Graphics Programming II – Assignment #2
Due on 3/7/2012

In this assignment, you will implement per-fragment lighting, normal mapping, and the Cook-Torrance BRDF. This should be implemented in several steps, listed below.

- The code previously provided generates per-vertex tangent and texture coordinates. Modify the C++ code to pass this data as attributes to the vertex shader. Modify the vertex shader to pass these values directly to the fragment shader. Write a test fragment shader the writes one of the values as the final color. Try both values. This lets you be sure that the correct data is getting into the vertex shader.
- Modify the vertex shader to emit the light position and the eye position in tangent space. To verify the credibility of the results, emit each as the per-vertex color. What sorts of colors should you expect to see?
- Modify the fragment shader to perform lighting calculations in tangent space. Initially implement just simple $n\dot{l}$ diffuse lighting. Implement this in a function called `LambertBRDF` that returns a constant 1.0. Use the value returned by this function in the $f(\omega_i, \omega_o)L(\omega_i)cos\theta_i$ lighting equation.
- Add Blinn specular lighting (as was previously performed in the vertex shader). Implement this a function called `BlinnBRDF`. Use the value returned by this function in the $f(\omega_i, \omega_o)L(\omega_i)cos\theta_i$ lighting equation.
- Calculate the Fresnel factor using Schlick's approximation. Initially emit just the Fresnel value calculated at each fragment.
- Implement the remaining components of the Cook-Torrance BRDF in a function called CookTorrance-BRDF.
- Load a normal map of your choosing. Modify the vertex shader to pass texture coordinates to the fragment shader. Modify the fragment shader to sample from the texture. Initially, just emit the values from the texture as the color. Once that works, use the sampled value as the normal for the shading operation.

Criteria	Excellent	Good	Satisfactory	Unacceptable
Completion	Program correctly implements all required elements in a manner that is readily apparent when the program is executed. User interface is complete and responsive to input. Program documents user interface functionality.	Program implements all required elements, but some elements may not function correctly. User interface is complete and responsive to input.	Program implements most required elements. Some of the implemented elements may not function correctly. User interface is complete and responsive to input.	Many required elements are missing. User interface is incomplete or is not responsive to input.
Correctness	Program executes without errors. Program handles all special cases. Program contains error checking code.	Program executes without errors. Program handles most special cases.	Program executes without errors. Program handles some special cases.	Program does not execute due to errors. Little or no error checking code included.
Efficiency	Program uses solution that is easy to understand and maintain. Programmer has analysed many alternate solutions and has chosen the most efficient. Programmer has included the reasons for the solution chosen.	Program uses an efficient and easy to follow solution (i.e., no confusing tricks). Programmer has considered alternate solution and has chosen the most efficient.	Program uses a logical solution that is easy to follow, but it is not the most efficient. Programmer has considered alternate solutions.	Program uses a difficult and inefficient solution. Programmer has not considered alternate solutions.
Presentation & Organization	Program code is formatted in a consistent manner. Variables, functions, and data structures are named in a logical, consistent manner. Use of white space improves code readability.	Program code is formatted in mostly consistent with occasional inconsistencies. Variables, functions, and data structures are named in a logical, mostly consistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted with multiple styles. Variables, functions, and data structures are named in a logical but inconsistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted in an inconsistent manner. Variables, functions, and data structures are poorly named. Use of white space hurts code reability.
Documentation	Code clearly and effectively documented including descriptions of all global variables and all non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of most global variables and most non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of the most important global variables and the most important local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted.	No useful documentation exists.

This rubric is based loosely on the "Rubric for the Assessment of Computer Programming" used by Queens University (http://educ.queensu.ca/ compsci/assessment/Bauman.html).