CS/SE 3GC3 Lab 6

November 17, 2019

1 Resources

1. Red Book Chapter 8 (Drawing Pixels, Bitmaps, Fonts, Images) `http://www.glprogramming.com/red/chapter08.html`
2. Red Book Chapter 9 (Texture Mapping) `http://www.glprogramming.com/red/chapter09.html`
3. GLUT documentation (e.g., `glutInitWindowSize`) `https://www.opengl.org/resources/libraries/glut/spec3/spec3.html`

2 Practice Exercises

Congratulations, you've made it to the last lab of the semester. No graded exercises today!

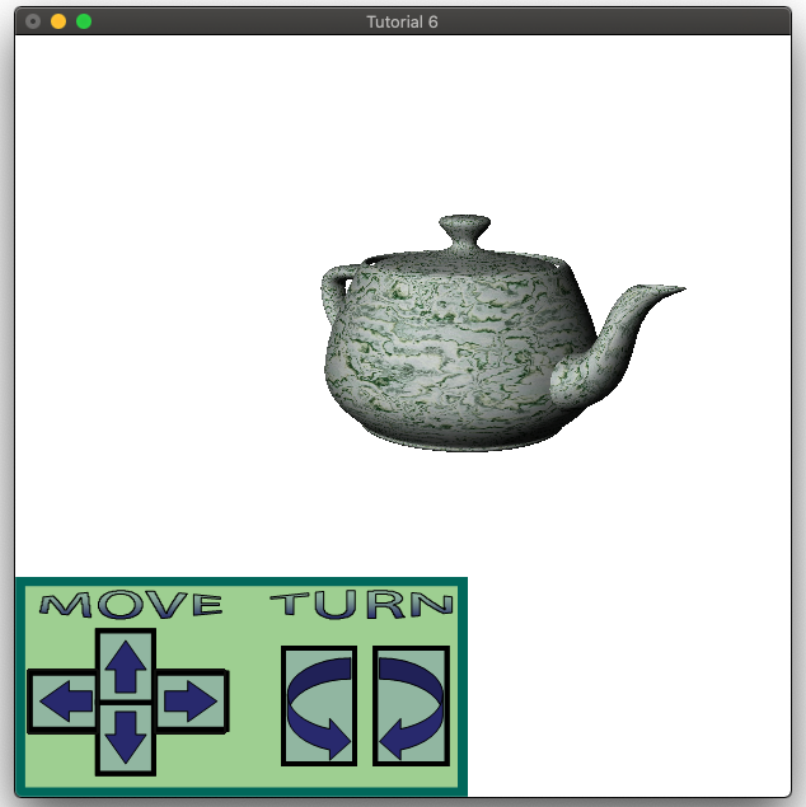

Figure 1: Final result of today's lab

1. Run `make` to compile and run `tut6.cc`.

2. The boilerplate provides a `Teapot` struct with a `draw` function. Complete the `draw` function.

3. We're going to load PPM images so that we can draw them to the screen in 2D and so that we can use them for textures on objects (imagine a marbled teapot instead of only one shaded with lights).

4. Code to parse PPM images is already provided for you in `PPM.cc`. You should not modify this code for this lab. A word of caution: this parsing code is not very good, it's from a previous year.

5. When we parse a PPM image we need to store a byte for each RGB channel for each pixel, as well as its dimensions. The `Image` struct provides the

framework necessary for this. We are going to make the `Image` provide a helper function to draw its image or to configure texture state from its image.

6. Complete the `load` function in `Image`. You should call `LoadPPM` and assign its output to `mImage`.

```
LoadPPM(filename, &mWidth, &mHeight)
```

7. Complete the `draw` function in `Image`.
   (a) The PPM parser we are using packs the image right-to-left instead of left-to-right. That is, the image in memory is mirrored horizontally. We need to do two things to fix this when drawing. We are going to draw it from the right-hand side, and we are going to "flip" the image. The code for this is provided for you with the `glRasterPos2i` and `glPixelZoom` functions. You should look up the documentation for these functions, they are **extremely** useful for working with rasters.
   (b) To actually draw the image to our framebuffer we must use `glDrawPixels`. Look up the documentation for this function and call it after `glPixelZoom` using the members of the struct. `https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glDrawPixels.xml`. Note that our type is `GL_UNSIGNED_BYTE` and our format is `GL_RGB`.

8. We have an `Image` instance named `hudInterfaceImage` ("hud" for "heads-up display"). We must call `load` on it with the `interface.ppm` file in our `main` function before we start GLUT's main loop.

```
hudInterfaceImage.load("interface.ppm");
```

9. We are now ready to draw our bitmap to the framebuffer.

10. We need to draw the bitmap in an orthographic projection, not a perspective one. However, we're going to render a teapot in our scene as well. This means we need to render 2D and 3D "objects"! Luckily, we can do this without too much trouble, but we must be careful.

11. The `display` function is much smaller than usual. It first calls `displayOrthographic` and then calls `displayPerspective`. The former will be responsible for setting an orthographic projection and then rendering 2D objects. The latter will be responsible for setting a perspective projection and then redering 3D objects.

12. Call `draw` on the `hudInterfaceImage` instance in the `displayOrthographic` function. We are going to render the image in the bottom-left of the screen, so we want to call it with parameters 0, 0. You can play with these values and see where it renders.

13. At this point, you should compile and see your image in the bottom-left corner. If you are on a newer laptop with a higher pixel density (e.g., Apple's "retina" display) then you should read the comment about the `glPixelZoom` call carefully.

14. Now we want to have our teapot translate or rotate when we click the "buttons" on the image we just drew.

15. The boilerplate includes the beginnings of a powerful framework for accomplishing this task.

16. All the GLUT lifecycle functions we have used have take a function as an argument! e.g., `glutKeyboardFunc` is called with the name of a function we've written, like `handleKeyboard`. These are called **callbacks** using "higher-order functions" (don't worry if this is scary, however take 3FP3 next semester if you're interested!).

17. We are going to set up the same framework for our buttons. We have a `Handler` struct. It has four numbers representing the 2D bounds on the screen of a given button. It also has a *function pointer* which stores a reference to the function we are going to call when our button is clicked. The syntax looks rather weird in C/C++ but for our purposes we can essentially ignore it and trust the boilerplate.

18. You'll notice there is a function given to you, `drawBoxVertices`. This function will let you preview the 2D bounds by drawing a box on the screen.

19. Complete the function `isInBounds`. This should return true if the given `x` and `y` coordinate is inside the rectangle defined by `mLeft`, `mRight`, `mTop`, `mBottom`.

20. Read the function `handleClickAt` carefully. This function takes the coordinates of a mouse click and then calls our callback if the mouse click is inside the rectangle bounds of this button.

21. An instance of the `Handler` struct defines the boundaries and function of a single button. We have six buttons on our interface, we'll need six `Handler` instances and a way to manage all of them.

22. The `InteractionHandler` struct is responsible for just that. Complete its `leftClickDown` function which loops through each of its handlers and calls its `handleClickAt` function with the given coordinates. You can see the completed `drawHandlers` function for an example.

23. The `addHandler` function takes a `Handler` instance and adds it to its vector. Complete this function by using `push_back` on `mHandlers`.

24. We need to hook up our `InteractionHandler` instance called `mouseHandler` to`glutMouseFunc`. We know that `mouseHandler.leftClickDown` will check all our buttons. Complete the `handleMouse` button to give the coordinates of a left (`GLUT_LEFT_BUTTON`) down (`GLUT_DOWN`) click to the `leftClickDown` function. REMEMBER: We need to flip our y-coordinate. Use `viewportHeight - y`.

25. Now we just need to define all our buttons and add them to our `mouseHandler` instance! Two handlers are provided for you: `leftButton` and `rightButton`.

26. Disable lighting temporarily so that we can see the unlit boundary previews.

27. Add the `leftButton` and `rightButton` handlers using `addHandler` in the `main` function. e.g.,

```
mouseHandler.addHandler(&leftButton);
```

28. Compile and make sure you can see the white boundaries of the left and right buttons. Clicking them should now move the teapot (you'll need to turn lighting back on to see this).

29. Add the remaining four handlers, two for up and down and two for rotating the teapot. You will need to add four new `Handler` instances and four new functions like `moveLeft` and `moveRight`.

30. It is a little annoying to get the coordinates of the boundaries correct. I recommend compiling often with lighting turned off so you can see the boundary previews. You can also click on the screen and see the coordinates, use this to click on the corners of each button to get the right coordinates.

31. Make sure you turn lighting back on and that your interface can translate and rotate the teapot.

32. We're now going to map a 2D texture to the 3D teapot. The texture is also in a PPM bitmap named `marble.ppm`. I strongly recommend reading Chapter 9 of The Red Book, at least the first section titled "An Overview and an Example".

33. Just like with `hudInterfaceImage`, we have an `Image` instance named `marbleTexture`. We must call `load` on this instance with `marble.ppm`. Do this in the `main` function.

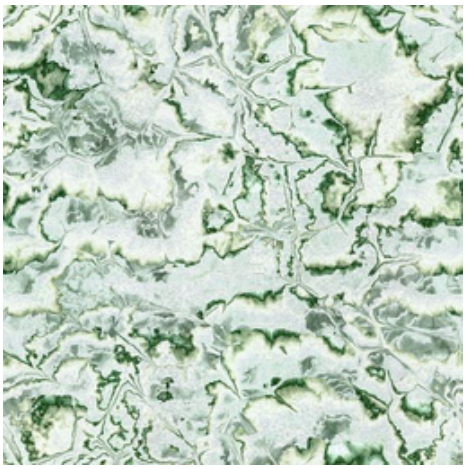

Figure 2: What the marble texture looks like in 2D

34. Complete the `texture` function of the `Image` struct. Look up the documentation for the `glTexImage2D` function. You'll need values like this,

```
glTexImage2D(
    GL_TEXTURE_2D,
    0,
    GL_RGB,
    mWidth,
    mHeight,
    0,
    GL_RGB,
    GL_UNSIGNED_BYTE,
    mImage
);
```

35. You'll also need to set the texture parameters. This is the same concept as setting material parameters before rendering vertices. Use `glTexParameterf` for this.

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

36. Now we need to call `texture` on `marbleTexture` before we render our teapot. Complete the code in `displayPerspective`.

37. One last thing! Just like we have to enable `GL_LIGHTING` we must also enable textures. Go to the `main` function and enable `GL_TEXTURE_2D`.

38. Compile and you should now have a marbled teapot!

2.1 Going Further

1. In the past we have had to set colors, normals, and material properties before rendering vertices. That is, we have something like this:

```
glColor3f(...);
glNormal3f(...);
glVertex3f(...);
```

2. `glutSolidTeapot` and the other GLUT convenience functions set normals and default material properties for us. They also set default texture mapping! Normally, we would need to set some state on each vertex to tell OpenGL where in the 2D texture this vertex will be mapped to. OpenGL then performs interpolation (this is the same concept as setting a color for each vertex and then having OpenGL interpolate all the pixels in between).

3. Draw a cube and set texture coordinates manually using `glTexCoord2f`. There is an example of this in Chapter 9 of The Red Book.

4. Render multiple teapots and write keyboard controls to switch which teapot the HUD interface affects.