CG Programming II – Assignment #2 (Tangent-space Lighting)

In this assignment you will explore several aspects of lighting and texture mapping. The ultimate goal is to render an object using the Blinn-Phong lighting equation and a normal map.

Part 1: Due on 6-February-2013 at the end of class

- Download the base code and get it to compile.
- Modify the provided vertex shader, `simple.vert`, to pass the camera-space position and camera-space normal to the fragment shader.
- Modify the provided fragment shader, `simple.frag`, to use the values provided in the previous step to calculate specular and diffuse lighting.
- Use the provided utility routines to load an image of your choosing, and store it in a `GL_TEXTURE_2D` texture.
- Modify the vertex shader to pass the `uv` input to the fragment shader. This will be used as the texture coordinate.
- Modify the fragment shader and associated C++ drawing code to apply the texture to the object.

Part 2: Due on 13-February-2013 at the start of class

- Modify the vertex shader to calculate the camera-space tangent vector.
- Use the camera-space tangent vector and the camera-space normal vector to create a transformation matrix ("TBN") to transform the light vector and the camera vector (vector from the vertex to the camera) to tangent-space (a.k.a. surface-space). Pass these vectors to the fragment shader.
- Modify the fragment shader to perform lighting calculations in tangent-space instead of camera-space.
- Use the provided utility routines to load another image of your choosing, and it store in another `GL_TEXTURE_2D` texture. This image will be your normal map.
- Modify the fragment shader and associated C++ drawing code to sample the normal map. Use the sampled value as the surface-space normal in the lighting calculation.

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).