COSC 2P91 – Assignment 4 (Mini-Project 2) – Penultimate Stage

(Note: This is a placeholder until the entire project is up)
Now that you've got a threaded project capable of rendering shapes, it will be a fairly easy matter to create a modified version that can achieve basic Ray Tracing.

For this part, you now much make either an option for your program or (probably better) a separate program that can render a scene. The requirements are as follows:

- You'll be incorporating lights
- You must have ambient, diffuse, and specular shading (explained below)
- You must have ***one*** of:
  - Reflection – i.e. recursive ray casting back into the scene
  - Refraction – same idea, but transmission *through* objects
  - Shadows – *recommended* – when Object A is in between a light and Object B, Object B receives no illumination from that light
- A scene may have any number of lights or shapes
- A scene may have any number of *materials*
- Each shape will have a material assigned to it
- If you choose to go with reflection or refraction, you'll need some safety mechanism to avoid infinite recursion. You may have the maximum recursion limit defined by the user, *or* you may use a hardcoded limit of 5
- If you're feeling ambitious, you *may* add the option for planes and triangles (or any other two shapes, e.g. cylinders, etc.) for a small bonus

Background Info:
First, a disclaimer: remember that this is a number-crunching problem. As such, many simplifications are included relative a true (traditional) ray tracer. Materials, specular shading, etc. will all be greatly simplified.

Before we get into the details, first know that, if you haven't already achieved this in the second part, you *do* need to determine the *closest* point of intersection (and the corresponding shape, of course). The math for doing so is included in the second part.

Lights are the simplest things to explain first:

- A light is a source of light in the scene
- In real life (and real ray tracers), lights may have numerous properties, including direction, etc.
  - For this task, just give each light a position in (X,Y,Z), and a single colour – e.g. (1.0,1.0,1.0) for white light
- You'll also need a special additional light called *ambient*, which is identical to the others, but doesn't have a position
  - If you like, you're more than welcome to simply have your program just ignore the position of the first light it's given in a scene file

Materials are similarly easy:

- A material roughly describes how a shape looks. e.g. its colour
  - Technically, it also describes things like how diffuse the shape is, how reflective, etc.
  - You have two choices on how you could approach ambient, diffuse, and specular:
    - Option 1: Separate values for all
      - Ambient-red, ambient-green, ambient-blue, diffuse-red, etc.
    - Option 2 – recommended: A single colour, and a coefficient
      - red, green, blue, and then numbers from 0 to 1 of the rest

    - You may also have an *index of refraction*, or a *shininess*, depending on which option you choose
- I'd suggest creating a single listing of materials, and then assigning a *material index* to each shape

Ambient lighting is the simplest lighting:

- It's a simplification of the light that tends to bounce off all of the surfaces within an environment to provide indirect lighting. There are different ways to calculate this, but the simplest is:
    - Multiply the colour of the material by the colour of the ambient lighting
    - e.g. if the material is (0.5,0.0,1.0), and the ambient lighting is (0.2,0.2,0.2), then the ambient portion of the pixel's colour will be (0.01,0.0,0.2) – roughly a very very dark indigo

For example, two intersecting spheres with only (very faint) white ambient lighting:

(I'm not sure how this will look on your monitor, but this totally isn't just a black rectangle; for seriouses)

Diffuse shading is slightly more complicated:

- Consider a piece of chalk
    - It's pretty visible, even in relatively dim lighting. That's because it reflects quite a bit of light
        - We don't think of it as being reflective, because we can't see ourselves in it, but it's actually reflecting a great deal of light. It's simply doing so in a *scattered* fashion
- To simulate this type of reflection, we typically rely on what's known as *Lambertian reflectance*
    - All we really care about is how directly the light is hitting the surface
        - If it's shining straight at the surface, it will be brightly illuminated. If the light is only glancing off, nearly tangential to the surface, it will be very dim

The math is pretty simple:
First, find the point of intersection.

- You already know the point of origin of the ray, as well as its initial direction, as vectors. From Part B, you also know the distance.
- Remember $\overline{X}=\overline{X}_0+t\overline{D}$? Since you have already solved for t, just calculate $\overline{X}$

Next, find the *unit normal* to the sphere's (or shape's) surface at that point:

- The normal for a sphere is easy; just extend a ray from the sphere's centre through the point
- A *unit* vector has simply been scaled to have a length of one

$$\vec{N}=\frac{\vec{X}-\vec{C}}{radius}$$

- In other words, Nx=(Xx-Cx)/radius, etc.

Next, you need a unit vector from that point to the light. The (non-unit) vector is:
$\overline{L}vector=\overline{L}pos-\overline{X}$

The unit vector is:

$$\vec{L}\,unit = \frac{\vec{L}\,vector}{|\vec{L}\,vector|}$$

(as a reminder, the length of a vector is simply the square root of the sum of the squares of the values)

Now, all we need to do is to determine the influence of diffuse lighting, based on the unit vectors towards the light and from the normal:

$$Multiplier = \vec{N} \cdot \vec{L}\,unit$$

There's more than one way to compute such a dot product, but the ridonkulously easy way is to just sum the products of the components.

- i.e. Nx*Lx+Ny*Ly+Nz*Lz

Finally, the actual colour components from diffuse shading are calculated as such:
($kd*Multiplier*Material_{red}$, $kd*Multiplier*Material_{green}$, $kd*Multiplier*Material_{blue}$)

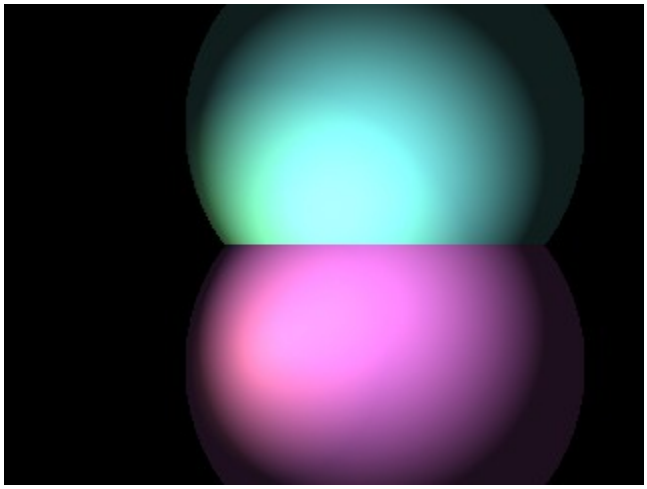
- where kd is simply a coefficient for diffuse reflection
- as mentioned above, you can have the diffuse coefficient also be a vector, with red, green and blue separately defined, but there's really no point for this level of rendering

As an extra note: you'll probably want to put in a check when computing your Multiplier. If the light is coming from the wrong side (i.e. shouldn't be illuminated anyway), it can be a negative value, which makes no sense.

- Basically, if your Multiplier is less than zero, set it to zero

Important: This is calculated for ***each*** light! For multiple lights, just add them together

- Similarly, this gets added onto the result of the ambient lighting as well



Specular reflection is slightly more complicated than diffuse, but can reuse some of the work you've already done for it.

- It represents the glare of having light reflecting from the light and right into your eyes
- This time, the angle of perception actually matters
- Basically, we just need the unit normal (which we've already calculated), unit vector to the light source (which we've also already calculated), and the *reflection unit vector* of the vector from the eye

To calculate the reflected vector:

$$factor = \vec{D} \cdot \vec{N}$$

$$\vec{R} = \vec{D} + 2 * factor * \vec{N}$$

Of course, we already know how to normalize this unit from above. Next:

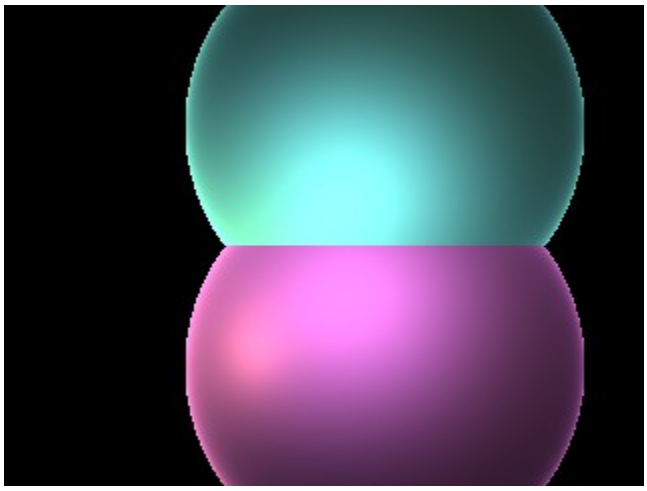
$$Multiplier = (\vec{R} \cdot \vec{L}unit)^{se}$$

where `se` is just some specular exponent (I'd just use 2).

Finally, we get:

($ks*Multiplier*Material_{red}$, $ks*Multiplier*Material_{green}$, $ks*Multiplier*Material_{blue}$)

- where ks is a coefficient for specular reflection, but again *can* be more complicated if you for some reason so desire



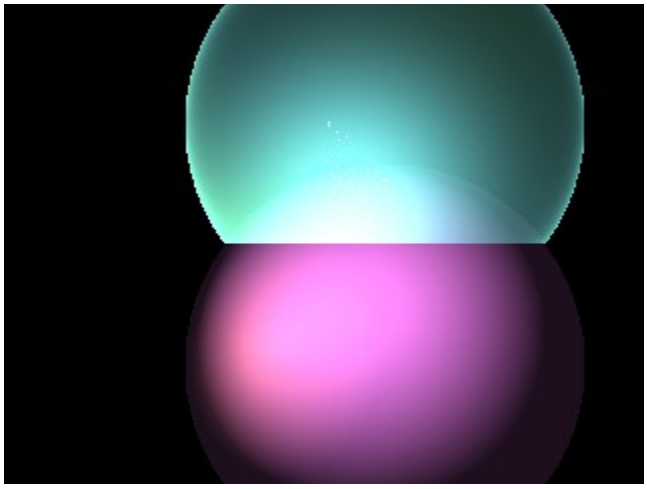
- As you've probably noticed, the simplified math we're using sometimes causes special cases for extrema – in this case, tangential collisions get a bit wonky. It isn't a cause for concern here

For our feature, again, we have a choice: reflection, refraction, or shadows:
It's really easy to add **shadows**:

- Remember how you're calculating *diffuse* and *specular* lighting for each light source? Well, before doing so, check if a ray from the point of intersection to that light source would intersect with any other shapes in the scene. If so, skip the diffuse and specular lighting for that light (for that ray).
- You may need to be careful that the test doesn't actually intersect the shape you're calculating for

Reflection is also relatively easy:

- Remember how you already calculated a vector that's the reflection of the 'eye's ray?
  - You could use that for an additional (recursive) fired ray
- If that ray were to then strike another reflective surface, it could fire its own ray
  - Again, it's just simple recursion
  - This is why you'd need a 'cap' or limit on the maximum recursive depth
- If you do this, each material will need an additional 'shininess' or 'reflectance' property, to determine how much colour from the new ray should be added onto the current pixel/ray intersection
  - If the shininess is zero (or 1.0, depending on how you scale and interpret it), and you know there won't be any apparent reflection, you don't normally bother firing out the additional rays



- Note that, for this shot:
  - Only the top sphere is reflective
  - The top sphere is only using specular shading
  - The bottom sphere is only using diffuse shading

If you want, you can add **refraction** instead.

- Because it requires using two *indices of refraction* for each intersection, it's certainly more complicated, and really isn't advised

Tips:

- For ambient, rather than explicitly describing both the ambient properties of the materials *and* the ambient lighting, I'd just define the ambient lighting, and multiply that by the material's colour
- It's very easy to get values that would exceed the legal bounds for colours
  - e.g. too many lights could potentially try for 'whiter than white'
  - Just ensure that you cap each value at 1.0/255
- In terms of your actual pixel rendering (i.e. the casting of a single ray), you might find it more convenient to simply declare a red, green, and blue, and pass pointers to them in to the function
  - If you do this, and also choose to do reflection for the additional component, make sure to declare a local, temporary red, green, and blue to pass in, so you don't clobber your existing results