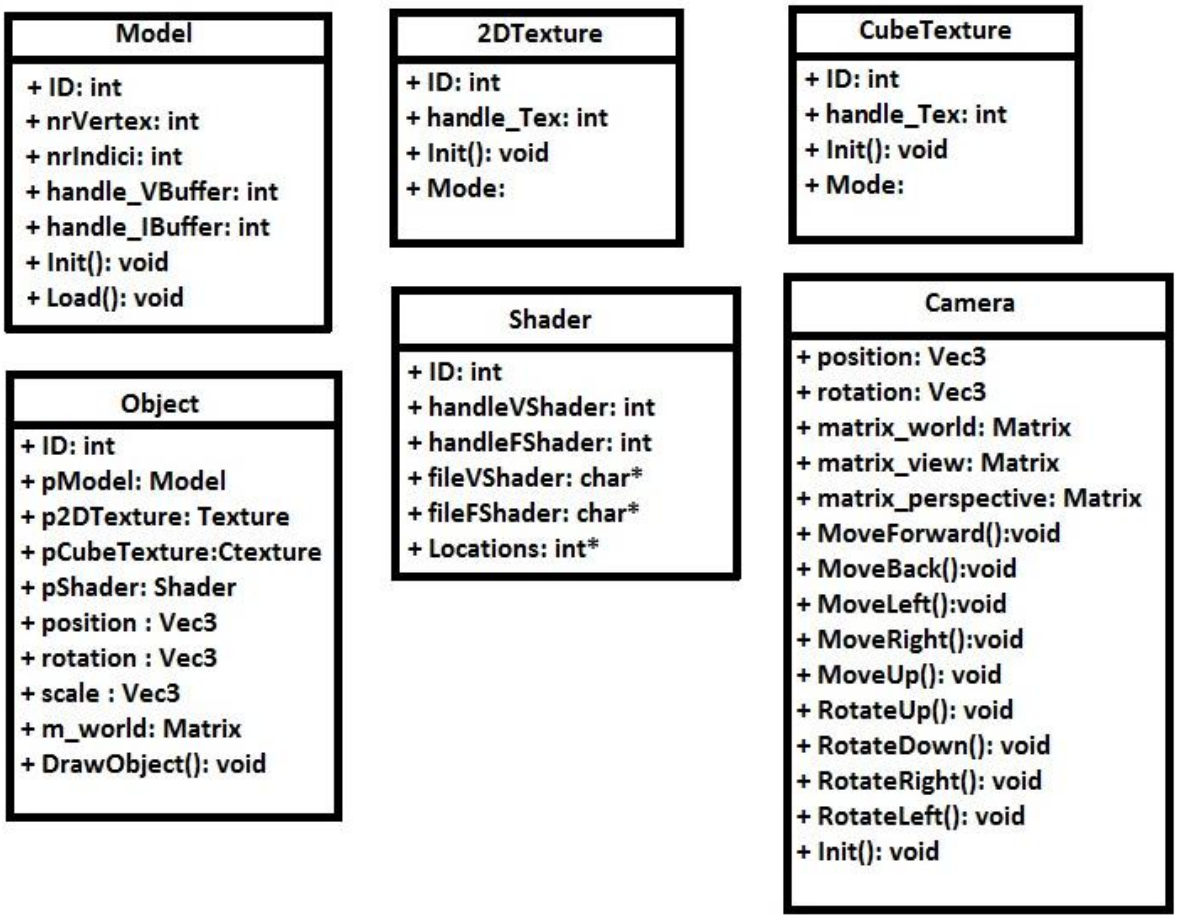
2 Diagrame de clase

2.1 RM / SM



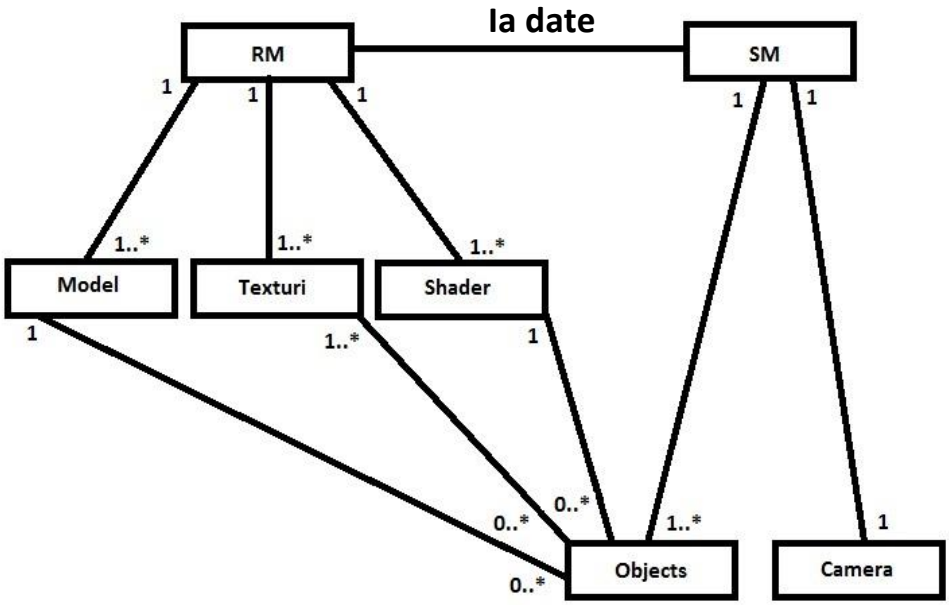
Ambele clase sunt construite dupa design-pattern-ul Singleton.Acesta presupune restrictionarea instantierii acestor clase la un singur obiect de tipul acestei clase. De aceea metoda GetInstance va intoarce un pointer static, fiindca vom aloca un singur obiect de tipul acestor clase.Clasa RM va contine un vector de modele, unul de 2DTexturi, unul de CubeTexture, unul de Shadere. Pe langa acestea va avea metodele prin care va returna un obiect din acesti vectori, depinde de cererea facuta de SM. Clasa SM va avea un vector de Obiecte pe care le va construi folosind functiile din RM. Pe langa obiecte va avea si Camera.

2.2 *Clase diverse*



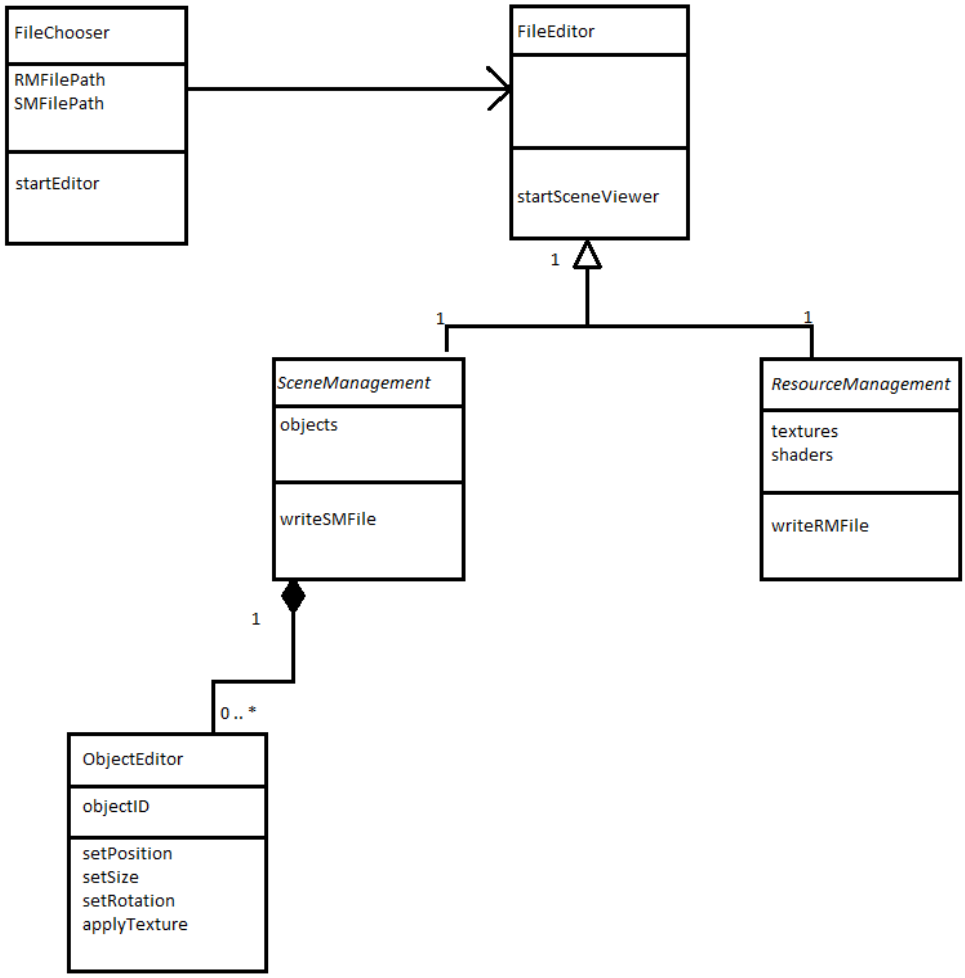
Clasa Model va reprezenta toata informatia necesara despre un Model incarcat dintr-un fisier.Folosind Init si Load, vom crea Buffere in care vom retine informatii despre varfurile primitivelor ce compun modelul. Clasa 2Dtexture/CubeTexture va contine informatia necesara despre textura respectiva, functia Init fiind cea care va crea bufferul cu datele din textura. Clasa Object va fi clasa ce va defini un obiect din scena noastra. Acesta va contine un model, o textura(sa mai multe), un shader(sau mai multe) precum si informatii despre pozitia/rotatia/scalarea acestuia in scena noastra. Clasa shader va contine cate un handle pentru fiecare tip de Shader. Clasa Camera va implementa functionalitatile unei camere intr-o scena 3D, mai exact, miscari pe toate cele 3 axe si rotiri fata de OX(Up/Down) si OY(Left/Right);

2.3 Diagrama de clase: Motor 3D



Descriere: Clasele RM si SM sunt clasele superioare care vor comunica intre ele prin schimb de date. Clasa RM trebuie sa contina cel putin un model,o textura si un shader. Relatia inversa este definita de faptul ca design patternul folosit implica o singura instanta a clasei RM, de aici rezultand faptul ca un mode/o textura/ un shader poate fi asociat intr-un singur obiect RM. La fel este si la SM, obiectele si camera vor fi asociate unui singur obiect de tipul SM. Clasa SM trebuie sa contina cel putin un obiect si o singura camera. Clasa Objects trebuie sa contina un singur model, trebuie sa contina cel putin o textura si cel putin un shader. Clasele Model/Texturi/Shader sunt definite in relatia cu clasa Objects ca fiind obiecte care pot sau nu sa fie instantiate in clasa Objects. Mai exact, un model/o textura/ un shader poate sa fie sau nu poate sa fie folosit in constructia unui Obiect.Invers, un obiect trebuie sa fie construit pe baza unui singur Model, cel putin unei texturi, cel putin unui shader.

2.4 *Diagrama de clase: Interfata grafica*



Descriere : Clasa FileChooser este clasa prin intermediul careia se face selectarea fisierelor RM si SM ce vor fi editate . Dupa selectarea fisierelor se trece direct la fereastra de editare . Prin intermediul clasei EditingFrame se va putea porni aplicatia si vizualiza scena . Clasele SceneManagement si ResourceManagement realizeaza inscrierea datelor in fisierele SM respectiv RM. Clasa ObjectEditor permite editarea atributelor unui obiect. Relatia dintre aceasta si clasa superioara este definita astfel incat o scena poate contine mai multe obiecte .