Lab Assignments
Computer Graphics DT3025, HT2016

Martin Magnusson and Daniel Ricão Canelhas
October 31, 2016

1 Getting started with OpenGL

In this first lab you will first learn how to communicate with the graphics hardware, and then learn the foundations of how to put 3D objects where you want them on the screen; first with some manual labour (to see some of what goes on under the surface), and then with tools that simplify life to some extent.

Learning goals

- Introduction to OpenGL:
  - creating a window (using GLFW),
  - using buffer objects to send data to the GPU,
  - drawing a triangle,
  - using basic vertex and fragment shaders,
  - passing parameters to the shaders with uniform variables.
- 3D geometry and algebra:
  - model, view, and perspective transformations.

1.1 Hello GPU

In this lab you will create a **GL context**, using GLEW and a window to draw in. GLEW is an OpenGL loading library that loads all the available OpenGL extensions and make them accessible to the developer. OpenGL itself is a cross-platform API for graphics programming, and has no way of creating a window on its own. This is due to windows being handled quite differently on different operating systems, such as Windows, Unix, Android, etc. Therefore we'll use a helper library called GLFW to create a window and create a context in which we can call OpenGL functions. There are alternatives to both GLEW and GLFW, though we discourage using them for these labs, but recommend that you investigate the available options for your future projects.

Compiling your code

Firstly, download the code from blackboard and unzip it in a location of choice.

If you're using Windows find the "visual_studio" directory and open the `all_labs.sln` solution file in Visual Studio. Right-click on the solution tree and select build to compile the lab skeletons. The executable files will be located in a build directory under the respective lab directory. To see the printed output from your program in this task, it may be necessary to use the windows command prompt ("cmd.exe") or powershell ("powershell.exe") to run your programs. Type `c:` at the command prompt to change to the correct drive (on

the lab computers), then `cd` to change directory and `dir` or `ls` (depending on which shell you use) to get a directory listing. To run the executable, simply type its name at the command prompt; e.g

```
C:\Users\Julia\source\computer_graphics\labs\Lab1\>lab1-1.exe
```

If you use Linux, run `cmake` followed by `make` from your terminal emulator of choice in a subdirectory relative to where you downloaded the source files; e.g.,

```
~/code/computer_graphics/labs/Lab1$ mkdir build
~/code/computer_graphics/labs/Lab1$ cd build
~/code/computer_graphics/labs/Lab1/build$ cmake ..
~/code/computer_graphics/labs/Lab1/build$ make
```

The executable should appear in the build directory and can be executed from there.

```
~/code/computer_graphics/labs/Lab1/build$ ./lab1-1
```

Task

The lab skeleton code uses GLEW to create a window and a rendering context. Your task is to query the graphics driver of the computer to check the capabilities of the graphics card in your computer.

✏ Use the `glGetString` function to query the graphics card for the **renderer** used and the **OpenGL version** that is supported. (Refer to the OpenGL documentation to see how `glGetString` is used). Provide this information in your report.

✏ For the renderer, try to find out what the hardware specifications are and what limits you have in terms of video RAM, number of processors, clock frequency, memory bandwidth, etc.

✏ For the supported OpenGL version, list which features are missing, relative to the latest version of the specification.

1.2 Your first triangle

Now you will also use the graphics card to actually draw something.

In recent versions of the OpenGL specification (and Vulkan) it might seem like there is a lot of work needed just to get a single triangle to show up on the screen. This is true, and compared to Legacy OpenGL it seems like things were simpler before! However, as GPUs have become more sophisticated and with more capabilities, graphics APIs have evolved to make fewer assumptions about how the graphics hardware will be used. This has resulted in more flexible interfaces but where the developer (you) has the burden of specifying what is to be done and how, in detail. In the so-called *next generation* of OpenGL, i.e. **Vulkan**, the developer is given an even greater degree of low-level access to the hardware and flexibility, at the cost of having to check for errors and undefined behaviour by herself.

In the skeleton code you find two code blocks referring to a "vertex shader" and a "fragment shader". These are the programs that will run on the GPU and are compiled at run-time by the graphics driver. They are written in the

OpenGL Shading Language or "GLSL" for short. The syntax is structurally similar to C but has some notable differences. For an overview, you are encouraged to look at this website.

Tasks

1. Begin by defining the vertices of your triangle as a regular array of floats. Look at the definition of the vertex shader in the template code, it should guide you as to the dimensionality of your vertices. Keep the x and y values in the range $[-1, 1]$ to remain within the limits of the screen.
2. Create a vertex array object (VAO). The OpenGL wiki on vertex specification[1] describes how this is done.
3. Create and initialize a vertex buffer object (VBO) with the float array.
4. *Enable* the vertex attribute related to the position of the vertices (that is, the vertex attributes you just stored in your VBO).
5. Configure the VAO by specifying the layout of the data inside the buffer (using one of the functions listed below).
6. Finally tell the GPU to draw the triangle data you have provided.

There are a few different ways in which to solve this task. If you found in Task 1.1 that your GPU is compatible with OpenGL 4.5, you can use direct state access functions (`glCreateBuffers` and `glCreateVertexArrays`), which don't require binding objects before use. Otherwise, you will have to first generate (`glGen*`) and then bind (`glBind*`) your VBO and VAO to the current context before making any changes to what is then the current object. Read and understand the documentation about the following functions and solve the task in the way you consider best.

- VAOs

```
glGenVertexArrays(...);
glBindVertexArray(...);
glCreateVertexArrays(...);
glEnableVertexAttribArray(...);
glEnableVertexArrayAttrib(...);
glVertexArrayVertexBuffer(...);
glVertexAttribPointer(...);
glVertexArrayAttribFormat(...);
```

- VBOs

```
glGenBuffers(...);
glBindBuffer(...);
glCreateBuffers(...);
glBufferData(...);
glNamedBufferData(...);
glBufferStorage(...);
glNamedBufferStorage(...);
```

Lastly, in the main event loop, issue the appropriate `glDraw*()` command. Don't forget to bind your VAO before drawing. When you have solved the task

[1] https://www.opengl.org/wiki/Vertex_Specification

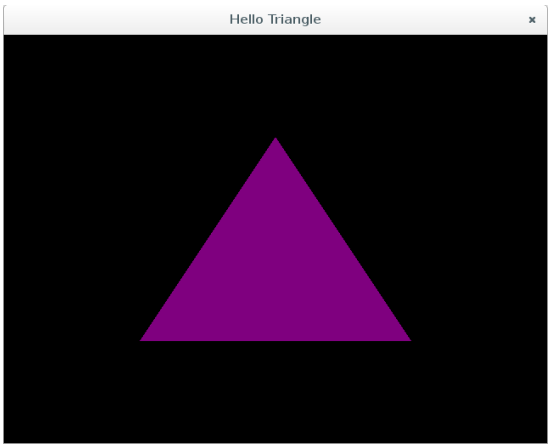

Figure 1: A purple triangle drawn in OpenGL in Task 1.2.

you should see something similar to Fig.1, though your vertices may be placed differently.

In addition to the OpenGL documentation and the wiki, the site learnopengl.com is also be a helpful resource.

✏ Describe in your report which functions you have used, and why. Also answer the following questions.

✏ • Which dimensions in the screen space do your x, y and z coordinates map to?

✏ • What are the limits for z and what happens if you exceed them?

1.3 Introduction to shaders

In this task you will pass information between shaders[2] and write a little more GLSL code. Vertex shaders, as the name implies, are executed on each vertex. Fragment shaders are executed on elements that might contribute to the final color of each pixel, in parallel.

Copy your solution of the previous task (that is, the code for specifying the triangle, your VBO and VAO, and the drawing command) into the source code for lab1-3, to draw the triangle. Note that the shaders are now loaded from a text file instead of being part of the definition of the cpp file. This has the advantage of not requiring you to recompile your CPU program to make changes to the shader code run on the GPU.

Tasks

1. Create a global struct or class on the CPU side for your shader programs (to make it cleaner and easier to implement the next step).
2. Edit the `key_callback` function to re-load the shaders when the "R" key is pressed.
3. Modify the vertex shader to declare an `out vec3 position` variable and write the position to it.

[2] https://www.opengl.org/wiki/shader

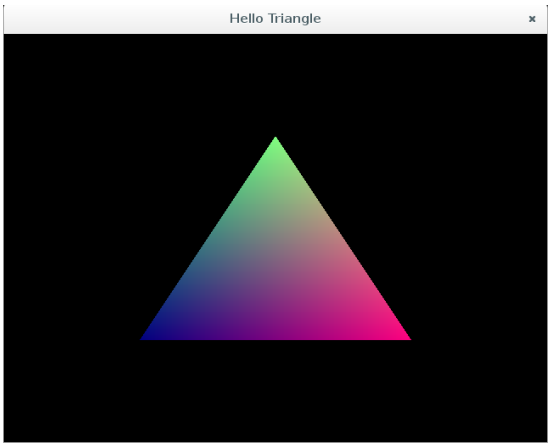

Figure 2: A multicolored triangle drawn in OpenGL in Task 1.3.

4. Modify the fragment shader to declare an `in vec3 position` variable and use this as output for the color from the fragment shader, so that the colour varies with the position of the pixels.

You may notice that some parts of your triangle are black. Fix this by modifying the output from the vertex shader. You should see something similar to Figure 2 when done.

✏ If the vertex shader runs only on the vertices, how is it that the fragment shader gets a different value for all the points on the triangle? Is there a way to control this behaviour? (See the section on interpolation qualifiers in the OpenGL wiki.[3])

1.4 Passing parameters to shader programs

You might be wondering how computer games and other interactive applications work, if the visible content is to respond to the user's inputs. Suppose we want the triangle to move when we press the arrow keys on the keyboard. Do we change the coordinates of the triangle and upload new values to the GPU at each iteration of the event loop? This would work for a single triangle, but imagine the case in which we have millions of triangles. In practice, we will keep the geometry definition as it is, and send the desired *transformation* as a parameter to the GPU instead. This is done using `uniform` variables;[4] that is, global variables that are the same for all executions of the shader program.

Tasks

1. See the API manual on `glUniform` for how to set the value of a uniform and how to get its location.
2. Declare a `uniform vec2 position_offset` in the vertex shader and update it using the keyboard. Have the left and right keys move the triangle horizontally while up and down changes the vertical position.
3. Declare a `uniform float modifier` parameter in your fragment shader and make the scroll wheel (or some other key) through different colors/-patterns depending on the value of the modifier.

[3] https://www.opengl.org/wiki/Type_Qualifier_(GLSL)#Interpolation_qualifiers
[4] https://www.opengl.org/wiki/Uniform_(GLSL)

1.5 3D geometry

So far, we have mostly been disregarding the three-dimensional nature of the data and concerned ourselves only with the two dimensions that relate to the screen. This changes now. We have provided the geometric definition of a 20-sided die below, also known as an icosahedron. You may copy this definition or use any of the other platonic solids of your choosing. Note that every surface has to be defined as set of triangles, so for a square face, you will require 2 triangles, for a pentagon you will need 3 triangles, etc.

```
float t = (1.0f + sqrtf(5.0f))*0.25f;
float points[] = {
// An icosahedron has 12 vertices
    -0.5,  t,  0,
     0.5,  t,  0,
    -0.5, -t,  0,
     0.5, -t,  0,
     0, -0.5,  t,
     0,  0.5,  t,
     0, -0.5, -t,
     0,  0.5, -t,
     t,  0, -0.5,
     t,  0,  0.5,
    -t,  0, -0.5,
    -t,  0,  0.5
  };
```

As you can see, although the shape has 20 faces, it only has 12 vertices, since every adjacent face shares vertices with its neighbours. To define all 20 faces, we therefore use an index array to specify which three vertices will compose each triangle, as follows:

```
unsigned short faces[]= {
// ... and 20 triangular faces, defined by these vertex indices:
    0, 11, 5,
    0, 5, 1,
    0, 1, 7,
    0, 7, 10,
    0, 10, 11,

    1, 5, 9,
    5, 11, 4,
    11, 10, 2,
    10, 7, 6,
    7, 1, 8,

    3, 9, 4,
    3, 4, 2,
    3, 2, 6,
    3, 6, 8,
    3, 8, 9,

    4, 9, 5,
    2, 4, 11,
    6, 2, 10,
    8, 6, 7,
    9, 8, 1
  };
```

In the previous tasks, we didn't have an index array, and it isn't strictly necessary if we repeat the vertices so that every three vertices in the vertex buffer

results in one triangle. In this case it would mean sending 60 floats to the GPU instead of 12 floats plus 60 shorts. Which results in fewer bytes?

To check, you can run the following program.

```
#include <iostream>

int main()
{
  std::cout << "alternative one: "
      << sizeof(float)*12 + sizeof(unsigned short)*60
      << std::endl;

  std::cout << "alternative two: "
      << sizeof(float)*60
      << std::endl;
}
```

The indices defined above must also be sent to a GPU buffer before they can be drawn. This information is not something that is stored for each vertex, so we can not use a vertex buffer object for this. Instead, we have to use another construct called an *element* buffer object (EBO).[5] Using these indexes to specify what to draw, instead of using a "naked" vertex buffer object, is called indexed drawing.

Tasks

1. Define your geometry.
2. Create and initialize a vertex buffer object (VBO) with the float array.
3. Create and initialize an element buffer object (EBO) with the index array.
4. Create a vertex array object (VAO).
5. Enable the vertex attribute related to the position of the vertices.
6. Bind your VAO.
7. Bind your EBO.
8. Issue the appropriate `glDraw*` command. This will be different from the previous tasks, since we now do indexed drawing; that is, we no longer draw using the vertex attribute array directly, but using the element buffer object.
9. Modify your fragment shader to display a color related to the z-value of the position forwarded from the vertex shader (note that you may need to flip the sign to get a positive value).

Depending on the geometry and how you defined you fragment shader, you will have something on your screen that looks like a flattened version of the 3D object with z-values, i.e. *depth*, mapped to different color values. An example is given in Figure 3.

[5] https://www.opengl.org/wiki/Vertex_Specification#Index_buffers

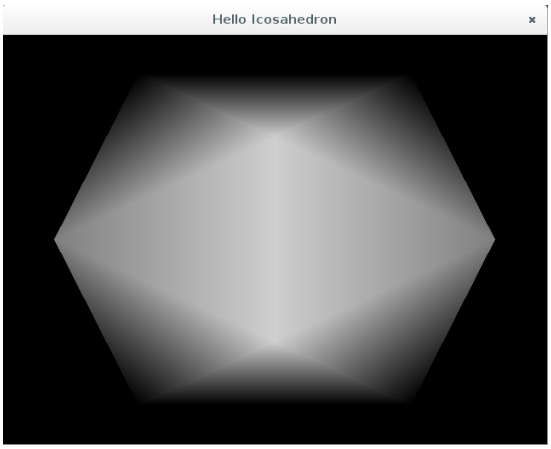

Figure 3: 3D icosahedron, displayed with gray-scale color as a linear function of z.

1.6 Model, view & projection transformations

Until now we have not bothered with the position of the geometry in the *scene*, but used the object's local coordinates directly. We have also done nothing to specify the type of camera we are using, or determined the pose of the camera. In this section we will deal with the these issues by defining the pose of the model in the world, the pose of the camera, and the camera's projective properties.

Any rigid-body transformation, i.e. one that changes the position and orientation of the geometry, can be described as the combined effects of rotating and translating the geometry.

$$\mathbf{v}_b = \mathbf{t} + \mathbf{R}\mathbf{v}_a \tag{1}$$

Where $\mathbf{v}_a$ and $\mathbf{v}_b$ are, respectively, the initial and final positions of a vertex, $\mathbf{t}$ is a translation vector and $\mathbf{R}$ is a 3×3 rotation matrix.

The operations of translating and rotating the geometry can be combined into a single matrix-vector multiplication by writing $\mathbf{v}$ in homogeneous coordinates and combining the rotation and translation into a 4×4 matrix

$$\boldsymbol{T} = \begin{pmatrix} R_{1,1} & R_{1,2} & R_{1,3} & t_1 \\ R_{2,1} & R_{2,2} & R_{2,3} & t_2 \\ R_{3,1} & R_{3,2} & R_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2}$$

Model transformations and View transformations are of the rigid-body sort, with the difference that model transformations are applied directly to the geometry and view transformations are applied inversely to the geometry. This means that moving the camera to the left is equivalent to moving the world to the right by the same distance and that a clockwise rotation of the camera is a counter-clockwise rotation of the world, etc.

The projection matrix is another 4×4 matrix that flattens the geometry onto the image plane, whilst scaling far away objects down to look smaller. In the lecture notes you have information on how to construct such a matrix.

In this task you will perform the necessary transformations by hand in order to fully see what goes on "under the hood." In later tasks, you will use library functions instead, which make these operations less tedious.

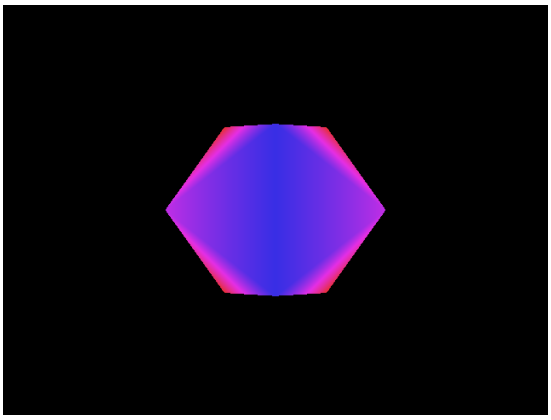

Figure 4: 3D icosahedron, with view and projection transformations applied, displayed with hue varying along the object's z coordinate.

Tasks

1. Define a `GLfloat projectionMatrix[4][4]` with parameters `near = 1.0` and `far = 100.0`.
2. Create a function that multiplies two 4×4 matrices.
3. Define a `GLfloat modelMatrix[4][4]` as the product of two transformation matrices, one applying a rotation around the x axis, the other applying a rotation around the y axis. Vary angles of rotation around each axis using input from the user (mouse or keyboard).
4. Create a function that inverts a rigid body transformation matrix. (Hint: Rotation matrices are orthonormal.)
5. Define a `GLfloat viewMatrix[4][4]` that moves the camera along the z axis by 2 units.
6. Define a `GLfloat modelViewProjectionMatrix[4][4]` as the product of projection, view and model matrices using your matrix multiplication function, and send it as a `uniform` variable to your vertex shader.
7. In your vertex shader, apply the transformation to the vertex (note that OpenGL handles `mat4-vec4` multiplications natively using the "$*$" operator).
8. Use the HSV (hue-saturation-value) to RGB function provided in the fragment shader to vary the hue along the z axis of the object.

The result should look something like Figure 4.

Questions

- Why is the order of multiplication important?
- What would happen if you multiplied the matrices in the reverse order?

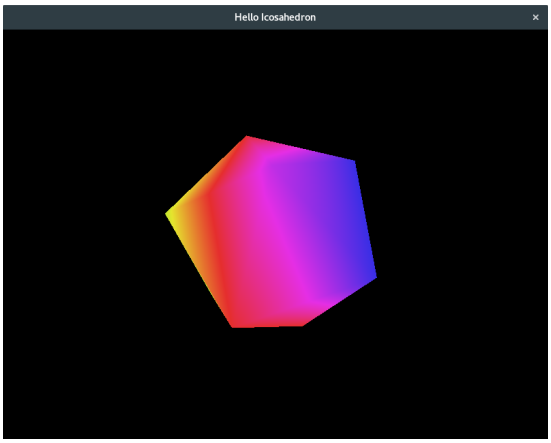

Figure 5: 3D icosahedron, displayed with hue varying along the object's z coordinate, geometric transformations applied using GLM.

1.7 Linear algebra isn't fun

As you perform more and more advanced math operations you will find it increasingly tedious to implement them by hand. To make the code more readable and easier to use, it is therefore recommended to use GLM (OpenGL Mathematics). This is an open-source headers-only C++ implementation of the GLSL primitives ('vec2', 'vec3', etc.). You will find it at http://glm.g-truc.net/.

Tasks

1. Look at the GLM documentation for how to define a perspective projection, translation, inversion and rotation.
2. Replace your own functions with those provided by GLM and your array-based matrices with `glm::mat4` objects.
3. Upload your resulting modelViewProjection matrix to the shader as a `uniform`. Since your data is now stored as a `glm::mat` instead of a plain C-style array, you cannot just pass the address of your matrix as you might have done earlier. Check the GLM documentation for `glm::value_ptr` to see how to get a pointer to the matrix data.

✏ • So far the `glDepthFunc` has been set to `GL_LESS`. What other options can you set for this parameter?

✏ • Is the original choice appropriate? Why? What can happen otherwise?

We can barely say that we are doing 3D graphics now, but geometry-wise it's mostly a matter of scale from here on out. We will increase the geometric detail later on, but the following labs will deal with the more complex issue of light and materials.

Assessment criteria

This lab is graded as pass or fail (G/U). To pass, the following criteria must be fulfilled.

- All questions marked in the tasks have been answered in the report.
- The complete code for the tasks is submitted together with the written report, and the code compiles and runs, as detailed in the task descriptions.
- The report is well-structured and written so that it is easily understandable by a reader not fully familiar with the student's code.
- All students in a group (pair) can also orally describe and argue for the methods used in solving each programming task.

Deadline

The deadline is **November 8** at noon. If you hand in your report after this, it will still be graded, but only when there is time.