Game Engine - Introducere

Se vor folosi doua clase manager:

- **ResourceManager** - care administreaza resursele ce pot fi folosite in cadrul scenei: modele, texturi, shadere, fisiere de sunet etc.
- **SceneManager** - care administreaza obiectele din cadrul scenei, luminile, camerele (eye) disponibile si setarile lor etc.

Ambele clase sunt de tip Singleton: pot avea o singura instanta existenta.

Design pattern: Singleton

In realizarea designului trebuie asigurat faptul ca respectiva clasa nu poate avea mai multe instante in acelasi timp. De aceea nu vom face public constructorul. In schimb vom crea o metoda *getInstance()*. Aceasta metoda, atunci cand e apelata, va verifica daca exista deja o instanta a clasei si, daca da, va returna un pointer catre acea instanta, iar, daca nu exista, va crea o noua instanta si va returna pointerul catre aceasta.

```
class Singleton
{
	private:
		static Singleton* spInstance;
		Singleton();
	Public:
		void Init();//initializari - pot fi si in constructor
		static Singleton* getInstance();
		void metodaInstanta();//o metoda oarecare, nestatica
		Void freeResources();//eliberarea zonelor de memorie alocate
dinamic - se poate realiza si in destructor
		~Singleton()
};

Singleton* Singleton::spInstance = NULL;

Singleton* Singleton::getInstance()
{
    if(!spInstance)
    {
        spInstance = new Singleton();
    }
    return spInstance;

}

void Singleton::metodaInstanta(){
	//nitel cod
}
```

Atunci cand vrem sa apelam o metoda nestatica, vom scrie o instructiune de forma:
`Singleton::getInstance()->metodaInstanta();`
Metodele Init si freeResources sunt optionale si codul lor se poate adauga si in constructor, respectiv destructor. Insa uneori putem avea nevoie sa reinitializam instanta, sau sa eliberam memoria inainte de a distruge efectiv instanta, de accea poate fi o practica buna sa facem aceste operatii in niste functii diferite, lasand in constructor si destructor acele operatii de care avem nevoie sigur doar la crearea obiectului sau distrugerea obiectului, nu si la o resetare partiala a acestuia. Un alt motiv pentru care putem avea nevoie de Init si freeRsources este necesitatea unor parametri. Nu putem transmite parametri constructorului decat prin getInstance(), insa ar trebui sa ii retransmitem si la urmatoarele apeluri, nu doar la primul, sau sa facem un overload pentru aceasta functie, ajungand sa avem oricum acea "functie de Init", dar cu alt nume...

Clasa ResourceManager

La initializare va parsa xml-ul de configurare si va memora datele obtinute de acolo. Atentie! Nu trebuie sa va bazati pe ordinea datelor din fisier: aceasta ordine poate fi schimbata (de exemplu sa avem mai intai texturile si apoi modelele, sau sa avem intai shaderele si apoi celelalte tipuri de resurse). De asemenea id-ul trebuie sa fie unic pe resursa, insa e posibil sa nu aveti resursele ordonate dupa id in fisierul de configurare, ci sa apara intr-o ordine aleatoare.
Vor fi folosite 3 structuri pentru fiecare tip de resursa:

- ModelResource
- ShaderResource
- TextureResource

Aceste structuri vor avea ca proprietati exact proprietatile specificate in xml-ul de configurare. Nu reprezinta decat o structura de date pentru a incarca in memorie ce s-a citit din fisierul xml. Clasa ResourceManager va avea ca proprietati cate un vector (sau map din STL) de pointeri, pentru fiecare tip de informatie (deci va fi un vector de ModelResource*, unul de ShaderResource* etc.). Cel mai indicat ar fi sa folositi un map in care cheia e id-ul din XML si valoarea e obiectul-resursa.

Tot ResourceManager va administra si resursele incarcate. Deci pentru fiecare tip de resursa va exista cate o clasa care are ca rol memorarea datelor in urma incarcarii resursei.
Clasa Model:

- ModelResource* mr; //structura pe baza careia e incarcat modelul
- iboId (id-ul bufferului care contine indicii)
- wiredIboId (id-ul bufferului care contine indicii pentru afisarea wired)
- vboId(id-ul bufferului care contine vertecsii)
- nrIndici, nrIndiciWired
- Constructor
- Destructor

- Load() - care incarca modelul din fisierul nfg

Clasa Texture:
- TextureResource* tr; //structura pe baza careia e incarcata textura
- Id-ul texturii (generat prin glGenTextures())
- Constructor
- Destructor
- Load() - incarca textura

Clasa Shader:
- ShaderResource* mr; //structura pe baza careia e incarcat shader-ul
- Id-ul programului (obtinut prin linkarea vs-ului si fs-ului)
- Constructor
- Destructor
- Load()

ResourceManager va avea un vector (sau map, unde cheia pentru un element e id-ul din fisierul de configurare) pentru fiecare dintre cele 3 resurse efective de mai sus (in care se vor pastra pointeri catre resursele respective. Insa nu le va incarca pe toate de la inceput. Se va implementa o tehnica de "load on demand". Prin urmare ResourceManager va avea fie o functie template, fie pentru fiecare tip de resursa o functie de load: **loadModel**, **loadTexture**, **loadShader**. Aceste functii vor fi apelate atunci cand un obiect din scena are nevoie de o resursa. Toate cele 3 functii de load vor avea un comportament similar:
- daca resursa respectiva e deja in vectorul corespunzator de resurse incarcate, atunci returneaza un pointer catre resursa incarcata.
- Daca resursa respectiva nu e in vector, atunci se creeaza o resursa de acel tip, se apeleaza Load pentru ea, e adaugata in vectorul de resurse si e returnat un pointer catre ea.

Clasa SceneManager

Se ocupa de toate setarile scenei, setari pe care le ia din fisierul de configurare asociat.
Atentie! Cand faceti parsarea si construirea instantei pentru clasa SceneManager trebuie sa abordati o strategie care sa va permita oricand sa adaugati setari noi in fisierul de configurare, pe care sa le puteti procesa apoi in program.
De asemenea, setarile pot sa apara in orice ordine. Elementele pot sa apara in orice ordine, fara sa se tina cont de valoarea id-ului.

Clasa SceneManager va avea o metoda de Init() care va parsa fisierul xml si va seta toate proprietatile necesare.

O proprietate importanta este camera, sau, optional, un vector de obiecte de tip camera. Se va folosi clasa Camera implementata in cadrul taskurilor introductive. In cazul in care avem vector

de obiecte de tip camera, vom mai avea si o proprietate numita activeCamera, care va tine minte care este camera curent folosita.

Clasa Scene Manager va cuprinde un vector de (pointeri la) obiecte (SceneObject). Acestea sunt obiectele care vor fi desenate in scena.
De asemenea va exista si un vector cu Luminile din scena (explicate mai tarziu). Pentru moment puteti sa sariti peste aceasta proprietate.

Metodele importante:

- Init() - in care se va crea si vectorul de obiecte. Atentie, trebuie apelat dupa ce s-a apelat Init-ul lui resourceManager. Aceasta functie trebuie apelata in Init-ul frameworkului.
- Draw(), care va face setarile generale pentru desenarea unui frame si apoi va itera prin vectorul de obiecte, apeland Draw-ul lor
- Update() - updatarea unor valori care nu tin in mod direct de desenarea scenei. Folosit de exemplu pentru preluarea coordonatelor cursorului, ca sa se observe daca s-a facut click pe un obiect din scena. In afara de asta, in cazul in care si obiectele au o functie de Update va itera prin vectorul de obiecte si va apela Update-ul lor.

Clasa SceneObject
Va reprezenta un obiect de desenat in scena. Proprietatile acestuia sunt cele din xml. Atentie, unele proprietati pot lipsi (sunt optionale) cum ar fi cele de traiectorie, sau cele pentru modele generate automat.
Vor exista in scena si obiecte speciale (cu un comportament deosebit). Aceste obiecte sunt identificate prin type(din xml). Pentru aceste obiecte se vor defini clase speciale care extind clasa SceneObject. Pentru obiectele normale, type are valoarea *normal*. Daca insa e un obiect special, de exemplu skybox sau terrain, type va indica tipul respectiv de obiect. In functie de valoarea lui type se va crea un obiect special, de exemplu TerrainObject, sau un obiect normal. Toate obicetele, insa, indiferent de tipul lor, se pun in vectorul de obiecte.
Va exista in cadrul acestei clase o metoda Draw() care deseneaza obiectul fie in format normal (cu primitive pline) fie in format wired (doar muchiile), in functie de valoarea proprietatii wiredFormat.
Proprietatile speciale vor fi explicate pe masura ce se dau taskurile. Proprietatile esentiale pentru a avea un game engine functional (in stadiu incipient) sunt:

- id (identificatorul din XML)
- position (indica pozitia in scena a obiectului -pe baza acestui vector se face matricea de translatie)
- rotation (Vector3 cu unghiurile de rotatie fata de OX, OY, OZ => matricile de rotatie)
- scale (Vector3 continand scalarile pe fiecare dimensiune => matricea de scalare)
- model (pointer catre modelul incarcat)
- shader (pointer catre shaderul incarcat)
- textures (vector de pointer catre texturi incarcate)
- depthTest (daca necesita sau nu DEPTH_TEST)