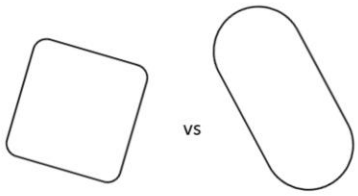
GJK and MPR Extensions

jodavis42@gmail.com

This lecture will be about ways to extend GJK and MPR for more complicated shapes tests, namely for ray-casting and for swept collision.

Convex shape collision detection

Can use GJK or MPR to test two arbitrary convex shapes

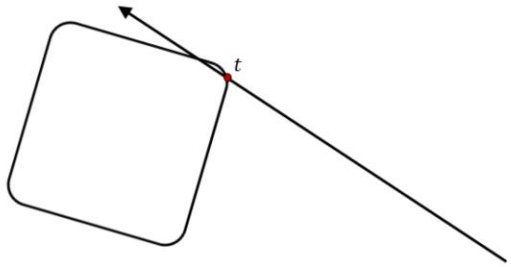


Two common operations missing!

Now that we have GJK and MPR under our belts we can test any two convex shapes for collision detection. There's still two tests that come up often that we haven't addressed yet. Both of these will require us to expand GJK and MPR a little bit for a full feature set.

Ray-Casting

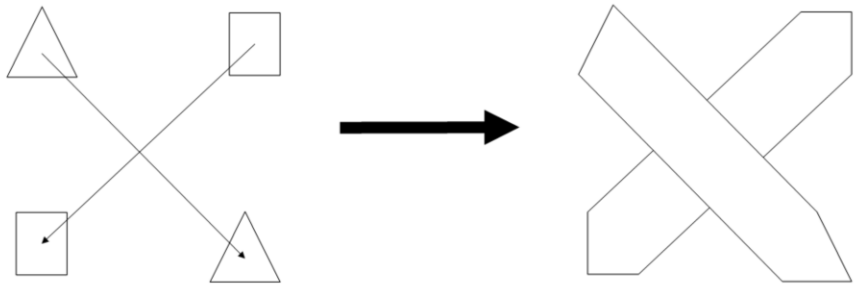
How do we determine if and when a ray hits a convex shape?



The first extra test we need to consider is ray-casting. With a polyhedral convex shape we can test the ray against each polygonal face of the mesh to determine the time of impact. This has two major problems: it's slow (comparatively) and it doesn't work for curved surfaces. Ideally we can re-use our Minkowski set operations with GJK and MPR to solve this.

Swept Collision (Linear Only)

How do we determine if two moving objects are colliding?

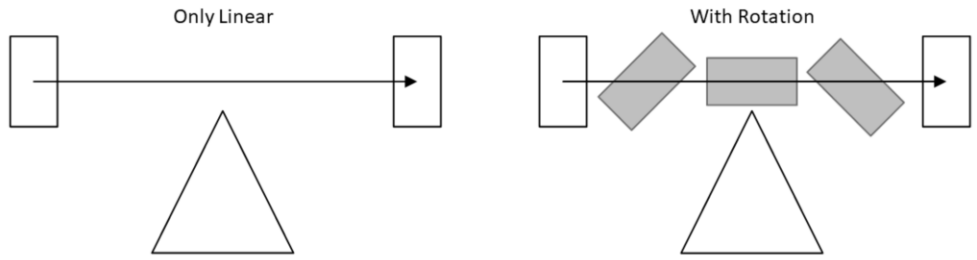


Sweep shapes out by displacement vector and test those

The second (and more nasty) problem is determining if two moving objects collide. This is a common problem in physics to deal with. If objects are moving fast enough then they can tunnel through each other and not detect collision (or even bad collision). The standard way to do this is to sweep out the shape by its displacement vector and test these shapes for collision. More details on this later!

Swept Collision (Rotation)

When rotation is involved detecting swept collision is even harder!



We'll ignore this for now…

The simplest solution to checking for swept collision involves only checking linear motion of objects and ignores rotation. For reasons we'll see later, adding rotation makes things a whole lot trickier. For now just know that we're only looking at linear translations.

Ray-Casting and Swept Collision

These two problems are very closely related

I'll jump back and forth between them

Easiest to look at MPR first, go to GJK later

MPR Linear Sweep

How do we represent a swept object?

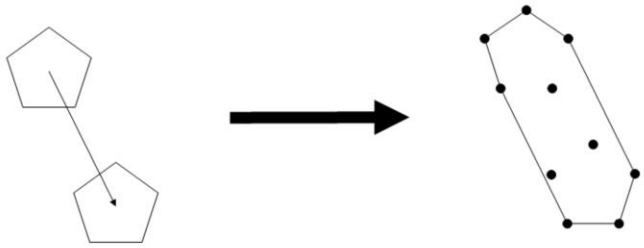
To start with we'll look at performing a linear sweep with MPR. It's much easier with MPR to perform linear sweeps and hence ray-casting.

The first question is how do we represent a swept object? We need to implement a support function for MPR to use.

MPR Linear Sweep

Solution 1: Just search the before and after points

MPR builds the convex hull automatically



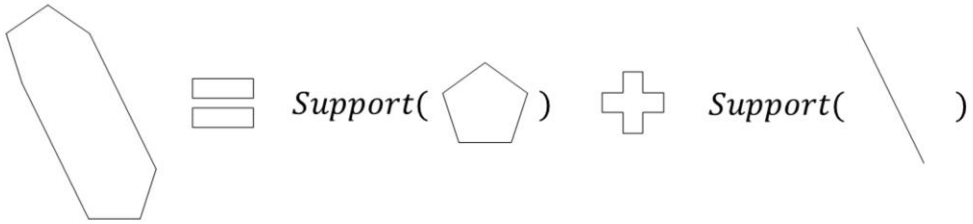
Slow and not necessary!

The first method to try and represent a linear sweep is to just represent a shape as its before and after points and search through those. As MPR (and GJK) implicitly search the convex hull of a set of points we don't technically have to remove any internal points to perform the search.

This however is really slow and wasteful as we'll spend twice as long performing the support search function. It's actually harder to implement this than it is the proper solutions. Just know that this is conceptually what we're after!

MPR Linear Sweep

Remember we can combine support shapes!

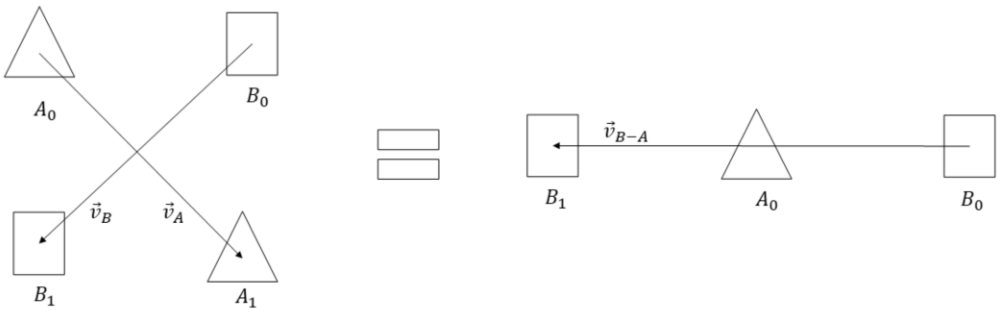


Equivalent to extending a shape by a displacement vector

The simplest way to sweep a shape out by a vector is to actual treat it as a combination of two support functions. If you add the shape's support function's result to the support function of the displacement vector (choosing between the displacement point and the origin) then you get a shape that is effectively swept out by the displacement vector.

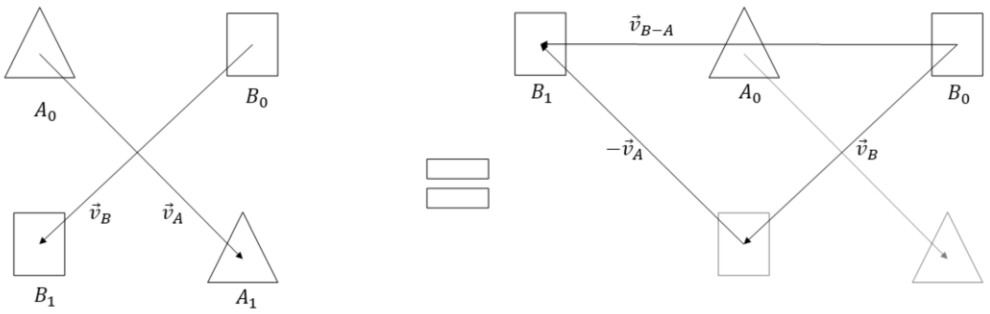
MPR Linear Sweep

Turn test of two dynamic objects into one static and one dynamic



More typically, we turn the problem of testing two dynamic objects into a test of one static and one dynamic object. This can be thought of as looking at the problem from A's frame (where everything is moving relative to him).

Linear Sweep Explanation



Just a simple picture explaining how we got our previous result. Basically we just move the displacement vector from one side of the equation to the other and sweep only 1 shape instead. Why we do this will become more important when we actually sweep for time of impact!

MPR Linear Sweep

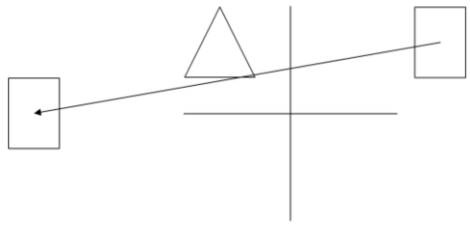
Linear sweep only detects if the objects overlap

How do we find Time of Impact (TOI)?

Just sweeping a shape with MPR won't give us any useful information other than if the objects collide (which in itself is useful and necessary for the next step). What we're really after is at what time in the frame the objects first start overlapping. This is known as a time of impact test or TOI for short. Luckily we can still solve our problem if we take a closer look at the CSO we're testing.

MPR Linear Sweep

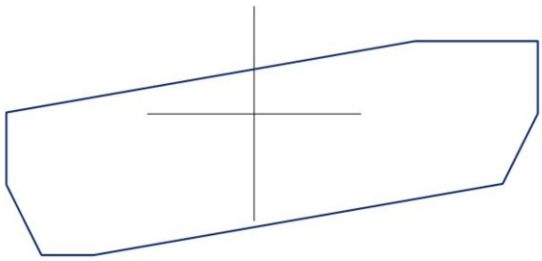
First look at a simple test



To see how we find the TOI with a linear sweep first look at this simple case.

MPR Linear Sweep

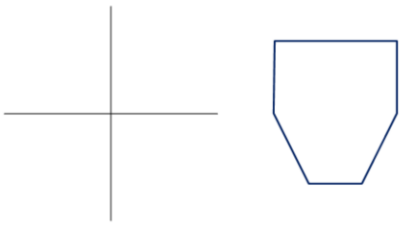
The test of $(B + \vec{v}_{B-A}) - A$ looks something like this



It's not too hard to see what the CSO of this shape would look like.

MPR Linear Sweep

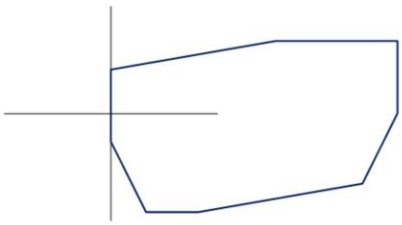
Let's rewind a bit and just look at $B - A$



Now if we take a step back and look at just the CSO of the two objects (without the sweep) we'll see something like this

MPR Linear Sweep

As we sweep by $\vec{v}$ at some point in time we'll reach this



We just need to find at what time this happens!

As we let this shape get swept out by the displacement vector $\vec{v}$ we'll reach some point where the CSO just barely touches the origin. If we can find at what t-value this happens we can solve for the TOI.

MPR Linear Sweep

How do we find t?

First note that MPR can find the minimum penetration distance in a given direction

Algorithm:

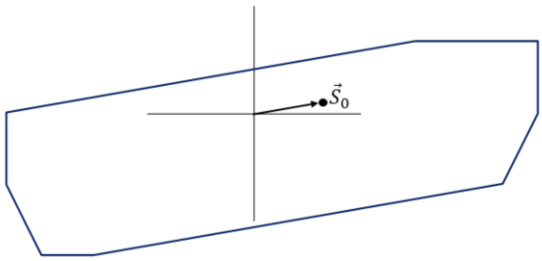
1. Run MPR and make sure the shapes collide
2. Initialize $\vec{S}_0 = \vec{O} - \hat{V}$
3. Run MPR algorithm until the portal reaches the surface of the CSO
4. Find the intersection of $\hat{V}$ with the portal face

The only problem is how do we find this? Well to get there I need to talk about another property of MPR real quick. In particular, MPR is able to find the minimum penetration distance in a given direction. That is, given a ray direction it can find how much overlap the shapes have in that direction.

The basic algorithm is listed above and I'll go through the basics step by step. Note that this part assumes that the shapes overlap so this is typically done as a secondary step after MPR returns true on the swept shapes.

MPR Linear Sweep

2. Initialize $\vec{S}_0 = \vec{O} - \hat{V}$

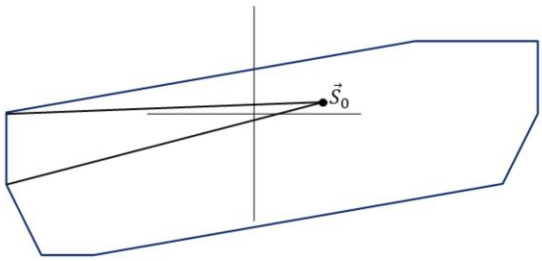


Note: This first assumes that you run MPR and the swept shape contains the origin

First we initialize the starting point of MPR to be a point behind the origin by the total displacement vector. That is, the starting simplex point looks right at the origin ($\vec{S}_0 + \vec{V} = \vec{0}$). Since we've already confirmed that the shapes overlap with each other it doesn't matter if this point ends up outside of the CSO.

MPR Linear Sweep

3. Run MPR algorithm until the portal reaches the surface of the CSO

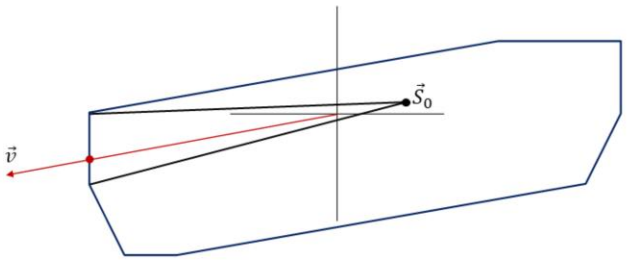


You should get a portal like this

Now we just run MPR with this starting simplex point as normal, making sure to continue the algorithm until MPR reaches the surface of the CSO.

MPR Linear Sweep

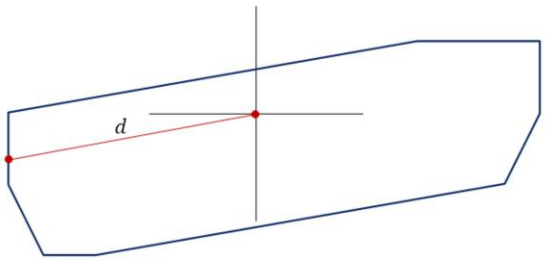
4. Find the intersection of $\hat{V}$ with the portal face



This portal is guaranteed to contain the displacement vector $\vec{v}$ and the origin and hence it will contain the point that is the intersection of $\vec{v}$ with the final portal face.

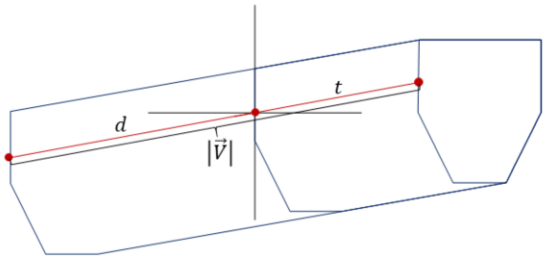
MPR Linear Sweep

This gives us the minimum penetration distance in the direction of $\vec{v}$



MPR Linear Sweep

Knowing all this we can solve: $t = 1 - \frac{d}{|\vec{V}|}$

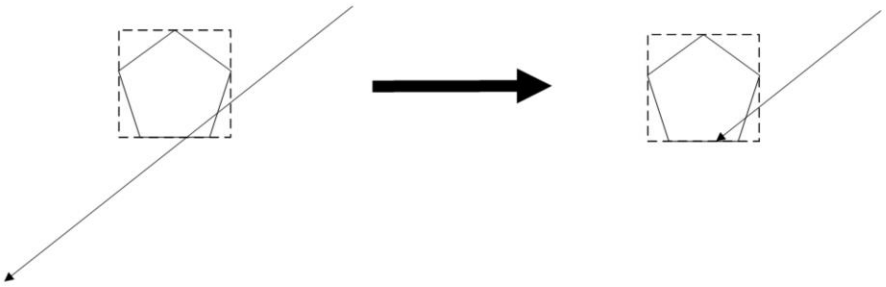


How do we use this distance? Well we have 2 knowns and 1 unknown. We know the total distance from the original CSO (not swept) to the point we just computed, that's simple the distance of the displacement vector $\vec{v}$. We also now know the minimum penetration distance d. Now it's simple to solve for t as it's just the remaining portion of the segment's length.

MPR Ray-Casting

Testing a shape against a ray isn't feasible (ray's are infinite)

Turn the ray into a segment by clipping to the object's Aabb



Knowing all this we can now also compute the TOI of a ray vs. a convex shape. First note though that we can't actually compute the TOI with a ray as sweeping a shape by an infinite line segment isn't feasible. Instead we need a line segment (or displacement vector) to cast against. We could pick some arbitrary distance to check for (like the far plane) but instead I recommend just clipping the ray's distance to the max TOI with the object's Aabb. In so doing we compute what the upper bound of the TOI is and if we first check the Aabb we can reject a lot of objects quickly.

MPR Line-Casting

Treat the line like a point with a displacement vector

Sweep the shape by the displacement vector and use linear sweep to compute the TOI

Now it's simple to test for ray vs. shape. Simply treat the segment as a point with a displacement vector. We don't want to actually sweep the point though as a segment is not numerically stable with MPR. Instead MPR is stable with points and shapes, so use the point's displacement vector to sweep out the convex shape. Now we can use the linear sweep to compute the TOI with the ray!

GJK

Ray-casting and linear sweeping are much harder than with MPR
GJK only gives useful information when shapes aren't colliding
EPA is too expensive!

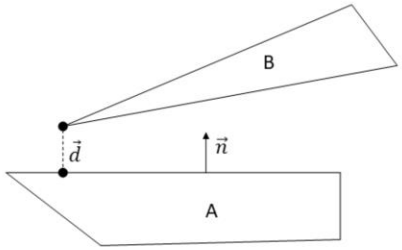
Solution: Conservative Advancement

Now we can look at ray-casting and linear sweeping with GJK. Unfortunately, both of these operations are significantly harder than with MPR. The main problem comes from GJK not being able to give any real information when the two shapes overlap. We could use EPA to find the minimum penetration distance in a given direction, but this is really expensive! In fact, we'd be better of just starting up MPR to determine this information than to use EPA.

The basic solution to these problems is what's known as Conservative Advancement. Do note though that this is likely to be more expensive than using MPR but we'll build up to somewhere interesting with it.

GJK- Conservative Advancement

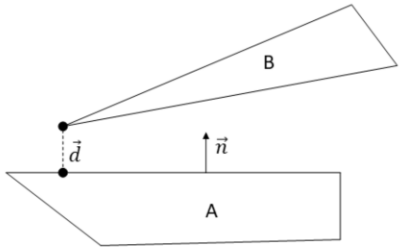
GJK gives closest features of two objects and a surface normal



Conservative Advancement is a small algorithm on-top of GJK. It first starts by assuming that your objects are not colliding (if they're already colliding then the TOI is 0). From a run of GJK we can get the closest features of our two shapes and the separation normal (and hence the separation distance) between the two objects. It's with this information that Conservative Advancement builds up an iterative algorithm.

GJK- Conservative Advancement

Make a conservative guess about when these features will collide



The earliest the objects can intersect is based upon the separating velocity:

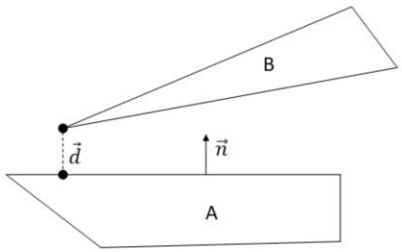
$$|d| \geq \Delta t[(\vec{v}_B - \vec{v}_A) \cdot \vec{n}]$$

At any given iteration Conservative Advancement makes a conservative guess (hence then name) about when the objects could possibly start colliding. A conservative guess means that if we advance forward some Δt that the objects can at most just start overlapping however they might not be overlapping at all (more later).

When only considering linear velocity the conservative guess is really simple. The earliest the shapes can start colliding is just a simple equation based upon the object's velocities projected onto the separation normal. That is, knowing the two object's velocities we can determine how long it takes them to move our separation distance.

GJK- Conservative Advancement

Make a conservative guess about when these features will collide



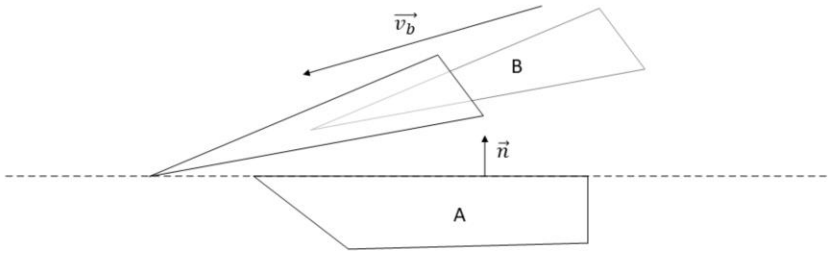
We can now just solve for Δt

$$\Delta t = \frac{|d|}{(\vec{v}_B - \vec{v}_A) \cdot \vec{n}}$$

So when we solve for the earliest Δt we get the above equation.

GJK- Conservative Advancement

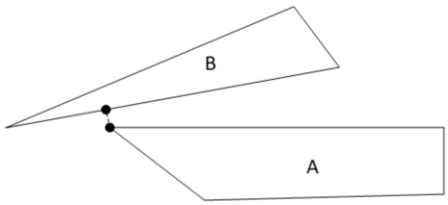
Not guaranteed to be an actual time of impact



So why did I say this was a conservative guess? Well if we give object B a velocity that was down and to the left in the previous example you'll see that we'd compute the time that the two points returned from GJK would reach zero distance along the separation normal. In this case shape B would move until the tip just hits the separation normal, but the shapes aren't actually colliding. If continue the cast we'd see that they would actually collide but we got back an earlier Δt. This is because we make a conservative guess.

GJK- Conservative Advancement

Conservative Advancement is an iterative algorithm!

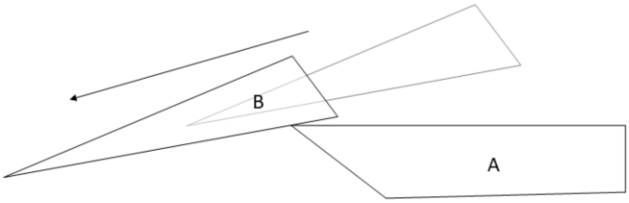


Find the new closest features with GJK and make a new conservative guess

So how does conservative advancement find the actual TOI? Remember it's an iterative algorithm so we have to repeat finding a new conservative guess. In this case GJK would compute a different pair of closest features and also a different separation normal. We can then solve to determine when these features would overlap on the separation axis.

GJK- Conservative Advancement

And advance again until reaching $d < \epsilon$ or $\Delta t > 1$



Conservative advancement continues until we either advance further than our allowed timestep (typically some other Δt, not to be confused with the one we're computing. This would be something like $\frac{1}{60}$) which means no intersection or until our separation distance drops below some threshold. If the separation distance is sufficiently small enough then we've computed the TOI of the objects.

GJK- Conservative Advancement

Pseudo-Code:

```
bool ConservativeAdvancement(Shape& shapeA, Shape& shapeB, float& toi, float epsilon)
{
  toi = 0.0;
  float distance;
  Vector3 normal;
  FindClosestFeatures(shapeA, shapeB, toi, normal, distance);

  // Run until we get close enough
  while(distance < epsilon)
  {
    // Reached end of frame and still no collision
    if(toi > 1)
      return false;

    float dt = distance / Math::Dot(normal, shapeB.Velocity - shapeA.Velocity);
    toi += dt;
    FindClosestFeatures(shapeA, shapeB, toi, normal, distance);
  }
  return true;
}
```

This is some basic pseudo-code for the Conservative Advancement algorithm. The algorithm continually finds the closest features and then computes the conservative dt for when the shapes could start colliding. This just continues in a loop until either we come close enough to be considered colliding or until we travel as far as we can for the frame.

GJK: Ray-Casting

Can test a ray now as a point against a swept shape

Now that we've defined how to perform a linear sweep using Conservative Advancement we can perform ray-casts. This is the exact same as with MPR where we first compute a segment from the ray and then treat that segment as a point with a velocity.

There's not really anything special here…

Swept rotating shapes

Much harder to implement!

Simplest implementation: Time-step sub-division

- Slow!
- Can still miss collisions (small or fast objects)

Not going to talk about MPR with this (I think you can only sub-step)

We'll now take a small look at sweeping rotating shapes (I'll save most of this for another lecture). First note that this is substantially harder to get working. In fact, this is not possible without some kind of an iterative algorithm (as far as I know).

The simplest method to implement this that works with any collision detection method is to perform sub-divide your time-step and test many in-between steps. This is obviously really slow and it can also miss collisions depending on the speed and size of your objects.

Unfortunately, I do not know any efficient method to improve this with MPR. The best idea I have is to wrap your swept object with a conservative shape (bounding sphere?) and then find the linear sweep's TOI to make a good first guess. You can then use that as a lower bound to try and sub-divide your time-step. As MPR doesn't give closest features there's no easy way to sub-divide like with conservative advancement.

GJK: Full Sweep

Can use Conservative Advancement to test for rotating objects as well

Need a fully conservative formula for

Can be really slow sometimes though

See Erin Catto's 2013 GDC presentation for full details

GJK can do much better than MPR with a full sweep. Conservative advancement can also be used with rotating objects as long as you properly update the conservative guess for Δt to account for rotation.

Unfortunately, Conservative Advancement can run into performance issues with some input configurations. Instead of going into the full Conservative Advancement algorithm, the performance issues, and the solution here I'll just defer that to another presentation. Erin Catto gave an excellent presentation at GDC 2013 which covers all of this as well as the Bilateral Advancement algorithm which solves this problem.

Questions?

References

- Brian Mirtich. "Impulse-based Dynamic Simulation of Rigid Body Systems", PhD Thesis, University of Berkeley (1996)
- Erin Catto, GDC 2013 "Continuous Collision". http://box2d.org/downloads/

Here's just two useful references. Erin's GDC presentation has a lot more useful links that you should check out!