2 Advanced Rendering

This exercise focuses on advanced rendering techniques, including various texture mapping approaches and instancing. The theoretical part reviews concepts from surface analysis.

2.1 Theory

In this exercise, we will calculate the surface area of a torus. The surface of the torus is given by

$$f(u,v) = R \begin{pmatrix} \cos u \\ \sin u \\ 0 \end{pmatrix} + r \begin{pmatrix} \cos u \cos v \\ \sin u \cos v \\ \sin v \end{pmatrix}, \tag{1}$$

where r is the radius of the tube, R is the radius of the circle that the tube follows, and u and v are parameters in the range $[0, 2\pi]$.

(a) Calculate the Jacobian of the above parametrization of the torus. **(1 point)**

(b) Calculate the first fundamental form and simplify. **(2 points)**

(c) Calculate the area element and show that the surface area of the torus equals $A = 4\pi^2 rR$ by integration of the area element over the entire parameter domain. **(2 points)**

2.2 Practical Part

Please update the source code repository (`git pull`) to get the most recent changes. When you start Exercise 2, you should see an empty scene with a sky.

The goal of this exercise is to implement an infinite procedural terrain as shown in Figure 1. This terrain is made up of several square patches and the application determines the visible patches at runtime based on the current view. We start by implementing a single patch.

2.2.1 Geometry for the terrain (1.5 points)

First, we need geometry information for a single terrain patch in the form of a vertex array object, vertex buffer, and index buffer. This geometry should not contain any height information as those are added later in the shader. Instead, the geometry should only represent a tessellated flat square patch as shown in Figure 2.

The patch should comprise a single triangle strip. There are two strategies to realize this: The index buffer can either contain degenerate triangles that reference the same vertex more than once, or it can contain a restart index that allows you to start a new substrip. Refer to the documentation for `glPrimitive-RestartIndex` for more information.

(a) Generate positions and indices for a flat terrain patch in `CreateGeometry` in `Viewer.cpp` by filling the vectors `positions` and `indices`. The square patch should have `PATCH_SIZE` vertices

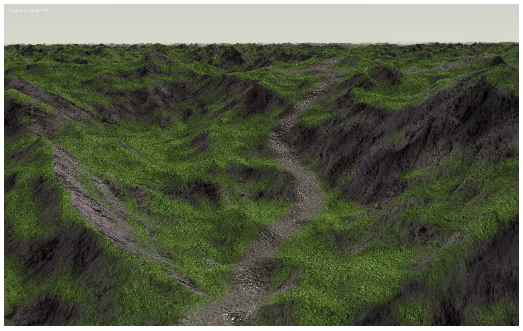


Figure 1: Final result of this exercise

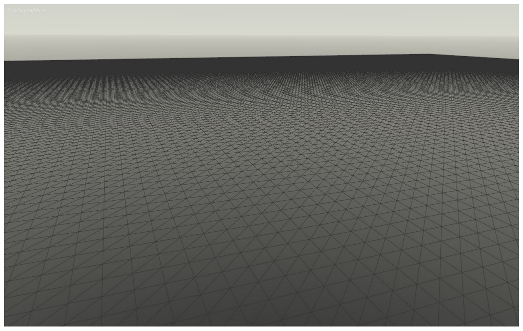


Figure 2: Wireframe of a single patch

along one side with a vertex spacing of 1 unit. The lower corner should start at the origin, the opposite corner should lie in the positive quadrant. The y-axis points upwards, i.e. the patch should extend in the xz-plane. **(1 point)**

(b) Add an appropriate draw call in `drawContents`. The shader and VAO are already bound and necessary uniform variables are already set. **(0.5 points)**

If you implemented everything correctly, you should see a flat gray plane in your scene.

2.2.2 Procedural height map (1 point)

We now want to use the vertex shader to change the height of the flat plane. For this task, the shader already contains a function `getTerrainHeight(vec2 p)` that returns the height of the terrain at a given `xz` position.

(a) Use the vertex shader `terrain.vert` to change the y-coordinate of the incoming vertex to the correct height. **(0.5 points)**

(b) Furthermore, calculate vertex normals in the vertex shader using finite differences. Pass them to the fragment shader and use them for lighting calculations. The illumination model is already implemented and invoked with the call to `calculateLighting`. You simply need to make sure that `n` contains the correct surface normal. **(0.5 points)**

If you implemented everything correctly, you should now see a shaded gray terrain in your scene.

2.2.3 Simple Texturing (2 points)

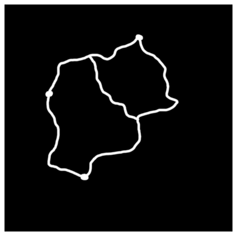
We now want to add a generic grass texture to our terrain. To achieve this, we first need to create a texture object and fill it with appropriate data. Then, we need to bind the texture to a shader variable and finally sample the texture in the fragment shader.

(a) Implement the generation of a filled texture in `CreateTexture` in `Viewer.cpp`. The function already contains code that loads pixel data, resulting in a byte array with RGB triplets. Create a texture object, set appropriate parameters (at least the wrap mode and the minification and magnification filters), upload the pixel data, and return the name of the texture. Also, make sure that mip maps are generated for the texture. If the parameter `repeat` is set to true, specify a repeating wrap mode; if it is set to false, specify a clamping wrap mode. **(1 point)**

(b) Add an appropriate `sampler2D` variable in the fragment shader and use it to sample the grass texture at the current location of the terrain. You can use the appropriately scaled `xz` coordinate of the fragment in world space as texture coordinates. The image should have a world-space size of $\frac{255}{10} \times \frac{255}{10}$ and repeat thereafter. Make sure that the texture is bound before the draw call (name is stored in `grassTexture`) and that the uniform variable is set correctly. **(1 point)**

If you implemented everything correctly, you should now see a shaded grass-covered terrain.

2.2.4 Advanced Texturing (3.5 points)



We now want to add some more interesting details to the terrain appearance. First, steep slopes will reveal the underlying rock. Second, we will add a road to our terrain. The road is based on a so-called alpha map that specifies the areas where the road should be (see right). In this alpha map, white areas depict roads, black areas depict grass/rock, and gray areas depict a mixture between the two.

(a) Add a second texture to the fragment shader that holds the rock texture (name stored in `rockTexture`). Determine steep slopes of the terrain based on the normal and show the rock texture instead of the grass texture. Realize a smooth transition between the two textures. The GLSL function `mix` can be of use for this task. **(1 point)**

(b) Add the alpha map and the road texture (names stored in `alphaMap` and `roadColorTexture`, respectively) to the fragment shader and show the road texture in the correct locations. You need differently scaled texture coordinates to sample the alpha map. The alpha map image should have a world-space size of 255×255 and is set up with a non-repeating wrap mode. **(1 point)**

(c) Use the specular map for the road (name stored in `roadSpecularMap`) to add localized specular highlights to the road. The specular map is a gray-scale texture that contains the specular intensity for each pixel. Feed this information to `calculateLighting`. **(0.5 points)**

(d) Add normal mapping to the road. The normal map (name stored in `roadNormalMap`) contains tangent-space normals as explained in the lecture (y-component needs to be inverted). Pass the tangent and bitangent (that you should already have from your normal calculations) to the fragment shader and use them to transform the tangent space normal from the texture to a world space normal. Use this normal for lighting calculations. **(1 point)**

If you have implemented everything correctly, you should now see a fully shaded and textured terrain patch in your scene. Specular highlights should be visible around the stones in the road texture.

2.2.5 Infinite terrain (2 points)

Finally, we want to render an infinite terrain by rendering multiple instances of the terrain patch with distinct offsets. We will use instanced rendering to render all required instances at once. To do this, we first need an instance attribute that holds the offset information for the terrain patches. The Viewer class already contains a vertex buffer named `offsetBuffer` for this purpose. Hints: You can bind the buffer with `offsetBuffer.bind()`. Furthermore, you can get the location of an attribute using `terrainShader.attrib("attributeName")`. You can upload data to a buffer by creating a `std::vector<Eigen::Vector2f>` and using `offsetBuffer.uploadData(vector)`.

(a) Add the offset attribute to the vertex shader and to the vertex array object in `CreateGeometry`. Declare the attribute as an instance attribute using `glVertexAttribDivisor`. In `drawContents`, fill the buffer with a single entry of $(0, 0)$. Replace the draw call with an instanced one and use the offset attribute appropriately in the shader. This should produce the same result as before. **(1 point)**

Finally, we must find all terrain patches that are visible. For this, we first calculate the view frustum and its bounding box. For all terrain patches that overlap the bounding box, we then perform view frustum culling, i.e., we check if the patch is actually within the view. The view frustum consists of six clipping planes. A patch is not within the view if it is completely behind any of these planes. You can use the functions `CalculateViewFrustum` and `IsBoxCompletelyBehindPlane` for this task. The height of the terrain is between 0 and 15.

(b) Collect all visible patches and load the corresponding offsets into the offset buffer to generate an infinite terrain. Store the number of visible patches in `visiblePatches` to show it in the top left corner of the screen. **(1 point)**

Submission

Zip all source code files in the folder `exercise2` into a zip archive and upload it to Opal.

2.2.6 Bonus Tasks (max. +5 points)

(a) Implement distance-based fog in the fragment shader to allow a smooth transition between the terrain and the sky in the background. For this, you need to bind the texture `backgroundTexture` to the shader variable `background`. After that, you can use the function `getBackgroundColor()` in the fragment shader to get the background color. Implement linearly increasing fog between distance 500 (no fog) and 1000 (complete fog). **(1 point)**

(b) Add view-dependent tessellation of the terrain patches using the tessellation shader, i.e., use a higher tessellation density for close patches. Optionally, make sure that there will never be cracks in the surface resulting from different resolutions of neighboring patches. **(2+1 points)**

(c) Implement water surfaces at a constant height in the terrain. You can choose how detailed you want your water to be (e.g. reflections and refractions). **(max. 5 points)**

(d) Implement an editor for the alpha map in the application that lets the user paint roads on the terrain. **(5 points)**