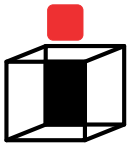


Introduction aux moteurs de jeu

Sylvia Chalençon **Adrien Revault d'Allonnes**

Septembre 2020

Table des matières

Chapitre 1

Introduction

Ce document est le support du cours « Introduction aux moteurs de jeux », il fait partie de la mineure « Conception et programmation de jeux vidéo » et est enseigné au premier semestre de première année de la licence Informatique & Vidéoludisme. L'objectif de ce cours est de présenter le fonctionnement des moteurs de jeu (leur rôle, leurs fonctionnalités), ainsi que la prise en main concrète d'un moteur de jeu en particulier (*Godot*).

Chapitre 2

Un moteur de jeu ?

2.1 Définition

La notion de moteur de jeu est née dans les années 90s, introduite par les jeux de tir à la première personne tels que *Doom*. L'architecture logicielle de ces jeux séparait distinctement les composants moteurs du jeu et les ressources numériques propres au jeu telles que les graphismes, les sons, la musique, les scènes et les règles de déroulement. Cette organisation a permis la réutilisation des composants moteurs pour développer de nouveaux jeu avec des nouveaux personnages, des nouvelles armes, des nouvelles cartes..., procurant un gain de temps considérable.

Un moteur de jeu est un ensemble de composants logiciels qui fournit des fonctionnalités nécessaires au développement d'un jeu vidéo parmi lesquelles on peut citer :

— un moteur 2D,
— un moteur 3D,
— un moteur physique,
— un moteur d'animation,
— une gestion des entrées/sorties,
— un moteur d'interface graphique,
— un gestionnaire des ressources,
— un gestionnaire de scènes ...

Le moteur de jeu aspire donc à permettre de réserver les efforts de développement au contenu et au déroulement du jeu plutôt qu'à la résolution de problèmes informatiques déjà traités préalablement. Le concepteur du jeu utilise des scripts, le langage dépendra du moteur de jeux choisi, afin de programmer les comportements des personnages non jouables [1] ou de modifier

1. PNJ : personnage non jouable

le *gameplay* par exemple.

2.2 Quelques fonctionnalités

2.2.1 Moteur 2D

Un moteur 2D sert à afficher à l'écran un environnement graphique bidimensionnel ou isométrique, à partir d'un point de vue spécifié. Il existe deux types de moteurs 2D : l'un affiche une image unique, l'autre une image en « tuile ». Dans le premier cas, un niveau est constitué d'une image unique de grande taile sur laquelle le personnage se déplace. Dans le second cas, un niveau est créé par assemblage d'une multitude de petites images appelées tuiles (*tiles* en anglais), pour former une grande image. L'ensemble des tuiles constitue ce qu'on appelle une palette de tuiles ou *tilemap*.

L'affichage du niveau se fait en vue de côté, en vue de dessus stricte (vue aérienne), en vue de dessus avec perspective (effet de profondeur suivant un seul axe) ou en vue isométrique (perspective dans laquelle les trois dimensions sont représentées avec la même importance).

2.2.2 Moteur 3D

Un moteur 3D est un composant logiciel qui crée des images matricielles à partir de coordonnées tridimensionnelles. Chaque élément du jeu est représenté par une suite de polygones (en général des triangles). Les coordonnées des sommets de ces polygones sont exprimées en trois dimensions. En fonction du type de rendu effectué, le moteur va donc permettre d'obtenir, à partir de ces coordonnées 3D, une matrice 2D qui correspondra à l'image affichée sur l'écran.

Pour dessiner ses scènes 2D ou 3D, le développeur peut utiliser des bibliothèques de rendu 3D (OpenGL, Direct3D pour citer les plus répandues) mais cela exige de maîtriser les modèles mathématiques sous-jacents, et le pipeline 3D (les opérations généralement réalisées par une carte graphique nécessaires au rendu d'un lot de données). Utiliser un moteur 3D permet d'obtenir le rendu désiré plus rapidement et sans maîtriser ces techniques complexes.

2.2.3 Moteur physique

Le moteur physique calcule la trajectoire des objets, leurs interactions, les forces subies (pesanteur, frottements, ...). Il calcule également la déformation des objets mous comme les étoffes ou les cheveux. Au moteur physique se

superpose en général le détecteur de collisions. Son rôle est de détecter la rencontre de deux objets afin de déterminer l'action à appliquer.

2.2.4 Moteur Son

Le moteur son effectue le mixage des bruits et de la musique tout au long du jeu. La gestion du son est composée d'un logiciel de lecteur audio associé à un logiciel de mixage et un générateur d'effets sonores. Le moteur son modélise un ensemble de sources sonores se déplaçant dans un espace en 3D quand le personnage évolue dans ce même espace. Des modifications sur les sources peuvent être ajoutées, comme de l'écho ou autres effets sonores.

2.2.5 Gestion des entrées/sorties

Cette partie traite les échanges d'informations entre le jeu et les périphériques qui lui sont associés. Classiquement, le moteur de jeu détecte les actions du joueur sur le joystick (état des boutons, action sur les sticks), sur la souris (état des boutons, position, mouvement), sur le clavier (appui de touches), et sur d'autres matériels selon les moteurs (périphériques AR/VR). C'est ici qu'est assurée la lecture des données du jeu, ainsi que les sauvegardes des données utilisateurs.

2.3 Quelques moteurs de jeux

Nous présentons brièvement les trois moteurs plus fréquemment rencontrés dans l'industrie et leurs conditions d'utilisation, avant de nous focaliser sur Godot.

2.3.1 *CryEngine*

CryEngine, développé par Crytek, est plutôt orienté jeux de tir à la première personne ou FPS [2]. Le premier jeu qui l'utilise est *Far Cry*, d'où son nom. Nous en sommes à la version 5 et de nombreux jeux célèbres reposent sur lui (*Crysis*, *Sniper*, ...).

Les langages utilisés pour son développement sont C++, *Lua* et C# et le scripting est disponible dans ces trois langages. Le développement se fait sur *Windows* mais un export multi-plateforme est disponible. La licence prévoit une redevance de 5% sur les bénéfices du jeu utilisant CryEngine au delà des premiers 5 000 $ annuels.

2. First Person Shooter

2.3.2 *Unreal Engine*

Unreal Engine est un moteur de jeu vidéo propriétaire développé par *Epic Games*, dont les principaux concurrents sont *Cry Engine* et *Unity*, également présentés ici. À l'origine, son code est très largement dédié au jeu *Unreal Tournament*. *Unreal Engine* est programmé en C++. La conception objet du moteur et sa modularité ont séduit la commmunauté des concepteurs de jeux vidéos et des jeux célèbres reposent sur *Unreal Engine* (*Fortnite*, *Dauntless*, *Rocket League*, ...).

Le développement est possible sur *Windows*, *Mac OS* et *Linux*, en *C++*, *Python* ou en *Visual scripting*.

La licence est gratuite dans le cas d'une utilisation à but non commercial et prévoit une redevance de 5% sur les bénéfices au delà du premier million de dollars.

2.3.3 *Unity*

Unity est un des moteur de jeu les plus répandus. Développé initialement sous *Mac OS* en C#, il a été porté sous *Windows* puis *Linux* même si tous les exports ne sont pas disponibles (pas d'export *Linux* vers *Windows* par exemple).

Il existe quatre niveaux de licence :

- *personal* : gratuit dans la limite de revenus ou financements inférieurs à 100 000 $ au cours des 12 derniers mois.
- *plus* : 369 € par an et par poste si vos revenus ou financements sont inférieurs à 200 000 $ sur les 12 derniers mois.
- *pro* : 1 656 € par an et par poste si vos revenus ou financements sont supérieurs à 200 000 $ sur les 12 derniers mois.
- *enterprise* : 183 € par mois et par poste au delà de 20 postes et pour des revenus ou financements supérieurs à 200 000 $ sur les 12 derniers mois.

Les scripts sont rédigeables uniquement en C#.

Chapitre 3

Présentation de *Godot*

Godot Engine est un moteur de jeu multiplateforme, riche en fonctionnalités permettant de créer des jeux 2D et 3D à partir d'une interface unifiée. Il fournit un ensemble complet d'outils, afin que les utilisateurs puissent se concentrer sur la création de jeux. Les jeux peuvent être exportés vers un certain nombre de plates-formes, notamment *Linux*, *MacOS* et *Windows* mais aussi vers les plates-formes mobiles (*Android*, *iOS*) et Web (*HTML5*).

Godot est entièrement gratuit et *open source* sous la licence MIT permissive. L'utilisateur est libre de le copier, le modifier, le fusionner, le publier, le distribuer, il suffit d'incorporer la notice de licence et de *copyright*. Le développement de *Godot* est indépendant et axé sur la communauté, permettant aux utilisateurs d'aider à façonner le moteur en fonction de leurs attentes. Il est soutenu par le *Software Freedom Conservancy* à but non lucratif.

3.1 Moteur 2D

Le moteur 2D de *Godot* utilise le pixel comme unité de base, tout en permettant de s'adapter à n'importe quelle taille d'écran ou format d'image. Le moteur 2D offre des outils pour dessiner des lignes, des polygones et des *sprites* (images en deux dimensions partiellement transparente affichée à l'écran par-dessus un ensemble d'autres images.). On dispose d'un outil pour animer ces *sprites* facilement. La modélisation 2D de la source lumineuse associée à une carte des normales permet de générer des ombres, ou de la lumière diffuse. Un effet de profondeur peut être créé en utilisant le défilement parallaxe (différentes vitesses de défilement en fonction de l'éloignement des calques par rapport à l'observateur). Enfin il est possible de génerer en CPU ou GPU des particules et d'utiliser des *shaders* personnalisés pour leur rendu.

3.2 Moteur 3D

Le moteur 3D de *Godot* utilise *OpenGL ES 3.0* sur les plateformes qui le supportent, *OpenGL ES 2.0* sinon. Le moteur prend en charge les techniques de rendu physique réaliste, telles que la spécularité, les ombres dynamiques, l'illumination globale, des effets de post-traitement (*bloom*, *HDR,* ...). Un langage de *shader* simplifié, similaire à *GLSL*, est également incorporé. Des *shaders* peuvent également être créés en *Visual scripting*.

3.3 Moteur physique

Le moteur physique permet d'animer, en simulant des lois de la physique, tout ce qui est disponible dans l'inspecteur, n'importe quel nœud ou ressource, les *sprites*, les éléments d'interface utilisateur, les particules ...

3.4 Graphe de scène

L'architecture de *Godot* s'articule autour du concept de scène. Une scène, est un ensemble de nœuds organisés en arbre. Chaque scène est réutilisable, instanciable et héritable. Il est possible d'imbriquer une scène créée au préalable, en tant que nœud d'une autre scène. Les nœuds peuvent également être des *sprites*, des formes servant de masques de collision, des sources lumineuses, des sources sonores, des objets physiques ...

3.5 Scripts

Il est possible d'attacher un script à chaque nœud du graphe de scène afin d'en modifier le comportement. Plusieurs langages sont disponibles :

- *GDScript* : le langage de script conçu pour *Godot*, sa syntaxe est très proche du *Python* ;
- C++ ;
- C# à travers *Mono* ;
- *VisualScript*, langage graphique propre à *Godot*.

3.6 L'éditeur

L'aventure *Godot* va commencer ! La première étape est de le télécharger ici. Lorsqu'on lance *Godot*, c'est le gestionnaire de projet qui s'ouvre.

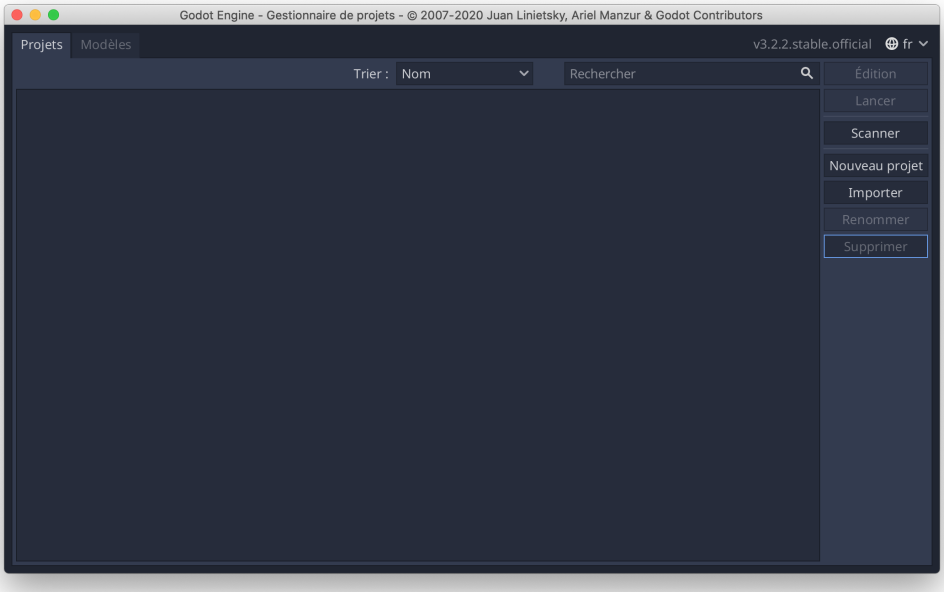


FIGURE 3.1 – Gestionnaire de projets

Ici (cf. fig. 3.1), il est possible d'ouvrir, de créer, d'importer ou de supprimer un projet. Pour importer un projet, il faut aller sélectionner parmi ceux proposés dans l'onglet « Modèles ». Une fois, téléchargé et installé le projet importé peut être ouvert et lancé. Les projets existants sont listés dans l'onglet « Projets ».

Pour créér un nouveau projet, il faut un répertoire vide qui servira de répertoire de travail pour ce projet. C'est au moment de la création du projet qu'a lieu le choix du moteur de rendu à utiliser même s'il sera possible de changer d'avis plus tard et de le modifier dans les paramètres du projet.

Très classiquement, les menus principaux se situent en haut à gauche de la fenêtre de l'éditeur (cf. fig. 3.2)

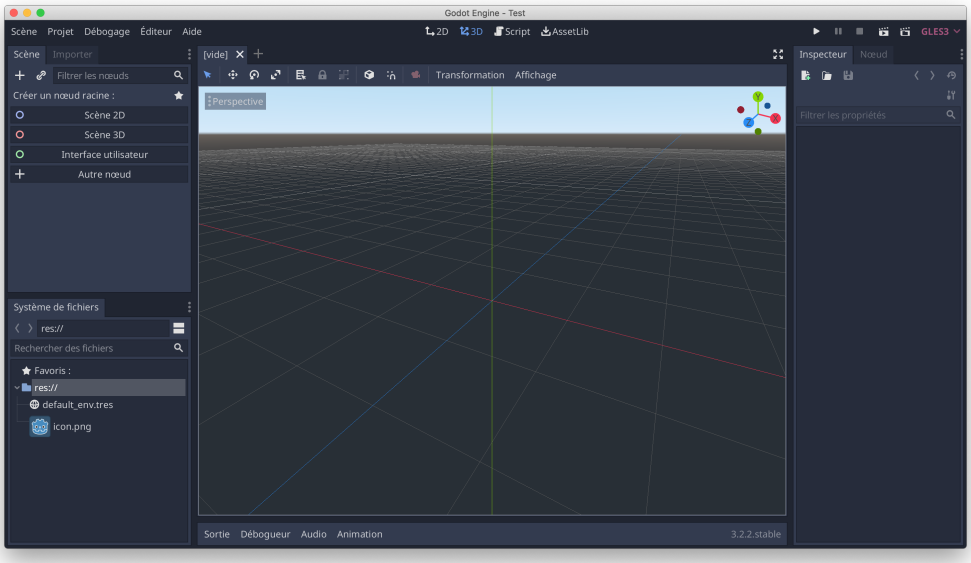


FIGURE 3.2 – Éditeur

En haut, au centre, se trouvent les boutons de choix de l'espace de travail. Ce choix déterminera le contenu de l'espace principal et central. Il y a 4 possibilités :

- l'espace de travail 2D utilisé pour les jeux en 2D, mais aussi pour construire vos interfaces.
- l'espace de travail 3D pour élaborer les jeux 3D, en manipulant les modèles 3D, lumières, caméras ...
- l'espace de travail script est un éditeur de code pour rédiger les scripts sans sortir de *Godot*. On a accès à la documentation dans l'éditeur.
- l'espace de travail *AssetLib* est une bibliothèque d'extensions, de scripts et de ressources gratuits.

Sur la gauche de la fenêtre se trouve le graphe de scène qui répertorie le contenu de la scène active, et en dessous, le gestionnaire de fichiers et ressources du projet. L'inspecteur se trouve sur la partie droite et permet d'ajuster les paramètres de la scène. Le bandeau du bas contient l'affichage de la sortie standard du programme, la console de débogage, l'éditeur d'animations, le mixeur audio. Tout se tient donc dans une fenêtre unique et il est possible de redimensionner et de déplacer chaque élément.

Chapitre 4

Composition de scènes

4.1 Les nœuds

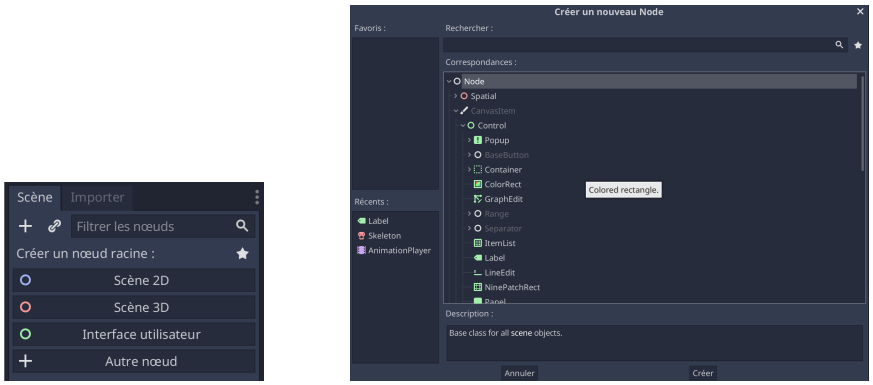
Le nœud est l'élément fondamental de toute scène et donc de tout projet *Godot*. Il existent de nombreux types de nœuds dédiés à l'affichage, au son, à l'animation .... Un nœud à sa création, quelque soit son type, a un nom et des propriétés éditables. On peut lui attacher une fonction de *callback* qui sera appelée à chaque *frame*. Mais surtout il peut être ajouté comme « fils » d'un autre nœud et réciproquement un nœud peut avoir plusieurs nœuds fils. C'est cette propriété qui conduit à l'organisation en arbre des nœuds pour former une scène. Une scène est donc un groupe de nœuds avec un nœud racine unique. Une scène peut être instanciée dans une autre scène. Un jeu est constitué de plusieurs scènes mais une seule d'entre elles doit être identifiée comme scène principale afin d'être lancée au démarrage du jeu.

4.2 Création d'une scène : *'Hello world !'*

4.2.1 Création de la scène

Comme première scène, nous avons choisi d'afficher un message de bienvenue. Dans un projet vide, le graphe de scène est vide et propose la création d'un nœud racine (cf. fig. 4.1a).

Pour le créer, on utilise sans distinction l'une ou l'autre des croix, celle située en haut à gauche qui permet d'ajouter un nœud enfant ou la croix « autre nœud » dans la liste des type de nœuds racines (cf. fig. 4.1b). Il est à noter que par la suite, lorsque la scène n'est pas vide, il faut utiliser le bouton « ajouter un nœud enfant ». Nous allons d'abord créér un nœud *Scène 2D*, puis dans notre cas, nous devons lui ajouter un nœud fils de type *Label*.



(a) Graphe de scène (b) Choix du type de nœud

FIGURE 4.1 – Évolution du graphe de scène

À la création du premier nœud, l'éditeur passe en 2D. À l'ajout du *Label*, son rectangle est affiché dans le coin supérieur gauche de l'espace de travail 2D, le nœud est ajouté au graphe de scène et ses propriétés sont accessibles dans l'inspecteur. C'est le moment de modifier la propriété *Text* et de lui donner la valeur « Bienvenue en L1 ! ». Les propriétés *Align* et *Valign* permettent de positionner le texte dans le rectangle (cf. fig. 4.2).

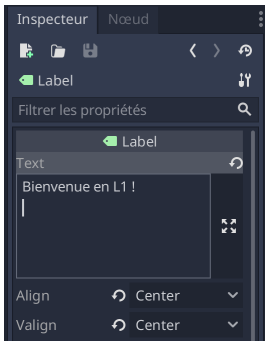


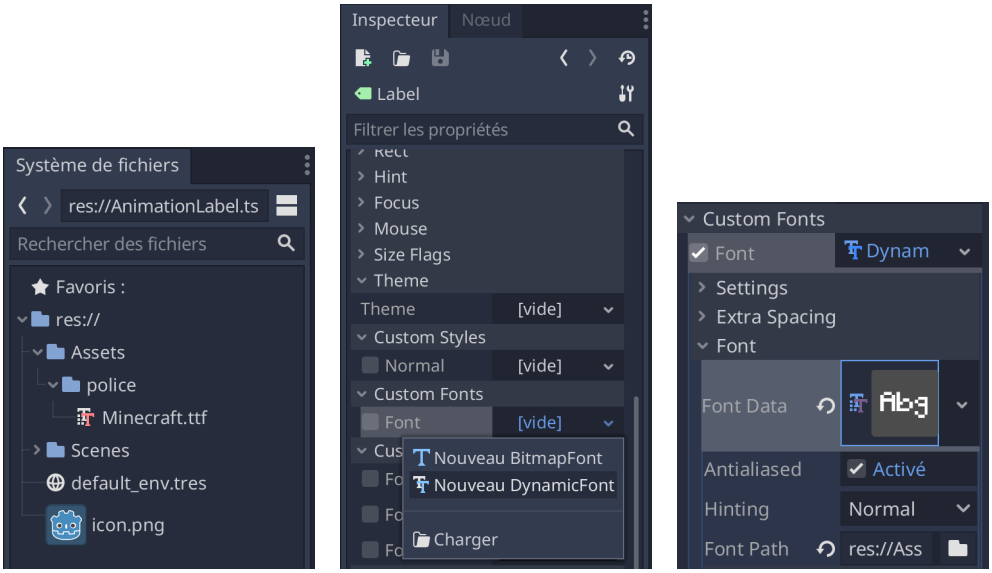
FIGURE 4.2 – Choix et positionnement du texte du label.

4.2.2 Ajout d'une police de caractères

Il est possible de personnaliser le style du texte (taille, couleur, police ...). Pour cela, la première étape est de choisir une police gratuite (pour les plus créatifs, en créer une), dans cet exemple, j'ai choisi la police *Minecraft*, disponible sur cette page https://www.dafont.com/fr/minecraft.font. D'une manière générale, il est utile de toujours réer un répertoire *Assets* à la racine

de votre projet afin d'y ranger les différentes ressources utilisées par celui-ci. Ainsi la police est enregistrée dans ce répertoire (dans l'idéal, dans un sous-répertoire police !)(cf. fig. 4.3a).

Pour utiliser cette police, il faut maintenant créer un *Nouveau DynamicFont* dans la la catégorie *Custum Font* de l'inspecteur du *Label* (cf. fig. 4.3b). Puis dans la sous-propriété *Font*, il faut renseigner le champs *Font Data* avec la police voulue (cf. fig. 4.3c).



(a) Ajout de la police au projet

(b) Création d'une nouvelle nouvelle police pour le *Label*

(c) Sélection de la police

FIGURE 4.3 – Personnalisation de la police d'un *Label*

Vous avez maintenant de quoi configurer votre texte, à vous de jouer !

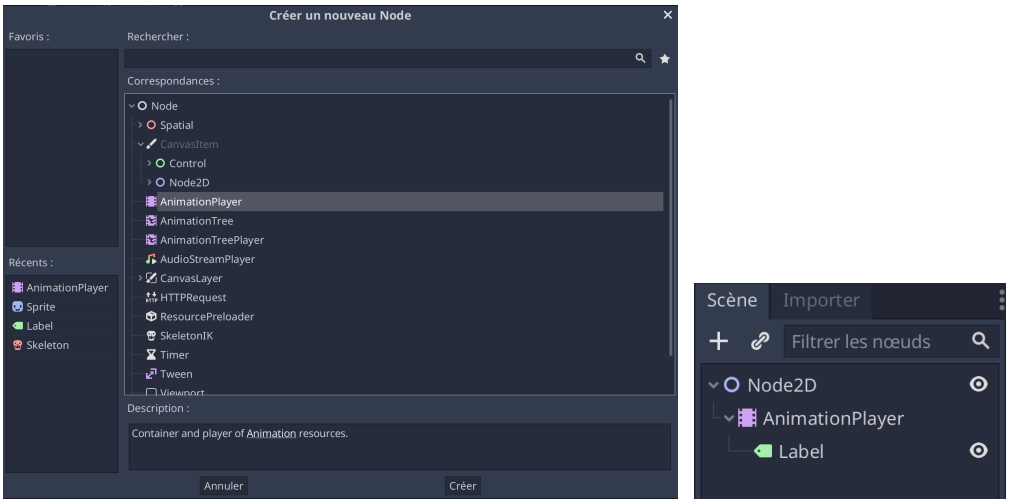
Il faut sauvegarder la scène avant d'en lancer la lecture grace au bouton « lancer la scène » de la barre supérieure (cf. fig. 4.4). Pour que le projet puisse se lancer, il ne reste plus qu'à définir notre scène comme scène principale. Il y a plusieurs moyen de le faire. Le plus simple est sûrement d'effectuer un clic droit sur notre scène dans le gestionnaire de fichiers et de sélectionner « définir comme scène principale » dans le menu contextuel. Et voilà ! Votre

FIGURE 4.4 – Barre de lancement

premier projet Godot est prêt et se lance.

4.3 Et si on animait ?

Dans *Godot*, il est possile d'animer tout élément disponible dans le graphe de scène. Les objets *AnimationPlayer* permettent de créer des animations. L'*AnimationPlayer* est utilisé pour la lecture des données d'animation. Il contient un dictionnaire d'animations (référencées par leur nom) et des temps de fondu personnalisés entre leurs transitions. Un nœud *AnimationPlayer* peut contenir plusieurs animations, qui peuvent s'enchainer automatiquement. Et si en s'inspirant de notre scène précédente, nous voulions animer notre *Label* ? Pour cela, il faut créer un nœud AnimationPlayer, enfant du nœud racine de notre scène (cf. fig. 4.5a).



(a) Création d'un *AnimationPlayer*

(b) Graphe de scène après la création

FIGURE 4.5 – Ajout d'un *AnimationPlayer* à la scène

Puis, nous pouvons déplacer notre *Label* dans le graphe de scène pour que celui-ci soit un nœud fils de l'*AnimationPlayer* créé à l'instant (cf. fig. 4.5b).

Pour générer une animation, il faut sélectionner le nœud *AnimationPlayer* dans notre scène et séléctionner l'onglet *Animation* dans le bandeau du bas (cf. fig. 4.6). Dans ce panneau, il faut commencer par créér une nouvelle animation, en allant cliquer sur *Animation* puis *nouveau* dans le menu contextuel.

Une animation est composée à partir de ce qu'on appelle des clés d'animation (*keyframes* en anglais). Ces clés représentent les principales positions

FIGURE 4.6 – Éditeur d'animations

de l'objet ou du personnage dans l'animation. Le moteur calcule ensuite les positions intermédiaires.

Puisque l'*AnimationPlayer* nous sert à animer le *Label*, il faut sélectionner ce dernier dans le graphe de scène. Dans la barre d'outil de l'espace central, il est possible de créer une clé d'animation pour ce *Label*. La première créée constitue donc la position initiale de l'objet dans l'animation. Nous allons en créer une qui ne gèrera que la translation (cf. fig. 4.7).

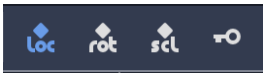


FIGURE 4.7 – Création d'une clé d'animation

Une fois cette clé créée, une piste pour le *Label* apparaît dans le panneau *Animation* (cf. fig. 4.8). C'est à cet endroit qu'il est possible de gérer la durée de l'animation, d'ajouter d'autre clés d'animation pour le *Label*. Pour ajouter une nouvelle clé, il suffit de déplacer le curseur de temps sur la piste, de déplacer le *Label* à l'endroit souhaité, puis de cliquer sur la clé dans la barre d'outil de l'espace de travail. Les différentes clés sont représentées par des losanges sur la piste dans le panneau *Animation*.

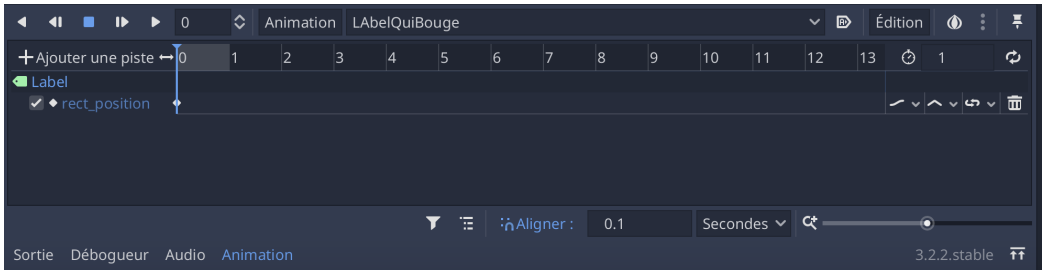


FIGURE 4.8 – Panneau de gestion de l'animation

Chapitre 5

Instanciation de scènes

5.1 Intérêt

Dans le cas d'un petit projet comme celui réalisé au chapitre précédent, il est possible d'envisager une scène unique mais dès que les projets se complexifient, leur graphe de scène et leur nombre de nœuds peuvent devinir colossaux (cf. fig. 5.1).

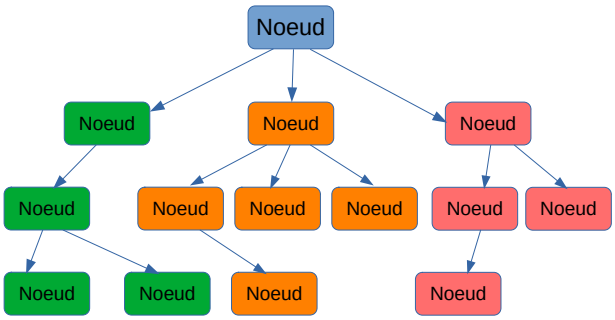
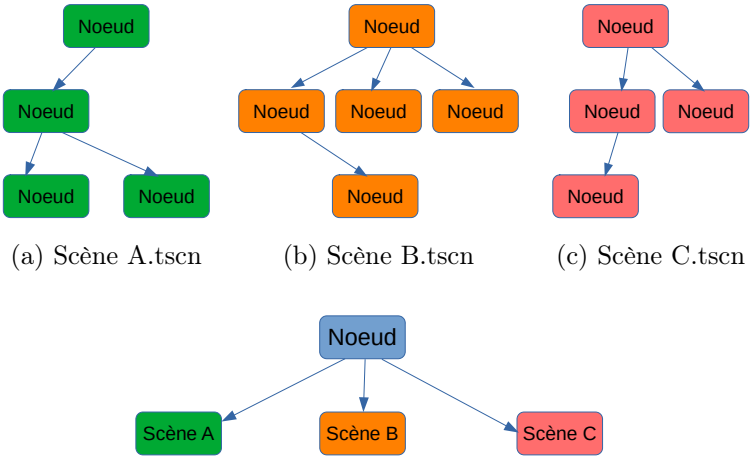


FIGURE 5.1 – Scène unique avant instanciation

C'est ainsi que *Godot* autorise un nombre de scènes quelconque dans un projet. Nous avons vu précédemment qu'une scènce est composée d'un ensemble de nœuds structuré en arbre avec un seul nœud pour racine. Il est possible de créer et sauvegarder autant de scènes que l'on souhaite. Une scène ainsi générée est appelée « packed scene » et a pour extension de nom de fichier *.tscn* (cf. fig.5.2a). Une scène sauvegardée et présente dans l'arborescence de fichiers du projet peur être instanciée dans une autre scène de celui-ci comme s'il sagissait d'un nœud (cf. fig. 5.2d). Cela simplifie considérablement le graphe de scène (cf. fig. 5.2).



(a) Scène A.tscn (b) Scène B.tscn (c) Scène C.tscn

(d) Instanciation

FIGURE 5.2 – Instanciation

5.2 Exemple

Quoi de mieux qu'un petit exemple pour illustrer cette notion ? Imaginons un projet dont la scène principale est constituée d'obstacles statiques et de balles (cf. fig. 5.3).

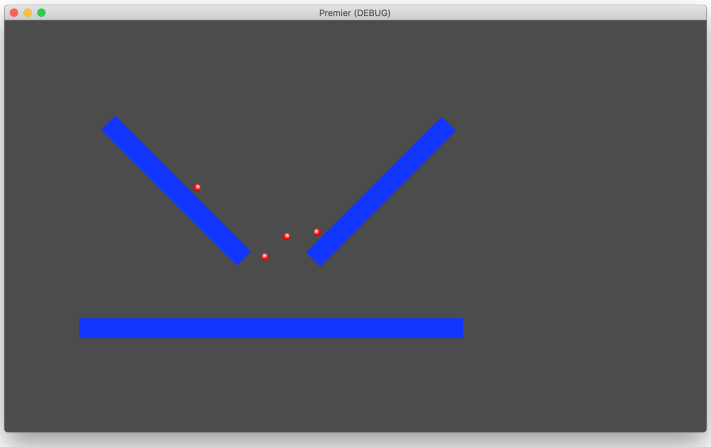


FIGURE 5.3 – Instanciation - Exemple

Puisqu'il y a plusieurs murs et plusieurs balles dans cette scène, il parait intéressant de disposer d'une scène pour représenter un mur (*Mur.tscn*) et

d'une pour représenter une balle (*Balle.tscn*). Ces scènes seront instanciées dans la scène principale (cf. fig. 5.4).

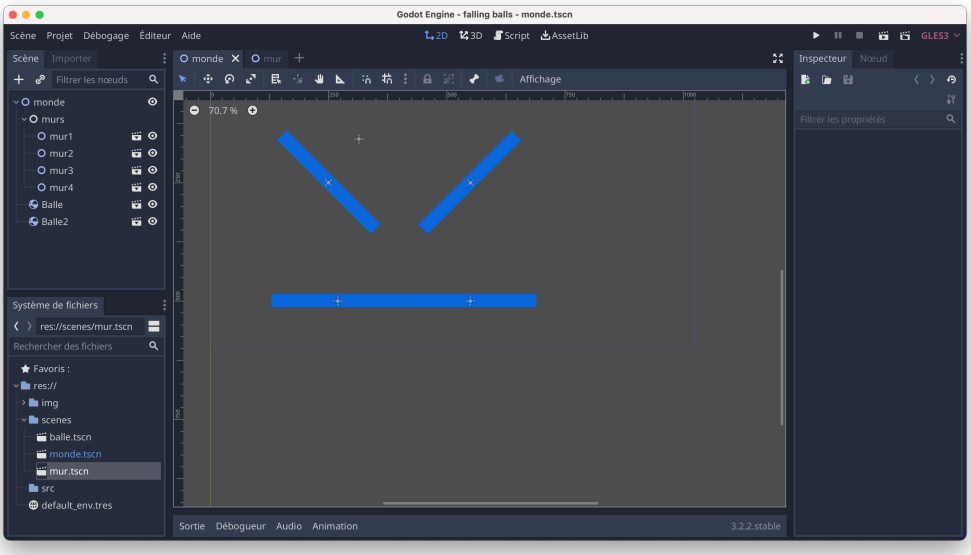


FIGURE 5.4 – Instanciation - Exemple : Scène principale

5.2.1 La scène *Mur.tscn*

Pour modéliser ces murs, nous utilisons des *StaticBody2D*, qui sont classiquement utilisés pour représenter des objets physiques 2D et immobiles dans l'environnement tels que des murs ou des plateformes. Ils permettent la détection des collisions, mais ne bougent pas en réponse à celles-ci. La scène doit donc avoir un nœud racine de type *StaticBody2D*. Il est possible de renommer ce nœud dans le graphe de scène. Pour que le *StaticBody2D* ait une apparence, nous lui ajoutons un nœud enfant de type *Sprite*. Puis , dans l'inspecteur du *Sprite*, il faut ajouter une texture. Je vous fournis une image *mur.png* sur ma page, vous pouvez l'utiliser ou en choisir/faire une autre. À ce moment, un point d'exclamation apparait dans le graphe de scène à côté du *StaticBody2D*. Lorsqu'on le survole, ce sernier fournit plus d'information sur sa présence : un *StaticBody2D* doit avoir un *CollisionShape2D* associé. Un *CollisionShape2D* permet de créer et éditer des formes de collision dans l'espace 2D. Avec ce type de nœud, il est possible de représenter toutes sortes de formes de collision. Une fois le nœud *CollisionShape2D* créé comme fils du *StaticBody2D*, il ne reste plus qu'à choisir sa forme en renseignant sa propiriété *Shape* dans l'inspecteur. Dans notre cas, le mur ayant une forme

rectangulaire, nous choisirons donc une *RectangleShae2D* dont nous adapterons les dimensions au plus près de l'image du *Sprite*. Une fois la forme de la collision bien alignée sur la forme du mur, je vous conseille vivement de rendre ces deux éléments non sélectionnables individuellement. Cela évitera d'en déplacer un et d'avoir des mauvaises surprises à l'éxecution. En effet, la *CollisionShape2D* étant invisible, on peut ne pas s'apercevoir qu'elle est à la mauvaise place et se retrouver avec des collisions à des endroits inattendus. Pour éviter cette erreur, il suffit de sélectionner le noeud *StaticBody2D* dans le graphe de scène et de rendre ses enfants non sélectionnables. La scène est prête, il ne reste plus qu'à sauvegarder la scène dans un répertoire *scenes* créé pour recevoir toutes les scènes du projet (cf. fig. 5.5).

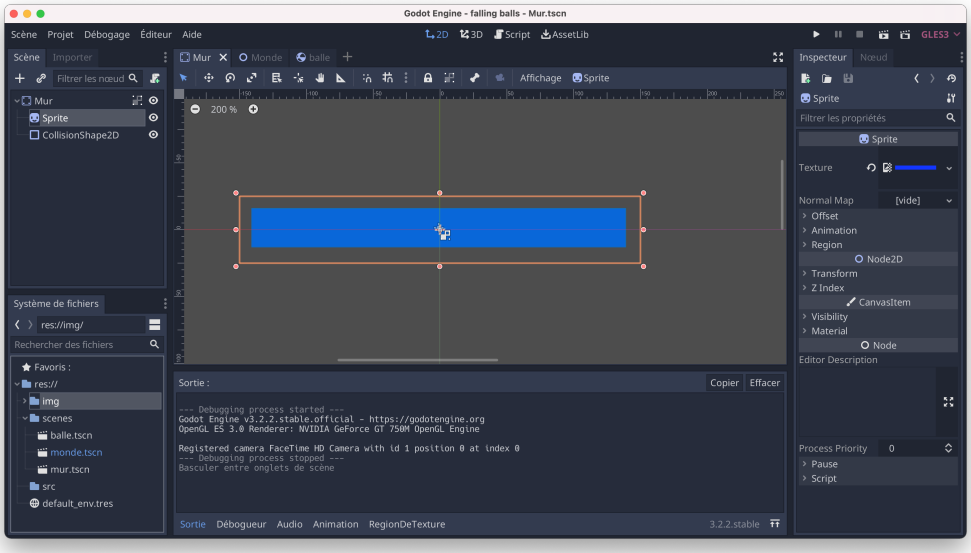


FIGURE 5.5 – Instanciation - Exemple : Scène « Mur.tscn »

5.2.2 La scène *Balle.tscn*

Maintenant, il nous faut une scène qui modélise une balle. Nous allons utiliser un *RigidBody2D* pour lui fournir un comportement physique. De même que pour le mur, le *RigidBody2D* aura deux nœuds fil : un*Sprite* et un *CollisionShape2D* (cf. fig. 5.6). La scène *Balle.tscn* doit faire partie du projet, le mieux est de l'enregistrer dans le sous-répertoire *scenes* du projet.

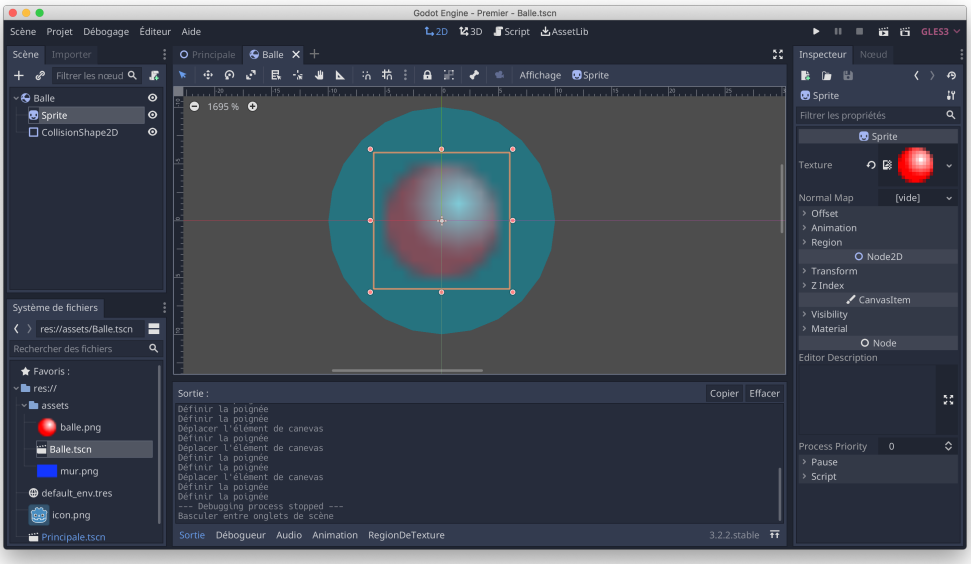


FIGURE 5.6 – Instanciation - Exemple : Scène « Balle.tscn »

5.2.3 La scène principale

Pour générer notre scène principale, nous allons déjà construire les murs. Le nœud racine de notre monde sera un *Node2D*. Nous allons lui ajouter plusieurs instances de la scène *mur.tscn*. Pour cela, une fois le nœud racine de la scène principale sélectionné, il faut cliquer sur le bouton en forme de lien (ou d'agraphe) dont le texte de survol est « Instancie un fichier de scène comme nœud. », puis sélectionner le fichier *Mur.tscn*. Le but est d'ajouter une instance de *Balle* dans la scène *Principale*. Une fois les murs ajoutés et positionnés à notre guise, nous pouvons ajouter des instances de balles à notre scène. Chaque nœud sera ajouté en (0 ; 0) mais il est possible de le déplacer à la souris. Il est ainsi possible d'ajouter autant d'instances que souhaité en répétant l'opération (cf. fig. 5.7).

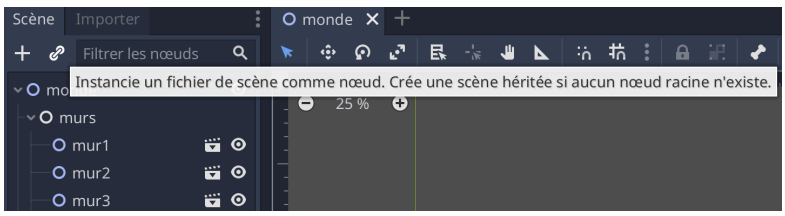


FIGURE 5.7 – Instanciation - Exemple : Création d'insntance

Le projet peut être lancé, les balles vont subir la pesanteur, tomber et

entrer en collision avec les obstacles (cf. fig. 5.3).

Chapitre 6

Les *tilemaps*

6.1 Les *tilemaps* dans *Godot*

Un jeu vidéo à base de tuiles (*tiles*) est un type de jeu vidéo dans lequel l'aire de jeu (*layout*) est constituée de petites images carrées (moins souvent, rectangulaires, parallélogrammatiques ou hexagonales) disposées sur une grille (*tilemap*). L'écran est ainsi représenté par une grille composée de nombreuses cases, sur lesquelles sont appliquées une image par case. L'ensemble complet de tuiles disponibles et utilisables est appelé *tileset*. En général, pour augmenter les performances, ces images sont regroupées dans une image unique, on parle d'« atlas de texture » (ou *sprite-sheet*). Une sous-image est dessinée à l'aide de ses coordonnées pour la sélectionner dans l'atlas. Les images constituant un atlas peuvent être de dimensions variables.

Les jeux basés sur des tuiles simulent généralement une vue de dessus, une vue de profil ou bien une vue « 2.5D » et sont presque toujours en deux dimensions. Dans *Godot*, la grille sur laquelle on va insérer les tuiles pour dessiner notre *layout* est appelée une *TileMap*. Il est possible d'ajouter des fonctionnalités à une tuile afin de gérér les collisions, occlusions, et de lui ajouter un shéma de déplacement.

Dans le cours de création de ressources, vous allez apprendre à générer vos propres *tiles-sheets*, mais d'ici là nous allons utiliser des *tiles-sheets* libres de droit, vous pourrez en trouver ici : https://kenney.nl/. Pour l'exemple que je vais évoquer ensuite, j'ai choisi d'utiliser le *Simplified platformer-pack*, et pour l'instant j'utiliserai le *tile-sheet* qui contient les éléments de décor (cf. fig. 6.1).

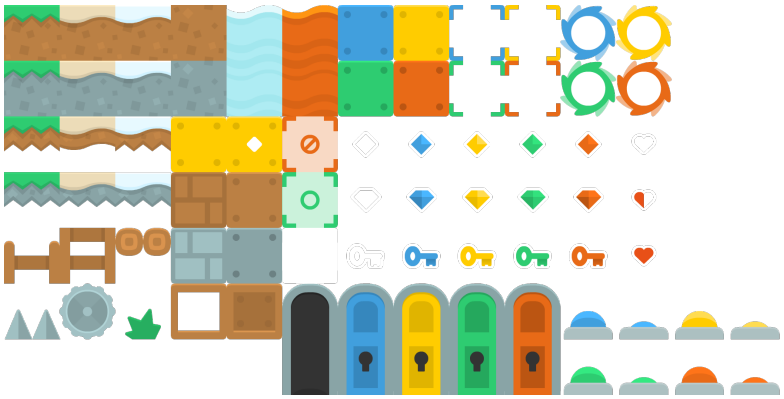


FIGURE 6.1 – *Tile-sheet* - Trouvée sur le site de Kenney

6.2 Utiliser les *tilemaps*

Nous allons créer un projet qui utilisera une *TileMap*. Pour cela, il faut ajouter le *sprite-sheet* que nous voulons employer dans le répertoire du projet afin qu'il soit disponible dans le gestionnaire de fichiers de ce dernier.

Avant de commencer, il faut sélectionner le *sprite-sheet* dans le gestionnaire de fichiers du projet, puis aller dans l'onglet « importer » pour désactiver la propriété « filtre » et ré-importer l'image. Cette action est à réaliser à cause du comportement par défaut de *Godot* lors de l'import des images 2D. En effet, il appplique un filtre par interpolation ce qui a des conséquences néfastes sur les bordures entre les tuiles (flou). Or, l'utilisation d'un *tileset*, impose que les tuiles voisines s'accordent parfaitement ensemble.

Puis, il faut ajouter un nœud *TileMap* à notre scène. La grille apparaît alors dans la vue 2D de la scène (cf. fig. 6.2)

Dans l'inspecteur de la *TileMap*, il est possible de choisir entre autres, la forme des tuiles et la taille des cellules. C'est également ici qu'il est possible de charger un *TileSet* existant ou d'en créér un nouveau (cf. fig. 6.3a). Un *TileSet* est une resource qui contient les données des tuiles : textures, informations de collision, d'occlusion .... Lorsque le jeu tourne, la *TileMap* génère un unique objet en combinant toutes les tuiles.

Une fois le nouveau *TileSet* créé (ou chargé dans le cas où on en réutilise un issu d'un précédent projet *Godot*), en allant le sélectionner (click sur le *TileSet* dans l'inspecteur), le panneau d'édition apparaît (cf. fig. 6.3b). On ajoute toutes les textures (dans notre cas, pour l'instant, il n'y a qu'un fichier) necessaires à la création des tuiles en cliquant sur le bouton en forme de +, dont le texte de survol est « Ajouter des textures au *TileSet* » (cf. fig. 6.3b). À partir des images chargées, nous pouvons créér nos tuiles, en cliquant sur

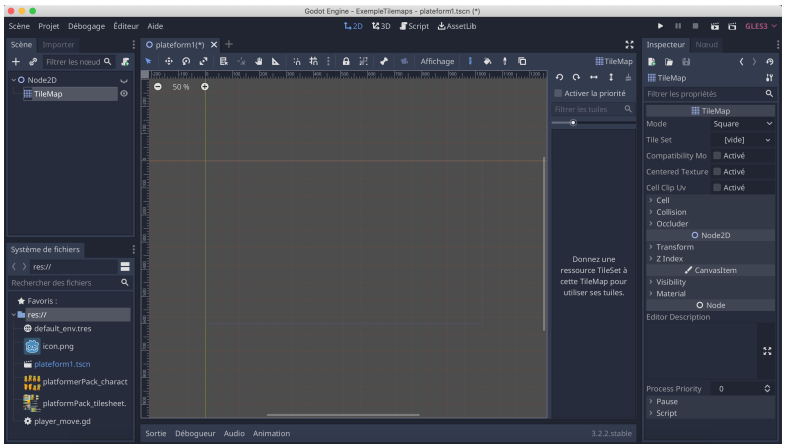
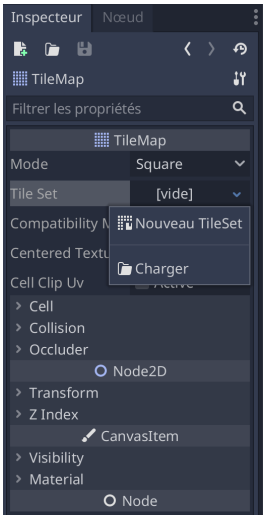
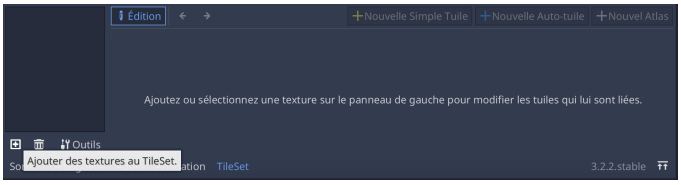


FIGURE 6.2 – Ajout d'un nœud *TileMap*



(a) Création d'un *TileSet* pour la *TileMap*



(b) Panneau d'édition du *TileSet*

FIGURE 6.3 – Création d'un *TileSet*

le bouton "Nouvelle tuile simple" (cf. fig. 6.4). Pour faciliter la sélection des éléments, je vous conseille de cliquer sur le bouton « Activer l'aimantation et afficher la grille ». La grille apparaît alors, il suffit de sélectionner sur l'image, l'ensembles des cellules qui constituent la tuile. Un rectangle jaune entoure toutes les tuiles créées (cf. fig. 6.4).

Lorsqu'on crée une tuile, si on souhaite que celle-ci soit un élément de terrain type sol ou obstacle, nous devons lui ajouter une forme de collision (cf. fig. 6.5).

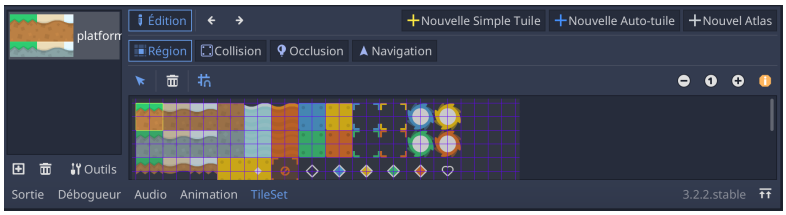


FIGURE 6.4 – Création d'une nouvelle tuile

FIGURE 6.5 – Ajout de la collision sur une tuile

Pour insérer les tuiles créées dans votre *TileMap*, il suffit de la sélectionner dans le graphe de scène. À côté de la vue 2D apparaît la liste des tuiles disponibles (cf. fig. 6.6). En cliquant sur une tuile, on la choisit ; pour la placer, il suffit de cliquer à l'endroit voulu. En cas d'erreur, un click droit sur la tuile ajoutée à tort permet sa suppression.

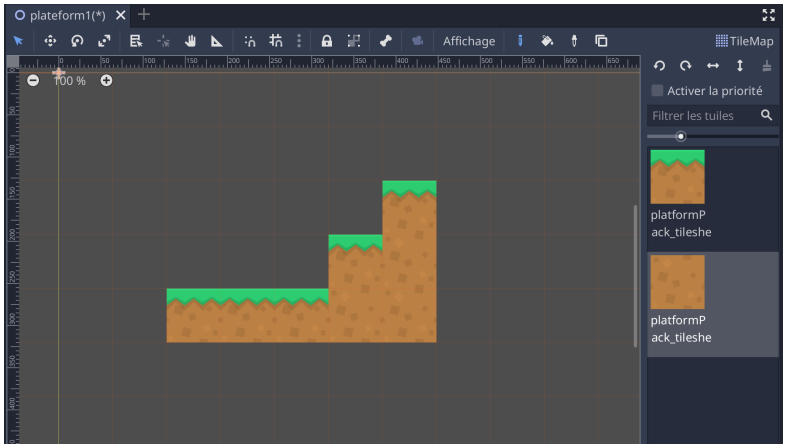


FIGURE 6.6 – Positionnement des tuiles sur la *TileMap*

6.3 Les *Autotiles*

Les *autotiles* permettent de définir des groupes de tuiles ainsi que des règles permettant de déterminer laquelle d'entre elles sera dessinée en fonc-

tion du contenu des tuiles voisines. Leur fonctionnement est très bien décrit sur le site officiel de *Godot*, je vous invite à aller lire cette page avant de commencer à utiliser les *autotiles*. Le *sprite-shheet* utilisé précédemment ne dispose pas d'assez de tuiles pour construire un *bitmask* 2×2. J'en ai donc choisi un autre, plus complet toujours ches *Kenney* (cf. fig. 6.7).

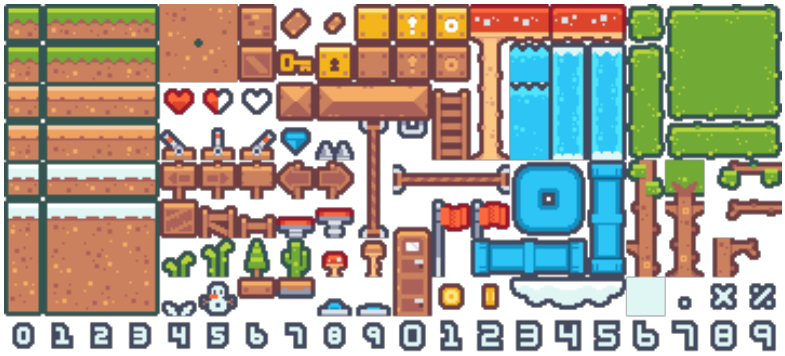


FIGURE 6.7 – *Tile-sheet* permettant de configurer un *bitmask* 2×2 - Trouvée sur le site de Kenney

Comme dans le cas des tuile classiques, il faut d'abord disposer d'une *TileMap*, puis d'un *TileSet* afin de pouvoir charger le *tile-sheet* choisi. Il faut ensuite créer une nouvelle *auto-tuile* et de séléctionner l'ensemble des tuiles qui vont permettre de définir le *bitmask* et qui seront utilisées dans la *map* (cf. fig. 6.8).

FIGURE 6.8 – Nouvelle*auto-tuile* et sélection de la région à utiliser.

Il faut ensuite, définir le bitmask sur cette sélection (cf. fig. 6.9). Dans cet exemple, je choisis un format de bitmask à 2×2, notons qu'on pourrait choisir 3×3, cela imposerait un plus grand nombre de tuiles utilisables et de cas gérés. Le principe est le même, vosu pourrez essayer !

Il ne faut pas oublier d'ajouter la détection de collision sur les tuiles qui seront utilisées (cf. fig. 6.10).

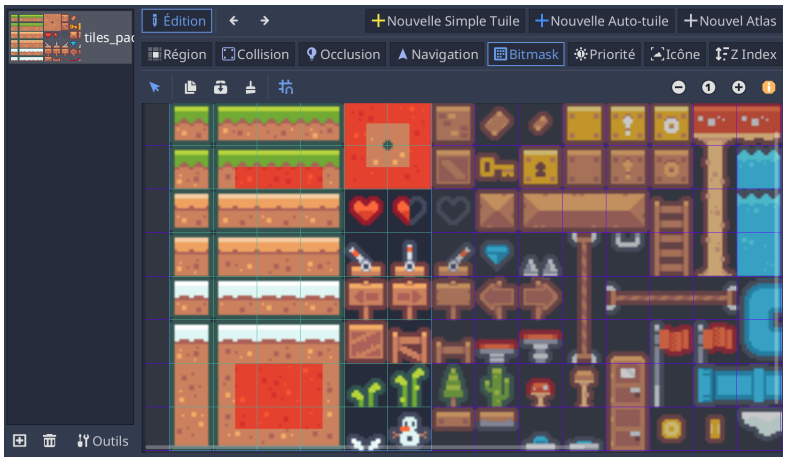


FIGURE 6.9 – Définition du *bitmask*.

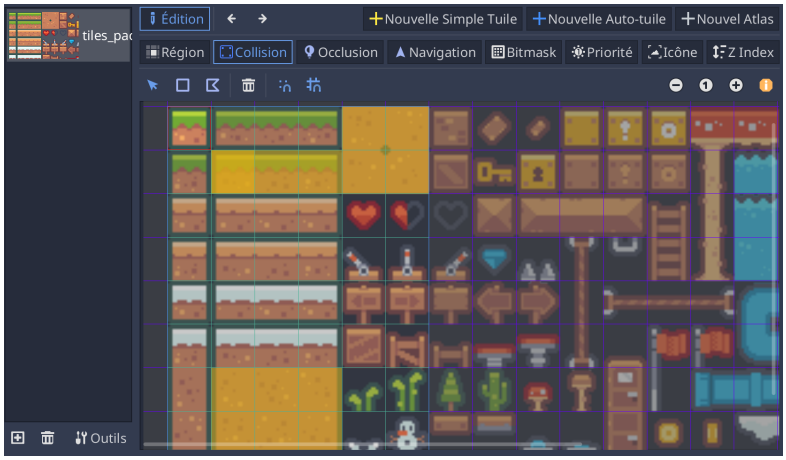


FIGURE 6.10 – Ajout des masques de collisions sur les tuiles de l'auto-tile.

L'*auto-tile* apparait dans en tant que tuile, lorsque la *TileMap* est sélectionnée dans le graphe de scène. Si on choisit cette *auto-tile*, et qu'on dessine dans notre scène, les tuiles sont choisies et modifiées automatiquement en fonction du contenu de leurs voisines (cf. fig. 6.11).

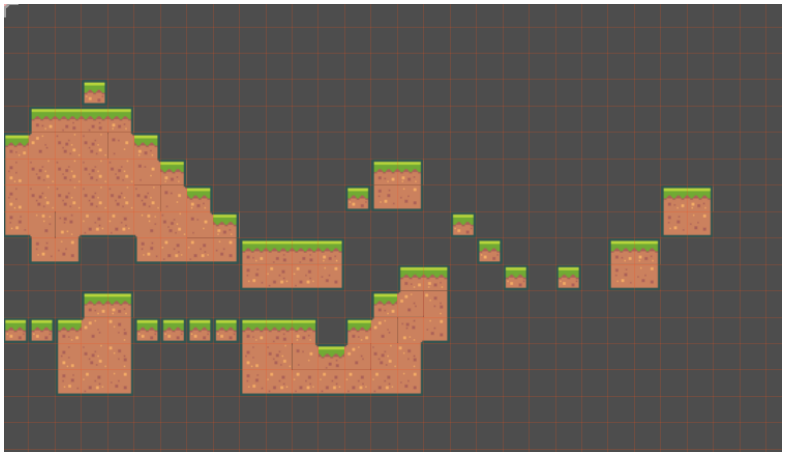


FIGURE 6.11 – Exemple de *TileMap* dessinée avec une auto-tuile.

Chapitre 7

Les scripts

Les langages de script les plus utilisés dans *Godot* sont *GDScript* et *VisualScript* grâce à leur niveau d'intégration dans l'éditeur *Godot*, tandis que le *C#* et *C++* doivent être édités dans un éditeur ou un IDE séparé. Un script ajoute un comportement à un nœud, il permet de contrôler le fonctionnement du nœud et de définir ses interactions avec d'autres nœuds.

7.1 Ajouter un script *GDScript* à un nœud

Dans la scène précédente, il serait intéressant d'ajouter un personnage qu'on pourra déplacer grâce aux événements clavier. On utilise les *KinematicBody2D* pour instancier des objets qu'on souhaite contrôler via un script. Ils détectent les collisions avec d'autres corps en mouvement, mais ne sont pas affectés par les propriétés physiques du moteur, comme la gravité ou la friction, c'est donc au développeur de les gérer dans le script définissant le comportement du nœud. Nous allons donc commencer par ajouter un nœud fils de type *KinematicBody2D* au nœud racine de notre scène. Il faut ajouter à ce dernier deux nœuds fils pour le rendre utilisable : un *Sprite* et un *CollisionShape2D*. Ajouter ensuite une image de texture au *sprite* et une forme au *CollisionShape*. Déplacez votre personnage où vous le souhaitez dans la scène, cela sera sa position initiale (cf. fig. 7.1).

Maintenant, pour pouvoir gérér les déplacements du *KinematicBody2D*, nous lui ajoutons un script par un click droit sur l'élément *KinematicBody2D* dans le graphe de scène, puis sélectionner « Attacher un script » (cf. fig. 7.2a).

La boîte de dialogue de création de script apparaît (cf. fig. 7.2b).

Cette boîte de dialogue vous permet de définir le langage du script, le nom de la classe et d'autres options. Dans *GDScript*, le fichier représente la classe, ce qui rend le nom de la classe non modifiable. Le nœud auquel nous

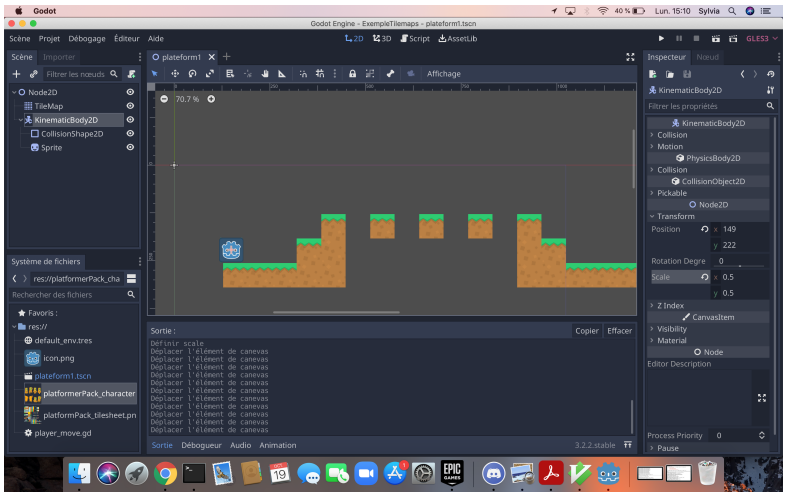
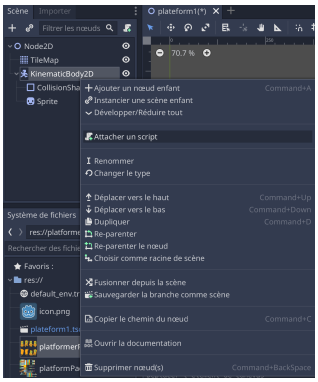
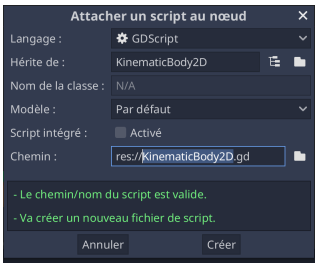


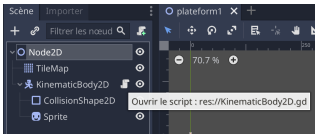
FIGURE 7.1 – Ajout et configuration d'un *KinematicBody2D*



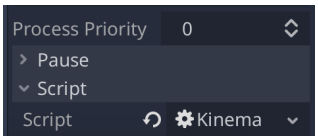
(a) Ajout d'un script au nœud *KinematicBody2D*



(b) Boîte de dialogue de création du script



(c) Graphe de scène après l'ajout du script



(d) Propriété Script du *KinematicBody2D*

FIGURE 7.2 – Ajout d'un script à la scène

attachons le script est un *KinematicBody2D*, donc le champ « Hérite de » sera automatiquement rempli avec *KinematicBody2D*.

Une fois le script créé, il est ajouté au nœud. L'icône « Ouvrir le script » apparaît à côté du nœud dans le graphe de scène (cf. fig. 7.2c) et le script est ajouté dans la propriété *Script* dans l'inspecteur du *KinematicBody* (cf.

fig. 7.2d).

7.2 Contenu du script *GDScript*

Nous allons maintenant pouvoir éditer le script et le compléter. En cliquant sur l'icône « Ouvrir le script » du nœud *KinematicBody2D*, notre script s'affiche dans l'espace de travail « Script » (cf. fig. 7.3).

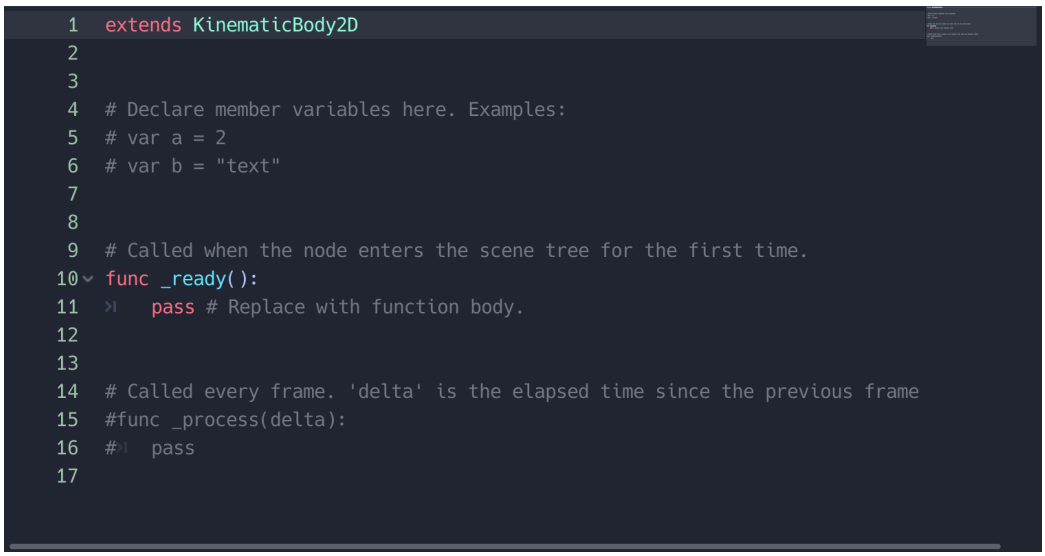


FIGURE 7.3 – Script généré lors de la création

Le script généré ne contient pas grand chose : l'héritage de la classe *KinematicBody2D* et la fonction `_ready()` qui est appelée lorsque le nœud entre dans la scène active mais n'est pas le constructeur, la fonction `_init()` l'est. Est également présente mais commentée la fonction `_process(delta)` où delta est le temps écoulé depuis la *frame* précédente. En dehors des déclarations de fonctions, en général en début de fichier (mais en réalité à l'endroit qu'on veut), il est possible de déclarer les variables membres (cf. fig. 7.4a).

L'utilisation du mot-clé `export` sur les variables `run_speed`, `jump_speed` nous permet d'avoir accès à leurs valeurs dans l'inspecteur. Ce mot-clé est donc optionnel, mais est pratique pour ajuster les valeurs de ces variables tel qu'on modifie les propriétés intégrées d'un nœud. Dans l'inspecteur du nœud *KinematicBody* apparaît la section "Script Variables" (cf. fig. 7.4b). Tout changement de la valeur ici, entraine une modification de cette valeur dans le script.

Pour définir les déplacement du *KinematicBody*, nous allons ajouter le code nécessaire dans la fonction `_process(delta)`, qui est appelée à chaque

```gdscript
extends KinematicBody2D

export var run_speed  = 1
export var jump_speed = -100
var gravity    = 75

var velocity = Vector2()
```

(a) Déclaration des variables dans le script

(b) Section « Script Variables » dans l'inspecteur

FIGURE 7.4 – Déclaration des variables membres

frame et donc idéale pour mettre à jour des informations de déplacement qui sont suceptibles de changer en permanence. Le déplacement du personnage sera controlé à l'aide des flèches directionnelles du clavier. Il faut donc détecter les actions sur les touches et appliquer un déplacement en fonction de ces actions. C'est ce que nous faisons dans la fonction `get_input()` (cf. fig. 7.5).

```gdscript
func get_input():
    velocity.x = 0
    if Input.is_action_pressed('ui_right'):
        velocity.x += run_speed
    if Input.is_action_pressed('ui_left'):
        velocity.x -= run_speed
    if Input.is_action_pressed('ui_up') and is_on_floor():
        velocity.y = jump_speed
```

FIGURE 7.5 – Fonction qui traite les événements clavier

L'appel à la fonction `Input.is_action_pressed()` avec le nom de la touche en paramètre, retourne `true` si la touche est effectivement enfoncée ou `false` sinon. Lorsque nous détectons un appui sur l'une des touches fléchées haut, droit ou gauche, nous mettons à jour le vecteur de déplacement. Cette fonction qui « écoute » les événements clavier doit donc être appelée à chaque *frame*, donc dans la fonction `_process`. Cela permet la mise à jour du vecteur de déplacement. Pour déplacer un KinematicBody2D, il ne faut pas définir directement sa propriété position. Les méthodes `move_and_collide()` ou `move_and_slide()` permettent son déplacement le long d'un vecteur donné et le déplacement s'arrête instantanément si une collision est détectée avec

un autre élément gérant lui aussi les collisions (cf. fig. 7.6). Si une réponse à la collision doit avoir lieu, elle doit être codée manuellement.

```
1  extends KinematicBody2D
2
3  export var run_speed  = 100
4  export var jump_speed = -100
5  var gravity    = 90
6
7  var velocity = Vector2()
8
9  func get_input():
10     velocity.x = 0
11     if Input.is_action_pressed('ui_right'):
12         velocity.x += run_speed
13     if Input.is_action_pressed('ui_left'):
14         velocity.x -= run_speed
15     if Input.is_action_pressed('ui_up') and is_on_floor():
16         velocity.y = jump_speed
17 # Called every frame. 'delta' is the elapsed time since the previous frame.
18 func _process(delta):
19     get_input()
20     velocity.y += gravity * delta
21     velocity = move_and_slide(velocity, Vector2(0,-1))
22
23 # Called when the node enters the scene tree for the first time.
24 func _ready():
25     pass # Replace with function body.
```

FIGURE 7.6 – Script de déplacement du *KinematicBody2D*

7.3 La même chose en *VisualScript* ?

Il est évidemment possible de gérer les déplacement du personnage avec un script *VisualScript*, mais il faut d'abord comprendre ce qu'est *VisualScript*. Il mérite bien un chapitre à lui tout seul ! C'est pourquoi, nous reviendrons faire le script associé au personnage quand nous aurons appris à manipuler *VisualScript*, rendez-vous au chapitre 9.

Chapitre 8

Animation de *sprites*

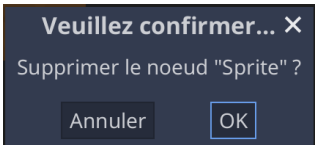
Pour créer des personnges animés, il existe la classe *AnimatedSprite*. Pour pouvoir animer un personnage, il faut disposer d'images individuelles, séquentielles du mouvement ou bien d'une *sprite-sheet* unique contenant toutes les images de l'animation. De la même manière que pour les *tilemaps*, bientôt vous aurez imaginé vos propres images représentant les différentes étapes du mouvement de vos personnages (stockées sous forme d'images individuelles ou de *sprite-sheets*), d'ici là je vais utiliser celles issues du même projet que précédemment de chez Kenney.

8.1 Animation d'un personnage à partir d'images séquentielles

Nous allons poursuivre le projet commencé, le but étant de rendre le personnage plus « mobile ». La première étape est de télécharger les différentes images du personnage et de les enregistrer dans le répertoire du projet, elles doivent alors apparaître dans le gestionnaire de fichier du projet dans *Godot* (cf. fig. 8.1a). Puis, il faut remplacer le nœud *Sprite* du nœud *KinematicBody2D* que j'ai renommé « Perso » par un nœud de type *AnimatedSprite* (cf. fig. 8.1). Dans l'inspecteur de l'élement *AnimatedSprite*, il faut créer un « Nouveau SpriteFrames » (cf. fig. 8.2a), puis le sélectionner afin d'ouvrir son éditeur (cf. fig. 8.2b). Un *SpriteFrames* est un conteneur de données qui permet de définir une animation pour le nœud *AnimatedSprite*.

Il est posssible et même recommandé de modifier le nom du *SpriteFrames* ; celui-ci s'appelant « default » pour le moment. Nous allons générer l'animation de course, je choisis donc de l'appeler « run ». Idéalement, il faut créer une animation pour chaque action possible du personnage : saut, accroupissement, escalade ....

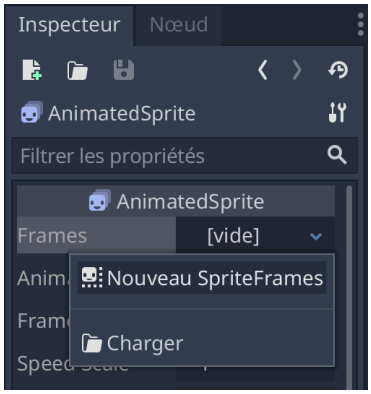
(a) Images destinées à animer notre personnage et intégrées à notre projet - Trouvées sur le site de Kenney



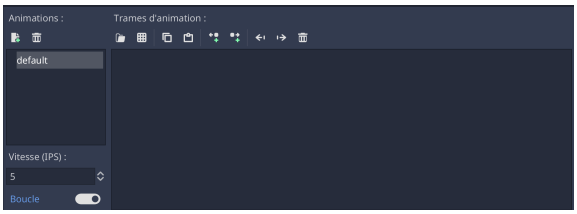
(b) Suppression du nœud *Sprite*

(c) Création du nœud *AnimatedSprite*

FIGURE 8.1 – Configuration du personnage



(a) Création d'un nouveau *SpriteFrames*



(b) Édition du *SpriteFrames*

FIGURE 8.2 – Création d'une animation

Depuis le gestionnaire de fichier, faites glisser les images *PlatformChar_walk1.png* et *PlatformChar_walk2.png* dans la partie centrale de l'éditeur de *SpriteFrames* (cf. fig. 8.3).

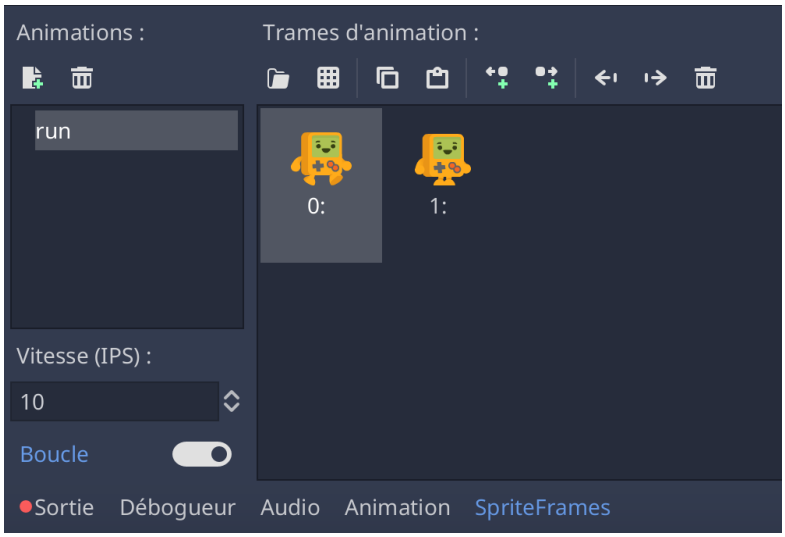


FIGURE 8.3 – Ajout des images permettant de générer l'animation du personnage

Dans l'inspecteur de l'*AnimatedSprite*, cochez la case propriété de *Playing*, l'animation se joue maintenant dans la fenêtre d'affichage. Par contre, celle-ci se joue en permanence. Le but est de contrôler cette animation afin qu'elle ne se joue que dans le cas où notre personnage court. Pour cela, il faut déclencher l'animation ou l'arrêter dans le script qui lui est attaché. J'ai choisi de lancer l'animation « run » en cas de déplacement horizontal et de stopper toute animation si la vitesse de déplacement est nulle (cf. fig. 8.4).

```
9  func get_input():
10     velocity.x = 0
11     if Input.is_action_pressed('ui_right'):
12         velocity.x += run_speed
13         $AnimatedSprite.play("run")
14     if Input.is_action_pressed('ui_left'):
15         velocity.x -= run_speed
16         $AnimatedSprite.play("run")
17     if Input.is_action_pressed('ui_up') and is_on_floor():
18         velocity.y = jump_speed
19 # Called every frame. 'delta' is the elapsed time since the previous frame.
20 func _process(delta):
21     get_input()
22     velocity.y += gravity * delta
23     velocity = move_and_slide(velocity, Vector2(0,-1))
24     if velocity.length()==0:
25         $AnimatedSprite.stop()
```

FIGURE 8.4 – Ajout de la gestion des animations du personnage dans le script

8.2 Animation d'un personnage à partir d'un *sprite-sheet*

Une autre solution pour animer notre personnage serait de le faire avec la *sprite-sheet*. Nous utiliserons celle issue du même projet que précédemment (cf. fig. 8.5). Pour cela, il faut placer le fichier *platformerPack_ character.png* dans le répertoire de travail du projet. (cf. fig. 8.5).

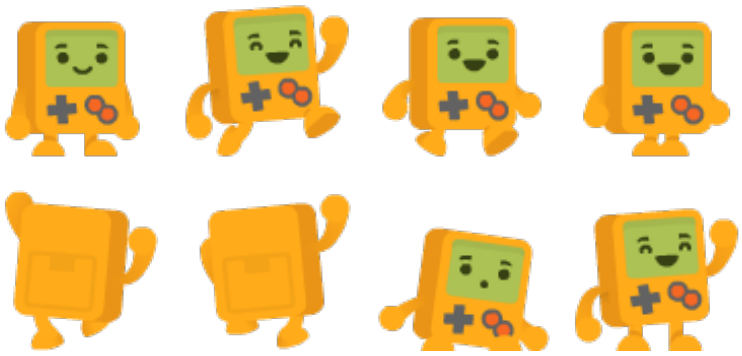


FIGURE 8.5 – *Sprite-sheet* destinée à animer notre personnage - Trouvée sur le site de Kenney

Le graphe de scène est identique à celui créé pour l'animation à partir de plusieurs images. C'est dans l'éditeur du *SpriteFrames* que la démarche diffère. Nous allons créer une nouvelle animation et sélectionner dans l'inspecteur « Ajouter des trames depuis une feuille de Sprite » (cf. fig. 8.6).

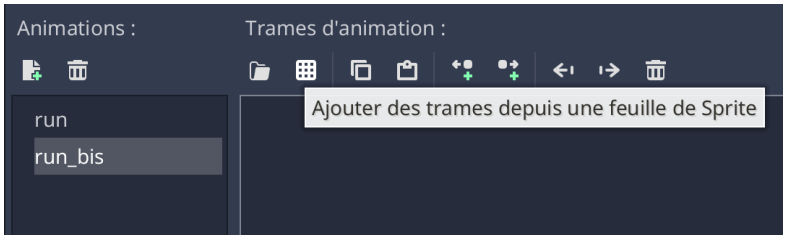


FIGURE 8.6 – Ajout de la feuille de *sprites*

Sélectionnez le fichier contenant la *sprite-sheet* voulue et une fenêtre l'affichant s'ouvre alors (cf. fig. 8.7a). L'image est quadrillée, le nombre de lignes et de colonnes de cette grille est paramétrable afin de pouvoir ajuster la taille des cellules à la taille des images contenues dans notre *sprite-sheet*,

dans notre cas il y a 2 lignes de 4 colonnes. Pour effectuer l'animation de course de notre personnage, il faut sélectionner les deux premières images et cliquer sur « Ajouter 2 Trame(s) » (cf. fig. 8.7b). La suite de la démarche

(a) Ajout d'une feuille de *sprites* à l'animation

(b) Choix des images à ajouter

FIGURE 8.7 – Configuration d'une nouvelle animation

est identique à la démarche présentée dans le cas des images séquentielles.

Chapitre 9

Visual-scripting

Ce chapitre est très largement inspiré du cours de Nicolas Jouandeau, je le remercie d'avoir partagé ses compétences ! Le Visual Scripting est présent dans les moteurs de jeu les plus célèbres. L'objectif est de permettre la définition de comportement logique uniquement avec des graphes et des représentations visuelles. Le Visual Scripting est un outil conçu pour faciliter l'accès à la programmation. Comme le code est plus visuel, il a besoin de moins de pensée abstraite pour être compris. Cela simplifie les interactions entre artiste, animateur, concepteur de jeux..., et programmeurs.

Le Visual Scripting consiste à définir un graphe définissant le comportement des nœuds composant la scène.

9.1 Pré-requis

La création d'un script *VisualScript* fonctionne de la même manière qu'avec *GDScript*, il suffit juste de choisir *VisualScript* au moment du choix du type de script lors de sa création. *VisualScript* est fortement basé sur les fonctions. Une fonction est un canevas individuel avec des nœuds connectés. Un script peut contenir plusieurs fonctions.

Dans le graphe d'un visual script, il faut savoir identifier les différents éléments :

- Les liens blancs matérialisent le fil d'exécution du programme (manipuler les liens blancs en les prenant par leur début)
- Les liens bleus clairs représentent le passage de paramètres (manipuler les liens bleus clairs en les prenant par leur fin)
- Les rectangles blancs correspondent à des définitions des fonctions
- Les rectangles verts représentent les accesseurs *Get-Set* de variables
- Les rectangles violets représentent les opérateurs

— Les rectangles rouges indiquent les appels de fonction
— Les rectangles roses représentent les constantes
— Quand un rectangle est sélectionné, il apparaît en bleu foncé

9.2 Premier exemple

9.2.1 Incrémenter un *label* lors d'un *click*

L'idée est simple : réaliser une scène composée d'un *Button* et d'un *Label* et lors d'un click sur le boutton, incrémenter la valeur du *label*. La première étape est donc de créer la scène pricipale. Le nœud racine de type *Control* est nommé *Main*. Il a deux nœuds fils : un *Label*, dont le champ *Text* est initialisé à 0 et un nœud *Button* dont le champ *Text* vaut *Count click* (cf. fig 9.1). On ajoute un *VisualScript* sur le nœud *Main*, ce script est nommé

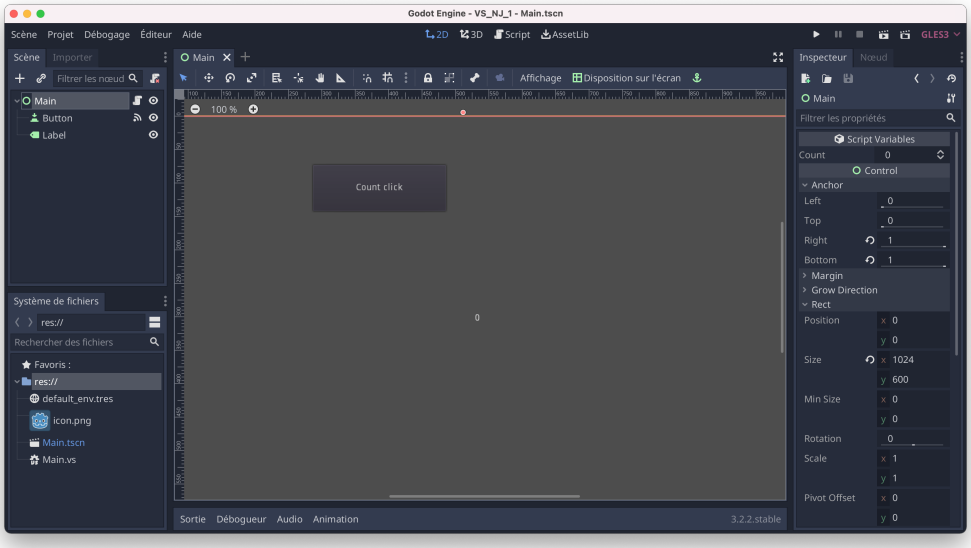
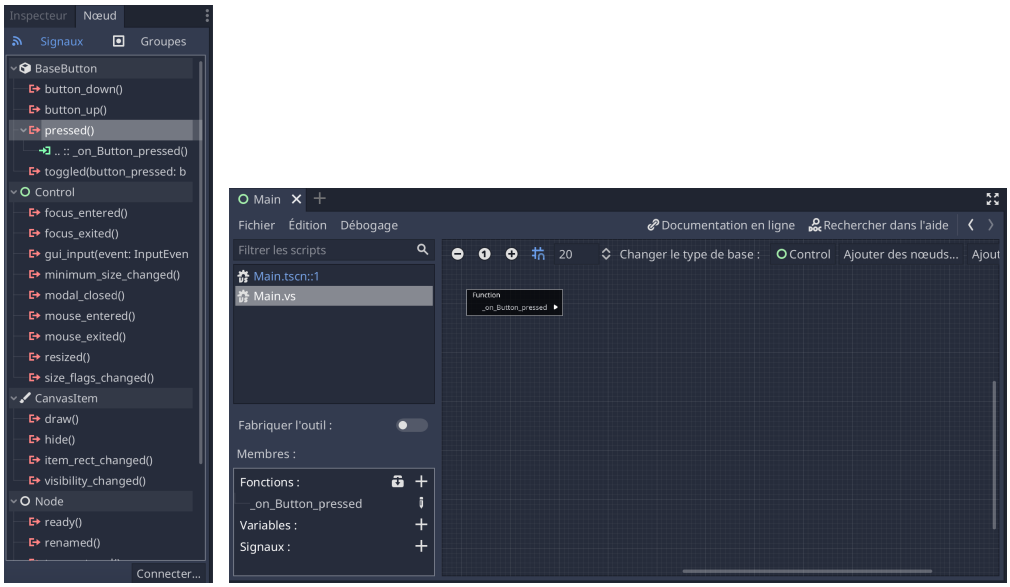


FIGURE 9.1 – Scène basique pour notre premier script *VisualScript*

Main.vs. Pour effectuer une action lors d'un click sur le bouton, il faut que celui-ci emette un signal en cas de détection de cette action. Pour cela, il faut sélectionner le bouton dans le graphe de scène puis, dans le panneau de l'inspecteur, sélectionner l'onglet *Noeud* et connecter le signal *pressed()* à la fonction *on_Button_presssed()* (cf. fig. 9.2a).

Cette fonction apparaît dans le script sous forme d'un rectangle blanc et apparaît dans sa liste des fonctions (cf. fig. 9.2b). La suite se déroule dans le



(a) Ajout d'un signal sur le bouton (b) Apparition de la fonction *on_ Button_ pressed()* dans le script

FIGURE 9.2 – Ajout d'un signal sur le bouton et conséquences dans le script

visual script, il faut réaliser cette suite d'opérations :

- Définir une variable *count* de type int, exportable et initialisée à 0. Ajouter ses accesseurs au script (Glisser-déplacer une variable dans la zone du script permet d'obtenir un nœud *Get*, glisser-déplacer avec Ctrl permet d'obtenir un nœud *Set*). Être exportable permet de modifier la valeur de la variable via l'inspecteur.
- Ajouter un opérateur *Add* de Math (click droit dans le *VisualScript*).
- Relier les noeuds *Get count*, *Set count* et *Add*.
- Sélectionner *Label* dans le graphe de scène et utiliser l'inspecteur pour ajouter un noeud *Set text* grâce à un glisser-déplacer de la propriété *Text*.
- Ajouter un noeud *Get count* pour redéfinir le texte de Label avec la valeur de *count*.
- Un noeud de conversion d'int en string est automatiquement créé.

Le script est terminé et devrait ressembler à celui de la fig. 9.3.

9.2.2 Incrémenter un label avec une fonction prédéfinie

Cette fois-ci, lors du *click*, la valeur du label est incrémentée avec une fonction prédéfinie *Add count*. Il faut reproduire une scène dont l'arborescence est identique à l'exemple précédent. De même, ajouter un *VisualScript* au

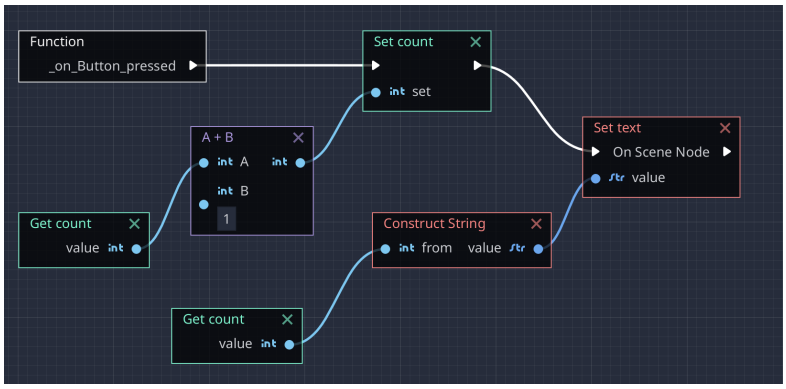


FIGURE 9.3 – Notre premier script *VisualScript*

nœud *Main*. Les fonctions et les variables sont également identiques à ceux de notre premiuer script. Pour définir l'interaction entre le bouton et le label, suivre la procédure ci-dessous pour obtenir le graphe présenté en fig.9.4 :

- Définir un nœud *_on_Button_pressed*
- Définir une variable *count* de type *int*, exportable et initialisée à 0.
- Ajouter un noeud Get de count
- Glisser-déplacer *count* depuis l'inspecteur crée une fonction *Set count* qu'il est possible de réassigner à la fonction Add en modifiant le champ *Assign Op*.
- Ajouter un noeud *Set text* pour le label.
- Relier les noeuds et exécuter

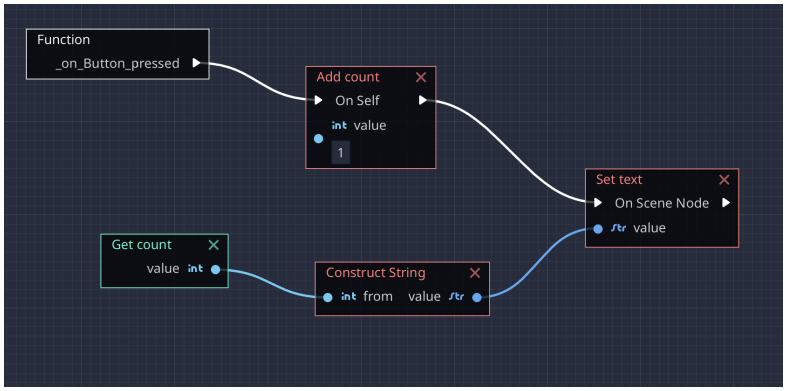


FIGURE 9.4 – Une autre version de l'interaction bouton-label

9.2.3 Afficher les secondes qui passent dans le terminal

Au fil des secondes, une valeur float correspondant aux secondes écoulées est affichée dans un terminal. La scène n'est composée que d'un seul nœud *Control* qu'on appellera *Main*, auquel on associe un script *Main.vs* Voici la liste des opérations à réaliser pour obtenir le graphe représenté en fig. 9.5 :

— Ajouter la fonction prédéfinie *Process(float)* (click droit dans le script).
— Définir une variable *seconds* de type *float*, exportable et initialisée à 0.
— Ajouter la valeur de *delta* à *seconds* avec l'opérateur *Add*.
— Ajouter la fonction *print* affichant la valeur de *seconds* obtenue grâce à un *Get*.
— Relier correctement les nœuds et exécuter.

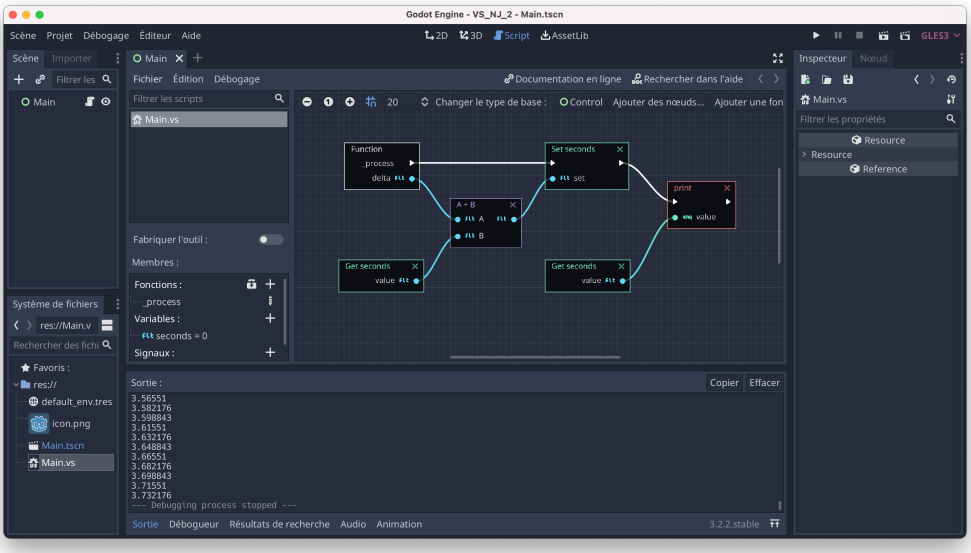


FIGURE 9.5 – Affichage du temps qui passe

Sur la fig. 9.5, on peut observer l'affichage de ce script, on constate que les dixièmes et centièmes de secondes sont également affichés. Pour afficher les secondes sans les dixièmes et les centièmes, on peut utiliser un *Timer* se déclenchant toutes les secondes pour incrémenter une variable *seconds* de type *int*. À partir d'une scène n'ayant qu'un nœud *Main* de type *Control* avec script *Main.vs* attaché, voilà la marche à suivre :

— Ajouter un nœud fils de type *Timer* à *Main*. Cocher *Autostart* dans l'inspecteur. Il est possible de modifier sa fréquence avec le champ *Wait Time*.

— Attacher un signal *timeout* de *Timer* vers le script *Main.vs* créant une fonction *_on_Timer_timeout*
— Définir une variable *seconds* de type *int*, exportable et initialisée à 0.
— Ajouter les fonctions *print*, *Add*, *Get* et *Set* de *seconds*.
— Ajouter la fonction *print* affichant la valeur de *seconds* obtenue grâce à un *Get*.
— Relier correctement les nœuds et exécuter (cf. fig 9.6).

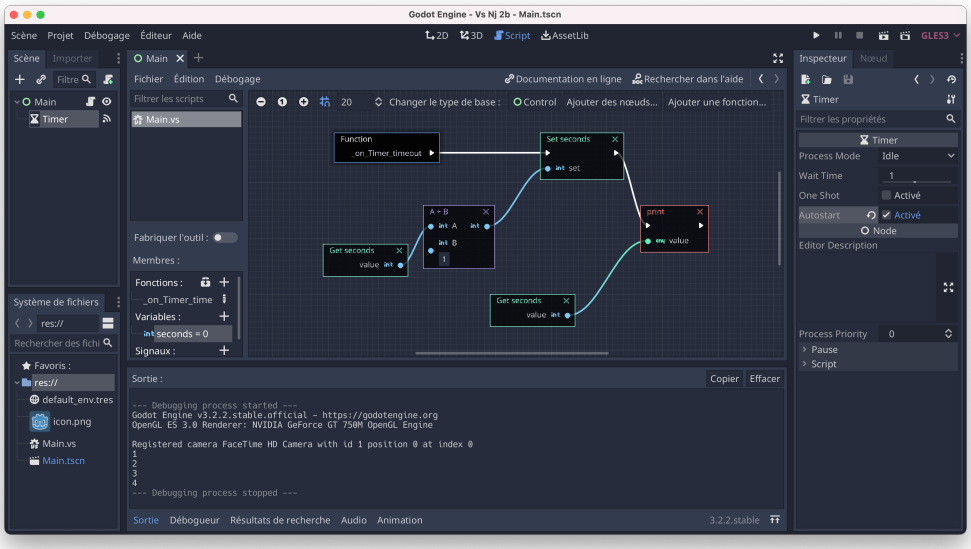


FIGURE 9.6 – Affichage du temps qui passe sans les dixièmes et centièmes

9.2.4 Conditionner un click par une valeur

La scène a un nœud racine de type *Control* appelé *Main* avec un script *Main.vs* attaché. *Main* possèdent deux nœuds fils de type *Label* (*label1* et *label2*) et deux de type *Button* (*button1* et *button2*). À chaque click sur *button1*, on incrémente *label1*. Cliquer sur *button2* incrémente la valeur de *label2* et soustrait 10 à la valeur de *label1* ; si la valeur de *label1* est inférieure à 10, cliquer sur *button2* ne fait rien et les valeurs de *label1* et *label2* restent inchangées. Marche à suivre :

— Ajouter à la scène le nœud *Main* et ses 4 nœuds fils *button1*, *button2*, *label1*, *label2*.
— Ecrire *count click* sur *button1* et 0 dans le champ *Text* de *label1*.
— Ecrire *count 10click* sur *button2* et 0 dans le champ *Text* de *label2*.

— Créer les fonctions *_on_button1_pressed* et *_on_button2_pressed*, en connectant les signaux des boutons dans l'inspecteur.
— Définir les variables *count1* et *count2* de type *int*, exportables et initialisées à 0.
— Ajouter un opérateur *Add* et un opérateur *CompareGreater*.
— Ajouter les fonctions *Get* et *Set* de *count1* et *count2*.
— Conditionner la mise à jour de *count2* par une valeur minimale de 10 *count1*.
— Clicker sur *button2* décrémente *count1* de 10 et incrémente *count2* et *count1*, décrémenter est réalisé avec la fonction *Subtract*.
— Ajouter les mises à jour des labels avec les valeurs de *count1* et *count2*.
— Relier correctement les nœuds et exécuter (cf. fig 9.7).

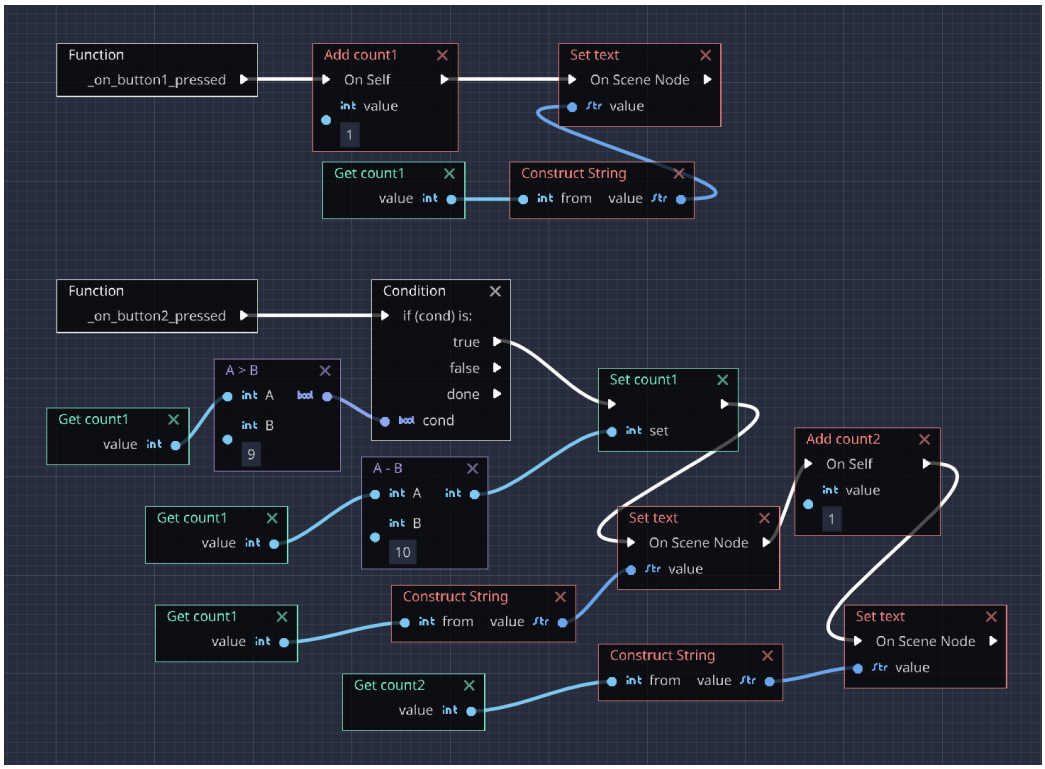


FIGURE 9.7 – Click conditionné

Chapitre 10

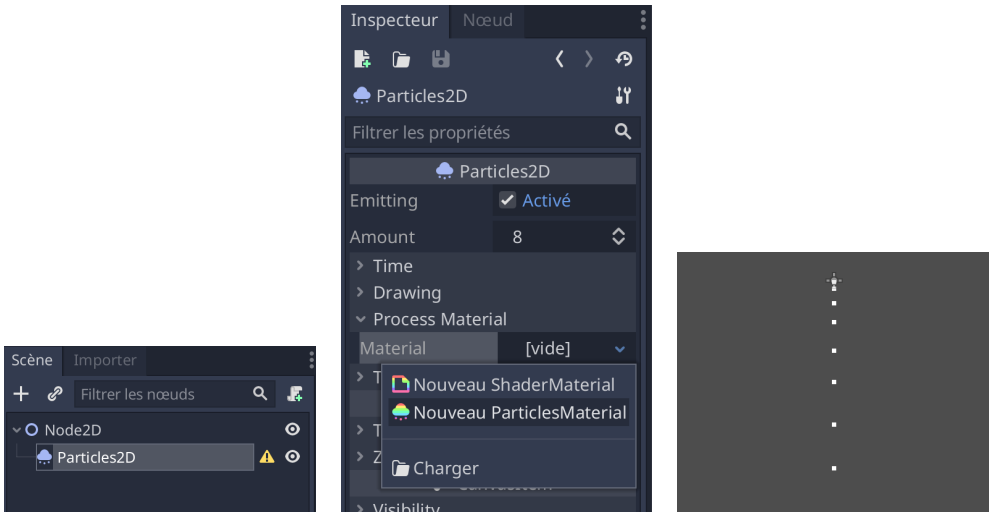
Les particules 2D

Un système de particules permet de simuler des effets physiques complexes tels que le brouillard, la fumée, le feu, les étincelles, l'écoulement de l'eau .... Le principe est simple : une particule est émise à intervalle de temps régulier et avec une durée de vie fixe. Durant leur vie, les particules ont toutes le même comportement de base. Ce qui va rendre chaque particule différente, est le caractère aléatoire de ses paramètres (position initiale, rotation ...).

10.1 Les nœuds particules

Godot fournit deux types de nœuds différents pour représenter les particules 2D : *Particles2D* and *CPUParticles2D*. *Particles2D* est plus performant et utilise le GPU pour calculer les effets des particules mais son utilisation nécessite de disposer d'OpenGLES 3. Si la plateforme cible du projet ne dispose que d'OpenGLES 2, il y a les *CPUParticles2D*, qui tournent sur le CPU et qui offrent presqu'autant de fonctionnalités que la version GPU, mais avec des performances moindres. Les *Particles2D* se configurent au moyen d'un *ParticlesMaterial* ou d'un *ShaderMaterial* pour utiliser un *shader* personnalisé. Les *CPUParticles2D* sont configurables via les propriétés du nœud. Il est possible de convertir un *Particles2D* en *CPUParticles2D* en cas de besoin. Dans ce chapitre nous allons voir comment configurer des particules en utilisant le nœud *Particle2D*. Il faut donc ajouter un nœud de ce type dans une scène (cf. fig. 10.1a).

On remarque tout de suite, qu'il y a un avertissement à côté du nœud tout juste créé. Celui-ci indique que ce type de nœud doit avoir un *ProcessMaterial* pour fonctionner. Il faut donc aller lui en ajouter un. Il faut sélectionner notre nœud *Particles2D* dans le graphe de scène, puis dans l'inspecteur dérouler *ProcessMaterial*, et créer un nouveau *ParticlesMaterial* (cf. fig. 10.1b). Il



(a) Ajout d'un nœud *Particles2D* dans la scène

(b) Création d'un *ParticlesMaterial*

(c) Résultat

FIGURE 10.1 – Création d'un nœud *Particles2D*

est également possible d'en charger un existant si nous voulons en réutiliser un, mais nous pouvons aussi créer ou charger un *ShaderMaterial* si nous souhaitons utiliser un shader personnalisé. Une fois le *material* créé, dans la fenêtre 2D, des particules (points blancs pour l'instant) sont émises vers le bas (cf. fig. 10.1c).

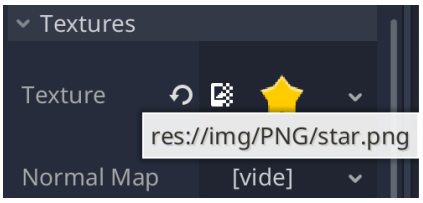
10.2 Configurer le système de particules

Le système de particules utilise une texture unique ou animée via *sprite-sheet*. Elle est définie via la propriété de texture appropriée dans l'inspecteur (cf. fig. 10.2a).

Dans l'inspecteur, il est possible de choisir combien de particules peuvent être en vie simultanément (cf. fig. 10.3a). Ces particules ont une durée de vie, qui correspond au nombre de secondes pendant lesquelles elles vont rester vivante après leur émission. Lorsque ce délai est passé, la particule disparaît, une nouvelle est créée pour la remplacer. Pour modifier cette durée de vie , il suffit de changer la valeur de la propriété *Lifetime*.

Dans l'inspercteur, vous pouvez également configurer les valeurs des propriétés liées au temps, en déroulant la catégorie *Time* (cf. fig. 10.3b) :

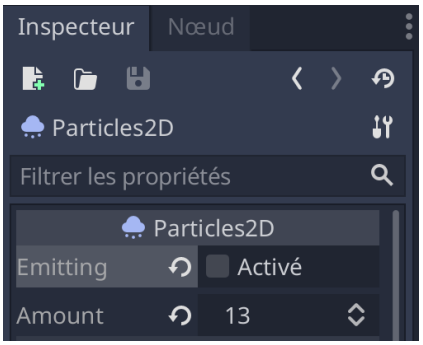
— La propriété *One Shot*, si elle est active, signifie que les particules seront émises une seule fois.



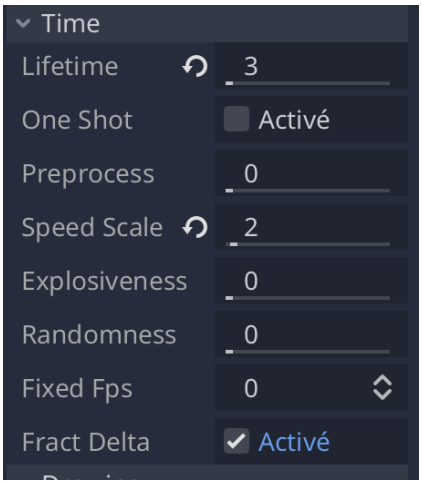
(a) Ajout d'une image de texture

(b) Résultat immédiat dans la fenêtre 2D

FIGURE 10.2 – Ajout d'une texture à notre *material*



(a) Choix du nombre de particules



(b) Propriétés temporelles

FIGURE 10.3 – Ajout d'une texture à notre *material*

— La propriété *Preprocess* permet de définir combien de secondes s'écoulent avant que la scène soit effectivement active et dessinée, autreùent dit avant que le système de particules commence à émettre. En effet, lorsque le nœud *Particles2D* active le système de particules, il commence avec 0 particule émise. Cela peut-être déplorable pour des scènes avec des torches ou du brouillard par exemple, car on veut que les torches soient allumées à l'instant même où on entre dans la scène.
— Il est possible de modifier la vitesse de déplacement de nos particules : augmenter la valeur de la propriété *Speed Scale* provoquera une ac-

célération des particules, à l'inverse diminuer sa valeur ralentira les particules.

— Par défaut, la valeur de la propriété *Explosiveness* est 0, ce qui correspond à une émission régulière et constante des particules. Par exemple, pour un nombre de particules fixé à 10, ces particules ayant une durée de vie de 1 seconde, alors une particule sera émise toutes les $0,1$ secondes. Le paramètre *Explosiveness* peut prendre des valeurs comprises entre 0 et 1, 0 signifiant donc une émisssion des particules à intervalles réguliers et 1 une émission simultanée de toutes les particules. Les valeurs intermédiaires permettant de répartir les émissions par à-coups.

— La propriété *Randomness* permet d'ajouter de l'aléatoire à toutes les propriétés physiques des particules selon la formule :
$initial\_value = param\_value + param\_value * randomness$
La valeur de *randomness* peut être comprise entre 0 et 1.

Dans la section *Drawing* (cf. fig. 10.4), se trouve la propriété *Local Coords*. Si elle est acitviée (ce qui est le cas par défaut), cela signifie que les particules sont émises dans l'espace relatif au Nœud. Lorsque celui-ci se déplace, les particules émises font de même. Si la propriété est désactivée, les particules sont émises dans l'espace global, si le nœud se déplace, les particules déjà émises ne sont pas impactées.

FIGURE 10.4 – Section *Drawing*

Dans cette section, il est également possible de choisir le critère déterminant l'ordre de dessin des particules grâce à la propriété *Draw Order*. Par défaut, les particules sont dessinées en fonction de leur ordre d'émission (*index*). On peut choisir de les dessiner dans l'ordre de leur durée de vie restante (*Lifetime*).

Une autre section importante à connaître est *ProcessMaterial* (cf. fig. 10.5). Il est possible de choisir la direction dans laquelle les particules sont émises grâce à la propriété *Direction*. Par défaut, le vecteur de direction est $(1, 0, 0)$,

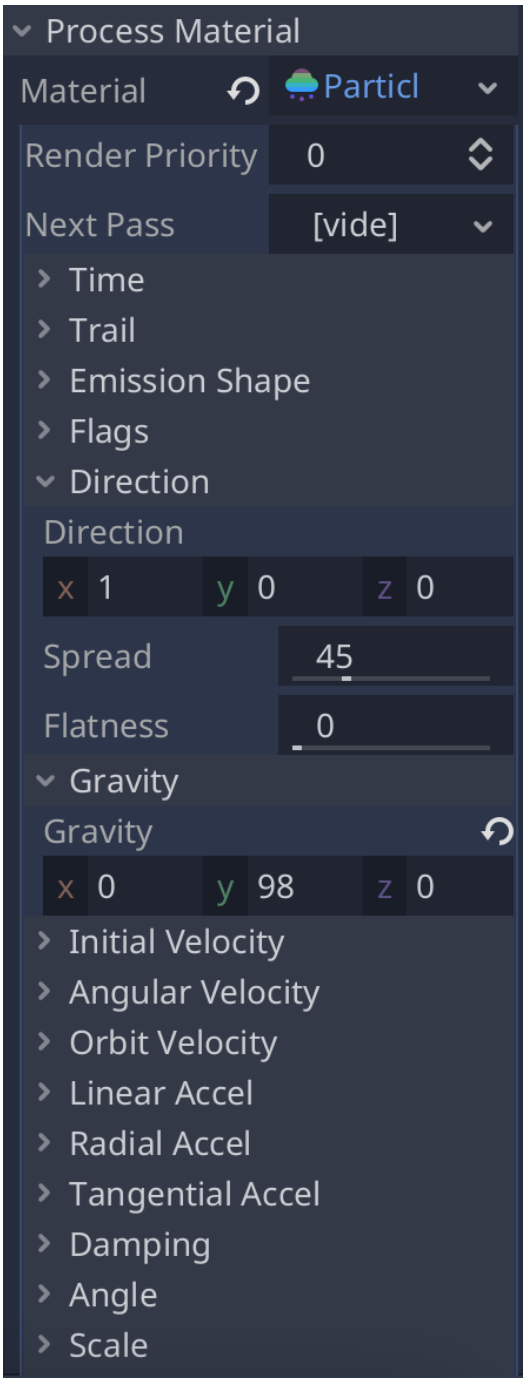


FIGURE 10.5 – Section *ProcessMaterial*

les particules sont donc émises vers la droite mais à cause de la gravité, cela ne se voit pas, elles tombent directement vers le bas (cf. fig. 10.6a). Pour

que la direction initiable soit apréciable, il faut que la vitesse initiale des particules soit supérieure à 0. Pour cela, il faut changer la valeur par défaut du paramètre *Initial Velocity* et la fixer à une valeur considérablement plus grande, 100 par exemple (cf. fig. 10.6b). Cette vitesse est exprimée en nombre de pixels par secondes. Elle peut être affectée au cours du temps en fonction des forces auxquelles est soumise la particule (gravité, frottements ...). Il est maintenant visible que les particules sont émises vers la droite mais leur direction initiale ne semble ne pas être constante (cf. fig. 10.6c). Cela est dû au paramètre *Spread* dont la valeur par défaut est 45°. Ce paramètre correspond à l'angle en degrés ajouté aléatoirement dans l'une ou l'autre direction à la direction initiale. Une propagation de 180° aura pour effet une émission dans toutes les directions ($+/-180°$). Si par contre cet angle est fixé à 0, les particules sont toutes émises exactement dans la même direction.

(a) Particules émises vers la droite sans vitesse initiale

(b) Particules émises vers la droite avec une vitesse initiale de 100 pixels par secondes

(c) Particules émises vers la droite avec une vitesse initiale de 100 pixels par secondes et un angle de 0°

FIGURE 10.6

Il est également possible d'agir sur la vitesse angulaire des particules, c'est à dire à quelle vitesse tournent-elle sur elles-mêmes. Cela se fait via le paramètre *Angular Velocity* (cf. fig. 10.7a), se valeur est exprimée en degrés par seconde. Le paramètre *Orbit Velocity* permet de déterminer la vitesse de rotation des particules autour de leur centre (cf. fig. 10.7b).

10.3 Paramétrer avec un script

Il se peut que l'on souhaite que des particules soient émises que lors d'une action précise du joueur, c'est le cas pour l'ouverture d'un coffre par exemple.

(a) Vitesse angulaire (b) Orbite

FIGURE 10.7 – Rotations

Cela se fait via un script. Voyons comment faire sur un exemple simple : nous allons créer une zone cliquable, qui enverra des particules en seule fois au moment du click. Pour cela, nous devons tout d'abord ajouter à notre scène un nœud de type *Area2D* et lui ajouter un nœud fils de type *CollisionShape2D* afin de gérér les collisions ou le rendre cliquable (cf. fig. 10.8).

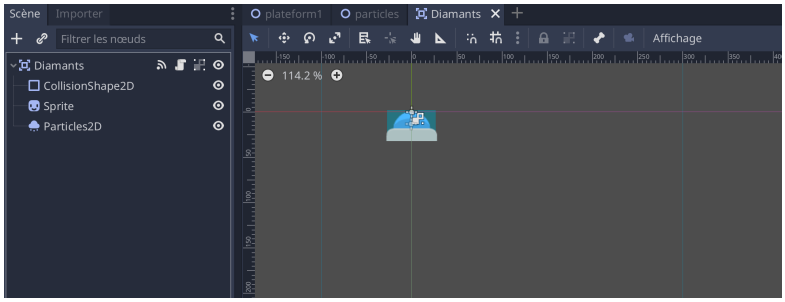


FIGURE 10.8 – Scène pour la gestion d'actions sur un système de particules

En imaginant que l'ont veur pouvroir déclencher notre émission de particules lors du passage d'un personnage, en *visual scripting*, cela est très simple (cf. fig. 10.9). À noter qu'il faut avoir connecté le signal *body_ entered* de l'*Area2D*.

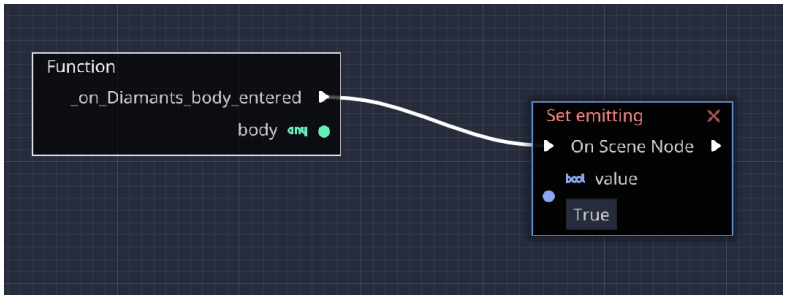


FIGURE 10.9 – Script permettant l'émission de particules lors du contact avec un personnage

Chapitre 11

L'interface graphique utilisateur

Comme tout le reste dans Godot, l'interface utilisateur est construite à base de nœuds, en particulier de nœuds de contrôle (*Control Node*). Il existe de nombreux types de contrôles qui sont utiles pour créer des types spécifiques d'interface graphique. Pour plus de simplicité, nous pouvons les séparer en deux groupes : le contenu et la mise en page. Pour créer des interfaces utilisateur, vous aurez besoin d'un mélange d'éléments d'interface utilisateur qui héritent des nœuds *Control* et de *Container*. Les nœuds *Control* de contenu sont majoritairement des *Button*, des *Label*, des *TextEdit* ou des nœuds qui en héritent. Du côté des nœuds *Conrol* destinés à la mise en page, on nommera les *BoxContainer*, les *MarginContainer*, les *ScrollContainer*, les *TabContainer* ou encore les *Popup*. Pour connaitre les spécificités de ces éléments, je vous invite à lire leur page de documentation sur le site de Godot.

11.1 Les nœuds *Control*

Tous les nœuds de l'interface utilisateur héritent de *Control*. Les ancres et les marges d'un nœud *Control* adaptent leur taille et position par rapport à celles du nœud parent. Le nœud *Control* comporte une boite englobante qui définit ses dimensions, une position d'ancrage par rapport à son parent ou à la fenêtre actuelle, et des marges qui représentent un décalage par rapport à l'ancrage. Les marges sont mises à jour automatiquement lorsque le nœud, l'un de ses parents ou la taille de l'écran changent.

11.1.1 Événemens d'entrée

Godot envoie d'abord les événements d'entrée au nœud racine de la scène, en appelant *Node._input*. *Node._input* transmet l'événement dans l'arbores-

cence des nœuds jusqu'aux nœuds sous le curseur de la souris ou concerné par le focus du clavier. Pour ce faire, il appelle *MainLoop._input_event*. Appelez *accept_event* afin qu'aucun autre nœud ne reçoive l'événement. Une fois que vous acceptez une entrée, elle est gérée afin que *Node._unhandled_input* ne la traite pas. Seuls les nœuds *Control* peuvent recevoir le focus clavier. Seul le nœud ciblé recevra les événements de clavier. Pour obtenir le focus, appelez *grabi_focus*. Un nœud *Control* perd le focus lorsqu'un autre nœud le saisit, ou si le nœud est masqué en focus.

11.2 Cas pratique

Je vous invite à réaliser la suite de tutoriel situé ici, suivi de celui-ci.