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

Exercise 2 - Shadow algorithm prerequisites

In the following lectures, different shadow algorithms will be introduced, e. g. shadow volumes and shadow maps. In this exercise sheet, you should implement some prerequisites that are necessary for the implementation of the shadow algorithms.

1 Shadow volume generation

An object that is lit by a light source (occluder) can be extruded to a so called *shadow volume*, which has the following meaning: Each object that is placed inside the shadow volume lies in the shadow that is cast by the occluder. The idea is shown in Fig. 1.

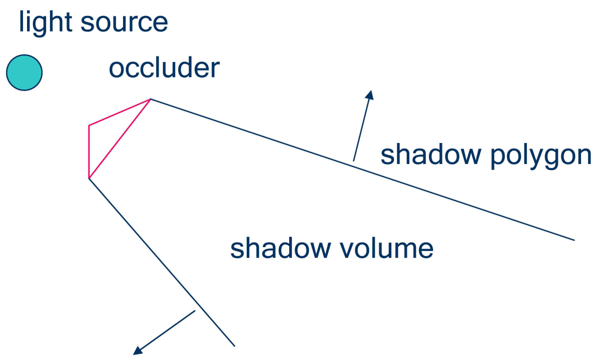

Figure 1: Shadow volume

In this exercise, we want to calculate the geometry of the shadow volumes as a preprocessing step for the shadow volume algorithm.

A simple algorithm would proceed as follows:

1. Loop over all faces and determine whether a face is lit by the light source or not.
2. For each lit face, project a ray through each of the face's vertex to a point at distance 100 and create a new vertex using this point as its position.
3. Extrude the edges of each lit face away from the light to construct the faces of the shadow volume. Therefore, construct two faces that employ the vertices of the edge and their respective projected vertices.
4. Construct two additional faces to close the shadow volume, namely the top and bottom faces. Note that the top face should be translated (i. e. by a 1/1000th unit) in the negative direction of the face normal to avoid numerical problems due to limited floating-point precision in the shadow algorithm later on.

Hints:

- Whether a face is lit by a light source can be evaluated by calculating the dot product of the face's normal and the vector pointing from the face's midpoint toward the light source.
- Think about the orientation of the shadow volumes' faces. Their normals should point outwards.
- The vertices of the mesh's faces are given in the *Face* data structure such that the normalized cross product of the vectors $\mathbf{v}_2 - \mathbf{v}_1$ and $\mathbf{v}_3 - \mathbf{v}_1$ give the face's normal.

1.1 Implementation

1. Make sure that the function *CShadow::instantiateShadow(std::string strMethod)* is called with string *"volume-shadow"* from within function *CLightManager::setHasShadow(int lightId, bool hasShadow)*.
2. Put the code that constructs the shadow volumes into the function *CVolumeShadow::build(int lightId, gmVector3& lightPos)*.
3. Use the data types and structures *ShadowVertex*, *ShadowEdge* and *ShadowFace* (see the file volume_shadow.h) to construct the shadow volumes and store them in the local variable *faceList*.
4. The function *CVolumeShadow::drawSceneWithShadows()* does not realize the volume shadow algorithm in this exercise, but instead visualizes the shadow volumes generated by your implementation and the original objects. Use *glEnable(GL_CULL_FACE)* and *glCullFace(GL_BACK)* to test whether the triangles of your shadow volumes are oriented correctly.

1.2 Optimizations

These tasks are optional, but the second is a good chance to get familiar with the STL container *map*. A good reference can be found here: `http://www.cplusplus.com/reference/`.

1. Compute the projected vertices before looping over the faces to save some redundant computations.
2. Only extrude the *silhouette edges* to construct the faces of the shadow volume. A silhouette edge separates a lit face from an unlit one. Therefore, extend the *CMesh* class by a vector that contains all the mesh's edges. Loop over all faces to find the edges, e. g. in the function *CMesh::loadObjFile(std::string& filename)* (see the file meshload.cpp). Use the STL-map to store each edge only once. Use the edge as the key and give the location of the edge within the edge vector as the value. Use the methods *find()* and *insert()* to find already constructed edges and add new edges, respectively. Note that the comparison operators < and == are already implemented for the *Edge* structure (see the file mesh.h).

2 Shadow maps generation

The shadow mapping algorithm first renders the scene from the light’s position in order to determine the occluders, which are exactly those objects that are visible from the light’s position.

In this exercise, we want to render the scene from the light’s position and provide the results stored in the depth buffer in the form of a texture in the second pass for depth comparisons. Traditionally, this required the copying of the depth buffer data to a texture, which is quite time-consuming.

Frame buffer objects (FBO) provide an efficient solution for so-called *render-to-texture objects*. FBOs allow to define one’s own frame buffer and the buffers within, e.g. a frame buffer that only contains a depth buffer. The depth buffer is generated and allocated similar to a texture and can be handled as such using only a few instructions.

Preliminaries:

1. Familiarize yourself with the syntax and usage of the frame buffer extensions for OpenGL. Good tutorials can be found here:
   `http://archive.gamedev.net/reference/programming/features/fbo1/`
   `http://archive.gamedev.net/reference/programming/features/fbo2/`

2.1 Implementation

1. Make sure that the function *CShadow::instantiateShadow(std::string strMethod)* is called with string *”shadow-map”* from within function *CLightManager::setHasShadow(int lightId, bool hasShadow)*.
2. Construct an FBO that contains a depth and a color buffer.
3. Render the scene with the camera at the light source’s position.
4. Enable texture mapping, bind the color buffer as a texture and render the scene again.
5. Put the code that constructs and deletes the FBO into the functions *CShadowMap::CShadowMap()* and *CShadowMap:: CShadowMap()*, respectively. Use the FBO and visualize the results using function *CShadowMap::drawSceneWithShadows()*. All functions can be found in the file shadowmap.cpp.

2.2 Optimizations

1. Remove the color buffer from the FBO and only render to the depth buffer.
2. Use the function *glColorMask()* to suppress the evaluation of the lighting model.
3. Visualize the depth buffer by reinterpreting the depth values as gray-scale color values. Therefore, bind the depth buffer texture and render a quad with textures enabled. Use the functions *glBegin(GL_QUADS)*, *glVertex3f()*, *glNormal3f()*, *glTexCoord3f()* and *glEnd()* to accomplish this task.