Image Processing and Computer Graphics
Prof. Dr. M. Teschner

Exercise 4 - Shadow maps

1 The shadow map algorithm

As part of exercise 2, we have constructed shadow maps, i. e. depth buffers, by employing the Frame Buffer Objects (FBO) as a render-to-texture technique. In this exercise we will build on the results and implement the full shadow algorithm based on shadow maps.

Recall from the lecture that the main passes and steps of the algorithm are:

1. Render the scene from the light position into the depth texture of the FBO (*light projection and transform*).
2. Render the scene from the view position into the general frame buffer (*view projection and transform*). For each fragment that passes the general depth test, transform its position using the *light projection and transform* from the first pass: $\mathbf{v}' = L_{proj} * L_{transf} * \mathbf{v}$, where L_{proj} and L_{transf} are the matrices for the light projection and the light transform, respectively.
3. Compare the transformed position's z-value with the one stored in the depth buffer. If it´s value is greater than the one stored in the texture, the fragment lies in shadow.

Hints:

- The FBO can be unbound using *glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0)* after the first pass is completed. If the second parameter is zero, OpenGL switches back to the standard frame buffer.
- OpenGL needs to be instructed to interpret the data stored in the texture as depth values for the depth comparison. This is accomplished by the functions: *glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_R_TO_TEXTURE)* and *glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL)*.
- The matrices L_{proj} and L_{transf} have to be stored for use in the second rendering pass. Use *glPushMatrix()*, *glLoadIdentity()*,*gluPerspective()*, *gluLookAt()* and *glGetDoublev()* to construct them.
- The matrices are needed to compute texture coordinates in order to access the shadow map. Therefore, construct a matrix $T = T_{bias} * L_{proj} * L_{transf}$ using *glLoadIdentity()*, *glLoadMatrixd()* and *glMultMatrixd()* and provide it for later use with *glActiveTexture(GL_TEXTURE7)*. The matrix T_{bias} maps from $[-1, 1]$ to $[0, 1]$, i. e. from the canonical view volume to the texture space. Note that all coordinates should be mapped to $[0, 1]$ since the z-value of a fragment lies in this range, too.

- Employ shaders to perform the actions necessary in the second rendering pass (see next section).

Optimization:

If the functions *CMesh::addTranslation()* or *CMesh::addRotation()* are used to transform the meshes, we will need to compute and provide an additional matrix that contains the inverse of the trackball transformation matrix (the reason is given in the next section). Again, employ the function *glGetDoublev()* to grab the model view matrix after the trackball transformation was applied (see the marked spot in function *CViewer::draw()* in file viewer.cpp). Compute the inverse of the matrix and then its transpose (recall the discussion about column-major and row-major matrices from the first exercise sheet) and provide it for later use with *glActiveTexture(GL_TEXTURE6)*.

Implementation:

1. Construct the FBO in *CShadowMap::CShadowMap()* and clean it up in *CShadowMap:: CShadowMap()* (see the file shadowmap.cpp).
2. Fill in the missing code sections of the shadow mapping algorithm in function *CShadowMap::drawSceneWithShadows()* (see the file shadowmap.cpp).

2 Vertex- and fragment shaders

We want to use shaders in the second rendering pass of the algorithm to perform the *light projection and transform*, the shadow map comparison and the application of a shadow color by using vertex- and fragment shaders. Recall that shaders allow to replace certain parts of the fixed functionality of the rendering pipeline with ones own implementation. More specifically, vertex- and fragment processing can be replaced.

Preliminaries:

1. Familiarize yourself with the syntax and usage of shaders. Good tutorials can be found here:
   `http://www.lighthouse3d.com/opengl/glsl/`.
2. An implementation of an abstract shader program class is already provided (see the files shaderProgram.h and shaderProgram.cpp).

Implementation:

1. Implement a vertex and fragment shader that only computes entries to the depth buffer texture in our FBO. Put the shader code into the file shader_depth.h. This file is read and compiled in the file shaderProgramDepth.cpp and used by calling *CShaderProgramDepth::use()* (see *CShadowMap::drawSceneWithShadows()*).
2. Implement a vertex and fragment shader that performs a depth comparison between a fragment's z-value and the depth stored in the shadow map. If the fragment's z-value is greater set the fragment's color to be the shadow color. We can access the texture matrix we stored in *GL_TEXTURE7* using the function *gl_TextureMatrix[7]* in the vertex shader. The result of the multiplication of this matrix with the vertex coordinates can be

forwarded to the fragment shader using a *varying variable*.
Put the shader code into the file shader_shadowmap.h. This file is read and compiled in the file shaderProgramShadowMap.cpp and used by calling *CShaderProgramShadowMap::use()* (see *CShadowMap::drawSceneWithShadows()*).

Optimization:

Based on the optimization in the previous section, we employ its results to consider the mesh transformations when computing the vertex's position in light space. It is important to know that the built-in GLSL variable *gl_Vertex* only contains the vertex position given by the function call *glVertex3d()*. Calls to *glTranslated()* or *glRotated()* are multiplied onto the model view matrix V and can be accessed via the built-in GLSL variable *gl_ModelViewMatrix*. However, this matrix also includes the trackball transformation which has to be overridden by its inverse. This leads to the following extended transformation of the vertex's position into light space: $\mathbf{v}' = L_{proj} * L_{transf} * TB^{-1} * V * \mathbf{v}$, where TB^{-1} is the inverse trackball transformation.