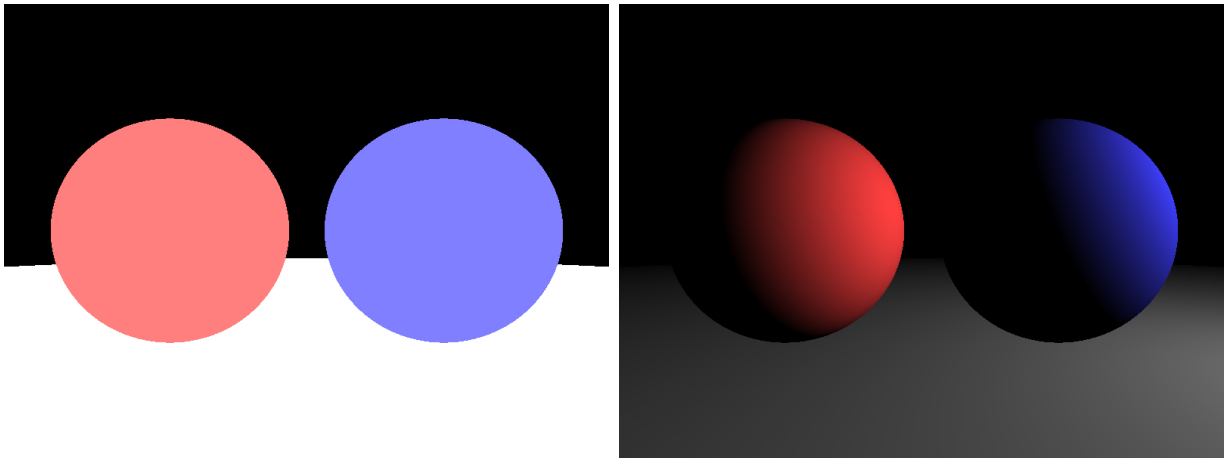
CPSC 478/578: Computer Graphics	Feb. 26, 2020
Spring 2020, Assignment 4	

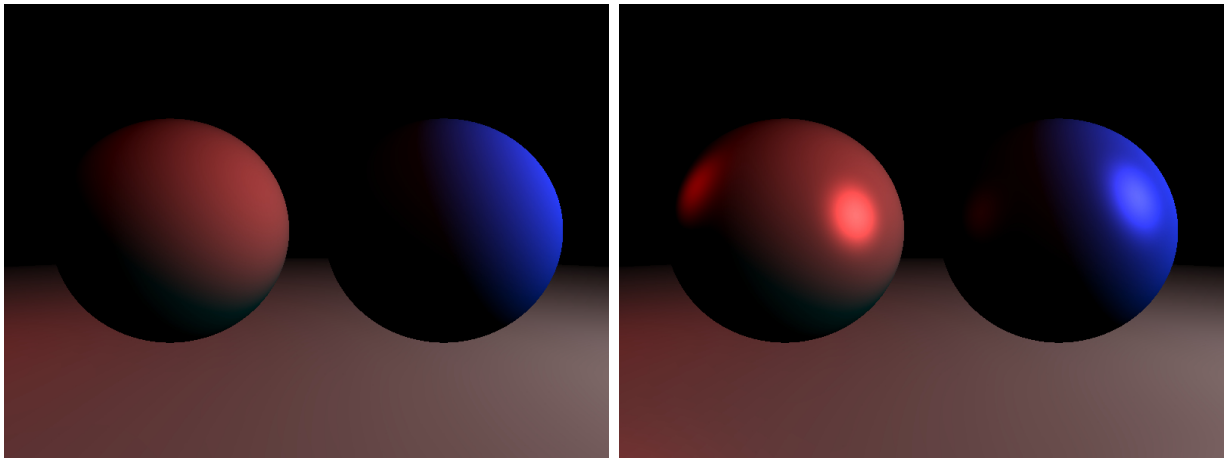
Due: Wednesday, March 25, 2020 at 11:59 PM

Homework 4: Make Your Own Ray Tracer



(a) Ray-Sphere Intersection (the floor is also a big sphere), `2.png`

(b) Diffuse Shading, `3.png`



(c) Multiple Lights, `4.png`

(d) Specular Highlights, `5.png`

Figure 1: The first four components of Part I (§1), when they are working.

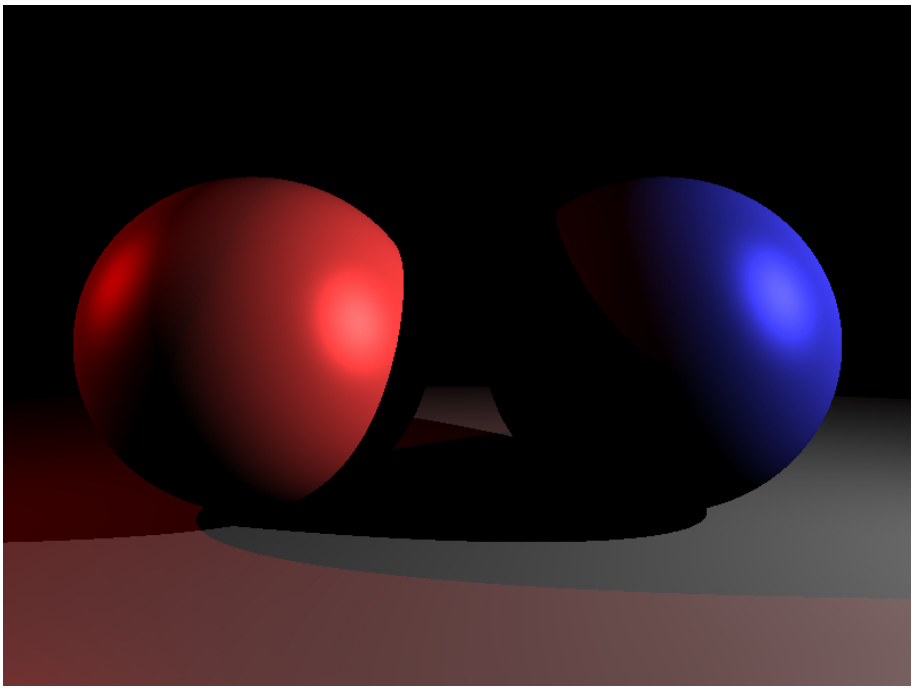


Figure 2: Part I (§1) after shadows are working, `6.png`

1 Ray Tracing, Part I

In this assignment, we will explore the other major rendering algorithm, *ray tracing*. To start, we will implement eye ray generation, ray-object intersection, and basic shading. In the next assignment, we will look at more advanced features such as reflection, refraction, and distribution effects. By the end of this component of the assignment, you should have all the features in Figs. 1 and 2 working.

The camera parameters for this scene are:

```
resolution = 800 x 600
eye = (0, 0, 0)
looking at = (0, 0, 1)
up = (0, 1, 0)
near plane = 1
field of view (y direction, top and bottom of the image) = 65 degrees
```

The scene geometry is:

```
Sphere 0, center = (-3.5, 0, 10), radius = 3, color = (1, 0.25, 0.25)
Sphere 1, center = (3.5, 0, 10), radius = 3, color = (1, 0.25, 0.25)
Sphere 2, center = (0, -1000, 10), radius = 997, color = (0.5, 0.5, 0.5)
```

```
Light 0, position = (10, 3, 5), color = (1, 1, 1)
Light 1, position = (-10, 3, 7.5), color = (0.5, 0, 0)
```

The Phong exponent everywhere is `10`, and `color` refers to the albedo of both the specular and diffuse components.

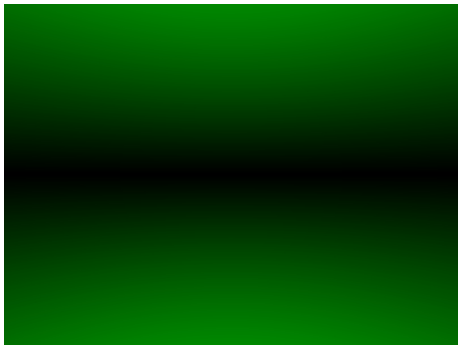
1.1 Eye Ray Generation

Writing and debugging the initial eye ray generation stage can be tricky if you do not have a clear idea of what the final result should be. One common strategy is to write the direction of the ray out to an image. However, the camera parameters of the example scene yield a fairly predictable image. If you draw the various components of the normalized eye rays, you should obtain images similar to Fig. 3.

(a) Raw `x` component, `1x.png`

(b) Absolute value of `x` component, `1xabs.png`

(c) Raw `y` component, `1y.png`



(d) Absolute value of `y` component, `1yabs.png`

Figure 3: Examples images when eye rays are working correctly.

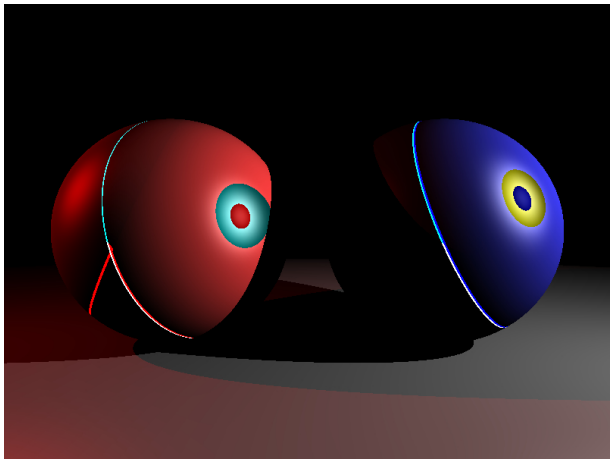
If you are not getting similar results, ask yourself why these images take on these particular structures.

1.2 Next Features

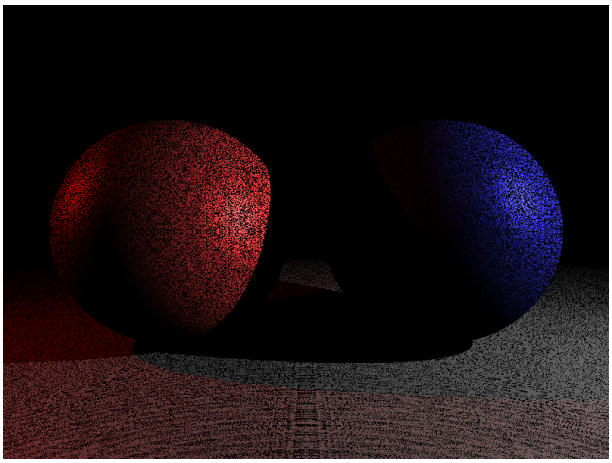
- **Scene intersection:** For this assignment, implement the ray-sphere intersection algorithm described in class. If multiple intersections are detected for a single pixel, use the closer one.
- **Shading:** Implement the diffuse and specular Phong shading model described in class. The book also has a slightly different "half-angle" formulation for the specular component which you are welcome to use instead.
- **Shadows:** You should be able to re-use your scene intersection code for shadow rays as well.

1.3 Debugging Tips

- While you do not need to compute any matrices for this assignment, you may find the code you used to compute the top and bottom edge of the viewing frustum to be helpful when generating the eye rays.
- Trying to get all of the components of a ray tracer working simultaneously is a classic beginner's mistake. Start by trying to reproduce the eye ray images from section 1.1, and then the sequence of images on the first page. Make sure each incremental feature works before writing any code for the next component.
- If strange bright regions appear in your image,



  then check the bounds on your final output values.
- If you are trying to add shadows but run into an image like this,



then you have encountered the 'shadow acne' problem. The good news is that your shadow code is almost working. The bad news is that you are going to have to think a little harder about why this is happening. The book offers some advice.

2 Ray Tracing, Part II

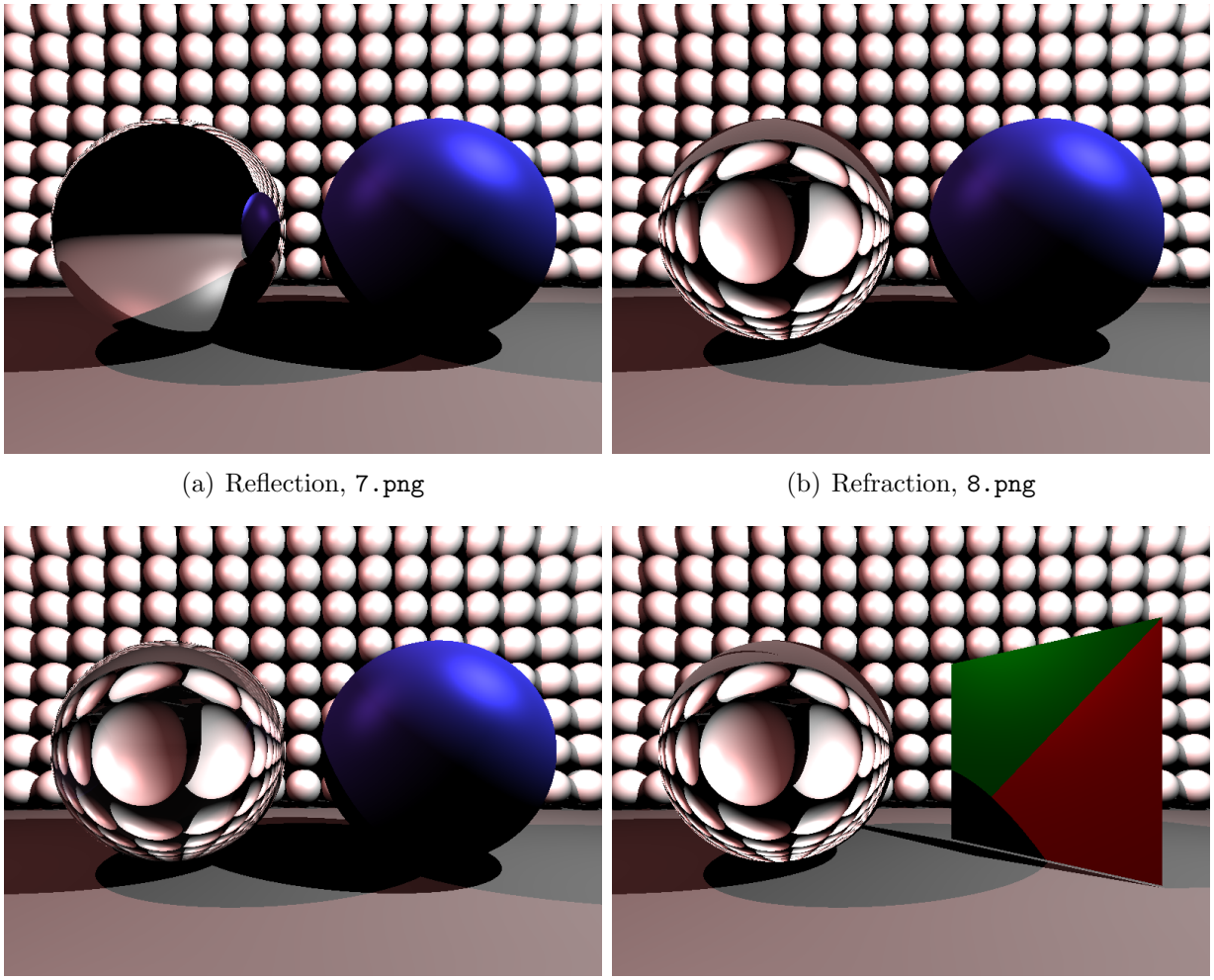
Now that our basic ray tracer is working, let's add some more features. In particular, we will look at reflection, refraction, and triangle intersection. By the end of this section, you should have Figs. 4 and 5.

The camera parameters for this scene are the same as the previous section. The lights are at `(10, 10, 5)`, `(-10, 10, 7.5)`, and respectively colored `(1, 1, 1)` and `(0.5, 0.25, 0.25)`. When present, the sphere positions are the same as before. Other features of the scene are:

- The index of refraction for the right sphere is $\alpha_{air} = 1$, $\alpha_{glass} = 1.5$. You do not need to do any exponential attenuation.
- The triangles form a `(6 x 6)` square centered at `(3.5,0,10)`. It has been rotated 45 degrees along the y-axis towards the glass sphere.
- The wall of white spheres at the rear of the scene is a `(20 x 10)` grid of spheres each with a radius of 1. They all have color `(1,1,1)` and Phong exponent 10. The sphere at the lower right corner of the grid is centered at `(-20, -2, 20)`.

  The remaining spheres are offset from this initial one such that they all barely touch ("kiss") each other. E.g. the next sphere to the left is centered at `(-18,-2,20)`, and the next sphere on top of the initial one is centered at `(-20,0,20)`.

Read Chapter 13 from Shirley and Marschner, particularly 13.1. Also revisit Chapter 4, particularly 4.4.2. Start from your implementation of Assignment 4. In order to rotate the vertices of the triangle, you may find it necessary to use the `MATRIX3` class. Alternatively, you could always compute these by hand.



(a) Reflection, `7.png`

(b) Refraction, `8.png`

(c) Fresnel Effects, `9.png`

(d) Triangle Intersection, `10.png`

Figure 4: The first four components of Part II §2 when they are working

2.1 Debugging Tips

- With the increased number of spheres in the scene, the rendering times will no longer be instant. It is **highly recommended** that you perform lower-resolution renders while debugging so that you get instant feedback. If you are finding yourself twiddling your thumbs or glancing at your browser in the ten seconds it takes to render an image, it may be time to re-evaluate your development cycle.

- You will encounter a problem similar to the 'shadow acne' problem from the previous assignment when trying to implement reflection and refraction. It can be dealt with in a similar manner.

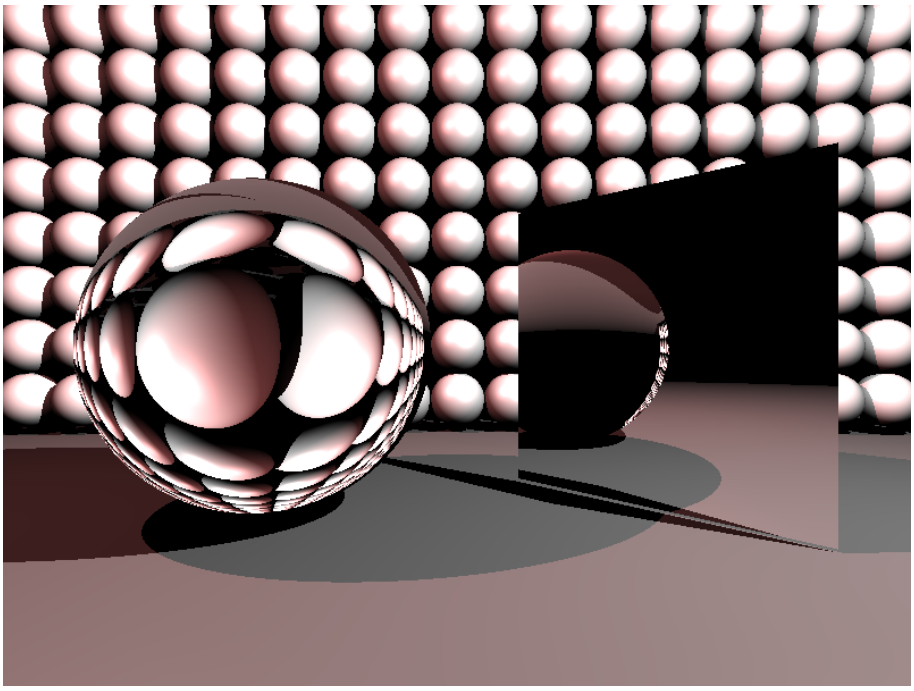


Figure 5: The triangles can serve as mirrors as well. The maximum recursion depth has been set to 10 in this image. Details will disappear if you set it to something lower. This should be your `11.png`.

- For several of the features you are implementing, the normal computed by your existing code may not actually point in the direction you want or expect. You may find it helpful to compute a `facingforward` version of the normal, whose dot product with respect to the incoming ray direction is always negative.

3 578 Only: Mandelbrot Texturing

Texture the quad from §2 using the Mandelbrot Set. Don't apply an image texture to the quad, compute it from scratch by interpolating a texture coordinate from the vertices and performing the $\mathbf{q} = \mathbf{q}^2 + \mathbf{c}$ iteration.

4 Formatting Instructions

- Turn in a ZIP file named LastName_FirstName_Assignment 4.zip.
- Unzipping should produce the `*.cpp` files, a shell script `render.sh`, and all your generated images in `*.png` format.

- If your `main.cpp` generates all images as PPM files, `render.sh` can simply be

  ```
  make
  ./run
  mogrify -format png *.ppm
  ```

  You can modify `render.sh` accordingly if you have multiple `*.cpp` files to generate images separately. Note that the `mogrify` command is from ImageMagick, which is installed on the Zoo machines, but might not be on your local machine.

- Make sure the image filenames exactly follow the format given in the points breakdown.
- Include a README.txt if necessary to explain why some pieces of your answer work while others do not.

5 Points Breakdown

The assignment will be graded *on a Zoo lab machine*. Make sure your code builds there.

- **(5 points)** `1x.png`, `1xabs.png`, `1y.png`, `1yabs.png`: Eye ray generation is working
- **(10 points)** `2.png`: Scene intersection is working
- **(10 points)** `3.png`: Diffuse shading is working
- **(5 points)** `4.png`: Support multiple lights
- **(10 points)** `5.png`: Specular shading is working
- **(10 points)** `6.png`: Shadows are working (half credit if they contain acne)
- **(10 points)** `7.png`: Reflection is working
- **(10 points)** `8.png`: Refraction is working
- **(5 points)** `9.png`: Fresnel effects are working
- **(10 points)** `10.png`: Triangle intersection is working
- **(5 points)** `11.png`: Reflection is working for triangles
- **(10 points, 578 only)** `12.png`: Mandelbrot texturing is working.
- **(8 points)** Follow the formatting instructions.
- **(2 points)** At the top level directory, include a `HOURS.txt` that lists approximately how many hours you spent on this assignment. There is no right answer; we will only be analyzing this data anonymously.

We will be running the Measure Of Software Similarity (MOSS) on the submissions for this assignment. Don't borrow code from your classmates.