Game Development Class

2. Gameplay Programming

par Panagiotis Tsiapkolis (Panthavma)

le March 3rd 2024

– Gameplay Programming

* It's about the logic of the mechanics, the rules of the game world
* Today, we're gonne see the keys to understanding and how to think in order to program efficiently
* We're going to see several practical examples, which are going to be immediately useful
* It's a HUGE session with a lot of concepts, don't hesitate to go back to it later and focus on the global idea now.

– QUESTION:

What is the best engine?

* Unreal Engine
* Unity
* Godot
* Castagne

– ANSWER:

Trap question, each will have advantages for various games and teams.

Tips

– Programming tips

* Programming is explaining a procedure, step by step
* You need to decompose each steap more and more, until you can explain the procedure in an understandable way to the computer, using functions.
* It's tricky until you get it, but practice makes perfect.
* In general, it is most useful to think before doing: this is one of the main mistakes programmers do.

Programming tips: example

* Exemple: We want to make a character walk towards the right when we press the corresponding key.
* How do you know the key is pressed?
  - We check that the "right arrow" key is currently pressed
  - The engine knows how to tell if a key is pressed, therefore we can know it
* How to walk?
  - We want to move the character towards the right.
  - We can move a character by changing its position, by adding its current position and a movement.
  - We can get the position of an object through the engine because it knows these concepts and has functions for it.
  - We compute the movement (ex. 20 pixels per second), and we add it to the position.
  - We tell the engine to use this as the new position.

– Clean code

* Better do more functions, avoid having too much code in each one.
* Organize your code in logical blocks.
* Use comments, not to repeat what is written, but more so to write the general throughts (think of a Director's Cut).
* We're gonna go over these more in depth in the code architecture session!

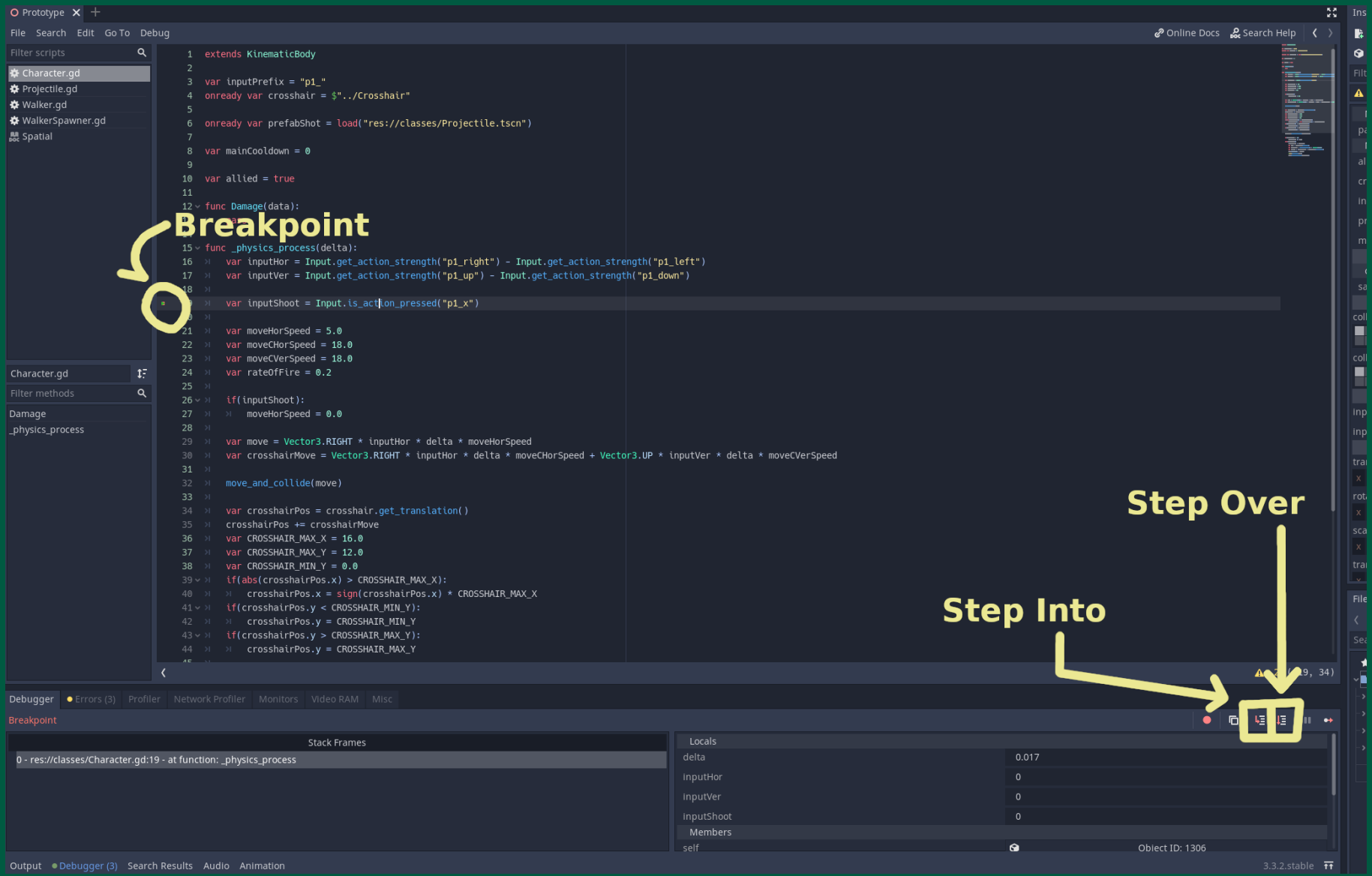
– Debugging

* A bug happens when the program works, but not as expected.
* It's one of the main work of programmers and QA
* To solve them, you must isolate the parts through several tests, like a detective
* Example questions to ask youself :
  - Can I reproduce the bug every time with a specific procedure (called Repro Steps)?
  - Can I take out some of them and still have the bug happen? (Isolation)
  - Does every sub-step do its job correctly? (Examination)

– Debugging: Extracting info

* In order to solve it, you often need to extract information.
* One method: logs. Writing text in the console (printf debugging), or in a file can help retrace execution.
  - This is a post-mortem method: you analyze it after the bug happened, which can help even if you weren't looking for it.
* Another one: using a debugger. We link to the program during its execution, and we advance bit by bit in the program's execution.

— Debugging : Breakpoints




```gdscript
extends KinematicBody

var inputPrefix = "p1_"
onready var crosshair = $"../Crosshair"

onready var prefabShot = load("res://classes/Projectile.tscn")

var mainCooldown = 0

var allied = true

func Damage(data):
    pass

func _physics_process(delta):
    var inputHor = Input.get_action_strength("p1_right") - Input.get_action_strength("p1_left")
    var inputVer = Input.get_action_strength("p1_up") - Input.get_action_strength("p1_down")

    var inputShoot = Input.is_action_pressed("p1_x")

    var moveHorSpeed = 5.0
    var moveCHorSpeed = 18.0
    var moveCVerSpeed = 18.0
    var rateOfFire = 0.2

    if(inputShoot):
        moveHorSpeed = 0.0

    var move = Vector3.RIGHT * inputHor * delta * moveHorSpeed
    var crosshairMove = Vector3.RIGHT * inputHor * delta * moveCHorSpeed + Vector3.UP * inputVer * delta * moveCVerSpeed

    move_and_collide(move)

    var crosshairPos = crosshair.get_translation()
    crosshairPos += crosshairMove
    var CROSSHAIR_MAX_X = 16.0
    var CROSSHAIR_MAX_Y = 12.0
    var CROSSHAIR_MIN_Y = 0.0
    if(abs(crosshairPos.x) > CROSSHAIR_MAX_X):
        crosshairPos.x = sign(crosshairPos.x) * CROSSHAIR_MAX_X
    if(crosshairPos.y < CROSSHAIR_MIN_Y):
        crosshairPos.y = CROSSHAIR_MIN_Y
    if(crosshairPos.y > CROSSHAIR_MAX_Y):
        crosshairPos.y = CROSSHAIR_MAX_Y
```

Main Loop

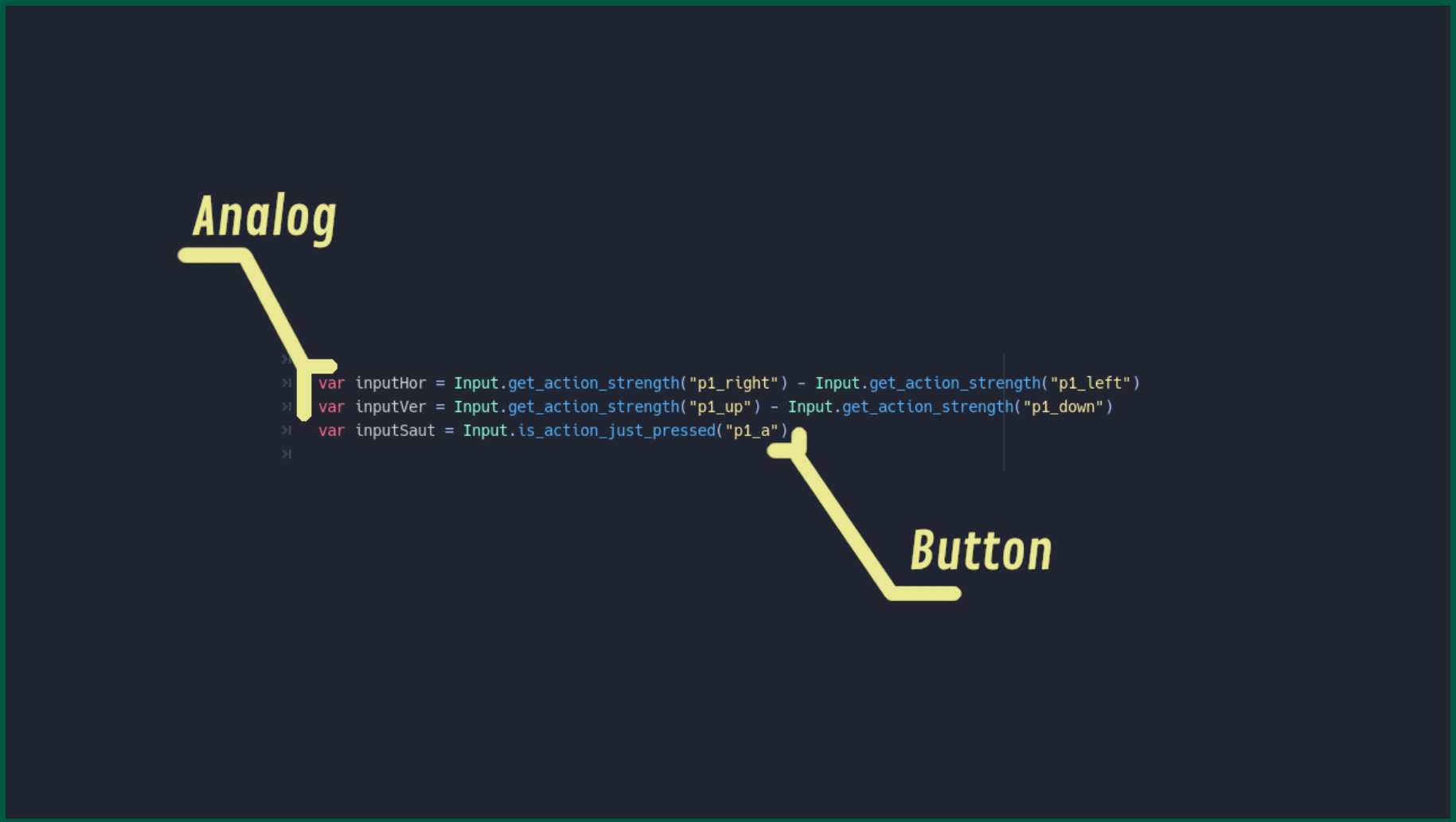
– Main Loop

* The game is based around a main loop that runs for the whole game.
* In modern games, we often have two loops: graphics, and physics.
* The various entities (characters, object, projectiles...) update every loop.

– Input methods

* We need to check the state of buttons every loop: it's the "polling" method.
  - It's opposed to event-based programming, where actions are provoked by discrete events like button presses.
* Engines offer interfaces to replace that with "actions", which offer a degree of indirection and thus flexibility (several buttons for one action, remapping).

– Input Methods: Example



– Additional Note: Buffers

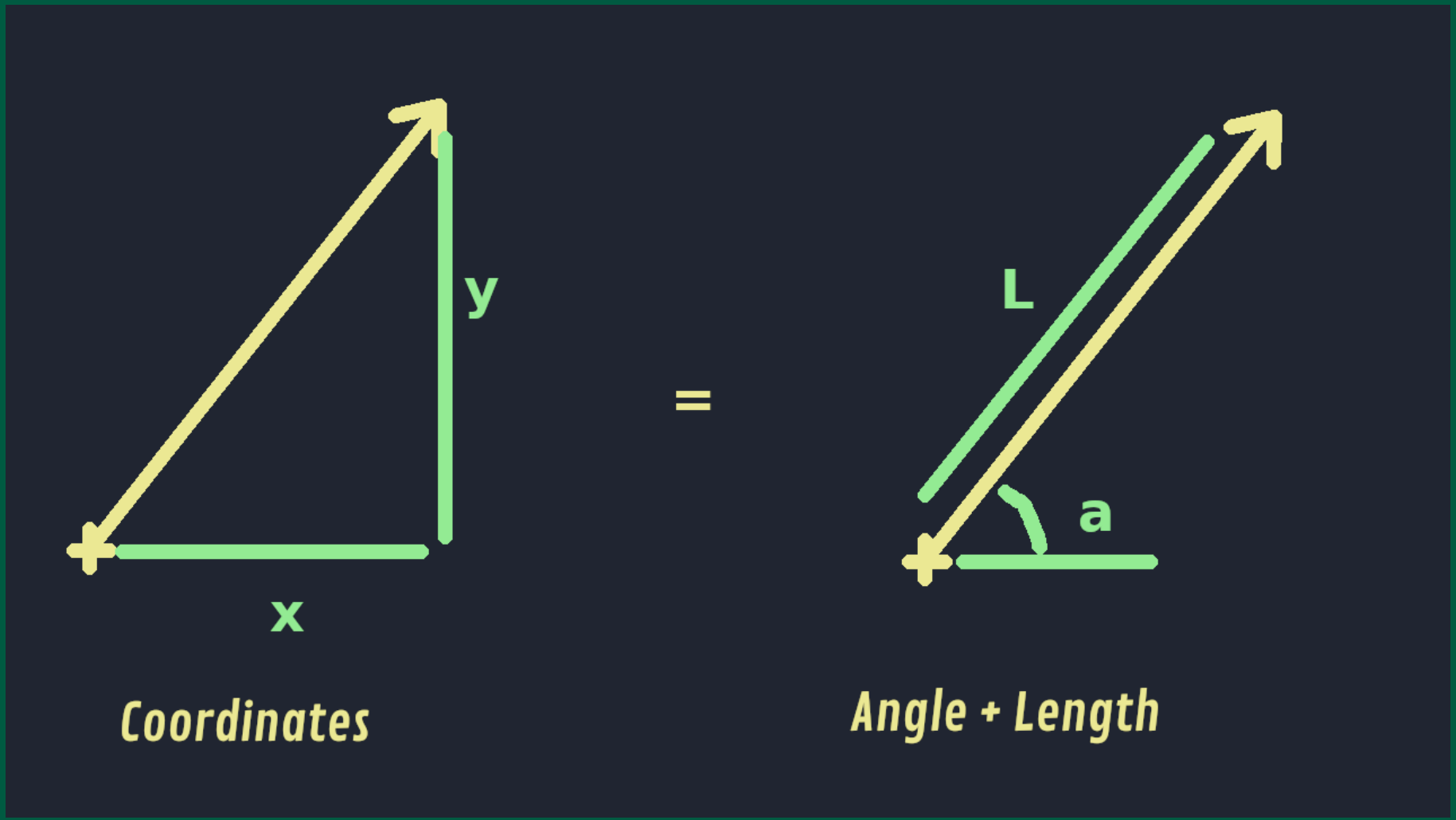
* Note: It's a sensitive part of control, and methods exist to make input more natural.
* Buffers allow pressing buttons in advance, and avoid inputs getting "eaten" while not available.
* For example: landing and rejumping. A buffer allows pressing the button a bit before landing and jump as soon as you hit the ground. It allows having a more approximative timing and thus better control the character.
* As an additional note, you can react to a button press or a button release ("negative edge"). Doing it like this allows for better handling of multiple actions on a single button (hold vs press), but adds lag.

Physics

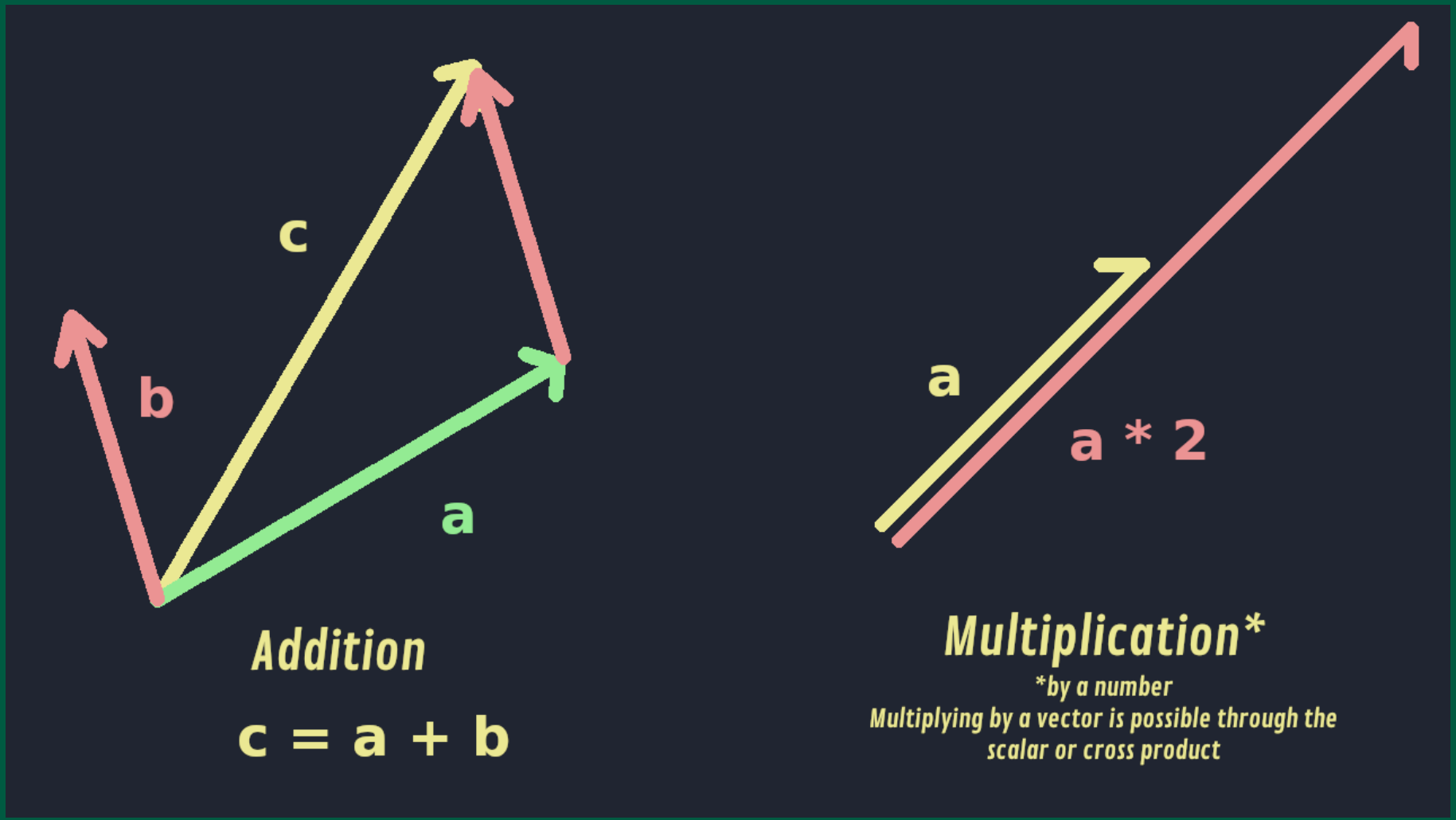
– Reminder: Linear Algebra

* We'll need a very useful mathematical tool: vectors.
* I'll talk about 2D vectors, but the concepts and formulas are the same in 3D.
* There are other algrebra concepts that are very useful but more advanced, and that won't be necessary at a basic level (dot/cross product, transformation matrices...).

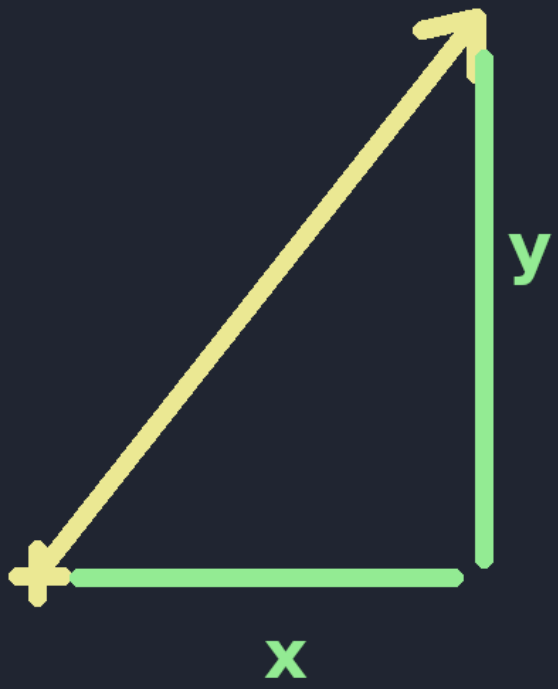
– Vectors (1/3)



– Vectors (2/3)


c
b
a
Addition
c = a + b
a
a * 2
Multiplication*
*by a number
Multiplying by a vector is possible through the scalar or cross product

– Vectors (3/3)



Length / Norm:

$$\sqrt{x^2 + y^2}$$

Normalize = divide by the norm
= set length to 1

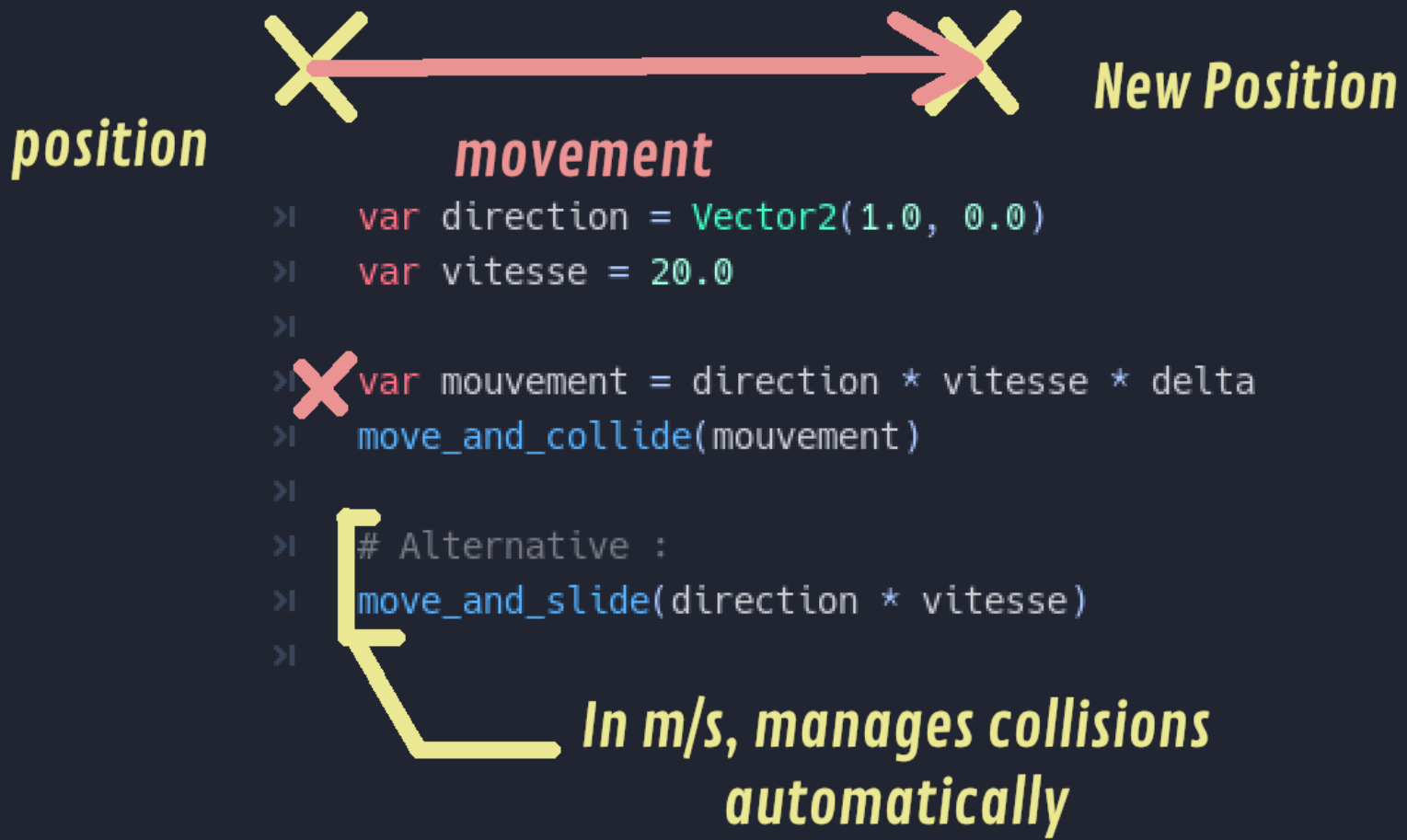
– Reminder: Kinematics

* Kinematics evaluate the evolution of position with external forces. Some tools are going to be useful.
* Speed is the change of position, while acceleration is the change of speed.
* Dimensional analysis is about doing an equation only using the units of the variables. Its useful to check if the formula is correct.
  - Position: m ; Speed: m/s ; Acceleration: m/s2
  - This means we can go from speed to position by multiplying with time.

– Simple movement

* We can start by making a simple constant movement.
* We want to find the movement vector for this frame (loop).
* This is doable by multiplying the direction of movement, speed, and delta time.
  - Direction (normalized vector = length 1) allows to direct the character easily.
  - Speed (in m/s or pixels/s) is a variable we filled in ourselves.
  - Delta time (time difference) is the time since the last frame. It allows to have an identical result despite the difference of execution time.

– Mouvement simple : Exemple



– Movement with acceleration

* Movement is still very artificial, we'll add acceleration to get a more plausible result.
* We'll change the speed used in the last formula to add an acceleration and brake force.
* The formula here consists in adding acceleration or break, multiplied by delta time. The subtlety here is the brake is a negative value.
* We'll also add lower and superior limits ("clamp") in order to have a max speed and be able to stop.

– Movement with acceleration: Example

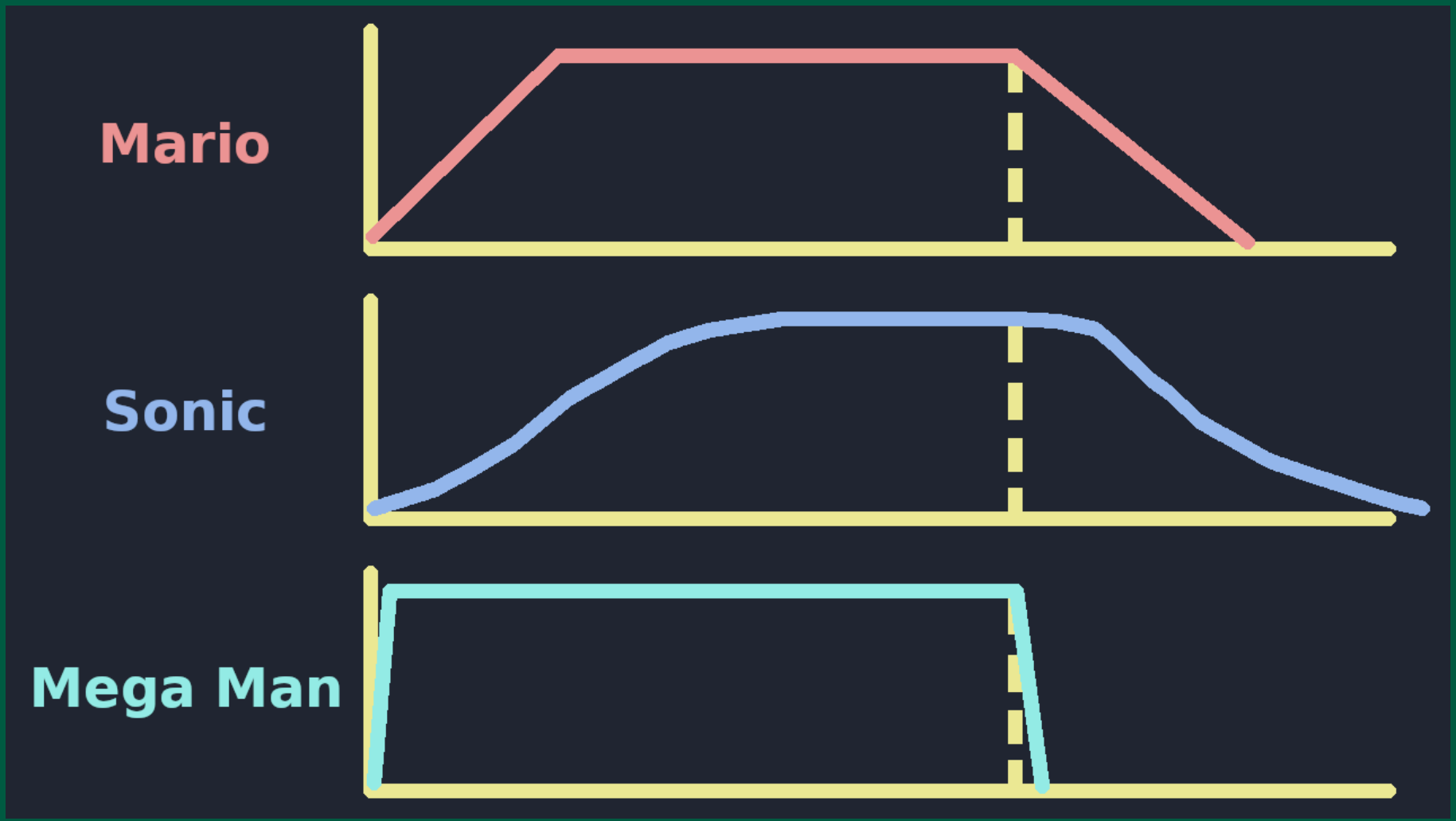
```
if(inputHor > 0.0):
    acceleration = 40.0
else:
    acceleration = -80.0

vitesse = vitesse + acceleration * delta
vitesse = clamp(vitesse, 0.0, vitesseMax)
```

Acceleration

Brake (Negative)

0.0 <= speed <= speedMax

– Control differences depending on algorithm: Example



– Rigidbody vs Kinematics

* The behavior here is said to be "Kinematic", where we control the entity directly.
* A physical behavior ("Rigidbody") will however work from the physics engine directly, and will be controlled through forces you add in Newtons (1N is the required force to give an accelation of 1m/s to a mass of 1kg)
* The simple method here consists in take the acceleration we want to give this frame, and multiply by the object weight.
* I recomment using Kinematic for most games, but it's my personnal preference.

Entities

– Object Oriented Programming

* Object-Oriented Programming is a paradygm present in a lot of modern languages, allowing simple reuse of code.
* The idea is to organize functions and variables in "classes", from which we may create "object", which will have their own associated variables.
* Example: A house. The class is the architectural drawing, how to build the house. From this plan, you may create several houses, which can then diverge through their variables (for example, paint a house blue instead of red).
* We'll see that in more detail in the code architecture session.

Entities in games

* Games tend to use an entity system, which goes a bit further.
* Entities are composed from one or more component, which have different roles (graphics, physics, behavior, audio, etc...)
* We may separate the concept of "prefabs" (classes, models) from "instances" (objects).
* We can then populate our levels with these entities.

Entities

* Projectiles, Collisions
* Object interfaces, global management
* State Machine

– Instanciation

* We have a rifle and wish to fire. To do so, we'll create a bullet.
* We thus need to create a prefab, which means loading the entity from disk, copy it, set its parameters, and add it to the world.

– Instanciation: Exemple

```gdscript
var prefabTir = preload("res://classes/Projectile.tscn")

func Tirer(directionTir):
    var projectile = prefabTir.instance()

    var positionDeTir = get_translation() + Vector3(0.0, 1.0, 0.0)
    projectile.set_translation(positionDeTir)
    projectile.direction = directionTir
    projectile.vitesse = 60.0
    projectile.degats = 12

    get_node("..").add_child(projectile)
```

Load

Create

Setup

Add in the world

– Colliders

* A collision happens when two colliders touch.
* A collider is a set of shapes, often constructed from primitives (cube, spheres, capsules, etc) or from a mesh (for static objects like terrain).
* The collision detection algorithm can be complex, and is part of the engine.

– Types of physical entities

* KinematicBody : Object following kinematics.
* RigidBody : Objects with a realistic physics behavior.
* StaticBody : Object that doesn't move (more optimized).
* Area : Intangible object that can detect what enters inside.

– QUESTION:

Which physics entity type do we have to use?

* KinematicBody
* RigidBody
* StaticBody
* Area

– ANSWER:

Area! We don't want the bullet to be solid, just to act on the shot object.

– Raycast vs Zones

* An alternative way of detecting collisions is sending a ray, like a laser.
* This ray has a potentially limited range, and returns the first object fount.
* It more adapted for instantaneous weapons ("hitscan", like a pistol) rather than missiles ("projectiles", like a bazooka).

Entities

* Projectiles, Collisions
* **Object interfaces, global management**
* State Machine

– Interfaces

* The concept of interfaces in object-oriented programming is the functions and variables of a class meant to be used by other objects ("public functions"). (Simplified)
* Godot uses duck-typing, meaning you may use those interface functions directly. (Simplified)
* Therefore, once you have a link to an object (here gathered through collision detection), we can call a function on that object (here a damage function).

– Damage through interface

– Global references: singletons

* Another way of getting references and data is through a global object (be careful with their use).
* The concept is called "Singleton" in object-oriented programming, and is available as Autoloads on Godot.

– Enemy AI

* AIs in games tend to be somewhat simple. We'll create an easy enemy:
* They start by walking for a couple meters
* Then, they fire on the player (gathered through the singleton)

– Walker AI

```
func _physics_process(delta):
    if(remainingMove > 0):                                    Movement
        var moveDist = min(speed * delta, remainingMove)
        translate(Vector3.RIGHT * moveDist)
        remainingMove -= moveDist
    else:
        timeUntilNextShot -= delta                            Shooting
        if(timeUntilNextShot <= 0):
            timeUntilNextShot += timeBetweenShots
            var shootPoint = get_translation() + Vector3(0,1,0)
            var shootDir = (character.get_translation() + Vector3(0,1,0) - sho
            var shot = prefabShot.instance()                        - shootPoint).normalized()
            $"..".add_child(shot)
            shot.direction = shootDir
            shot.allied = false
            shot.set_translation(shootPoint)
```

Entities

* Projectiles, Collisions
* Object interfaces, global management
* **State Machine**

What's a state machine?

- A state machine is a tool that defines states, and transition between those states.
- An example: we can have states like "grounded", "walk", "crouch", "got hit", etc...
- Making a character can become maze-like without this tool, as you'll have to manage lots of state variables, some not compatible...
- This point will be expanded upon in the session on code architecture.

State Machine: Implementation

```
enum ETATS {
    Debout, EnTrainDeTirer, ATerre
}



if(etat == ETATS.Debout):
    Deplacement()
    Visee()
elif(etat == ETATS.EnTrainDeTirer):
    Visee()
    Tir()
elif(etat == ETATS.ATerre):
    pass
```

Putting it in practice

Proto: Character

```gdscript
extends KinematicBody2D

export var playerInputPrefix = "p1_"

var VITESSE_COURSE = 140.0
var ACCEL_COURSE = 550.0
var PV_MAX = 1200

var _vitesse = Vector2()
var _direction = Vector2(1,0)
var _pv = PV_MAX

onready var arme = $Pistolet

func _physics_process(delta):
    # Lecture des entrées
    var inputMouvement = Vector2(
        Input.get_action_strength(playerInputPrefix+"right") - Input.get_action_strength(playerInputPrefix+"left"),
        Input.get_action_strength(playerInputPrefix+"down") - Input.get_action_strength(playerInputPrefix+"up"))

    if(inputMouvement.length_squared() > 1):
        inputMouvement = inputMouvement.normalized()
    if(inputMouvement.length_squared() > 0):
        _direction = inputMouvement.normalized()

    var vitesseCible = inputMouvement * VITESSE_COURSE
    _vitesse = _vitesse.move_toward(vitesseCible, ACCEL_COURSE * delta)

    if(_vitesse.length_squared() > VITESSE_COURSE*VITESSE_COURSE):
        _vitesse = _vitesse.normalized() * VITESSE_COURSE

    move_and_slide(_vitesse)

    arme.Gestion(self, delta)

func PrendreCoup(degats):
    _pv -= degats

```

– Proto: Weapon

```gdscript
extends "Arme.gd"

export var PREFAB_TIR = "res://armes/pistolet/Balle.tscn"

export var VITESSE_TIR = 600.0
export var FREQ_TIR = 3.0
export var DEGATS = 100

var _prefabTir
func _ready():
    _prefabTir = load(PREFAB_TIR)

var _tempsCooldown = 0.0
func Gestion(perso, delta):
    var inputTir = Input.is_action_pressed(perso.playerInputPrefix+"a")

    if(DoitTirer(perso, inputTir, delta)):
        Tir(perso)

func DoitTirer(perso, inputTir, delta):
    if(_tempsCooldown <= 0):
        if(inputTir):
            _tempsCooldown = 1.0/FREQ_TIR
        return inputTir
    else:
        _tempsCooldown -= delta
        return false

func Tir(perso):
    var tir = _prefabTir.instance()
    tir.set_position(perso.get_position())
    tir.vitesse = VITESSE_TIR
    tir.direction = perso._direction
    tir.perso = perso
    tir.degats = DEGATS
    $"../..".add_child(tir)

```

```gdscript
extends Node2D

export var MENU_NOM = "NOM ARME"
export var MENU_DESCRIPTION = """DESCRIPTION"""


func Gestion(perso, delta):
    pass

```

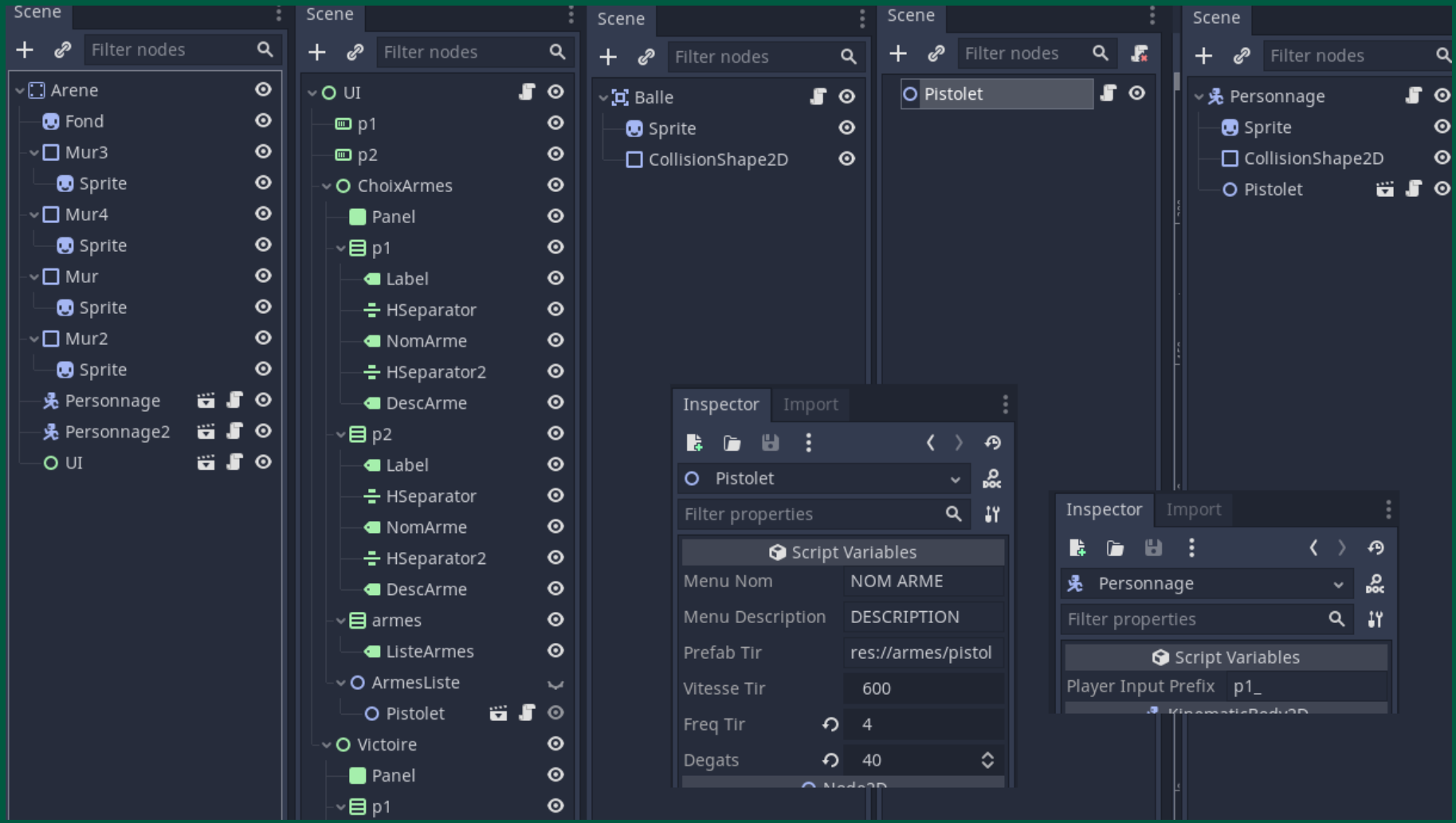
– Proto: Shot

```gdscript
extends Area2D

var vitesse = 0.0
var direction = Vector2(1,0)
var perso = null
var degats = 100

func _ready():
    connect("body_entered", self, "on_body_entered")

func _physics_process(delta):
    global_translate(vitesse*direction*delta)

func on_body_entered(body):
    if(perso != null and body == perso):
        return
    if(body.has_method("PrendreCoup")):
        body.PrendreCoup(degats)
    queue_free()

```

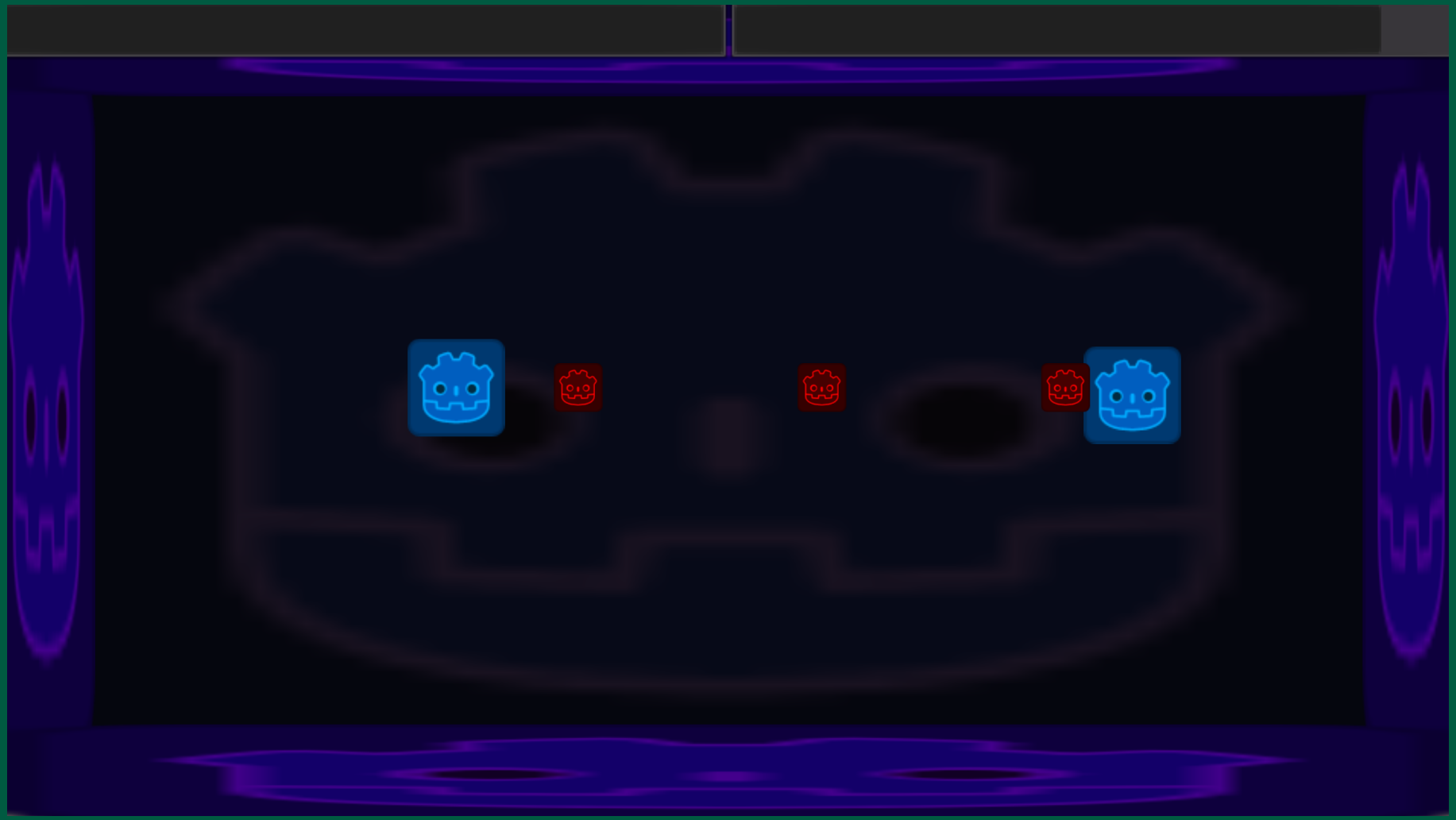
– Proto: Menu

```gdscript
extends Control

var modeEntrainement = true
enum ETAT_JEU {
    CHOIX_PERSO, JEU, ECRAN_FIN
}

onready var p1 = $"../Personnage"
onready var p2 = $"../Personnage2"

var _etat = ETAT_JEU.JEU

func _ready():
    ChangerEtat(ETAT_JEU.JEU)

func _process(delta):
    if(_etat == ETAT_JEU.JEU):
        GameProcess(delta)
    elif(_etat == ETAT_JEU.CHOIX_PERSO):
        MenuChoix(delta)
    elif(_etat == ETAT_JEU.ECRAN_FIN):
        EcranVictoire(delta)

func GameProcess(delta):
    UpdatePlayer(p1, $p1)
    UpdatePlayer(p2, $p2)

func UpdatePlayer(perso, ui):
    ui.set_max(perso.PV_MAX)
    ui.set_value(perso._pv)

func ChangerEtat(etat):
    _etat = etat
    $ChoixArmes.set_visible(etat == ETAT_JEU.CHOIX_PERSO)
    $Victoire.set_visible(etat == ETAT_JEU.ECRAN_FIN)
    _tempsVictoire = 2.0


func MenuChoix(delta):
    pass

var _tempsVictoire
func EcranVictoire(delta):
    if(_tempsVictoire <= 0):
        ChangerEtat(ETAT_JEU.CHOIX_PERSO)
    _tempsVictoire -= delta

```

– Proto: Scene


Scene
Filter nodes
Arene
Fond
Mur3
Sprite
Mur4
Sprite
Mur
Sprite
Mur2
Sprite
Personnage
Personnage2
UI
Scene
Filter nodes
UI
p1
p2
ChoixArmes
Panel
p1
Label
HSeparator
NomArme
HSeparator2
DescArme
p2
Label
HSeparator
NomArme
HSeparator2
DescArme
armes
ListeArmes
ArmesListe
Pistolet
Victoire
Panel
p1
Scene
Filter nodes
Balle
Sprite
CollisionShape2D
Scene
Filter nodes
Pistolet
Scene
Filter nodes
Personnage
Sprite
CollisionShape2D
Pistolet
Inspector
Import
Pistolet
Filter properties
Script Variables
Menu Nom
NOM ARME
Menu Description
DESCRIPTION
Prefab Tir
res://armes/pistol
Vitesse Tir
600
Freq Tir
4
Degats
40
Inspector
Import
Personnage
Filter properties
Script Variables
Player Input Prefix
p1_

– Proto: Result



Questions

* Discord : https://discord.gg/CWjWfC9K9T
* Website : https://panthavma.com/gamedevclass/
* Next Time : Static 2D and 3D Art (March 10th)
* On the side : Try to code a prototype, no pressure. Don't hesitate asking questions on Discord!