Computer Graphics and Virtual Reality
IGR202 Practical - Shadow

Kiwon Um, Telecom Paris
Jean-Marc Thiery, Telecom Paris

The objective of this exercise is to learn and deepen the knowledge about how to program a modern interactive graphics application using OpenGL (version 3.3 or later). The final goal is to implement two real-time rendering techniques: *normal mapping* and *shadow mapping*. Once you finish all the tasks described here, you can extend your codes further as you want.

1 Codebase

Your first task is to understand the overall structure of the provided codebase. Please read through the `main.cpp` and the other files. It may take ca. 20 minutes. The provided codebase is written in C/C++; you may need to refer to any resources about C/C++ programming. The codebase uses the following libraries:

- OpenGL for accessing your graphics processor
- GLFW to interface OpenGL and the window system of your operating system
- GLM for the basic mathematical tools (vectors, matrices, etc.).

1.1 Build and run

Important! This guideline is written for Linux systems. If you use other operating system, you should adapt it accordingly.

The codebase uses *cmake* as a build system. You can easily build an executable via general cmake commands. See Code 1. You should make sure that the two libraries, i.e., glfw and glm, are installed on your machine.

Code 1. Build and run

```
mkdir build       # under your main source directory where CMakeLists.txt exists
cmake -B build    # or, 'cd build' and then 'cmake ..'
make -C build     # or, simply 'make' under the build directory
./tpShadow        # You should be careful with your working directory!
                  # When running, it will try to load ./src/*.glsl and ./data/*.{png,off}.
```

Congratulations! If everything is compiled well thus you run your executable, you should be able to see an initial OpenGL window as on the left of Fig. 1; the middle and right of Fig. 1 are examples you may achieve over the series of tasks. You can use Esc to quit. Pressing F1 toggles visualizing the wire structure (useful for debugging) and surfaces of your meshes. t will turn on/off a simple animation of your scene. For more details, press h and see the console.

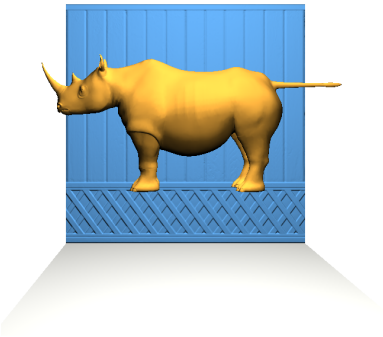
Fig. 1. Screen captures of (left) the very first run and (middle and right) two examples of this exercise.

2 Initial Setup

The given package already contains a significant amount of codes. Before adding yours, you first need to understand the structure of the codebase. In particular, you should take a close look at two functions:

- `initScene()`: scene setup including meshes and lights
- `Scene::render()`: main rendering procedure

3 Normal Mapping



Let's start with making your back-wall more realistic. The back-wall simply consists of two triangles with four vertices. (Try F1.) Without using any complex geometry meshes, you will add nice surface details via a texture mapping technique, which we call *normal mapping*. (See the inset figure.)

- Use `data/normal.png` file as a normal map texture.
- You should keep in mind that each color value (loaded from the texture) is defined in [0, 1]; thus, those values should be remapped into the range of [-1, 1] such that you properly interpret the data as normal vectors.
- You should update `initScene()` such that it loads and binds the texture.
- You should update `Scene::render()` such that it sets necessary variables of your shader.
- You should update `fragmentShader.glsl` such that it uses your normal map texture as you intend.

4 Texture Mapping

Let's advance the texture mapping further. The back-wall still does not present nice colors. You will add colors on the surface via additional texture. (See the inset figure.)

- Use `data/color.png` file as an albedo (i.e., diffuse color) texture of the back-wall.
- You should update `initScene()` such that it loads and binds the textures.
- You should update `Scene::render()` such that it sets necessary values for your shader.
- You should update `fragmentShader.glsl` such that it uses your textures as you intend.

5 Shadow Mapping

Rendering shadows takes two steps: generate shadow maps and then use them when rendering the scene. You will implement these two steps.

5.1 Shadow Map Generation

To generate shadow maps, you need to render (i.e., store depth buffers of) your scene as if your camera is placed at each light position and shooting the entire scene. This procedure can be taken in both the `Light::setupCameraForShadowMapping()` and `Scene::render()` functions. You do not need to modify two shader programs, `vertexShaderShadowMap.glsl` and `fragmentShaderShadowMap.glsl`. However, you have to use them at the right place. Hints:

- You can use `glm` functions (e.g., `glm::ortho<float>()` or `glm::perspective<float>()` as well as `glm::lookAt()`) in `Light::setupCameraForShadowMapping()`.
- You should properly set the shader variable, `uniform mat4 depthMVP`.
- To visually check your shadow maps, you can save them as PPM files by pressing s. (See the inset figure.)

5.2 Shadow Rendering

Now, you are ready to render your scene with shadows! Hints:

- Your shadow map textures are already bound and storing the depth values if successfully generated. (See `initScene()`.) To use them, you should set sampler variables (in your fragment shader).
- You should also set the transformation matrices that you used for shadow map generation. These matrices will transform your fragments into the same space where you generated your shadow map.
- You should keep in mind that the transformed fragments are in [-1, 1] (i.e., normalized device coordinates), but texture sampling should happen in [0, 1].

6 Extensions

There is no limitation in this exercise. You are strongly encouraged to further investigate any extensions you are interested in. You can find a list of potential directions in the following:

- Adding interactive handling of lights, e.g., turning on/off or moving
- Adding different objects in the scene
- ...