Light propagates along rays, each of which can be represented by a point and a direction. This chapter sketches a mathematical representation and software implementation of a light ray. It then defines the **light field**, a mathematical formalization of the light in a scene in terms of rays. The light field function is defined in terms of itself by an integral equation. I give an intutitive explanation of this equation, which is then refined in the subsequent chapter.

Digital Images are Measurements

A real digital camera is a measurement device. It measures the amount of light striking each pixel of its sensor and records it as a number. The resulting digital image is just a two-dimensional array of the measurements. Given that digital image, a display can later synthesize light in the same pattern to reproduce the visual stimulus of the original scene.

Since images are measurements, rendering is measurement problem. We want to simulate the amount of light that a virtual camera measures in a virtual scene. We can then present that synthetic image as a real visual stimulus to a human observer.

To produce a rendering algorithm, we require a quantitative model of “light” under which we can make measurements. Measurements are taken with respect to a domain and have ranges in units. For example, in measuring the distance that you can walk in an hour, the hour is the measurement domain and the range units are distance units, such as meters. Our measurements will be virtual, but need to model their counterparts in the real world so that the result is a realistic image.

Rays of Light

Light propagates along a ray through a medium. You can only see light rays that enter your eyes, which means that you're seeing those rays end-on and they look like points. The collections of points form the image that you perceive. A light ray directed somewhere other than your eye sends its energy to that other location and is never measured by your eye, so it remains unseen by you.

However, you can see the area around the *paths* that a large collection of related light rays take through a dispersive medium such as fog. This is a nice natural visualization of light paths and gives some intuitive support for the notion of a light ray. These **crepuscular rays**, often informally called “God rays” when they emerge from clouds, are the shafts of illuminated medium visible when a relatively bright and tight bundle of light rays enters it. The “rays” in this case are figuratively named, since the rays that you actually observe are the ones coming from the illuminated fog to your eye, not the ones that pass through the fog to the ground.

Based on an image by Fir0002 / Flagstaffotos under the GDFL 1.2 license.

Crepuscular Ray [crpsclrDgrm]

A mathematical **ray** is described by a point and a direction. It is one half of a line, which extends infinitely in each direction. The rays in computer graphics begin at some point of interest, such as a light-emitting or light scattering surface. The actual light travels along a finite **line segment** to another surface, not an infinite ray, but when we're in the middle of a computation we often won't know where the terminating end of the segment is. So we consider the ray formed by the known point of origin and known direction of propagation, and use the geometry of that ray to later construct the necessary segment.

Transport Paths

We already have a mathematical representation and software implementation of 3D points, from the Surface chapter. Points are described by three real numbers. Points in the scene (as opposed to abstract mathematical constructs) have distance units such as meters. I denote points with italic capital serifed letters, such as P and X. Their compontents in some reference are written in equations using the name of the point as a subscript, e.g., Let $P = (x, y, z); k = x_P + 2\text{m}$ and as fields in code, e.g.,

`Point3 P(x, y, z); ... float k = P.x + 2;.`

A **line segment** is described by a starting point and ending point; often distinguishing which is the starting point is useful because that gives the segment orientation, and the direction of propagation is significant for light traveling along the segment. For example, there are usually different amounts of light traveling in opposite directions on the segment.

A light transport **path** is a collection of line segments. It can be described by an ordered set of points, e.g., implemented as an array of point objects. The transport paths of interest for rendering originate at a surface that emits photons. This is interchangably called an **emitter** (general case), a **luminaire** (if a non-natural source, such as a bulb), or a **light source** (if relatively small in extent).

We now need a representation for directions in order to represent the rays along which the segments of the path lie.

Vectors

A geometric **vector** is the difference of two points. It has a magnitude and direction. By convention, vectors are written as italic lower-case serifed letters with an arrow hat, e.g., $\vec{v} = B - A$. Because they are differences of points, in 3D a vector has three components. It carries distance units if the points were in the scene. A straightforward implementation is similar to the `Point3` class:

```
class Vector3 {
public:
   float x, y, z;
   ...
};
```

Note that the structural type (the stored state) of a vector is the same as that of a point. However, the semantic type (the interpretation of the state) is distinct. One cannot add two points together because they are just positions in space. But one can add the vector offset between two points to another point, and can add two offsets together. It is a common practice to leverage a single implementation class for both points and vectors. This is convenient for both the class implentor and consumer. However, it muddles the semantics, which can lead to confusion and errors. I choose to split the difference in my own code. I use C++ `typedef`s to create aliases. In this design, `Vector3` and `Point3` are distinct names in source code read by programmers, but actually have the same implementation and are not distinguished by the compiler.

Geometric vectors are related to linear algebra vectors and general computer science vectors. We'll explore the relationship to linear algebra vectors later. Beware that I occasionally treat a 3D vector as an array of three values, e.g., $v[2] = z_{\vec{v}}$ in code; this is in the spirit of that relationship more than the computer science notation of a vector. In general, graphics practitioners tend to call call computer science "vectors" "dynamic arrays" or "buffers" for clarity.

Addition and substraction of vectors proceeds element-wise, e.g.,

$$(i, j, k) + (x, y, z) = (i + x, j + y, k + z)$$

Multiplication or division by a scalar applies to each element individually:

$$k(x, y, z) = (kx, ky, kz)$$

The Euclidean **magnitude** or **length** of a vector is the Euclidean (i.e., straight-line) distance between two points separated by that distance, which is given by the Pythagorean theorem:

$$\text{Let } \vec{v} = (x, y, z)$$
$$||v|| = \sqrt{x^2 + y^2 + z^2}$$

The length has the same units as the components of the vector.

Unit Vectors

A **unit vector** has length 1 — not 1m, but unitless 1. We can use unit vectors to express directions without concern for distance traveled in that direction, which is ideal for describing the direction of propagation of light. Unit vectors are conventionally written with caret hats, e.g., $\hat{\omega}, \hat{n}$.

The operation of creating a unit vector in the direction of an arbitrary vector is common. To do this, simply divide the vector by its length:

$$\hat{\omega} = \frac{\vec{v}}{||\vec{v}||}$$

This process is called **normalizing** the vector and is often supported by special instructions in computer graphics hardware since square root and division tend to be relatively expensive operations.

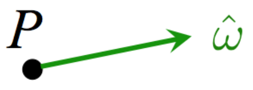
Unit vectors in 3D correspond to (abstract, unitless) points on the **unit sphere** $\mathbb{S}^2$. I'll declare them in mathematics as $\hat{\omega} \in \mathbb{S}^2$, and use the unit sphere as an integration domain to integrate an expression over all possible directions, e.g,

$$\int_{\mathbb{S}^2} \dots \, d\hat{\omega}.$$

Normalizing a vector projects it from the origin along its direction to the corresponding point on the unit sphere. I denote this operation with the sphere projection operator (in any dimension),

$$S(\vec{v}) = \frac{\vec{v}}{||\vec{v}||}$$

Geometric Rays



Geometric rays have a point P and a direction $\hat{\omega}$, so a natural representation for them in a rendering program is a class containing a point and a (typically unit) vector:

```
class Ray {
public:
    Starting point P
    Point3   origin;
    Unit vector ω̂
    Vector3  direction;

};
```

> ***Exercise* [ray]:** How many bytes does this `Ray` class consume in memory?

To reduce the number of distinct variables in the notation, for the next few chapters I always describe the point and vector explicitly in both source code and equations. However, you'll soon want to transition to using a class for rays in your renderer implementation.

The Surface chapter showed that geometry can be represented using either an explicit or implicit equation. The explicit equation for the ray with origin P in direction $\hat{\omega}$ is:

$$X(t \in \mathbb{R}_+) \in \mathbb{R}^n\mathrm{m} = P + t\hat{\omega}$$

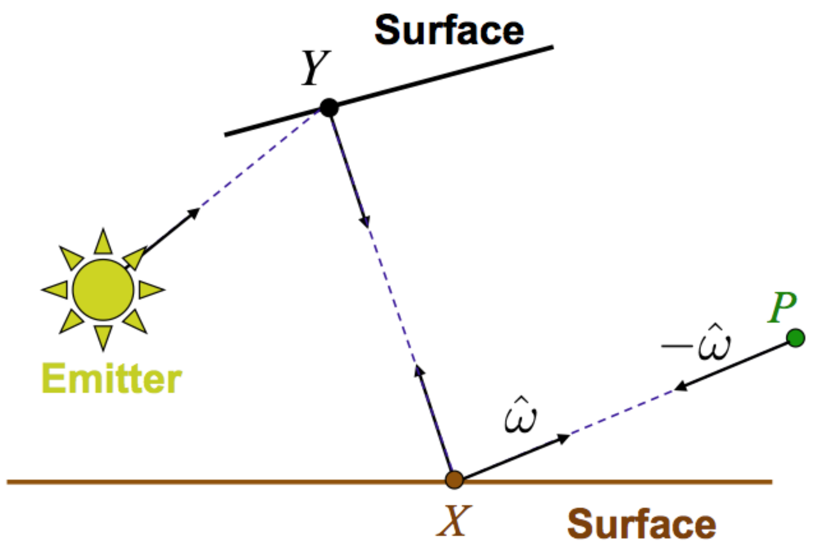
The Light Field

We can now formalize light. It propagates along rays, so we'll measure the light at a point traveling in a direction. Let the **light field** $L(P, \hat{\omega})$ denote the light at P propagating in direction $\hat{\omega}$. (The use of capital "L" is historical and doesn't follow my other naming conventions.) To produce color images we need to consider multiple frequencies of light, but for now assume that L is a scalar-valued function.

The light field was not invented for computer graphics. It has a long history in astronomy, art, and optics. The term ``light field'' was coined by Moon and Timoshenko's translation [Gershun1936Lightfield] of Gershun's earlier work. The light field is also known as the **plenoptic function** [Adelson1991Plenoptic p4] and is closely related to the **4D light field** [Levoy1996Lightfield] a.k.a. **light slab** [Levoy1996Gightfield] a.k.a. **Lumigraph** [Gortler1996Lumigraph] . The general notion of a field of light was first formally presented by Faraday [Faraday1846] [FaradayBritannica] and appears in Da Vinci's journals from the 15th century.

Light is Conserved Along a Ray

Let us set aside the issue of absorption in a medium and consider only vacuum or media like air in which absorption is negligible for small scenes. I'll call these media “empty space”. Because light propagates along rays, the amount of light must be *conserved along a ray* through empty space — there is no way for other light on different rays to get onto this one, and no way for light to leave it in empty space. That means that for two points X and P *in empty space* and with no surfaces between them, the light leaving X in the direction of P must all actually reach P. This is critical for efficient rendering: it means that we need only consider what happens at the points where a light path changes direction and need not simulate the (infinitely-many more) points along the path through space.



Scattering events at points Y and X along a light path from the emitter to P.

We can express the conservation of light along a ray formally as:

$$L(P - \epsilon\hat{\omega}, -\hat{\omega}) = L(X + \epsilon\hat{\omega}, \hat{\omega})$$

$$\text{where } \hat{\omega} = S(P - X)$$

Constant ϵ denotes a vanishingly small distance. It appears in the equation so that we can ignore what happens exactly *at* points X and P and instead consider only the light transported through the space *between* them. What happens exactly at a surface is complicated and ambiguous. For example, X is a point on a surface. If the surface is opaque to visible light, then $L(X, \hat{\omega}) = 0$ for any direction because no light can flow *through* the surface. In contrast, point $X + \epsilon\hat{\omega}$ is in the "empty" space immediately above the surface and presents none of these problems.

Althought ϵ and negated vectors make our parameterization clear and correct, they are also cumbersome notations that obscure the equality relationship. Some new notation can make our statements about light more terse and reveal the important symmetry. Let:

$$L_\text{o}(X, \hat{\omega}) = L(X + \epsilon\hat{\omega}, \hat{\omega})$$

i.e., the light **o**utgoing from point X in direction $\hat{\omega}$. Let

$$L_\text{i}(X, \hat{\omega}) = L(X - \epsilon\hat{\omega}, -\hat{\omega})$$

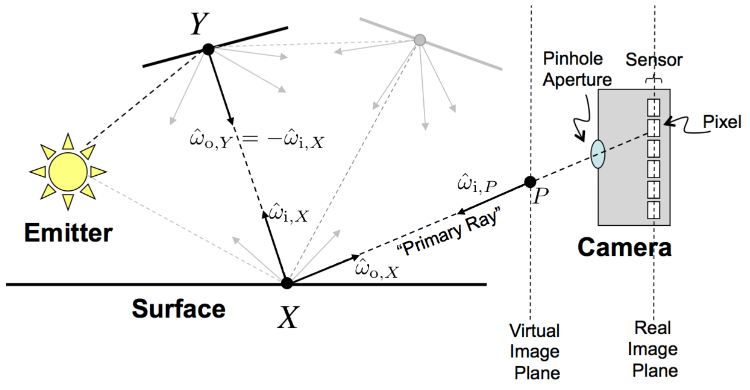
i.e., the light **i**ncoming to point X that propagates in direction $-\hat{\omega}$.

Since we'll want to consider light transported not just along one segment but also along a entire paths, we'll require a compact way of refering to distinguish several directions. For this, I place subscripts on $\hat{\omega}$ that denote the "in" and "out" direction and the point on the path. For example, let $\hat{\omega}_{\mathrm{i},X} = \hat{\omega}$ be the direction from which (i.e., the negative direction of propagation) light arrived at X along the path, and $\hat{\omega}_{\mathrm{o},X}$ be the direction in which the light left X. When only one set of directions is considered in an equation, I'll drop the point subscript (it is almost always X in that case). Now, we can express the conservation equation as:

$$L_{\mathrm{i}}(P, \hat{\omega}_{\mathrm{i},P}) = L_{\mathrm{o}}(X, \hat{\omega}_{\mathrm{o},X})$$

If P is a point on the camera, then solving for the light into the camera is indeed exactly what we want to do in order to render an image. So, the ray conservation equation is now the beginning of an equation for rendering an image.

Now consider the right side of the equation, which describes the quantity that we need to compute. The light leaving X is some fraction the light that entered X from Y and scattered there, plus any new light that was emitted at X if the surface was glowing.


Y
$\hat{\omega}_{\mathrm{o},Y} = -\hat{\omega}_{\mathrm{i},X}$
$\hat{\omega}_{\mathrm{i},X}$
$\hat{\omega}_{\mathrm{i},P}$
P
"Primary Ray"
$\hat{\omega}_{\mathrm{o},X}$
Emitter
Surface
X
Pinhole Aperture
Sensor
Pixel
Camera
Virtual Image Plane
Real Image Plane

That is,

$L_{\mathrm{i}}(X, \hat{\omega}_{\mathrm{o},X})$ is the sum of light emitted at X in direction $\hat{\omega}_{\mathrm{o},X}$ and light scattered at X in direction $\hat{\omega}_{\mathrm{o},X}$. Let L_{e} be the subset of the light that is emitted; it will end up being trivial in the implementation of a renderer.

Of course, light doesn't reach X just from Y. There are plenty of other paths that have XP as their final segment. So to consider the light that reaches P after scattering at X, we have to consider all possible points from which light might reach X initially:

$$L_\mathrm{o}(X, \hat{\omega}_{\mathrm{o},X}) = L_\mathrm{e}(X, \hat{\omega}_{\mathrm{o},X}) + \int L_\mathrm{i}(X, \hat{\omega}_{\mathrm{i},X}) \ldots \square \ldots dY$$

$$\text{where } \hat{\omega}_{\mathrm{i},X} = -\hat{\omega}_{\mathrm{o},Y}$$

In this equation, the empty box ($\ldots\square\ldots$) abstracts how much light scatters at X, and some measurement details about the integration. The domain of integration is "every point Y visible from X". We'll formalize both of these shortly and fill in the box.

To clean up the measurement details we need to attach units to what we're measuring. I've carefully and ambiguously refered to "light" so far and set aside units. In the next chapter, I'll explain what the light that we're measuring is so that we can complete the equation.

Symbols

Symbol	*Type*	*Description*	*Ref*
$L(X, \hat{\omega})$	$\mathrm{W}/(\mathrm{m}^2\mathrm{sr})$	Radiance through X in direction $\hat{\omega}$.	[L]
$L_\mathrm{i}(X, \hat{\omega}_\mathrm{i})$	$\mathrm{W}/(\mathrm{m}^2\mathrm{sr})$	Radiance transported through $X + \epsilon\hat{\omega}_\mathrm{i}$ in direction $-\hat{\omega}_\mathrm{i}$.	[Li]
$L_\mathrm{o}(X, \hat{\omega}_\mathrm{o})$	$\mathrm{W}/(\mathrm{m}^2\mathrm{sr})$	Radiance transported through $X + \epsilon\hat{\omega}_\mathrm{o}$ in direction $\hat{\omega}_\mathrm{o}$.	[Lo]
$\Phi(\mathbb{A}, \Gamma)$	W	Power (flux) propagating through points in $\mathbb{A}$ in directions in Γ.	[power]
$S(\vec{v})$	$\mathbb{S}^n$	Normalize $\vec{v} \in \mathbb{R}^n$ by projecting it onto $\mathbb{S}^n$.	[nrmlz]
X	$\mathbb{R}^3\mathrm{m}$	A point in the scene at which photons may interact with	[sctvar]

		matter.	
$\hat{n}$	$\mathbb{S}^2$	Unit normal to the surface point X.	[sctvar]
$\hat{\omega}_\mathrm{i}$	$\mathbb{S}^2$	Unit incident light direction (opposite the direction of photon propagation, pointing back at where the light came from).	[sctvar]
$\hat{\omega}_\mathrm{o}$	$\mathbb{S}^2$	Unit exiting light direction (in the direction of photon propagation, pointing forward to where the light is going).	[sctvar]
P	$\mathbb{R}^3\mathrm{m}$	A point on the image plane / the origin of a primary ray.	

References

[Adelson1991Plenoptic]

The Plenoptic Function and the Elements of Early Vision
Edward H. Adelson and James R. Bergen
in *Computational Models of Visual Processing*, p. 3-20, MIT Press, 1991.

[Gershun1936Lightfield]

The Light Field
A. Gershun
Journal of Mathematics and Physics 18, p. 51-151, translated by P. Moon and G. Timoshenko (originally published in Moscow, 1936), MIT, 1939.

[Gortler1996Lumigraph]

The Lumigraph
Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen
in *Proceedings of the 23rd annual conference on Computer*

graphics and interactive techniques, p. 43-54, ACM Press, 1996.
http://doi.acm.org/10.1145/237170.237200

[Levoy1996Lightfield]

Light Field Rendering
Marc Levoy and Pat Hanrahan
in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, p. 31-42, ACM Press, 1996.
http://doi.acm.org/10.1145/237170.237199