Introduction to Computer Graphics
Practical Session 4 – OpenGL

In this 4th practical session you will learn the basics of OpenGL and glsl programming. The following exercises are not graded. Download *p4.zip* from *moodle*, it contains everything you need for Practical 4.

1 OpenGL Pipeline

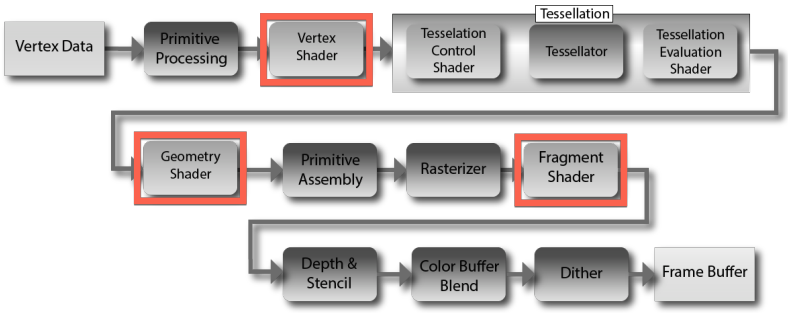


Figure 1: An illustration of the programmable OpenGL pipeline. In this practical session we will only consider *vertex*, *geometry* and *fragment* shaders. *Tassellation* shaders will be the subject of week 6.

You have learned about the OpenGL pipeline during the lecture. A high-level overiew of that pipeline is presented in Figure 1. When rendering primitives such as triangles, they have to undergo several processing stages before they finally appear on the screen. In this practical we will discuss the two most important stages: the *vertex shader* and the *fragment shader*. Former shader processes and transforms the vertices of a primitive while latter shader is executed for every fragment (a fragment is a small segment of a rasterized primitive, you can think of it as a pixel of the primitive in screen space). Note that the vertex and fragments shaders are always required for your application to run.

Further we will also introduce the *geometry shader* stage. It allows to change the primitives by injecting new vertices and by changing the primitive type (e.g., from triangle to lines). The geometry shader is optional and can be omitted.

The different shader stages can be seen as small individual sub-programs. They are written in *glsl* (OpenGL Shading Language). All shaders are loaded, compiled, and linked during runtime. This is done by passing a string that contains all the code of the shader program to OpenGL and by calling the corresponding build functions[1]. It is common to store shaders in .glsl files and use helper functions to load file contents into strings.

Note that you can always find a complete documentation of all OpenGL and glsl functionality either in the quick reference guide or in the OpenGL man pages.

[1] If you want to know how we do it, look at *external/opengp/include/OpenGP/GL/shader_helpers.h*

2 Shaders

Let us jump directly into shader development. We will temporarily ignore how the information is passed from the C++ application to the shader programs as this will be the subject of Section 5. In Section 2.1, we will briefly summarize the basic syntax and capabilities of a glsl program. Then we will see examples of vertex (Section 2.2), fragment (Section 2.3) and finally geometry (Section 2.4) shaders.

2.1 GLSL Basics (syntax, math, predefined)

The GL shading language has a syntax that will be *very similar*[2] to C/C++ . Differently from C/C++, GLSL has default types to represent vector and matrices like *vec3* and *mat4*, as well as functions to operate on them, e.g., *transpose( ), inverse( )*. The example below quickly illustrates most of the functionality that you will need for this class.

```
/// Example of a GLSL function
/// in:    parameter passed by copy
/// inout: parameter passed by reference
/// @note the "-" and transpose() could be removed ;)
mat2 rotateme(inout vec2 vec, in int alpha_deg){
    /// Conversion
    float alpha = radians(-alpha_deg);

    /// Build 2D rotation matrix
    mat2 rotmat;
    rotmat[0][0] = cos(alpha);
    rotmat[0][1] = sin(alpha);
    rotmat[1][0] = -rotmat[0][1];
    rotmat[1][1] = rotmat[0][0];

    /// Apply transformation
    vec = transpose(rotmat) * vec;

    return rotmat;
}

/// A global array
vec2 myvecs[2] = vec2[](
    vec2(1.0,0.0),
    vec2(0.0,1.0));

/// Shader entry point
void main(){
    /// Example initialization
    int alpha = 30;
    float vx = 1.0;
    float vy = 1.0;
    vec2 vec = vec2(vx,vy);
    // vec2 vec(vx,vy); ///< WRONG: note the difference from C!!!

    /// Example function call
    mat2 rotmat = rotateme(vec, alpha);

    /// Call on some array data
    rotateme(myvecs[0], alpha);
    rotateme(myvecs[1], alpha);
}
```

Built-in shader variables something to remember is that each shader has its built-in (i.e., predefined) input-output variables[3].

[2]The OpenGL Shading Language is a C-style language, so it covers most of the features you would expect with such a language. This page will note the differences between GLSL and C: `https://www.opengl.org/wiki/Core_Language_(GLSL)`

[3]The OpenGL Shading Language defines a number of special variables for the various shader stages. These predefined variables (or built-in variables) have special properties. They are usually for communicating with certain fixed-functionality. By convention, all predefined variables start with "*gl_*"; no user-defined variables may start with "*gl_*": `http://www.opengl.org/wiki/Built-in_Variable_(GLSL)`

2.2 Vertex Shader

[wikipedia] "Vertex shaders are run once for each vertex given to the graphics processor. The purpose is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen (as well as a depth value for the Z-buffer). Vertex shaders can manipulate properties such as position, color and texture coordinate, but cannot create new vertices."[4]

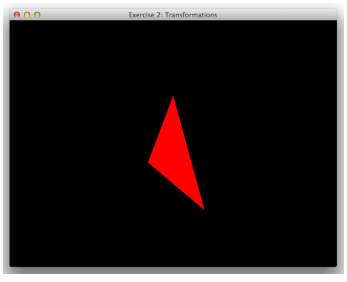
Below you can see a very simple vertex shader. The *Model View Projection (MVP)* matrix converts the vertex positions from object space to normalized device coordinates (i.e., into the cube reaching from [-1, -1, -1] to [1, 1, 1]. Note that the *MVP* is the same for all vertices of our mesh, thus it is was declared as a *uniform* variable. Conversely, each vertex has its own *position* – it cannot be a uniform! Both *position* and *MVP* are provided by the C++ application as we will see in Section 5.

```
/// Inform compiler we are using GLSL 3.30 syntax (mandatory!!)
#version 330 core
in vec3 position; ///< vertex positions passed as "attributes" in the C++
uniform mat4 MVP; ///< used to convert object (3D) to image (2D) coordinates

/// TASK1: declare "const vec3 colors[3]" and define its values
/// TASK2: declare the output "fcolor" variable

void main() {
    /// Homogeneous transformation of vertex coordinates
    gl_Position = MVP * vec4(position, 1.0);

    /// TASK3: assign a color to the fragment (hint: see gl_VertexID built-in)
}
```

Passing parameters across shaders Global variables with either an `in` or an `out` qualifier are used do declare input and ouput attributes for shaders. To pass information from one shader to another, both shaders need to declare the same attribute variable using the same name. The `in` qualifier will be used for in first shader, `out` will be used in the second shader. The input to the very first shader stage, the vertex shader, is specified in the C++ code (see Section 5).

Complete the following tasks:

- **compile and run** the "ex2_trans"; see *Practical Session # 1*.
- what happens if you remove the multiplication by the *MVP* matrix?
- complete the **TASKs** described in the code above. You are drawing a triangle with three vertices. Their colors should be respectively: red, green and blue.
- **execute** the application again and make sure there are **no (shader!!) compiler errors**
- the image produced (shown below) will be... the same as before!!! why?
  because you still have to use that information in the fragment shader!



[4] `http://en.wikipedia.org/wiki/Shader#Vertex_shaders`

2.3 Fragment Shader

[wikipedia] "Pixel shaders, also known as fragment shaders, compute color and other attributes of each fragment. Pixel shaders range from always outputting the same color, to applying a lighting value, to doing bump mapping, shadows, specular highlights, translucency and other phenomena."[5]

The fragment shader below simply colors each pixel in red. Note how *color* is not a built-in parameter. Differently from *gl_Position*, this needs to be **explicitly defined**. By default OpenGL assigns the first *out* variable to the default frame buffer.

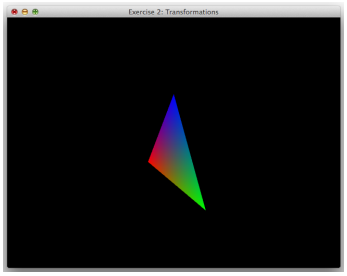
```
#version 330 core
/// TASK1: define the input parameter fcolor
out vec3 color; ///< buffer color

void main() {
    /// TASK2: use the fcolor to define the fragment color
    color = vec3(1.0, 0.0, 0.0); /// color is red for every fragment
}
```

[opengl.org] "The user-defined outputs from the last Vertex Processing stage will be **interpolated** according to their assigned interpolation qualifiers [perspective-correct linear interpolation by default]. The fragment shader **must** define its user-defined inputs **exactly as they were defined** in the vertex shader."

In other words: The vertex shader writes attribute values just for the vertex positions. The fragment shader, however, will run for every pixel on the primitive. The input attribute value for a given pixel will be interpolated between the values of the vertices.

Complete the following tasks:

- edit "ex2_trans" by following the **TASKs** inlined in the code above
- **execute** the application again and make sure there are **no (shader!!) compiler errors**
- the rendered image should have **interpolated colors** as shown below



2.4 Geometry Shader

[wikipedia] "Geometry shaders are a (relatively) new type of shader (OpenGL$\geq$ 3.2). This type of shader can *generate new primitives*, such as points, lines, and triangles. [...] Geometry shaders take as *input a primitive*, possibly with adjacency information. For example, when operating on triangles, the three vertices are the geometry shader's input. The shader can then emit zero or more primitives, which are rasterized and their fragments ultimately passed to a fragment shader."[6]

[5] http://en.wikipedia.org/wiki/Shader#Pixel_shaders
[6] http://en.wikipedia.org/wiki/Shader#Geometry_shaders

When you insert a geometry shader between the vertex and fragment shader stages, you also have to adapt the names of the input and output attributes. The output attributes of the vertex shader need to exist as input attributes in the geometry shader. The output attributes of the geometry shader need to exist as input attributes in the fragment shader. Unfortunately the qualifier `inout`, which would facilitate that task, doesn't exist for attributes passed between stages.

Below you find the implementation of a "pass-through" geometry shader – that is, it does absolutely nothing. Unfortunately a geometry shader cannot be introduced transparently in the pipeline – some re-naming of variables is necessary as illustrated below:

```
#version 330 core

uniform mat4 MVP;
in vec3 position;
const vec3 colors[3] = vec3[](
    vec3(1.0,0.0,0.0),
    vec3(0.0,1.0,0.0),
    vec3(0.0,0.0,1.0));

out vec3 vcolor; ///< per vertex color

void main() {
    gl_Position = MVP * vec4(position, 1.0);
    vcolor = colors[gl_VertexID];
}
```

```
#version 330 core

layout(triangles) in; ///< receives triangles (@see glDrawArrays in c++)
layout(triangle_strip, max_vertices=3) out; ///< outputs triangles
in vec3 vcolor[3]; ///< receives a color per vertex
out vec3 fcolor; ///< outputs a color per fragment

void main() {
    for(int i=0; i<3; i++){
        gl_Position = gl_in[i].gl_Position;
        fcolor = vcolor[i];
        EmitVertex();
    }
    EndPrimitive();
}
```

```
#version 330 core

in vec3 fcolor; ///< passed by gshader
out vec3 color; ///< fragment color

void main() {
    color = fcolor; /// fragcolor is interpolated!!
}
```

Complete the following tasks:

- **Create** a *gshader.glsl*, then **modify** the *CMakeLists.txt* – add a "stringify" rule!!
- **Include** *gshader.h* in the C++ file and add it as 3rd argument to *compile_shaders(...)*.
- **Modify** the shader files as illustrated in the code snippets above.
- **Execute** the application again and make sure there are **no (shader!!) compiler errors**
- Tweak the geometry shader to draw only the **boundary** of the triangle (hint: *line_strip*)

3 External C++ Libraries (Eigen, OpenGP)

In glsl there are data types for vector and matrices and also linear algebra functions. The OpenGL C API doesn't provide such functionality (by design choice) for the rest of our application, i.e., outside the glsl files. There are several libraries that can be used instead. In E. Angel's book the glm library is used. It aims at duplicating glsl as a C++ library. For our course and exercises we decided to go with the popular linear algebra library Eigen[7].

For those of you who are familiar with *Matlab*, this cheat sheet gives a good overview on how to port Matlab operations to Eigen functions.

Below you find some examples that show how to create a projection matrix (as you learned in the lecture), a view matrix, and a scaling transformation. `Eigen::lookAt` creates a view matrix given the camera position, view direction, and an up vector. That means if the matrix is multiplied by a position vector in world space, that vector will get transformed into the camera's coordinate system.

```
#include <iostream>
using namespace std;

/// Matrix and vector math
#include <Eigen/Dense>

/// Geometry module of Eigen
#include <Eigen/Geometry>

/// Eigen support for OpenGL4 is not yet 100% ready.
/// OpenGP adds support to Eigen::perspective() and Eigen::lookat()
#include <OpenGP/GL/EigenOpenGLSupport3.h>

/// The code below is very similar to what we use in the init()
/// function of the source file ex2_trans/ex2.cpp
int main(int, char**){
    /// Defines types similarly to the glm library
    typedef Eigen::Vector3f vec3;
    typedef Eigen::Matrix4f mat4;

    /// Define projection matrix (camera intrinsics)
    mat4 projection = Eigen::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);
    cout << "projection:" << projection << endl;

    /// Define the view matrix (camera extrinsics)
    vec3 cam_pos(4,3,3);
    vec3 cam_look(0,0,0);
    vec3 cam_up(0,1,0);
    mat4 view = Eigen::lookAt(cam_pos, cam_look, cam_up);
    cout << "view:" << view << std;

    /// Define a uniform scale (x2) modelview matrix
    mat4 model = mat4::Identity();
    model(0,0) = 2;
    model(1,1) = 2;
    model(2,2) = 2;

    /// Assemble the "Model View Projection" matrix
    mat4 mvp = projection * view * model;
    cout << "MVP:" << mvp << endl;

    return EXIT_SUCCESS;
}
```

[7] the code is in *external/eigen/*

4 Setting up an OpenGL Context

This is the barebone example of an OpenGL application. There are many different libraries that one could use for setting up an OpenGL context and the corresponding window, e.g., glut, SDL, Qt, SFML, GLFW. We chose to use *GLFW* since it's actively used and developped and since it supports the latest OpenGL versions on all major platforms.

Note that we also use the helper functions defined in *glfw_helpers.h* to make the context creation more similar[8] to what you would do with the *glut* API[9].

The provided *ex2_trans/ex2.cpp* follows the same structure as the barebone below.

```
/// Cross-platform support for OpenGL
#include <GL/glew.h>
/// The glfw2 API we use to open a window
#include <GL/glfw.h>
/// Some wrappers to make glfw behave more like glut (see textbook)
#include <OpenGP/GL/glfw_helpers.h>

void init(){
    /// Compile the shaders
    /// Create the buffers
    /// Initialize uniforms
    /// Initialize attributes
}

void display(){
    /// Draw your buffers here
}

int main(int, char**){
    glfwInitWindowSize(640, 480);
    glfwCreateWindow("Simple OpenGL Window");
    glfwDisplayFunc(display);   ///< registers the callback function
    init();                     ///< initialize your context above
    glfwMainLoop();             ///< calls display() and swaps buffers
    return EXIT_SUCCESS;
}
```

Shader loading The shaders in our framework are specified in .glsl files. We use the CMake build system to directly convert the shader files into .h files that contains a single `const char*` string variable that contains the shader code (this saves us the struggle of having to write a C++ function which loads the file contents into a string). The name of that variable is the same as the original glsl file (without the extension). To use this conversion you have to add a *stringify* rule in *CMakeLists.txt*.

Shader loading workflow

1. Create *my_shader.glsl*
2. Add `target_stringify_shader(program_name my_shader)` to *CMakeLists.txt*
3. `#include my_shader.h` in your C++ file
4. Pass variable `my_shader` to `compile_shaders(...)`

[8] You can find a full glfw context creation example without the use of these helpers in *ex1_hello/ex1.cpp*.
[9] this is what E. Angel uses in his book, but OpenGL4 support was really problematic on some platforms.

5 Passing information to shaders

The input to the shaders needs to be defined in the C++ application. There are two fundamentally different types of input data for shaders, *uniform* and *attribute* variables. Uniforms contains data that doesn't change for an entire draw procedure, e.g., the MVP matrix in our example. On the other hand, an attribute contains data which is different for every vertex, e.g., the vertices positions in our case. Both shader input types have to be set up differently.

5.1 Initializing Vertex Buffers and Attributes

Since attributes contain data for several vertices, we want to store them in something like an array. We can allocate a contiguous region of memory on the GPU with `glGenBuffers`. A OpenGL buffer filled with vertex data is called a *VBO* (Vertex Buffer Object). Generating such a VBO returns a handle which must be bound (`glBindBuffer`) before we can write to it (or read from it). In OpenGL there is always only one active or current buffer, the one that was bound most recently. `glBufferData` is called to copy the vertex positions from a CPU array into the VBO in the GPU memory of the bound buffer.

The second required step is to tell our shader where (in GPU memory) the vertex positions are. The following code shows hot to set up the a VBO and map it to input attribute named `position` in the shader. We must get a handle to that shader variable (`glGetAttribLocation`), activate it (`glEnableVertexAttribArray`), and map it to our VBO and define the alignment of the data (`glVertexAttribPointer`).

```
/// For a working example see the init() function of the source file ex2_trans/ex2.cpp

/// Vertex positions of a triangle
const GLfloat vertices[] = {
       -1.0f, -1.0f, 0.0f,
       1.0f, -1.0f, 0.0f,
       0.0f,  1.0f, 0.0f,};

void init(){
    /// [...] loads and compiles the shaders
    /// [...] create vertex array (described in a future practical)

    /// Create and bind a vertex buffer
    GLuint vertexbuffer;                        ///< an ID to identify the buffer
    glGenBuffers(ONE, &vertexbuffer);           ///< we just need one
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer); ///< make it "current"

    /// Let OpenGL know where vertex coordinates data is
    glBufferData(GL_ARRAY_BUFFER,
                 sizeof(vertices),              ///< how much data to load
                 vertices,                      ///< pointer to the first data element
                 GL_STATIC_DRAW);               ///< hint on how buffer will be used


    ///------------------------------------------------------------------------
    /// But now they have have not been named yet, it is just a chunk of
    /// memory, the shader does not yet know what name to use, whether
    /// they are 3D vertices, 4x4 Matrices, etc... let us fix that!
    ///------------------------------------------------------------------------

    /// Fetch the "position" variable by name in the shader program
    GLuint position = glGetAttribLocation(programID, "position");
    glEnableVertexAttribArray(position);        ///< enable it
    glVertexAttribPointer(position,             ///< in the data chunk position
                          3,                    ///< each element has three elements (vec3)
                          GL_FLOAT,             ///< and they are floats
                          DONT_NORMALIZE,       ///< (IGNORE FOR NOW)
                          ZERO_STRIDE,          ///< (IGNORE FOR NOW)
                          ZERO_BUFFER_OFFSET);  ///< (IGNORE FOR NOW)
}
```

5.2 Initializing uniform variables

Uniforms don't need to be set up as buffers since they (usually[10]) don't contain large arrays of data. For uniforms it's enough to query the location on the GPU (`glGetUniformLocation`) and to write the value to that location (`glUniformMatrix4fv`). `glUniform*` exists in many forms for floats, int, vectors, and matrices, see `http://www.opengl.org/wiki/GLAPI/glUniform`.

```
/// For a working example see the init() function of the source file ex2_trans/ex2.cpp
void init(){
    /// [...] portion of code that loads and compiles the shaders

    /// Create some data
    typedef Eigen::Matrix4f mat4;
    mat4 mvp = mat4::Identity();

    /// Pass it to the (vertex) shader. Note that MVP is the exact shader variable name
    GLuint mvp_id = glGetUniformLocation(programID, "MVP");
    glUniformMatrix4fv(mvp_id, 1, GL_FALSE, mvp.data());
}
```

Once everything is set up, calling `glDrawArrays(GL_TRIANGLES, 0, 3)` will draw a triangle (start at position 0 in the buffer and draw 3 vertices as a triangle). See *ex2_trans/ex2.cpp* for the full code.

You find more information about uniforms and vertex attributes on the OpenGL Wiki.

[10] it's possible though: `http://www.opengl.org/wiki/Uniform_Buffer_Objects`