



Chapter 4 - Bounding Volumes

Real-Time Collision Detection

by Christer Ericson

Focal Press © 2005 Citation

Recommend? ☐ yes ☐ no

Previous

Next

4.4 Oriented Bounding Boxes (OBBs)

An oriented bounding box (OBB) is a rectangular block, much like an AABB but with an arbitrary orientation. There are many possible representations for an OBB: as a collection of eight vertices, a collection of six planes, a collection of three slabs (a pair of parallel planes), a corner vertex plus three mutually orthogonal edge vectors, or a center point plus an orientation matrix and three halfedge lengths. The latter is commonly the preferred representation for OBBs, as it allows for a much cheaper OBB-OBB intersection test than do the other representations. This test is based on the separating axis theorem, which is discussed in more detail in [Chapter 5](#).

```
// Region R = { x | x = c+r*u[0]+s*u[1]+t*u[2] }, |r|<=e[0], |s|<=e[1], |t|<=e[2]
struct OBB {
    Point c;           // OBB center point
    Vector u[3];       // Local x-, y-, and z-axes
    Vector e;          // Positive halfwidth extents of OBB along each axis
};
```

At 15 floats, or 60 bytes for IEEE single-precision floats, the OBB is quite an expensive bounding volume in terms of memory usage. The memory requirements could be lowered by storing the orientation not as a matrix but as Euler angles or as a quaternion, using three to four floating-point components instead of nine. Unfortunately, for an OBB-OBB intersection test these representations must be converted back to a matrix for use in the effective separating axis test, which is a very expensive operation. A good compromise therefore may be to store just two of the rotation matrix axes and compute the third from a cross product of the other two at test time. This relatively cheap CPU operation saves three floating-point components, resulting in a 20% memory saving.

4.4.1 OBB-OBB Intersection

Unlike the previous bounding volume intersection tests, the test for overlap between two oriented bounding boxes is surprisingly complicated. At first, it seems a test to see if either box that is fully outside a face of the other would suffice. In its simplest form, this test could be performed by checking if the vertices of box *A* are all on the outside of the planes defined by the faces of box *B*, and vice versa. However, although this test works in 2D it does not work correctly in 3D. It fails to deal with, for example, the case in which *A* and *B* are almost meeting edge to edge, the edges perpendicular to each other. Here, neither box is fully outside any one face of the other. Consequently, the simple test reports them as intersecting even though they are not. Even so, the simple test may still be useful. Although it is not always correct, it is conservative in that it never fails to detect a collision. Only in some cases does it incorrectly report separated boxes as overlapping. As such, it can serve as a pretest for a more expensive exact test.

An exact test for OBB-OBB intersection can be implemented in terms of what is known as the separating axis test. This test is discussed in detail in [Chapter 5](#), but here it is sufficient to note that two OBBs are separated if, with respect to some axis *L*, the sum of their projected radii is less than the distance between the projection of their center points (as illustrated in [Figure 4.9](#)). That is, if

$|T \cdot L| > r_A + r_B.$

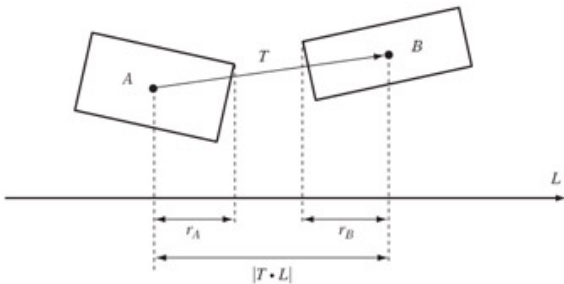


Figure 4.9: Two OBBs are separated if for some axis *L* the sum of their projected radii is less than the distance between their projected centers.

For OBBs it is possible to show that at most 15 of these separating axes must be tested to correctly determine the OBB overlap status. These axes correspond to the three coordinate axes of *A*, the three coordinate axes of *B*, and the nine axes perpendicular to an axis from each. If the boxes fail to overlap on any of the 15 axes, they are not intersecting. If no axis provides this early out, it follows that the boxes must be overlapping.

The number of operations in the test can be reduced by expressing *B* in the coordinate frame of *A*. If *t* is the translation vector from *A* to *B* and *R* = [*r*_{*ij*}] (the rotation matrix bringing *B* into *A*'s coordinate frame), the tests that must be performed for the different axes *L* are summarized in [Table 4.1](#).

Table 4.1: The 15 separating axis tests needed to determine OBB-OBB intersection. Superscripts indicate which OBB the value comes from.

L	$ T \cdot L $	r_A	r_B
u_0^A	$ t_0 $	e_0^A	$e_0^B r_{00} + e_1^B r_{01} + e_2^B r_{02} $
u_1^A	$ t_1 $	e_1^A	$e_0^B r_{10} + e_1^B r_{11} + e_2^B r_{12} $
u_2^A	$ t_2 $	e_2^A	$e_0^B r_{20} + e_1^B r_{21} + e_2^B r_{22} $
u_0^B	$ t_0 r_{00} + t_1 r_{10} + t_2 r_{20} $	$e_0^A r_{00} + e_1^A r_{10} + e_2^A r_{20} $	e_0^B
u_1^B	$ t_0 r_{01} + t_1 r_{11} + t_2 r_{21} $	$e_0^A r_{01} + e_1^A r_{11} + e_2^A r_{21} $	e_1^B
u_2^B	$ t_0 r_{02} + t_1 r_{12} + t_2 r_{22} $	$e_0^A r_{02} + e_1^A r_{12} + e_2^A r_{22} $	e_2^B
$u_0^A \times u_0^B$	$ t_2 r_{10} - t_1 r_{20} $	$e_1^A r_{20} + e_2^A r_{10} $	$e_1^B r_{02} + e_2^B r_{01} $
$u_0^A \times u_1^B$	$ t_2 r_{11} - t_1 r_{21} $	$e_1^A r_{21} + e_2^A r_{11} $	$e_0^B r_{02} + e_2^B r_{00} $
$u_0^A \times u_2^B$	$ t_2 r_{12} - t_1 r_{22} $	$e_1^A r_{22} + e_2^A r_{12} $	$e_0^B r_{01} + e_1^B r_{00} $
$u_1^A \times u_0^B$	$ t_0 r_{20} - t_2 r_{00} $	$e_0^A r_{20} + e_2^A r_{00} $	$e_1^B r_{12} + e_2^B r_{11} $
$u_1^A \times u_1^B$	$ t_0 r_{21} - t_2 r_{01} $	$e_0^A r_{21} + e_2^A r_{01} $	$e_0^B r_{12} + e_2^B r_{10} $
$u_1^A \times u_2^B$	$ t_0 r_{22} - t_2 r_{02} $	$e_0^A r_{22} + e_2^A r_{02} $	$e_0^B r_{11} + e_1^B r_{10} $
$u_2^A \times u_0^B$	$ t_1 r_{00} - t_0 r_{10} $	$e_0^A r_{10} + e_1^A r_{00} $	$e_1^B r_{22} + e_2^B r_{21} $
$u_2^A \times u_1^B$	$ t_1 r_{01} - t_0 r_{11} $	$e_0^A r_{11} + e_1^A r_{01} $	$e_0^B r_{22} + e_2^B r_{20} $
$u_2^A \times u_2^B$	$ t_1 r_{02} - t_0 r_{12} $	$e_0^A r_{12} + e_1^A r_{02} $	$e_0^B r_{21} + e_1^B r_{20} $

This test can be implemented as follows:

```

int TestOBBOBB(OBB &a, OBB &b)
{
    float ra, rb;
    Matrix33 R, AbsR;

    // Compute rotation matrix expressing b in a's coordinate frame
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            R[i][j] = Dot(a.u[i], b.u[j]);

    // Compute translation vector t
    Vector t = b.c - a.c;
    // Bring translation into a's coordinate frame
    t = Vector(Dot(t, a.u[0]), Dot(t, a.u[1]), Dot(t, a.u[2]));

    // Compute common subexpressions. Add in an epsilon term to
    // counteract arithmetic errors when two edges are parallel and
    // their cross product is (near) null (see text for details)
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            AbsR[i][j] = Abs(R[i][j]) + EPSILON;

    // Test axes L = A0, L = A1, L = A2
    for (int i = 0; i < 3; i++) {
        ra = a.e[i];

```

```

    rb = b.e[0] * AbsR[i][0] + b.e[1] * AbsR[i][1] + b.e[2] * AbsR[i][2];
    if (Abs(t[i]) > ra + rb) return 0;
}

// Test axes L = B0, L = B1, L = B2
for (int i = 0; i < 3; i++) {
    ra = a.e[0] * AbsR[0][i] + a.e[1] * AbsR[1][i] + a.e[2] * AbsR[2][i];
    rb = b.e[i];
    if (Abs(t[0] * R[0][i] + t[1] * R[1][i] + t[2] * R[2][i]) > ra + rb) return 0;
}

// Test axis L = A0 x B0
ra = a.e[1] * AbsR[2][0] + a.e[2] * AbsR[1][0];
rb = b.e[1] * AbsR[0][2] + b.e[2] * AbsR[0][1];
if (Abs(t[2] * R[1][0] - t[1] * R[2][0]) > ra + rb) return 0;

// Test axis L = A0 x B1
ra = a.e[1] * AbsR[2][1] + a.e[2] * AbsR[1][1];
rb = b.e[0] * AbsR[0][2] + b.e[2] * AbsR[0][0];
if (Abs(t[2] * R[1][1] - t[1] * R[2][1]) > ra + rb) return 0;

// Test axis L = A0 x B2
ra = a.e[1] * AbsR[2][2] + a.e[2] * AbsR[1][2];
rb = b.e[0] * AbsR[0][1] + b.e[1] * AbsR[0][0];
if (Abs(t[2] * R[1][2] - t[1] * R[2][2]) > ra + rb) return 0;

// Test axis L = A1 x B0
ra = a.e[0] * AbsR[2][0] + a.e[2] * AbsR[0][0];
rb = b.e[1] * AbsR[1][2] + b.e[2] * AbsR[1][1];

if (Abs(t[0] * R[2][0] - t[2] * R[0][0]) > ra + rb) return 0;

// Test axis L = A1 x B1
ra = a.e[0] * AbsR[2][1] + a.e[2] * AbsR[0][1];
rb = b.e[0] * AbsR[1][2] + b.e[2] * AbsR[1][0];
if (Abs(t[0] * R[2][1] - t[2] * R[0][1]) > ra + rb) return 0;

// Test axis L = A1 x B2
ra = a.e[0] * AbsR[2][2] + a.e[2] * AbsR[0][2];
rb = b.e[0] * AbsR[1][1] + b.e[1] * AbsR[1][0];
if (Abs(t[0] * R[2][2] - t[2] * R[0][2]) > ra + rb) return 0;

// Test axis L = A2 x B0
ra = a.e[0] * AbsR[1][0] + a.e[1] * AbsR[0][0];
rb = b.e[1] * AbsR[2][2] + b.e[2] * AbsR[2][1];
if (Abs(t[1] * R[0][0] - t[0] * R[1][0]) > ra + rb) return 0;

// Test axis L = A2 x B1
ra = a.e[0] * AbsR[1][1] + a.e[1] * AbsR[0][1];
rb = b.e[0] * AbsR[2][2] + b.e[2] * AbsR[2][0];
if (Abs(t[1] * R[0][1] - t[0] * R[1][1]) > ra + rb) return 0;

// Test axis L = A2 x B2
ra = a.e[0] * AbsR[1][2] + a.e[1] * AbsR[0][2];
rb = b.e[0] * AbsR[2][1] + b.e[1] * AbsR[2][0];
if (Abs(t[1] * R[0][2] - t[0] * R[1][2]) > ra + rb) return 0;

// Since no separating axis is found, the OBBs must be intersecting
return 1;
}

```

To make the OBB-OBB test as efficient as possible, it is important that the axes are tested in the order given in [Table 4.1](#). The first reason for using this order is that by testing three orthogonal axes first there is little spatial redundancy in the tests, and the entire space is quickly covered. Second, with the setup given here, where *A* is transformed to the origin and aligned with the coordinate system axes, testing the axes of *A* is about half the cost of testing the axes of *B*. Although it is not done here, the calculations of *R* and *AbsR* should be interleaved with the first three tests, so that they are not unnecessarily performed in their entirety when the OBB test exits in one of the first few *if* statements.

If OBBs are used in applications in which they often tend to have one axis aligned with the current world up, for instance, when traveling on ground, it is worthwhile special-casing these "vertically aligned" OBBs. This simplification allows for a much faster intersection test that only involves testing four separating axes in addition to a cheap test in the vertical direction.

In some cases, performing just the first 6 of the 15 axis tests may result in faster results overall. In empirical tests, [\[Bergen97\]](#) found that the last 9 tests in the OBB overlap code determine nonintersection about 15% of the time. As perhaps half of all queries are positive to start with, omitting these 9 tests results in false positives about 6 to 7% of the time. When the OBB test is performed as a pretest for an exact test on the bounded geometry, this still leaves the test conservative and no collisions are missed.

4.4.2 Making the Separating-Axis Test Robust

A very important issue overlooked in several popular treatments of the separating-axis theorem is the robustness of the test. Unfortunately, any code implementing this test must be very carefully crafted to work as intended. When a separating axis is formed by taking the cross product of an edge from each bounding box there is a possibility these edges are parallel. As a result, their cross product is the null vector, all projections onto this null vector are zero, and the sum of products on each side of the axis inequality vanishes. Remaining is the comparison $0 > 0$. In the perfect world of exact arithmetic mathematics, this expression would trivially evaluate to false. In reality, any computer implementation must deal with inaccuracies introduced by the use of floating-point arithmetic.

For the optimized inequalities presented earlier, the case of parallel edges corresponds to only the zero elements of the rotation matrix \mathbf{R} being referenced. Theoretically, this still results in the comparison $0 > 0$. In practice, however, due to accumulation of errors the rotation matrix will not be perfectly orthonormal and its zero elements will not be exactly zero. Thus, the sum of products on both sides of the inequality will also not be zero, but some small error quantity. As this accumulation of errors can cause either side of the inequality to change sign or shift in magnitude, the result will be quite random. Consequently, if the inequality tests are not very carefully performed these arithmetic errors could lead to the (near) null vector incorrectly being interpreted as a separating axis. Two overlapping OBBs therefore could be incorrectly reported as nonintersecting.

As the right-hand side of the inequalities should be larger when two OBBs are interpenetrating, a simple solution to the problem is to add a small epsilon value to the absolute values of the matrix elements occurring on the right-hand side of the inequalities. For near-zero terms, this epsilon term will be dominating and axis tests corresponding to (near) parallel edges are thus made disproportionately conservative. For other, nonzero cases, the small epsilon term will simply disappear. Note that as the absolute values of the components of a rotation matrix are bounded to the range $[0, 1]$ using a fixed-magnitude epsilon works fine regardless of the sizes of the boxes involved. The robustness of the separating-axis test is revisited in [Chapter 5](#).

4.4.3 Computing a Tight OBB

Computing tight-fitting oriented bounding boxes is a difficult problem, made worse by the fact that the volume difference between a poorly aligned and a well-aligned OBB can be quite large ([Figure 4.10](#)). There exists an algorithm for calculating the minimum volume bounding box of a polyhedron, presented in [\[O'Rourke85\]](#). The key observation behind the algorithm is that given a polyhedron either one face and one edge or three edges of the polyhedron will be on different faces of its bounding box. Thus, these edge and face configurations can be searched in a systematic fashion, resulting in an $O(n^3)$ algorithm. Although it is an interesting theoretical result, unfortunately the algorithm is both too complicated and too slow to be of much practical value.

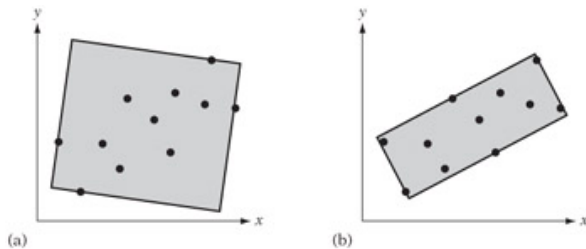


Figure 4.10: (a) A poorly aligned and (b) a well-aligned OBB.

Two other theoretical algorithms for computing near approximations of the minimum-volume bounding box are presented in [\[Barequet99\]](#). However, the authors admit that these algorithms are probably too difficult to implement and would be impractical even so, due to the large constant-factor overhead in the algorithms. Thus, with the currently available theoretical algorithms of little practical use OBBs must be computed using either approximation or brute-force methods.

A simpler algorithm offered in [\[Barequet99\]](#) provides a coarse approximation of the optimal OBB for a point set by first computing the minimum AABB of the set. From the point set, a pair of points on the two parallel box sides farthest apart are selected to determine the length direction of the OBB. The set of points is then projected onto the plane perpendicular to the OBB length direction. The same procedure is now applied again, only this time computing the minimum axis-aligned rectangle, with points on the two parallel sides farthest apart determining a second axis for the OBB. The third OBB axis is the perpendicular to the first two axes. Although this algorithm is very easy to code, in practice bounding boxes much closer to optimal can be obtained through other algorithms of similar complexity, as described in the following.

For long and thin objects, an OBB axis should be aligned with the direction of the objects. For a flat object, an OBB axis should be aligned with the normal of the flat object. These directions correspond to the principal directions of the objects, and the principal component analysis used in [Section 4.3.3](#) can be used here.

Computing bounding boxes based on covariance of model vertices generally works satisfactorily for models whose vertices are uniformly distributed over the model space. Unfortunately, the influence of internal points often bias the covariance and can make the OBB take on any orientation regardless of the extremal points. For this reason, all methods for computing bounding volumes based on weighting vertex positions should ideally be avoided. It is sufficient to note that the defining features (center, dimensions, and orientation) of a minimum bounding volume are all independent of clustering of object vertices. This can easily be seen by considering adding (or taking away) extra vertices off-center, inside or on the boundary, of a bounding volume. These actions do not affect the defining features of the volume and therefore should not affect its calculation. However, adding extra points in this manner changes the covariance matrix of the points, and consequently any OBB features directly computed from the matrix. The situation can be improved by considering just extremal points, using only those points on the convex hull of the model. This eliminates the internal points, which can no longer misalign the OBB. However, even though all remaining points are extremal the resulting OBB can still be arbitrarily bad due to point distribution. A clustering of points will still bias an axis toward the cluster. In other words, using vertices alone simply cannot produce reliable covariance matrices.

A suggested solution is to use a continuous formulation of covariance, computing the covariance across the entire face of the primitives [\[Gottschalk00\]](#). The convex hull should still be used for the calculation. If not, small outlying geometry would extend the bounding box, but not contribute enough significance to align the box properly. In addition, interior geometry would still bias the covariance. If the convex hull is already available, this algorithm is $O(n)$. If the convex hull must be computed, it is $O(n \log n)$.

Given n triangles n triangles (p_k, q_k, r_k) , $0 \leq k < n$, in the convex hull, the covariance matrix is given by

$$C_{ij} = \left(\frac{1}{a_H} \sum_{0 \leq k < n} \frac{a_k}{12} (9m_{k,i}m_{k,j} + p_{k,i}p_{k,j} + q_{k,i}q_{k,j} + r_{k,i}r_{k,j}) \right) - m_{H,i}m_{H,j}$$

Where $a_k = \|(q_k - p_k) \times (r_k - p_k)\|/2$ is the area and $m_k = (p_k + q_k + r_k)/3$ is the centroid of triangle k .

The total area of the convex hull is given by

$$a_H = \sum_{0 \leq k < n} a_k,$$

and the centroid of the convex hull,

$$m_H = \frac{1}{a_H} \sum_{0 \leq k < n} a_k m_k,$$

is computed as the mean of the triangle centroids weighted by their area. The i and j subscripts indicate which coordinate component is taken (that is, x , y , or z). Code for this calculation can be found in the publicly available collision detection package RAPID. A slightly different formulation of this covariance matrix is given in [Eberly01], and code is available on the companion CD-ROM for this book.

The method just described treats the polyhedron as a hollow body, computing the covariance matrix from the surface areas. A related method treating the polyhedron as a solid body is described in [Mirtich96a]. Given an assumed uniform density polyhedron, Mirtich's polyhedral mass property algorithm integrates over the volume of the polyhedron, computing its 3×3 inertia tensor (also known as the inertia or mass matrix). The eigenvectors of this symmetric matrix are called the principal axes of inertia, and the eigenvalues the principal moments of inertia. Just as with the covariance matrices before, the Jacobi method can be used to extract these axes, which in turn can then serve as the orientation matrix of an OBB. Detailed pseudocode for computing the inertia matrix is given in Mirtich's article. A public domain implementation in C is also available for download on the Internet. Mirtich's article is revisited in [Eberly03], in which a more computationally efficient approach is derived and for which pseudocode is provided.

Note that neither covariance-aligned nor inertia-aligned oriented bounding boxes are optimal. Consider an object A and its associated OBB B . Let A be expanded by adding some geometry to it, but staying within the bounds of B . For both methods, this would in general result in a different OBB B' for the new object A' . By construction, B and B' both cover the two objects A and A' . However, as the dimensions of the two OBBs are in the general case different, one OBB must be suboptimal.

4.4.4 Optimizing PCA-Based OBBs

As covariance-aligned OBBs are not optimal, it is reasonable to suspect they could be improved through slight modification. For instance, perhaps the OBB could be rotated about one of its axes to find the orientation for which its volume is smallest. One improved approach to OBB fitting is to align the box along just one principal component. The remaining two directions are determined from the computed minimum-area bounding rectangle of the projection of all vertices onto the perpendicular plane to the selected axis. Effectively, this method determines the best rotation about the given axis for producing the smallest-volume OBB.

This approach was suggested in [Barequet96], in which three different methods were investigated.

- All-principal-components box
- Max-principal-component box
- Min-principal-component box

The all-principal-components box uses all three principal components to align the OBB, and is equivalent to the method presented in the [previous section](#). For the max-principal-component box, the eigenvector corresponding to the largest eigenvalue is selected as the length of the OBB. Then all points are projected onto a plane perpendicular to that direction. The projection is followed by computing the minimum-area bounding rectangle of the projected points, determining the remaining two directions of the OBB. Finally, the min-principal-component box selects the shortest principal component as the initial direction of the OBB, and then proceeds as in the previous method. Based on empirical results, [Barequet96] conclude that the min-principal-component method performs best. A compelling reason max-principal-component does not do better is also given: as the maximum principal component is the direction with the maximum variance, it will contribute the longest possible edge and is therefore likely to produce a larger volume.

A local minimum volume can be reached by iterating this method on a given starting OBB. The procedure would project all vertices onto the plane perpendicular to one of the directions of the OBB, updating the OBB to align with the minimum-area bounding rectangle of the projection. The iterations would be repeated until no projection (along any direction of the OBB) gives an improvement, at which point the local minimum has been found. This method serves as an excellent optimization of boxes computed through other methods, such as Mirtich's, when performed as a preprocessing step.

Remaining is the problem of computing the minimum-area bounding rectangle of a set of points in the plane. The key insight here is a result from the field of computational geometry. It states that a minimum-area bounding rectangle of a convex polygon has (at least) one side collinear with an edge of the polygon [Freeman75].

Therefore, the minimum rectangle can trivially be computed by a simple algorithm. First, the convex hull of the point set is computed, resulting in a convex polygon. Then, an edge at a time of the polygon is considered as one direction of the bounding box. The perpendicular to this direction is obtained, and all polygon vertices are projected onto these two axes and the rectangle area is computed. When all polygon edges have been tested, the edge (and its perpendicular) giving the smallest area determines the directions of the minimum-area bounding rectangle. For each edge considered, the rectangle area is computed in $O(n)$ time, for a total complexity of $O(n^2)$ for the algorithm as a whole. This algorithm can be implemented as follows:

```

// Compute the center point, 'c', and axis orientation, u[0] and u[1], of
// the minimum area rectangle in the xy plane containing the points pt[].
float MinAreaRect(Point2D pt[], int numPts, Point2D &c, Vector2D u[2])
{
    float minArea = FLT_MAX;

    // Loop through all edges; j trails i by 1, modulo numPts
    for (int i = 0, j = numPts - 1; i < numPts; j = i, i++) {
        // Get current edge e0 (e0x,e0y), normalized
        Vector2D e0 = pt[i] - pt[j];
        e0 /= Length(e0);

        // Get an axis e1 orthogonal to edge e0
        Vector2D e1 = Vector2D(-e0.y, e0.x); // = Perp2D(e0)

        // Loop through all points to get maximum extents
        float min0 = 0.0f, min1 = 0.0f, max0 = 0.0f, max1 = 0.0f;
        for (int k = 0; k < numPts; k++) {
            // Project points onto axes e0 and e1 and keep track
            // of minimum and maximum values along both axes
            Vector2D d = pt[k] - pt[j];
            float dot = Dot2D(d, e0);
            if (dot < min0) min0 = dot;
            if (dot > max0) max0 = dot;
            dot = Dot2D(d, e1);
            if (dot < min1) min1 = dot;
            if (dot > max1) max1 = dot;
        }
        float area = (max0 - min0) * (max1 - min1);

        // If best so far, remember area, center, and axes
        if (area < minArea) {
            minArea = area;
            c = pt[j] + 0.5f * ((min0 + max0) * e0 + (min1 + max1) * e1);
            u[0] = e0; u[1] = e1;
        }
    }
    return minArea;
}

```

The minimum-area bounding rectangle of a convex polygon can also be computed in $O(n \log n)$ time, using the method of *rotating calipers* [Toussaint83]. The rotating calipers algorithm starts out bounding the polygon by four lines through extreme points of the polygon such that the lines determine a rectangle. At least one line is chosen to be coincident with an edge of the polygon. For each iteration of the algorithm, the lines are simultaneously rotated clockwise about their supporting points until a line coincides with an edge of the polygon. The lines now form a new bounding rectangle around the polygon. The process is repeated until the lines have been rotated by an angle of 90 degrees from their original orientation. The minimum-area bounding rectangle corresponds to the smallest-area rectangle determined by the lines over all iterations. The time complexity of the algorithm is bounded by the cost of computing the convex hull. If the convex hull is already available, the rotating calipers algorithm is $O(n)$.


4.4.5 Brute-Force OBB Fitting

The last approach to OBB fitting considered here is simply to compute the OBB in a brute-force manner. One way to perform brute-force fitting is to parameterize the orientation of the OBB in some manner. The space of orientations is sampled at regular intervals over the parameterization and the best OBB over all sampled rotations is kept. The OBB orientation is then refined by sampling the interval in which the best OBB was found at a higher subinterval resolution. This hill-climbing approach is repeated with smaller and smaller interval resolutions until there is little or no change in orientation for the best OBB.

For each tested coordinate system, computing the candidate OBB requires the transformation of all vertices into the coordinate system. Because this transformation is expensive, the search should exit as soon as the candidate OBB becomes worse than the currently best OBB. In that it is cheap to compute and has a relatively good fit, a PCA-fitted OBB provides a good initial guess, increasing the chances of an early out during point transformation [Miettinen02a]. To further increase the chance of an early out, the (up to) six extreme points determining the previous OBB should be the first vertices transformed. In [Miettinen02b] it is reported that over 90% of the tests are early-exited using this optimization. Brute-force fitting of OBBs generally results in much tighter OBBs than those obtained through PCA-based fitting.

The hill-climbing approach just described considers many sample points in the space of orientation before updating the currently best OBB. The optimization-based OBB-fitting method described in [Lahanas00] hill climbs the search space one sample at a time, but employs a multisample technique to aid the optimizer escape from local minima.



 **PRIVACY FEEDBACK**

Powered by **TRUSTe**