CPS234 Computational Geometry **September 1, 2005**

Lecture 2: Convex Hulls

Lecturer: Pankaj K. Agarwal *Scribe: Sam Slee*

2.1 Introduction to Convex Hulls

This lecture will introduce the idea of a *convex hull* along with the necessary mathematical definitions for understanding it. After introducing the background concepts, two algorithms for computing convex hulls will be given. Only concepts in 2D ($\Re^2$) will be covered in this lecture, leaving later lectures to generalize to higher dimensions.

2.1.1 Convex Sets and Half Planes

Convex Sets One common problem that arises in computational geometry is the problem of computing the *convex hull* of a set of points. In this lecture, we will cover what a convex hull is and introduce some techniques for computing a convex hull for points in $\Re^2$. These are also known as *planar convex hulls*. Later in the course this will be generalized to higher dimensions for points in $\Re^d$ for $d > 2$. Before defining what a convex hull is though, it is helpful to first define a *convex set*.

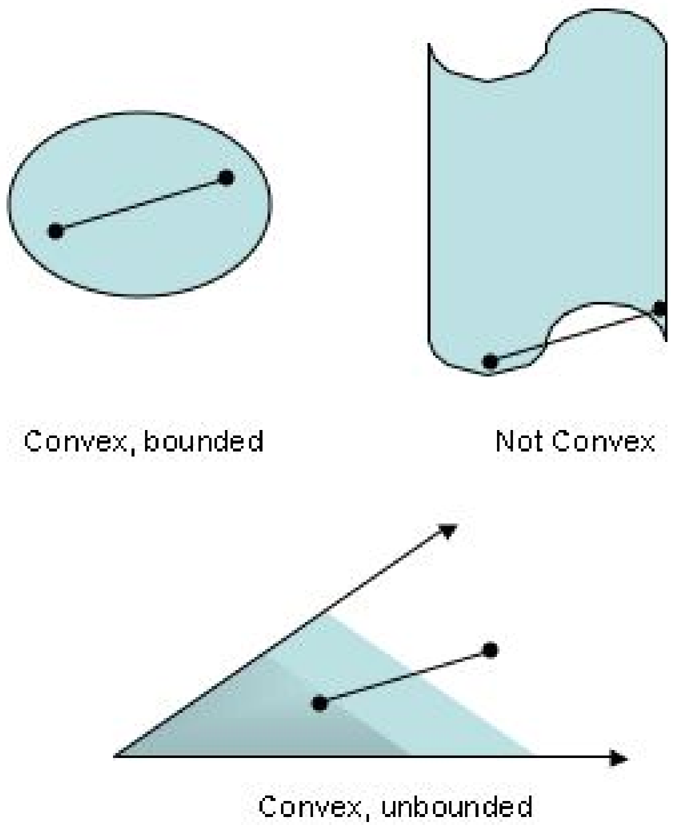
Definition 1 *Given a set of points $C \subseteq \Re^d$, C is convex and forms a* ***convex set*** *if $\forall p, q \in \Re^2$:*

$$\begin{aligned} p, q, \in C \quad &\Rightarrow \quad \overline{pq} \subseteq C, \\ p, q \in C \quad &\Rightarrow \quad \lambda p + (1-\lambda)q \in C, \\ \forall\, 0 \leq \lambda \leq 1. \end{aligned}$$

That is, if we have two points p and q that are contained in the set C, if C is convex then all points on a straight line segment between p and q must also be contained in C. As part of that definition a *convex combination* of p and q was used.

Definition 2 *A* ***convex combination*** *of two points p and q is any combination $\lambda p + (1-\lambda)q$ such that* $0 \leq \lambda \leq 1$.

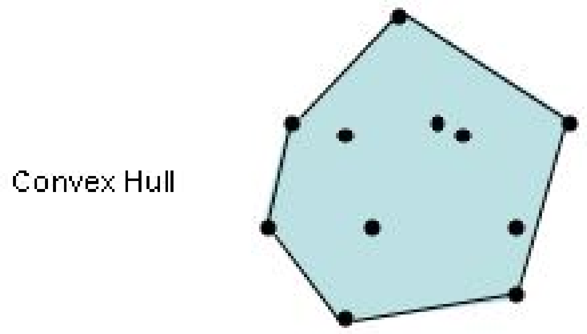
Below some examples of convex and non-convex sets are given. Note that convex sets do not have to be bounded, as is shown in the last example.



With the definition of convex sets in hand we are now ready to define convex hulls. The following definition is for sets of points in $\Re^2$ but may be generalized to higher dimensions.

Definition 3 *Let S be a set of points $\subseteq \Re^2$. The* ***convex hull*** *of S, denoted conv(S), is the smallest convex set that contains S.*

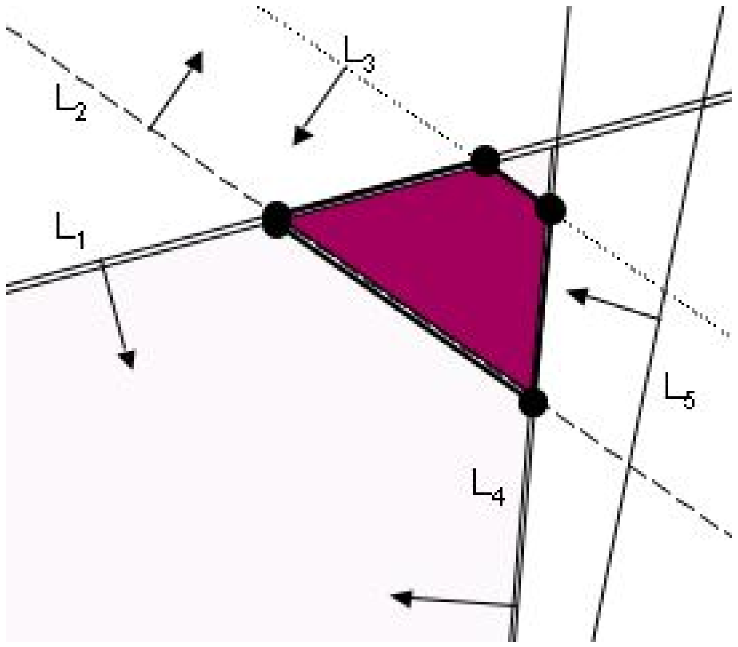
In the case of two dimensional points, one common analogy is to think of the points as pegs or nails in a board. Then take a rubber band, stretched to contain all of the pegs, and allow that rubber band to shrink down in size around those pegs as far as it can. This will form straight edges between the outer pegs/points which mark the boundaries of the convex set. An example convex hull is shown below.



Half Planes Another way to think about convex hulls involves *half planes*. Consider a line $y = ax + b$ in $\Re^2$. That line divides the 2D plane into two **half planes**: $y \geq ax+b$ and $y \leq ax+b$. Note that the distinction between a *line*, which continues infinitely in two directions, and a *line segment*, which is finite and has two endpoints. This is important here as only a line will work for dividing a plane into two halves. In higher dimensions this idea of lines and half planes may be generalized to *hyper planes* in $\Re^d$ dividing spaces in $\Re^d$ into *half spaces*.

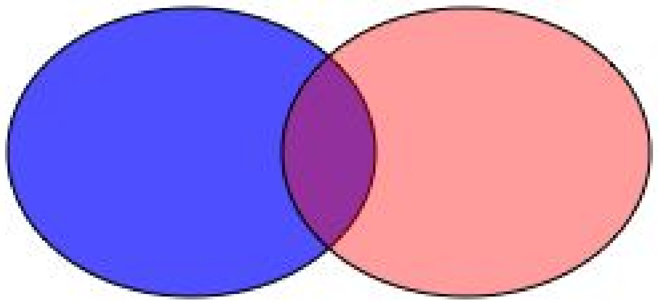
Returning to the concept of convex hulls, we may see now that a convex hull in $\Re^2$ is just a combination of half planes. A definition is given below, followed by an example diagram of a convex hull made from the intersection of half planes.

Definition 4 *conv(S) =* $\cap h \mid h$*: is a half plane and S* $\subseteq$ *h*



In the diagram above, it can be seen that the convex hull pictured is comprised of the intersection of several half planes. Everything below line L_1 is a half plane, as is everything above L_2, below L_3 and to the left of L_4. Most of the lines that define these half planes lie on the line segments that comprise the boundary of the convex hull. Everything to the left of L_5 is a half plane that also intersects the others, but it does nothing to change the convex hull formed by the other half planes. A half plane boundary that does not lie on the boundary of the convex hull is called **redundant**. To denote the boundary of a convex hull of a set S, we use the notation: ∂ conv(S).

Aside: Note that an *intersection* of convex sets is also convex. One example of this was the intersection of half planes shown above. However, a *union* of convex sets is *not* convex. A clear example of this would be the union of two slightly over-lapped circles (like you might see for a Venn Diagram). Both circles form convex sets separately, but when slightly overlapped we can see that their union is clearly not a convex set. In the diagram below, the blue circle and the red circle each separately form convex sets. Their intersection — the purple area — is also convex. However, their union — all of both circle areas combined — is clearly not convex.



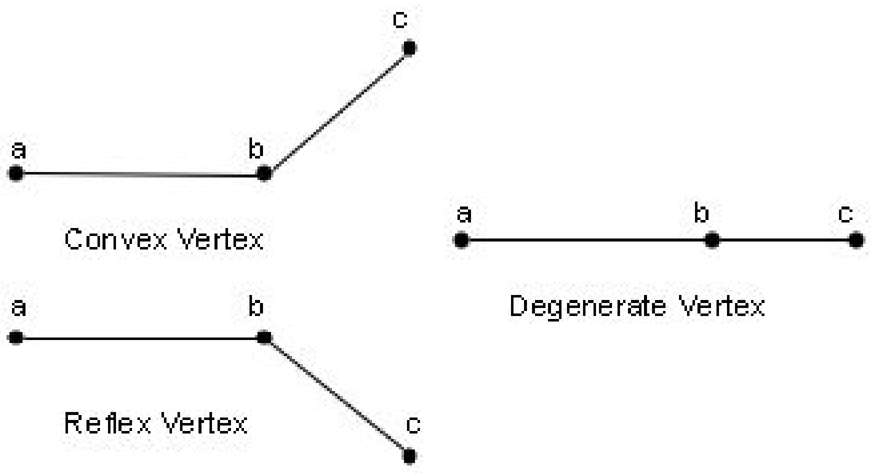
2.1.2 Convex Hull Properties

Returning to convex hulls, in the examples that have been shown so far it can be seen that the boundary of a convex hull is comprised of vertices and edges. A given convex hull may then be defined by the points on its boundary, which are typically given in counterclockwise order. Note that when the boundary points are sorted in this order, the slopes of the edges between them are also sorted. This fact will be utilized in some of the methods for computing the convex hull of a set of points.

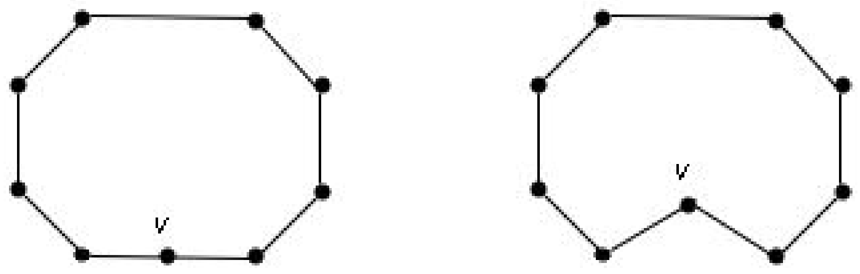
When traversing the edges on the boundary of a convex hull in $\Re^2$ it can be seen that we only make *left* turns. In fact, given 3 consecutive points $a, b, c \in \Re^2$ we can perform an **orientation test** to decide which of 3 cases the middle vertex b falls into.

reflex vertex	–	$\angle$abc makes a right turn
convex vertex	–	$\angle$abc makes a left turn
degenerate vertex	–	$\angle$abc makes a straight line

Example angles showing these three cases are given below.



It can be noted that, when we traverse the vertices in counterclockwise order, a convex polygon will only have convex vertices. Thus you will only make left turns when traversing its boundary in this order. Neither a reflex vertex nor a degenerate vertex should be found on the convex hull. That having a reflex vertex would cause the polygon to no longer be convex is clear. However, having a degenerate vertex would keep the polygon convex, but is unnecessary for specifying the boundary of the polygon. The polygon will have the exact same boundary and shape whether that degenerate vertex is included or not. Examples of polygons with reflex or degenerate vertices are given below.



Polygon with degenerate vertex *v* Polygon with reflex vertex *v*

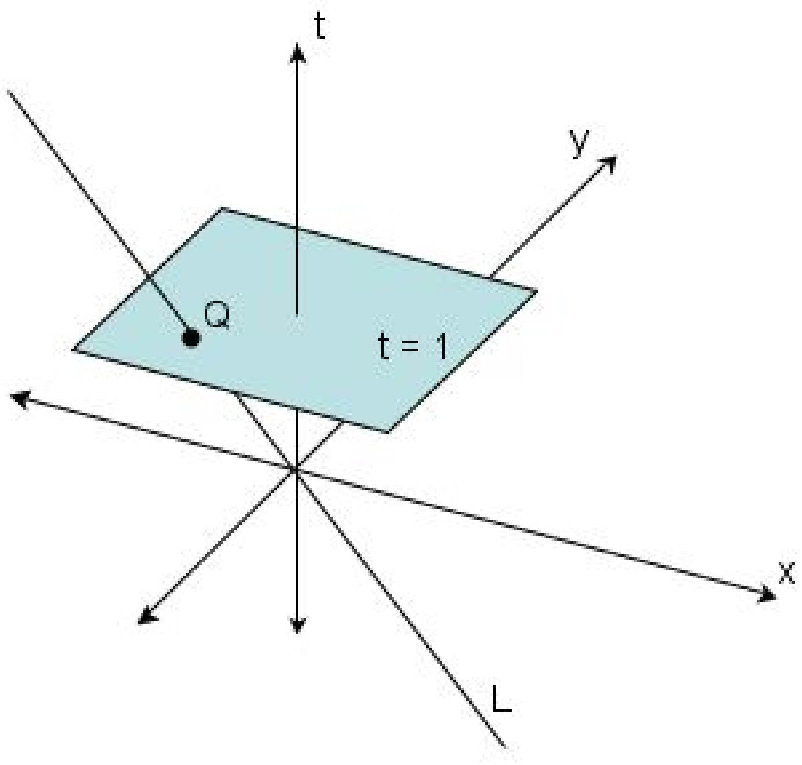
Now that we have seen the 3 possible cases for a given angle, we can now look at how an orientation test might be performed to categorize an angle formed by 3 points $a, b, c \in \Re^2$. Let $a = (a_1, a_2), b = (b_1, b_2)$, and $c = (c_1, c_2)$ be the three points comprising an angle. We then define a matrix

$$M = \begin{pmatrix} a_1 & a_2 & 1 \\ b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \end{pmatrix}$$

and perform the orientation test *orient(a,b,c)* by finding the value of the determinate of the matrix M, det(M), as follows:

$$\textit{orient(a,b,c)} = \begin{cases} \text{left (convex vertex)} & \text{if } \det(M) > 0 \\ \text{linear (degenerate vertex)} & \text{if } \det(M) = 0 \\ \text{right (reflex vertex)} & \text{if } \det(M) < 0 \end{cases}$$

Homogenous Coordinates In computing the matrix M above we used the *homogenous coordinates* for the points a, b, and c. A triple of real numbers (x, y, t) forms a set of homogenous coordinates for a point $p \in \Re^2$ with coordinates $(x/t, y/t)$. In our example, we used homogenous coordinates with a value of $t = 1$. This could possibly correspond to the point $Q = (x_0, y_0, 1)$ in the diagram below. In that diagram, a line L is drawn from the origin to Q. Any point on this line, excluding the origin, would serve as a set of homogenous coordinates for (x_0, y_0). Homogenous coordinates have many uses, typically for making certain transformation calculations easier (such as our matrix calculation above).



2.2 Algorithms For Computing 2D Convex Hulls

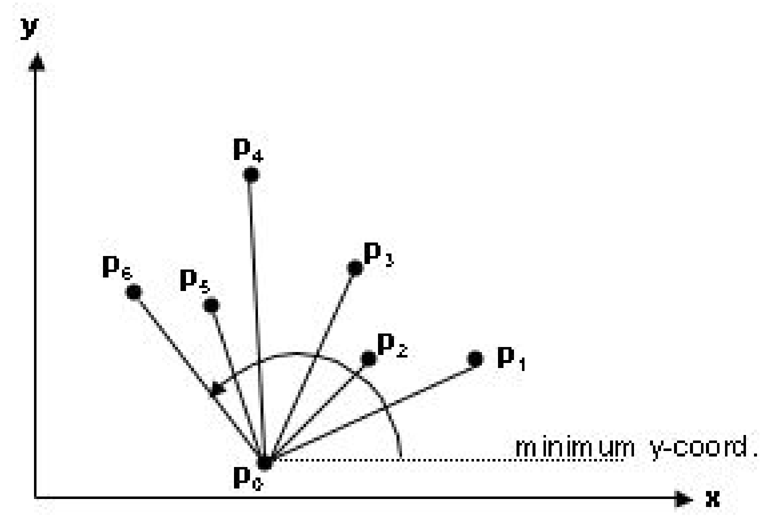
Now that we have built up all of the necessary background, we are finally ready to consider the question of actually computing a convex hull of a set of points.

Input: $S \subseteq \Re^2$, a set of n points.
Output: conv(S).

This problem commonly occurs in computational geometry. It may be calculated as a problem of its own interest, or as a set in an algorithm for solving a larger problem. One algorithm for computing a convex hull in $\Re^2$ is **Graham's Scan Algorithm**.

2.2.1 Graham's Scan Algorithm

The Graham's Scan algorithm is a common one for computing the convex hull of a set of points in $\Re^2$. Conceptually, it works as follows:



- Choose the point with the lowest y coordinate and label this point p_0.
- Sort the remaining points according to their angle with p_0 (the angle formed by a horizontal line from p_0 out to $x = \infty$ and the line segment from p_0 to the point in question). Label those sorted points p_1 to p_{n-1} in their sorted order.
- Add p_0 and p_1 to the convex hull to start.
- Visit points p_2 to p_{n-1} in that sorted order, gradually adding points to the convex hull.
- Let p_i be the point that you're visiting next. If you make a left turn to get from the last two points added into the convex hull to p_i, then add p_i to the convex hull.
- If, however, you had to make a right turn to get to p_i, keep removing the last point added to the convex hull until you do make a left turn to get to p_i.

In the above algorithm, we are only keeping points such that we keep making left turns around the boundary of the convex hull. We're traveling around the boundary in counterclockwise order because that's how we initially sorted the points p_1 to p_{n-1} with respect to their angle with p_0. Then, traveling around the boundary in this order, if we keep turning left then we have a boundary for a convex polygon. A right turn would give a non-convex shape and a straight line (no turn) would indicate a degenerate point that didn't need to be on the hull.

In generating this convex hull, we're adding to the end of a set of points and sometimes removing from that end of points. Since we're only adding or removing from one end, using a *stack* data structure is a good choice for storing our convex hull. (Recall that a stack works just like a stack of plates, adding and removing only from the top of the stack.) With our stack data structure named Σ we'll use PUSH(p_i, Σ) to denote adding p_i to the top of the stack. We'll also use POP(Σ) to denote removing the top element from the stack. Furthermore, top(Σ) and second(Σ) will denote the top and second from the top elements on the stack, respectively. While in a stack we usually can't see the second element from the top, this could easily be implemented by adding a few holding variables to go along with our stack. Finally, we are ready to give a more rigorous pseudo-code implementation for Graham's Scan algorithm.

Graham's Scan Algorithm

Input: S, a set of n points in $\Re^2$.
Output: Σ, a stack containing conv(S) with the points in counter-clockwise order.

```
$p_0 \leftarrow$ the point in $S$ with the minimum $y$-coordinate.
Sort $S$ - $\{p_0\}$ with the orientations of $\overline{p_0 p_i}$ so that they are ordered $p_1, p_2, \ldots, p_{n-1}$.
PUSH($p_0, \Sigma$)
PUSH($p_1, \Sigma$).
for ($i$ == 2) to $n - 1$ do
    while ( orient( second($\Sigma$), top($\Sigma$), $p_i$ ) $\leq 0$ ) do
        POP($\Sigma$)
    end-while
    PUSH($p_i, \Sigma$)
end-for
▷▷ Comment: $\Sigma$ now contains the sequence of points forming conv($S$) in counter clockwise order.
return $\Sigma$
```

In analyzing the running time of this algorithm, we can note two major parts: (1) sorting the points by their orientation $\overline{p_0 p_i}$, and (2) building conv(S) by traversing the points, pushing and popping as necessary. The sorting part may take up to $O(n \log n)$ time, but we will denote it as *Sort(n)* here.

In the second part, we may perform pushes and pops in many different combinations. However, any given point p_i may only be pushed once and may only be popped off at most once. Thus, we can say that this section takes $O(n)$ time. Therefore, the entire running time may be given as: ***Sort(n) + O(n)***. Furthermore, if sorting takes $O(n \log n)$ time in the worst case (in the decision tree model), then the worst case running time for Graham's Scan Algorithm is ***O(n* log *n)***.

2.2.2 Gift Wrapping Algorithm

Another notable algorithm for computing 2D convex hulls is the gift wrapping algorithm (also known as the *Jarvis March algorithm* for the 2D case). As the name suggests, we will basically be traversing or "wrapping" around the convex hull of the set of points. Unlike with Graham's Scan Algorithm, we will not ever ”remove” points from the convex hull that we're building. We will, however, still start out with the point having the minimum y-coordinate — labeled ”a” — out of the given set of points S. Note that the important fact of the starting point - with this algorithm and with the Graham's Scan algorithm - is that we start with a point that is ”extreme” in some direction. In this case, our starting point is ”extreme” in the sense that it has the minimum y-coordinate. This ensures that it will be on the convex hull. (We're ignoring the case for multiple points tying for the lowest y-coordinate).

The initial point a is first added to the convex hull (since we picked in knowing it would be included). Then, starting with this initial point a, the gift wrapping algorithm next finds a point b such that all other points in the given set S are to the ”left” of $\overline{ab}$. That is, for any other point $c \in S - \{a, b\}$ traversing a to b to c causes a left turn. Such a point b may be computed in $O(n)$ time by simply comparing the polar angles of all points in $S - \{a\}$ with respect to a.

Once the point b has been found, b is added to the convex hull and is relabeled as the new ”a”. We now repeat the process from this new point a and find a new suitable point b. This continues until we wrap back around to having b equal the first point added to the convex hull. By this time we will have fully computed the convex hull. More rigorous pseudo-code for this algorithm is given below.

Gift Wrapping Algorithm

Input: S, a set of n points in $\Re^2$.
Output: Σ, a stack containing conv(S) with the points in counter-clockwise order.

```
$p_0 \leftarrow$ the point in $S$ with the minimum $y$-coordinate.
$a \leftarrow p_0$
$\triangleright \triangleright$ Comment: For some$(p_i \neq p_0)$
$b \leftarrow p_i$
PUSH($a, \Sigma$)
**while**( $b \neq p_0$ )
    find the point $p_i \in S - \{a, b\}$ such that $\angle abp_i$ forms the largest possible angle
    PUSH($p_i, \Sigma$)
    $a \leftarrow b$
    $b \leftarrow p_i$
**end-while**
$\triangleright\triangleright$ Comment: $\Sigma$ now contains the sequence of points forming conv($S$) in counter clockwise order.
**return** $\Sigma$
```

In analyzing this algorithm we note that it runs through the *while* loop once for every point on the convex hull. If the number of these points is h, then that's h such loops. Furthermore, on every loop it has to check almost every point, taking constant time to check each one, for a total time of $O(n)$ on each loop. This leads to an overall running time of ***O(nh)***, where n is the number of points in the set S and h is the number of points on its convex hull. This running time may be reduced to $O(n \log h)$ — which is optimal under the decision tree model. We will see some techniques for how this is done in the next lecture.

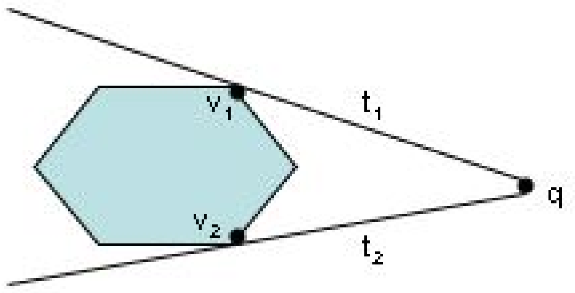
Even for the above version of the Gift Wrapping Algorithm that runs in ***O(nh)*** time it can be useful if we have a small value for h. The best possible case would then be for only 3 points forming a triangle for the convex hull. All other points would then rest inside of this triangle. Conversely, the other extreme of having all n points on the convex hull would be the worst possible case for this algorithm. You can think of such a case happening if the n points all lie on the boundary of an imaginary circle.

2.2.3 More About Convex Hulls

Polygon Containment Another useful question that we might ask is if for a given point q and convex polygon P, is $q \in P$. This can be quickly determined in $O(\log h)$ time if we already have the convex hull of P computed and properly organized. The organization required is to have the points on the convex hull divided into two subsets: the upper convex hull and the lower convex hull. Furthermore, the points in those points need to be sorted by x-coordinate within their subsets.

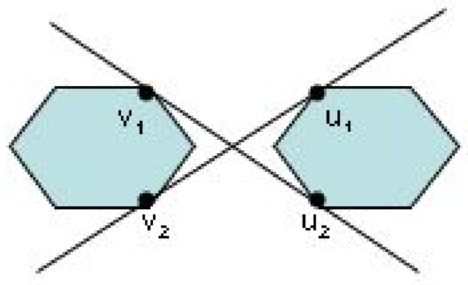
Once we have the upper and lower convex hull subsets organized, we are ready to determine if we have $q \in P$. If so, then a vertical line drawn straight up from q's should cross exactly one edge on the upper convex hull. Similarly, a vertical line drawn down from q should cross exactly one edge on the lower convex hull. We can perform binary search on the vertices in the upper hull and on the vertices in the lower hull to find if such edges exist. This takes only $O(\log h)$ time if we already have the upper and lower hulls organized.

Adding To a Polygon For a point q that is outside of the convex polygon P, we can prepare for the possible case of adding that point q to the convex hull of P. To do this, think of the point q as a light source shining onto the polygon P. The edges on the polygon that are lighted by q fall between two vertices that are called **supporting tangent vertices**. These are the outermost vertices that the two **tangent vectors** t_1 and t_2 intersect. This is shown in the diagram below. In the representation, q is the point outside of the polygon, t_1 and t_2 are the two tangent vectors and v_1 and v_2 are the supporting tangent vertices on the convex hull of the polygon. A claim about computing these tangent lines is also given below, without proof.



Claim 1 *The tangent lines t_1 and t_2 can be computed in $O(\log n)$ time by binary search.*

In addition, a similar claim may be made when we are dealing with two disjoint polygons instead of a polygon and an exterior point.



Claim 2 *Given two disjoint convex polygons, the 4 supporting tangent vertices v_1, v_2, u_1, u_2 may be found in $O(\log n)$ time.*

Again, this claim is given without proof, but it may be inferred that binary search may again be used to get this result. Once we have the ability to compute the tangent lines and supporting tangent vertices — for a point not contained in a set of points P — then we are ready to add that point to P and update the convex hull accordingly.

Insert(*q*, *P*)

Input: A point q and a polygon P.
Output: The new convex hull, conv(P).

```
if $q \notin P$
    Compute $P_L$, $P_R$ : the supporting vertices of $q$
    Discard the portion of the conv($P$) between $P_L$ and $P_R$
    Insert $q$ into conv($P$).
end-if
return conv($P$)
```

Bonus: Width of a Convex Polygon One other thing that we might like to compute about a convex polygon is the width of that polygon.

Definition 5 *The* ***width*** *of a convex polygon is the minimum distance between parallel lines of support.*

One method for computing this is the **rotating caliper algorithm**. It works by considering parallel lines drawn tangent to the polygon. If the parallel tangent lines are first horizontal (degree 0) then each is resting either on a vertex or an edge on the convex hull of the polygon. We can then rotate the angle of the parallel tangent lines and sweep through an entire 360 degree rotation. Along the way the separation between the tangent lines will change as the lines rest on different vertices/edges.

The width of the polygon is then the minimum separation between the parallel tangent lines during the sweep. Although this is a simple concept, a single picture is inadequate for demonstrating it. However, this website: *http://www-cgrl.cs.mcgill.ca/ godfried/research/calipers.html*, provides an animated image for rotating calipers as well as some more information about the algorithm and its use. In addition to calculating the width of a single convex polygon, the rotating caliper algorithm may also be modified to calculate the maximum or minimum distance between two disjoint convex polygons, among other things.