Lab 3

Mapping techniques

Group 1:
Victor Svensson
920101-2714
Martin Hjalmarsson
860313-8218

Delivered to:
Daniel Canelhas
Date:
30 November 2016

3.1 Texture mapping

3.1.1 Implementation description

First of all, the UV_cords array was rearranged to match our points array. The given UV_cords array was arranged in a pattern of: [1,1,1,1,1,2,2,2,2,2,3,3,3,3,3…..] and our points array was arranged [1,2,3...50,1,2,3….50]. The rearranging was done manually in code. Another VBO was created for the UV_cords so they could be sent to the graphics card.

After this a texture2d was created following the lab-instructions and an image file was loaded in to memory with help of the lodepng library.

When calling the function glTexImage2D it is important to get the parameters for the memory right, otherwise it wont work. The group called the function with:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image_w, image_h, 0, GL_RGBA,
GL_UNSIGNED_BYTE, image_data);
```

And later generated a mip map for that texture. Final step was to send a texture_sampler to the graphics card as a uniform.

3.1.2 Implementation discussion

An option for generating a mip-map is to set the texture_parameters for sampling to another technique than mip-maping. (Default is mip-map thats why these calls are not needed if an mipmap is generated).

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

3.1.3 Parameter discussion

In the **glTexImage2D** it is important to specify the memory format of the data_array to RGBA. This because lodepng stores the data in the format of RGBARGBARGBA... in the float array after loading an image. The other GL_RGBA parameter is to set what the shaders should receive.

3.1.4 Pen questions

3.1.4.1 What happens if you set the last argument in the glUniform1i function call to something else than zero? What is the maximum number you can use?

The last argument specifies which texture unit to use. If the argument something else to zero, the texture-sampler must have been bound to the corresponding texture unit number, otherwise it won’t work.

The maximum value that can be used is stored in GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS and is platform dependent.

3.1.5 Result

3.2 Environment mapping

3.2.1 Implementation description

Every side of the cube was loaded in to the program with lodepng as before, so a total of six images. These images were then submitted in six function calls of glTexImage2D with the first argument specifying which side of the cube this image corresponded to in form of constants such as GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y etc.

A mip map was then generated for the GL_TEXTURE_CUBE_MAP that was bound and activated earlier in the same fashion as the GL_TEXTURE_2D was in the previous assignment 3.1

Finally, the cube_sampler was sent out to the shaders in a uniform variable as in previous assignment.

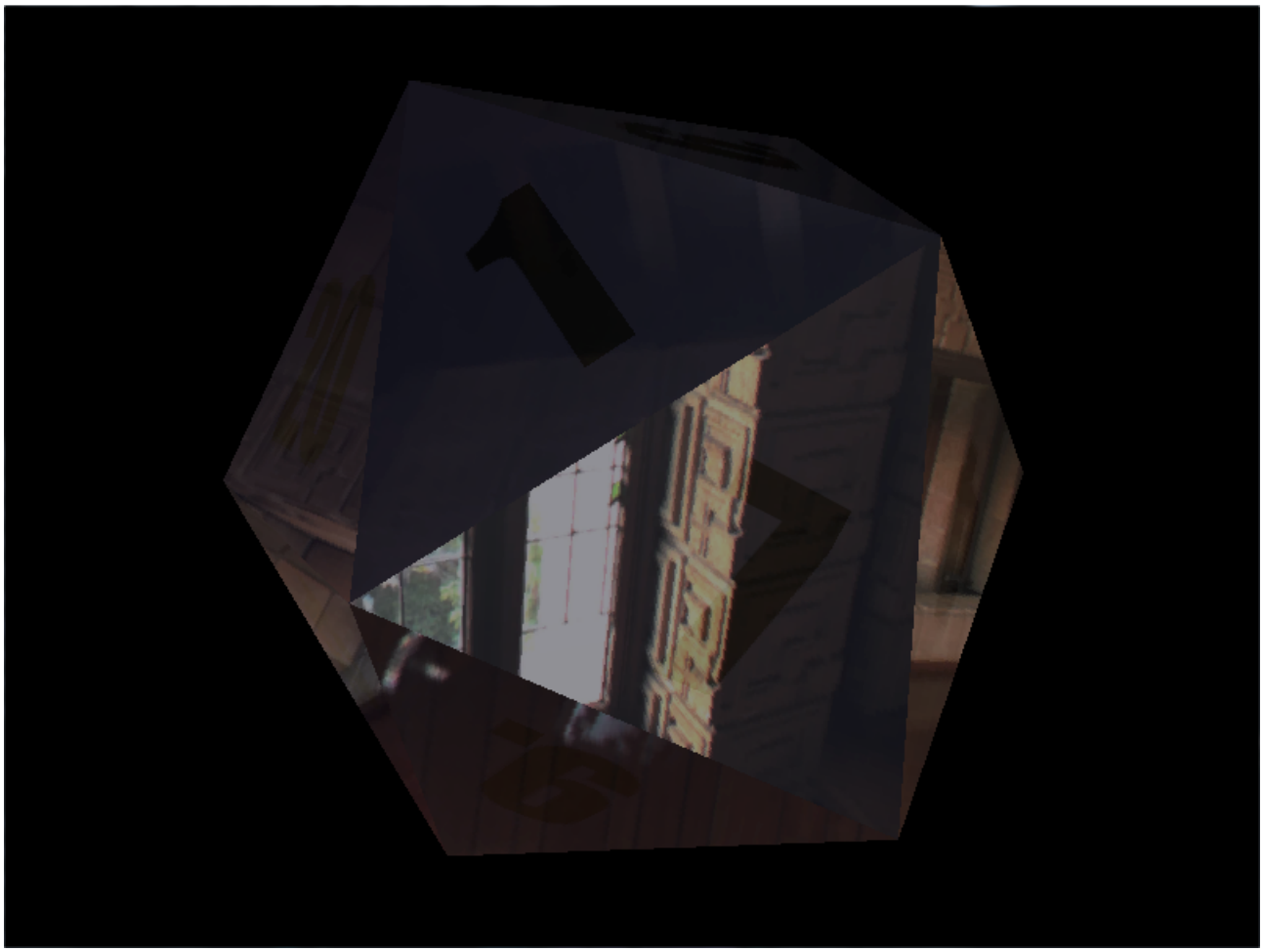
3.2.2 Implementation discussion

One tricky thing is to bind the correct image to the correct side of the cube, otherwise the environment will look really weird in the reflections. Thankfully, the error is easy to spot.

3.2.3 Parameter discussion

In the fragment shader in the texture() call for the cube_sampler, the 3D-reflectance vector is used to sample from the cube map instead of the 2D-uv vector used in the texture_sampler. The mapping from 3D to 2D on the image is done automatically.

3.2.4 Result



3.3 Normal mapping

3.3.1 Implementation description

Before the normal mapping can be implemented, the tangent and bitangent must be given. Thankfully these vectors can be calculated using the vertices and the uv coordinates, and is quite straightforward to implement in code.

This was done by modifying the function that calculated the normals from the points. In the same iteration as a normal for three verticies is calculated, the bitangent and tangent vectors can be calculated as well, so it was smooth to just inject new code in to that for-loop.

Since all math was given in the lab instructions it wont be explained here. The specific implementation in code can be seen in the .cpp files submitted.

Bitangent and tangent vectors are multiplied with the normalMatrix in the vertex shader to bring them in the same space as the normals in the fragment shader.

In the fragment shader, some complicated stuff happens described below in sequence:

1. Sample the value from the normal map using the uv coordinates

2. Bring the sample value from interval [0,1] to [-1,1] again. Now it is a normal vector in tangent space.

3. Multiply this normal vector with the TBN matrix that consists of the tangent, bitangent and original normal vector. By doing this you bring the new normal vector from tangent space to view space.

4. Proceed treating the new vector as the normal vector and all original lightning equations will apply.

3.3.2 Implementation discussion

One pitfall for the group on this assignment was that initially the bitangent and tangent vectors were just appended in to two vectors before sending them to the shaders. By doing this, the bitangent and tangent vectors followed the order of the index array, not how the points are ordered in the vertex array. This made the result look completely random.

To fix this, the group used the same technique as the normals to put them in to correct positions in a float array instead.

3.3.3 Parameter discussion

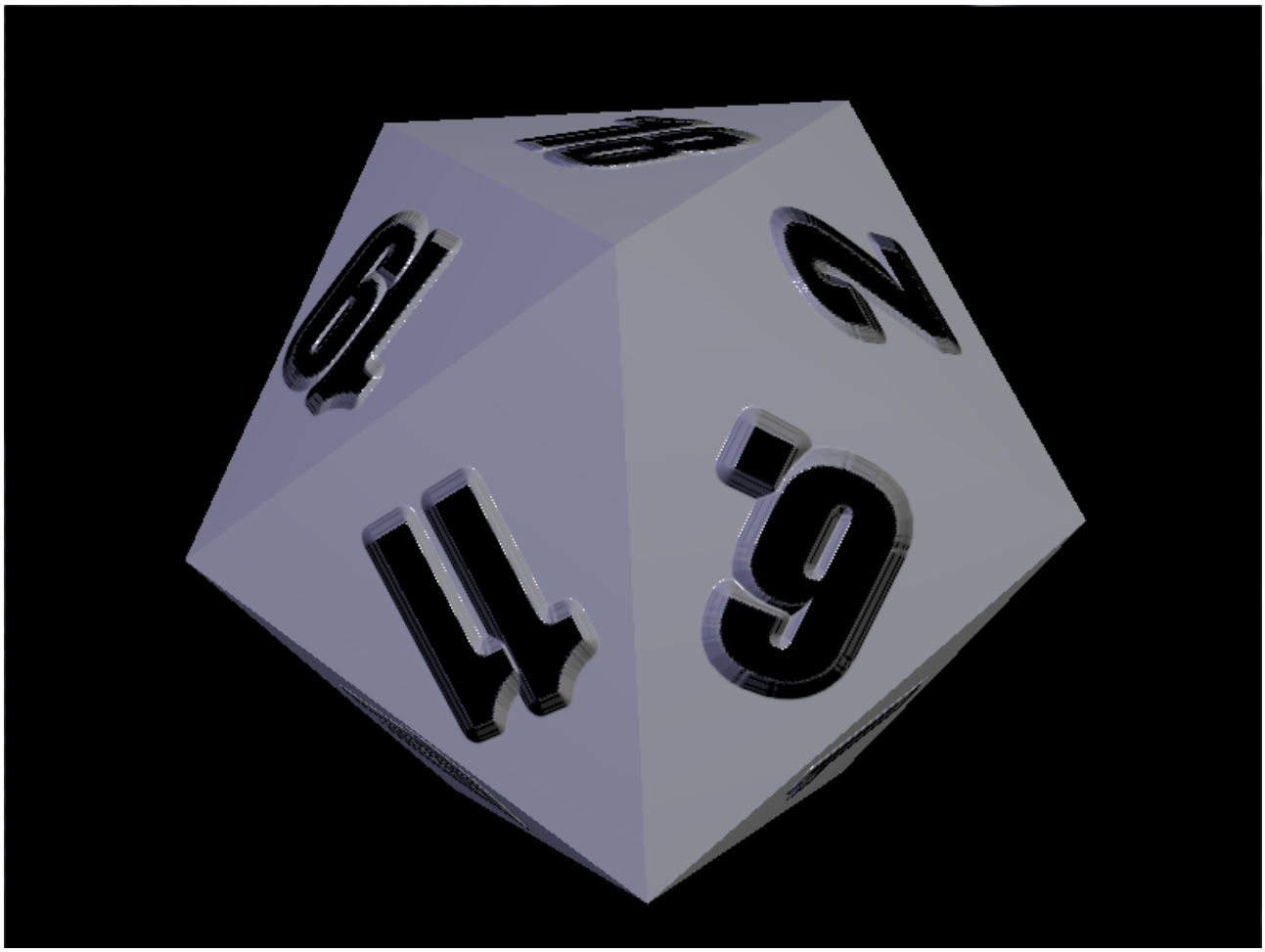
Nothing to discuss here for this assignment.

3.3.4 Pen questions

3.3.4.1 What value do you specify in the 2nd argument in the glUniform1i call for the normalsampler?

The corresponding value of the texture unit this sampler was bound to.

3.3.5 Result



The numbers are supposed to look like they are extruding out of the surface.

3.4 Displacement mapping

3.4.1 Implementation description

For the .cpp file in this assignment, there is only already familiar stuff to implement except for one major change. In the **`glDrawElememements()`** call the first argument is GL_PATCHES instead of GL_TRIANGLES.

All of the work was done in the tessellation evaluation shader (except an interesting typo explained in the discussion below).

The work done in the TES shader is explained below in sequence, corresponding number can be found in code comments:

1. The interpolated position is found using the gl_TessCoord variable and the original verticies for this triangle. Since gl_TessCoord is given in barycentric coordinates when working with triangular patches the implementation is straight forward. The same procedure is done for the UV coordinates.

2. The normal is calculated for the triangle connected by the three tcPositions.

3. The amount of displacement for the interpolated vertex is calculated by sampling the heightmap at the interpolated texture coordinate. This displacement is then added in the calculated normal direction.

4a. To calculate a new normal that corresponds to the displaced geometry two neighboring coordinates are calculated. This is done by traveling in to two directions a certain amount and check the displacement for those points.The same is done for the texture coordinates so the new coordinates can sample at the correct texture coordinate. After this the two near points are displaced in the normal direction as well.

4b. The three displaced positions form a plane that a normal vector is calculated from. This normal vector is now considered the normal vector for this interpolated position.

5. Apply the matrix transformations that was previously applied in the vertex shader.

3.4.2 Implementation discussion

The group had huge problems of getting this program to work initially. The program crashed (triangle could not move using our keyboard input and colors and shape of triangle was way off) even though the group did not get any compile errors from the shaders at all. This was before the group even started implementing the thing that the assignment required.

The group got the idea of examining linking errors when linking the shaders together, so using the OpenGL reference the correct code was written down that could potentially give us an error message if there was some problem with the linking of the shaders.

It turned out the crash was because of an linking error. The error was from legacy code given in the Lab3.zip file, the skeleton for the vertex shader and the tessellation control shader. There was identifier inconsistency between the vertex shader and the tessellation control shader (vTexcoord identifier was set in the vertex shader as an out variable and the tessellation control shader read identifier vTexCoord (Note capital C) as an in variable). We recommend that the other groups should implement linking error reports as well.

Instead of using the gaussian_sample() declared in the TES shader, we blurred the image in photoshop using gaussian blur instead and sampled it using the regular texture() calls.

3.4.3 Parameter discussion

When calculating the new normal for the displaced point you choose two neighboring points. Depending how you choose these points you could get a different normal. So the new normal is probably not the correct normal but hopefully good enough.

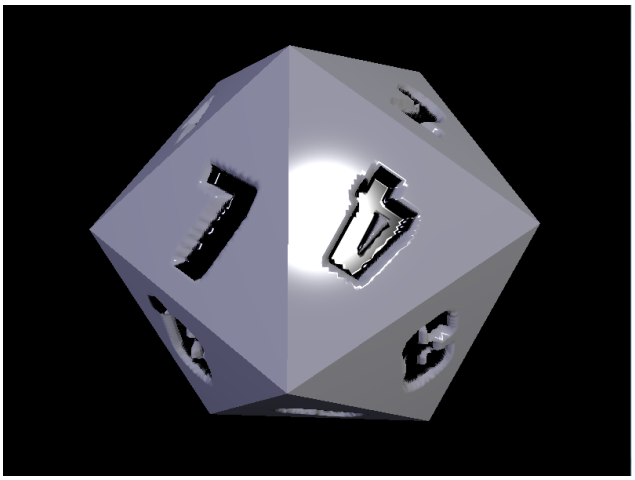
Parameters in this case would be the two direction vectors, and amount traveled in each direction.

3.4.4 Pen questions

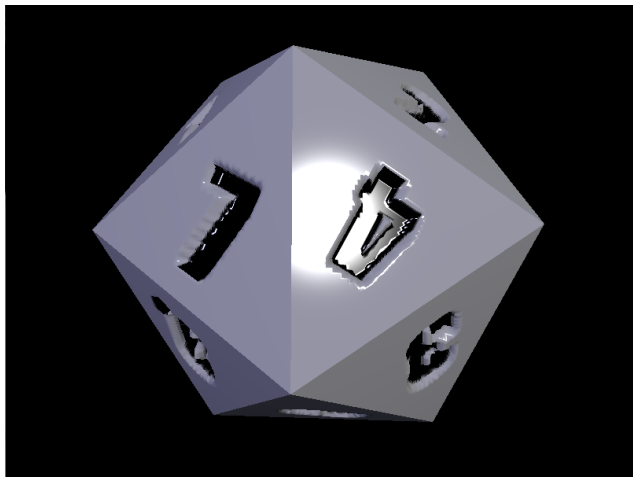
3.4.4.1 Experiment with different inner and outer tessellation levels and report on the outcome.

Inner tessellation level = 10
Outer tessellation level = 10

Inner tessellation level = 10
Outer tessellation level = 64



Inner tessellation level = 64
Outer Tessellation level =10



Inner tessellation level = 64
Outer tessellation level =64

Conclusion from this is that in our case, the inner tessellation level is the most significant. In this case, the numbers are supposed to look like cavities.

General comments on tessellation:

A high inner tessellation level adds detail inside the triangles keeping the edges as they were with no new vertices.

A high outer tessellation value adds detail to the edges of the triangles.

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

For performance intensive games normal mapping would probably be preferred in most cases since its usually enough to make the object surfaces look non-flat.

Generally, normal mapping works well for small variations in depth like dents on surface etc.

If you need to display bigger depth difference a displacement map should be used. For example a brick wall with very discrete bricks. Looked from the side using a normal map would expose the fraud.