INF01009 – Computação Gráfica

Manuel M. Oliveira

Programming Assignment 3

Extending Close2GL to Support Rasterization

Total of Points of the Assignment: 200

Extending Close2GL to Support Rasterization

Your software implementation of the graphics pipeline can transform, shade and project triangular models. For the final rendering, however, it still relies on OpenGL to rasterize the projected polygons. The goal of this assignment is to extend Close2GL to implement rasterization.

Your new viewer should support the following features:

a) Rasterization of the projected triangles, given the coordinates and color associated with their vertices (30 points);
b) Phong Lighting Model (20 points);
c) FLAT and GOURAUD shading, wireframe and point rendering (30 points);
d) Interactive change of the Ambient, Diffuse and Specular properties of the surface, and change of the light source color (10 points);
e) Handling of visibility by implementing z-buffering (25 points);
f) Perspectively correct Interpolation and texture mapping (40 points);
g) Texture filtering as: *nearest neighbors*, *bilinear re-sampling*, and *mip-mapping* (25 points);
h) Resizing the Close2GL window (20 points).

In order to display the result of your rendering, you are allowed to use the following three OpenGL/GLUT commands: glRasterPos2i(0, 0), glDrawPixels and glutSwapBuffers. Each command should be used at most once per displayed frame, and glRasterPos2i(0, 0), should be issued using (0,0) as parameters.

For this assignment, use the same test files from the previous assignments. You can modify the ***cube.in*** file to store texture coordinates. For the cow model (***cow_up.in***), you should use some technique for automatic texture coordinate generation. As always, do not forget to write a two-page report about your implementation discussing the usual questions described in the *Guidelines for Submitting the Assignments*.

Layout of the new input file

Object name = <obj_name>
# triangles = <num_tri>
Material count = <material_count>
ambient color <r_a> <g_a> <b_a>
diffuse color <r_d> <g_d> <b_d>
specular color <r_s> <g_s> <b_s>
material shine <shine_coeff>
Texture = <YES/NO>
-- 3*[pos(x,y,z) normal(x,y,z) color_index [texture coordinates(s,t)]] face_normal(x,y,z)
v0 <x> <y> <z> <Nx> <Ny> <Nz> <material_index> [<s> <t>]
v1 <x> <y> <z> <Nx> <Ny> <Nz> <material_index> [<s> <t>]
v2 <x> <y> <z> <Nx> <Ny> <Nz> <material_index> [<s> <t>]
face normal <FNx> <FNy> <FNz>

Example 1: A file describing a textured triangle

```
Object name = TRIANGLE_TEXTURE
# triangles = 1
Material count = 1
ambient color 0.5 0.5 0.5
diffuse color 0.5 0.5 0.5
specular color 0.5 0.5 0.5
material shine 10.0
Texture = YES
-- 3*[pos(x,y,z) normal(x,y,z) color_index text_coord] face_normal(x,y,z)
v0 -1.0 -1.0 -2.0 0.0 0.0 1.0 0 0.0 0.0
v1  1.0 -1.0 -2.0 0.0 0.0 1.0 0 1.0 0.0
v2  1.0  1.0 -2.0 0.0 0.0 1.0 0 1.0 1.0
face normal 0.0 0.0 1.0
```

Example 2: A file describing a non-textured triangle

```
Object name = TRIANGLE_NO_TEXTURE
# triangles = 1
Material count = 1
ambient color 0.5 0.5 0.5
diffuse color 0.5 0.5 0.5
specular color 0.5 0.5 0.5
material shine 10.0
Texture = NO
-- 3*[pos(x,y,z) normal(x,y,z) color_index text_coord] face_normal(x,y,z)
v0 -1.0 -1.0 -2.0 0.0 0.0 1.0 0
v1  1.0 -1.0 -2.0 0.0 0.0 1.0 0
v2  1.0  1.0 -2.0 0.0 0.0 1.0 0
face normal 0.0 0.0 1.0
```

Tips on How to Complete the Assignment

1) Extend your data structures to support both a color and a depth (z) buffer with the dimensions of the Close2GL window. The color buffer should store R, G, B and A values for each pixel. The z-buffer should accommodate a float pixel. Remember that the user might want to resize the Close2GL and your implementation should handle this situation.

2) Write a function to rasterize triangles. Your rasterization function will write to the color buffer and read and write the depth buffer. As you rasterize a polygon, compare the depth of the current pixel covered by the polygon with the value stored for that pixel in the depth buffer. If the current value should replace the previous one, update both buffers. Do not forget to implement perspective correct interpolation.

3) You should pay special attention to special cases, such as the ones involving polygons having horizontal edges. Also, be careful with all kinds of conversions between floats or doubles to integers, since this might cause you problems.

4) Once you have rasterized all triangles that are visible in the current frame, load the resulting image onto the Close2GL window using the command glDrawPixels (see the OpenGL red book) and, don't forget to swap buffers.

5) In order to simplify your debugging and the grading of the assignments, replace the command gluOrtho2D(0.0, 1.0, 0.0, 1.0) in the Close2GL reshape function with gluOrtho2D(0.0, w, 0.0, h). w and h are the parameters of the reshape function. This way, the coordinates of the projected vertices will be expressed in pixel units. Don't forget to change your viewport matrix to reflect such a change.

Here are some observations that will make your implementation of Close2GL really simple.

Reading a image file (.jpg)

You can use a simple API to read a JPEG image file to be used as texture. To use this API, you will need the following files (available from the Course web page):

i) ***jpeg.li*b**, a static library for reading and writing JPEG files;
ii) ***jconfig.h*** and ***jmorecfg.*h**, these are include files to be used with the library;
iii) ***jpeg_api32.li*b**, an API for the jpeg.lib that I prepared to make your life easier;
iv) ***jpeg_api.*h**, a header file for the API. This file contains an example of how to use the API.

Using Textures in OpenGL

Study (not only read) Chapter 9 (Texture Mapping) of the OpenGL Red Book.

If the object is textured, after you setup your OpenGL window, you can use the following commands:

```
if (<object_is_textured>) {
   glEnable(GL_COLOR_MATERIAL);
   read_texture_image_and_set OpenGL_texture environment(<your parameters>);
   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
 }
```

The template of the function below will setup a mipmap pyramid for you to use with OpenGL. But if you do not understand the meaning of these commands, you will probably not be able to change OpenGL state machine to render using other resampling strategies such as nearest neighbors and bilinear.

```
function read_texture_image_and_set OpenGL_texture environment(<your parameters>)
{
 <read the input texture file here>

 glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
 glGenTextures(1, &texName);

 glBindTexture(GL_TEXTURE_2D, texName);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
//
//   setting mip map pyramid
//
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, <tex_width>, <tex_height>,, 0,
        GL_RGB, GL_UNSIGNED_BYTE, tex.texture);
 gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, <tex_width>, <tex_height>,
        GL_RGB, GL_UNSIGNED_BYTE, <texture>);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
}
```

Before using a texture, you need to enable texturing and bind it. You can use the following commands to do so (remember that in order to switch between renderings with and without texture you will need to enable and disable the texture environment appropriately).

glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, <texName>);

Implementing Texture Mapping for Close2GL

a) *Without perspective correct interpolation*: During rasterization just bilinear interpolate the texture coordinates associated with the projected vertices and use them to resample the texture.

b) *With perspective correct interpolation*: Compute *1/w* and (*s/w*, *t/w*) for each projected vertex, where *s* and *t* are the texture coordinates of the vertex. During rasterization, bilinear interpolate the values of *1/w*, *s/w* and *t/w*. For each pixel, compute *s'* = interpolated(*s/w*)/ interpolated(*1/w*) and *t'* = interpolated(*t/w*)/ interpolated(*1/w*). Use (*s', t'*) for texture resampling.

Building a Mip Map Pyramid for Close2GL

Starting for the original texture (it is assumed to have 2^nx 2^n texels), create the next level of the pyramid by averaging groups of 2x2 texels and storing the result as the corresponding pixel at the next level.

Other Observations:

When you interpolate values of s, t in [0,1], due to limited precision you may end up with values slightly smaller than zero or slightly bigger than one. Make sure you clamp such values to the interval [0,1] before you resample the texture.

Good luck.