Spring 2002 ICS 186a
Programming Assignment 3

Due: May 10, 2002, 11:59 PM PDT

1 Problem Description

In this lab, the task will be the subdivision of triangular faces in an arbitrary, manifold triangle mesh. This will require the student to understand the indexed face list data structure, as introduced in lab two and basic memory management in C and/or C++. This lab builds upon the previous two labs which implies the use of the code implemented in `myGL.c` along with the animation and lighting features from lab two. In this lab, you should have mastered how to specify material properties and lighting. The goal of this lab is to successfully read in a triangle mesh from an input file and interactively subdivide the mesh faces. For each subdivision, the new vertices are offset by some delta value, depending on the orientation of the normals along the edge being subdivided. The end result should be a program which enables the user to view the input mesh at its initial resolution (input number of vertices) and interactively step up or down through the subdivided mesh.

The binary that is to be built will be called `subdivide`. The command line invocation of your lab will look like this:

```
prompt% subdivide 0.1 bunny.ply
```

The user controls are identical in this lab to the previous labs save for the addition of two keyboard mapped callback functions. The *space bar* should be mapped to the callback procedure which subdivides the input surface which then displays the new higher resolution mesh. That is, if the user has pressed the *space bar* n times, then the surface should be subdivided n times. The *'b'* key should be mapped to the callback procedure which redisplays the current surface with $n-1$ subdivisions. In this manner, the user can step up or down through the subdivided mesh.

2 Implementation

2.1 PLY File Reader

The input will be specified as a plain ASCII text file in the PLY file format. A PLY file reader will be provided in the framework code. The reader will read in the file and initialize an indexed face list data structure. The definition for the indexed face list data structure is in the file `sceneModule.h` and `sceneModule.c`. For further details about the data structure, refer to the code in `/home/graphics/teaching/ics186a/gopi-S02/framework-lab3`.

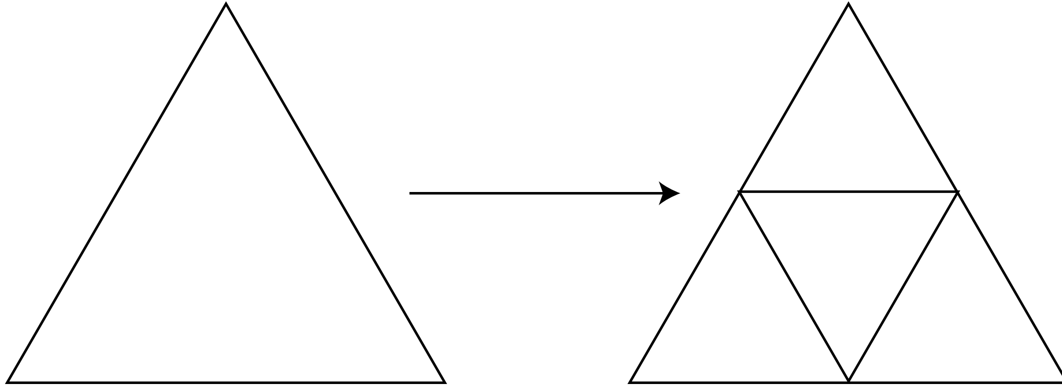

Figure 1: A face after subdivision

2.2 Subdivision

Each face of the input mesh must be divided into four faces. The rudimentary algorithm follows: For each face:

1. Find the midpoint of each edge
2. *(Translate the new vertices - see next section)*
3. Update the vertex list with the computed midpoints
4. Update the face list: delete the old face, insert the four new faces

The subdivision must be interactive. That is, every time the user depressed the space bar, the program will run the subdivision routine and refresh the display. In previous assignments, the model being rendered was stored in an OpenGL display list, however in this lab each polygon will be rendered directly from the indexed face list data structure. For students who have the inclination, allocating and deallocating OpenGL display lists should be relatively simple to implement.

To successfully implement the subdivision portion of this lab, a face adjacency list is needed. Imagine subdividing a face which shares its edges with three other faces. Subdividing an adjacent face will re-subdivide one of the previously subdivided edges. With an adjacency list, each edge should only be visited once in each subdivision iteration. The adjacency list should be added to the indexed face list data structure. Do not forget to also update each face's normal information and each vertex's normal information.

Since vertices and faces are being added to this data structure, it is imperative that the memory for the indexed face list be allocated dynamically. The relevant functions in C are `malloc`, `calloc`, `realloc`, and `free`. More information about these functions can be found using the relevant man pages or by refering to a C programming language reference text. Furthermore, completed labs should not have memory leak. Memory leak, as defined from the Jargon File, is "[a]n error in a program's dynamic-store allocation logic that causes it to fail to reclaim discarded memory, leading to eventual collapse due to memory exhaustion." That is, if you allocate memory, make sure you `free( )` it. (Do not use the built in garbage collection in C++.)

(Completed labs using static memory allocation will not be accepted.)

Two routes can be taken in order to step up or down through the subdivision process. One would be to have a counter which keeps track of how many steps up or down the user goes. Thus

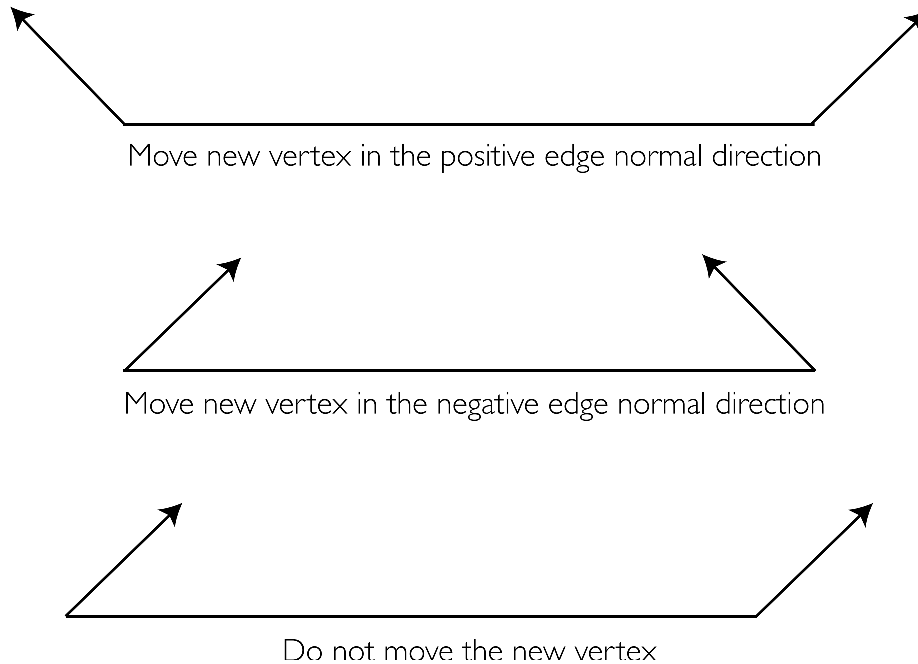

Figure 2: An illustration of what should be observed when translating a new vertex.

each event generated by the user will trigger the destruction of the old data structure and the generation of the new data structure. This approach wastes *precious* CPU cycles.

The better approach would be to store the different resolutions in a linked list. As the user steps up or down in the mesh subdivision process, CPU time is not wasted recreating the data structure. This approach sacrifices memory but improves interactivity.

Students who have the time and inclination are invited to implement alternative data structures which conserve memory while at the same time saving CPU cycles. (This is not a replacement to this assignment's requirements, it is in addition to; for those who wish to undertake the challenge.)

2.3 Vertex Translation

2.3.1 Overview

Once you have completed the subdivision routine of this lab, the next step is to augment it to translate the vertices depending upon the orientation of the normals along the edge being subdivided, and length of the edge. The delta value, the scale factor by which a newly generated vertex will be translated by, should be specified on the command line. The value of delta should be a floating point number between -1.0 and 1.0, inclusive.

For a particular edge, look at the vertex normals for the two vertices at either end of the edge. If the vertex normals point away from each other, then translate the new vertex along the positive edge normal (average of the normals at the end vertices of the edge) direction. If the vertex normals point towards each other, then translate the new vertex along the negative edge normal direction. If the vertex normals point in the same direction or opposite directions, then do nothing. Refer to figure 2 for an illustration of the cases enumerated.

2.3.2 Computing the Direction of the translation vector

Let ABC be the vertices of a triangle in anti-clockwise direction (as is the convention in computer graphics). Let N_a, N_b, and N_c be normal vectors at A, B, and C respectively. Assume that you are subdividing the edge AB and let D be the new mid point. $N = N_a \times N_b$ would give a vector perpendicular to N_a and N_b. Check if the third vectex (C) lies above or below the plane defined by N_a and N_b. How to check this? Let AC is the vector from A to C. If $AC \cdot N$ is positive then C is *"above"* the plane, if 0, then on the plane, and if negative then *"below"* the plane. (Actually, you should not get the condition zero in your models. But check for this condition and print an error if that happens.) If $AC \cdot N$ is positive, then the direction of translation is $N_a + N_b$. If $AC \cdot N$ is negative, then the direction of translation is $-(N_a + N_b)$.

2.3.3 Computing the Magnitude of the translation vector

The magnitude of translation depends on the resolution of the mesh and the value of delta. The amount of translation of the mid-point of the edge is the product of delta and the length of the edge. Given an edge AB, the amount (magnitude) of the translation is the length of $AB \times \delta$. As the length of the edge decreases over multiple subdivisions, the amount of translation also converges to zero. (For each new subdivision, say level *'n'*, the new values of edge lengths of *level (n-1)* should be used.)

You should experiment with different values of delta. Several things to try are: 1) use a small value for delta, but do many subdivisions, 2) use a large value for delta, but do few subdivisions, 3) use a large value for delta, but do many subdivisions, 4) use a small value for delta, but do few subdivisions. Make note of the results from these experiments and any other observations in your `README` file. (This is part of the graded assignment, do not forget!)