Homework #5

Shaded Rendering II

In this homework you will convert your shaded renderer from last homework to include lighting and material calculations.

Scene Description

The input language used is the OpenInventor language from Homework #3. We will be adding some extra sections for the purpose of this homework, to define materials, point lights, and surface normals (marked in red below).

```
PerspectiveCamera {
   position x y z
   orientation x y z theta
   nearDistance n
   farDistance f
   left l
   right r
   top t
   bottom b
}

# Zero or more lights
PointLight {
   location x y z
   color    r g b
}
```

```
# One or more Separator blocks
Separator {
   # One or more Transform blocks per separator
   Transform {
      translation tx ty tz
      rotation x y z theta
      scaleFactor sx sy sz
   }

   # Zero or one Material block
   Material {
      ambientColor   r g b
      diffuseColor   r g b
      specularColor  r g b
      shininess      p
   }

   # One block per Separator
   Coordinate3 {
      point [
          x0 y0 z0,
          x1 y1 z1,
          ...
         xN yN zN]
   }

   # One block per Separator
   Normal {
      vector [
         x0 y0 z0,
         x1 y1 z1,
         ...
         xN yN zN]
   }

   # One block per Separator
   IndexedFaceSet {
     # Indices of vertices of the faces
     coordIndex [
       face0p0, face0p1, face0p2, face0pn, -1,
       face1p0, face1p1, … face1pn, -1,
       ...
       faceNp0, faceNp1, …. -1]

     # Indices of the normal vectors of the faces
     normalIndex [
       face0normal0, face0normal1, face0normal2, …, -1,
       face1normal0, face1normal1, face1normal2, … , -1,
       ...
       faceNnormal0, …., -1]
   }
}
```

The block "*Normal*" defines a number of 3D normal vectors that will make up the normal vectors at the vertices, and numbering starts at *zero*. These will be indexed into from the "*normalIndex*" section below.

The block "*IndexedFaceSet*" defines the faces of the object by defining the vertices and normal vectors at the vertices.. Indexing is done into the "*Coordinate3*" and "*Normal"* blocks. It has two sub-blocks:

- "*coorIndex*": contains the indices of the vertices of each face, where faces are separated by a value of "**-1**". For example, 0 1 2 3 corresponds to a face whose vertices are the first four points and 1 3 5 6 corresponds to a face consisting of the points 1, 3, 5, and 6.
- "*normalIndex*": contains the indices of the normal vectors at the vertices, again separated by "**-1**".

The block "*PointLight*" defines a point light located at *(x, y, z)* with the given color [*r, g, b*] where $(r, g, b) \in [0, 1]$. For example:

```
PointLight
Location 0 0 1
Color 0 0 1
```

creates a blue light at location (0, 0, 1). We will add another block to define the materials of the objects:

The block "*Material*" defines the materials of the objects in the Separator block. They define the ambient color coefficient (ambientColor) k_a , diffuse color coefficient (diffuseColor) k_d , specular color coefficient (specularColor) k_s , and the specular exponent (shininess) p .

Transforming Normals

Remember that when we are transforming points with a Transform block (which has rotation R, translation T, and scale S), we apply the following transformation $O = TRS$. To transform normal vectors, we use the transpose of the inverse of O without the translation i.e. we use *N=transpose(inverse(RS)*. If all the transformation matrices are 4x4, you can just get the inverse-transpose of the 4x4 transformation *O*, but make sure the vector has a **0** (not a **1**) in its 4th homogeneous coordinate, which has the same effect as canceling the translation i.e. just use *N=transpose(inverse(O)).*

Hidden Surface Removal

You will use your Z-buffer implementation from last homework to remove hidden surfaces.

Lighting

This is the most important part of the homework. Given a surface point position, normal vector, surface material, light position, and camera position, you need to compute the color at that point on the surface. This information is then used for rendering the pixels. The lighting function takes 5 parameters as follows:

1. point in world space
2. normal vector in world space (after transformation from object space by the Transform blocks).
3. Material for the Separator block.
4. Light sources (position in world space and color).
5. Camera position in world space (given in the PerspectiveCamera block).

Please note the following:

- No perspective/camera transformations are applied. Everything is done in world space and only the object transformations are applied (Transform blocks).
- Normals are transformed as above, and make sure to **normalize** the normal vectors after transformation.

Here is some pseudo-code for how the lighting function should work:

```
-- some helper functions
zeroclip(X) = [(x when x > 0.0 else 0.0) for x in X]
oneclip(X)  = [(x when x < 1.0 else 1.0) for x in X]
unit(x) = x / |x| when |x| != 0.0 else 0.0

-- some pseudo-code to do the lighting
lightfunc(n, v, material, lights, camerapos) : (r,g,b) =
do {

    -- let n = surface normal (nx,ny,nz)
    -- let v = point in space (x,y,z)
    -- let lights = [light0, light1, ... ]
    -- let camerapos = (x,y,z)

    scolor = material.specularcolor -- (r,g,b)
    dcolor = material.diffusecolor  -- (r,g,b)
    acolor = material.ambientcolor  -- (r,g,b)
    shiny =  material.shininess     -- (a scalar, an exponent >= 0)

    -- start off the diffuse and specular
    -- at pitch black
    diffuse = [0.0, 0.0, 0.0]
    specular = [0.0, 0.0, 0.0]
    -- copy the ambient color (for the eyelight ex/cred
    -- code, you can change it here to rely on distance
    -- from the camera)
    ambient = acolor

    for l in lights
    do {
        -- get the light position and color from the light
        -- let lx = light position (x,y,z)
        -- let lc = light color (r,g,b)
        lx, lc = l

        -- first calculate the addition this light makes
        -- to the diffuse part
        ddiffuse = zeroclip(lc * (n . unit(lx - v))
        -- accumulate that
        diffuse += ddiffuse

        -- calculate the specular exponent
        k = zeroclip(n . unit(unit(camerapos - v) + unit(lx - v)))
        -- calculate the addition to the specular highlight
        -- k^shiny is a scalar, lc is (r,g,b)
        dspecular = zeroclip(k^shiny * lc)
        -- acumulate that
        specular += dspecular
    }
    -- after working on all the lights, clamp the diffuse value to 1
    d = oneclip(diffuse)
    -- note that d,dcolor,specular and scolor are all (r,g,b).
    -- * here represents component-wise multiplication
    rgb = oneclip(ambient + d*dcolor + specular*scolor)
    return rgb
}
```

Shading Models

You will implement two shading models: flat and Gourard:

- **Flat Shading**: For each triangle, take the average position of the vertices and the average normal vector at the vertices (remember to normalize the normal vector afterwards!) and call the lighting function once to get the color of that triangle.
- **Gourard Shading**: Call the lighting function at each vertex of the triangle i.e. 3 calls to get 3 colors. To get the color at each pixel of the triangle use Barycentric Interpolation from the vertices.

Rendering

You will use your triangle rasterization code from homework #4. You will need to interpolate z-values, in addition to colors (for Gourard shading).

Input and Output

Your program should be called "*shaded*" and should take as input:

- Which shading model to use (0 for flat and 1 for Gourard)
- Resolution in pixels (xRes and yRes)

The scene description is given in the format described above on stdin and your program should produce a PPM image file on stdout.

```
shaded n xRes yRes < input.iv | display -
```

The parser is given in the directory "*shaded*", and it has a sample program called "`shaded.cc`" which can read input from stdin and parse it. Make sure you can compile and run it using the supplied Makefile:

```
make all
./shaded < ../data/cube2.iv
```

Hints

- The flow of your program should go as follows:
  - Parse in the input
  - For each Separator block
    - Transform all the points (vertices) by the Transform block(s) to convert to World Space.
    - Transform all the normal vectors to convert to World Space.
    - For each triangle:
      - Compute colors at the vertices using:
        - Flat Shading:
          - Compute the mean vertex and normal vector
          - Compute the light/color for the face
          - Set the color at the vertices with that color
        - For Gourard shading:
          - Compute the color at each vertex using its normal vector and position
      - Transform the vertices to pixel coordinates (apply camera and viewport transformations). This will also give you the z-values at the vertices.
      - Rasterize the triangle given the pixel coordinates of the vertices, the colors at the vertices, and the z-values at the vertices.
- Make sure the normal vectors are always normalized before calling the lighting function.
- Make sure the normal vectors are transformed by the inverse-transpose of the transformation without the translation (as described above).
- Make sure the vertex positions used in the lighting function are in World Coordinates (no camera/perspective transformations applied).
- Make sure all colors are clipped to be in the interval [0.0, 1.0].
- When converting to PPM, convert colors to [0, 255] by multiplying by 255 and taking the floor i.e. floor(255 * color).

- Build your program in pieces, and test each piece separately.

Acknowledgment

This homework is adapted from CS 171 at Caltech.