ICT289: Computer Graphics Principles and Applications

Semester 1 2021

Practical Labs of Weeks 3 and 4:

Introduction to Computer Graphics and OpenGL

The main objectives of these exercises are to:

- Understand some basic mathematics used in Computer Graphics,
- Create and render (visualize) simple shapes in 2D and 3D using OpenGL,
- Start thinking of what you would like to do for the graphics project,
- Find out about team work issues.

These exercises will not be assessed. However, **you should note that even though an exercise is not assessed, not attempting the exercise would make it very difficult for you to understand subsequent material.**

For these exercises, you may find the OpenGL programming guide (red book) useful. An online version is available from: http://www.glprogramming.com/red/

The best advice that I can give when doing graphics programming is to start simple – do the exercises in this lab sheet first. These are not the most efficient programs but they are a good starting point for learning, without going off into the deep end.

Exercise 2.1.

Before working on this exercise, make sure that you can build and run cube1.c. You don't have to understand the code in cube1.c. We will cover this later.

In this exercise, we will learn how to create an OpenGL program. OpenGL programs, in general, follow a common structure. Here, we will look at this basic structure.

Step 1: Create a new project in CodeBlocks and then a new .c file and name it my_2dhouse.c.

Step 2: To use OpenGL routines, you need to include the appropriate libraries. In our case, we will use freeglut.h.

```
#include <gl/freeglut.h>
```

Step 3:
The main program in OpenGL has the following structure:

```
int main(int argc, char** argv) {

    glutInit(&argc,argv); /* Standard GLUT initialization */

    glutInitWindowSize(500,500);     /* 500 x 500 pixel window */

    glutInitWindowPosition(0,0);     /* place window top left on display */

    glutCreateWindow("Draw house"); /* window title */

    myinit(); /* set attributes */

    glutDisplayFunc(display); /* display callback invoked when window is opened */

    glutMainLoop(); /* enter event loop */



    return 0;

}
```

The first line of the code performs the standard GLUT initializations. The second and third lines initialize the size and position of the window in which the 3D world will be displayed. In our case, it will be 500 x 500 and will be placed at the top left of the display.

The command glutCreateWindow("Draw house"); creates the window and sets its title to "Draw house".

The function myinit(); is a function that you need to implement. In this function, you set all the necessary initializations, including the color of the background, the drawing color, etc. It is not required. However, it is highly recommended to set all the initializations specific to your program in this function.

The function glutDisplayFunc(display); sets the function that will be called every time the screen needs to be refreshed. OpenGL programs are event-driven. In other words, the program waits until an event is triggered. If an event is triggered, then the appropriate function is called to handle that event. In this case, if a display event is triggered (i.e., the window needs to be displayed), the function display() will be called. YOU NEED to implement this function in order to draw the 3D world.

Finally, by calling the function glutMainLoop();, the program will enter into an endless loop waiting for events to occur. Every time an event occurs, the appropriate function is triggered.

Step 4: Let’s look now at the mynit() function.

```
void myinit(void){

    /* attributes */

    glClearColor(1.0, 1.0, 1.0, 1.0); /* draw on white background */

    glColor3f(1.0, 0.0, 0.0); /* draw in red */

    glLineWidth(5.0);     /* draw using lines 5 pixels wide */



    /* switch matrix mode to 'projection' and

       load an identity matrix as the projection matrix */

    glMatrixMode(GL_PROJECTION);

    glLoadIdentity();



    /* set up an orthographic projection in 2D with a clipping

     rectangle which has its lower left corner at the origin (0.0,0.0) */
    gluOrtho2D(0.0,500.0, 0.0,500.0);


    /* switch matrix mode back to 'model view' */

    glMatrixMode(GL_MODELVIEW);

}
```

The first line sets the background color of the window. A color in OpenGL has 3 components: the Red, the Green, and the Blue components. The fourth value is the. Transparency of the window. 1.0 means it is opaque and 0 means it is entirely transparent.

The second line sets the drawing color. In this case, everything will be drawn in red.

The third color sets the line width (5 pixels).

The next two tasks that one needs to do in every OpenGL program is to setup a camera and then place various 3D objects in the scene.

A camera has two types of properties

- Its type, i.e., whether it uses orthographic or perspective projection.
- Its position, location and orientation in the 3D world.

In our case, since we are dealing with a 2D house, we will use orthographic projection camera. To setup this, you will first need to call glMatrixMode(GL_PROJECTION); followed by

glLoadIdentity();

- The first one tells OpenGL that now I am going to update the camera type.
- The second one glLoadIdentity();will be explained in detail in the class.

Finally, gluOrtho2D(0.0,500.0, 0.0,500.0); sets the 2D viewing field to 0.0 to 500.0 in the X direction and 0.0 to 500.0 in the Y direction.

Note that, before exiting the function myinit(), there is this line of code:

glMatrixMode(GL_MODELVIEW);

This switches to MODELVIEW, which means that whatever comes after will be for setting the location and orientation of the different objects. Recall that, a camera is just another object.

Step 5: Let’s look now at display() function

```
void display( void ){
    /* declare a point data type */
    typedef GLfloat point2[2];
    /* define coordinates for a   rectangle - the main "building block"*/
    point2 vertices[4]= {{100.0, 50.0},{100.0, 200.0},
        {120.0, 200.0},{120.0, 50.0}
    };

    glClear(GL_COLOR_BUFFER_BIT);   /*clear the window to background colour specified by
glClearColor(...)*/

    /* plot primitives to draw the house*/
    glBegin(GL_LINE_LOOP);
     glVertex2fv(vertices[3]);
     glVertex2fv(vertices[0]);
     glVertex2fv(vertices[1]);
     glVertex2fv(vertices[2]);

     glVertex2f(140.0, 50.0);
     glVertex2f(140.0, 200.0);
     glVertex2f(160.0, 200.0);
     glVertex2f(160.0, 50.0);
    glEnd();
    glBegin(GL_LINE_LOOP);
     glVertex2fv(vertices[3]);
     glVertex2fv(vertices[2]);
     glVertex2f(250.0,400.0);
    glEnd();

    glBegin(GL_LINE_LOOP);
     glVertex2f(250.0,50.0);
     glVertex2f(250.0,250.0);
     glVertex2f(300.0,250.0);
     glVertex2f(300.0,50.0);
    glEnd();

    glFlush(); /* flush buffers */
}
```

The function glClear(GL_COLOR_BUFFER_BIT);   clears the window. The following lines of code draw line loops.

To draw something in openGL, you need to do it within glBegin(…) and glEnd();

The last line glFlush(); tells OpenGL to finish drawing everything before moving to the next instructions.

The way the code of display() is written in not clean. To make cleaner, use the following:

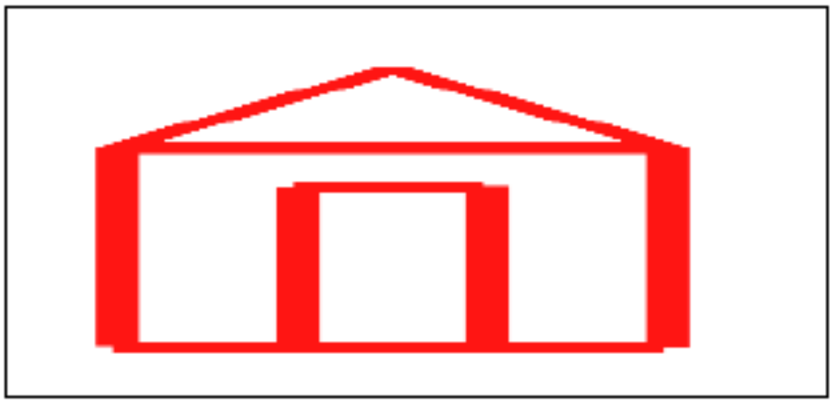
```
typedef GLfloat point2[2];

void display( void ){

     /* declare a point data type */

     glClear(GL_COLOR_BUFFER_BIT);   /*clear the window to background colour specified by

glClearColor(...)*/


     draw2DHouse();

     glFlush(); /* flush buffers */

}

void draw2DHouse(){


      /* define coordinates for a   rectangle - the main "building block"*/

      point2 vertices[4]= {{100.0, 50.0},{100.0, 200.0},

           {120.0, 200.0},{120.0, 50.0}

      };


      /* plot primitives to draw the house*/

     glBegin(GL_LINE_LOOP);

      glVertex2fv(vertices[3]);

      glVertex2fv(vertices[0]);

      glVertex2fv(vertices[1]);

      glVertex2fv(vertices[2]);
      glVertex2f(140.0, 50.0);

      glVertex2f(140.0, 200.0);

      glVertex2f(160.0, 200.0);

      glVertex2f(160.0, 50.0);

    glEnd();



    glBegin(GL_LINE_LOOP);

      glVertex2fv(vertices[3]);

      glVertex2fv(vertices[2]);

      glVertex2f(250.0,400.0);

    glEnd();



    glBegin(GL_LINE_LOOP);

      glVertex2f(250.0,50.0);

      glVertex2f(250.0,250.0);

      glVertex2f(300.0,250.0);

      glVertex2f(300.0,50.0);

    glEnd();



}
```

Question: If you execute this program, you will notice that the drawing does not look like a house. Fix the program so that the output produced should look like below but on a larger scale (you decide the scale).

Make sure that:

- you add comments to the program indicating what the code between each *glBegin* and *glEnd* pair does. There is no need to comment every line unless you want to indicate what each point does.

The code that draws the house should be on a separate function called draw2DHouse.



Exercise 2.2. Write an OpenGL program which draws and visualizes a wireframe cube of length 1, such that:

- The cube should be centered at the origin.
- The camera is located at (0, 0, 5) and looking towards the origin

Change the camera location and observe how the output image looks like.

Exercise 2.3

Do not start on this exercise without completing exercises 2.1 and 2.2 above. Also, do not start from scratch. Start from the code you have written for Exercise 2.2.

You may find the Lighthouse3D site useful:
http://www.lighthouse3d.com/tutorials/glut-tutorial/

Convert the 2D house that you created in the previous exercise to 3D. Note that this is not just giving each of the points a 3rd coordinate z, but actually creating a 3D structure.

You need to create a 3D world using glOrtho. In the line drawing of the house, you were using gluOrtho2D(0.0,500.0, 0.0,500.0). This specifies a 2D world of size 0 to 500 units across (x-axis) and 0 to 500 units bottom to top (y-axis). The origin is taken to be the bottom left corner.

In the line drawing of the house, try having any point whose x or y value is outside the specified range. Increase the size of the 2D world by changing the value of the parameters of gluOrtho2D(...) to see if this point becomes visible again.

To understand, gluOrtho2D(), think of it as creating a 2D viewing world where the left, right, bottom and top of the 2D world is specified by gluOrtho2D(left, right, bottom, top). Anything drawn inside the left, right, bottom and top of the 2D world is visible. Anything outside of these boundaries is not visible.

To create at 3D world, you need to use glOrtho(left, right, bottom, top, near, far). As before left, right, bottom and top specify boundaries of the world on the x and y axis but near and far, which are on the z axis, have a different meaning. The 2 values represent distances from the camera located at (0, 0, 0). Its usage is as follows;

```
// This is orthographic projection

        GLdouble left     = -220;

        GLdouble right    = 220;

        GLdouble bottom = -220;

        GLdouble top      = 220;

        GLdouble nearVal = 0.1;        // Near clipping plane

        GLdouble farVal   = 1000;       // Far clipping planes

        glOrtho(left,   right,

                  bottom,   top,
                    nearVal,   farVal);
```

- glOrtho(...) creates a viewing volume and only things inside this viewing volume are visible. The values of near and far are usually positive meaning they are in front of the camera with far > near. There can be some confusion as the objects in front of the camera need to have negative z values as the camera is located at (0,0,0) but looking towards the negative z axis.
- Try other 3D viewing modes, such as gluPerspective(…) and glFrustum(…). Play with the parameters and compare the results

In the 2D house, vertices were specified with glVertex2f(..) as in glVertex2f(300.0,250.0). How would you specify a 3D point?

Arrange the house so that the camera is looking at one outer corner of the house. This way, you will see the 3D version otherwise you might not see the vertices of the back of the house.

After you have done the above, use polygons instead of lines and give each polygon a different colour. Make sure you comment each set of coordinates with the side of the house you are drawing. Note the visual difference between the line drawing of a house and that using polygons.

Write a function called drawAxes(), which draws the X axis in red, the Y axis in green and Z axis in blue.

Exercise 2.4 - Graphics project/Assignment 2

Put some thought in what your project is going to be on. The LMS site has details about the project requirements as well as examples of previous projects. Do not try to build a game in this unit although this has been done before. Building a game in this unit can lead to a lot of stress.

Have you considered who your team members are going to be? Please see the section on group work in the unit guide. You can work in groups of up to 4 students per team.

External students need to email me so that arrangements can be made for working on the final project in a team.

Exercise 2.5 – Loading and visualizing complex 3D models.

Often, in Game design and development, complex 3D models are created with 3D modelling software, e.g., Maya, 3DStudioMax or AutoCAD among others. The created 3D models are then imported to the OpenGL program, displayed and animated. The objective of this exercise is to learn how to import a 3D model and display it.

In the folder "3D data", you will find two files called "bone.off" and "bone_normalized_aligned.off". Use the function readdOFFFile that you have created in the previous exercise and create a OpenGL program, which

- Prompts to the user to type the .OFF file name

- Loads the .OFF file name and displays it on a window.

Test your program using the two ".off" files included on LMS, and explain the results.

Note: If you have your personal computer, you can download and install a Software called MeshLab (http://www.meshlab.net/), which you can use to visualize 3D model

Exercise 2.6 (Extension to Exercise 2.5)
Now, we need to make sure that every time you load a 3D model,

(1) It will be always displayed at the origin, and
(2) It fits within a bounding sphere of radius 1.

To achieve this, we need to normalize the 3D model. This is done in three steps

Step 1: Find the center of mass of the 3D model. The center of mass is defined as the sum of all the vertices of the 3D model divided by the number of vertices of the 3D model.

Let **c** be the center of mass.

Step 2: Find the radius **r** of the model's bounding sphere. It is defined as follows:

- For each vertex v_i, compute the magnitude of the vector (v_i – c). Let di be this magnitude. Let d_i be this magnitude.
- The radius r is defined as the maximum among all the d_i's.

Step 3:

- Translate the entire shape with **–c**. This is equivalent to subtracting **c** from each vertex of the 3D model.
- Scale the entire shape by 1 divided by r. This is equivalent to dividing all the vertices (after subtracting **c**) by **r**.

Write a C function that performs the above steps.