Assignment #1:
Shape Analysis: Bounding Box, Euler Characteristic, and Surface Curvature Computation

Due Feb 15th, before midnight

Goals:

The first goal of this assignment is to become familiar with the PLY format and the common data structure for mesh representation and processing. The second goal is to review some important mathematical concepts and characteristics of surface as well as their numerical computation algorithms.

The detailed tasks are described below. You need to submit the source code and the associated library to ensure it can be compiled and run. Otherwise, you will be asked to do a live demo. *Please also remember to submit a well-written report with the description of the implemented algorithms and data structure if you have developed your own one. Figures and tables showing the results should be included as well as the discussion to address the questions asked in the individual tasks*. Fail to do so will result in the loss of points for this project.

The provided skeleton code includes all you need to read and write a PLY format data file, and a mesh data structure as described in the class. You are welcome to develop your own data structure. Note, that we will use this data structure for the next assignment as well. The skeleton code also contains a GLUI interface. The detailed document of how to use this interface can be found here (http://www.cs.unc.edu/~rademach/glui/src/release/glui_manual_v2_beta.pdf). *Please use the version of the library included in the skeleton code in order to avoid the potential compiling issue*. If you have better GUI library that can run across different platforms, go ahead to use it. Just make sure to include the necessary library in your submission.

Tasks:

1. Compute Euler Characteristic (50 points)

Print the numbers of vertices (V), edges (E), and faces (F) of all the test models. Compute the Euler characteristic (V-E+F) for all the models. Count the numbers of handles in these models. A handle is also called a tunnel.

(What to report) *Report your results in a Table with each column corresponding to each of the numbers you get above. What relationship do you find the number of handles with the Euler characteristic of the shape? Can you provide some intuition about it?*

2. Compute Bounding Box (100 points)

Compute the oriented bounding box (OBB) for all the test models. The algorithm has been described in the class.

1) Compute the center of mass from the vertices of each model
2) Derive the covariance matrix

3) Compute the eigenvalues and eigenvectors of the covariance matrix. Obtain the major, middle, and minor eigenvectors based on the order of the eigenvalues.
4) Derive the bounding box from these three axes. Note that you need to evaluate how far away each vertex is from the center of mass along the three axes direction (NOT the X, Y, Z directions).
5) Visualize your bounding box as a wireframe hexahedron with the same color being assigned to the edges that are parallel to each other. There should have three groups of edges corresponding to the major, middle, and minor directions of the shape.

To implement this algorithm, you will need to compute the eigenvalues and eigenvectors of a 3x3 symmetric matrix. The skeleton code has provided a method to allow you to solve for this problem using singular value decomposition (SVD). See the description of the following function in the "svd3.h/.cpp"

```
void svd3(double * U, double * S, double * V,
        const double * A);
```

Another option is to use the algorithm and C implementation from Numerical Recipe (Section 11.1):

http://www.cs.helsinki.fi/u/tvvlappi/muuta/Numerical_Recipes/bookc/c11-1.ps

You are also welcome to use other library and write your own implementation for this. But please make a note in your report if you do so.

(What to report)

Show the bounding boxes for all the test models. The skeleton code already has an implementation of the axis-aligned bounding boxes (ABB). Use the same visualization for this ABB. Modify the interface and *insert two checkboxes* to allow the user to choose to display ABB and OBB, respectively (see the skeleton code to learn how to add checkbox). Compared to OBB approach, which one is more preferred, and why?

There are some models that using the OBB method cannot produce a valid bounding box. Find out those models and explain why this happens? Can you propose a solution to that?

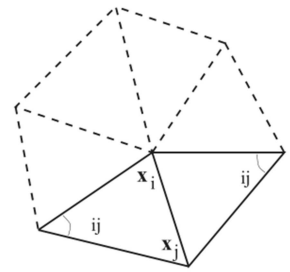
Can you develop an object collision detection algorithm based on the bounding box of an object? What issue could you encounter if only one bounding box is used for each object? Can you propose a solution to that?

3. Curvature Estimation (200 points)

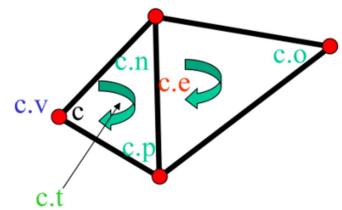
Implement the discrete curvature estimation algorithms by Meyer et al. http://www.multires.caltech.edu/pubs/diffGeoOps.pdf. You will need to estimate and visualize the mean curvature and Gaussian curvature. Use proper color coding to visualize these different curvature estimations.

3.1 (Estimate Gaussian Curvature, 40 points) The algorithm for computing a discrete Gaussian curvature is in Section 4 of Meyer et al.'s paper. Specifically, for each vertex of the mesh, you need to compute Equation (9). Again, Fig. 4 provides the pseudo-code for computing the area of the Voronoi region around a specific vertex, which is the A_{Mixed} in Equation (9).

3.2 (Estimate Mean Curvature, 40 points) The algorithm for computing a discrete mean curvature is in Section 3 of Meyer et al.'s paper. Specifically, for each vertex of the mesh, you need to compute Equation (8). Fig. 4 provides the pseudo-code for computing the area of the Voronoi region around a specific vertex, which is the A_{Mixed} in Equation (8).



3.3 (Implement Corner Table, 40 points) In order to obtain the opposite angle of an edge that is incident to the vertex v on which you are computing the mean curvature, you can use a data structure, called Corner table. Basically, a triangle contains three corners, each of which corresponds to a vertex and consists of the two edges that are incident to the vertex. The following shows the data structure of a corner.




```
class Corner{
public:
        unsigned char Edge_count;   //special variable for edges search
        int index;
        int v;            //the ID of the vertex of the corner
        int n;            //the next corner according to the orientation
        int p;            //the previous corner according to the orientation
        int t;            //the triangle the corner belongs to
        int ot;           //the index of its opposite triangle for traversal
        int o;            //the index of its opposite corner
        Edge *e;          //the opposite edge of the corner
        float angle;       //the angle of the corner
        float BeginAng, EndAng;  //for correct angle allocation of the corner around vertex v
        float r;
        bool orient;

        ////////////////////////////////////////////////
        /* Optional variables */
        Edge *edge[2];    //two edges associated with this corner

        Corner()
        {
                e = NULL;
                edge[0] = edge[1] = NULL;
        }

        double get_Angle();
};
```

You will need to implement the construction of Corner table in order to get the information for the individual member variables for each corner.

The Corner table is stored in the member variable of class "`Polyhedron`".

```
        CornerList clist;          //the corner list
```

For a vertex `vlist[i]` and its given jth corner, you can use the following piece of code to get the two angles.

```
Corner *Cur_corner = vlist[i]->corners[j];
float theta1 = clist.corners[Cur_corner->p]->angle;

int oppositecorner = clist.corners[Cur_corner->p]->o;
float theta2 = clist.corners[oppositecorner]->angle;
```

3.4 (Smooth the discrete mean and Gaussian curvatures, 40 points)

Implement a simple smoothing scheme for the previously obtained curvatures. The pseudo-code of doing this smoothing is as follows.

For iter = 1 to *Maximum_iteration*

For each vertex V_i

$$curvature(V\_i) = old\_curvature(V\_i) + dt \sum_{j \in N(i)} \omega_{ij} (old\_curvature(V_j) - old\_curvature(V_i))$$

Here N(i) are the one-ring neighborhood of vertex V_i. Simply use $\omega_{ij} = \frac{1}{N}$ for this implementation.

3.5 (Visualizing the mean and Gaussian curvature, 40 points)

The obtained mean and Gaussian curvatures give rise to two scalar fields on the surface. Visualize them using color plot. A mapping from the scalar value to color value is provided at the end of this description. You can also develop your own mapping strategy. In the user interface, please include four sliders to allow the user to adjust the truncated thresholds for the maximum and minimum curvatures, respectively.

3.6 (What to report)

Explain the color coding you choose for each curvature visualization. Does your curvature estimation reflect the expected surface characteristics, i.e. highlighting creases, ridges, and sharp features? What issue do you think the current estimation has? Can you propose a solution to addressing that?

Compute the total Gaussian curvature by summing up this curvature at all vertices. Report your result for each model. What do you find? Can you think of a way to estimate the total Gaussian curvature of a surface via the numbers of vertices, edges, and faces on the mesh?

Compare the result before and after smoothing, what improvement you see? Report the parameters, *Maximum_interation* and *dt* you use for the smoothing.

Is the discrete estimation of mean and Gaussian curvature sufficiently accurate? What can affect the accuracy? What is the other way to compute these two curvatures?

Grades:

Tasks	***Total points***
1	50
2	100
3	200

Submission of the assignment

You will need to submit the report and a link to the code for this assignment via the blackboard learn system. You will also need to send me and TA an email about this submission.

Suggestions

Using the GLUI Range Sliders

We have added range sliders to the GLUI library. Here is how to use them.

For each variable, define a text-display format, a 2-element array to hold the low and high end of the range, a pointer to the created slider, and a pointer to the created text-display:

```
// in the global variables:
#define TEMP     0
const float              TEMPMIN = {   0. };
const float              TEMPMAX = { 100. };
const char *             TEMPFORMAT = { "Temperature: %5.2f - %5.2f" };
float                    TempLowHigh[2];
GLUI_HSlider *           TempSlider;
GLUI_StaticText *        TempLabel;
. . .
```

```
// in the function prototypes:
void     Buttons( int );
void     Sliders( int );
. . .
```

```
// in InitGlui( ):
         char str[128];
         . . .
         TempSlider = Glui->add_slider( true, GLUI_HSLIDER_FLOAT, TempLowHigh,
                  TEMP, (GLUI_Update_CB) Sliders );
         TempSlider->set_float_limits( TEMPMIN, TEMPMAX );
         TempSlider->set_w( 200 );         // good slider width
         sprintf( str, TEMPFORMAT, TempLowHigh[0], TempLowHigh[1] );
         TempLabel = Glui->add_statictext( str );
         . . .
```

The arguments to `Glui->add_slider( )` are, in order:

range_slider	*true* means this is a 2-edged range slider
type	Use GLUI_HSLIDER_FLOAT
array	2-element float array to store values in
id	unique id to be passed into the callback routine (0, 1, 2, ...)
callback	callback routine to call when a slider is used

The arguments to `TempSlider->set_float_limits( )` are the minimum and maximum values on that slider.

The argument to `TempSlider->set_w( )` is the width, in pixels, of that slider in the GLUI window. 200 is a good number.

The argument to `Glui->add_statictext( )` is the text string to display.

The Button Callback Routine

The buttons callback routine needs to be modified do re-do all the text strings if the Reset button is selected:

```
void
Buttons( int id )
{
        char str[256];

        switch( id )
        {
                case RESET:
                        Reset( );
                        sprintf( str, TEMPFORMAT, TempLowHigh[0], TempLowHigh[1] );
                        TempLabel->set_text( str );
                          . . .
```

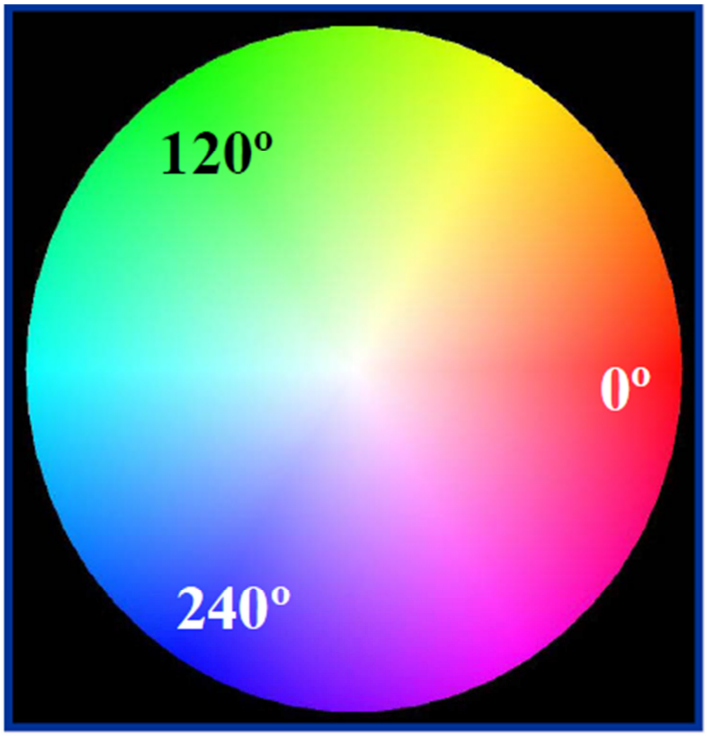
The Slider Callback Routine

All range sliders can use the same callback routine:

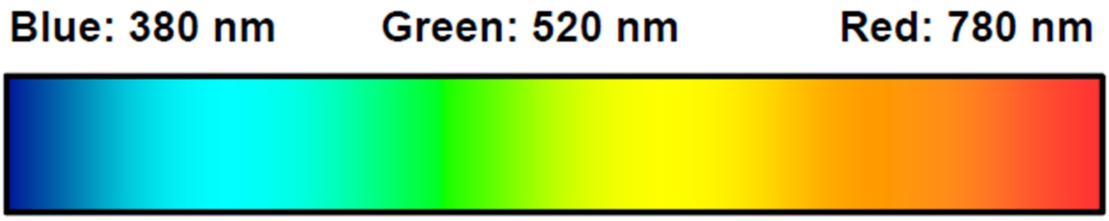
```
void
Sliders( int id )
{
        char str[32];

        switch( id )
        {
                case TEMP:
                        sprintf( str, TEMPFORMAT, TempLowHigh[0], TempLowHigh[1] );
                        TempLabel->set_text( str );
                        break;

                 . . .
         }

        glutSetWindow( MainWindow );
        glutPostRedisplay( );
}
```

Color Coding:

For many visualization applications, a simpler way to specify additive color is to use **Hue-Saturation-Value** space.



Notice that blue-green-red in HSV space corresponds to the visible portion of the electromagnetic spectrum.



The easiest way to turn a scalar value into a hue when using the Rainbow Color Scale

$$Hue = 240. - 240. \frac{S - S_{min}}{S_{max} - S_{min}}$$

Here, *s* is the scalar value defined at a vertex, S_{min} and S_{max} are the minimum and maximum scale values through the whole mesh.