IGR202 Practical Assignment

Rendering (3 x 3 hours)

The objective of this assignment is to develop a modern interactive graphics application using OpenGL (v4.5). It's a good idea to keep the OpenGL documentation opened all the time: www.opengl.org. Refer to this documentation for both CPU-side OpenGL (C/C++) call and GPU-side GLSL shader code. Note that underlined text provide hyperlinks to useful documentations. The base code (BaseGL application) is written in C++: one can refer to the resources from INF224 for details about C++ and object-oriented programming.

The following libraries are used by BaseGL:

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

For building BaseGL, we will use cmake:

```
cd <path-to-BaseGL>
mkdir build
cd build
cmake ..
cd ..
cmake --build build
```

The main program and its dependencies will be generated using your local compiler.
To run the program:

```
cd <path-to-BaseGL>
./BaseGL <filename.off>
```

This should work on all platforms with cmake properly installed and configured. For now, the program loads the mesh and display it with a plane behind it. It provides basic interaction e.g., camera navigation (press 'h' to print the help). The general philosophy of this program is:

to define and manipulate a set of C++ objects representing the application entities in Main.cpp – as global variables

to initialize them in init ()

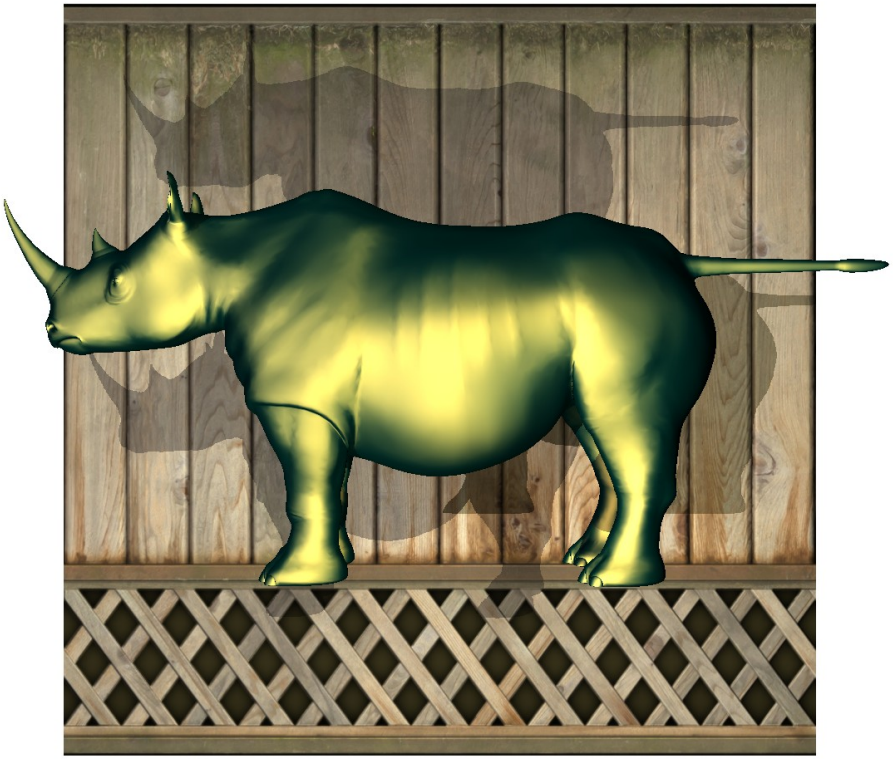
to use them in render () to synthesize an image, by mapping them to structures defined in the shader code, passing their variables from C++ classes to GPU shaders (see how modelview and projection matrices are managed for instance).

Preprocessing algorithms should be implemented in C++ on CPU side, while real time shading methods should be implemented in GLSL (vertex and fragment shaders located in Resources/Shaders) on GPU side. You are free to modify the provided class depending on your needs (e.g., adding new set methods to the ShaderProgram class).

Goal of the assignment

You will learn in this assignment:
- how to setup a complex shader and set uniform variables appropriately
- how to use Frame Buffer Objects, in particular in the context of shadow map rendering
- how to setup shadow maps for lights in your 3D scene
- how to filter shadows or compute soft shadows
- how to setup a complex lighting model
- how to use PBR (Physically Based Rendering) materials

Eventually, you will produce this type of output:



0) Initial setting

Currently, the lighting model implemented in the fragment shader should look like what you implemented at the end of IGR201. There is a single point light in the scene, and the BRDF is a standard Diffuse + Specular Phong model.

Analyze the structure of the function

```
void Scene::render() ;
```

In a first pass, for each light in the scene, a Frame Buffer Object is attached (see class), and the scene geometry is rendered from the point of view of the light (you will need to implement the camera parameters later), with only the depth as an active output. The shader that is used is at this step shadomMapShaderProgramPtr .

In a second pass, the main shader shaderProgramPtr is used, and the geometry is rendered on screen. Later on, you will use the shadow maps computed in the first pass to render shadows in real time.

1) Lights

Add a number NL > 1 of lights in the scene.
It is required of you to adapt the code in the fragment shader, and in particular **it is required of you to use an array of lights as uniform variable**.

Your fragment shader should therefore contain the following lines:

```
struct LightSource {
    vec3 position;
    // etc ...
};

int number_of_lights = 3; // for example

uniform LightSource lightSources[ 3 ];
```

To set the uniform variables from the CPU side in this case, refer to the slides of the course.

2) Construction of the shadow maps

a) Set an appropriate camera for each light in your scene.
This computation can take place in the
`void Light::setupCameraForShadowMapping(glm::vec3 scene_center , float scene_radius)`
function for example.

You can use the following functions of the GLM library:
`glm::ortho<float>  /  glm::perspective<float>`
`glm::lookAt`

To see what your shadow maps look like, run the program and press "s". It will save as many shadow maps as you have lights in PPM files.

b) Analyze the shadow map shader (FragmentShader_shadowMap.glsl / VertexShader_shadowMap.glsl).
This shader is rather trivial: it simply renders the geometry from the point of view of the light (whose camera you just computed) and outputs the depth.

3) Rendering shadows

You will now use the shadow maps that you have computed in order to render the shadows in real time.
This step should be the most time-consuming. You can refer to the course slides.

There are several required steps:
- Give the shadow map texture as well as the corresponding shadow map camera to the main shader.
- Given a fragment, for each light, test whether the fragment lies in the shadow or is lit by the light.

Implement simple heuristics to avoid the (in)famous self-shadowing artifact.

4) Enriched lighting

- For each light, attenuate its color based on the distance from the point to the light position, using a quadratic polynomial in d. (Is it physically well motivated?)

$$L(d) = \frac{L}{a_c + a_l \cdot d + a_q \cdot d^2}$$

- For some lights, setup a cone within which the light is contained.

- Animate a few parameters of your lights (you can look at the function void **update** (float currentTime) in which the position of the first light is animated using a timer – press "t" to restart/stop the timer).

~~*5) Shadow filtering*~~

~~Now that you know how to test if a 3D point is in a shadow or not, you can implement simple filtering schemes to compute soft shadows. More details given if you reach this step by the end of the first assignment (should not be the case).~~

6) PBR Materials

Implement a simple microfacet-based (not using textures) based on the GGX and Smith distributions. Important parameters will be:
- the Fresnel value F_0
- the roughness value α of the surface.

You can refer to rendu2 for that part.

You should include a uniform variable to your shader, to specify which type of rendering you will use (Phong, Microfacets).

To visualize the expected effect, have a look at the figure given in page 2, where the rhino has been given a "gold" material).

7) Toon rendering

Implement a simple toon shading.

Specifically, display contours, and apply a strong quantization on interesting values (e.g., set the same albedo value for a wide range of dot(n,wo)).

You should include a uniform variable to your shader, to specify which type of rendering you will use (Phong, Microfacets, TOON).

We have talked about toon rendering in rendu1. You can have a look at the figure on the right side of the previous to last slide to see the expected result.

8) PBR Materials

Implement the PBR rendering using textures that describe the spatially-varying PBR material, and use these textures to shade the plane in your 3D scene.
These materials can be found in BaseGL/Resources/Materials/
This directory contains a few subdirectories, each of which containing a specific set of maps (Base_Color.png, Roughness.png, etc) for a given material.

To visualize the expected effect, have a look at the figure given in page 2, where the rhino has been given a “Wood” material).