Assignment 1c: Polygons, Texture, Parallel Projection Due: Weds Oct. 16, 2019

In this assignment, you are asked to extend the raytracing program that you wrote for assignment 1b to: 1) render triangles as well as spheres; 2) support smooth shading of triangle meshes, and 3) incorporate texture mapping of spheres and triangles. For 5% extra credit, you can extend your program to allow the specification of a parallel projection, in addition to the default perspective projection that it currently achieves.

Detailed instructions:

1. The first step in extending your program with the ability to handle triangles, as well as spheres, is to enable your program to read basic triangle data from your input file. There are several types of data you will need to handle associated with triangles. The first is the geometric data that specifies the vertex positions, and how they are connected into triangles. Please use the following syntax to do this:

```
# list of vertex positions
   v     x1 y1 z1
   v     x2 y2 z2
   v     x3 y3 z3
   v     x4 y4 z4
   ...

# list of face definitions, consisting of appropriate
     indices into the vertex array, starting at 1 (not 0)
   f     v1 v2 v3
   f     v1 v2 v4
   ...
```

Note that the vertex ordering must start at 1, not at 0.

Later on (in this same assignment), you will need to be able handle additional information associated with each vertex, such as surface normals and texture coordinates.

2. Next, you will need to extend your program with the ability to identify ray/triangle intersection points, as well as ray/sphere intersections. You should start by calculating the point where the ray intersects the plane defined by the triangle's three vertices, and then test to see if the ray/plane intersection point is contained within the triangle.

3. For flat-shaded triangles you would just use a single constant plane normal when evaluating the Phong illumination equation at each ray/triangle intersection point. To support smooth shading across the surface of a triangle, you will need to define an interpolated surface normal direction at the ray/triangle intersection point. This interpolated normal direction should be computed as the weighted sum of the surface normals provided by the user for each of the triangle vertices. You should use the barycentric coordinates α, β, γ as the weights.

a) You will need to allow the user to provide, in the input file, a list of surface normal directions. There could be the same number of surface normal entries as vertex entries, but not always - some models may use smooth shading across some of the triangles a vertex belongs to but not to other triangles the same vertex belongs to. Please use this syntax to specify the vertex normal directions, where **vn** is a keyword, followed by three numbers:

```
# list of surface normal directions
     vn   nx1 ny1 nz1
     vn   nx2 ny2 nz2
     ...
```

b) You will also need to allow the user to specify, for each triangle, an entry into the list of surface normal directions as well as into the vertex arrays.

Your program should be able to handle triangles with either flat or smooth shading, and with or without texture coordinates. For non-textured triangles without smooth shading, your program would not need any information about surface normals at the vertex locations. For smooth shaded triangles where no texture, will be applied, you will need to use this syntax:

```
# list of face definitions, consisting of appropriate
   indices into the vertex and surface normal arrays
   (only)
  f   v1//vn1   v2//vn2  v3//vn3
```

When texture coordinates are also specified for a triangle, they would appear between the consecutive / symbols:

```
# list of face definitions, consisting of appropriate
    indices into the vertex, texture coordinate, and
    surface normal arrays
  f     v1/vt1/vn1  v2/vt2/vn2  v3/vt3/vn3
```

4. Texture mapping. In assignment 1b, surface material properties were defined such that each object possessed a single "intrinsic" color, denoted as $O_{d\lambda}$. In this assignment, we broaden the concept of "intrinsic surface color" from a constant single value to an array of values, allowing the color at each point on a surface to be individually retrieved from a *texture map*.

a) As a first step towards implementing texture mapping, you will need to allow users to specify the name of an ascii .ppm file containing an image that will be applied as a texture to all subsequently-specified surfaces. To keep things simple, you should treat the texture definition as a state variable, similarly to how you are currently handling the association of material properties to objects. This means that if your scene will contain both textured and untextured spheres, or textured and untextured triangles, your input file should specify all of the untextured geometry first. Please use the following syntax to indicate texture images in your input file:

`texture` *`texturefilename.ppm`*

Although the official ascii PPM file format is already fairly simple, I want to make things even simpler for this assignment by only requiring your program to be capable of reading input image data from one specific type of ascii PPM file, in which all of the header information is provided on a single line and comments are not present. It is easy to edit (by hand) any ascii PPM file to conform to this simplified format.

As with material properties, your program should be capable of handling scenes that include more than one texture. There are different ways you could accomplish this. One possibility is to maintain an array of texture images, like your array of material properties, and to store, with each object, an appropriate index into the texture array or a special flag, such as –1, if the object is not textured.

b) As a second step in extending your program to handle texture mapping, you will need to allow users to specify how the texture is applied to each of the objects in your scene.

i) In the case of spheres, to keep things simple, you can hard-code the calculation of appropriate texture coordinates into your program. Thus, the texture coordinate can be computed as needed, at each ray/sphere intersection point.

ii) In the case of triangles, you will need to allow the user to provide an array of 2D texture coordinate values (similar to the array of 3D vertex positions or the array of 3D surface normal directions). Please use this syntax to do that:

```
# list of texture coordinates
     vt    u1  v1
     vt    u2  v2
     ...
```

For each face, you will then need to read, for each vertex, both the vertex position from the array of vertex positions and the texture coordinate value, from the array of texture coordinates. If smooth shading is not being applied, you would use the following syntax:

```
# list of textured triangle definitions,
consisting of appropriate indices into the vertex
and texture coordinate arrays
     f     v1/vt1     v2/vt2     v3/vt3
```

When rendering a textured triangle, you will need to determine the texture coordinate at each ray/triangle intersection point. You should use the barycentric coordinates α, β, γ to compute the interpolated texture coordinate as a weighted sum of the texture coordinates at the triangle vertices.

c) Once your program has computed the texture coordinate values at the ray/object intersection point, it will need to use these texture coordinates to determine the pixel location, in the texture image, from which to retrieve the color to use for $O_{d\lambda}$ when evaluating the Phong illumination equation at that point.

5. If you decide to do the extra credit part: a parallel projection will be achieved when all of the viewing rays (through each point in the image plane) have the same constant direction. You can assume that this constant direction is orthogonal to the image plane. In that case, your input file could allow the user to simply specify a flag indicating if a parallel projection is desired or not. You can use the keyword **`parallel`** for this purpose. If you want to get more fancy, you could allow the user to specify an offset from the normal direction to the viewing plane to use to obtain an oblique parallel projection. In that case, you can still use the keyword **`parallel`** but then follow that either by a 3D offset vector that you could add to the viewing plane normal to obtain a constant oblique direction or by two angles by which you would rotate the viewing plane normal direction vector, in the horizontal and vertical directions, to define an oblique projection direction.

What you should turn in

- all of your source code, including detailed comments and instructions on how to compile your code
- at least one input file containing a scene description that shows off of the various capabilities of your program
- at least one output image showing the results of running your program using your provided scene description file(s) as input