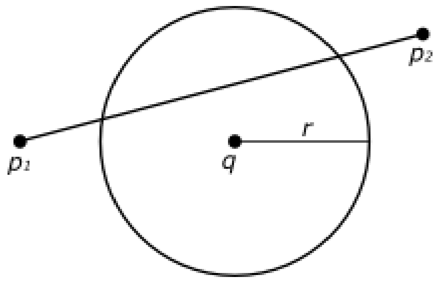
1 Answer

14

In computer geometry, **always use vectors** if possible! Code gets more complicated if you try to work with Cartesian co-ordinates (x, y) or with line equations $y = mx + b$. Here, for example, you have special cases for horizontal lines, $m = 0$, and vertical lines, $m = \infty$.

So let's try to program this, sticking to vectors throughout.

First, let's review the problem. We have a line segment from `p1.p` to `p2.p` and we want to find the points of intersection with a circle centred at `self.p` and radius `self.r`. I'm going to write these as p_1, p_2, q, and r respectively:



Any point on the line segment can be written $p_1 + t(p_2 - p_1)$ for a scalar parameter t between 0 and 1. We'll be using $p_2 - p_1$ often, so let's write $v = p_2 - p_1$.

Let's set this up in Python. I'm assuming that all the points are `pygame.math.Vector2` objects, so that we can add them and take dot products and so on. I'm also assuming that we're using Python 3, so that division returns a `float`. If you're using Python 2, then you'll need:

```
from __future__ import division
```

I'm going to use capital letters for vectors and lower case for scalars:

```
Q = self.p                  # Centre of circle
r = self.r                  # Radius of circle
P1 = constraint.point1      # Start of line segment
V = constraint.point2 - P1  # Vector along line segment
```

Now, a point x is on the circle if its distance from the centre of the circle is equal to the circle's radius, that is, if

$$|x - q| = r.$$

So the line intersects the circle when

$$|p_1 + tv - q| = r.$$

Squaring both sides gives

$$|p_1 + tv - q|^2 = r^2,$$

Expanding the dot product and collecting powers of t gives

$$t^2(v \cdot v) + 2t(v \cdot (p_1 - q)) + (p_1 \cdot p_1 + q \cdot q - 2p_1 \cdot q - r^2) = 0$$

which is a quadratic equation in t with coefficients

$$\begin{aligned} a &= v \cdot v \\ b &= 2(v \cdot (p_1 - q)) \\ c &= p_1 \cdot p_1 + q \cdot q - 2p_1 \cdot q - r^2 \end{aligned}$$

and solutions

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Let's compute the coefficients in Python:

```
a = V.dot(V)
b = 2 * V.dot(P1 - Q)
c = P1.dot(P1) + Q.dot(Q) - 2 * P1.dot(Q) - r**2
```

The value $b^2 - 4ac$ inside the square root is known as the *discriminant*. If this is negative, then there are no real solutions to the quadratic equation; that means that the line misses the circle entirely.

```
disc = b**2 - 4 * a * c
if disc < 0:
    return False, None
```

Otherwise, let's call the two solutions t_1 and t_2.

```
sqrt_disc = math.sqrt(disc)
t1 = (-b + sqrt_disc) / (2 * a)
t2 = (-b - sqrt_disc) / (2 * a)
```

If neither of these is between 0 and 1, then the line segment misses the circle (but would hit it if extended):

```
if not (0 <= t1 <= 1 or 0 <= t2 <= 1):
    return False, None
```

It's not clear to me from your code exactly what you want to return in the case where there is an intersection, but it looks as if you want the closest point on the line segment to the centre of the circle. (Can you explain the geometric significance of this?)

Now, the closest point on the extended line to the centre of the circle is $p_1 + tv$ where

$$t = \frac{(q - p_1) \cdot v}{} = \frac{-b}{}.$$

See Wikipedia for an explanation. But we want to ensure that the point is on the line segment, so we must clamp t to lie between 0 and 1.

```
t = max(0, min(1, - b / (2 * a)))
return True, P1 + t * V
```

Notes

1. I've changed the `return` statements so that instead of sometimes returning `False` and sometimes returning a point of intersection, the function always returns a tuple whose first element is a Boolean indicating whether there was an intersection, and whose second element is the point of intersection. When a function always returns the same kind of data, it's less likely that the caller will make a mistake in handling it.
2. I haven't tested any of the code in this answer (since I don't have a version of PyGame that supports `pygame.math.Vector2`). So there might be a bug or two. But here's a JavaScript demo I wrote using the technique described here.

edited Oct 17 '15 at 12:31

answered Apr 9 '15 at 17:56

Gareth Rees
47.4k 3 113 199

Thanks for the answer (and concise explanation). Your code pretty much functioned as you said expect that it was returning either point 1 or point 2 upon collision. What I meant by 'the closest point' was the closest point that lay on the line segment. The value of t for this solution is essentially the midpoint of the two scalar parameters: (t2+t1)/2 the significance of this is mainly that the output of the function is used to separate (constrain) the circle and line segment by moving them away from each other such that the line segment is at a perfect tangent to the circle. – Tochi Obudulu Apr 9 '15 at 20:59

@TochiObudulu: I made a sign error in the final calculation of t: it needs to be $\frac{-b}{2a}$. Now fixed. Sorry about that! – Gareth Rees Apr 9 '15 at 21:16

@GarrethRoss: Ahh, - works fine now. Just out of interest, in my case, would it be less expensive to use (t1+t2)/2 or the clamps that you suggested or does the mid-point approach not always yield the right result? – Tochi Obudulu Apr 9 '15 at 22:14

2 @TochiObudulu: $\frac{t_1+t_2}{2}$ is the same as $\frac{-b}{2a}$: it gives the point on the extended line that's closest to the centre of the circle. But it is not necessarily on the line *segment* itself. If this matters for your application, then you need to clamp it. (I don't know exactly what is involved in "moving them away from each other" so I can't be sure what is the right result.) – Gareth Rees Apr 9 '15 at 22:18