Assignment 1. OpenGL Basics

Total of Points of the Assignment: 15

In this first programming assignment you will be asked to write a program to render polygonal meshes. For this, you will need to refresh your memory about some important features of a graphics application and their implementation using OpenGL and GLU commands. For example, you will remember how to:

- Specify a virtual camera with arbitrary position and orientation;
- Render an object using different kinds of primitives, such as points, wireframe and solid polygons;
- Perform backface culling to reduce the number of primitives actually drawn;
- Change the field of view of the camera to achieve some zooming effects.

Besides supporting all the features listed above, your program should be able to:

(a) Read and display arbitrary geometric models represented as triangle meshes. These objects are described in text files whose layout will be presented next. Once you read the objects, these should be displayed in the center of the window **(3 points)**;
(b) Translate the virtual camera along its own axes (u, v, n) (not along the world coordinate system axes) **(2 point)**;
(c) Rotate the virtual camera along its own axes **(2 point)**;
(d) Reset the camera to its original position (*i.e.*, object centered inside the window) **(1 point)**;
(e) Support for rendering objects whose polygon vertices were defined using CW (clockwise) and CCW (counter clockwise) orientation – this will affect the behavior of the backface culling procedure **(1.5 point)**;
(f) Support for changing the values of the *near* and *far* clipping planes **(1.5 point)**;
(g) Support for interactive change of colors (R, G, B) for the models, making sure that the color change is apparent under all rendering modes (without lighting for this assignment). A single RGB color is assigned to all triangles of the model **(1 point)**;
(h) Support for rendering the object using different kinds of primitives, such as points, wireframe and solid polygons **(1.5 point)**;
(i) Support for reading a new model file through the user interface **(1.5 point)**.

Submit your source code and a self-contained executable file. You will need to demonstrate your program to the instructor (or the TA) during office hours or lab sessions.

Here is the **Layout of the 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>
-- 3*[pos(x,y,z) normal(x,y,z) color_index] face_normal(x,y,z)
v0 <x> <y> <z> <Nx> <Ny> <Nz> <material_index>
v1 <x> <y> <z> <Nx> <Ny> <Nz> <material_index>
v2 <x> <y> <z> <Nx> <Ny> <Nz> <material_index>
face normal <FNx> <FNy> <FNz>

Example

```
Object name = SQUARE
# triangles = 2
Material count = 1
ambient color 0.694 0.580 0.459
diffuse color 0.992 0.941 0.863
specular color 1.000 1.000 1.000
material shine 0.250
-- 3*[pos(x,y,z) normal(x,y,z) color_index] 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
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
```

Reading Input Files

This code is showed for the purposes of illustration. Essentially, you can reuse the *fscanf* sequence and its formats, but you will have to adapt the rest of the code to fit your own data structures.

```
void <tri model>::>.<your function for reading an input file>(char *FileName)
{
   <3d vector>   ambient[MAX_MATERIAL_COUNT],
                 diffuse[MAX_MATERIAL_COUNT],
                 specular[MAX_MATERIAL_COUNT];
   float shine[MAX_MATERIAL_COUNT];
   int material_count, color_index[3], i;
   char ch;

   FILE* fp = fopen(FileName,"r");
   if (fp==NULL) { printf("ERROR: unable to open TriObj [%s]!\n",FileName); exit(1); }

   fscanf(fp, "%c", &ch);
   while(ch!= '\n') // skip the first line – object's name
```

```
        fscanf(fp, "%c", &ch);
//
    fscanf(fp,"# triangles = %d\n", &NumTris); // read # of triangles
    fscanf(fp,"Material count = %d\n", &material_count); // read material count
//
    for (i=0; i<material_count; i++) {
        fscanf(fp, "ambient color %f %f %f\n", &(ambient[i].x), &(ambient[i].y), &(ambient[i].z));
        fscanf(fp, "diffuse color %f %f %f\n", &(diffuse[i].x), &(diffuse[i].y), &(diffuse[i].z));
        fscanf(fp, "specular color %f %f %f\n", &(specular[i].x), &(specular[i].y), &(specular[i].z));
        fscanf(fp, "material shine %f\n", &(shine[i]));
    }
//
    fscanf(fp, "%c", &ch);
    while(ch!= '\n') // skip documentation line
    fscanf(fp, "%c", &ch);
//
// allocate triangles for tri model
//
    printf ("Reading in %s (%d triangles). . .\n", FileName, NumTris);
    Tris = new <triangle data struct> [NumTris];
//
    for (i=0; i<NumTris; i++) // read triangles
    {
            fscanf(fp, "v0 %f %f %f %f %f %f %d\n",
                        &(Tris[i].v0.x), &(Tris[i].v0.y), &(Tris[i].v0.z),
                        &(Tris[i].normal[0].x), &(Tris[i]. normal [0].y), &(Tris[i]. normal [0].z),
                        &(color_index[0]));
            fscanf(fp, "v1 %f %f %f %f %f %f %d\n",
                        &(Tris[i].v1.x), &(Tris[i].v1.y), &(Tris[i].v1.z),
                        &(Tris[i].Norm[1].x), &(Tris[i].Norm[1].y), &(Tris[i].Norm[1].z),
                        &(color_index[1]));
            fscanf(fp, "v2 %f %f %f %f %f %f %d\n",
                        &(Tris[i].v2.x), &(Tris[i].v2.y), &(Tris[i].v2.z),
                        &(Tris[i].Norm[2].x), &(Tris[i].Norm[2].y), &(Tris[i].Norm[2].z),
                        &(color_index[2]));
            fscanf(fp, "face normal %f %f %f\n", &(Tris[i].face_normal.x), &(Tris[i].face_normal.y),
                        &(Tris[i].face_normal.z));
            //
            Tris[i].Color[0] = (unsigned char)(int)(255*(diffuse[color_index[0]].x));
            Tris[i].Color[1] = (unsigned char)(int)(255*(diffuse[color_index[0]].y));
            Tris[i].Color[2] = (unsigned char)(int)(255*(diffuse[color_index[0]].z));
    }
    fclose(fp);
}
```

Tips on How to Complete the Assignment

Rendering the object in the center of the window

In order to render the object in the center of the window, you will need to do some calculations. For instance, as you read the object description from the file, given the vertices' coordinates in WCS, the object might be behind the camera or outside of its field of view. It could also be too big and be only partially inside the view frustum. You will then need to reposition the camera in order to make sure the object will be completely visible and centered in the window. In order to accomplish this, you will need to identify the range (minimum and maximum coordinates) of the object in both X, Y and Z. With these values at hand, you can then imagine a bounding box (a parallelepiped) for the object. In order for the object to appear centered, the x and y coordinates for the position of the camera can be computed as the average of the corresponding min and max values. Note, however, that this might not be enough if the object is too big or if the field of view is too small. In these cases, the object might be partially outside of the view frustum. You should then use your trigonometric skills to figure out what should be the z coordinate of the camera so that the object is completely visible and as close as possible.

Rotating the Camera

In order to perform camera rotation (translation) around (along) the camera's axes, you will need to keep track of the vectors that define the camera coordinate system. As you start your program, let these vectors be: u = (1, 0, 0), v = (0, 1, 0), and n = (0, 0, -1).

As we rotate the camera, we hange the vectors that define the CCS (note that this does not happen when we perform just a translation). Thus, we need to recalculate them. Fortunately, this is not difficult and can be accomplished using the same basic ideas used to derive the rotations of points in 2D and 3D.

After you have calculated the new vectors that define the CCS you will have all information you need to call the *gluLookAt()* command with the appropriate parameters.

Changing the values of the near and far clipping planes

This can be accomplished with *gluPerspective()*. Don't forget to select and initialize the projection matrix before you call *gluPerspective()* and to set the current matrix back to the model view matrix after you are done.

Initialize Znear = 1.0f and Zfar = 3000.0 and play with these values. What happens when Znear becomes very close to zero?

Selecting the orientation (CW, CCW) for the front facing polygons

Your program should support change of the orientation interactively. For this, use the command *glFrontFace*().