TNM046 Computer Graphics
Lab instructions 2022

Martin Falk

March 30, 2022, v2

Contents

Note

The code examples shown in the listings below can be found in `listings.zip` in case you want to copy them.

1 Introduction

In these lab exercises you will learn how to program interactive computer graphics using *OpenGL*, a popular and useful framework for 3D graphics. In contrast to most OpenGL tutorials you will find on the web, we will not take any shortcuts, but instead make you understand the details and ask you to implement as much as possible yourself. You will not just be using existing code libraries, you will write your own code to understand properly what you are doing. The exercises focus on the fundamentals. There will not be time for more advanced things during this introductory course, but you will learn enough to continue on your own using other resources. You will also have plenty of opportunities to learn and use more of OpenGL and 3D graphics in general in several other MT courses.

1.1 OpenGL and C++

OpenGL (opengl.org) is a cross-platform *application programming interface* (API). It is available on most computing platforms today: Microsoft Windows, MacOS X, Linux, iOS, and Android. It was originally designed for the programming language C, but it can be used with many programming languages. In contrast, WebGL uses JavaScript and is intended for providing 3D graphics with hardware acceleration for web-based content. The concepts learned in this lab can readily be applied there as well.

Even though OpenGL has a plain C interface without any object orientation, C++ is nowadays the most common language for modern OpenGL programming. Your adventures in OpenGL will begin with opening a window and drawing a single triangle in 2D and end with drawing an interactive view of a complicated 3D model loaded from a file, complete with a surface texture and simulated lighting. More advanced things like large scenes, animated models, advanced visual effects, and more complicated navigation and interaction will be saved for later courses in your second year and beyond.

1.2 C++ programming

This lab series is written in C++ while avoiding some of the more advanced C++ features. You have used C++ without the object oriented parts in your introductory course on programming and might even have heard about object oriented design like classes and inheritance. We will use objects, classes, and methods, but we will not use inheritance, templates, or operator overloading as they are not needed for the tasks presented here.

In this Section (1.2), we present a brief introduction to the C++ concepts which are useful in these lab exercises. Some details might be new, but for the most part they should be familiar.

1.2.1 C++ syntax

Let's recapitulate some basics of the C++ syntax briefly: identifiers are case sensitive, a semicolon terminates declarations and statements, and curly braces are used to group statements together. Control structures include `if`-statements, `switch` statements, `while` and `for` loops. Commonly used types for primitives are `int`, `double`, `float` and more. Most operators have a familiar syntax, an array index is placed within square brackets, and single-line comments are preceded by `//` while multi-line comments are indicated by `/* */`.

1.2.2 `float` versus `double`

In modern computers, there are two kinds of floating point representations: *single precision* and *double precision* floating point numbers, named `float` and `double` in C++. Unless you have a specific reason to use single precision, it is often a good choice to use double precision. In a modern CPU it is not really slower, and the extra precision can be quite useful.

However, most graphics hardware still uses single precision since `float` takes up half as much memory as a `double` (4 bytes vs. 8 bytes). In addition, using double precision on the GPU, if supported, typically comes with a performance penalty. Therefore you will often see constants in OpenGL/C++ code with the suffix “f”: `float x = 10.0f;` instead of just `float x = 10.0;`. The “f” tells the compiler that you are specifying a `float` and not a `double` constant. Omitting the “f” for float values will generate a compiler warning. Note that there should not be a space between the number and the trailing “f”.

1.2.3 Vectors, arrays and pointers

C and C++, both being hardware oriented languages, expose direct memory access to the programmer. Any variable may be referenced by its address in memory by means of *pointers*. A pointer is literally a memory address, denoting where the variable is stored in the memory of the computer. Some OpenGL functions require you to pass in your data using a pointer, for example uploading texture images or creating vertex buffers.

In order to avoid dealing with raw pointers and memory management, C++ provides so-called containers. The ones you will be using here are `std::vector`, a dynamic array which can grow and shrink, and `std::array`, an array with a fixed size. A vector of floats is defined as `std::vector<float> vec;` while an array of 4 integers is defined as `std::array<int, 4> arr;`. Use the `size()` member function to get the number of elements stored in the container (`vec.size();`).

You can access container element using `[]`, `arr[4]` will access the fifth element (indexing in C++ always starts at 0). Unfortunately, no bounds checking is performed, so an index beyond the last element of an array, or even a negative index, is possible and will compile. However, indexing out of bounds of an array is undefined behavior which can override data in other variables and cause fatal errors later in the program. A program that does out-of-bounds indexing will most likely be wrong or result in a crash.

For both containers, the array elements stored in adjacent memory addresses. The `data↩ ()` member function will return a pointer to the first element of the array. This pointer can then used for the OpenGL functions.

When dealing with regular variables or plain C arrays, it might be necessary to create a pointer yourself or request the address of the variable. Here are the main points to consider:

- A pointer is an address in memory. The value of a pointer describes *where* some other value is stored.
- An asterisk (`*`) is used both when declaring a pointer and accessing the data to which it points (*dereferencing* the pointer). The declaration `int *p` means that `p` is a pointer to an `int`, and `*p` gives you the value of the integer it points to.
- An ampersand (`&`) in front of a variable name creates a pointer with the address of that variable. If `q` is a variable, `&q` gives you a pointer to it. Careful: in case of a `std::vector` this will give you the address of the vector and *not* the data.
- Any variable of any type can have a pointer to it.
- Array variables (`int v[6];`) in C and C++ are actually just pointers to a sequence of variables of the same type, stored in adjacent memory locations.

1.2.4 Classes in C++

C++ is an *object oriented* language, which means it has objects, classes, and methods. Classes and structs are defined with the keywords `class` and `struct`, respectively. Both structs and classes can hold member variables as well as member functions. The only difference being that classes can be inherited and all members and functions are by default marked as private.

A simple example class is shown in Listing 1. The class has one data member (a variable inside the class) and three methods (functions inside the class). Two methods are `public`, meaning that they can be called from outside, and one is `private`, meaning that it can only be called from within the same class.

```
// A class that does nothing much useful
class FunAndProfit {
public:
    void fun() {  // public method
        gain += Profit();
    }

    float getGain() const { return gain; }

private:
    float profit() {  // private method
        return 1000000.0f;
    }

    float gain = 0.0f;  // private data member
};
```

Listing 1: An example class.

1.2.5 Constructors and destructors

More complex classes in C++ usually have one or more *constructor* methods and one *destructor* method. A constructor is called when you create an object, an *instance* of a class, and it provides a convenient way of setting it up for use. The destructor is called when the object is destroyed and it is responsible for cleaning up any memory that was allocated by the object while it was in existence.

Constructors are methods with the same name as the class without a return value. Constructors are called either when a variable of a class-based type is declared or when a statement with the keyword `new` is executed. A destructor is always called without arguments, in direct response to a `delete` operation on the object. The destructor has the same name as the class but with a ∼ (tilde) sign in front of it no return value.

Listing 2 shows a slightly extended version of the example class with two constructors and one destructor. The object being constructed can be referred to by the keyword `this`, but all data members and methods in the class are available by their unqualified name inside the class. In the example, `this->gain` would refer to the same variable as `gain`. In this case, the constructor with no arguments and the destructor only initialize the variable. If they are not provided, the C++ compiler will create *default constructors* and *default destructors*. If you manually allocate memory using the `new` statement, the destructor *must* call the corresponding `delete` statements. It is the responsibility of the programmer to keep track of such things and make all objects clean up properly after themselves. Failure to clean up will result in a *memory leak*, leaving portions of memory allocated but inaccessible and impossible to reclaim, which is a common but nasty bug in C++ programs.

```cpp
// A class that does nothing much useful
class FunAndProfit {
public:
    // constructor (initialize gain)
    FunAndProfit() : gain(0.0f) {}

    // constructor with arguments
    FunAndProfit(float funds) : gain(funds) {}

    // default destructor, since it doesn't do anything
    ~FunAndProfit() = default;

    void fun() {  // public method
        gain += Profit();
    }

    float getGain() const { return gain; }

private:
    float profit() {  // private method
        return 1000000.0f;
    }

    float gain;  // private data member
};
```

Listing 2: Constructor and destructor methods.

1.2.6 Accessing class members

To access fields of a struct and members of a class, you use the familiar "dot" notation, `myClass.fieldname`, as presented in Listing 3. If your variable is a *pointer* to a struct or class, you use the "arrow" notation instead, `myClassPointer->fieldname`.

```cpp
// Class member access
SomeClass myClass;          // a class
SomeClass* myClassPointer;  // a pointer to a class

myClassPointer = new SomeClass();  // dynamic creation, don't forget to delete it

myClass.someint = 43;          // set value by direct member access
myClassPointer->someint = 43;  // set value by pointer member access

(*myClassPointer).someint = 43;  // Equivalent but less clear syntax for "->"
(&myClass)->someint = 43;        // "&" creates a pointer to a variable

delete myClassPointer;  // don't forget to free dynamically created objects
```

Listing 3: Access to members of a class (or fields of a struct)

1.2.7 Source and header files

The source code for a C++ program is typically split into several files, but there are no strict rules for what goes where, how many files you should use, or even what the files should be called. The conventions for file organization and naming are not enforced by the compiler but there are recommendations.

We recommend to use *header files*, often with the extension `.hpp`, for *declarations* of classes and regular functions and *source files*, often with the extension `.cpp`, for the *implementations* that is the program code.

As an example, consider the simple class in Listing 1. Separating this class into a header file and a source file, the result looks like Listing 4 and Listing 5. Note that the syntax for the declaration of a class keeps the name of the class nicely at the top, but for the implementation you need to specify both the class name and the method name for each method. In the source file, the class name and the method name are separated by two colons (`FunAndProfit::fun()`).

Splitting code into two files for each class might seem like a nuisance, but it is very convenient. The header files are easy to read both by humans and compilers to get an overview of *what* a class does, without having to care *how* it does it. Header files are used to tell the compiler what is where in a large collection of files, and a header file, with appropriate comments, can work reasonably well as a first level of documentation.

Note that the source file has an `#include` statement to include its corresponding header file, which means that the source file actually contains both the declaration *and* the implementation of the class, while the header file contains only the declaration.

To use this class in your program insert the statement `#include "funAndProfit.hpp"` at the top of every source file where the class is used and make sure the source file for the class is compiled and included when linking the program.

1.2.8 The function `main()`

A stand-alone C++ program must have a function called `main`. It is defined like this:

```cpp
int main() {}
// or with command line arguments
int main(int argc, char *argv[]) {}
```

```cpp
#pragma once  // ensure this header is included at most once

// A class that does nothing much useful
class FunAndProfit {
public:
    FunAndProfit();
    ~FunAndProfit() = default;

    void fun();

    float getGain() const;

private:
    float profit();

    float gain;  // private data member
};
```

Listing 4: Header file `funAndProfit.hpp` for the example class.

```cpp
#include "funAndProfit.hpp"

FunAndProfit::FunAndProfit() : gain(0.0f) {}

void FunAndProfit::fun() {  // public method
    gain += Profit();
}

float FunAndProfit::getGain() const { return gain; }

float FunAndProfit::profit() {  // private method
    return 1000000.0f;
}
```

Listing 5: Source file `funAndProfit.cpp` for the example class.

The `main()` function is invoked when a C++ program starts. The program exits when the `main()` function exits or the statement `exit` is called.

1.2.9 Input and output

A program that does something useful might need some text input and output. In C++ this is commonly done using the `<iostream>` header. An alternative is the C functionality provided by `<cstdio>`, which you might see being used in some parts of the lab.

You will see some examples of their use in the example code in the following pages, but for more information on the C (and C++) standard library functions, please refer to separate documentation. A brief example of a "Hello World" program using both libraries is presented in Listing 6.

1.3 Programming with OpenGL

In this course we will be using modern OpenGL (version 3 and up). Unfortunately, there are still tutorials around teaching you the old, outdated way. If you see code which uses any of `glBegin()`, `glEnd()`, and `glVertex*()`, *stop reading it* and look for a different tutorial. OpenGL in its modern incarnation has three layers of coding, as presented below.

```cpp
#include <iostream>
#include <cstdio>

int main(int argc, char* argv[]) {
    std::cout << "Hello World from <iostream>!\n";    // cout is the console
    fprintf(stdout, "Hello World from <cstdio>!\n");  // stdout is also the console
    printf("Hello again from <cstdio>!\n");           // printf() sends to stdout
}
```

Listing 6: Hello World using both `<iostream>` and `<cstdio>`.

1.3.1 Window and input handling

The first level involves opening a window on the monitor and getting access to drawing things in it with OpenGL commands – in technical terms referred to as getting an *OpenGL context*. This is done differently under different operating systems. You will use a library called *GLFW* (glfw.org) to handle all the things that are specific to the operating system like creating a window. GLFW is available for Windows, Linux, and MacOS X and you may use any of those operating systems to do the lab work in this course. We will use GLFW for opening a window, setting its size and title, and getting an OpenGL context for it, and also to handle mouse and keyboard input. In addition, we will be using GLEW (github.com/nigels-com/glew). GLEW is short for "GL Extension Wrangler" and is pronounced like "glue". It manages and fetches platform specific OpenGL extensions.

1.3.2 CPU graphics programming

The second level of coding for OpenGL is about specifying vertex coordinates, triangle lists, normals, colors, textures, lights, and transformation matrices and sending this data to the graphics card (*graphics processing unit* or GPU) for drawing. Today, interactive 3D graphics is almost always drawn with hardware acceleration. This level is where we are going to spend much of our effort.

1.3.3 GPU graphics programming

The third level is *shader programming*. Shaders are small programs that are executed by the GPU. The ones important in this course are: *vertex shaders* and *fragment shaders*. There are four more shaders namely geometry shaders, tessellation control shaders, tessellation evaluation shaders, and compute shaders, which have more specialized and advanced purposes.

A **vertex shader** operates on each vertex of a 3D model. It receives vertex coordinates, normals, colors, and more from arrays in your main program and transforms the coordinates and normals to screen coordinates using transformation matrices. The reason for this being done on the GPU is that matrix and vector multiplications can be performed very efficiently on the parallel architecture of the GPU.

A **fragment shader** operates on the pixel level. Between the vertex shader and the fragment shader, a triangle is subdivided into a collection of pixel samples, *fragments*, that are to be drawn to the screen. For each of those pixels, the fragment shader receives interpolated coordinates, normals, and other data from the vertex shader, and is responsible for computing a color value for the pixel. In most cases, the color consists of four values: three color components (red, green and blue or RGB) and a transparency value (alpha or A).

Shaders for OpenGL are written in a special programming language called *OpenGL Shading Language* or GLSL. The full documentation is available in the Khronos OpenGL registry (khronos.org, check out the *Core Profile Specification*). It is a small language with a syntax similar to C++ but specialized for computations on vertex and pixel data. There are convenient vector and matrix data types available (`vec2, vec3, mat4, ...`) and it is quite easy to learn GLSL if you know computer graphics and a C-like programming language.

There are lecture notes for this course that cover the fundamentals of shader programming and GLSL, but we will not go into details. You will not be required to learn all the intricacies of GLSL. In the exercises, you will write simple vertex shaders to transform vertices and normals from model space to screen space and likewise simple fragment shaders to compute lighting and color for your objects. Further adventures in shader programming are encouraged, but they are outside the scope of this course. You will learn more in later courses on computer graphics. If you are curious and do not want to wait, we recommend the tutorials at opengl-tutorial.org.

1.4 OpenGL and its C roots

OpenGL was originally written for C. In the OpenGL API, there are no classes, no objects, no constructors or destructors, no methods, no method overloading, and no packages. You use only variables and functions. There are not even any composite types (`struct`). Everything is handled by primitive types, arrays, and pointers. The OpenGL header introduces some type names like `GLint`, `GLuint`, and `GLfloat`, but they are simply aliases to other primitive types to compensate for platform differences. A `GLint` is guaranteed to be a 32-bit integer while `GLuint` refers to a 32-bit unsigned integer and `GLfloat` to a regular `float`.

Because they are not organized in classes or namespaces, all OpenGL functions have a globally unique name starting with `gl`. One result of this is that functions with similar purposes that operate on different kinds of data need to be distinguished by unique names. The OpenGL function to send a parameter value to a shader program (a concept we will learn about later) may take floating point, integer, or unsigned integer arguments and possibly 2, 3, and 4-element vectors of either type. This results in the following twelve function names being used for what is basically the same purpose:

```
glUniform1f(), glUniform2f(), glUniform3f, glUniform4f(),
glUniform1i(), glUniform2i(), glUniform3i(), glUniform4i(),
glUniform1ui(), glUniform2ui(), glUniform3ui(), glUniform4ui()
```

You will therefore see some documentation referring to `glUniform3*()` instead.

1.5 OpenGL bottlenecks

The execution of an interactive 3D graphics application can encompass many different tasks: object preparation, drawing, creation and handling of texture images, animation, simulation, and handling of user input. In addition to that there may be a considerable amount of network communication and file access going on. 3D graphics is taxing even for a modern computer and real world 3D applications often end up hitting the limit for what a computer can manage in real time.

The bottleneck can be in pure rendering power, meaning that the GPU has trouble keeping up with rendering the frames fast enough in the desired resolution. Such problems can often be alleviated by reducing the resolution of the rendering window. However, sometimes the bottleneck is at the vertex level, either because the models are very detailed or because some particularly complicated calculations are performed in the vertex shader. A solution in that case is to reduce the number of triangles in the scene.

Sometimes the bottleneck is not the graphics, but the simulation or animation performed on the CPU before the scene is ready for rendering. The remedy for that is different for each case and it is not always easy to know what the problem is. A lot of effort in game development is spent on balancing the requirements from the gameplay simulation, the geometry processing, and the pixel rendering in a manner that places reasonable demands on the CPU and GPU horsepower.

Your exercises in this lab series will probably not hit the limit on what a modern computer can manage in real time, but in the last exercise, where you load models from file, we will provide you with an example of a large model with lots of triangles to show what OpenGL is capable of and to show that there is still a limit to what you can do if you want interactive frame rates.

2 Exercises

The remainder of this document consists of five exercises, Sections 2.1 to 2.5, each containing several tasks. Each task is enclosed in a frame, like this:

> This is a task which you should perform.

Information on *how* to perform the tasks can be found in this text and in the textbook (Steven J. Gortler: Foundations of 3D Computer Graphics). In some cases you may need to look up OpenGL documentation on the Internet. Good documentation on OpenGL is available on opengl.org. A good set of more general tutorials going quite a bit further than these exercises is published on opengl-tutorial.org. Other sources of information are the lecture notes, scheduled lectures, and practice sessions.

In most cases you also need to ***prepare*** and ***think*** to perform the tasks presented to you. You will probably need to spend extra time outside of scheduled lab sessions to prepare the tasks, and perhaps to finish up in case you don't quite manage to do the tasks during the scheduled hours.

Be prepared to demonstrate your solutions to the lab assistant for approval and comments. You are expected to demonstrate the result of all tasks of Sections 2.1 to 2.5 during the six scheduled lab sessions.

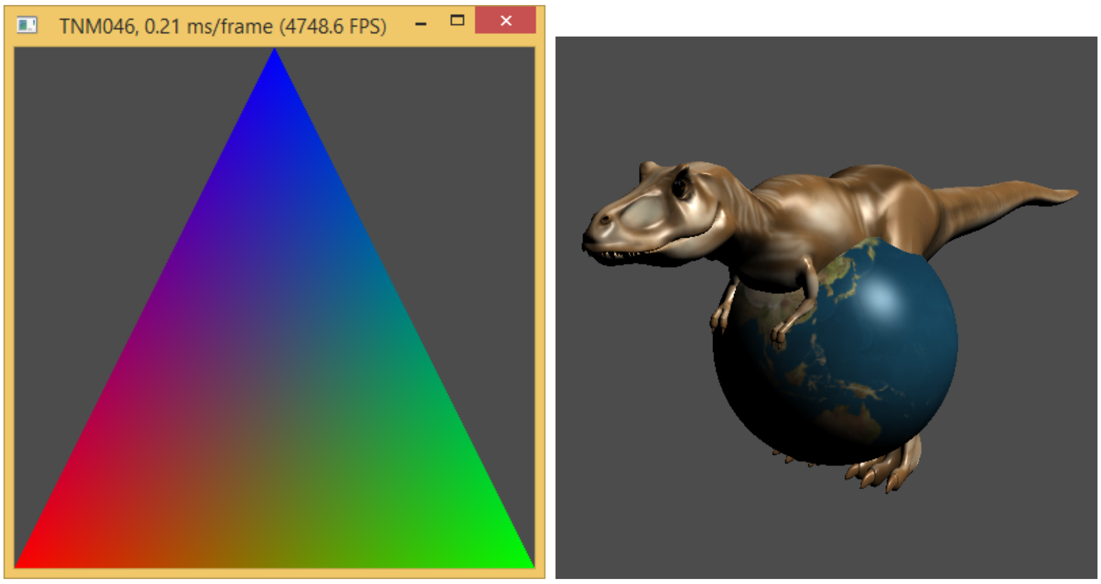

Figure 1: *Left:* A single triangle (after exercise 1). *Right:* Thousands of triangles with textures and lighting rendered in an interactive perspective view (after exercise 5).

2.1 Your first triangle

2.1.1 Preparations

This first exercise is designed to introduce you to rendering a single triangle. Read Section 2.1 **before the lab**, look at the code, and get an overview of what you are supposed to do during the lab session.

2.1.2 OpenGL program structure

There is a general structure and sequence of operations which is common to all OpenGL rendering. First, OpenGL needs to be initialized, various settings and preparations need to be made, and data needs to be uploaded in advance to the GPU to be available during rendering. Then, the program enters the *rendering loop* to draw the objects. The general structure of a typical OpenGL program is:

- Create and show a window
- Set up OpenGL
- Load meshes, textures, and shaders from files
- Until the user asks the program to quit, repeat:
  - Clear the screen
  - Set up data for drawing (meshes, matrices, shaders, textures)
  - Draw triangles
  - Repeat the previous two steps if you want to draw more objects
  - Swap the back and front buffers to show your rendered image
- Close down OpenGL

OpenGL is a state machine where settings made by function calls take effect immediately and remain active for subsequent draw calls until the setting is cleared or changed. Drawing commands issued in the drawing phase are executed right away. Because the GPU works quite separately from the CPU, triangles from a previous drawing command may still be in the process of being drawn while the main CPU program prepares the data for the next drawing command. Similarly, changes to OpenGL settings will only influence drawing commands following the changes. The CPU does not need to wait for the GPU to finish drawing before sending the next command. All commands are sent to the GPU without any exact knowledge of when they will actually execute, how long they will take or exactly when they finish. The only thing that is guaranteed is the *order* of the operations.

2.1.3 The OpenGL window

To get you started, we present you with an example that opens a window, enters a rendering loop and waits for the program to be terminated by the user either by closing its window or by pressing the ESC key. For the moment, the rendering loop just erases the frame, so all you will see is a blank screen in a specified color. Listing 7 presents working C++ code for a complete program, but a file more suitable as a base for you to build upon for the upcoming exercises is in the file `GLprimer.cpp` in the lab material. That file contains some more details and has more extensive comments.

Note that even this very simple program has the general structure presented in Section 2.1.2, even though many of the steps are still missing.

Study the code in `GLprimer.cpp` and in Listing 7 carefully. We recommend you print it out and write on the printout until you understand its structure. It is not a lot of code, and it has lots of comments, so it should be readable to you. This code will be the foundation for all the following exercises, and you should be as familiar with it as if you had written it yourself. Take particular note of the overall structure, with a setup part that is executed once and a rendering loop that is executed again for each frame.

Make sure to **read the code** in `GLprimer.cpp`. Read it line by line including the comments which contain additional information. Do *not* proceed until you have done so.

When you have read the code, use CMake and the contents of the zip file to create a project for your programing environment of your choice (for example Visual Studio, XCode, Visual Studio Code, . . .). Open the project, compile the code, and make sure it runs. Your window should look like Figure 2, left.

The window will be blank because we are not drawing anything. It should open and close without error messages. If you download the `.zip` package with the lab material for this course, it will include a CMake file, the source files, and both the GLFW and GLEW libraries. Using CMake and setting the paths of the source code and a build folder should give you project files for your compiler and operating system. We recommend that the build folder is *not* the same as your source code and not a subfolder. Pressing configure the first time will present you with a choice of available compilers, for example Visual Studio 2019, Xcode, Makefiles, ... If you want to undo your choice you can either select a different build folder or delete the CMake cache (in the File menu). After running continue, *Generate* will create the project files which you can work with. This setup should make sure that your program is linked with the OpenGL library and both GLFW and GLEW. You need to use CMake only once, then you can open the project directly in the programming environment of your choice.

There is a *rendering loop* in the program, and it follows the general structure presented in Section 2.1.2. However, we are not drawing anything yet – the rendering loop just clears the screen. We are not even doing the clearing quite right. To make OpenGL aware of

```cpp
/*
 * A C++ framework for OpenGL programming in TNM046 for MT1
 *
 * Authors: Stefan Gustavson (stegu@itn.liu.se) 2013-2015
 *          Martin Falk (martin.falk@liu.se) 2021
 *
 * This code is in the public domain.
 */
#include <GL/glew.h>     // provides easy access to advanced OpenGL functions and extensions
#include <GLFW/glfw3.h>  // GLFW handles the window and user input

int main(int argc, char* argv[]) {
    glfwInit();  // Initialise GLFW

    const GLFWvidmode* vidmode;  // GLFW struct to hold information about the display
    vidmode = glfwGetVideoMode(glfwGetPrimaryMonitor());  // Determine the desktop size
    // Make sure we are getting an OpenGL context of at least version 3.3.
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

    // Open a square window (aspect 1:1) of size 500x500 pixels
    GLFWwindow* window = glfwCreateWindow(500, 500, "GLprimer", nullptr, nullptr);
    if (!window) {
        glfwTerminate();  // No window was opened, so we can't continue
        return -1;
    }
    // Make the newly created window the "current context"
    glfwMakeContextCurrent(window);
    // Initialize glew
    GLenum err = glewInit();
    if (GLEW_OK != err) {
        glfwTerminate();
        return -1;
    }

    // Set the viewport (specify which pixels we want to draw to)
    int width, height;  // width and height of the window
    glfwGetWindowSize(window, &width, &height);
    glViewport(0, 0, width, height);  // Render to the entire window
    glfwSwapInterval(0);              // Do not wait for screen refresh between frames

    // Rendering loop: exit if window is closed by the user
    while (!glfwWindowShouldClose(window)) {
        // Set the clear color to a dark gray (RGBA)
        glClearColor(0.3f, 0.3f, 0.3f, 0.0f);
        // Clear the color and depth buffers for drawing
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        /* ---- (Rendering code should go here) ---- */

        // Swap buffers, i.e. display the image and prepare for next frame.
        glfwSwapBuffers(window);

        glfwPollEvents();  // Poll events (read keyboard and mouse input)
        // Exit also if the ESC key is pressed.
        if (glfwGetKey(window, GLFW_KEY_ESCAPE)) {
            glfwSetWindowShouldClose(window, GL_TRUE);
        }
    }

    // Close the OpenGL window, terminate GLFW, and quit.
    glfwDestroyWindow(window);
    glfwTerminate();
}
```

Listing 7: C++ code for a very small OpenGL program using GLFW.

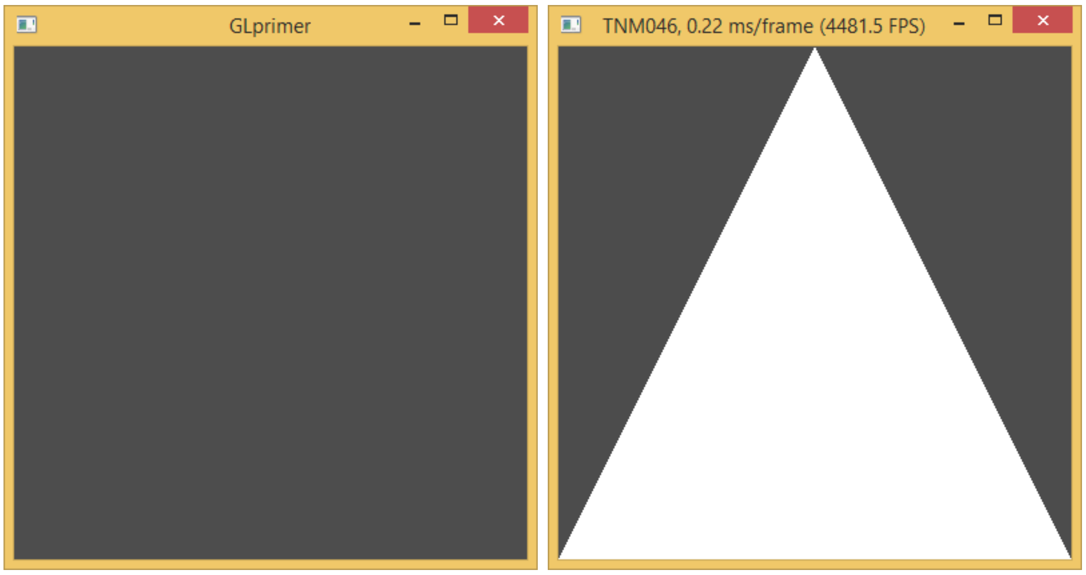

Figure 2: The empty window from Listing 7, and your first triangle.

window resizing, we need to check the window size in the rendering loop, before we draw each frame.

Try using the mouse to resize the window to a larger size after it opens. In most cases, only the original region of pixels will be cleared.

The two lines that check the window size and set the drawing area for OpenGL are the calls to `glfwGetWindowSize()` and `glViewport()`. Move those two lines to the very beginning of the rendering loop, before the call to `glClear()`. Now resize the window and see what happens. Every pixel should now be cleared correctly even if you make the window larger after it has opened.

2.1.4 Vertex coordinates

Drawing in OpenGL is generally performed by sending vertex coordinates and triangle lists in large chunks to the GPU. This is done by filling arrays with data and using special OpenGL functions to copy that data to *vertex buffers*. All the data is sent in one go, and it is copied to the GPU memory to speed up the drawing and to enable efficient repeated rendering of the same object in many consecutive frames. The code we ask you to use below might seem like a lot of work for drawing a single triangle, or even a few dozen triangles, but keep in mind that the process is designed to make it possible to draw thousands or even millions of triangles in an efficient and structured manner.

What follows is a detailed presentation of how to create and use a vertex buffer. You do not need to learn how to do this by heart. The main point here is that you need to do it the right way. You are expected to remember *what* to do, but not exactly how.

A *vertex buffer* consists of a sequence of vectors, packed together in sequence as a linear array of `float` numbers. OpenGL defines its own data type called `GLfloat`, an alias for `float`, and you may use either type interchangeably in your code.

A simple demo to get us started is to define constant arrays for the vertex and index data and initialize them on creation in the C++ code. In this manner, simple vertex arrays can be created and set to suitable values directly in your source code, and compiled into the program. This is not the common way of getting geometry into an OpenGL program, but it is useful for simple demonstrations.

To send data to the GPU, you need to go through several steps:

1. Create a vertex array object (VAO) to refer to your geometry
2. Activate ("bind") the vertex array object
3. Create a vertex buffer
4. Activate ("bind") the vertex buffer
5. Create a data array that describes your vertex coordinates
6. Copy the array from CPU memory to the buffer object in GPU memory
7. Enable the vertex attributes, so the shader can access the vertex buffer
8. Create an index buffer
9. Activate ("bind") the index buffer
10. Create a data array that describes your triangle list
11. Copy the array from CPU memory to the buffer object in GPU memory
12. Possibly repeat steps 1 through 10 to specify more objects for the scene

This sequence of operations is designed to make the data easy to for the GPU to interpret and store. Other data may be associated with each vertex than just the coordinates: for example vertex colors, normals, and texture coordinates. In this case, the sequence of operations changes somewhat from what was presented above, but it follows the same general structure: first bind a buffer, then copy data. More on this later.

After creating the vertex array and populating it with data, it can be drawn:

1. Activate the vertex array object you want to draw ("bind" it again)
2. Issue a drawing command to the GPU
3. Possibly repeat steps 1 and 2 until the entire scene has been drawn

The code for these operations of a single object is given in Listing 9. Note that the code should be put in three different places in your program: in the variable declarations, in the set-up part before the rendering loop, and in the rendering loop.

The files `Utilities.hpp` and `Utilities.cpp` contain helper functions in the `util`↩ namespace. The function `displayFPS()` shows the frame rate for the rendering loop in the window title bar. Note that although `Utilities.hpp` does not declare a class, it does declare a namespace so you need to call the functions with their qualified names `util::displayFPS()` or insert the statement `using namespace util` at the top of your program (only in the cpp file!). We recommend using the qualified names for clarity. You will be using this function only once in your program.

Insert the lines in Listing 8 at the top of your program and in your rendering loop. Compile and run. You should see the frame rate displayed in the window title.

```cpp
// --- Add this line to your includes
#include "Utilities.hpp"

// --- Insert this line into your rendering loop.
util::displayFPS(window);
```

Listing 8: C++ function call to display the frame rate.

If the frame rate is not thousands of frames per second but rather 60 FPS, you need to disable "vertical sync" in your graphics driver. Exactly how to do this depends on your GPU and your operating system, but in the course lab room with Windows computers and NVidia GPUs right click on the desktop, select "NVidia Control Panel", find the tab "Manage 3D Settings" and set "Vertical Sync" to "off". Limiting the frame rate to the update frequency of the monitor is generally a good idea, if nothing else to conserve power, but disabling it gives a good hint on how close you are to the performance limit of your GPU.

In the C++ code, *vertex array objects* and *buffer objects* are just unsigned integers (`GLuint`). They are not even pointers, they are just numbers. The actual vertex data resides in a data structure in the GPU memory, and the integers are just ID numbers that are associated with that data structure, to keep track of the data that was sent to the GPU and to refer to it later during rendering. Think of these ID numbers as "handles" or "references", unique identifiers that are used to refer to data structures that are actually stored on the GPU rather than in the main CPU memory.

Once you have familiarized yourself with what the code in Listing 9 does, add it to your program, compile and run it. Your window should look like Figure 2, right.

If you are unfamiliar with the notation `1.0f` in the code, see Section 1.2.2.

Running the code, you should see a triangle on the screen, and it should appear white depending on your particular combination of GPU and OpenGL driver. In some instances, it is possible that nothing is drawn until you specify shaders, see Section 2.1.5 below. Note where your vertices end up. The default coordinate system in an OpenGL window extends across the entire window and ranges from -1 to 1 in both x and y, regardless of the pixel dimensions. The z direction points straight out from the screen. By default, the z coordinate of a vertex does not affect its apparent position on the screen. The view

```cpp
#include <vector>
// ------------------------------------------------------------------------
// --- Put this code at the top of your main() function.
// Vertex coordinates (x,y,z) for three vertices
const std::vector<GLfloat> vertexArrayData = {
    -1.0f, -1.0f, 0.0f,  // First vertex, xyz
    1.0f,  -1.0f, 0.0f,  // Second vertex, xyz
    0.0f,  1.0f,  0.0f   // Third vertex, xyz
};
const std::vector<GLuint> indexArrayData = {0, 1, 2};

// ------------------------------------------------------------------------
// ---- Put this code after glewInit(), but before the rendering loop

// Generate 1 Vertex array object, put the resulting identifier in vertexArrayID
GLuint vertexArrayID = 0;
glGenVertexArrays(1, &vertexArrayID);
// Activate the vertex array object
glBindVertexArray(vertexArrayID);

// Generate 1 buffer, put the resulting identifier in vertexBufferID
GLuint vertexBufferID = 0;
glGenBuffers(1, &vertexBufferID);
// Activate the vertex buffer object
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferID);
// Present our vertex coordinates to OpenGL
glBufferData(GL_ARRAY_BUFFER, vertexArrayData.size() * sizeof(GLfloat), vertexArrayData.data(),
             GL_STATIC_DRAW);
// Specify the format of the data in the vertex buffer, and copy the data.
// The six arguments specify, from left to right:
// Attribute 0, must match the "layout" statement in the shader.
// Dimensions 3, means 3D (x,y,z) - this becomes a vec3 in the shader.
// Type GL_FLOAT, means we have "float" input data in the array.
// GL_FALSE means "no normalization". This has no meaning for float data.
// Stride 0, meaning (x,y,z) values are packed tightly together without gaps.
// Array buffer offset 0 means our data starts at the first element.
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
// Enable vertex attribute array 0 to send xyz coordinates to the shader.
glEnableVertexAttribArray(0);

// Generate 1 buffer, put the resulting identifier in indexBufferID
GLuint indexBufferID = 0;
glGenBuffers(1, &indexBufferID);
// Activate (bind) the index buffer and copy data to it.
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferID);
// Present our vertex indices to OpenGL
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indexArrayData.size() * sizeof(GLuint), indexArrayData.data(),
             GL_STATIC_DRAW);
// Deactivate the vertex array object again to be nice
glBindVertexArray(0);

// ------------------------------------------------------------------------
// ---- Put the following code in the rendering loop
// Activate the vertex array object we want to draw (we may have several)
glBindVertexArray(vertexArrayID);
// Draw our triangle with 3 vertices.
// When the last argument of glDrawElements is nullptr, it means
// "use the previously bound index buffer". (This is not obvious.)
// The index buffer is part of the VAO state and is bound with it.
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, nullptr);

// ------------------------------------------------------------------------
// ---- Put the following code directly after the rendering loop (before glfwDestroyWindow())
// release the vertex and index buffers as well as the vertex array
glDeleteVertexArrays(1, &vertexArrayID);
glDeleteBuffers(1, &vertexBufferID);
glDeleteBuffers(1, &indexBufferID);
```

Listing 9: C++ code to define and use minimal vertex and index arrays.

can of course be changed and objects can be transformed to make their z depth influence how they are drawn. Also, objects outside the range $[-1, 1]$ in z get *clipped* (they are not rendered), which can lead to somewhat unexpected results. We will explain more on z clipping later, in Section 2.4.4. For the time being, keep your objects to within $[-1, 1]$ in z.

2.1.5 Shaders

Modern use of OpenGL requires shaders. We might have gotten away without them above, but only because many OpenGL implementations are somewhat forgiving. A failure to specify shaders will often result in a default behavior of not transforming the vertices (vertex shader) and drawing all pixels in a solid white color (fragment shader). Not having shaders is formally an OpenGL error.

Create two empty files and write code in them according to Listings 10 and 11. You will extend it later. Note also that the first line `#version 330 core` is required. It is not a comment, it is a *directive* to the GLSL compiler telling it which version of the language we are using. Name the files whatever you like, but use the extension `.glsl` (or alternatively `.vert` and `.frag`) and use descriptive names. We suggest `vertex.glsl` and `fragment.glsl`.

```glsl
#version 330 core

layout(location = 0) in vec3 Position;

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

Listing 10: (GLSL) A minimal vertex shader

```glsl
#version 330 core

out vec4 finalcolor;

void main() {
    finalcolor = vec4(1.0, 0.5, 0.0, 1.0);
}
```

Listing 11: (GLSL) A minimal fragment shader

In most development environments you can add the files to your project and open and edit them with the C++ editor (there might even be a GLSL plugin available). Just make sure you *don't* include the shaders in the list of files that are to be compiled by the C++ compiler. The shader code is not meant for the CPU and it is not compiled into the executable file. Instead, the shader files are read when the program is run and the GLSL code is compiled by the OpenGL driver and sent to the GPU for execution. This means that if you only make changes to the shaders, you do not need to recompile your main program. Just change the text in the shader files and run your program again.

To use a shader program of one vertex and one fragment shader, they need to be compiled, linked and activated together as a *program object* in OpenGL. The class `Shader`, defined in `Shader.hpp` and `Shader.cpp`, takes care of the necessary steps. The method `Shader↩ ::createShader()` takes as input arguments the names of two files, one for the vertex shader and one for the fragment shader, and compiles them into a program object. If the compilation fails for some reason, like if a file cannot be found or if you have a syntax error in your code, an error message is printed and no valid program object is created.

Using the program object after it has been created is done by calling the OpenGL function `glUseProgram()` with the program object as argument. All subsequent objects will be rendered using that shader program until you make another call to `glUseProgram()`.

The C++ code you need to add to your program is presented in Listing 12. As you can see from the class declaration in `Shader.hpp`, shader program objects are plain integers in C++, much like vertex array objects. The actual shader program is stored in GPU memory, and the integer is just a numeric ID, a reference to the shader program that was sent to the GPU. The exact integer value does not mean anything, and it should never be changed by the programmer. The integer value 0 is special, however, and means "no program". It is used to turn off any program objects that were activated previously – you deactivate a shader program by calling `glUseProgram(0)`.

Using the code in Listing 12, add shaders to your program. Compile and run it to see that things work the way they should. Your window should look like Figure 3, left.

```
// --- Add this to the includes --------------------------------------------------
#include "Shader.hpp"

// --- Add this to the variable declarations --------------------------------------
Shader myShader;

// --- Add this in main() after glewInit() and before the rendering loop ----
myShader.createShader("vertex.glsl", "fragment.glsl");

// --- Add this to the rendering loop, right before the call to glBindVertexArray()
glUseProgram(myShader.id());
```

Listing 12: Shader activation using the class `Shader`

Orange is nice, but we might want to draw the triangle in a different color. The pixel color is set in our fragment shader by assigning an RGBA value to the the output `vec4` variable named `finalcolor`. The four components are in the range 0.0 to 1.0.

Edit your fragment shader to draw the triangle in a different color.

Note that you do not need to recompile your code if you only change the GLSL files. It is enough to restart your program. The shaders are read from files and compiled by the OpenGL driver every time the program starts.

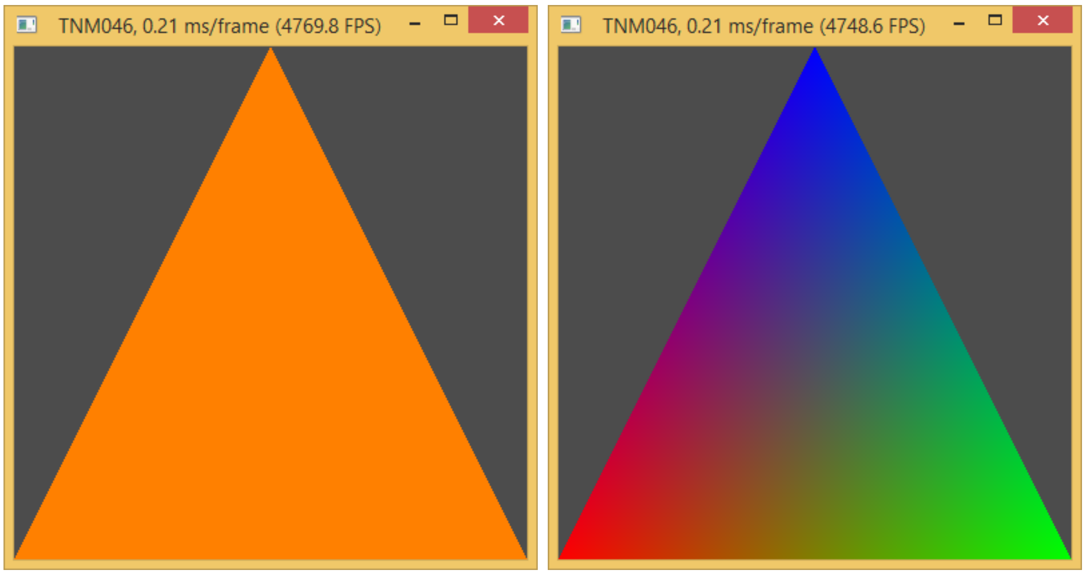
TNM046, 0.21 ms/frame (4769.8 FPS)
TNM046, 0.21 ms/frame (4748.6 FPS)

Figure 3: The triangle with proper shaders and added vertex colors.

Change the position of some vertices by editing the array in the C++ code.

Changes to the C++ code, however, require a recompilation.

The vertex positions can also be changed in the vertex shader. That, in fact, is its very purpose. The vertex shader is executed for each vertex, so changes to the vertex shader affect all vertices in the object.

Change the position of all vertices by editing the vertex shader. Translate the triangle by adding a constant number or a constant vector, and scale the triangle by multiplying the coordinates with a scalar (uniform scaling) or a vector (non-uniform scaling).

2.1.6 Vertex colors

The triangle is colored with a single color. Graphics hardware has a built-in mechanism for *interpolation* between vertices. Any property of a triangle is allowed to vary smoothly across its surface, including normal vectors, texture coordinates, and colors. To specify additional properties for each vertex, you need to activate additional *vertex attribute arrays* similar to the one you used for the vertex coordinates. Each additional array needs to be connected to input variables in the vertex shader, and the vertex shader needs to pass the interpolated values on to the fragment shader through a pair of variables: one `out` variable from the vertex shader and one `in` variable in the fragment shader. Both should have the same name and type.

To specify a different color for each vertex, you can use the code in Listing 13. You also need to make changes to the shader code to pass the color values to the shaders and use them for rendering. The required additional shader code is presented in Listing 14. Note that you need to define a pair of variables with the same name and type: an `out` variable in the vertex shader variable and an `in` variable in the fragment shader. Such pairs of variables are interpolated between vertices across the surface of a triangle. In older versions of GLSL, they were called `varying` variables, and even though that keyword has been replaced with `in` and `out`, the name *varying* was quite appropriate: these variable are being interpolated between vertices and *vary* over the surface of a triangle.

Note that the vertex position which we set in our first vertex shader is special in GLSL, in that it does not require you to define a pair of in/out shader variables to pass it from the vertex shader to the fragment shader. The special, pre-declared variable `vec4 ↩ gl_Position` in the vertex shader holds the transformed vertex position and *must* be assigned a value. This is because vertex positions are essential to OpenGL and are required to draw anything. (If we don't know the positions of the vertices, we have no idea which pixels to draw.) All *other* properties of a vertex (colors, normals, texture coordinates etc.) are optional.

```
// --- Add this after the other vertex array declarations --------------
const std::vector<GLfloat> colorArrayData = {
    1.0f, 0.0f, 0.0f,  // Red
    0.0f, 1.0f, 0.0f,  // Green
    0.0f, 0.0f, 1.0f,  // Blue
};

// --- Add this after glEnableVertexAttribArray(0) ---------------------
// Generate a second vertex buffer, activate it and copy data to it
GLuint colorBufferID = 0;  // Vertex colors
glGenBuffers(1, &colorBufferID);
glBindBuffer(GL_ARRAY_BUFFER, colorBufferID);
glBufferData(GL_ARRAY_BUFFER, colorArrayData.size() * sizeof(GLfloat), colorArrayData.data(),
             GL_STATIC_DRAW);
// Tell OpenGL how the data is stored in our color buffer
// Attribute #1, 3 dimensions (R,G,B -> vec3 in the shader),
// type GL_FLOAT, not normalized, stride 0, start at element 0
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
// Enable a second attribute (in this case, to hold vertex colors)
glEnableVertexAttribArray(1);

// ------------------------------------------------------------------------
// ---- Put the following code directly after the rendering loop next to the others
// release the color buffers
glDeleteBuffers(1, &colorBufferID);
```

Listing 13: Specifying vertex colors as a vertex attribute array

```
// --- Add this to the declarations in the vertex shader
layout(location = 1) in vec3 Color;
out vec3 interpolatedColor;
// And somewhere in its main() function, add this:
interpolatedColor = Color; // Pass interpolated color to fragment shader

// --- Add this to the declarations in the fragment shader
in vec3 interpolatedColor;
// And in its main() function, set the output color like this:
finalcolor = vec4(interpolatedColor, 1.0);
```

Listing 14: (GLSL) Using vertex colors in the shaders

Specify different colors for each vertex of your triangle and edit the shaders to render the triangle with an interpolated color across its surface. Your window should look like Figure 3, right.

2.1.7 Refactoring

Looking at the code you have in `main()`, it is now getting rather long and difficult to navigate. There are a lot of low-level OpenGL details that make the code hard to read, and you have a clear repetition of code where you create the two vertex buffers. It is a good idea to break out that repetitive code to a function `createVertexBuffer()`, according to Listing 15. Move the index buffer creation to a function `createIndexBuffer()`, also presented in Listing 15. Even though that code is used only once it does expose some unnecessary details cluttering up the code in `main()`. You can put the function at the top of you main file `GLprimer.cpp`, or you can add it to `Utilities.cpp` if you feel like hiding it from view even more. If you put it in `Utilities.cpp`, remember to also update the header file `Utilities.hpp`, otherwise the compiler will not find the functions.

Using your newly created functions, part of the `main()` function can be shortened to a few lines in Listing 16.

Refactor your code according to the suggestions in Listings 15 and 16. Compile and run the code. Make sure the visual result is still the same as before. Remember to store the buffer IDs returned by these functions and release them after the rendering loop. Each call of `glGenBuffers()` or `glCreateVertexArrays()` must be matched with `glDeleteBuffers()` or `glDeleteVertexArrays()`, respectively!

Most problem solving in programming involves *incremental* programming, where you start with a simple program and add stuff to it over time. You should always be on the lookout for this kind of restructuring that improves the abstraction by removing unnecessary details from higher levels of code. Sometimes you can identify the need for a new class, but sometimes it is enough to just move a long sequence of code into a separate function.

```cpp
GLuint createVertexBuffer(int location, int dimensions, const std::vector<float>& vertices) {
    GLuint bufferID;
    // Generate buffer, activate it and copy the data
    glGenBuffers(1, &bufferID);
    glBindBuffer(GL_ARRAY_BUFFER, bufferID);
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(float), vertices.data(), GL_STATIC_DRAW);
    // Tell OpenGL how the data is stored in our buffer
    // Attribute location (must match layout(location=#) statement in shader)
    // Number of dimensions (3 -> vec3 in the shader, 2-> vec2 in the shader),
    // type GL_FLOAT, not normalized, stride 0, start at element 0
    glVertexAttribPointer(location, dimensions, GL_FLOAT, GL_FALSE, 0, nullptr]);
    // Enable the attribute in the currently bound VAO
    glEnableVertexAttribArray(location);

    return bufferID;
}

GLuint createIndexBuffer(const std::vector<unsigned int>& indices) {
    GLuint bufferID;
    // Generate buffer, activate it and copy the data
    glGenBuffers(1, &bufferID);
    // Activate (bind) the index buffer and copy data to it.
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, bufferID);
    // Present our vertex indices to OpenGL
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), indices.data(),
                 GL_STATIC_DRAW);

    return bufferID;
}
```

Listing 15: Functions to move some details out of `main()`.

```cpp
// Generate 1 Vertex array object, put the resulting identifier in vertexArrayID
glGenVertexArrays(1, &vertexArrayID);
// Activate the vertex array object
glBindVertexArray(vertexArrayID);

// Create the vertex buffer objects for attribute locations 0 and 1
// (the list of vertex coordinates and the list of vertex colors).
GLuint vertexBufferID = createVertexBuffer(0, 3, vertexArrayData);
GLuint colorBufferID = createVertexBuffer(1, 3, colorArrayData);
// Create the index buffer object (the list of triangles).
GLuint indexBufferID = createIndexBuffer(indexArrayData);

// Deactivate the vertex array object again to be nice
glBindVertexArray(0);

// -----------------------------------------------------------------------
// ---- Put the following code directly after the rendering loop
// release the vertex array and the buffers
glDeleteVertexArrays(1, &vertexArrayID);
glDeleteBuffers(1, &vertexBufferID);
glDeleteBuffers(1, &colorBufferID);
glDeleteBuffers(1, &indexBufferID);
```

Listing 16: Using `createVertexBuffer()` and `createIndexBuffer()` to send vertex data to the GPU.

2.2 Transformations, more triangles

2.2.1 Preparations

This exercise requires some preparations before the lab. Otherwise you will not be able to finish the exercise in the 4 hours of the lab session.

- Read Section 2.2, look at the code and get an overview of what you are supposed to do during the lab session.
- **Before** you arrive at the scheduled session, write the code for matrix multiplication in C++. Do it on paper. Verify the code by hand.
- Write down what transformation matrices should look like for rotations around the x, y and z axes, and how they should be represented as 16-element arrays in C++.
- Write down the vertex array and the index array for a cube with six square faces, with each square built from two triangles. Do it on paper. Note the winding order.

A brief explanation of winding order is given in Section 2.2.3. For a more thorough explanation of winding order see Gortler chapter 12.2 or the section on back face culling in any other computer graphics textbook.

2.2.2 Transformations and uniform variables

A very fundamental part of 3D graphics are *transformations*. They are represented by 4×4 matrices and homogeneous coordinates as described in the lectures and the textbook. Transformations are usually performed on the GPU, because they can be done faster and more efficiently there. Matrices can be created and used entirely within the shader code, but for the most part the transformations need to be sent from the main program to the shader in order to reflect camera movements and other dynamic changes to the scene. Data that needs to change, but is constant, or uniform, across an entire object rather than unique for every vertex, is sent to shaders by a mechanism called *uniform variables*. A variable in CPU memory is sent to the GPU by a call to one of a set of OpenGL functions starting with `glUniform`. The full mechanism for defining and using uniform variables in a shader program consists of four steps:

- Create a variable of the desired type in the shader program.
- Create and set a variable of the corresponding type in the CPU program.
- Find the uniform variable by name in the shader program object.
- Copy the value from the CPU variable to the shader variable.

A uniform variable `time` of type `float` is declared in GLSL as follows

```glsl
uniform float time;
```

How to find this variable by name and set it from a C++ program is shown in Listing 17. Note that if you are not using the uniform variable in the shader, the shader compiler might optimize it away and you will not find its location.

```cpp
// Do this before the rendering loop
GLint locationTime = glGetUniformLocation(myShader.id(), "time");
if (locationTime == -1) {  // If the variable is not found, -1 is returned
    std::cout << "Unable to locate variable 'time' in shader!\n";
}

// Do this in the rendering loop to update the uniform variable "time"
float time = static_cast<float>(glfwGetTime());  // Number of seconds since the program was started
glUseProgram(myShader.id());                     // Activate the shader to set its variables
glUniform1f(locationTime, time);                 // Copy the value to the shader program
```

Listing 17: Setting the `uniform float time` shader variable from C++.

In GLSL, uniform variables are declared in a global scope: at the top of the shader code, outside of any functions. In most structured programming languages, global variables are considered bad programming style and can mostly be avoided, but GLSL requires them.

Uniform variables can be declared in both vertex shaders and fragment shaders. If a uniform variable of the same name is declared in both the vertex and the fragment shader, it gets the same value in both places. It is an error to declare uniform variables of the same name but different type in shaders that are supposed to be compiled into the same program and work together. A uniform variable which has been set in a program object retains its value within that program object until it is set to a different vale, or until the program object is destroyed.

Add animation to your shader by including the uniform `time` and the code in Listing 17 in your program from the previous exercise. Experiment with using the uniform variable `time` in your shaders to scale or translate your vertices with `sin(time)` in the vertex shader and to set the color in the fragment shader to something that depends on `sin(time)`. Try both!

Now it is time to start using transformation matrices in our vertex shader. A transformation matrix can be declared as a 4×4 array in C++, often declared as a one-dimensional array of 16 values. The memory layout is typically the same – a sequence of 16 numbers. Thus, a uniform matrix in GLSL like

```glsl
uniform mat4 T;
```

can be set from C++ as shown in Listing 18. Note that the name of the variable in the C++ code may be different from its name in the GLSL code. The call to `glUniform()` performs a copy of the value of one variable in C++ to a *different* variable in GLSL. There is no direct association by name between them. The only connection is through the address that is returned from `glGetUniformLocation()` and the copy made by the call to `glUniform`.

The individual elements of a matrix in OpenGL are specified in a column by column fashion (column-major), not row by row (see also matrixindexing.pdf).

```
std::array<GLfloat, 16> matT = {
  1.0f, 0.0f, 0.0f, 0.0f,
  0.0f, 1.0f, 0.0f, 0.0f,
  0.0f, 0.0f, 1.0f, 0.0f,
  0.0f, 0.0f, 0.0f, 1.0f
};

GLint locationT = glGetUniformLocation(myShader.id(), "T");
glUseProgram(myShader.id());  // Activate the shader to set its variables
glUniformMatrix4fv(locationT, 1, GL_FALSE, matT.data());  // Copy the value
```

Listing 18: Setting that uniform 4×4 matrix variable from C++.

Therefore, the list of initialization values in C++ for the variable `T` in Listing 18 should be read as the *transpose* of the matrix. The element at row i, column j in the OpenGL matrix, where i and j are both in the range 0 to 3, is element `T[4*j+i]` in the 16-element array in C++. The rightmost column, where the translation components should be, consists of elements T[12], T[13], T[14], and T[15], that is the last row of initialization values in the C++ code.

The function copying a 4×4 matrix from a C++ array to a uniform variable in a shader is `glUniformMatrix4fv()`. Compared to `glUniform1f()`, the `glUniformMatrix4fv()` in Listing 18 has two extra parameters.

The *count*, 1, means that this is a single matrix, and that it is being copied to a single `mat4` variable in the shader. If a shader needs many different matrices, it is a good idea to create one large array in C++ with $16N$ numbers and copy them all at once to a `mat4` array in GLSL. For the purpose of these labs, you will only need a few matrices which you should keep in separate variables both in C++ and in GLSL.

The *transpose flag*, `GL_FALSE`, tells OpenGL to copy the matrix as it is specified. Substituting `GL_TRUE` would transpose the matrix during upload. We do not recommend using this flag as it can be confusing between your code and how OpenGL handles matrices.

Once a matrix is available in the vertex shader, coordinates can be multiplied with it instead of using them directly for drawing, as shown here. Note that GLSL has built-in support for matrix multiplication.

```
// Transform (x,y,z) vertex coordinates with the 4x4 matrix T
gl_Position = T * vec4(Position, 1.0);
```

To experiment with matrix-based transformations, declare and initialize a matrix in your C++ program, declare a corresponding `uniform mat4 T;` in your vertex shader, send the matrix to the shader by the method presented in Listing 18 and use it for transformation in the vertex shader according to the listing above.

Edit the values for your matrix to perform different kinds of transformations: uniform scaling in x and y, non-uniform scaling, translation, and finally a rotation around the z axis. Make sure you understand what happens, and why. Note where the translation coefficients should be specified in your matrix in the C++ code. A few different transformations are shown in Figure 4.

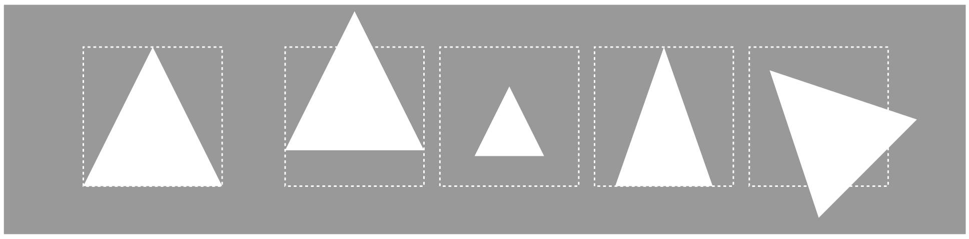

Figure 4: The original triangle, and versions that are translated in y, scaled uniformly by 0.5, rotated 45° around y, and rotated 45° around z.

Create a second matrix in your C++ code and in your vertex shader in GLSL. Set one to a translation and the second to a rotation around the z axis. Transform your vertices by multiplying both matrices with the vertex coordinate vector in your shader. Change the order of the multiplication and observe the difference it makes.

Even though the GPU can multiply matrices quickly, keep in mind that operations in the vertex shader are performed once for every vertex. If a scene contains millions of vertices, then the matrix multiplication is done for each vertex. To avoid this, the matrices can be multiplied already on the CPU and the result sent to the shader as a single uniform matrix.

A function to multiply two matrices could be implemented as shown in Listing 19. The function takes the two matrices as arguments and returns the result. There are several ways to implement the matrix multiplication function. One way is to do it using loops over row and column indices. Another solution is to compute each element of the output matrix explicitly and write sixteen very similar lines to compute the entire output matrix

In the following, you will be asked to create a couple of functions. You have to decide where to put them in your project. You can for example place them in the same file as your main program, before the `main()` function (recommended). This is simple, but it makes the file a more difficult to navigate. You can also add them to the file `Utilities.cpp` to keep things more tidy in your main program. In that case, remember to also add the function prototypes to `Utilities.hpp`, or your program won't compile. Or, you can create a new C++ source file and corresponding header file and put your code there. Do not to forget to add the source and header file names to CMakeLists.txt in that case so they are included in the project.

Implement a C++ function like the one in Listing 19 to multiply two matrices.

```cpp
// Multiply 4x4 matrices m1 and m2 and return the result
std::array<float, 16> mat4mult(const std::array<float, 16>& m1, const std::array<float, 16>& m2) {
    std::array<float, 16> result;

    // Your code goes here: compute and set each element of result, e.g.:
    // result[0] = m1[0] * m2[0] + m1[4] * m2[1] + m1[8] * m2[2] + m1[12] * m2[3];
    // etc. for the remaining 15 elements.
    return result;
}
```

Listing 19: A function stub for matrix multiplication in C++

Test your matrix multiplication function. It will be used a lot in the upcoming exercises so make sure it generates the correct result.

You can test the `mat4mult()` function in several ways. One way is to compare it to what GLSL does by supplying your shader with a composite matrix built from a rotation and a translation and making sure it generates the same result as when multiplying them in the same order in the GLSL shader code.

Another approach is to print both input matrices and the result and then verify that it is correct by using pen and paper. Test at least two different combinations, like a translation and a rotation performed in different order. Listing 20 contains a function to print matrices using `printf()` from `<cstdio>` for formatting.

```cpp
// Print the elements of a matrix m
void mat4print(const std::array<float, 16>& m) {
    printf("Matrix:\n");
    printf("%6.2f %6.2f %6.2f %6.2f\n", m[0], m[4], m[8], m[12]);
    printf("%6.2f %6.2f %6.2f %6.2f\n", m[1], m[5], m[9], m[13]);
    printf("%6.2f %6.2f %6.2f %6.2f\n", m[2], m[6], m[10], m[14]);
    printf("%6.2f %6.2f %6.2f %6.2f\n", m[3], m[7], m[11], m[15]);
    printf("\n");
}
```

Listing 20: A function to print the elements of an OpenGL matrix

Write a C++ function `std::array<float, 16> mat4identity()` to return an identity matrix (a 16-element array of `float` values).

Write three C++ functions returning a rotation matrix for the three principal axes x, y, and z. Each function should take the rotation angle as argument and return the rotation matrix around the respective axis. Note that the trigonometric functions in C++ like `sin()` and `cos()` expect radians not degrees! Also add functions for scaling and translation.

Suggested function signatures are in Listing 21. Note that these functions, just like `mat4identity()`, are designed to return a new matrix.

```cpp
std::array<float, 16> mat4identity();
std::array<float, 16> mat4rotx(float angle);
std::array<float, 16> mat4roty(float angle);
std::array<float, 16> mat4rotz(float angle);
std::array<float, 16> mat4scale(float scale);
std::array<float, 16> mat4translate(float x, float y, float z);
```

Listing 21: Function prototypes for creation of transformation matrices

Use the rotation matrices to rotate the triangle around z. Make sure the result is as expected. Continue by rotating the triangle around the x or y axis.

Add animation to your scene by making the rotation matrix dependent on time. The C++ function `double glfwGetTime()`, which is a part of the GLFW library, returns the amount of time that has passed since the program was started in seconds. Use it to rotate your triangle around y at a speed of one revolution per 2π seconds (hint: use `M_PI`). In order to use an animated matrix in a shader, you need to update the C++ matrix and copy it to the shader program object for each frame, so both of those operations should be performed inside the rendering loop.

When drawing a triangle in 3D, sometimes the front and sometimes the backside is visible. Without any hints from lighting or perspective, it is impossible to make out which is which. Experiment with hiding the back face of the triangle by using `glEnable(GL_CULL_FACE)` before the rendering loop. This is called *back face culling*.

Typically, many 3D objects have a closed surface and the triangle backsides are never visible. Culling can be used to avoid drawing the insides. It is performed on the GPU after the vertex shader and culled faces are never processed by the fragment shader.

Enable back face culling. This should make the rotating triangle disappear when the backside is facing the camera.

A useful method for seeing all faces of a triangle mesh is to draw the object as a *wireframe*. OpenGL can draw triangles either as filled surfaces, outlines, or points at the vertices. You can set this with `glPolygonMode()`. The first argument must be `GL_FRONT_AND_BACK` for historical reasons. The second argument sets the mode (`GL_FILL`, `GL_LINE`, `GL_POINT`). By default, faces are painted as filled surfaces, which is equivalent to calling `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)`. A wireframe rendering is achieved by `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`.

Change the polygon rendering mode for all faces so that they are drawn as outlines. This can help you see all faces in a more complex object regardless of orientation and drawing order, and it might come in handy if you need to find errors in your vertex or index arrays. It is also a common way of showing the triangle structure of a large mesh during interactive editing in 3D design software.

One way to distinguish between front and back faces is to draw the front faces as filled polygons and the back faces as outlines. To do this, you need to render your triangle twice: once with `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)` and `glCullFace(GL_BACK)` and a second time with `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)` and `glCullFace(GL_FRONT)`. Try it!

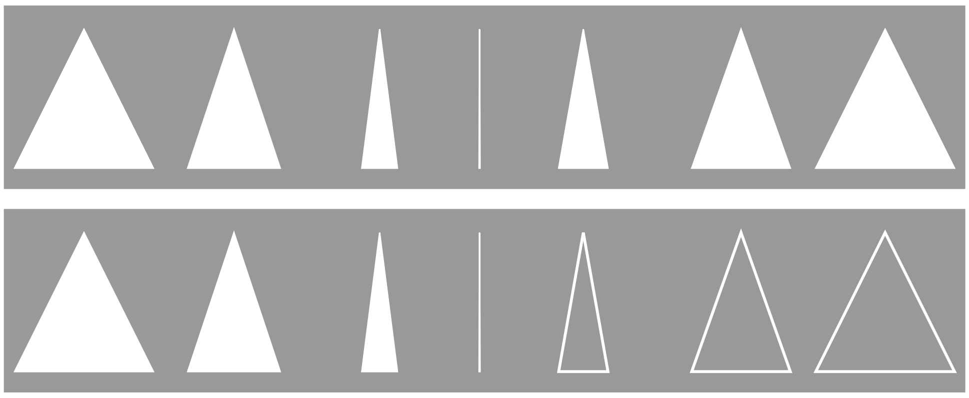

Figure 5: The triangle rotating around y, with both sides filled (top) and with the front face filled and the back face drawn as outlines (bottom).

2.2.3 More triangles – a cube

A cube is a basic object which can be built out of individual triangles by splitting each cube face along its diagonal. However, care has to be taken with the order of the vertices when specifying triangles in the index array. In OpenGL, the "front face" of a triangle has a *positive winding order*, that is vertices are drawn counter-clock wise when viewed from above as depicted in Figure 6.

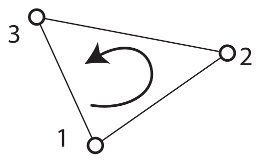

Figure 6: Positive winding order when seen from the front of a triangle.

Create vertex arrays to make a cube with corners at ±1 in each of the (x, y, z) axes, and a different color for each vertex. Use bright and contrasting colors for clarity. Pay particular attention to the winding order. Sketch it on paper and put the vertex indices next to each corner of the cube.

To draw more than one triangle, the size of the vertex and index arrays needs to be adjusted before calling `glBufferData()`. Similarly, the number of drawn vertices in `glDrawElements()` must be changed.

Test your cube by drawing it with an animated rotation. Perform a fixed rotation around one axis and an animated rotation around a different axis so you get to see all the faces. You might also want to scale the cube to a smaller size so it doesn't cover your entire window. To see whether you got the winding order correct, enable back face culling with `glEnable(GL_CULL_FACE)` and render the front faces as filled polygons. This makes any triangles with the wrong winding order stand out more clearly as holes in the surface.

Create an animation of your cube similar to the motion of a planet in orbit around a sun. The transformation should be a composite matrix made from four transformations:

1. View rotation V: rotate the entire scene around x using a small fixed angle to see it from slightly above.
2. Orbit rotation R_{orbit}: rotate slowly around y to simulate the orbit of the planet.
3. Translation T: translate along either x or z to move the planet from the origin.
4. Spin rotation R_{spin}: Rotate more quickly around y to simulate the planet's spin around its own axis.

The final transformation should be equal to $VR_{\text{orbit}}TR_{\text{spin}}$. See Figure 7 for an illustration. Apply these transformations to the cube by sending only one matrix to the shader (hint: perform the matrix multiplications in C++).

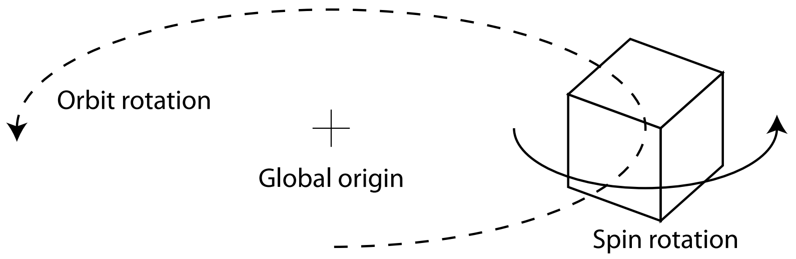

Figure 7: A cube spinning around its local y axis (solid arrow) and also rotating around y in an orbit around the world origin (dashed arrow).

2.3 Face colors, object abstraction, normals, and shading

2.3.1 Preparations

This exercise requires some preparations before the lab. Otherwise you will not be able to finish the exercise in the 4 hours of the lab session.

- Read Section 2.3, look at the code, and get an overview of what you are supposed to do during the lab session.
- Think about how to split your cube to have a different color for each face instead of a different color for each vertex. This requires you to split each vertex into three new vertices (Section 2.3.2). Write down the vertex and index arrays for a cube with six square faces and different constant colors for each face.

2.3.2 Face colors

In OpenGL, all geometry attributes like color are specified per vertex. Put differently, if a vertex is shared between multiple triangles, the triangles have the same color in this corner. Therefore you need to split the corner vertices into several vertices to color the faces of the cube differently. The split vertices have the same positions but different colors. In addition, all vertices belonging to the same triangle must have the same color.

Create a cube with each face having a different, constant color. You need to split each vertex into three to make 24 vertices in total. Test your new object to make sure it displays correctly.

2.3.3 Object abstraction

Hard coding the vertex arrays directly in code for objects like the triangle or the cube is one way to specify objects. Another option is to generate vertices programmatically. Regular shapes like spheres and cylinders consisting of many triangles can easily be created using loops and some code (Exercises 2.3 and 2.4). Finally, triangle meshes can also be created in some 3D modeling software and then imported as vertex lists (Exercise 2.5).

By using a class we can encapsulate and structure all the details for generating, storing, and rendering geometry. The `TriangleSoup` class holds all the information necessary for a vertex array, see Listing 22. `TriangleSoup` uses `std::vector<>` for the vertex and index arrays. In contrast to `std::array<>` with a fixed size as used before, `std::vector<>` can grow and shrink at runtime. For teaching purposes, this class keeps the vertex and index data on the CPU around. This could be optimized and the data released after uploading it to the GPU for maximum efficiency.

To hide the internal data representation all data members are declared `private`. Such a class is called *opaque* and it is often a good idea to design a class that way. It allows the implementation details of the class to be changed without affecting any code that uses

```cpp
class TriangleSoup {
private:
    GLuint vao_;                        // Vertex array object, the main handle for geometry
    int nverts_;                        // Number of vertices in the vertex array
    int ntris_;                         // Number of triangles in the index array (may be zero)
    GLuint vertexbuffer_;               // Buffer ID to bind to GL_ARRAY_BUFFER
    GLuint indexbuffer_;                // Buffer ID to bind to GL_ELEMENT_ARRAY_BUFFER
    std::vector<GLfloat> vertexarray_;  // Vertex array on interleaved format: x y z nx ny nz s t
    std::vector<GLuint> indexarray_;    // Element index array
};
```

Listing 22: Data members of the class TriangleSoup holding the vertex array information.

the class, as long as the functionality stays the same and the public methods retain their signatures (their name, arguments and return type).

The class contains a few methods as well, see Listing 23. The complete class declaration is in the file `TriangleSoup.hpp`.

```cpp
/* Constructor: initialize a triangleSoup object to all zeros */
TriangleSoup();

/* Destructor: clean up allocated data in a triangleSoup object */
~TriangleSoup();

/* Clean up allocated data in a triangleSoup object */
void clean();

/* Create a very simple demo mesh with a single triangle */
void createTriangle();

/* Create a simple box geometry */
void createBox(float xsize, float ysize, float zsize);

/* Create a sphere (approximated by polygon segments) */
void createSphere(float radius, int segments);

/* Load geometry from an OBJ file */
void readOBJ(const std::string& filename);

/* Print data from a triangleSoup object, for debugging purposes */
void print();

/* Print information about a triangleSoup object (stats and extents) */
void printInfo();

/* Render the geometry in a triangleSoup object */
void render();
```

Listing 23: Methods of the class TriangleSoup to create and render triangle meshes.

Given a variable `TriangleSoup mySoup`, the call `mySoup.createSphere(1.0f, 20)` will generate an approximation of a sphere of radius 1.0, using 20 polygon segments from pole to pole and 40 polygon segments around the equator to approximate the smooth shape. Put the statement `#include "TriangleSoup.hpp"` at the top of your main program and add the file `TriangleSoup.cpp` to your project if not already included.

To see how a `TriangleSoup` object is intended to work have a look at the implementation of the method `TriangleSoup::createTriangle()`. It is a very simple example of a mesh object with only a single triangle in it similar to the first example object you used.

Draw a single triangle using a `TriangleSoup` object and `createTriangle()` as shown in Listing 24. You should no longer specify any vertex and index arrays in your main() function. Note how this simplifies the code in `main()`. Apparently, the `TriangleSoup` object was a good and much needed abstraction.

```
// --- Put this before your rendering loop
// Generate a triangle
TriangleSoup myShape;
myShape.createTriangle();

// --- Put this in the rendering loop
// Draw the triangle
myShape.render();
```

Listing 24: Creating, initializing and drawing a TriangleSoup object.

2.3.4 Interleaved arrays

As you can see, there is only one vertex array buffer in `TriangleSoup`, which we use to store both vertex coordinates, normals, and texture coordinates. This is the recommended way of specifying geometry in OpenGL and it is called *interleaved vertex arrays*. The layout of the array is specified by making appropriate calls to `glVertexAttribPointer()`, where we specify where in the array the different attributes are located. It removes the need for multiple vertex buffer objects, and during rendering it is generally more efficient to have all data for one vertex stored in nearby memory locations.

We define a single data array in C++, but use three calls to `glVertexAttribPointer()` with different parameters to copy data of different types to the corresponding vertex attributes (see `TriangleSoup::createSphere()`). The particular data layout we will be using with 8 floats per vertex according to Listing 22 is a very common format for OpenGL programming. Vertex coordinates (x, y, z), per-vertex normals (n_x, n_y, n_z), and per-vertex texture coordinates (s, t) can be considered the bare minimum of what is required to describe a triangle mesh.

Compare and identify the similarities between `TriangleSoup::createTriangle()` and `TriangleSoup::createSphere()`. The way the coordinates are determined is quite different but the general structure is the same.

Read the methods `TriangleSoup::createSphere()` and `TriangleSoup::render()` and make sure you understand what they do and how they do it.

If you did not make any changes to your shader code, the triangle from `TriangleSoup` should appear blue (RGB $0, 0, 1$). This is because the vertex attribute with index 1 is now not the vertex *color*, but the vertex *normal*, which is $(0, 0, 1)$ for all vertices of the triangle. OpenGL does not know what kind of data you send to the vertex attributes. If you incorrectly send one kind of data to an attribute that expects another kind of data, the program will still run. However, you will get unexpected results.

For now, you can use the normals to set the vertex colors. This will be fixed later on. When rendering into the regular display buffer (or framebuffer), color values will be forced to lie in the range of $[0, 1]$. This is called *clamping* and there is even a GLSL function named `clamp()`.

Use the method `TriangleSoup::createSphere()` to create a sphere and draw it. First draw the sphere as a wireframe model (triangles rendered as lines) and vary the number of segments to see how it works. Then render it as a solid object (filled triangles). Draw it with an animated rotation around the y axis to make all sides visible over time.

2.3.5 Normals and shading

Since `TriangleSoup` uses interleaved vertex arrays with multiple vertex attributes, the shaders need to be adjusted. Add the corresponding layout locations and `in` and `out` specifications. The naming we recommend is shown in Listing 25. Remember that for each `out` variable in your vertex shader you also need to declare a corresponding `in` variable in your fragment shader with the same name and type.

```glsl
layout(location=0) in vec3 Position;
layout(location=1) in vec3 Normal;
layout(location=2) in vec2 TexCoord;

out vec3 interpolatedNormal;
out vec2 st;

void main() {
  gl_Position = vec4(Position, 1.0); // Special, required output
  interpolatedNormal = Normal; // Will be interpolated across the triangle
  st = TexCoord; // Will also be interpolated across the triangle
}
```

Listing 25: (GLSL) A vertex shader to use all three attributes in a `TriangleSoup` object.

Write a vertex shader according to Listing 25 and write a fragment shader to render a sphere with a color that varies smoothly across the surface. Try using both the three-dimensional normals (vertex attribute 1, `interpolatedNormal` in the fragment shader) or the two-dimensional texture coordinates (vertex attribute 2, `st` in the fragment shader) to set the RGB color for the rendering, just to see that you get the additional attributes through to your shader program. The colors will show you the values of the normals and texture coordinates. Make sure you understand why the colors come out the way they do.

The sphere surface has a color which changes across the surface, but it looks flat. For a more realistic surface, we need to add some kind of approximation of how light interacts with a smooth, curved surface.

Change your shader to use the second vertex attribute of the sphere as normal direction. Use the vertex shader input `Normal` to compute the fragment shader input `interpolatedNormal` and set all color components to the z component of the normal direction by the statement shown in Listing 26. Use the scene with the sphere rotating around the y axis. Run your program and watch what happens.

```glsl
in vec3 interpolatedNormal;

out vec4 finalcolor;

void main() {
  finalcolor = vec4(vec3(interpolatedNormal.z), 1.0);
}
```

Listing 26: (GLSL) A fragment shader to compute simple, normal-dependent shading.

2.3.6 Transforming the normals

The sphere looks as if it was lit from one side and the light source rotating with it. This is because we transform the vertex coordinates, but not the normals. To make a correct object transformation, we need to transform the normals as well. When transforming normals, you need to keep two things in mind:

- Normals are *directions*, not positions, so they should never be translated.
- Normals should always have *unit length* and may need re-normalization after transformations.

In some cases we can directly use the same transformation matrix for transforming vertices and normals, but we need to use some caution. To avoid translations, either multiply the transformation matrix with the four-dimensional vector $(N_x, N_y, N_z, 0.0)$, or multiply the `vec3` (N_x, N_y, N_z) with the upper left 3×3 part of the 4×4 matrix. If `T` is a `mat4`, that 3×3 sub-matrix can be created by writing `mat3(T)` in your vertex shader. To remove the effects of scaling, normalize the vector after transformation. This can be done by the GLSL function `normalize()`. See example code in Listing 27. Keep in mind that you need to normalize the normal again in the fragment shader since the interpolation across the triangle may affect its length.

Note that this transformation of normals works properly only for rotation, reflection, and uniform scaling. To handle fully general transformations including non-uniform scaling and skewing, you need to compute a separate matrix for transformation of normals. More specifically, you should use the *inverse transpose* of the upper 3×3 part of the vertex transformation matrix to transform normals and then normalize your result. Rotation and uniform scaling matrices are *orthogonal matrices*, and as you may recall from linear algebra, the inverse of an orthogonal matrix is equal to its transpose except for a scale factor. Therefore, the inverse transpose of an orthogonal matrix M is a possibly scaled version of the matrix itself: $(M^{-1})^T = kM, k \in \mathbf{R}$.

```
layout(location = 1) in vec3 Normal;

out vec3 interpolatedNormal;

uniform mat4 T;

vec3 transformedNormal = mat3(T) * Normal;
interpolatedNormal = normalize(transformedNormal);
```

Listing 27: (GLSL) Vertex shader statements to transform normals.

Transform your normals along with the object before you compute the fragment color from the normal direction and see that it makes a visual difference.

With the current shader, illumination appears to come from positive z, which is the direction of the viewer. This is a useful special case, but we want to be able to simulate light from any direction. You can create a light direction vector as a local `vec3` variable in the fragment shader code. (In a more general shader program, the direction and color of the light would typically be a uniform variable sent to the vertex shader that is transformed and sent to the fragment shader.) As you know from the lectures and the course book, diffuse illumination can then be computed by a scalar product between the light direction and the normal. Scalar products are computed by the function `dot()` in GLSL, see Listing 28.

```
in vec3 interpolatedNormal;

out vec4 finalcolor;

void main() {
    vec3 lightDirection = vec3(1.0, 1.0, 1.0);
    float shading = dot(interpolatedNormal, lightDirection);
    shading = max(0.0, shading);  // Clamp negative values to 0.0
    finalcolor = vec4(vec3(shading), 1.0);
}
```

Listing 28: (GLSL) A fragment shader to with an arbitrary light direction.

Simulate lighting from a direction different from positive z. Try a few different directions to make sure your code works the way it should. You may want to disable the rotation of the object to see more clearly what you are doing.

Disable the animated rotation of the object and instead use the transformation matrix to rotate only the light direction. You can rotate the light vector either in the vertex shader or in the fragment shader. It is more efficient to do the transformation in the vertex shader and send a transformed light direction vector to the fragment shader. This saves a considerable amount of work for the GPU. Declare an `out` variable of type `vec3` in the vertex shader and an `in` variable of the same type and name in the fragment shader. This will make OpenGL send the value of that variable from the vertex shader to the fragment shader and interpolate it across the surface of the triangle if it varies between vertices.

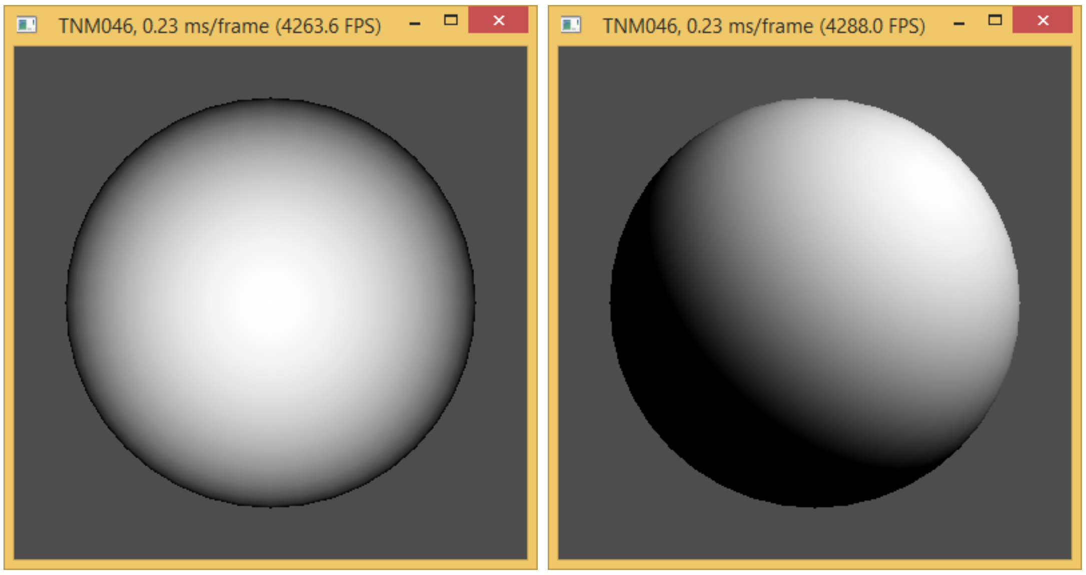
TNM046, 0.23 ms/frame (4263.6 FPS)
TNM046, 0.23 ms/frame (4288.0 FPS)

Figure 8: The sphere lit straight from the front and from a different angle.

The sphere is now lit, as shown in Figure 8. The *Phong illumination model* is a basic illumination model to create an appearance of shiny objects. You have everything you need to perform the necessary computations in the vertex shader: the surface normal, the light direction, and the view direction. The direction of the viewer is always $(0, 0, 1)$ in screen coordinates, which is the coordinate system at the output from the vertex shader. Note also that there is a convenient function `reflect()` in GLSL if you don't want to write your own code to compute the reflection vector. Refer to OpenGL documentation to see how it works.

Listing 29 shows the GLSL code to compute the Phong illumination model. Refrain from combining `Id` and `kd`, and similar, as they represent different physical properties which are independent of each other. `Id` represents the intensity and color of the *light source*, and `kd` is the reflective color of the *surface*. Note that the variable names were chosen to be similar to the classic Phong notation. Your variables probably have different names. Note also that the code is not a complete shader. Figure 9 shows the Phong model applied to a sphere. In Exercise 2.5, you will use textures to modulate the surface color `kd`.

Add specular reflection to your computations. Keep the object still and rotate only the light source. Use white for the specular color and a slightly darker color for the diffuse color to make the specular highlight clearly visible. Make sure you get the specular highlight in the correct position and that you understand why it appears where it does. Where is the specular highlight when the light source is behind the sphere?

```
// vec3 L is the light direction
// vec3 V is the view direction - (0,0,1) in view space
// vec3 N is the normal
// vec3 R is the computed reflection direction
// float n is the shininess'' parameter
// vec3 ka is the ambient reflection color
// vec3 Ia is the ambient illumination color
// vec3 kd is the diffuse surface reflection color
// vec3 Id is the diffuse illumination color
// vec3 ks is the specular surface reflection color
// vec3 Is is the specular illumination color

// This assumes that N, L and V are normalized.
vec3 R = 2.0 * dot(N, L) * N - L;   // Could also have used the function reflect()
float dotNL = max(dot(N, L), 0.0);  // If negative, set to zero
float dotRV = max(dot(R, V), 0.0);
if (dotNL == 0.0) {
  dotRV = 0.0;  // Do not show highlight on the dark side
}
vec3 shadedcolor = Ia * ka + Id * kd * dotNL + Is * ks * pow(dotRV, n);
finalcolor = vec4(shadedcolor, 1.0);
```

Listing 29: (GLSL) Fragment shader code to compute the Phong illumination model.

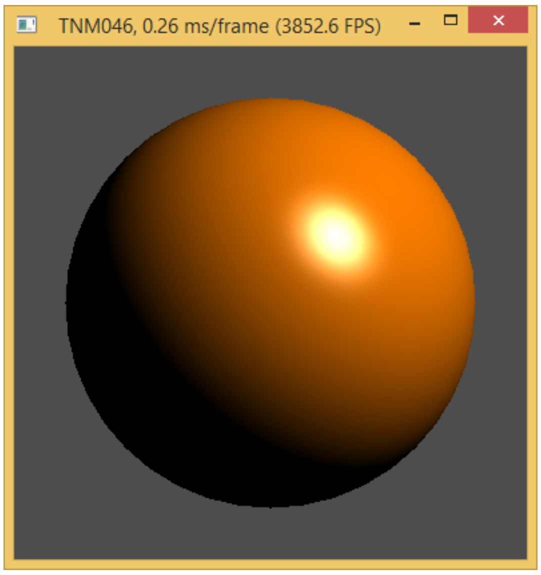

Figure 9: The sphere rendered with a Phong illumination model.

You may wonder why the illumination calculations are performed in the fragment shader and not in the vertex shader to save some computations. The reason is interpolation. Since the diffuse color and, in particular, the specular highlight can change quite dramatically across the surface of a single triangle, interpolating the illumination of the vertices would loose such details. Computing the illumination first and then interpolating the colors is called *Gouraud shading* whereas interpolating the normals and then computing the illumination is called *Phong shading*. Do not confuse this with the *Phong illumination model*!

(Optional exercise) Move the shading computations to the vertex shader. Compute a final color at each vertex instead of at each pixel and send the color through an `out`/`in` variable pair to the fragment shader. The fragment shader becomes very simple – you just need to set the output color to the value of a single `in` variable.

Try different reflections, diffuse and specular, to see the difference and understand how interpolated per-vertex shading works. Increase the number of triangles to make the sphere look better when you render it with a strong specular reflection. You may have to use rather a lot of triangles.

With the shading computations in the fragment shader, use a shiny reflection for a stationary sphere with a rotating light source, or, if you want, a rotating sphere with a stationary light source. Then try using a fairly low number of segments for the call to `createSphere()`, and see how few triangles you can get away with for the sphere before it no longer looks smooth.

Computing lighting once for each fragment instead of once for each vertex means more computations, but it does give a better looking result, particularly for specular reflections. It also enables more advanced illumination models for more interesting surface appearances. However, at least *some* computations can often be performed per vertex instead of per fragment, so don't blindly push everything to the fragment shader. There is often a balance to be struck in terms of what to perform in the vertex and the fragment shader stages, respectively. This is similar to the decision regarding what to do in on the CPU and what to do in GPU shaders where different circumstances might call for different solutions.

2.4 Camera and perspective

2.4.1 Preparations

This exercise requires some preparations before the lab. Otherwise you will not be able to finish the exercise in the 4 hours of the lab session.

- Read Section 2.4, look at the code, and get an overview of what you are supposed to do during the lab session.
- Take special care to understand how perspective projection works and how a perspective projection matrix can be constructed.

2.4.2 The OpenGL camera model

Until now we have been using a parallel, also called *orthographic*, projection to draw our 3D objects. While this type of projection is used for example in computer aided design and technical drawings, a *perspective projection* is typically used for rendering 3D scenes.

Usually, the transformations are split into a *modelview matrix* and a *projection matrix*. The modelview matrix transforms object coordinates to world coordinates (model transformation) and world coordinates to camera coordinates (view transformation). The projection matrix is responsible for the perspective projection and usually does not translate or rotate the camera. It is a good idea to keep those two matrices separate. In some cases like when dealing with light sources it might even be advisable to split the modelview matrix into the model and view matrices.

> Create a vertex shader that accepts two 4×4 matrices as uniform variables, one named `MV` for the modelview matrix and one named `P` for the projection matrix. Make the necessary changes in your main program to create and upload both matrices to your shader program. Let the P matrix remain an identity matrix for now, but change your vertex shader to transform vertices in a manner that involves both matrices.

The modelview matrix is mostly built from many different transformation matrices that are multiplied together. It is typically different for each object and it may change over time whenever the camera or the object moves. The projection matrix on the other hand is mostly created once and then left unchanged. You change it only when you want to change the field of view or the aspect ratio of the window, or if the distance to the scene changes so radically that you need to change the near and far clipping distances.

Generally speaking, you should also use the *normal matrix* to transform normals. For these lab exercises, we will use the upper 3×3 part of the modelview matrix. This is OK as long as we take care not to introduce any non-uniform scaling that changes the relative orientations of surfaces in our objects. However, in general you should create a normal matrix, set it to the inverse transpose of the modelview matrix without the translations, upload it as a separate uniform matrix to the vertex shader, and use that matrix for transforming normals.

2.4.3 A box object

To demonstrate the perspective projection we use a box since the perspective foreshortening and the vanishing point are more apparent than for a sphere.

The method `TriangleSoup::createBox()` in `TriangleSoup.cpp` is not implemented. Use your code from previous exercises to implement the method. The method `TriangleSoup::createTriangle()` provides a template for you to start. Pay attention to the normals. The texture coordinates should also be included in your array, but set to zero for now. They will be used in Exercise 2.5.

Test your object by animating a cube. Draw the cube both as a wireframe model then as a solid object. You can render it with a simple diffuse shading as if the illumination came from positive z or you can treat the normals as vertex colors for a more colorful display. In this case the three sides of the cube that face towards the negative (x, y, z) directions will be black, but you can change the fragment shader to set the final RGB color to `interpolatedNormal * 0.5 + 0.5` to make all color components positive.

Try some different parameters for `createBox()` to make sure that your code can create boxes with different sizes and proportions.

Using your new `createBox()` method, create a box with sizes $x = 0.2$, $y = 0.2$, and $z = 1.0$. Render it with an animated rotation around the y axis.

Look at your animated display of the elongated box. Due to the orthographic projection the result looks flat even though the object is clearly wider in one direction and changes its on-screen appearance from a square to an oblong rectangle during the animated rotation.

2.4.4 The perspective projection matrix

The perspective matrix P is commonly defined as shown in Figure 10. It is very similar to the matrix in the textbook by Gortler. The only difference being the sign for the final computed depth, the z coordinate.

$$P = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} & -\frac{2z_{near}z_{far}}{z_{far}-z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\text{where } f = \cot(vfov/2)$$

Figure 10: The suggested matrix to create by `mat4perspective()`.

Create a projection matrix that performs a suitable perspective projection for this object. Write a function `mat4perspective()` matching the declaration in Listing 30 to complement your previous matrix functions from Exercise 2.2. The function should return the perspective matrix from Figure 10.

```
// create and return a perspective matrix
//
// vfov is the vertical field of view (in the y direction)
// aspect is the aspect ratio of the viewport (width/height)
// znear is the distance to the near clip plane (znear > 0)
// zfar is the distance to the far clip plane (zfar > znear)
std::array<float, 16> mat4perspective(float vfov, float aspect, float znear, float zfar);
```

Listing 30: Function prototype for creating a perspective matrix.

The perspective matrix in Figure 10 includes near and far clipping distances, z_{near} and z_{far}. Just like the x and y coordinates in OpenGL are restricted to the range $[-1, 1]$, the z coordinate is also restricted to the same range. Anything falling outside of that range after the transformations is *clipped*, that is discarded from view. The primary reason for this is that we need to set a fixed range for the depth buffer. With the matrix according to Figure 10, z coordinates at $-z_{near}$ end up at $z = -1$ after the transformation and perspective division whereas z coordinates at $-z_{far}$ end up at $z = 1$.

Because of the near and far clip planes, it is necessary to take into consideration how large the scene is and how far away it is from the camera when you create a perspective projection matrix. The effect of the perspective division is that the depth buffer has the highest precision close to the near clipping plane (or camera) and the precision gets worse as the ratio z_{far}/z_{near} increases.

Setting the near clipping distance very close to zero or the far clipping distance to large values can therefore cause precision issues with the depth buffer. You can set the clipping distances with some margin to cover your entire, but try to use reasonable values. Using values like $z_{near} = 0.0001$ and $z_{far} = 10^6$ should be avoided.

2.4.5 Positioning the camera

The standard form of the perspective matrix, see Figure 10, places the eye or camera at the origin. However, all the objects drawn so far are also centered on the origin. Thus, the camera will be inside the objects if you apply the modelview matrix as before. To be able to look at the objects from outside, the camera needs to move back (or the scene moves away from the camera depending on how you look at it). To achieve this, a translation in $-z$ needs to be multiplied from the left to the existing modelview matrix. This camera transformation is known as the *view matrix*.

Add a view translation of -3 units in z to your transformations. Create a perspective matrix with $vfov = \pi/4$, $aspect = 1$, $z_{Near} = 0.1$, and $z_{Far} = 100.0$. Render the box with the perspective transformation. The visual result should look similar to Figure 11.

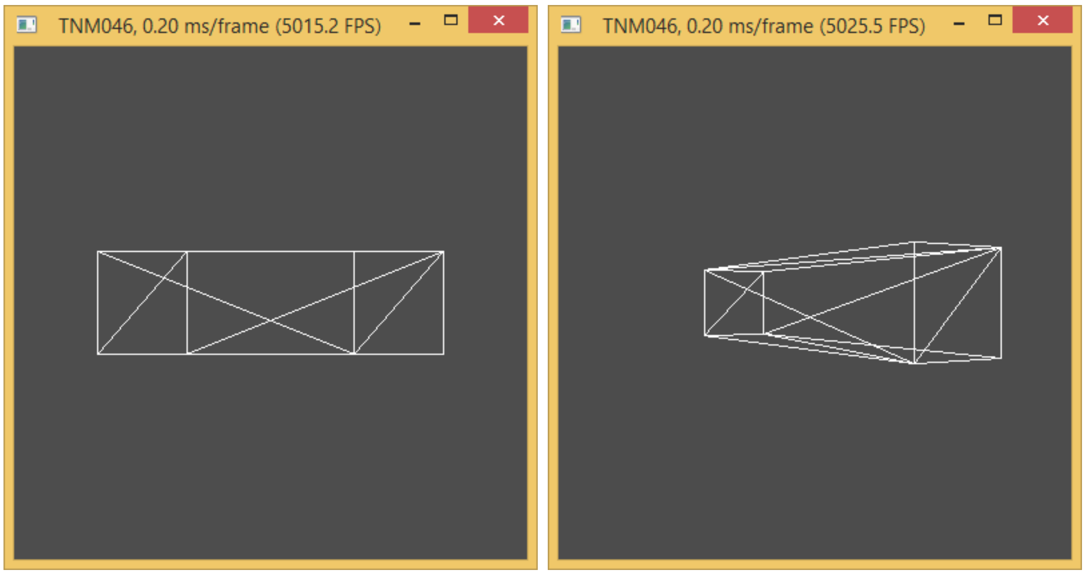
TNM046, 0.20 ms/frame (5015.2 FPS)
TNM046, 0.20 ms/frame (5025.5 FPS)

Figure 11: The box rotated around y. Left: orthographic view. Right: perspective view.

Add a small fixed rotation around x as part of the view matrix to your modelview matrix **after** the view translation to view the object from a different and more interesting angle. Also translate the object in x, y, and z directions and see what happens.

Depending on your choice of parameters for the view translation and the perspective matrix, the perspective effect may be quite subtle or it may be too strong. Speaking in terms of field of view (FOV) for a small window on an ordinary monitor, the field of view larger should be smaller than 60° ($\pi/3$ radians). A small FOV will be similar to an orthographic projection and create a very zoomed-in view of the scene.

Experiment with different fields of view (FOV) for your projection. Adjust the view translation after you change the field of view to make the object fit the window.

Try simulating the field of view for a real camera, like the camera that may be built into your phone or your laptop. Experiment by holding your finger in front of the real camera while looking at the viewfinder and make a rough estimation of the FOV angle. How does the FOV change if you use zoom on the camera?

What is the real field of view for the window on the monitor from where you are sitting? What would be the field of view for a window covering the entire monitor? Such questions are important for virtual reality (VR) displays and for games with a first-person view, but matter less for ordinary use of 3D graphics. Humans seem to be quite happy with interpreting the 3D content in an image from a virtual camera with its own field of view rather than looking through a simulated window into a 3D world behind the monitor.

2.5 Textures, models from file, interaction

2.5.1 Preparations

This exercise requires some preparations before the lab. Otherwise you will not be able to finish the exercise in the 4 hours of the lab session.

- Read Section 2.5, look at the code, and get an overview of what you are supposed to do during the lab session.
- Think about how to map a texture to the box you created in the previous exercise and prepare the texture coordinates for the vertex array before you arrive at the lab session. There are many different ways to texture map a box, pick one. If you want to, feel free to draw your own texture image and save it as an *uncompressed* TGA file.

2.5.2 Textures

So far, only constant surface colors are used in the illumination equations. In 3D computer graphics surfaces are typically drawn with a *texture* to add visual detail to otherwise plain surfaces. OpenGL has supported them since the beginning since textures are so important in 3D graphics. Textures are associated with an OpenGL *texture object*, which is created similar to vertex array objects and shader program objects:

- Create a texture object.
- Bind the texture object (activate it).
- Configure the parameters for the texture
- Load the texture data (the pixels of the image) into CPU memory.
- Transfer the texture data to the GPU memory.
- Unbind the texture object.

When rendering an object with the texture, you need to

- Bind the texture object.
- Activate a shader program that uses a texture.
- Tell the shader program which texture you want to use.
- Draw your geometry.
- Unbind the texture object.

```
#version 330
uniform sampler2D tex;  // A uniform variable to identify the texture

in vec2 st;             // Interpolated texture coords, sent from vertex shader

void main() {
    finalcolor = texture(tex, st);  // Use the texture to set the surface color
}
```

Listing 31: (GLSL) Texture lookup in a very simple fragment shader.

An OpenGL texture object is represented in C++ by a (`GLuint`) integer value like vertex array and shader program objects. The integer is an ID that is associated with a data structure that resides on the GPU. In GLSL shaders, textures are accessed using *samplers* and `texture()` functions in combination with *texture coordinates*. Listing 31 shows the use of a texture in the fragment shader.

To load textures into CPU memory, the most common method is to read image pixels from files. In C++, you need to use external libraries to import different image file formats or write your own file reader. Modern compressed file formats like PNG and JPEG are more complicated to read and typically require additional libraries. The class `Texture` therefore supports loading uncompressed TGA images. You can find example TGA files in the lab material, but you can create your own in any image editing program that can save files as uncompressed TGA. Note that `Texture` can only handle RGB and RGBA color formats.

Read the code in `Texture.hpp` and `Texture.cpp`. You don't have to concern yourself with the private methods to deal with the TGA file format, but look at the code in the method `Texture::createTexture()` to see how it works, and see that its structure corresponds to the itemized list at the beginning of Example 2.5.2.

The use of a `Texture` object and its OpenGL textures is shown in Listing 32. The shader variable `uniform sampler2D tex;` in Listing 31 deserves more attention. A `sampler2D` is formally an integer ID, but it is considered an *opaque handle* in GLSL, a value that is not supposed to be changed. It is only to be used as the first argument to the GLSL function `texture()`.

Note that `glUniform1i()` is called with 0 and **not** the ID of the texture object. The 0 here refers to the *texture unit* the texture object is attached to. This is important, and it is a common cause of errors among beginner OpenGL programmers.

The default texture unit is 0. To activate a different texture unit to bind additional textures you need to call `glActiveTexture()` before binding the texture with `glBindTexture()`. If you want to, you can add a line with `glActiveTexture(GL_TEXTURE0)` immediately before the line where you call `glBindTexture()` to make it clear that the default texture unit 0 is used. Note the use of `GL_TEXTURE0` here. Nowadays GPUs support up to 16 texture units (`GL_TEXTURE0, GL_TEXTURE1, ..., GL_TEXTURE15`). The corresponding sampler in the shader however uses just the corresponding numerical values $0, \ldots, 15$.

```cpp
// --- Put this before the rendering loop
// Locate the sampler2D uniform in the shader program
GLint locationTex = glGetUniformLocation(myShader.id(), "tex");
// Generate one texture object with data from a TGA file
Texture myTexture;
myTexture.createTexture("textures/earth.tga");

// --- Put this in the rendering loop
// Draw the TriangleSoup object mySphere
// with a shader program that uses a texture

glBindTexture(GL_TEXTURE_2D, myTexture.id());

glUseProgram(myShader.id());
glUniform1i(locationTex, 0);
mySphere.render();

// restore previous state (no texture, no shader)
glBindTexture(GL_TEXTURE_2D, 0);
glUseProgram(0);
```

Listing 32: Example use of a texture in OpenGL

Before transferring the texture data from CPU memory to the GPU there is some setup necessary using `glTexParamter*()`. These functions are used to set various parameters of the texture like automatic resizing for close-ups (magnification) and distant views (minification) as well as texture wrapping along the edges. Uploading the texture data is done by calling `glTexImage2D()`. Listing 33 shows how the texture parameters are set and the data is uploaded in `Texture::createTexture()`.

```cpp
void Texture::createTexture(const std::string& filename) {
    image_ = loadUncompressedTGA(filename);

    if (image_.data.empty()) {
        return;
    }

    if (textureID_ == 0) {
        glGenTextures(1, &textureID_);  // Create the texture ID if it does not exist
    }

    glBindTexture(GL_TEXTURE_2D, textureID_);
    // Set parameters to determine how the texture is resized
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // Set parameters to determine how the texture wraps at edges
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // Read the texture data from file and upload it to the GPU
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image_.width, image_.height, 0, image_.type,
                 GL_UNSIGNED_BYTE, image_.data.data());

    glEnable(GL_TEXTURE_2D);  // Required for glGenerateMipmap() to work
    glGenerateMipmap(GL_TEXTURE_2D);

    // Image data was copied to the GPU, release contents of the std::vector
    // When using clear() the std::vector would still hold on to the memory.
    image_.data = std::vector<GLubyte>();
}
```

Listing 33: Creation of a texture in OpenGL

The sphere object you rendered in Exercise 2.3 has 2D texture coordinates (s, t) defined in the third vertex attribute. Include all three vertex attributes (0, 1, and 2) in the layout for `in` variables in your vertex shader. Give them appropriate names and render the sphere where the color is determined by a texture lookup in the fragment shader.

Texture coordinates need to be interpolated across each triangle, which means they should be sent as an `out` variable from the vertex shader to an `in` variable in the fragment shader. Keep in mind that in the `TriangleSoup` structure, vertex coordinates and normals are `vec3` attributes, but texture coordinates are `vec2`.

Modify `TriangleSoup::createBox()` from Exercise 2.4 to include texture coordinates and render a textured cube with a texture. There are several options for how to map a texture onto a cube. Do you want the same map on all faces or should different faces of the cube correspond to different parts of the texture map? Decide for yourself which mapping you want and implement it.

Write a fragment shader computing diffuse and specular lighting like in Exercise 2.3. Modify the shader so that the diffuse color of the surface `kd` is now taken from a texture. The specular color `ks` should remain white.

2.5.3 Geometry from files

The most common method to render arbitrary objects is to import models that have been generated in other software packages and saved to files in an appropriate exchange format. The class `TriangleSoup` provides a method `readOBJ()` to load a `TriangleSoup` structure from a Wavefront OBJ file, a text-based format representing triangle meshes. The code in `TriangleSoup::readOBJ()` supports the basic functionality of the file format. Have a brief look at `TriangleSoup::readOBJ()` before you use it. Also take a look at the contents of an OBJ file.

Load a Wavefront OBJ file using `TriangleSoup::readOBJ()`. Display it first as a wireframe model to show its polygon structure. Then display it with the surface color taken from a texture. Finally, display it with diffuse and specular lighting and a texture.

The lab material contains some models as OBJ files with matching textures in the `meshes/` and `textures/` folders. You can also use a 3D modeling software and create your own models by exporting them as Wavefront OBJ. Note that `TriangleSoup::readOBJ()` only handles OBJ files with triangles. Polygons with more than three vertices are not supported. Many exporters will provide you with an option to export the mesh only using triangles.

2.5.4 Depth test

You may see some artifacts when rendering more complex objects caused by some parts of the model occluding other parts. This happens since multiple triangles will be rendered into the same pixels and the order is not correct. OpenGL has built-in support to address this issue: the *depth test*. The depth test can be enabled with `glEnable(GL_DEPTH_TEST)`. If enabled, the depth test is performed for each pixel after the fragment shader and ensures that only fragments are visible which are closer to the camera than anything drawn before. Thus, the occluded parts of the object stay hidden and intersections between different objects are handled correctly, see Figure 12.

Load a Wavefront OBJ file and render the object with a sphere circling in an orbit around it.

You need to create two separate TriangleSoup objects for this and render them with different modelview matrices. Set up the matrix and and copy it to the shader then render the first object. Change the matrix and copy it again to the shader and render the second object.

Make sure that the occlusion looks right, both within an object and between objects. Try rendering a sphere that is large enough and close enough to intersect the object it orbits. The depth test should handle intersections as well as occlusions properly.

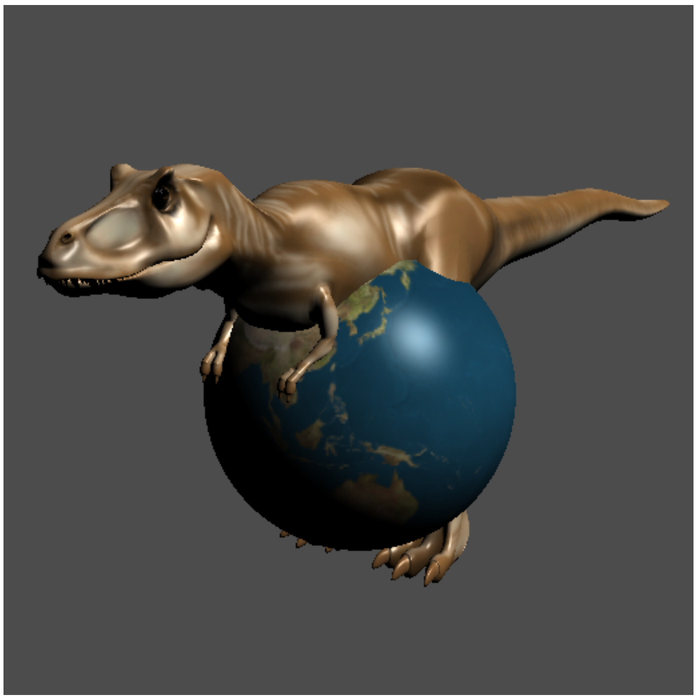

Figure 12: Two intersecting objects rendered using the depth test.

2.5.5 Interaction

So far, the objects in the scene have been animated but is not possible to *interact* with the scene and camera directly. Interaction is an integral part of real-time computer graphics and enables the user to modify scene parameters without the need to edit code and recompile the program. The most common way to interact with a 3D scene is to manipulate the camera, that is moving and rotating it.

User interaction through GLFW is supported using either keyboard, mouse, or game controller/joystick. GLFW handles only the low level input, which means that for each frame we can access information on each keyboard key, the position of the mouse and its button state, and the position of all sticks and buttons of a game controller.

In order to set a rotation of an object or of the camera, we first need to save its current rotation and update it in response user input. If we are using the keyboard or a joystick, we want to keep the rotation speed independent of the frame rate and therefore we also need to consider the time of the most recent update of the rotation. When using the mouse, the most natural interaction is to make the rotation depend on how far the mouse was moved and in which direction. To be able to determine how much the mouse has moved, we need to keep track of where the mouse was in the last frame. And to know whether we just moved the mouse or clicked and dragged with a mouse button down, we need to also remember the up/down state of the mouse buttons from the last frame.

To keep track of these states, the lab project already contains two classes `MouseRotator` and `KeyRotator` (`Rotator.hpp` and `Rotator.cpp`). The classes keep track of the mouse state and current time, and by calling `poll()` for each of these objects, some private data members are updated. These class members are used to perform rotations in spherical coordinates `phi` and `theta`. They are accessible via the member functions `phi()` and `theta()`, respectively. The angle `phi` is updated when you press the left or right arrow key or click and drag the mouse sideways. Likewise, `theta` is updated in response to up and down arrow keys and mouse dragging up and down.

Study the classes `KeyRotator` and `MouseRotator`. Make sure you understand what they do and how they do it. An example usage is shown in Listing 34.

```cpp
#include "Rotator.hpp"

// --- Put this before the rendering loop, but after the window is opened.
KeyRotator myKeyRotator(window);
MouseRotator myMouseRotator(window);

// --- Put this in the rendering loop
myKeyRotator.poll();
// Create a rotation matrix that depends on myKeyRotator.phi and myKeyRotator.theta
...
myMouseRotator.poll();
// Create rotation matrix that depends on myMouseRotator.phi and myMouseRotator.theta
...
```

Listing 34: Example use of MouseRotator and KeyRotator.

Create a program that displays a `TriangleSoup` object with a Phong illumination model. Use a MouseRotator object to rotate it interactively with the mouse. The most suitable way of interpreting the rotation angles `phi` and `theta` is to use `phi` to make a matrix R_z that rotates around z and `theta` to create a rotation matrix R_x around x. Then the two rotations can be combined in the order $R_x R_z$.

2.5.6 Everything at once

In the final part of this lab series, create a single program including all the different parts: transformations, perspective projection, complex objects, illumination, texturing, and interaction.

Add a `uniform vec3 lightDirection;` to your vertex shader, transform it with a separate matrix that is also a `uniform` variable and send the transformed variable to your fragment shader as a `vec3`. Use that light direction to compute the Phong illumination model.

Use a `KeyRotator` to be able to rotate the light source and the object separately. Decide if you want to rotate the light in world space or if you want to position the light source relative to the object and construct your matrices accordingly.

This concludes the lab series. You have learned to use OpenGL for modern 3D graphics programming. This should give you a good foundation and understanding to continue on your own. There are also several upcoming courses in the MT program where you will use these skills and build upon them.