

# Laborator 1

---

## 1 Obiective

Obiectivul acestui laborator este de a descrie pe scurt operațiile matematice de bază folosite în domeniul Graficii Computerizate și modul în care poate fi folosită biblioteca OpenGL Mathematics (GLM) pentru a implementa aceste operații într-o aplicație OpenGL.

## 2 Biblioteca GLM

OpenGL Mathematics (GLM) este o bibliotecă de matematică C++ (compusă doar din fișiere antet – „header”) pentru software-ul grafic bazat pe specificațiile OpenGL Shading Language (GLSL). De altfel, această bibliotecă este utilă în orice context de dezvoltare software care necesită operații matematice simplu de utilizat. Pentru a include și utiliza biblioteca în aplicațiile noastre, trebuie să includeti doar fișierele antet corecte, fără nici o legătură suplimentară sau compilare. Ultima versiune a GLM poate fi descărcată de pe site-ul dedicat: <http://glm.g-truc.net>.

GLM este scris în C++ 98 dar poate beneficia de C++ 11 atunci când este suportat de compilator. Este o bibliotecă independentă de platformă fără alte dependințe și este suportată oficial următoarele compilatoare:

- Apple Clang 4.0 sau mai recent
- GCC 4.2 sau mai recent
- Intel C++ Composer XE 2013 sau mai recent
- LLVM 3.0 sau mai recent
- Visual C++ 2010 sau mai recent
- CUDA 4.0 sau mai recent (experimental)
- Orice compilator care se conformează standardelor C++98 sau C++11

Pentru a utiliza GLM, trebuie doar să includeți fișierul **glm.hpp** în aplicațiile voastre, cu condiția ca directorul rădăcină al GLM să fie în calea specificată prin directiva “Include”.

```
#include <glm/glm.hpp>
```

Caracteristicile principale GLM pot fi incluse folosind anteturi individuale pentru a permite compilarea mai rapidă a programelor.

```
<glm/vec2.hpp>: vec2, bvec2, dvec2, ivec2 and uvec2  
<glm/vec3.hpp>: vec3, bvec3, dvec3, ivec3 and uvec3  
<glm/vec4.hpp>: vec4, bvec4, dvec4, ivec4 and uvec4  
<glm/mat2x2.hpp>: mat2, dmat2
```

```
<glm/mat2x3.hpp>: mat2x3, dmat2x3
<glm/mat2x4.hpp>: mat2x4, dmat2x4
<glm/mat3x2.hpp>: mat3x2, dmat3x2
<glm/mat3x3.hpp>: mat3, dmat3
<glm/mat3x4.hpp>: mat3x4, dmat2
<glm/mat4x2.hpp>: mat4x2, dmat4x2
<glm/mat4x3.hpp>: mat4x3, dmat4x3
<glm/mat4x4.hpp>: mat4, dmat4
<glm/common.hpp>: all the GLSL common functions
<glm/exponential.hpp>: all the GLSL exponential functions
<glm/geometry.hpp>: all the GLSL geometry functions
<glm/integer.hpp>: all the GLSL integer functions
<glm/matrix.hpp>: all the GLSL matrix functions
<glm/packing.hpp>: all the GLSL packing functions
<glm/trigonometric.hpp>: all the GLSL trigonometric functions
<glm/vector_relational.hpp>: all the GLSL vector relational functions
```

Cele mai multe funcționalități GLM pe care le vom solicita pot fi accesate prin includerea următoarelor trei anteturi:

```
#include <glm/glm.hpp> // functionalitate de baza GLM
#include <glm/gtc/matrix_transform.hpp> // extensie GLM pentru generarea matricelor comune de transformare
#include <glm/gtc/type_ptr.hpp> // extensie GLM pentru interacțiunea dintre pointeri și matrice sau vectori
```

### 3 Operații cu Vectori

Pentru a utiliza vectorii în GLM, trebuie să includeți anteturile corespunzătoare:

```
#include <glm/glm.hpp> // include toate anteturile GLM
```

sau să includeți numai anteturile corespunzătoare vectorilor (optimizat):

```
#include <glm/vec2.hpp>
#include <glm/vec3.hpp>
#include <glm/vec4.hpp>
```

### 3.1 Declararea unui vector GLM

Acum putem declara o variabilă vectorială, de exemplu un vector **coloană** cu 4 elemente:

```
glm::vec4 newVector(1.0f, 2.0f, 3.0f, 1.0f);
```

Vectorii cu 2 sau 3 elemente sunt declarați într-un mod similar, folosind tipurile de date **vec2** și **vec3**.

### 3.2 Operații aritmetice

GLM suportă toate operațiile vectoriale prin supraîncărcarea (overloading) operatorului:

```
glm::vec4 newVector1(1.0f, 2.0f, 3.0f, 1.0f);
glm::vec4 newVector2(2.0f, 1.0f, 3.0f, 1.0f);
glm::vec4 newVector3;

newVector3 = 2.0f + newVector1; // adunare sau scadere cu un scalar
newVector3 = 2.0f * newVector1; // inmulțire cu un scalar
newVector3 = newVector1 + newVector2; //adunare sau scadere cu vectori
newVector3 = -newVector1; //negatia vectorului
```

### 3.3 Modulul unui vector și înmulțirea dintre vectori

Modulul (lungimea) unui vector poate fi obținut prin utilizarea funcției **glm::length()**. Produsul scalar și produsul vectorial pot fi calculate folosind **glm::dot()** și **glm::cross()**. Pentru normalizarea vectorului putem folosi **glm::normalize()**.

```
glm::vec4 newVector1(1.0f, 2.0f, 3.0f, 1.0f);
glm::vec4 newVector2(2.0f, 1.0f, 3.0f, 1.0f);

float length = glm::length(newVector1); //modulul unui vector
float dotProduct = glm::dot(newVector1, newVector2); //produs scalar
glm::vec3 newVector4 = glm::cross(glm::vec3(1.0f, 2.0f, 3.0f), glm::vec3(3.0f, 2.0f, 1.0)); //produs vectorial
glm::vec3 normalizedVector = glm::normalize(newVector1); //normalizeaza newVector1 si salveaza rezultatul intr-o
alta variabila
```

Notă: Funcția **length()** membră a claselor **vec\*** (de exemplu, **newVector1.length ()**) returnează numărul elementelor din vector (2,3 sau 4). Utilizați întotdeauna **glm::length()** pentru a calcula magnitudinea (modulul) unui vector.

## 4 Matrice și transformări

Pentru a utiliza matricele GLM, trebuie să includeți anteturile corespunzătoare GLM:

```
#include <glm/glm.hpp> // include toate anteturile GLM
```

Sau să includeți doar anteturile matricelor (optimizat)

```
#include <glm/mat2x2.hpp>
#include <glm/mat2x3.hpp>
#include <glm/mat2x4.hpp>
#include <glm/mat3x2.hpp>
#include <glm/mat3x3.hpp>
#include <glm/mat3x4.hpp>
#include <glm/mat4x2.hpp>
#include <glm/mat4x3.hpp>
#include <glm/mat4x4.hpp>
```

### 4.1 Declararea unei matrice GLM

Acum putem declara o variabilă matrice, de exemplu o matrice de  $2 \times 3$  **coloană-major**:

```
glm::mat2x3 newMatrix(1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f);
```

Alte tipuri de matrice pot fi definite într-un mod similar folosind celealte tipuri de date tip matrice pe care le oferă GLM. Matricele pătratice pot fi de asemenea definite folosind tipurile de date **mat2**, **mat3** și **mat4**. Acestea pot fi inițializate cu aceeași valoare pe diagonală folosind constructorii cu un singur parametru. De exemplu, o matrice identitate cu dimensiunea  $4 \times 4$  poate fi declarată ca:

```
glm::mat4 identity4Matrix(1.0f);
```

Constructorul implicit pentru toate tipurile de matrice GLM creează o matrice cu elemente de 1 pe diagonala principală și 0 pe toate celelalte poziții.

### 4.2 Operații aritmetice cu matrice

GLM suportă toate operațiile cu matrice prin supraîncărcarea (overloading) operatorului:

```
glm::mat4 newMatrix(1.0f);
glm::mat4 newMatrix(2.0f);
glm::mat4 newMatrix3;
```

```
newMatrix3 = 2.0f + newMatrix1; // adunare sau scadere cu un scalar  
newMatrix3 = 2.0f * newMatrix1; // inmultire cu un scalar  
newMatrix3 = newMatrix1 + newMatrix2; // adunare sau scadere cu matrice  
newMatrix3 = -newMatrix1; // negatia matricei  
newMatrix3 = newMatrix1 * newMatrix2; //inmultirea matricelor
```

### 4.3 Înmulțirea matrice-vector

Deoarece GLM construiește matricele în format **coloană-major**, o matrice și un vector pot fi înmulțite numai sub forma **M \* v**, unde **M** este matricea și **v** este vectorul. Operația inversă, **v \* M**, nu va genera eroare de compilare, cu toate acestea, rezultatul va fi incorrect.

```
glm::vec4 newVector1(1.0f, 2.0f, 3.0f, 1.0f);  
glm::mat4 newMatrix1(1.0f);  
glm::vec4 newVector3;  
  
newVector3 = newMatrix1 * newVector1; //inmultire matrice-vector
```

### 4.4 Matrice de transformare

GLM oferă o extensie pentru generarea celor mai utilizate matrice de transformare. Pentru a le folosi, trebuie să includeți anteturile corespunzătoare GLM:

```
#include <glm/gtc/matrix_transform.hpp> // extensie GLM pentru generarea matricelor de transformare
```

Acum putem genera matrice de transformare utilizând convenții standard OpenGL cu funcții fixe.

```
glm::mat4 oldTransformationMatrix;  
glm::mat4 newTransformationMatrix;  
  
newTransformationMatrix = glm::translate(oldTransformationMatrix, glm::vec3(1.0f, 2.0f, 3.0f)); // genereaza o matrice  
de translatie cu (1.0f, 2.0f, 3.0f)  
newTransformationMatrix = glm::scale(oldTransformationMatrix, glm::vec3(2.0f, 1.0f, 3.0f)); // genereaza o matrice de  
scalare cu (2.0f, 1.0f, 3.0f)  
newTransformationMatrix = glm::rotate(oldTransformationMatrix, glm::radians(45.0f), glm::vec3(0.0f, 1.0f, 0.0f)); //  
genereaza o matrice de rotatie cu 45.0 de grade in jurul punctului (0.0f, 1.0f, 0.0f) pe axa (Y axis)  
newTransformationMatrix = glm::lookAt(glm::vec3(0.0f, 0.0f, 7.0f), glm::vec3(0.0f, 0.0f, -10.0f), glm::vec3(0.0f, 1.0f,  
0.0f)); // genereaza o matrice de camera (pozitia camerei, punct de referinta, vector „sus”)
```

```
newTransformationMatrix = glm::perspective(45.0f, (GLfloat)screen_width / (GLfloat)screen_height, 1.0f, 1000.0f);  
//genereaza o matrice perspectiva (vertical fov, aspect ratio, zNear, zFar)
```

Funcțiile care generează transformări de model (translație, scalare, rotație) primesc un parametru suplimentar, reprezentând o matrice de transformare veche care va fi utilizată pentru a genera noua matrice de transformare după regula **newTransformationMatrix = oldTransformationMatrix \* <matricea de transformare generată>**. Acest lucru permite o compoziție simplă a transformărilor. De exemplu, pentru a genera o matrice care conține translația, scalarea și rotația de mai sus (în această ordine), putem scrie:

```
glm::mat4 translationScaleRotation(1.0f); //incepem cu matricea identitate  
  
translationScaleRotation = glm::rotate(translationScaleRotation, glm::radians(45.0f), glm::vec3(0.0f, 1.0f, 0.0f)); //  
genereaza o matrice de rotație cu 45.0 de grade în jurul punctului (0.0f, 1.0f, 0.0f) pe axa (Y axis)  
translationScaleRotation = glm::scale(translationScaleRotation, glm::vec3(2.0f, 1.0f, 3.0f)); // genereaza o matrice de  
scalare cu (2.0f, 1.0f, 3.0f)  
translationScaleRotation = glm::translate(translationScaleRotation, glm::vec3(1.0f, 2.0f, 3.0f)); // genereaza o matrice  
de translație cu (1.0f, 2.0f, 3.0f)
```

Acest cod va genera o matrice de **translațieScalareRotație** egală cu **Identitate \* rotație \* scalare \* translație**. Amintiți-vă că atunci când compuneți transformările, prima transformare care urmează să fie aplicată unui punct este ultima care urmează să fie inclusă în matricea de transformare compusă.

Pentru a aplica o transformare unui punct, trebuie să înmulțim matricea de transformare cu punctul:

```
glm::mat4 translationScaleRotation(1.0f); // incepem cu matricea identitate  
glm::vec4 originalPoint(1.0f, 2.0f, 3.0f, 1.0f);  
glm::vec4 transformedPoint;  
  
translationScaleRotation = glm::rotate(translationScaleRotation, glm::radians(45.0f), glm::vec3(0.0f, 1.0f, 0.0f));  
///genereaza o matrice de rotație cu 45.0 de grade în jurul punctului (0.0f, 1.0f, 0.0f) pe axa (Y axis)  
translationScaleRotation = glm::scale(translationScaleRotation, glm::vec3(2.0f, 1.0f, 3.0f)); // genereaza o matrice de  
scalare cu (2.0f, 1.0f, 3.0f)  
translationScaleRotation = glm::translate(translationScaleRotation, glm::vec3(1.0f, 2.0f, 3.0f)); // genereaza o matrice  
de translație cu (1.0f, 2.0f, 3.0f)
```

```
transformedPoint = translationScaleRotation * originalPoint; //transforma punctul original si salveaza punctul  
transformat intr-o noua variabila
```

## 5 Lectură suplimentară

- Documentație GLM – <http://glm.g-truc.net/>

## 6 Temă

1. Descărcați biblioteca GLM.
2. Creați un nou proiect Microsoft Visual Studio (C++ Win32 console application) și importați biblioteca GLM.
3. Includeți în acest proiect fișierele de pe pagina web a laboratorului (moodle).
4. Implementați o funcție numită **TransformPoint** care translatează un punct 3D în coordonate omogene cu (2.0f, 0.0f, 1.0f) și apoi îl rotește cu 90 de grade în jurul axei X. Funcția ar trebui să primească punctul ca parametru și ar trebui să returneze punctul transformat.
5. Implementați o funcție numită **ComputeAngle** care calculează și returnează unghiul dintre doi vectori v1 și v2 trimiși ca parametri.
6. Implementați o funcție numită **IsConvex** care testează dacă un poligon 2D este convex sau concav. Poligonul va fi definit prin vârfurile sale. Funcția va returna adevărat dacă poligonul este convex și fals dacă poligonul este concav.
7. Implementați o funcție numită **ComputeNormals** care calculează și returnează normalele normalize ale unui poligon convex 2D (Figura 1). Poligonul va fi definit prin vârfurile sale.

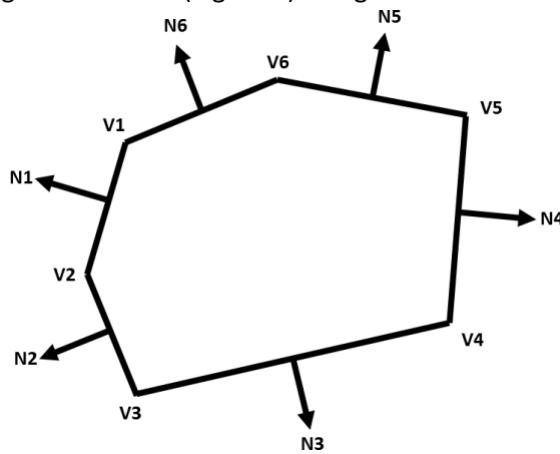


Figura 1 – Exemplu de poligon

# Tutorial – Microsoft Windows

## Crearea unui proiect Microsoft Visual Studio pentru o aplicație OpenGL/GLFW

Acest tutorial are ca scop familiarizarea cu mediul de dezvoltare Microsoft Visual Studio și cu pașii necesari creării unui proiect nou pentru o aplicație OpenGL care utilizează biblioteca GLFW pentru gestionarea ferestrei și a evenimentelor generate de dispozitivele de intrare (mouse, tastatură).

### 1. Biblioteca GLFW

Pentru a putea dezvolta o aplicație folosind biblioteca GLFW este nevoie să se obțină fișierele header (extensia .h) ale bibliotecii și fișierele binare (extensia .lib) ale acesteia.

**Metoda recomandată pentru a obține fișierele binare este de a compila întreaga bibliotecă din surse. Doar compilarea din surse asigură compatibilitate maximă între bibliotecă și platforma pe care se vor dezvolta aplicațiile.** Alternativ, binarele precompilate pentru diferite platforme predefinite pot fi descărcate de pe website-ul <https://www.glfw.org/>.

#### 1.1. Compilarea bibliotecii din surse

- Sursele bibliotecii GLFW pot fi descărcate de pe website-ul <https://www.glfw.org/download.html> (Figura 1)

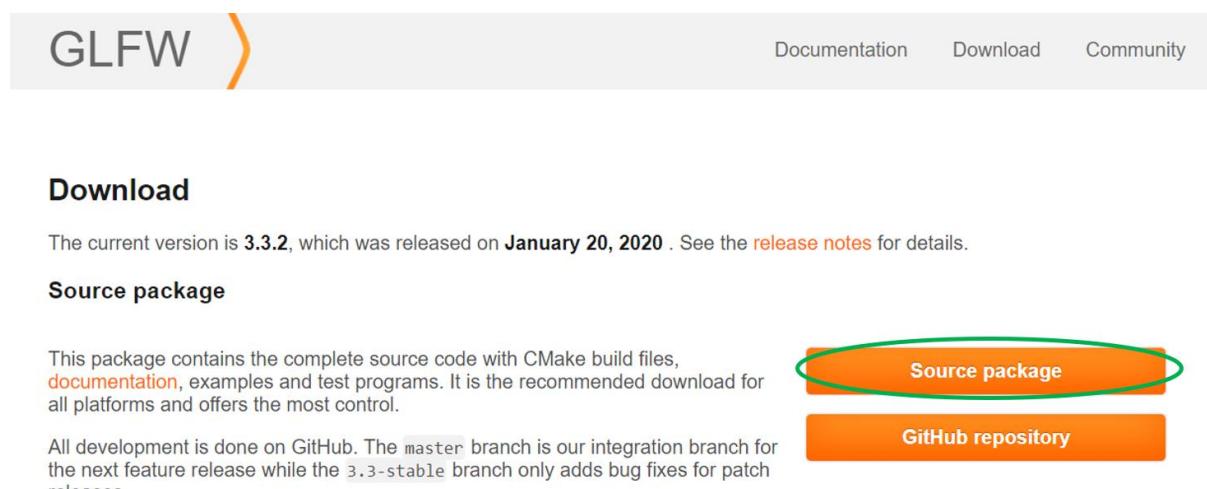


Figura 1 - Descărcarea surselor bibliotecii GLFW

- Extrageți întreg conținutul arhivei folosind Windows Explorer sau aplicația voastră preferată pentru gestionarea arhivelor (ex. 7-zip, WinRAR, etc).

Name	Date modified	Type	Size
glfw-3.3.2	20.01.2020 02:12	File folder	
glfw-3.3.2.zip	16.09.2020 09:39	zip Archive	1.401 KB

Figura 2 - Extragerea conținutului arhivei

- Pentru a putea compila sursele bibliotecii GLFW folosind Microsoft Visual Studio (MVS), descărcați și instalați CMake pentru Microsoft Windows <https://cmake.org/download/>. Pentru instrucțiuni despre instalarea Microsoft Visual Studio, mergeți la secțiunea 3.1.

Platform	Files
Unix/Linux Source (has \n line feeds)	cmake-3.18.2.tar.gz
Windows Source (has \r\n line feeds)	cmake-3.18.2.zip

Binary distributions:	
Platform	Files
Windows win64-x64 Installer: Installer tool has changed. Uninstall CMake 3.4 or lower first!	cmake-3.18.2-win64-x64.msi
Windows win64-x64 ZIP	cmake-3.18.2-win64-x64.zip
Windows win32-x86 Installer: Installer tool has changed. Uninstall CMake 3.4 or lower first!	cmake-3.18.2-win32-x86.msi
Windows win32-x86 ZIP	cmake-3.18.2-win32-x86.zip
Mac OS X 10.7 or later	cmake-3.18.2-Darwin-x86_64.dmg
	cmake-3.18.2-Darwin-x86_64.tar.gz
Linux x86_64	cmake-3.18.2-Linux-x86_64.sh
	cmake-3.18.2-Linux-x86_64.tar.gz

Figura 3 - Descărcarea și instalarea CMake

- Rulați aplicația CMake și selectați la rubrica „Where is the source code” calea spre folderul extras din arhivă (Figura 4). Selectați la rubrica „Where to build the binaries” calea către folderul *build* din folderul extras din arhivă (Figura 4). Dacă folderul *build* nu există, el va fi creat în timpul generării proiectului.

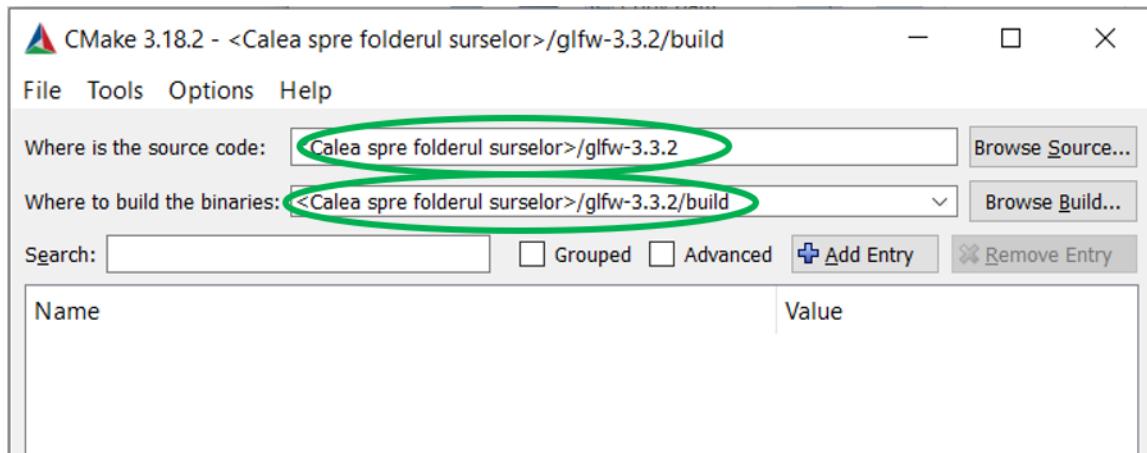


Figura 4 - Selectarea folderului sursă și a folderului în care se va genera proiectul MVS

- Apăsați Configure pentru actualizarea opțiunilor existente în generarea proiectului MVS (Figura 5)

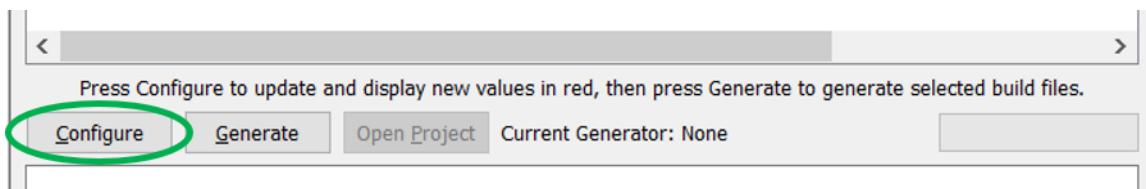


Figura 5 - Actualizarea opțiunilor pentru generarea proiectului MVS

- Selectați varianta de Microsoft Visual Studio pe care o aveți instalată (Figura 6). Dacă nu aveți instalat încă MVS, urmați pașii descriși în secțiunea 3.1 pentru instalare.

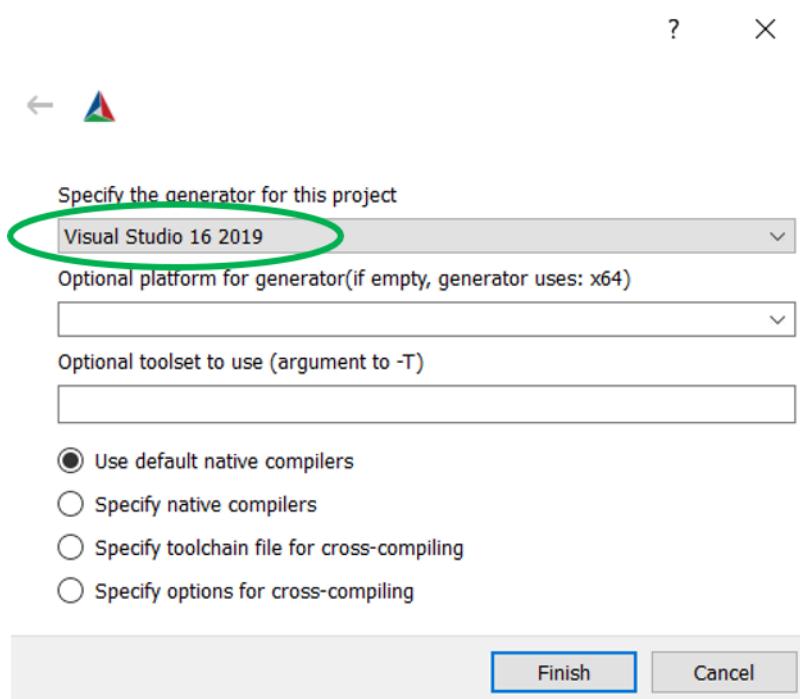


Figura 6 - Selectarea variantei de MVS instalate

- **Pentru utilizatorii avansați:** Dacă doriți, modificați starea implicită a opțiunilor de generare a proiectului MVS (ex. Dacă doriți generarea unor binare partajate (.dll) a bibliotecii GLFW, bifați opțiunea BUILD\_SHARED\_LIBS). **Dacă nu sunteți un utilizator avansat sau doriți compilarea bibliotecii GLFW doar pentru a fi utilizată pentru proiectele necesare la laborator, lăsați opțiunile în starea lor implicită!**
- Apăsați **din nou** butonul Configure pentru a confirma opțiunile cu care se va genera proiectul MVS. Fundalul opțiunilor ar trebui să se schimbe din roșu în alb (Figura 7).

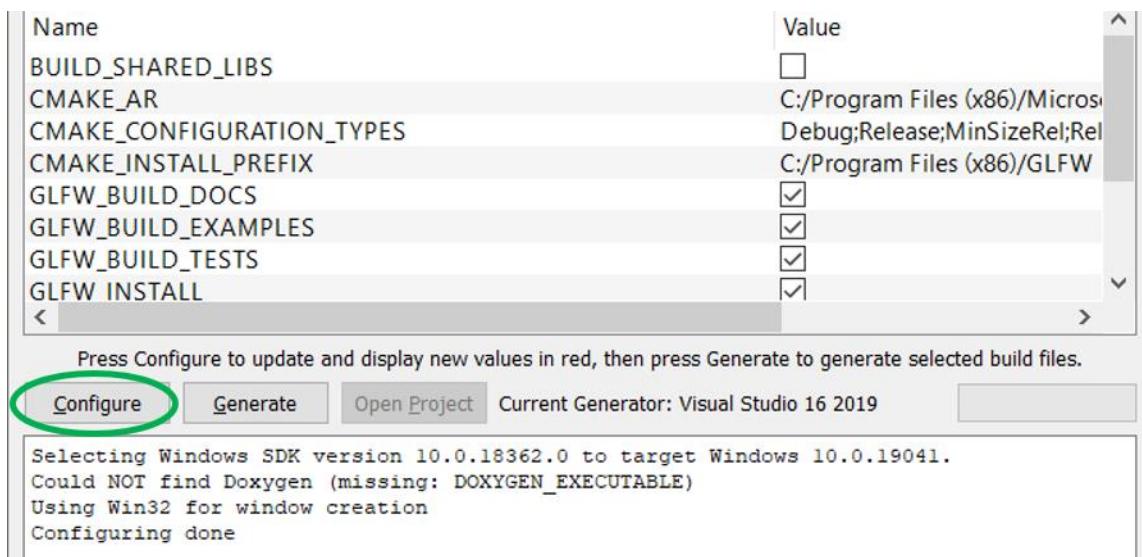


Figura 7 - Confirmarea opțiunilor pentru generarea proiectului MVS

- Dacă nu apar mesaje de eroare de configurație, în acest moment se poate genera soluția MVS care va fi utilizată pentru compilarea bibliotecii GLFW apăsând butonul Generate (Figura 8)

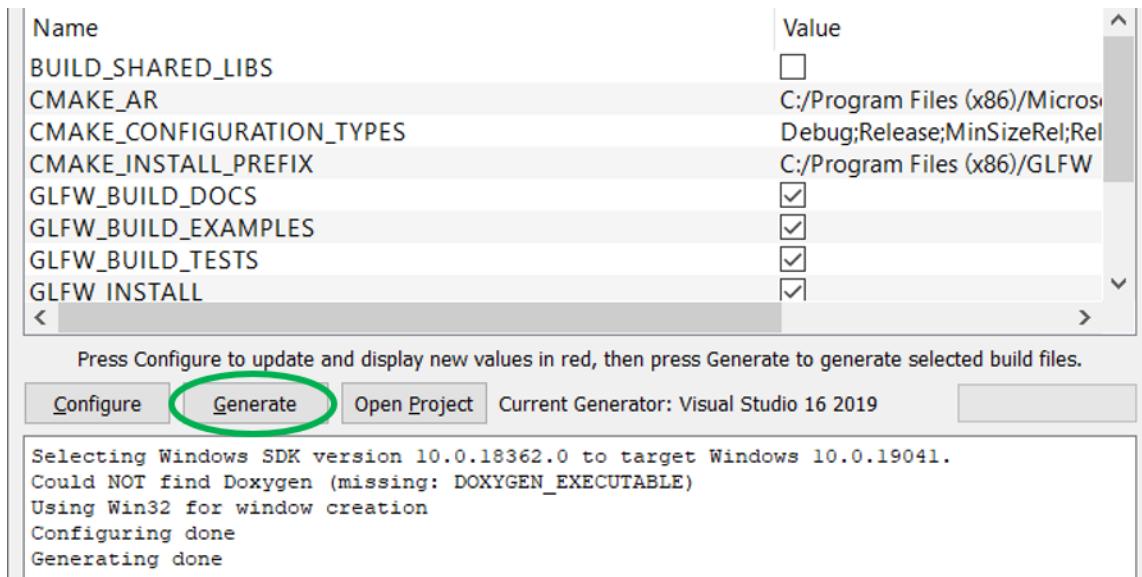


Figura 8 - Generarea proiectului MVS

- Dacă nu există erori, CMake va genera soluția și proiectul MVS cu numele `GLFW.sln` în folderul `build` din folderul care conține sursele bibliotecii. Dacă MVS este instalat, proiectul poate fi deschis direct din aplicația CMake, folosind butonul Open Project (Figura 9).

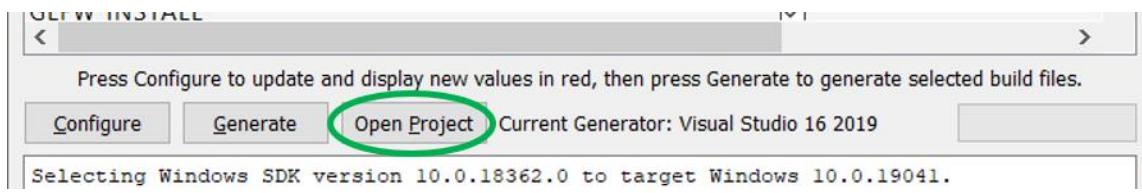


Figura 9 - Deschiderea proiectului MVS

- Selectați configurația dorită (ex. Debug sau Release), selectați platforma dorită (ex. x86 sau x64) (se recomandă utilizarea platformei x64) (Figura 10) și selectați din meniu fereastră MVS Build -> Rebuild Solution; **Important!** Configurația și platforma selectată acum trebuie să coincidă cu configurația și platforma cu care va fi creat proiectul OpenGL ulterior, altfel vor apărea mesaje de eroare! Întrucât s-ar putea ca proiectul OpenGL final să fie compilat atât în configurația Debug cât și în configurația Release, se recomandă să se genereze fișierele binare ale bibliotecii GLFW pentru ambele (ceea ce presupune două compilări diferite pentru proiectul curent: o dată în configurația Debug și platforma x64 și a doua oară în configurația Release și platforma x64).

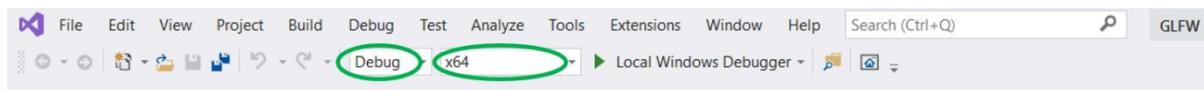


Figura 10 - Selectarea configurației și a platformei

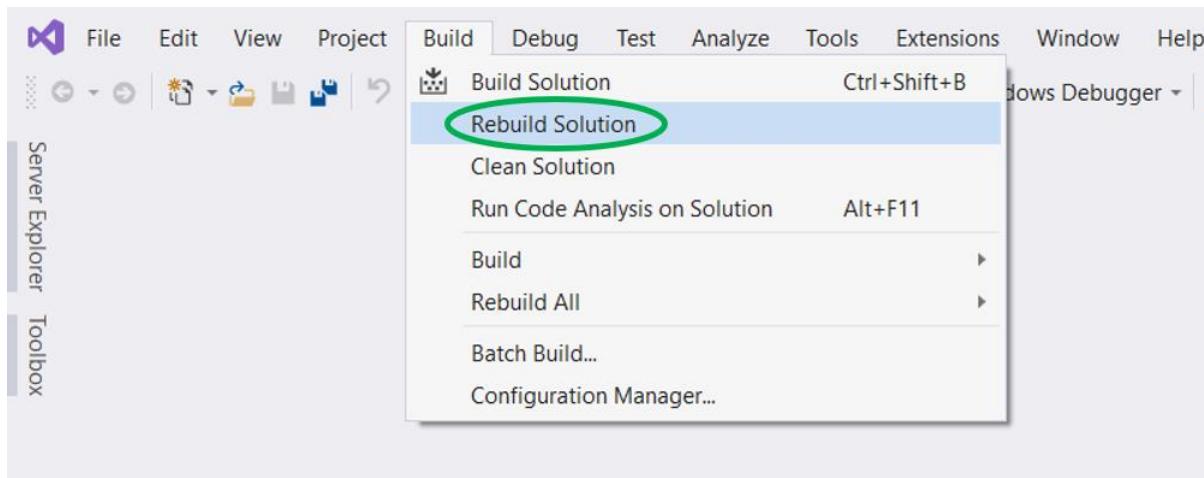


Figura 11 - Compilarea proiectului și generarea binarelor bibliotecii GLFW

- Fișierele header ale bibliotecii GLFW se vor găsi în folderul *include* din folderul rădăcină al bibliotecii (Figura 12), iar fișierele binare se vor găsi în folderul build->src-><>numele configurației (Debug sau Release)>> (Figura 13).

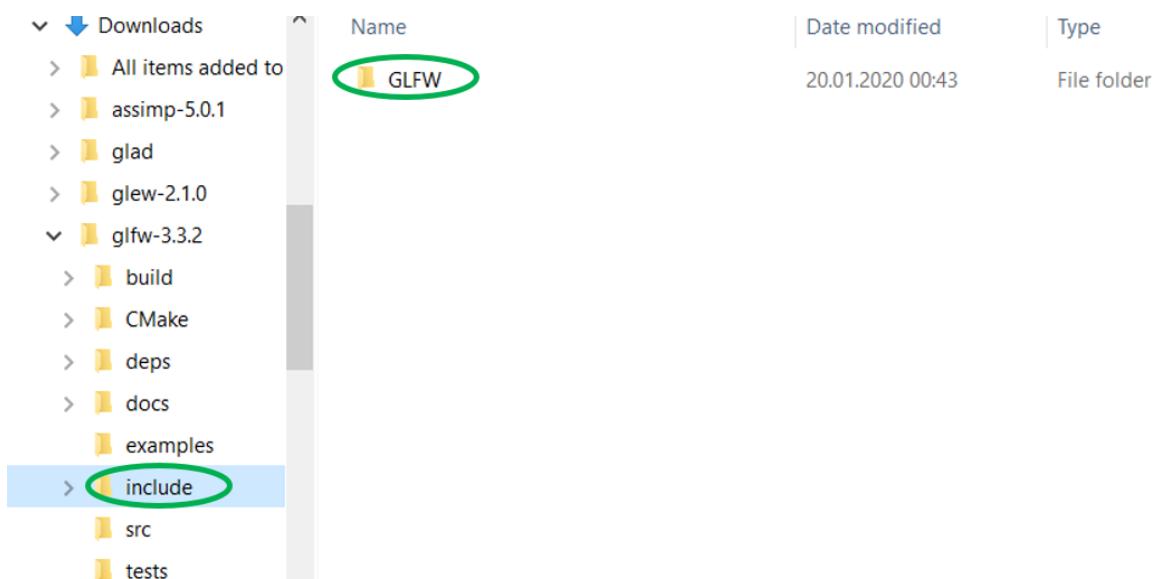


Figura 12 - Calea spre fișierele header

	Name	Date modified	Type
	glfw3.lib	22.09.2020 16:58	Object File Library

Figura 13 - Calea spre fișierele binare

## 1.2. Organizarea fișierelor bibliotecilor auxiliare pentru aplicațiile OpenGL

- Fișierele header și cele binare pot fi copiate într-un loc din care să poată fi adăugate cu ușurință în toate proiectele OpenGL viitoare (ex. Într-un folder cu numele „OpenGL dev libs”). Se recomandă următoarea structură a folderului cu biblioteci auxiliare: folderul va conține un subfolder cu numele „include” în care se vor pune toate fișierele header, fiecare bibliotecă într-un folder separat (ex. fișierele header (extensia .h) ale bibliotecii GLFW vor fi puse într-un folder cu numele „GLFW” în interiorul folderului *include*) și un folder „lib” cu două subfoldere, „Debug” și „Release” în care se vor pune fișierele binare ale tuturor bibliotecilor utilizate, în funcție de configurația pentru care au fost compilate (ex. toate fișierele din folderul *Debug* din Figura 13 vor fi copiate în folderul „Debug”, iar toate fișierele din folderul *Release* din Figura 13 vor fi copiate în folderul „Release”). Puteți observa structura recomandată în Figura 14, iar Figura 15 prezintă structura folderului după adăugarea fișierelor bibliotecii GLFW.

	Name	Date modified	Type
	include	22.09.2020 17:20	File folder
	lib	22.09.2020 17:21	File folder

Figura 14 - Structura recomandată

OpenGL dev libs	Name	Date modified	Type
include	glfw3.h	20.01.2020 00:43	C/C++ Header
GLFW	glfw3native.h	20.01.2020 00:43	C/C++ Header
lib			
Debug			
Release			

OpenGL dev libs	Name	Date modified	Type
include			
GLFW			
lib			
Debug	glfw3.lib	22.09.2020 16:58	Object File Library
Release			

OpenGL dev libs	Name	Date modified	Type
include			
GLFW			
lib			
Debug			
Release	glfw3.lib	22.09.2020 16:58	Object File Library

Figura 15 - Structura folderului după adăugarea fișierelor bibliotecii GLFW

## 2. Biblioteca GLEW

Similar cu biblioteca GLFW, pentru a putea dezvolta o aplicație folosind biblioteca GLEW este nevoie să se obțină fișierele header (extensia .h) ale bibliotecii și fișierele binare (extensia .lib) ale acesteia.

**Metoda recomandată pentru a obține fișierele binare este de a compila întreaga bibliotecă din surse. Doar compilarea din surse asigură compatibilitate maximă între bibliotecă și platforma pe care se vor dezvolta aplicațiile.** Alternativ, binarele precompilate pentru diferite platforme predefinite pot fi descărcate de pe website-ul <http://glew.sourceforge.net/>.

### 2.1. Compilarea bibliotecii din surse

- Sursele bibliotecii pot fi descărcate de pe website-ul <http://glew.sourceforge.net/>. (Figura 16)

#### The OpenGL Extension Wrangler Library

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file. GLEW has been tested on a variety of operating systems, including Windows, Linux, Mac OS X, FreeBSD, Irix, and Solaris.

##### Downloads

GLEW is distributed as source and precompiled binaries.  
The latest release is 2.1.0[07-31-17]:



Figura 16 - Descărcarea surselor bibliotecii GLEW

- Procesul de compilare a bibliotecii GLEW este identic cu cel folosit pentru biblioteca GLFW, cu mențiunea că fișierele necesare pentru CMake se găsesc în <folderul surselor> -> build -> cmake, ceea ce duce la căile ilustrate în Figura 17.

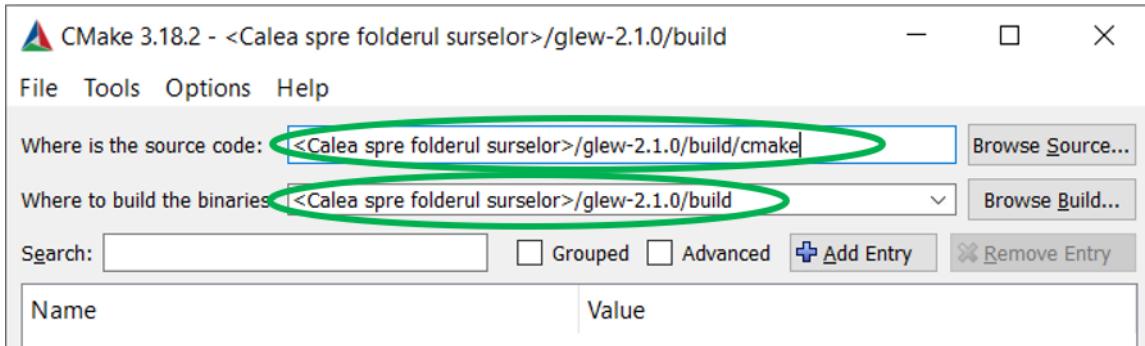


Figura 17 - Calea spre sursele bibliotecii GLEW

- Atenție!** Din cauza unor modificări pe care Microsoft Visual Studio 2019 le-a suferit față de variantele anterioare, pentru a putea configura și compila cu succes proiectul MVS al bibliotecii GLEW, este nevoie să editați înainte fișierul >folderul surselor>->build->cmake-> CMakeLists.txt, adăugând bibliotecile `vcruntime` în procesul de compilare (Figura 18).

```
if (MSVC)
    # add options from visual studio project
    target_compile_definitions (glew PRIVATE "GLEW_BUILD;VC_EXTRALEAN")
    target_compile_definitions (glew_s PRIVATE "GLEW_STATIC;VC_EXTRALEAN")
    target_link_libraries (glew LINK_PRIVATE vcruntime -BASE:0x62AA0000)
    # kill security checks which are dependent on stdlib
    target_compile_options (glew PRIVATE -GS-)
    target_compile_options (glew_s PRIVATE -GS-)
    # remove stdlib dependency
    target_link_libraries (glew LINK_PRIVATE -nodefaultlib -noentry)
    string(REGEX REPLACE "/RTC(su|lsu)" "" CMAKE_C_FLAGS_DEBUG ${CMAKE_C_FLAGS_DEBUG})
elseif (WIN32 AND ((CMAKE_C_COMPILER_ID MATCHES "GNU") OR (CMAKE_C_COMPILER_ID MATCHES "Clang")))
    # remove stdlib dependency on windows with GCC and Clang (for similar reasons)
    # as to MSVC - to allow it to be used with any Windows compiler
    target_compile_options (glew PRIVATE -fno-builtin -fno-stack-protector)
    target_compile_options (glew_s PRIVATE -fno-builtin -fno-stack-protector)
    target_link_libraries (glew LINK_PRIVATE -nostdlib)
endif ()
```

Figura 18 - Modificarea fișierului CMakeLists.txt

- După compilare, fișierele header ale bibliotecii GLFW se vor găsi în folderul `include` din folderul rădăcină al bibliotecii (Figura 19), iar fișierele binare se vor găsi în folderul build->lib-><<numele configurației (Debug sau Release)>> (Figura 21). Întrucât biblioteca GLEW se compilează implicit ca o bibliotecă partajată (dinamică),
- Pentru utilizatorii avansați:** Dacă doriți să utilizați varianta dinamică (partajată) a bibliotecii, fișierul DLL se va găsi în folderul build->bin-><<numele configurației (Debug sau Release)>> (Figura 20).

	Name	Date modified	Type
glew-2.1.0	GL	31.07.2017 14:25	File folder

The screenshot shows a file explorer window with the 'glew-2.1.0' directory expanded. The 'include' folder is selected and highlighted with a blue bar at the bottom. A green circle highlights the 'GL' folder within the 'glew-2.1.0' directory.

Figura 19 - Calea spre fișierele header

	Name	Date modified	Type
glew-2.1.0	glew32d.dll	22.09.2020 18:33	Application extens...
	glew32d.ilk	22.09.2020 18:33	Incremental Linker...
	glew32d.pdb	22.09.2020 18:33	Program Debug D...
	glewinfo.exe	22.09.2020 18:33	Application
	glewinfo.ilk	22.09.2020 18:33	Incremental Linker...
	glewinfo.pdb	22.09.2020 18:33	Program Debug D...
	visualinfo.exe	22.09.2020 18:33	Application

The screenshot shows a file explorer window with the 'glew-2.1.0' directory expanded. The 'bin' folder is selected and highlighted with a blue bar at the bottom. Within the 'bin' folder, the 'Debug' subfolder is also highlighted with a blue bar. A green circle highlights the 'glew32d.dll' file.

Figura 20 - Calea spre fișierele dll

	Name	Date modified	Type
glew-2.1.0	glew32d.exp	22.09.2020 18:33	Exports Library File
	glew32d.lib	22.09.2020 18:33	Object File Library
	libglew32d.lib	22.09.2020 18:33	Object File Library

The screenshot shows a file explorer window with the 'glew-2.1.0' directory expanded. The 'lib' folder is selected and highlighted with a blue bar at the bottom. Within the 'lib' folder, both the 'Debug' and 'Release' subfolders are highlighted with blue bars. A green circle highlights the 'glew32d.lib' file.

Figura 21 - Calea spre fișierele binare

## 2.1. Organizarea fișierelor bibliotecii GLEW

- Fișierele bibliotecii GLEW vor fi organizate conform convenției prezentate în secțiunea 1.2. Puteți observa în Figura 22 structura recomandată după ce se vor adăuga fișierele GLEW.

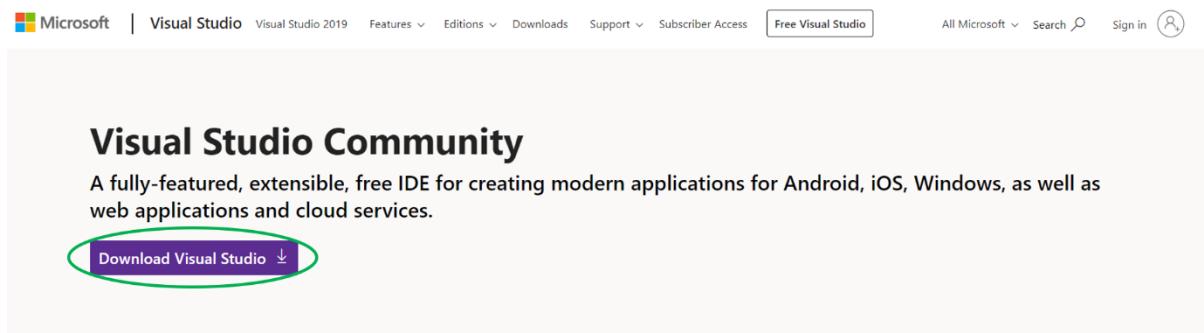
	Name	Date modified	Type
OpenGL dev libs			
include	eglew.h	31.07.2017 14:25	C/C++ Header
GL	glew.h	31.07.2017 14:25	C/C++ Header
GLFW	glew.h	31.07.2017 14:25	C/C++ Header
lib	wglew.h	31.07.2017 14:25	C/C++ Header
OpenGL dev libs			
include	glew32d.dll	30.09.2020 16:35	Application exten
GL	glew32d.exp	30.09.2020 16:35	Exports Library Fil
GLFW	glew32d.lib	30.09.2020 16:35	Object File Librar
lib	glfw3.lib	22.09.2020 16:58	Object File Librar
Debug	libglew32d.lib	30.09.2020 16:35	Object File Librar
Release			
OpenGL dev libs			
include	glew32.dll	30.09.2020 16:36	Application exten
GL	glew32.exp	30.09.2020 16:36	Exports Library Fil
GLFW	glew32.lib	30.09.2020 16:36	Object File Library
lib	glfw3.lib	22.09.2020 16:58	Object File Library
Debug	libglew32.lib	30.09.2020 16:36	Object File Library
Release			

Figura 22 - Structura folderului după adăugarea fișierelor bibliotecii GLEW

### 3. Microsoft Visual Studio

#### 3.1. Instalarea Microsoft Visual Studio

- Microsoft Visual Studio Community poate fi descărcat gratuit de pe website-ul <https://visualstudio.microsoft.com/vs/community/> (Figura 23). Acesta necesită doar un cont Microsoft pentru a putea fi folosit fără încrerere.



Everything you need all in one place

Figura 23 - Descărcarea Microsoft Visual Studio

- Modulul necesar și suficient pentru laborator este cel de dezvoltare aplicații desktop folosind C++ (Figura 24)

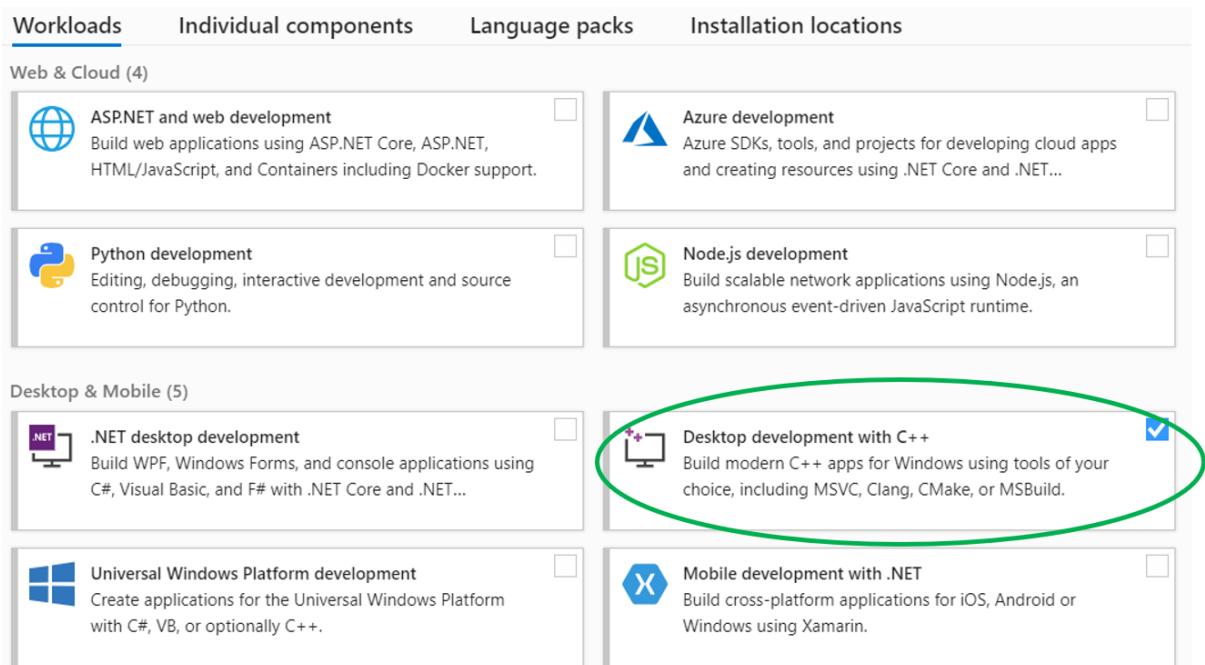


Figura 24 - Selectarea modulelor MVS

### 3.2. Crearea unui proiect nou OpenGL în Microsoft Visual Studio

- Creați un nou proiect Visual Studio (Figura 25).

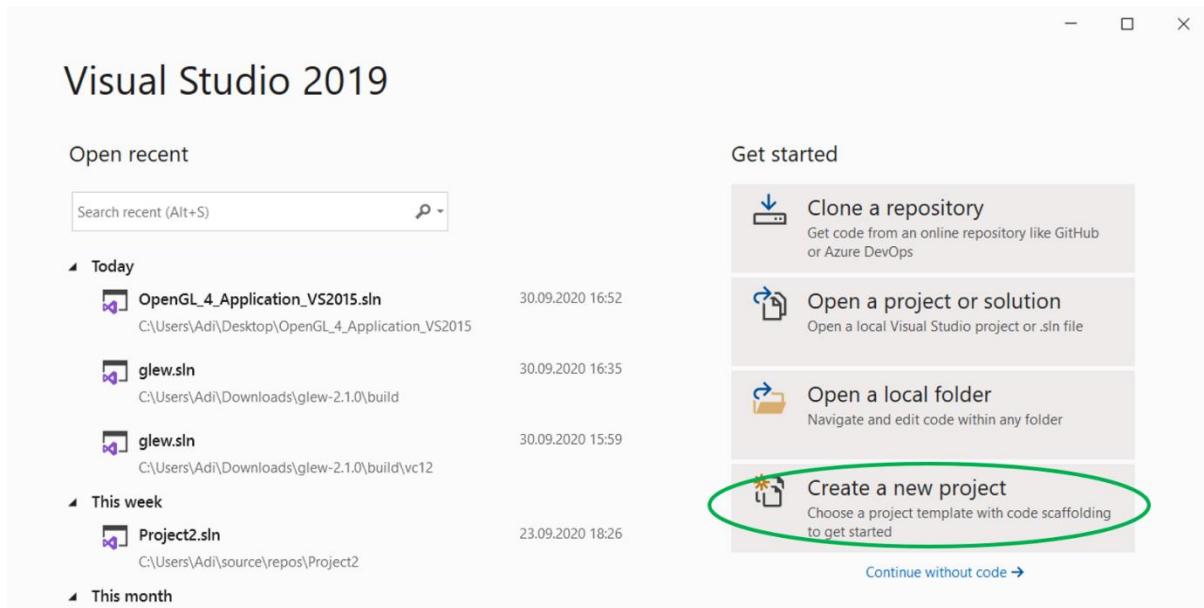


Figura 25 - Crearea unui proiect nou

- Utilizați şablonul „Empty Project” (Figura 26).

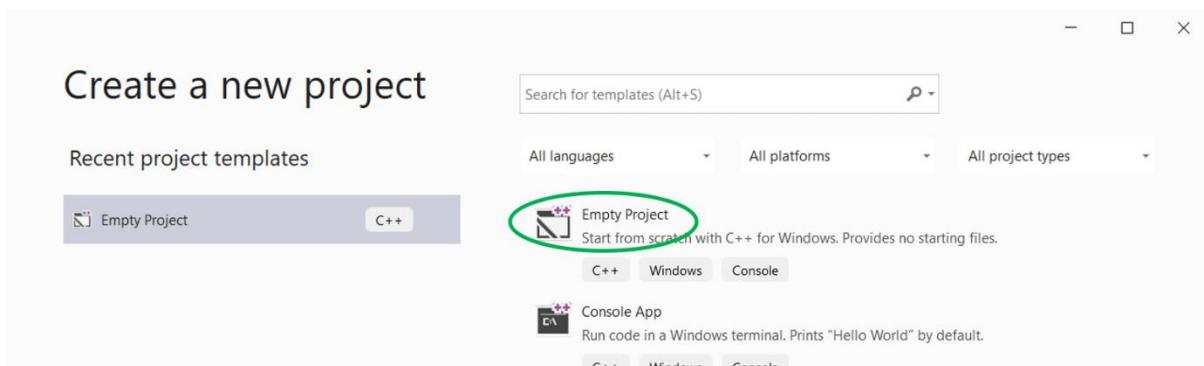


Figura 26 - Alegerea şablonului

- Copiați sursele proiectului în folderul rădăcină al acestuia. Pentru a deschide folderul rădăcină, Efectuați click-dreapta pe numele proiectului și selectați opțiunea „Open Folder in File Explorer” (Figura 27)

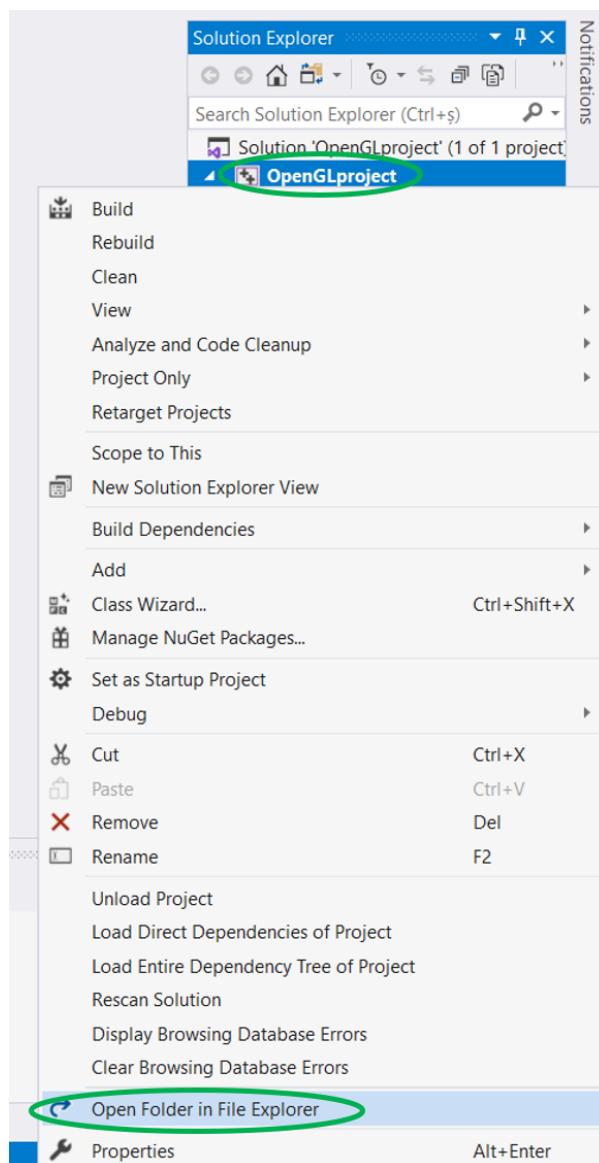


Figura 27 - Accesarea folderului proiectului

- Adăugați sursele în proiect efectuând click-dreapta pe numele proiectului și selectând opțiunea „Add->Existing Item”. **Atenție!** Adăugați sursele din folderul proiectului, nu alte copii ale acestora existente în alte locații.
- Adăugați calea spre fișierele header ale bibliotecilor auxiliare în *Include Path*
  - Selectați din meniu Project->Properties (Figura 28)

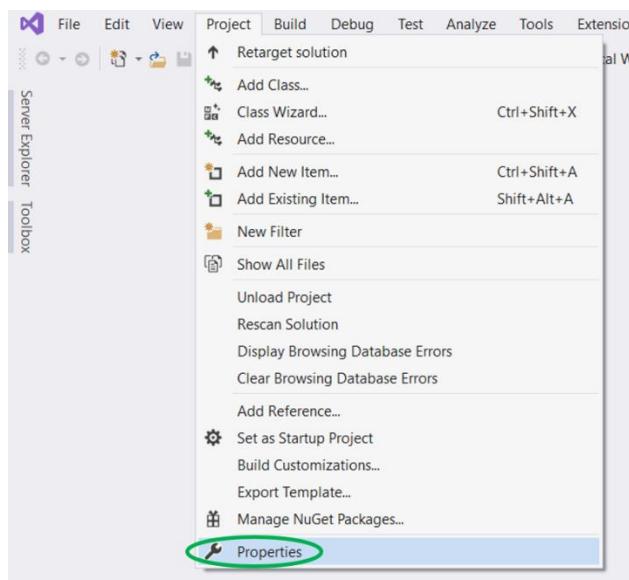


Figura 28 - Accesarea proprietăților proiectului

- Adăugați calea spre folderul ce conține fișierele header ale bibliotecilor auxiliare la rubrica „Additional Include Directories” din Proprietățile „C/C++”, secțiunea „General”. Asigurați-vă că faceți modificările pentru **toate configurațiile** (atât Debug cât și Release) (Figura 29 și Figura 30)

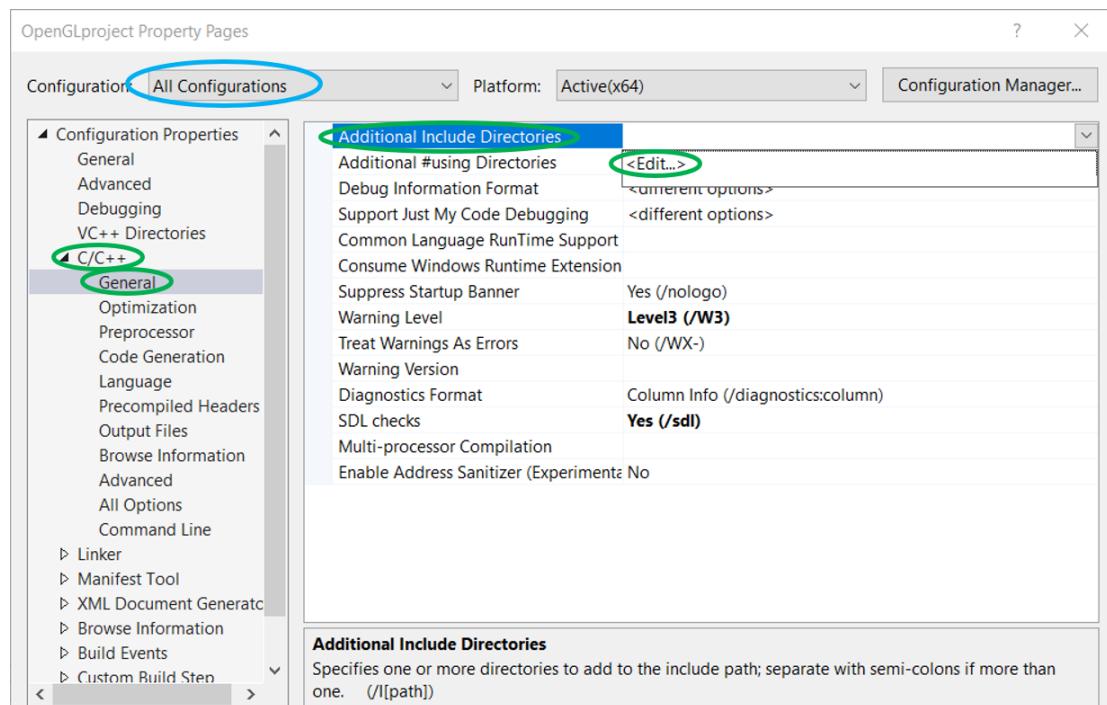


Figura 29 - Adăugarea folderelor cu fișiere header suplimentare

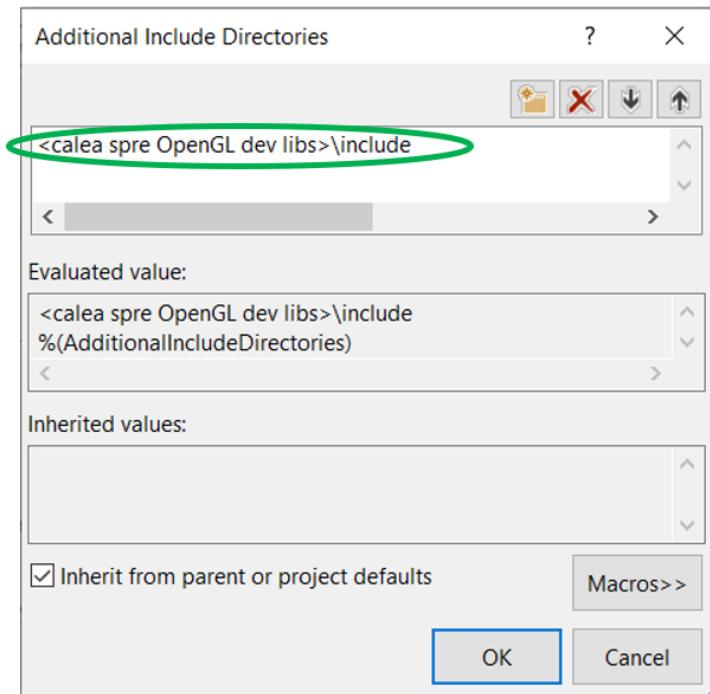


Figura 30 - Calea spre fișierele header

- Adăugați calea spre fișierele binare ale bibliotecilor auxiliare în *Library Path*
  - Adăugați calea spre folderul ce conține fișierele binare ale bibliotecilor auxiliare la rubrica „Additional Library Directories” din Proprietățile „Linker”, secțiunea „General”. Asigurați-vă că pentru configurația Debug adăugați calea spre folderul care conține varianta Debug a binarelor (<calea spre OpenGL dev libs>\lib\Debug), iar pentru configurația Release adăugați calea spre folderul care conține varianta Release a binarelor (<calea spre OpenGL dev libs>\lib\Release) (Figura 31)

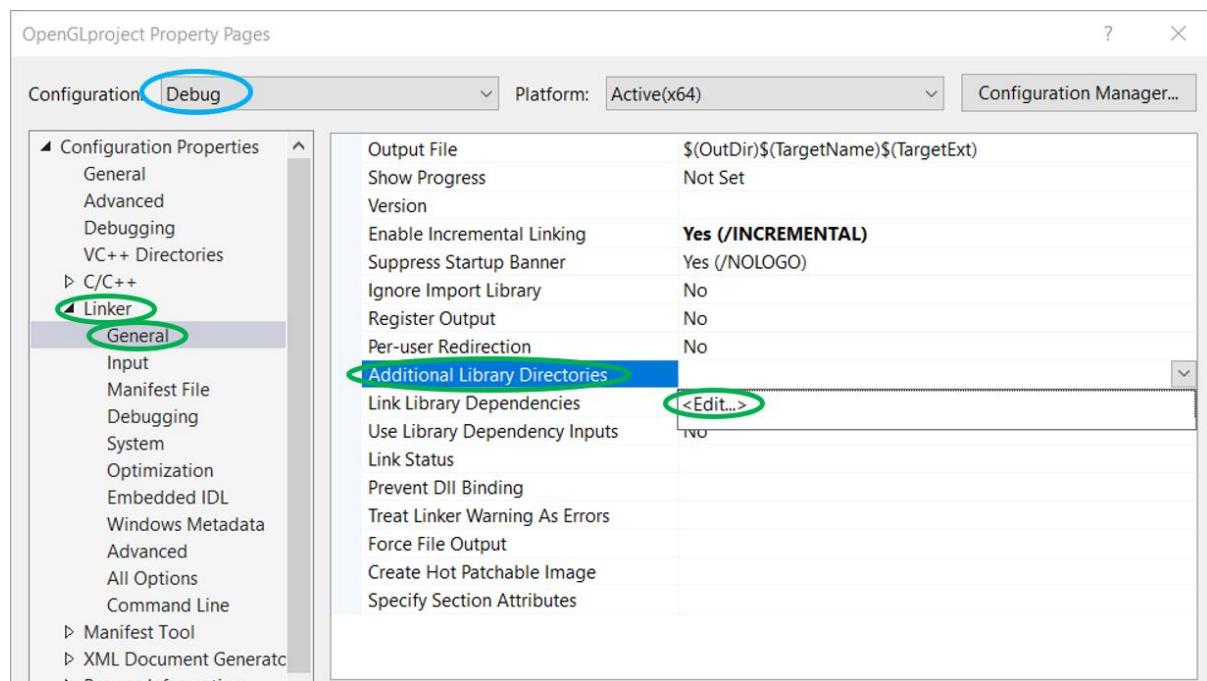


Figura 31 - Adăugarea folderelor cu fișiere binare suplimentare

- Adăugați numele bibliotecilor auxiliare care vor fi folosite la rubrica „Additional Dependencies” din Proprietățile „Linker”, secțiunea „Input” (Figura 32). **Atenție!** Se poate întâmpla ca numele binarelor să difere în funcție de configurația pentru care au fost create (Debug sau Release). Se recomandă consultarea numelor fișierelor binare din folderul cu biblioteci auxiliare înainte de adăugarea lor pentru cele două configurații.

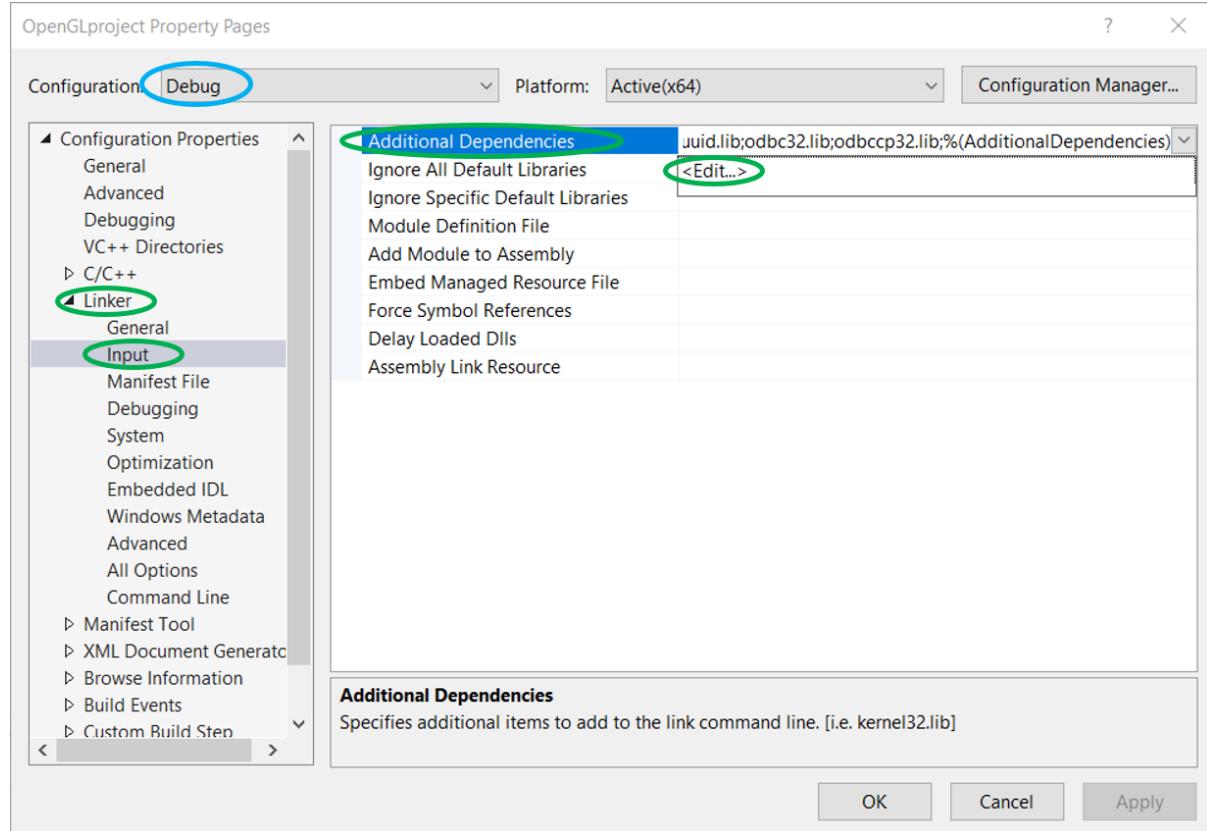


Figura 32 - Adăugarea numelor bibliotecilor

- Numele bibliotecilor adiționale necesare sunt ilustrate în Figura 33 (Configurația Debug în stânga și configurația Release în dreapta)

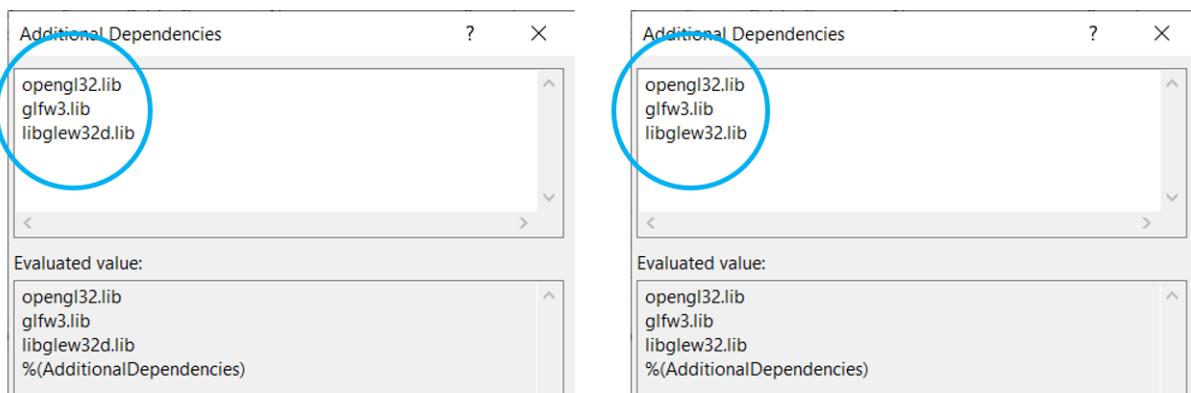


Figura 33 - Numele bibliotecilor adiționale

### 3.3. Crearea unui nou şablon de proiect pe baza proiectului complet configurat

- După configurarea completă a proiectului, acesta poate fi folosit ca baza unui nou şablon pentru Microsoft Visual Studio
- Selectați din meniu Project->Export Template (Figura 34)

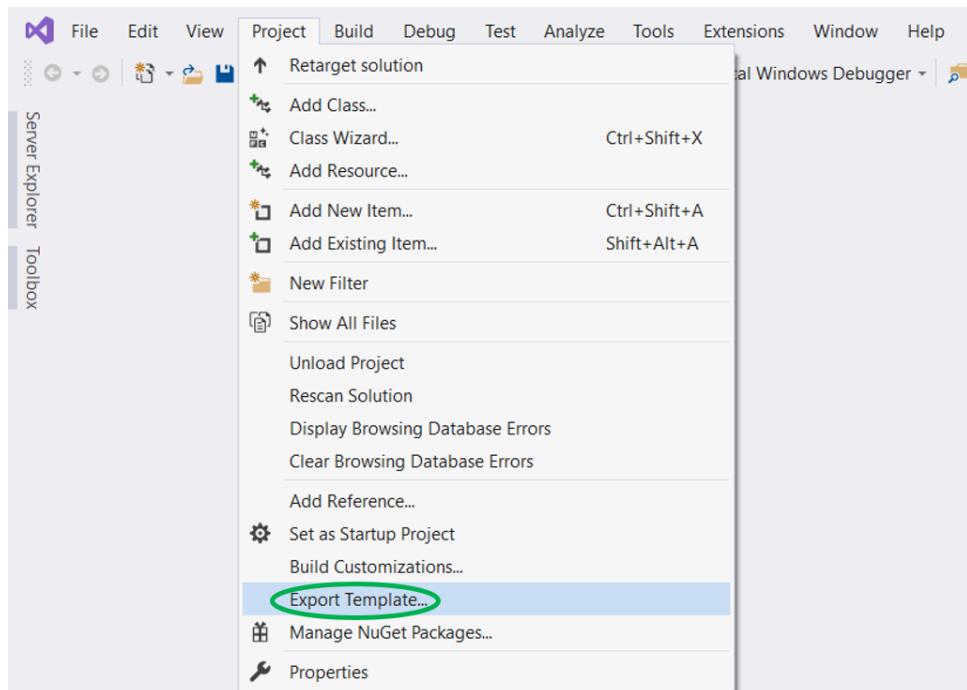


Figura 34 - Exportarea proiectului ca şablon

- Selectați opțiunea „Project template” (Figura 35)

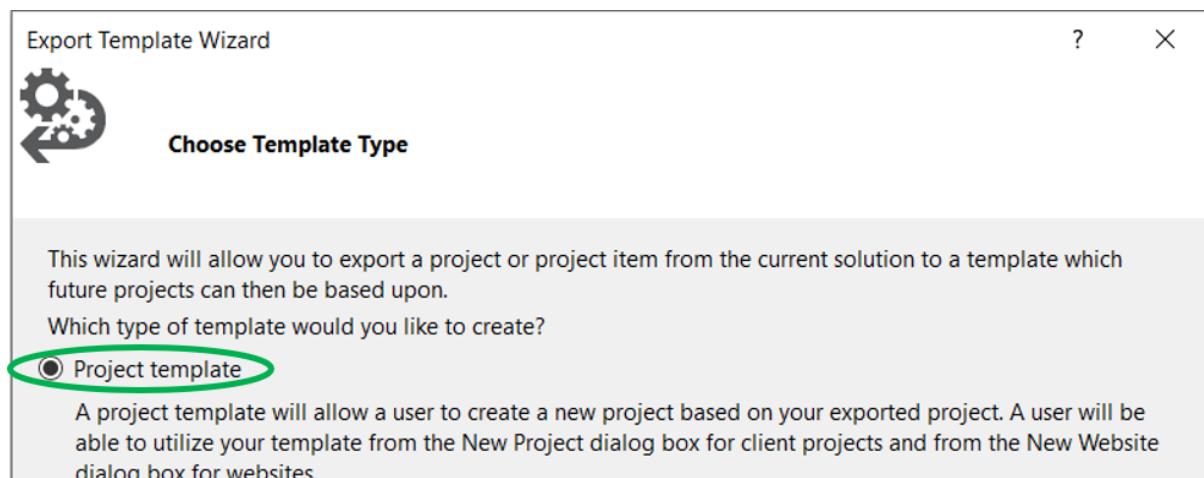


Figura 35 - Tipul sablonului

- Completăți câmpurile descriptive ale şablonului (Figura 36)

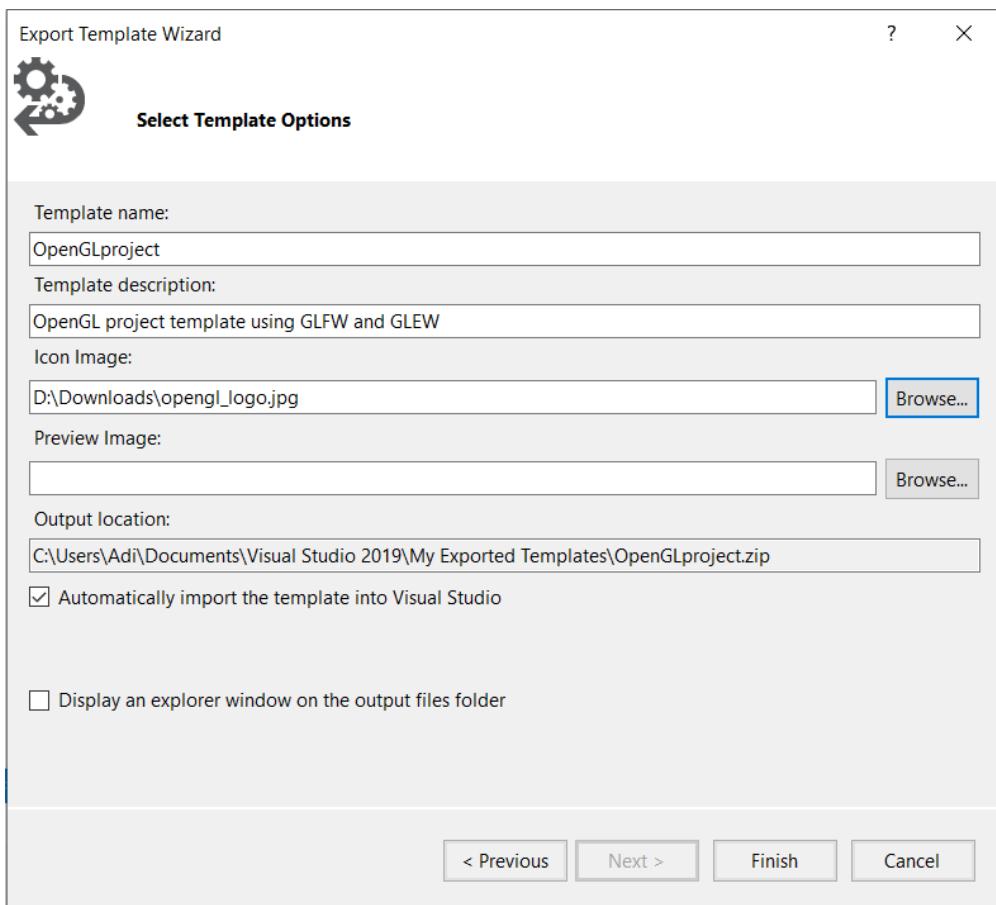


Figura 36 - Câmpurile descriptive ale şablonului

- Şablonul poate fi acum utilizat pentru crearea unui proiect nou (Figura 37)

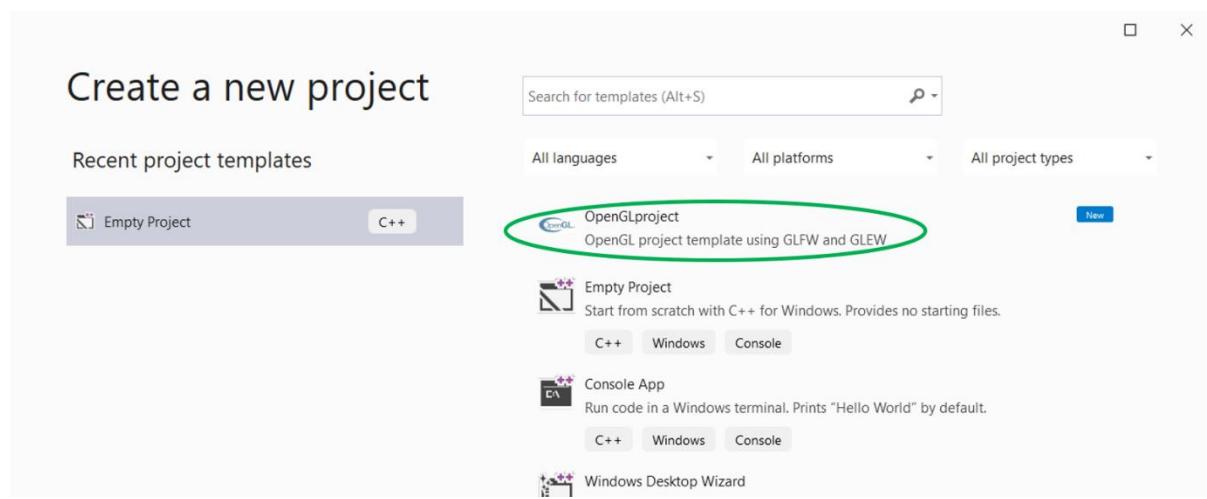


Figura 37 - Utilizarea şablonului

# Laborator 3

---

## 1 Obiective

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

## 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 aşa-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 funcții 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 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 stagile 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.

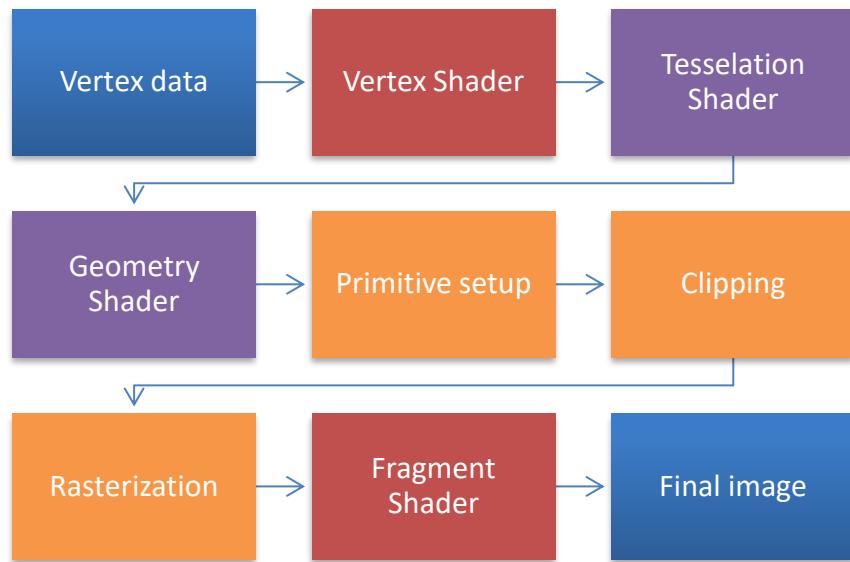


Figure 1 – Pipeline-ul programabil OpenGL

## 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 sănătă.

### 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 aşa-numit obiect de tip **program shader**. Pașii pentru aceasta sunt după cum urmează:

- creăm un program shader: `glCreateProgram(...)`
- atașăm 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 opțiune `Add existing 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 sistemul de coordinate normalizez
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 compilăm 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);
    //specifică locația și dimensiunea ferestrei
    glViewport (0, 0, retina_width, retina_height);

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

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

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

    //activează VAO
    glBindVertexArray(triangleVAO);
    //specifică tipul primitiei, indicele de început și 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
<a href="#">glGenBuffers</a>	Generează nume de obiecte buffer
<a href="#">glBindBuffer</a>	Asociază un obiect buffer punctului de asociere specificat
<a href="#">glBufferData</a>	Creează și inițializează setul de date al unui obiect buffer
<a href="#">glGenVertexArrays</a>	Generează nume pentru obiecte vertex array object
<a href="#">glBindVertexArray</a>	Asociază un obiect vertex array
<a href="#">glVertexAttribPointer</a>	Definește un vector cu date generice ale vârfurilor
<a href="#">glEnableVertexAttribArray</a>	Activează un vertex array cu date generice
<a href="#">glCreateShader</a>	Creează un obiect shader
<a href="#">glShaderSource</a>	Înllocuiește codul sursă al unui obiect shader
<a href="#">glCompileShader</a>	Compilează un obiect shader
<a href="#">glCreateProgram</a>	Creează un obiect program
<a href="#">glAttachShader</a>	Atașează un obiect shader unui obiect program

<a href="#">glLinkProgram</a>	Conectează(linking) un obiect program
<a href="#">glDeleteShader</a>	Șterge un obiect shader
<a href="#">glUseProgram</a>	Instalează un obiect program ca parte a stării de rasterizare actuale
<a href="#">glDrawArrays</a>	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.

# 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 ieșire
out type outputVariableName;

//lista variabilelor de tip uniform
uniform type uniformName;

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

### 2.2 Tipuri de date

În GLSL, putem folosi majoritatea tipurilor implicate 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 atributelor vârfului (ar putea fi poziția, culoarea, vectorul normal etc.). Pentru a se potrivi atributelor 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**: aşteaptă un unsigned int
- **3f**: aşteaptă un 3 valori float
- **fv**: aşteaptă 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. Luati 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 normalizează 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 utilizati 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 aplelăm `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 primită 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);
}
```

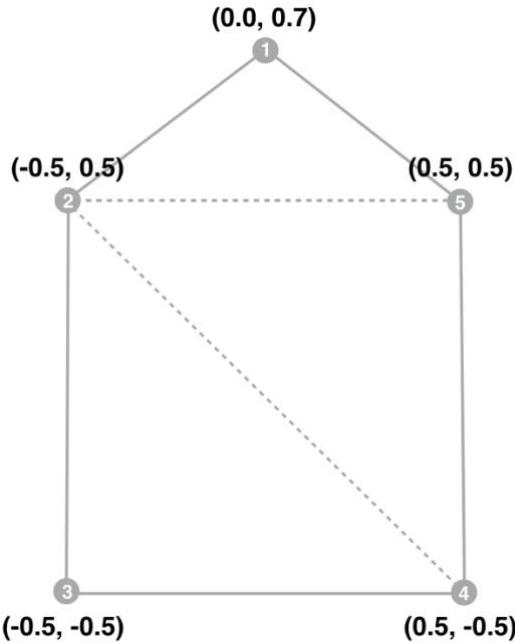
## 4 Lectură suplimentară

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

Funcție (și link)	Descriere
<a href="#">glGenBuffers</a>	Generează nume de obiecte buffer
<a href="#">glBindBuffer</a>	Asociază un obiect buffer punctului de asociere specificat
<a href="#">glBufferData</a>	Creează și inițializează setul de date al unui obiect buffer
<a href="#">glGenVertexArrays</a>	Generează nume pentru obiecte vertex array object
<a href="#">glBindVertexArray</a>	Asociază un obiect vertex array
<a href="#">glVertexAttribPointer</a>	Definește un vector cu date generice ale vârfurilor
<a href="#">glEnableVertexAttribArray</a>	Activează un vertex array cu date generice
<a href="#">glCreateShader</a>	Creează un obiect shader
<a href="#">glShaderSource</a>	Înlocuiește codul sursă al unui obiect shader
<a href="#">glCompileShader</a>	Compilează un obiect shader
<a href="#">glCreateProgram</a>	Creează un obiect program
<a href="#">glAttachShader</a>	Atașează un obiect shader unui obiect program
<a href="#">glLinkProgram</a>	Conectează(linking) un obiect program
<a href="#">glDeleteShader</a>	Șterge un obiect shader
<a href="#">glUseProgram</a>	Instalează un obiect program ca parte a stării de rasterizare actuale
<a href="#">glDrawArrays</a>	Rasterizează primitive din vectorul de date
<a href="#">glGetUniformLocation</a>	Returnează locația unei variabile uniform
<a href="#">glUniform</a>	Specifică valoarea unei variabile uniform pentru shader program-ul curent
<a href="#">glDrawElements</a>	Rasterizează primitive din vectorul de date

## 5 Temă

- Refaceti urmatoarea figura folosind EBO si definiți culori diferite pentru fiecare vârf. Nu replicați nodurile!!!



- Modificați coordonatele vârfurilor (pe axa x și y) utilizând două variabile uniform care sunt trimise către vertex shader. Actualizați valorile uniformelor utilizând intrări de la tastatură.
- Comutați între reprezentările plane și wireframe utilizând 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);
```

# Laborator 5

---

## 1 Obiective

Acest laborator prezintă noțiunile cheie privind transformările 3D. Acestea va acoperi modelul de bază (traducerea, scara, rotația), transformările camerei și proiecției.

## 2 Definirea punctelor 2D și 3D

Un punct 2D este definit într-un sistem de coordonate omogen prin  $(x^*w, y^*w, w)$ . Pentru simplitate, în sistemele bidimensionale, parametrul  $w$  este setat la 1. Prin urmare, definiția punctului este  $(x, y, 1)$ .

O altă reprezentare pentru acest punct este următoarea:  $P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ .

În mod similar, în 3D definim punctele ca  $(x^*w, y^*w, z^*w, w)$  și reprezentăm punctul ca vector coloană:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

## 3 Transformări

După ce au fost procesate de Vertex Shader, se preconizează că nodurile care urmează a fi redată vor avea coordonatele lor încadrate în intervalul  $[-1, +1]$  - denumit și Normalized Device Coordinates (NDC). Pentru a obține vârfurile în această etapă, ele trebuie să sufere o serie de transformări, ajungând din Spațiul Obiectului Local (Local Object Space) în Spațiul de Global de Coordonate al Scenei (World Coordinate System), apoi în pe Spațiul Coordonatelor de Vizualizare (View Coordinates Space) și, în final, în Spațiul de Clipping (Clipping Space).

### 3.1 Transformări de model

Atunci când definiți un obiect, vârf cu vârf, faceți acest lucru relativ la un sistem de coordonate locale. De cele mai multe ori, pentru comoditate, definiți obiectele centrate în originea acestui sistem - aceasta este o practică standard pentru majoritatea instrumentelor de modelare a obiectelor. Când plasați obiectul într-o anumită scenă (lume), trebuie să îl poziționați în locul potrivit și poate să îl re-dimensionați pentru a-l încadra în interiorul scenei.

Transformările (translație, rotație, scalare) necesare pentru poziționarea modelului într-o scenă se mai numesc și transformări de model. Ele transformă vârfurile obiectului din spațiul local în spațiul global.

### 3.1.1 Translație

Transformarea de translație este folosită pentru a muta un obiect (punct) cu un anumit deplasament.

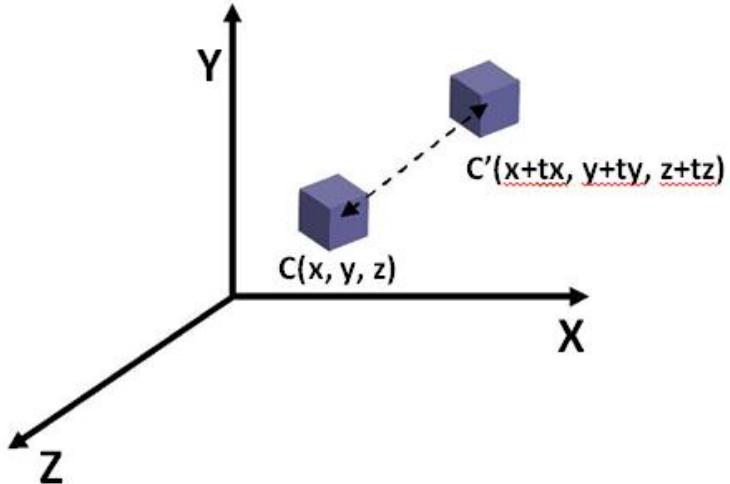
Matricea de translație este:

$$T = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matricea de translație inversă este:

$$T' = \begin{bmatrix} 1 & 0 & 0 & -Tx \\ 0 & 1 & 0 & -Ty \\ 0 & 0 & 1 & -Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$Tx$ ,  $Ty$  și  $Tz$  reprezintă factorii de translație pe axele  $x$ ,  $y$  și  $z$ .



Puteți genera o matrice de translație utilizând:

```
glm::translate(glm::mat4 init_matrix, glm::vec3(float Tx, float Ty, float Tz))
```

### 3.1.2 Scalare

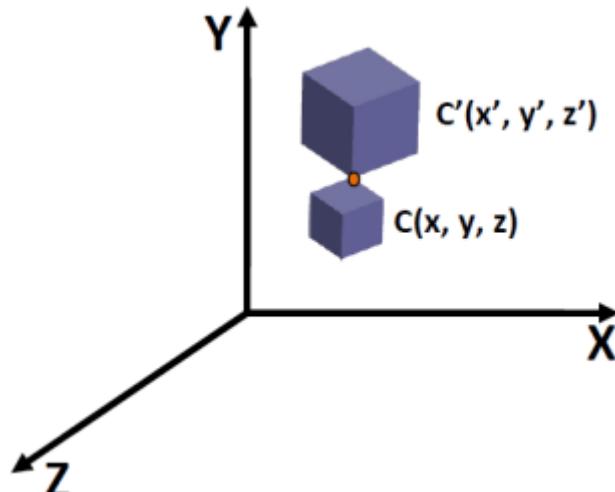
Transformarea de scalare mărește sau reduce dimensiunile unui obiect. Transformarea este *relativă la origine*.

Matricea de scalare este:

$$S = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matricea inversă de scalare este:

$$S' = \begin{bmatrix} 1/Sx & 0 & 0 & 0 \\ 0 & 1/Sy & 0 & 0 \\ 0 & 0 & 1/Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$Sx$ ,  $Sy$  și  $Sz$  reprezintă factorii de scalare pe axele  $x$ ,  $y$  și  $z$ . Dacă  $Sx$ ,  $Sy$  și  $Sz$  sunt egali, transformarea de scalare este uniformă. Dacă  $Sx$ ,  $Sy$  și  $Sz$  nu sunt egali, transformarea de scalare este neuniformă.

Dacă setați factorii de scalare la  $+/ - 1$ , atunci puteți reflecta forma originală.

Puteți genera o matrice de scalare folosind:

```
glm::scale(glm::mat4 init_matrix, glm::vec3(float Sx, float Sy, float Sz))
```

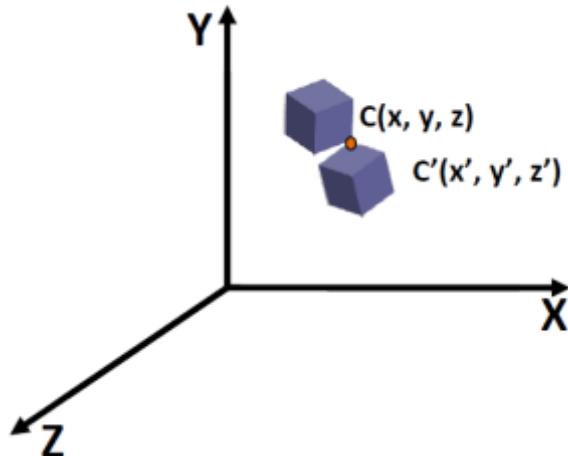
### 3.1.3 Rotatie

Această transformare rotește un obiect cu un unghi dat. Această transformare este, de asemenea, relativă la origine.

Pentru un punct 2D, matricile de rotație și rotație inversă sunt următoarele:

$$R = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

În spațiul 3D, specificăm rotația în mod independent pe axele x, y și z. Rotația în jurul axei z este similară cu rotația din 2D (coordonatele z rămân neschimbate).



$$R_z = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_y = \begin{bmatrix} \cos \alpha & 0 & \sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Pentru a schimba centrul de rotație - adică pentru a face o rotație în jurul unui punct dat în locul originii - trebuie să facem o serie de trei transformări:

- Translați noul punct central în origine
- Realizați rotația
- Translația inversă a punctului

Din punct de vedere matematic, aceasta ar însemna:  $\mathbf{P}_{\text{new}} = \mathbf{T}_{\text{inverse}} * \mathbf{R} * \mathbf{T} * \mathbf{P}$

Puteți genera o matrice de rotație folosind:

```
glm::rotate(glm::mat4 init_matrix, float angle, glm::vec3(float x, float y, float z))
```

- x, y și z reprezintă vectorul 3D (axa) în jurul căruia este efectuată rotația
- unghiu de rotație este specificat în grade, nu în radiani.

## 3.2 Transformări ale camerei

După ce scena a fost configurată, trebuie să definiți poziția și direcția de vizionare a observatorului (camera). Acest lucru este făcut pentru a trasa grafic scena din perspectiva observatorului. Dacă până acum coordonatele vârfurilor erau relative la originea scenei, acum ele vor fi exprimate în funcție de

poziția camerei. Trebuie să efectuăm o serie de rotații și translații pentru a aduce originea scenei în poziția camerei. Aceasta se numește transformarea din Spațiul Global în Spațiul Coordonatelor de Vizualizare.

Seria de transformări necesare pentru a converti scena în Spațiul de Vizualizare este cuprinsă într-o matrice, numită matricea cameră sau matricea de vizualizare. Se construiește ținând cont de poziția camerei, de orientarea acesteia și de vectorul "UP" specificat de utilizator.

Matricea de vizualizare poate fi construită utilizând următoarea funcție GLM:

```
glm::lookAt(glm::vec3 camPosition, glm::vec3 viewPoint, glm::vec3 UP)
```

unde:

- camPosition – un vec3 specificând coordonatele 3D ale poziției camerei;
- viewPoint – un vec3 specificând coordonatele 3D ale punctului însprecare este îndreptată camera. Rețineți că vectorul definit ca viewPoint - camPosition specifică direcția de vizualizare;
- UP – vectorul "UP" – specifică direcția axei Y și este de obicei egală cu (0,1,0);

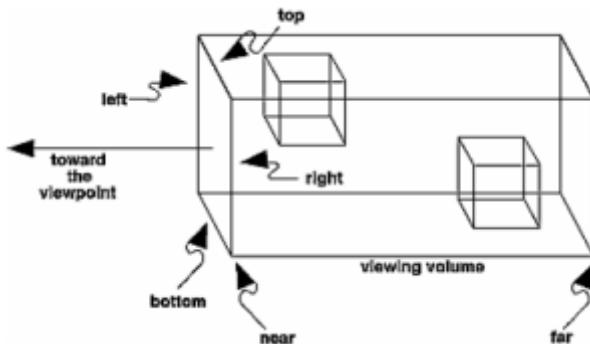
### 3.3 Transformări de proiecție

Aceste transformări sunt necesare pentru a crea o reprezentare 2D dintr-o porțiune a scenei 3D de obiecte definite în cadrul proiectului dvs. "Porțiunea" scenei care contribuie la reprezentarea 2D obținută este delimitată de un volum de vizualizare. Aceasta, la rândul său, depinde de parametrii cum ar fi poziția spectatorului, câmpul vizual și direcția vectorului de vizualizare. În termeni simpli, vă puteți gândi la o proiecție ca la umbra pe care un obiect o aruncă pe un plan sau o suprafață 2D.

Deși există un număr de tipuri diferite de proiecții, cele mai frecvent folosite sunt proiecțiile ortografice și de perspectivă.

#### 3.3.1 Proiecție ortografică

O transformare de proiecție ortografică definește volumul de vizionare ca formă cubică, folosind patru parametri: lățimea, înălțimea, un plan apropiat și un plan îndepărtat. Fiecare vîrf 3D va fi reprezentat ca o poziție determinată de coordonatele X și Y ale acestuia.



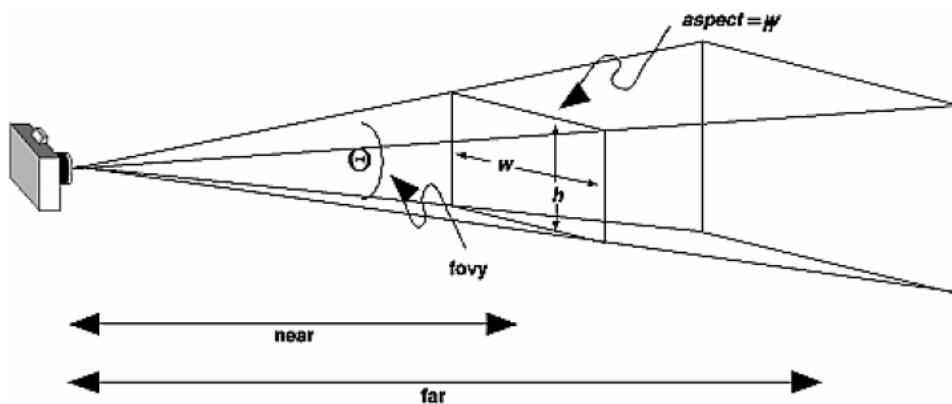
Puteți crea o matrice de proiecție ortografică utilizând funcția `glm::ortho`, văzută mai jos:

```
glm::ortho(GLint left, GLint right, GLint bottom, GLint top, GLfloat near, GLfloat far)
```

Fiecare parametru al acestei funcții este o variabilă ce definește limitele X, Y și Z ale volumului de vizualizare.

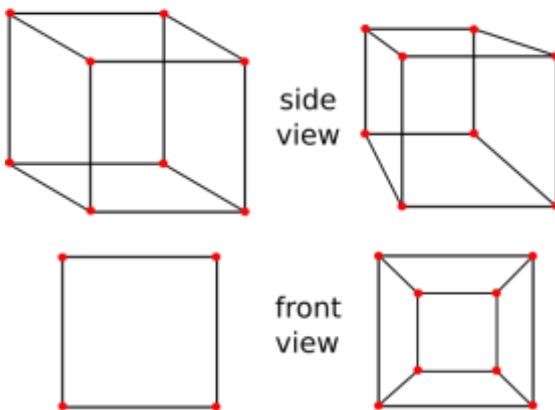
### 3.3.2 Proiecție perspectivă

Transformarea de perspectivă definește volumul de vizionare sub formă de frustum, pornind de la poziția camerei și extinzându-se spre exterior, de-a lungul axei Z.



În acest caz, toate cele trei coordonate ale vârfului afectează coordonatele 2D ale proiecției sale. Rezultatul net este că, pe măsură ce un obiect este mai departe de vizualizator, cu atât mai mic va apărea. Acest lucru îi va da privitorului o senzație de distanță în ceea ce privește obiectele din scenă.

Orthographic                      Perspective



Puteți crea o matrice de transformare perspectivă utilizând următoarea funcție GLM:

```
glm::perspective(GLfloat fov, GLfloat aspect, GLfloat near, GLfloat far);
```

unde:

- **fov** – unghiul care definește cât de mare este frustumul de vizionare (de obicei 45 de grade)
- **aspect** – definește raportul de aspect și este egal cu lățimea ferestrei de vizualizare împărțită la înălțimea acesteia
- **near & far** – coordonatele Z ale planurilor apropiate și îndepărtate, care sunt capetele frustumului vizual

## 4 Tutorial

Trebuie să creăm matricele de transformare pentru fiecare din cele trei etape:

- Transformări de model
- Transformări de vizualizare/cameră
- Transformări de proiecție

După definirea matricelor, le vom folosi pentru a transforma fiecare vârf al scenei. Pentru a face acest lucru, vom aplica cele trei transformări în interiorul Vertex Shader-ului. După aplicarea transformărilor, vârful rezultat va arăta astfel:

$$\mathbf{P}_{\text{final}} = \mathbf{M}_{\text{projection}} * \mathbf{M}_{\text{view}} * \mathbf{M}_{\text{model}} * \mathbf{P}_{\text{initial}}$$

**Rețineți că ordinea în care transformările sunt aplicate la vârf este contrarui ordinii în care apar în formula de mai sus.** Pentru a obține ordinea corectă în care sunt aplicate la vârf, trebuie să citiți transformările de la dreapta la stânga.

### 4.1 Transformări de model

Începeți prin descărcarea aplicației de pe site-ul web al laboratorului. Construiți și executați aplicația.

Ar trebui să vezi un pătrat purpuriu. Acesta este de fapt un cub multi-colorat, vazut din fata. Pentru a-l percepe ca pe un cub, vom adăuga mai întâi o transformare de rotație.

Așa cum am menționat mai devreme, transformările de modelare ar trebui să fie aplicate pe vârfuri în cadrul **Vertex Shader-ului**. În acest scop, îi vom trimite matricea de transformare a modelului ca variabilă uniformă:

```
#version 400

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

out vec3 colour;
```

```
uniform mat4 model;

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

Trebuie să trimitem matricea modelului din programul principal către vertex shader. Declarați matricea model și locația sa ca variabile globale:

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

gps::Shader myCustomShader;

glm::mat4 model;
GLint modelLoc;
```

Creați și trimiteți matricea model:

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

    . . . .

    model = glm::mat4(1.0f);
    modelLoc = glGetUniformLocation(myCustomShader.shaderProgram, "model");
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

    while (!glfwWindowShouldClose(glWindow)) {
        renderScene();

        glfwPollEvents();
        glfwSwapBuffers(glWindow);
```

```

    }

    //close GL context and any other GLFW resources
    glfwTerminate();

    return 0;
}

```

Să adăugăm o transformare de rotație în interiorul "renderScene ()". Aceasta se va activa la apăsarea tastei "R":

```

float angle=0;

void renderScene()
{
    . . . . .

    model = glm::mat4(1.0f);

    if (glfwGetKey(glWindow, GLFW_KEY_R)) {
        angle += 0.0002f;
    }

    // create rotation matrix
    model = glm::rotate(model, angle, glm::vec3(0, 1, 0));
    // send matrix data to vertex shader
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

    myCustomShader.useShaderProgram();
    glBindVertexArray(objectVAO);
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}

```

Încercați rotația apăsând tasta "R".

## 4.2 Transformări de cameră și proiecție

Următorul pas este să aplicăm transformările de vizualizare și de proiecție pe scena noastră de obiecte. Aceasta va permite o reprezentare mai realistă și mai controlabilă a scenei.

Începeți prin adăugarea a încă două matrici în interiorul vertex shader:

```
#version 400

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

out vec3 colour;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

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

Acum trebuie să definim, să inițializăm și să trimitem cele două matrice la vertex shader. Adăugați următoarele:

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

    . . .

    model = glm::mat4(1.0f);
    modelLoc = glGetUniformLocation(myCustomShader.shaderProgram, "model");
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

    // initialize the view matrix
```

```

        glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 5.0f), glm::vec3(0.0f, 0.0f, -10.0f), glm::vec3(0.0f, 1.0f,
0.0f));
        // send matrix data to shader
        GLint viewLoc = glGetUniformLocation(myCustomShader.shaderProgram, "view");
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));

        // initialize the projection matrix
        glm::mat4 projection = glm::perspective(70.0f, (float)glWindowWidth / (float)glWindowHeight, 0.1f, 1000.0f);
        // send matrix data to shader
        GLint projLoc = glGetUniformLocation(myCustomShader.shaderProgram, "projection");
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));
        . . .
    }
}

```

### 4.3 Adăugarea transformărilor

Desenați un al doilea cub și plasați-l în partea stângă a primului. În mod implicit, un obiect va fi întotdeauna centrata la origine. Pentru a evita acest lucru, efectuăm o translație înainte de a desena cel de-al doilea cub:

```

void renderScene()
{
    . . .

    // create rotation matrix
    model = glm::rotate(model, angle, glm::vec3(0, 1, 0));
    // send matrix data to vertex shader
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

    myCustomShader.useShaderProgram();
    glBindVertexArray(objectVAO);
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}

```

```

// create a translation matrix
model = glm::translate(model, glm::vec3(2, 0, 0));
// send matrix data to vertex shader
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
// draw the second cube
myCustomShader.useShaderProgram();
glBindVertexArray(objectVAO);
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}

```

Cubul va fi desenat tot în origine, dar transformarea de translație tocmai a mutat originea cu două unități la dreapta.

Efectuați o rotație ținând apăsată tasta "R". Observați cum se aplică transformarea pe întreaga scenă.

Acum, să aplicăm o rotație pe cel de-al doilea cub:

```

void renderScene()
{
    . . .

    // create rotation matrix
    model = glm::rotate(model, 0.3f, glm::vec3(0, 1, 0));
    // send matrix data to vertex shader
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

    // create a translation matrix
    model = glm::translate(model, glm::vec3(2, 0, 0));
    // send matrix data to vertex shader
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    // draw the second cube
    myCustomShader.useShaderProgram();
    glBindVertexArray(objectVAO);
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}

```

## 5 Temă

1. Controlați rotația celui de-al doilea cub folosind tasta "T".
2. Modificați rotația celui de-al doilea cub rotindu-l în jurul centrului său - punctul (2, 0, 0).  
\* Sfat: utilizați secvența de operații  $T_{\text{inverse}} * R * T$
3. Implementați clasa Camera (pornire de la Camera.hpp prezentă pe site-ul web). Controlați camera utilizând intrările de la tastatură și mouse. **Sugestie:** Înregistrați câteva funcții „callback” pentru a procesa intrările de la tastatură și mouse:

```
void keyboardCallback(GLFWwindow* window, int key, int scancode, int action, int mode);
void mouseCallback(GLFWwindow* window, double xpos, double ypos);

...
glfwSetKeyCallback(glWindow, keyboardCallback);
glfwSetCursorPosCallback(glWindow, mouseCallback);
glfwSetInputMode(glWindow, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

# Laborator 6

---

## 1 Obiective

Acest laborator vă prezintă noțiunile de bază privind conceptul de textură și utilizarea sa într-o scenă 3D a obiectelor. Acesta vă va oferi, de asemenea, un cadru de bază pentru încărcarea și includerea obiectelor 3D în proiectul OpenGL, prezentând apoi câteva notiuni despre animația obiectelor.

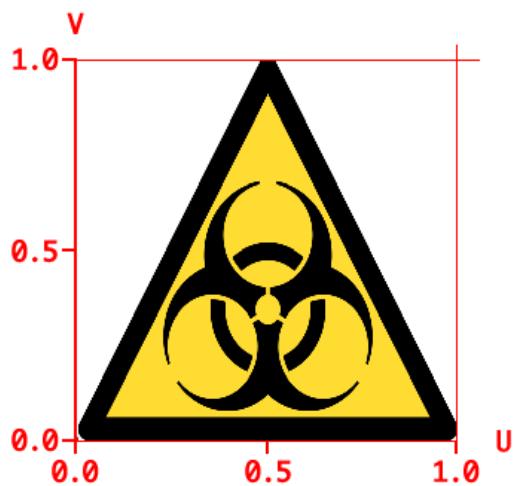
## 2 Fundament teoretic

Puteți să vă gândiți la o textură ca la o imagine 2D care poate fi aplicată pe un obiect 3D. Deși este posibil să existe texturi 1D și 3D, acestea sunt dincolo de sfera de interes a acestui laborator.

În general, texturile sunt aplicate pe obiecte 3D pentru a crește fotorealismul, adăugând detalii pe suprafețele lor. Spre exemplu, este posibil să se adauge detalii de culoare unui model prin aplicarea unor culori asociate fiecărui vârf (conform exemplului din laboratorul precedent), dar acest lucru ar fi foarte risipitor, atât în ceea ce privește volumul de date, cât și puterea de procesare necesară.

Texturile sunt în principiu hărți de pixeli 2D încărcate în memoria video, care pot fi apoi asociate unui obiect 3D dat. Asocierea se face prin atribuirea fiecărui vârf 3D unui punct din cadrul imaginii 2D. Spațiul de textură 2D este de obicei exprimat folosind numele parametrilor "u" și "v", spre deosebire de "x" și "y", pentru a evita confuzia cu valorile poziției vârfului.

Spre deosebire de imaginile normale ale căror coordonate sunt numere întregi care exprimă valoarea unui pixel dat, coordonatele UV ale unei texturi sunt valori în virgulă variabilă în intervalul [0, 1], unde punctul (0, 0) reprezintă colțul din stânga jos al imaginii, iar (1, 1) reprezintă colțul din dreapta sus. Aceste coordonate pot fi transformate în pixeli prin înmulțirea valorilor lor cu lățimea și înălțimea imaginii.



Pentru a utiliza o textură, trebuie să o încărcați mai întâi dintr-un fișier extern:

- Încărcați datele de imagine dintr-un fișier imagine (de preferință ".png" sau ".tex" - întotdeauna cu lățime și înălțime egale cu o putere de 2 - ex: 128, 256, 512 etc.)
- Generați un ID de textura - folosind funcția: "glGenTextures"
- Legați sau "activați" textura - folosind funcția "glBindTexture"
- Generați o textură din datele imaginii - utilizând funcția "glTexImage2D"
- Configurați parametrii de textură

Funcția de încărcare a texturii este implementată în cadrul "ReadTextureFromFile". Examinați această funcție. Pentru a aplica textura pe un obiect, este suficient să o legați (bind) înainte de a desena obiectul - folosind funcția "glBindTexture".

## 3 Tutorial

### 3.1 Aplicarea texturii

În scopul demonstrării utilizării texturilor, vom încerca mai întâi un exemplu simplu.

Să începem prin desenarea a două triunghiuri. Definiți datele de vârf - poziția și coordonatele texturii:

```
//vertex position and UV coordinates
GLfloat vertexData[] = {
    // primul triunghi
    -5.0f, 0.0f, 0.0f,    0.0f, 0.0f,
    5.0f, 0.0f, 0.0f,    1.0f, 0.0f,
    0.0f, 8.0f, 0.0f,    0.5f, 1.0f,
    // al doilea triunghi
    0.1f, 8.0f, 0.0f,    0.0f, 0.0f,
    5.1f, 0.0f, 0.0f,    0.0f, 0.0f,
    10.1f, 8.0f, 0.0f,   0.0f, 0.0f
};
GLuint vertexIndices[] = {
    0,1,2,
    3,4,5
};
```

Încărcați datele triunghiului apelând funcția predefinită "loadTriangles ()" - înainte de începerea buclei de rasterizare:

```
int main(int argc, const char * argv[]) {
    .....
    loadTriangleData();
    while (!glfwWindowShouldClose(glWindow)) {
        renderScene();
```

```

        glfwPollEvents();
        glfwSwapBuffers(glWindow);
    }
    .....
}
```

În cele din urmă, desenați triunghiurile:

```

void renderScene()
{
    .....
    glBindVertexArray(objectVAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
}
```

Construiți și executați aplicația. Ar trebui să vedeați două triunghiuri negre.

Pentru a aplica textura, trebuie să trimitem coordonatele texturii către Fragment Shader. Acest lucru înseamnă că ele trebuie să treacă mai întâi prin Vertex Shader:

```

#version 400

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexNormal;
layout(location = 2) in vec2 textcoord;

out vec3 colour;
out vec2 passTexture;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    colour = vertexNormal;
    passTexture = textcoord;
    gl_Position = projection * view * model * vec4(vertexPosition, 1.0);
}
```

În interiorul Fragment Shader, avem nevoie de acces la o unitate de textură, pe care o să o denumim "diffuseTexture". Funcția "texture" va returna o valoare din textura activă curentă, pe baza coordonatelor 2D UV pe care le primește ca argumente.

Fragment Shader ar trebui să arate aşa:

```

#version 400

in vec3 colour;
```

```
in vec2 passTexture;  
  
out vec4 fragmentColour;  
  
uniform sampler2D diffuseTexture;  
  
void main() {  
    fragmentColour = texture(diffuseTexture, passTexture);  
}
```

Asigurați-vă că încărcați textura din fișierul imagine:

```
int main(int argc, const char * argv[]) {  
    .....  
  
    loadTriangleData();  
    texture = ReadTextureFromFile("textures\\hazard2.png");  
  
    while (!glfwWindowShouldClose(glWindow)) {  
        renderScene();  
  
        glfwPollEvents();  
        glfwSwapBuffers(glWindow);  
    }  
    .....  
}
```

Acum activați textura înainte de a desena triunghiurile:

```
void renderScene()  
{  
    .....  
  
    glBindVertexArray(objectVAO);  
    glActiveTexture(GL_TEXTURE0);  
    glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "diffuseTexture"), 0);  
    glBindTexture(GL_TEXTURE_2D, texture);  
  
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);  
}
```

Construiți și executați aplicația.

Modificați coordonatele UV ale texturii pentru al doilea triunghi, astfel încât aplicația dvs. să arate astfel:



## 3.2 Încărcarea obiectelor 3D

În scopul încărcării și desenării obiectelor 3D complexe, o opțiune pe care o aveți este să utilizați clasa "Model3D" furnizată în pachetul de resurse al acestui laborator. Această clasă ar trebui să poată încărca orice fel de obiect, de la obiecte simple la scene complexe, cu multiple texturi aplicate. După cum puteți vedea în interiorul "Model3D.hpp", acesta conține o listă de obiecte de tip „mesh” (fiecare reprezentând câte un obiect 3D) precum și colecția de texturi încărcate - necesare pentru acele obiecte.

Să vedem cum putem folosi clasa "Model3D" pentru a încărca un model dintr-un fișier ".obj". Modificați funcția principală. Stergeți funcționalitatea adăugată anterior și adăugați:

```
int main(int argc, const char * argv[]) {  
    .....  
    //send matrix data to shader  
    GLint projLoc = glGetUniformLocation(myCustomShader.shaderProgram, "projection");  
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));  
  
    myModel.LoadModel("objects\\stall.obj");  
  
    while (!glfwWindowShouldClose(glWindow)) {  
        renderScene();  
  
        glfwPollEvents();  
        glfwSwapBuffers(glWindow);  
    }  
    .....  
}
```

Pentru a rasteriza modelul încărcat, modificați și adăugați următoarele linii în cadrul funcției "renderScene":

```
void renderScene()  
{  
    .....  
    //create rotation matrix  
    model = glm::rotate(model, angle, glm::vec3(0, 1, 0));  
    model = glm::rotate(model, 3.14f, glm::vec3(0, 1, 0));  
    //send matrix data to vertex shader
```

```

    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    myModel.Draw(myCustomShader);
}

```

Construiți și executați aplicația. După cum puteți vedea, **modelul nu este texturat**. Pentru a aplica o textură, trebuie să încărcați o textură dintr-un fișier extern:

```

int main(int argc, const char * argv[]) {
    .....
    myModel.LoadModel("objects\\stall.obj");
    texture = ReadTextureFromFile("textures\\stall_texture.png");
    .....
}

```

Activați textura înainte de a desena modelul:

```

void renderScene()
{
    .....
    glBindTexture(GL_TEXTURE_2D, texture);

    glActiveTexture(GL_TEXTURE0);
    glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "diffuseTexture"), 0);
    myModel.Draw(myCustomShader);
}

```

Rulați aplicația. Utilizați tastele "W", "A", "S" și "D" pentru navigare; "Q" și "E" pentru a roti modelul.

Amintiți-vă când am spus că un obiect Model3D poate reține atât obiectele „mesh” 3D cât și texturile lor. Cel mai adesea, atunci când descărcați obiecte 3D din surse online, o să primiți un fișier ".obj" și un ".mtl", împreună cu o colecție de imagini cu textură deja aplicată. Fișierul .mtl specifică datele de asociere ale texturii pentru obiectul dat. În aceste cazuri, obiectul Model3D va citi obiectul 3D și va activa texturile asociate după cum este necesar - **NU VA MAI FI NEVOIE CA DVS. SĂ ÎNCĂRCATI SI SĂ ACTIVATI TEXTURILE.**

Puteți găsi fișiere ".obj" la [www.turbosquid.com](http://www.turbosquid.com), [www.tf3dm.com](http://www.tf3dm.com) și <https://sketchfab.com/features/free-3d-models>.

Să încercăm să adăugăm un fișier ".obj" cu texturile incluse. Modificați funcția principală pentru a încărca noul obiect:

```

int main(int argc, const char * argv[]) {
    .....
    //argument 1 = obj file; argument 2 = textures folder
    myModel.LoadModel("objects\\Farmhouse.obj", "textures\\");
    //texture = ReadTextureFromFile("textures\\stall_texture.png");

    while (!glfwWindowShouldClose(glWindow)) {
        renderScene();
    }
}

```

```

        glfwPollEvents();
        glfwSwapBuffers(glWindow);
    }
    .....
}

```

Comentați instrucțiunile de activare a texturii din renderScene. Acest obiect conține informații de mapare a texturii, prin urmare obiectul Model3D va încerca să încarce automat texturile:

```

void renderScene()
{
    .....
    //glActiveTexture(GL_TEXTURE0);
    //glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "diffuseTexture"),0);
    // glBindTexture(GL_TEXTURE_2D, texture);

    myModel.Draw(myCustomShader);
}

```

Regulile generale de urmat la încărcarea fișierelor .obj de pe internet:

- Căutați fișierul .mtl - conține date despre materialele și texturile obiectului. În interiorul .obj ar trebui să aveți o referință la fișierul .mtl.
- În cadrul fișierului .mtl, **căutați referințele fișierelor de textură** => asigurați-vă ca acestea nu conțin si nume de folder, doar numele fișierului
- Asigurați-vă că texturile sunt în format ".png" sau ".tga" și au dimensiuni (lățime și înălțime) egale cu puterile de 2 (ex: 128, 256, 512 etc.).
- Încercați să păstrați proiectul curat și ordonat - păstrați obiectele 3D și texturile (împreună cu fișierul .mtl) în propriile lor foldere, separate.

### 3.3 Animation

Să adăugăm o transformare de translație obiectului nostru - translatăm pe axa Ox cu o cantitate egală cu "delta". Creșteți delta în interiorul funcției "renderScene".

```

float delta = 0;
void renderScene()
{
    .....
    delta += 0.001;
    model = glm::translate(model, glm::vec3(delta, 0, 0));

    //create rotation matrix
    model = glm::rotate(model, angle, glm::vec3(0, 1, 0));
    model = glm::rotate(model, 3.14f, glm::vec3(0, 1, 0));
    //send matrix data to vertex shader
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

    glBindTexture(GL_TEXTURE_2D, texture);
    myModel.Draw(myCustomShader);
}

```

```
}
```

Rulați aplicația. Încercați apoi să desenați 500 de instanțe ale aceluiași obiect. Observați scăderea vitezei de animație:

```
void renderScene()
{
    .....
    for (int i = 0; i < 500;i++)
        myModel.Draw(myCustomShader);
}
```

Această animație depinde de frecvența cu care se invocă funcția - adică de frecvența de refresh a aplicației. Așa cum ati putea să ghiciți, frecvența de refresh variază în funcție de resursele hardware sau de complexitatea scenelor. În mod ideal, am dori ca animația să funcționeze independent de capacitatele de refresh ale aplicației noastre.

Soluția este de a crește delta pe baza unei variabile de viteză date - "unități pe secundă". Măsurăm timpul dintre actualizările succesive și calculăm distanța pe care ar trebui să o deplaseze obiectul:

$$Distance = speed * time$$

```
float delta = 0;
float movementSpeed = 2; // units per second
void updateDelta(double elapsedSeconds) {
    delta = delta + movementSpeed * elapsedSeconds;
}
double lastTimeStamp = glfwGetTime();

void renderScene()
{
    .....
    //initialize the view matrix
    glm::mat4 view = myCamera.getViewMatrix();
    //send matrix data to shader
    GLint viewLoc = glGetUniformLocation(myCustomShader.shaderProgram, "view");
    glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));

    // get current time
    double currentTimeStamp = glfwGetTime();
    updateDelta(currentTimeStamp - lastTimeStamp);
    lastTimeStamp = currentTimeStamp;
    model = glm::translate(model, glm::vec3(delta, 0, 0));
    .....
}
```

Puteți încerca animația cu un număr tot mai mare de instanțe de obiecte, pentru a vedea dacă viteza de mișcare se modifică sau nu.

## 4 Documentație

Funcție (și legătură)	Descriere
<a href="#">glGenTextures</a>	Generează nume de texturi
<a href="#">glBindTexture</a>	Legați o textură numită într-o textură țintă
<a href="#">glTexImage2D</a>	Specificați o imagine textura bidimensională
<a href="#">glTexParameter</a>	Setați parametrii de textură

## 5 Temă

- Încărcați cel puțin 3 modele. Schimbați poziția unui obiect folosind tastele și animați un alt obiect.

# Laborator 7

## 1 Obiective

Obiectivul acestui laborator este de a descrie pe scurt modul de adăugare a iluminării în scena dvs. OpenGL 4.0. Vom prezenta noțiunile matematice de bază din spatele iluminatului OpenGL și structura programelor Vertex și Fragment Shader pentru iluminarea direcțională per-vârf și per-pixel.

## 2 Fundament teoretic

Modelul de iluminare din OpenGL cu pipeline fix (modelul Gouraud) reprezintă modelul de bază pentru aplicații grafice și este o încercare decentă de a aproxima modul în care funcționează iluminarea din lumea reală. Cu toate acestea, funcționalitatea din pipeline-ul fix vine cu anumite constrângeri, fiind lipsită de realism și de opțiuni atunci când vine vorba de compromisuri calitate-performață. Shaderele programabile pot să ofere rezultate mult mai bune, mai ales d.p.d.v. al realismului. Cu toate acestea, este destul de important să înțelegem foarte bine modelul de bază OpenGL, deoarece acest model de iluminare furnizează în continuare fundamentul pe care sunt construite modelele avansate.

Modelul de iluminare OpenGL cu pipeline fix însumează un set de componente independente pentru a obține efectul de iluminare general al unui punct de pe suprafața unui obiect. Aceste componente pot fi observate în Figura 1 de mai jos.

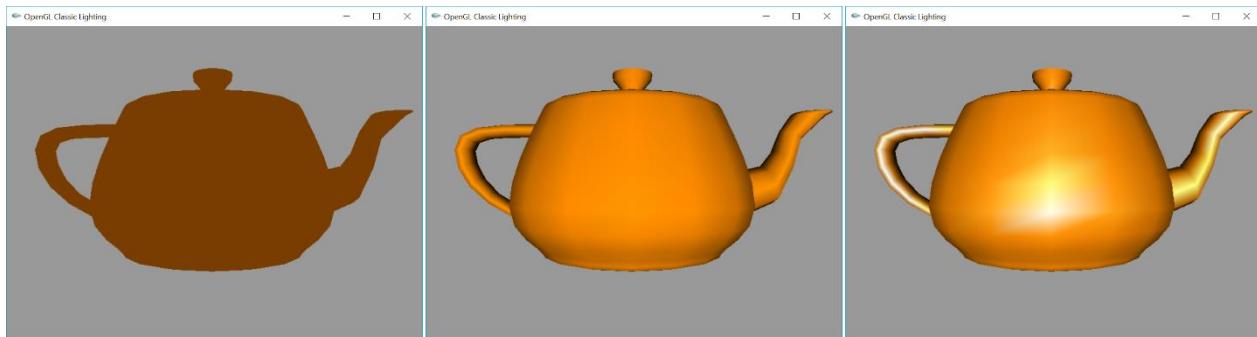


Figura 1 - Componente de lumină OpenGL (de la stânga la dreapta: Ambient, Ambient + Difuz, Ambient + Difuz + Specular)

*Lumina ambientală* nu vine dintr-o anumită direcție, fiind o aproximare decentă a luminii care există împrăștiată în jurul unei scene. Calculul său nu depinde de poziția spectatorului sau de direcția luminii. Valoarea iluminării ambientale poate fi precalculată ca efect global sau adăugată independent pentru fiecare sursă de lumină.

*Lumina difuză* este împrăștiată în mod egal în toate direcțiile pentru o sursă de lumină. Cantitatea de lumină difuză care există pe un punct nu depinde de poziția spectatorului, dar depinde de direcția luminii. Intensitatea luminoasă este mai puternică pe suprafețele care sunt orientate spre sursa de lumină și mai slabă pe cele orientate în direcția opusă. Calculul luminii difuze depinde de normala suprafeței și de direcția sursei de lumină, dar nu și de direcția de vizualizare.

*Lumina speculară* este lumina reflectată direct de către suprafață și se referă la cât de asemănătoare este suprafața (materialul) obiectului cu o oglindă. Intensitatea acestui efect este denumită Shininess (stralucire). Calculul său necesită să se știe cât de apropiată este orientarea suprafeței de reflecția dintre direcția luminii și ochi. Astfel, calculul luminii speculare depinde de normala suprafeței, de direcția luminii și de direcția de vizionare.

### 3 Implementarea funcționalității de iluminare în OpenGL 4

Acum vom construi treptat shaderele necesare pentru implementarea modelului de iluminat clasic OpenGL (iluminat Gouraud) folosind limbajul GLSL. Deoarece modelul clasic calculează iluminarea pentru fiecare vârf, shaderul pe care îl vom schimba treptat este Vertex Shader. Fragment Shader va rămâne același pe parcursul acestui proces, aşa cum este prezentat mai jos.

```
//pass-through fragment shader for OpenGL lighting
#version 410 core

uniform vec3 baseColor;

in vec3 color;

out vec4 fColor;

void main()
{
    fColor = vec4(color, 1.0f);
}
```

În teorie, calculul iluminării poate fi efectuat în orice spațiu (sistem de coordonate) din pipeline-ul de transformare. Cu toate acestea, pentru rezultate corecte (și astfel mai realiste), iluminarea ar trebui să fie efectuată în spațiul de coordonate globale sau cel de vizualizare. Ambele vor oferi rezultate de iluminare identice. Pentru acest laborator, vom efectua calculul iluminării în spațiul de vizualizare.

#### 3.1 Lipsa efectului de iluminare

Atunci când nu se utilizează nici un model de iluminare, OpenGL redă fiecare obiect în mod uniform, folosind culoarea sa de bază. Toate efectele de iluminare vor modula această culoare, folosind-o ca bază pentru următoarele calcule de iluminare. Vertex Shader necesar pentru desenarea unui obiect fără efecte de iluminare este prezentat mai jos. Pentru acest laborator, culoarea de bază este furnizată din aplicație ca valoare **uniform**. În aplicațiile din lumea reală, culoarea de bază este, de obicei, furnizată ca atribut al vârfului.

```
//vertex shader for no lighting
#version 410 core

layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vNormal;

//matrices
uniform mat4 model;
```

```

uniform mat4 view;
uniform mat4 projection;

//lighting
uniform vec3 lightColor;
uniform vec3 baseColor;

out vec3 color;

void main()
{
    //compute final vertex color
    color = baseColor;

    //transform vertex
    gl_Position = projection * view * model * vec4(vPosition, 1.0f);
}

```

### 3.2 Iluminarea ambientală

Lumina ambientală este uniformă și, prin urmare, nu se schimbă între vârfuri. Putem trimite componenta Ambientală din aplicație ca o variabilă **uniform**. Rezultatul interacțiunii dintre lumina ambientală și culoarea suprafeței pe care o lovește este modelat prin înmulțire. În timpul diferitelor înmulțiri necesare pentru a calcula efectele diferitelor componente luminoase pe o suprafață, valorile componentelor pot să depășească valoarea de 1,0. Cu toate acestea, culoarea finală trebuie să fie saturată la alb (adică nici o componentă de lumină nu trebuie să fie mai mare de 1,0). Din acest motiv se utilizează funcția **min** pentru a atinge această saturăție. Adăugările și modificările la vertex shader pentru încorporarea luminii ambientale sunt evidențiate mai jos.

```

//vertex shader for ambient lighting
#version 410 core

layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vNormal;

//matrices
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

//lighting
uniform vec3 lightColor;
uniform vec3 baseColor;

out vec3 color;

vec3 ambient;
float ambientStrength = 0.2f;

void main()
{
    //compute ambient light
}

```

```

        ambient = ambientStrength * lightColor;

        //compute final vertex color
        color = min(ambient * baseColor, 1.0f);

        //transform vertex
        gl_Position = projection * view * model * vec4(vPosition, 1.0f);
    }

```

### 3.3 Iluminarea ambientală și difuză

Iluminarea difuză depinde de direcția luminii care atinge punctul pe care îl calculăm în prezent. Deoarece lumina noastră este una **direcțională** (situată la infinit, spre deosebire de sursele de lumină punctiforme care au o poziție 3D definită), toate razele luminoase sunt paralele și au aceeași direcție. Aceasta înseamnă că direcția luminii este constantă pe toate vârfurile și poate fi calculată în aplicație și trimisă ca o variabilă **uniform** în shader. Cantitatea de lumină difuză împrăștiată depinde de orientarea suprafeței (adică de unghiul dintre direcția luminii și normala suprafeței). Cantitatea maximă de lumină împrăștiată este obținută atunci când lumina atinge perpendicular suprafața (adică unghiul dintre normală și direcția luminii este 0,0, astfel cosinusul este 1,0). Pe măsură ce acest unghi crește (suprafața se îndepărtează de lumină), cosinusul scade spre 0,0, astfel încât cantitatea de lumină difuză împrăștiată este mai mică. Cosinul unghiului dintre doi vectori normalizați poate fi calculat folosind produsul scalar (dot product).

Normalele vârfurilor sunt transmise din aplicație (ele sunt pre-calculate în modelul GLM) ca atribut per-vârf (similar cu pozițiile vârfurilor). Deoarece modelul este afectat de transformări ( fiecare vârf este înmulțit cu matricea Model, Vedere și Proiecție), normala trebuie supusă acelorași transformări ca și vârful obiectului, altfel iluminarea nu va funcționa corect (adică normala trebuie să se miște împreună cu obiectul). În OpenGL, normalele nu sunt transformate folosind matricea Model-View-Projection, ci folosind aşa-numita Matrice Normală. Matricea normală este matricea 3x3 din stânga-sus a inversului transpus al matricei model (dacă se efectuează calculul luminii în spațiul global) sau al matricei Model-View (dacă se efectuează calculul iluminării în spațiul de vizualizare). Aceasta poate fi ușor calculată în aplicație utilizând biblioteca OpenGL matematică (GLM). După calculare, matricea normală este trimisă shader-ului ca variabilă **uniform**. Adăugările la vertex shader pentru a încorpora lumina difuză sunt evidențiate mai jos.

```

//vertex shader for ambient and diffuse lighting
#version 410 core

layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vNormal;

//matrices
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat3 normalMatrix;

//lighting
uniform vec3 lightDir;
uniform vec3 lightColor;

```

```

uniform vec3 baseColor;

out vec3 color;

vec3 ambient;
float ambientStrength = 0.2f;
vec3 diffuse;

void main()
{
    //compute ambient light
    ambient = ambientStrength * lightColor;

    //normalize the light's direction
    vec3 lightDirN = normalize(lightDir);

    //compute eye coordinates for normals (transform normals)
    vec3 normalEye = normalize(normalMatrix * vNormal);

    //compute diffuse light
    diffuse = max(dot(normalEye, lightDirN), 0.0f) * lightColor;

    //compute final vertex color
    color = min((ambient + diffuse) * baseColor, 1.0f);

    //transform vertex
    gl_Position = projection * view * model * vec4(vPosition, 1.0f);
}

```

### 3.4 Iluminarea ambientală, difuză și speculară

Iluminarea speculară depinde atât de direcția luminii, cât și de direcția de vizualizare. Componenta speculară depinde de cosinusul unghiului ( $\alpha$ ) dintre direcția de vizualizare (ochi) și reflexia provenită de la lumina (R) (figura 2). Deoarece efectuăm calculul iluminării în coordonatele camerei (de vizualizare), camera este situată la origine, astfel poziția ochiului este (0, 0, 0).

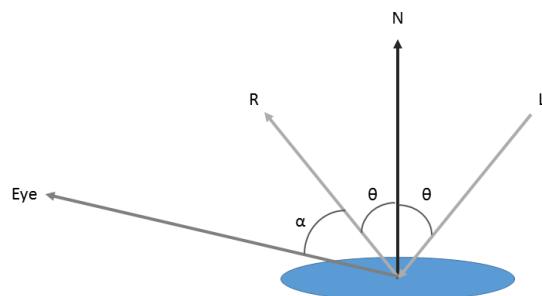


Figura 1 – Calculul iluminării speculare

Shininess (strălucirea) este folosit ca un exponent pentru a obține o măsură a clarității căderii unghiulare dintr-o reflecție directă. Cu cât este mai mare exponentul, cu atât devine mai intensă iluminarea speculară, deoarece ridicarea unui număr mai mic decât 1.0 la o putere exponențială scade valoarea sa. Unghiurile de reflexie apropiate de 0.0 vor duce la valori apropiate de 1.0 pentru iluminarea speculară, în timp ce pentru alte unghiuri aceasta se va îndrepta rapid înspre 0.0. Deoarece lumina speculară reprezintă lumina

reflectată, componenta speculară nu mai este modulată cu culoarea obiectului. Modificările din vertex shader pentru a încorpora iluminarea speculară sunt evidențiate mai jos.

```
//vertex shader for ambient, diffuse and specular lighting
#version 410 core

layout(location=0) in vec3 vPosition;
layout(location=1) in vec3 vNormal;

//matrices
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat3 normalMatrix;

//lighting
uniform vec3 lightDir;
uniform vec3 lightColor;
uniform vec3 baseColor;

out vec3 color;

vec3 ambient;
float ambientStrength = 0.2f;
vec3 diffuse;
vec3 specular;
float specularStrength = 0.5f;
float shininess = 32.0f;

void main()
{
    //compute ambient light
    ambient = ambientStrength * lightColor;

    //normalize the light's direction
    vec3 lightDirN = normalize(lightDir);

    //compute eye coordinates for normals
    vec3 normalEye = normalize(normalMatrix * vNormal);

    //compute diffuse light
    diffuse = max(dot(normalEye, lightDirN), 0.0f) * lightColor;

    //compute the vertex position in eye coordinates
    vec4 vertPosEye = view * model * vec4(vPosition, 1.0f);

    //compute the view (Eye) direction (in eye coordinates, the camera is at the origin)
    vec3 viewDir = normalize(- vertPosEye.xyz);

    //compute the light's reflection (the reflect function requires a direction pointing towards the vertex, not away
    from it)
    vec3 reflectDir = normalize(reflect(-lightDir, normalEye));

    //compute specular light
```

```

float specCoeff = pow(max(dot(viewDir, reflectDir), 0.0f), shininess);
specular = specularStrength * specCoeff * lightColor;

//compute final vertex color
color = min((ambient + diffuse) * baseColor + specular, 1.0f);

//transform vertex
gl_Position = projection * view * model * vec4(vPosition, 1.0f);
}

```

## 4 Tutorial

Descărcați resursele disponibile pe pagina web a laboratorului. Prima arhivă (Laboratory 7 resources - part 1.zip) conține o aplicație cu shaderele vertex și fragment necesare pentru a desena un ceainic fără iluminare (a se vedea subsecțiunea 3.1).

Extindeți vertex shader pentru a încorpora iluminatul ambiental în aplicația dvs. (vezi secțiunea 3.2). Aplicația dvs. ar trebui să afișeze ceainicul ca în figura 3 de mai jos.



Figura 2 –Rasterizare cu iluminare ambientală

Extindeți vertex shader pentru a încorpora iluminarea difuză în aplicația dvs. (vezi secțiunea 3.2).

Aplicația dvs. ar trebui să afișeze ceainicul ca în figura 4 de mai jos.



Figura 3 – Rasterizare cu iluminare ambientală și difuză

Extindeți vertex shader pentru a încorpora iluminarea speculară în aplicația dvs. (vezi secțiunea 3.3).

Aplicația dvs. ar trebui să afișeze ceainicul ca în figura 4 de mai jos.



Figura 4 – Rasterizare cu iluminare ambientală, difuză și speculară

## 5 Lectură suplimentară

- OpenGL Programming Guide 8<sup>th</sup> Edition – Capitolul 7

## 6 Temă

1. Implementați iluminatul per-vertex OpenGL (Gouraud) pentru aplicația dvs. urmând pașii descriși în secțiunea 4
2. Implementați modelul de iluminare per-pixel (Phong). Iluminarea per-pixel calculează culorile finale din fragment shader, folosind **normalele interpolate**, în loc să folosească normalele vârfului în vertex shader. Comparați calitatea generală a iluminării cu aceea a modelului per-vertex obținut la pasul 1, pentru toate rezoluțiile obiectului ceanic (există mai multe ceanice disponibile, generate folosind 4, 10, 20 și 50 de segmente). Pentru această exercițiu, utilizați resursele furnizate în cea de-a doua arhivă (Laboratory 7 resources - part 2.zip) și modificați setul de shader-e shaderPPL (vert / frag).
3. Extindeți implementarea iluminării per-pixel pentru a încorpora proprietățile materialelor pentru obiecte. Proprietățile materialelor pot fi trimise din aplicație ca variabile **uniform**, iar culoarea finală va fi modulată (înmulțită) cu componenta de material corespunzătoare (Ambient, Diffuse și Specular) în locul culorii de bază a obiectului.
4. Extindeți implementarea iluminării per-pixel pentru a încorpora intensitățile componentelor luminoase. În loc să aibă o lumină cu o singură culoare, folosind coeficienți “hard-codăți” pentru a decide contribuția fiecărui componentă a luminii, descrieți o sursă de lumină folosind trei componente diferite (ambient, difuz și specular) și folosiți-le pentru a calcula intensitățile finale ale culorii.

# Laborator 8

## 1 Obiective

Obiectivul acestui laborator este de a descrie succint câteva tehnici avansate de iluminare în OpenGL 4.

## 2 Modelul de iluminare Blinn-Phong

Modelul de iluminare Blinn-Phong este o versiune ușor modificată a modelului Phong și a fost dezvoltat de către Jim Blinn. Scopul său este de a îmbunătăți reflexiile speculare în anumite condiții, cum ar fi un coeficient foarte scăzut de strălucire (rezultând astfel o zonă speculară mare). Reflexiile speculare ale modelului Phong tind să fie întrerupte imediat ce unghiul dintre direcția de vizionare și reflexia luminii crește peste 90.0 grade (cosinusul devine negativ și astfel este fixat la 0.0).

Soluția propusă de Blinn este de a folosi un așa-zis **semi-vector** în locul vectorului de reflexie pentru a calcula componenta speculară. Semi vectorul (**H**) este un vector unitar la jumătatea distanței dintre direcția de vizionare și direcția luminii (Figura 1).

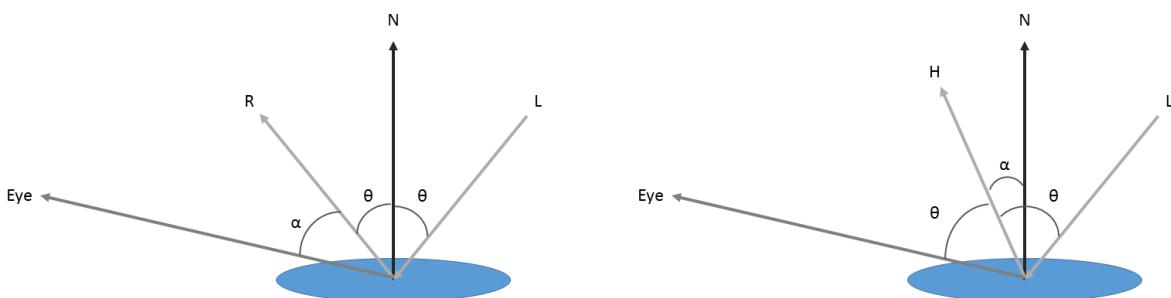


Figura 1 - Modelul specular Phong (stânga) versus modelul specular Blinn (dreapta)

Semi-vectorul este calculat ca:

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{Eye}}{\|\mathbf{L} + \mathbf{Eye}\|}$$

Acest lucru se traduce în codul GLSL după cum urmează:

```
//normalize light direction
vec3 lightDirN = normalize(lightDir);

//compute view direction
vec3 viewDirN = normalize(viewPosEye - fragPosEye.xyz);

//compute half vector
vec3 halfVector = normalize(lightDirN + viewDirN);
```

Componenta speculară este apoi calculată ca fiind cosinusul unghiului dintre normală și semi-vector:

```
//compute specular light
float specCoeff = pow(max(dot(normalEye, halfVector), 0.0f), shininess);
specular = specularStrength * specCoeff * lightColor;
```

Modelul Blinn-Phong îmbunătățește performanța, evitând calculul costisitor al vectorului de reflexie.

### 3 Lumini punctiforme

Laboratorul 8 a introdus conceptul de **lumini direcționale**, care sunt surse de lumină infinit de îndepărtate. Toate razele de lumină care provin dintr-o sursă de lumină direcțională sunt paralele și astfel direcția luminii implicate în calculul iluminării este constantă între obiecte și toate fragmentele acestora. Astfel, aceste surse de lumină sunt ideale pentru iluminarea globală din cadrul scenelor noastre.

**Luminile punctiforme** sunt surse de lumină cu o poziție dată care iluminează radial și uniform în toate direcțiile. Spre deosebire de luminile direcționale, razele generate de luminile punctiforme se estompează în funcție de distanță, făcând astfel obiectele mai apropiate de sursă să pară mai iluminate decât obiectele mai îndepărtate.

Atunci când se utilizează luminile punctiforme, direcția luminii nu este constantă între fragmente și trebuie calculată pentru fiecare fragment diferit:

```
//compute light direction
vec3 lightDirN = normalize(lightPosEye - fragPosEye.xyz);
```

Pentru a reduce intensitatea luminii punctiforme cu distanța, trebuie să aplicăm un coeficient de atenuare. Atenuarea poate fi calculată ca o funcție liniară sau patratică dependentă de distanță. Atenuarea liniară necesită un calcul mai puțin complex, dar oferă mai puțin realism, deoarece, în lumea reală, sursele de lumină sunt strălucitoare atunci când stau aproape de obiecte, dar își diminuează rapid strălucirea, cu cât ne aflăm mai departe de ele. Realismul este considerabil crescut atunci când se încorporează atenuarea patratică în funcție de distanță. Astfel, atenuarea poate fi calculată folosind:

$$Att = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

unde  $K_c$  este un termen constant, de obicei menținut la 1.0,  $K_l$  este un termen liniar,  $K_q$  este un termen patratic și  $d$  este distanța.

Alegerea valorilor potrivite pentru aceste constante depinde de mai mulți factori cum ar fi mediul, distanța necesară pentru călătoria luminii, tipul de lumină etc. Puteți folosi Tabelul 1 ca ghid pentru specificarea coeficienților de atenuare pe baza distanței pe care lumina trebuie să o parcurgă.

Pentru a implementa atenuarea, ar trebui să definim coeficienții de atenuare în shader sau să îi trimitem din aplicație prin variabile **uniform**:

```
float constant = 1.0f;
```

```
float linear = 0.0045f;
float quadratic = 0.0075f;
```

Distance	Constant	Linear	Quadratic
7	1.0	0.7	1.8
13	1.0	0.35	0.44
20	1.0	0.22	0.20
32	1.0	0.14	0.07
50	1.0	0.09	0.032
65	1.0	0.07	0.017
100	1.0	0.045	0.0075
160	1.0	0.027	0.0028
200	1.0	0.022	0.0019
325	1.0	0.014	0.0007
600	1.0	0.007	0.0002
3250	1.0	0.0014	0.000007

Tabel 1 – Valori comune pentru coeficienții de atenuare (sursa: [Ogre3D wiki](#))

Ar trebui să calculam atenuarea folosind ecuația de mai sus:

```
//compute distance to light
float dist = length(lightPosEye - fragPosEye.xyz);
//compute attenuation
float att = 1.0f / (constant + linear * dist + quadratic * (dist * dist));
```

În cele din urmă, ar trebui să aplicăm atenuarea componentelor ambientale, difuze și speculare:

```
//compute ambient light
ambient = att * ambientStrength * lightColor;

//compute diffuse light
diffuse = att * max(dot(normalEye, lightDirN), 0.0f) * lightColor;

specular = att * specularStrength * specCoeff * lightColor;
```

## 4 Hărți de iluminare

Până acum, când era vorba de iluminare, am considerat obiectele ca fiind compuse dintr-un singur material (având o singură culoare de bază) (Figura 2). Cu toate acestea, această abordare nu oferă prea multă flexibilitate pentru reprezentarea vizuală a unui obiect. Vom extinde acum modelul de iluminare pentru a include **hărți de iluminare**.

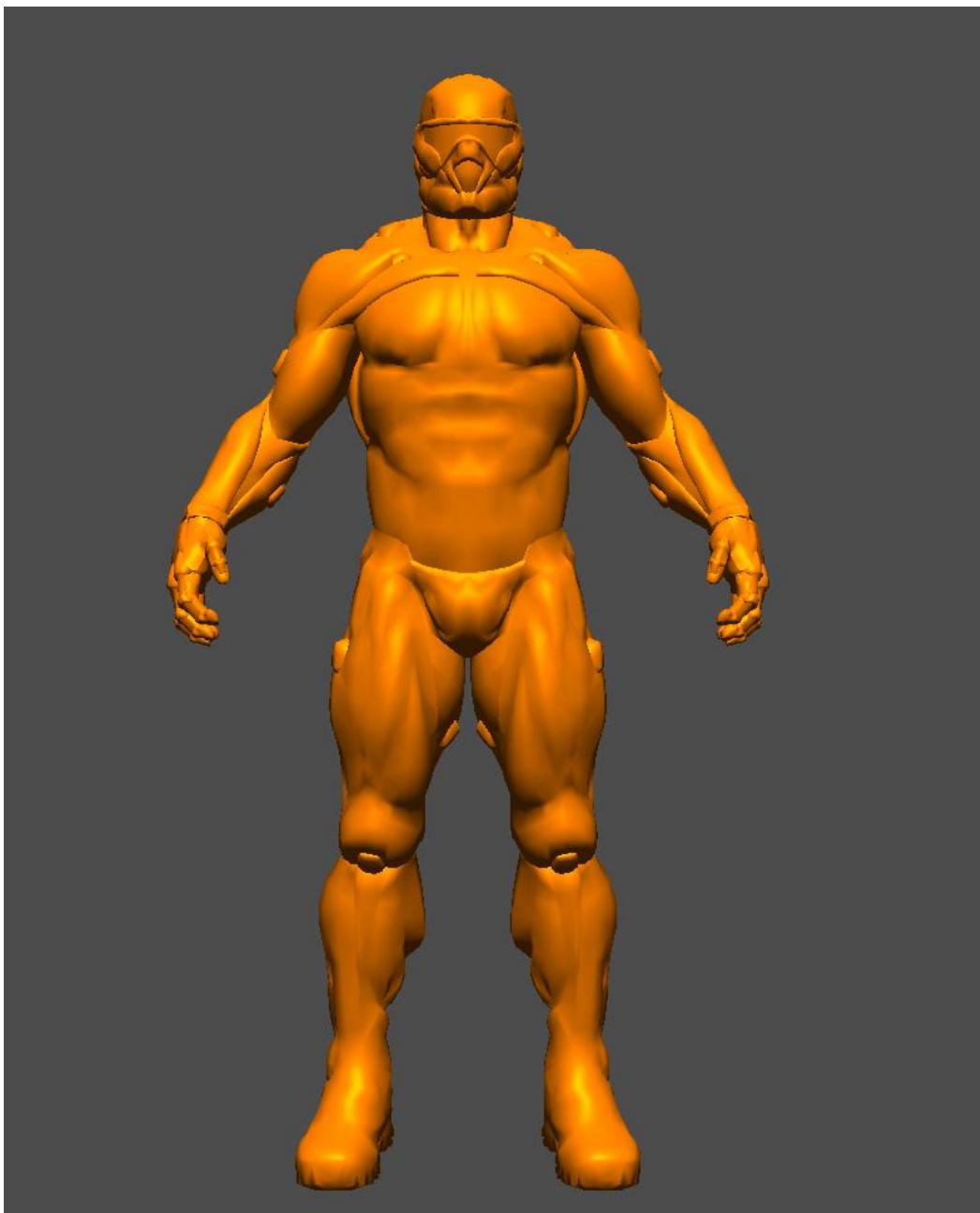


Figura 2 – Crysis 2 nanosuit rasterizat cu lumină punctiformă și culoare de bază portocalie

Hărțile de iluminare ne furnizează un mijloc de eşantionare a culorii de bază pentru fiecare fragment dintr-o textură și pentru fiecare componentă a luminii (ambientală, difuză și speculară). Ele sunt încorporate în proprietățile materialelor asociate modelelor OBJ (specificate în fișierul MTL).

**Hărțile difuze** definesc culorile unui obiect. Informațiile despre culoare se eşantionează dintr-o textură și se combină cu efectele iluminării pentru a determina culoarea finală a fragmentelor (Figura 3). Hărțile hibride folosesc de obicei și culoarea de bază pentru componenta ambientală. Astfel, calculul componentelor ambientale și difuze trebuie modificate:

```
ambient *= texture(diffuseTexture, fragTexCoords);  
diffuse *= texture(diffuseTexture, fragTexCoords);
```



Figure 3 – Crysis 2 nanosuit rasterizat cu hărți de iluminare difuze

De obicei, nu toate materialele care compun un obiect reflectă aceeași cantitate de strălucire speculară. Pentru a diferenția între reflexia diferitelor materiale, putem folosi **hărți speculare** care atribuie un coeficient specular mai mare materialelor cu grad ridicat de reflexie și unul inferior materialelor care nu prezintă în mod normal componentă speculară (Figura 4). Hărțile speculare pot fi eșantionate exact ca hărțile difuze, dar sunt aplicate la calculul componentelor speculare:

```
specular *= texture(specularTexture, fragTexCoords);
```

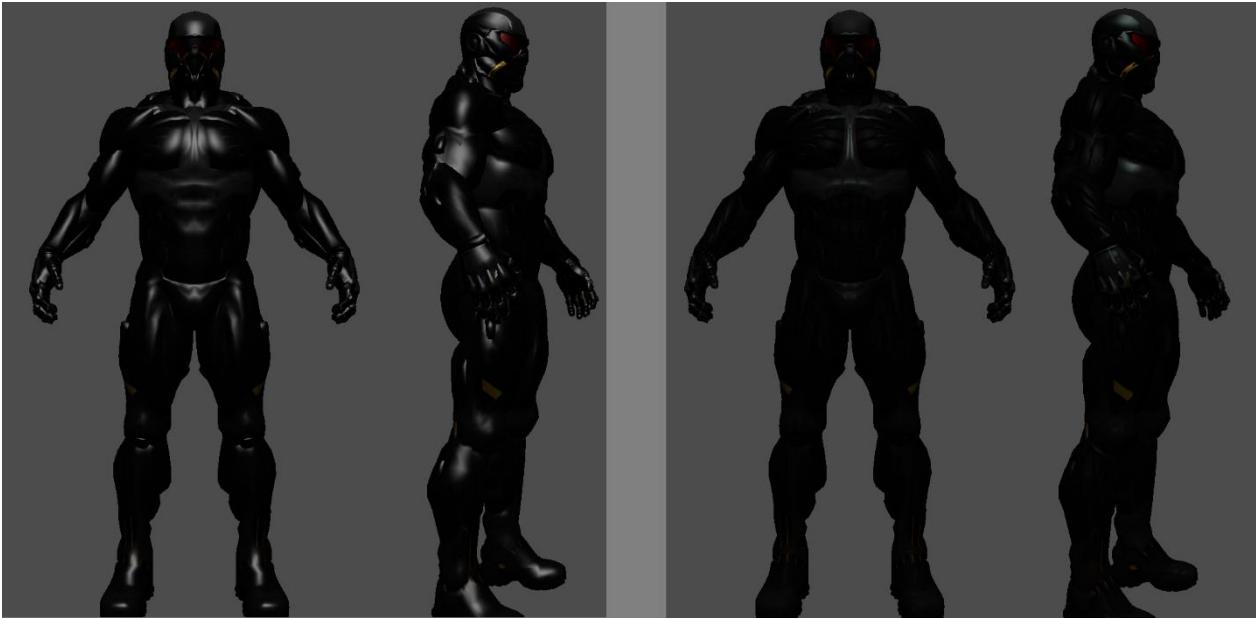


Figure 4 – Crysis 2 nanosuit rasterizat fără hărți speculare (stânga) și cu hărți speculare (dreapta)

Biblioteca de încărcare a obiectelor OBJ pe care o utilizati include deja funcționalitate pentru încărcarea hărților difuze și speculare (texturi) și legarea corespunzătoare a acestor obiecte de textură în cadrul claselor Model3D și Mesh. Variabilele **uniform** "diffuseTexture" și "specularTexture" trebuie să fie definite în programul shader pentru a permite eșantionarea hărților de iluminare ale obiectului.

## 5 Lectură suplimentară

- OpenGL Programming Guide 8<sup>th</sup> Edition – Capitolul 7
- Tutoriale OpenGL – Light casters, Lighting maps, Multiple lights, Advanced lighting - <http://www.learnopengl.com/>

## 6 Temă

1. Descărcați și explorați mostrele de cod furnizate ca resurse pentru acest laborator
2. Implementați modelul de iluminare Blinn-Phong pentru aplicația demo dată
3. Transformați lumina direcțională într-o lumină punctiformă pentru aplicația demonstrativă
4. Afisati mai multe obiecte împrăștiate prin toată scena pentru a observa efectul luminilor punctiforme
5. Incorporați hărți de iluminare în aplicația demo
6. Definiți o lumină direcțională și un al doilea punct (locație) de lumină în aplicația demonstrativă

# Laborator 9

---

## 1 Obiective

Obiectivul acestui laborator este să descrie pe scurt un mod simplu de adăugare a umbrelor în aplicația dvs. OpenGL 4 folosind tehnica **Shadow Mapping**.

## 2 Shadow Mapping

Tehnicile de iluminat discutate în laboratoarele anterioare funcționează excelent pentru iluminarea obiectelor. Totuși, efectul de iluminare obținut nu ia în considerare ocluziile. În lumea reală, atunci când un obiect nu este lovit în mod direct de razele emise de o sursă de lumină din cauza obstrucțiilor (alte obiecte), atunci obiectul este în umbră. Astfel, umbrele adaugă un grad de realism scenelor noastre OpenGL, permitând, de asemenea, o percepție mai profundă a adâncimii (Figura 1 și Figura 2).



Figura 1 – Rasterizare fără umbre

Există o mulțime de tehnici moderne care pot fi folosite pentru a produce umbre (calculul umbrelor) și toate acestea pot fi implementate în OpenGL. Cu toate acestea, nu există algoritmi perfecti în timp real pentru acest lucru, fiecare având propriile sale avantaje și dezavantaje. Pentru acest laborator, vom explora o tehnică ce oferă rezultate decente și este ușor de implementat, și anume shadow mapping (“hărți de umbre”).



Figura 2 – Rasterizare cu umbre

Shadow mapping este o tehnică multi-trecere care utilizează texturi de adâncime pentru a decide dacă un punct se află în umbră sau nu. Cheia este aceea de a observa scena din punctul de vedere al sursei de lumină în loc de locația finală de vizionare (locația camerei). Orice parte a scenei care nu este direct observabilă din perspectiva luminii va fi în umbră.

Etapele principale ale algoritmului sunt descrise mai jos::

1. Rasterizarea scenei din punctul de vedere al luminii. Nu contează cum arată scena (informații despre culoare); singurele informații relevante în acest moment sunt valorile de adâncime. Aceste valori sunt stocate într-o hartă de umbră (sau hartă de adâncime) și pot fi obținute prin crearea unei texturi de adâncime, atașarea la un obiect framebuffer și rasterizarea întregii scene (așa cum este văzută din poziția luminii) în acest obiect. În acest fel, textura de adâncime este umplută direct cu valorile relevante ale adâncimii.
2. Rasterizarea scenei din punct de vedere al observatorului (poziția camerei). Se compară adâncimea fiecărui fragment vizibil (proiectat în cadrul de referință al luminii) cu valorile de adâncime din harta umbrelor. Fragmentele care au o adâncime mai mare decât cea care a fost stocată anterior în harta de adâncime nu sunt direct vizibile din punctul de vedere al luminii și sunt, prin urmare, în umbră.

## 2.1 Generarea hărții de adâncime

Prima trecere a algoritmului de shadow mapping generează o hartă de adâncime a întregii scene privită din perspectiva luminii. Această hartă poate fi salvată direct într-o textură prin atașarea texturii la un obiect framebuffer și rasterizarea directă în acesta.

OpenGL face toată rasterizarea (informații despre culoare, adâncime și şablon) într-un framebuffer. Culoarele stocate în framebuffer sunt utilizate de ecran atunci când se afișează conținutul vizual generat de aplicațiile noastre. Obiectele Framebuffer ne permit să creăm propriile framebuffere și, în loc să rasterizăm în zone speciale de memorie GPU, putem scrie informații (culori, adâncime și informații şablon) în texture. Aceste texture sunt atașate ca obiecte buffer de culoare, adâncime și stencil pentru obiectele noastre framebuffer.

### 2.1.1 Crearea unui obiect framebuffer

Un obiect framebuffer este creat după cum se arată mai jos:

```
GLuint shadowMapFBO;  
//generate FBO ID  
 glGenFramebuffers(1, &shadowMapFBO);
```

Pentru a ataşa o textură ca un buffer de adâncime pentru obiectul framebuffer, trebuie mai întâi să creăm textura:

```
GLuint depthMapTexture;  
//create depth texture for FBO  
 glGenTextures(1, &depthMapTexture);  
 glBindTexture(GL_TEXTURE_2D, depthMapTexture);  
 glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,  
 SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);  
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
 float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };  
 glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);  
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);  
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
```

Textura de adâncime este creată folosind tipul de textură GL\_DEPTH\_COMPONENT. De asemenea, pentru a evita artefactele vizuale nedorite atunci când generați umbre, modul de infășurare a texturii este setat la GL\_CLAMP\_TO\_EDGE. SHADOW\_WIDTH și SHADOW\_HEIGHT sunt constante și reprezintă rezoluția hărții de adâncime, care ar trebui să fie cel puțin de dimensiunea framebuffer-ului ferestrei OpenGL. Rezoluțiile prea scăzute au tendința de a conduce la rezultate mai slabe, iar rezoluțiile prea mari au tendința de a irosi memoria.

Odată ce textura de adâncime este creată, ea trebuie atașată ca buffer de adâncime pentru obiectul framebuffer:

```
//attach texture to FBO  
 glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);  
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMapTexture,  
 0);
```

Deoarece prima trecere a algoritmului de mapare a umbrelor nu necesită atașament de coloare sau şablon, dar un obiect framebuffer nu ar fi complet fără ele, putem să atribuim în mod explicit **nimic** acestor puncte de atașament:

```
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
```

Odată ce un obiect framebuffer este complet, ar trebui să îl dezactivăm până când suntem gata să îl folosim:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Cu un obiect framebuffer configurat corect, etapa completă de redare, folosind harta de adâncime, este după cum urmează:

```
// rasterizați în harta de adâncime
Schimbați portul de vizualizare pentru a acoperi dimensiunea texturării de adâncime;
Activăți obiectul framebuffer;
    Curățați (inițializați) buffer-ul de adâncime;
    Configurați shader-ul (pentru completarea texturii de adâncime) și matricile de transformare (pentru sistemul de coordinate al luminii);
    Rasterizați scenă;
Dezactivați obiectul framebuffer;
// rasterizați în framebuffer
Modificați portul de vizualizare pentru a acomoda obiectul framebuffer al ferestrei OpenGL;
    Curățați (inițializați) buffer-ul de adâncime și culoarea;
    Configurați shader-ul (pentru rasterizarea cu umbre) și matricile de transformare;
    Activăți (legeați) textura de adâncime;
    Rasterizați scenă;
```

## 2.1.2 Transformări în spațiul luminii

Prima trecere ar trebui să rasterizeze scenă din punctul de vedere al luminii. Aceasta înseamnă că toate coordonatele tuturor obiectelor din scenă trebuie transformate astfel încât să fie relative la lumină și nu la cameră. Astfel, putem folosi aceeași funcție ca și în cazul transformarea spațiului camerei, **lookAt**, dar cu poziția luminii ca prim parametru. Pentru acest laborator, vom folosi o lumină direcțională. Deoarece luminile direcționale nu au poziție (sunt amplasate la infinit), putem folosi ca poziție orice punct de-a lungul direcției luminii, inclusiv direcția însăși (interpretată ca un punct):

```
glm::mat4 lightView = glm::lookAt(lightDir, glm::vec3(0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

Deoarece toate razele provenite de la o lumină direcțională sunt paralele, vom folosi o proiecție ortografică pentru a evita orice deformare perspectivă:

```
const GLfloat near_plane = 0.1f, far_plane = 5.0f;
glm::mat4 lightProjection = glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, near_plane, far_plane);
```

Matricea finală de transformare pentru spațiul luminii este:

```
glm::mat4 lightSpaceTrMatrix = lightProjection * lightView;
```

## 2.1.3 Rasterizarea în harta de adâncime

Vertex Shader-ul care transformă toate vîrfurile în spațiul luminii este prezentat mai jos:

```
#version 410 core
```

```

layout(location=0) in vec3 vPosition;

uniform mat4 lightSpaceTrMatrix;
uniform mat4 model;

void main()
{
    gl_Position = lightSpaceTrMatrix * model * vec4(vPosition, 1.0f);
}

```

În OpenGL 4, este permis să nu aveți un shader de fragment numai atunci când rasterizarea este oprită. Astfel, un simplu shader de fragment cu ieșire “dummy”(falsă) trebuie implementat<sup>1</sup>:

```

#version 410 core

out vec4 fColor;

void main()
{
    fColor = vec4(1.0f);
}

```

Rasterizarea în harta de adâncime devine:

```

//render the scene to the depth buffer

depthMapShader.useShaderProgram();

glUniformMatrix4fv(glGetUniformLocation(depthMapShader.shaderProgram, "lightSpaceTrMatrix"),
    1,
    GL_FALSE,
    glm::value_ptr(computeLightSpaceTrMatrix()));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
 glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
 glClear(GL_DEPTH_BUFFER_BIT);

RenderTheScene();

glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

Dacă ar fi să vizualizăm harta de adâncime în acest moment, ar arăta în figura 3.

## 2.2 Rasterizarea cu umbre

Odată ce hartă adâncimii a fost creată, ea poate fi utilizată pentru a rasteriza scena cu umbre. Când se compară adâncimea fragmentului curent cu adâncimea existentă în harta de adâncime, ambele trebuie să

---

<sup>1</sup> Am putea avea, de asemenea, un shader de fragment gol, deoarece ne pasă doar de valorile de adâncime, care sunt generate în spatele scenei de OpenGL oricum.

se afle în sistemul de referință al luminii. Deoarece primul nu este, trebuie transformat folosind aceeași matrice ca atunci când se generează harta. Transformările ar trebui efectuate în vertex shader:

```
...
out vec4 fragPosLightSpace;
...
uniform mat4 lightSpaceTrMatrix;
...
fragPosLightSpace = lightSpaceTrMatrix * model * vec4(vPosition, 1.0f);
...
```

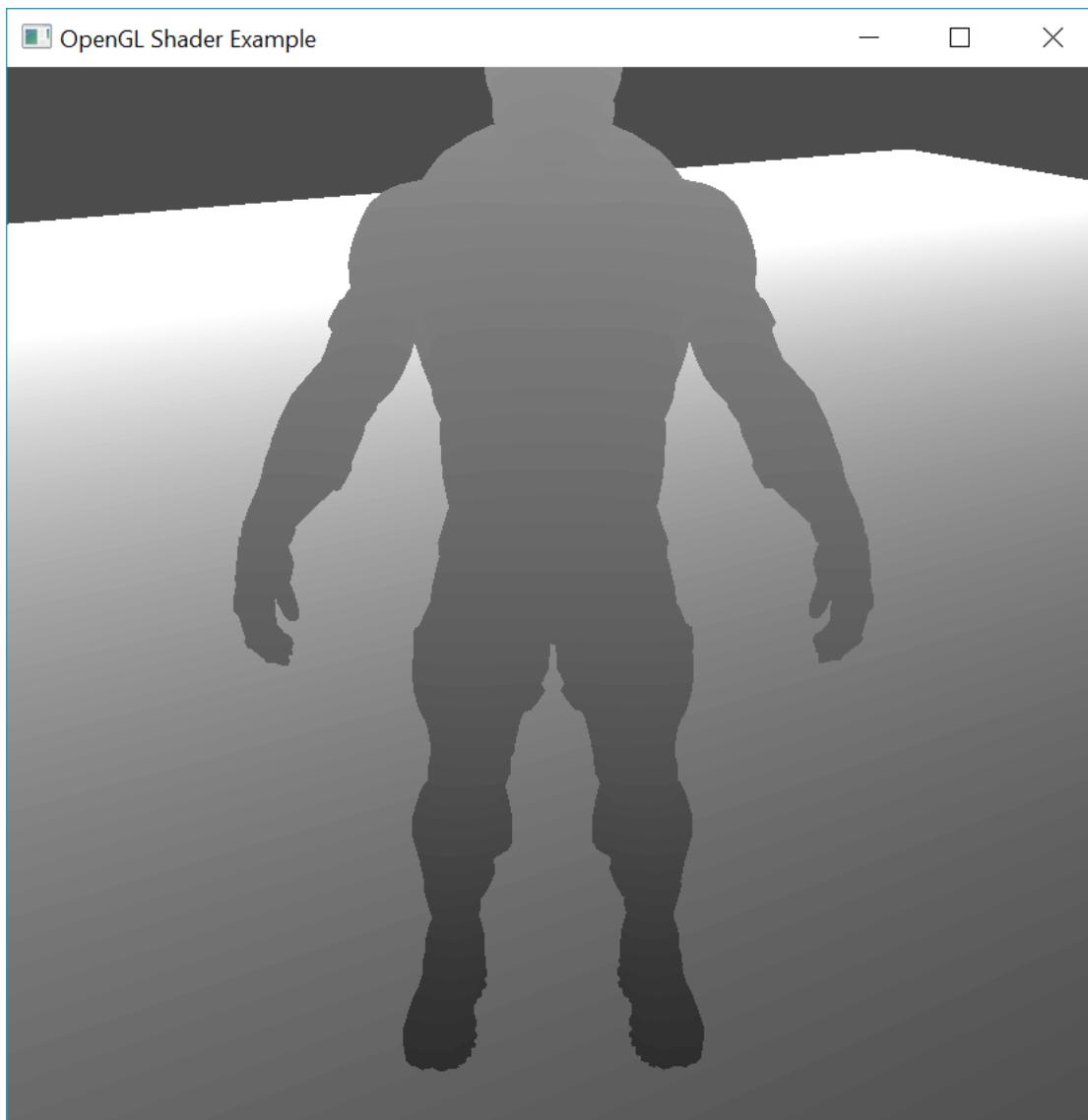


Figura 3 – Exemplu de hartă de adâncime

Calculul real al umbrelor va avea loc în shader-ul fragmentului. Culoarea finală a fiecărui fragment (pentru componente difuze și speculare) va fi modulată cu valoarea de umbră calculată (1.0 sau 0.0)

```

...
float computeShadow()
{
...
}

//modulate with shadow
shadow = computeShadow();
vec3 color = min((ambient + (1.0f - shadow)*diffuse) + (1.0f - shadow)*specular, 1.0f);

```

Pentru a efectua calculul efectiv al umbrei, primul lucru pe care ar trebui să-l facem este să transformăm poziția fragmentului în coordonatele normalizate ale spațiului luminii. Deoarece transformarea în spațiu luminii a fost făcută în vertex shader, rămâne să aplicăm diviziunea perspectivă pentru a obține coordonatele normalizate:

```

in vec4 fragPosLightSpace;

...
float computeShadow()
{
...

// perform perspective divide
vec3 normalizedCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
...
}
...

```

Aceasta returnează poziția fragmentului curent în intervalul [-1,1]. Deoarece valorile adâncimii fragmentului sunt în intervalul [0,1], ar trebui să transformăm coordonatele normalizate în consecință:

```

...
float computeShadow()
{
...

// Transform to [0,1] range
normalizedCoords = normalizedCoords * 0.5 + 0.5;
...
}
...

```

Acum putem eșantiona adâncimea existentă în harta de adâncime folosind aceste coordonate, deoarece acestea corespund acum cu coordonatele NDC (Normalized Device Coordinates) din prima trecere:

```

...
uniform sampler2D shadowMap;
...
float computeShadow()
{
...

// Get closest depth value from light's perspective
float closestDepth = texture(shadowMap, normalizedCoords.xy).r;
...

```

```
}
```

```
...
```

Acest lucru ne oferă cea mai apropiată valoare de adâncime din perspectiva luminii. Apoi luăm adâncimea (în sistemul de referință al luminii) fragmentului curent:

```
...
float computeShadow()
{
...
// Get depth of current fragment from light's perspective
float currentDepth = normalizedCoords.z;
...
}
```

Acum putem compara cele două valori. Dacă adâncimea fragmentului curent este mai mare decât valoarea din harta adâncimii, fragmentul curent este în umbră. Altfel, este iluminat:

```
...
float computeShadow()
{
...
// Check whether current frag pos is in shadow
float shadow = currentDepth > closestDepth ? 1.0 : 0.0;
...
}
```

Atunci când utilizați aceste shadere, cu textura de adâncime corect legată, ieșirea trebuie să fie similară celei din figura 4.



Figura 4 – Rasterizare cu umbre

## 2.3 Îmbunătățirea calității

### 2.3.1 Shadow acne

Privind mai atent, putem observa un model nedorit, aşa cum este ilustrat în Figura 5.

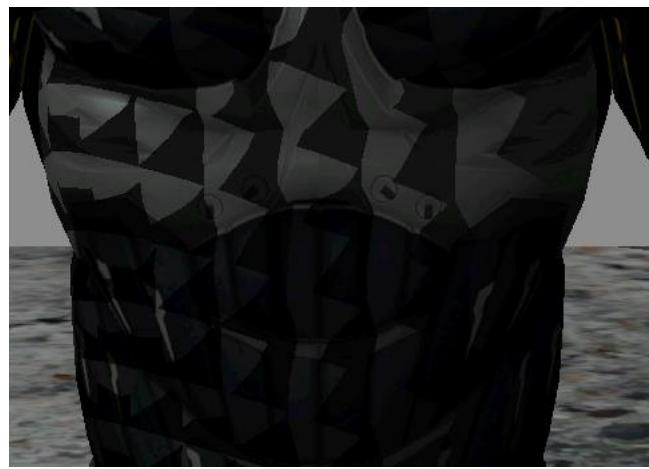


Figura 5 – Shadow acne

Acet lucru se întâmplă din cauza rezoluției limitate a hărții de adâncime și apare atunci când mai multe fragmente eșantionează aceeași valoare din harta de adâncime. De obicei, putem rezolva această problem folosind o mică modificare la adâncimea fragmentului, fie o valoare fixă, fie o valoare bazată pe unghiul

pe care suprafața îl face cu direcția luminii (aceasta rezolvă problemele pentru suprafete care au un unghi abrupt față de lumină):

```
...
float computeShadow()
{
...
// Check whether current frag pos is in shadow
float bias = 0.005f;
float shadow = currentDepth - bias > closestDepth ? 1.0f : 0.0f;
...
}
```

sau

```
...
float computeShadow()
{
...
// Check whether current frag pos is in shadow
float bias = max(0.05f * (1.0f - dot(normal, lightDir)), 0.005f);
float shadow = currentDepth - bias > closestDepth ? 1.0f : 0.0f;
...
}
```

Ieșirea vizuală ar trebui să devină acum similară cu cea din Figura 6.

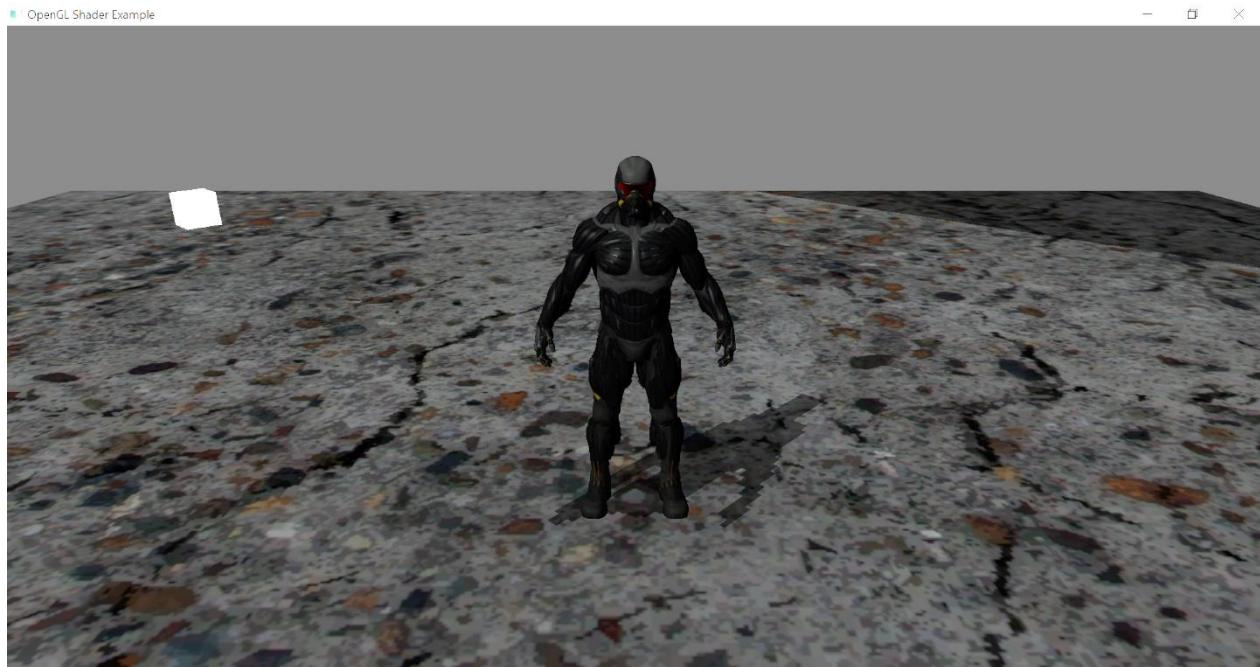


Figura 6 – Fără shadow acne

### 2.3.2 Supra-eşantionare

Observați că există zone în umbră care ar trebui să fie iluminate (colțul din dreapta din Figura 6). Acest lucru se întâmplă deoarece fragmentele în afara frustumului hărții de adâncime au o valoare a adâncimii mai mare de 1,0. Acest lucru poate fi ușor rectificat în fragment shader:

```
...
float computeShadow()
{
...
if (normalizedCoords.z > 1.0f)
    return 0.0f;
...
}
```

Ieșirea vizuală ar trebui să devină acum similară cu cea din Figura 7.



Figure 7 – Fără supra-eşantionare

## 3 Lectură suplimentară

- OpenGL Programming Guide 8<sup>th</sup> Edition – Capitolul 7 (Shadow Mapping)
- OpenGL tutorials – Advanced OpenGL (Depth testing, Face culling, Framebuffers), Advanced lighting (Shadows) – <http://www.learnopengl.com/>

## 4 Temă

1. Descărcați resursele de pe site

2. Adăugați obiectul sol în scenă
3. Adăugați un obiect framebuffer la aplicația existentă
4. Adăugați o textură de adâncime la obiectul framebuffer
5. Adăugați codul și shaderele necesare pentru a rasteriza harta de adâncime a luminii în textura de adâncime
6. Adăugați codul necesar pentru a rasteriza scenă cu umbre
7. Efectuați ajustările necesare pentru a rezolva diferitele probleme vizuale întâlnite (shadow acne umbră, supra-eșantionare)

# Laborator 4

---

## 1 Obiective

Obiectivul acestui laborator este de a prezenta diferite metode de depanare a aplicațiilor OpenGL.

## 2 Notiuni teoretice

### 2.1 glGetError()

OpenGL va genera mesaje de eroare de fiecare dată când există o utilizare incorectă a funcțiilor sau a funcționalității. Puteți căuta mesaje de eroare folosind funcția `glGetError()`. Fiecare eroare are asociate un cod numeric și un nume simbolic. În caz de eroare, este setat un semnal (flag cu valoarea corespunzătoare a codului de eroare). Nu sunt generate alte erori până când se apelează `glGetError`. La fiecare apel, codul de eroare este returnat și semnalul (flag-ul) este resetat. Se pot genera următoarele coduri de eroare:

Error code	Error flag	Error description
<b>0</b>	GL_NO_ERROR	Nu a fost înregistrată nicio eroare. Valoarea acestei constante simbolice este garantată a fi 0.
<b>1280</b>	GL_INVALID_ENUM	Se specifică o valoare inacceptabilă pentru un argument enumerat. Comanda necorespunzătoare este ignorată și nu are alt efect secundar decât să seteze semnalul de eroare.
<b>1281</b>	GL_INVALID_VALUE	Un argument numeric este în afara intervalului acceptat. Comanda necorespunzătoare este ignorată și nu are alt efect secundar decât să seteze semnalul de eroare.
<b>1282</b>	GL_INVALID_OPERATION	Operația specificată nu este permisă în starea actuală. Comanda necorespunzătoare este ignorată și nu are alt efect secundar decât să seteze semnalul de eroare.
<b>1286</b>	GL_INVALID_FRAMEBUFFER_OPERATION	Obiectul framebuffer nu este complet. Comanda necorespunzătoare este ignorată și nu are alt efect secundar decât să seteze semnalul de eroare.
<b>1285</b>	GL_OUT_OF_MEMORY	Nu este suficientă memorie pentru executarea comenzi. Starea GL este nedefinită, cu excepția stării semnalelor de eroare, după ce această eroare este înregistrată.
<b>1284</b>	GL_STACK_UNDERFLOW	A fost făcută o încercare de a efectua o operațiune care ar determina o stivă internă să se prăbușească.

1283	GL_STACK_OVERFLOW	A fost făcută o încercare de a efectua o operație care ar determina o stivă internă să depășească limita admisa.
------	-------------------	--

Pentru a verifica erorile, utilizați următoarea funcție:

```
GLenum glCheckError_(const char *file, int line)
{
    GLenum errorCode;
    while ((errorCode = glGetError()) != GL_NO_ERROR)
    {
        std::string error;
        switch (errorCode)
        {
            case GL_INVALID_ENUM:           error = "INVALID_ENUM"; break;
            case GL_INVALID_VALUE:          error = "INVALID_VALUE"; break;
            case GL_INVALID_OPERATION:      error = "INVALID_OPERATION"; break;
            case GL_STACK_OVERFLOW:         error = "STACK_OVERFLOW"; break;
            case GL_STACK_UNDERFLOW:        error = "STACK_UNDERFLOW"; break;
            case GL_OUT_OF_MEMORY:          error = "OUT_OF_MEMORY"; break;
            case GL_INVALID_FRAMEBUFFER_OPERATION: error = "INVALID_FRAMEBUFFER_OPERATION"; break;
        }
        std::cout << error << " | " << file << "(" << line << ")" << std::endl;
    }
    return errorCode;
}

#define glCheckError() glCheckError_(__FILE__, __LINE__)
```

## 2.2 Depanare la nivel de shader

Din păcate la nivelul shaderului nu putem folosi funcții precum glGetError () și nu putem face o depanare pas cu pas prin programul nostru de shader. O soluție pentru depășirea acestei restricții este afișarea unor variabile pe canalul de ieșire al fragment shader-ului. De exemplu, putem afișa vectorii normale pentru a verifica dacă au valori corecte.

Actualizați datele varfurilor pentru a încorpora vectorii normala:

```

GLfloat vertexData[] = {
    //vertex position, colour and normal vector
    0.0f, 0.7f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
    -0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
    0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f,
    0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f
};

GLuint vertexIndices[] = {
    0, 1, 4,
    1, 2, 3,
    1, 3, 4
};

GLuint verticesVBO;
GLuint verticesEBO;
GLuint objectVAO;

```

Schimbați funcția initObjects pentru a trimite atributul normală către vertex shader.

Actualizați vertex shader:

```

#version 400

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColour;
layout(location = 2) in vec3 vertexNormal;

out vec3 colour;
out vec3 normal;

void main() {
    colour = vertexColour;

```

```
normal = vertexNormal;  
gl_Position = vec4(vertexPosition, 1.0);  
}
```

Actualizați fragment shader:

```
#version 400  
  
in vec3 colour;  
in vec3 normal;  
  
out vec4 fragmentColour;  
  
void main() {  
    fragmentColour.rgb = normal;  
    fragmentColour.a = 1.0;  
}
```

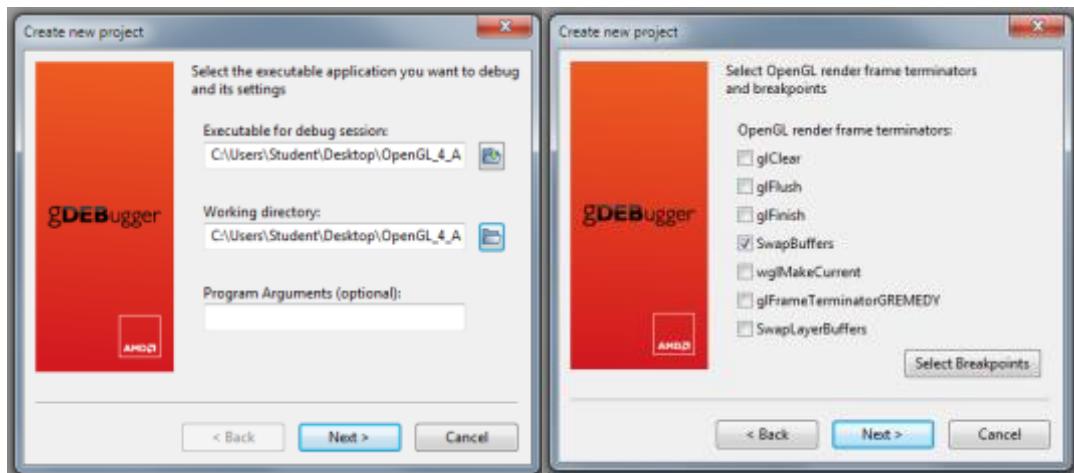
Întrebare: De ce obiectul este acum afișat în albastru?

### 2.3 Aplicație terță parte

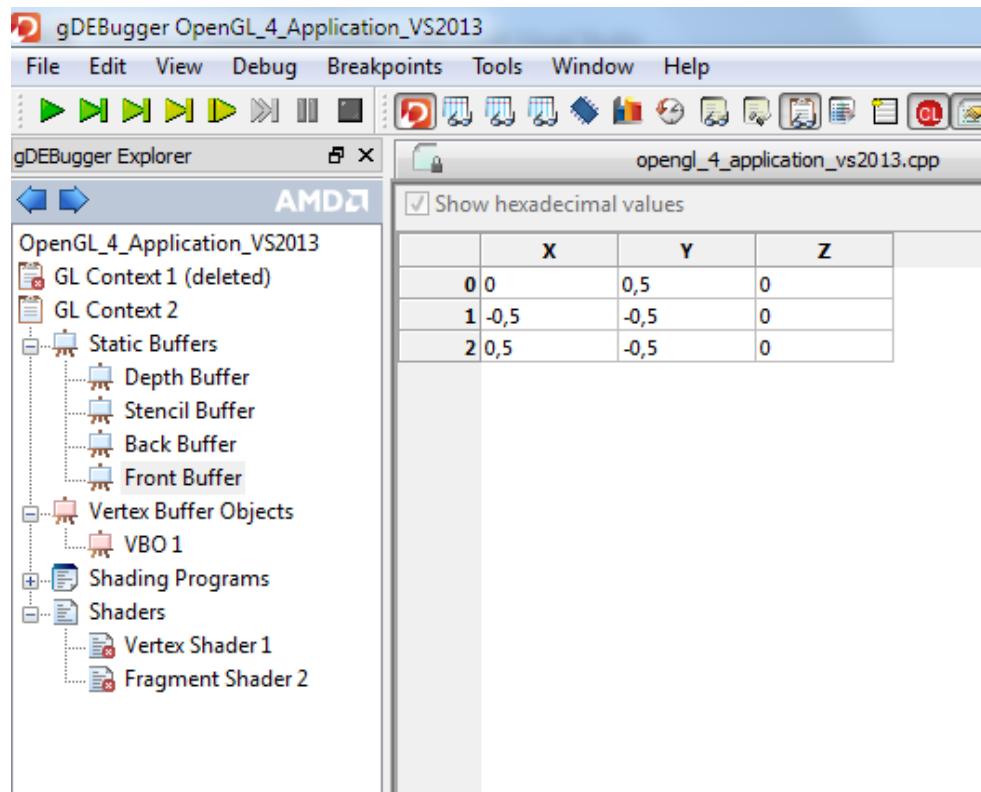
Aplicațiile de la terțe părți pot intercepta apeluri OpenGL diferite și pot fi folosite pentru a urmări utilizarea funcțiilor OpenGL, pentru a găsi blocaje, a inspecta memoria buffer și pentru a afișa texturile și atașamentele framebuffer. gDebugger este unul dintre instrumentele de depanare disponibile pentru aplicațiile OpenGL. Este o aplicație cross-platform care oferă o prezentare detaliată a stării aplicatiei OpenGL care rulează. Descărcați aplicația gDebugger de la:

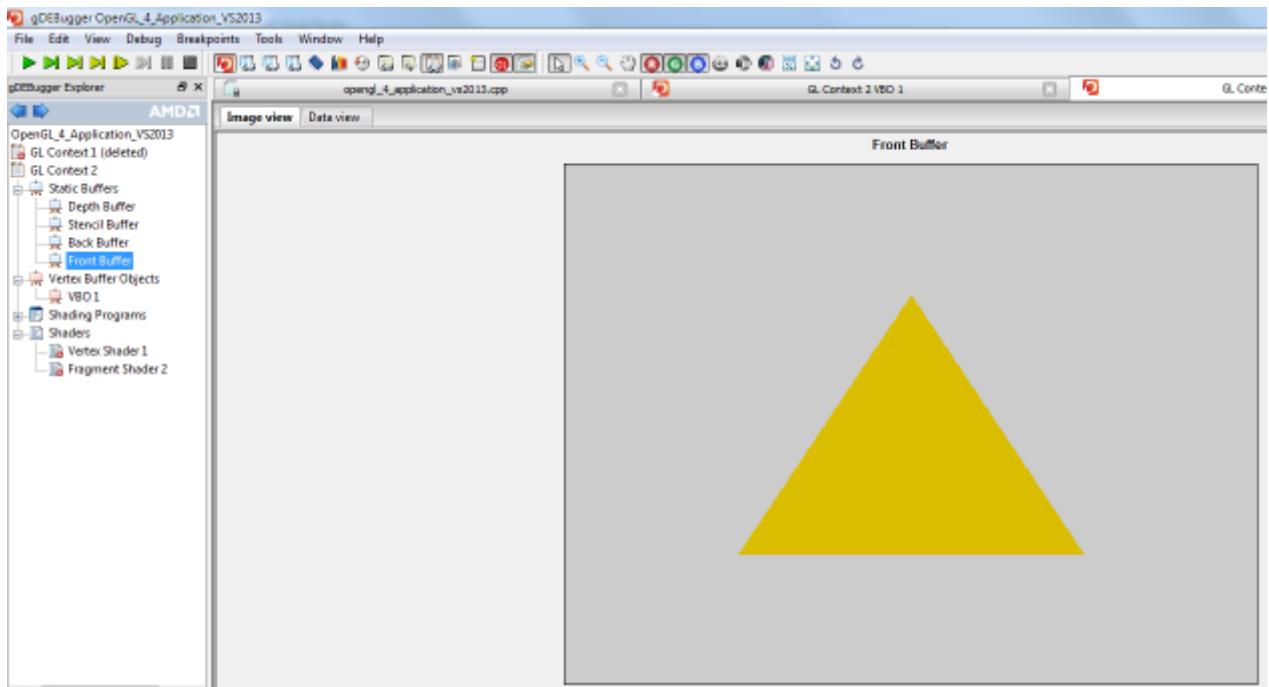
<http://developer.amd.com/tools-and-sdks/archive/compute/amd-gdebugger/>

Pentru a depana o aplicație OpenGL folosind gDebugger, trebuie să creați un nou proiect și să selectați locația programului și directorul de lucru al executabilului OpenGL.



Dacă întrerupeți aplicarea în orice moment, puteți examina starea curentă, memoria texturii și / sau utilizarea bufferului.





## 2.4 Sfaturi și trucuri

Ce trebuie să faceți dacă aplicația dvs. OpenGL afișează doar un ecran negru:

- Verificați erorile folosind glGetError ()
- Verificați dacă geometria se află în interiorul frustumului de vizualizare
- Schimbați culoarea de fundal diferită de negru
- Verificați dacă sunt configurate atributele vertexului
- Verificați dacă variabilele uniforme sunt setate

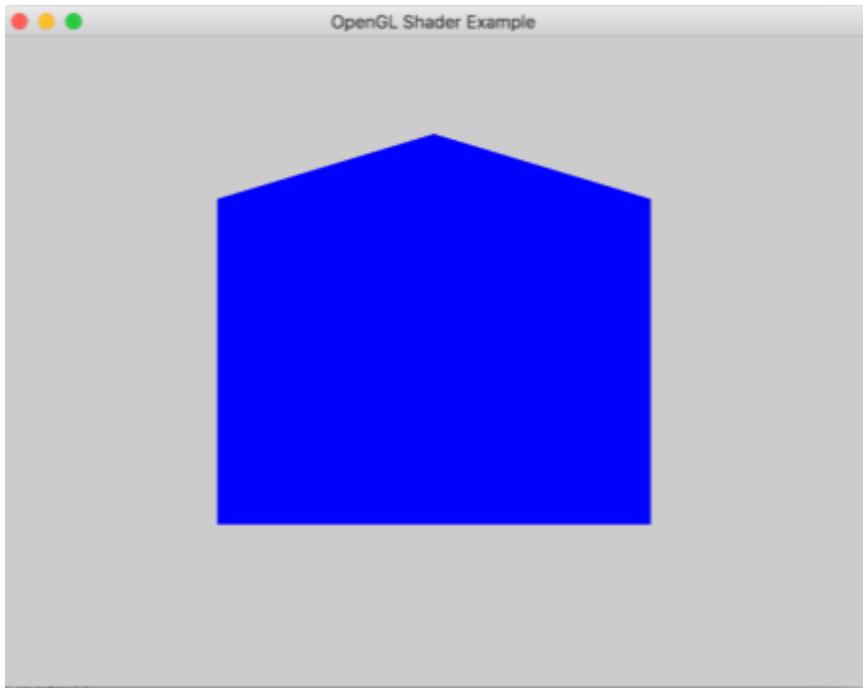
## 3 Further reading

Funcție (și legătură)	Descriere
<a href="#">glGenBuffers</a>	generează nume de obiecte tampon buffer
<a href="#">glBindBuffer</a>	leagă un obiect buffer la punctul de legare al buffer-ului specificat
<a href="#">glBufferData</a>	creează și inițiază stocarea datelor unui obiect buffer
<a href="#">glGenVertexArrays</a>	generează nume de vertex array object
<a href="#">glBindVertexArray</a>	activează un vertex array object
<a href="#">glVertexAttribPointer</a>	definește un sir de date generice asociate unui vârf
<a href="#">glEnableVertexAttribArray</a>	activează un sir generic de atribut
<a href="#">glCreateShader</a>	creează un obiect shader
<a href="#">glShaderSource</a>	înlocuiește codul sursă într-un obiect shader
<a href="#">glCompileShader</a>	compilează un obiect shader
<a href="#">glCreateProgram</a>	creează un obiect program
<a href="#">glAttachShader</a>	atașează un obiect shader unui obiect program
<a href="#">glLinkProgram</a>	operație de linking pentru un obiect program
<a href="#">glDeleteShader</a>	șterge obiect shader

<a href="#">glUseProgram</a>	instalează un obiect de program ca parte a stării curente de rasterizare
<a href="#">glDrawArrays</a>	rasterizează primitive folosind şiruri de date
<a href="#">glGetUniformLocation</a>	returnează locația unei variabile uniform
<a href="#">glUniform</a>	specifică valoarea unei variabile uniform pentru obiectul program curent
<a href="#">glDrawElements</a>	rasterizează primitive folosind şiruri de date
<a href="#">glGetError</a>	Returnează valoarea semnalului de eroare

## 4 Temă

- Descărcați fișierele de pe site și actualizați proiectul anterior cu acestea. Programul ar trebui să se compileze, dar va afișa o fereastră gri. Încercați să depanați programul pentru a afișa următoarele:



# Laborator 11

---

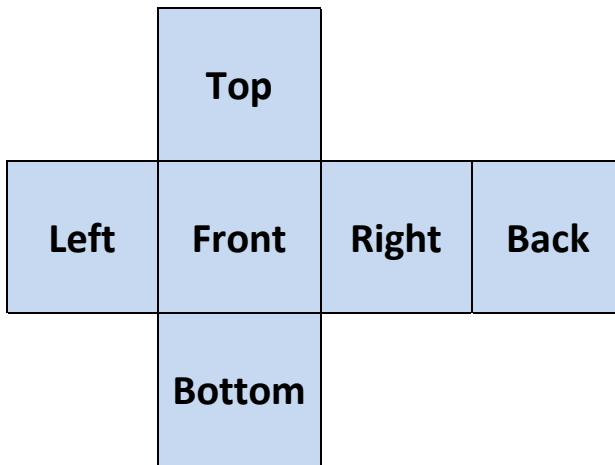
## 1 Obiective

Acest laborator vă prezintă noțiunile de bază despre conceptul de cubemaps (care sunt folosite pentru definirea skybox-urilor) și proprietățile de reflecție ale obiectelor 3D.

## 2 Fundament teoretic

### 2.1 Cubemap

În multe situații, dorim să afișăm o imagine "de fundal" reprezentând partea din scena 3D pe care nu am modelat-o. Putem realiza acest lucru prin includerea scenei noastre 3D în interiorul unui cub cu texturi diferite aplicate pe fiecare față. Acest cub se numește skybox și este compus din 6 texturi (puteți descărca mostre de [aici](#)) urmând acest model:



Reprezentarea OpenGL a acestui tip de textură este un **cubemap**, care conține 6 texturi individuale 2D. Prin combinarea acestor texturi și maparea acestora pe un cub unitar, putem foarte ușor să eșantionăm o valoare a texturii utilizând un vector de direcție. Acest vector poate fi preluat din poziția interpolată a vârfului cubului.

Pentru a crea un cubemap, trebuie să generăm un id de textură și să îl legăm la ținta `GL_TEXTURE_CUBE_MAP`.

```
GLuint textureID;  
 glGenTextures(1, &textureID);  
 glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

Pentru fiecare față a cubemap-ului, trebuie să apelam funcția `glTexImage2D` și să setăm un parametru țintă specific pentru față curentă:

Target	Orientation
<code>GL_TEXTURE_CUBE_MAP_POSITIVE_X</code>	Right
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_X</code>	Left
<code>GL_TEXTURE_CUBE_MAP_POSITIVE_Y</code>	Top
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_Y</code>	Bottom
<code>GL_TEXTURE_CUBE_MAP_POSITIVE_Z</code>	Back
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_Z</code>	Front

Putem incrementa liniar ținta pentru a seta toate texturile cubului:

```
for(GLuint i = 0; i < skyBoxFaces.size(); i++)
{
    image = stbi_load(skyBoxFaces[i], &width, &height, &n, force_channels);
    if (!image) {
        fprintf(stderr, "ERROR: could not load %s\n", skyBoxFaces[i]);
        return false;
    }
    glTexImage2D(
        GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
        GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
}
```

În **fragment shader**, folosim un “sampler” diferit în acest caz (**samplerCube**), dar folosim aceeași funcție (“texture”) pentru a eșantiona valoarea de culoare. Aici, variabila `textureCoordinates` reprezintă vectorul de direcție.

```
#version 410 core

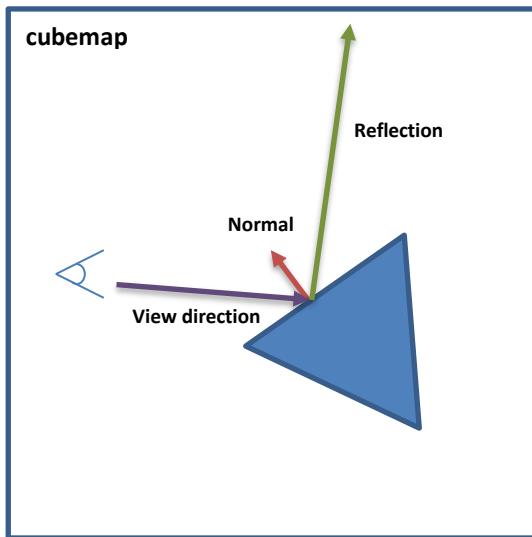
in vec3 textureCoordinates;
out vec4 color;

uniform samplerCube skybox;

void main()
{
    color = texture(skybox, textureCoordinates);
}
```

## 2.2 Reflecții

Având acum un mediu înconjurător (reprezentat de cubemap), putem specifica proprietățile reflexive ale diferitelor obiecte. Reflecția se bazează pe unghiul privitorului. Acest vector de reflecție poate fi calculat utilizând funcția GLSL numită “reflect”. Trebuie să specificăm vectorul normală și vectorul de direcție a camerei. Vectorul rezultat este folosit pentru a eșantiona din textura cubemap.



```
vec3 viewDirectionN = normalize(viewDirection);
vec3 normalN = normalize(normal);
vec3 reflection = reflect(viewDirectionN, normalN);
vec3 colorFromSkybox = vec3(texture(skybox, reflection));
```

### 3 Tutorial

Începeți prin a descărca resursele și a le include în proiectul dvs. Clasa SkyBox va fi utilizată pentru încărcarea și desenarea cubemap-urilor.

Adăugați într-un vector texturile (mai exact, calea lor) care vor fi mapate pe cubemap

```
std::vector<const GLchar*> faces;
faces.push_back("textures/skybox/right.tga");
faces.push_back("textures/skybox/left.tga");
faces.push_back("textures/skybox/top.tga");
faces.push_back("textures/skybox/bottom.tga");
faces.push_back("textures/skybox/back.tga");
faces.push_back("textures/skybox/front.tga");
```

Definiți următoarele variabile care vor reprezenta skybox-ul nostru și shader-ul folosit pentru desenarea acestuia.

```
gps::SkyBox mySkyBox;
gps::Shader skyboxShader;
```

Încărcați shader-ul skybox-ului și inițializați matricea de vizualizare (camera) și de proiecție pentru acesta.

```
mySkyBox.Load(faces);

skyboxShader.loadShader("shaders/skyboxShader.vert", "shaders/skyboxShader.frag");
```

```
skyboxShader.useShaderProgram();

view = myCamera.getViewMatrix();
glUniformMatrix4fv(glGetUniformLocation(skyboxShader.shaderProgram, "view"), 1, GL_FALSE,
glm::value_ptr(view));

projection = glm::perspective(glm::radians(45.0f), (float)retina_width / (float)retina_height, 0.1f, 1000.0f);
glUniformMatrix4fv(glGetUniformLocation(skyboxShader.shaderProgram, "projection"), 1, GL_FALSE,
glm::value_ptr(projection));
```

În funcția renderScene() apelați funcția de desenare a skybox-ului după ce ați apelat toate celelalte funcții pentru desenarea restului obiectelor.

```
mySkyBox.Draw(skyboxShader, view, projection);
```

## 4 Temă

- Urmați tutorialul pentru a afișa un skybox.
- Încărcați un alt obiect 3D în scenă și calculați culoarea obiectului, luând în considerare și reflecțiile provenite de la skybox.

# Laborator 12

---

## 1 Obiective

Acest laborator vă prezintă noțiuni de bază despre concepțele de efect de ceață, combinarea culorilor și eliminarea fragmentelor.

## 2 Ceață

Putem să creștem realismul unei scene 3D prin adăugarea unui efect de ceață care va crea o atmosferă specială. Prin manipularea atributelor asociate efectului de ceață putem să personalizăm atmosfera și, de asemenea, să ameliorăm percepția de adâncime (axa Z).

Culoarea de fundal trebuie aleasă în funcție de culoarea efectului de ceață.

```
glClearColor(0.5, 0.5, 0.5, 1.0);
```

În versiunile de OpenGL cu pipeline fix existau trei metode de calculare a ceții (liniară, exponențială și exponențială pătratică). Aceeași funcționalitate o putem realiza și în versiunea cu pipeline programabil prin adăugarea calculelor de ceață în shader-ul nostru de fragmente.

### 2.1 Ceață lineară

Prima metodă de calculare a ceții este prin folosirea unei funcții de interpolare liniară care va amesteca culoarea ceții cu aceea a fragmentului, pe baza distanței fragmentului (calculată în spațiul de vizualizare).

$$fogFactor = \frac{fogEnd - fragmentDistance}{fogEnd - fogStart}$$

Valorile factorului de ceață trebuie să fie între 0.0 and 1.0.

### 2.2 Ceață exponențială

Un rezultat mai bun poate fi obținut luând în considerare reducerea intensității luminii în funcție de distanță. Factorul de atenuare care va fi folosit reprezintă densitatea de ceață, această densitate fiind constantă în toată scena. Rezultatul este o scădere rapidă a factorului de ceață în comparație cu abordarea liniară.

$$fogFactor = e^{-fragmentDistance * fogDensity}$$

### 2.3 Ceață exponențială pătratică

Formula este practic aceeași cu cea precedentă, singura diferență provenind din faptul că am ridicat la patrat distanța fragmentului și densitatea de ceață.

$$fogFactor = e^{-(fragmentDistance * fogDensity)^2}$$

## 2.4 Tutorial

Adăugați următoarea funcție în interiorul fragment shader-ului:

```
float computeFog()
{
    float fogDensity = 0.05f;
    float fragmentDistance = length(fragmentPosEyeSpace);
    float fogFactor = exp(-pow(fragmentDistance * fogDensity, 2));

    return clamp(fogFactor, 0.0f, 1.0f);
}
```

Funcția **clamp** constrânge o valoare să se afle între două limite (minValue și maxValue).

```
float fogFactor = computeFog();
vec4 fogColor = vec4(0.5f, 0.5f, 0.5f, 1.0f);
fColor = mix(fogColor, color, fogFactor);
```

Funcția **mix** interpolează liniar între două valori. Culoarea rezultată poate fi calculată și prin:

```
fColor = fogColor * (1 - fogFactor) + color * fogFactor;
```

Variabila **color** reprezintă culoarea calculată folosind sursele de lumină specificate în scenă.

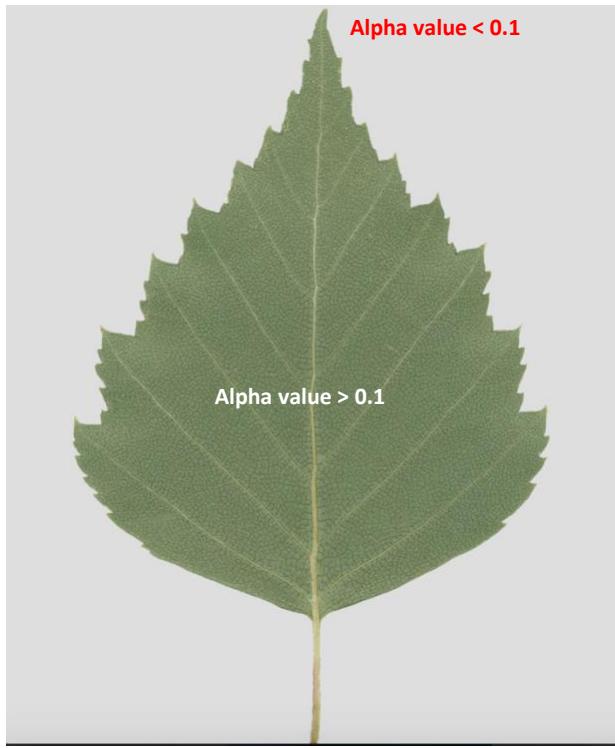
## 3 Eliminarea fragmentelor

Valoarea alfa din texturi poate fi utilizată pentru a elimina un fragment. De exemplu, doriți să afișați o frunză, iar modelul 3D este reprezentat de un quad pe care este mapată o textură a frunzei. Puteți verifica valoarea canalului alfa urmând să eliminați fragmentul dacă valoarea este mai mică de 0,1.

```
vec4 colorFromTexture = texture(diffuseTexture, interpolatedTexCoords);
if(colorFromTexture.a < 0.1)
    discard;
fColor = colorFromTexture;
```

Actualizați apelul funcției `glTexImage2D` în interiorul metodei `ReadTextureFromFile` (din clasa `Model3D`):

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, x, y, 0, GL_RGBA, GL_UNSIGNED_BYTE, image_data);
```



## 4 Combinarea culorilor

Puteți face ca obiectele să fie transparente (sau semi-transparente) prin combinarea culorii obiectului și a culorii actuale din framebuffer (reprezentând obiectele deja redate) într-o singură culoare. Cantitatea de transparență este definită de canalul alfa din reprezentarea culorii. Dacă a patra componentă a culorii este 1.0f, atunci obiectul este opac, dacă valoarea este 0.0f atunci obiectul este transparent. Valorile din intervalul (0.0f, 1.0f) reprezintă o cantitate diferită de transparență/opacitate.

Combinarea culorilor este activată în OpenGL utilizând următoarea funcție:

```
glEnable(GL_BLEND);
```

Culoarea rezultată după aplicarea acestei operații se calculează folosind următoarea ecuație:

$color = sourceColor * sourceFactor + destinationColor * destinationFactor$  unde,

- $sourceColor$  – reprezintă culoarea obiectului care este rasterizat în prezent
- $sourceFactor$  – definește influența culorii sursă
- $destinationColor$  – reprezintă culoarea din framebuffer
- $destinationFactor$  – definește influența culorii destinație

SourceFactor și destinationFactor sunt specificați folosind `glBlendFunction`. În general, folosim valoarea alfa a sursei ca  $sourceFactor$  și  $(1 - alpha)$  ca  $destinationFactor$ . Pentru aceasta trebuie să apelăm următoarea funcție:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Ordinea în care rasterizăm obiectele în această situație este foarte importantă. Când desenați o astfel de scenă, trebuie să:

- desenați toate obiectele opace
- sortați obiectele transparente (ordinea adâncimii)
- desenați obiectele transparente în ordine (ordinea adâncimii)

## 5 Temă

- Implementați toate noțiunile prezentate în această lucrare de laborator.