Shading Techniques

John Fang, Vincent Lee

May 4, 2018

1 Objective

Our final project proposal was to implement two techniques for creating shadows when rendering a scene, namely shadow maps and volumes. Since we finished early we decided to implement another technique for extra credit, screen space ambient occlusion.

The project is executed by running the binary, passing an OBJ file to load. This file must represent a closed mesh. E.g. `bin/shadow ../assets/gourd.obj‘`.

The idea behind shadow mapping is to first render the scene from the light's "point of view" and save the depth information of this render, which effectively encodes how far away the nearest surfaces would be if you fired rays from the light into the scene. Next, you can draw the real scene, while referencing the depth buffer to check if a fragment is in shadow. For each fragment, first compute the distance between the fragment and the light. Second, use the depth buffer to look up the distance from the light to the nearest surface for a ray shot in the direction of the fragment. By comparing these two distances, you can tell if this fragement is blocked by another object, and thus in shadow.

On the other hand, shadow volumes requires more a few more steps. The idea is that for every triangle in the scene, you can imagine that the light projects an infinite triangular pyramid, which represents the places that the triangle would cast shadow on. Then determining if a fragment is in shadow is equivalent to checking if it is inside one of these volumes. There is an algorithm to do this that utilizes the depth and stencil buffers which, at a high level, counts the number of times a ray fired from the light would "enter" and "exit" the shadow volumes before hitting the surface being drawn. If it has "entered" more times, then it is still inside a shadow volume. Otherwise, the numbers are equal and it has exited all the shadow volumes.

For screen space ambient occlusion, the idea is to, for each fragment, sample its neighbors in camera space. If the neighbors have a depth value that is substantially higher than the fragment's own depth value, then the fragment should have occlusion applied since it is likely to be behind the neighbor object.

In our project, we allow the user to load an OBJ file, and render a scene with triangle mesh and a checker-patterned floor. There is a hard-coded number of lights, although we are able to handle multiple lights for both techniques.

2 Implementation Details

2.1 Shadow Maps

For shadow maps, we first create an array of 2D textures to serve as the depth maps. Then, for each light, we initialize a separate framebuffer to render into. Each layer of the texture array is attached respectively to a corresponding framebuffer, such that rendering to the framebuffer will write depth information into the texture layer.

At render time, we first bind the texture array. Then, we loop over all lights. First, the framebuffer corresponding to that light is bound, and we compute a view matrix, as if the camera were at the light's position with the light's direction. From this point of view, the floor and object are rendered, writing depth information into the texture. The combined MVP matrix is also saved for each light for use later.

After all depth maps have been created, we render the main scene. The fragment shader for the floor is passed the 2D texture array, and the MVP matrices for each light. This allows it to, for each pixel, determine the UV of that pixel in the depth maps. Looping over each depth map, if the pixel is deeper in the fragment shader than in that depth map, then the shadow darkening factor is increased. We implement percentage-closer filtering, meaning the depth maps are sampled in a coarser area (5 x 5 in our case) and averaged instead of at a single pixel to obtain smoother shadows. We also solve the shadow acne problem (where there are precision issues comparing the fragment's depth and the depth map's value) by adding a slight constant bias to the fragment depth before comparison. Using a constant bias has the limitation that it can cause *peter-panning*, where the shadow can detach from the object when the light is in certain positions.

2.2 Shadow Volumes

For shadow volumes, we likewise first create an array of 2D textures to hold intermediate color values. Then, for each light, we initialize a separate framebuffer to render into. Each layer of the texture array is attached respectively to a corresponding framebuffer, such that rendering to the framebuffer will write the color output into the texture layer. We also allocate a depth and stencil buffer for each framebuffer.

Additionally, when the object is loaded, we compute full adjacency information for the model, meaning that for each triangle sent to the GPU, we also send three extra vertices identifying its neighbor triangles. Thus, this limits our renderer to meshes that are closed, meaning each triangle edge has exactly two neighboring faces.

At render time, we first bind the texture array. Then, we loop over all lights. First, the framebuffer corresponding to that light is bound. Then, we draw the scene as-normal into the depth buffer. We then set the stencil operation to increment the stencil value if a back face fails the depth test, and decrement the stencil value if a front face fails the depth test. We then compute the shadow volume of the object for that light. This computation is done by finding the silhouette edges, edges of the triangle whose two neighboring faces have opposite dot product signs to the light. Given the two vertices defining a silhouette edge, we can get a quad by projecting those vertices to infinity along the light direction. Caps for the volume are also generated similarly by taking the triangle face itself and its projection to infinity along the light direction. When all these quads are combined, a shadow volume results, which we draw to the stencil buffer. We then render the scene for real into the color buffers, but with stencil testing set such that only pixels where the stencil buffer is 0 will be drawn. This will leave areas in shadow for the given light as black, and colored otherwise.

After finishing this process for all lights, we then use a simple shader that draws a quad on the screen, sampling from each layer of the texture and summing the result together.

2.3 Screen Space Ambient Occlusion

On startup, a set of 64 random sample vectors in the hemisphere pointing towards positive Z is generated. The vectors are then quadratically scaled so that there are more samples near the origin. Then, a set of 16 random rotation vectors are generated and stored in a 4x4 texture. This is used to randomly rotate the hemisphere above.

Each frame, we render the entire scene into a set of textures, storing the camera space position, normals, and diffuse colors. Then, in a second pass fragment shader, for each fragment, we rotate the hemisphere randomly (by tiling the 4x4 rotation texture across the screen and sampling it), orient the hemisphere so that it corresponds with the camera space normal, then add the result to the fragment position to yield 64 camera space neighbor positions to sample. If the neighbor Z value is greater than some epsilon than the fragment's own Z, then an amount of occlusion is added. This second pass produces a 1-dimensional scalar texture of ambient occlusion, but due to how the 4x4 texture was tiled across the screen, has banding artifacts. Thus, we perform a third pass which is just a simple blur fragment shader with radius 2 pixels.

The final pass, we sample the diffuse texture (accounting for Lambert's Cosine Law for each light), then multiply in the shadow factor from the ambient occlusion texture, and draw the result to the screen.