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 includeți 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 interactiunea dintre pointeri si 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; // inmultire 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 x 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 celelalte 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 x 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 o eroare de compilare, cu toate acestea, rezultatul va fi incorect.

```
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, reprezintând 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 rotatie cu 45.0 de grade in 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 translatie 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 rotatie cu 45.0 de grade in 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 translatie 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 normalizate ale unui poligon convex 2D (Figura 1). Poligonul va fi definit prin vârfurile sale.

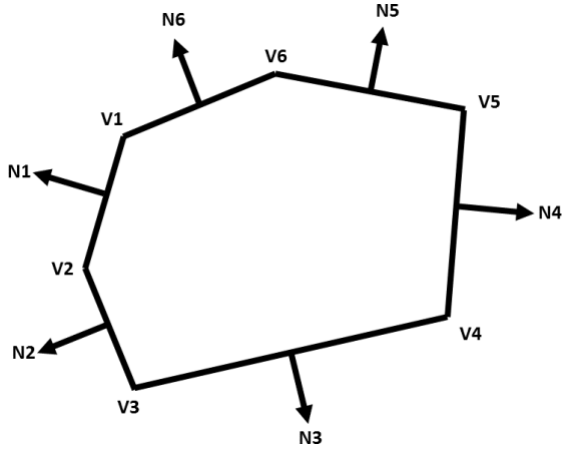

Figura 1 – Exemplu de poligon