Discussion Project 3

bajaj@cs.utexas.edu

October 30, 2007

Two Main Goals in Project 3

1. Use an adapted version of *4 Point Subdivision* to subdivide curves in 3-D
2. Perform Phong and Gouraud shading

Subdivision involves taking a set of control points that represent an object and then on each subdivision iteration adding new control points and possibly adjusting old control points. In general, because the number of control points is increasing on each subdivision iteration, the object will look smoother and smoother as it is further subdivided.

Phong and Gouraud shading allow one to determine the color of a pixel based on interpolation of only a sample of color information, which usually specified at the vertices of the geometry. Colors at each individual pixel do not have to be specified, which greatly reduces the communication required between the program and the graphics card. More realistic lighting can also be achieved.

The 4 Point Scheme

Consider a 2-D curve which is represented as a sequence of vertices where each vertex is connected to its two immediate neighbors in the sequence. The vertices are treated as control points. When these control points are subdivided, additional control points are created.

Going from subdivision level j to subdivision level j+1 (also referred to in these notes as iteration j to iteration j+1), we add one new control point to each edge between existing control points (you can think of this process as adding the new control point onto the edge, although it is unlikely that the new control point will lie exactly on the edge after the averaging). In the 4 Point Scheme, we will also be keeping all of our old control points (i.e. all control points in iteration j are in iteration j+1, unchanged). Thus, since we are adding one control point to each edge, and we are keeping all of our old control points, we will be approximately doubling the number of control points each time we subdivide.

For a closed curve (i.e. no loose ends) with n control points in iteration j, iteration j +1 will have 2*n control points. For an open curve with n control points in iteration j, iteration j+1 will have (2*n) - 1 control points.

Transforming Pixel Locations to Viewing Frustum Coordinates

The Glut mouse routine in the starter code returns the pixel coordinates when the

user clicks the mouse in the Glut window. The pixel origin is located at the upper left corner of this window. We need to transform these pixel coordinates (within the window) to viewing frustum coordinates, which are what we use to specify objects to OpenGL. Let the screen be (window_width x window_height) and let the viewing frustum be (world_width x world_height). Let world_left be the leftmost clipping plane of the frustum and let world_bottom be the bottommost clipping plane of the frustum. To transform a point (window_x, window_y) in pixel coordinates, we use:

```
world_x = [(window_x + 0.5) / window_width] * world_width +
world_left
world_y = [(window_height - window_y) + 0.5) / window_height] *
world_height + world_bottom
```

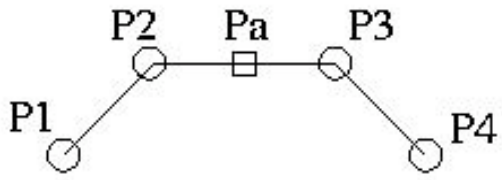
The "0.5" term is related to the dimensions of a pixel, which have a padding distance of 0.5 from their edges to their centers. Also notice since the pixel origin is in the upper left hand corner of the screen, we reverse the y axis by calculating `(window_height - window_y)` before applying the scaling.

The Weighting Rule

The location of the new control point is defined by the location of the four contiguous control points that are closest to an edge (the 2 control points immediately to the left of an edge, and the 2 control points immediately to the right). Let Pa be the new control point that will appear "on" edge (P1, P2), and let P0, P1, P2, P3 be the control points closest to edge (P1, P2). The weighting rule is:

```
Pa = (-1/16) * P1 + (9/16) * P2 + (9/16) * P3 + (-1/16) * P4
```

This weighting scheme is derived from the a single row in the local subdivision matrix for the 4 Point Scheme in the Lecture notes. Basically, all control points that are not coincident with an existing control point (i.e. that will appear on edges) are derived using these weights.



A Word on Vector Notation

We are working with 2-D points for the 4 Point Scheme, so you can think of the point P1 as the vector P1 = (P1_x, P1_y). Let a and b be scalar constants. The formula:

P = a * P1 + b * P2

... is shorthand notation for the two scalar equations:

P_x = a * P1_x + b * P2_x
P_y = a * P1_y + b * P2_y

The 2-D 4 Point Subdivision equation is in this form. Notice no x coordinate appears in the same equation as a y coordinate, so we can handle all subdivision calculations on x coordinates first, then all subdivision calculations on y coordinates, and then put the x and y results back together. This extends to the z dimension, as well.

Overview of Implementing the 2-D Version of 4 Point Subdivision

The basic algorithm for each level of subdivision:

```
curr_points[] = display_points[];  // the previous level's
control points

allocate an array for the new points that we generate (this
should be the
   size of curr_points[])
let this new array be stored in variable new_points[]

foreach set of 4 contiguous control points in curr_points[] {
    find the new control point by the weighting rule on the
current 4 points
    add this new point to new_points[]
}

curr_points[] = merge curr_points[] and new_points[] by
interleaving
                (i.e. picking from array 1, array 2, array1,
array 2...)
```

Alternate: Instead of interleaving two arrays, use a current and next array. Copy the *point i* from current into *point 2i* in next. Fill in the odd points with the new control points.

Open Curves

Problem: What do you do for an open curve? For the two outermost edges, there are not enough control points.

Recommended Solution: Use the endpoints twice. This method has its advantages

(simplicity) and disadvantages (the last edge may become sharp). Alternatives include computing a phantom control point past the outermost control point which is not displayed but is used for subdivision.

Drawing the 2-D Curve

You need to draw 2 different parts for the curve:

- The location of the iteration 0 control points. These are the control points the user has entered with the mouse. One way of doing this is to use: glPointSize(5.0f), followed by glBegin(GL_POINTS), and then iterating through the control points and drawing each one with a glVertex3f(), and then call glEnd().
- The subdivision curve, which is defined by the control points at the current subdivision level (aka iteration level). These points should be stored in a separate location than the iteration 0 control points. To draw the curve that is defined by these points, you should call glBegin(GL_LINE_LOOP) and iterate through the points, just like for VRML objects in Project 1. You should not draw the actual control points here, just the edges that connect them.

Representing the 3-D Surface

The 3-D Surface is represented by a mesh of control points. Initially, you will have 3 * [num of c.p.'s entered by their use in 2-D mode] control points in the mesh. You may exploit the surfaces even distribution of control points by using a 2-D array. In the 2-D array, a row can represent the control points that are next to each other in a stack (going across), and a column can represent the control points that are next to each other in a slice (going down).

4 Point Subdivision in 3-D

There are two types of subdivision: Horizontal and Vertical Subdivision

- **Horizontal Subdivision** - increases the number of control points in the horizontal division. Points that are in the same "ring" around the model are treated as one curve and subdivided.
- **Vertical Subdivision** - increases the number of control points in the vertical division. Points that are on a "vertical stripe" down the model are treated as one curve and subdivided.

In both cases, you can adapt the 2-D version of 4 point subdivision, although notice that in vertical subdivision, there are 2n - 1 control points (new and old) after subdivision of n control points. Why is this? How many control points result from horizontal subdivision of m control points.

Managing Memory is a Pain... When You Have to Do It

Although its not the best of programming practice, for this assignment you may allocate two static arrays, each big enough to hold all of the information you will use. For example:

```
GLfloat currCP[max_h][max_v][3]
GLfloat nextCP[max_h][max_v][3]
```

The values of `max_h` and `max_v` should be constants that you determine are big enough to hold all of the control points (remember, the initial 3D version has 30 x 3 control points, and there can be up to 6 vertical and 6 horizontal subdivisions).

What is the last dimension for? Why is it 3?

Using this method, you will also need to keep two variables num_h and num_v that indicate the bounds of the "active" region of your control point array. These bounds will be used, for example, when looping through your points in the display routine.

On each subdivsion, use currCP as the source for filling in nextCP. When the subdivision is complete, swap the currCP and nextCP pointers, to allow for future subdivisions.

Computing Normals

To get the normal of a triangle, determine the vectors representing two of its sides. Normalize them. Then, take the cross product. Note that you should use the right-hand rule to make sure you are finding the outward normal. In this assignment, you may use a similar technique to find the normal of a planar quadrilateral by using vectors of its sides.

The normal of a vertex, which ultimately needs to be computed so it can be sent to OpenGL, is the average of the normals of all quadrilaterals, incident to it. In this case, all vertices (except the topmost and bottomost vertices) are part of 4 quads.

Why do we need to know how to find a normal of a triangle / quadrilateral?

Performing Gouraud Shading

You will need to use the OpenGL lighting model. Here is a brief example (from lecture notes) of how to set up lighting in OpenGL,

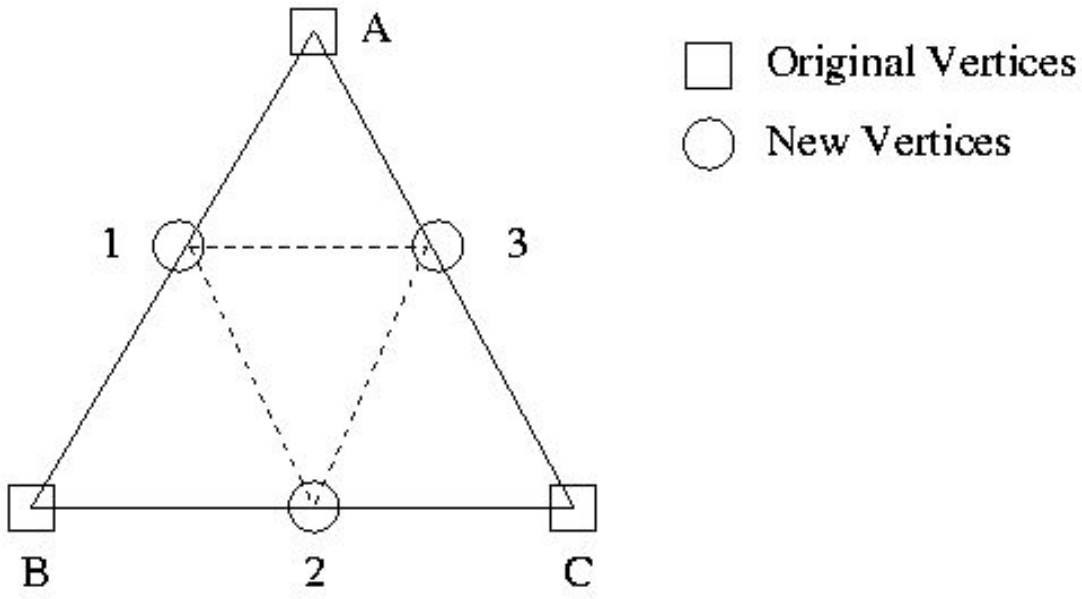
```
void init() {
  // Set up material (surface) property
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  glShadeModel (GL_SMOOTH); // aka Gouraud shading

  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

  // Set up light
  GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
```

```
  GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

  glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
}

void display() {
  glEnable(GL_COLOR_MATERIAL);
  glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
  /* now glColor* changes ambient and diffuse values for the
material */
  glColor3f(0.2, 0.5, 0.8);

  /* draw some objects here, using glColor* */

  glDisable(GL_COLOR_MATERIAL);
}
```

An important thing to keep in mind is that the light and the material all have an ambient, diffuse, and specular color associated with them. A light also has a position, and a material has additionally a shininess factor.

Performing (approximate) Phong Shading

Phong shading can be approximated in OpenGL by splitting polygons into smaller polygons and computing the interior normals.



For Phong lighting, split each quad into a triangle, and then use the midpoints of the triangle edges to further split the triangle into 4 subtriangles (consider making this a subprocedure). Using the normals of A, B, and C, use interpolation to find the normals of 1, 2, and 3. Now display all of these triangles in place of the original quad.

Displaying the Surface

Iterate through the 2-D array of control points and use adjacent elements of the array to display the faces. When you implement shading, create an analogous 2-D x 3 array for the normals at each vertex. After each subdividivision step, compute the normals for each vertex. Then, use both the c.p. array and the normal array in your display routine. Specify the normals by using `glNormal(nx, ny, nz)` before each call to `glVertex()`. This is just like using `glColor()`, where you set the color of all subsequent vertices.