Collision Detection in Computer Games

Fangkai Yang[1]

[1]Computational Science and Technology
KTH Royal Institute of Technology

Outline

Outline

Who is Fangkai?

Definition

Fangkai Yang is a crazy Gamer, Mathematician and Programmer.

- PhD candidate in Computer Science. Research on Crowd-simulation and Game AI.
- Game developer: Just Cause 3 (Avalanche Studios), War Rage (NetEase Games).

Collision Detection: A Gentle Introduction

Collision detection concerns the detection of collisions between objects in the virtual environment. Primarily employed to stop objects moving through each other and the environment.

Collision Detection is everywhere in computer games: between characters and characters, between characters and terrain, etc.

Remember:

- Physical laws do not exist in the virtual world by default.
- Must computationally model and program them.

Dayz Standalone

What my friends think I was playing:

Dayz Standalone

What I was actually playing:

Dayz Standalone

What I was actually playing:

FIFA

What my friends think I was playing:

FIFA

What I was actually playing:

Basic Geometry Types

Rigid-bodies

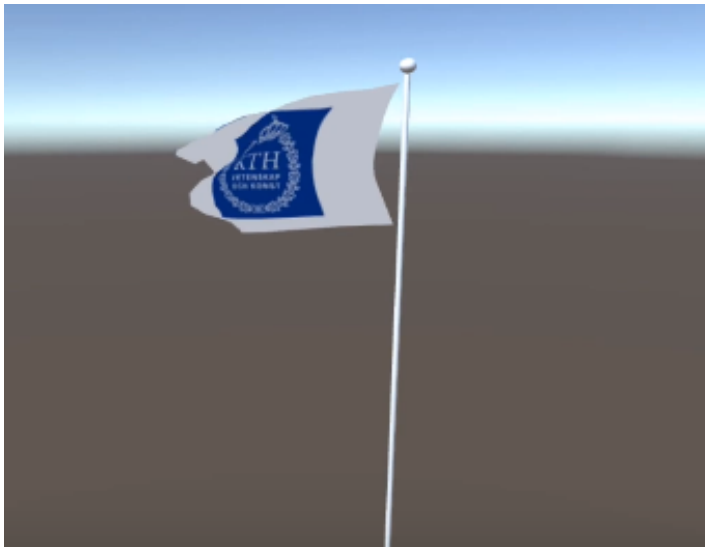
- Hard objects (ideal solid bodies)
- Constant distance between vertices in the mesh
- Convex vs. concave

Soft-bodies

- Cloth, for example
- More complicated to simulate
- Soft-bodies like cloth may collide with themselves

Basic Geometry Types

Basic Geometry Types



https://www.youtube.com/watch?v=M5Ygpxfmcg8

Outline

King of Fighter 97

I want a volunteer!

Question:

How to test if a character has been hit?

Outline

Collision Detection

Definition

Collision detection refers the detection of the intersection of two or more objects.

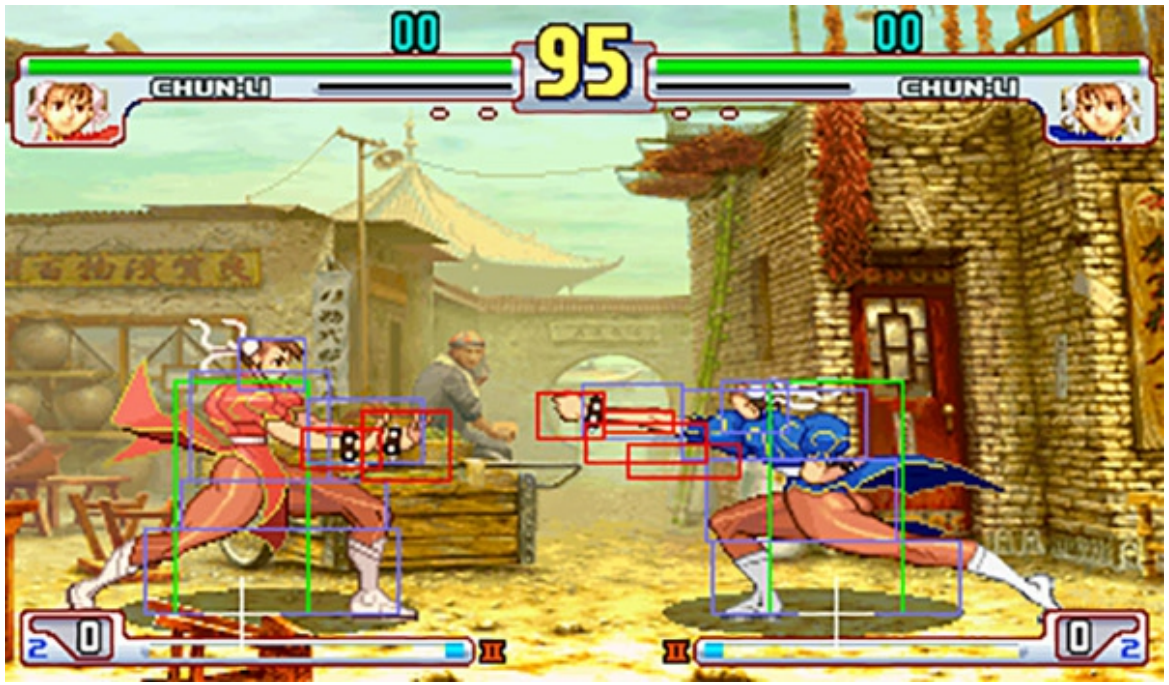


Figure: Collision detection in Street Fighter

Figure: Overlaps of characters.

How about collision detection in 3D games?

Penetration

When collision is detected, the penetration follows.

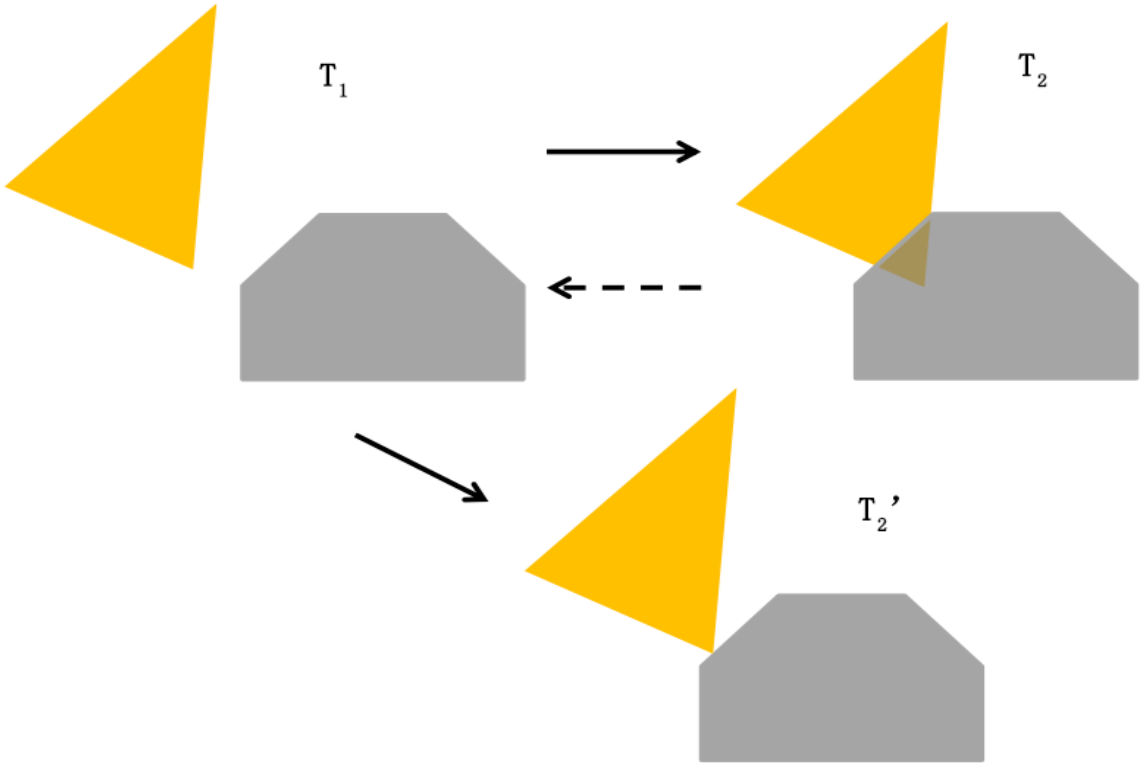
Two forms of collision detection:

- Continuous: very expensive. Simulate solid objects in real life.
- Discrete: objects will end up with penetrating each other.

Backtracking

Backtracking

Like Tom Cruise in *Edge of Tomorrow*, it turns time backwards and fix the penetration that occurred in the last frame.

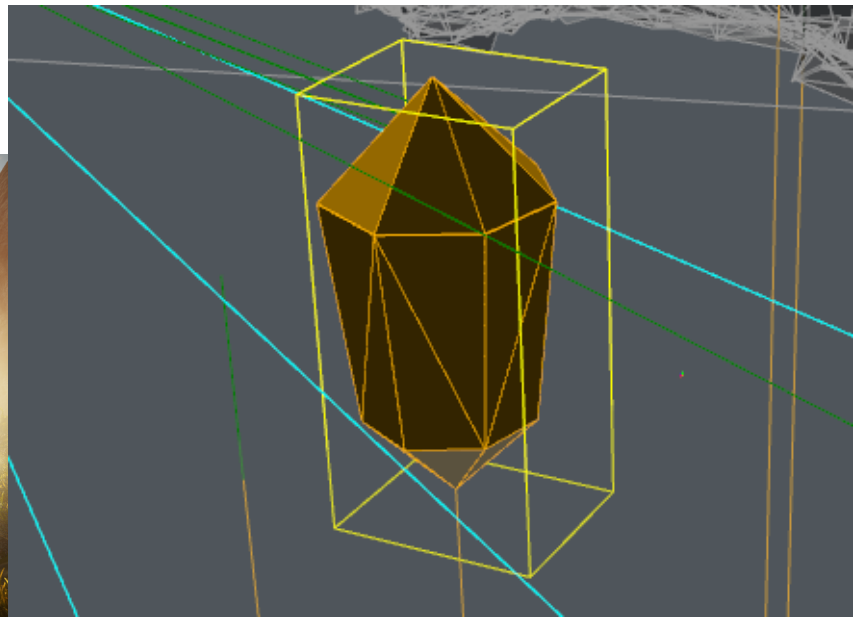


Bounding Volume

Definition

Bounding volume for a set of objects is a closed volume that completely contains the union of the objects in the set.

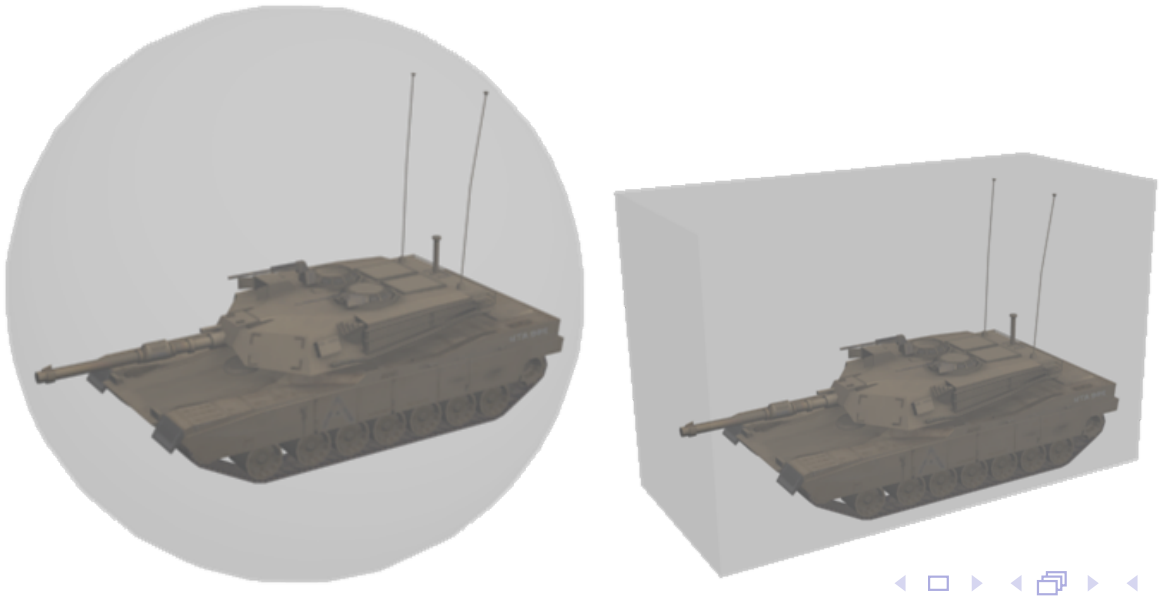
(a) In the game scenario.



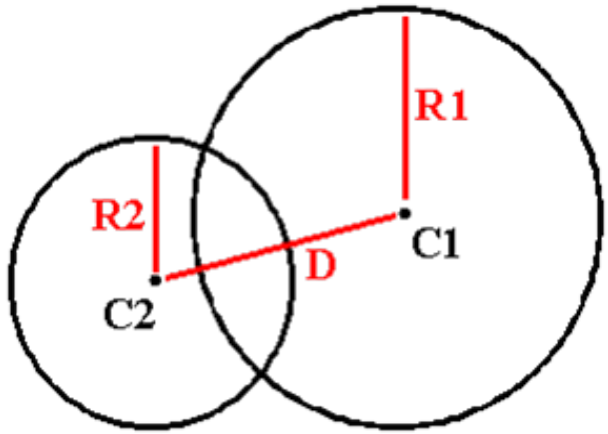
(b) In the physics engine.

Bounding Volume types

- Bounding Box: Axis Aligned Bounding Box (AABB), Oriented Bounding Box (OBB), etc. For objects that rest upon other, such as a car resting on the ground, using a bounding box is a better choice.
- Bounding Sphere: They are very quick to test for collision with each other.

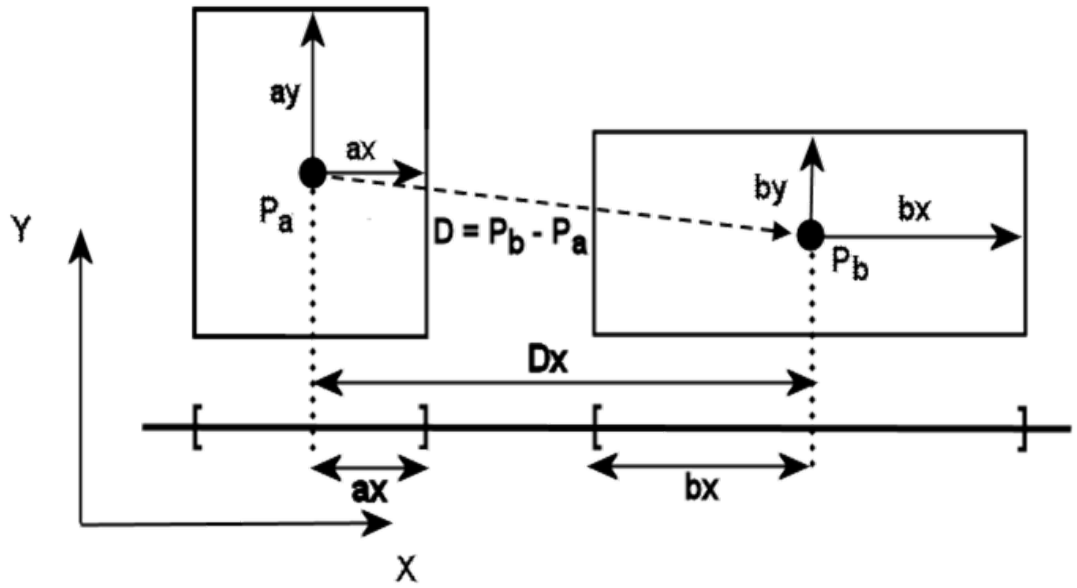


Sphere Collision Detection



Circles are colliding if D <= (R1 + R2)
d is the distance between the circle centers
R1 and R2 are the radii of both circles respectively

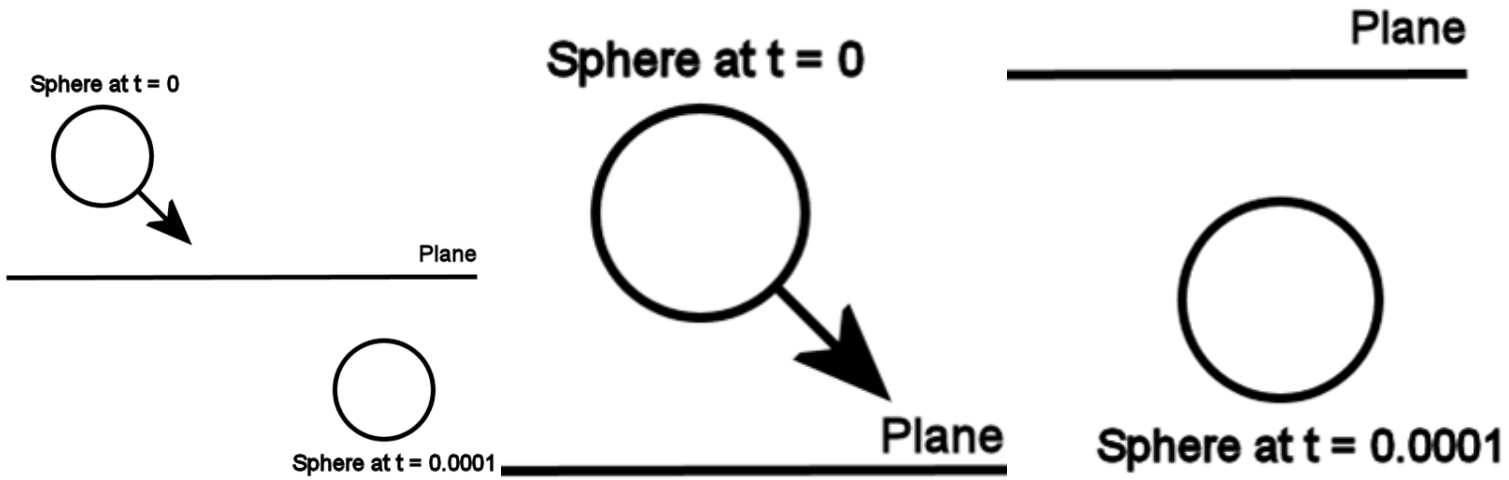
AABB Collision Detection



- Recall: Faces of an AABB are aligned with the coordinates axes.
- To see if A and B overlap, separating axes test can be used along the x,y and z axes.
  This is faster, as only need to search x,y and z axes.

Some Problems

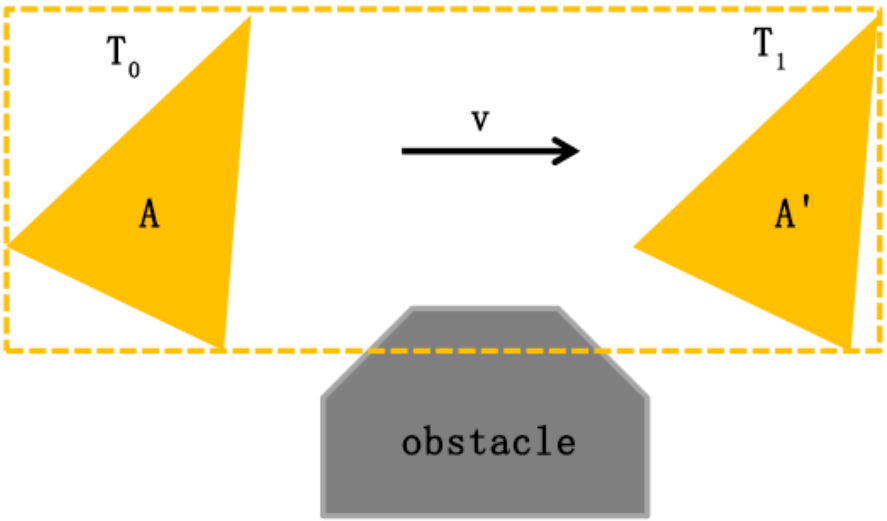
- Naive collision detection approaches may simply test if two objects are overlapping at every time-step.
- Problem: quickly moving objects can pass right through each other without detection



(b) First test. (c) Second test.

Swept Volume

The AABB in the game bounds not the object, but the *Swept Volume* of the object from one frame to the next.



Broad Phase and Narrow Phase

Most efficient implementations use a two-phase approach: a broad phase followed by a narrow phase.

- Broad phase: collision tests are based on bounding volumes only. Quickly prune away pairs of objects that do not collide with each other. The output is the potentially colliding set of pairs of objects.
- Narrow phase: collision tests are exact—they usually compute exact contact points and normals—thus much more costly, but performed for only the pairs of the potentially colliding set.

Broad Phase

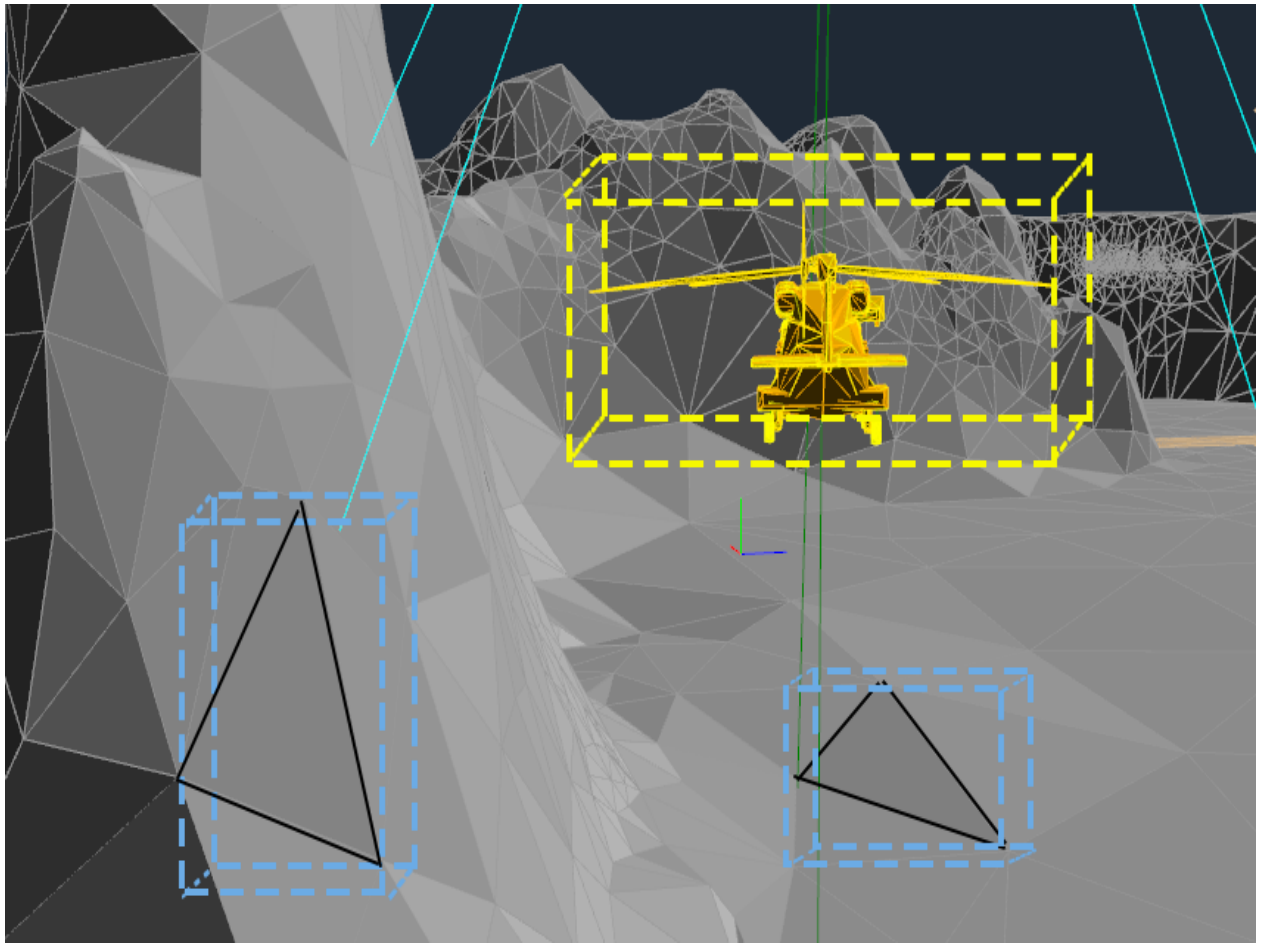
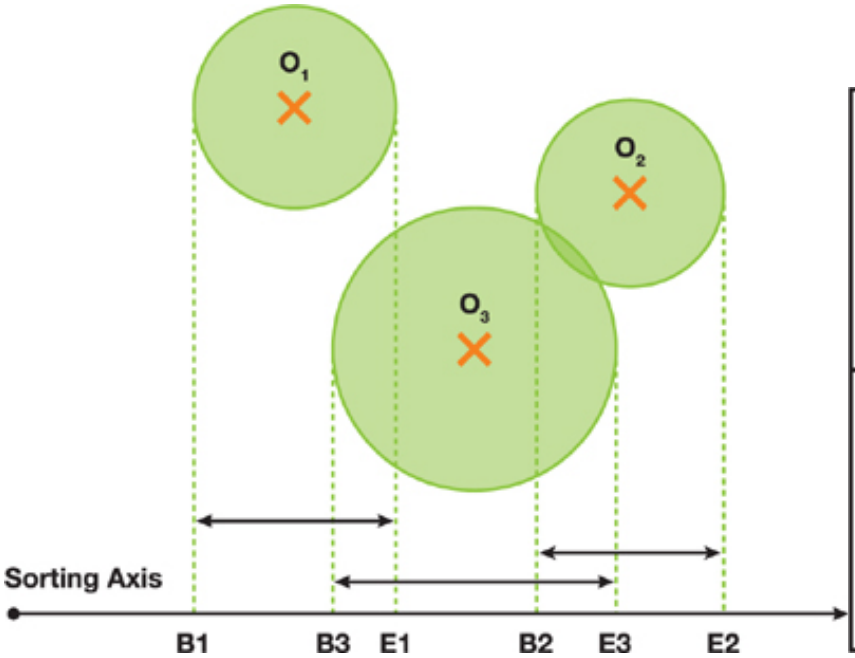


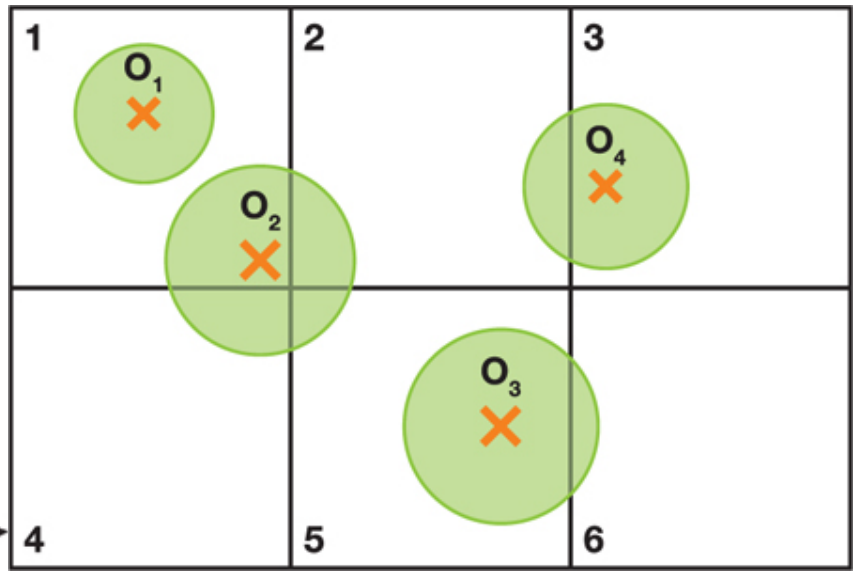
Figure: Collision detection between AABBs (Axis Aligned Bounding Box).

Broad Phase

The broad phase collision detection helps to filtrate the potentially colliding object sets.
The brute-force way is doing collision test for all pairs of objects, and the complexity is $\mathcal{O}(n^2)$.



(a) Sweep and prune.



(b) Spatial subdivision.

Narrow Phase

Separating Axis Tests

- Many algorithms are based on the separating hyperplane theorem

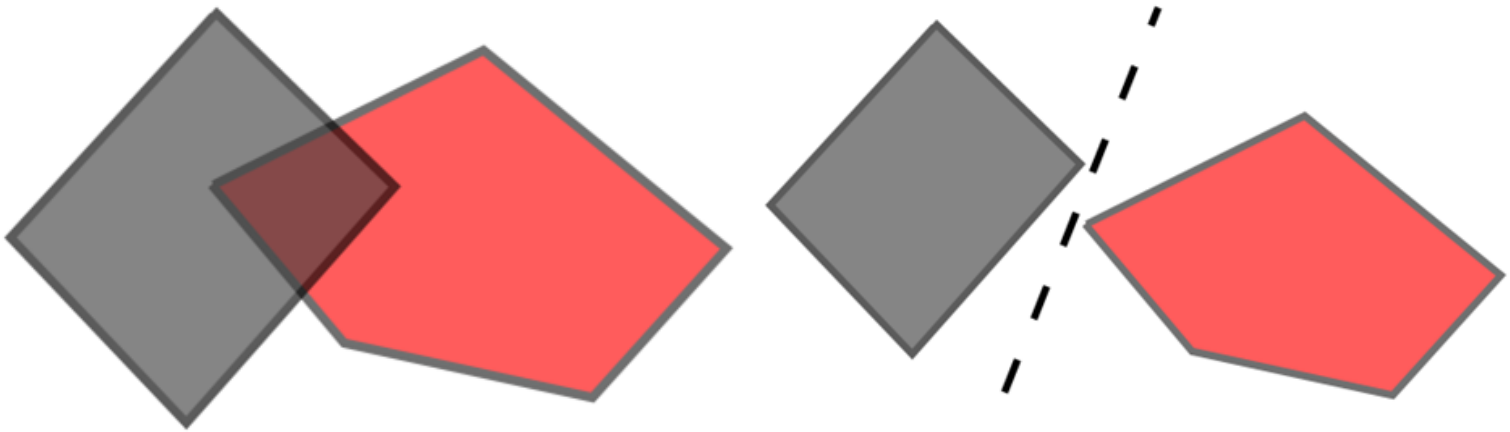
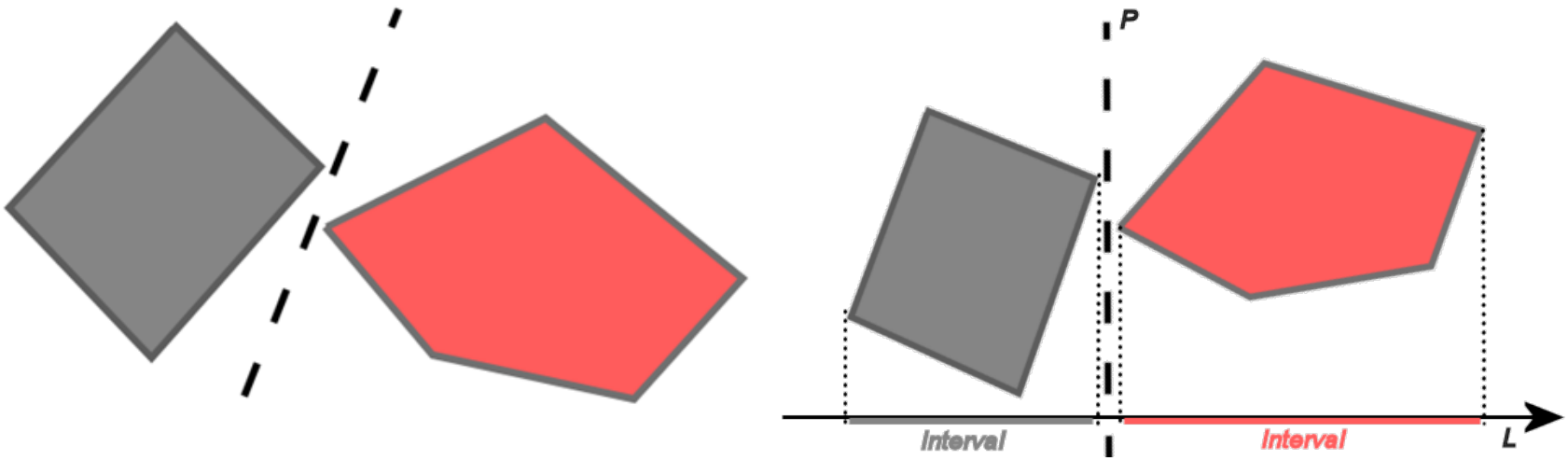


Figure: Two disjoint convex sets are separable by a hyperplane.

Separation w.r.t. a plane P => separation of the orthogonal projections onto any line L parallel to plane normal

Separating Axis Tests

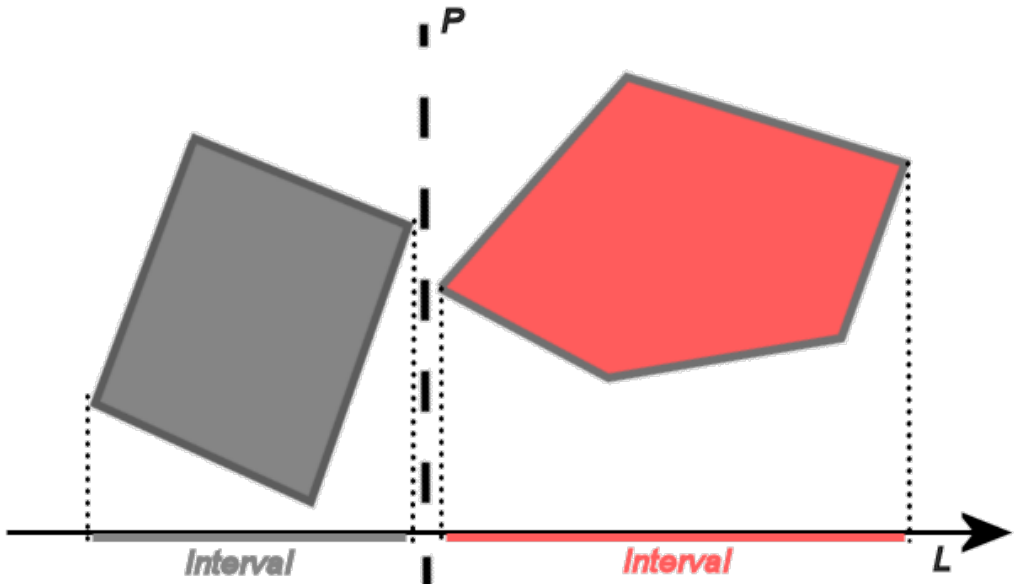
Think of it like this: do a twist to simplify calculations.



Separating axis is a line for which the projection intervals do not overlap.

Separating Axis Tests

Separated if $A_{max} < B_{min}$ or $B_{max} < A_{min}$

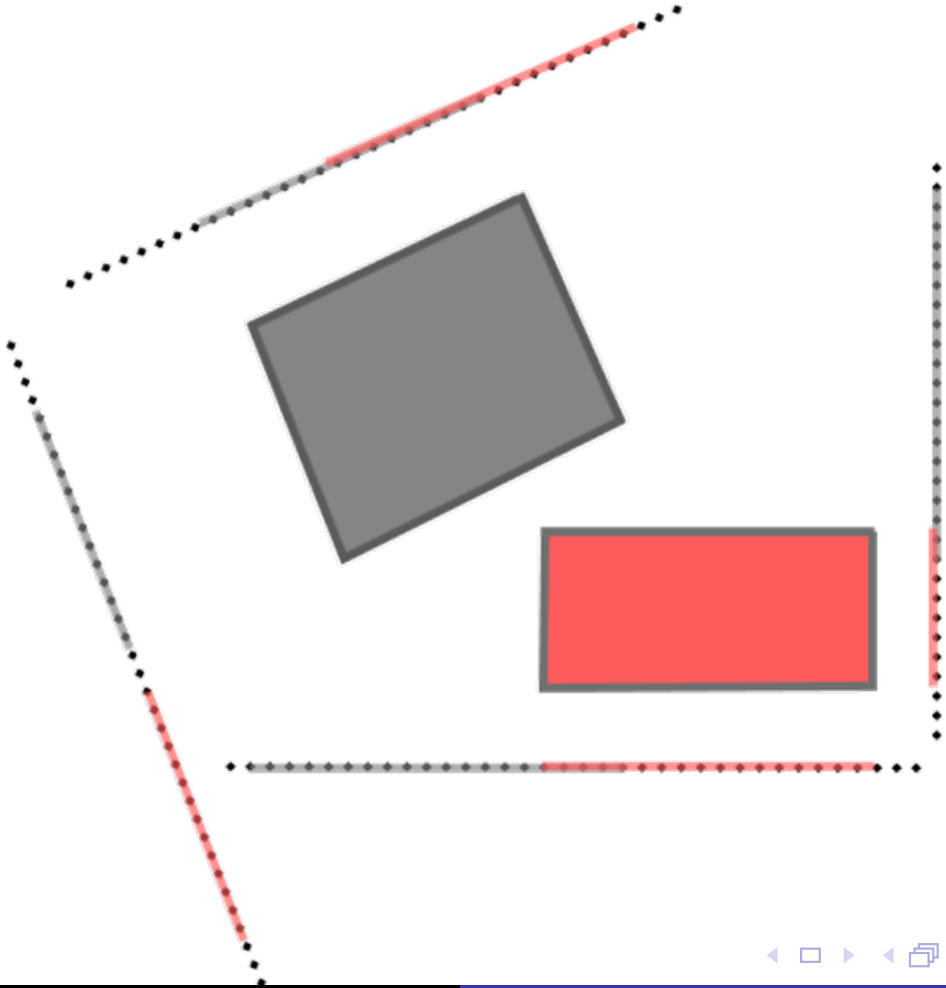


Question:

Which axes to be tested?

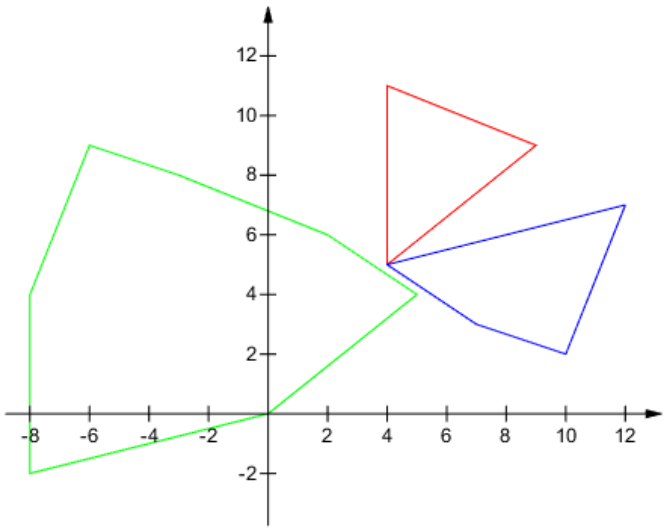
Separating Axis Tests

Axes to test:

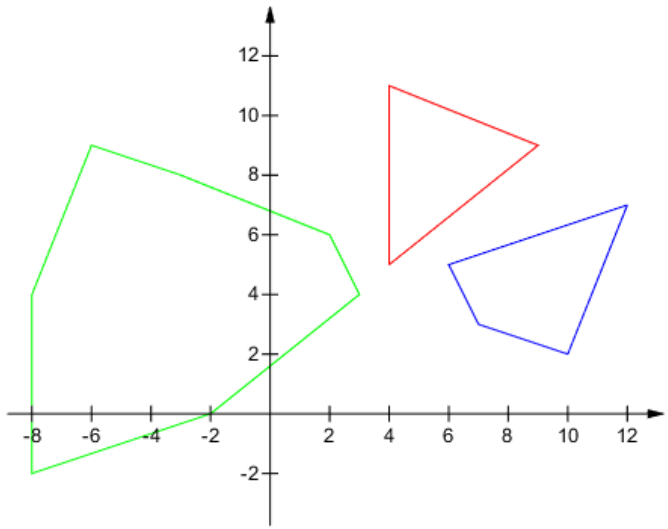


Convex versus Convex Collision Detection

One of the most commonly used algorithm in game engine is *GJK* (Gilbert-Johnson-Keerthi) algorithm. This algorithm relies on a support function to iteratively get closer simplices to the solution using Minkowski difference.



(a) Two shapes collide on one vertex.

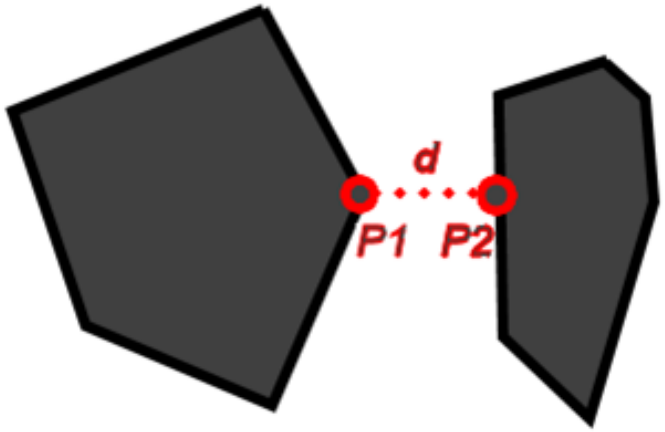


(b) No collision.

GJK

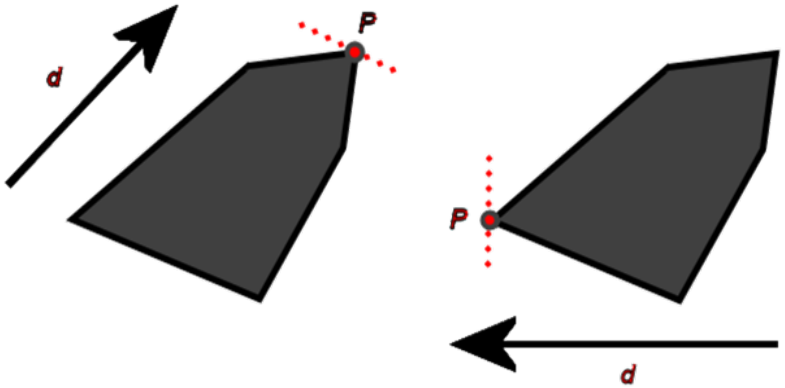
Given the two polyhedra:

- Computes the distance d between them
- Can also return the closest pair of points on each polygons



Support Mapping Function

- Supporting point. This is an extreme point on the polygon for a given direction d.
- The support mapping function returns this point



The first simplex is built using what is called a *support function*. Support function returns one point inside the Minkowski difference given two sets K_A and K_B.

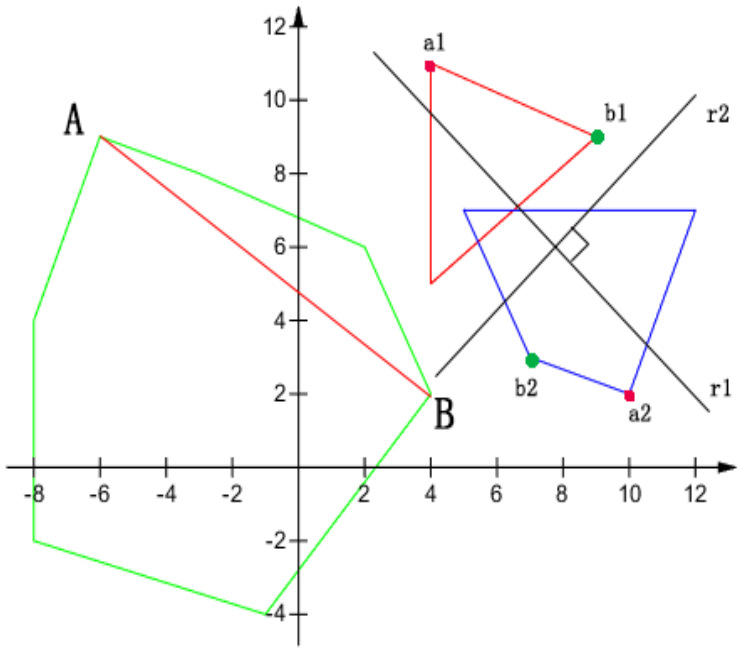
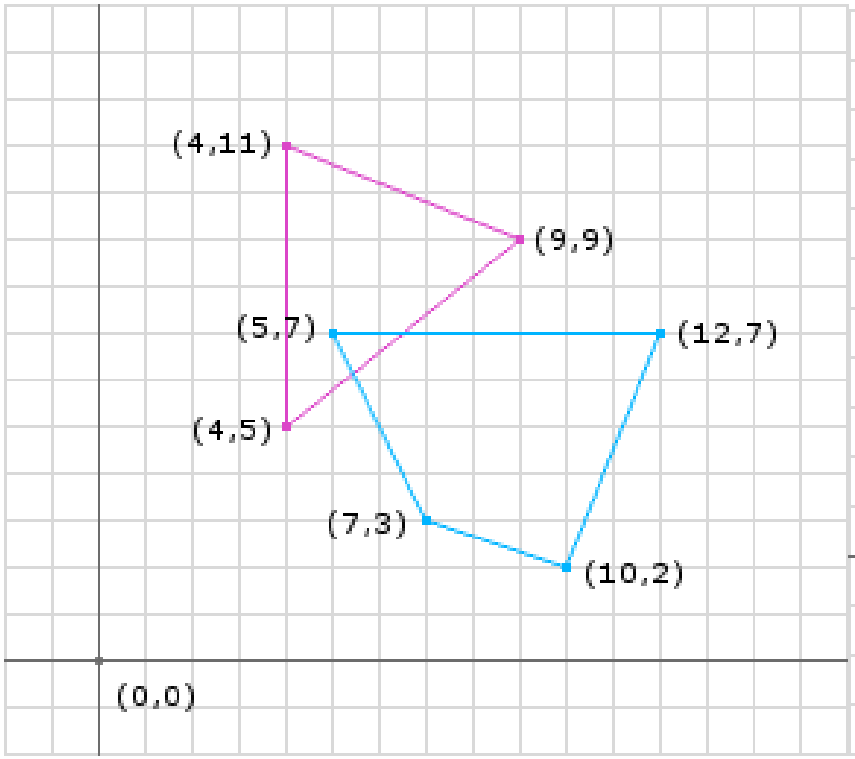
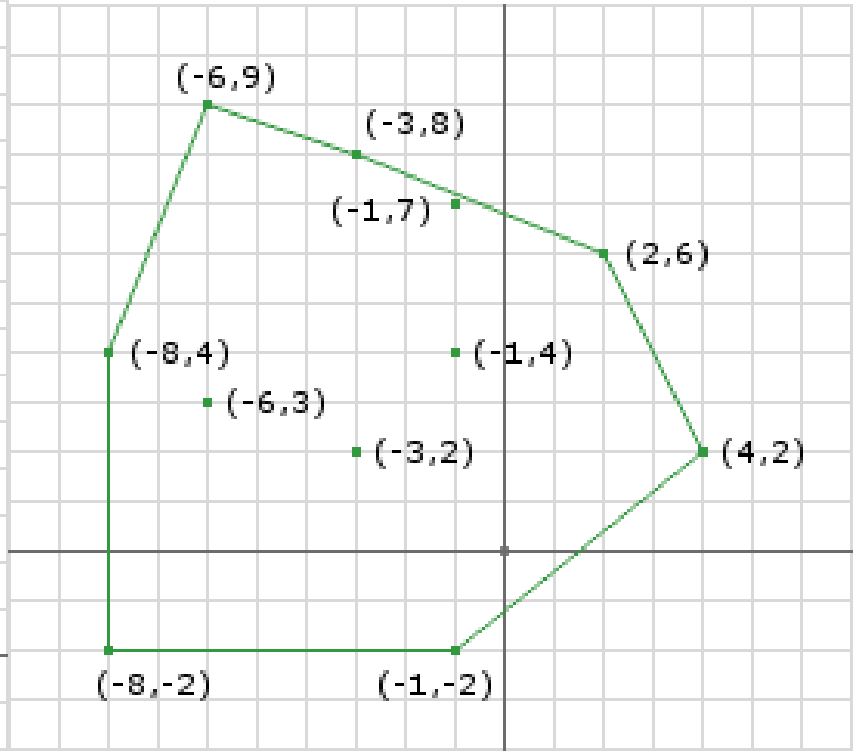


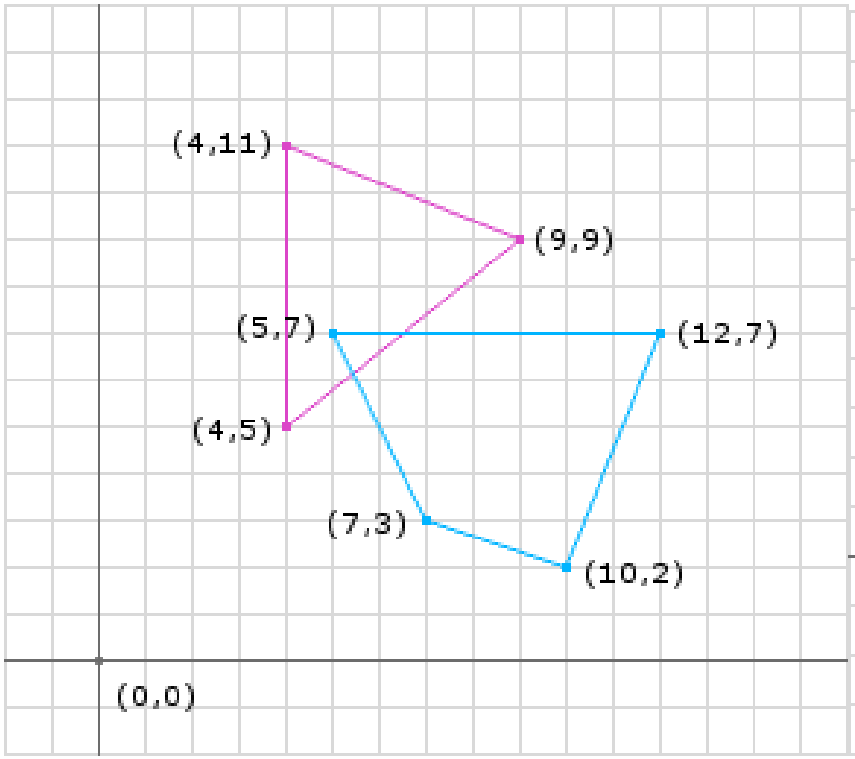
Figure: The first two vertices in the first simplex.

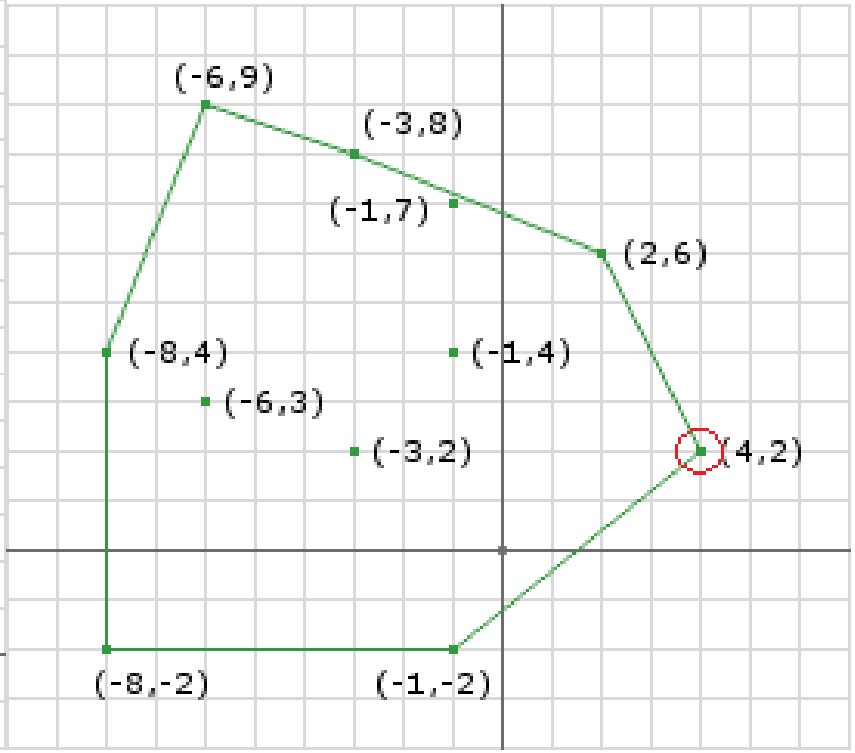
An Example





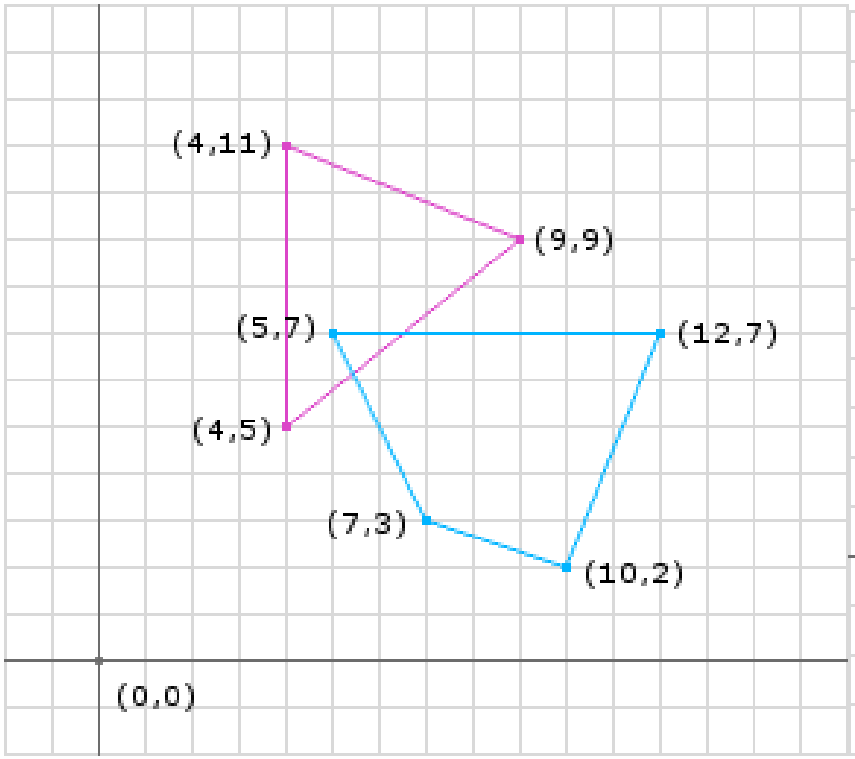
An Example

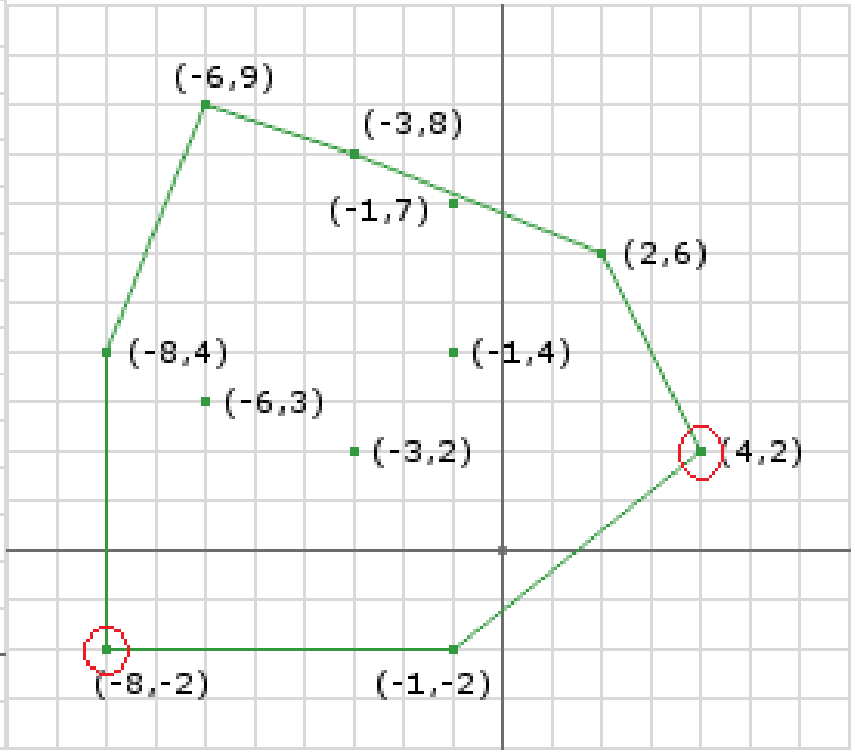




Select direction $d = (1, 0)$:
$p1 = (9, 9); p2 = (5, 7); p3 = p1 - p2 = (4, 2);$

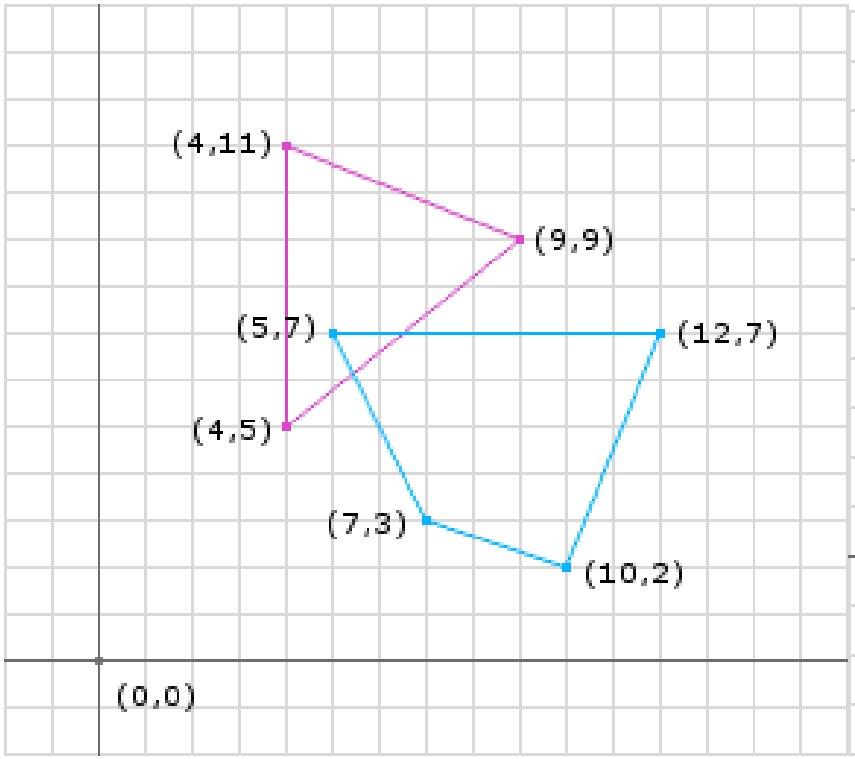
An Example

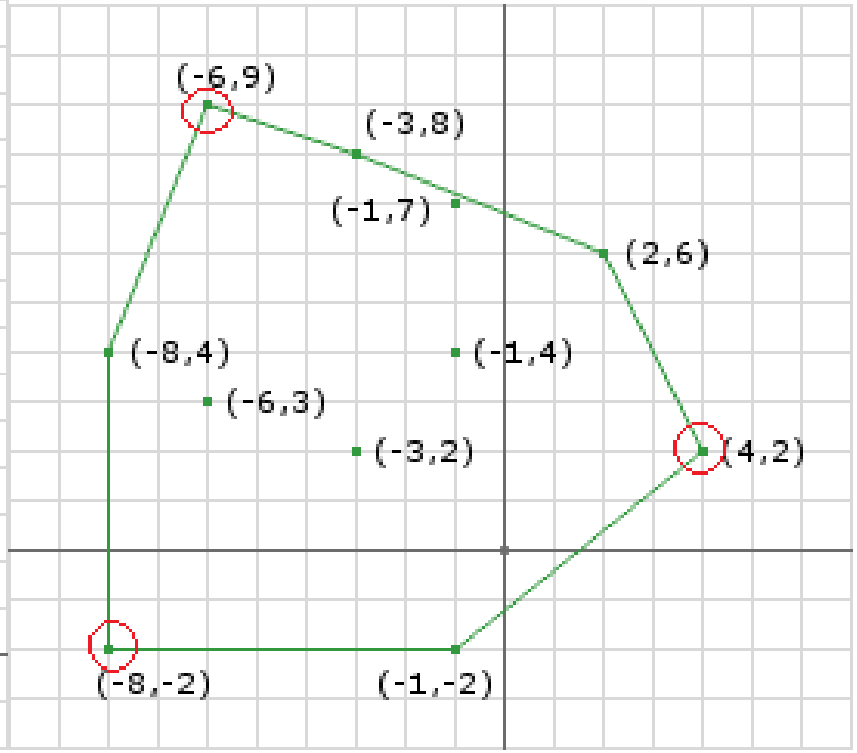




Select direction $d = (-1, 0)$:
$p1 = (4, 5); p2 = (12, 7); p3 = p1 - p2 = (-8, -2);$

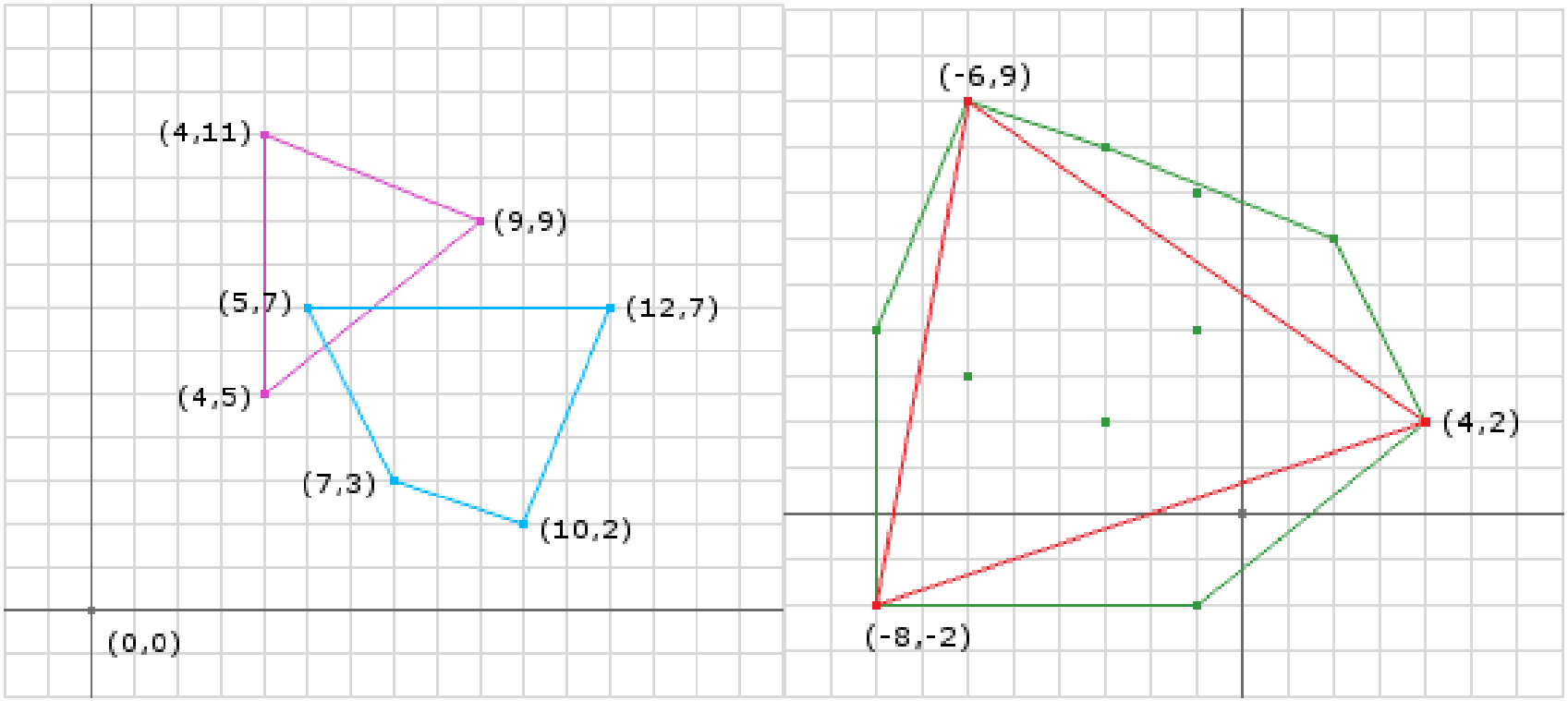
An Example





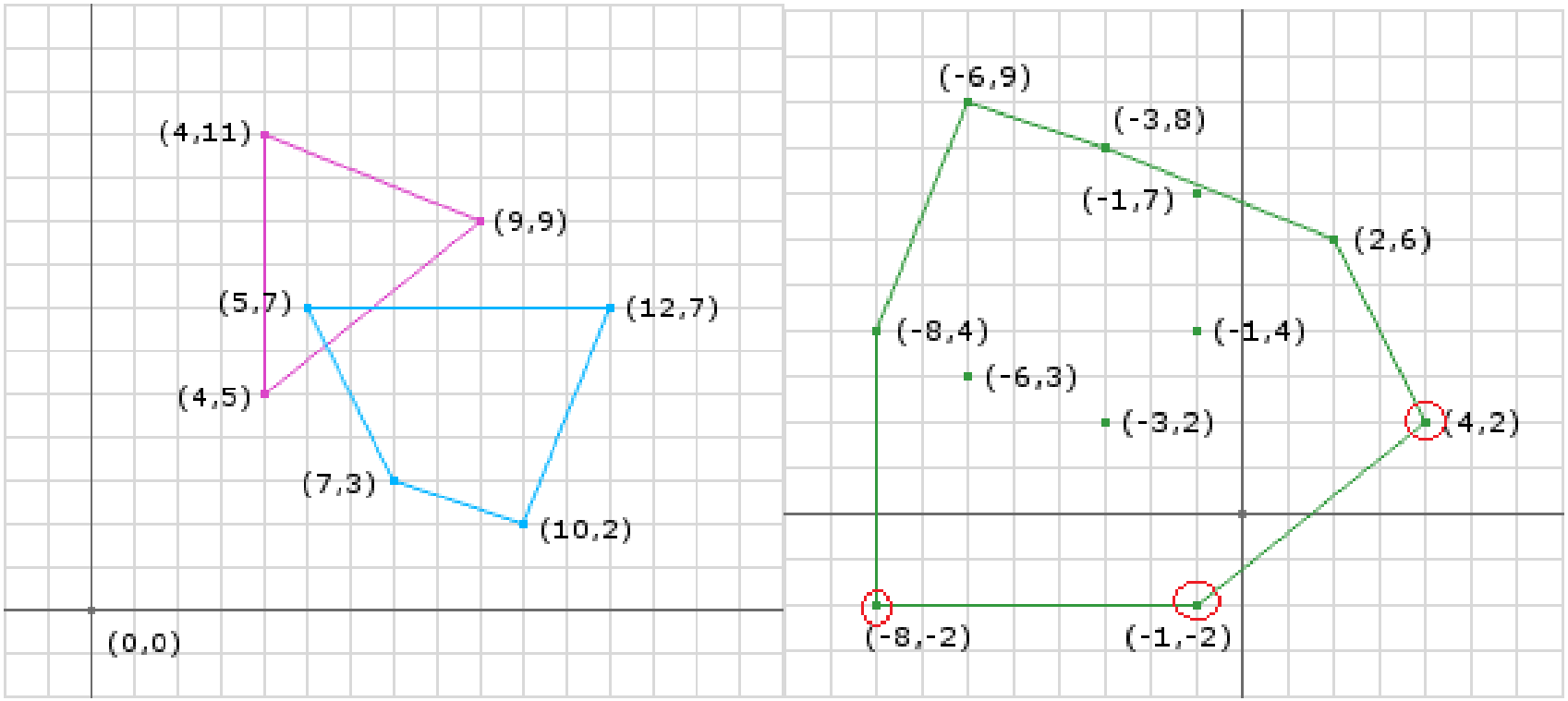
Select direction $d = (0, 1)$:
$p1 = (4, 11); p2 = (10, 2); p3 = p1 - p2 = (-6, 9);$

An Example



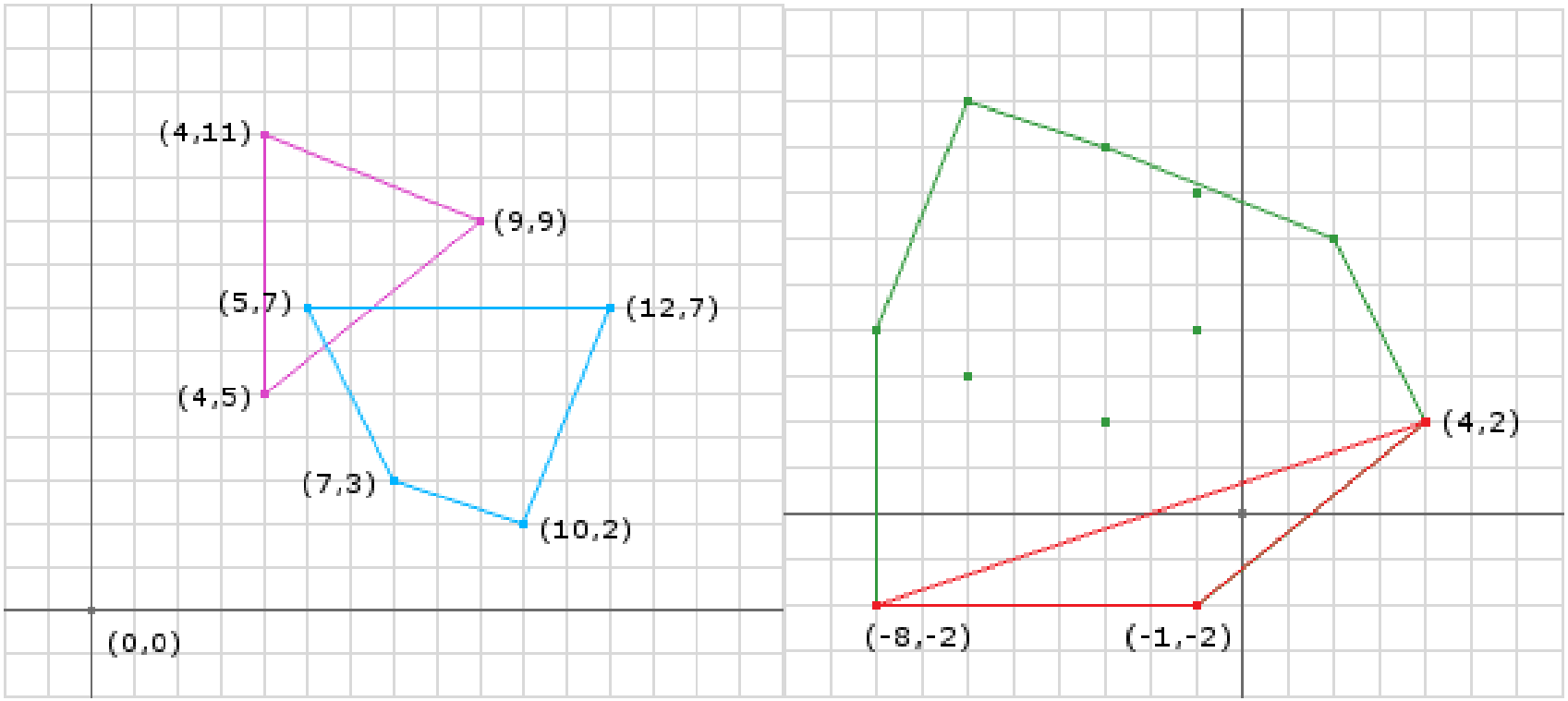
The Simplex doesn't contain the origin, but we know that the two shapes are intersecting. The problem here is that our first guess (at choosing directions) didn't yield a Simplex that enclosed the origin.

An Example



Select direction $d = (0, -1)$:
$p1 = (4, 5); p2 = (5, 7); p3 = p1 - p2 = (-1, -2);$

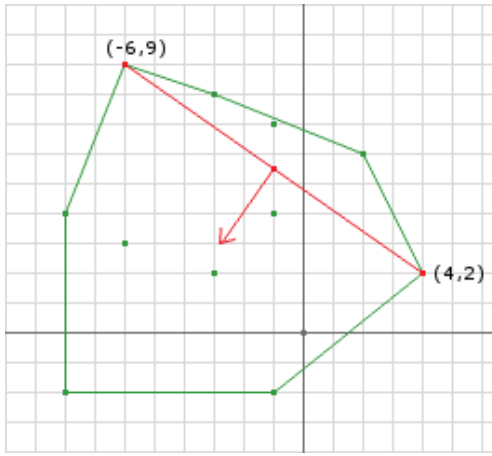
An Example



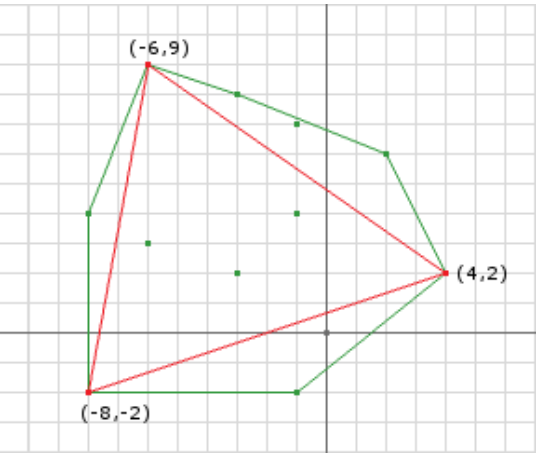
Now we contain the origin and can determine that there is a collision

An Example

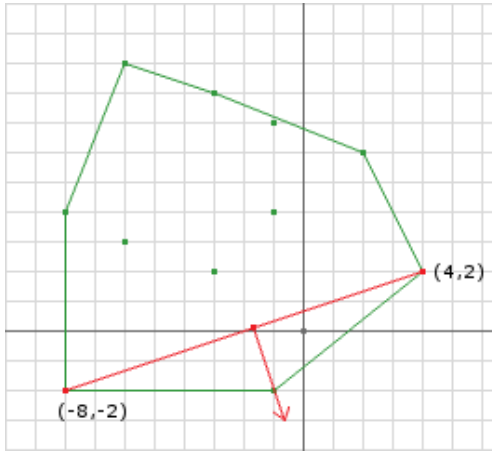
```
1  Vector d = // choose a search direction
2  // get the first Minkowski Difference point
3  Simplex.add(support(A, B, d));
4  // negate d for the next point
5  d.negate();
6  // start looping
7  while (true) {
8    // add a new point to the simplex because we haven't terminated yet
9    Simplex.add(support(A, B, d));
10   // make sure that the last point we added actually passed the origin
11   if (Simplex.getLast().dot(d) <= 0) {
12     // if the point added last was not past the origin in the direction of d
13     // then the Minkowski Sum cannot possibly contain the origin since
14     // the last point added is on the edge of the Minkowski Difference
15     return false;
16   } else {
17     // otherwise we need to determine if the origin is in
18     // the current simplex
19     if (Simplex.contains(ORIGIN)) {
20       // if it does then we know there is a collision
21       return true;
22     } else {
23       // otherwise we cannot be certain so find the edge who is
24       // closest to the origin and use its normal (in the direction
25       // of the origin) as the new d and continue the loop
26       d = getDirection(Simplex);
27     }
28   }
29 }
```



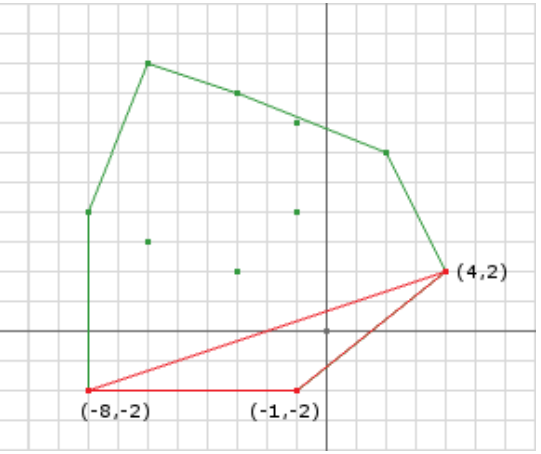
(a) The first iteration.



(b) New simplex does not contain the origin.



(c) The second iteration.



(d) New simplex contains the origin.

Thank you for Listening and Sleeping!