

# Design Document

## **Story:**

A crazy scientist is working in a cellar room on creating a portal to the past. On one day, he managed to create one. He walked through it and was never seen again since that day.

## **Controls:**

- When the program is started, the camera moves automatically. To stop the automatic camera, just press Enter. You can now move the camera freely, as explained in the next point. To restart the automatic camera again, just press the Enter key again on your keyboard.
- Movable Camera: The W, A, S, D, Space and Shift Keys allow the user to move around. Forward (W), backward (S), to the left (A), to the right (D), up (Space) and down (Shift). The mouse can be used to change the view direction.
- Esc key: Makes the mouse visible. To regain control, simply click on the window.

## **Effects:**

- Specular Mapping
- Normal Mapping
- Point Lights with PCF Shadows (Point Shadows)
- SSAO
- Parallax Occlusion Mapping

## **Description of Implementation:**

- **SSAO** (Screen Space Effects: Slide 31 - 35): To implement SSAO, we had to change the rendering pipeline to deferred shading. First, the geometry is rendered without lighting. We therefore get framebuffers for the positions, normals, colors and specularity values. We also create an SSAO color buffer, in which the results

of the SSAO shader is written. For the calculation of the occlusion value, a normal-oriented hemisphere is used as a kernel. The random samples within the kernel are firstly calculated on the CPU (64 samples in our case) and are then sent to the SSAO fragment shader, where the samples are converted from tangent to view space and the occlusion value is calculated. The resulting values are written into the SSAO color buffer. In the lighting stage of our pipeline, we simply multiply the ambient value of each fragment with its occlusion value. For the implementation, we mostly oriented ourselves on the tutorial by Joey de Vries (<https://learnopengl.com/Advanced-Lighting/SSAO>) and Brian Will (<https://www.youtube.com/watch?v=7hxrPKoELpo>)

- **Parallax Occlusion Mapping** (Shading: Slide 39 - 42):

POM is a technique to create a sense of depth on flat textures, with small performance impact. In our scene, this effect can be observed on the walls and the floor. To use Parallax Mapping, a displacement map must be added to the model. The tangents, which are calculated from Assimp (modelexporter.cpp), are used in the vertex shader to transform the vectors from world space into tangent space with the so called TBN-Matrix. In the fragment shader (ssao\_geometry.frag) the view vector of the camera is traced, and the depth value of the current fragment depth from the displacement map is compared with the current depth layer. If the current depth layer is lower than the current fragment depth, the fragment depth will be shifted along the direction. This is repeated until the current fragment depth is above the surface of the texture. Next, the final texture value is calculated by interpolation between the layers below and above the found depth from the displacement map. Eventually, the new texture position value is used to generate the normals for a sense of depth. For the implementation, we also mostly used Joey de Vries' tutorials.

(<https://learnopengl.com/Advanced-Lighting/Parallax-Mapping>)

- **Point Shadows with PCF:** To generate the shadows, a cube texture is generated, which represents the depth map from the view direction of the lamp. With it, the depth at every fragment position of the objects can be sampled and a corresponding shadow value can be calculated accordingly. The generation of the depth map is done by rendering the scene from the view of the lamp for every of the six viewing directions. For every view direction, we “emit” every vertex of every triangle onto the cube map. This is easily possible with a geometry shader, as it can be seen in our implementation with “depth.geom”, as OpenGL even comes with the `EmitVertex()` function. In the lighting stage of our pipeline, we then sample the depth map and calculate the shadow value for each fragment. To soften the shadows, percentage close filtering was implemented. For that, a disk radius was defined, which represents the area we sample the cube map for every fragment. To reduce the number of samples we have to scrutinize, we defined an array of offset directions, called “gridSamplingDisk”. Those vectors all point in completely different directions, therefore, we reduce the number of sub-directions that are close together. (<https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows>)
- **Additional Features:** For faster loading times, we wrote our own model exporter, which converts all model types, which are supported by assimp, in our own *.bmf* format, which basically stores all model data in binary format. This has the advantage, that we do not have to convert our models by assimp with every start of the program. In our `model.h`, we then read the model data and create all the necessary texture buffers and vertex buffers. This idea is from Pilzschaf and his OpenGL tutorial. (<https://github.com/Pilzschaf/OpenGLTutorial>)

### **Development Status:**

We implemented all our effects as planned. Also, the automatically moving camera was implemented and our final scene was modeled successfully.

### **Used Tools:**

- Blender for creating our scene and the models.

### **Additional Libraries:**

- Assimp: For loading models in our engine (<http://www.assimp.org/>)
- GLEW: Querying and loading OpenGL extensions (<http://glew.sourceforge.net/>)
- SDL: Creating and managing OpenGL window and input management (<https://www.libsdl.org/>)
- GLM: Mathematics for OpenGL (<https://glm.g-truc.net/0.9.9/index.html>)

**Tested Graphics Cards:** AMD Radeon (TM) R5 M430, NVIDIA RTX 2070 Super and NVIDIA GTX970m.

**Tested Graphics Card in Lab:** AMD RX580