Hw2d Project Addenda

CS 320

Note the following differences between the project description for Harvard's CS 175 class and our class:

1. The CS 175 due date is not applicable.
2. The starter code location is not applicable.
3. Ignore Section 1.1; I suggest using the reachup.ppm image in the project archive.
4. Section 4 — I used the 'u', 'd', 'l', and 'r' keys for translating my triangle.
5. Ignore Section 5. Make sure that comments in your programs indicate both members of your programming pair. Export your project as an archive file and submit it through Goucher-Learn.

CS175 - Assignment 1
Hello World (OpenGL)

Due: Monday, September 16, 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.

You can download the assignment 1 starter code (C++) at: `http://www.fas.harvard.edu/~lib175/asst/asst1.zip`

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 10.

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 supports neither GL2 nor GL3 and you should probably use the science linux boxes. 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.

You should write your own README.txt file for assignment submission.

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 3 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. If you are working on the Science Center machines, this has already been done for you. On Intel-Macs this is expected to not be an issue (if it does not work on your Intel-Mac, please post on Piazza or email the course staff at `lib175@fas.harvard.edu` about it).

- **Installing GLUT and GLEW**. GLUT and GLEW are cross-platform libraries that make interfacing with OpenGL and OS specific tasks easier. GLUT handles windowing and message control. GLEW loads the various advanced extensions that have come out since OpenGL's conception. You will need to download and install the headers (.h), and binaries (.lib, .dll/so/a) for them. If you are working on the Science Center machines, this has been done

for you. More info can be found on the class website under the Miscellaneous heading at: `http://www.fas.harvard.edu/~lib175/resources.html`

Once you get all the hardware and library issues sorted out, build the program and make sure it runs. If there are remaining outstanding issues, feel free to post on Piazza or e-mail the course staff at `lib175@fas.harvard.edu`.

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:

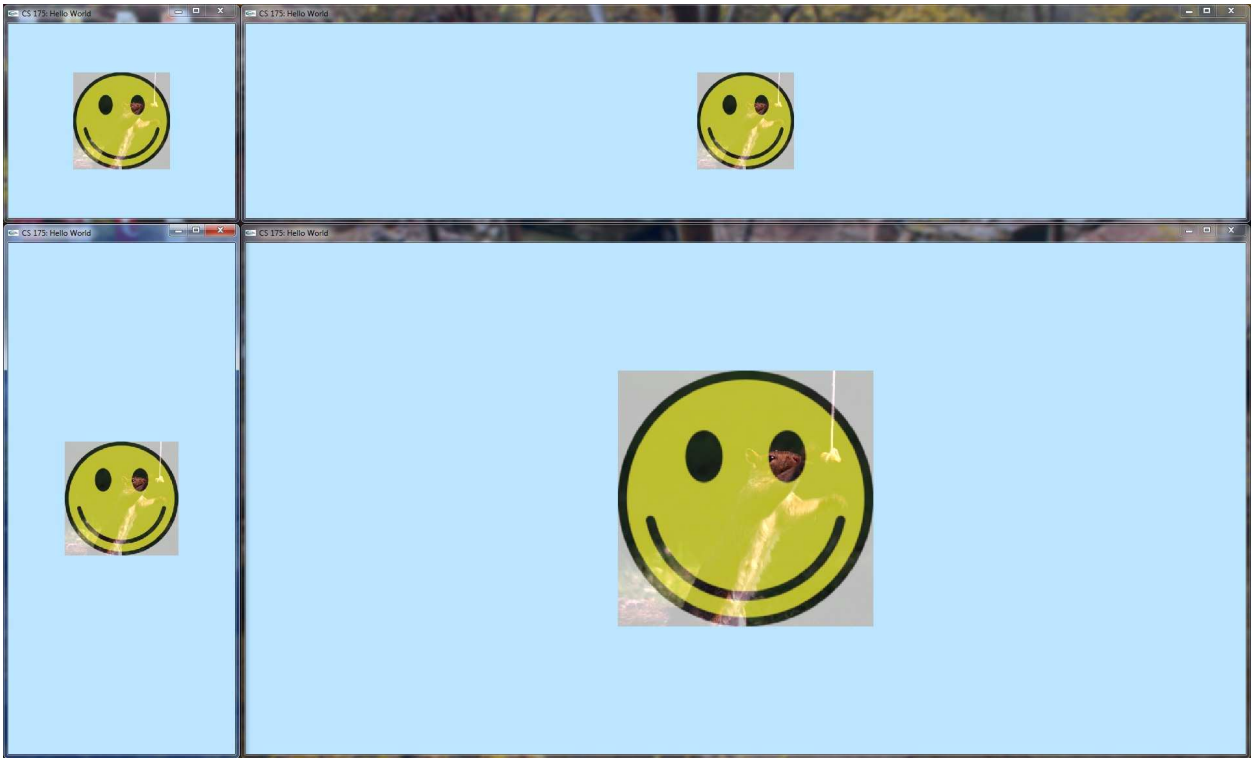

Figure 1: Some sample resized windows and the expected behavior: the drawing is not cropped, and resizing tries to make the drawing as large as possible within the window.

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`).

- [**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.

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

To submit your assignment, you will use your `nice.harvard.edu` (NICE) account (using `fas.harvard.edu` might not work). First upload your project directory onto your NICE account. On windows you can use a GUI client like WinSCP. On Mac or Linux, you can use the standard `scp` command. For example, suppose your NICE user name is `abc`, and your local project directory is `~/asst1`. Then to upload your local project directory to your NICE home directory, use the following command under shell: `scp -r ~/asst1 abc@nice.harvard.edu:~/asst1`.

If you use Visual Studio, you will also want to delete the following files and directories from your project directory before uploading: `*.sdf`, `ipch`, `Debug`, and `Release`. These files and folders take up enormous amount of space, whereas your NICE account has a relatively small disk quota. Exceeding quota while you are uploading might cause some of the actual source files to be missing in the submission.

If you use the Science Center Linux boxes, you can skip the above step, since you will already be working inside your NICE home folder when you develop on those machines.

After you have uploaded your local project directory to NICE, you will need to log into your NICE account using an SSH client (e.g., PuTTY for Windows, and the standard ssh command for Mac or Linux), and use the `submit` script from there. Once in your account, and from the prompt, you will want to change directories into the directory containing the files you want to submit. Inside that directory, type: `submit lib175 1 ‘pwd‘` where the ticks are not apostrophes but back-ticks. Instead of `‘pwd‘`, you can also write down an absolute path like `~/src/asst1`, if that is where your assignment is. The submit script works with absolute paths and will submit all the files and subdirectories within the specified directory. You should get a message claiming success or failure after running the script. (Please do not submit your whole user account: do not execute submit from your home directory). In addition to your source code, you should also include a `README` file. Also, you can check if your submission is successful using the `submit` command. Details on submission can be found at: `http://www.fas.harvard.edu/~lib175/pages/asstinfo.html`