Computer Graphics
Practical 1 - Getting started

INSA Fourth Year - 2023/2024
Maud Marchal, Glenn Kerbiriou

1 Objectives

This first practical is divided into two parts: tutorials and exercises. Tutorials gradually present the framework used in the practicals while the exercises are about the basic of the graphics library standard, OpenGL. Along the practical we put notes about the graphics pipeline. We strongly advise you to explore the API documentation as often as possible and come back to the notes as many times as you need.

2 What is OpenGL?

Similarly to other graphics API like DirectX or the more recent Vulkan, OpenGL is a graphics API which is intended to create real-time rendering engines. Using OpenGL means interacting with your graphics card through an interface written in C which allows you to call its available hardware functions. The graphics card possesses its own memory (VRAM) and its own processor (GPU) which has the specific ability to be highly parallelizable. The first stage of a rendering engine is to create an OpenGL context which defines the way we want to use OpenGL. Then, the usual second stage consists in loading data resources like meshes, textures, and shaders into the VRAM. And finally, the OpenGL pipeline can proceed as a loop to render images on the screen.

There is an important distinction between what we call "modern OpenGL" versus "old compatibility". Since OpenGL 3.3, it is possible to use lower-level functions from your graphics card. It has the advantage to obtain more customization on the OpenGL pipeline despite of a bit higher complexity in terms of programming. It is what we call "modern OpenGL". That's why it exists two usage modes, "core profile" and "immediate mode". The difference between both is that "core profile" prohibits you to use depreciated high-level OpenGL functions whereas the "immediate mode" does not, but this one assures compatibility with older hardware. In our case, we will use OpenGL 4 with the enabled "core profile".

3 OpenGL context

Actually, we can see the created OpenGL context as a data structure in your graphics card, containing information which could be relevant for next function calls. For instance, if you want to write texture information in a memory buffer located in VRAM, you have to be sure that the writing will be performed on the appropriate memory buffer knowing that the buffer selection is stored inside your OpenGL context. For this reason, it is important to be aware of the state of the OpenGL context, as it is often necessary to alter it before some function calls.

4 Framework

In order to avoid to start completely from scratch, we will use a small framework (engine) that you will be charged to explore and complete along the practicals. At the end, you will have to use it to realize your project. This framework uses four external libraries:

- GLEW: OpenGL core and extension wrapper that detects which OpenGL extensions are available.
- SFML: Window system and detection of keyboard/mouse input events.
- GLM: Mathematics library respecting the OpenGL Shading Langage data structures convention.
- Freetype: Fonts library for text display.

For a better understanding of OpenGL, as a complement of your courses, we recommend two great websites:

- `https://learnopengl.com` (OpenGL tutorials with explanations)
- `http://docs.gl` (The best OpenGL documentation)

5 Tutorial 1: Build the practical framework

Download on Moodle the source code for this practical. In the extracted archive you will find two folders: the folder "sfmlGraphicsPipeline" contains the source code of the framework whereas the folder "sampleProject" is a front program which uses the framework. Building the framework outcomes a static library whereas building the front program leads to an executable which depends on the latter. Roughly, you can visualize the framework as a toolbox and the front program as a worker.

Instead of directly using a C++ compiler, the build of both programs is performed thanks to CMake, a higher-level tool. CMake allows you to generate a Makefile (or other kind of files) you can then execute to compile your program through GCC or another compiler. Thus, to compile the external libraries, the framework and the executable, you have to type the following commands:

```
cd sfmlGraphicsPipeline

cd extlib
make -j6  # compile the external libraries
cd ../

mkdir build
cd build
cmake ../   # generate the Makefile using CMakeLists.txt
make -j6  # compile the framework (static library)
cd ../../

cd sampleProject

mkdir build
cd build
cmake ../ # generate the Makefile using CMakeLists.txt
make -j6  # compile the front program (executable)
```

These commands create a build/ directory that will contain all files created to build the executables. Calling `cmake ../` will use the `CMakeLists.txt` file at the root directory to setup the build system and automatically generate a Makefile. Whenever you add new source files in the src/ or include/ directories, you need to call again `cmake ../` from the build/ directory in order to use them for building the executable. The executable is named main ; it should be created (by typing `make`) and executed in the build/ directory. Note that for the last `make` in sampleProject/build you can specify a target to build like so : `make practical1`. If no specific target is provided, all the `.cpp` files in sampleProject will be compiled as individual executables. To facilitate the compilation and lauching of your program you can use the script `run.sh` : `./run.sh [target] [r (run) | cr (compile run) | ccr (cmake compile run)]`

Now, you can launch the produced executables.

6 Tutorial 2: Window and main loop

Generally, a window does several things:

1. It initializes an OpenGL context.
2. It handles events such as mouse or keyboard events.
3. It allows to display on screen what we have drawn in a frame.

In these practicals, the window is wrapped inside the `Viewer` class. Roughly, the viewer represents your virtual scene. First of all, explore the pre-filled and commented main function in file sampleProject/src/practical1.cpp. You can notice the presence of the 3 usual stages that were explained in the introduction of this practical. But the second stage is empty, so for now there are no entities (like physical objects) in your virtual scene. Launch the executable, you should simply see an empty scene.

```
cd sampleProject/build/
chmod +x practical1
./practical1    # launch the practical 1 executable
```

Explore the method "initializeGL()" in the Viewer class. It loads the API using GLEW library and then it performs the OpenGL context creation. The next tutorials are about how to add elements in your virtual scene. In other words, we will fill the second stage.

7 Tutorial 3: Renderable

The `Renderable` class simply depicts an object that can be added to the Viewer so that it will be drawn on the window. In practice, the Viewer class contains a set of Renderable. When calling the `Viewer::draw()` function, it loops over the set of Renderable and draws each of them.

We will instantiate objects to create our scene. First, include the following headers in the file src/practical1.cpp.

```
# include "./../include/ShaderProgram.hpp"
# include "./../include/FrameRenderable.hpp"
```

Second, instantiate a **shader program** that will be used to draw the Renderable. Add it to the viewer.

```
// Path to the vertex shader glsl code
std::string vShader = "./../../sfmlGraphicsPipeline/shaders/defaultVertex.glsl";
// Path to the fragment shader glsl code
std::string fShader = "./../../sfmlGraphicsPipeline/shaders/defaultFragment.glsl";
// Compile and link the shaders into a program
ShaderProgramPtr defaultShader = std::make_shared<ShaderProgram>(vShader, fShader);
// Add the shader program to the Viewer
viewer.addShaderProgram(defaultShader);
```

Finally, instantiate a renderable and add it to the viewer.

```
// Shader program instantiation
// ...
// When instantiating a renderable ,
// you must specify the shader program used to draw it .
FrameRenderablePtr frame = std::make_shared<FrameRenderable>(defaultShader);
viewer.addRenderable(frame);
```

When launching the executable you should now get the axes of your coordinate system in your window. Note that you can control the view using `z|q|s|d`, `lshift|space` and `up|down|left|right` keys. This is done by a `Camera` class that updates the view matrix for us. You can change the camera mode by pressing `c` key.

About the shared pointers
You may have been confused by the use of shared pointers, `std::make_shared<>`. Simply put, it is a convenient way to automatically handle the allocation and release of the memory managed by the pointer. No call to the operators new and delete are needed anymore. If you want more information, we strongly recommend you to read the documentation as it is now a standard in C++.

8 Notes about the graphics pipeline

8.1 About shaders

About shaders
Some steps of the pipeline (see Figure 1) are programmable by the user through programs called shaders:

- **Vertex shader**: A vertex is an object that will be grouped with others to form primitives. In OpenGL, you have three kinds of primitives: triangles (3 vertices by primitives), lines (2 vertices by primitives) and points (1 vertex by primitives). Any other primitives need to be built from these three. A vertex is represented by its attributes, such as its position, color and texture coordinates. The purpose of the vertex shader is to perform computations on the attributes of each vertex, such as world to screen transformation.
- Tessellation shader: Evaluate if new vertices should be created (optional; in fact, this is two different shaders).
- Geometry shader: Performs computations on primitive (optional). It can discard a primitive or creates more primitives.
- **Fragment shader**: Once all the primitives had been defined, the rasterization process takes place. This process discretizes a primitive into fragments. For now, you can think of fragments as the pixel of the screen. The attributes of the vertices (position, color, ...) are linearly interpolated on the fragment and are used to compute the final color of the fragment, for instance using a local illumination model. The fragment shader performs computations on all these fragments, e.g. to compute the local illumination or to texture them.
- Compute shader: Performs general purpose computations on the GPU (aka GPGPU) as a substitute to CUDA or OpenCL. This shader is the only step out of the graphics pipeline and is optional.

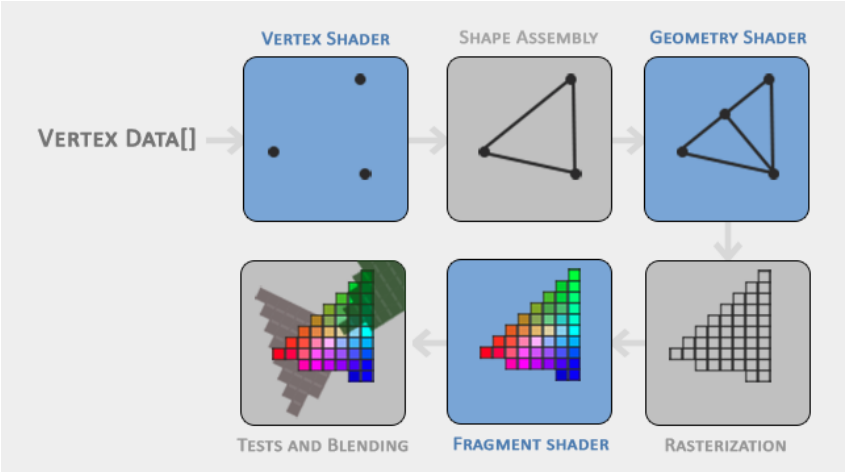

Figure 1: OpenGL graphics pipeline

In these practicals we will focus on the Vertex shader and the Fragment shader. In contrast with the other shaders, they do not have a default implementation. Therefore, they always must be implemented by the user and provide an output to the next stage of the graphics pipeline.

- The vertex shader should at least provide the position of vertices in the clipped coordinates, `vec4 gl_Position`.
- The fragment shader should at least provide the color of the fragments resulting from the rasterization stage, a variable with type `vec4`.

About GLSL
Shaders are written in c-like language called GLSL (OpenGL Shading Language). Each of them define a function void `void main()` that will be called on the GPU to process all vertices in the vertex shader and fragments in the fragment shader. For now, the more important is to understand the following keywords:

- `in/out`, respectively used to specify input/output variables of the shader. Please note that if you want to transfer variables from the vertex shader to the fragment shader, then, the outputs of the vertex shader **must match the names and the types** of the inputs of the fragment shader.
- `uniform`, used to define global variables that will remain the same for every vertex and every fragment within the same drawing command. Such variables are typically constants of an object to render, like its position, orientation.

And the following built-in variables:

- `vec4 gl_Position`, the clip coordinates position of a vertex, that should be set for every vertex in the vertex shader.

9 Tutorial 4: Create your own Renderable

9.1 Shaders

First of all, you need to prepare the vertex and fragment shaders. In the folder `shaders/`, create two files "flatVertex.glsl" and "flatFragment.glsl".

Vertex shader As mentioned before, the vertex shader is mostly used to place the vertices of a geometry in the clipped space.

```
# version 400 // GLSL version, fit with OpenGL version
uniform mat4 projMat, viewMat, modelMat;
in vec3 vPosition;
out vec4 color;

void main ()
{
  // Transform coordinates from local space to clipped space
  gl_Position = projMat * viewMat * modelMat * vec4 (vPosition, 1);
  color = vec4 (1,0,0,1); // RGBA color defined in [0,1]
}
```

Fragment shader The fragment shader is mainly used to define a color for the fragment of the rasterized primitives. The following shader is one of the simplest fragment shader: it only transfers the color of the fragment to the frame buffer. It is important to keep in mind that this color has been linearly interpolated from the colors of the primitive vertices at the fragment position.

```
# version 400 // GLSL version, fit with OpenGL version
in vec4 color;
out vec4 fragmentColor;

void main ()
{
  fragmentColor = color;
}
```

Shader program We just defined what happens in the graphics pipeline. We now have to focus on the CPU side. This starts by compiling and linking your shaders into a shader program. This process has been encapsulated in the `ShaderProgram` class.

```
// Viewer instanciation
// ...

// Default shader instanciation
// ...

// Compile and link the flat shaders into a shader program
vShader = "./../../sfmlGraphicsPipeline/shaders/flatVertex.glsl";
fShader = "./../../sfmlGraphicsPipeline/shaders/flatFragment.glsl";
ShaderProgramPtr flatShader = std::make_shared<ShaderProgram>(vShader, fShader);

// Add the shader to the Viewer
viewer.addShaderProgram(flatShader);

// Renderable instantiation
// ...
```

9.2 Create a 3D model

The 3D model is created inside a renderable object. For this tutorial, the files CubeRenderable.hpp and CubeRenderable.cpp have been prepared and will progressively be filled. First, let's prepare the file `practical1.cpp` while keeping in mind that the viewer must render an instance of `CubeRenderable`.

```
// Do not forget the include
# include "./../include/CubeRenderable.hpp"

// ...

// Instantiate a CubeRenderable while specifying its shader program
CubeRenderablePtr cube = std::make_shared<CubeRenderable>(flatShader);

// Add the renderable to the Viewer
viewer.addRenderable ( cube );
```

The CubeRenderable class contains two members:

- *m_positions*: the vector of vertex positions that describe the object, on the CPU.
- *m_vBuffer*: the identifier of the vertex position buffer on the GPU that stores the positions.

About vector of positions
As mentioned above, positions are stored as a vector of 3D positions. A vector is c++ collection from the standard library. Positions are represented as an array of three float values. Many libraries exist to manipulate this very basic data structure. In the practicals, we chose to use the GLM library.

The class also contains two inherited members from the `Renderable` abstract class.

- m_shaderProgram: A smart pointer to the shader program used to render the Renderable.
- m_model: A model matrix to place the object in world space.

The first thing to do is to define a 3D geometry and a default world transformation. This is done in the constructor of `CubeRenderable`:

```
CubeRenderable::CubeRenderable(ShaderProgramPtr shaderProgram)
  : Renderable(shaderProgram)
{
    //Build the geometry: just a simple triangle for now
    m_positions.push_back( glm::vec3(-1,0,0) );
    m_positions.push_back( glm::vec3(1,0,0) );
    m_positions.push_back( glm::vec3(0,1,0) );

    //Set the model matrix to identity
    m_model = glm::mat4(1.0);
}
```

A common beginner mistake is to recompute the object geometry at every frame (for example, in the `do_draw()` function). As this geometry does not change, there is no need to recompute it. Thus, computing only one time in the constructor is the right way to do it.

Then, the geometry is sent to your graphics card in VRAM, in an allocated memory stamp that is called **buffer**. As in classic CPU programmation, the buffer is first declared, then memory is allocated and data are assigned to it. The data structure containing your geometry, in the OpenGL convention, is called Vertex Buffer Object (VBO).

Check your GL commands
OpenGL commands all start with the `gl` prefix. Such commands can fail, because the state is incoherent with what you are trying to do or because you sent wrong parameters. These errors are silent: no exception is thrown, no log entry is displayed. Thus, debugging an OpenGL program can quickly become a hassle. In order to ease the debugging process, we provide a c++ macro `glcheck( ... )`. Be sure to use this macro for ALL your OpenGL command. If you are concerned about the runtime overhead such a macro would cause, be assured, the macro does nothing when the code is compiled in release mode.
In these instructions, we will omit `glcheck(...)`, in order to improve the readability.

```
// Still in the constructor. Following previous code.

// Create a new buffer identifier
// (This is the "name" of a pointer variable on the GPU)
glGenBuffers(1, &m_vBuffer);
// Bind the buffer to the GL_ARRAY_BUFFER binding point
// (This is a place to perform vertex attributes operations)
glBindBuffer(GL_ARRAY_BUFFER, m_vBuffer);
// Transfer data to our new buffer thanks to this binding point
// This function resize the buffer to the requested side
glBufferData(GL_ARRAY_BUFFER, m_positions.size() * sizeof(glm::vec3), m_positions.data(),
     GL_STATIC_DRAW);
```

Finally, never forget to release the buffer in the destructor.

```
CubeRenderable::~CubeRenderable()
{
  glDeleteBuffers(1, &m_vBuffer);
}
```

At this point, `CubeRenderable` creates the geometry of a triangle and sends it to the GPU. However, there is no drawing command yet to render this geometry using a shader program. This is done in the inherited function `do_draw()` of the `Renderable` class.

In this function, all the inputs of the shader program have to be set. For that purpose, the internal format of the GPU vertex attribute buffers are specified and explicitly linked to the GLSL code of the shaders. Once this is done, the drawing command can be issued. Looking at the GLSL code of the so-called flatShader, the inputs are:

- projMat: The projection matrix. This is handled by the Viewer's camera.
- viewMat: The view matrix. This is handled by the Viewer's camera.
- modelMat: The model matrix. We need to sent it to the GPU and link it.

- vPosition: The vertex position. We need to link it.

```
CubeRenderable::do_draw()
{
  // Get the identifier (location) of the uniform modelMat in the shader program
  int modelLocation = m_shaderProgram->getUniformLocation("modelMat");
  // Send the data corresponding to this identifier on the GPU
  glUniformMatrix4fv(modelLocation, 1, GL_FALSE, glm::value_ptr(m_model) );

  // Get the identifier of the attribute vPosition in the shader program
  int positionLocation = m_shaderProgram->getAttributeLocation("vPosition");
  // Activate the attribute array at this location
  glEnableVertexAttribArray(positionLocation);
  // Bind the position buffer on the GL_ARRAY_BUFFER target
  glBindBuffer(GL_ARRAY_BUFFER, m_vBuffer );
  // Specify the location and the format of the vertex position attribute
  glVertexAttribPointer(positionLocation, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

  // Draw the triangles
  glDrawArrays(GL_TRIANGLES, 0, m_positions.size());

  // Release the vertex attribute array
  glDisableVertexAttribArray(positionLocation);
}
```

When launching the executable, you should get your triangle in your window.

Quick summary of OpenGL calls in a Renderable

Initialization - Constructor

```
//Create buffers on the GPU and returns its ID
//The ID is similar to a pointer name in CPU code
glGenBuffers(...);

//Activate a buffer: Next operations will occur on this buffer
//Specify that data on the buffer are attributes of vertex
glBindBuffer(GL_ARRAY_BUFFER, ...);

//Allocate the buffer memory and transfer data from CPU to GPU
glBufferData(GL_ARRAY_BUFFER, ...);
```

Runtime - do_draw()

```
//Enable
//Activate the attribute array at this location
glEnableVertexAttribArray(...);

//Activate a buffer: Next operations will occur on this buffer
//Specify that data on the buffer are attributes of vertex
glBindBuffer(GL_ARRAY_BUFFER, ...);

//Specify the location and the format of the vertex attribute
glVertexAttribPointer(...);

//Draw openGL primitives
glDrawArrays(GL_TRIANGLES,...);

//Release the vertex attribute array
glDisableVertexAttribArray(...);
```

Destruction - Destructor

```
//Delete the buffers
glDeleteBuffers(...);
```

10 Exercice 1: Adding new vertex attributes

1. Modify flatVertex.glsl so that it also takes color as vertex attribute.
2. In CubeRenderable, add a list of colors for the vertices.
3. In CubeRenderable, send the colors to the GPU and link them with the shaders.

11 Exercise 2: Geometry without indexing

1. In CubeRenderable, create the geometry of a cube without indexing.
2. Assign one color to each triangle that composes the cube.
3. How many triangles do you need? How many vertices?

What is indexing?
Geometric primitives, i.e. points, lines, and triangles, are represented using positions. Very often, some adjacent primitives have vertices that share a common 3D position. In such a case, it would be helpful to share the same vertex between the different primitives instead of duplicating that vertex. This is achieved by indexing: the index of a vertex (its order of appearance in vertex buffers) is used to describe primitives. Then, for every primitives using a particular index, the vertex attributes (position, color, normal, ...) of the vertex at that index are re-used.
In OpenGL, indexing only means adding a new buffer that will contain the vertex indices to use to build the primitives we want to render. This buffer will be bind on the GL_ELEMENT_ARRAY_BUFFER target. This drawing command is changed to a version that will use the index buffer. See the code below for implementation detail:

Initialization - Constructor

```
//For position, color and index
glGenBuffers(...);

//For position and color buffer
glBindBuffer(GL_ARRAY_BUFFER, ...);
glBufferData(GL_ARRAY_BUFFER, ...);

//For index buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ...);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, ...);
```

Runtime - do_draw()

```
//For position and color
glEnableVertexAttribArray(...);
glBindBuffer(GL_ARRAY_BUFFER, ...);
glVertexAttribPointer(...);

//For index
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,...);

//Draw openGL primitives
glDrawElements(GL_TRIANGLES,...);

//For position and color
glDisableVertexAttribArray(...);
```

Destruction - Destructor

```
//For position, color and index
glDeleteBuffers(...);
```

12 Exercice 3: Geometry with indexing

Copy CubeRenderable in a new renderable called IndexedCubeRenderable and modify it so that it uses indexing.

1. Modify the list of positions and create a list of indices.
2. Use only one color per vertex.
3. Send the indices to the GPU and link them with the shaders.
4. How many triangles do you need? How many vertices?

13 Exercice 4: Basic transformation

1. In practical1.cpp, instantiate a CubeRenderable and an IndexedCubeRenderable.
2. Use the following functions to position the cubes next to each other in the scene.
   - `Renderable::setModelMatrix()`
   - `glm::translate()`
3. Use the following functions to deform one of the cube and rotate the other one.
   - `glm::rotate()`
   - `glm::scale()`
4. As you can see the syntax for creating a matrix is not very convenient. Read the following easing functions in Utils.cpp.
   - `getTranslationMatrix()`
   - `getScaleMatrix()`
   - `getRotationMatrix()`