Project 3 - Three textures

In this project, you'll be applying textures to 3D models. The project has three separate parts and each part is about a different kind of texture. You'll apply a 3D marble texture and a 2D spherical environment map to an arbitrary model (with spherical environment map, the model should look like a mirror or an object made of the same material as the sphere shown in the environment map texture) and produce an image of a doughnut with a pattern given by a tilable texture.

For full credit **you have to incorporate illumination into the 3D texture and doughnut images**. Feel free to experiment with it. What we would like to see is not just texture lookup, but also illumination in your images. A simple way to do this is to evaluate the Phong's intensity formula somehow with some scalar values of k_a, k_d and k_s (either per fragment or per vertex) and use this intensity to scale the color looked up from the texture. See the sample output section on Blackboard for examples.

1 3D texture

We are providing one $128 \times 128 \times 128$ texture, representing a marble-like material, generated using noise functions. Your goal is to apply this texture to an arbitrary 3D model. The model should appear as if it was carved from the block of material with colors defined by the texture. The size of the object must be comparable to the size of the block.

2 Environment mapping

We are providing a few photographs of the light probe, you need to apply the images as textures so that the model looks as if it was a mirror object in the corresponding environment. Note that the camera should be fixed (with respect to the environment) as the model rotates.

3 Tilable texture

Draw a nice brick or wooden doughnut using one of the provided textures.

4 Display of the results

You can either cut and paste code from your 3D viewer, allowing the user to use trackball interface to rotate the textured model or just rotate the model as the sample code does.

5 Menu

Implement the following menu items:

Environment mapping if selected, the model should show up with environment mapping

3D texture if selected, the model should show up with 3D texture applied

Doughnut If selected, show textured doughnut; use one of the tilable textures

Zoom in zoom in

Zoom out zoom out

Faster animate faster (say, about 30% faster); don't include this option if you use trackball interface

Slower animate slower (again, about 30%); don't include this option if you use trackball interface

Stop/Run toggle between animation and stationary image. The default should be no animation: when we start your programs, the displayed model should be stationary; don't include this option if you use trackball interface

6 Texture formats

The textures will be given in the raw format. The environment maps and the tilable textures are binary PPM images with no comment lines.

The 3D/volumetric texture is given in the raw format, i.e. it just contains RGB values of consecutive pixels/voxels (consecutive in the lexicographical order). The resolution is $128 \times 128 \times 128$.

Your code should read the input model (to be used with the 3D texture and the environment map) from `input.t` and textures from files with the following names (case sensitive!):

(i) `envmap.ppm` for the environment texture

(ii) `tilable.ppm` for the doughnut texture

(iii) `marble.rgb` for the marble texture.

When we test your code, we will unpack and compile it and move the texture files, the mesh file (`input.t`) and your executable to the same directory. Finally, we'll run the executable. Note that the environment map will look best on high resolution models such as `horse2.t`, `bunny2.t` or `feline2.t`.

7 Comments on sample code

The sample code should make it easy to deal with textures. At least initially, take as much advantage of it as possible. The sample code is very similar to the sample code for project 2. It adds texture related classes, wrappers of OpenGL texture objects. As with all other classes provided here, don't attempt to copy or assign objects of these types - use only pointers to them in your code. If you'd like to see implementations of the texture related classes and function, look at the `texture.[ch]` files.

7.1 Using textures in the shaders

Textures are represented as uniform sampler variables in shaders. There are a number of different sampler types in GLSL. Here, we will use only the basic 2D and 3D samplers. As the name suggest, the basic operation on samplers is sampling. You can sample a 2D sampler at any 2D point and a 3D sampler at any 3D point. Sampling means texture lookup. Here are example declarations of sampler uniforms:

```
layout (binding=1) uniform sampler2D tex1;
layout (binding=2) uniform sampler3D tex2;
```

The binding parameter must match the texture attachment point (TAP) the corresponding texture is plugged into. In this particular case, `tex1` will represent the 2D texture attached to TAP #1 and `tex2` - the 3D texture attached to TAP # 2. To attach a texture to a TAP in the CPU code, use the `attach` method described below in Section 7.2.2.

To look up values (here: color) from the samplers, use the `texture` function. There are two overloaded versions you need:
`texture(sampler3D tex,vec3 tcoord)` looks up color from 3D sampler and
`texture(sampler2D tex,vec2 tcoord)` looks up from 2D sampler.
In the case of simple samplers used here, lookup is the same as texture lookup described in class. Note that the return value in both cases is of type `vec4` (the function returns RGBA values). Most likely, you will want to use a swizzle operation to extract the RGB components. For example (see the fragment shaders provided with the sample code), if `fragcolor` is the output variable of the fragment shader of type `vec3`, you may want to do this:

```
fragcolor = texture(tex,tcoord).rgb;
```

7.2 Useful texture related classes

It is sufficient to use RGB textures in this project (i.e. texture with values being colors in the RGB format). The wrapper classes for 2D and 3D texture objects are `RGBTexture2D` and `RGBTexture3D`.

7.2.1 Reading textures from files and creating texture objects

The sample code provides convenient functions to read the textures from files and create objects that represent texture object. For 2D textures in PPM format, use the `createRGBTexture2D` function. The first argument is the name of the PPM file containing the texture. The following 3 optional arguments are R, G and B values for the border color. Border color can be used to create nice borders around images (see sample code). In your project code, you won't need to deal with borders so you can just use the file name as the only argument to `createRGBTexture2D`. The return value is of type `RGBTexture2D*`.

There is a similar function for creating 3D texture objects, `createRGBTexture3D`. The arguments are the x-, y- and z- resolutions of the texture and the file name.

The sample code has global variables `t2D` and `t3D` for storing the 2D and 3D texture objects (you will need three to complete the project - two for tilable and environment map textures and one for the 3D textures). They are declared as follows:

```
RGBTexture2D *t2D = NULL;
RGBTexture3D *t3D = NULL;
```

Values are assigned to these variables in the `setup_textures` function like this:

```
  t2D = createRGBTexture2D("textures/mines.ppm",0.5,0.25,0.25);
  t3D = createRGBTexture3D(128,128,128,"textures/marble.rgb");
```

7.2.2 Methods of the texture classes

`RGBTexture2D` and `RGBTexture3D` provide the following methods. You'll really need only `attach` and `on` to complete the project, but feel free to experiment with other ones.

The `attach` method attaches the texture object to a texture attachment point given by the value of its argument. For example, in the sample code we have:

```
t2D->attach(1);
t3D->attach(2);
```

which means `t2D` is plugged into TAP No. 1 and `t3D` - into TAP No. 2. Please use small numbers for TAPs - say, up to 10 or so. The sample code does not implement any check for bounds and makes assumptions that are guaranteed only for small TAP numbers. The binding parameter of uniform sampler variables in the shader code has to match. For example, to use the 3D texture in any of the shaders, you need to do something like this:

```
layout (binding=2) uniform sampler3D tex;
```

And for the 2D texture - this:

```
layout (binding=1) uniform sampler2D tex;
```

The argument-less *on* method turns the texture on. Similarly, *off* method turns the texture off. In the project, you can attach all textures to different TAPs, turn them on and never turn them off. Note that any changes to the texture object turn it off. To make things work, call the *on* method as the last method for any texture object before using it for rendering (see sample code).

To switch to multi-linear and nearest neighbor interpolation, use argument-less `linear` and `nearest` methods (respectively). Linear interpolation is the default one, so in the project you don't have to call it explicitly.

`clampToEdge`, `clampToBorder` and `repeat` (all of them argument-less) control lookups with out-of-bounds texture coordinates. `clampToEdge` clamps the coordinates to $[0, 1]$. `clampToBorder` causes the lookup to return the border color if any texture coordinate is not in $[0, 1]$. `repeat` repeats the texture.

Finally, you can use the set the border color with the `setBorderColor` method (its arguments are RGB values, floats in the range $[0, 1]$).

8 Grading

Each part is worth the same, 100/3 per cent of the score. As usual, the project is individual, you are welcome to discuss high level approaches or ideas, but the code should be your own. Result of texture lookup must be modulated with illumination for the marble texture and for the doughnut. Place the light somewhere above the light source (not too high) and don't use light source attenuation to make finding eye-pleasing parameters easier.