Physics Programming

Advanced Collisions

Slides & lesson materials by Hans Wichman & Paul Bonsma

Image: http://communities.ptc.com/people/DanMarotta

Previous Lectures

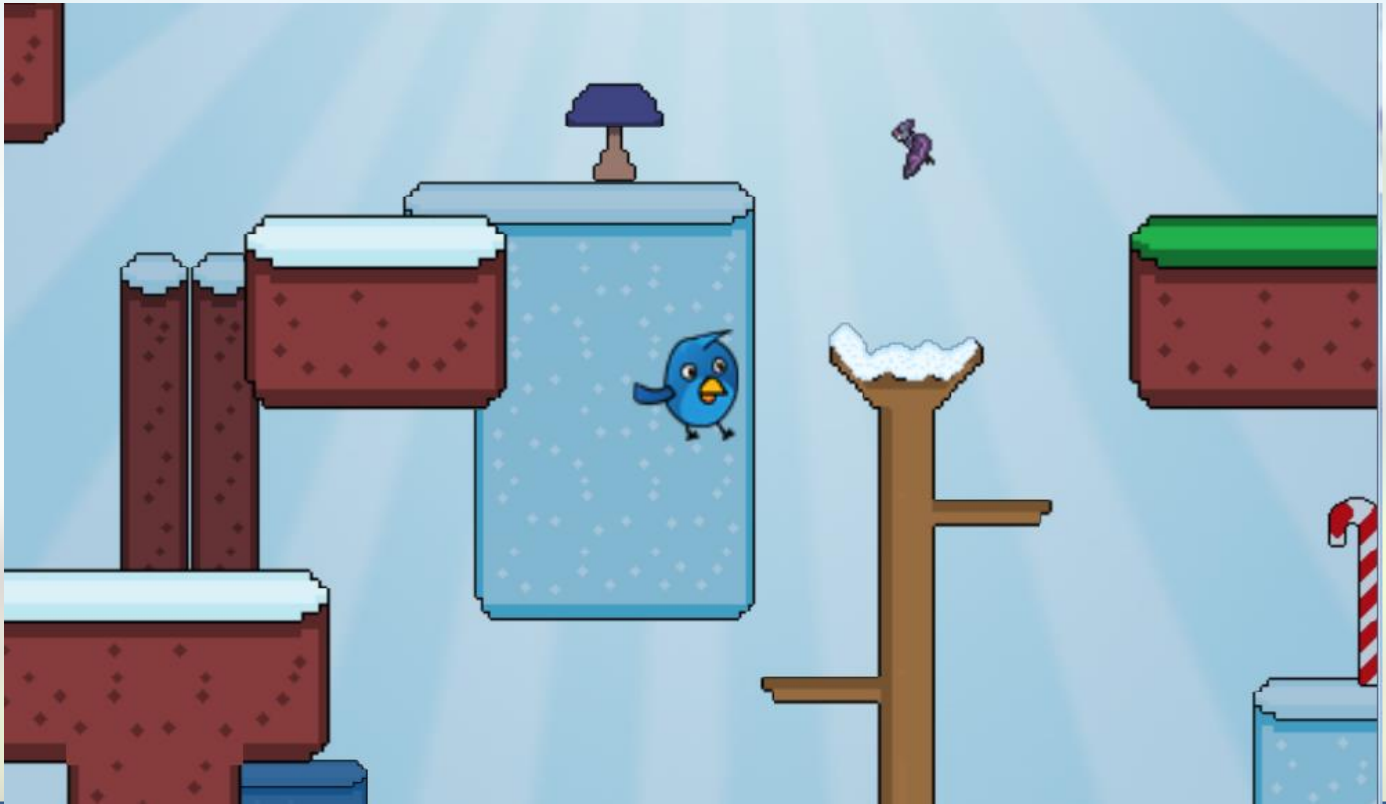
1. Vector basics
2. Trigonometry, angles

Top down shooter / bullet hell game


Previous Lectures

3. Newton's laws, collisions, engine setup. *(Axis-aligned) Block-block collisions*

Platformer

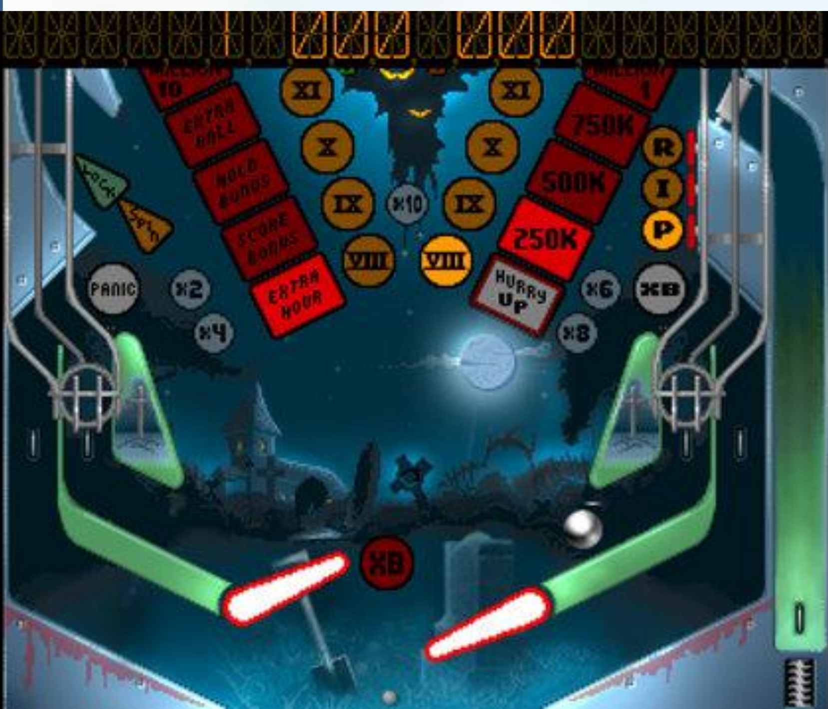


Previous Lectures / this lecture

4. Angled lines, projection, dot product. *Ball-line collisions*

Now: Ball-ball collisions, ball-line segment collisions

➔Pinball / pool!



Line segments + circles: all we need

Lecture overview

- Final course assignment
- Circle / circle collision detection & resolve:
  - Discrete collision detection & resolve
  - Continuous collision detection & resolve
  - Theory vs practice: fixing bugs
- Circle / line **segment** collision detection:
  - Segment detection
  - Bouncing on/off line caps
  - Theory vs practice: fixing bugs
- Assignment 5

Final Course Assignment

Final Assignment

- *Create an interactive physics demo / game that features at least aiming and ball/angled line collisions*

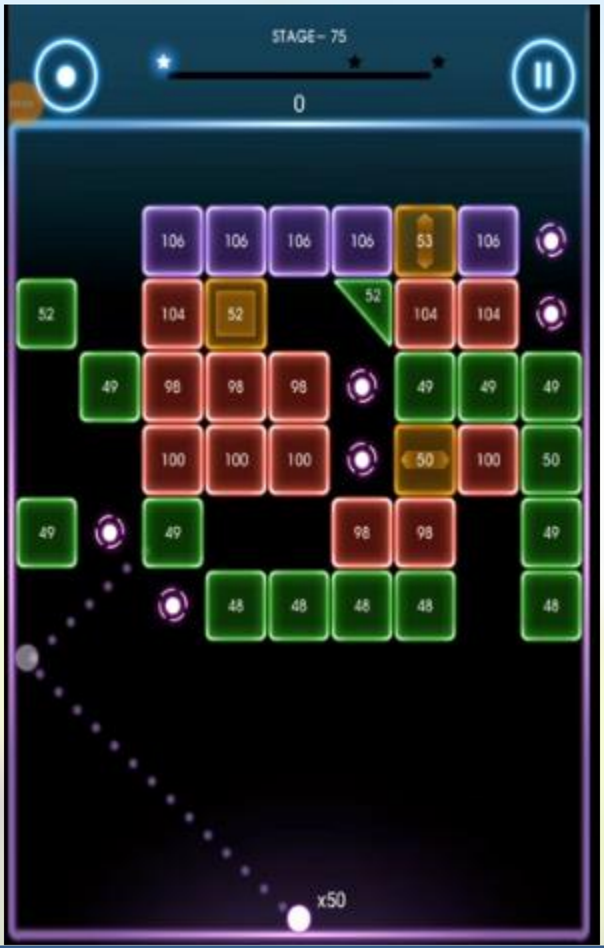
- You have to make one of the game types in the manual (7 different types)
- We don't want everyone to make the same game, so a game type will be assigned to you
- However, during this week's labs, you can give input / indicate preferences

Possible Games

Description:	**Examples:**
A top-down or side-view "bullet hell" shooter	Sky Force Reloaded / Rayman Origins shooter levels
A game where the player has limited control over a bouncing ball and needs to hit targets	Pinball, or a generalization of Breakout / Arkanoid
"Incredible machine": the player has to place items, before running a physics simulation, with the goal of activating a target.	Incredible Machine games
A platformer (The player controls a moving and jumping character, sideview, with gravity)	Sonic the Hedgehog
Aiming and shooting a bouncing ball to break bricks	Bricks Breaker Quest, or pool variants (with angled lines)
A turn-based multiplayer aiming game	Worms / Scorched earth
A top-down racing game	Micro Machines V2

Details / Examples

- Some of these games seem clearer / easier than others…?



→Brick Breaker Quest

- “I can do this!”

→ The collisions, aiming, angled surfaces are clear

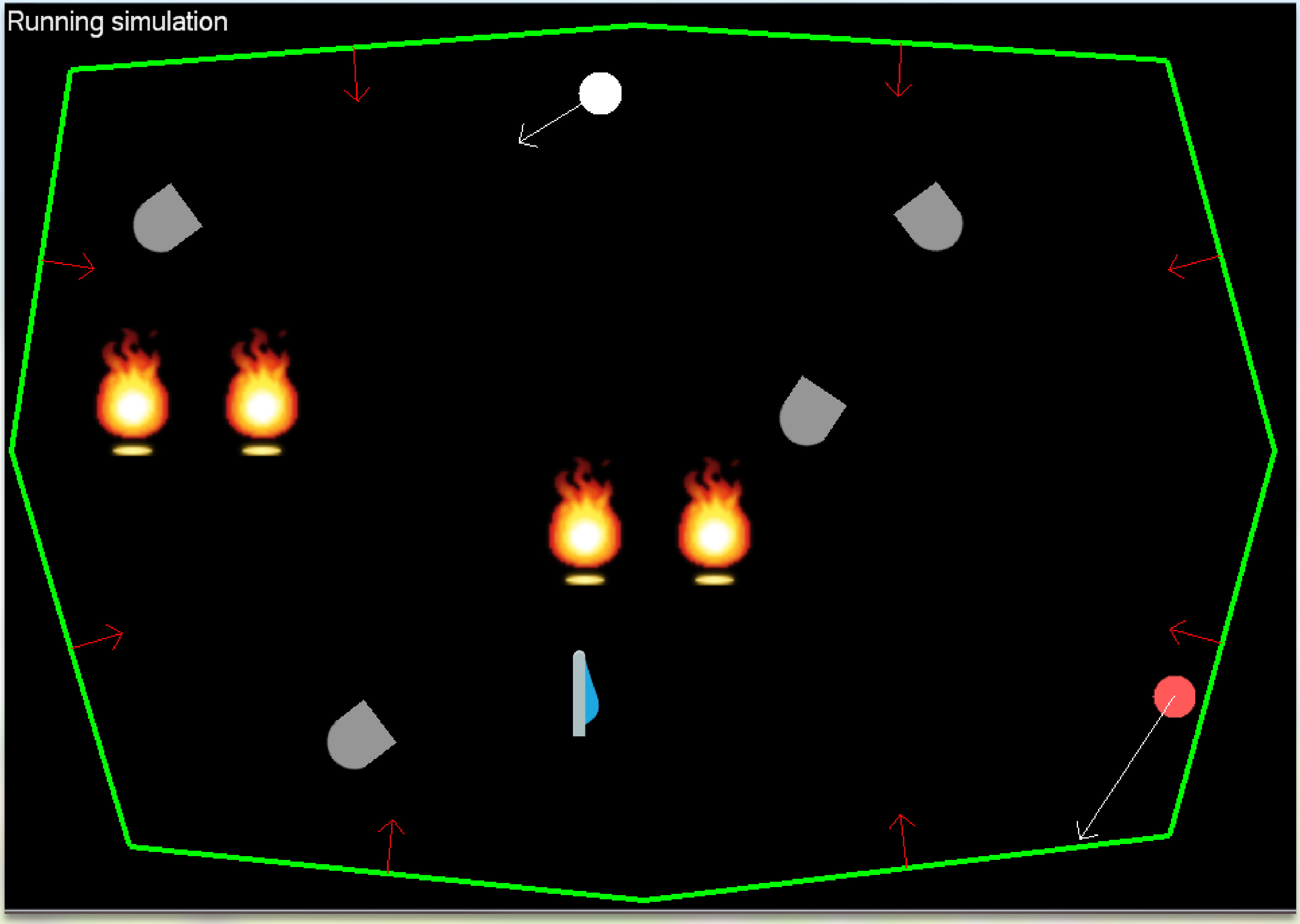
Details / Examples

- Incredible Machine
- “That looks complex!”
- Where exactly are the collisions & aiming?

Final Assignment

- For each game type, we expect similar effort
- So for Incredible Machines, choose core mechanics that are related to the lab assignments
- We only expect a “greybox” → no need to focus on graphics, level loading, game mechanics other than physics.
- Next up: an example implementation for Incredible Machines

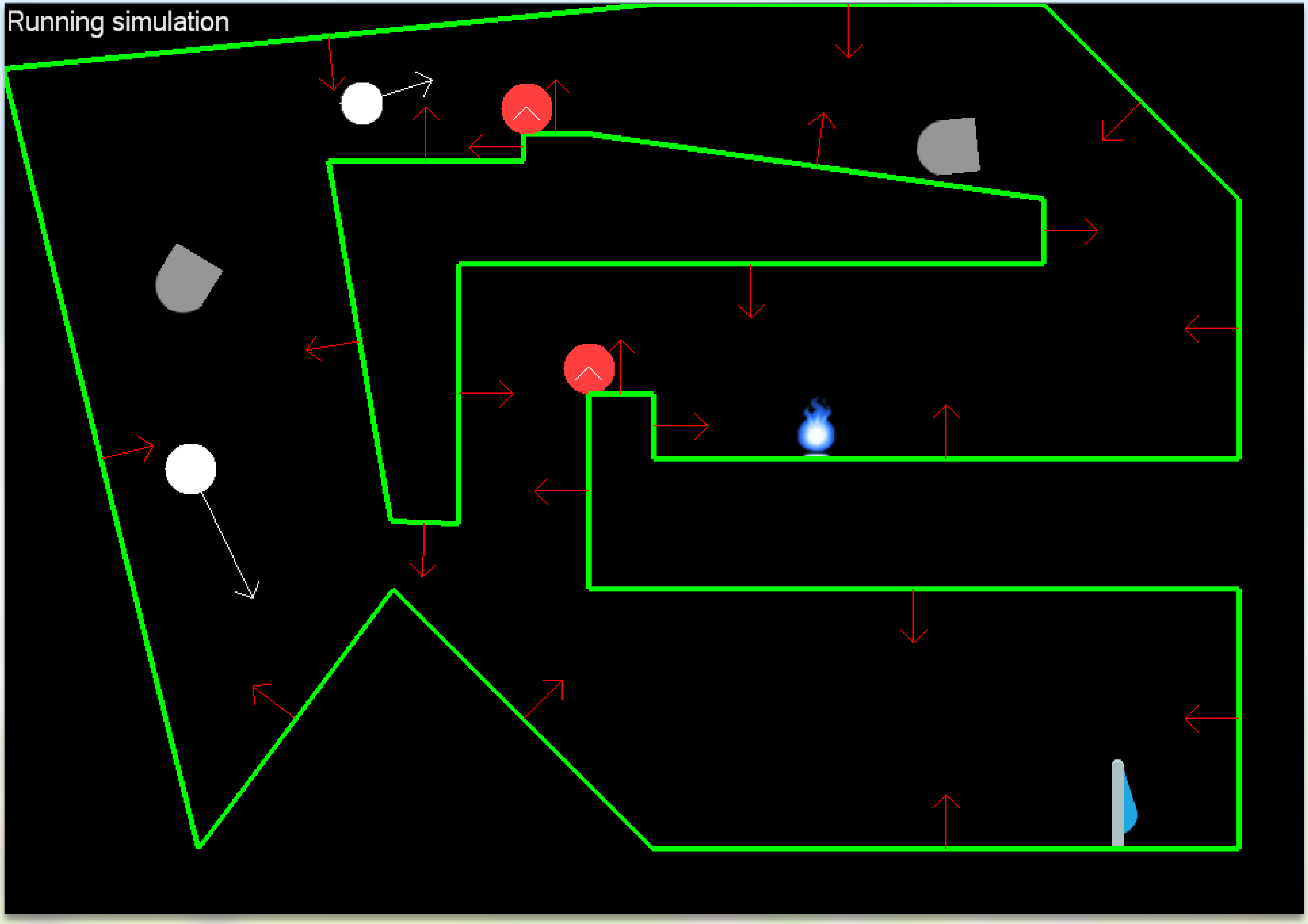
Incredible Machines: Sufficient



Incredible Machines: Sufficient

Aiming and shooting (20%)	**0 pts** Aiming is not implemented correctly or the sprite rotation does not match the movement direction.	**12 pts** Aiming (+ shooting) in the current direction, or aiming to a target is implemented, using a rotated sprite (without using the GXP Engine's methods such as Move)	**16 pts** S + Aiming in the current direction and aiming to a target are both implemented.	**20 pts** G + Advanced aiming functionality has been added. (Examples: leading a moving target, aiming a gravity-influenced projectile, timing fixed angle shots to hit a moving target.)
Collisions (30%)	**0 pts** Collision detection + resolve contains bugs, or is not included for angled lines, or no bouncing is included.	**18 pts** Correct collision detection + resolve (incl. bouncing / velocity reflection for angled lines) without using the GXP Engine's methods such as MoveUntilCollision).	**24 pts** S + Correct point of impact calculation, correct collision with line segments (without using the GXP Engine's methods such as MoveUntilCollision).	**30 pts** G + Robust handling of advanced collisions (Examples: multiple moving objects following Newton's laws, combining gravity with sliding or rolling, kinematic objects such as moving platforms, or collision friction)
Extra Physics Functionality (15%)	**0 pts** -	**9 pts** Given	**12 pts** Original physics-based mechanics (e.g. magnets, explosion force, wind force, thrusters)	**15 pts** Original and advanced physics-based mechanics (e.g. ropes, floating on water, rotating non-circular objects)

Incredible Machines: Good/Excellent



Incredible Machines: Good/Excellent

Aiming and shooting (20%)	**0 pts** Aiming is not implemented correctly or the sprite rotation does not match the movement direction.	**12 pts** Aiming (+ shooting) in the current direction, or aiming to a target is implemented, using a rotated sprite (without using the GXP Engine's methods such as Move)	**16 pts** S + Aiming in the current direction and aiming [illegible] a target are both imple[illegible]d.	**20 pts** G + Advanced aiming functionality has been added. (Examples: leading a moving target, aiming a gravity-influenced projectile, timing fixed angle shots to hit a moving target.)
Collisions (30%)	**0 pts** Collision detection + resolve contains bugs, or is not included for angled lines, or no bouncing is included.	**18 pts** Correct collision detection + resolve (incl. bouncing / velocity reflection) on angled lines (without using the GXP Engine's methods such as MoveUntilCollision).	**24 pts** S + Correct point of impact calculation, correct collision with line segments (without using the GXP Engine's methods such as MoveUntilCollision).	**30 pts** G + Robust handling of advanced collisions (Examples: m[illegible]e moving obje[illegible]s follo[illegible] Newton's la[illegible]ng gravity with sliding or [illegible]g, kinematic objects such as moving platforms, or collision friction)
Extra Physics Functionality (15%)	**0 pts** -	**9 pts** Given	**12 pts** Original physics-based mechanics (e.g. ma[illegible]ts, explosio[illegible] force [illegible]orce, thrust[illegible]	**15 pts** Original and advanced physics-based mechanics (e.g. ropes, floating on water, rotating non-circular objects)

Ball / ball collisions, Newton, line segments

Explosion force

Final Assignment - Approach

- Step 1: Make sure your ‘physics engine’ works correctly, and is tested thoroughly!
  - This is what the lab assignments are about!
  - Especially Assignment 4 and 5 provide good starting points for the final assignment
- Step 2: Implement the desired game play in your physics engine

Todays Topics

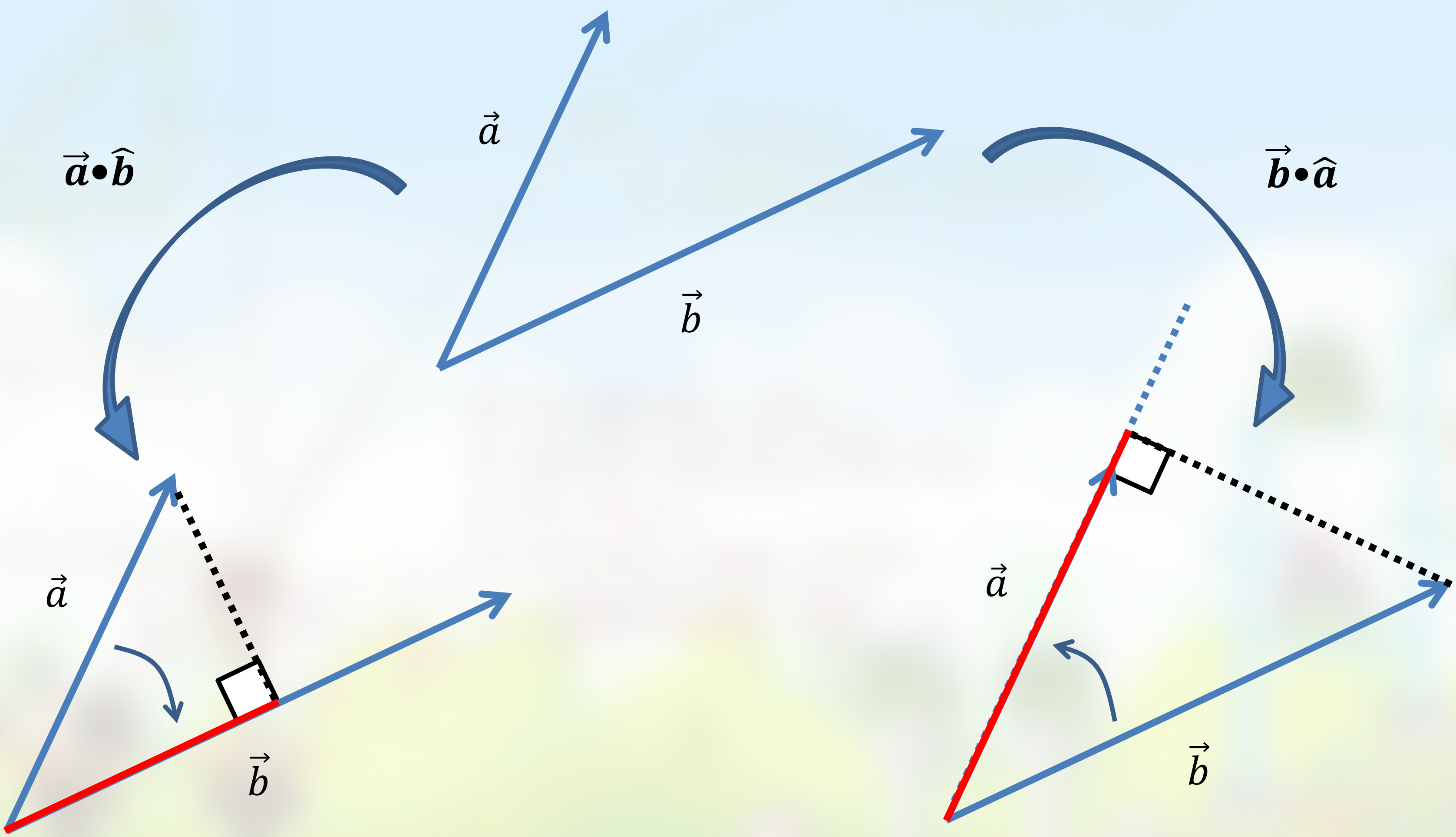
- Except for the first part (discrete circle-circle collisions), the topics of this lecture / this weeks assignment are a bit more complex
  - You can score `sufficient' on collisions without this
  - …but it is necessary to score good/excellent, or actually to create interesting game play
  - (Without it, you really need to design your game around what you can do…)

Dot Product Recap

Dot Product recap

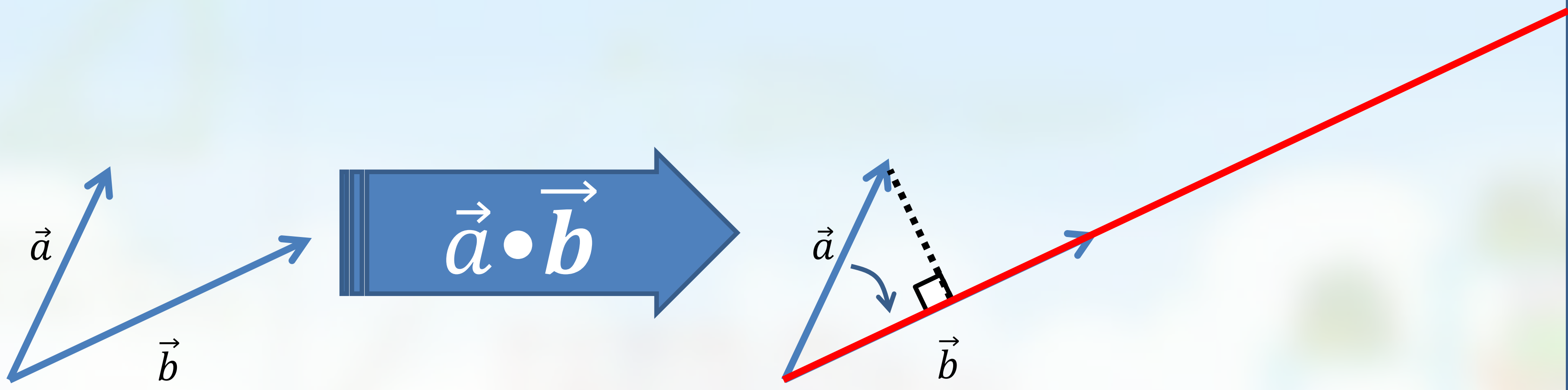
- We will make heavy use of previous lectures, in particular the *dot product* $\vec{\boldsymbol{a}} \bullet \vec{\boldsymbol{b}}$. Recap:
  - $\vec{\boldsymbol{a}} \bullet \vec{\boldsymbol{b}} = |\vec{a}||\vec{b}|\, \boldsymbol{cos}(\boldsymbol{\alpha})$ ($\boldsymbol{\alpha}$ is angle between $\vec{a}, \vec{b}$)
  - $\vec{\boldsymbol{a}} \bullet \vec{\boldsymbol{b}}$ is *positive* if $\boldsymbol{\alpha} < 90$ degrees
  - $\vec{\boldsymbol{a}} \bullet \widehat{\boldsymbol{b}}$ gives the scalar projection of $\vec{\boldsymbol{a}}$ onto $\vec{\boldsymbol{b}}$.

Dot product recap - visual



Don't forget to normalize $\vec{\boldsymbol{b}}$!!

Otherwise the projection is $|\vec{\boldsymbol{b}}|$ times too big !



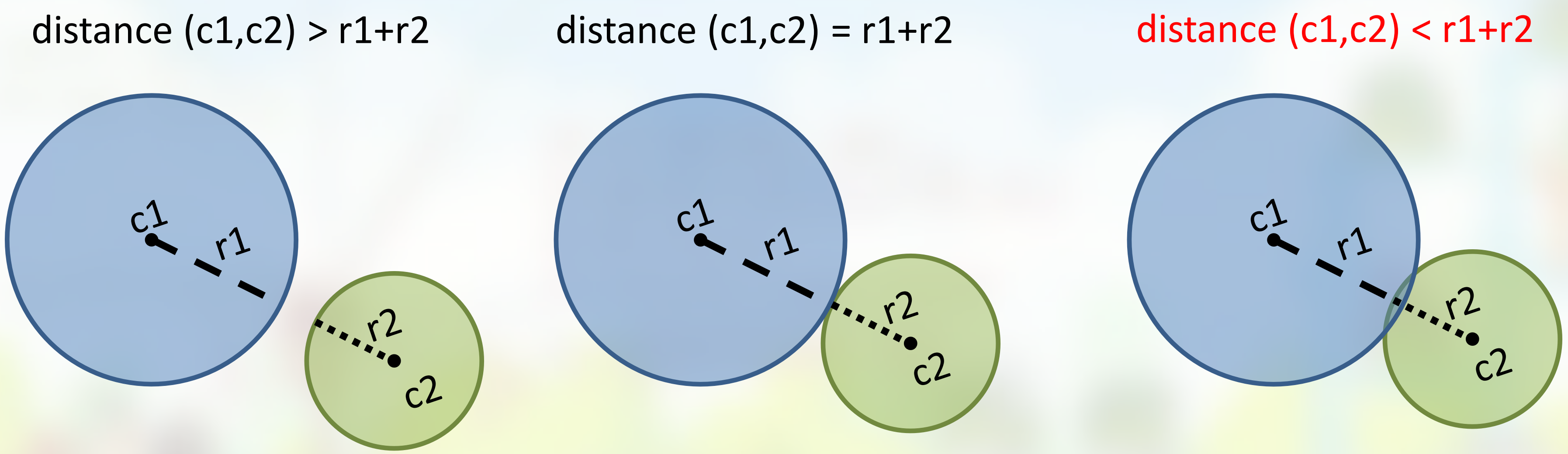
Circle / circle collision - discrete

Circle-circle collision resolving

- The same steps apply as to all other collisions:
  - detect a collision
  - resolve a collision:
    - position reset
    - reflect the velocity using the *collision normal*

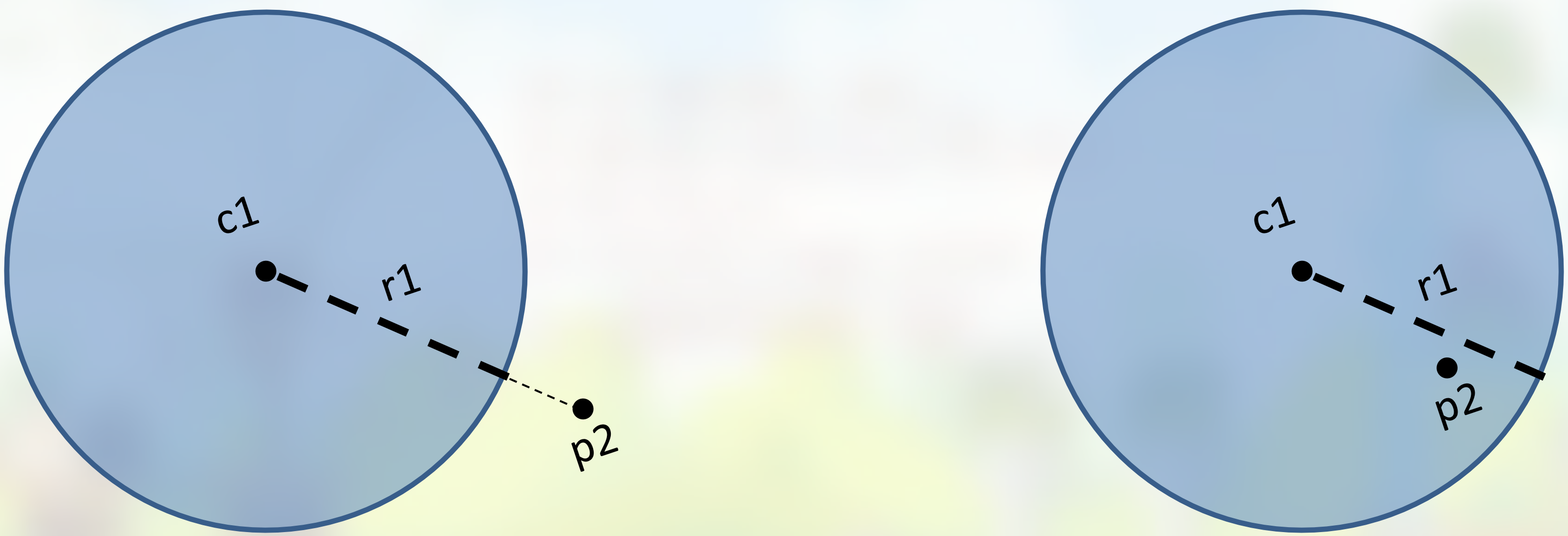
Circle circle collisions

- Two circles are colliding if the distance between their centers is less than their combined radii:
  - if (distance (c1,c2) < r1+r2) then collision



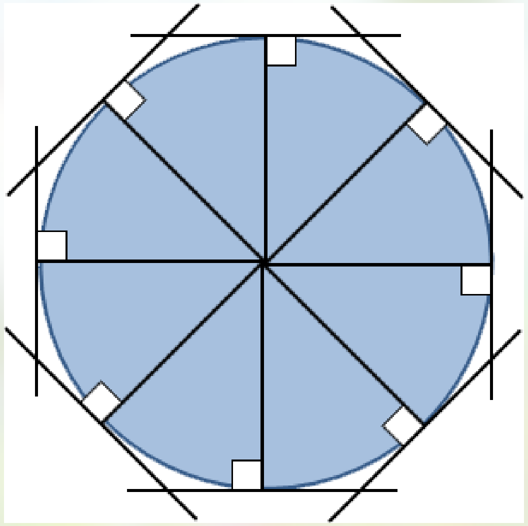
Special case: Circle-point collisions

- A circle and a point "collide" if the distance between the circle center and the point is less than the radius of the circle:
  - if (distance (c1,p2) < r1) then collision



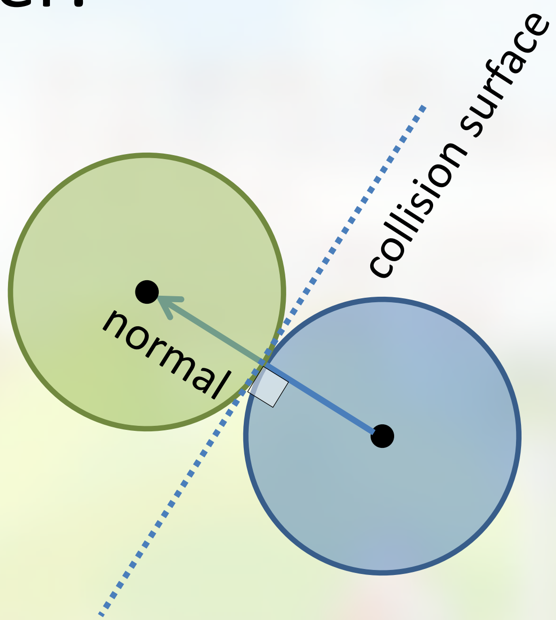
Resolving circle-circle collisions

- Just as before: to resolve a collision we need a line/surface/normal to resolve against…
- Unlike ball-line collision, where a line only has one direction (and thus one normal), a circle defines a lot of directions to resolve against:



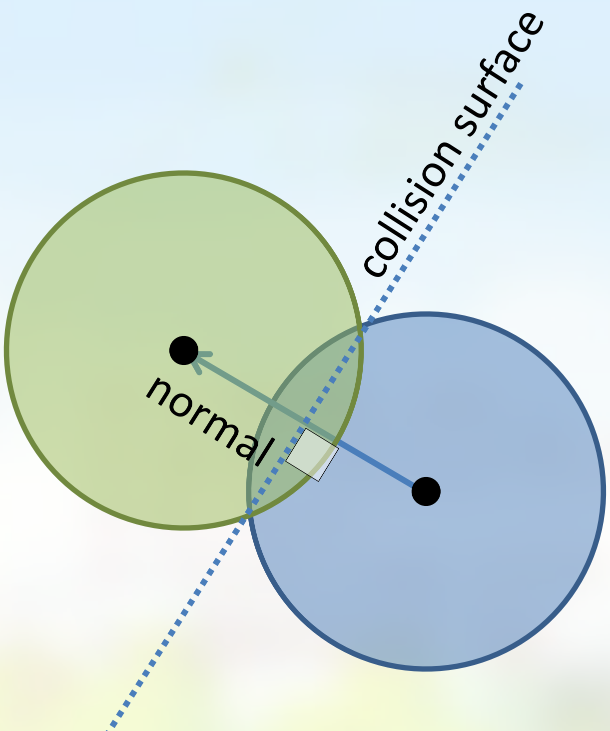
Resolving circle-circle collisions

- The exact collision surface (and normal to resolve against) at the time of impact, is determined by the exact Point of Impact.
- The *collision normal* is the direction vector from center to center:



Resolving circle-circle collisions

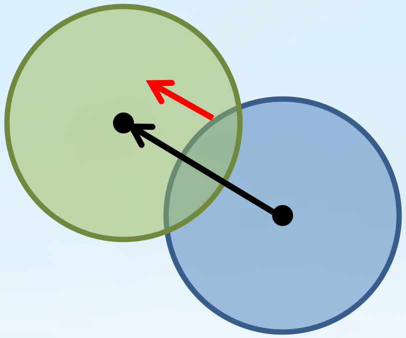
What if we do not have the exact POI?



We can still calculate a normal based on the current overlapping position...

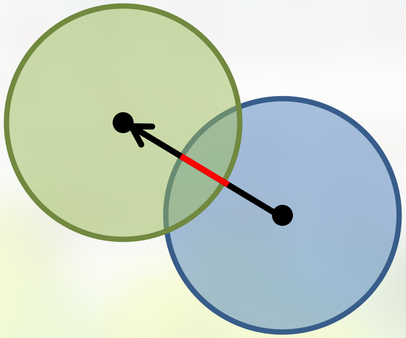
Resolving the position

- Calculate the *normal and unit normal* for the collision:



- Calculate the overlap:

  overlap = r1 + r2 – distance



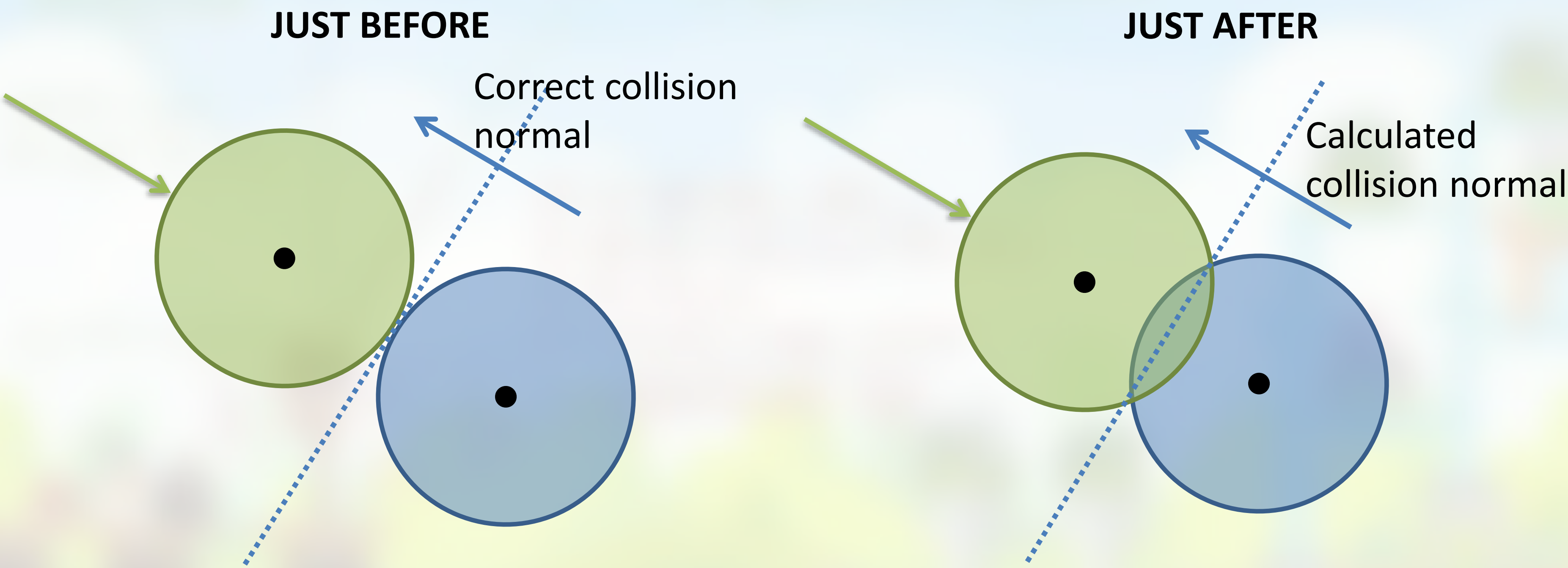
- Move the ball back by: *unit normal * overlap*

  Related example: see 001_simple_ball_ball_collision

Resolving circle-circle collisions

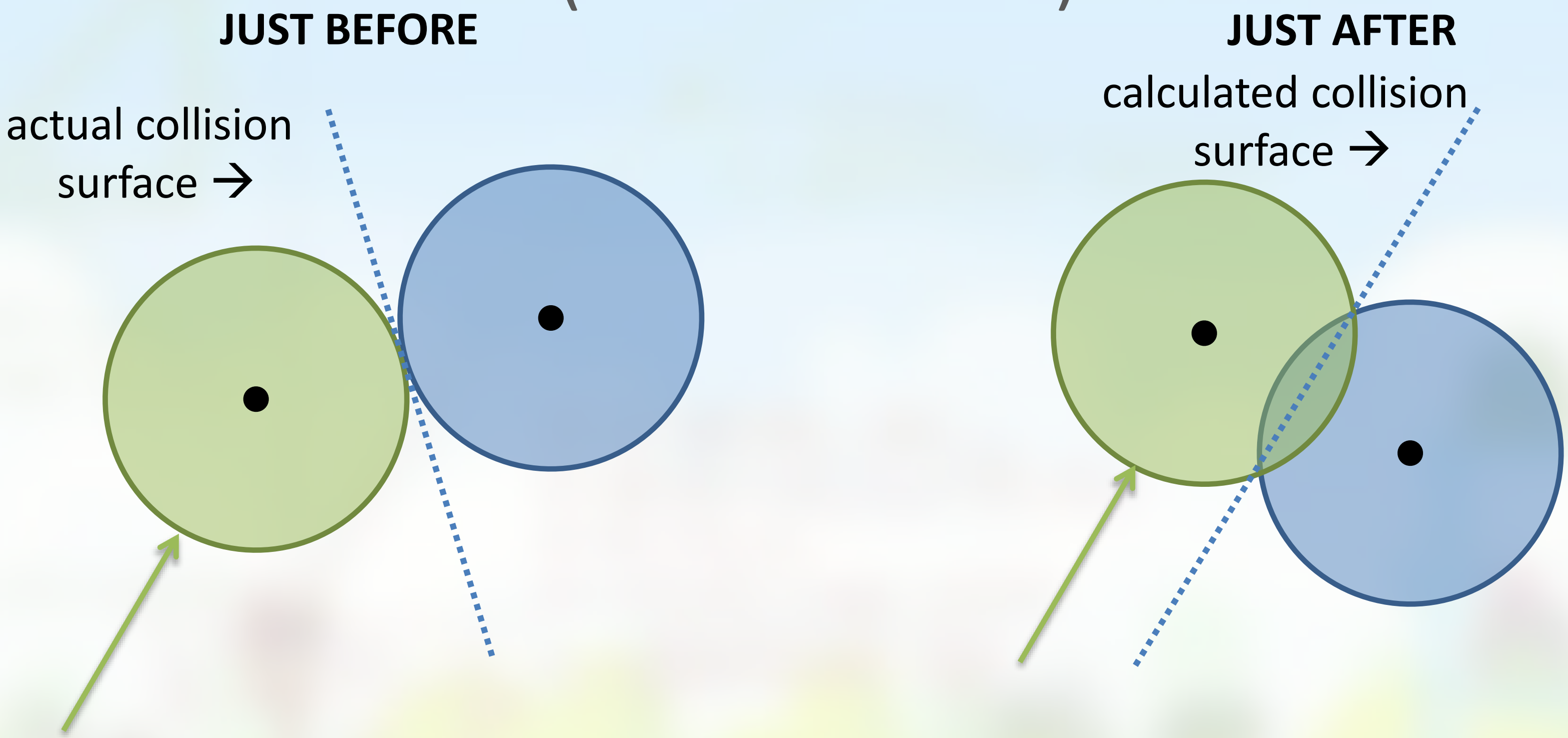
If the balls are moving like this, our normal calculation + collision resolve is accurate:

(demo: scene 0)



Resolving circle-circle collisions

But they might also have been moving like this…
(demo: scene 1)

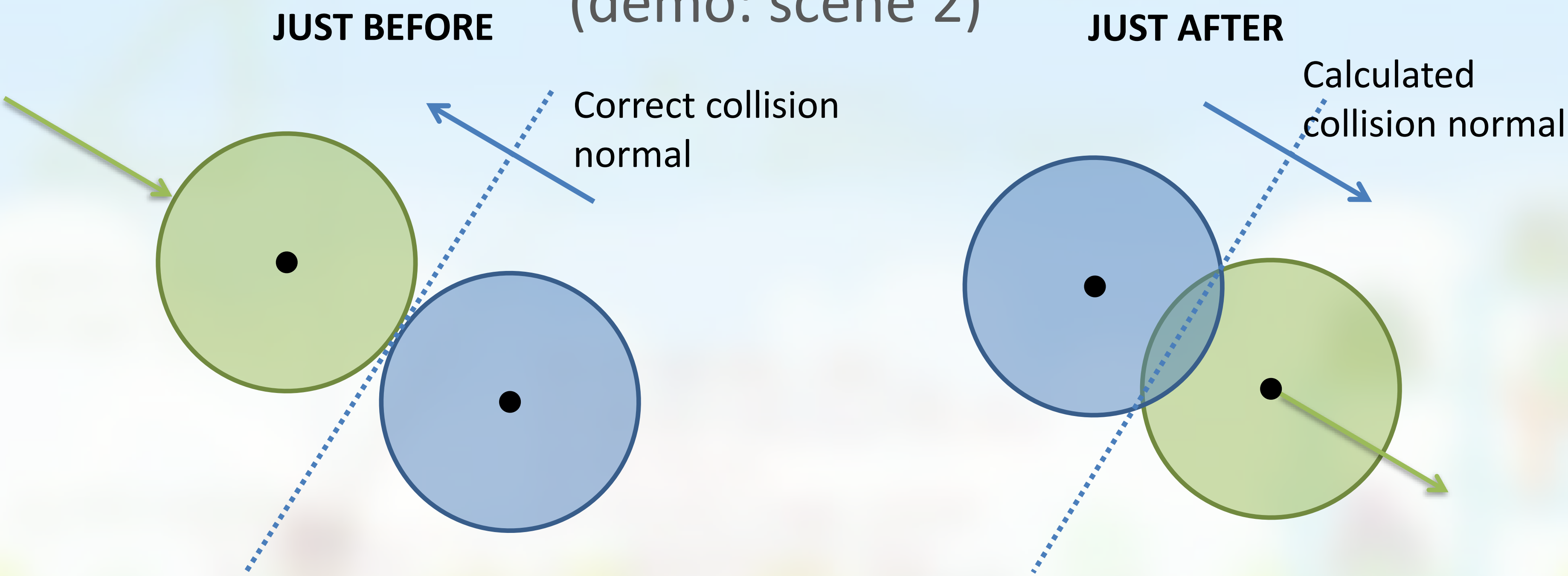


which means the *calculated normal* is wrong, and so will our resolve/reflect math be…

Resolving circle-circle collisions

Worst case scenario…

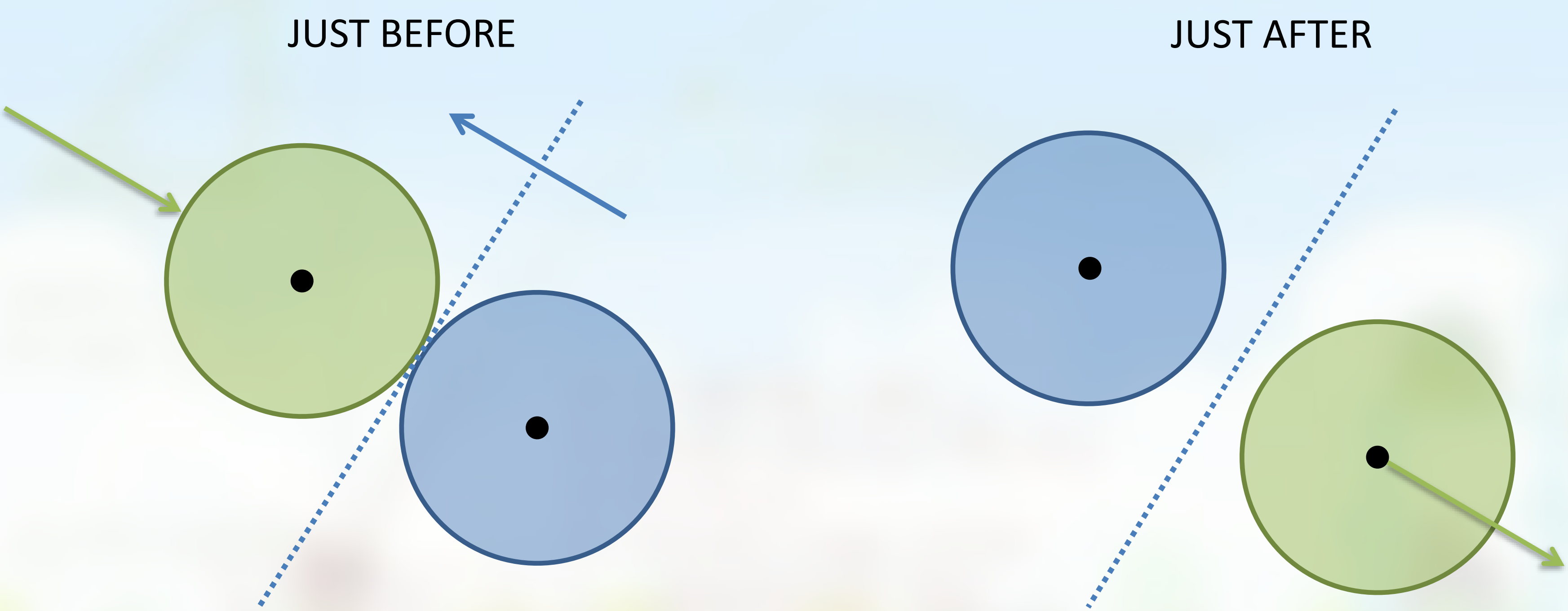
(demo: scene 2)



which means both position and velocity will be resolved *completely opposite* of what they should be!

Resolving circle-circle collisions

And this is also possible using *discrete collision detection:*



which means collision will not be detected at all…

(=tunneling)

Conclusion & Solutions

- Just as with line segments, not knowing the **exact** Point Of Impact might cause us to resolve and reflect the wrong way!
- We might also completely miss collisions (tunneling)
- If velocities are small (compared to radius), this risk is acceptable.
- When using gravity (gives a stack of balls), resolving one collision makes another collision worse → balls passing through each other… (demo: scene 3)
- High velocity / gravity (pool / pinball): we need to do something better…

- Solution:
  - Use *continuous collision detection:* calculate exact Point of Impact (POI).

Circle / circle collision - continuous

Circle / circle POI computation

- Computing the exact POI requires solving a *quadratic equation*
- For that we need a high school math recap (?)… (proof omitted, but see https://www.mathsisfun.com/algebra/quadratic-equation-derivation.html)

abc formula

If you have a quadratic equation of the form

$$ax^2 + bx + c = 0$$

and you want to solve for x:

- If $b^2 - 4ac < 0$ there is no solution
- Otherwise there are two solutions:

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

(Taking either + or – at the place of the ± symbol gives the two solutions.)

Example:

$$2x^2 - 5x + 2 = 0$$

So *a = 2, b = -5, c = 2*

$$x = \frac{5 \pm \sqrt{25-16}}{4} = \frac{5 \pm 3}{4}$$

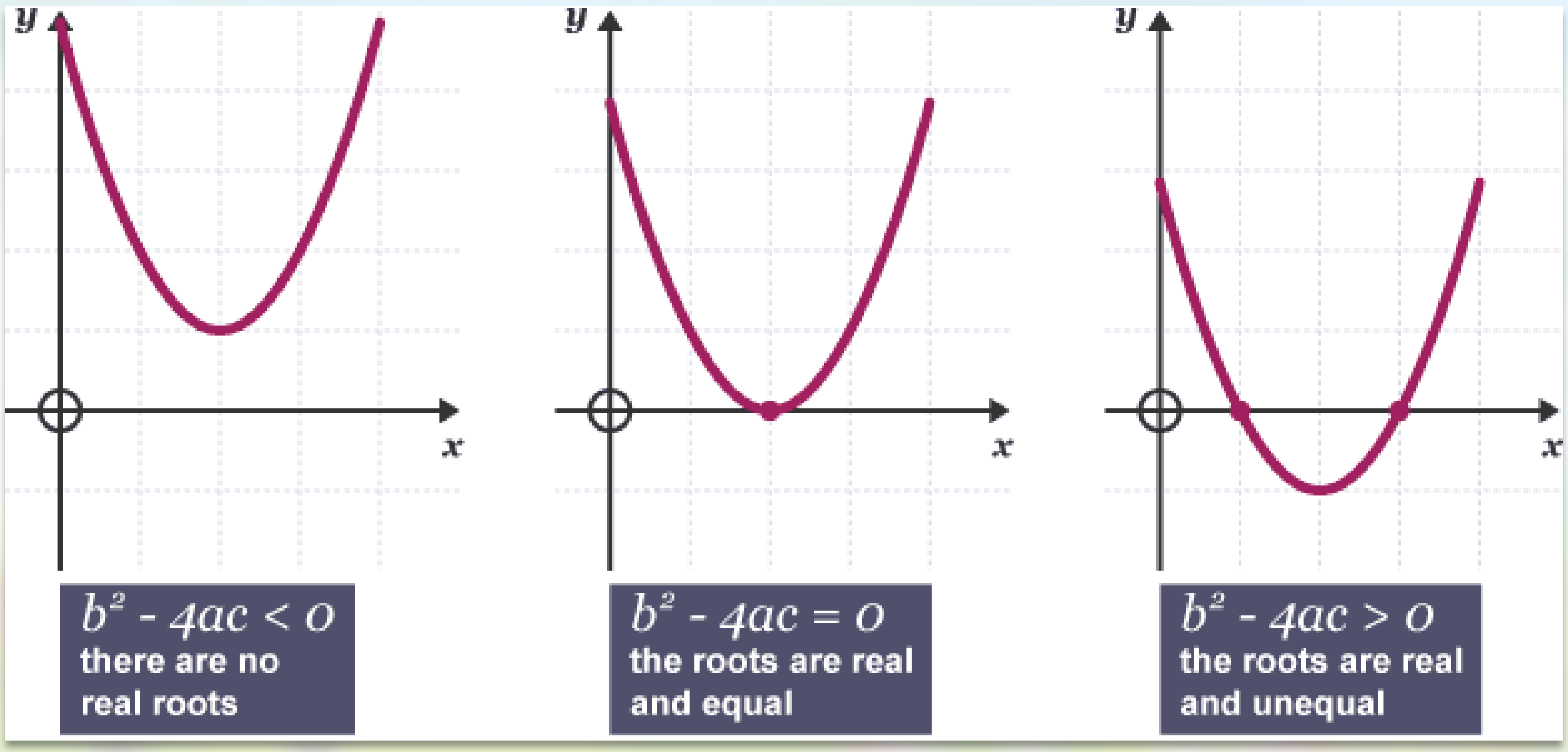
Conclusion:

$x = 2$ or $x = 1/2$

are the two solutions.

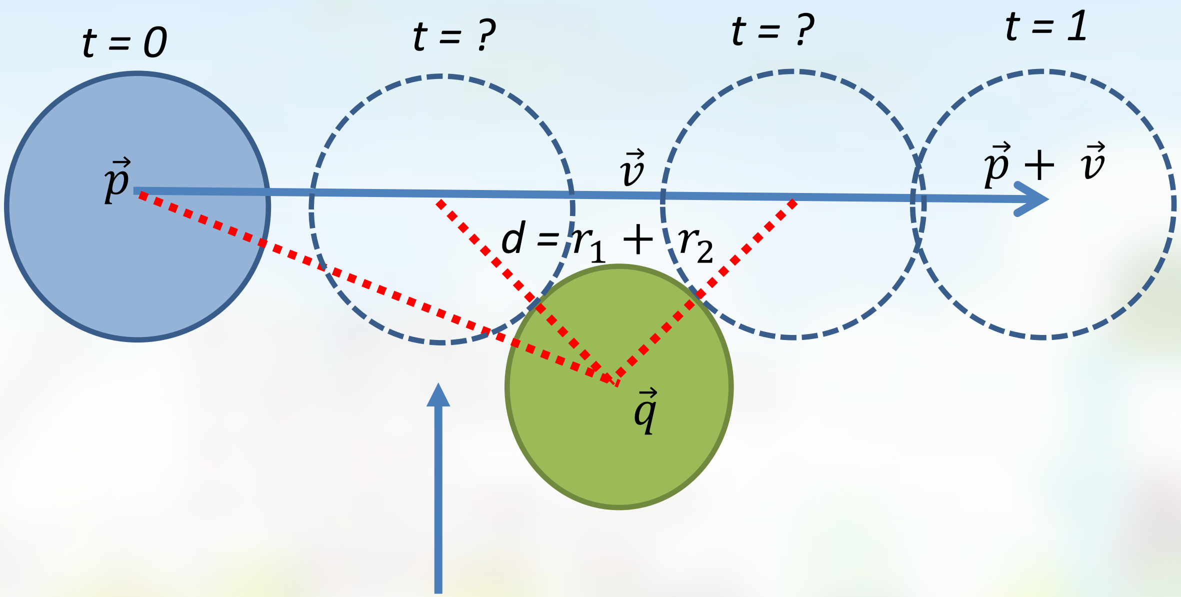
Function: $y = ax^2 + bx + c$

Solve: $ax^2 + bx + c = 0$



Computing Point of Impact

We know: ball positions at time 0 *($\vec{p}$ and $\vec{q}$),* position at time 1 *(= $\vec{p} + \vec{v}$).*



We want to know: the *first time* at which the distance is equal to $r_1 + r_2$.

Ball Position at Time *t*

- Let $\vec{p}$ denote the ball position at time 0, and let $\vec{v}$ denote its velocity.
- Then the position at time *t* is:

$$p(t) = \vec{p} + \vec{v}\,t$$

- So the distance from point $\vec{q}$ at time *t* is:

$$|\vec{p} + \vec{v}\,t - \vec{q}\,|$$

- So we want to solve for *t:*

$$|\vec{u} + \vec{v}\,t\,| = r_1 + r_2$$

where $\vec{u} = \vec{p} - \vec{q}$ (the *relative position* of ball 1).

Ball Position at Time *t*

- In other words:

$$|\vec{u} + \vec{v}\, t|^2 = (r_1 + r_2)^2$$

- ...Some rewriting gives...:

$$|\vec{v}|^2\, t^2 + (2\vec{u} \bullet \vec{v}\,)\, t + |\vec{u}|^2 - (r_1 + r_2)^2 = 0$$

- So in terms of the *abc formula:*

$$a = |\vec{v}|^2$$

$$b = 2\vec{u} \bullet \vec{v}$$

$$c = |\vec{u}|^2 - (r_1 + r_2)^2$$

Time of Impact (TOI)

- In terms of the *abc formula:*

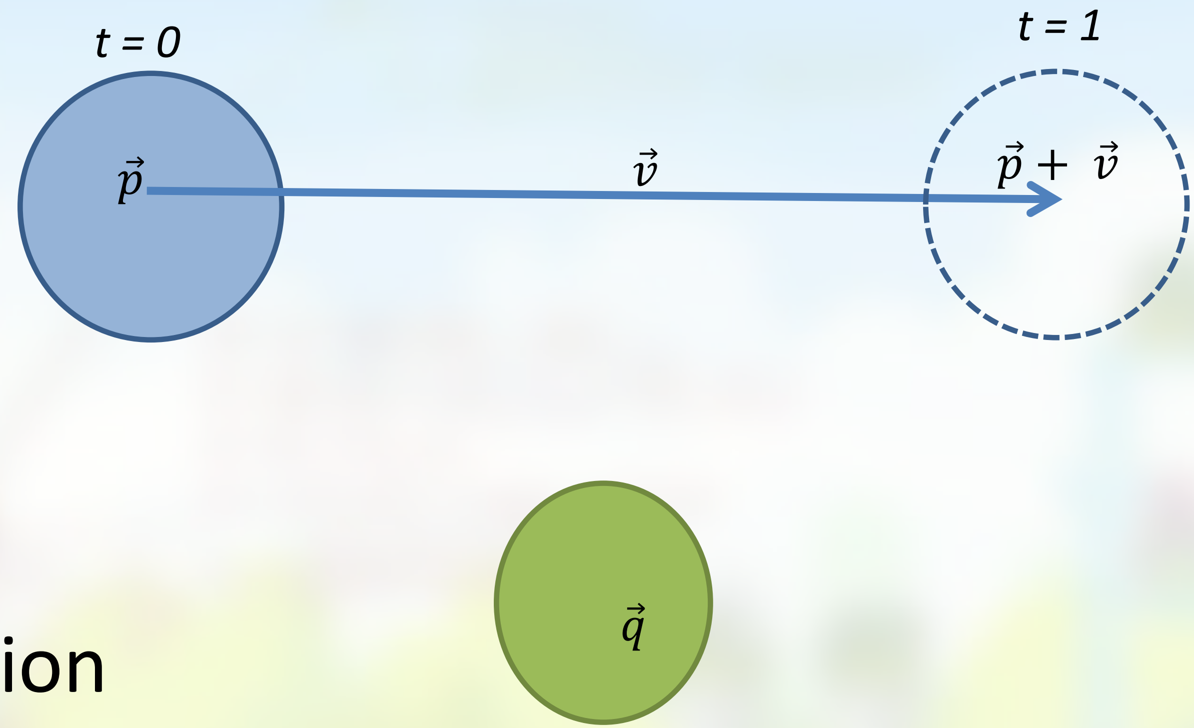
$$a = |\vec{v}|^2$$
$$b = 2\vec{u}\bullet\vec{v}$$
$$c = |\vec{u}|^2 - (r_1 + r_2)^2$$

Time of impact:

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

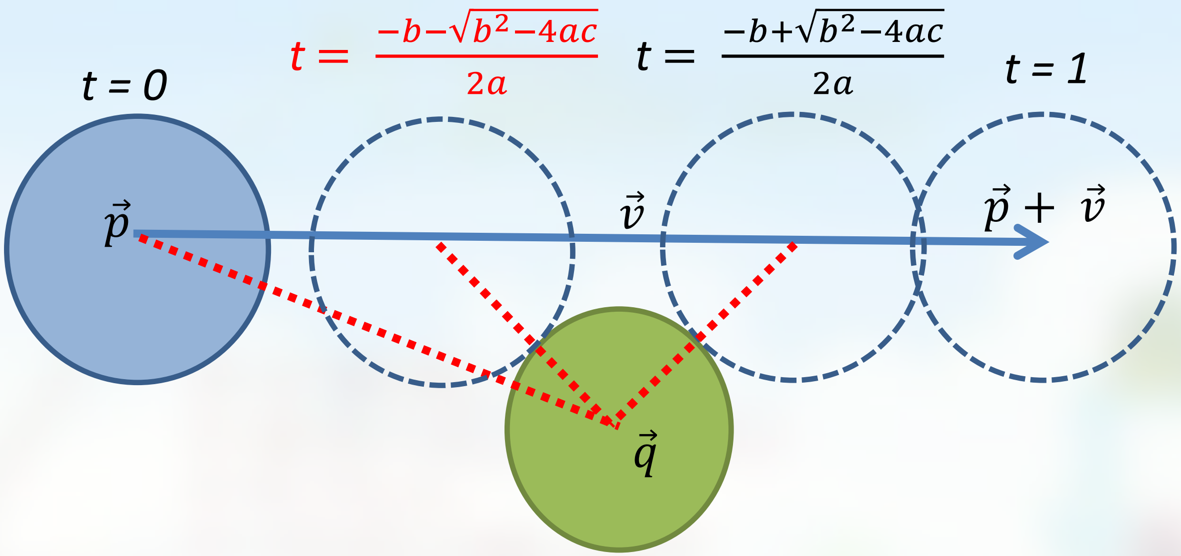
- Analysis:
  - a always positive, only zero when velocity = 0 → in that case, skip the collision test!
  - We are only interested in the *minimum solution*, which has a minus sign (at the position of ±).

Case: no solutions ($b^2 - 4ac < 0$)



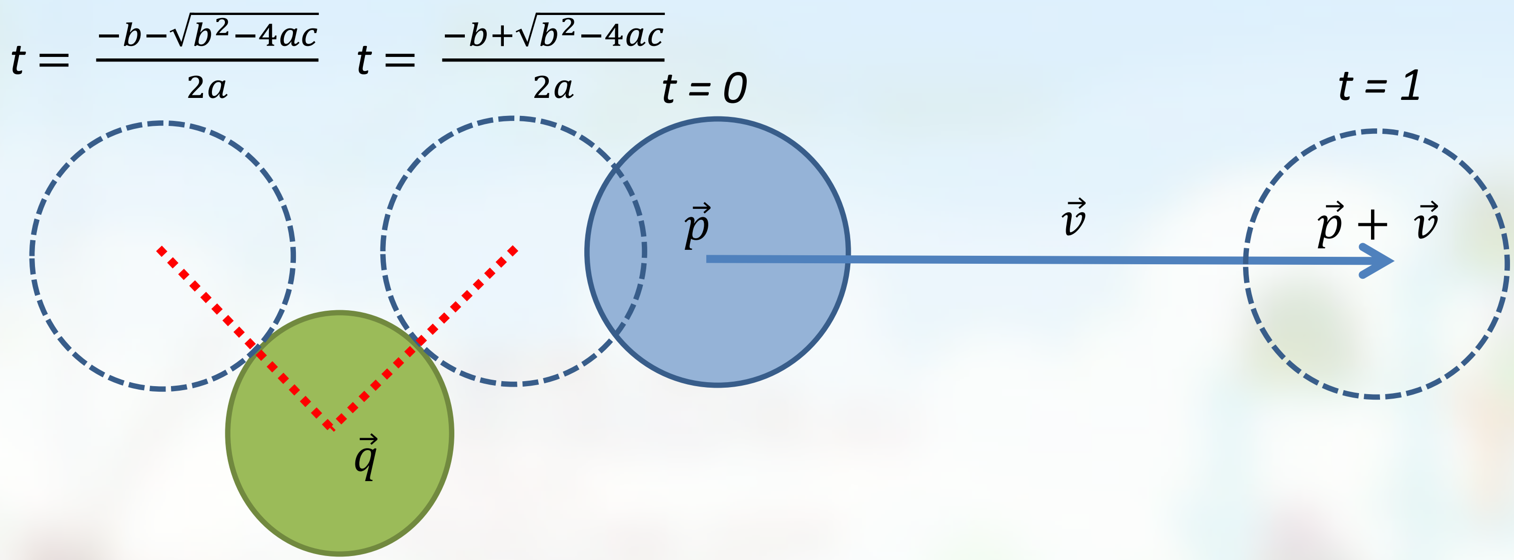
→ No collision

Case: two positive solutions



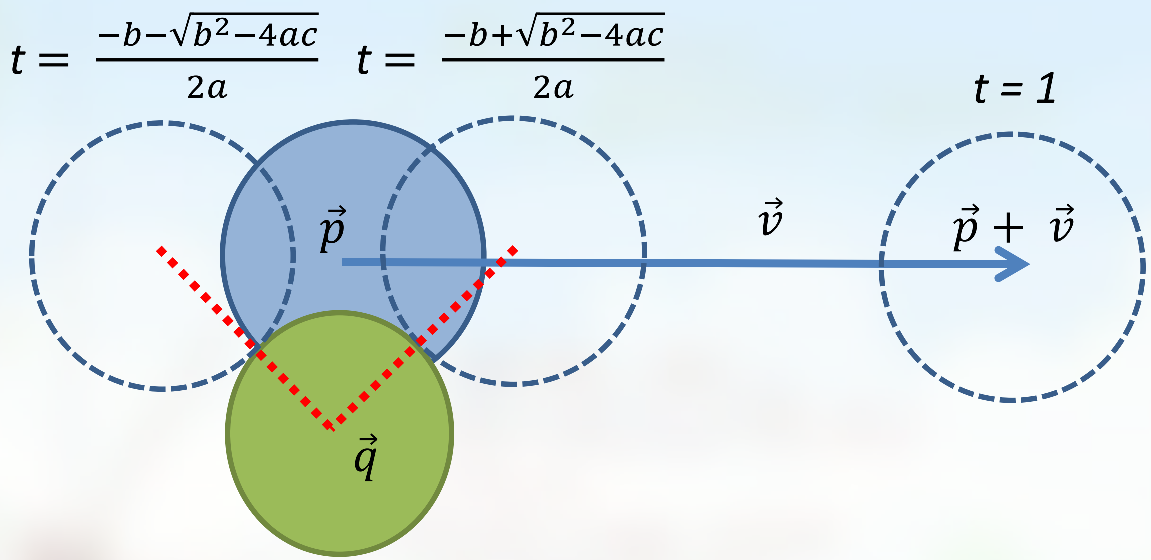
→ Collision at minimum solution

Case: two negative solutions



→ No collision (currently or in the future)

Case: one negative, one positive solution



→View this as no collision…?

(and wonder how you got into this mess…)

Pseudo code – Continuous – attempt 1

Given *relative position, velocity* and *radius1 & 2:*

Compute *a,b,c* as shown before

If *a ≈ 0* return *no collision*

Recall:

$$a = |\vec{v}|^2$$

So speed=0 if $a = 0$

$$D = b^2 - 4ac$$

If *D < 0* return *no collision*

$$t = \left(-b - \sqrt{D}\right) / (2a)$$

If $0 \leq t < 1$ return *collision at time t*

Return *no collision* (this frame)

Resolving the collision

- If a collision with *time of impact t* is found:
- Compute POI = $\vec{p} + \vec{v}\,t$
- Reset the ball position to POI

Then similar to before (discrete case):

- Compute the collision(unit) *normal* using POI.
- Reflect the velocity using the normal

Trying the code in practice

- This code works perfectly without gravity
- This code works well with gravity…
- …until the ball should come at rest – instead the balls pass through each other! (demo: scene 4)

Trying the code in practice

- This code works perfectly without gravity
- This code works well with gravity…
- …until the ball should come at rest – instead the balls pass through each other! (demo: scene 4)

Q: What's going on here?

A: *floating point rounding errors:*

- After a collision, we reset the position to POI, at distance exactly $r_1 + r_2$
- But sometimes the distance is *slightly less than* $r_1 + r_2$
- Then our code gives a TOI $t < 0$, which is ignored
- How to solve this?

Pseudo code – Continuous – attempt 2

Given *relative position u, velocity* and *radius1 & 2:*

Compute *a,b,c* as shown before

If *c < 0* return *collision at time 0*

Recall:
$c = |\vec{u}|^2 - (r_1 + r_2)^2$
So *overlapping* if c<0

If *a ≈ 0* return *no collision*

$D = b^2 - 4ac$

If *D < 0* return *no collision*

$t = \left(-b - \sqrt{D}\right) / (2a)$

If $0 \leq t < 1$ return *collision at time t*

Return *no collision* (this frame)

Trying the code in practice

- In this version of the code, balls sometimes "stick together"…. (demo: scene 3)

Q: What's going on here?

A: We should not return a collision when the balls are already *moving away from each other!*

Q: How can we check this?

A: Using the *dot product,* comparing the relative position and the velocity: $\vec{u} \bullet \vec{v}$ → *positive* when *moving away*

Pseudo code – Continuous – attempt 3

Given *relative position u, velocity v* and *radius1 & 2:*

Compute *a,b,c* as shown before

If *c < 0:*

  if b < 0 return *collision at time 0*

  else return *no collision*

If *a ≈ 0* return *no collision*

$$D = b^2 - 4ac$$

If *D < 0* return *no collision*

$$t = \left(-b - \sqrt{D}\right) / (2a)$$

If $0 \leq t < 1$ return *collision at time t*

Return *no collision* (this frame)

Recall:

$$c = |\vec{u}|^2 - (r_1 + r_2)^2$$

So *overlapping* if c<0

Recall:

$$b = 2\vec{u} \bullet \vec{v}$$

So *moving away* if b>0

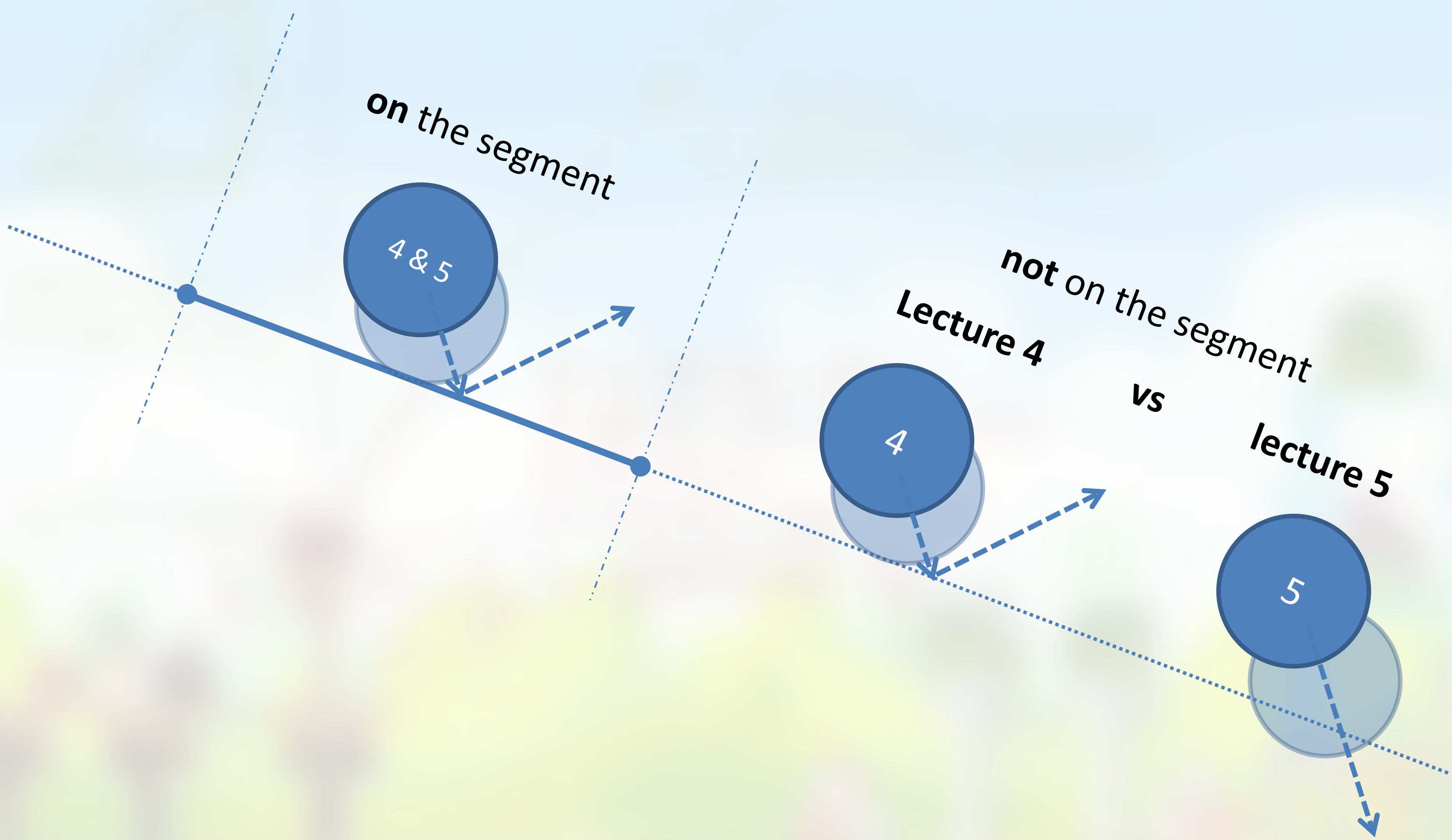
Trying the code in practice

- This version of the code works perfectly ☺

Circle / line segment collision

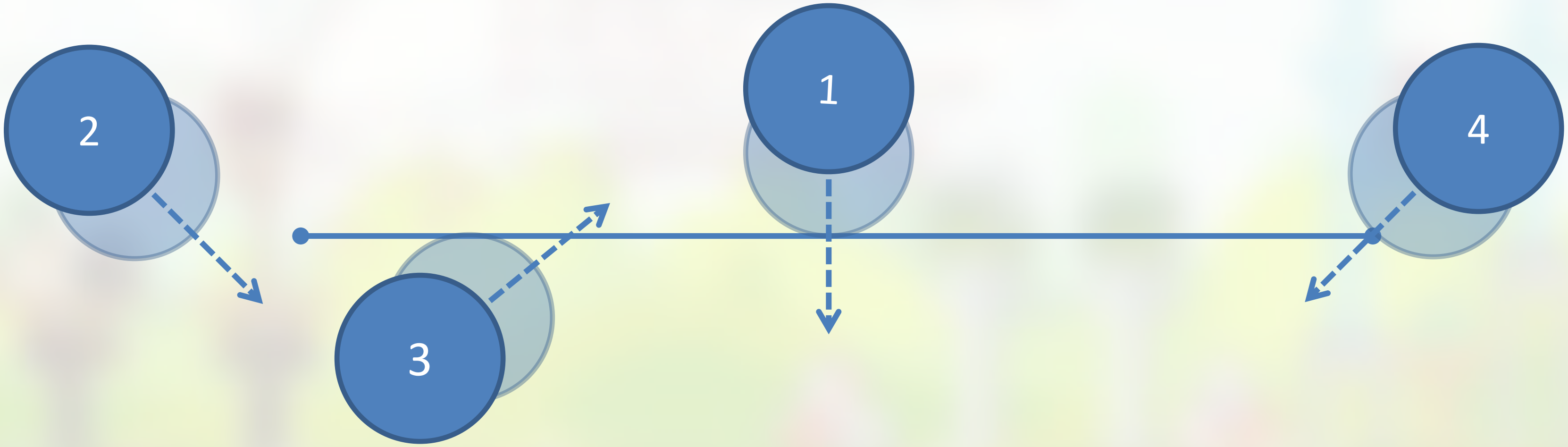
Collision on a line **segment**

A distinction which lecture 4 didn't make:



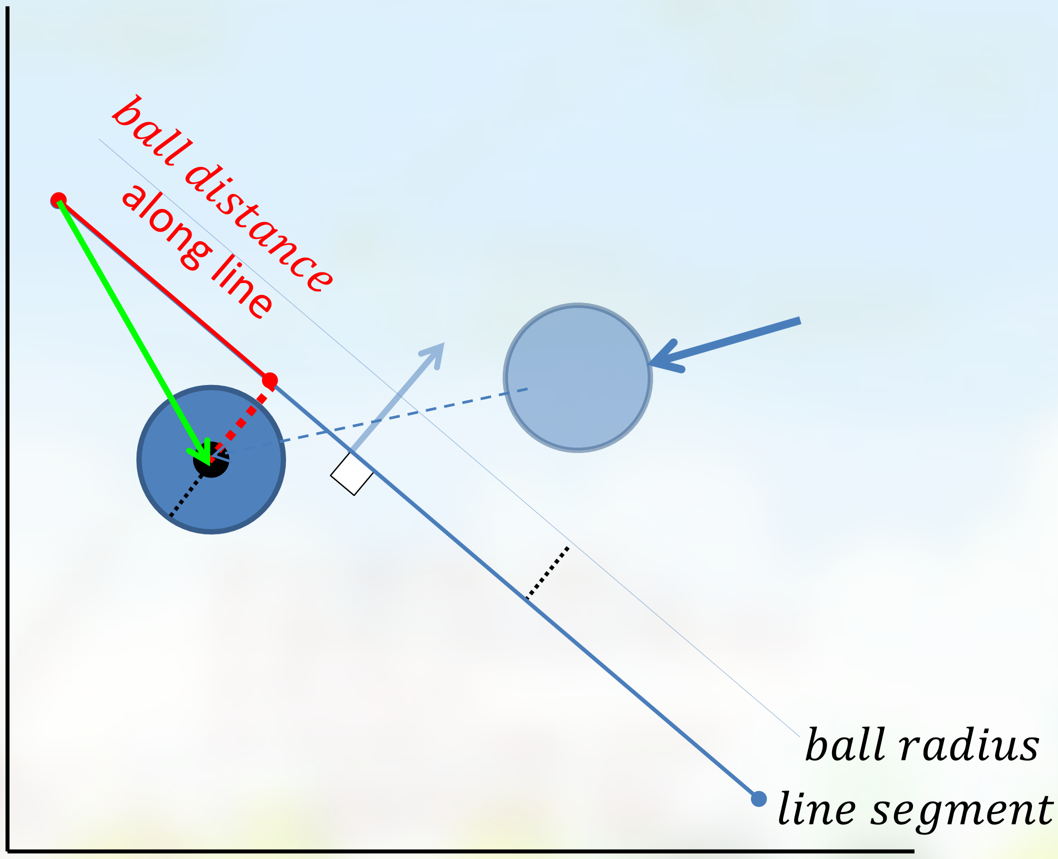
Line segments

- Line segment collision has to take into account:
  1. collisions with the actual line **segment**
  2. being able to move behind the line segment
  3. double sided bouncing on the line segment
  4. collisions with the caps of the line



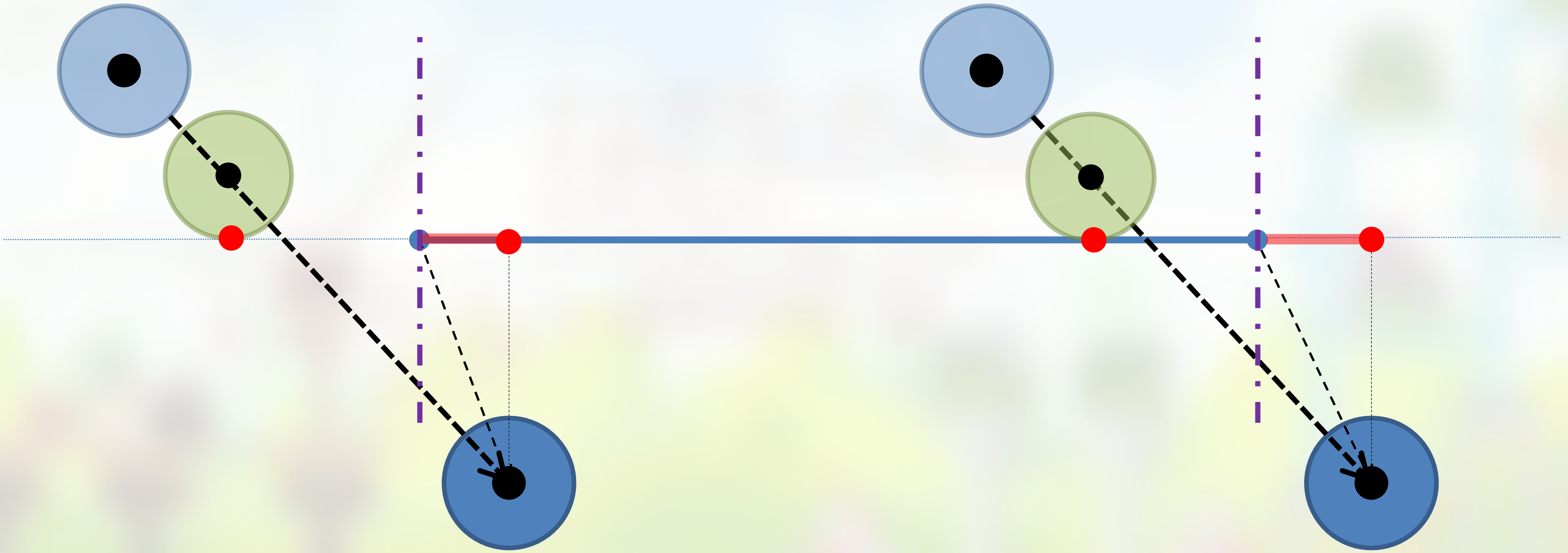
How not to do it…

Checking *distance along the line* after move…

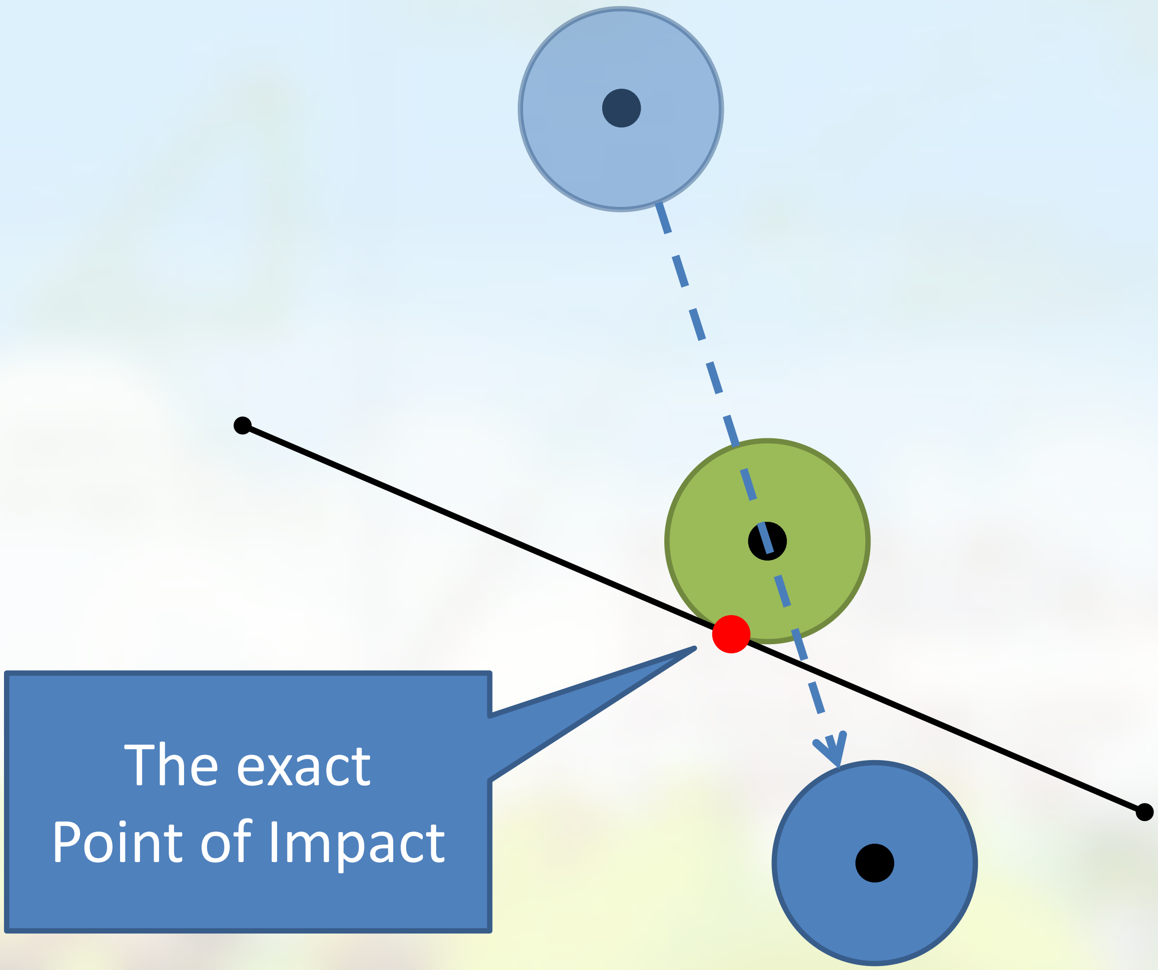


This doesn't work too well, due to…

- False negatives/positives
  - Left side: false positive, check tells us we are on the segment, although in reality we went past it
  - Right side: false negative, check tells us we went past the segment, although in reality we hit it

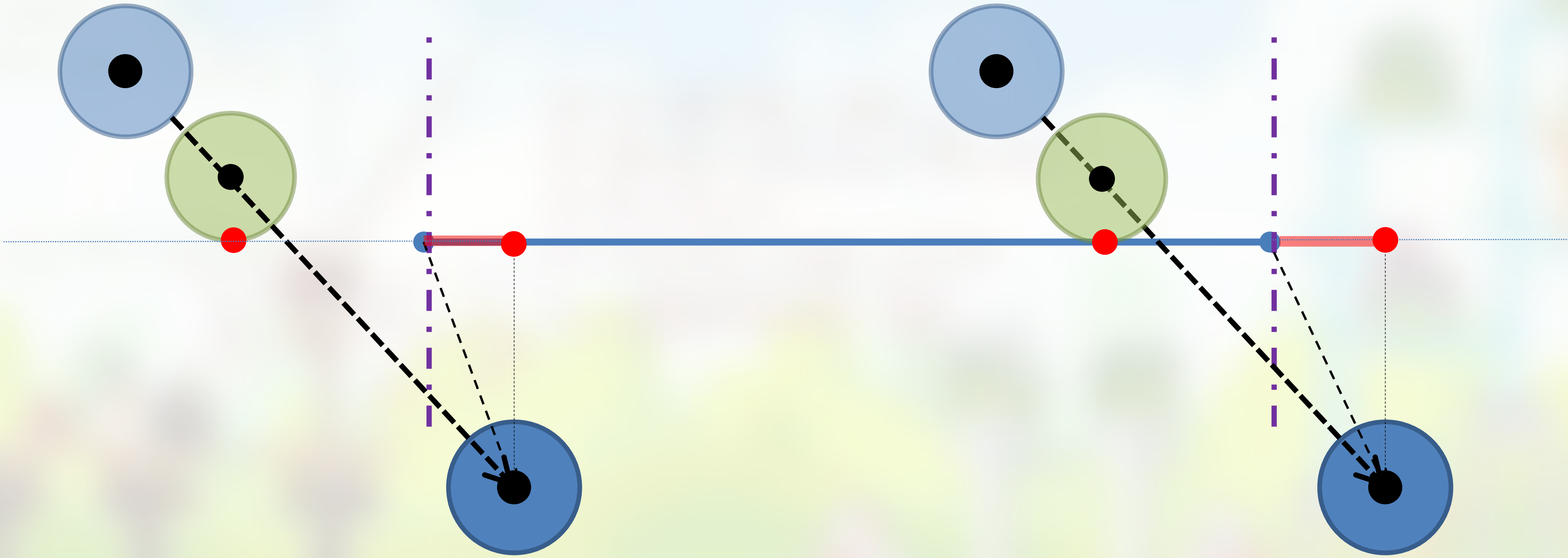


Better way: using exact Point of Impact

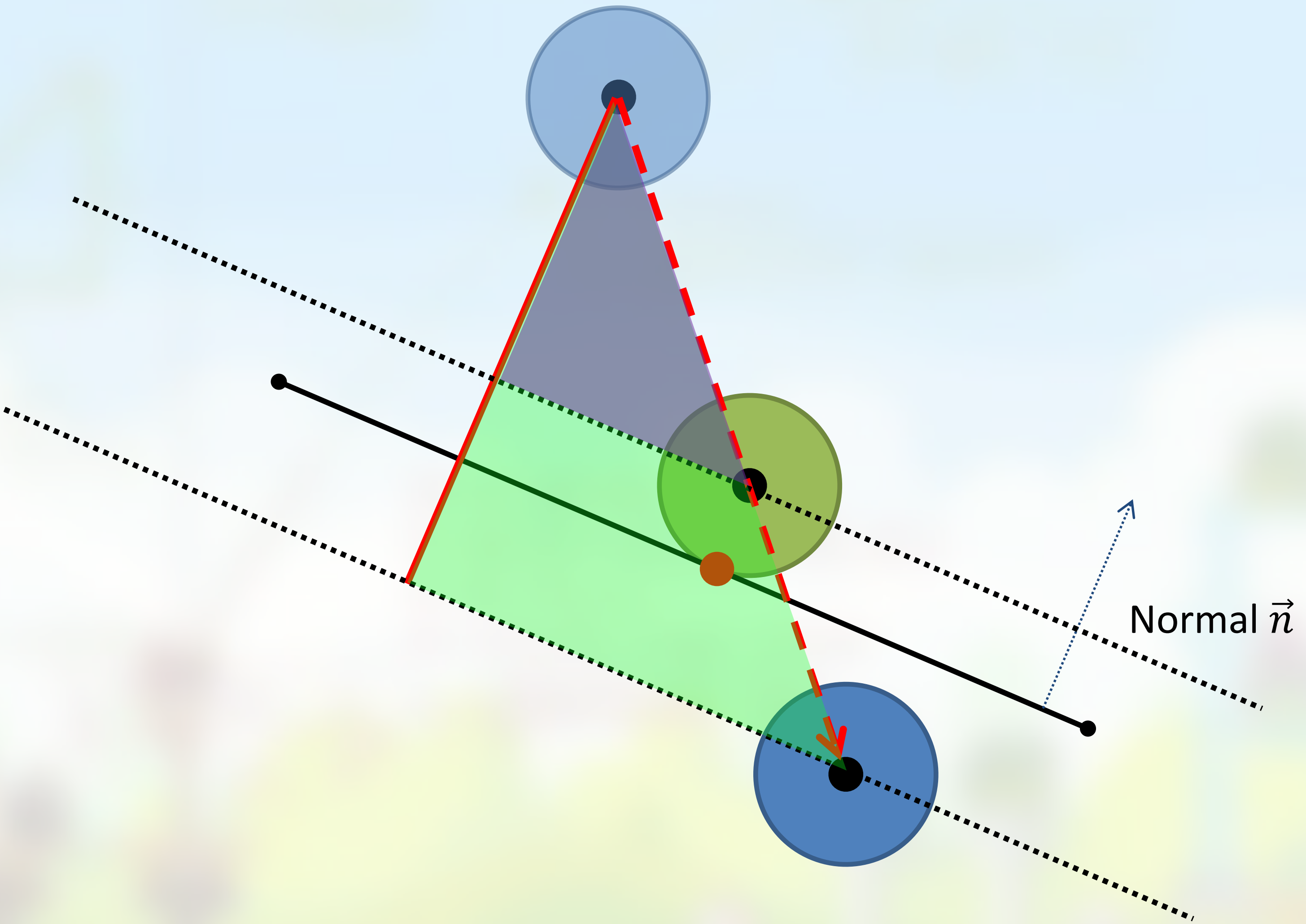


Works much better !

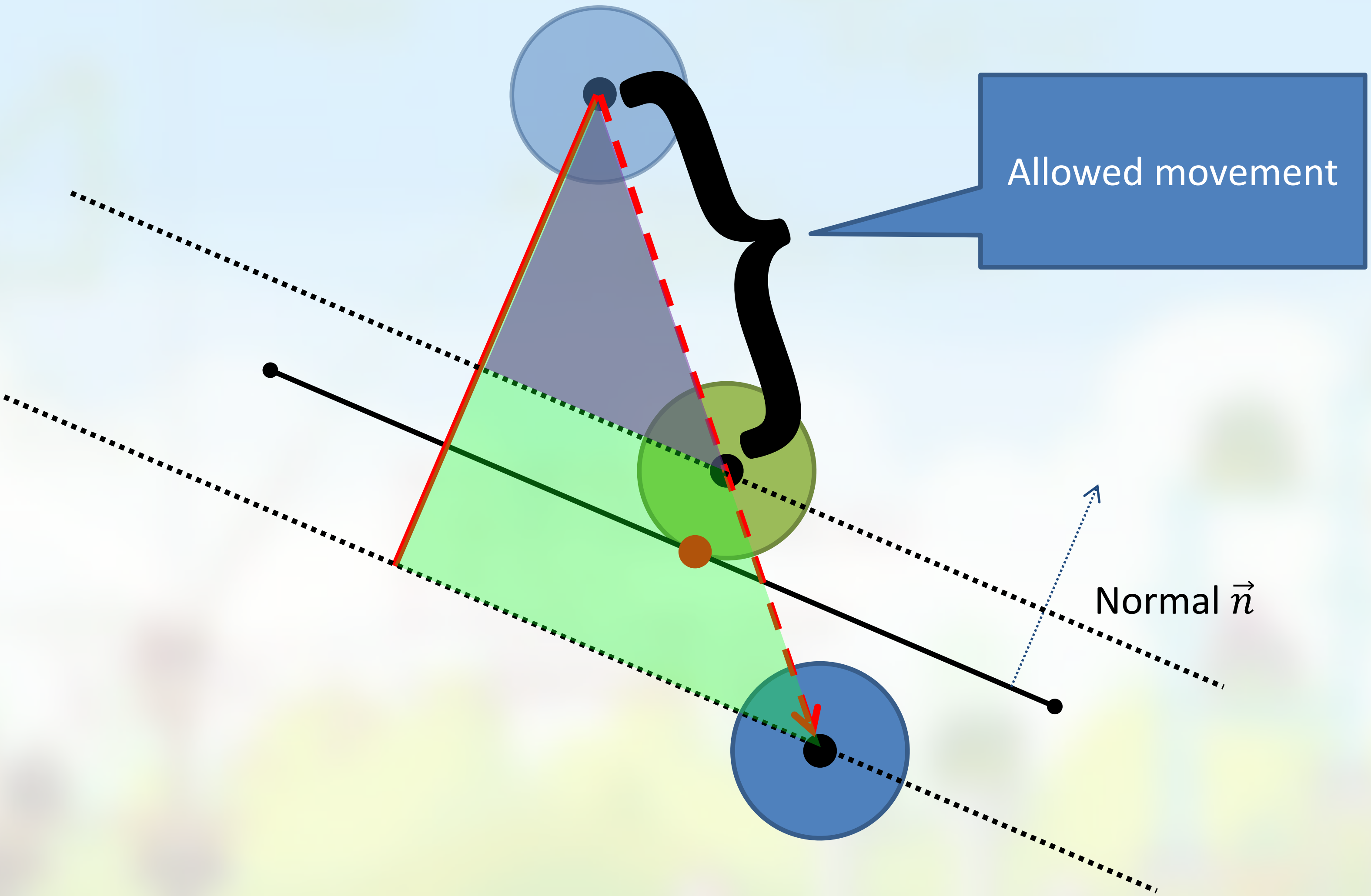
- No more false negatives/positives:
  - Left side: POI not on segment → we went past the line
  - Right side: POI on segment → hit!



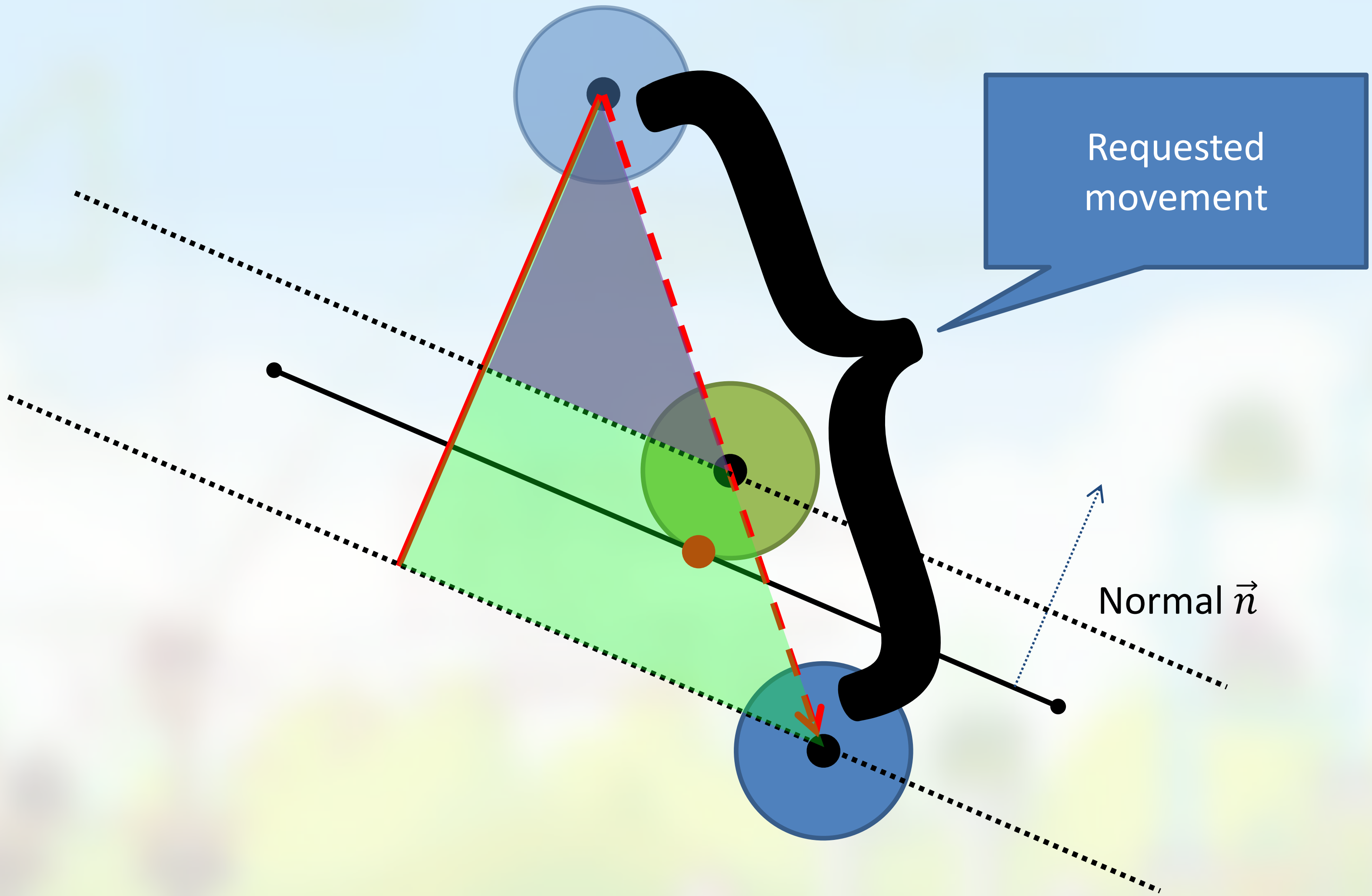
Computing POI on a line - recap



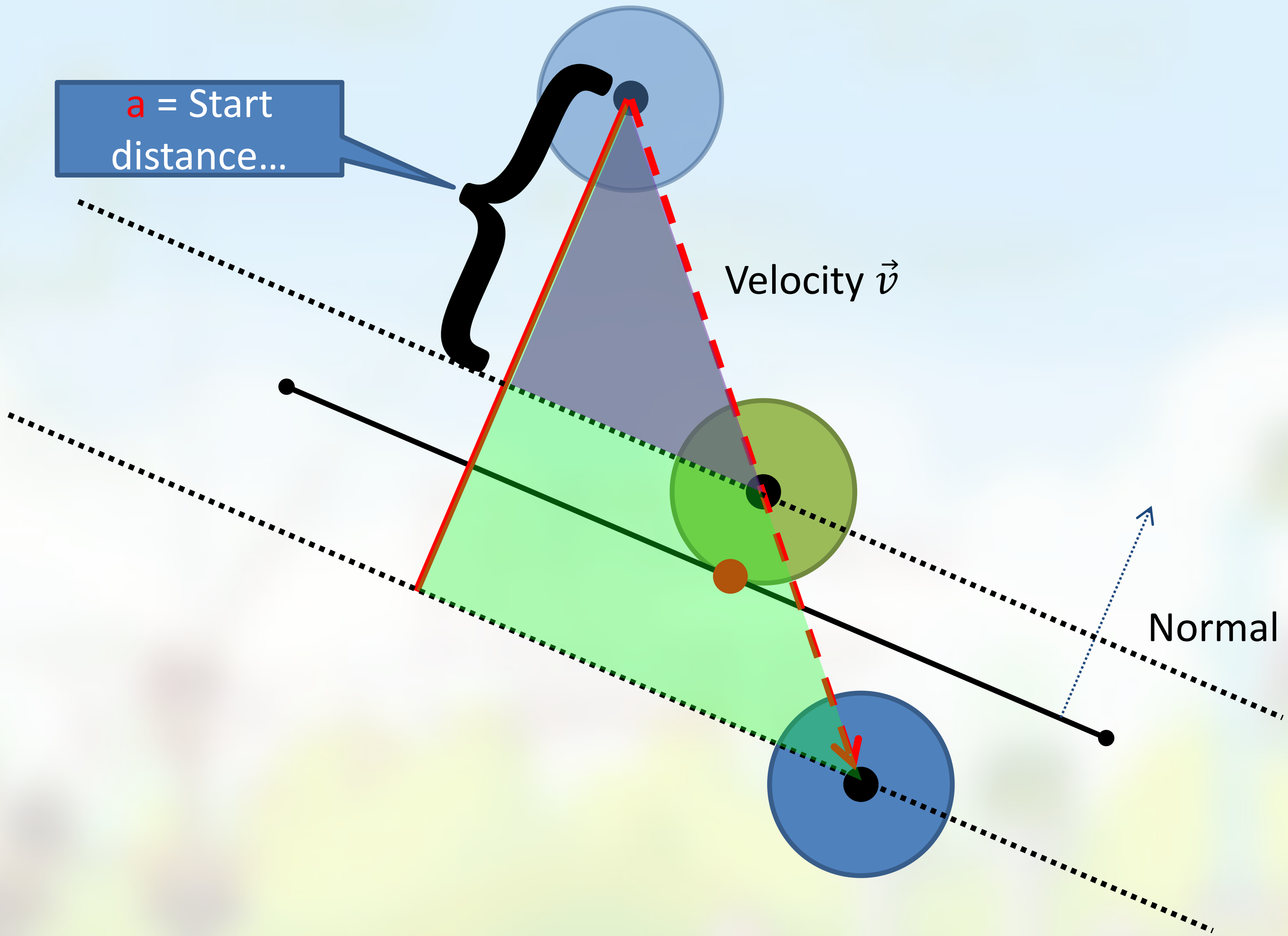
The *ratio* between the *allowed movement*…



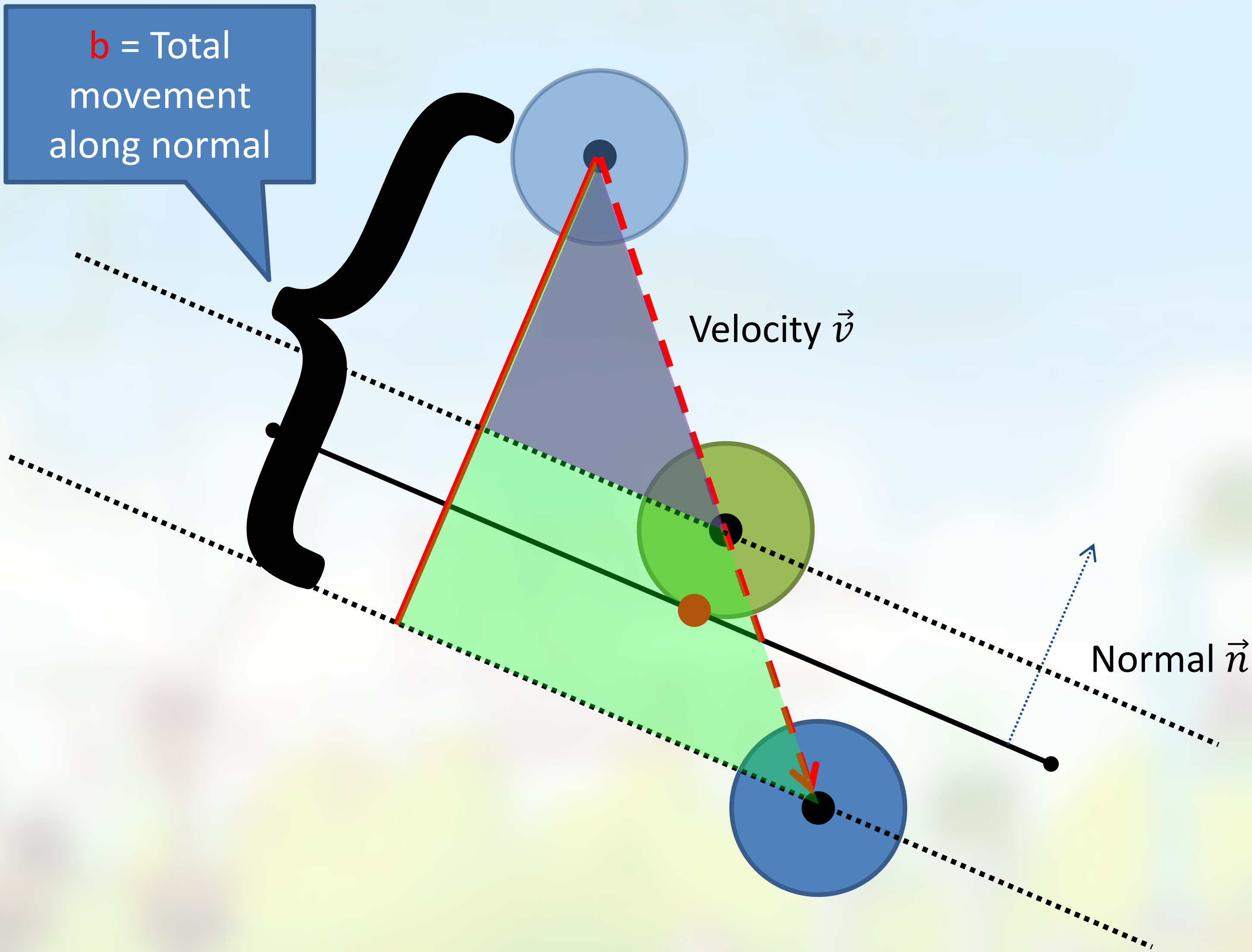
…and the “requested” movement…



…is the same as the ratio between this bit…



…and this bit:



- Compute *a* and *b* using scalar projection:

  a = ? .Dot(?) – radius

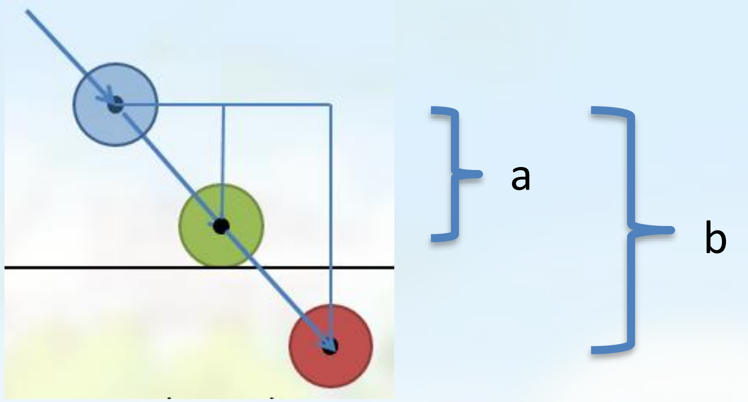
  b = - ? .Dot(?)

- (In the case where this POI calculation is relevant, you *would expect* both *a* and *b* to be positive…→ more on this later)

- Next: a familiar slide…

Point of Impact (POI) Calculation

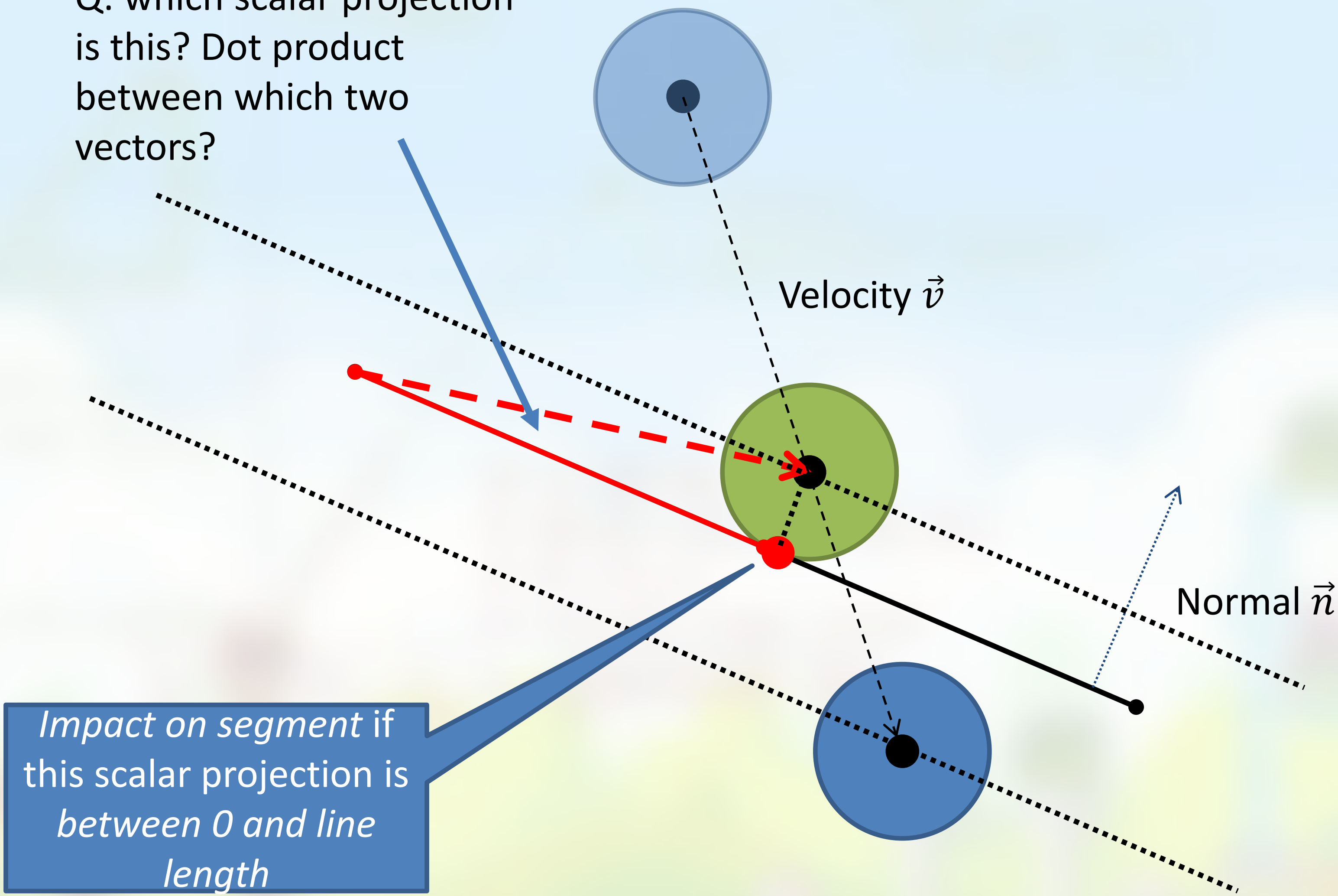
- Consider three values:
  - oldDistance →
  - Radius →
  - newDistance →



- They define lengths a and b. (with a<b)
- Define *time of impact* $t = a/b$.
  - If t=0: impact at "start of current frame"
  - If t=1: impact at "end of current frame"
- Set: POI = oldPosition + t * velocity

Knowing the POI, we compute the "distance along the line" using another *scalar projection*

Q: which scalar projection is this? Dot product between which two vectors?



Pseudo Code: Line Segment Collision

Compute *a* and *b* as shown before

If $b \leq 0$ return *no collision* (→ *moving away)*

If $a < 0$ return *no collision* (→ *already below)*

$t = a/b$ (=*"time of impact"*)

if $t \leq 1$:

    POI = oldPosition + velocity * *t*

    Compute *d* = distance along the line

    if $d \geq 0$ and $d \leq$ LineLength:

        return *Collision at time t*

Return *no collision*

Evaluation version 2

Problem: the ball only bounces on one side of the line segment

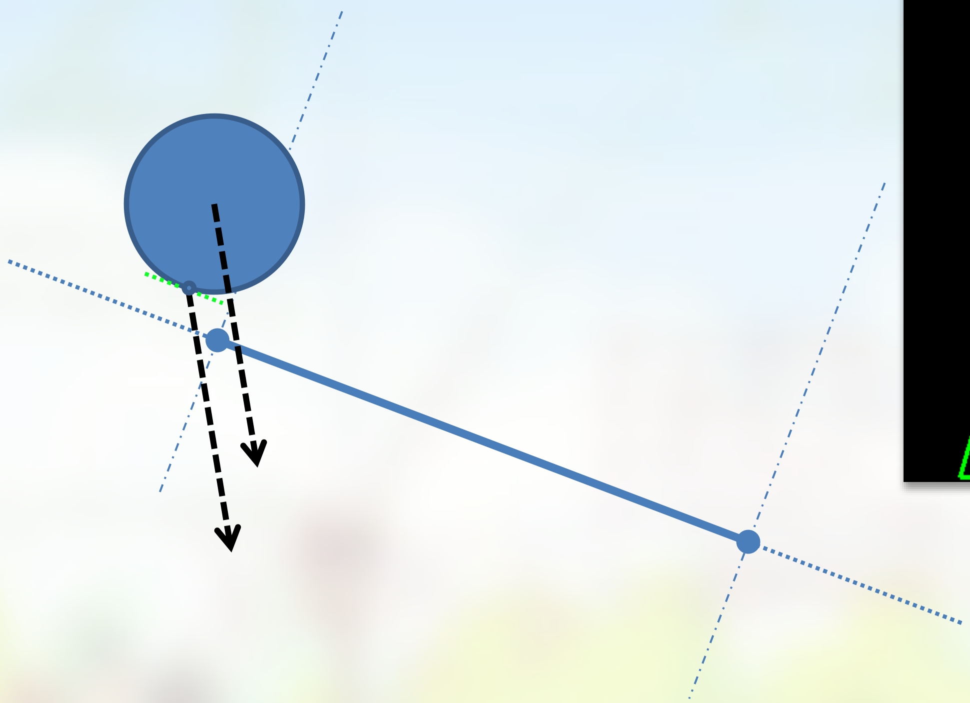
Q: How can we solve this?

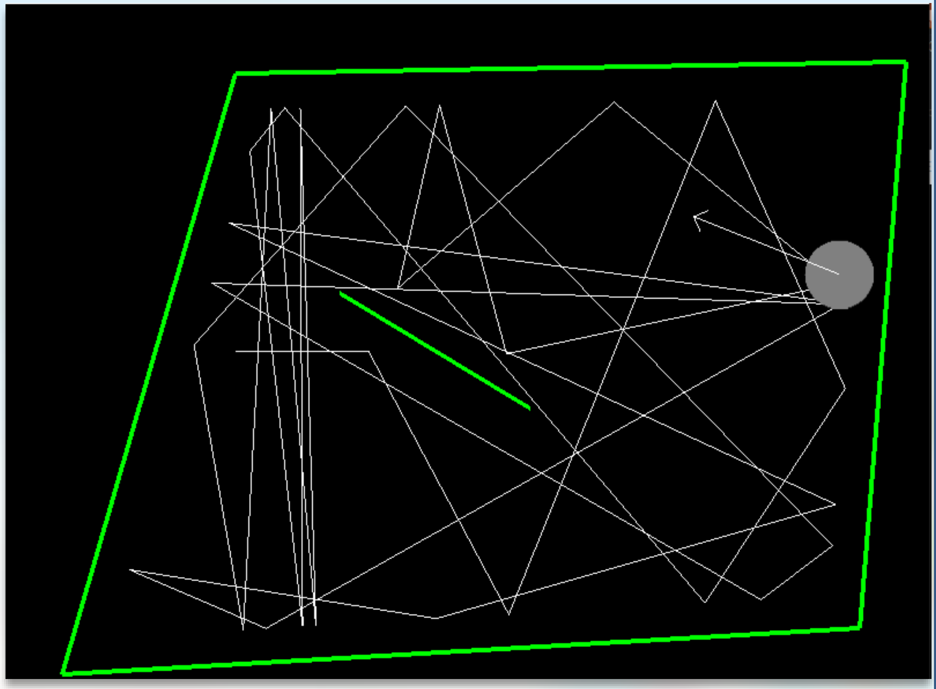
A: Best solution:

Add two line segments with *opposite normals:*

- One from point A to point B.
- One from point B to point A.

Sometimes we still go through...





Evaluation version 3

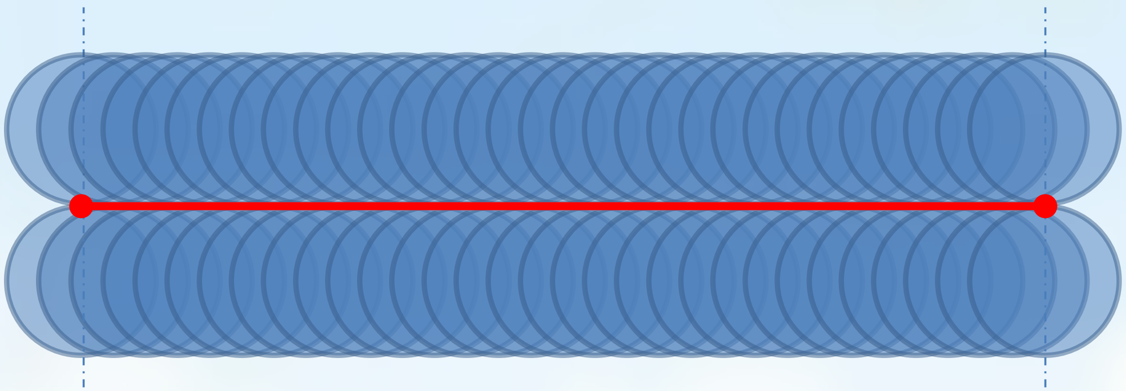
Problem: the ball still goes through the ends of the line segment

Q: How can we solve this?

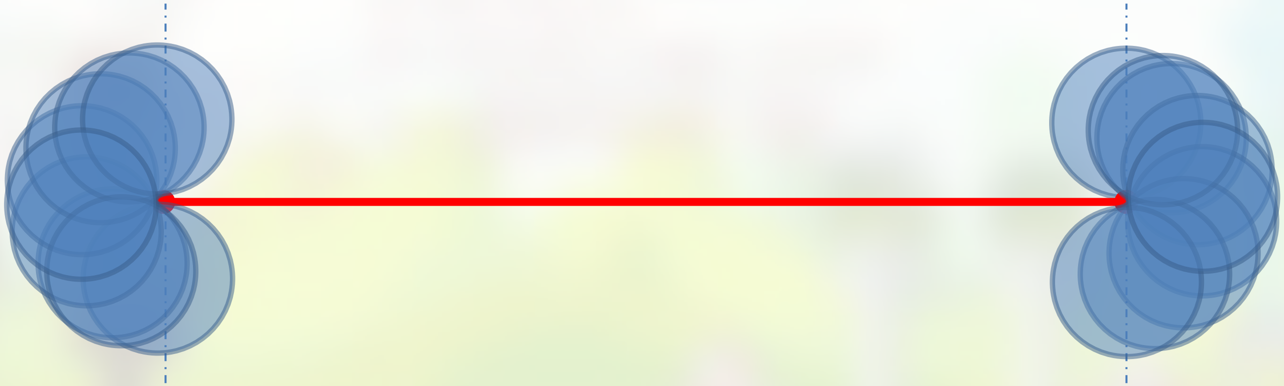
A: Add two circles (balls) with radius 0 to the ends of the line segment → called *line caps.*

Line segment vs line caps

Contact points for the segment

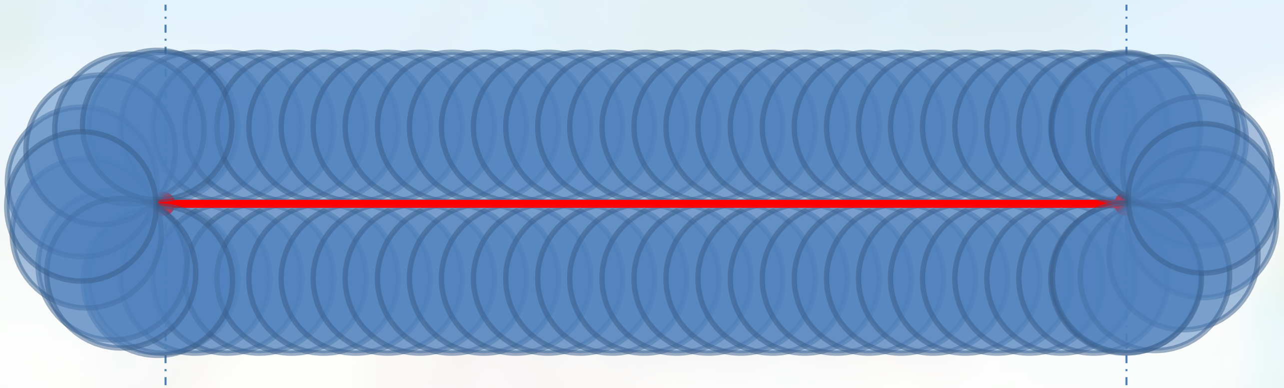


Contact points for the caps



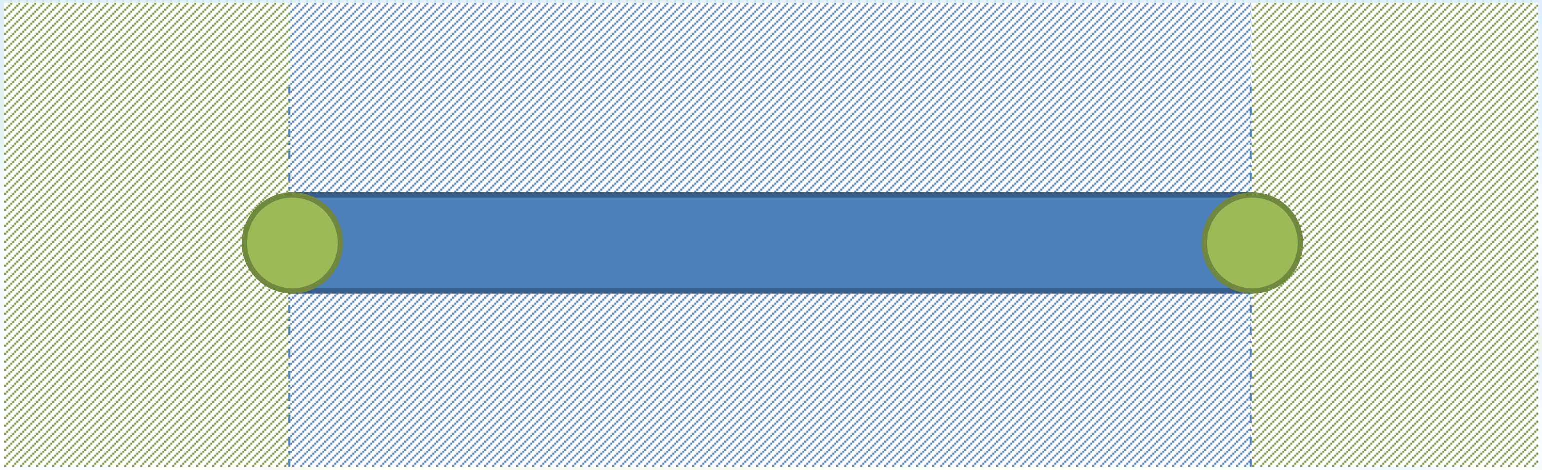
Line segment + caps

Together the line segment & caps will form a collision shape like a bumper with rounded edges



Rounded line caps

- That means that if we would zoom in on a line with rounded line caps it would look like:



- In other words: hitting the line cap is like bouncing on a circle with a really really small radius (aka 0)…

Evaluation version 4

- This code works perfectly without gravity
- This code works well with gravity…
- …until the ball should come at rest – instead the ball passes through the line segment!

Q: What's going on here?

A: Again: *floating point rounding errors:*

- After a collision, we reset the position to POI, at distance exactly r from the line
- But sometimes the distance is *slightly less than* r
- Then our code gives *a<0* and a TOI $t < 0$, which is ignored
- How to solve this?

Possible solutions:

- Compute TOI *t* using negative *a,* reset position using negative *t.*

  → not ideal: resets the position above the line, but possibly overlapping with other things…

  Plus you have to be careful to not do this when the ball is already far below the segment.

- Better: if *a* is between –*radius* and 0, use *t=0* (so POI = oldPosition)

  → If *a* is even more negative, the ball center is already past the line. → It's better to continue moving

Pseudo Code: Line Segment Collision

Compute *a* and *b* as shown before

If $b \leq 0$ return *no collision*

If $a \geq 0$:

    $t = a/b$ *(so always positive)*

else if $a \geq -r$ *(going towards deeper collision)*

    $t = 0$

else return *no collision* *(ball center already past line)*

if $t \leq 1$:

    POI = oldPosition + velocity * *t*

    Compute *d* = distance along the line

    if $d \geq 0$ and $d \leq$ LineLength:

        return *Collision at time t*

Return *no collision*

Collision check order

- We now have pieces of code for:
  - detecting ball/ball collisions + finding TOI

    (also used for *line caps)*
  - detecting ball/line (segment) collisions + finding TOI
  - resolving collisions

    (=reset position to POI and

    change velocity by reflecting using collision normal)

Q: In which order should this be done?

A:

- First find the *earliest collision (=minimum TOI),* and
- only resolve that one!

Assignment 5

Assignment 5

Parts:

- Circle / circle collision: discrete
- Circle / circle collision: continuous
- Circle / line segment collision
- Adding gravity
- Multiple moving balls: Newton’s laws

  → Tips for this: next lecture!

- Check blackboard for details.

Next week

- Summary of the course topics
- Combining everything
- A look ahead / next steps: what else is there in physics programming?
- Physics Programming final assignment preparation