

186.140 - Echtzeitgraphik - 2018W

Documentation Group 3 (Bath)



Ahmed Othman (01325531)

Jakob Knapp (01327386)

IMPLEMENTATION

- **APIs:** OpenGL 3.3. (using **GLEW** and **GLFW**)
- **Illumination model:** Phong lighting model was implemented
- **Textures:** We are using diffuse textures, as well as normal-maps and specular-maps to provide a three-dimensional impression.
- **Camera:** The animated camera we are using is implemented by linearly interpolating between keyframes. Each keyframe has a certain timestamp and position, at which the camera should be when the timestamp is reached. We chose this approach over animation with curves, such as b-splines or hermite-curves, because it gives the impression of an interior design showreel, which fits our demo the best. At the end of the animated camera-tour, the camera automatically switches to debug-mode, which means that it is freely movable. It is also possible to end the animated tour early by pressing the X-key. The controls for the debug-camera are as follows:

Controls:

- W: Move forward
 - S: Move backwards
 - D: Move to the right
 - A: Move to the left
 - Mouse: Looking around
-
- **Scene elements:** The scene has changed a lot compared to the first submission, while keeping all relevant elements. The aim was to present a bigger, modern bathroom when compared to the prototype-model. It features white tiles on the walls, a checkerboard-pattern on the floor and a doorframe painted in blue, where the effects of the different texture maps can be observed. Besides the broad mirror, which is also a central element of the scene, there are three different water outlets that can be turned on to activate the water flow, which is implemented using particles. Various objects, such as the toilet bowl, act as shadow casters to display the impact of omni-directional shadow-mapping. The bloom effect can be noticed on

any surface with high specular reflection.

The bathroom model was created in Maya from scratch and no external or prefabricated models were used.

ADDITIONAL LIBRARIES

- **OpenGL Mathematics (GLM)**

Glm.g-truc.net. (n.d.). OpenGL Mathematics. [online] Available at: <https://glm.g-truc.net/0.9.8/index.html> [Accessed 25 Nov. 2018].

- **Open Asset Import Library (Assimp)** for loading models into our scene

Assimp.org. (n.d.). The Open-Asset-Importer-Lib. [online] Available at: <http://assimp.org/> [Accessed 25 Nov. 2018].

- **STB_IMAGE** for loading Textures

Barrett, S. (2018). nothings/stb. [online] GitHub. Available at: https://github.com/nothings/stb/blob/master/stb_image.h [Accessed 25 Nov. 2018].

Source Code for Shader.h, Model.h and Mesh.h was taken from a web tutorial, as it will remain mostly unchanged.

Shader.h:

de Vries, J. (n.d.). LearnOpenGL - Particles. [online] Learnopengl.com. Available at https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/shader.h [Accessed 25 Nov. 2018].

Model.h:

de Vries, J. (n.d.). LearnOpenGL - Particles. [online] Learnopengl.com. Available at https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/model.h [Accessed 25 Nov. 2018].

Mesh.h:

de Vries, J. (n.d.). LearnOpenGL - Particles. [online] Learnopengl.com. Available at https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/mesh.h [Accessed 25 Nov. 2018].

IMPLEMENTED EFFECTS

Planar reflection using the stencil-buffer



Figure 1: The stencil-buffer is used to create a planar reflection of the bathroom-scene in the mirror on the left wall.

The big mirror on the left wall of the bathroom shows a reflection of the scene, including shadows. To achieve this impression, the scene is rendered twice and the stencil-buffer is used to keep the part of the reflected scene, which is covered by the mirror, from being drawn over by the wall behind the mirror. For mirroring the scene, we chose the approach of reflecting the scene, as opposed to reflecting the eye-point, across the xy -plane by scaling and translating the model-matrix accordingly. This reflection, however, changes the winding direction of the triangles of the model, which is why front-faces have to be culled instead of back-faces in the reflected scene. After drawing the reflected scene, which also includes rendering it into the omni-directional shadow-map, the mirror-polygon, which is a plane parallel to the xy -plane, is drawn into the stencil-buffer. It is important to not draw it into the color buffer, as this would inevitably overdraw the reflected scene. The values in the stencil buffer keep the pixels, which are covered by the mirror-polygon, from being overdrawn as the original scene is rendered in the final step.

Sources:

<https://www.opengl.org/archives/resources/code/samples/advanced/advanced97/notes/node90.html>
<https://open.gl/depthstencils>

Bloom

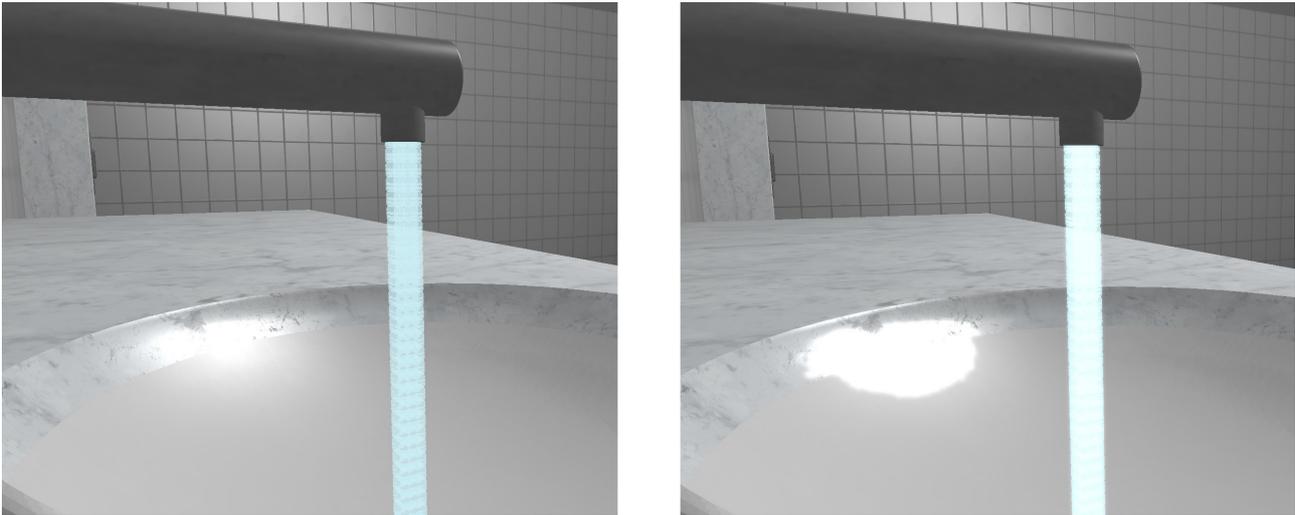


Figure 2: Left: Screenshot of the sink without bloom. Right: Activating the bloom effect adds a glowing impression to bright parts of the scene, such as the water-particles and the specular highlight at the edge of the sink.

The effect of bloom is used to give very bright areas of the rendered image a glowing impression. To achieve this, several steps have to be taken. As bloom is considered a screenspace-effect, the scene is not rendered to the display directly, but rather into a texture. This texture is then modified using image-processing techniques and finally rendered onto a plane in screen-space.

To achieve the desired result, we had to create new framebuffer-objects, the first of which has two color-buffers attached. This allows us to produce two output-images in just one render-pass by adjusting the layout locations of the used fragment shader. While the main image of the scene is rendered to the FBO's first color-buffer without any change, an additional image is drawn to the second color-buffer. This image contains only pixels, which are above a certain brightness-threshold, while all others stay black.

In the next step, this brightness-image is blurred using two FBOs, which alternately render into each others attached textures. By doing this, we follow the approach presented in the lecture of separating a Gaussian blue kernel into two smaller kernels, where one is blurring the brightness image in horizontal and one in vertical direction. The last step is rendering the final image to a screen-space plane. This is done by using a shader, which takes the original scene and the blurred brightness-image as texture input and combines them into one final output image.

In our Demo, the effect is noticeable on any bright spots, such as the specular highlights in the tub or the water coming out of the outlets.

Sources:

Lecture Slides: Screenspace Effects (2018 W)
<https://learnopengl.com/Advanced-OpenGL/Framebuffers>
<https://learnopengl.com/Advanced-Lighting/HDR>
<https://learnopengl.com/Advanced-Lighting/Bloom>

Normal Mapping

Normal mapping is implemented to give surfaces a huge boost in detail by using per-fragment normals. These per-fragment normals are stored in a normal map that is being passed to another texture sampler in the fragment shader. Since a normal obtained from a normal map is in the range $[0, 1]$, it has to be remapped to the range $[-1, 1]$ before proceeding with the lighting calculations.

Because these normal vectors from the normal maps are in tangent space, we need a matrix to transform them into world space. This matrix, also called the TBN matrix, is constructed by using the tangent, bitangent and normal vector. Instead of manually calculating the tangent and bitangent vectors, we are making Assimp calculate these vectors for us. After that, we pass only the normal and the tangent vector to the vertex shader. The bitangent vector can be calculated by taking the cross product of the normal and tangent vector. All three vectors are then transformed into world space by multiplying them with the model matrix. The three vectors are then combined into the TBN matrix and then passed to the fragment shader. The sampled normal in the fragment shader is then transformed from tangent space into world space using the TBN matrix.

Sources:

<https://learnopengl.com/Advanced-Lighting/Normal-Mapping>

Instancing

To simulate the water coming out of the tap, instancing was used to draw thousands of water particles. Each water particle is a small light-blue sphere with a little transparency. For all three taps in the scene we have in total 9550 particles, where each particle has its own model matrix and all model matrices for all particles are stored in a single array. Before entering the game loop, the positions of each of these particles is set in its desired position. In the game loop the model matrix of each particle is updated and with each iteration of the game loop every particle is translated in the $-y$ direction by a certain value. If a particle passes beyond a certain y -value, which is different for each simulation, the particle is teleported back up to defined position, which is also different for each simulation.

We use instanced arrays, which is a vertex attribute, to store the model matrices of the particles. In order to do that, we store the content in a vertex buffer object and then configure its attribute pointer. Because the amount of data allowed as a vertex attribute cannot be greater than vec4 and we have to pass in a mat4 , which is four times greater than vec4 , we use 4 vertex attributes. In our implementation we use the locations 2,3,4 and 5 for the model matrices. In the end we configure each of them as instanced arrays by calling `glVertexAttribDivisor`.

In the game loop, after updating all model matrices, we also update the vertex buffer object with the new model matrices and finally call `glDrawElementsInstanced` to draw the particles.

Sources:

<https://learnopengl.com/Advanced-OpenGL/Instancing>

Omnidirectional shadow mapping

In order to simulate shadows, we initially have to generate a depth map from the light's perspective. In our case, a cube map (which will be attached to a framebuffer) with six faces will contain the depth map for the shadow calculation. To avoid rendering the scene six times to create the depth cube map, we use a geometry shader to reduce the number of rendering passes to only one.

We will need a light space transformation matrix to transform the world around the light source to six different light spaces, one for each direction of the light and each face of the cube map. For the light space transformation matrix, we need a perspective projection matrix, which will stay constant for all six directions, and six distinct view matrices for each direction. In total, we will have six light space transformation matrices, one of which can be acquired by multiplying the projection matrix with one of the six view matrices.

The calculated transformation matrices are then passed to the geometry shader to transform the world-space vertices to six different light spaces. After that, the fragment shader takes the fragment position from the geometry shader as input and then calculates the distance between the fragment and the light source and maps the result to $[0, 1]$, which corresponds to the fragment's depth value. The result of this first render pass gives us a complete depth cube map that we can then pass to our main fragment shader, which is responsible for rendering our actual scene.

In the fragment shader we calculate a new depth value between the current fragment and the light source and then compare that value with the one sampled from the depth cube map. This way we can determine if our current fragment is in shadow or not.

Additionally, percentage-closer filtering (PCF) was implemented to smooth the edges of the shadows. To get rid of shadow acne, we include a shadow bias of 0.2. The depth map resolution is set to 1024 x 1024.

Sources:

<https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>

<https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows>

ADDITIONAL INFORMATION

Starting the application:

The Demo can be started from the command-line like this:

```
Bath-Demo.exe 1920 1080 1
```

where the first two parameters are the resolution of the window and the third parameter indicates, whether or not the application should be started in fullscreen mode. If the application is started without any parameters or just by double-clicking the .exe-file, it is started in windowed mode with a default resolution of 1920x1080.

Additional Controls:

In addition to the debug-camera controls, the following buttons are used:

- X** – end animated camera-tour and enter debug-mode
- B** – turn bloom on/off (only available in debug-mode)
- T** – activate water outlet at the sink
- U** – activate water outlet in the bathtub
- I** – activate shower

Lab Test:

The demo has been tested in the lab using the Nvidia Graphics Card on the AMAROK PC, where it successfully ran at 120 frames per second. It has been developed at home using a GTX 1060 Card.