TU BRAUNSCHWEIG
Dipl. Inform. Martin Eisemann
Institut für Computergraphik
Lorenz Rogge (rogge@cg.tu-bs.de)

June 7, 2011

Echtzeit Computergraphik SS 2011
Assignment 8

In this exercise we want to create realistic shadows using shadow volume rendering, as presented in the lecture. Please present your solution to this exercise on Monday, June 20th, 2011.

8.1 Creating shadows using shadow volumes (40 Points)

In this task, we want to implement a technique of shadow volume rendering. In order to do so, we first need to find out, how the shadow volume for a given object looks like. To create a proper shadow volume for each triangle, you need to project every vertex of a triangle along the ray connecting the light source and a particular vertex. In theory the shadow volume is infinitely long. Since we cannot model this in our program, we just use a very long projection along the light-vertex-direction. Figure 1 shows the projection of three vertices P_0, P_1, and P_2 along the directional vectors connecting each vertex with the light source L. The projected points are P_0', P_1', and P_2', which can now be used to create the correct shadow volume for the given triangle.

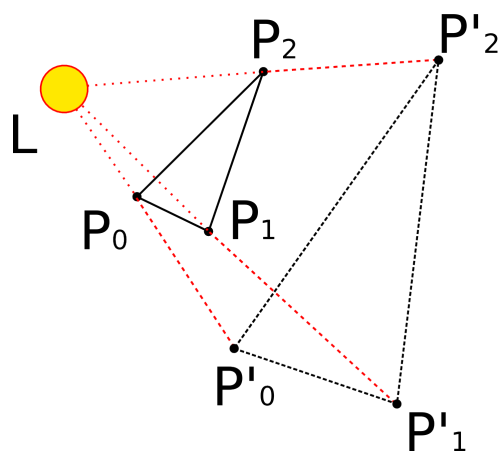

Figure 1: Vertex projection for a given light source and triangle.

Note that all these projections are dependent on the light source's position. When changing the light source position or modifying the geometry (e.g. rotation) the shadow volume has to be computed again. Modify `MeshObj` in the following way:

- Implement the empty method `initShadowVolume`, which allows to generate the shadow volume for this `MeshObj` for a given light source position.

  When creating the shadow volume, clone the existing vertex data array and add a projected version of each vertex at *infinity* (i.e. far away enough from the light source) to this array. After all vertices have been projected along their light-ray-directions, iterate over all faces to compute the corresponding shadow volumes.

Be sure to process the original faces in the correct vertex order. Create the shadow volume using 6 new triangles created from the original vertices P_i and the projected correspondences P_i' (you need to create two more triangles to cap the volume at both ends, if you want to implement the better performing *Carmack's reverse* technique). Very important is, that all shadow volume faces are defined with their face normal pointing outwards (CCW). When processing triangles on the back of an object (light source sees the backside), you might need to inverse the vertex order from $(0, 1, 2)$ to $(0, 2, 1)$ to create correct facing shadow volume triangles. Create these triangles by adding the correct vertex indices to a new index array.

When you are done defining the shadow volumes, create a new VertexBufferObject from the cloned and extended vertex data array and an IndexBufferObject from the newly created index array. The member variables for these buffer object are already predefined in the header file (`mShadowVBO` and `mShadowIBO`). Upload the data to these buffer objects like for the rendering routine. Make sure to save the number of indexes for later rendering.

- Having the vertex buffer for shadow volumes set up, implement the rendering of it in `renderShadowVolume`. This works just like the rendering of the object itself, except that only vertex positions and no additional vertex attributes are needed.
- To enable the use of stencil buffers, add the option `GLUT_STENCIL` to the method call `glutInitDisplayMode` at the top of your `main`. This enables the stencil buffer for our rendering context.
- Implement the empty method `renderShadow`, which is being called right after rendering the scene. Render the shadow volumes as presented in the lecture. First initialize the shadow volumes to the current position of the light source. After disabling the rendering to screen and to the depth buffer (`glColorMask`, `glDepthMask`) enable stencil testing and render two passes of your shadow volumes. In the first pass render only the front faces of the shadow volumes and increase the stencil buffer each time the depth-test is successful. Then, in the second render pass, draw all backfaces and decrement the stencil each time the depth-test is successful.

  Finally disable face culling and enable rendering to screen again. Using the provided method `renderScreenFillingQuad` render a black quad over the screen buffer content. Use the correct stencil function to trigger rendering only for those pixel positions where the stencil buffer is not equal to zero. You may want to use blending (`glEnable(GL_BLEND)` + `glBlendFunc`) to create shadows, that are not completely pitch black.

When you are done, you should see a result like in figure 2.

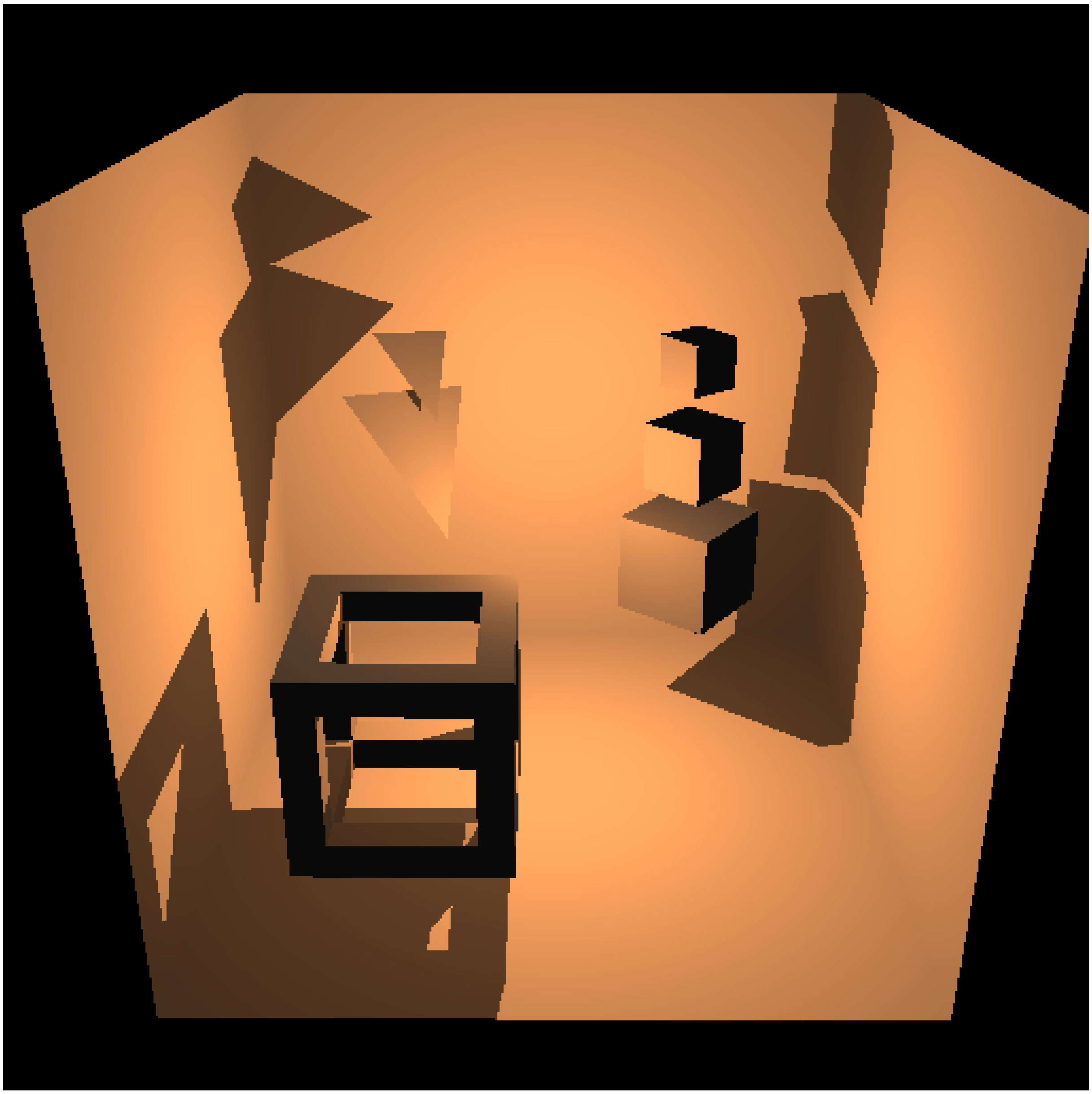

Figure 2: Pixel exact shadows created by shadow volume rendering