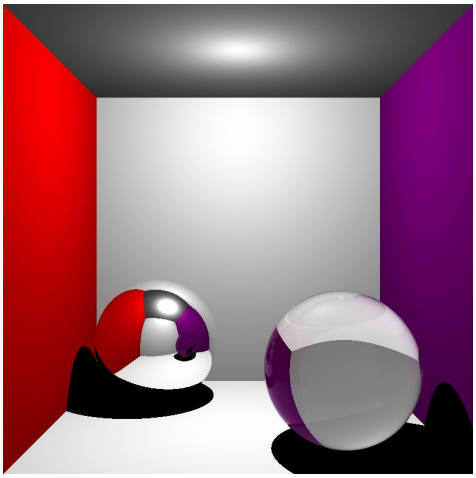
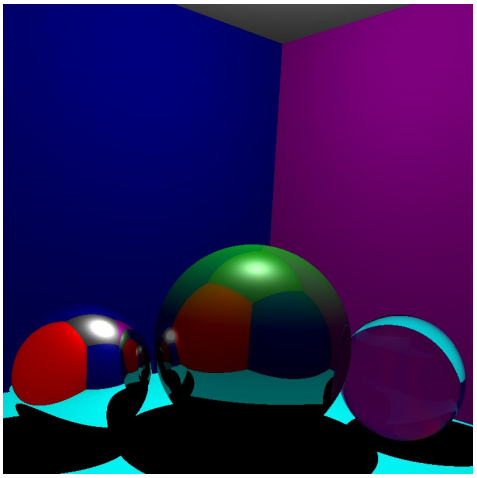
Raytracing Final Project
Due: Dec. 15th (no extensions will be given)

Template code can be found at: https://bitbucket.org/summateaching/raytracing_template.git
(Note that the fragment shader is missing. Use your shader from Assignment 3)

Project: Create a Whitted-style raytracer.

Turn in: Code + 3-4 raytraced images with varying scenes and/or views

Template Code: I've provided some template code that you can use for this assignment, but you are not required to do so. All of the functions that I used to construct my raytracer are included although the pertinent code has been removed (look for TODO comments in the code). This code base is similar to your HW3, but you'll need to adjust more than that assignment so I'll document some important aspects.

- **ObjMesh** class is your mesh loader from HW3 but now has a subdivision sphere as a potential mesh
- **Object** is a super class for all the scene's objects. It contains:
    - The mesh needed for OpenGL rendering
    - Attributes used for local shading
    - Matrices like C (modelview), C^{-1}, $(C^*)^{-1}$, and $(C^{-1})^T$. These are computed by the setModelView function
- **Sphere** and **Square** are child classes of **Object**.
    - Each contain an intersect function that you'll need to complete these functions.
    - Each contain a more general ray/sphere and ray/square intersection routines that are used by the intersect function. You'll need to complete these functions.
- I tried to keep most of the important code main.cpp:
    - *rayTraceReceptor* and *write_image* are functions that create the output png and can be ignored

- *initGL* and *drawObject* are handing the OpenGL window and can be ignored.
- *findRay* and *rayTrace* are provided for you. *findRay* provides the ray start and direction for a pixel with result is returned as a length 2 vector (ray_start, ray_dir). *rayTrace* calls *findRay* for every pixel in your raytraced image, calls *castRay*, for each and outputs a png with the results.
- As you can see, *castRay* is the recursive function that is the major chunk of this assignment. It's input is the ray start, the ray direction, the last object hit by the ray (think about why this might be important), and the depth of the recursion.
- *shadowFeeler* sends a shadow ray to any point lights from a point. It also takes as input the object casting the shadow ray (think about why this might be useful).
- I have also included a *castRayDebug* function that I used to cast a single ray in order to look at the output it produces. It might be useful for you as well, to create something similar. (no code is provided)

Template Scenes: I have provided 3 example scenes a sphere, a cube, and a cornell box: 6 sides, two spheres (one mirrored and one glass-like), and a single point like at the center of the ceiling. You will be graded on the cornell box scene.

Milestones: Below I've given milestones on how I would progress to create a raytracer. Remember, too much code all at once is a recipe for failure and frustration. Incremental updates with testing ... always. I'm giving you weeks to complete this assignment because that's how long it may take. DO NOT PROCRASTINATE.

1. Replace the template scene with a single diffuse sphere sitting at the origin. No user input yet. Note that diffuse means no recursion yet.
    a. Raytrace a flat shading (single color) to test your intersection calculations
        i. Test moving and scaling the the sphere's position to make sure it's working with transformations.
        ii. Add another sphere, test rotations.
    b. Add a point light source and Phong shade the sphere.
        i. Test moving the light around and moving/scaling the the sphere to make sure it's working
2. Do the same thing but now for a diffuse unit square (XY plane) at the origin.
3. Test 1 and 2 now with user rotate, scale, translation of the scene.
4. Create (or use my) cornell box scene with only diffuse and phong specular shading.
5. Add shadow rays
6. Make the back sphere pure specular (reflective). Recursion now necessary.
7. Make the front sphere glass transparent. You can use user defined transmissive and specular values, or use Fresnel equations to determine the proper physically accurate values. (used in example images above).