Praktikum Echtzeit-Computergrafik
Assignment 6 – Textures

Technische Universität München
Institut für Informatik
Lehrstuhl für Computergrafik & Visualisierung

Christoph Neuhauser, Simon Niedermayr – SS 24

In the last assignment, you have implemented different lighting models in ShaderLabWeb (https://vmwestermann10.in.tum.de/). In this assignment, you will have a closer look at the topic of texturing.

Tasks

Task 1

In Moodle, you can find a state file you can import in ShaderLabWeb (https://vmwestermann10.in.tum.de/). When importing the state file, you should see a plane with a checkerboard texture mapped to it.

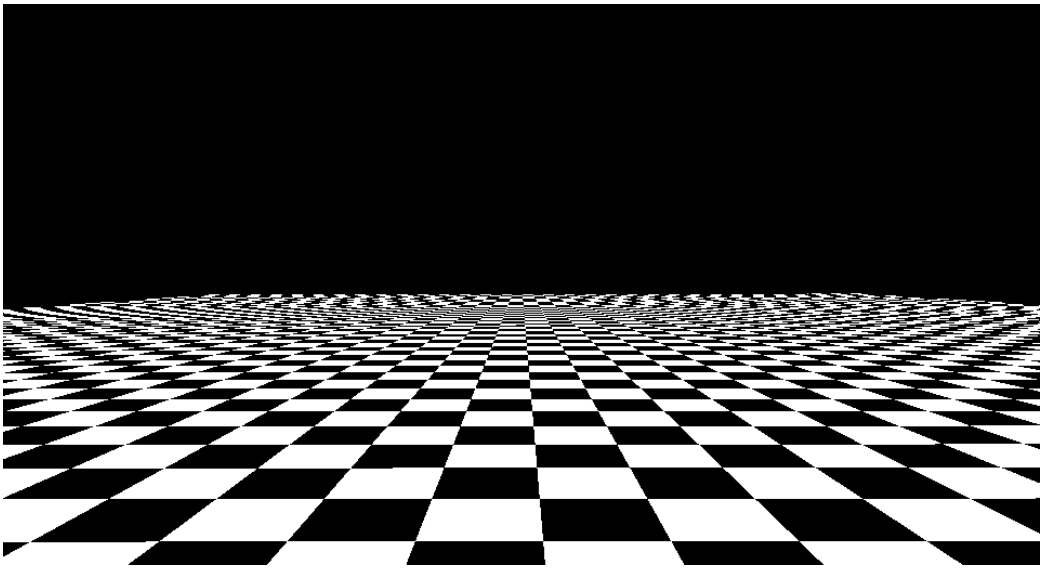

Figure 1: Scene in the provided state file.

First of all, explore in the base vertex and fragment shader how this texture is referenced in the shader.

- In the vertex shader, we pass the texture coordinates to the fragment shader.
- In the fragment shader, we define a `uniform sampler2D` object. In the `Uniforms` tab, the checkerboard image is bound. You can also bind other images from your local harddrive. In the shader, the function `texture`[1] is used to look up the value stored in the image at the specified texture coordinates. The texture coordinates lie in the range [0, 1],

[1] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/texture.xhtml

but we can also specify when using a graphics API like WebGL what to do when the coordinates lie outside of this range (i.e., clamp to border, repeat, mirror, ...).

Your task is to play around with the parameters of the texture mapping process. In the tab `Model`, you can specify different texture filtering modes and set the so-called anisotropy for anisotropic filtering. Move the camera around when changing the values and look at the terrain at a shallow angle.

- Nearest filtering returns the color of the nearest texel in the image without interpolation.
- Linear filtering returns the linearly interpolated color using the 2x2 surrounding texels at the texture coordinate sampling position.
- Anisotropic filtering uses more than one sampling position. It distributes the samples based on the pixel derivatives you got to know in the last assignment to sample more in one direction than the other one if necessary[2].
- Mipmapping internally creates multiple downscaled versions of the texture. Each downscaled versions has half the width and height of the last level. When sampling the texture in the shader, a mipmap level is selected based on the pixel derivatives. This reduces the aliasing, i.e., flickering, that is produced when sampling from a texture with too high resolution. However, mipmapping without anisotropic filtering may result in a more blurry output image.

[2] https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_texture_filter_anisotropic.txt

Task 2

In Assignment 3, you generated texture coordinates for your terrain in addition to the normal vectors. In this task, we are going to use these texture coordinates to no longer map single colors to the terrain based on the height, but colors we read from input grass, rock and snow textures provided together with this assigment on Moodle. You may use your code from Assignment 3 as a base for this and just use the simpler shading model used back in that assignment. You can try scaling the texture coordinates with a scalar factor and look at the results.

Task 3

On Moodle, a directory `Pebbles_028_SD` with different PBR (physically-based rendering) textures is provided. In the folder, you can find a base color texture, a normal map, a roughness map and an ambient occlusion texture. In this exercise, you should apply these textures to the Plane mesh (`Model` $\rightarrow$ `Mesh` $\rightarrow$ `Plane`). Our plane provides a (flat) normal already, however, we will ignore it and use the normals from the normal map instead. Hints:

- The normal map RGB colors are in the range $[0, 1]^3$. To convert them to $[-1, 1]^3$, use the formula $n = 2.0 \cdot \text{rgb} - 1.0$.
- The normals need to be transformed using $(\texttt{mMatrix}^{-1})^T$. For this, you can use the GLSL functions `inverse`[3] and `transpose`[4]. In Task 2 and Assignment 3 we just used the object space normals for the terrain and didn't transform them for the sake of simplicity.
- The camera position can be computed as $\texttt{vMatrix}^{-1} \cdot (0, 0, 0, 1)^T$.
- Use the ambient occlusion texture for adapting the strength of the ambient lighting.
- Use the roughness map for adapting the specular exponent of the Blinn-Phong lighting model.

[3] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/inverse.xhtml

[4] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/transpose.xhtml

Figure 2: Resulting rendered image of the code for Task 3.

For this task, you can build upon the Blinn-Phong model shading code from the last assignment.

If you are interested in how to apply a normal map on top of, e.g., a terrain mesh that already has normals defined, you may refer to this article: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

The link above gives you some advanced knowledge about so-called tangent space transforms, which is not necessary for this course, but can be interesting as a resource for further reading.

Task 4 (optional)

So far, the textures and models we have used in the assignments were always completely static. In this assignment, you will implement a water effect using displacement maps as found in many video games, e.g., from the Japanese company Nintendo. A video explaining this kind of effect can be found here: https://www.youtube.com/watch?v=8rCRsOLiO7k

Your tasks:

- Extend your terrain mesh generation code and include a plane (i.e., a quad consisting of two triangles) slightly above the ground level representing water. The geometry for the plane should be the first data in the mesh file.
- In the model vertex/fragment shader, add a new input/output declaration `flat out/in int fragmentVertexID;`. In the vertex shader, add `fragmentVertexID = gl_VertexID;`.
- In the fragment shader, compute the fragment color depending on whether the fragment was generated by the river plane or the terrain mesh. For the terrain you can reuse the code from task 2. Please note that a separate set of textures is provided for this task on Moodle.

```
if (fragmentVertexID > 5) {
  outputColor = getColorTerrain();
} else {
  outputColor = getColorWater();
}
```

  NOTE: In a real-world computer graphics application, it would be more beneficial to render terrain and water separately using two different shaders. Here, we are using the technique above to circumvent the restriction of ShaderLabWeb that it can only handle one mesh at a time.

- In your fragment shader, define a uniform time variable `uniform int timeMS;` and attach the elapsed time in milliseconds to it using `Uniforms` → `timeMS` → `attach to: Time in Milliseconds`.
- In the function `getColorWater`, convert the integer time to a floating point time, e.g., in seconds. Use it for shifting the texture coordinates you use for looking up the values in the water texture.

- Implement a wobbling effect for the water using a scrolling displacement map similar to what is described in the video above. It is beneficial to let the water and the displacement map scroll at different speeds and scale the size of the displacement map compared to the water texture.

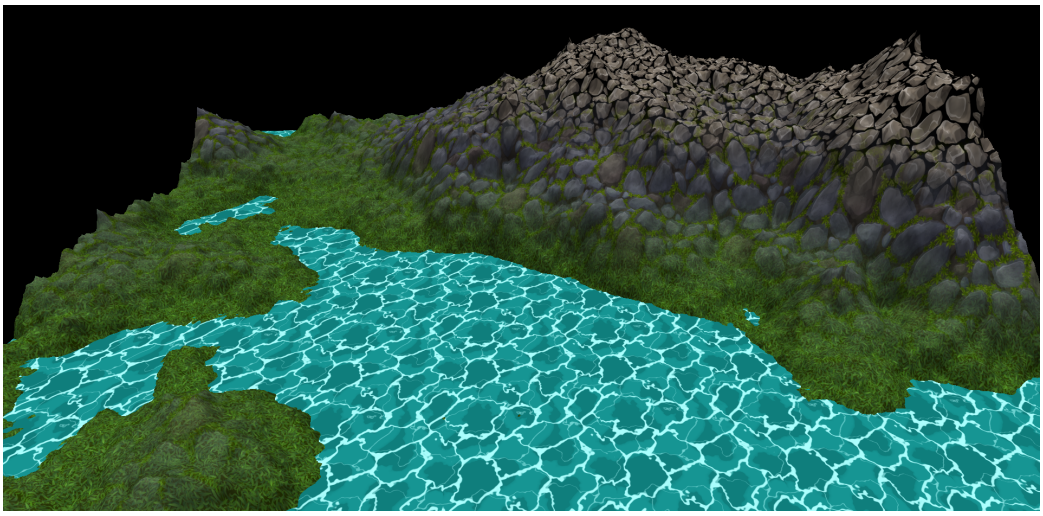

Figure 3: Terrain with displaced and moving water effect.