CS175 - Assignment 1
Hello World (OpenGL)

Due: Monday, September 17, 11:59pm

1 Objectives

The purpose of this assignment is to give you experience with programming simple OpenGL applications. You will also get some initial experience with programmable shaders.

Please follow the setup instructions on the course website to get the assignment.

The following is a brief description of the files you will find:

- `asst1.cpp` contains the `main` function and has code to initialize OpenGL and draw a textured square. It also performs proper resource management such as deallocating shaders, buffer objects, and textures appropriately, and gracefully handles runtime errors. You will work from this file.
- `ppm.[cpp,h]`: provides functions for reading/writing images in a very simple image file format (PPM).
- `glsupport.[cpp,h]`: provides utility functions that make working with OpenGL easier.
- `Makefile`: makefiles for Linux and Mac users.
- `[reachup, smiley, shield].ppm`: example image files (in PPM format) to experiment with.
- `shaders/asst1-sq-gl[2,3].[f,v]shader`: shader files loaded by the program.
- `asst1.[vcxproj,vcxproj.filters,sln]`: solution and project files for Visual Studio.
- `cs175-asst1.xcodeproj`: project files for XCode.

The starter code supports both OpenGL 3.x+ with GLSL 1.3+ and OpenGL 2.x with GLSL 1.0, controlled by the global variable `g_Gl2Compatible`. By default it is set to `false`, meaning OpenGL 3.x+ should be used. Before running, you should try upgrading your graphics driver to the latest. Next you should try to run the compiled executable without changing `g_Gl2Compatible` first, and only if an exception "Error: card/driver does not support OpenGL Shading Language v1.3" is thrown, should you consider setting `g_Gl2Compatible` to `false`. If, after the change, the exception "Error: card/driver does not support OpenGL Shading Language v1.0" is thrown, then your system is extremely old. Depending on if `g_Gl2Compatible` is `true`, you should edit either `shaders/asst1-sq-gl2.[f,v]shader` (for GL2 shaders) or `shaders/asst1-sq-gl3.[f,v]shader`. You only need to complete the assignment for one choice of the above.

Use GL 2 only if you absolutely have to. Please email the TF to let the TF know.

You should write your **own** README.txt file for assignment submission. Indicate with whom you worked.

1.1 PPM Image format

Feel free to create your own PPM texture files. If you do so, you'll want to note that OpenGL works best with textures that have dimensions that are powers of two (e.g., 256x256, 512x512, ...). On the Science Center linux and Mac machines, `xv`, `gimp`, `convert`, and `display` may be convenient for image viewing and editing. On Mac and Windows, Photoshop provides a lot of relevant features. Windows users can download a free PPM viewer called `ifranview` to view PPM files. The PPM image format itself is quite simple, but you do not have to understand it to do the assignments: it has a header giving the dimensions of the image (n, m) followed by n*m triplets of bytes representing red, green, and blue. The pixels in a ppm file appear from left to right within a scanline (**row-major order**), with scanlines appearing from top to bottom. PPM files can be written in ASCII text mode or binary mode. We use the binary mode in our provided functions for compactness. Therefore, when editing and saving files in your chosen paint program be sure to choose the binary option.

We provide you with source code for functions that read in a ppm file, and store the data into a (n*m) array of pixels. A pixel is represented as three bytes (unsigned chars). The functions we provide flip the vertical order, so that when indexing into the array, we can interpret the coordinate system to be (x,y) with x going left to right, and y going from bottom to top. You can look specifically at the function:
`void` **`ppmRead`**`(const char *filename, int& width, int& height, std::vector<PackedPixel>& pixels)`
for more information.

2 Task 1

The first step is to set up your programming environment so that you can begin to work with OpenGL programs. You will need the following:

- **Hardware support** You will need hardware that supports OpenGL 3.0 and OpenGL Shading Language 1.3, or alternative OpenGL 2.0 and OpenGL Shading Language 1.0. In practice any computer purchased with an NVIDIA or ATI graphics card within the last 7 years should support it. To test this, you can compile and execute the code provided, if your hardware does not support OpenGL 3.0 then the program will output a message notifying you of it.
- **Driver support** Even if you have a new or up-to-date graphics card installed in your machine, your programs may not be able to utilize the advanced features unless you have installed new drivers. OpenGL is a continually evolving standard and the various graphics chip vendors make regular releases of drivers exposing more and more advanced features (as well as fixing bugs in previous versions). If you are working on your own computer, you will want to download and install the latest drivers. On Mac, you shouldn't have to do this.
- **GLUT and GLEW**. We have include the headers (.h), and binaries (.lib, .dll/so/a) for them.

Build the program and make sure it runs. If you get an error saying "shared_pointer is ambiguous", try commenting out lines 13-15 and line 29 in asst1.cpp. If there are remaining outstanding issues, feel free to post on Piazza.

3 Task 2

Now that the build environment is set up, it is time to get experience playing with OpenGL. Modify the program so that when the window is reshaped, the aspect ratio of the object drawn **does not change** (i.e., if a square is rendered, resizing the window should not make it look like a rectangle with different edge lengths), and it **doesn't crop** the image. Making the window uniformly larger should also make the object uniformly larger. Likewise for uniform shrinking. Figure 1 shows examples of how a window should look when resized:

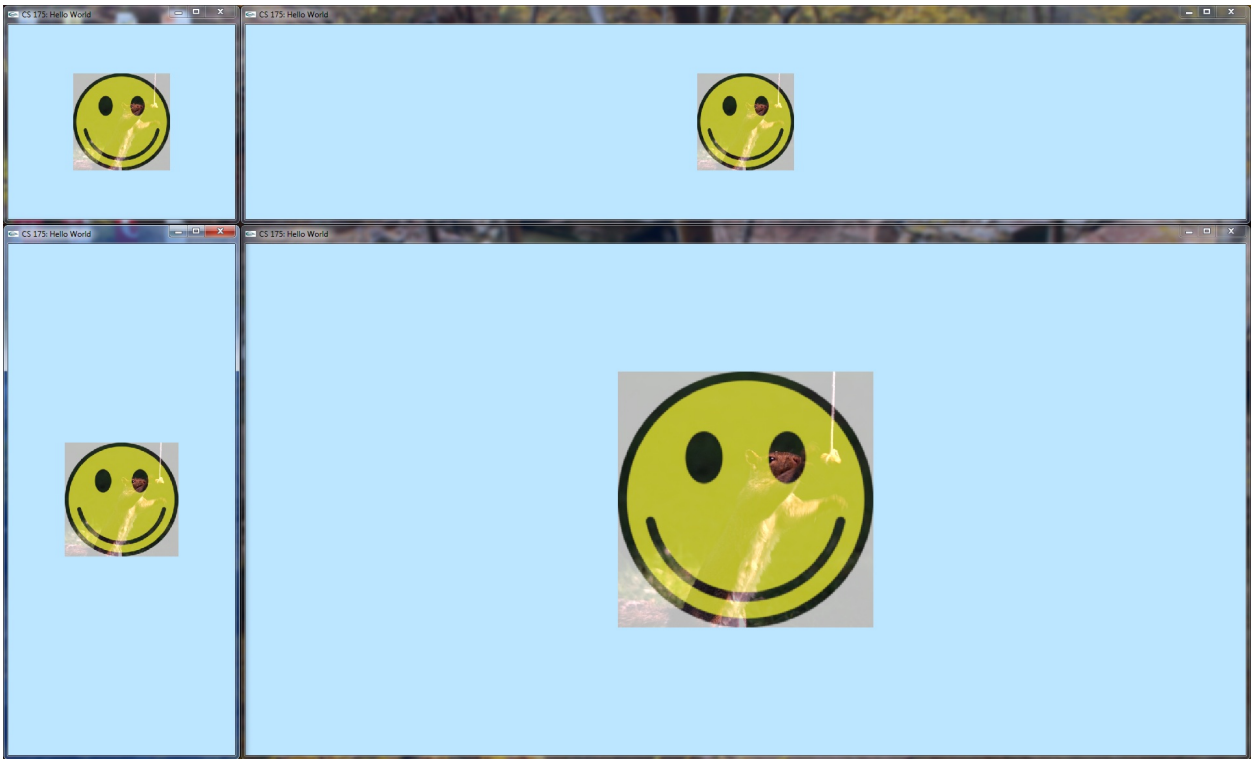

Figure 1: Some sample resized windows and the expected behavior: the drawing is not cropped, and resizing tries to make the drawing as large as possible within the window.

There are some important constraints:

- Your solution cannot change the vertices' coordinates that are sent by your program to the vertex shader.
- Your solution cannot modify the call to `glViewport` (which is in the `reshape` function in `asst1.cpp`).
- The existing behavior that you can change the rectangle width by right mouse button dragging must be preserved. So if you have made the square a rectangle by right dragging, and then resize the window, your program should preserve the aspect ratio of the rectangle.
- [**Hint:**] Your solution can modify the vertex shader, as well as change uniform variables that are used in the vertex shader.
- When possible, it is often better to have `if` statements and any other complex statements in your C++ code than in your shader codes as long as the two versions produce the same

result. This is so because your C++ code and your shader codes are executed at very different frequencies. The C++ code might be executed once per frame, whereas vertex shader codes are executed once per vertex per frame, and shader codes are executed once per fragment per triangle per frame.

4 Task 3

For this task, you will draw an extra colored and textured geometry on the screen using OpenGL. You need to implement the following requirements:

- The geometry cannot be a square (since we already have it in the code), but it can be as simple as a triangle. Any other shape will also do.
- The geometry has at least the following per-vertex attributes: position (2 floats), color (3 floats), and texture coordinates (2 floats).
- The vertices of the geometry are differently colored.
- The triangle is texture mapped with the "shield.ppm" image file in the project directory. Alternatively, you can supply and use your own favorite texture.
- The geometry's on-screen position is controlled by the keyboard using keys "i", "j", "k" and "l" for moving up, left, down and right.
- The aspect ratio of the triangle stays constant independent of the window size, as the square in Task 2.

Hint For this task, you will want to create your own pair of vertex and fragment shaders, and load them in the C++ code. You also want to create your own ShadersState struct and Geometry struct.

Towards completing this task, you will need to write and load your own vertex and fragment shaders, define and load your own `ShaderState` and `Geometry` structs, load new textures, check for custom key presses in the GLUT keyboard callback, and finally bind all the GL resources and draw the geometry. Refer to the solution binary for an example.

5 Submission

Please refer the submission instructions on the course website.