3D-Programming for Civil Engineers

- **Project: Techniques Description**

BLEKINGE TEKNISKA HÖGSKOLA

Joakim Sjöberg 19950420-5577
William Osberg Resin 19970929-2198
DV1542 - 3D-Programming for Civil Engineers
Course Evaluator: Francisco Lopez Luro
Date of Submission: 2019-01-15

Back face Culling using Geometry Shader

When we implemented this technique, we simply added an “if-statement” in the geometry shader of our “geometry pass” which checks the angle between the normal of the triangle and the vector from the triangle to the camera. To calculate this angle we used dot-product between the normal and the vector from the surface to the camera. If that “angle” turns out to be greater than 0. Then we send through the triangle to the next pass. In this way, the triangles that are “backfaced” will not be rendered.

Normal Mapping

Our implementation of Normal Mapping first takes into account that normals exist in the [-1,1] range. Because a texture usually stores colors, which are in the range [0,1], we transform the normals into [0,1] range temporarily so that we are able to store the normals in a texture. Because we are using normal maps that already are created we only need to do the second part, namely convert from [0,1] to [-1,1]. This means that when we have read a Normal Map we need to transform the normals back into [-1,1] range before we use them for lighting calculations. We also needed to transform the normals from tangent space to world space so that the lights are calculated correctly. Otherwise the normals would seem like they are pointing in the positive Z direction at all times.This was done by calculating a TBN matrix and multiplying the normals with it.

Particle system with Billboarded particles

We store the information regarding each particle in an array. Because all particles have the same appearance we do not need to save that. Each particles has a “time to live” which decides for how long we should render that particle. We simulate the particles by updating the array each frame. To solve depth and to be prepared for future functionality (for example blending), we sort the particles at the end of each frame so that the particles farthest away are rendered first. In the vertex shader we translate the “base-quad” with the input position variables from the particle. Then we transform the particles to billboards, meaning that the front of the quad is always shown to the camera. To do this, we multiplied the camera’s right-vector with the x position of the vertex. And the camera’s up-vector with the y position of the vertex.

Shadow Mapping

We have implemented Shadow Mapping for point lights, that means that it is omnidirectional. This was done in a separate pass, where all objects were rendered. A camera was mathematically simulated at the

lightsource, and with help of six different shadow matrices the depth for all directions were saved to a depth cubemap. When we later used this cubemap in the light'pass we sampled from this cubemap using the vector to the fragment. The sampled depth value were then compared to the distance from the light to the fragment. We also used a bias to prevent "Shadow Acne". The bias makes a difference when the pixel isn't in shadow. It works by making a small difference when the depth values are equal so that the shadows are consistent.

Bloom (Glow Effect)

In our implementation of Shadow mapping we saved a "Bright Map" from the scene when all the light calculations were done and all objects had been rendered. This was done by setting a threshold of how bright a fragment should be to be saved. This texture was blurred using "Gaussian Blur", and then it was combined with the original scene by adding the color values of the blurred version.

Deferred Rendering

We use an FBO to store three color attachments (Position, Diffuse and Normal). In the first pass(Geometry Pass), we output to all of the colorattachments of the FBO, which means that after the geometry pass, we have three 2D textures filled with information. In the second pass (light pass), we force the fragment shader to run for each pixel in the screen by rendering a triangle that covers the screen. Then we read from the FBO and do the lightning calculations as usual.

Object Loader

Our object loader reads a file from top to bottom. Since we read line by line, the first letter of each line tells us what type of information is on the rest of that line. So we have a bunch of if-statements that stores the information in the correct array.
After we've read and stored all of the vertices, UV's and normals we use them to draw the object.