**Reprezentarea obiectelor in scena 3D (matricea MVP)**

Se presupune cunoscut faptul că pentru a aplica o transformare geometrică precum scalare, rotație, translație, înmulțim coordonatele punctelor/vectorilor (pe care vrem să îi transformăm) cu matricea de transformare corespunzătoare.

Pentru a putea plasa obiectul in scena si apoi pentru a ne putea “deplasa” in scena creata trebuie sa avem in vedere 3 matrici:

* **Model** (notata *M*) – este matricea care cuprinde transformarile aplicate asupra obiectului: scalari, rotatii, translatii (in aceasta ordine). Rotatiile se aplica dupa regula roll-pitch-yaw (fata de axa Ox, fata de axa Oy, fata de axa Oz). Vedeti si: <http://msl.cs.uiuc.edu/planning/node102.html>
* **View** (notata *V*) – matricea care ne permite sa ne deplasam in scena, folosind punctul abstract numit *camera.* Practic daca vrem sa ne deplasam, de exemplu, la dreapta, atunci scena, relativ la noi se va deplasa la stanga. Aceasta deplasare va fi continuta de View.
* **Projection** (notata *P*) – matricea care proiecteaza obiectele in perspectiva (astfel incat obiectele mai apropiate apar in scena mai mari, iar cele mai departate apar mai mici).

Pentru a reprezenta obiectul in scena, vertecsii lui vor fi inmultiti cu matricea *MVP=M\*V\*P.*

# **Clasa Camera**

| **Proprietati** | **Metode** |
| --- | --- |
| * Vector3 position (punctul în care se află camera) * Vector3 target (punctul spre care “se uită” camera) * Vector3 up (direcția “în sus” a camerei) * GLfloat moveSpeed (viteza de translație a camerei) * GLfloat rotateSpeed * GLfloat nearPlane * GLfloat farPlane * GLfloat fov * GLfloat deltaTime * Vector3 xAxis * Vector3 yAxis * Vector3 zAxis * Matrix viewMatrix * Matrix worldMatrix | * void moveOx(int sens) * void moveOy(int sens) * void moveOz(int sens) * void rotateOx(int sens) * void rotateOy(int sens) * void rotateOz(int sens) * void updateAxes() //actualizează coordonatele axelor camerei exprimate în coordonatele scenei (xAxis, yAxis, zAxis) * void updateWorldView() //actualizeaza întâi axele apelând updateAxes(), apoi matricile World și View * void setDeltaTime(GLfloat) * constructori:   + un constructor fără parametri undeva setați valorile proprietăților cum doriți voi ca să puteți vizualiza scena – acest constructor în general va fi folosit pentru testări rapide   + un constructor care are drept parametri valori pentru toate proprietățile camerei: position, target, up, moveSpeed, rotateSpeed, nearplane, farPlane, fov. Constructorul va avea pentru fiecare parametru (in afară de position, target și up) câte o valoare implicită. |

Pentru proprietățile position, target și up (căt și pentru axe: xAxis, yAxis, zAxis) se poate alege și tipul Vector4. Daca alegeti Vector4 pentru proprietăți va trebui uneori sa faceti conversia la Vector3 sau sa definiti o parte din operatii pentru Vector4. Daca alegeti Vector3 va trebui sa faceti conversia la Vector4 cand inmultiti cu matricile de transformare. Prin urmare, indiferent de tipul ales, trebuie să definiți în Math.h/cpp și o metodă de conversie de la Vector3 la Vector4 sau invers.

**Important!** Un punct scris în format Vector4 are întotdeauna coordonata w nenulă. Un vector liber (segment orientat) are coordonata w nulă.

Proprietatea deltaTime va fi setată în funcția Update a framework-ului folosind funcția setter dedicată (va reprezenta timpul care a trecut intre doua update-uri și deci între două frame-uri).

## **Reprezentarea versorilor sistemului de coordonate al camerei în coordonatele scenei.**

**Aceste formule vor fi folosite în metoda UpdateAxes()**

*zAxis = -(target - position).Normalize(); //observație, în funcție de sistemul ales pentru coordonatele camerei (sistem right/left handed), minusul de la început poate să lipsească*

*yAxis = up.Normalize();*

*xAxis = zAxis.Cross(yAxis).Normalize();*

## Formule pentru implementarea metodelor camerei

### Translații

**Mișcare pe o axă a camerei (axa poate fi xAxis, yAxis, zAxis). Aceste formule vor fi folosite în metodele moveOx, moveOy, moveOz.**

*forward = axis \* sens;*

*vectorDeplasare = forward \* moveSpeed \* deltaTime;*

*position += vectorDeplasare;*

*target += vectorDeplasare;*

### Rotații

Pentru rotații se vor calcula matricile de rotație folosind metodele clasei Matrix din Math.h. Acestea primesc ca parametru (float) un unghi care este calculat după formula:

**unghiRotatie = sens\*rotateSpeed\*deltaTime**

unde rotateSpeed și deltaTime sunt proprietățile din clasa Camera, iar sens este dat ca parametru în metoda de rotație a camerei și dă sensul de rotație (poate avea valorile +/-1).

Observație: metodele care încep cu prefixul "set" din clasa Matrix schimbă valorile din matrice. De exemplu dacă dorim să facem o matrice de rotație cu unghiul 0.3 radiani, față de axa OY am scrie:

Matrix mr;

mr.SetRotationY(0.3);

În formulele de mai jos, *localTarget* și *localUp* se referă la coordonatele punctului *target* și vectorului *up* în sistemul local al camerei. De exemplu, localUp este versorul pentru OY al camerei, deci va avea coordonatele (0,1,0,0). Punctul corespunzător proprietății *position* a camerei, în spațiul local al camerei ar fi chiar originea sistemului local de coordonate, deci (0,0,0).

Observație: localTarget și localUp sunt de tip Vector4 pentru a putea fi înmulțite cu matricile de transformare. Totuși target și up sunt Vector3.

**Aceste formule vor fi folosite în metodele rotateOx, rotateOy, rotateOz.**

#### **Rotație față de OY**

*localTarget = Vector4(0.0f, 0.0f, -(target - position).Length(), 1.0f);*

*rotatedTarget = localTarget \* mRotateOY;*

*target = rotatedTarget \* worldMatrix;*

#### **Rotație față de OX**

*rotatedLocalUp = localUp \* mRotateOX;*

*up = rotatedLocalUp \* worldMatrix*

*up = up.Normalize();*

*localTarget = Vector4(0.0f, 0.0f, -(target - position).Length(), 1.0f);*

*rotatedTarget = localTarget \* mRotateOX;*

*target = rotatedTarget \* worldMatrix*

**Rotație față de OZ**

Exercițiu

**Atenție! La finalul fiecărei funcții de translație sau rotație trebuie apelată metoda updateWorldView() pentru a lucra cu matricile actualizate (acestea depind de proprietățile position, target, up). Nu uitați să apelați funcția updateWorldView() și în constructori, după ce setați valorile parametrilor.**

### **Matricile worldMatrix si viewMatrix**

**Aceste formule vor fi folosite în metoda updateWorldView().**

*worldMatrix = R\*T*

unde T reprezinta matricea de translație pentru vectorul (position.x, position.y, position.z);

*viewMatrix =*

*R* este matrice ortogonala =>

*viewMatrix= \**

= *R* transpus

iar e chiar translația cu vectorul (-position.x, -position.y, -position.z)

Matricea de rotație are formula de mai jos: :

*R*=

Valorile se vor seta pentru variabila R direct în proprietatea ***m*** care e de tip GLfloat[4][4], declarată deja în clasa Matrix. De exemplu:

R.m[0][0]=x.axis.x; //etc.

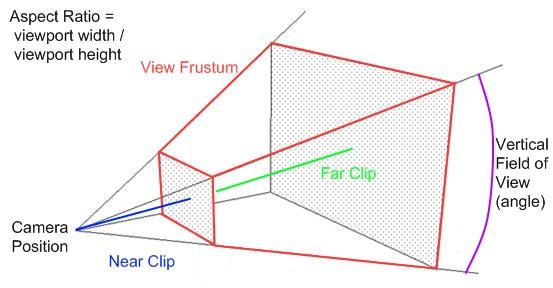
# **Proiecția**

Tipurile de proiectii cu care vom lucra sunt:

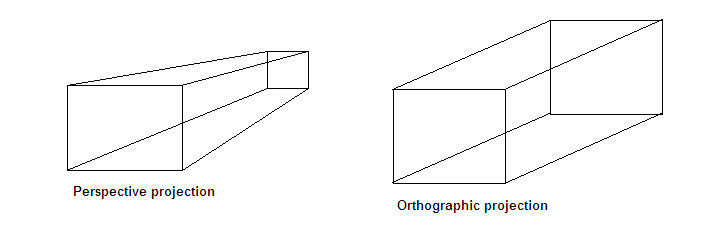
* **1) proiectia in perspectiva** (folosita pentru obiectele in scena 3D). Folositi setPerspective() din Math.h.

Avem 3 date importante pentru proiectia in perspectiva:

* **fov** (field of view) – deschiderea trunchiului de piramida (unghiul din varf)
* planul **near** (baza mica a trunchiului de piramida; planul mai apropiat de *camera*).
* planul f**ar** (baza mare a trunchiului de piramida; planul mai departat de *camera*).
* **Aspectul** (aspect ratio) = widthFerestra/heightFereastra

(imagine preluata de la: <https://ksgamedev.wordpress.com/tag/camera/> ) 

* **2) proiectia ortografica, numita si ortogonala** (folosita in general pentru ecrane 2D – cum ar fi ecrane de setari, meniuri etc)

(imagine preluată de la: <http://gamedev.stackexchange.com/questions/76111/open-gl-perspective-projection-vs-orthographic-projection> )

Cititi si:

* <http://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix>
* <https://wiki.blender.org/index.php/Doc:RO/2.4/Manual/3D_interaction/Navigating/3D_View> (aici gasiti si o imagine care compara cele doua tipuri de proiectii)

Pentru a aplica perspectiva, se va crea o proprietate numită perspectiveMatrix (de tip Matrix) în clasa Camera, care va fi calculată în constructor.

# În shader

Pentru a aplica transformările M (model), V(view) și P (perspective), trimitem matricea MVP către shader drept variabilă uniform de tip mat4.

gl\_Position = uMVP \* vec4(a\_posL, 1.0);

# Cerințe

1. Implementați clasa Camera împreună cu metodele sale. Puteți considera drept coordonate inițiale (în caz că implementați camera imediat după taskul cu triunghiul):
   1. position(0,0,-1) - poate fi și (0,0,1) dar va vedea triunghiul din partea cealaltă. în niciun caz nu setați coordonata z la 0 (deoarece triunghiul e la z=0, și se vor suprapune, deci nu îl veți mai vedea în scenă)
   2. target(0,0,0) - ca să se uite spre triunghi
   3. up(0,1,0) - fiindcă de fapt e OY local (deci x=0, z=0, y=1 pentru că e versor)
   4. fov (45 grade= 0.785398radiani)
   5. nearPlane(0.2) - ca să fie foarte aproape de camera, dar nu chiar egal cu 0 (altfel am avea împărțire la 0 în formulă)
   6. farPlane(10) - sau oricum o valoare strict mai mare decât 1 fiindcă trebuie să cuprindă triunghiul (al cărui plan e la distanță 1 față de camera)
   7. moveSpeed(0.1) - viteza de translație
   8. rotateSpeed(0.1) - viteza de rotație (se consideră în radiani/s)
2. Pentru a evita un frame rate variabil (scena se randează în continuu la intervale mici de timp dar și la fiecare eveniment creat de utilizator), stabiliți unui interval minim între realizarea operațiilor din funcția Update a framework-ului (de exemplu 0.05 secunde). Modul de implementare este următorul:
   1. Pentru acest interval de timp, veți defini o proprietate statică cu numele frameTime de tip float în fisierul Globals.h. Atenție, fiind de tip float, trebuie să o inițializați în Globals.cpp (pe care îl veți crea; nu uitați să includeți în el Globals.h)
   2. În fisierul NewTrainingFramework.cpp, veți folosi o variabilă float globală (de exemplu numită totalTime) care e inițial 0.
   3. În funcția Update veți incrementa această variabila cu timpul care a trecut între doua frame-uri (deci cu deltaTime).
   4. Operațiile de update se vor realiza doar când totalTime depășește frameTime (veți verifica acest lucru cu o instrucțiune if. **Toate operațiile de Update se vor realiza în acest if).**
   5. În cadrul if-ului, totalTime va fi resetat la totalTime-frameTime. **Atenție, proprietatea deltaTime a camerei va fi setată la frameTime în interiorul if-ului!**
3. Folosind functia Key() a framework-ului, afisati tastele apasate si observati ce se intampla (care e rolul parametrului bIsPressed din funcția Key?)
4. Asociați tastelor w,s,a,d,q,e translatia camerei inainte, inapoi, stanga, dreapta, sus si jos. Sagetile vor fi folosite pentru rotatii (+ inca o pereche de taste aleasa de voi pentru rotatia fata de axa OZ). Pentru asta veți folosi funcția Key() a framework-ului. Veți folosi o instrucțiune de tip switch. Pentru litere, veți testa codul ASCII din parametrul key și pentru litere mici si pentru litere mari. Pentru tastele săgeți veți compara parametrul key cu tastele virtuale VK\_UP, VK\_DOWN, VK\_LEFT, VK\_RIGHT).

Observație: pentru case se poate folosi direct și tipul char. Adică în loc de

case 65:

puteti scrie:

case 'A':

deoarece în C++ char este văzut tot ca un tip de date numeric, deci 'A' este de fapt echivalent cu valoarea 65.

De asemenea, deoarece și pentru literele mici și pentru cele mari aveți aceleași instrucțiuni, puteți scrie case-uri înlănțuite:

case 'A': case 'a':