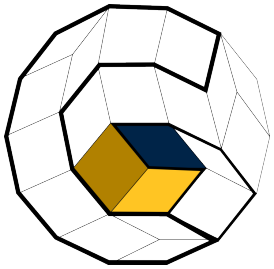
TU BRAUNSCHWEIG
Dr.-Ing. Martin Eisemann
Institut für Computergraphik
Lorenz Rogge (rogge@cg.tu-bs.de)

June 18, 2012



Echtzeit Computergraphik SS 2012
Assignment 9

Present your solution to this exercise on Monday, June 25th, 2012.

The exercises are going to take place in the CIP pool, room G40 in Mühlenpfordstraße 23. Please make sure your solutions compile and run on the CIP pool computers. Note that you need a y-number, which can be obtained at the Gauß-IT-Zentrum, to use the computers. If for some reason you are not able to attend the exercise, you may send your solution to rogge@cg.tu-bs.de instead.

In this exercise we will make use of deferred shading to accelerate the given rendering code. The provided program already fully implements a normal mapped rendering of 225 complex objects. 10 lightsources are used within the scene, which again can be dynamically switched on or off. Using the techniques learned so far, this rendering of almost 4 million triangles is *very* slow. To accelerate this we will implement deferred shading an rendering into frame buffer objects (FBO). Using the command line parameter "1" when starting the application triggers the deferred shading routine instead of normal rendering.

9.1 Creating a FBO and attached textures(20 Points)

The most important thing for deferred shading is to create an enable a frame buffer object. We also need to attach several textures to this render buffer.

The FBO in this exercise needs to be able to handle depth information, since it is needed when rendering the scene the first time. We don't need to access this depth information by ourselves, but the FBO needs to have an attachment to the depth component.
The given code already provides two global handles:

```
GLuint fbo;
GLuint rb;
```

These represent the frame buffer object an the attached depth buffer as render buffer object.

Complete the missing code in `initFBO()`. Generate a new frame buffer object and bind it.
For depth information, create and bind a render buffer object `GL_RENDERBUFFER` in the size of your current window and attach it to your FBO.
Also, for every vertex attribute you want to export from your first rendering to the second rendering pass, you will need to have a separate texture. Implement the missing code in `createEmptyTexture(...)` to create an empty texture in the given size. Use `GL_RGBA32F` as texture format, `GL_RGBA` as internal format, `GL_FLOAT` as internal data type and pass a `NULL` pointer to the data position, when setting up the texture. This now enables you to create empty texture objects at the size of your window, which you can now attach to the color attachments of your FBO. (Note that textures can be accessed easily using another prefedined map in this exercise. Simply assign every texture a proper id-string and access the regarding texture object via `textures[texID]`).

Now that you have created your FBO you can use it for deferred shading in the following task.

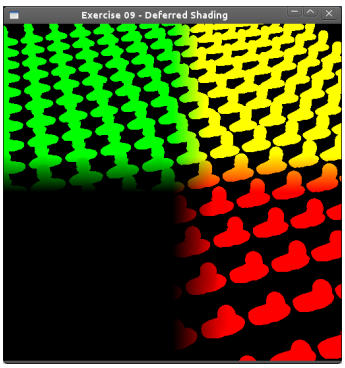
9.2 Deferred Shading (30 Points)

Deferred shading decouples the actual rendering of the geometry of the shading process for every pixel. In very complex scenes with many overlapping objects, it is not neccessary to compute the shading for pixels, that will be covered by another object anyways. We cannot really control the order, that the objects will be rendered in, so it might happen, that the visible pixel is rendered last after several other rendering of now occluded objects. To reduce this overhead, deferred shading renders the geoemetry first, storing only the important vertex (or pixel) attributes in separate texture layers (see figure 1). In a second rendering pass, the previously created textures will be mapped to a simple screen-filling quad. Instead of texturing this quad directly with the given textures, the per-pixel-data will be used to evaluate the shading computation of the first render pass. Now only the truely visible pixels are processed and shaded. This speeds up the rendering a lot.

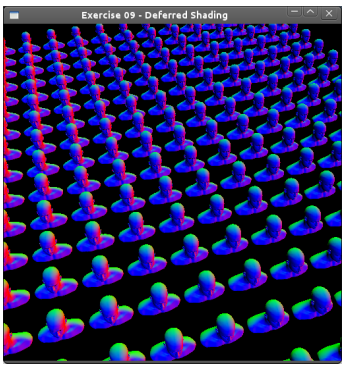
The given exercise uses normal mapping and one color texture layer for object shading. So the per-pixel-information that we need for the shading in the second pass will be:

- The position of the pixel in camera space.
- The surface normal of this pixel. (Since we are using normal mapping, we store the already modified surface normal here. So the normal map has to be accessed in the *first* render pass).
- The texture coordinate used by this pixel.

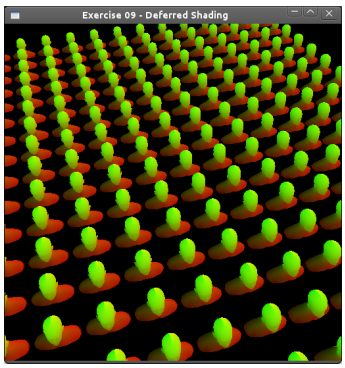
Implement the shader used for the first rendering pass in `deferred_pass1.vert` and `deferred_pass1.frag`. This shader has the original vertex data a input, transforms it as usual according to the modelview and perspective transform. Since we need to compute the modified surface normal in this pass, too, transform the provided normal, binormal and tangent information in the vertex shader using the normal matrix and pass it to the fragment shader. In this shader, compute the tangent space matrix as in the last exercise, but now transform the normal given in the normal map into camera space and not the other way around. Use the inverse of the tangent space matrix for this conversion.
Now, that you have all the data needed, pass the values for pixel position, normal and texture coordinate to the color outputs of your shader. Make sure to bind these position to proper color attachment positions after you compiled your shader in the `C++` code (Hint: `glBindFragDataLocation(...)` in `initShader()`). So everything you need in the first rendering pass is geometry data, the modelview matrix and the projection matrix and the normal map, as it is being accessed in the fragment stage of the first pass.



(a) Pixel positions in camera space



(b) Normals in camera space



(c) Texture coordinates

The second rendering pass finally does the shading of the pixels. Now we need to pass the information about light, material and surface texture to the shader.
Implement the missing parts in `deferred_pass2.vert` and `deferred_pass2.frag`.
The rendering will be triggered by a screen filling quad, so that ever screen pixel will be evaluated exactly once.
In the vertex stage, simply process the given geometry and pass it to the fragment stage. Also transform

the lights positions into camera space using the provided `view` matrix, being the modelview matrix for the lights. The modelview and projection matrix used here are different from the matrices of the first pass. To render the screen filling quad, we use a simple orthogonal projection instead of a perspective one. Also the modelview matrix is simply the identity.
Pass the transformed light positions as well as vertex texture coordinates to the fragment stage.

In this stage, get all the needed information from the three textures of the last rendering pass (position, normal, texture coordinate) and use them to compute the shading as usual.
Use the texture coordinate provided by the attribute texture to access the surface texture at the correct position. Combine the material shading with this texture value to shade your final pixel.

One problem still remains: When you render your screen filling quad, you will need to find a way to decide, whether the pixel you are processing was covered by geometry in the first rendering pass or not. Find a way, to evaluate this. If a pixel has not been covered by any geometry, you can simply `discard` it in your fragment shader of the second pass.

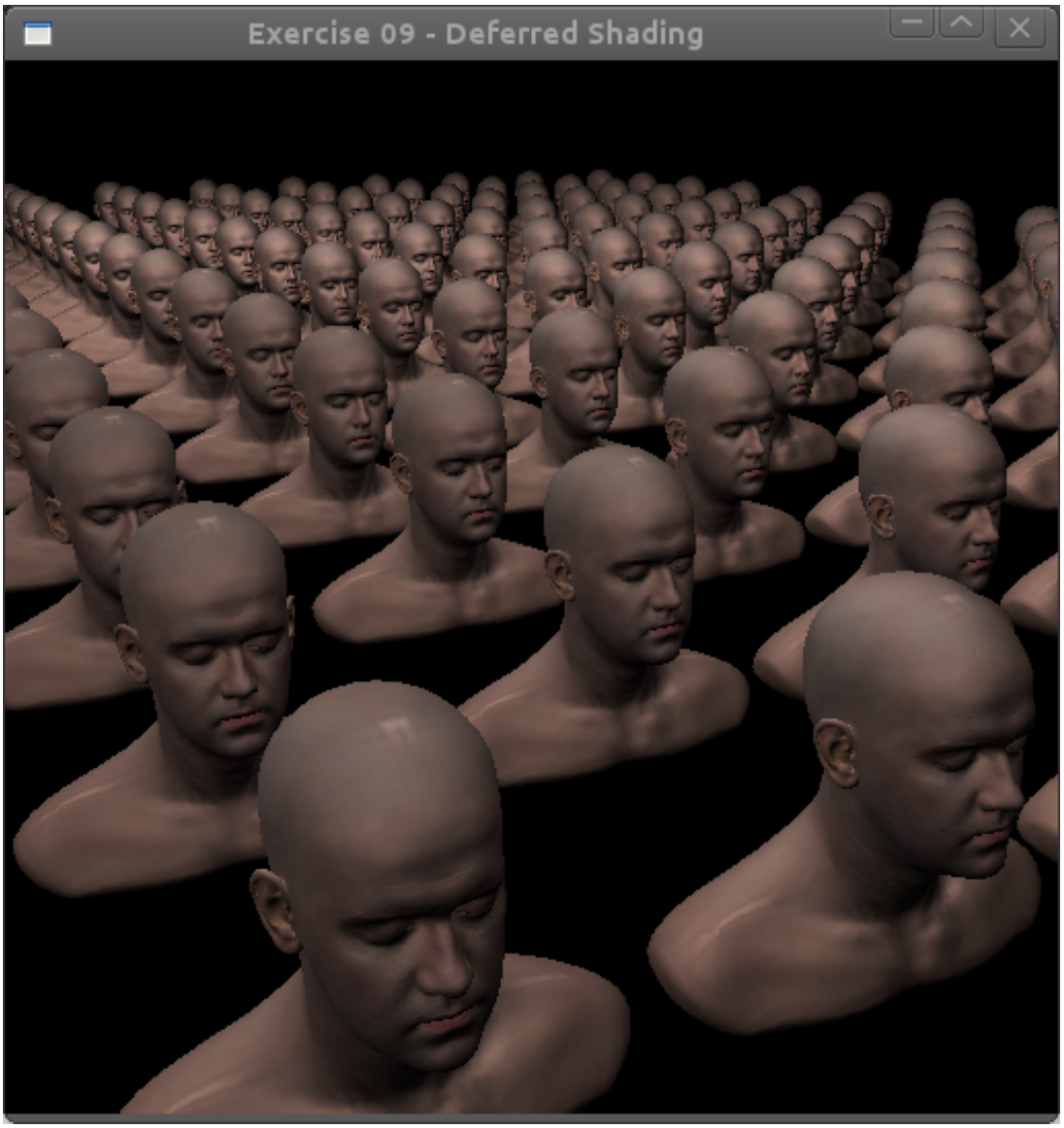


Figure 1: Final shading after second render pass