Hit Testing

June 12

Implementing GUI Direct Manipulation

We expect our GUIs to be interactive: graphical elements are directly manipulated using a pointing device

- manipulation includes graphical content, widgets, etc.

Key requirement is to detect *what* the mouse cursor is pointing at

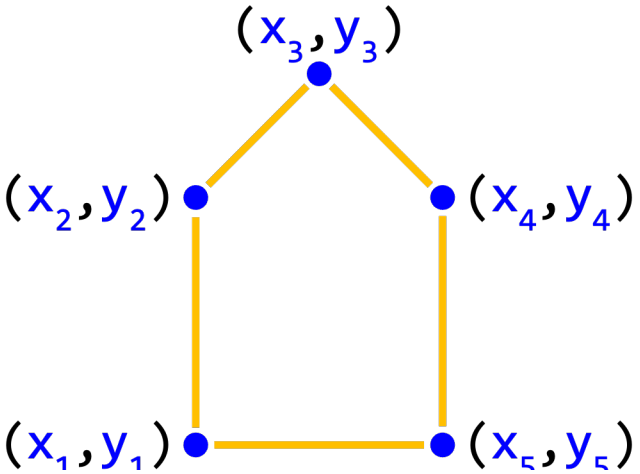
- all graphical content can be described as some “shape”
- shape could be filled, outlined, or special case like text – each supports a different type of interaction (border, vs. interior)
- need to consider reasonable tolerances for usability (consider near misses as hits for small / narrow shapes)

Today we walk through how to do this from the ground up (i.e. case where we want to draw using GC, or some other system).

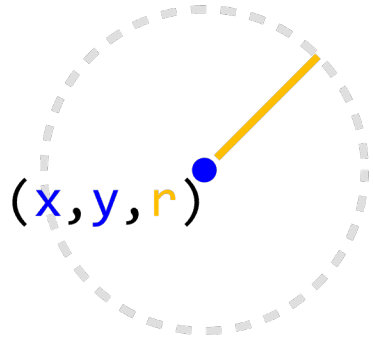
- a general **model** to describe shapes
- **hit-tests** to detect when a cursor is inside shape or on its edge

Shape Model Geometry

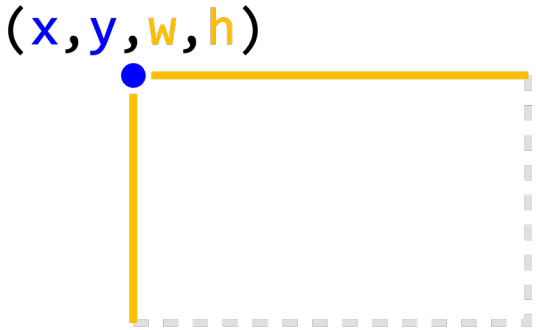
Different shapes have different *geometric representations*:



Polygon
list of points



Circle
center, radius



Rectangle
top-left corner,
width,height

- Alternate geometric representations are possible
- Many other kinds of shapes: Line, Polyline, Polygon, Ellipse, …
- Shape models can even be combinations of shapes

Simple Shape Model Class

What does a shape model class require?

Properties

- geometry that defines the Shape (a list of points)
- geometry properties (isFilled, isStroked)
- visual style properties (fill, stroke, strokeWeight)

Methods

- method to **draw itself** into a provided graphics context (i.e. render)
- method to do **hit-testing** with an x-y cursor position - *new*

Shape Model Implementation

Define a Shape base class:

```
abstract class Drawable(var x: Double, var y: Double,
                        var col: Color) {

    abstract fun draw(gc: GraphicsContext)
    abstract fun isHit(x: Double, y: Double): Boolean

    override fun toString(): String {
        return col.getName()
    }
}

// Extension function for Double
fun Double.between(low: Double, high: Double): Boolean {
    return this in low .. high
}
```

Demo: HitTesting/ShapeModels

Rectangle Shape Model Implementation

```
class FillRect(var x: Double, var y: Double,
               var w: Double, var h: Double,
               col: Color): Drawable(x, y, col) {

    override fun draw(gc: GraphicsContext) {
        gc.apply {
            save()
            fill = col
            fillRect(x, y, w, h)
            restore()
        }
    }

    override fun isHit(x: Double, y: Double): Boolean {
        // ...
    }
}
```

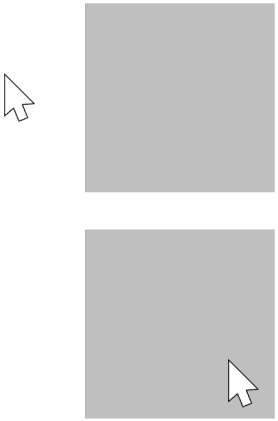
Circle Shape Model Implementation

```
class FillCirc(x: Double, y: Double,
               var d: Double,
               col: Color):
               Drawable(x, y, col) {

    override fun draw(gc: GraphicsContext) {
        gc.apply {
            save()
            fill = col
            fillOval(x, y, d, d)
            restore()
        }
    }

    override fun isHit(x: Double, y: Double): Boolean {
        // ...
    }
}
```

Hit-Test Paradigms

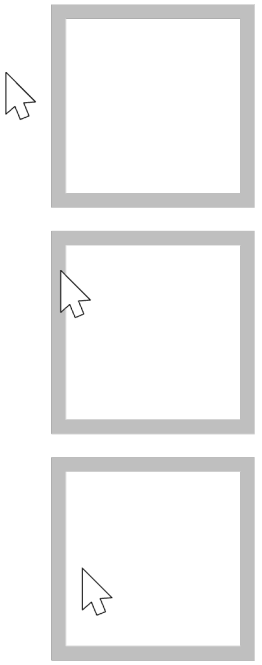
Inside Hit-Test: is mouse cursor inside shape?

- Applies to closed and filled shapes like ovals, rectangles, and polygons.



Edge Hit-Test: is mouse cursor on shape outline?

- Applies to open and “non-filled” shapes like strokes, lines, and polylines.



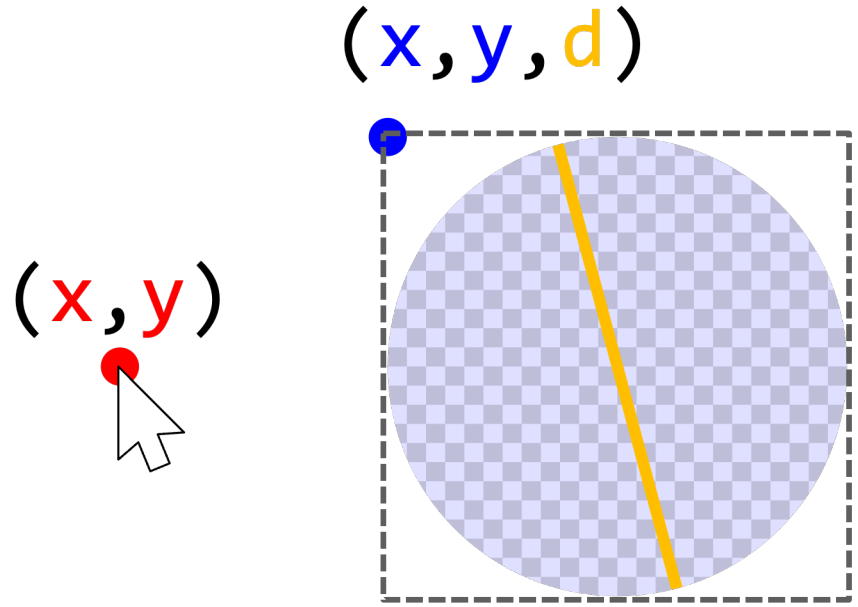
A hit-test is tailored to the shape type and properties

- if no fill, hit-test should be on shape outline only
- hit-test should factor in the thickness of stroke

Filled Circle Hit-Test

Given:

- Mouse position (x,y)
- Upper-left bound of circle (x,y)
- Diameter d



Compute:

- Origin (x + d/2, y + d/2)

Hit:

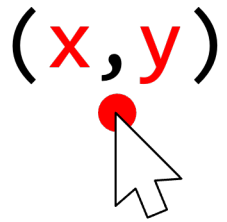
```
if distance
      from (x,y)
      to (x + d/2, y + d/2)
      <= d/2
```

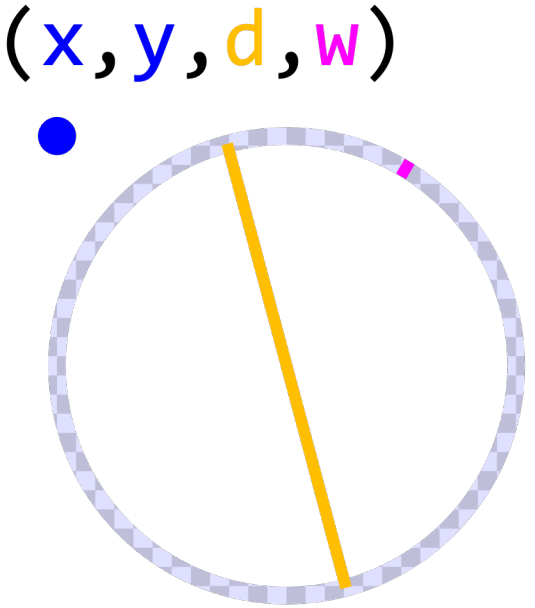
```
override fun isHit(x: Double, y: Double): Boolean {
    return sqrt(sqr(x - this.x - d / 2.0) +
                sqr(y - this.y - d / 2.0)) <= d / 2.0
}
```

Stroke Circle Hit-Test

Given:

- Mouse position (x,y)
- Upper-left bound of circle (x,y)
- Diameter d
- Stroke width w





Hit:

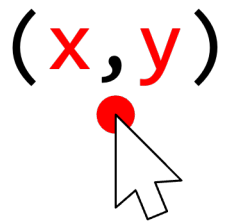
```
if distance
      from (x,y)
      to (x + d/2, y + d/2)
      between (d/2 + w/2) and (d/2 - w/2)
```

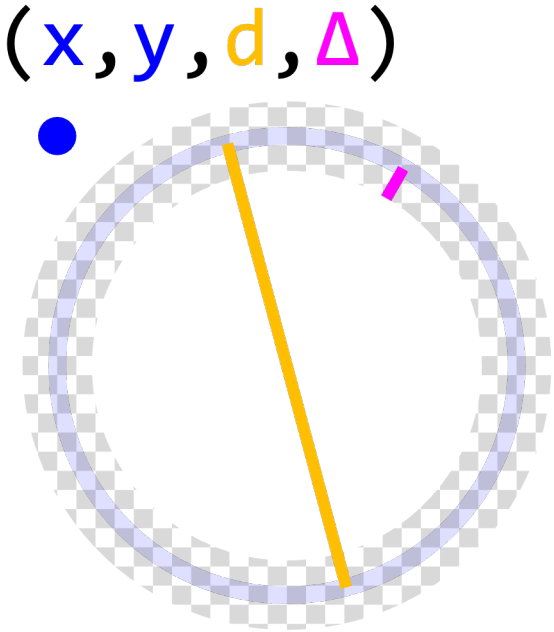
```
override fun isHit(x: Double, y: Double): Boolean {
    return sqrt(sqr(x - this.x - d / 2.0) +
                sqr(y - this.y - d / 2.0)).between(d/2.0 + 0.5,
                                                   d/2.0 - 0.5)
}
```

Stroke Circle Hit-Test

Given:

- Mouse position (x,y)
- Upper-left bound of circle (x,y)
- Diameter d
- Delta Δ





Hit:

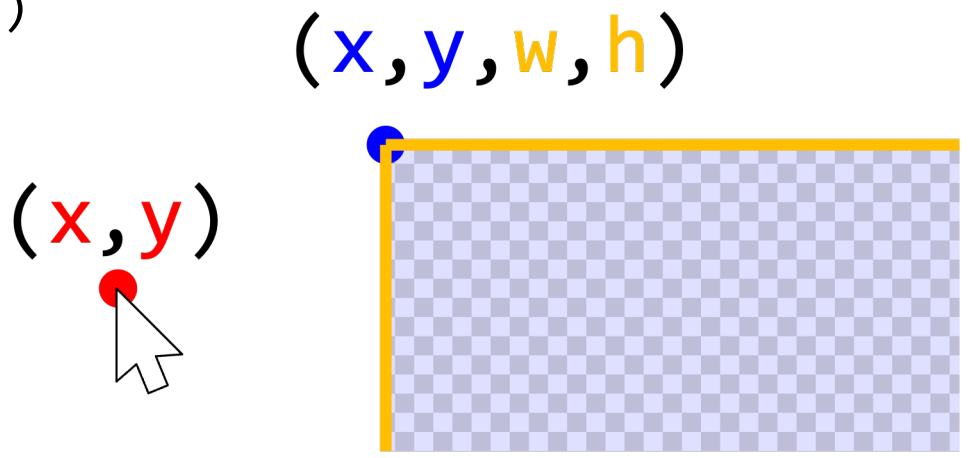
```
if distance
      from (x,y)
      to (x + d/2, y + d/2)
      between d/2 - Δ and d/2 + Δ
```

```
override fun isHit(x: Double, y: Double): Boolean {
    return sqrt(sqr(x - this.x - d / 2.0) +
                sqr(y - this.y - d / 2.0)).between(d/2.0 - delta,
                                                   d/2.0 + delta)
}
```

Filled Rectangle Hit-Test

Given:

- Mouse position (x,y)
- Upper-left bound of rectangle (x,y)
- Width and height (w,h)



Hit:

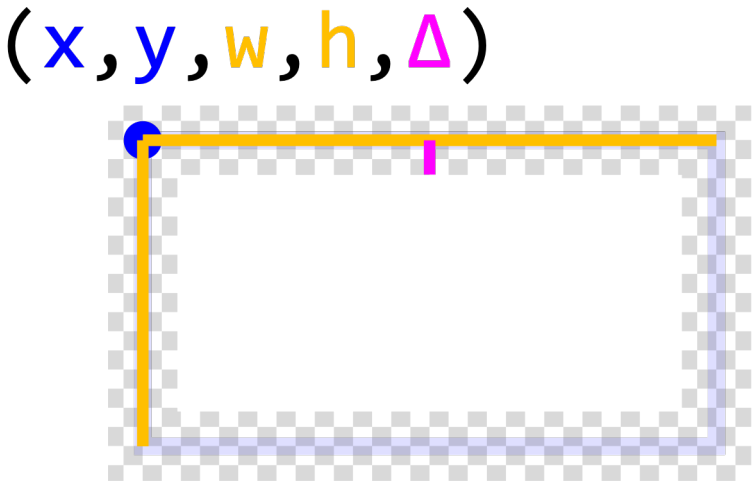
```
if x between x and x + w and
   y between y and y + h
```

```
override fun isHit(x: Double, y: Double): Boolean {
    return x.between(this.x, this.x + w) and
           y.between(this.y, this.y + h)

}
```

Stroke Rectangle Hit-Test

Given:

- Mouse position (x,y)
- Upper-left bound of rectangle (x,y)
- Width and height (w,h)
- Delta Δ



Hit:

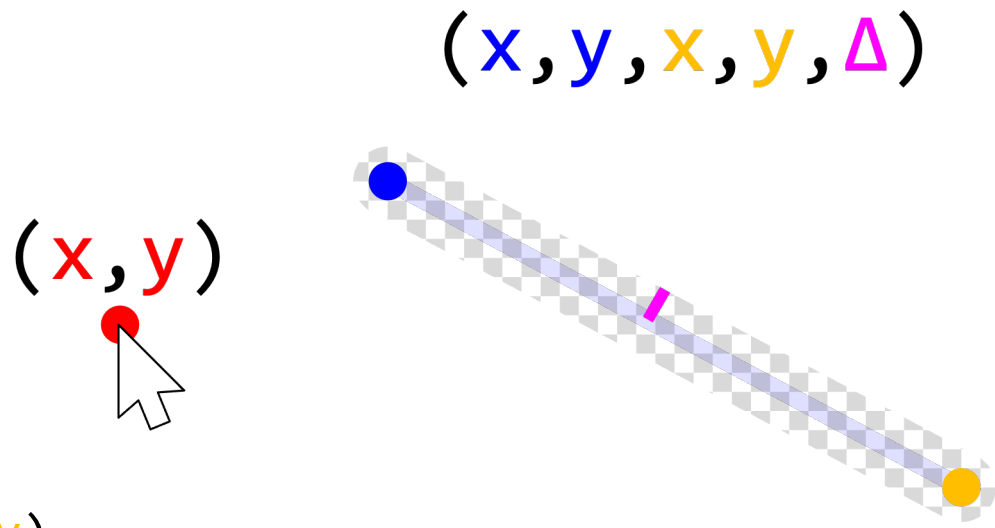
```
if (x,y) is
    inside (x-Δ,y-Δ, x+w+Δ, y+h+Δ) and
    not inside (x+Δ,y+Δ, x+w-Δ, y+h-Δ)
```

```
override fun isHit(x: Double, y: Double): Boolean {
  return (x.between(this.x - delta, this.x + w + delta) and
          x.between(this.x + delta, this.x + w - delta).not() and
          y.between(this.y - delta, this.y + h + delta)) or
         (y.between(this.y - delta, this.y + h + delta) and
          y.between(this.y + delta, this.y + h - delta).not() and
          x.between(this.x - delta, this.x + w + delta))
}
```

Stroke Line Hit-Test

Given:

- Mouse position (x,y)
- Start vertex (x,y)
- End vertex (x,y)
- Delta Δ



Hit:

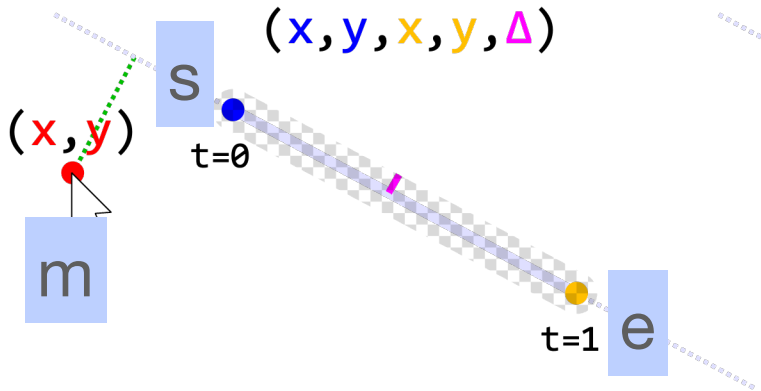
```
if distance
    from (x,y)
    closest point on (x,y, x,y)
    <= Δ
```

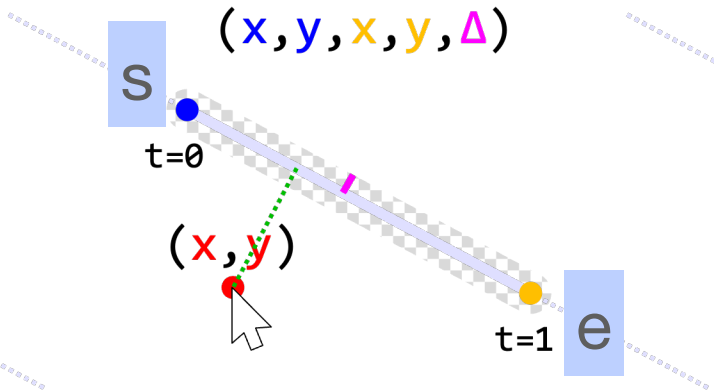
Demo: HitTesting/ClosestPoint

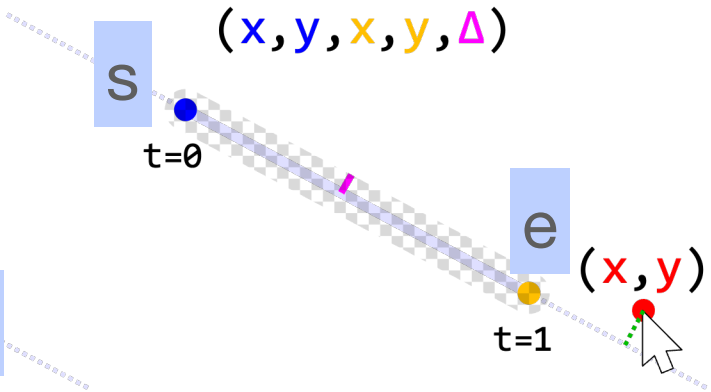
Stroke Line Hit-Test

Hit:

```
if distance
    from m(x,y)
    closest point on line (x,y, x,y)
    <= Δ
```







Closest point on (x,y, x,y) from (x,y):

$$\vec{u} = \vec{m} - \vec{s}$$

$$\vec{v} = \vec{e} - \vec{s}$$

$$t = \frac{\vec{u} \cdot \vec{v}}{\vec{v} \cdot \vec{v}}$$

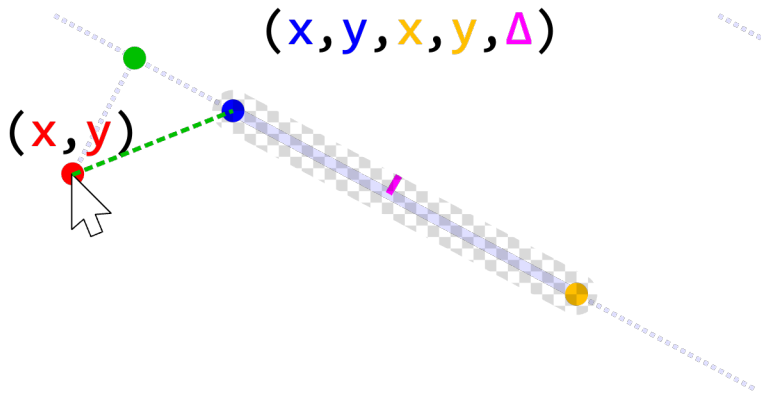
LEGEND
s = start point
e = end point
m = mouse point

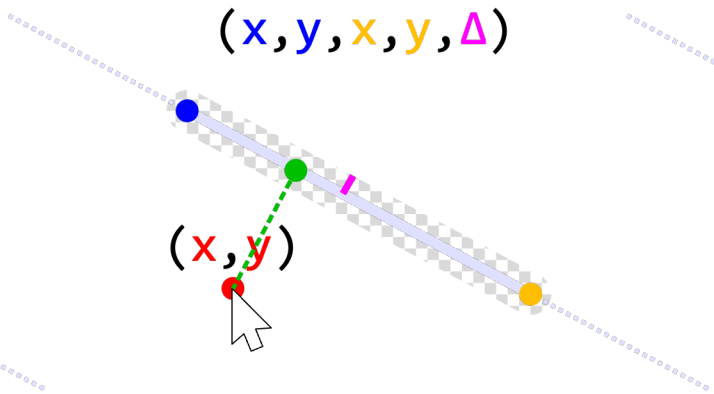
u = vector start-mouse
v = vector start-end

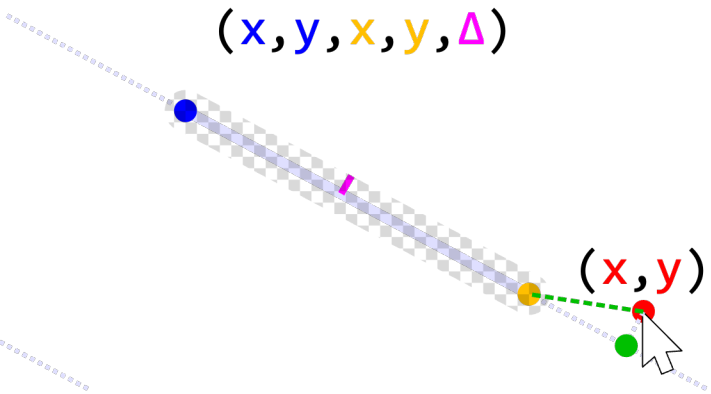
Stroke Line Hit-Test

Hit:

```
if distance
    from (x,y)
    closest point on (x,y, x,y)
    <= Δ
```



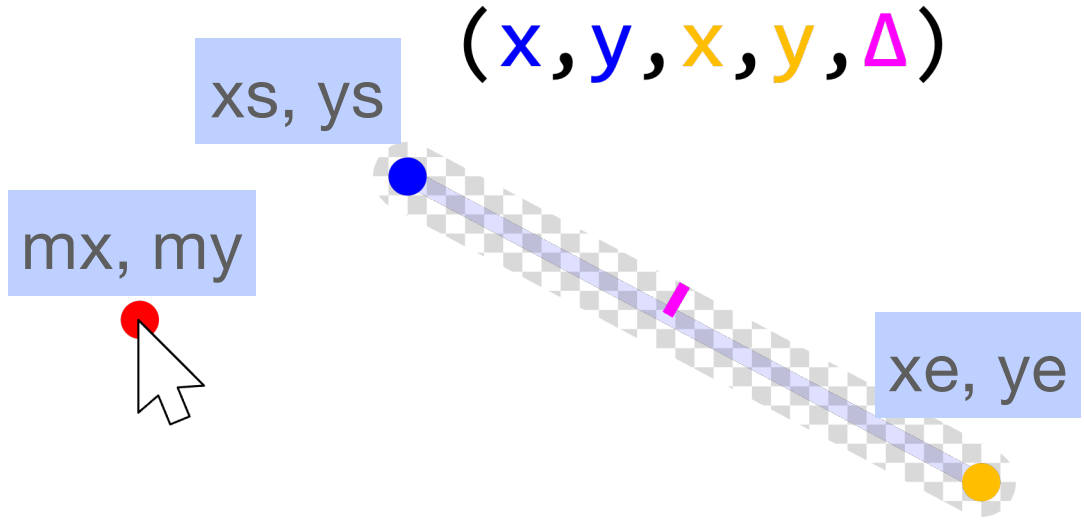




Closest point on (x,y, x,y) from (x,y):

$$= \begin{cases} \vec{s}, & t \le 0 \\ \vec{s} + t\vec{v}, & 0 < t < 1 \\ \vec{e}, & t \ge 1 \end{cases}$$

Stroke Line Hit-Test




```
override fun isHit(xs: Double, ys: Double, xe: Double, ye: Double,
                   mx: Double, my: Double,
                   hitDelta: Double): Boolean {
    val ux = mx - xs
    val uy = my - ys
    val vx = xe - xs
    val vy = ye - ys
    val t = (ux * vx + uy * vy ) / (vx * vx + vy * vy)
    val dst =
        if (t < 0.0) sqrt(sqr(mx - xs) + sqr(my - ys))
        else if (t > 1.0) sqrt(sqr(mx - xe) + sqr(my - ye))
        else sqrt(sqr(mx - (xs + vx * t)) + sqr(my - (ys + vy * t)))
    return dst <= hitDelta
}
```

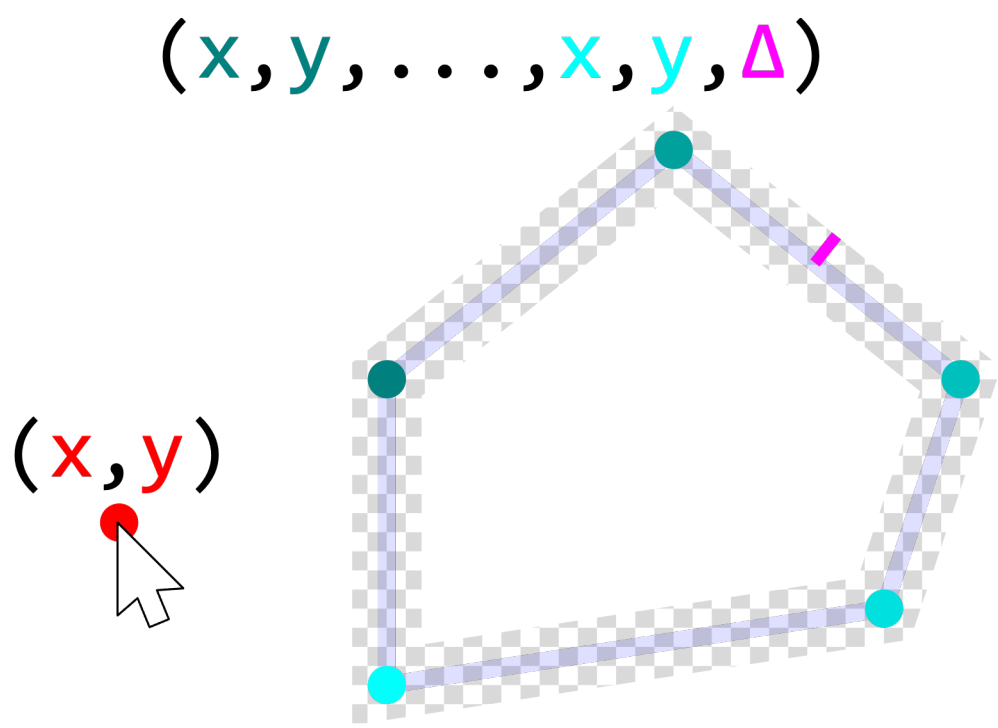
Polyline / Stroke Polygon Hit-Test

Given:

- Mouse position (x,y)
- Vertices (x,y, ..., x,y)
- Delta Δ

Hit:

if any line segment is hit…
We can check each segment

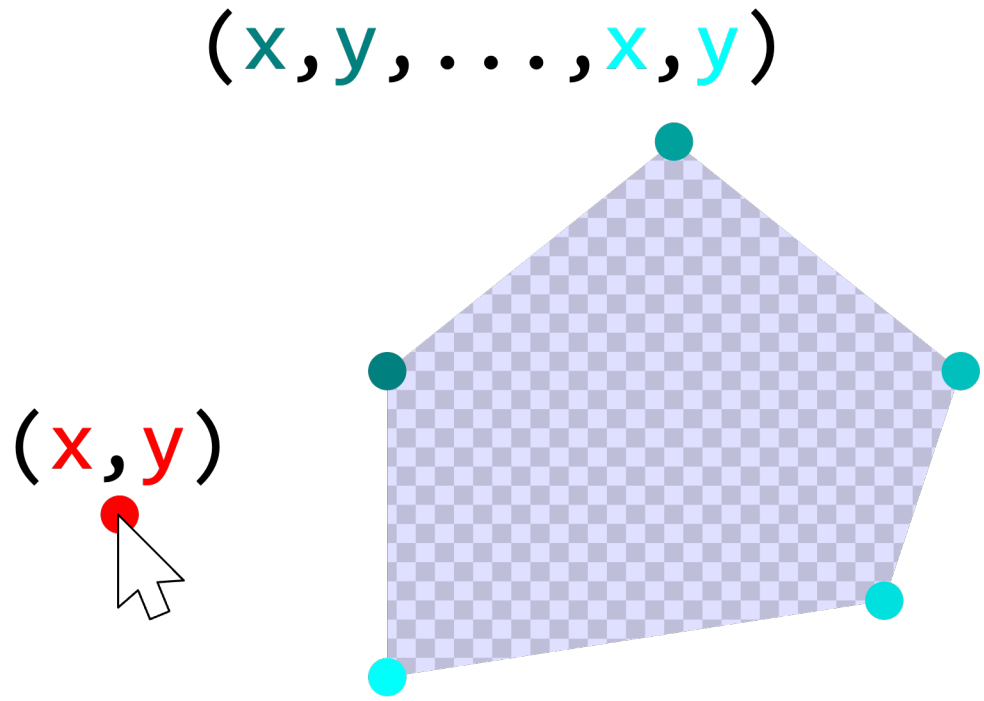


Demo: HitTesting/ShapeModels

Filled Polygon Hit-Test

Given:

- Mouse position (x,y)
- Vertices (x,y, ..., x,y)



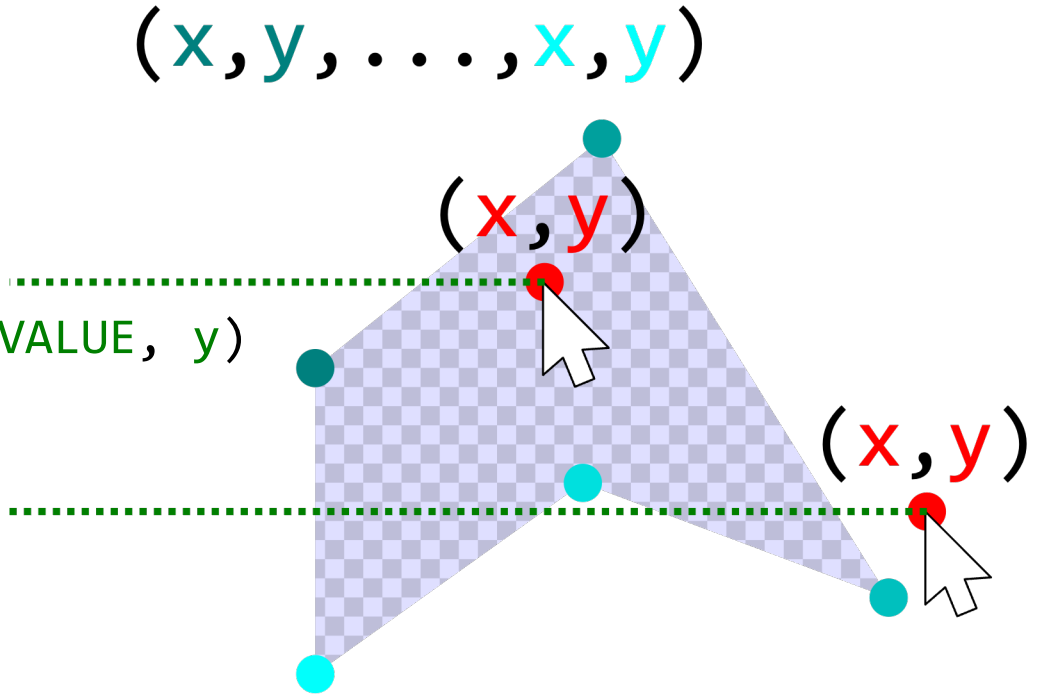
Filled Polygon Hit-Test

Given:

- Mouse position `(x,y)`
- Vertices `(x,y, ..., x,y)`

Hit:

```
if ray cast to (Double.MIN_VALUE, y)
intersects 2k+1 edges (*)
```

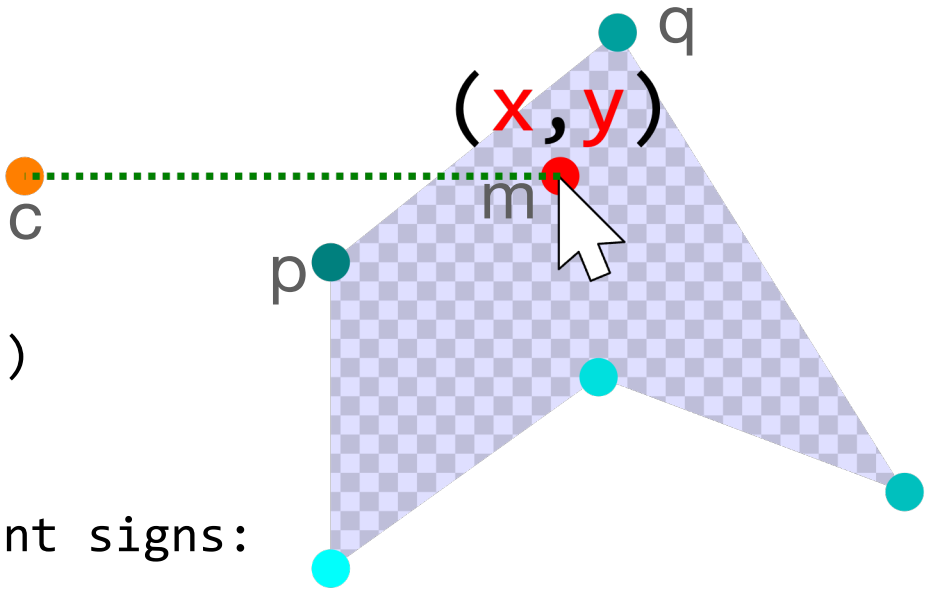


Filled Polygon Hit-Test (Line Test)

Given:

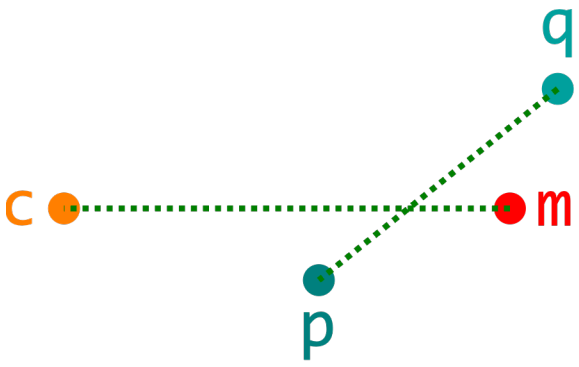
- Mouse position (x,y)
- Vertices (x,y, ..., x,y)

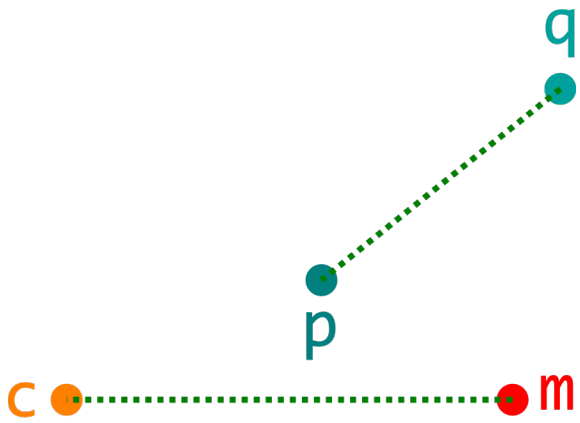
(x,y,...,x,y)



Intersection:

two line segments (m,c), (p,q)
intersect if the orientations
(m,c,p), (m,c,q) and
(p,q,m), (p,q,c) have different signs:





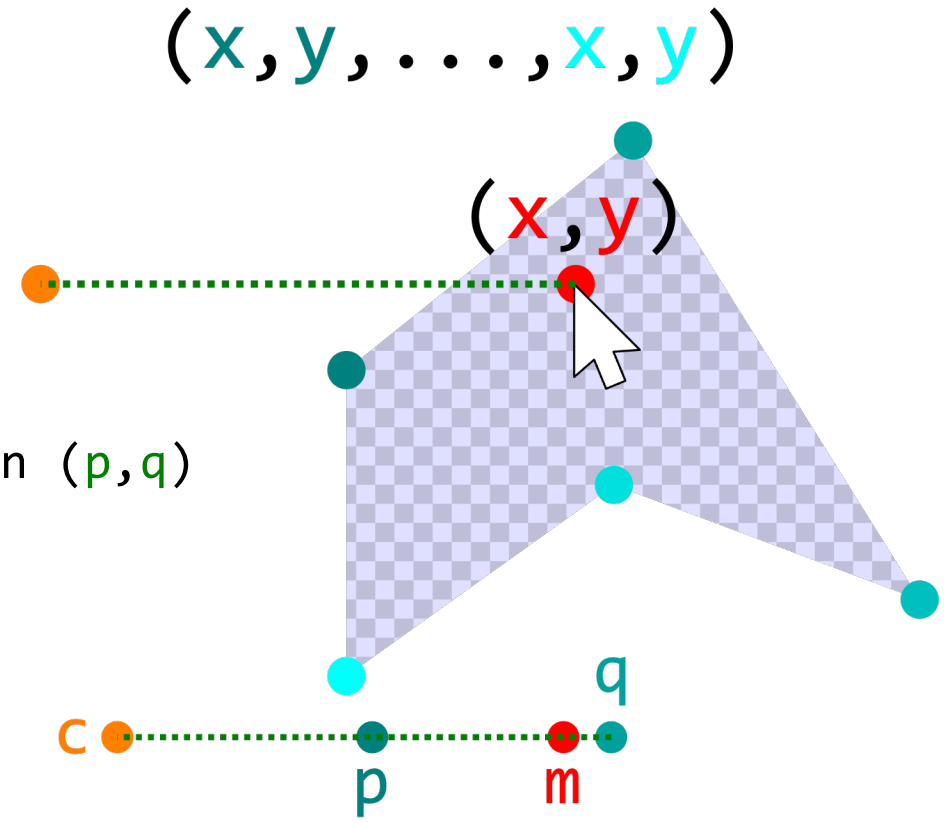
Filled Polygon Hit-Test (Edge Case)

Given:

- Mouse position (x,y)
- Vertices (x,y, ..., x,y)

Intersection:

if (m,c,p), (m,c,q), (p,q,m), and (p,q,c) are co-linear, check if m is on (p,q)



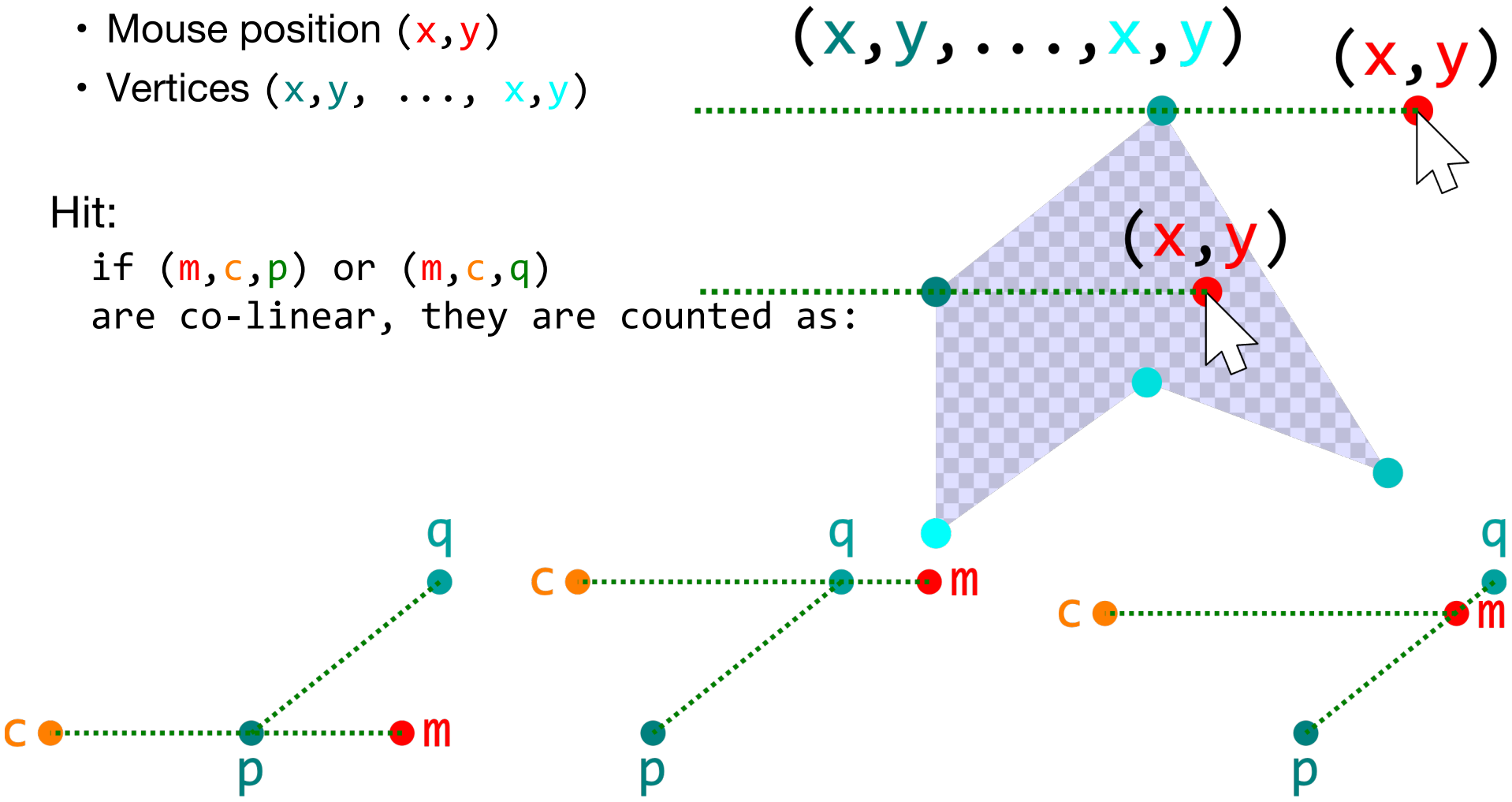
Filled Polygon Hit-Test (Edge Case)

Given:

- Mouse position (x,y)
- Vertices (x,y, ..., x,y)

Hit:

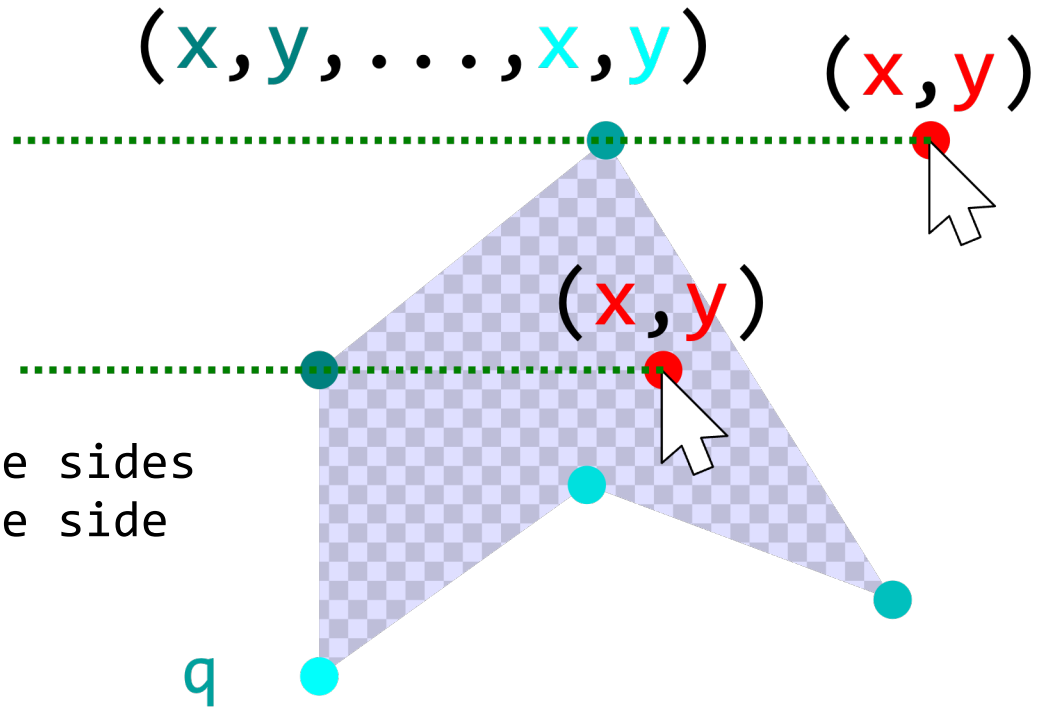
```
if (m,c,p) or (m,c,q)
are co-linear, they are counted as:
```



Filled Polygon Hit-Test (Edge Case)

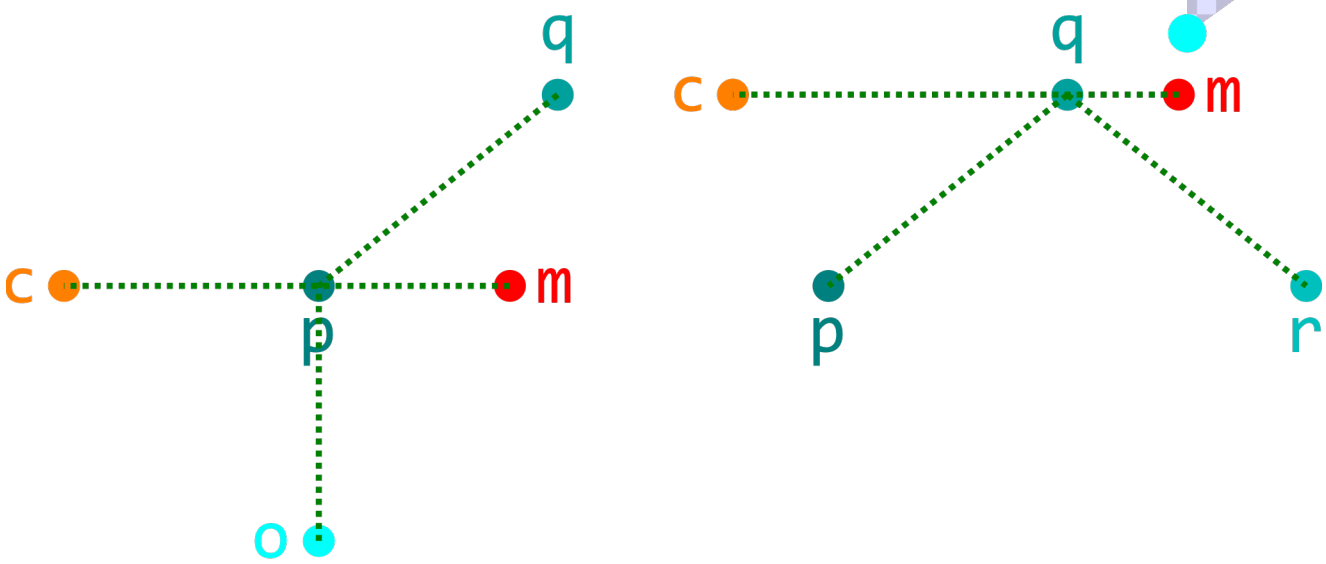
Given:

- Mouse position (x,y)
- Vertices (x,y, ..., x,y)



Hit:

```
+ 1 if edges are on opposite sides
+ 0 if edges are on the same side
```



Optimizations

Hit-testing could become computationally intensive

- There could be hundred of shapes in a scene
- Polygon or Polyline shapes could have hundreds of edges

Approaches to reduce hit-testing computation:

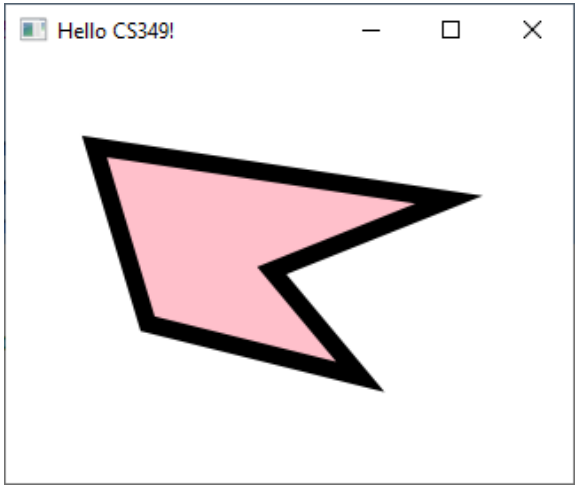
- Avoid `sqrt` in distance calculations
  (for circles, check if `sqr(dist)` is less than `sqr(diameter / 2.0)`)
- Use simpler less precise hit-test first for an “early” reject
  (e.g., start with a bounding-rectangle, or bounding circle hit-test)
- Split scene into cells, and track which ones each shape is in (e.g., octree or binary space partition)

JavaFX Shape Hit-Testing

All JavaFX Shapes implement a `contains` to hit-test against a point:

```
val poly = Polygon().apply {
    fill = Color.PINK
    points.addAll( 50.0,  50.0, 250.0, 80.0, 150.0, 120.0,
                   200.0, 180.0,  80.0, 150.0)
    strokeWidth = 10.0
    stroke = Color.BLACK
}
val scene = Scene(Group(poly), 320.0, 240.0).apply {
    scene.addEventFilter(MouseEvent.MOUSE_MOVED) {
        println(poly.contains(it.sceneX, it.sceneY))
}}
```

It handles stroke thickness (hit if point is on visible stroke) and unfilled shapes (true if point is on visible stroke area).



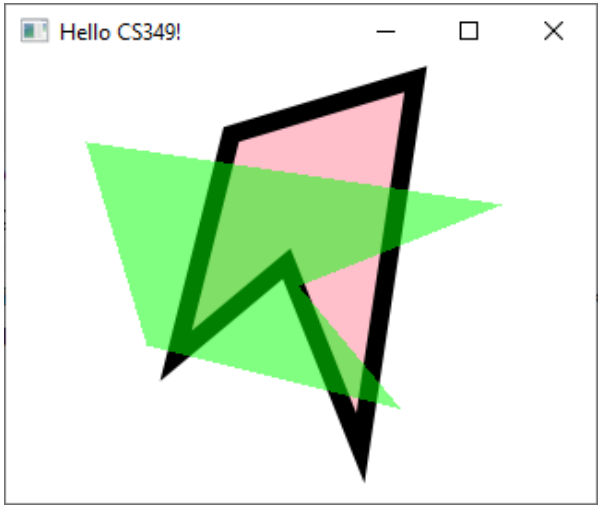
Demo: HitTesting/Shape

JavaFX Shape Hit-Testing

All JavaFX Shapes implement a `contains` to hit-test against a point:

```
val poly = Polygon().apply {
    fill = Color.PINK
    points.addAll( 50.0,  50.0, 250.0, 80.0, 150.0, 120.0,
                   200.0, 180.0,  80.0, 150.0)
    strokeWidth = 10.0
    stroke = Color.BLACK
    rotate = 90.0
}
val scene = Scene(Group(poly), 320.0, 240.0).apply {
    scene.addEventFilter(MouseEvent.MOUSE_MOVED) {
        println(poly.contains(it.sceneX, it.sceneY))
}}
```

It does not handle transformations! (Instead, use event handling in Polygon directly.)



End of Chapter

Any further questions?