INF01009 – Computação Gráfica
Manuel M. Oliveira
Programming Assignment 2

The Graphics Pipeline

Total of Points of the Assignment: 200

By successfully completing the previous programming assignment, you learnt how to perform many useful functions using OpenGL, including:

- Display arbitrary geometric models represented as triangle meshes;
- Translate the virtual camera along its own axes (u, v, n);
- Rotate the virtual camera along its own axes;
- Support for rendering objects whose polygon vertices were defined using CW (clockwise) and CCW (counter clockwise) orientation;
- Implement a graphical user interface (GUI).

In addition to this, you also experimented with changes in color and in the near and far clipping planes.

The goals of this assignment are to teach you how to implement some of the major steps of the geometry-based rendering pipeline. By successfully completing it, you should feel very comfortable with concepts such as geometrical transformations, perspective projection, mapping to viewport, changes in the field of view, clipping and backface culling. To achieve this, you will implement a substantial portion of OpenGL in software, which I will refer to as Close2GL. Your program should allow the visualization, from arbitrary viewpoints, of 3D objects represented as triangle meshes whose vertices are defined with respect to a world coordinate system.

Description of the Assignment

Your program should take as input a text file containing the description of the model and render it. In order to render a model, you need perform the following sequence of operations:

1. Transform vertex coordinates from the world coordinate system (WCS) to the camera coordinate system (CCS).
2. Apply a perspective transformation that maps the coordinates from the CCS to the screen coordinate system (SCS).
3. Perform clipping.
4. Perform the perspective division.
5. Map the result to the viewport.
6. Draw the triangles.

The model should be rendered in two separate windows. One of the windows will show the rendering produced by a straight OpenGL code (essentially, this is what you did for your first programming assignment). The other window will show an equivalent rendering produced by your software implementation (Close2GL). The two windows should show the same result, *i.e.*, as you explore the scene with a virtual camera, the renderings produced by both OpenGL and Close2GL should match. While for the OpenGL renderer you can use any OpenGL function, for Close2GL you must use your own functions to transform vertices from the world coordinate system into pixel coordinates. Once you have performed such transformations, **you are allowed to use OpenGL 2D functions to draw points, lines or polygons only**.

Your viewer should provide at least the following features:

a) Render the models using points, wireframe and solid representations.
b) Support translations and rotations of the virtual camera. Translations and rotations should be defined with respect to the coordinate system of the camera. Thus, for instance, a translation in Z should move the camera forward (backward).
c) Render the objects at the center of the windows.
d) Support changes in both the horizontal and vertical fields of view.
e) Support backface culling.
f) Support some simple clipping.

You should write a two-page report describing your experience, problems you faced, how you solved them and, in retrospect, what you would do differently. Also, remember that writing clean code is important for reusability as well as for easy understanding from other people. This project involves a fair amount of complexity and I advise you to design your project before your start coding. This will allow you to clearly identify the necessary modules and the relationship among them. The use of C++ can help you produce a clean implementation of your design. Without clear and insightful comments it will be difficulty to understand your program and this should affect your score adversely.

For testing your implementation, use the same input files used for Assignment 1.

For this assignment, you don’t need to shade the renderings produced by Close2GL, although you should use the colors defined through the user interface.

Evaluation Criteria

1) (175 points) **Close2GL** part supports:

(10 points) Points (   )
(10 points) Wireframe (   )
(10 points) Solid (   )
(15 points) Translation (   )
(20 points) Supports translation while keeping fixed look at point?
(30 points) Rotation (   )
(25 points) Backface culling (   )
(15 points) Change Hfov (   )
(15 points) Change Vfov (   )
(10 points) Simple clipping (   )
(15 points) Works after windows have been resized? (   )

2) (25 points) Renderings from OpenGL and Close2GL are Synchronized? (   )

Tips on how to complete the assignment

Here are some observations that will make your implementation of Close2GL really simple. But in order for you to be able to take the most from these tips, **you should study chapter 7 of Hill's book (Three-Dimensional Viewing) and the Lecture Notes on *Geometrical Transformations*, *Change of Coordinate Systems* and *Planar Projections***.

a) You will need to define a camera object (a good idea is to create a camera class) with at least the following attributes:
   i. *Position* (x, y, z). This defines the position of the camera in 3D. The camera position is also known as its center of projection (COP), viewpoint or eye point.
   ii. A coordinate system associated with the camera (CCS). This will define the camera's orientation in 3D. In our general derivation of change of basis, we represented the CCS using three 3D vectors: U_1, U_2 and U_3. In order to use the same notation used in Hill's book, we can rename these vectors as $u = U_1$, $v = U_2$ and $n = U_3$.
   iii. Horizontal and vertical fields of view (hfov and vfov).
   iv. Distances associated with the near and far clipping planes.

   Initialize your camera with the following parameters (remember that in order for the rendering of your program to match OpenGl's, you should use a right-hand coordinate system):

   Position = (0, 0, 0);
   u = (1, 0, 0), v = (0, 1, 0), n = (0, 0, 1)
   hfov = vfov = 60 degrees.

b) All transformations and projections are performed via matrix multiplication. Since matrices play such an important role in this project, you should spend some time designing a good and flexible matrix class. Remember that since we are using homogeneous representations of points in 3-D, you will need 4 by 4 matrices. Please, write your own code instead of copying a matrix library from the internet.

   The matrix class should provide at least the following methods:

   i) Multiplication between two matrices
   ii) Multiplication between a matrix and a vector
   iii) Set a ModelView Matrix (see Slide Set on Change of Coordinate Systems, or matrix 7.2 on page 365 of Hill's book. Note that there is a typo in the book and the element v[4][4] of matrix 7.2 should be "1" instead of "0")
   iv) Set a Projection Matrix (see Slide Set on Planar Projections, or matrix R in page 674 of the OpenGL red book, or matrix 7.13 in page 385 of Hill's book)
   v) Set a Viewport Transformation Matrix (see details below)

   With these methods you are ready to complete the assignment.

c) There are three important matrices involved in the transformation from world coordinates to pixel coordinates:

   i) **Model/View Matrix (M)**: It represents the change of coordinate system (from WCS to CCS). If $c = (c_x, c_y, c_z)$ is the position or center of projection the camera and the CCS is defined by the vectors u, v, and n, then

$$M = \begin{bmatrix} u_x & u_y & u_z & dx \\ v_x & v_y & v_z & dy \\ n_x & n_y & n_z & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where dx = -c dot u, dy = -c dot v and dz = -c dot n.

ii) **Projection Matrix (P)**: Projects points defined with respect to the CCS onto the image plane of the normalized perspective view volume.

iii) **Viewport Matrix (V)**: Maps coordinates from the image plane of the normalized perspective view volume to the actual pixel coordinates in the window (on the computer screen) where the final rendering will be displayed. This matrix implements translation in X and Y, and scaling in X and Y. Why?

d) Once you have these three important matrices, the rendering of the scene/model can be described by:

i) Compute PM = P*M (why not M*P?). This composite matrix embodies both the geometric and projection transformations.

ii) For each triangle (of the model) with vertices v1, v2 and v3, compute v1', v2' and v3', where vi' = PM*vi. What did you just do?

- vi' is expressed in homogeneous coordinates. However, before you perform the perspective division (i.e., force all points to be on the hyperplane w=1 by dividing all coordinates by w), you need to clip vertices that are outside of the normalized perspective view volume. Why? (hint: what would happen if w=0? What if w<0?).

- Clipping against the normalized perspective view volume is trivial. Points inside the view volume are defined by abs(x), abs(y), abs(z) ≤ abs(w). Why?
(to simplify your task, you can clip the whole triangle if at least one of its vertices fall outside the view volume).

- Now that you are sure that no division by zero will happen, nor points behind the center of projection will be mapped back onto the image plane, perform the division by w. The resulting x" (= vi'_x/vi'_w) and y" (= vi'_y/vi'_w) coordinates after the division are the projection of the vertex into the image plane of the normalized perspective view volume.

- Map the (x",y") coordinates of the projected vertices to the desired window coordinates using the Viewport Matrix: vi"'= V*vi".

iii) Draw the 2-D triangle defined by v1"', v2"' and v3"' specified in pixel coordinates and you are done.

e) Initializing a Viewport Matrix.

i) Suppose you have initialized the 2-D window to be used by Close2GL using the command
gluOrtho2D(lv, rv, bv, tv) (What do these parameters represent?)

and suppose (left, bottom) and (right, top) are the corners of the image plane of the normalized perspective view volume. At this point you should know what these (constant) values are, otherwise you did not understand what the normalized perspective view volume is. In such a case, you should redo your readings. After you understand it, you can proceed.

The Viewport matrix should:
- scale the x coordinate by (rv-lv)/(right-left)
- scale the y coordinate by (tv-bv)/(top-bottom)
- translate the x coordinate by (lv+rv)/(right-left)
- translate the y coordinate by (bv+tv)/(top-bottom)

Explain what these operations represent.

This is all you need to complete this assignment, besides a few OpenGL calls to implement its OpenGL portion. Please, re-read these tips carefully and make sure you understand what needs to be done. If things are still unclear, you probably haven't understood the *visualization problem* I stated (and repeated for several lectures) and haven't understood the logic behind the rendering pipeline. If that is the case, you need to go back to your notes and make sure you understand them.

Rendering the object in the center of the windows

In order to render the object in the center of the windows, you will need to perform some work. 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.

Performing the "expected" translation

Camera movements should be defined with respect to the CCS independent of its orientation with respect to the WCS. Otherwise, the movement will appear non-intuitive. Translation implies change of the camera position. Thus, for instance, in order to produce an intuitive forward translation by k units, one can compute the new position as pos = pos + k*(-n), where *n* is the axis in the CCS associated with the viewing direction. Notice that since we are adopting a right-hand coordinate system convention, we need to use a minus sign before *n*. The left/right and up/down translations are similar. See section *Sliding the Camera* on page 368 of Hill's book

Rotating the Camera

As we rotate the camera, we change 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 not difficulty and can be accomplished using the same basic ideas used to derive the rotations of points in 2 and 3D. See section *Rotating the Camera* on page 368 of Hill's book.

Backface Culling

In order to perform backface culling, you should read pages 57, 58 and top of page 59 of the OpenGL red book. In order to implement backface culling in Close2GL you will need the *advanced* section at the bottom of page 58.

Other Tips

How to Render to Multiple Windows using OpenGL

You can create as many windows as you want using the command *glutCreateWindow*. This function returns a unique integer identifier for each created window. Before rendering to a window, you should explicitly select a specific one using the command *glutSetWindow*. It is a good idea to save the identifiers returned by *glutCreateWindow* in an array and use some defined constants in order to improve readability. For instance, you can define the following:

```
#include <GL/glut.h>

#define OPENGL_WINDOW     0
#define CLOSE2GL_WINDOW 1

int win_id[2];

float Znear = 0.1,                      // near clipping plane
      Zfar  = 10.0,                     // far clipping plane
      Hfov  = 60.0,                     // horizontal and
      Vfov  = 60.0;                     // vertical field of view
```

and use it with the following template of main function. Notice that I will be using C++ notation.

```
//*******************************************************************************
//
//  main function for controlling the implementation of your assignment
//
//*******************************************************************************

int main(int argc, char *argv[])
{
//
//   load the triangle model by calling your function that reads the triangle model description file
//   passed as the first argument to your program
//
  <tri model>.<your function for reading an input file>(argv[1]);
//
//   Initialize two windows, one for rendering OpenGL and another one for rendering Close2GL.
//   Make sure you understand the meaning of the parameters used with each command. In particular,
// make sure you understand the meaning of the parameters used with glutInitDisplayMode and understand
// its relationship to the glutSwapBuffers command. You can find detailed explanations about them in the
// OpenGL red book
//
//   First, initialize OpenGL window
//
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowPosition(0, 0);
  glutInitWindowSize(<win width>, <win height>);
  win_id[OPENGL_WINDOW] =  glutCreateWindow("OpenGL");
  glutDisplayFunc(<name of your function for rendering using OpenGL >);
  glutReshapeFunc(openglReshape);
  glutMouseFunc(<mouse control function>);
  glutMotionFunc(<mouse motion control function>);
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glClearColor(0.0, 0.0, 0.0, 0);
  glEnable(GL_DEPTH_TEST);
//
//   Now, initialize Close2GL window
```

```
//
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowPosition(450, 0);
  glutInitWindowSize((<win width>, <win height>);
  win_id[CLOSE2GL_WINDOW] =  glutCreateWindow("Close2GL");
  glutDisplayFunc(<name of your function for rendering using Close2GL >);
  glutReshapeFunc(close2glReshape)
  glutMouseFunc(<mouse control function>);
  glutMotionFunc(<mouse motion control function>);
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_ALPHA);
  glClearColor(0.0, 0.0, 0.0, 0);
  glEnable(GL_DEPTH_TEST);
//
//    call your function for initializing your user interface
//
<your function for initializing your user interface>;
//
//    call glutMainLoop()
//
  glutMainLoop();
  return 0;
}
```

1) Supporting Functions

```
//******************************************************************************
//
//  opengl reshape function
//
//******************************************************************************
void openglReshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(VFov, Hfov/Vfov, Znear, Zfar);
  glMatrixMode(GL_MODELVIEW);
}

//******************************************************************************
//
//  close2gl reshape function
//
//******************************************************************************
void close2glReshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, 1.0, 0.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
//
//   create a projection matrix for your close2gl implementation using the initial parameters
//   yfov, aspect ratio, Znear, Zfar. Remember that the projection matrix is defined in terms of
//   left, right, bottom, top, near and far. So, you will need to recover left, right, bottom and top
//   from near and from Hfov and Vfov in your function.
//
<close2gl projection matrix>.<your function to create a projection matrix>(Vfov, Hfov/Vfov, Znear, Zfar);
//
//   create a viewport matrix for your close2gl implementation. Use the same (x,y) range specified in glOrtho
//   above. See the tips on how to create a viewport matrix in the document that describes the programming
//   assignment.
//
<close2gl viewport matrix>.<your function to create a viewport matrix>(0.0, 1.0, 0.0, 1.0);
}


//*******************************************************************************
//  opengl display function. <render mode> specifies whether you
//  should render using points, wireframe or solid mode. <clip mode>
//  says whether you should clip or not vertices that fall outside of the
//  view frustum.
//
//******************************************************************************
void openglDisplay(void)
{
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity(); // initialize ModelView Matrix as the Identity Matrix

  gluLookAt(<put here the parameters you can extract from your camera>);
 <tri model>.<your opengl renderer> (<render mode>, <clip mode>);
}

//*******************************************************************************
//  close2gl display function. <render mode> specifies whether you
//  should render using points, wireframe or solid mode. <clip mode>
//  says whether you should clip or not vertices that fall outside of the
//  view frustum.
//
//*******************************************************************************
void close2glDisplay(void)
{
<set a model view matrix based on the most current camera parameters to be used in your rendering>
 <tri model>.<your close2gl renderer> (<render mode>, <clip mode>);
}
```

Synchronizing the Renderings in all Windows

A simple way to guarantee synchronized renderings in all windows is to loop through all windows and call *glutPostRedisplay* for each one of them. The next code fragment illustrates this for the case of changing rendering mode and for changing of field of view.

```
switch (Option) {
   case RENDER_MODE:
//
//      redraw all windows
//
        for (i=0; i<MAX_WINDOWS; i++) {
              glutSetWindow(win_id[i]);
              glutPostRedisplay();
        }
        break;
   case H_FOV:
   case V_FOV:
//
//     code for reinitializing OpenGL projection matrix
//
       glutSetWindow(OPENGL_WINDOW);
       glMatrixMode(GL_PROJECTION);
       glLoadIdentity();
       gluPerspective(Vfov, Hfov/Vfov, Znear, Zfar);
       glMatrixMode(GL_MODELVIEW);
//
//          reinitialize Close2GL's projection matrix and redraw all windows
//
    <close2gl projection matrix>.<your function to create a projection matrix>(Vfov, Hfov/Vfov, Znear, Zfar);
    glutSetWindow(CLOSE2GL_WINDOW);
    for (i=0; i<MAX_WINDOWS; i++) {
        glutSetWindow(win_id[i]);
        glutPostRedisplay();
    }
    break;
}
```

Good luck.