Homework #6

Texture Mapping

In this homework you will implement texture mapping in two ways: by specifying texture coordinates for vertices, and by using spherical mapping to generate texture coordinates automatically at vertices.

Scene Description

The input language is augmented with few more sections to define texture files and texture coordinates (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]
}

# Texture image file
Texture2 {
  filename texture.ppm
}

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

# 2D texture coordiantes
TextureCoordinate2 {
   point [
       u0 v0,
       u1 v1,
       ...
       uN vN]
}

# 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]

    # Indices of texture coordinates for the vertices
    textureCoordIndex[
      face0t0, face0t1, … , -1,
      …
      faceNt0, faceNt1, … -1]
  }
}
```

The block "*Texture2*" defines a filename for the texture image. It is in PPM format, and can be easily read in your program.

The block "*TextureCoordinate2*" defines a list of 2D texture coordinates (u, v) pairs, that will be used for defining the texture coordinates of vertices.

The block "*IndexedFaceSet*" defines the faces of the object by defining the vertices and normal vectors at the vertices.. This homework adds the new sublock of "*textureCoordIndex*" which contains the indices of the texture coordinates at the vertices, again separated by "**-1**".

There will be **no** lighting calculations in this homework, just texture calculations that will define the color of every pixel.

Texture Mapping

You will be implementing texture mapping in two different ways:

1. Given Texture

When texture coordinates are given in the input scene description and the mode is set to *GIVEN_TEXTURE*, you will interpolate texture coordinates at each pixel given the texture coordinates at the vertices, and use these to calculate the color at the pixel from the image texture using bi-linear interpolation.

Recall that you will need to interpolate the texture coordinates at the pixels in a perspective correct manner to avoid the nonlinear distortions introduced by the projection. So, given the barycentric coordinates (α, β, γ) of a pixel for a triangle with vertices a, b, c, we calculate its texture coordinates as: $u = \alpha_w u_a + \beta_w u_b + \gamma_w u_c$ and $v = \alpha_w v_a + \beta_w v_b + \gamma_w v_c$ where

$\beta_w = \dfrac{h_a h_c \beta}{h_b h_c + h_c \beta (h_a - h_b) + h_b \gamma (h_a - h_c)}$ and $\gamma_w = \dfrac{h_a h_b \gamma}{h_b h_c + h_c \beta (h_a - h_b) + h_b \gamma (h_a - h_c)}$ and $\alpha_w = 1 - \beta_w - \gamma_w$

and h_a is the fourth coordinate of vertex a after the transformation is done (before homogenization). See **Chapter 11.3** of the textbook for more details.

2. Auto Texture

When the texture coordinates are not given in the input scene description, or when the mode is set to *AUTO_TEXTURE*, you will compute the texture coordinate at each vertex automatically and use the texture file called "**`checker.ppm`**". The textures are computed assuming we are using a Spherical mapping i.e. assuming the texture is on a big sphere surrounding the object.

The automatic texture coordinates at a vertex p are computed as follows:

1. First find the centroid of the object in object space. This can be done by computing the bounding rectangle of the object, which is computed by finding the minimum/maximum in every direction (x, y, and z). Then the centroid is $c=\left(\frac{x_{min}+x_{max}}{2}, \frac{y_{min}+y_{max}}{2}, \frac{z_{min}+z_{max}}{2}\right)$
2. For every vertex p, compute the unit vector from c to p as $k=\frac{c-p}{|c-p|}$
3. Compute the u and v coordinates of the point p as: $u=0.5+\frac{\text{atan2}(k_x, k_z)}{2\pi}$ and $v=\frac{\text{acos}(k_y)}{\pi}$
   where k_x is the x-component of k, and similarly for k_y and k_z.

Once you have texture coordinates for the vertices, you can interpolate them as above to obtain texture coordinates for the pixels of the rasterized triangles.

Note: the formulas above for u and v assume the y-axis is up (not the z-axis). So these calculations should be done in the camera coordinates. If you want to do them in world coordinates instead, then you should use the these formulas $u=0.5+\frac{\text{atan2}(k_y, k_x)}{2\pi}$ and $v=\frac{\text{acos}(k_z)}{\pi}$

Bilinear Interpolation

Given the (u, v) texture coordinates of a pixel, you need to compute the color of that pixel from the texture. To do this, you will use bilinear interpolation, which is a generalization of linear interpolation. The pixel numbering of an image starts at the top-left and goes through the right and the bottom.

You can do this as follows:

1. Compute the base pixel $i=\lfloor n_x u \rfloor$ and $j=\lfloor n_y v \rfloor$
2. Compute the offsets from (u, v) to the base pixel $u'=n_x u-i$ and $v'=n_y v-j$
3. Interpolate from the colors of pixels $(i, j), (i+1, j), (i, j+1), (i+1, j+1)$:

$$\begin{aligned} c(u,v)=&(1-u')(1-v')c(i,j)+\\ &u'(1-v')c(i+1,j)+\\ &(1-u')v'c(i,j+1)+\\ &u'v'c(i+1,j+1) \end{aligned}$$

See **Chapter 11.2** of the textbook for more details.

Input and Output

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

- Which texture mode to use (0 for GIVEN_TEXTURE and 1 for AUTO_TEXTURE)
- Resolution in pixels (xRes and yRes)

Note that for files which have no texture defined, the texture is computed using the AUTO_TEXTURE even if the input mode is **0**.

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

```
texture mode xRes yRes < input.iv | display -
```

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

```
make all
./texture 0 256 256 < ../data/geotex.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.
    - If you are in mode AUTO_TEXTURE, then compute the texture coordinates for the vertices (either in camera coordinates or in world coordinates, see above).
    - For each triangle:
      - 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, texture coordinates at the vertices, and the z-values at the vertices.
- Ignore all lighting calculations for this homework (point lights and surface normals).
- Build your program in pieces, and test each piece separately.

Acknowledgment

This homework is adapted from CS 171 at Caltech.