MATH 580A Assignment 4 — Ray Tracing

1 Introduction

Ray tracing is a technique used for rendering scenes with 3d geometry. It is well-suited for rendering objects realistically and is based on modeling how individual light rays bounce and scatter before reaching our eyes. The downside however is that ray tracing is fairly time-consuming and often too slow for interactive 3d applications. In this assignment, you will implement a simple raytracer that can render planes and spheres with shading, shadow, reflections, and transparency. By the end of the assignment, you will be able to render a realistic-looking scene as in Fig. 1.

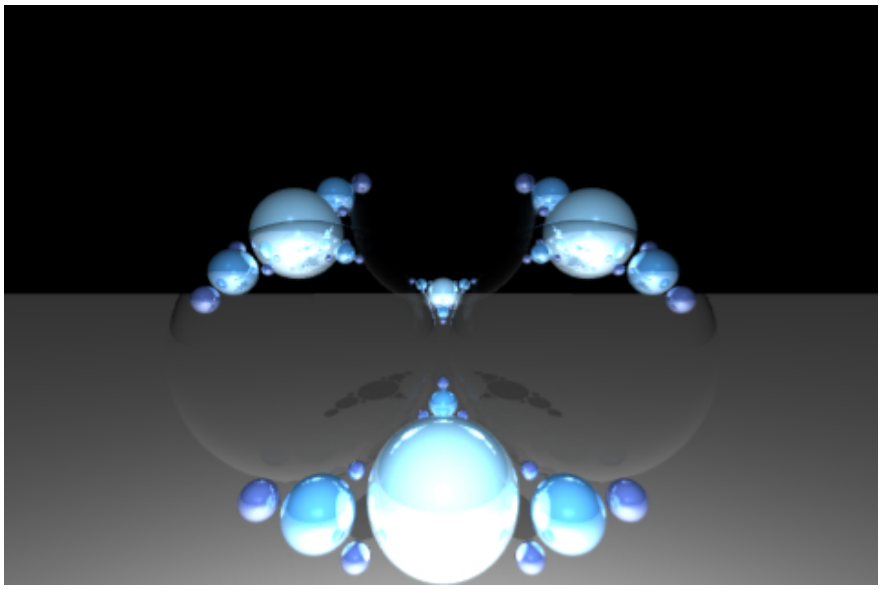

Figure 1: Raytracer Example

We will first summarize the basic principle behind raytracing in this section. Each subsequent section will introduce the necessary mathematics and physics for implementing an additional feature of the raytracer. Finally, there are a couple sections at the end of this handout on implementation details. You may wish to read the implementation details sections concurrently with the other more theoretical sections.

There are three main types of objects we will be concerned with: the ***camera***, ***lights***, and ***geometries***. The camera models how the scene is viewed. It is located at a specific point $\vec{\mathbf{p}}$ in $\mathbb{R}^3$, has a ***facing vector*** $\vec{\mathbf{f}}$ and an ***up vector*** $\vec{\mathbf{u}}$. The window screen sits in front of the camera in the direction of the facing vector and is oriented such that the up vector is aligned with the upwards direction on the window screen. The window screen is subdivided into many pixels and a ray is cast from the camera position to the center of each pixel in order to determine the color of that pixel.

Lights are what emit light rays that bounce or scatter off of objects so that they can be seen through the camera. In this assignment, we will consider ***point lights***, ***directional lights***, and ***ambient lights***. A point light has all the light concentrated at one point in $\mathbb{R}^3$ and a base intensity I. Its light attenuates, meaning that the intensity diminishes as per the inverse square law. That is, if a point light is illuminating a point at a distance r away from the source, then the effective intensity will be $\frac{I}{(1+r)^2}$. A directional light models the sun and has an intensity I and direction $\mathbf{d}$ in which light rays are cast. The light rays do not attenuate and do not emanate from a single point; a directional light emits all possible rays pointing in direction $\mathbf{d}$. Finally, there is usually a single ambient light in the scene. An ambient light does not physically correspond to any light source but instead provides a global fixed amount of illumination with intensity I to all objects in the scene.

Finally, geometries are the physical objects that we will see in a scene. In this assignment, we will only consider ***planes*** and ***spheres***. Geometries can have different properties that affect how light rays interact when intersecting their surface. The properties we will handle in this assignment are "shininess" via specular reflections and transparent objects with a certain refractive index. Our geometries will either be a single solid color or fully transparent.

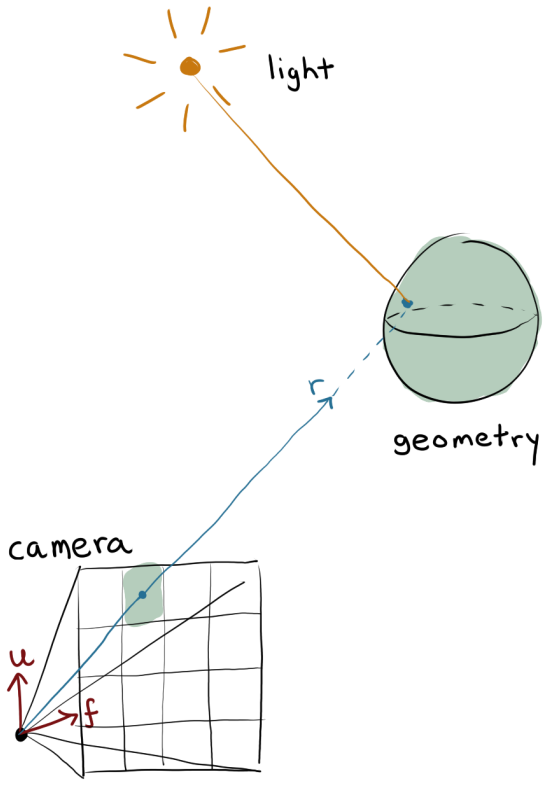

Figure 2: A Sample Scene with a Camera, Light and Geometry.

Fig. 2 depicts these three types of object in a scene. The dotted ray $\mathbf{r}$ is traced outwards from the camera position and goes through the center of a pixel on the screen in front of the camera. The ray intersects a sphere geometry at a certain point. We then look for lights in the scene that illuminate the point of intersection. In this case, we have a single point light that illuminates the point of intersection. By computing the intensity of light reflected off the sphere, we are able to

fill in the corresponding pixel with the appropriate color.

2 Emitting Rays

The very first order of business for implementing a ray tracer is to emit one ray per pixel on the screen. A camera object is set up for you already, but will need to implement the `Camera.pixel()` method that takes the `(x,y)` coordinate of a pixel and returns a ray with origin at the camera's position and going through the center of the appropriate pixel.

There are a couple parameters to be aware of. First, a camera has a ***field of view angle (fov)*** θ which determines the viewing angle of the frustrum. By default, we will use a field of view angle of $\frac{\pi}{4}$, but by adjusting this value towards $\frac{\pi}{2}$ or towards 0, we can simulate a wide-angle camera lens or a very narrow pinhole view. Next, the screen we are rendering will have a resolution `width x height` pixels and we need to subdivide the rendering plane appropriately. The size of each pixel on the rendering plane will be called the ***pixel size***.

From the facing vector $\vec{\mathbf{f}}$ and up vector $\vec{\mathbf{u}}$, we can compute the right vector $\vec{\mathbf{r}} = \vec{\mathbf{f}} \times \vec{\mathbf{u}}$. The rendering plane at distance $\|\vec{\mathbf{f}}\|$ will be a square of side length $s = \frac{2\tan(\theta)\|\vec{\mathbf{f}}\|}{\max\{w,h\}}$. For the pixel at position (x, y), we want to cast a ray starting from $\vec{\mathbf{p}}$ in the direction $\vec{\mathbf{f}} + \left(x - \frac{1}{2}\right) s\vec{\mathbf{r}} + \left(y - \frac{1}{2}\right) s\vec{\mathbf{u}}$. The camera setup and corresponding calculations are summarized in Fig. 3. Although the rendering plane is square, the actual screen will only involve a $w \times h$ subgrid of pixels.

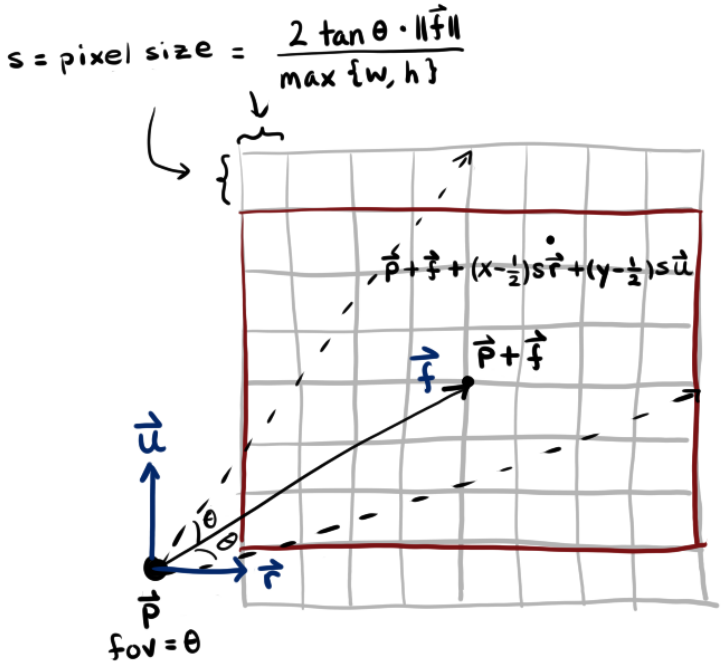

Figure 3: Camera Setup and Ray Calculations.

The first step of implementing the ray tracer, is to compute the correct ray for the `Camera.pixel()` function in `objects.py`, and call `color(r)` for each ray `r` corresponding to a pixel on the screen. At this point, rays are being cast but we have no logic used to detect whether the rays are hitting objects, so `color(r)` will just return a vector representing pure black `Vector3(0,0,0)`.

2.1 [Optional] Implement Supersampling Anti-Aliasing

By casting one ray per pixel, we will be determining the pixel's color based solely on the ray cast through the center of the pixel. This can lead to jagged or pixelated images, especially at low screen resolutions. The process of smoothing out jagged edges is known as ***anti-aliasing***. One simple method of anti-aliasing is to cast more than one ray per pixel and averaging the colors computed across the multiple rays. This is known as ***supersampling anti-aliasing (SSAA)*** and as an optional extension, you can implement a version of SSAA that samples four rays per pixels.

A naive approach may be to split each pixel into 4 subpixels in a 2×2 arrangement and use the centers of the subpixels for sampling. These positions of these centers would be then be $(x \pm \frac{1}{4}, y \pm \frac{1}{4})$. However, it turns out that a 2×2 grid shape often leads to suboptimal antialiasing because of jagged edges that align with the grid horizontally or vertically. Rotating the positions of these centers to be $(x + \frac{1}{8}, y + \frac{3}{8})$, $(x + \frac{3}{8}, y - \frac{1}{8})$, $(x - \frac{1}{8}, y - \frac{3}{8})$, $(x - \frac{3}{8}, y + \frac{1}{8})$ will significantly improve the antialiasing quality for the same number of samples. See Fig. 4 for a visualization of the sampling pattern.

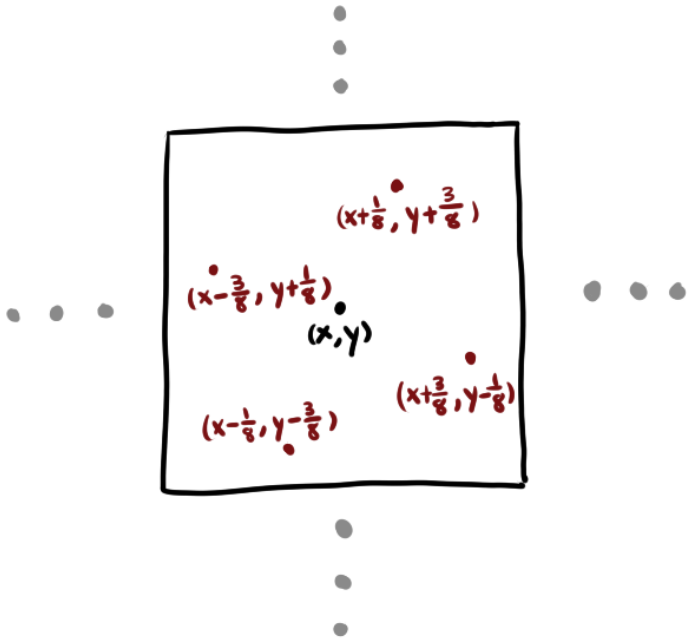

Figure 4: SSAA Rotated Grid Pattern

3 Plane and Sphere Geometries

In this section, you will implement the geometric functions necessary for detecting ray intersections with planes and spheres. You will implement the function `intersect_ray()` which takes a ray, represented as a line segment $(\vec{\mathbf{p}}_1, \vec{\mathbf{p}}_2)$, and return the nearest intersection of the ray with the geometry.

3.1 Planes

We will represent planes with a `center` point $\vec{\mathbf{c}}$ on the plane and a `normal` vector $\vec{\mathbf{n}}$. To compute the intersection with a ray, we wish to find some $0 \leq s \leq 1$ such that $(\vec{\mathbf{p}}_1 + s(\vec{\mathbf{p}}_2 - \vec{\mathbf{p}}_1) - \vec{\mathbf{c}}) \cdot \vec{\mathbf{n}} = 0$. Solving for s, we find that the point of intersection occurs when

$$s = \frac{(\vec{\mathbf{c}} - \vec{\mathbf{p}}_1) \cdot \vec{\mathbf{n}}}{(\vec{\mathbf{p}}_2 - \vec{\mathbf{p}}_1) \cdot \vec{\mathbf{n}}}. \tag{1}$$

You will also need to handle the case where no intersection occurs or when $s < 0$ or $s > 1$. In these cases, you should return `None`.

3.2 Spheres

We will represent spheres with a `center` point $\vec{\mathbf{c}}$ and a radius $r > 0$. Spheres are a little trickier than planes. To compute the intersection with a ray, we wish to find some $0 \leq s \leq 1$ such that $\|\vec{\mathbf{p}}_1 + s(\vec{\mathbf{p}}_2 - \vec{\mathbf{p}}_1) - \vec{\mathbf{c}}\| = r$. This is equivalent to solving the following quadratic equation,

$$\|\vec{\mathbf{p}}_2 - \vec{\mathbf{p}}_1\|^2 s^2 + 2\left((\vec{\mathbf{p}}_2 - \vec{\mathbf{p}}_1) \cdot (\vec{\mathbf{p}}_1 - \vec{\mathbf{c}})\right) s + \|\vec{\mathbf{p}}_1 - \vec{\mathbf{c}}\|^2 = r^2. \tag{2}$$

If the quadratic has multiple solutions, you will want to compute the nearest intersection, which corresponds to the minimum value of s such that $0 \leq s \leq 1$. Again, you should handle the case where no intersection occurs and return `None` appropriately.

3.3 Raycasting and Testing

At this point, you will be able to test your raytracer implementation thus far and start to see some images! First, implement `raycast()` in `utilities.py`. This function computes the first intersection with a geometry along the given ray and returns the intersection point and the geometry. Then, in `color()`, you can cast a ray and return the color of the geometry the ray hits. This will result in an image that accurately draws the geometry in the scene, but without any shading. When rendering `SCENE1`, you should see an image like in Fig. 5.

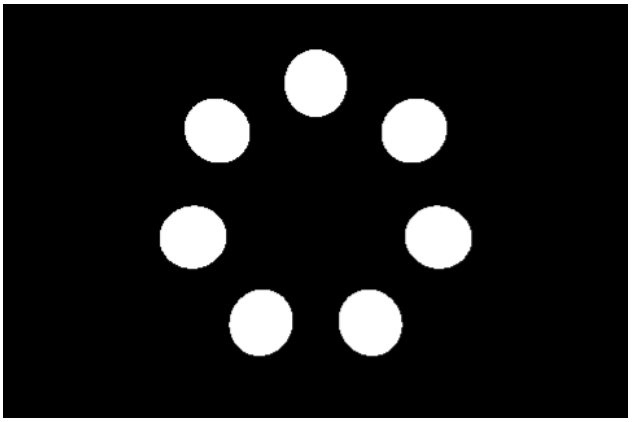

Figure 5: Initial Render of `SCENE1`.

4 Diffuse and Specular Shading

The next step is to shade our geometries based on the intensity of the light hitting their surfaces and reflecting into the camera. In real life, different materials reflect or scatter light in very different ways, depending on how matte or glossy the material is. We will model objects that can range from being completely matte to being very shiny. The particular shading model we will use is called ***Blinn-Phong shading***, named after Jim Blinn and Bui Tuong Phong. This shading model applies point lights and directional lights. Ambient lights will illuminate all objects uniformly, regardless of their position and angle relative to the camera.

When computing the color seen through the camera at a particular intersection point on a geometry, we will sum over the color contributes of each point light and directional light in the scene. The relevant variables for this computation are:

- The color C of the geometry.
- The intensity I of the light.
- The shininess of the geometry α.
- The normal vector $\vec{\mathbf{N}}$ at the point of intersection.
- The direction vector $\vec{\mathbf{V}}$ from the point of intersection back to origin of the ray cast.
- The direction vector $\vec{\mathbf{L}}$ from the point of intersection towards the light source.

All vectors will be normalized to be unit vectors.

The Blinn-Phong shading model first computes a unit ***halfway vector*** $\vec{\mathbf{H}} = \frac{\vec{\mathbf{L}}+\vec{\mathbf{V}}}{\|\vec{\mathbf{L}}+\vec{\mathbf{V}}\|}$. The color contribution from a given light is then equal to

$$\left((\vec{\mathbf{L}} \cdot \vec{\mathbf{N}})I + (\vec{\mathbf{H}} \cdot \vec{\mathbf{N}})^{\alpha} I\right) C. \tag{3}$$

See Fig. 6 for a visualization of the relevant vectors for computing Blinn-Phong shading at a point on a sphere.

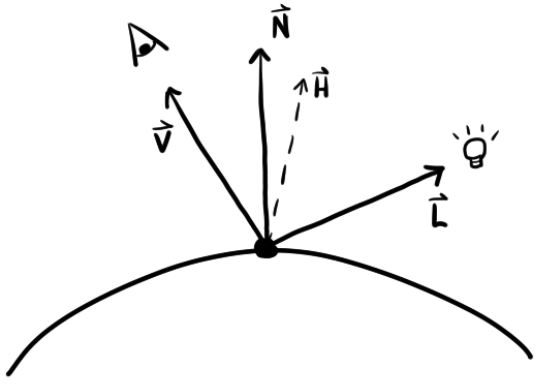

Figure 6: Vectors Used in Blinn-Phong Shading.

In Eq. (3), the term $(\vec{\mathbf{L}} \cdot \vec{\mathbf{N}})$ is the ***diffuse*** contribution. This term is what contributes to materials looking matte or dull. It is not dependent on angle the camera makes with the intersection point and light is assumed to scatter or diffuse in all directions when it hits the geometry. It does however depend on the angle the direction of light makes with the normal. Since $\vec{\mathbf{L}}$ and $\vec{\mathbf{N}}$ are both unit vectors, their dot product simply represents the cosine of the angle θ between them. The property that matte surfaces diffuses light based on $\cos(\theta)$ is known as ***Lambert's cosine law***. Intuitively, we expect that an object is most well-lit if the light shines head on when $\theta = 0$ and the object is not lit at all, if the light is tangent to the sphere when $\theta = \frac{\pi}{2}$.

The term $(\vec{\mathbf{H}} \cdot \vec{\mathbf{N}})^{\alpha}$ is the ***specular*** contribution. This is an approximation proposed by Blinn as a modification to Phong's original specular reflection model. Specular highlights make a geometry look glossy or shiny and correspond to light rays that reflect directly off the surface and into the camera. An ideal reflection occurs when the angle between $\vec{\mathbf{N}}$ and $\vec{\mathbf{L}}$ is equal to the angle between $\vec{\mathbf{N}}$ and $\vec{\mathbf{V}}$. To approximate this, we take the dot product of the halfway vector with the normal vector. Observe that this dot product tends to 1 in the ideal reflection case and tapers off to 0 otherwise. The exponent α controls how fast the tapering off occurs. When $\alpha = \infty$, we obtain a perfectly reflective surface. We will use values of $\alpha \approx 1000$ to model shiny surfaces.

4.1 Implementation Details

The original Blinn-Phong shading defined constants k_d and k_s used to weight the amount of contribution from diffuse and specular reflections respectively. Instead of specifying these constants manually for our geometries, we will always assume $k_d = 1$ and set $k_s = \frac{1-e^{-\alpha/200}}{1+e^{-\alpha/200}}$. This will ensure that objects with low shininess also contribute less light reflected specularly.

Another detail that you need to be aware of when implementing Blinn-Phong shading is that it's possible for some of our dot products to be negative. For example, a point light may graze the intersection point on the sphere at an angle greater than $\frac{\pi}{2}$ from the normal. In these cases, there should be no contribution to the diffuse or specular reflection. Thus, if a dot product results in a negative value, we should replace it with 0.

Finally, be sure to implement attenuation for point lights. The effective intensity for a point light is $\frac{I}{(1+r)^2}$ where r is the distance from the point light to the point on the geometry in question.

You should implement the Blinn-Phong shading computation in `blinn_phong()` in `utilities.py`, then call this function in `compute_intensity()` for both `PointLight` and `DirectionalLight`. You will also want to implement `AmbientLight.compute_intensity()` which simply returns IC without any consideration of vectors. Once these are all implemented, you should loop through each light and sum their intensity contributions in `color()` to obtain the color for each pixel in the raytracer. After implementing Blinn-Phong shading, you can try rendering `SCENE1` and `SCENE2`. At this point, they should look like as in Fig. 7.

4.2 Shadows

One simple graphical feature missing from our raytracer thus far is shadows. In order for shadows to be properly rendered, when computing the contribution from a given light source, we need to check if there are any other geometries in between our point of intersection and the light source. If there is an obscuring geometry, then we should return `Vector3(0,0,0)` as there is no light contribution. In addition, transparent objects do not obscure light, so you will only compute intersections with geometries where `is_transparent` is `False`.

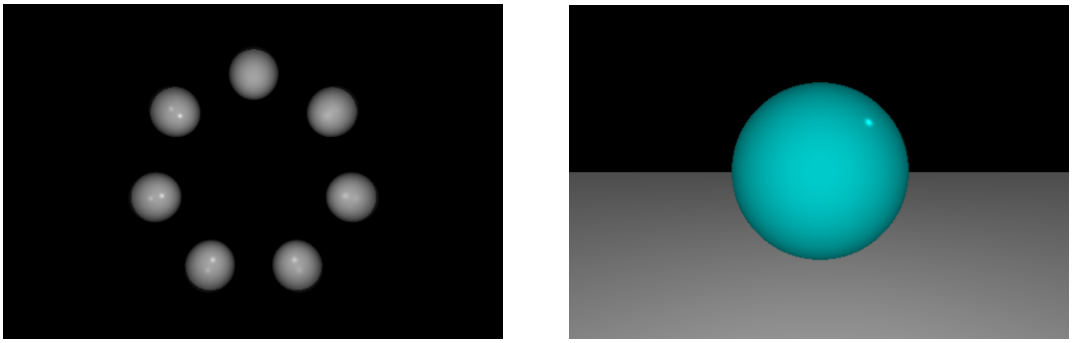

Figure 7: `SCENE1` and `SCENE2` with Blinn-Phong Shading.

When checking for obscuring geometry, you should cast a ray in the direction of $\vec{\mathbf{L}}$ from the point of intersection $\vec{\mathbf{p}}$. In order to avoid detecting the original point of intersection as an obscuring geometry, you can shift the origin of this ray by $0.1\vec{\mathbf{L}}$ so that it does not begin exactly at the point of intersection.

Once you have implemented shadows, you can render `SCENE2` again and it should look like as in Fig. 8. Observe that a sphere can obscure itself as in this scene, which produces the shadows along half the sphere.

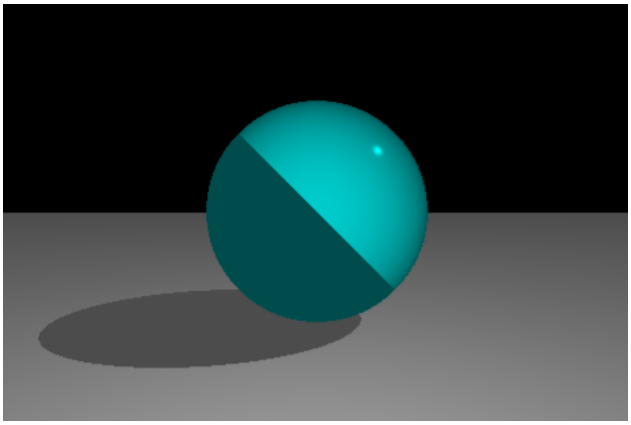

Figure 8: `SCENE2` Rendered With Shadows.

5 [Optional] Reflections

Besides having specular lighting, glossy geometries should also reflect other geometries on their surface. In this section, we will implement these reflections by modifying our raytracing to be *recursive*.

First, when we have a ray $\vec{\mathbf{r}}$ casted and intersecting with our geometry at a point $\vec{\mathbf{p}}$, we need to compute the reflected ray $\vec{\mathbf{r}}'$. Let $\vec{\mathbf{n}}$ be the normal vector of our geometry at point $\vec{\mathbf{p}}$. Our reflected ray should lie in the same plane as $\vec{\mathbf{r}}$ and $\vec{\mathbf{n}}$ and make the same angle with the normal as the original ray does. Assuming our normal vector is a unit vector, the reflected ray can be computed via:

$$\vec{\mathbf{r}}' = \vec{\mathbf{r}} + 2(\text{proj}_{\vec{\mathbf{n}}}(\vec{\mathbf{r}}) - \vec{\mathbf{r}}). \tag{4}$$

Here, $\text{proj}_{\vec{\mathbf{n}}}(\vec{\mathbf{r}}) = (\vec{\mathbf{r}} \cdot \vec{\mathbf{n}})\vec{\mathbf{n}}$ is the projection of $\vec{\mathbf{r}}$ onto $\vec{\mathbf{n}}$. The reflected ray is depicted in Fig. 9.

Once we have computed the direction of the reflected ray, we want to add a reflective contribution based on the color seen by tracing along the reflected ray $\vec{\mathbf{r}}'$. Thus, we will recursively call `color()` with our new ray to compute what color we would see from our point of intersection at the geometry if we looked towards $\vec{\mathbf{r}}'$. Note that if our reflected ray intersects another geometry, we may have to recursively compute reflections on that geometry first! When our scene contains a lot of geometries, this could lead to a lot of recursive calls and we will implement a ***maximum recursion depth*** for our raytracing. By keeping track of how many times we have recursed so far, we can stop and return `Vector3(0,0,0)` if we have reached the maximum recursion depth.

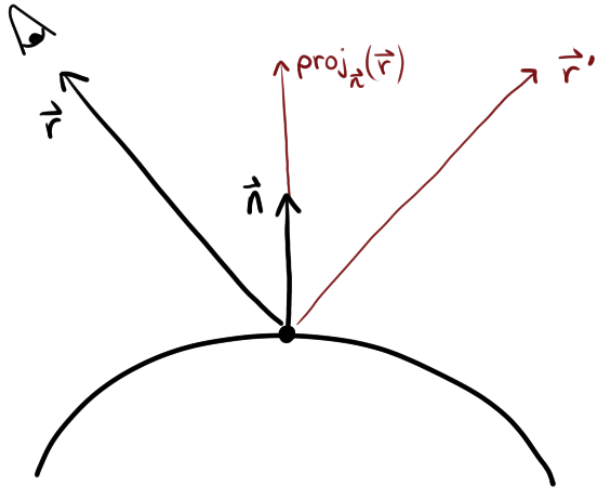

Figure 9: Reflected Ray $\vec{\mathbf{r}}'$.

One other detail to keep in mind when implementing reflections is that like specular lighting, we want to weight how much we reflect based on the shininess α of our geometry. We will use the same weight $\frac{1-e^{-\alpha/200}}{1+e^{-\alpha/200}}$ as for specular lighting. Once you have implemented reflections, you can try rendering `SCENE3`. You should see something similar to Fig. 10

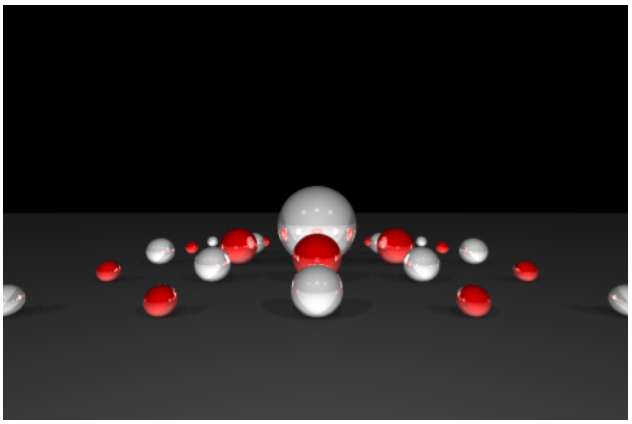

Figure 10: Rendering `SCENE3`.

6 [Optional] Transparent Objects and Refraction

The final feature we will add to our raytracer is the ability to handle transparent geometries. In the real world, there is a wide spectrum of transparent materials, such as water, tinted acrylic, translucent materials (e.g. frosted glass), etc. Our transparent geometries will approximate fully transparent colorless glass.

To compute the color of a ray that intersects a transparent geometry, we will need to consider both reflection and ***refraction***. Reflection we have already seen how to implement in the previous section. We will focus on computing the refraction ray and then see ***Schlick's appproximation*** for how to weight the reflection and refraction rays.

6.1 Refraction

When light enters a different medium, it bends in a phenomenon known as refraction. For this assignment, you will need to simulate light refracting as it passes from air into a glass sphere and back out into the air. Note that we will not consider planes in this assignment.

Let $\vec{\mathbf{v}}$ be the ray from the point of intersection at a sphere back towards the camera and let $\vec{\mathbf{N}}_1$ be the unit normal at this point of intersection. Let θ_1 be the angle between the vectors $\vec{\mathbf{v}}$ and $\vec{\mathbf{N}}_1$. ***Snell's law*** describes how light gets refracted as it passes through the medium. If we let $\vec{\mathbf{v}}'$ be the refracted ray and θ_2 be the angle between $\vec{\mathbf{v}}'$ and $-\vec{\mathbf{N}}_1$, then Snell's law tells us that

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n}{1}. \tag{5}$$

Here, n is called the ***refractive index*** of the medium. The right hand side is written with a denominator of 1 because the refractive index of air is approximately 1. For the refractive index of glass, we will default to 1.5 which is a good approximation empirically, though you can play around with different refractive indices for different effects.

The ray $\vec{\mathbf{v}}'$ will travel through the glass sphere and intersect the sphere at a different intersection point. When this ray exits from the second intersection point, we apply Eq. (5) again, though the right hand side now has the refractive indices reversed, as light is exiting glass back into the air. This will result in a new ray $\vec{\mathbf{v}}''$. Finally, you should call `color()` starting from this new intersection point on the sphere and in the direction of $\vec{\mathbf{v}}''$. The relevant geometry is summarized in Fig. 11. Note that necessarily $\phi_1 = \theta_2$ and $\phi_2 = \theta_1$, though you don't need this fact for the implementation.

Observe that when you solve $\phi_2 = \sin^{-1}(n\sin\phi_1)$, it is possible for numerical inaccuracy issues to lead to $|n\sin\phi_1| > 1$. To handle this edge case, you can either use $\phi_2 = \theta_1$ or render the contribution from refraction as `Vector3(0,0,0)`.

6.2 Schlick's Approximation

When a light ray hits a glass object, what happens is that part of the light is reflected off of the glass and part of the light passes through and is refracted. The ratio of reflected light to refracted light is determined by the ***Fresnel equations***. These equations are relatively complicated and a common approximation used in raytracing is known as Schlick's approximation, named after Christophe Schlick.

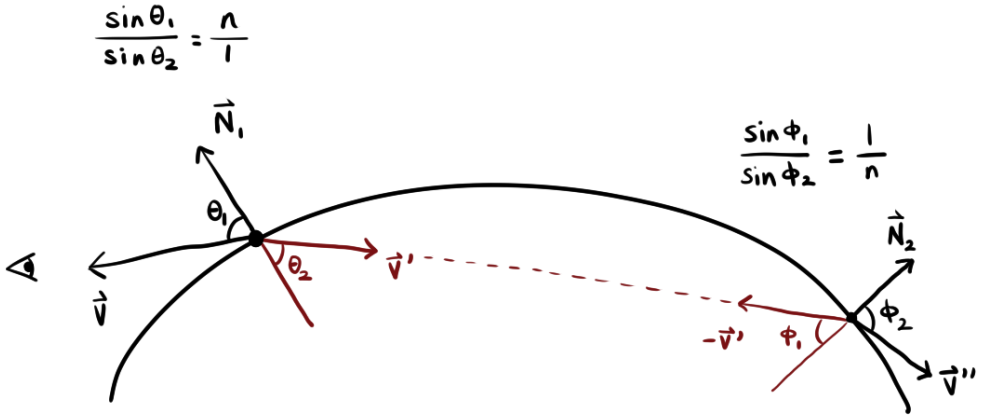

Figure 11: Refraction Through a Glass Sphere.

Let θ be the angle the view vector $\vec{\mathbf{v}}$ makes with the normal vector $\vec{\mathbf{N}}$. Then

$$R = \left(\frac{1-n}{1+n}\right)^2 + \left(1 - \left(\frac{1-n}{1+n}\right)^2\right)(1-\cos\theta)^5, \tag{6}$$

is Schlick's approximation for the reflection coefficient. Here n is the refractive index of the medium. Observe that $0 \leq R \leq 1$. If C_{reflect} is the color contribution from reflection and $C_{\text{refraction}}$ is the color contribute from refraction, then the total color contribution is

$$C = R \cdot C_{\text{reflect}} + (1-R) \cdot C_{\text{refraction}}. \tag{7}$$

6.3 Implementation

In this assignment, we render transparent geometries and non-transparent geometries in completely different ways. In particular, the Blinn-Phong shading computations do not apply to transparent objects. In `color()` you should check if the point of intersection lies on a transparent geometry or an opaque geometry and handle the two cases accordingly.

Once you have implemented transparent geometries, you can try rendering `SCENE4` and `SCENE5`. `SCENE5` was depicted in the introduction and your `SCENE4` should look similar to Fig. 12.

7 Object-Oriented Programming

A feature of Python that we have skipped over thus far are ***objects*** and ***classes***. These are features common in a type of programming known as ***object-oriented programming (OOP)***. In this section we will briefly cover the relevant OOP details.

A class refers to a type of data. For example, numbers, strings, lists, dictionaries are all different classes in Python. Classes can store data in different ways and operators can have different meanings for different classes. For example, the + operator refers to numerical addition on numbers but list concatenation for lists.

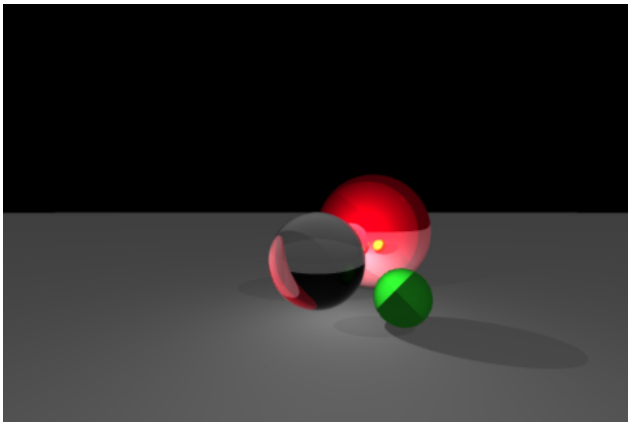

Figure 12: Rendering `SCENE4`.

Classes can also have custom ***methods***. For example, lists have a `<list>.sort()` method which sorts a list in place. Methods differ from functions in that they are tied to a specific class. So although you might be able to pass in different classes into a `print()` function, you can only call `.sort()` on a list.

Large programs tend to have their own custom classes to represent data they want to work with. In this assignment, the following classes are set up for you:

- `Vector3`: This represents a vector in $\mathbb{R}^3$. Addition, subtraction, and multiplication with a scalar behaves as expected.
- `Camera`: This represents the camera object discussed in the introduction.
- `Plane`, `Sphere`: These represent plane and sphere geometries respectively.
- `PointLight`, `DirectionalLight`, `AmbientLight`: These represent the different types of light objects.

An object is a particular instance of a class. So the list type abstractly is a class, but one specific list instance is an object. In order to create an instance of a custom class, you call the class name as if it were a function. For example `Vector3(1.0, -1.5, 0.0)` would create a 3-dimensional vector object with coordinates $(1, -1.5, 0)$.

In this assignment, you will not need to define new methods or new classes, but you will need to instantiate objects and fill out the logic for methods on some of the classes. As you look through the classes in `objects.py`, you will notice that each method begins with an argument called `self`. This is a special argument that corresponds to the specific object instance the method is called on. For example, when implementing `def normal_at(self, point)` in the `Sphere` class, you can use `self.center` and `self.radius` to access that specific sphere object's center and radius respectively.

There is also a special method called `__init()__` that is used for initializing objects. By looking in that method, you can see what attributes are accessible on a given class. For example, if you look at the `__init()__` method on the `Sphere` class, you'll see that `Sphere`'s have a `center` and `radius` attribute. Both `Sphere` and `Plane` also have some additional common attributes that you can find on the `Geometry` class. This is a bit of special syntax in Python that allows multiple

classes to share some code. So both `Sphere` and `Plane` have a `shininess` or `is_transparent` attribute for example.

8 Implementation Details

There is a fair amount of starter code provided for you. All the functions and methods you will need to implement are written out, with `#TODO` comments where appropriate. Furthermore, a relatively comprehensive `Vector3` class implementations provided. For example, you can call `v.dot(w)` to obtain the dot product of `v` with `w`. The full list of methods provided is in the implementation of `Vector3` in `utilities.py`.

In the past, we have seen colors represented as 3-tuples of integers between 0 and 255 inclusive. In this assignment, we will work with colors as `Vector3`'s with components in the range `0.0` to `1.0`. After computing the final color for each pixel, you should call `color_clamp()`. This will ensure that color values beyond `1.0` are clamped down to `1.0`.

Graphical applications often have a coordinate system where the x-axis points to the right and the y-axis points downwards on the screen. You do not have to worry about this convention and may instead treat the screen as going from `-WIDTH // 2` to `WIDTH // 2` in the x direction and `-HEIGHT // 2` to `HEIGHT // 2` in the y direction. There is code provided for you to handle the difference in convention.

Finally, raytracing is a slow process! Complicated scenes with lots of objects, SSAA, and reflection can take nearly a minute to render. When testing your code, you can render a low quality version by setting the `WIDTH` and `HEIGHT` parameters to a smaller resolution and turning SSAA off. When you've verified that your implementation looks correct, you can adjust the settings to render a high quality image.