Data structures & Algorithms for Geometry
Assignment #3 (3D Convex Hull)
Part 1 due on 11/10/2007, part 2 due on 11/17

This assignment will be completed in two parts. In the first part you will implement and test a data structure for storing and manipulating mesh data. In the second part you will use that data structure to implement a 3D convex hull algorithm.

For part 2, additional code will be provided to visualize the generated hull. It is *very* important for your code to conform to the specified interfaces. Otherwise, it will not work within my testing framework or with the visualization code.

- Part 1
  - Implement a data structure for storing and manipulating mesh data. Several acceptable data structures were covered in class on 10/27. Code for a mesh and polygon base classes will be provided.
  - For the `mesh` class, implement the following:
    - an STL-style *iterator* that will visit all of the polygons stored in the mesh.
    - an `add_polygon` member function
    - a `remove_polygon` member function
    - a constructor that takes four points as parameters.
  - For the `polygon` class, implement the following:
    - an STL-style *iterator* that will visit all of the points that make up the polygon.
    - a `plane_equation` method to get the plane equation for the polygon.
    - a `tesselate` method that divides the polygon in the n triangles, where n is the number of edges. This method takes a point as a parameter. The new triangles are made of the new point and the two points on each edge of the original triangle.
- Part 2
  - Use the mesh data structure from part 1 to implement a 3D convex hull algorithm.
  - Validate the implemeted hull algorithm with a series of tests. At a *minimum* the following tests must be implemented:
    - Validate that the generated hull is convex. One way to do this is by checking the angle between the normals of surfaces that share an edge.
    - Validate that all points are inside the hull. One way to do this is by checking that all points are in the negative half-space of all polygons. The `plane_equation` method will likely be useful here.

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).