Introduction to Computer Graphics
Project Part 2 – Rendering

Andrea Tagliasacchi, Sofien Bouaziz

Handout date: Mon Apr 14
Submission deadline: Tue Apr 29 (8pm)
Late submissions are not accepted

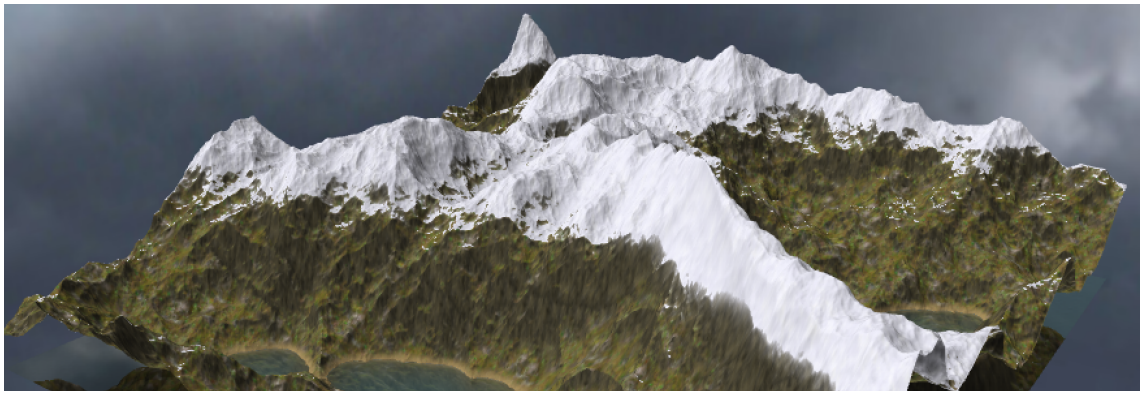


Figure 1: Procedurally generated terrain with multi-texturing and environment map.

In the first part of the project you procedurally generated a terrain geometry and built a very simple shader for texturing. In this second phase, we will develop more advanced shading techniques to achieve a cool result like the one shown in Figure 1. We have grouped the tasks in two different tiers. Completing the tasks outlined in Section 1 represents the minimal requirement to obtain a 4/6 grade. Section 2 contains several tasks you can complete to improve your grade. Finally, we invite you to take a look at Section 3, where we show you a simple C++ class that can be used to modularize your project.

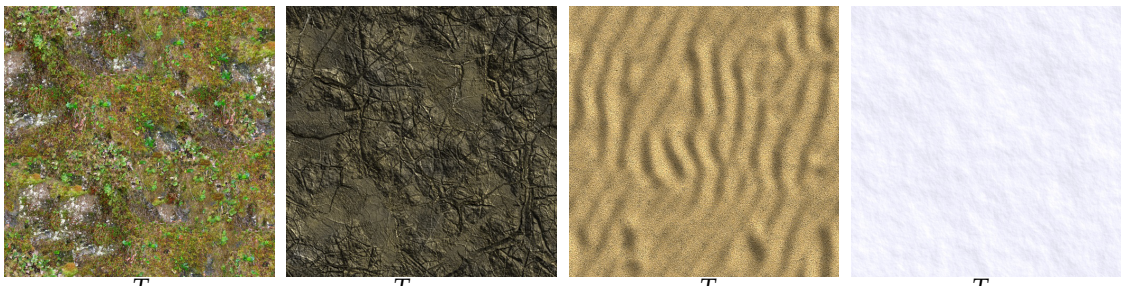
1 Basic

1.1 Texturing (blending with height+normal)

To obtain a textured terrain we will use several textures, each representing a specific material. Note that the textures in Figure 7 are *tileable*, that is, you can place two of them beside each other and there will be no discontinuities. Mathematically, the following equation fully describes the texturing for your project:

$$T(\mathbf{x}) = \alpha_1 T_{grass}(10\mathbf{x}) + \alpha_2 T_{rock}(10\mathbf{x}) + \alpha_4 T_{snow}(30\mathbf{x}) + \alpha_3 T_{sand}(60\mathbf{x}) \quad \text{s.t.} \quad \mathbf{x} \in [0,1]^2 \tag{1}$$

Above, $\mathbf{x}$ is the 2D vector of texture uv coordinates. Note that the modulation coefficients $[10,10,30,60]$ have the effect of *tiling* each texture. For example T_{rock} was tiled 10 times (in both u and v) to obtain the result you see in Figure 1. Note that when you interpolate colors you want



T_{grass} T_{rock} T_{sand} T_{snow}

Figure 2: To obtain the texturing visualized in Figure 1 you will interpolate across the textures above. We took our textures from `http://www.spiralgraphics.biz/packs/terrain_rocky/index.htm`, but we strongly encourage you to find your own.

to make sure that you have a convex combination[1]: $\alpha_i > 0$ and $\Sigma_i \alpha_i = 1$. We recommend you to approach the problem step by step. First map the *grass* texture on the whole domain.

$$T(\mathbf{x}) = T_{grass}(\mathbf{x}) \tag{2}$$

Then tile it (search online on the effects of *GL_TEXTURE_WRAP_** and *GL_REPEAT*):

$$T(\mathbf{x}) = T_{grass}(10\mathbf{x}) \tag{3}$$

After that, attempt to mix grass with *rock* – by taking in account the simple approximation that *grass cannot grow on very steep inclines*[2]:

$$T(\mathbf{x}) = (1-\alpha)T_{grass}(10\mathbf{x}) + \alpha T_{rock}(10\mathbf{x}) \tag{4}$$

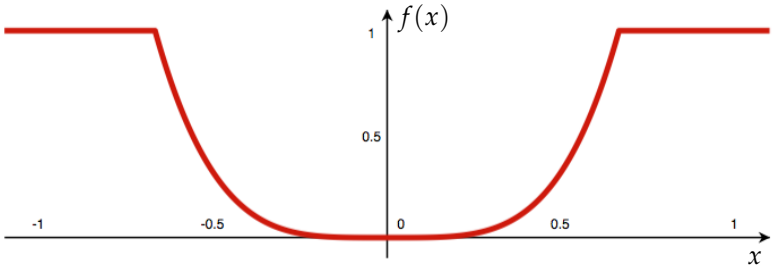


Figure 3: An example of a function used to interpolate the grass and rock texture.

The interpolation parameter $\alpha(\mathbf{x})$ is spatially varying and selected proportionally to the *surface slope*[3]:

$$\alpha(\mathbf{x}) = f(\mathbf{n}(\mathbf{x}) \cdot [0,1,0]) \tag{5}$$

Figure 3 illustrates the function we have used to interpolate across grass/rock in the rendering of Figure 1. Finally, following a very similar approach, proceed to texture *snow* and *sand*. Use the *sand* texture when you are close or below the sea-height, and the *snow* texture when the height is above a certain threshold.

Rendering the sea. To model the geometry of the sea, you can just create a second instance of the vertices/triangles you used to represent the terrain. For the moment being, create a new shader that simply colors the sea plane in blue, more advanced effects will be treated later.

[1] hint: use the GLSL functions $mix(\mathbf{c}_1, \mathbf{c}_2, \alpha)$ and $clamp(\alpha, 0, 1)$

[2] hint: blend according to the surface normal you computed by finite differences.

[3] note: our terrain lives in the $x - z$ plane

1.2 Modeling the sky

Figure 4: (left) An example of a skybox texture. (right) Using a cube for environment mapping will result in visible artifacts if not properly handled.

In order to have an interesting sky (rather than a solid color), we will extend what we have done in *practical session #6*. Enclose your entire scene (the scene we implemented lives in the $x-z$ plane and spans $[-1,1]^2$) in a large cube (e.g. $[-10,10]^3$) which you will then color using a sky texture like the one in Figure 5.

Optional. The method above is quick to implement as it reutilizes code from *practical session #6*. However, the texture map in Figure 5 has discontinuities, and this results in visible seams in the sky. A possible fix to this problem consists in adding some padding with similar colors in the black regions. However, OpenGL offers specific functionality supporting this type of task[4].

1.3 Self shadowing

A terrain without shadows is boring! Compute shadows by following the approach we presented in the *practical session #6*: in a first pass, render the scene from the point of view of the light source; then compute the depth value of the pixel in the light coordinate system. If the distance from the camera is above the one measured in the depth buffer of the first pass, *darken* the color obtained by regular shading.

Implement a simple control of the light position by binding the $[1-9]$ keyboard keys[5]. This will be used to test/debug your shadows. If the terrain is defined in the $x-z$ plane, then your light will be defined in the $x-y$ plane. In this setup "1" should be sunrise (light is located on the $x-z$ plane) and "9" should be noon (light located along the $+y$ axis). Keep it simple, more advanced lighting control will be one of the advanced tasks.

[4] `http://www.opengl.org/wiki/Cubemap_Texture`
[5] GLFW2 documentation `http://www.glfw.org/GLFWReference27.pdf` pg. 30/64

2 Advanced

2.1 Approximating water reflections/refractions

To enhance the quality of water bodies a possibility is to efficiently model their reflective/refractive behavior. Its *reflective* component can be modeled by flipping the camera position w.r.t. the sea-level line and then storing the rendered image in a texture $T_{reflective}$. Conversely, for its *refractive* (i.e. transparent) component, we will simply assume that the light rays are not affected by the change of density[6]. $T_{refractive}$ can simply be generated by rendering the world *without* the water geometry.

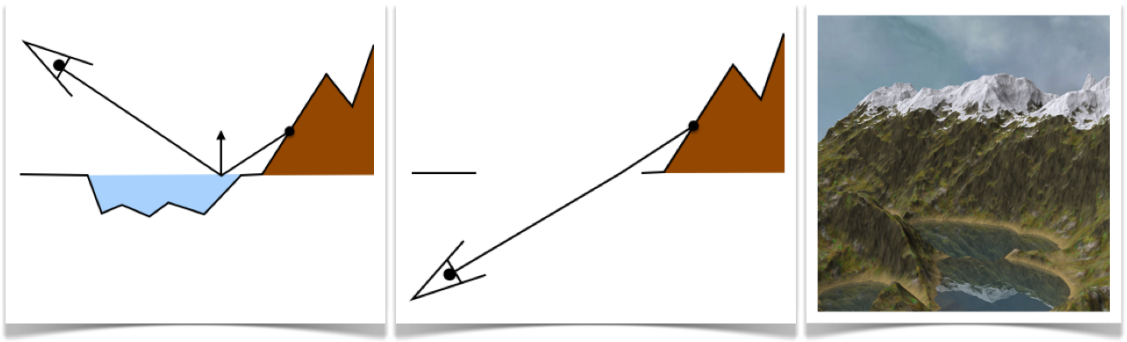


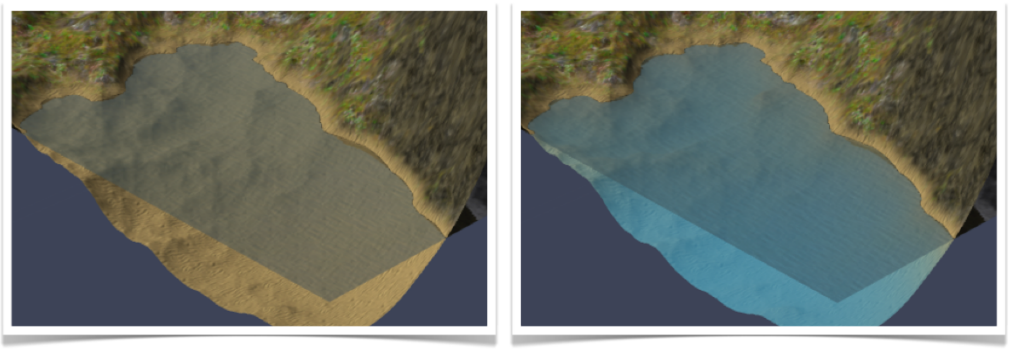
Figure 5: (left) the reflection of a view ray on a specular surface. (right) the same result can be computed by mirroring the camera position w.r.t. the reflective surface and rendering the resulting scene; note that the geometry below the "sea level" needs to be clipped. (right) an example of our real-time rendering of water bodies.

Summing up, the color at pixel $\mathbf{x}$ can be obtained as a simple weighted combination:

$$color(\mathbf{x}) = \alpha T_{reflective}(\mathbf{x}) + (1 - \alpha) * T_{refractive}(\mathbf{x}) \tag{6}$$

where $\alpha(\mathbf{x})$ is a function of the angle in between view direction $\mathbf{l}(\mathbf{x})$ and and surface normal $\mathbf{n}(\mathbf{x})$ measured at the point $\mathbf{x}$. Design a function in such a way that when you look at the water straight from above it will appear transparent (i.e. $\alpha = 0$), while when you look at the water from a grazing angle it will behave like a reflective surface (i.e. $\alpha = 1$).

2.2 Water depth effect (participating media)



Extend the shading from the previous section by modifying $T_{refractive}$ in such a way to approximate a participating media effect[7] like the one you see in the image above. This can be easily achieved by blending the pixel color in $T_{refractive}$ with a target color (blueish) according to depth (y terrain coordinate).

[6] i.e. we assume Fresnel coefficient one `http://en.wikipedia.org/wiki/Fresnel_equations`.
[7] `http://graphics.ucsd.edu/~henrik/images/parti.html`

2.3 Water dynamics

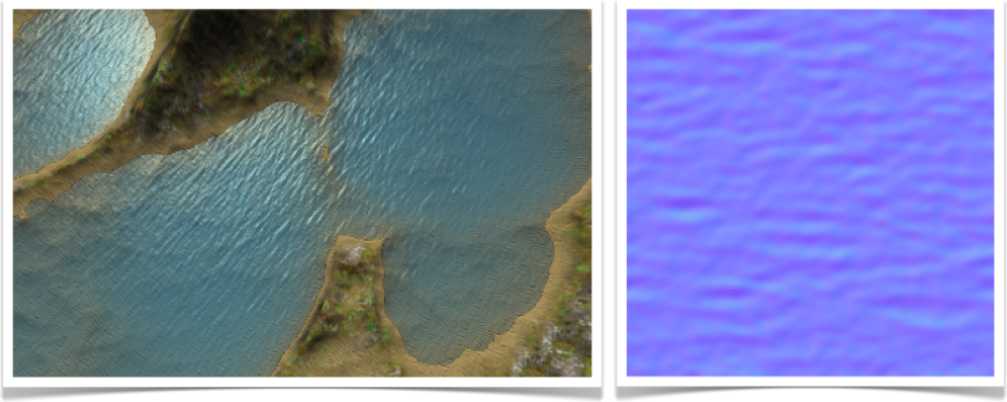


Figure 6: (left) The normal map applied to the water geometry. (right) The normal map is assembled by three appropriately rotated, scaled and translated version of this texture.

A more interesting appearance for water can be obtained by applying a *normal map* to its surface. The image above shows a snapshot and the normal map (a texture whose RGB components represent normal orientations) that was used to generate the effect, while a brief video can be downloaded here: `http://lgg.epfl.ch/SHARE/waves.mp4`. The dynamic effect in the video can be obtained by translating the texture coordinates in time. Note that the geometry of the water is completely flat! The shading, both Lambertian and specular highlights, are the result of using the normals from the normal map instead of the ones of the sea geometry (which are $[0,1,0]$ everywhere).

2.4 Time of the day

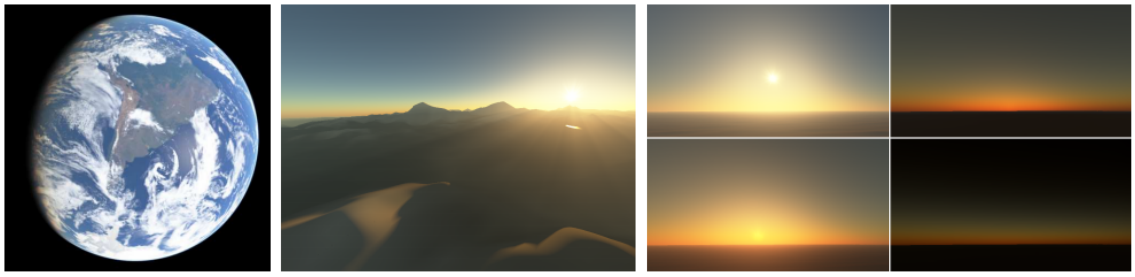


Figure 7: The sky can be modeled by taking into account scattering of light in the atmosphere.

An alternative to using a *skybox* to model the environment is to accurately model the *scattering effect* of the sun rays into the atmosphere. Below you find some tutorials to get you started in this direction. Possible extensions include adding stars at nighttime and an animation of the time of the day / sun position (allow the time speed to be controlled from the keyboard).

- `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter16.html`
- `http://vterrain.org/Atmosphere`

3 Modularizing your code

A vertex array object holds references to the vertex buffers, the index buffer and the layout specification of the vertex itself. At runtime, you can just *glBindVertexArray* to recall all of these information. Below we provide a simple example of a class that you can use as a template to modularize your code. This is an excellent starting point for the skybox portion of the project[8].

```
#pragma once
#include "common.h"          ///< LGG common header (libraries, etc)
#include "cube.h"            ///< cube vertex/texture coordinates
#include "skybox_vshader.h" ///< stringified vertex shader
#include "skybox_fshader.h" ///< stringified fragment shader

/// @brief OpenGL wrapper class template
class Skybox{
private:
    GLuint skybox_vao;         ///< Vertex array objects
    GLuint skybox_program_id; ///< GLSL program ID
    GLuint skybox_tex;         ///< Texture IDs

public:
    void init(){
        //--- Skybox stuff stored in a vertex array
        glGenVertexArrays(1, &skybox_vao);
        glBindVertexArray(skybox_vao);

        //--- Load textures from file
        skybox_tex = loadTGA("skybox.tga");

        //--- Compile shaders
        skybox_program_id = compile_shaders(skybox_vshader, skybox_fshader);

        //--- Generate vertex buffers & attributes
        // glGenBuffers                 <--- generate buffer pointer
        // glBindBuffers                <--- make it current
        // glBufferData                 <--- tell it where to find data
        // glGetAttribLocation          <--- fetch attribute ID
        // glEnableVertexAttribArray    <--- make it the current
        // glVertexAttribPointer        <--- specify layout of attribute
    }

    void draw(mat4& projection, mat4& model_view){
        //--- Tell which shader we want to use
        glUseProgram(skybox_program_id);

        //--- Bind the necessary textures
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, skybox_tex);

        //--- Update the content of the uniforms (texture IDs, matrices, ...)
        // glGetUniformLocation         <--- fetch texture ID from shader
        // glUniform1i                  <--- set texture ID value
        // glGetUniformLocation         <--- fetch matrix ID from shader
        // glUniformMatrix4fv           <--- set matrix value

        //--- Load vertex array & render
        glBindVertexArray(skybox_vao);
        glDrawArrays(GL_TRIANGLES, 0, nCubeVertices);

        //--- Make sure the VAO is not changed from the outside
        glBindVertexArray(0);
    }
};
```

[8] you might want to add "#pragma once" at the beginning of "common.h".