Introduction à Godot

18/10/2023
Écrit pour Godot 3.5.1
(destiné à des gens qui ont déjà les bases de la programmation)

Présentation générale:

Godot est un moteur de jeu 2D et 3D, gratuit et open-source, aucun coût n'y est associé dans le développement de jeux commerciaux. Le projet existe seulement grâce au travail des bénévoles, aux donations des utilisateurs, aux sponsors du projet et aux bourses qu'il a gagnées.

Langages de programmation supportés:

- GDScript: langage "principal" de Godot. Facile d'utilisation mais très pratique! Langage exclusif à Godot et conçu pour le développement de jeux vidéo. Éditeur inclus au moteur.
- C#: plus rapide que GDScript, mais moins pratique. Besoin d'un éditeur externe.
- C, C++: également supportés via une extension, mêmes remarques que C#. .
- VisualScript: introduit dans Godot 3 et retiré dans Godot 4. Assez peu utilisé par la communauté (déconseillé).

Il existe d'autres langages utilisables via des extensions, mais les 5 mentionnés sont ceux officiellement supportés par les développeurs. Différents langages peuvent être utilisés dans un même projet. Il est conseillé d'utiliser GDScript pour la programmation générale, et
Godot a également un langage de programmation spécifique à ses shaders, et des éléments de visual scripting pour ses matériaux.

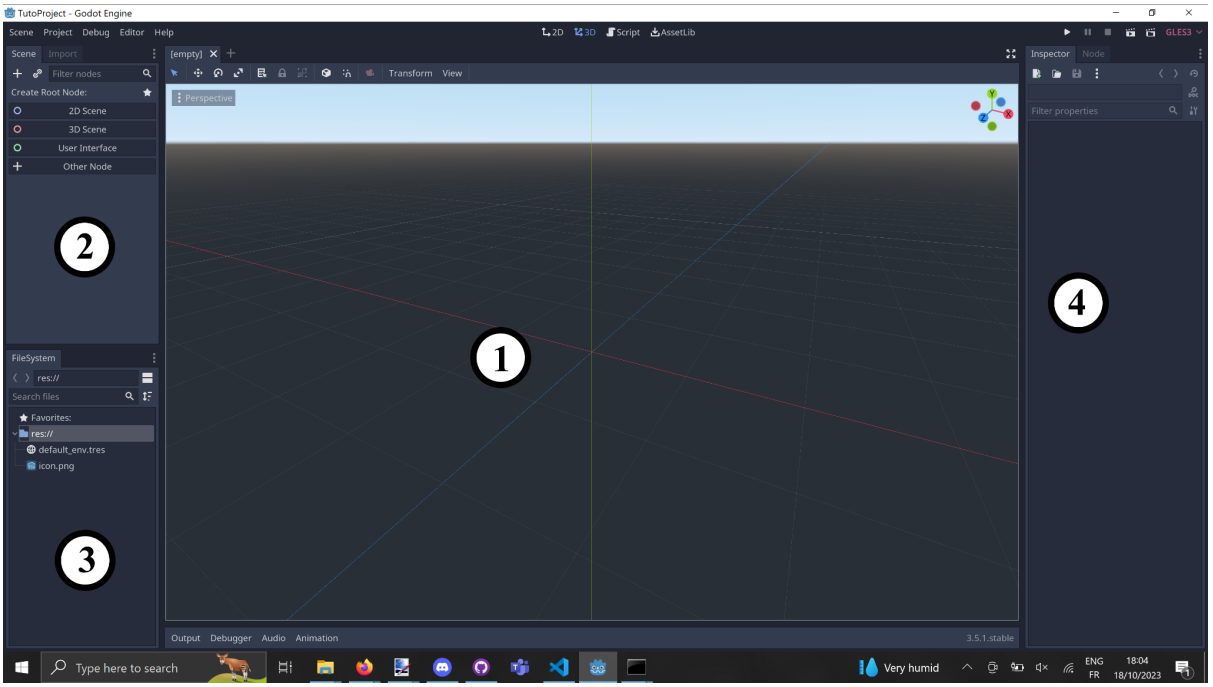
Ressources pratiques:

- docs.godotengine.org: documentation officielle de Godot. Très pratique, contient plusieurs tutoriels intéressants.
- godotshaders.com: catalogue de shaders pour Godot. Voir aussi shadertoy.com pour un site plus général.
- itch.io: beaucoup d'assets gratuits sont disponibles sur le site, toujours pratique pour les game jams et prototypes. Pensez à créditer les ressources que vous utilisez. Voir aussi kenney.nl, craftpix.net, opengameart.org etc.

Indie game dev free open source stuff starter pack:

- Godot: moteur de jeu, éditeur de code
- Github Desktop: très bonne interface pour faciliter l'utilisation de GitHub
- Visual Studio Code: meilleur éditeur de code externe (pas de débat)
- Blender: modélisation et animation 3D
- Audacity: édition de fichiers audio, toujours pratique
- Bosca Ceoil: composition musicale pour les nuls (aka indie game devs)
- Gimp / Paint.net: édition d'images vectorielles (Paint.net exclusif à Windows)

Moteur:



Fenêtre d'accueil à l'ouverture d'un projet vide.

1 - **Current scene**: fenêtre de visualisation sur la scène actuelle. On peut passer de la vue en 2D (Node2D et Control) à 3D (Node3D) à Script (code) à AssetLib (librairie de ressources en ligne). La scène actuelle peut être un niveau, un personnage ou un objet du jeu.

2 - **Scene**: arbre de la scène actuelle. Permet d'ajouter ou retirer des objets, de visualiser et changer la hiérarchie entre objets, et enfin de sélectionner l'objet affiché dans les onglets **Inpector** et **Node**. On peut également gérer les Scripts des Nodes depuis cette fenêtre.

2 - **Import**: permet de visualiser et modifier les paramètres d'importation du fichier sélectionné dans **FileSystem**, ou de réimporter un fichier. Les paramètres d'importation dépendent du type de fichier sélectionné (image et audio sont les deux types de fichiers importants).

3 - **FileSystem**: permet de visualiser et utiliser les fichiers importés. Il est important de bien organiser ses fichiers, vous allez passer beaucoup de temps à naviguer dans cette fenêtre!

4 - **Inspector**: permet de modifier les propriétés (propriétés de bases et variables exportées) de la Node sélectionnée dans l'onglet **Scene**.

4 - **Node**: permet d'utiliser les signaux et de modifier les groupes de la Node sélectionnée dans l'onglet **Scene**.

Notes d'utilisation:

- **Documentation:**

Toujours important mais allez consulter la documentation quand vous avez un problème! Elle est très bien détaillée et couvre l'ensemble des éléments du moteur.

En particulier, si vous codez en GDScript, vous pouvez faire Ctrl + Clic Gauche sur un mot-clé du code (nom de classe, fonction, type…) pour accéder à la documentation de cet élément hors ligne, au sein du moteur. C'est une fonctionnalité extrêmement pratique car très rapide à utiliser quand on a un doute sur la syntaxe, le rôle d'une variable, ou si l'on cherche une fonction précise dans une classe. Cette documentation est un aspect vraiment important du moteur, en particulier si votre rôle principal est de programmer.

- **Project/Project Settings/General/Display/Window:**

Permet de gérer la taille de la fenêtre et l'adaptation du jeu à différents formats d'écrans. C'est un élément assez important des projets, je vous conseille de regarder la documentation (assez bien détaillée).

- **Project/Project Settings/Input map:**

C'est dans cette section que l'on peut assigner des actions particulières à des touches, boutons ou joysticks. En particulier, Godot supporte naturellement le passage QWERTY-AZERTY avec la fonctionnalité Physical Key (utilise la position de la touche sur le clavier plutôt que le caractère associé). Sur mobile, le doigt est assimilé à la souris (pas besoin de faire la différence).

- **Project/Export:**

Exporter un projet, android, cross platform, mode debug et mode release

---

Fichiers:

Les différents types fichiers utilisés sous Godot et leurs usages.

A. **Nodes (.tscn):**
   Les Nodes sont les objets qui composent le jeu (ceux qu'on trouve dans les fenêtres **Scene**, **Node** et **Inspector**). Tous les objets héritent de Node, mais on a ensuite plusieurs familles.

- Node2D: éléments de jeu 2D. Cela englobe les sprites, les éléments d'animation, les sons spatialisés en 2D, les éléments liés à la physique et aux collisions, les décors, les lumières ou la navigation des PNJ en 2D.
  Exemples: Sprite, AnimatedSprite, KinematicBody2D, Polygon2D, Particle2D, Collision2D, Tilemap, Skeleton2D, RayCast2D, ParallaxLayer, AudioStreamPlayer2D…

- Spatial (ou Node3D sous Godot 4): éléments de jeu 3D. Mesh, Squelettes, sons spatialisés en 3D, éléments de collision, physique et navigation en 3D…
  Exemples: Camera3D, Raycast, Skeleton, Path, CollisionPolygon, AudioStreamPlayer3D, Joints, Area, RigidBody…

- Control: éléments d'interface. Tout ce qui est lié aux menus, boîtes de dialogue, entrée de texte, éléments de HUD, etc…
  Exemples: Button, TextureRect, ColorRect, Container, Popup, TextEdit, VideoPlayer, Scrollbar, ProgressBar, GraphEdit, ColorPicker…

Certains éléments très utiles héritent de Node directement, comme l'AnimationPlayer, le Tween (nodes d'animation), le HTTPRequest (internet), le WorldEnvironment (qualité visuelle de la scène) ou le CanvasLayer (lié à l'écran du joueur).
Dans une même scène, on peut utiliser des nodes de différentes catégories, mais essayez de garder vos scènes petites et lisibles.
Il est aussi possible de créer ses propres classes de Nodes, avec de nouvelles propriétés, à partir des Node existantes, de ressources et de code.

B. Resources:

a. Ressources Godot (.tres):

Il existe de nombreux types de ressources différents, vous les découvrirez au fur et à mesure, quand vous en aurez besoin pour une Node particulière.
Quelques exemples: Textures, Gradients, Noise, Curves, Environments, Animations, Audio Buses, Audio Bus Effects, Bitmap Font, Dynamic Font, Shapes, 2D Shapes, Materials, Shaders, ShaderMaterials, Polygons, RichTextEffects, StyleBoxes, Tilesets…

b. Ressources externes:

Tous les assets créés à l'extérieur du moteur.

- Images: PNG, JPEG, SVG (mais transformés en images matricielles au moment de l'importation), BPM, EXR…
- Audio: WAVE (lourds en mémoire mais légers sur le processeur, bien pour les sfx), OGG (très bonne compression, lourd à exécuter, bien pour les longs morceaux), MP3 (entre les deux)
- Scènes 3D: glTF (recommandé), BLEND, DAE, OBJ, FBX. Les animations sont créées automatiquement avec un AnimationPlayer.
- Polices de caractères: TTF, TTC, OTF, OTC… A partir d'un fichier police de caractère, on peut créer une ressource DynamicFont, utilisable sous Godot.

Built-in Classes:

Les classes présentées ici sont les classes héritant de Node2D et Control (interface), ainsi que certaines classes utiles héritant directement de Node. Pour les classes héritant de Spatial (Node3D pour Godot 4.0) regardez la documentation, mais beaucoup sont des équivalents des Node2D.

Node2D:

- **Sprite:** affiche une image à l'écran
- **AnimatedSprite:** affiche une succession d'images, plusieurs animations possibles
- **CollisionObject2D:** objet gérant les collisions et interactions entre objets 2D
    - **Area2D:** zone sans physique, permet de détecter les corps entrant
    - **StaticBody2D:** corps immobile (ex: obstacle, mur)
    - **RigidBody2D:** corps suivant les règles physiques de l'univers (ex: ballon, projectile)
    - **KinematicBody2D:** corps suivant des règles simples, avec des méthodes pratiques pour le déplacement de personnage (ex: ennemi, joueur, projectile)

- **CollisionPolygon2D/CollisionShape2D:** nécessaire aux CollisionObjects2D pour définir la forme du collider
- **AudioStreamPlayer2D:** son spatialisé en 2D (ex: effet sonore de porte qui s'ouvre)
- **Particle2D:** effet de particules (ex: fumée d'une fusée, effet magique)
- **Camera2D:** caméra dans un monde 2D (ex: vision du joueur, peu d'autres utilisations)
- **Light2D/LightOccluder2D/CanvasModulate:**
- **Line2D:** ligne dessinée à l'écran suivant plusieurs points, pratique pour le prototypage et plein de choses
- **Polygon2D:** polygone visuel affiché à l'écran (pratique pour le prototypage et les obstacles ayant une forme variable, cf décors de Rayman legends)
- **Path2D:** permet de définir un chemin pour des personnages avec pathfinding, également pratique pour améliorer les formes créées avec Polygon2D et Line2D avec la fonction get_baked_points() cf ce tutoriel
- **PathFollow2D:** peuvent suivre un chemin 2D en évitant les obstacles (algorithme A*)
- **RayCast2D:** permet de détecter s'il y a un obstacle entre deux points (ex: vision d'un PNJ)
- **TileMap:** tilemap, pratique pour créer des décors ou maps type RPG, possibilité d'y ajouter des éléments de collision, des interactions avec la lumière, etc…

Control:

Note: Godot est fait de ses propres Nodes, en particulier les Nodes héritant de Control.

- **Container:** élément d'interface appliquant certaines règles de position à ses enfants, très pratique pour positionner peu importe le format de l'écran de l'utilisateur
  - **HBoxContainer, VBoxContainer:** ordonne ses enfants horizontalement ou verticalement (ex: menu avec plusieurs options à sélectionner)
  - **ColorPicker:** permet de choisir une couleur à la souris
- **Button/TextureButton:** élément d'interface pouvant être cliqué (ex: sélection de niveau)
- **ColorRect**: rectangle d'une couleur unie (ex: arrière-plan)
- **TextureRect:** affiche une texture (ex: point d'exclamation au-dessus d'un PNJ)
- **Range:**
  - **ProgressBar/TextureProgress**: progress bar (ex: barre de mana)
  - **HSlider, VSlider**: slider à bouger (ex: changement du volume dans un menu)
  - **SpinBox:** permet d'entrer un nombre, à la souris (flèches haut-bas) ou au clavier
- **LineEdit:** ligne d'édition de texte (ex: identifiant, mot de passe…)
- **Panel:** cadre pouvant s'agrandir sans augmenter la taille des coins (ex: arrière-plan de menu)
- **Label:** zone dans laquelle on peut afficher du texte (ex: vie restante dans un HUD)
- **RichTextLabel:** label auquel on peut ajouter de jolis effets (ex: texte de boîte de dialogue)
- **TextEdit:** zone d'édition de texte multi-ligne (ex: lettre à écrire, édition de code…)

Node:

- **AudioStreamPlayer:** joue un fichier sonore
- **CanvasLayer:** positionne les nodes Control sur l'écran plutôt que dans le monde
- **HTTPRequest:** permet d'interagir avec l'INTERNET 💥
- **Timer:** timer. pratique et léger.
- **WorldEnvironment:** permet d'ajouter de jolis effets à la scène (glow, color adjustments, fog, tonemap, auto exposure, ambient light, depth of field…)
- **AnimationPlayer:** permet d'animer n'importe quelle variable de n'importe quelle Node (pratique pour définir des animations précises et joulies)
- **Tween:** permet (entre autres) d'interpoler des valeurs procéduralement, très pratique pour faire de l'animation variable, où l'on ne connaît pas les valeurs de début et de fin

---

GDScript:

La syntaxe est inspirée du python, mais GDScript est un langage conçu spécialement pour Godot. C'est un langage à indentations, sans points-virgules ni accolades.
Pour ce qui est de son exécution, le GDScript peut être compilé just-in-time (fichiers .gd) ou ahead-of-time (fichiers .gdc, générés quand le jeu est exporté). On a donc un gain de performance quand on exporte le jeu (d'autant plus quand on ignore les commandes debug).

Mots-clés:
A écrire en tête de fichier, au-dessus du corps du code.`var`

- `tool` : exécute le script dans l'éditeur. Pratique pour visualiser des changements d'objets dans l'éditeur (ex: mettre à jour le polygon d'un CollisionPolygon2D en mettant à jour le Polygon2D correspondant).
- `class_name` : donne un nom à la classe, et l'enregistre dans les classes par défaut (ajoutables depuis le bouton + de l'onglet **Scene**). Pratique pour les références à cette classe depuis une autre scène, permet de regarder les fonctions facilement et d'éviter certaines erreurs. Attention, les cross-references ne sont pas supportées par Godot 3.X.
- `extends` : Héritage. Il est possible d'extend une classe par défaut, une classe créée par class_name, ou un autre script (dans ce cas il faut noter l'adresse du script dans **FileSystem**). Pas d'héritage de deux classes différentes.

Variables, constants, signaux:

- **Définition de constantes:**

Plus rapides que les variables dans un code exporté. Mieux pour la lisibilité que d'écrire en dur des valeurs dans le code.
Exemple de déclaration de constante:

```
const GRAVITY = 9.8
```

- **Enums:**

Ensemble de valeurs ayant un nom. Se déclare avec `enum NomDeLenum {nom_1, nom_2, nom_3, nom_4}`. Il est optionnel de nommer l'enum.
Exemple d'utilisation d'enums:

```
enum {Good, Bad}
enum PlayerStates {Idle, Walking, Jumping}
var current_movement_state := PlayerState.Idle
var is_good = Bad

func _process(_delta):
    match current_movement_state:
        PlayerState.Idle:
```

```
            print("Player idle")
        PlayerState.Walking:
            print("Player walking")
        PlayerState.Jumping:
            print("Player jumping")
    if is_good == Good:
        print("tis good")
```
Très pratique pour la lisibilité.

- **Définition de variables:**

Fait avec le mot-clé var.
Exemples:
```
var a                   # variable non typée
var b = 0               # variable non typée à valeur 0
var c : float = 0.0     # variable de type float à valeur 0
var d := "0"            # valeur de type String à valeur "0"
```
Dans Godot, les variables n'ont pas nécessairement de type, mais c'est une bonne pratique de leur en assigner un, soit explicitement avec var a : type, soit implicitement avec var a := valeur.
Une variable typée ne peut pas changer de type, et l'éditeur affiche une erreur quand l'on essaie.

- **Exporter des variables:**

On peut exporter une variable en la déclarant avec le mot-clé export: export var a : int = 1.
Cela permet d'éditer la variable depuis la fenêtre **Inspector**. Très pratique pour le level design.
`export var a : int = 1`
On peut ajouter des paramètres au mot-clé export, si l'on veut limiter les valeurs prises par la variable. Par exemple, `export(Enemy) var enemy_state = Enemy.Idle` force la sélection des valeurs depuis l'**Inspector** aux valeurs de l'enum donné.
Le mot-clé `export(NodePath)` var a réduit les valeurs de la variable aux noms des Nodes présentes dans la scène. Ex:
```
export(NodePath) var player_camera_path
onready var player_camera : Camera2D = get_node(player_camera_path)
```
Pratique pour lier des Nodes entre eux (ex: une clé avec une porte).

- **Signaux:**

Les signaux permettent de détecter quand certaines conditions sont remplies. Par exemple, quand un objet entre dans une Area, quand des RigidBody entrent en collision, quand une animation est complétée, ou encore quand un objet entre dans l'écran du joueur.
Il est possible de créer des signaux customisés avec `signal nomDuSignal`, que l'on peut ensuite appeler dans le code avec la fonction `emit_signal("nomDuSignal")`. Ce sont le plus souvent d'autres Nodes qui utilisent le signal d'une Node.
Exemple:
```
signal IAmDyingHelp
var health := 100.0
```

```
func _physic_process(_delta):
    if health <= 0.0:
        emit_signal("iAmDyingHelp")
```

Types:
Les variables peuvent avoir de nombreux types, soit des types par défaut, soit des classes de Nodes (y compris les classes créées avec class_name).

Types par défaut (non exhaustif):
- `bool` : true ou false
- `int` : nombres entiers, stockés sur 64 bits
- `float` : nombres à virgule, stockés sur 64 bits dans les floats et 32 dans les vecteurs (mais il existe une option pour les faire passer en 64 bits)
- `String` : texte.
- `Vector2` : pour les coordonnées/scales/sizes en 2D et UI
- `Vector3` : pour les positions/rotations/scales en 3D
- `Color` : ensemble de 3 floats (r, g, b, ou 4 avec la transparence: r, g, b, a) correspondant à une couleur. Color(1, 1, 1) == Color.white et Color(0, 0, 0) == Color.black
- `Array` : ensemble d'éléments dans un tableau ordonné. Chaque case du tableau peut contenir une valeur de n'importe quel type. Certains types d'arrays (`PoolIntArray` , `PoolFloattArray` , `PoolStringArray`) ne contiennent que des valeurs d'un même type et sont plus optimisés
- `Dictionary` : ensemble de clé:valeur, très utile pour sauvegarder des données en format json par exemple.

Écrire ses fonctions:

On peut écrire ses propres fonctions avec le mot-clé "`func`".
La définition minimale d'une fonction est "`func nom_de_la_fonction():`".
On peut également ajouter un ou plusieurs arguments, éventuellement les typer, leur donner une valeur par défaut (dans ce cas on peut appeler la fonction sans préciser d'arguments), ou encore ajouter une valeur de retour.

Exemples:
```
func take_damage(amount)-> void:
    health -= amount

func reset_location(new_pos : Vector3, player_number : int = 1):
    Players[player_number].position = new_pos

func get_remaining_development_time() -> float:
    return INF
```

Ces fonctions peuvent être appelées avec:

```
take_damage(30)
reset_location(Vector3.ZERO, 3)
reset_location(Vector3(45, 230, 55))
get_remaining_sanity()
var dev_sanity := 1.0 / get_remaining_development_time()
```

Built-in functions and methods:

Méthodes et syntaxe:

- Gestion des Nodes:

```
get_node(player_camera)  #node au chemin player_camera
$PlayerCamera            #node locale au chemin “PlayerCamera”
```

- Flow control:

```
for i in range(0, 10):
    print("boucle répétée 10 fois")

var i = 0
while i < 10:
    i += 1
    print("boucle répétée 10 fois")

while true:
    print("boucle infinie")

match variable_donnee:
    val1:
        print("effet 1")
    val2:
        print("effet 2")
    _:
        print("effet par défaut")
```

Fonctions appelées automatiquement:

- `_ready()`:
- `_process(delta)`: appelé à chaque frame. Delta est le temps écoulé depuis le dernier appel.
- `_physic_process(delta)`: appelé à chaque pas de calcul des physiques du jeu.
- `_input(event)`: appelé à chaque pas de temps ou l'utilisateur interagit avec le jeu (clavier, souris, manette…). La nature de l'événement d'interaction est stockée dans la variable event, avec différentes propriétés selon le type d'événement.

Note: Si une fonction par défaut a un argument que vous n'utilisez pas, vous pouvez mettre un "_" avant le nom de l'argument pour ne pas avoir de warning.

Fonctions à appeler:

- Gestion du temps:

```
yield(get_tree().create_timer(1.5), “timeout”) #délai de 1.5 secondes
# avant exécution de la ligne suivante
yield(n, "event")  #délai jusqu’à émission du signal event de la
# node n avant exécution de la ligne suivante
```