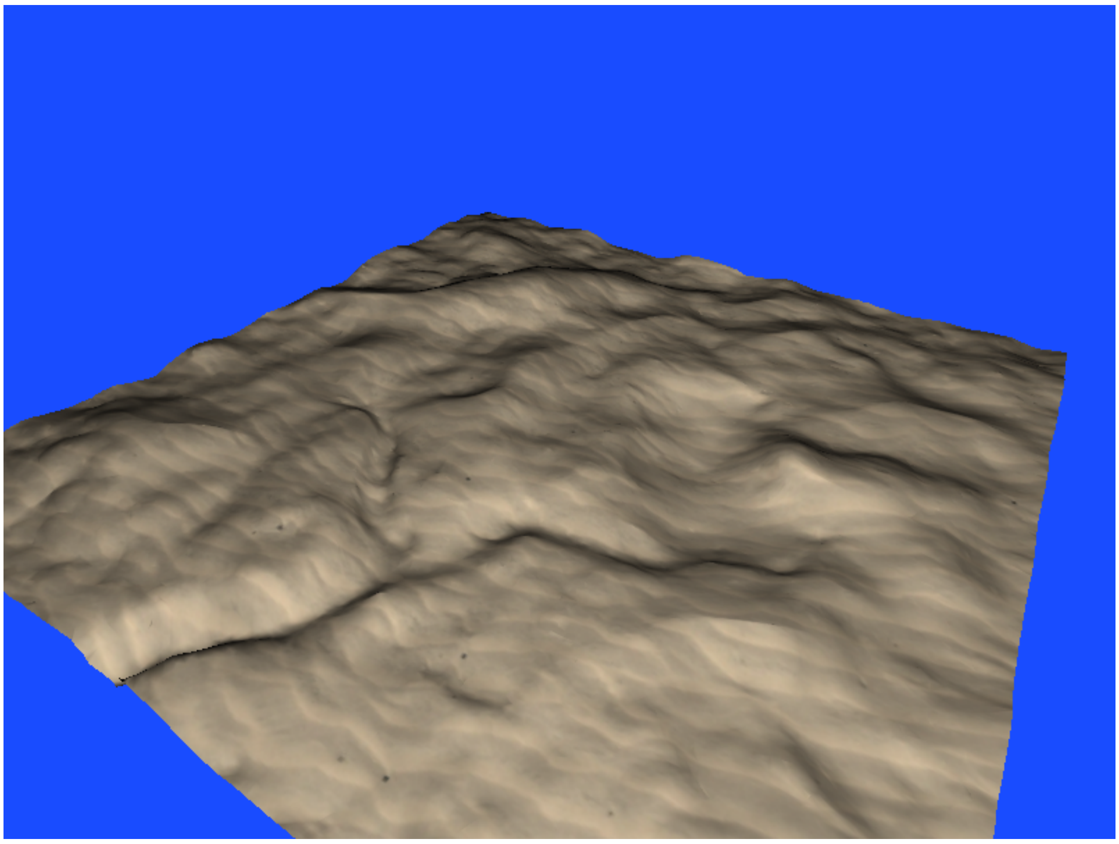

Assignment 4 — 3D object

Deadline: 2016-12-18 23:59

To go beyond simple objects, it will be necessary to load and display arbitrary geometry, created, e. g., using modeling software. The goals of this exercise are to explore how a 3D object can be stored in and loaded from a file by example of Wavefront OBJ,[1] a very common model file format, and to learn how to use textures.

[1]http://en.wikipedia.org/wiki/Wavefront_.obj_file

Task 4 (20 points)

Extend the application «`task4`» such that it reads a 3D object from an OBJ file and renders the object with a texture applied. You should be able to reuse and build upon parts of your solution to the previous exercise, e. g., the camera setup and shaders.

- Load a simple three-dimensional object.
  - Read geometry including normals and texture coordinates from an OBJ file. You do not need to support the full feature set of OBJ. Handling of erroneous files or files with contents other than a single triangle mesh is not required. Your application may treat any file it encounters as if it contains only a single triangle mesh.
  - Process and store the geometry in buffer objects suitable for rendering using OpenGL.
  - Set up an appropriate vertex array object describing your vertex data layout.
- Load a texture.
  - Load image data from a PNG file. You can use the facilities provided by the framework through the `PNG::loadImage2D()` function[2] to take care of reading a PNG file for you.
  - Create a two-dimensional, single-level OpenGL texture using the image data read from the file.
- Write a vertex shader that
  - maps the object to the screen according to a given view, and projection matrix as well as
  - outputs normal vectors and texture coordinates for the fragment shader.
- Write a fragment shader that
  - implements simple lambertian shading as in the previous task
  - samples the texture loaded from the PNG file and uses the color obtained from the texture instead of a constant color as diffuse reflectivity $\mathbf{c}_d$ in the shading

[2]As described in the framework documentation available on the course website.

computation.

- Set up a camera looking at your object.
- Draw the object.

A number of OBJ files and textures can be found in the directory «rtg/assets».

4.1 The OBJ file format

OBJ files are simple text files. The following listing shows an example OBJ file containing a cube:

```
# cube.obj

v -1.0 -1.0 -1.0     # back lower left
v -1.0 -1.0  1.0     # front lower left
v -1.0  1.0 -1.0     # back upper left
v -1.0  1.0  1.0     # front upper left
v  1.0 -1.0 -1.0     # back lower right
v  1.0 -1.0  1.0     # front lower right
v  1.0  1.0 -1.0     # back upper right
v  1.0  1.0  1.0     # front upper right

vn  0.0  0.0  1.0
vn  0.0  0.0 -1.0
vn  0.0  1.0  0.0
vn  0.0 -1.0  0.0
vn  1.0  0.0  0.0
vn -1.0  0.0  0.0

vt 0.0 0.0
vt 0.0 1.0
vt 1.0 0.0
vt 1.0 1.0

# back face
f  1/4/2  7/1/2  5/2/2
f  1/4/2  3/3/2  7/1/2
# left face
f  1/2/6  4/3/6  3/1/6
f  1/2/6  2/4/6  4/3/6
# top face
f  3/1/3  8/4/3  7/3/3
f  3/1/3  4/2/3  8/4/3
# right face
```

```
f  5/4/5  7/3/5  8/1/5
f  5/4/5  8/1/5  6/2/5
# bottom face
f  1/2/4  5/4/4  6/3/4
f  1/2/4  6/3/4  2/1/4
# front face
f  2/2/1  6/4/1  8/3/1
f  2/2/1  8/3/1  4/1/1
```

Each line of text in an obj file forms a so-called *statement* which defines object data. Every line starts with a *keyword* identifying the type of data that follows, delimited by whitespace. There are two types of data found in the above sample OBJ file. All statements starting with the character `v` define vertex data, while the statement `f` specifies a polygon (“face”):

`v` defines a vertex position. Following `v` are exactly three floats that specify the x, y, and z coordinate of a vertex position.

`vn` defines a vertex normal. Following `vn` are exactly three floats that specify the x, y, and z coordinate of a vertex normal.

`vt` defines a pair of texture coordinates. Following `vt` are exactly two floats that specify the s, and t texture coordinate.

`f` defines a polygon. Following `f` comes an arbitrary number of 3-tuples of indices that reference the vertex position, texture coordinates, and vertex normal that should be used to form a vertex of the polygon. You only need to handle the case of triangles, i. e., exactly three vertices. Individual components are separated b a `/` character. Indices start at 1!

The `#` character starts a comment. When a `#` is encountered, the `#` and all following characters are ignored until the beginning of a new line. For simplicity, we recommend you use the standard C++ stream classes (`std::ifstream` in combination with the stream input operator >>) for parsing OBJ files in C++.

4.2 Textures in OpenGL

An OpenGL Texture Object is created by first allocating a *name*, i. e., a texture handle, using `glGenTextures()` and then binding this name to a texture *target* (e. g., `GL_TEXTURE_2D`) using `glBindTexture()`. Using `glTexStorage2D()`, you can allocate a texture image for the bound texture object and then upload image data using `glTexSubImage2D()`. Alternatively (especially if you’re using an older graphics driver that does not support `glTexStorage2D()`), you can use `glTexImage2D()`.

After a texture has been created, shaders can *sample* the texture to obtain a filtered color value for a given location specified in texture coordinates. In GLSL, a texture is represented by a so-called *sampler*. To sample a texture, we use the function `texture()` which returns a `vec4` containing the sampled RGBA values:

```
#version 330

uniform sampler2D my_texture;

in vec2 tex_coords;
layout(location = 0) out vec4 color;

void main()
{
  color = texture(my_texture, tex_coords);
}
```

More than one texture can be in use at the same time. Whenever an application binds a texture object using `glBindTexture()`, the texture is bound to an *active texture unit*. There is a limited number of active texture units, each identified by an index. To specify, which active texture unit should be used, call `glActiveTexture()` before `glBindTexture()`. For example, the following code would bind `my_texture` to the active texture unit with index `i`:

```
glActiveTexture(GL_TEXTURE0 + i);
glBindTexture(GL_TEXTURE_2D, my_texture);
```

A GLSL sampler uniform variable can be set by the application to an integer value selecting the active texture unit to sample from, e. g., using `glUniform1i()`.

Submission format

Hand in your solution in the form of a ZIP archive using our submission system.[3] Once uploaded, your submission will automatically be built and run on our reference system. You receive an email notification as soon as the results of this automatic test run are available.

Your submission will be unpacked over an unmodified version of the framework. Therefore, only files you modified or added[4] need to be included. Make sure to keep the directory structure in your submitted archive the same as in the framework (root directory: «`rtg`»),

[3] `https://courseware.icg.tugraz.at`

[4] If you modify any of the CMake scripts, remember to also include them in your submission, otherwise the build will fail.

otherwise, automated unpacking and reassembly will fail. You can assume, e. g., for the sake of loading assets from files, that the initial working directory of your application is the «`rtg`» root directory.

Words of wisdom

- Don't forget that indices start at 1 in OBJ.
- Be aware that the `sampler2D` uniform variable does not hold the texture name itself, but rather the index of the active texture unit the texture is bound to.
- Take advantage of the excellent OpenGL online manual[5].
- Make sure to check out the tutorial on error handling
- Do not forget to display your rendered image using the `swapBuffers()` method of the `context`.
- Consulting online tutorials, you may often see the `glm` library referenced. This library is not installed on the test server and must not be used for these assignments. The in-house math framework already supports many vector and matrix operations. However, in contrast to `glm`, it does not provide methods for setting up a particular matrix, e.g. for projection. This is intentional, since the assignments are in part about understanding the composition and the effects of these matrices, and how their manipulation affects the eventual image.

[5]`http://www.opengl.org/sdk/docs/man/`