CS-C3100 Computer Graphics, Fall 2021

Lehtinen / Aho, Kaskela, Kynkäänniemi

Programming Assignment 6: Real-Time Shading

Due Sun Dec 19th at 23:59.

It's time for the last assignment of the course! Please note that this assignment is optional. 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, despite having some hairy looking formulas for the shading calculations. Since the formulas are given to you, all you have to do is to translate them into shader code, which hopefully 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 5p) ***on top of which you can do extra credit***

1. **Sample diffuse and normal textures (1p)**
2. **Lambertian diffuse shading (1p)**
3. **GGX distribution function D (1p)**
4. **GGX masking-shadowing function G (1p)**
5. **Fresnel reflectivity coefficient F_r (1p)**

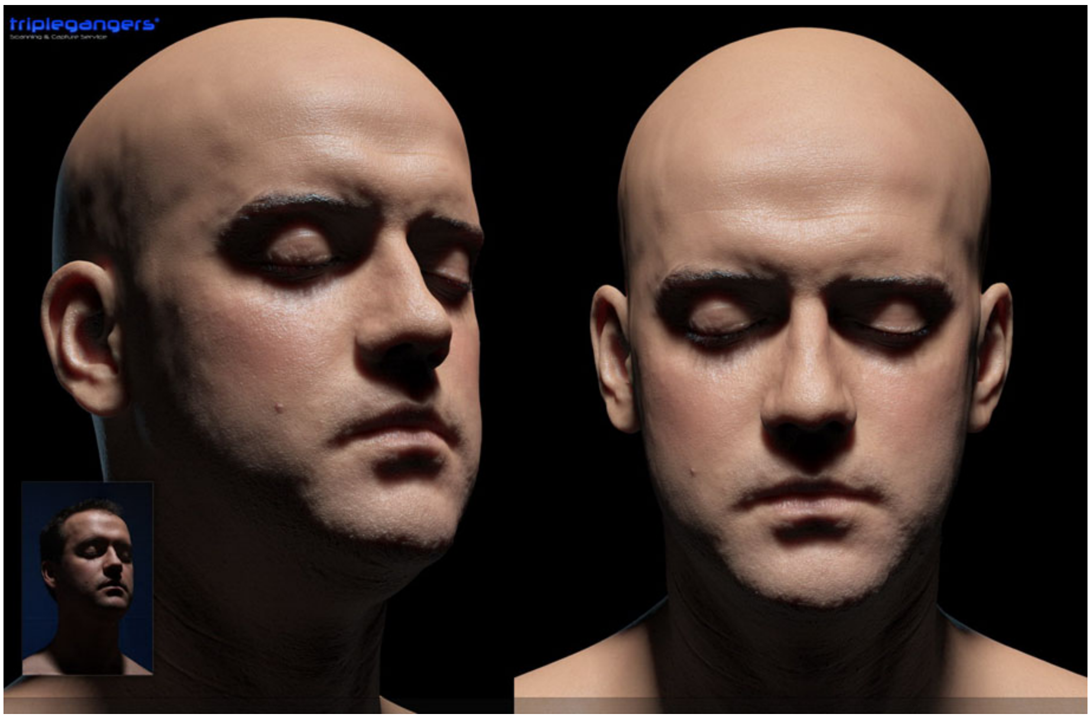

Figure 1: If your rendering looks this good, you'll get lots of points.

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.

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 "Materials" in MyCourses) 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.

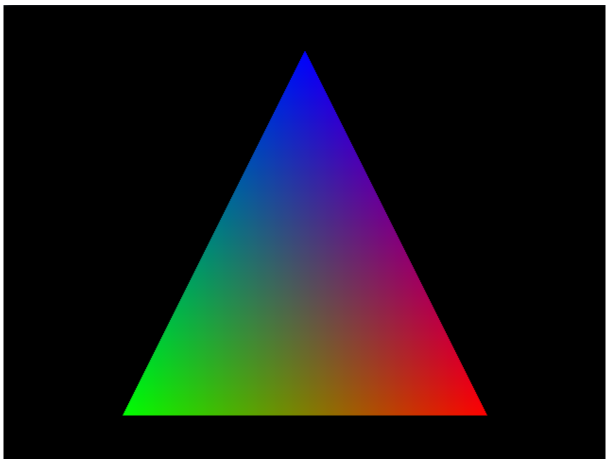

Figure 2: Three colors (red, green and blue) defined, others interpolated. This happens when vertex shader outputs its color to a varying, and pixel shader outputs its color directly from an input varying of the same name.

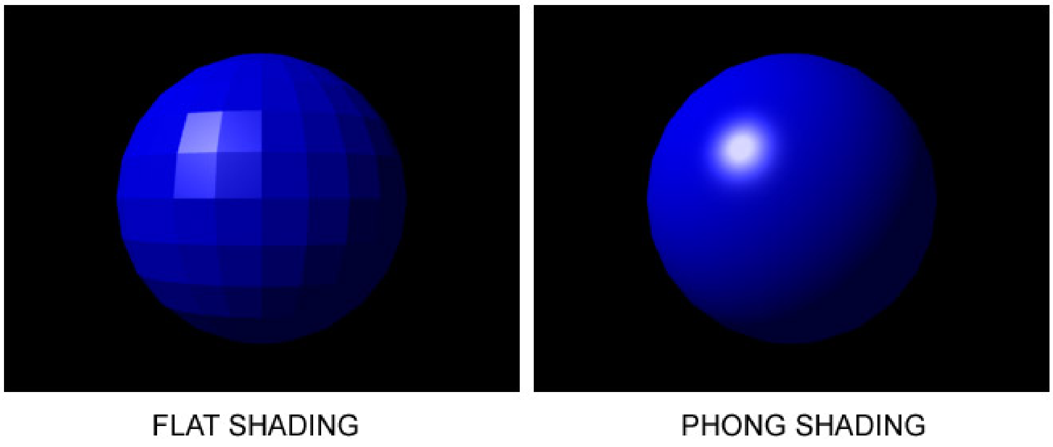

Figure 3: Flat vs. phong shading: same mesh, different normal computation. Phong interpolates the normal linearly between vertices; in flat shading, the normal stays the same throughout the whole face.

3 Detailed instructions

R1 Sample diffuse and normal textures (1p)

Diffuse texture

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 Learning Modern 3D Graphics Programming online-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, `texture()`. Pay attention to the vector sizes - colors have four components, where the last one is the (often unnecessary) alpha channel.

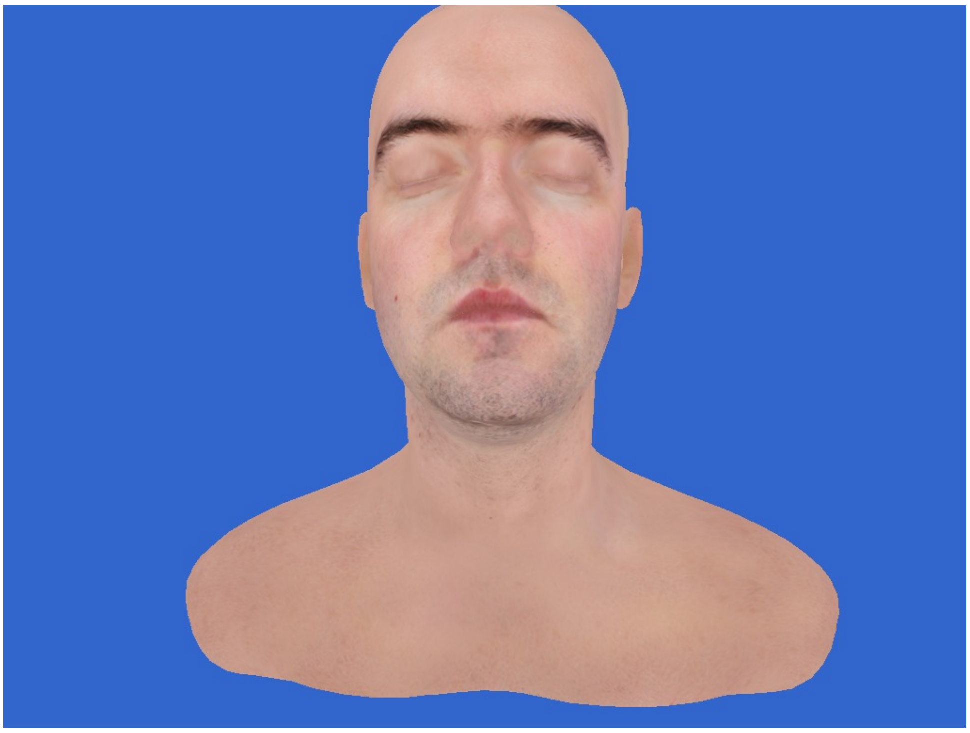

Figure 4: Plain diffuse colors directly from the texture

Normal texture

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.

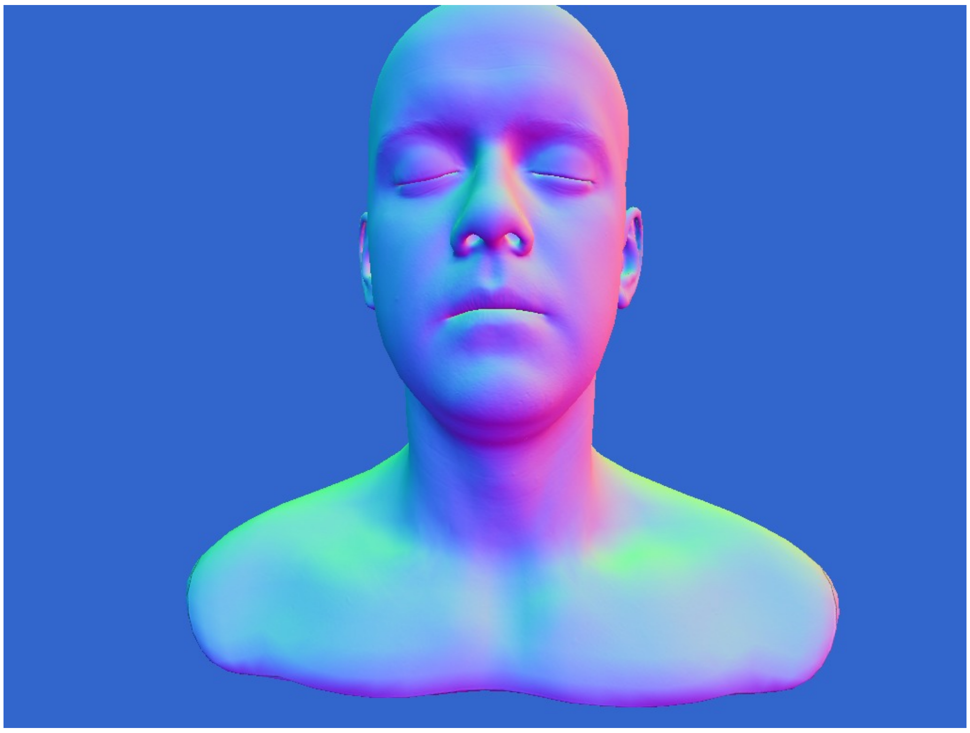

Figure 5: The head colored by the normal vectors

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.

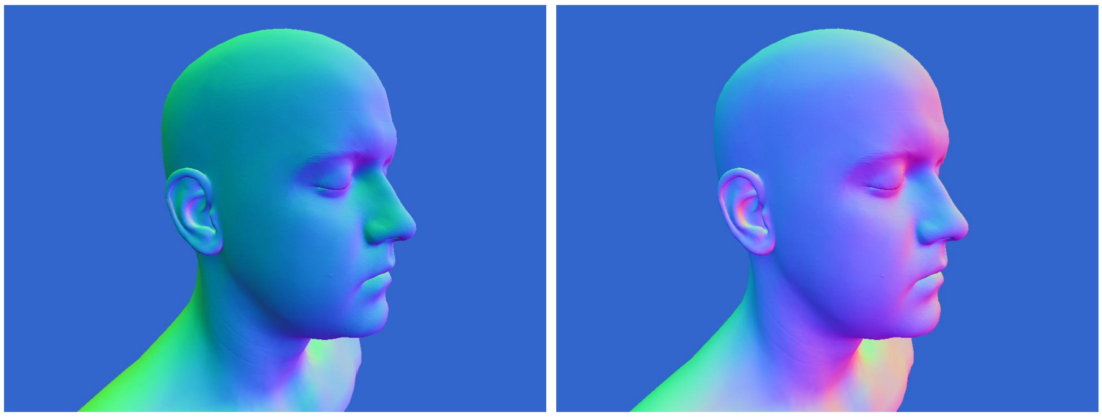

Figure 6: Head normals from the side. Left: object space, right: camera space. Note how camera space's Z axis points always to the eye, while on the other hand the head's face points to object space's Z. The image on the left has exactly the same colors as the one in 5, if you look closely.

R2 Lambertian diffuse shading (1p)

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, $\mathbf{l_i}$ 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:

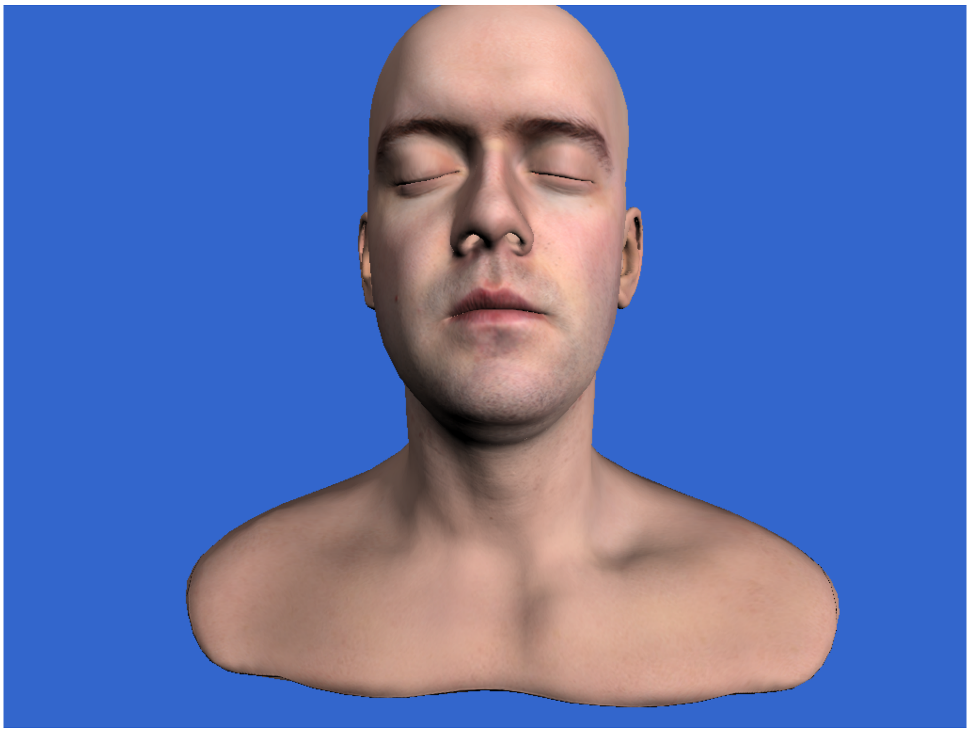

Figure 7: Diffuse head taking into account the light positions

3.1 Physically based shading: Cook-Torrance microfacet model

Whereas Assignment 5 called for Phong specular shading, this time we'll use the physically based Cook-Torrance microfacet model for specular reflections. We extend the previously presented diffuse only model to also include a specular component. Our total shading model becomes

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

where k_s is the specular albedo that behaves similarly to the diffuse one, and S_i is the specular reflection as defined by the Cook-Torrance microfacet model

$$S_i = \frac{F_r\, D\, G}{4\,|\mathbf{n} \cdot \mathbf{l}_i|\,|\mathbf{n} \cdot \mathbf{v}|}.$$

There are many possible choices for the distribution functions. We will use a model known as **GGX** for the terms D and G, while F_r is given by the Fresnel equations. Each of these functions will be explained in further detail in their respective sections.

The starter code contains a function that evaluates the Cook-Torrance model and stubs for all of the separate terms.

R3 GGX distribution function D (1p)

$$\begin{aligned} D(\mathbf{n}, \mathbf{h}) &= \frac{\alpha^2}{\pi\, cos^4\theta_h\, (\alpha^2 + tan^2\theta_h)^2} && \text{when}\, \mathbf{n} \cdot \mathbf{h} > 0 \\ D(\mathbf{n}, \mathbf{h}) &= 0 && \text{when}\, \mathbf{n} \cdot \mathbf{h} \leq 0 \end{aligned}$$

Here α is the surface `roughness`, θ_h is the angle between the surface normal $\mathbf{n}$ and the half-angle vector $\mathbf{h}$. The half-angle vector $\mathbf{h}$ is the vector that bisects the vector pointing towards the light and the vector pointing towards the camera

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{||\mathbf{l} + \mathbf{v}||}$$

Vector $\mathbf{v}$ you can figure out from the camera position and the position of the point you are currently shading. It will also help to take a moment to think in which space our lighting calculations take place in. Now that you have the vectors $\mathbf{n}$, $\mathbf{h}$, $\mathbf{v}$ and $\mathbf{l}$, you can call the CookTorrance function and go on to implement the D function as the formula describes.

Note that you can compute the cosine of the angle between two unit vectors using the dot product, i.e. $cos\,\theta_h = \mathbf{n} \cdot \mathbf{h}$. You can also utilize the trigonometric identity

$$tan\,\theta = \frac{sin\,\theta}{cos\,\theta} = \frac{\sqrt{1 - cos^2\theta}}{cos\,\theta}.$$

After you have implemented the D function, by pressing the button `Debug render: distribution term D` , you should see the following result:

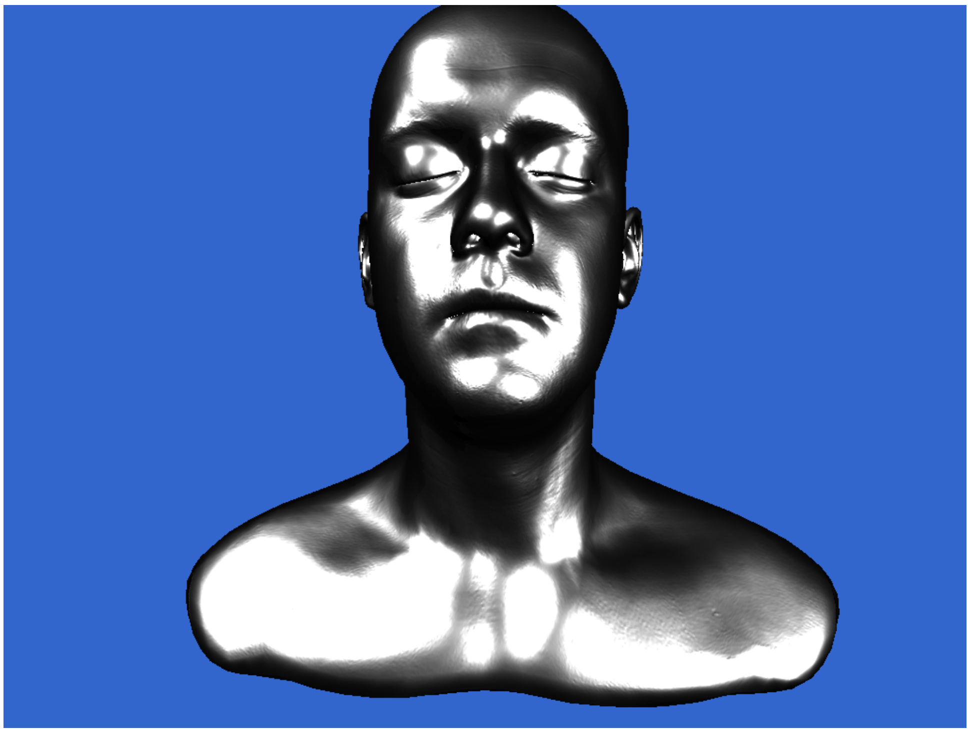

Figure 8: Distribution function D visualized

R4 GGX masking-shadowing function G (1p)

The masking-shadowing function G describes the portion of reflected light that is shadowed by other microfacets. Shadowing occurs on both incident and out going light, so G is comprised of two Smith masking functions G_1.

$$G(\mathbf{v}, \mathbf{l}, \mathbf{h}) = G_1(\mathbf{v}, \mathbf{h}) \cdot G_1(\mathbf{l}, \mathbf{h})$$

and

$$G_1(\mathbf{x}, \mathbf{h}) = \frac{2}{1 + \sqrt{1 + \alpha^2 \, tan^2 \theta_x}}$$

where α is the surface `roughness`, and θ_x is the angle between the vectors $\mathbf{x}$ and $\mathbf{h}$.

Implement the G function as the formulas describe. All the required parameters should be readily available to you at this point.

By pressing the button `Debug render: geometry term G` , you should now see the following result:

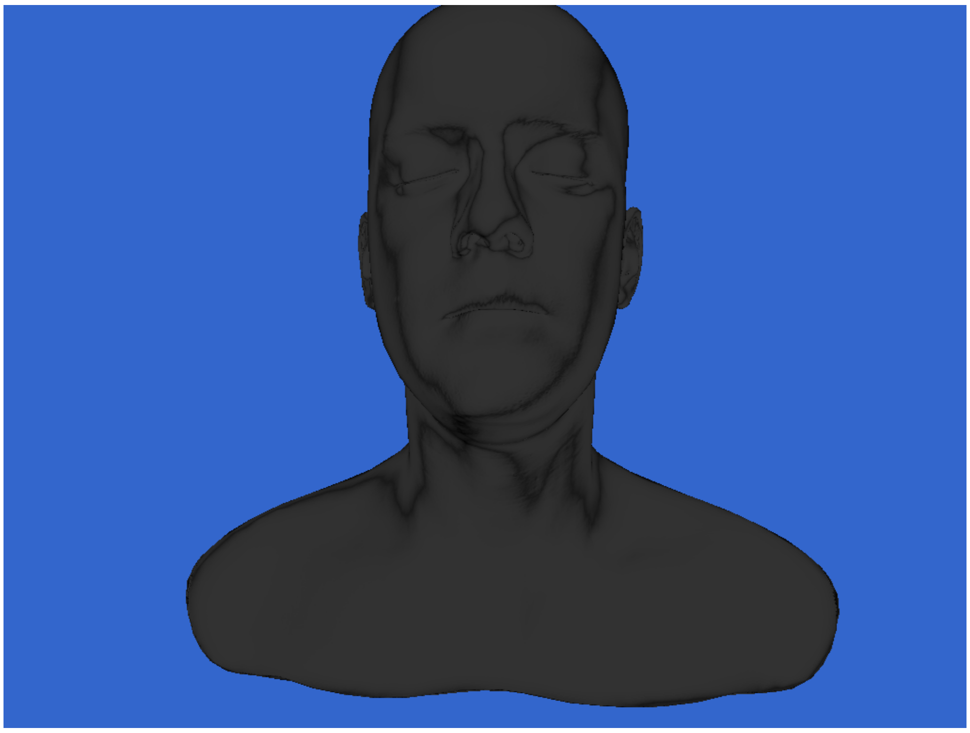

Figure 9: Masking-shadowing function function G visualized

R5 Fresnel reflectivity coefficient F_r (1p)

The Fresnel reflectivity coefficient is a function of the angle between our microsurface normal (i.e. the half-vector) and an incident light vector. F_r has a value between $[0, 1]$ which describes the ratio of incident light that gets specularly reflected for a given incident angle. Due to conservation of energy, the remaining portion of light gets transmitted through the interface, and is given by $F_t = 1 - F_r$. Our material model assumes that all transmitted light is absorbed inside the material.

Note that the following approach only applies to interfaces between dielectric (i.e. non-conducting) materials. Interfaces with conducting materials such as metals would require accounting for complex valued indices of refraction. However, our model has only dielectric materials so we don't need to worry about it.

Fresnel reflectivities for s- and p-polarized light are

$$R_s(\mathbf{l}, \mathbf{h}) = \left(\frac{n_1 cos\,\theta_l - n_2\sqrt{1 - (\frac{n_1}{n_2} sin\,\theta_l)^2}}{n_1 cos\,\theta_l + n_2\sqrt{1 - (\frac{n_1}{n_2} sin\,\theta_l)^2}} \right)^2$$

$$R_p(\mathbf{l}, \mathbf{h}) = \left(\frac{n_1\sqrt{1 - (\frac{n_1}{n_2} sin\,\theta_l)^2} - n_2 cos\,\theta_l}{n_1\sqrt{1 - (\frac{n_1}{n_2} sin\,\theta_l)^2} + n_2 cos\,\theta_l} \right)^2$$

where n_1 and n_2 are the indices of refraction of our two mediums and θ_l angle between the incident light vector $\mathbf{l}$ and the microsurface normal $\mathbf{h}$.

Natural light can be described as unpolarized, the effective reflectivity is then the average of the two reflectivities

$$F_r(\mathbf{l},\mathbf{h}) = \frac{1}{2}(R_s(\mathbf{l},\mathbf{h}) + R_p(\mathbf{l},\mathbf{h}))$$

We can simplify to express these only in terms of $cos\,\theta_l$ and $\eta = \frac{n_1}{n_2}$

$$R_s(\mathbf{l},\mathbf{h}) = \left(\frac{cos\,\theta_l - \sqrt{1/\eta^2 + cos^2\theta_l - 1}}{cos\,\theta_l + \sqrt{1/\eta^2 + cos^2\theta_l - 1}}\right)^2$$

$$R_p(\mathbf{l},\mathbf{h}) = \left(\frac{\eta^2\sqrt{1/\eta^2 + cos^2\theta_l - 1} - cos\,\theta_l}{\eta^2\sqrt{1/\eta^2 + cos^2\theta_l - 1} + cos\,\theta_l}\right)^2$$

Notice how the term $\sqrt{1/\eta^2 + cos^2\theta_l - 1}$ repeats. Let's define

$$\beta = \sqrt{1/\eta^2 + cos^2\theta_l - 1}$$

If β is imaginary, i.e. we have a case of total internal reflection, we set $F_r = 1$. In our scene this shouldn't happen though, because we always have $n_1(air) < n_2(model\,surface)$.

Now we have the final formulae

$$R_s(\mathbf{l},\mathbf{h}) = \left(\frac{cos\,\theta_l - \beta}{cos\,\theta_l + \beta}\right)^2$$
$$R_p(\mathbf{l},\mathbf{h}) = \left(\frac{\eta^2\beta - cos\,\theta_l}{\eta^2\beta + cos\,\theta_l}\right)^2$$
$$F_r(\mathbf{l},\mathbf{h}) = \frac{1}{2}(R_s(\mathbf{l},\mathbf{h}) + R_p(\mathbf{l},\mathbf{h}))$$

where naturally, $cos\,\theta_l = \mathbf{l} \cdot \mathbf{h}$

Again, all the parametes are already there so all you need to do is to translate these formulas to code.

By pressing the button `Debug render: Fresnel term F` , you should see the following result. Note that for clarity the term is set to 0 whenever the light is not visible from the point.

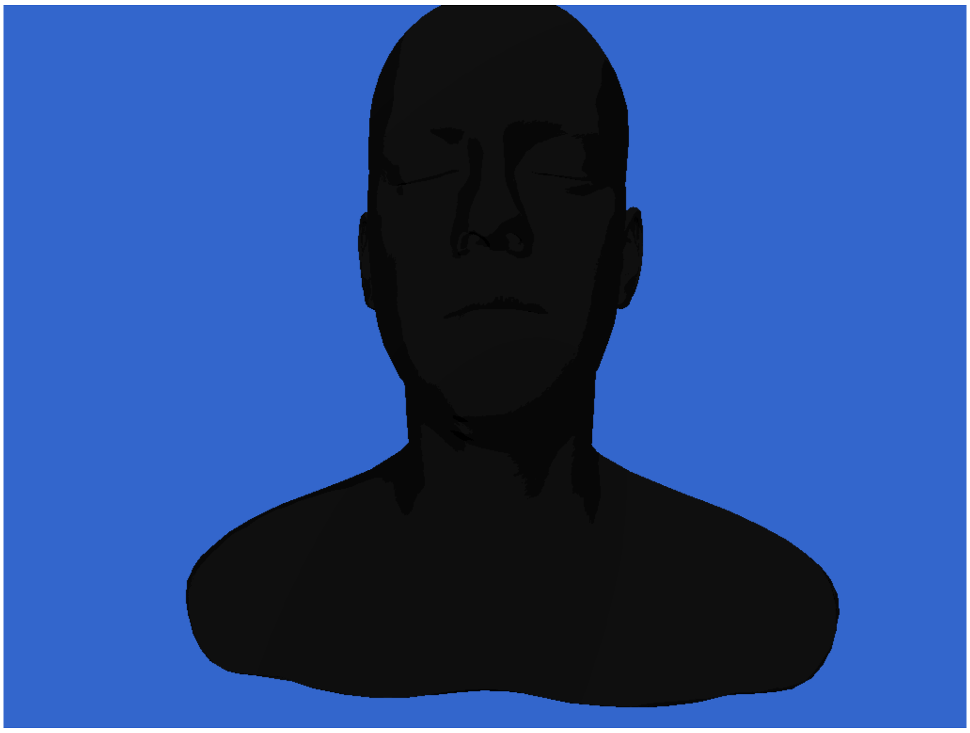

Figure 10: Fresnel reflectivity coefficient F_r visualized

After all the requirements are done, pressing the `Final render` toggle (on by default) should give the following result:

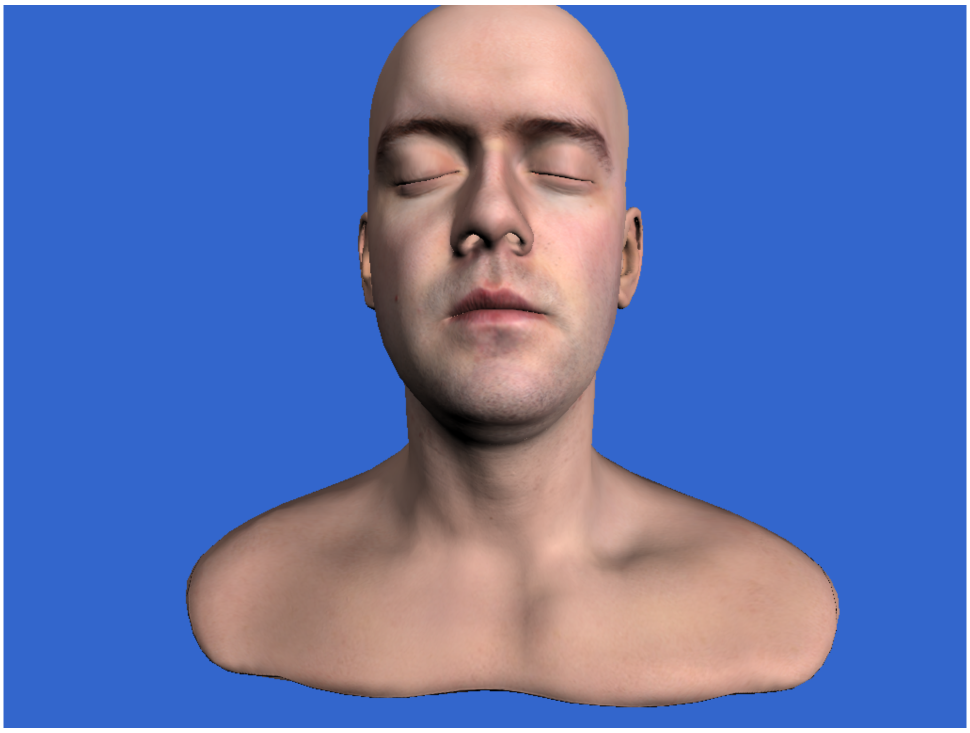

Figure 11: The final result with all the terms enabled

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.

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.

4.1 Recommended

- **World space lighting (2-3p)**

  Supply working code for R2-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++.

- **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.

- **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.

- **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://developer.download.nvidia.com/books/HTML/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 Medium

- **Shadow maps, self-shadowing (4 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.

- **Subsurface scattering using depth maps(2p)**
  Approximate subsurface scattering by using a shadow map to estimate the distance travelled by light inside the object. This is the second method described in the following article:

  `http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch16.html`

- **Environment mapping (3+ p)**

  Implement a simple perfect specular reflection using an environment map as the incoming light. More points are awarded for more advanced effects such as refraction, Fresnel or a more interesting material model.

  Nvidia tutorial

- **Tangent space normal mapping (3p)**

  Extend your code to support tangent space normal mapping. This means that the normal map is in the local tangent space of each separate triangle instead of object space, which is far more convenient if your object gets deformed. First, you'll need a scene that comes with tangent space normal maps. You can use Crytek Sponza, available e.g. with CS-E5520 assignment 1 (look for a previous year) or look for another scene online. FW already loads the normal maps for you, so you'll just need to add a toggle button to enable the tangent space version of normal maps instead of the object space ones used by our head object. For more in-depth instructions, you can have a look at this OpenGL tutorial.

- **Terrain rendering on GPU Part II(?p).** Check the blog post about Fast Terrain Rendering with Continuous Detail on a Modern GPU by Morgan McGuire. This time go all out on the materials, lighting and the finishing touches. You may also start by completing the first part of the extra listed in Assignment 2. Points will be awarded for each feature completed.

4.4 Advanced material models

- **Area light shading with linearly transformed cosines (4+p)**

  Implement area light shading using a simple approximation that allows you analytically integrate over the surface of a polygonal area light. Links: paper and youtube presentation

4.5 Deferred shading

- **Deferred shading (4p)** For more complex scenes, deferred shading gives the benefit of eliminating unnecessary pixel shader invocations due to surfaces overlapping each other in the depth direction. In naïve (forward) rendering, each visible pixel of every triangle is shaded after the vertex shader for that triangle is completed. This often leads to later triangles overwriting previous pixel shader results, as a closer triangle is rasterized over the previous one. Deferred shading postpones the shading pass until after all triangles are rasterized, which means every pixel only needs to be shaded once. This can give significant performance benefits if scene geometry is complex enough and the pixel shader is expensive. Here's a link to an OpenGL tutorial which describes the technique in more detail.

- **Tiled deferred shading (6p)** Standard deferred shading has the unfortunate effect of requiring a G-buffer read for each light that overlaps a pixel, causing memory bandwidth bottlenecks with a high number (1000) of lights. Tiled deferred shading solves this problem by precomputing the list of lights affecting each tile, and then computing light contributions for each pixel in the tile. This has the benefit of requiring only a single G-buffer read per pixel, but requires the use of an additional precompute pass for light contribution per tile. See this article for more detailed instructions.

- **Forward+ shading (5p)** Forward+ or forward tiled shading is an improvement to tiled deferred rendering, getting rid of the memory-consuming G-buffer as well as making anti-aliasing and transparent objects easier to handle. In tiled forward rendering, the G-buffer is replaced by a simple depth prepass that eliminates unnecessary work due to overlapping surfaces, but consumes no additional memory. In addition, lights are culled using the same tiled scheme as in tiled deferred shading. See here for a more detailed description.

4.6 Texture-Space Diffusion

- **Subsurface scattering by Texture-Space Diffusion(5p)**
  Use Texture-Space Diffusion, the third method in the article below, to estimate subsurface scattering. You'll want to render the illumination into a separate texture — this is straightforward since our lights are directional and the normal map is already in object space. Blur the results and use them for rendering.

  http://http.developer.nvidia.com/books/HTML/gpugems/gpugems_ch16.html

- **Ideas for further extras**
  Real-time rendering is an active research topic. A variety of interesting effects can be approximated fast enough using consumer hardware. Here are some interesting articles to give you ideas for implementing neat effects.

  http://developer.download.nvidia.com/shaderlibrary/docs/shadow_PCSS.pdf
  https://www.slideshare.net/TiagoAlexSousa/graphics-gems-from-cryengine-3-siggrap
  http://jcgt.org/published/0003/04/04/paper.pdf
  https://de45xmedrsdbp.cloudfront.net/Resources/files/The_Technology_Behind_
  the_Elemental_Demo_16x9-1248544805.pdf

  Note that you can also submit any earlier extras from any round.

5 Submission

- Make sure your code compiles and runs both in Release and Debug modes on Visual Studio 2019, preferably in the VDI environment. Comment out any functionality that is so buggy it would prevent us seeing the good parts. Check that your `README.txt` (which you hopefully have been updating throughout your work) 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 the code, project and solution files required to build your submission, the `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive.

- Sanity check: 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.)

- If you're submitting extras from another assignment, specify in your README which ones you did and which round they were originally in. Also if you're resubmitting an improved solution, specify how the previous submission was graded.

Submit your archive in MyCourses folder:
"Assignment 6: Real-time Shading".