Geometricks

(sorry)

Zino

@krugazor

It is

* “obvious” when you look at it
* Painful memories from high school
* Quite the head scratcher when it surprises you while coding

It is also

* Probably the best understood domain in algorithmic
* A lot of tips and tricks that look magical
* A decent way to shine (and get invited to talk at a conference)

If you have to

* Draw complicated stuff (zealous UI design, games)
* Transform stuff (image manipulation)
* Use stuff that rely violently on it (MapKit)

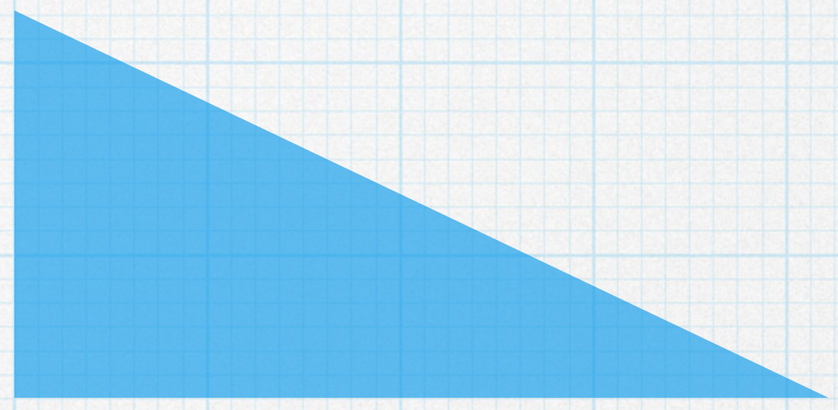
Back to School (I know, I know)

* Line: $y = ax+b$ <u>or</u> $mx+ny+p = 0$
* If you pick 3 points in a plane, there probably isn't any straight line that goes through them all
* But triangles are pretty cool, actually

In practice (let's start slow)

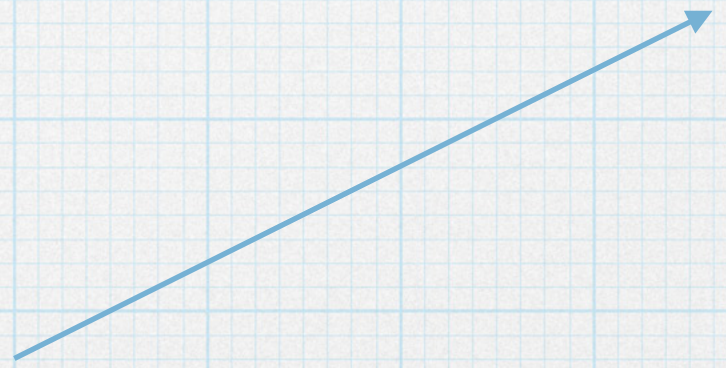
Distance between two points

$\sqrt{((y_2-y_1)^2+(x_2-x_1)^2)}$



Inclination of a line

$(y_1-y_2)/(x_2-x_1)$

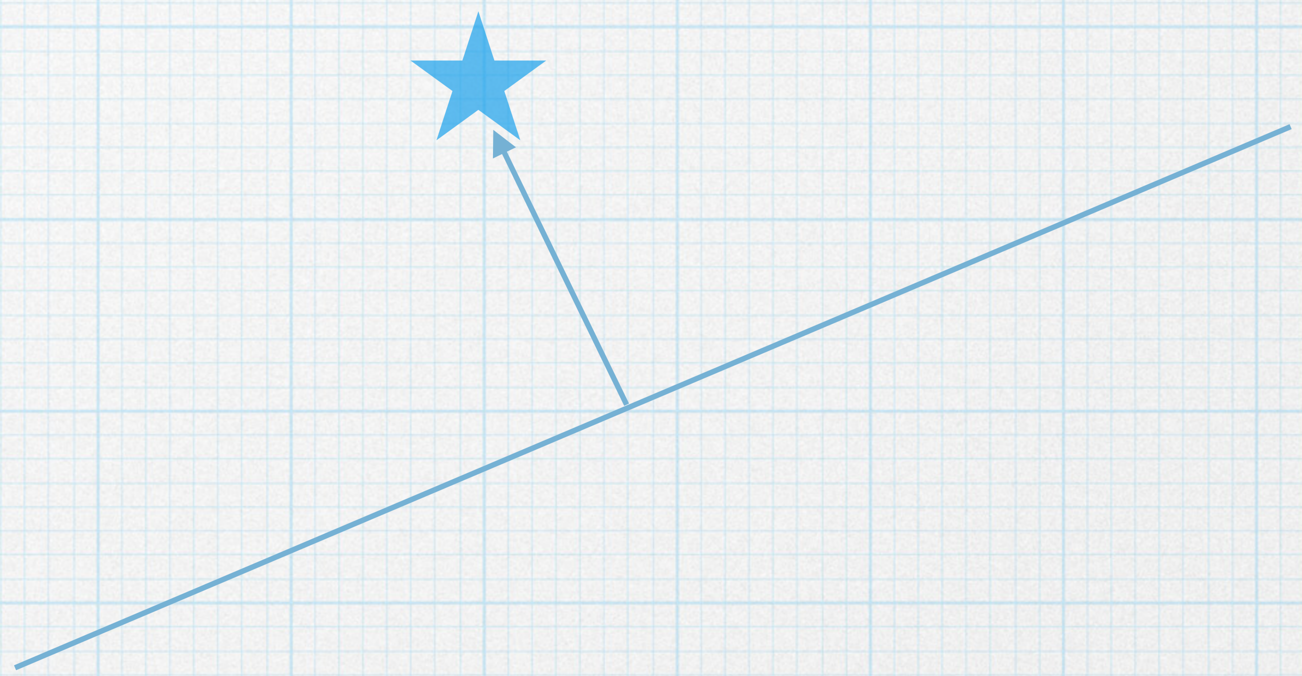


In practice

Distance between a line and a point

* Line: $Ax + By + C = 0$
    * $A = (y_1 - y_2)$ $B = (x_2 - x_1)$

      $C = (x_1 - x_2) * y_1 + (y_2 - y_1) * x_1$
* Point: (m, n)
* Distance: $| Am + Bn + C | / \sqrt{(A^2 + B^2)}$



A bit of Swift

* typealias Point = (x: Double, y: Double)
* func distance(p1: Point, p2: Point) -> Double {
  return sqrt((p2.y-p1.y)*(p2.y-p1.y)+(p2.x-p1.x)*(p2.x-p1.x))
  } // pow is slower

Not true!

```swift
let a = 3.0
let b = 4.0

let startPow = Date()
for _ in 1..<100000000 {
    let _ = pow(a+b,2)
}
let endPow = Date()

let start = Date()
for _ in 1..<100000000 {
    let _ = (a+b)*(a+b)
}
let end = Date()

let timePow = endPow.timeIntervalSince(startPow)
let time = end.timeIntervalSince(start)
```

```
3
4

"Feb 3, 2019 at 22:42"



"Feb 3, 2019 at 22:42"

"Feb 3, 2019 at 22:42"



"Feb 3, 2019 at 22:43"

40.25218307971954
32.06360900402069
```

(your mileage may vary)

Not true!
("It's Playground's fault!")

```swift
typealias Point = (x: Double, y: Double)

let p1 : Point = (3.0,7.0)
let p2 : Point = (4.0,-2.0)

let iterations = 10000000000

let startPow = Date()
for _ in 0..<iterations {
    let _ = sqrt(pow(p2.y-p1.y,2) + pow(p2.x-p1.x,2))
}
let endPow = Date()

let start = Date()
for _ in 0..<iterations {
    let _ = sqrt((p2.y-p1.y)*(p2.y-p1.y) + (p2.x-p1.x)*(p2.x-p1.x))
}
let end = Date()

let timePow = endPow.timeIntervalSince(startPow)
let time = end.timeIntervalSince(start)

print("\(timePow) vs \(time)")
print("~\(round((timePow*100)/time) - 100)% faster on average")
```

```
46.832216024398804 vs 45.93707299232483
~2.0% faster on average
Program ended with exit code: 0
```

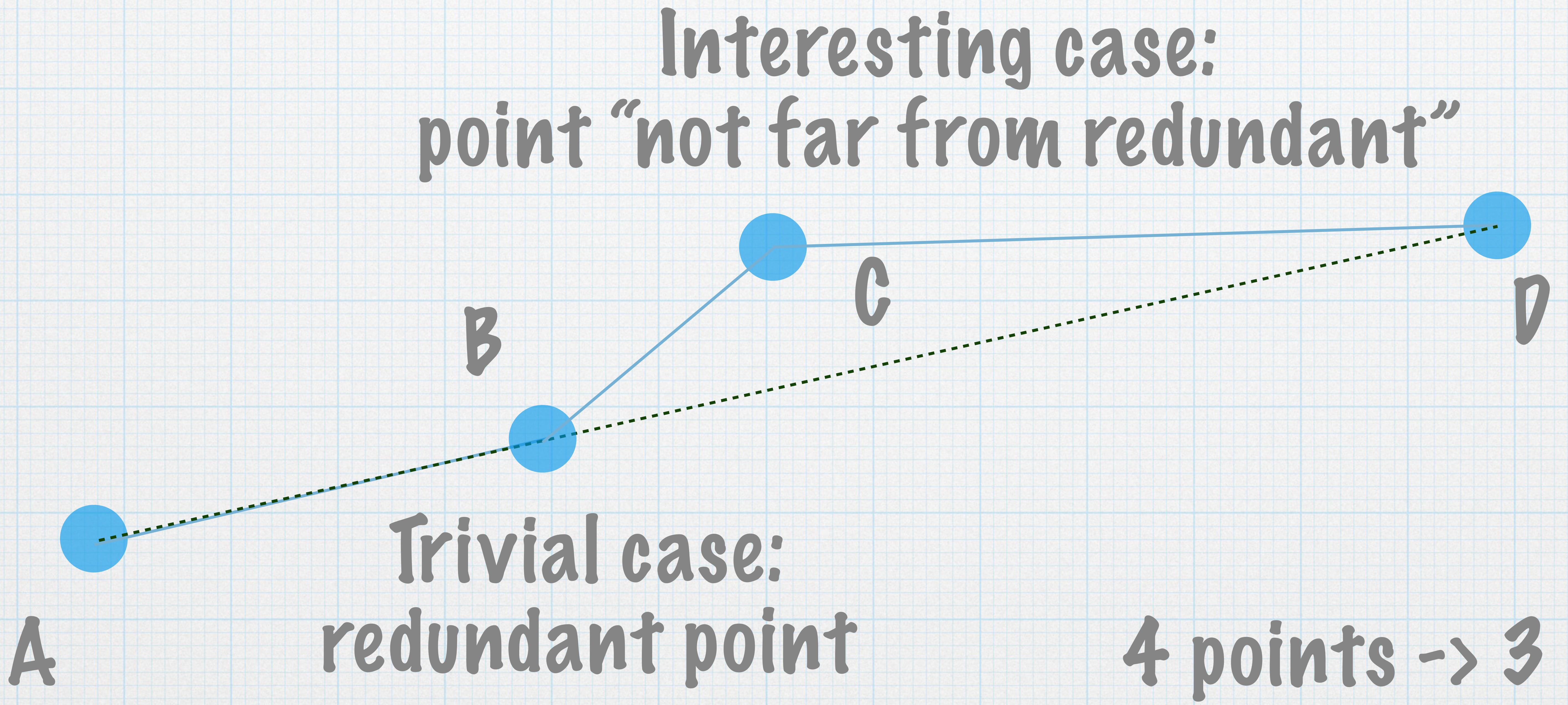
(your mileage may vary)

A bit of Swift

* ```
  func getLineParameters(point1: Point, point2: Point)
      -> (a: Double, b: Double, c: Double) {
    let a = point1.y - point2.y
    let b = point2.x - point1.x
    let c = ((-b)*point1.y) + ((-a)*point1.x) // yup!

    return (a,b,c)
  }
  ```
  ```

A bit of Swift

* ```
  func getPerpendicularDistance(line: (a: Double, b: Double, c: Double), point: Point) -> Double {
    let num = abs(line.a * point.x + line.b * point.y + line.c)
    let den = sqrt(line.a * line.a + line.b * line.b)

    return num/den
  }
  ```
  ```

Ramer-Douglas-Peucker



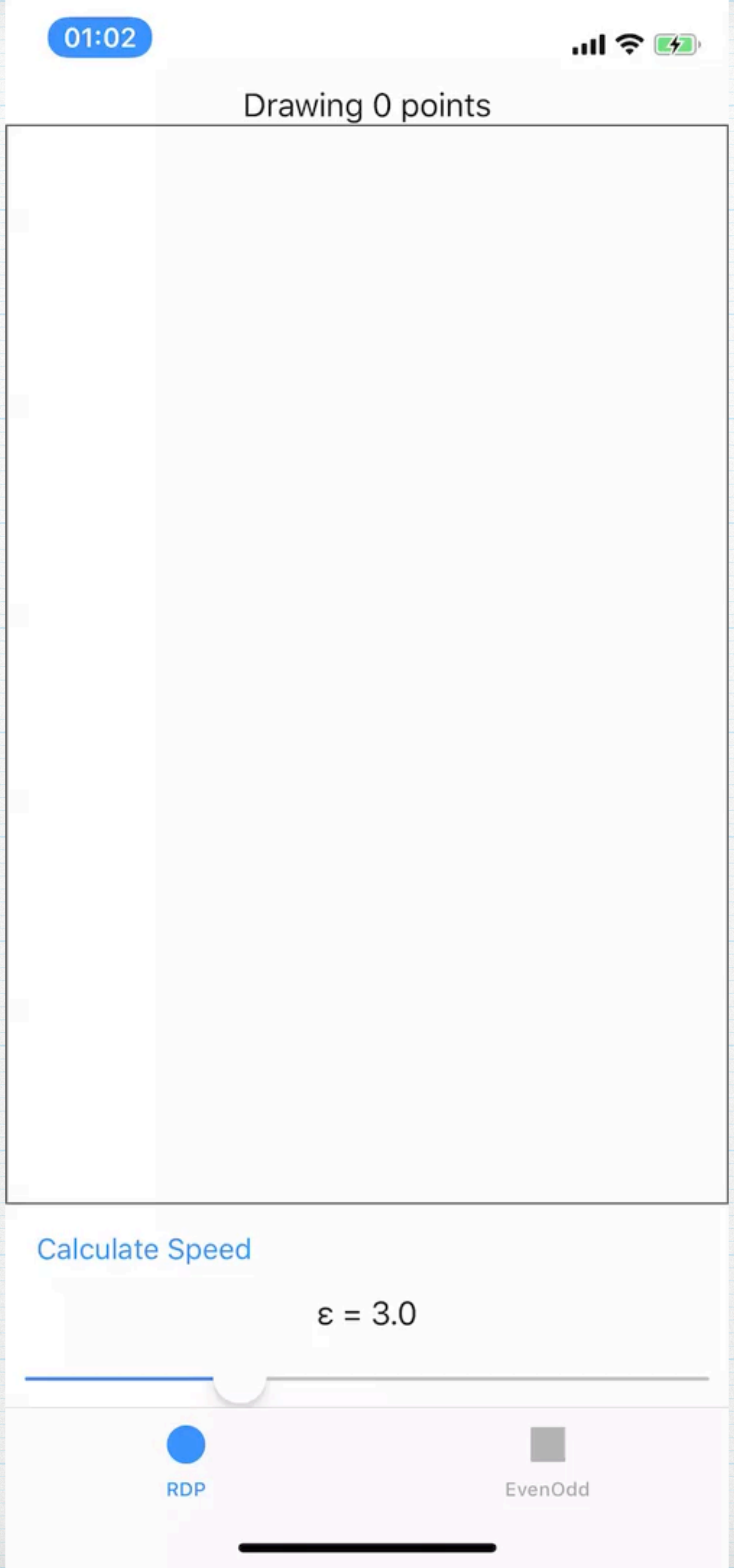
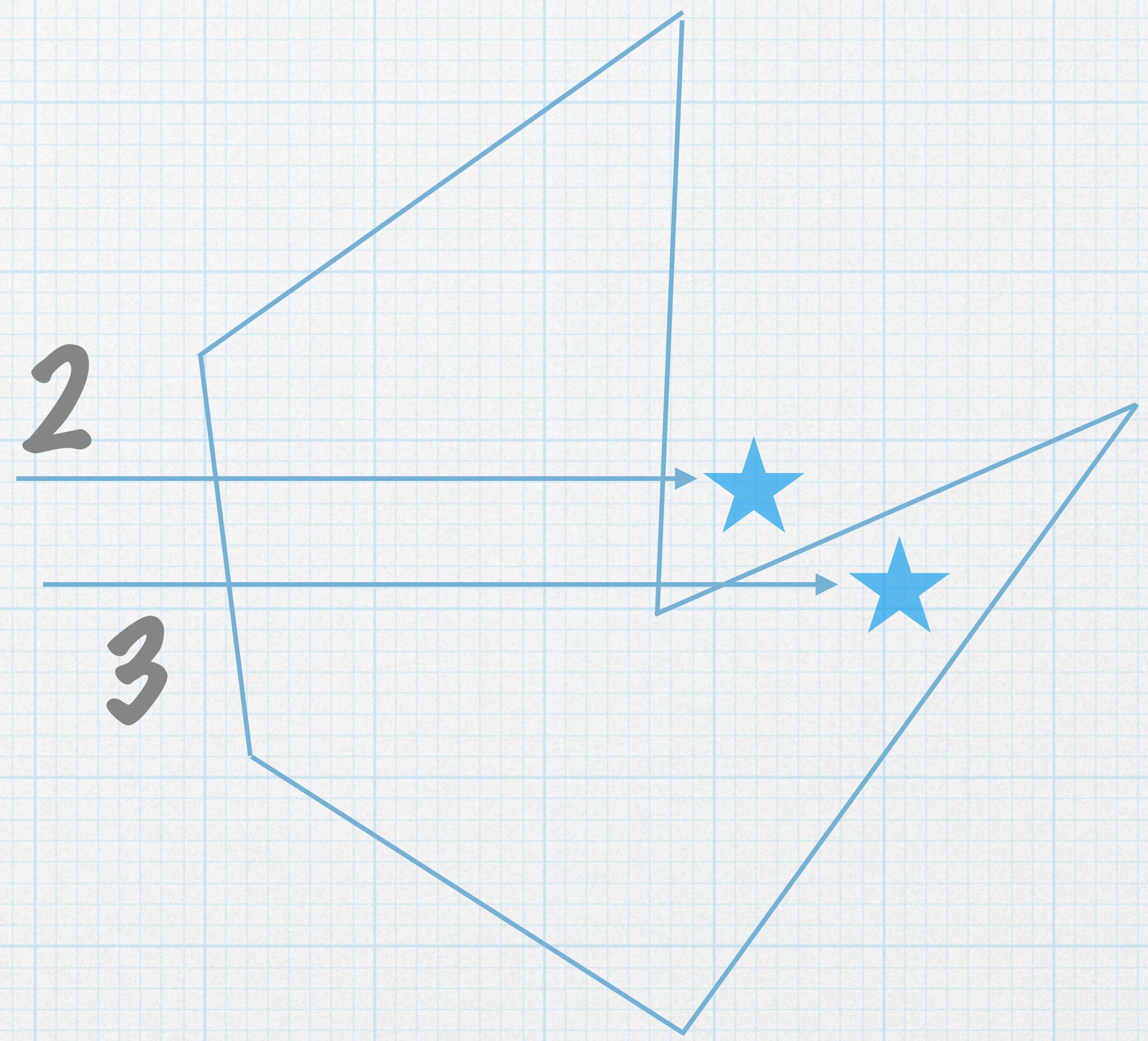
4 points -> 3 points (or even 2)

Ramer-Douglas-Peucker

* Start at the extremities (A & D)
* Look for point that deviates the most (C)
* If the distance between that point and the segment exceeds a minimal ε, start again with (A & C) and (C & D)
* If not, keep only the extremities

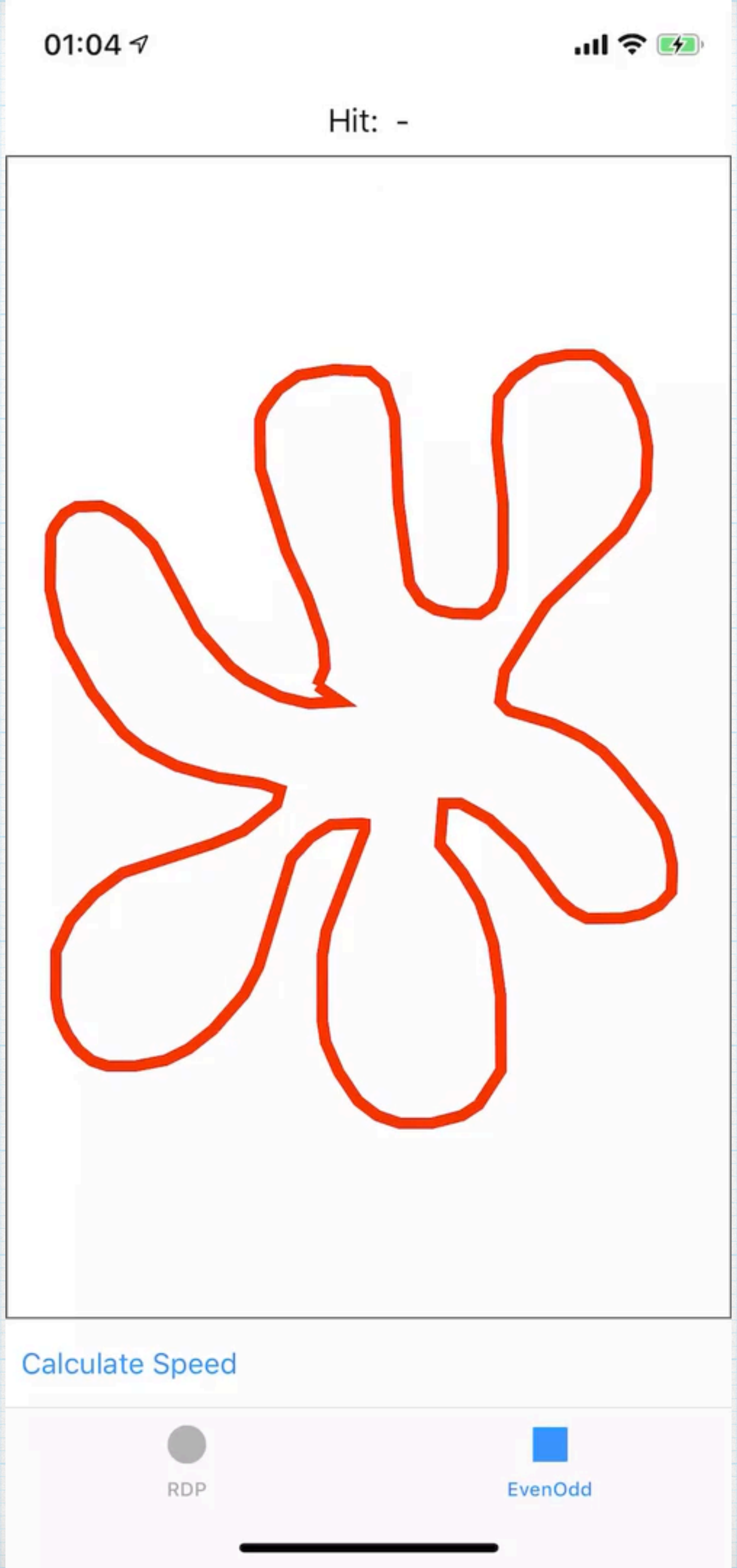
A bit of Swift

* ```
func douglaspeuckerSimplification(line: [Point], epsilon: Double) -> [Point] {
  if line.count <= 2 { return [line.first!, line.last!] }
  // Find the point with the maximum distance
  var dmax : Double = 0
  var index = 0
  let (a,b,c) = getLineParameters(point1: line.first!, point2: line.last!)
  for i in 1..<(line.count-1) {
    let d = getPerpendicularDistance(line: (a,b,c), point: line[i])
    if dmax < d {
      dmax = d
      index = i
    }
  }
```
```

A bit of Swift

*
```
if dmax > epsilon {
  let sub1  = Array(line[0..<index+1])
  let sub2 = Array(line[index..<line.count])
  let res1  = douglaspeuckerSimplification(line: sub1 , epsilon: epsilon)
  var res2 = douglaspeuckerSimplification(line: sub2, epsilon: epsilon)
  res2 = Array(res2.dropFirst())

  return res1 + res2
} else {
  return [line.first!, line.last!]
}
}
```

Demo

iPhone Xs



Click bait

* point in a polygon
* square (easyyyyyy)
* circle or even oval (duh)
* ... machine learning?

Raytracing

* Technique from the 3D world
* A ray starts from the camera, then “hits” an object
* From that point, find the lights that shine on that point
* What does that have to do with 2D?

Even-Odd

* How many times does a horizontal "ray" to the point hit a segment?
* Even number? we're outside (eg, 0)
* Odd Number? we're inside (eg, 1)



Even-Odd

* If the point is above or below the segment, no intersection

* If solving the equation y = point.x == y = line(segment) has no solution, we don't intersect

* Else we intersect

* We alternate outside/inside/outside/...

A bit of Swift

* ```
  typealias Polygon = [Point]
  func evenOdd(_ point: Point, in poly: Polygon) -> Bool {
   var inside = false
   var j = poly.count - 1
   for i in 0..<poly.count {
     if (poly[i].y > point.y) != (poly[j].y > point.y)
       &&
     ( point.x < poly[i].x + ( poly[j].x - poly[i].x ) * ( point.y - poly[i].y) / (poly[j].y - poly[i].y) ) {
         inside = !inside
     }
     j = i
   }
   return inside
  }
  ```
  ```

Demo

iPhone Xs



(Vague) Conclusions

* Disrupting dairy consumption, why not. Disrupting geometry is a little harder (and useless)
* A tiny teeny bit of maths makes the performance soar
* Stack Overflow, that's nice. Swift Algorithm Club, that's better

Swift Algorithm Club

* Ray Wanderlicht
* Classical algorithms with code, explanations, demos and even animations!
* https://github.com/raywenderlich/swift-algorithm-club/

Swift Algorithm Club

Least Common Multiple

An idea related to the GCD is the *least common multiple* or LCM.

The least common multiple of two numbers `a` and `b` is the smallest positive integer that is a multiple of both. In other words, the LCM is evenly divisible by `a` and `b`.

For example: `lcm(2, 3) = 6` because 6 can be divided by 2 and also by 3.

We can calculate the LCM using Euclid's algorithm too:

```
              a * b
lcm(a, b) = ---------
            gcd(a, b)
```

In code:

```
func lcm(_ m: Int, _ n: Int) -> Int {
  return m / gcd(m, n) * n
}
```

And to try it out in a playground:

```
lcm(10, 8)    // 40
```

You probably won't need to use the GCD or LCM in any real-world problems, but it's cool to play around with this ancient algorithm. It was first described by Euclid in his Elements around 300 BC. Rumor has it that he discovered this algorithm while he was hacking on his Commodore 64.

Written for Swift Algorithm Club by Matthijs Hollemans

Swift Geometric Club?

* Eeeeeeeeeeh...
* 😎
* Other questions?