

# COMP 471 Computer Graphics

Fall 1997

## Contents

<b>1</b>	<b>Drawing a Straight Line</b>	<b>1</b>
<b>2</b>	<b>Drawing a Circle</b>	<b>4</b>
<b>3</b>	<b>Spaces and Transformations</b>	<b>7</b>
3.1	Scalar, Vector, Affine, and Euclidean Spaces . . . . .	7
3.2	Matrix Transformations . . . . .	9
<b>4</b>	<b>Viewing</b>	<b>11</b>
<b>5</b>	<b>Light and Illumination</b>	<b>13</b>
5.1	Achromatic Light . . . . .	13
5.2	Chromatic Light . . . . .	14
5.3	Illumination . . . . .	15
5.4	Multiple Light Sources . . . . .	18
5.5	Polygon Shading . . . . .	18

## 1 Drawing a Straight Line

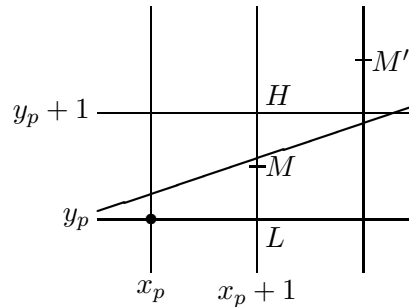
The **midpoint algorithm** scan converts a straight line using only integer addition. Bresenham (1965) had the original idea, Pitteway (1967) gave the midpoint formulation, and the version given here is due to Van Aken (1984).

The problem is to scan convert a line with end points  $(x_0, y_0)$  and  $(x_1, y_1)$ . We assume that the end points have integer coordinates (that is, they lie on the pixel grid). Let

$$\begin{aligned} dx &= x_1 - x_0 \\ dy &= y_1 - y_0 \end{aligned}$$

We assume that  $dx > 0$ ,  $dy > 0$ , and  $\frac{dy}{dx} < 1$ . (Note that  $dx$  and  $dy$  are integers, not differentials.) Since the slope is less than one, we will need a pixel at every X-ordinate. The problem is to choose the Y coordinates of the pixels.

Assume that we have plotted a pixel at  $(x_p, y_p)$ . We have two choices for the pixel at  $x_p + 1$ : it should be at either  $H$  or  $L$  in the diagram below. Suppose that  $M$  is midway between  $L$  and  $H$ . If the line passes below  $M$ , we plot pixel  $L$ ; if the line passes above  $M$  (as in the diagram), we plot pixel  $H$ .



The equation of the line is

$$y = x \frac{dy}{dx} + B \quad (1)$$

where  $B = y_0 - x_0 \frac{dy}{dx}$  is the intercept with the axis  $x = 0$  (we do not actually need  $B$  in the subsequent calculations). We can rewrite (1) as  $x dy - y dx + B dx = 0$  and we define

$$F(x, y) \equiv x dy - y dx + B dx. \quad (2)$$

If  $P$  is a point on the line, clearly  $F(P) = 0$ . We can also show (as in Assignment 1) that

$$F(P) \begin{cases} < 0, & \text{if } P \text{ is above the line;} \\ > 0, & \text{if } P \text{ is below the line.} \end{cases}$$

Consequently, we can use  $F$  to decide which pixel to plot. If  $F(M) < 0$ , then  $M$  is above the line and we plot  $L$ ; if  $F(M) > 0$ , then  $M$  is below the line and we plot  $H$ .

We can easily compute  $F(M)$ , using definition (2), as

$$F(M) = (x_p + 1) dy - (y_p + \frac{1}{2}) dx + B dx.$$

What happens next? Suppose we plot pixel  $L$ . Then the next midpoint,  $M'$ , is one step “east” of  $M$  at  $(x_p + 2, y_p + \frac{1}{2})$ . We have

$$\begin{aligned} F(M') &= F(x_p + 2, y_p + \frac{1}{2}) \\ &= (x_p + 2) dy - (y_p + \frac{1}{2}) dx + B dx \\ &= F(M) + dy. \end{aligned}$$

If, instead, we plot pixel  $H$ , the next midpoint is one step “northeast” of  $M$  at  $(x_p + 2, y_p + \frac{3}{2})$ , as in the diagram. In this case,

$$\begin{aligned} F(M') &= F(x_p + 2, y_p + \frac{3}{2}) \\ &= (x_p + 2) dy - (y_p + \frac{3}{2}) dx + B dx \\ &= F(M) + dy - dx. \end{aligned}$$

Using these results, we need to compute  $d = F(M)$  only once, during initialization. In subsequent iterations, we:

- increment  $x$ ;
- if  $d < 0$ , add  $dy$  to  $d$ ;
- else if  $d > 0$ , increment  $y$  and add  $dy - dx$  to  $d$ .

There are two points to note:

- We have implicitly dealt with the case  $F(M) = 0$  in the same way as  $F(M) > 0$ . In some situations, we might need a more careful choice.
- The algorithm still has fractions (with denominator 2). Since we need only the sign of  $F$ , not its value, we can use  $2F$  instead of  $F$ .

Figure 1 shows a simple C version of the algorithm. Remember that this handles only the case  $0 < \frac{dy}{dx} < 1$ : a complete function would have code for all cases.

```

void line (int x0, int y0, int x1, int y1)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;
    int L = 2 * dy;
    int H = 2 * (dy - dx);
    int x = x0;
    int y = y0;
    for (; x < x1; x++)
    {
        pixel(x, y);
        if (d < 0)
            d += L;
        else
        {
            d += H;
            y++;
        }
    }
    pixel(x1, y1);
}

```

Figure 1: A C function for lines with slope less than 1.

## 2 Drawing a Circle

We can use the ideas that we used to draw a straight line to draw a circle. First, we use symmetry to reduce the amount of work eightfold. Assume that the centre of the circle is at the origin  $(0, 0)$ . Then, if  $(x, y)$  is a point on the circle, then the following seven points are also on the circle:  $(-x, y)$ ,  $(x, -y)$ ,  $(-x, -y)$ ,  $(y, x)$ ,  $(-y, x)$ ,  $(y, -x)$ , and  $(-y, -x)$ .

The equation of a circle with radius  $R$  and centre at the origin is  $x^2 + y^2 = R^2$ . Let

$$F(x, y) = x^2 + y^2 - R^2.$$

Then, for any point  $P$ :

$$F(P) \begin{cases} < 0, & \text{if } P \text{ is inside the circle;} \\ = 0, & \text{if } P \text{ is on the circle; and} \\ > 0, & \text{if } P \text{ is outside the circle.} \end{cases}$$

Assume we have plotted a pixel at  $(x_p, y_p)$  (see Figure 2). The decision variable  $d$  is given by

$$\begin{aligned} d &= F(x_p + 1, y_p - \frac{1}{2}) \\ &= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2 \end{aligned}$$

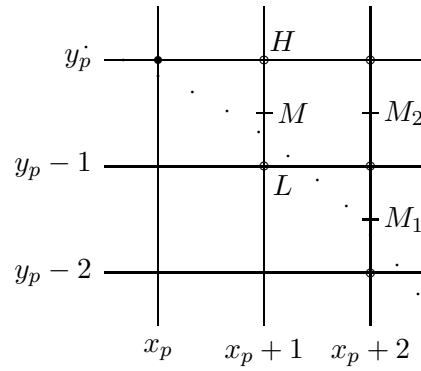


Figure 2: Drawing a circle

If  $d \geq 0$ , as in Figure 2, we plot  $L$ , the next decision point is  $M_1$ , and

$$\begin{aligned} d' &= F(x_p + 2, y_p - \frac{3}{2}) \\ &= (x_p + 2)^2 + (y_p - \frac{3}{2})^2 - R^2 \\ &= d + 2x_p - 2y_p + 5. \end{aligned}$$

If  $d > 0$ , we plot  $H$ , the next decision point is  $M_2$ , and

$$\begin{aligned} d' &= F(x_p + 2, y_p - \frac{1}{2}) \\ &= (x_p + 2)^2 + (y_p - \frac{1}{2})^2 - R^2 \\ &= d + 2x_p + 3. \end{aligned}$$

For the first pixel,  $x = 0$ ,  $y_0 = R$ , and

$$\begin{aligned} M &= (x_0 + 1, R - \frac{1}{2}) \\ &= (1, R - \frac{1}{2}), \end{aligned}$$

and

$$\begin{aligned} F(M) &= F(1, R - \frac{1}{2}) \\ &= 1^2 + (R - \frac{1}{2})^2 - R^2 \\ &= \frac{5}{4} - R. \end{aligned}$$

From this algorithm, it is straight forward to derive the code shown in Figure 3. For each coordinate computed by the algorithm, the function `circlepoints` in Figure 4 plots eight pixels at the points of symmetry.

```
void circle (int radius) {
    int x = 0;
    int y = radius;
    double d = 1.25 - radius;
    circlepoints(x, y);
    while (y > x) {
        if (d < 0)
            d += 2.0 * x + 3.0;
        else {
            d += 2.0 * (x - y) + 5.0;
            y--;
        }
        x++;
        circlepoints(x, y);
    }
}
```

Figure 3: Computing points in the first octant

```
void circlepoints (int x, int y) {
    pixel(x, y);
    pixel(-x, y);
    pixel(x, -y);
    pixel(-x, -y);
    pixel(y, x);
    pixel(-y, x);
    pixel(y, -x);
    pixel(-y, -x);
}
```

Figure 4: Plotting eight symmetrical points

### 3 Spaces and Transformations

We could build graphical models and view them in a single coordinate frame, but this would be very tedious. Instead, we work with many coordinate frames (or “reference systems”) and move between them by using transformations. In the following examples, we move from a frame  $(x, y, z)$  to a new frame  $(x', y', z')$ .

- Move the origin to  $(a, b, c)$ .

$$\begin{aligned}x' &= x - a \\y' &= y - b \\z' &= z - c\end{aligned}$$

- Rotate the frame about the  $z$  axis through an angle  $\theta$ .

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta \\z' &= z\end{aligned}$$

- Scale the axes by factors  $r$ ,  $s$ , and  $t$ .

$$\begin{aligned}x' &= r x \\y' &= s y \\z' &= t z\end{aligned}$$

If  $M = m_{ij}$  is a  $3 \times 3$  matrix, we have

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} m_{11} x + m_{12} y + m_{13} z \\ m_{21} x + m_{22} y + m_{23} z \\ m_{31} x + m_{32} y + m_{33} z \end{bmatrix}$$

which shows that we can represent a scaling or rotation, but not a translation, using a  $3 \times 3$  matrix.

The solution is to use  $4 \times 4$  matrices. We can view this as an algebraic trick, but there is a mathematical justification that we present briefly in Section 3.1. For details, consult the Appendix of *Computer Graphics: Principles and Practice*, by Foley and van Dam.

#### 3.1 Scalar, Vector, Affine, and Euclidean Spaces

**Scalar Space** An element of a scalar space is just a real number. The familiar operations (add, subtract, multiply, and divide) are defined for real numbers and they obey familiar laws such as commutation, association, distribution, etc. (In the following, we will use lower-case Greek letters  $(\alpha, \beta, \gamma, \dots)$  to denote scalars.)

**Vector Space** A vector space is a collection of vectors that obey certain laws. (We will use bold lower-case Roman letters ( $\mathbf{u}, \mathbf{v}, \mathbf{w}, \dots$ ) to denote vectors.)

We can add vectors (notation:  $\mathbf{u} + \mathbf{v}$ ); vector addition is commutative and associative. Vector subtraction (notation:  $\mathbf{u} - \mathbf{v}$ ) is the inverse of addition. For any vector  $\mathbf{v}$ , we have  $\mathbf{v} - \mathbf{v} = \mathbf{0}$ , the unique *zero vector*.

We can also multiply a vector by a scalar (notation:  $\alpha \mathbf{v}$ ). The result is a vector and we have laws such as  $\alpha(\beta \mathbf{v}) = (\alpha\beta) \mathbf{v}$ ,  $\alpha(\mathbf{u} + \mathbf{v}) = \alpha \mathbf{u} + \alpha \mathbf{v}$ , etc.

We say that a set,  $U$ , of vectors *spans* a vector space  $V$  if every vector in  $V$  can be written as in the form  $\alpha_1 \mathbf{u}_1 + \alpha_2 \mathbf{u}_2 + \dots + \alpha_n \mathbf{u}_n$  with  $\mathbf{u}_1 \in U, \dots, \mathbf{u}_n \in U$ .<sup>a</sup> The *dimension* of a vector space is the number of vectors in its smallest spanning set. We can represent every vector in a  $n$ -dimensional vector space as a tuple of  $n$  real numbers. For example, in a 3-dimensional vector space, every vector can be written as  $(x, y, z)$  where  $x, y$ , and  $z$  are real numbers.

The zero vector plays a special role in a vector space: it defines an origin. We cannot move the origin, and we cannot translate a vector. The only operations we can perform on vectors are scaling and rotation. This is just the limitation we found with  $3 \times 3$  matrices above.

**Affine Space** An affine space, like a vector space, contains scalars and vectors with the same operations and laws. An affine space also contains points (notation:  $P, Q$ , etc.). Points appear in two operations:

1. The *difference* between two points  $P$  and  $Q$  is a point. Notation:  $P - Q$ .
2. The *sum* of a point  $P$  and a vector  $\mathbf{v}$  is a vector. Notation:  $P + \mathbf{v}$ .

Given a scalar  $\alpha$  and points  $P, Q$ , the expression  $P + \alpha(Q - P)$  is called the *affine combination of the points  $P$  and  $Q$  by  $\alpha$* . Note first that the expression is well-formed:  $Q - P$  is a vector, that we can multiply by the scalar  $\alpha$  to obtain another vector that we can add to the point  $P$ .

If  $\alpha = 0$ , then  $P + \alpha(Q - P)$  simplifies to  $P + \mathbf{0} = P$ . If  $\alpha = 1$ , then  $P + \alpha(Q - P)$  simplifies to  $P + (Q - P) = Q$ . (This reduction can be derived from the axioms of an affine space, which we have not given here. Intuitively,  $Q - P$  is an arrow (vector) representing a trip from  $Q$  to  $P$ ; by adding this vector to  $P$ , we get back to  $Q$ .)

The set of points  $\{P + \alpha(Q - P) \mid \alpha \in \mathcal{R}\}$  is defined to be a *line* in affine space. If  $0 \leq \alpha \leq 1$ , then  $P + \alpha(Q - P)$  is *between*  $P$  and  $Q$ .

Continuing in this way, we can express many geometrical properties in terms of the axioms of an affine space. There is no absolute scale (like inches or centimetres) but we can represent ratios. (An *affine transformation* is a transformation that preserves certain ratios.)

**Representation of an Affine Space** We define a point to be a tuple  $(x, y, z, 1)$  and a vector to be a tuple  $(x, y, z, 0)$ , with  $x, y, z \in \mathcal{R}$ . Using component-wise addition and subtraction:

- $(x, y, z, 1) - (x', y', z', 1) = (x - x', y - y', z - z', 0)$ : the difference between two points is a vector;

- $(x, y, z, 1) + (x', y', z', 0) = (x + x', y + y', z + z', 1)$ : the sum of a point and a vector is a point; and
- $(x, y, z, 0) \pm (x', y', z', 0) = (x \pm x', y \pm y', z \pm z', 0)$ : the sum or difference of vectors is a vector.

It is possible to show formally that this representation satisfies the axioms of an affine space. (The space represented is actually a three-dimensional *affine subspace* of a four-dimensional vector space.)

The advantage of this representation is that all of the transformations that we need for graphics can be expressed as  $4 \times 4$  matrices. In particular, the translation that we could not do with  $3 \times 3$  matrices is now possible.

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + a \\ y + b \\ z + c \\ 1 \end{bmatrix}$$

In this example and below, we write points in the affine space as  $4 \times 1$  matrices (column matrices), and point is always the second (or final) term in a product. In running text, we write  $[x, y, z, 1]^T$  (transpose of a  $1 \times 4$  matrix) for a  $4 \times 1$  matrix.

**Euclidean Space** We cannot make measurements in an affine space. We introduce distance *via* inner products. The *inner product* of vectors  $\mathbf{u} = [u_x, u_y, u_z, 0]^T$  and  $\mathbf{v} = [v_x, v_y, v_z, 0]^T$  is

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z$$

The *length* of a vector  $\mathbf{v}$  is

$$\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}}$$

Consequently, the distance between two points  $P$  and  $Q$  is  $\|P - Q\|$ .

We also define the *angle*  $\alpha$  between two vectors  $\mathbf{u}$  and  $\mathbf{v}$  by:

$$\cos \alpha = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

Note that, if  $\mathbf{u} \cdot \mathbf{v} = 0$ , then  $\cos \alpha = 0$  and  $\alpha = \frac{\pi}{2}$  and the vectors  $\mathbf{u}$  and  $\mathbf{v}$  are orthogonal (or perpendicular).

A space in which we can measure distances and angles in this way is called a *Euclidean space*. Defining measurement in other ways yields non-Euclidean spaces of various kinds.

### 3.2 Matrix Transformations

We now have the machinery required to define transformations in three-dimensional Euclidean space as  $4 \times 4$  matrices. The important transformations are summarized in Figure 5. The identity transformation doesn't change anything; translation moves the origin to the point  $[-t_x, -t_y, -t_z, 1]^T$ ; scaling multiplies  $x$  values by  $s_x$ , etc; and the rotations rotate the frame counterclockwise through an angle  $\theta$  about the given axis.

$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$
Identity	Translation	Scale
$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Rotation $\theta$ about $X$ -axis	Rotation $\theta$ about $Y$ -axis	Rotation $\theta$ about $Z$ -axis

Figure 5: Basic Transformations

**Sequences of Transformations** If  $T_1$  and  $T_2$  are transformations expressed as matrices, then the matrix products  $T_1 T_2$  and  $T_2 T_1$  are also transformations. In general,  $T_1 T_2 \neq T_2 T_1$ , corresponding to the fact that transformations do not commute.

Consider a translation  $d$  along the  $X$ -axis and a rotation  $R$  about the  $Z$  axis. Assume

$$T = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then we have

$$TR = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & d \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad RT = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & d \cos \theta \\ \sin \theta & \cos \theta & 0 & d \sin \theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying these transformations to a point  $\mathbf{p} = (x, y, z, 1)$ , we obtain:

$$TR\mathbf{p} = \begin{bmatrix} x \cos \theta - y \sin \theta + d \\ x \sin \theta + y \cos \theta \\ z \\ 1 \end{bmatrix} \quad \text{and} \quad RT\mathbf{p} = \begin{bmatrix} x \cos \theta - y \sin \theta + d \cos \theta \\ x \sin \theta + y \cos \theta + d \sin \theta \\ z \\ 1 \end{bmatrix}$$

These results show that  $TR$  is a rotation *followed by* a translation and  $RT$  is a translation *followed by* a rotation. In other words, the transformations are applied in the opposite order

to their appearance in the matrix product. This is not surprising, because we can write  $TR\mathbf{p}$  as  $T(R\mathbf{p})$ .

If we view transformations as operating on the coordinate system rather than points, the sequence of operations is reversed. (Algebraically, the reversal can be expressed by the law  $(T_1T_2)^{-1} = T_2^{-1}T_1^{-1}$ .) We can think of the transformation  $TR$  as first moving to a new coordinate system translated from the original system, and then rotating the new coordinates. Similarly,  $RT$  is viewed as first rotating the coordinate system and then translating it. With this viewpoint, the sequence of matrices in a composition of transformations corresponds to the sequence of operations. It is also closer to the way that graphics software, such as OpenGL, actually works.

## 4 Viewing

In order to view a three-dimensional object on a two-dimensional screen, we must *project* it. The simplest kind of projection simply ignores one coordinate. For example, the transformation  $[x, y, z, 1]^T \mapsto (x, y)$  provides a screen position for each point by ignoring its  $z$  coordinate. A projection of this kind is called an *orthographic* or *parallel* projection.

Orthographic projections do not give a strong sensation of depth because the size of an object does not depend on its distance from the viewer. In fact, an orthographic projection simulates a view from an infinite distance away.

Perspective transformations provide a better sense of depth by drawing distant objects with reduced size. The size reduction is a natural consequence of viewing the object from a finite distance. In Figure 6,  $E$  represents the eye of the viewer viewing a screen. Conceptually, the model is behind the screen. An object at  $P$  in the model, for example, appears on the screen at the point  $P'$ .

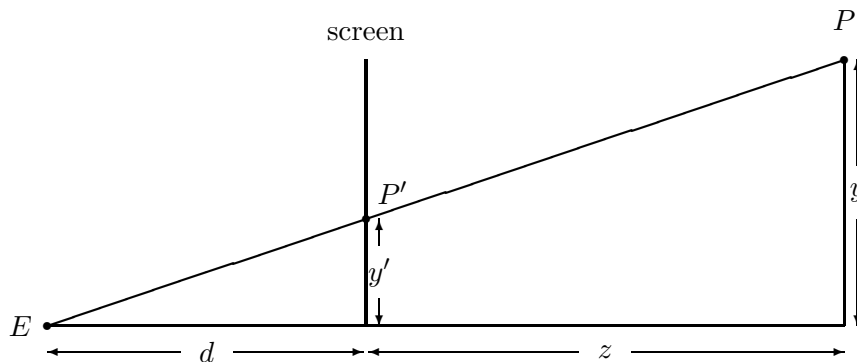


Figure 6: Perspective transformation

The point  $P$  in the model is  $[x, y, z, 1]^T$ . (The  $x$  value does not appear in the diagram because it is perpendicular to the paper.) The transformed coordinates on the screen are  $(x', y')$ : there is no  $z$  coordinate because the screen has only two dimensions. By similar triangles:

$$\frac{x'}{d} = \frac{x}{z + d}$$

$$\frac{y'}{d} = \frac{y}{z+d}$$

and hence

$$x' = \frac{x d}{z+d}$$

$$y' = \frac{y d}{z+d}$$

To obtain the perspective transformation in matrix form, we note that

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d + 1 \end{bmatrix}$$

Unlike the previous points we have seen, the transformed point has a fourth coordinate that is not equal to 1. Homogenizing this point gives  $\left[ \frac{x d}{z+d}, \frac{y d}{z+d}, \frac{z d}{z+d}, 1 \right]$ , as we obtained above.

The  $z$  component has no obvious physical significance but there are advantages in retaining. Since transformed  $z$  values depend monotonically on the original  $z$  values, the transformed values can be used for hidden surface elimination. Moreover, the transformation can be inverted only if the  $z$  information is retained. This is useful for transforming normal vectors and for selecting objects from a scene.

## 5 Light and Illumination

### 5.1 Achromatic Light

*Achromatic* means “without colour”. Although achromatic light rarely occurs in nature, it is simpler to study than coloured light. Also, there are various devices and processes that cannot handle colour, and it is important to understand how they work. Black and white (also called “monochrome”) television, black and white photographic film, monochrome computer monitors, and many printing processes operate without colour.

Achromatic light can be described by a single variable, called its *intensity*, *luminance*, or *brightness*. We will assume for simplicity that the value of this variable is between 0 (representing no light, or black) and 1 (representing the maximum possible light, or white).

For some applications, such as printing text, 0 and 1 are the only values that we require. For other applications, we need shades of grey in between black and white. Since the response of our eyes is logarithmic, we choose a minimum intensity,  $I_0$ , and a series of brightnesses with a constant ratio between each pair:

$$I_0 \quad I_0 r \quad I_0 r^2 \quad I_0 r^3 \quad \dots$$

If we have  $k$  distinct brightnesses, then  $I_0 r^{k-1} = 1$  and

$$r = \left( \frac{1}{I_0} \right)^{\frac{1}{k-1}}$$

For example, if  $I_0 = 1/50$  and  $k = 64$ , then  $r = 50^{\frac{1}{63}} \approx 1.064$ .

The intensity  $I$  of a pixel on the screen of a cathode ray tube depends on the grid voltage  $V$  as

$$I = k V^\gamma$$

where  $\gamma$  is a constant that usually has a value between 2.2 and 2.6. Comparing the two formulas for brightness, we have

$$I_0 r^j = k V^\gamma$$

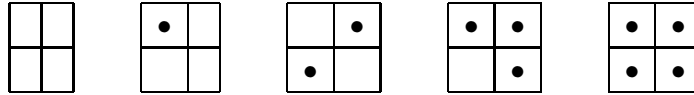
or

$$V = \left( \frac{I_0 r^j}{k} \right)^{\frac{1}{\gamma}}$$

This formula shows how the grid voltage  $V$  depends on the brightness required, as given by  $j$  where  $0 \leq j < k$ . The voltage can be generated by analog hardware or digitally using a look-up table. High-quality monitors have controls for “gamma correction”; the purpose of these controls is to adjust the conversion from input levels to grid voltages.

**Dithering** Some devices, such as laser printers, cannot print any shade other than black. We can simulate shades of grey using these devices by printing patterns of black dots with different black/white ratios. If the dots are small enough, we will see grey rather than black and white. The technique is called “clustered-dot ordered dithering” or *dithering* for short. In the printing industry (books, newspapers, etc) it is called *halftoning*.

For example, we could obtain three intermediate levels (five levels altogether) using  $2 \times 2$  patterns, as shown below. In general,  $n \times n$  patterns give  $n^2 + 1$  distinct levels.



## 5.2 Chromatic Light

Light consists of photons, each with its own wavelength. The photons of monochromatic (“one colour”) light all have the same wavelength. Laser light is monochromatic with a high degree of purity. Sodium vapour street lights (the yellow ones) emit almost monochromatic light, but their light actually consists of photons with two different wavelengths that are very close together.

Most light is a mixture of wavelengths and can be characterized by a *power spectrum*,  $P(\lambda)$ , which gives the number of photons as a function of the wavelength,  $\lambda$ . Light is just a special case of electromagnetic radiation which happens to have wavelengths between 400 nm and 700 nm. (1 nm = 1 nanometer =  $10^{-9}$  metres). Radio waves, heat, and infra-red radiation have longer wavelengths than visible light, and ultra-violet and X-rays have shorter wavelengths.

The power spectrum of sunlight corresponds roughly to that of a “black body” with a temperature of 6,000°K (degrees Kelvin, i.e., degrees above absolute zero). The peak power of the sun’s spectrum lies in the visible region, presumably because our eyes evolved to make maximum use of sunlight.

The complexity of light creates a problem for devices and processes that create or reproduce colour. The amount of information in light is enormous: encoding light in the same way that we encode audio signals (e.g., music on CDs) would require a bit rate of the order of  $10^{16}$  b/s. (This estimate is based on the fact that the frequency of light is between  $4 \times 10^{14}$  Hz (red light) and  $7.5 \times 10^{14}$  Hz (violet light).) Fortunately, we do not have to encode light in the same way as we encode audio. Our ears have sensors for about 15,000 different frequencies, but our eyes have receptors for only three different frequencies.

Suppose that light  $L_1$  stimulates the three kinds of cone receptors in our eyes with values  $(r, g, b)$ . (We can think of the receptors as responding to red, green, and blue light, although this is not strictly accurate.) Then another light  $L_2$  that provides the same  $(r, g, b)$  stimulus will look exactly the same although its power spectrum may be completely different. For example, we cannot distinguish pure yellow light from a particular mixture of green and red light.

This model of perception suggests that we could simulate all possible colours by mixing red, green, and blue light in various ways. In fact, we can achieve a fairly good approximation.

Modern colour theory is based on *CIE colours*, defined by the *Commission Internationale de l’Éclairage* in 1931. CIE light has three components,  $X$ ,  $Y$ , and  $Z$ . There are three colour-matching functions,  $\bar{x}_\lambda$ ,  $\bar{y}_\lambda$ , and  $\bar{z}_\lambda$ , defined by tables. The claim is that light with a power

spectrum  $P(\lambda)$  can be represented by components  $(X, Y, Z)$ , where

$$X = k \int P(\lambda) \bar{x}_\lambda d\lambda, \quad Y = k \int P(\lambda) \bar{y}_\lambda d\lambda, \quad Z = k \int P(\lambda) \bar{z}_\lambda d\lambda$$

where  $k$  is a constant (680 lumens/watt for a cathode ray tube) and the integration is performed over the wavelengths of visible light.

If we factor out the intensity of the light by normalizing, so that  $X + Y + Z = 1$ , the normalized factors  $(X, Y, Z)$  give the *chromaticity* of the light (in other words, its colour).

Other colour systems are related to the CIE model by linear transformations. For example, the colour monitors used for computer graphics use RGB (red-green-blue) colours. The relationship between XYZ and RGB is given by

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = M \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

where  $M$  is a  $3 \times 3$  matrix. As another example, colour television in North America uses YIQ encoding, designed to allow colour signals to be received by black and white receivers. The relationship between RGB and YIQ is

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

### 5.3 Illumination

To obtain realistic images in computer graphics, we need to know not only about light but also what happens when light is reflected from an object into our eyes. The nature of this reflection determines the appearance of the object. The general problem is to compute the apparent colour at each pixel that corresponds to part of the object on the screen.

We provide various techniques for solving this problem, increasing the realism at each step. In each case, we define the *intensity*,  $I$ , of a pixel in terms of a formula. The first few techniques ignore colour.

**Technique #1.** We assume that each object has an *intrinsic brightness*  $k_i$ . Then

$$I = k_i$$

This technique is used simple graphics, but it is clearly unsatisfactory. There is no attempt to model properties of the light or its effect on the objects.

**Technique #2.** We assume that there is *ambient light* (light from all directions) with intensity  $I_a$  and that each object has an *ambient reflection coefficient*  $k_a$ . This gives

$$I = I_a k_a$$

In practice, the effect of technique #2 is very similar to that of technique #1.

**Technique #3.** We assume that there is a single, point source of light and that the object has *diffuse* or *Lambertian* reflective properties. This means that the light reflected from the object depends only on the incidence angle of the light, not the direction of the viewer.

More precisely, suppose that:  $\mathbf{N}$  is a vector normal to the surface of the object;  $\mathbf{L}$  is a vector corresponding to the direction of the light; and  $\mathbf{V}$  is a vector corresponding to the direction of the viewer. Figure 7 shows these vectors: note that  $\mathbf{V}$  is not necessarily in the plane defined by  $\mathbf{V}$  and  $\mathbf{N}$ . Assume that all vectors are normalized ( $\|\mathbf{N}\| = \|\mathbf{L}\| = \|\mathbf{V}\| = 1$ ). Then the apparent brightness of the object is proportional to  $\mathbf{N} \cdot \mathbf{L}$ . Note that  $\mathbf{V}$  does not appear in this expression, that the brightness is greatest when  $\mathbf{N}$  and  $\mathbf{L}$  are parallel ( $\mathbf{N} \cdot \mathbf{L} = 1$ ), and that the brightness is smallest when  $\mathbf{N}$  and  $\mathbf{L}$  are orthogonal ( $\mathbf{N} \cdot \mathbf{L} = 0$ ).

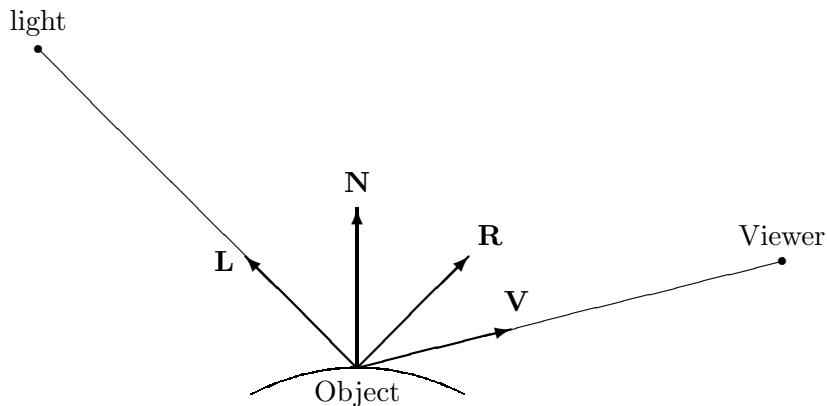


Figure 7: Illuminating an object

We can account for Lambertian reflection in the following way. Suppose that the beam of light has cross-section area  $A$  and it strikes the surface of the object at an angle  $\theta$ . Then the area illuminated is approximately  $A/\cos\theta$ . After striking the object, the light is scattered uniformly in all directions. The apparent brightness to the viewer is inversely proportional to the area illuminated, which means that it is proportional to  $\cos\theta$ , the inner product of the light vector and the surface normal.

We introduce  $I_p$ , the incident light intensity from a point source, and  $k_d$ , the *diffuse reflection coefficient* of the object. Then

$$I = I_p k_d (\mathbf{N} \cdot \mathbf{L})$$

If we include some ambient light, this equation becomes

$$I = i_a k_a + I_p k_d (\mathbf{N} \cdot \mathbf{L})$$

The value of  $\mathbf{N} \cdot \mathbf{L}$  can be negative: this will be the case if the light is underneath the surface of the object. We usually assume that such light does not contribute to the illumination of the surface. In calculations, we should use  $\min(\mathbf{N} \cdot \mathbf{L}, 0)$  to keep negative values out of our results.

**Technique #4.** Light attenuates (gets smaller) with distance from the source. The theoretical rate of attenuation for a point source of light is quadratic. In practice, sources are not

true points and there is always some ambient light from reflecting surfaces (except in outer space). Consequently, we assume that attenuation,  $f$ , is given by

$$f = \frac{1}{C + Ld + Qd^2}$$

where:

- $d$  is the distance between the light and the object;
- $C$  (constant attenuation) ensures that a close light source does not give an infinite amount of light;
- $L$  (linear term) allows for the fact that the source is not a point; and
- $Q$  (quadratic term) models the theoretical attenuation from a point source.

Then we have

$$I = i_a k_a + f I_p k_d (\mathbf{N} \cdot \mathbf{L})$$

**Technique #5.** Techniques #1 through #4 ignore colour. We assume that:

- the object has *diffuse colour factors*  $O_{dr}$ ,  $O_{dg}$ , and  $O_{db}$ ;
- the light has *intensity colour factors* corresponding to ambient sources ( $I_{ar}$ ,  $I_{ag}$ , and  $I_{ab}$ ); and
- point sources ( $I_{pr}$ ,  $I_{pg}$ , and  $I_{pb}$ ).

All of these numbers are in the range  $[0, 1]$ . We now have three intensity equations of the form

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f I_{p\lambda} k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L})$$

for  $\lambda = r, g, b$ .

**Technique #6.** Lambertian reflection is a property of dull objects such as cloth or chalk. Many objects exhibit degrees of shininess: polished wood has some shininess and a mirror is the ultimate in shininess. The technical name for shininess is *specular reflection*. A characteristic feature of specular reflection is that it has a colour closer to the colour of the light source than the colour of the object. For example, a brown dresser made of polished wood that is illuminated by a white light will have specular highlights that are white, not brown.

Specular reflection depends on the direction of the viewer as well as the light. We introduce a new vector,  $\mathbf{R}$  (the *reflection vector*), which is the direction in which the light would be reflected if the object was a mirror (see Figure 7). The brightness of specular reflection depends on the angle between  $\mathbf{R}$  and  $\mathbf{V}$  (the angle of the viewer). For *Phong shading* (developed by Phong Bui-Tong), we assume that the brightness is proportional to  $(\mathbf{R} \cdot \mathbf{V})^n$ , where  $n = 1$  corresponds to a slightly glossy surface and  $n = \infty$  corresponds to a perfect mirror. We now have

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f I_{p\lambda} (k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L}) + k_s (\mathbf{R} \cdot \mathbf{V})^n)$$

where  $k_s$  is the *specular reflection coefficient* and  $n$  is the *specular reflection exponent*.

**Technique #7.** In practice, the colour of specular reflection is not completely independent of the colour of the object. To allow for this, we can give the object a *specular colour*  $O_{s\lambda}$ . Then we have

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f I_{p\lambda} (k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L}) + k_s O_{s\lambda} (\mathbf{R} \cdot \mathbf{V})^\sigma)$$

This equation represents our final technique for lighting an object and is a close approximation to what OpenGL actually does. The actual calculation performed by OpenGL is:

$$V_\lambda = O_{e\lambda} + M_{a\lambda} O_{a\lambda} + \sum_{i=0}^{n-1} \left( \frac{1}{k_c + k_l d + k_q d^2} \right)_i s_i [I_{a\lambda} O_{a\lambda} + (\mathbf{N} \cdot \mathbf{L}) I_{d\lambda} O_{d\lambda} + (\mathbf{R} \cdot \mathbf{V})^\sigma I_{s\lambda} O_{s\lambda}]_i$$

where

$V_\lambda$	=	Vertex brightness
$M_{a\lambda}$	=	Ambient light model
$k_c$	=	Constant attenuation coefficient
$k_l$	=	Linear attenuation coefficient
$k_q$	=	Quadratic attenuation coefficient
$d$	=	Distance of light source from vertex
$s_i$	=	Spotlight effect
$I_{a\lambda}$	=	Ambient light
$I_{d\lambda}$	=	Diffuse light
$I_{s\lambda}$	=	Specular light
$O_{e\lambda}$	=	Emissive brightness of material
$O_{a\lambda}$	=	Ambient brightness of material
$O_{d\lambda}$	=	Diffuse brightness of material
$O_{s\lambda}$	=	Specular brightness of material
$\sigma$	=	Shininess of material

the subscript  $\lambda$  indicates colour components and the subscript  $i$  denotes one of the lights.

## 5.4 Multiple Light Sources

If there are several light sources, we simply add their contributions. If the sum of the contributions exceeds 1, we can either “clamp” the value (that is, use 1 instead of the actual result) or reduce all values in proportion so that the greatest value is 1. Clamping is cheaper computationally and usually sufficient.

## 5.5 Polygon Shading

The objects in graphical models are usually defined as many small polygons, typically triangles or rectangles. We must choose a suitable colour for each visible pixel of a polygon: this is called *polygon shading*.

**Flat Shading** In flat shading, we compute a vector normal to the polygon and use it to compute the colour for every pixel of the polygon. The computation implicitly assumes that:

- the polygon is really flat (not an approximation to a curved surface);
- $\mathbf{N} \cdot \mathbf{L}$  is constant (the light source is infinitely far away); and
- $\mathbf{N} \cdot \mathbf{V}$  is constant (the viewer is infinitely far away.)

Flat shading is efficient computationally but not very satisfactory: the edges of the polygons tend to be visible and we see a polyhedron rather than the surface we are trying to approximate. (The edges are even more visible than we might expect, due to the *Mach effect*, which exaggerates a change of colour along a line.)

**Smooth Shading** In smooth shading, we compute normals at the vertices of the polygons, averaging over the polygons that meet at the vertex. If we are using polygons to approximate a smooth surface, these vectors approximate the true surface normals at the vertices. We compute the colour at each vertex and then colour the polygons by *interpolating* the colours at interior pixels.

**Gouraud Shading** Gouraud shading is a form of smooth shading that uses a particular kind of interpolation for efficiency.

1. Compute the normal at each vertex of the polygon mesh. For analytical surfaces, such as spheres and cones, we can compute the normals exactly. For surfaces approximated by polygons, we use the average of the surface normals at each vertex.
2. Compute the light intensity for each colour at each vertex using a *lighting model* (e.g., Technique #7 above).
3. Interpolate intensities along the edges of the polygons.
4. Interpolate intensities along scan lines within the polygons.

**Phong Shading** Phong shading is similar to Gouraud shading but interpolates the normals rather than the intensities. Phong shading requires more computation than Gouraud shading but gives better results, especially for specular highlights.