CS6533: Interactive Computer Graphics
Meshes and Subdivision

11/05/2013

Assignment Objectives

The purpose of this assignment is to use a mesh data structure and implement Catmull-Clark subdivision.

We provide you with a mesh data structure, a mesh file, and some shaders. You will build on top of the previous assignment and reuse the scene graph, the material infrastructure, and the corresponding picking and manipulation functionality. We provide a file "mesh.interface" which documents the interface provided by the `Mesh` class, and a sample usage of the `Mesh::VertexIterator` class which you will need in order to walk around the 1-ring of a vertex.

Preparation

Copy all the files in the starter zip into your project directory. This time you will use another `Geometry` implementation, the `SimpleGeometryPN` in addition to the `SimpleIndexGeometryPNTBX` employed in the previous assignment. The `SimpleGeometry*` classes internally do not use an index buffer to store vertex indices for each triangle. Instead triangles are formed by grouping together three consecutive vertices in the vertex buffer. Thus the i-th triangle is made up from the (3i)-th, (3i+1)-th and (3i+2)-th vertex in the vertex buffer (counting from zero). This unindexed format is useful since later on, you will draw the subdivision surface in "faceted shading" mode first, which means different triangles cannot share vertices as they have different normals.

Both `SimpleGeometry*` and `SimpleIndexeGeometry*` have an `upload` function for (re)specifying the geometry data, e.g.,

```
void SimpleGeometryPN::upload(const VertexPN* vertices, int numVertices);
void SimpleIndexedGeometryPNTBX::upload(const VertexPNTBX* vertices,
                                        const unsigned short* indices,
                                        int numVertices, int numIndices);
```

Basically you need to pass it a pointer to a contiguous block of memory containing the vertex data, and the number of vertices contained in that region. You need to do the same for indices when dealing with `SimpleIndexedGeometry*`. You can also optionally provide the geometry data in the constructors of the `Simple*Geometry*` classes.

There is also a new fragment shader `specular-gl{2|3}.fshader`. You should pair it up with the `basic` vertex shader and load them into a new `Material`. Specify a color for that material using the `uColor` uniform variable, and use it for drawing the subdivision mesh.

Task 1 (2 points)

Read in the cube mesh from the provided file "cube.mesh". The mesh will just be a cube with 6 quads as faces.

Implement a function that uploads a `Mesh` to an existing `SimpleGeometryPN` object. This allows the mesh to be drawn on screen. Specifically, add and implement another `upload` function in the `SimpleGeometryPN` class:

```
void SimpleGeometryPN::upload(Mesh& mesh, bool isSmoothShading);
```

If `isSmoothShading` is `false`, we will draw the mesh using "flat shading", wherein you use the same normal for all vertices in a face. You can use the mesh data structure call to `Face.getNormal()` to obtain the proper normals for each face.

Although the `Mesh` data structure stores quads, you will need to turn each quad into two triangles before loading into the `SimpleGeometryPN` object. Once you have obtained the vertex positions and normals from the `Mesh`, specify the geometry by calling already existing `upload` member function as previously described.

Next add a new `SgRbtNode` to the scene graph, along with a `MyShapeNode` child constructed from your `SimpleGeometryPN` instance and your `Material` instance (created off the "specular" fragment shader).

Now when the scene graph gets drawn, the dynamic geometry for the subdivision surface should be drawn, although it is only a cube for now.

Task 2 (2 points)

Next you will implement "smooth shading", which requires an "average normal" for each vertex. To achieve this, implement the following function in the `Mesh` class:

```
void Mesh::updateNormals();
```

An average-normal at a vertex will be the average of the normals of all the faces incident to the vertex in the mesh. Once you have set the normal at a vertex using `Vertex.setNormal()`, you can read it back using `Vertex.getNormal()`.

One way to compute the average normal would be to use a vertex iterator. In order to deal with non-manifold meshes, which may appear in later assignments, we instead suggest the following. First zero out all of the vertex normals. Then, iterate through the faces of the mesh. For each face, accumulate its normal to all of its surrounding vertices. Once this is done, to get the correct average vertex normals, you visit each vertex and normalize the accumulated normal. This last step will use a call to `Vertex.getNormal()`.

To draw the mesh with "smooth shading", use the average-normal of each mesh vertex to set its value in the `SimpleGeometryPN` object.

Task 3 (1 point)

Register a GLUT timer callback (as in previous assignments) and create a "bubbling" animation by "animating" the vertices of the cube. In particular just scale the object coordinates of each of the cube's vertices by a periodic, time-varying scalar. (You should have a different scale for each vertex. Also the `sin` function could be used here.) Every time, the cube's vertices are animated, you should dump it to the geometry, so the updated geometry will be drawn when the scene graph gets drawn. Similar to the animation play/stop controls in previous assignments, also add the hot key 'b' to toggle between playing and pausing the bubbling animation.

Task 4 (5 points)

Implement subdivision. Use the Catmull-Clark rules to calculate the coordinates for new face-vertices, new edge-vertices, and new vertex-vertices (in `asst7.cpp`):

```
static void subdivideMeshCatmullClark(Mesy& mesh);
```

The easiest way to do this is to first loop over all of the faces of the mesh and compute faceVertex values. Then loop over all of the edges and compute edgeVertex values. Finally loop over all of the vertices and compute vertexVertex values. This last computation for vertexVertex values will require using vertexIterators to walk around the neighborhood of a vertex (for more details on how to use the `Mesh::VertexIterator`, refer to the `Note` at the end). As they are computed, you can place these coordinates in the appropriate slots in the data structure Using calls to `setNewFaceVertex, setNewEdgeVertex` and `setNewVertexVertex`. This calculation will also require looking up faceVertex values just computed so you will also need calls to `getNewFaceVertex`.

Once all of the slots are filled in, then call the subdivide routine. This routine will use the coordinates you provided for face-vertices, edge-vertices, and vertex-vertices to compute the new, subdivided mesh.

Once you are done with the subdivision, you will compute average-normals for the final vertices, and use these normals when drawing the mesh using "smooth-shading".

You final version will include the following hot keys:

- 'f' will toggle between smooth and flat shading.
- 'b' will toggle between the play/pause of the bubbling animation.
- '0' will increase the number of subdivision steps applied to the cube before being drawn. You should cap the subdivision steps around 6 and 7.
- '9' will decrease the number of subdivision steps applied to the cube before being drawn.
- '7' will half the speed at which the cube deforms.
- '8' will double the speed at which the cube deforms.

When the subdivision level is high, your program might run really slow. It helps to instruct the compiler to build the software in optimized mode, which will makes the program much faster at the expense of debugging convenience. To do so, under Visual Studio, choose "Release" as opposed to "Debug" from the drop down list box in the tool bar, and build. Under Mac/Linux, first do a `make clean` to remove any compiled files. Next do a `make OPT=1` to rebuild with optimization turned on.

Note:

Remember that you will be starting from the simple cube and applying subdivision every time that you need to animate the cube and update the dynamic geometry. To do this you can use two meshes: a "reference" mesh that holds the original cube unmodified, and a "temporary" mesh. Every time that your idle function is called, you can create a temporary mesh to be a copy of the reference mesh (a copy constructor and copy assignment operator is provided in the `Mesh` class), you can then subdivide the temporary mesh and draw it. This way the reference mesh is unchanged and still represents the simple cube, so you can use it the next time you need to draw a frame.

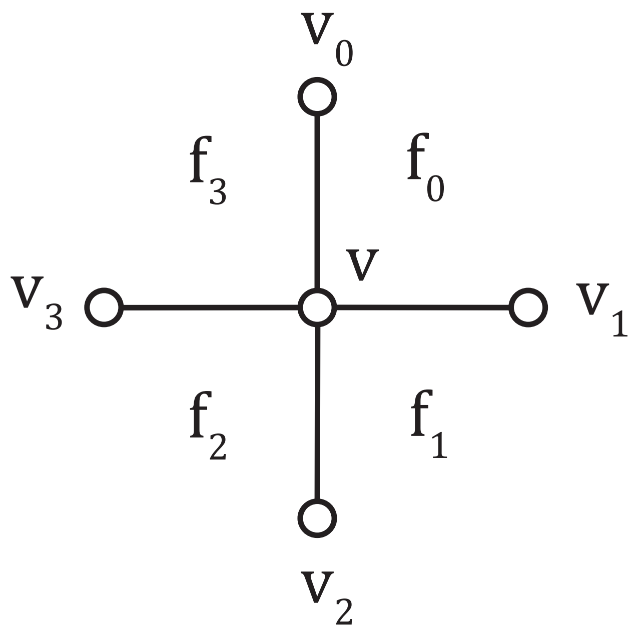

Figure 1: Accessing the one-ring neighborhood of vertex v

When calculating the new vertex-vertex positions in the Catmull-Clark subdivision, for each vertex v, you need to access the n_v neighboring vertices v_j's and the n_v surrounding faces f_j's. You can use `Mesh::VertexIterator` to access the one ring neighbor of v. Specifically, given a `vertex` (of type `Mesh::Vertex`) in a `mesh`, you can obtain a vertex iterator with the following:

```
Mesh::VertexIterator vIt(vertex.getIterator());
```

`vIt` gives you access to its neighboring vertices and faces. For example, in Fig. 1 , after initializing `vIt`, we can access v_0 and f_0 using:

```
Mesh::Vertex v0 = vIt.getVertex();
Mesh::Face f0 = vIt.getFace();
```

To move on to the next neighbor, do the following:

```
++vIt;
```

Then calling `getVertex()` and `getFace()` will give you v_1 and f_1.