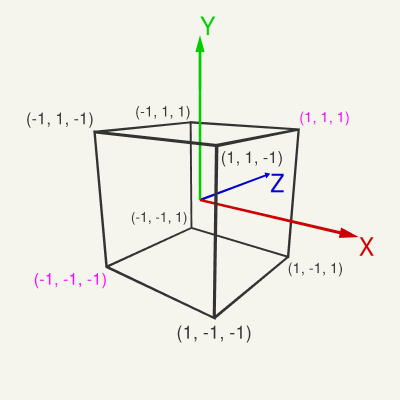
# Scena și pipeline-ul grafic

Pentru a afișa scena, avem nevoie de descrierea acesteia care constă în enumerarea modelelor și plasarea lor în spațiul scenei.

Spațiul OpenGL este un cub cu coordonatele puntelor constituente, în intervalul [-1,1]:



(Sursa: https://i.stack.imgur.com/iqFzD.png)

Observăm că este un spațiu "stângaci" (left handed), în sensul că pentru a determina sensul pozitiv al axei OZ, considerăm sensurile pozitive ale axelor OX și OY ca fiind indicate de degetul arătător, respectiv mijlociu al mâinii stângi; în acest caz, sensul pozitiv al axei OZ este indicat de degetul mare al mâinii stângi.

Un **model** (obiect în scenă) este definit prin structuri geometrice (triunghiuri, patrulatere (quad-uri), linii) care la rândul lor sunt definite prin vertecși. Vertecșii sunt vârfurile acestor forme geometrice simple (de exemplu, avem 3 vertecși pentru un triunghi).

Un **vertex** este o structură cu informații privitoare la:

* poziția vârfului (acesta e o informație obligatorie, deoarece un punct nu are sens în scenă fără coordonate asociate)
* culoarea vârfului (această culoare poate fi folosită în interpolare, ajutând la determinarea culorii punctelor intermediare din forma geometrică ce cuprinde vertexul).
* coordonate de textură (când vrem să preluăm culoarea vertexului dintr-o imagine-textură specificăm coordonatele punctului de culoare în cadrul imaginii)
* normala la suprafața din care face parte vertexul în coordonatele corespunzătoare vertexului.
* și multe alte diverse informații, de obicei utile pentru a calcula prin interpolare alte informații necesare în determinarea culorii fragmentelor scenei.

Informațiile vertecșilor sunt puse într-un buffer, care poate fi citit de placa grafică. Putem specifica modul de prelucrare a acestor informații prin *shadere*. Acestea sunt programe executate de placa grafică. Vertecșii sun prelucrați de un vertex shader, executatt în paralel pentru fiecare vertex, de obicei cu scopul de a calcula coordonatele în spațiul OpenGL. În urma executării acestuia, se generează fragmentele (punctele intermediare care umplu spațiul dintre vertecși) prin interpolare.

# Shader (shaders)

Sunt scrise în limbajul GLSL.

## Vertex shader

Este executat de placa grafică pentru fiecare vertex.

Este obligatoriu să setăm gl\_Position penru a se cunoaște locația (coordonatele) fiecărui vertex.

Programul de tip vertex shader defineste niste variabile globale speciale atât de intrare cât și de ieșire.

**Variabile de intrare:**

* **attribute**. Atributele sunt proprietățile vertecșilor. Fiecare vertex are propriile lui atribute precum coordonate, culoare, normală etc.
* **uniform**. Variabilele de tip uniform au aceeași valoare pentru toți vertecșii și toate fragmentele.

**Variabile de ieșire:**

* **varying**. Variabilele de tip varying sunt setate în vertex shader pentru a notifica placa grafică despre faptul că acestea trebuie interpolate. De exemplu dacă modul de desenare e cu GL\_TRIANGLES, pentru triunghiul curent la generarea fiecărui fragment (punct intern triunghiului) se va realiza o interpolare în triunghi pentru a calcula proprietatea cu aceeași nume a fragmentului. Practic din 3 varying de vertecși rezultă o mulțime de valori interpolate corespunzătoare tuturor punctelor din interiorul triunghiului.
* **gl\_Position** (un varying special, obligatoriu, care setează coordonatele, între -1 și 1, ale vertexului).

## Fragment shader

Este executat de placa grafică pentru fiecare fragment în parte.

În acest shader se decide culoarea fiecărui fragment prin setarea (obligatorie) a gl\_FragColor (de tip vec4). Cele 4 componente ale acestuia sunt canalele pentru rosu,verde, albastru (red, green, blue) și alpha (transparența). Valorile pentru cele 4 canale se setează în intervalul [0,1].

**Variabile de intrare:**

* **varying** . Variabilele varying din fragment shader trebuie să aibă același nume și tip cu variabilele varying din vertex shader. Nu putem avea o variabilă varying definită în fragment shader care să nu fie definită în vertex shader.
* **uniform**. Variabilele de tip uniform au aceeași valoare pentru toți vertecșii și toate fragmentele.

**Variabile de ieșire:**

* **gl\_FragColor**. Culoarea fragmentului.

# Fișierele framework-ului

## Proiectul NewTrainingFramework

### NewTrainingFramework.cpp

Fișierul NewTrainingFramework.cpp cuprinde programul principal (funcția main()) care afișează scena.

Funcțiile folosite sunt:

#### **int Init ( ESContext \*esContext )**

Această funcție inițializează scena.

Realizează următoarele acțiuni:

1. Setează culoarea de background (care va fi folosită pentru resetarea frame-ului) cu funcția glClearColor(r,g,b,a). Funcția primește 4 argumente de tip floati corespunzătoare celor 4 canale de culoare.
2. creează vectorul de vertecși de desenat (vertecșii fac parte din structura Vertex și deocamdată au ca proprietate doar poziția (Vector3) dată prin coordonatele x, y și z).
3. creează și deschide bufferul de vertecși (Vertex Buffer Object, VBO) de tip GL\_ARRAY\_BUFFER. În acesta se vor pune datele vertecșilor spre a fi trimiși către vertex shader, și apoi închide bufferul.
4. initializează shaderele și returnează 0 în caz de succes sau -1, -2 în caz de eșec

#### **void Draw ( ESContext \*esContext )**

Această funcție realizează desenarea unui cadru (frame) prin următoarele acțiuni:

Folosind funcția glClear, se șterge conținutul bufferului de culori (practic se șterge cadrul anterior). Funcția glClear este apelată cu parametrul GL\_COLOR\_BUFFER\_BIT astfel umplând tot bufferul de culoare cu culoarea de background setată în funcția Init (cu funcția glClearColor).

glClear(GL\_COLOR\_BUFFER\_BIT);

Specifică shaderul folosit cu glUseProgram

glUseProgram(myShaders.program);

Deschide bufferul de vertecși și trimite datele acestora în variabilele din shader.

Pentru a desena trebuie să trimitem datele vertecșilor către vertex shader. Vom folosi funcțiile glEnableVertexAttribArray și glVertexAttribPointer.

Funcția glEnableVertexAttribArray este utilizată pentru a activa un atribut de vertex, astfel încât datele asociate acestuia să fie folosite în timpul procesului de desenare. Fără activarea atributului, nu se vor accesa datele corespunzătoare din buffer!

Are sintaxa:

void glEnableVertexAttribArray(GLuint index);

Parametrul index reprezintă locația atributului de vertex pe care dorim să-l activăm. Acesta corespunde unui atribut definit în vertex shader (de exemplu, attribute vec3 a\_posL;) și este obținut prin funcția glGetAttribLocation (vedeți fișierul Shaders.cpp).

**Observație**: Dacă nu mai este necesar să folosim un atribut, îl putem dezactiva cu funcția glDisableVertexAttribArray. Aceasta poate fi utilă dacă reutilizăm pipeline-ul grafic pentru alte operații de desen și vrem să eliberăm unele resurse.

Exemplu:

glDisableVertexAttribArray(myShaders.positionAttribute);

Dar nu este necesar să faceți asta acum în programul vostru.

Funcția glVertexAttribPointer este utilizată pentru a specifica modul în care datele unui atribut de vertex (cum ar fi poziția, normalele, coordonatele texturii) sunt stocate într-un buffer și cum să fie accesate de pipeline-ul grafic în timpul desenării. Această funcție configurează conexiunea între un buffer de vertecși (Vertex Buffer Object, VBO) și un atribut de vertex din vertex shader.

Are sintaxa:

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

Descrierea parametrilor:

* index. Specifică locația atributului de vertex. Această locație este asociată cu variabilele din shaderul vertex prin funcția glGetAttribLocation (vedeți fișierul Shaders.cpp). Exemplu: Dacă avem atributul vec3 a\_posL în vertex shader și am obținut locația acestuia folosind

positionAttribute = glGetAttribLocation(program, "a\_posL")

atunci în index va trebui să folosim valoarea returnată de această funcție.

* size. Numărul de componente pe care atributul le are pentru fiecare vertex. Valori acceptate:
  + 1, pentru un atribut scalar (de exemplu, float).
  + 2, pentru un vector cu 2 componente (de exemplu, coordonate 2D).
  + 3, pentru un vector cu 3 componente (de exemplu, poziții 3D).
  + 4, pentru un vector cu 4 componente (de exemplu, culori RGBA).

Acest parametru trebuie să fie compatibil cu tipul declarat al atributului în shader (de exemplu, un atribut vec3 din shader trebuie să aibă size = 3).

* type. Specifică tipul de date al fiecărei componente. Valorile posibile includ:
  + GL\_FLOAT: Componentele sunt de tip float.
  + GL\_BYTE, GL\_UNSIGNED\_BYTE: Componentele sunt de tip byte.
  + GL\_SHORT, GL\_UNSIGNED\_SHORT: Componentele sunt de tip short.
* normalized. Specifică dacă valorile fixe (cum ar fi GL\_BYTE sau GL\_UNSIGNED\_BYTE) trebuie normalizate la intervalul [0, 1] (pentru valori pozitive) sau [-1, 1] (pentru valori semnate) înainte de a fi utilizate în shader.
  + GL\_TRUE: Normalizează valorile.
  + GL\_FALSE: Lasă valorile neschimbate.

Exemplu: Dacă stocăm culori ca valori întregi între 0-255 și dorim să le normalizăm automat la 0-1 în shader, setăm normalized la GL\_TRUE.

* stride. Distanta (în bytes) între începutul unui set de date pentru un vertex și începutul următorului set. Acest parametru permite OpenGL să știe cum să sară de la un vertex la altul în buffer. Dacă toate datele sunt stocate secvențial în buffer fără spațiu între ele, putem seta stride la 0, iar OpenGL va calcula automat distanța necesară.

Exemplu: Dacă fiecare vertex are doar poziții (vec3), stride = 0. Dacă avem și poziții și culori, iar fiecare vertex ocupă 32 bytes (12 bytes pentru poziții și 16 pentru culori, și 4 biți de padding), atunci stride = 32.

* pointer \*. Specifică offset-ul (în bytes) de la începutul bufferului până la datele efective ale acestui atribut. În contextul unui VBO bind-uit (folosind glBindBuffer(GL\_ARRAY\_BUFFER, ...)), acest parametru este tratat ca un offset. Dacă nu folosim un VBO, atunci este un pointer către datele efective din memoria RAM.

Exemplu: Dacă pozițiile vertexului încep de la primul byte al bufferului, pointer = 0. Dacă culorile încep după poziții, pointer = sizeof(Vector3) (dimensiunea unui Vector3).

În cod verificăm dacă locația atributului a\_posL (care va conține coordonatele vertecșilor) este diferită de -1 (adică există acest atribut, este folosit în shader), caz în care transmitem datele în el.

if(myShaders.positionAttribute != -1)

{

glEnableVertexAttribArray(myShaders.positionAttribute);

glVertexAttribPointer(myShaders.positionAttribute, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), 0);

}

Desenarea scenei pe baza vertecșilor se face cu funcția glDrawArrays.

Funcția glDrawArrays este folosită pentru desenarea primitivelor (triunghiuri, linii, puncte etc.) direct din datele stocate în buffer-ul de vârfuri (VBO). Este folosită pentru cazurile în care ordinea vertecșilor în buffer este directă, fără ca aceștia să fie reutilizați prin indici.

Funcția are sintaxa:

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

Descrierea parametrilor:

* mode. Specifică tipul primitivei geometrice care va fi desenată. Valorile posibile sunt:
  + GL\_POINTS: Desenează puncte individuale pentru fiecare vertex.
  + GL\_LINES: Desenează segmente, fiecare format din doi vertecși.
  + GL\_LINE\_STRIP: Desenează o linie frântă deschisă, formată din toți vertecșii, în ordine.
  + GL\_LINE\_LOOP: Similar cu GL\_LINE\_STRIP, dar desenează o linie frântă închisă, conectând ultimul vertex la primul.
  + GL\_TRIANGLES: Desenează triunghiuri independente, fiecare folosind 3 vertecși.
  + GL\_TRIANGLE\_STRIP: Desenează o bandă de triunghiuri conectate. Fiecare triunghi după primul folosește doi vertecșii din triunghiul anterior.
  + GL\_TRIANGLE\_FAN: Desenează un evantai de triunghiuri, folosind primul vertex drept centru.
* first. Specifică indexul primului vertex din buffer care va fi folosit pentru desen. Acest index este relativ la datele stocate în bufferul activ de vertecși.
* count. Specifică numărul total de vertecși care vor fi procesați pentru desen. Este important să ne asigurăm că numărul este suficient de mare pentru primitiva geometrică aleasă. Pentru GL\_TRIANGLES, count trebuie să fie multiplu de 3 (fiecare triunghi necesită 3 vertecși). Pentru GL\_LINES, count trebuie să fie multiplu de 2 (fiecare linie necesită 2 vertecși).

Cum dorim să desenăm un triunghi și avem un buffer cu 3 vertecși din care dorim să îi folosim pe toți (deci dorim să începem chiar de la primul vertex), apelăm:

glDrawArrays(GL\_TRIANGLES, 0, 3);

La final se închide bufferul de vertecși:

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

Deoarece folosește tehnica de *double buffering* interschimbă bufferul de desenare cu cel de afișare

#### **int \_tmain(int argc, \_TCHAR\* argv[])**

Această funcție este, de fapt, funcția main. Realizează următoarele:

1. initializează variabila esContext care contine datele despre fereastră și face inițializările bibliotecii OpenGL
2. creează fereastra aplicației
3. apelează funcția Init și in cazul în care aceasta eșuează, oprește porgramul
4. înregistrează funcțiile apelate pentru desenare, update și pentru tratarea evenimentelor de tastatură
5. porneste bucla "jocului" (a motorului grafic) cu funcția esMainLoop
6. după ce se termină programul, eliberează resursele apelând CleanUp()

#### **void Update ( ESContext \*esContext, float deltaTime )**

Această funcție se apelează la randarea fiecărui cadru. Va fi folosită pentru calculele necesare afișării cadrului. Vom evita să efectuăm calcule în Draw, lăsând acea funcție strict pentru operațiile de afișare.

#### **void Key ( ESContext \*esContext, unsigned char key, bool bIsPressed)**

Această funcție este apelată automat de către framework în urma evenimentelor de tastatură (la apăsarea sau eliberarea unei taste). Parametrii:

* esContext - variabila context
* key - tasta apăsată
* bIsPressed - are valoare true dacă funcția a fost apelată în urma unei apăsări de tastă și false dacă a fost apelată în urma unei eliberări de tastă

### Shaders.h Shaders.cpp

Clasa **Shaders** administrează shaderele încărcate (combinația de vertexShader și fragmentShader).

Proprietăți:

* vertexShader, fragmentShader - sunt id-urile celor două shadere în cazul în care acestea au fost compilate cu succes
* program - este id-ul obiectului rezultat prin legarea (linking) a celor două shadere (variabilele de tip varying din vertexshader care sunt de output, devin variabile de input în fragmentShader)
* fileVS, fileFS - căile la care se găsesc shaderele de tip vertex și fragment
* positionAttribute - locația la care se găsește variabila "a\_posL" în vertexShader. Această locație este folosită pentru a trimite date în shader (în această situație va fi folosită pentru coordonatele vertecșilor).

Metode:

* Init() - compilează si leagă shaderele, sau, în caz de eroare, afișează erorile în output.
* ~Shaders() - destructorul șterge obiectele de tip shader

## Proiectul Utilities

### Math.h, Math.cpp

Conține metode matematice utile pentru calcul vectorial și matricial.

### esUtil.h, esUtil.cpp

Conține clasa EsContext.

### esUtil\_win.h, esUtil\_win.cpp

Conține funcțiile responsabile cu crearea ferestrei și preluarea evenimentelor din ea.

# Taskuri

I) Interpolarea culorilor din vârfurile triunghiului (rosu, verde, albastru)

Pași:

1. Se definește în **Vertex.h**, după *pos*, încă o proprietate de tip Vector3, pentru culoare (numită *color*)
2. În funcția Init din **NewTrainingFramework.cpp** se setează valorile pentru culoarea fiecărui vertex ( de exemplu: rosu, verde, albastru)
3. În shaderul de tip vertex, **TriangleShaderVS.vs**, se creează un atribut de tip vec3, numit a\_color, pentru culoarea vertexului.
4. În **TriangleShaderVS.vs** se definește o variabilă de ieșire (**varying**) de tip vec3, numită v\_color. Pentru a realiza interpolarea culorilor aceasta va fi setată la valoarea lui a\_color
5. În **Shaders.h**, veți defini încă o proprietate de tip GLint, numită colorAttribute (care să conțină locația din shader a variabilei a\_color). În Shaders.cpp, folosind funcția glGetAttribLocation(program\_shader, nume\_variabila\_shader) veți prelua locația atributului a\_color
6. În funcția Draw din **NewTrainingFramework.cpp** veți verifica dacă locația atributului culoare (colorAttribute) este diferită de -1 (adică atributul e declarat în shader), caz în care veți trimite datele, ca și pentru poziție, schimbând însă offset-ul din structura Vertex (ultimul parametru al funcției glVertexAttribPointer (în loc să fie 0, va fi sizeof(Vector3), deoarece în cadrul structurii trebuie să sară peste un Vector3, corespunzător poziției, ca să ajungă la culoare. Atenție, ultimul parametru al funcției e de tip pointer (adresă) deci valoarea va trebui convertită la void\* (adică (void\*)(sizeof(Vector3)).
7. În shaderul de tip fragment, **TriangleShaderFS.fs**, se va defeni un varying cu același nume și tip precum cel din vertex shader (adică vec3 v\_color), și această valoare va fi folosită pentru a seta culoarea fragmentului (gl\_FragColor). Atenție, gl\_FragColor e vec4, deci va rebui realizată o conversie de la vec3 la vec4, setând canalul alpha (a 4-a coordonată pe 1)

gl\_FragColor=vec4(v\_color, 1.0);

II) Triunghiul să se rotească cu o viteză constantă față de OZ

Pași:

1. Deoarece e vorba de schimbarea locației figurii geometrice, vom lucra cu vertecșii. În vertex shader definim variabila uniform de tip mat4, numită u\_rotation. Aceasta va fi înmulțită (la stânga) cu poziția vertecșilor si rezultatul va fi pus în gl\_Position.
2. În **Shaders.h** definim propritatea matrixUniform de tip GLint în care vom prelua locația variabilei u\_rotation
3. În **Shaders.cpp**, preluăm locația lui u\_roation cu glGetUniformLocation(program\_shader, nume\_variabila\_shader)
4. În **NewTrainingFramework.cpp** definim o variabilă globală de tip float (initializată cu 0) numită *angle*, și o variabilă de tip float, numită *step*, inițializata cu 0.1 (sau ce valoare vreți voi - va determina viteza de rotație)
5. În funcția Update() vom crește variabila angle cu valoarea din step. Dacă angle depășește 2\*PI, se scade 2\*PI din el ca să prevenim ajungerea la valori foarte mari. Veți defini constanta PI în **NewTrainingFramework.cpp** (cu #define)
6. În funcția Draw() definim o matrice de tip Matrix, notată mRotation, și folosind funcția SetRotationZ() cu unghiul calculat, se creează matricea de rotație
7. În funcția Draw() verificați dacă matrixUniform e diferită de -1 (adică exista variabila u\_rotation). Dacă există, folosiți funcția glUniformMatrix4fv(locatie\_variabila, nr\_matrici, transpunere, valoare) pentru a trimite valoarea în variabila *u\_rotation*. Locația variabilei e *matrixUniform*, trimitem o matrice, nu dorim să fie transpusă deci folosim GL\_FALSE pentru al treilea parametru. Valoarea matricii e de forma (float\*) deci se va lua din proprietatea *m* a matricii *mRotation* (proprietatea *m* e de tip float[4][4], deci trebuie o conversie la float\*).