
SAT.js

- Classes
- Collision Tests
- Examples

About

SAT.js is a simple JavaScript library for performing collision detection (and projection-based collision response) of simple 2D shapes. It uses the Separating Axis Theorem (hence the name)

It supports detecting collisions between: - Circles (using Voronoi Regions.) - Convex Polygons (and simple Axis-Aligned Boxes, which are of course, convex polygons.)

It also supports checking whether a point is inside a circle or polygon.

It's released under the MIT license.

Current version: 0.9.0.

Nicely compresses with the Google Closure Compiler in **Advanced** mode to about 6KB (2KB gzipped)

To use it in node.js, you can run `npm install sat` and then use it with `var SAT = require('sat');`

Classes

SAT.js contains the following JavaScript classes:

SAT.Vector (aliased as SAT.V)

This is a simple 2D vector/point class. It is created by calling:

```
1 // Create the vector (10,10) - If (x,y) not specified, defaults to (0,0).
2 var v = new SAT.Vector(10, 10)
```

It has the following properties:

- `x` - The x-coordinate of the Vector.
- `y` - The y-coordinate of the Vector.

It contains the following methods:

- `copy(other)` - Copy the value of another Vector to this one.
- `clone()` - Return a new vector with the same coordinates as this one.
- `perp()` - Change this vector to be perpendicular to what it was before.
- `rotate(angle)` - Rotate this vector counter-clockwise by the specified number of radians.
- `reverse()` - Reverse this Vector.
- `normalize()` - Make the Vector unit-lengthed.
- `add(other)` - Add another Vector to this one.
- `sub(other)` - Subtract another Vector from this one.
- `scale(x,y)` - Scale this Vector in the X and Y directions.
- `project(other)` - Project this Vector onto another one.
- `projectN(other)` - Project this Vector onto a unit Vector.
- `reflect(axis)` - Reflect this Vector on an arbitrary axis Vector.
- `reflectN(axis)` - Reflect this Vector on an arbitrary axis unit Vector.
- `dot(other)` - Get the dot product of this Vector and another.
- `len2()` - Get the length squared of this Vector.
- `len()` - Get the length of this Vector

SAT.Circle

This is a simple circle with a center position and a radius. It is created by calling:

```
1 // Create a circle whose center is (10,10) with radius of 20
2 var c = new SAT.Circle(new SAT.Vector(10,10), 20);
```

It has the following properties:

- `pos` - A Vector representing the center of the circle.
- `r` - The radius of the circle
- `offset` - Offset of center of circle from `pos`.

It has the following methods:

- `setOffset(offset)` - Set the current offset
- `getAABB()` - Compute the axis-aligned bounding box. Returns a new Polygon every time it is called.
- `getAABBAsBox()` - Compute the axis-aligned bounding box. Returns a new Box every time it is called.

SAT.Polygon

This is a **convex** polygon, whose points are specified in a counter-clockwise fashion. It is created by calling:

```
1 // Create a triangle at (0,0)
2 var p = new SAT.Polygon(new SAT.Vector(), [
3     new SAT.Vector(),
4     new SAT.Vector(100,0),
5     new SAT.Vector(50,75)
6 ]);
```

Note: The points are counter-clockwise *with respect to the coordinate system*. If you directly draw the points on a screen that has the origin at the top-left corner it will *appear* visually that the points are being specified clockwise. This is just because of the inversion of the Y-axis when being displayed.

You can create a line segment by creating a [Polygon](#) that contains only 2 points.

Any identical consecutive points will be combined. (this can happen if you convert a [Box](#) with zero width or height into a [Polygon](#))

It has the following properties:

- [pos](#) - The position of the polygon (all points are relative to this).
- [points](#) - Array of vectors representing the original points of the polygon.
- [angle](#) - Angle to rotate the polygon (affects [calcPoints](#))
- [offset](#) - Translation to apply to the polygon before the [angle](#) rotation (affects [calcPoints](#))
- [calcPoints](#) - (Calculated) The collision polygon - effectively [points](#) with [angle](#) and [offset](#) applied.
- [edges](#) - (Calculated) Array of Vectors representing the edges of the calculated polygon
- [normals](#) - (Calculated) Array of Vectors representing the edge normals of the calculated polygon (perpendiculars)

You should *not* manually change any of the properties except [pos](#) - use the [setPoints](#), [setAngle](#), and [setOffset](#) methods to ensure that the calculated properties are updated correctly.

It has the following methods:

- [setPoints\(points\)](#) - Set the original points
- [setAngle\(angle\)](#) - Set the current rotation angle (in radians)
- [setOffset\(offset\)](#) - Set the current offset
- [rotate\(angle\)](#) - Rotate the original points of this polygon counter-clockwise (around its local coordinate system) by the specified number of radians. The [angle](#) rotation will be applied on top of this rotation.

-
- `translate(x, y)` - Translate the original points of this polygon (relative to the local coordinate system) by the specified amounts. The `offset` translation will be applied on top of this translation.
 - `getAABB()` - Compute the axis-aligned bounding box. Returns a new Polygon every time it is called. Is performed based on the `calcPoints`.
 - `getAABBAsBox()` - Compute the axis-aligned bounding box. Returns a new Box every time it is called. Is performed based on the `calcPoints`.
 - `getCentroid()` - Compute the Centroid of the polygon. Is performed based on the `calcPoints`.

SAT.Box

This is a simple Box with a position, width, and height. It is created by calling:

```
1 // Create a box at (10,10) with width 20 and height 40.  
2 var b = new SAT.Box(new SAT.Vector(10,10), 20, 40);
```

It has the following properties:

- `pos` - The bottom-left coordinate of the box (i.e the smallest `x` value and the smallest `y` value).
- `w` - The width of the box.
- `h` - The height of the box.

It has the following methods:

- `toPolygon()` - Returns a new Polygon whose edges are the edges of the box.

SAT.Response

This is the object representing the result of a collision between two objects. It just has a simple `new Response()` constructor.

It has the following properties:

- `a` - The first object in the collision.
- `b` - The second object in the collision.
- `overlap` - Magnitude of the overlap on the shortest colliding axis.
- `overlapN` - The shortest colliding axis (unit-vector)
- `overlapV` - The overlap vector (i.e. `overlapN.scale(overlap, overlap)`). If this vector is subtracted from the position of `a`, `a` and `b` will no longer be colliding.
- `aInB` - Whether the first object is completely inside the second.

-
- **bInA** - Whether the second object is completely inside the first.

It has the following methods:

- **clear()** - Clear the response so that it is ready to be reused for another collision test.

Note: The **cleared** value for a **Response** has what may seem to be strange looking values:

```
1 {  
2   a: null,  
3   b: null,  
4   overlap: 1.7976931348623157e+308,  
5   overlapV: Vector(0, 0),  
6   overlapN: Vector(0, 0),  
7   aInB: true,  
8   bInA: true  
9 }
```

These just make calculating the response simpler in the collision tests. If the collision test functions return **false** the values that are in the response should not be examined, and **clear()** should be called before using it for another collision test.

Collision Tests

SAT.js contains the following collision tests:

SAT.pointInCircle(p, c)

Checks whether a given point is inside the specified circle.

SAT.pointInPolygon(p, poly)

Checks whether a given point is inside a specified convex polygon.

SAT.testCircleCircle(a, b, response)

Tests for a collision between two **Circles**, **a**, and **b**. If a response is to be calculated in the event of collision, pass in a **cleared Response** object.

Returns **true** if the circles collide, **false** otherwise.

If it returns **false** you should not use any values that are in the **response** (if one is passed in)

SAT.testPolygonCircle(polygon, circle, response)

Tests for a collision between a [Polygon](#) and a [Circle](#). If a response is to be calculated in the event of a collision, pass in a [cleared Response](#) object.

Returns **true** if there is a collision, **false** otherwise.

If it returns **false** you should not use any values that are in the [response](#) (if one is passed in)

SAT.testCirclePolygon(circle, polygon, response)

The same thing as [SAT.testPolygonCircle](#), but in the other direction.

Returns **true** if there is a collision, **false** otherwise.

If it returns **false** you should not use any values that are in the [response](#) (if one is passed in)

Note: This is slightly slower than [SAT.testPolygonCircle](#) as it just calls that and reverses the result

SAT.testPolygonPolygon(a, b, response)

Tests whether two polygons [a](#) and [b](#) collide. If a response is to be calculated in the event of collision, pass in a [cleared Response](#) object.

Returns **true** if there is a collision, **false** otherwise.

If it returns **false** you should not use any values that are in the [response](#) (if one is passed in)

Note: If you want to detect a collision between [Boxes](#), use the [toPolygon\(\)](#) method

Examples

Test two circles

```
1 var V = SAT.Vector;
2 var C = SAT.Circle;
3
4 var circle1 = new C(new V(0,0), 20);
5 var circle2 = new C(new V(30,0), 20);
6 var response = new SAT.Response();
7 var collided = SAT.testCircleCircle(circle1, circle2, response);
8
9 // collided => true
10 // response.overlap => 10
11 // response.overlapV => (10, 0)
```

Test a circle and a polygon

```
1 var V = SAT.Vector;
2 var C = SAT.Circle;
3 var P = SAT.Polygon;
4
5 var circle = new C(new V(50,50), 20);
6 // A square
7 var polygon = new P(new V(0,0), [
8     new V(0,0), new V(40,0), new V(40,40), new V(0,40)
9 ]);
10 var response = new SAT.Response();
11 var collided = SAT.testPolygonCircle(polygon, circle, response);
12
13 // collided => true
14 // response.overlap ~> 5.86
15 // response.overlapV ~> (4.14, 4.14) - i.e. on a diagonal
```

Test two polygons

```
1 var V = SAT.Vector;
2 var P = SAT.Polygon;
3
4 // A square
5 var polygon1 = new P(new V(0,0), [
6     new V(0,0), new V(40,0), new V(40,40), new V(0,40)
7 ]);
8 // A triangle
9 var polygon2 = new P(new V(30,0), [
10     new V(0,0), new V(30, 0), new V(0, 30)
11 ]);
12 var response = new SAT.Response();
13 var collided = SAT.testPolygonPolygon(polygon1, polygon2, response);
14
15 // collided => true
16 // response.overlap => 10
17 // response.overlapV => (10, 0)
```

No collision between two Boxes

```
1 var V = SAT.Vector;
2 var B = SAT.Box;
3
4 var box1 = new B(new V(0,0), 20, 20).toPolygon();
5 var box2 = new B(new V(100,100), 20, 20).toPolygon();
6 var collided = SAT.testPolygonPolygon(box1, box2);
7
8 // collided => false
```

Hit testing a circle and polygon

```
1 var V = SAT.Vector;
2 var C = SAT.Circle;
3 var P = SAT.Polygon;
4
5 var triangle = new P(new V(30,0), [
6   new V(0,0), new V(30, 0), new V(0, 30)
7 ]);
8 var circle = new C(new V(100,100), 20);
9
10 SAT.pointInPolygon(new V(0,0), triangle); // false
11 SAT.pointInPolygon(new V(35, 5), triangle); // true
12 SAT.pointInCircle(new V(0,0), circle); // false
13 SAT.pointInCircle(new V(110,110), circle); // true
```

Tests

To run the tests from your console:

```
1 npm install
2 npm run test
```