Lab 7 - Texture and normal mapping, environmental and refraction

Laura Mazzuca - matr. 0000919489

16/09/2021

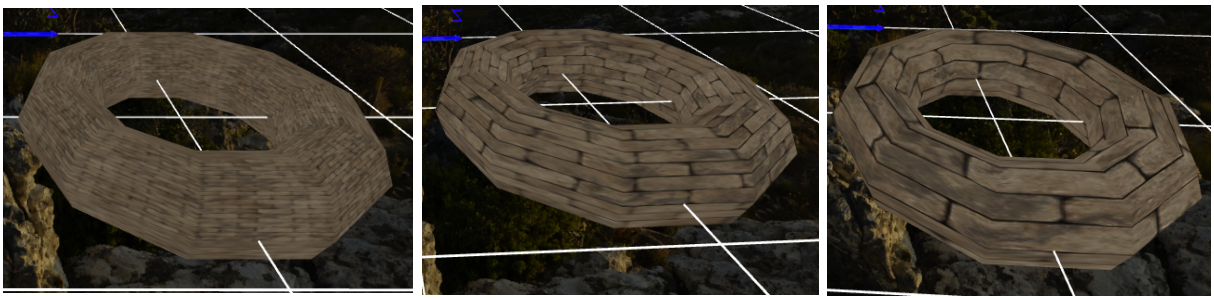
0 Assignments

1. Torus mappings
   (a) texture mapping 2D on the torus
   (b) texture mapping 2D + Phong shader on the torus
   (c) procedural mapping on torus
2. Normal mapping
3. Environment mappings
   (a) skybox
   (b) reflection
   (c) refraction
4. semi-transparent objects

1 Torus mappings

1.1 Texture Only

To apply a Texture Only Shader to the torus the mapping of the texture coordinates was added to the $computeTorusVertex()$ function. This mapping is done assigning to each vertex texture coordinates of the torus a certain pair of values. Depending on the texture coordinates, the texture will look either more stretched or compressed on the torus' surface. In Figure 1 we can see different results for different texture coordinates.



(a) (θ, ϕ) (b) $(\theta/\pi, \phi/\pi)$ (c) $(\theta/2 * \pi, \phi/2 * \pi)$

Figure 1: The different looks of the torus' surface depending of the texture coordinates values.

1.2 Phong Texture Shader

To apply a Phong Texture Shader the **f_texture_phong.glsl** was implemented. This implementation only required to copy-paste the lab-03 Phong shader and change the $material.diffuse$ parameter with the function to obtain the rgb of the texture in this fragment.

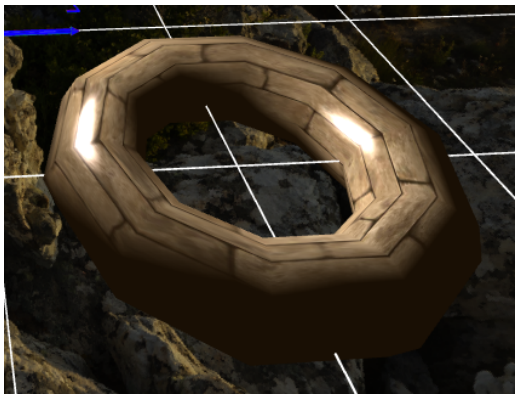


Figure 2: The result of the Texture Phong shader on the Torus to which was assigned the material BRASS.

1.3 Procedural

The procedural texture was generated as Perlin noise **compute_torus_procedural_texture()**. All the functions to generate the noise have been implemented following closely Perlin's original implementation[1]. In generating the texture, a problem came up where the result would be a completely black texture. To solve this, a frequency of 0.01 has been applied to the coordinate values representing the pixels of the texture[2].

[1] Perlin's website.

[2] as suggested in this article, at the bottom of its discussion.

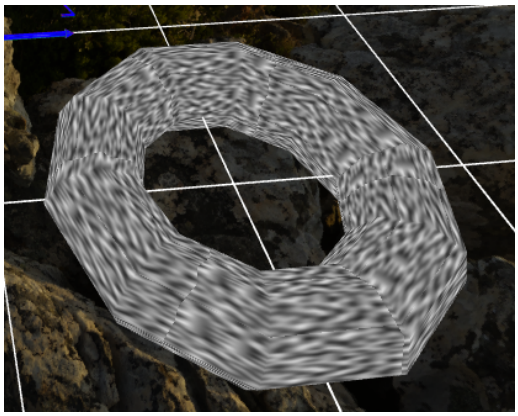
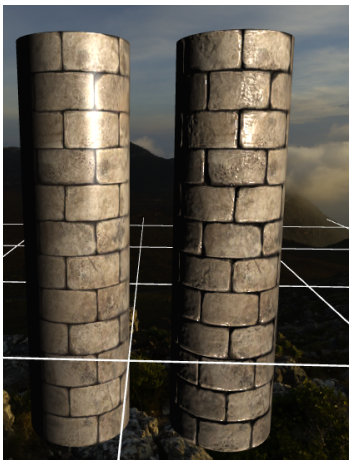


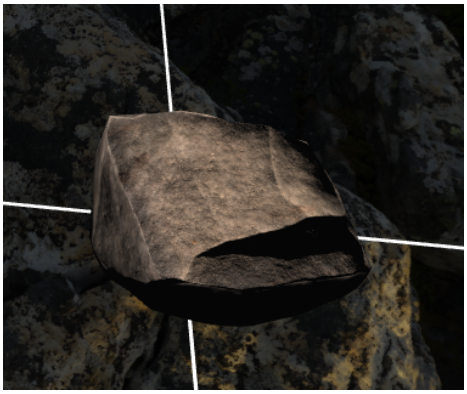
Figure 3: The result of the procedural texture mapping done with Perlin noise.

2 Normal

To implement the normal mapping, the Tangent, Bi-tangent and Normal inverse matrix, defined in the **v_normal_map.glsl** are used to translate world space vectors into tangent space, then forwarded to the fragment shader **f_normal_map.glsl**. In the ladder, the phong algorithm was implemented by sampling the normal from the Normal Map and sampling the color from the Texture Map. The normal also needed to be converted from the [0,1] space to [-1,1]. The two variables were then substituted into the algorithm computing the fragment color. The visual effect was then applied to the column and the rock.



(a) On the left a column with Phong Texture Shading, on the right the column with Normal Shading.



(b) The rock with Normal Shading.

Figure 4

3 Environment mappings

3.1 Skybox

The only change that was made was to center the skybox in (0,0,0) and uncomment the *reverse* function applied to $surface.vertices$.

Figure 5: The skybox properly positioned.

3.2 Reflection

The reflection shader was implemented in **v_reflection.glsl** and **f_reflection.glsl**. In the first one the normal and the position are computed. Note that the Normal is computed only with respect to M, so in World Coordinate System, and only after the View and Projection matrices are applied. Finally, in the fragment shader we need to compute the reflection ray and assign the color to the fragment with respect to the point hit on the cube map. The result can be seen in Figure 6.

3.3 Refraction

The reflection shader was implemented in **v_reflection.glsl** and **f_reflection.glsl**. The vertex shader is the same as the reflection one, while in the fragment shader the ray is computed by ging in input to the $refract$ function the Snell ratio for materials. The result can be seen in Figure 6.

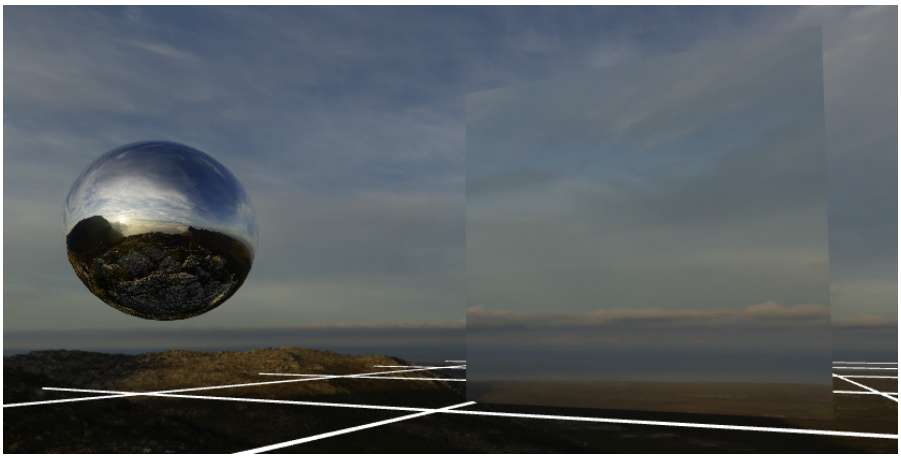
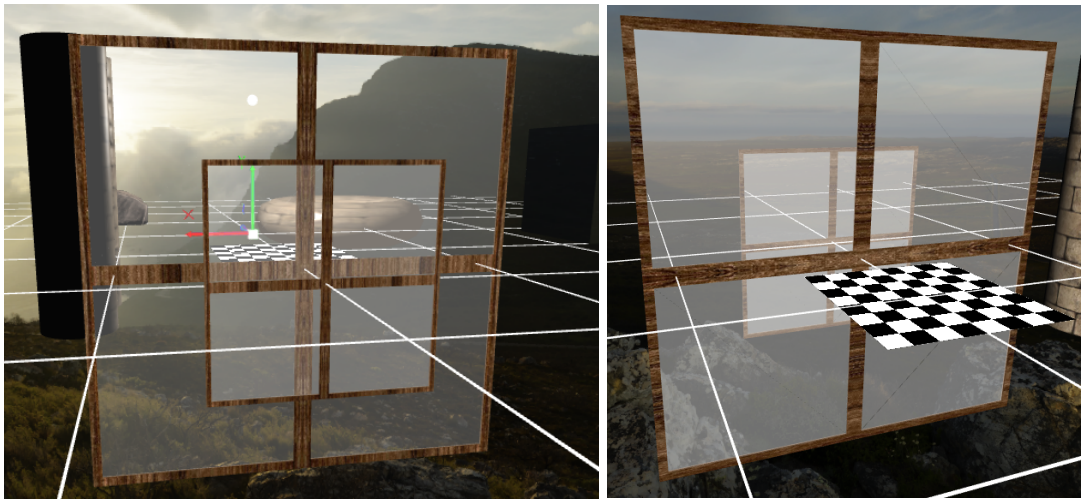


Figure 6: On the left the ball rendered with the Reflection Shader, on the right the cube rendered with the Refractive Shader.

4 Semi-transparent objects

To render semi-transparent objects, the blending in fragment rendering was enabled. To do so the $glEnable(GL\_BLEND)$ was added in the $init()$ function. Then, to render the objects in the proper order, the $drawScene()$ function was modified to first render all the non-transparent objects and then render the transparent ones in order of distance.



(a) Small window rendered in front. (b) Big window rendered in front.

Figure 7: The semi-transparent windows.