RAY TRACING

Joe Redmon

Adviser: Matthew Dickerson

A Thesis

Presented to the Faculty of the Computer Science Department

of Middlebury College

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Arts

May 2011

ABSTRACT

Ray tracing is a powerful 3-dimensional rendering algorithm that produce highly realistic images of geometric data about a scene. This thesis examines the underlying geometery and physics of ray tracing as well as the algorithms associated with these concepts. It ends with examples of ray tracing generated using the descibed algorithms.

ACKNOWLEDGEMENTS

I'd like to thank Aseem Mulji for letting me use his account to print the color pages of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

AN INTRODUCTION TO RAY TRACING

A central problem in computer graphics is how to best represent 3-dimensional data in two dimensions so that it can be drawn on a screen. The process of taking 3-dimensional data and producing a 2-dimensional image is known as 3-dimensional rendering. From visualizing complex data sets to playing computer games, 3-dimensional rendering lies at the heart of how humans interact with computers.

Ray tracing is an intuitive, extensible 3-dimensional rendering technique capable of producing photorealistic images from geometric data about a scene. To understand how ray tracing works we need to explore how humans process visual stimulus in the real world.

1.1 The Visual Process

When you look at an object, there is a long chain of events that occurs prior to you being able to actually see the object. Light is emitted from a light source in some proximity to the object. Some of this light travels in the direction of the object, some of it shoots off in a completely different direction. Of the light that travels toward the object, some of it actually collides with the object. If the object is not a black hole (and chances are it isn't!) some of that light will bounce off the object, and some of that light might even be absorbed by your eye.

Light travels into your eye through your pupil which acts as a focal point. From here the light is projected onto your retina, where specialized nerve endings send information about the light that strikes them to your brain. Your brain interprets these signals, and translates the information into a cohesive picture.

A focal point is necessary so that we get a clear, focused image of the scene. If we restrict light so that it must pass through a focal point, we ensure all the light hitting

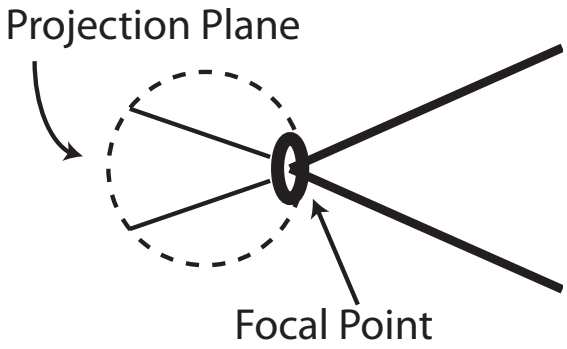

Figure 1.1: Light from a scene travels through the pupil and is projected onto the retina at the back of the eye.

a single nerve ending in your eye had to travel along the same path to get there, and thus originated from the same location. Otherwise light could come from anywhere and stimulate the same nerve ending, creating a blurry, confusing picture.

Cameras mimic this process. Their lens has a focal point that eliminates extraneous light, while the light that does pass through the focal point gets projected onto an image plane in the back of the camera, similar to the retina in the back of our eyes. This plane contains photosensitive film or electronics that record the type of light hitting it. Using this data we can reconstruct the image.

1.2 Modeling Vision

We can imagine modeling this exact process on a computer. We would pick an arbitrary point as our focal point. We also need a projection plane, which we can actually put in front of the focal point instead of behind it. The focal point need not be in front of the image plane, we only need to ensure that the light passes through it eventually for the image to be in focus. We can think of this plane as screen that we are looking through and into the scene. We can subdivide the plane into the pixels of a computer screen. Then we generate light rays from a source and send them out to bounce around in the scene. Any time a light wave travels through the screen and to the focal point, we would

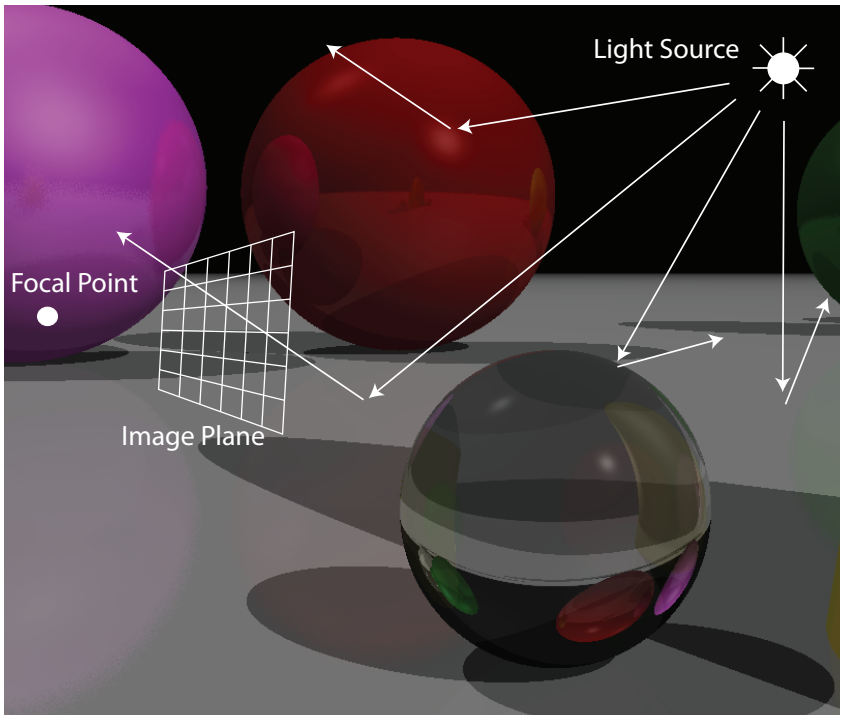

Figure 1.2: Modeling a real scene on a computer. Few, if any of the light rays actually pass through the image plane and hit the focal point.

color the pixel it passes through accordingly.

This would accurately model what happens in real life, but if we actually implemented it we would be sorely disappointed. The odds of a random ray actually traveling through the image plane and hitting the focal point are astronomical. In real life there are billions of photons crashing around at any given time, so some of them are bound to end up flying into your eye. However, we simply do not have the computing power to model sufficient photons to ensure any of them actually make it to the image plane, and there is no way to know a priori if a given ray will end up in the image or not.

1.3 Ray Tracing

Ray tracing takes this model and flips it around. Instead of tracing light from the light source into the image plane (or wherever else it happens to go), we trace rays from the

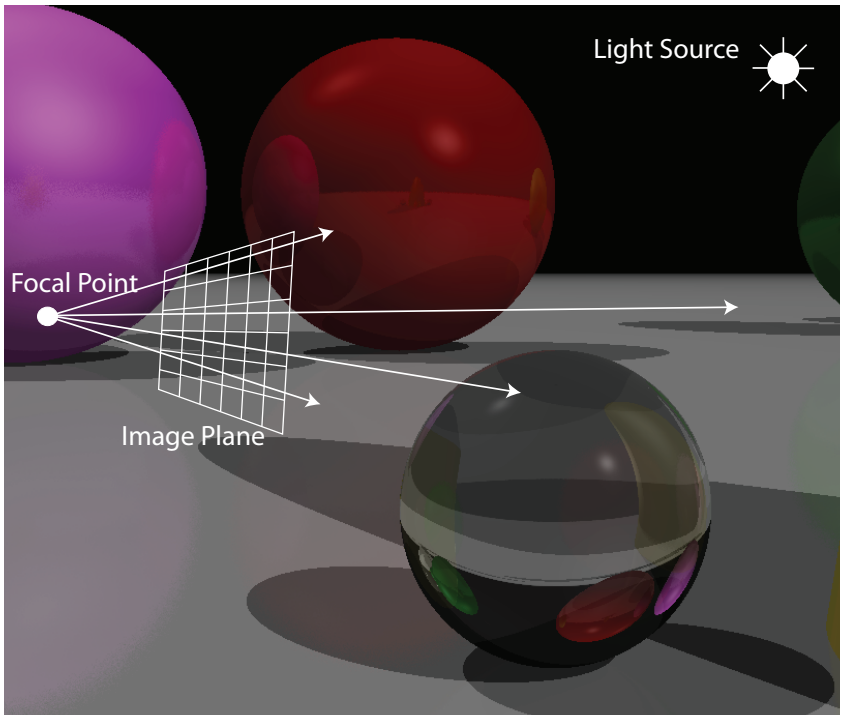

Figure 1.3: Ray tracing projects rays from the focal point through the image plane and out into the scene. It uses these rays to gather information about the scene, and uses this information to color the appropriate pixels in the image plane.

image plane out into the scene. When they collide with an object, we examine the point of collision to determine what kind of light could be reflected from that point and back along the incident ray into the scene. This cuts the computational power exponentially, since we are no longer generating extraneous rays that contribute nothing to the final image.

The algorithm is as follows: For each pixel in the image plane, trace a ray from the focal point through that pixel into the image plane. If that ray intersects any object in the scene, find the closest point of intersection. We can only see the closest object to us, anything behind it is obscured by it. Gather data about the light that could be reflected off of the intersection point into the focal plane. Use this data to color the pixel accordingly.

Ray tracing has clear advantages over other 3-dimensional rendering techniques. Be-

cause it approximates closely how human vision actually works, it can produce highly realistic output. This also means that it natively handles complicated lighting effects, shading, and shadows while other techniques require numerous add-ons to simulate these features.

However, this realism comes at a price. Ray tracing can be computationally expensive and slow to render large images or complicated scenes. Due to its accuracy and realism, it is often used for rendering that can be done in advance, such as CGI for movies. Though is not widely used for real-time application due to the computational power it requires, this may change as computer performance continues to increase.

To implement this algorithm we have to understand some fundamental geometry and physics. First, we need to be able to find intersection points with the rays we generate and the objects in our scene. Then we have to be able to examine those points and determine the properties of the light that reflects off of them and back into the image plane. The following chapters provide an overview of the algorithms involved in ray tracing as well as tips on how to implement them.

CHAPTER 2

THE GEOMETRY OF RAY TRACING

At the core of any ray-tracing program lies the problem of finding ray-object intersections in the scene. The algorithms we use for ray-object intersections must extract two key pieces of information: the closest intersection point between the ray and an object in the scene, and the surface normal of the object at that point. The surface normal of the object will play an important role later when we discuss the physics behind light reflection and transmission. For some types of rays, we don't need all this information. For example, shadow feelers can halt immediately after the first collision with another object. They do not need to find the closest intersection because if there is anything in between the object and the light source, the point is obscured from the light, regardless of how many objects are in the way. However, in general we will need to find the nearest intersection and surface normal efficiently for a wide variety of object types.

The basic strategy for intersection algorithms is to use the general equation for an object as well as the parametric description of the ray to solve a system of equations yielding the point of intersection (or points of intersection depending on the surface). A ray is described in vector form as a line with one extra constraint.

$$\mathbf{r} = \mathbf{p} + \lambda \mathbf{v}, \lambda \geq 0 \tag{2.1}$$

Where $\mathbf{p}$ is the starting point for the ray and $\mathbf{v}$ is the direction vector. By constraining λ to be non-negative, we enforce the directionality of the ray. By pulling apart the vector equation, we get the parametric equation for a ray:

$$x_{\mathbf{r}} = x_p + \lambda x_v$$

$$y_{\mathbf{r}} = y_p + \lambda y_v$$

$$z_{\mathbf{r}} = z_p + \lambda z_v$$

Combining this parametric equation with the equations for different objects we can find a general formula for finding intersections with that object.

2.1 Spheres

Spheres are a natural starting point for ray tracing programs because they have simple equations, and they exemplify precisely what ray-tracing algorithms excel at, the rendering of curved surfaces. First we must determine if a given ray $\mathbf{r}$ intersects our sphere.

2.1.1 Deriving Ray-Sphere Intersection

We can think of a sphere as a collection of points that are all equidistant from a center point $\mathbf{c}$ by some radius r. A point $\mathbf{q}$ is in the circle if and only if it satisfies the following equation:

$$||\mathbf{q} - \mathbf{c}|| = r \tag{2.2}$$

Thus a point is in the circle if the magnitude of the vector from the center to that point is r. To find the intersections of our ray and the circle, we substitute in place of $\mathbf{q}$ our equation for a ray.

$$||\mathbf{p} + \lambda\mathbf{v} - \mathbf{c}|| = r \tag{2.3}$$

Now we can solve for λ and plug the result into our ray formula to find the intersection points. To do this we recall our equation for the magnitude of a vector:

$$\sqrt{(x_p + \lambda x_v - x_c)^2 + (y_p + \lambda y_v - y_c)^2 + (z_p + \lambda z_v - z_c)^2} = r \tag{2.4}$$

Rearranging and factoring, we get the following quadratic equation:

$$\begin{aligned}(x_v^2 + y_v^2 + z_v^2)\lambda^2 + 2(x_p x_v + y_p y_v + z_p z_v - x_c x_v - y_c y_v - z_c z_v)\lambda \\ + (x_p^2 + y_p^2 + z_p^2 + x_c^2 + y_c^2 + z_c^2 - 2x_p x_c - 2y_p y_c - 2z_p z_c - r^2) = 0\end{aligned} \tag{2.5}$$

Which simplifies to:

$$(x_v^2 + y_v^2 + z_v^2)\lambda^2 + 2((x_p - x_c)x_v + (y_p - y_c)y_v + (z_p - z_c)z_v)\lambda \\ + ((x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2 - r^2) = 0 \quad (2.6)$$

This looks like a mess, but since it is a quadratic equation, we can apply the quadratic formula to find the roots of this equation. Recall that a quadratic equation:

$$a\lambda^2 + b\lambda + c = 0 \quad (2.7)$$

has roots

$$\lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (2.8)$$

In this case we have the following values for a, b, and c:

$$a = x_v^2 + y_v^2 + z_v^2$$
$$b = 2((x_p - x_c)x_v + (y_p - y_c)y_v + (z_p - z_c)z_v)$$
$$c = (x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2 - r^2$$

Since our coefficient b is even we can actually use a modified version of the quadratic formula, that factors out the extra 2, meaning the quadratic equation:

$$a\lambda^2 + 2\beta\lambda + c$$

has roots

$$\lambda = \frac{-\beta \pm \sqrt{\beta^2 - ac}}{a}$$

Thus we have modified coefficients:

$$a = x_v^2 + y_v^2 + z_v^2$$
$$\beta = (x_p - x_c)x_v + (y_p - y_c)y_v + (z_p - z_c)z_v$$
$$c = (x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2 - r^2$$

There are three possibilities for finding roots, and thus intersections. Either:

$$\beta^2 - ac < 0 \tag{2.9}$$

and there are no real roots to the equation, therefore no points of intersection between the ray and sphere. Or:

$$\beta^2 - ac = 0 \tag{2.10}$$

and there is a single root to the equation, and thus the ray lies tangent to the sphere, with

$$\lambda = \frac{-\beta}{a} \tag{2.11}$$

Or finally:

$$\beta^2 - ac > 0 \tag{2.12}$$

meaning there are two real roots to the equation, and two potential points of intersection, with

$$\lambda = \frac{-\beta \pm \sqrt{\beta^2 - ac}}{a} \tag{2.13}$$

In the case that (2.10) or (2.12) holds, we still must perform additional checks to make sure these intersection points are valid. In both cases we must ensure that λ is non-negative, otherwise the point of intersection is on the wrong side of the ray. In case (2.12) we have to find the closest point of intersection to the start of the ray, which will be the smallest of possible values for λ such that λ is still non-negative.

2.1.2 Finding Sphere Surface Normals

Once we have found an intersection point on the sphere, determining the surface normal at that point is easy. At any point on a sphere, the surface normal is simply the vector from the center of the sphere to that point. Thus for intersection point $\mathbf{p}$ on the sphere, the surface normal, $\mathbf{N}$ at that point is given by:

$$\mathbf{N} = \mathbf{p} - \mathbf{q} \tag{2.14}$$

However this formula only yields one of two possible normals. At any point, a 3-dimensional surface has two surface normals, since if a vector $\mathbf{N}$ is perpendicular to the surface at that point, so in the vector $-\mathbf{N}$. Equation (2.14) gives the formula for the surface normal that points outward from the sphere, which is the surface normal we want if the incident vector originates from outside the sphere. Often times this is the case, however if a ray originates inside the sphere, we will want the internal surface normal, not the external normal. There is an easy test to figure out which normal is appropriate.

Given incident vector $\mathbf{I}$ and potential surface normal $\mathbf{N}$, we consider the angle between the two vectors. We always want the normal vector to point back in the direction of the incident vector, thus we want the angle between the vectors to be obtuse. We can use the dot product of the two vectors to test whether this is the case. Recall that the dot product of two 3-dimensional vectors is given by the following:

$$\mathbf{v} \cdot \mathbf{t} = x_v x_t + y_v y_t + z_v z_t \tag{2.15}$$

Also recall the following property about the angle between two vectors θ_{vt}

$$\cos(\theta_{vt}) = \frac{\mathbf{v} \cdot \mathbf{t}}{||\mathbf{v}|| \, ||\mathbf{t}||} \tag{2.16}$$

Thus if:

$$\mathbf{N} \cdot \mathbf{I} < 0 \tag{2.17}$$

The cosine of their angle is negative (since magnitudes of vectors must be positive), and the angle is obtuse. In this case we can return $\mathbf{N}$ as the correct surface normal. On the other hand if

$$\mathbf{N} \cdot \mathbf{I} > 0 \tag{2.18}$$

then the angle is acute, and the incident vector originated from inside the sphere. In this case we can return $-\mathbf{N}$ as the correct surface normal.

Note that if

$$\mathbf{N} \cdot \mathbf{I} = 0 \tag{2.19}$$

we are in a special case where the incident vector is perpendicular to the surface normal, thus the incident ray lies tangent to the surface. In the case of spheres this implies that the ray originated outside of the sphere, and we can return $\mathbf{N}$ as the proper surface normal.

2.2 Planes and Flat Surfaces

Next we will examine ray-plane intersections, and then extend the result to cover bounded polygons.

2.2.1 Deriving Ray-Plane Intersections

A plane can be described using only the normal vector $\mathbf{N}$ and a point on the plane $\mathbf{q}$. Another point $\mathbf{p}$ is in the plane if and only if:

$$\mathbf{N} \cdot (\mathbf{p} - \mathbf{q}) = 0 \tag{2.20}$$

If we recall (2.16), (2.20) should make intuitive sense, because if the dot product between $\mathbf{N}$ and a vector from $\mathbf{q}$ to $\mathbf{p}$ is 0, the cosine of the angle between the two vectors must be 0, hence they are perpendicular. Since the vector is perpendicular to the normal, it must lie parallel to the plane, and since one end point of the vector is in the plane, the other must be too.

Thus for some given plane defined by $\mathbf{N}$ and $\mathbf{q}$ we can plug in our parametric equation for a ray in place of $\mathbf{p}$ and solve for λ to find any intersection.

$$\mathbf{N} \cdot (\mathbf{p} + \lambda\mathbf{v} - \mathbf{q}) = 0 \tag{2.21}$$

Expanding the dot product we get:

$$x_n(x_p + \lambda x_v - x_q) + y_n(y_p + \lambda y_v - y_q) + z_n(z_p + \lambda z_v - z_q) = 0 \tag{2.22}$$

Distributing the components of the normal vector, we get a sum of two dot products:

$$\mathbf{N} \cdot (\mathbf{p} - \mathbf{q}) + \lambda(\mathbf{N} \cdot \mathbf{v}) = 0 \tag{2.23}$$

Rearranging, we get a final formula for λ

$$\lambda = \frac{-\mathbf{N} \cdot (\mathbf{p} - \mathbf{q})}{\mathbf{N} \cdot \mathbf{v}} = \frac{\mathbf{N} \cdot (\mathbf{q} - \mathbf{p})}{\mathbf{N} \cdot \mathbf{v}} \tag{2.24}$$

As with the ray-sphere intersection formula, there are a few options here. If we have:

$$\mathbf{N} \cdot \mathbf{v} = 0 \tag{2.25}$$

we know the normal vector of the plane and the direction vector of the ray are perpendicular, hence the ray lies parallel to the plane. In this case, it is possible that:

$$\mathbf{N} \cdot (\mathbf{q} - \mathbf{p}) \neq 0 \tag{2.26}$$

and the result of the quotient is undefined, so the ray lies parallel to the plane, but not in it, and there is no intersection. Alternatively, if

$$\mathbf{N} \cdot (\mathbf{q} - \mathbf{p}) = 0 \tag{2.27}$$

then the vector from $\mathbf{p}$ to $\mathbf{q}$ is also perpendicular to the normal of plane, thus the ray lies in the plane.

If the denominator is non-zero, and

$$\lambda = \frac{\mathbf{N} \cdot (\mathbf{q} - \mathbf{p})}{\mathbf{N} \cdot \mathbf{v}} \geq 0 \tag{2.28}$$

then we have our point of intersection!

2.2.2 Deriving Ray-Polygon Intersections

Objects in 3-dimensional scenes are commonly represented in terms of polygon meshes. Thus calculating ray-polygon intersections is an important extension to add to our ray-plane intersection formula. Ray-polygon intersections can be thought of as first finding a ray-plane intersection, and then determining if that point lies within a bounding polygon.

For our purposes we will assume that a polygon is described by a set of three or more points $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_i$ that are coplanar, and if traversed in order, bound the polygon. First we must find the plane on which these points lay and determine if there is a potential intersection point with a given ray. Any three unique, non-collinear points will describe a plane, whose normal is given by the following equation:

$$\mathbf{N} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1) \tag{2.29}$$

Recall that the cross product of two vectors will give us a vector that is perpendicular to both of them. Since all three points lie in the plane, all are unique, and they are not collinear, $(\mathbf{p}_2 - \mathbf{p}_1)$ and $(\mathbf{p}_3 - \mathbf{p}_1)$ will describe two unique, nonparallel vectors that are both parallel to the plane. Their cross product, $\mathbf{N}$ will be perpendicular to both of them, and thus perpendicular to any vector in the plane, so it is one of the normal vectors for the plane.

Now that we have a normal vector, we can apply our previous formula for ray-plane intersections to get a potential intersection point. Once we have a potential intersection point $\mathbf{q}$, we must determine whether this point lies within the bounding polygon. To do this, we use the crossing number algorithm.

The crossing number algorithm, or even-odd algorithm works off of the simple premise that if you draw a ray from the point in question in any direction, the number of intersections that ray has with the polygon determines whether it is contained in the polygon.

The last point of intersection between the ray and the polygon's bounding lines has to

be the polygon exiting for the last time. Also, every intersection represents the ray either entering or exiting the polygon. Thus if we count the number of times the ray intersects the bounding edges of the polygon, we know whether it originated on the inside or outside. If it never intersects the polygon, we know that it must have started outside the polygon, since if it started inside, it would have had to intersect it at some point to exit it. A single intersection implies that the ray originated inside the polygon, since it exited the polygon, but never entered it. In general, an even number of intersections means the ray started outside the polygon, since for every point that the ray exited the polygon, there exists a complementary point where the ray entered it. An odd number of intersections implies that the ray started inside the polygon, since the ray exits the polygon exactly one more time than it enters it.

The algorithm works regardless of the ray chosen, as long as it starts at the point in question. To simplify the algorithm, we can consider the test point as being at the origin, translating the polygon's bounds accordingly, and then use the positive x-axis as our test ray, examining x-intercepts of the polygon. Here is the algorithm in full:

If $points$ is an array of n bounding points for the polygon in cyclic order, and $test$ is the test point:

Algorithm 1 Determines if $test$ is in polygon defined by $points$ [10][4]

$in \leftarrow$ false
$points[n] \leftarrow points[0]$
for $i = 0$ to $n - 1$ **do**
  if $(test.y \leq points[i].y) = (test.y > points[i + 1].y) \wedge test.x - points[i].x - [(test.y - points[i].y) \times (points[i+1].x - points[i].x)]/(point[i+1].y - points[i].y)$ **then**
    $in \leftarrow$ not in
  end if
end for
return in

However, we still have a problem to address. This algorithm works for a polygon and a point defined in 2-dimensions, but we are dealing with a 3-dimensional polygon

and point. One option is to translate to the coordinate system of the plane defined by the polygon. This is computationally expensive, though, and there is an easier method. Instead we can simply drop one of the coordinates from the points defining the polygon, as well as from the point we are testing, and use these 2-dimensional points for the algorithm.

We cannot simply drop any random coordinate and expect the algorithm to be robust. If, for example, our polygon is parallel to the y-axis, dropping the y-axis would result in a gross loss of information. We can ensure this does not happen by checking the normal of the plane the polygon lies in. If any of the entries are 0, the polygon is parallel to this axis, and it must be preserved. In general, the smaller an entry is in the normal vector, the larger the polygon will vary in that axis. Thus to minimize the amount of information lost about the polygon, we find the largest entry in the normal and always drop that axis.

Once we have chosen an axis to drop, we remove it from all points on the bounding polygon, as well as from the point we are testing, and run Algorithm 1 to test for an intersection. If we find a point of intersection, we need find the surface normal of the plane or polygon at that point.

2.2.3 Finding Plane Surface Normals

Finding the surface normal of a plane is almost trivial, since a plane is defined by a normal and a point. Similar to the sphere, there are two possible surface normals we can return for a plane. If the plane is defined by normal vector $\mathbf{N}$, both the vector $\mathbf{N}$ and $-\mathbf{N}$ are possible surface normals, depending on what side of the plane the incident vector originates. As before, we want the dot product of the normal and the incident vector to be less than zero, so we choose whichever normal satisfies that requirement.

CHAPTER 3

THE PHYSICS OF RAY TRACING

Finding ray-object intersections is an important part of ray tracing, but it is only half the battle. Once we have traced rays into the scene, we must examine the objects they collide with, and the light incident to those objects, to determine what that ray contributes to the image as a whole. Some basic questions are:

- Is the object illuminated in the scene or is it in the shadows?
- If a light source illuminates the object, how much light does the object reflect and how does this reflection change the wavelength of that light?
- Is the object reflective? Can I see nearby objects reflected in the surface, and if so, how strong is this reflection?
- Is the object translucent? How does the object change light that it transmits?
- Given all these variables, how do we perceive a point on this object? What color and intensity light do we see?

To properly answer all of these questions we must delve into the physics behind light-surface interactions. We will examine the different paths light can take when it collides with an object. We must also explore the relationship between wavelength and color, and how we perceive the visual world.

3.1 Light and Color

Before we discuss light-surface interactions, we must understand some basic properties of light. Visible light is electromagnetic radiation whose wavelength falls in the visible spectrum, between 380-750 nanometers. However, when we perceive light visually, the photo-receptors in our eyes are not merely absorbing individual photons, one at a time.

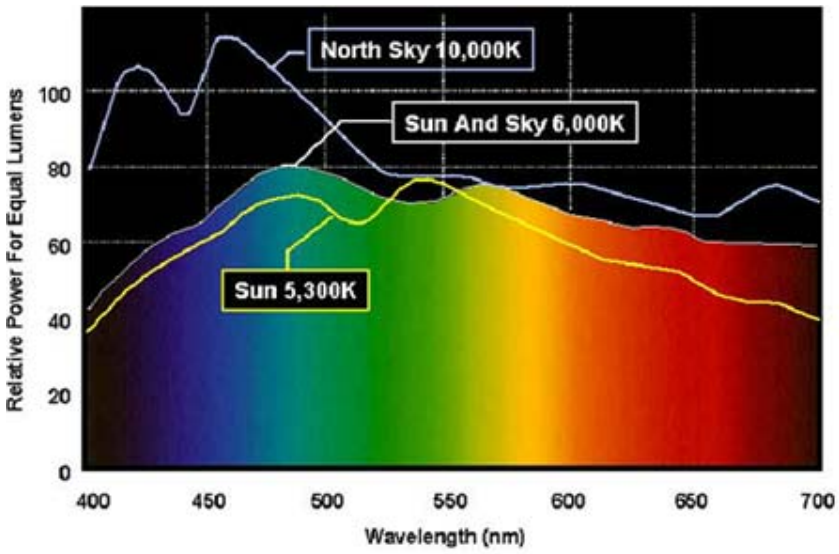

Figure 3.1: Spectral distribution for the sun [7]

Instead they are continuously absorbing a barrage of photons that they must interpret in some cohesive manner. In ray tracing, we face a similar problem.

Our approach is to model this barrage of waves that make up what our eyes are actually absorbing as a wave packet. This packet models a conglomeration of photons, that might all have different wavelengths and intensities. We model this variety as a distribution curve of wavelengths across the visible spectrum.

3.1.1 Wave Packets

Wave packets are simply a description of a group of photons. We model the group as a curve across the visible spectrum representing the intensity of the energy of the photons at that wavelength. Here are some examples of the spectral curve over the visible spectrum.

Solar spectral distribution (Figure 3.1) is governed by the laws of black body radiation. It has a smooth curve that is centered directly on the visible spectrum.

The spectral distribution of an incandescent light bulb (Figure 3.2) also demonstrates

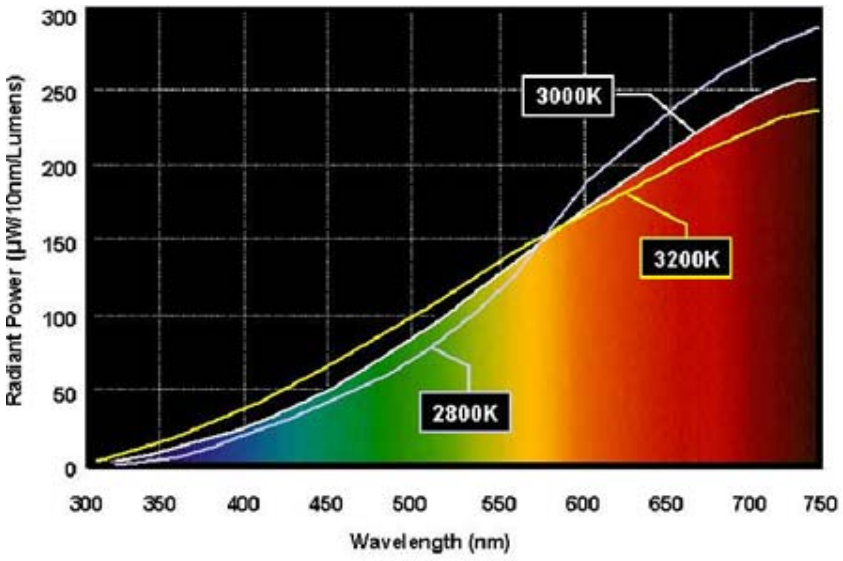

Figure 3.2: Spectral distribution for an incandescent light bulb [7]

black body radiation, but as light bulbs are significantly colder than the sun, the curve is centered over the infrared spectrum, hence the upward sloping curve in the visible spectrum.

Fluorescent lamps have a strikingly different spectral distribution (Figure 3.3) as they do not emit black body radiation. Instead they rely on the fluorescence of doped phosphors to produce light. Each phosphor emits a narrow band of radiation, hence the spikes in the curve, instead of the smooth curves we just looked at. [6]

These examples should give you an idea of the variety of waveforms light packets can take. Interestingly, though their wave forms are drastically different, each of these examples represents white light. You might ask, why represent the full wave form in our algorithm then, if each of these waves should appear as white light? We want to maintain as much information about the scene as possible to get our model as close to reality as possible. All three of these lights might appear the same if viewed directly, or reflected off of a white surface, but they would illuminate a scene differently due to their different distributions. The incandescent light would emphasize reds and yellows in a scene while daylight and fluorescent light might highlight bluer tones.

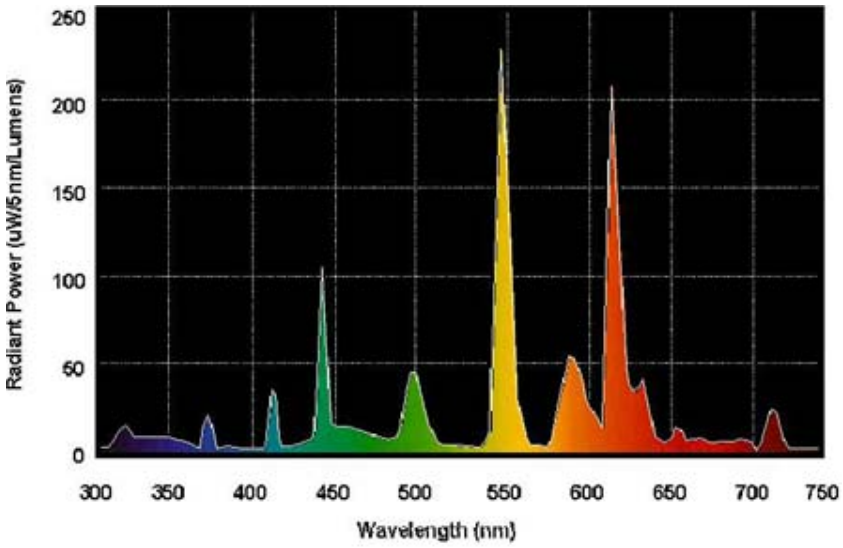

Figure 3.3: Spectral distribution for a fluorescent light bulb [7]

3.1.2 Converting Wavelength to RGB

Translating from wavelength to RGB color space can be tricky. Translating individual wavelengths is easy; for example the pure wavelengths 700 nm, 546 nm, and 436 nm correspond to the primary colors red, green, and blue. However, many colors and hues do not have associated pure wavelengths, but instead are the result of our eye averaging a distribution of wavelengths. There is no individual wavelength that our eye perceives as magenta colored, since magenta results from a distribution of blue and red light.

We need some way to simulate the averaging that our eyes perform so we can translate these wave packets into some meaningful data about their color. The International Commission on Illumination studied this problem in the early 1930s and developed color matching functions that allow us translate wavelength data into RGB color values. They use a slightly modified color space called the CIE XYZ color space, where X, Y, and Z roughly correlate to standard RGB values. There are post processing steps that we can do after we have the XYZ values of a wave packet to get standard RGB color. [5] Their color matching functions are shown in Figure 3.4.

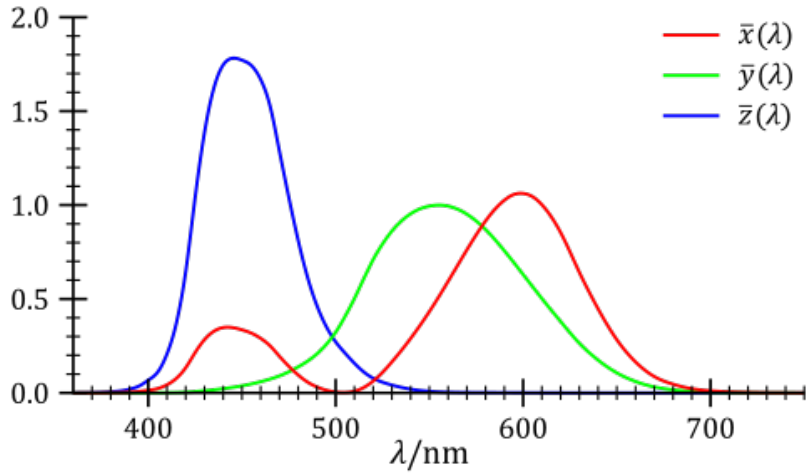

Figure 3.4: The CIE XYZ color matching functions [1]

To translate a wave form into XYZ coordinates, we simply multiply it by each of these functions, and the area under the resulting curve is the value for that respective variable. The XYZ values for a wave form I are:

$$X = \int_{380}^{750} I(\lambda)\bar{x}(\lambda)d\lambda \tag{3.1}$$

$$Y = \int_{380}^{750} I(\lambda)\bar{y}(\lambda)d\lambda \tag{3.2}$$

$$Z = \int_{380}^{750} I(\lambda)\bar{z}(\lambda)d\lambda \tag{3.3}$$

Now that we have the color in the XYZ color space, we want to translate it into an RGB color space so that we can display it. One of the most used color spaces is $sRGB$, a standardized RGB color space developed by Hewlett-Packard and Microsoft for color management on the internet. [11] The standard defines a simple matrix to convert CIE XYZ colors to $sRGB$.

$$\begin{bmatrix} R_{sRGB} \\ G_{sRGB} \\ B_{sRGB} \end{bmatrix} = \begin{bmatrix} 3.2410 & -1.5374 & -0.4986 \\ -.9692 & 1.8760 & 0.0416 \\ 0.0556 & -0.2040 & 1.0570 \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.4}$$

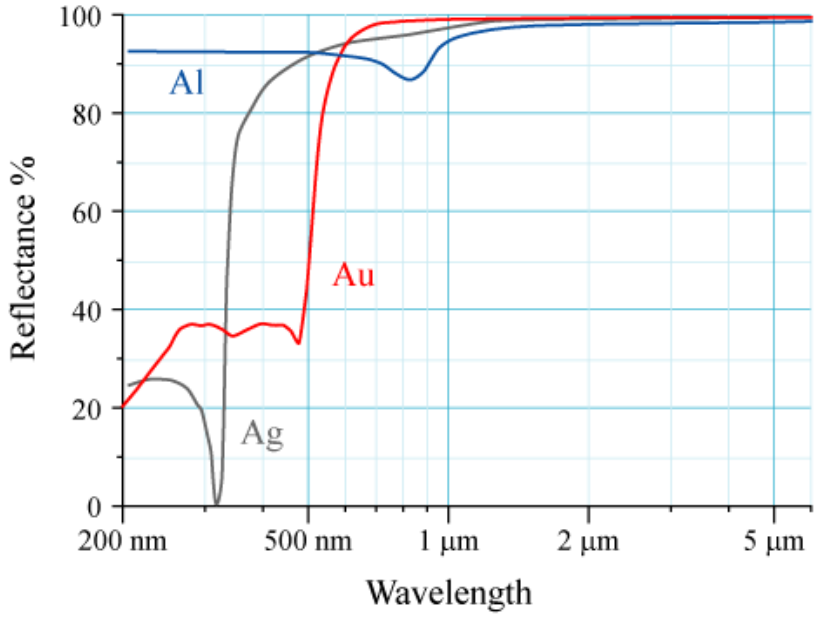

Figure 3.5: Reflectance curves for common metal surfaces. [8]

3.1.3 Reflectance Curves

In most scenes, light does not just emanate from a source and go straight to the observer, it bounces off of objects in the scene first. When light strikes an object, typically some wavelengths are absorbed, while others are reflected back into the scene. Thus the waveform of the light changes depending on which wavelengths are reflected and which are absorbed. We model this process by defining reflectance curves for each object. Reflectance curves are similar to the spectral curves we defined for wave packets. They range the visible spectrum, and take values from 0 to 1, depending on the amount they reflect of a given wavelength. Some example reflectance curves are shown in Figure 3.5.

If a wave packet with spectral distribution I reflects off of a surface with a reflectance distribution R, the reflected light would have a distribution D such that:

$$D(\lambda) = I(\lambda)R(\lambda)$$

3.1.4 Implementing Spectral Distributions and Reflectance Curves

It would be too computationally expensive to store and process general curves for spectrum distributions in our program. Instead, we can represent these curves as vectors of values, sampled at regular intervals. This greatly reduces the amount of time to perform basic operations on the distributions. If we want to scale a distribution by some amount we perform scalar multiplication on the vector. Multiplying two distributions simply becomes multiplying corresponding entries in the two vectors. The integral of a distribution is the sum of the components of the vector.

We can sample at a higher frequency on smaller intervals to increase the nearness of our approximation, or we can use fewer samples to sacrifice accuracy for speed.

3.2 Surface-Light Interaction

Now that we understand how to represent light and reflectance curves, and how to find ray-object intersections, we can start tracing rays into the scene and interpreting the results. A few things can happen when light intersects an object. If the surface is shiny, like a mirror or polished obsidian, the light could bounce off, like a ball bouncing off a hard surface. If a surface is matte, like a piece of paper, the light will scatter evenly in all directions. If the surface is transparent, like glass or diamond, the light will pass through it, refracting by some angle based on the substance it is passing through. These are examples of specular reflection, diffuse reflection, and transmission, three of the basic mechanisms of light-surface interaction.

In practice, a surface is not purely specular, diffuse, or transmissive, but some combination of the three. When we trace a ray out into the scene and it intersects an object, we must must account for all mechanisms of light transport when calculating the light that we see from that ray. If the object is being illuminated by a light source, we must

account for the light that is diffusely reflected from that light source. If the surface is glossy, we must reflect the incident ray and trace the new ray into the scene to see what light could be specularly reflected by the surface. If the surface is transparent, we must trace the ray as it refracts through the object to see if any light is being transmitted. All of these components add together to form the total light that returns along the incident ray.

Thus we can define any surface as having three coefficients for each of the mechanisms of light transport. Generally, when light hits an object, all of the light must be accounted for in one of these three mechanisms, so the coefficients should sum to one. Each coefficient indicates the proportion of light that undergoes each mechanism of interaction. In practice we have some freedom to play with these numbers. For instance if you wanted a particularly dark object you could make the coefficients sum to less than one, simulating extreme light absorption by the object. If you wanted an object to glow a little brighter in a dark scene you could add a little more onto the diffuse coefficient so that it actually introduces light into the scene instead of absorbing it.

3.2.1 Illumination and Shadows

To determine how an object or a point on an object appears in a scene, we must first determine if the object is being illuminated by any of the light sources in the scene. The algorithm to do this is straightforward.

If ray $\mathbf{r}$ collides with an object at point $\mathbf{p}$, start by iterating through all the light sources in the scene. If light source i is at point $\mathbf{l}_i$, create a new ray that starts at point $\mathbf{p}$ and travels along the vector $\mathbf{l}_i - \mathbf{p}$. This ray travels from the intersection point to the light source. We trace this ray through the scene and if we intersect with any other objects before we reach the light source, we know the point is shaded. Otherwise the point is illuminated.

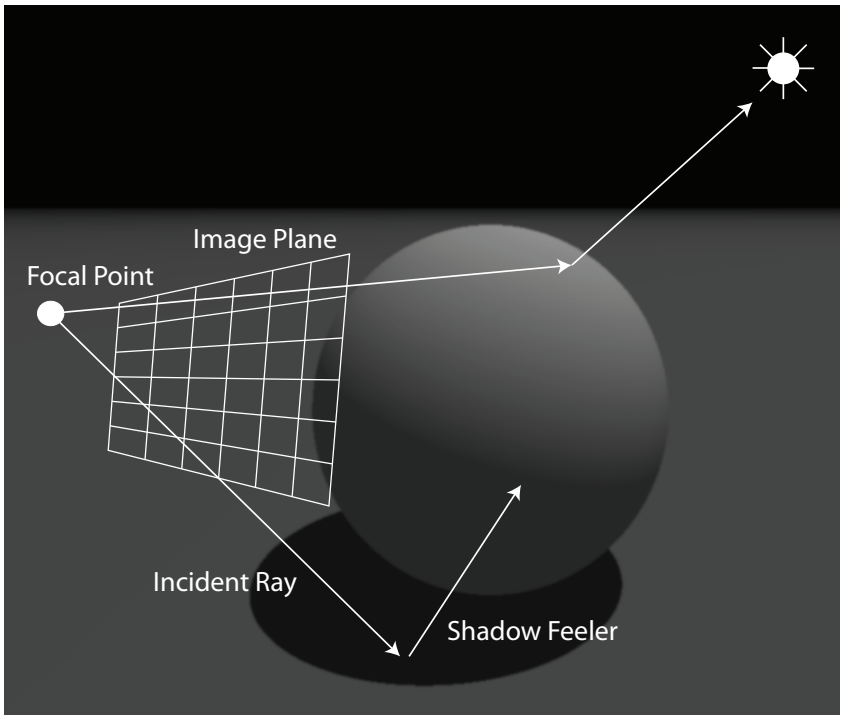

Figure 3.6: Shadow feelers test to see if a point is illuminated

This ray is called a shadow feeler. We can use the same algorithms for tracing shadow feelers through a scene as with any normal ray, however we can optimize our performance by halting the tracing process as soon as we find any intersection point that is closer than the light source, since as soon as we find one object in the way, we know the point is shaded.

This approach to shadows is easy to understand and implement, and is built on top of fundamental ray tracing concepts. Native shadow support is one advantage ray-tracing has over rasterizing programs like OpenGL, which require advanced methods for rendering shadows in scenes. In fact one such method to render shadows in OpenGL is to write a basic ray tracing algorithm that only supports shadow feelers and use its output to determine what to shade in the scene.

When implementing shadow feelers, be aware of potential errors that can result from floating point arithmetic. When generating a shadow feeler from an intersection point,

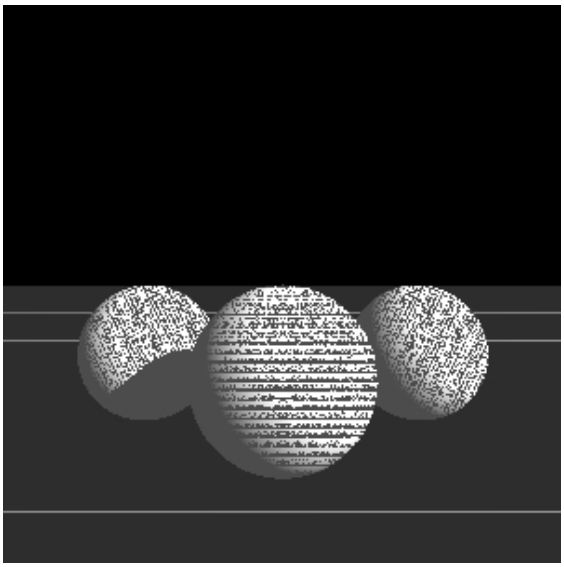

Figure 3.7: Floating point errors when calculating shadow feelers can result in surfaces shading themselves. This splotchy appearance is the result.

small floating point errors can cause the feeler to originate slightly inside the object, and then collide with the object at, or very close to the original intersection point. In effect this causes parts of the object to shade themselves, something we do not want to happen. Add in an extra check to make sure the feeler travels some non-trivial distance before it starts looking for intersections to avoid this issue.

3.2.2 Diffuse Reflection

Diffuse reflection models the scattering that occurs when light strikes a rough or matte surface and accounts for the base color of an object. During diffuse reflection the material and the light interact a great deal, and the object absorbs or reflects the light based on its reflectance curve.

In ideal diffuse reflection, light is scattered in all directions evenly. The magnitude of the reflected light is only proportional to the angle at which the light strikes the surface, and is independent of the angle of the viewing vector relative to the surface.

In practice, the magnitude of the reflected light is directly proportional to the cosine

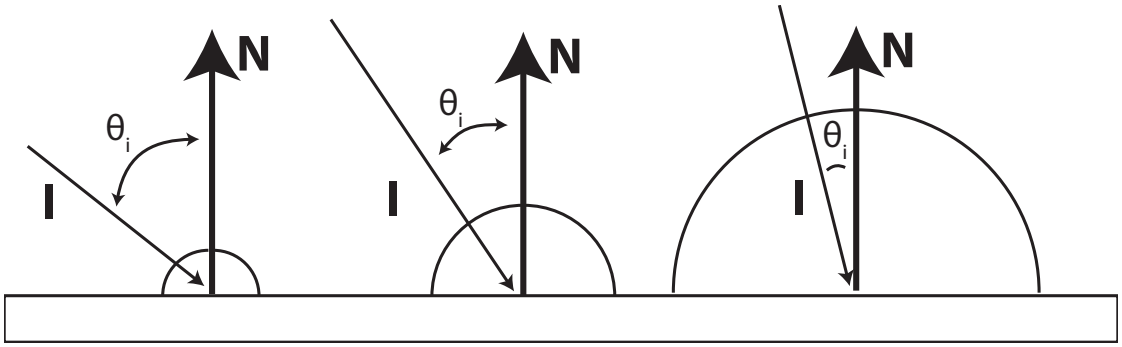

Figure 3.8: Diffuse reflection. The magnitude of the scattered light is proportional to $\cos(\theta_i) = \mathbf{N} \cdot \mathbf{I}$

of the angle between the surface normal and the direction of incident light, a property known as the "cosine law" [9]. The magnitude of reflected light will also be proportional to the amount of light that is diffusely reflected, as opposed to specularly reflected or transmitted. This makes modeling diffuse reflection easy, as we only need to consider the surface normal and the vector at which light hits the surface as well as the diffuse coefficient of the surface.

When calculating diffuse reflection for a point from a given light source, we must first determine if the point is being illuminated by that light source. The shadow feelers discussed above accomplish this task for us, and also give us a vector describing the direction of incident light from the light source if the object is not shaded, it is simply the reverse of the vector that describes the shadow feeler.

If we are viewing a point $\mathbf{P}$ on a surface with diffuse coefficient d, surface normal $\mathbf{N}$, and reflectance curve $F(\lambda)$ with incident light with spectral distribution $I(\lambda)$ coming from vector $\mathbf{L}$ we calculate the distribution of the diffusely reflected light as follows:

$$R(\lambda) = F(\lambda)I(\lambda)(\mathbf{N} \cdot \mathbf{L})d \tag{3.5}$$

To get the total amount of diffuse reflection at a point, we must loop over all light sources that illuminate the point and sum the results of (3.5). This model only accounts for diffuse reflection originating from light sources, while in reality, light rays that orig-

inate from elsewhere in the scene, and bounce around, eventually hitting a point can also contribute to its total diffuse reflection. It is a good idea to add in an extra step to model this ambient light. We can add in an extra spectral distribution to model the wavelength and intensity of the ambient light in a scene. The calculation to find the amount of diffuse reflection at a point from ambient light is as follows:

$$R_{ambient}(\lambda) = F(\lambda)I_{ambient}(\lambda)d \tag{3.6}$$

This is the simplest option for modeling ambient light, however there are other, more accurate approaches. For instance we could generate a random sampling of rays from the point out into the scene to gather data about the inbound light from sources other than light sources. This would give a more accurate representation of the ambient light arriving at each individual point, but at the cost of performance.

3.2.3 Specular Reflection

Specular reflection can contribute to the light returned from an incident vector in two ways. The first is pure specular reflection, where light from elsewhere in the scene bounces off of the surface and toward the image plane. Mirrors and smooth lakes exemplify pure specular reflection.

Specular surfaces can also reflect light incident from light sources, creating specular highlights. Because we model light sources as points, our algorithm will not factor in light coming directly from light sources as a possible source for pure specular reflection. Instead, we handle specular highlights separately. Glare from a car driving past on a sunny day, or from a shimmering lake, are examples of specular highlights.

Pure Specular Reflection

To model specular reflection, we reflect the incident ray off of the object and recursively call our tracing algorithm on the reflected ray to find any light that may be hitting the

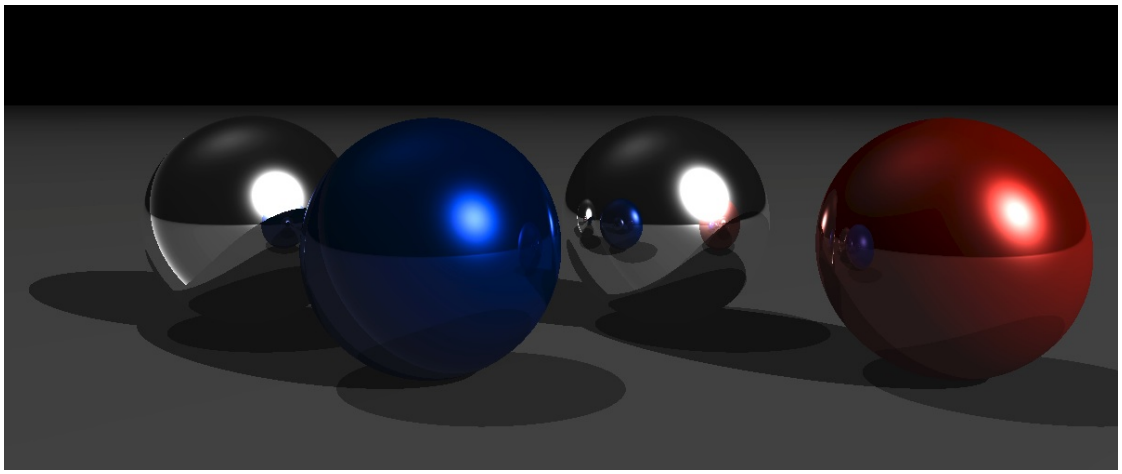

Figure 3.9: An example of pure specular reflection and specular highlights from two light sources

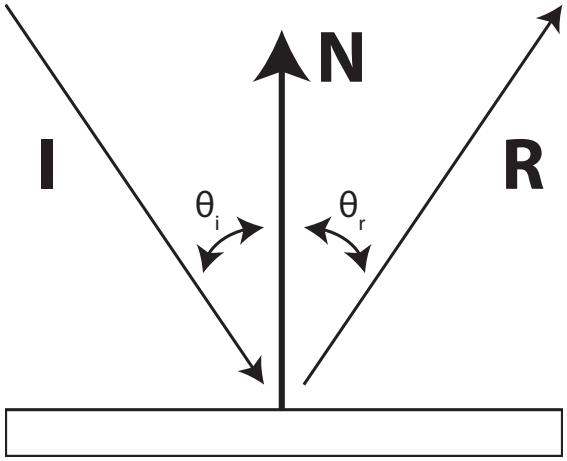

Figure 3.10: Specular reflection, $\theta_i = \theta_r$

object at the intersection point. We scale this light by the specular coefficient for the surface to find the final amount specular reflection contributes to the return light.

First we must find the reflected ray so we can recursively trace it into the scene. To find the reflection vector $\mathbf{R}$ of normalized incident vector $\mathbf{I}$ given surface normal $\mathbf{N}$, we rely on the property that the reflected vector will lie in the same plane as the incident vector and the surface normal, and that it will form the same angle with the surface normal as the incident vector does. The following derivation comes from Andrew Glassner's *An Introduction to Ray Tracing* [3].

Since the $\mathbf{R}$, $\mathbf{I}$, and $\mathbf{N}$ all lie in the same plane, we can represent one as the linear combination of the other two:

$$\mathbf{R} = \alpha\mathbf{I} + \beta\mathbf{N} \tag{3.7}$$

If we reverse $\mathbf{I}$, we also know that $\mathbf{N}$ and $-\mathbf{I}$ form the same angle that $\mathbf{N}$ and $\mathbf{R}$ form. We have to reverse $\mathbf{I}$ so that both vectors point away from the point of intersection. This implies that:

$$-\mathbf{I} \cdot \mathbf{N} = \mathbf{R} \cdot \mathbf{N} \tag{3.8}$$

Substitution our formula for $\mathbf{R}$ we get:

$$\begin{aligned} -\mathbf{I} \cdot \mathbf{N} &= \mathbf{N} \cdot (\alpha\mathbf{I} + \beta\mathbf{N}) \\ &= \alpha(\mathbf{N} \cdot \mathbf{I}) + \beta(\mathbf{N} \cdot \mathbf{N}) \\ &= \alpha(\mathbf{N} \cdot \mathbf{I}) + \beta \end{aligned} \tag{3.9}$$

since $\mathbf{N} \cdot \mathbf{N} = 1$. If we set $\alpha = 1$, we get:

$$\beta = -2(\mathbf{N} \cdot \mathbf{I}) \tag{3.10}$$

So our final equation for the reflected vector is:

$$\mathbf{R} = \mathbf{I} - 2(\mathbf{N} \cdot \mathbf{I})\mathbf{N} \tag{3.11}$$

We combine this $\mathbf{R}$ with the intersection point to for the reflected ray. Tracing this ray into the scene yields the light that travels back along the ray and reflects off of the surface, and into the image plane. We simply multiply the resulting distribution from tracing $\mathbf{R}$ by the specular coefficient of the object to see how much this reflected light contributes to the final light returned from the incident ray.

Generally a light ray reflecting off of a surface interacts with the surface itself very little, so the spectrum of the reflected ray is almost unchanged from the incident ray. For this reason we can safely leave out any information about the reflectance of a surface for specular reflection.

In the real world there is some interaction between the surface and incident light, even if the light is specularly reflected. A surface's reflectance curve will change proportional to the angle of the incident light. As the incident light approaches an orthogonal angle to the surface normal, there is less interaction between the surface and the light, leading to a reflectance curve that is uniformly one [2]. We can extend our model by multiplying the reflected light by a calculated reflectance curve for the angle of incidence, but the details are not covered here.

Specular Highlights

Specular highlights result when light from a light source reflects off of a surface and towards the viewer. We cannot account for specular highlights by tracing the reflected viewing ray backward because light sources are modeled as points, so we would never find an intersection. Instead we reflect the ray incident light would take off the surface and compare this to the viewing vector. If the angle between these two is close, the reflected light nearly reflects into the viewer, and we give that spot more specular reflection. If the angle is large, the reflected light does not pass very close to the viewer so we do not highlight the area as much, if at all.

Different surfaces display different types of specular highlighting. An extremely smooth surface will have a very sharp, bright highlights. A rougher surface will have duller, and more spread out highlights. To simulate this affect, we add in a new coefficient that models how smooth the surface is. To see how this coefficient affects the final highlight, lets examine the full equation for specular highlights.

For a light source with wavelength $I(\lambda)$ striking a surface with specular coefficient s and smoothness value n, reflecting along vector $\mathbf{L}_r$ viewed along vector $\mathbf{V}$, the specular highlight $S_h(\lambda)$ is given by:

$$S_h(\lambda) = I(\lambda)(\mathbf{L}_r \cdot -\mathbf{V})^n s \tag{3.12}$$

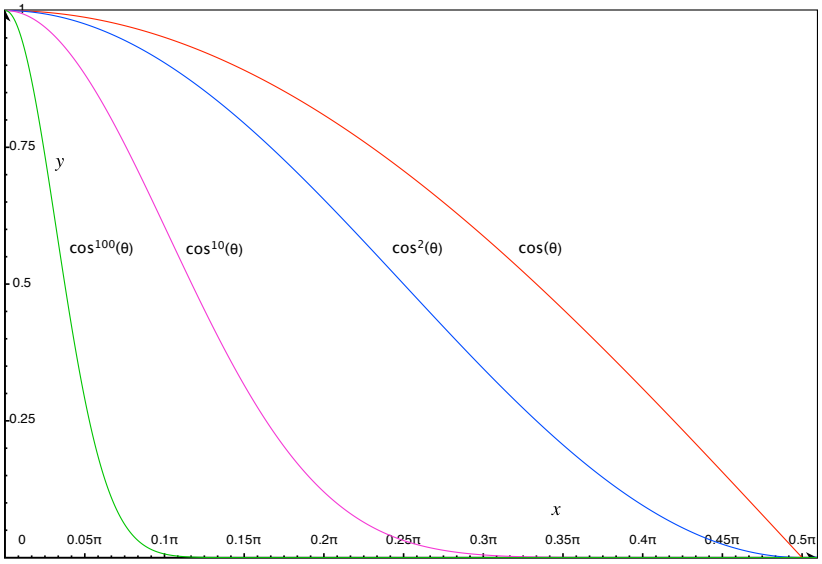

Figure 3.11: Curves for $\cos(\theta)$ raised to different powers. As the power increases, the curve's steepness increases, giving objects more precise, vivid highlights the smoother they are.

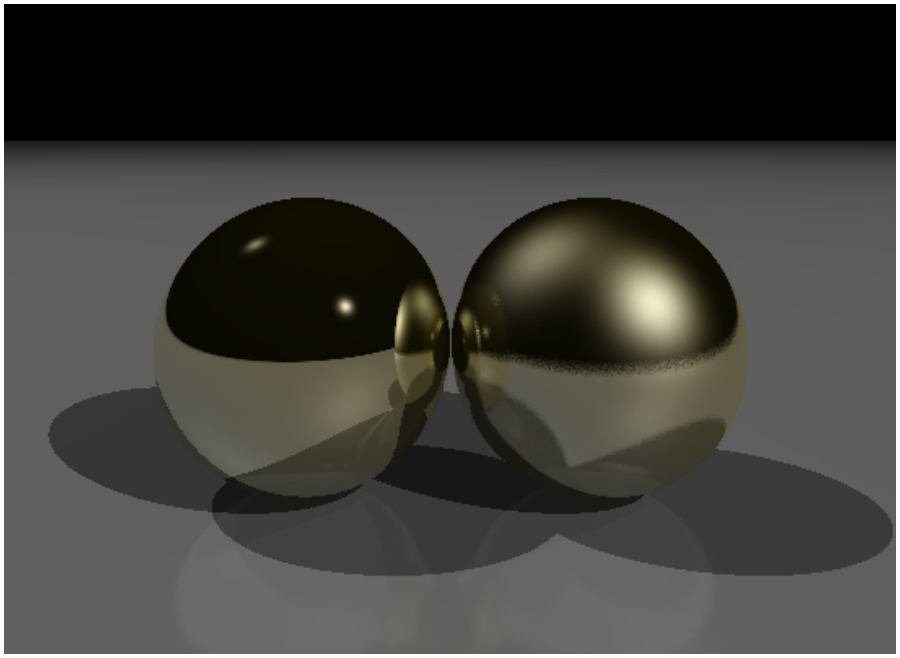

Figure 3.12: An example of using different smoothness values to simulate polished and brushed metal.

Material	Index of refraction
Air	1.0003
Water	1.33
Ice	1.31
Glass	1.485-1.925
Diamond	2.419

Table 3.1: Refractive indices of common materials

First we negate the viewing vector so that it will point away from the object, in the same direction as the reflected light. Then we find the cosine of the angle between these two vectors by taking their dot product. Finally we raise this value to the power of the smoothness value for the surface. By raising it to this power, we determine how steeply the curve falls off if the angle is not close to zero. This is shown in Figure 3.11. If the smoothness is high, fewer points will be highlighted, making the highlight sharper.

3.2.4 Transmission

Modeling transmission, like specular reflection, involves generating a new ray and tracing it through the scene. This new ray is generated traveling through the object instead of outside of it. When light enters a new material, it often bends, or refracts so it is traveling in a new direction We see this effect when looking through curved glass, or at an object partially submerged in water. This is due to the objects having different refractive indexes.

A material's refractive index is a measure of how fast light moves through that material. If a material has a refractive index of η, it means light travels $\frac{1}{\eta}$ times as fast through that medium as it does through a vacuum. Obviously a vacuum has a refractive index of 1. Some common refractive indexes are listed in Table 3.2.4.

When light enters a new material, it diffracts due to the change in velocity. The new angle is derived from the diffraction coefficients of the material it was in and the material

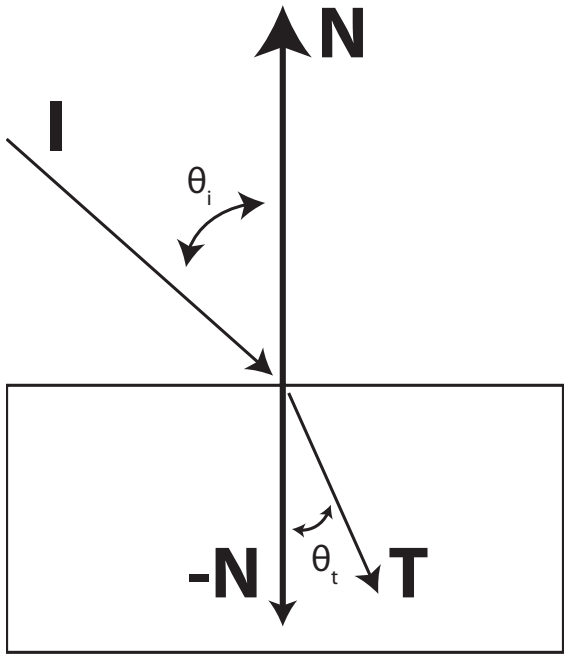

Figure 3.13: Transmission of light through a surface. θ_i and θ_t are related by their corresponding coefficients of refraction.

is entering. For some incident ray traveling along vector $\mathbf{I}$ in a surface with diffraction coefficient η_i entering a surface with diffraction coefficient η_t at a point with surface normal $\mathbf{N}$, the relationship of the angle of incidence θ_i to the angle of transmission θ_t is as follows (see Figure 3.13):

$$\eta_i \sin(\theta_i) = \eta_t \sin(\theta_t) \tag{3.13}$$

However, there are many different vectors that could form angle θ_t with $-\mathbf{N}$. Like in specular reflection, we add in the constraint that the true vector of transmission will be coplanar to both the surface normal and the incident vector, thus it can be written as a linear combination of the two:

$$\mathbf{T} = \alpha\mathbf{I} + \beta\mathbf{N} \tag{3.14}$$

With these two pieces of information, we can find what a concrete equation for $\mathbf{T}$. This basis for this derivation can also be found in Glassner [3]. First, we square both

sides of (3.13) and rewrite the sines as cosines (dot products).

$$\eta_i^2(1 - [\mathbf{N} \cdot \mathbf{I}]^2) = \eta_t^2(1 - [-\mathbf{N} \cdot \mathbf{T}]^2) \tag{3.15}$$

Now we can substitute in (3.14) for $\mathbf{T}$. We can also get rid of the -1 inside the square on the right side, leaving:

$$\begin{aligned} \eta_i^2(1 - [\mathbf{N} \cdot \mathbf{I}]^2) &= \eta_t^2(1 - [\mathbf{N} \cdot (\alpha\mathbf{I} + \beta\mathbf{N})]^2) \\ &= \eta_t^2(1 - [(\mathbf{N} \cdot \alpha\mathbf{I}) + (\mathbf{N} \cdot \beta\mathbf{N})]^2) \\ &= \eta_t^2(1 - [\alpha(\mathbf{N} \cdot \mathbf{I}) + \beta]^2) \end{aligned} \tag{3.16}$$

Rearranging we get:

$$\begin{aligned} 1 - \frac{\eta_i^2(1 - [\mathbf{N} \cdot \mathbf{I}]^2)}{\eta_t^2} &= [\alpha(\mathbf{N} \cdot \mathbf{I}) + \beta]^2 \\ &= \alpha^2(\mathbf{N} \cdot \mathbf{I})^2 + 2\alpha\beta(\mathbf{N} \cdot \mathbf{I}) + \beta^2 \end{aligned} \tag{3.17}$$

We are one equation describing two variables, so we still need more information about the system. We know that $\mathbf{T}$ should be a unit vector, so we can say:

$$\begin{aligned} 1 &= \mathbf{T} \cdot \mathbf{T} \\ &= (\alpha\mathbf{I} + \beta\mathbf{N}) \cdot (\alpha\mathbf{I} + \beta\mathbf{N}) \\ &= \alpha^2 + 2\alpha\beta(\mathbf{I} \cdot \mathbf{N}) + \beta^2 \end{aligned} \tag{3.18}$$

so

$$2\alpha\beta(\mathbf{N} \cdot \mathbf{I}) + \beta^2 = 1 - \alpha^2 \tag{3.19}$$

Substituting this into (3.17) we have:

$$\begin{aligned} 1 - \frac{\eta_i^2(1 - [\mathbf{N} \cdot \mathbf{I}]^2)}{\eta_t^2} &= \alpha^2(\mathbf{N} \cdot \mathbf{I})^2 + 1 - \alpha^2 \\ -\frac{\eta_i^2(1 - [\mathbf{N} \cdot \mathbf{I}]^2)}{\eta_t^2} &= \alpha^2[(\mathbf{N} \cdot \mathbf{I})^2 - 1] \\ \frac{\eta_i^2([\mathbf{N} \cdot \mathbf{I}]^2 - 1)}{\eta_t^2} &= \alpha^2[(\mathbf{N} \cdot \mathbf{I})^2 - 1] \end{aligned} \tag{3.20}$$

Canceling we have:

$$\frac{\eta_i^2}{\eta_t^2} = \alpha^2 \tag{3.21}$$

Whew! We made it! So:

$$\alpha = \pm\frac{\eta_i}{\eta_t}$$

You should feel a little disconcerted that α has two potential values. This makes sense though, considering there are two possible vectors in the plane that have an angle θ_t to $-\mathbf{N}$. However, we know that our transmitted vector will be traveling roughly the same direction as $\mathbf{I}$, so we want a positive value to satisfy the equation:

$$\mathbf{T} = \alpha\mathbf{I} + \beta\mathbf{N}$$

Both η_i and η_t are positive, thus to keep α positive we must have:

$$\alpha = \frac{\eta_i}{\eta_t} \tag{3.22}$$

Substituting our equation for α into (3.18) we get:

$$1 = \left(\frac{\eta_i}{\eta_t}\right)^2 + 2\left(\frac{\eta_i}{\eta_t}\right)\beta(\mathbf{I}\cdot\mathbf{N}) + \beta^2 \tag{3.23}$$

Using the modified quadratic formula from Section 2.1.1 we can solve for β:

$$\begin{aligned}\beta &= -\left(\frac{\eta_i}{\eta_t}\right)(\mathbf{I}\cdot\mathbf{N}) \pm \sqrt{\left(\frac{\eta_i}{\eta_t}\right)^2(\mathbf{I}\cdot\mathbf{N})^2 - \left(\frac{\eta_i}{\eta_t}\right)^2 + 1} \\ &= -\left(\frac{\eta_i}{\eta_t}\right)(\mathbf{I}\cdot\mathbf{N}) \pm \sqrt{1 + \left(\frac{\eta_i}{\eta_t}\right)^2[(\mathbf{I}\cdot\mathbf{N})^2 - 1]}\end{aligned} \tag{3.24}$$

Again we have two options, however some experimentation reveals that

$$\beta = -\left(\frac{\eta_i}{\eta_t}\right)(\mathbf{I}\cdot\mathbf{N}) - \sqrt{1 + \left(\frac{\eta_i}{\eta_t}\right)^2[(\mathbf{I}\cdot\mathbf{N})^2 - 1]} \tag{3.25}$$

is the correct solution.

At last we have a solution for $\mathbf{R}$!

$$\mathbf{R} = \frac{\eta_i}{\eta_t}\mathbf{I} - \left[\left(\frac{\eta_i}{\eta_t}\right)(\mathbf{I}\cdot\mathbf{N}) + \sqrt{1 + \left(\frac{\eta_i}{\eta_t}\right)^2[(\mathbf{I}\cdot\mathbf{N})^2 - 1]}\right]\mathbf{N} \tag{3.26}$$

We now take this equation for $\mathbf{R}$ and combine it with the intersection point to form a new ray. We recursively trace this ray through the scene, and whatever light information it returns back with, we multiply by the transmissive coefficient of the material to find the transmissive component of light at the intersection point.

Note that in our equation for $\mathbf{R}$, β can be non-real if the coefficient in the square root is negative. This is due to a phenomenon called total internal reflection. If light intersects a surface at too shallow of an angle, it can be deflected away completely, even if the surface has a transmissive component. Jewel cutters use this effect to make gems sparkle, cutting them in a way that light can only exit from certain facets. This reflected light takes the same path as if the light were specularly reflected. Thus if we obtain a non-real β, we simply pretend the reflection is specular instead of transmissive, find the appropriate reflected vector, and proceed as usual.

CHAPTER 4

EXAMPLES

With just these basic algorithms, and with little optimization, ray tracers can generate some truly marvelous images. These examples should give you an idea of the quality and realism possible with ray tracing.

The following images were rendered on a 2.4 GHz Core 2 Duo iMac using a ray tracer written in C++ and saved as bitmaps using the CImg library.

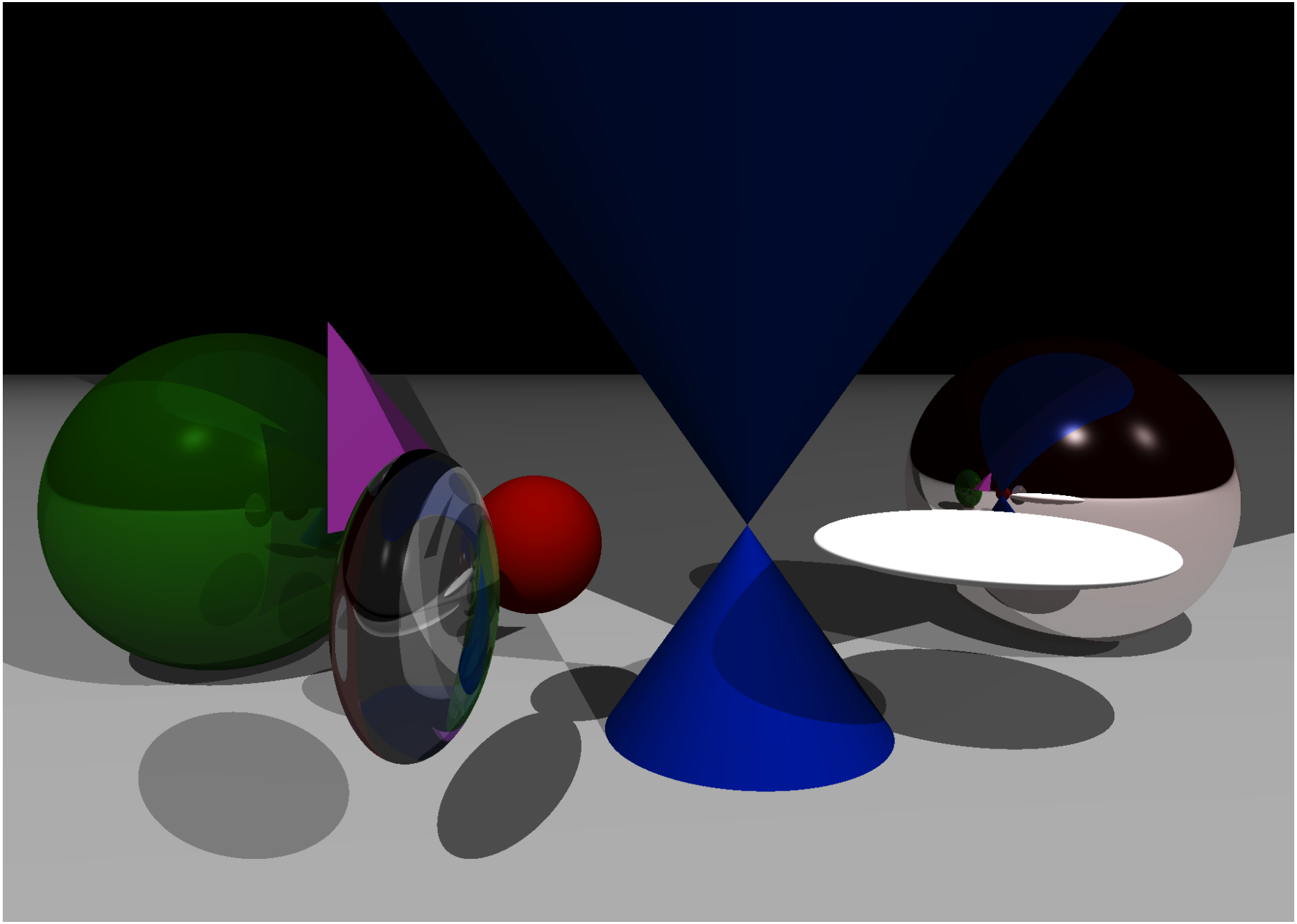

Figure 4.1: Demonstration of basic program functionality. 2 light sources illuminate spheres, quadrics, and polygons. The image was rendered at 2000x2000 pixels in 2 minutes and 14 seconds (approx. 38,000 pixels/sec).

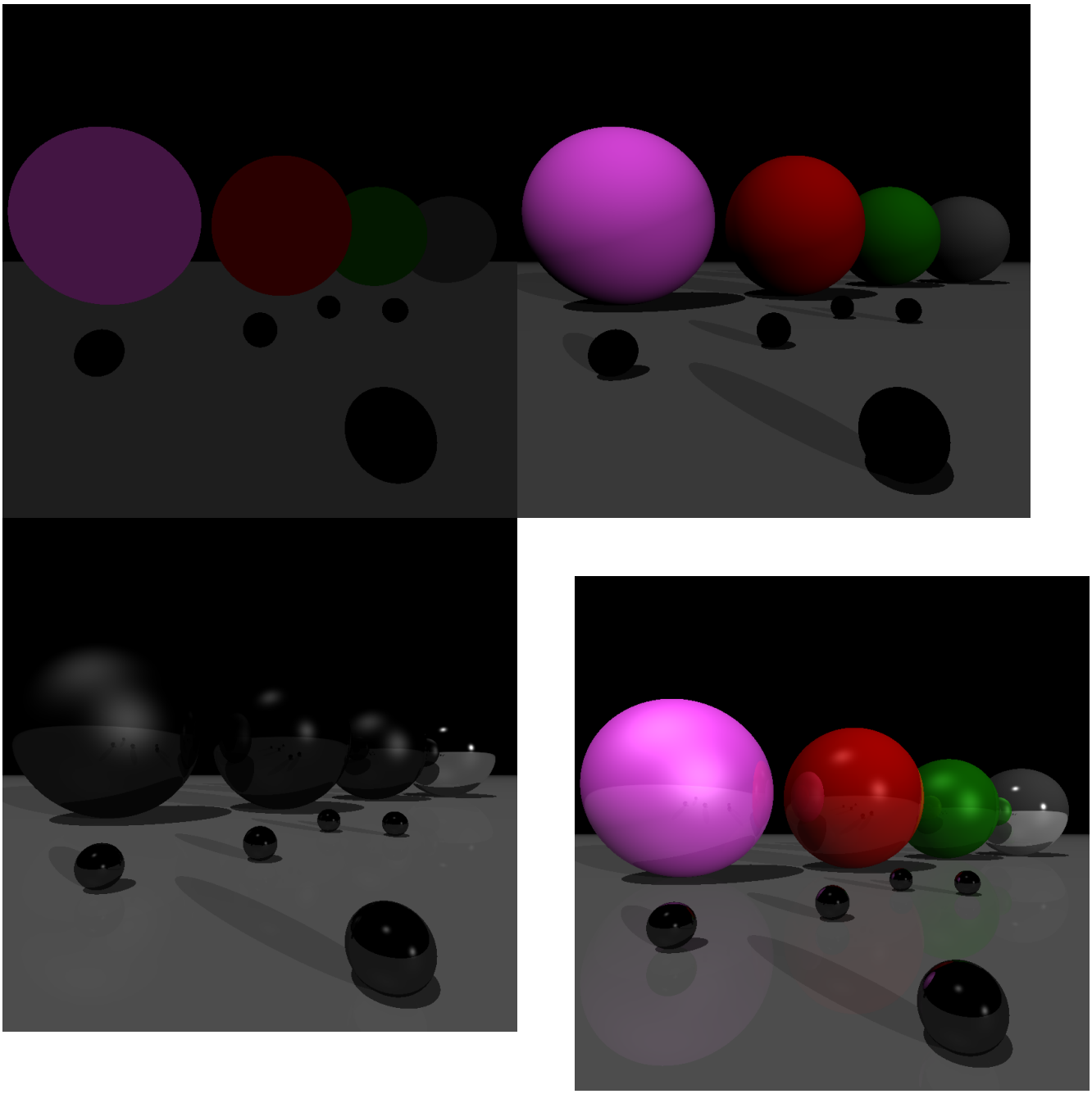

Figure 4.2: Demonstration of the shading model. Ambient, diffuse, and specular components sum to make the final image, but something is still missing...

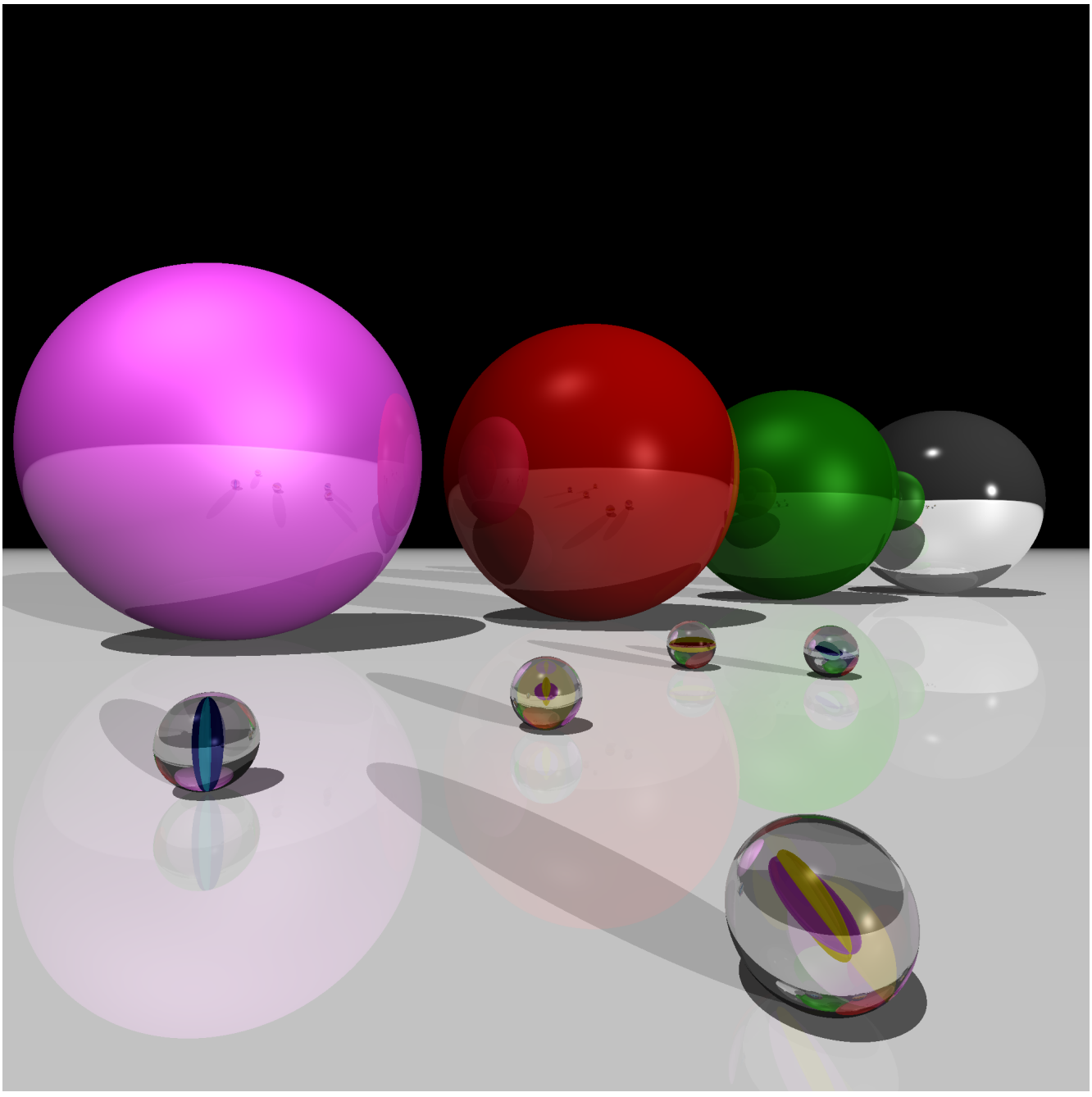

Figure 4.3: Adding in transmission, we get the full picture! This image took 4 minutes and 21 seconds to render at 2000x2000 pixels.

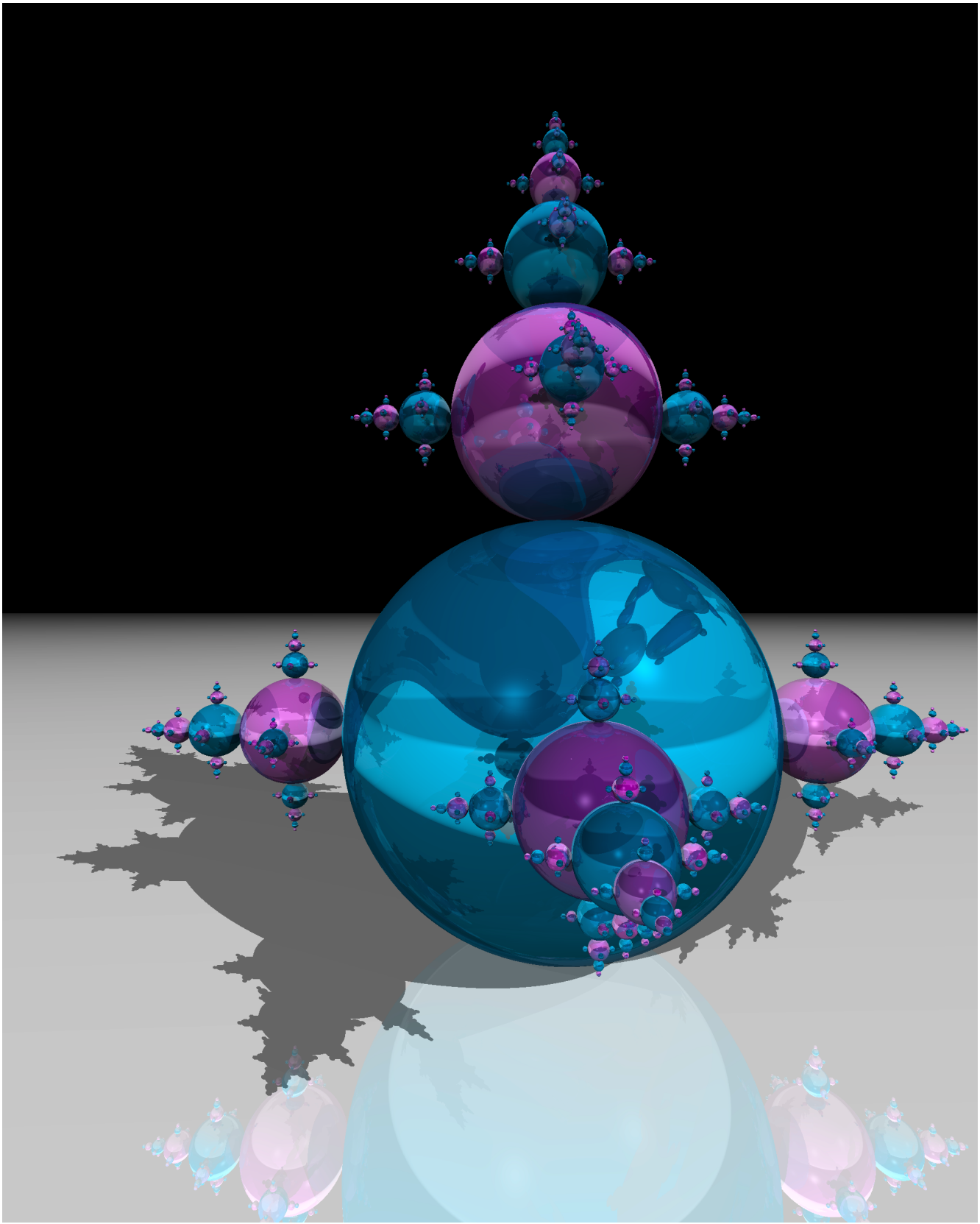

Figure 4.4: The geometry for this image was generated with a python script and then rendered using the ray tracer. It contains over 700 spheres and took 85 minutes and 20 seconds to render at 2000x2500 pixels.

APPENDIX A

SOURCE CODE

Source code for this thesis can be found at `https://github.com/pjreddie/RayTracer`.

BIBLIOGRAPHY

[1] Acdx. Cie 1931 xyz color matching functions. http://en.wikipedia.org/wiki/File:CIE_1931_XYZ_Color_Matching_Functions.svg.

[2] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *SIGGRAPH Comput. Graph.*, 15:307–316, August 1981.

[3] Andrew S. Glassner, editor. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989.

[4] Richard Hacker. Certification of algorithm 112: Position of point relative to polygon. *Commun. ACM*, 5:606–, December 1962.

[5] Gernot Hoffmann. Cie color space. http://www.fho-emden.de/~hoffmann/ciexyz29082000.pdf.

[6] F.A. Jenkins and H.E. White. *Fundamentals of Optics*. McGraw-Hill, 2001.

[7] GE Lighting. Spectral power distribution curves. http://www.gelighting.com/na/business_lighting/education_resources/learn_about_light/distribution_curves.htm.

[8] Bob Mellish. Image-metal-reflectance. http://en.wikipedia.org/wiki/File:Image-Metal-reflectance.png.

[9] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18:311–317, June 1975.

[10] M. Shimrat. Algorithm 112: Position of point relative to polygon. *Commun. ACM*, 5:434–, August 1962.

[11] Michael Stokes, Matthew Anderson, Srinivasan Chandrasekar, and Ricardo Motta. A standard default color space for the internet: srgb. Technical Report 1.10, International Color Consortium, Hewlett-Packard, and Microsoft, 1995.