Cours de création de jeux vidéo

2. Programmation Gameplay

par Panagiotis Tsiapkolis (Panthavma)
le 18 Février 2024

Programmation Gameplay

* Il s'agit de la logique des mécaniques, les règles du monde du jeu
* Aujourd'hui on va voir les clés et façons de penser pour le faire efficacement
* On va le faire à travers plusieurs examples pratiques, qui seront immédiatement utiles pour vous.
* Il s'agit d'une séance massive avec plein de concepts, n'hésitez pas à référencer les slides par la suite et vous concentrer sur comprendre l'idée globale.

Conseils

Conseils de programmation

* Programmer, c'est expliquer une procédure complète étape par étape.
* Il est nécéssaire de décomposer chaque étape de plus en plus, jusqu'à arriver à expliquer la procédure à un niveau que l'ordinateur comprend, à l'aide de fonctions.
* C'est assez déroutant au début tant qu'on a pas eu le déclic, mais à force de le faire ça devient naturel.
* En général, il est très utile de prévoir avant de faire : c'est une des erreurs principales des programmeurs.

Conseils de programmation : exemple

* Exemple : On veut faire marcher un personnage vers la droite quand on appuie sur le bouton correspondant.
* Comment on sait qu'on appuie sur le bouton ?
  - On vérifie que le bouton "Flèche Droite" est appuyé à ce moment
  - Le moteur sait dire si un bouton est appuyé, on peut donc récupérer l'information.
* Comment marcher ?
  - Il faut déplacer le personnage vers la droite.
  - On peut déplacer un objet en changeant sa position, en additionnant sa position actuelle et un déplacement.
  - On peut récuperer la position d'un objet via le moteur car il connait ces concepts et a des fonctions associées.
  - On calcule le déplacement (par exemple, 20 pixels chaque seconde), et on l'additionne à la position.
  - On dit au moteur d'utiliser le résultat comme nouvelle position.

Code Propre

* Faire plein de fonctions, éviter d'avoir trop de code dans une seule fonction.
* Organiser en blocs logiques.
* Utiliser des commentaires, pas pour répéter ce qui est marqué, mais plutôt pour marquer les pensées globales (comme dans un Director's Cut).
* On en parlera plus lors de la séance sur l'organisation de code.

– Debugging

* Un bug arrive quand le programme fonctionne, mais pas comme prévu.
* Il s'agit d'une des parties majeures du travail de programmeur et de QA.
* Pour les résoudre, il est nécéssaire d'isoler les parties à travers plusieurs tests, tel un enquêteur.
* Exemples de questions à poser :

  - Est-ce que j'arrive à reproduire le bug à chaque fois avec une procédure spécifique ? (appellée Repro Steps)

  - Est-ce qu'il est possible d'en retirer et tout de même faire que le bug se produit ? (isolement)

  - Est-ce que chaque sous-étape fait son travail correctement ? (examination)

– Debugging : Extraction d'information

* Afin de réussir ensuite, il est souvent nécéssaire d'extraire des informations.
* Une méthode : des logs. Ecrire du texte dans la console ("printf debugging") ou dans un fichier permet de retracer l'exécution.
  - C'est une méthode post-mortem, où tu analyses après que le bug se soit produit, ce qui les rend utiles même si on ne cherchait pas le bug.
* Une autre : utiliser un débugger. On s'attache au programme lors de son exécution, et on peut avancer petit a petit dans l'exécution du programme.

Debugging : Breakpoints

Prototype

File Search Edit Go To Debug

Online Docs Search Help

Character.gd
Projectile.gd
Walker.gd
WalkerSpawner.gd
Spatial

Character.gd

Damage
_physics_process

```
1  extends KinematicBody
2
3  var inputPrefix = "p1_"
4  onready var crosshair = $"../Crosshair"
5
6  onready var prefabShot = load("res://classes/Projectile.tscn")
7
8  var mainCooldown = 0
9
10 var allied = true
11
12 func Damage(data):
15 func _physics_process(delta):
16     var inputHor = Input.get_action_strength("p1_right") - Input.get_action_strength("p1_left")
17     var inputVer = Input.get_action_strength("p1_up") - Input.get_action_strength("p1_down")
18
19     var inputShoot = Input.is_action_pressed("p1_x")
20
21     var moveHorSpeed = 5.0
22     var moveCHorSpeed = 18.0
23     var moveCVerSpeed = 18.0
24     var rateOfFire = 0.2
25
26     if(inputShoot):
27         moveHorSpeed = 0.0
28
29     var move = Vector3.RIGHT * inputHor * delta * moveHorSpeed
30     var crosshairMove = Vector3.RIGHT * inputHor * delta * moveCHorSpeed + Vector3.UP * inputVer * delta * moveCVerSpeed
31
32     move_and_collide(move)
33
34     var crosshairPos = crosshair.get_translation()
35     crosshairPos += crosshairMove
36     var CROSSHAIR_MAX_X = 16.0
37     var CROSSHAIR_MAX_Y = 12.0
38     var CROSSHAIR_MIN_Y = 0.0
39     if(abs(crosshairPos.x) > CROSSHAIR_MAX_X):
40         crosshairPos.x = sign(crosshairPos.x) * CROSSHAIR_MAX_X
41     if(crosshairPos.y < CROSSHAIR_MIN_Y):
42         crosshairPos.y = CROSSHAIR_MIN_Y
43     if(crosshairPos.y > CROSSHAIR_MAX_Y):
44         crosshairPos.y = CROSSHAIR_MAX_Y
```

Breakpoint

Step Over

Step Into

Debugger | Errors (3) | Profiler | Network Profiler | Monitors | Video RAM | Misc

Breakpoint

Stack Frames

0 - res://classes/Character.gd:19 - at function: _physics_process

Locals	
delta	0.017
inputHor	0
inputVer	0
inputShoot	0
Members	
self	Object ID: 1306

Output | Debugger (3) | Search Results | Audio | Animation | 3.3.2.stable

Boucle Principale

Boucle Principale

* Un jeu est basé autour d'une boucle principale qui tourne en permanence.
* Dans les moteurs modernes, on peut compter deux boucles : une graphique, et une physique.
* Les différentes entités (personnages, objets, projectiles...) se mettent à jour à chaque tour de la boucle.

– Méthodes d'entrée

* Il est nécéssaire de vérifier à chaque tour de boucle l'état des entrées : il s'agit de la méthode de polling.
  - C'est opposé à la programmation dite événementielle, où les actions sont mises en oeuvres par des événements discrets comme un appui de bouton.
* Les moteurs offrent des interfaces pour remplacer ça par des "actions", qui offrent un degré d'indirection et ainsi de flexibilité (plusieurs boutons, remapping).

Méthodes d'entrée : Exemple

Analogique

```
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 inputSaut = Input.is_action_just_pressed("p1_a")
```

Bouton

– Note Supplémentaire : Buffers

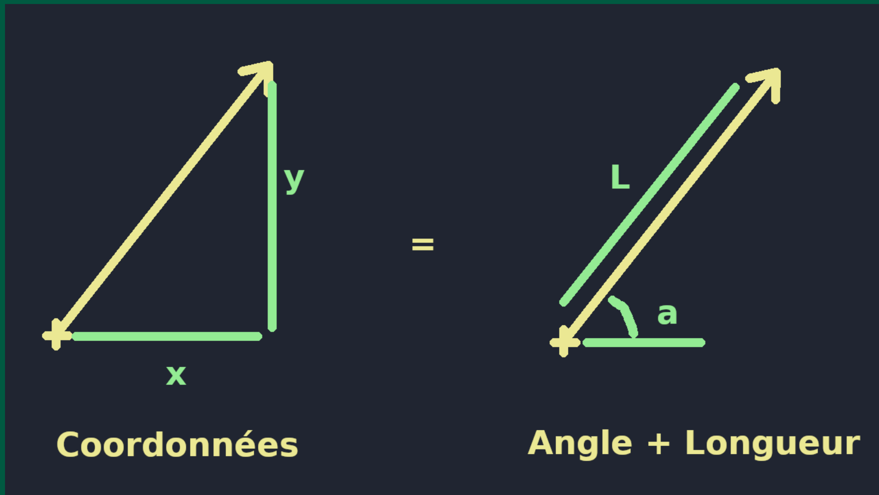
* Note : Il s'agit d'une partie assez sensible pour le contrôle, et des méthodes existent pour rendre le jeu beaucoup plus naturel.
* Les buffers permettent l'appui d'un bouton à l'avance et évitent que les entrées soient "mangées" car non disponible.
* Par exemple : atterir et resauter. Un buffer permet d'appuyer sur le bouton un peu avant l'atterissage, et fait resauter dès le contact avec le sol. Ca permet d'avoir un timing approximatif et ainsi mieux contrôler le personnage.
* En note de plus, on peut réagir à l'appui d'un bouton, ou au moment où on relache ("negative edge"). Faire ainsi permet de mieux gérer les actions multiple (par exemple maintenir vs appui court), mais ajoute de la latence.

Physique

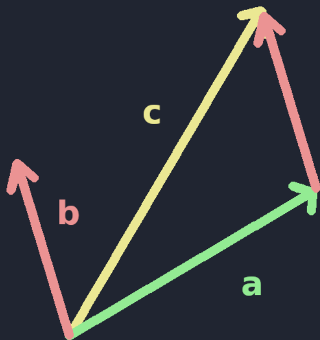
– Rappels : Algèbre linéaire

* On aura besoin d'un outil mathématique très pratique : les vecteurs.
* Je vais parler des vecteurs en 2D, mais les concepts et calculs sont les mêmes en 3D.
* Il y a d'autres concepts d'algèbre utiles mais plus avancés, qui ne sont pas nécéssaires à un niveau basique (produit scalaire/vectoriel, matrices de transformation...).

Vecteurs (1/3)

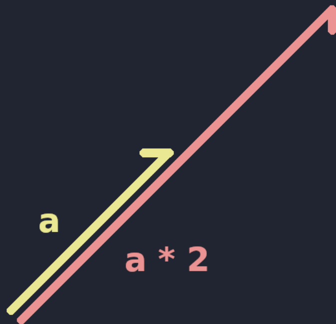


Vecteurs (2/3)



Addition

c = a + b

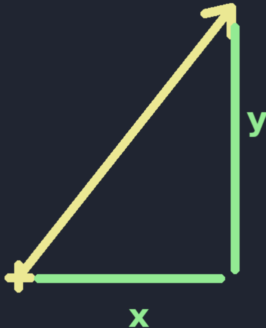


Multiplication*

*par un nombre

Multiplier par un vecteur est possible à travers le produit scalaire ou vectoriel

Vecteurs (3/3)



Longeur / Norme :

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

Normaliser = diviser par la norme
= mettre la longeur à 1

– Rappels : Cinétique

* La cinétique évalue l'évolution de la position avec des forces externes. Certains des outils nous sont utiles.
* La vitesse est la variation de la position, tandis que l'accélération est la variation de la vitesse.
* L'analyse dimentionelle consiste à faire une équation en ne prenant en compte que les unités des variables. C'est utile pour vérifier / débugger les formules.
  - Position : m ; Vitesse : m/s ; Accélération : m/s2
  - Ca veut dire qu'on peut passer de vitesse à position en multipliant par une valeur de temps.

– Mouvement simple

* On peut commencer par faire un déplacement simple constant.
* On veut trouver le vecteur de déplacement de cette frame (tour de boucle).
* Ceci est possible en multipliant la direction du déplacement, la vitesse, et le delta time.

  - La direction (vecteur normalisé = longueur 1) permet de diriger facilement le personnage.

  - La vitesse (en m/s ou pixels/s) est une variable qu'on a renseignés nous mêmes.

  - Le delta time (différence de temps) est le laps de temps depuis le dernier tour de boucle. Ceci permet d'avoir un résultat identique malgré les différentes vitesses d'exécution.

– Mouvement simple : Exemple

```
var direction = Vector2(1.0, 0.0)
var vitesse = 20.0

var mouvement = direction * vitesse * delta
move_and_collide(mouvement)

# Alternative :
move_and_slide(direction * vitesse)
```

En m/s, gère la collision automatiquement

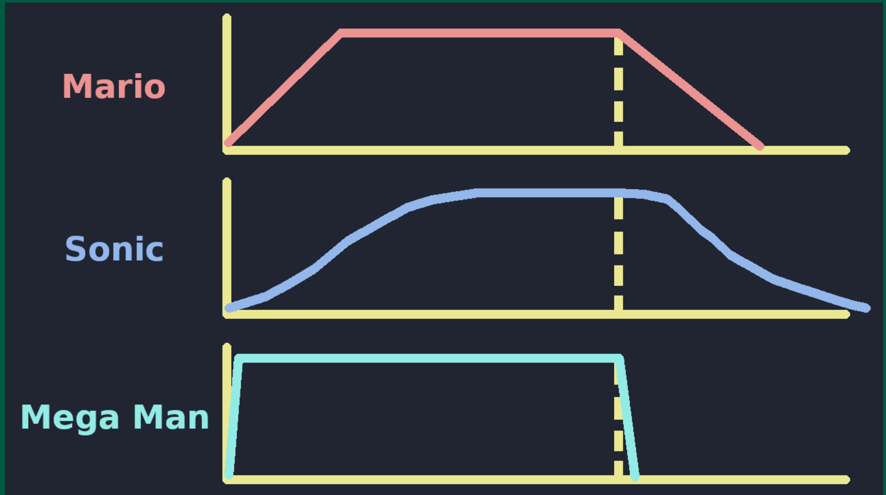
Mouvement avec accélération

- Le mouvement est encore assez artificiel, on va ajouter l'accélération pour obtenir un résultat plus plausible.
- On va modifier la vitesse utilisée pour la formule précédente en ajoutant une force d'accélération et de frein.
- La formule ici est d'ajouter l'accélération ou le frein, multipliés par le delta time. La subtilité ici est que le frein est négatif.
- Nous allons aussi rajouter des limites inférieures et supérieures ("clamp") afin d'avoir une vitesse maximum.

– Mouvement avec accélération : Exemple

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

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

– Différences de contrôle selon l'algorithme : Exemple



– Rigidbody vs Kinematics

* Le comportement ici est de type dit "Kinematic", où on contrôle l'entité directement.
* Un comportement physique ("Rigidbody") va en revanche fonctionner depuis le moteur physique, et va se contrôler à travers des forces qu'on ajoute en Newtons (1N est la force requise pour donner une accélération de 1m/s à une masse de 1kg).
* La méthode simple ici consiste à prendre l'accélération qu'on veut donner cette frame-ci, et multiplier par le poids de l'objet.
* Je conseille d'utiliser de la Kinematic pour le genre de jeux arcade que l'on fait.

Entités

– Programmation Object

* La programmation objet est un paradygme présent dans beaucoup de languages modernes, permettant une réutilisation simple de code.
* Le concept est d'organiser les fonctions et variables dans des "classes", à partir desquelles nous pouvons créer des "objets", qui auront leur propres variables associées.
* Exemple : Une maison. La classe correspond au plan d'architecte, comment construire la maison. A partir de ce plan, il est possible de créer plusieurs maisons, qui peuvent à partir de là diverger sur leurs variables (par exemple, repeindre une maison en bleu au lieu de rouge).
* On verra ça plus en détail dans la partie organisation de code.

– Concepts d'entités dans les jeux

* Les jeux ont tendance à utiliser un système d'entités, qui va plus loin.
* Les entités sont composées de un ou plusieurs composants, qui ont différents rôles (graphismes, physique, comportement, audio, etc...).
* On peut séparer ici le concept de "prefab" (classe, modèle) des "instances" (objets).
* On peut désormais populer nos niveaux avec ces entités.

Entités

- Projectiles, Collisions
- Interfaces d'objets, Gestion Globale
- Machines à état

– Instanciation

* Nous avons un fusil et souhaitons tirer. Afin de ce faire, nous allons créer une balle.
* Il nous faut donc instancier une prefab, c'est à dire charger l'entité depuis le disque, la copier, la paramétrer, et l'ajouter dans le monde.

– Instanciation : Exemple

```
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)
```

Charger

Créer

Paramétrer

Ajouter dans le monde

Colliders

- Une collision arrive quand deux colliders se touchent.
- Un collider est l'ensemble des formes, souvent constuit à partir de primitives (cube, sphères, capsules, etc) ou depuis un mesh (souvent pour les objets statiques tels que le terrain).
- L'algorithme de détection de collisions peut être complexe, et relève plus du moteur.

Types d'entités physiques

* KinematicBody : Objet qui suit la cinématique.
* RigidBody : Objet qui suit un comportement physique réaliste.
* StaticBody : Objet qui ne bouge pas (plus optimisé).
* Area : Objet intangible qui peut détecter ce qui entre à l'intérieur.

– Lancer de Rayon (Raycast) vs Zones

* Une façon alternative de détecter les collisions est d'envoyer un rayon, un peu comme un laser.
* Ce rayon a une portée qui peut être limitée, et renvoie le premier objet trouvé.
* C'est plutôt adapté pour des armes instatanées ("hitscan", comme un pistolet) plutôt que des missiles ("projectiles", comme un bazooka).

Entités

* Projectiles, Collisions
* Interfaces d'objets, Gestion Globale
* Machines à état

– Interfaces

* Le concept d'interface en prog objet est les fonctions et variables d'un objet prévues pour être utilisées par d'autres objets ("fonctions publiques"). (Simplifié)
* Godot utilise le duck-typing, qui permet d'utiliser ces fonctions d'interface directement. (Simplifié)
* Par conséquent, du moment qu'on a un lien vers l'objet (ici obtenu via la détection de collision), on peut appeller une fonction présente sur l'objet (ici une fonction de dommages).

Dégats via interface

```
31 func Damage(data):
32     hp -= data["Damage"]
33     if(hp <= 0):
34         queue_free()
35
```

Ennemi

```
21
22 func Collide(other):
23     if(other.has_method("Damage") and allied != other.allied):
24         other.Damage(GetDamageData())
25         queue_free()
26     else:
27         pass
28
29 func _on_Projectile_area_entered(area):
30     Collide(area)
31 func _on_Projectile_body_entered(body):
32     Collide(body)
33
```

Vérifie si c'est un ennemi

Fait des dégats

Détruit le projectile

Signals

– Références globales : singletons

* Une autre façon d'obtenir des références et données est via un objet global.
* Le concept se nomme Singleton en prog objet, et est disponible sous la forme d'Autoload sur Godot.

– IA d'un ennemi

* Les IA dans ce genre de jeux vont être assez simples. On va créer un ennemi facile :
* Il commence par marcher sur quelques mètres
* Ensuite il tire en boucle sur le joueur (récupéré via le singleton).

IA du Walker

```
func _physics_process(delta):
	if(remainingMove > 0):
		var moveDist = min(speed * delta, remainingMove)
		translate(Vector3.RIGHT * moveDist)
		remainingMove -= moveDist
	else:
		timeUntilNextShot -= delta
		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)
```

Déplacement

Tir

Entités

* Projectiles, Collisions
* Interfaces d'objets, Gestion Globale
* Machines à état

– Qu'est ce qu'une machine à états ?

* Une machine a états est un outil qui consiste à avoir des états, et des transitions entre ces états.
* Un exemple : on peut avoir des états "au sol", "marche", "accroupi", "pris des dégats", etc...
* Réaliser un personnage ou autre peut rapidement devenir labyrinthique sans cet outil, à devoir maintenir beaucoup de variables d'état, certaines non compatibles entre elles...
* Ce point sera développé plus par la suite lors de la partie sur l'organisation de code.

Machine à états : Implémentation

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



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

Mettre en pratique

Proto: Personnage

```
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: Arme

```
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)

```

```
extends Node2D

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


func Gestion(perso, delta):
	pass

```

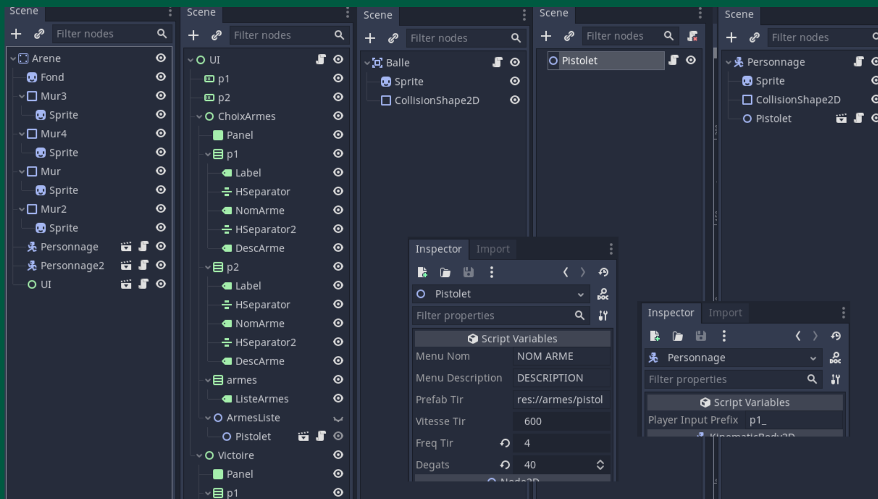
Proto: Tir

```
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

```
1  extends Control
2
3  var modeEntrainement = true
4  enum ETAT_JEU {
5      CHOIX_PERSO, JEU, ECRAN_FIN
6  }
7
8  onready var p1 = $"../Personnage"
9  onready var p2 = $"../Personnage2"
10
11 var _etat = ETAT_JEU.JEU
12
13 func _ready():
14     ChangerEtat(ETAT_JEU.JEU)
15
16 func _process(delta):
17     if(_etat == ETAT_JEU.JEU):
18         GameProcess(delta)
19     elif(_etat == ETAT_JEU.CHOIX_PERSO):
20         MenuChoix(delta)
21     elif(_etat == ETAT_JEU.ECRAN_FIN):
22         EcranVictoire(delta)
23
24 func GameProcess(delta):
25     UpdatePlayer(p1, $p1)
26     UpdatePlayer(p2, $p2)
27
28 func UpdatePlayer(perso, ui):
29     ui.set_max(perso.PV_MAX)
30     ui.set_value(perso._pv)
31
32 func ChangerEtat(etat):
33     _etat = etat
34     $ChoixArmes.set_visible(etat == ETAT_JEU.CHOIX_PERSO)
35     $Victoire.set_visible(etat == ETAT_JEU.ECRAN_FIN)
36     _tempsVictoire = 2.0
37
38
39 func MenuChoix(delta):
40     pass
41
42 var _tempsVictoire
43 func EcranVictoire(delta):
44     if(_tempsVictoire <= 0):
45         ChangerEtat(ETAT_JEU.CHOIX_PERSO)
46     _tempsVictoire -= delta
47
```

– Proto: Scene



Proto: Résultat

Questions ?

- Lien Discord : https://discord.gg/CWjWfC9K9T
- Site du cours : https://panthavma.com/gamedevclass/fr/
- La prochaine fois : Art 2D et 3D statique.
- A côté : Essayer de coder un prototype, sans pression. N'hésitez pas à poser des questions sur Discord !