Department of Computer Science
Worcester Polytechnic Institute

CS-4731: Computer Graphics
Assignment 1
Prep Work Due: October 30, 2006 at 11:59pm
New Stuff Due: November 03, 2006 at 11:59pm

Objective: In this assignment, you will learn some of the basics of OpenGL. The assignment consists of two parts: a "Preparation" part and a "New Stuff" part.

NOTE: The "prep" portion is due on October 30, and the "new" stuff is due on November 03.

Preparation: The aim of this preparation part is to get your feet wet in OpenGL. You will learn how to install and set up your OpenGL system and perform a few simple, fun tasks. You will mostly be given some readings from the book and be expected to type in an accompanying sample program or cut and paste examples. Follow the instructions and type in code carefully. **Note** that due to the large number of operating systems and compilers out there, the TAs and myself cannot provide help on all platforms. We will be glad to provide assistance on how to compile your code on ccc.wpi.edu or MinGW. You can develop on any platform with C/C++ and OpenGL, but grading will be done on CCC, so it is your responsibility to get it working there.

Setup: Here is a Makefile that works on ccc.wpi.edu, and here is a Makefile for Mac OSX. These are just samples, and assume that your OpenGL program is in a file called cs4731_ass1.cpp. Change the names accordingly for your program. Here's a sample output from using the Makefile:

```
> ls
cs4731_ass1.cpp  Makefile
> make
g++ -o ass1 cs4731_ass1.cpp -lglut -lGL -lGLU -L/usr/X11R6/lib -lX11 -lXmu -lXi -lm
> ls
cs4731_ass1.cpp  ass1  Makefile
> ./ass1
[program runs]
> make
make: `ass1' is up to date.
> make clean
rm ass1
```

Prep Coding:

1. **Draw three dots:** Read section 2.2 (page 42) of Hill book. Use the sample code in figure 2.10 (page 46) of Hill book to write a program for drawing three dots to the screen. Name the file appropriately and save it (e.g. ass1_ThreeDots.cpp). Compile and run the program!!

2. **Sierpinski Gasket:** Read example 2.2.2 (page 47) of Hill book. Use the following skeleton of the Sierpinski gasket to write a program which draws the Sierpinski gasket. The bodies of the GLIntPoint class, the random function, the drawDot function (figure 2.8, page 45) and the Sierpinski function (figure 2.14) have all been omitted. Type them in from example 2.2.2 and the appropriate figures in the text. Compile and run the program!!!

To make your Sierpinski gasket prettier, in the for loop in the Sierpinski( ) function, change the number of iterations from 1,000 to 50,000. Drawing point size should be at 1.

3. **Reading and drawing polyline files:** Read section 2.3, especially example 2.3.2 on page 54 of Hill book to draw polyline files.

   Using the following skeleton write a program to read in a polyline file and draw it to the screen. The body of the drawPolyLineFile function has been omitted. Type it in from figure 2.22 on page 56. Save and use the polyLine file dino.dat for your work.

   The basic structure of a GRS file is:

   - A number of comment lines, followed by a line starting with at least one asterisk: '*'.
   - The "extent" of the figure: (left, top, right, bottom).
   - The number of polylines in the figure.
   - The list of polylines: each starts with the number of points in the polyline, followed by the (x, y) pairs for each point.
   - Note that a lot of the GRS files start with comments. You should read them in and ignore them in your program.
   - The format for dino.dat is a little different in that it doesn't have the window dimensions (or comments) right at the top. Therefore, off the bat, a program which reads other GRS files without problems will have new problems with your old dino.dat file. You can either throw in a dummy extents window at the top of dino.dat or come up with a solution that works.

     Hint: A value of (0, 640, 0, 480) should work!!

   Compile and run the program!!

Note: Make sure you read the examples in the book before typing in the examples.

New Stuff: The aim of this part is to get you comfortable with working in 2D using OpenGL calls, as well as applying window-to-viewport mapping in zooming and mouse selection routines.

Here goes:

1. **Step 1: Read more polyline files in GRS format**

   The polyline file you drew in the preparation part was in GRS format, a homegrown format. Here are a few more GRS polyline files to work with:

   - birdhead.dat
   - dragon.dat
   - house.dat
   - knight.dat
   - rex.dat
   - scene.dat
   - usa.dat
   - vinci.dat

   So, first make a copy of your Preparation program(s) and compile the copy. Next, modify your code that previously read in dino.dat so that you can replace dino.dat

filename with any of the above .dat files. The plan is that when you save and compile it, the new file is drawn on the screen, just like dino.dat. There are a few things you should note for your implementation:

1. Be careful with how you pass parameters to the glViewport( ), glOrtho2D( ) calls (Practice exercise 3.2.1 on page 86 of Hill may prove useful). Make sure you understand how they work.
2. The "extent" line of the GRS file will be used to set the world window i.e. passed to glOrtho2D( )
3. If you look at the vertex coordinates of some of the GRS files, they are specified in floating point numbers, so you'll have to switch to float or double number formats. For instance, the x and y coordinates of the vertex are currently (in the preparation) being read in as integers and their values will be truncated badly if you don't change them to floating point numbers. Also, remember that glVertex2i( ) uses integers, so you'll need a different call to work with floats. Also, in general, using a glVertex3f( ) with the z value set to zero (see examples) can be used in place of glVertex2f( ) commands. i.e. glVertex3f(x, y, 0) usually works same as glVertex2f(x, y).

2. **Step 2: Tiling and viewports**

   Your program should have the following behavior and user (keyboard and mouse) interaction capabilities when you run it:

   Event: A key is pressed:

   - **'d' key (state d) Response:** Draw three dots (as explained above)
   - **'s' key (state s) Response:** Draw the Sierpinski gasket (as explained above)
   - **'p' key: (state p) Response:** the program creates a 6x6 non-distorted tiling of the 9 polyline files provided (8 new ones above + dino.dat). The position of polyline files is completely random such that repeatedly hitting the 'p' key produces new arrangements of 6x6 tiles.

     In this state, if a user clicks on any of the 6x6 tiles, the polyline file in that tile becomes the current drawing. The screen is erased and the polyline in that tile is redrawn to fill the entire screen window. When one polyline file is drawn to fill the entire window, this is known as **state 'f'**.

     In state 'f', the program can respond to two key strokes, 'v' and 'h'. The polyline which now covers the entire window is known as the current polyline. From state 'f' the user can also use two mouse clicks to specify a portion (i.e., bottom right and top left corners) of the current polyline file, which your program then zooms in on **(state 'z')**. This zooming action can be repeated over and over until the user is satisfied.

   - **'h' key: (state h) Response:** The 6x6 tiling is drawn using the current polyline. However, each alternate polyline is inverted along the horizontal axis. i.e., each polyline file 1, 3, 5, ... in row 1 is drawn upside down. Polyline 2, 4, 6, ... in row 2 is drawn upside down, etc.
   - **'v' key: (state v) Response:** The 6x6 tiling is drawn using the current polyline. However, each alternate polyline is inverted along the vertical axis. i.e., each polyline file 1, 3, 5, ... in row 1 is drawn with the left and right sides flipped. Polyline 2, 4, 6, ... in row 2 are drawn with the left and right sides flipped.

   Note: State 'd' is the initial state when your program starts.

Here is a state-transition matrix to help clarify things:

		...to one of these states, do what it says in the cell.						
		d	**s**	**p**	**f**	**h**	**v**	**z**
To go from one of these states...	**d**	Press 'd'	Press 's'	Press 'p'				
	s	Press 'd'	Press 's'	Press 'p'				
	p	Press 'd'	Press 's'	Press 'p'	Click on a tile			
	f	Press 'd'	Press 's'	Press 'p'		Press 'h'	Press 'v'	Click 2 extents
	h	Press 'd'	Press 's'	Press 'p'			Press 'v'	
	v	Press 'd'	Press 's'	Press 'p'		Press 'h'		
	z	Press 'd'	Press 's'	Press 'p'				Click 2 extents

State transition summary:

- From any state, you can press 'd' , 's' , or 'p' to go into the corresponding state
- From 'p' you can click on a tile to go to 'f' or re-randomize the grid by pressing 'p'
- From 'f' you can press 'h' or 'v' or click two points to zoom in ('z')
- From 'h' you can press 'v'
- From 'v' you can press 'h'
- From 'z' you can click two points and zoom in further 'z'

Clarification of states:

d	drawThreeDots
s	Serpinski gasket
p	6x6 grid of randomized polyline files
f	a full-screen version of the polyline that was chosen from the grid that was just on the screen.
h	the single polyline file that was just displayed on the screen should be made into a 6x6 grid with alternating tiles being flipped along the horizontal axis (top becomes bottom)
v	the single polyline file that was just displayed on the screen should be made into a 6x6 grid with alternating tiles being flipped along the vertical axis (leftside becomes rightside)
z	a zoomed in area of the previous polyline file that was displayed

Make sure that reshape works for all states (d, s, p, f, h, v, and z). i.e., if the user grabs the lower right corner of the window and increases or reduces the screen window size, whatever was drawn in it before is redrawn to the largest possible size, without distortion.

Structure: As some of you have probably noticed, a good approach of how to attack this assignment is through divide and conquer (successive refinement). Start out *simple*, and just get the basics to work. You might start out with a high-level structure of function calls you plan to code. For example, to do the polyline part, you might have a main that looks like this:

```
int main( int argc, char ** argv )  {
  readPolyLineFiles( );
  drawTiling( );
}
```

Then drill down into each of these routines to add more functionality, testing each one as you go. Don't try to solve the whole problem at once, just divide it up.

Documentation: You must create adequate documentation, both internal and external, along with your assignment. The best way to produce internal documentation is by including inline comments. The preferred way to do this is to write the comments *as you code*. Get in the habit of writing comments as you type in your code. A good rule of thumb is that all code that does something non-trivial should have comments describing what you are doing. This is as much for others who might have to maintain your code, as for you (imagine you have to go back and maintain code you have not looked at for six months -- this **WILL** happen to you in the future!).

I use these file and function (method) headers, in my code. Please adopt these (or the official CS ones) for all your assignments. The file header should be used for both ".h" and ".c" (or ".cpp") files.

Create external documentation for your program and submit it along with the project. The documentation does not have to be unnecessarily long, but should explain briefly what each part of your program does, and how your filenames tie in. Most importantly, tell the TA how to compile and run your program.

What to Turn in: Submit everything you need to compile and run your program (source files, data files, etc.)

BEFORE YOU SUBMIT YOUR ASSIGNMENT, put everything in one directory on ccc.wpi.edu, compile it, and make sure it runs. Then tar everything up into a single archive file.

The command to tar everything, assuming your code is in a directory "ass1", is:

tar cvf FirstName_LastName_ass1.tar ass1

To submit your work, you will use the turnin utility on CCC. Here is a link to instructions about how to do this. The turnin ID for the "Prep" part is "hw1-prep" and for the "new stuff" part is: "hw1", so to submit your new stuff, you would type something like this when logged in to the CCC machine:

/cs/bin/turnin submit cs4731 hw1 FirstName_LastName_ass1.tar

General Hints: Here are a few more hints you might find useful:

- You need to modify your keyboard( ) function in order to react to keyboard input from the user and your mouse( ) function to react to mouse input.
- Think for a moment (okay, maybe a *few* moments), what you would consider ***the most difficult*** part of this project. How are you going to do that part?
- The following sections of Hill may be useful in doing your work:
  - Section 2.4 of Hill (pp. 63 - 67, 5 pages) explains simple mouse and keyboard interaction using openGL.
  - Practice exercise 3.2.1 (pp. 86-87, 2 pages) is about using gluOrtho2D( ) and glViewport( ) to do window-to-viewport mapping using OpenGL
  - Example 3.2.4 of Hill, (pp. 88-89, 2 pages) tells you how to do tiling

    - Section 3.2.2 (pp. 92 - 95, 4 pages) on how to set Window-to-Viewport mappings, while preserving aspect ratios i.e. no distortion.
    - Read example 2.4.2, figure 2.38 (pp 64, 0.5 page), to understand how to collect mouse points two different sets of mouse points. Since this routine is actually for drawing the selected screen rectangle, you will probably need to modify this.
  - For the reshape part, after a user changes your screen window (viewport) dimensions by dragging the lower right corner, simply call glViewport again using the new width and height, and then redraw using glutPostRedisplay( ).

Note: Don't blindly call glOrtho2D and glViewport without thinking about how they work.

Academic Honesty: Remember the policy on Academic Honesty: You may discuss the assignment with others, but you are to do your own work. The official WPI statement for Academic Honesty can be accessed HERE.

Back to course page.