Hello everyone!

Thank you for coming to my tutorial today.

My name is Erin Catto and I’m a programmer at Blizzard Entertainment.

Today I’m going to talk about computing distance.
Suppose we have a pair convex polygons.
How do we compute the closest points?

Knowing the closest points gives us the distance.
How do we detect overlap?

In this case the distance is zero.
The goal of this presentation is to describe an algorithm for computing the distance between convex polygons.
Keep in mind

- 2D
- Code not optimized

I will only be covering 2D in this presentation.

However, most of the concepts extend to 3D.

You may see algorithms and code that are not optimized.

This is a good sign.

It means you understand the material.
In this presentation I will use a bottom-up approach.

I will start with simple problems with simple solutions and then move towards more complex problems and solutions.

And each level builds from the previous levels.
I will try to build your geometric intuition for the algorithms.

After all, we are trying to solve a geometric problem.

As an added bonus you will get to see many nice pastel colors.
If all else fails ...

To complement the presentation, I have created an demo of the algorithm with source code.

I will post a link later.
Quickly show the demo.

All examples are solved with the distance algorithm you will learn today.
Outline

1. Point to line segment
2. Point to triangle
3. Point to convex polygon
4. Convex polygon to convex polygon

Here is an outline of the remainder of the this presentation.

Each topic builds from the previous one and many of the computations will be recycled.
A long the way, I will introduce several important concepts.

The first concept is Voronoi regions, which allow us to carve up the plane into closest feature regions.

The second concept is barycentric coordinates, which provide an object-based coordinate system based on a weighted sum of points.

The third concept is the GJK algorithm, which we will use to iteratively solve the point versus convex polygon problem.

The fourth concept is the Minkowski difference, which lets us convert the problem of polygon to polygon into point to polygon.
Let us begin with our first topic which covers point to line segment.
Say we have an line segment with vertices A and B.
Now, say we have a query point Q.
Closest point

We want to find the closest point P on the line segment.
We can find the closest point by projecting Q onto the line passing through A and B.

There are 3 regions where Q can project onto the line.

In this slide Q projects outside of the segment on side A.
In this slide Q projects inside the segment.
Finally, in this case Q projects outside of the segment on side B.

So that shows 3 cases.
So this brings up the concept of Voronoi regions.

We can carve the plane into closest feature regions.

All points in region A are closest to vertex A.

All points in region B are closest to vertex B.

All points in region AB are closest to the interior of AB.

These are called Voronoi regions.

Computing the closest point is a just a matter of determining the Voronoi region of Q.
We will use barycentric coordinates to compute the projection of $Q$ onto the line passing through segment $AB$.

What are barycentric coordinates?

Any point $G$ on the line passing through $AB$ can be represented as a weighted sum of $A$ and $B$.

Here we have labeled the weights $u$ and $v$.

These weights must sum up to one.

These weights are the barycentric coordinates of $G$ with respect to the line segment $AB$. 

\[ G(u,v) = uA + vB \]

\[ u + v = 1 \]
We can view the barycentric coordinates as the fractional lengths of partial segments.

In this slide the partial segments are balanced, so $u$ and $v$ are 0.5.
Fractional lengths

\[ G(u,v) = uA + vB \]
\[ u + v = 1 \]

In this slide the partial segments are not balanced.

Notice that as \( G \) approaches \( A \), \( u \) becomes larger and \( v \) becomes equally smaller.
Fractional lengths

\[ G(u,v) = uA + vB \]

\[ u + v = 1 \]
Let us define the unit vector $n$ pointing from A to B.

$n = \frac{B-A}{|B-A|}$
(u,v) from G

Now we can determine (u,v) based on the position of G.

(u,v) are determine by dotting the appropriate sub-segment onto n and then normalizing by the total length of AB.

Notice that u and v can individually be negative.

Also notice that u and v sum to one.
We can obtain \((u,v)\) directly from the query point \(Q\).

Notice that the dot products remain unchanged if we substitute \(Q\) for \(G\).

This is due to the projective nature of the dot product.
Voronoi region from \((u,v)\)

Any point \(G\) on the line passing through \(AB\) can be represented as a weighted sum of \(A\) and \(B\).

The weights must sum up to one.

These weights are the barycentric coordinates of \(G\) with respect to the line segment \(AB\).
Any point $G$ on the line passing through $AB$ can be represented as a weighted sum of $A$ and $B$.

The weights must sum up to one.

These weights are the barycentric coordinates of $G$ with respect to the line segment $AB$. 
Voronoi region from \((u,v)\)

Any point \(G\) on the line passing through \(AB\) can be represented as a weighted sum of \(A\) and \(B\).

The weights must sum up to one.

These weights are the barycentric coordinates of \(G\) with respect to the line segment \(AB\).
We can now write down our closest point algorithm.

We are given line segment $AB$ and query point $Q$.

First, we compute the barycentric coordinates.

From those we determine the Voronoi region and the closest point $P$. 
So we have covered point to line segment in great detail.

Let us now move on to a bit more challenging problem: point to triangle.
Here is a triangle in 2D.

Much like line segments, we can identity the Voronoi regions for a given query point.
Here the closest feature is vertex B.
Closest feature: edge

Here the closest feature is edge BC.
In this case, the closest feature is the triangle’s interior.
Let us carve up the plane again into Voronoi regions.

Remember these are closest feature regions.

For example, a point in region AB is closest to the interior of edge AB.

Note that the dashed lines are perpendicular to the neighboring edges.

We have 3 vertex regions, 3 edge regions, and 1 interior region.
We can treat the 3 edges of the triangle as 3 line segments.

We can determine the uv’s of Q with respect to each line segment.

We can combine the line segment barycentric coordinates to help us determine the triangle’s Voronoi regions.
By combining the line segment $uv$’s, we can determine if $Q$ is in a vertex region.

For example region A is determined by $u$ from segment CA and $v$ from segment AB.

That is what the subscripts indicate.
We can partially determine if a point is in an edge region using the line segment barycentric coordinates.

But we are missing some information because they do not tell us whether a point is above or below \( AB \).
Also, we don’t have a way to determine if a point is in the interior.
We can get the extra information we need by computing the barycentric coordinates of Q with respect to the triangle.

Let us express Q as a weighted sum of the triangle vertices.

As with the line segment, we require the barycentric coordinates to add up to 1, except now we have 3 coordinates instead of 2.

We are still putting no restriction on the sign, so the coordinates can be individually negative.

These new barycentric coordinates let us represent any point in the plane in terms of $(u,v,w)$. 
If we combine the previous equations, we can compute the barycentric coordinates of Q using linear algebra.

We will not compute them this way because that would reduce our geometric understanding.
Let us try to understand the barycentric coordinates for a triangle geometrically. For now we assume Q is in the interior.

Recall the point to line segment problem.

In that case, we related the barycentric coordinates to the fractional lengths of partial segments.

In a similar fashion, we can relate the triangle’s barycentric coordinates to the fractional areas of partial triangles.

We can compute the barycentric coordinates using these inscribed triangles.

Notice that the sum of the areas of the 3 inscribed triangles add up to the area of the parent triangle.

This indicates that the barycentric coordinates are proportional to the areas of the inscribed triangles.
The barycentric coordinates are the fractional areas

So $u$ is proportional to the area of $BCQ$.

Notice that $u$ is the barycentric coordinate for $A$, but it is proportional to the area of the only sub-triangle that doesn’t include $A$.

A similar rule applies to $v$ and $w$. 
Imagine if we move Q towards A.

Then u covers the whole triangle while v and w vanish.

Therefore, u is clearly the weighting of A.
Barycentric coordinates

\[ u = \frac{\text{area}(QBC)}{\text{area}(ABC)} \]

\[ v = \frac{\text{area}(QCA)}{\text{area}(ABC)} \]

\[ w = \frac{\text{area}(QAB)}{\text{area}(ABC)} \]

Based on the inscribed triangles, we arrive at these formulas for \( u \), \( v \), and \( w \).
Barycentric coordinates are fractional

- line segment: fractional length
- triangles: fractional area
- tetrahedrons: fractional volume

You may notice a trend here.

For line segments, we found that barycentric coordinates are fractional lengths.

For triangles, the barycentric coordinates are fractional areas.

You might guess that for tetrahedrons, the barycentric coordinates are fractional volumes.

And you would correct.
So how do we compute the area of a triangle?

Recall that the area of a parallelogram is found from the cross product of two of its edges.

In 2D the cross product is just a scalar.

The area of triangle ABC is half the area of the associated parallelogram.

Notice the area can be negative, depending on the winding order of ABC.
Since we use signed areas for the triangles, we can handle the case where Q is outside ABC.
When Q is outside edge BC, v and w remain positive and their sum becomes larger than one.
On the other hand, the winding order of BCQ has reversed, so $u$ becomes negative.

Nevertheless, the sum $u+v+w = 1$. 
Before we go continue, I need to make an important distinction.

Our approach is to compute closest points using Voronoi regions.

We compute barycentric coordinates to help find those regions.

In the case of a line segment, we found that the barycentric coordinates immediately gave us the Voronoi regions.

But for a triangle the Voronoi regions are not the same as the regions of determine by the barycentric coordinates.

We can deal with this, but lets first look at the barycentric regions for a triangle.
Here is our triangle.

What are the barycentric regions?
We have the interior region when all the coordinates are positive.
If $u$ is negative, then $Q$ is outside edge $BC$. 
Negative $v$

If $v$ is negative, then $Q$ is outside edge CA.
Finally, if $w$ is negative, the Q is outside edge AB.
The barycentric regions are not exclusive.

Two barycentric coordinates can be negative at the same time.

In this case the query point is outside of two edges simultaneously.

In general this does not indicate that Q is in a vertex region.

In this slide we see that Q is outside BC and CA, yet is closest to BC.
Finding the Voronoi region

- Use the barycentric coordinates to identify the Voronoi region
- Coordinates for the 3 line segments and the triangle
- Regions must be considered in the correct order

We now have enough information to determine a triangle’s Voronoi regions.

A query point $P$ can have barycentric coordinates with respect to the triangle and the 3 line segments independently.

This gives us a total of 9 barycentric coordinates.

Two for each line segment and three for the triangle.

We can use these nine scalars to determine the Voronoi region.

We must be careful to organize our logic correctly.

The correct approach is to consider the lowest dimensional features first: vertices, then edges, then the triangle’s interior.
We first test the vertex regions.

Here we use the 6 coordinates from the 3 line segments.
Recall that the line segment uv’s were not sufficient to determine if the query point was inside an edge Voronoi region.

Conveniently, the triangle’s uv’s provide the missing information.
We can combine the line segment uv’s with the triangle uv’s to determine if the query point is in an edge region.

We have two line segment uv’s to indicate if the query point is between A and B.

We now have the triangle w coordinate to indicate that the query point is above AB.
After all the other tests fail, we determine that Q must be in the interior region.

Since there can be no other case, we assert that triangle uv’s are all positive.
Closest point

- Find the Voronoi region for point Q
- Use the barycentric coordinates to compute the closest point Q

Finally we are prepared to compute the closest point on the triangle.

First we find the Voronoi region for point Q.

Then we use the appropriate barycentric coordinates to compute the closest point.
Here is an example of how to compute the closest point.

We start with a triangle ABC and a query point Q.
Example 1

We determine that $u_{AB}$ is negative.
Example 1

And that $v_{BC}$ is also negative.
We conclude that Q is in vertex region B.

So $P = B$.

A and C do not contribute to $P$. 

Example 1
Here is another example with Q in a different position.
First, we determine that Q is not inside any vertex region.
Example 2

We determine that Q is to the right of B.
Example 2

And to the left of A.

So Q is between A and B.
Example 2

We use the barycentric coordinate from the triangle to determine that Q is outside AB.

Therefore Q is in edge region AB.
Example 2

The closest point $P$ is on $AB$.

So we use the barycentric coordinates from $AB$ to determine $P$.

Conclusion:

$$ P = u_{AB} \cdot A + v_{AB} \cdot B $$
We have the triangle vertices and the query point Q as input.

Compute all the line segment barycentric coordinates.

Then compute the triangular barycentric coordinates.

Then we begin testing the Voronoi regions.
Testing the vertex regions

// Region A
if (vAB <= 0 && uCA <= 0)
    P = A
    return

// Similar tests for Region B and C

First we test the 3 vertex regions.

We can get an early return here.
Testing the edge regions

// Region AB
if (uAB > 0 && vAB > 0 && wABC <= 0)
    P = uAB * A + vAB * B
    return

// Similar for Regions BC and CA

Next we test the edge regions.
At this point, Q must be in the interior of the triangle, so we assert this is true.

Then the barycentric coordinates are just the triangular barycentric coordinates and all triangle vertices contribute to P.
So we have conquered point versus triangle.

Now let use move closer to our goal by looking at point to convex polygon.
Here is an arbitrary convex polygon ABCDE.

We say the polygon is convex because there exists no line segment between interior points that crosses an edge.
Polygon structure

```c
struct Polygon
{
    Vec2* points;
    int count;
};
```

Here is the code polygon structure.

This is just an array of 2D points.
Convex polygon: closest point

So now we have a query point Q.
We want to determine the closest point $Q$.

We can visualize this easily, however getting the computer to do this accurately and efficiently is another matter.
How do we compute $P$?
What do we know?

Closest point to point

Closest point to line segment

Closest point to triangle

These are the things we already know how to do.
For convenience we group points, line segments, and triangles under a common heading.

A simplex is a point, a line segment, a triangle, or a tetrahedron.

We say 0-simplex for a zero dimensional simplex (a point), 1-simplex for a 1 dimensional simplex (a line segment), and so on.
So here is an idea for computing $P$.

We inscribe a triangle in the polygon.

Why do that?

Because we know how to compute the closest point on a triangle.
Vertex C is the closest point on the simplex to Q.
Idea: evolve the simplex

Now use the closest point on the simplex to somehow evolve the simplex towards Q.
Here is the simplex vertex structure.

The point is copied from a polygon vertex.

We also store the index.

We include the barycentric coordinate u for closest point calculations.
Here is the simplex data structure.

It can represent a point, a line segment, or a triangle.
We are onto a winner!

So I think we are onto a winner.

We just have some details to work out.

Mainly we need to figure out how to evolve the simplex.
The GJK distance algorithm

- Computes the closest point on a convex polygon
- Invented by Gilbert, Johnson, and Keerthi

Fortunately, Gilbert, Johnson, and Keerthi have worked out the details for us in an algorithm called GJK.

Gilbert and company wrote several papers on the is algorithm in the late 1980s.

They are all very mathematical and difficult to understand.

They also lack nice pastel colors.
Despite it’s deep mathematical roots, GJK has an intuitive geometric interpretation.

GJK is an iterative algorithm.

We start with an arbitrary simplex inscribed in our convex polygon.

We compute the closest point on the simplex and use the result to evolve the simplex towards the closest point on the polygon.

In principle, we get an exact solution in a finite number of iterations. In reality, there are some numerical problems we have to deal with.

Rather than explain the GJK algorithm further, at this point it is better to look at an example of how it works.
Starting simplex

We have our convex polygon and query point Q.

We start the GJK algorithm by picking an arbitrary simplex.

In this case we pick a 0-simplex with vertex E.
Next we determine the closest point on our simplex to Q.

In this case P = E.
Now we build a search vector.

This vector points from the closest point on our simplex to P.

Since we have a 0-simplex, the closest point is E.
Find the support point

We now determine vertex furthest in the search direction.

This is call the support point.
Here’s the code for computing the support point.

The distance is determined with a simple dot product.

The code loops over all the polygon vertices and keeps track of the largest dot product.

Since the comparisons are all relative, the search direction d does not need to be normalized.
In this case it is visually obvious that C is the support point.

So what do we do with C?
Evolve the simplex

We merge C with the current simplex.

Since we had a 0-simplex with E, we now have a 1-simplex with CE.
We now repeat the process.

We find the closest point on our current simplex CE.
We build the new search direction to point from $P$ to $Q$. 

Build $d$ as a line pointing from $P$ to $Q$. 
Next, we compute the new support point in the direction $d$.

We find $D$ as the support point.
Evolve the simplex

Create triangle CDE.
This is a 2-simplex.

We merge D with our simplex and create the 2-simplex CDE.
We compute P as the closest point on the simplex.
E is worthless

Closest point is on CD.
E does not contribute.

We find that E does not contribute to P.

So we drop E from the simplex.

We must cull non-contributing vertices so that the simplex never has more than 3 vertices.
We dropped E, so we now have a 1-simplex.

The current simplex is now CD.

Note that we can drop from a 2-simplex to a 1 or 0 simplex.

So we can cull up to two simplex vertices at the same time.
We form a new search vector pointing from P to Q.

Now the new support point is either C or D.

We cannot determine the point uniquely.

But it doesn’t matter.

Since these vertices are already in the current simplex, we cannot make any more progress.

Therefore, we terminate the algorithm with P as the closest point.
Here is pseudo code for the GJK algorithm.

It is an iterative algorithm, so we must loop.

This code does not include termination conditions.

I will cover those in a moment.
DEMO!!!

But first, let's look at a demo of GJK iteration works.
Numerical Issues

- Search direction
- Termination
- Poorly formed polygons

As I warned you, there are a number of numerical problems we need to deal with.

In my experience these problems affect the search direction and loop termination.
So far we have computed the search direction as the vector from P to Q.

This is not always the best approach.

In this example, P is computed from barycentric coordinates on the line segment CE.

Barycentric coordinates suffer from some round off error so PQ may not be perpendicular to segment CE.
Here is a real world example I encountered.

I have a line segment $AB$ and a query point $Q$.

I compute the closest point $P$ and use that to form the search direction $d$.

When I compute dot search direction with the segment direction, I get a non-zero value.

Yes, it is a small error, but small errors in the search direction can be magnified.
Inaccurate search directions are bad because they can lead to incorrect support points.

This may lead to excessive iterations or even incorrect results.
It turns out we can get an exact search direction without difficulty.

When the closest point is on an edge, you should not use the closest point to compute the search direction.

Instead it is better to compute the search direction by forming a vector perpendicular to the edge.

We can get a perpendicular vector by crossing the edge vector with the plane normal.
The dot product is exactly zero

edge direction: \( e = (x, y) \)
search direction: \( d = (-y, x) \)
dot product: \( e \cdot d = -xy + yx = 0 \)

This means the search direction is perpendicular to the edge.

The dot product is numerically zero, even with floating point arithmetic.
Then we just need to flip the sign of $d$ so that it points towards $Q$. 

Flip the sign of $d$ so that:

$$\text{dot}(d, Q - C) > 0$$

Perk: no divides
Along with inaccurate search directions, termination conditions can cause a lot of grief.

We don’t want to terminate too early or not at all.

There is nothing that will kill your frame-rate more than an infinite loop.

So we should try to do a good job on termination conditions.
Our first case is a repeated support point.

In this case we compute a support point that belongs to our current simplex.

This indicates we can make no more progress, so we should terminate with P as the closest point.
Case 2: containment

At some point in our GJK iterations we may find a 2-simplex where all triangular barycentric coordinates for Q are positive.

This means Q is inside the 2-simplex, and therefore inside the parent polygon.
Case 3a: vertex overlap

We will compute $d = Q - P$ as zero. So we terminate if $d = 0$.

If $Q$ overlaps a vertex then $P = Q$ and we will compute a zero search direction.

So we should terminate if $d$ is zero.
Case 3b: edge overlap

It is possible that Q overlaps an edge.

We can compute \( \mathbf{d} \) as a perpendicular vector.

However, we cannot determine its sign.

Since the sign of \( \mathbf{d} \) will be arbitrary, we should inspect both cases.
Case 3b: d points left

If we search left, we get a duplicate support point.
In this case we terminate.

If d points left then we get Case 1: a duplicate support point.
Case 3b: $d$ points right

If we search right, we get a new support point ($A$).

If $d$ points right, then we get a new support point.
Case 3b: d points right

But then we get back the same $P$, and then the same $d$.

Soon, we detect a repeated support point or detect containment.

Depending on round-off error, we will find that either $P$ is on edge CD or $P$ is inside the triangle ACD.

So will get a duplicate vertex or detect containment.
Case 4: interior edge

The final case we consider is when Q overlaps an interior edge of our simplex.

In this case our simplex is an interior line segment.

Because Q overlaps the line segment, we are unable to determine d uniquely.
Case 4: interior edge

This is ultimately the same as Case 3b.

No matter which sign of $d$ we choose, we should be able to detect a repeated vertex.
Termination in 3D

- May require new/different conditions
- Check for distance progression

GJK in 3D likely needs different termination conditions.

For example, you may need to test that the distance from P to Q decreases each iteration.
What happens if we get a non-convex polygon?
Non-convex polygon

GJK won’t see the non-convex vertex because it will never show up as a support point.
Collinear vertices

What happens with collinear points?

B, C, and D are collinear
Collinear vertices

We could end up with a degenerate triangular simplex.
The triangle area is zero, so the triangular barycentric coordinates are infinite.

You can avoid dividing by the area.

You should ensure that your triangle solver will pick a vertex or edge region in this case.

See the demo code for details.
Section 4

Convex Polygon to Convex Polygon

So that is it for GJK.

We are now ready to tackle our end goal: closest point between a pair of convex polygons.
So we want to compute the closest point between two convex polygons.

This may seem much more complex than closest point on a single polygon.

However, there is a way to relate the two problems.
What do we know?

We applied our knowledge of the point to simplex problem to solve point to polygon using GJK iteration.
What do we need to know?

Now, how can we apply our knowledge of point to polygon to solve polygon to polygon?
Idea

- Convert polygon to polygon into point to polygon
- Use GJK to solve point to polygon

The basic idea is that we convert polygon to polygon into point to polygon.

Then we can use GJK to solve the point to polygon problem.
It turns out there is a geometrical construction called the Minkowski difference that will give us exactly what we need.

The Minkowski difference lets us combine two polygons into a single convex polygon.
Here is the mathematical definition of the Minkowski difference.

\[ Z = \left\{ y_j - x_i : x_i \in X, y_j \in Y \right\} \]
Building the Minkowski difference

Input: polygon X and Y
array points
for all xi in X
    for all yj in Y
        points.push_back(yj - xi)
    end
end

polygon Z = ConvexHull(points)

We can build the super polygon using a straightforward technique.

We use two nested loops to compute the difference between the points of the two polygons.

Then we compute the convex hull of the point cloud.

Of course this is not cheap to do, but please be patient.

There is gold at the end of the rainbow.
Example point cloud

Compute $Y_i - X_j$ for $i = 1$ to $4$ and $j = 1$ to $3$

Here is what the point cloud looks like for this triangle and square.

Yes, I actually compute all those points.
I will not cover convex hull algorithms, but conceptually we imagine a convex hull algorithm as a shrink wrapping process.
Building the convex hull

We start with extreme point and make our way around the perimeter.
Building the convex hull

Compute the convex hull by shrink wrapping the points.
Building the convex hull

Compute the convex hull by shrink wrapping the points.
Building the convex hull

Compute the convex hull by shrink wrapping the points.
Building the convex hull

Compute the convex hull by shrink wrapping the points.
Building the convex hull

Compute the convex hull by shrink wrapping the points.
Building the convex hull

Compute the convex hull by shrink wrapping the points.
Building the convex hull

Compute the convex hull by shrink wrapping the points.
The final polygon

We have now arrived at the Minkowski difference polygon.
The Minkowski difference has several remarkable properties.

First, the distance between X and Y is equal to the distance between the origin and the super polygon.
Second, we can determine a support point of Z by combining the support points of X and Y.

We use the support point in direction \( d \) on Y and the support point in direction \(-d\) in X.

The difference of these two support points is equal to the support point in Z.

We now know how to compute the support point of Z without building Z explicitly!
This means we do not need to build $Z$ explicitly.

So we do not need to compute the convex hull.

This saves us an enormous amount of space and time in our applications.
Modifying GJK

- Change the support function
- Simplex vertices hold two indices

We can easily modify GJK to handle the Minkowski difference.

We just need to change the support function and add a little bit more bookkeeping.

See the demo code for details.
Closest point on polygons

- Use the barycentric coordinates to compute the closest points on X and Y
- See the demo code for details

We can also modify GJK to compute the closest points on the original polygons.

Then we apply the barycentric coordinates to the original vertices to determine the closest points.

Again, please see the demo code for details.
Show GJK iteration on triangle versus square.

You can get this presentation and the demo code at box2d.org.

You’ll have to click the download link, which will lead you to a Google code page that has another download link.
Further reading

- Collision Detection in Interactive 3D Environments, Gino van den Bergen, 2004
- Real-Time Collision Detection, Christer Ericson, 2005

There are many references on the topics I covered. Some of the better ones are listed here.

I definitely recommend that you watch Casey Muratori's video.

It has some interesting optimizations that can be applied to GJK.
Box2D

- An open source 2D physics engine
- [http://www.box2d.org](http://www.box2d.org)
- Written in C++

The demo code was originally born in the Box2D project. Box2D is an open source 2D physics engine.

Box2D uses GJK for its continuous collision algorithm.