Info curs
- Elemente de Grafică pe Calculator
- Infographie

Cataloage EGC
- TBA

Laboratoare
- Laboratorul 01
- Laboratorul 02
- Laboratorul 03
- Laboratorul 04
- Laboratorul 05
- Laboratorul 06
- Laboratorul 07
- Prezentare Tema 1
- Laboratorul 08
- Laboratorul 09
- Vacanță
- Prezentare Tema 2
- Recuperări laborator
- Prezentare Tema 3
- Resurse: Redare text

Teme
- Regulament General
- Tema 1 - Bow and Arrow
- Tema 2 - Skyroads
- Tema 3 - Stylised Runner

Resurse
- Resurse Utile
- Notare

Table of Contents

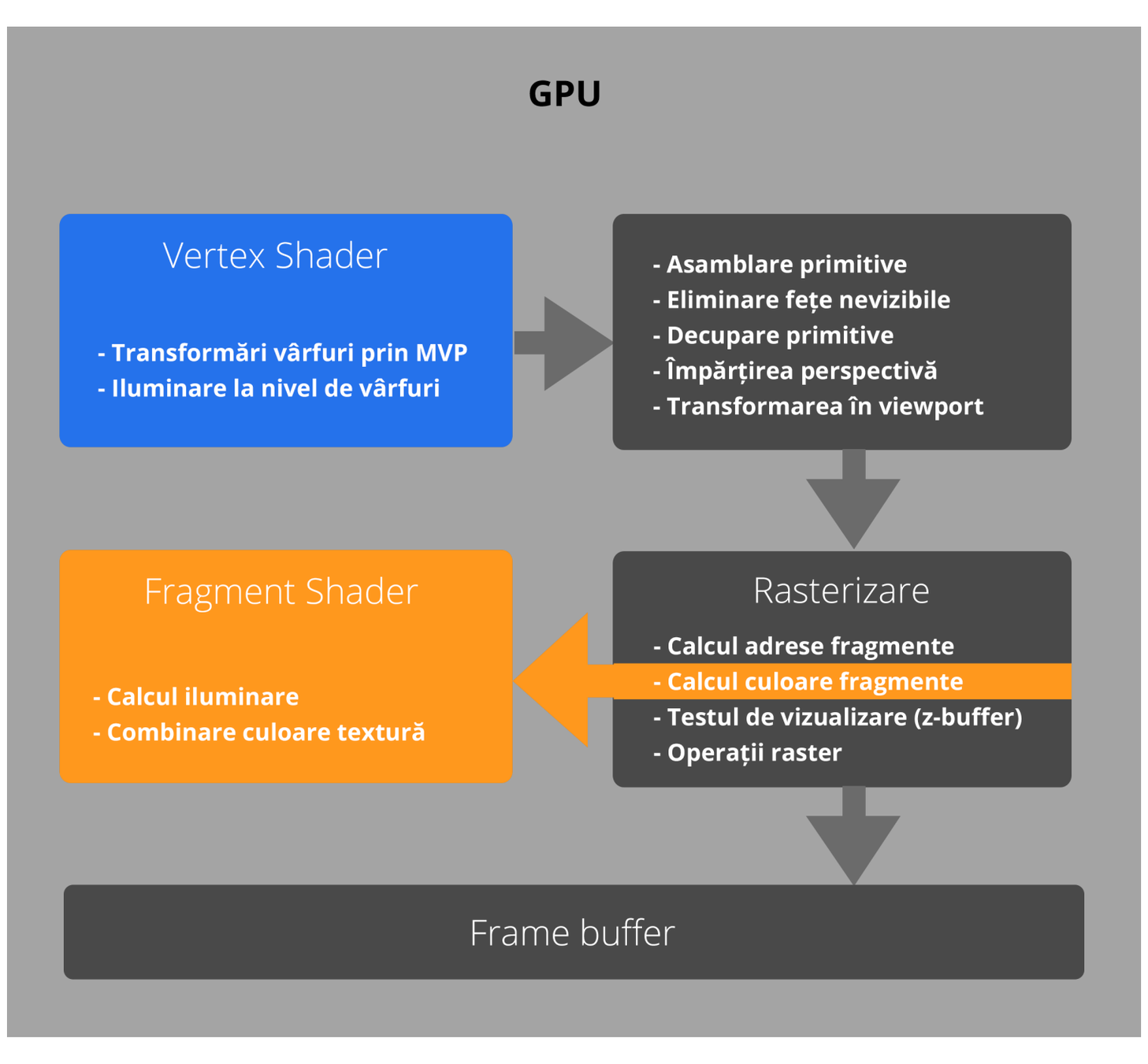
Laboratorul 06

Video Laborator 6: https://youtu.be/f7q2TGCRly0
Autor: Anca Băluțoiu

Banda Grafica

Banda Grafica este un lant de operatii executate de procesoarele GPU. Unele dintre aceste operatii sunt descrise in programe numite **shadere** (eng. ***shaders***), care sunt scrise de programator si transmise la GPU pentru a fi executate de procesoarele acestuia. Pentru a le deosebi de alte operatii executate in banda grafica, pe care programatorul nu le poate modifica, **shaderele** sunt numite „etape programabile”. Ele dau o mare flexibilitate in crearea de imagini statice sau dinamice cu efecte complexe redate in timp real (de ex. generarea de apa, nori, foc etc prin functii matematice).

Folosind OpenGL sunt transmise la **GPU**: coordonatele varfurilor, matricile de transformare a varfurilor (M: modelare, V: vizualizare, P: proiectie, MV: modelare-vizualizare, MVP: modelare-vizualizare-proiectie), topologia primitivelor, texturi si ale date.



1. In **etapa programabila VERTEX SHADER** se transforma coordonatele unui varf, folosind matricea MVP, din coordonate obiect in coordonate de decupare (eng. *clip coordinates*). De asemenea, pot fi efectuate si calcule de iluminare la nivel de varf. Programul VERTEX SHADER este executat in paralel pentru un numar foarte mare de varfuri.

2. Urmeaza o **etapa fixa**, in care sunt efectuate urmatoarele operatii:

- asamblarea primitivelor folosind varfurile transformate in vertex shader si topologia primitivelor;
- eliminarea fetelor nevizibile;
- decuparea primitivelor la frontiera volumului canonic de vizualizare (ce inseamna?);
- impartirea perspectiva, prin care se calculeaza coordonatele dispozitiv normalizate ale varfurilor: xd = xc/w; yd = yc/w;zd = zc/w, unde [xc,yc,zc,w] reprezinta coordonatele unui varf in sistemul coordonatelor de decupare;
- transformarea fereastra–poarta: din fereastra (-1, -1) – (1, 1) in viewport-ul definit de programator.

3. Urmatoarea etapa este **Rasterizarea**. Aceasta include:

- calculul adreselor pixelilor in care se afiseaza fragmentele primitivelor (bucatele de primitive de dimensiune egala cu a unui pixel);
- calculul culorii fiecarui fragment, pentru care este apelat programul **FRAGMENT SHADER**
- in etapa programabila **FRAGMENT SHADER** se calculeaza culoarea unui fragment pe baza geometriei si a texturilor; programul **FRAGMENT SHADER** este executat in paralel pentru un numar mare de fragmente.
- testul de vizibilitate la nivel de fragment (algoritmul z-buffer);
- operatii raster, de exemplu pentru combinarea culorii fragmentului cu aceea existenta pentru pixelul in care se afiseaza fragmentul.

Rezultatul etapei de rasterizare este o **imagine** memorata intr-un tablou de pixeli ce va fi afisat pe ecran, numit ^^*frame buffer*^^.

> Incepand cu a cincea generatie de procesoare video integrate si OpenGL 3.x, intre etapele 2 si 3 exista inca o etapa programabila, numita ***Geometry shader***.

Shader OpenGL

Pentru implementarea de programe SHADER in OpenGL se foloseste limbajul dedicat GLSL (GL Shading Language).

Legarea unui shader la programul care foloseste OpenGL este o operatie complicata, de aceea va este oferit codul prin care se incarca un shader.

Un **VERTEX SHADER** e un program care se executa pentru **FIECARE** vertex trimis catre banda grafica. Rezultatul transformarilor, care reprezinta coordonata post-proiectie a vertexului procesat, trebuie scris in variabila standard **gl_Position** care e folosita apoi de banda grafica. Un vertex shader are tot timpul o functie numita main. Un exemplu de vertex shader:

```
#version 330

layout(location = 0) in vec3 v_position;

// Uniform properties
uniform mat4 Model;
uniform mat4 View;
uniform mat4 Projection;

void main()
{
    gl_Position = Projection * View * Model * vec4(v_position, 1.0);
}
```

Un **FRAGMENT SHADER** e un program ce este executat pentru **FIECARE** fragment generat in urma operatiei de rasterizare (ce inseamna?). Fragment shader are in mod obligatoriu o functie numita main. Un exemplu de fragment shader:

```
#version 330

layout(location = 0) out vec4 out_color;

void main()
{
    out_color = vec4(1, 0, 0, 0);
}
```

Cum legam un obiect geometric la shader?

Legarea intre obiecte (mesh, linii etc.) si shadere se face prin atribute. Datorita multelor versiuni de OpenGL exista multe metode prin care se poate face aceasta legare. In laborator vom invata metoda specifica OpenGL 3.3 si OpenGL 4.1. Metodele mai vechi nu mai sunt utilizate decat in atunci cand hardware-ul utilizat impune restrictii de API.

API-ul OpenGL modern (3.3+) utilizeaza metoda de legare bazata pe layout-uri. In aceasta metoda se folosesc pipe-uri ce leaga un atribut din OpenGL de un nume de atribut in shader.

```
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(VertexFormat), (void*)0);
```

Prima comanda seteaza pipe-ul cu numarul 2 ca fiind utilizat. A doua comanda descrie structura datelor in cadrul VBO-ului astfel:

- pe pipe-ul **2** se trimit la shader 3 float-uri (argument 3) pe care nu le normalizam (argument 4)
- argumentul 5 numit si **stride**, identifica pasul de citire (in bytes) in cadrul VBO-ului pentru a obtine urmatorul atribut; cu alte cuvinte, din cati in cati octeti sarim cand vrem sa gasim un nou grup de cate 3 float-uri care reprezinta acelasi lucru
- argumentul 6 identifica offsetul inital din cadrul buffer-ul legat la GL_ARRAY_BUFFER (VBO); cu alte cuvinte, de unde plecam prima oara.

In Vertex Shader vom primi atributul respectiv pe pipe-ul cu indexul specificat la legare, astfel:

```
layout(location = 2) in vec3 vertex_attribute_name;
```

Mai multe informatii se pot gasi pe pagina de documentatie Vertex Shader attribute index.

> Pentru mai multe detalii puteti accesa:
> - API-ul de OpenGL aici: https://www.opengl.org/sdk/docs/man/
> - API-ul pentru GLSL aici: https://www.opengl.org/sdk/docs/manglsl/

> Un articol despre istoria complicata a OpenGL si competitia cu Direct3D/DirectX poate fi citit aici.

Cum trimitem date generale la un shader?

La un shader putem trimite date de la CPU prin variabile uniforme. Se numesc uniforme pentru ca nu variaza pe durata executiei shader-ului. Ca sa putem trimite date la o variabila din shader trebuie sa obtinem locatia variabilei in programul shader cu functia glGetUniformLocation:

```
int location = glGetUniformLocation(int shader_program, "uniform_variable_name_in_shader");
```

- **shader_program** reprezinta ID-ul programului shader compilat pe placa video
- in cadrul framework-ului de laborator ID-ul se poate obtine apeland functia `shader→GetProgramID()` sau direct accesand vriabila membru `shader→program`

Apoi, dupa ce avem locatia (care reprezinta un offset/pointer) putem trimite la acest pointer informatie cu functii de tipul glUniform:

```
//void glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value)
glm::mat4 matrix(1.0f);
glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(matrix));

// void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3)
glUniform4f(location, 1, 0.5f, 0.3f, 0);

//void glUniform3i(GLint location, GLint v0, GLint v1, GLint v2)
glUniform3i(location, 1, 2, 3);

//void glUniform3fv(GLint location, GLsizei count, const GLfloat *value)
glm::vec3 color = glm::vec3(1.0f, 0.5f, 0.8f);
glUniform3fv(location, 1, glm::value_ptr(color));
```

Functiile **glUniform** sunt de forma **glUniform[Matrix?]NT[v?]** (regex) unde:

- Matrix - in cazul in care e prezent identifica o matrice
- N - reprezinta numarul de variabile de tipul **T** ce vor fi trimise:
  - **1, 2, 3, 4** in cazul tipurilor simple
  - pentru matrici mai exista si **2×3, 2×4, 3×2, 3×4, 4×2, 4×3**
- T - reprezinta tipul variabilelor trimise
  - **ui** - unsigned int
  - **i** - int
  - **f** - float
- v - datele sunt specificate printr-un vector, se da adresa de memorie a primei valori din vector

Comunicarea intre shadere-le OpenGL

In general pipeline-ul programat este alcatuit din mai multe programe shader. In cadrul cursului de EGC vom utiliza doar Vertex Shader si Fragment Shader. OpenGL ofera posibilitatea de a comunica date intre programele shader consecutive prin intermendiul atributelor **in** si **out**

> In metoda specifica OpenGL 3.3 numele de atribut **attribute_name** trebuie sa fie acelasi atat in Vertex Shader cat si in Fragment Shader pentru a se stie legatura intre input/output.

Vertex Shader:

```
#version 330  // GLSL version of shader (GLSL 330 means OpenGL 3.3 API)

out vec3 attribute_name;
```

Fragment Shader:

```
in vec3 attribute_name;
```

> In caz ca avem support pentru GLSL 410 (OpenGL 4.1) se poate specifica si locatia attributului astfel, caz in care doar locatiile vor fi folosite pentru a lega iesirea unui Vertex Shader de intrarea la Fragment Shader si nu numele atributului.
> Mai multe detalii se pot obtine de la: Program separation linkage

Vertex Shader:

```
#version 410  // GLSL 410 (OpenGL 4.1 API)

layout(location = 0) out vec4 vertex_out_attribute_name;
```

Fragment Shader:

```
#version 410

layout(location = 0) in vec4 fragment_in_attribute_name;
```

Cerinte laborator

> tasta **F5** - reincarca shaderele in timpul rularii aplicatiei. Nu este nevoie sa opriti aplicatia intrucat shaderele sunt compilate si rulate de catre placa video si nu au legatura cu codul sursa C++ propriu zis.

1. Descarcati framework-ul de laborator
2. Completati functia `RenderSimpleMesh` astfel inca sa trimiteti corect valorile uniform catre Shader
   - Se interogeaza locatia uniformelor “Model”, “View” si “Projection”
   - Folosind `glUniformMatrix4fv` sa se trimita matricile corespunzatoare catre shader
   - Daca ati completat corect functia, si ati completat gl_Position in vertex shader, ar trebui sa vedeti un cub pe centrul ecranului rottit 45 grade in jurul lui Y si colorat variat
3. Completati Vertex Shaderul
   a. Se de clara atributele de intrare pentru `Vertex Shader` folosind layout location
   ```
   layout(location = 0) in vec3 v_position;
   // same for the rest of the attributes ( check Lab6.cpp CreateMesh() );
   ```
   b. Se declara atributele de iesire catre `Fragment Shader`
   ```
   out vec3 frag_color;
   // same for other attributes
   ```
   c. Se salveza valorile de iesire in `main()`
   ```
   frag_color = vertex_color;
   // same for other attributes
   ```
   d. Se calculeaza pozitia in clip space a vertexului primit folosind matricile Model, View, Projection
   ```
   gl_Position = Projection * View * Model * vec4(v_position, 1.0);
   ```
4. Completati Fragment Shaderul
   - Se primesc valorile atributelor trimise de la `Vertex Shader`
   - Valoarea de intrare ale fiecarui atribut e calculata prin interpolare liniara intre vertexii ce formeaza patch-ul definit la desenare (triunghi, linie)
   ```
   in vec3 frag_color;
   ```
   - Se calculeaza valoarea fragmentului (pixelului) de output
   ```
   out_color = vec4(frag_color, 1);
   ```
5. Sa se utilizeze normala vertexilor pe post de culoare de output in cadrul Fragment Shader-ului
   - Inspectati de asemenea structura `VertexFormat` pentru a intelege ceea ce se trimite pe fiecare pipe
6. Sa se interschimbe **pipe-ul 1** cu **pipe-ul 3**. Trimiteti normala pe **pipe-ul 3** si culoarea vertexului pe **pipe-ul 1**
   - Se inspecteaza rezultatul obtinut
7. Bonus: sa se trimita timpul aplicatiei (Engine::GetElapsedTime()), si sa se varieze pozitia si culoarea (unul sau mai multe canale de culoare) dupa o functie de timp (trigonometrica etc.)