24-783 Problem Set 8

Before starting update your public, data, and course_files directories by typing:

svn update ~/24783/src/public

svn update ~/24783/data

svn update ~/24783/src/course_files

If you haven't checked out data files for testing your program, files can be downloaded by:

```
svn checkout https://ramennoodle.me.cmu.edu/svn/teaching/data
```

Deadline: 04/18 (Wed) 23:59

Preparation: Set up CMake projects

You first create projects for the two problem sets.

1. In the command line window, change directory to:
   ~/24783/src/*yourAndrewId*
2. Update the course_files, data, and public repositories.
3. Use "svn copy" to copy base code to your directory:
   ```
   svn copy ~/24783/src/course_files/ps8 .
   ```
4. Add ps8/ps8_1 and ps8/ps8_2 sub-directories to your top-level CMakeLists.txt
5. Run CMake, compile, and run ps6 executable.
6. Commit to the SVN server.

Optional: Check out your directory in a different location and see if all the files are in the server.

PS8-1 Dihedral-Angle based Segmentation

In this assignment, you write a code that creates groups of polygons separated by high dihedral-angle edges. Your program must take two parameters from the command line for example:

```
(executable_file_name) c172r.stl 30
```

The first parameter is the STL file to be loaded, and the second parameter is the angle threshold in degree. (Therefore you must convert it to radian in the program).

Your program must then open the STL file and:

(1) Create a segmentation (groups of polygons) separated by edges that the dihedral angle of the two polygons sharing the edge is greater than the user-specified angle threshold. The polygons must be drawn white (unless you do (3)).
(2) Draw group boundaries in blue lines. Use glLineWidth(4).

Optionally,

(3) Paint polygons so that polygons within the same group have the same color, and the polygons of the neighboring group always have a (visibly) different color. (10 points bonus).

Dihedral angle can be calculated between two polygons. You use GetNormal function to get two normal vectors, and use operator* to calculate the cosine of the dihedral angle.

To create groups, you start traversal from each polygon that is not yet in a group. Create a temporary group that consists of only one polygon, and then grow the group by visiting edge-connected polygons while growing a group. However, if the dihedral angle between the current polygon and a neighboring polygon is greater than the user-specified threshold, do not grow the group into the neighboring polygon. Also you must not grow the group into a polygon that is already in a group. The traversal must stop when the group can no longer grow.

For this assignment, you implement two functions (if you go for the bonus question, three functions) in dha.cpp.

```
std::unordered_map <YSHASHKEY,int> MakeDihedralAngleBasedSegmentation(const YsShellExt &mesh,const double dhaThr);
```

```
std::vector <float> MakeGroupBoundaryVertexArray(const YsShellExt &mesh,const std::unordered_map <YSHASHKEY,int> &faceGroupInfo);
```

```
void MakeFaceGroupColorMap(YsShellExt &mesh,const std::unordered_map <YSHASHKEY,int> &faceGroupInfo)
```

Data type YSHASHKEY is an unsigned integer. It is possible to specialize std::hash for YsShellExt::PolygonHandle, however, because std::hash already has an implementation for unsigned integer, it is easier to use a vertex search key. Use GetSearchKey to get a search key from a polygon handle, and FindPolygon to get a polygon handle from a search key.

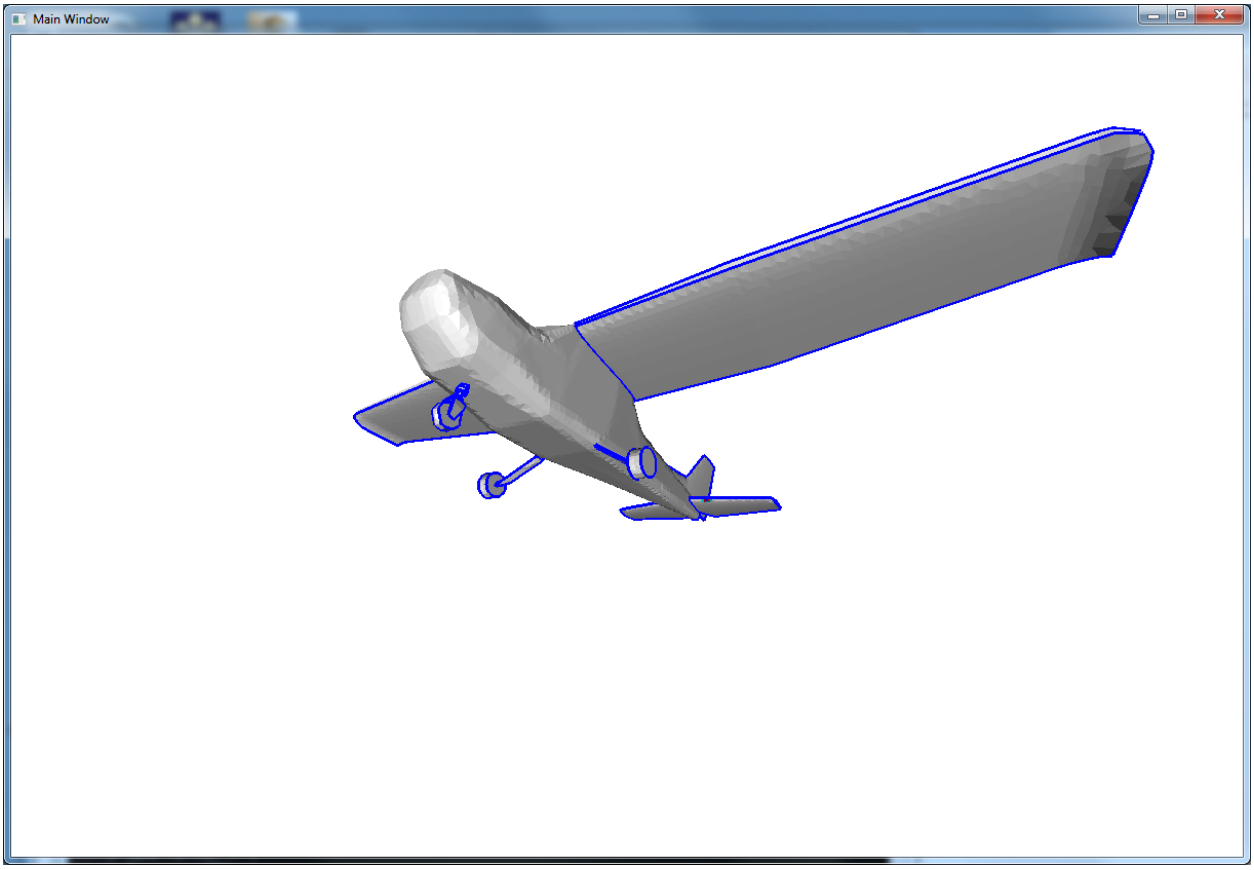
You can add your own functions in dha.cpp if you feel necessary. However, these three functions must be in the prototypes as defined in dha.h. Also the library "dha" may be tested by a different test program. If your own functions are used from these three functions, write it in dha.cpp.

The first function, MakeDihedralAngleBasedSegmentation, takes a mesh and dihedral-angle threshold as input parameters. The return value is an unordered_map from a polygon search key to a segment (face-group) identifier. Segmentation is equivalent to giving labels to the polygons. If two polygons have an identical label, the two polygons belong to the same group.

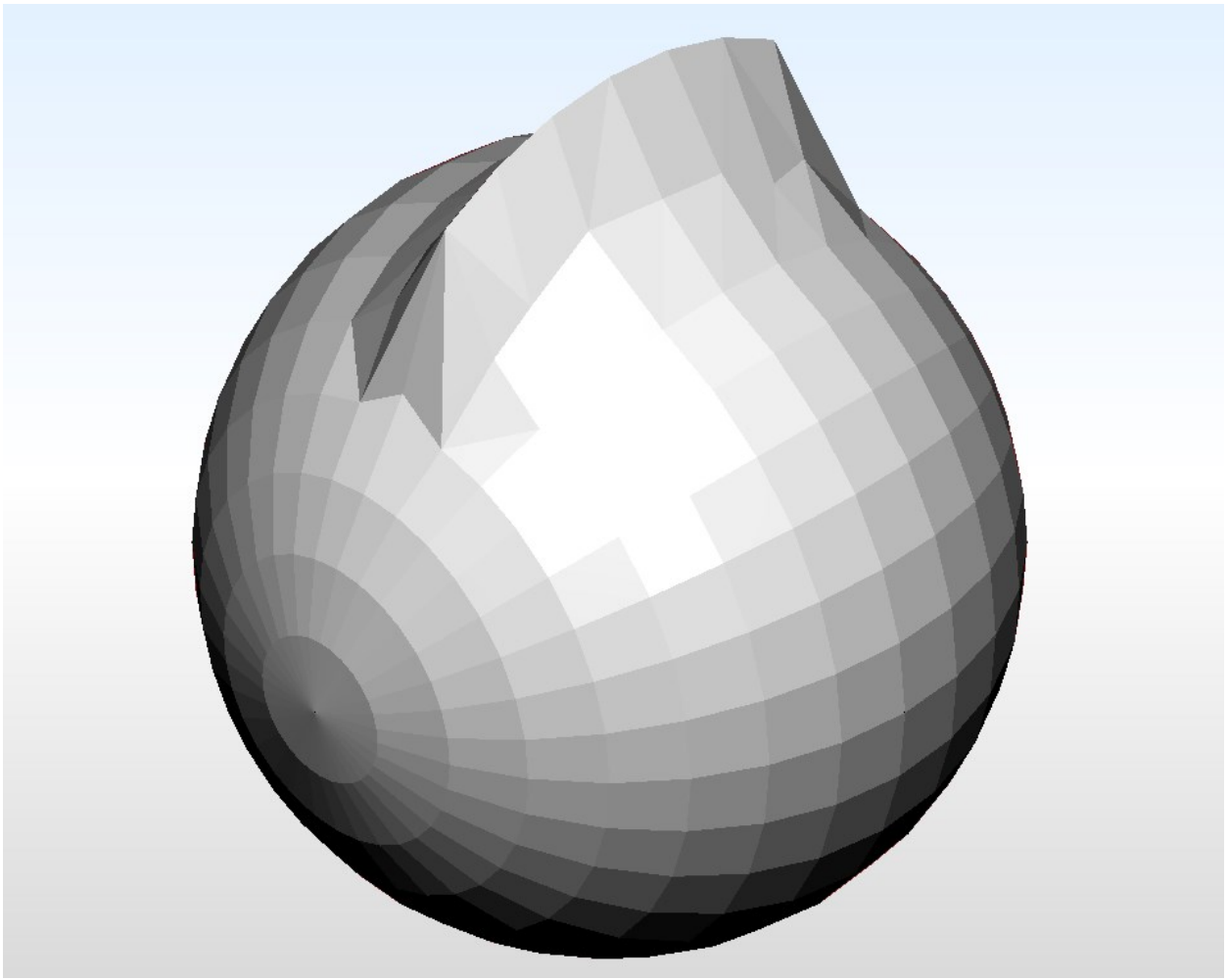
The second function, MakeGroupBoundaryVertexArray, takes a mesh and the segmentation created by MakeDihedralAngleBasedSegmentation as input, and returns a vertex array of the face-group boundaries that can be drawn as GL_LINES.

If you go for the 10-point extra credit, you implement the third function, MakeFaceGroupColorMap. In this function, polygons must be painted by SetPolygonColor, so that (1) all polygons in the same face group has an identical color, and (2) no two neighboring face-groups (two face-groups that shares at least one edge piece) are painted in the same color.

The following screenshot is of c172r.stl with dihedral threshold of 45 degrees.



Note that not all high dihedral-angle edges are a group boundary. For example, the following mohawk has high dihedral-angle edges. However, since it does not divide a group of polygons, these high dihedral-angle edges must not be drawn in your program.



PS8-2 A*Path Finding Algorithm

A* (A-star) is a very popular path-finding altorighm. You can find a very detailed explanation of the algorithm on Wikipedia. https://en.wikipedia.org/wiki/A*_search_algorithm.

You write a function (actually two functions) to find a shortest path connecting between two vertices on YsShellExt.

You implement two functions in astar.cpp:

```
std::vector <YsShellExt::VertexHandle> A_Star(
    const YsShellExt &shl,
    YsShellExt::VertexHandle startVtHd,
    YsShellExt::VertexHandle goalVtHd);

std::vector <YsShellExt::VertexHandle> A_Star_ReconstructPath(
    const YsShellExt &shl,
    const std::unordered_map <YSHASHKEY,YsShellExt::VertexHandle> &cameFrom,
    YsShellExt::VertexHandle current);
```

You can add your own functions in astar.cpp if you feel necessary. However, these two functions must be in the prototypes as defined in astar.h. Also the library "astar" may be tested by a different test program. If your own functions are used from these three functions, write it in astar.cpp.

Data type YSHASHKEY is an unsigned integer. std::unordered_map, std::unordered_set can be used for closedSet, g_score, and f_score. You can use YsAVLTree class, which is compatible with the AVL tree class you wrote for the assignment for keep vertices sorted in the order of f_score. You also need a map from a vertex to the AVL-tree node because in each iteration f_score of a node may change, and then the order needs to be updated. (I believe std::priority_queue is not designed for re-ordering based on the updated cost.)

It is possible to specialize std::hash for YsShellExt::VertexHandle, however, because std::hash already has an implementation for unsigned integer, it is easier to use a vertex search key. Use GetSearchKey to get a search key from a vertex handle, and FindVertex to get a vertex handle from a search key.

The pseudocode from Wikipedia suggest that you should add neighbor to openSet if neighbor is not in openSet during the iteration. However, if we use an AVL tree as openSet, you need to wait until you have fScore for the vertex until you can add it to openSet. Also, when you update the score of the neighbor, if the neighbor is already in the AVL tree, the tree node for the neighbor must first be deleted, and then re-inserted. Therefore, you need to re-organize the pseudocode to translate into the working program.

The heuristic cost estimate can just be a straight-line distance between two points.

If you implement correctly, when you build and run the executable ps8_2 with an STL file as the first command-line parameter, you will see three line strips that connects vertices of minimum x and maximum x, minimum y and maximum y, and minimum z and maximum z. Some models do not have

such paths, for example c172r.stl does not have a path connecting max-y and min-y vertices because front wheel is disconnected from the fuselage.

Pseudocode from Wikipedia.org
(https://en.wikipedia.org/wiki/A*_search_algorithm captured 03/31/2017)

```
function A*(start, goal)
    // The set of nodes already evaluated.
    closedSet := {}
    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}
    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := the empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity
    // The cost of going from start to start is zero.
    gScore[start] := 0
    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity
    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue          // Ignore the neighbor which is already evaluated.
            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if neighbor not in openSet    // Discover a new node
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue          // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor,
goal)

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```