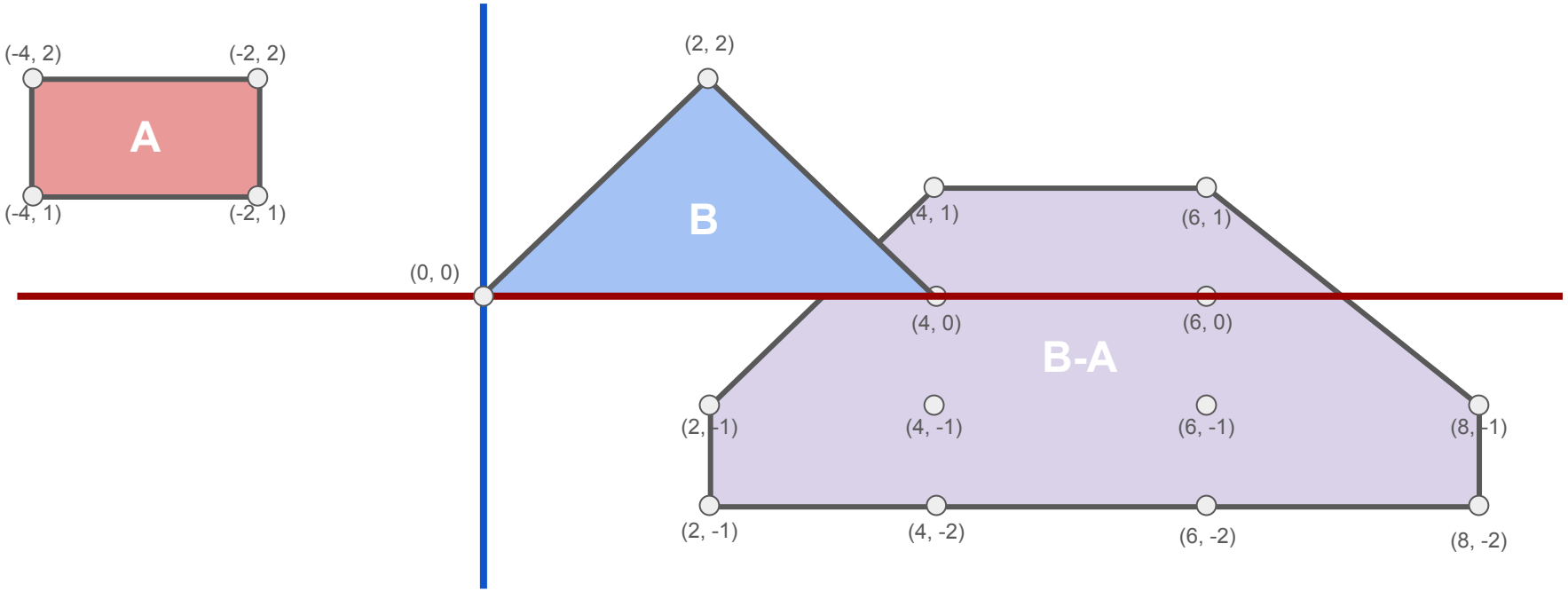
GJK Implementation

Using GJK for 3D collision prediction

Minkowski difference

- The difference between two shapes
- Imagine shapes ***A*** and ***B*** as sets of points:
  - Point ***a*** ∈ (***A***)
  - Point ***b*** ∈ (***B***)
- Represented as a new shape, or set ***A-B***
  - Point ***ab*** ∈ (***A-B***), if ***a*** ∈ (***A***) and ***b*** ∈ (***B***)
- If the origin is contained in the set ***A-B***, the two shapes are overlapping
  - ***a - b =*** 0, ***a = b***; Both sets contain the same point

Minkowski Difference



Infinite points and Convex Hulls

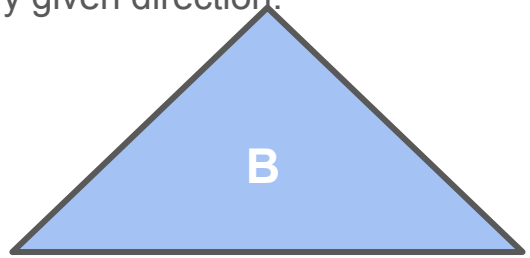
The number of points in a line between any two points is infinite, a computer cannot evaluate that many points.
A 2D polygon has an even larger infinity of points, and 3D larger still…
Way too many points to evaluate every single one!

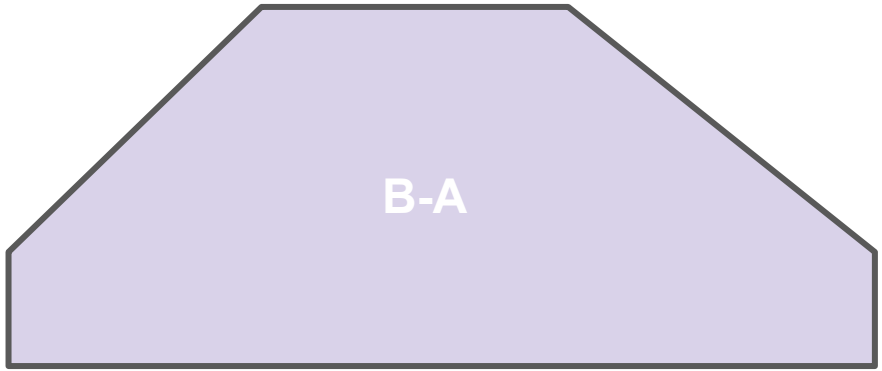
We can solve this by generating a ‘convex hull’: calculate the points only on the outside of our Minkowski difference

Convex Hull

The convex hull is the set of the exterior points on our Minkowski difference

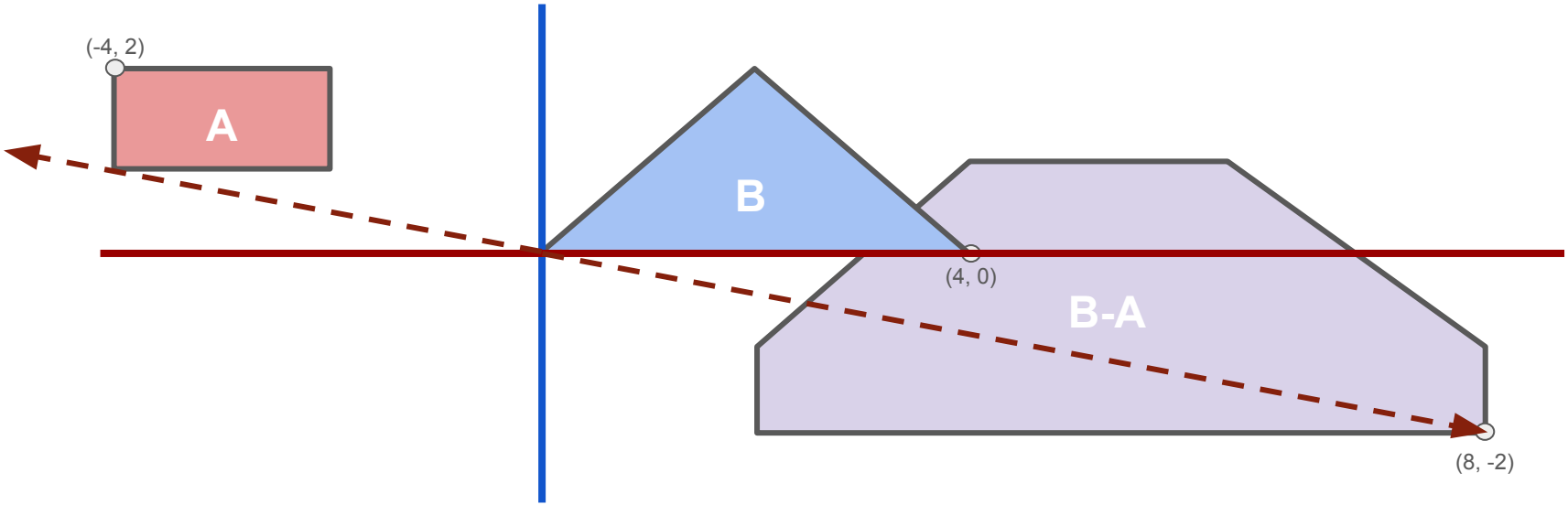
Each individual point on the convex hull is calculated from the farthest points in each shape, in any given direction.





Convex Hull

The support point at position (***x*** = 8, ***y*** = -2) is the difference between the point inside ***A*** in direction vector -***d***, and the point inside ***B***, along direction vector ***d***.



Support Points

To calculate the farthest point along vector ***d*** inside shape ***A***, we simply check the dot product of every point in ***A*** and ***d***. Calculating the dot product isn't very expensive.

$$\boldsymbol{a} = [\boldsymbol{a_1}, \boldsymbol{a_2}, \boldsymbol{a_3}, \ldots, \boldsymbol{a_n}]$$

$$\boldsymbol{b} = [\boldsymbol{b_1}, \boldsymbol{b_2}, \boldsymbol{b_3}, \ldots, \boldsymbol{b_n}]$$

$$\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a_1 b_1} + \boldsymbol{a_2 b_2} + \ldots + \boldsymbol{a_n b_n}$$

For 2D vectors, we have **2n** multiplications, and **2(n - 1)** additions, where **n** is the number of points in the shape

In 3D, this is **3n** multiplications, and **3(n - 1)** additions, or simply **O(n)**

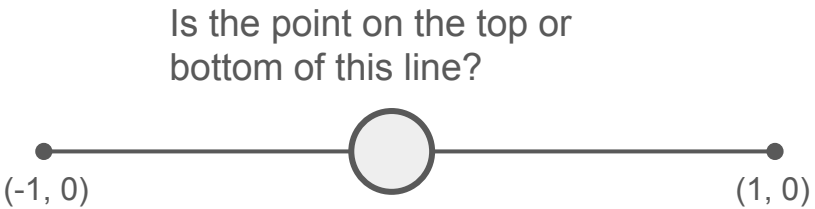
Simplex

A *simplex* is the simplest, or smallest number of points, that can be used to represent some ‘volume’.
For **1D**, we need at least two points to represent a measurable amount of space -- our line.
In **2D**, we add another point, giving us a triangle with *area*.
For **3D**, it is the same; adding a fourth point creates a tetrahedron, and we have *volume*.

The number of points required in **N-**dimensional space is **N+1**. We aren’t going to be looking at any dimensions past **3D** for our purposes.

Simplex and Timesaving

Remember: to check for a collison, we see if our Minkowski difference contains the origin. In **2D**, a line does not contain any space *(area)*, being infinitely small along one axis. Our origin might even appear to be along the line, but it can't be 'inside' the line in **2D** space. Once we have area, we can verify if the origin (or any other point) is inside.



Simplex and Timesaving

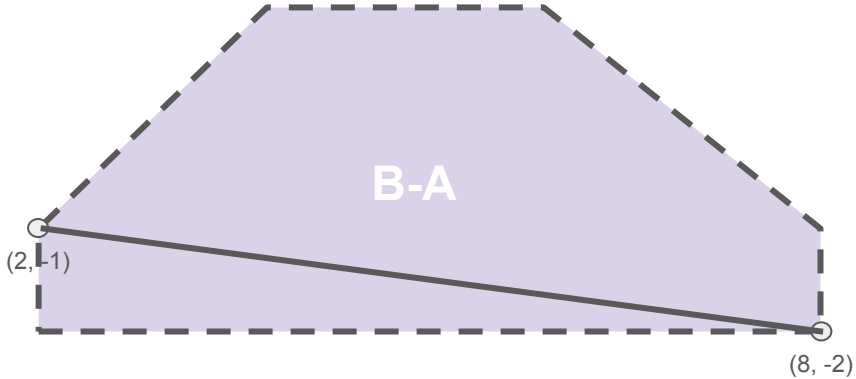
In other words, we need to at least have a simplex.

Any 3D polyhedron or 2D polygon can be broken into simplices… an **infinite** number of simplices. We should narrow that down...

Limiting Simplices

We can limit the possible simplices by only using points on our convex hull:

To limit the number of times we check, we want the largest simplex possible. Assuming that we only get points on our Minkowski difference as we pick directions, we should start by picking two points in opposite directions.

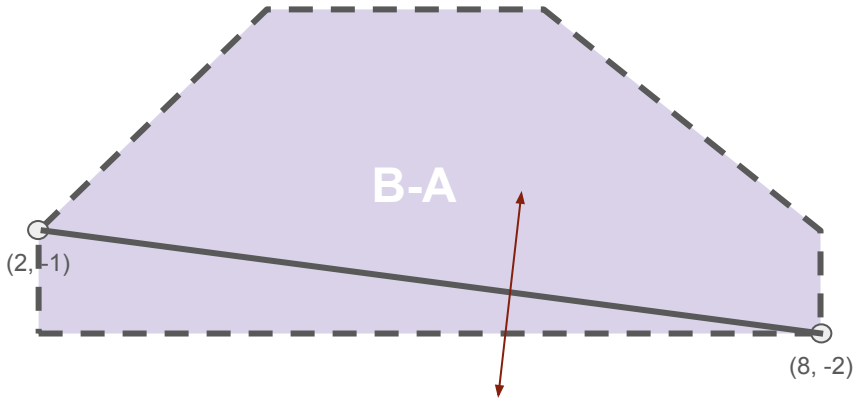


Although the minkowski difference is already generated here, that is only for reference. Calculating the full Minkowski difference this way in real-time is expensive - we would have to check every single directional vector, which is an infinite set.

Encapsulate the Origin

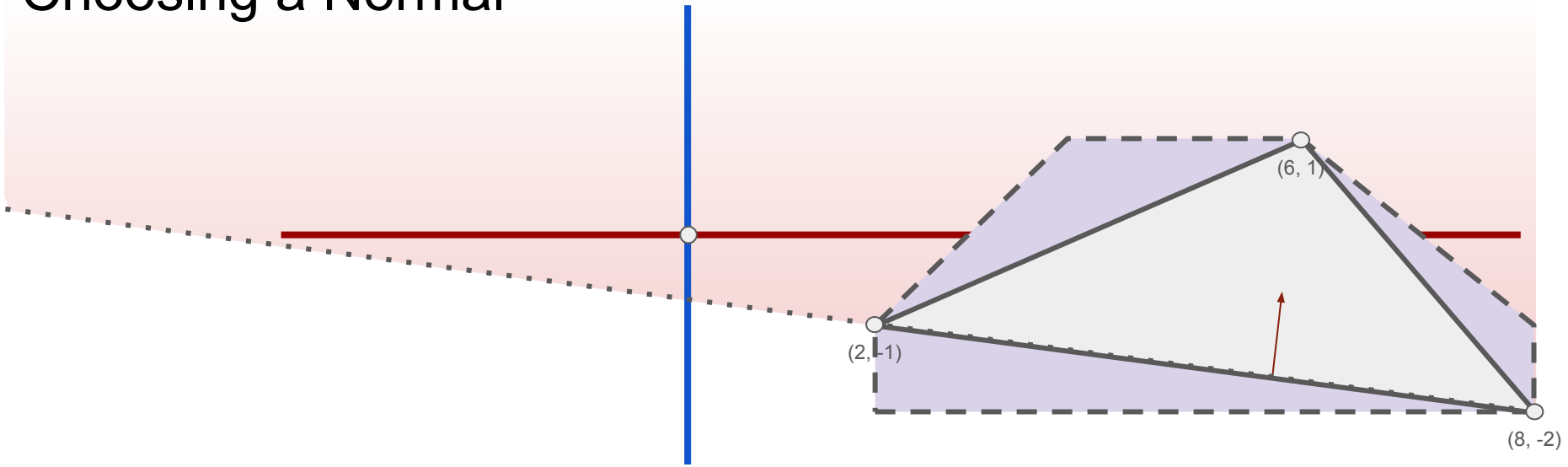
The next point should be the most likely point to make our simplex contain the origin. That gives us **two** criteria to look for:

1. Maximizes the area inside the simplex
2. ‘Moves’ towards the origin *(at least one point added to our set is closer to the origin than the closest point from the previous iteration.)*



To maximize the area of our simplex, we simply search for a new point that is perpendicular to our existing line. We can take one of our two possible ‘normals’.
Only one of these actually moves towards the origin, so we want to search along that one.

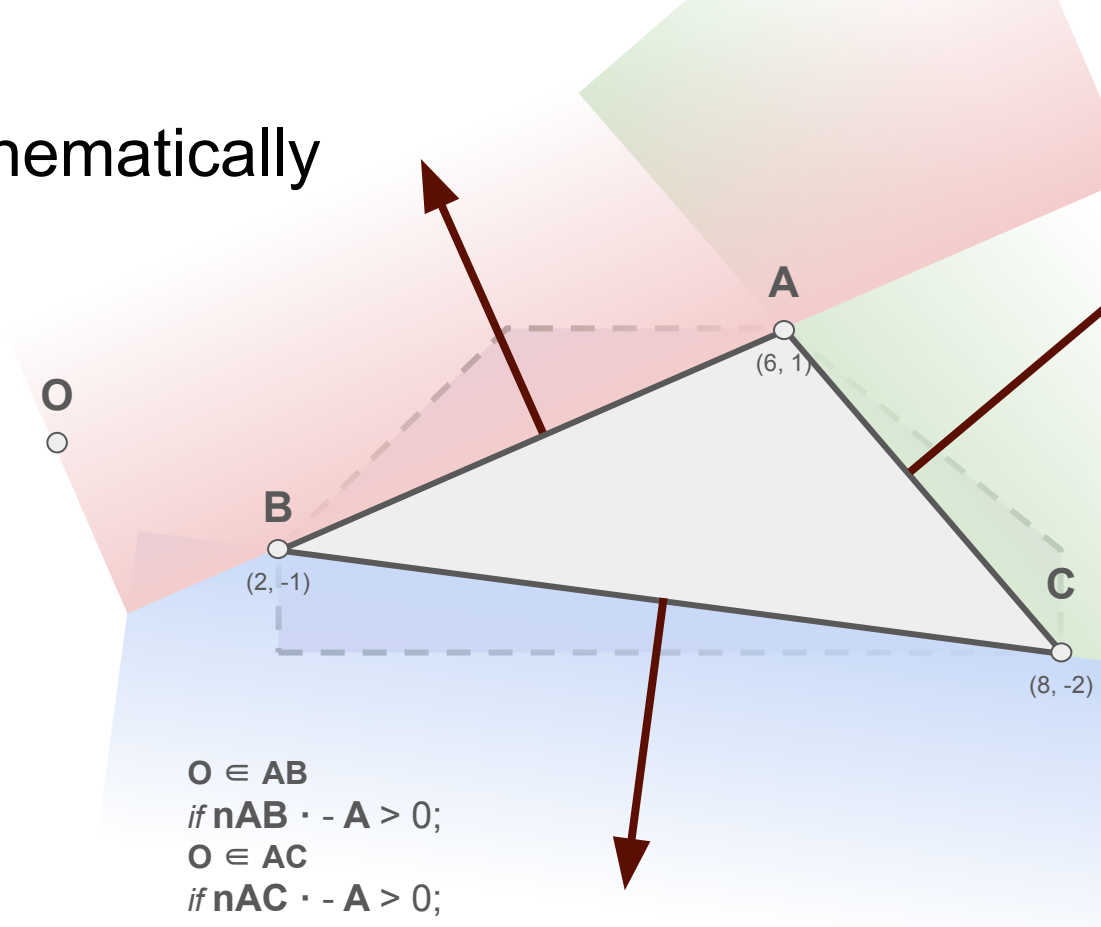
Choosing a Normal



In this case, our origin is inside the ‘upwards’ normal, so we will search in that direction

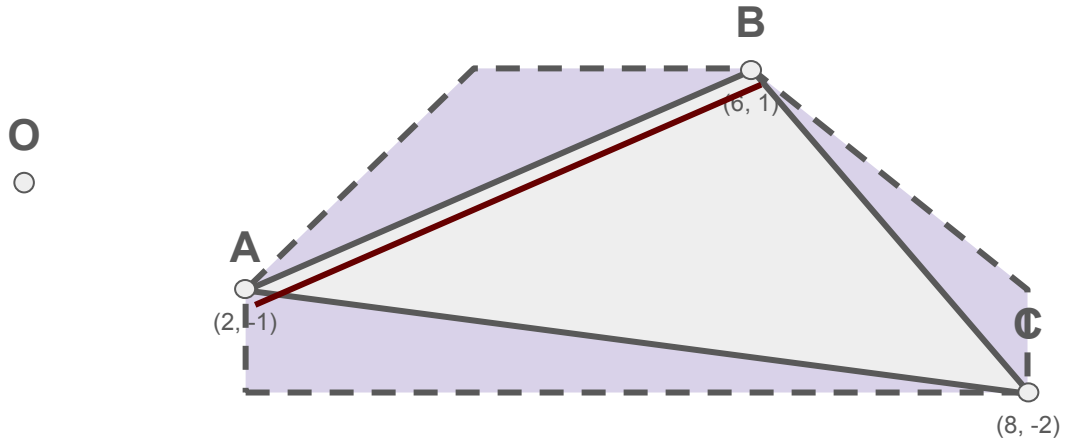
Checking the Origin Mathematically

We can calculate whether or not the origin is inside any area by evaluating the sign of the respective normal, dotted with any point on that edge.
If all values are negative, the origin is not inside any of the regions, instead, it must be inside the shape.



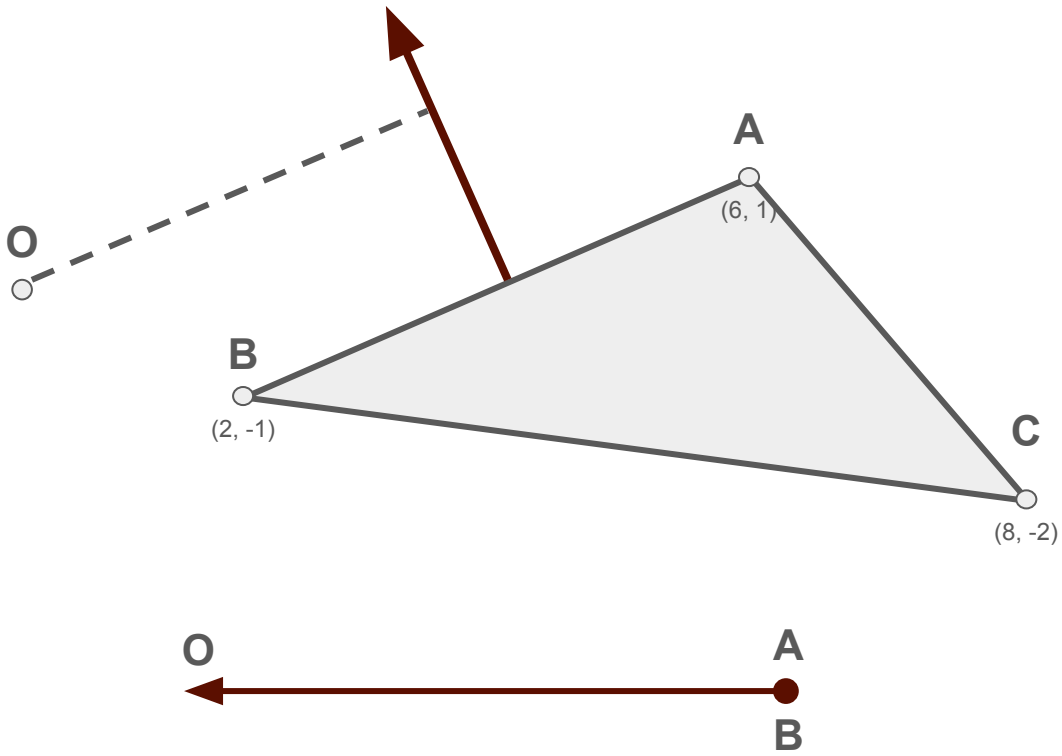
Choosing Next Point

We only want to keep one line here - the line closest to the origin. To be more accurate, we want the line that contains the *point* closest to the origin. *Visually*, it is clearly line **AB**, but *mathematically* it is a little unclear.



Choosing Next Point

We can instead ‘map’ the line we want to check, **AB**, and the origin, **O**, to a new axis.
To avoid calculating the distance from the origin to every single point in our line *(infinite)*, we can once again use the normal of that line - reducing the line to a single point!



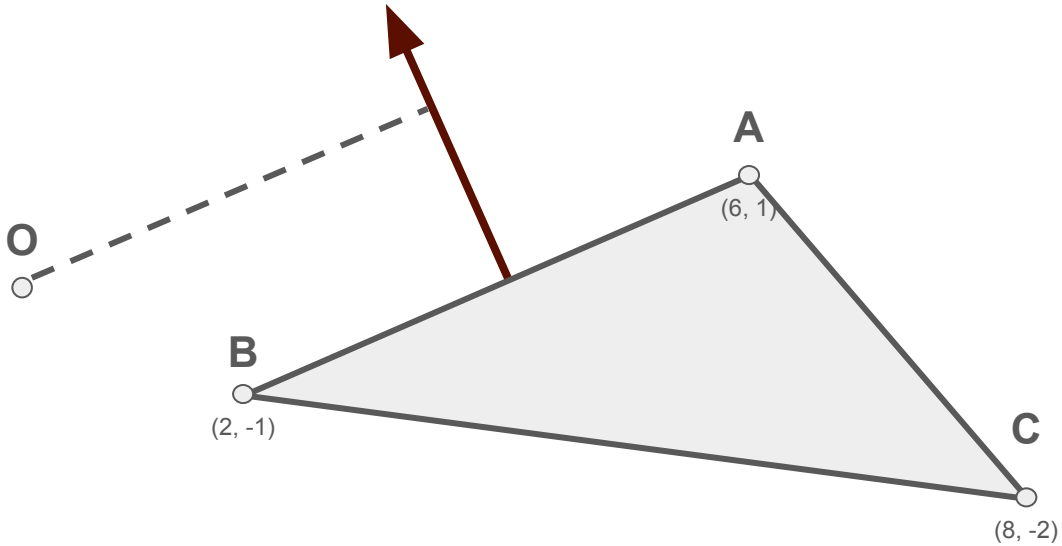
A and **B** always have the same value here, *(when projected onto the normal)* so we can use either one

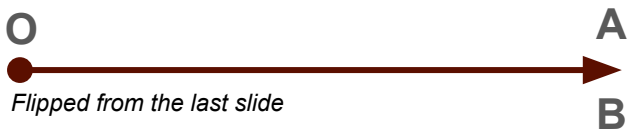
Choosing Next Point

The position on our line is a single value, or a scalar. To calculate it, we evaluate the **scalar product**, which is also known as the **dot product**.

O will be always become 0 when dotted with another vector. Our normal faces away from **AB**, so the dot product of **A** or **B** with **nAB** will be negative. We can just flip the sign

Distance = -**A** · **nAB**



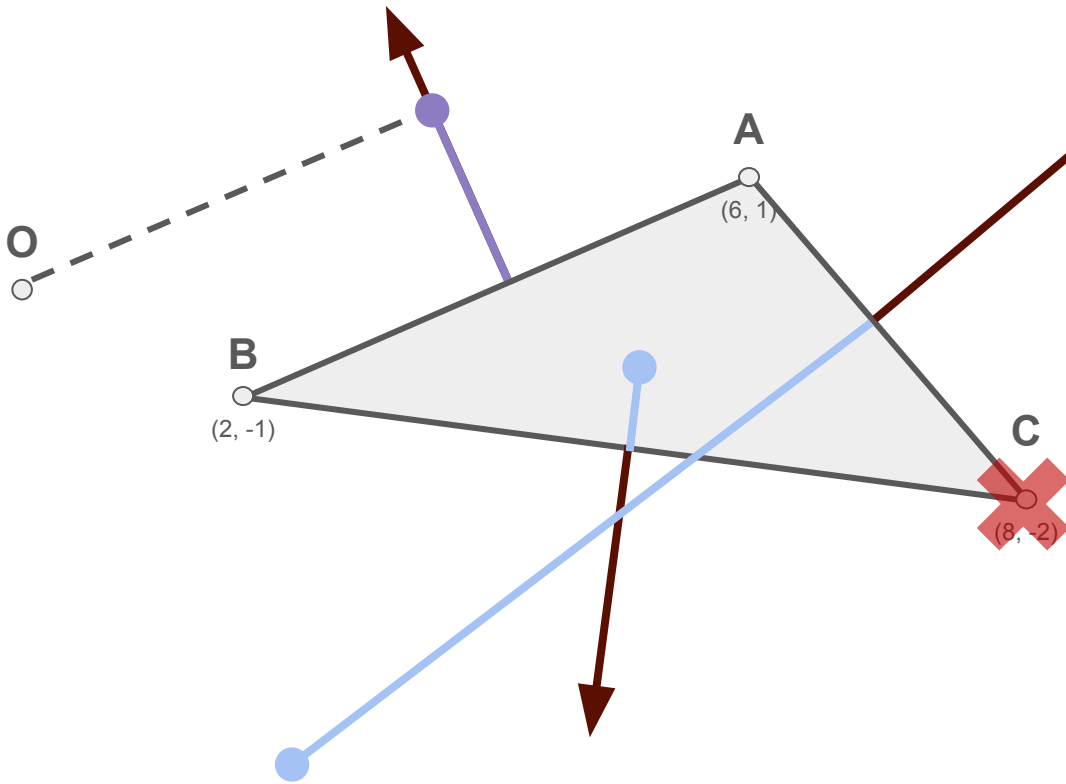


Flipped from the last slide

Removing a Point

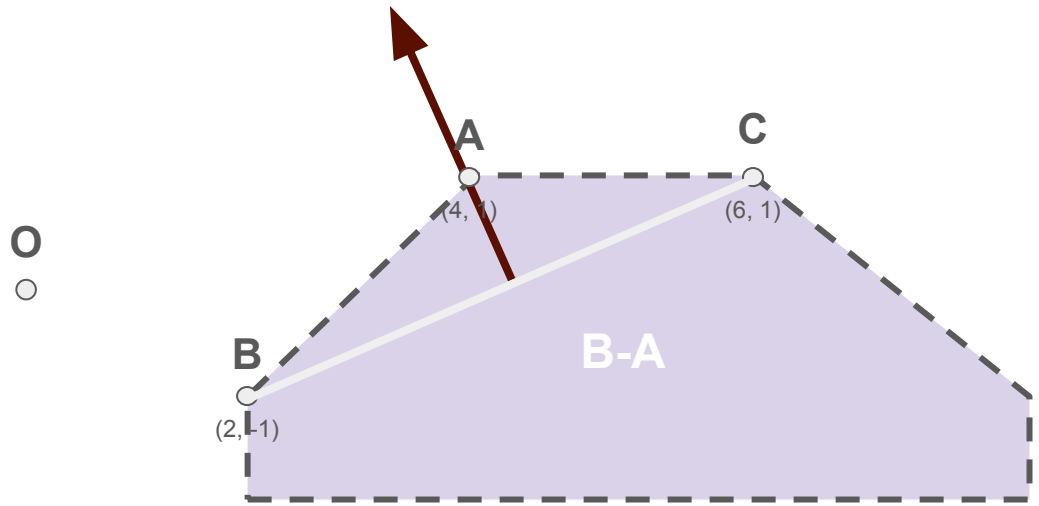
In this case, only the dot product of normal **nAB** and **A** is positive, so the point closest to the origin must be along line **AB** - we can't remove either point, so we instead remove point **C**.

We don't actually have to check if normal **nBC** faces the origin, because we added point **A** by looking towards the origin. This would make either **AB** or **AC** closer to the origin. Remember that we are trying to find a theoretical point on these lines.



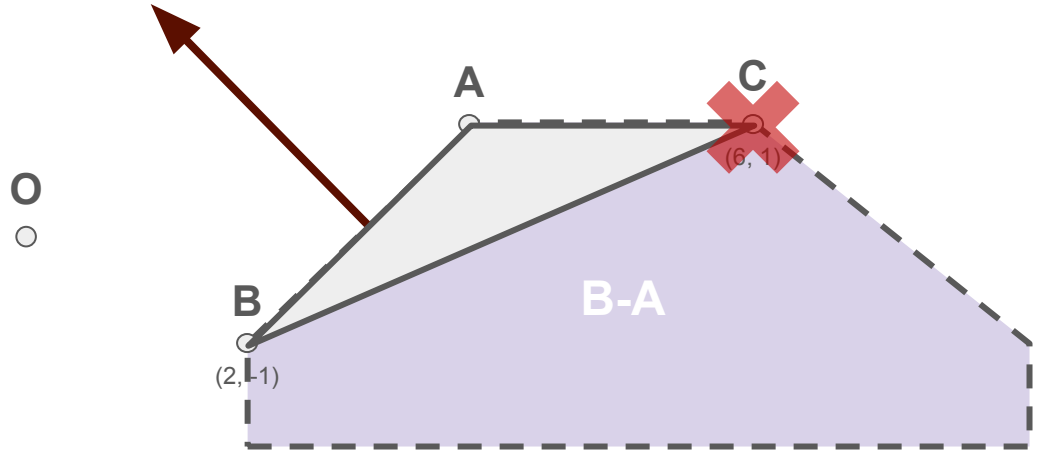
Checking Points

We can search for a support point along the normal we already calculated, and rename our points so that **A** is our newest point.



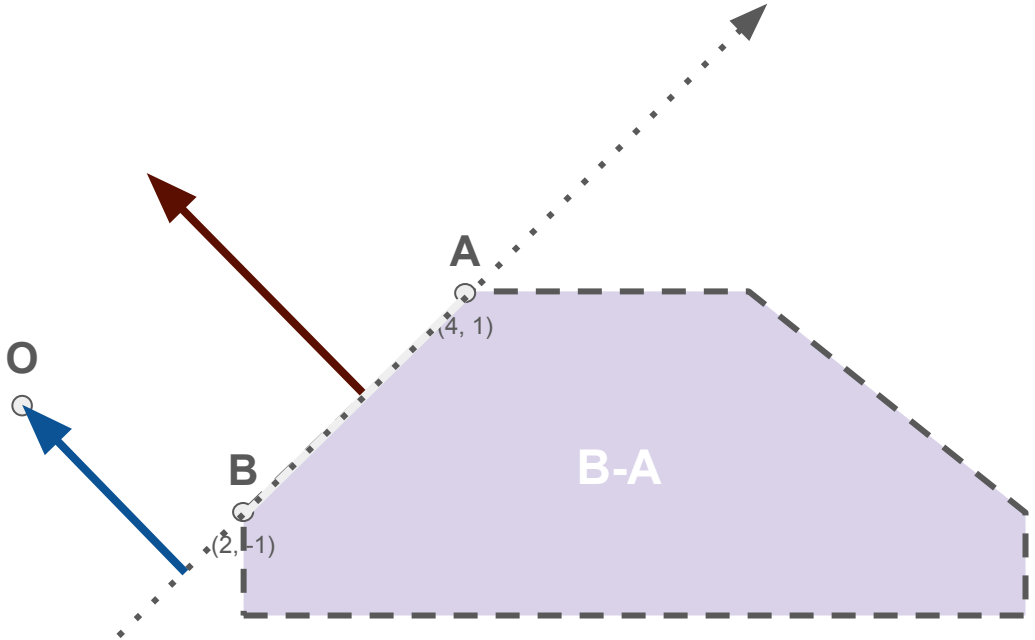
Exit Conditions

Following the rules we established before, we continue the process and remove point **C**. However, when we search for a new point along normal **nAB**, depending on how our floating point numbers round, we will find either point **A** or **B**. Since they are both in the simplex, we know that it cannot get any closer to the origin.



Distance Calculations

We know the simplex does not contain the origin, so the shapes do not overlap.
However, we can determine the minimum distance to the origin.

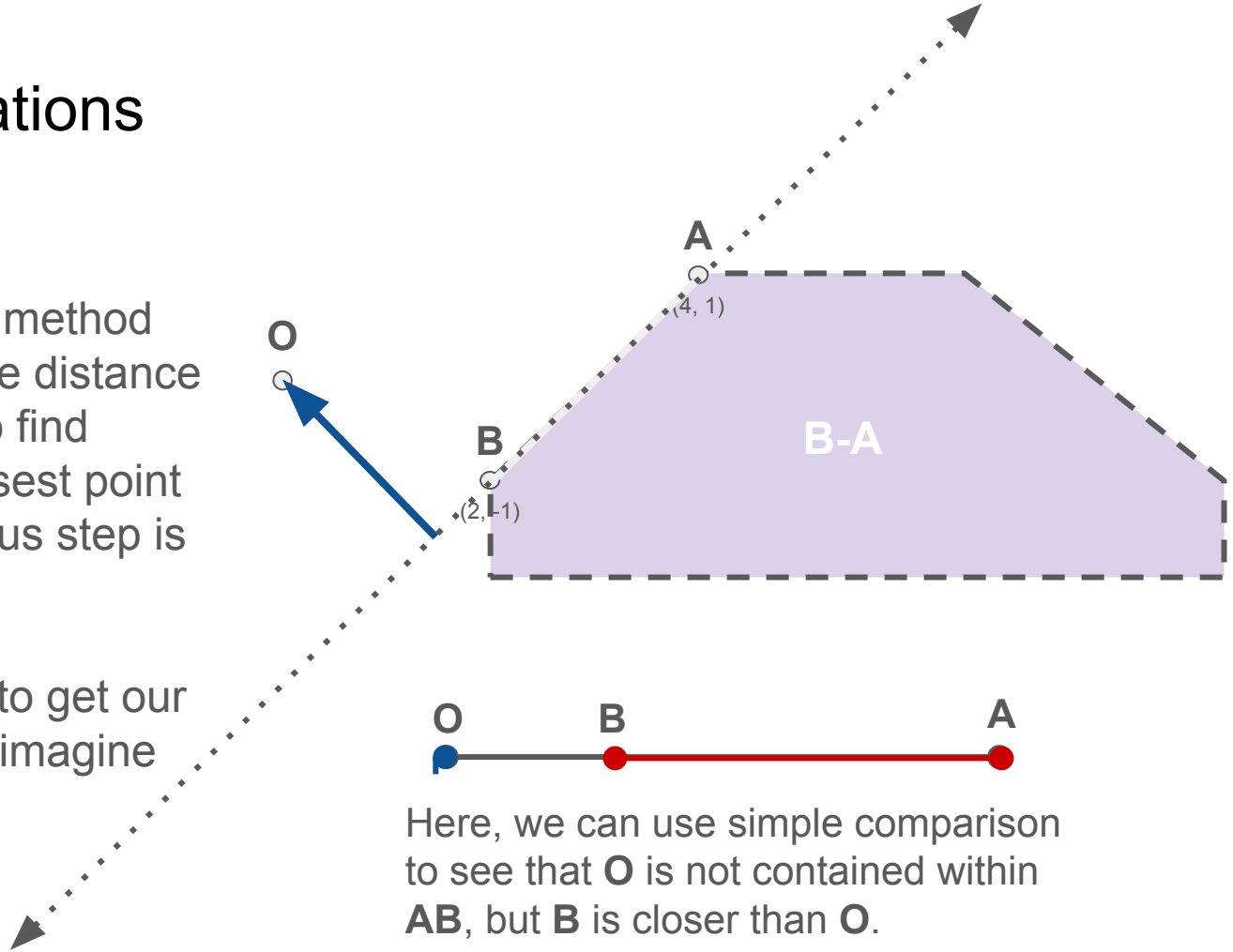


The distance value we calculated in the last step is the minimum distance from the origin to any points on the line, continuing forever. Importantly, that point is not *actually* inside the Minkowski difference. The closest point is one that is both on the infinite line **AB** and inside shape **B - A**.

Distance Calculations

We can use the same method that we used to find the distance from line **AB** and **O**, to find whether or not the closest point we found in the previous step is part of our simplex.

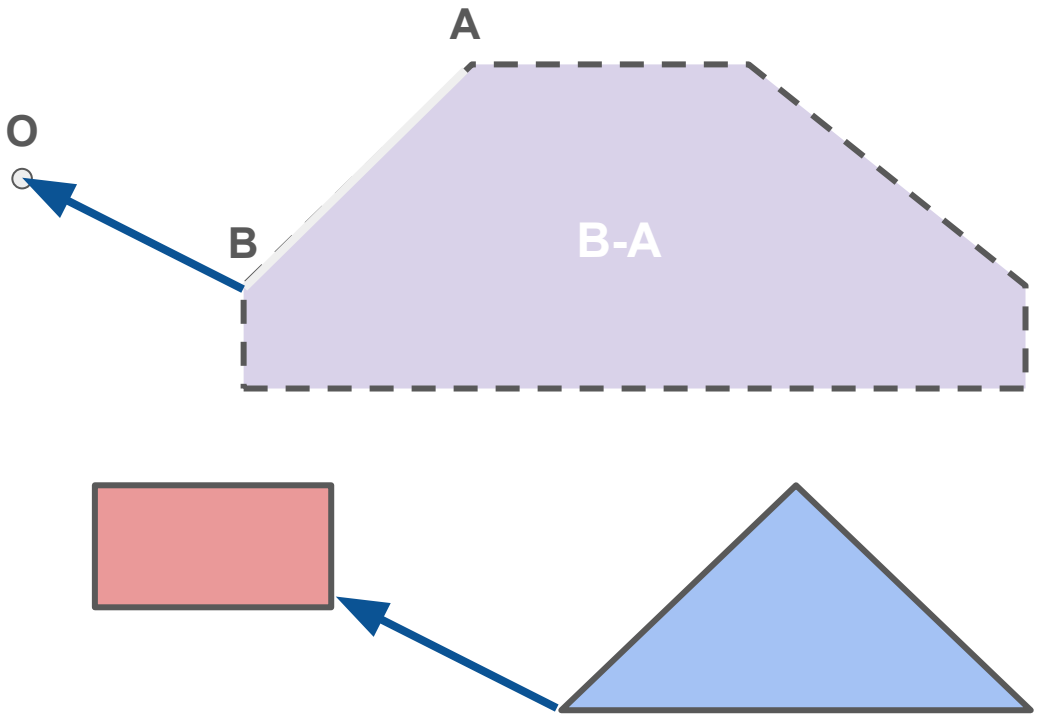
Using the dot product to get our scalar values, we can imagine them as being in 1D:



Here, we can use simple comparison to see that **O** is not contained within **AB**, but **B** is closer than **O**.

Final Result

Since point **B** was the closest to the origin on our line, **B** is also the closest to our origin. The Vector **BO** indicates both the direction and distance from **B - A** to the origin.
This is also the shortest distance between our original shapes



Edge Cases

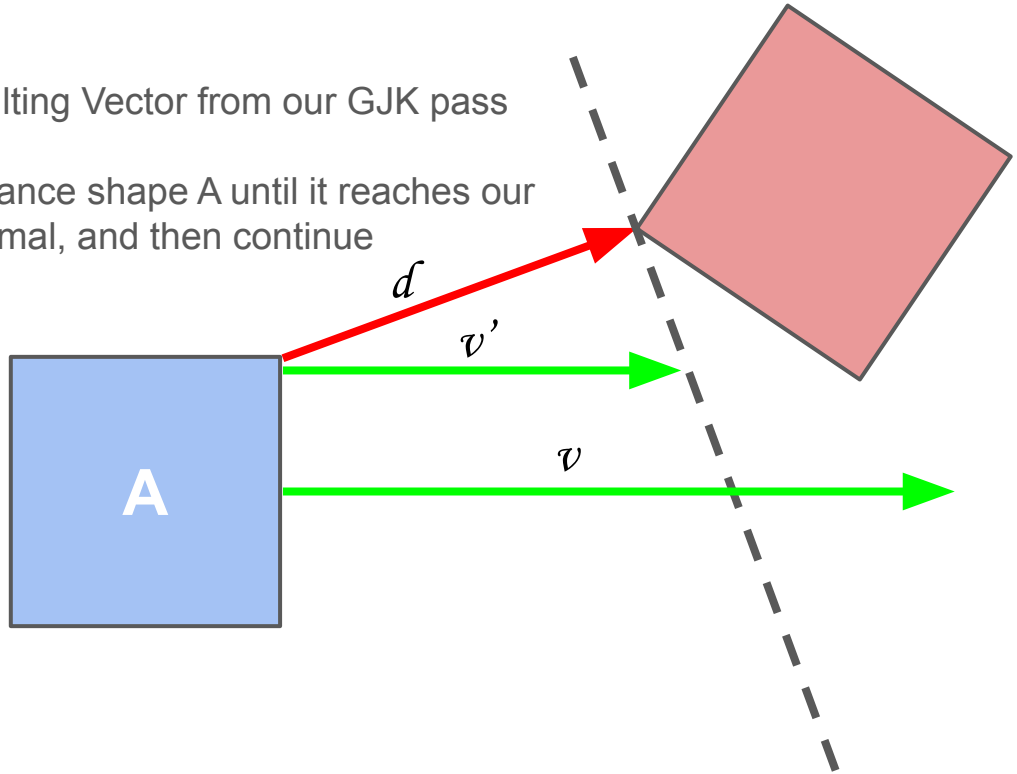
What if the two shapes are identical? What happens to the points on our Minkowski difference?

What if the Minkowski difference is a line?

Implementing GJK for CCD

d is the resulting Vector from our GJK pass

We can advance shape A until it reaches our collision normal, and then continue

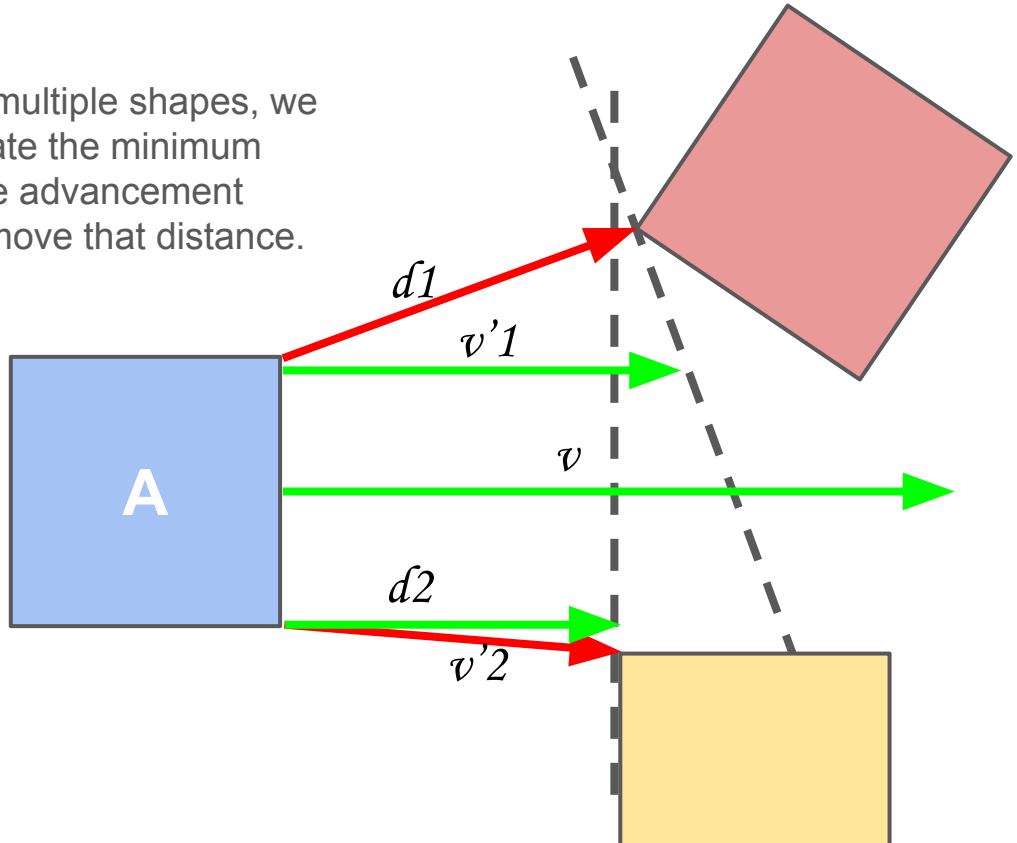


The distance we can travel before reaching this collision normal is v'.

$v' = (d \cdot vN)\, vN$

Handling Multiple Shapes

If there are multiple shapes, we must calculate the minimum conservative advancement value, and move that distance.



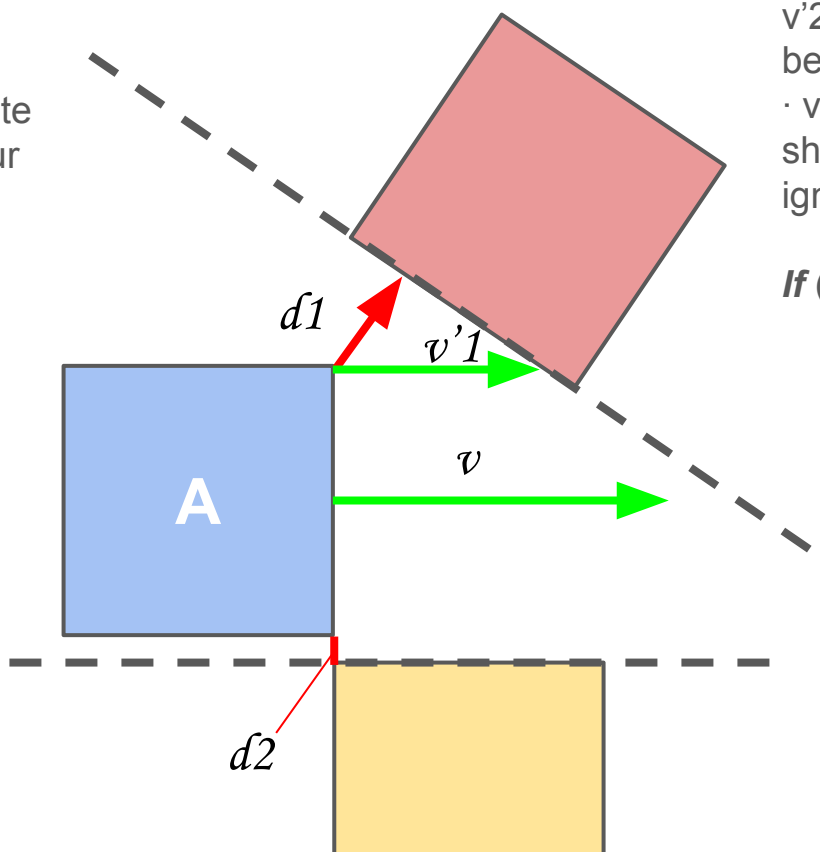
$v'1 = (d1 \cdot vN)\ vN$

$v'2 = (d2 \cdot vN)\ vN$

$v' = \text{Min}(|v'1|, |v'2|, \ldots, |v'n|)$

Handling Multiple Shapes

After advancing, we recalculate the distance between all of our shapes again.



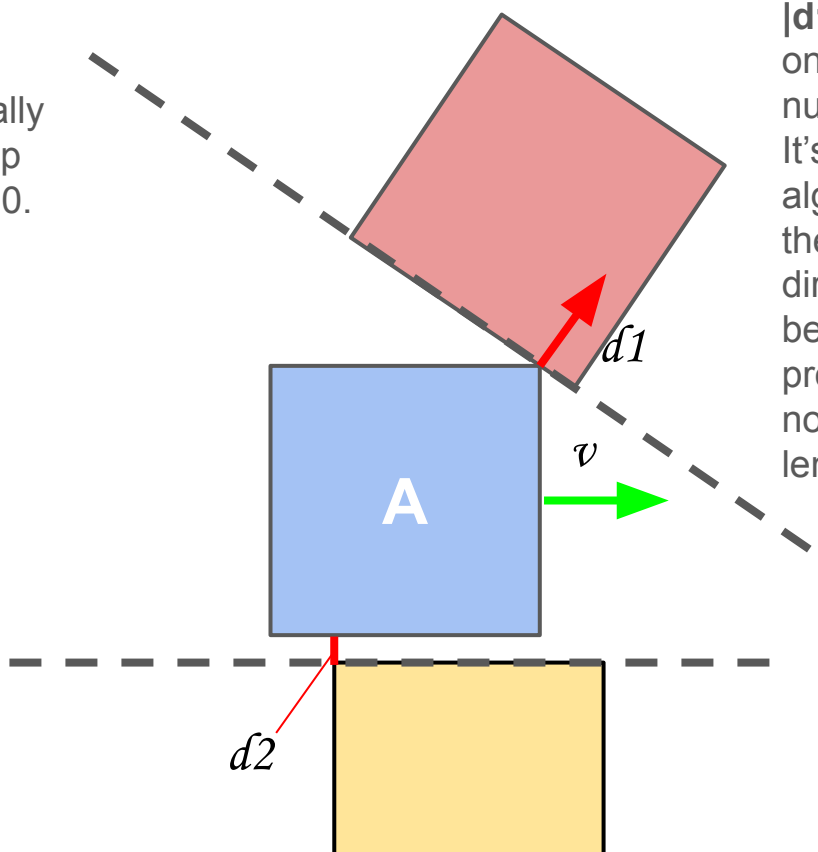
v'2 would become zero, because the expression: (d2 · vN), evaluates to 0, so we should catch this case and ignore it.

If (d2 · vN) ≤ 0; **ignore**

Handling Multiple Shapes

Now that our shape has actually collided (**|d1|** ≈ 0), we can stop advancing, even though **|v|** ≠ 0.

Unless we want to start calculating a deflection...



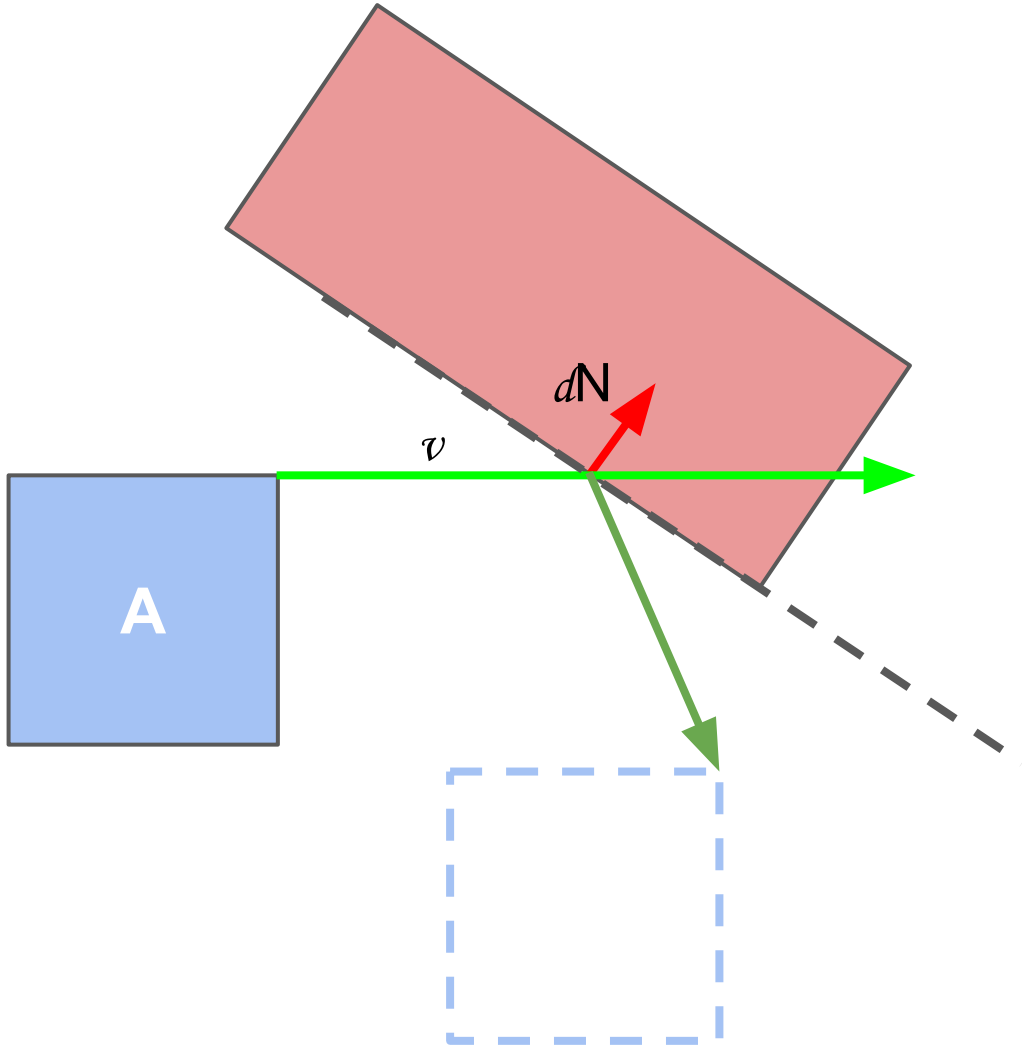
|d1| is almost 0, depending on how our floating-point numbers are handled. It's important that the GJK algorithm can return both the collision distance and direction separately, because we may run into problems if we try to normalize a vector of 0 length

Reflections

Typically, our object would bounce off of the other one *(if the collision is perfectly elastic and there is no friction.)*

This would be fine if we were simulating all forces, but this isn't the case right now.

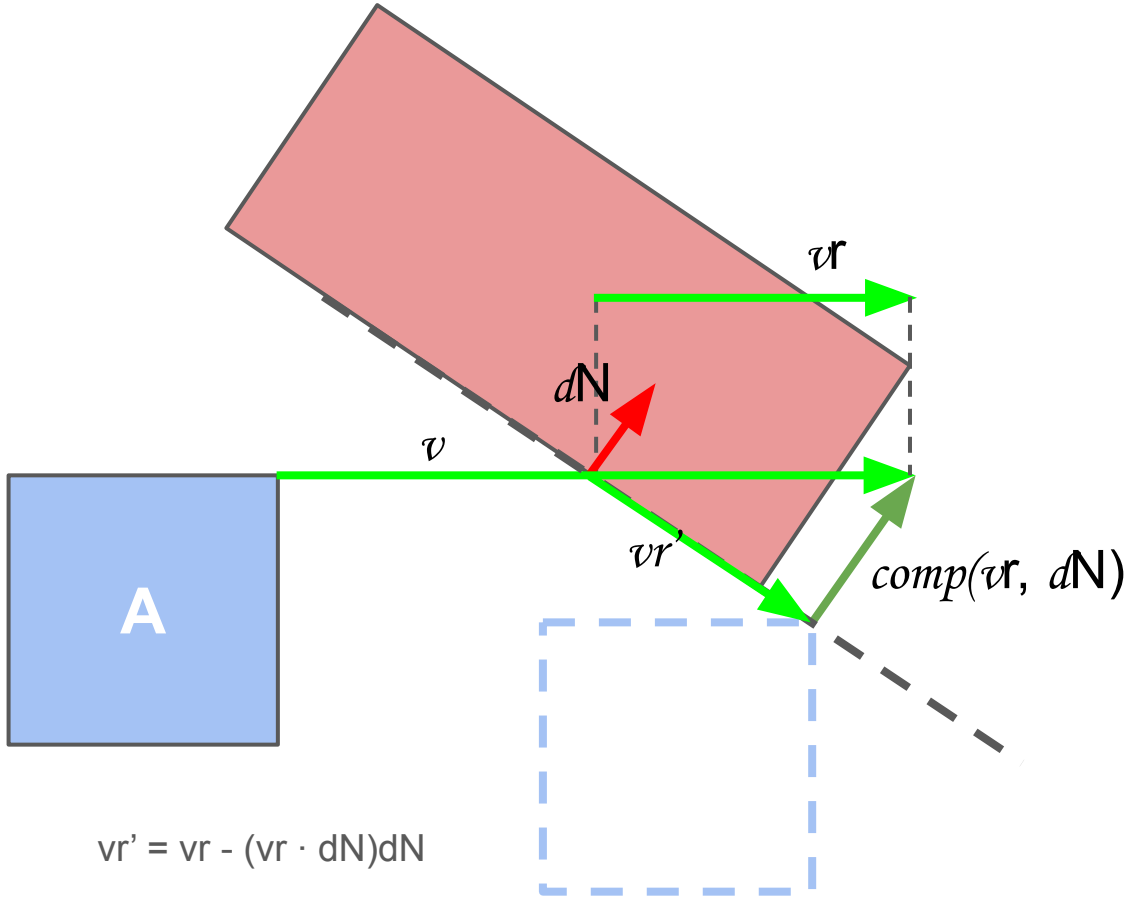
With simpler dynamics, we can assume that the initial force would be constantly applied, that is, our object would prefer to move in direction v**N** whenever possible.



Reflections

If we assume that whatever force is being applied constantly (such as if shape **A** is the player in a platformer), we can approximate how shape **A** would slide.

vr is the remaining velocity component after our advancement, so we can simply subtract the component of our velocity that would go ‘into’ the wall.

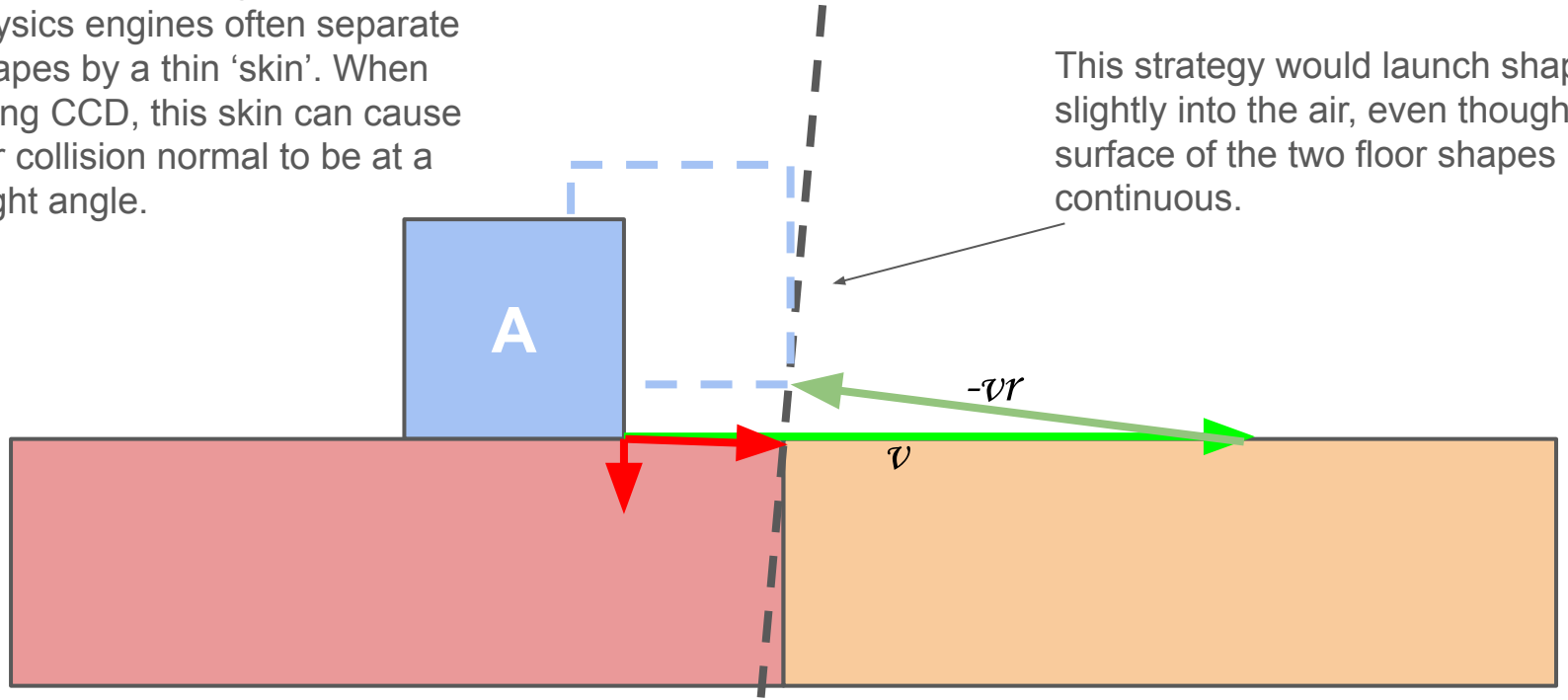


$vr' = vr - (vr \cdot dN)dN$

Ghost collisions

To prevent floating point errors, physics engines often separate shapes by a thin ‘skin’. When using CCD, this skin can cause our collision normal to be at a slight angle.

This strategy would launch shape **A** slightly into the air, even though the surface of the two floor shapes is continuous.



Ghost collisions

We would have to calculate many advancements, *just* to find out that our collision normal will eventually be perpendicular to our velocity...