CG Programming II – Term Project
Due on 03/18/2008 – Day of the final

For the term project you are tasked to implement *one* of the following projects. Each project has a milestone that is due in class on 3/4/2008. This milestone will be graded.

Project #1 – Fins and Shells Fur

Implement "fins and shells" style fur with the full set of extensions detailed by Isidoro and Mitchell in "User customizable real-time fur." In addition, Gary Sheppard's "Real-Time Rendering of Fur" contains numerous useful implementation details that are missing from the Isidoro paper.
For class on 3/4/2008, have the following elements complete:

- Dynamically generate fur shell textures at program initialization.
  - Fur must be colored from a base color ("albedo" in the Isidoro paper) texture. This texture may be loaded from disk.
  - Generate a per-shell texture that contains a value that determines whether or not that texel contains a hair or not. In the initial implementation, you may want to disable mipmapping on this texture.
- Fur must be shaded as a factor of distance from the outermost layer of fur.
- Shells for straight hair must be implemented.

For class on 3/18/2008, have the following elements complete:

- Implement rendering of blended fins. It is advisable to implement fin rendering in several stages. While developing the fir rendering code, it is advisable to only render the base shell. Doing so will make it much easier to tell what is happening.
  - Render fully extruded fins *without* the fin textures. Set the color to always be white.
  - Implement alpha blending of fins. Architect this code so that it can be disabled easily. This will be a helpful debugging aid later.
  - Dynamically generate the fin textures and use them.
- Implement curved hairs with correct coloring. An additional "offset" texture is required per shell. This offset texture contains the location, relative to the current texture coordinate, of the hair's color in the base hair color texture. This texture should be combined with the other per-shell texture. Store the offsets in the red and green components, and store the hair mask in the blue component.
- Implement per-fragment fur lighting. In an additional per-shell texture, store the hair direction (tangent vector) at that location. The information in this texture for shell N is the negative direction of the information stored in the offset texture for shell $N+1$. The Z component of the hair direction depends on the shell-to-shell spacing.
- Implement shell texture mipmapping. Leave this step until the very last.

Project #2 – Nonphotorealistic Rendering

Implement a toon-style nonphotorealistic shader and a Gooch-style technical illustration shader. A user interface must be provided so that the user can switch between these two shaders at run-time.
For class on 3/4/2008, have the following elements complete:

- Implement 3-level, toon-style shading. The thresholds for light-dark and specular must be adjustable at run-time. Provide a user interface to modify these values while the application is running.
- Implement Gooch-style warm-to-cool shading.

- With both shaders, draw *all* edges as a second pass. To implement this correctly, you will have to become friends with `glPolygonOffset`.

For class on 3/18/2008, have the following elements complete:

- Implement correct “inking” of crease and silhouette edges. For the toon shader all inked edges should be black. For the Gooch shader the crease edges should be white and the silhouette edges should be black. Note that the sphere and torus models do *not* have any crease edges. This means that you will have to come up with some additional models. Simple platonic solids are perfectly acceptable.
- Implement variable thickness edges. The thickness for creases and silhouettes do not need to be independently adjustable. However, the thickness of all edges of a particular type should be consistent. The thickness of the edges should decrease with object distance (i.e., be perspective correct).

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).