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

Exercise 3 - Shadow volumes

1 The shadow volume algorithm

As part of exercise 2, we have constructed shadow volumes from closed triangular meshes based on the current positions of the light sources. In this exercise, we want to employ those shadow volumes to add the global-illumination effect of shadow-casting occluders to our scene.

The z-fail algorithm we want to implement executes the following render passes and steps:

1. Render the scene to initialize the depth buffer and to fill the color buffer with the ambient intensities of the Phong-illumination model in the first pass.
2. Enable the stencil buffer and disable rendering to the depth and color buffers.
3. Render the back faces of the shadow volumes and increment the stencil when the depth test fails in the second pass.
4. Render the front faces of the shadow volumes and decrement the stencil when the depth test fails in the third pass.
5. Render the scene and blend the ambient values already stored in the color buffer with the diffuse and specular intensities of the Phong-illumination model for all the pixels that are not in shadow (the pixel's stencils are non-zero).

Hints:

- Recall that the Phong-Illumination model computes the light intensity as the sum of the ambient, diffuse and specular intensities: $I = I_a + I_d + I_s$. For example, the ambient intensity is the component-wise multiplication of the color channels of the light's ambient intensity l_a and the ambient reflection coefficient of a mesh's material k_a: $I_a = l_a * k_a$. To switch off the diffuse illumination, one can simply set l_d of all lights to zero, assuming that there are more meshes than there are light in the scene.
- Familiarize yourself with the available OpenGL API calls regarding the stencil buffer. For example, information can be found here: `http://www.opengl.org/resources/code/samples/sig99/advanced99/notes/node117.html`. Specifically, *glStencilOp()* and *glStencilFunc* are needed to increase/decrease the stencils and test for non-zero stencils in the respective render passes.

Implementation:

- Implement the functions *CLight::setAmbientIntensity()* and *CLight::setAllIntensitiesButAmbient()* (see the file lightmanager.cpp). Use the function *glLightfv()* and employ the member functions of the class **CLight** to save intermediate results.
- Implement the z-fail algorithm as described above. Therefore, implement the function *CVolumeShadow::m_drawShadowVolumes()* and use it in the implementation of the function *CVolumeShadow::drawSceneWithShadows()* (see the file volume_shadow.cpp). It might be helpful to not disable rendering to the color buffer and use the visualization of the shadow volumes for debugging.
- You can alter the amount of objects and their material properties in the function *main()* (see the file main.cpp). Furthermore, experiment with the amount of light sources and their material properties in the function *CViewer::initialize()* (see the file viewer.cpp).

Optimization:

The z-fail algorithm necessitates the shadow volumes to be capped at the bottom, which is not necessary for the z-pass algorithm. One method to avoid the clipping plane to *slice open* the shadow volume is to set both the projected vertices of the shadow volume as well as the far clipping plane of the view frustum to infinity. The first is accomplished by setting the w-coordinate of the projected vertex to zero. Note that because of this, the *ShadowVertex* is defined to be a four-dimensional vector (see the file volume_shadow.h). Moving the far clipping plane to infinity requires the projection matrix to be constructed manually.

Recall from the lecture slides that the view frustum is defined by six planes with parameters $left$, $right$, top, $bottom$, $znear$ and $zfar$. If we assume that $right = -left$ and $top = -bottom$, the matrix for the perspective projection is reduced to:

$$P = \begin{pmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & \frac{-(zfar+znear)}{zfar-znear} & \frac{-2zfar*znear}{zfar-znear} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{1}$$

Now, how does the matrix look like if $zfar = \infty$?

Notice that only the third row of P contains the variable $zfar$. Thus, we are searching for the unknowns A and B of the following matrix:

$$P_{inf} = \begin{pmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{2}$$

1. Determine A and B in equation 2 by solving $\lim_{zfar \to \infty} P = P_{inf}$.

Alternatively, the homogeneous notation allows to omit the concept of limits completely because a three-dimensional point at infinity becomes a four-dimensional finite point in the homogeneous notation (with w = 0). To solve for A and B, recall that the perspective projection matrix maps the perspective view volume to the canonical view volume which spans the interval $[-1, 1]$ along all coordinate axes. Thus, a point $\mathbf{p} = (0\ 0\ -znear\ 1)$ has to be mapped to $\mathbf{p}' = (0\ 0\ -1)$ and a point $\mathbf{q} = (0\ 0\ z\ 0)$ has to be mapped to $\mathbf{q}' = (0\ 0\ 1)$ after perspective division. Form two equations and solve for the unknowns A and B.

2. Change your implementation of the generation of the shadow volumes such that the projected vertices are projected to infinity.

3. Replace the call of *gluPerspective()* in the function *StaticViewer::reshape()* (see the file viewer.cpp) by a manual construction of the projection matrix with the far plane at infinity and push it onto the projection matrix stack. Use the API functions *glMatrixMode(GL_PROJECTION)*, *glLoadIdentity()* and *glLoadMatrixd()*. Do not forget to switch back to the model view matrix stack.