

Adrian SABOU

Constantin Ioan NANDRA

Graphic Processing

Laboratory Works



U.T.PRESS
Cluj-Napoca, 2026
ISBN 978-606-737-860-3

Adrian SABOU

Constantin Ioan NANDRA

Graphic Processing

Laboratory Works



U.T. PRESS

Cluj-Napoca, 2026

ISBN 978-606-737-860-3



Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Recenzia: Prof.dr.ing. Dorian Gorgan

Conf.dr.ing. Victor Bâcu

Pregătire format electronic on-line: Gabriela Groza

Copyright © 2026 Editura U.T.PRESS

Reproducerea integrală sau parțială a textului sau ilustrațiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

ISBN 978-606-737-860-3

Preface

This book contains 11 laboratory works related to the computer graphics domain. Its target audience are the students from the third year of the Computer Science department from the Automation and Computer Science faculty, Technical University of Cluj-Napoca, but it can be used by any engineer interested in this domain.

The content follows the structure of the Graphic Processing course taught at the Technical University of Cluj-Napoca.

Each laboratory work is structured into three main sections. The first section presents the objectives and what students are supposed to learn, while the second section offers an overview of the theoretical background supporting the presented material along with step-by-step tutorials. The last section contains assignments.

Cluj-Napoca,
29.09.2026

Authors

Contents

Laboratory work 1 – The GLM library	1
Laboratory work 2 – Introduction to OpenGL	7
Laboratory work 3 – Dataflow in the OpenGL pipeline	14
Laboratory work 4 – Debugging in OpenGL	22
Laboratory work 5 – Transformations in OpenGL	28
Laboratory work 6 – Loading 3D objects	38
Laboratory work 7 – Lighting models.....	47
Laboratory work 8 – Advanced lighting.....	56
Laboratory work 9 – Shadow Mapping.....	62
Laboratory work 10 – Cube maps.....	74
Laboratory work 11 – Effects in OpenGL.....	78

Laboratory work 1 – The GLM library

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.

The GLM library

OpenGL Mathematics (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
```

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>
```

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.

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
```

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, 1.0f)); //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.

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>
```

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.

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
```

Matrix-vector multiplication

Since GLM constructs **column-major** matrices, a matrix and a vector can be multiplied only $\mathbf{M} * \mathbf{v}$, where $\mathbf{M}$ is the matrix and $\mathbf{v}$ is the vector. The reverse operation, $\mathbf{v} * \mathbf{M}$, 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
```

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, 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 $\text{newTransformationMatrix} = \text{oldTransformationMatrix} * \text{the}$

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, 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, 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
```

Further reading

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

Assignment

1. Download the GLM library
2. Create a new Microsoft Visual Studio project (C++ Win32 console application)
3. Include the files from the laboratory webpage (Moodle) in this project.
4. Implement a function named **TransformPoint** that translates a 3D point by the vector (2.0f, 0.0f, 1.0f) and then rotates it 90 degrees around the X-axis. The function should accept the point as a parameter and return the transformed point.

5. Implement a function named **ComputeAngle** that computes and returns the angle between two vectors, $v1$ and $v2$, passed as parameters.
6. Implement a function named **IsConvex** that tests whether a 2D polygon is convex or concave. The polygon is defined by its vertices. The function should return true if the polygon is convex and false if it is concave.
7. Implement a function named **ComputeNormals** that computes and returns the normalized normals of a 2D convex polygon (Figure 1). The polygon is defined by its vertices.

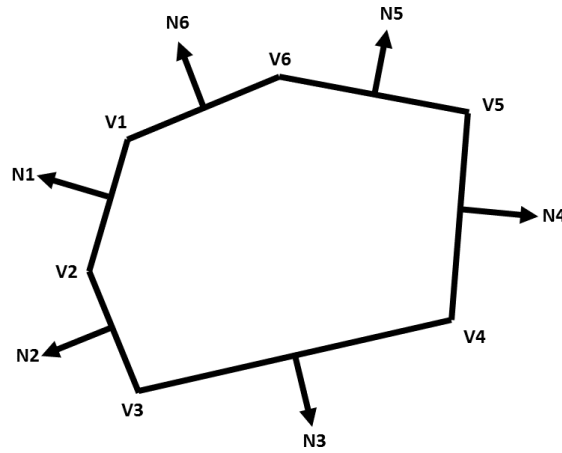


Figure 1 - Example polygon

Laboratory work 2 – Introduction to OpenGL

Objectives

The objective of this laboratory is to present a brief introduction to the **OpenGL programmable pipeline**.

Theoretical background

OpenGL is considered an Application Programming Interface (API) and is used to develop graphical applications by accessing features available in the graphics hardware. However, OpenGL is a specification developed and maintained by the Khronos Group. The OpenGL specification describes what is the desired result or output of each function. The actual implementation can be different between various OpenGL libraries. These libraries are implemented mainly by the graphics card manufacturers.

The first versions of OpenGL used a so called **fixed function pipeline**, in which most of the functionality was predefined and the developers could only modify some parameters but not change the actual algorithms used to transform 3D objects to 2D images. This pipeline is very easy to understand, but on the other hand is very inefficient. Starting with OpenGL 3.2, this fixed function pipeline has become deprecated, developers preferring instead the larger degree of flexibility offered by the **programmable pipeline**.

There are some changes in writing OpenGL applications for the programmable pipeline compared to the fixed pipeline:

- *No immediate mode drawing* – `glBegin()` and `glEnd()` are no longer supported in order to specify graphical primitives
- *All data must be stored in buffer objects* – such as Vertex Buffer Objects (VBOs) or Vertex Array Objects (VAOs)
- *No transformations* – `glTranslatef()`, `glRotatef()`, `glScalef()`, `gluLookAt()`, `gluPerspective()` are no longer supported
- *No lighting* – `glLight*()` and `glMaterial*()` are no longer supported and you have to implement them in shaders

OpenGL operates as a large state machine, consisting of a collection of variables that define how it should operate. The OpenGL context represents the current state. Often, before rendering using the current context, we change the state by modifying some parameters and also manipulating some buffers. Being independent of the operating system or windowing system used, it offers flexibility for developing cross-platform applications. By writing small programs (called **shaders**) we can use the GPUs at their full capacity, compared to the fixed pipeline. Shaders are attached to an OpenGL applications and run on the GPU. They are written in OpenGL Shading Language (GLSL), which is very similar to the C language.

The rendering pipeline implemented in OpenGL is illustrated in the figure below. The programmable pipeline contains at least the **vertex shading** and the **fragment shading** stages, and replaces the fixed-pipeline, which is no longer supported (lighting and transformations). The vertex shading stage processes each vertex independently of the other ones; vertex data is received from the application through vertex buffer objects. The rasterization phase generates a set of fragments which are processed in the fragment shading stage; the output is represented by the fragment's color and depth. Some stages (such as primitive setup, clipping and rasterization) are not programmable. Tessellation Shader and Geometry Shader are optional.

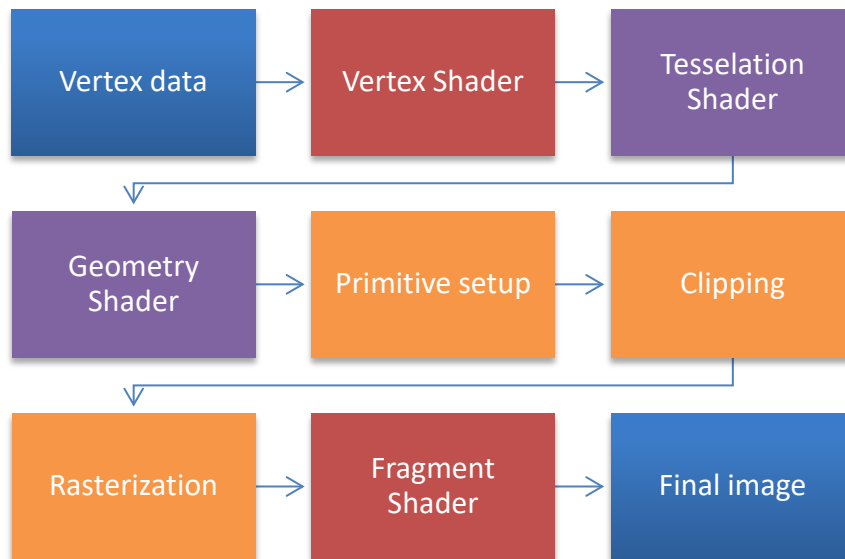


Figure 2 - OpenGL programmable pipeline

A basic OpenGL 4 application

Libraries

We are using the **GLFW** library instead of GLUT to create a window with OpenGL contexts. GLFW is written in C and has native support for Windows, OS X and many Unix-like systems. You can download pre-compiled binaries or the source code from <http://www.glfw.org>. A step by step tutorial on compiling the library on various platforms can be found [here](#).

The OpenGL Extension Wrangler Library (**GLEW**) is a cross-platform open-source C/C++ extension loading library. You can download pre-compiled binaries or the source code from <http://glew.sourceforge.net/>. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

Representation of 3D objects

A 3D object is represented using vertices and topological information. For each vertex we need attributes such as position, color, texture coordinate and other relevant information. Vertex data is stored in Vertex Buffer Objects (VBOs). A series of VBOs can be stored in a Vertex Array Object (VAO).

A **Vertex Buffer Object (VBO)** is a buffer used to hold an array of vertex attributes (such as position, color, normal vector, etc.). A **Vertex Array Object (VAO)** groups together multiple VBOs. In order to use VBOs we need to:

- generate VBO names (IDs): *glGenBuffers(...)*
- bind (select) a specific VBO for initialization: *glBindBuffer(GL_ARRAY_BUFFER, ...)*
- load data into the VBO: *glBufferData(GL_ARRAY_BUFFER, ...)*

A similar procedure is needed for Vertex Array Objects:

- generate VAO names (IDs): *glGenVertexArrays(...)*
- bind (select) a specific VAO for initialization: *glBindVertexArray(...)*
- update VBOs associated with this VAO: *glBindBuffer(...)* and *glVertexAttribPointer(...)*
- bind VAO for use in rendering: *glDrawArrays(...)*

Vertex and fragment shaders

In order to be able to render 3D objects, we need to write at least a vertex shader and a fragment shader. For this we need to:

- generate a shader object (referenced by its unique ID): *glCreateShader(...)*
- attach the shader source code to the shader object: *glShaderSource(...)*
- compile the shader: *glCompileShader(...)*

After we have compiled the vertex and fragment shaders we need to combine them in a so called shader program object. The steps for this are the following:

- create a shader program: *glCreateProgram(...)*
- attach the previous compiled shaders: *glAttachShader(...)*
- link them in the shader program: *glLinkProgram(...)*

Tutorial

This section presents a basic OpenGL 4 application. The starting source code is available on the laboratory web page.

Step 1

Open and run the OpenGL_4_Application project. It is a basic application that just creates an OpenGL window without rendering anything.

Step 2

We start by specifying a triangle, and for this, we need to define the coordinates of three vertices, a variable to store the Vertex Buffer Object ID and another one to store the Vertex Array Object ID.

```
//vertex coordinates in normalized device coordinates
GLfloat vertexCoordinates[] = {
    0.0f,  0.5f,  0.0f,
    0.5f, -0.5f,  0.0f,
    -0.5f, -0.5f,  0.0f
};
```

```
GLuint verticesVBO;  
GLuint triangleVAO;
```

Add the following function, which will initialize the OpenGL objects.

```
void initObjects()  
{  
    //generate a unique ID corresponding to the verticesVBO  
    glGenBuffers(1, &verticesVBO);  
    //bind the verticesVBO buffer to the GL_ARRAY_BUFFER target,  
    //any further buffer call made to GL_ARRAY_BUFFER will configure the  
    //currently bound buffer, which is verticesVBO  
    glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);  
    //copy data into the currently bound buffer, the first argument specify  
    //the type of the buffer, the second argument specify the size (in bytes) of data,  
    //the third argument is the actual data we want to send,  
    //the last argument specify how should the graphic card manage the data  
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexCoordinates), vertexCoordinates, GL_STATIC_DRAW);  
  
    //generate a unique ID corresponding to the triangleVAO  
    glGenVertexArrays(1, &triangleVAO);  
    glBindVertexArray(triangleVAO);  
    glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);  
    //set the vertex attributes pointers  
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);  
    glEnableVertexAttribArray(0);  
    //unbind the triangleVAO  
    glBindVertexArray(0);  
}
```

Step 3

The vertex and fragment shader programs are already included in the project. The source code is laid out below. Do not worry about their content at this stage. In the next laboratory work, we will analyze the content and structure of these shader programs.

Vertex shader

```
#version 400  
  
layout(location = 0) in vec3 vertex_position;  
  
out vec3 colour;  
  
void main() {  
    //specify the vertex color  
    colour = vec3(1.0, 0.0, 0.0);  
    gl_Position = vec4(vertex_position, 1.0);  
}
```

Fragment shader

```
#version 400

in vec3 colour;
out vec4 frag_colour;

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

Step 4

Next we need to read, compile and link the shader programs. The following function is already present in the source code; you don't need to add it.

```
GLuint initBasicShader(std::string vertexShaderFileName, std::string fragmentShaderFileName)
{
    //read, parse and compile the 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);
    //check compilation status
    shaderCompileLog(vertexShader);

    //read, parse and compile the vertex 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);
    //check compilation status
    shaderCompileLog(fragmentShader);

    //attach and link the shader programs
    shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    //check linking info
    shaderLinkLog(shaderProgram);

    return shaderProgram;
}
```

```
}
```

Step 5

Finally, we need to specify which primitive OpenGL should render. For this purpose, we use the `glDrawArrays` function, which employs the current active shader, VAO and VBOs in order to draw the specified primitive.

```
void renderScene()
{
    //clear the color and depth buffer before rendering the current frame
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //specify the background color
    glClearColor(0.8, 0.8, 0.8, 1.0);
    //specify the viewport location and dimension
    glViewport (0, 0, retina_width, retina_height);

    //process the keyboard inputs
    if (glfwGetKey(glfwWindow, GLFW_KEY_A)) {
        //TODO
    }

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

    //bind the shader program, any further rendering call
    //will use this shader program
    glUseProgram(shaderProgram);

    //bind the VAO
    glBindVertexArray(triangleVAO);
    //specify the type of primitive, the starting index and
    //the number of indices to be rendered
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Further reading

- OpenGL Programming Guide 8th edition – chapter 1

Function (and link)	Description
glGenBuffers	generates buffer object names
glBindBuffer	binds a buffer object to the specified buffer binding point
glBufferData	creates and initializes a buffer object's data store
glGenVertexArrays	generates vertex array object names
glBindVertexArray	binds a vertex array object

<u>glVertexAttribPointer</u>	defines an array of generic vertex attribute data
<u>glEnableVertexAttribArray</u>	enables a generic vertex attribute array
<u>glCreateShader</u>	creates a shader object
<u>glShaderSource</u>	replaces the source code in a shader object
<u>glCompileShader</u>	compiles a shader object
<u>glCreateProgram</u>	creates a program object
<u>glAttachShader</u>	attaches a shader object to a program object
<u>glLinkProgram</u>	links a program object
<u>glDeleteShader</u>	deletes a shader object
<u>glUseProgram</u>	installs a program object as part of current rendering state
<u>glDrawArrays</u>	renders primitives from array data

Assignment

8. Define and display two triangles. Define different buffer objects for them (VBO and VAO) and arrange them in such a manner that they form a square.
9. Define a new shader program that uses a different fragment shader. Set the rendering color to green for this new fragment shader. Display one triangle using the old shader program and the other one using the newly defined shader program.
10. Change the shader program used to render each triangle by using different keyboard inputs.

Laboratory work 3 – Dataflow in the OpenGL pipeline

Objectives

The objective of this laboratory is to present different methods in which vertex data (attributes) is transferred between the application (CPU) and GPU levels.

Theoretical background

OpenGL Shaders

The programmable pipeline in OpenGL relies on shaders, which are small programs written in the GLSL language (very similar with C, and oriented towards graphics operations). This language contains useful features related to vector and matrix operations. Every shader starts with a version declaration, followed by a list of input and output variables, uniforms and at least the main function (which is the entry point). The typical structure of a shader is the following:

```
//OpenGL version
#version versionNumber

//list of input variables
in type inputVariableName;

//list of output variables
out type outputVariableName;

//list of uniforms
uniform type uniformName;

void main()
{
    //implements the functionality and output the variables
    outputVariableName = processVariable();
}
```

Data types

In GLSL, we can use most of the default basic data types, such as int, float, double, uint and bool. Besides these basic types, GLSL also supports vectors and matrices. A vector represents a container for any of the basic types and may have different dimensions - of 1,2,3 or 4 elements. We can define vectors like this (usually we will be using **vecn**):

- **vecn**: vector of n floats

- **bvecn**: vector of n booleans
- **ivec n**: vector of n integers
- **uvecn**: vector of n unsigned integers
- **dvecn**: vector of n double components

In order to access different components from the vectors, we will be using **.x**, **.y**, **.z** and **.w** (first, second, third and fourth component). For code readability, we can use **.r**, **.g**, **.b** and **.a** for colors - or **.s**, **.t**, **.p** and **.q** for texture coordinates.

Inputs and outputs

The inputs and outputs are specified by the keywords **in** and **out**.

If the name of an input variable (from one shader) matches the name of an output variable (from another shader) then the data passes from one shader to the other.

The inputs of the vertex shader represent vertex attributes (could be position, color, normal vector, etc.). In order to match the vertex attributes specified at the CPU level with the ones defined at the GPU level, we use the following statement: **layout (location = locationID).**

At the CPU level, we use the following functions:

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

At the GPU level, we specify the input variable:

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

The output of the fragment shader should be a vec4 color output variable. Otherwise, the result (image) will be incorrect.

Uniforms

Another way of transferring data from our application (which runs on the CPU) to our shaders (which run on the GPU) is by **uniforms**. They are global (meaning unique for a shader program object), and can be accessed by any shader of the shader program. In addition, their value will remain the same until we modify it.

To declare a uniform at the shader level, we use:

```
uniform type uniformName;
```

In order to update the uniform, we need (at application level) to get the uniform location and then we can change its value.

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

To update the uniform value, we use the function **glUniform***, which has multiple definitions (different by the postfix):

- **f**: expecting a float
- **i**: expecting an int
- **ui**: expecting an unsigned int
- **3f**: expecting 3 floats
- **fv**: expecting a float vector/array

Vertex attributes

In the VBO we can store more attributes than just the vertices' position. For instance, we can store color, normal vector, texture coordinates, etc. Consider the following example, in which we have two attributes for each vertex (position and color).

Vertex 1						Vertex 2					
X	Y	Z	R	G	B	X	Y	Z	R	G	B
0... 3	4... 7	8...1 1	12...1 5	16...1 9	20...2 3	24...2 7	28...3 1	32...3 5	36...3 9	40...4 3	44...4 7
Vertex position stride: 24 Vertex offset: 0						Vertex color stride: 24 Vertex color offset: 12					

We need to modify the vertex shader to receive the color values, and for this, we need to set the corresponding location:

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

We need to change also the way in which we specify the vertices attributes in our program:

```
//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* - number of components (1, 2, 3, 4)
- *type* - data type of each component (GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, and GL_UNSIGNED_INT)
- *normalized* - specifies whether fixed-point data values should be normalized or not
- *stride* - the byte offset between consecutive generic vertex attributes
- *pointer* - the offset of the first component of the first generic vertex attribute

Element Buffer Objects

In order to avoid duplicating (re-defining) some vertices, we can use EBOs to store indices. Indices will be used by OpenGL to decide what vertices to choose from when forming polygons. The procedure of creating EBOs is very similar to the one used to define VBOs:

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

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

When calling `glDrawElements`, we draw using the indices provided within the EBO currently bound. The first parameter specifies the type of the primitive, the second represents the number of elements we draw (vertices), the third parameter represents the type of indices (typically `GL_UNSIGNED_INT`) and the last one is an offset.

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

Tutorial

Transferring data from vertex to fragment shader

Start from the solution that you find on the website and modify the vertex and fragment shaders. Remember that if the name of an input variable (from one shader) matches the name of an output variable (from another shader), then the data passes from one shader to the other one.

The code from **vertex shader**:

```
#version 400

layout(location = 0) in vec3 vertexPosition;

//the color variable that we will send to the fragment shader
out vec3 colour;

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

The code from **fragment shader**:

```
#version 400

//the color variable that we received from the vertex shader
in vec3 colour;

out vec4 fragmentColour;

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

Sending data using uniforms

Start from the solution that you find on the website, and modify the vertex and fragment shaders.

The vertex shader:

```
#version 400

layout(location = 0) in vec3 vertexPosition;

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

The fragment shader:

```
#version 400

out vec4 fragmentColour;

uniform vec3 uniformColour;

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

Modify the main function to get the uniform location from the shader program and update the uniform value:

```
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;
}
```

Element Buffer Object and attributes example

Start from the solution that you find on the website. In this example, we want to draw a rectangle composed of two triangles. Instead of duplicating some of the vertices' coordinates, we will use an Element Buffer Object in order to save storage space. Define, globally, the vertex data (representing position and color) and the buffers that will be used for it.

```
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;

```

We need to update the initialization of our object. The difference is that, in this case, we have also indices and a new attribute for each vertex. For details about the parameters needed for each function, please read again the “theoretical background” section and also the “further reading” section.

```

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);
}

```

In the renderScene() function, we need to update the drawing call to 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);
}

```

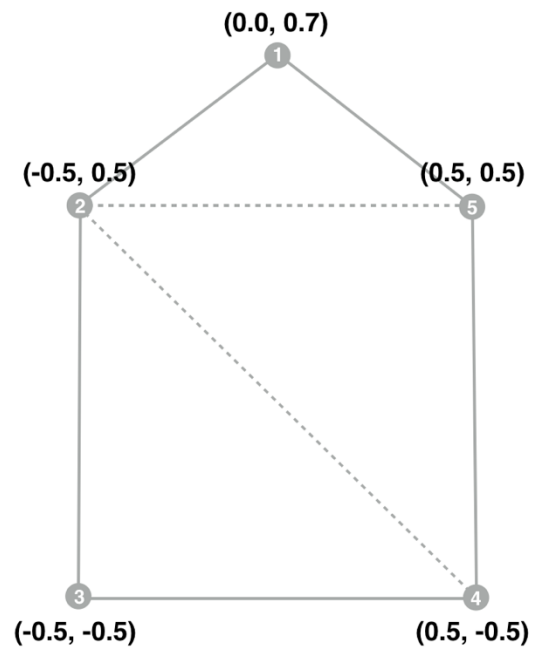
Further reading

- OpenGL Programming Guide 8th edition – chapters 2 and 3

Function (and link)	Description
glGenBuffers	generates buffer object names
glBindBuffer	binds a buffer object to the specified buffer binding point
glBufferData	creates and initializes a buffer object's data store
glGenVertexArrays	generates vertex array object names
glBindVertexArray	binds a vertex array object
glVertexAttribPointer	defines an array of generic vertex attribute data
glEnableVertexAttribArray	enables a generic vertex attribute array
glCreateShader	creates a shader object
glShaderSource	replaces the source code in a shader object
glCompileShader	compiles a shader object
glCreateProgram	creates a program object
glAttachShader	attaches a shader object to a program object
glLinkProgram	links a program object
glDeleteShader	deletes a shader object
glUseProgram	installs a program object as part of the current rendering state
glDrawArrays	renders primitives from array data
glGetUniformLocation	returns the location of a uniform variable
glUniform	specifies the value of a uniform variable for the current program object
glDrawElements	renders primitives from array data

Assignment

11. Render the following figure using EBO and define different colors for each vertex. Do not replicate the vertices!!!



12. Modify the vertices coordinates (on x and y axis) using two uniforms that are sent to the vertex shader. Update the uniforms' values using different keyboard inputs.
13. Switch between flat and wireframe representation by using different keyboard inputs.
 - To change the rendering to a wireframe representation, you can use the following function:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```
 - To change back the rendering to a flat representation, you the following function:

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

Laboratory work 4 – Debugging in OpenGL

Objectives

The objective of this laboratory is to present different methods of debugging OpenGL applications.

Theoretical background

glGetError()

OpenGL will generate error flags every time there is an incorrect use of functions or functionality. You can query for error flags using the `glGetError()` function. Every error has assigned a numeric code and symbolic name. When something goes wrong the error flag is set (with the appropriate error code value). No other errors are set until `glGetError` is called. On each call the error code is returned and the flag is reset. The following error codes are generated:

Error code	Error flag	Error description
0	GL_NO_ERROR	No error has been recorded. The value of this symbolic constant is guaranteed to be 0.
1280	GL_INVALID_ENUM	An unacceptable value is specified for an enumerated argument. The offending command is ignored and has no other side effect than to set the error flag.
1281	GL_INVALID_VALUE	A numeric argument is out of range. The offending command is ignored and has no other side effect than to set the error flag.
1282	GL_INVALID_OPERATION	The specified operation is not allowed in the current state. The offending command is ignored and has no other side effect than to set the error flag.
1286	GL_INVALID_FRAMEBUFFER_OPERATION	The framebuffer object is not complete. The offending command is ignored and has no other side effect than to set the error flag.
1285	GL_OUT_OF_MEMORY	There is not enough memory left to execute the command. The state of the GL is undefined, except for the state of the error flags, after this error is recorded.
1284	GL_STACK_UNDERFLOW	An attempt has been made to perform an operation that would cause an internal stack to underflow.

1283	GL_STACK_OVERFLOW	An attempt has been made to perform an operation that would cause a stack to overflow.
-------------	-------------------	--

To check for errors, use the following function:

```
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__)
```

Debugging at shader level

Unfortunately, at shader level we cannot use functions such as `glGetError()` and we cannot do a step-by-step debugging through our shader program. One solution to overcome this restriction is to display to fragment shader's output channel some variables. For instance, we can display the normal vectors and see if they are correct.

Update the vertices data to incorporate the normal vectors:

```
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;

```

Change the initObjects function in order to send the normal attribute to the vertex shader.

Update the 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);
}

```

Update the fragment shader:

```

#version 400

in vec3 colour;
in vec3 normal;

out vec4 fragmentColour;

void main() {
    fragmentColour.rgb = normal;
    fragmentColour.a = 1.0;
}

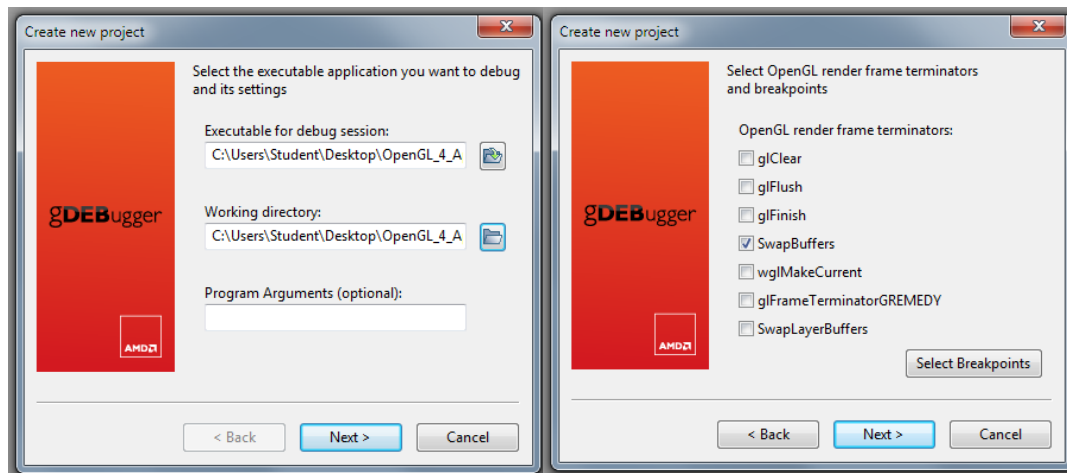
```

Question: Why is the object now blue?

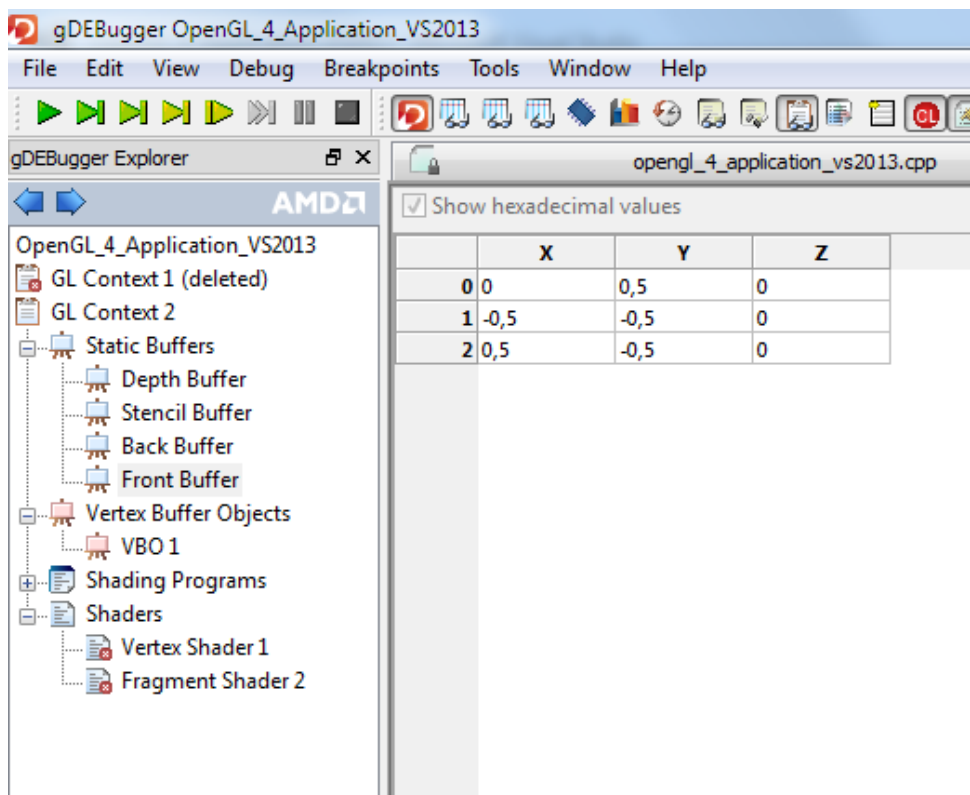
Third-party application

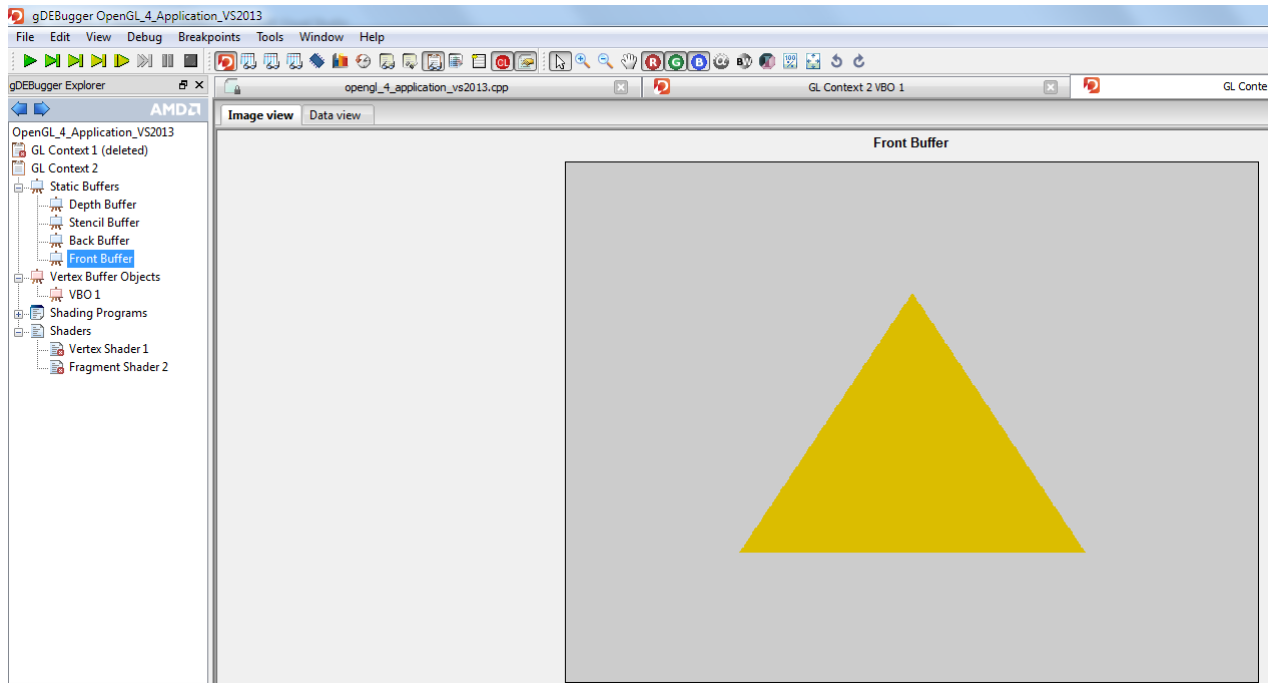
Third party applications are able to intercept different OpenGL calls and can be used for profiling OpenGL function usage, finding bottlenecks, inspecting buffer memory and displaying textures and framebuffer attachments. gDebugger is one of the debugging tools available for OpenGL applications. It is a cross-platform application that provides a detailed overview of the running OpenGL state.

In order to debug an OpenGL application using gDebugger you have to create a new project and select the program location and the working directory of your OpenGL executable.



By pausing the application at any moment, you can inspect the current state, texture memory and/or buffer usage.





Tips and tricks

What to do if your OpenGL application renders just a black screen:

- Check for errors using `glGetError()`
- Check if the geometry is inside the view frustum
- Change the background color to be different than black
- Check if the vertex attributes are set up
- Check if uniforms are set

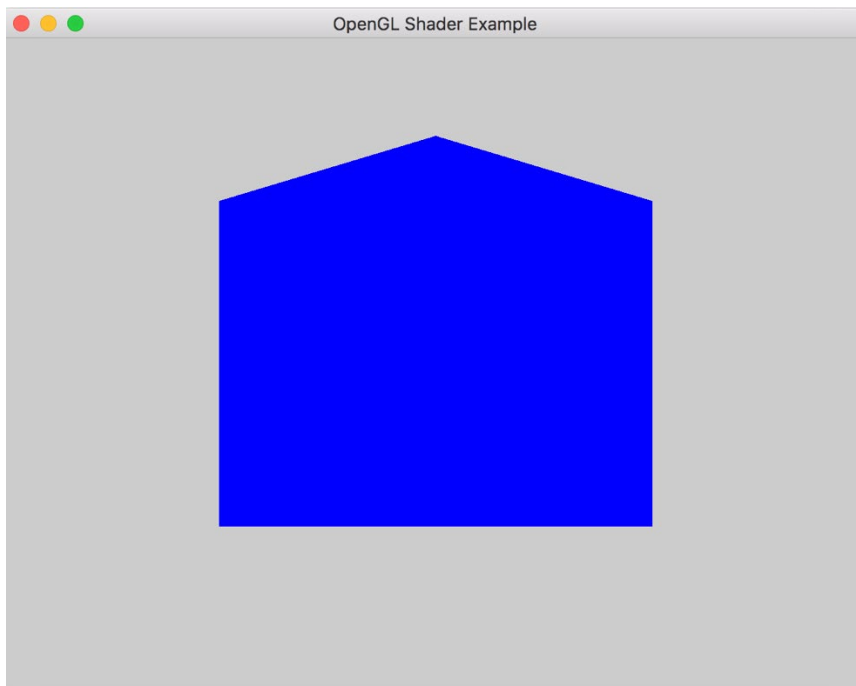
Further reading

Function (and link)	Description
<code>glGenBuffers</code>	generates buffer object names
<code>glBindBuffer</code>	binds a buffer object to the specified buffer binding point
<code>glBufferData</code>	creates and initializes a buffer object's data store
<code>glGenVertexArrays</code>	generates vertex array object names
<code>glBindVertexArray</code>	binds a vertex array object
<code>glVertexAttribPointer</code>	defines an array of generic vertex attribute data
<code>glEnableVertexAttribArray</code>	enables a generic vertex attribute array
<code>glCreateShader</code>	creates a shader object
<code>glShaderSource</code>	replaces the source code in a shader object
<code>glCompileShader</code>	compiles a shader object
<code>glCreateProgram</code>	creates a program object
<code>glAttachShader</code>	attaches a shader object to a program object
<code>glLinkProgram</code>	links a program object
<code>glDeleteShader</code>	deletes a shader object

<u>glUseProgram</u>	installs a program object as part of the current rendering state
<u>glDrawArrays</u>	renders primitives from array data
<u>glGetUniformLocation</u>	returns the location of a uniform variable
<u>glUniform</u>	specifies the value of a uniform variable for the current program object
<u>glDrawElements</u>	renders primitives from array data
<u>glGetError</u>	returns the value of the error flag

Assignment

- Download the files from the web site and update the previous project with them. The program should compile but it will display a grey window. Try to debug the program in order to display the following:



Laboratory work 5 – Transformations in OpenGL

Objectives

This laboratory presents the key notions on 3D transformations. It will cover the basic model (translation, scale, rotation), camera and projection transformations.

Defining 2D and 3D points

A 2D point is defined in a homogenous coordinate system by $(x*w, y*w, w)$. For simplicity, in bi-dimensional systems the w parameter is set to 1. Therefore, the point definition is $(x, y, 1)$.

Another representation for the point is the following: $P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$.

Similarly, in 3D we define points as $(x*w, y*w, z*w, w)$ and we represent the point as a column vector:

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

Transformations

After being processed by the Vertex Shader, the vertices that are to be rendered are expected to have their coordinates fit within the $[-1, +1]$ range – also called Normalized Device Coordinates (NDC). In order to get the vertices to this stage, they must undergo a series of transformations, bringing them from the Local Object Space to the World Coordinates Space, on to the View Coordinates Space and finally to the Clipping Space.

Modeling transformations

When you define an object, vertex-by-vertex, you do so with respect to a local coordinates system. Most often, for convenience, you define your objects centered around this system's origin – this is standard practice for most object modeling tools. When placing the object within a given scene (world), you have to position it in the right place and maybe re-dimension it to fit inside the scene.

The series of transformations (translation, rotation, scaling) needed to position the model within a scene are called modeling transformations. They transform the vertices of the object from Local Space to World Space.

Translation

The translation transformation is used to move an object (point) by a given amount.

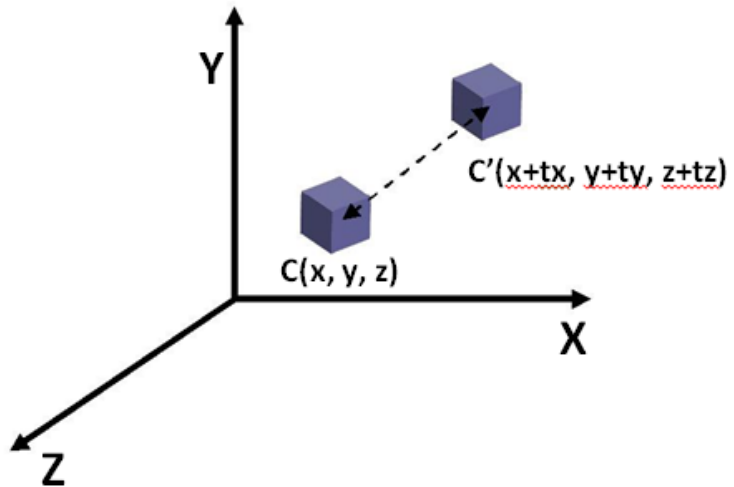
The translation matrix is:

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

The matrix for the inverse translation is:

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

T_x , T_y and T_z represent the translation factors on x, y and z axes.



You can generate a translation matrix using:

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

Scale

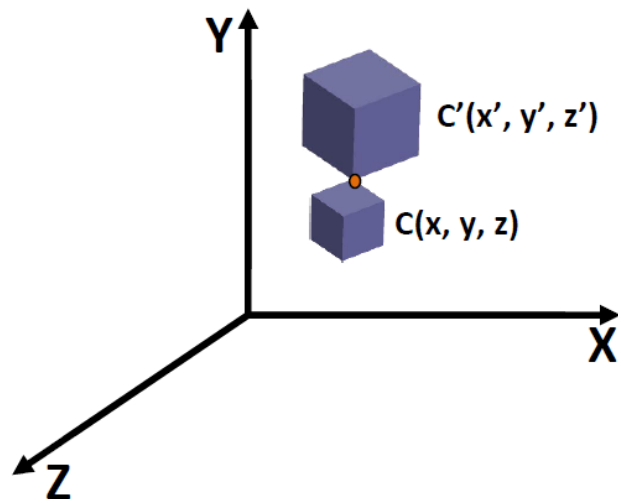
The scale transformation enlarges or reduces an object. The transformation is *relative to the origin*.

The matrix for the scale operation is the following:

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

The matrix for the inverse transformation is the following:

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



S_x , S_y and S_z represent the scale factors on x, y and z axes. If the S_x , S_y and S_z factors are equal, then the scaling transformation is uniform. If the S_x , S_y and S_z factors are not equal, then the scaling transformation is non-uniform.

If you set the scaling factors to ± 1 , then you can reflect the original shape.

You can generate a scaling matrix using:

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

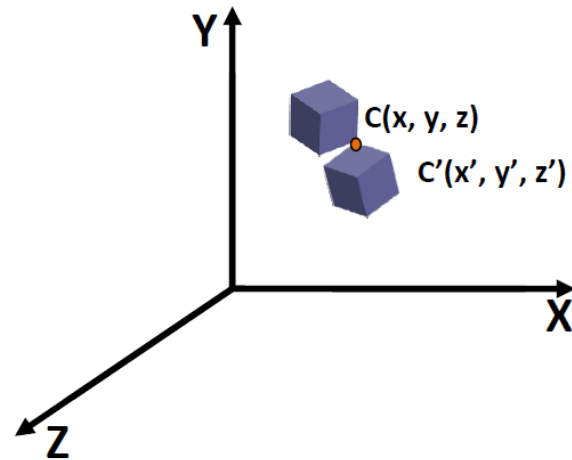
Rotation

This transformation rotates an object with a given angle. This transformation is also relative to the origin.

For a 2D point, the rotation and inverse rotation matrices are the following:

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

In 3D space, we specify rotation independently on the x, y, and z axes. Rotation around the z axis is similar to the rotation in 2D (the z coordinate remains unchanged).



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

To change the rotation center – i.e. rotate around a different point instead of the origin, one must do a series of three transformations:

- Translate the new center point to the origin
- Perform the rotation
- Do the inverse translation transformation

Mathematically, this would mean: $\mathbf{P}_{\text{new}} = \mathbf{T}_{\text{inverse}} * \mathbf{R} * \mathbf{T} * \mathbf{P}$

You can generate a rotation matrix using:

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

- x, y and z represent the 3D vector (axis) around which the rotation is performed
- the rotation angle is specified in radians, not degrees.

Camera transformations

After the scene has been set up, you need to define a viewer's position (camera) and viewing direction. This is done in order to render the scene from that viewer's perspective. Whereas until now the vertex coordinates were relative to the scene's origin, now they will be expressed relative to the camera's position. We need to perform a series of rotations and translations to bring the scene's origin to the camera's position. This is called transforming from World Space to View Space.

The series of transformations needed in order to convert the scene to View Space are contained within one matrix, called camera or view matrix. It is built by taking into account the camera position, its orientation and a user specified “UP” vector.

The view matrix can be built using the following GLM function:

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

where:

- camPosition – a vec3 specifying the 3D coordinates of the camera position;
- viewPoint – a vec3 specifying the 3D coordinates of the point at which the camera is pointed. Note that the vector defined as viewPoint - camPosition specifies the viewing direction;
- UP – the “UP” vector – specifies the direction of the Y axis and is usually equal to (0,1,0);

Projection transformations

These transformations are required in order to create a 2D mapping from a portion of the 3D scene of objects defined within your project. The “portion” of the scene contributing to the obtained 2D representation is delimited by a viewing volume. This, in turn, is dependent upon parameters such as the viewer’s position, field of view and the direction of the viewing vector. In the simplest of terms, you may think of a projection as the shadow that an object casts upon a 2D plane or surface.

While there are a number of different projection types, the most common are the perspective and orthographic projections.

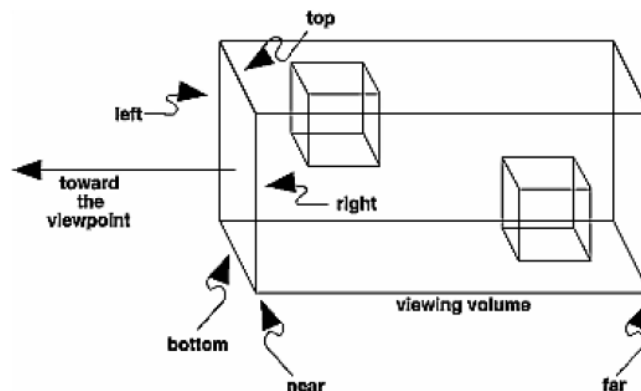
Orthographic projection

An orthographic projection transformation defines the viewing volume as a cube-like shape, using four parameters: width, height, a near and a far plane. Each 3D vertex will get mapped to a position determined by their X and Y coordinates.

You can create an orthographic projection matrix by using the GLM function `glm::ortho`, seen below:

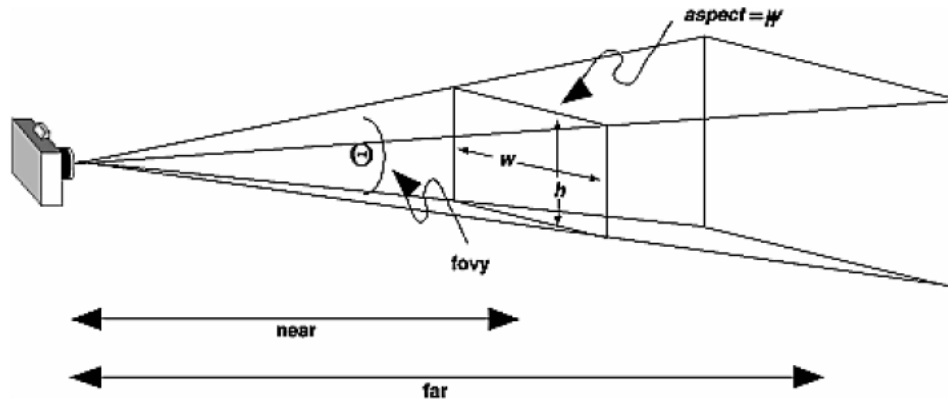
```
glm::ortho(GLint left, GLint right, GLint bottom, GLint top, GLfloat near, GLfloat far)
```

Each parameter of this function is a simple variable defining the X, Y or Z limits of the viewing volume.

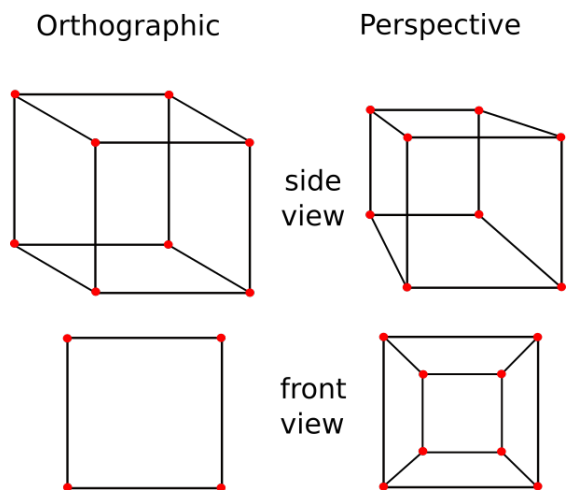


Perspective projection

The perspective transformation defines the viewing volume as frustum, starting from the camera position and expanding outwards, along the Z axis.



In this case, all three of the vertex's coordinates affect the 2D coordinates of its projection. The net result is that the further away an object is from the viewer, the smaller it will appear. This will give the viewer a sense of distance with regards to the objects in the scene.



You can create a perspective transformation matrix using the following GLM function:

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

Where:

- fovy – the angle that defines how large the viewing frustum is (usually 45 degrees)
- aspect – defines the aspect ratio, and is equal to the viewport's width divided by its height
- near & far – the Z coordinates of the near and far planes, which are the ends of the view frustum

Tutorial

We need to create the transformation matrices for each of the three steps:

- Model transformation
- View/camera transformation
- Projection transformation

After defining the matrices, we will use them to transform each and every vertex of the scene. To do this, we will apply the three transformations inside the Vertex Shader. After applying the transformations, the resulting vertex will look like this:

$$P_{\text{final}} = M_{\text{projection}} * M_{\text{view}} * M_{\text{model}} * P_{\text{initial}}$$

Keep in mind that the order in which the transformations are applied to the vertex is the opposite of the order in which they appear in the formula above. To get the correct order in which they are applied to the vertex, you should read the transformations for right to left.

Modeling transformations

Start by downloading the application from the laboratory website. Build and run the application.

You should see a purple square. This is in fact a multi-colored cube, seen from the front. In order to perceive it as a cube we will first add a rotation transformation to it.

As mentioned earlier, modeling transformations should be applied on the vertices inside the **Vertex Shader**. To this end, we will send it the modeling transformation matrix as a uniform variable:

```
#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);
}
```

We need to send the model matrix from the main program to the vertex shader. Declare the model matrix and its location as global variables:

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

gps::Shader myCustomShader;
glm::mat4 model;
GLint modelLoc;
```

Create and send the model matrix:

```

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(glfwWindow)) {
        renderScene();

        glfwPollEvents();
        glfwSwapBuffers(glfwWindow);
    }
    //close GL context and any other GLFW resources
    glfwTerminate();

    return 0;
}

```

Let us add a rotation transform inside “renderScene()”. This will activate on pressing the “R” key:

```

float angle=0;
void renderScene()
{
    . . . . .

    model = glm::mat4(1.0f);

    if (glfwGetKey(glfwWindow, 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);
}

```

Try out the rotation by pressing the “R” key.

Camera and projection transformations

The next step is to apply the view and projection transformations on our scene of objects. This will allow for a more realistic and controllable representation of the scene.

Start by adding two more matrices inside the 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);
}
```

We now have to define, initialize and send the two matrices to the vertex shader. Add the following:

```
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));

    . . . .

}
```

Adding transformations

Let us draw a second cube and place it 2 units to the left of the first one. By default, an object will always be drawn centered at the origin. To avoid this, we perform a translation before drawing the second cube:

```
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);
}
```

The cube will still be drawn at the origin, but the translation transformation has just moved that origin two units to the right.

Do a rotation by holding the “R” key. Notice how the transformation applies to the whole scene.

Now, let us apply a rotation on the second cube:

```
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);
}

```

Assignment

1. Control the rotation of the second cube with the “T” key.
2. Modify the rotation of the second cube by making it spin around its center – the point (2, 0, 0).
* Tip: Use the sequence of operations $T_{\text{inverse}} * R * T$
3. Implement the Camera class (start from the Camera.hpp present on the web site). Control the camera using keyboard and mouse inputs. **Hint:** Register some callback functions to process keyboard and mouse inputs:

```

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);

```

Laboratory work 6 – Loading 3D objects

Objectives

This laboratory presents you with the basic notions on the concept of texture and its usage within a 3D scene of objects. It will also provide you a basic framework for loading and including 3D objects into your OpenGL project and then briefly touch upon the subject of animation.

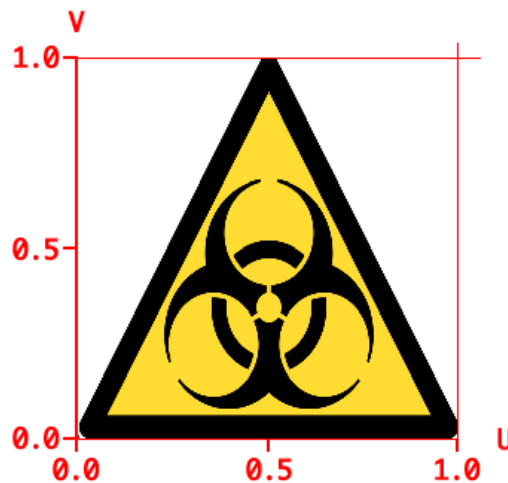
Theoretical background

You may think of a texture as a 2D image that can be applied on a 3D object. While it is also possible to have 1D and 3D textures, these are beyond the scope of this laboratory.

Generally, textures are applied on 3D objects in order to increase the photorealism, by adding detail onto their surfaces. One could conceivably add color detail to a model by applying different colors on a vertex-by-vertex basis, but this would be very wasteful, both in terms of the data volumes and processing power required.

Textures are basically 2D pixel maps loaded within the video memory, which can then be mapped to a given 3D object. The mapping is done by assigning each 3D vertex a point within the 2D image. The 2D texture space is usually expressed using the “u” and “v” parameter names, as opposed to “x” and “y” in order to avoid confusion with vertex position values.

As opposed to normal images, whose coordinates are integers expressing the value of a given pixel, a texture’s UV coordinates are floating point values ranged within the interval $[0, 1]$, where $(0, 0)$ represents the bottom-left corner of the image and $(1, 1)$ the top-right. These coordinates can be turned into pixels by multiplying them with the image’s width and height.



In order to use a texture, one must first load it from an external file:

- Load the image data from an image file (preferably “.png” or “.tex” – always with width and height equal to a power of 2 – ex: 128, 256, 512 etc.)
- Generate a texture ID - using the function: “glGenTextures”
- Bind, or “activate” the texture – using the function “glBindTexture”
- Generate a texture from the image data – using the function “glTexImage2D”
- Set up texture parameters

The texture loading functionality is implemented within the **“ReadTextureFromFile”**. Take a look at it.

In order to apply the texture on an object, it suffices to bind it before drawing the object - using the function “glBindTexture”.

Tutorial

Texture application

For the purpose of demonstrating the use of textures, we will first try out a simple example.

Let’s start by drawing two basic triangles. Define the vertex data – vertex position and texture UV coordinates:

```
//vertex position and UV coordinates
GLfloat vertexData[] = {
    // first triangle
    -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,
    // second triangle
    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
};
```

Load the triangle data by calling the predefined function “loadTriangles()” - before the start of the main loop:

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

    .....

    loadTriangleData();

    while (!glfwWindowShouldClose(glfwWindow)) {
        renderScene();
    }
}
```

```

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

```

Finally, draw the triangles:

```

void renderScene()
{
    .....

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

```

Build and run the application. You should see two black triangles.

In order to apply the texture, we need to send the texture coordinates to the Fragment Shader. This means that we have to pass them along from within the Vertex Shader:

```

#version 410 core

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);
}

```

Inside the Fragment Shader, we need access to a texture unit, called “diffuseTexture”. The function “texture” will return a value from the currently active texture, based on the 2D UV coordinates it receives as argument.

Fragment Shader should look like this:

```

#version 410 core

in vec3 colour;

```

```

in vec2 passTexture;

out vec4 fragmentColour;

uniform sampler2D diffuseTexture;

void main() {
    fragmentColour = texture(diffuseTexture, passTexture);
}

```

Make sure you load the texture from the image file:

```

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

    loadTriangleData();
    texture = ReadTextureFromFile("textures/hazard2.png");

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

        glfwPollEvents();
        glfwSwapBuffers(glfwWindow);
    }
    .....
}

```

Now activate the texture before drawing the triangles:

```

void renderScene()
{
    .....

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

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

```

Build and run the application.

Modify the texture UV coordinates for the second triangle, so that your application will look like this:



Loading 3D objects

For the purpose of loading and drawing complex 3D objects, one option you have is to use the Model3D class provided within this laboratory resource bundle. This class should be able to load anything, from simple objects to complex scenes of multiple, textured objects. As you can see inside “Model3D.hpp”, it holds a list of meshes, where each mesh represents a 3D object, as well as the collection of loaded textures necessary for those objects.

Let’s see how we can use the Model3D class in order to load a model from an “.obj” file. Modify the main function. Delete the previously added functionality and add:

```
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(glfwWindow)) {  
        renderScene();  
  
        glfwPollEvents();  
        glfwSwapBuffers(glfwWindow);  
    }  
    .....  
}
```

To render the loaded model, modify and add the following lines within the “renderScene” function:

```
void renderScene()  
{  
  
    .....  
    //create rotation matrix  
    model = glm::rotate(model, glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));  
    model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));  
}
```

```

//send matrix data to vertex shader
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
myModel.Draw(myCustomShader);
}

```

Build and run the application. As you can see, **the model is not textured**. To apply a texture to it, you must load one from an external file:

```

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

```

Activate the texture before drawing the model:

```

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

    myModel.Draw(myCustomShader);
}

```

Run the application. Use the “W”, “A”, “S” and “D” keys to navigate; “Q” and “E” to rotate the model.

Remember when we said that a Model3D object can hold both the 3D meshes and their textures? Most often, when you will be downloading 3D objects from online sources, you will be provided both with an “.obj” file and an “.mtl” one, together with a collection of applied texture images. The “.mtl” file specifies the texture association data for the object. **In these cases, the Model3D object will read the object and activate its associated textures as required – there will be no need for you to load and activate the textures yourselves.**

You may find “.obj” files at www.turbosquid.com, www.tf3dm.com and

<https://sketchfab.com/features/free-3d-models>.

Let’s try to add an “.obj” file with included textures. Modify the main function to load the new object:

```

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(glfwWindow)) {
        renderScene();
    }
}

```

```

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

```

Comment the texture activation instruction from renderScene – this object has texture mapping information included, so the Model3D object will try to load the textures for you automatically:

```

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

    myModel.Draw(myCustomShader);
}

```

General rules to follow when loading .obj files from the internet:

- Look for the .mtl file – it contains data about the object’s materials and textures. Inside the .obj you have a reference to the .mtl file. **Make sure that its path is correct** – it should be relative to your project’s location.
- Within the .mtl file, **look for texture files references** – make sure the texture paths are set up correctly – path relative to the project location. If they cannot be found, they will not be loaded.
- Make sure the textures are in “.png” or “.tga” format and have dimensions (width and height) equal to powers of 2 (ex: 128, 256, 512, etc.).
- Try to keep your project clean and ordered - keep the 3D objects and textures within their own separate folders.

Animation

Let us add a translation transformation to our object – translate on Ox axis by an amount equal to “delta”. Increase delta inside the “renderScene” function.

```

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

    //create rotation matrix
    model = glm::rotate(model, glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
    //send matrix data to vertex shader
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
}

```

```

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

```

Run the application. Then try drawing 500 instances of the same object. Notice the decrease in animation speed:

```

void renderScene()
{
    .....

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

```

This animation is dependent upon the frequency with which the function is invoked – that is the refresh frequency of your application. As you might guess, the refresh frequency varies depending upon your hardware resources or scene complexity. Ideally, we would want the animation to run independently of the FPS capabilities of our application.

The solution is to increment delta based on a given speed variable – “units per second”. We measure the time between successive updates and compute the distance the object should move:

$$Distance = speed * time$$

```

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));  
.....  
}
```

You can try the animation with increasing numbers of object instances, to see whether the movement speed holds.

Further reading

Function (and link)	Description
glGenTextures	Generate texture names
glBindTexture	Bind a named texture to a texturing target
glTexImage2D	Specify a two-dimensional texture image
glTexParameter	Set texture parameters

Assignment

- Load at least 3 models. Change the position of one object using keyboard controls and animate another object.

Laboratory work 7 – Lighting models

Objectives

The objective of this laboratory is to briefly describe how to add lighting to your OpenGL 4 scene. We will cover the basic mathematics behind OpenGL lighting and the structure of the vertex and fragment shader for per-vertex and per-fragment directional lighting.

Theoretical background

OpenGL fixed-pipeline lighting (the Gouraud model) represents the basic lighting model for graphics applications and is a decent attempt at modeling how real-world illumination works. However, the fixed functionality is constraining, lacks in realism and in performance-quality tradeoffs. Programmable shaders can provide far superior results, especially in realism. Nevertheless, it is quite important to thoroughly understand the basic OpenGL lighting model, since this lighting model still provides the fundamentals on which advanced models are based.

OpenGL fixed-pipeline lighting model sums-up a set of independently computed components to obtain the overall illumination effect of a spot on an object. These components can be visually observed in Figure 1 below.

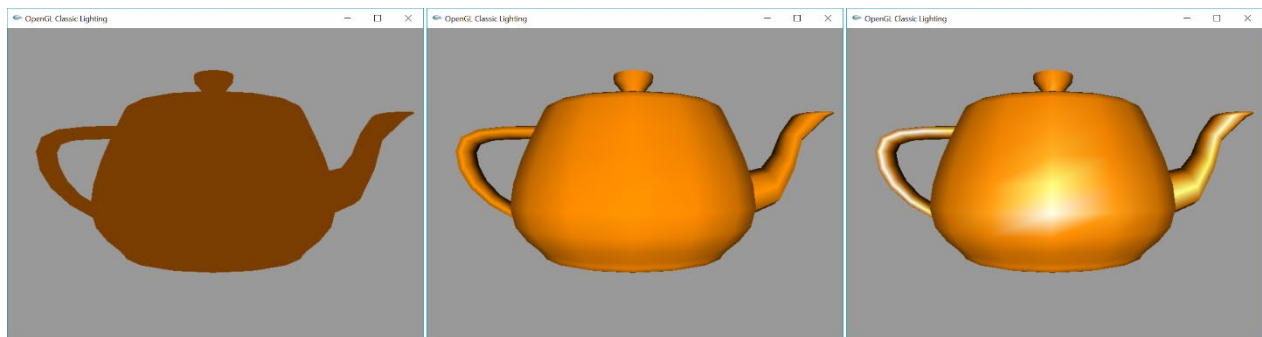


Figure 3 - OpenGL light components (from left to right: Ambient, Ambient + Diffuse, Ambient + Diffuse + Specular)

Ambient light does not come from any specific direction and is a decent approximation for light that exists scattered around in a scene. Its computation does not depend on the viewer's position or on the direction of the light and it can be either precomputed as a global effect or added on a per-light basis.

Diffuse light is scattered equally in all directions for a light source. The amount of diffuse light existent on a spot does not depend on the viewer's position, but it does depend on the direction of the light. It is brighter on the surfaces that are oriented towards the light and dimmer on the ones oriented away from it. Its computation depends on the surface **normal** and on the direction of the light source, but not on the direction of the eye.

Specular highlighting is light directly reflected by the surface and it refers to how much the material behaves similarly to a mirror. The strength of this effect is referred to as **Shininess**. Its computation requires knowing how close the surface's orientation is to the direction of direct reflection between the

light direction and the eye, hence it depends on the surface **normal**, on the direction of the light and on the viewing direction.

OpenGL 4 lighting implementation

We will now incrementally build the shaders required for implementing the OpenGL classic lighting model (Gouraud lighting) using the GLSL shading language. Since the classical model is implemented on a per-vertex basis, the shader that we will incrementally change is the vertex shader. The fragment shader will remain the same throughout this process and it will be a simple pass through shader as presented below.

```
//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);
}
```

In theory, lighting computation can be carried out in any space (coordinate system) along the transformations pipeline. However, for correct (and thus more realistic) results, lighting should be performed in an affine space, namely in world or eye space. Both will offer identical lighting results. For this laboratory, we will be carrying out lighting computation in eye space.

No light

When no lighting is used, OpenGL renders each object uniformly using its base color. All lighting effects will modulate this base color as a basis of following lighting calculations. The vertex shader for drawing an object without lighting effects is presented below. For this laboratory, the base color is passed from the application, as a uniform value. In real-world applications, the base color is usually passed as a vertex attribute.

```
//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);
}

```

Ambient light only

Ambient light is uniform and thus doesn't change across vertices. We can pass the Ambient component from the application as a uniform variable. The interaction result between the ambient light and the color of the surface it hits is modeled through multiplication. During the various multiplications required to compute the effects of various light components on a surface, it is ok for components' values to go above 1.0. However, the final color needs to be saturated at white (i.e. no light component must be above 1.0), thus the usage of the **min** function to achieve this saturation. The additions and alterations to the vertex shader for incorporating ambient light are highlighted below.

```

//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);
}

```

Ambient and diffuse lighting

Diffuse lighting depends on the direction of the light that hits the spot we are currently computing. Since our light is a **directional** one (located at infinity, as opposed to point light sources which have a defined 3D position), all light rays are parallel and have the same direction. This means that the direction of the light is constant across all vertices and can be computed in the application and passed as a **uniform** variable to the shader. The amount of scattered diffuse light depends on the surface's orientation (i.e. the angle between the direction of the light and the surface's normal). The maximum amount of scattered light is obtained when the light hits the surface perpendicularly (i.e. the angle between the normal and the direction of the light is 0.0, thus the cosine is 1.0). As this angle increases (the surface turns away from the light), the cosine decreases towards 0.0, thus the amount of scattered diffuse light is lower. The cosine of the angle between two normalized vectors can be computed as their dot product.

The vertex normal are passed in from the application (they are pre-computed in the GLM model) as a per-vertex attribute (similarly to vertex positions). As the model is affected by transformations (each vertex is multiplied with the Model, View and Projection matrices) the normal must be subjected to the same transformations as the object, otherwise the illumination will not work properly (i.e. the normal must move together with the object). In OpenGL, normals are not transformed using the Model-View-Projection matrix, but using the so called Normal Matrix. The Normal Matrix is the upper left 3x3 matrix of the transposed inverse of the Model matrix (if carrying out lighting computation in world space) or Model-View matrix (if carrying out lighting computation in eye space) and can be easily computed in the application using the OpenGL mathematics (GLM) library and passed to the shader as a **uniform** variable. The additions to the vertex shader to incorporate diffuse light are highlighted below.

```

//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);
}

```

Ambient, diffuse and specular lighting

Specular highlights depend both on the direction of the light and the direction of viewing. The specular component depends on the cosine of the angle (α) between the viewing direction (Eye) and the reflection coming from the light (R) (Figure 2). Since we carry out lighting computation in eye coordinates, the camera is situated at the origin, thus the Eye position is (0, 0, 0).

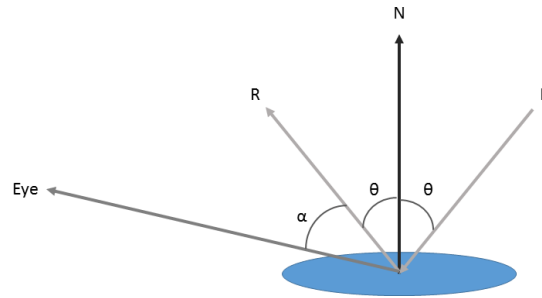


Figure 4 – Specular highlight computation

Shininess is used as an exponent to obtain a measure of the sharpness of the angular fall from a direct reflection. The greater the exponent, the more concentrated the specular highlight, since raising a number smaller than 1.0 to an exponential power decreases its value. Reflection angles close to 0.0 will result in values close to 1.0 for the specular highlight, while for other angles it quickly decays to 0.0. Because specular highlights are reflected light, the specular component is not modulated with the object's color. The additions to the vertex shader to incorporate specular highlights are highlighted below.

```
//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);
}

```

Tutorial

Download the resources available on the laboratory web page. The first archive (Laboratory 8 resources - part 1.zip) contains an application with the vertex and fragment shaders required for rendering a teapot without lighting (see subsection 3.1)

Extend the vertex shader as to incorporate ambient lighting into your application (see section 3.2). Your application should render the teapot as in Figure 3 below.



Figure 5 – Rendering with ambient lighting

Extend the vertex shader as to incorporate diffuse lighting into your application (see section 3.3). Your application should render the teapot as in Figure 4 below.



Figure 6 – Rendering with ambient and diffuse lighting

Extend the vertex shader as to incorporate specular lighting into your application (see section 3.4). Your application should render the teapot as in Figure 5 below.



Figure 7 – Rendering with ambient, diffuse and specular lighting

Further reading

- OpenGL Programming Guide 8th Edition – Chapter 7

Assignment

1. Implement OpenGL per-vertex lighting (Gouraud) for your application following the steps described in section 4
2. Implement per-pixel lighting (Phong). Per-pixel lighting computes the final colors in the fragment shader, using the **interpolated normals** instead of using the vertex normals in the vertex shader. Compare the overall lighting quality with the per-vertex model obtained at step 1, for all teapot resolutions (there are several teapots available, generated using 4, 10, 20 and 50 segments). For this task, use the resources provided in the second archive (Laboratory 8 resources - part 2.zip) and modify the set of shaders shaderPPL.(vert/frag).
3. Extend the per-pixel lighting implementation to incorporate material properties for objects. Material properties can be passed from the application as uniform variables and the final color will be modulated (multiplied) with the corresponding material component (Ambient, Diffuse and Specular) instead of the object's base color.
4. Extend the per-pixel lighting implementation to incorporate light component intensities. Instead of having a single light color and using hard-coded coefficients to decide the contribution of each lighting component, describe a light source using three different components (ambient, diffuse and specular) and use them to compute the final color intensities.

Laboratory work 8 – Advanced lighting

Objectives

The objective of this laboratory is to briefly describe several advanced lighting techniques in OpenGL 4.

The Blinn-Phong Lighting Model

The Blinn-Phong lighting model is a slightly modified version of the Phong model and was developed by Jim Blinn. Its aim is to improve the specular reflections in certain conditions like a very small shininess coefficient (thus resulting a large specular area). Specular reflections in the Phong model tend to be cut-off immediately when the angle between the viewing direction and the light reflection increases above 90.0 degrees (the cosine becomes negative and thus is clamped to 0.0).

The solution proposed by Blinn is to use a so called **half vector** instead of the reflection vector to compute the specular component. The half vector (**H**) is a unit vector halfway between the viewing direction and the light direction (Figure 1).

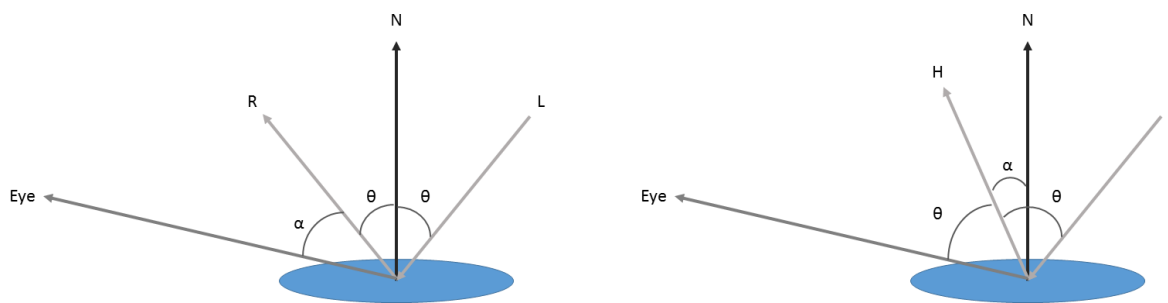


Figure 8 - Phong specular model (left) versus Blinn specular model (right)

The half vector is computed as:

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

This translates into GLSL code as follows:

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

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

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

The specular component is then computed as the cosine of the angle between the normal and the half vector:

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

The Blinn-Phong model also improves performance by avoiding the expensive computation of the reflection vector.

Point lights

Laboratory work 8 introduced **directional lights**, which are light sources infinitely far away. All light rays coming from a directional light source are parallel and thus the direction of the light involved in lighting computation is constant across objects and all their fragments, making them ideal for global illumination in our scenes.

Point lights are light sources with a given position that illuminate radially and uniformly in all directions. Unlike directional lights, rays generated by point lights fade out over distance, thus making objects closer to the source appear brighter than objects situated farther away.

When using point lights, the direction of the light is not constant across fragments and must be computed for each different fragment:

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

To reduce the intensity of the point light over the distance we should apply an attenuation coefficient. Attenuation can be computed as a linear or quadratic function over the distance. Linear attenuation requires less complex computation but offers less realism since, in the real world, light sources are bright when standing close to them but quickly diminish in brightness the farther we stand from them. Realism is considerably increased when incorporating quadratic attenuation over the distance. Thus, attenuation can be computed using:

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

where K_c is a constant term, usually kept at 1.0, K_l is a linear term, K_q is a quadratic term and d is the distance.

Choosing the right values for these constants depends on many factors such as the environment, the distance required for the light to travel, the type of light etc. You can use Table 1 as a guide for specifying the attenuation coefficients based on the distance the light is required to travel.

To implement attenuation, we should define the attenuation coefficients in the shader or send them from the application via uniform variables:

```
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

Table 1 – Common values for attenuation coefficients (source: [Ogre3D wiki](#))

We should then compute the attenuation using the equation provided above:

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

Last, we should apply the attenuation to the ambient, diffuse and specular components:

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

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

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

Lighting Maps

Up until now, when it comes to lighting, we have considered objects as being composed of a single material (having a single base color) (Figure 2). However, this approach does not offer too much flexibility on the visual output of an object. We will now extend the lighting model to incorporate **lighting maps**.



Figure 9 – Crysis 2 nanosuit rendered with point light and orange base color

Lighting maps provide us with means to sample the base color for each fragment from a texture for each lighting component (ambient, diffuse and specular). They are incorporated into the material properties of OBJ models (the MTL file).

Diffuse maps define the colors of an object. Color information is basically sampled from a texture and combined with the effects of lighting to determine the final fragment color (Figure 3). Diffuse maps usually also serve as the base color for the ambient component. Thus, the computation of the ambient and diffuse components should change accordingly:

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



Figure 10 – Crysis 2 nanosuit rendered with diffuse maps

Usually not all the materials an object is composed of reflect the same amount of specular highlights. To differentiate between the reflectivity of different materials we can use **specular maps** that assign a higher specular coefficient to highly reflective materials and a lower one to materials that do not normally present

specular components (Figure 4). Specular maps can be sampled exactly as diffuse maps but are instead applied to the computation of the specular components:

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

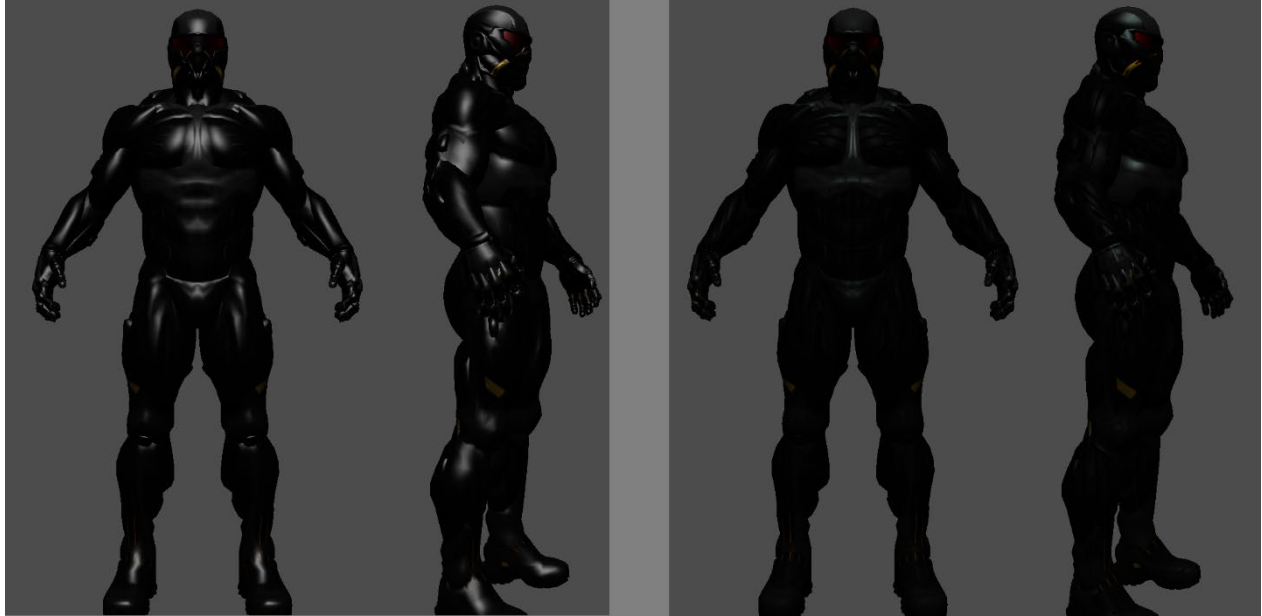


Figure 11 – Crysis 2 nanosuit rendered without specular maps (left) and with specular maps (right)

The OBJ loading library provided already includes loading diffuse and specular maps (textures) and proper binding of these texture objects inside the Model3D and Mesh classes. The uniforms “diffuseTexture” and “specularTexture” must be defined in the shader program to allow sampling of the lighting maps of an object.

Further reading

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

Assignment

5. Download and explore the code samples provided as resources for this laboratory
6. Implement the Blinn-Phong lighting model for the given demo application
7. Transform the directional light into a point light for the demo application
8. Render several objects scattered throughout the scene to observe the effect of point lights
9. Incorporate lighting maps into the demo application
10. Define a directional light and a second point light in the demo application

Laboratory work 9 – Shadow Mapping

Objectives

The objective of this laboratory is to briefly describe how to add basic shadows to your OpenGL 4 application using the **Shadow Mapping** technique.

Shadow Mapping

The lighting techniques discussed in previous laboratories work great for illuminating objects. However, the lighting effect obtained takes no account of occlusions. In the real world, when an object is not directly hit by rays emitted by a light source due to obstructions (other objects), then that object is in shadow. Thus, shadows add a very important degree of realism to our OpenGL scenes, also allowing for better depth perception (Figure 1 and Figure 2).

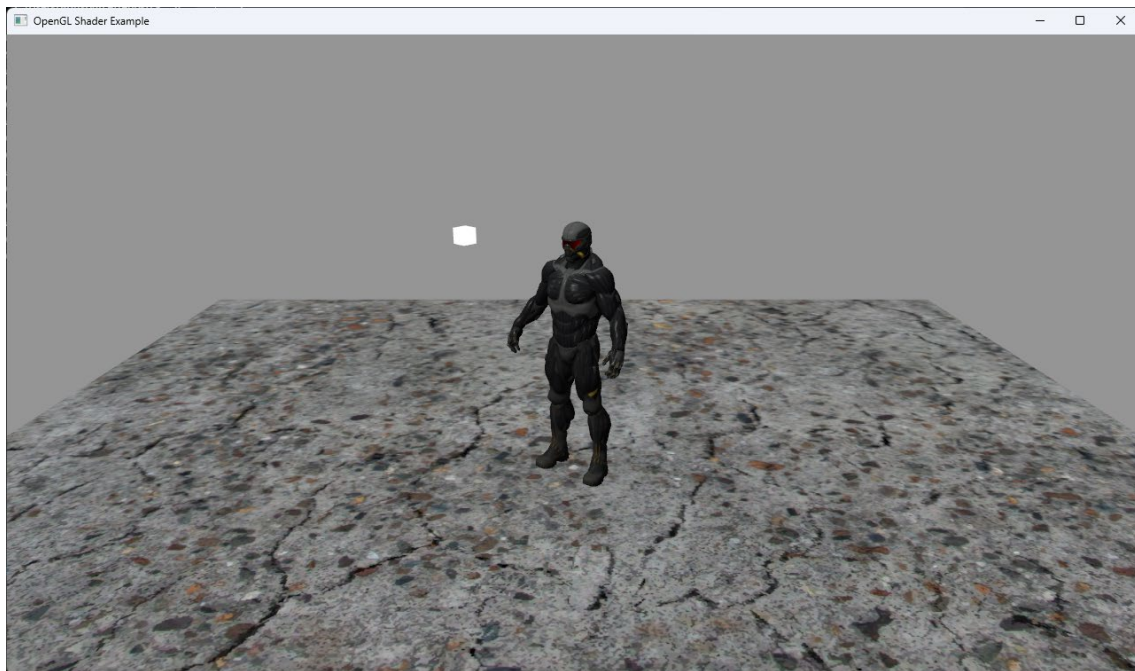


Figure 12 – Rendering without shadows

There is a plethora of modern techniques that can be employed to produce shadows (shadow computation) and all of them can be implemented in OpenGL. However, there are no perfect real-time algorithms for this, each having each its own, advantages and disadvantages. For this laboratory, we will cover a technique that offers decent results and is easy to implement, namely shadow mapping.

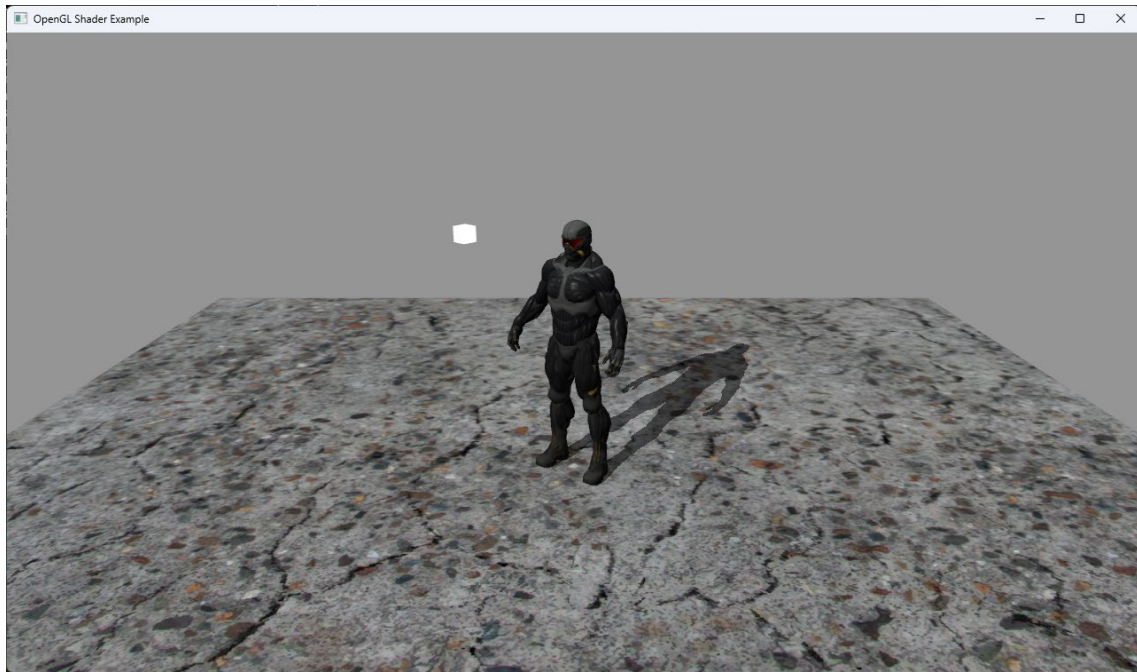


Figure 13 – Rendering with shadows

Shadow mapping is a multi-pass technique that uses depth textures to decide whether a point lies in shadow or not. The key is to observe the scene from the light source's point of view instead of the final viewing location. Any part of the scene that is not directly observable from the light's perspective will be in shadow.

The main passes of the algorithm are described below:

1. Render the scene from the light's point of view. It does not matter how the scene looks like (color information); the only relevant information at this point are the depth values. These values are stored in a *shadow map* (or *depth map*) and can be obtained by creating a depth texture, attaching it to a framebuffer object and rendering the entire scene (as viewed by the light) into this object. This way, the depth texture is directly filled with the relevant depth values.
2. Render the scene from the final point of view (the camera's position) and compare the depth of each visible fragment (projected into the light's reference frame) with depth values in the shadow map. Fragments that have a depth greater than what was previously stored in the depth map are not directly visible from the light's point of view and are, thus, in shadow.

Generating the depth map

The first pass of the shadow mapping algorithm generates a depth map of the entire scene viewed from the light's perspective. This map can be saved directly into a texture by attaching the texture to a framebuffer object and rendering directly into it.

OpenGL does all its rendering (color, depth and stencil information) into a framebuffer. The colors stored into the framebuffer are used by the screen when displaying the visual content our applications generate. Framebuffer objects allow us to create our own framebuffers and instead of rendering into special areas

of GPU memory, we can write information (color, depth and stencil information) into textures. These textures are attached as the color, depth and stencil buffers for our framebuffer objects.

Creating a framebuffer object

A framebuffer object is created as shown below:

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

To attach a texture as a depth buffer for our framebuffer object, we should first create the texture:

```
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);
```

The depth texture is created using `GL_DEPTH_COMPONENT` as the texture type. Also, to avoid unwanted visual artefacts when generating shadows, the texture wrap mode is set to `GL_CLAMP_TO_BORDER`. `SHADOW_WIDTH` and `SHADOW_HEIGHT` are constants and represent the depth map's resolution, which should be at least the size of the OpenGL window's framebuffer. Too low resolutions tend to lead to poorer results and too high resolutions tend to waste memory.

Once the depth texture is created, it should be attached as the framebuffer object's depth buffer:

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

Since the first pass of the shadow mapping algorithm does not require a color or stencil attachment but a framebuffer object would not be complete without them, we can explicitly bind nothing to these attachment points:

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

Once a framebuffer object is complete, we should unbind it until we are ready to use it:

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

With a properly configured framebuffer object, the complete rendering stage, using the depth map, is as follows:

```
// render to depth map
Change the viewport to accommodate the depth texture size;
Bind the framebuffer object;
    Clear the depth buffer;
    Configure shader (for filling the depth texture) and transformation matrices (for the light space);
    Render the scene;
Unbind the framebuffer object;
// render to the main framebuffer
Change the viewport to accommodate the OpenGL window's framebuffer;
    Clear the color and depth buffer;
    Configure shader (for rendering with shadows) and transformation matrices;
    Bind the depth texture;
    Render the scene;
```

Light space transforms

The first pass should render the scene from the light's point of view. This means that the coordinates of all objects in the scene must be transformed to be relative to the light instead of the camera. Thus, we can use the same function as for the Eye space transformation, **lookAt**, but with the light's position as the first parameter. For this laboratory, we shall use a directional light. Since directional lights have no position (they are placed at infinity), we can use as a position any point along the light's direction, including the direction itself (interpreted as a point):

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

Because all rays coming from a directional light are parallel, we shall use an orthographic projection to avoid any perspective deforming:

```
const GLfloat near_plane = 0.1f, far_plane = 6.0f;
glm::mat4 lightProjection = glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, near_plane, far_plane);
```

The final transformation matrix for the light's space is:

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

Rendering to the depth map

The vertex shader that transforms all vertices into the light's space is presented below:

```
#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);
}
```

In OpenGL 4, it is legal to have no fragment shader only when rasterization is turned off. Thus, a simple fragment shader that writes dummy output should be implemented¹:

```
#version 410 core

out vec4 fColor;

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

Rendering into the depth map now becomes:

```
//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);
```

If we were to visualize the depth map at this point, it would look like in Figure 3.

Rendering with shadows

Once the depth map has been created, it can be used to render the scene with shadows. When comparing the depth of the current fragment with the depth existent in the depth map, both must be in the light's reference system. Since the first one is not, it must be transformed using the same matrix as when generating the map. Transformations should be performed in the vertex shader:

```
...
out vec4 fragPosLightSpace;
...
uniform mat4 lightSpaceTrMatrix;
```

¹ We could also have an empty fragment shader since we only care about depth values, which are generated behind the scenes by OpenGL anyway.

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

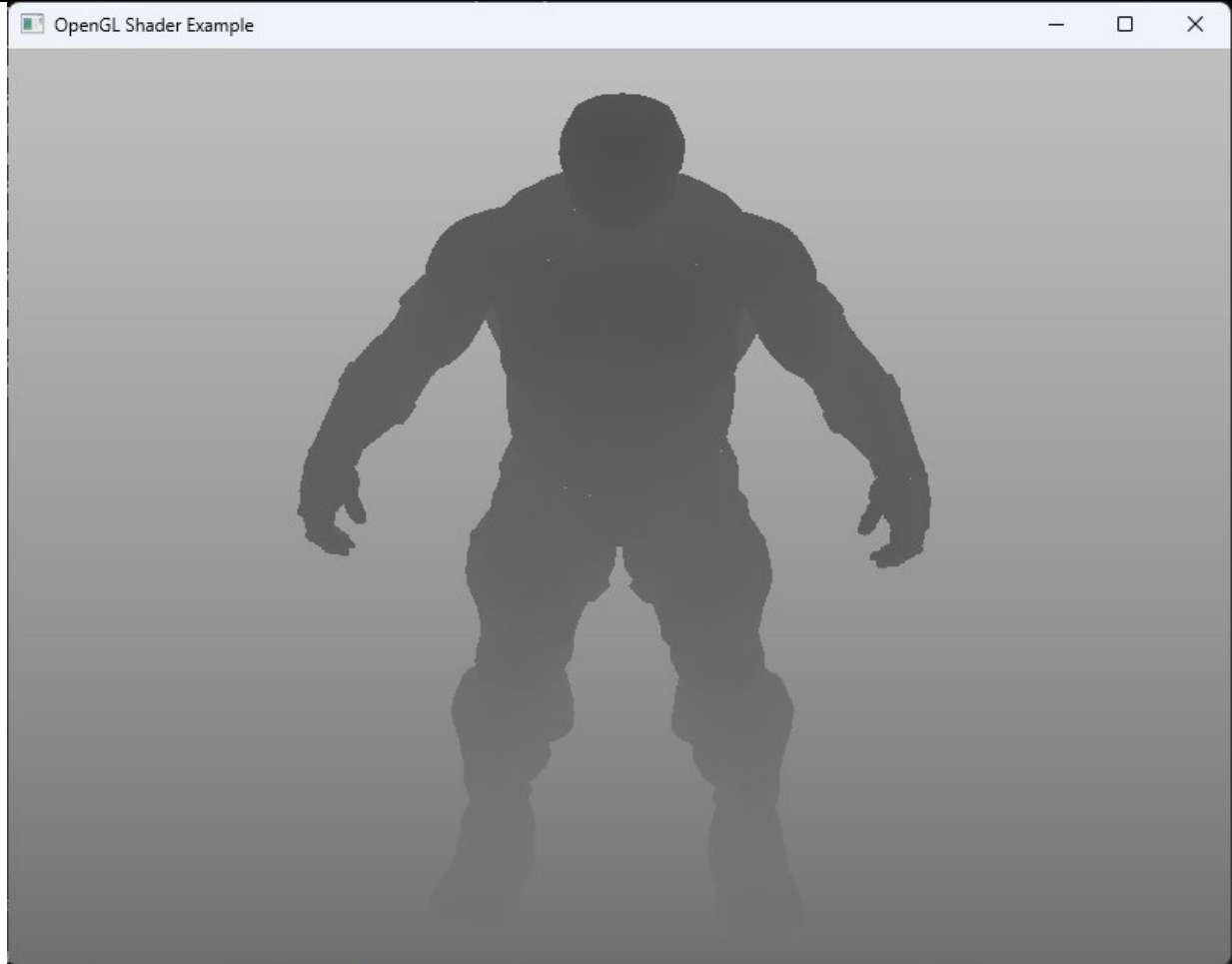


Figure 14 – Depth map example

The actual shadow computation will take place in the fragment shader. The final color of each fragment (for the diffuse and specular components) will be modulated with the computed shadow value (either 1.0 or 0.0)

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

To perform the actual computation of the shadow, the first thing we should do is to transform the fragment's position into the light's clip space normalized coordinates. Since the transformation to the light's clip space was done in the vertex shader, what remains is to apply perspective division to get the normalized coordinates:

```
in vec4 fragPosLightSpace;

...

float computeShadow()
{
    ...
    // perform perspective divide
    vec3 normalizedCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    ...
}
```

This returns the current fragment's position in the [-1,1] range. Since fragment depth values are in the [0,1] range, we should transform the normalized coordinates accordingly:

```
...

float computeShadow()
{
    ...
    // Transform to [0,1] range
    normalizedCoords = normalizedCoords * 0.5 + 0.5;
    ...
}
```

We can now sample the existing depth in the depth map using these coordinates since they now correspond with the NDC coordinates from the first pass:

```
...

uniform sampler2D shadowMap;

...

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

This gives us the closest depth value from the light's perspective. Next, we take the depth (in the light's reference system) of the current fragment:

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

We can now compare the two values. If the current fragment's depth is greater than the value in the depth map, the current fragment is in shadow. Else, it is illuminated:

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

When using this shader, with the properly bound depth texture, the output should be similar to the one in Figure 4.

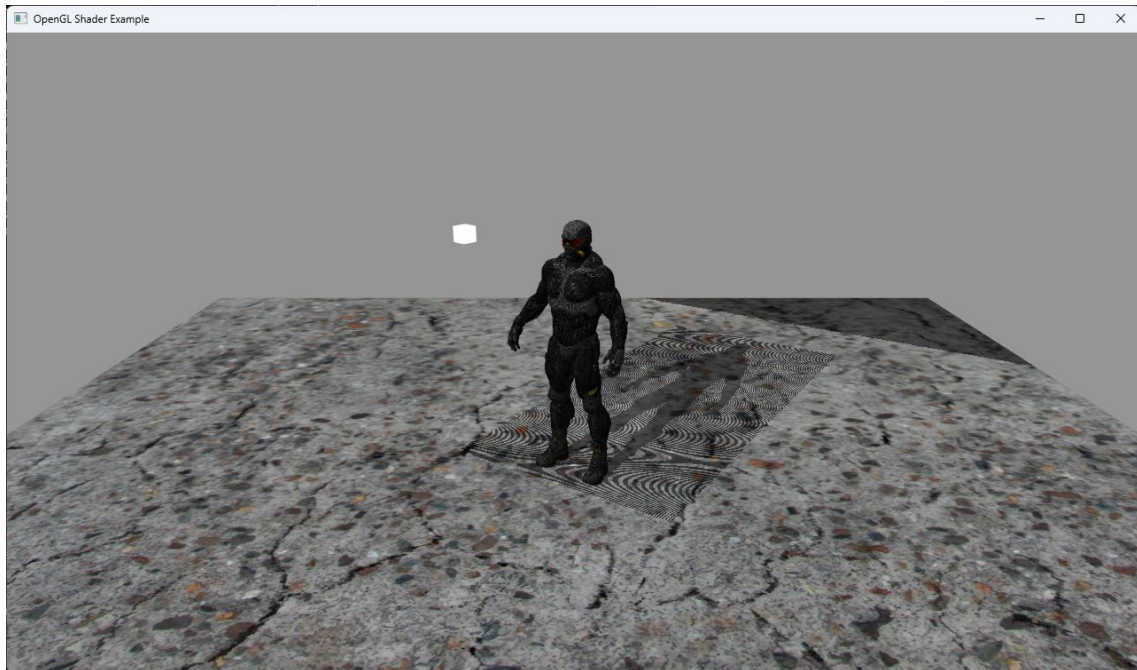


Figure 15 – Rendering with shadows

Improving quality

Shadow acne

On a closer look, we can spot an unwanted pattern as illustrated in Figure 5.

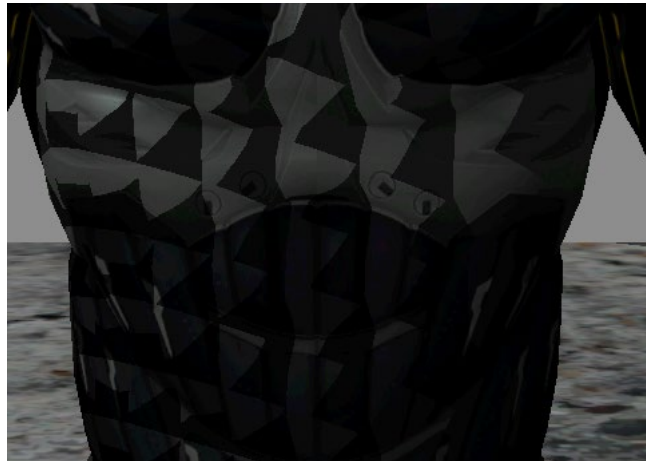


Figure 16 – Shadow acne

This happens due to the limited resolution of the depth map and it occurs when multiple fragments sample the same value from the depth map. We can usually solve this using a small bias to the fragment's depth,

either a fixed value or a value based on the angle the surface makes with the direction of the light (this solves issues for surfaces that have a steep angle to the light):

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

or

```
...
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;
    ...
}
```

The visual output should now become similar to the one in Figure 6.

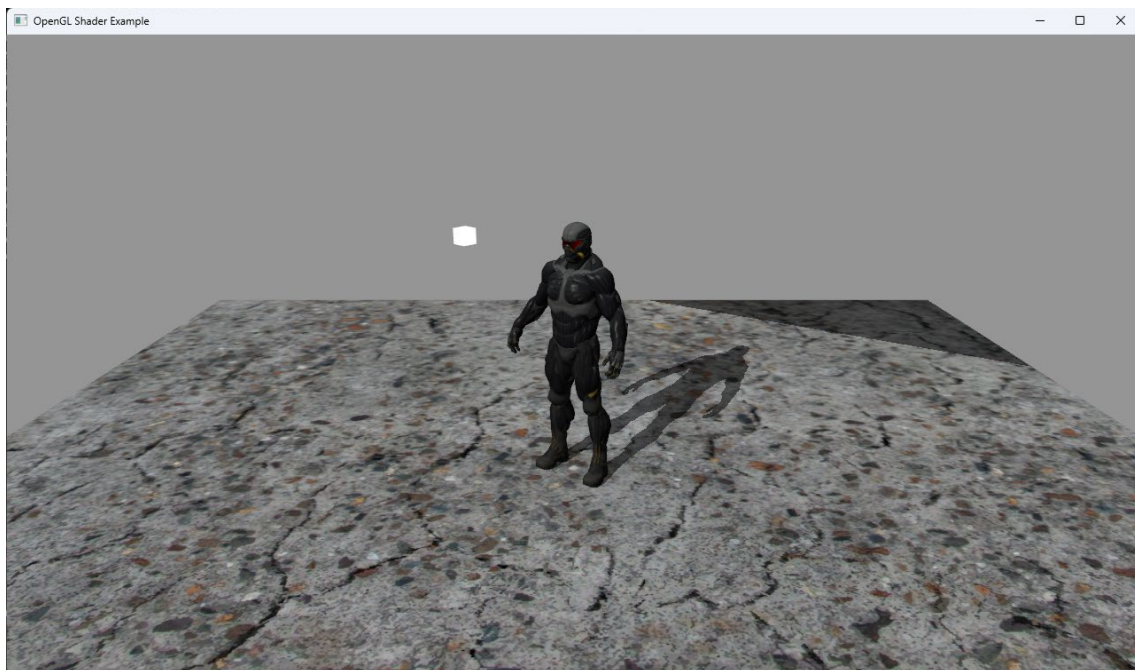


Figure 17 – No shadow acne

Over sampling

Notice that there are areas in shadow which actually should be illuminated (the far-right corner in Figure 6). This happens because fragments outside the depth map's frustum have a depth value greater than 1.0. This can be easily rectified in the fragment shader:

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

The visual output should now become similar to the one in Figure 7.

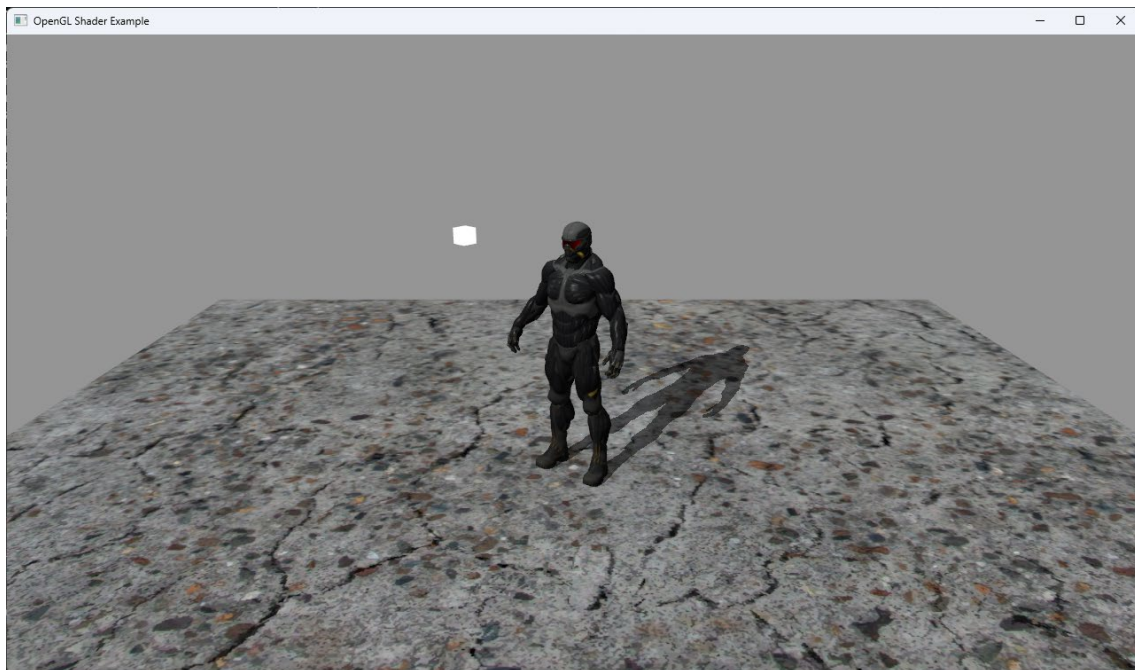


Figure 18 – No over sampling

Further reading

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

Assignment

11. Download the resources from Laboratory work 9
12. Add the ground object to your scene

13. Add a framebuffer object to the existing application
14. Add a depth texture to the framebuffer object
15. Add the necessary code and shaders to render the light's depth map into the depth texture
16. Add the necessary code to render the scene with shadows
17. Make the necessary adjustments to solve the various visual issues encountered (shadow acne, over sampling)

Laboratory work 10 – Cube maps

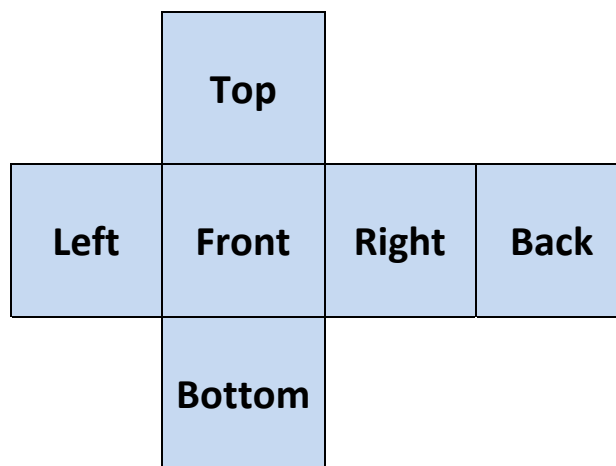
Objectives

This laboratory presents you with basic notions about the concept of cubemaps (that are used for defining skyboxes) and reflection properties of 3D objects.

Theoretical background

Cubemap

In many situations we want to display a “background” image representing the part of the 3D scene that we didn’t model. We can accomplish this by “including” our 3D scene of objects inside a cube and map different textures on each face. This cube is typically referred as a skybox and consists of 6 textures following this pattern:



The OpenGL representation of this kind of texture is a **cubemap**, containing 6 individual 2D textures. By combining these textures together and mapping them on a unit cube, we can very easily sample a texture value using a direction vector. This vector can be retrieved from the interpolated vertex position of the cube.

To create a cubemap we have to generate a texture id and bind it to the `GL_TEXTURE_CUBE_MAP` target.

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

For each face of the cubemap, we have to call the `glTexImage2D` function and set a proper target parameter specific to the current face:

Target	Orientation
--------	-------------

GL_TEXTURE_CUBE_MAP_POSITIVE_X	Right
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Left
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	Top
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	Bottom
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	Back
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	Front

We can linearly increment the target in order to set all the textures of the cubemap:

```
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);
}
```

In the fragment shader, we use a different sampler in this case (**samplerCube**) but we use the same function (texture) to sample the color value. Here, the textureCoordinates variable represents the direction vector.

```
#version 410 core

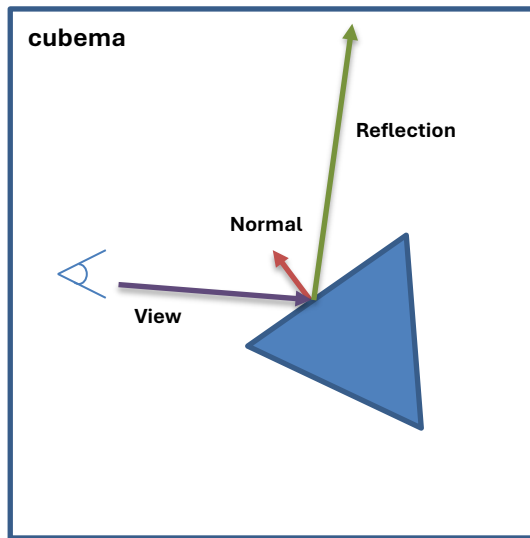
in vec3 textureCoordinates;
out vec4 color;

uniform samplerCube skybox;

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

Reflections

Having a surrounding environment (represented by the cubemap) we can now specify reflective properties to various objects. The reflection is based on viewer's angle. This reflection vector can be computed using the GLSL function called **reflect**. We have to specify the normal vector and the view direction vector. The resulting vector is used to sample from the cubemap texture.



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

Tutorial

Start by downloading the resources and include them in your project. The SkyBox class will be used to load and draw the cubemaps.

Add in a vector the textures (namely their path) that will be mapped on the cubemap. Create a void function called **initSkybox()** for the following code and call it after initializing the objects in your scene.

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

mySkyBox.Load(faces);
```

Define the following global variables that will represent our skybox and the shader used to draw it.

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

In the **initShaders()** function, load the skybox shader.

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

In the **renderScene()** function call the drawing function of the skybox after you called all the other drawing functions of the rest of the objects. If your project has shadows enabled using the shadow mapping technique, make sure not to draw the skybox in a depth map render pass, but rather in the normal render pass.

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

Assignment

- Follow the tutorial in order to display a skybox.
- Load another 3D object in your scene and compute the object's color taking into account also the reflections coming from the skybox.

Laboratory work 11 – Effects in OpenGL

Objectives

This laboratory presents you with basic notions about the concepts of fog effect, blending of colors and discarding of fragments.

Fog

We can increase the realism of a 3D scene by adding a fog effect which will create a particular atmosphere. By manipulating the attributes associated with the fog effect we can personalize the atmosphere and also we enhance the perception of depth.

The clear color should be chosen accordingly with the color of the fog.

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

In the fixed pipeline version of OpenGL there were three methods to compute fog (linear, exponential and square exponential). The same functionality can be accomplished in the programmable pipeline by adding the fog computation in our fragment shader.

Linear fog

The first method of computing fog is by using a linear interpolation function which will blend the fog color with the fragment color based on the distance of the fragment (computed in eye space).

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

The fogFactor values should be between 0.0 and 1.0.

Exponential fog

A better result can be obtained by taking into consideration the reduction of light intensity over the distance. The attenuation factor which will be using represents the fog density and it is constant over the scene. The result is a fast decrease of the fog factor compared to the linear approach.

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

Square exponential fog

The formula is basically the same with the previous one, the only difference come from the fact that we squared the fragment distance and the fog density.

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

Tutorial

Add the following function inside the fragment shader:

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

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

The clamp function constrains a value to lie between two values (minValue and maxValue).

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

The mix function linearly interpolates between two values. The resulting color can be computed also by:

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

The color variable represents the color computed using the light sources specified in your scene.

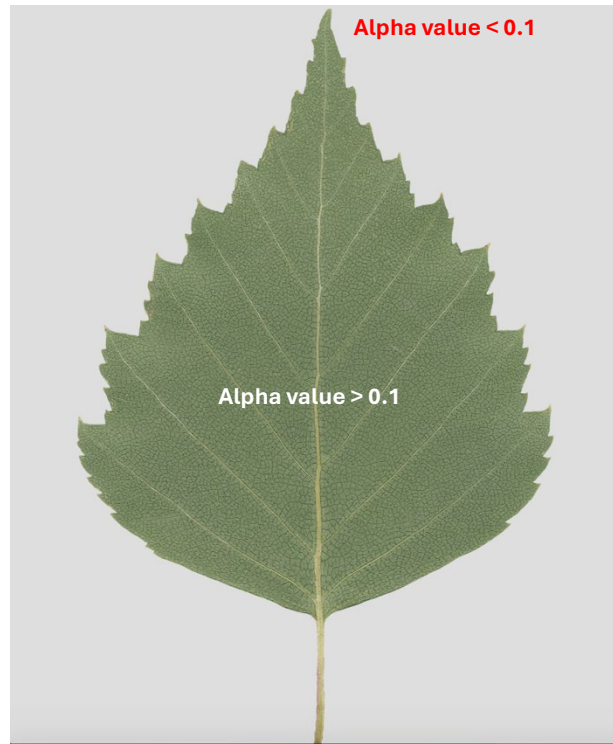
Discarding fragments

The alpha value from textures can be used to discard a fragment based on this value. For instance, you want to display a leaf and the 3D model is represented by a quad on which a leaf texture is mapped. We can check the value of the alpha channel and discard the fragment if the value is less than 0.1.

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

Update the call of the glTexImage2D function inside the ReadTextureFromFile method (from the Model3D class):

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



Blending colors

We can make objects to be transparent (or semi-transparent) by blending the object's color and the current color from the framebuffer (representing the objects already rendered) into a single color. The amount of transparency is defined by the alpha channel from the color representation. If the 4th component of the color is 1.0f then the object is opaque, if the value is 0.0f then the object is transparent. Values inside the (0.0f, 1.0f) interval represents a different amount of transparency.

Blending of colors is enabled in OpenGL by using the following function:

```
glEnable(GL_BLEND);
```

The resulting color after applying this operation is computed using the following equation:

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

- *sourceColor* – represents the color of the object that is currently rendered
- *sourceFactor* – defines the influence of the source color
- *destinationColor* – represents the color from the framebuffer
- *destinationFactor* – defines the influence of the destination color

The sourceFactor and destinationFactor are specified by using the glBlendFunction. In general, we use the alpha value of the source as the sourceFactor and (1 – alpha) as the destinationFactor. For this we have to call the following function:

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

The order in which we render objects in this situation is very important. When drawing such scene, we need to:

- draw all opaque objects
- sort the transparent objects
- draw the transparent objects in order (reverse depth order)

Assignment

- Implement all the topics presented in this laboratory work.