Physics Programming

Lecture 3 Newton's Laws & Collisions

Slides & lesson materials by Hans Wichman & Paul Bonsma

Where are we?

	Insufficient	Sufficient	Good	Excellent
Vectors and Unit Testing (20%)	**0 pts** The Vec2 struct is not consistently used, basic functionality missing, or not all of its methods are unit tested.	**12 pts** The Vec2 struct is used for most vector operations, all the basic functionality is present (see the weekly assignments), and functional unit tests are	**16 pts** S + the Vec2 struct is used consistently for almost all vector operations, methods are implemented with code reuse and efficiency in mind, and good unit tests are chosen.	**20 pts** G + methods are all implemented efficiently in terms of code reuse or computation time. Useful extra functionality is added to the Vec2 struct.
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) 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: multiple moving objects following Newton's laws, combining gravity with sliding or rolling, kinematic objects such as moving platforms, or collision friction)

Week 1,2,4

Week 2

(See week 1 + 4 for some tips)

Week 4 + 5: angled lines

Assignment 3.1 Assignment 3.2 Assignment 3.3 + extra research

Need to Know

- The first part of this lecture (Assignment 3.1 / basic collision detection and resolve) is essential for the final assignment + assessment.
- The second part (Assignment 3.2 / point of impact) is necessary to score `good’ on `collisions’
- The third part (Assignment 3.3 / multiple moving objects) is necessary to score `excellent’ on `collisions’
- However, to create e.g. a dynamic platformer, you still need more than what’s explained in detail in this lecture!

Conclusion

- Use your time well: if you're already overwhelmed, don't stress about part 2 and 3 of this lecture / this week's assignment. (Not necessary to pass the course.)

- Nevertheless, for becoming a good physics programmer and creating dynamic game play, this lecture is just a starting point for self study!

Lecture overview

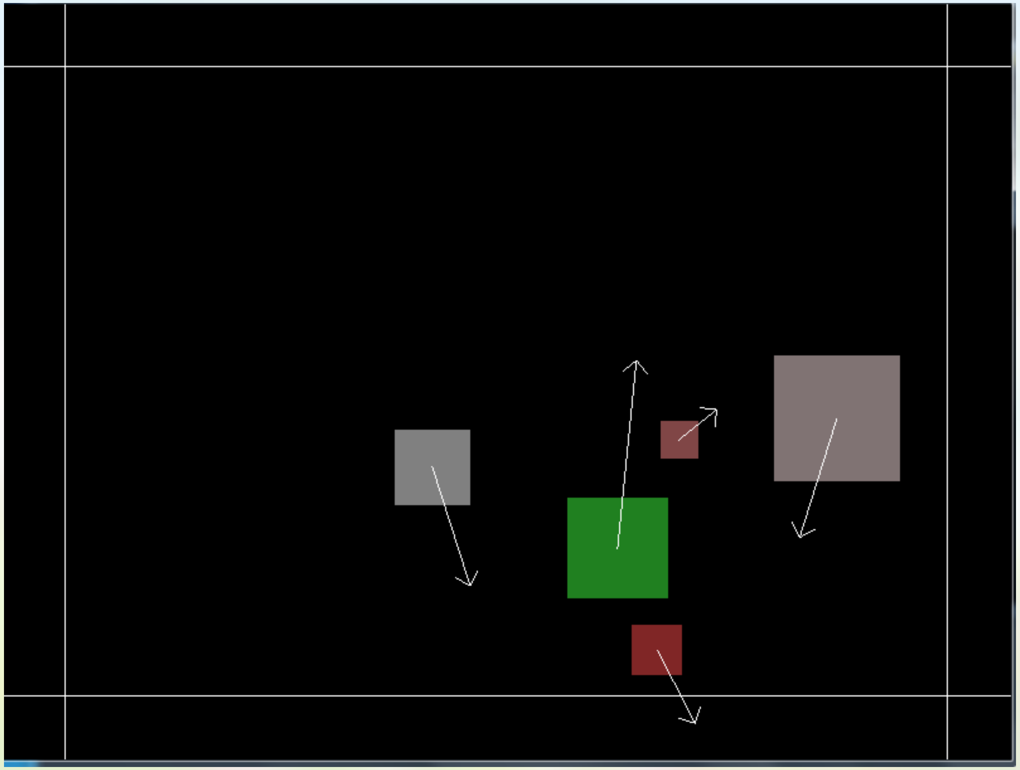
1. Collisions:
   – *detecting* (=did I hit something?) and
   – *resolving* (=what should I do now?)
2. Correct resolve: *point of impact*
3. Actual physics:
   1. *Newton's laws of motion,* and how to apply them*:*
   2. *Conservation of momentum*
4. *Bouncing blocks*: first steps towards creating your own physics engine. *(physics engine setup)*
   1. *Tips for Assignment 3.3*
5. Summary & outlook

Assignment 3

End result:

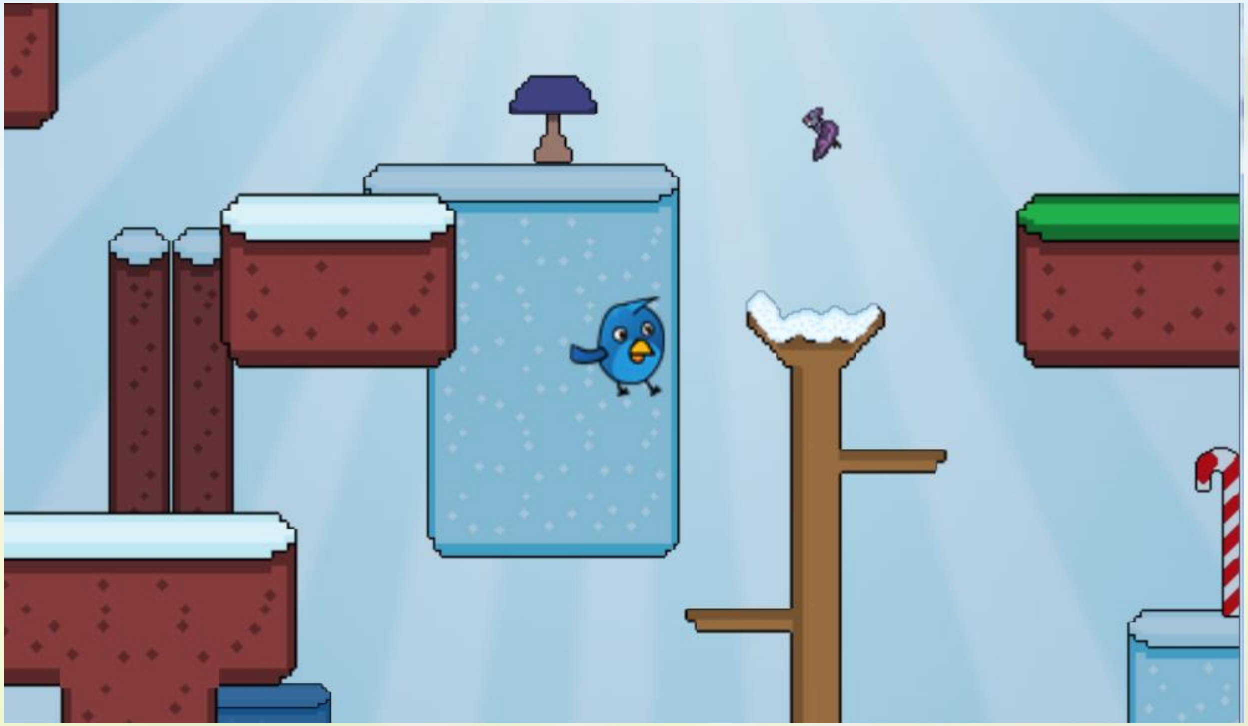
- Moving (axis aligned) square blocks
- Collisions with each other and boundaries
- Gravity

(demo)



Application

Using these techniques, you can create an interesting *platformer* with physics-based movement (bouncing, friction, pushing, moving objects, etc.)



(demo)

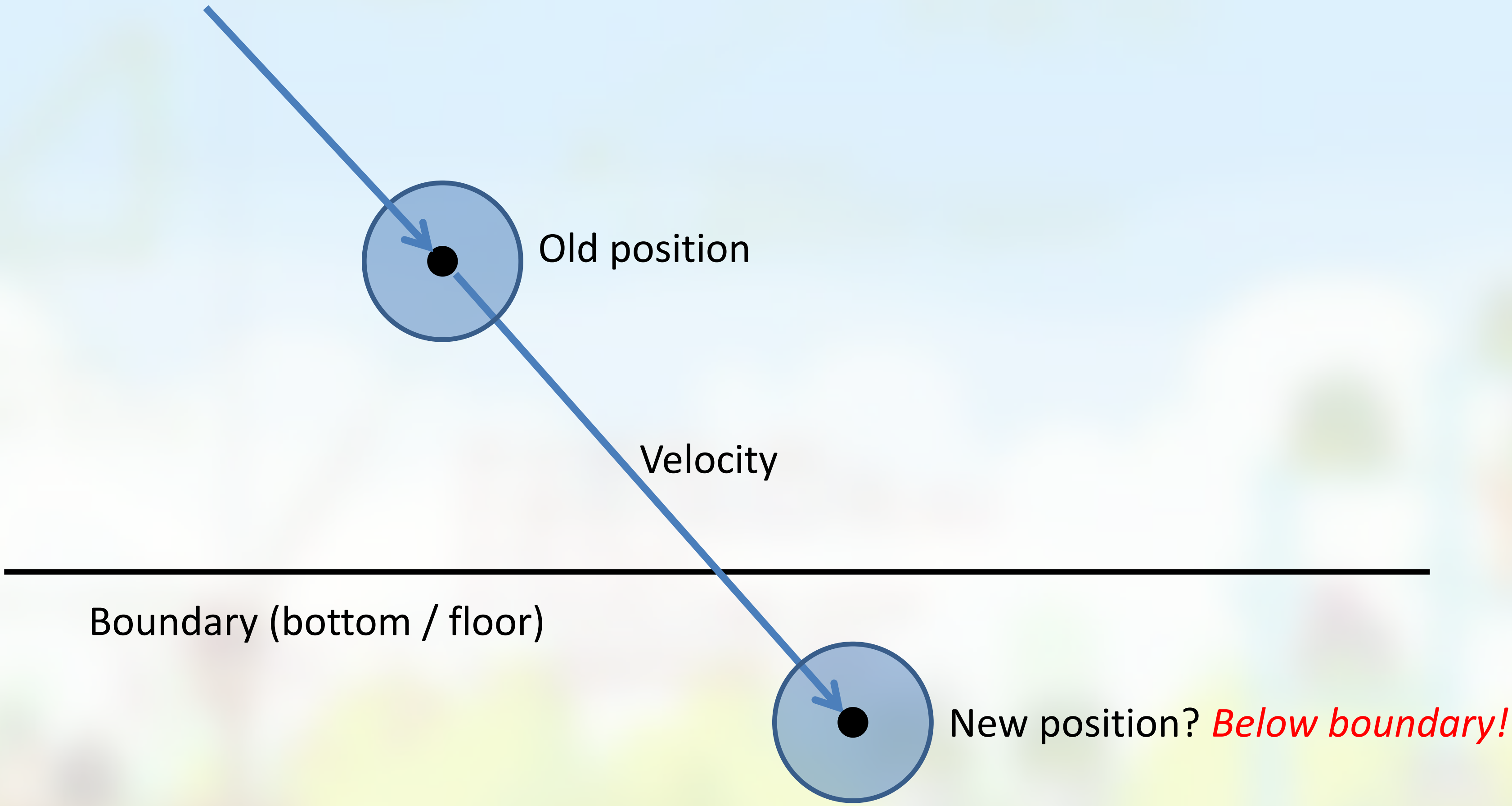
Collisions: detect and resolve

Collisions

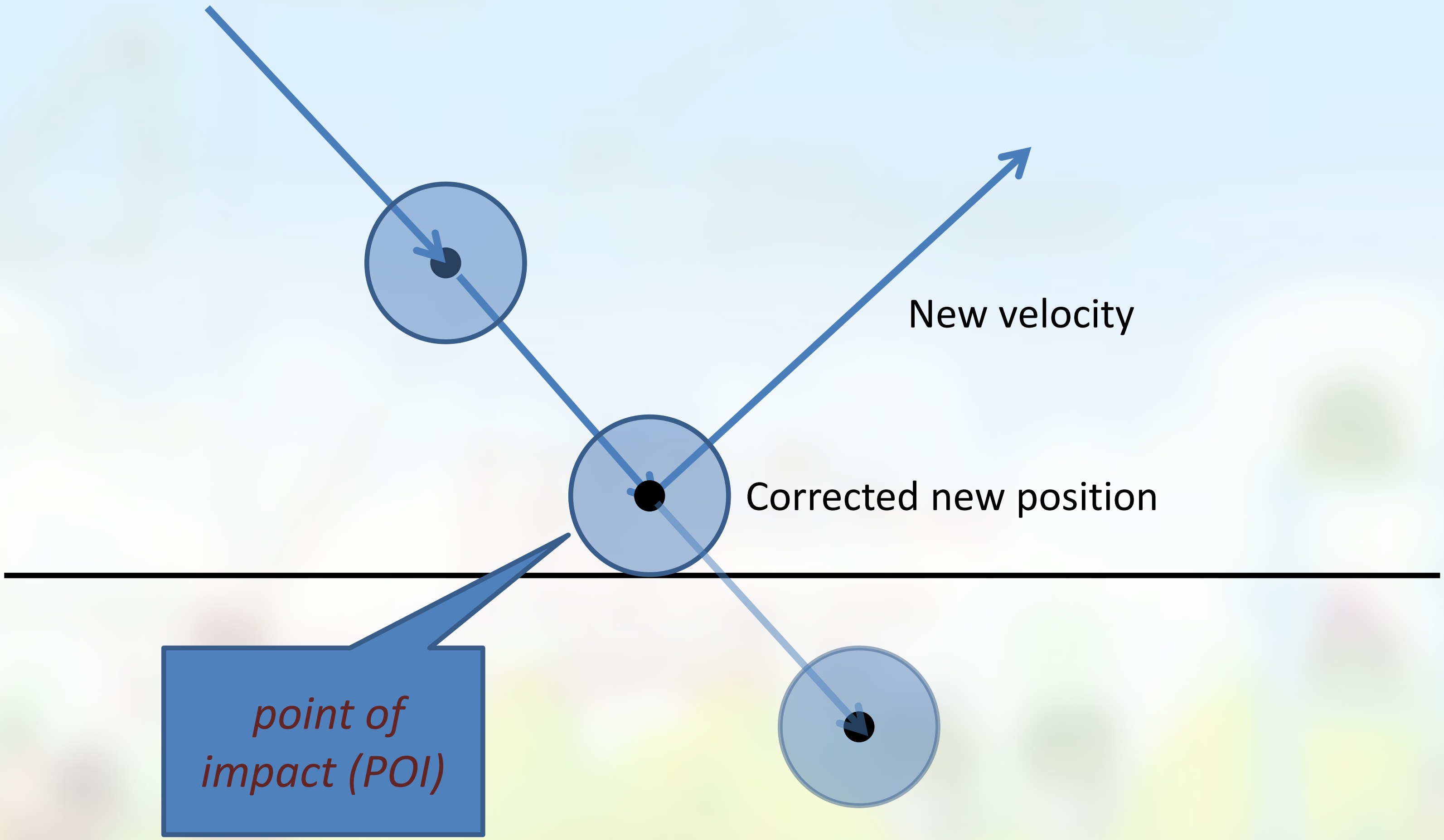
- Recall: general setup = Euler integration:
  - Change *position* by adding *velocity* each frame:

    newPosition = oldPosition + velocity
- *Collision detection:*
  - Does the moving object hit another object this frame? Which object? From which side?
- *Collision resolve:*
  - Correct the new position (no overlap with other obj)
  - Change velocity (=bounce away)

Collision detection



Collision resolve



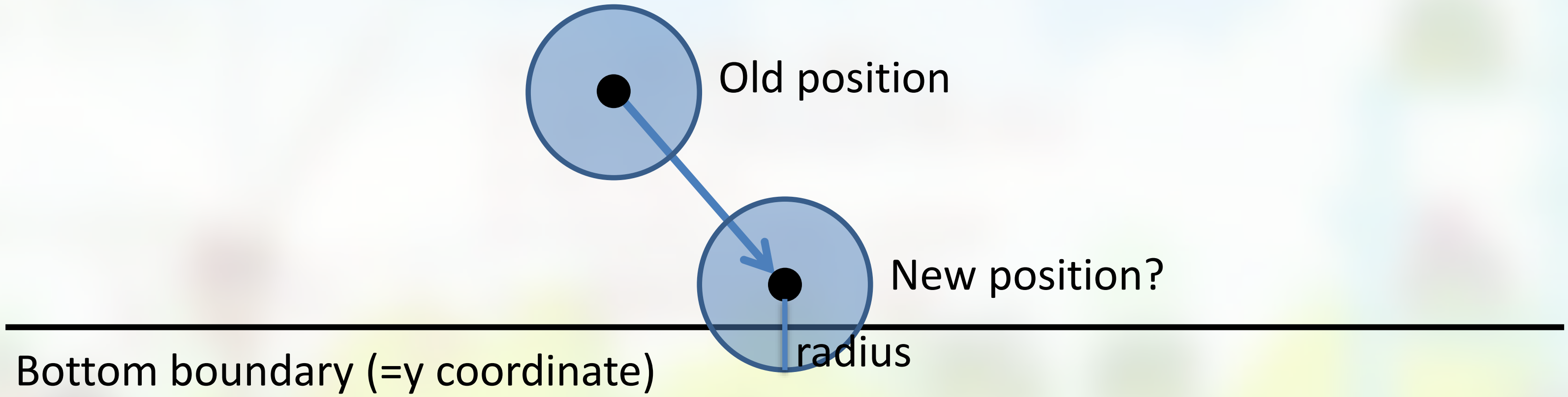
Discrete vs Continuous Collision Detection

- *Discrete collision detection*:
  - At the *new position*, does the object overlap with another object?
- *Continuous collision detection*:
  - When moving from oldPosition to newPosition, does the object collide with another object?
- Discrete is easier to implement.
- Discrete can give errors with high velocities: *tunneling (demo).*
- For assignment 3, discrete is sufficient.

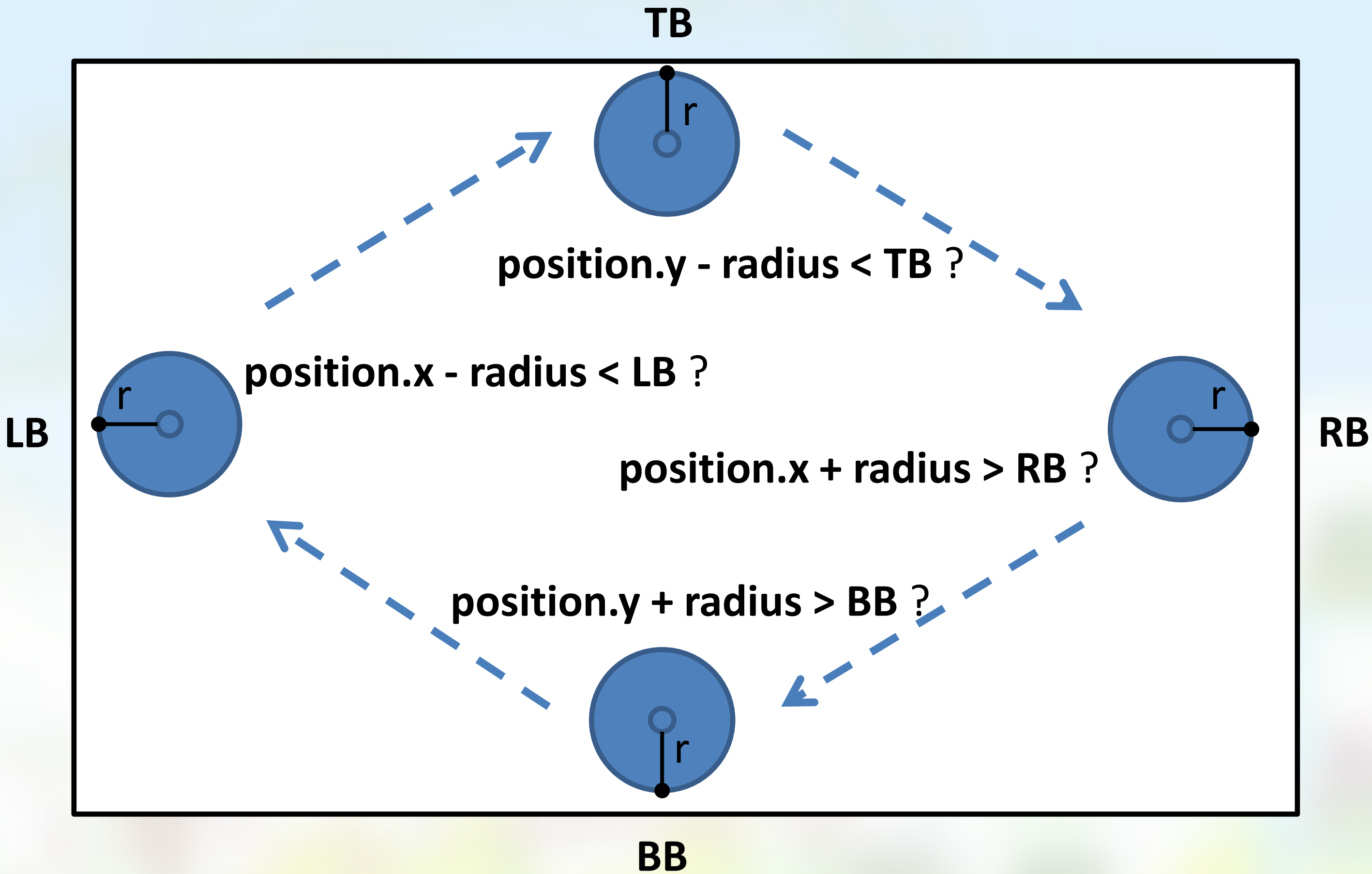
Boundary collision detection

- Collision with bottom (GXP coords) if:

  newPosition.y + radius >= BottomYBoundary
- Collision with top/left/right boundary: similar – see +001_bouncing_squares_setup



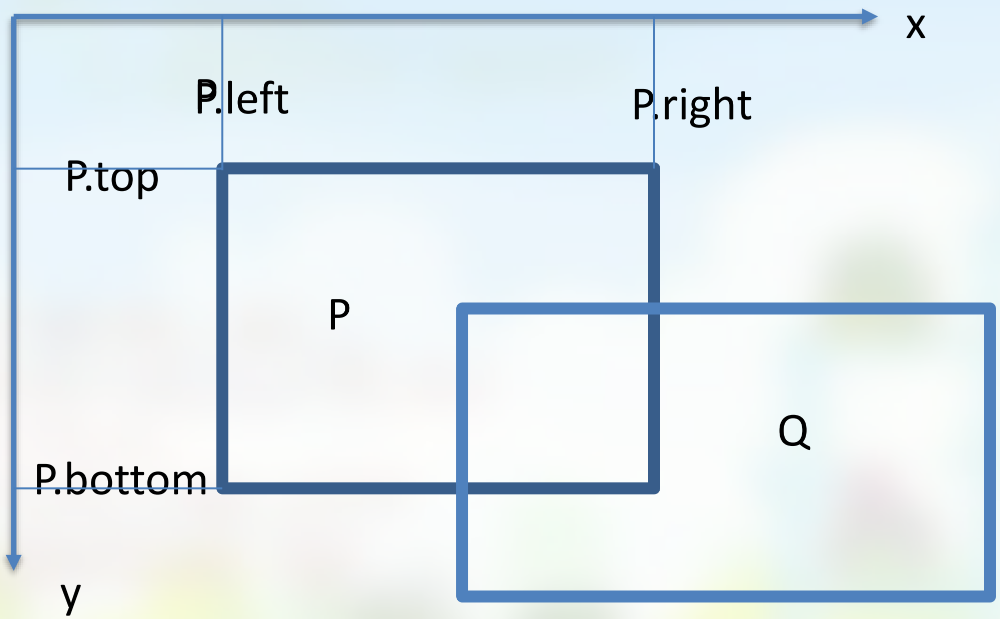
Boundary collision detection



TP, RB, BB, LB = Top, Right, Bottom, Left Boundary
+001_bouncing_squares_setup

AABB collision detection

- *AABB: axis-aligned bounding box* (= rectangle, not rotated).
- Observe: AABBs *P* and *Q* overlap if *all* of the following hold:
  - P.right>Q.left
  - P.left<Q.right
  - P.top<Q.bottom
  - P.bottom>Q.top



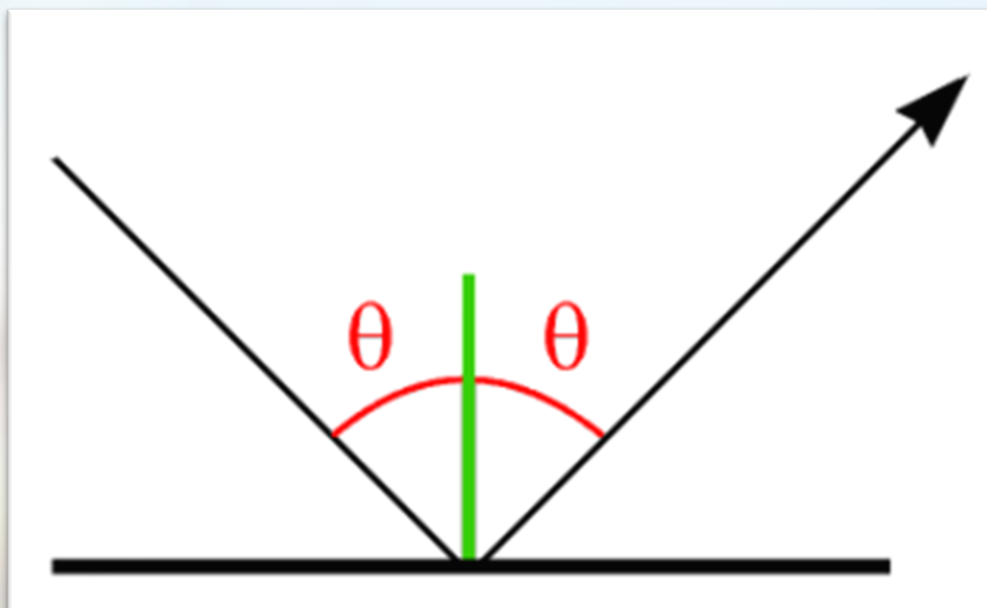
(GXP coords, so *top* means minimum y!)

Collision Resolve I: Velocity *Reflection*

Angle of incidence is the angle of reflection

For horizontal/vertical boundaries that means:

negate velocity in x or y direction



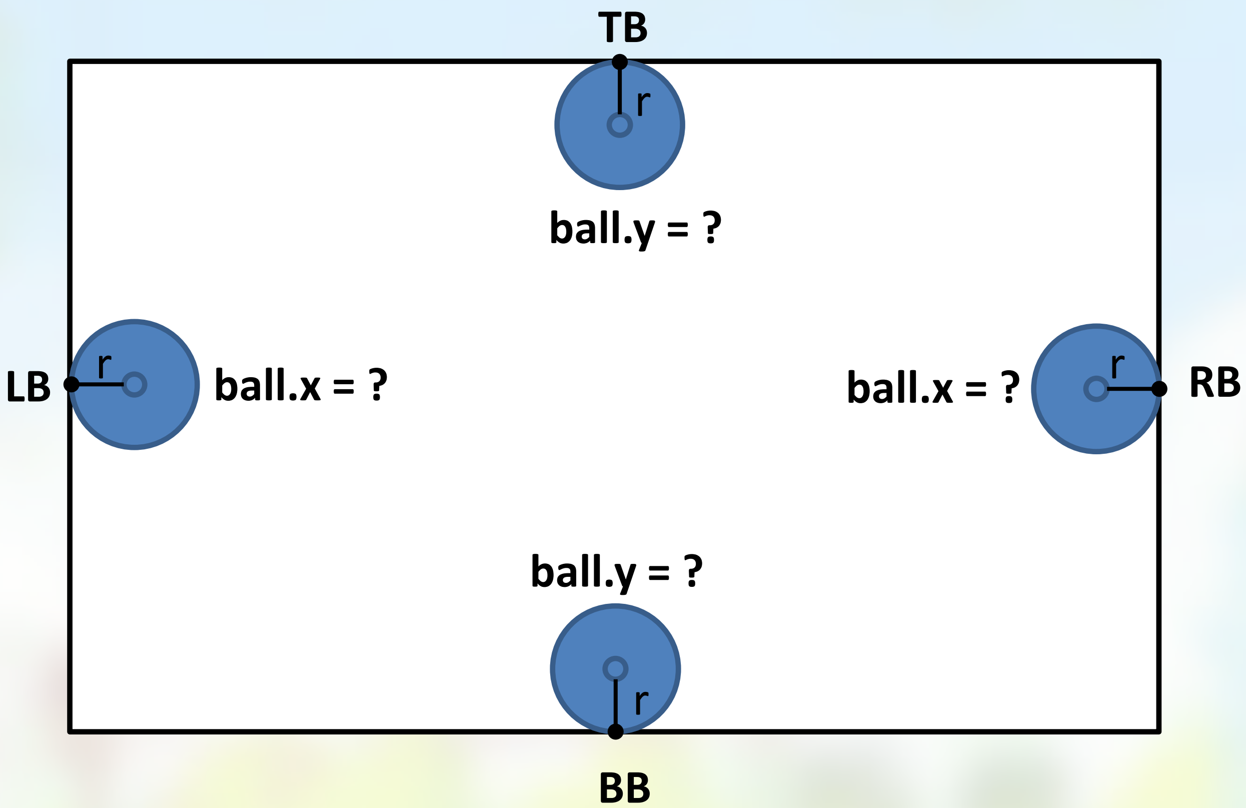
Bounce elasticity

- Perfectly elastic collision:
  - newVelocity.y = -oldVelocity.y
  - Reflection is 100% :
    - *speed remains the same*
    - *So same kinetic energy:* $m \cdot |\vec{v}|^2 / 2$
- Non perfect elastic collision:
  - newVelocity.y = -bounciness * oldVelocity.y
  - Bounciness is between 0 and 1, denoted by *C*.

    (More formal term: *Coefficient of reflection)*
  - Physics explanation: if bounciness<1:
    - speed decreases.
    - some kinetic energy transforms to other types of energy (sound, heat).
  - (demo)

Collision Resolve II: Reset Position

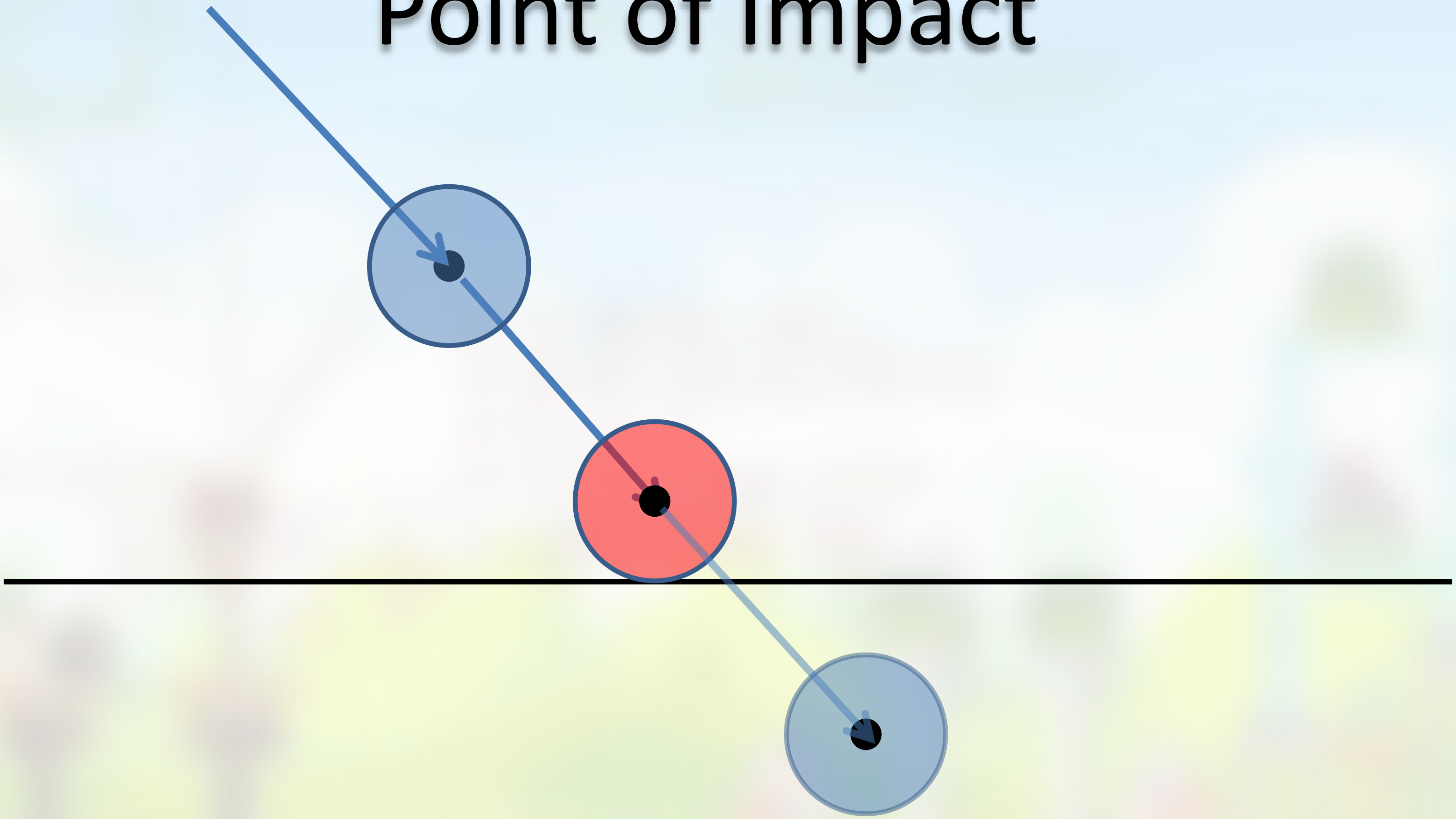
Position has to be reset onto boundary



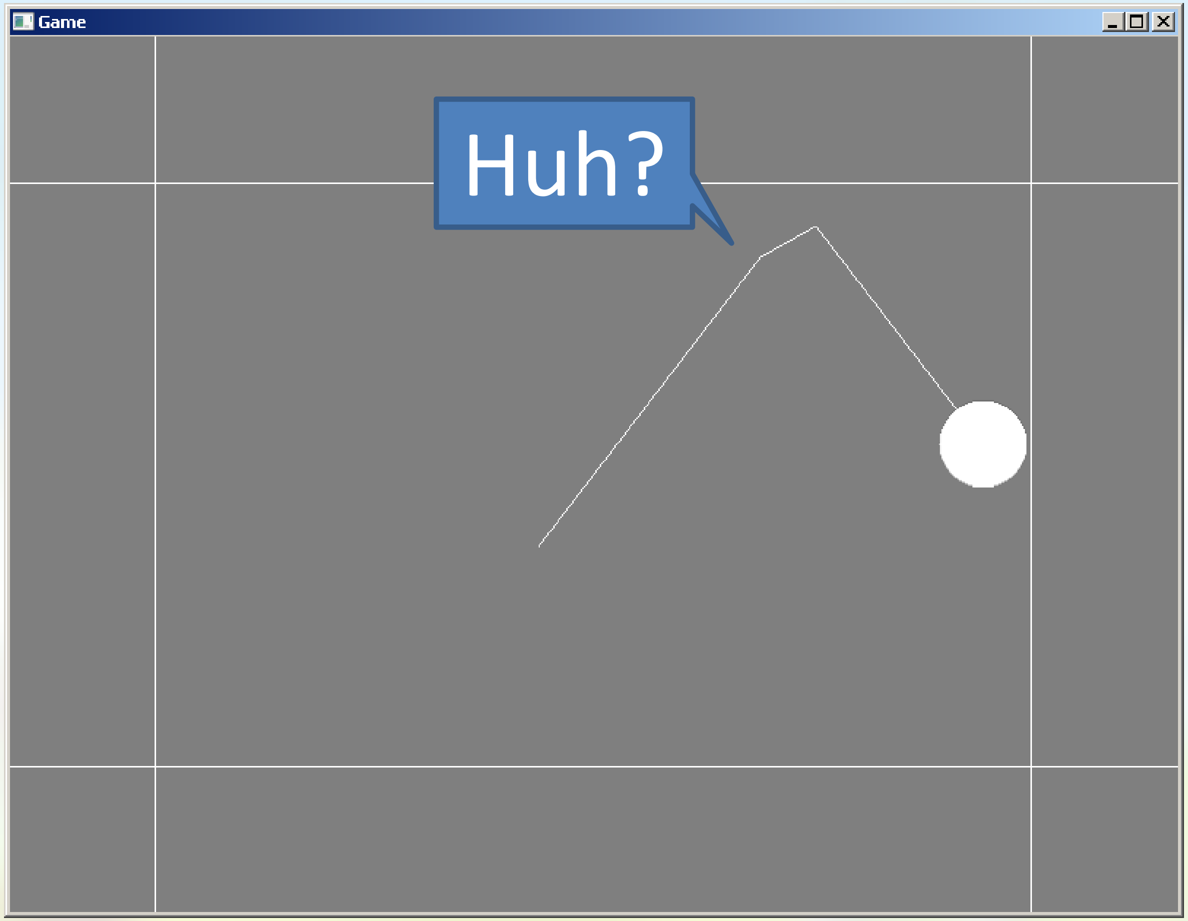
Debug Options

- To study exactly what happens, debug options have been added to the starting code:
  - R resets to starting position
  - P pauses
  - Spacebar slows framerate to 5 FPS
  - D toggles drawing trajectory
  - C clears all trajectory drawings
  - …more: see information printed to console.

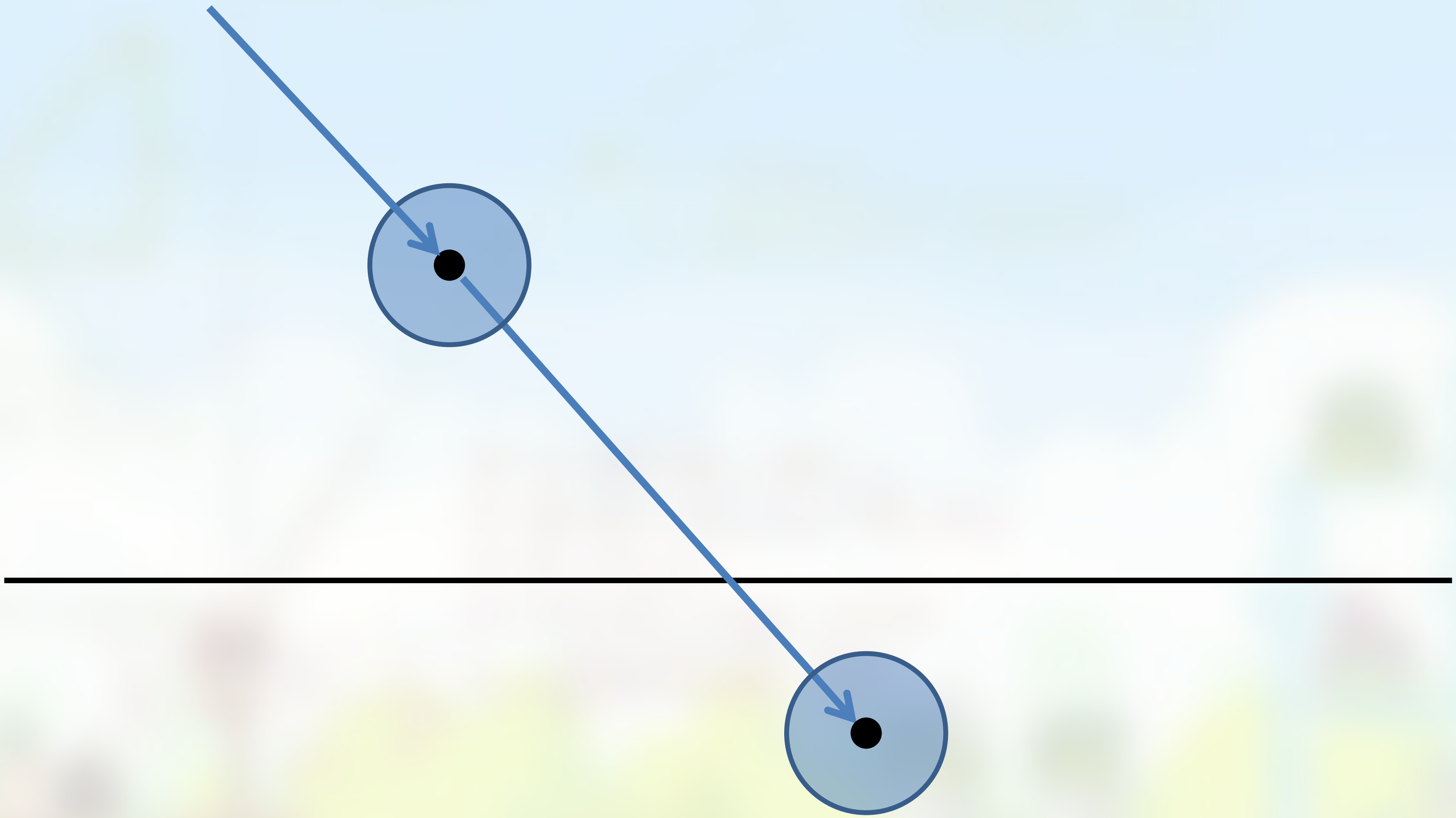
Point of Impact



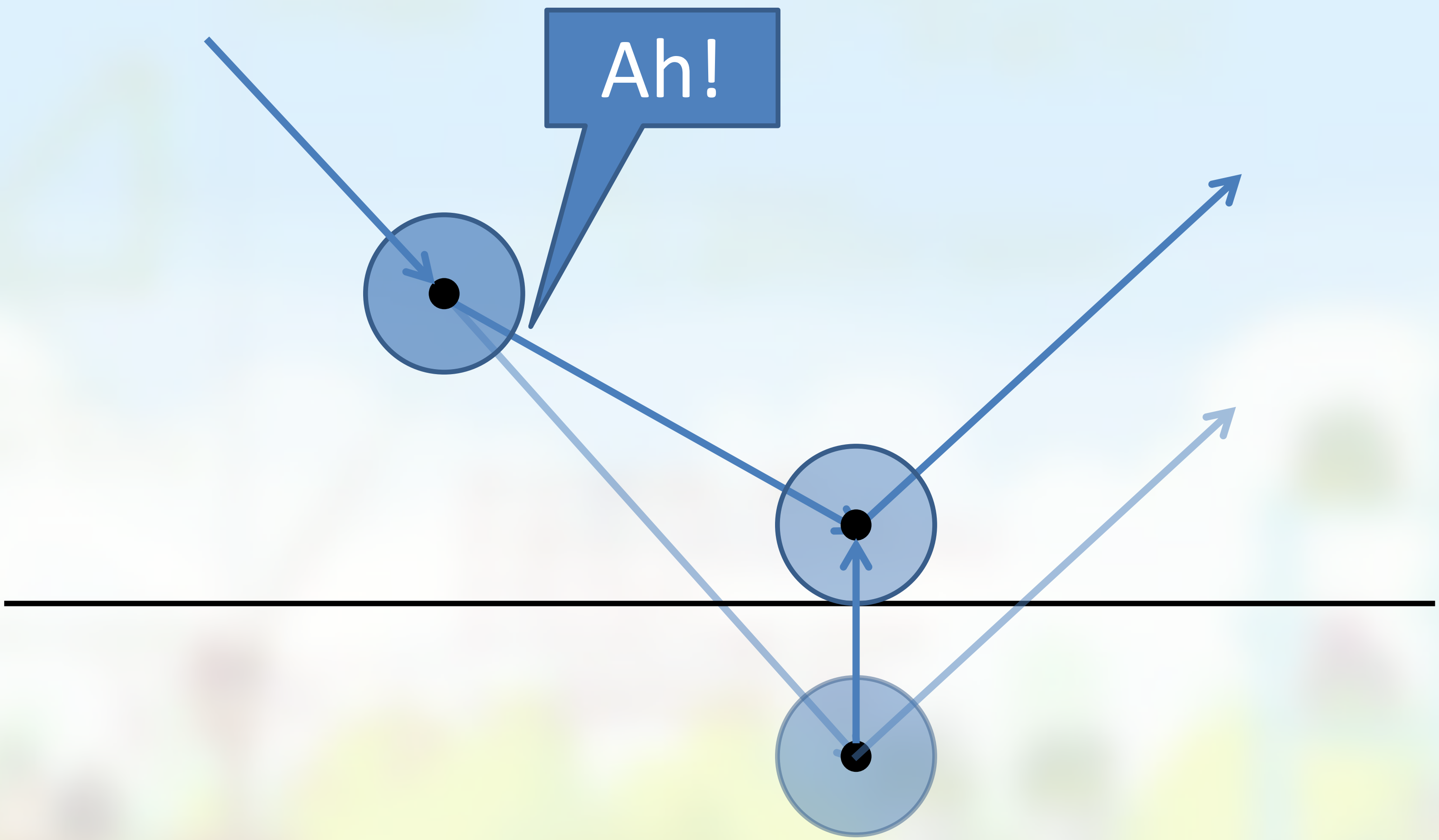
Why is the angle “crooked” ?



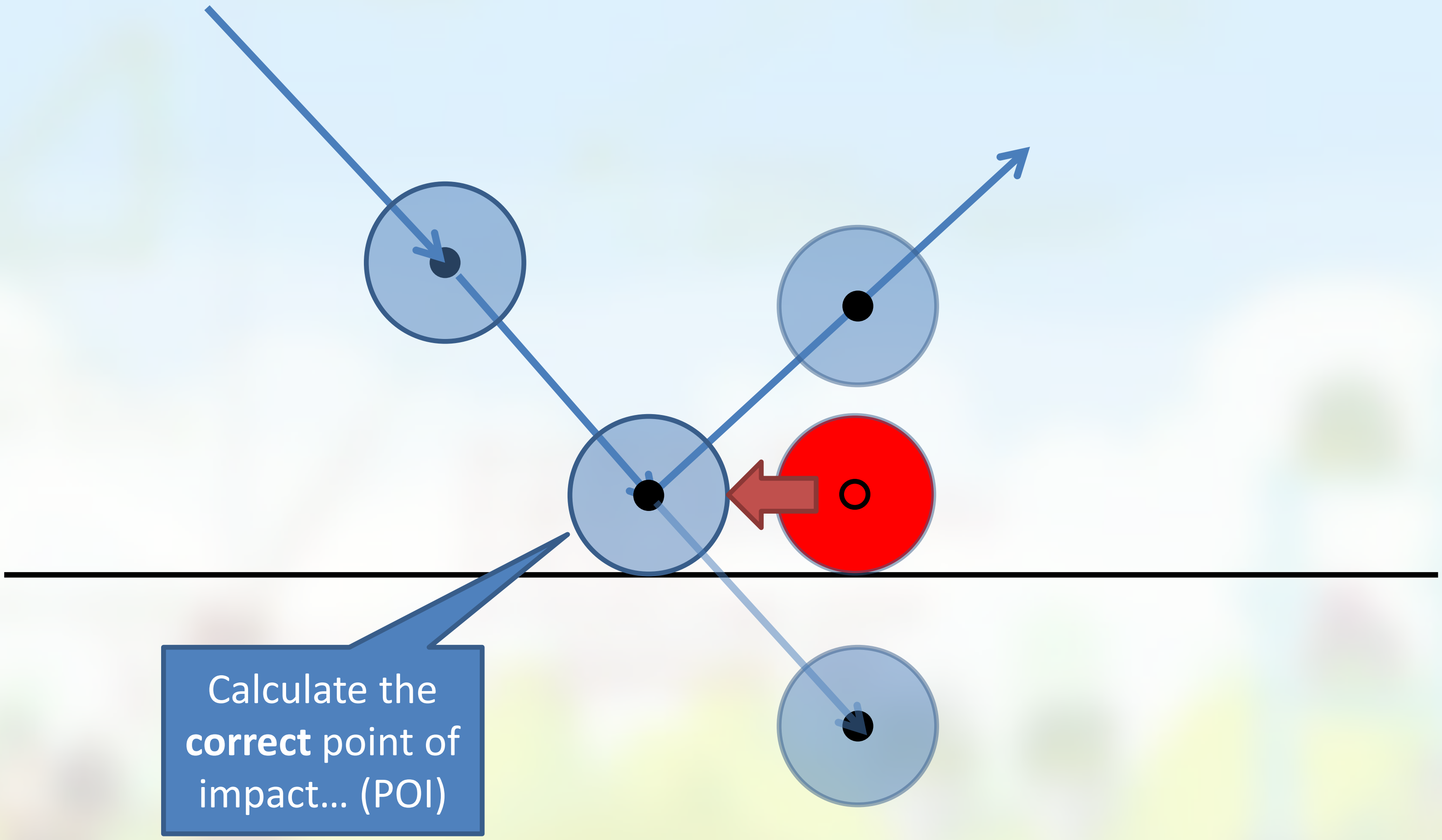
This is what we are trying to fix…



And we do it like this:

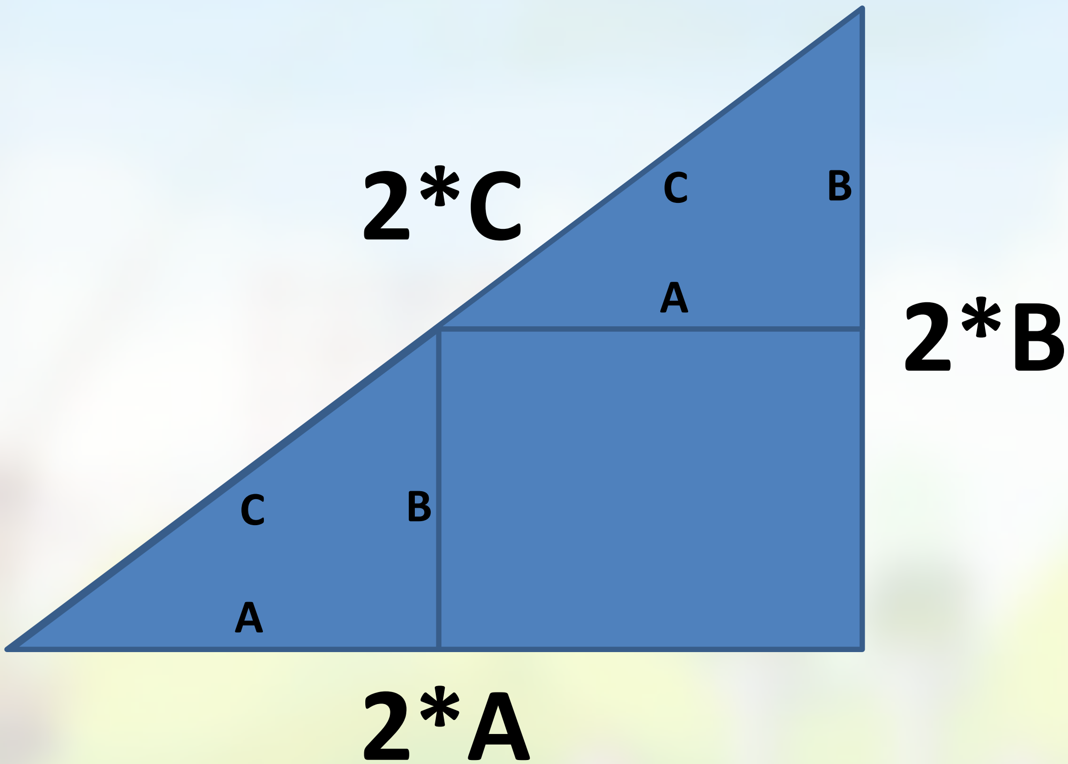


Correct approach would be this:



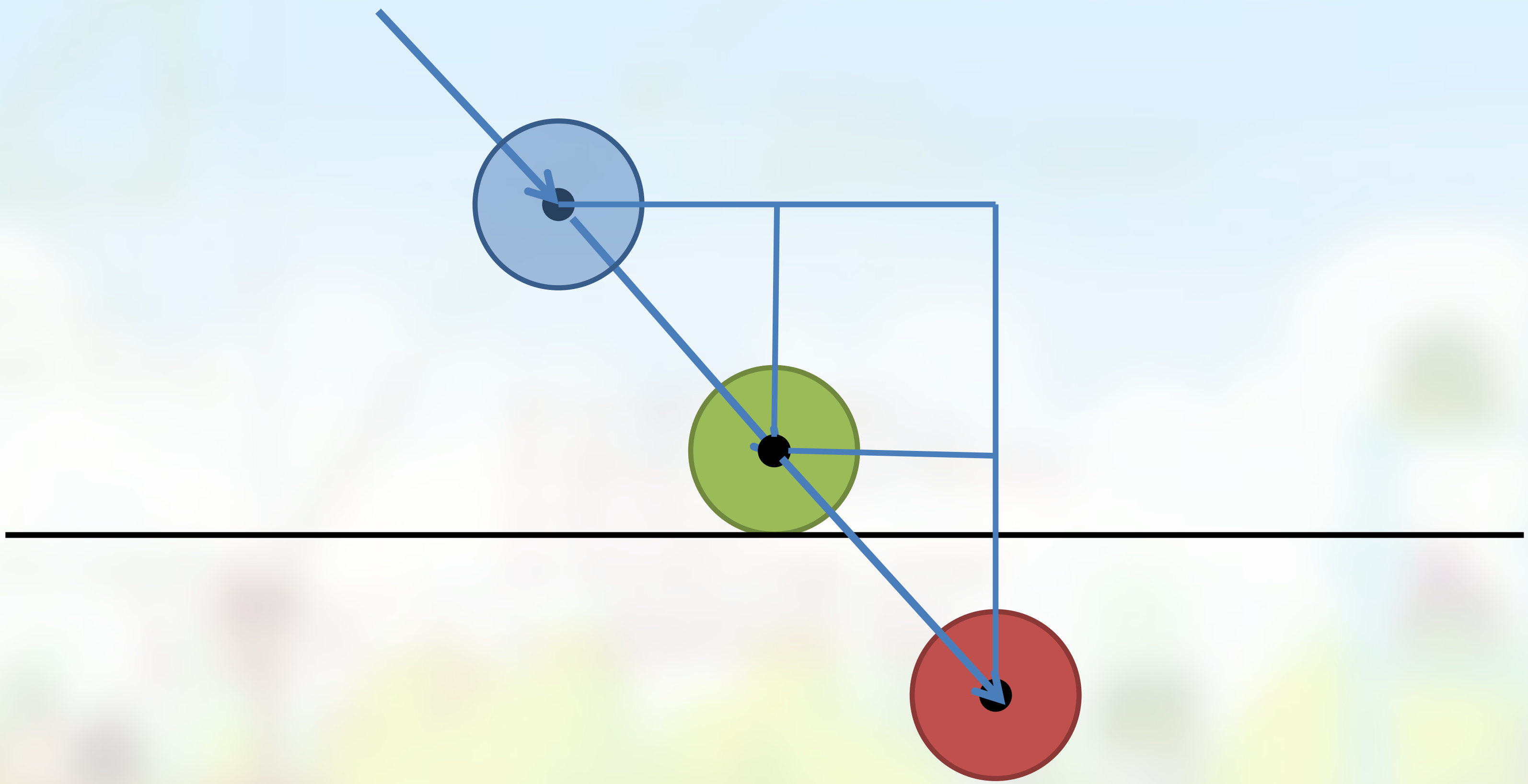
Solution: use triangle ratios

When proportionally scaling a triangle, all sides grow/shrink proportionally



How does that apply to this problem?

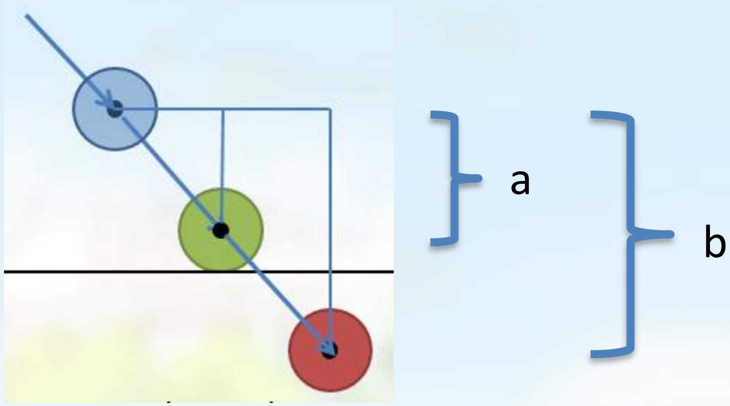
What is known and what is not? Where are the ratios?



(demo)

Point of Impact (POI) Calculation

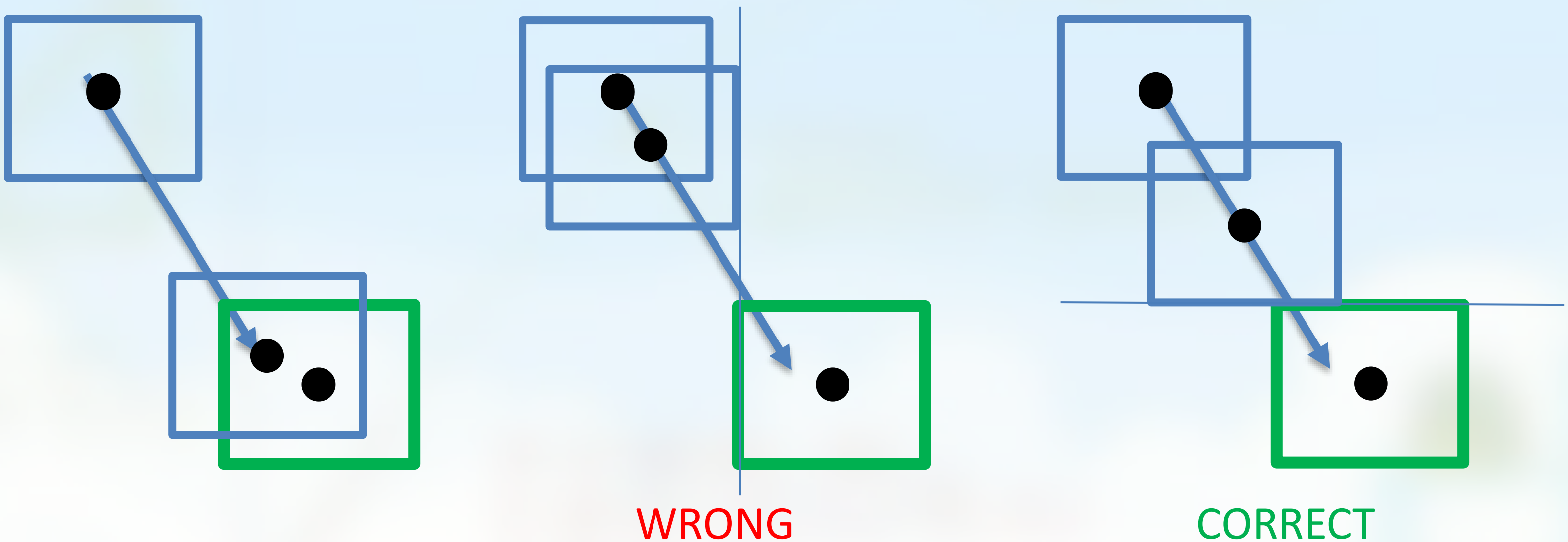
- Consider three values:
  - oldPosition.y →
  - ImpactY →
  - newPosition.y →



- 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

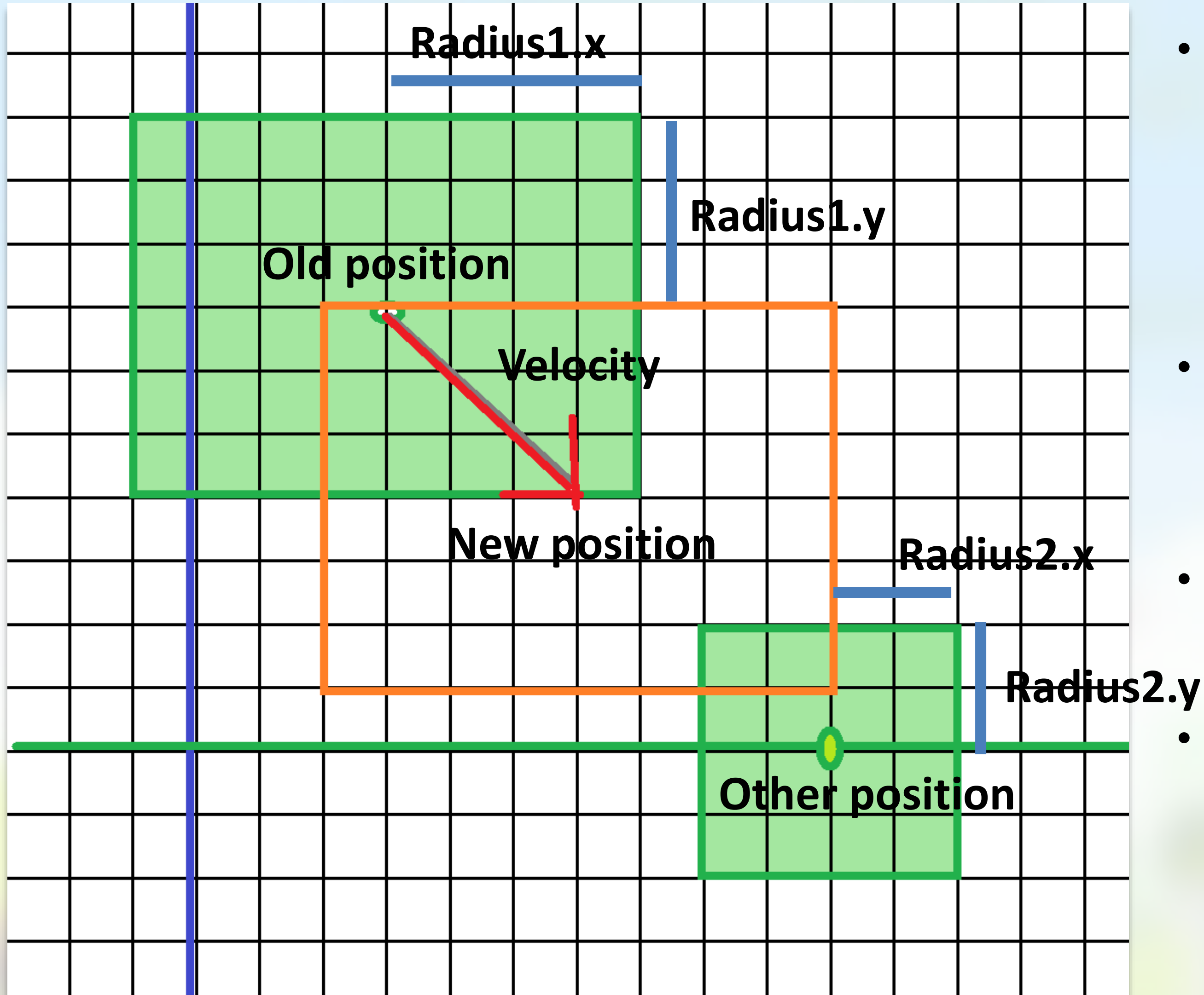
Point of Impact for AABBs

- What is the point of impact here?



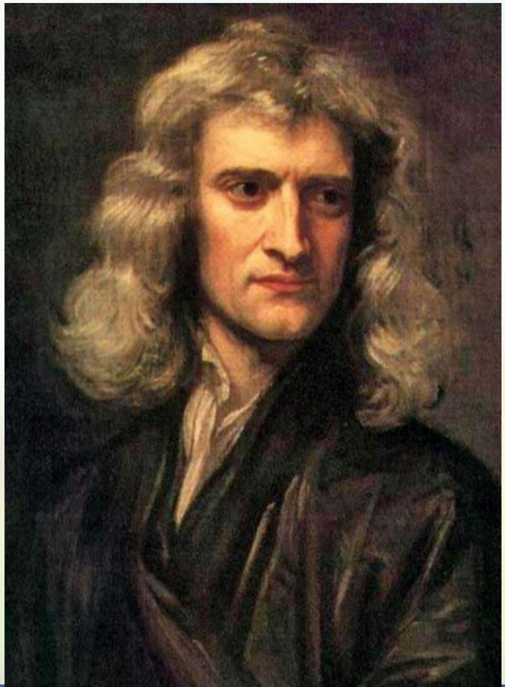
- (demo)
- Solving this puzzle is part of Assignment 3.3. (Note: you may not use the GXPEngine's MoveUntilCollision method!)

Assignment 3.3 help



- What is the distance between right (bottom) of the moving block to the left (top) of the other block, before moving?
- Can you express that using *old position*, *other.position*, *radius1* and *radius2*?
- What is it after moving? (Should be negative, since overlap!)
- Can you express the *time of impact* using these values? (And *velocity.x* / *velocity.y*)

Newton's Laws of Motion



Newton's laws of Motion

Law 1: Law of Inertia

A body moves at constant velocity unless an external force is applied to it.

Law 2: Fundamental Law of Dynamics

The *acceleration* of an object equals the total *force* acting on it, divided by its *mass*. ($a = F / m$ or $F = m \cdot a$)

Law 3: Law of Reciprocal Actions ("action = -reaction")

When a body exerts force on a second body, the second body exerts an equal opposite force upon the first body

Game Physics Simplification

- Let's study Law 1 and 2 first
- Position only changes through velocity
- Velocity only changes **through acceleration**

→This leads to the updated code loop (Euler integration with acceleration) that was shown at the end of last week's lecture.

(For convenience, those slides are included again.)

Updating our code loop:

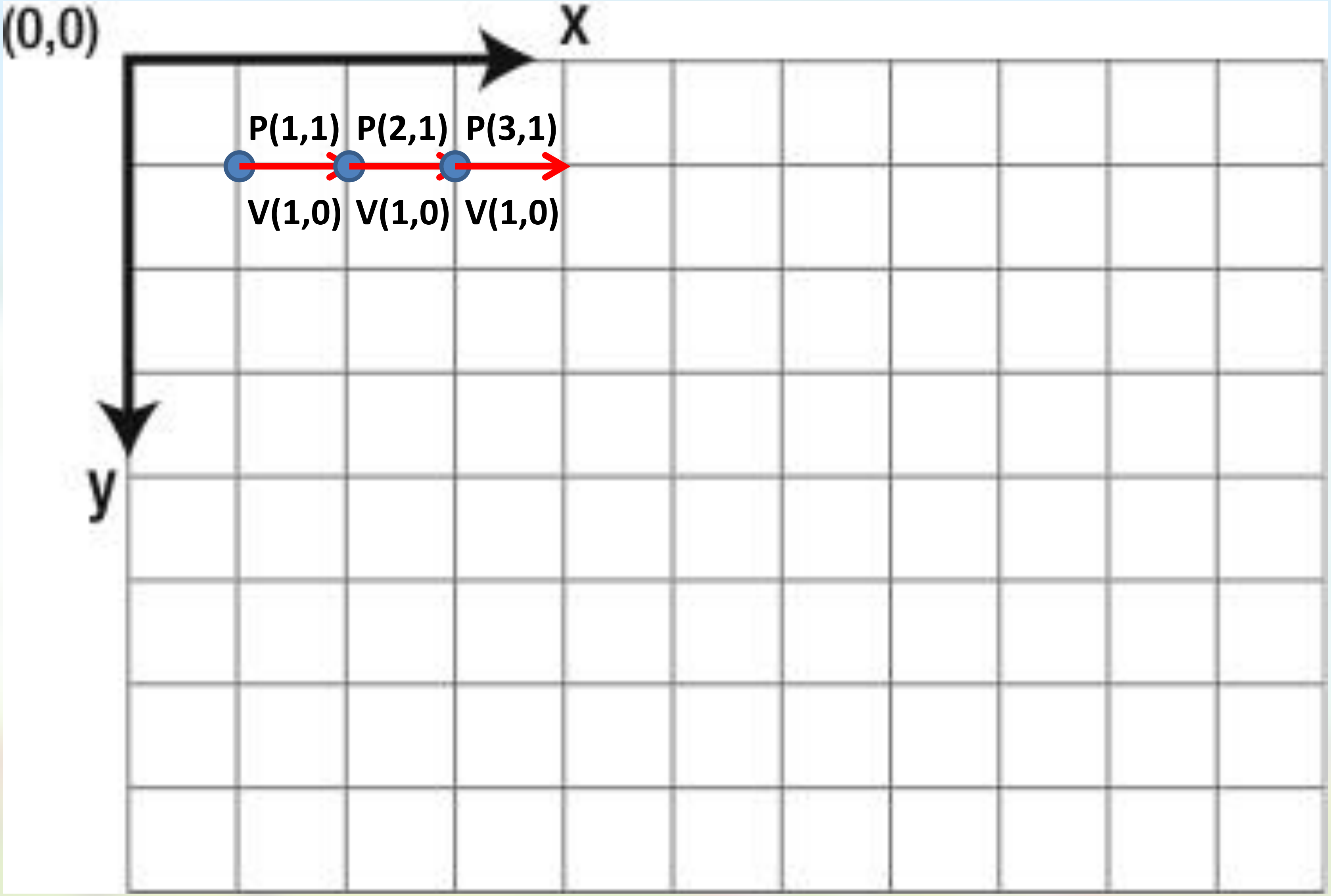
Applying *Law 1:*

$$velocity = \ \ldots some\ value\ \ldots$$
$$position = position + velocity$$

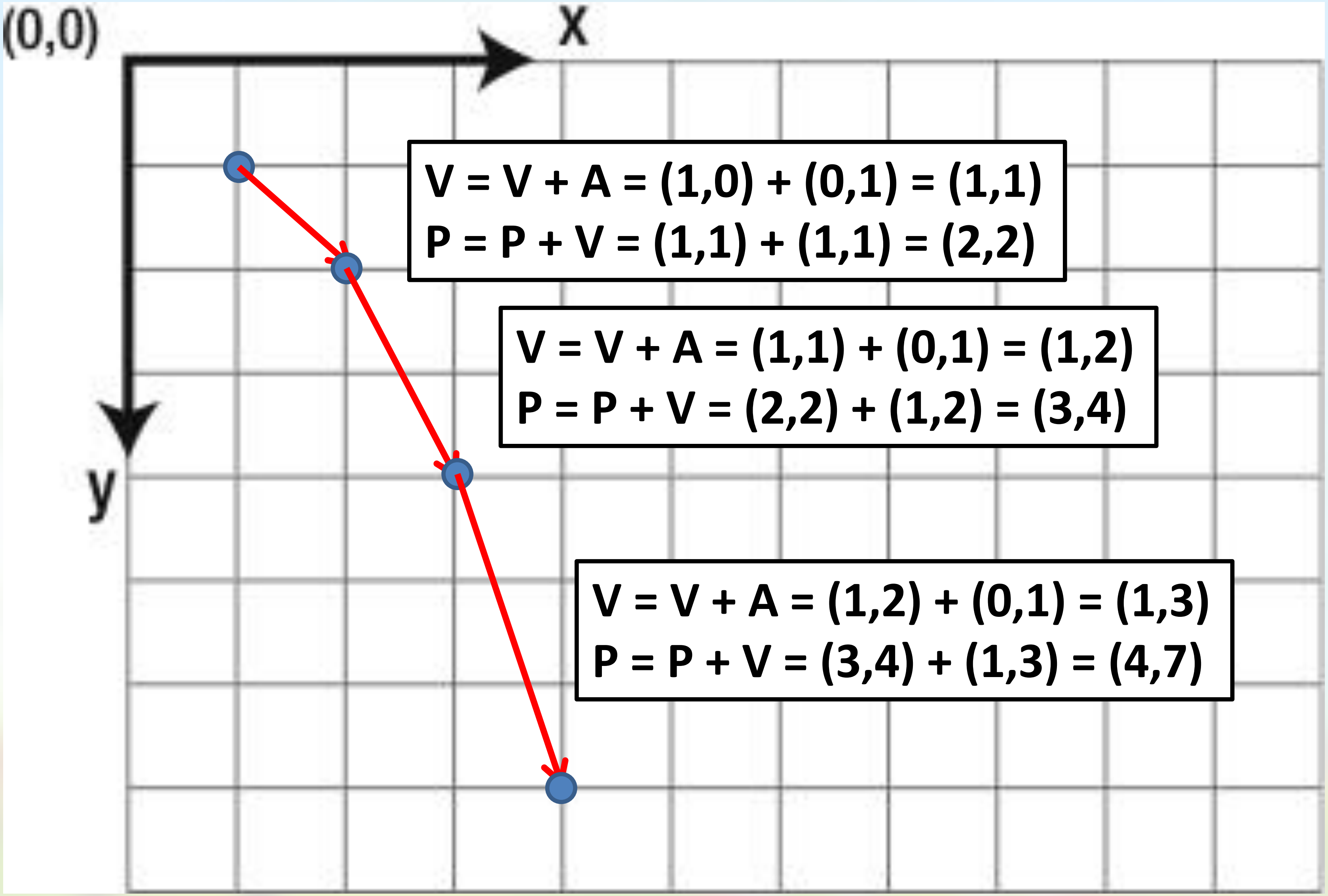
becomes

$$acceleration = \ \ldots some\ value\ \ldots$$
$$velocity = velocity + acceleration$$
$$position = position + velocity$$

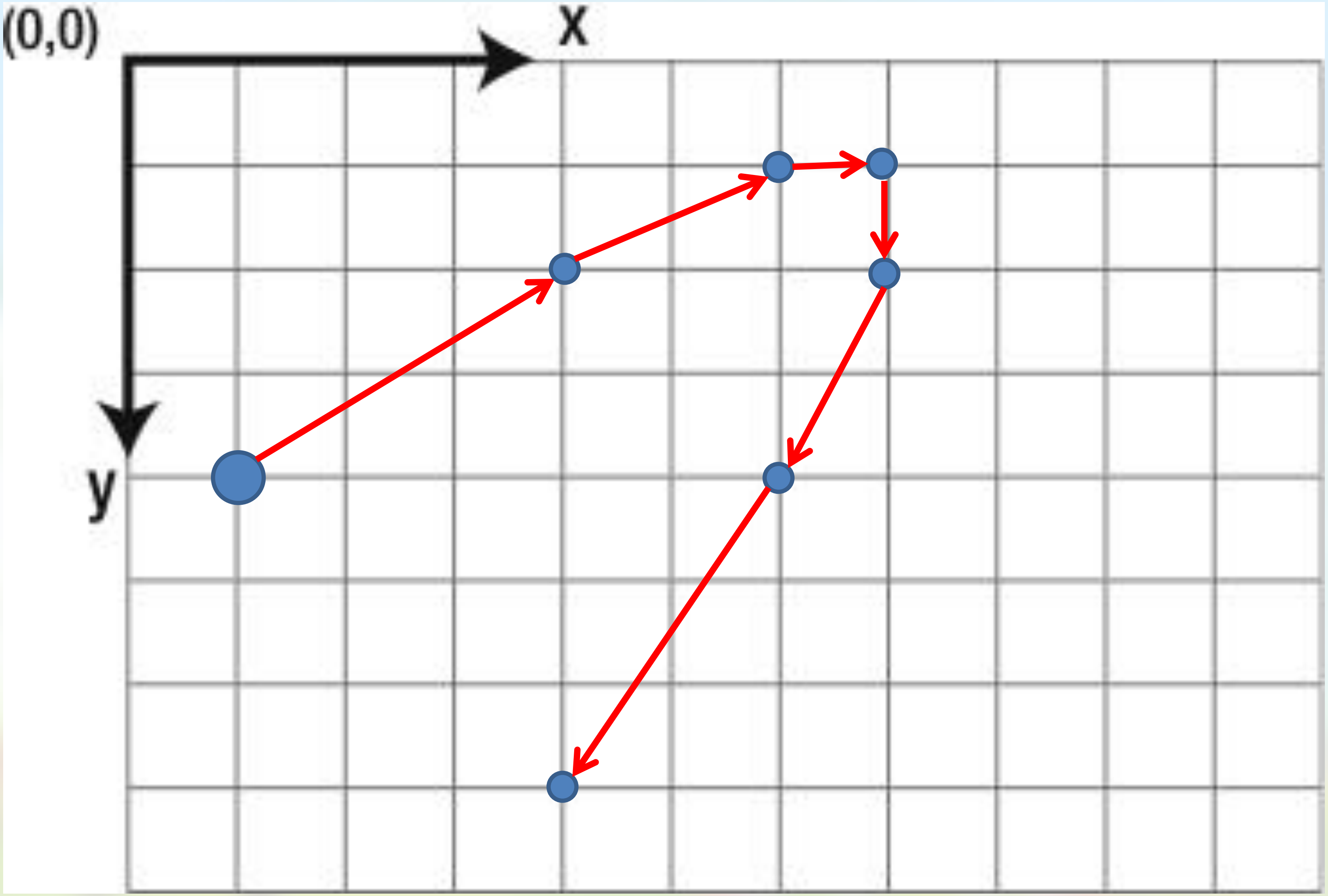
P = (1,1), V = (1,0), A = (0,0)



P = (1,1), V = (1,0) , A = (0,1)


(0,0)
x
y
V = V + A = (1,0) + (0,1) = (1,1)
P = P + V = (1,1) + (1,1) = (2,2)
V = V + A = (1,1) + (0,1) = (1,2)
P = P + V = (2,2) + (1,2) = (3,4)
V = V + A = (1,2) + (0,1) = (1,3)
P = P + V = (3,4) + (1,3) = (4,7)

P = (1,4), V = (4,-3) , A = (-1,1)



Integration

$$acceleration = \ \ldots some\ value \ldots$$
$$velocity = velocity + acceleration$$
$$position = position + velocity$$

- This core step is also called "semi-implicit Euler integration"
- It is just an *approximation* of the true movement! (Piecewise linear instead of parabola)
- There are many alternatives, see e.g. https://gafferongames.com/post/integration_basics/
- …but this is the best choice for now.

Law 2: Forces and Mass

Some examples of forces in games:

- *Gravity:* force is proportional to mass, so *acceleration* is constant. (Independent of mass!)
- *Spaceship thruster:*
  - Normal engine, big ship: slow acceleration
  - Normal engine, small ship: fast acceleration
- *Viscosity/drag* (="fluid/air friction"): proportional to velocity^2 & area. (Therefore a feather falls slower than a stone.)

Applying multiple forces

- We can add forces (D’Alembert’s principle):
  - Cooperating forces strengthen each other
  - Opposing forces cancel each other out
- In other words:
  - The total force applied to an object is the sum of all forces (sounds logical right?)
  - Similarly: The total acceleration of an object is the sum of all accelerations.
- (Remember: these are all *vectors.*)

- What about Newton's third Law?
  (Reciprocal actions / action=-reaction)

- We need this to resolve the collision when two moving objects hit each other!
- Example: Newton Ball Tricks:
  https://www.youtube.com/watch?v=JadO3RuOJGU

- Demo: bouncing squares; with/without Newton

Momentum

Resolving Collisions: two moving bodies

- During a collision:
  - For a *very short time,* two colliding bodies exert a *very high force* upon each other.
- How long? What force?

→ A better way to analyze this is using *momentum:*

- The *momentum* of a body is $m \cdot \vec{v}$

  (Note: this is again a *vector: mass m* is a scalar, *velocity* $\vec{v}$ is a vector.)

- The *(total) momentum* of a *system* of bodies is the *sum* of their momentums.

Conservation of Momentum

Theorem: Conservation of Momentum

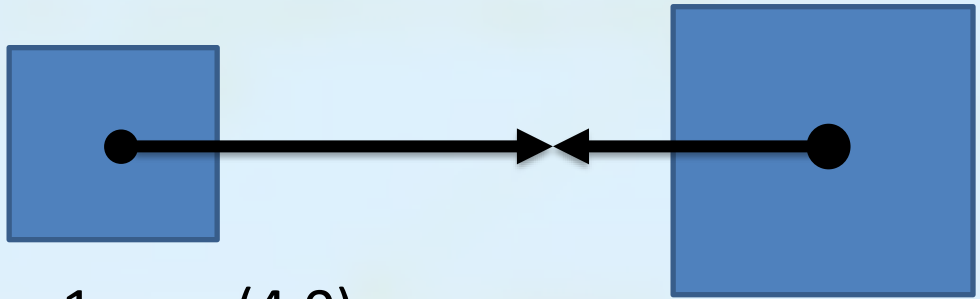
If the total force on a system of bodies is zero, the total momentum remains constant.

This theorem can be deduced from Newton's three laws.

Details:

https://www.physicsclassroom.com/class/momentum/Lesson-2/Momentum-Conservation-Principle

Example



$m_1 = 1$, $v_1 = (4,0)$

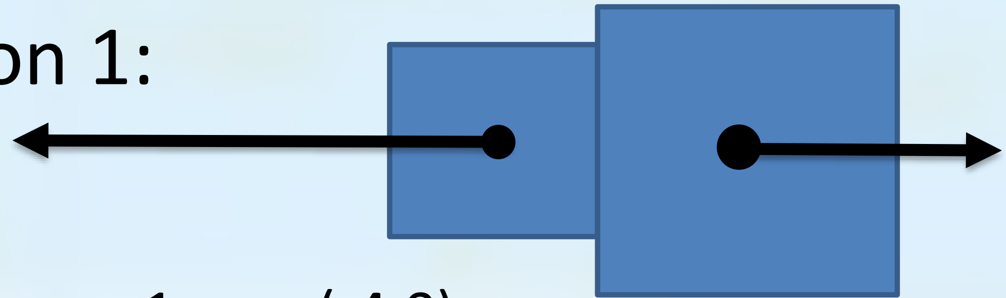
$m_1 = 2$, $v_2 = (-2,0)$

- What are the resulting velocities?

  Momentum1 = (4,0), Momentum2 = (-4,0) ➔ total momentum = (0,0)

Example

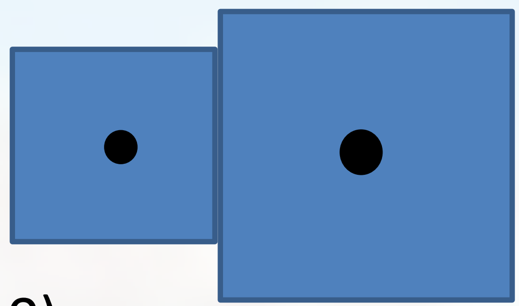
- Solution 1:



$m_1 = 1, v_1 = (-4,0)$

$m_1 = 2, v_2 = (2,0)$

- Solution 2:



$m_1 = 1, v_1 = (0,0)$

$m_1 = 2, v_2 = (0,0)$

- Both solutions are possible, and have the same momentum! Which one is it?

Collisions on a Line - Easy Case

- Both solutions are possible, and have the same momentum. Which one is it?

A: Depends on *bounciness C* *(a.k.a. Coefficient of Reflection)*

  – *Solution 1: all kinetic energy is preserved: C = 1.*
  – *Solution 2: all kinetic energy dissipates: C = 0.*

Conclusion:

If the total momentum is zero, then the resulting velocities are $v'_i = -C \cdot v_i$ (for i=1,2).

- Deducing the new velocities in this case seems easy enough. But what to do when the total momentum is not zero?

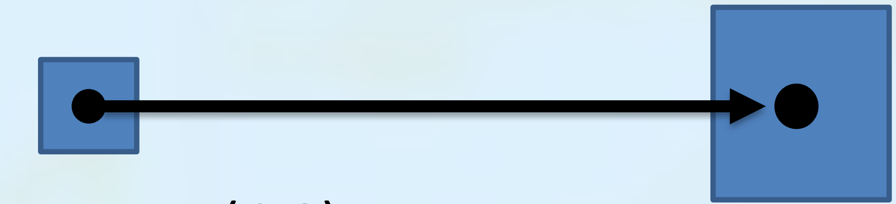
A: Change your *frame of reference!* Consider the velocities relative to the *velocity of the center of mass.*

- This is the *weighted average* of velocities, with weight = mass. More precisely:
- *Velocity of center of mass (=vector)*:

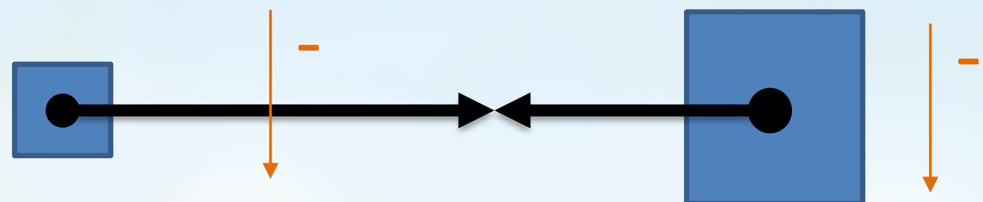
$$u = (m_1 \cdot v_1 + m_2 \cdot v_2) / (m_1 + m_2)$$

Example

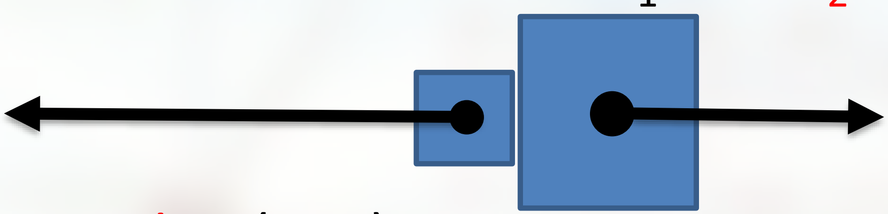
	Total momentum:	CoM velocity:



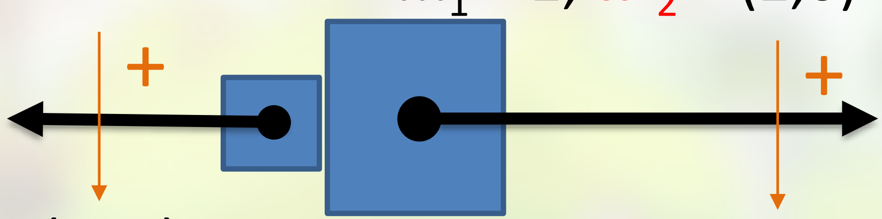
$m_1 = 1$, $v_1 = (6,0)$ $m_1 = 2$, $v_2 = (0,0)$ (6,0) (2,0)



$m_1 = 1$, $w_1 = (4,0)$ $m_1 = 2$, $w_2 = (-2,0)$ (0,0) (0,0)



$m_1 = 1$, $w'_1 = (-4,0)$ $m_1 = 2$, $w'_2 = (2,0)$ (0,0) (0,0)



$m_1 = 1$, $v'_1 = (-2,0)$ $m_1 = 2$, $v'_2 = (4,0)$ (6,0) (2,0)

Collisions on a Line – Full Formula

If two bodies collide on a line, where:

- v_i is the original velocity (i=1,2)
- u is the velocity of the center of mass
- C is the bounciness
- v'_i is the *resulting velocity* (i=1,2)

Then:

$$\boxed{v'_i = u - C \cdot (v_i - u)}$$

(Proof sketch: see end of slides)

- Does the previous formula also apply to collisions with boundaries?

A: *Yes!*

- Since they cannot move, you can consider walls to have *infinite mass.*
- Then the *center of mass velocity* u *is zero*, and the formula is exactly the same as before:

$$v'_i = u - C \cdot (v_i - u) = - C \cdot v_i$$

Practical Application - Blocks

- What if the bodies do not collide on a line? (demo)
- Solution for now: we apply this only to *Axis-Aligned Boxes ("Blocks").*
- In that case, we may *apply the previous formula only to the x-component / y-component of the velocity, and keep the other coordinate unchanged.*
- Upcoming lectures: solving the general case, using *collision normal & velocity projection.*
- *(demo)*

Physics Engine Setup

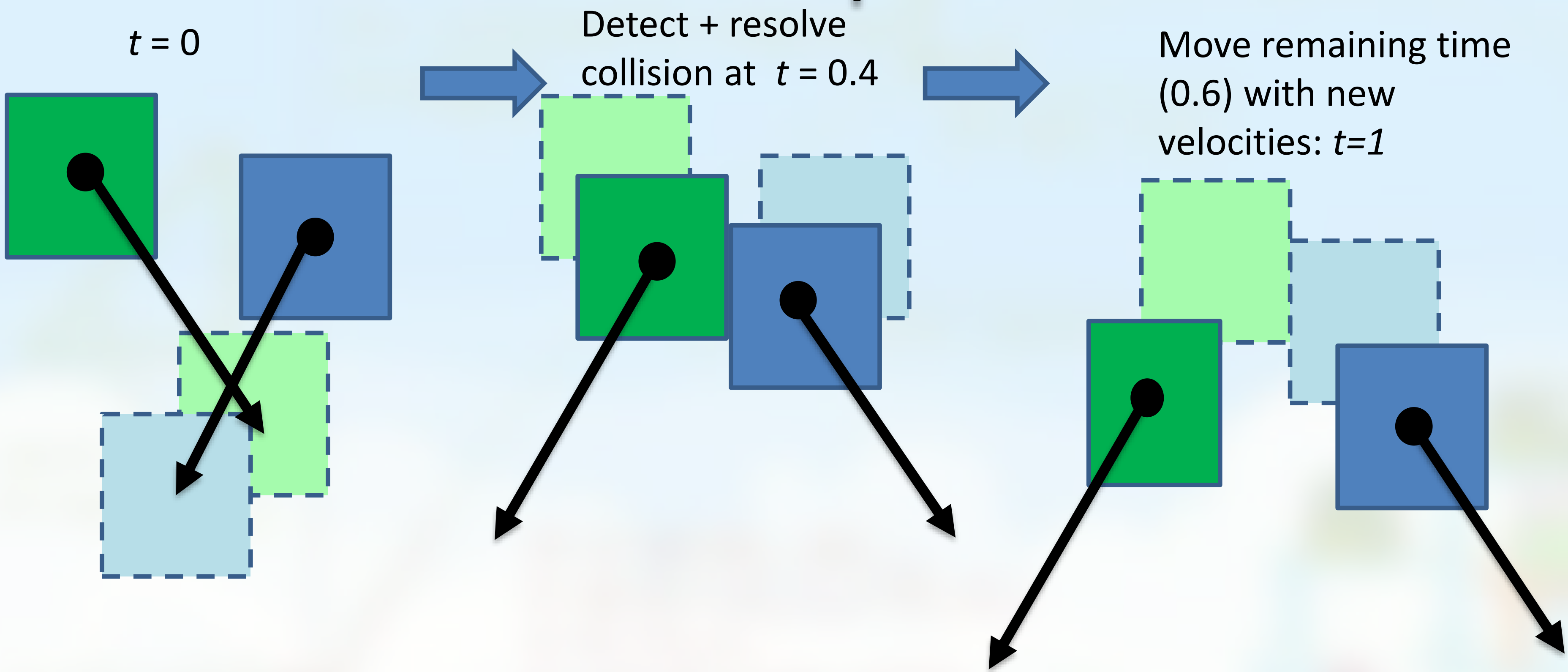
Time Step Methods

Goal: Our physics engine should compute (approximate) the position of all bodies after time Δt has passed.

- We now know enough math to detect and resolve collisions between blocks.
- But how to implement this?
- Bodies "move simultaneously", and collisions happen "halfway during a frame"…

→ Study *time step methods.*

Ideal Time Step Method



Ideal Time Step Method

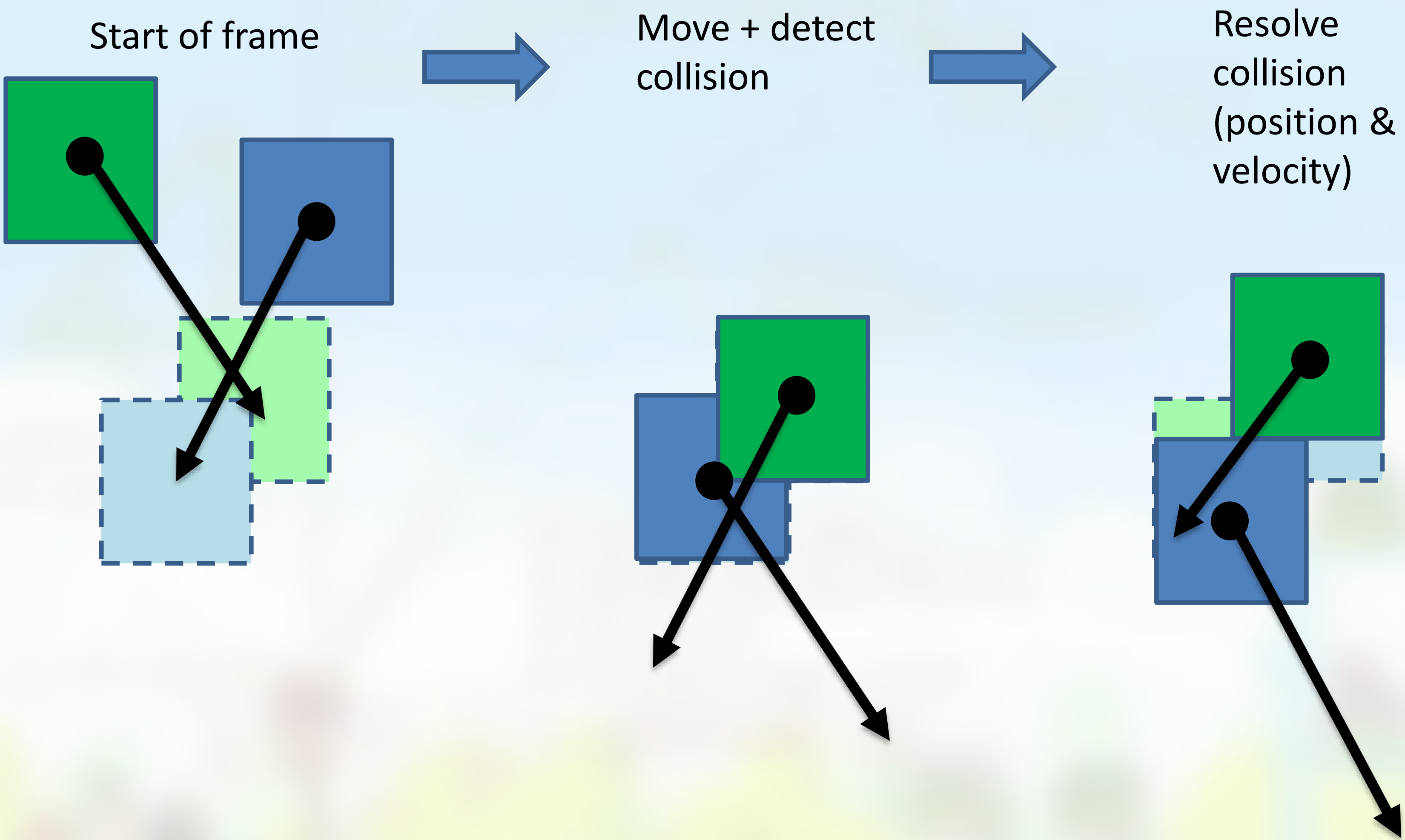
Pseudo code:

> Update velocities using gravity and other forces
> CurrentTime = 0
> While CurrentTime < TargetTime:
>> compute time *t* of earliest collision
>> *t = min (t,* TargetTime)
>> move all bodies over time *t* – CurrentTime
>> resolve possible collisions
>> CurrentTime = *t*
>
> End while

Advantage: very good simulation!

Disadvantage: computationally very expensive! (There's no limit on the number of iterations/collisions!)

Fixed Time Step Method



Fixed Time Step Method

Alternative (=“just do it, clean up mess later”):

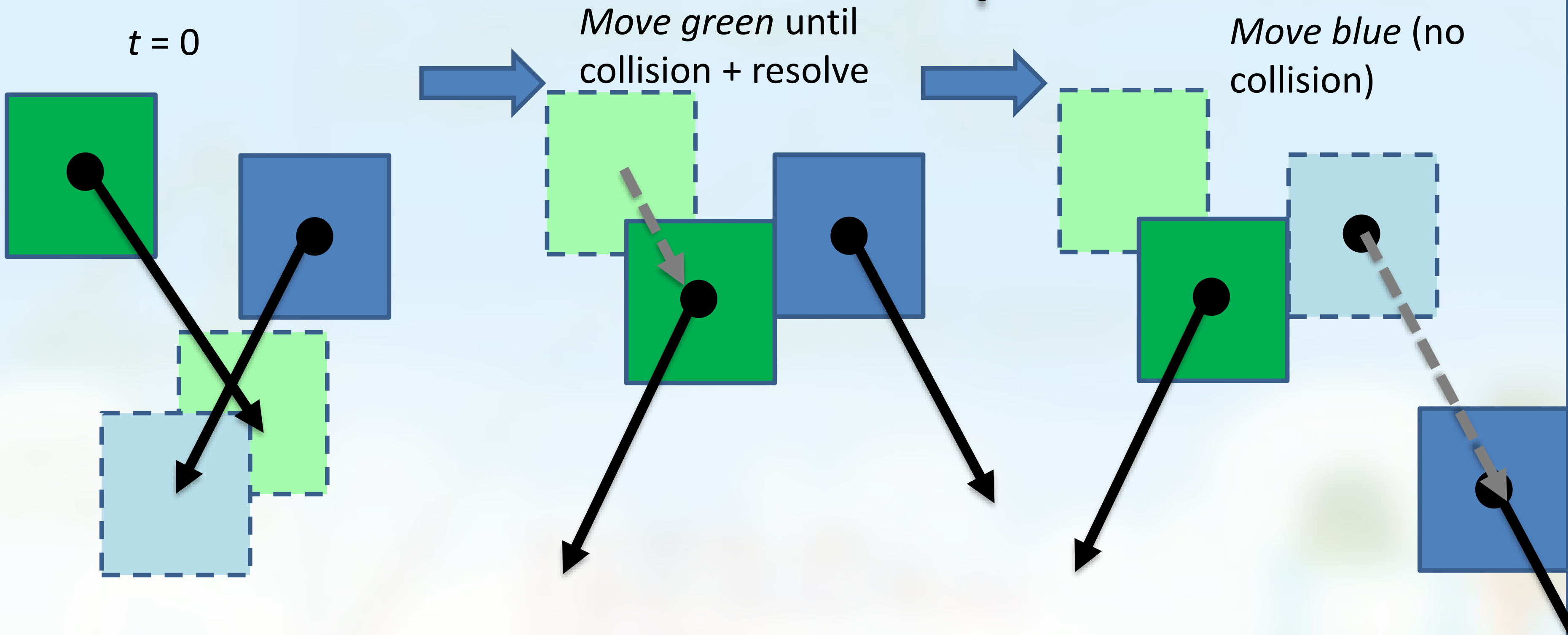
Update velocities using gravity and other forces
Move all bodies over time TargetTime
Try to correct overlapping positions
Resolve collisions

Advantage: cheap and fast

Disadvantages:

- This will get messy: overlaps will occur, cleaning up the mess is hard, and when moving fast, collisions are resolved in the wrong direction…

Turn-based Time Step Method



Turn-based Time Step Method

Strategy:

Update velocities using gravity and other forces
Move bodies *one by one* over time TargetTime, or until first collision
Resolve collisions as soon as they are detected.

(demo)

Advantages:

- efficient
- no overlaps
- collisions resolved in right direction.
- By decreasing TargetTime, we can *approximate* the perfect solution.

Disadvantages:

- Positions are slightly off: order matters
- You'll find "collisions" that are not really collisions…

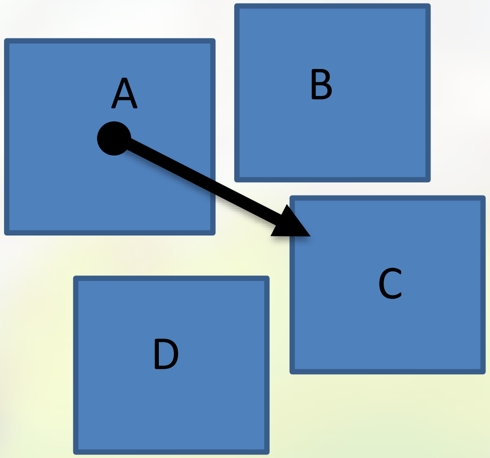
- The turn-based time step method is implemented in the given code sample.

  (Press S to try it out!)

- Disclaimer:
  - professional physics engines such as Box2D use more complex time step methods! (key: constraint solving)
  - This method is chosen because of the *balance between power and simplicity*
- Next: tips for implementing collision detection and resolve in this setting.

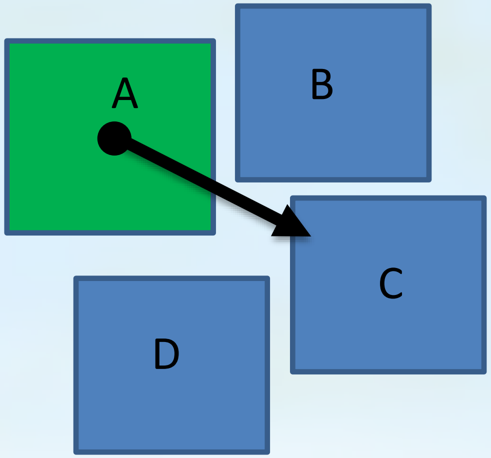
Tips – Collision Detection

- Tips for *collision detection:*
  - Assume that the other blocks do not move during one Step: update the position (by adding velocity) *only* for the current block, then apply a discrete collision check!
  - There may be multiple possible collisions during a frame: *find the first one, and only resolve that one!*
  - Point of Impact calculation *must* be exact.



- Only block B gets hit this frame!
- Resetting position of A should not cause overlap with D!

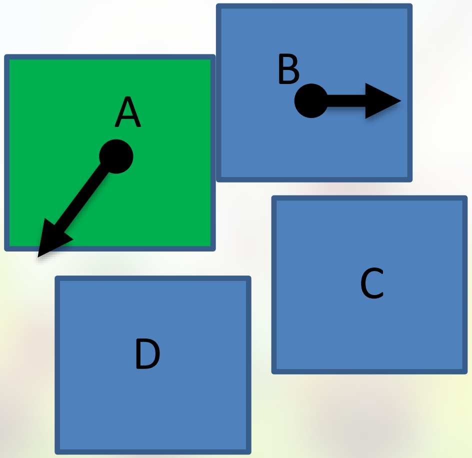
Tips – Collision Detection

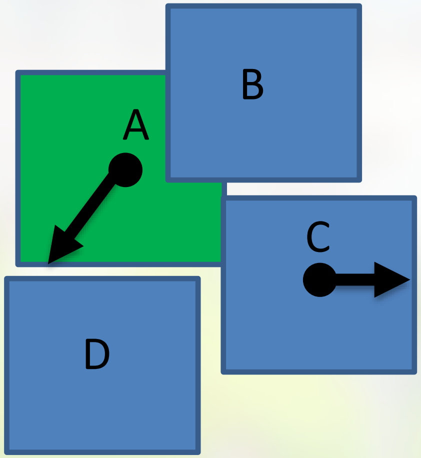


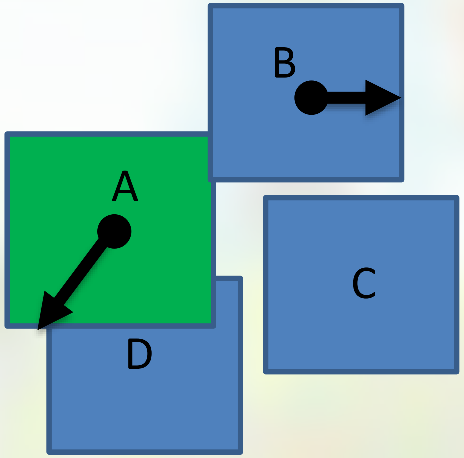
GOOD

BAD (resolving wrong collision)

BAD (no exact point of impact)

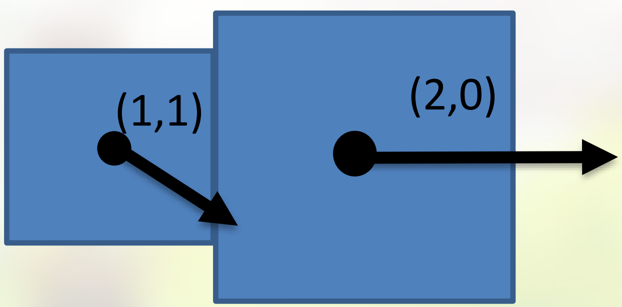




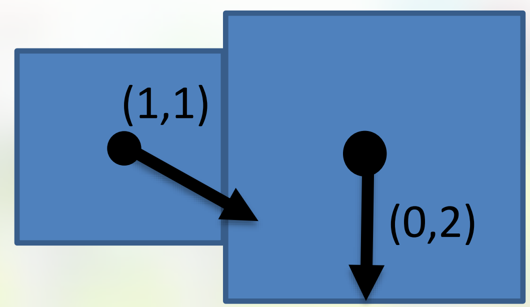


Tips – Collision Resolve

- Tips for *collision resolve:*
  - consider the velocities of both blocks, and use the formula from before, applied to only one coordinate of the velocities (*x* or *y*).
  - Because of the turn-based setup, you must consider the *relative velocity*, and only resolve the collision (change velocities) when (the relevant coordinate of) this is positive!



Rel. vel. = (-1,1) ➔ No real collision!



Rel. vel. = (1,-1) ➔ Actual collision

Summary

Collisions:

- detection: discrete vs continuous
- resolve:
  - position: POI vs move back
  - velocity: reflection (bounciness)

Physics:

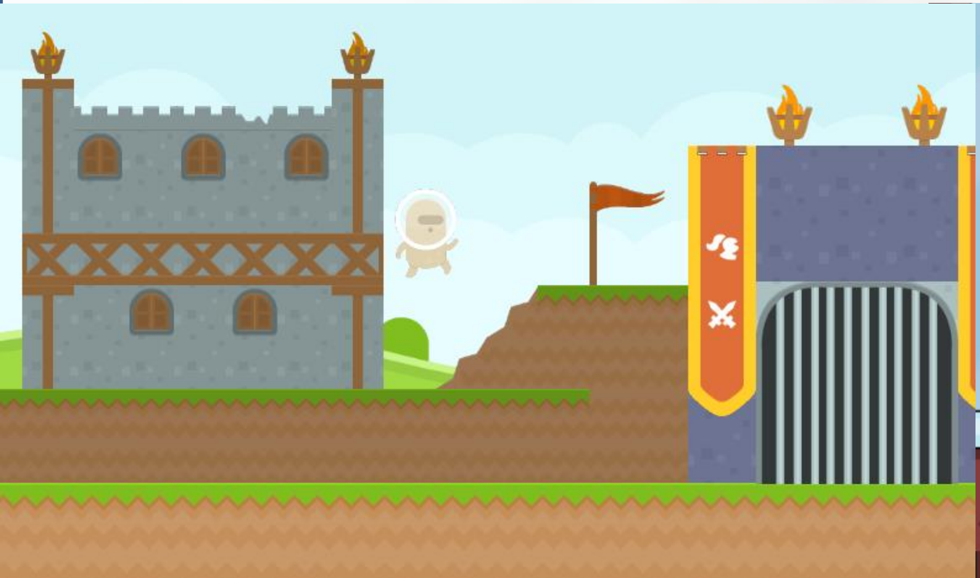
- Newton's laws
- Forces
- Momentum (conservation law) (→ for multiple moving objects)

Physics engine setup:

- Euler integration (with acceleration)
- Time step methods
- Turn based movement + tips

Application: Platformer

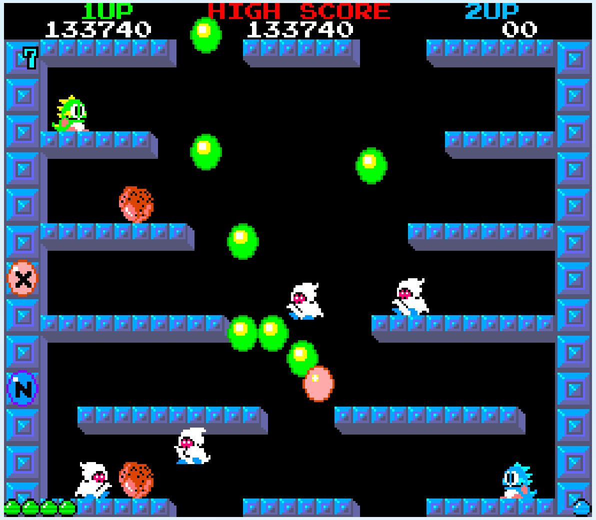
- Using your own rectangle-based physics engine, you can create e.g. platformer games.
  - Example physics functionality: moving platforms, friction, pushing objects…



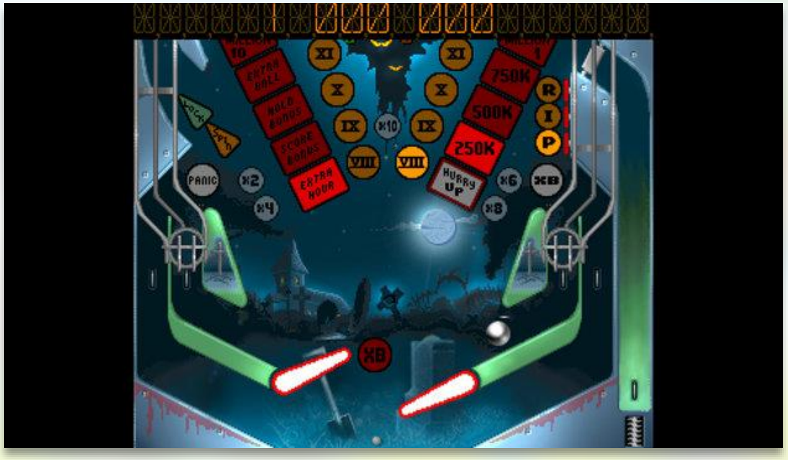
GXPEngine and Collisions

- When working with *rectangles* (sprites) in the (new) GXPEngine, you can use the *MoveUntilCollision* method instead of computing Point of Impact yourself. ☺
- This lecture's information about Newton's Laws (momentum) and Time Step methods is relevant regardless.
- When working with circles (next week!), we will need the manual Point of Impact calculation again.

Next lecture



- Today everything was horizontal or vertical
- What about angled lines?
- This requires:
  - Normals
  - Dot product




That's all Folks!
Any Question?

Detailed info

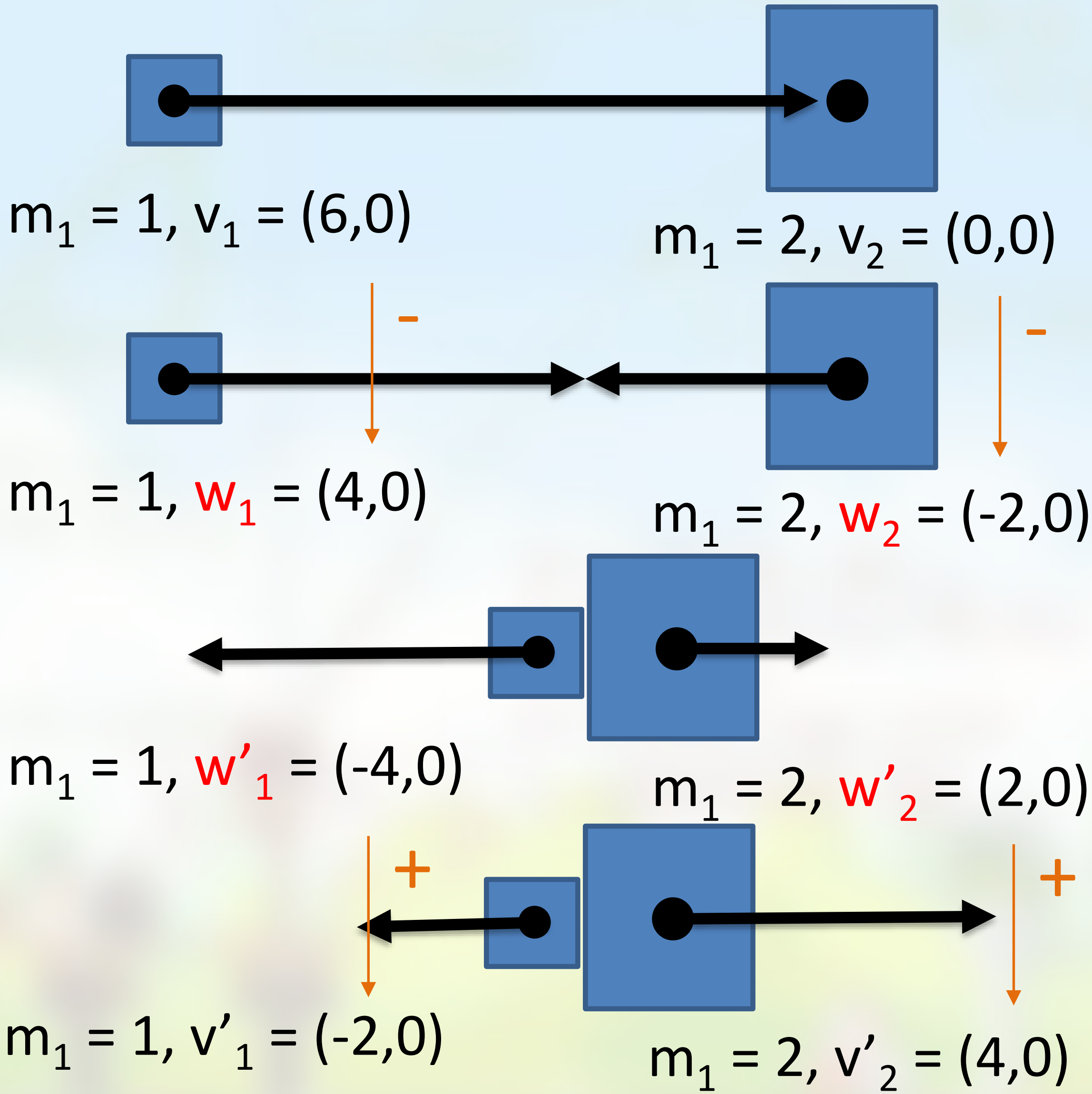
You should not read on, if you are already overwhelmed…

Line Collisions Formula – Proof Sketch

To prove: $v'_i = u - C \cdot (v_i - u)$

- Denote $w_i = v_i - u$, the velocities relative to CoM velocity (i=1,2).
- Then relative to w_1 and w_2, the new CoM velocity is (0,0).
- So after collision, $w'_i = -C\, w_i$ (we may use the simple formula)
- Substituting this in $v'_i = w'_i + u$ gives the above formula.

Example



Total momentum:	CoM velocity:
(6,0)	(2,0)
(0,0)	(0,0)
(0,0)	(0,0)
(6,0)	(2,0)