CSC418 Assignment 3 & 4
Sandro Young

In general, I completed the assignment in the order in which tasks were listed in the handout:

- For assignment 3
    - First I read all the code and tried to get a handle on how things were organized
    - Then I implemented the sphere and plane intersection functions
    - Then I implemented proper normal & point-of-intersection calculation
    - Then I implemented the phong lighting model, one component at a time
    - Then I implemented shadows
    - Then I implemented reflections
- For assignment 4
    - First I implemented the two mandatory new features: area lights and anti-aliasing
    - Then I chose and implemented optional new features: two new primitives, texture mapping, environment mapping, multithreading, and glossy reflections
    - Then I built my custom scene

At each stage, I produced rendered output and compared it against the expected output. I didn't proceed to the next step until I had debugged the previous step.

The most difficult part was tracking down bugs. There were two especially pernicious bugs. First, in my sphere-intersect function, whenever there were two possible solutions, I was returning the wrong solution. This had the effect of subtly messing up all of the lighting. I spent a really long time trying to debug my normal computation and Phong lighting model implementation, only to find that the problem was elsewhere. Second, when I implemented multithreading, I didn't realize I was using a random number generator (drand48) that was not thread safe. I eventually had to switch to erand48, and use separate generators for each thread.

Most of my assignment 3 code is straightforward. The only thing I think needs to be described is my plane intersection code. The plane intersection code that I implemented works by using the same technique as the triangle intersection code from class. However, instead of allowing points where beta + gamma < 1, I allow all points where beta < 1 and gamma < 1 (this gives a complete rectangular section of plane instead of a triangle). In hindsight, the implementation could have been much simpler, since we don't need to handle intersections with arbitrary planes; we only need to handle the canonical plane. Thus I could have just solved the one-variable linear equation of finding the lambda which sets the ray's z coordinate to zero, and then checked if that intersection point lay within [-1, 1] x [-1, 1] in the XY plane.

For assignment 4, I implemented the following features:

- Anti-aliasing. I used the technique we learned in class of firing multiple rays through each pixel (100 in my implementation). I subdivide the pixel into a 10x10 grid. Then in each grid cell, I randomly choose a point to fire the light ray (to avoid Moiré patterns as discussed in class).

- Area light sources. As required, I implemented these as a series of point light sources, arranged in a grid. I arranged these point light sources in a grid on the canonical plane, and then used the transformation matrix of the plane to move them to the correct location.
- New primitive: I added a cylinder primitive. I implemented intersection in three parts: the top and bottom caps of the cylinder, and the curved sides of the cylinder. The top and bottom intersections were first treated as intersections with infinite planes, and then I further required that the intersection point be inside a unit circle. The curved intersection was treated as an intersection with the 2D circle x^2 + y^2 = 1 (using a quadratic equation, similar to spheres), and then further requiring that the intersection point have a z coordinate between 0 and 1.
- New primitive: I added a hemisphere primitive. I implemented intersection the same way as sphere intersection, with the additional requirement that the intersection point have z > 0.
- Glossy reflections: I used the algorithm presented in tutorial. Note that for this to work nicely, antialiasing must be turned on. I only fire one outgoing glossy reflection ray per incoming light ray, so in order to get a nice “soft” effect, we need the averaging provided by antialiasing.
- Texture mapping: I implemented texture mapping for both spheres and planes. Since the intersection code works with canonical objects, I only needed to produce texture coordinates for the canonical objects. For planes, texture coordinates are simply scaled x and y coordinates. For spheres, I convert the point of intersection to spherical coordinates and use phi and theta. I implemented bilinear interpolation to get the colour for points with non-integer texture coordinates.
- Environment mapping. I use the pseudocode that was linked in tutorial, here: https://en.wikipedia.org/wiki/Cube_mapping. I piggybacked off of the existing texture mapping code, and just implemented a couple of helper utility functions and structs to load and store the 6 cube face images.
- Multi-threading: I used OpenMP. This was mostly straight-forward, but drand48 is not thread-safe, so I had to use erand48 and have separate random-number-generator state for each thread.

The final scene that I implemented shows off all of these features:

- It uses environment mapping to create a chapel background. The image used for the environment map was taken from here: http://www.humus.name/index.php?page=Textures&ID=110
- It uses texture mapping on the marble table and on the fruits. The texture map images were taken from:
    - http://www.austincc.edu/sfarr/online/3dls/images/Apple-Texture-map.jpg
    - https://s-media-cache-ak0.pinimg.com/originals/93/9f/a2/939fa2cfa6f6361484a6ed8982a08f9f.jpg
    - http://www.sharecg.com/images/medium/9067.jpg
    - https://c1.staticflickr.com/1/167/483428291_333db59648.jpg

    - http://2.bp.blogspot.com/-p_2XDye_UG4/UmpPMNhKc1I/AAAAAAAAEy4/CleBQJoGUD0/s1600/Tileable+marble+floor+tile+texture+(26).jpg
- It uses glossy reflection on the bottom of the fruit bowl
- It uses cylinder primitives on the table legs and hemisphere primitives for the bowl
- It uses an area light source as illumination
- It uses multithreading to speed up rendering
- It was rendered using anti-aliasing

My renders are in the following files:

- basic_lighting.ppm: the assignment 3 scene, with a unique colour identifier for each object
- no_spectral.ppm: the assignment 3 scene, with only diffuse and ambient lighting components.
- full_lighting.ppm: the assignment 3 scene, with all three components of the Phong lighting model
- render_area.ppm: my assignment 4 scene, lit by an area light
- render.ppm: my assignment 4 scene, lit by a point source

I think I definitely gained a better understanding of ray tracers through this assignment. The coolest part is that once you have a basic ray-tracing framework working, adding complex new features is often almost trivial. Ray tracing is a really powerful rendering model.