GJK

jodavis42@gmail.com

GJK (Gilbert-Johnson-Keerthi)

Can be used to finds the closest points of arbitrary convex shapes

Technically finds closest point on convex hull to another point

GJK (Gilbert-Johnson-Keerthi) is an algorithm often used for collision detection that is very different than SAT. GJK, works for arbitrary convex polygons and scales much better than SAT does. However, GJK is actually an algorithm to find the closest features between a convex shape and a point. How this relates to collision detection we'll see later.

Also, this technique originated in robotics to find the closest distance between two objects (before they collided).

Carathéodory's theorem

If a convex hull in dimension d contains a point $\vec{p}$ then there is a subset of the convex hull that will contain $\vec{p}$ that consists of $d + 1$ or fewer points

There's a few important things to go over before getting to the GJK algorithm itself. The first one is Carathéodory's theorem. This theorem is at the heart of all algorithms like GJK.

This theorem basically states that if we have a 3d convex hull and can find a subset of that hull of 4 or less points that contains our query point then the hull contains the point. Likewise in 2d we only need up to 3 points. This theorem should make a bit of intuitive sense as we need a certain number of points to contain a "volume" in a given dimension.

This smallest sub-set of points is often referred to as a simplex. The simplex of 0d is a point, 1d is a line, 2d is a triangle, and 3d is a tetrahedron. GJK utilizes this by evolving a simplex up to a tetrahedron to try to contain the origin.

Closest Points

Need to compute closest point for:

- Point
- Line
- Triangle
- Tetrahedron

As Carathéodory's theorem states we need only up to a tetrahedron for 3d let’s start with each shape. In particular, let’s look at how to find the closest point to each simplex up to a tetrahedron.

Notation

$\vec{Q}$ is the query point
$\vec{S}_0, \vec{S}_1, \vec{S}_2, \vec{S}_3$ are simplex points
$\vec{P}$ is the closest point

Some real quick notation here to avoid confusion. First of all, we have the query point $\vec{Q}$, that is we are trying to find what point is closest to $\vec{Q}$ on the convex hull. Next we have to represent the points of the simplex. These are represented as $\vec{S}_i$. Finally we have the resultant point $\vec{P}$ that is the closest point on the convex shape to $\vec{Q}$.

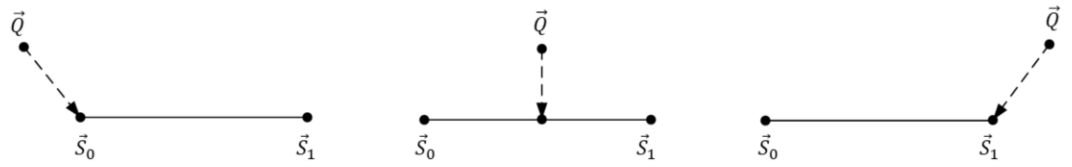
Closest Point to Point

Closest point to point is always $\vec{S}_0$

Hopefully this case is fairly straightforward. When our simplex is just one point, $\vec{S}_0$, the closest point to $\vec{Q}$ will always be $\vec{S}_0$.

Closest Point to Line

Three cases:

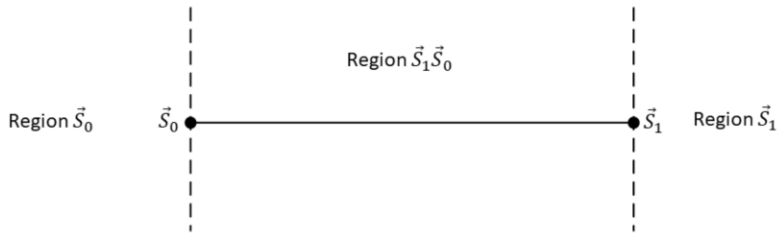


Closest point to line is pretty easy. We can break it up into 3 main cases. In the "simple" case, the closest point will be the projection of $\vec{Q}$ onto the line segment $\vec{S}_1\vec{S}_0$ (line from $\vec{S}_0$ to $\vec{S}_1$). This projection point can occasionally be outside of the line segment though. One method to deal with this would be to compute the t value of the point with respect to the line. If $t < 0$ then $\vec{S}_0$ is the closest, else if $t > 1$ then $\vec{S}_1$ is the closest, else the projection point is the closest.

While this is one method that does work instead we'll introduce a new concept (and then come back to the t-value).

Voronoi Regions

Divide space up by closest features

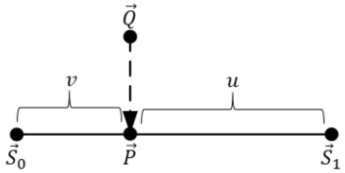


Instead we can look at the concept of voronoi regions. With a line segment we can have 2 kinds of features being the closest: a point and an edge. With this we can subdivide all space into 3 regions: The region where $\vec{S}_0$ is the closest feature, $\vec{S}_1$ is the closest feature, and the edge $\vec{S}_1\vec{S}_0$ is the feature.

Now how do we quickly identify which region any point $\vec{Q}$ is in?
As a quick side note, there's many different ways to compute the same thing. I'll only be showing one method that is very easy to implement. It's not too hard to make more efficient tests, but for simplicity sake I will not cover them.

Barycentric Coordinates Refresher

Barycentric coordinates are a weighted ratio on input points



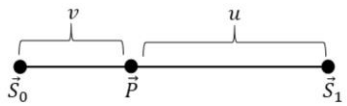
$$\vec{n} = \vec{S}_1 - \vec{S}_0$$
$$v = \frac{(\vec{Q} - \vec{S}_0) \cdot \vec{n}}{|\vec{n}|^2}$$
$$u = 1 - v$$
$$\vec{P} = u\vec{S}_0 + v\vec{S}_1$$

We're going to use barycentric coordinates to determine voronoi regions. As a quick refresher, remember that barycentric coordinates are a weighted average of the input points. Any point on the line S_0S_1 can be represented using u and v

The easiest method to determine voronoi regions is to use barycentric coordinates. Remember, barycentric coordinates are a weighted average of the points and they can span the shape. Pictured above is the barycentric coordinates of a given point. Remember, you don't have to project a query point onto the line to determine the barycentric coordinates, the point will be implicitly projected when computing the coordinates.

Line Voronoi Regions

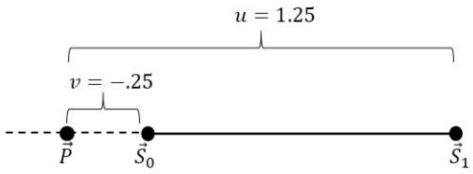
If $u > 0$ and $v > 0$ then $\vec{P}$ is in the line's voronoi region



Remember that a point is within the line segment if the barycentric coordinates are in the range of $0 < u, v < 1$. Note that we exclude the edge cases here as if $v = 0$ then $\vec{P} = \vec{S}_0$ which means that it is not in the line's voronoi region.

Line Voronoi Regions

What happens when $u \leq 0$ or $u \geq 1$?

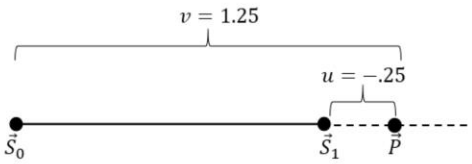


When $v \leq 0$ ($u \geq 1$) then $\vec{P}$ is in $\vec{S}_0$'s region

It's important to inspect the other cases of the barycentric coordinates, what happens when they're not between 0 and 1? Well if we first look at the case of $v < 0$ we'll see that $\vec{P}$ ends up to the left of the line, or more importantly in $\vec{S}_0$'s voronoi region.

Line Voronoi Regions

What happens when $u \leq 0$ or $u \geq 1$?



When $u \leq 0$ then $\vec{P}$ is in $\vec{S}_1$'s region

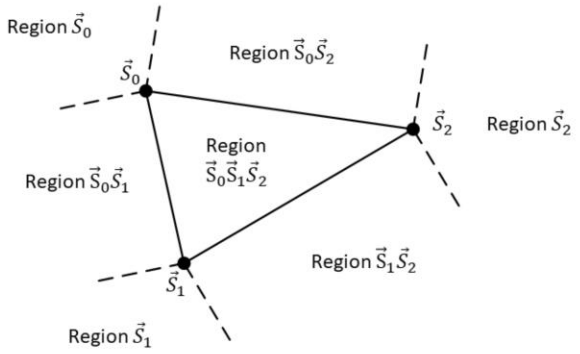
Similarly we can see what happens when u is negative. In this case $\vec{P}$ is to the right of the line segment, or in $\vec{S}_1$'s voronoi region.

Closest Point To Line

```
Vector3 FindClosestPoint(Vector3 q, Vector3 s0, Vector3 s1)
{
  float u,v;
  BarycentricCoordinates(q, s0, s1, u, v);
  if(v <= 0)
    return s0;
  else if(u <= 0)
    return s1;
  return u * s0 + v * s1;
}
```

Now we can write the final closest point to line function. Simply compute u and v from $\vec{Q}$ and then use them to determine which region we're within. If we're in the line's region then the closest point is the projection of $\vec{Q}$ onto the line segment which is simply the point computed from the barycentric coordinates.

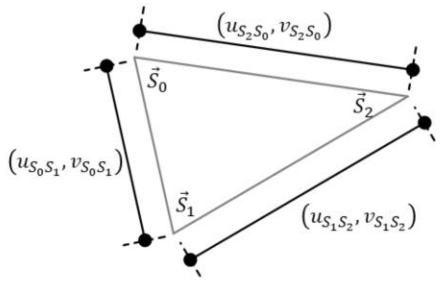
Closest Point to Triangle



Now we can continue on to a triangle. Triangle's have 7 voronoi regions: 3 point regions, 3 edge regions, and the triangle itself. We can once again use barycentric coordinates to determine the voronoi region (and the closest feature) although it's a bit more complicated than before.

Closest Point to Triangle

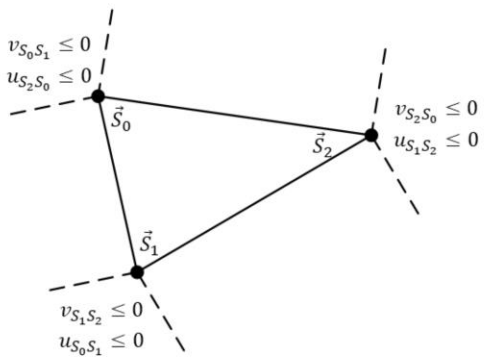
Can combine line coordinates to help determine regions



Let's look at what we have so far. To start with we have the barycentric coordinates of all 3 line segments, each of which have their own u and v coordinates. Combining these together will help us determine our voronoi regions.

Closest Point to Triangle – Point Regions

$\vec{Q}$ is in a point's region if it's on the "backside" of both outgoing edges

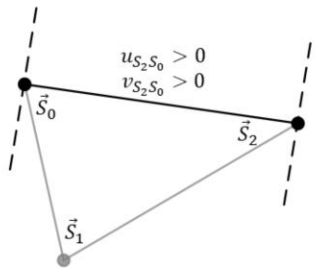


It's not too hard to see now that if $\vec{Q}$ is on the back side of all edges leaving $\vec{S}_0$ then it is in $\vec{S}_0$'s voronoi region. We can easily determine this by checking the barycentric coordinates of all 3 edges. Note: be careful about which coordinate you're checking, remember that if v is negative then $\vec{Q}$ is behind the first point.

Also note to be careful of the vector "winding order". The coordinate $v_{S_2S_0}$ is for the line from $\vec{S}_2$ to $\vec{S}_0$ (counter-clockwise ordering here). This is important in determining which coordinate you're checking. Realize that the actual winding order doesn't matter as long as you're consistent (if all lines are counter-clockwise or all are clockwise it doesn't matter, as long as you don't mix).

Closest Point to Triangle – Edge Regions

The line's coordinates are not sufficient to determine edge or triangle regions!

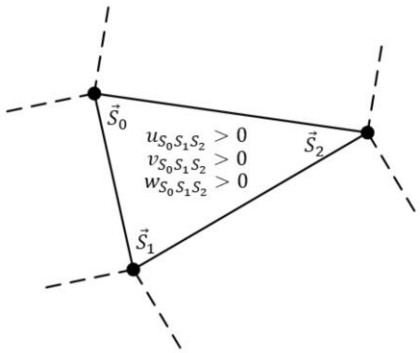


We need more information!

To determine if a point is within an edge's voronoi region is a bit trickier. In fact, with just the 3 line barycentric coordinates we can't determine if a point is within an edge's region. We also can't use this information to determine if a point's within the triangle's voronoi region. We need more information in order to determine this!

Closest Point to Triangle – Triangle Region

Use triangle's barycentric coordinates to check the triangle voronoi region

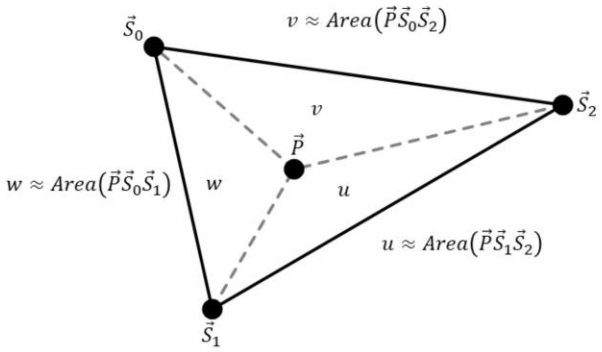


We'll come back to the line voronoi regions in just a bit

I'm going to temporarily ignore the edge's voronoi region and look first at the triangle's region. As mentioned before, we need more information than the edge barycentric coordinates. In particular we need the triangle's barycentric coordinates. Remember that if all of a triangle's barycentric coordinates are in the range of 0 to 1 then the point is within the triangle. This make it easy to test the triangle's voronoi region by just checking all of the triangle's barycentric coordinates.

Understanding Triangle Barycentric Coordinates

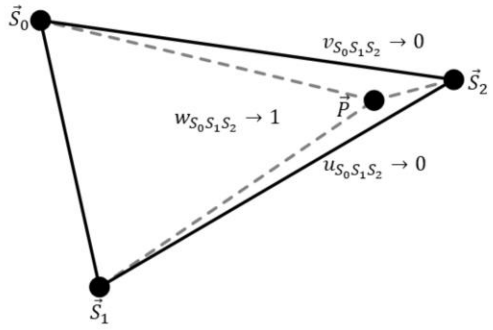
Barycentric coordinates are proportional to triangle areas



Now it's time to take a small detour and revisit barycentric coordinates of a triangle, in particular trying to build up a geometric understanding of what they mean. For our purposes here, the most useful way to think of barycentric coordinates is as fractional signed areas of the original triangle's area.

Understanding Triangle Barycentric Coordinates

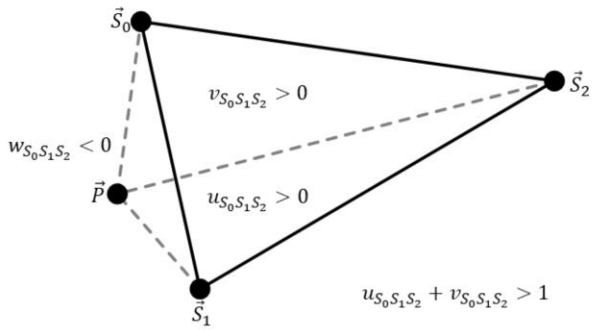
As $w \to 1$ note which triangles grow and shrink



We can build up a bit of an understanding by moving our test point around. In particular, notice as w approaches 1 and u and v approach 0 which triangles grow and shrink. This specific example isn't useful for us so far but helps to build up understanding.

Understanding Triangle Barycentric Coordinates

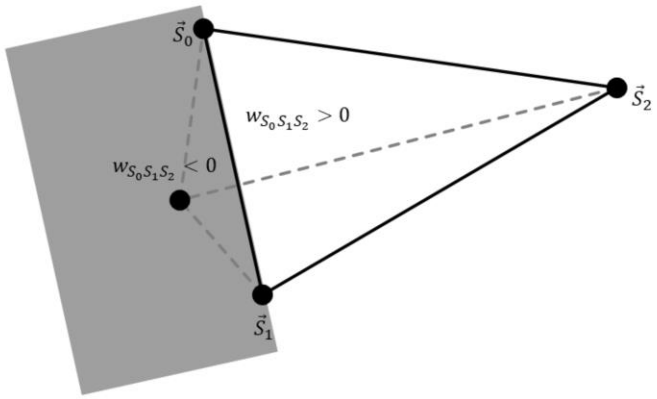
Negative barycentric coordinates are still valid



The more useful thing to understand is what happens when a coordinate goes negative. Remember any set of coordinates are valid as long as the sum adds up to 1, even if 1 (or 2) coordinates are negative.

Understanding Triangle Barycentric Coordinates

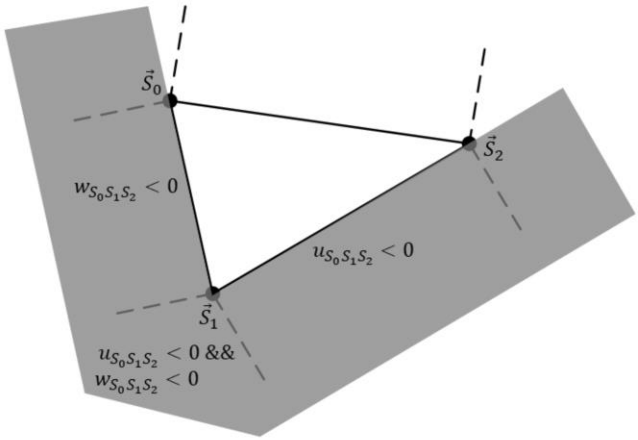
Can identify each half-space of $\vec{S}_0\vec{S}_1$ using w



Now it should be clear that the w barycentric coordinate can be used to determine which half-space we're on of the $\vec{S}_0\vec{S}_1$ edge. This can be applied more generally to all the edges; if a coordinate is negative then the point is outside of the half-space defined by the other two points. Alternatively, think of the barycentric coordinates as a vector: $\vec{b} = (u, v, w)$ with the subscript of b referring to index (i.e. $\vec{b}_1 = v$). Then you can say if $\vec{b}_i < 0$ then the point is outside the half-space of the edge $\vec{S}_{i+1}\vec{S}_{i+2}$.

Understanding Triangle Barycentric Coordinates

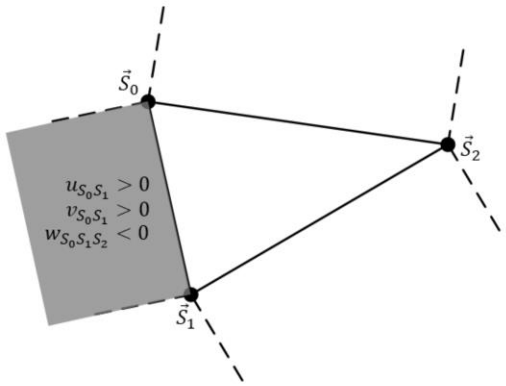
Identifying the edge half-space isn't enough



Unfortunately, identifying which half of the edge $\vec{Q}$ is on isn't enough to fully determine the voronoi region. Several voronoi regions intersect with this half-space.

Closest Point to Triangle – Edge Regions

Combine line and triangle coordinates to identify edge regions

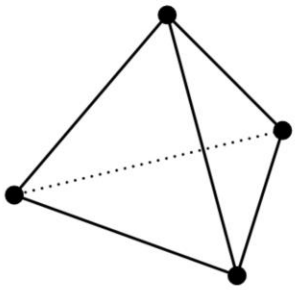


Just remember, barycentric coordinates != Voronoi Regions

Finally, we now have all the information needed to correctly identify the edge voronoi regions. To be in the edge's region the edge must be the closest feature, meaning it was in-between $\vec{S}_0$ and $\vec{S}_1$ but it also has to be on the outside of the edge, otherwise the triangle would be closer.

While it's important to remember that barycentric coordinates are not the same as the voronoi regions, by intelligently combining the coordinates we can determine the regions.

Closest Point to Tetrahedron



15 voronoi regions: 4 points, 6 edges, 4 faces, 1 tetrahedron

Finally we just have to deal with a tetrahedron's closest features. I'll do my best with pictures, but drawing 3d is tricky…

There are a total of 15 voronoi regions we have to check. If we can't determine that a point is in any of these voronoi regions then the point is inside the tetrahedra.

Closest Point to Tetrahedron – Point Regions

Combine all line coordinates just as before!

$\vec{S}_0$ region if:

$$v_{S_0S_1} < 0 \;\&\&\; v_{S_0S_2} < 0 \;\&\&\; v_{S_0S_3} < 0.$$

$\vec{S}_2$ region if:

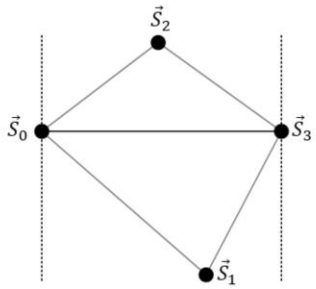
$$u_{S_0S_2} < 0 \;\&\&\; u_{S_1S_2} < 0 \;\&\&\; v_{S_2S_3} < 0.$$

Dealing with tetrahedron point regions is just a simple extension of what we already did for triangles. If we're on the "backside" of all outgoing edges then we're in the point's region. Just be careful to get your u and v's correct (remember the coordinate applies to other point, i.e. $u_{S_0S_2}$ means we're outside $\vec{S}_2$ while $v_{S_0S_2}$ means we're outside $\vec{S}_0$.

Closest Point to Tetrahedron – Edge Regions

Edge regions are the trickiest to visualize and understand

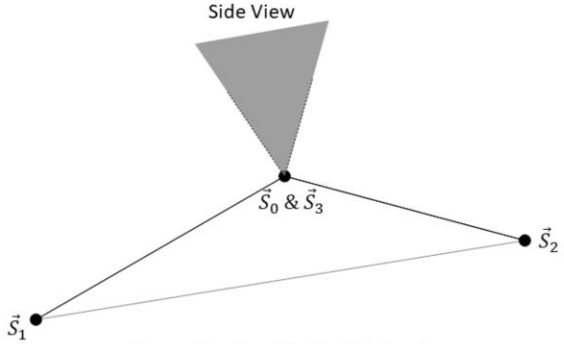
Top View



Within the "infinite line" :

$u_{S_0S_3} > 0$ && $v_{S_0S_3} > 0$

Side View



Also on the "back" of both triangles

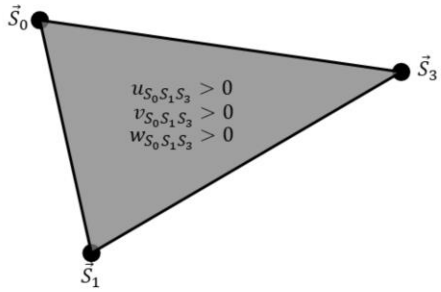
$v_{S_0S_1S_3} < 0$ && $v_{S_0S_2S_3} < 0$

Perhaps the hardest region to understand for tetrahedra is the edge regions. Obviously, to be in an edge's region $\vec{Q}$ has to between the two points on the edge. With a triangle the only other rule was we had to be on the "back side" of the edge defined by that triangle. This can be thought of as making sure that the edge was closer than the triangle (and hence all other edges). In a tetrahedron there are 2 triangles connected to an edge so logically we have to be on the "back" of both of those triangles. This is simply combining each triangle's regions.

Closest Point to Tetrahedron – Triangle Regions

Two rules for being inside a triangle's region:

1. The projection has to be within the triangle

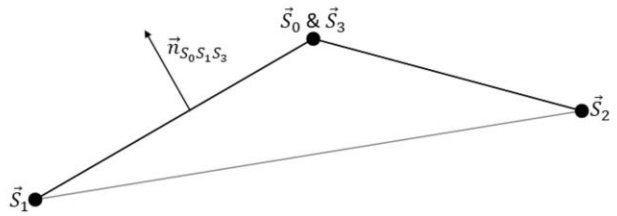


Triangle regions are also a little tricky to visualize. There's two key rules to understanding the triangle regions. The first rule is that the point has to project onto the triangle. This is simple to think about as if we project $\vec{Q}$ onto the plane of the triangle then the point has to be within that triangle's voronoi region, otherwise one of the other triangle features (point or line) is guaranteed to be closer.

Closest Point to Tetrahedron – Triangle Regions

Two rules for being inside a triangle's region:

2. $\vec{Q}$ must be on the positive side of the triangle



This can be computed 2 different ways

The other rule is similar to how we viewed triangle edges before, the point $\vec{Q}$ must be on the outside of the tetrahedron, or rather on the positive side of the triangle face (as long as the positive side points away from the tetrahedron's center). There's two different ways that are easy to compute this.

Triangle Region Side: Test 1

Use the triangle's normal, just be careful of its direction

```
Vector3 normal = Math::Cross(p1 - p0, p3 - p0);
// Make sure the normal points outwards
if(Math::Dot(normal, p2 - p0) > 0)
  normal *= -1;
return Math::Dot(q - p0);
```

The easiest way to think of testing this is to compute the triangle normal and make sure $\vec{Q}$ is on the positive side of the triangle. You do have to be careful that the triangle normal points the correct direction. This can be achieved with proper maintenance of winding order, or more easily by making sure it points away from the other point on the tetrahedron.

Triangle Region Side: Test 2

Use barycentric coordinates

$$\vec{P} = u\vec{A} + v\vec{B} + w\vec{C} + t\vec{D}$$

$$\vec{P} - \vec{D} = u(\vec{A} - \vec{D}) + v(\vec{B} - \vec{D}) + w(\vec{C} - \vec{D})$$

$$\overrightarrow{PD} = \begin{pmatrix} \overrightarrow{AD} & \overrightarrow{BD} & \overrightarrow{CD} \end{pmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix}$$

$$\begin{pmatrix} \overrightarrow{AD} & \overrightarrow{BD} & \overrightarrow{CD} \end{pmatrix}^{-1} \overrightarrow{PD} = \begin{pmatrix} u \\ v \\ w \end{pmatrix}$$

$\vec{Q}$ is on the positive side of triangle $S_0S_1S_3$ if $w < 0$

The other method that's more in line with what we've covered so far is to continue using barycentric coordinates. While I haven't covered how to compute barycentric coordinates of a tetrahedron before it's quite simple. We can do the same thing that we did before with a 2D triangle as we'll get 3 equations with 3 unknowns. We can then simply use the barycentric coordinates like before, remembering that if $\vec{Q}$ is outside a triangle then the coordinate we check is the one belonging to the point not in the triangle.

Closest Point to Tetrahedron – Tetrahedron Region

Either check if all of the tetrahedron's coordinates are positive

Or test this last (process of elimination)

Finally we have to determine if we're in the tetrahedron's regions. We can use the barycentric coordinates of a tetrahedron to do this but more efficiently we can just test this last. If we're not in a point's, edge's, or triangle's region then we must be in a tetrahedron's region.

GJK algorithm

Given an arbitrary simplex

Add a new point to the simplex that is guaranteed to contain points closer to $\vec{Q}$

Now we can start the actual GJK algorithm. The basic idea is:
Given a simplex, if $\vec{Q}$ is contained inside it then we are done. Otherwise update the simplex with new points guaranteed to be closer to $\vec{Q}$ than the old simplex.

GJK algorithm

1. Initialize the simplex (to one point for us) by searching in a random direction (difference of centers)
2. Determine which voronoi region $\vec{Q}$ is in and reduce to the smallest simplex
3. Compute $\vec{P}$ by projecting $\vec{Q}$ onto the new simplex
4. If $\vec{P}$ is equal to $\vec{Q}$ then terminate
5. Compute the new search direction $(\vec{Q} - \vec{P})$ and search for a new point
6. If the new point is no further than $\vec{P}$ in the search direction then terminate. The length of the vector $(\vec{Q} - \vec{P})$is the separation distance.
7. Add the new point to the simplex and go to 2.

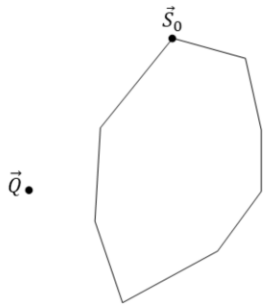
That basic idea translates into these explicit steps.

A few quick things to point out. When initializing the simplex we need to pick some direction to search initially. One method is to use some point inside the convex hull to then search in the direction $\vec{d} = \vec{Q} - \vec{C}$ but if we don't have the center than a random direction will suffice.

Also there are some steps that are hard to show in 2d as they'll never natively happen using a good initial search direction. The following example will pick a search direction to help show all of the remaining steps.

GJK algorithm – Example

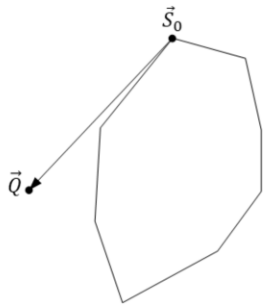
Initialize the simplex



We first initialize the simple to a random point. I choose $\vec{S}_0$ explicitly here so I can show all of the necessary steps.

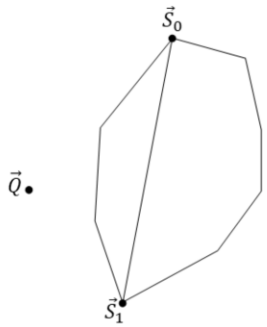
GJK algorithm – Example

Search in direction $\vec{Q} - \vec{S}_0$



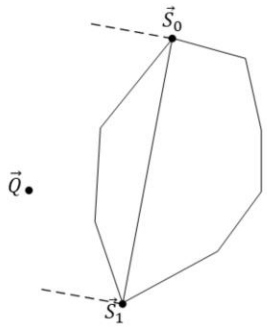
GJK algorithm – Example

Add point furthest in direction $\vec{d}$ to the simplex



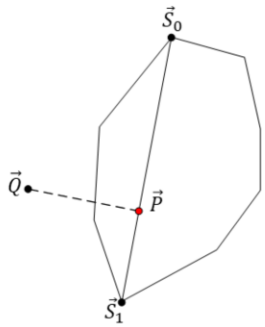
GJK algorithm – Example

Identify the voronoi region of $\vec{Q}$ and reduce (nothing happens)



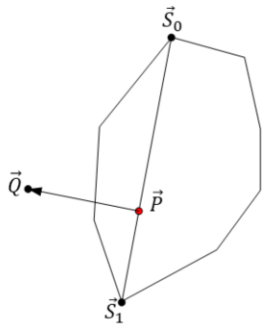
GJK algorithm – Example

Project $\vec{Q}$ onto the simplex



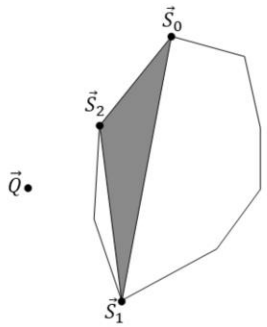
GJK algorithm – Example

Search in direction $\vec{Q} - \vec{P}$



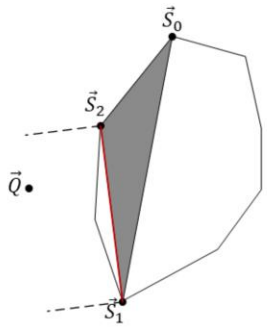
GJK algorithm – Example

Add point furthest in the direction $\vec{d}$ to the simplex



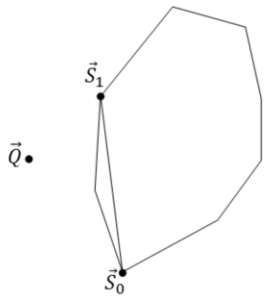
GJK algorithm – Example

Identify the voronoi region of $\vec{Q}$ as the line region of S_1S_2



GJK algorithm – Example

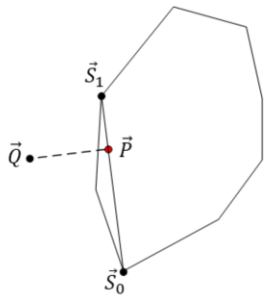
Reduce the simplex



When we reduced to the edge $\vec{S}_1\vec{S}_2$ we re-labeled the points. How exactly you do this isn't terribly important. I just slid everything down (so the lowest number stayed the lowest).

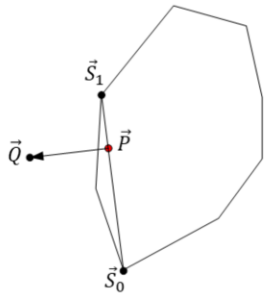
GJK algorithm – Example

Project $\vec{Q}$ onto the simplex



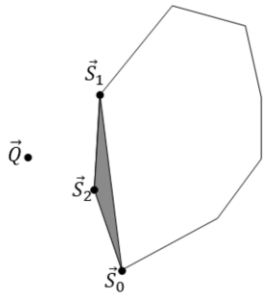
GJK algorithm – Example

Search in direction $\vec{Q} - \vec{P}$



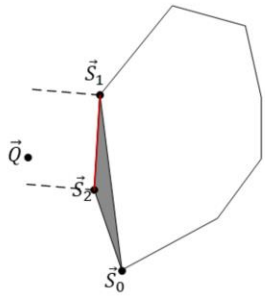
GJK algorithm – Example

Add point furthest in direction $\vec{d}$ to the simplex



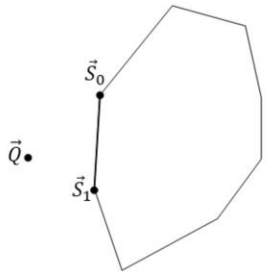
GJK algorithm – Example

Identify the voronoi region of $\vec{Q}$ as the line region of S_1S_2



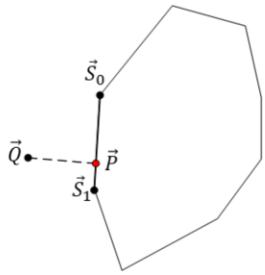
GJK algorithm – Example

Reduce the simplex



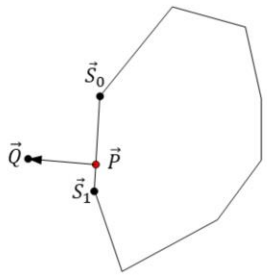
GJK algorithm – Example

Project $\vec{Q}$ onto the simplex



GJK algorithm – Example

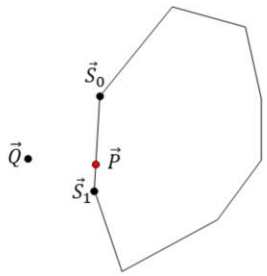
Search in direction $\vec{Q} - \vec{P}$



This search will return either $\vec{S}_0$ or $\vec{S}_1$, which one isn't important.

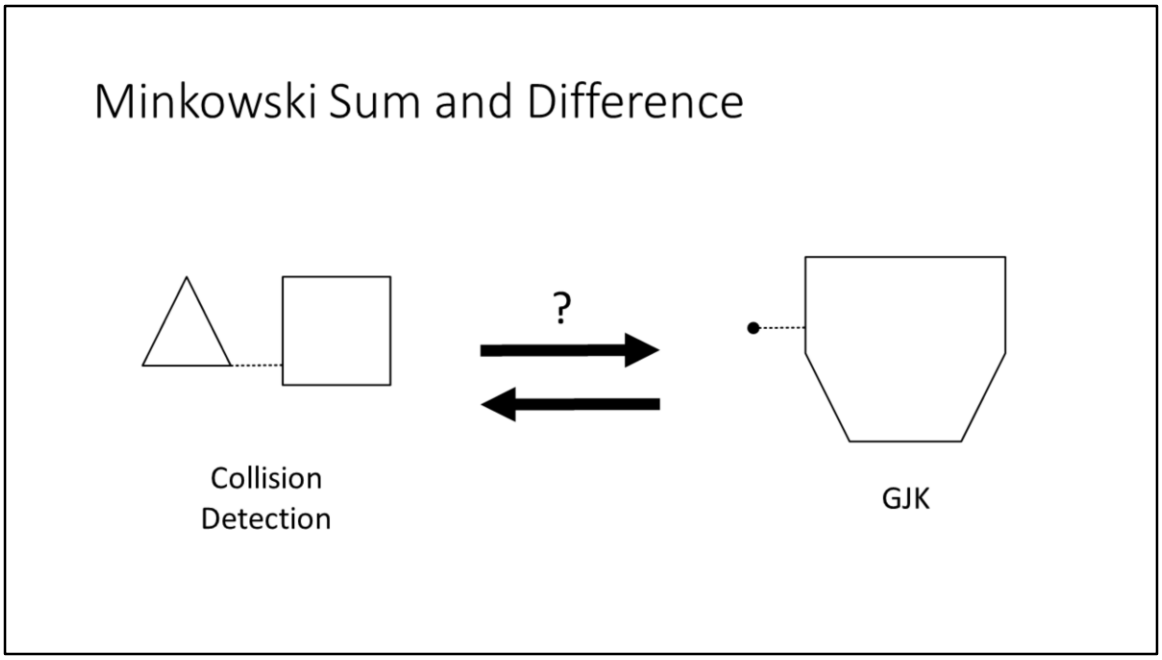
GJK algorithm – Example

New point is no further in the direction of $\vec{d}$ (it's either $\vec{S}_0$ or $\vec{S}_1$)
No further progress can be made.



$\vec{P}$ is the closest point

Let's pretend we got a new point here, $\vec{S}_2$, as our search point result. We can check $Dot\left(\overrightarrow{S_2} - \vec{P}, \vec{d}\right)$ here and see that no progress was made. This means GJK has terminated and we're done. Also $\vec{P}$ is the closest point on the convex hull to $\vec{Q}$.

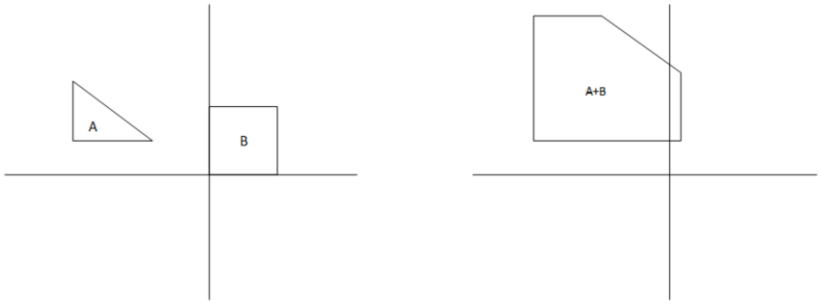


We now have the complete GJK algorithm and can find the closest point between a point and a convex shape. However our original goal was to find the closest point between two convex shapes. So we need a way to convert between these two problem sets.

Luckily there's a tool we can use to convert the problem of two convex shapes into a convex shape and a point. This is performed by the concepts of Minkowski sums and differences.

Minkowski Sum

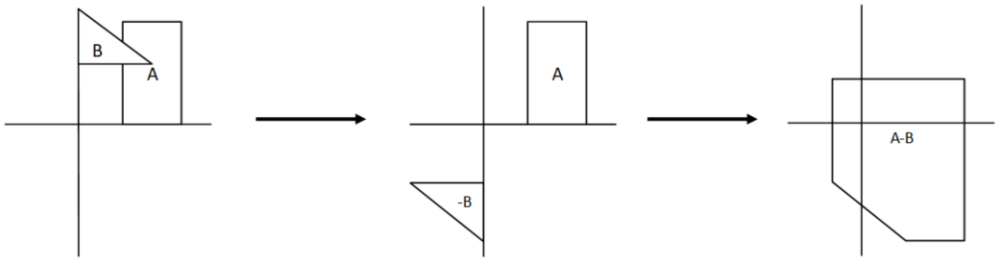
Minkowski Sum: $A + B = \{a + b : a \in A,\ b \in B\}$



The first identity is the Minkowski sum. The Minkowski sum of two objects is a shape that is composed of every point in A plus every point in B. This is easiest to visual with shapes at the origin (position matters) as the sum is the shape you get when sweeping B across the surface of A. More generally, we can just add every vertex position in A with every vertex position in B and the convex hull of this is our new shape.

Minkowski Difference

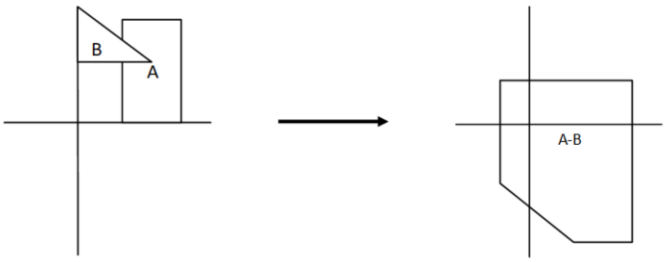
Minkowski Difference: $A - B = \{a - b : a \in A,\ b \in B\}$



The Minkowski difference is defined just as the subtraction of every point in B from every point in A. To visualize this first flip B across the origin then add it to A.

Minkowski Difference

If the two shapes overlap then the CSO will contain the origin

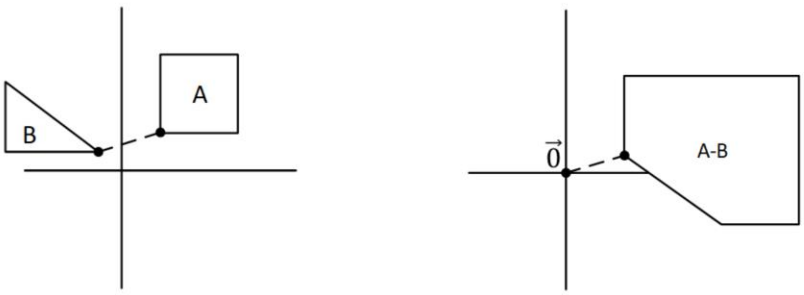


For GJK (and other collision detection algorithms) the Minkowski difference is what we care about. But why? What does this gain us?

If we take a quick step back and think about it, if we our objects overlap then there will be a point in A that is the same as a point in B. If we subtract these points then we'll get the origin. Hence, if the two shapes collide then the Minkowski difference shape (often called the CSO or Configuration Space Object) will contain the origin. Now we can use GJK to find if the origin is inside the CSO which equates to determining if the two shapes are overlapping.

Separation Distance

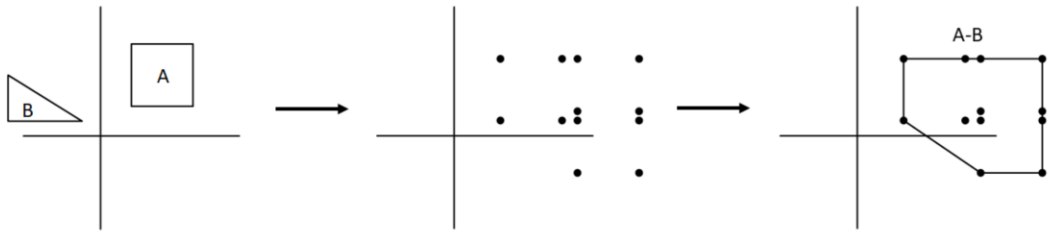
$$Distance(A, B) == Distance(\vec{0}, A - B)$$



Another important thing to realize about the Minkowski difference is that it preserves the minimum separation distance of the two objects. That is the distance between A and B is the same as the distance between the origin and (A-B).

Building the CSO

Naïve way: subtract and compute convex hull

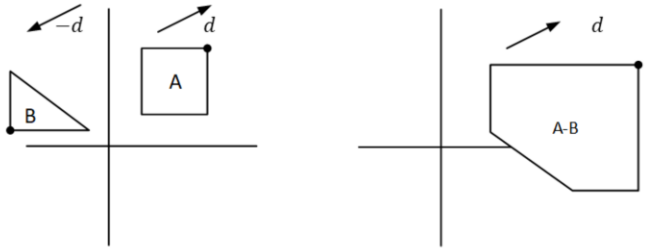


To use the Minkowski difference for GJK we need a way to find points on the Minkowski difference polygon. We know the Minkowski difference is defined as $A - B$ for all points on each shape. We can compute all of these points and then compute the convex hull of these points to get the final Minkowski difference polygon.

Building the CSO

Can implicitly build CSO from support functions

No convex hull needed!

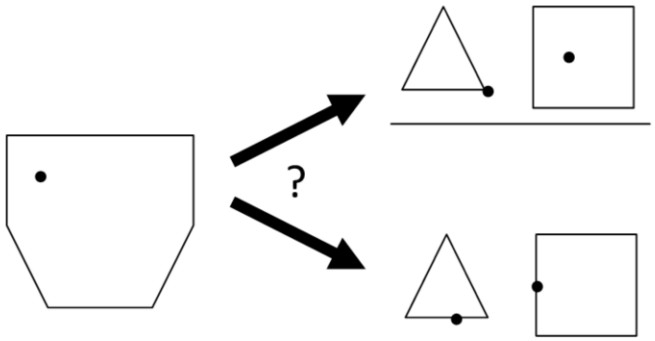


$$Support(A - B, d) = Support(A, d) - Support(B, -d)$$

There's one incredibly important property of the Minkowski difference that make our life a lot easier. We can compute a support point in the difference polygon from the supports of each shape. This means we don't need to explicitly compute the convex hull!

Building the CSO

How do we map a point from the CSO back?



There's just one more issue left. We defined how to convert shape vs. shape into point vs. shape, and we know the distance between these two are equal, but how do we convert a point on the CSO back to each object?

Unfortunately the mapping of the Minkowski difference isn't 1-1, so how do we map them back?

Building the CSO

First, store points used to construct CSO point

```
class SimplexPoint
{
  Vector3 mPoint;
};
```

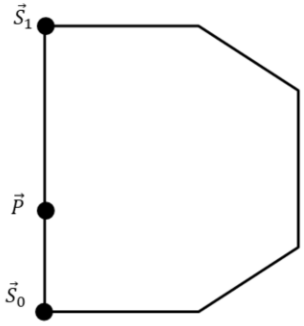


```
class SimplexPoint
{
  Vector3 mPointA;
  Vector3 mPointB;
  Vector3 mPointCso;
};
```

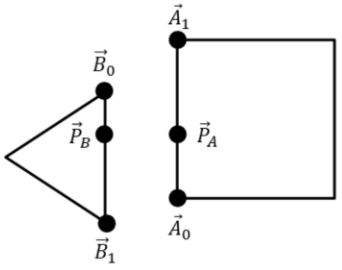
The first thing we have to do is alter our GJK algorithm to store 3 points for every simplex point we calculate. As a CSO point isn't unique we need to know which points from each shape were used to construct that point.

Building the CSO

Barycentric coordinates!



$$\vec{P} = \frac{2}{3}\vec{S}_0 + \frac{1}{3}\vec{S}_1$$



$$\vec{P}_B = \frac{2}{3}\vec{B}_0 + \frac{1}{3}B_1 \qquad \vec{P}_A = \frac{2}{3}\vec{A}_0 + \frac{1}{3}A_1$$

We can use barycentric coordinates to solve our problem! They have this neat property of being invariant under projections, meaning they can transform points across almost any transformation. Hence, we can compute the barycentric coordinates of $\vec{P}$ and use them to compute the closest points on each shape.

Collision Detection

GJK doesn't provide contact info (use EPA)

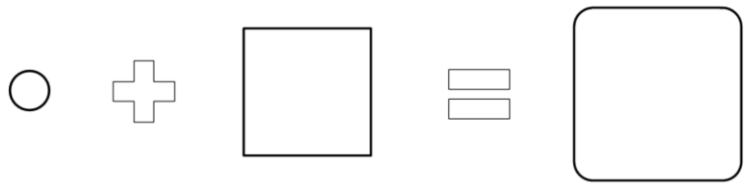
Some engines use object skins

Now that we know how to compute the closest points how do we actually compute contact information? Unfortunately GJK can only be used to get the minimum separation distance. A whole other algorithm (EPA) has to be used to find the minimum penetration distance.

One method commonly used to work around this (as EPA isn't that fast) is to add a skin around each object. The skin is an inflated distance where we consider the objects to actually be colliding. With this we can computed the separation distance between the objects and as longs as they're within the skin threshold then we count them as colliding.

Support Shape Combining

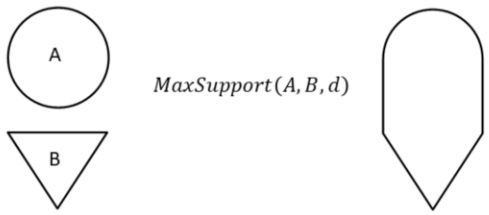
Minkowski Sum!



We can do all sort of interesting things to combine support functions. The simplest one is to use the Minkowski sum. This can be used to sweep one shape across the surface of another, for example to make a chamfered cube.

Support Shape Combining

Max of two support shapes

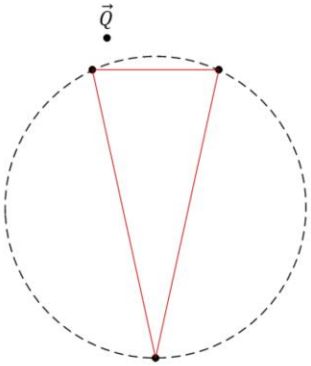


Another interesting one is to take the max of two support functions. We can run two independent support functions and then take whichever point is further in the search direction. This effectively shrink wraps the two shapes together.

Lots of various other support function combining is possible. For instance you can combine a line segment and a sphere to create a capsule!

Robustness

Analytic shapes will never terminate!

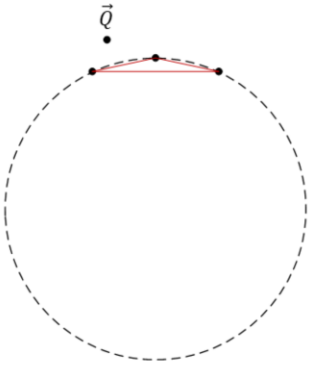


One thing we have to be careful of is analytic shapes. As they define a smooth surface we may have a problem with GJK's termination condition. Currently we only terminate when we stop making progress towards the origin, but what if we forever make small progress?

This example will never be able to terminate!

Robustness

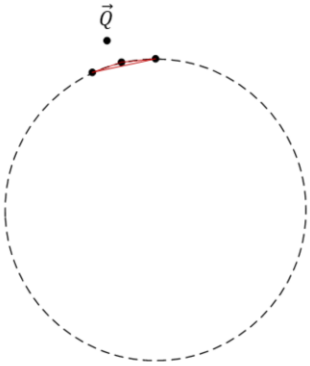
Analytic shapes will never terminate!



This example will never be able to terminate!

Robustness

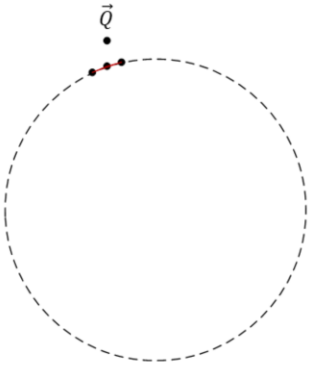
Analytic shapes will never terminate!



This example will never be able to terminate!

Robustness

Analytic shapes will never terminate!



This example will never be able to terminate!

Robustness

Add an epsilon check for step 6

Add a max iterations

There's two solutions that should both be implemented in GJK. The first one is to alter the termination condition. Instead of only terminating when we don't make progress, we should terminate when we don't make enough progress. That is we add an epsilon to see if the new search point is far enough away from the simplex.

The other solution is to add a max iterations. If we can't find a solution after enough tries then we must not be able to make enough progress so terminate. This can help out with numerical robustness, particularly in any case that causes an infinite flip-flop of states. Typical max iterations are around 20.

Optimizations

Not all voronoi regions need to be checked

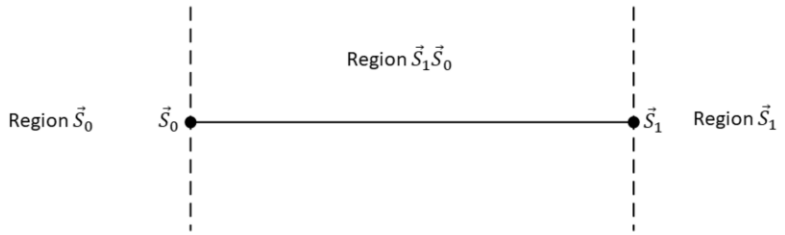
$\vec{Q}$ can't be in some

No optimizations for a point…

There are several optimizations that can be made to GJK by not checking voronoi regions that the query point can't be contained in. We'll start looking at a line, as no optimizations can be made for checking voronoi regions for a point.

Optimizations - Line

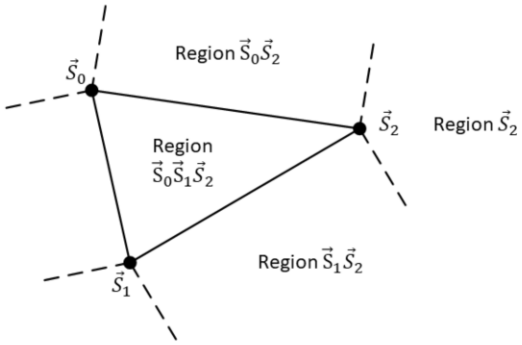
Point $\vec{S}_0$'s voronoi region can't be the closest



When we went from a point to a line, we searched in the direction of $\vec{Q}$ and found $\vec{S}_1$. Because of this $\vec{S}_0$ can't be the voronoi region that contains $\vec{Q}$, as by definition $\vec{Q}$ is on the front side of the line: $\vec{S}_1 - \vec{S}_0$ since we searched.

Optimizations - Triangle

Regions $\vec{S}_0, \vec{S}_1, \vec{S}_0\vec{S}_1$ can't be the closest



Likewise, for a triangle there are 3 voronoi regions that can't be the closest. If point's $\vec{S}_0$ or $\vec{S}_1$ were the closest then we would've reduced to them instead of expanding to a triangle, hence they can't be the closest. Likewise the edge region $\vec{S}_0\vec{S}_1$ can't be the closest since we just searched towards $\vec{Q}$ and found $\vec{S}_2$.

Optimizations - Tetrahderon

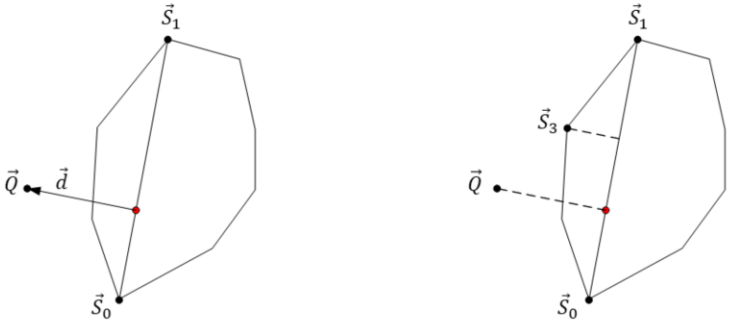
Regions $\vec{S}_0,\ \vec{S}_1,\ \vec{S}_2,\ \vec{S}_0\vec{S}_1,\ \vec{S}_1\vec{S}_2,\ \vec{S}_2\vec{S}_0,\ \vec{S}_{012}$ can't be the closest

General rule: Can't be the previous simplex's features

Finally for a tetrahedron there are 7 voronoi regions that can't be the closest. Hopefully a pattern starts to become visible here. When we expand a simplex the old simplex's features can't be the closest, if they were we would've reduced to that feature in the previous step.

Optimizations – Collision Detection

If we only care about Boolean results we can exit early



$\vec{S}_3$ isn't further in $\vec{d}$ that $\vec{Q}$; $\vec{Q}$ can't be contained in hull

There are some extra optimizations that can be made if we aren't interested in the closest point on the hull. That is, if we only care about a Boolean signifying if the hull contains $\vec{Q}$.

The first thing to note is that we have to alter GJK to have another early out. When we search for a new point in the direction $\vec{d}$ instead of just adding it to the simplex we can first see if we can even contain the origin. If the new point isn't further in the direction of $\vec{d}$ than $\vec{Q}$ then there's no way we can ever get the simplex to contain $\vec{Q}$.

In the above picture, $\vec{S}_3$ isn't further that $\vec{Q}$ in the direction of $\vec{d}$, hence we can never reach $\vec{Q}$.

Optimizations – Collision Detection

Extra regions that can be skipped

Line: $\vec{S}_1$

Triangle: $\vec{S}_2$

Tetrahedron $\vec{S}_3$

*Don't do this for your assignment!

Because of this we can skip 1 extra voronoi region per simplex. This region is the newest added point, as by definition the origin can't be beyond it or we would've already returned false.

Do remember, this is only valid for Boolean tests and we no longer generate closest features with this!

Questions?