Laboratory work 1

1 Objectives

The objective of this laboratory is to briefly describe the basic mathematics involved in Computer Graphics and how the OpenGL Mathematics (GLM) library can be used to implement math operations in an OpenGL application.

2 The GLM library

Open**GL M**athematics (GLM) is a header only C++ mathematics library for graphics software based on the specifications of the OpenGL Shading Language (GLSL), well suited for any development context that requires a simple and convenient mathematics library. This means that in order to include and use the library in our applications, we only have to include the proper header files, without any supplementary linking or compiling. The latest version of GLM can be downloaded from their website, http://glm.g-truc.net.

GLM is written in C++98 but can take advantage of C++11 when supported by the compiler. It is a platform independent library with no dependence and it officially supports the following compilers:

- Apple Clang 4.0 and higher
- GCC 4.2 and higher
- Intel C++ Composer XE 2013 and higher
- LLVM 3.0 and higher
- Visual C++ 2010 and higher
- CUDA 4.0 and higher (experimental)
- Any conform C++98 or C++11 compiler

In order to use GLM, we only have to include **glm.hpp** in our applications, providing that the root folder of GLM is in the Include Path.

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

Core GLM features can be included using individual headers to allow faster user program compilations.

```
<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
```

Most GLM functionality that we will require can be accessed by including the following three headers:

```
#include <glm/glm.hpp> //core glm functionality
#include <glm/gtc/matrix_transform.hpp> //glm extension for generating common
transformation matrices
#include <glm/gtc/type_ptr.hpp> //glm extension that handles interaction between
pointers and vector and matrix types
```

3 Vector operations

In order to use GLM vectors, we have to include the corresponding GLM headers:

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

or include only the vector headers (optimized)

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

3.1 Declaring a GLM vector

We can now declare a vector variable, for example a **column** vector with 4 elements:

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

Vectors with 2 or 3 elements are declared in a similar way, using the vec2 and vec3 data types.

3.2 Vector arithmetic

GLM supports all vector operations through operator overloading:

```
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; //addition or subtraction of a scalar
newVector3 = 2.0f * newVector1; //multiplication with a scalar
newVector3 = newVector1 + newVector2; //vector addition or subtraction
newVector3 = -newVector1; //vector negation
```

3.3 Vector length and vector-vector multiplication

The length of a vector can be obtained by using the **glm::length()** function. Dot product and cross product can be computed by using **glm::dot()** and **glm::cross()** respectively. For vector normalization we can use **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); //length of a vector
float dotProduct = glm::dot(newVector1, newVector2); //dot product
glm::vec3 newVector4 = glm::cross(glm::vec3(1.0f, 2.0f, 3.0f), glm::vec3(3.0f, 2.0f,
1.0)); //cross product
glm::vec3 normalizedVector = glm::normalize(newVector1); //normalize newVector1 and
save the result in another variable
```

Note: The **length()** member function of the **vec*** classes (e.g. **newVector1.length()**) returns the number of elements in the vector (2,3 or 4). Always use **glm::length()** to compute the magnitude of a vector.

4 Matrices and Transformations

In order to use GLM matrices, we have to include the corresponding GLM headers:

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

or include only the matrix headers (optimized)

```
#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 Declaring a GLM matrix

We can now declare a matrix variable, for example a 2 x 3 **column-major** matrix:

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

Other types of matrices can be defined in a similar way using the other matrix data types that GLM provides. Square matrices can also be defined using the mat2, mat3 and mat4 data types. Square matrices can be initialized with the same value on the diagonal using constructors with a single parameter. E.g. an identity matrix of size 4 x 4 can be declared as:

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

The default constructor for all GLM matrix types creates a matrix with elements of 1 on the main diagonal and 0 on all other positions.

4.2 Matrix arithmetic

GLM supports all matrix operations through operator overloading:

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

newMatrix3 = 2.0f + newMatrix1; //addition or subtraction of a scalar
newMatrix3 = 2.0f * newMatrix1; //multiplication with a scalar
newMatrix3 = newMatrix1 + newMatrix2; //matrix addition or subtraction
newMatrix3 = -newMatrix1; //matrix negation
newMatrix3 = newMatrix1 * newMatrix2; //matrix multiplication
```

4.3 Matrix-vector multiplication

Since GLM constructs **column-major** matrices, a matrix and a vector can be multiplied only **M * v**, where **M** is the matrix and **v** is the vector. The reverse operation, **v * M**, will not generate a compile error, however, the result will not be correct.

```
glm::vec4 newVector1(1.0f, 2.0f, 3.0f, 1.0f);
glm::mat4 newMatrix1(1.0f);
glm::vec4 newVector3;

newVector3 = newMatrix1 * newVector1; //matrix-vector multiplication
```

4.4 Transformation matrices

GLM offers an extension for generating the most commonly used transformation matrices. In order to use it, we have to include the corresponding GLM headers:

```
#include <glm/gtc/matrix_transform.hpp> //glm extension for generating common
transformation matrices
```

We can now generate transformation matrices using standard OpenGL fixed-function conventions.

```
glm::mat4 oldTransformationMatrix;
glm::mat4 newTransformationMatrix;

newTransformationMatrix = glm::translate(oldTransformationMatrix, glm::vec3(1.0f,
2.0f, 3.0f)); //generate a translation matrix with (1.0f, 2.0f, 3.0f)
newTransformationMatrix = glm::scale(oldTransformationMatrix, glm::vec3(2.0f, 1.0f,
3.0f)); //generate a scale matrix with (2.0f, 1.0f, 3.0f)
newTransformationMatrix = glm::rotate(oldTransformationMatrix, glm::radians(45.0f),
glm::vec3(0.0f, 1.0f, 0.0f)); //generate a rotation matrix with 45.0 degrees around
the (0.0f, 1.0f, 0.0f) axis (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)); //generate a camera transformation
(camera position, reference point, up vector)
```

```
newTransformationMatrix = glm::perspective(45.0f, (GLfloat)screen_width /
(GLfloat)screen_height, 1.0f, 1000.0f); //generate a perspective projection matrix
(vertical fov, aspect ratio, zNear, zFar)
```

The functions that generate model transformations (translation, scale, rotation) take a supplementary parameter that represents an old transformation matrix that will be used to generate the new transformation matrix following the rule newTransformationMatrix = oldTransformationMatrix*<the generated transformation matrix>. This allows for simple composition of transformations. For example, in order to generate a matrix containing the above translation, scale and rotation (in this order), we can write:

```
glm::mat4 translationScaleRotation(1.0f); //start with the identity matrix

translationScaleRotation = glm::rotate(translationScaleRotation, glm::radians(45.0f),
glm::vec3(0.0f, 1.0f, 0.0f)); //generate a rotation matrix with 45.0 degrees around
the (0.0f, 1.0f, 0.0f) axis (Y axis)
translationScaleRotation = glm::scale(translationScaleRotation, glm::vec3(2.0f, 1.0f,
3.0f)); //generate a scale matrix with (2.0f, 1.0f, 3.0f)
translationScaleRotation = glm::translate(translationScaleRotation, glm::vec3(1.0f,
2.0f, 3.0f)); //generate a translation matrix with (1.0f, 2.0f, 3.0f)
```

This will generate a **translationScaleRotation** matrix equal with **Identity*rotate*scale*translate**. Remember that, when composing transformations, the first transformation to be applied to a point is the last one to be included in the composed transformation matrix.

To apply a transformation to a point, we have to multiply the transformation matrix and the point:

```
glm::mat4 translationScaleRotation(1.0f); //start with the identity matrix
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)); //generate a rotation matrix with 45.0 degrees around
the (0.0f, 1.0f, 0.0f) axis (Y axis)
translationScaleRotation = glm::scale(translationScaleRotation, glm::vec3(2.0f, 1.0f,
3.0f)); //generate a scale matrix with (2.0f, 1.0f, 3.0f)
translationScaleRotation = glm::translate(translationScaleRotation, glm::vec3(1.0f,
2.0f, 3.0f)); //generate a translation matrix with (1.0f, 2.0f, 3.0f)

transformedPoint = translationScaleRotation * originalPoint; //transform the original
point and save the transformed point in a new variable
```

5 Further reading

- GLM documentation – http://glm.g-truc.net/

6 Assignment

1. Download the GLM library

2. Create a new Microsoft Visual Studio project (C++ Win32 console application) and import the GLM library
3. Include in this project the files from the laboratory web page.
4. Implement a function called **TransformPoint** that translates a 3D point in homogeneous coordinates with (2.0f, 0.0f, 1.0f) and then rotates it with 90 degrees around the X axis. The function should receive the point as a parameter and should return the transformed point.
5. Implement a function called **ComputeAngle** that computes and returns the angle between two vectors *v1* and *v2* passed as parameters.
6. Implement a function called **IsConvex** that tests if a 2D polygon is convex or concave. The polygon will be defined through its vertices. The function will return true if the polygon is convex and false if the polygon is concave.
7. Implement a function called **ComputeNormals** that computes and returns the normalized normals of a convex 2D polygon (Figure 1). The polygon will be defined through its vertices.

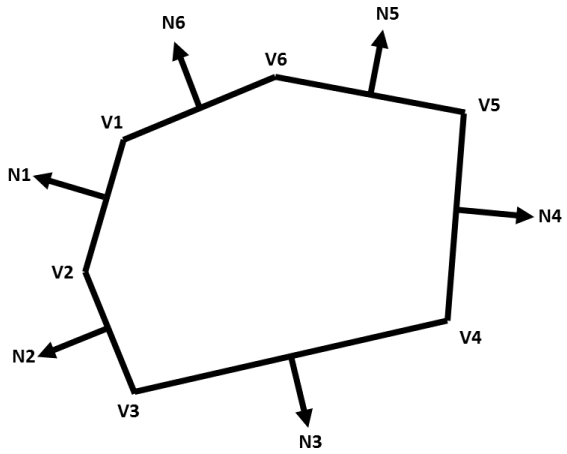

Figure 1 - Example polygon