Collision detection

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

19 października 2015

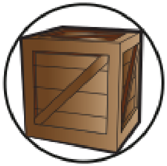
Table of contents

Collision in games

Algorithms to detect collision in games depend on the type of shapes that can collide (e.g. rectangle to rectangle, rectangle to circle, circle to circle). Generally we have a simple generic shape known as a "hitbox" that covers somehow the entity so even though collision may not be pixel perfect, it will look good enough and be performant across multiple entities.

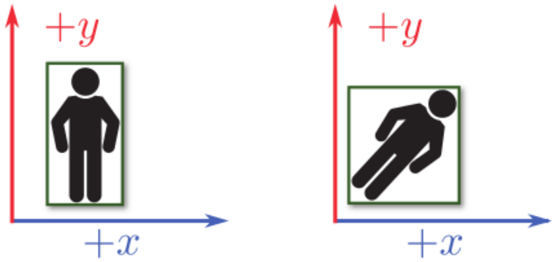
Collision in games

Different hitbox: bounding sphere



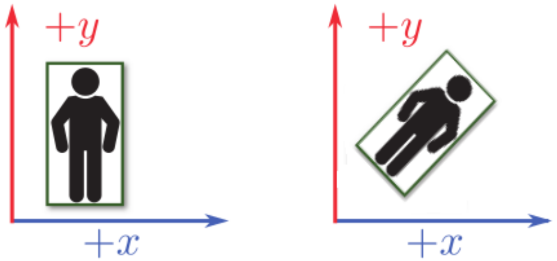
Collision in games

Different hitbox: axis aligned bounding boxes (AABB)



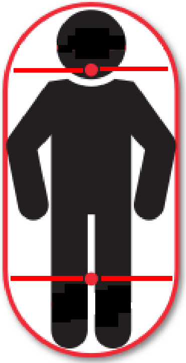
Collision in games

Different hitbox: oriented bounding boxes (OBB)



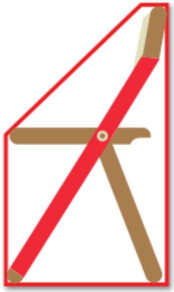
Collision in games

Different hitbox: capsule

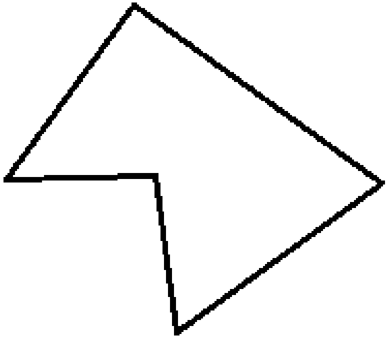


Collision in games

Different hitbox: convex polygon



Different hitbox: concave polygon

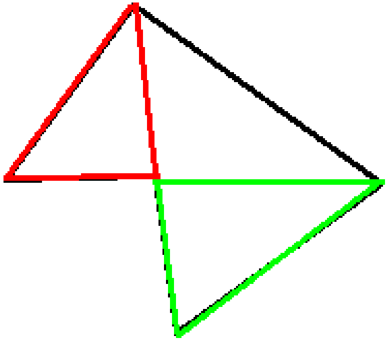


A simple polygon that is not convex is called concave or non-convex (sometimes also reentrant). It is always possible to partition a concave polygon into a set of convex polygons[1].

[1] A polynomial-time algorithm for finding a decomposition into as few convex polygons is described in: Chazelle, Bernard; Dobkin, David P., *Optimal convex decompositions*, in Toussaint, G.T., Computational Geometry, Elsevier 1985, pp. 63–133.

Collision in games

Different hitbox: concave polygon

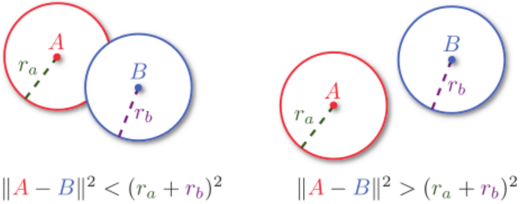


It's also worth noting that it's not uncommon for games to have multiple levels of collision geometry per object. This way, a simple collision check (such as with spheres) can be performed first to check whether there's any possibility whatsoever that two objects collided. If the simple collision check says there might be a collision, we can then perform calculations with more complex collision geometry.

Collision in games

Sphere versus sphere intersection

Two spheres intersect if the distance between their center points is less than the sum of their radii. However, computing a distance requires the use of a square root, so to avoid the square root, it is common instead to check the distance squared against the sum of the radii squared.



The algorithm for this is only a couple of lines of code. It's also extremely efficient, which is what makes using spheres a very popular basic collision detection option.

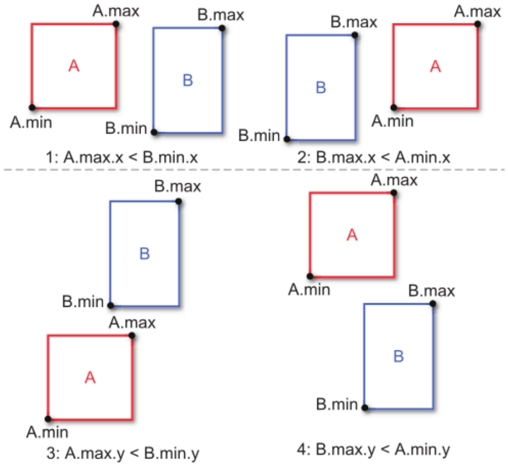
```
function SphereIntersection(BoundingSphere a, BoundingSphere b)
  // Construct a vector between centers, and get length squared
  Vector3 centerVector = b . center - a . center
  // Recall that the length squared of v is the same as v dot v
  float distSquared = DotProduct(centerVector, centerVector)
  // Is distSquared < sum of radii squared?
  if distSquared < ((a.radius+b.radius)*(a.radius+b.radius))
    return true
  else
    return false
  end
end
```

As with spheres, AABB intersection is not very expensive, even for 3D games.

Collision in games

AABB versus AABB intersection

In 2D case when checking for intersection between two AABBs, rather than trying to test the cases where the two AABBs do intersect, it's easier to test the four cases where two AABBs definitely cannot intersect.



Collision in games

AABB versus AABB intersection

```
function AABBIntersection(AABB2D a, AABB2D b)
  bool test = (a.max.x < b.min.x) || (b.max.x < a.min.x) ||
    (a.max.y < b.min.y) || (b.max.y < a.min.y)
  return !test
end
```

Separating Axis Theorem (SAT)

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. 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.

Separating Axis Theorem

Only for convex

SAT can only handle convex shapes, but this is not a problem because as we have see non-convex shapes can be represented by a combination of convex shapes (called a convex decomposition). We can then test each convex shape to determine collision for the whole shape.

Separating Axis Theorem (SAT)

An idea of projection

The 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".

Separating Axis Theorem (SAT)

Theorem (Separating Axis Theorem (SAT))

If two disjoint convex sets in n-dimensional Euclidean space sets are closed and at least one of them is compact, then there is a hyperplane in between them and even two parallel hyperplanes in between them separated by a gap.

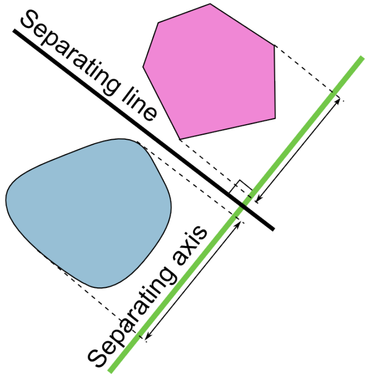
We can refolmulate this a little bit mathematical statement into more readable version

Theorem (Separating Axis Theorem (SAT))

If two convex objects are not collide, there exists an axis for which the projection of the objects will not overlap.

Separating Axis Theorem (SAT)

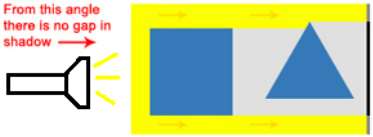
Theorem



Separating Axis Theorem (SAT)

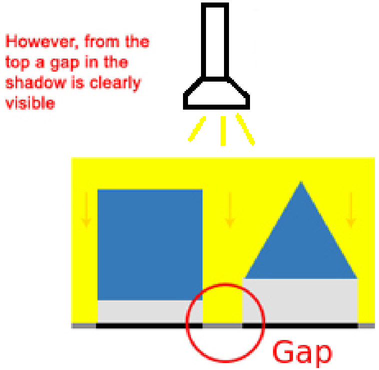
Analogy for SAT

Imagine taking a torch and shining it on the two shapes we are testing



from different angles.

If we work our way around the shapes and never find a gap in the shadows then the objects must be touching. If we find a gap, then they


However, from the top a gap in the shadow is clearly visible
Gap

are clearly not touching.

From a programming point of view it would be to intensive to check every possible angle. Luckily, due to the nature of the polygons, there is only a few key angles we need to check.
The number of angles we need to check are the same as the number of sides of the polygons. This means that the maximum number of angles to check is the sum of the number of sides the two shapes we are testing have.

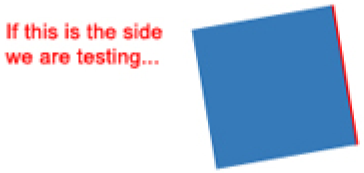
Separating Axis Theorem (SAT)

Obtaining the separating axes

From a programming point of view it would be to intensive to check every possible angle. Luckily, due to the nature of the polygons, there is only a few key angles we need to check.
The number of angles we need to check are the same as the number of sides of the polygons. This means that the maximum number of angles to check is the sum of the number of sides the two shapes we are testing have.

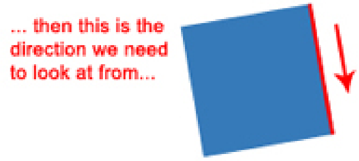
Separating Axis Theorem (SAT)

Obtaining the separating axes



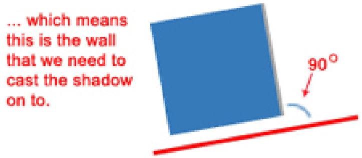
Separating Axis Theorem (SAT)

Obtaining the separating axes

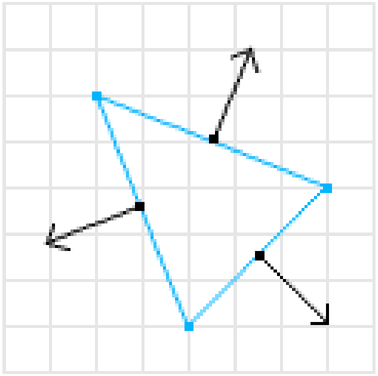


Separating Axis Theorem (SAT)

Obtaining the separating axes


... which means this is the wall that we need to cast the shadow on to.
90°

From the programmer point of view, the axes we must test are the

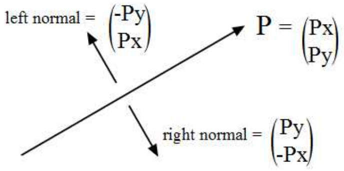


normals of each shape's edges.

Separating Axis Theorem (SAT)

Normals

The normals of the edges can be easily obtained by flipping the


left normal = $\begin{pmatrix}-Py\\Px\end{pmatrix}$
$P = \begin{pmatrix}Px\\Py\end{pmatrix}$
right normal = $\begin{pmatrix}Py\\-Px\end{pmatrix}$

coordinates and negating one.

Separating Axis Theorem (SAT)

Determine that the shapes are not intersecting

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

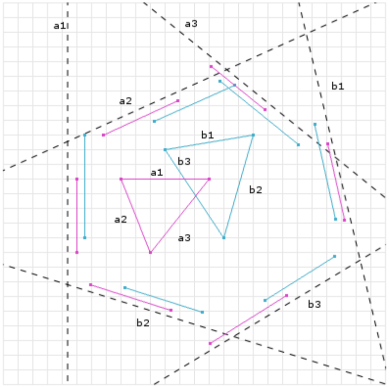
Separating Axis Theorem (SAT)

Determine that the shapes are intersecting

If, for all axes, the shape's projections overlap, then we can conclude that the shapes are intersecting. Note that **to determine intersection, all axes must be tested for overlap**.

Separating Axis Theorem (SAT)

Determine that the shapes are intersecting



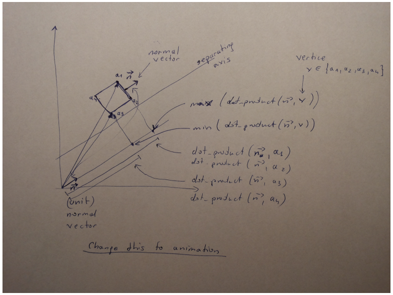
Separating Axis Theorem (SAT)

Projecting a shape onto an axis

When we know how to obtain the separating axes another question is how to project a shape onto an axis. Fortunately to project a polygon onto an axis is relatively simple. It is enought to loop over all the vertices performing the dot product with the axis and storing the minimum and maximum.

Separating Axis Theorem (SAT)

Projecting a shape onto an axis


normal vector
separating axis
vertice
$v \in \{a_1, a_2, a_3, a_4\}$
a_1 $\vec{n}$
a_4
a_2
a_3
~~max~~ (dot_product($\vec{n}$, v))
min (dot_product($\vec{n}$, v))
dot_product ($\vec{n}$, a_1)
dot_product ($\vec{n}$, a_2)
dot_product ($\vec{n}$, a_3)
dot_product ($\vec{n}$, a_4)
$\vec{n}$
(unit) normal vector
Change this to animation

Separating Axis Theorem (SAT)

Projecting a shape onto an axis

As we have seen, SAT is a simple but repetitive method. Here we have one iteration of it, step by step:

1. Take one side from one of the polygons we are testing and find the normal (perpendicular) vector from it. This will be the 'axis'.
2. Loop through every point on the first polygon and project it onto the axis. Keep track of the highest and lowest values found for this polygon.
3. Do the same for the second polygon.
4. Check the values you found and see if they overlap.

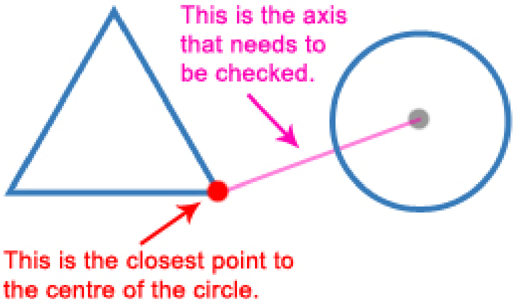
If there is no gap, then polygons might be touching and we have to keep checking until we have gone through every side of both polygons. If we get through them all without finding a gap then they collide.

Separating Axis Theorem (SAT)

Circle case

Testing a circle against a polygon in SAT is a little bit strange but it can be done.
The main thing to note is that **a circle does not have any sides so there is no obvious axis that we can test against**. Other words, **there are infinitely many axis we should take into account**. There is one 'not so obvious' axis we do need to test however. This is the axis that runs from the centre of the circle to the closest vertex on the polygon.

Separating Axis Theorem (SAT)
Circle case



After that it is just a matter of going through the usual routine of looping through every axis on the other polygon and checking for overlaps.
To project a circle onto the axis, we simply project the centre point of the circle and then add and subtract the radius.

Separating Axis Theorem (SAT)

Finding the Minimum Translation Vector (MTV)

In addition to returning true or false depending on whether the two shapes are intersecting SAT can return a Minimum Translation Vector (MTV). The MTV is the minimum magnitude vector used to push the shapes out of the collision. That axis which has the smallest overlap 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 to be able to return a MTV when the shapes intersect.

Collision in games

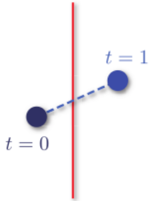
Bullet-through-paper problem

To this point, we have covered instantaneous collision detection algorithms. This means the algorithm checks to see if the two objects collide on the current frame. Although this can work in many cases, there are some instances where it won't.

Collision in games

Bullet-through-paper problem

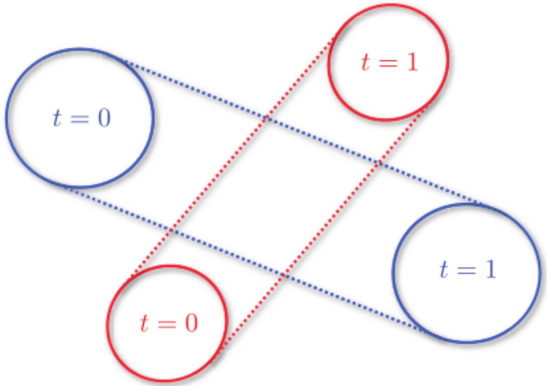
If a bullet is fired at a piece of paper, it's unlikely there will be a precise frame where the bullet and the paper intersect with each other. That's because the bullet is travelling so fast and the paper is so thin. This problem is fittingly known as the bullet-through-paper problem.



In order to solve this problem, we can use either form of continuous collision detection (CCD) or simpler swept sphere intersection.
In swept sphere intersection, there are two moving spheres. The inputs are the positions of both spheres during the last frame ($t = 0$) and the current frame ($t = 1$). Given these values, we can determine whether or not the two spheres collided at any point between the two frames. So unlike the instantaneous sphere-sphere intersection, it won't miss out on any intersections that occur between frames.

Collision in games

Swept sphere intersection



We might notice that a swept sphere looks a lot like a capsule. That's because a swept sphere is a capsule. A swept sphere has a start point, end point, and a radius, which is exactly like a capsule. So the algorithm discussed for this case can also be used for "capsule versus capsule" intersection.
Algorithm:

- use SAT,
- use dedicated algorithm like one described in: Sanjay Madhav, *Game Programming Algorithms and Techniques*, Addison-Wesley, 2014, pp 141-145.

Collision performance

Broad and narrow phase

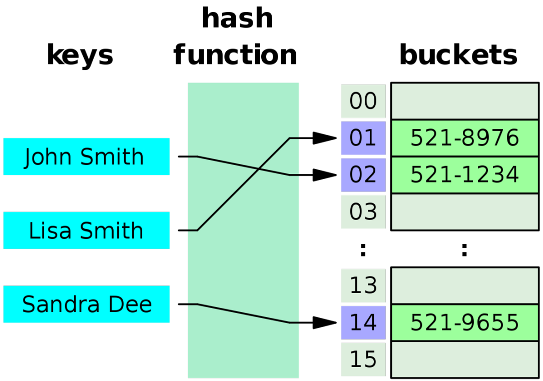
Although some of algorithms for collision detection are simple enough to calculate, it can be a waste of time to test every entity with every other entity. Usually games will split collision into two phases, broad and narrow.

Broad Phase Broad phase should give us a list of entities that possibly could be colliding. This can be implemented with a spacial data structure that will give us a rough idea of where the entity exists and what exist around it. Some examples of spacial data structures are Quad Trees, R-Trees but we can also try to use less sophisticated but not less effective like **spacial hashmap**.

Narrow Phase When you have a small list of entities to check you will want to use a narrow phase algorithm (like the ones listed above) to provide a certain answer as to whether there is a collision or not.

Hash table

A hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.
The idea of hashing is to distribute the entries (key/value pairs) across an array of buckets. Given a key, the algorithm computes an index that suggests where the entry can be found.



Spatial Hashing

A spatial hash is a 2 or 3 dimensional extension of the hash table. Typically the keys to a hash table would be strings, but in a spatial hash we use 2 or 3 dimensional points as the keys. And here is where the twist comes in: for a normal hash table, a good hash function distributes keys as evenly as possible across the available buckets, in an effort to keep lookup time short. The result of this is that keys which are very close (lexicographically speaking) to each other, are likely to end up in distant buckets. But in a spatial hash we are dealing with locations in space, and locality is very important to us (especially for collision detection), so our hash function will not change the distribution of the inputs.

Spatial Hashing

There are a few problems with inserting objects other than simple points into the spatial hash:

- the object may overlap several cells/buckets if it is near the edge of a bucket,
- an object may actually be larger than a single bucket.

To solve this problem we add the object to all relevant buckets.