Laborator 3

1 Obiective

Obiectivul acestui laborator este de a prezenta o scurtă introducere în principiile **pipeline-ului programabil OpneGL**.

2 Fundament teoretic

OpenGL este considerată o interfață de programare a aplicațiilor (API), utilizată pentru a dezvolta aplicații grafice accesând funcțiile disponibile în hardware-ul grafic. Cu toate acestea, OpenGL este o specificație dezvoltată și întreținută de Grupul Khronos. Specificația OpenGL descrie care este rezultatul dorit sau ieșirea fiecărei funcții. Implementarea reală poate fi diferită între diferitele biblioteci OpenGL. Aceste biblioteci sunt implementate în principal de producătorii de plăci grafice.

Primele versiuni ale OpenGL au folosit un așa-numit **pipeline cu funcții fixe**, în care cea mai mare parte a funcționalității era predefinită, iar dezvoltatorii puteau modifica doar câțiva parametri, dar nu puteau să schimbe algoritmii actuali folosiți pentru a transforma obiecte 3D în imagini 2D. Acest pipeline este foarte ușor de înțeles, dar pe de altă parte este foarte ineficient. Începând cu OpenGL 3.2, acest pipeline cu funcții fixe a devenit depreciat, dezvoltatorii preferând în schimb gradul mai mare de flexibilitate oferit de **pipeline-ul programabil**.

Există câteva modificări în scrierea aplicațiilor OpenGL cu pipeline programabil, în comparație cu versiunea cu pipeline fix:

- *Nu mai este posibilă trasarea în mod imediat* – glBegin() și glEnd() pentru a specifica primitivele grafice;
- *Toate datele trebuie stocate în obiecte de tip buffer* – cum ar fi Vertex Buffer Objects (VBO) sau Vertex Array Objects (VAO);
- *Fără fucții pentru transformări* – glTranslatef(), glRotatef(), glScalef(), gluLookAt(), gluPerspective() nu mai sunt suportate;
- Fără funcții pentru iluminare – glLight*() și glMaterial*() nu mai sunt suportate și trebuie implementate în programele *shader*;

OpenGL funcționează ca un automat de stări finite, constând dintr-o colecție de variabile care definesc modul în care automatul ar trebui să funcționeze. Contextul OpenGL reprezintă starea curentă. Adesea, înainte de rasterizare folosind contextul actual, modificăm starea manipulând câțiva parametri și și unele buffer-e. Fiind independent de sistemul de operare sau de sistemul de trasare a ferestrelor utilizat, OpenGL oferă flexibilitate pentru dezvoltarea aplicațiilor cross-platform. Prin utilizarea unor programe mici (denumite **shader-e**) putem folosi GPU-urile la capacitatea lor maximă, comparativ cu implementarea cu pipeline fix. Shaderele sunt atașate unei aplicații OpenGL și rulează pe GPU. Acestea sunt scrise în OpenGL Shading Language (GLSL), care este foarte similar cu limbajul C.

Pipeline-ul de rasterizare implementat în OpenGL este ilustrat în figura de mai jos. Pipeline-ul programabil conține cel puțin stagiile de **vertex shading** și **fragment shading**, și înlocuiește pipeline-ul fix, care nu mai este suportat (iluminare și transformări). Faza de **vertex shading** procesează fiecare vârf independent de celelalte; datele vârfurilor sunt primite de la aplicație prin obiecte de tip vertex buffer (VBO). Faza de rasterizare generează un set de fragmente care sunt procesate în faza de **fragment shading**; ieșirea este reprezentată de culoarea și adâncimea fragmentului (coordonata Z). Unele etape (cum ar fi *primitive setup*, *clipping* și *rasterization*) nu sunt programabile. *Tesselation Shader* și *Geometry Shader* sunt opționale.

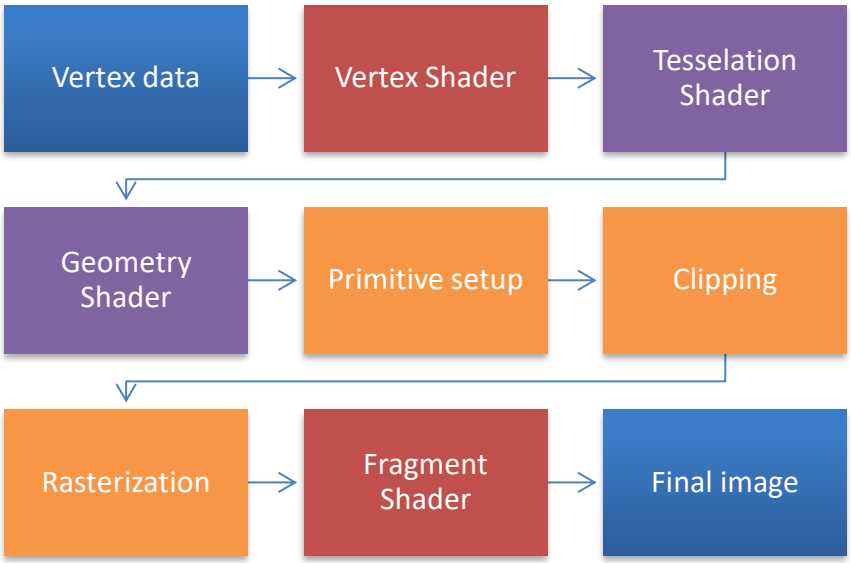

Figure 1 – Pipeline-ul programabil OpenGL

3 O aplicație simplă OpenGL 4

3.1 Biblioteci

Folosim biblioteca **GLFW** în loc de GLUT pentru a crea o fereastră cu contexte OpenGL. GLFW este scris în C și are suport nativ pentru Windows, OS X și multe sisteme asemănătoare de tip Unix. Puteți descărca binare precompilate sau codul sursă de la http://www.glfw.org. Un tutorial pas cu pas privind compilarea bibliotecii pe diferite platforme poate fi găsit aici.

Biblioteca OpenGL Extension Wrangler (GLEW) este o bibliotecă open-source, cross-platform, de încărcare a extensiilor C/C++. Puteți descărca binare precompilate sau codul sursă de la http://glew.sourceforge.net/. GLEW oferă mecanisme de rulare eficiente pentru a determina ce extensii OpenGL sunt acceptate pe platforma țintă.

3.2 Reprezentarea obiectelor 3D

Un obiect 3D este reprezentat folosind informații topologice ale vârfurilor. Pentru fiecare vârf avem nevoie de atribute cum ar fi poziția, culoarea, coordonatele de textură și alte informații relevante. Datele de vârf sunt stocate în obiecte Vertex Buffer (VBO). O serie de VBO pot fi stocate într-un obiect Array

Vertex. (VAO).
Un **obiect Vertex Buffer (VBO)** este un buffer folosit pentru a păstra o serie de atribute de vârf (cum ar fi poziția, culoarea, vectorul normală etc.). Un **obiect Vertex Array (VAO)** grupează mai multe VBO-uri. Pentru a utiliza VBO trebuie să:

- generăm nume pentru VBO (ID-uri): *glGenBuffers(…)*
- selectăm un anumit VBO pentru inițializare: *glBindBuffer(GL_ARRAY_BUFFER, …)*
- încărcăm date în VBO: *glBufferData(GL_ARRAY_BUFFER, ...)*

O procedură similară este necesară pentru obiecte Vertex Array:

- generăm nume pentru VAO (ID-uri): *glGenVertexArrays(…)*
- selectăm un anumit VAO pentru inițializare: *glBindVertexArray(…)*
- reîmprospătăm VBO-urile associate cu VAO-ul curent: *glBindBuffer(…)* și *glVertexAttribPointer(…)*
- selectăm VAO-ul folosit pentru rasterizare: *glDrawArrays(…)*

3.3 Vertex și Fragment Shader

Pentru a putea rasteriza obiecte 3D, trebuie să scriem cel puțin un vertex shader și fragment shader. Pentru aceasta avem nevoie să:

- generăm un obiect shader (identificabil printr-un ID unic): *glCreateShader(…)*
- atașăm codul sursă al shader-ului la obiectul de tip shader: glShaderSource(…)
- compilăm shader-ul: glCompileShader(…)

După ce am compilat programele vertex shader și fragment shader, trebuie să le combinăm într-un așa-numit obiect de tip **program shader**. Pașii pentru aceasta sunt după cum urmează:

- creăm un program shader: *glCreateProgram(…)*
- atasăm shader-ele compilate anterior: *glAttachShader(…)*
- operația de linking a shader-elor: *glLinkProgram(…)*

4 Tutorial

Această secțiune prezintă o aplicație de bază OpenGL 4. Codul sursă inițial este disponibil pe pagina web a laboratorului.

4.1 Pasul 1

Folosiți aplicația obținuta in laboratorul anterior si adăugați resursele disponibile pe pagina web a laboratorului in directorul proiectului, iar apoi fișierul sursa *main.cpp* folosind optiune *Add exiting item*.

4.2 Pasul 2

Începem prin definirea unui triunghi și, pentru aceasta, trebuie să definim coordonatele a trei noduri, o variabilă pentru a stoca ID-ul obiectului Vertex Buffer și altul pentru a stoca ID-ul obiectului Vertex Array.

```
//coordonatele varfurilor in systemul de coordinate normalizate
GLfloat vertexCoordinates[] = {
    0.0f,   0.5f,       0.0f,
```

```
    0.5f, -0.5f,      0.0f,
    -0.5f, -0.5f,     0.0f
};

GLuint verticesVBO;
GLuint triangleVAO;
```

Adăugați următoarea funcție, care va inițializa obiectele OpenGL.

```
void initObjects()
{
    //genereaza un ID unic pentru verticesVBO
    glGenBuffers(1, &verticesVBO);
    //asociaza buffer-ul verticesVBO variabilei OpenGL GL_ARRAY_BUFFER,
    //orice referire ulterioara la GL_ARRAY_BUFFER va configura buffer-ul asociat momentan,
    //care este verticesVBO
    glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);
    //copiaza datele in buffer-ul current asociat – specificat prin intermediul primului argument
    //tipul buffer-ului – al doilea argument specifica dimensiunea (in Bytes) datelor
    //al treilea argument reprezinta datele pe care vrem sa le trimitem
    //al patrulea argument specifica modul in care vor fi tratate datele de catre placa video
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexCoordinates), vertexCoordinates, GL_STATIC_DRAW);

    //genereaza un ID unic, care corespunde  obiectului triangleVAO
    glGenVertexArrays(1, &triangleVAO);
    glBindVertexArray(triangleVAO);
    glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);
    //seteaza pointer-ul atributelor de varf
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
    glEnableVertexAttribArray(0);
    //de-selecteaza obiectul triangleVAO
    glBindVertexArray(0);
}
```

4.3 Pasul 3

Programele vertex și fragment shader sunt deja incluse în proiect. Codul sursă este prezentat mai jos. Nu vă faceți griji cu privire la conținutul lor în acest stadiu. În următoarea lucrare de laborator, vom analiza conținutul și structura acestor programe shader.

4.3.1 Vertex shader

```
#version 400

layout(location = 0) in vec3 vertex_position;

out vec3 colour;

void main() {
    //definim culoarea varfului
    colour = vec3(1.0, 0.0, 0.0);
    gl_Position = vec4(vertex_position, 1.0);
}
```

4.3.2 Fragment shader

```
#version 400

in vec3 colour;
out vec4 frag_colour;

void main() {
		frag_colour = vec4 (colour, 1.0);
}
```

4.4 Pasul 4

În continuare, trebuie să citim, să compilam și să conectăm programele shader. Următoarea funcție este deja prezentă în codul sursă; nu este necesar să o adăugați.

```
GLuint initBasicShader(std::string vertexShaderFileName, std::string fragmentShaderFileName)
{
   //citeste, parseaza si compileaza vertex shader
   std::string v = readShaderFile(vertexShaderFileName);
   const GLchar* vertexShaderString = v.c_str();
   GLuint vertexShader;
   vertexShader = glCreateShader(GL_VERTEX_SHADER);
   glShaderSource(vertexShader, 1, &vertexShaderString, NULL);
   glCompileShader(vertexShader);
   //verifica statusul operatiei de compilare
   shaderCompileLog(vertexShader);

   //citeste, parseaza si compileaza fragment shader
   std::string f = readShaderFile(fragmentShaderFileName);
   const GLchar* fragmentShaderString = f.c_str();
   GLuint fragmentShader;
   fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
   glShaderSource(fragmentShader, 1, &fragmentShaderString, NULL);
   glCompileShader(fragmentShader);
   //verifica statusul operatiei de compilare
   shaderCompileLog(fragmentShader);

   //ataseaza si conecteaza programul shader
   shaderProgram = glCreateProgram();
   glAttachShader(shaderProgram, vertexShader);
   glAttachShader(shaderProgram, fragmentShader);
   glLinkProgram(shaderProgram);
   glDeleteShader(vertexShader);
   glDeleteShader(fragmentShader);
   //verifica informatiile de linking
   shaderLinkLog(shaderProgram);

   return shaderProgram;
}
```

4.5 Pasul 5

În cele din urmă, trebuie să specificăm ce primitive ar trebui rasterizate de către OpenGL. În acest scop, folosim funcția glDrawArrays, care utilizează shaderul activ, VAO și VBO-urile active pentru a rasteriza primitiva specificată.

```
void renderScene()
{
    //initializeaza buffer-ele de culoare si adancime inainte de a rasteriza cadrul curent
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //defineste culoarea de fundal
    glClearColor(0.8, 0.8, 0.8, 1.0);
    //specifica locatia si dimensiunea ferestrei
    glViewport (0, 0, retina_width, retina_height);

    //proceseaza evenimentele de la tastatura
    if (glfwGetKey(glWindow, GLFW_KEY_A)) {
        //TODO
    }

    if (glfwGetKey(glWindow, GLFW_KEY_D)) {
        //TODO
    }

    //activeaza program shader-ul; apeluri ulterioare de rasterizare vor utiliza acest program
    glUseProgram(shaderProgram);

    //activeaza  VAO
    glBindVertexArray(triangleVAO);
    //specifica tipul primitiei, indicele de inceput si numarul de indici utilizati pentru rasterizare
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

5 Lectură suplimentară

- OpenGL Programming Guide 8th edition – capitolul 1

Funcție (și link)	Descriere
glGenBuffers	Generează nume de obiecte buffer
glBindBuffer	Asociază un obiect buffer punctului de asociere specificat
glBufferData	Creează și inițializează setul de date al unui obiect buffer
glGenVertexArrays	Generează nume pentru obiecte vertex array object
glBindVertexArray	Asociază un obiect vertex array
glVertexAttribPointer	Definește un vector cu date generice ale vârfurilor
glEnableVertexAttribArray	Activează un vertex array cu date generice
glCreateShader	Creează un obiect shader
glShaderSource	Înlocuiește codul sursă al unui obiect shader
glCompileShader	Compilează un obiect shader
glCreateProgram	Creează un obiect program
glAttachShader	Atașează un obiect shader unui obiect proram

glLinkProgram	Conectează(linking) un obiect program
glDeleteShader	Șterge un obiect shader
glUseProgram	Instalează un obiect program ca parte a stării de rasterizare actuale
glDrawArrays	Rasterizează primitive din vectorul de date

6 Temă

1. Definiți și afișați două triunghiuri. Definiți diferite obiecte buffer pentru ele (VBO și VAO) și aranjați-le astfel încât să formeze un pătrat.
2. Definiți un nou program shader care folosește un shader de fragment diferit. Setați culoarea de rasterizare verde pentru acest shader de fragment nou. Afișați un triunghi folosind programul shader vechi, iar celălalt utilizând programul shader nou definit.
3. Schimbați programul shader folosit pentru a rasteriza fiecare triunghi utilizând diferite taste.