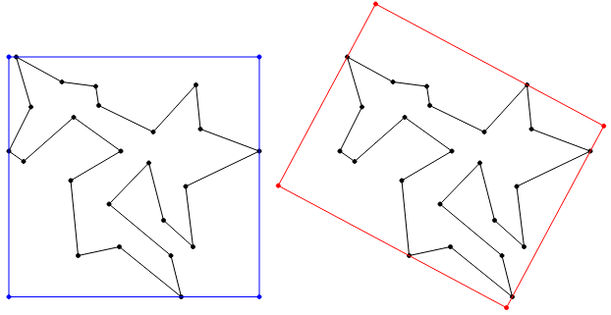
[14 Comments](https://geidav.wordpress.com/2014/01/23/computing-oriented-minimum-bounding-boxes-in-2d/#comments)

[**Computing oriented minimum bounding boxes in 2D**](https://geidav.wordpress.com/2014/01/23/computing-oriented-minimum-bounding-boxes-in-2d/)

**Introduction**

*Oriented bounding boxes* are an important tool for visibility testing and collision detection in computer graphics. In this post I want to talk about how to compute the oriented *minimum*bounding box (OMBB) of an arbitrary polygon in two dimensions. As a polygon just enforces an ordering on a set of points (vertices), everything described in the following equally applies to simple point sets. *Minimum* in this context refers to the area of the bounding box. A minimum oriented bounding box is also known as *smallest-area enclosing rectangle*. However, I will stick to the former term throughout this article as it is more frequently used in the computer graphics world.

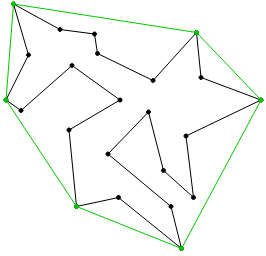
The easiest way of computing a bounding box for a polygon is to determine the minimum and maximum x– and y– coordinates of its vertices. Such an *axis aligned bounding box* (AABB) can be computed trivially but it’s in most cases significantly bigger than the polygon’s OMBB. Finding the OMBB requires some more work as the bounding box’ area must be minimized, constrained by the location of the polygon’s vertices. Look at the following figure for an illustration (AABB in blue, OMBB in red).



The technique for computing OMBBs presented in the following consists of two detached steps. In the first step the *convex hull* of the input polygon is computed. If the polygon is convex this step can be omitted because a convex polygon is equal to its convex hull. In the second step the *Rotating Calipers* method is employed on the convex hull to compute the resulting OMBB. I will focus on the Rotating Calipers method because it’s not very widely known in comparison to the numerous ways of computing convex hulls.

**Convex hulls**

In less mathematical but more illustrative terms the *convex hull* of a set of n points can be described as the closed polygonal chain of all outer points of the set, which entirely encloses all set elements. You can picture it as the shape of a rubber band stretched around all set elements. The convex hull of a set of two-dimensional points can be efficiently computed in O(n\log n). In the figure below the convex hull of the vertices of a concave polygon is depicted.



There are numerous algorithms for computing convex hulls: *Quick Hull*, *Gift Wrapping* (also known as *Jarvis March*), *Graham’s Algorithm* and some more. I’ve chosen the Gift Wrapping algorithm for my implementation because it’s easy to implement and provides good performance in case n is small or the polygon’s convex hull contains only a few vertices. The runtime complexity is O(nh), where h is the number of vertices in the convex hull. In the general case Gift Wrapping is outperformed by other algorithms. Especially, when all points are part of the convex hull. In that case the complexity degrades to O(n^2).

As there are many good articles on the Gift Wrapping algorithm available online, I won’t describe it another time here. Instead I want to focus on the lesser-known Rotating Calipers method for computing OMBBs. However, take care that your convex hull algorithm correctly handles collinear points. If multiple points lie on a convex hull edge, only the spanning points should end up in the convex hull.

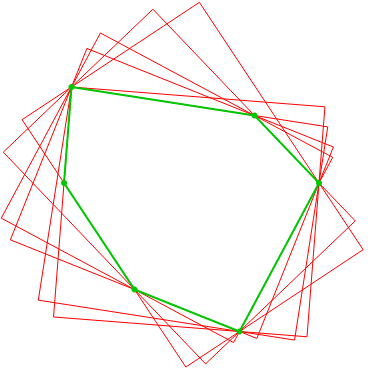
**Rotating Calipers**

*Rotating Calipers* is a versatile method for solving a number of problems from the field of computational geometry. It resembles the idea of rotating a dynamically adjustable caliper around the outside of a polygon’s convex hull. Originally, this method was invented to compute the diameter of convex polygons. Beyond that, it can be used to compute OMBBs, the minimum and maximum distance between two convex polygons, the intersection of convex polygons and many things more.

The idea of using the Rotating Calipers method for computing OMBBs is based on the following theorem, establishing a connection between the input polygon’s convex hull and the orientation of the resulting OMBB. The theorem was proven in 1975 by Freeman and Shapira[1](https://geidav.wordpress.com/tag/rotating-calipers/#fn-1057-FreemanShapira1975):

*The smallest-area enclosing rectangle of a polygon has a side collinear with one of the edges of its convex hull.*

Thanks to this theorem the number of OMBB candidates is dramatically reduced to the number of convex hull edges. Thus, the complexity of the Rotating Calipers method is *linear* if the convex hull is already available. If it isn’t available the overall complexity is bound by the cost of computing the convex hull. An example of a set of OMBB candidates (red) for a convex hull (green) is depicted in the figure below. Note, that there are as many OMBB candidates as convex hull edges and each OMBB candidate has one side flush with one edge of the convex hull.



To determine the OMBB of a polygon, first, two orthogonally aligned pairs of parallel *supporting lines*through the convex hull’s *extreme points* are created. The intersection of the four lines forms a rectangle. Next, the lines are simultaneously rotated about their supporting points until one line coincides with an edge of the convex hull. Each time an edge coincides, the four lines form another rectangle / OMBB candidate. This process is repeated until each convex hull edge once coincided with one of the four caliper lines. The resulting OMBB is the OMBB candidate with the smallest area. The entire algorithm is outlined step by step below.

1. Compute the convex hull of the input polygon.
2. Find the the extreme points p_\text{min}=(x_\text{min},y_\text{min})^T and p_\text{max}=(x_\text{max},y_\text{max})^T of the convex hull.
3. Construct two vertical supporting lines at x_\text{min} and x_\text{max} and two horizontal ones at y_\text{min} and y_\text{max}.
4. Initialize the current minimum rectangle area A_\text{min}=\infty.
5. Rotate the supporting lines until one coincides with an edge of the convex hull.
   1. Compute the area A of the current rectangle.
   2. Update the minimum area and store the current rectangle if A<A_\text{min}.
6. Repeat step 5 until all edges of the convex hull coincided once with one of the supporting lines.
7. Output the minimum area rectangle stored in step 5.2.

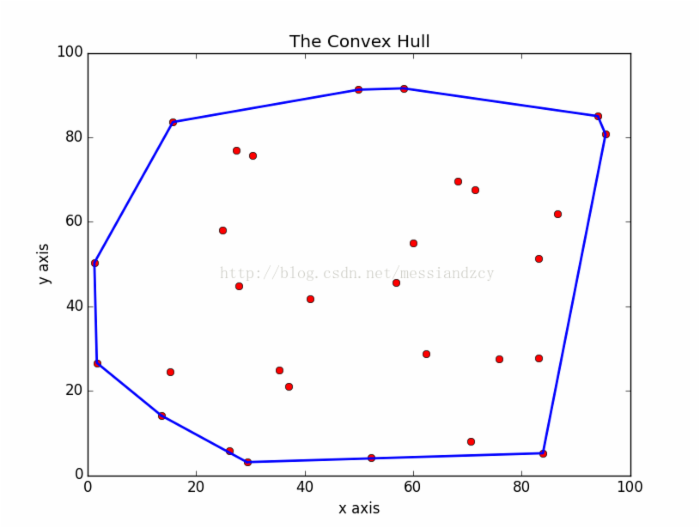
In practice, in every iteration the smallest angle \phi_\text{min} between each caliper line and its associated, following convex hull edge is determined. Then, all caliper lines are rotated at once by \phi_\text{min} and the associated convex hull edge of the caliper line enclosing the smallest angle is advanced to the next convex hull edge.

**Wrap up**

Rotating Calipers is a very elegant method for computing OMBBs in two dimensions. O’Rourke generalized it to three dimensions, yielding an algorithm of cubic runtime complexity. However, in practice approximation algorithms are used for three dimensional data because they’re usually faster. Beyond that, it's worth knowing the Rotating Calipers technique as it can be employed with minor changes to numerous other geometric problems. Depending on the programming language the implementation of the entire algorithm, including the convex hull computation, requires merely 150-200 lines of code. My sample implementation in Javascript can be found in my [github](https://github.com/geidav/ombb-rotating-calipers)repository.

1. H. Freeman, R. Shapira. *Determining the minimum-area encasing rectangle for an arbitrary closed curve*. Communications of the ACM, 18 Issue 7, July 1975, Pages 409-413

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | *# convex hull (Graham scan by x-coordinate) and diameter of a set of points*  *# David Eppstein, UC Irvine, 7 Mar 2002*  **from** \_\_future\_\_ **import** generators  **def** orientation**(**p**,**q**,**r**):**  *'''Return positive if p-q-r are clockwise, neg if ccw, zero if colinear.'''*  **return** **(**q**[1]-**p**[1])\*(**r**[0]-**p**[0])** **-** **(**q**[0]-**p**[0])\*(**r**[1]-**p**[1])**  **def** hulls**(**Points**):**  *'''Graham scan to find upper and lower convex hulls of a set of 2d points.'''*  U **=** **[]**  L **=** **[]**  Points**.**sort**()**  **for** p **in** Points**:**  **while** len**(**U**)** **>** **1** **and** orientation**(**U**[-2],**U**[-1],**p**)** **<=** **0:** U**.**pop**()**  **while** len**(**L**)** **>** **1** **and** orientation**(**L**[-2],**L**[-1],**p**)** **>=** **0:** L**.**pop**()**  U**.**append**(**p**)**  L**.**append**(**p**)**  **return** U**,**L  **def** rotatingCalipers**(**Points**):**  *'''Given a list of 2d points, finds all ways of sandwiching the points*  *between two parallel lines that touch one point each, and yields the sequence*  *of pairs of points touched by each pair of lines.'''*  U**,**L **=** hulls**(**Points**)**  i **=** **0**  j **=** len**(**L**)** **-** **1**  **while** i **<** len**(**U**)** **-** **1** **or** j **>** **0:**  **yield** U**[**i**],**L**[**j**]**    *# if all the way through one side of hull, advance the other side*  **if** i **==** len**(**U**)** **-** **1:** j **-=** **1**  **elif** j **==** **0:** i **+=** **1**    *# still points left on both lists, compare slopes of next hull edges*  *# being careful to avoid divide-by-zero in slope calculation*  **elif** **(**U**[**i**+1][1]-**U**[**i**][1])\*(**L**[**j**][0]-**L**[**j**-1][0])** **>** \  **(**L**[**j**][1]-**L**[**j**-1][1])\*(**U**[**i**+1][0]-**U**[**i**][0]):**  i **+=** **1**  **else:** j **-=** **1**  **def** diameter**(**Points**):**  *'''Given a list of 2d points, returns the pair that's farthest apart.'''*  diam**,**pair **=** max**([((**p**[0]-**q**[0])\*\*2** **+** **(**p**[1]-**q**[1])\*\*2,** **(**p**,**q**))**  **for** p**,**q **in** rotatingCalipers**(**Points**)])**  **return** pair |



S