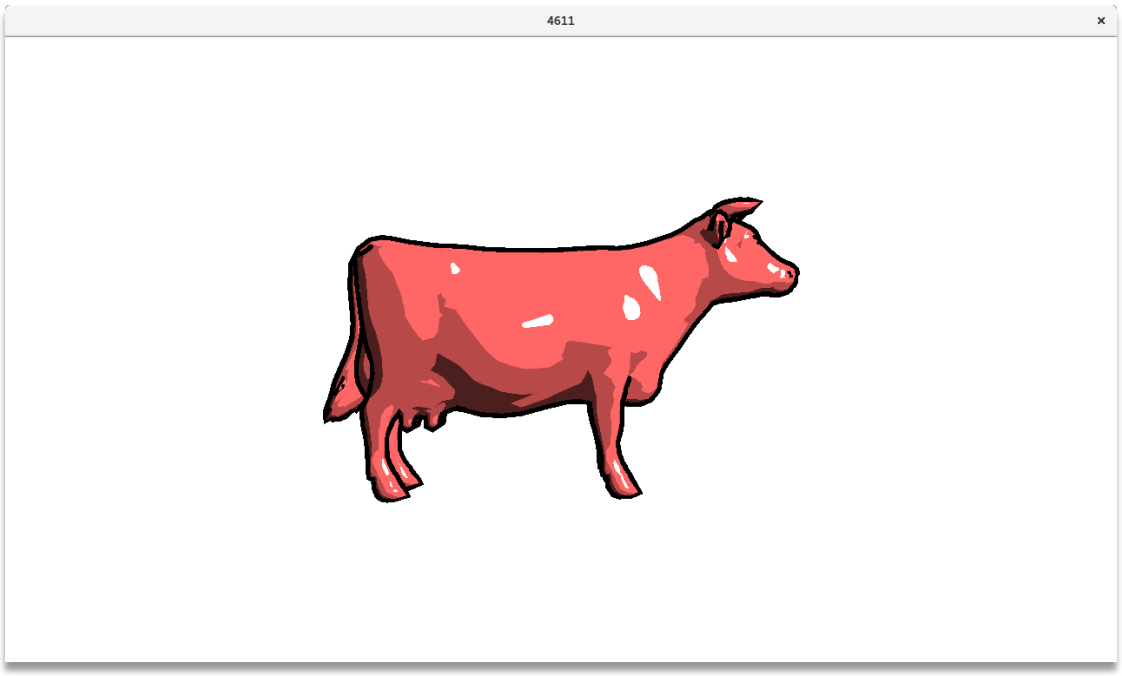
CSCI 4611: Programming Interactive Graphics and Games

Assignment 5: Artistic Rendering

Handed out: Wednesday, April 5 (updated Tuesday, April 11)

Due: Wednesday, April 19

Introduction



GLSL shaders make it possible for us to create some amazing lighting effects in real-time computer graphics. These range from photorealistic lighting to artistically inspired non-photorealistic rendering, as featured in games like *The Legend of Zelda: The Wind Waker* and *Team Fortress 2*. In this assignment, you will implement a GLSL shader that can produce both realistic per-pixel lighting, "toon shading" as shown above, and a variety of other effects. You will also implement another shader that adds silhouette edges (the black outlines seen above) to complete the cartoon effect.

In this assignment, you will learn:

- how to calculate realistic and artistic per-pixel lighting in real time,
- how to modify geometry on the fly to create viewpoint-dependent effects such as silhouette edges, and
- how to implement and use your own shader programs.

Requirements

There are three requirements for this assignment:

1. Per-pixel Phong shading using the standard (Blinn-)Phong lighting model.
2. Flexible shading using texture images.
3. Drawing outlines using silhouette edge.

These correspond to “C”, “B”, and “A” level work respectively. You will have to implement one shader program (i.e. a pair of vertex and fragment shaders) to satisfy the first two requirements, and a second shader program to satisfy the third.

Per-pixel Phong shading

During the next few class sessions we’ll work on some shader programs that calculate ambient, diffuse, and specular lighting using per-vertex (Gouraud) shading. Your job is to extend the concepts and programs we develop in class to implement per-pixel Phong shading with the same lighting model. You should be able to build this by extending the shaders that we discuss and develop in class.

Implement a shader program that performs all the calculations to accurately calculate the Blinn-Phong lighting model for each pixel. The lighting terms must vary per-pixel based on the normal and the light position, as well as the various material properties (such as the specular exponent). You should implement this shader following the lighting model equations discussed in class. For the specular component, you may use either the half-vector or the reflection vector method, but document your choice in the `README` file.

None of the relevant shader input parameters (light color, position, material reflection coefficients, etc.) should be hardcoded as constants in your shaders. Although we will probably hardcode some of these values in the examples we do in class, you should take the time to do this the right way in your assignment. Pass these parameters from your C++ program to your shader program using the `program.setUniform()` method.

Flexible shading using texture images

Once you have Phong shading working (including ambient, diffuse, and specular lighting), adapt your shader as follows. Rather than setting the final color based on the intensity of light you calculate for the Blinn-Phong model, you instead use this intensity value as a lookup into a texture and use that to compute the final color. A texture used in this way is typically called a “ramp”. With this strategy, you’ll be able to get a wide range of different lighting effects just by switching the texture you use for input.

Suppose we use the dot product in the diffuse term, $\mathbf{n} \cdot \boldsymbol{\ell}$, to look up the texture, so that if its value is −1, we pick the color from the leftmost pixel in the texture; if it is 1, we pick the color

from the rightmost pixel; and similarly in between. If we use `standardDiffuse.png` (see below), which is zero in the left half corresponding to negative $\mathbf{n} \cdot \boldsymbol{\ell}$, and increases linearly from 0 to 1 for positive $\mathbf{n} \cdot \boldsymbol{\ell}$, then we'll get back the standard diffuse lighting term.

`standardDiffuse.png`

But, if we use `toonDiffuse.png`, we'll get something that looks like a cartoon, as if an artist were shading using just 3 colors of paint.

`toonDiffuse.png`

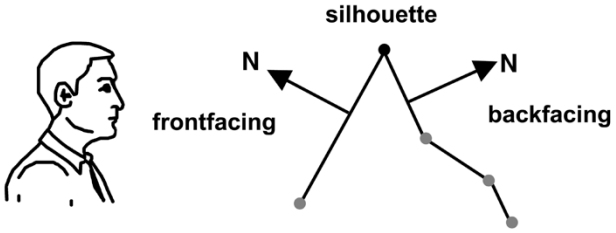
Note that this is the same type of lighting effect you see in many games, including *The Legend of Zelda: The Wind Waker* and *Team Fortress 2* (below). *Wind Waker* uses a very simplified light model. In this example, it looks like there are just 2 values used in the shading: each surface is either in bright light or dark. *Team Fortress 2* is a bit more subtle: it reduces the brightness variation in lit areas without completely flattening them out. You can read more about this in Mitchell et al.'s article "Illustrative Rendering in *Team Fortress 2*", linked in the "Further Reading" section.

Inside your Phong shading program, you will have equations that calculate the intensity of reflected light for ambient, diffuse, and specular. For the diffuse portion, the key quantity will be $\mathbf{n} \cdot \boldsymbol{\ell}$, which should range from −1 to 1. This is the value that you want to use to lookup the lighting color to apply from the texture ramp. If the value of $\mathbf{n} \cdot \boldsymbol{\ell}$ is −1, then you want to use the color on the leftmost side of the texture, and if it is 1, then you want to use the color on the rightmost side of the texture. That means your texture coordinates for this lookup will be $(0.5(\mathbf{n} \cdot \boldsymbol{\ell}) + 0.5, 0)$, because texture coordinates only go from 0 to 1. (You could actually use any value you want for the *y* coordinate, since the color only varies from left to right.) After

calculating these texture coordinates, you can get the color from the texture image using the GLSL built-in function `texture()` as discussed in class.

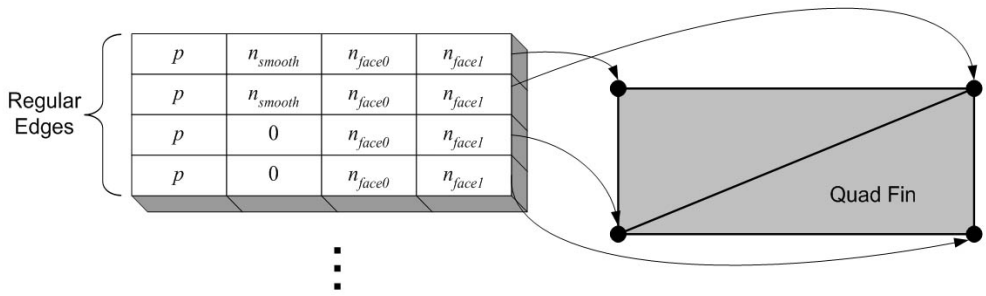
You need to add two separate texture lookups to your shader, one for the diffuse component and one for the specular component. For the specular component, we need to clamp $\mathbf{h} \cdot \mathbf{n}$ to positive values anyway before taking the exponent, so you should directly use the intensity $\max(\mathbf{h} \cdot \mathbf{n}, 0)^s$ as the texture coordinate without rescaling. Note that adding these texture lookups to your code will change the Blinn-Phong shader that you made in requirement 1. You do *not* need to include an explicit option to draw in the old Blinn-Phong lighting mode, because using the provided `standardDiffuse.png` and `standardSpecular.png` textures should reproduce the original appearance.

Silhouette edges



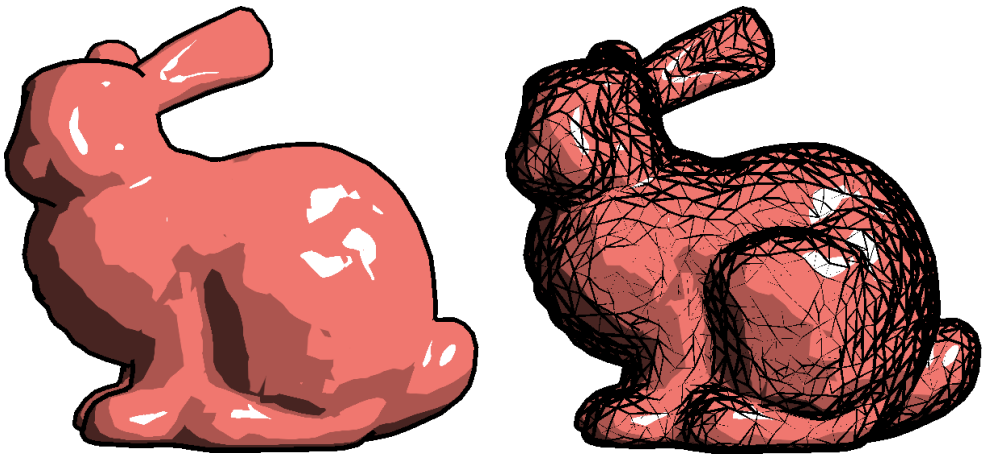
There are lots of different ways to draw silhouette contours on 3D shapes. We will use a simple method described by Card and Mitchell, linked under "Further Reading". For each edge of the triangle mesh, we check whether it lies on the silhouette, that is, on the boundary between the triangles facing towards the camera and the triangles facing away from it. If so, it is a silhouette edge, and we will draw it as a thick black line segment to create the outline of the shape.

Drawing a thick line segment takes a little bit of work in OpenGL 3 and above, since the function `glLineWidth()` is no longer officially supported. Instead, we will have to draw the line segment as a quadrilateral whose width is the desired thickness. Since we don't know in advance which edges will be silhouette edges and which will not, we will create a zero-width quadrilateral for *every* edge. In the vertex shader, we will check whether the vertex is part of a silhouette edge, and if so, displace it by the desired thickness. Thus, silhouette edges will be drawn as thick quadrilaterals, while all other edges will be drawn as quadrilaterals of zero width, which can't be seen.



[Card and Mitchell 2002]

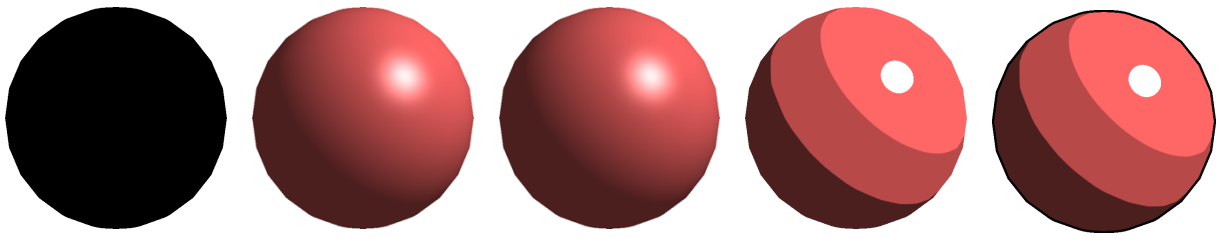
The starter code provides a class `EdgeMesh` that stores the information needed to draw these silhouette edges. It creates a quadrilateral (4 vertices and 2 triangles) for every edge in the original mesh, as shown above. Each vertex stores its position, its displacement direction `direction` (labeled $\mathbf{n}_{smooth}$ in the diagram above), and the normals of the adjacent faces `leftNormal` and `rightNormal`. Implement a shader program that checks each vertex for whether it lies on a silhouette edge, that is, whether $\mathbf{n}_{left} \cdot \mathbf{v}$ and $\mathbf{n}_{right} \cdot \mathbf{v}$ have different signs. If so, displace the vertex by `thickness*direction` when computing the output `gl_Position`. The thickness should be passed into the shader from your main program as a uniform variable.



Left: `bunnyLowres.obj` with only vertices on silhouette edges displaced.
Right: with vertices on all edges displaced.

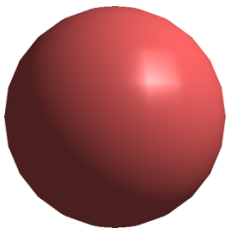
Reference Images

I've added a new mesh called `sphere.obj` to the data zip file. It's just a sphere with 24 slices and 12 stacks. The results of your program on this mesh should look like the following as you progress through the assignment:



From left to right: the initial view before you implement anything; Phong shading with the standard Blinn-Phong model; ramp shading with `standardDiffuse.bmp` and `standardSpecular.bmp` (this is identical to the Blinn-Phong model); ramp shading with `toonDiffuse.bmp` and `toonSpecular.bmp`; and the same with silhouette edges drawn.

If you fail to normalize the fragment normal when doing Phong shading, you will get an *incorrect* result that looks like the image on the right. Note that even if your vertex normals are normalized, the rasterizer will interpolate them to fragments by averaging, and the average of two unit vectors may not itself be a unit vector!



Above and Beyond

All the assignments in the course will include great opportunities for students to go beyond the requirements of the assignment and do cool extra work. We don't offer any extra credit for this work — if you're going beyond the assignment, then chances are you are already kicking butt in the class. However, we do offer a chance to show off… While grading the assignments the TAs will identify the best 4 or 5 examples of people doing cool stuff with computer graphics. After each assignment, the selected students will get a chance to demonstrate their programs to the class!

Once you get the hang of them, shaders can be really fun! Try out some different textures and lighting effects. One interesting possible extension of our 1D ramp textures could be to use a single 2D texture that you look up using both the diffuse and specular intensities. Other things you can do with shaders is to add stripes, waves, random noise, or bumps to the surface.

If your ideas for going beyond the requirements would make your code more difficult for the TAs to grade, please help them out by submitting a standard version of your assignment first through the normal website link, and then email the TAs the fancier version of your assignment.

Support Code

The webpage where you downloaded this assignment description also has a download link for support code to help you get started. The support code for this assignment is a simple program using the SDL-based engine, similar to the ones we have used before. You should also download separately the zip file containing the meshes and texture images.

The support code defines a program structure and everything you need to read the mesh data. **To make locating data files simpler, we have a header file called `config.hpp` that contains absolute paths to your data files. You should edit this file with the full path (e.g. `"C:\Users\Turing\a5\data"` or `"/home/turing/a5/data"`) where you've placed the data files.** We will modify this file appropriately when grading your assignment.

Handing It In

When you submit your assignment, you should include a `README` file. This file should contain, at a minimum, your name and descriptions of design decisions you made while working on this project. If you attempted any "above and beyond" work, you should note that in this file and explain what you attempted.

When you have all your materials together, zip up the source files and the `README`, and upload the zip file to the assignment hand-in link on our Moodle site. Any late work should be handed in the same way, and points will be docked as described in our syllabus.

Further Reading

Note: You don't need to read these articles to implement the assignment. They're only provided in case you're curious and want to learn more about non-photorealistic rendering.

Mitchell, Francke, and Eng, "Illustrative Rendering in Team Fortress 2", *Non-Photorealistic and Artistic Rendering* 2007.
http://www.valvesoftware.com/publications/2007/NPAR07_IllustrativeRenderingInTeamFortress2.pdf

Gooch, Gooch, Shirley, and Cohen, "A Non-Photorealistic Lighting Model for Automatic Technical Illustration", *SIGGRAPH* 1998.
http://www.cs.northwestern.edu/~ago820/SIG98/abstract.html

Card and Mitchell, "Non-Photorealistic Rendering with Pixel and Vertex Shaders", in *ShaderX: Vertex and Pixel Shaders Tips and Tricks*, 2002.
http://developer.amd.com/wordpress/media/2012/10/ShaderX_NPR.pdf

Gooch, Hartner, and Beddes, "Silhouette Algorithms".
https://www.cs.rutgers.edu/~decarlo/readings/gooch-sg03c.pdf