Physics Programming

Lecture 1 Vectors

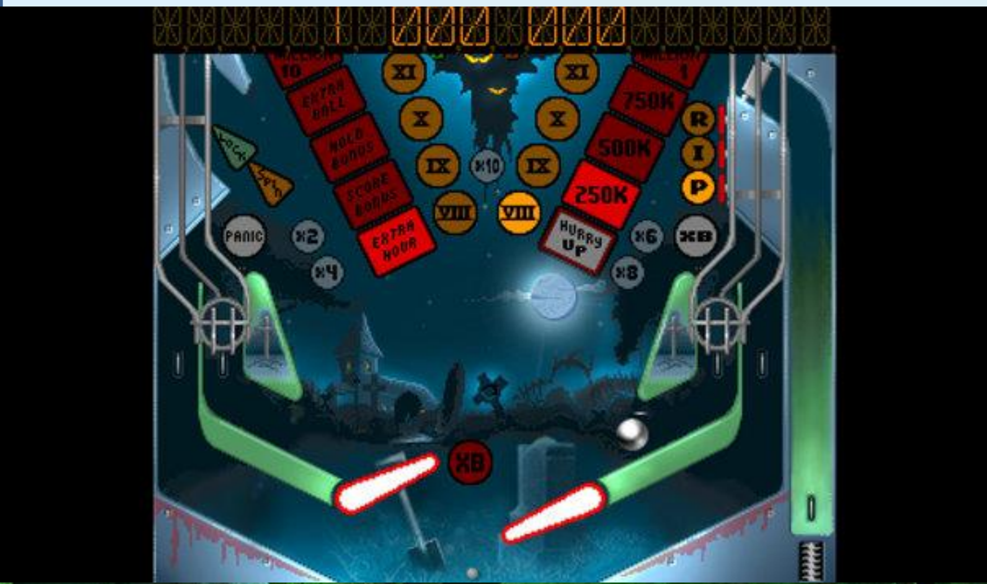
Slides made by Hans Wichman & Paul Bonsma

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

What is physics programming about?

- In an interactive application: *making objects move realistically* → inspired by laws of physics
- Note: *Good interaction / Fun game mechanics* are more important than *super realistic physics simulation!*
- But: good physics simulation can help for:
  - Intuitive controls
  - Dynamic game play
  - Realism / immersion

Physics Game Examples



What kind of physics are typically used in games?

- Mainly: *Rigid body* physics.
- *Collisions* / objects bouncing into each other (=impulses)
- *Continuous forces*:
  - Gravity
  - Explosion force
  - Attracting / repelling (e.g. magnets)
  - Springs, hinges
  - Friction / viscosity

- During this course, we won’t be able to create a full featured physics engine – not even for 2D.
- For advanced physics: use a physics engine (Bullet, Farseer, Unity physics, …)

- So why do we learn about game physics?

- So why do we learn about game physics?
  - To create custom physics / dynamic movement for simple games (e.g. platformers, pinball, race game)
  - To get a better understanding of what physics engines can and cannot do
  - To learn about *vector math.* Also useful for:
    - Computer graphics: 3D rendering
    - Procedural generation
    - …
    - Basically: *the basis for many upcoming courses*

More reasons to pay attention

- You will create reusable code and learn more about Object Oriented Design
(vectors, collisions, etc)
- Will be a source of reference for upcoming project
(prepare you for next project)
- To allow you to study further on your own
(more complicated subjects)
- In conclusion: effects ripple way beyond this course!

How difficult is Physics Programming?

- We start at the beginning, assuming only basic math and programming knowledge.
- We assume you passed 1.1 Programming, and can work with the GXPEngine + Visual Studio.
- We won't assume you have already seen vectors / trigonometry.

- Nevertheless:
  - If all of this is new to you and math is not your strength, it's a steep learning curve!
  - Staying on top of it every week is essential!

How difficult is Physics Programming?

It will **seem** hard, but practice makes perfect... So:

- *Go to the labs, ask feedback*
- Use your time well
- *Don't start too late with the assignments*
- If needed: study additional resources
- *Expected time investment: 80 hours!*

  (>13 hours per week!)

Grading

- Week 3.9: *Assessment:* based on a final programming assignment (grading criteria: see manual)
  → Main grade (between 1 – 10, where 5.5 = sufficient)

- There will be *five lab assignments*, that directly prepare for the final assignment.

- Discussing your solutions to the assignments with your lab teacher is perfect preparation for the assessment!

- To access the assignments, you need to complete a *short test of understanding* on Blackboard every week.

- Details on Blackboard

Outline Today

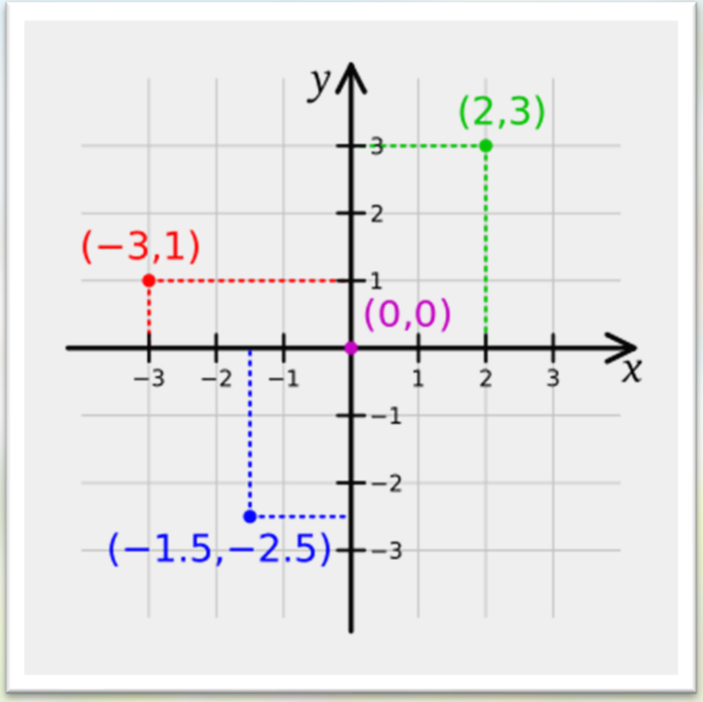
- Coordinate systems
- Vectors
- Vector addition and subtraction
- Vectors in code:
  - Struct vs class
  - Operator overloading
- Length (Pythagoras), scaling, normalizing
- Assignment 1 explanation

Coordinate systems

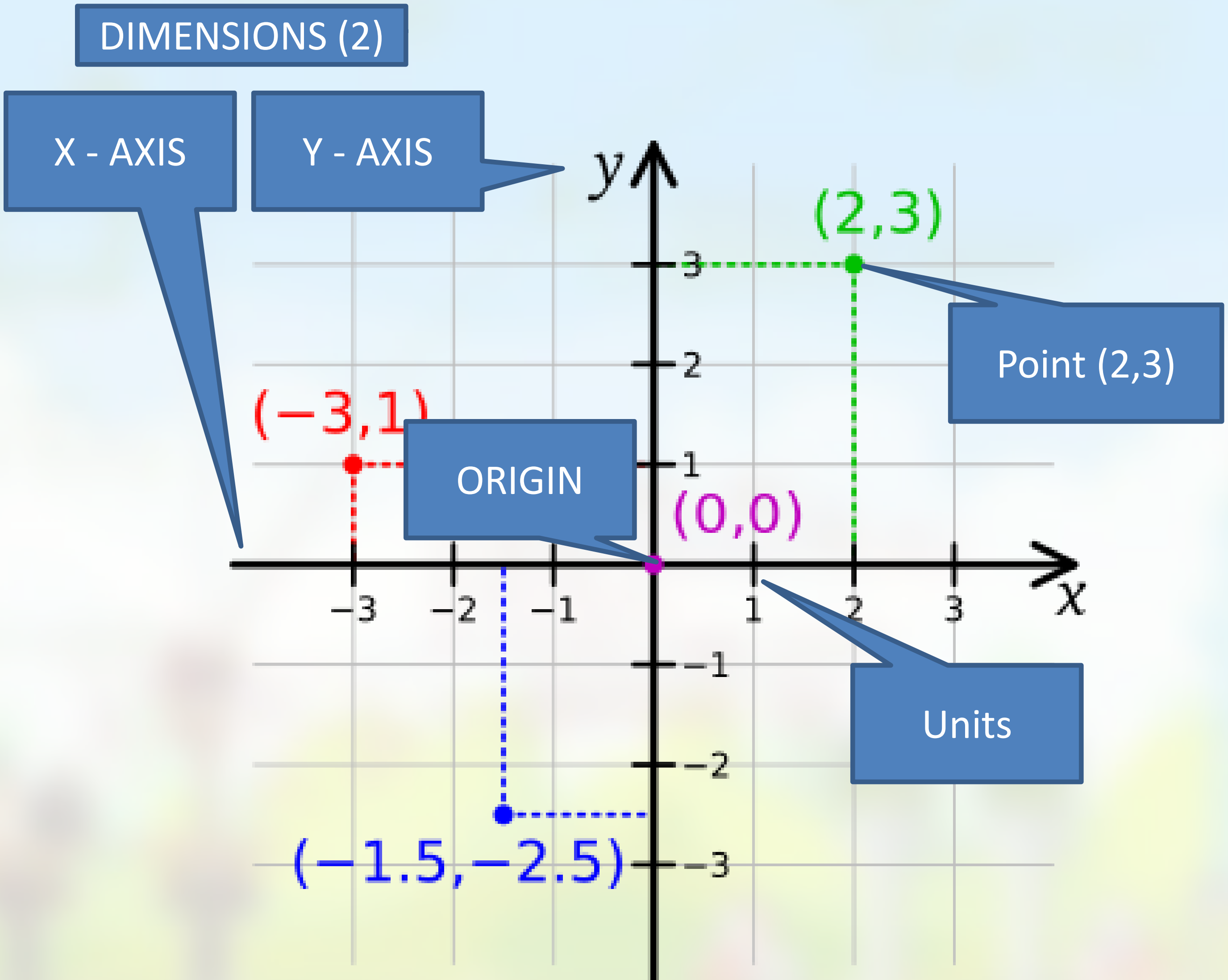
On axis, units, and your freedom to choose…

What is a Coordinate System?

- A framework to describe *positions in space* using *numbers* (a.k.a. *coordinates*).
- Most well known framework is the *Cartesian Coordinate System*:
  - *Perpendicular axes (=90°)*
  - Same *unit* on all
- There are other systems (e.g. *polar*) → not used for now.



Elements of Cartesian Coordinate System: Dimensions, Axis, Origin, Units & Points

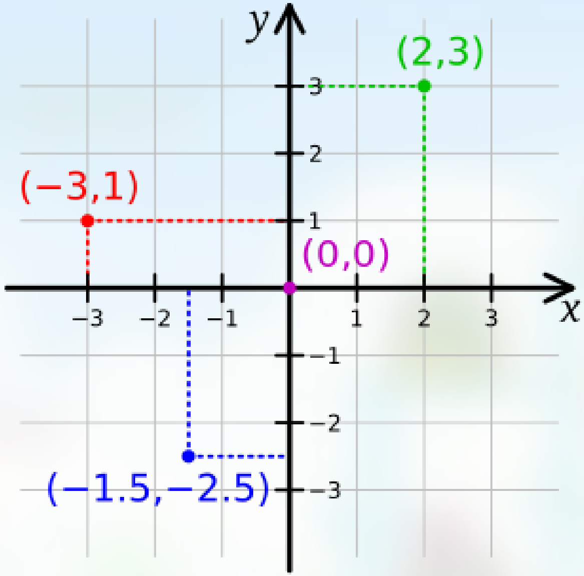


Freedom of choice

- Some things are fixed (because Cartesian):
  - Same unit on every axis
  - Axes are perpendicular.
- …we still have to choose:
  - The number of dimensions (1D/2D/3D/…)
  - The directions of the axis
  - The unit size

Example 1: High school math

- One customary approach in high school math is:
  - 2 dimensions
  - 1 unit = 1 centimeter
  - +x to the right
  - +y is up
- Lot of general math subjects will be discussed in this space (for example trigonometry)

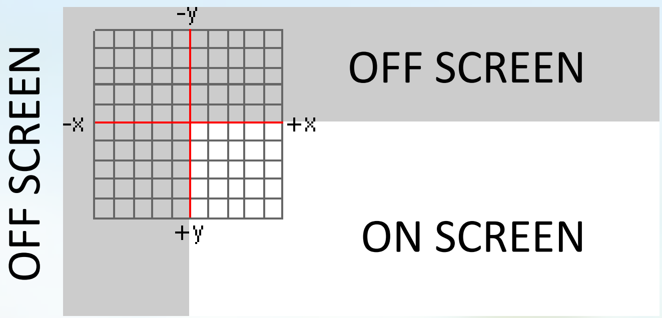


Example 2: Cocos2D

- Some game engines, such as Cocos2D use **almost** the same approach:
  - 2 dimensions
  - 1 unit = **1 pixel**
  - +x to the right
  - +y is up
  - (0,0) is bottom left
  - (screen width, screen height) is top right

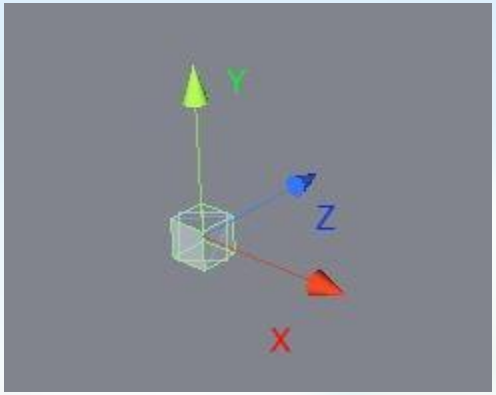
Example 3: GXPEngine

- The GXP Engine uses:
  - 2 dimensions
  - 1 unit = 1 pixel
  - +x to the right
  - +y is **DOWN**
  - **(0,0) is top left of screen**
  - The “floor” is at game.height



Example 4: Unity

- Unity3D uses:
  - 3 dimensions
  - 1 unit = 1 meter
  - +x to the right
  - +y is up
  - +z into the screen



Vectors

About general terminology often used in physics

Vectors

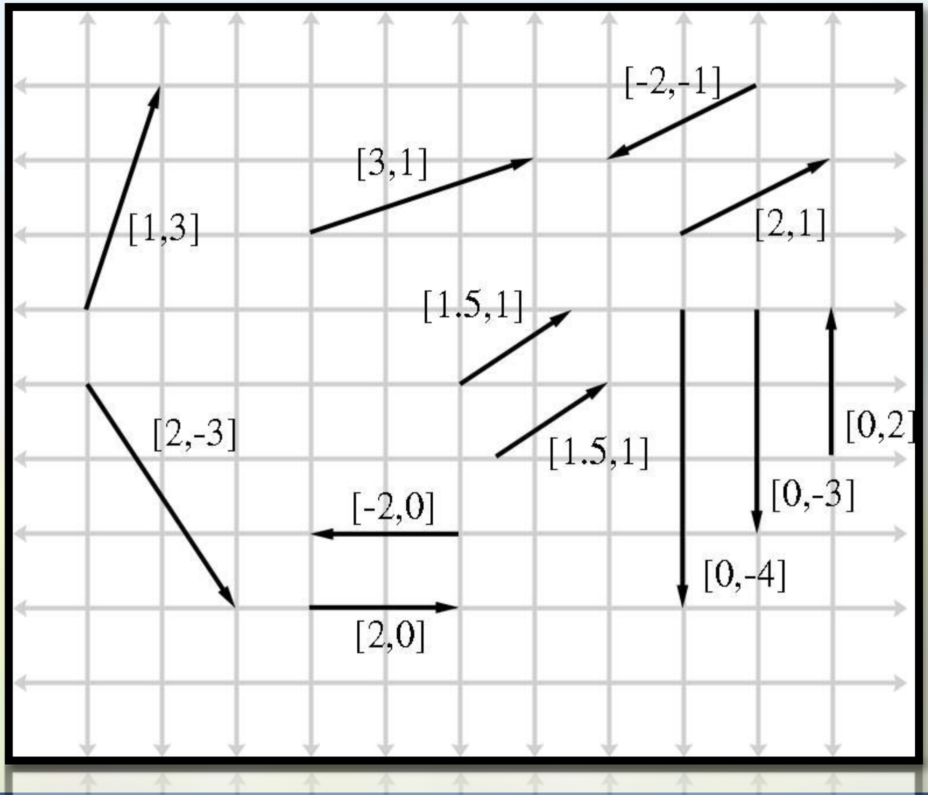
- In 2D games we use x & y properties to position and move objects around, e.g. an object is at position (x,y)
- In physics / 2d math we call this (x,y) pair …
  - a point
  - a position
  - a vector
- All are common, but for game math the name **Vector** is **most common.**

Vectors

- A 2d vector describes an (x,y) pair
- A 3d vector describes an (x,y,z) tuple
- The "textbook" notation for a vector v is $\vec{v}$
- When we refer to a Vector's x and y components *(coordinates)*, it's written like:
  - $\vec{v}.x$ or $\vec{v}_x$
  - $\vec{v}.y$ or $\vec{v}_y$
- Both notations are common!

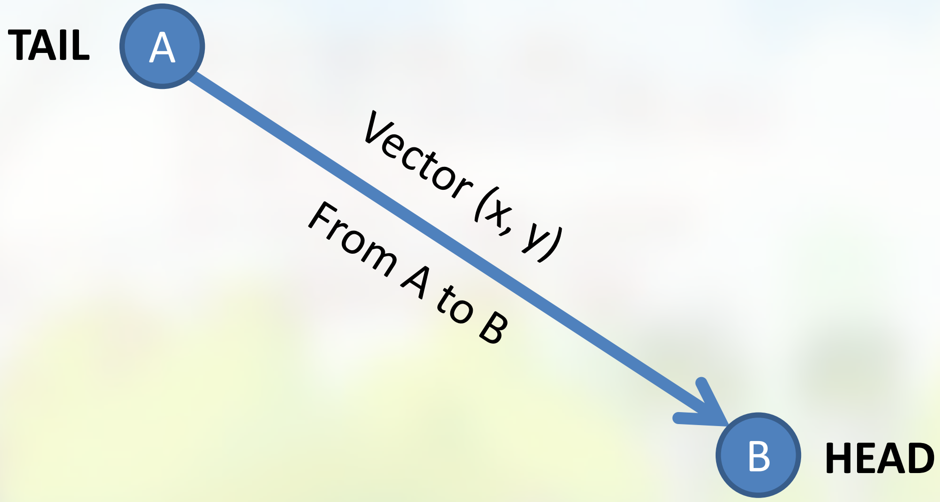
Drawing vectors / equality

- Vectors are often drawn as *arrows.*
- A vector can be *drawn anywhere.*
- Two vectors are *equal* if and only if all their coordinates are equal.



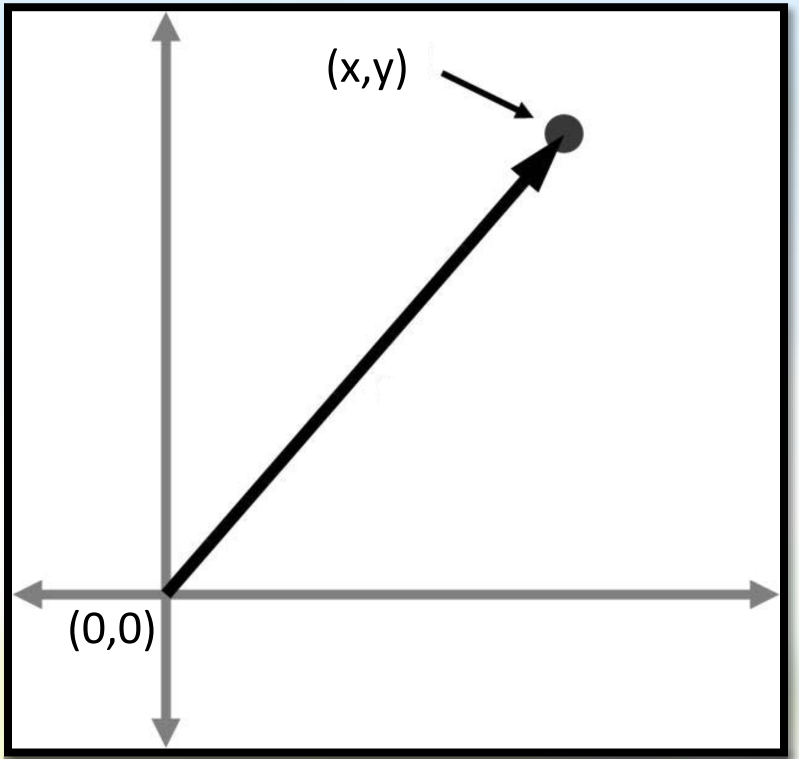
Vector terminology: tail & head / from .. to ..

- A *drawn vector* has a **tail** and a **head**:
- Below, vector (x,y) is drawn *from A to B:*



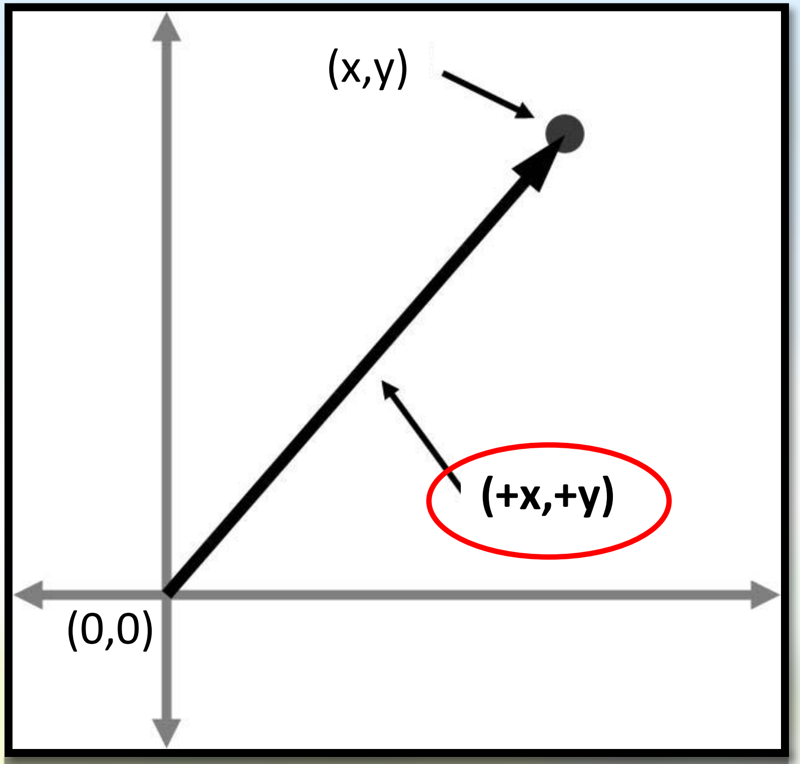
Vectors can describe a *position*

We are **at** (x,y) → absolute value



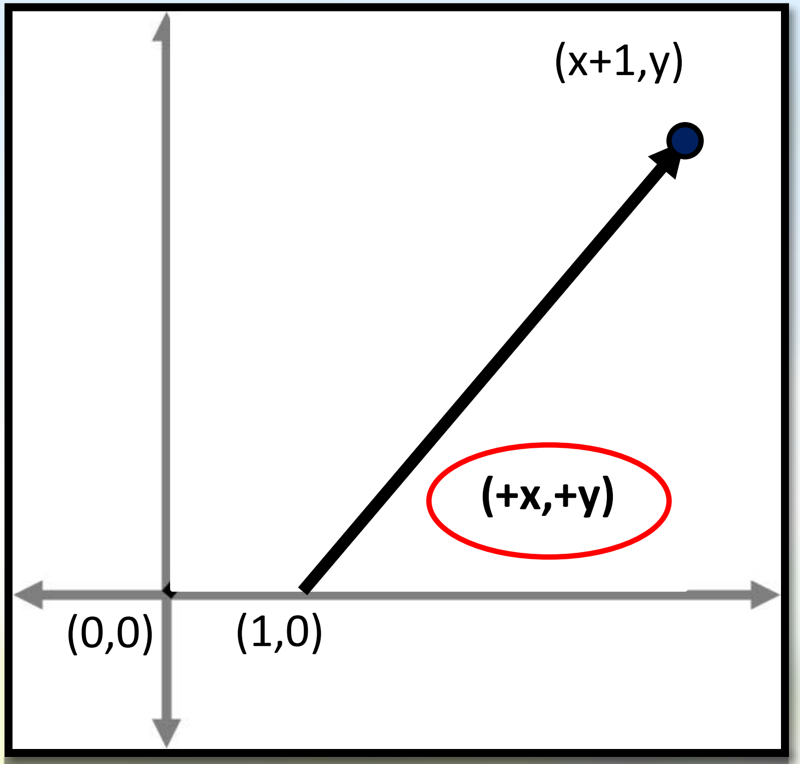
Vectors can describe a *displacement* / directional movement

Here **from** (0,0) **to** (x,y) → relative value



Vectors can describe a *displacement* / directional movement

Here **from** (1,0) **to** (x+1,y) → relative value



Modifying positions directly?

- We can make objects move by storing their position in a vector, and modifying this.
  - Quick demo: 002_ball_vector_position
- However, in real life, positions only change because we apply force, which causes acceleration, which causes velocity...

  (These are actually *Newton's* first two *laws* → later)
- Let's start with velocity...
  - what is velocity?
  - how does velocity change an object's position?

Velocity

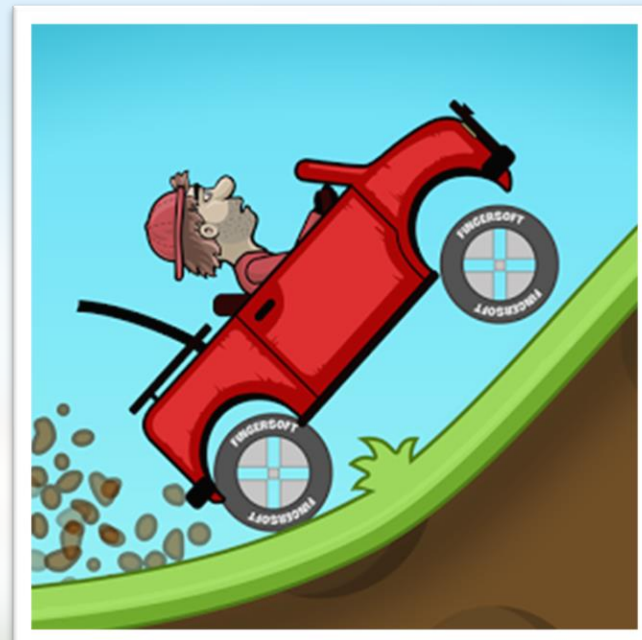
- **Velocity is speed in a specific direction**

- Velocity, speed or direction?:
  - 80 miles an hour?
  - Going north?
  - Heading left?
  - Going west at 20 mph ?

Velocity

- **Velocity is speed in a specific direction**

- Velocity, speed or direction?:
  - 80 miles an hour? *speed*
  - Going north? *direction*
  - Heading left? *direction*
  - Going west at 20 mph? *velocity*



Using Vec2 for velocity

- Velocity **is** directional movement
- Vectors **can describe** directional movement ("displacement")
- → we can store velocity as a vector too!

- For example:
  - If I say my velocity is (3,4) that means that I want to go 3 pixels to the right, and 4 down each frame.

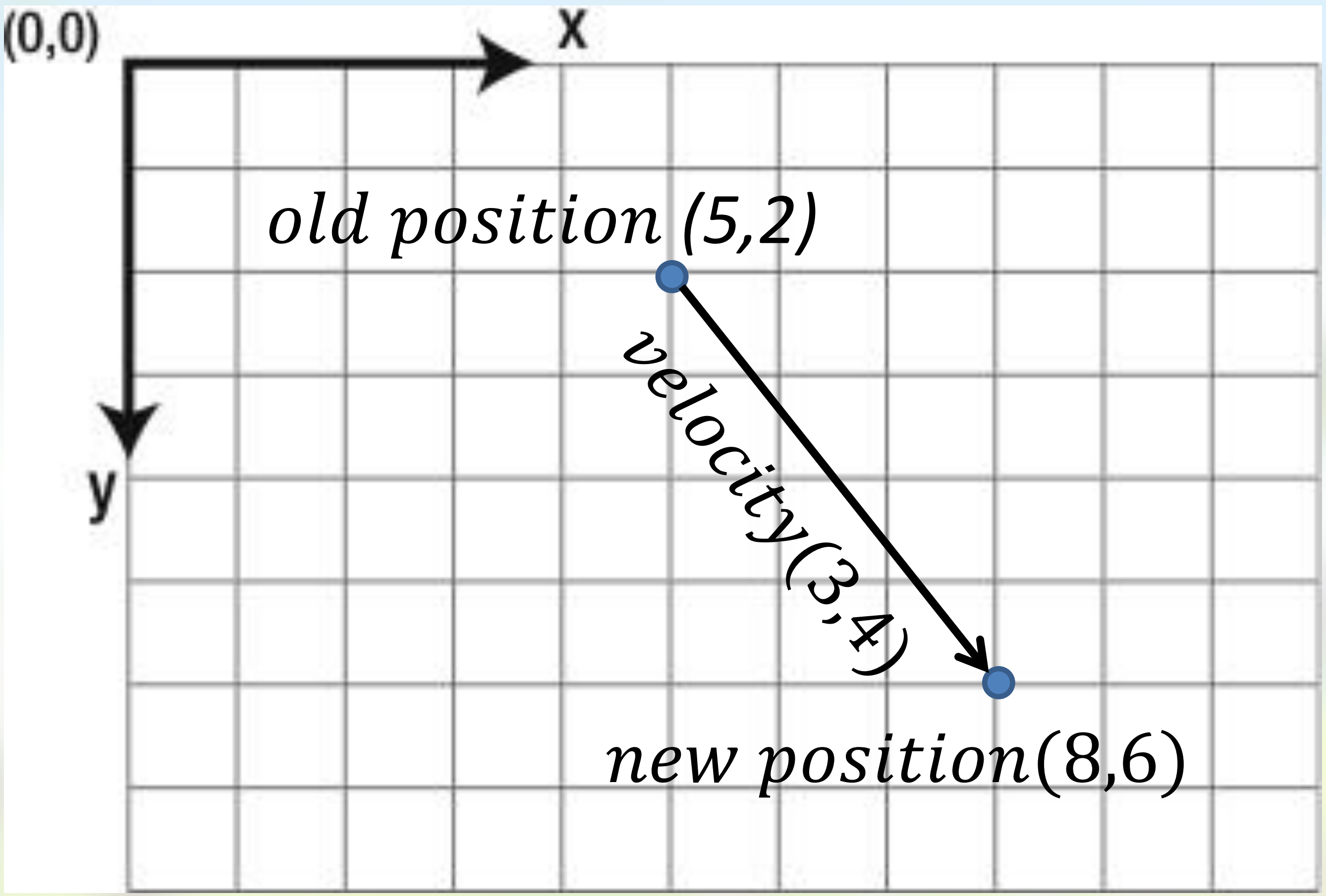
- This requires vector **addition.**

Vector addition

On how to get movin’

Vector addition in a picture

new position = old position + velocity

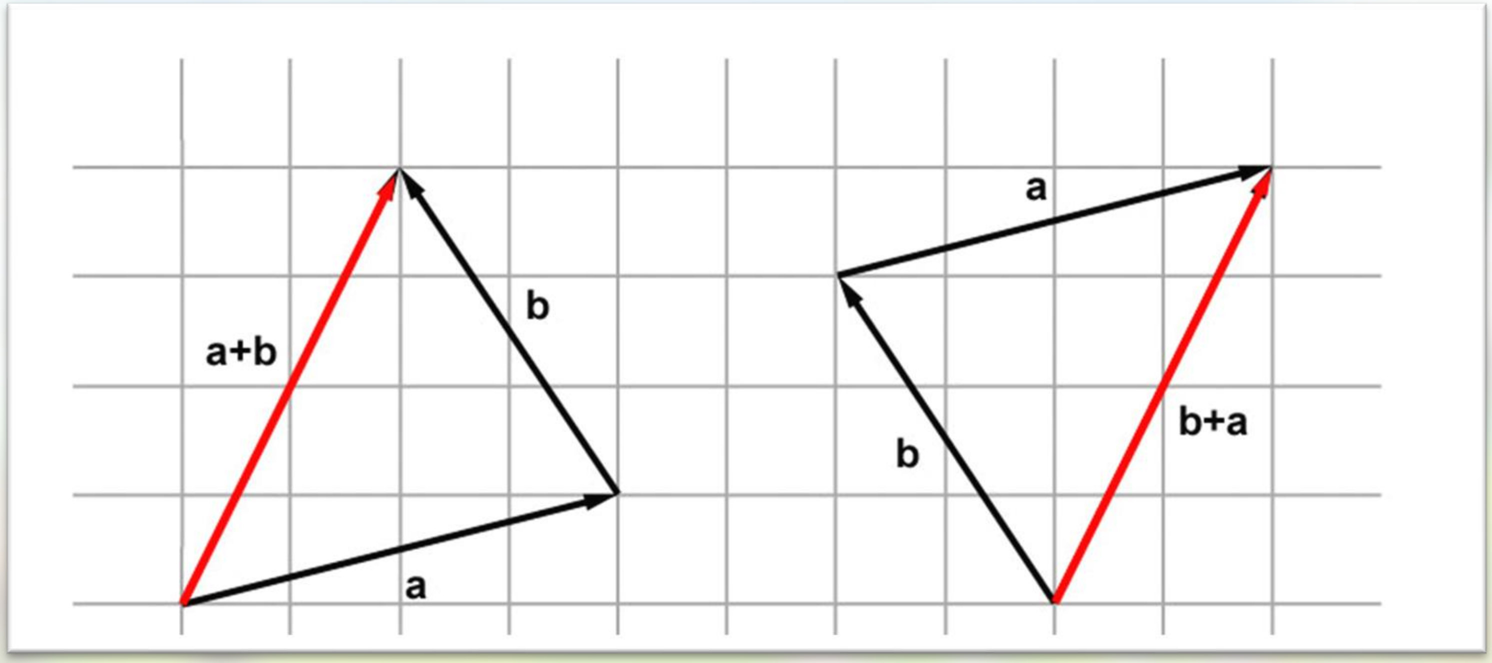


Vector addition: (mathematical) definition

- new position = old position + velocity
- Officially written as $\vec{\boldsymbol{p}}' = \vec{\boldsymbol{p}} + \vec{\boldsymbol{v}}$
- The $'$ in $\vec{\boldsymbol{p}}'$ means "**modified version of**"
- Is done through **component-wise** addition:
  - Given $\vec{p}$ = ($\vec{p}$.x, $\vec{p}$.y) and $\vec{v}$ = ($\vec{v}$.x, $\vec{v}$.y)
  - Then $\vec{p}'$ = $\vec{p}$ + $\vec{v}$ = ($\vec{p}$.x + $\vec{v}$.x, $\vec{p}$.y + $\vec{v}$.y)
  - So $\vec{p}'$ = $\vec{p}$ + $\vec{v}$ = (5,2) + (3,4) = $(5 + 3, 2 + 4)$ = (8,6)

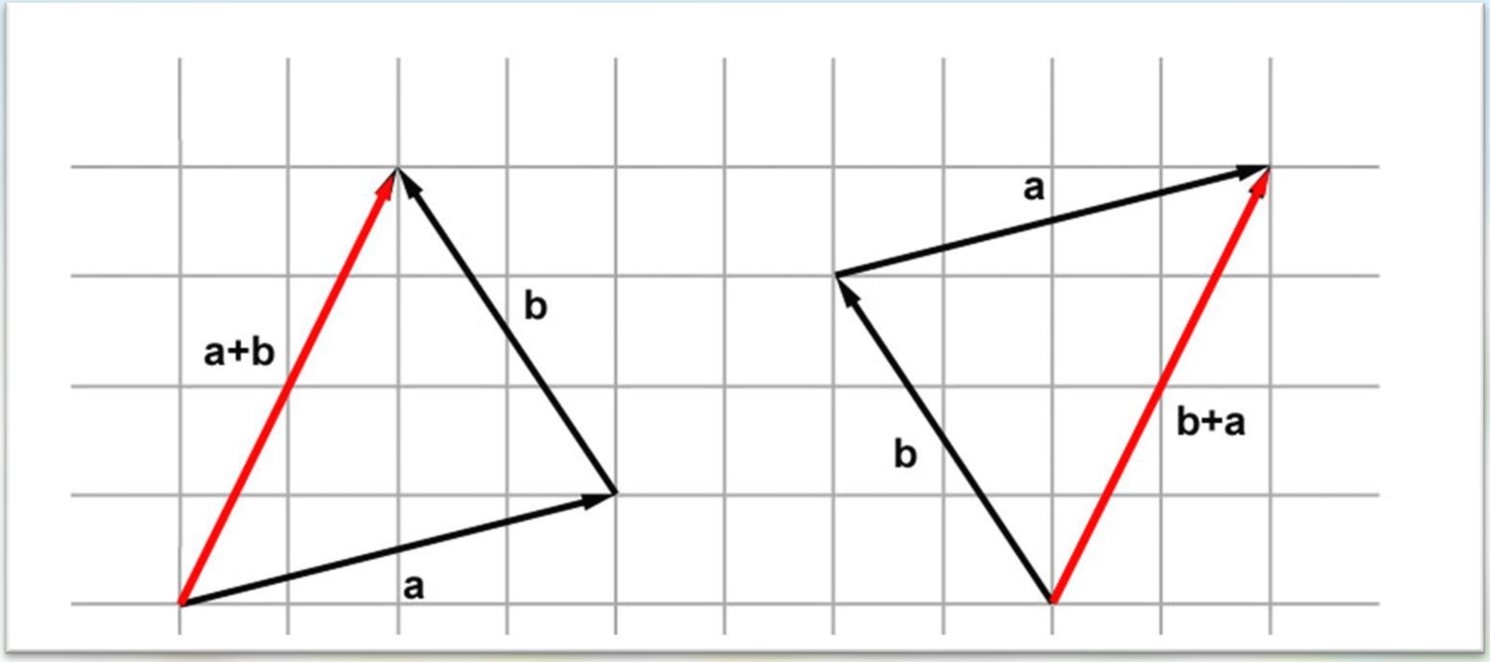
Visualizing vector addition

- The vector $\vec{a} + \vec{b}$ can be visualized by drawing $\overrightarrow{a}$ and $\vec{b}$ "head to tail" as follows:
- Order doesn't matter! → *"Commutative"*



Vector addition is *commutative*

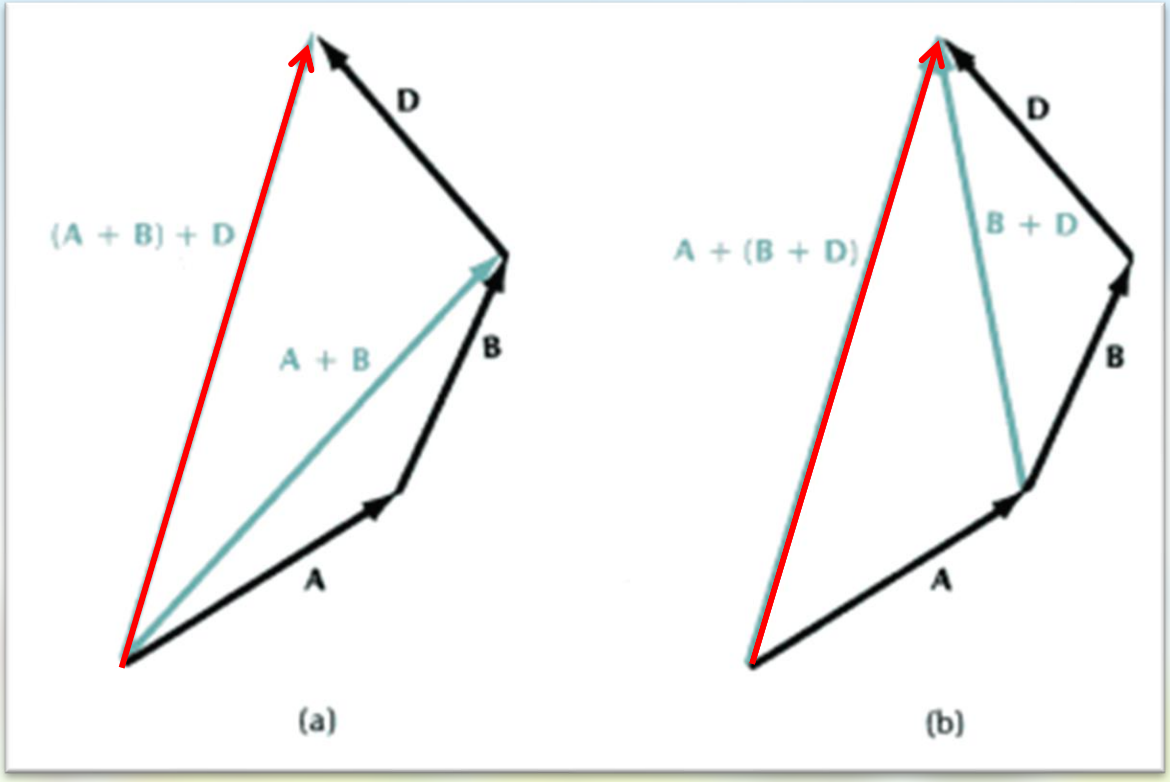
$$\vec{a} + \vec{b} = \vec{b} + \vec{a}$$



Note the end result (red) after adding

Vector addition is *associative*

$$(\vec{a} + \vec{b}) + \vec{d} = \vec{a} + (\vec{b} + \vec{d})$$



Note the end result (red) after adding

Vector addition

- **Summing up:**
  - the **order** in which we add vectors **doesn't matter**.

Treasure hunting

- You just found a map! Find the treasure with the least amount of steps possible, which way do you go?

Follow these steps to find the treasure!:

(-1,0)
(0,-2)
(1,-3)
(1, 3)
(0,2)

How many steps do you have to take?

Move to Target

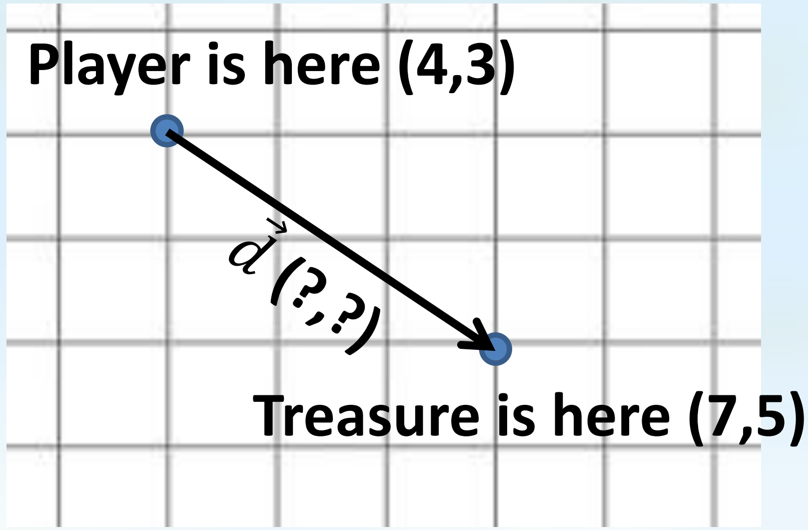
- What if instead, we know where we are (position $\vec{p}$), where the treasure is (target $\vec{t}$), and we want to know how to get there?
- $\vec{p} + ? = \vec{t}$
- So: $? = \vec{t} - \vec{p}$
- ➔ We need vector *subtraction*
- Definition:

$$t - \vec{p} = (\vec{t}.x - \vec{p}.x, \vec{t}.y - \vec{p}.y)$$

Subtracting vectors

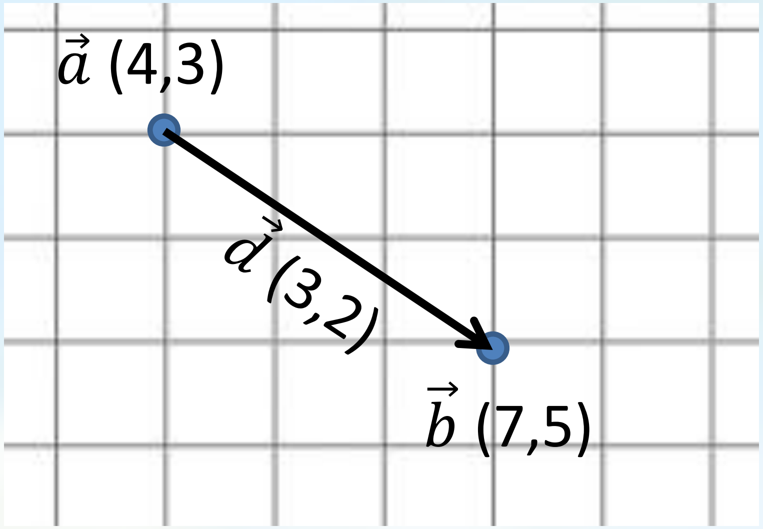
Where you learn how to go where you want

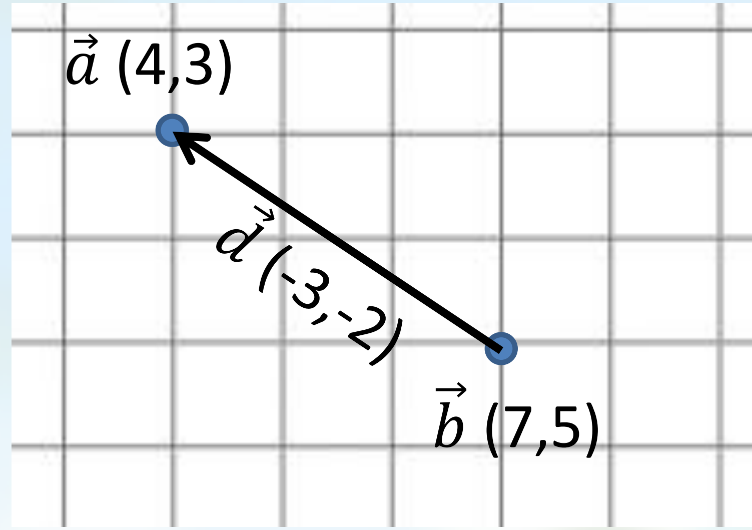
Imagine this scenario:



- Player is at (4,3), treasure is at (7,5)
- Player needs to move over vector $\vec{d}$
- $\vec{d}$ stands for **d**elta, delta means change/difference

Calculate delta using vector subtraction





- More general:
  - $\vec{b}$ - $\vec{a}$ describes the displacement from $\vec{a}$ to $\vec{b}$
  - $\vec{a}$ - $\vec{b}$ describes the displacement from $\vec{b}$ to $\vec{a}$
- Subtraction is not commutative: $\vec{b} - \vec{a} \neq \vec{a} - \vec{b}$!

  (…nor associative)

Vectors in code

On what vectors know and do, and how we can integrate them into the GXPEngine

Defining vectors in code

- Object oriented design:
  - What do vectors know?
  - What can vectors do?
  - What can we do with them?

Defining vectors in code

- Object oriented design:
  - What do vectors know?
    - they know about their x & y coordinates (and maybe a z in 3d space)
    - They know their length?
    - They know their direction?
  - What can vectors "do"?
    - they can be added
    - they can be subtracted
    - they can be ….

Defining vectors in code

- Object oriented design:
  - What do vectors know?
    - they know about their x & y coordinates (and maybe a z in 3d space)
    - ~~They know their length?~~ →can be deduced from *x & y!*
    - ~~They know their direction?~~ →can be deduced from *x & y!*
  - What can vectors "do"?
    - they can be added
    - they can be subtracted
    - they can be ….

Defining vectors in code

- Object oriented design:
  - What do vectors know?
    - they know about an x & y (and maybe a z in 3d space)
  - What can vectors "do"?
    - they can deduce their length
    - they can deduce their direction
    - they can be added
    - they can be subtracted
    - they can be ….

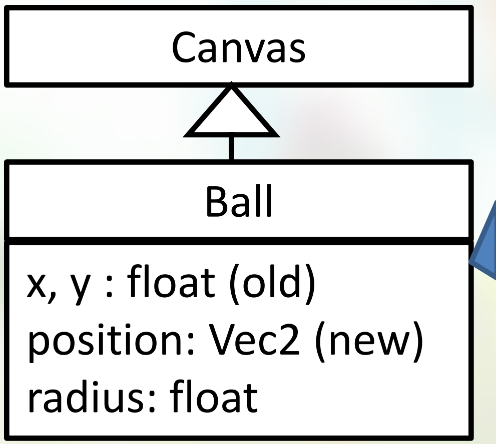
Vector2
x : float **y : float**
Length():float Angle():float Add (…):…. etc

Defining vectors in code

```
class Vec2 {
    public float x = 0;
    public float y = 0;
}
```

How do we use our Vec2 for positions?

- Give objects a **Vec2** instance called **position**
- Change **object.position.**x/y instead of object.x/y
- On object Step() (or Update())
  copy object.position.x/y to object.x/y
- Demo +002_ball_vector_position



Vec2: Class or struct?

In C#, *classes* and *structs* are very similar. The only difference:

- A *class* is a *reference type* (it refers to an object in memory that has to be explicitly created using *new).*
- A *struct* is a *value type* (its value is stored directly where the variable is "defined").

Consider this code – what happens? What is the value of `v.x`?

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Class Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables:

Vec2	v
(null)	

Vec2	w
(null)	

Memory (heap):

Class Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables: Memory (heap):

Class Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

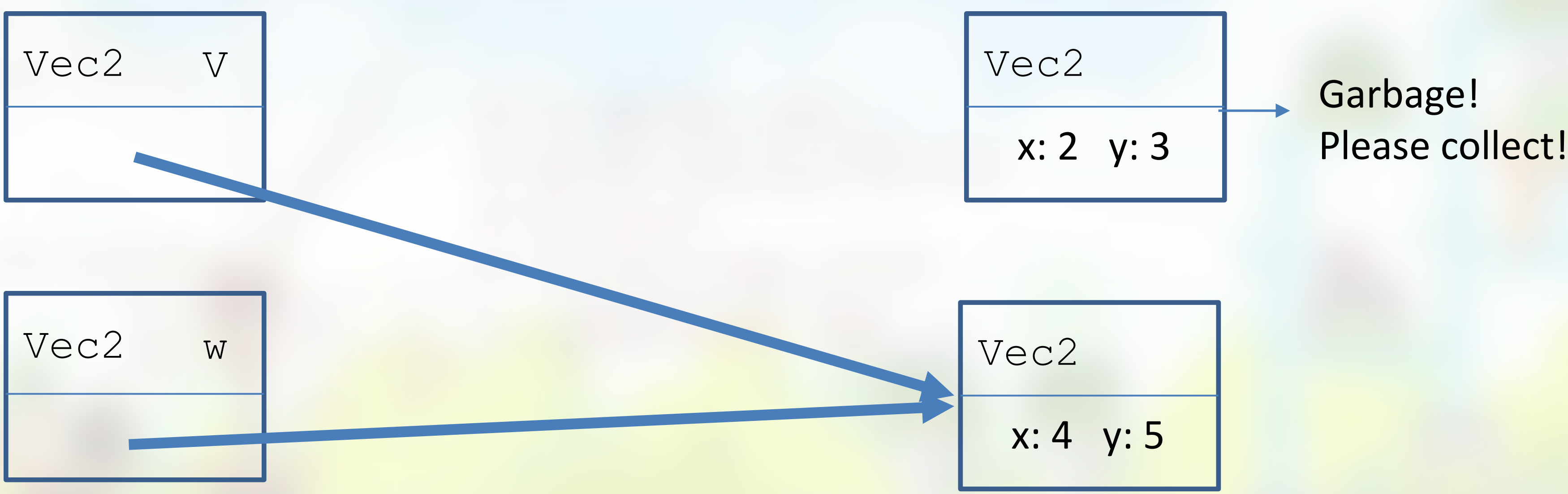
Variables: Memory (heap):

Class Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables:

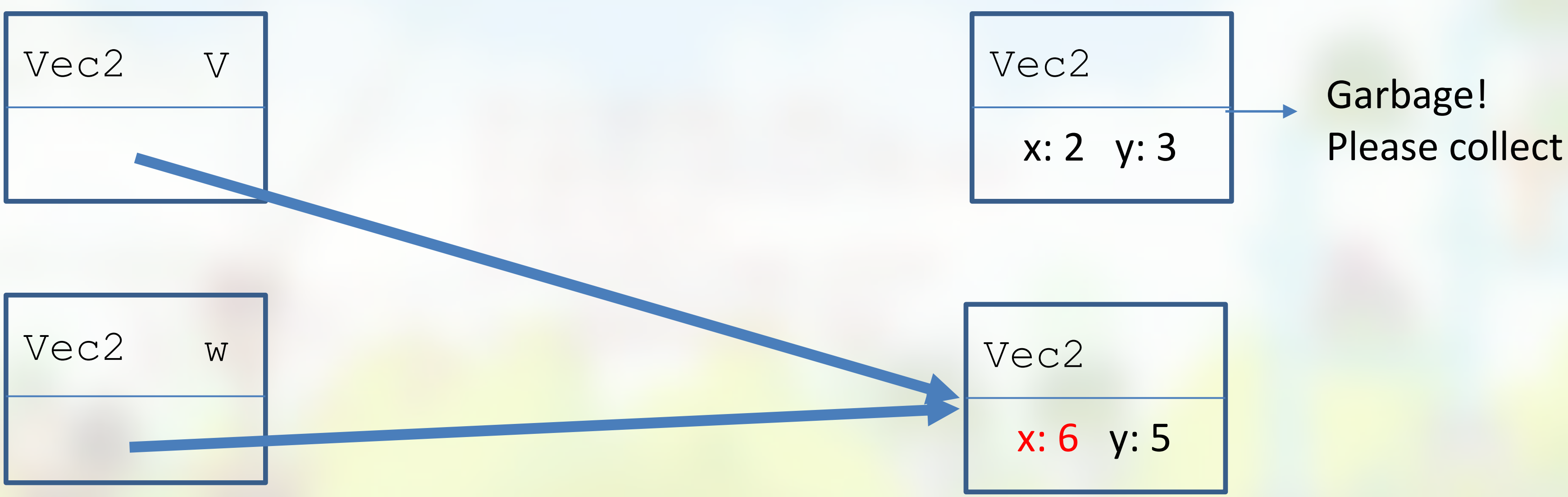
Memory (heap):



Class Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables: Memory (heap):



Struct Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables:

`Vec2    v`

x: 0   y: 0

`Vec2    w`

x: 0   y: 0

Struct Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables:

Vec2	V
x: 2	y: 3

Vec2	w
x: 0	y: 0

Struct Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables:

Vec2	V
x: 2	y: 3

Vec2	w
x: 4	y: 5

Struct Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables:

Vec2 v
x: 4 y: 5

Vec2 w
x: 4 y: 5

Struct Case

```
Vec2 v = new Vec2(2,3);
Vec2 w = new Vec2(4,5);
v = w;
w.x = 6;
```

Variables:

Vec2	v
x: 4	y: 5

Vec2	w
x: 6	y: 5

(Dis)advantages of using structs?

Advantages:

- No “shared reference” bugs – modifying a struct cannot have unexpected consequences

  (you’ll be *so* thankful for that!)

- Efficient: No *memory allocation, pointer dereferencing, garbage collection* needed for doing simple operations

  (…and by the end of this course, you’ll be doing a *lot* of operations with vectors!)

- Easy to understand

  (…unless you’re only used to classes, and don’t really know the difference!)

(Dis)advantages of using structs?

Disadvantages:

- For every assignment operation / method call with struct arguments, *all data in the struct is copied* – inefficient for large structs.

  (…but we only have two floats)
- Structs cannot be *null*.

  (…which is sometimes useful as a “special value”)
- Sometimes you *want* to copy something, change it, and have the original changed too.
- You cannot use *inheritance* for structs.

Conclusion

- You should use classes for most things, especially for complex objects, but
- …it's best to make our Vec2 a struct.

  (See also Unity, and other C# libraries with vectors)

- …but you need to know the differences between structs and classes to avoid bugs!

- Study the example 001_class_vs_struct closely!

Vector addition in code

Vector addition in code:

- Here's one way to implement addition for Vec2 structs:

```
struct Vec2 {
    public float x = 0;
    public float y = 0;

    public void Add (Vec2 other) {
        x += other.x;
        y += other.y;
    }
}
```

Vector addition in code:

```
public void Add (Vec2 other) {
      x += other.x;
      y += other.y;
}
```

Some OO design decisions are already made here:

- Method parameter is Vec2, not two floats

  (Conceptually better, and we should start using Vec2 for everything anyway.)

- Calling the method *modifies* the struct

  (…which is somewhat controversial: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/struct)

- We don't return anything

  (Returning *this* is useful, but confusing combined with modification)

Operator overloading

Q: Can't we just add a bunch a vectors together without modifying them? Preferably with very readable code, like this?:

```
Vec2 sum = v1 + v2 + v3 + v4;
```

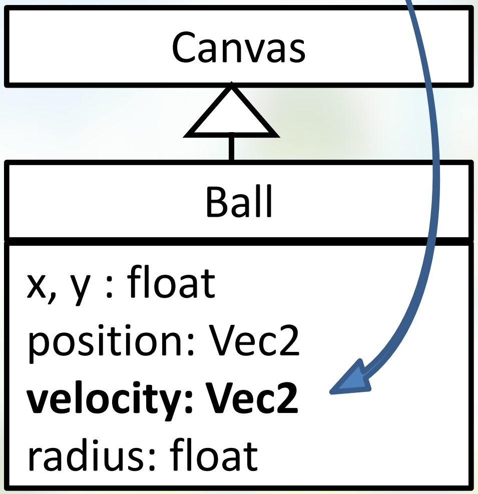
A: Yes, using *operator overloading*!

→ see 001_class_vs_struct

(and https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/operator)

Using Vec2 for adding velocity

- How do we use Vec2 to implement velocity?
  - Give objects a **Vec2** instance called **velocity**
  - Stop updating **object.position** x,y directly
  - Change only **object.velocity** x,y directly
  - On object Step:
    - **Add** object velocity to object position
    - **Copy** object position to object x,y
  - This is called **Euler Integration** **(Oiler):**
    - We store explicit position
    - We store explicit velocity
    - We add velocity to position each frame



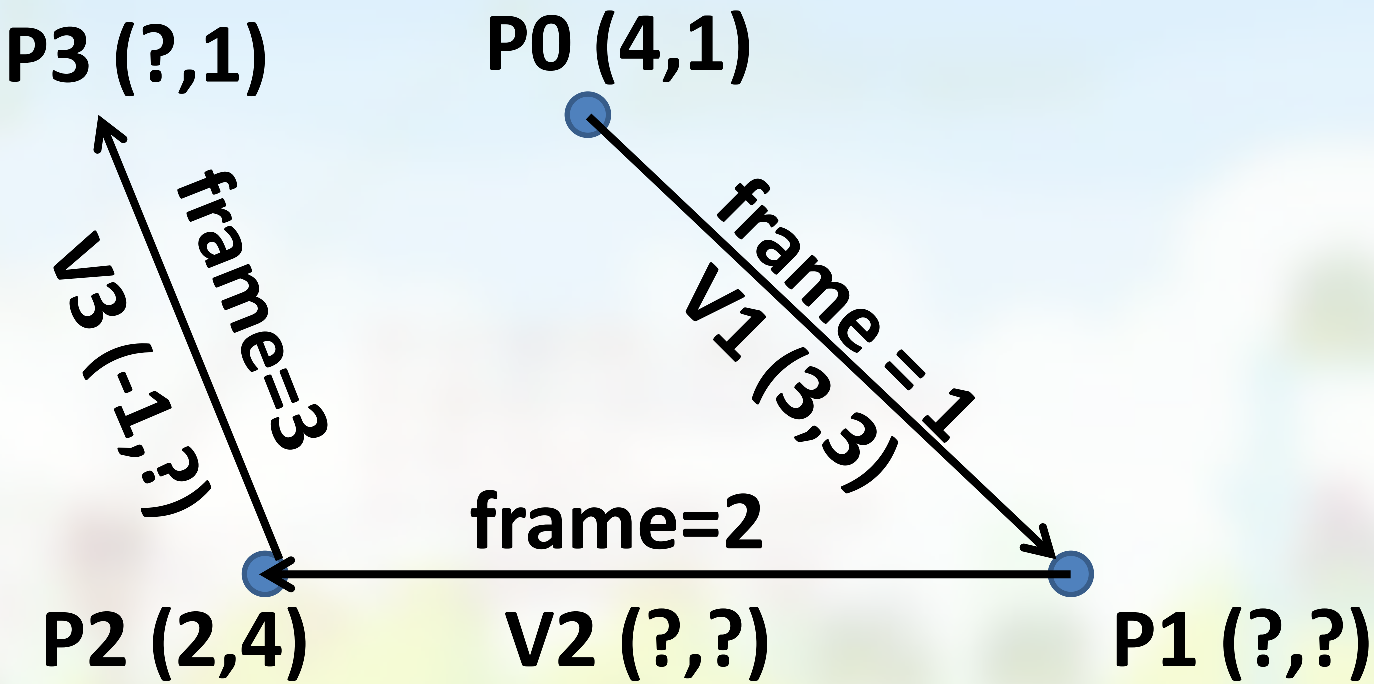
- Example +003_ball_vector_velocity
  - Note: we seem to move faster diagonally… Why?

Velocity, units?

- Velocity is movement over time
- This implies two different measurements:
  - one for distance
  - one for time
- For example:
  - kilometers/miles per hour
  - meters per second
- In our code, velocity is defined in pixels/frame

Velocity (and thus position) can change each frame!

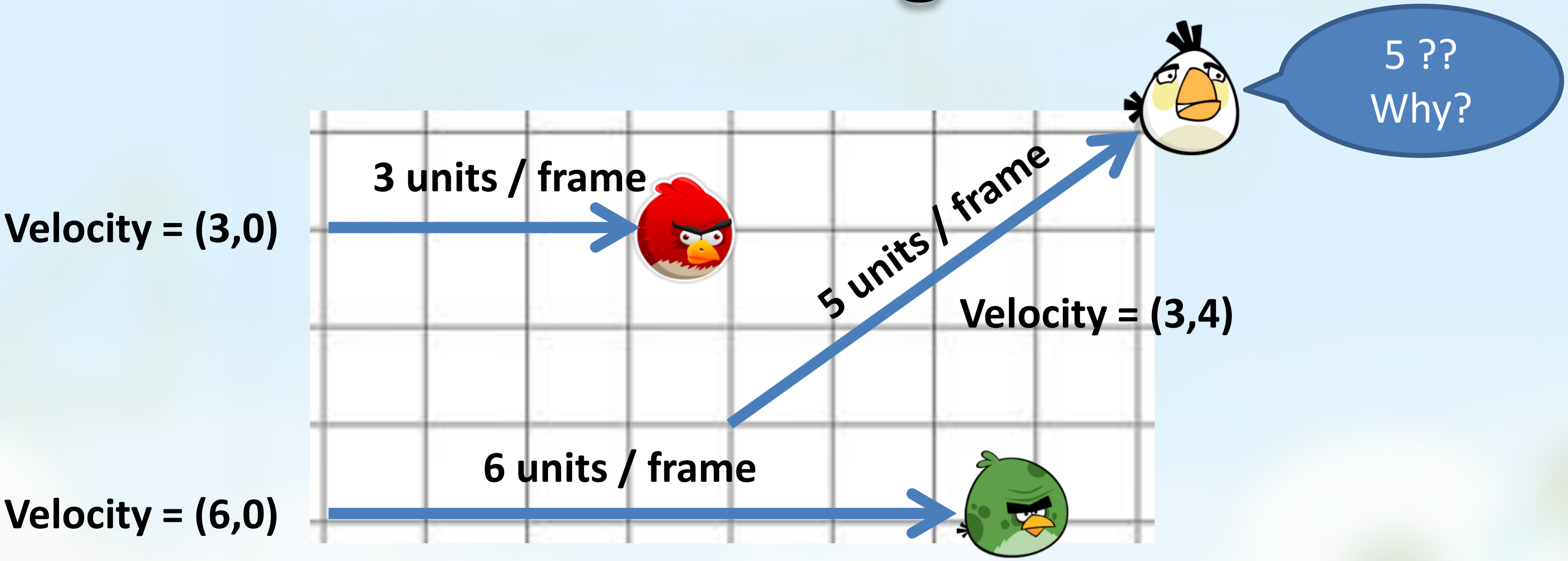
Exercise: fill in the correct details



Vector length

On calculating the length of a vector to understand why we go faster diagonally…

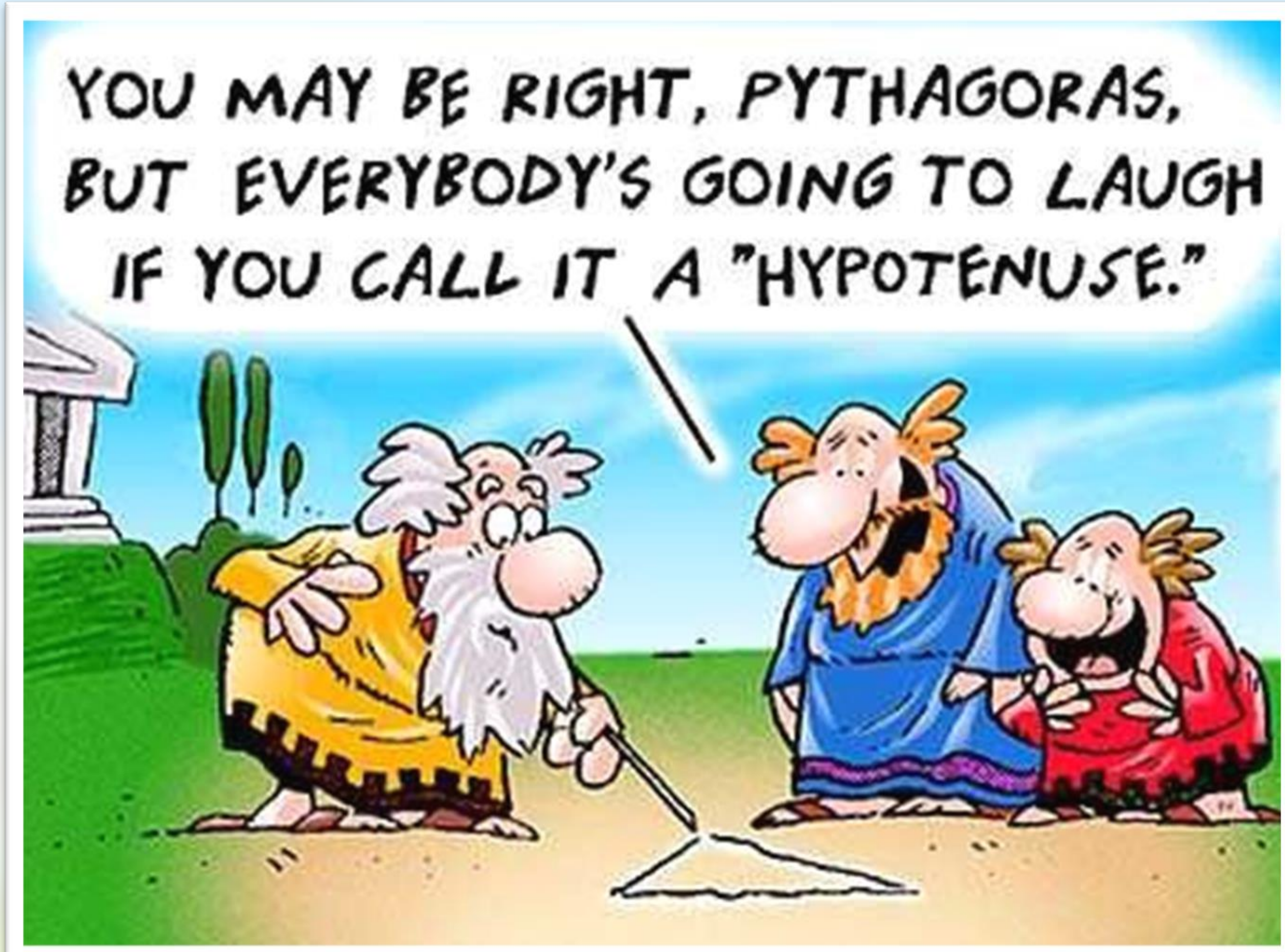
Vector length



- A vector defines both a direction and a length
- For velocity, vector length corresponds with speed
- The *length* of vector $\vec{v}$ is called *magnitude*, in textbook notation written as $|\vec{v}|$
- How can we calculate the length?

*direction == orientation / magnitude == size == length

Pythagoras, 572 BC – 500 BC (!)



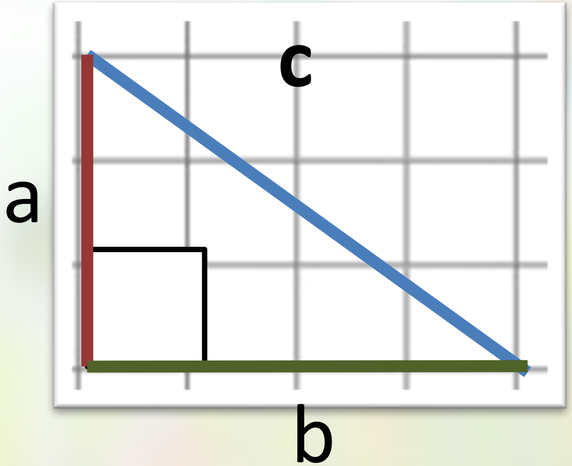
Pythagoras’ Theorem

- A *right-angled triangle* has one 90° angle.
- The side opposite to the 90° angle is called *hypotenuse*.

Pythagoras’s Theorem:

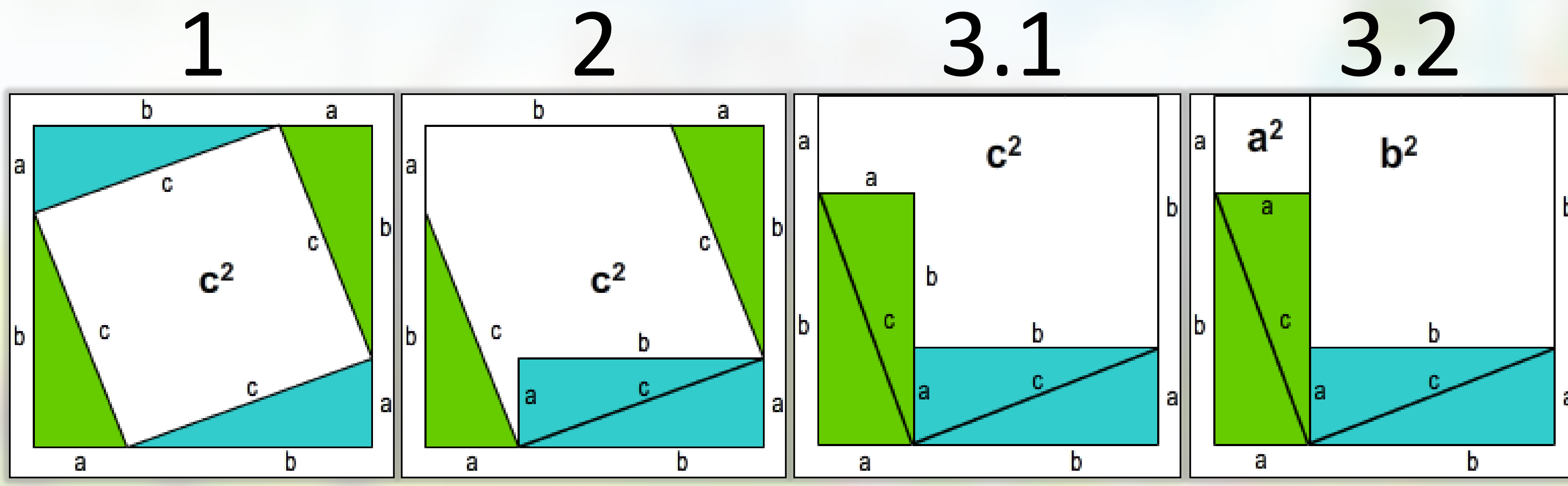
In a right-angled triangle, where *c* is the length of the hypotenuse and *a* and *b* are the lengths of the other sides, it holds that:

$$a^2 + b^2 = c^2$$



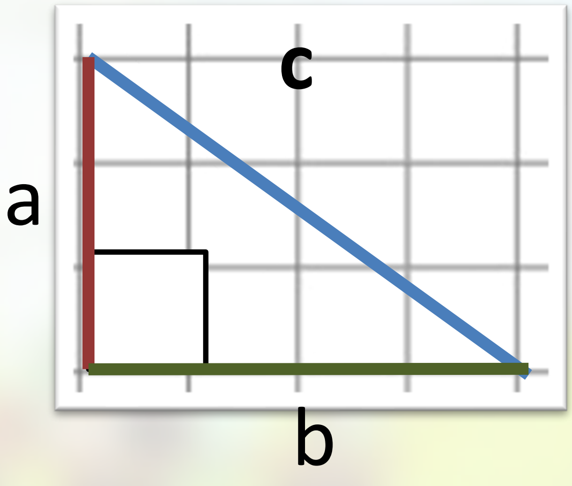
Pythagoras's Theorem – Proof sketch

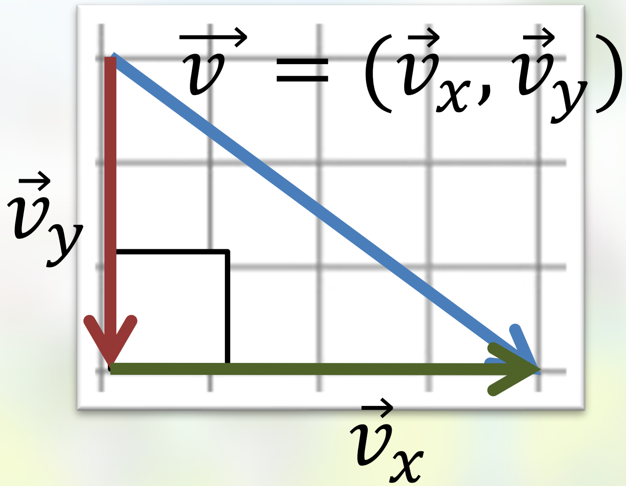
- Draw an outer rectangle with area: (a+b)*(a+b)
- Draw an inner rectangle with area: c*c
- Rearrange the colored pieces to demonstrate that the empty area c*c == a*a + b*b



Pythagoras

- Pythagoras's Theorem: $c^2 = a^2 + b^2$
- Applied to vectors: $|\vec{v}|^2 = \vec{v}_x{}^2 + \vec{v}_y{}^2$





Resulting math for vector length

$$|\vec{v}|^2 = {\vec{v}_x}^2 + {\vec{v}_y}^2 \quad \rightarrow \quad |\vec{v}| = \sqrt{{\vec{v}_x}^2 + {\vec{v}_y}^2}$$

- Practice, calculate length of these vectors:
  - (3,0)
  - (0,4)
  - (3,4)
  - (4,5)
  - (5,0)
  - (5,5)

The result of these two prove we are indeed moving faster diagonally

Vector length in code

Implementing Vector length

- Add a method Length() to the Vec2 class:

```
public float Length() {
    return ...;
}
```

- This is part of assignment 1 !
- But this only **gets** the current length !

Setting vector length?

- How can we **change** the length of the velocity vector to enforce a **specific** speed?
  - For example: we want to go south east with a speed of 5 pixels per frame:
    - (5,5) ?
    - Direction is correct, but speed isn't since it's ≈ 7

Changing the vector length

On how we can change the length of a vector to enforce a specific speed

Vector scaling

- Scaling a vector $\vec{v}$ with a factor k *(=scalar)* is written as: $\vec{v}' = k * \vec{v}$
- Or shorter: $\vec{v}' = k\vec{v}$
- Scaling is done component-wise, like adding:

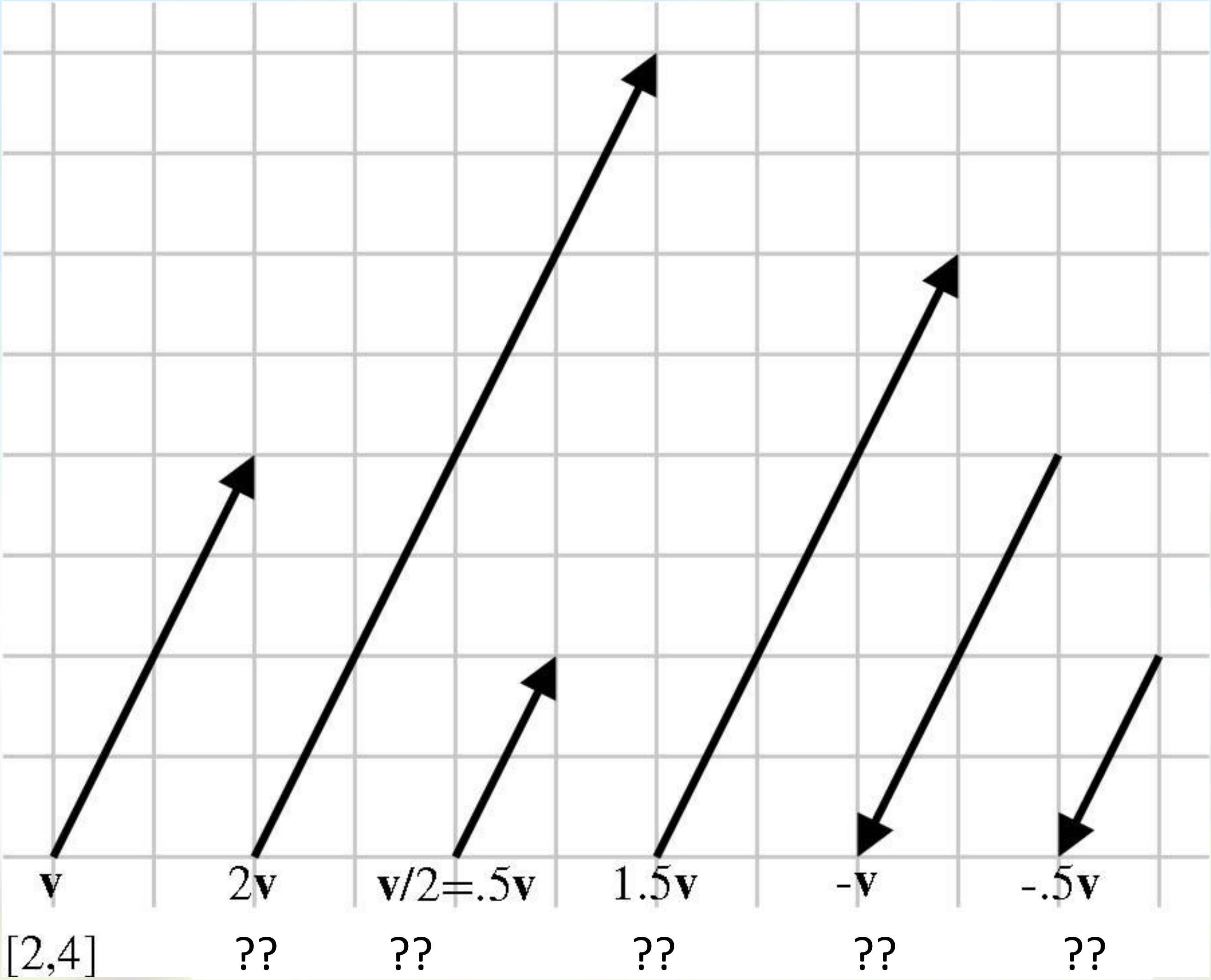
$$k * \vec{v} = (k * \vec{v}_x\,, k * \vec{v}_y)$$

- In code: implement a method that allows you to write:

```
Vec2 scaledVec = 3 * origVec;
```

(= *overload* the * operator.)

Vector scaling practice



Setting vector length

- Now we know how to change the length of a vector **without** changing its direction:
  - through **scaling**

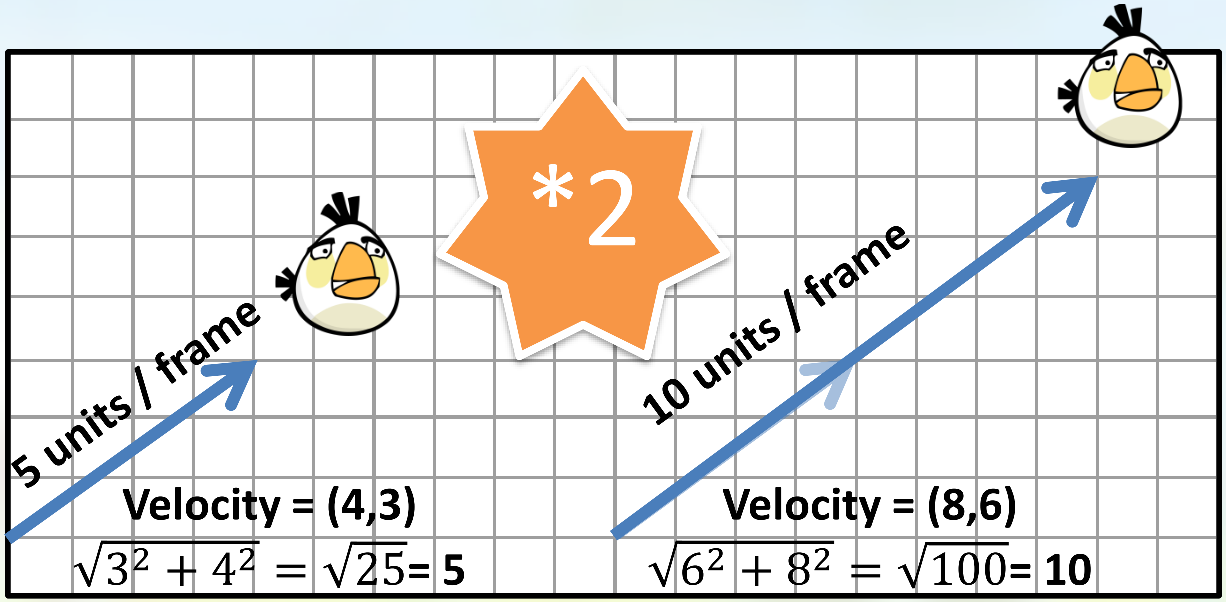
- Which brings us back to our previous question:
  - How can we change (scale) the length of the velocity vector to enforce a **specific** speed?
  - E.g. how much do I have to scale (8,6) to end up with a speed of 5?

Unit vectors

Where we learn to separate length from direction, so that we can travel at any desired speed

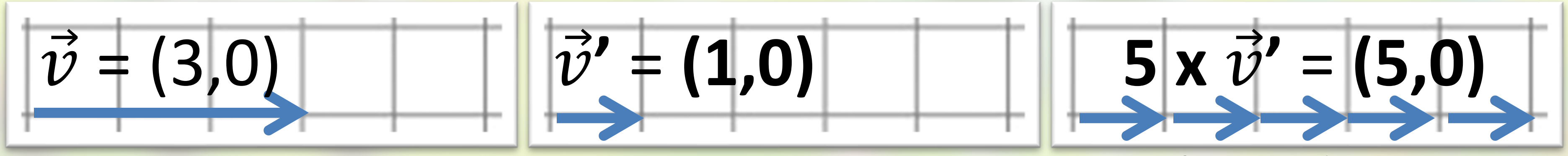
Effect of scaling on length

When we scale a vector,
the length change is proportional to the scale factor:



Scaling a vector to length L?

- So how do we scale the length of vector $\vec{v}$ to a specific length L?
- The easiest approach is:
  1. **First** scale $\vec{v}$ by $\frac{1}{|\vec{v}|}$ so that it has length 1
  2. **Then** scale this new $\vec{v}$ times L so that it has length L
- In pictures for $\vec{v}(3,0)$ and **desired** speed = **5**:



convert to length 1

scale speed times

Normalizing & Unit vector

- Scaling a vector's length to **1** is called **normalizing:**
  - To normalize $\vec{v}$ you scale it by $\frac{1}{|\vec{v}|}$
  - The result is called a **unit vector**, written as $\hat{v}$:
  - $\hat{v} = \frac{1}{|\vec{v}|} * \vec{v}$ which equals $\hat{v} = (\frac{\vec{v}_x}{|\vec{v}|}, \frac{\vec{v}_y}{|\vec{v}|})$

Vector properties: Normalizing

- A little bit of practice, normalize this:
  - $\vec{v}(5,0)$
  - $\vec{v}(1,1)$
  - $\vec{v}(3,4)$

=??????

- How can we verify that a vector is normalized?

Vector normalizing

- Implement a method Normalize():

```
public void Normalize () {
        .....
}
```

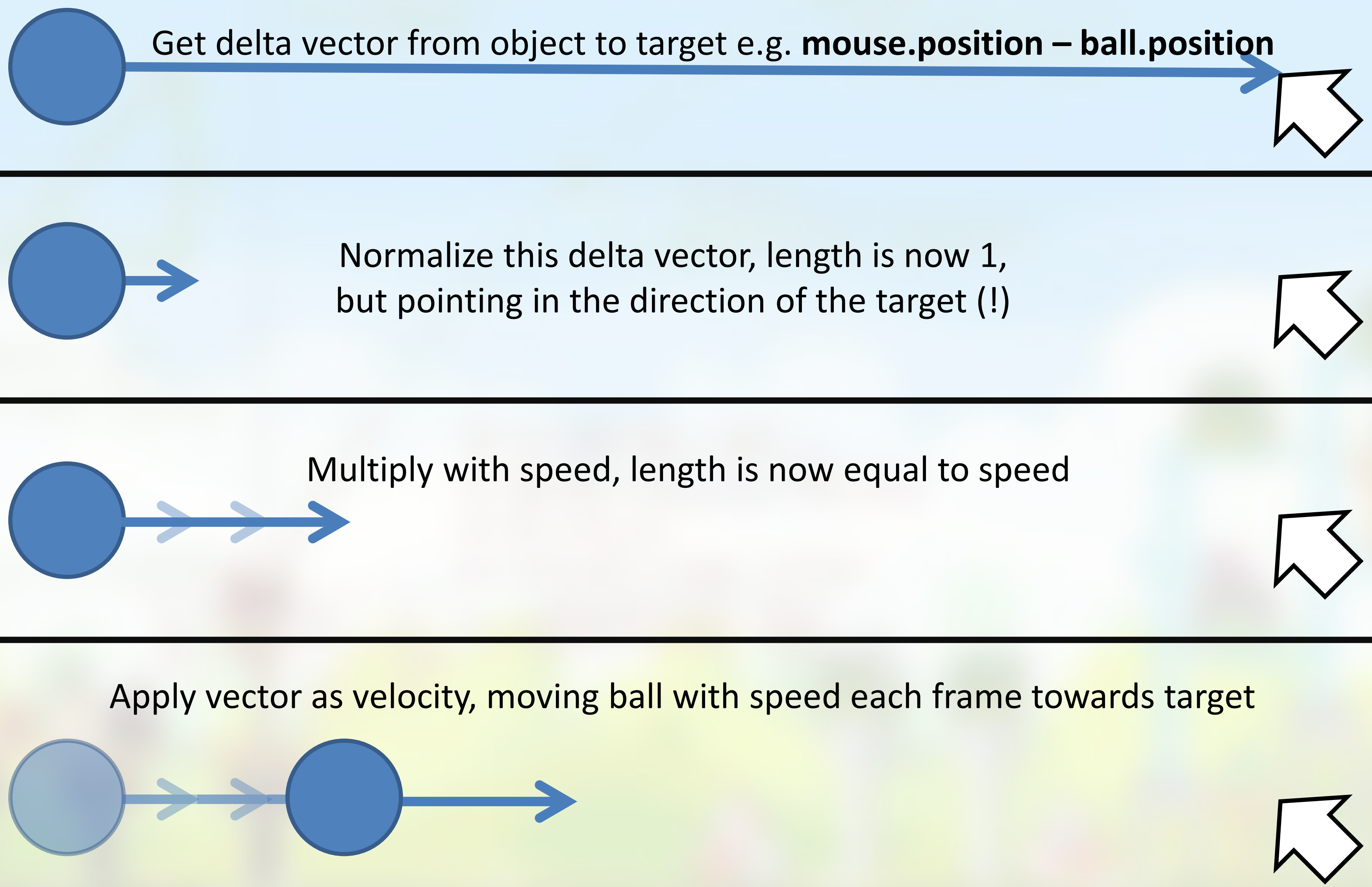
- Watch out for zero length vectors…

Updating example 2

- Now you can fix example 003:
  - set the velocity as before
  - normalize the velocity
  - scale the velocity with desired speed

- Can we also use this knowledge to move **to** a specific **target** at a desired speed?

How can we use it to follow a target?



Mouse Following

- Example 004_ball_follow_mouse:
  - Gets difference vector between ball and mouse
  - Normalizes difference vector (speed is now 1)
  - Multiplies the normalized vector with desired speed
  - Increases desired speed every frame
- Result:
  - Ball follows mouse at desired speed
- Part of assignment 1!

Lab work / Home work

- Complete the short test on Blackboard about vector basics.
- Doing this *unlocks* Assignment 1:

  Highly recommended:
  - Build the start of your Vec2 struct
  - Test whether you did this correctly → *unit tests*
  - Implementing small mouse following “game”
- Show this to your lab teacher for *feedback!*

Look ahead: other things you can do

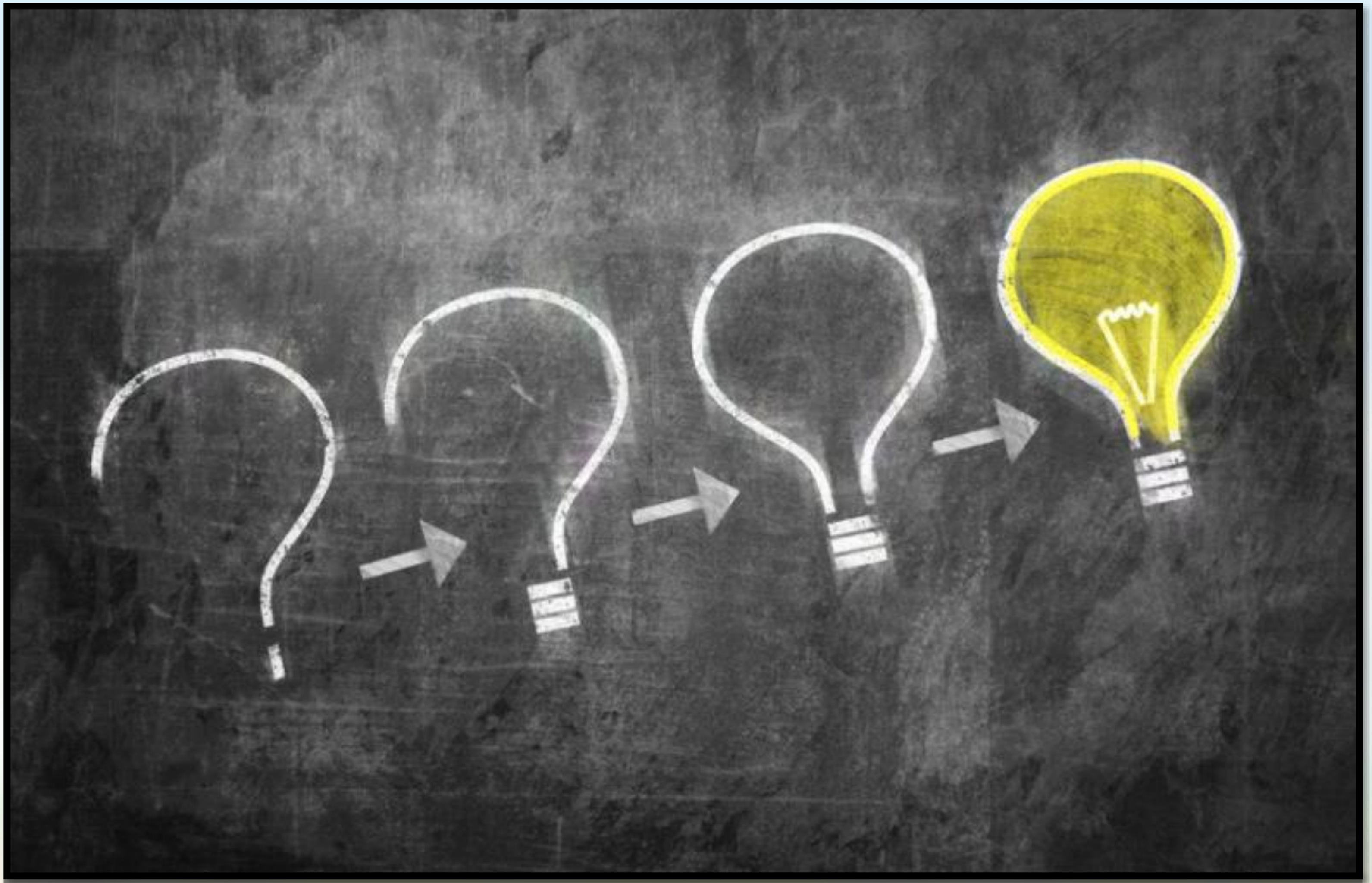
- Add *acceleration:* don't "overwrite" velocity every frame, but change it by *adding* to it (based on gravity, magnetism, spring constraints, …).
- Add *friction / viscosity:* scale the velocity every frame by a factor $0 < f < 1$, such that the object slows down gradually.
- *(Advanced:) Leading a target:* don't aim at the target's current position, but where it will be when the bullet arrives.

- <demo's>

Extra Resources Week 1

Basics:

- Introduction to vectors, by 3Blue1Brown:
  - https://www.youtube.com/watch?v=fNk_zzaMoSs
- See also the links in the manual, e.g.:
  - Math for Game Developers (check the first few video's): https://www.youtube.com/channel/UCEhBM2x5MG9-e_JSOzU068w
- Now that you know the right search terms, google can find a lot more info for you. ☺



Additional proofs

Scale proof

- Scaling $\vec{v}$ by k, changes the length of $\vec{v}$ by k:
  - If $\vec{v}' = k\vec{v}$, then $|\vec{v}'| = |k\vec{v}| = k|\vec{v}|$

$\vec{v}' = k\vec{v} = (k\vec{v}_x, k\vec{v}_y)$	1. According to definition of scaling
$\lvert k\vec{v}\rvert = \sqrt{(k\vec{v}_x)^2+(k\vec{v}_y)^2}$	2. According to pythagoras
$\lvert k\vec{v}\rvert = \sqrt{k^2\vec{v}_x{}^2 + k^2\vec{v}_y{}^2}$	3. According to $(k\vec{v}_x)^2 = k\vec{v}_x k\vec{v}_x = kk\vec{v}_x\vec{v}_x = k^2\vec{v}_x{}^2$
$\lvert k\vec{v}\rvert = \sqrt{k^2 * (\vec{v}_x{}^2 + \vec{v}_y{}^2)}$	4. Bring k^2 outside of the bracket
$\lvert k\vec{v}\rvert = \sqrt{k^2} * \sqrt{(\vec{v}_x{}^2 + \vec{v}_y{}^2)}$	5. Split roots
$\lvert k\vec{v}\rvert = k * \sqrt{\vec{v}_x{}^2 + \vec{v}_y{}^2} = k\lvert\vec{v}\rvert$	6. Simplify and use pythagoras definition

Normalizing & Unit vector

- To normalize $\vec{v}$ you scale it by $\frac{1}{|\vec{v}|}$
- The result is a **unit vector** written as $\hat{v}$,
  - $\hat{v} = \frac{1}{|v|} * v$ which equals $\hat{v} = (\frac{v_x}{|v|}, \frac{v_y}{|v|})$
- Proof:
  - Given $|\vec{v}'| = |k\vec{v}|$ $=k|\vec{v}|$ and k = $\frac{1}{|\vec{v}|}$
  - Then $|\hat{v}| = |k\vec{v}|$ = $k|\vec{v}|$ = $\frac{1}{|\vec{v}|}$ * $|\vec{v}|$ = $\frac{|\vec{v}|}{|\vec{v}|}$ = 1