Lab Assignments
Computer Graphics DT3025, HT2016

Martin Magnusson and Daniel Ricão Canelhas
November 19, 2016

3 Mapping techniques

Learning goals

- Getting acquainted with mapping techniques for textures and reflections, in general.
- Managing texture mapping in OpenGL.
- Managing environment mapping in OpenGL.
- Optionally: managing bump mapping in OpenGL.
- Optionally: using tessellation shaders for displacement mapping.

3.1 Texture mapping

In this task you should build on your previous code and add a texture to draw numbers on your 20-sided die. (Formally speaking, the texture will modify the *albedo*; i e, the "colour" in your BRDF; of the object.)

In addition to these instructions, you may also want to refer to the OpenGL wiki.[1]

Start by generating an OpenGL texture, using he following lines of code:

```
GLuint texture_handle;
glGenTextures(1, &texture_handle);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture_handle);
```

What this does is to create a new texture object (with no actual content) that is referenced by `texture_handle`, and sets it to the currently active one.

Now load an image to the texture. For this purpose you can use the function `lodepng_decode32_file()` available from `lodepng.h` (given as part of the lab code base).

```
unsigned char* image_data;
unsigned image_w;
unsigned image_h;
unsigned image_file =
lodepng_decode32_file(&image_data, &image_w,  &image_h,
                      "../../common/data/numberline_hires.png");
std::cout << "Read " << image_h << " x " << image_w << " image\n";
```

Now that you have loaded an image on the CPU side, you need to send it to the graphics card. This is done with the `glTexImage2D` function.

Using the `glTexParameter*` functions, a number of parameters can be set to determine how the texture image is treated. E g, what happens if you try

[1] https://www.opengl.org/wiki/Texture

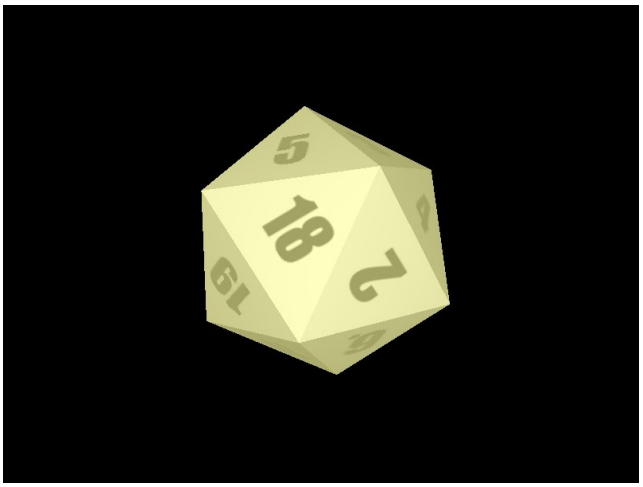


Figure 1: A textured 20-sided die.

to access a texel that is outside the dimensions of the image? (Look up the description of `GL_TEXTURE_WRAP_*` in the docs.)

A caveat at this point is that the default texture sampling parameters might result in you getting no visible texture at all. Two terms related to MIP/RIP mapping are *texture minification* and *magification*. How to sample a value from the texture is governed by `glTexParameteri(GL_TEXTURE_2D, ↩ GL_TEXTURE_MIN_FILTER, *)` and `GL_TEXTURE_MIN_FILTER`. The default value of `GL_TEXTURE_MIN_FILTER` is `GL_NEAREST_MIPMAP_LINEAR`, which requires that a MIP map has been generated. So you should either call `glGenerateMipMap` to do that, or select another sampling parameter that doesn't require a MIP map. You can read more in the `glTexParameter` docs.

The fragment shader will use a `sampler2D`[2] variable to access the texture from the *texture unit* on the graphics card. (A texture unit is a dedicated hardware unit that can quickly read from and interpolate values from the texture memory in an efficient manner, and cache results between different invocations.)

From the CPU code, you will need to let the shader know which unit to use. As before, when passing global variables from the CPU to the GPU, use a `uniform` variable. And since you now just need to send an integer (the number of the texture unit), you should use a `glUniform1i`. Assuming that you have a `uniform sampler2D tex_sampler` in your fragment shader, you can set this to texture unit 0 as follows.

```
glUniform1i(glGetUniformLocation(shader_program, "tex_sampler"), 0)
```

✏ What happens if you set the last argument in the line above to something else than zero? What is the maximum number you can use?

You are now almost ready to start performing texturing inside your fragment shader. But first you need to decide which part of the texture images should correspond to each fragment that you are drawing. This is the mapping function from uv texture coordinates to the 3D object.

Load the `uv_coords` array in the skeleton code into a VBO, along with the `points` and `faces` vectors, similar as how you did with positions and normals

[2] https://www.opengl.org/wiki/Sampler_(GLSL)

earlier. The uv coordinates specified here depend on having the same order of the `points` vector as in the skeleton code, so use this geometry instead of what you did in Lab 2.

Let the vertex shader pass along these coordinates unchanged to the fragment shader in a new `out` variable.

Finally, you should add the fragment shader code that will actually fetch the data from the texture units and do something interesting with it. You can do this by adding:

```
vec4 tex = texture(tex_sampler, uv_coords);
```

Now you can use the `tex` variable in your lighting computation. For example, try changing the colour using `tex.rgb`, or the specular exponent of a Blinn-Phong BRDF using `1+tex.r*100`.

You should now have textured 20-sided die with a number on each side, as in Figure 1.

3.2 Environment mapping

The goal of this task is to simulate a reflective (mirror-like) material by using an environment map. There are two sets of images that you can use for your environment map under `common/data/`. One set is useful for debugging (`cube-test_*.png`) and the other is more nice-looking (`cube-room_*.png`). You should use these images as a texture cube in OpenGL, from which a sample can be taken for the reflected view vector, which will then add to the colour of the surface, and make it appear as if it is more shiny.

1. Create a new texture with `glGenTextures` and activate it.
2. Bind it as a `GL_TEXTURE_CUBE_MAP` (instead of `GL_TEXTURE_2D`).
3. Load the six sides of the cube map with `lodepng_decode32_file`.
4. Now you'll have to call `glTexImage2D` to assign the six images to the texture unit. This is done a little differently for cube maps than for the 2D textures you have used before. The *target* (i e, the first argument to `glTexImage2D`) should now not be a `GL_TEXTURE_2D` but rather `GL_TEXTURE_CUBE_MAP_[POSITIVE/NEGATIVE]_[X/Y/Z]` depending on which side of the cube it is. (You can also exploit the fact that the targets are defined as an ordered set of ints, so `GL_TEXTURE_CUBE_MAP_POSITIVE_X + i` will also work.)
5. Tell the shader program which texture unit the environment map is in, by declaring a new `uniform samplerCube env_sampler` and passing the correct texture unit number with a call to `glUniform1i`, as you did before. (The `samplerCube` picks texels from a cube map, given a 3D vector, as described in the lecture. So you don't have to implement the mapping from the 3D vector to the cube surface yourself.)
6. Update your fragment shader to pick a colour from the environment map when computing the colour of a fragment.
   (a) In order to do so, you will need to compute the *reflected view direction*; i e, where in the environment does the light that gets specularly reflected into the camera come from? (Recall how you computed the reflected light vector in Task 2.1.)

(b) Pick a colour from the environment map. This is again done using a call to the GLSL `texture` function, but using your reflected view vector: `texture(env_sampler, reflection_vector)`.

(c) Mix it with the colour you computed in your BRDF. You can set a parameter $r \in [0,1]$ and set the final colour to

$$r \cdot (\text{colour from environment map}) + (1-r) \cdot (\text{colour from Lab 2}).$$

3.3 Normal mapping

This task is required only for grade 4.

In addition to modifying the material properties with an image, you can also use images to cheaply simulate more complex geometry, by tweaking the normal vector of the otherwise flat surfaces of your object.

Provided with the lab skeleton code (under `common/data`) is another image `numberline_nmap_hires.png`. Have a look at that image. It specifies normals as full 3D coordinates in the *tangent space* of the surface, scaled so that $[-1,1]$ maps to $[0,1]$. In other words, a normal vector that agrees with the actual normal of the surface would be $(0,0,1)$ in the tangent space, and maps to colour $(0.5,0.5,1)$ in the normal map. Your task now is to use that image as a normal map (bump map).

1. Load that image with `lodepng_decode32_file()` as you did with the texture map in Task 3.1.
2. Instantiate a new texture, using `glGenTextures`, `glActiveTexture`, and `glBindTexture` as before.
3. Assign the normal map to the active texture using `glTexImage2D`.
4. In your fragment shader, make a `uniform sampler2D normal_sampler` to read from the normal map.
5. On the CPU side, say which texture unit the normal map is in:

```
GLint normalSampler =
  glGetUniformLocation(shader_program, "normal_sampler");
glUniform1i(normalSampler, *);
```

✏ What value do you specify instead of the star here?

Now comes the tricky bit: specifying tangent and bitangent vectors (illustrated in Figure 2) as vertex attributes to be passed through the vertex shader to the fragment shader. First of all, declare two `std::vector<glm::vec4>` where you'll store the new vectors (just as you did with the normal vectors before.) Computing the tangent and bitangent vectors is best done in the same loop where you loop over the triangle faces of the object and compute the normals.

For each face, compute the difference in uv and xyz coordinates for two of its edges:

$$\mathbf{e}_1 = \mathbf{p}_2 - \mathbf{p}_1, \tag{1}$$
$$\mathbf{e}_2 = \mathbf{p}_3 - \mathbf{p}_1, \tag{2}$$
$$\mathbf{f}_1 = \mathbf{u}_2 - \mathbf{u}_1 = (\Delta u_1, \Delta v_1), \tag{3}$$
$$\mathbf{f}_2 = \mathbf{u}_3 - \mathbf{u}_1 = (\Delta u_2, \Delta v_2). \tag{4}$$

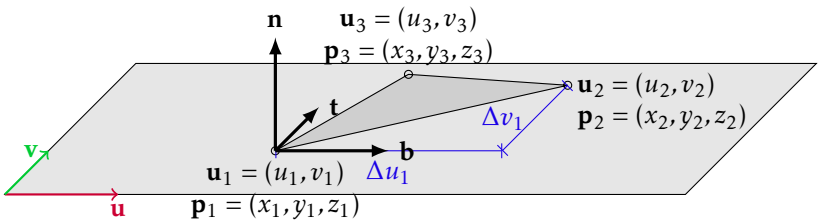


Figure 2: A triangle on a texture. Given the 2D uv coordinates at its vertices, we also need to compute the 3D vectors **t** and **b** for each vertex.

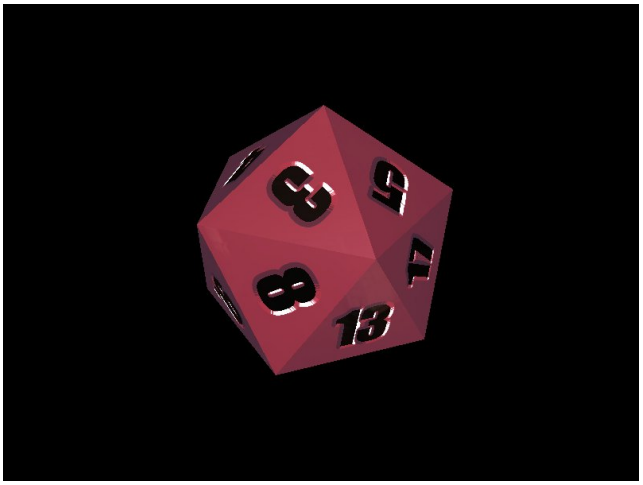


Figure 3: Now with a bump map! (And a different colour.)

Now, the tangent and bitangent can be computed as

$$r = (\Delta u_1 \Delta v_2 - \Delta v_1 \Delta u_2)^{-1}, \tag{5}$$
$$\mathbf{t} = (\Delta v_2 \mathbf{e}_1 - \Delta v_1 \mathbf{e}_2)/r, \tag{6}$$
$$\mathbf{b} = (\Delta u_1 \mathbf{e}_2 - \Delta u_2 \mathbf{e}_1)/r. \tag{7}$$

In the vertex shader, send the normal, tangent, and bitangent vectors (in view space, not perspective) to the fragment shader (just like you have done with the normal vectors before).

In the fragment shader, do the following.

1. Scale the vector that you get from your `normal_sampler` so that it is again on the interval $[-1, 1]$.

2. The normals in the supplied normal map make it appear as the numbers are outdented from the surface (bumps). If you would rather have them indented (cavities) you can negate the xy (not z) coordinates of the normal vector here.

3. The three vectors **t**, **b**, **n** form an orthogonal base that can be used as a rotation matrix.

```
mat4 TBN = mat4(tangent, bitangent, normal, vec4(0,0,0,1));
```

Now TBN can be used to rotate the normal vector you get from the normal map to the world space, so that you can plug it in the lighting equations you have from before.

- Alternatively, you can create TBN in the vertex shader and use the inverse of it to transform the light and view vectors into tangent space, rather than the other way around. This will lead to a little more work in the vertex shader, but less in the fragment shader (which is run more often) so will usually be more efficient — but it doesn't matter for this small scene!

The result should look something like Figure 3.

3.4 Displacement mapping

This task is required only for grade 5.

The final task of this lab is to improve on the bump mapping in Task 3.3 by doing displacement mapping instead, and actually changing the geometry of the object (resulting in something that looks like Figure 4). With modern graphics hardware, it is possible to do use the tessellation shaders in the graphics pipeline to do all the heavy lifting, and compute displacement mapping in real time.

It's good to refer to the OpenGL wiki when doing this task:

- Tessellation
- Tessellation Control Shader
- Tessellation Evaluation Shader

And these are some other good web links about displacement mapping and tessellation shaders:[3]

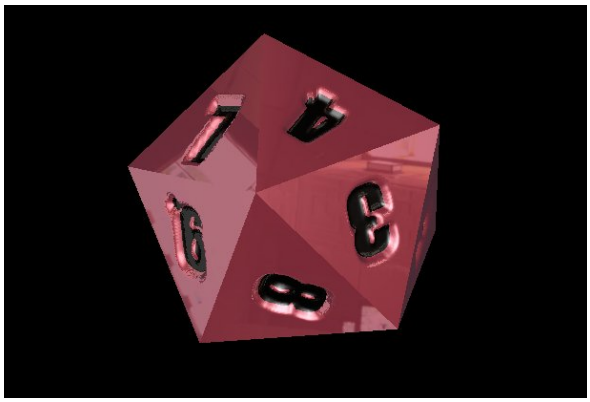
- The Little Grasshopper
- Codeflow
- Mali Developer Center

Tasks

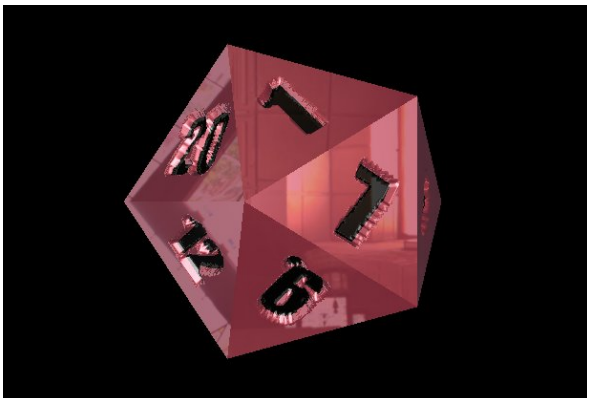
In addition to the vertex and fragment shaders you already have, you now need to add two tessellation shaders to your shader program: a *tessellation control shader* (TCS) and a *tessellation evaluation shader* (TES).

```
GLuint tcs = glCreateShader (GL_TESS_CONTROL_SHADER);
glShaderSource (tcs, 1, "lab3-4_tc.glsl", NULL);
glCompileShader (tcs);
checkShaderCompileError(tcs);

GLuint tes = glCreateShader (GL_TESS_EVALUATION_SHADER);
glShaderSource (tes, 1, "lab3-4_te.glsl", NULL);
glCompileShader (tes);
checkShaderCompileError(tes);

glAttachShader (shader_program, tcs);
glAttachShader (shader_program, tes);
```

[3] And this TechReport article is a nice read about what happens when you get overexcited about tessellation shaders...



(a) Using multiple weighted samples in the Tessellation Evaluation Shader to get a smoother surface.



(b) Using just one lookup in the displacement map gives a more jagged appearance.

Figure 4: A 20-sided die, with texture mapping, environment mapping, and displacement mapping.

You'll also need to load `common/data/numberline_df_inv.png` and assign it to a texture unit using `glTexImage2D`.

That's all on the CPU side!

Now, all the magic happens in the tessellation shaders. Both of these shaders are between the vertex and fragment shader stages of the shader program. Because they are run before rasterisation, they work on the 3D geometry of the object and not on the pixel level. These shaders operate on *patches*, which can be different kinds of surface primitives. But in this case, we will only work with triangular patches, so whenever you see "patch", think "triangle".

The variable names in the skeleton files follow the same convention as in the Grasshopper tutorial linked to above, so that it is easy to trace a variable as it is passed down through the stages of the rendering pipeline: If `Foo` is the original attribute sent from the CPU (into the vertex shader), then `vFoo` is the output from the Vertex Shader to the Tessellation Control Shader, `tcFoo` is the output from Tessellation Control to the Tessellation Evaluation Shader, and `teFoo` is the output from Tessellation Evaluation to the Fragment Shader.

A new Vertex Shader The vertex shader supplied with the lab skeleton is very bare-bones, without any model/view/projection matrix multiplications. The reason is that the tessellation shaders should work on the original geometry, and you will apply the transformation matrices later, in the Tessellation Evaluation Shader.

Tessellation Control Shader The TCS determines *how much* tessellation to do. It is called once per vertex in a patch.

The main job of the TCS is to tell the TES how much tessellation to do, and is specified as *inner and outer tessellation levels*. Start with the following values:

```
float tess_level_inner = 64;
float tess_level_outer = 1;
```

✏ There's a good illustration of these parameters on The Little Grasshopper's article. Experiment with different inner and outer tessellation levels and report on the outcome.

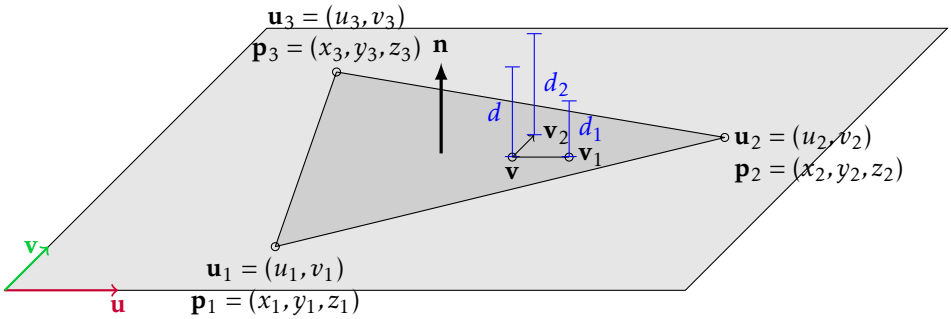


Figure 5: Computing the displacement and normals in the tessellation evaluation shader. Given a triangular patch, and a vertex $\mathbf{v}$ somewhere inside it, displace $\mathbf{v}$ some distance, and compute a normal for it. In this figure, $\mathbf{p}_1$ corresponds to `tcPosition[0]`, $\mathbf{u}_1$ corresponds to `tcTexCoord[0]`, etc.

Now, in the `main` of the TCS, the positions and texture coordinates that come from the vertex shader as just passed through to the TES.

Writing a Tessellation Evaluation Shader The TES is also invoked multiple times per triangle: once for each vertex of the tessellated output after the TCS. You can think of this as another vertex shader stage, for the newly created vertices. The main difference is that the TES also has access to the other vertices that belong to the same patch.

Have a look at the skeleton `lab3-4_te.glsl`. It specifies that we have triangular patches by doing

```
layout(triangles) in;
```

The TES gets the texture coordinates and positions of all three vertices of the original triangle from the TCS

```
in vec3 tcPosition[];
in vec2 tcTexcoord[];
```

It also gets the default input variable `gl_TessCoord` which are the coordinates of the current vertex *within the patch*. Note that these are *barycentric coordinates*, since we specified that our patches are `triangles`. (Recall the work-out with barycentric coordinates from the lecture.)

The TES takes the brand-new geometry created after the TCS and moves it around. This is where you will compute new positions and normals for the vertices, in order to actually change the appearance of the object. (If you wouldn't do any work in the TES, the result would be the same flat-sided icosahedron, just with unnecessarily many triangles in it...)

1. The first thing to do in the `main` of the TES is to find the interpolated position (x, y, z) and texture coordinate (u, v) for the vertex by interpolating with barycentric coordinates. This time, you can't let OpenGL do it for you, but have to do it yourself. The (α, β, γ) parameters in this case are the input variables `gl_TessCoord.x/y/z` that the shader gets. Multiply the triangle corners `tcPosition[]` with these coordinates to get the 3D coordinate of the vertex, and do the same with the corner texture coordinates

`tcTexCoord` to get the 2D texture coordinate at the vertex (which you will soon use to check how far it should be displaced from the original surface).

2. Compute the normal $\mathbf{n}$ for the patch.

3. Compute the amount of displacement for the vertex (d in Figure 5) by sampling from the displacement-map texture. Use a call to the GLSL `texture()` function, as in the previous tasks, using your interpolated texture coordinates. This will give you a sampled colour from the image. Since it is a greyscale image, and you just want a scalar, you can simply pick the red component of it (for example). You should scale the value to something reasonable: -0.1 is a good starting point for this texture image.

4. After adding this displacement (along the normal direction of the original patch), you need to compute a new normal for the vertex (since the shape of the surface has changed). If you omit this step, you will still get a die with properly indented (or outdented) numbers, but it will *look* flat, unless you view it from a shallow angle! (So the opposite effect of applying a normal map, where the look is different but the shape is the same.)

   In order to compute the normal, you need to sample two neighbouring points in the patch in order to see what their positions would be. You'll need to take the same *relative* step size in both texture and spatial coordinates. To do that, you should compute the edge vectors in both spatial and texture coordinates:

$$\mathbf{e}_1 = \mathbf{p}_2 - \mathbf{p}_1 \tag{8}$$
$$\mathbf{e}_2 = \mathbf{p}_3 - \mathbf{p}_1 \tag{9}$$
$$\mathbf{t}_1 = \mathbf{u}_2 - \mathbf{u}_1 \tag{10}$$
$$\mathbf{t}_2 = \mathbf{u}_3 - \mathbf{u}_1 \tag{11}$$
$$\tag{12}$$

   Then you can take a step that is, for example, 2% of the edge length along each edge direction:

$$\mathbf{v}_1 = \mathbf{v} + 0.02\|\mathbf{e}_1\|\,\mathbf{e}_1 \tag{13}$$

   and similarly for $\mathbf{v}_2$ and the texture coordinates.

   Sample the displacement map at those points (d_1 and d_2 in Figure 5) and use that to compute the displaced 3D positions of $\mathbf{v}_1$ and $\mathbf{v}_2$. Now you have three new points that you can use to compute a normal, and assign it to `teNormal` for the current vertex.

5. At the end of your TES, the MVP and MV matrices are finally applied the vertex positions.

The result after following the above steps will look a little jagged (as in Figure 4b). If you want, you can use the `gaussian_sample` function provided in the skeleton TES to get better samples (not just nearest-neighbour lookups in the displacement map).

✏ • Discuss examples of cases where displacement mapping would be better than normal mapping, and vice versa.

Assessment criteria

The criteria for passing this lab with grade 3 are the same as for passing Lab 1. To get grade 4, you also need to complete Task 3.3. To get grade 5, you also need to complete Task 3.4.

Deadline

The deadline is December 7. If you hand in your report after this, it will still be graded, but only when there is time.