Computer Graphics - Lab 3
Mapping techniques

Dec 15, 2016

Introduction

In this lab we will get familiar with different mapping techniques for textures and reflections in OpenGL. The tasks involve creating a texture map, environment map, normal (bump) map and displacement map (using tessellation shaders).

Source code from previous lab assignments will be used. This code includes the OpenGL Extension Wrangler (GLEW) and GLFW library. Transformation matrix calculations are done with the OpenGL Mathematics library (GLM).

3.1 Texture mapping

In this task a texture will be created for an icosahedron model. The first thing to do was to create an empty texture object by using the function *glGenTextures*. It was then assigned to the first position of *GL_TEXTURE* by the function *glActiveTexture(GL_TEXTURE0)*. It was then bound to set it as the object to modify, using the function *glBindTexture(GL_TEXTURE_2D, texture_obj)*. The first argument *GL_TEXTURE_2D*, binds the object to be a 2D texture.

The file which stores the texture image, was loaded in the program with the *lodepng_decode32_file* function, to the CPU. To send it to the GPU, a different function was used, *glTexImage2D*. This function takes information about the image file and what target type it is, in this case *GL_TEXTURE_2D.* The loaded image looks like this:

The function *glTexParameteri* enables the possibility to specify how the texture image is going to be processed and treated. It also specifies its wrapping and filtering methods. If a texel is outside the dimensions of the image, depending on what parameters was enabled, the texture may either repeat if wrapping is enabled, or it may be clamped so that the outermost texel is sampled (clamp to edge), or a border texel will be sampled (clamp-to-border).

The texture is then sent to a uniform variable of type **sampler2D** called *tex_sampler.* This uniform variable is received in the fragment shader, by sending the texture unit from the CPU, calling the function *glGetUniformLocation* to get the id of the uniform location, and then *glUniform1i* which takes two parameters, uniform id and an integer value. The uniform id is the id of the variable and an integer 0, which tells it to take the texture unit at a specific point in memory, and pass it to the uniform variable in the fragment shader.

The chosen texture has to be activated on the matching position with the *glActiveTexture* function, which takes the argument GL_TEXTUREn, where n = [0, 31]. The texture is then processed in the fragment shader and applied as color, with the uv-coordinates.

If the index first parameter of the function glUniform1i is not correct, the object will not be rendered with a texture. This is because the second argument tells the function, which texture unit to sample from, and in this case the only texture that is active, exists on position zero.

Result

The resulting figure with applied texture above is using the Blinn-Phong reflectance model and a purple color. The outcome looks like what was expected.

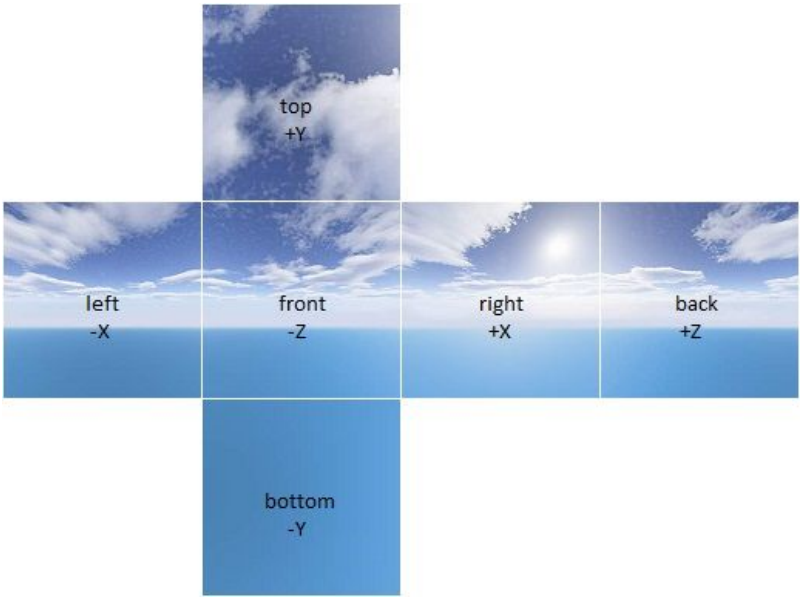
3.2 Environment mapping

In this task several texture files were going to be loaded in and combined into a cubemap, which contains six different 2D textures, which was then going to be applied on the same icosahedron object. To load two textures, one from previous and one from now, another texture object had to be generated using the same steps as last task. After, the texture object was set as active on position 1 on the texture “array”, *glActiveTexture(GL_TEXTURE1)* and then bound as a *GL_TEXTURE_CUBE_MAP* since the finished texture will resemble an opened cube texture, which is then clamped together.

After loading all the texture files, they were processed into a single texture using the same function as in the previous task, *glTexImage2D*, the target type this time consists of six different types since as previously mentioned a cubemap has six 2D textures. The target type is set depending on which side we would like to set the specific texture image to. The wrap parameter for *glTexParameteri* is then set to make the texture coordinates, (s, t, r) to be clamped to the edge by using the *GL_CLAMP_TO_EDGE* parameter. This parameter causes the coordinates to be clamped to [0 + 0.5*texel, 1 - 0.5*texel]. Since the texel center are considered to be at 0.5’s in OpenGL, clamping to [0,1] causes bilinear filtering to be applied between the edge pixels and the border colors.

The texture is then sent into the fragment shader and output as previous, except this time, the texture is applied on the view reflection of the icosahedron.

Here is a representation of the cubemap texture model:

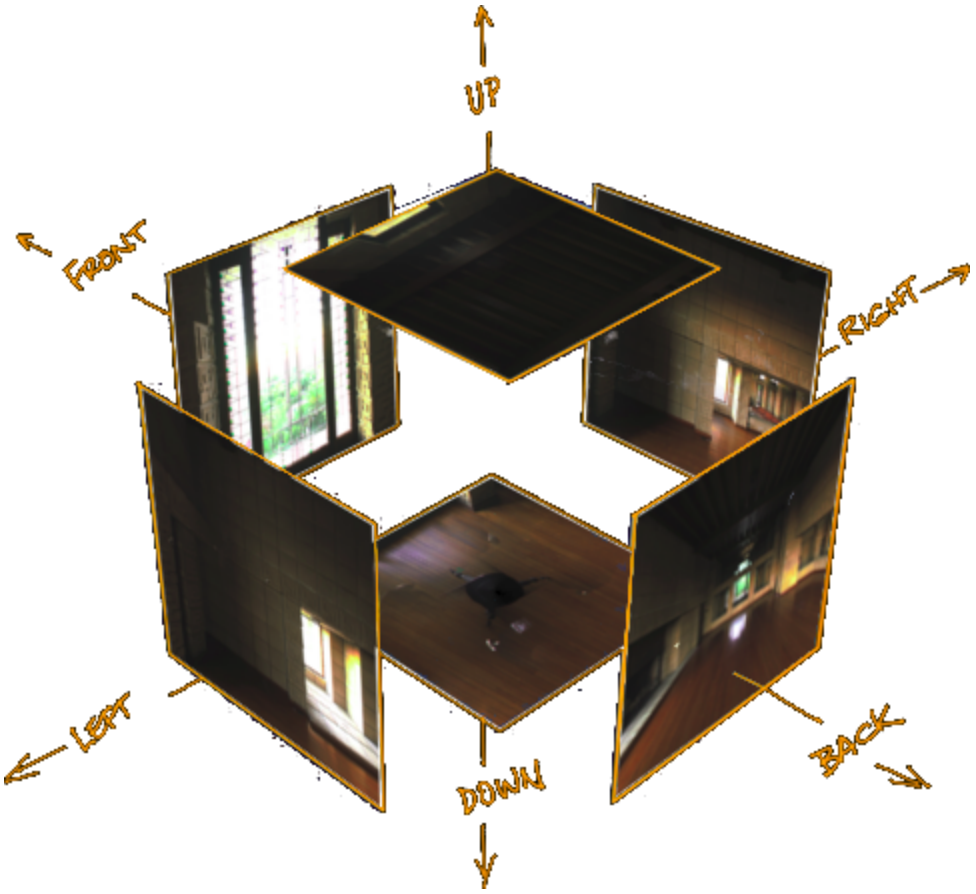


Here is a representation of the generated cubemap with the loaded image:

The target types are as follows:

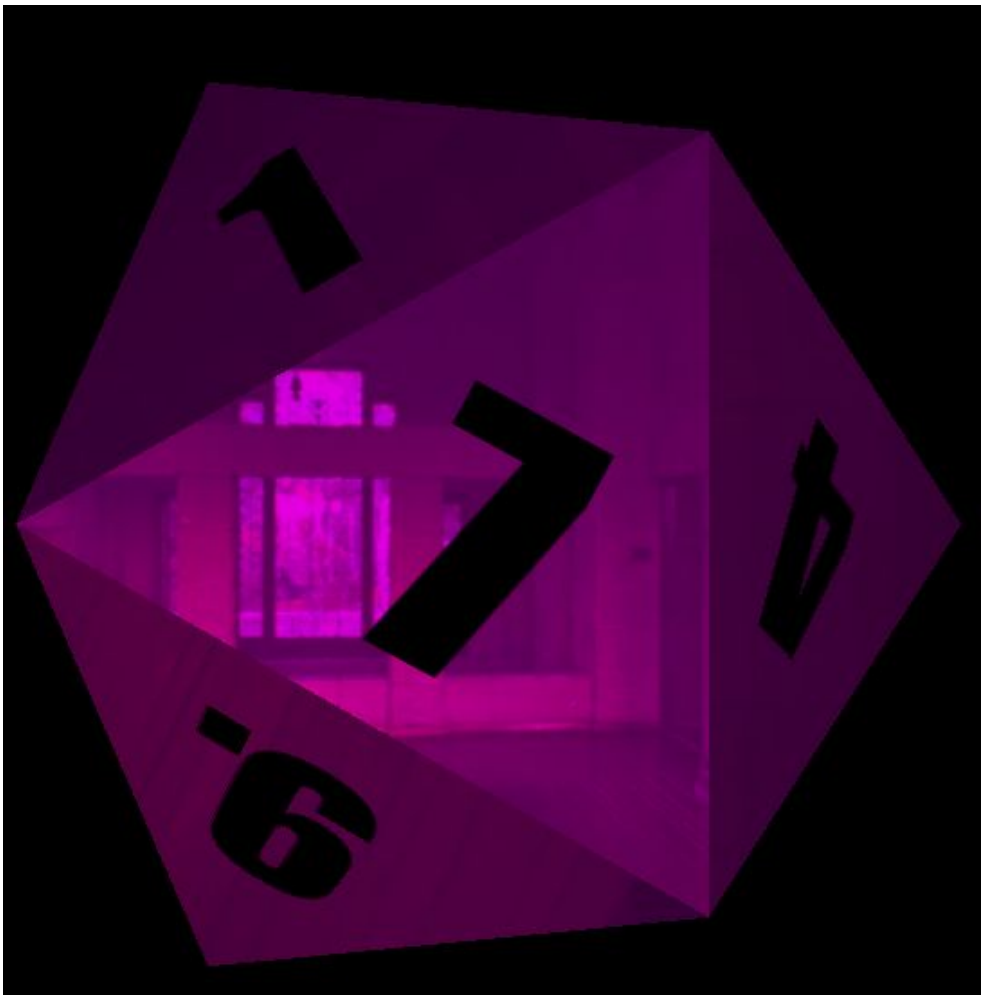
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Right
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Left
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	Top
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	Bottom
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	Back
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	Front

Generated cubemap texture with the loaded texture images would look like this:



Result

The resulting figure below is using the environment texture with the Blinn-Phong reflectance model and a purple color is added. The cubemap is visible as a reflection on the icosahedron, since the window image was applied as being the FRONT texture, the window on the other side is visible as a reflection for the viewer.



3.3 Normal mapping

One way to simulate complex geometry on textures, is to modify the normal vector of the object, to make it look more complex than it is, with help from a bump map. The image below was used and scaled so that [-1, -1] maps to [0, 1]. The task is to use this image as the bump map. Note that normal mapping only affects shading and not the surface itself, which displacement mapping does.

The image was loaded the same way as in previous task and a 2D texture was created, with the parameter GL_TEXTURE_MIN_FILTER with the value GL_NEAREST_MIPMAP_LINEAR. This gives a linear filtering on the two nearest mipmaps and does a linear blend between the two mipmaps, avoiding lines between mipmap levels, which is equivalent to trilinear filtering. A uniform was created in the fragment shader, *normal_sampler*, which takes the texture unit that was created from the image. The texture unit was assigned to index 2, with *glUniform1i(texLoc, 2)*, when the texture was created. To set the uniform to the correct texture unit, the same index was used in this function, *glUniform1i(texLoc, 2).*

Since the normals specify coordinates in the tangent space of the surface, the tangents will need to be calculated, as well as the bitangents. For each vertex, the tangent and the bitangent was calculated using the uv-coordinates. This was done in the same loop where the normals for the triangles are calculated. These were then loaded into a buffer for the shaders to access. In the vertex shader, the tangents and the bitangents was multiplied with the normal matrix, before being sent to the fragment shader.

In the fragment shader, the **sampler2D** uniform *normal_sampler* has received the texture unit at index 2 from the CPU.

Here the texture is sampled with texture unit at the texture coordinates, *uv-coords* by using the GLSL texture function:

```
vec3 bump = texture(normal_sampler, fs_uv.xy).rgb
```

The sampled values are then normalized before setting the interval back to [-1, -1]:

```
bump = normalize(bump)     // normalize before interval

bump = bump * 2.0 - 1.0    // sets to the interval [-1, 1]
```

Now the normals in the bump (normal) map make it seem as the numbers on the icosahedron are outdented, for them to be indented instead, the x and y of the bump (normal) vector needs to be set to a negative value:

```
bump = vec3(-bump.x, -bump.y, bump.z)
```

Next step would be to calculate the three vectors: tangent, bitangent and normal to make an orthogonal base to be used as a rotational matrix, so that it may be used in the lighting equation. The vectors are passed from the vertex shader.

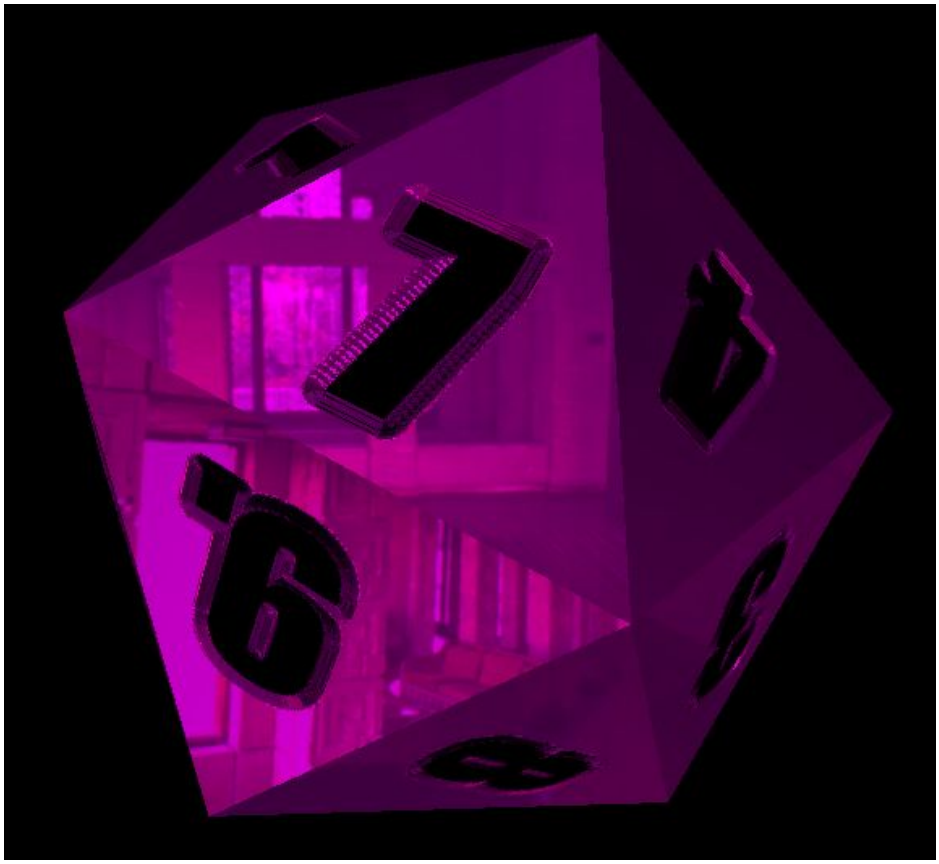
```
vec3 t = normalize(vs_tangent.xyz)

vec3 b = normalize(vs_bitang.xyz)

vec3 n = normalize(vs_normal.xyz)

mat3 tbn = mat3(t, b, n)
```

The bump map is multiplied with the TBN matrix to properly rotate the normals and the norm variable is now used as the default normal in the BRDF calculations.

```
vec3 norm = normalize(tbn * bump)
```

Result

The produced result is seen in the image below combined with the Texture mapping and the Environment mapping from previous tasks, with Blinn-Phong:

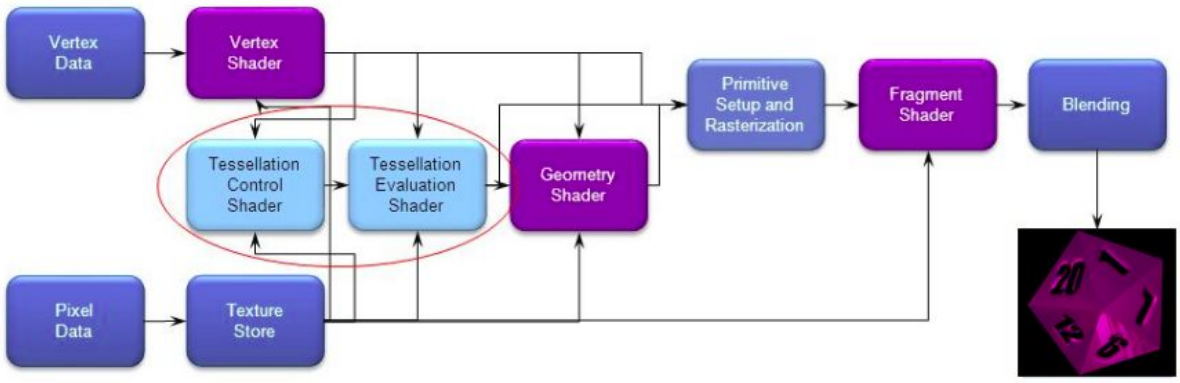


3.4 Displacement mapping

In the previous task (Normal mapping) the normals were manipulated to make the geometry of the object (the icosahedron) look more complex than it was. The object in itself was not physically altered in any way, however in this section, a physical manipulation (modification of the geometry) was computed in real time through the use of the tessellation shaders in combination with a displacement map.

The displacement is done by sampling a displacement value from the texture and from that displace the vertex along its normal. The tessellation shaders, Tessellation Control (TCS) and Tessellation Evaluation (TES), are between the vertex shader and the fragment shader. The pipeline, in this case, looks like this: Vertex -> Tessellation Control -> (Tessellation Engine) -> Tessellation Evaluation -> Fragment. A clarification may be had from the image below.

Note that the Geometry shader is not used in this task and between the TCS and the TES there's a Tessellation Engine which does the actual tessellation and calculates the barycentric points to the built-in variable glTessCoord.



The first step in this task was to add the new shaders to the new shader program. The parameters that was used for creating the shaders were *GL_TESS_CONTROL_SHADER* and *GL_TESS_EVALUATION_SHADER* to enable the programming of the specific shaders. Next step was to load the image below into a 2D texture:

Creating the texture, the index 3 was used for the texture unit and the uniform was passed to the sampler2D uniform, *heightmap*, which was located in the tessellation evaluation shader. Also, to draw the texture, in the *glDrawElements* function in the rendering loop, the first parameter was replaced with *GL_PATCHES*, to be able to draw the triangles from the tessellation shaders.

The vertex shader was only used to pass the values of the positions and the texture coordinates to the Tessellation Control shader.

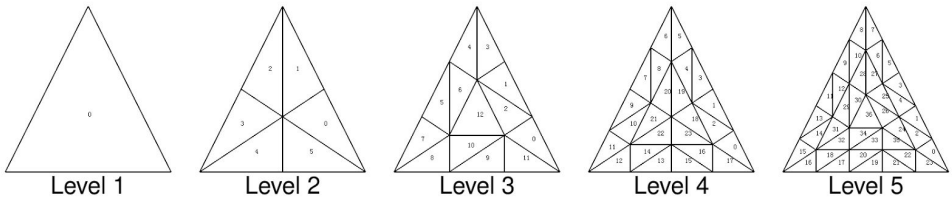
Tessellation Control Shader

The positions and the texture coordinates from the vertex shader is received in this shader and simply passed on to the Tessellation Evaluation shader, without any interpolation, which is the work that TES will do. The TCS controls how much tessellation a patch (triangle in this case) gets and defines the size of it. Since the drawing command is drawing 20 patches and each patch has 3 vertices, it will be a total of 60 TCS invocations. The output patch size will thereby be the integral constant of 3.

Tessellation levels

The inner tessellation level sets the quantity of new inner vertices (in a patch) to be added for eventual surface displacement. The higher amount of inner vertices, the more details in displacement can be shown.

Changes in the outer tessellation level has no visual effects in this case with the icosahedron since there are no displacements on the edges of the patches according to the displacement map. There will only be unnecessary many triangles to take into account.



Inner tessellation level modifications

				
Level 1	Level 2	Level 3	Level 4	Level 5

As can be seen above, a higher level of inner tessellation increases the number of inner triangles (and vertices). This makes the displacements from the displacement map to be applied more times and therefore makes the integers more defined.

Tessellation Evaluation Shader

The main work in the TES is to compute the displacement and the normals from the interpolated positions from the TCS and taking each vertex and giving it a position.

First the temporary position and the tex-coordinate from the TCS was computed and interpolated. The new position for the vertex was set by multiplying *gl_TessCoord* from the Tessellation Engine (which holds the barycentric points) with the vertex positions received from TCS. Interpolation of the tex-coordinates was calculated through multiplying *gl_TessCoord* with the texture coordinates received from the TCS.

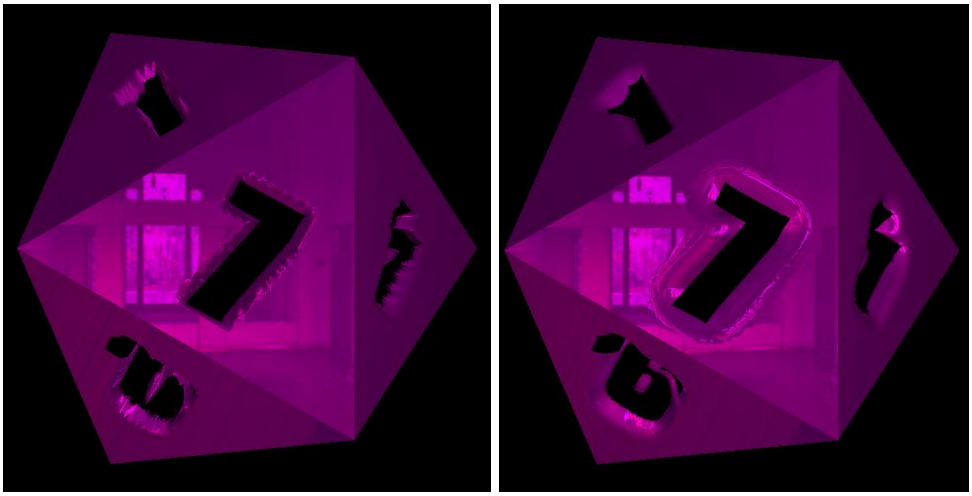
The normal was then computed for the patch before computing the displacement for the new vertex. The displacement was computed using the sampler2D *heightmap* uniform and the new tex-coordinate. Everything was then multiplied with a displacement coefficient which decides how much displacement there's going to be (indentation of the numbers on the icosahedron).

Now the vertex has a new position but the textures don't. Which means the new texture coordinates needs be calculated using the new vertex positions, and multiplied with a sampling offset to tell it how often the sampling should occur. The larger the number is, less often it is sampled and vice versa.

The new normal is calculated from the displaced points, new positions and the new texture coordinates and everything is then sent to the fragment shader along with the new positions, which are multiplied with the Model-View matrix from the previous lab. The final results are shown in the result section on the next page.

Result

The below images show the produced result with the texture and environment mapping and the displacement applied. And as usual the Blinn-Phong and a purple color is applied.



Parameters

In both resulting figures, the parameters are set to the following.

Inner tessellation level	64
Outer tessellation level	1
Displacement coefficient (Sets the level of the indenting/outdenting.)	-0.2 (Negative values makes the integers indented. Positive makes it outdented.)
Offset sampling (Sets the application rate of offset to the surface. The smaller number, the more often the offset is applied and therefore gives a more detailed representation.)	0.01

The displacement maps for the figures are originally the same, but the right one has some gaussian blur added to it. This makes the first one appear more edgy and jagged. The second with more blur looks softer.

Normal mapping vs. Displacement mapping

When would it better to use normal mapping instead of displacement mapping and vice versa? The answer is very dependant on the object one would wish to render and the distance to said object from the camera. If the displacement offset would be very low on a rendered object, it could be better to use normal mapping instead to increase performance.

Take this lid as an example:

Doing displacement mapping for the letter indentations on the lid, would be quite unnecessary since they're not very deep and quite small to be worth the cost of performance. Instead a normal mapping could be better suited. However, the deeper half-circle shaped indentation on the lid would probably be worth to consider using displacement mapping to make it more realistic.

Objects that are often close and intensively observed from many different angles could be worth using displacement mapping for. For example a human's face in a first person shooter computer game because of the many geometric details like ears, nose and chin.