OpenGL 3

March 13, 2017

Computer Graphics

Deferred shading

Please note that you will need to re-use your code from the first and second OpenGL assignment.

Introduction

The first two OpenGL exercises used a method called *forward shading*; every fragment was immediately shaded after it was placed. Most modern high-end real time graphics applications use *deferred shading* instead. This technique defers the lighting calculations to a later pass. During the first pass, the fragment color, normal and position for each fragment are stored in buffers (no lighting calculations are done yet). During the second pass, all this data is used to determine the final shaded color of each pixel on the screen.
For the last OpenGL exercise, you need to implement deferred shading in your OpenGL application. If everything is done correctly, the output will be the same as it is with the forward shaded implementation.

Advantages

Deferred shading has become standard in modern real time computer graphics. There are several benefits:

- When using forward rendering, each fragment has to be shaded. In practice, many fragments will be behind other fragments. The lighting calculations for hidden fragments are then wasted. Advanced graphics applications may have a complex shading algorithm wich should be used sparsely.
- When multiple light sources exist, a forward shading implementation would need to loop over each light source for each fragment (similar

to the raytracer implementation). Deferred shading does not require this, because lighting calculations are deferred. Instead, in the second pass, only the pixels on the screen that lie inside a light's zone of influence will be shaded. Not every fragment needs to take every light into account. This allows for a much larger number of light sources.

Implementation

The implementation of deferred shading requires the following steps:

1. Render the existing scene (first pass). Instead of rendering a shaded scene, all information required for shading must be rendered. For each fragment, the color, the surface normal and the depth are stored. This is enough information for the second pass to shade the pixels.

2. Shade the result (second pass). For every pixel on the screen, it is calculated where the original coordinate of that fragment was. Distance from light sources to that point can be used to calculate light influence, and the color and normal of that pixel can be read to perform phong lighting.

An example is shown in Figure 1. The color of each fragment is stored in the first texture. The second texture contains the normals. The third texture is the Z-buffer (it's red, because it has only one channel). The final image is a combination of these three buffers and the light sources in the scene.

Two shader programs are needed for this exercise. The currently existing shader should be modified to render to three targets, and a new shader should be created in which the shading will be implemented.

Multiple render targets

The first pass of deferred shading requires multiple render targets. Your current fragment shader outputs a color to only one location (E.G. `layout (location = 0)out vec4 color`). As seen in image 1, deferred shading requires three outputs. Outputs for locations 1 and 2 therefore also need to be defined in the fragment shader. Writing to multiple outputs is called using *multiple render targets*, which exists in OpenGL since version 3.

In OpenGL, the shader program renders to the active framebuffer object, or *FBO*. Your program currently renders to the default framebuffer object which is the window, it does this by default. This needs to change; you will need to create an FBO with three buffers (the three render targets mentioned above) which will be bound initially. When this FBO is filled with data, the second pass will render the shaded result to the default framebuffer again.

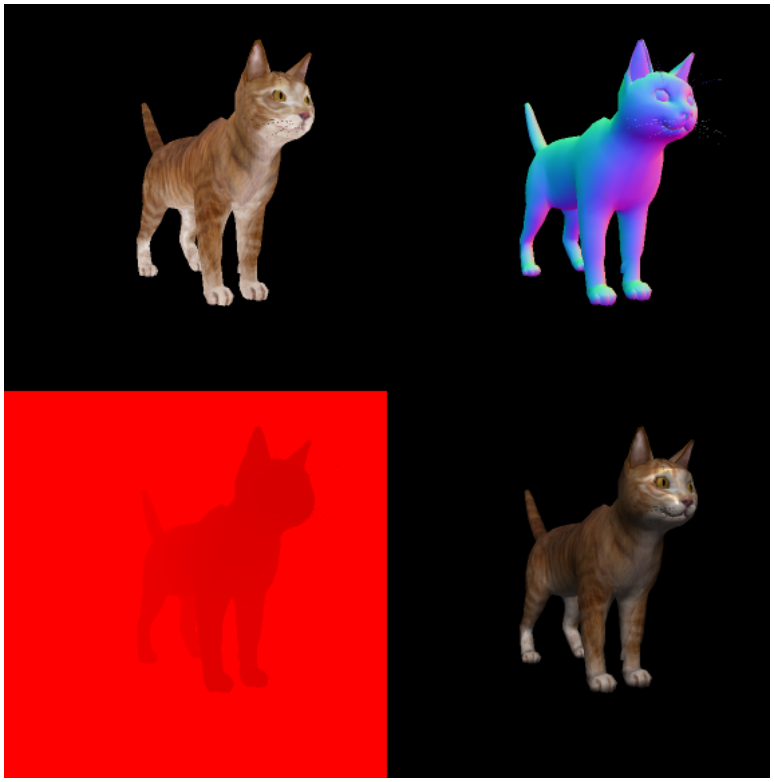

Figure 1: A cat rendered using deferred shading.

Creating the framebuffer object

The three render targets associated with this FBO are called *gBuffers*. These are textures just like the one used in the previous OpenGL exercise. They will be attached to the FBO. The framebuffer can be created in the *createBuffers* function of the class *MainView*.

1. Create three textures (gBuffers) using `glGenTextures()`.

2. Initialize the textures using `glTexImage2D()` (Don't forget to bind each texture using `glBindTexture(GL_TEXTURE_2D, <texPtr>)` before working on it). More information about this function and its parameters is explained in the previous exercise. Because we are going to write to these textures, we do not need to upload an image to them. The last parameter can therefore be *NULL*. Make sure they have the size of the OpenGL window (**Note!**also update if the screen is resized!), since they need to encode data for every pixel in the window. The first two textures can have *GL_RGB* as a format, because the color and the normals both require three floats (RGB and XYZ) to store. The third texture however will store depth. This is a special format. The type

for the depth gBuffer should be *GL_DEPTH_COMPONENT*.

3. The gBuffers are never stretched or transformed. No linear interpolation between pixels required. Code snippet 1 shows how this can be disabled. Make sure these functions are called while the texture is still bound.

4. Create a framebuffer object using `glGenFramebuffers()`. Note that this function uses the same syntax as functions like `glGenTextures()` and `glGenVertexArrays()`.

5. Bind the framebuffer using `glBindFramebuffer(GL_DRAW_FRAMEBUFFER, <fboPtr>)`. This syntax should again be familiar, it's the same as vertex buffer objects and vertex attribute objects. Do not forget to unbind the framebuffer when you are done. While this framebuffer is bound, all rendering will be done on the bound framebuffer object. It will not be visible in the window until you bind the default framebuffer again and render to it.

6. We now need to associate the previously created gBuffers with the FBO. This is done using the function `glFramebufferTexture2D()`, which takes the following parameters:

   (a) Target to which the framebuffer is bound, in this case *GL_DRAW_FRAMEBUFFER*.

   (b) Attachment. This should be *GL_COLOR_ATTACHMENT0* for the first texture and *GL_COLOR_ATTACHMENT1* for the second texture. The third (depth) texture requires *GL_DEPTH_ATTACHMENT*. If you would want to add more render targets, up to 16 color attachments can be added in OpenGL 3.3.

   (c) A texture target, which should be *GL_TEXTURE_2D*. This is the type of gBuffer textures we have created earlier.

   (d) The texture.

   (e) The mipmap level of the attached texture. We don't use mipmapping for our gBuffers, so this can be zero.

7. The final step of the framebuffer object creation is to specify which color attachments to draw to. This can be done using the function `glDrawBuffers()` which takes two arguments. The first argument is the number of color attachments to draw to, which is two in our case (color and normals). The second argument is an array of `GLenum` constants to specify which color attachments to draw. Code snippet 2 shows how this can be done.

Code snippet 1: `Texture filtering for gBuffers`

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_NEAREST);
```

Code snippet 2: `Configuring an FBO's draw buffers`

```
GLenum drawBuffers[2] = {GL_COLOR_ATTACHMENT0,
    GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, drawBuffers);
```

Rendering to the framebuffer object

After you bind the newly created FBO during rendering, all OpenGL render calls will render to your buffer. You will need to change the fragment shader. It should no longer perform lighting calculations and output data to the gBuffers. The vertex shader however doesn't need to be changed; the transformations are still the same.

1. To start the first rendering pass, bind the new FBO.
2. Change your fragment shader. Instead of outputting its shaded color at `location = 0`, it should output its color at location 0 and its normal at location 1. Beware though, that the output vectors only take floating point values between 0 and 1. The normal's coordinates range from -1 to 1. You will have to scale and shift them to fit in the 0 to 1 range. You do not need to output to the depth attachment explicitly; this is done implicitly by OpenGL.
3. Bind the default framebuffer again. The second pass starts now.
4. Bind the shader that will perform the actual shading.
5. Render a full screen quad (ranging from $(-1, 1)$ to $(1, -1)$).

Deferred lighting

After implementing the previous steps and binding the FBO, your gBuffers should contain data required for deferred shading. The second shader can now be implemented.

Rendering a full screen quad

To test whether the last steps were successful, you can make a very simple shader first: just output the color of one of the gBuffers. This should yield

either one of the first three images in figure 1. The source of this simple shader is printed in Code snippet 3.

Code snippet 3: `Rendering a gBuffer - fragment shader`

```
#version 330 core

uniform sampler2D diffuse;
uniform sampler2D normals;
uniform sampler2D depth;

in vec2 uv;

layout (location = 0) out vec4 fColor;

void main()
{
    fColor = vec4(texture2D(diffuse, uv).rgb, 1);
}
```

The vertex shader only needs to provide a UV coordinate to sample the texture. It should render a quad from $(-1, 1)$ to $(1, -1)$, and the UV coordinates should range from $(0, 0)$ to $(1, 1)$ to span the entire texture.

Note that there are three sampler uniforms instead of one. The three gBuffer textures must be bound using `glBindTexture()`. Make sure that these three textures are not all on *GL_TEXTURE0* but range from *GL_TEXTURE0* to *GL_TEXTURE2* instead. The active texture can be selected by calling `glActiveTexture()` as explained in the second OpenGL exercise. The example only renders the diffuse texture, but you can sample any of the three textures.

Shading

When you can read from your gBuffers, you have all information required for shading. Previously, the light location was passed to the shader using a uniform variable. This can be done again, but now to the second pass shader.

1. To determine the light direction, you will need to know what the original location of the fragment was. You will have to inverse the projection. The current location was reached by multiplying the matrices with the vertex position:
   $projection * view * model * vertex = fragPos$.
   This process can be reversed:
   $vertex = fragPos * inverse(projection * view * model)$.
   The fragment position consists of two components. The first two elements are the screen space coordinates in the range $[-1, 1]$. These can be passed from the vertex shader which renders the full screen quad

using these coordinates. The third element is the value from the Z buffer, which can be read using `texture2D()`. Note that the Z value originally ranged from -1 to 1 but was mapped to $[0, 1]$ by OpenGL to fit on the texture. Reverse this process by $vertex.z = z * 2 - 1$. The fourth value is 1, just like the vertex coordinate in the vertex shader.

2. The formula above mentions the glsl function *inverse*. 4x4 matrix inversion is a computationally expensive operation to perform for every fragment on the screen, especially since the result is the same for every pixel. It's better to calculate the inverse every frame on the GPU and pass it as a uniform to the shader.

3. Since the original fragment position is known, you should now be able to perform phong shading using the same formula as before.

Deadline

See Nestor (*Time Schedule*). Details on how to submit your work can also be found on Nestor (*Lab Assignments*).

Assignment submission

Please use the following format:

- Main directory name: `Lastname1_Lastname2_OpenGL_3`, with the last names in alphabetical order, containing the following:
- Sub-directory named Code, containing the modified Qt framework (please do **NOT** include executables)
- Sub-directory named Screenshots with your screenshots or videos.
- README, a plain text file with a short description of the modifications/addings to the framework along with user instructions. We should not have to read your code to figure out how your application works!

The main directory and its contents should be compressed (resulting in a .zip or .tar.gz archive) which is the file that should be submitted (using the *Assignment Dropbox*). Example: the name of the file to be submitted associated with the first OpenGL assignement would, in our case, be `Kliffen_Talle_OpenGL_3.tar.gz`

Assessment

See Nestor (*Assessment & Rules* $\rightarrow$ Assignment assessment form).