Tutorial on Light Field Rendering

Mel Slater

Presented at VRST’2000
Seoul, Korea, 22nd October, 2000.

(C) 2000 Mel Slater
These notes are available for the purposes of the VRST’2000 Tutorial, Seoul, Korea, October, 2000.

The notes are partially based on a forthcoming book to be published by Addison-Wesley Longmans in the summer of 2001:

M. Slater, A. Steed, Y. Chrysanthou
Computer Graphics and Virtual Environments: From Realism to Real-Time

No part of these notes may be reproduced without written permission of the author.
| CHAPTER 1  | Preface  | 5 |
|           | Scope    | 5 |

| CHAPTER 2  | Mathematics for Virtual Environments  | 7 |
| Introduction | 7 |
| Geometry for Virtual Environments | 7 |
| Flatness Preserving Transformations | 20 |
| Quaternions | 32 |
| Summary | 32 |

| CHAPTER 3  | Lighting - the Radiance Equation  | 35 |
| Lighting The Fundamental Problem for Computer Graphics | 35 |
| Light | 36 |
| Simplifying Assumptions | 39 |
| Radiance | 41 |
| Reflectance | 44 |
| The Radiance Equation | 46 |
| Solutions of the Radiance Equation | 48 |
| Visibility | 53 |
| Summary | 53 |

| CHAPTER 4  | An Introduction to Light Fields  | 55 |
| Introduction | 55 |
| Rendering: Interpolation | 58 |
| Representing Light Fields | 63 |
| Practical Issues | 64 |
CHAPTER 5     Virtual Light Fields     67

Introduction 67
The Virtual Light Field (VLF) 69
Implementation 76
Preliminary Error Analysis 81
Results 84
Discussion 87

CHAPTER 6     References     89
CHAPTER 1

Preface

Scope

These notes introduce the concept and some aspects of the implementation of light fields for computer graphics rendering. All computer graphics rendering may be thought of as solutions to a integral equation which expresses the radiance at any point on a surface in a scene in any direction. Light fields may be thought of as a kind of brute force solution to this equation, made possible today by the vast increases in memory and processing power that have occurred over the past few years. We therefore first introduce this radiance equation, and provide a framework for discussion of the various approaches to rendering in this context, and prior to this we present some of the background mathematics for completeness. Then we introduce the traditional approach to light field representation and rendering. Finally we argue that the light field approach provides a different paradigm for computer graphics rendering of virtual scenes, and which can include global illumination. We call this a virtual light field, and give an illustration as to how the goal might be achieved.
Preface
Introduction

In this chapter we introduce some of the essential mathematics used to describe and reason about virtual environments. Fundamentally this mathematics is used to describe objects, to manipulate objects, and also for the representation of illumination.

Geometry for Virtual Environments

Dimension

The geometry of objects in a VE are specified using the rich and powerful language supplied by mathematics. In this section we specify some of the main ideas that are required to describe and reason about the geometry of objects. This discussion is certainly not meant to be mathematically rigorous, but rather relies on intuition, and is sufficient for our purposes.
First, the concept of *dimension* of a space. This refers to the amount of freedom of movement that objects within the space have, while remaining totally within the confines of that space. If it were possible to think of ‘objects’ contained within an environment specified with a 0 dimension, they would be unable to move at all. Moreover, no information is required in order to specify the position of an object within 0 dimensional space: it is just There. A 0 dimensional space is just a single point (and from the point of view of objects within such a space, of course there is nothing outside of it). Recall that a mathematical point has no ‘thickness’, it has no area, it is ‘infinitesimally small’, it is as close as it is possible to get to nothing while still being something.

Objects within a space of dimension 1 can move ‘left’ or ‘right’: that is they can move in a direction, and then perhaps move in the opposite direction retracing the previous path, and that is all. The words ‘left’ and ‘right’ are in quotes, since the labelling of the direction is quite arbitrary. Actually these words can be misleading, since they imply a notion of horizontalness. However, in a 1D space it is impossible to think about the notion of ‘being horizontal’ since that depends on a frame of reference that is more than one dimensional. A 1D space can be thought of as an infinitely long and continuous line (for beings within such a space there is nothing except this line!). In order to describe such a space we can construct a coordinate frame. Choose an arbitrary point and label it as 0, the origin. One direction from the origin is called *positive*, and the other is called *negative*. The entire part of the line in the positive direction is called the positive half-space, and similarly there is the negative half-space. The position of any point in the space is then specified by a single number representing the distance from the origin according to some agreed metric, and with a + or - sign according to whether it is in the positive or negative half-space on the line.

There can be infinitely many ‘objects’ represented in 1D. Objects might be 0-dimensional (i.e., be single points) or be 1-dimensional, for example, intervals of the form: the object represented by all points greater than or equal to 1.0 but less than or equal to 2.0 also denoted as \{x|1 \leq x \leq 2\} or more simply as [1.0,2.0]. In general such objects can be represented as almost any set of points in the 1D space. (Why ‘almost’?)

Such a space could be used, for example, to provide a very simple representation of traffic flow on a highway. Here the highway is itself the ‘line’ and the vehicles are represented as intervals of the form \([a,b]\). As a vehicle moves along the highway, so the values represented by \([a,b]\) would change (although the length \(b-a\) would remain constant). Even in this simple space there are some computational problems that are certainly non-trivial. For example, given a collection of objects
quickly find the set of objects with overlapping intervals. Or suppose that in an interactive setting a human operator wishes to select an object with a (1D) cursor: the program would have to solve the following problem: given any point \( x \) on the line find as quickly as possible the (set of) objects being pointed at (if any), that is the set of intervals \([a_i, b_i]\) containing \( x \).

### Coordinate Systems, Points and Vectors

Objects within a space of two dimensions can move ‘left’ and ‘right’, or ‘up’ or ‘down’ (again these words are chosen as arbitrary labels). Moreover, any point that an object can reach by moving through sequences of moves from this set can also be reached directly by a straight line. In order to describe such a space we require two infinite lines that are orthogonal (perpendicular) to one another, and intersecting at a point labelled as the origin. This intersection point is the origin of a 2D coordinate system. By convention, one axis is ‘horizontal’, and is called the X axis, and the other is ‘vertical’ and called the Y axis (why are ‘horizontal’ and ‘vertical’ in quotes?). The origin is the point \((0,0)\). Any point in the 2D space can referenced as a coordinate pair \((x,y)\), where \( x \) represents the distance along the X axis from the origin, and \( y \) along the Y axis. Again, in such a coordinate system (usually called Cartesian) a metric is required to specify distance. Let \( p_1 = (x_1, y_1) \) and \( p_2 = (x_2, y_2) \) be two distinct points, then the distance \( p_1 \) to \( p_2 \) is given by:

\[
|p_1 - p_2| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}
\]  

\text{(EQ 1)}

A similar description can be given for a 3D space. We use a Cartesian coordinate system, so that for a 3D space, we choose some convenient (but arbitrary) point as origin, and three orthogonal directions labelled as X, Y and Z axes, the \textit{principal axes}. All points are specified as offsets from the origin, along directions parallel to the principal axes. Note that the origin is \((0,0,0)\), and that the coordinate system is continuous. In general a point is represented by \((x,y,z)\). In 3D we have a choice about the configuration of positive and negative conventions. Figure 1 on page 10 shows so-called left- and right-hand 3D coordinate frames. In each case identify your first finger as the Y axis, and thumb as the X axis. The Z axis is specified as your second finger.
Representing positions by points in this way is arbitrary up to the choice of origin and principal axes. A vector specifies a direction, and the magnitude of the direction, and is the difference of two points. In 3D, we will represent a vector also by \((x,y,z)\), where it will be clear from context whether a point or a vector is required. So suppose \((4,5,6)\) and \((3,4,5)\) are points, then \((1,1,1) = (4,5,6) - (3,4,5)\) is a vector, representing the direction \((1,1,1)\). A vector \((x,y,z)\) may be visualised as a line with one end tied to the origin, and the other at the point \((x,y,z)\) with an arrow at this point. This is shown in Figure 2 on page 11.

A vector, then, is a difference of two points:

\[
\text{vector} = \text{point}_2 - \text{point}_1 \tag{EQ 2}
\]

from which it follows that a point may be specified as the addition of a point and a vector:

\[
\text{point}_2 = \text{point}_1 + \text{vector} \tag{EQ 3}
\]
Vectors may be summed, using the familiar parallelogram rule (EQ 4). An example is shown in Figure 3 on page 11.

\[ v_1 = (x_1, y_1, z_1) \]

\[ v_2 = (x_2, y_2, z_2) \]

\[ v_1 + v_2 = (x_1 + y_1, x_2 + y_2, z_1 + z_2) \]  

**FIGURE 2. A Vector in 3D**

**FIGURE 3. Addition of Vectors**
Notice that although it is meaningful to add vectors together, this is not the case for points (why not?).

Suppose \( \mathbf{v} = (x, y, z) \) is any vector, then this can be scaled (for example, stretched) by multiplying throughout by a constant real number, say \( \lambda \). Then,

\[
\lambda \mathbf{v} = (\lambda x, \lambda y, \lambda z) \tag{EQ 5}
\]

is a new vector (and the new vector has the same direction as the old one if \( \lambda > 0 \)). The new length can easily be found from the result:

\[
|\lambda \mathbf{v}| = |\lambda||\mathbf{v}| \quad \text{where} \quad |\lambda| = \lambda \text{ if } \lambda \geq 0 \text{ and } -\lambda \text{ if } \lambda < 0 \tag{EQ 6}
\]

So note then, that given any two vectors \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \) and any two real numbers \( \lambda_1 \) and \( \lambda_2 \), that \( \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 \) is also a vector.

There are three noteworthy vectors that are said to form a basis for the entire space:

\[
\begin{align*}
\mathbf{e}_1 & = (1, 0, 0) \\
\mathbf{e}_2 & = (0, 1, 0) \\
\mathbf{e}_3 & = (0, 0, 1)
\end{align*} \tag{EQ 7}
\]

This means that any \((x, y, z)\) can be expressed as a linear combination of these three, since:

\[
(x, y, z) = xe_1 + ye_2 + ze_3 \tag{EQ 8}
\]

The norm of a vector gives its length (in fact this provides the metric of the 3D space). Suppose \( \mathbf{v} = (x, y, z) \) is a vector, then its norm is denoted \(|\mathbf{v}|\), and is defined as:

\[
|\mathbf{v}| = \sqrt{x^2 + y^2 + z^2} \tag{EQ 9}
\]

This is the distance from the origin to the point \((x, y, z)\). So if \( p_1 \) and \( p_2 \) are two points, then \( p_1 - p_2 \) is a vector, and the distance between the points is the norm \(|p_2 - p_1|\).
Given any vector \( \mathbf{v} = (x, y, z) \), suppose we are only interested in its direction and not at all in its length. Then for many purposes it is convenient to use another vector that has the same direction, but length 1. We can find such a vector by normalising the original one. This is achieved as follows:

\[
\text{norm}(\mathbf{v}) = \frac{\mathbf{v}}{|\mathbf{v}|} \quad \text{(EQ 10)}
\]

It is very easy to show that \(|\text{norm}(\mathbf{v})| = 1\). (Show this).

Suppose we again have two vectors \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \). The dot product of these two vectors, denoted by \( \mathbf{v}_1 \cdot \mathbf{v}_2 \) gives us important information about the angular relationship between them. This is defined by:

\[
\mathbf{v}_1 \cdot \mathbf{v}_2 = x_1x_2 + y_1y_2 + z_1z_2 \quad \text{(EQ 11)}
\]

Normalise each of the vectors \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \) and then take the dot product of the normalised vectors. This gives us an important result:

\[
\cos \theta = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{|\mathbf{v}_1| \times |\mathbf{v}_2|} \quad \text{(EQ 12)}
\]

\[
\text{(EQ 13)}
\]

where \( \theta \) is the angle between the two vectors.

Hence the dot product is proportional to the cosine of the angle between two vectors. Recall that \( \cos 0 = 1 \) and \( \cos \frac{\pi}{2} = 0 \).

The implication of the first result is that the dot product of a unit length vector with itself is 1. The implication of the second result is that vectors that are perpendicular to one another (we call such vectors orthogonal) have a zero dot product. This is a result of great use for later chapters.
Finally, in this section we consider another binary operation on vectors that is geometrically significant - the cross product of two vectors, denoted $v_1 \times v_2$. This is results in a new vector as follows:

$$v_1 \times v_2 = (y_1z_2 - y_2z_1, x_2z_1 - x_1z_2, x_1y_2 - x_2y_1) \tag{EQ 14}$$

The properties of the cross product are as follows: First,

$$|v_1 \times v_2| = |v_1||v_2|\sin\theta \tag{EQ 15}$$

where $\theta$ is the angle between the two vectors. Second, and of significant use in computer graphics, $v_1 \times v_2$ is orthogonal to both $v_1$ and $v_2$. Another way of saying this (though jumping ahead a little) is that $v_1$ and $v_2$ form a plane. Their cross product is normal to that plane (that is, it is perpendicular to it). Now an important question concerns the direction of the new vector - for example, if $v_1$ and $v_2$ were inscribed on the page in front of you (as in Figure 4 on page 14), then would the new vector point out towards you or behind the plane away from you? The answer is that $v_1$, $v_2$, and $v_1 \times v_2$ form a right-handed system: if you identify $v_1$ with your thumb, and $v_2$ with your first finger, then $v_1 \times v_2$ will correspond to your second finger. In Figure 4 on page 14, the cross product $v_1 \times v_2$ points out of the page, and $v_2 \times v_1$ points behind the page. In general it is easy to see that $v_1 \times v_2 = -v_2 \times v_1$ (such a property is called anti-symmetric. The dot product is, of course, symmetric).

**FIGURE 4. Forming the Cross Product**
A vector represents a magnitude (the norm or length of the vector) and a direction. In computer graphics, we are mostly interested in the directional properties of vectors. It is interesting and important to note that there is redundancy in representing a vector with three quantities $x$, $y$, and $z$. It should be clear from (EQ 10) that in fact only two quantities are needed: the space of all possible directions in 3D is in fact a 2D space! The reason is that to specify a direction we only need to consider normalised vectors (since the length doesn’t matter). For any normalised vector $(x, y, z)$, it is obviously the case that:

$$x^2 + y^2 + z^2 = 1 \quad \text{(EQ 16)}$$

Hence, if we knew the values of $x$ and $y$ (say) then we can compute the value of $z$ (except for its sign) using (EQ 16). More intuitively, any direction can be thought of as a line from the origin to a point on a unit sphere centred at the origin (the sphere represented by (EQ 16)). So all possible directions correspond to all possible points on the unit sphere. But the surface of a sphere is a two-dimensional entity - hence the space of all possible directions is two-dimensional.

**Solid Angles**

This motivates another representation of a direction vector: using angles to specify a point on a unit sphere. Figure 5 on page 16 shows a unit sphere and vector $v$ intersecting the sphere at $P$. A perpendicular is dropped from $P$ to meet the XY plane at $Q$. The direction corresponding to $v$ is completely specified by the two angles $\phi = X0Q$ and $\theta = Z0P$. (See “Spherical Coordinate Representation” on page 30.) The important result to note here is that the set of all direction vectors can be represented by the 2D space:

$$\Omega = \left\{(\theta, \phi) | 0 \leq \phi < 2\pi, \frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\right\} \quad \text{(EQ 17)}$$
In Figure 5 on page 16 we have deliberately shown a small area on the sphere around the point $P$. Suppose this area is $A$. The solid angle corresponding to $A$ is defined by:

$$\Gamma = \frac{A}{r^2}$$  \hfill (EQ 18)

where $r$ is the radius of a sphere. The unit of measurement is called a steradian. An entire sphere comprises $4\pi r^2 / r^2 = 4\pi$ steradians. The solid angle may be thought of by analogy with an ordinary angle, which is determined by the length of the circumference of a circle that it circumvents, divided by the radius of the circle.

A differential solid angle is the solid angle subtended by a ‘differential area’ - i.e., a patch ‘tending to area zero’ on the surface of a sphere. This is usually denoted by $d\omega$. We can tie things together here by showing that an explicit formula for $d\omega$ can be found in terms of the angles $\theta$ and $\phi$ specifying a direction, i.e., a point on a sphere.
Figure 6 on page 17 shows a point $p = (x, y, z)$ on a sphere of radius $r$, and a differential solid angle around $p$. We suppose that $p$ has spherical coordinates $(\theta, \phi)$. The point $p$ lies on a circle at height $z$ on the sphere with centre $C$. The radius of this circle is $r \sin \theta$. Since the angle subtended by the small horizontal sweep along the circumference of this circle must be $d\phi$, we have:

$$d\phi = \frac{dh}{r \sin \theta} \quad (EQ \ 19)$$

$$\therefore dh = r \phi \sin \theta$$

Similarly, if we consider the vertical sweep through angle $d\theta$ on the great circle containing $p$:

$$d\theta = \frac{dv}{r} \quad (EQ \ 20)$$

$$\therefore dv = r d\theta$$

Since a solid angle is area divided by the square of the radius, it follows from (EQ 19) and (EQ 20) that the required differential solid angle is:
The final idea we wish to present here is that of **projected area**. In Figure 7 on page 18(a) we show an area $A$ and a plane $P$. Consider the projection of $A$ onto $P$ - that is the image formed on $P$ when each point of $A$ follows a ray in direction $v$ intersecting with $P$. If $A$ and $P$ are parallel, and the direction of projection is perpendicular to $A$ (shown as $n$ in the figure, usually called the *normal to $A$*) then the projected area is the same as the original area. If the direction of projection is not perpendicular (such as $v$ in the figure), or equivalently if the area and plane are not parallel (as in Figure 7 on page 18(b)) then the projected area is smaller than the original area. What matters only is the cosine of the angle between the direction of projection and the normal (perpendicular) to the area. The smaller the angle the closer the cosine is to 1, the projected area tending to be the same as the original area. The larger the angle the smaller the projected area, until in the limit when the angle is $\pi/2$ the projected area vanishes. Note that the cosine is the dot product of the normal and the direction of projection: $\cos \theta = n \cdot v$ assuming that the vectors are normalised.

**FIGURE 7. Projected Area**

![Projected Area Diagram](image)

The definition of projected area is:
To avoid confusion, by \( A \) we do not mean the numerical area - rather we are using the word ‘area’ here to mean ‘region’. Hence \( A \) is the original region and \( A' = A \cos \theta \) denotes the projected region. If the actual area of \( A \) is written \( |A| \), then \( |A'| = |A| \cos \theta \). However, we will usually not use the \( |...| \) symbol for notational convenience. It should be clear from context when a numerical area is meant rather than the region which has that area.

**FIGURE 8. Solid angle of a differential area**

Projected areas can also be used to find solid angles of differential areas (areas ‘tending to zero’). Figure 8 on page 19 shows an arbitrary but small area denoted \( dA \) and we wish to find the solid angle from point \( O \), distance \( r \) from \( dA \). Construct a sphere of radius \( r \) around \( O \). The projected area towards \( O \) is \( dA \cos \theta \) - i.e., this is its projected area onto the sphere. Then the differential solid angle subtended by the area is:

\[
d\omega = \frac{dA \cos \theta}{r^2}
\]  

**(EQ 23)**
Flatness Preserving Transformations

Barycentric Combinations: Flatness

In this Section we study transformations that preserve ‘flatness’. This is of great importance in 3D computer graphics, where the fundamental primitive is the planar polygon. When we transform an object that is constructed from polygons, we wish to be sure that the transformations leave the flatness (or planarity) of the polygons invariant. Also such transformations are very useful for computational reasons - since we can simply transform the vertices of a polygon, and be sure that the transformed vertices correctly describe the new polygon. If this were not the case we would have to transform ‘every point’ belonging to the original polygon (vertices, boundary, and inside) in order to get the shape of the transformed object. First we define our notion of ‘flatness’, and then we specify the property of such a flatness preserving transformation, usually called an affine transformation.

Line Segments

A fundamental form in computer graphics is a straight line joining two points. This can be best represented in parametric form: given two points \( p_1 \) and \( p_2 \) in 3D space (actually in a space of any dimension), then the straight line that passes through them can be represented as all points \( p(t) \) satisfying:

\[
p(t) = (1-t)p_1 + tp_2
\]  

for any real number \( t \).

If we want the line segment that connects \( p_1 \) to \( p_2 \) only then \( t \) is restricted to the range \([0,1]\). Notice that this equation can be rewritten as:

\[
p(t) = p_1 + t(p_2 - p_1)
\]  

This form shows that the line segment can be considered as an origin point \( p_1 \) plus a vector \( p_2 - p_1 \) scaled by \( t \in [0, 1] \).
FIGURE 9. Parametric Representation of a Line Segment

Barycentric Combinations

The parametric line representation is a special case of what is called a barycentric combination, which is a weighted sum of points, where the weights sum to 1. Let \( p_1, p_2, \ldots, p_n \) be a sequence of points, and \( \alpha_1, \alpha_2, \ldots, \alpha_n \) be any sequence of real numbers summing to 1, then

\[
p = \sum_{i=1}^{n} \alpha_i p_i
\]

with

\[
\sum_{i=1}^{n} \alpha_i = 1
\]

defines a barycentric combination. Obviously, the special case \( n = 2 \) gives the straight line as in (EQ 25).

Assume for the moment that the points \( p_1, p_2, \ldots, p_n \) all lie on the same plane \((P)\).

Suppose we let the \( \alpha \) values vary arbitrarily subject to the constraint that they always sum to 1. Each set of values \( (\alpha_1, \alpha_2, \ldots, \alpha_n) \) corresponds to a point \( p \) in space. If we consider the collection of all such points (of course there are infinitely
many) then it can be shown that the resulting surface will be the plane $P$ that passes through each of the points $p_1, p_2, \ldots, p_n$. Intuitively, this is quite easy to see. If we select any two of the points, say $p_a$ and $p_b$, we can set all except for $\alpha_a$ and $\alpha_b$. Then the constraint requires that $\alpha_a + \alpha_b = 1$, which therefore means that all points on the straight line joining $p_a$ and $p_b$ are on $P$. Again now take any arbitrary point on this straight line, and call this point $p_{ab}$. Now choose another one of the original points, say $p_c$. Then repeating the same argument implies that all points on the line joining $p_{ab}$ to $p_c$ must also be on $P$. It follows from this that all points on the plane formed by $p_a$, $p_b$, and $p_c$ must belong to $P$. Repeating this argument with each point from the original set shows that all such points must lie on the same plane, which must be $P$.

If we further restrict the weights to be non-negative, $\alpha_i \geq 0$, $i = 0, 1, \ldots, n$ then it is also easy to see that $P$ becomes the smallest convex polygon (including the interior) that encloses all of the vertices $p_1, p_2, \ldots, p_n$. Should these vertices in fact describe a convex polygon, then $P$ itself will be this polygon.

More generally, suppose that the $p_i$ do not all lie on the same plane (which will usually be the case for $n = 4$). Then if we restrict the $\alpha_i$ to be non-negative, the shape $P$ that is mapped out as $p$ varies with varying $\alpha_i$ is called the convex hull of the points. The convex hull is the smallest convex polyhedron that encloses all of the points $p_i$. Mathematically, the convex hull is such that the straight line segment joining any two points on its boundary will always be inside (or on the boundary) of the convex hull. In particular note that the centroid of the points (or ‘average’) point will be inside the convex hull, where the average is:

$$\bar{p} = \frac{1}{n} \sum_{i=1}^{n} p_i$$  \hspace{1cm} (EQ 27)
Affine Transformations

An affine transformation is one that preserves barycentric combinations - and therefore preserves flatness. This property can be specified exactly as follows. Let \( p \) be a point in 3D space that is a barycentric combination as in (EQ 26).

Suppose \( f \) is a mapping from 3D space to 3D space, so that \( f(p) \) is a point in this space. Then \( f \) is affine if and only if:

\[
 f(p) = \sum_{i=1}^{n} \alpha_i f(p_i) \tag{EQ 28}
\]

In other words, we get the same transformed point whether we first find the barycentric combination \( p \) and then compute \( f(p) \) or alternatively whether we first find all the \( f(p_i) \) and then find the barycentric combination of all the transformed points. Computationally it is always more efficient to do the latter when transforming the entire object formed from such barycentric combinations in the discussion above. In view of our discussion of flatness, it should be clear that an affine transformation of a planar polygon can be found by tranforming each of its vertices, and then constructing the new polygon from these.

Matrix Representation of Affine Transformations

The formulation (EQ 28) gives the property that defines an affine transformation, but does not give us a constructive formulation, i.e., what form does such a transformation take? We will derive the form of such a transformation in this Section.

Since \( p \) is a point in 3D space, it is expressible as a coordinate \( (x_1, x_2, \ldots, x_n) \).

Moreover, let

\[
 e_1 = (1, 0, 0) \\
 e_2 = (0, 1, 0) \\
 e_3 = (0, 0, 1)
\]
be the principle unit vectors. Also define \( e_4 = (0, 0, 0) \). Then,

\[
p = x_1 e_1 + x_2 e_2 + x_3 e_3 + x_4 e_4
\]

\[
= \sum_{i=1}^{4} x_i e_i
\]

\[\text{(EQ 29)}\]

where \( x_4 \) is anything. Since \( x_4 \) can be whatever we like without affecting (EQ 29), we shall choose

\[
x_4 = 1 - x_1 - x_2 - x_3
\]

and so

\[
\sum_{i=1}^{4} x_i = 1
\]

\[\text{(EQ 30)}\]

Consider now the principle unit vectors \( e_i \). Let \( f \) be an affine transformation as before, then \( f(e_i) \) is in 3D space, and therefore there must be coordinates \( \lambda_{ij} \) with

\[
f(e_i) = \{\lambda_{i1}, \lambda_{i2}, \lambda_{i3}\}
\]

\[
= \sum_{j=1}^{3} \lambda_{ij} e_j
\]

\[\text{(EQ 31)}\]

Now from (EQ 29) and (EQ 30)

\[
f(p) = \sum_{i=1}^{4} x_i f(e_i)
\]

\[\text{(EQ 32)}\]

Substituting (EQ 31) into (EQ 32):

\[
f(p) = \sum_{i=1}^{4} x_i \sum_{j=1}^{3} \lambda_{ij} e_j
\]

\[\text{(EQ 33)}\]
Re-arrange the order of summation in (EQ 33) to give:

\[ f(p) = \sum_{j=1}^{3} \varepsilon_j \sum_{i=1}^{4} x_i \lambda_{ij} \]  

(EQ 34)

Recalling that \( x_4 = 1 - x_1 - x_2 - x_3 \), we can write \( \mu_{ij} = \lambda_{ij} - \lambda_{4j} \) to obtain:

\[ f(p) = \sum_{j=1}^{3} \varepsilon_j \left( \sum_{i=1}^{3} x_i \mu_{ij} + \lambda_{4j} \right) \]  

(EQ 35)

Expanding out using the meaning of the \( \varepsilon_j \), we get \( f(p) \) in coordinate form:

\[ f(p) = \begin{pmatrix} \sum_{i=1}^{3} x_i \mu_{i1} + \lambda_{41} \\ \sum_{i=1}^{3} x_i \mu_{i2} + \lambda_{42} \\ \sum_{i=1}^{3} x_i \mu_{i3} + \lambda_{43} \end{pmatrix} \]  

(EQ 36)

Putting this in perhaps more familiar notation, we now suppose that \( p = (x, y, z) \) and \( f(p) = (x', y', z') \). Then if \( f \) is an affine transformation, each new coordinate may be represented as a simple affine combination of each of the original coordinates, in other words:

\[ x' = a_{11}x + a_{21}y + a_{31}z + a_{41} \]
\[ y' = a_{12}x + a_{22}y + a_{32}z + a_{42} \]
\[ z' = a_{13}x + a_{23}y + a_{33}z + a_{43} \]  

(EQ 37)

for some constants \( a_{ij} \).

**Matrix Representation**

Expression (EQ 37) gives the general form of an affine transformation of a point \((x, y, z)\) to the point \((x', y', z')\). It is convenient to express (EQ 37) in matrix form. As before suppose \( p = (x, y, z) \) and \( f(p) = (x', y', z') \) where \( f \) is an affine, flatness preserving transformation. There are three possibilities for representing this in matrix form. The first way is:
Alternatively, we could get rid of the translation vector, and use:

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33} \\
  a_{41} & a_{42} & a_{43}
\end{bmatrix}
\]

\[
(x', y', z') = (x, y, z, 1)
\]

(EQ 39)

We wish to make use of matrix multiplication in order to compute a sequence of such transformations. This is impossible using the forms of (EQ 38) and (EQ 39), since the successive matrices would not be suitable for multiplication. Instead, we introduce the homogeneous representation of a 3D point as \((x, y, z) = (x, y, z, 1)\), to write:

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & 0 \\
  a_{21} & a_{22} & a_{23} & 0 \\
  a_{31} & a_{32} & a_{33} & 0 \\
  a_{41} & a_{42} & a_{43} & 1
\end{bmatrix}
\]

\[
(x', y', z', 1) = (x, y, z, 1)
\]

(EQ 40)

as the general matrix form of an affine transformation. Now all representative matrices are 4×4, and can therefore be multiplied together.

**Standard Transformations**

In this Section we study the standard transformations, for translation, scaling and rotation about the principal axes.

**Translation Matrix.** To translate by a vector \((a, b, c)\) we use:
Flatness Preserving Transformations

$$T(a, b, c) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c & 1 \end{bmatrix}$$  \hspace{1cm} (EQ 41)$$

since \(T(a, b, c) = (x + a, y + b, z + c)\).

Scaling Matrix. To scale relative to the origin by \(a\), \(b\) and \(c\) applied to \(x\), \(y\) and \(z\) respectively:

$$S(a, b, c) = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \\ 0 & 0 & 1 \end{bmatrix}$$  \hspace{1cm} (EQ 42)$$

since \(S(a, b, c) = (ax, by, cz)\).

Rotation Matrices. Rotation can only be relative to an axis - rotation with respect to a point is undefined since there are an infinite number of possible paths corresponding to a rotation by angle \(\theta\) about a point. With respect to a single axis there are only two possible paths, depending on the direction taken. We will define the positive direction as anti-clockwise when looking down the axis of rotation towards the origin (recalling that we are assuming a right-handed coordinate system, Chapter 1). Here we consider rotation about the principal X, Y and Z axes.

Rotation about the Z axis.

When we rotate a point \((x, y, z)\) about the Z axis, the z-coordinate remains unchanged, since the path of the rotation is with respect to a circle at constant distance z, along the Z axis, from the origin. Therefore, we can ignore the z-coordinate, and consider only what happens to \(x\) and \(y\). Here we rotate the point \((x, y, z)\) around the Z axis, by angle \(\theta\) in a positive (anti-clockwise) direction. The transformation is \(R_z(x, y, z) = (x', y', z)\).

Mathematics for Virtual Environments 27

Great Events of the Twentieth Century 27
The following result is then obvious:

\[
\begin{align*}
    x' &= x \cos \theta - y \sin \theta \\
    y' &= x \sin \theta + y \cos \theta \\
    z' &= z
\end{align*}
\]

(EQ 43)

This can be expressed in matrix form:

\[
R_\zeta(\theta) = \begin{bmatrix}
    \cos \theta & \sin \theta & 0 & 0 \\
    -\sin \theta & \cos \theta & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]

(EQ 44)

Rotation about the Y axis.

An argument by symmetry (replacing x by z and y by x) shows that the Y axis rotation is:
Flatness Preserving Transformations

\[ R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]  
(EQ 45)

Rotation about the X axis.

A similar argument results in:

\[ R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]  
(EQ 46)

Inverse Transformations. Each of the transformations has an inverse:

\[ T^{-1}(a, b, c) = T(-a, -b, -c) \]

\[ S^{-1}(a, b, c) = S\left(\frac{1}{a'}, \frac{1}{b'}, \frac{1}{c'}\right) \]  
(EQ 47)

\[ R_s^{-1}(\theta) = R_s(-\theta) \text{ for all } s = x, y, z \]

Composition of Transformations. Suppose we start with point \( p = (x,y,z) \equiv (x,y,1) \). If we apply a transformation \( M_0 \) to this point, we obtain \( p_1 = pM_0 \). If we apply a transformation \( M_1 \) to the new point we obtain \( p_2 = p_1M_1 = pM_0M_1 \). If we continue in this way applying transformations \( M_3, M_4, \ldots \) it is easy to see that in general \( p_i = pM_0M_1\ldots M_i \). Hence a sequence of transformations applied to a point composes to a multiplication of all the matrices involved, in the same order as the transformations. From the point of view of efficiency, especially as the sequence of transformations is likely to be applied to many points, we would find \( M = M_0M_1\ldots M_i \) and then apply the single matrix \( M \) to all of the points.
It should be clear that such transformations, like matrix multiplications are non-commutative. That is, \( M_i M_j \neq M_j M_i \) in general. The order matters: scaling an object and then translating it does not result in the same new object as first translating it and then scaling.

In order to scale an object with respect to a point \( q = (x, y, z) \neq (0, 0, 0) \) we would first apply a translation so that \((x, y, z)\) is shifted to the origin, then apply the scaling, and finally translate back. Hence scaling with respect to an arbitrary point \((x, y, z)\) would be obtained by:

\[
T(-x, -y, -z)S(a, b, c)T(x, y, z)
\]

Rotation about an arbitrary axis is more complicated and is discussed below.

**Spherical Coordinate Representation.** The spherical coordinate representation of a point in 3D space is often useful, and will be specifically used in the next section. In Figure 11 on page 31, \( P \) is some point \((x, y, z)\) and the distance from the origin \( O \) to \( P \) is \( r = \sqrt{x^2 + y^2 + z^2} \). \( Q \) is the perpendicular projection of \( P \) to the XY plane. Angle \( \theta \) is the angle between the X axis and \( OQ \), and angle \( \phi \) is the angle between the Z axis and \( OP \).

By construction, the angle \( \Delta OPQ = \phi \), and so \( OQ = r \sin \phi \). From this it follows that:

\[
\begin{align*}
  x &= r \sin \phi \cos \theta \\
  y &= r \sin \phi \sin \theta \\
  z &= r \cos \phi
\end{align*}
\]
Therefore, from the spherical coordinates \((r, \theta, \phi)\), it is easy using these formulae to find the Cartesian coordinates \((x, y, z)\). The inverse relationships can also be found from:

\[
\phi = \acos \frac{z}{r} \\
\theta = \atan \frac{y}{x}
\]  

(EQ 50)

**Rotation about an Arbitrary Axis.** The rotation matrices \(R_x\), \(R_y\) and \(R_z\) allow for rotation about the principal axes. These rotations in themselves are not sufficient - for example, unless an object happens to be centred on one of the principal axes, we would not be able to spin an object around an axis that intersects it. This kind of rotation is important - often in order to get a good understanding of the shape of an object we need to spin it around an axis that passes through it, or is at least near to it.

In this section we construct a composite transformation matrix \((R)\) for rotation by angle \(\alpha\) around an axis specified by two points \(p_1 = (x_1, y_1, z_1)\) and \(p_2 = (x_2, y_2, z_2)\). \(R\) can be constructed as follows:

1. Translate so that \(p_1\) is at the origin: \(T(-p_1)\)
2. Let \((x,y,z) = p_2 - p_1\) is the other end of the line segment, and write this in spherical form as \((r,\theta,\phi)\) as in the previous section. In fact the appropriate diagram for this situation is now as in the previous section.

3. Apply the rotation \(-\theta\) about the Z axis to bring Q on the ZX plane: \(R_z(-\theta)\)

4. Apply the rotation \(-\phi\) about the Y axis, so that OP is now coincident with the Z axis: \(R_y(-\phi)\)

Combining these we define the matrix \(M = T(-p_1)R_z(-\theta)R_y(-\phi)\) that transforms the axis of rotation to be the Z axis. Now we can rotate by \(R_z(\alpha)\), and then apply the inverse transformation \(M^{-1}\). Hence the complete transformation is:

\[
R = MR_y(\alpha)M^{-1}
\]

(EQ 51)

where:

\[
M = T(-p_1)R_z(-\theta)R_y(-\phi)
\]

\[
M^{-1} = R_y(\phi)R_z(\theta)T(p_1)
\]

(EQ 52)

**Quaternions**

**Summary**

We have spent some time reminding readers of the very basics of points and vectors since these ideas and operations are fundamental to computer graphics. Graphics primitives are typically planar. For the vast majority of this book, reflecting the vast amount of computer graphics practice, the type of geometry used to represent objects is that of simple polygons (often just triangles) inscribed on planes. Such planar polygons are mathematically and computationally easy to deal with. They have the advantage that most shapes can be approximated by collections of polygons or triangles, even curved shapes, provided that a relatively large number of
triangles are used in the approximation. The ‘phantoms’ that comprise virtual worlds are usually made of polygons.
CHAPTER 3

Lighting - the Radiance Equation

Lighting The Fundamental Problem for Computer Graphics

So far we have a scene composed of geometric objects. In computing terms this would be a data structure representing a collection of objects. Each object, for example, might be itself a collection of polygons. Each polygon is sequence of points on a plane.

The ‘real world’, however, is ‘one that generates energy’. Our scene so far is truly a phantom one, since it simply is a description of a set of forms with no substance. Energy must be generated: the scene must be lit; albeit, lit with virtual light.

Computer graphics is concerned with the construction of virtual models of scenes. This is a relatively straightforward problem to solve. In comparison, the problem of lighting scenes is the major and central conceptual and practical problem of computer graphics. The problem is one of simulating lighting in scenes, in such a way that the computation does not take forever. Also the resulting 2D projected images should look as if they are real. In fact, let’s make the problem even more interesting and challenging: we do not just want the computation to be fast, we want it in real time. Image frames, in other words, virtual photographs taken within the scene must be produced fast enough to keep up with the changing gaze (head and eye moves) of
people looking and moving around the scene - so that they experience the same visual sensations as if they were moving through a corresponding real scene. Ideally, the simulation of lighting would be so accurate, and the speed of computation so great that they would be unable to distinguish between the real and the virtual. Even more: objects in the scene could be moving, the light could be changing (e.g., a sunset), and the effect of the representations of the people immersed in the scene should also be taken into account as part of the lighting computation. In other words, I should see a reflection of my virtual self in the eyes of your virtual self.

Yes, well, this problem awaits solution.

**Light**

Why is the problem so hard to solve? It is because of the complexity of the interaction between light and objects in a scene. A full description of this is beyond the scope of this book (see, for example, Glassner, 1995). In this section we present some of the problems.

Visible light is electromagnetic radiation with wavelengths approximately in the range from 400nm to 700nm. Wavelength gives rise to the sensation of colour (more about this later). It is well known that light has properties compatible with it being a wave phenomenon and also a particle phenomenon. In other words if a wave model of light is adopted, and experiments carried out on this basis then results are obtained which are compatible with the wave theory. On the other hand, the same is true if a particle model of light is adopted - results are compatible with this view of light too.

The particles are called photons, packets of energy which travel in a straight line in vacuum with velocity \(c\) (the usual symbol for the speed of light, approximately 300,000m per second). The photons may also of course be thought of as ‘waves’. Each photon carries energy \(E\) which is proportional to frequency of its corresponding wave:

\[
E = hf
\]  

(EQ 53)

where \(h\) is called Planck’s constant, and \(f\) is the frequency. The associated wavelength is inversely proportional to the frequency (the greater the frequency per unit time shorter the wavelength). In fact:
since length of the wave times the number of waves per second must be the velocity.

Photons have another unusual property that they do not interfere with one another - two rays of light crossing one another do not interact. For example, photons along paths that reach into my eyes are not affected by other rays of photons crossing in front of my eyes.

The problem of how light interacts with surfaces in a volume of space is an example of a transport problem. In general this is concerned with the distribution of moving particles in a space - for example, the distribution of vehicles on a road system can be studied with equations similar to those that occur in light transport (though the study of most transport problems, including - unfortunately - vehicles, must take into account the effect of collisions amongst the particles).

We denote by $\Phi$ the radiant energy or flux in a volume $V$. The flux is the rate of energy flowing through a surface per unit time (it is measured in watts). The energy is proportional to the particle flow, since each photon carries energy. The flux may be thought of as the flow of photons per unit time.

In fact the energy is proportional to the wavelength, so that to fully specify radiant energy in a volume we should use the notation $\Phi_{\lambda}$, i.e., the radiant energy at wavelength $\lambda$. For the time being we shall drop the $\lambda$, and consider $\Phi$ as representing a specific wavelength. In perceptual terms, we have said that $\lambda$ produces the sensation of colour. Flux is related to the sensation of brightness.

Now consider the total flux in volume. First, this must be in dynamic equilibrium - that is, although of course particles are flowing through the volume, the overall distribution remains constant (for example, parts of a scene do not spontaneously become brighter or darker over time, other things being equal). It seems to an observer that immediately a light emitter is ‘switched on’ in a scene that the light energy is instantaneously distributed through the scene, and that the illumination remains constant. Of course it is not instantaneous, but appears so because of the speed of light.

Second, the law of conservation of energy applies. The total light energy input into the volume must be equal to the total energy that is output by or absorbed by matter within the volume. Light may be input into the volume in two ways: it flows in...
from outside (in-scattering) or it is emitted from within the volume itself (emis-
sion). Light may be output from the volume by streaming through it without in-
teracting with any matter within the volume, or by interacting with matter in the
volume and then being reflected out (out-scattering) or by being absorbed by that
matter (absorption).

Hence we have the equation:

\[ \text{emission} + \text{in-scattering} = \text{streaming} + \text{out-scattering} + \text{absorption} \]  \hspace{1cm} (EQ 55)

Let’s tie this down a bit further. Let \( \Phi(p, \omega) \) be the flux at point \( p \) in direction \( \omega \),
where \( p \in V \) the volume, and \( \omega \in \Gamma \) a set of directions of interest (hence \( \omega \) is of the
form \((\theta, \phi)\)). Now each of the events, other than emission, in (EQ 55) is expressed
in probabilistic terms. For example, absorption is expressed as the probability per
unit time of a particle at point \( p \) travelling in direction \( \omega \) being absorbed: \( a(p, \omega) \).
Hence the total absorbed at this point and direction is \( a(p, \omega) \Phi(p, \omega) \). If this is now
integrated over the volume and directions, the total absorption in the volume is:

\[ \Phi_a = \int_{\Gamma V} a(p, \omega) \Phi(p, \omega) dp d\omega \]  \hspace{1cm} (EQ 56)

Similarly, let \( k(p, \omega, \omega') \) be the probability that a photon travelling in direction \( \omega \) at
point \( p \) will be deflected into direction \( \omega' \). Then the total out-scattering is:

\[ \Phi_o = \int_{\Gamma V \Omega} k(p, \omega, \omega') \Phi(p, \omega) d\omega' dp d\omega \]  \hspace{1cm} (EQ 57)

(Recall that \( \Omega \) is the set of all directions on the unit sphere).

There is a similar expression for in-scattering, \( \Phi_i \), but with the roles of \( \omega \) and
\( \omega' \) reversed.

The streaming is the total flux over the whole surface, \( S \), of \( V \) in all directions. This
is easy to write down as the integral:

\[ \Phi_s = \int_{\Gamma S} \Phi(p, \omega) dp d\omega \]  \hspace{1cm} (EQ 58)
Finally emission is specified as $\varepsilon(p, \omega)$, the flux emitted (photon energy per unit time) at point $p$ in direction $\omega$. Hence total emission in the volume is:

$$\Phi_e = \int_{\Gamma V} \varepsilon(p, \omega) d\omega$$

(EQ 59)

Now (EQ 55) may be written as:

$$\Phi_e + \Phi_i = \Phi_s + \Phi_o + \Phi_a$$

(EQ 60)

Why do we care about all this? What has it got to do with computer graphics? The answer is surprising: if we knew $\Phi(p, \omega)$ then we would have a complete solution to the problem of lighting in computer graphics. It tells us the light energy flowing in every direction in which we might be interested ($\Gamma$) at every point throughout a volume of space ($V$). For example, suppose we had a virtual eye or camera in the scene, and wished to use that to make an image (‘the phantom world of projections’). Then we would find all the rays of light that enter the lens (i.e., a set of $\omega \in \Omega$) and reach the retina of the eye or photographic film. This lens would correspond to a set of points $p$. Each ray that hits the surface of the lens therefore carries energy $\Phi(p, \omega)$ which determines the response of the eye or camera. An image would be formed, on the virtual retina or ‘film’ plane, which when viewed by a person generates the sensations associated with seeing the virtual scene. (Of course how this can be done is the subject of the rest of this book!).

Now how can we find $\Phi(p, \omega)$? We could try to solve the integral equation (EQ 60) for $\Phi(p, \omega)$. This is extremely hard and in practice cannot be achieved. Computer graphics is essentially about a series of different approximations and approaches to the solution of (EQ 60). The types of solution adopted depend on the requirements: a solution that delivers real-time performance is totally different to one that produces illumination realism, and there are compromises between these two extremes.

**Simplifying Assumptions**

In order to get anywhere at all with the solution of (EQ 60) a number of simplifying assumptions are made which are usually applied across almost the whole realm of computer graphics.
Wavelength independence. It is customary to assume that there is no interaction between wavelengths. Therefore (EQ 60) can be solved at a sample of different wavelengths, and then these solutions combined together to estimate the wavelength distribution at a point along a ray. This rules out fluorescence. This is the process by which material absorbs light at one wavelength to reflect it out at a different wavelength within a short period of time.

Time invariance. It is assumed that any solution of the equation for the distribution of energy remains valid over time unless something changes in the scene itself (eg, an object moves position). This rules out phosphorescence which occurs when energy absorbed at one moment is emitted later after a (relatively) large time delay.

Light transports in a vacuum. A very important simplifying assumption is that the volume of space through which light transports is a vacuum, though one dotted with the graphical objects of interest in a scene. In other words the medium through which light travels is a vacuum (a non-participating medium) rather than a participating medium. In this case a number of elements of the equation become simpler: absorption and out-scattering only occur at the boundaries of surfaces. There is no emission except from objects. There is no scattering or absorption, again except at the surfaces of objects. What this means is that a light ray that is unobstructed (i.e., a ray between two objects - sometimes called ‘free space’) has none of its energy absorbed, or scattered along that ray, and there are no particles of matter along the ray which themselves generate additional light energy. This is a reasonable assumption for scenes representing building interiors - though even in this case the effects of dust cannot be taken into account. It is a less reasonable assumption for outdoor scenes - it rules out rain and fog or any kind of lighting effects caused by the atmosphere.

Objects are isotropic. When photons strike a surface their energy may be partially absorbed and partially reflected. In an isotropic material, if we consider the incoming (or incident) direction of light, and an outgoing (reflected) direction, the relationship between these is the same over the whole surface of the object. This simplifies the types of expression involved in the $k(p, \omega, \omega')$ term, since now $k$ is independent of $p$ on the surface of such an object. An anisotropic material does not have this property.
Radiance

We have used flux in the discussion above because it is something that is easy to visualise - particles travelling through a volume, the number and rate of flow of such particles, and the effects of their collisions with matter in the volume. However, the real quantity of interest in computer graphics is not the flux itself, but a derivative of this quantity called the radiance. Radiance \( L \) is the flux that leaves a surface, per unit projected area of the surface, per unit solid angle of direction.

Let \( dA \) be the surface area from which the energy is leaving in direction \( \theta \) relative to the normal to \( dA \), through a differential solid angle \( d\omega \). This is shown in Figure 12. Then if the radiance leaving the ‘differential area’ \( dA \) is \( L \), the corresponding flux is:

\[
d\Phi = LdA\cos\theta d\omega
\]

As we let \( dA \) and \( d\omega \) become vanishingly small \( d\Phi \) represents the flux along a ray in direction \( \theta \). Another way to think about radiance then is that \( L(p, \omega) \) is the function that is integrated over the solid angle and projected area in order to obtain the radiant power (flux) from that area.

**FIGURE 12. Radiance is flux per unit projected area per unit solid angle**

\[
d\Phi = LdA\cos\theta d\omega \quad \text{(EQ 61)}
\]

*Lighting - the Radiance Equation* 41

*Great Events of the Twentieth Century* 41
In Figure 13 we show two small patches, \( dA \) and \( dB \). Suppose \( r \) is the distance between them, and the radiance from \( dA \) to \( dB \) is \( L \). Let’s write \( \Phi(dA, dB) \) as the radiant power transferred. Then:

\[
\Phi(dA, dB) = L dA \cos \theta_A d\omega_A \tag{EQ 62}
\]

![FIGURE 13. Flux between two patches](image)

However, using (EQ 23),

\[
d\omega_A = \frac{dB \cos \theta_B}{r^2}
\]

\[
\therefore \Phi(dA, dB) = \frac{L dA \cos \theta_A dB \cos \theta_B}{r^2} \tag{EQ 63}
\]

Note also that by again making use of (EQ 23) and rearranging, we obtain:
Radiance

\[ \Phi(dA, dB) = LdB \cos \theta_B \left( \frac{dA \cos \theta_A}{r^2} \right) \]

\[ = LdB \cos \theta_B d\omega_B \]

\[ = \Phi dB, dA \]  \hspace{1cm} \text{(EQ 64)}

(EQ 64) shows that reversing the direction of the flow of light makes no difference to the flux - following the general principle that equations involving light energy are invariant when the direction is reversed.

(EQ 63) shows that flux diminishes by an amount equal to the inverse square of the distance - so that when the two areas are moved further apart the flux is diminished accordingly. However, this is not the case for radiance: radiance along a ray is constant - it is independent of the distance from the source.

This leads to a fundamental principle in computer graphics: we tend to forget about photons flowing through a volume of space, and instead abstract away from this underlying physics to think just about rays carrying light energy represented as radiance. \textit{For computer graphics the basic particle is not the photon and the energy it carries but the ray and its associated radiance.}

There are three other characterisations of light energy that are important in computer graphics. The first is called \textit{radiant intensity}. It is the radiant power (or flux) per unit solid angle (steradians). Hence, if \( I \) is the radiant intensity, then the associated radiant power is given in (EQ 65):

\[ d\Phi = Id\omega \]  \hspace{1cm} \text{(EQ 65)}

Hence, comparing with (EQ 62):

\[ I = LdA \cos \theta \]  \hspace{1cm} \text{(EQ 66)}

where \( L \) is the corresponding radiance.

The second is called \textit{radiosity}. It is the flux per unit area that radiates from a surface, and is usually denoted by \( B \). Hence, if \( B \) is the radiosity associated with the energy leaving area \( dA \), the flux can be recovered as:

\[ d\Phi = BdA \]  \hspace{1cm} \text{(EQ 67)}
Irradiance is the flux per unit area that arrives at a surface. It is usually denoted by $E$, and if the irradiance arriving at $dA$ is $E$ then the flux is:

$$d\Phi = EdA \quad (\text{EQ 68})$$

Suppose $L(p, \omega)$ is radiance arriving at point $p$ along direction $\omega$, then it follows from (EQ 61) that:

$$E(p, \omega) = \frac{d\Phi}{dA} = L(p, \omega)\cos\theta d\omega \quad (\text{EQ 69})$$

Reflectance

FIGURE 14. A specularly reflecting surface

Up to now we have talked about the distribution of light energy in a scene, and made oblique references to light being absorbed and reflected from surfaces. Here we outline how this is taken into account in the computer graphics model for lighting. Suppose a ray of light energy hits a surface at $p$, with incident direction $\omega_i$. The volume over which that energy may be reflected is the hemisphere whose base is the tangent plane at $p$. The reason for this should be clear: the hemisphere contains the entire set of directions visible from $p$ which are not obscured by the surface of which $p$ is a part. (A 2D analogue of this is shown in Figure 15.)
FIGURE 15. Light is radiated in a hemisphere about a point on a surface

The next question to ask is how much energy leaves the surface as a result in a reflected direction $\omega_r$? We introduce the term $f(p, \omega_i, \omega_r)$ called the bidirectional reflectance distribution function (BRDF) which relates the reflected radiance at point $p$ in direction $\omega_i$ to the incoming irradiance to point $p$ in incident direction $\omega_i$. Then:

$$L(p, \omega_r) = f(p, \omega_i, \omega_r)E(p, \omega_i)$$  \hspace{1cm} (EQ 70)

Specification of $f(p, \omega_i, \omega_r)$ correctly for real surfaces in the world is an extremely complex task, especially if the surface is anisotropic (in which case $f$ does vary with $p$). Computer graphics practice abstracts away from the complexity of the real world and mainly uses two idealisations of material properties called specular reflectors and diffuse reflectors, and mixtures of these. A specular reflector is a mirror-like surface, which reflects the incident ray in completely in one outgoing ray such that the angle of incidence is equal to the angle of reflection (Figure 14). These angles are measured relative to the normal to the surface at $p$. Also the vectors corresponding to $\omega_i$ and $\omega_r$ lie on the same plane as the normal. (A very interesting proof of this can be found in Feinman). A diffuse reflector is a ‘rough’ surface, which scatters the energy of the incoming ray equally in all directions over the hemisphere above $p$. In this case it possible to show that:

$$f(p, \omega_i, \omega_r) \propto \frac{1}{\pi}$$  \hspace{1cm} (EQ 71)
where the constant of proportionality is the reflectivity of the material (i.e., the proportion of incoming flux which is reflected out across all directions).

More complex BRDFs can also be constructed. For example, with glossy reflection, an incident ray would have its energy dispersed in a cone about the ideal specularly reflected ray $\omega_r$. A BRDF is typically constructed as a mixture of diffuse and specular components.

We have tacitly assumed that surfaces are opaque. However, transparent surfaces must also be taken into account. The approach is much the same, except that the light rays now would pass through different media (for example from air to inside a block of ice) and the direction of transmission depends on the density of the material. This is discussed in more detail in Chapter XXX (but also see Feinman).

**The Radiance Equation**

We said that the lighting problem would be completely solved if we could find $\Phi(p, \omega)$ for all points and directions, and that there is an equation for this function (EQ 60). We then introduced a number of restrictive assumptions on the road to making this equation more palatable, and also stated that the function really of interest is radiance rather than radiant power, $L(p, \omega)$ rather than $\Phi(p, \omega)$. With all the simplifying assumptions that have been suggested (and additional ones, beyond the scope of this book), and using the various terms that have been introduced, it is possible to derive a new equation from (EQ 60) that is much simpler to understand, expressed in terms of radiance rather than flux. This is called the radiance equation or rendering equation, and provides an equation for radiance at point $p$ on a surface along the ray given by direction $\omega$.

This equation expresses the fact that this radiance must be the sum of two quantities. The first is the amount of radiance directly emitted at this point (if any). For example, the point may itself be on a surface which is a light source. The second quantity is the amount of radiance reflected at this point. The amount reflected can be computed as the sum over all rays coming into $p$ (the total irradiance) multiplied by the BRDF for this surface. Hence,

$$\text{radiance} = \text{emitted radiance} + \text{total reflected radiance} \quad (\text{EQ 72})$$
The Radiance Equation

For any incoming direction \( \omega_i \) the reflected radiance in direction \( \omega_f \) is the irradiance multiplied by the BRDF. Using (EQ 69) and (EQ 70) this is:

\[
f(p, \omega_i, \omega) L(p, \omega_i) \cos \theta_i d\omega_i
\]

(EQ 73)

If we integrate (i.e., 'sum') this over the hemisphere of all incoming directions at \( p \), we obtain:

\[
\text{total reflection} = \int f(p, \omega_i, \omega) L(p, \omega_i) \cos \theta_i d\omega_i
\]

(EQ 74)

The radiance equation is therefore:

\[
L(p, \omega) = L_e(p, \omega) + \int f(p, \omega_i, \omega) L(p, \omega_i) \cos \theta_i d\omega_i
\]

\[
= L_e(p, \omega) + \int \int f(p, \omega_i, \omega) L(p, \omega_i) \cos \theta_i \sin \theta_i d\theta d\phi
\]

(EQ 75)

where \( L_e(p, \omega) \) is the emitted radiance.

This is an integral equation for the radiance function \( L \), and the rest of this book is about how to solve it.

We have restricted \( p \) to lie on surfaces but in fact this restriction is unnecessary. Suppose \( p \) is not on a surface, then since radiance does not change along a ray in free space, we can trace the ray backwards in direction \( \omega \) starting from \( p \) until we encounter a surface at \( p' \). Then \( L(p, \omega) = L(p', \omega) \).

Figure 16 shows a schematic representation of the radiance equation. Consider any incoming ray at \( p \). According to the equation we need to compute the radiance along this ray. So we trace backwards along the incoming direction \( \omega_i \) until we hit another surface at (say) \( p' \), and find the radiance \( L(p', \omega_i) \). But in order to find this we need to invoke the radiance equation again. In other words, corresponding to each of the (uncountably infinite) number of incoming rays suggested in Figure 16 there is another copy of the same diagram, showing how the radiance along that ray was generated.
Putting this in everyday terms, the radiance equation highlights the fact that illumination is *globally determined*. The light you see reflected from this page of the book depends on the incoming light to the page, and on the material properties of the surface of the page (the BRDF - how it reflects light). The incoming light to the page depends on the direct light sources where you are located, but also all the indirect light from other surfaces in your environment. The light reflected from those surfaces in turn depends on the direct light reaching them, and all the indirect light from other surfaces in the environment, and so on.

**FIGURE 16. Illustrating the radiance equation**

![Diagram of radiance equation]

**Solutions of the Radiance Equation**

In this section we consider various approaches to the solution of the radiance equation, and in a real sense this section is an abstract guide to much of the rest of this book.

In computer graphics applications the main interest in the radiance equation is that it implicitly embodies the totality of all possible 2D views, i.e., images, of the scene that it represents. The problem then becomes how to extract the information needed to construct such images. The process of extracting 2D projected images from the radiance equation is called *rendering*. 
There are two approaches to this: the view-independent and the view-dependent solution. View-dependence means that the radiance equation is solved only for that set of rays that are needed to form an image. We will study the mechanics of specifying such a set of rays in Chapter 2 and onwards - for the time being, as we mentioned earlier, it is the set of rays that are entering an ‘eye’ or the lens of a camera. These are the only rays that are visible to such an eye, the only rays that cause the sensation of vision. Such a solution is called ‘view-dependent’ because if the viewing situation changes, for example, if the eye looks in a different direction, then the entire process of computing the radiance along the eye-visible rays must be carried out again. So a view-dependent solution specialises in computing \( L(p, \omega) \) for the set of \((p, \omega)\) such that \(p\) is on the surface representing the lens, and \(\omega\) corresponds to the directions of rays through that lens.

A view-independent solution concentrates on pre-computing values for \(L(p, \omega)\) across all surfaces of a scene and for as many directions as possible. Now when a particular image is required, the set of \((p, \omega)\) corresponding the surface and directions through the lens are computed and the corresponding values of \(L(p, \omega)\) are ‘looked up’ rather than computed. This approach is called ‘view-independent’ because a solution for \(L(p, \omega)\) is found without regard to any particular viewing directions. View-independence has the advantage that producing an image of the scene is a constant time operation independently of the scene complexity and the particular view: it is just the time to compute the ray directions and intersections with the lens, and look up the corresponding \(L\) values.

Approaches to the solution of the radiance equation can be independently classified in two further ways - depending on whether the solution is local or global. A local solution at most takes into account only the direct effect of light sources on objects, and does not take into account inter-object reflection. This completely eliminates the recursion in (EQ 75). The integral is effectively replaced by a sum over those incident directions corresponding to rays coming from light sources (who’s radiance values are known and therefore do not have to be computed recursively). To simplify things further, it is usually the case that such light sources are single points, so that for each point on the surface of an object there will be one ray only that represents an incoming radiance from a specific light source.

A global solution is one that takes into account the recursive nature of the radiance equation - in other words at least some type of object inter-reflection is accounted for. The method called ‘ray tracing’ only takes into account object interreflection for specular surfaces. The method called ‘radiosity’ only takes into account object interreflection between diffuse surfaces. Other methods, usually called ‘Monte
Carlo’ methods statistically sample large sets of ray directions and find an approximate global solution based on these.

**TABLE 1. Types of Solution of the Radiance Equation**

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>View dependent</td>
<td>‘real-time’ graphics</td>
<td>ray tracing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Monte Carlo path tracing</td>
</tr>
<tr>
<td>View independent</td>
<td>‘flat-shaded’ graphics</td>
<td>radiosity</td>
</tr>
<tr>
<td></td>
<td>(image based rendering)</td>
<td>Monte Carlo photon tracing</td>
</tr>
</tbody>
</table>

TABLE 1. classifies a number of approaches to computer graphics according to the type of solution of the radiance equation (local or global) and the phase-space of the solution (view dependent, or view independent). We briefly consider each in turn.

**Flat-shaded graphics.** The solution adopted for the radiance equation (EQ 75) is extremely simple: it is to ignore the integral completely so that \( L(p, \omega) = L_e(p, \omega) \). In practice this means that each object has a pre-assigned radiance (think of this as a colour). For any viewpoint the objects are simply projected to 2D and displayed in their colour. It is view-independent because clearly the ‘solution’ is obtained without taking any viewpoint into account. It is ‘local’ because obviously there is no recursion in the solution, no impact of even primary light sources - except in the sense that each object is treated as a primary source.

**Ray tracing.** The radiance equation is simplified by only allowing point light sources and with BRDFs that only account for specular reflection. Suppose for example that there is only one (point) light source in a scene. Then each \( f(p, \omega_i, \omega_o) \) will be non-zero for only one combination of \( \omega \) and \( \omega_o \) - that is where the angle of incidence equals the angle of reflection. Hence there will be (at most) one incoming ray and one outgoing ray for each each surface point. Now consider an ‘eye-visible ray’ that is a ray that is passing in an appropriate direction through the lens of the eye. Find the surface point \( p \) in the scene that is the origin for that ray. Trace a ray from \( p \) to the point light source and thereby compute the quantity \( L_e(p, \omega) \) (which will be easily computable given the known radiance emitting properties of the light). This quantity also includes a fudge called ‘ambient light’ which is supposed to represent a total background illumination due to indirect lighting, and another fix called ‘diffuse light’ which adds in a term to represent any diffuse characteristics of the surface (which is not globally accounted for in ray
tracing). Now the remainder of the radiance equation is taken care of by recursively tracing the ray in the reflected direction from \( p \) until it hits another surface, and the radiance along that ray again computed recursively. The recursion continues until the amount of radiance added by a ray falls below some pre-assigned threshold. So each primary (eye-visible) ray spawns a whole tree of recursively followed reflected rays, each of which carries their radiance back up the tree towards the primary. Note that the light paths are being followed *backwards* - from the eye out to the scene! Ray tracing is view-dependent because solutions are found only for the specific set of rays entering the eye. It is global because the solution includes a special form of the recursive integral term.

**Monte Carlo Path Tracing.** This is similar to ray tracing - but can produce an estimated solution to the radiance equation that includes both specular and diffuse reflection. It follows the same general principle in that primary rays are traced into the scene from the eye. However, instead of the recursive rays following specific paths determined by the direction for specular reflection, rays are generated that follow a randomly selected direction. Thus given an intersection point \( p \) on a surface, the BRDF is sampled to randomly choose a ray to follow. This is continued from intersection point to intersection point until successive contributions are negligible. The whole process is repeated many times for each primary ray, and the results averaged together. Hence the space of all solutions for the particular eye is sampled. This is a view-dependent method because it starts from the primary eye-visible rays, and the solution is only valid for a particular view. It is clearly a global illumination method since it is a stochastic solution to the full radiance equation.

**Real-time graphics.** The radiance equation is simplified further - the recursive element is completely removed. Light sources are again points. Only direct lighting is included - which means that the integral is replaced by a sum over light sources only, and only the local contribution of each light to \( p \) on a surface is computed. This is clearly a local solution. This approach differs from ray-tracing in another sense: in ray tracing the primary eye-visible rays are traced out through the scene. In real-time graphics whole objects are ‘projected’ to the lens surface - thus becoming 2D entities (rays are not explicitly used at all). In this 2D projected space pre-computed radiance values on the boundaries of the objects are interpolated to fill in the 2D visible solid areas of the objects. Ray tracing involves searching along the path of rays for objects that intersect the rays. Real-time graphics avoids this by directly projecting the objects to the lens surface - thus there are no computationally intensive ray-object intersection calculations. It is a view-dependent solution because the projections of objects are, of course, only valid for a particular viewpoint and direction of view of an eye.
Radiosity. The solution adopted for the radiance equation is to reduce the BRDF $f(p, \omega_i, \omega_o)$ to a constant with respect to direction, and entirely eliminate all directional aspects of the equation. As the name implies the method also transforms the equation to be re-expressed in terms of radiosity rather than radiance. Now how is it possible to get away with eliminating all directional considerations? The answer is to assume that all surfaces are diffuse reflectors only. Recall that ideal diffuse reflectors scatter the energy of an incoming ray equally in all directions - hence direction is irrelevant. The surfaces of the scene are divided into small surface elements. The radiance equation reduces to a system of linear equations (one equation for each small surface element in the scene) where the unknowns are the radiosities associated with the surface elements. The solution to this equation therefore results in a radiosity being associated with each small surface element in the scene, or alternatively to points on the boundaries of those surface elements. It is a view-independent solution because it is in terms of the scene itself rather than for any particular set of rays for a specific eye. It is clearly a global solution, since it does take into account inter-reflection between the diffuse surfaces. Once the radiosity computation has been carried out it is then possible to use the approach of real-time graphics to produce specific views.

Monte Carlo Photon Tracing. Jensen, 1996 used a two-pass method in tracing rays from the light sources, combined with the usual method of tracing rays to the viewpoint. This has been extended by Shirley et al., 1995 into a multi-stage method which has the advantage of producing a globally illuminated scene for real-time walkthrough, which makes use of statistical density estimation techniques. The method involves producing a fine triangulation of all (polygonal) objects in the scene. Photons are traced from light sources, with energy determined by the energy distribution of the emitters. Intersections with objects are followed by further Monte Carlo based tracing, until the particles are finally absorbed. When there is a hit on an object surface, the information is stored in the corresponding triangle. At the end of this process (which can take several tens of hours for a realistic scene) all triangles are sorted into object order. Each triangle potentially has an associated distribution of particle hits, which is then used to estimate radiance density function over that patch. Finally adjacent triangles which are similar enough according to some tolerance criterion are merged together (this last step only to reduce the total number of triangles in the interest of rendering speed). The scene can then be rendered in real-time using Gouraud shading. However, this is not then a full global solution for specular highlights and reflections are not treated correctly at this rendering stage.
Image Based Rendering. This is included in Table 1 on page 50 in brackets, because it doesn’t quite belong. In image based rendering (light field rendering being a part of IBR) there is typically no lighting computation at all within the method. Some other means has been used to capture a sample of images (whether digital photography or some other rendering system) with known viewing parameters and the essence of this approach to computer graphics is to synthesise new images representing viewing parameters not in the original set from this sample. Now if the original images were captured by a technique that preserves global illumination (for example, photographs of real scenes - provided that they are diffuse reflectors) then IBR will produce ‘globally illuminated’ images. The essential point is that the lighting calculations themselves are not part of the IBR method itself.

Visibility

There is an implicit but highly computationally intensive aspect of the radiance equation which is not obvious - this is the problem of visibility. Consider an incident ray to a surface point \( p \). Where did that ray come from? Clearly from another surface. Which one? Ray tracing has to explicitly search for intersections of the ray potentially with all objects in the scene (though there are ways of speeding up this search). This is one example of the visibility problem: we are only interested in incident rays from surfaces that are visible to the point \( p \). (Even here there are complications, because rays through transparent objects are bent, and so even though a surface may not have straight-line visibility to \( p \), it may still contribute light energy).

The problem of visibility also occurs in real-time graphics, even though there is no explicit ray tracing. We said that all objects are projected to the surface of the lens. But of course not all objects will be visible to the eye - and of course objects may partially obscure one another with respect to a particular viewpoint. So it is clearly not a matter of straight-forward projection of objects - visibility relationships between objects with respect to a particular eye have to be taken into account.

Summary

This chapter has covered several issues:
[Text content from the image]

- We have shown that efficient computation of scene lighting may be regarded as the central problem of computer graphics - it gives substance to the geometric form of scene description.
- We have introduced basic radiometric terms such as flux or radiant power, radiance, irradiance, and radiosity. This is important because computer graphics texts often have a habit of being rather vague about the quantities that it measures, often loosely calling everything ‘intensity’.
- We have shown how the distribution of light in a scene may be described by as a field of radiant power or more appropriately of radiance. A field is a function over points in a multi-dimensional space. The space here is 5-dimensional: each point of the form \((p, \omega)\). All information about the scene is encapsulated in the function \(L(p, \omega)\).
- We introduced the radiance equation - an equation that \(L(p, \omega)\) has to obey to satisfy the requirements of radiance distribution through a scene. This is the central equation of computer graphics, and all 3D display methods used in computer graphics are attempts at solving this equation - or (sometimes gross) simplifications of the equation.
- The process of rendering (2D projected) images of a scene is that of extracting the relevant information from the radiance field. This is equivalent to finding a way to solve the radiance equation - and we have briefly touched on several different methods used in computer graphics.

Computer graphics is about solving the radiance equation. In the next chapter we discuss a particularly elegant, and in principle simple method for solving this equation known as light field rendering.
CHAPTER 4

An Introduction to Light Fields

Introduction

Levoy and Hanrahan, 1996 and also Gortler et al., 1996 provided an interesting image-based solution to the radiance equation (EQ 75) under a number of restrictions. The radiance equation is a recursive expression for radiance $L(p, \omega)$ where $p$ is any surface point and $\omega$ is the set of all directions. Any specific $p$ and $\omega$ together form a ray, hence we can think of $L(p, \omega)$ being defined over the set of all rays with origins at points on surfaces. Hence the domain of $L$ is a five dimensional ray space.

However, radiance is constant along a given direction in ‘free space’, that is, where there are no discontinuities due to intersections with objects. Hence radiance is constant along each direction outside the convex hull of a scene. Figure 17 shows a simple 2D example. Consider the set of all rays with origin on line $s$ which terminate on line $u$. All such rays intersecting the scene consisting of the two shaded boxes can be parameterised in 2D using $(s,u)$. Suppose that a radiance value were (somehow) assigned to each such $(s,u)$, i.e., that $L(s,u)$ was known. Now to form an image, we place an image plane and centre of projection in the scene, and collect all those rays passing through the COP which intersect the image plane.
Clearly this parameterisation is only valid for views outside the convex hull of the scene, for it is only for these rays that the radiance is constant. Views from anywhere in the interior would not be possible, since the radiance abruptly changes along such paths.

For example, suppose an image for a pin-hole camera were required from point \( p \) for the view ‘volume’ within the arrowed lines. Then by capturing the radiance from all the rays between the arrowed lines such an image could be constructed.

In order to construct the complete set of rays (for the convex hull of the scene) four copies of the ‘light slab’ defined by \( s \) and \( u \) would be needed - one as shown, another with the directions reversed, and two others with, for example, two vertical lines on either side of the scene. Nevertheless, the overall representation of the lumigraph is still two-dimensional, albeit with four such two-dimensional representations.

Such a representation is called a light field. A lumigraph is the same idea, though with some additional information.
FIGURE 18. Discrete Representation of a Light Field (One Light Slab)

Staying with this two-dimensional example, and with the single ‘light slab’ for \( s \) and \( u \), the light field radiance would be a continuous function of \( s \) and \( u \), \( L(s,u) \). For computational purposes a discrete representation is required. The two-line parameterisation is illustrated in Figure 18 (with a 2D analogue). Grids are imposed on the \( s \) and \( u \) axes, and then all possible lines between each grid point on one to each grid point on the other forms the discrete representation of the set of all such lines.

The radiance attached to each ray in the light-field would be estimated by choosing each grid point \( s_i \) on \( s \) as a centre of projection with \( u \) as the image plane (and a fixed interval of \( u \) as the view plane window). In this way, every ray that intersects the convex hull of the scene would have an associated radiance determined by the ‘rendering’ of the image formed by the set of perspective projections.

For 3D scenes the ‘two-line’ parameterisation is replaced by a two-plane parameterisation (2PP), the first parameterised by \( (s,t) \) and the second by \( (u,v) \). The light field (in fact a sub-set of it) is then represented by \( L(s,t,u,v) \), and discretised by all possible lines between the two planes defined by a rectangular grid imposed on each. The \( st \) plane is subdivided into squares with each vertex forming a centre of projection, with a rectangular subset of the \( uv \) plane as the view plane window to form an associated image. Hence there is an image associated with each grid point of the \( st \) plane, and a radiance associated with each \( (s,t,u,v) \) combination representing lines that intersect the convex hull of the scene. This describes how to form one
light slab. In order to cover all possible ray directions, six copies are needed - three orthogonal rotations, and two directions in each case.

The light field once constructed can be used for synthesis of an image from a virtual camera which does not correspond to the set of cameras on the \( st \) planes. A new image can be formed by sampling the corresponding set of lines through the viewpoint and in the required directions.

The light field approach was designed as a method of forming new views from images of real scenes. Suppose digital photographs were taken under strict conditions forming the set of viewpoints and directions associated with the \( st \) planes. Such a set of images can clearly be used to construct a light field. A virtual light field, that is a light field for virtual scenes, can also be constructed by using some other rendering system in order to form the images. The only possible advantage of this is where the rendering system includes global illumination with a mixture of diffuse and specular surfaces. The light field approach then provides, in principle, a means for real-time walkthrough of a globally illuminated scene - something which is still not possible in any other way.

In the next sessions we consider some issues in more detail. We mainly consider one slight slab parameterised by \( st \) and \( uv \). The extensions to multiple light slabs are obvious.

**Rendering: Interpolation**

Suppose that the \( st \) plane is discretised with an \( M \times M \) grid, and the \( uv \) plane with \( N \times N \). Then the light field will consist of \( M^2 \) perspective images, each of resolution \( N^2 \). This is illustrated in 2D in Figure 19, the \( st \) point marked COP will have an associated image with view volume between the two thicker lines. All rays converging on this COP will therefore have an associated radiance determined by this image. The image might either have been formed by placing a digital camera in a real scene, or by rendering a virtual scene with the given COP and view volume. The COP is shifted to each point on the \( st \) plane in turn, thus completing the full light slab. The whole process must be repeated for each of five other light slabs forming the full light field in order to attain a complete ray coverage.
Once the light field has been created it may be used for rendering images from camera positions and orientations outside the convex hull of the scene. First we describe the basic way in which this is carried out.

Figure 20 shows the $st$ and $uv$ planes and a new camera position and orientation. Each ray through a pixel of the virtual camera will intersect the $st$ and $uv$ planes. For example, the ray $r$ intersects the $st$ plane at $(s_0, t_0)$ and the $uv$ plane at $(u_0, v_0)$. Hence the pixel corresponding to that ray would be set with the value of
In other words, each primary ray generated from the camera, is used to look up the nearest ray in the 4D space of rays corresponding to the light field.

This method can be implemented very efficiently using texture mapping. Figure 22 shows one grid point \((s_0, t_0)\) on the \(st\) plane, and a square containing all the points on the plane that would be approximated by this particular grid point. Now the square projects to points \(a, b, c, d\) on the \(uv\) plane. Corresponding to \((s_0, t_0)\) there is an image on the \(uv\) plane. This image may be used as a texture to render the \(st\) square for \((s, t)\), with texture coordinates given by \(a, b, c, d\). In this approach therefore the square corresponding to each \(st\) grid-point is rendered using as a texture map the image corresponding to that grid-point. This involves finding the set of texture coordinates for each square, but since neighbouring squares will share grid points the number of projections required is approximately the number of squares drawn. Of course not all \(M^2\) squares would be rendered, depending on the view volume of the camera.

**FIGURE 21. Using Texture Mapping**
In practice this simplistic technique will lead to significant aliasing. An alternative scheme is to use quadrilinear interpolation. There are four nearest neighbours to the intersection point on the \( st \) plane, and four nearest neighbours on the \( uv \) plane, as shown in Figure 22. Consider first interpolation on \( s \) only, with all other parameters fixed. (EQ 76) shows an identity expression \( s \) as a barycentric combination of \( s_0 \) and \( s_1 \).

\[ s = \left( \frac{s_1 - s}{s_1 - s_0} \right) s_0 + \left( \frac{s - s_0}{s_1 - s_0} \right) s_1 \]

or

\[ s = \alpha_0 s_0 + \alpha_1 s_1 \]

where \( \alpha_0 + \alpha_1 = 1 \)

Quadrilinear interpolation makes the assumption that the \( L \) function is itself affine (of course it will not be so). Hence applying this to (EQ 76) we obtain:

\[ L(s, t_0, u_0, v_0) = \left( \frac{s_1 - s}{s_1 - s_0} \right) L(s_0, t_0, u_0, v_0) + \left( \frac{s - s_0}{s_1 - s_0} \right) L(s_1, t_0, u_0, v_0) \]

(EQ 77)

\[ = \alpha_0 L(s_0, t_0, u_0, v_0) + \alpha_1 L(s_1, t_0, u_0, v_0) \]

Repeating this argument for each of the parameters we obtain:

\[ L(s, t, u, v) = \sum_{i=0}^{i} \sum_{j=0}^{j} \sum_{k=0}^{k} \sum_{l=0}^{l} \alpha_i \beta_j \gamma_k \delta_l L(s_i, t_j, u_k, v_l) \]

(EQ 78)
where each of the $\beta_j$, $\gamma_k$, $\delta_l$ are defined in the same way as $\alpha_i$ in (EQ 76), and correspond to $t,u,$ and $v$ respectively. Now of course all the interpolated $s$, $t$, $u$, $v$ values can be easily found using a method similar to finding interpolated texture coordinates during the process of rendering the $st$ squares and $uv$ squares as polygons.

Gortler et al., 1996 show that this method cannot exactly be implemented using the texture mapping hardware. The support for the quadrilinear interpolation at $(s_j, t_j)$ extends out to ABCD in Figure 23, and hence neighbouring grid-points will have overlapping supports. Consider the six triangles surrounding the grid-point. Gortler shows that if each of these 6 triangles is rendered with texture mapping as before plus alpha blending, where $\alpha = 1$ at the grid point and $\alpha = 0$ at the other triangle vertices, then this is equivalent to linear interpolation on $st$ and bilinear interpolation on $uv$. It is suggested that results are not visually distinguishable from full bilinear interpolation.

**FIGURE 23. Support for the Quadrilinear Basis**

Gortler et al., 1996 show that keeping some geometric information together with the light field rays can lead to an improvement in radiance estimation. Figure 24 shows a viewing ray with origin at the COP intersecting an object surface. Now the nearest grid-point on the $st$ plane is shown as $s_0$, and normally $u_0$ and $u_1$ would be used for interpolation. However, it is clear that more accurate information can be obtained from the ray $s_0$ through $u$, and therefore the better interpolation would be between $u_1$ and $u_2$. Gortler suggests storing a polygonal mesh approximating the scene geometry in order to allow for such depth corrections. This approach can result in less blurred images for the same $st$ resolution, since a compensation is introduced for the errors involved in the interpolation. A further analysis concerned
Representing Light Fields

with speed-up to the rendering process, in particular tradeoffs between quality and time can be found in (Sloan, Cohen and Gortler, 1997).

**FIGURE 24. Depth Correction**

![Figure 24. Depth Correction](image)

**Representing Light Fields**

The 2PP has obvious computational advantages, especially with respect to storage (rectangular images) and the use of rendering (especially texture) hardware for reconstruction. However, the representation does not provide a uniform distribution of rays in 4-space - in other words, if we think of any arbitrary ray in space, not all rays are equally likely to have the same degree of approximation of representation in the light field space. In practice, what this means is that the quality of the rendered image will be a function of the viewpoint and viewing direction. Viewpoints near the centre of the \( st \) plane, and directions orthogonal to this plane will result in better images than viewpoints off to one side with oblique viewing directions - simply because the number and distribution of rays will be different at such positions. A thorough analysis of the 2PP as a 4D space can be found in (Gu and Cohen, 1997).

Camahort and Fussel, 1999 have carried out a theoretical study of three alternative light field representations - the 2PP as above, a two-sphere representation (2SP), and a direction and point representation (DPP). These were considered by (Camahort, Lerios, and Fussell, 1998). The 2SP involves placing a bounding sphere around the scene, and partitioning the sphere into a uniform grid. Then all connections between all vertices on the sphere form the parameterisation. The DPP simi-
larly involves placing a sphere around the scene. Now choose a point uniformly at random on the surface of the sphere. This defines a vector from the origin through that point. Consider the plane orthogonal to that vector and through the centre of the sphere, and bounded by the sphere (i.e., a disc at the origin of the sphere). Now choose a uniformly distributed set of points on the disc, as the origin of rays in the same direction as the original vector. The collection of all such rays will form a uniformly distributed set of rays in ray space. Each ray is represented by its direction, and its intersection on the disc through the origin. A different two sphere parameterisation was introduced in (Ihm, Park and Lee, 1997) which puts a bounding sphere around the scene, and then a set of small spheres on the surface of the bounding spheres to represent directions. They also introduce an efficient wavelet compression scheme for this data structure.

Camahort and Fussel, 1999 provide an in-depth analysis of these different schemes (as well as a thorough discussion of uniformity), and show that the direction and point approach results in less rendering bias, and corrections that are simple to apply. A ‘rectangularisation’ of this type of approach is adopted for virtual light fields in the next chapter.

### Practical Issues

There are a number of practical issues to consider. First, what is the ideal placement of the $st$ and $uv$ planes? Clearly $st$ is outside of the scene, whereas $uv$ should be through the centre of the scene. Ideally the geometry of the scene should be as close as possible to the $uv$ plane, so that the grid points are (very rough!) approximations to surface geometry. In practice this means that light fields cannot adequately represent scenes of significant depth. A distribution of rays which is adequate for the representation of near surfaces will not be sufficient for the representation of far away surfaces.

What resolutions should be used for $M$ and $N$? The latter is more important since, as suggested, it determines the degree of approximation of scene geometry. The higher the resolution on the $uv$ plane the greater the accuracy of this representation. Hence in practice $N > M$, and values of $N=256$ and $M=32$ have been shown to be sufficient for images of size $256 \times 256$.

Assume that this resolution is used, then one light slab will require $2^{26}$ rays. Suppose each ray carries 3Bytes, then 192MB is needed, and therefore over 1GB for a
full light field (6 light slabs). Remember that this will only enable static scene walkthrough and relatively low resolution images. Clearly a compression scheme is needed! Levoy and Hanrahan, 1996 used vector quantization (Gersho and Gray, 1992). This is a compression scheme, where the data set is partitioned into clusters, each being represented by one vector found during a training phase. Each vector corresponds to a codebook which stores indices of the members of the cluster which it represents. Decoding is a very fast operation - especially useful in the context of rendering. Levoy and Hanrahan demonstrated compression ratios of more than 100:1 using this scheme.

**Further Developments**

The original light field / lumigraph papers were published in 1996. Since then a great deal of interest has been generated, and we briefly summarise some of the significant developments.

As mentioned earlier, one of the drawbacks of the light field scheme is that it is most appropriate for scenes without significant depth. (Isaksen, McMillan, and Gortler, 2000) discuss how to overcome this problem, in addition to supporting depth-of-field effects and showing how light fields can be used for the basis of auto-stereoscopic display systems. This latter development has also taken up in detail in (Perlin, Paxia and Kollin, 2000).

The light field approach is clearly for static scene walkthrough rather than interaction with objects in the scene. Nevertheless (Seitz and Kutulakos, 1998) allow some degree of interaction by showing how it is possible to edit one image of a scene and propagate the changes to all other images, maintaining overall consistency amongst all the images.

A surface light field is a light field consisting of rays emanating from surfaces - in principle for any point on a surface the set of all rays leaving that point with their associated radiances. Wood, et al., 2000 show how to construct, edit and display such light fields for real scenes. An important general contribution is to show how such compressed light fields may be directly rendered from the compressed representation.

The theory of light field representations was considered in (Camahort and Fussel, 1999) as discussed earlier. Chai, Tong, Chan, and Shum, 2000 provide a thorough
analysis of sampling requirements. This includes a minimum sampling curve which provides the relationship between scene complexity, output resolution, and the image sampling.

Summary

This chapter has introduced the idea of light fields, as an image based approach to providing a ‘solution’ to the radiance equation. The LF approach is almost ‘brute force’ for this purpose - it works with a discrete representation of all possible rays covering a scene, and then assigns radiance to those rays by image rendering. Based on a finite sample of images, which have been used to colour rays, it was shown how images from new viewpoints can be constructed. There was some discussion of efficient rendering using the texture mapping hardware in the context of interpolation schemes. Alternative schemes for representing a uniform set of rays were considered, followed by a brief discussion of recent developments.

In this chapter we have concentrated on the basic ideas of LFs, without considering the source of the images from which they are constructed. The most widespread use of light fields is for virtual walkthroughs of real scenes. In the next chapter we consider the issues involved in using light fields for virtual scenes, and consider one approach, still in its early stages, in more detail.
CHAPTER 5

Virtual Light Fields

Introduction

As we have seen, computer graphics rendering is concerned with estimation of the function $L(p, \omega)$, the radiance at position $p$ in a 3D space, in direction $\omega$. Kajiya, 1986 realised that this function can be expressed as an integral equation: and computer graphics rendering had been (implicitly) concerned with finding approximate solutions to that equation. The integral equation can be written in operator form, and expanded as an infinite series. Then, for example, classical rendering can (almost) be viewed as the approximate solution equal to the first term only of the expansion. This deals only with light transport between emitters and reflecting surfaces, and excludes inter-reflection between non-emitting surfaces. Classical rendering is not exactly the same as this first term approximation because it usually only uses point-light sources and does not easily produce shadows. Its advantage though is it is extremely rapid, to the extent that it has been embodied in current hardware able to produce real-time frame rates - at the cost of sacrificing realistic illumination. At the other extreme, Kajiya derived path tracing, a Monte Carlo approach to the solution of the integral equation. This does produce realistic global illumination, but of course a real-time solution is out of the question. The most up-to-date embodiment of this is in the RADIANCE program (Ward Larson and Shakespear, 1998).
As we have seen the light field and lumigraph approaches provide a kind of brute force approach to the solution of the radiance equation. The light field approach is ideally suited for the reconstruction of views of real scenes. In this case digital pictures must be captured with accurate viewing parameters as required by the light slabs. Of course the same technique can clearly be used for synthetic scenes: a rendering program can produce all the images, and then the light field method used to synthesise new views. If the rendering program is one that uses the standard viewing pipeline with local illumination only (such as OpenGL) then there is really not much point to this - for the original program itself could obviously be used to create an image from any viewpoint. However, if a ray tracing or other global illumination system is used which accounts for directional lighting (as caused by specular or glossy surfaces), then there is a point in using a light field. For, there is today, no other method capable of producing real-time walkthrough for such globally illuminated virtual scenes. As Gortler noted though, the production of a sufficient number of images from a rendering program which supports global illumination can take weeks of processing. For example, a light field with parameters $M = 32$ and $N = 256$ was found to give reasonable results. Hence 6K images each of 256*256 resolution would have to be produced. Suppose that the rendering is to be carried out by path tracing or photon tracing since there is a mixture of diffuse and specularly reflecting surfaces in the scene. Suppose optimistically that each image takes an average of only 5 minutes to produce. Then it would take about 4 days to produce enough images for construction of the light field, and in practice we have found that for quite simple scenes using the RADIANCE package, more than 100 hours of processing were required to produce light field images (for a scene with about 20 lights).

Gortler used the term ‘virtual light field’ to refer to a light field for virtual scenes, as discussed above. In this proposal the term virtual light field (VLF) is extended to mean a method where the light field is constructed for a synthetic scene, but without the use of a separate rendering system. The scene is ‘rendered’ into a light field in such a way that the radiance values are computed directly during the construction of the light field itself. In this approach global illumination, for a mix of specularly and diffusely reflecting materials, can be handled. Moreover, the light field is constructed in such a way that arbitrary walkthroughs are possible, not just those restricted to a convex region around the scene. Image space rendering may be accomplished by filtering interpolated rays through a pin-hole camera model, or through a lens system. Rendering time is constant, and independent of the original number of objects in the scene.

There is growing interest in the use of light fields for synthetic scenes in the context of global illumination. Interest has focussed on the adaptive production of
images in order to construct the lumigraph (Schirmacher, Heidrich and Seidel, 1999), and (Miller and Rubin, 1998) introduce a compression scheme for real-time walkthrough of globally illuminated scenes with precomputation for surface light fields based on stochastic ray tracing.

In the remainder of this chapter we introduce an idea which shows how it is possible to use the light field type of approach to directly compute the global illumination. The method is still in its infancy and should be thought of as an idea, and a programme of research rather than a finished method. The idea may be thought of as a combination of light field and illumination network (Buckalew and Fussell, 1989). Both employ the idea of a fixed ray based data structure which is a discretisation of the distribution of radiance. However, the illumination network carries radiant information in a data structure attached to each object, which includes ‘links’ from object to object. Two objects are linked in this way if there is an unoccluded ray that joins them, and the link is a pointer from one object connecting to the other along such a ray. Since the illumination network carries information on objects its rendering time is still linear dependent on the number of objects in the scene. Objects are subdivided into patches, and the illumination network determines the radiance associated with the patches. It is finally the objects which are rendered. The light field and VLF approach does not render objects, rather objects illuminate the rays, and in a sense, it is the rays that are rendered.

The Virtual Light Field (VLF)

Rendering into a Space of Rays

Any ray in a scene can be represented as a point in 4-dimensional space. In general a sphere can be placed around the scene, for example, and any ray will intersect the sphere at two distinct points. The spherical angles $\theta$ and $\phi$ at each intersection point could be used to represent the ray. Hence the space of ‘all possible rays in the scene’ can be represented by $R^4$, where $R$ may be restricted to a particular interval of the real line valid for the representation.
FIGURE 25. A Ray Intersecting Two Objects: The ray is split into intervals \([0,t_1], [t_1,t_2], \ldots, [t_5,t_6], [t_6,1]\). These are alternately outside and inside objects.

Now an object can be ‘rendered’ into \(R^4\). This involves finding every ray that intersects the object, and dividing it into intervals based on the intersection points. Each successive interval along the ray, intersected by a single object, will be either outside or inside the object. For example, if the object is convex, it will split the ray into three intervals – an outside one, followed by an inside one, followed by an outside one. This is shown for two objects in Figure 25. Each ray can be parameterized as \(p(t) = p + t(q - p)\) where \(p\) is the ray origin, \(q\) is its terminal, and \(0 \leq t \leq 1\). Hence a ray that intersects the object has an associated vector of intervals \([0, t_1, t_2, \ldots, t_n, 1]\) where the \(t_i\) are the parametric intersection points.

Clearly every object in the scene can be rendered into the ray space in this way. Hence initially, associated with every 4D point in \(R^4\) there is an interval \([0,1]\), but as the objects are rendered into the space, the corresponding intervals are subdivided.

In fact, each intersection point is not simply represented by its corresponding t-value, but other information is stored as well. The intersection corresponds to a record, called a T-Intersection, containing the following fields: an identifier for the object intersected, the t-value, a direction, a radiance, and an unshot radiance. The radiance values are initialized to zero. The ‘direction’ allows for energy to be sent in both directions along the ray. One direction is labelled arbitrarily as ‘right’ (from \(p\) to \(q\)) and the other as ‘left’. Given any intersection, the direction is recorded as ‘right’ if the normal to the object at the intersection point makes an acute angle with the vector \(q - p\), else ‘left’.

Hence associated with each ray (or point in 4-dimensional space) that intersects one or more objects is a vector of T-Intersections, ordered in ascending t-value.
along the ray. A ray may be considered as a ‘look-up’ index into an array of vectors of T-Intersections. The ‘virtual light field’ (VLF) is this collection of vectors of T-intersections, indexed by points \( r \in \mathbb{R}^4 \). Initially each vector consists of one interval \([0, 1]\). The vectors are updated as objects are rendered into the VLF.

**Propagating Light**

So far the light field is still virtual, there is no light in the scene. Objects in the VLF may emit, reflect or transmit light. Suppose a light is now ‘switched on’. Find all rays through this emitter. For any ray identified as intersecting the emitter, the interval, or pair of T-Intersection records, containing the intersection along which light is to be directed can be found, and the identifier of the object at the other end of the interval (if any) is immediately available from the T-Intersection at that end of the interval. When an object is found in this way, it is marked as being in receipt of ‘unshot radiance’. This radiance value is written into the ‘radiance’ and ‘unshot radiance’ fields, with the appropriate ‘direction’ of the T-Intersection which has its origin at the emitter. This is repeated for each emitter in the scene, and completes the first cycle of the iteration.

Now each object that is in receipt of unshot energy is considered in turn. Take any such object (O). For each ray through the object, find the corresponding T-Intersection for that ray with the object. Suppose that this T-intersection has unshot radiance incident to the object. From the material properties of the object (its bi-directional reflectance distribution function, BRDF) a set of rays can be found that will reflect this incoming unshot radiance. For example, if the object were an ideal specular reflector then the reflected ray can be found, the appropriate outwards pointing T-Intersection of this reflected ray with the object can be found, and the incoming unshot radiance propagated outwards along this reflected ray (and set to zero on the incident ray). The object identifier at the other end of the reflected ray interval can be looked up, and marked as in receipt of unshot energy. If the object O is an ideal diffuse reflector, then all rays over the hemisphere through the intersection point can be found, and treated similarly – with the corresponding radiance propagated along them, and the receiving objects marked as in receipt of unshot energy. When all rays with incoming unshot energy to O have been treated in this way, the next object is considered, and treated similarly. Of course, once an incident ray has been reflected, its unshot radiance is reset to zero.

The second cycle is complete when all objects have been treated in this way. Then subsequent cycles through all objects with unshot energy are carried out, and in
each cycle, objects are selected in descending order of total unshot energy (so that the most important objects in each cycle are considered first). This scheme is very similar to that used in progressive refinement radiosity (Cohen, Shenchang, Wallace, and Greenberg, 1988).

Unshot radiance is only propagated provided that it is above a given tolerance. The cycles continue until there is no unshot radiance left anywhere in the system above this tolerance, or when the total absorbed radiant power converges within a given tolerance.

The following pseudo-code outlines the overall algorithm. The reader should start from ‘main’ and work backwards.

```plaintext
lightObject(Object O){
    for each ray R intersecting O{
        for each interval (T1,T2) radiating out from O along R{
            set the radiance in T1;
            set the unshot radiance in T1;
            O_2 = object at T2;
            mark O_2 has having received unshot radiance;
        }
    }
}

reflectObject(Object O){
    unmark this object;
    for each ray R intersecting O{
        for each interval (T1,T2) incoming to O at T2 with unshot energy at T1 {
            /*using R and the BRDF(O) to find the set of reflected rays....*/
            for each reflected ray S{
                for each interval (U1,U2) radiating out from O along S{
                    update the radiance in U1;
                    set the unshot radiance in U1;
                    O_2 = object at U2;
                    mark O_2 has having received unshot radiance;
                }
                set the unshot radiance at T1 to 0.0;
            }
        }
    }
}

main(){
    initialise the VLF;
    render all objects into the VLF;
}
```
The Virtual Light Field (VLF)

for each light emitter O lightObject(O);

for maximum number of cycles{
    finished = true;
    for each object O in decreasing order of total unshot radiance{
        if O has unshot radiance{
            finished = false;
            reflectObject(O);
        }
    }
    if(finished) stop;
}

Realization

The above is essentially just one description of how light propagates outwards from light sources through the scene. In order for this to be the basis of a computable method there has to be a way of realizing it so that it executes in reasonably finite time on a real computer. The basis for accomplishing this is through a discrete representation of the VLF.

A discrete representation must have certain properties. First, the representation should be parameterized according to its ‘size’ or density of rays. Suppose it is parameterized by a variable $n$, representing the size of the discrete representation (for example, $n$ could be related to the number of rays). Let $r$ be any arbitrary scene ray. Suppose $\mathcal{R}_n$ is the discrete representation of the VLF. For any reasonable measurement of error $Err$, and any given error $\epsilon > 0$, it must be possible to choose $n$ large enough so that a ray $r_n \in \mathcal{R}_n$ can be found with $Err(r_n - r) < \epsilon$. Second, it must be possible to find the appropriate $r_n \in \mathcal{R}_n$, which is an approximation to $r$, in constant time. Third, there must be a constant time lookup from any $r_n$ to its associated vector of T-Intersections.

With this discretization the description in the previous section is essentially unchanged, but becomes a computable method. The set of all T-Intersection vectors corresponding to rays in $\mathcal{R}_n$ could be represented as a 4-dimensional array indexed on the $r_n$. The T-Intersection vectors could, for example, be represented as binary interval trees. The ‘set of all rays through an object’ becomes computable.
Suppose an object is a perfectly specular reflector. When an incident ray strikes the object, the true direction of specular reflection $r$ can be computed in the usual way. However, $r$ itself is not used, but its approximation in the VLF is used instead. Similarly, if the object is a perfectly diffuse reflector, then all rays in $\mathcal{R}_n$ through the intersection point can be computed, and radiance propagated in the appropriate directions along those. Of course, in the discrete representation there will be no rays exactly through the intersection point - the closest set of rays through the intersection point would be used instead.

**Representation**

It is clear that several representations $\mathcal{R}_n$ of the VLF are possible. The two-plane parameterization, for example, would be suitable. However, in practice an alternative representation is used, described in Section Implementation. An important point about the representation of $\mathcal{R}_n$ is that it should be decomposable into ‘parallel subfields’. A parallel subfield is a collection of all rays in $\mathcal{R}_n$ parallel to some direction vector. $\mathcal{R}_n$ may be thought of as a partition of parallel subfields - each $r_n \in \mathcal{R}_n$ will belong to a unique parallel subfield. Now if this is the case then only one canonical case is required for rendering objects into $\mathcal{R}_n$ – that is, when all rays are vertical, through the $xy$ plane, parallel to the $z$-axis. This is taken up in Section Rendering Objects into the VLF.

**Avoiding Combinatorial Explosion for Diffuse Objects**

Although the method described above is clearly implementable as described assuming a suitable choice of representation for the VLF, a scene with several diffuse reflectors would generate an enormous combinatorial explosion. When an incident ray hits a diffuse surface, all rays – approximately – through the intersection point must be followed, and any of these may hit other diffuse surfaces, and so on. This combinatorial explosion can be completely avoided by borrowing an idea from photon maps.

In fact a diffuse surface can be handled by only two passes in each cycle, where a pass means ‘find all rays through the object’. For explanatory purposes only, suppose that the object has been sub-divided into small surface elements. Each element will maintain an accumulated irradiance carried by all rays incident to the surface in the current iteration cycle. A ‘gathering’ phase finds all rays through the
object, and accumulates the unshot irradiance into the surface elements. Then a ‘shooting’ phase finds all rays through the object, and propagates out the accumulated irradiance found at the surface elements (which are reinitialized to zero at the end of this phase). However, in the method actually used here, rather than subdivide objects into surface elements, which is somehow ‘alien’ to the light-centered philosophy of the VLF method, the energy is accumulated into an additional field of that T-Intersection corresponding to the normal of the object at the intersection point. This has the advantage that the discretization of the object corresponds exactly to that of the VLF.

Each T-intersection therefore has an additional field which potentially stores two values: an accumulated irradiance, and an iteration cycle number. When an incident ray hits a diffuse reflector, the closest ray in the VLF to the normal at that point is found, and the T-intersection along that ray corresponding to its intersection with the object found. The unshot irradiance is accumulated into that T-intersection. During the shooting phase when a ray intersects the object, the T-Intersection corresponding to the normal at that point is found again, and the required fraction of accumulated irradiance is propagated out along the appropriate interval of the original intersecting ray.

It is necessary to avoid finding all rays through the object for a third time to reinitialize all the accumulated irradiances to zero when an object’s shooting phase is complete. Instead, for each ray intersecting the object during the gathering operation, the current iteration cycle is compared with the one written into the T-Intersection field, and the irradiance is reset to the new value if this does not match the actual iteration cycle (and the stored iteration cycle in the T-Intersection is updated).

**Viewing and Display**

In order to produce images a virtual ‘eye’ can be placed anywhere in the scene at any viewing orientation. The eye consists of a lens (or any arrangement of lenses) and an image plane. Consider the simplest case of a single convex thin lens. Clearly the lens is just a planar polygon, and all rays through the lens can be trivially and rapidly generated. The T-Intersection corresponding to any ray striking the lens can be looked up. If the T-Intersection carries radiance in the appropriate direction (to inside the ray) then the lens formula may be used to compute the refracted ray, which can then be intersected with the image plane. Thus radiance is carried to the image plane. The image plane can be mapped to a 2-dimensional
window on workstation display, itself divided into ‘super-pixels’, and any sort of filtering required at this last stage can be carried out as desired.

There are two important features of this method. First, obviously the lens properties (focal length etc.) can be altered as desired between frames at no cost. Second, the time for this whole operation of capturing an image onto the image plane through a lens is essentially constant for any eye position anywhere in the scene. It is independent of the original number of objects in the scene. It depends only on the size of the discrete VLF representation.

**Implementation**

**Requirements for the Representation of the VLF**

The VLF representation must have the properties outlined in section ‘Realization’:

- $\mathbb{R}_n \rightarrow R^4$ as $n \rightarrow \infty$, determining the accuracy of the representation. Two representations can be compared as to how large $n$ has to be to achieve a given level of accuracy.
- If $T(r_n)$ is the T-Intersection vector corresponding to $r_n \in \mathbb{R}_n$, then $T(r_n)$ must be found in (small) constant time.
- If $r \in R^4$ is any ‘true’ ray, and $\mathbb{R}_n(r) = r_n \in \mathbb{R}_n$ is the closest approximating ray, then $\mathbb{R}_n(r)$ must be computable in (small) constant time.
- $\mathbb{R}_n$ must be partitioned into an exhaustive and mutually exclusive set of parallel subfields.

A representation should achieve a uniform distribution of rays over the space. Ideally this is equivalent to finding a discrete uniform distribution of points over a sphere. Although this is easily possible using a recursive triangulation this method does not satisfy the second two requirements (referred to here as indexability). The two-plane parameterization could be used, but it has been avoided for two reasons. First, the complication of requiring three orthogonal ‘light slabs’ introduces an inconvenient discontinuity into the representation. Second, although it can be constructed in a way compatible with the parallel-subfield requirements, additional information would have to be stored about the spherical angles associated with
each subfield. An alternative representation has been tried which satisfies the first three requirement, and naturally forms a parallel subfield partition.

**Spherical Angles Representation**

A representation based on spherical angles has been explored, used by other authors in the context of computing approximate visibility between portions of a scene in (Chrysanthou, Cohen-Or and Lischinski, 1998), though the specific sampling scheme used is different here.

Any ray direction can be represented in spherical coordinate form, as two angles \((\theta, \varphi)\) (with \(\varphi\) as the longitude). Consider, for example, the special case of rays parallel to the z-axis (vertical rays), with \(\varphi = 0\) (in this case \(\theta\) can be anything).

Take a cuboid volume defined by \([-1, 1]^3\). Impose a rectangular grid on the \(xy\) plane, with \(n\) equally spaced subdivisions on each axis, the width of each interval being \(2/n\) along each of the \(x\) and \(y\) axes. Discrete coordinates are defined by the centers of these intervals. This imposes a coordinate grid on \(-1 \leq x \leq 1, -1 \leq y \leq 1\), referenced by \((i,j)\). The real coordinate on the \(xy\) plane corresponding to \((i,j)\) is therefore:

\[
(x_i, y_j) = \left(\frac{2i + 1}{n} - 1, \frac{2j + 1}{n} - 1\right)
\]

(79)

This is used to define the vertical rays \((x_i, y_j, -1)\) to \((x_i, y_j, 1)\). Hence this results in an \(n \times n\) array of rays spanning the cuboid in a vertical direction.

Now let \((\theta, \varphi)\) vary to cover ‘all directions’ in the hemisphere above the \(xy\) plane. (This, of course, covers all directions in a hemisphere around the origin). Consider the spherical triangle with vertices \(P(0,0,1), Q(\cos \frac{4\pi}{3}, \sin \frac{4\pi}{3}),\) and \(R(1,0,0)\).

\(P, Q\) and \(R\) are on the sphere, and also form a planar triangle. The edges of the planar triangle \(PQR\) are subdivided, forming a new triangle \(pqr\) (FIGURE 26). These new vertices are projected to the sphere. The process is repeated recursively, subdividing the spherical triangles \(Ppq, pQr, pqr, qrR\), and so on to any depth of recursion. This method is simultaneously carried out on the three similar triangles...
bounding the hemisphere (Figure 27). At each level of subdivision, all the vertices obtained will be on the hemisphere and form a uniform partition.

The problem is that this method is not indexable. Given an arbitrary ray direction a recursive subdivision would be required to find the nearest ray in this representation. There is also no natural way to use this subdivision to index into the array of vectors of T-Intersections.

Considering the subdivision of the planar triangles only it is easy to see that if the level of recursion is $m$, then there will be $2^{m-1}$ vertical subdivisions, and for the $u$th such subdivision there will be $3u$ horizontal subdivisions. This is shown for two levels of recursion in Figure 27, with $2^{2-1}$ longitudinal subdivisions, and 6 on the base. (Level 1 is the original three triangles).

**FIGURE 26. Recursive Subdivision of Triangle on a Sphere**
This motivates the following scheme. For the parameter \( m \) corresponding to the number of levels of recursion in the uniform triangular subdivision, choose the \((\theta, \phi)\) range as:

\[
\begin{align*}
\phi &= \frac{\pi}{2R^u}, \quad u = 1, 2, \ldots, R \\
\theta &= \frac{2\pi}{3u}, \quad v = 0, 1, \ldots, 3u - 1
\end{align*}
\]

where \( R = 2^{m-1} \) \hfill (80)

The case \( u = 0 \) is obviously special, corresponding to a vertical ray.

Now this sampling scheme, although producing the same number of vertices on the hemisphere as the uniform subdivision, is clearly not uniform itself, although it is ‘close’ to this. Figure 28 shows a 3D plot of the scheme, looking from the origin vertically upwards, using \( m=5 \).

The ray representation is therefore of the form \([i,j,u,v]\), from which an array lookup can be constructed giving constant time access to a corresponding vector of \( T\)-Intersections. For parametric value \( m \), the total number of vertices generated is:

\[
1 + \left(\frac{3}{2}\right)2^{m-1}(2^{m-1} + 1)
\]

In order to find the nearest ray to any given ray \( r \), find the direction vector of \( r \), from which \( \theta \) and \( \phi \) can be computed, and hence the nearest \( u \) value. The nearest \( v \) value can then also be found using (80). Rotate the ray to the vertical, using rota-
tion matrix \( R_y(-\theta) R_z(-\varphi) \) (post-multiplication), and then project to the \( xy \) plane. Find the nearest \((i,j)\) grid-point using the inverse of (79).

**Rendering Objects into the VLF**

With this representation scheme the problem of rendering an object into the VLF is simple. If the object is a polygon or polyhedra, then it can be rendered into a vertical parallel subfield by using a 2-dimensional polygon fill algorithm, with \( z \)-interpolation to find the intersection points, almost identical to raster scanning a polygon in 2D and also computing the \( z \)-values. (Unlike a \( z \)-buffer, however, all \( z \)-intersections with the object are needed, not just the nearest). If this can be done for a vertical parallel subfield, then any other parallel subfield is just a rotation of the vertical, and hence can make use of this solution.

**FIGURE 28. A 3D Plot of the hemisphere sampling scheme for \( m=5 \)**

Finding all rays (approximately) through a point is also particularly simple for the parallel subfield representation. For the vertical subfield, project the point to the \( xy \) plane and find the nearest grid intersection to give \( i \) and \( j \). For each \( u,v \) combination, the corresponding \( \theta \) and \( \varphi \) values are used to rotate back to the vertical, and again project to the \( xy \) plane to find \( i \) and \( j \).

If an object is any arbitrary shape, then all rays through the bounding box of the object can be trivially found in the vertical subfield case, and then each such ray
Preliminary Error Analysis

Ray Approximation Error

A study was carried out to understand the relationship between the parameters determining the VLF resolution, and the error in ray approximations.

Recall that \( n^2 \) is the number of cells on the xy plane, and \( m \) determines the number of ray directions over the hemisphere according to (81). For values of \( n \) ranging from 16 to 1024 in steps of 32, and \( m \) ranging between 2 and 10 in steps of 1, 1000 random rays with origins in the unit sphere were generated. For each random ray the approximation ray in the VLF was computed. Each actual ray and approximation ray of course intersected the sphere at two points, and the distance between their corresponding intersection points was taken as a measure of error. The mean error was taken over each of the 1000 randomly generated rays. Also the mean of the correlations between each ray and its VLF approximation was computed. FIGURE 29. shows the relationship between \( m \) and the ray error. This graph includes the results for all values of \( n \), but notice that there is a clear relationship between the error and \( m \) independently of \( n \) (each point in the graph is an overlay of many original data points generated by the different values of \( n \) for each \( m \)). The correlation (equivalent to cos-angle) between rays is not really a good measure of error, because two rays which are exactly parallel but which are far apart have correlation 1. Nevertheless it is worth noting as FIGURE 30. shows that generally the correlations are very high, the lowest across the range of \( m \) and \( n \) values was about 0.9.
A statistical analysis of the relationship between the error, $m$ and $n$ was carried out. This found that 98% of the variation in the error could be accounted for by a simple model:

$$Error = \frac{1}{m^3} + \frac{1}{n} - 0.3 \quad (82)$$

As would be expected, $m$ dominates. No matter how fine a grid on the $xy$ plane, a small number of ray directions will make it impossible to find a good approximation to any arbitrary ray.
Number of Object Intersections

The second issue that will be important in influencing the quality of images is the number of ray-object intersections. The greater the number of intersections in general, the greater the number of ‘hits’ on the lens and ultimately on the image plane window, allowing the formation of higher resolution images. A simple argument can determine an approximation for the number of ray-object intersections. Consider a polygon with area $A$, and normal parallel to the $z$-axis of the VLF construction. Since the area of the base of the construction is 4, and the number of vertical rays emanating from the base is $n^2$, the number of rays hitting the polygon for the vertical subfield alone is approximately:

$$\frac{A}{4}n^2$$  \hspace{1cm} (83)

Now consider the polygon at some other orientation, with angle $\phi$ between its normal and the $z$-axis. Then its projected area on the $xy$ plane is approximately $(A/4)\cos \phi$, so that the approximate number of intersections for one such orientation is:

$$\frac{A}{4}n^2 \cos \phi$$  \hspace{1cm} (84)

For each value of $\phi$ there will be a number of rotations in the $\theta$ direction, but each will result in the same number of intersections given by (84). Using (80) the total number of intersections will be:

$$\frac{A}{4}n^2 \left( 1 + \sum_{u=1}^{\kappa} \cos \left( \frac{\pi u}{2R} (3\mu) \right) \right)$$  \hspace{1cm} (85)

where $R = 2^{m-1}$

By computing terms of the summation and doing a simple log-regression analysis, it is easy to show that in the range $m$ from 1 to 20, and to very good approximation:

$$\sum_{u=1}^{R} \cos \left( \frac{\pi u}{2R} (3\mu) \right) \approx 0.22 e^{1.37m}$$  \hspace{1cm} (86)

Therefore the approximate number of intersections $I(n,m,A)$ with a polygon of area $A$, using a VLF parameterized by $n$ and $m$ is:
For example, when $n=40$, $m=7$, and for a polygon with area 0.30, the true number of intersections was 387,916, and the predicted number according (87) was 386,032. In another case with $n=100$ and $m=7$, the predicted number of intersections for a polygon with area 0.36 is 2,895,238, and for two polygons of this size, the actual numbers were 2,555,458 and 2,903,294. The exactitude is less important than getting some idea of the relative impact of $m$ and $n$ in determining the accuracy and the predicted number of intersections on the lens. Of course, not all rays that hit the lens will actually be lit, and even if so will not necessarily hit the image plane window, so (87) gives more of an upper bound on the number of likely intersections.

**Results**

**Basics**

All results in reported here were obtained using a Silicon Graphics O2, with a MIPS R5000 (Rev 2.1) processor, with processor speed 180MHz, and 1GB RAM. The machine was running as a stand-alone Unix workstation. Timings were taken by in-line program calls to `time`. Since most of the times are minutes or hours, it was not thought necessary to use profiling to get a very accurate millisecond timing. All images were generated using a 200 by 200 display window, which could be partitioned into at most 200 by 200 equal rectangular cells. These 2D cells were rendered using OpenGL.

The method employs radiance computations along the ray segments. A conversion into XYZ color space, and onto RGB has not been carried out yet. The simplistic approach of sampling in RGB space has been taken, and then radiance values converted to RGB colors by a scaling technique, which allows adjustment of overall brightness\(^1\).

---

\(^1\)The brightness of the images was calibrated for the associated O2 monitor.
The scenes have objects with combined perfectly diffuse and specular reflection. Recall that the VLF space of rays is normalized to \([-1, 1]^3\) and all scenes were defined in this space.

**Does it work at all?**

The important question to ask is whether anything useful happens at all with this method - this is not to expect something that looks ‘good’, but that it looks as if some kind of acceptable light transport is happening.

Figure 31 shows a ‘sphereman’ image. This has 31 objects, a fully enclosed 6-sided room containing the ‘sphereman’ consisting of 25 spheres. The parameterization used was \(n=160\) and \(m=6\), with still 39MR. This took 2 minutes and 19 seconds to render into the VLF, and converged in 5 cycles, taking 1 hour and 51 minutes. About 170,000 rays hit the lens. The image is slightly out of focus looking towards the light. Although a small scene, in radiosity terms it would not be so small, since the objects would have to be subdivided into many fragments.

**FIGURE 31. A ‘sphereman’ with \(n=160\) and \(m=6\)**

![sphereman image](image)

Figure 32 shows the same scene generated with \(n=80\) and \(m=7\). Rendering to the VLF took 2 minutes and 26 seconds. Convergence occurred in 4 cycles, which took 1 hour and 27 minutes. The resolution is 200 by 200. The focus is near the sphereman, but most notable are the reflections in left-wall, the floor and the ceiling.
A more recent implementation has dramatically improved the images and reduced the memory requirements. Two illustrations are shown: Figure 33 shows a room with diffusely reflecting walls and a diffuse box. Figure 34 shows the same scene, but where one of the walls is specularly reflecting. The reflection of the box can clearly be seen in this wall.
Discussion

The VLF method is a different way of thinking about computer graphics. Although it naturally contains elements of many previous methods, in combination it is different to all of them. It is a light-centered paradigm rather than an object- or image-centered one. It represents the light in a scene rather than displaying the objects. It may seem strange because there is no traditional rendering pipeline, although it can make use of the rendering pipeline hardware (geometric transformations and scan-conversion) during the process of rendering polygons into the VLF.

As with any alternative paradigm, it has many major problems of its own. The first is a very large memory requirement. The scenes produced here could not have been possible without 1GB memory. Another major problem is one of scale. It is unlikely that the VLF to represent an entire building will be suitable for representing a tea-spoon within that building.

The method of course is an approximation. Some aspects of the approximation were discussed in Section Preliminary Error Analysis. However, the nature of the approximation remains to be explored. Image space rendering is an approximation in the sense that it is a 2D discrete sample of a continuum. A VLF being in a higher order space, has a higher order of error: when a ray is ‘wrong’ it might hit the wrong object. Visibility relationships might be generated that do not exist in the actual scene. Given the discussion in Section Ray Approximation Error, this is not
likely to be a major source of error, given the very close correspondence between real rays and actual rays for relatively low values of $m$. Nevertheless, this issue needs to be thoroughly explored.
Buckalew and Fussell, 1989

Camahort and Fussel, 1999

Camahort, Lerios, and Fussell, 1998

Chai, Tong, Chan, and Shum, 2000
References

Chrysanthou, Cohen-Or and Lischinski, 1998

Cohen, Shenchang, Wallace, and Greenberg, 1988

Glassner, 1995

Gersho and Gray, 1992

Gortler et al., 1996

Gu and Cohen, 1997

Ihm, Park and Lee, 1997
Isaksen, McMillan, and Gortler, 2000


Jensen, 1996


Kajiya, 1986


Levoy and Hanrahan, 1996


Miller and Rubin, 1998


Perlin, Paxia and Kollin, 2000


Schirmacher, Heidrich and Seidel, 1999

Seitz and Kutulakos, 1998


Shirley et al., 1995


Sloan, Cohen and Gortler, 1997


Walter et al., 1997


Ward Larson and Shakespear, 1998


Wood, et al., 2000