Assignment 3 — Shaders

Lund University Graphics Group

This assignment introduces ones of the major shading languages, GLSL, as well as applications for some common shading techniques such as *Phong shading*, *cube mapping* and *normal mapping*. You will develop vertex and fragment shaders—often referred to in tandem simply as shaders—and apply them to scene graph objects to spice up their appearance.

Similar to the previous assignment, you can change shaders and the polygon mode from the user interface. Note that changing shaders will only affect the `demo_sphere` variable. If you would like to change other objects independently, look for the following code in the rendering-loop, and duplicate and modify it as needed:

```
auto demo_sphere_selection_result =
↪  program_manager.SelectProgram("Demo sphere",
↪  demo_sphere_program_index);
if
↪  (demo_sphere_selection_result.was_selection_changed)
↪  {
        demo_sphere.set_program(
                demo_sphere_selection_result.program,
                set_uniforms);
}
```

The `SelectProgram()` function takes a string that will be displayed in the GUI, and a `int32_t` which stores the index of the currently selected shader. It returns a structure which tells you

was_selection_changed was a different shader selected;

program the program ID of the selected shader program;

name the name you gave the shader when registering it with the `ShaderProgramManager`, and which is being displayed in the combo box.

1 Extending `createSphere()`

In case you did not compute the texture coordinates for your parametric sphere in the previous assignment, now is the time.

Exercise 1:
Compute the texture coordinates in `parametric_shapes::createSphere()` and upload them to GPU memory; do not forget to extend the size of your buffer object if no memory was pre-allocated for them.

Use the *texture coordinates* shader at your disposition (selectable from the GUI, in the "Scene controls" window) to verify that your texture coordinates (see Figure 1) are correct.

If the border between the green and the orange does not look as sharp as in Figure 1, make sure that the last vertex on a 360° horizontal (or vertical) slice

- has a texture coordinate of 1 along that axis;
- is located at the same position as the first vertex on that slice, to avoid holes;
- there is no need to connect the last and first vertex, as the resulting triangles would be infinitesimal.

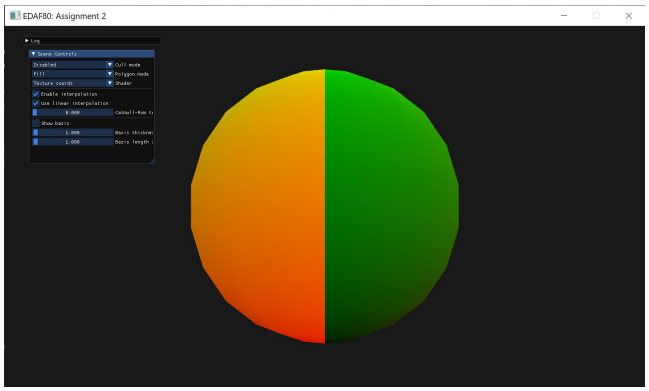

Figure 1: Sphere front with the texture coordinates shader.

2 Writing shaders

You are to implement the following shading techniques: *Phong shading*, *cube mapping* and *normal mapping*. Use as starting point any of the existing shaders, such as the *Lambert*-shader. They contain code with basic shader functionality.

For GLSL support, you are referred to the seminar and the GLSL Specification (available on the source web page).

In your vertex shader, you will automatically receive the following data as `in` (you can name your variables as you want, as long as you use the correct location):

- `vertices`: available as a `vec3` at location 0.
- `normal`: available as a `vec3` at location 1.
- `texture coordinates`: available as a `vec3` at location 2; only the x and y components contain useful data.
- `tangent`: available as a `vec3` at location 3.
- `binormal`: available as a `vec3` at location 4.

Those values are set up when creating the mesh of the shape, either in `src‣EDAF80‣parametric_shapes.cpp` or in the function `loadObjects` found in `src‣core‣helpers.cpp`.

In both your vertex and fragment shaders, the following data will be automatically set by the `Node::render()` function (found in `src‣core‣node.cpp`) as `uniform` (you have to use the following variable names):

- `vertex_model_to_world`: available as a `mat4`.
- `normal_model_to_world`: available as a `mat4`.
- `vertex_world_to_clip`: available as a `mat4`.
- `has_textures`: available as an `int`, set to 1 if textures are set, 0 otherwise.
- `has_diffuse_texture`: available as an `int`, set to 1 if you attached a `diffuse_texture` to your node object.
- `has_opacity_texture`: available as an `int`, set to 1 if you attached a `opacity_texture` to your node object.
- all textures you attached to your object, available under the name you specified as first argument to `Node::add_texture`.

Figure 2: Using the *NissiBeach2* cubemap as an environment map.

Figure 3: Rendering the demo sphere using Phong shading with the *leather_red_02_*_2k.jpg* textures.

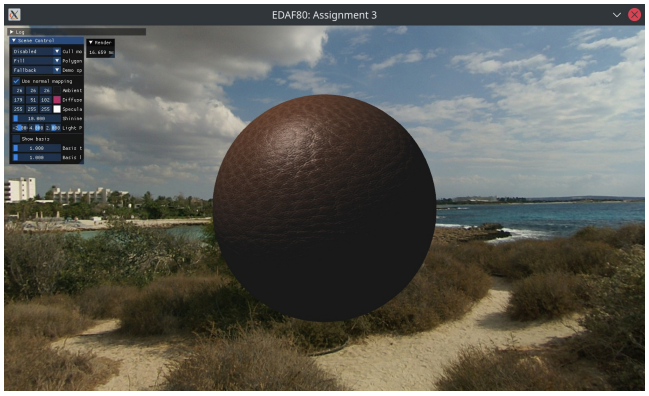

Figure 4: Rendering the demo sphere using Phong shading with the *leather_red_02_*_2k.jpg* textures and normal mapping enabled.

2.1 Cube mapping

Exercise 2:

1. Fill in the `loadTextureCubeMap()` function in `src` ▸ `core` ▸ `helpers.cpp` so that all six faces of the cube map are properly set up; the code contains comments about what you need to do there.
2. Implement a skybox (i.e. applying a cube map on a very large sphere to represent the environment far away; the camera should be placed inside that sphere) in `shaders` ▸ `EDAF80` ▸ `skybox.vert` and `shaders` ▸ `EDAF80` ▸ `skybox.frag`.

   Attention A cubemap is not sampled using the regular texture coordinates (found at location 2 in your vertex shader), but instead requires a `vec3` (a 2-D texture is sampled using a 2-D vector, and a 1-D texture is sampled with a simple scalar). Think about which 3-dimensional coordinate you could use to map a given texel of the cubemap to a given location on the sphere.

   You can find several cubemaps to choose from in the `res` ▸ `cubemaps` folder; the `res` ▸ folder can be found in the same folder as the `src` ▸ and `shaders` ▸ folders, however it might not be listed within the file explorer view of your IDE in which case you should instead use your operating system's native file explorer, like *Windows Explorer* on Windows, *Finder* on macOS, or *Dolphin* or *Nautilus* on GNU/Linux.

   In Figure 2, you can see how it would look like using the *NissiBeach2* cubemap.

2.2 Phong shading and normal mapping

Exercise 3:

1. Implement phong shading in `shaders` ▸ `EDAF80` ▸ `phong.vert` and `shaders` ▸ `EDAF80` ▸ `phong.frag`.

   For comparison, you can look at Figure 3 which demonstrates Phong shading with `textures` ▸ `leather_red_02_coll1_2k.jpg` as diffuse texture and `textures` ▸ `leather_red_02_rough_2k.jpg` as specular map. The other parameters are the default ones, but you can always zoom on the picture to check what they are as they are visible via the GUI.
2. Add normal mapping to your phong shader. You can use in your shader the variable `use_normal_mapping` to decide whether to apply normal mapping or not, and then toggle the effect on and off from the GUI to more easily check its impact.

   On top of the setup presented for the Phong shading, `textures` ▸ `leather_red_02_nor_2k.jpg` was used as a normal map to give Figure 4 which adds normal mapping to the Phong shader.

   In the `res` ▸ `textures` folder, you can find different sets of texture that have a diffuse, normal map, and roughness (you can use it as a replacement of the specular colour) texture.

3 Suggestions and things to ponder

- What are the specific roles of the Phong shading parameters? Examine how each and every parameter influences the final appearance of the object.
- Two of the features involved when creating a texture object are related to filtering and wrapping:
  - Look in the `loadTexture2D()` functions in `src` ▸ `core` ▸ `helpers.cpp` for the `glTexParameteri()` function. This function sets one of many texture parameter values. The second argument is the parameter, and the third is what that parameter is set to.
  - Consider what the `GL_TEXTURE_WRAP_S` parameter does? How can textures be made to repeat or crop on a surface by exploiting this feature (assuming access to the texture coordinates)?
  - Consider what the `GL_TEXTURE_MAG_FILTER` parameter does? Compare the texture filtering techniques using `GL_NEAREST` and `GL_LINEAR`. How and when do the end results differ?

Table 1: Various controls when running an assignment."Reload the shaders" is not available in assignments 1 and 2 of EDAF80, while "Toggle fullscreen mode" is missing from assignment 2 of EDAN35.

Action	Shortcut
Move forward	W
Move backward	S
Strafe to the left	A
Strafe to the right	D
Move downward	Q
Move upward	E
"Walk" modifier	⇧
"Sprint" modifier	Ctrl
Reload the shaders	R
Hide the whole UI	F2
Hide the log UI	F3
Toggle fullscreen mode	F11

Check out the documentation for the `glTexParameter` function on the OpenGL documentation webpage: http://docs.gl/gl4/glTexParameter.

- Normal mapping creates the illusion of an uneven surface without actually altering the geometry. When, would you argue, is or isn't normal mapping a suitable replacement for increased geometric detail?

4 Common causes of errors

Incorrect tangent space vectors Ensure that the tangent space vectors — notably the normal — are correctly constructed during tessellation, lest shader computations are bound to be erroneous.

Wrong space Ensure that points and vectors are transformed from and into the correct space (world, TBN, etc.). Vector operations make no real sense if the vectors are expressed in different spaces.

Wrong direction Define vectors in consistence with the model you are applying them to (Phong, for instance).

Vector normalization Remember to normalize where needed. Once normalized, vectors will yield normalized cross-products, reflections etc. Also note that `in`/`out`-vectors that are normalized in the vertex shader need to be re-normalized in the fragment shader.

A Framework controls

The framework uses standard key bindings for movement, such as W, A, S, and D. But there are also custom key bindings for moving up and down, as well as controlling the UI. All those key bindings are listed in Table 1.

There is only one action currently bound to the mouse, and that is rotating the camera. To do so, move the mouse while holding the left mouse button.

GUI elements can be toggled being a collapsed and expanded state by double clicking on their title bar. And they can be moved around the window by dragging their title bar wherever desired (within the window).

B IDE key bindings

To help with getting certain tasks done more efficiently, Table 2 lists key bindings of different IDEs for several common actions.

Table 2: Various keyboard shortcuts for Visual Studio 2019 and 2017, and Xcode.

Action	Shortcut	
	Visual Studio	Xcode
Build	Ctrl + B	⌘ + B
Run (with the debugger)	F5	⌘ + R
Run (without the debugger)	Ctrl + F5	
Toggle breakpoint at current line	F9	⌘ + \
Stop debugging	⇧ + F5	⌘ + .
Continue (while in break mode)	F5	ctrl + ⌘ + Y
Step Over (while in break mode)	F10	F6
Step Into (while in break mode)	F11	F7
Step Out (while in break mode)	⇧ + F11	F8
Comment selection	Ctrl + K, Ctrl + C	⌘ + /
Uncomment selection	Ctrl + K, Ctrl + U	⌘ + /
Delete entire row	Ctrl + X	