UNIVERSITATEA “TITU MAIORESCU” DIN BUCUREŞTI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

COORDONATOR ŞTIINȚIFIC:

Conf.univ.dr.ing. Mironela Pîrnău

ABSOLVENT:

Bogdan Florin Ioniță

SESIUNEA IUNIE-IULIE

2017

UNIVERSITATEA “TITU MAIORESCU” DIN BUCUREŞTI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Implementarea transformărilor grafice folosind OpenGL

COORDONATOR ŞTIINȚIFIC:

Conf.univ.dr.ing. Mironela Pîrnău

ABSOLVENT:

Bogdan Florin Ioniță

SESIUNEA IUNIE-IULIE

2017

UNIVERSITATEA TITU MAIORESCU

FACULTATEA DE INFORMATICĂ

DEPARTAMENTUL DE INFORMATICĂ

REFERAT

DE APRECIERE A LUCRĂRII DE LICENȚĂ

***TITLU**: Implementarea transformărilor grafice folosind OpenGL*

***ABSOLVENT:** Bogdan Florin Ioniță*

***PROFESOR COORDONATOR:** Conf.univ.dr.ing. Mironela Pîrnău*

Referitor la conținutul lucrării, fac următoarele aprecieri:

A.	Conținutul științific al lucrării	1 2 3 4 5 6 7 8 9 10
B.	Documentarea din literatura de specialitate	1 2 3 4 5 6 7 8 9 10
C.	Contribuția proprie	1 2 3 4 5 6 7 8 9 10
D.	Calitatea exprimării scrise și a redactării lucrării	1 2 3 4 5 6 7 8 9 10
E.	Conlucrarea cu coordonatorul științific	1 2 3 4 5 6 7 8 9 10
F.	Realizarea aplicației practice	1 2 3 4 5 6 7 8 9 10
	Punctaj total = (A+B+C+D+E+2F)/7	

În concluzie, consider că lucrarea de licență întrunește/ nu întrunește condițiile pentru a fi susținută în fața comisiei pentru examenul de licență din sesiunea iunie-iulie 2017 și o apreciez cu nota_____.

CONDUCĂTOR ȘTIINȚIFIC,

_____

I. CUPRINS

II. INTRODUCERE

OpenGL este o librarie de nivel jos cu ajutorul careia pot fi create aplicatii cu continut grafic bogat, atat 2D cat si 3D, folosind capabilitatile de randare grafica accelerata ale chipsetului dedicat, daca acesta este prezent.

OpenGL a fost creat initial ca o librarie pentru limbajul C, dar ulterior au fost dezvoltate interferete peste aceasta pentru majoritatea limbajelor de programare comune, inclusiv pentru WEB (javascript / WebGL)

Spre deosebire de alte API-uri grafice (cum ar fi DirectX, Vulkan etc), OpenGL reflecta mai indeaproape arhitectura si modul de functionare al hardware-ului, avand o structura relative simpla, dar care ofera toate functiile necesare si o performanta de top.

Structurile de date si operatiile aplicate pe acestea in lucrul cu OpenGL se muleaza perfect pe echivalentele lor matematice, astfel avem de a face cu spatii vectoriale, vectori, operatii pe vectori (produs scalar / produs vectorial), baze (un set de trei axe ortogonale intre ele) si transformari de la o baza la alta (reprezentate prin matrici).

Transformarile in teoria spatiilor vectoriale reprezinta operatii de traducere a unor vectori exprimati intr-o anumita baza in vectori echivalenti, dar exprimati in cadrul unei baze diferite, atat prin pozitionare cat si prin orientare sau dimensiune relativa.

Lucrarea de fata isi propune sa exploreze acest univers matematic al spatiilor vectoriale si al transformarilor, precum si aplicabilitatea lor practica in programarea grafica 3D, folosind limbajul C++.

III. CONTINUTUL LUCRARII

3.1 Definitii

In algebra, un spatiu vectorial reprezinta o multime de obiecte numite vectori, pe care sunt definite urmatoarele operatii: adunarea a doi vectori, inmultirea unui vector cu un scalar (numar real de obicei), produsul scalar a doi vectori, produsul vectorial a doi vectori. Exista si spatii vectoriale ale caror scalari pot fi numere complexe sau numere rationale. Operatiile de adunare a vectorilor si inmultire cu un scalar trebuie sa satisfaca niste cerinta, numite axiome, mentionate mai jos.

Un spatiu vectorial este caracterizat prin dimensiunea sa, care, in termeni generali, reprezinta numarul de directii independente din acel spatiu. Un element (vector) dintr-un spatiu n-dimensional va avea n componente – fiecare componenta reprezinta o coordonata pe axa corespunzatoare din spatiu.

Din punct de vedere istoric, primele idei referitoare la spatii vectoriale pot fi regasite in secolul al 17-lea in geometria analitica, sisteme de ecuatii liniare si vectori euclidieni. In forma actuala, mai abstracta, formulata de Giuseppe Peano in 1888, spatiile vectoriale pot fi construite dintr-o paleta mai larga de obiecte decat vectori euclidieni, cum ar fi functii si alte obiecte matematice, dar in mare teoria poate fi privita ca o extensie a geometriei clasice in care apar drepte, planuri si analoagele lor in mai multe dimensiuni.

Axiomele spatiilor vectoriale:

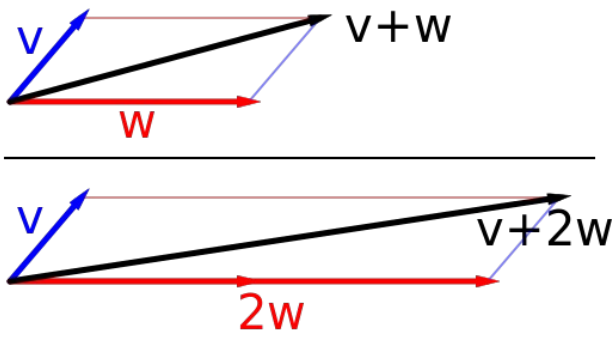
- Asociativitatea adunarii:
  - $u + (v + w) = (u + v) + w$
- Comutativitatea adunarii:
  - $u + v = v + u$
- Element neutru la adunare:
  - exista $0 \square V$ astfel incat $v + 0 = v$, oricare ar fi $v \square V$
- Element opus la adunare:

- pentru orice v □ V exista un element -v □ V astfel incat v + (-v) = 0

- Compatibilitatea inmultirii cu un scalar fata de inmultirea scalarilor:
  - a(bv) = (ab)v
- Element neutru la inmultirea cu un scalar:
  - **1**v = v
- Distributivitatea inmultirii cu un scalar fata de adunarea vectorilor:
  - a(u+v) = au + av
- Distributivitatea inmultirii cu un scalar fata de adunarea scalarilor:
  - (a+b)v = av + bv

3.2 Proprietati ale spatiilor vectoriale

In continuare vom face referire la spatiile vectoriale bi sau tridimensionale (2, respectiv 3 directii independente) peste numere reale (fiecare coordonata este exprimata printr-un numar real) intrucat acestea fac obiectul lucrarii de fata.



*Aditia vectorilor si inmultirea cu un scalar: sus, un vector **v** este adunat cu un alt vector **w**;*

*jos, vectorul **w** este scalat cu 2, apoi adunat cu **v***

Fie spatiul vectorial $V^3(\mathbf{R})$, cu elemente de forma

$$v = (v1, v2, v3), \text{ unde } v1, v2, v3 \in \mathbf{R}.$$

Elementul v0(0, 0, 0) se numeste “originea” spatiului V.

Punctele v1(1, 0, 0), v2(0, 1, 0), v3(0, 0, 1) impreuna cu v0 definesc 3 semi-drepte ortogonale intre ele, orientate de la origine catre v1, v2, respectiv v3. Deoarece sunt ortogonale intre ele, aceste semidrepte definesc 3 directii liniar-independente, formand deci baza canonica a spatiului V.

Vectorii i=v1 – v0, j=v2 – v0, k=v3 – v0, componentele bazei canonice, sunt folositi pentru a exprima intr-o forma convenabila orice vector din spatiul V sub forma:

$$v = v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$$

In aceasta expresie, scalarii v_x, v_y si v_z reprezinta coordonatele vectorului v in baza canonica a spatiului V, iar vectorii **i**, **j**, **k** reprezinta axele bazei.

3.3 Baze si coordonate

Intr-un spatiu vectorial de forma $V^3(\mathbf{R})$ pot fi definite oricate baze cu orientari ale axelor diferite, dar cu origine comuna in (0, 0, 0). Din punct de vedere matematic, aceste baze nu trebuie neaparat sa aiba axele perpendiculare intre ele (liniar independente), insa in majoritatea cazurilor, in practica vom folosi doar baze ortogonale.

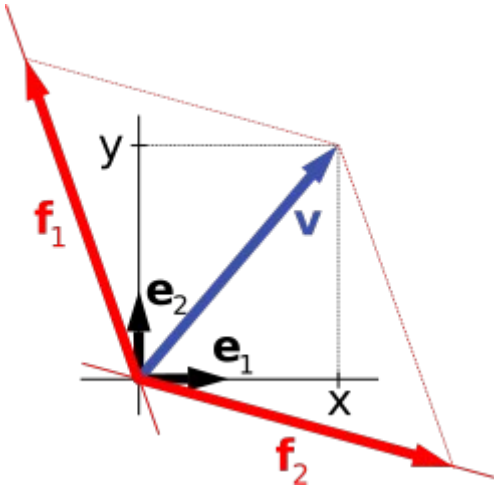
Fie baza $B_0 = \{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ si $B = \{\mathbf{r}, \mathbf{s}, \mathbf{t}\}$. Un vector $\mathbf{v} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ din spatiul V poate fi exprimat relativ la oricare din aceste baze, dar va avea coordonate diferite in fiecare dintre ele.

Coordonatele vectorului v in baza B pot fi calculate cu ajutorul produsului scalar intre vectorul v si fiecare axa a bazei B:

$$v_x(B) = x' = \mathbf{v}*\mathbf{r},\ v_y(B) = y' = \mathbf{v}*\mathbf{s},\ v_z(B) = z' = \mathbf{v}*\mathbf{t}$$

Astfel putem acum exprima vectorul **v** relativ la baza B astfel:

$$\mathbf{v} = x'\mathbf{r} + y'\mathbf{s} + z'\mathbf{t}$$



*Un vector **v** exprimat in functie de doua baze diferite (**x**, **y**) si (**f1**, **f2**)*

Pentru calcularea produsului scalar, putem reprezenta coordonatele vectorilor sub forma unor matrici linie sau coloana, tinand cont ca aceste reprezentari sunt relative la o anumita baza:

$\mathbf{v}(B_0) = [x\ y\ z]$, $\mathbf{r}(B_0) = [r_x\ r_y\ r_z]$, $\mathbf{s}(B_0) = [s_{xs}r_y\ s_z]$, $\mathbf{t}(B_0) = [t_x\ t_y\ t_z]$

Astfel:

$x' = [x\ y\ z] * [r_x\ r_y\ r_z] = x*r_x+y*r_y+z*r_z$

$y' = [x\ y\ z] * [s_x\ s_y\ s_z] = x*s_x+y*s_y+z*s_z$

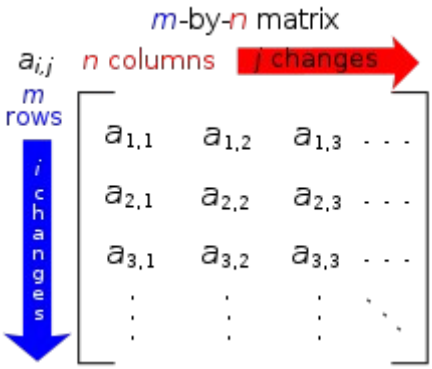
$z' = [x\ y\ z] * [t_x\ t_y\ t_z] = x*t_x+y*t_y+z*t_z$

Putem concatena aceste 3 operatii intr-una singura, inmultind vectorul **v** cu o matrice M ale carei coloane reprezinta coordonatele axelor bazei B:

$[x'\ y'\ z'] = [x\ y\ z] * \begin{bmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{bmatrix} = [x*r_x+y*r_y+z*r_z \quad x*s_x+y*s_y+z*s_z \quad x*t_x+y*t_y+z*t_z]$

sau pe scurt: $\mathbf{v}_B = \mathbf{v}_{B0} * M$

Matricea M se numeste **matricea de trecere** de la baza B_0 la baza B.



Structura unei matrici de transformare

3.4 Transformari

O transformare reprezinta operatia bijectiva de traducere a unui vector **v** intr-un alt vector **v'** prin inmultirea acestuia cu o matrice in modul indicat mai sus. In functie de tipul matricei, transformarea poate avea diferite valente:

- trecere de la o baza la alta
- rotatie in jurul unei axe care trece prin origine
- scalare
- deformare – cand se trece de la o baza ortogonala la una neortogonala sau invers

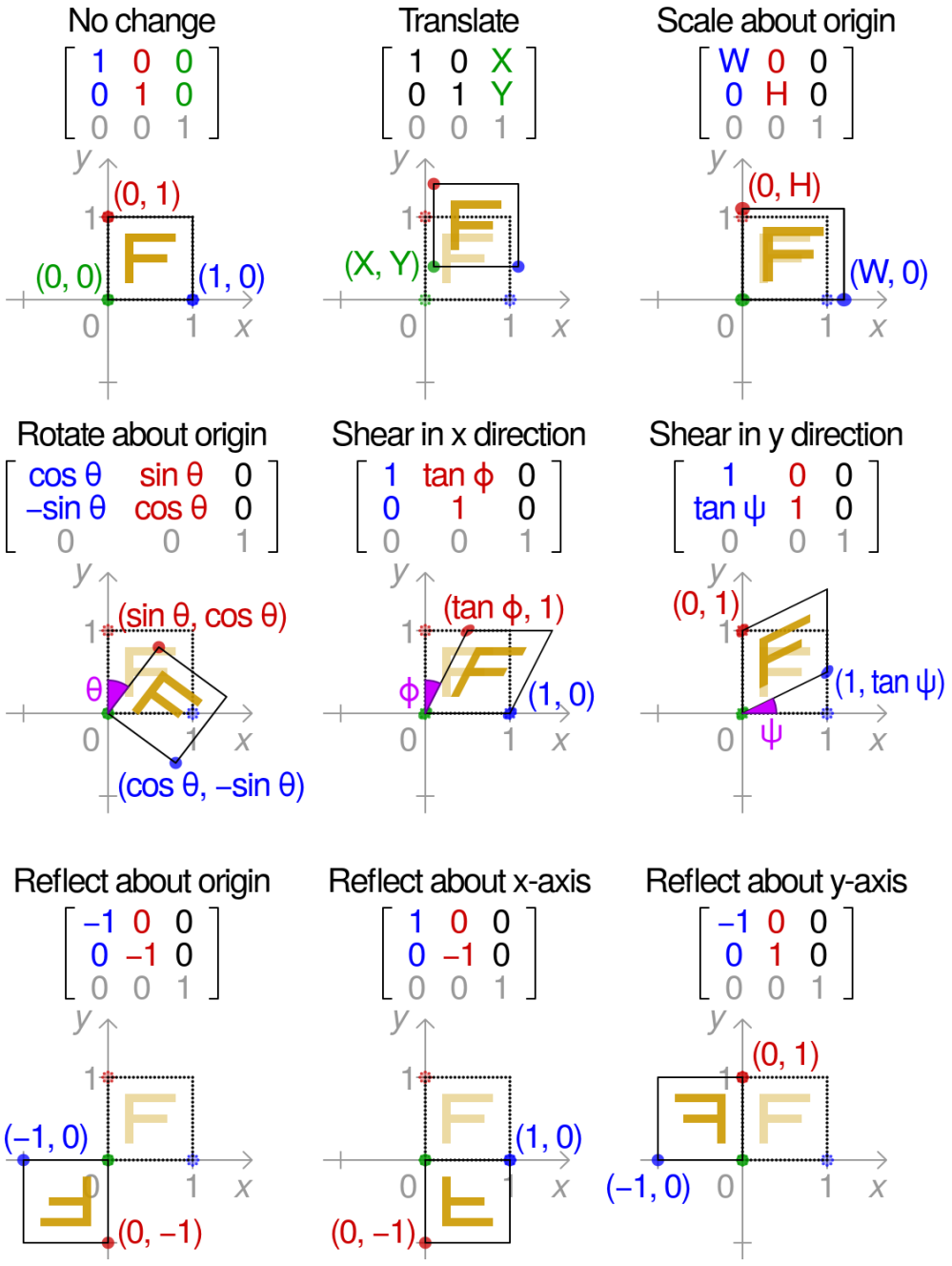
In principiu se poate construi o matrice de transformare pentru orice fel de manipulare geometrica. De asemenea matricile de transformare pot fi combinate, prin inmultirea lor.

Orice transformare (rotatie, scalare, etc) poate fi privita ca o trecere de la o baza la alta, daca consideram coordonatele vectorului initial definite relativ la baza initiala si transformarea ca matricea de trecere de la acea baza la baza finala. Acest aspect este important in realizarea practica a transformarilor geometrice in grafica pe calculator.

Din punct de vedere matematic, o transformare reprezinta o functie bijectiva care atribuie fiecarui vector **v** un alt vector **v'** dupa niste reguli clare, pastrand anumite relatii dintre elemente.

Pentru a putea reprezenta si translatiile in forma matriceala, este nevoie sa se foloseasca **coordonatele omogene**, codificand fiecare vector v de dimensiune n printr-un vector v' de dimensiune n+1, al carui ultima coordonata va fi 1. Astfel, in matricea transformarii ultima linie (sau coloana daca se foloseste inmultirea vector-coloana cu matrice) va contine translatia.

In continuare vom prezenta succint principalele tipuri de transformari, exemplificate pe un spatiu vectorial bidimensional pentru simplitate.



sursa: https://en.wikipedia.org/wiki/Transformation_matrix#/media/File:2D_affine_transformation_matrix.svg

3.4.1 Notiuni specifice in aplicatiile grafice

In aplicatiile grafice folosim notiunea de vector in doua moduri specifice:

- pentru a defini un punct in spatiu – **VERTEX**

- pentru a defini o directie - **NORMALA**

Normalele se folosesc de obiecei pentru a indica orientarea unei suprafete (sau poligon) pentru a putea calcula iluminarea intr-un mod realist.

Vertecsii se folosesc pentru a defini geometria obiectelor care trebuiesc randate.

Un obiect 3D consta intr-o lista de vertecsi, o lista de indecsi care definesc ordinea in care sunt parcursi acesti vertecsi pentru a forma triunghiuri, o lista de normale pentru fiecare triunghi in parte si alte proprietati (cum ar fi coordonate in textura etc).

In acest context aplicarea unei transformari se refera la aplicarea unei matrici de transformare peste toti vertecsii unui obiect pentru a-l manipula ca intreg (in anumite cazuri se transforma si normalele, nu doar vertecsii, dar pentru simplitate nu ne preocupam acest aspect).

3.4.2 Tipul notatiei si ordinea inmultirii

Este important sa tinem cont de modul in care notam vectorii (linie sau coloana) si matricile (coloane-axe sau linii-axe) deoarece acest aspect influenteaza ordinea in care trebuie inmultite atat matricile intre ele cat si pozitia in care apare vectorul in ecuatie (la stanga sau la dreapta).

Astfel, daca alegem notatia vector-coloana si matrice cu linii-axe, ordinea inmultirii este

$$M * v$$

Pentru ca inmultirea matricilor nu este comutativa, ordinea in care inmultim mai multe matrici ce reprezinta transformari va avea un impact asupra rezultatului final. Astfel, transformarea combinata aplicata vectorului **v,** va fi echivalenta cu o serie de transformari succesive, incepand de la cea mai apropiata de **v** inspre exterior:

Fie $M = M_1 * M_2$ atunci $M * v$ va avea acelasi efect ca $M_1*(M_2 * v)$, diferit de $M_2*(M_1*v)$

3.4.2 Translatia

Translatia reprezinta o transformare care “deplaseaza” uniform toti vertecsii pe o distanta **d,** independenta de pozitia initiala a vertecsilor:

$$\mathbf{v'_i} = \mathbf{v_i} * M(d) = \mathbf{v_i} + \mathbf{d}$$

Asadar distantele si pozitiile relative ale vertecsi sunt conservate de catre translatie, precum si orientarea atat relativa cat si absoluta a suprafetelor.

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

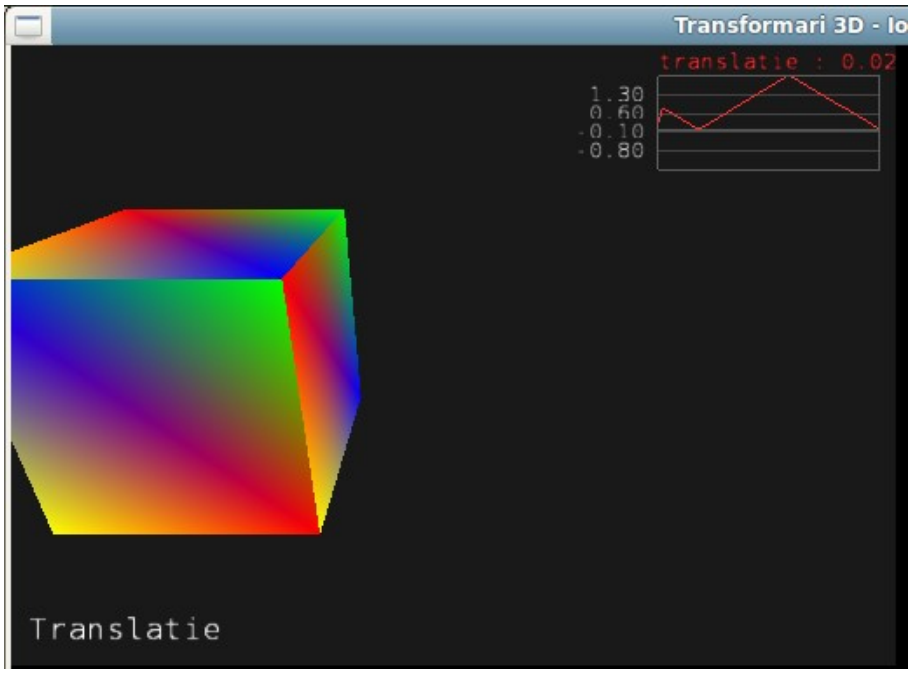
$$P' = P \cdot T$$

$$[X'\ Y'\ Z'\ 1] = [X\ Y\ Z\ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

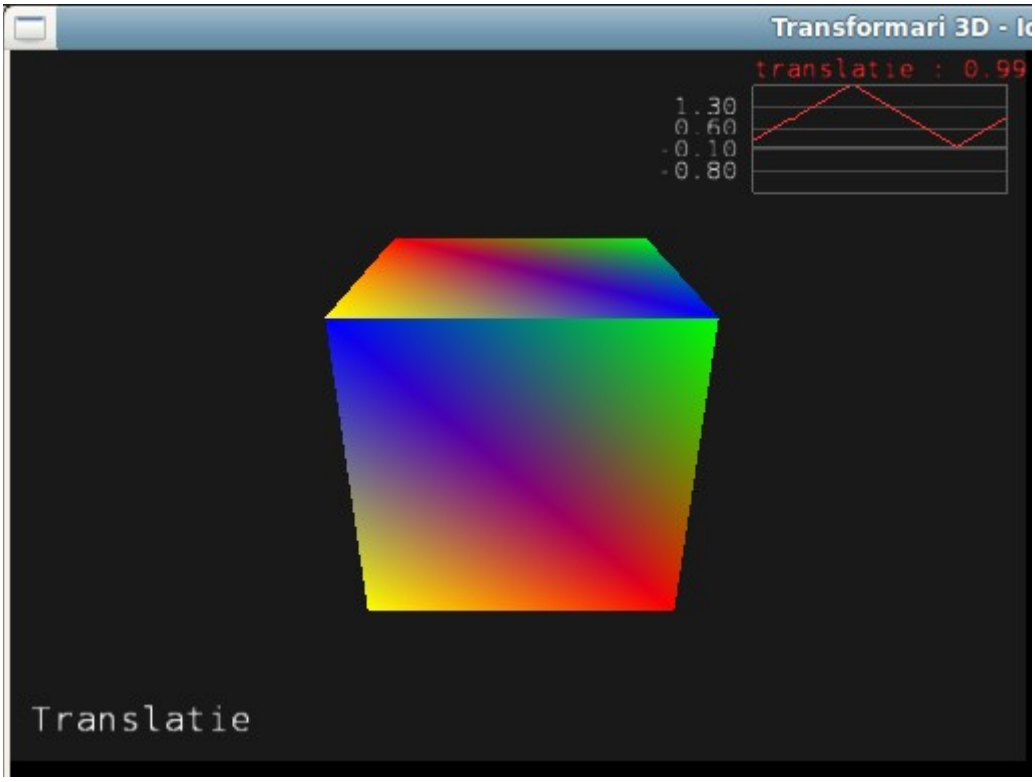
$$= [X + t_x\ \ Y + t_y\ \ Z + t_z\ \ 1]$$

Expresia matematica a translatiei

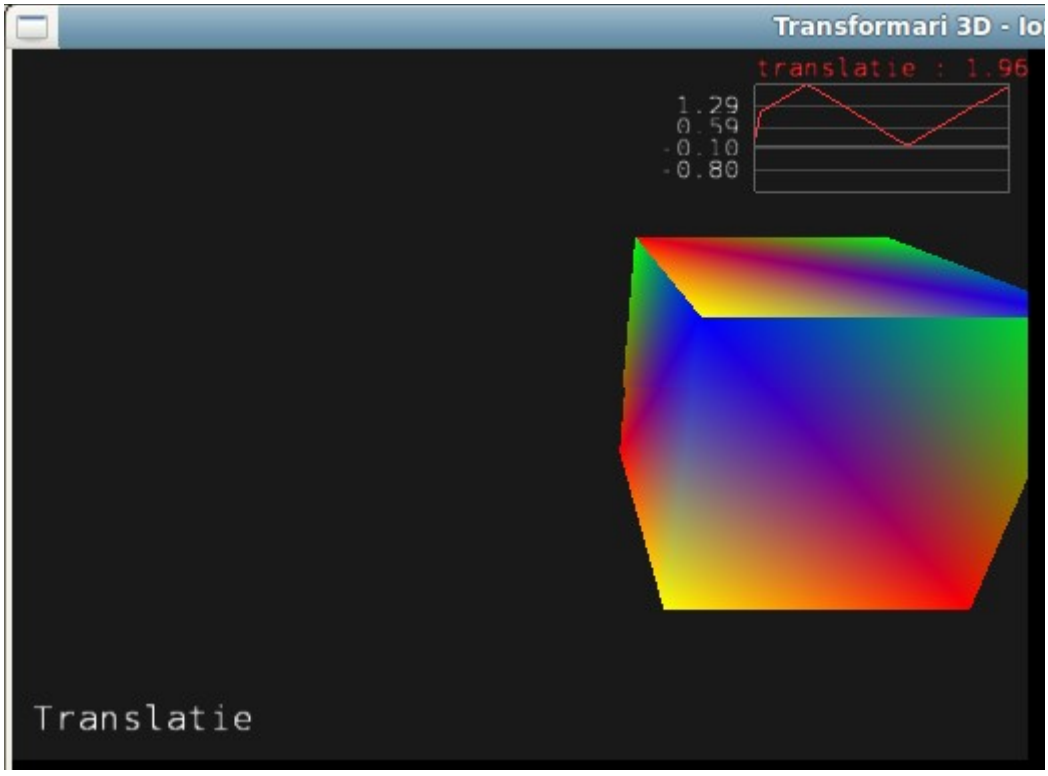
In exemplul de mai jos putem vedea translatia in actiune:



Pozitia originala z=-1, trans=0



pozitie intermediara z=0, trans=1



pozitie finala z=1, trans=2

Secventa anterioara a fost obtinuta printr-o operatie de interpolare liniara intre doua pozitii fixe (capetele), ceea ce ofera posibilitatea animarii imaginei intr-un mod facil:

```
auto box1 = std::make_unique<Box>(1, 1, 1);
auto pc = std::make_unique<PathController>(box1->body());
wld_->takeOwnershipOf(std::move(box1));
pc->addVertex({{-3, 0, -1}, glm::fquat()});  // pozitia initiala
pc->addVertex({{-3, 0, +1}, glm::fquat()});  // pozitia finala
pc->addRedirect(0);
pc->start(1.5f);
```

Actualizarea transformarilor obiectului din rezultatul interpolarii:

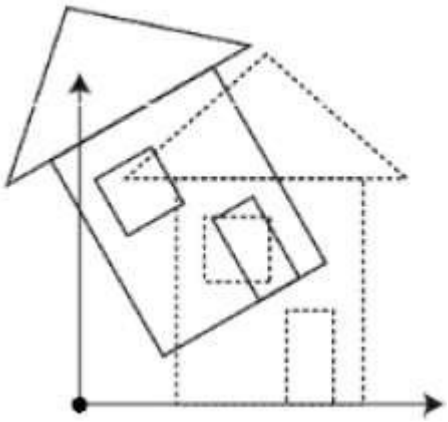
```
void PathController::update(float dt) {
        lerper_.update(dt);
        body_->setPosition(lerper_.value().position,
                        physics::DynamicBody::TransformSpace::Parent);
        body_->setOrientation(lerper_.value().orientation,
                        physics::DynamicBody::TransformSpace::Parent);
        body_->setScale(lerper_.value().scale);
}
```

Calcularea pozitiei obiectului:

```
void DynamicBody::setPosition(glm::vec3 pos, TransformSpace space) {
        switch (space) {
        case TransformSpace::Parent:
                break;
        case TransformSpace::World:
                // transform pos from world into parents
                if (parent_) {
                        auto mInv = glm::inverse(parent_->worldTransform());
                        pos = glm::vec3{mInv * glm::vec4{pos, 1}};
                }
                break;
        default:
                assert(!"invalid space");
        }
        transform_.position_ = pos;
        frameTransformDirty_ = wldTransformDirty_ = true;
}
```

3.4.3 Rotatia

Rotatia este de asemenea o transformare care pastreaza distantele si pozitiile relative dintre vertecsi si suprafete, insa orientarea per ansamblu a obiectului se modifica. Obiectul rezultat este asemenea celui initial.



Pozitiile vertecsilor sunt alterate de rotatie intr-un mod care depinde de pozitiile lor initiale, iar vertecsii care se gasesc pe axa de rotatie nu sunt modificati deloc.

Matricea de rotatie se construieste in functie de axa de rotatie aleasa, X, Y, sau Z. O matrice de rotatie poate fi obtinuta si prin inmultirea mai multor matrici de rotatie cu axe diferite, obtinandu-se astfel o rotatie compusa, care va avea o axa noua de rotatie, diferita de cele initiale.

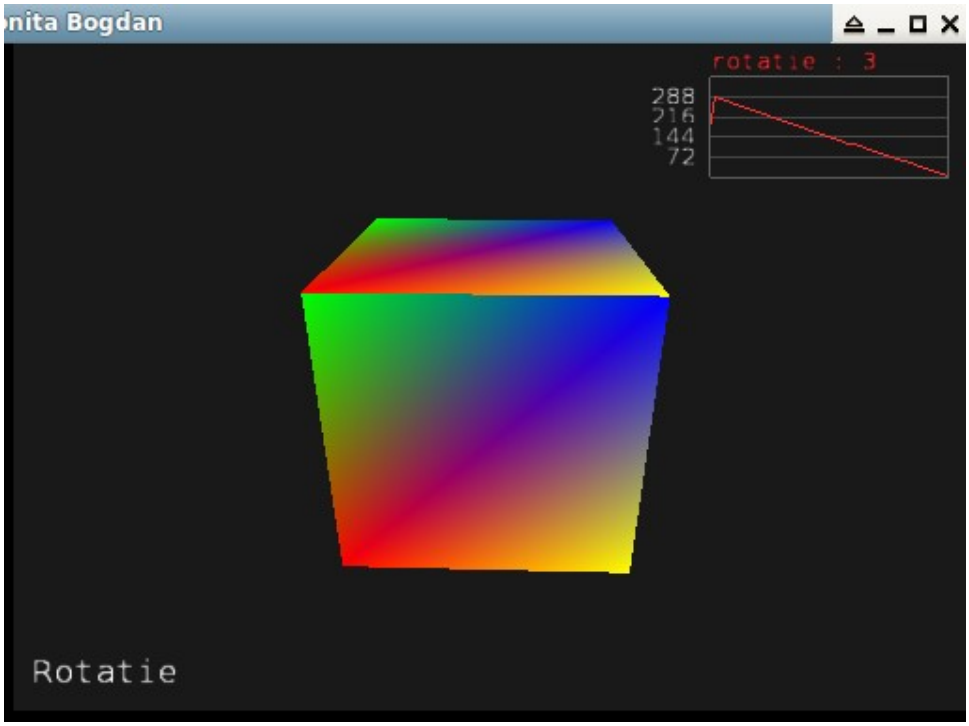
$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_z(\theta)$$

$$= \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

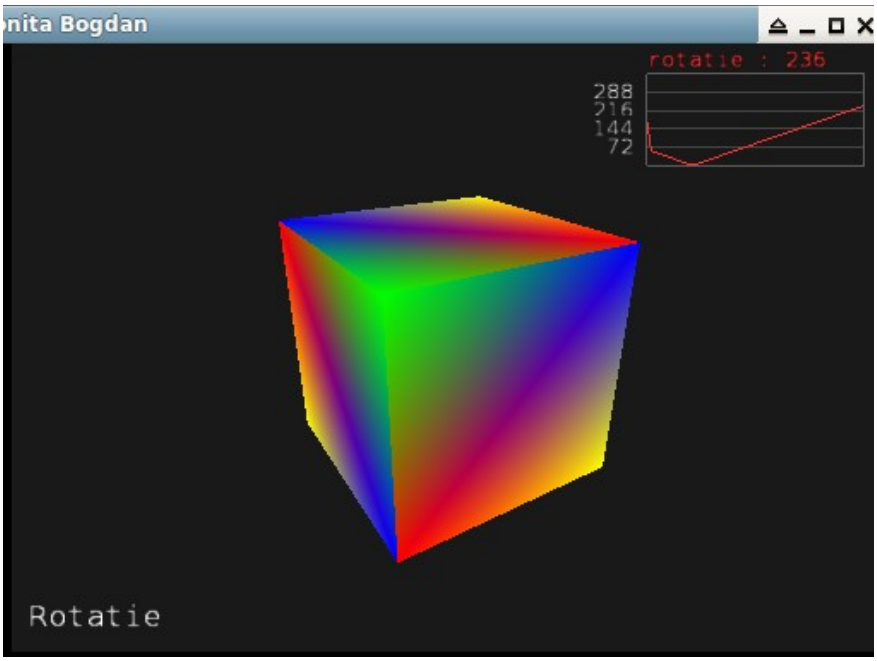
Matrici de rotatie pentru axele principale

In programarea grafica se folosesc deasemenea si rotatiile reprezentate prin **quaternioni**, acestia fiind niste obiecte matematice 4-dimensionale cu proprietati avantajoase, cum ar fi ca se pot combina usor intre ei si nu au anumite probleme pe care le au matricile, cum ar fi *driftingul*.

In exemplul de mai jos vom vedea rotatia in actiune:



orientarea initiala r_y=0



orientare intermediara r_y=240 grade

Secventa de interpolare pentru rotatie:

```
auto box2 = std::make_unique<Box>(1, 1, 1);
pc = std::make_unique<PathController>(box2->body());
wld_->takeOwnershipOf(std::move(box2));
pc->addVertex({glm::vec3{0.f, 0.f, 3.f},
                              glm::angleAxis(0.f, glm::vec3{0, 1, 0})});
pc->addVertex({glm::vec3{0.f, 0.f, 3.f},
                              glm::angleAxis(PI/1.5f, glm::vec3{0, 1, 0})});
pc->addVertex({glm::vec3{0.f, 0.f, 3.f},
                              glm::angleAxis(2*PI/1.5f, glm::vec3{0, 1, 0})});
pc->addRedirect(0);
pc->start(2.5f);
```

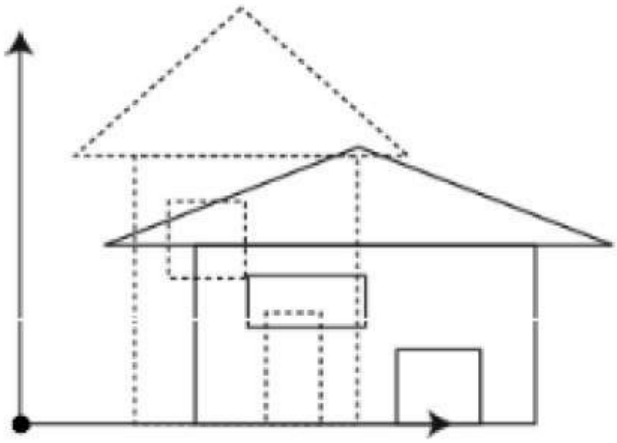
Se observa ca s-a folosit o rotatie in jurul axei {0, 1, 0} adica axa Y.

Calcularea orientarii obiectului:

```
void DynamicBody::setOrientation(glm::fquat o, TransformSpace space) {
        switch (space) {
        case TransformSpace::Parent:
                break;
        case TransformSpace::World:
                // transform pos from world into parent
                if (parent_) {
                        auto mInv = glm::inverse(parent_->worldTransform());
                        o = glm::quat_cast(mInv) * o;
                }
                break;
        default:
                assert(!"invalid space");
        }
        transform_.orientation_ = o;
        frameTransformDirty_ = wldTransformDirty_ = true;
}
```

3.4.4 Scalarea

Scalarea este o transformare care modifica atat distantele relative dintre vertecsi, cat si pozitiile relative ale acestora, pe una sau mai multe axe. Obiectul rezultat nu mai este asemenea obiectului initial decat daca se foloseste o scalare uniforma, adica cu aceiasi factori pe toate axele.



Scalarea intr-un spatiu tridimensional este definita prin 3 valori {Sx, Sy, Sz} care reprezinta factorii de scalare pe fiecare axa in parte.

Daca cei trei factori sunt egali, atunci scalarea se numeste *uniforma* si va pastra forma obiectului, altfel obiectul va aparea deformat (intins sau comprimat pe una sau mai multe directii).

Daca factorul de scalare pe o anumita axa este subunitar, atunci scalarea va avea micsora obiectul; daca este supraunitar, il va mari, iar daca este egal cu 1, scalarea nu are nici un efect.

$S < 1 \Rightarrow$ micsorare

$S > 1 \Rightarrow$ marire

$S = 1 \Rightarrow$ nici un efect

Distanta dintre doi vertecsi $\mathbf{v}_1$ si $\mathbf{v}_2$ este modificata de scalare astfel:

Fie vectorul $\mathbf{d} = \mathbf{v}_2 - \mathbf{v}_1$ astfel $|\mathbf{d}| = dist(\mathbf{v}_1, \mathbf{v}_2)$ reprezinta distanta liniara dintre cei doi vertecsi.

Fie scalarea $S = \{s_x, s_y, s_z\}$ si $\mathbf{v}_1'$ si $\mathbf{v}_2'$ vertecsii transformati, iar $\mathbf{d}' = \mathbf{v}_2' - \mathbf{v}_1'$, $|\mathbf{d}'|$ distanta dintre acestia. Coordonatele vectorilor $\mathbf{v}_1'$ si $\mathbf{v}_2'$ se calculeaza astfel:

$$\mathbf{v}_1' = \{\mathbf{v}_1.x * s_x, \mathbf{v}_1.y * s_y, \mathbf{v}_1.z * s_z\} \text{ si } \mathbf{v}_2' = \{\mathbf{v}_2.x * s_x, \mathbf{v}_2.y * s_y, \mathbf{v}_2.z * s_z\}$$

rezulta ca $\mathbf{d}' = \{s_x*(\mathbf{v}_2.x - \mathbf{v}_1.x), s_y*(\mathbf{v}_2.y - \mathbf{v}_1.y), s_z*(\mathbf{v}_2.z - \mathbf{v}_1.z)\}$, ceea ce inseamna ca distanta dintre vertecsii transformati, raportata la fiecare axa in parte, este inmultita cu factorul de scalare.

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

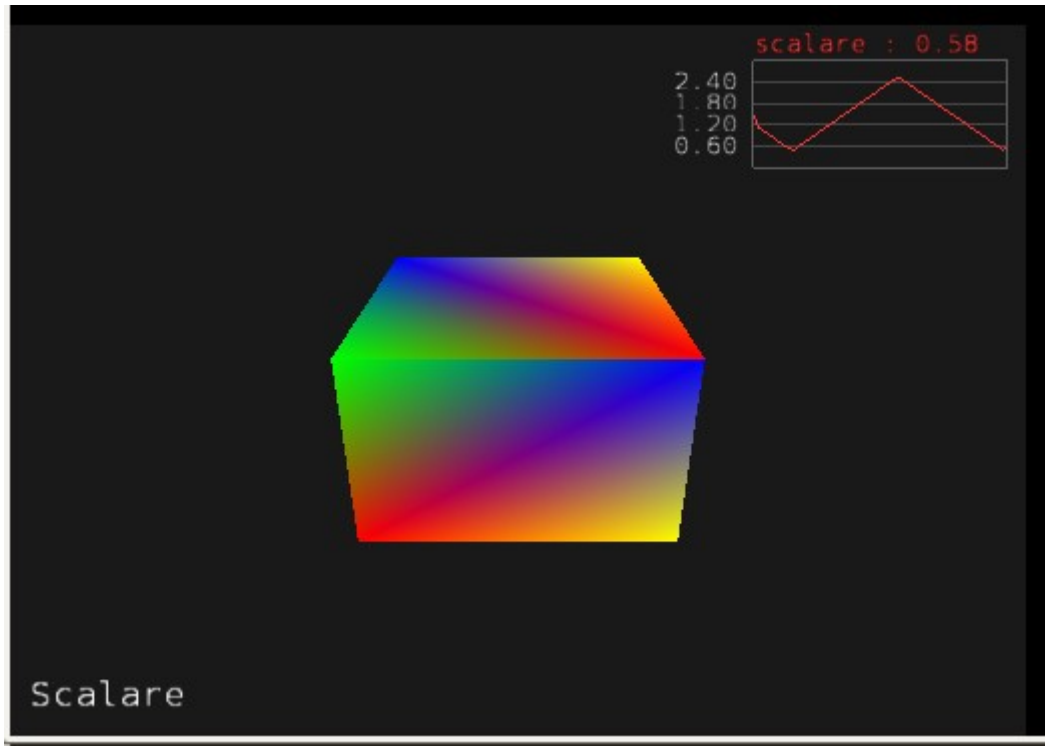
$$P' = P \cdot S$$

$$[X' \quad Y' \quad Z' \quad 1] = [X \quad Y \quad Z \quad 1] \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

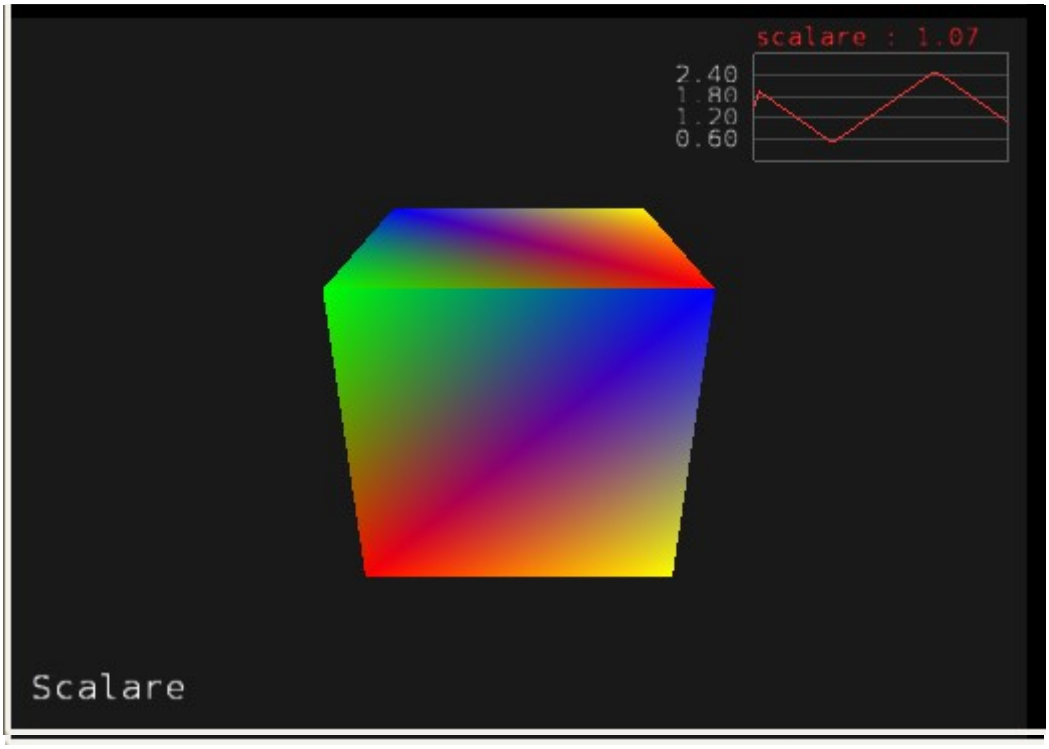
$$= [X.S_x \quad Y.S_y \quad Z.S_z \quad 1]$$

Structura unei matrici de scalare

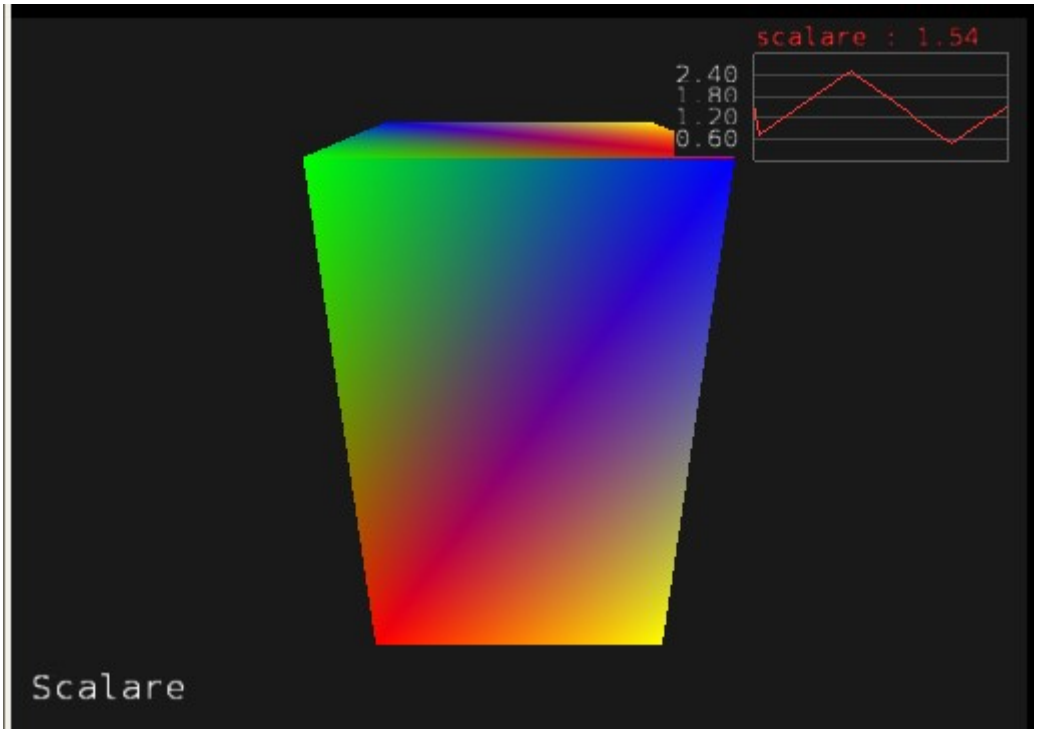
Mai jos vom observa scalarea in actiune:



Scalare pe axa Y cu factor subunitar $s_y = 0.58$



Scalare neutra cu factor ~= 1



Scalare pe axa Y cu factor supraunitar s_y=1.54

Un caz aparte al scalarii il reprezinta **reflexia**, in care se aplica un factor de scalare de -1 pe axa dorita.

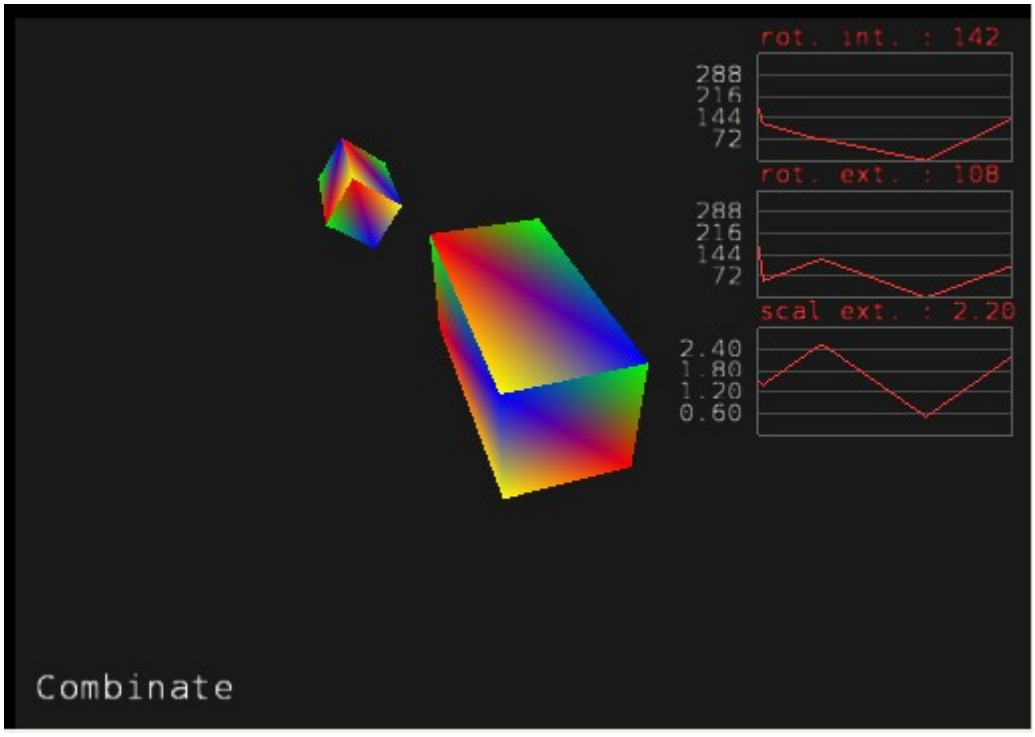
3.4.5 Combinarea transformarilor

Avantajul reprezentarii transformarilor sub forma matriceala este acela ca putem combina cu usurinta mai multe transformari, inmultind matricile acestora.

Deoarece inmultirea matricilor nu este comutativa, trebuie sa acordam atentie ordinii in care inmultim matricile de transformare, pentru ca o ordine diferita va conduce la efecte diferite, asa cum este si de asteptat: o translatie aplicata inaintea unei scalari va fi scalata si ea la randul ei, pe cand translatia aplicata dupa scalare va actiona ca atare.

Mai jos putem observa o combinare a urmatoarelor transformari:

1. cubul mic este rotit in jurul axei proprii X
2. cubul mic este pozitionat (translatat) relativ la cubul mare
3. cubul mic este atasat cubului mare si scalate impreuna pe axa X → axa X a cubului mic este rotita
4. cele doua cuburi sunt rotite impreuna pe axa Y → scalarea va fi si ea rotita



Combinarea a trei transformari

Astfel, cea de-a doua rotatie are ca efect si modificarea pozitiei cubului mic, deoarece si translatia aplicata anterior este rotita.

3.5 MODUL DE FUNCTIONARE AL OPENGL

Libraria OpenGL are doua moduri de functionare:

1. modul **legacy** (versiunea 1) – nu mai este de actualitate, folosirea sa este descurajata, insa este pastrat in continuare in scop academic.

2. modul standard actual (de la versiunea 2 in sus; versiunea curenta este 4)

Modul legacy se bazeaza pe ceea ce se numeste “fixed pipeline”, adica algoritmul de procesare a datelor in vederea obtinerii imaginii finale este fix, dar parametrizabil. Astfel, utilizatorul pune la dispozitia librariei date pentru diverse scopuri predefinite (vertecsi, texturi, indecsi etc) si configureaza diversi pasi ai procesului de randare prin comenzi date librariei.

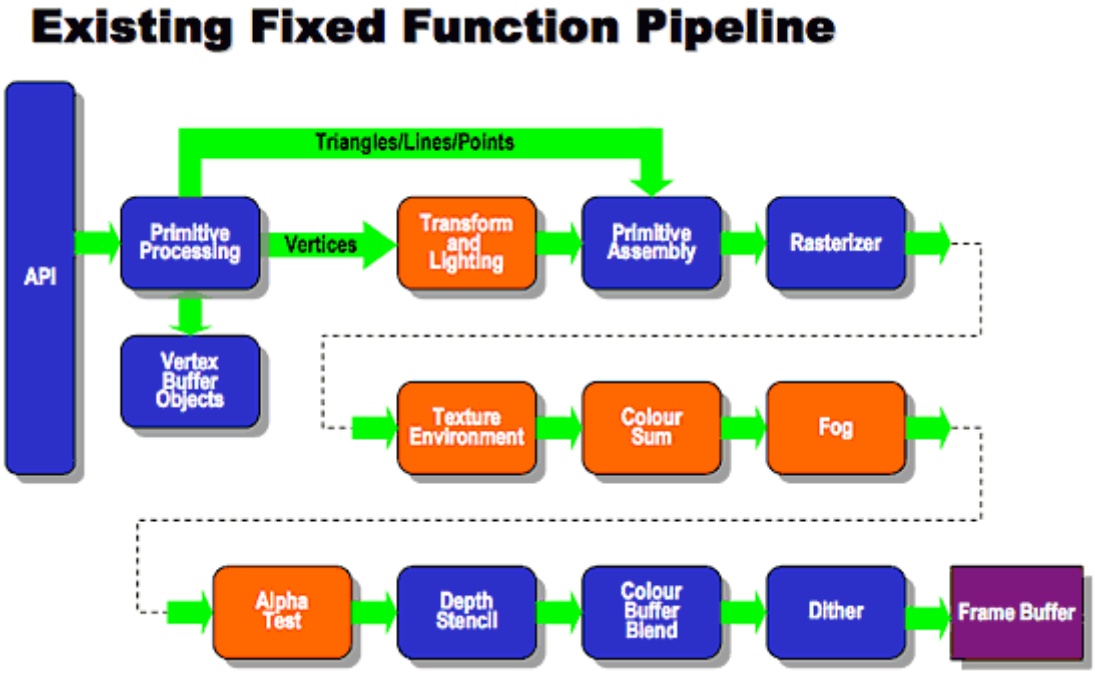


Diagrama procesului fixed-pipeline

Pasii principali ai procesului de randare sunt urmatorii:

- procesarea primitivelor → are ca rezultat o lista de verteci ce trebuiesc procesati ulterior
- transformare si iluminare → aplica transformarile (in modul legacy exista tipuri predefinite de transformari: **world** – pozitionarea in spatiu, **view** – transformarea in functie de pozitia cameri, **projection** – proiectia din 3D in 2D, **viewport** – asezarea in imaginea finala) apoi

calculeaza iluminarea pe suprafete (in modul legacy exista un numar fix de lumini ce pot fi folosite)

- asamblarea primitivelor – aici sunt taiate triunghiurile ce nu se incadreaza in spatiul vizibil si creati noi vertecsi pentru a le “inchide”
- rasterizare – procesul de transformare a primitivelor in pixeli
- texturare, culoare, ceata – alterarea culorii pixelilor in functie de textura aplicata, culoarea vertecsilor interpolata, setarile de “ceata”
- alpha test – se verifica transparenta pixelilor
- depth test si stencil test – anumiti pixeli pot fi refuzati daca nu indeplinesc conditiile de adancime sau daca e folosit un sablon
- color buffer blend – se mixeaza culoarea pixelului cu cel existent deja in imagine

Modul modern se bazeaza pe un proces complet programabil de catre utilizator. Astfel, utilizatorul librariei isi defineste singur rolul datelor pe care le foloseste si modul in care acestea sunt procesate.

Procesarea vertecsilor si a texturilor se face de catre procesorul grafic dedicat (GPU) prin programe speciale scrise de utilizator cunoscute sub denumirea de “shader”. Astfel programatorul poate procesa in orice mod doreste datele grafice, putand obtine astfel efecte mult mai complexe si o imagine finala foarte realista. Dezavantajul insa este complexitatea crescuta a intregului proces de dezvoltare.

Operatiile de transformare se intampla in programele shader care sunt executate de catre GPU. Acolo ajung matricile de transformare transmise din programul principal, precum si secventele de vertecsi, normalele, indecsii, coordonatele de textura, etc.

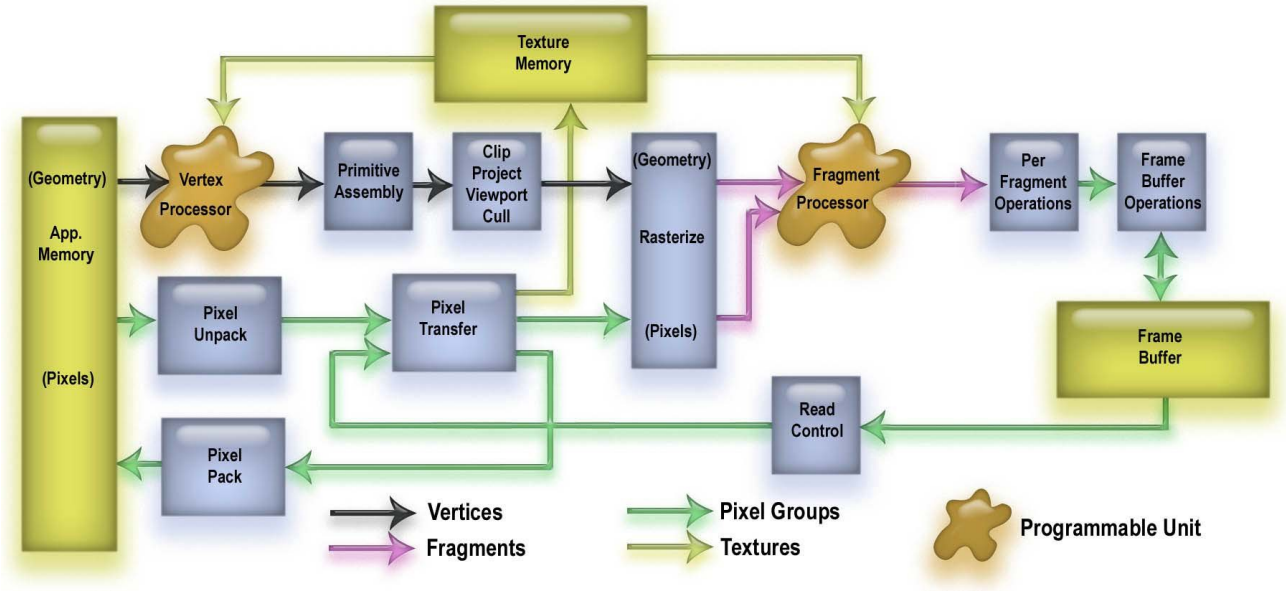


Diagrama modului programabil

Astfel anumiti pasi din proces sunt lasati la latitudinea programatorului, cum ar fi

- transformarea
- iluminarea (iluminarea poate fi facuta per pixel in loc de vertex pentru un efect mai realist)
- aplicarea texturilor si a cetii
- post-procesarea (poate avea mai multi pasi)

In versiunile mai noi, programatorul are si posibilitatea generarii dinamice a vertecsilor direct in procesorul grafic, prin programele de tip **geometry shader.**

In aplicatia demonstrativa am folosit versiunea de API OpenGL 4, cu programe shader destul de simple, neavand nevoie de efecte grafice complicate.

Pe pagina urmatoare sunt cateva exemple de astfel de programe folosite.

Programul shader pentru procesarea vertexilor cuburilor:

```
attribute vec3 vPos;
attribute vec3 vNormal;
attribute vec4 vColor;
attribute vec2 vUV1;

varying vec4 fColor;
varying vec2 fUV1;
varying vec3 fNormal;

uniform mat4 mPVW;

void main() {
    gl_Position = mPVW * vec4(vPos, 1);
    fNormal = (mPVW * vec4(vNormal, 0)).xyz;
    fColor = vColor;
    fUV1 = vUV1;
}
```

Programul shader pentru procesarea elementelor 2D (linii, dreptunghiuri etc):

```
attribute vec3 vPos;
attribute vec4 vColor;

varying vec4 fColor;

uniform mat4 mViewportInverse;

void main() {

    gl_Position = mViewportInverse * vec4(vPos, 1);
    fColor = vColor;
}
```

Programul shader pentru procesarea vertecsilor textului:

```
#version 120

// Input vertex data, different for all executions of this shader.
attribute vec3 vertexPosition_screenspace;
attribute vec2 vertexUV;
attribute vec4 vertexColor;

uniform vec2 viewportHalfSize;
uniform vec2 translation;

// Output data ; will be interpolated for each fragment.
varying vec2 UV;
varying vec4 color;

void main(){

        // Output position of the vertex, in clip space
        // map [0..vW][0..vH] to [-1..1][-1..1]
```

```
        vec2 vertexPosition_homoneneousspace = vertexPosition_screenspace.xy +
translation.xy - viewportHalfSize + 0.5f;
        vertexPosition_homoneneousspace.y *= -1;
        vertexPosition_homoneneousspace /= viewportHalfSize;
        gl_Position =
vec4(vertexPosition_homoneneousspace,vertexPosition_screenspace.z,1);

        // UV of the vertex. No special space for this one.
        UV = vertexUV;
        color = vertexColor;
}
```

Programul shader pentru procesarea pixelilor textului:

```
#version 120

// Interpolated values from the vertex shaders
varying vec2 UV;
varying vec4 color;

// Values that stay constant for the whole mesh.
uniform sampler2D myTextureSampler;

void main(){

        gl_FragColor = color * texture2D( myTextureSampler, UV );
}
```

Secventa de cod care controleaza randarea cuburilor si transmite parametrii la shader:

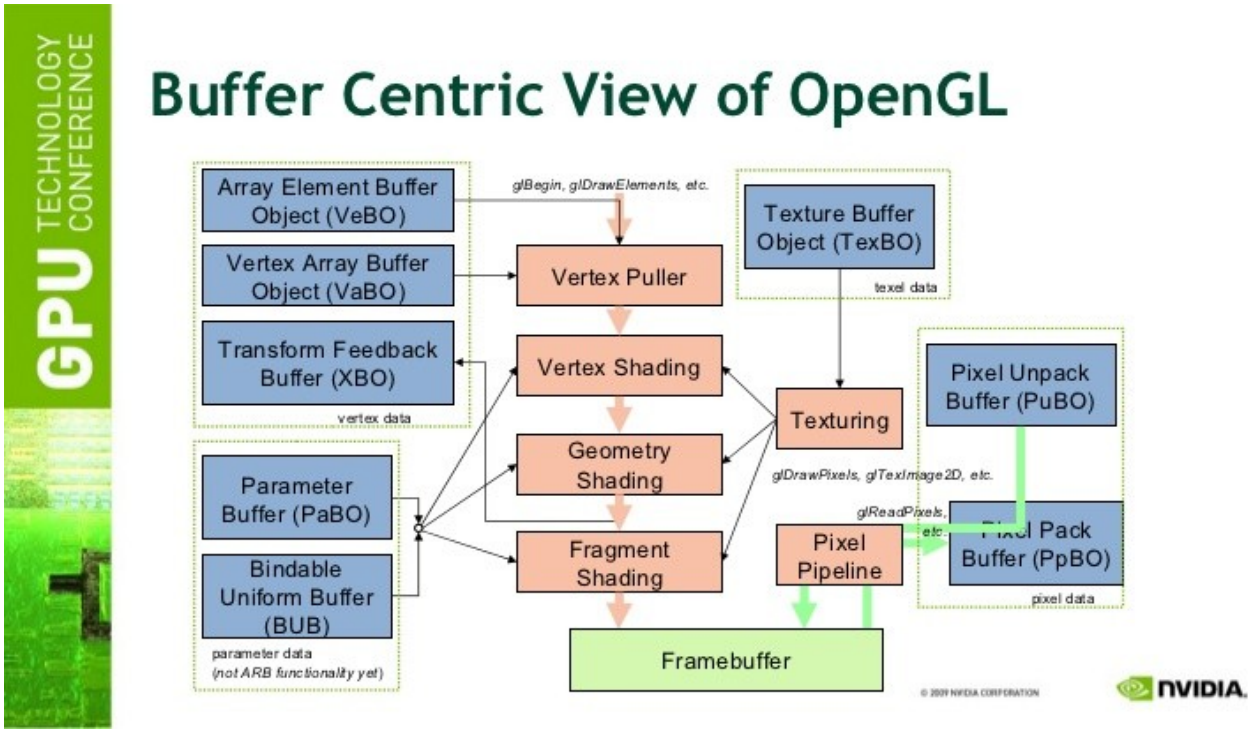
```
MeshRenderer::MeshRenderer(Renderer* renderer) {
        renderer->registerRenderable(this);
        meshShaderProgram_ = Shaders::createProgram("data/shaders/mesh.vert",
                                        "data/shaders/mesh-texture.frag");
        if (meshShaderProgram_ == 0) {
                throw std::runtime_error("Unable to load mesh shaders!!");
        }
        indexPos_ = glGetAttribLocation(meshShaderProgram_, "vPos");
        indexNorm_ = glGetAttribLocation(meshShaderProgram_, "vNormal");
        indexUV1_ = glGetAttribLocation(meshShaderProgram_, "vUV1");
        indexColor_ = glGetAttribLocation(meshShaderProgram_, "vColor");
        indexMatPVW_ = glGetUniformLocation(meshShaderProgram_, "mPVW");
}

void MeshRenderer::render(Viewport* vp) {
        glUseProgram(meshShaderProgram_);
        glEnableVertexAttribArray(indexPos_);
        glEnableVertexAttribArray(indexNorm_);
        glEnableVertexAttribArray(indexUV1_);
        glEnableVertexAttribArray(indexColor_);

        auto matPV = vp->camera()->matProjView();

        for (auto &m : renderQueue_) {
```

```
                if (m.pMesh_->isDirty()) {
                        m.pMesh_->commitChanges();
                }
                auto matPVW = matPV * m.wldTransform_;
                glUniformMatrix4fv(indexMatPVW_, 1, GL_FALSE,
                                                glm::value_ptr(matPVW));

                glBindBuffer(GL_ARRAY_BUFFER, m.pMesh_->getVertexBuffer());
                glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m.pMesh_
                                                ->getIndexBuffer());
                glVertexAttribPointer(indexPos_, 3, GL_FLOAT, GL_FALSE,
                        sizeof(Mesh::s_Vertex), (void*)offsetof(Mesh::s_Vertex,
                                                position));
                glVertexAttribPointer(indexNorm_, 3, GL_FLOAT, GL_FALSE,
                        sizeof(Mesh::s_Vertex), (void*)offsetof(Mesh::s_Vertex,
                                                normal));
                glVertexAttribPointer(indexUV1_, 2, GL_FLOAT, GL_FALSE,
                        sizeof(Mesh::s_Vertex), (void*)offsetof(Mesh::s_Vertex,
                                                UV1));
                glVertexAttribPointer(indexColor_, 4, GL_FLOAT, GL_FALSE,
                        sizeof(Mesh::s_Vertex), (void*)offsetof(Mesh::s_Vertex,
                                                color));

                glDrawElements(GL_TRIANGLES, m.pMesh_->getIndexCount(),
                        GL_UNSIGNED_SHORT, 0);
        }
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
```

Libraria OpenGL are un mod de functionare asincron, in sensul ca procesul de randare se desfasoara pe un procesor separat, dedicat, in paralel cu programul principal, ceea ce confera un plus de performanta.

Astfel, comenzile date de programator sunt inregistrate intr-o coada de unde sunt preluate apoi de catre procesorul grafic atunci cand este liber. In momentul in care se doreste afisarea imaginii, libraria asteapta automat pana cand procesorul grafic termina de executat toate operatiile.

Pentru optimizarea timpului de calcul, este recomandat ca toate datele si programele shader sa fie incarcate in avans in memoria speciala video de unde pot fi accesate cu rapiditate de procesorul grafic. Pentru acest scop exista obiecte speciale numite VBO (Vertex Buffer Object) in care pot fi incarcate datele de vertecsi, Texturi in care pot fi incarcate imaginile ce sunt aplicate pe primitive, etc:



Diferitele tipuri de bufferi in OpenGL

3.6 MODELUL DE DATE IN OPENGL

OpenGL foloseste anumite tipuri specifice de date pentru vertecsi, indecsi, matrici, texturi, etc.

Pentru programator este important sa cunoasca aceste tipuri pentru a putea trimite datele in formatul corect, altfel aplicatia nu poate functiona.

In geometria analitica aplicata in informatica, pentru precizie sporita, se foloseste de obicei tipul de date **double** stocat in memorie pe 64 biti. Acest tip confera precizie foarte mare in calcule, insa ocupand multa memorie este inadecvat pentru lucrul cu grafica de performanta.

De aceea OpenGL foloseste tipul **float** pe 32 biti. Acesta ofera suficienta precizie la o performanta superioara. In functie de tipul datelor transmise, fiecare element poate avea unul sau mai multe componente **float:**

- vertex: 3 componente **float**
- matrice: 4x4=16 elemente **float**
- quaternion: 4 elemente **float**
- coordonate in textura UV: 2 elemente **float**
- culoare RGB: 3 elemente float
- culoare RGBA: 4 elemente float
- normala: 3 elemente float

In programele shader aceste tipuri de date sunt reprezentate prin numele vec2, vec3, vec4, mat4.

Un alt aspect important de care trebuie tinut cont este ordinea in care sunt definite matricile (row-major sau column-major):

- row-major: elementele liniilor sunt consecutive in memorie
- column-major: elementele coloanelor sunt consecutive in memorie.

Inversarea modului de reprezentare are ca efect transpunerea matricii, ceea ce poate crea erori in program.

In modul legacy (fixed-pipeline) este folosita ordinea column-major, dar in modul modern programatorul poate alege oricare ordine, atat timp cat tine cont apoi de aceasta la ordinea de inmultire a matricilor intre ele si cu vectori.

Row-major order

$$[x \quad y \quad z] * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \qquad \begin{aligned} x' &= x*a + y*d + z*g \\ y' &= x*b + y*e + z*h \\ z' &= x*c + y*f + z*i \end{aligned}$$

Column-major order

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad \begin{aligned} x' &= a*x + d*y + g*z \\ y' &= b*x + e*y + h*z \\ z' &= c*x + f*y + i*z \end{aligned}$$

Pentru indecsi, OpenGL foloseste date de tip intreg, pe 8, 16, sau 32 biti, la alegerea programatorului.

Pentru texturi, in general este folosit tipul RGBA pe 32 biti, insa pentru economia memoriei, pot fi folosite texturi comprimate.

In aplicatia practica am folosit matrici in ordinea column-major si indecsi pe 16 biti.

```
std::vector<uint16_t> indices_;
```

3.7 LIBRARIA GLM

Aceasta librarie este independenta de OpenGL, insa ofera in C++ tipuri de date ce se muleaza perfect pe cele native din shader, cum ar fi vec3, vec4, mat4x4 etc.

Deoarece este necesar ca anumite calcule vectoriale sa fie facute in programul principal, apare nevoia unei librarii de acest gen, care sa ofere tipuri de date pentru vectori si matrici, precum si operatii pe vectori, matrici, si transformari.

GLM este o librarie header-only, adica compusa doar din fisiere **.h* in care sunt definite tipuri templatizate si operatiile aferente.

https://github.com/g-truc/glm

pagina librariei GLM

In aplicatia practica am folosit tipurile de date din GLM si operatiile pentru calcularea matricilor de transformare, calcul pe quaternioni si pe vectori, interpolare intre quaternioni si vectori, etc.

Calcularea inversei unei matrici:

```
auto mInv = glm::inverse(parent_->worldTransform());
```

Transformarea unui quaternion intr-o matrice de rotatie si compunerea cu un alt quaternion:

```
o = glm::quat_cast(mInv) * o;
```

Transformarea unui vector folosind o matrice:

```
pos = glm::vec3{mInv * glm::vec4{pos, 1}};
```

Inmultirea a doua matrici:

```
matWorldCache_ = parent_->worldTransform() * matFrameCache_;
```

Constructia unui cub folosind tipul de date glm::vec3:

```
void Mesh::createBox(glm::vec3 center, float width, float height, float depth) {
        float left = center.x - width * 0.5f;
        float right = center.x + width * 0.5f;
        float bottom = center.y - height * 0.5f;
        float top = center.y + height * 0.5f;
        float back = center.z - depth * 0.5f;
        float front = center.z + depth * 0.5f;
        glm::vec3 nBack(0, 0, -1);
        glm::vec3 nFront(0, 0, 1);
        glm::vec3 nLeft(-1, 0, 0);
        glm::vec3 nRight(1, 0, 0);
        glm::vec3 nTop(0, 1, 0);
        glm::vec3 nBottom(0, -1, 0);
        glm::vec3 white(1, 1, 1);
        vertices_.clear();
        vertices_.assign({
        // back face
                // #0 back bottom left
                {
                        {left, bottom, back}, nBack, {0, 0}, white
                },
                // #1 back top left
                {
                        {left, top, back}, nBack, {0, 1}, white
                },
                // #2 back top right
                {
                        {right, top, back}, nBack, {1, 1}, white
                },
                // #3 back bottom right
                {
                        {right, bottom, back}, nBack, {1, 0}, white
                },
        // top face
                // #4 back top left
                {
                        {left, top, back}, nTop, {0, 0}, white
                },
                // #5 front top left
                {
                        {left, top, front}, nTop, {0, 1}, white
                },
                // #6 front top right
                {
                        {right, top, front}, nTop, {1, 1}, white
                },
                // #7 back top right
                {
                        {right, top, back}, nTop, {1, 0}, white
                },
        // front face
                // #8 front top right
                {
                        {right, top, front}, nFront, {0, 0}, white
                },
                // #9 front top left
```

```
                {
                        {left, top, front}, nFront, {0, 1}, white
                },
                // #10 front bottom left
                {
                        {left, bottom, front}, nFront, {1, 1}, white
                },
                // #11 front bottom right
                {
                        {right, bottom, front}, nFront, {1, 0}, white
                },
        // bottom face
                // #12 front bottom left
                {
                        {left, bottom, front}, nBottom, {0, 0}, white
                },
                // #13 back bottom left
                {
                        {left, bottom, back}, nBottom, {0, 1}, white
                },
                // #14 back bottom right
                {
                        {right, bottom, back}, nBottom, {1, 1}, white
                },
                // #15 front bottom right
                {
                        {right, bottom, front}, nBottom, {1, 0}, white
                },
        // left face
                // #16 front bottom left
                {
                        {left, bottom, front}, nLeft, {0, 0}, white
                },
                // #17 front top left
                {
                        {left, top, front}, nLeft, {0, 1}, white
                },
                // #18 back top left
                {
                        {left, top, back}, nLeft, {1, 1}, white
                },
                // #19 back bottom left
                {
                        {left, bottom, back}, nLeft, {1, 0}, white
                },
        // right face
                // #20 back bottom right
                {
                        {right, bottom, front}, nRight, {0, 0}, white
                },
                // #21 back top right
                {
                        {right, top, front}, nRight, {0, 1}, white
                },
                // #22 front top right
                {
                        {right, top, back}, nRight, {1, 1}, white
                },
                // #23 front bottom right
```

```
                {
                        {right, bottom, back}, nRight, {1, 0}, white
                }
        });

        // assign colors
        glm::vec3 c[] { {1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {1, 1, 0} };
        for (uint i=0; i<vertices_.size(); i++) {
                vertices_[i].color = c[i % (sizeof(c) / sizeof(c[0]))];
        }

        // indices
        indices_.clear();
        indices_.assign({
                // back face
                0, 1, 2, 0, 2, 3,
                // top face
                4, 5, 6, 4, 6, 7,
                // front face
                8, 9, 10, 8, 10, 11,
                // bottom face
                12, 13, 14, 12, 14, 15,
                // left face
                16, 17, 18, 16, 18, 19,
                // right face
                20, 21, 22, 20, 22, 23
        });

        dirty_ = true;
}
```

Crearea matricilor de transformarea View si Projection folosind GLM:

```
void Camera::updateView() {
        matView_ = glm::lookAtLH(position_, position_ + direction_, {0, 1, 0});
}

void Camera::updateProj() {
        float zNear = 0.5f;
        float zFar = 50.f;
        if (fov_ == 0) {
                // set ortho
                matProj_ = glm::ortho(ortho_.x, ortho_.x + ortho_.z, ortho_.y,
                                        ortho_.y+ortho_.w, zNear, zFar);
        } else {
                // set perspective
                matProj_ = glm::perspectiveFovLH(fov_, (float)pViewport_
                        ->width(), (float)pViewport_->height(), zNear, zFar);
        }
}
```

3.8 APLICAREA TRANSFORMARILOR IN OPENGL

Aplicarea unei transformari geometrice in cadrul OpenGL se poate face in doua moduri:

1. pe CPU (in software) – vertecsii, normalele etc sunt transformate in cadrul programului principal, folosind operatii de genul celor puse la dispozitie de libraria GLM, apoi datele astfel pregatite sunt transmise catre OpenGL pentru a fi randate.
2. Pe GPU (in hardware) – vertecsii, normalele etc sunt transmise ca atare catre OpenGL, netransformate, impreuna cu matricile de transformare aferente. Transformarile vor fi executate de procesorul grafic pe baza acestor date, fie prin procesul fixed-pipeline, fie in programele shader scrise de programator

In general este preferata a doua varianta, pentru ca ofera flexibilitate mai mare si o performanta mult mai buna, deoarece procesorul grafic este optimizat pentru a efectua acest tip de operatii. Procesorul grafic are zeci sau sute de unitati de transformare care pot procesa tot atatia vertecsi in paralel, practic simultan. De asemenea se poate lucra usor cu ierarhii de obiecte in acest mod, deoarece singurele date care se modifica sunt matricile de transformare, iar vertecsii pot fi copiati de la inceput in memoria video in obiecte de tipul VBO si nu mai sunt modificati apoi, ceea ce duce la un castig de performanta suplimentar.

In aplicatii sunt folosite ambele metode impreuna, prima pentru anumite calcule (cum ar fi coliziuni, calcularea traiectoriilor etc), iar a doua pentru randare.

3.8.1 Transmiterea matricilor de transformare

Matricile de transformare din punctul de vedere al OpenGL sunt reprezentate printr-o secventa de 16 numere de tip float, ce sunt interpretate ca o matrice de 4 pe 4, row-major sau column-major (depinde de programator). In aplicatia practica, matricile sunt continute in obiecte de tip **glm::mat4**, care pe langa stocarea propriuzisa a numerelor ofera si operatii pe acestea.

In cadrul programelor shader, exista mai multe categorii de date:

- Uniform – date care isi pastreaza valoarea de la un vertex la altul
- Varying – date de iesire din vertex shader care sunt interpolate apoi de GPU si transmise catre fragment shader (de tipul culoare, pozitie, etc)

- Attribute – date de intrare in vertex shader care se schimba de la un vertex la altul (pozitia vertexului, normala, coordonatele de textura etc)

Matricile fac parte din prima categorie (Uniform). Astfel, pentru a transmite matricile catre OpenGL se folosesc urmatoarele functii:

void glUniformMatrix2fv(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

void glUniformMatrix3fv(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

void glUniformMatrix4fv(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

void glUniformMatrix2x3fv(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

void glUniformMatrix3x2fv(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

void glUniformMatrix2x4fv(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

void glUniformMatrix4x2fv(	GLint *location*,

	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

void glUniformMatrix3x4fv(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

void glUniformMatrix4x3fv(	GLint *location*,
	GLsizei *count*,
	GLboolean *transpose*,
	const GLfloat **value*);

Dintre acestea, cea mai des folosita este **glUniformMatrix4fv** care primeste ca parametru o matrice de dimensiune 4x4 cu element de tip float.

Documentatia pentru aceste metode se gaseste pe site-ul OpenGL la pagina aceasta:

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glUniform.xhtml

Primul parametru al acestei metode este indexul elementului uniform din shader pentru care se transmite valoarea, care poate fi obtinut apeland metoda

GLint glGetUniformLocation(	GLuint *program*,
	const GLchar **name*);

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glGetUniformLocation.xhtml

Primul parametru al acestei metode este programul shader, al doilea numele elementului Uniform din program.

In programul vertex shader matricea este declarata cu un nume ales de programator astfel:

```
uniform mat4 nume_matrice;
```

Exemplu de transmitere a matricilor din aplicatia practica:

```
indexMatPVW_ = glGetUniformLocation(meshShaderProgram_, "mPVW");
glUniformMatrix4fv(indexMatPVW_, 1, GL_FALSE, glm::value_ptr(matPVW));
```

3.8.2 Transmiterea vertecsilor catre OpenGL

Vertecsii pot fi transmisi direct in momentul in care se face randarea, sau pot fi copiati in memoria video intr-un obiect VBO (Vertex Buffrer Object) dinainte, metoda preferata pentru performanta superioara. In continuare vom prezenta aceasta metoda.

Crearea unui obiect VBO:

void glGenBuffers(	GLsizei *n*,
	GLuint * *buffers*);

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glGenBuffers.xhtml

Popularea VBO cu date presupune 2 pasi: declararea scopului bufferului si transferul propriuzis al datelor:

void glBindBuffer(	GLenum *target*,
	GLuint *buffer*);

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBindBuffer.xhtml

void glBufferData(	GLenum *target*,
	GLsizeiptr *size*,
	const GLvoid * *data*,
	GLenum *usage*);

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBufferData.xhtml

Exemplu din aplicatia practica de transmitere a vertecsilor si indecsilor in obiecte buffer:

```
void Mesh::commitChanges() {
        glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer_);
        glBufferData(GL_ARRAY_BUFFER, vertices_.size() * sizeof(vertices_[0]),
                                        vertices_.data(), GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        vertexCount_ = vertices_.size();

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer_);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices_.size() *
                        sizeof(indices_[0]), indices_.data(), GL_STATIC_DRAW);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
        indexCount_ = indices_.size();

        dirty_ = false;
}
```

Odata transmise datele, mai este necesar un pas, si anume crearea unei legaturi intre aceste date si atributele ce vor ajunge in programele shader, precum si specificarea formatului datelor:

In OpenGL exista 3 functii care indeplinesc aceasta sarcina:

void glVertexAttribPointer(	GLuint *index*,
	GLint *size*,
	GLenum *type*,
	GLboolean *normalized*,
	GLsizei *stride*,
	const GLvoid * *pointer*);

void glVertexAttribIPointer(	GLuint *index*,
	GLint *size*,
	GLenum *type*,
	GLsizei *stride*,
	const GLvoid * *pointer*);

void glVertexAttribLPointer(	GLuint *index*,
	GLint *size*,
	GLenum *type*,
	GLsizei *stride*,
	const GLvoid * *pointer*);

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttribPointer.xhtml

Exemplu din aplicatia practica:

```
glBindBuffer(GL_ARRAY_BUFFER, m.pMesh_->getVertexBuffer());
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m.pMesh_->getIndexBuffer());

glVertexAttribPointer(indexPos_, 3, GL_FLOAT, GL_FALSE, sizeof(Mesh::s_Vertex),
                                    (void*)offsetof(Mesh::s_Vertex, position));
glVertexAttribPointer(indexNorm_, 3, GL_FLOAT, GL_FALSE, sizeof(Mesh::s_Vertex),
                                    (void*)offsetof(Mesh::s_Vertex, normal));
glVertexAttribPointer(indexUV1_, 2, GL_FLOAT, GL_FALSE, sizeof(Mesh::s_Vertex),
                                    (void*)offsetof(Mesh::s_Vertex, UV1));
glVertexAttribPointer(indexColor_, 4, GL_FLOAT, GL_FALSE,sizeof(Mesh::s_Vertex),
                                    (void*)offsetof(Mesh::s_Vertex, color));
```

3.9 Transmiterea comenzilor de randare in OpenGL

Randarea reprezinta procesul prin care toate datele sunt procesate in vederea producerii imaginii finale. O data transmise datele, programatorul trebuie sa execute comenzi care sa comunice librariei OpenGL ce operatii se doresc a fi facute cu datele respective. Aceste comenzi pornesc efectiv procesul de randare in procesorul grafic.

Exista doua functii diferite pentru acest scop:

void glDrawArrays(	GLenum *mode*,
	GLint *first*,
	GLsizei *count*);

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawArrays.xhtml

Aceasta este pentru date neindexate (vertecsii sunt cititi si interpretati secvential).

void glDrawElements(	GLenum *mode*,
	GLsizei *count*,
	GLenum *type*,
	const GLvoid * *indices*);

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawElements.xhtml

Aceasta este pentru date indexate, in care ordinea vertecsilor este descrisa separat printr-o secventa de indecsi. Astfel un vertex poate fi refolosit specificand mai multi indecsi cu aceeasi valoare.

De obicei este preferata a doua metoda pentru ca nu necesita duplicarea vertecsilor in anumite situatii, si de asemenea este si mai performanta.

Exemplu de comenzi

```
glDrawElements(GL_TRIANGLES, m.pMesh_->getIndexCount(), GL_UNSIGNED_SHORT, 0);
```

pentru randarea cuburilor

```
for (unsigned i=0; i<viewportFiltersLine_.size(); i++) {
    if (viewportFiltersLine_[i].empty() || viewportFiltersLine_[i].find(
            vp->name()) != viewportFiltersLine_[i].end()) {
        glDrawElements(GL_LINES, 2, GL_UNSIGNED_SHORT, &indices_[i*2]);
    }
}
```

pentru randarea elementelor 2D

3.10 Ierarhii de obiecte

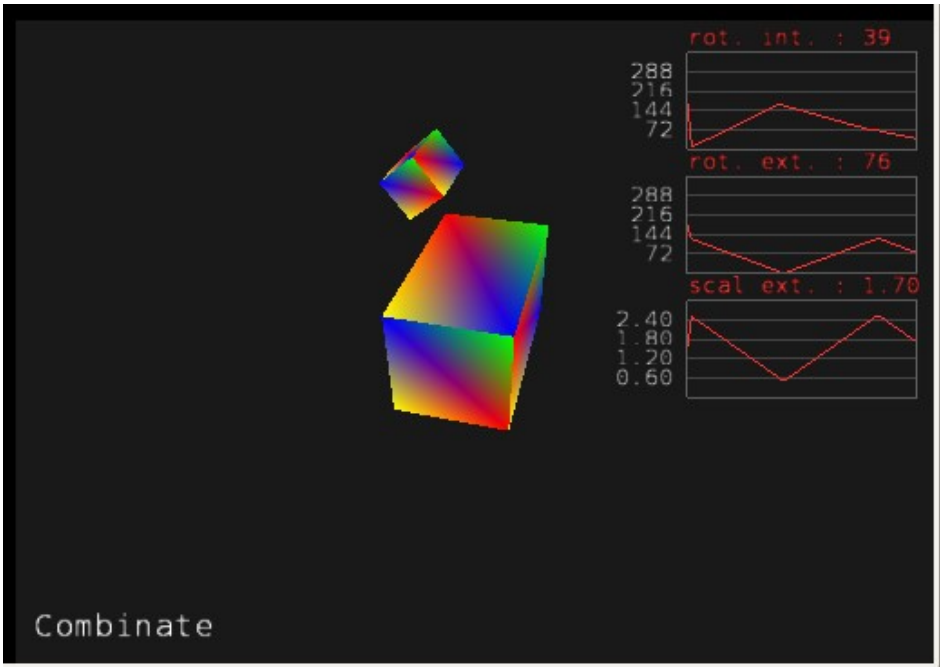
O ierarhie de obiecte reprezinta o colectie de obiecte ce se comporta la nivel macro ca un tot unitar, insa fiecare element component al sau are anumte grade de libertate, reprezentate prin transformari de translatie, rotatie sau scalare. Aceasta colectie are o structura arborescenta, cu un singur parinte la origine (radacina sau root).

Acest model este foarte des utilizat pentru ca permite crearea animatiilor complexe cu usurinta, programand doar transformarile relative dintre obiecte, ca apoi ele sa se compuna intr-o transformare in scena complexa.

Pentru a obtine acest efect, este necesar ca fiecare element al ierarhiei sa defineasca o transformare proprie, si sa i se atribuie o baza proprie. Transformarea fiecarui element este considerata relativa la baza parintelui elementului, care este si ea exprimata relativ la baza parintelui acestuia si asa mai departe pana la elementul radacina.

Pentru calcularea transformarii in spatiul scenei al fiecarui element, se inmultesc succesiv matricile elementului cu matricea parintelui, si asa mai departe, pana la elementul radacina. Transformarea astfel obtinuta este apoi aplicata doar pe vertecsii elementului initial.

Exemplu de ierarhie de obiecte din aplicatia practica:



Elementul cheie pentru realizarea acestei ierarhii il constituie metoda setParent din clasa DynamicBody:

```
auto box4 = std::make_unique<Box>(1, 1, 1);
auto box5 = std::make_unique<Box>(0.5f, 0.5f, 0.5f);
box5->body()->setParent(box4->body());  // attach box5 to box4
```

```
void DynamicBody::setParent(DynamicBody* parent) {
        if (parent_) {
                // first remove from old parent
                parent_->removeChild(this);
        }
        parent_ = parent;
        if (parent_)
                parent_->children_.insert(this);
        wldTransformDirty_ = true;
}
```

Calcularea matricilor de transformare in functie de ierarhie:

```
glm::mat4 DynamicBody::worldTransform() const {
        if (wldTransformDirty_) {
                if (frameTransformDirty_)
                        computeFrameTransform();
                if (parent_)
                        matWorldCache_ = parent_->worldTransform() *
                                                        matFrameCache_;
                else
                        matWorldCache_ = matFrameCache_;
                wldTransformDirty_ = false;
        }
        return matWorldCache_;
}
```

Se observa inmultirea matricii locale cu matricea parintelui.

3.11 Detalii tehnice

In acest capitol vom descrie librariile folosite pentru realizarea aplicatiei precum si pasii necesari pentru a putea compila si rula aplicatia.

Aplicatia este scrisa in limbajul **C++14** folosind doar librarii **cross-platform**, ceea ce inseamna ca poate fi compilata pentru orice sistem de operare: Linux, Windows, MacOS, etc

Aplicatia a fost dezvoltata pe **Linux**, folosind compilatorul **g++ 4.9.2** si mediul de dezvoltare **Eclipse** si foloseste urmatoarele librarii:

- GLEW http://glew.sourceforge.net/
  - o librarie ce ajuta lucrul cu OpenGL detectand automat ce extensii exista pe sistem
- GLFW 3 http://www.glfw.org/
  - o librarie ce abstractizeaza mediul vizual, pune la dispozitie functii de creare si manipulare a ferestrelor si de detectare a inputului (mouse/tastatura)
- GL 4 (pe linux este in pachetul mesa-devel)
  - libraria de OpenGL
- libpng http://www.libpng.org/pub/png/libpng.html
  - pentru incarcarea texturilor

Sursele aplicatiei sunt disponibile pe GitHub la aceasta adresa: https://github.com/bog2k3/licenta

In continuare vom descrie pasii necesari pentru compilarea aplicatiei pe **Linux,** cu referire la distributia Debian (pentru comenzi)

1. Instalare git
   - `sudo apt-get install git`
2. Clonare repository
   - `mkdir app`
   - `git clone git@github.com/bog2k3/licenta.git`
3. instalare compilatoare
   - `sudo apt-get install make`

- `sudo apt-get install gcc`

4. instalare librarii

   - pentru GLEW se va descarca de pe site-ul indicat
   - pentru GLFW se va descarca de pe site-ul indicat
   - libraria GLM poate fi instalata din repository, dar este o versiune veche
     - `sudo apt-get install glm`
     - este de preferat sa se instaleze de pe site (https://github.com/g-truc/glm)

   - `sudo apt-get install mesa-devel`
   - `sudo apt-get install X11-devel`
   - `sudo apt-get install libpng-devel`

5. compilarea aplicatiei

   - `cd licenta`
   - `./build-r.sh`

Daca compilarea se termina cu succes, aplicatia poate fi rulata cu comanda:

```
Release/licenta
```

Se poate folosi si mediul Eclipse pentru a deschide proiectul si a-l compila.

IV. CONCLUZII

OpenGL ofera functii de nivel jos pentru accesarea capabilitatilor de randare 3D ale chipsetului dedicat. Modul sau de lucru este foarte apropiat de modelul matematic ceea ce il face un instrument foarte bun pentru mediul academic.

In aceasta lucrare am abordat intr-o forma prescurtata subiecte foarte vaste din lumea programarii grafice, insa in literatura de specialitate se pot gasi atat explicatii mult mai ample cat si numeroase exemple pentru majoritatea situatiilor intalnite in practica.

Transformarile grafice reprezinta un element care sta la baza tuturor Engine-urilor grafice existente precum si a oricarei aplicatii ce doreste sa foloseasca animatii sau pozitionari in scena ale obiectelor.

Exista mai multe moduri de abordare a implementarii acestor transformari, precum am mentionat de-a lungul lucrarii, iar fiecare Engine existent foloseste o variatiune a acestora.

Cele mai dezvoltate domenii in care sunt folosite intens transformarile sunt aplicatiile ingineresti de proiectare (ex AutoCAD) si jocurile video in care grafica 3D este elementul principal. In jocuri se folosesc multiple tehnici de implementare a transformarilor, cum ar fi interpolarile pe curbe sau ierarhiile complexe de obiecte, un caz aparte fiind reprezentat de personajele din jocuri care folosesc o tehnica de nimatie numita “skinning”. Aceasta tehnica reprezinta aplicarea de transformari pe un sistem de oase invizibil de care sunt legati apoi vertecsii modelului, iar transformarile sunt mixate si aplicate pe acestia.

Transformarile sunt folosite intens nu numai la miscarea obiectelor in scena, ci si la vizualizarea scenelor statice, unde transformarile sunt aplicate pe camera virtuala, care este deplasata si rotita prin scena pentru a oferi vederi din diferite unghiuri.

De asemenea o ramura la fel de importanta o reprezinta transformarile in doua dimensiuni (2D) care sunt aplicate in general in jocuri video sau in aplicatii de procesare de imagini sau video pentru manipularea texturilor si a imaginilor. Astfel pe un obiect static poate fi obtinut un efect de animatie aplicand o transformare progresiva pe textura obiectului.

V. BIBLIOGRAFIE

1. **Documentatia oficiala OpenGL** https://www.khronos.org/registry/OpenGL-Refpages/gl4/
2. **Geometric Transformations for 3D Modeling (***Michael E Mortenson, Digital CD, Published: September, 2005, ISBN: 9780831133481***)** http://new.industrialpress.com/geometric-transformations-for-3d-modeling-ebook-on-cd.html
3. **Euclidean Geometry and Transformations** (*Clayton W. Dodge ISBN 0486434761*) http://store.doverpublications.com/0486434761.html
4. **Linear Algebra, Geometry and Transformation** (*Chapman and Hall/CRC 2014*) https://www.crcpress.com/Linear-Algebra-Geometry-and-Transformation/Solomon/p/book/9781482299281
5. **Geometric transformations and objects** (*Manikandan Bakthavatchalam, Franccois Chaumette, Eric Marchand, Filip Novotny, Antony Saunier, Fabien Spindler, Romain Tallonneau*) http://visp-doc.inria.fr/manual/visp-tutorial-geometric-objects.pdf
6. **Euclidean and Affine Transformations 1st Edition** (*P. S. Modenov A. S. Parkhomenko, ISBN: 9781483261485*) https://www.elsevier.com/books/euclidean-and-affine-transformations/modenov/978-1-4832-2801-3
7. **OpenGL Transformations** (*Ed Angel Professor of Computer Science, Electrical and Computer Engineering, and Media Arts University of New Mexico*) https://www.cs.unm.edu/~angel/CS433.03/CS433Lecture13.pdf
8. **Basics of Geometric Transformations** (*CS 650: Computer Vision Bryan S. Morse*) https://canvas.instructure.com/courses/743674/files/20066753/download?verifier=wXPOasSEeGl4D8btWrSRv7n1dp6AY5aiAN5ELpwe
9. **Geometric Transformations: Warping, Registration, Morphing** (*Yao Wang Polytechnic University, Brooklyn, NY 11201*) http://eeweb.poly.edu/~yao/EL5123/lecture12_ImageWarping.pdf
10. Basic Geometric Transformations in 2D and 3D (*Jorge Alberto Márquez Flores, PhD Image Analysis and Visualization Laboratory, CCADET-UNAM, 2012*)

http://www.academicos.ccadet.unam.mx/jorge.marquez/cursos/imagenes_neurobiomed/Basic%20transformations%20in%203D.pdf

11. **Geometrical Primitives, Transformations and Image Formation** (*EECS 598-08 Fall 2014 Foundations of Computer Vision*) https://web.eecs.umich.edu/~jjcorso/t/598F14/files/lecture_0915_cameras.pdf
12. **Introduction to Computer Graphics** (Helena Wong, 2000) http://www.cs.cityu.edu.hk/~helena/cs31622000A/Notes03.pdf
13. **Google**

https://www.google.com/

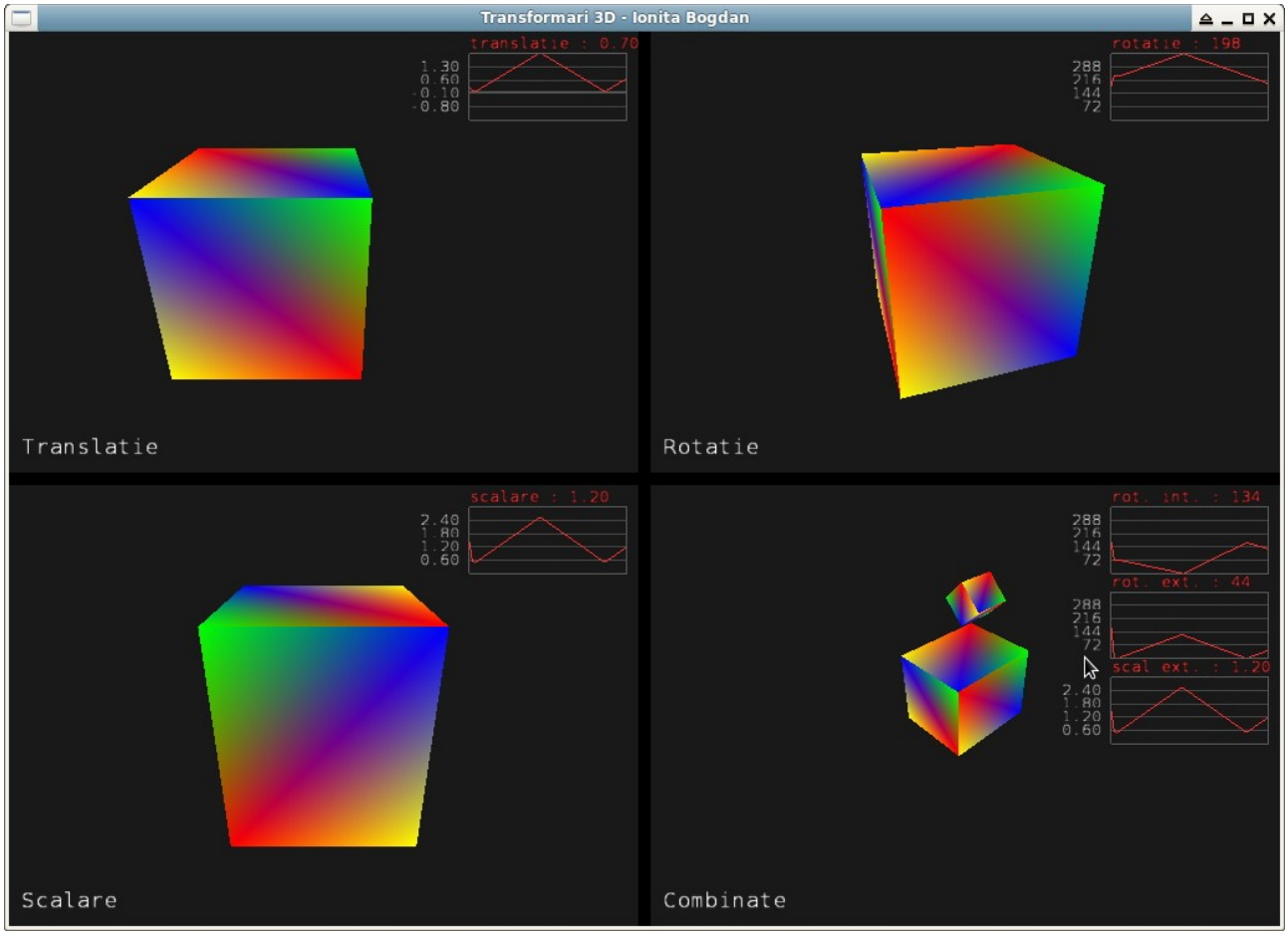
VI. ANEXE

6.1 Interpolarea parametrilor de transformare:

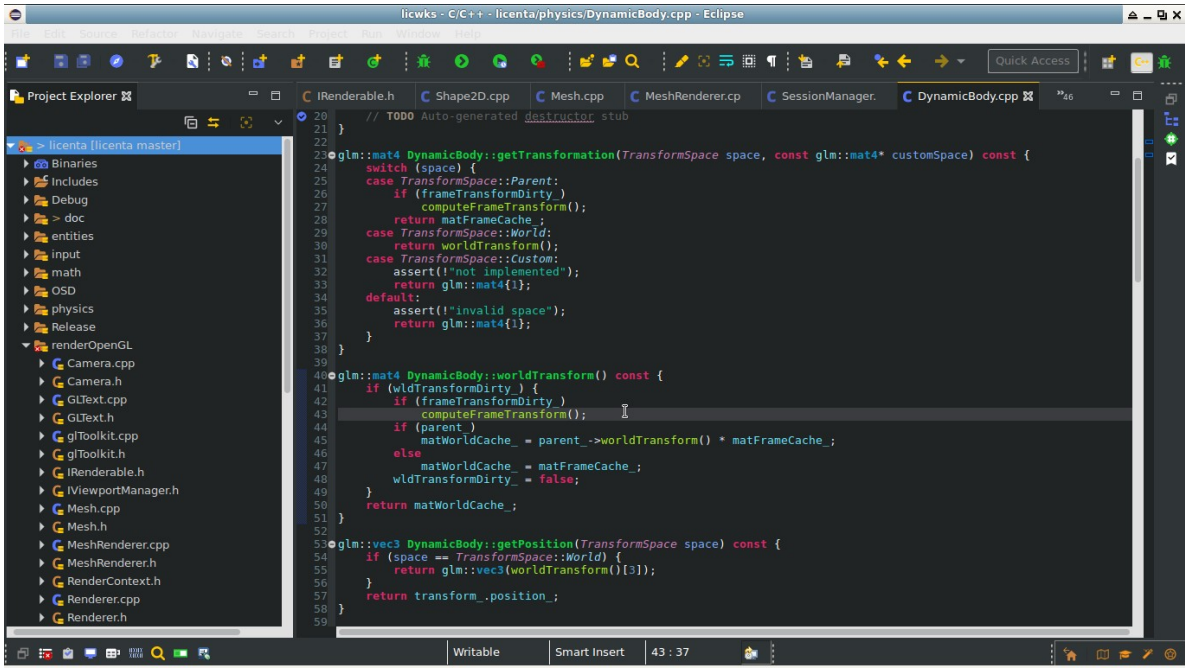
```
template<class NodeValue>
NodeValue genericLerp(NodeValue const& n1, NodeValue const& n2, float factor) {
	return n1 * (1-factor) + n2 * factor;
}


template<class NodeValue>
float genericDistance(NodeValue const& n1, NodeValue const& n2) {
	return abs<float>(n2 - n1);
}

template<class NodeValue,
	class LerpFunction,
	class DistanceFunction>
void PathLerper<NodeValue, LerpFunction, DistanceFunction>::update(float dt) {
	if (path_.empty())
		return;

	if (lerpFactor_ >= 1) {
		// reached the next vertex
		if (pathIndex_ > 0 && (unsigned)pathIndex_ == path_.size()) {
			path_.clear(); // finished the path
			return;
		}

		jumpNext_ = path_[pathIndex_].type == PathNodeType::jump;
		if (path_[pathIndex_].type == PathNodeType::jump ||
			path_[pathIndex_].type == PathNodeType::redirect) {
			pathIndex_ = clamp(path_[pathIndex_].targetIndex, 0u,
					(unsigned)path_.size() - 1);
			return;
		}
		if (jumpNext_) {
			jumpNext_ = false;
			pathIndex_++;
			return;
		}

		lerpFactor_ = 0;
		auto &next = path_[pathIndex_];
		origin_ = last_;
		segmentLength_ = distFn_(origin_, next.value);
		maxLerpSpeed_ = cruiseSpeed_ / segmentLength_;
	}

	if (lerpFactor_ > 0.9f) { // nearing the end
		// slow down
		float delta = maxLerpSpeed_ * 0.5f * dt;
		if (lerpSpeed_ > delta)
			lerpSpeed_ -= delta;
	} else if (lerpSpeed_ < maxLerpSpeed_) {
		// speed up
		lerpSpeed_ += maxLerpSpeed_ * 0.5f * dt;
```

```
        }
        lerpFactor_ += dt; //lerpSpeed_ * dt;
        last_ = lerpFn_(origin_, path_[pathIndex_].value, clamp(lerpFactor_,
                                                                0.f, 1.f));
        if (lerpFactor_ >= 1) {
                pathIndex_++;
        }
}
```

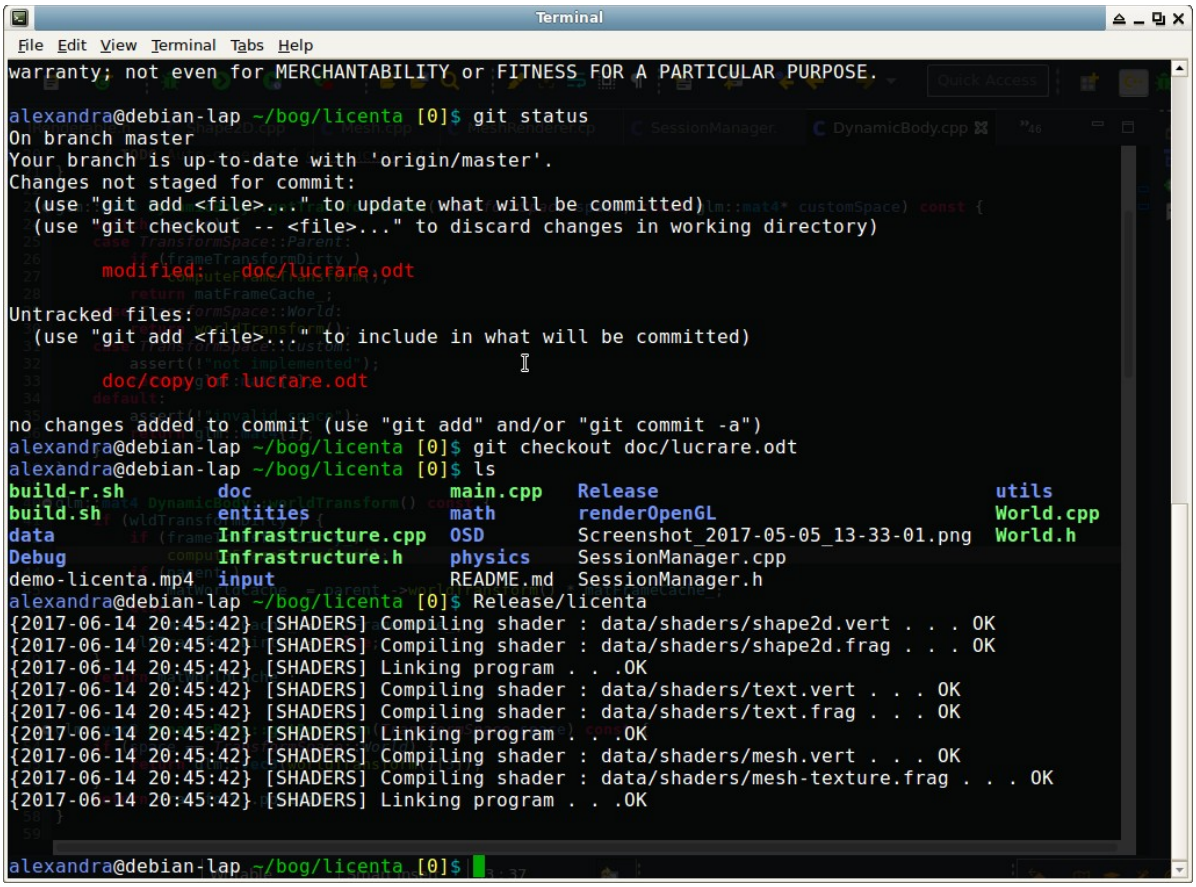
6.2 Screenshot ansamblu aplicatie



6.3 Screenshot mediu de dezvoltare



Exemplu de rulare aplicatie:



6.4 Secventa de cod care genereaza si configureaza toate elementele din scena

```
void SessionManager::createTransformSession() {
        // set up viewports
        int hw = vpm_->windowWidth()/2;
        int hh = vpm_->windowHeight()/2;
        // viewport 1
        auto vp = std::make_unique<Viewport>(0, hh+5, hw-5, hh-5);
        vp->camera()->moveTo({-1, 1, 0});
        vp->camera()->lookAt({-2, 0.5f, 0});
        vp->setBkColor({0.1f, 0.1f, 0.1f});
        vpm_->addViewport("translation", std::move(vp));
        // viewport 2
        vp = std::make_unique<Viewport>(hw+5, hh+5, hw-5, hh-5);
        vp->camera()->moveTo({0, 1, 1});
        vp->camera()->lookAt({0, 0.5f, 2});
        vp->setBkColor({0.1f, 0.1f, 0.1f});
        vpm_->addViewport("rotation", std::move(vp));
        // viewport 3
        vp = std::make_unique<Viewport>(0, 0, hw-5, hh-5);
        vp->camera()->moveTo({1, 1, 0});
        vp->camera()->lookAt({2, 0.5f, 0});
        vp->setBkColor({0.1f, 0.1f, 0.1f});
        vpm_->addViewport("scale", std::move(vp));
        // viewport 4
        vp = std::make_unique<Viewport>(hw+5, 0, hw-5, hh-5);
        vp->camera()->moveTo({0, 3, -4});
        vp->camera()->lookAt({0, 0.5f, -7});
        vp->setBkColor({0.1f, 0.1f, 0.1f});
        vpm_->addViewport("combined", std::move(vp));

        // create entities
        auto box1 = std::make_unique<Box>(1, 1, 1);
        auto pc = std::make_unique<PathController>(box1->body());
        wld_->takeOwnershipOf(std::move(box1));
        pc->addVertex({{-3, 0, -1}, glm::fquat()});
        pc->addVertex({{-3, 0, +1}, glm::fquat()});
        pc->addRedirect(0);
        pc->start(1.5f);
        auto sv1 = std::make_unique<SigViewerEntity>(
                        ViewportCoord(27, 5, ViewportCoord::percent,
                        ViewportCoord::top | ViewportCoord::right), 1.f,
                        ViewportCoord(25, 15, ViewportCoord::percent),
                        std::set<std::string>{"translation"});
        auto pcp1 = pc.get();
        wld_->takeOwnershipOf(std::move(pc));
        sv1->get().addSignal("translatie",
                        [pcp1]() -> float { return (pcp1->value().position -
                        pcp1->vertex(0).position).z; },
                        glm::vec3(1.f, 0.2f, 0.2f), 0.05f, 50, 1.5f, -1.5f, 2);
        wld_->takeOwnershipOf(std::move(sv1));

        auto box2 = std::make_unique<Box>(1, 1, 1);
        pc = std::make_unique<PathController>(box2->body());
        wld_->takeOwnershipOf(std::move(box2));
        pc->addVertex({glm::vec3{0.f, 0.f, 3.f}, glm::angleAxis(0.f,
                        glm::vec3{0, 1, 0})});
        pc->addVertex({glm::vec3{0.f, 0.f, 3.f}, glm::angleAxis(PI/1.5f,
                        glm::vec3{0, 1, 0})});
```

```
        pc->addVertex({glm::vec3{0.f, 0.f, 3.f}, glm::angleAxis(2*PI/1.5f,
                        glm::vec3{0, 1, 0})});
        pc->addRedirect(0);
        pc->start(2.5f);
        auto sv2 = std::make_unique<SigViewerEntity>(
                        ViewportCoord(27, 5, ViewportCoord::percent,
                        ViewportCoord::top | ViewportCoord::right), 1.f,
                        ViewportCoord(25, 15, ViewportCoord::percent),
                        std::set<std::string>{"rotation"});
        auto pcp2 = pc.get();
        wld_->takeOwnershipOf(std::move(pc));
        sv2->get().addSignal("rotatie",
                        [pcp2]() -> float { return (pcp2->value() -
                        pcp2->vertex(0)) * 180 / PI; },
                        glm::vec3(1.f, 0.2f, 0.2f), 0.05f, 50, 360, 0, 0);
        wld_->takeOwnershipOf(std::move(sv2));

        auto box3 = std::make_unique<Box>(1, 1, 1);
        pc = std::make_unique<PathController>(box3->body());
        wld_->takeOwnershipOf(std::move(box3));
        pc->addVertex({glm::vec3{3, 0, 0}, glm::fquat{},
                        glm::vec3{1, 0.5f, 1}});
        pc->addVertex({glm::vec3{3, 0, 0}, glm::fquat{},
                        glm::vec3{1, 2.5f, 1}});
        pc->addRedirect(0);
        pc->start(2.2f);
        auto sv3 = std::make_unique<SigViewerEntity>(
                        ViewportCoord(27, 5, ViewportCoord::percent,
                        ViewportCoord::top | ViewportCoord::right), 1.f,
                        ViewportCoord(25, 15, ViewportCoord::percent),
                        std::set<std::string>{"scale"});
        auto pcp3 = pc.get();
        wld_->takeOwnershipOf(std::move(pc));
        sv3->get().addSignal("scalare",
                        [pcp3]() -> float { return pcp3->value().scale.y; },
                        glm::vec3(1.f, 0.2f, 0.2f), 0.05f, 50, 3, 0, 2);
        wld_->takeOwnershipOf(std::move(sv3));

        auto box4 = std::make_unique<Box>(1, 1, 1);
        auto box5 = std::make_unique<Box>(0.5f, 0.5f, 0.5f);
        box5->body()->setParent(box4->body());  // attach box5 to box4

        pc = std::make_unique<PathController>(box5->body());
        wld_->takeOwnershipOf(std::move(box5));
        pc->addVertex({glm::vec3{-1.f, 0.5f, -1.f}, glm::fquat{}});
        pc->addVertex({glm::vec3{-1.f, 0.5f, -1.f}, glm::angleAxis(-160.f,
                        glm::vec3{1, 0, 0})});
        pc->addVertex({glm::vec3{-1.f, 0.5f, -1.f}, glm::angleAxis(-240.f,
                        glm::vec3{1, 0, 0})});
        pc->addRedirect(0);
        pc->start(2.4f);
        auto sv4 = std::make_unique<SigViewerEntity>(
                        ViewportCoord(27, 5, ViewportCoord::percent,
                        ViewportCoord::top | ViewportCoord::right), 1.f,
                        ViewportCoord(25, 15, ViewportCoord::percent),
                        std::set<std::string>{"combined"});
        auto pcp5 = pc.get();
        wld_->takeOwnershipOf(std::move(pc));
        sv4->get().addSignal("rot. int.",
```

```
                        [pcp5]() -> float { return (pcp5->value() -
                        pcp5->vertex(0)) * 180 / PI; },
                        glm::vec3(1.f, 0.2f, 0.2f), 0.05f, 50, 360, 0, 0);

        pc = std::make_unique<PathController>(box4->body());
        wld_->takeOwnershipOf(std::move(box4));
        pc->addVertex({glm::vec3{0, 0, -8.f}, glm::fquat{},
                        glm::vec3{0.5f, 1, 1}});
        pc->addVertex({glm::vec3{0, 0, -8.f}, glm::angleAxis(180.f,
                        glm::vec3{0, 1, 0}), glm::vec3{2.5f, 1, 1}});
        pc->addRedirect(0);
        pc->start(1.8f);
        auto pcp4 = pc.get();
        wld_->takeOwnershipOf(std::move(pc));
        sv4->get().addSignal("rot. ext.",
                        [pcp4]() -> float { return (pcp4->value() -
                        pcp4->vertex(0)) * 180 / PI; },
                        glm::vec3(1.f, 0.2f, 0.2f), 0.05f, 50, 360, 0, 0);
        sv4->get().addSignal("scal ext.",
                        [pcp4]() -> float { return pcp4->value().scale.x; },
                        glm::vec3(1.f, 0.2f, 0.2f), 0.05f, 50, 3, 0, 2);
        wld_->takeOwnershipOf(std::move(sv4));

        wld_->takeOwnershipOf(std::make_unique<LabelEntity>("Translatie", 22,
                        ViewportCoord{10, 10, ViewportCoord::absolute,
                        ViewportCoord::left | ViewportCoord::bottom}, 0,
                        glm::vec3{1, 1, 1},
                        std::set<std::string>{"translation"}));
        wld_->takeOwnershipOf(std::make_unique<LabelEntity>("Rotatie", 22,
                        ViewportCoord{10, 10, ViewportCoord::absolute,
                        ViewportCoord::left | ViewportCoord::bottom}, 0,
                        glm::vec3{1, 1, 1}, std::set<std::string>{"rotation"}));
        wld_->takeOwnershipOf(std::make_unique<LabelEntity>("Scalare", 22,
                        ViewportCoord{10, 10, ViewportCoord::absolute,
                        ViewportCoord::left | ViewportCoord::bottom}, 0,
                        glm::vec3{1, 1, 1}, std::set<std::string>{"scale"}));
        wld_->takeOwnershipOf(std::make_unique<LabelEntity>("Combinate", 22,
                        ViewportCoord{10, 10, ViewportCoord::absolute,
                        ViewportCoord::left | ViewportCoord::bottom}, 0,
                        glm::vec3{1, 1, 1}, std::set<std::string>{"combined"}));
}
```