

Brute Force Convex Hull Computation

Computational geometry is a discipline within computer science and mathematics devoted to geometric algorithms—solving problems involving points, lines, shapes in two and three dimensions, and so forth. While much of the field is dedicated to just solving straightforwardly stated geometry problems, much of the work in the field can be applied to solve problems in computer vision, robotics, route planning, and circuit design.

One of the most popular introductory computational geometry problems reads as follows: Given a set of points in two-dimensional space, compute the smallest convex polygon that contains all of the points. There are a large number of algorithms that can be used to compute such a polygon—the smallest **convex hull** or **convex envelope**, as it's called—but we'll concern ourselves with a fairly obvious, brute force algorithm, viewing it not so much as a computational geometry problem that needs to be solved as efficiently as possible, but more as a novel application where our CS106 container classes can be used to cleanly articulate a working program.

Core Data Structures

I'm hoping it's quite apparent that the following record definition does a spectacular job representing a point in two-dimensional space:

```
struct point {
    double x;
    double y;
};
```

The plan is to model a set of points as—of all things!—a **Set<point>**. Since the **point** is a custom data structure, it doesn't automatically respond to infix **==** and **<** like the built-in primitives do, and that means we need to define a comparator function that **imposes** an ordering so that the **Set** can use it to organize the points it stores behind the scenes. For reasons that'll be clear in a bit, we're going to use the **y** coordinate as the primary coordinate, relying on the **x** coordinate only when the two points being compared share the same **y** value.

```
int PointCompare(point one, point two) {
    if (one.y < two.y) return -1;
    if (one.y > two.y) return +1;
    if (one.x < two.x) return -1;
    if (one.x > two.x) return +1;
    return 0;
}
```

This has the nice side effect of bringing the point with the smallest **y** coordinate—the "smallest" point as far as our **PointCompare** function is concerned—to the front of a **Set**'s iterator.

We can start off easy and figure out how to generate a set of points within a certain radius of graphic window—configured to be perfectly square.

```
static const int kNumPoints = 35;
static const double kDiameter = 4.5;
point center = { GetWindowWidth()/2, GetWindowHeight()/2 };
Set<point> points(PointCompare);
populateWithRandomPoints(points, kNumPoints, center, kDiameter/2);
```

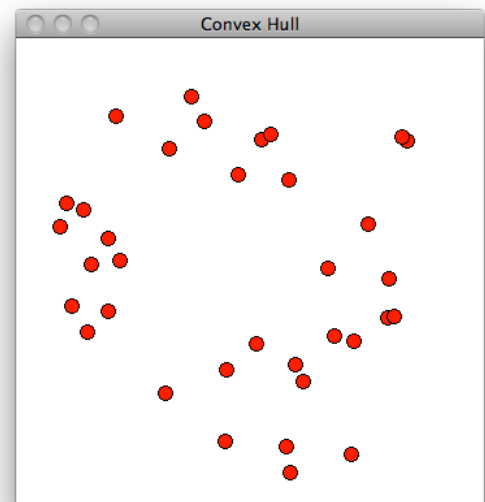
Notice the **Set<point>** is constructed to rely on our **PointCompare** function to sort elements behind the scenes (not because it has to, but because it can, since sets aren't concerned with order).

The collection of points is generated via the following pair of functions:

```
point createRandomPointWithinRadius(point center, double radius) {
    while (true) {
        double dx = RandomReal(-radius, radius);
        double dy = RandomReal(-radius, radius);
        point p = {center.x + dx, center.y + dy};
        if (sqrt(dx * dx + dy * dy) <= radius) return p;
    }
}

void populateWithRandomPoints(Set<point>& points, int numPoints,
                             point center, double radius) {
    for (int i = 0; i < numPoints; i++) {
        points.add(createRandomPointWithinRadius(center, radius));
    }
}
```

The **populateWithRandomPoints** function itself is straightforward, and the only strange part of the **createRandomPointWithinRadius** function is the last line of the **while** loop's body, which effectively rejects the randomly generated point unless it's within a distance of **radius** from the provided **center**. So that the points seems to randomly pepper the graphics window with no particular bias toward the center, I just generate a random point on the screen, and elect to keep it if it falls within the circle that's perfectly circumscribed by the graphics window itself.



At this point, we want to compute the convex hull enclosing all 35 points. In other words, we want the following line to just do the right thing:

```
Vector<point> convexHull = computeConvexHull(points);
```

Here's the implementation of **computeConvexHull**:

```
Vector<point> computeConvexHull(Set<point>& points) {
    point first = points.iterator().next(); // identifies the bottommost point

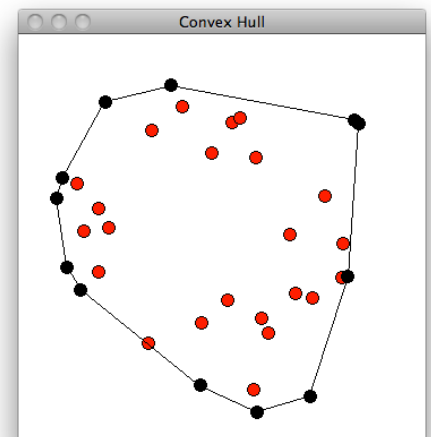
    Vector<point> convexHull;
    convexHull.add(first);
    double minAngle = 0.0;

    while (true) {
        point curr = convexHull[convexHull.size() - 1];
        point next = computeNextOnConvexHull(points, curr, minAngle);
        convexHull.add(next);
        if (PointCompare(first, next) == 0) break;
        minAngle = computePolarAngle(curr, next); // increase threshold
    }

    return convexHull;
}
```

The first point is set to be the smallest point in the **Set** as far **PointCompare** is concerned, and I get that point by asking its iterator to produce its very first element. This bottommost point is first in the list of points describing the hull we're interested in, which is why it's immediately added to the running **Vector<point>** that will ultimately describe the convex hull.

The first iteration of the while loop looks for the **next** point in the hull by identifying the one that, relative to the first one, sits at the lowest positive polar angle. We effectively sweep from 3 o'clock, in a counterclockwise manner—think of an old-school submarine radar!—until we hit our first point, and call that point the next one in the hull. The polar angle of the line between the first two points defines the new start angle (note that **minAngle** goes from 0.0 to something larger¹) where the sweep for the third point should begin, and the process is repeated until we return back to our bottommost point.



The **computeNextOnConvexHull** function is about as brute force a search for a minimum as they come. It essentially redefines **curr** as the origin, and we reduce all other points to their relative polar angles, keeping track of the smallest such angle (and the point that produces it) less than 360 degrees (or 2π radians) but greater than or equal to the provided **minAngle**. The first call sets **minAngle** to be 0.0, but subsequent calls pass in larger and larger values to reflect the fact that we have an increasing lower threshold on what polar angle we're willing to associate with the next on the convex hull.

¹ Technically, the threshold angle could stay the same if the smallest point and its successor happen to share the same y value, but for simplicity we'll assume no three points are collinear (although our solution works even if that happens!)

```
double computePolarAngle(point from, point to) {
    double angle = atan2(to.y - from.y, to.x - from.x);
    if (angle < 0) angle += 2 * kPi;
    return angle;
}

point computeNextOnConvexHull(Set<point>& points, point curr, double minAngle) {
    point next;
    double minAngleSoFar = 2 * kPi;

    points.remove(curr);
    foreach (point p in points) {
        double angle = computePolarAngle(curr, p);
        if (angle < minAngleSoFar && angle >= minAngle) {
            next = p;
            minAngleSoFar = angle;
        }
    }
    points.add(curr);

    return next;
}
```

Ta da!