Fast tetrahedron-tetrahedron overlap algorithm

F. Ganovelli, F. Ponchio and C. Rocchini

August 11, 2002

Abstract

We present an algorithm to test two tetrahedra for overlap. The algorithm is based on a dimension reduction technique that allows to apply the Separating Axis Theorem avoiding part of the computation needed to perform the Separating Axis Test. Source code is available online.

1 Introduction

The algorithms to test overlap between two convex polyhedra made of few faces [4] work trying to find a plane that separates the vertices of the two polytopes. The Separating Axis Theorem [3, 1, 2] states that if two convex polytopes a and b do not overlap, then there exists an axis on which their projections do not overlap and such axis satisfies one of the following:

1. it is orthogonal to a face of one of the two polytopes;
2. it is orthogonal to an edge for each of the two polytopes;

The application of SAT consists of testing if the plane lying on the face of a polytope separates the two of them. If this is not the case, we should find out if there is a pair of edges, one for each polytope, such that the projections of the two polytopes on $\bar{e}_a \times \bar{e}_b$ do not overlap, where $\bar{e}_a(\bar{e}_b)$ is the vector obtained as the difference between the extremes of the edge $e_a(e_b)$.

The cost in terms of floating point operations of this procedure over a tetrahedron is easily determined as follows:

1. The cost of determining if there is a separating axis orthogonal to a face is: 18 multiplications, 8 summations, 21 subtractions, and 4 comparisons;
2. The cost of determining if there is a separating axis orthogonal to a pair of edges is: 30 floating point multiplications, 19 summations, 6 subtractions and 16 comparisons.

Since the cost of the test 2) is higher and the number of times it must be done is quadratic on the number of edges, for standard SAT is generally more efficient to do test 1) first for all faces of one polyhedron, and then do test (2) for all

pairs of edges.
The algorithm we present reduces the cost of the test 2) to 2 multiplications, 1 subtraction and 1 comparison, and so allows to change the order in which tests are performed to optimize the final performance. Source code is available at the web site listed at the end of this paper. The rest of the paper proceeds as follows: Section 2 is the core part of the paper and it describes our idea for a smart implementation of the Separating Axis Test, along with Section 3 that contains the pseudocode of the whole algorithm. Results are reported in Section 4 and a final discussion in Section 5.

2 Our approach

The main advantage of our approach, referred as **GPR** algorithm in the following, is that it does not involve explicitly testing pairs of edges to find a separating axis. Instead, reusing some of the quantities computed to find a separating plane lying on a face, we give a way to discard the possibility that a separating plane lying on a edge exists, at the small cost of 2 scalar multiplications and 1 summation (i.e. a bidimensional vector product).

The approach is based on the following observation: lets e be an edge, f_0, f_1 faces containing that edge, p_0, p_1 half-planes extending those faces,and W be the resulting wedge (which contains the tetrahedron), then if the other tetrahedron intersects W there cannot be a separating plane containing e.

The algorithm is based on three kind of tests: `Face(i)`, `Edge(f0,f1)`, and `PointInside()` explained below.

`Face(i)`
This function implements test (1) described in Section 1: it returns *true* iff the plane P_i containing the face i is a separating plane (or, equivalently, if the face normal is a separating axis). Firstly, the normal to the face i pointing outwards the tetrahedra is computed as the vector product of two vectors oriented like two edges of the face :

$$n(i) = \overline{e_0} \times \overline{e_1}$$

Then each vertex of the other tetrahedron is tested to see if it belongs to the halfspace h_i defined by the plane P_i and the normal n_i.

$$if((coord[i][j] = (n(i) \cdot (b_j - a_0)) > 0)\ masks[i]\ | =\ 2^j$$

If the test succeeds for all of the four vertices, then the plane P_i is a separating plane and the function returns *true*. Note that we store the results of this test: $masks[i]$ stores in which the side of the plane each vertex of the other tetrahedron is included (see Figure 1) and $coord[i][j]$ stores the result of the dot product.

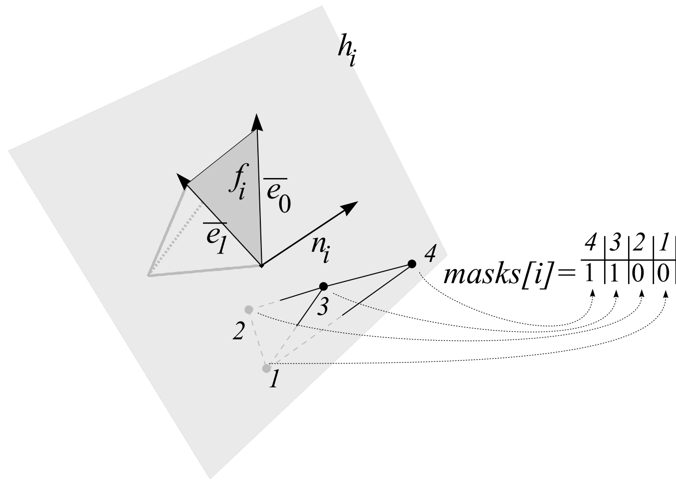

Figure 1: The vertices $1 \ldots 4$ are tested to see if they are contained in the halfspace h_i and the result is stored in $masks[i]$ as a bit

`Edge(f0,f1)`
This function implements test (2) described in Section 1 returns *true* iff there exists a separating plane containing the edge e shared by the faces $f0$ and $f1$ of the tetrahedron a.

Figure 2.(a) shows shows the application of the test for the edge e shared by faces $f0$ and $f1$ of tetrahedron a. Lets P_w be a plane orthogonal to e: a separating plane containing e exists iff there exists a line passing through point e' (projection of e on P_w) separating the projections of the two tetrahedra a and b.

This observation allows us to reduce dimension of the problem by 1, projecting everything on P_w (see Figure 2.(b)).

We can express points of P_w in a coordinate system centered in e' with axes n_0 and n_1 orthogonal respectively to f_0 and f_1 (see Figure 2.(c)).

Coordinates of the projection ${b_j}'$ of a vertex of b will be $(b_j \cdot n_0, b_j \cdot n_1)$ and we find them conveniently stored in $coord[i][j], i = f_0, f_1 \ and \ j = 1 \ldots 4$.
Furthermore, the values $masks[j], j = 1 \ldots 4$ tell us in which quadrant the vertex ${b_j}'$ is.

Now all we need to test is that no point of b' lies in the quadrant $(-,-)$, as this will grant the existence of a separating line (and viceversa). It will be enough to test that no vertex b_i lies in $(-,-)$ and that no edge ${b_i}', {b_j}'$ intersect $(-,-)$.
Since edges having one or two extremes in $(+,+)$ cannot intersect $(-,-)$, we

can use the masks computed in $FaceA(f0)$ and $FaceA(f1)$ to perform a quick rejection test:

```
int maskf0 = masks[f0];
int maskf1 = masks[f1];
if( (maskf0 | maskf1) != 017) // if there is a vertex of b
    return false;             // included in (-,-) return false
```

Next, we exclude vertices in (+,+):

```
maskf0 & = (maskf0 ^ maskf1); // exclude the vertices in (+,+)
maskf1 & = (maskf0 ^ maskf1);
```

For the edges having an extreme in $(-,+)$ and the other in $(+,-)$ the test is a mere bidimensional vector product ${b_i}' \times {b_j}'$ (see Figure 2.(c)).For edge $0 - -1$ of b, we test:

```
if( ((masks[f0] && 001) &&    // the vertex 0 of b is in (-,+)
    (mask[f1] & 002)) &&      // the vertex 1 of b is in (+,-)
    ( ((coord[f0][1] * coord[f1][0]) -
    (coord[f0][0] * coord[f1][1]))>0 ) )
                              // the edge (0,1) crosses (-,-)
      return false;
```

and the other edges of b, 0–2, 0–3, 1–2, 1–3 and 2–3, are tested similarly.

`PointInside()`
PointInside() returns *true* if at least a vertex of of b is inside the tetrahedron a. It is performed after the tests on the faces have computed the values of the array *masks* and it consists of the following row:

$$if(masks[0] \mid masks[1] \mid masks[2] \mid masks[3] != \ 1111) \ return \ true;$$

If the test succeeds than at least a bit is zero in every masks, that means that the corresponding vertex is inside all of the four halfspaces.

3 Overall algorithm

The algorithm is straightforward:

```
bool algorithm tet_a_tet(Tetrahedron A, Tetrahedron B)
{
   if (FaceA(0)) then return false
```

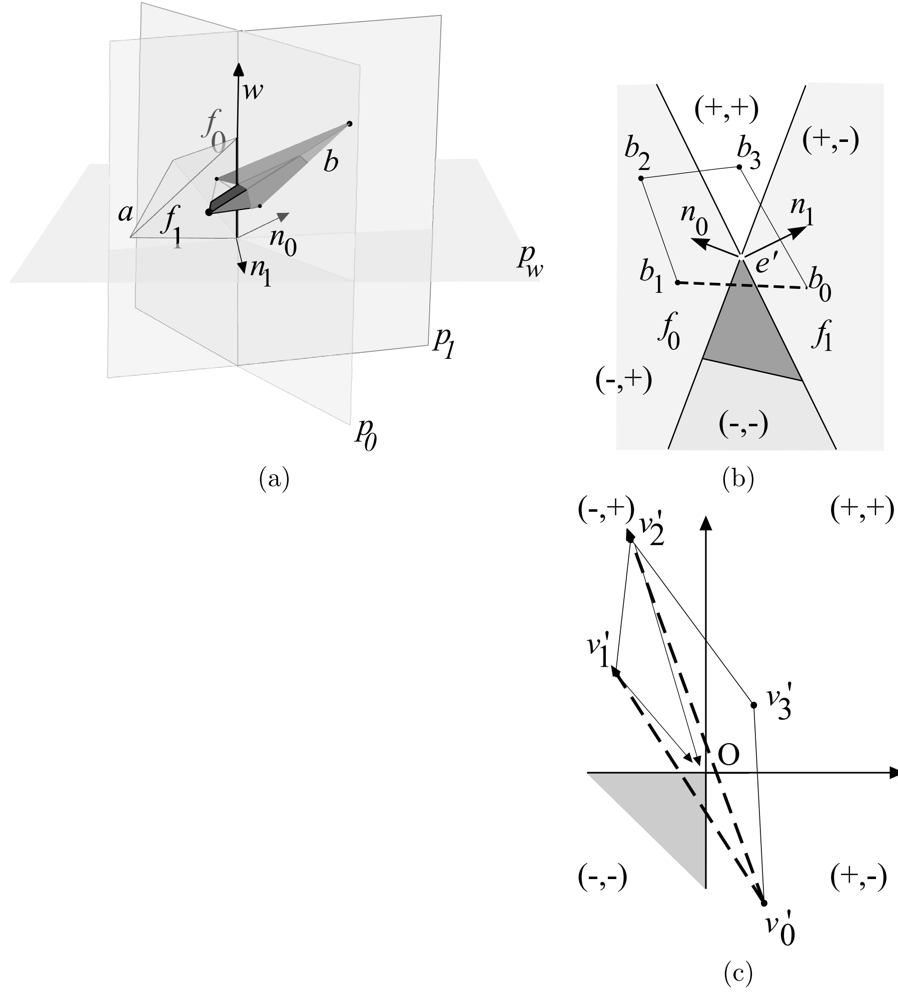

Figure 2: (a) If the tetrahedra a and b overlap, so do their projections on P_w (b) the same scene as (a) seen from a point along the w axis; (c) Since $b'_0 \times b'_2 > 0$, $b'_0 \rightarrow b'_2 \rightarrow O$ is a left turn and $\overline{b'_0 b'_2}$ does not intersect the quadrant (-,-). Conversely, $b'_0 \times b'_1 < 0$, $v'_0 \rightarrow v'_1 \rightarrow O$ is a right turn and $\overline{b'_0 b'_1}$ intersects the quadrant (-,-)

```
   if (FaceA(1)) then return false

   if (EdgeA(0,1)) then return false

   if (FaceA(2)) then return false
   if (EdgeA(0,2)) then return false
   if (EdgeA(1,2)) then return false

   if (FaceA(3)) then return false

   if (EdgeA(0,3)) then return false
   if (EdgeA(1,3)) then return false
   if (EdgeA(2,3)) then return false
   if (PointInside()) return true

   if (FaceB(0)) then return false
   if (FaceB(1)) then return false
   if (FaceB(2)) then return false
   if (FaceB(3)) then return false
   return true
}
```

Note that the `EdgeA(i,j)` is performed as soon as `FaceA(i)` and `FaceA(j)` have been performed, so that the values of *coords* and *masks* for i and j are known. Also note that no tests on the edges of the tetrahedron b are done, because if there existed a separating axis $\bar{e}_a \times \bar{e}_b$, then there would exist a separating plane laying on an edge of a and after the last `EdgeA()` it is already known that there cannot be such a plane.
The test `PointInside()` performs a trivial accept based on already-computed values, to avoid the remaining computations; it test if at least one of the vertices of tetrahedron b is inside tetrahedron a and could be omitted with no effect on the correctness.

4 Results

We compare our algorithm with the standard Separating Axis Test algorithm, where the tests on the separating axis orthogonal to a pair of edges are left after the tests on the faces.

The test-bed consists of testing overlap of two tetrahedra randomly generated. The random generation of a tetrahedron is done by generating its four vertices in a unit cube. In the first experiment such a cube is the same for both tetrahedra; in the second the two cubes share a face and in the third one are far apart. This subdivision is done in order to test our algorithm in different circumstances: when the tetrahedra overlap, when they do not overlap but are close to each other and when they are far away. Table 1 reports the results of

	set 1	set 2	set 3
time GPR	29.1	17.1	13.4
time SAT	101.2	19.4	13.0
#faces GPR	2.41	2.14	1.74
#faces SAT	3.34	2.62	1.75

Table 1: Results: the first two rows report the number of milliseconds to perform 10^4 overlap tests; the second two rows the average number of face tested. In set 1, the tetrahedra are likely to overlap, in set 2 they are close but on overlapping, and in set 3 they are far away from each other.

these tests in milliseconds over 10^4 pairs of tetrahedra to test (Intel Pentium IV 1Ghz). You can see how GPR algorithm is much faster when the two tetrahedra are likely to collide, while its performance is substantially the same as SAT algorithm in the other conditions. This because if the tetrahedra are far apart, then the separating plane will (almost) always lay on a face and it will be found without testing pairs of edges. As a consequence the two algorithms perform the same operations with the difference that GPR pays the overhead of storing the result of the tests on the faces. On the contrary, when the tetrahedra are close to each other, the GPR algorithm tests less faces of SAT, thanks to the fact that testing for separating planes lying on edges is almost negligible and it can be done earlier in the algorithm.

5 Discussion

This paper gives a fast overlap test between two tetrahedra. No divisions are involved and the only errors that can arise are the ones given by scalar products and subtractions. If the first tetrahedron is flat, i.e. its four vertices lay on a plane, then it is possible to get a wrong answer. In particular the coordinate system given by the normals of two faces with opposite orientation is degenerate and any point on a line perpendicular to the axes is transformed on the same point in the affine coordinate system (see Figure 3.(a)). The more, all of the points will be on the same straight line passing through the origin of the affine coordinate system (see Figure 3.(b)). This means that no meaningful answer is given in this case.

Finally we stress that the **GPR** algorithm is mainly based on the dimensional reduction technique that offers the way to save operations at the price of storing some more data. The idea we presented can be used as it is with general convex polygons, as long as the overhead caused by accessing memory to store such data is not greater than the time saved. This of course depends on the hardware architecture, so we cannot say *a priori* the number of faces the polygons must have in order to be convenient to use our algorithm, but we empirically estimate such number to be under 10.

Web Information

Source code for the GPR algorithm is available at
http://www.acm.org/jgt/papers/GanovelliPonchioRocchini02

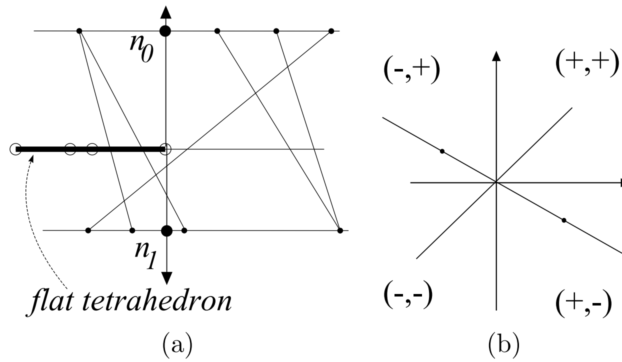

Figure 3: (a) Degenerate case: all of the points project on the same on the affine coordinates system (b) The projection of any edge in affine coordinate system pass through the origin.

References

[1] Tomas Akenine-Moller. Fast 3d triangle-box overlap testing. *JGT*, 2(2):25–30, 1997.

[2] Van Den Bergen G. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1998.

[3] S. Gottschalk, M. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[4] Rich Rabbitz. Fast collision detection of moving convex polyhedra. In Paul Heckbert, editor, *Graphics Gems IV*, pages 83–109. Academic Press, Boston, 1994.