ME-C3100 Computer Graphics, Fall 2013

Lehtinen / Peussa, Hölttä

Programming Assignment 5: Real-Time Shading

Due Wed Dec 11th at 23:59.

It's time for the last assignment of the course! This time you'll be writing OpenGL shader code to render a reasonably detailed model of a human head in real time, and making use of texture and normal maps. The requirements are quite easy this time, but that leaves you with more time and freedom to do interesting extra credit work. All the required code on this assignment will be in GLSL rather than C++.

Requirements (maximum 10p) ***on top of which you can do extra credit***

1. **Sample diffuse texture (1p)**
2. **Sample normal texture (1p)**
3. **Blinn-Phong diffuse shading (2p)**
4. **Blinn-Phong specular shading (2p)**
5. **Normal transform insight (4p)**

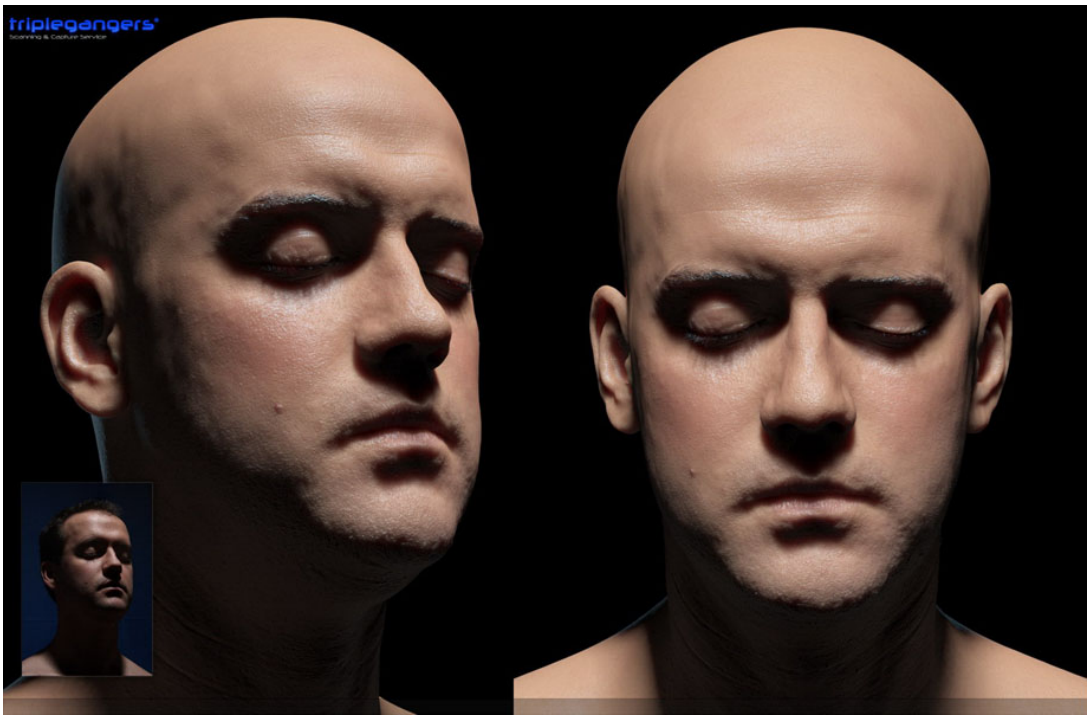

Figure 1: If your rendering looks this good, you'll get lots of points.

1 Getting Started

Take a look at the assets we'll be using to render the head in the `head/` folder. There's a mesh file, material definition file, texture and normal map images.

If you want your head to look a lot nicer, download hi-res texture and normal maps here: https://mediatech.aalto.fi/~jaakko/ME-C3100/S2013/head-maps/
To use them instead of the low-resolution assets, you have to make a trivial modification to the `head.mtl` file. It's plain text; just change the map file names.

You'll write all requirement code into the fragment shader `pshader.glsl`. Note that you can reload the shaders while the application is running by pressing enter; this will cause the shaders to be recompiled and reloaded. If you write an error in a shader, you'll get error messages at runtime from the shader compiler - Visual Studio won't warn you. Watch the console window for possible compilation errors.

If you haven't written GLSL code before, no worries; it is actually easier than doing math in C++. The previously released math handout (under "Additional reading" in Noppa) has a short section on GLSL at the end, with links to many good references. There is a really nice reference card available; GLSL information is in pages 8-11. The full list of separate versions might also be interesting.

2 Rendering, GLSL and the GPU

This section goes deeper than necessary for the requirements, and touches on code you don't have to modify, but at least skim through it. When writing shader code, we are operating closer to hardware than at previous points of the course. It's useful to have at least some idea of what's happening under the surface.

2.1 Running shader programs

There is a relatively small amount of C++ code in this assignment; most of it just feeds data and instructions to the GPU hardware. Take a look at the `renderWithNormalMap` function. It uses a lot of `FW` functionality in addition to raw OpenGL, but you should be able to follow what happens. *Attributes* are per-vertex values that are fed to the vertex shader, such as positions and normals. *Uniforms* are constant values that do not change during a single draw call, e.g. camera and light positions. *Varyings* you will not see in the render function, but in the shader source; they are outputs of the vertex shader, and inputs for the fragment shader.

The mesh object that we use contains lots of data and functionality, but we rewrite some of it here to show what actually happens, and also because its default draw function does not support our normal maps. `FW`'s meshes consist of one or more submeshes; each vertex of a particular submesh shares the same material properties. Many OpenGL calls are hidden inside the `GLContext`, `Mesh` and `Program` objects; you can take a look at their

source for OpenGL API specific details.

The uniforms are set via the `GLContext` object (`setUniform()` method). The call needs an identifier for the particular uniform value, and the value itself. When the shader is compiled, OpenGL assigns each uniform and attribute a unique number. These numbers can be retrieved with the `getUniformLoc()` and `getAttribLoc()` calls from the shader program object, as you see in the code.

A texture is sampled in a shader with a *sampler*. (This usually includes a lot more than just fetching one value - think back to discussions about interpolation and mipmaps on the lectures.) The OpenGL API mostly uses plain integers as identifiers; here, the diffuse and normal samplers are set to the values 0 and 1, respectively. These correspond to the index given to `glActiveTexture()` call that appears in the submesh loop (`GL_TEXTURE0` for index 0, etc.) The texture data (previously uploaded to the GPU by `Mesh`) is *bound* to the active texture that is itself nothing but a number.

The vertex shader in the base code does not do anything interesting; it simply passes the positions, colors, normals etc. to the *varying* variables that work as inputs to the fragment shader. The values received by the fragment shader are results of interpolation between the values of the three vertices in the triangle, so that the inputs vary smoothly from one fragment to the next.

The fragment shader (also called a pixel shader) is what does the actual work in this assignment. It determines the final color of each pixel on the screen. It's where you will write the texturing and lightning.

Producing a binary program from source code is composed of stages that are common for both GPU and CPU. Compiling a normal C or C++ program is done with a separate compiler program (such as Visual Studio's compiler or GCC) and it produces an executable file; shaders are compiled with the drivers produced by your graphics card's vendor and they are run on the GPU while rendering. GLSL's compilation model resembles that of C: the first compilation stage does not actually produce a program but an object; all the objects for separate pipeline stages (where vertex and fragment shaders run, for example) are then linked together to a usable program.

The linker decides memory locations for the variables used by the shaders; some of them are decidable by the user, and some are queried later. Linking connects the data and objects for separate shaders to a single program, so that the programs actually receive the data that you pass around. The `GLContext` object's `getUniformLoc()` retrieves these locations; that's why it has to be repeated many times while setting the variables - the GPU knows nothing about the program running on the CPU and its variables.

2.2 Interpolation is your friend

Again, the varyings are defined in the vertex shaders for each visible vertex; while rasterizing the scene to the screen, they are interpolated linearly (taking account the perspective, as in the lecture slides). Think barycentrics! See e.g. figure 2 where the colors are interpolated with just three colored vertices.

Interpolating the normals for every pixel is what enables shading models like Phong to produce rather smooth-looking renderings even with models with low vertex counts; look at figure 3, and see the bunny in slide 52 in the shading lecture slides.

The same mechanism is also at work when texturing models. First, UV coordinates are defined at every vertex, e.g. by an artist using some modeling tool, and then they get automatically interpolated when passed from the vertex shader to the fragment shader. These coordinates can then be used as x and y coordinates to sample a texture image from the correct location for every individual fragment.

In the assignment code, vertex colors, normals and UV coordinates are passed in the standard way as attributes, one of each per vertex.

Figure 2: Three colors (red, green and blue) defined, others interpolated. This happens when vertex shader outputs its color to a varying, and pixel shader outputs its color directly from an input varying of the same name.

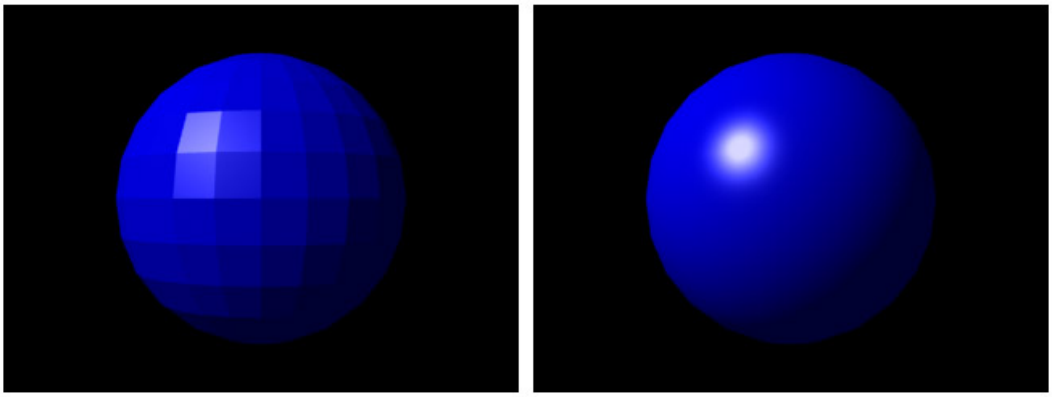

Figure 3: Flat vs. phong shading: same mesh, different normal computation. Phong interpolates the normal linearly between vertices; in flat shading, the normal stays the same throughout the whole face.

3 Detailed instructions

R1 Sample diffuse texture (1p)

With the program running, switch to "Debug render: diffuse texture" mode. Fly around with the WASD keys and the mouse. You'll see a flat color. Looking at the fragment shader `pshader.glsl`, you can see that it comes from a uniform diffuse color set for the whole material multiplied by interpolated vertex colors; since the result is flat, obviously the vertex colors in the model are currently all the same.

Instead of using a constant diffuse color for the whole head, or colors interpolated based on vertex colors, we want a diffuse texture to determine the diffuse color of the fragment, as in figure 4. The C++ code has already loaded the texture onto the GPU, and assigned a GLSL texture sampler (of type `sampler2D` as it samples using 2D texture coordinates, and named `diffuseSampler`) to sample it. Use the texture sampler to get the diffuse color. If you aren't sure how to do this, or just want to see a good exposition of basic OpenGL texturing, check out the texturing chapter of McKesson's textbook. Also, take a look at page 11 of the reference card linked in the getting started section; it describes the function prototypes of all the texel lookup functions. We need just the simplest one. Pay attention to the vector sizes - colors have four components, where the last one is the (often unnecessary) alpha channel.

Figure 4: Plain diffuse colors directly from the texture

R2 Sample normal texture (1p)

With the program running, switch to "Debug render: final normal" mode. You see a visualization of normals. Inspect the fragment shader code, and you'll see this normal data comes from interpolated vertex normals. These normals are very smooth, but the surface of the skin isn't really actually smooth at all; if you implement light-based shading based on this data, the resulting final head render will also look unnaturally smooth. Let's use a normal map instead. Switch into "Debug render: normal map texture" mode to verify no normal map is currently being loaded. The data we want is already loaded onto the GPU as a texture, and `normalSampler` in the shader is configured to sample it. Write the normal map sample into `normalFromTexture`. Note that since textures are originally designed to represent colors, the data in them is in the range [0, 1]. Normal components range from [-1, 1], so the normal data has been scaled into the smaller range to get it in a texture. Reverse the scaling after you sample the data. After you are done, reload the shaders and you should see the normal map, figure 5.

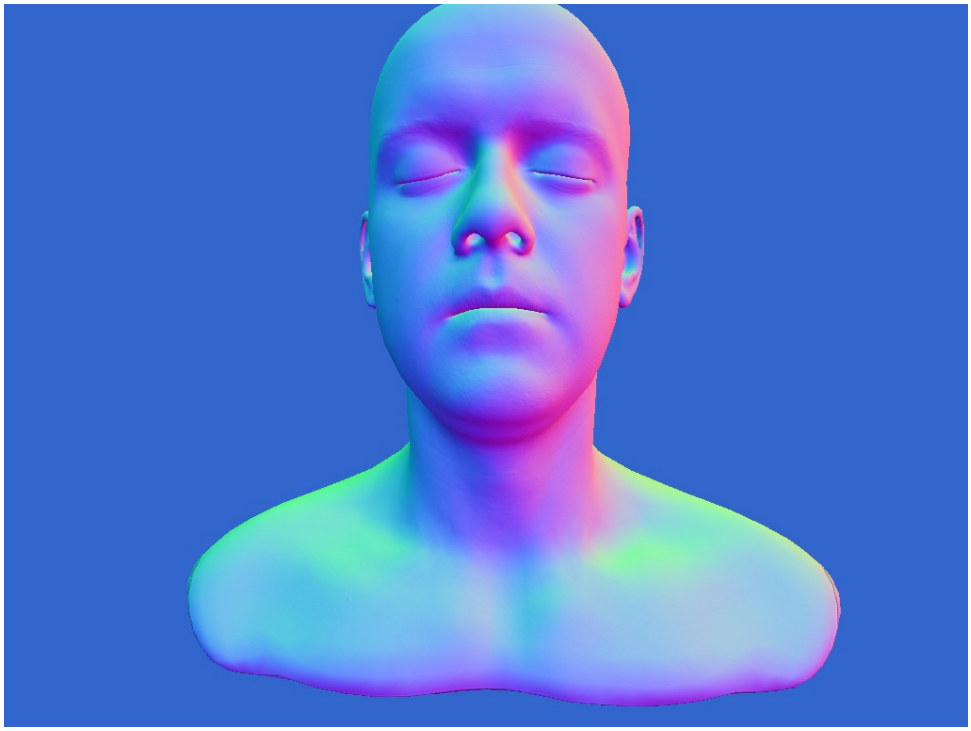

Figure 5: The head colored by the normal vectors

Finally, transform the new normal to camera space and assign into `mappedNormal`. These are the actual normals we'll use for lighting calculations. Use the "Debug render: final normal" mode to verify that your normals are correct. With the visualization we are using, the normal map and the final camera-space normals look very similar when viewing the head from the front; check out the head from the side to see the difference (figure 6). X component in the normal map corresponds to red color, Y to green, and Z to blue.

Figure 6: Head normals from the side. Left: object space, right: camera space. Note how camera space's Z axis points always to the eye, while on the other hand the head's face points to object space's Z. The image on the left has exactly the same colors as the one in 5, if you look closely.

R3 Diffuse shading (2p)

We obtain the diffuse shading exactly like in last assignment, by summing up the contributions of different light sources: (note that vector-to-vector multiplications are dot products; if in doubt, consult the lecture slides)

$$I = \sum_i D_i = \sum_i k_d \cdot \max(0, \mathbf{n} \cdot \mathbf{l}_i) \cdot L_i$$

where k_d is the diffuse color of the material, $\mathbf{n}$ the surface normal, $\mathbf{l}$ the direction to the light and L_i the incident intensity from the light. Note that our shader this time does not apply any ambient lighting. Fill in the missing row in the shader, and you should get the following result:

Figure 7: Diffuse head taking into account the light positions

R4 Blinn-Phong specular shading (2p)

Whereas Assignment 4 called for Phong specular shading, this time we'll use the Blinn-Phong half-vector method. Our total shading becomes

$$I = \sum_i \left[D_i + S_i \right]$$

where S_i is the specular term for the ith light source:

$$S_i = k_s \cdot \max(0, \mathbf{h}_i \cdot \mathbf{n})^q \cdot L_i$$

Here, k_s is the specular coefficient, $\mathbf{h}_i$ is the half-angle vector, $\mathbf{n}$ is the surface normal, and q is the specular exponent. The specular coefficient and exponent are assumed to be constant over our model, and are fed into the shader as `specularUniform` and `glossiness`. The half-angle vector $\mathbf{h}_i$ is the vector that bisects the vector pointing towards the light and the vector pointing towards the camera; for computing it, see the shading lecture slides.

Fill in the missing parts in the shader. Use "Zero Diffuse" in the application to see only the specular lighting, or switch "Zero Specular" on and off to see the difference specular lighting makes in the final image. Compare the two in figure 8.

Figure 8: Left: diffuse again, right: diffuse + specular highlights.

R5 Normal transform insight (4p)

In R2, you transformed the local surface normal from world space to camera space. (The guy's normal map is actually in object space (local to the head), but that's the same as world space when the object shares the same coordinate axes.) This affects how the lights are used in R3 and R4. Explain what would need to be changed in the following, if we did not transform the normals to camera space but would use the world space coordinates instead? The goal is to get an identical picture on the screen. You can use snippets of pseudocode or GLSL code to support your explanation. Don't write an essay; a few concise paragraphs is enough.

For R3: What would you change in computing the diffuse shading, why, and how? You can explain changes in either the shader or the C++ code. Would the light sources change somehow? (2 p)

For R4: How about the specular shading? Would you change something in the shader or in the C++ code? Would you add something? How and why? (2 p)

Hints: look at the "Debug render: final normal" mode, and compare it to "Debug render: normal map texture". How do they change while flying around the object? Also, look at how the light positions are described and what happens to them before they get to the light equations. One further thing you can consider is whether there would be a performance difference.

Write your answers either in the README file or include a separate PDF file.

(You can get extra credit points if you actually write full code for this; see the extra credit section.)

4 Extra Credit

As always, you are free to do other extra credit work than what's listed here — just make sure to describe what you did in your README. We'll be fair with points, but if you want to attempt something grand, better talk to us beforehand. On some items, you have the choice to implement more or less features; we'll give you points based on how much you did.

4.1 Recommended

- **Easy: R5 extended (2-3p)**

  Supply working code for R5 so that lighting computation is done without transforming the normals to the camera space. Add a new button to the UI (next to "Zero specular (for debugging diffuse)") to toggle it on or off. 2p if only the shader is changed, 3p if you do the obvious optimizations for lamp and camera position beforehand in C++.

- **Easy: Point lights (3p)**

  Add point lights in addition to the infinitely-distant directional lights. To actually see the differences, place these lights near the head. The half-vector is computed slightly differently now; you need the position for it.

- **Easy: Moving lights (1p)**

  Animate the light directions (or point light positions, if you implemented them), e.g. make them rotate or wobble around the head in some manner. This is pretty simple, and you might have done similar things already in R0, the animation extra. The effect will be pretty nice; you'll see how the surface reacts to different directions of light.

- **Easy: Subsurface scattering approximation (2p)**

  Modeling subsurface scattering is important for realistic rendering of skin and some other translucent materials. Add a crude approximation of subsurface scattering to your shader using the first method described here:
  http://http.developer.nvidia.com/GPUGems/gpugems_ch16.html
  Give the added light a red tint.

4.2 Easy

- **Color/position variations (1.5p)**

  Tint or move parts of the head a bit locally using vertex and fragment shaders - e.g. make it nod, or become sick and green over time. Supply a free-running timer value as a uniform to the shader, and vary the diffuse color and possibly the vertex positions as a function of time (sine/cosine are handy friends here), localized to some particular position. Just the diffuse color of the whole head would be trivial

to change in C++, but e.g. modifying the color of the nose vertices to reddish or making the ears flap a bit will make use of the parallel computing power of the GPU. The position doesn't have to be super-accurate – just experiment and find the approximate body part boundaries by trial and error, for example.

4.3 Hard

- **Better subsurface scattering (? p)**
  Implement some subsurface scattering technique that gives better results than the trivial one in the recommended section. Following the same link gives you several alternative techniques such as ones using depth maps. You don't necessarily have to implement a technique that results in a better simulation of skin; for example, you could provide an alternative rendering mode that disables the provided diffuse texture and instead renders the head as if it was translucent marble.

- **Shadow maps, self-shadowing (? p)**
  Use a directional light (or several), and use shadow maps to make the head cast shadows on itself. This is particularly noticeable in the nostrils, below the nose and around the lips, behind the ears, and in the neck. Additional points if you render also a floor or a wall and make the whole head cast a shadow on it.

  We will probably release some skeleton code later to help out with this, making it easier to concentrate more on the actual technique and less on OpenGL annoyances.

- **Environment mapping (? p)**

  Nvidia tutorial

5 Submission

- Make sure your code compiles and runs on **Visual Studio 2012 on the Aalto computers**. Comment out any functionality that is so buggy it would prevent us seeing the good parts. If the code does not compile, you *will* lose points.
- Check that your `README.txt` accurately describes the final state of your code. Fill in whatever else is missing, including any feedback you want to share. We were not kidding when we said we prefer brutally honest feedback.
- Package all your code, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive. Do **not** include the `build` folder or the `assignment.sdf` file in the project root; these are huge, automatically generated by Visual Studio, and we have no use for them. Also, do **not** include large non-code files we've given you, like the `.obj` models or this PDF.
- Sanity check (1): look at the size of your ZIP archive. If it's over 10MB, you probably put in something you shouldn't have. Try again. (If you are purposely including large files like your own models, it's fine.)
- Sanity check (2): look inside your ZIP archive. Are the files there? (Better yet, unpack the archive into another folder, and see if you can still open the solution, compile the code and run.)

Submit your archive in Optima folder "Assignment 5".