Graphics Homework Assignment 2

Erin McGowan

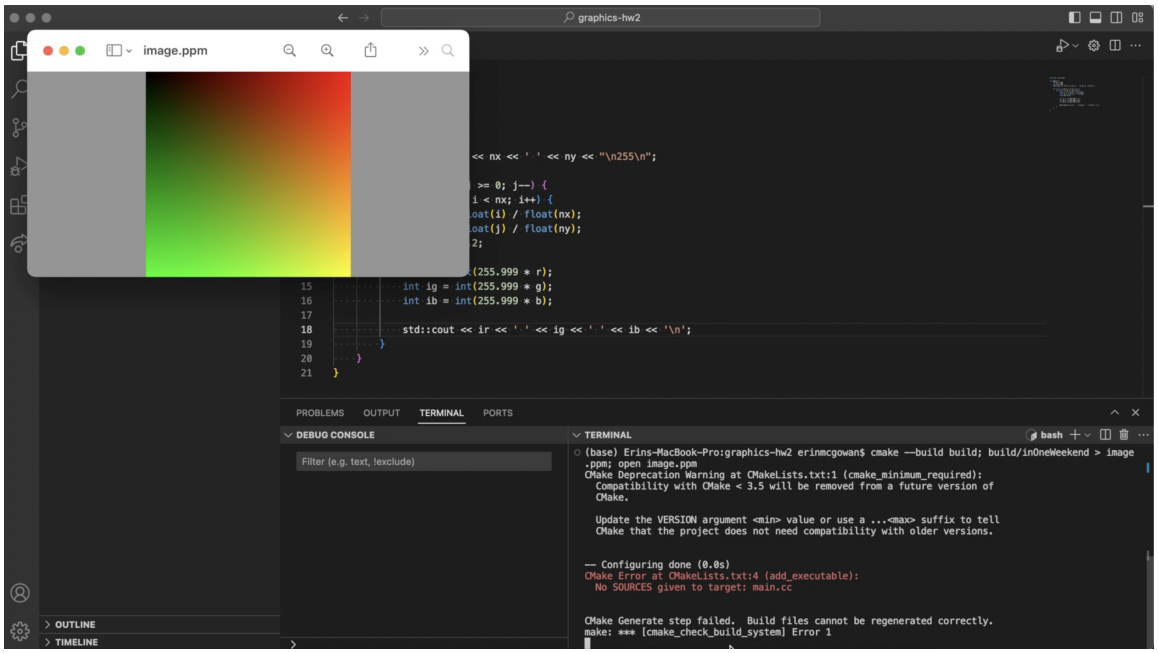
Fall 2023

1 General Information

Operating system: macOS Big Sur version 11.5.2
Compiler: clang, specifically "C/C++: clang++ build and debug active file" in VSCode (as described here: https://code.visualstudio.com/docs/cpp/config-clang-mac)

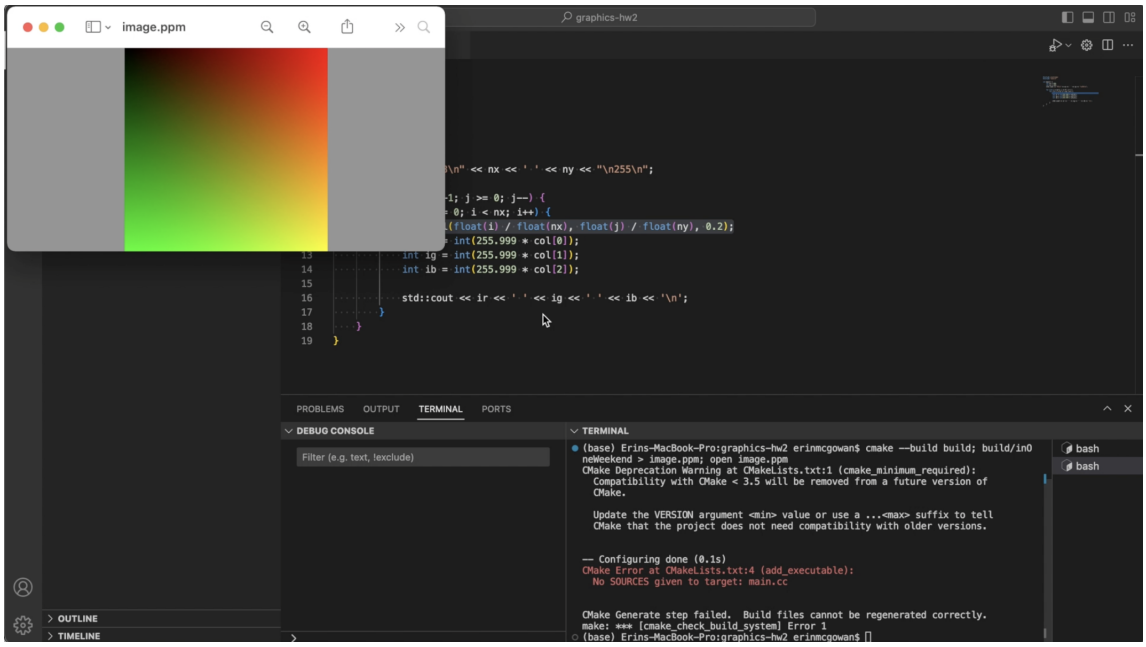
2 Chapter 1: Output an Image

We write all images to a ppm file *image.ppm*. The first line of this file is the image width, nx, and image height, ny. Each subsquent line represents the RGB values of a pixel in the image (which we iterate through left to right, top to bottom), scaled between 0-255. As an example, we set the R value of each pixel to its column within the image divided by the image width, the G value of each pixel to its row within the image divided by the image height, and the B value of each pixel to 0.2. This produces the red-green gradient below.



3 Chapter 2: The vec3 Class

Next, we introduce the vec3 class, which allows us to store triplets of real numbers. We also define a series of vector operations, include addition, subtraction, multiplication, division, scaling, normalization, dot product, and cross product. We can store the RGB values of each pixel in a vec3 object while producing the same gradient as follows:



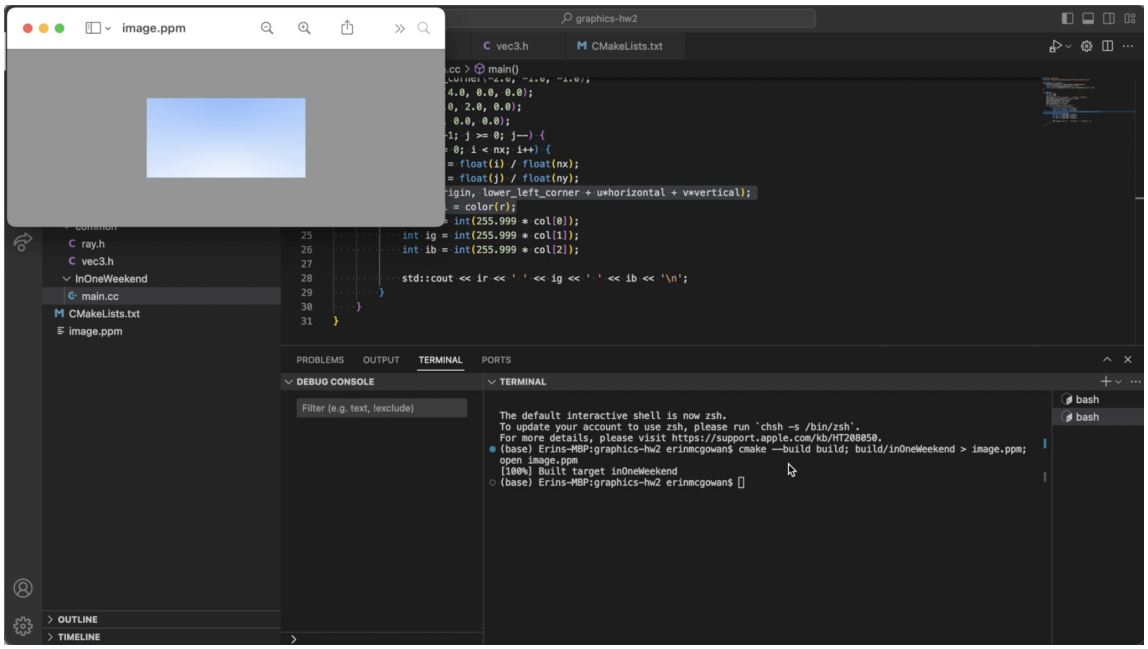
4 Chapter 3: Rays, a Simple Camera, and Background

We now add a "ray" class, which allows us to define a ray by the function

$$p(t) = A + t * B,$$

where A is the origin and B is the direction of the ray. We can use this class to obtain the color of pixels along a given ray.
We experiment with this class by creating a linear interpolation of white and blue (shown below), following the standard form

$$\text{blended\_value} = (1 - t)\text{*start\_value} + t\text{*end\_value}$$



5 Chapter 4: Adding a Sphere

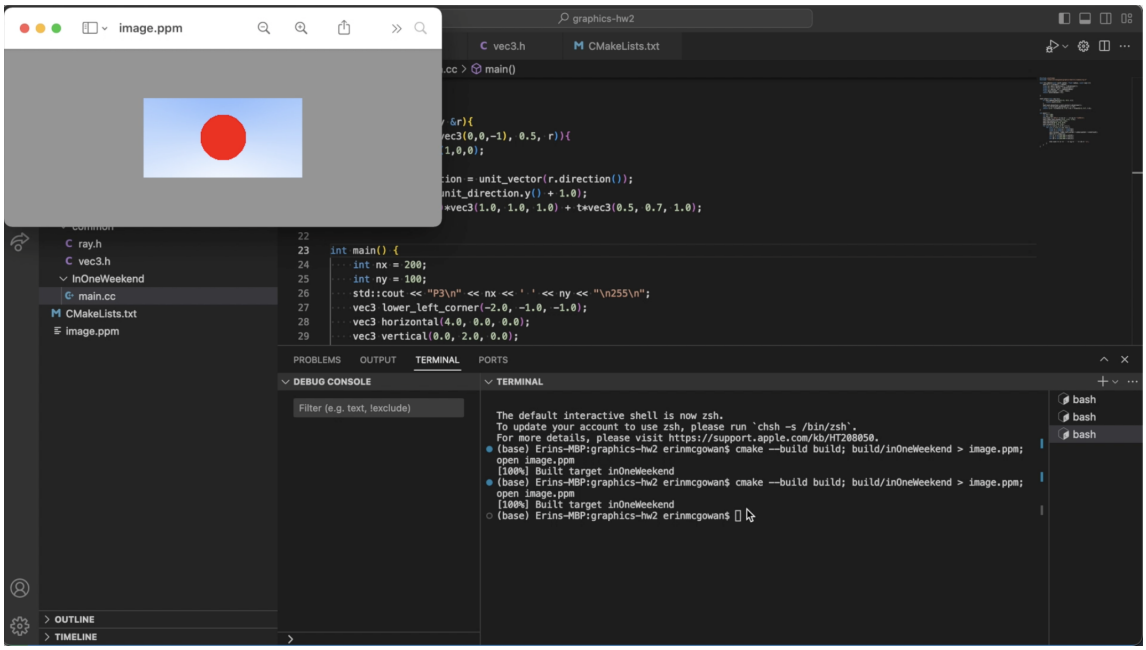
We draw a sphere, using the sphere equation

$$t^2(B \cdot B) + 2t(B \cdot A - C) + (A - C \cdot A - C) - R^2 = 0,$$

where C denotes the center of the sphere (and A and B are carried over from our ray function). If t has zero solutions, the given ray does not intersect with the sphere. If t has one solution, the ray intersects with the sphere at exactly one point. If t has two solutions, the ray intersects with the sphere at two points. Since we do not yet need to know the actual coordinates of these "hit points" (just the number of hit points for a given ray), we compute this information using the discriminant:

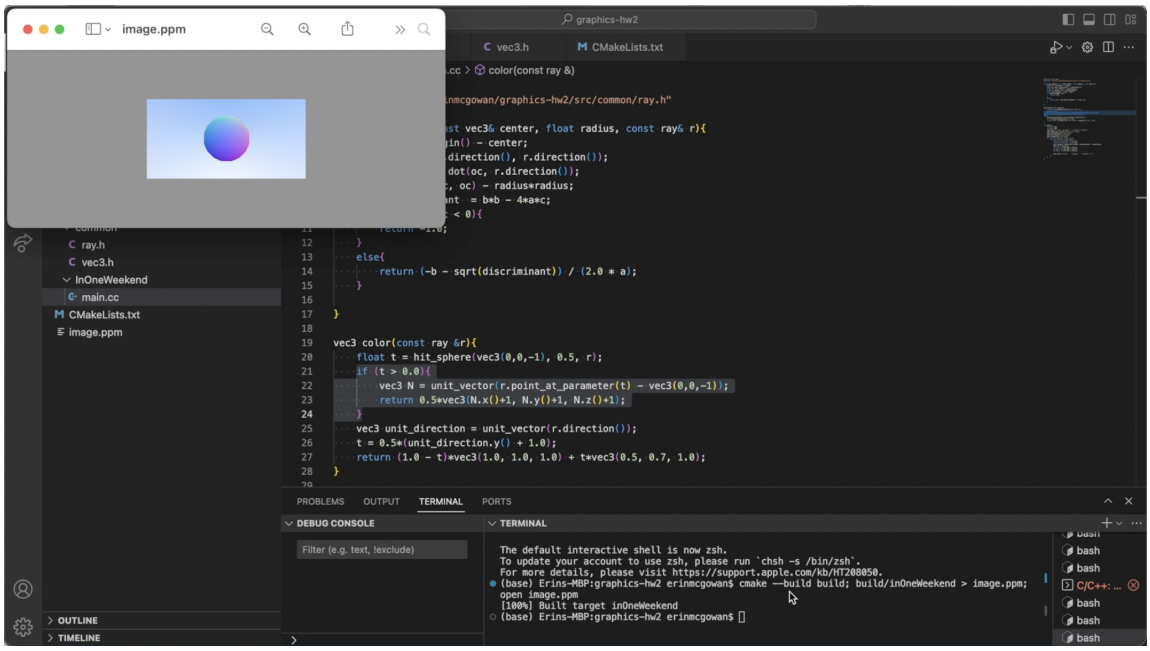
$$\text{discriminant} = b^2 - 4ac.$$

We use this hit information to determine which pixels along each ray are part of the sphere, and color them accordingly:

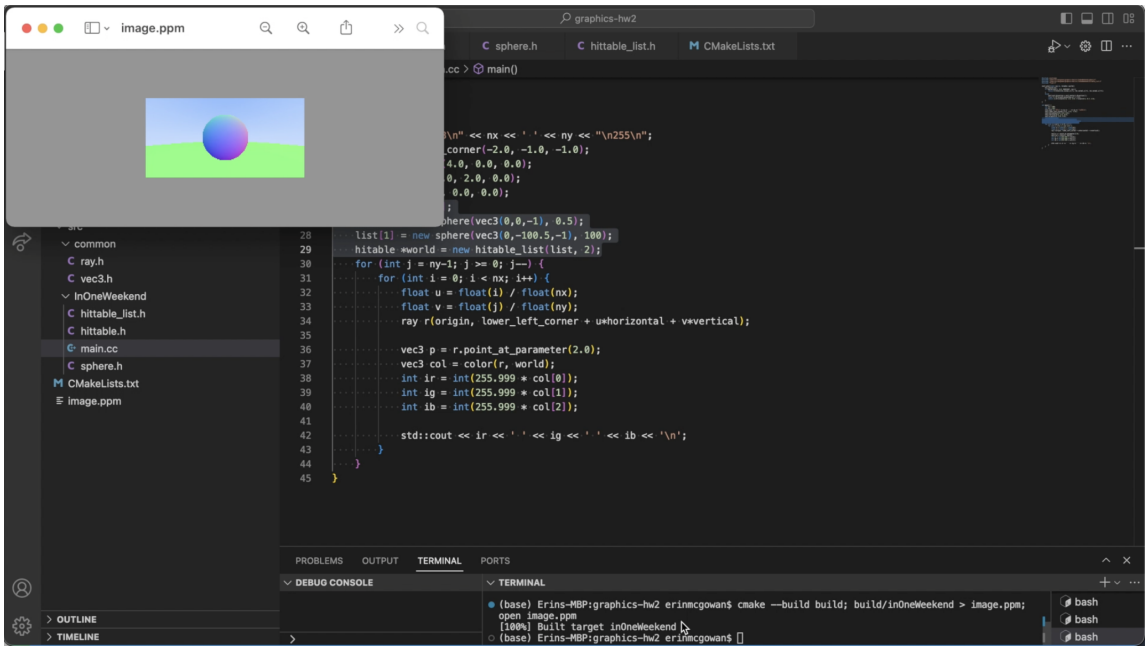


6 Chapter 5: Surface Normals and Multiple Objects

In order to shade the sphere, we must compute its surface normals, which are rays cast in the direction of a hit point minus the sphere's center. This requires us to calculate the actual coordinates of each hit point (rather than just the number of points for each ray). We also normalize each surface normal to the length of a unit vector. These normals allow us to dictate the colors of specific pixels within the sphere, shown below.

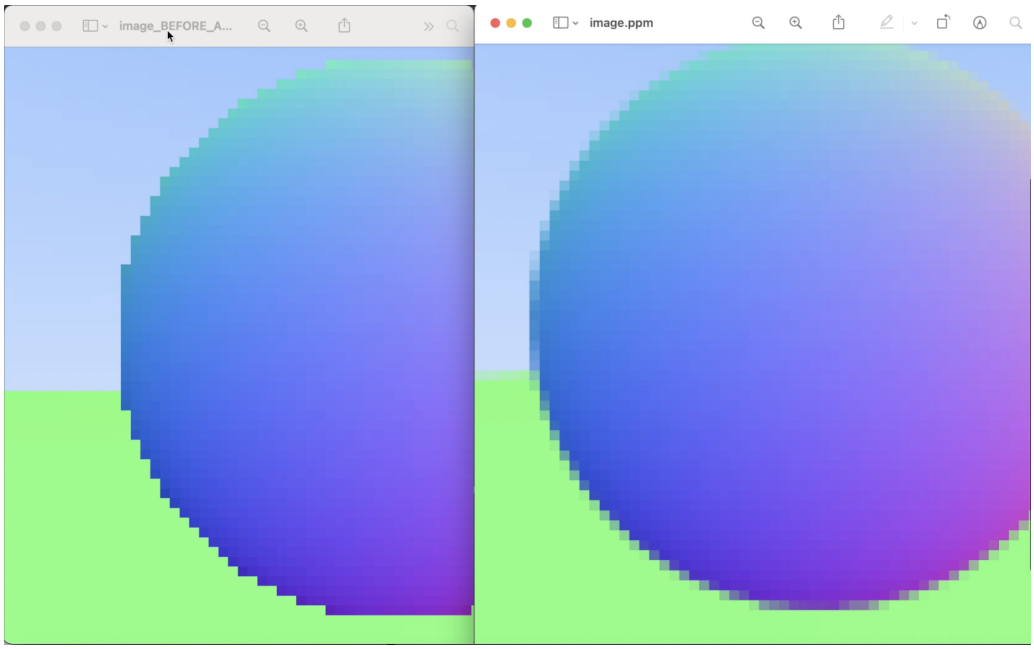


We can generalize this shading technique by creating an abstract class, *hittable*. This, along with its associated *hittable_list*, take a given ray as well as minimum and maximum values for the parameter t as inputs and return a record of that ray's hit points on a given sphere. We may now construct multiple sphere objects with different shading, as shown below.



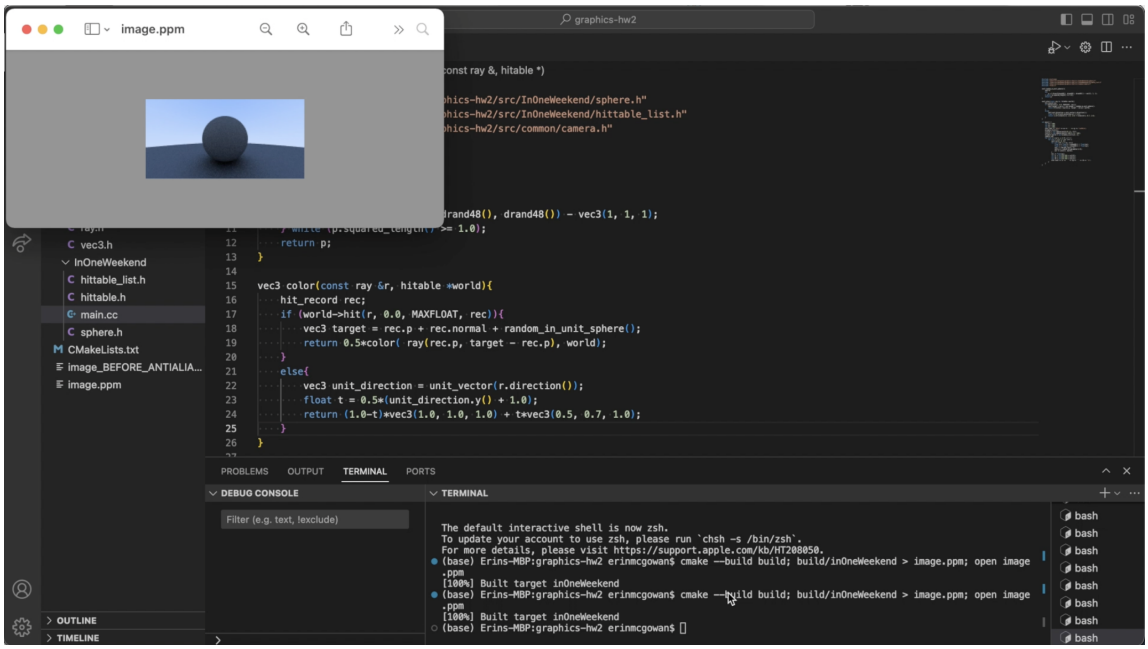
7 Chapter 6: Antialiasing

We now introduce a simple camera, which will allow us to compute the average of the colors returned by several rays intersecting with the same pixel. This allows us to perform antialiasing. In the image below on the right, we see that before antialiasing, the pixels along the edge of the sphere either take on the color of the sphere or the color of the background. However, in the image on the right, we see that after antialiasing the pixels along the edge take on an average of those two colors.

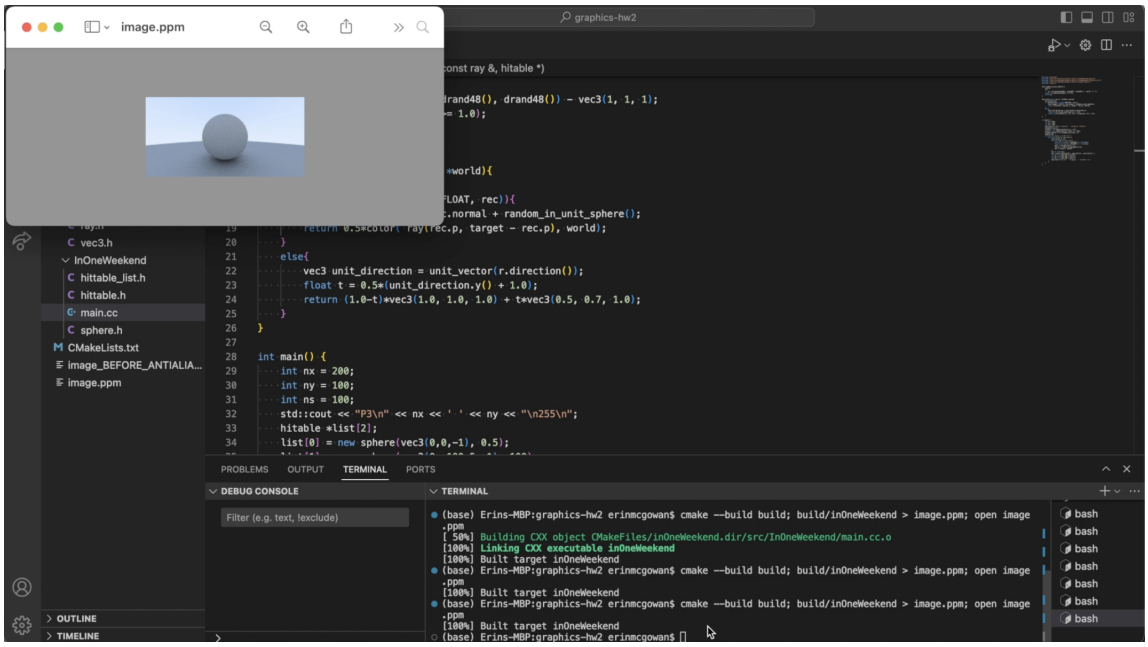


8 Chapter 7: Diffuse Materials

We now imitate the reflective behaviors of rays of light on different materials in order to create the illusion of that material in our shading. We begin with diffuse (matte) material, shading our sphere as follows:

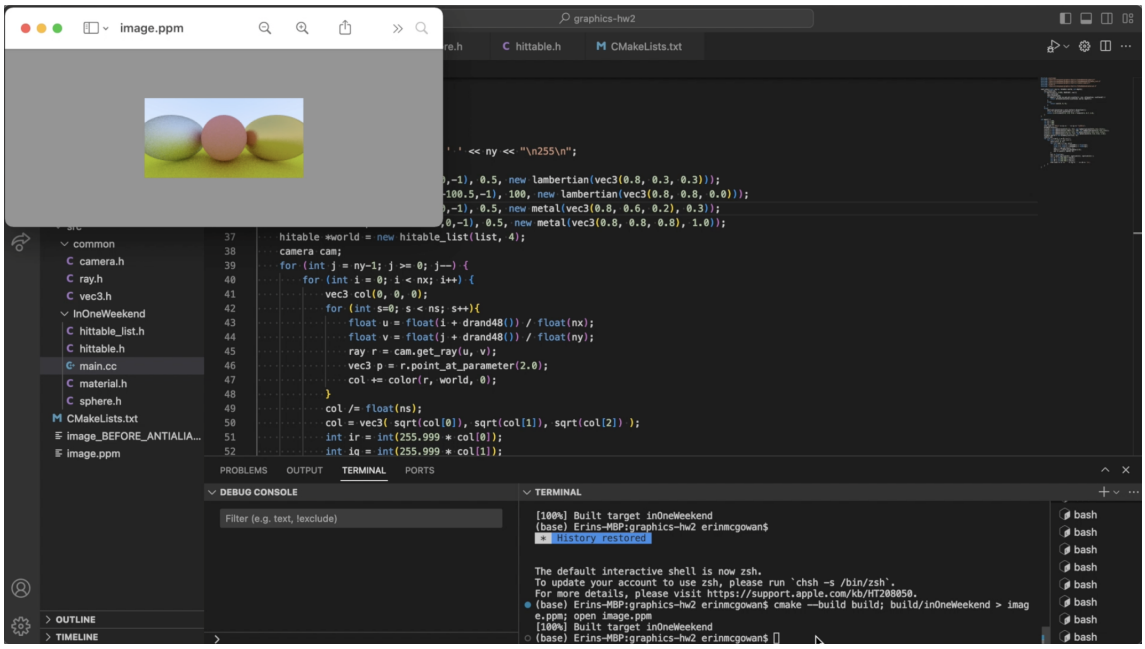


We see that this shading is a bit too dark, and makes it difficult to see the sphere's shadow. We rectify this by gamma correcting the image, taking the square root of each RGB pixel value before scaling between 0-255. We also ignore hits close to zero. This yields the following shading, which is much lighter and contains a more visible shadow.



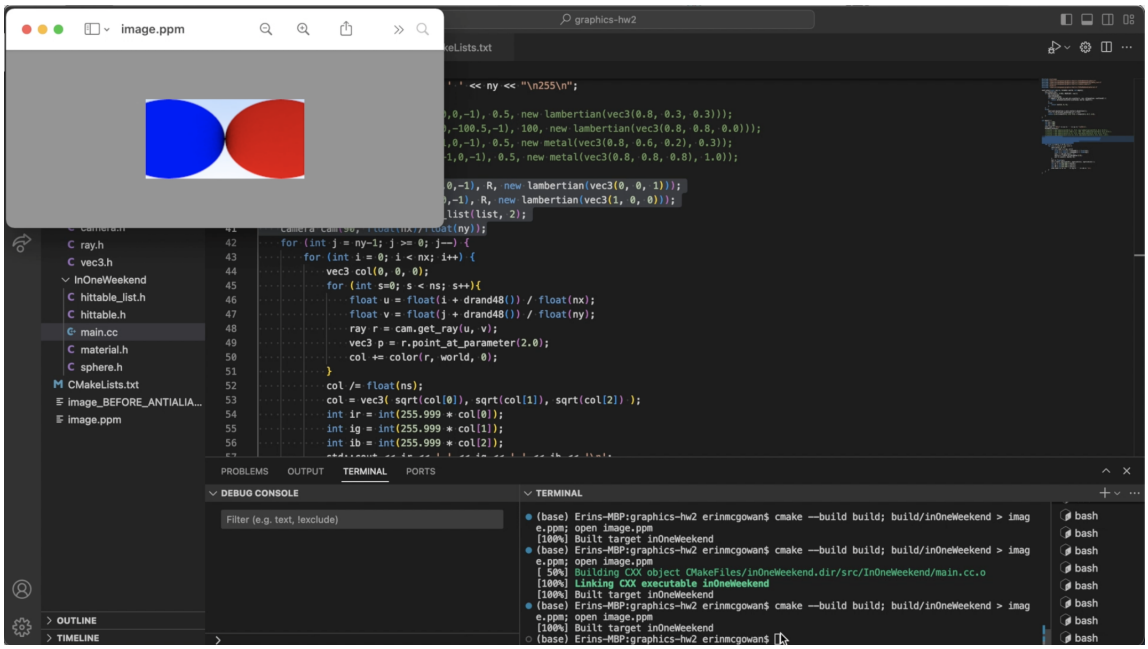
9 Chapter 8: Metal

We can generalize our material shading technique by creating an abstract material class, as well as classes for diffuse material (which mimics lambertian, or ideal ray scattering off of a matte material) and metal material (which randomly scatters the rays it reflects). We add a pointer to the material to both the sphere and hittable classes. This allows us to specify the material and its parameters when we construct a new sphere. Below are two metal and one lambertian spheres (on a lambertian sphere horizon), each with different colors (and in the case of the metal spheres, fuzz values).

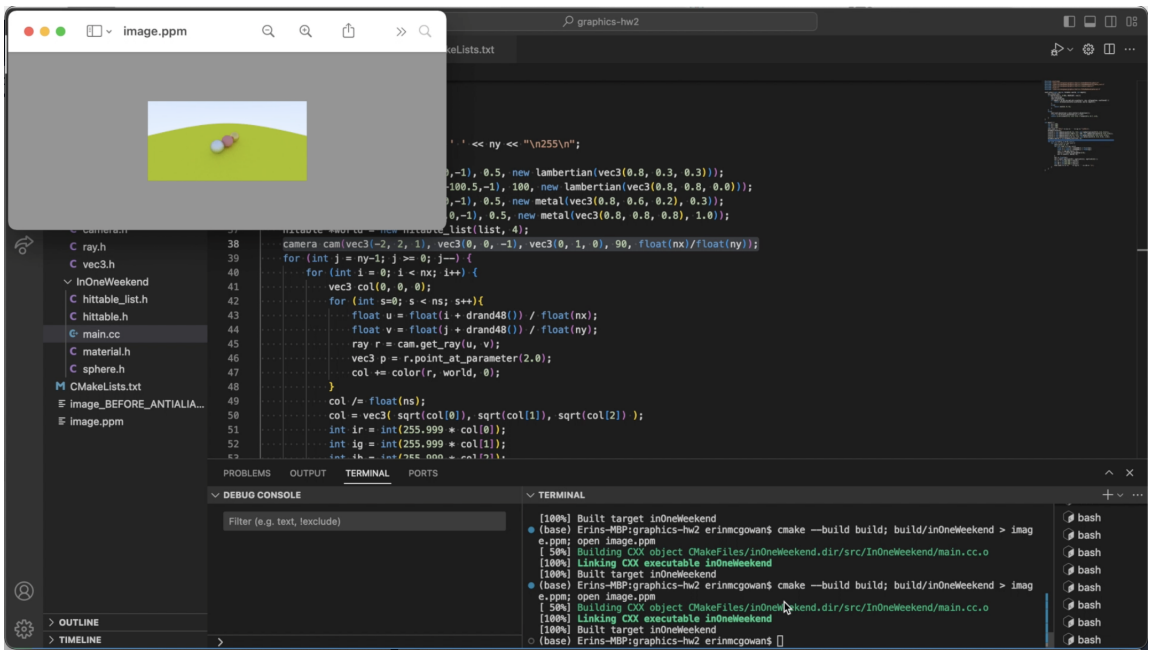


10 Chapter 10: Positionable Camera

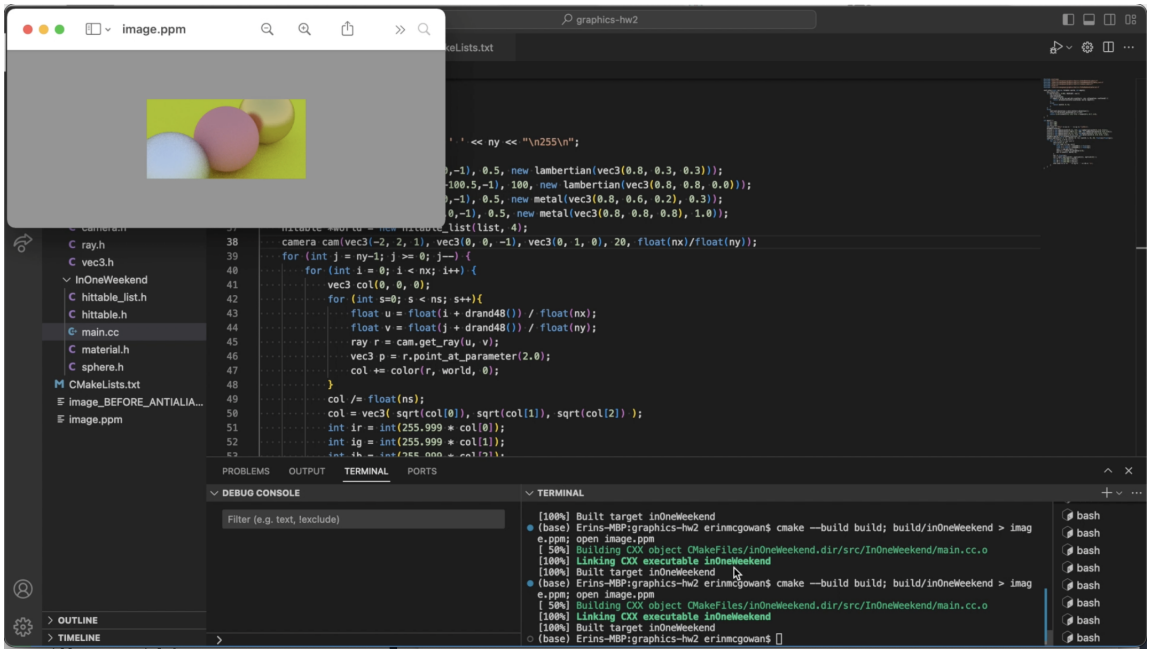
We now specify parameters for both the field of view (we explicitly specify the vertical field of view, but both are adjusted in proportion) and aspect ratio of the camera.



After this, we add the parameter *lookfrom*, *lookat*, and *vup* to the camera, which allows us to position the perspective of the camera within the scene by specifying where the camera is positioned, what it is looking at, and the true direction of "up" within the world, respectively.



We can also lower the field of view to "zoom in" on our spheres.



11 Chapter 12: Putting it all Together

Finally, we can use the functions implemented in the previous chapters to create a function, *random_scene*, which generates a random assortment of diffuse and metal spheres. We use *drand48()* select a random value for *choose_mat*, our "choose material" parameter. If *choose_mat* is less than 0.8, we create a diffuse sphere at a random position and color. If *choose_mat* is greater than 0.8, we create a metal sphere at a random position, color, and fuzz value. We also "clamp" the RGB values for these random colors between 0 and 255 (I added my own clamp implementation in the vec3.h file, as it was not available in the version of c++ I used).
An example of an image generated using *random_scene* is below. We set the image width to 1200, the aspect ratio to 16/9, and the vertical field of view to 20 degrees.

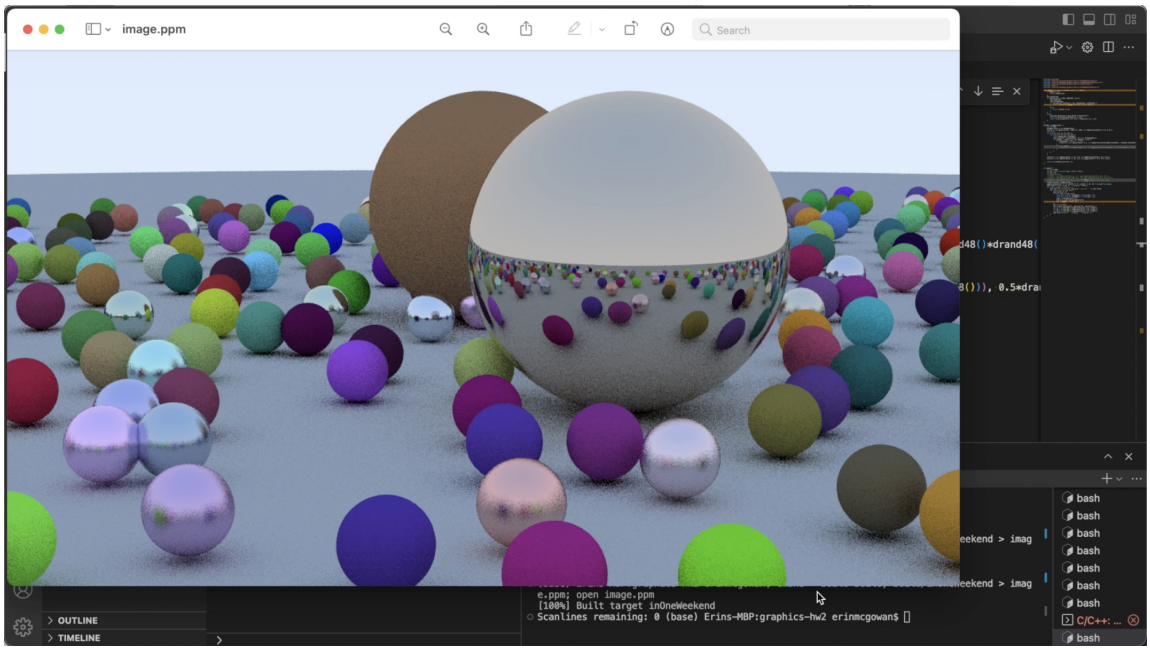


image.ppm
Search
OUTLINE
TIMELINE
e.ppm; open image.ppm
[100%] Built target inOneWeekend
Scanlines remaining: 0 (base) Erins-MBP:graphics-hw2 erinmcgowan$
bash
C/C++: ...