Laborator 3

1 Obiective

Obiectivul acestui laborator este de a prezenta diferite metode în care datele vârfurilor (atribute) sunt transferate între nivelele aplicației (CPU) și GPU.

2 Fundament teoretic

2.1 Shader OpenGL

Pipeline-ul programabil OpenGL se bazează pe shader-e, care sunt programe de mici dimensiuni scrise în limbajul GLSL (foarte asemănător cu C și orientat spre operații grafice). Acest limbaj conține funcții utile legate de operațiile vectoriale și de matrice. Fiecare shader începe cu o declarație de versiune, urmată de o listă de variabile de intrare și ieșire, variabile **uniform** și cel puțin funcția principală (care este punctul de intrare). Structura tipică a unui shader este următoarea:

```
//Versiune OpenGL
#version versionNumber

//lista variabilelor de intrare
in type inputVariableName;

//lista variabilelor de iesire
out type outputVariableName;

//lista variabilelor de tip uniform
uniform type uniformName;

void main()
{
   //implementeaza functionalitatea si se ocupa de variabilele de iesire
   outputVariableName = processVariable();
}
```

2.2 Tipuri de date

În GLSL, putem folosi majoritatea tipurilor implicite de date de bază, cum ar fi int, float, double, uint și bool. Pe lângă aceste tipuri de bază, GLSL suportă și vectori și matrice. Un vector reprezintă un container pentru oricare dintre tipurile de bază și poate avea dimensiuni diferite - de 1,2,3 sau 4 elemente. Putem defini vectori de genul asta (de obicei vom folosi **vecn**):

- **vecn**: vector de n numere reale (float)
- **bvecn**: vector de n boolean
- **ivecn**: vector de n întregi
- **uvecn**: vector de n întregi fără semn
- **dvecn**: vector de n numere reale (double)

Pentru a accesa diferite componente din vectori, vom folosi **.x, .y, .z** și **.w** (prima, a doua, a treia și a patra componentă). Pentru lizibilitatea codului, putem folosi **.r, .g, .b** și **.a** pentru culori - sau **.s, .t, .p** și **.q** pentru coordonatele de textură.

2.3 Intrări și ieșiri

Intrările și ieșirile sunt specificate folosind cuvintele cheie **in** și **out**.

Dacă numele unei variabile de intrare (dintr-un shader) se potrivește cu numele unei variabile de ieșire (de la un alt shader), atunci datele trec de la un shader la altul.

Intrările vertex shader reprezintă atributele vârfului (ar putea fi poziția, culoarea, vectorul normal etc.). Pentru a se potrivi atributele de vârf specificate la nivelul CPU cu cele definite la nivelul GPU, vom folosi următoarea declarație: **layout (location = locationID).**

La nivelul procesorului, folosim următoarele funcții:

```
glVertexAttribPointer(locationID, otherParameters);
glEnableVertexAttribArray(locationID);
```

La nivelul GPU, specificăm variabila de intrare:

```
layout(location = locationID) in vec3 vertex_position;
```

Ieșirea fragment shader ar trebui să fie o variabilă de ieșire vec4, reprezentând culoarea. În caz contrar, rezultatul (imaginea) va fi incorect.

2.4 Variabile Uniform

Un alt mod de a transfera datele de la aplicația noastră (care rulează pe CPU) la shader-e (care rulează pe GPU) este prin variabile **uniform**. Acestea sunt variabile globale (adică unice pentru un obiect program shader) și pot fi accesate de orice shader al programului. În plus, valoarea lor va rămâne aceeași până când sunt modificate.

Pentru a declara o variabilă uniform la nivelul shader-ului, folosim:

```
uniform type uniformName;
```

Pentru a actualiza variabila uniform, avem nevoie (la nivel de aplicație) de locația variabilei și apoi putem schimba valoarea sa.

```
glUseProgram(shaderProgram);
GLint uniformLocation = glGetUniformLocation(shaderProgram, "uniformName");
glUniform3f(uniformLocation, 0.15f, 0.0f, 0.84);
```

Pentru a actualiza valoarea variabilei uniform, folosim funcția **glUniform*,** care are mai multe definiții (diferite de postfix):

- **f**: așteaptă un float
- **i**: așteaptă un int

- **ui**: așteaptă un unsigned int
- **3f**: așteaptă un 3 valori float
- **fv**: așteaptă un vector float

2.5 Atribute de vârf

În VBO putem stoca mai multe atribute decât poziția vârfurilor. De exemplu, putem stoca culoarea, vectorul normală, coordonatele texturii etc. Luați în considerare următorul exemplu, în care avem două atribute pentru fiecare vârf (poziție și culoare).

Vertex 1						Vertex 2					
X	Y	Z	R	G	B	X	Y	Z	R	G	B
0...3	4...7	8...11	12...15	16...19	20...23	24...27	28...31	32...35	36...39	40...43	44...47
Vertex position stride: 24 **Vertex offset: 0**											
			Vertex color stride: 24 **Vertex color offset: 12**								

Trebuie să modificăm vertex shader pentru a primi valorile culorilor și, pentru aceasta, trebuie să setăm locația corespunzătoare:

```
layout(location = locationID1) in vec3 vertexPosition;
layout(location = locationID2) in vec3 vertexColor;
```

Trebuie să schimbăm și modul în care specificăm atributele de vârfuri din programul nostru:

```
//vertex position attribute
glVertexAttribPointer(locationID1, size, type, normalized, stride, pointer);
glEnableVertexAttribArray(locationID1);

//vertex colour attribute
glVertexAttribPointer(locationID2, size, type, normalized, stride, pointer);
glEnableVertexAttribArray(locationID2);
```

- *size* – număr de componente(1, 2, 3, 4)
- *type* – tipul de date pentru fiecare componentă (GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, and GL_UNSIGNED_INT)
- *normalized* - specifică dacă valorile datelor trebuie să fie normalizate sau nu
- *stride* - decalajul de octeți între atributele vertex generice consecutive
- *pointer* - decalajul primei componente a primului atribut generic de vârf

2.6 Obiecte Element Buffer

Pentru a evita duplicarea (redefinirea) unor noduri, putem folosi EBO-urile pentru a stoca indicii. Indicii vor fi utilizați de OpenGL pentru a decide ce vârfuri să aleagă atunci când formează poligoane. Procedura de creare a EBO este foarte asemănătoare cu cea utilizată pentru definirea VBO:

```
glGenBuffers(1, &verticesEBO);
```

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, verticesEBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(vertexIndices), vertexIndices, GL_STATIC_DRAW);
```

Când apelăm glDrawElements, desenăm folosind indicii furnizați în cadrul EBO activat în prezent. Primul parametru specifică tipul primitivei, al doilea reprezintă numărul elementelor pe care le trasăm (vârfuri), al treilea parametru reprezintă tipul de indici (de obicei GL_UNSIGNED_INT), iar ultimul este un offset.

```
glDrawElements(GL_TRIANGLES, size, type, offset);
```

3 Tutorial

3.1 Transferul datelor din vertex în fragment shader

Începeți de la soluția pe care o găsiți pe site și modificați vertex and fragment shader. Amintiți-vă că dacă numele unei variabile de intrare (de la un shader) se potrivește cu numele unei variabile de ieșire (de la un alt shader), atunci datele trec de la un shader la celălalt.

Codul din **vertex shader**:

```
#version 400

layout(location = 0) in vec3 vertexPosition;

//variabila culoare pe care o trimitem in fragment shader
out vec3 colour;

void main() {
    colour = vec3(0.74, 0.16, 0.0);
    gl_Position = vec4(vertexPosition, 1.0);
}
```

Codul din **fragment shader**:

```
#version 400

//variabila culoare primita din vertex shader
in vec3 colour;

out vec4 fragmentColour;

void main() {
    fragmentColour = vec4(colour, 1.0);

}
```

3.2 Trimiterea datelor folosind variabile uniform

Începeți de la soluția pe care o găsiți pe site și modificați vertex și fragment shader.

Vertex shader:

```
#version 400
```

```
layout(location = 0) in vec3 vertexPosition;

void main() {
   gl_Position = vec4(vertexPosition, 1.0);
}
```

Fragment shader:

```
#version 400

out vec4 fragmentColour;

uniform vec3 uniformColour;

void main() {
   fragmentColour = vec4(uniformColour, 1.0);
}
```

Modificați funcția principală pentru a obține locația variabilei uniform din programul shader și actualizați valoarea:

```
int main(int argc, const char * argv[]) {

   .. .. ..

   myCustomShader.useShaderProgram();
   GLint uniformColourLocation = glGetUniformLocation(myCustomShader.shaderProgram, "uniformColour");
   glUniform3f(uniformColourLocation, 0.15f, 0.0f, 0.84);

   while (!glfwWindowShouldClose (glWindow)) {
      .. .. ..

   return 0;
}
```

3.3 Exemplu cu obiect Element Buffer

Începeți de la soluția pe care o găsiți pe site. În acest exemplu, dorim să desenăm un dreptunghi compus din două triunghiuri. În loc de a duplica unele coordonate ale nodurilor, vom folosi un Element Buffer Object pentru a economisi spațiu de stocare. Definiți, global, datele de vârf (reprezentând poziția și culoarea) și buffer-ele care vor fi utilizate pentru aceasta.

```
GLfloat vertexData[] = {
   //vertex position and vertex color
   -0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f,
   -0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f,
   0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f,
   0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 1.0f
};

GLuint vertexIndices[] = {
   0, 1, 2,
```

```
    0, 2, 3
};

GLuint verticesVBO;
GLuint verticesEBO;
GLuint objectVAO;
```

Trebuie să actualizăm inițializarea obiectului nostru. Diferența este că, în acest caz, avem și indici și un nou atribut pentru fiecare vârf. Pentru detalii despre parametrii necesari fiecărei funcții, citiți din nou secțiunea "Fundal teoretic" și secțiunea "Lectură suplimentară".

```
void initObjects()
{
    glGenVertexArrays(1, &objectVAO);
    glBindVertexArray(objectVAO);

        glGenBuffers(1, &verticesVBO);
        glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);

        glGenBuffers(1, &verticesEBO);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, verticesEBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(vertexIndices), vertexIndices, GL_STATIC_DRAW);

        //vertex position attribute
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
        glEnableVertexAttribArray(0);

        //vertex colour attribute
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(float)));
        glEnableVertexAttribArray(1);

    glBindVertexArray(0);
}
```

În funcția renderScene(), trebuie să actualizăm apelul de rasterizare la glDrawElements.

```
glBindVertexArray(objectVAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Vertex shader:

```
#version 400

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColour;

out vec3 colour;

void main() {
    colour = vertexColour;
    gl_Position = vec4(vertexPosition, 1.0);
}
```

Fragment shader:

```
#version 400

in vec3 colour;

out vec4 fragmentColour;

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

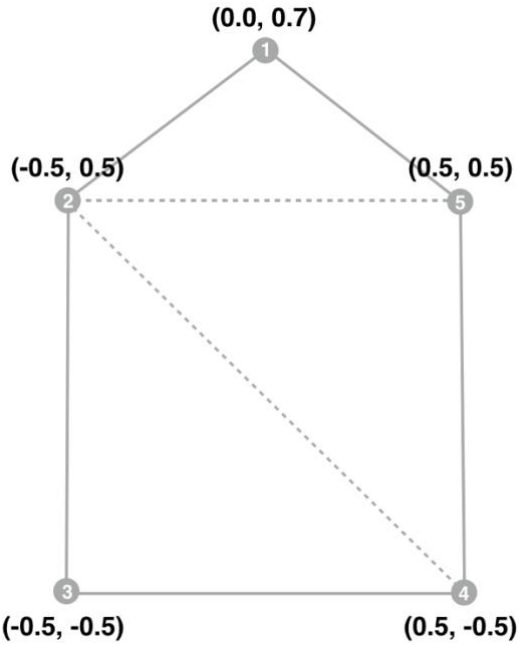
4 Lectură suplimentară

- OpenGL Programming Guide 8th edition – capitolele 2 și 3

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
glGetUniformLocation	Returnează locația unei variabile uniform
glUniform	Specifică valoarea unei variabile uniform pentru shader program-ul curent
glDrawElements	Rasterizează primitive din vectorul de date

5 Temă

1. Refaceți următoarea figură folosind EBO și definiți culori diferite pentru fiecare vârf. Nu replicați nodurile!!!



2. Modificați coordonatele vârfurilor (pe axa x și y) utilizând două variabile uniform care sunt trimise către vertex shader. Actualizați valorile uniformelor utilizând diferite intrări de la tastatură.
3. Comutați între reprezentările plane și wireframe utilizând diferite intrări de la tastatură.
    - Pentru a schimba rasterizarea într-o reprezentare wireframe, puteți utiliza următoarea funcție:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

    - Pentru a schimba rasterizarea într-o reprezentare plată, aveți următoarea funcție:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```