Geometry

This chapter will deal with programming problems associated with "real" geometry – lines, points, circles, and so forth.

Although you did geometry in high school, it can be surprisingly difficult to program even very simple things. One reason is that floating point arithmetic introduces numerical uncertainty.

Another difficulty of geometric programming is that certain "obvious" operations you do with a pencil, such as finding the intersection of two lines, requires non-trivial programming to do correctly with a computer.

Finally, special cases or *degeneracies* require extra attention when doing geometric programming. For these reasons I recommend you carefully study my code fragments before writing your own.

There is more geometry to come next week, when we consider problems associated with line segments and polygons a field known as *computational geometry*,

Lines

Straight *lines* are the shortest distance between any two points. Lines are of infinite length in both directions, as opposed to *line segments*, which are finite. We limit our discussion here to lines in the plane.

Every line l is completely represented by any pair of points (x_1, y_1) and (x_2, y_2) which lie on it.

Lines are also completely described by equations such as $y = mx + b$, where m is the *slope* of the line and b is the y-*intercept*, i.e., the unique point $(0, b)$ where it crosses the x-axis.

Vertical lines cannot be described by such equations, however, because dividing by Δx means dividing by zero. The equation $x = c$ denotes a vertical line that crosses the x-axis at the point $(c, 0)$.

We use the more general formula $ax + by + c = 0$ as the foundation of our line type because it covers all possible lines in the plane:

```
typedef struct {
        double a;               /* x-coefficient */
        double b;               /* y-coefficient */
        double c;               /* constant term */
} line;
```

Multiplying these coefficients by any non-zero constant yields an alternate representation for any line. We establish a canonical representation by insisting that the y-coefficient equal 1 if it is non-zero. Otherwise, we set the x-coefficient to 1:

```
points_to_line(point p1, point p2, line *l)
{
        if (p1[X] == p2[X]) {
                l->a = 1;
                l->b = 0;
                l->c = -p1[X];
        } else {
                l->b = 1;
                l->a = -(p1[Y]-p2[Y])/(p1[X]-p2[X]);
                l->c = -(l->a * p1[X]) - (l->b * p1[Y]);
        }
}

point_and_slope_to_line(point p, double m, line *l)
{
        l->a = -m;
        l->b = 1;
        l->c = -((l->a*p[X]) + (l->b*p[Y]));
}
```

Line Intersection

Two distinct lines have one *intersection point* unless they are *parallel*; in which case they have none. Parallel lines share the same slope but have different intercepts and by definition never cross.

```
bool parallelQ(line l1, line l2)
{
     return ( (fabs(l1.a-l2.a) <= EPSILON) &&
              (fabs(l1.b-l2.b) <= EPSILON) );
}
```

The intersection point of lines $l_1 : y = m_1 x + b_1$ and $l_2 : y_2 = m_2 x + b_2$ is the point where they are equal, namely,

$$x = \frac{b_2 - b_1}{m_1 - m_2}, \qquad y = m_1 \frac{b_2 - b_1}{m_1 - m_2} + b_1$$

```
intersection_point(line l1, line l2, point p)
{
     if (same_lineQ(l1,l2)) {
         printf("Warning: Identical lines, all points intersect.\n");
         p[X] = p[Y] = 0.0;
         return;
     }

     if (parallelQ(l1,l2) == TRUE) {
         printf("Error: Distinct parallel lines do not intersect.\n");
         return;
     }

     p[X] = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);

     if (fabs(l1.b) > EPSILON)       /* test for vertical line */
             p[Y] = - (l1.a * (p[X]) + l1.c) / l1.b;
     else
             p[Y] = - (l2.a * (p[X]) + l2.c) / l2.b;
}
```

Angles

An *angle* is the union of two rays sharing a common endpoint.

The entire range of angles spans from 0 to 2π radians or, equivalently, 0 to 360 degrees. Most trigonometric libraries assume angles are measured in radians.

A *right* angle measures 90^{O} or $\pi/2$ radians.

Any two non-parallel lines intersect each other at a given angle. Lines $l_1 : a_1x + b_1y + c_1 = 0$ and $l_2 : a_2x + b_2y + c_2 = 0$, written in the general form, intersect at the angle θ given by:

$$\tan\theta = \frac{a_1b_2 - a_2b_1}{a_1a_2 + b_1b_2}$$

For lines in slope-intercept form this reduces to $\tan\theta = (m_2 - m_1)/(m_1m_2 + 1)$.

Two lines are *perpendicular* if they cross at right angles to each other. The line perpendicular to $l : y = mx + b$ is $y = (-1/m)x + b'$, for all values of b'.

Closest Point

A very useful subproblem is identifying the point on line l which is closest to a given point p. This closest point lies on the line through p which is perpendicular to l, and hence can be found using the routines we have already developed:

```
closest_point(point p_in, line l, point p_c)
{
      line perp;              /* perpendicular to l through (x,y) */

      if (fabs(l.b) <= EPSILON) {     /* vertical line */
              p_c[X] = -(l.c);
              p_c[Y] = p_in[Y];
              return;
      }

      if (fabs(l.a) <= EPSILON) {     /* horizontal line */
              p_c[X] = p_in[X];
              p_c[Y] = -(l.c);
              return;
      }

      point_and_slope_to_line(p_in,1/l.a,&perp); /* normal case */
      intersection_point(l,perp,p_c);
}
```

Triangles and Trigonometry

Each pair of rays with a common endpoint defines an *internal angle* of a radians and an *external angle* of $2\pi - a$ radians. The three internal (smaller) angles of any triangle add up to $180^{\text{O}} = \pi$ radians.

The *Pythagorean theorem* enables us to calculate the length of the third side of any *right* triangle given the length of the other two. Specifically, $|a|^2 + |b|^2 = |c|^2$, where a and b are the two shorter sides, and c is the longest side or *hypotenuse*.

We can go farther to analyze triangles using trigonometry. The trigonometric functions *sine* and *cosine* are defined as the x- and y-coordinates of points on the unit circle centered at $(0, 0)$. A third important trigonometric function is the *tangent*, defined as the ratio of sine over cosine.

These functions are important, because they enable us to relate the lengths of any two sides of a right triangle T with the non-right angles of T. The non-hypotenuse edges can be labeled as *opposite* or *adjacent* edges in relation to a given angle a. Then

$$\cos(a) = \frac{|\text{adjacent}|}{|\text{hypotenuse}|}, \quad \sin(a) = \frac{|\text{opposite}|}{|\text{hypotenuse}|}, \quad \tan(a) = \frac{|\text{opposite}|}{|\text{adjacent}|}$$

Use the famous Indian Chief Soh-Cah-Toa to remember these relations, where each syllable of his name encodes a different relation. "Cah" means *C*osine equals *A*djacent over *H*ypotenuse, for example.

Using trigonometry described in the text, we can solve arbitrary triangles for lengths and angles given enough partial information.

Area of a Triangle

The area $A(T)$ of a triangle T is given by $A(T) = (1/2)ab$, where a is the altitude and b is the base of the triangle.

Another approach to computing the area of a triangle is directly from its coordinate representation. Using linear algebra and determinants, it can be shown that the *signed* area $A(T)$ of triangle $T = (a, b, c)$ is

$$2 \cdot A(T) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y$$

This formula generalizes nicely to compute $d!$ times the volume of a simplex in d dimensions.

Note that the signed areas can be negative, so we must take the absolute value to compute the actual area. The sign of this area can be used to build important primitives for computational geometry.

```
double signed_triangle_area(point a, point b, point c)
{
        return( (a[X]*b[Y] - a[Y]*b[X] + a[Y]*c[X]
                - a[X]*c[Y] + b[X]*c[Y] - c[X]*b[Y]) / 2.0 );
}

double triangle_area(point a, point b, point c)
{
        return( fabs(signed_triangle_area(a,b,c)) );
}
```

Circles

A *circle* is defined as the set of points at a given distance (or *radius*) from its *center*, (x_c, y_c). A circle can be represented in two basic ways, either as triples of boundary points or by its center/radius. For most applications, the center/radius representation is most convenient:

```
typedef struct {
        point c;                /* center of circle */
        double r;               /* radius of circle */
} circle;
```

The equation of a circle of radius r follows directly from its center/radius representation, $r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$.

Many important quantities associated with circles are easy to compute. Specifically, $A = \pi r^2$ and $C = 2\pi r$.

A *tangent* line l intersects the boundary of c but not its interior. The point of contact between c and l lies on the line perpendicular to l through the center of c. Since the triangle with side lengths r, d, and x is a right triangle, we can compute the unknown tangent length x using the Pythagorean theorem. From x, we can compute either the tangent point or the angle a. The distance d from O to the center is computed using the distance formula.

Two circles c_1 and c_2 of distinct radii r_1 and r_2 will intersect if and only if the distance between their centers is at most $r_1 + r_2$.

The points of intersection form triangles with the two centers whose edge lengths are totally determined (r_1, r_2, and the distance between the centers), so the angles and coordinates can be computed as needed.

Assigned Problems

111301 (Dog and Gopher) – Find a hole for a slow gopher to hide from a fast dog. Is the closest hole *really* the safest spot for the gopher?

111302 (Rope Crisis in Ropeland!) – What is the length of the shortest rope between two points around a circular post

111305 (Birthday Cake) – How do you cut a circular cake such that both pieces are the same area *and* contain the same number of cherries? There is always a solution to this problem if we remove the constraint that the cutline have integer coordinates – can you prove it? Is there a more efficient solution than trying all possible A, B pairs?

111308 (How Big Is It?) – What is the tightest way to order a non-overlapping collection of circles so as to minimize the 'shadow' of them? Is it better to order the circles from largest to smallest, or to interleave them? Does the order ever *not* matter? Will backtracking work for this problem?