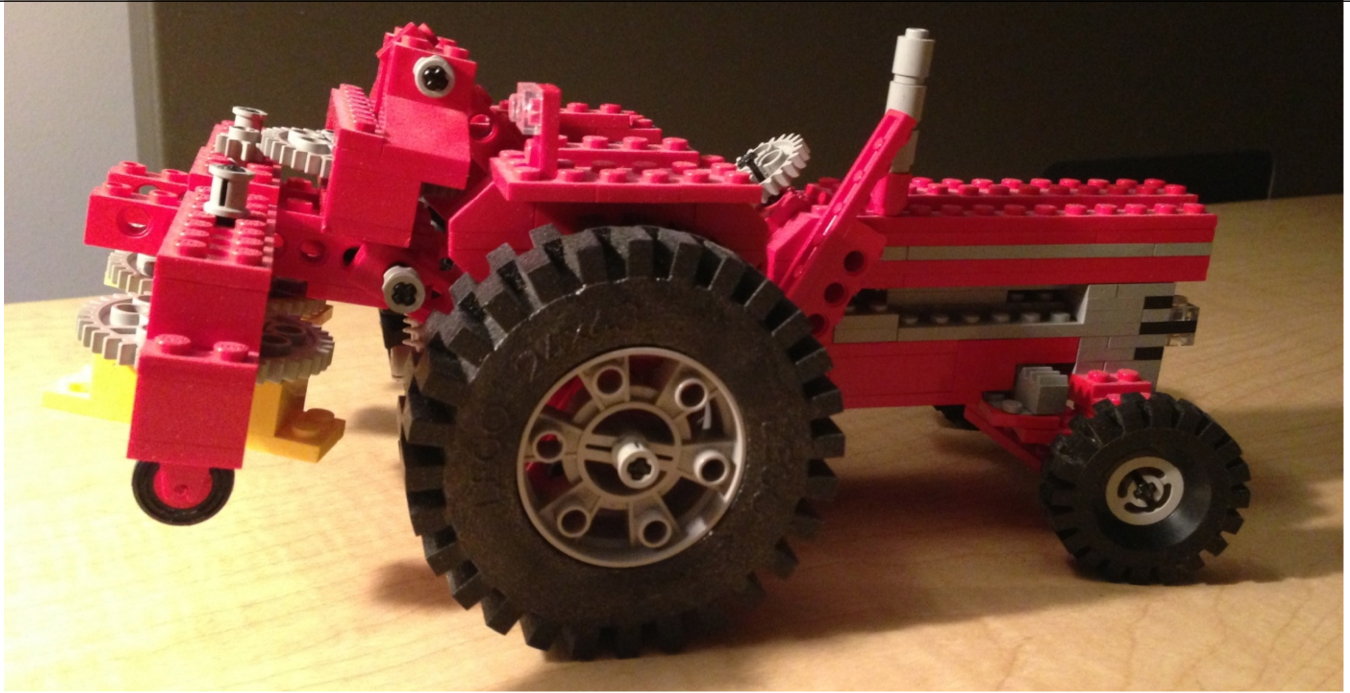
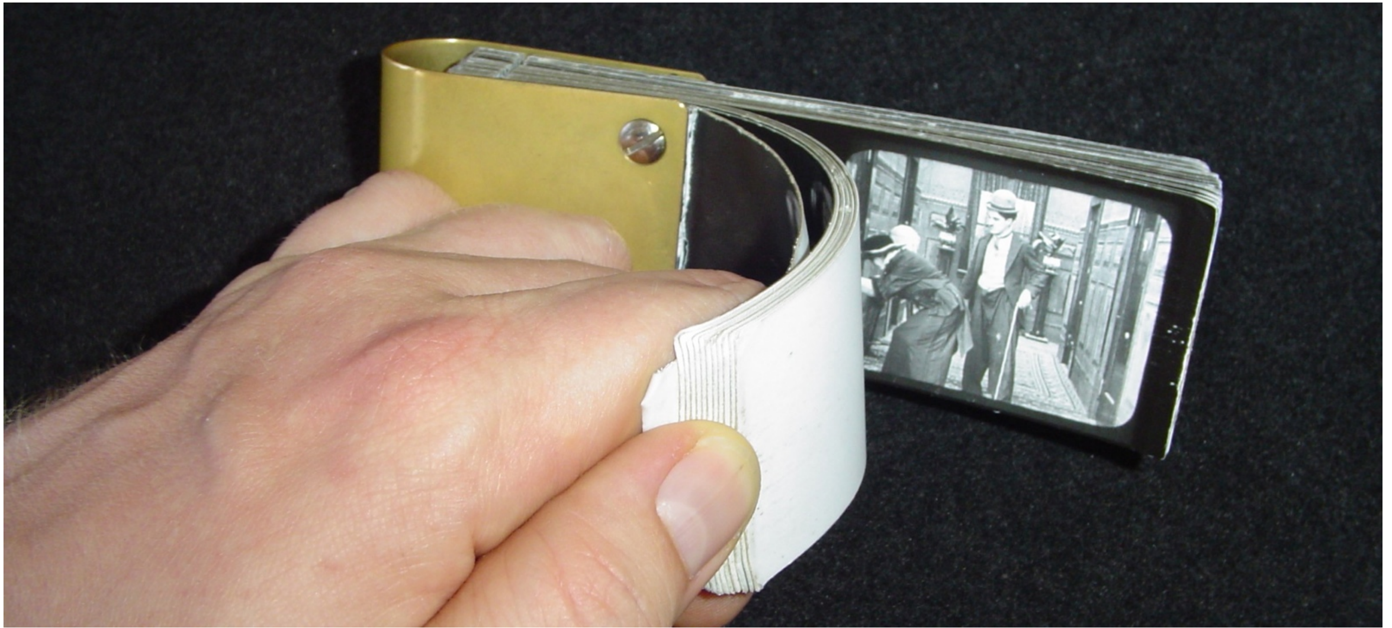


# Continuous Collision

Erin Catto, @erin_catto
Principle Software Engineer, Blizzard

Expert Lego Set Number 952, 315 pieces, 1978

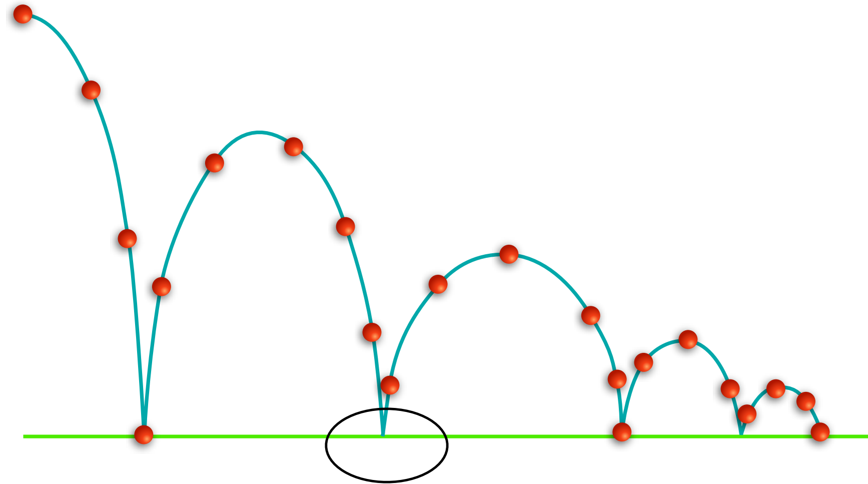
## Games are fancy flipbooks



Games are just fancy flip books. We draw discrete frames that are snapshots of a moving world. Of course the difference is that in a game, the player can influence what is drawn in each frame.

Physics engines usually operate in the same way. The engine executes discrete time steps, usually of a fixed size, that march the simulation forward in time. When we do this, the physics engine can miss events that happen in between frames.

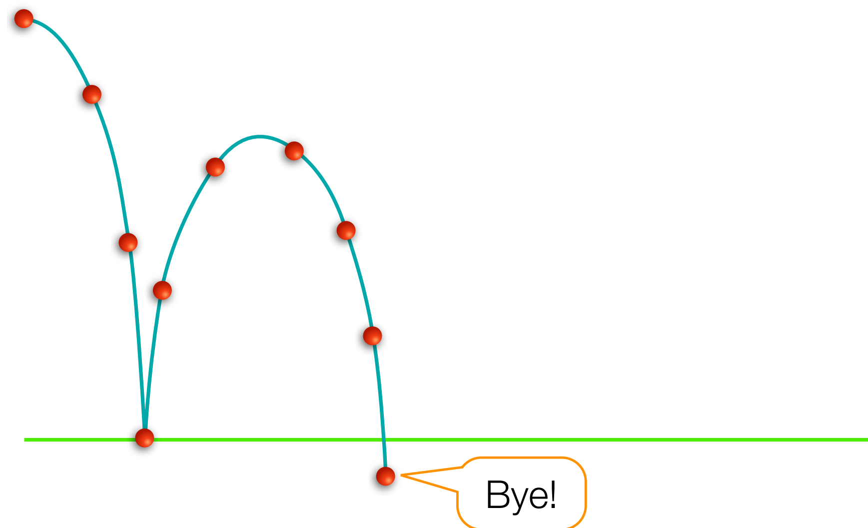
## Discrete steps lead to missed events



Consider a bouncing ball. Discrete time steps are good enough for most of the simulation.

However, suppose the discrete time steps skip over the time where the ball hits the floor. How can the ball bounce if it never touches the floor? Well it won't and this is a big problem for physics engines.

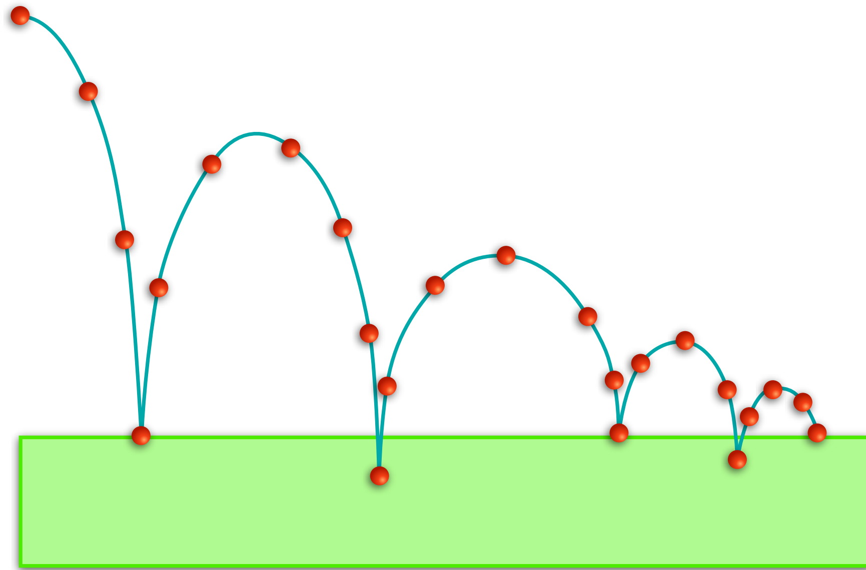
## Solution #1: Ignore the bug




If you ignore the missed collision you can get tunneling. In this case the ball falls out of the world.

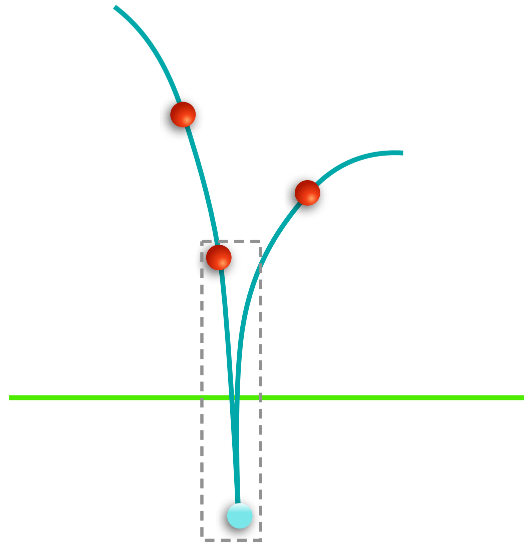
Many physics engines don't address this problem and leave it up to the game to fix (or ignore the problem). In some cases this is a reasonable choice. For example, if two pieces of debris pass through each other quickly in a game, you may never notice and it doesn't effect the outcome of the game.

## Solution #2: Make the floor thicker



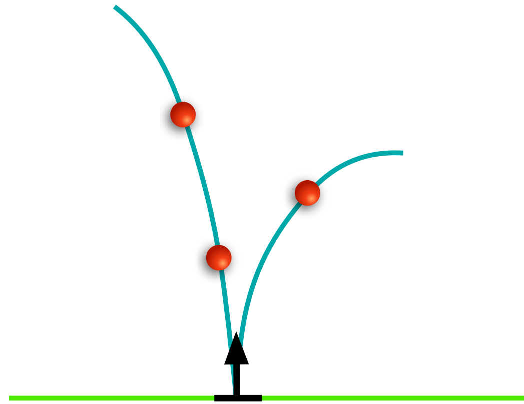
You can prevent missed collisions by using more forgiving geometry. In this case I made the floor thicker to catch the ball. However, at higher speeds the collision can be missed again. We can solve that by limiting the maximum speed of moving objects.

## Solution #3: Add speculative contact points



A recent development in physics engine technology is the use of speculative contacts. This method looks for potential contact points in the future and adds additional constraints to the contact solver.

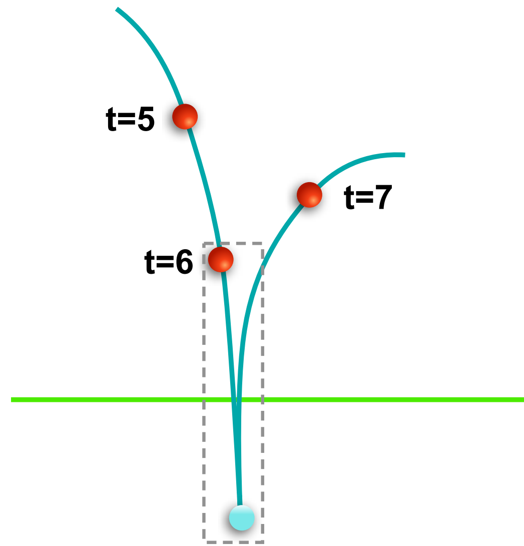
## Solution #3: Add speculative contact points



These additional constraints limit the velocity towards the speculative contact point. This effectively slows down the ball before it reaches the floor. This method works well in practice but there are a couple downsides:

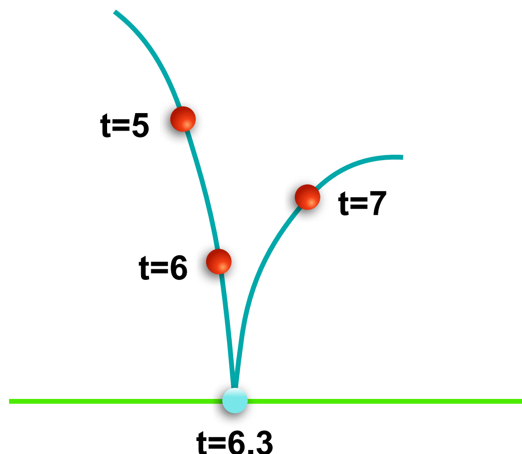
- restitution needs special handling because the velocity will be zero when the ball hits the floor
- speculative constraints may be invalid and cause ghost collisions where an object may appear to hit an invisible wall

## Solution #4: Use the time of impact




Another way to prevent missed events is to compute the time of impact between objects. In this case we wrap a bounding box around the movement of the ball and check that against the ground. Since the bounding box intersects with the ground, we compute the time of impact.

## Solution #4: Use the time of impact




The time of impact is some time between the discrete time steps where a collision occurs. Once we have the time of impact we have a couple choices:

- we can stop the ball at the time of impact
- we can move the ball to the time of impact and then perform sub-step

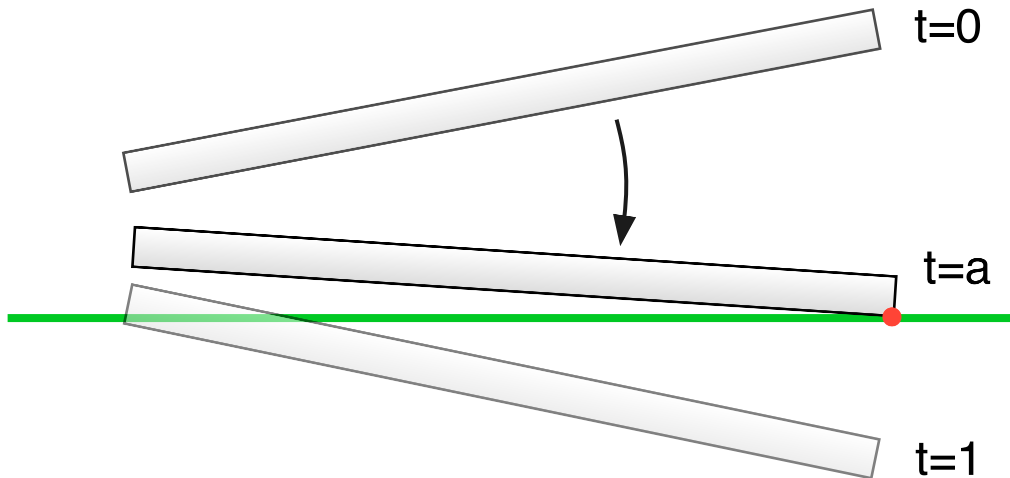
Stopping the ball a the TOI means the ball loses some time. This can lead to a visual hitch in the motion. Sub-stepping eliminates hitching but it can become quite CPU intensive.

## Tunneling is a nasty physics bug

<Show two videos, Diablo3 with and without continuous collision detection.>

Diablo 3 uses the custom Domino physics engine. Domino provides continuous collision handling using a TOI solver and sub-stepping. Since this is expensive, we only use continuous collision for collisions between dynamic and static objects. We need continuous collision detection because we have actions that can throw ragdolls and debris around at high speeds. We don’t want these objects to fall out of the world or get stuck in walls.

Goal: a method for computing the time of impact between two convex polygons




The time of impact is the time during a discrete step when two shapes first begin to touch.

So I will mainly be covering a geometry problem. I will not go into detail on the resolution of time of impact events. I believe that TOI resolution is still an open problem (at least for me), so hopefully I'll have some more details on this at a later date.

Nevertheless, I believe that computing the time of impact is an important problem in game physics.

## There are several ways to compute the time of impact

- Ray casts
- Linear shape casts (van den Bergen)
- Conservative advancement (Mirtich)
- Brute force (Redon)

There are a number of techniques to compute the time of impact with varying levels of accuracy and performance.

1) You can perform one or more ray casts between the two discrete states to estimate the time of impact between two shapes. For circles this is quite effective and fast, but doesn't work well for oblong shapes. I used ray casts for Tomb Raider: Legend.
2) You can use a linear shape cast between two positions. This ignores rotation but it might be a good starting point.
3) Conservative advancement considers rotation and is accurate. CA iteratively computes the time of impact by balancing the shape-to-shape distance with the relative velocity. I used CA as an attempt to improve on my old ray casting technique. However, I found many cases where CA would need hundreds of iterations to converge. It is often too conservative when shapes are close together, but not touching.
4) There is also a brute force approach where each edge and vertex is swept against every edge and vertex on the other polygon. This generates several non-linear equations that have to be solve. Often the swept motion is interpolated with functions designed to yield solvable polynomials. This approach has an O(N^2) cost in 2D and O(N^3) cost in 3D. In other words, it is horribly expensive.

## Desirable qualities of a time of impact algorithm

- Oblong shapes
- Rotation
- Convergence
- Accuracy is *good enough* for games

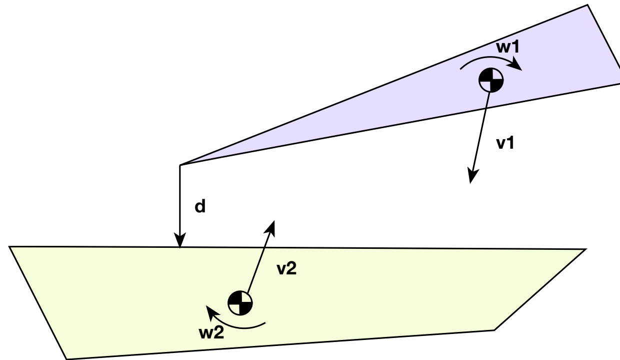
After implementing a ray-cast solution for Tomb Raider, I began to pursue better methods.

The ray-cast method is too crude and it doesn't work well with oblong shapes. I had to limit the aspect ratio of shapes in Tomb Raider to prevent tunneling. I would rather not impose this limitation on content creators.

I wanted the algorithm to handle rotation if possible. I also wanted the algorithm to have reliable convergence to avoid performance spikes. I was willing to sacrifice some accuracy to achieve this. After all, we are making games, not space ships.

So I began to search for a new method. I chose conservative advancement and I found that it worked quite well, in many cases.

## Conservative advancement




$$\left[(v_2 - v_1) \cdot \frac{d}{\|d\|} + \|\omega_1\| r_1 + \|\omega_2\| r_2\right] \Delta t \le \|d\|$$

Conservative advancement works by considering the distance between two moving shapes. If the distance is non-zero then the shapes can move by some amount without driving the distance to zero.

The formula above states that the maximal motion along the distance vector must be less than the distance between the shapes. This considers the linear and angular velocities of both bodies about the center of mass. The formula uses radius scalars that are the maximum distance between a point on the shape and its centroid.

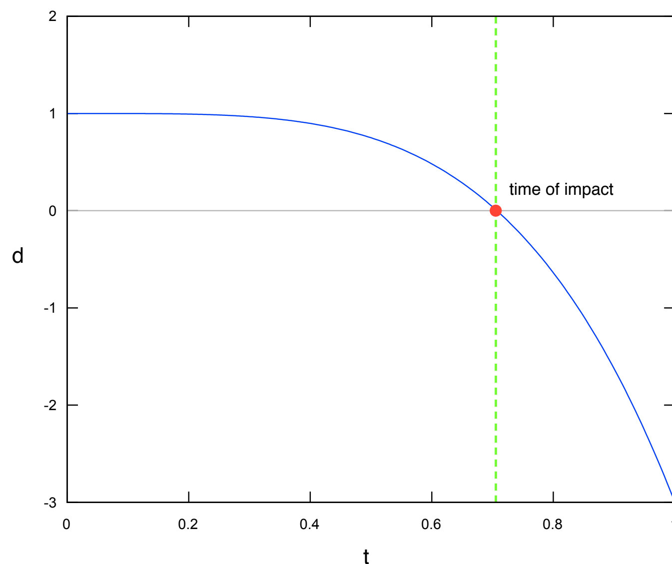
## Conservative time of impact

$$\Delta t = \frac{\|d\|}{(v_2 - v_1) \cdot \frac{d}{\|d\|} + \|\omega_1\| r_1 + \|\omega_2\| r_2}$$

We can solve the motion bound for the time step. This time step is guaranteed to be safe, but there may still be a significant gap between the shapes. So we need to restart the algorithm from the new configuration (and new distance vector).

We need a precaution when implementing this algorithm. We must use an accurate algorithm for integrating the angular velocity to make sure the rotation does not overshoot and lead to penetration.

## Conceptually the time of impact problem is a root finding problem



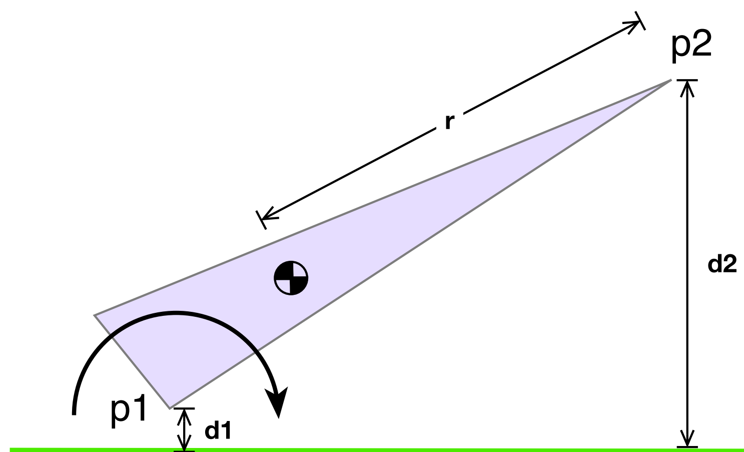

$$d(t) = 0$$

$$t \in [0, 1]$$

We want to find a time when the distance between the shapes is zero. In particular the time must be between 0 and 1. There may be zero or more roots in this region. We want to find the first root or determine that no root exists. When no root exists there is no collision and we are done.

The problem with conservative advancement is that it tries to solve a root finding problem from one side. The distance never goes negative. This hinders our ability to use high performance root-finding algorithms.

## A worst-case scenario for conservative advancement

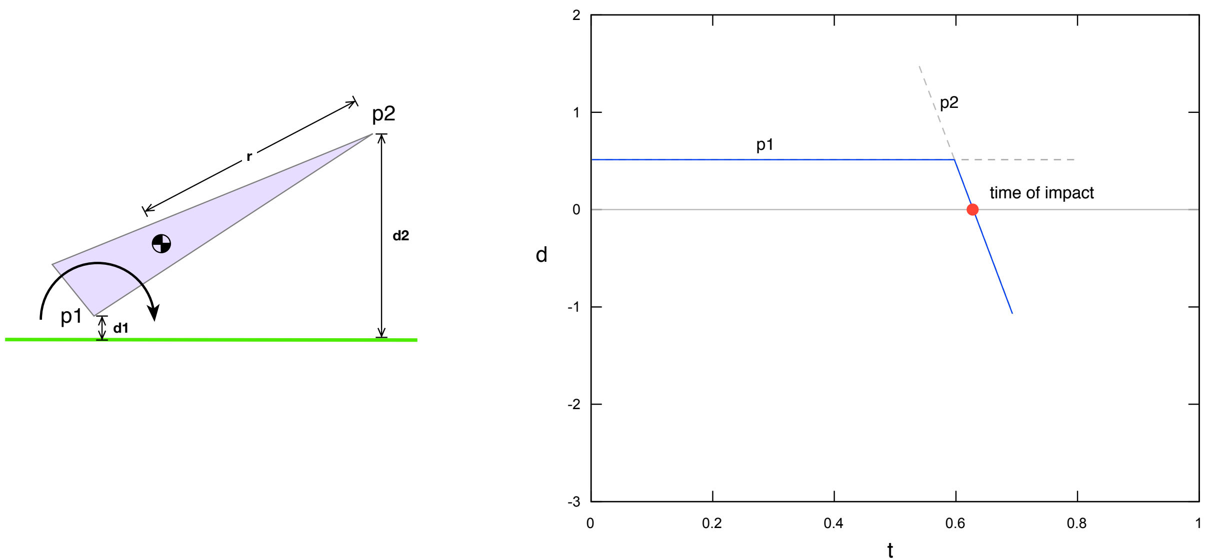



$$\Delta t = \frac{\|d_1\|}{v \cdot \frac{d_1}{\|d_1\|} + \|\omega\| r}$$

Let’s consider a worst-case scenario for conservative advancement. This figure shows a moving triangle colliding with an infinite stationary plane. The triangle is rotating around point p1. The point p1 is close to the ground plane but is not getting any closer as the triangle rotates. Actually what will happen is that point p2 hits the plane first.

The CA formula shows that the time step is proportional to the closest distance, d1. But d1 can be arbitrarily small making the time steps arbitrarily small. This leads to an arbitrarily large number of iterations. In other words, the performance sucks.

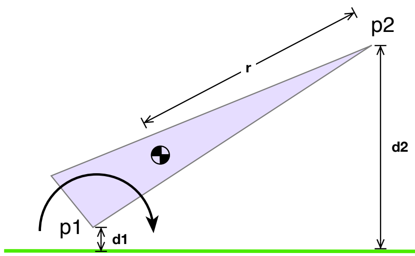
# Worst-case scenario as a root finding problem

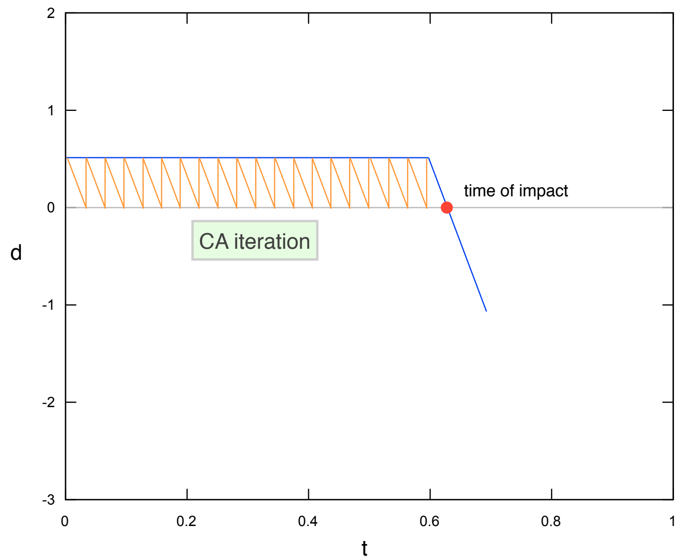



Here we see this scenario plotted as a root finding problem (the graph is approximate).

The graph shows the distance between the triangle and the plane as a function of time. At t=0 the distance is equal to d1 and the distance remains constant until point p2 swings down below point p1. This creates a kink in the curve.

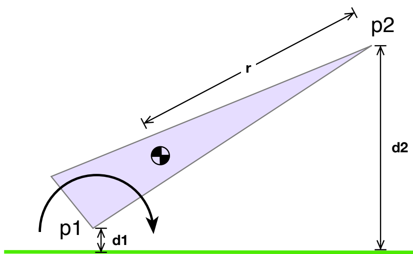
## Worst-case scenario as a root finding problem

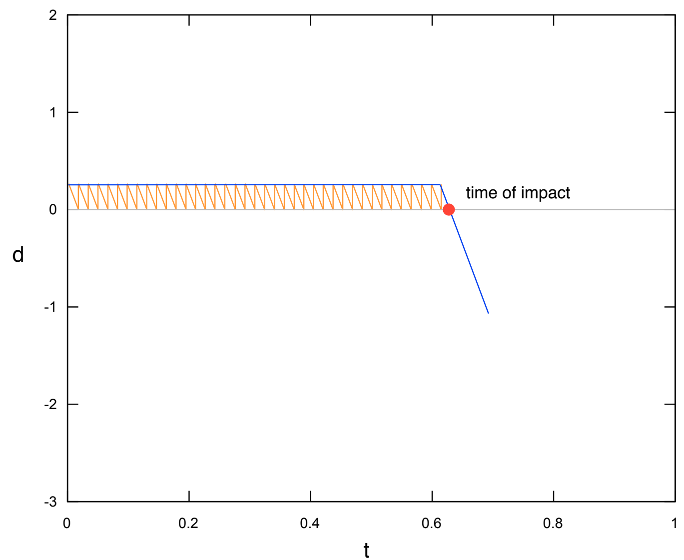



Unfortunately, conservative advancement uses the worst-case slope, that of point p2, to advance time. This leads to many iterations that ping-pong towards the time of impact. The algorithm is too conservative.

## Getting closer makes it worse



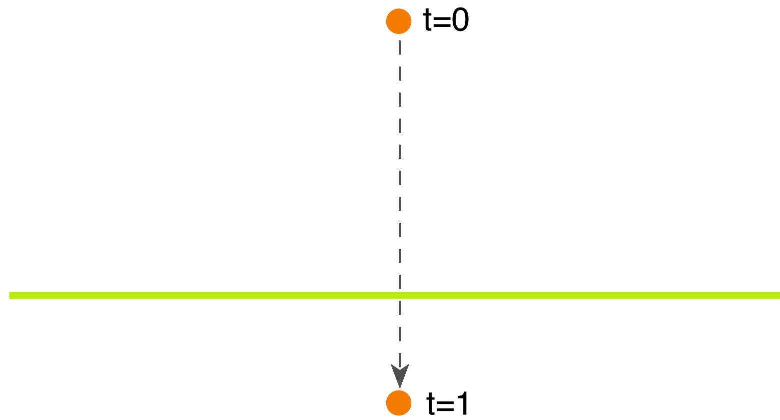




If d1 is divided by 2, then the number of iterations required doubles! This can lead to hundreds of iterations.

Now this graph shows the distance going negative, but the way CA works, the distance cannot go negative. If we can somehow let the distance go negative, we can easily beat worst-case CA with simple bi-section.

How can we let the distance go negative? We use our old friend, the separating axis!

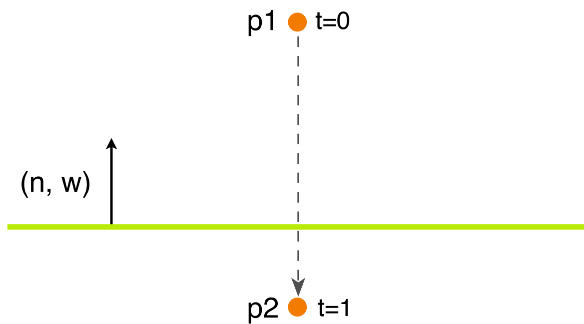
## Consider a point versus a plane




Now I'll start describing a bi-lateral advancement algorithm. And I'll start at the most basic level and build up to the full algorithm.

First consider a point with linear motion versus a plane.

## A linear equation yields the time of impact



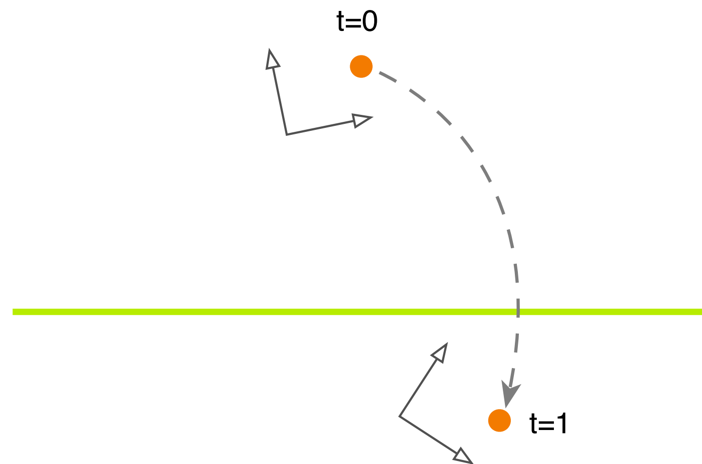

$$(p_1 + \alpha(p_2 - p_1)) \cdot n = w$$

$$\alpha = \frac{w - p_1 \cdot n}{(p_2 - p_1) \cdot n}$$

This is just a linear equation and we can find the root by putting the parametric line formula into the plane equation. Then we can solve for the time of impact alpha.

I would treat the case of a zero denominator the same as no impact.

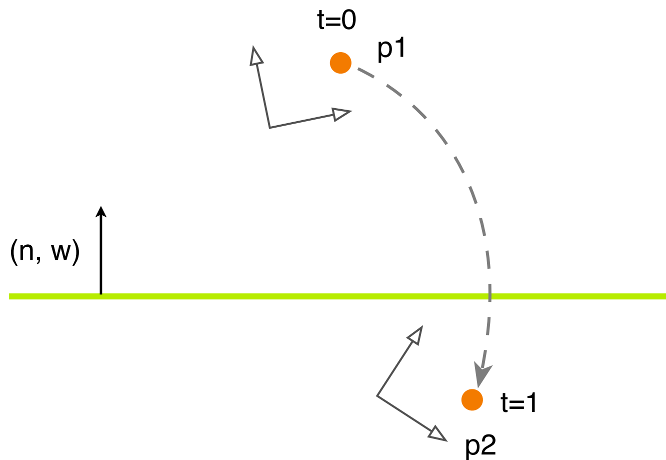
## Now put the point on a moving frame




Things get a bit more complicated when the point is attached to a rotating frame. For now, we don't really care about the nature of the frames motion. The only restriction is that it be smooth and be guided by a single parameter (time). We also assume the point is fixed in the moving frame (like a rigid body).

This is nice because we don't have to worry about the accuracy of the angular velocity integration. Anyhow, we now want to compute the time of impact for this point.

## Project the point motion onto the plane normal

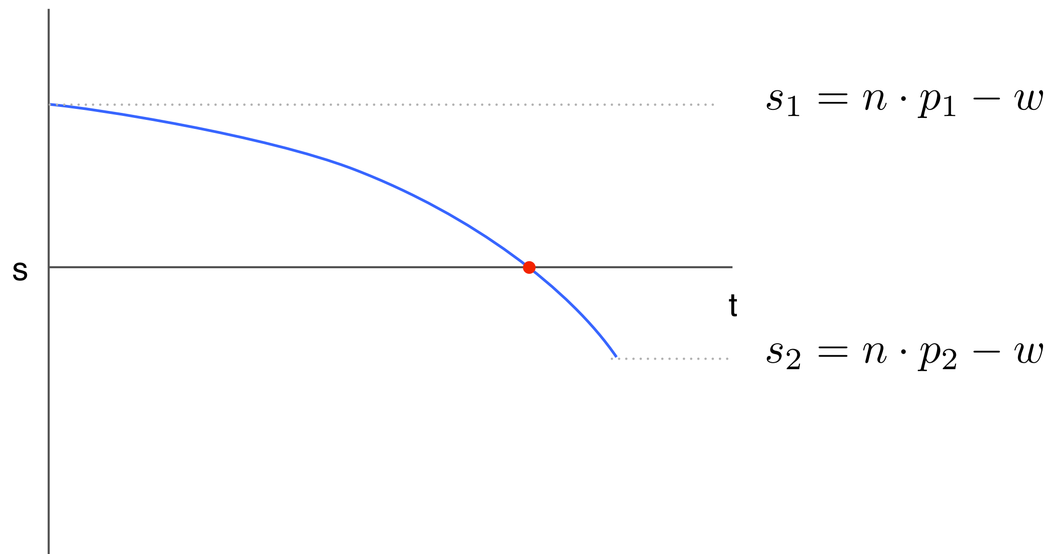



$$s(t) = n \cdot p(t) - w$$

We can compute the signed distance of the point by projecting it onto the plane normal. This returns us to our simple one dimensional function of time. And the function can be negative or positive!

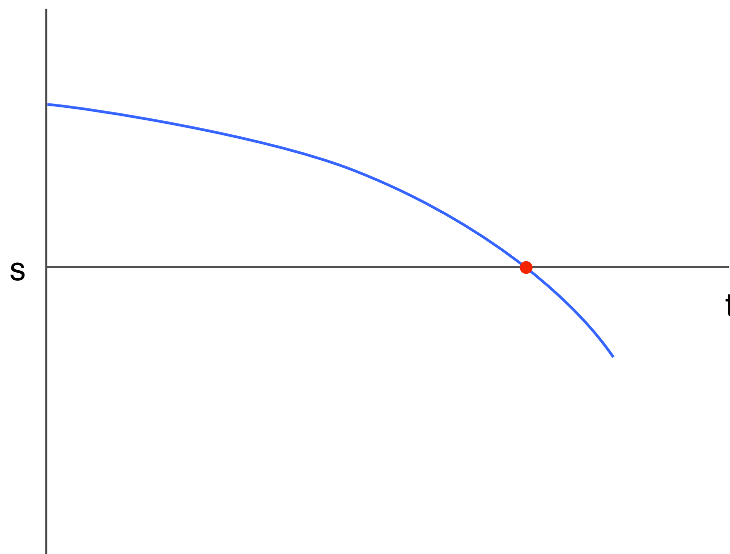
So we are now using the plane normal as a separating axis, like I promised.

## Plot the separation versus time




Now we can plot the separation versus time. Note that the separation begins positive and goes negative. If we have reasonable formulas for integrating linear and angular velocity, we will get a nice smooth plot like this.

## Bracketing is the key to finding the root




$$s_1 \geq 0$$
$$s_2 \leq 0$$

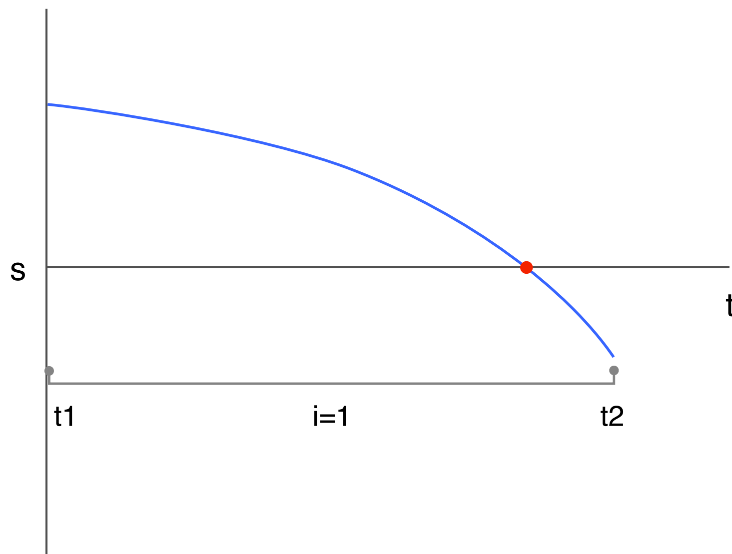
$$s(\alpha) = 0$$

$$\alpha \in [0, 1]$$

If we know that s1 is positive and s2 is negative then we are guaranteed that there is a root between 0 and 1. This means we can compute the time of impact safely using traditional root finding techniques.

We could try to compute an analytic solution and I welcome you to give that a shot. But I think you will see later that this quickly can become intractable, especially when you are dealing with two moving bodies in 3D.

# Root finding via bisection

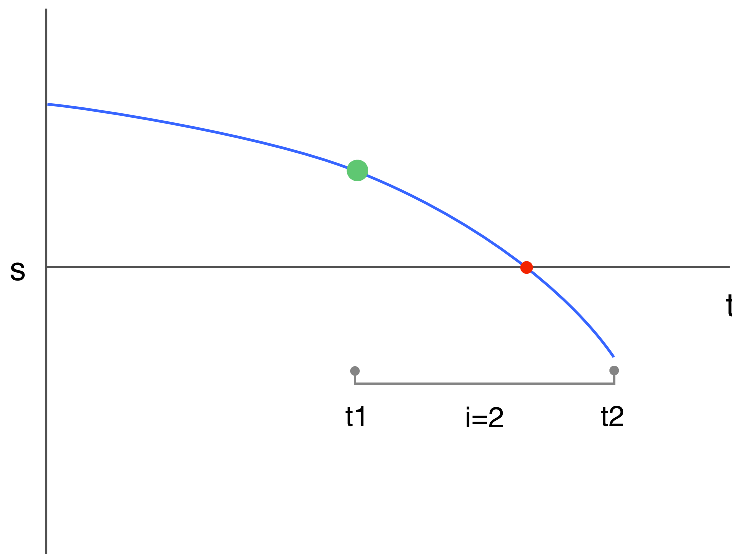



| i | t1 | t2 |
|---|---|---|
| 1 | 0 | 1 |
| | | |
| | | |
| | | |

Bisection is a simple but incredibly robust method that works when you have a bracketed root. It is not terribly fast, but it is reliable and it makes a guaranteed amount progress each iteration.

We start with t1=0 and t2=1. And we know that $s(t=0)>0$ and $s(t=1)<0$.

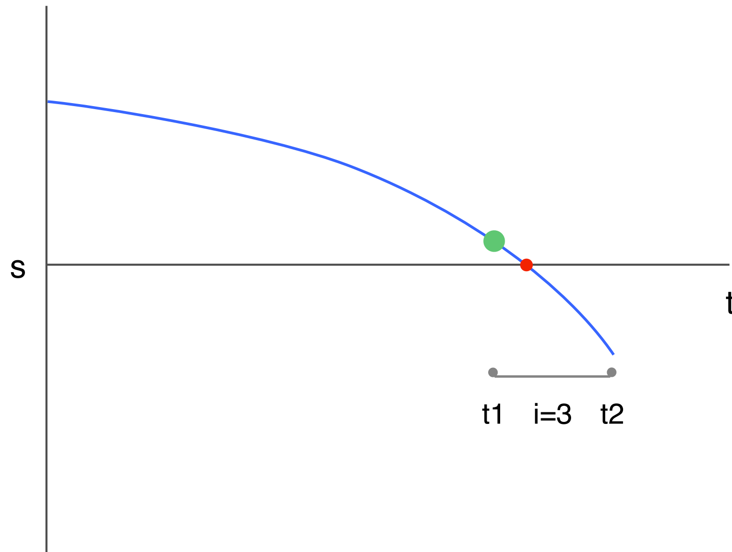
## Root finding via bisection




| i | t1 | t2 |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 0.5 | 1 |
| | | |
| | | |

We sample the mid-point, and find that s(t=0.5) > 0. So we forget about s(t=0) and create a smaller bracket between t=0.5 and t=1.

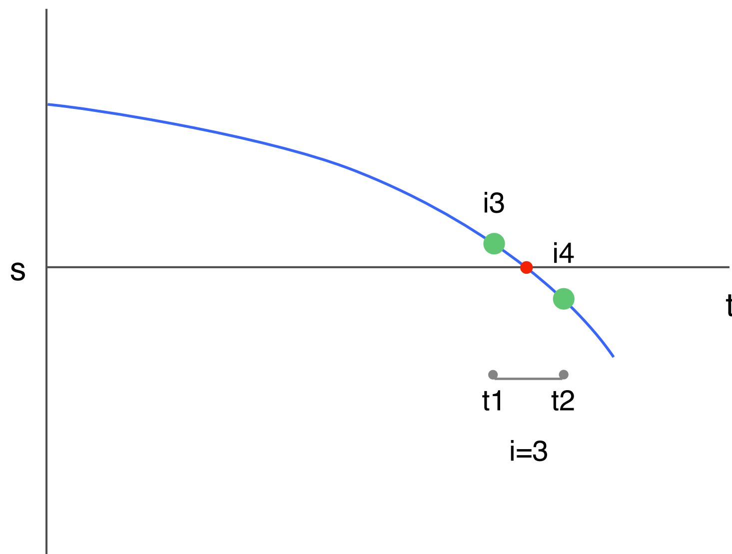
## Root finding via bisection




| i | t1 | t2 |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 0.5 | 1 |
| 3 | 0.75 | 1 |
| | | |

We continue by sampling at t=0.75 (half-way between 0.5 and 1.0).

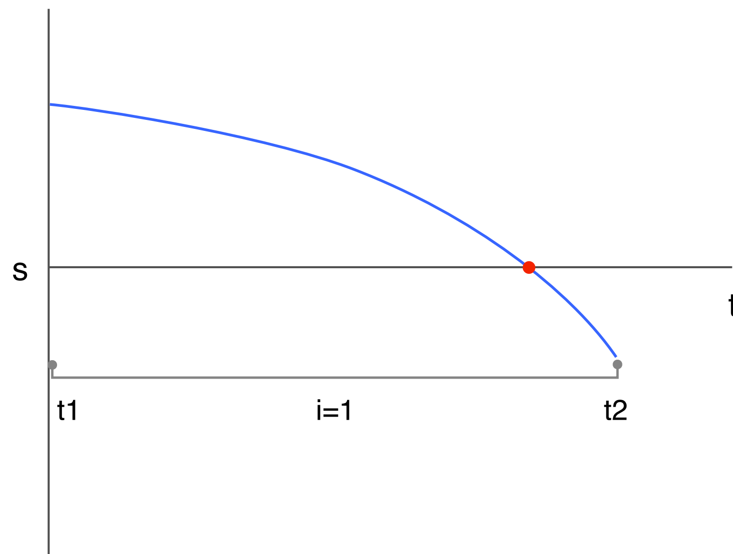
## Root finding via bisection




| i | t1 | t2 |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 0.5 | 1 |
| 3 | 0.75 | 1 |
| 4 | 0.75 | 0.875 |

Finally we sample at 0.875 = (0.75 + 1) / 2. We can continue iterating like this until the absolute separation is less than some tolerance and take this as our time of impact.

## Root finding via the false position method

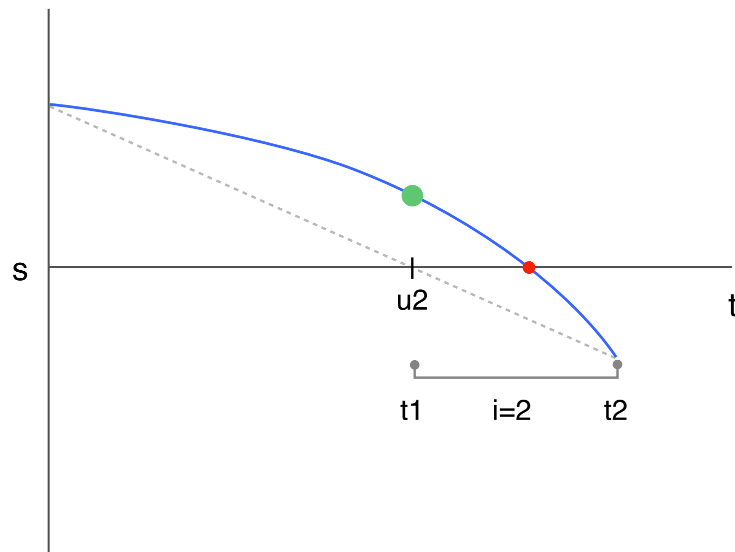



| i | t1 | t2 |
|---|---|---|
| 1 | 0 | 1 |
| | | |
| | | |
| | | |

We can get faster convergence in many cases by using the false position method.

The false position method starts out the same way as bisection. We have a bracket with t1=0 and t2=0.

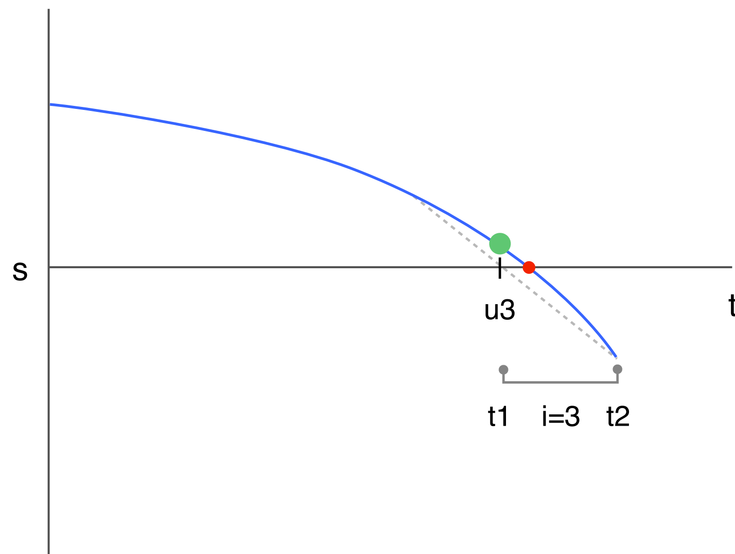
## Iterating the false position method




| i | t1 | t2 |
|---|---|---|
| 1 | 0 | 1 |
| 2 | u2 | 1 |
| | | |
| | | |

Next we draw a line between (t1,s1) and (t2,s2). We then compute the t-intercept ui and compute s(ui). We then adopt a new bracket depending on the sign of s(ti).

## Iterating the false position method




| i | t1 | t2 |
|---|---|---|
| 1 | 0 | 1 |
| 2 | u2 | 1 |
| 3 | u3 | 1 |
| | | |

The false method proceeds starting from the current bracket. You can see that the false method is moving quickly towards the root. However, for this curve the false position method never moves t2.

## Mixed method for root finding

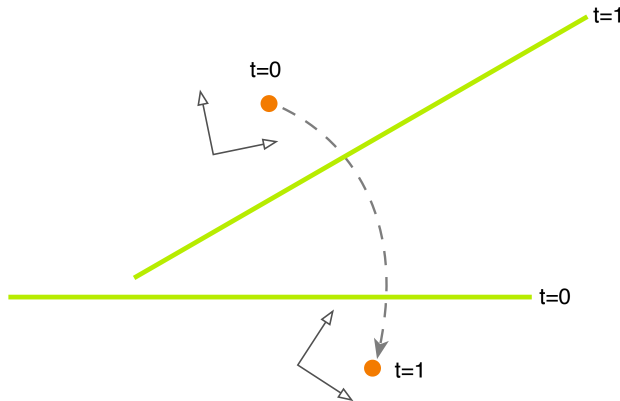
```
t1 = 0
t2 = 1
for i = 1 to max_iter
  if (i & 0)
    bisect
  else
    false_position
  end

  if abs(si) < tol
    break
  end
end
```

Root finders can be scary, especially when the description reads “this method converges if the initial guess is close enough to the root.” So I have taken a very conservative route with my root finder. The root must always be bracketed and each iteration must reduce the bracket size.

The false position method works when the function is close to linear in the region of the root and bisection can handle everything you throw at it. Therefore, I decided to combine the two methods.

There are many root finding algorithms that would do well on this problem, so feel free to experiment. You might find something faster.

## Now let the plane move

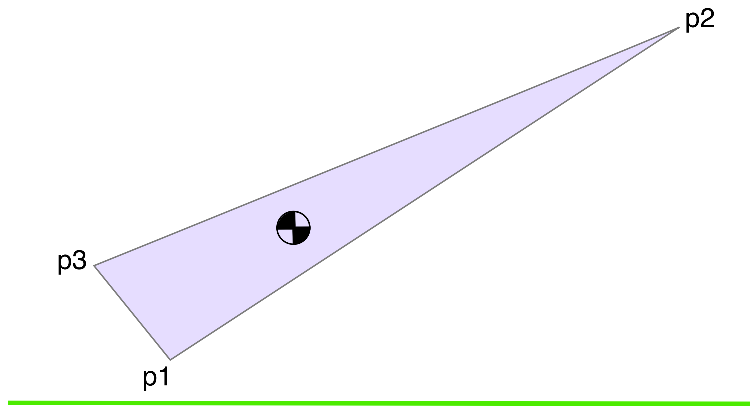



$$s(t) = n(t) \cdot p(t) - w(t)$$

Now if we let the plane move we have a normal and offset that are a function of time. Nevertheless, we can still apply our root finder to this problem. The bracketing and convergence are largely the same.

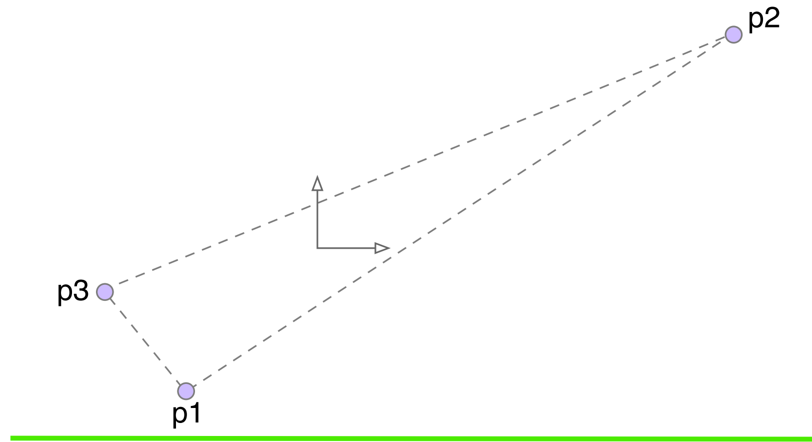
Note that we don’t need to know the formula for the normal and offset explicitly. We just need a way to integrate the linear and angular velocity of the plane and update its transform.

## Consider a polygon versus a plane




Ok, so we have handled point versus plane. How about polygon versus plane?

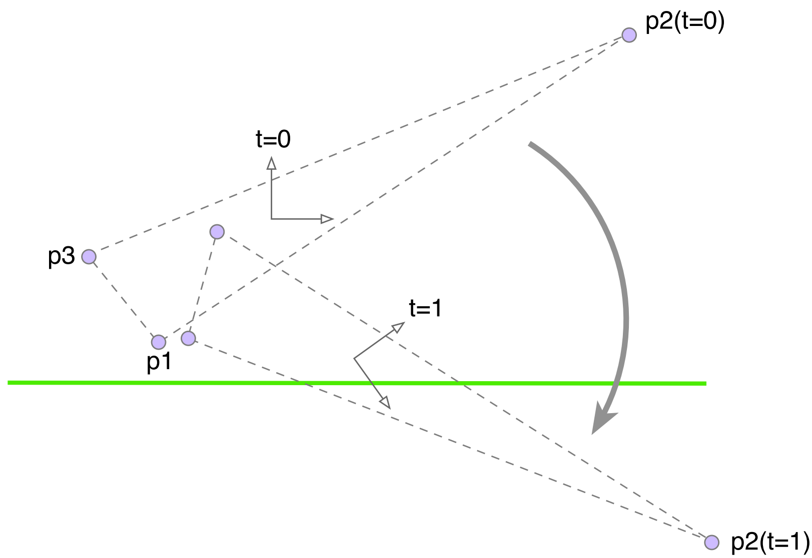
## Consider the polygon as a point cloud




I'll now consider the polygon as just some points on a moving frame. The plane can be moving. It does not matter as we already saw.

Each point has a separate time of impact with the plane. We could just compute the TOI for each point and take the minimum. But perhaps we can do better.

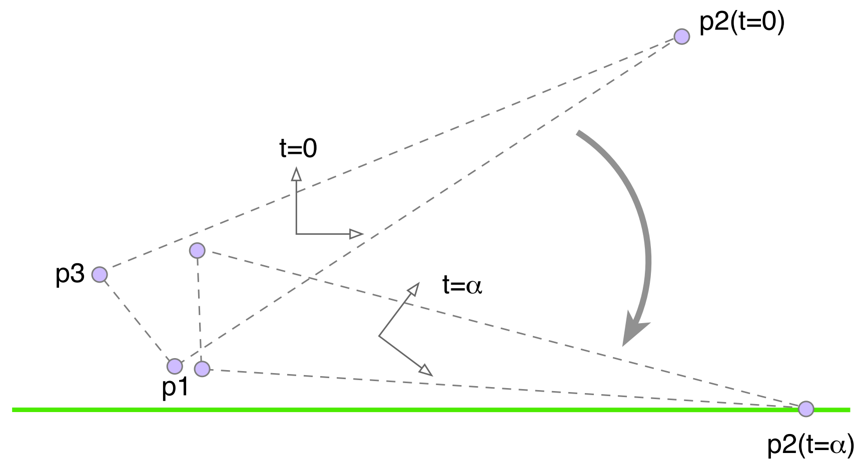
## Look for the deepest point




We can example the configuration at t=1 and find the deepest point. We look for the point furthest in the direction opposite the plane normal. So this is just a few dot products.

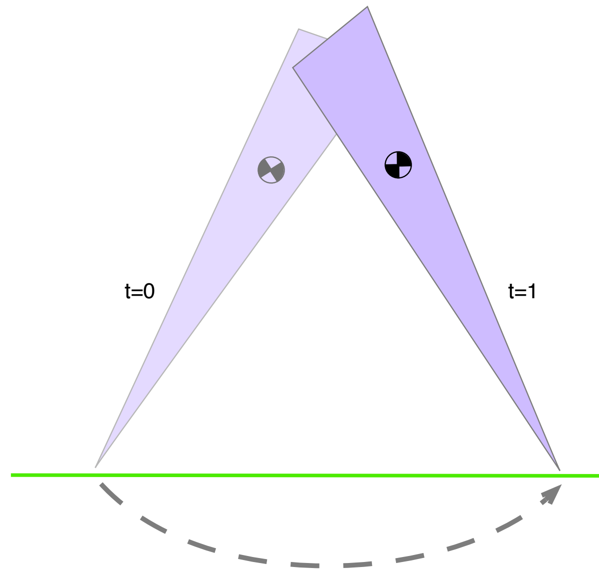
In this case we identify p2 as the deepest point.

## Use our root-finder to push the deepest point up to the plane




We use our point versus plane solver to find the time of impact alpha for p2. Then we move the polygon to t=alpha. Then we repeat the process of looking for the deepest point. It fhere are no points below the plane at t=alpha, then alpha is the time of impact for the entire polygon.

## Did we miss something?




Recall that the algorithm looks at the deepest point at t=1. If no point is penetrated, the algorithm returns no impact.

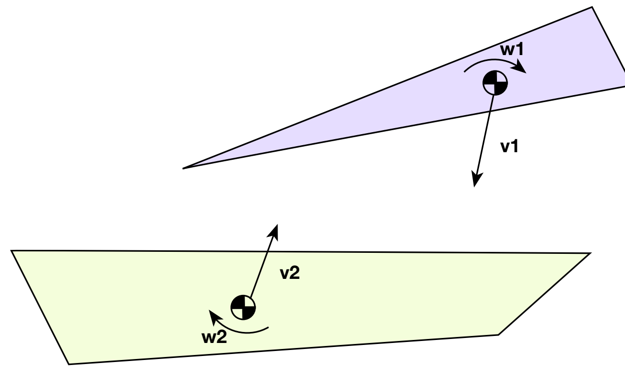
So the current algorithm can miss some rotational collisions. In particular, it can miss a collision where the polygon rotates in and out of the plane somewhere in the time step. I call these “glancing collisions”.

You could modify the root finder to handle this case, but then you have to examine every point and then you have a brute force method. This is not worth it in my opinion.

In my opinion these missed collisions are acceptable. In my games, I’m mainly trying to keep objects from falling outside the world. I don’t need a perfect simulation. In my experience this sort of missed “glancing collision” is never noticed in games. In the end you’ll have to be the judge.

The current algorithm is quite fast, so I think it is a fair trade-off.

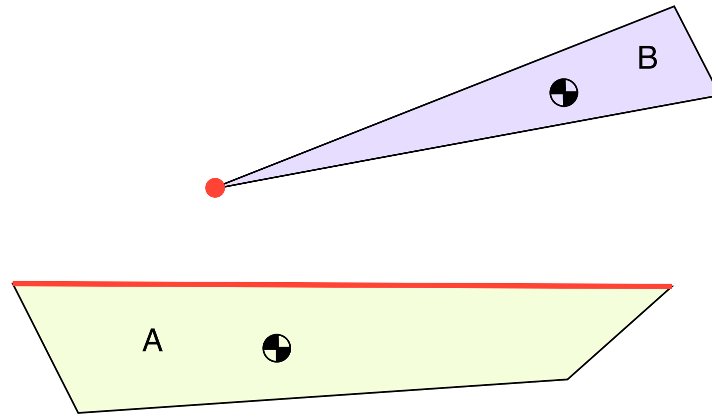
## Polygon versus polygon




We currently have an algorithm that can handle polygon versus plane. Both the plane and the polygon can move.

So how can we handle the polygon versus polygon case? Well, my approach is to reduce polygon versus polygon to polygon versus plane. We can do this by looking at the closest features.

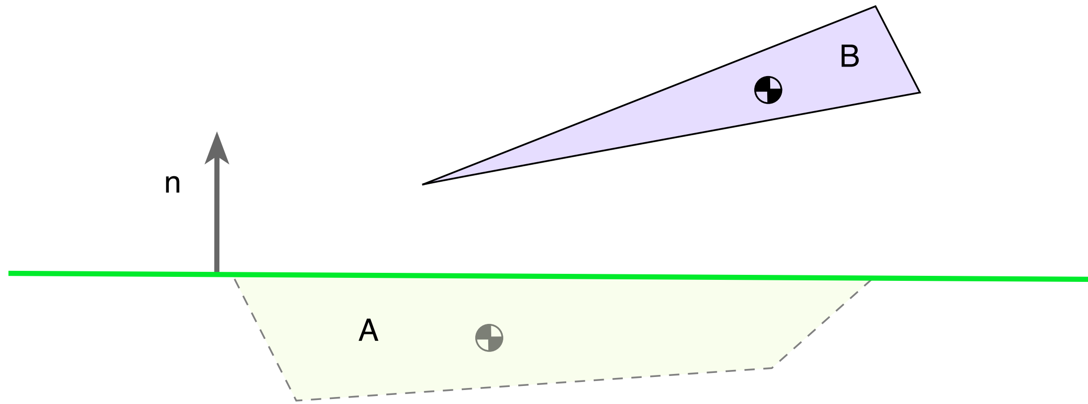
## Identify the closest features




We can get the closest feature using GJK. I already had GJK for conservative advancement, so I adapted it to provide the closest features. There may be other algorithms that a suitable for identifying the closest features, such as V-Clip or SAT.

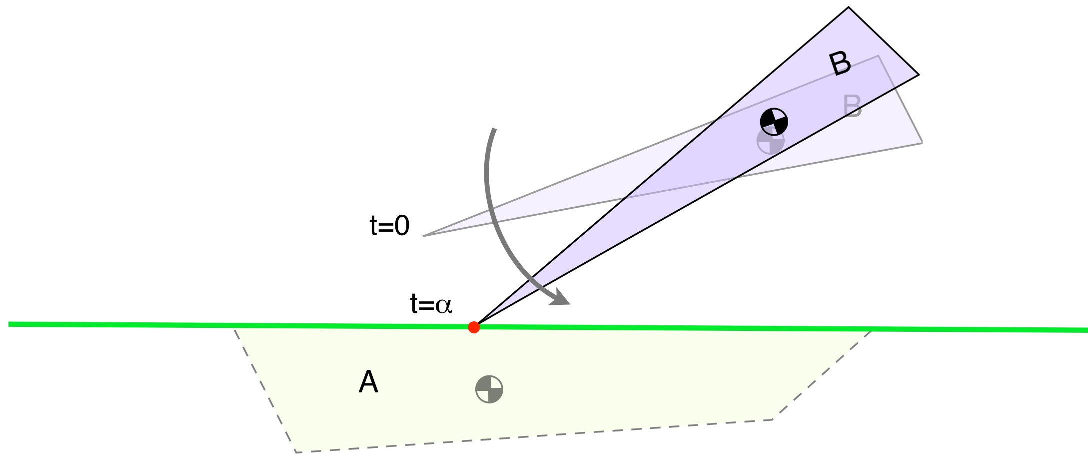
In this configuration the close feature is an edge on A and a vertex on B.

## Create a separating plane




We can create a separating plane by using the upper edge of polygon A as an infinite plane and apply our polygon versus plane algorithm. So we just treat polygon B as point cloud swept against a moving plane local to polygon A.

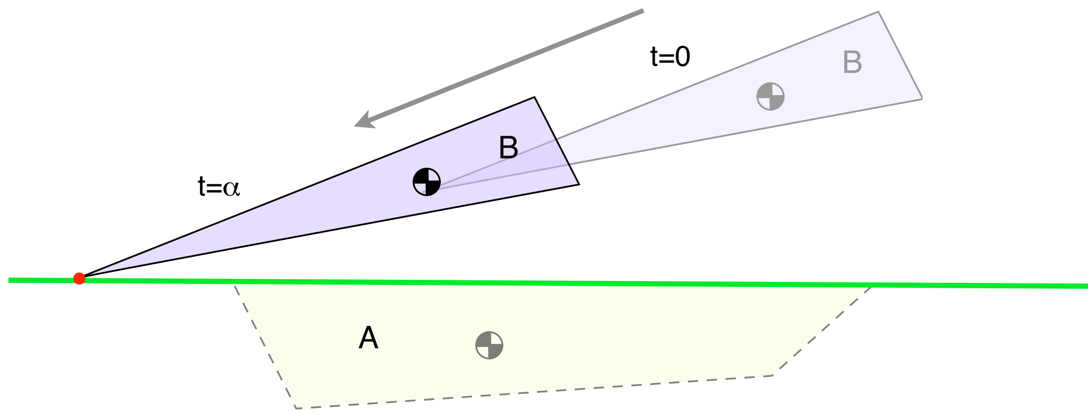
## Can this really work?




In this case polygon B rotates and hits the upper edge of polygon A. So this plane is a true witness to the collision. We have the time of impact and we are done.

Can you think of a different motion where this might fail? Keep in mind the paths are generated by constant linear and angular velocity.

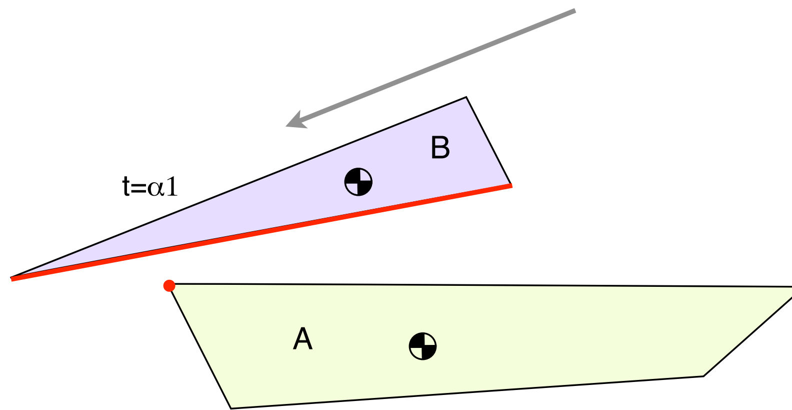
## What about this case?




In this case polygon B is moving under pure translation. Polygon B hits the plane, but it goes beyond the edge of polygon A. So the time of impact is invalid.

Can we repair this situation and find the true time of impact? I think we can!

## Re-compute the closest features



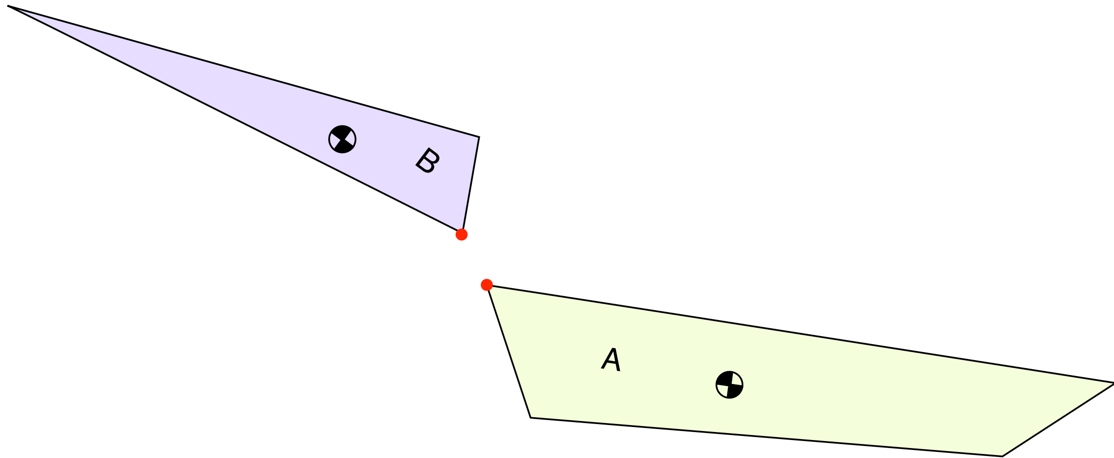

Let's reboot the algorithm from the current configuration. We re-compute the closest features and find that they have changed. Now we have a plane on polygon B.

Our polygon versus plane algorithm still works if the plane is moving and the polygon is stationary, so we can solve for the next tentative time of impact.

Therefore, can keep applying the polygon versus plane algorithm until the algorithm converges. Once the polygons are touching (within tolerance) we have the true time of impact.

We are almost done forming the bilateral advancement algorithm. There is one more situation to consider.

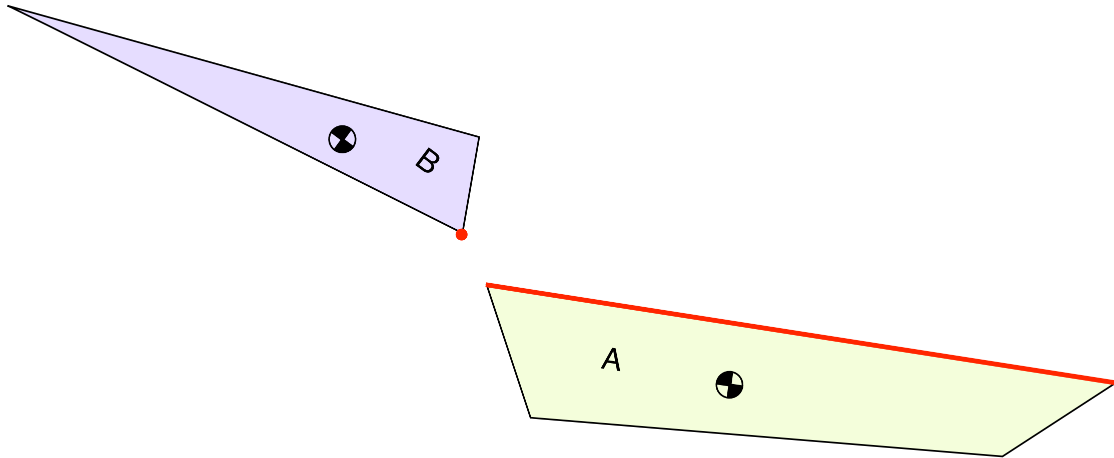
## The closest features might be two vertices




I need to cover one more case before I'm done. And that is the case where the close features are two vertices. In this case there is no obvious plane.

We don't actually need a plane, we just need a way to compute a signed separation value. So we actually have quite a few choices.

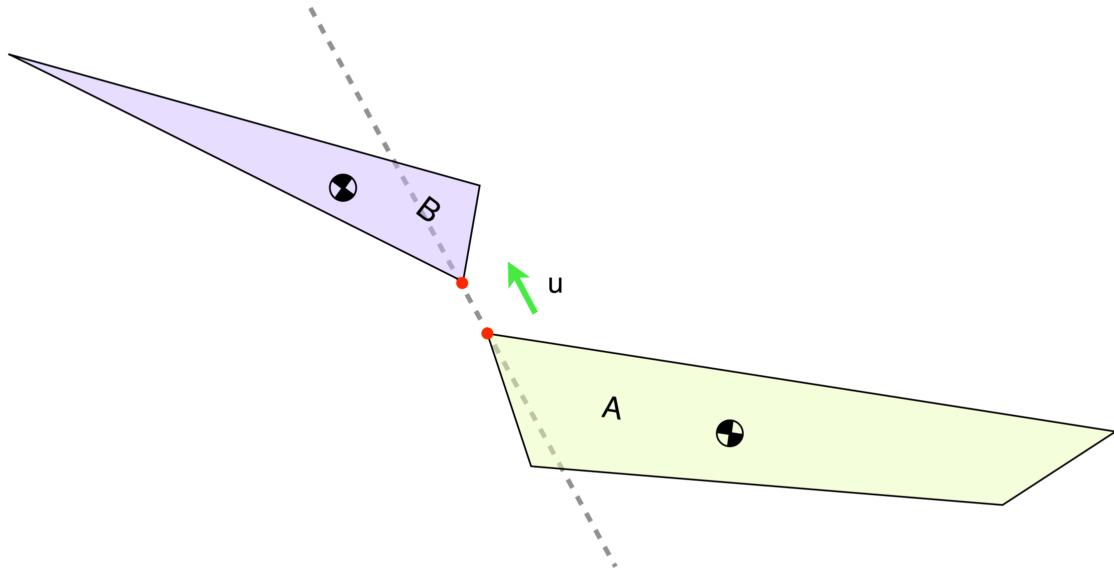
## Choice 1: choose an arbitrary edge




Here I chose a neighboring edge that defines a plane with positive separation. Notice that all points of B are above the upper edge of A, so this is a valid choice.

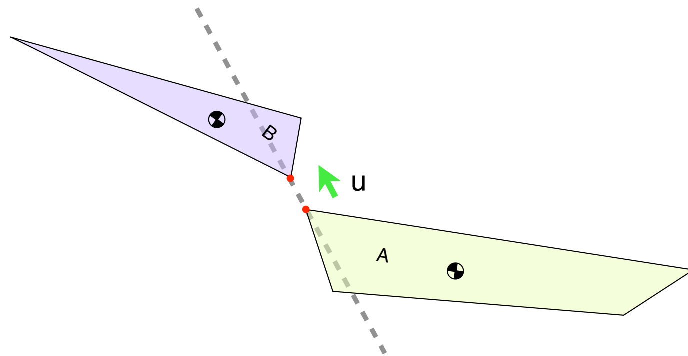
If you stare at the picture long enough, you might find other valid planes defined by polygon edges.

## Choice 2: a fixed projection axis




I handle the vertex-vertex case by forming a projection axis fixed in space. Then I can compute the separation along that axis in any configuration using the appropriate support points.

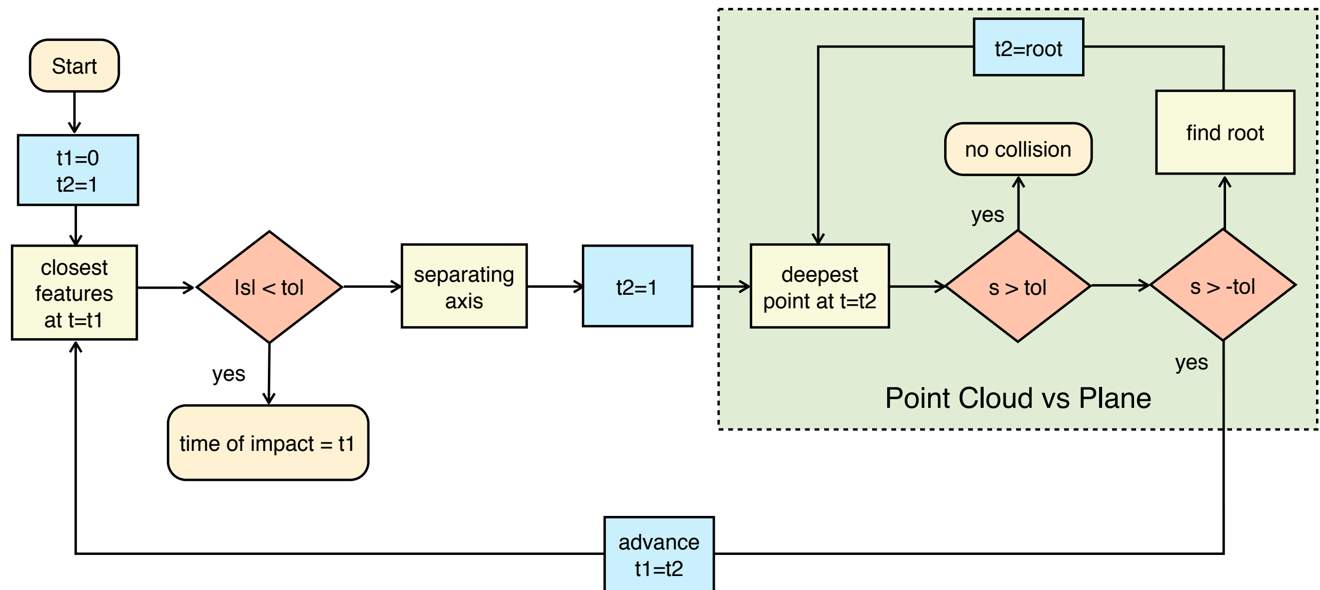
## Choice 2: a fixed projection axis




Steps:
1) Get the unit vector u pointing from A to B
2) Move the shapes to t=1
3) Find pA = polyA.support(u)
4) Find pB = polyB.support(-u)
5) s(t) = dot(u, pB(t) - pA(t))
6) Perform root finding on s(t) to get t1
7) Check for penetration at t1

Handling this case is fairly easy, but the details are important.

## The algorithm terminates when the closest feature is within tolerance




Here is a flowchart for the bilateral advancement algorithm. There are three key ingredients: finding the closest features, constructing a separating axis or plane, and then solving the point cloud versus plane problem.

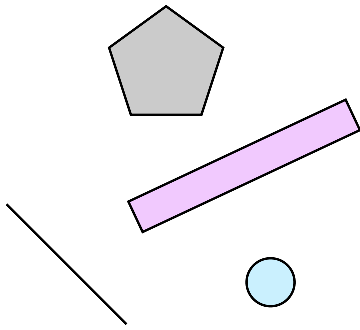
There are a few exit points:
1. If the shapes are initially overlapped, the algorithm returns a failure code (this is not shown in the chart)
2. If the closest features are within tolerance, the algorithm returns the time of impact
3. If the root finder determines the shapes are not overlapped at t2

There are other situations which can lead to an early exit, like the root finder running out of iterations. These cases may rarely occur, but you should handle them. In failure cases I just revert to discrete collision. In this way continuous collision becomes an optional feature.

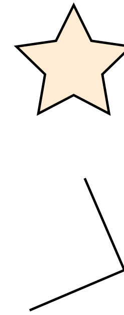
I encourage you to look at the Box2D code for details.

## What works?

Supported



Not Supported



You can use bilateral advancement to solve convex polygons, line segments, and points.

Like most high performance collision algorithms, it does not work on convex shapes. However, you can use bodies with multiple convex pieces.

You can also handle circles and capsules by adding a uniform radius. The radius comes into the computation when you compute the separation.

## What about 3D?

<screen shot of Domino continous test>

The bilateral advancement algorithm can be adapted to 3D. The key aspect you have to adapt is finding the closest features (GJK) and identifying suitable separation axes for root finding.

## References


- Brian Mirtich. “Impulse-based Dynamic Simulation of Rigid Body Systems”, PhD Thesis, University of Berkeley (1996).
- Gino van den Bergen, Ray Casting against General Convex Objects with Application to Continuous Collision Detection, 2004
- Stéphane Redon et al, Fast Continuous Collision Detection between Rigid Bodies, 2002
- Speculative Contacts: http://jitter-physics.com/wordpress/?p=109
- Vincent Robert, A Different Approach for Continuous Physics, GDC 2012
- David Baraff, An Introduction to Physically Based Modeling: Rigid Body Simulation II—Nonpenetration Constraints
- http://en.wikipedia.org/wiki/Brent%27s_method



box2d.org
@erin_catto