

SAT (Separating Axis Theorem)

Posted on *January 1, 2010*

This is a post I have been meaning to do for some time now but just never got around to it. Let me first start off by saying that there are a ton of resources on the web about this particular collision detection algorithm. The problem I had with the available resources is that they are often vague when explaining some of the implementation details (probably for our benefit).

I plan to explain the algorithm and also fill in some of the blanks that I had when implementing this myself.

First let me start off by saying there is a great tutorial here with interactive flash examples.

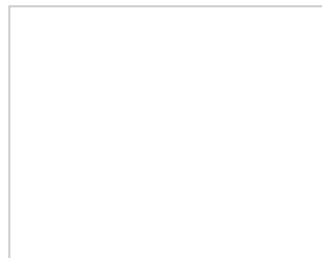
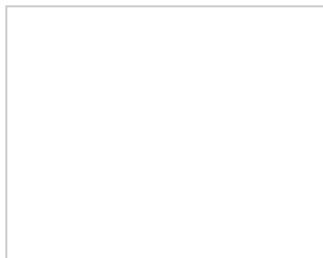
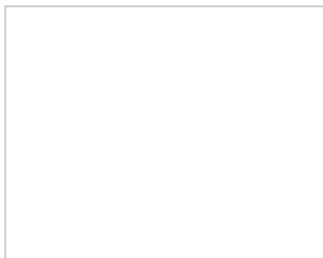
1. Introduction
2. Convexity
3. Projection
4. Algorithm
 1. No Intersection
 2. Intersection
5. Obtaining The Separating Axes
6. Projecting A Shape Onto An Axis
7. Finding the MTV
8. Curved Shapes
9. Containment
10. Other Things To Note

Introduction

The Separating Axis Theorem, SAT for short, is a method to determine if two convex shapes are intersecting. The algorithm can also be used to find the minimum penetration vector which is useful for physics simulation and a number of other applications. SAT is a fast generic algorithm that can remove the need to have collision detection code for each shape type pair thereby reducing code and maintenance.

Convexity

SAT, as stated before, is a method to determine if two convex shapes are intersecting. A shape is considered convex if, for any line drawn through the shape, that line crosses only twice. If a line can be drawn through the shape and cross more than twice the shape is non-convex (or concave). See Wiki's definition and MathWorld's definition for more mathematical and formal definitions. So lets look at some examples:



The first shape is considered convex because there does not exist a line that can be drawn through the shape

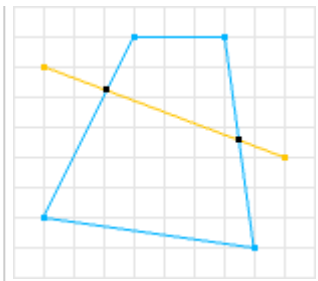


Figure 1: A Convex Shape

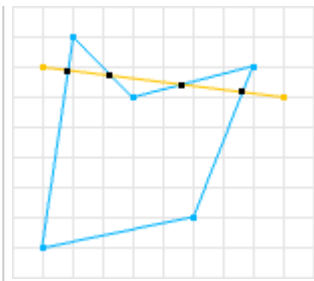


Figure 2: A Non-Convex Shape

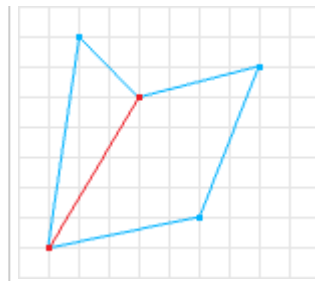


Figure 3: A Convex Decomposition

where it will cross more than twice. The second shape is not convex because there does exist a line that crosses more than twice.

SAT can only handle convex shapes, but this is

OK because non-convex shapes can be represented by a combination of convex shapes (called a convex decomposition). So if we take the non-convex shape in figure 2 and perform a convex decomposition we can obtain two convex shapes. We can then test each convex shape to determine collision for the whole shape.

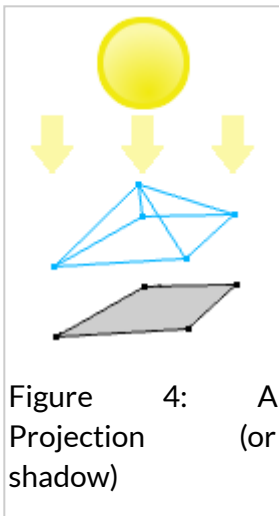


Figure 4: A Projection shadow (or)

Projection

The next concept that SAT uses is projection. Imagine that you have a light source whose rays are all parallel. If you shine that light at an object it will create a shadow on a surface. A shadow is a two dimensional projection of a three dimensional object. The projection of a two dimensional object is a one dimensional "shadow".

Algorithm

SAT states that: **"If two convex objects are not penetrating, there exists an axis for which the projection of the objects will not overlap."**

No Intersection

First let's discuss how SAT determines two shapes are not intersecting. In figure 5 we know that the two shapes are not intersecting. A line is drawn between them to illustrate this.

If we choose the perpendicular line to the line separating the two shapes in figure 5, and project the shapes onto that line we can see that there is no overlap in their projections. A line where the projections (shadows) of the shapes do not overlap is called a separation axis. In figure 6 the dark grey line is a separation axis and the respective colored lines are the projections of the shapes onto the separation axis. Notice in figure 6 the projections are not overlapping, therefore according to SAT the shapes are not intersecting.

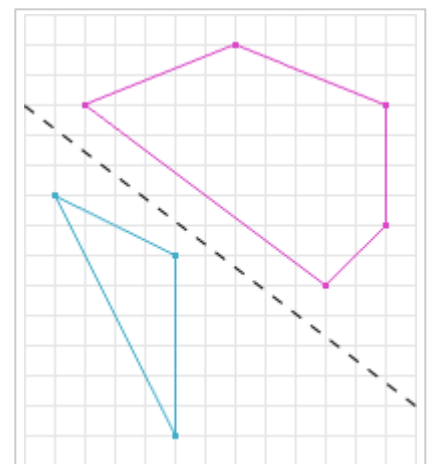


Figure 5: Two Separated Convex Shapes

SAT may test many axes for overlap, however, the first axis where the projections are not overlapping, the

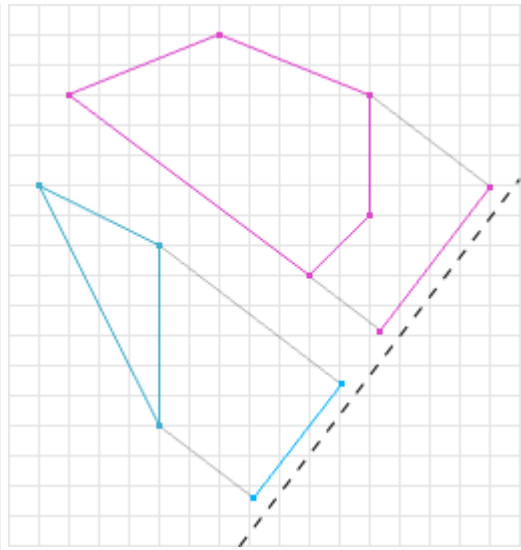


Figure 6: Two Separated Convex Shapes With Their Respective Projections

algorithm can

immediately exit determining that the shapes are not intersecting. Because of this early exit, SAT is ideal for applications that have many objects but few collisions (games, simulations, etc).

To explain a little further, examine the following psuedo code.

```

1 Axis[] axes = // get the axes to test;
2 // loop over the axes
3 for (int i = 0; i < axes.length; i++) {
4     Axis axis = axes[i];
5     // project both shapes onto the axis
6     Projection p1 = shape1.project(axis);
7     Projection p2 = shape2.project(axis);
8     // do the projections overlap?
9     if (!p1.overlap(p2)) {
10         // then we can guarantee that the shapes do not overlap
11         return false;
12     }
13 }

```

Intersection

If, for all axes, the shape's projections overlap, then we can conclude that the shapes are intersecting. Figure 7 illustrates two convex shapes being tested on a number of axes. The projections of the shapes onto those axes all overlap, therefore we can conclude that the shapes are intersecting.

All axes must be tested for overlap to determine intersection. The modified code from above is:

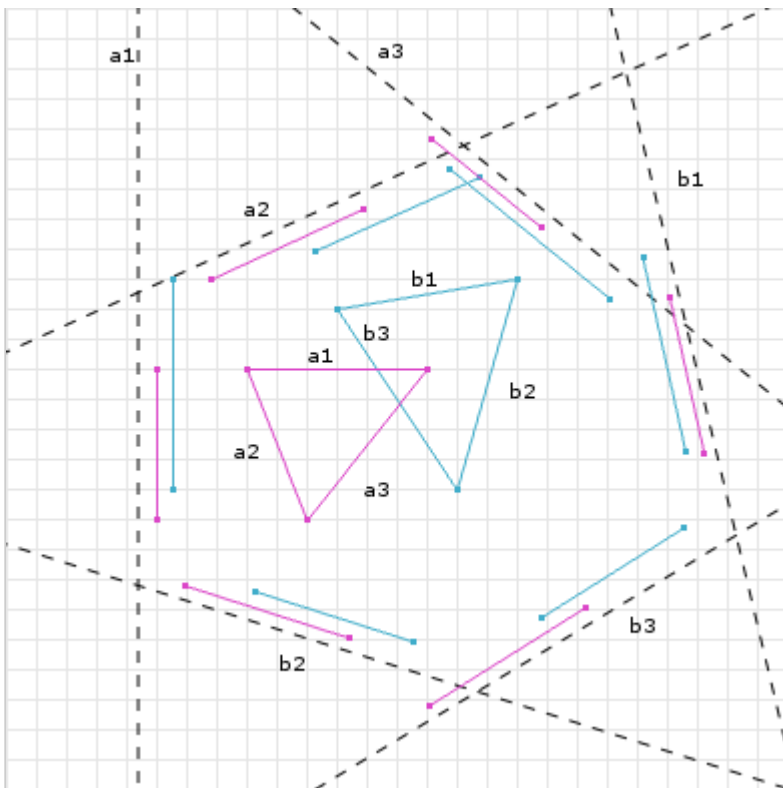


Figure 7: Two Convex Shapes Intersecting

```

1 Axis[] axes = // get the axes to test;
2 // loop over the axes
3 for (int i = 0; i < axes.length; i++) {
4     Axis axis = axes[i];
5     // project both shapes onto the axis
6     Projection p1 = shape1.project(axis);
7     Projection p2 = shape2.project(axis);
8     // do the projections overlap?
9     if (!p1.overlap(p2)) {
10         // then we can guarantee that the shapes do not overlap
11         return false;
12     }
13 }
14 // if we get here then we know that every axis had overlap on it
15 // so we can guarantee an intersection
16 return true;

```

Obtaining The Separating Axes

The first question I had when implementing this algorithm was how do I know what axes to test? This actually turned out to be pretty simple:

The axes you must test are the normals of each shape's edges.

The normals of the edges can be obtained by flipping the coordinates and negating one. For example:

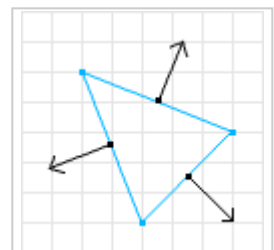


Figure 8: Edge Normals

```

1 Vector[] axes = new Vector[shape.vertices.length];
2 // loop over the vertices
3 for (int i = 0; i < shape.vertices.length; i++) {
4     // get the current vertex
5     Vector p1 = shape.vertices[i];
6     // get the next vertex
7     Vector p2 = shape.vertices[i + 1 == shape.vertices.length ? 0 : i + 1];
8     // subtract the two to get the edge vector
9     Vector edge = p1.subtract(p2);
10    // get either perpendicular vector
11    Vector normal = edge.perp();
12    // the perp method is just (x, y) => (-y, x) or (y, -x)
13    axes[i] = normal;
14 }

```

*In the method above we return the perpendicular vector to each edge of the shape. These vectors are called “normal” vectors. These vectors are **not** normalized however (not of unit length). If you need only a boolean result from the SAT algorithm this will suffice, but if you need the collision information (which is discussed later in the MTV section) then these vectors will need to be normalized (see the Projecting A Shape Onto An Axis section).*

Perform this for each shape to obtain two lists of axes to test. Doing this changes the pseudo code from above to:

```

1 Axis[] axes1 = shape1.getAxes();
2 Axis[] axes2 = shape2.getAxes();
3 // loop over the axes1
4 for (int i = 0; i < axes1.length; i++) {
5     Axis axis = axes1[i];
6     // project both shapes onto the axis
7     Projection p1 = shape1.project(axis);
8     Projection p2 = shape2.project(axis);
9     // do the projections overlap?
10    if (!p1.overlap(p2)) {
11        // then we can guarantee that the shapes do not overlap
12        return false;
13    }
14 }
15 // loop over the axes2
16 for (int i = 0; i < axes2.length; i++) {
17     Axis axis = axes2[i];
18     // project both shapes onto the axis
19     Projection p1 = shape1.project(axis);
20     Projection p2 = shape2.project(axis);
21     // do the projections overlap?
22     if (!p1.overlap(p2)) {
23         // then we can guarantee that the shapes do not overlap
24         return false;
25     }
26 }
27 // if we get here then we know that every axis had overlap on it
28 // so we can guarantee an intersection
29 return true;

```

Projecting A Shape Onto An Axis

Another thing that wasn't clear was how to project a shape onto an axis. To project a polygon onto an axis is relatively simple; loop over all the vertices performing the dot product with the axis and storing the minimum and maximum.

```
1 double min = axis.dot(shape.vertices[0]);
2 double max = min;
3 for (int i = 1; i < shape.vertices.length; i++) {
4     // NOTE: the axis must be normalized to get accurate projections
5     double p = axis.dot(shape.vertices[i]);
6     if (p < min) {
7         min = p;
8     } else if (p > max) {
9         max = p;
10    }
11 }
12 Projection proj = new Projection(min, max);
13 return proj;
```

Finding the MTV

So far we have only been returning true or false if the two shapes are intersecting. In addition to this, SAT can return a Minimum Translation Vector (MTV). The MTV is the minimum magnitude vector used to push the shapes out of the collision. If we refer back to figure 7 we can see that axis C has the smallest overlap. That axis and that overlap is the MTV, the axis being the vector portion, and the overlap being the magnitude portion.

To determine if the shapes are intersecting we must loop over all the axes from both shapes, so at the same time we can keep track of the minimum overlap and axis. If we modify our pseudo code from above to include this we can return a MTV when the shapes intersect.

```

1  double overlap = // really large value;
2  Axis smallest = null;
3  Axis[] axes1 = shape1.getAxes();
4  Axis[] axes2 = shape2.getAxes();
5  // loop over the axes1
6  for (int i = 0; i < axes1.length; i++) {
7      Axis axis = axes1[i];
8      // project both shapes onto the axis
9      Projection p1 = shape1.project(axis);
10     Projection p2 = shape2.project(axis);
11     // do the projections overlap?
12     if (!p1.overlap(p2)) {
13         // then we can guarantee that the shapes do not overlap
14         return false;
15     } else {
16         // get the overlap
17         double o = p1.getOverlap(p2);
18         // check for minimum
19         if (o < overlap) {
20             // then set this one as the smallest
21             overlap = o;
22             smallest = axis;
23         }
24     }
25 }
26 // loop over the axes2
27 for (int i = 0; i < axes2.length; i++) {
28     Axis axis = axes2[i];
29     // project both shapes onto the axis
30     Projection p1 = shape1.project(axis);
31     Projection p2 = shape2.project(axis);
32     // do the projections overlap?
33     if (!p1.overlap(p2)) {
34         // then we can guarantee that the shapes do not overlap
35         return false;
36     } else {
37         // get the overlap
38         double o = p1.getOverlap(p2);
39         // check for minimum
40         if (o < overlap) {
41             // then set this one as the smallest
42             overlap = o;
43             smallest = axis;
44         }
45     }
46 }
47 MTV mtv = new MTV(smallest, overlap);
48 // if we get here then we know that every axis had overlap on it
49 // so we can guarantee an intersection
50 return mtv;

```

Curved Shapes

We have seen how polygons can be tested using SAT, but what about curved shapes like a circle? Curved shapes pose a problem for SAT because curved shapes have an infinite number of separating axes to test. The way this problem is usually solved is by breaking up the Circle vs Circle and Circle vs Polygon tests and doing some more specific work. Another alternative is to not use curved shapes at all and replace them with high vertex count polygons. The second alternative requires no change to the above pseudo code, however I do want to cover the first option.

Let's first look at Circle vs Circle. Normally you would do something like the following:

```

1 Vector c1 = circle1.getCenter();
2 Vector c2 = circle2.getCenter();
3 Vector v = c1.subtract(c2);
4 if (v.getMagnitude() < circle1.getRadius() + circle2.getRadius()) {
5     // then there is an intersection
6 }
7 // else there isnt

```

We know two circles are colliding if the centers are closer than the sum of the circle's radii. This test is actually a SAT like test. To achieve this in SAT we could do the following:

```

1 Vector[] axes = new Vector[1];
2 if (shape1.isCircle() && shape2.isCircle()) {
3     // for two circles there is only one axis test
4     axes[0] = shape1.getCenter().subtract(shape2.getCenter());
5 }
6 // then all the SAT code from above

```

Circle vs Polygon poses more of a problem. The center to center test along with the polygon axes is not enough (In fact the center to center test can be omitted). For this case you must include another axis: the axis from the closest vertex on the polygon to the circle's center. The closest vertex on the polygon can be found in a number of ways, the accepted solution using Voronoi regions which I will not discuss in this post.

Other curved shapes are going to be even more of a problem and must be handled in their own way. For instance a capsule shape could be decomposed into a rectangle and two circles.

Containment

One of the problems that many developers choose to ignore is containment. What happens when a shape contains another shape? This problem is usually not a big deal since most applications will never have this situation happen. First let me explain the problem and how it can be handled. Then I'll explain why it should be considered.

If one shape is contained in another shape SAT, given the pseudo code we have so far, will return an incorrect MTV. Both the vector and magnitude portions may not be correct. Figure 9 shows that the overlap returned is not enough to move the shapes out of intersection. So what we need to do is check for containment in the overlap test. Taking just the if statement from the above SAT code:

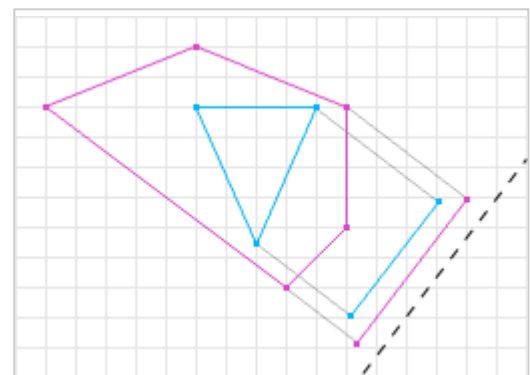


Figure 9: Containment


```

1  if (!p1.overlap(p2)) {
2      // then we can guarantee that the shapes do not overlap
3      return false;
4  } else {
5      // get the overlap
6      double o = p1.getOverlap(p2);
7      // check for containment
8      if (p1.contains(p2) || p2.contains(p1)) {
9          // get the overlap plus the distance from the minimum end points
10         double mins = abs(p1.min - p2.min);
11         double maxs = abs(p1.max - p2.max);
12         // NOTE: depending on which is smaller you may need to
13         // negate the separating axis!!
14         if (mins < maxs) {
15             o += mins;
16         } else {
17             o += maxs;
18         }
19     }
20     // check for minimum
21     if (o < overlap) {
22         // then set this one as the smallest
23         overlap = o;
24         smallest = axis;
25     }
26 }

```

Reason #1: It IS possible that the shapes could get in this type of configuration. Not handling this would require two or more iterations of SAT to resolve the collision depending on the relative sizes of the shapes.

Reason #2: If you plan to support Line Segment vs. Other shapes you have to do this because the overlap can be zero in some cases (this is due to the fact that a Line Segment is an infinitely thin shape).

Other Things To Note

Some other things to note:

- The number of axes to test can be reduced by not testing parallel axes. This is why a rectangle only has two axes to test.
- Some shapes like a rectangle can perform faster if it has its own projection and getAxes code since a rectangle doesn't need to test 4 axes but really just 2.
- The last separation axis could be used to prime the next iteration of SAT so that the algorithm could be $O(1)$ in non-intersection cases.
- SAT in 3D can end up testing LOTS of axes.
- I'm not an expert and please excuse my terrible graphics.

This entry was posted in *Blog*, *Collision Detection*, *Game Development* and tagged *Collision Detection*, *Game Development*, *SAT*. Bookmark the *permalink*.

GJK (Gilbert-Johnson-Keerthi) →