10

Geometric Algorithms

Do not disturb my circles!

—Archimedes (287–212 B.C.E.)

Geometry needs no introduction. This most visual branch of mathematics is used whenever you see pictures on your computer screen, and in this chapter we'll explore algorithms useful for tasks such as these:

Web image maps
: How can you tell whether the mouse click fell within an oddly shaped area? See the section "Inclusion."

Arranging windows
: How do you open up new windows so that they obscure existing windows as little as possible? See the section "Boundaries."

Cartography
: You have a set of scattered points (say, fenceposts) and want to draw the region that they define. See the section "Boundaries."

Simulations
: Which pair of 10,000 points are closest to each other and therefore in danger of colliding? See the section "Closest Pair of Points."

In this chapter, we explore geometric formulas and algorithms. We can only provide building blocks for you to improve upon; as usual, we can't anticipate every use you'll have for these techniques. We'll restrict ourselves to two dimensions in almost all of the code we show. We don't cover the advanced topics you'll find in a book devoted solely to computer graphics, such as ray tracing, radiosity, lighting, animation, or texture mapping, although we do cover splines in the section "Splines" in Chapter 16, *Numerical Analysis*. For deeper coverage of these topics, we recommend *Computer Graphics: Principles and Practice*, by Foley, van Dam,

Feiner, and Hughes, and the *Graphics Gems* books. Those of a more practical persuasion will find information about windowing toolkits, business graphs, OpenGL (a 3-D graphics language) and VRML (Virtual Reality Markup Language) at the end of the chapter.

For simplicity, almost all the subroutines in this chapter accept coordinates as flat lists of numbers. To interface with your existing programs, you might want to rewrite them so that you can pass in your points, lines, and polygons as array or hash references. If you have a lot of them, this will be faster as well. See the section "References" in Chapter 1, *Introduction*, for more information.

One last caveat: Many geometric problems have nasty special cases that require special attention. For example, many algorithms don't work for concave objects, in which case you'll need to chop them into convex pieces before applying the algorithms. Complicated objects like people, trees, and class F/X intergalactic dreadnoughts fighting huge tentacled monsters from Orion's Belt are frequently represented as polygons (typically triangles, or tetrahedrons for three dimensions), and collisions with them are checked using *bounding boxes convex hulls*. More about these later in the chapter.

Distance

One of the most basic geometric concepts is *distance*: the amount of space between two objects.

Euclidean Distance

There are many ways to define the distance between two points; the most intuitive and common definition is *Euclidean distance*: the straight-line distance, as the crow flies.* In mathematical terms, we compute the differences along each axis, sum the squares of the differences, and take the square root of that sum. For two dimensions, this is the familiar *Pythagorean theorem*: $d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$.† Figure 10-1 illustrates the Euclidean distance in different dimensions. The last two cases are mere projections onto the printed page; the last one doubly so.

We can compute the Euclidean distance of any dimension with a single subroutine, as follows:

```
# distance( @p ) computes the Euclidean distance between two
# d-dimensional points, given 2 * d coordinates.  For example, a pair of
# 3-D points should be provided as ( $x0, $y0, $z0, $x1, $y1, $z1 ).
```

* Euclid: fl. 370 B.C.E.

† Pythagoras 570–490 B.C.E.

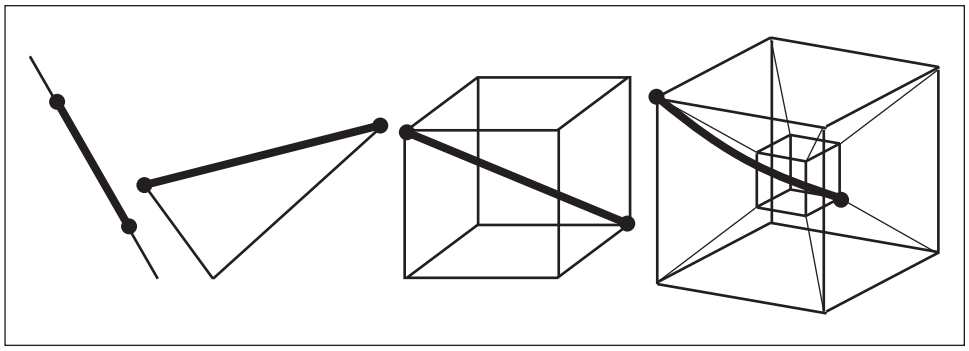

Figure 10-1. Euclidean distance in 1, 2, 3, and 4 dimensions

```
sub distance {
    my @p = @_;                 # The coordinates of the points.
    my $d = @p / 2;             # The number of dimensions.

    # The case of two dimensions is optimized.
    return sqrt( ($_[0] - $_[2])**2 + ($_[1] - $_[3])**2 )
        if $d == 2;

    my $S = 0;                  # The sum of the squares.
    my @p0 = splice @p, 0, $d;  # The starting point.

    for ( my $i = 0; $i < $d; $i++ ) {
        my $di = $p0[ $i ] - $p[ $i ];  # Difference...
        $S += $di * $di;                # ...squared and summed.
    }

    return sqrt( $S );
}
```

The Euclidean distance between the points (3, 4) and (10,12) is this:

```
print distance( 3,4, 10,12 );
10.6301458127346
```

Manhattan Distance

Another distance metric is the *Manhattan distance*, depicted in Figure 10-2. This name reflects the rigid rectangular grid on which most of Manhattan's streets are arranged; good New York cabbies routinely think in terms of Manhattan distance. Helicopter pilots are more familiar with Euclidean distance.

Instead of squaring the differences between points, we sum their absolute values:

```
# manhattan_distance( @p )
#   Computes the Manhattan distance between
#   two d-dimensional points, given 2*d coordinates.  For example,
#   a pair of 3-D points should be provided as @p of
#   ( $x0, $y0, $z0, $x1, $y1, $z1 ).
```

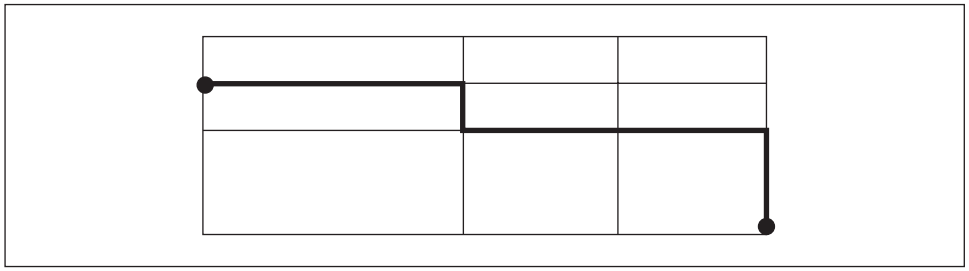

Figure 10-2. Manhattan distance

```
sub manhattan_distance {
    my @p = @_;                 # The coordinates of the points.
    my $d = @p / 2;             # The number of dimensions.

    my $S = 0;                  # The sum of the squares.
    my @p0 = splice @p, 0, $d;  # Extract the starting point.

    for ( my $i = 0; $i < $d; $i++ ) {
        my $di = $p0[ $i ] - $p[ $i ];  # Difference...
        $S += abs $di;                  # ...absolute value summed.
    }

    return $S;
}
```

For example, here is the Manhattan distance between (3, 4) and (10, 12):

```
print manhattan_distance( 3, 4, 10, 12 );
15
```

Maximum Distance

Sometimes the distance is best defined simply as the maximum coordinate difference: $d = \max d_i$, where d_i is the ith coordinate difference.

If you think of the Manhattan distance as a degree-one approximation of the distance (because the coordinate differences are raised to the power of 1), then the Euclidean distance is a degree-two approximation. The limit of that sequence is the maximum distance:

$$\sqrt[k]{\sum_{k=1}^{\infty} d_i^k}$$

In other words, as k increases, the largest difference increasingly dominates, and at infinity it completely dominates.

Spherical Distance

The shortest possible distance on a spherical surface is called the *great circle distance*. Deriving the exact formula is good exercise in trigonometry, but the programmer in a hurry can use the `great_circle_distance()` function in the Math::Trig module bundled with Perl 5.005_03 and higher. You'll find Math::Trig in earlier versions of Perl, but they won't have `great_circle_distance()`. Here's how you'd compute the approximate distance between London (51.3° N, 0.5° W) and Tokyo (35.7° N, 139.8° E) in kilometers:

```
#!/usr/bin/perl

use Math::Trig qw(great_circle_distance deg2rad);

# Notice the 90 minus latitude: phi zero is at the North Pole.
@london = (deg2rad(-  0.5), deg2rad(90 - 51.3));
@tokyo  = (deg2rad( 139.8), deg2rad(90 - 35.7));

# 6378 is the equatorial radius of the Earth, in kilometers.
print great_circle_distance(@london, @tokyo, 6378);
```

The result is:

```
9605.26637021388
```

We subtract the latitude from 90 because `great_circle_distance()` uses *azimuthal spherical coordinates*: $\phi = 0$ points up from the North Pole, whereas on Earth it points outward from the Equator. Thus we need to tilt the coordinates by 90 degrees. (See the Math::Trig documentation for more information.)

The result is far from exact, because the Earth is not a perfect sphere and because at these latitudes 0.1 degrees is about 8 km, or 5 miles.

Area, Perimeter, and Volume

Once we feel proficient with distance, we can start walking over areas and perimeters, and diving into volumes.

Triangle

The area of the triangle can be computed with several formulas, depending on what parts of the triangle are known. In Figure 10-3, we present one of the oldest, *Heron's formula*.*

* Heron lived around 65–120.

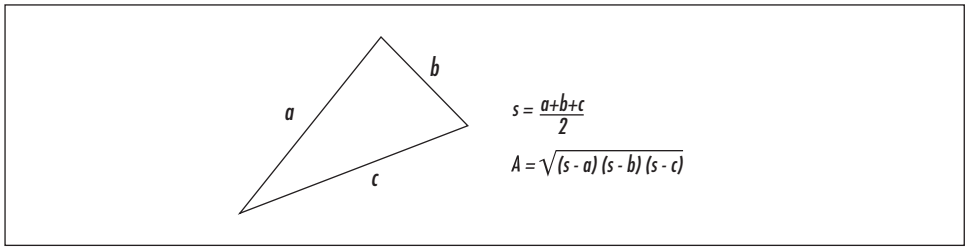

Figure 10-3. Heron's formula computes the area of a triangle given the lengths of the sides

Our code to implement Heron's formula can accept either the side lengths of the triangle, or its vertices—in which case `triangle_area_heron()` computes the side lengths using the Euclidean distance:

```
#!/usr/bin/perl

# triangle_area_heron( $length_of_side,
#                      $length_of_other_side,
#                      $length_of_yet_another_side )
#   Or, if given six arguments, they are the three (x,y)
#   coordinate pairs of the corners.
# Returns the area of the triangle.

sub triangle_area_heron {
    my ( $a, $b, $c );

    if ( @_ == 3 ) { ( $a, $b, $c ) = @_ }
    elsif ( @_ == 6 ) {
        ( $a, $b, $c ) = ( distance( $_[0], $_[1], $_[2], $_[3] ),
                           distance( $_[2], $_[3], $_[4], $_[5] ),
                           distance( $_[4], $_[5], $_[0], $_[1] ) );
    }

    my $s = ( $a + $b + $c ) / 2;               # The semiperimeter.
    return sqrt( $s * ( $s - $a ) * ( $s - $b ) * ( $s - $c ) );
}

print triangle_area_heron(3, 4, 5), " ",
      triangle_area_heron( 0, 1,   1, 0,   2, 3 ), "\n";
```

This prints:

```
6 2
```

Polygon Area

The area of a convex polygon (one that doesn't "bend inwards") can be computed by slicing the polygon into triangles and then summing their areas, as shown in Figure 10-4.

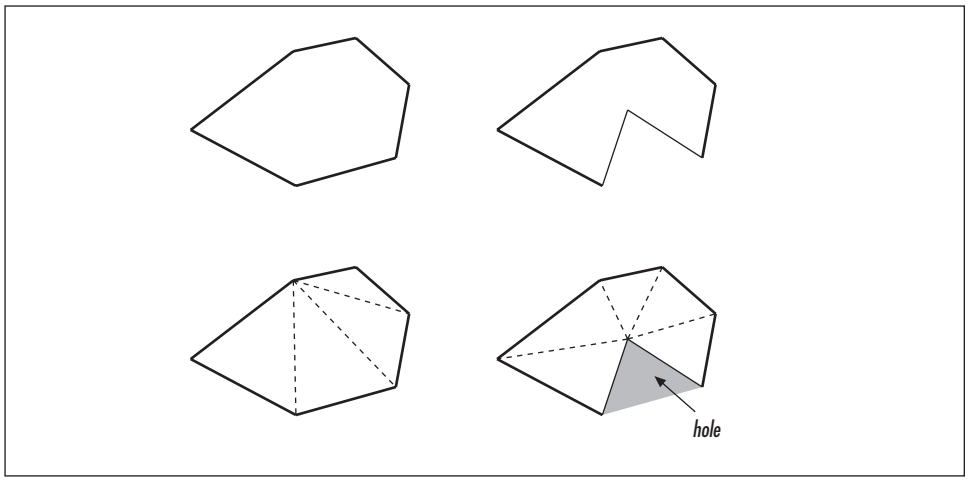

Figure 10-4. Concave and convex polygons, sliced into triangles

For concave polygons, the situation is messier: we have to ignore the "holes." A much easier way is to use determinants (see the section "Computing the Determinant" in Chapter 7, *Matrices*), as shown in Figure 10-5 and the following equation:

$$A = \frac{1}{2}\left(\begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \end{vmatrix} + \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} + \ldots + \begin{vmatrix} x_{n-1} & y_{n-1} \\ x_0 & y_0 \end{vmatrix}\right)$$

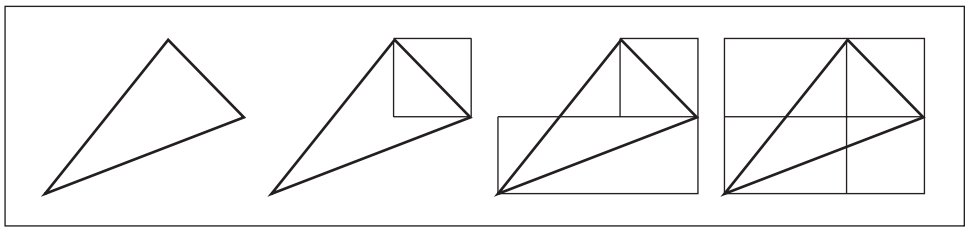

Figure 10-5. How the determinants yield the area

Each determinant yields the area of the rectangle defined by two of the polygon vertices. Since each edge of the polygon bisects the rectangle, we want to halve each area. The overlap of rectangles (the lower left in Figure 10-5) can be ignored because they conveniently cancel one another.

Notice how the formula wraps around from the last point, (x_{n-1}, y_{n-1}), back to the first point, (x_0, y_0). This is natural; after all, we want to traverse all n edges of the polygon, and therefore we had better sum exactly n determinants. We just need the determinant of a 2×2 matrix, which is simply:

```
# determinant( $x0, $y0, $x1, $y1 )
#   Computes the determinant given the four elements of a matrix
#   as arguments.
#
sub determinant { $_[0] * $_[3] - $_[1] * $_[2] }
```

Armed with the determinant, we're ready to find the polygon area:

```
# polygon_area( @xy )

#   Compute the area of a polygon using determinants.  The points
#   are supplied as ( $x0, $y0, $x1, $y1, $x2, $y2, ....)
#

sub polygon_area {
    my @xy = @_;

    my $A = 0;                          # The area.

    # Instead of wrapping the loop at its end
    # wrap it right from the beginning: the [-2, -1] below.
    for ( my ( $xa, $ya ) = @xy[ -2, -1 ];
          my ( $xb, $yb ) = splice @xy, 0, 2;
          ( $xa, $ya ) = ( $xb, $yb ) ) { # On to the next point.
        $A += determinant( $xa, $ya, $xb, $yb );
    }

    # If the points were listed in counterclockwise order, $A
    # will be negative here, so we take the absolute value.

    return abs $A / 2;
}
```

For example, we can find the area of the pentagon defined by the five points (0, 1), (1, 0), (3, 2), (2, 3), and (0, 2) as follows:

```
print polygon_area( 0, 1,  1, 0,  3, 2,  2, 3,  0, 2 ), "\n";
```

The result:

```
2
```

Note that the points *must* be listed in clockwise or counterclockwise order; see the section "Direction" for more about what that means. If you list them in another order, you're describing a different polygon:

```
print polygon_area( 0, 1, 1, 0, 2, 0, 3, 2, 2, 3 ), "\n";
```

Moving the last point to the middle yields a different result:

```
1
```

Polygon Perimeter

The same loop used to compute the polygon area can be used to compute the polygon perimeter. Now we just sum the lengths instead of the determinants:

```
# polygon_perimeter( @xy )

#   Compute the perimeter length of a polygon.  The points
#   are supplied as ( $x0, $y0, $x1, $y1, $x2, $y2, ....)
#

sub polygon_perimeter {
    my @xy = @_;

    my $P = 0;                          # The perimeter length.

    # Instead of wrapping the loop at its end
    # wrap it right from the beginning: the [-2, -1] below.
    for ( my ( $xa, $ya ) = @xy[ -2, -1 ];
          my ( $xb, $yb ) = splice @xy, 0, 2;
          ( $xa, $ya ) = ( $xb, $yb ) ) { # On to the next point.
        $P += distance( $xa, $ya, $xb, $yb );
    }

    return $P;
}
```

We can find the perimeter of the pentagon from the last example as follows:

```
print polygon_perimeter( 0, 1,  1, 0,  3, 2,  2, 3,  0, 2 ), "\n";
```

The result:

```
8.89292222699217
```

Direction

We need to know which objects are right of us (*clockwise*) or left of us (*counterclockwise*), this is useful, for example, in finding out whether a point is inside a triangle or not. We'll restrict ourselves to two dimensions in our discussion; in three dimensions the meaning of "left" and "right" is ambiguous without knowing which way is up.

Given any three points, you can specify whether they follow a clockwise path, a counterclockwise path, or neither. In Figure 10-6, the points at (1, 1), (4, 3), and (4, 4) specify a counterclockwise path: the path turns left. The points (1, 1), (4, 3), and (7, 4), specify a clockwise path: the path turns right.

The `clockwise()` subroutine accepts three points, and returns a single number: positive if the path traversing all three points is clockwise, negative if it's counterclockwise, and a number very close to 0 if they're neither clockwise nor counterclockwise—that is, all on the same line.

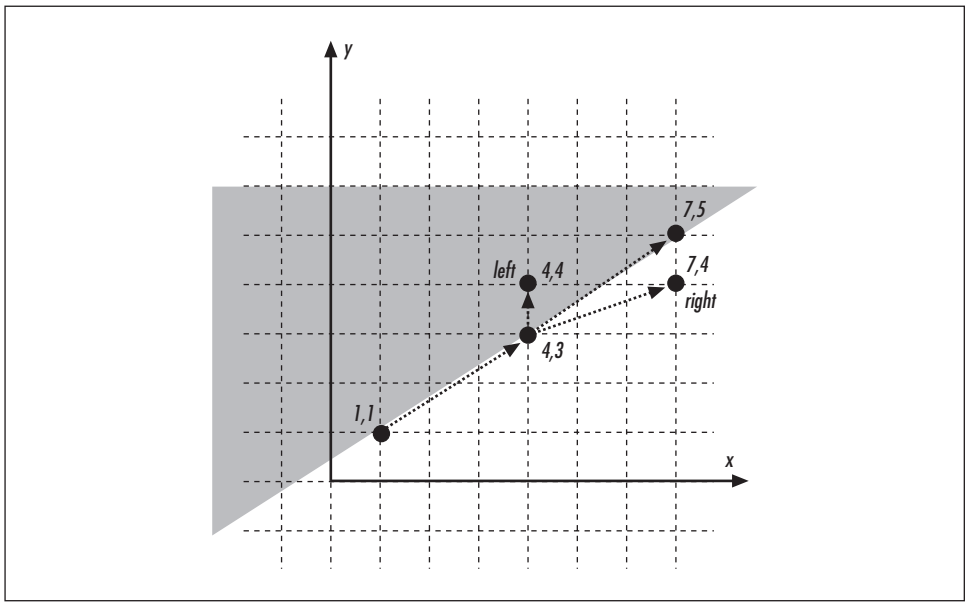

Figure 10-6. Clockwise and counterclockwise: right and left

```
# clockwise( $x0, $y0, $x1, $y1, $x2, $y2 )
#    Return positive if one must turn clockwise (right) when moving
#    from p0 (x0, y0) to p1 to p2, negative if counterclockwise (left).
#    It returns zero if the three points lie on the same line --
#    but beware of floating point errors.
#
sub clockwise {
    my ( $x0, $y0, $x1, $y1, $x2, $y2 ) = @_;
    return ( $x2 - $x0 ) * ( $y1 - $y0 ) - ( $x1 - $x0 ) * ( $y2 - $y0 );
}
```

For example:

```
print clockwise( 1, 1,   4, 3,   4, 4 ), "\n";
print clockwise( 1, 1,   4, 3,   7, 5 ), "\n";
print clockwise( 1, 1,   4, 3,   7, 4 ), "\n";
```

will output:

```
-3
0
3
```

In other words, the point (4, 4) is left (negative) of the vector from (1, 1) to (4, 3), the point (7, 5) is *on* (zero) the same vector, and the point (7, 4) is right (positive) of the same vector.

`clockwise()` is actually a flattened two-dimensional version of the *cross product* of vector algebra. The cross product is a three-dimensional object, pointing away from the plane defined by the vectors $p_0 - p_1$ and $p_1 - p_2$.

Intersection

In this section, we'll make frequent use of `epsilon()` for our floating point computations. Epsilon is for you to decide; we recommend one ten-billionth:

```
sub epsilon () { 1E-10 }
```

or the faster version:

```
use constant epsilon => 1E-10;
```

See the section "Precision" in Chapter 11, *Number Systems*, for more information.

Line Intersection

There are two flavors of line intersection. In the general case, the lines may be of any slope. In the more restricted case, the lines are confined to horizontal and vertical slopes, and these are called *Manhattan intersections*.

Line intersection: the general case

Finding the intersection of two lines is as simple as finding out when the two lines $y_0 = b_0x + a_0$ and $y_1 = b_1x + a_1$ cross, and the techniques in the section "Gaussian Elimination" in Chapter 7 and the section "Solving Equations" in Chapter 16 can find the answer for us. But those general techniques won't always work: if we are to avoid divide-by-zero errors, we need to look out for situations in which either line is horizontal or vertical, or when the lines are parallel. Figure 10-7 illustrates some different line intersections.

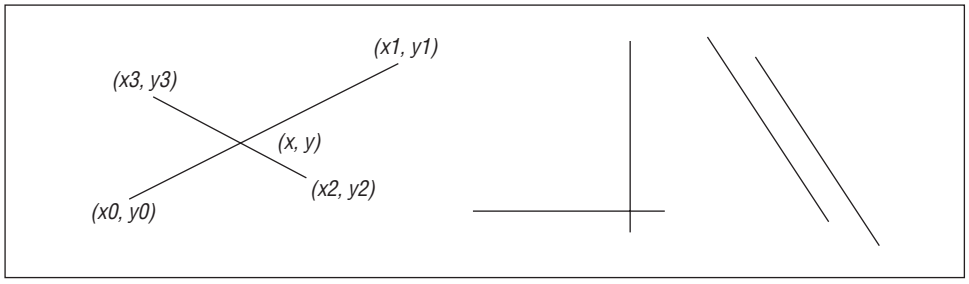

Figure 10-7. Line intersection: general case, horizontal and vertical cases, parallel case

With all the special cases, line intersection isn't as straightforward as it might seem. Our implementation is surprisingly long:

```
# line_intersection( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#
#    Compute the intersection point of the line segments
#    (x0,y0)-(x1,y1) and (x2,y2)-(x3,y3).
#
#    Or, if given four arguments, they should be the slopes of the
#    two lines and their crossing points at the y-axis.  That is,
#    if you express both lines as y = ax+b, you should provide the
#    two 'a's and then the two 'b's.
#
#    line_intersection() returns either a triplet ($x, $y, $s) for the
#    intersection point, where $x and $y are the coordinates, and $s
#    is true when the line segments cross and false when the line
#    segments don't cross (but their extrapolated lines would).
#
#    Otherwise, it's a string describing a non-intersecting situation:
#      "out of bounding box"
#      "parallel"
#      "parallel collinear"
#      "parallel horizontal"
#      "parallel vertical"
#    Because of the bounding box checks, the cases "parallel horizontal"
#    and "parallel vertical" never actually happen.  (Bounding boxes
#    are discussed later in the chapter.)
#
sub line_intersection {
 my ( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 );

 if ( @_ == 8 ) {
     ( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 ) = @_;

     # The bounding boxes chop the lines into line segments.
     # bounding_box() is defined later in this chapter.
     my @box_a = bounding_box( 2, $x0, $y0, $x1, $y1 );
     my @box_b = bounding_box( 2, $x2, $y2, $x3, $y3 );

     # Take this test away and the line segments are
     # turned into lines going from infinite to another.
     # bounding_box_intersect() defined later in this chapter.
     return "out of bounding box"
         unless bounding_box_intersect( 2, @box_a, @box_b );
 } elsif ( @_ == 4 ) { # The parametric form.
     $x0 = $x2 = 0;
     ( $y0, $y2 ) = @_[ 1, 3 ];
     # Need to multiply by 'enough' to get 'far enough'.
     my $abs_y0 = abs $y0;
     my $abs_y2 = abs $y2;
     my $enough = 10 * ( $abs_y0 > $abs_y2 ? $abs_y0 : $abs_y2 );
     $x1 = $x3 = $enough;
     $y1 = $_[0] * $x1 + $y0;
     $y3 = $_[2] * $x2 + $y2;
```

```
}

my ($x, $y);  # The as-yet-undetermined intersection point.

my $dy10 = $y1 - $y0; # dyPQ, dxPQ are the coordinate differences
my $dx10 = $x1 - $x0; # between the points P and Q.
my $dy32 = $y3 - $y2;
my $dx32 = $x3 - $x2;

my $dy10z = abs( $dy10 ) < epsilon; # Is the difference $dy10 "zero"?
my $dx10z = abs( $dx10 ) < epsilon;
my $dy32z = abs( $dy32 ) < epsilon;
my $dx32z = abs( $dx32 ) < epsilon;

my $dyx10;                          # The slopes.
my $dyx32;

$dyx10 = $dy10 / $dx10 unless $dx10z;
$dyx32 = $dy32 / $dx32 unless $dx32z;

# Now we know all differences and the slopes;
# we can detect horizontal/vertical special cases.
# E.g., slope = 0 means a horizontal line.

unless ( defined $dyx10 or defined $dyx32 ) {
    return "parallel vertical";
} elsif ( $dy10z and not $dy32z ) { # First line horizontal.
    $y = $y0;
    $x = $x2 + ( $y - $y2 ) * $dx32 / $dy32;
} elsif ( not $dy10z and $dy32z ) { # Second line horizontal.
    $y = $y2;
    $x = $x0 + ( $y - $y0 ) * $dx10 / $dy10;
} elsif ( $dx10z and not $dx32z ) { # First line vertical.
    $x = $x0;
    $y = $y2 + $dyx32 * ( $x - $x2 );
} elsif ( not $dx10z and $dx32z ) { # Second line vertical.
    $x = $x2;
    $y = $y0 + $dyx10 * ( $x - $x0 );
} elsif ( abs( $dyx10 - $dyx32 ) < epsilon ) {
    # The slopes are suspiciously close to each other.
    # Either we have parallel collinear or just parallel lines.

    # The bounding box checks have already weeded the cases
    # "parallel horizontal" and "parallel vertical" away.

    my $ya = $y0 - $dyx10 * $x0;
    my $yb = $y2 - $dyx32 * $x2;

    return "parallel collinear" if abs( $ya - $yb ) < epsilon;
    return "parallel";
} else {
    # None of the special cases matched.
    # We have a "honest" line intersection.
```

```
    $x = ($y2 - $y0 + $dyx10*$x0 - $dyx32*$x2)/($dyx10 - $dyx32);
    $y = $y0 + $dyx10 * ($x - $x0);
}

my $h10 = $dx10 ? ($x - $x0) / $dx10 : ($dy10 ? ($y - $y0) / $dy10 : 1);
my $h32 = $dx32 ? ($x - $x2) / $dx32 : ($dy32 ? ($y - $y2) / $dy32 : 1);

return ($x, $y, $h10 >= 0 && $h10 <= 1 && $h32 >= 0 && $h32 <= 1);
}
```

Figure 10-8 shows a collection of lines, illustrating the different ways they can (and cannot) intersect.

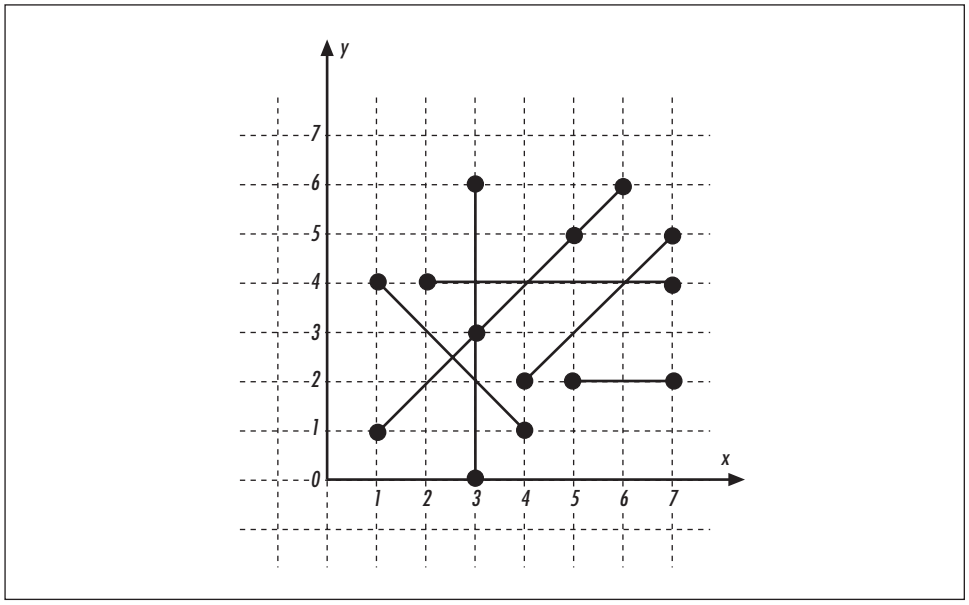

Figure 10-8. Line intersection example

We'll examine six potential intersections with `line_intersection()`:

```
print "@{[line_intersection( 1, 1,  5, 5,  1, 4,  4, 1 )]}\n";
print "@{[line_intersection( 1, 1,  5, 5,  2, 4,  7, 4 )]}\n";
print "@{[line_intersection( 1, 1,  5, 5,  3, 0,  3, 6 )]}\n";
print "@{[line_intersection( 1, 1,  5, 5,  5, 2,  7, 2 )]}\n";
print     line_intersection( 1, 1,  5, 5,  4, 2,  7, 5 ), "\n";
print     line_intersection( 1, 1,  5, 5,  3, 3,  6, 6 ), "\n";
```

The results:

```
2.5 2.5 1
4 4 1
3 3 1
2 2
```

```
parallel
parallel collinear
```

Finding the exact point of intersection is too much work if all we care about is *whether* two lines intersect at all. The intersection, if any, can be found by examining the signs of the two cross products $(p_2 - p_0) \times (p_1 - p_0)$ and $(p_3 - p_0) \times (p_1 - p_0)$. The `line_intersect()` subroutine returns a simple true or false value indicating whether two lines intersect:

```
# line_intersect( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#    Returns true if the two lines defined by these points intersect.
#    In borderline cases, it relies on epsilon to decide.

sub line_intersect {
    my ( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 ) = @_;

    my @box_a = bounding_box( 2, $x0, $y0, $x1, $y1 );
    my @box_b = bounding_box( 2, $x2, $y2, $x3, $y3 );

    # If even the bounding boxes do not intersect, give up right now.

    return 0 unless bounding_box_intersect( 2, @box_a, @box_b );

    # If the signs of the two determinants (absolute values or lengths
    # of the cross products, actually) are different, the lines
    # intersect.

    my $dx10 = $x1 - $x0;
    my $dy10 = $y1 - $y0;

    my $det_a = determinant( $x2 - $x0, $y2 - $y0, $dx10, $dy10 );
    my $det_b = determinant( $x3 - $x0, $y3 - $y0, $dx10, $dy10 );

    return 1 if $det_a < 0 and $det_b > 0 or
                $det_a > 0 and $det_b < 0;

    if ( abs( $det_a ) < epsilon ) {
        if ( abs( $det_b ) < epsilon ) {
            # Both cross products are "zero".
            return 1;
        } elsif ( abs( $x3 - $x2 ) < epsilon and
                  abs( $y3 - $y2 ) < epsilon ) {
            # The other cross product is "zero" and
            # the other vector (from (x2,y2) to (x3,y3))
            # is also "zero".
            return 1;
        }
    } elsif ( abs( $det_b < epsilon ) ) {
        # The other cross product is "zero" and
        # the other vector is also "zero".
        return 1 if abs( $dx10 ) < epsilon and abs( $dy10 ) < epsilon;
    }
```

```
    return 0; # Default is no intersection.
}
```

We'll test `line_intersect()` with two pairs of lines. The first pair intersects at (3, 4), and the second pair of lines do not intersect at all because they're parallel:

```
print "Intersection\n"
    if     line_intersect( 3, 0,  3, 6,  1, 1,  6, 6 );
print "No intersection\n"
    unless line_intersect( 1, 1,  6, 6,  4, 2,  7, 5 );
Intersection
No intersection
```

Line intersection: the horizontal-vertical case

Often, the general case of line intersection is *too* general: if the lines obey Manhattan geometry, that is, if they're strictly horizontal or vertical, a very different solution for finding the intersections is available.

The solution is to use *binary trees*, which were introduced in Chapter 3, *Advanced Data Structures*. We will slide a horizontal line from bottom to top over our plane, constructing a binary tree of lines as we do so. The resulting binary tree contains vertical lines sorted on their *x*-coordinate; for this reason, the tree is called an *x-tree*. The *x*-tree is constructed as follows:

- The points will be processed from bottom to top, vertical lines before horizontal ones, and from left to right. This means that both endpoints of a horizontal line will be seen simultaneously, while the endpoints of a vertical line will be seen separately.
- Whenever the lower endpoint of a vertical line is seen, that node is added to the binary tree, with its *x*-coordinate as the value. This divides the points in the tree in a left-right manner: if line *a* is left of line *b*, node *a* will be left of node *b* in the tree.
- Whenever the upper endpoint of a vertical line is seen, the corresponding node is deleted from the binary tree.
- Whenever a horizontal line is encountered, the nodes in the tree (the active vertical lines) are checked to determine whether any of them intersect the horizontal line. The horizontal lines are not added to the tree; their only duty is to trigger the intersection checks.

Figure 10-9 shows how an *x*-tree develops as the imaginary line proceeds from the bottom of the picture to the top. The left picture simply identifies the order in which line segments are encountered: first *c*, then *e*, and so on. The middle picture shows the *x*-tree just after *e* is encountered, and the right picture after *a* and *d* are encountered. Note that *d* is not added to the tree; it serves only to trigger an intersection check.

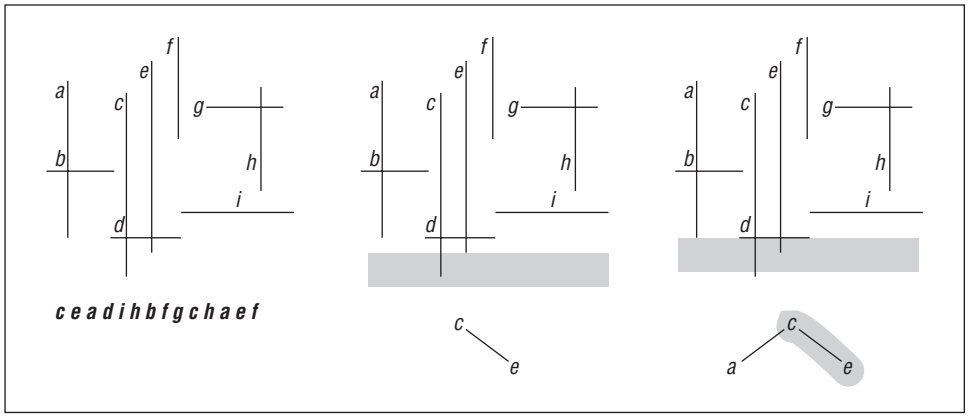

Figure 10-9. Horizontal-vertical line intersection

The `manhattan_intersection()` subroutine implements this algorithm:

```
# manhattan_intersection( @lines )
#   Find the intersections of strictly horizontal and vertical lines.
#   Requires basic_tree_add(), basic_tree_del(), and basic_tree_find(),
#   all defined in Chapter 3, Advanced Data Structures.
#
sub manhattan_intersection {
    my @op; # The coordinates are transformed here as operations.

    while (@_) {
        my @line = splice @_, 0, 4;

        if ($line[1] == $line[3]) {     # Horizontal.
            push @op, [ @line, \&range_check_tree ];
        } else {                        # Vertical.
            # Swap if upside down.
            @line = @line[0, 3, 2, 1] if $line[1] > $line[3];

            push @op, [ @line[0, 1, 2, 1], \&basic_tree_add ];
            push @op, [ @line[0, 3, 2, 3], \&basic_tree_del ];
        }
    }

    my $x_tree; # The range check tree.
    # The x coordinate comparison routine.
    my $compare_x = sub { $_[0]->[0] <=> $_[1]->[0] };
    my @intersect; # The intersections.

    foreach my $op (sort { $a->[1] <=> $b->[1] ||
                           $a->[4] == \&range_check_tree ||
                           $a->[0] <=> $b->[0] }
                           @op) {
        if ($op->[4] == \&range_check_tree) {
            push @intersect, $op->[4]->( \$x_tree, $op, $compare_x );
```

```
            } else { # Add or delete.
                $op->[4]->( \$x_tree, $op, $compare_x );
            }
        }

        return @intersect;
    }

    # range_check_tree( $tree_link, $horizontal, $compare )

    #    Returns the list of tree nodes that are within the limits
    #    $horizontal->[0] and $horizontal->[1].  Depends on the binary
    #    trees of Chapter 3, Advanced Data Structures.
    #
    sub range_check_tree {
        my ( $tree, $horizontal, $compare ) = @_;

        my @range          = ( );   # The return value.
        my $node           = $$tree;
        my $vertical_x     = $node->{val};
        my $horizontal_lo  = [ $horizontal->[ 0 ] ];
        my $horizontal_hi  = [ $horizontal->[ 1 ] ];

        return unless defined $$tree;

        push @range, range_check_tree( \$node->{left}, $horizontal, $compare )
            if defined $node->{left};

        push @range, $vertical_x->[ 0 ], $horizontal->[ 1 ]
            if $compare->( $horizontal_lo, $horizontal ) <= 0 &&
               $compare->( $horizontal_hi, $horizontal ) >= 0;

        push @range, range_check_tree( \$node->{right}, $horizontal,
                                       $compare )
            if defined $node->{right};

        return @range;
    }
```

`manhattan_intersection()` runs in $O(N \log N + k)$, where k is the number of intersections (which can be no more than $(N/2)^2$).

We'll demonstrate `manhattan_intersection()` with the lines in Figure 10-10.

The lines in Figure 10-10 are stored in an array and tested for intersections as follows:

```
@lines = ( 1, 6,  1, 3,  1, 2,  3, 2,  1, 1,  4, 1,
           2, 4,  7, 4,  3, 0,  3, 6,  4, 3,  4, 7,
           5, 7,  5, 4,  5, 2,  7, 2 );

print join(" ", manhattan_intersection(@lines)), "\n";
```

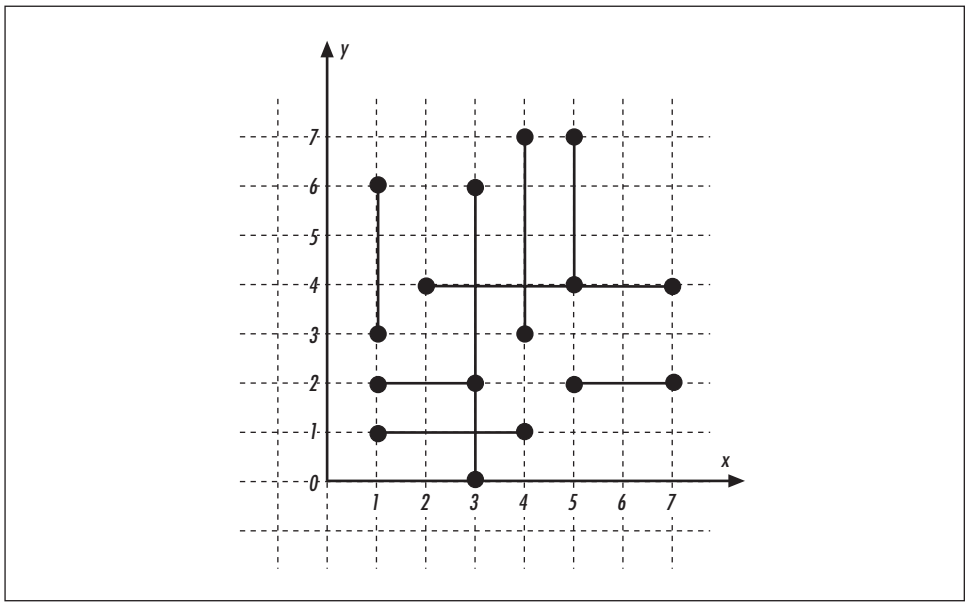

Figure 10-10. Lines for the example of Manhattan algorithm

We get:

```
3 1 3 2 1 4 3 4 4 4 5 4
```

This tells you the six points of intersection. For example, (3, 1) is the bottommost intersection, and (5, 4) is the upper-rightmost intersection.

Inclusion

In this section, we are interested in whether a point is *inside* a polygon. Once we know that, we can conduct more sophisticated operations, such as determining whether a line is partially or completely inside a polygon.

Point in Polygon

Determining whether a point is inside a polygon is a matter of casting a "ray" from the point to "infinity" (any point known to be outside the polygon). The algorithm is simple: count the number of times the ray crosses the polygon edges. If the crossing happens an odd number of times (points *e*, *f*, *h*, and *j* in Figure 10-11), we are inside the polygon; otherwise, we are outside (*a*, *b*, *c*, *d*, *g*, and *i*). There are some tricky special cases (rare is the geometric algorithm without caveats): What if the ray crosses a polygon vertex? (points *d*, *f*, *g*, and *j*) Or worse, an edge? (point *j*) The algorithm we are going to use is guaranteed to return true for

truly inside points and false for truly outside points. For the borderline cases, it depends on how the "glancing blows" are counted.

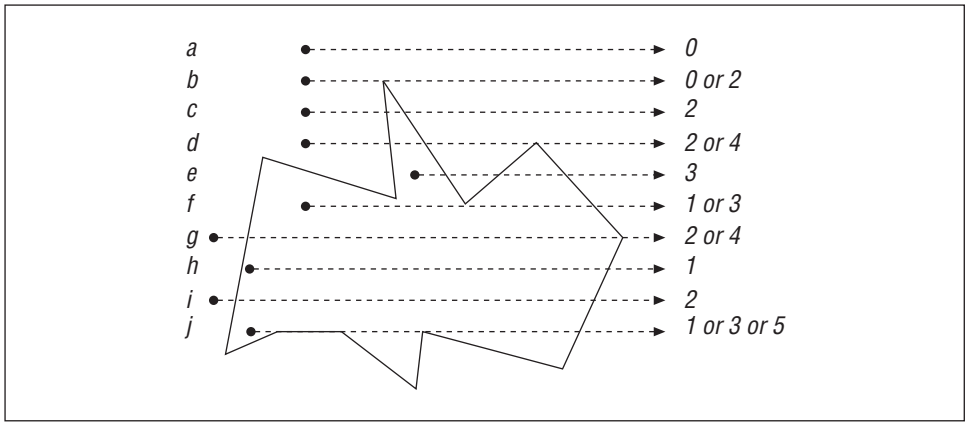

Figure 10-11. Is the point in the polygon? Count the edge crossings

The `point_in_polygon()` subroutine returns a true value if the given point (the first two arguments) is inside the polygon (described by the subsequent arguments).

```
# point_in_polygon ( $x, $y, @xy )
#
#    Point ($x,$y), polygon ($x0, $y0, $x1, $y1, ...) in @xy.
#    Returns 1 for strictly interior points, 0 for strictly exterior
#    points. For the boundary points the situation is more complex and
#    beyond the scope of this book.  The boundary points are
#    exact, however: if a plane is divided into several polygons, any
#    given point belongs to exactly one polygon.
#
#    Derived from the comp.graphics.algorithms FAQ,
#    courtesy of Wm. Randolph Franklin.
#
sub point_in_polygon {
    my ( $x, $y, @xy ) = @_;

    my $n = @xy / 2;                      # Number of points in polygon.
    my @i = map { 2 * $_ } 0 .. (@xy/2);  # The even indices of @xy.
    my @x = map { $xy[ $_ ]     } @i;     # Even indices: x-coordinates.
    my @y = map { $xy[ $_ + 1 ] } @i;     # Odd indices: y-coordinates.

    my ( $i, $j );                        # Indices.

    my $side = 0;                         # 0 = outside, 1 = inside.

    for ( $i = 0, $j = $n - 1 ; $i < $n; $j = $i++ ) {
        if (
            (
```

```
            # If the y is between the (y-) borders ...
            ( ( $y[ $i ] <= $y ) && ( $y < $y[ $j ] ) ) ||
            ( ( $y[ $j ] <= $y ) && ( $y < $y[ $i ] ) )
           )
           and
           # ...the (x,y) to infinity line crosses the edge
           # from the ith point to the jth point...
           ($x
            <
            ( $x[ $j ] - $x[ $i ] ) *
            ( $y - $y[ $i ] ) / ( $y[ $j ] - $y[ $i ] ) + $x[ $i ] )) {
         $side = not $side; # Jump the fence.
      }
    }

    return $side ? 1 : 0;
}
```

To detect whether the number of intersections is even or odd, we don't actually need to count them. We can do something much faster: simply toggle the Boolean variable `$side`.

Using the polygon in Figure 10-12, we can test whether the nine points are inside or outside as follows:

```
@polygon = ( 1, 1,  3, 5,  6, 2,  9, 6,  10, 0,  4,2,  5, -2);
print "( 3, 4): ", point_in_polygon( 3, 4, @polygon ), "\n";
print "( 3, 1): ", point_in_polygon( 3, 1, @polygon ), "\n";
print "( 3,-2): ", point_in_polygon( 3,-2, @polygon ), "\n";
print "( 5, 4): ", point_in_polygon( 5, 4, @polygon ), "\n";
print "( 5, 1): ", point_in_polygon( 5, 1, @polygon ), "\n";
print "( 5,-2): ", point_in_polygon( 5,-2, @polygon ), "\n";
print "( 7, 4): ", point_in_polygon( 7, 4, @polygon ), "\n";
print "( 7, 1): ", point_in_polygon( 7, 1, @polygon ), "\n";
print "( 7,-2): ", point_in_polygon( 7,-2, @polygon ), "\n";
```

The results:

```
( 3, 4): 1
( 3, 1): 1
( 3,-2): 0
( 5, 4): 0
( 5, 1): 0
( 5,-2): 0
( 7, 4): 0
( 7, 1): 1
( 7,-2): 0
```

This tells us that that the points (3, 4), (3, 1), and (7, 1) are inside the polygon, and the rest are outside.

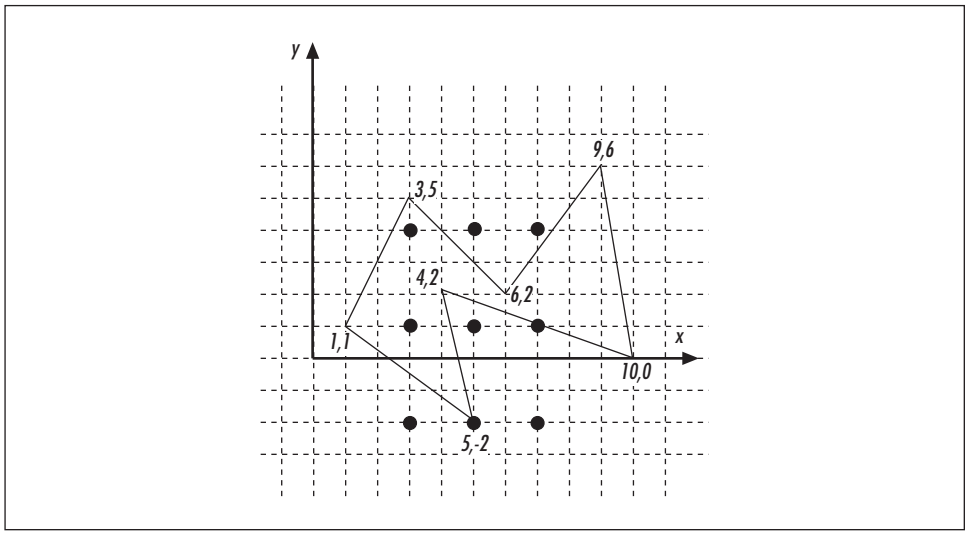

Figure 10-12. A sample polygon with some inside and outside points

Point in Triangle

For simple polygons such as triangles, we can use an alternative algorithm. We start from a corner of the triangle and determine whether we have to look to the left or right to see the point. Then we travel to the next corner and look at the point. If the side we had to look to changed, we know that the point cannot be within the triangle. We visit the final corner and check again; if the side still hasn't changed, we can safely conclude that the point is inside the triangle. Also, if we detect that the point is on an edge, we can immediately return true.

In Figure 10-13, we can envision traveling counterclockwise around the vertices of the triangle. Any point inside the triangle will be to our left. If the point is outside the triangle, we'll notice a change from left to right.

This algorithm is implemented in the `point_in_triangle()` subroutine:

```
# point_in_triangle( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2 ) returns
# true if the point ($x,$y) is inside the triangle defined by
# the following points.

sub point_in_triangle {
    my ( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2 ) = @_;

    # clockwise() from earlier in the chapter.
    my $cw0 = clockwise( $x0, $y0, $x1, $y1, $x, $y );
    return 1 if abs( $cw0 ) < epsilon; # On 1st edge.

    my $cw1 = clockwise( $x1, $y1, $x2, $y2, $x, $y );
```

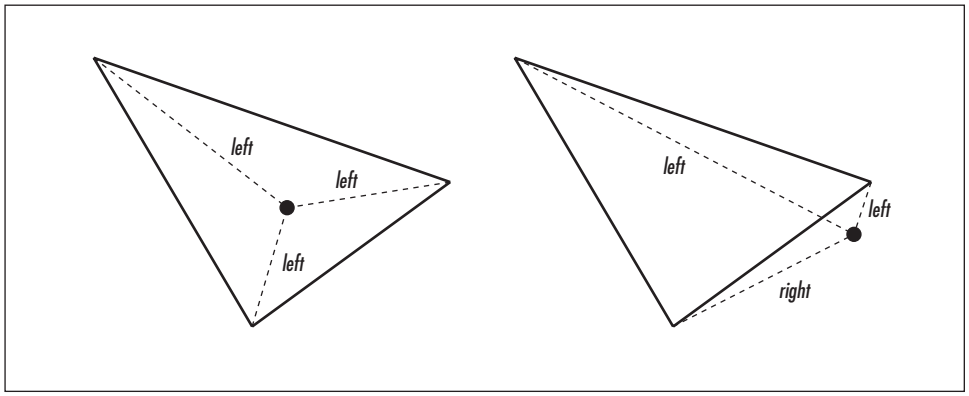

Figure 10-13. Determining whether a point is inside a triangle

```
    return 1 if abs( $cw1 ) < epsilon; # On 2nd edge.

    # Fail if the sign changed.
    return 0 if ( $cw0 < 0 and $cw1 > 0 ) or ( $cw0 > 0 and $cw1 < 0 );

    my $cw2 = clockwise( $x2, $y2, $x0, $y0, $x, $y );
    return 1 if abs( $cw2 ) < epsilon; # On 3rd edge.

    # Fail if the sign changed.
    return 0 if ( $cw0 < 0 and $cw2 > 0 ) or ( $cw0 > 0 and $cw2 < 0 );

    # Jubilate!
    return 1;
}
```

Let's define a triangle with vertices at (1, 1), (5, 6), and (9, 3), and test seven points for inclusion:

```
@triangle = ( 1, 1,  5, 6,  9, 3 );
print "(1, 1): ", point_in_triangle( 1, 1,  @triangle ), "\n";
print "(1, 2): ", point_in_triangle( 1, 2,  @triangle ), "\n";
print "(3, 2): ", point_in_triangle( 3, 2,  @triangle ), "\n";
print "(3, 3): ", point_in_triangle( 3, 3,  @triangle ), "\n";
print "(3, 4): ", point_in_triangle( 3, 4,  @triangle ), "\n";
print "(5, 1): ", point_in_triangle( 5, 1,  @triangle ), "\n";
print "(5, 2): ", point_in_triangle( 5, 2,  @triangle ), "\n";
```

The output:

```
(1, 1): 1
(1, 2): 0
(3, 2): 1
(3, 3): 1
(3, 4): 0
(5, 1): 0
(5, 2): 1
```

This tells us that the points (1, 2), (3, 4), and (5, 1) are outside the triangle and the rest are inside.

Point in Quadrangle

Any convex *quadrangle* (a four-sided polygon—all squares and rectangles are quadrangles) can be split into two triangles along any two opposing points. We can combine this observation with the `point_in_triangle()` subroutine to determine whether a point is in the quadrangle. (Beware of degenerate quadrangles: quadrangles that have overlapping corner points so that they reduce to triangles, lines, or even points.) A split of a quadrangle into two triangles is illustrated in Figure 10-14.

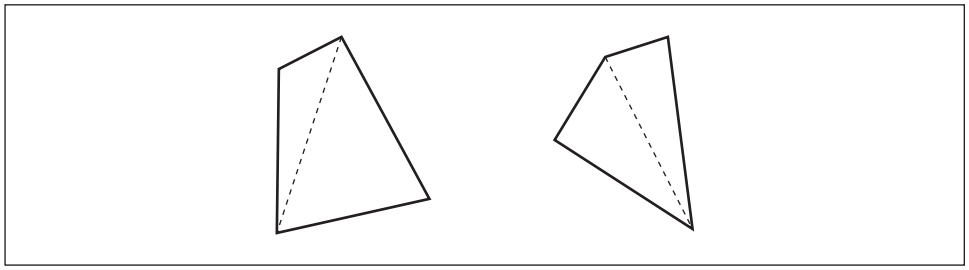

Figure 10-14. Splitting a quadrangle into two triangles

The `point_in_quadarangle()` subroutine simply calls `point_in_triangle()` twice, one for each triangle resulting from the split:

```
# point_in_quadrangle( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#   Return true if the point ($x,$y) is inside the quadrangle
#   defined by the points p0 ($x0,$y0), p1, p2, and p3.
#   Simply uses point_in_triangle.
#

sub point_in_quadrangle {
    my ( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 ) = @_;

    return point_in_triangle( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2 ) ||
           point_in_triangle( $x, $y, $x0, $y0, $x2, $y2, $x3, $y3 )
}
```

`point_in_quadrangle()` will be demonstrated with the quadrangle and points shown in Figure 10-15.

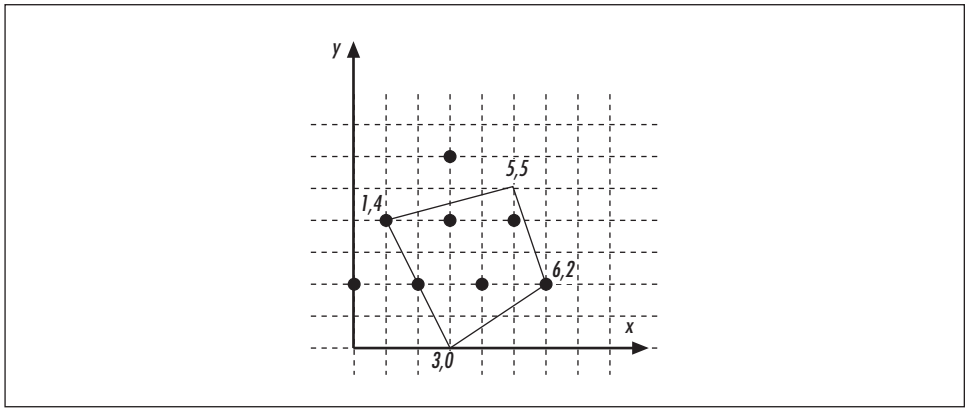

Figure 10-15. Determining whether a point is in a quadrangle

The quadrangle's vertices are at (1, 4), (3, 0), (6, 2), and (5, 5), so that's what we'll provide:

```
@quadrangle = ( 1, 4,  3, 0,  6, 2,  5, 5 );
print "(0, 2): ", point_in_quadrangle( 0, 2,  @quadrangle ), "\n";
print "(1, 4): ", point_in_quadrangle( 1, 4,  @quadrangle ), "\n";
print "(2, 2): ", point_in_quadrangle( 2, 2,  @quadrangle ), "\n";
print "(3, 6): ", point_in_quadrangle( 3, 6,  @quadrangle ), "\n";
print "(3, 4): ", point_in_quadrangle( 3, 4,  @quadrangle ), "\n";
print "(4, 2): ", point_in_quadrangle( 4, 2,  @quadrangle ), "\n";
print "(5, 4): ", point_in_quadrangle( 5, 4,  @quadrangle ), "\n";
```

The output:

```
(0, 2): 0
(1, 4): 1
(2, 2): 1
(3, 6): 0
(3, 4): 1
(4, 2): 1
(5, 4): 1
(6, 2): 1
```

This means that the points (0, 2) and (3, 6) are outside the quadrangle and the rest are inside.

Boundaries

In this section, we explore the boundaries of geometric objects, which we can use to determine whether objects seem to overlap. We say "seem" because these boundaries give only the first approximation: concave objects confuse the issue.

Bounding Box

The *bounding box* of a geometric object is defined as the smallest *d*-dimensional box containing the *d*-dimensional object where the sides align with the axes. The bounding box can be used in video games to determine whether objects just collided. Three bounding boxes are shown in Figure 10-16.

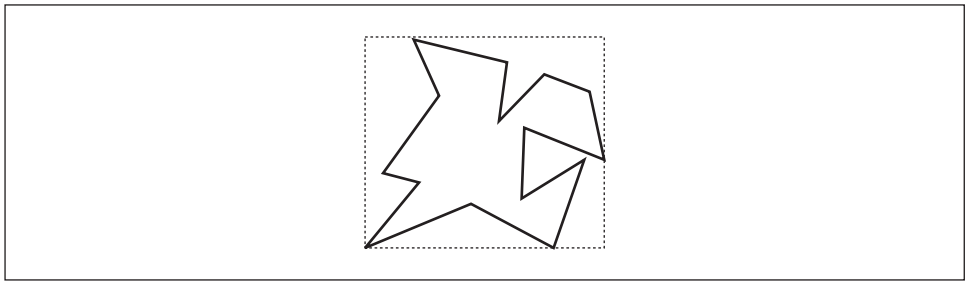

Figure 10-16. A polygon and its bounding box (dotted line)

The `bounding_box()` subroutine returns an array of points. For `$d = 2` dimensions, the bounding box will be a rectangle, and so `bounding_box()` returns four elements: two corners of the rectangle.

```
# bounding_box_of_points($d, @p)
#   Return the bounding box of the set of $d-dimensional points @p.

sub bounding_box_of_points {
    my ($d, @points) = @_;

    my @bb;

    while (my @p = splice @points, 0, $d) {
        @bb = bounding_box($d, @p, @bb); # Defined below.
    }

    return @bb;
}

# bounding_box($d, @p [,@b])
#   Return the bounding box of the points @p in $d dimensions.
#   The @b is an optional initial bounding box: we can use this
#   to create a cumulative bounding box that includes boxes found
#   by earlier runs of the subroutine (this feature is used by
#   bounding_box_of_points()).
#
#   The bounding box is returned as a list.  The first $d elements
#   are the minimum coordinates, the last $d elements are the
#   maximum coordinates.
```

```
sub bounding_box {
    my ( $d, @bb ) = @_; # $d is the number of dimensions.
    # Extract the points, leave the bounding box.
    my @p = splice( @bb, 0, @bb - 2 * $d );

    @bb = ( @p, @p ) unless @bb;

    # Scan each coordinate and remember the extrema.
    for ( my $i = 0; $i < $d; $i++ ) {
        for ( my $j = 0; $j < @p; $j += $d ) {
            my $ij = $i + $j;
            # The minima.
            $bb[ $i      ] = $p[ $ij ] if $p[ $ij ] < $bb[ $i      ];
            # The maxima.
            $bb[ $i + $d ] = $p[ $ij ] if $p[ $ij ] > $bb[ $i + $d ];
        }
    }

    return @bb;
}

# bounding_box_intersect($d, @a, @b)
#   Return true if the given bounding boxes @a and @b intersect
#   in $d dimensions.  Used by line_intersection().

sub bounding_box_intersect {
    my ( $d, @bb ) = @_; # Number of dimensions and box coordinates.
    my @aa = splice( @bb, 0, 2 * $d ); # The first box.
    # (@bb is the second one.)

    # Must intersect in all dimensions.
    for ( my $i_min = 0; $i_min < $d; $i_min++ ) {
        my $i_max = $i_min + $d; # The index for the maximum.
        return 0 if ( $aa[ $i_max ] + epsilon ) < $bb[ $i_min ];
        return 0 if ( $bb[ $i_max ] + epsilon ) < $aa[ $i_min ];
    }

    return 1;
}
```

To demonstrate, we'll find the bounding box of the polygon in Figure 10-17. We pass `bounding_box_of_points()` 21 arguments: the dimension 2 and the 10 pairs of coordinates describing the 10 points in Figure 10-17:

```
@bb = bounding_box_of_points(2,
                             1, 2,  5, 4,  3, 5,  2, 3,  1, 7,
                             2, 5,  5, 7,  7, 4,  5, 5,  6, 1), "\n";
print "@bb\n";
```

The result is the lower-left and upper-right vertices of the square, (1, 1) and (7, 7):

```
1 1 7 7
```

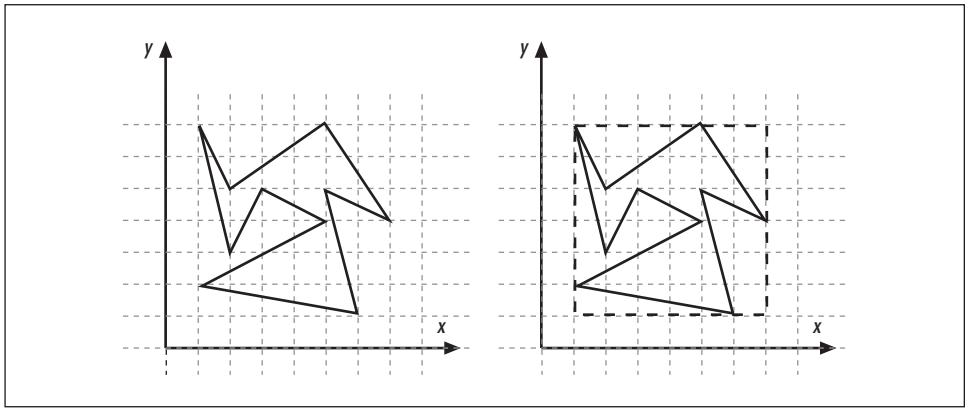

Figure 10-17. A polygon and its bounding box

Convex Hull

A *convex hull* is like a bounding box that fits even more closely because it doesn't have to be a box at all. The convex hull is stretched along the outermost possible points, like a rubber band around a collection of nails hammered into a board. (Imagine you're Christo, trying to plastic-wrap a forest. The plastic wrap forms a convex hull.)

In two dimensions, the convex hull is the set of edges of some convex polygon. In three dimensions, the convex hull is the set of sides of a convex polyhedron, all of whose sides are triangular. A two-dimensional convex hull is shown in Figure 10-18.

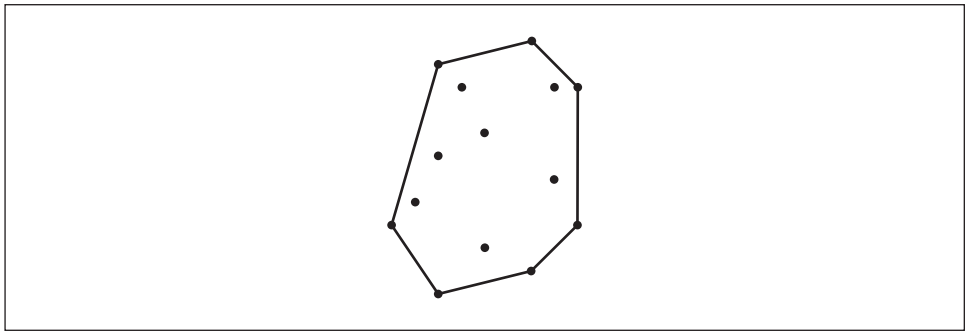

Figure 10-18. The convex hull of a point set

The most well-known algorithm for finding the convex hull in two dimensions is *Graham's scan*. It begins by finding one point known for a fact to lie on the hull,

typically the point having the smallest *x*-coordinate or the point having the smallest *y*-coordinate. This is demonstrated in Figure 10-19(a).

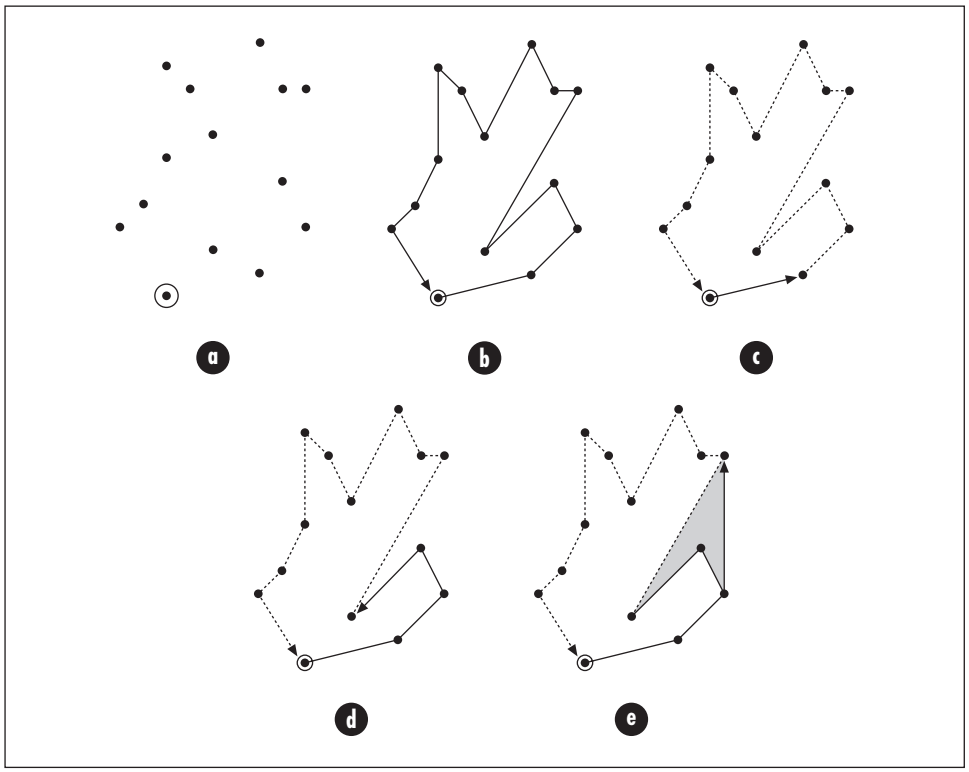

Figure 10-19. Graham's scan: find a starting point

All the other points are then sorted according to the angle they make with the starting point, illustrated in Figure 10-19(b). Because of how we chose the starting point, the angles are guaranteed to be between 0 and π radians.

The initial hull then starts from the minimum point and goes to the first of these sorted points. A complication develops when the next point is directly ahead along the hull. This can be taken care of by more intricate sorting: if the angles are equal, we sort on the *x*- and *y*-coordinates.

Now we look for the next point: whenever we must turn left to go to the next point, we add that next point to the hull.

If, however, we must turn right, the point we just added to the hull cannot be in the hull and must be removed. This removal may escalate backwards until we again turn left. This growing and shrinking of the hull suggests the use of a stack (described in the section "Stacks" in Chapter 2, *Basic Data Structures*).

As you can see, the "no-right-turns" policy backs away from concavities (shaded in Figure 10-19(e)), leaving only the convex hull. The above process is continued in the angle order until we return to the starting point. The Graham's scan algorithm is calculated by `convex_hull_graham()`:

```
# convex_hull_graham( @xy )
#   Compute the convex hull of the points @xy using the Graham's scan.
#   Returns the convex hull points as a list of ($x,$y,...).

sub convex_hull_graham {
    my ( @xy ) = @_;

    my $n = @xy / 2;
    my @i = map { 2 * $_ } 0 .. ( $#xy / 2 ); # The even indices.
    my @x = map { $xy[ $_ ]     } @i;
    my @y = map { $xy[ $_ + 1 ] } @i;

    # First find the smallest y that has the smallest x.

    # $ymin is the smallest y so far, @xmini holds the indices
    # of the smallest y(s) so far, $xmini will the index of the
    # smallest x, $xmin the smallest x.
    my ( $ymin, $xmini, $xmin, $i );

    for ( $ymin = $ymax = $y[ 0 ], $i = 1; $i < $n; $i++ ) {
        if ( $y[ $i ] + epsilon < $ymin ) {
            $ymin  = $y[ $i ];
            @xmini = ( $i );
        } elsif ( abs( $y[ $i ] - $ymin ) < epsilon ) {
            $xmini = $i # Remember the index of the smallest x.
                if not defined $xmini or $x[ $i ] < $xmini;
        }
    }

    $xmin  = $x[ $xmini ];
    splice @x, $xmini, 1;         # Remove the minimum point.
    splice @y, $xmini, 1;

    my @a = map {  # Sort the points according to angle with that point.
                   atan2( $y[ $_ ] - $ymin,
                          $x[ $_ ] - $xmin)
                } 0 .. $#x;

    # An unusual Schwartzian Transform.  This leaves us the sorted
    # indices so that we can apply the sort multiple times -- a permutation.

    my @j = map { $_->[ 0 ] }
                sort {      # Sort by the angles, then by x, then by y.
                      return $a->[ 1 ] <=> $b->[ 1 ]             ||
                             $x[ $a->[ 0 ] ] <=> $x[ $b->[ 0 ] ] ||
                             $y[ $a->[ 0 ] ] <=> $y[ $b->[ 0 ] ];
                     }
                    map { [ $_, $a[ $_ ] ] } 0 .. $#a;
```

```
    @x = @x[ @j ];          # Permute.
    @y = @y[ @j ];
    @a = @a[ @j ];

    unshift @x, $xmin;      # Put back the minimum point.
    unshift @y, $ymin;
    unshift @a, 0;

    my @h = ( 0, 1 );        # The hull.
    my $cw;

  # Backtrack: while there are right turns or no turns, shrink the hull.
    for ( $i = 2; $i < $n; $i++ ) {
        while (
            clockwise( $x[ $h[ $#h - 1 ] ],
                       $y[ $h[ $#h - 1 ] ],
                       $x[ $h[ $#h ] ],
                       $y[ $h[ $#h ] ],
                       $x[ $i ],
                       $y[ $i ] ) < epsilon
               and @h >= 2 ) {  # Keep two points in hull at all times.
            pop @h;
        }
        push @h, $i;  # Grow the hull.
    }

    # Interlace x's and y's of the hull back into one list, and return.
    return map { ( $x[ $_ ], $y[ $_ ] ) } 0 .. $#h;
}
```

We can speed up Graham's scan by reducing the number of points that the scan needs to consider. One way to do that is *interior elimination*: throw away all the points that are known *not* to be in the convex hull. This knowledge depends on the distribution of the points: if the distribution is random or even in both directions, a marvelous interior eliminator would be a rectangle stretched between the points closest to the corners. All the points strictly inside the rectangle can be immediately eliminated, as shown in Figure 10-20.

The points closest to the corners can be located by minimizing and maximizing the sums and differences of points:

- smallest sum: lower-left corner
- largest sum: upper-right corner
- smallest difference: upper-left corner
- largest difference: lower-right corner

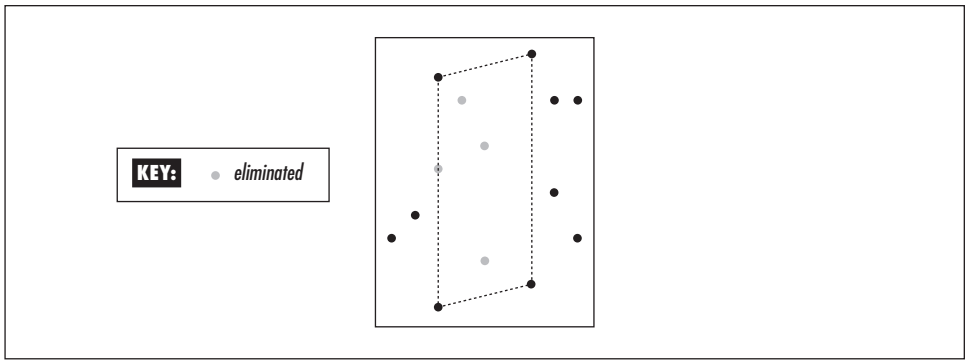

Figure 10-20. Graham's scan: interior elimination for obviously internal points

In Perl, this would be something like the following:

```
# Find out the largest and smallest sums and differences
# (or rather, the indices of those points).

my @sort_by_sum =
        map { $_->[ 0 ] }
            sort { $a->[ 1 ] <=> $b->[ 1 ] }
                map { [ $_, $x[ $_ ] + $y[ $_ ] ] } 0..$#x;

my @sort_by_diff =
        map { $_->[ 0 ] }
            sort { $a->[ 1 ] <=> $b->[ 1 ] }
                map { [ $_, $x[ $_ ] - $y[ $_ ] ] } 0..$#x;

my $ll = $sort_by_sum [  0 ]; # Lower left (of the elimination box).
my $ur = $sort_by_sum [ -1 ]; # Upper right.
my $ul = $sort_by_diff[  0 ]; # Upper left.
my $lr = $sort_by_diff[ -1 ]; # Lower right.
```

This approach has a problem, though: we can safely eliminate only the points *strictly* in the interior of the quadrangle. Points on the quadrangle edges might still be part of the hull, and points exactly at the vertices *will* be on the hull. One way to proceed is to construct a smaller quadrangle that is some tiny (*epsilon*) distance inside of the larger quadrangle. If we choose epsilon well, the points inside the smaller quadrangle will be strictly interior points and can immediately be eliminated from our scan.

The time complexity of `graham_scan()` is $O(N \log N)$, which is optimal.

Closest Pair of Points

Given a set of points, which two are closest to one another? The obvious solution of simply calculating the distance between every possible pair of points works, but not well: it's $O(N^2)$. A practical application would be traffic simulation and control: two jumbo jets shouldn't occupy the same space. While bounding boxes are used to detect collisions, closest points are used to anticipate them. We'll use the set of points in Figure 10-21 as our example.

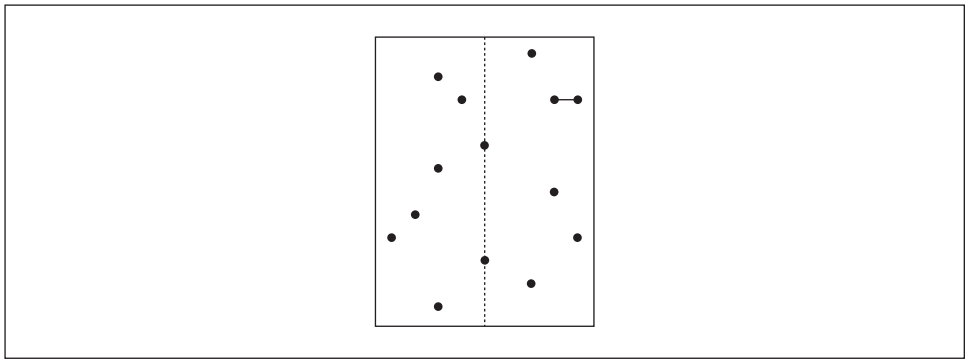

Figure 10-21. A set of points and the closest pair

We can use the intrinsic locality of the points to attack this problem: A point on the left side is likely to be closer to other points on the left than to points on the right. We will once again use the divide-and-conquer paradigm (see the section "Recurrent Themes in Algorithms"), recursively dividing the set of points into left and right halves, as shown in Figure 10-22.

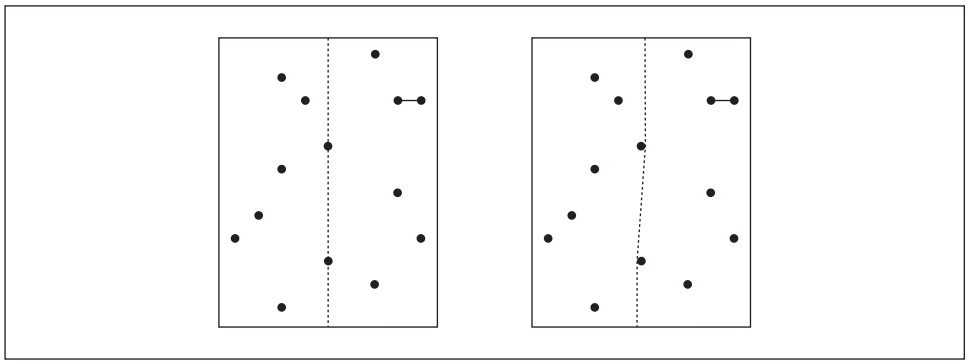

Figure 10-22. Recursive halving: a physical and a logical view

Wondering about the wiggly line of the logical view in Figure 10-22? The halfway of the point set happens to fall on two points that have exactly the same

x-coordinate, so we also show the "logical" view where the dividing line is wiggled ever so slightly to disambiguate the halves.

In Figure 10-23, the vertical slices resulting from the left-right recursion are shown. The slices are labeled; for example, `lrr` is the slice resulting from a left cut followed by two right cuts.

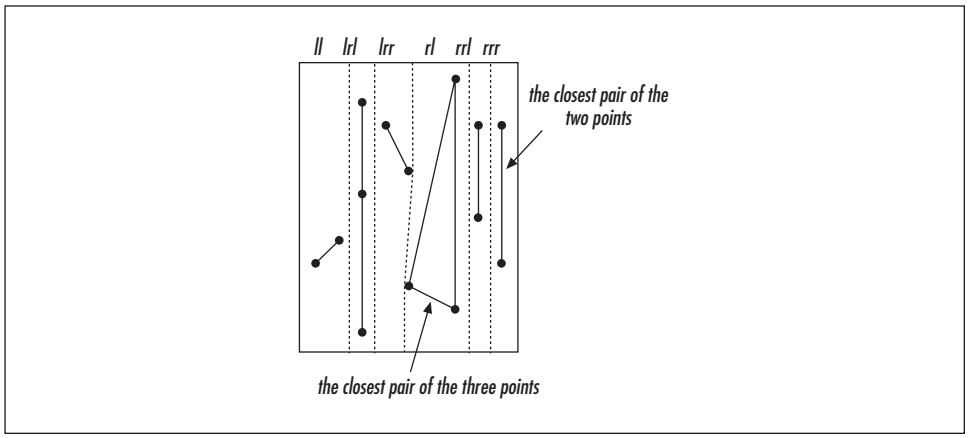

Figure 10-23. All the recursed slices and their closest pairs

The recursion stops when a slice contains only two or three points. In such a case, the shortest distance, or, in other words, the closest pair of points, can be found trivially (see Figure 10-24).

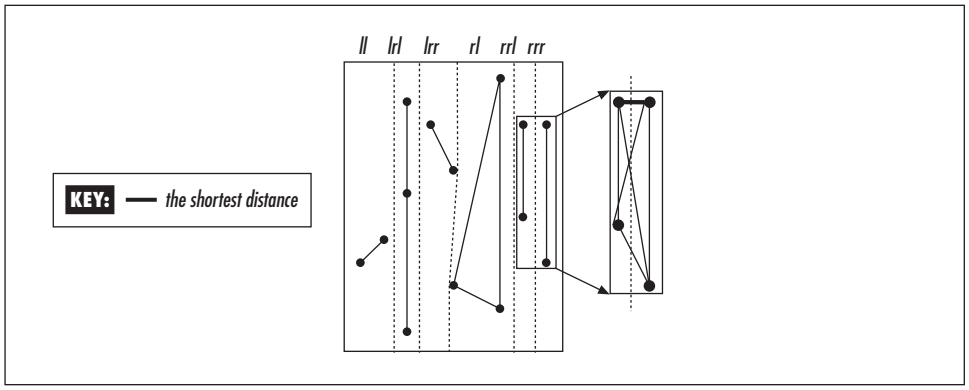

Figure 10-24. Merging the recursed slices

But what should we do when returning from the recursion? Each slice has its own idea of its shortest distance. We cannot simply choose the minimum distance of

the left and right slices because the globally closest pair might straddle the dividing line, illustrated in Figure 10-25.

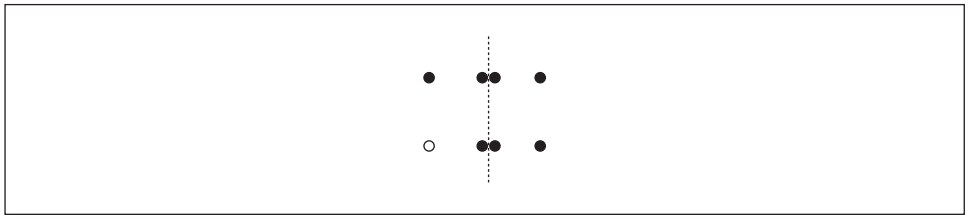

Figure 10-25. The maximal merging scan: the current point is marked white

The trick is as follows: for each dividing line, we must find which points in the bordering halves are closer to the dividing line than the shortest distance found so far. After that we walk these points in *y*-order. For one point we need to check, at most, the seven other points shown in Figure 10-25.

The resulting Perl code is somewhat complex because it needs to maintain several orderings of the point set simultaneously: the original ordering, the points ordered horizontally (this is how we divide the point set horizontally), and the points ordered vertically (scanning the straddling points). These multiple views of the same point sets are implemented by computing various *permutation vectors* implemented as Perl arrays. For example, `@yoi` contains the "vertical rank" of every point, from bottom to top.

Also note that the basic divide-and-conquer technique yields a seemingly $O(N\log N)$ algorithm, but this assumes that the recursion requires only $O(N)$ operations. We cannot repeatedly `sort()` (in either direction) within the recursion without jeopardizing our $O(N\log N)$ rating, so we perform the horizontal and vertical sorts once and then recurse.

Here, then, is the frighteningly long `closest_points()` subroutine:

```
sub closest_points {
    my ( @p ) = @_;

    return () unless @p and @p % 2 == 0;

    my $unsorted_x = [ map { $p[ 2 * $_     ] } 0..$#p/2 ];
    my $unsorted_y = [ map { $p[ 2 * $_ + 1 ] } 0..$#p/2 ];

    # Compute the permutation and ordinal indices.

    # X Permutation Index.
    #
    # If @$unsorted_x is (18, 7, 25, 11), @xpi will be (1, 3, 0, 2),
    # e.g., $xpi[0] == 1 meaning that the $sorted_x[0] is in
```

```
    # $unsorted_x->[1].
    #
    # We do this because we may now sort @$unsorted_x to @sorted_x
    # and can still restore the original ordering as @sorted_x[@xpi].
    # This is needed because we will want to sort the points by x and y
    # but might also want to identify the result by the original point
    # indices: "the 12th point and the 45th point are the closest pair".

    my @xpi = sort { $unsorted_x->[ $a ] <=> $unsorted_x->[ $b ] }
                   0..$#$unsorted_x;

    # Y Permutation Index.
    #
    my @ypi = sort { $unsorted_y->[ $a ] <=> $unsorted_y->[ $b ] }
                   0..$#$unsorted_y;

    # Y Ordinal Index.
    #
    # The ordinal index is the inverse of the permutation index: If
    # @$unsorted_y is (16, 3, 42, 10) and @ypi is (1, 3, 0, 2), @yoi
    # will be (2, 0, 3, 1), e.g. $yoi[0] == 1 meaning that
    # $unsorted_y->[0] is the $sorted_y[1].

    my @yoi;
    @yoi[ @ypi ] = 0..$#ypi;

    # Recurse to find the closest points.
    my ( $p, $q, $d ) = __closest_points_recurse( [ @$unsorted_x[@xpi] ],
                                                  [ @$unsorted_y[@xpi] ],
                                                  \@xpi, \@yoi, 0, $#xpi
                                                );

    my $pi = $xpi[ $p ];                          # Permute back.
    my $qi = $xpi[ $q ];

    ( $pi, $qi ) = ( $qi, $pi ) if $pi > $qi;     # Smaller id first.
    return ( $pi, $qi, $d );
}

sub _closest_points_recurse {
    my ( $x, $y, $xpi, $yoi, $x_l, $x_r ) = @_;

    # $x, $y:  array references to the x- and y-coordinates of the points
    # $xpi:    x permutation indices: computed by closest_points_recurse()
    # $yoi:    y ordering indices: computed by closest_points_recurse()
    # $x_l:    the left  bound of the currently interesting point set
    # $x_r:    the right bound of the currently interesting point set
    #          That is, only points $x->[$x_l..$x_r] and $y->[$x_l..$x_r]
    #          will be inspected.

    my $d;     # The minimum distance found.
    my $p;     # The index of the other end of the minimum distance.
    my $q;     # Ditto.
```

```
my $N = $x_r - $x_l + 1;      # Number of interesting points.

if ( $N > 3 ) {               # We have lots of points.  Recurse!
    my $x_lr = int( ( $x_l + $x_r ) / 2 ); # Right bound of left half.
    my $x_rl = $x_lr + 1;                  # Left bound of right half.

    # First recurse to find out how the halves do.

    my ( $p1, $q1, $d1 ) =
        _closest_points_recurse( $x, $y, $xpi, $yoi, $x_l, $x_lr );
    my ( $p2, $q2, $d2 ) =
        _closest_points_recurse( $x, $y, $xpi, $yoi, $x_rl, $x_r );

    # Then merge the halves' results.

    # Update the $d, $p, $q to be the closest distance
    # and the indices of the closest pair of points so far.

    if ( $d1 < $d2 ) { $d = $d1;  $p = $p1;  $q = $q1 }
    else             { $d = $d2;  $p = $p2;  $q = $q2 }

    # Then check the straddling area.

    # The x-coordinate halfway between the left and right halves.
    my $x_d = ( $x->[ $x_lr ] + $x->[ $x_rl ] ) / 2;

    # The indices of the "potential" points: those point pairs
    # that straddle the area and have the potential to be closer
    # to each other than the closest pair so far.
    #
    my @xi;

    # Find the potential points from the left half.

    # The left bound of the left segment with potential points.
    my $x_ll;

    if ( $x_lr == $x_l ) { $x_ll = $x_l }
    else {                                    # Binary search.
        my $x_ll_lo = $x_l;
        my $x_ll_hi = $x_lr;
        do { $x_ll = int( ( $x_ll_lo + $x_ll_hi ) / 2 );
             if ( $x_d - $x->[ $x_ll ] > $d ) {
                $x_ll_lo = $x_ll + 1;
             } elsif ( $x_d - $x->[ $x_ll ] < $d ) {
                $x_ll_hi = $x_ll - 1;
             }
        } until $x_ll_lo > $x_ll_hi
            or ( $x_d - $x->[ $x_ll ] < $d
                 and ( $x_ll == 0 or
                       $x_d - $x->[ $x_ll - 1 ] > $d ) );
    }
    push @xi, $x_ll..$x_lr;
```

```
# Find the potential points from the right half.

# The right bound of the right segment with potential points.
my $x_rr;

if ( $x_rl == $x_r ) { $x_rr = $x_r }
else {                                      # Binary search.
    my $x_rr_lo = $x_rl;
    my $x_rr_hi = $x_r;
    do { $x_rr = int( ( $x_rr_lo + $x_rr_hi ) / 2 );
         if ( $x->[ $x_rr ] - $x_d > $d ) {
            $x_rr_hi = $x_rr - 1;
         } elsif ( $x->[ $x_rr ] - $x_d < $d ) {
            $x_rr_lo = $x_rr + 1;
         }
    } until $x_rr_hi < $x_rr_lo
        or ( $x->[ $x_rr ] - $x_d < $d
             and ( $x_rr == $x_r or
                   $x->[ $x_rr + 1 ] - $x_d > $d ) );
}
push @xi, $x_rl..$x_rr;

# Now we know the potential points.  Are they any good?
# This gets kind of intense.

# First sort the points by their original indices.

my @x_by_y   = @$yoi[ @$xpi[ @xi ] ];
my @i_x_by_y = sort { $x_by_y[ $a ] <=> $x_by_y[ $b ] }
               0..$#x_by_y;
my @xi_by_yi;
@xi_by_yi[ 0..$#xi ] = @xi[ @i_x_by_y ];

my @xi_by_y = @$yoi[ @$xpi[ @xi_by_yi ] ];
my @x_by_yi = @$x[ @xi_by_yi ];
my @y_by_yi = @$y[ @xi_by_yi ];

# Inspect each potential pair of points (the first point
# from the left half, the second point from the right).

for ( my $i = 0; $i <= $#xi_by_yi; $i++ ) {
    my $i_i = $xi_by_y[ $i ];
    my $x_i = $x_by_yi[ $i ];
    my $y_i = $y_by_yi[ $i ];
    for ( my $j = $i + 1; $j <= $#xi_by_yi; $j++ ) {
        # Skip over points that can't be closer
        # to each other than the current best pair.
        last if $xi_by_y[ $j ] - $i_i > 7; # Too far?
        my $y_j = $y_by_yi[ $j ];
        my $dy = $y_j - $y_i;
        last if $dy > $d;                  # Too tall?
        my $x_j = $x_by_yi[ $j ];
        my $dx = $x_j - $x_i;
        next if abs( $dx ) > $d;           # Too wide?
```

```
                # Still here?  We may have a winner.
                # Check the distance and update if so.
                my $d3 = sqrt( $dx**2 + $dy**2 );
                if ( $d3 < $d ) {
                    $d = $d3;
                    $p = $xi_by_yi[ $i ];
                    $q = $xi_by_yi[ $j ];
                }
            }
        }
    } elsif ( $N == 3 ) {      # Just three points?  No need to recurse.
        my $x_m = $x_l + 1;
        # Compare the square sums and leave the sqrt for later.
        my $s1 = ($x->[ $x_l ]-$x->[ $x_m ])**2 +
                 ($y->[ $x_l ]-$y->[ $x_m ])**2;
        my $s2 = ($x->[ $x_m ]-$x->[ $x_r ])**2 +
                 ($y->[ $x_m ]-$y->[ $x_r ])**2;
        my $s3 = ($x->[ $x_l ]-$x->[ $x_r ])**2 +
                 ($y->[ $x_l ]-$y->[ $x_r ])**2;
        if ( $s1 < $s2 ) {
            if ( $s1 < $s3 )  { $d = $s1;  $p = $x_l;  $q = $x_m }
            else              { $d = $s3;  $p = $x_l;  $q = $x_r }
        } elsif ( $s2 < $s3 ) { $d = $s2;  $p = $x_m;  $q = $x_r }
          else                { $d = $s3;  $p = $x_l;  $q = $x_r }

        $d = sqrt $d;
    } elsif ( $N == 2 ) {         # Just two points?  No need to recurse.
        $d = sqrt(($x->[ $x_l ]-$x->[ $x_r ])**2 +
                  ($y->[ $x_l ]-$y->[ $x_r ])**2);
        $p = $x_l;
        $q = $x_r;
    } else {                      # Less than two points?  Strange.
        return ( );
    }

    return ( $p, $q, $d );
}
```

The time complexity of `closest_points()` is $O(N \log N)$, which should be both a familiar expression and good news by now. We'll test it with the points in Figure 10-26.

We can find the closest pair of points out of the set of ten points in Figure 10-26 as follows:

```
@clopo = closest_points( 1, 2,  2, 5,  3, 1,  3, 3,  4, 5,
                         5, 1,  5, 6,  6, 4,  7, 4,  8, 1 ), "\n";
print "@clopo\n";
```

The result:

```
7 8 1
```

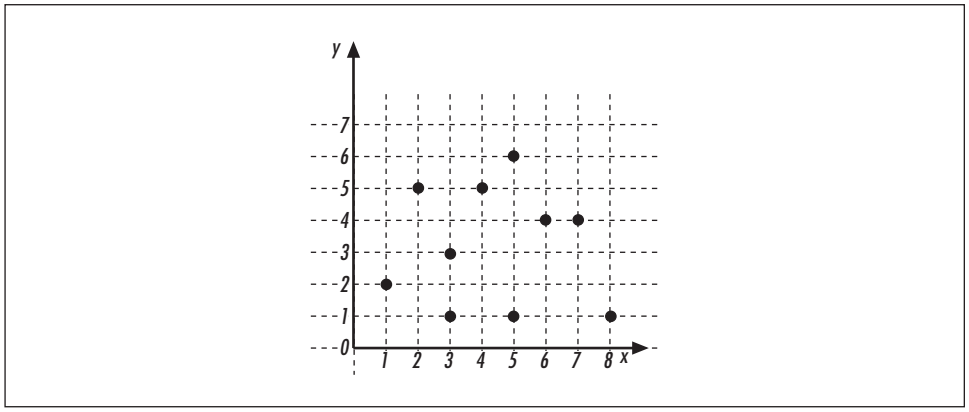

Figure 10-26. An example of the closest pair of points problem

This tells us that the eighth and ninth points—(6, 4) and (7, 4), since Perl arrays are zero-indexed—are the closest pair of points, and that they have a distance of 1.

Geometric Algorithms Summary

Geometric algorithms are often based on familiar geometry formulas, but be careful: often, translating them to a computer program is not as straightforward as it might seem. The main source of problems is the conflict between the ideal numbers of mathematics and the inaccurate representation of real numbers in computers (*discretization* is the fancy name for this unavoidable translation). You may think a point lies exactly at the intersection of $x-1$ and $1-2x$, but that's not what your computer thinks. And your circle of radius 1 doesn't contain π pixels, either.

CPAN Graphics Modules

The algorithms we discussed in this chapter never actually paint points on your screen. For that, you need one of the packages discussed in this section. Most of these modules are interfaces to external libraries; you need to install those libraries first. The documentation bundled with the modules tells you where to find them. The modules themselves can all be found at *http://www.perl.com/CPAN/modules*.

2-D Images

There are five CPAN modules for manipulating two-dimensional images: PerlGimp, GD, Image::Size, PerlMagick, and PGPLOT.

Perl-Gimp

The Gimp is a popular Linux utility similar to Adobe Photoshop; see *http://www.gimp.org*. Perl-Gimp, by Marc Lehman, is a Perl API to Gimp, letting you warp, speckle, shadow, and perform countless other effects on your images.

GD

The GD module, by Lincoln D. Stein, is an interface to *libgd*, a library that allows you to "draw" GIF images. For example, you can produce a GIF image of a circle like this:

```
use GD;

# Create the image.
my $gif = new GD::Image(100, 100);

# Allocate colors.
my $white = $gif->colorAllocate(255, 255, 255);
my $red   = $gif->colorAllocate(255,   0,   0);

# Background color.
$gif->transparent($white);

# The circle.
$gif->arc(50, 50,        # Center x, y.
          30, 30,        # Width, Height.
          0, 360,        # Start Angle, End Angle.
          $red);         # Color.

# Output the image.
open(GIF, ">circle.gif") or die "open failed: $!\n";
binmode GIF;
print GIF $gif;
close GIF;
```

Image::Size

Randy J. Ray's Image::Size is a special-purpose module for peeking at graphics files and determining their size or dimensions. This may sound like a strangely specific task, but it has a very common and important real-world use. When a web server is transmitting a web page, it should print out the image size as soon as possible, before transmitting the actual image. That way, the web browser can render bounding boxes of the images as soon as possible. That enables a much smoother rendering process because the page layout won't jump abruptly when the images finally arrive.

PerlMagick

The PerlMagick module, by Kyle Shorter, is an interface to *ImageMagick*, an extensive image conversion and manipulation library. You can convert from one graphics format to another and manipulate the images with all kinds of filters ranging from color balancers to cool special effects. See *http://www.wizards.dupont.com/cristy/www/perl.html*.

PGPLOT

Karl Glazebrook's PGPLOT module is an interface to the PGPLOT graphics library. You can use PGPLOT to draw images with labels and all that, but coupled with the PDL numerical language (yet another Perl module, see Chapter 7) it becomes a very powerful tool indeed. Because all the power of Perl is available to PDL, it's getting scary. See *http://www.ast.cam.ac.uk/AAO/local/www/kgb/pgperl/* for more information.

Charts a.k.a. Business Graphics

If by "graphics" you mean "business graphics" (bar charts, pie charts, and the like), check out the Chart and GIFgraph modules, by David Bonner and Martien Verbruggen. You can use them, say, to create web site usage reports on the fly. They both require the GD module.

3-D Modeling

Only in recent years has realistic three-dimensional modeling become possible on computers that everyone can afford. Three toolkits have freely available CPAN modules: OpenGL, Renderman, and VRML.

OpenGL

Stan Melax' OpenGL module implements the OpenGL interface for Perl. OpenGL is an open version of Silicon Graphics' GL language; it's a 3-D modeling language with which you can define your "worlds" with complex objects and lighting conditions. The popular Quake game is rendered using OpenGL. There is a publicly available implementation of OpenGL called *Mesa*; see *http://www.mesa3d.org/*.

Renderman

The Renderman module, by Glenn M. Lewis, is an interface to the Pixar's Renderman photorealistic modeling system. You may now start writing your own Toy Story with Perl.

VRML

Hartmut Palm has implemented a Perl interface to the Virtual Reality Markup Language, which lets you define a three-dimensional world and output the VRML describing it. If people visiting your web site have the appropriate plug-in, they can walk around in your world. The module is called, rather unsurprisingly, VRML.

Widget/GUI Toolkits

If you want to develop your own graphical application independent of the Web, you'll need one of the packages described in this section. Perl/Tk is far and away the most feature-filled and portable system.

Perl/Tk

Perl/Tk, by Nick Ing-Simmons, is the best graphical widget toolkit available for Perl.* It works under the X11 Window System and under Windows 95/98/NT/2K.

Perl/Tk is easy to launch. Here's a minimal program that displays a button:

```
use Tk;
$MW = MainWindow->new;
$hello = $MW->Button(
      -text    => 'Hello, world',
      -command => sub { print STDOUT "Hello, world!\n"; exit; },
);
$hello->pack;
MainLoop;
```

The button has an *action* bound to it: when you press it, `Hello, world!` is printed to the controlling terminal. You can implement sophisticated graphical user interfaces and graphical applications with Tk, and it's far too large a subject to cover in this book. In fact, Tk is worthy of a book of its own: *Learning Perl/Tk*, by Nancy Walsh (O'Reilly & Associates).

Other windowing toolkits

There are Perl bindings for several other windowing toolkits. The toolkits mainly work only under the X Window System used in Unix environments, but some have upcoming Windows ports (Gtk, as of mid 1999).

---

* Perl's Tk *module* should not be confused with the Tk *toolkit*, which was originally written by John Ousterhout for use with his programming language, Tcl. The Tk toolkit is language-independent, and that's why it can interface with, for example, Perl. The Perl/Tk module is an interface to the toolkit.

Gnome by Kenneth Albanowski
The GNU Object Model Environment (*http://www.gnome.org*)

Gtk by Kenneth Albanowski
The toolkit originally used by Gimp

Sx by Frederic Chaveau
Simple *Athena Widgets* for X

X11::Motif by Ken Fox
A Motif toolkit