

UNIVERSITATEA “TITU MAIORESCU” DIN BUCUREȘTI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

COORDONATOR ȘTIINȚIFIC:

Conf.univ.dr.ing. Mironela Pîrnău

ABSOLVENT:

Bogdan Florin Ioniță

SESIUNEA IUNIE-IULIE

2017

UNIVERSITATEA “TITU MAIORESCU” DIN BUCUREȘTI
FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ
Implementarea transformărilor grafice
folosind OpenGL

COORDONATOR ȘTIINȚIFIC:

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

| | |
|--|----|
| REFERAT..... | 3 |
| I. CUPRINS..... | 4 |
| II. INTRODUCERE..... | 5 |
| III. CONTINUTUL LUCRARII..... | 6 |
| 3.1 Definitii..... | 6 |
| 3.2 Proprietati ale spatiilor vectoriale..... | 7 |
| 3.3 Baze si coordonate..... | 8 |
| 3.4 Transformari..... | 10 |
| 3.4.1 Notiuni specifice in aplicatiile grafice..... | 11 |
| 3.4.2 Tipul notatiei si ordinea inmultirii..... | 12 |
| 3.4.2 Translatia..... | 13 |
| 3.4.3 Rotatia..... | 16 |
| 3.4.4 Scalarea..... | 19 |
| 3.4.5 Combinarea transformarilor..... | 22 |
| 3.5 Modul de functionare al OpenGL..... | 23 |
| 3.5.1 Modul legacy (v1)..... | 23 |
| 3.5.2 Modul modern (v2+)..... | 24 |
| 3.5.3 Asincronicitate..... | 28 |
| 3.5.4 Obiecte Buffer..... | 28 |
| 3.6 Modelul de date in OpenGL..... | 30 |
| 3.7 Libraria GLM..... | 32 |
| 3.8 Aplicarea transformarilor in OpenGL..... | 36 |
| 3.8.1 Transmiterea matricilor de transformare..... | 36 |
| 3.8.2 Transmiterea vertecsilor catre OpenGL..... | 39 |
| 3.9 Transmiterea comenzilor de randare in OpenGL..... | 41 |
| 3.10 Ierarhii de obiecte..... | 42 |
| 3.11 Detalii tehnice..... | 44 |
| IV. CONCLUZII..... | 46 |
| V. BIBLIOGRAFIE..... | 47 |
| VI. ANEXE..... | 49 |
| 6.1 Interpolarea parametrilor de transformare..... | 49 |
| 6.2 Screenshot ansamblu aplicatie..... | 50 |
| 6.3 Screenshot mediu de dezvoltare..... | 51 |
| 6.4 Secventa de cod care genereaza si configureaza toate elementele din scena..... | 52 |

II. INTRODUCERE

OpenGL este o librărie de nivel jos cu ajutorul căreia pot fi create aplicații cu conținut grafic bogat, atât 2D cât și 3D, folosind capabilitățile de randare grafică accelerată ale chipsetului dedicat, dacă acesta este prezent.

OpenGL a fost creat inițial ca o librărie pentru limbajul C, dar ulterior au fost dezvoltate interfețe 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 reflectă mai îndeaproape arhitectura și modul de funcționare al hardware-ului, având o structură relativă simplă, dar care oferă toate funcțiile necesare și o performanță de top.

Structurile de date și operațiile aplicate pe acestea în lucrul cu OpenGL se mulează perfect pe echivalentele lor matematice, astfel avem de a face cu spații vectoriale, vectori, operații pe vectori (produs scalar / produs vectorial), baze (un set de trei axe ortogonale între ele) și transformări de la o bază la alta (reprezentate prin matrici).

Transformările în teoria spațiilor vectoriale reprezintă operații de traducere a unor vectori exprimați într-o anumită bază în vectori echivalenți, dar exprimați în cadrul unei baze diferite, atât prin poziționare cât și prin orientare sau dimensiune relativă.

Lucrarea de față își propune să exploreze acest univers matematic al spațiilor vectoriale și al transformărilor, precum și aplicabilitatea lor practică în programarea grafică 3D, folosind limbajul C++.

III. CONTINUTUL LUCRĂRII

3.1 Definiții

În algebra, un spațiu vectorial reprezintă o mulțime de obiecte numite vectori, pe care sunt definite următoarele operații: adunarea a doi vectori, înmulțirea unui vector cu un scalar (număr real de obicei), produsul scalar a doi vectori, produsul vectorial a doi vectori. Există și spații vectoriale ale căror scalari pot fi numere complexe sau numere rationale. Operațiile de adunare a vectorilor și înmulțire cu un scalar trebuie să satisfacă niste cerințe, numite axiome, menționate mai jos.

Un spațiu vectorial este caracterizat prin dimensiunea sa, care, în termeni generali, reprezintă numărul de direcții independente din acel spațiu. Un element (vector) dintr-un spațiu n -dimensional va avea n componente – fiecare componentă reprezintă o coordonată pe axa corespunzătoare din spațiu.

Din punct de vedere istoric, primele idei referitoare la spații vectoriale pot fi regăsite în secolul al 17-lea în geometria analitică, sisteme de ecuații liniare și vectori euclideni. În forma actuală, mai abstractă, formulată de Giuseppe Peano în 1888, spațiile vectoriale pot fi construite dintr-o paletă mai largă de obiecte decât vectori euclideni, cum ar fi funcții și alte obiecte matematice, dar în mare teorie poate fi privită ca o extensie a geometriei clasice în care apar drepte, planuri și analogele lor în mai multe dimensiuni.

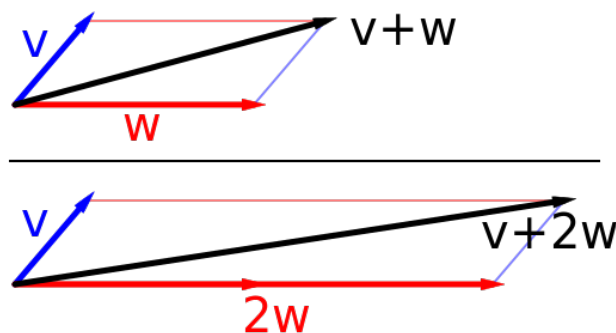
Axiomele spațiilor vectoriale:

- Asociativitatea adunării:
 - $u + (v + w) = (u + v) + w$
- Comutativitatea adunării:
 - $u + v = v + u$
- Element neutru la adunare:
 - există $0 \in V$ astfel încât $v + 0 = v$, oricare ar fi $v \in V$
- Element opus la adunare:

- pentru orice $v \in V$ exista un element $-v \in V$ 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:
 - $1v = v$
- 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 lucrării 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 spațiul vectorial $V^3(\mathbf{R})$, cu elemente de forma

$$\mathbf{v} = (v_1, v_2, v_3), \text{ unde } v_1, v_2, v_3 \in \mathbf{R}.$$

Elementul $\mathbf{v}_0(0, 0, 0)$ se numește “originea” spațiului V .

Punctele $\mathbf{v}_1(1, 0, 0)$, $\mathbf{v}_2(0, 1, 0)$, $\mathbf{v}_3(0, 0, 1)$ împreună cu \mathbf{v}_0 definesc 3 semi-drepte ortogonale între ele, orientate de la origine către \mathbf{v}_1 , \mathbf{v}_2 , respectiv \mathbf{v}_3 . Deoarece sunt ortogonale între ele, aceste semidrepte definesc 3 direcții liniar-independente, formând deci baza canonică a spațiului V .

Vectorii $\mathbf{i} = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{j} = \mathbf{v}_2 - \mathbf{v}_0$, $\mathbf{k} = \mathbf{v}_3 - \mathbf{v}_0$, componentele bazei canonice, sunt folosiți pentru a exprima într-o formă convenabilă orice vector din spațiul V sub forma:

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

În această expresie, scalarii v_x , v_y și v_z reprezintă coordonatele vectorului \mathbf{v} în baza canonică a spațiului V , iar vectorii \mathbf{i} , \mathbf{j} , \mathbf{k} reprezintă axele bazei.

3.3 Baze și coordonate

Într-un spațiu vectorial de forma $V^3(\mathbf{R})$ pot fi definite oricâte baze cu orientări ale axelor diferite, dar cu origine comună în $(0, 0, 0)$. Din punct de vedere matematic, aceste baze nu trebuie neapărat să aibă axele perpendiculare între ele (liniar independente), însă în majoritatea cazurilor, în practică vom folosi doar baze ortogonale.

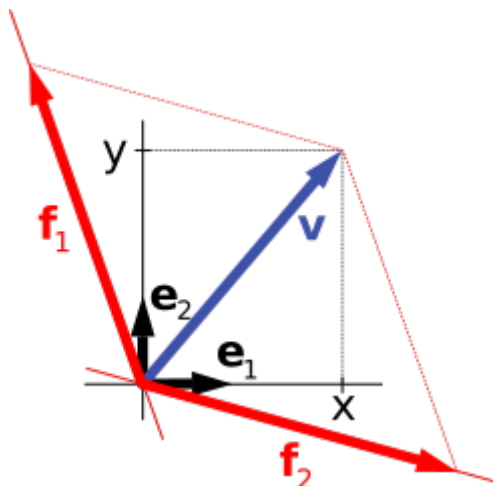
Fie baza $B_0 = \{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ și $B = \{\mathbf{r}, \mathbf{s}, \mathbf{t}\}$. Un vector $\mathbf{v} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ din spațiul V poate fi exprimat relativ la oricare din aceste baze, dar va avea coordonate diferite în fiecare dintre ele.

Coordonatele vectorului \mathbf{v} în baza B pot fi calculate cu ajutorul produsului scalar între vectorul \mathbf{v} și fiecare axă a bazei B :

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

Astfel putem acum exprima vectorul \mathbf{v} relativ la baza B astfel:

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



Un vector v exprimat în funcție de două baze diferite (x, y) și (f_1, f_2)

Pentru calcularea produsului scalar, putem reprezenta coordonatele vectorilor sub forma unor matrici linie sau coloană, ținând cont că aceste reprezentări sunt relative la o anumită bază:

$$v(B_0) = [x \ y \ z], \quad r(B_0) = [r_x \ r_y \ r_z], \quad s(B_0) = [s_x \ s_y \ s_z], \quad t(B_0) = [t_x \ t_y \ t_z]$$

Astfel:

$$x' = [x \ y \ z] * \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = x * r_x + y * r_y + z * r_z$$

$$y' = [x \ y \ z] * \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix} = x * s_x + y * s_y + z * s_z$$

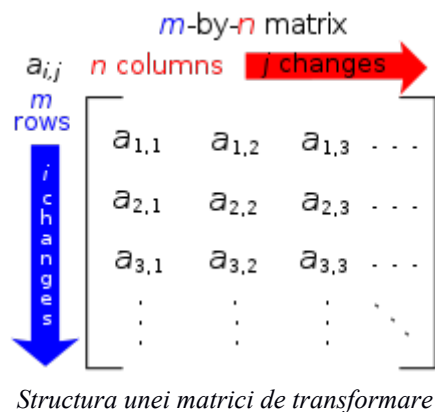
$$z' = [x \ y \ z] * \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = x * t_x + y * t_y + z * t_z$$

Putem concatena aceste 3 operații într-una singură, înmulțind vectorul v cu o matrice M ale cărei coloane reprezintă 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: $\mathbf{v}_B = \mathbf{v}_{B_0} * \mathbf{M}$

Matricea \mathbf{M} se numeste **matricea de trecere** de la baza B_0 la baza B .



3.4 Transformari

O transformare reprezinta operatia bijectiva de traducere a unui vector \mathbf{v} intr-un alt vector \mathbf{v}' prin inmultirea acestuia cu o matrice in modul indicat mai sus. In functie de tipul matricii, 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.

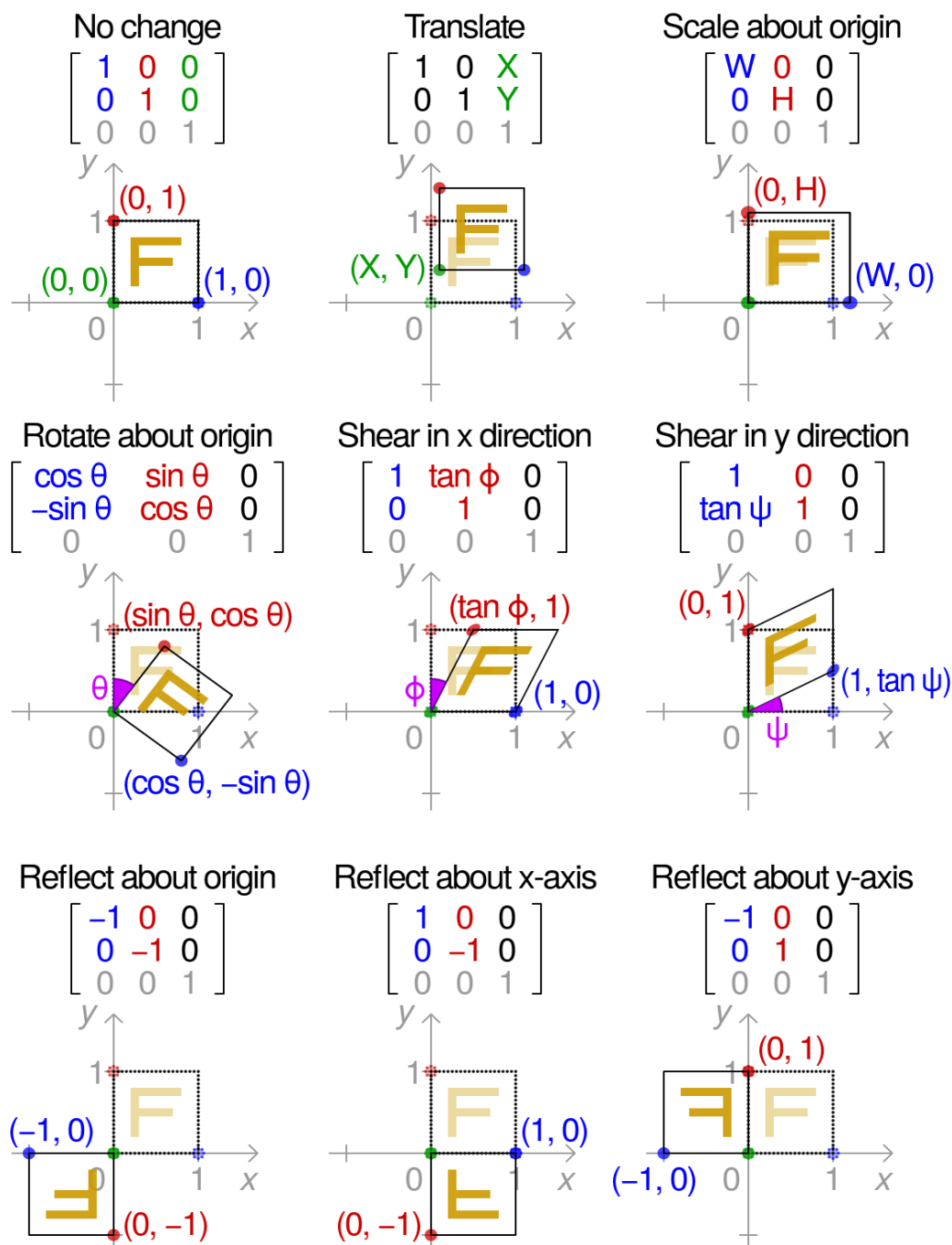
Orice transformare (rotatie, scalare, etc) poate fi privita ca o trecere de la o baza la alta, daca consideram coordonatele vectorului initial definite relativ la baza initiala si transformarea ca matricea de trecere de la acea baza la baza finala. Acest aspect este important in realizarea practica a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

Din punct de vedere matematic, o transformare reprezinta o functie bijectiva care atribuie fiecarui vector \mathbf{v} un alt vector \mathbf{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 \mathbf{v} de dimensiune n printr-un vector \mathbf{v}' de

dimensiune $n+1$, al cărui ultima coordonată va fi 1. Astfel, în matricea transformării ultima linie (sau coloana dacă se folosește înmulțirea vector-coloana cu matrice) va conține translația.

În continuare vom prezenta succint principalele tipuri de transformări, exemplificate pe un spațiu 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 obicei pentru a indica orientarea unei suprafete (sau poligon) pentru a putea calcula iluminarea intr-un mod realist.

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

Un obiect 3D consta intr-o lista de verteci, o lista de indecsi care definesc ordinea in care sunt parcursi acesti verteci 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 vertecsi unui obiect pentru a-l manipula ca intreg (in anumite cazuri se transforma si normalele, nu doar vertecsi, 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 reprezintă o transformare care “deplasează” uniform toți vertexii pe o distanță \mathbf{d} , independentă de poziția inițială a vertexilor:

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

Asadar distanțele și pozițiile relative ale vertexilor sunt conservate de către translatie, precum și orientarea atât relativă cât și absolută a suprafețelor.

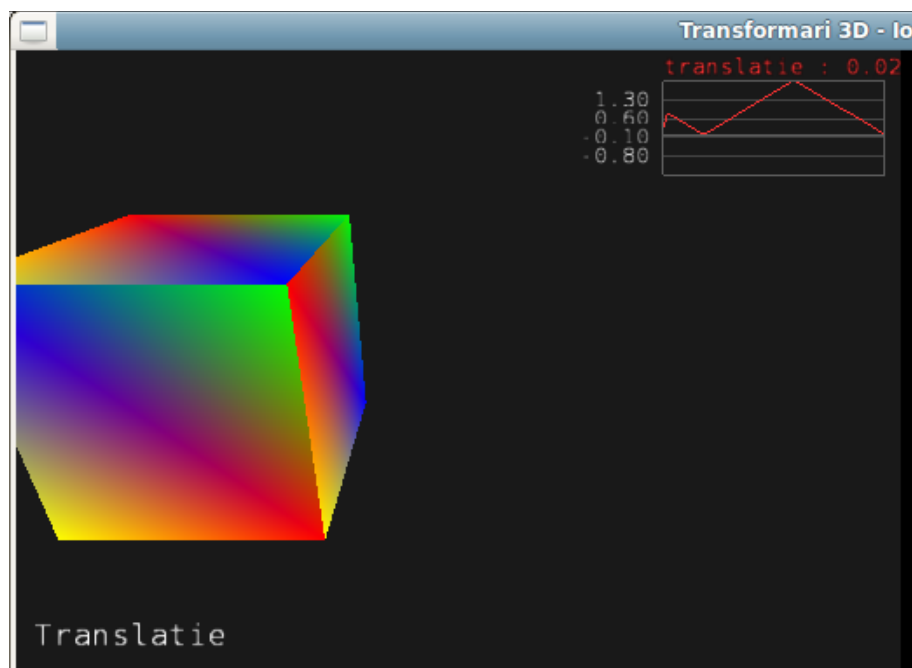
$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

$$P' = P \cdot T$$

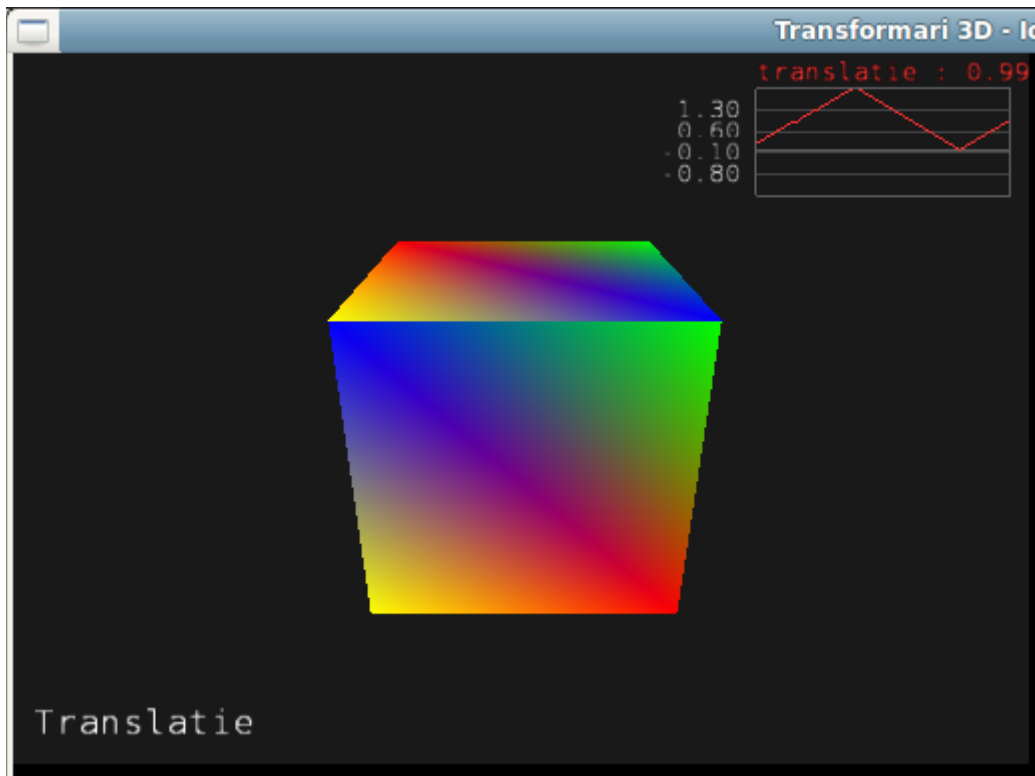
$$\begin{aligned} [X' \ Y' \ Z' \ 1] &= [X \ Y \ Z \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \\ &= [X + t_x \ Y + t_y \ Z + t_z \ 1] \end{aligned}$$

Expresia matematică a translatiei

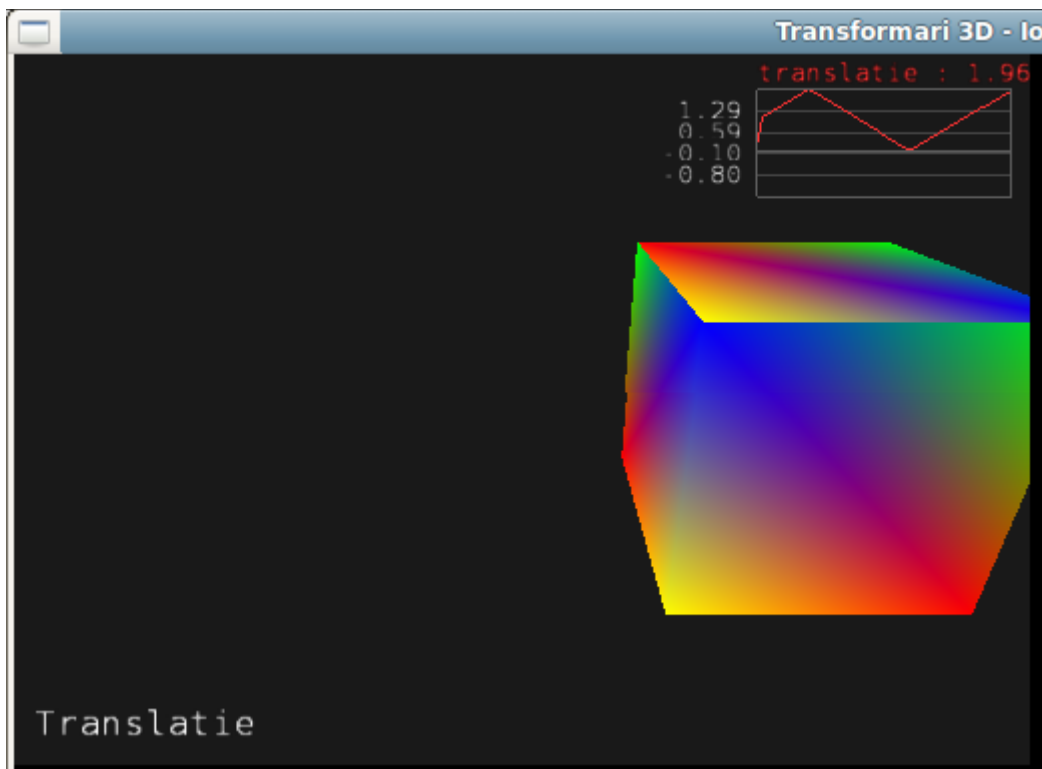
În exemplul de mai jos putem vedea translatia în acțiune:



Poziția originală $z=-1$, $trans=0$



pozitie intermediara $z=0$, $trans=1$



pozitie finala $z=1$, $trans=2$

Secvența anterioară a fost obținută printr-o operație de interpolare liniară între două poziții fixe (capetele), ceea ce oferă posibilitatea animării imaginii într-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 transformărilor obiectului din rezultatul interpolării:

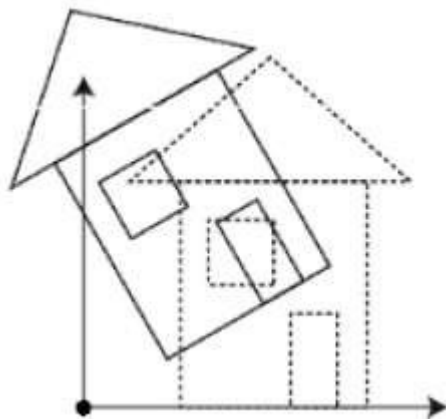
```
void PathController::update(float dt) {
    lerper_.update(dt);
    body_->setPosition(lerper_.value().position,
                      physics::DynamicBody::TransformSpace::Parent);
    body_->setOrientation(lerper_.value().orientation,
                          physics::DynamicBody::TransformSpace::Parent);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

Calcularea poziției 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 vertexsi si suprafete, insa orientarea per ansamblu a obiectului se modifica. Obiectul rezultat este asemenea celui initial.



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

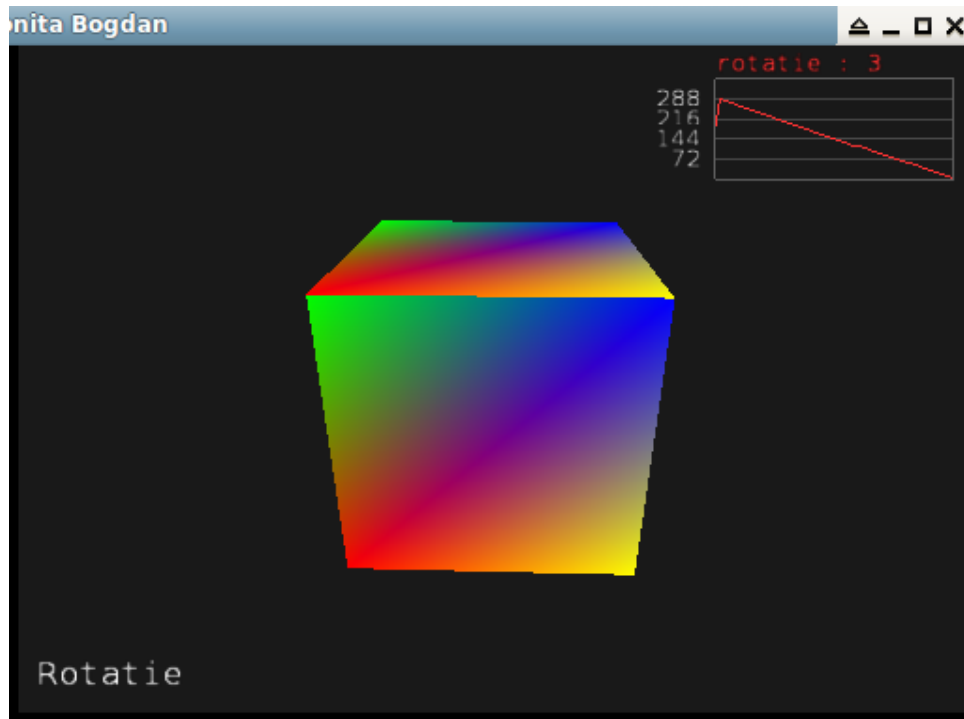
Matricea de rotatie se construiește in functie de axa de rotatie aleasa, X, Y, sau Z. O matrice de rotatie poate fi obtinuta si prin inmultirea mai multor matrici de rotatie cu axe diferite, obtinandu-se astfel o rotatie compusa, care va avea o axa noua de rotatie, diferita de cele initiale.

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

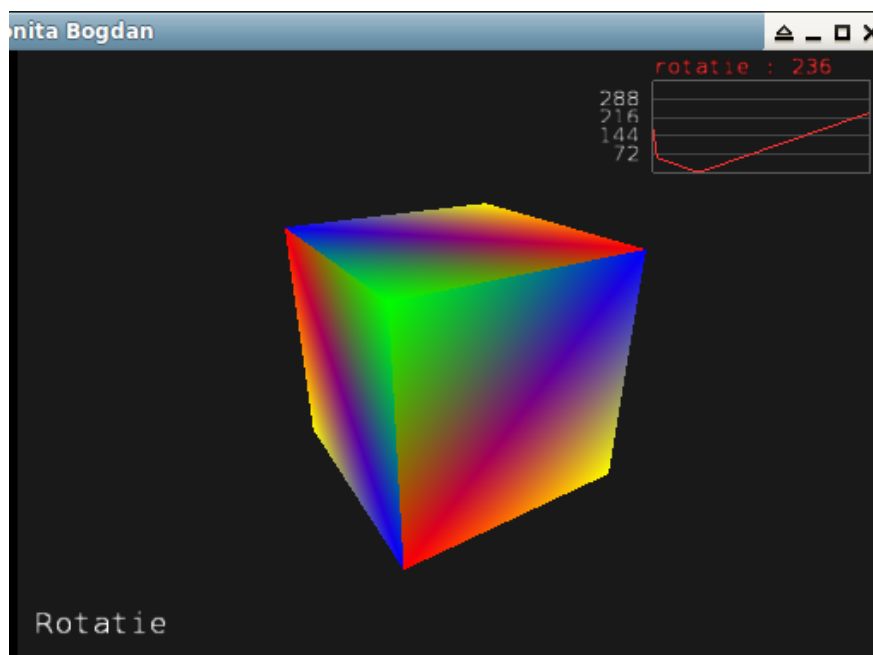
Matrici de rotatie pentru axele principale

În programarea grafică se folosesc de asemenea și rotațiile reprezentate prin **quaternioni**, aceștia fiind niște obiecte matematice 4-dimensionale cu proprietăți avantajoase, cum ar fi că se pot combina ușor între ei și nu au anumite probleme pe care le au matricile, cum ar fi *driftingul*.

În exemplul de mai jos vom vedea rotația în acțiune:



orientarea inițială $r_y=0$



orientare intermediară $r_y=240$ grade

Secvența de interpolare pentru rotație:

```
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);
```

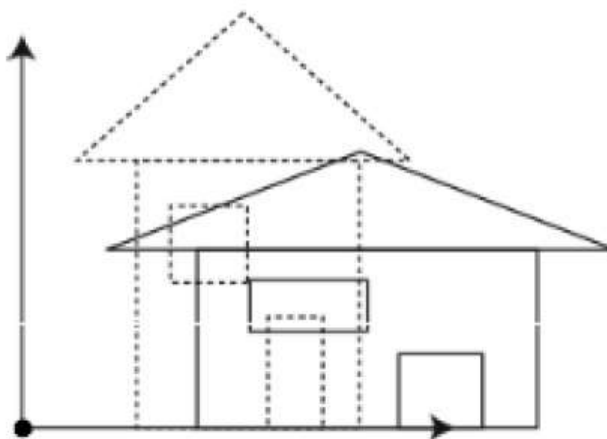
Se observă că s-a folosit o rotație în jurul axei $\{0, 1, 0\}$ adică axa Y.

Calcularea orientării 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 modifică atât distanțele relative dintre vârfuri, cât și pozițiile relative ale acestora, pe una sau mai multe axe. Obiectul rezultat nu mai este asemenea obiectului inițial decât dacă se folosește o scalare uniformă, adică cu aceiași factori pe toate axele.



Scalarea într-un spațiu tridimensional este definită prin 3 valori $\{S_x, S_y, S_z\}$ care reprezintă factorii de scalare pe fiecare axă în parte.

Dacă cei trei factori sunt egali, atunci scalarea se numește *uniformă* și va păstra forma obiectului, altfel obiectul va apărea deformat (întins sau comprimat pe una sau mai multe direcții).

Dacă factorul de scalare pe o anumită axă este subunitar, atunci scalarea va avea micsorarea obiectului; dacă este supraunitar, îl va mari, iar dacă este egal cu 1, scalarea nu are nici un efect.

$$S < 1 \Rightarrow \text{micsorare}$$

$$S > 1 \Rightarrow \text{marire}$$

$$S = 1 \Rightarrow \text{nici un efect}$$

Distanța dintre doi vârfuri \mathbf{v}_1 și \mathbf{v}_2 este modificată de scalare astfel:

Fie vectorul $\mathbf{d} = \mathbf{v}_2 - \mathbf{v}_1$ astfel $|\mathbf{d}| = \text{dist}(\mathbf{v}_1, \mathbf{v}_2)$ reprezintă distanța liniară dintre cei doi vârfuri.

Fie scalarea $S = \{s_x, s_y, s_z\}$ și \mathbf{v}_1' și \mathbf{v}_2' vârfurile transformate, iar $\mathbf{d}' = \mathbf{v}_2' - \mathbf{v}_1'$, $|\mathbf{d}'|$ distanța dintre acestea. Coordonatele vectorilor \mathbf{v}_1' și \mathbf{v}_2' se calculează astfel:

$$\mathbf{v}_1' = \{v_{1.x} * s_x, v_{1.y} * s_y, v_{1.z} * s_z\} \text{ și } \mathbf{v}_2' = \{v_{2.x} * s_x, v_{2.y} * s_y, v_{2.z} * s_z\}$$

rezultă ca $\mathbf{d}' = \{s_x * (v_{2.x} - v_{1.x}), s_y * (v_{2.y} - v_{1.y}), s_z * (v_{2.z} - v_{1.z})\}$, ceea ce înseamnă că distanța dintre vârfurile transformate, raportată la fiecare axă în parte, este înmulțită cu factorul de scalare.

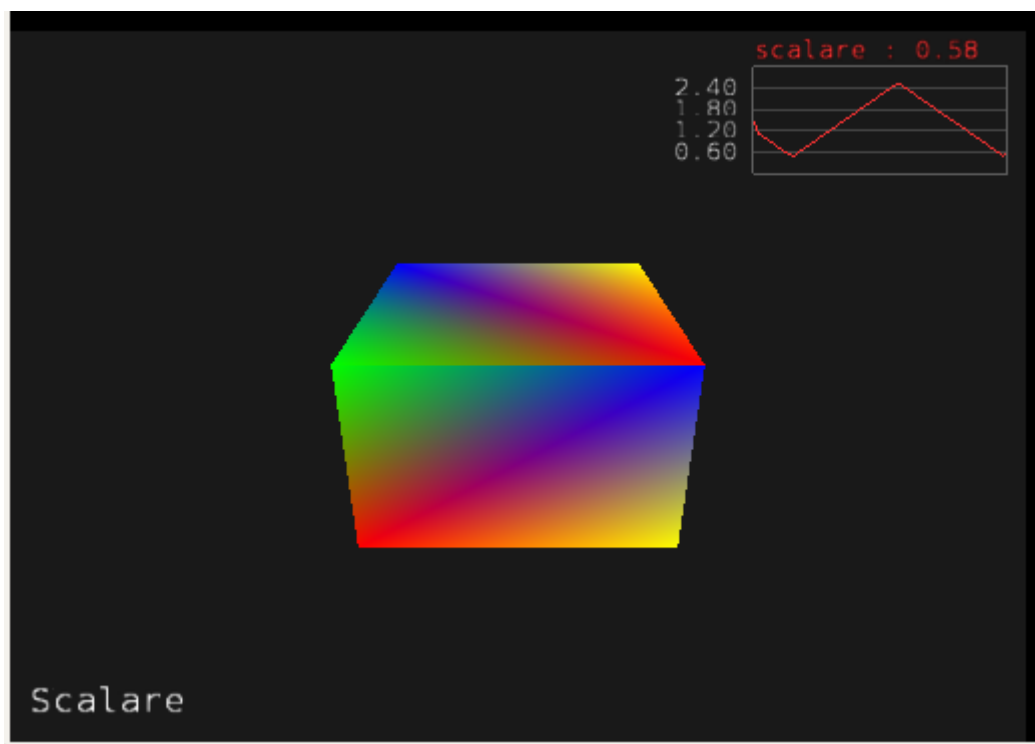
$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P' = P \cdot S$$

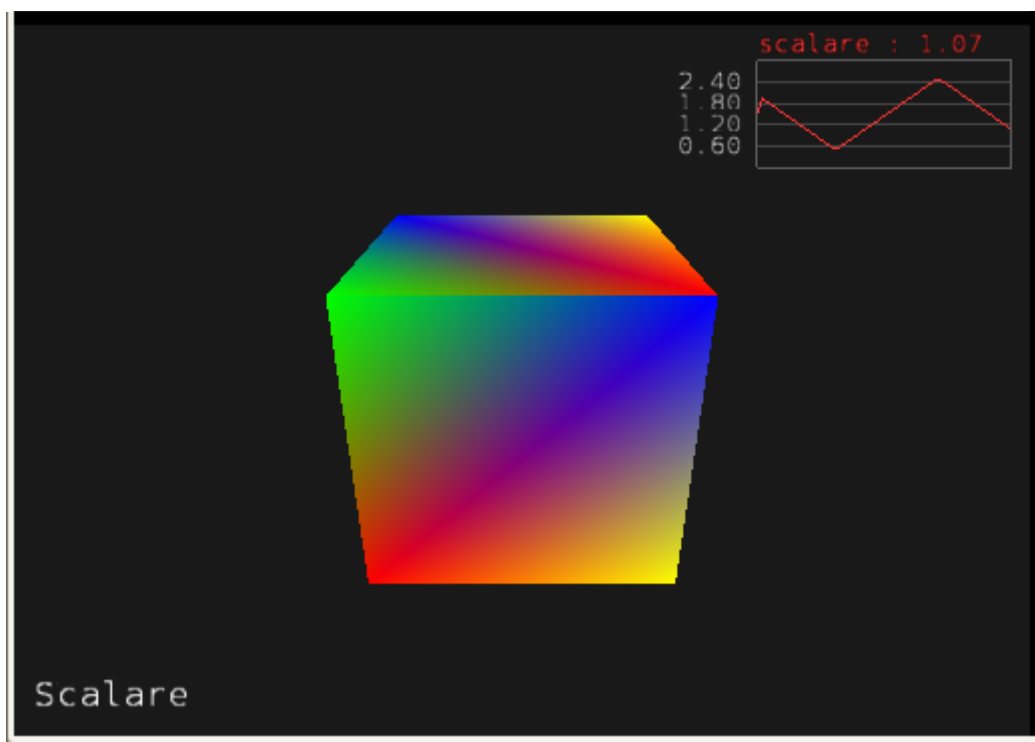
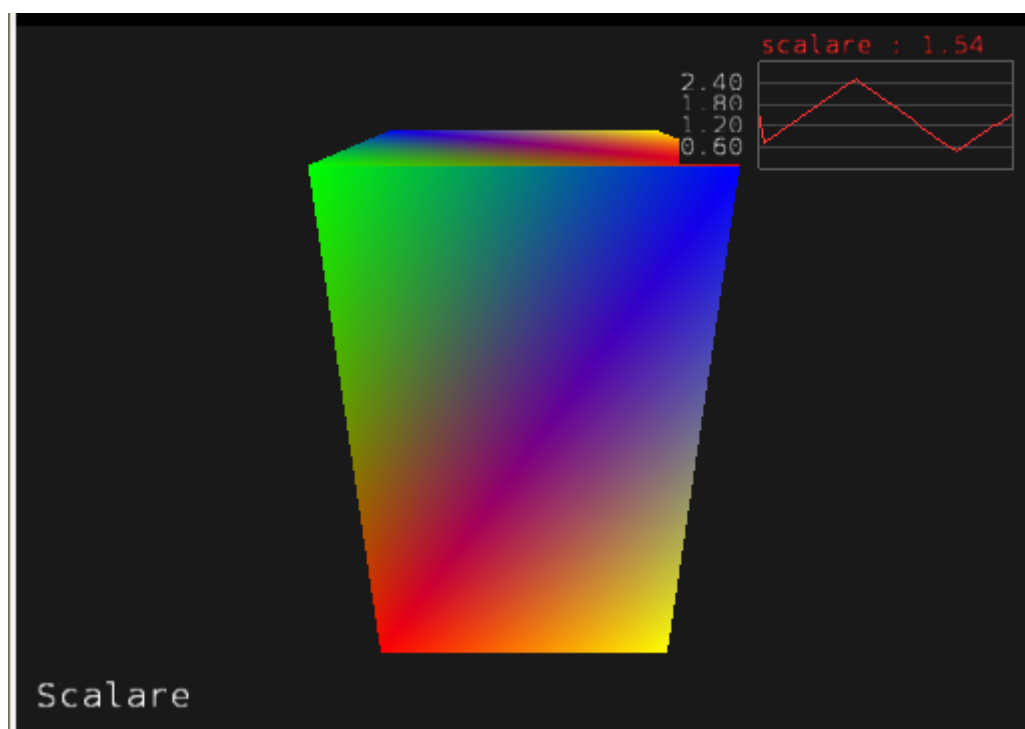
$$\begin{aligned} [X' \ Y' \ Z' \ 1] &= [X \ Y \ Z \ 1] \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= [X \cdot S_x \ Y \cdot S_y \ Z \cdot S_z \ 1] \end{aligned}$$

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 scalariei il reprezinta **reflexia**, in care se aplica un factor de scalare de -1 pe axa dorita.

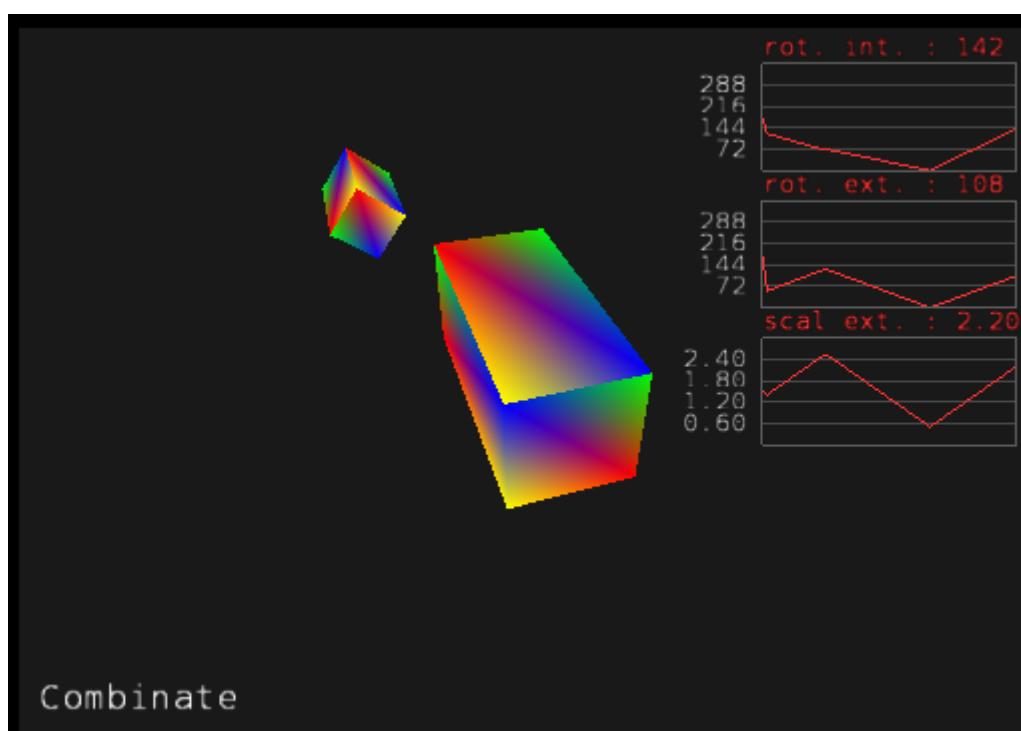
3.4.5 Combinarea transformărilor

Avantajul reprezentării transformărilor sub forma matriceală este acela că putem combina cu ușurință mai multe transformări, înmulțind matricile acestora.

Deoarece înmulțirea matricilor nu este comutativă, trebuie să acordăm atenție ordinii în care înmulțim matricile de transformare, pentru că o ordine diferită va conduce la efecte diferite, așa cum este și de așteptat: o translație aplicată înaintea unei scalări va fi scalată și ea la rândul ei, pe când translația aplicată după scalare va acționa ca atare.

Mai jos putem observa o combinație a următoarelor transformări:

1. cubul mic este rotit în jurul axei proprii X
2. cubul mic este poziționat (translatat) relativ la cubul mare
3. cubul mic este atașat cubului mare și scalat împreună pe axa X \rightarrow axa X a cubului mic este rotită
4. cele două cuburi sunt rotite împreună pe axa Y \rightarrow scalarea va fi și ea rotită



Combinarea a trei transformări

Astfel, cea de-a doua rotație are ca efect și modificarea poziției cubului mic, deoarece și translația aplicată anterior este rotită.

3.5 Modul de funcționare al OpenGL

Libraria OpenGL are două moduri de funcționare:

1. modul **legacy** (versiunea 1) – nu mai este de actualitate, folosirea sa este descurajată, însă este păstrat în continuare în scop academic.
2. modul standard actual (de la versiunea 2 în sus; versiunea curentă este 4)

3.5.1 Modul legacy (v1)

Se bazează pe ceea ce se numește “fixed pipeline”, adică algoritmul de procesare a datelor în vederea obținerii imaginii finale este fix, dar parametrizabil. Astfel, utilizatorul pune la dispoziția bibliotecii date pentru diverse scopuri predefinite (verteci, texturi, indecsi etc) și configurează diverși pași ai procesului de randare prin comenzi date bibliotecii.

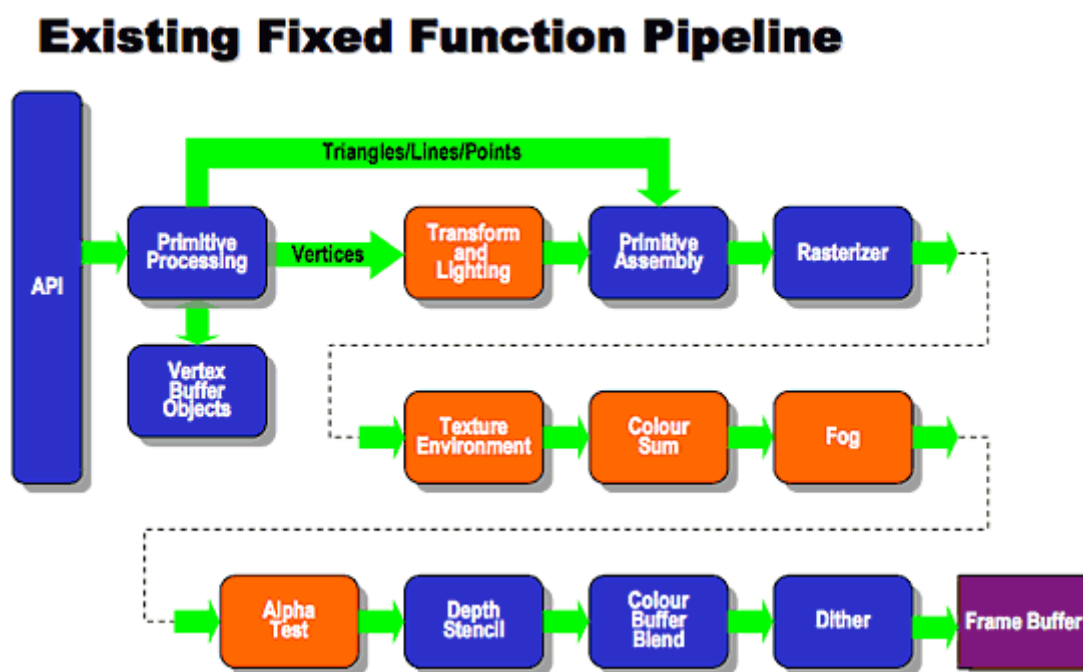


Diagrama procesului fixed-pipeline

Pășii principali ai procesului de randare sunt următorii:

- procesarea primitivelor → are ca rezultat o listă de verteci ce trebuie procesate ulterior
- transformare și iluminare → aplică transformările (în modul legacy există tipuri predefinite de transformări: **world** – poziționarea în spațiu, **view** – transformarea în funcție de poziția camerei, **projection** – proiecția din 3D în 2D, **viewport** – așezarea în imaginea finală) apoi

calculează iluminarea pe suprafețe (în modul legacy există un număr fix de lumini ce pot fi folosite)

- asamblarea primitivelor – aici sunt tăiate triunghiurile ce nu se încadrează în spațiul vizibil și creați noi vertecși pentru a le “închide”
- rasterizare – procesul de transformare a primitivelor în pixeli
- texturare, culoare, ceață – alterarea culorii pixelilor în funcție de textura aplicată, culoarea vertecșilor interpolată, setările de “ceață”
- alpha test – se verifică transparența pixelilor
- depth test și stencil test – anumiți pixeli pot fi refuzați dacă nu îndeplinesc condițiile de adâncime sau dacă e folosit un șablon
- color buffer blend – se mixează culoarea pixelului cu cel existent deja în imagine

3.5.2 Modul modern (v2+)

Se bazează pe un proces complet programabil de către utilizator. Astfel, utilizatorul bibliotecii își definește singurul rol al datelor pe care le folosește și modul în care acestea sunt procesate.

Procesarea vertecșilor și a texturilor se face de către procesorul grafic dedicat (GPU) prin programe speciale scrise de utilizator cunoscute sub denumirea de “shader”. Astfel programatorul poate procesa în orice mod dorește datele grafice, putând obține astfel efecte mult mai complexe și o imagine finală foarte realistă. Dezavantajul însă este complexitatea crescută a întregului proces de dezvoltare.

Operațiile de transformare se întâmplă în programele shader care sunt executate de către GPU. Acolo ajung matricile de transformare transmise din programul principal, precum și secvențele de vertecși, normalele, indicii, coordonatele de textură, etc.

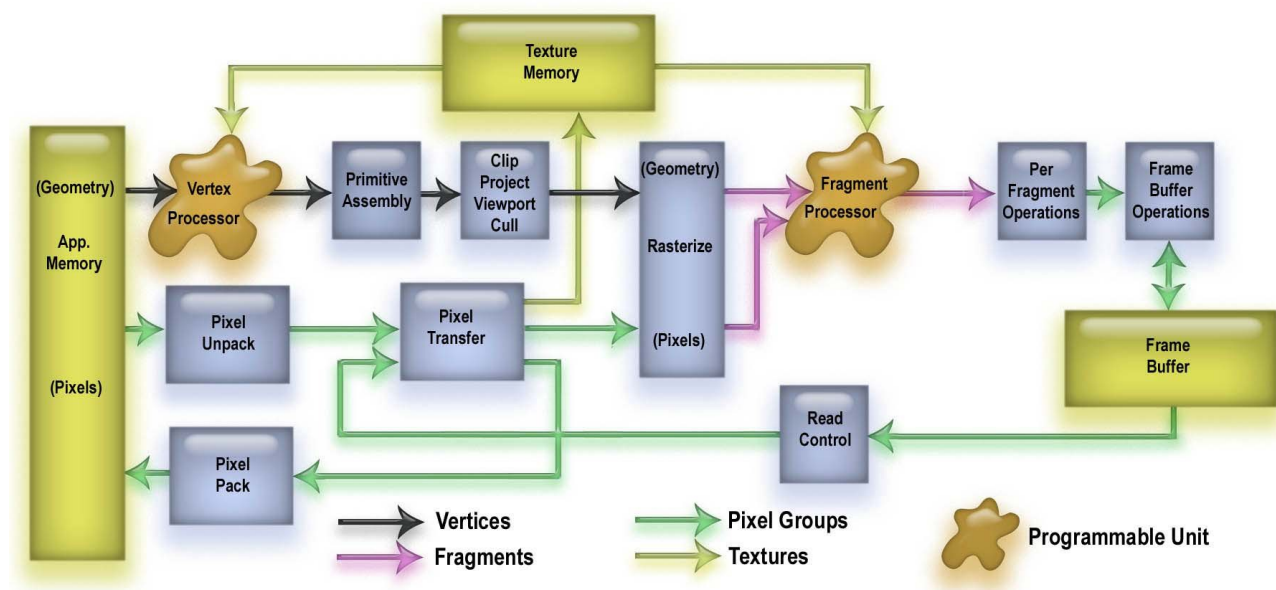


Diagrama modului programabil

Astfel anumiți pași din proces sunt lăsați la latitudinea programatorului, cum ar fi

- transformarea
- iluminarea (iluminarea poate fi făcută per pixel în loc de vertex pentru un efect mai realist)
- aplicarea texturilor și a cettii
- post-procesarea (poate avea mai mulți pași)

În versiunile mai noi, programatorul are și posibilitatea generării dinamice a vertexurilor direct în procesorul grafic, prin programele de tip **geometry shader**.

În aplicația demonstrativă am folosit versiunea de API OpenGL 4, cu programe shader destul de simple, neavând nevoie de efecte grafice complicate.

Pe pagina următoare sunt câteva exemple de astfel de programe folosite.

Programul shader pentru procesarea vertexurilor 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 vertexurilor 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 );
}

```

Secvența de cod care controlează randarea cuburilor și 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);
}

```

3.5.3 Asincronicitate

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.

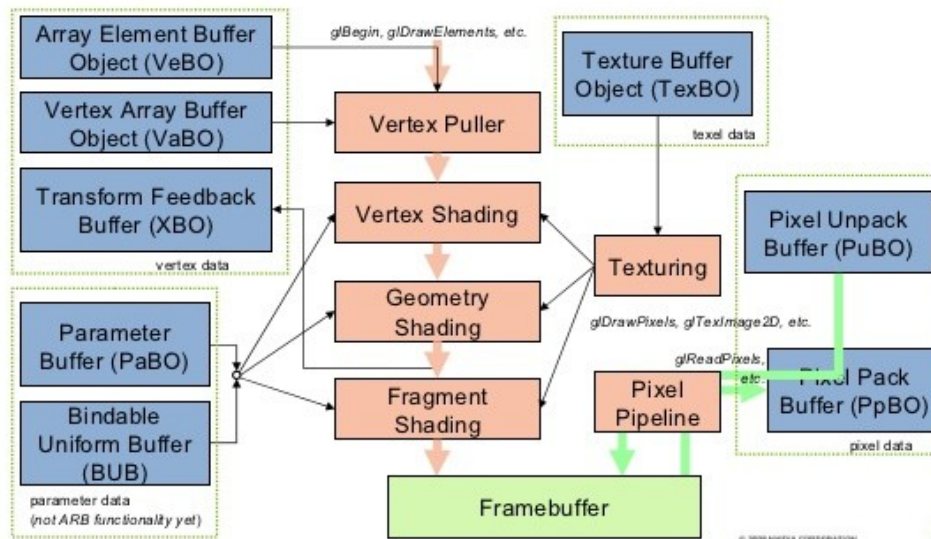
Astfel, comenzile date de programator sunt inregistrate intr-o coada de unde sunt preluate apoi de catre procesorul grafic atunci cand este liber. In momentul in care se doreste afisarea imaginii, libraria asteapta automat pana cand procesorul grafic termina de executat toate operatiile.

3.5.4 Obiecte Buffer

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 verteci, Texturi in care pot fi incarcate imaginile ce sunt aplicate pe primitive, etc:



Buffer Centric View of OpenGL



Diferitele tipuri de bufferi in OpenGL

3.6 Modelul de date in OpenGL

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

Pentru programator este important să cunoască aceste tipuri pentru a putea trimite datele în formatul corect, altfel aplicația nu poate funcționa.

În geometria analitică aplicată în informatică, pentru precizie sporită, se folosește de obicei tipul de date **double** stocat în memorie pe 64 biți. Acest tip conferă precizie foarte mare în calcule, însă ocupând multă memorie este inadecvat pentru lucrul cu grafică de performanță.

De aceea OpenGL folosește tipul **float** pe 32 biți. Acesta oferă suficientă precizie la o performanță superioară. În funcție de tipul datelor transmise, fiecare element poate avea unul sau mai multe componente **float**:

- vertex: 3 componente **float**
- matrice: $4 \times 4 = 16$ elemente **float**
- quaternion: 4 elemente **float**
- coordonate în textură UV: 2 elemente **float**
- culoare RGB: 3 elemente float
- culoare RGBA: 4 elemente float
- normală: 3 elemente float

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

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

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

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

În modul legacy (fixed-pipeline) este folosită ordinea column-major, dar în modul modern programatorul poate alege oricare ordine, atât timp cât ține cont apoi de aceasta la ordinea de înmulțire a matricilor între ele și cu vectori.

Row-major order

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

$$x' = x * a + y * d + z * g$$

$$y' = x * b + y * e + z * h$$

$$z' = x * c + y * f + z * i$$

Column-major order

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

$$x' = a * x + d * y + g * z$$

$$y' = b * x + e * y + h * z$$

$$z' = c * x + f * y + i * z$$

Pentru indici, OpenGL folosește date de tip întreg, pe 8, 16, sau 32 biți, la alegerea programatorului.

Pentru texturi, în general este folosit tipul RGBA pe 32 biți, însă pentru economia memoriei, pot fi folosite texturi comprimate.

În aplicația practică am folosit matrici în ordinea column-major și indici pe 16 biți.

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

3.7 Libraria GLM

Această bibliotecă este independentă de OpenGL, însă oferă în C++ tipuri de date ce se mulează perfect pe cele native din shader, cum ar fi `vec3`, `vec4`, `mat4x4` etc.

Deoarece este necesar ca anumite calcule vectoriale să fie făcute în programul principal, apare nevoia unei biblioteci de acest gen, care să ofere tipuri de date pentru vectori și matrici, precum și operații pe vectori, matrici, și transformări.

GLM este o bibliotecă header-only, adică compusă doar din fișiere `*.h` în care sunt definite tipuri templatizate și operațiile aferente.

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

pagina bibliotecii GLM

În aplicația practică am folosit tipurile de date din GLM și operațiile pentru calcularea matricilor de transformare, calcul pe quaternioni și pe vectori, interpolare între quaternioni și vectori, etc.

Calcularea inversei unei matrici:

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

Transformarea unui quaternion într-o matrice de rotație și 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}};
```

Înmulțirea a două matrici:

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


Construcția 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
        },
    });
}
```

```

{
    {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 transformărilor în OpenGL

Aplicarea unei transformări geometrice în cadrul OpenGL se poate face în două moduri:

1. pe CPU (în software) – vertexii, normalele etc sunt transformate în cadrul programului principal, folosind operații de genul celor puse la dispoziție de biblioteca GLM, apoi datele astfel pregătite sunt transmise către OpenGL pentru a fi randate.
2. Pe GPU (în hardware) – vertexii, normalele etc sunt transmise ca atare către OpenGL, netransformate, împreună cu matricile de transformare aferente. Transformările vor fi executate de procesorul grafic pe baza acestor date, fie prin procesul fixed-pipeline, fie în programele shader scrise de programator

În general este preferată a doua variantă, pentru că oferă flexibilitate mai mare și o performanță mult mai bună, deoarece procesorul grafic este optimizat pentru a efectua acest tip de operații. Procesorul grafic are zeci sau sute de unități de transformare care pot procesa tot atâtea vertexi în paralel, practic simultan. De asemenea se poate lucra ușor cu ierarhii de obiecte în acest mod, deoarece singurele date care se modifică sunt matricile de transformare, iar vertexii pot fi copiați de la început în memoria video în obiecte de tipul VBO și nu mai sunt modificați apoi, ceea ce duce la un câștig de performanță suplimentar.

În aplicații sunt folosite ambele metode împreună, 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 secvență de 16 numere de tip float, ce sunt interpretate ca o matrice de 4 pe 4, row-major sau column-major (depinde de programator). În aplicația practică, matricile sunt continuate în obiecte de tip **glm::mat4**, care pe lângă stocarea propriu-zisă a numerelor oferă și operații pe acestea.

În cadrul programelor shader, există mai multe categorii de date:

- Uniform – date care își păstrează valoarea de la un vertex la altul
- Varying – date de ieșire din vertex shader care sunt interpolate apoi de GPU și transmise către fragment shader (de tipul culoare, poziție, 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 <i>location</i> , |
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

| | |
|--------------------------|--------------------------------|
| void glUniformMatrix3fv(| GLint <i>location</i> , |
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

| | |
|--------------------------|--------------------------------|
| void glUniformMatrix4fv(| GLint <i>location</i> , |
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

| | |
|----------------------------|--------------------------------|
| void glUniformMatrix2x3fv(| GLint <i>location</i> , |
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

| | |
|----------------------------|--------------------------------|
| void glUniformMatrix3x2fv(| GLint <i>location</i> , |
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

| | |
|----------------------------|--------------------------------|
| void glUniformMatrix2x4fv(| GLint <i>location</i> , |
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

| | |
|----------------------------|-------------------------|
| void glUniformMatrix4x2fv(| GLint <i>location</i> , |
|----------------------------|-------------------------|

| | |
|--|--------------------------------|
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

| | |
|----------------------------|--------------------------------|
| void glUniformMatrix3x4fv(| GLint <i>location</i> , |
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

| | |
|----------------------------|--------------------------------|
| void glUniformMatrix4x3fv(| GLint <i>location</i> , |
| | GLsizei <i>count</i> , |
| | GLboolean <i>transpose</i> , |
| | const GLfloat <i>*value</i>); |

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 <i>program</i> , |
| | const GLchar <i>*name</i>); |

<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 vertecșilor către OpenGL

Vertecșii pot fi transmiși direct în momentul în care se face randarea, sau pot fi copiați în memoria video într-un obiect VBO (Vertex Buffer Object) înainte, metoda preferată pentru performanță superioară. În continuare vom prezenta această metodă.

Crearea unui obiect VBO:

| | |
|--------------------|----------------------------|
| void glGenBuffers(| GLsizei <i>n</i> , |
| | GLuint * <i>buffers</i>); |

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/genBuffers.xhtml>

Popularea VBO cu date presupune 2 pași: declararea scopului bufferului și transferul propriu-zis al datelor:

| | |
|--------------------|-------------------------|
| void glBindBuffer(| GLenum <i>target</i> , |
| | GLuint <i>buffer</i>); |

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/bindBuffer.xhtml>

| | |
|--------------------|------------------------------|
| void glBufferData(| GLenum <i>target</i> , |
| | GLsizeiptr <i>size</i> , |
| | const GLvoid * <i>data</i> , |
| | GLenum <i>usage</i>); |

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/BufferData.xhtml>

Exemplu din aplicația practică de transmitere a vertecșilor și indcșilor în 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;
}
```

Odată transmise datele, mai este necesar un pas, și anume crearea unei legături între aceste date și atributele ce vor ajunge în programele shader, precum și specificarea formatului datelor:

În OpenGL există 3 funcții care îndeplinesc această sarcină:

| | |
|--|--|
| <code>void glVertexAttribPointer(</code> | <code>GLuint <i>index</i>,</code> |
| | <code>GLint <i>size</i>,</code> |
| | <code>GLenum <i>type</i>,</code> |
| | <code>GLboolean <i>normalized</i>,</code> |
| | <code>GLsizei <i>stride</i>,</code> |
| | <code>const GLvoid * <i>pointer</i>);</code> |

| | |
|---|--|
| <code>void glVertexAttribIPointer(</code> | <code>GLuint <i>index</i>,</code> |
| | <code>GLint <i>size</i>,</code> |
| | <code>GLenum <i>type</i>,</code> |
| | <code>GLsizei <i>stride</i>,</code> |
| | <code>const GLvoid * <i>pointer</i>);</code> |

| | |
|---|--|
| <code>void glVertexAttribLPointer(</code> | <code>GLuint <i>index</i>,</code> |
| | <code>GLint <i>size</i>,</code> |
| | <code>GLenum <i>type</i>,</code> |
| | <code>GLsizei <i>stride</i>,</code> |
| | <code>const GLvoid * <i>pointer</i>);</code> |

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

Exemplu din aplicația practică:

```
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 reprezintă procesul prin care toate datele sunt procesate în vederea producerii imaginii finale. Odată transmise datele, programatorul trebuie să execute comenzi care să comunice bibliotecii OpenGL ce operații se doresc a fi făcute cu datele respective. Aceste comenzi pornesc efectiv procesul de randare în procesorul grafic.

Există două funcții diferite pentru acest scop:

| | |
|---------------------------------|-------------------------------------|
| <code>void glDrawArrays(</code> | <code>GLenum <i>mode</i>,</code> |
| | <code>GLint <i>first</i>,</code> |
| | <code>GLsizei <i>count</i>);</code> |

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

Aceasta este pentru date neindexate (vertex-urile sunt citite și interpretate secvențial).

| | |
|-----------------------------------|--|
| <code>void glDrawElements(</code> | <code>GLenum <i>mode</i>,</code> |
| | <code>GLsizei <i>count</i>,</code> |
| | <code>GLenum <i>type</i>,</code> |
| | <code>const GLvoid * <i>indices</i>);</code> |

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

Aceasta este pentru date indexate, în care ordinea vertex-urilor este descrisă separat printr-o secvență de indici. Astfel un vertex poate fi refolosit specificând mai mulți indici cu aceeași valoare.

De obicei este preferată a doua metodă pentru că nu necesită duplicarea vertex-urilor în anumite situații, și de asemenea este și mai performantă.

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

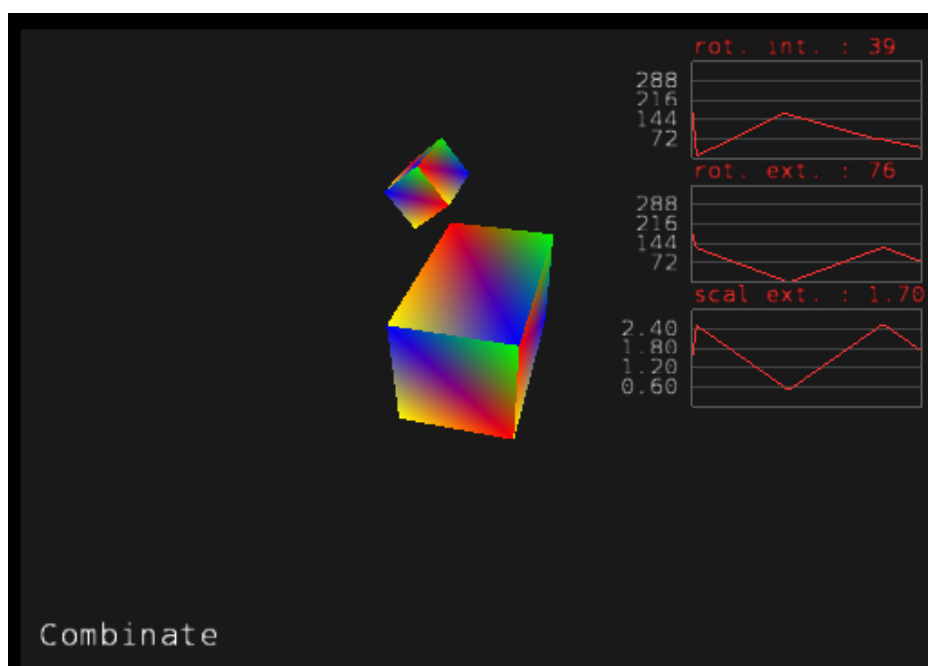
O ierarhie de obiecte reprezintă o colecție de obiecte ce se comportă la nivel macro ca un tot unitar, însă fiecare element component al său are anumite grade de libertate, reprezentate prin transformări de translație, rotație sau scalare. Această colecție are o structură arborescentă, cu un singur părinte la origine (radacina sau root).

Acest model este foarte des utilizat pentru că permite crearea animațiilor complexe cu ușurință, programând doar transformările relative dintre obiecte, ca apoi ele să se compună într-o transformare în scena complexă.

Pentru a obține acest efect, este necesar ca fiecare element al ierarhiei să definească o transformare proprie, și să i se atribuie o bază proprie. Transformarea fiecărui element este considerată relativă la baza părintelui elementului, care este și ea exprimată relativ la baza părintelui acestuia și așa mai departe până la elementul radacină.

Pentru calcularea transformării în spațiul scenei al fiecărui element, se înmulțesc succesiv matricile elementului cu matricea părintelui, și așa mai departe, până la elementul radacină. Transformarea astfel obținută este apoi aplicată doar pe vârfurile elementului inițial.

Exemplu de ierarhie de obiecte din aplicația practică:



Elementul cheie pentru realizarea acestei ierarhii îl 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 în funcție 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 observă înmulțirea matricii locale cu matricea părintelui.

3.11 Detalii tehnice

În acest capitol vom descrie librăriile folosite pentru realizarea aplicației precum și pașii necesari pentru a putea compila și rula aplicația.

Aplicația este scrisă în limbajul **C++14** folosind doar librării **cross-platform**, ceea ce înseamnă că poate fi compilată pentru orice sistem de operare: Linux, Windows, MacOS, etc

Aplicația a fost dezvoltată pe **Linux**, folosind compilatorul **g++ 4.9.2** și mediul de dezvoltare **Eclipse** și folosește următoarele librării:

- GLEW <http://glew.sourceforge.net/>
 - o librărie ce ajută lucrul cu OpenGL detectând automat ce extensii există pe sistem
- GLFW 3 <http://www.glfw.org/>
 - o librărie ce abstractizează mediul vizual, pune la dispoziție funcții de creare și manipulare a ferestrelor și de detectare a inputului (mouse/tastatură)
- GL 4 (pe linux este în pachetul mesa-devel)
 - librăria de OpenGL
- libpng <http://www.libpng.org/pub/png/libpng.html>
 - pentru încărcarea texturilor

Sursele aplicației sunt disponibile pe GitHub la această adresă: <https://github.com/bog2k3/licenta>

În continuare vom descrie pașii necesari pentru compilarea aplicației pe **Linux**, cu referire la distribuția 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 librării

- pentru GLEW se va descarca de pe site-ul indicat
- pentru GLFW se va descarca de pe site-ul indicat
- librăria GLM poate fi instalată din repository, dar este o versiune veche
 - `sudo apt-get install glm`
 - este de preferat să 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 aplicației

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

Dacă compilarea se termină cu succes, aplicația poate fi rulată cu comanda:

```
Release/licenta
```

Se poate folosi și mediul Eclipse pentru a deschide proiectul și 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.

Transformările 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 transformările 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 transformărilor, cum ar fi interpolările pe curbe sau ierarhiile complexe de obiecte, un caz aparte fiind reprezentat de personajele din jocuri care folosesc o tehnica de animatie numita “skinning”. Aceasta tehnica reprezinta aplicarea de transformari pe un sistem de oase invizibil de care sunt legati apoi vertecsii modelului, iar transformările sunt mixate si aplicate pe acestia.

Transformările sunt folosite intens nu numai la miscarea obiectelor in scena, ci si la vizualizarea scenelor statice, unde transformările 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 transformările 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, Francois 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_;
    }

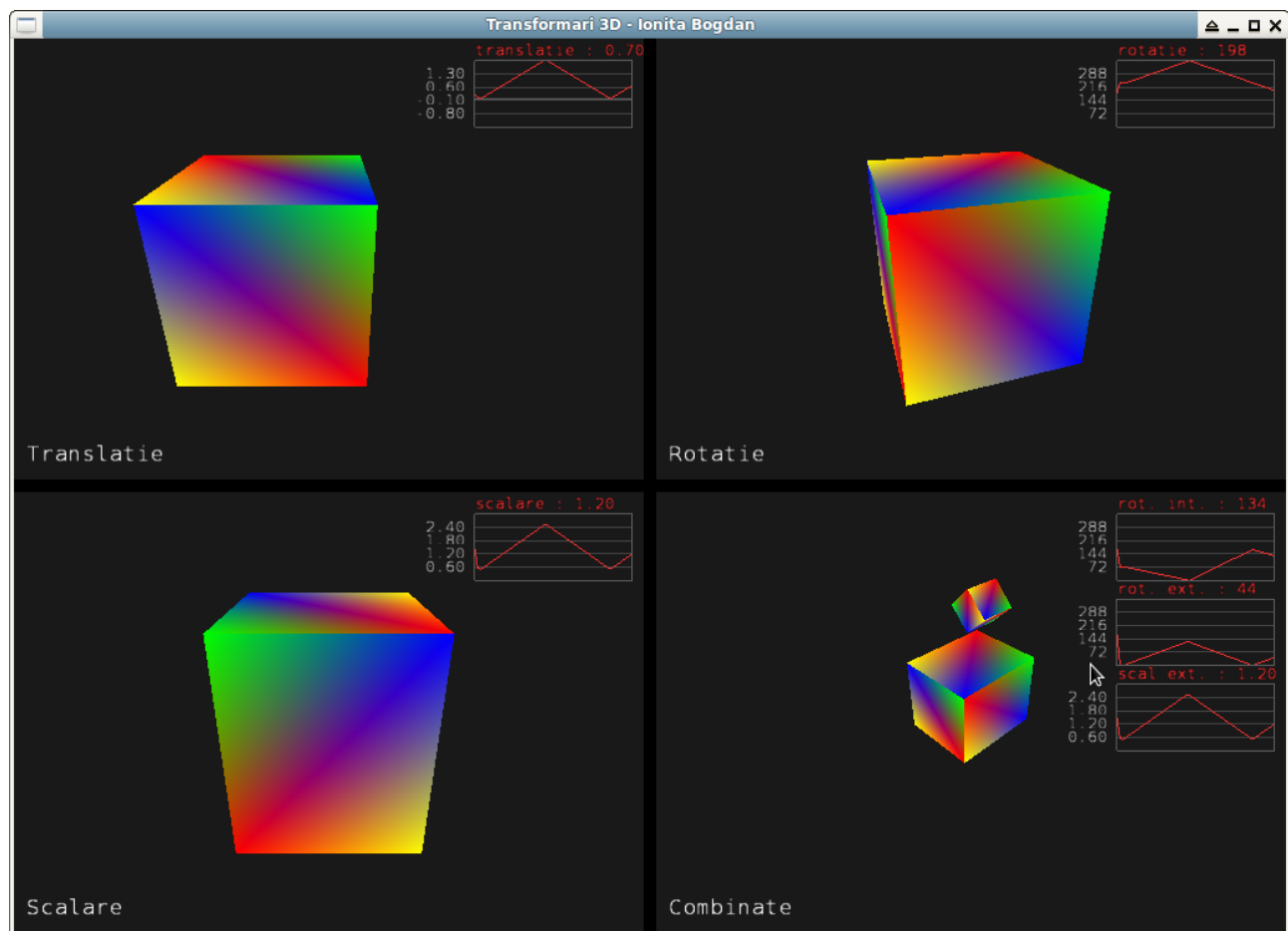
    if (lerpFactor_ > 0.9f) { // nearing the end
        // slow down
        float delta = maxLerpSpeed_ * 0.5f * dt;
        if (lerpSpeed_ > delta)
            lerpSpeed_ -= delta;
    } else if (lerpSpeed_ < maxLerpSpeed_) {
        // speed up
        lerpSpeed_ += maxLerpSpeed_ * 0.5f * dt;
    }
}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

```

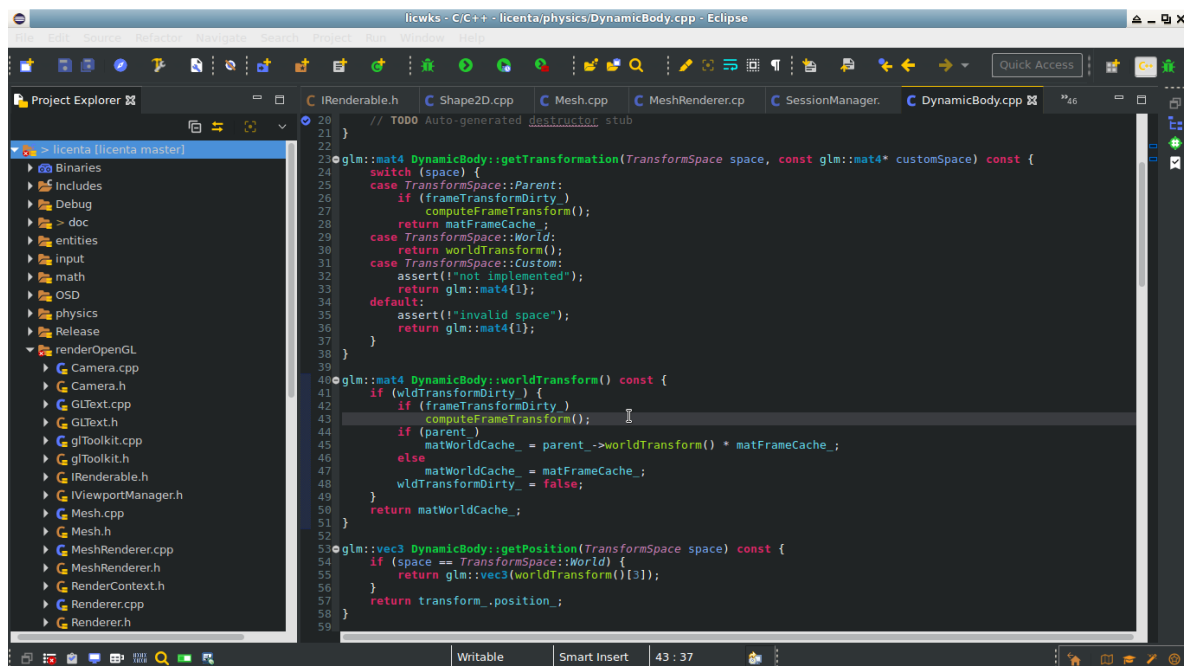
    }
    lerpFactor_ += dt; //lerpSpeed_ * dt;
    last_ = lerpFn_(origin_, path_[pathIndex_].value, clamp(lerpFactor_,
                                                             0.f, 1.f));
    if (lerpFactor_ >= 1) {
        pathIndex_++;
    }
}

```

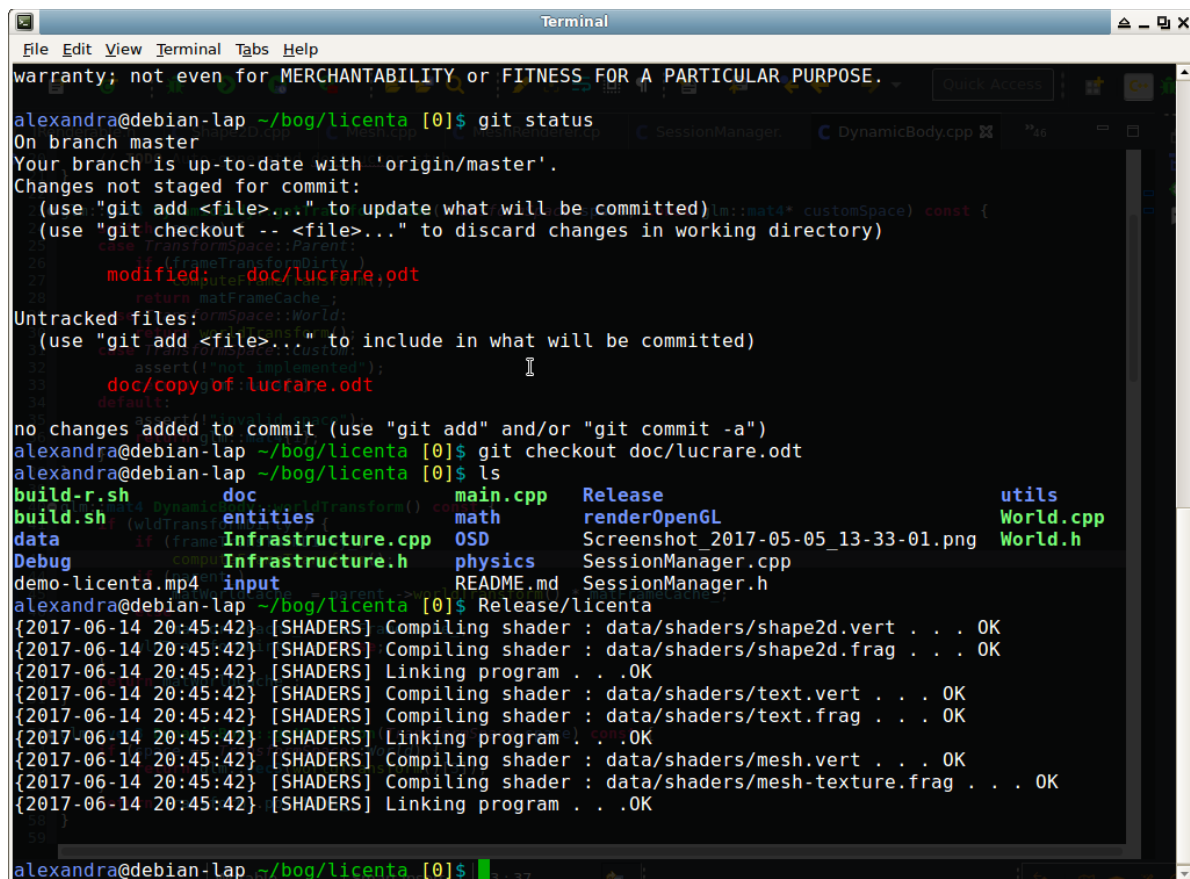
6.2 Screenshot ansamblu aplicatie



6.3 Screenshot mediu de dezvoltare



Exemplu de rulare aplicatie:



6.4 Secvența de cod care generează și configurează toate elementele din scenă

```

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(pcp4));
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>("Combinatie", 22,
ViewportCoord{10, 10, ViewportCoord::absolute,
ViewportCoord::left | ViewportCoord::bottom}, 0,
glm::vec3{1, 1, 1}, std::set<std::string>{"combined"}));
}

```