Introduction aux moteurs de jeux avec Godot

Benjamin DUPONT
Licence Informatique - Université Paris 8
2022-2023

Notes inspirées des cours rédigés par Sylvia Chalençon et Adrien Revault d'Allones, et Nicolas Jouandeau.

Le cours *Introduction aux moteurs de jeux* fait partie de la mineure *Conception et programmation de jeux vidéo* au premier semestre de la licence Informatique & Vidéoludisme. L'objectif de ce cours est de présenter le fonctionnement d'un moteur de jeu, en particulier Godot, sa prise en main et ses premières fonctionnalités.

Table des matières

Chapitre 1

Moteurs de Jeux

1.1. Qu'est-ce qu'un moteur de jeu ?

La notion de moteur de jeu est née dans les années 90, introduite par les jeux de tir à la première personne. L'architecture logicielle de ces jeux séparait distinctement les composants moteurs du jeu et les ressources numériques propres au jeu, à savoir les graphismes, les sons, la musique, les scènes et les règles du déroulement. Cette organisation a permis la réutilisation des composants moteurs pour développer de nouveaux jeux avec d'autres personnages, d'autres armes, de nouvelles cartes etc., permettant ainsi 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, notamment:

1. un moteur 2D, permettant d'afficher à l'écran un environnement graphique bidimensionnel (ou "planaire") 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 taille sur laquelle le personnage se déplace. Dans le second cas, un niveau est créé par assemblage d'une multitude de petites images appelées tuiles (ou *tiles* en anglais), pour former une grande image. L'ensemble des tuiles constitue ce qu'on appelle une *tilemap*. L'affichage d'un niveau se fait en vue de côté, en vie 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. un moteur 3D, permettant de créer 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 ce polygone sont exprimées en 3 dimensions. En fonction du type de rendu effectué, le moteur va permettre d'obtenir à partir de ces coordonnées 3D, une matrice 2D qui correspond à l'image affichée à l'écran. Pour dessiner des scènes 2D ou 3D, le développeur peut utiliser des bibliothèques de rendu 3D (par exemple OpenGL ou Direct3D) mais cela exige de maîtriser les modèles mathématiques sous-jacents, et les opérations réalisées par une carte graphique nécessaires au rendu d'un lot de données. Utiliser un moteur de jeu permet d'obtenir le rendu désiré plus rapidement et sans ces contraintes.

3. un moteur physique, qui calcule la trajectoire des objets, leurs interactions, les forces subies (gravité, frottements, etc.). Il calcule également la déformation des objets "mous",

comme par exemple des tissus ou 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.

4. un moteur d'animation, qui permet de réaliser des animations aussi bien 2D que 3D. L'animation peut se faire en image par image, chaque image étant redessinée au fil du temps (souvent on utilise des images *bitmap*, ou encore en animant des formes géométriques (cercles, polygones); on parle d'animation vectorielle.

5. un moteur d'interface graphique, permettant un rendu du jeu dans un environnement graphique manipulable par le joueur, le plus souvent avec la souris.

6. un moteur son, effectuant 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 cet espace. Des modifications sur les sources peuvent être ajoutées, par exemple de l'écho.

7. une gestion des entrées et sorties, qui traite les échanges d'information entre le jeu et les périphériques qui lui sont associés. Classiquement, le moteur de jeu détecte les actions du joueur sur le clavier (appui de touches), la souris (état des boutons, mouvement, position), sur un joystick, et éventuellement d'autres matériels selon les moteurs. C'est ici qu'est assurée la lecture des données du jeu, ainsi que les sauvegardes des données utilisateurs.

8. un gestionnaire de ressources,

9. un gestionnaire de scènes.

L'idée d'un moteur de jeu est donc de concentrer les efforts de développement d'un jeu sur le contenu et le déroulement du jeu, plutôt qu'à la résolution de problèmes purement informatiques déjà résolus précédemment. Le concepteur du jeu utilise alors des scripts, permettant par exemple de programmer le comportement des personnages non jouables (PNJ) ou encore de modifier le gameplay. Souvent, ces scripts sont réalisés à l'aide d'un langage propre au moteur de jeu choisi, mais nous verrons qu'il existe d'autres possibilités.

1.2. QUELQUES MOTEURS DE JEUX

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

1.2.1. CryEngine. *CryEngine*, développé par Crytek, est plutôt orienté jeux de tir à la première personne ou FPS (First Person Shooter). Le premier jeu qui l'utilise est *Far Cry*, d'où son nom. Les langages utilisées 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 de jeu utilisant CryEngine au delà des premiers 5000$ annuels.

1.2.2. Unreal Engine. *Unreal Engine* est un moteur de jeu vidéo propriétaire développé par *Epic Games*. 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 communauté des concepteurs de jeux-vidéos et des jeux célèbres reposent sur *Unreal Engine*, par exemple *Fortnite* ou encore *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.

1.2.3. Unity. *Unity* est l'un des moteurs de jeux 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 (par exemple pas d'export de Linux vers Windows). Il existe quatre niveaux de licences :

- *personal*, gratuit dans la limite de revenus ou financements inférieurs à 100000$ au cours des 12 derniers mois;
- *plus*, à 369 € par an et par poste si vos revenus ou financements sont inférieurs à 200000$ sur les 12 derniers mois;
- *pro*, à 1656 € par an et par poste si vos revenus ou financements sont supérieurs à 200000$ sur les 12 derniers mois;
- *entreprise*, à 183 € par an et par poste au delà de 20 postes et pour des revenus ou financements sont supérieurs à 200000$ sur les 12 derniers mois.

Les scripts sont rédigeables uniquement en C#.

1.3. Liens utiles

Vous trouverez ci-dessous quelques liens utiles pour trouver des images, des textures ou des décors bien dimensionnés:

1. Spriters resource
2. Opengameart
3. Open pixel project
4. Kenney

Chapitre 2

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 certains nombres de plate-formes, notamment Linux, Mac OS et Windows mais aussi vers les plateformes 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 pour cela 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.

2.1. 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 outiles pour dessiner des lignes, des polygones, des *sprite* (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 générer en CPU ou GPU des particules et d'utiliser des *shaders* personnalisés pour leur rendu.

2.2. Moteur 3D

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

2.3. 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 ressources, les *sprites*, les éléments d'interface utilisateur, les particules.

2.4. 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 et instanciable. Il est possible d'imbriquer une scène créée au préalable, en tant que nœud d'une autre scène. Par exemple, on peut avoir une scène modélisant une balle et son mouvement soumis aux actions du joueurs, que l'on peut réutiliser dans différents jeux. Les nœuds peuvent également être des *sprites*, des formes servant de masques de collision, des sources lumineuses, des sources sonores, des objets physiques...

2.5. 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 pour créer ces scripts, notamment le C++ ou le C# à travers *Mono* que nous n'utiliserons pas ici. Dans ce cours, nous nous concentrerons sur deux langages de scripts en Godot:

- le *VisualScript*, langage graphique permettant de coder visuellement sans écrire de code, presque exclusivement à la souris;
- le *GDScript*, le langage de script conçu spécialement pour *Godot*. Sa syntaxe est très proche du *Python*.

2.6. L'éditeur Godot

Passons maintenant aux choses pratiques. La première étape est de télécharger *Godot*`https://godotengine.org/download/linux` pour Linux et `https://godotengine.org/download/windows` pour Windows (de préférence la version 3.x qui utilise encore du *VisualScript*, devenant obsolète à partir de la version 4).

Lorsqu'on lance *Godot*, c'est le gestionnaire de projet qui s'ouvre. Dans celui-ci, il est possible 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 créer un nouveau projet, il faut un répertoire vide qui servira de répertoire de travail pour ce projet, pensez donc à avoir dans votre répertoire courant un dossier Godot dans lequel vous créerez un nouveau dossier pour chaque 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.

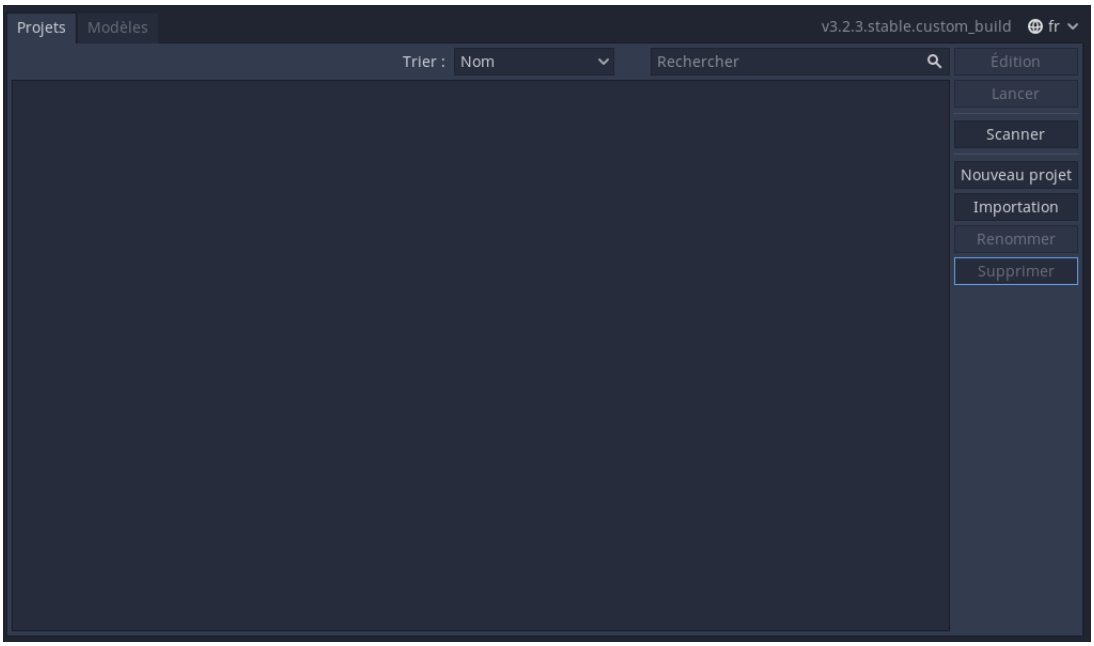


Figure 2.1: Gestionnaire de projets *Godot*

Créons alors un premier projet, que l'on nommera Test, accessible dans *home/Godot/test*. Très classiquement, les menus principaux se situent en haut à gauche de la fenêtre de l'éditeur :

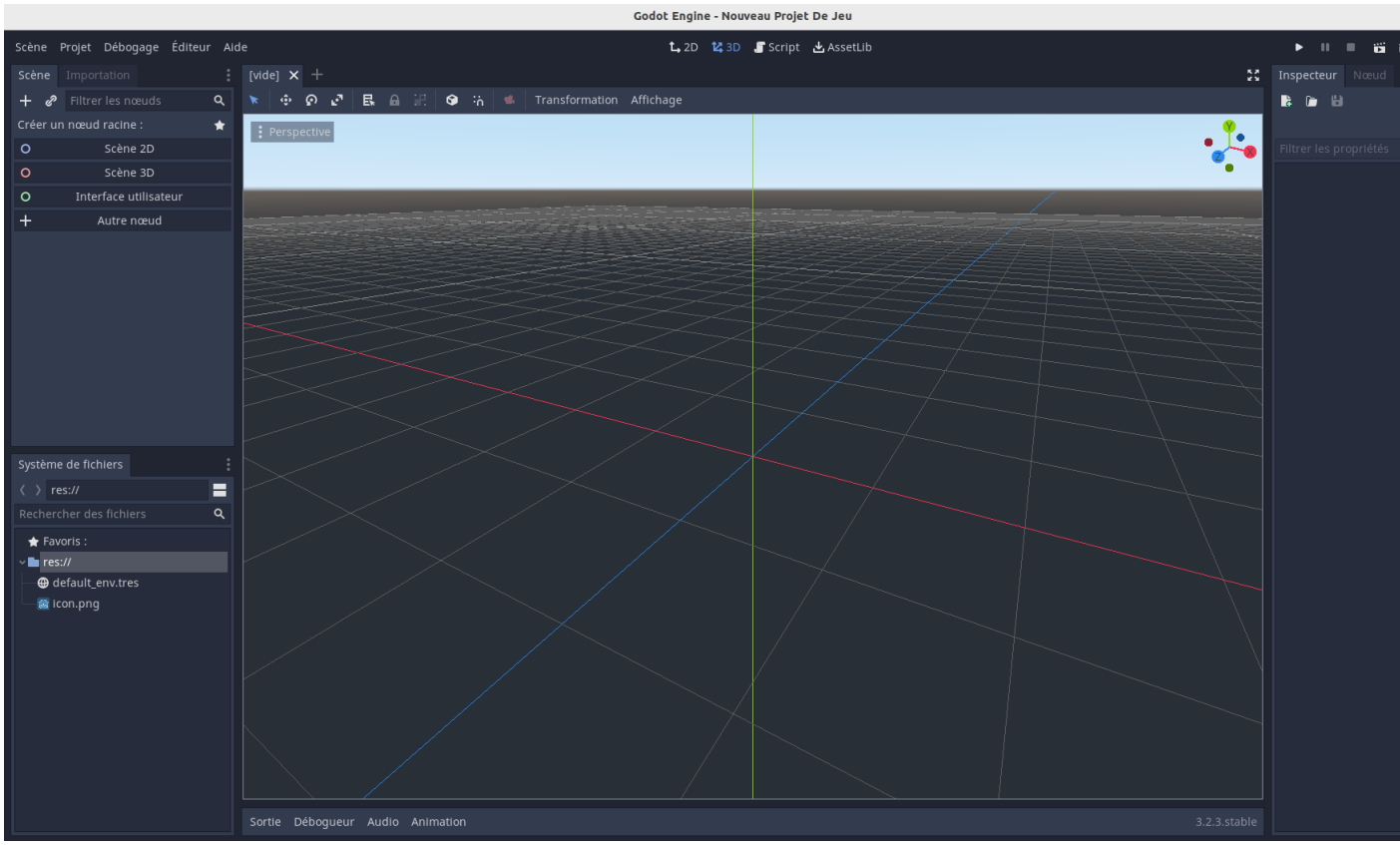


Figure 2.2: Éditeur *Godot*

En haut au centre, se trouvement 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 des 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 en cliquant sur *Documentation en ligne*.
- 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 projets**. 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 3

Scènes et nœuds

3.1. Les nœuds

Le nœud est l'élément fondamental de toute scène et donc de tout projet *Godot*. Il existe de nombreux types de nœuds dédiés à l'affichage de différents objets, au son, à l'animation, etc. À sa création, un nœud a, quelque soit son type, des propriétés éditables. On peut lui attacher une fonction de *callback* qui sera appelée à chaque *frame*. Par ailleurs, un nœud peut être ajouté comme "fils" d'un autre nœud et réciproquement un nœud donné 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 unique racine. 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 principale, afin d'être lancée au démarrage du jeu.

3.2. Création d'une première scène : Hello world !

Nous allons maintenant apprendre à réaliser une première scène, un classique lorsque l'on commence à manipuler un langage : l'affichage d'un message "Hello world !".

3.2.1. Création de la scène. Dans notre projet vide, le **graphe de scène** en haut à gauche est vide, et propose la création d'un nœud racine en appuyant soit sur le bouton + en haut à gauche du graphe de scène, soit sur le bouton *Autre nœud* dans la liste des types de nœuds racine, voir Figure 3.3. Il est à noter que, par la suite, lorsque la scène n'est pas vide et équipée d'un nœud principal, il faut utiliser le bouton *Ajouter un nœud enfant*, représenté par le symbole + en haut à gauche.

Nous allons créer un nœud *Scène 2D*, puis nous lui ajouterons un nœud fils de type *Label*, permettant l'affichage d'un texte. Notons que pour créer un nœud *Scène 2D*, nous aurions pu directement cliquer sur le bouton *Scène 2D* du graphe de scène proposé au départ.

À la création du premier nœud, l'éditeur passe alors en 2D. À l'ajout du *Label*, son rectangle est affiché dans le coin supérieur gauche de l'espace de travail 2D (visible par une délimitation bleue dans le plan), le nœud est ajouté au graphe de scène et ses propriétés sont accessibles dans l'inspecteur sur la droite. Notez que la barre "Rechercher" du menu *Créer un nouveau Node* nous permet de trouver le nœud recherché, même si il n'apparaît pas directement dans la liste.

Figure 3.1: Graphe de scène

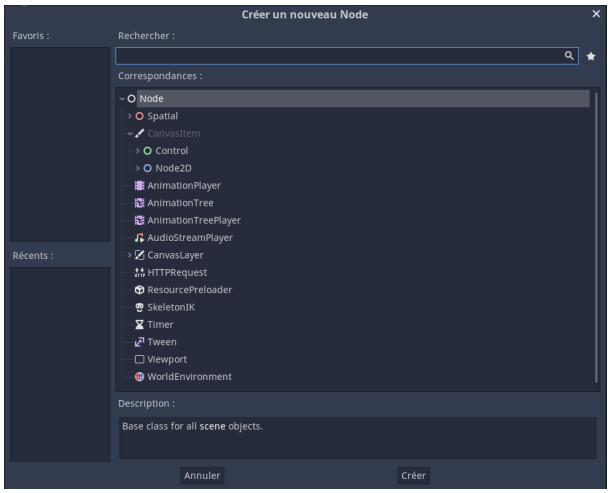
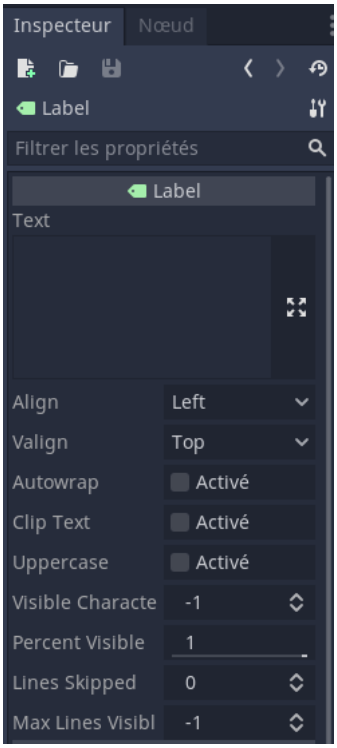


Figure 3.2: Création du nœud racine

Figure 3.3: Évolution du graphe de scène

Voici alors ce qui apparaît dans l'inspecteur sur la droite pour le moment. On peut alors modifier la propriété *Text* pour y écrire ce que l'on souhaite : Hello world ! Les propriétés *align* (position horizontale) et *Valign* (position verticale) nous permettent alors de positionner le texte dans le rectangle de la scène.



Un bon réflexe est alors de sauvegarder notre scène principale ! Pour ceci, cliquez sur *Scène* dans le menu en haut à gauche, puis *Enregistrer la scène sous...*, créer un dossier Scènes à l'aide du bouton *Créer un dossier* en haut à droite, donner un nom à notre scène (par exemple *helloworld.tscn*, puis enregistrer.

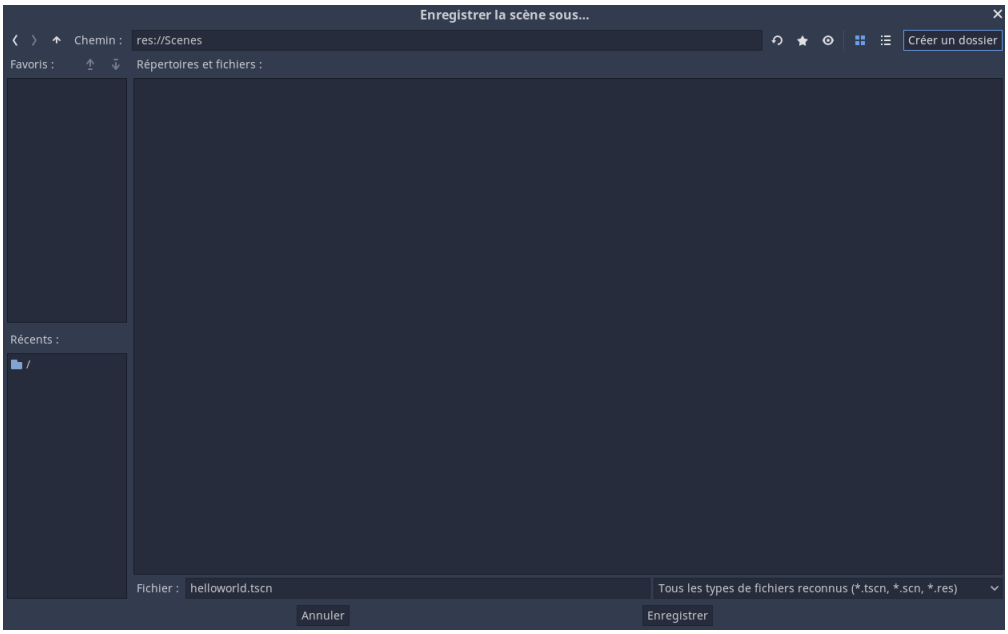


Figure 3.4: Enregistrement de la scène

3.2.2. 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 (ou en créer une...). Par exemple, on peut utiliser la police *Minecraft*, disponible sur https://www.dafont.com/fr/minecraft.font. On peut télécharger cette police, extraire le .zip obtenu afin d'obtenir un fichier .ttf. On peut ensuite faire glisser ce fichier dans le gestionnaire de ressources en bas à droite de Godot afin d'avoir cette police utilisable dans la scène.

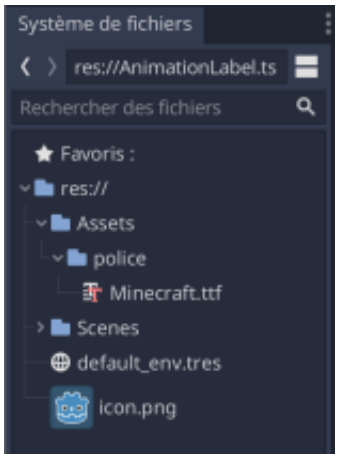


Figure 3.5: Ajout de la police au projet

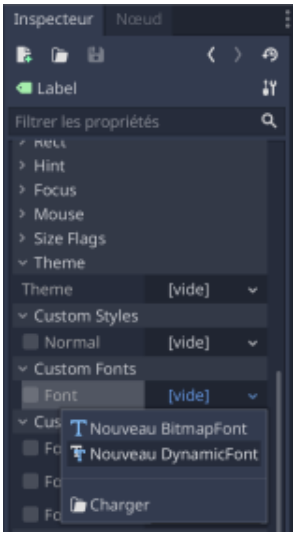


Figure 3.6: Création d'une nouvelle police pour le *Label*

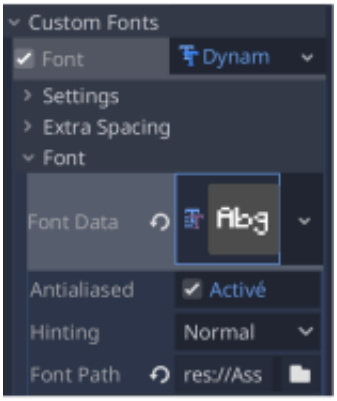


Figure 3.7: Sélection de la police

Pensez bien à structurer ce gestionnaire : c'est ici que nous ajouterons tous les fichiers que nous utiliserons dans la scène (police, images, sons, etc.) : créez donc des dossiers pour chaque type d'objet. Si vous utilisez beaucoup de fichiers, il est même préférable de créer un gros dossier *Assets* contenant tous les dossiers propres à chaque type d'objet.

Pour utiliser cette police, il faut maintenant créer un *Nouveau Dynamic Font* dans la catégorie *Custom Font* de l'inspecteur du *Label*, voir Figure 3.6. Puis dans la sous-propriété *Font*, il faut renseigner le champs *Font Data* avec la police voulue, par exemple en faisant glisser la police désirée depuis le gestionnaire de ressources jusque dans cette case, voir Figure 3.7.

Vous pouvez maintenant personnaliser le texte à votre convenance. Maintenant, nous allons essayer de jouer la scène à partir du bouton *Lancer la scène* de la barre supérieure (le bouton Play) :

Figure 3.8: Lancement de la scène

Pour que le projet puisse se lancer, il faut définir la scène comme scène principale. Il y a plusieurs moyens de se faire : le plus simple est d'effectuer un clic droit sur notre scène *helloworld.tscn* dans le gestionnaire de fichiers, et de sélectionner *Définir comme scène principale* dans le menu contextuel. Vous pouvez alors rappuyer sur *Lancer la scène*, et le projet se lance alors. Notez que pour l'instant, le texte apparaît dans le coin supérieur gauche de la scène, mais il est possible de déplacer le nœud *Node 2D* à l'aide du bouton déplacement (ou du raccourci W) afin de le centrer dans le rectangle de la scène.

3.2.3. Ajouter des animations. Dans *Godot*, il est possible d'animer tout élément disponible dans le graphe de scène. Les nœuds *Animation Player* permettent de créer des animations. L'*AnimationPlayer* est utilisé pour la lecture des données 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'enchaîner automatiquement.

Reprenons notre exemple précédent : nous voudrions maintenant ajouter un peu d'animation. Pour ce faire, il faut créer un nœud *AnimationPlayer* (voir Figure 3.9) enfant du nœud racine de notre scène, et dont le nœud *Label* sera lui-même un fils, voir Figure 3.10.

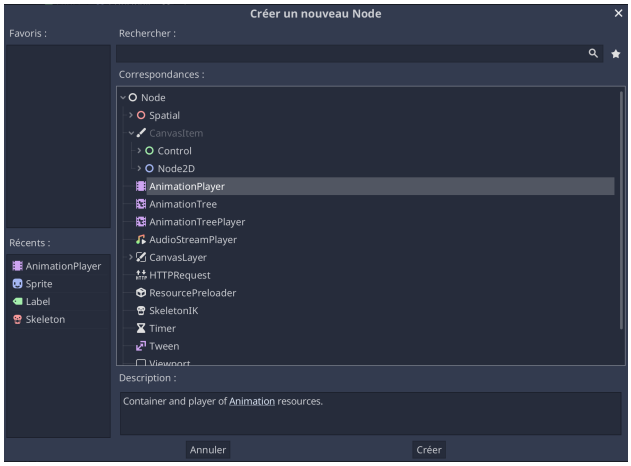


Figure 3.9: Création d'un *AnimationPlayer*

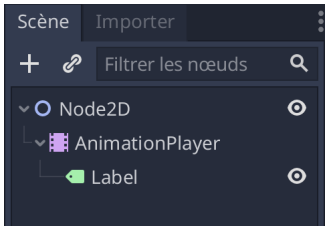


Figure 3.10: Nouveau graphe de scène

Après la création de l'*AnimationPlayer*, le nœud *Label* restera un fils du nœud principal *Node 2D*. Pour l'instancier en tant que fils du *AnimationPlayer*, il suffit de faire glisser le *Label* sur ce dernier avec la souris pour obtenir ce que l'on voit en Figure 3.10.

Pour créer une animation, il faut sélectionner le nœud *AnimationPlayer* dans notre scène et sélectionner l'onglet Animation dans la bandeau du bas, voir Figure 3.11.

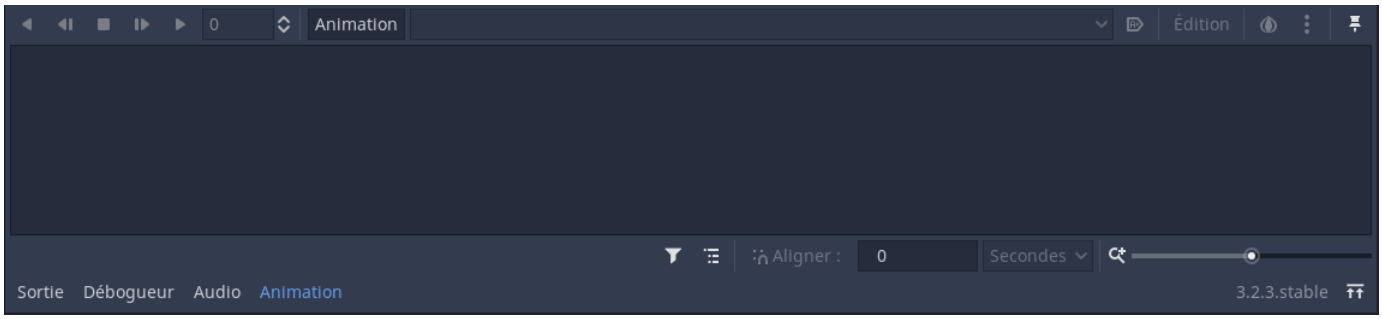


Figure 3.11: Éditeur d'animations

Dans ce panneau, il faut créer une nouvelle animation en cliquant sur *Animation* (en haut) 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 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 (en haut), 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, voir figure 3.12.

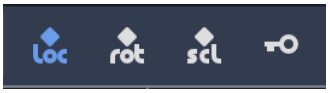


Figure 3.12: Création d'une clé d'animation

Les boutons correspondent respectivement à : position (location), rotation et échelle (scale). Une fois cette clé créée, une poste pour le *Label* apparaît dans le panneau Animation :

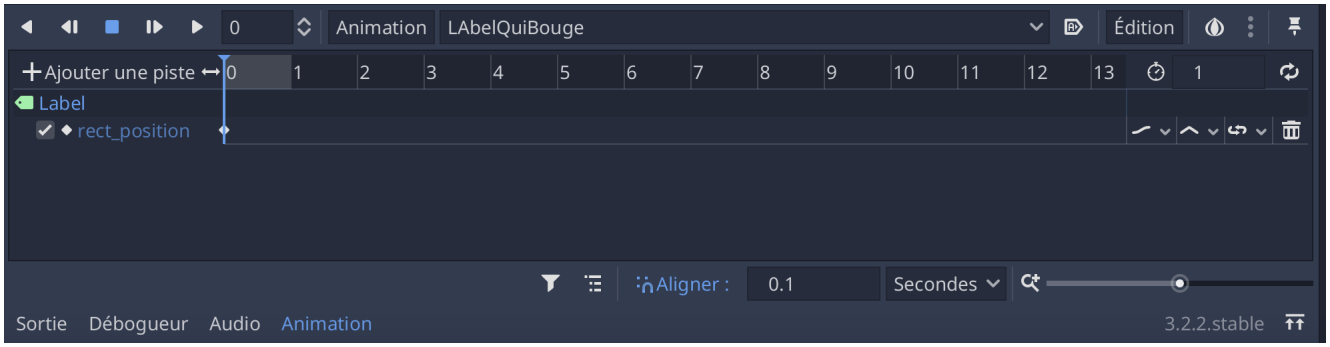


Figure 3.13: Gestion de l'animation

C'est à cet endroit qu'il est possible de gérer la durée de l'animation, d'ajouter d'autres clés d'animation pour le *Label*. Pour ajouter une nouvelle clé, il suffit de déplacer le curseur de temps sur la poste, 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*.

Exercice. Vous pouvez maintenant vous entraîner à animer votre *Label*. Essayez :

1. de le faire aller de la gauche vers la droite, puis de la droite vers la gauche;
2. de le faire tourner sur lui même;
3. de faire en sorte que le texte suive les contours de la zone, en restant parallèle au bord.

Chapitre 4

Gestion de scènes

4.1. Instancier des scènes

Dans le cas d'un projet comme celui du chapitre 3, on peut utiliser une scène unique mais pour des projets plus compliqués, leur graphe de scène et leur nombre de nœuds peuvent présenter une arborescence beaucoup plus compliquée, voir Figure 4.1.

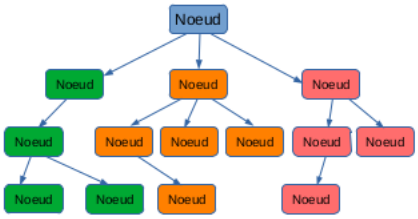


Figure 4.1: Scène plus complexe, sans instanciation

*Godot*autorise un nombre de scènes quelconque dans un projet, et nous avons vu précédemment qu'une scène 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 créée est appelée *packed scene*, et a pour extension de nom de fichier *.tscn*. Une scène sauvegardée et présente dans l'arborescence de fichiers du projet peut être instanciée dans une autre scène de celui-ci, comme si il s'agissait d'un nœud, voir Figure 4.2. Ainsi, pour l'exemple ci-dessus, dans lequel le nœud principal a 3 nœuds fils ayant eux-même des fils, on peut sauvegarder 3 scènes distinctes correspondant à chacun des fils, voir Figures 4.2, 4.3 et 4.4. Ceci simplifie alors grandement le graphe de scène, qui peut être représenté comme en Figure 4.5.

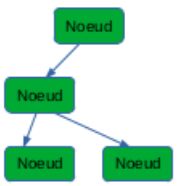


Figure 4.2: ScèneA.tsvn

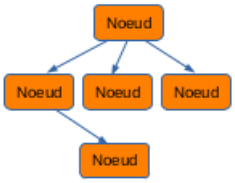


Figure 4.3: ScèneB.tsvn

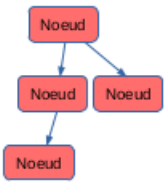


Figure 4.4: ScèneC.tsvn

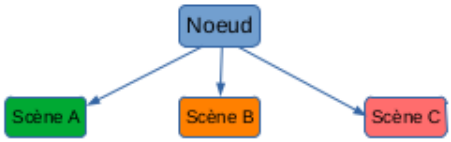
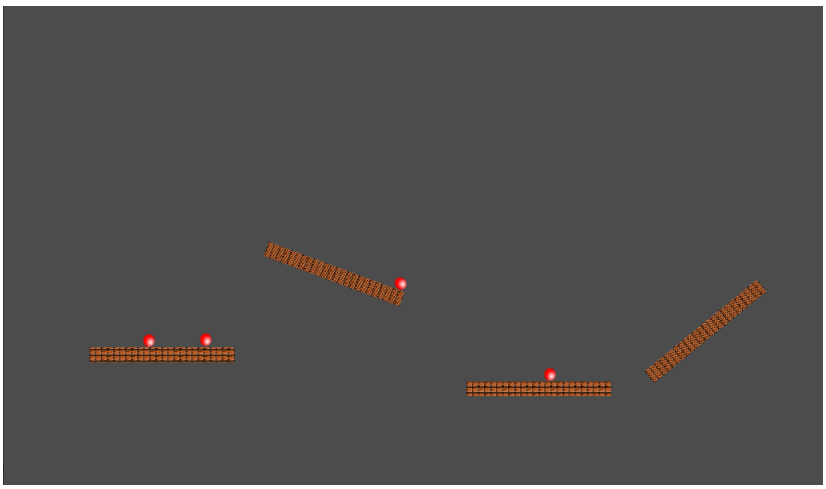


Figure 4.5: Instanciation de la scène

4.2. Un exemple d'une scène plus ambitieuse

On va maintenant illustrer cette notion au travers d'un exemple plus ambitieux : on va vouloir modéliser une scène 2D comprenant des balles, ainsi que des murs (qui doivent servir d'obstacles aux balles lorsqu'elles les rencontrent), par exemple :



Disposant de deux éléments distincts, à savoir les balles et les murs, nous allons créer deux scènes *Mur.tscn* et *Balle.tscn* qui vont modéliser ces objets. Ces scènes seront ensuite instanciées dans la scène principale.

4.2.1. Création des murs. Pour modéliser des murs, nous allons utiliser des nœuds fils de type *StaticBody2D* qui sont, comme leur nom l'indique, utilisés pour représenter des objets physiques 2D et immobiles dans l'environnement, en particulier des murs ou des plateformes. Ils permettent la détection de collision avec d'autres objets, mais ne seront jamais transformés par ces collisions.

Une fois le nœud fils *StaticBody2D* créé, vous pouvez remarquer qu'un symbole Attention apparaît : indiquant, lorsque l'on passe la souris dessus, que notre *StaticBody2D* n'a pas de

forme. Pour résoudre ceci, il va falloir lui associer un nœud fils de type *CollisionShape2D* (on aurait pu aussi utiliser *CollisionPolygon2D*), qui va permettre de lui donner une forme. Une fois ceci fait, cliquez sur ce nœud et dans l'inspecteur, la case *Shape* nous permet de choisir une

forme. Ici, on va choisir *Nouveau RectangleShape2D*, puisque l'on souhaite un mur de forme rectangulaire, voir Figure 4.6.

On peut également donner une apparence graphique à un mur ! Pour ce faire, il faut lui associer un autre nœud fils, de type *Sprite*. Dans l'inspecteur de ce nœud, il est alors possible de renseigner une donnée dans le champ *Texture*, voir Figure 4.7. Vous pouvez alors chercher le rendu graphique (simple !) d'un mur qui vous convient le mieux, et de l'enregistrer dans votre dossier de travail. Faites la ensuite glisser vers le gestionnaire de ressources (pensez à créer un dossier Images si ce n'est pas encore fait), et vous pouvez maintenant l'ajouter en tant que texture de votre mur.

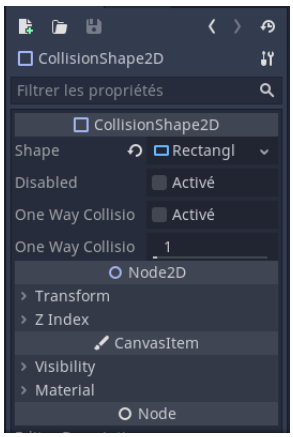


Figure 4.6: Choix de la forme d'un *CollisionPolygon2D*

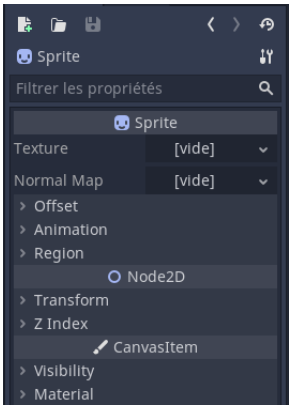


Figure 4.7: Ajout d'une texture au *Sprite*

Veillez alors à bien aligner la texture avec le *CollisionShape2D* (de préférence en haut à gauche du rectangle, sur le nœud *StaticBody2D*), quitte à utiliser les boutons Position, Rotation et Mise à l'échelle du menu standard. Vous pouvez ne modifier qu'un seul des deux nœuds *CollisionShape2D* et *Sprite* à la fois, ou bien les deux simultanément si vous sélectionnez directement le *StaticBody2D*. Une fois l'alignement effectué, faites bien attention de ne pas modifier un des deux éléments saus l'autre pour ne pas avoir de mauvaises surprises à l'exécution. Au cas où, vous pouvez rendre ces deux éléments non sélectionnables en sélectionnant le nœud *StaticBody2D*, et en réduisant ses fils pour ne pas les sélectionner individuellement.

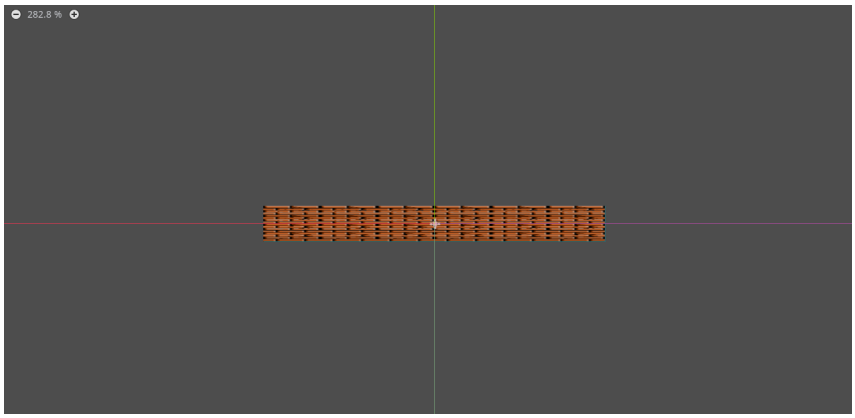


Figure 4.8: Scène *mur.tscn*

Voici maintenant un premier mur créé ! Vous pouvez maintenant enregistrer cette scène (par exemple avec un clic droit sur la scène en haut, puis *enregistrer la scène*, et la nommer *mur.tscn*, comme en Figure 4.8. Celle-ci sera alors accessible dans le dossier Scènes du gestionnaire de ressources.

4.2.2. Création d'une balle. Nous pouvons maintenant créer une nouvelle scène, que l'on nomme *balle.tscn*. Pour créer une balle, nous allons utiliser un nœud de type *RigidBody2D*, pour lui fournir un comportement physique sensible aux collisions. De même que pour le mur, un message de vigilance apparaît et on va associer au *RigidBody2D* deux nœuds fils : un *CollisionShape2D* qui permet de lui donner une forme (ici bien sûr *Nouveau CircleShape2D*) et un *Sprite* pour lui donner une texture, où vous êtes encore libre de choisir.

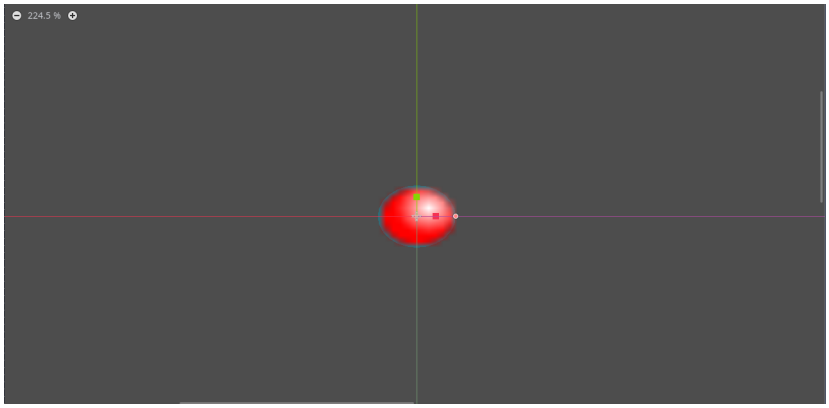


Figure 4.9: Scène *balle.tscn*

4.2.3. Scène principale. Pour générer la scène principale, nous allons ouvrir une nouvelle scène dont le nœud racine sera de type *Node2D*. Pensez à l'enregistrer directement, et à la définir comme scène principale du projet. Nous allons déjà construire les murs: après avoir sélectionné le nœud principal, 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 la scène *mur.tscn*. Nous pouvons alors répéter l'opération, voir Figure 4.10. Chaque nouveau nœud sera ajouté en haut à gauche (en $(0, 0)$), mais nous pouvons alors les déplacer à la souris pour les placer où nous voulons. Une fois tous les murs ajoutés et positionnés, nous pouvons alors de même ajouter les instances de *balle.tscn* suivant le même procédé. Notez qu'on peut également dupliquer des objets avec le raccourci *ctrl* + *d*, un nouvel objet sera alors créé à la même position que celui sélectionné.

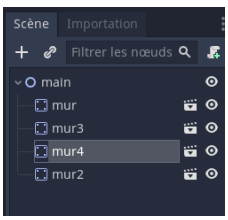


Figure 4.10: Plusieurs instances de la scène *mur.tscn*

Le projet peut alors être lancé, et nous verrons alors les balles tomber, due à la pesanteur qui affecte les *RigidBody2D*, et les murs resteront immuables. Essayez donc de placer une des balles au dessus d'un mur...

Chapitre 5

VisualScript

Nous avons vu comment créer un environnement comprenant différents types de nœuds. Maintenant, nous allons essayer d'écrire de certains scripts de base en *VisualScript*, qui permet la définition du comportement logique du jeu de manière graphique, uniquement avec des graphes et des représentations visuelles. Pour chacun de ces scripts, nous donnerons leur équivalent en *GDScript*, langage de programmation propre à Godot qui sera plus utilisé au second semestre.

5.1. Principe du VisualScript

Le *Visual Scripting* est présent dans les moteurs de jeu tels que *Unreal Engine* et *Unity*. De par son aspect purement graphique, il permet plus de collaborations entre programmeurs, artistes et designers pour réaliser plus rapidement des prototypes de jeu. Le *Visual Scripting* dans un moteur de jeu présuppose la définition d'une interface contenant des composants graphiques, la définition d'un graphe définissant le comportement de l'interface, et la définition d'une arborescence de nœuds correspondant aux composants graphiques. D'un point de vue fonctionnel, l'interface place visuellement les éléments graphiques des nœuds, le graphe définit le corps des fonctions des nœuds associés à un script, et l'arborescence définit la hiérarchie entre les nœuds.

Voici un exemple de graphe correspondant à un *VisualScript* :

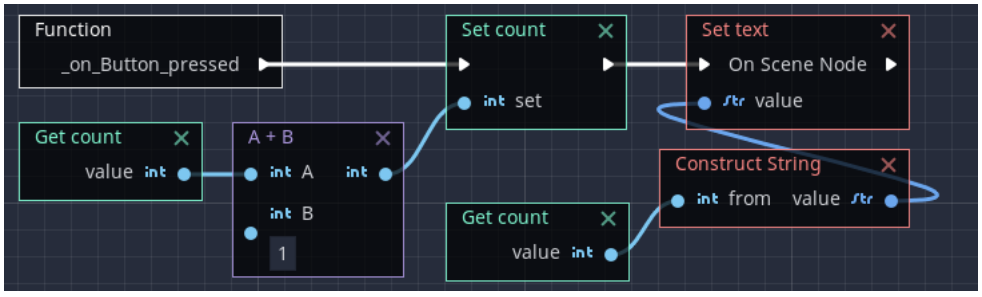


Figure 5.1: Exemple de *VisualScript*

Dans un tel graphe, on peut observer :

- des liens blancs, qui dictent l'exécution du programme;
- des liens bleus clairs pour donner des paramètres à des fonctions;

- des rectangles blancs pour les définitions de fonctions;
- des rectangles verts pour les accesseurs *Get-Set* de variables, permettant de récupérer la valeur d'une certaine variable à un moment de l'exécution;
- des rectangles violets pour les opérations logiques;
- des rectangles rouges pour les appels de fonctions;
- des rectangles roses pour les constantes: il n'y en a pas ici mais elles existent;
- Des types d'objets utilisés en bleu clair : ici on a des *int* pour désigner des entiers et des *str* pour désigner des chaînes de caractères.

5.2. PREMIERS EXEMPLES DE SCRIPTS

5.2.1. Incrémenter un *label* avec un opérateur *add*. Nous allons maintenant voir des premiers exemples de ce script. Ici, nous allons vouloir incrémenter la valeur d'un label en fonction du nombre ce clics. Pour ce faire, ouvrir une nouvelle scène avec un nœud principal de type *Control*, et lui associer deux nœuds fils: un nœud *Button* et un nœud *Label*, voir Figure 5.2.

Enregistrer la scène dans un fichier *main.tscn*. Dans l'interprète du *Button*, on peut écrire "Click count" dans le texte du nœud Button pour étiqueter le bouton, et mettre "0" dans le texte du Label, étant la valeur qu'on voudra incrémenter. En cliquant droit sur le nœud principal, on va alors choisir *Attacher un script*, et choisir le langage *VisualScript*, voir Figure 5.3.

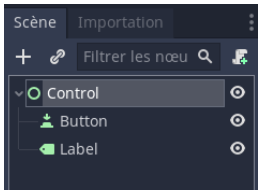


Figure 5.2: Arborescence

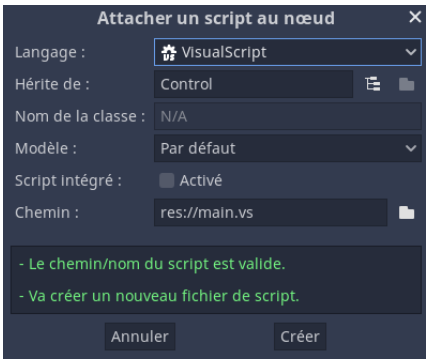


Figure 5.3: Choix du langage *VisualScript*

Le reste de la procédure se déroule alors dans l'onglet Script de l'éditeur *Godot*, voir Figure 5.4, et nous allons pouvoir définir notre script. Notez qu'en base à gauche de l'espace central, nous pouvons ajouter des fonctions, variables et signaux (nous y reviendrons) à l'aide du symbole +. L'autre symbole associé à Fonctions permet de rechercher des fonctions bien précises associées au type de nœud que l'on manipule. Nous pouvons aussi ajouter des fonctions à la main : ici nous voulons ajouter une fonction appelée *on_Button_pressed* au bouton, détectant si un bouton est pressé ou non. Pour ce faire, on peut sélectionner le *Button*, et aller dans l'onglet *Nœud*, situé au niveau de l'inspecteur, et double cliquer sur *pressed()* pour le connecter à la fonction *on_Button_pressed*, voir Figure 5.5. En conséquence, la fonction *on_Button_pressed* apparaît maintenant dans la liste des fonctions disponibles, voir Figure 5.6.

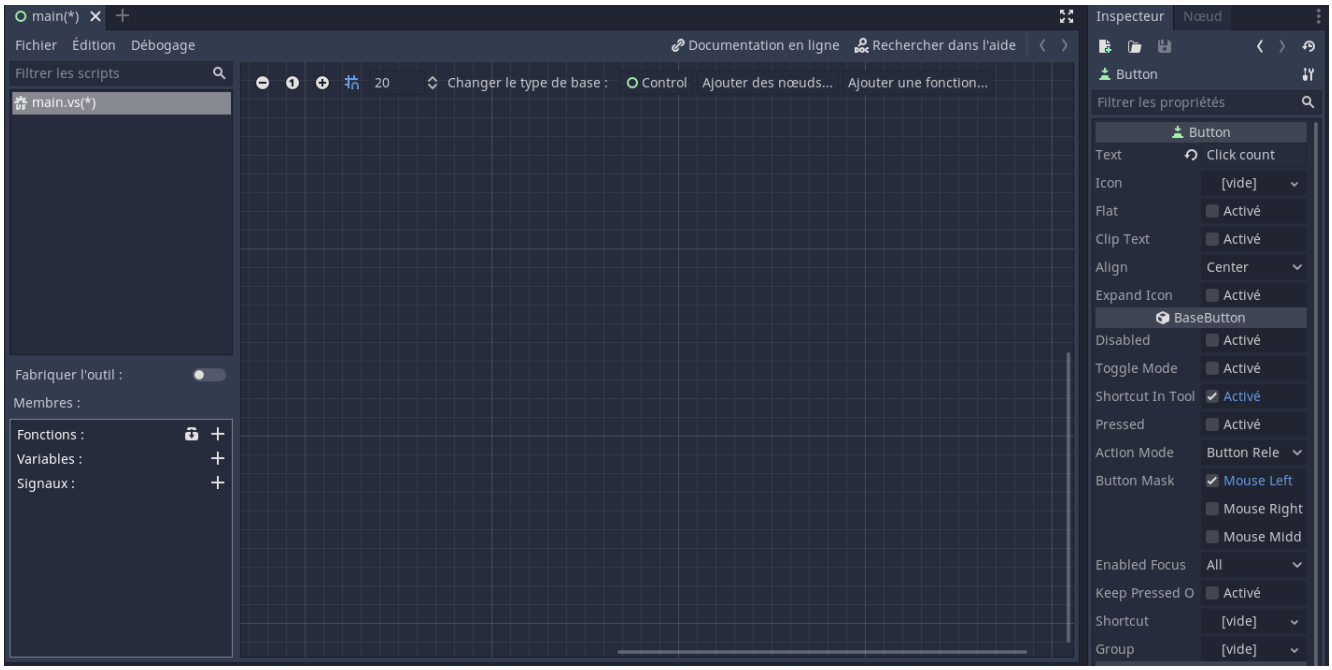


Figure 5.4: Interface du *VisualScript*

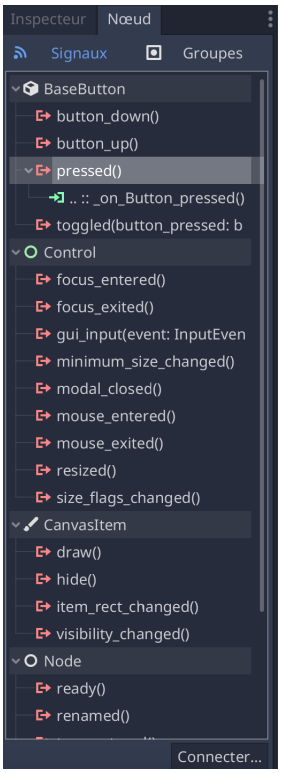


Figure 5.5: Ajout d'un signal sur le bouton

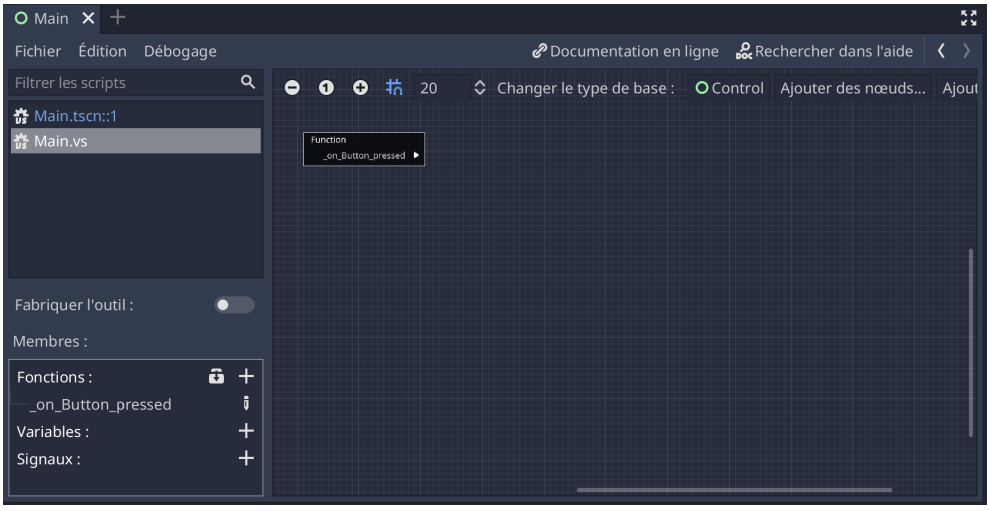


Figure 5.6: Ajout de la fonction *on_Button_pressed*

Ensuite, la procédure est la suivante:

1. Définir une variable à l'aide du symbole +. Renommez la en *count* en double cliquant sur cette variable, puis clic-droit et choisissez *Modifier le membre*. Choisissez un type

int, initialisez à 0 et rendez la exportable :ceci est important, cela permettra de modifier la valeur de la variable via l'inspecteur. Attention, pour initialiser à 0, il faudra parfois changer la valeur et la remettre à 0, autrement l'éditeur gardera la valeur *null* par défaut.

Vous pouvez maintenant ajouter ses accesseurs au script. Glisser-déplacer une variable dans la zone du script permet d'obtenir un nœud *Get*, tandis que glisser-déplacer en appuyant simultanément sur Ctrl permet d'obtenir un nœud *Set*.

2. Ajouter un opérateur *Add* de Math, via un click droit dans le *VisualScript* et une recherche de l'opérateur.

3. Relier l'entrée de *Get count* à une des entrées de *Add*; l'autre entrée devra alors être initialisée à 1 puisqu'on voudra ajouter 1 à chaque clic. Le résultat de cette addition devra être la nouvelle variable *count*, on peut donc relier la sortie de cet opérateur à un nœud *Set count*. Relier également la fonction *on_Button_pressed* au lien blanc de *Set count*.

4. Sélectionner le nœud *Label*, et utiliser l'inspecteur pour ajouter un nœud *Set text*, par un glisser-déplacer de la propriété *Text* de l'inspecteur.

5. *Set text* attend un paramètre de type *str*; il faut donc récupérer la valeur de la variable *count* avec *Get count*, et la convertir en string avec la fonction *Construct String*. Cette fonction s'ajoute automatiquement si vous essayez de relier la sortie de *Get count* à l'entrée de *Set text*.

Voici donc le *VisualScript* obtenu en Figure 5.7, et le résultat de l'éxécution après aucune action en Figure 5.8, et après 3 clicks en Figure 5.9.

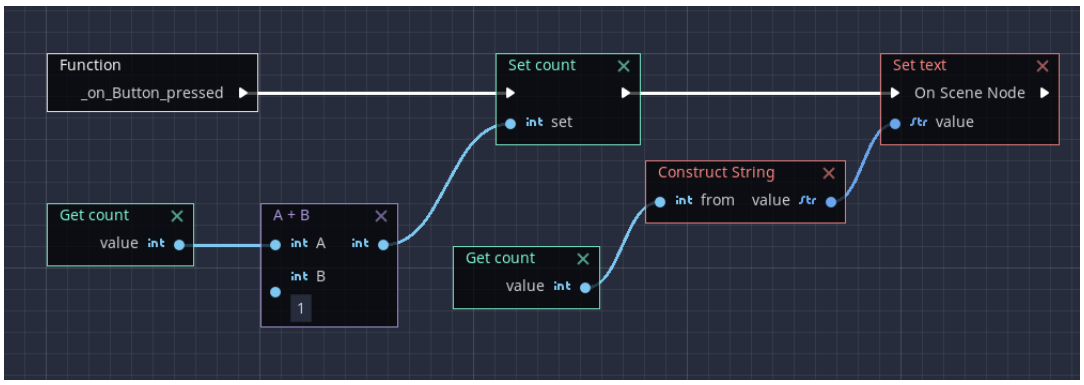


Figure 5.7: *VisualScript* d'incrémentation du *label*.

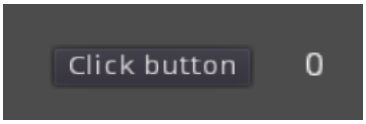


Figure 5.8: Exécution sans clic

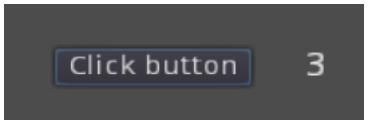


Figure 5.9: Exécution après 3 clics.

5.2.2. L'équivalent *GDScript*. On peut remplacer le *VisualScript* par le *GDScript* en Figure 5.10, qui a l'avantage d'être beaucoup plus concis. Pour ceci, on peut recréer une scène avec l'arborescence de la Figure 5.2, et au moment d'attacher un script au nœud *Control*, on choisit cette fois le langage *GDScript*. S'ouvre alors toujours l'onglet Script de *Godot*, avec déjà quelques lignes de code automatiques.

```
1  extends Control
2
3  var count
4
5  func _ready():
6      count = 0
7
8
9  func _on_Button_pressed():
10     count += 1
11     $Label.text = str(count)
```

Figure 5.10: *GDScript* équivalent.

Tous les nœuds correspondant à des interfaces graphiques héritent de la classe de base *Control*, d'où la ligne 1. La déclaration d'une variable est réalisée à l'aide du mot clé *var*. La fonction *_ready* est créée automatiquement. La fonction *on_Button_pressed* est ajoutée en sélectionnant *Button*, puis l'onglet *Node* de l'inspecteur et en double-cliquant sur *pressed*. La variable *count* est globale.

La fonction *_ready* est appelée à la création de l'interface et initialise la variable *count* à 0; et à chaque pression réalisée sur le composant *Button*, la fonction *on_Button_pressed* est appelée, ce qui incrémente la variable *count* et met à jour le texte du composant *Label*. Dans un *GDScript*, on accède à un composant en utilisant le nom du composant précédé du symbole $.

5.2.3. Incrémenter un label avec une fonction prédéfinie. Quand on clique sur un *Button*, la valeur d'un *label* est incrémentée avec une fonction prédéfinie *Add count*. L'arborescence, les fonctions et les variables sont identiques à la Section 5.2.1. On commence donc par créer un nœud principal *Control*, auquel on attache en nœuds fils un *Label*, dont on initialise le Text à 0, et un *Button* dans lequel on écrit " Click count " dans le champ Text. On attache ensuite au nœud principal un *VisualScript* Main.vs, et on définit la fonction *on_Button_pressed* comme précédemment. On définit une variable *count* de type *int* et exportable. La procédure est ensuite la suivante :

1. En sélectionnant *Control* et *count*, faire glisser la variable *count* depuis **l'inspecteur** (et pas le gestionnaire de variables) pour obtenir une fonction *Set count* (encadrée en rouge) qu'il est possible de réassigner à la fonction *Add* en modifiant le champ *Assign Op* dans l'inspecteur.

2. Ajouter un nœud *Get* pour la variable *count*.

3. Ajouter un nœud *Set text* pour le *label*.

4. Relier les nœuds et exécuter.

Figure 5.11: *VS* Incrémenter un *label* avec une fonction

Il est important de noter que l'inspecteur permet d'obtenir des fonctions plus paramétrables (telles que la fonction *Add count* dans la procédure ci-dessus). Le résultat sera identique en utilisant le *GDScript* de la section précédente.

5.2.4. Afficher dans un terminal les secondes qui passent. Au fil des secondes, une valeur *float* correspondant aux secondes écoulées est affichée dans un terminal. Voici l'arborescence et les fonctions et variables utilisées pour cette scène.

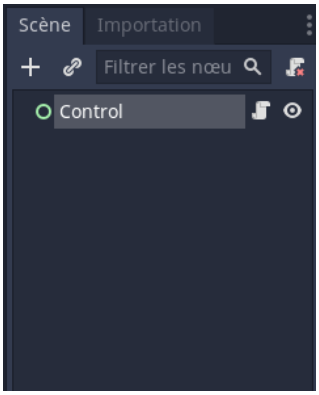


Figure 5.12: Arborescence

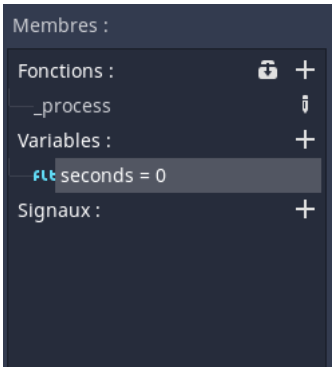


Figure 5.13: Fonctions et variables

Pour ce script, voici la procédure:

1. Définir un nœud principal *Control* (que l'on peut renommer si besoin), et attacher un VisualScript *Main.vs*.
2. Ajouter une fonction prédéfinie de type *process(float)*.
3. Définir une variable *seconds* de type *float* et exportable.
4. Ajouter la valeur de *process* à *seconds* avec *Add*.
5. Ajouter une fonction *print* avec un clic-droit et une recherche, et lui donner en argument un *Get* de *seconds*.

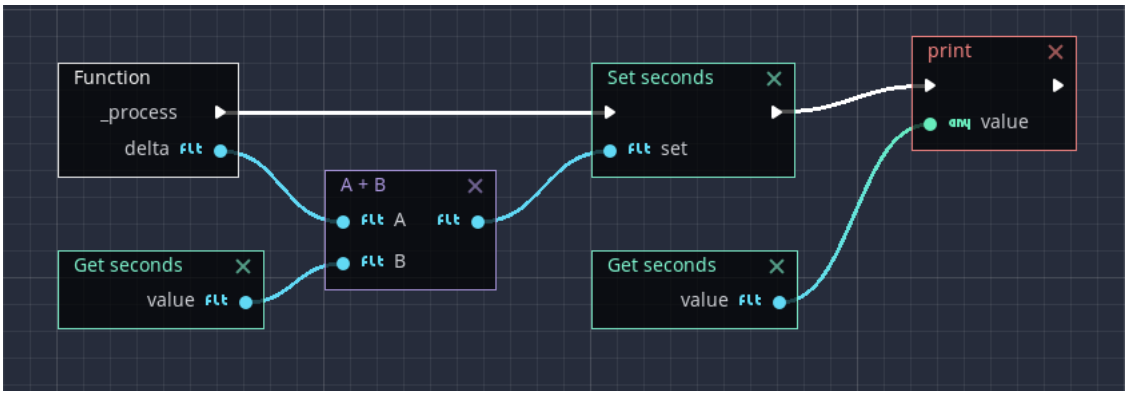


Figure 5.14: VS pour afficher les secondes dans un terminal.

Si vous allez voir dans l'aide, la fonction *process*, prenant en paramètres un flottant (et éventuellement un booléen), sert à rappeler à l'utilisateur quand un certain nœud est utilisé, d'où la mesure de temps 'discrète'. Lancez votre scène : vous voyez les secondes défiler, avec plusieurs décimales, dans le champ Sortie de l'éditeur *Godot*, voir Figure 5.16. Pour afficher les secondes sans dixièmes et centières, on peut utiliser un *Timer* se déclenchant toutes les secondes pour incrémenter une variable *seconds* de type *int*.

Dans ce cas, l'arborescence est composée d'un nœud principal de type *Control* auquel on ajoute un *Timer* comme fils. Pour celui-ci, cochez la propriété *Autostart* dans l'inspecteur. Il est possible de modifier sa fréquence avec le champ *Wait time*; nous le laisserons à la valeur par défaut 1. Attachez un *Main.vs* au nœud principal, et suivez la procédure suivante :

- Attachez un signal *timeout* de *Timer* vers le script *Main.vs* créant une fonction *_on_Timer_timeout*.
- Définir une variable *seconds* de type *int* et exportable.
- Ajouter les fonctions *print*, *Add*, *Get* et *Set* de *seconds*.

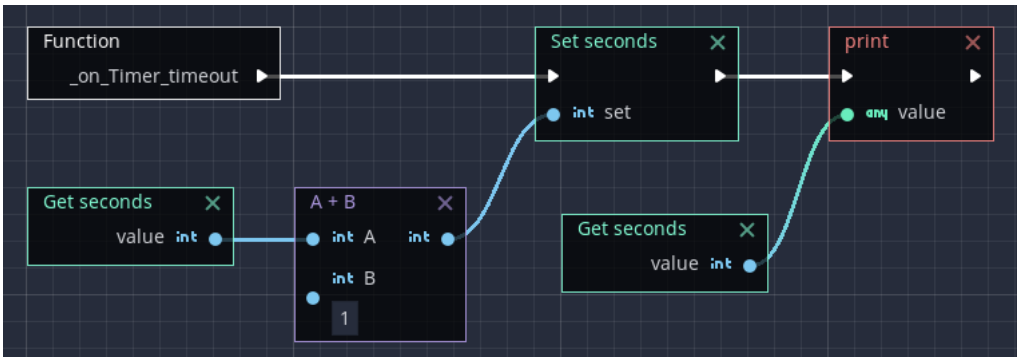


Figure 5.15: VS bis pour afficher les secondes dans un terminal.

Lancez alors votre scène : vous verrez cette fois-ci en sortie l'affichage des secondes entières, comme en Figure 5.17.

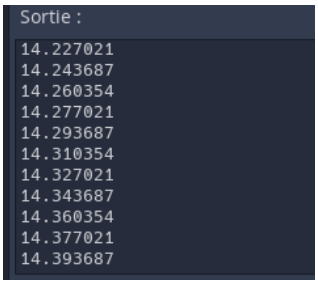


Figure 5.16: Résultat avec *process()*

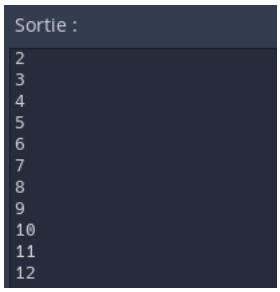


Figure 5.17: Résultat avec le *Timer*

Voici ci-dessous les équivalents *GDScript* des deux *VisualScript* précédents.

```
1  extends Control
2
3  var seconds : float
4
5  func _ready():
6      seconds=float(0)
7
8  func _process(delta):
9      seconds += delta
10     print(seconds)
```

Figure 5.18: *GDScript* avec *process()*

```
1  extends Control
2
3  var seconds : int
4
5  func _ready():
6      seconds=int(0)
7
8  func _on_Timer_timeout():
9      seconds += int(1)
10     print(seconds)
```

Figure 5.19: *GDScript* avec le *Timer*

Le mot-clé *var* abrège la classe *Variant* qui permet de stocker la plupart des types de données natifs de *Godot*, de réaliser des opérations sur les données par transformation d'une donnée ou application d'un opérateur sur plusieurs données, de réaliser des conversions, d'utiliser des structures de données telles que les tableaux, les dictionnaires, etc.

Dans les types atomiques, on a *bool*, *int*, *float* et *string*. Dans les types mathématiques, on a *vector2*, *rect2*, *vector3*, *transform2d*, *plane*, *quat*, *aabb*, *basis*, *transform*. Dans les types divers, on a *color*, *node_path*, *_rid*, *object*, *dictionnary* et *array*. Dans les types dits tableaux, on a les types *pool_byte_array*, *pool_int_array*, *pool_real_array*, *pool_string_array*, *pool_vector2_array*, *pool_vector3_array* et *pool_color_array*. Comme précisé en ligne 3, on pourra spécifier le type d'une variable. Comme présentée en ligne 6, on pourra réaliser un cast dans un type. La fonction *print* permet d'afficher des messages et des valeurs sur la console. Nous reviendrons sur plus de spécificités du *GDScript* dans le chapitre suivant.

5.2.5. Cumuler les clics et les secondes qui passent. On va vouloir reprendre l'exemple précédent d'incrémentation du *Label*, mais cette fois on souhaite que celui-ci soit incrémenté:

- non seulement à chaque clic sur un *button*,
- mais également au fil des secondes par les signaux d'un *timer*.

Nous aurons donc besoin d'un nœud principal de type *Control*, et de 3 nœuds fils: un *Button*, un *Label* et un *Timer*, reprenant les mêmes propriétés dans l'inspecteur que les exemples précédents. Nous aurons également besoin d'appeler les fonctions *_on_Button_pressed* et *_on_Timer_timeout* des Sections 5.2.1 et 5.2.4.

En s'inspirant des exemples précédents, voici le *VisualScript* correspondant à cette action.

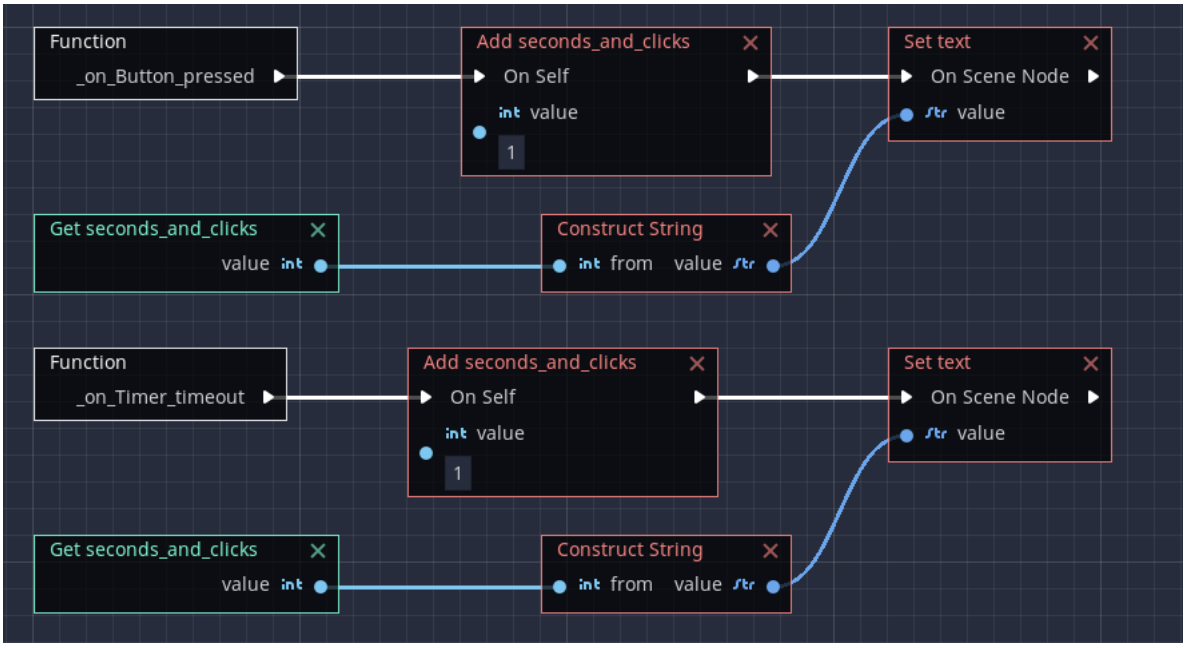


Figure 5.20: VS pour incrémenter le *Label* à chaque clic et chaque seconde.

Notez que ce script est séparé en deux composantes distinctes, chacune correspondant aux *VisualScripts* des Sections 5.2.1 et 5.2.4. On a juste une variable *seconds_and_clicks* qui s'incrémente à chaque seconde et à chaque clic. Ci-dessous le *GDScript* correspondant.

```
1  extends Control
2
3
4  var seconds_and_clicks : int
5
6  func _ready():
7      seconds_and_clicks = int(0)
8
9  func inc_count():
10     seconds_and_clicks += 1
11     $Label.text = str(seconds_and_clicks)
12
13 func _on_Button_pressed():
14     inc_count()
15
16 func _on_Timer_timeout():
17     inc_count()
```

Figure 5.21: *GDScript* pour incrémenter le *Label* à chaque clic et chaque seconde.

5.2.6. Conditionner un clic par une valeur minimale. On veut créer une scène contenant deux boutons et deux labels appelés respectivement *Button1*, *Button2*, *Label1* et *Label2* de telle sorte que chaque clic sur *Button1* incrémente *Label1*, et chaque clic sur *Button2* incrémente la

valeur de *Label2* **et** soustrait 10 à la valeur de *Label1* **si et seulement si** la valeur de *Label1* est inférieure à 10. Sinon, *Button2* ne fait rien, et les valeurs de *Label1* et *Label2* restent inchangées.

Les figures 5.22 et 5.23 présent l'arborescence, les fonctions et variables utilisées dans la scène.

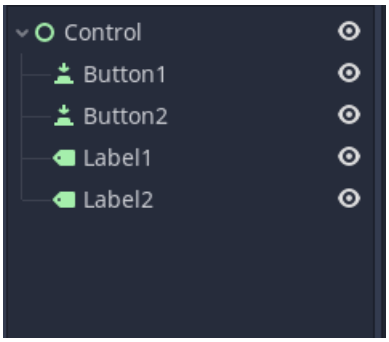


Figure 5.22: Arborescence de la scène.

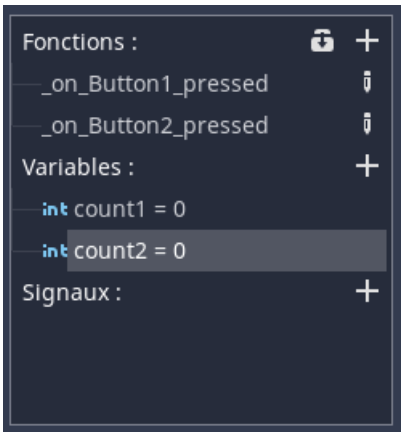


Figure 5.23: Fonctions et variables de la scène.

Voici la procédure à suivre pour rédiger le *VisualScript* :

- Attacher un VisualScript *Main.vs* au nœud *Control*, et définir les fonctions *_on_Button1_pressed* et *_on_Button2_pressed* en utilisant l'inspecteur. Ajouter les variables *count1* et *count2* de type *int* et exportables.
- Ajouter un opérateur *Add* et un opérateur *CompareGreater* (symbole A > B), qui permettra de tester si la valeur de *Label1* est bien supérieur à 10.
- Ajouter des nœuds *Get* et *Set* pour les variables *count* et *count2*, et conditionner la mise à jour de *count2* par un minimum de 10 sur *count* avec le *CompareGreater*.
- Cliquer sur *Button2* décrémente *count* de 10 et incrémente *count2* de 1. Pour décrémenter, on utilisera l'opérateur *Subtract*.
- Ajouter les mises à jour des *label* aux mises à jour des valeurs *count* et *count2*.
- Relier les nœuds et éxécuter.

La partie supérieur du VisualScript, déconnectée du reste, ne concerne que l'incrémentation du *Label1* par un clic sur le *Button1*, et est semblable au script de la Section 5.2.1. En revanche, la partie inférieure est plus compliquée : la fonction *_on_Button2_pressed* est reliée par un lien blanc à une condition, dont la sortie blanche est sur la valeur *true*. Cela signifie que la fonction sera appelée si la condition est vraie. La condition est ce qui arrive en entrée de *Condition* dans le graphe, à savoir tester si *count1* est strictement supérieur à 9 (ou supérieur ou égal à 10). Si oui, alors on va modifier *count1* en lui retirant 10 (partie inférieure gauche) et on va incrémenter *count2* (partie inférieure centrale). Il reste alors à afficher le texte des *Label1* et *Label2* par la méthode qu'on connaît: on récupère les valeurs des variables avec *Get*, et on les envoie sous forme de string à la fonction *Set text*.

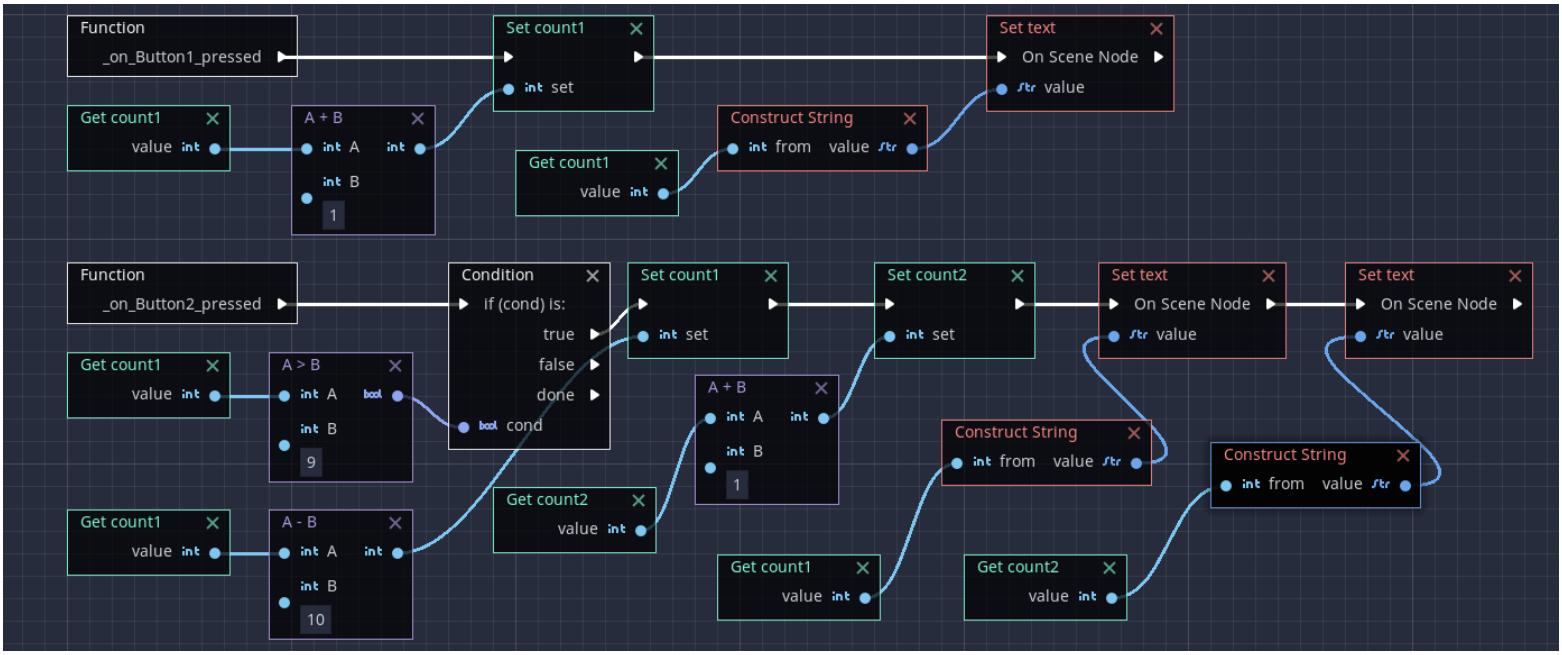


Figure 5.24: *VisualScript* pour condition de clic par une valeur minimale.

Voici ci-dessous le code *GDScript* correspondant.

```gdscript
extends Control

var count1 : int
var count2 : int

func _ready():
    count1 = int(0)
    count2 = int(0)

func _on_Button1_pressed():
    count1 += 1
    $Label1.text = str(count1)

func _on_Button2_pressed():
    if count1 > 9 :
        count1 -= 10
        count2 += 1
        $Label1.text = str(count1)
        $Label2.text = str(count2)
```

Figure 5.25: *GDScript* pour condition de clic par une valeur minimale.

5.2.7. Activer un bouton selon une valeur minimale. Étudions maintenant un dernier exemple dans cette section. Reprenons la scène précédent avec deux boutons et deux labels. On veut cette fois faire en sorte que lorsque la variable *count1* est strictement inférieure à 10, le *Button2* ne soit pas cliquable (au lieu de pouvoir cliquer mais que cela ne change rien).

Pour ce faire, on garde la même arborescence que pour la scène précédente, mais on va utiliser en outre la fonction *_process*, qui va permettre de définir la relation entre les deux boutons. Voici la procédure à suivre:

- Attacher un VisualScript *Main.vs* au nœud *Control*, et définir les fonctions *_on_Button1_pressed* et *_on_Button2_pressed* en utilisant l'inspecteur. Ajouter les variables *count1* et *count2* de type *int* et exportables.
- Ajouter des opérateurs *Add* pour ajouter 1 et mettre à jour les *Label* correspondants; on a deux procédures indépendantes pour *Button1* et *Button2*. Puis, on va ajouter une troisième sous-procédure pour désactiver *Button2*.
- Définir *Button2* comme initialement désactivé en cochantle champ *Disabled* dans l'inspecteur.
- Utiliser la fonction *_process* qui vérifie la condition sur *count* pour activer ou désactiver *Button2*, à l'aide d'un opérateur *Compare Greater*.
- Ajouter la fonction *Set Disable* en glissant-déplaçant le champ *Disabled* de l'inspecteur, et relier les sorties *true* et *false* de la condition.

Voici ci-dessous les scripts récapitulant les trois sous-procédures indépendantes:

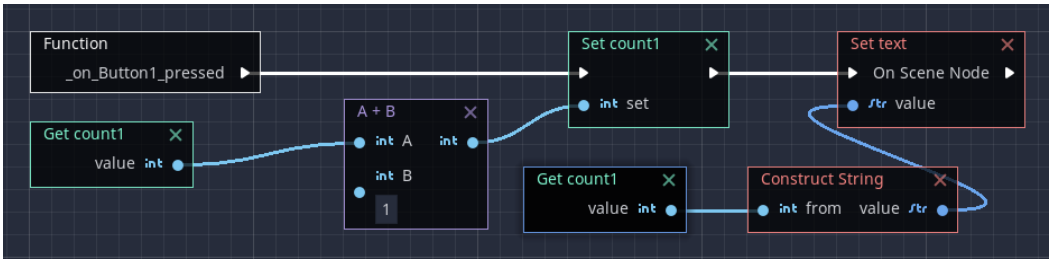


Figure 5.26: Clic sur le *Button1*.

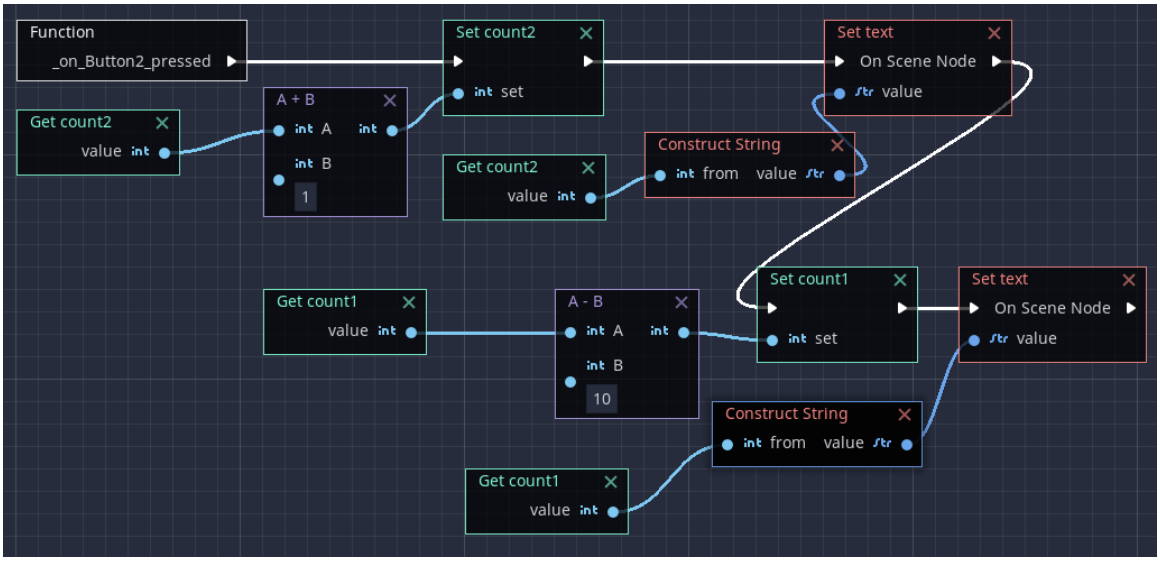


Figure 5.27: Clic sur le *Button2* avec modification du *Button1*

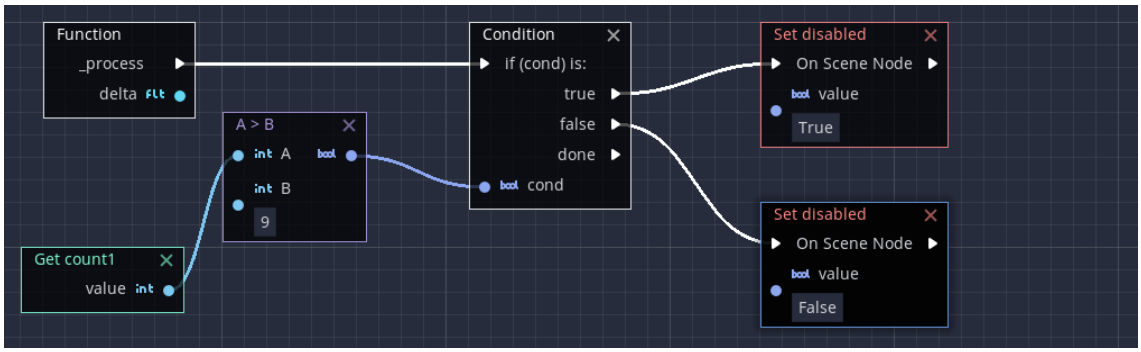


Figure 5.28: Activation et désactivation du *Button2* selon la condition.

Voici ci-joint le script *GDScript* correspondant:

```
1  extends Control
2
3  var count1 : int
4  var count2 : int
5
6  func _ready():
7      count1 = int(0)
8      count2 = int(0)
9
10 func _on_Button1_pressed():
11     count1 += 1
12     $Label1.text = str(count1)
13
14 func _on_Button2_pressed():
15     count1 -= 10
16     count2 += 1
17     $Label1.text = str(count1)
18     $Label2.text = str(count2)
19
20 func _process(delta):
21     if count1 < 10:
22         $Button2.disabled = true
23     else:
24         $Button2.disabled = false
```

Chapitre 6

Sprites animés et *tilemaps*

Dans le Chapitre 4, nous avons déjà ajouté des nœuds *Sprite* en tant que nœuds fils de *StaticBody2D* et *RigidBody2D*, et nous avions vu comment leur ajouter des textures. Dans ce chapitre, nous irons plus loin sur afin notamment de voir comment animer des *Sprite*, et dessiner des *tilemaps* permettant d'animer l'environnement ambiant.

6.1. Sprites animés

On rappelle que pour dessiner un *Sprite*, il suffit de créer un nœud principal de type *Node2D*, et de lui associer un nœud fils *Sprite*. On peut ensuite associer à ce *Sprite* une texture, en choisissant une image que l'on ajoute dans le dossier Images du gestionnaire de ressources, et en la faisant glisser dans le champ *Texture* de l'interprète.

6.1.1. *Sprite* animés. On va ici placer quatre *sprite* animée d'un *Block?* dans la scène, dont l'arborescence se trouve en Figure 6.1. Voici la procédure pour les dessiner :

- Ajouter un nœud principale de type *Node2D*, renommez le en *Main* si vous le souhaitez, puis enregistrez la scène en *Main.tscn*.
- Ajouter un nœud *AnimatedSprite* et créer un nouveau *SpriteFrame* dans le champ *Frames*. On peut alors placer les images du répertoire *Images* dans l'animation par défaut du *SpriteFrame*, voir Figure 6.3.
- Dupliquer le nœud *AnimatedSprite* avec Ctrl + D pour créer automatiquement un nœud *AnimatedSprite2*. Le nœud nouvellement créé est superposé avec l'original dans la scène; le déplacer. Répéter l'opération jusqu'à obtenir 4 *Block?* alignés.
- Synchroniser les animations en sélectionnant les 4 *AnimatedSprite*, en désactivant *Playing*, en fixant *Frame* à 0, et en activant *Playing*.
- Exćuter pour avoir une animation synchronisée !

Figure 6.1: Arborescence de la scène.

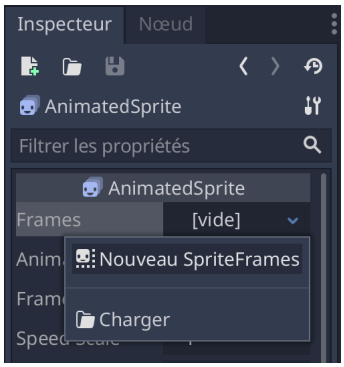


Figure 6.2: Création d'une *SpriteFrame*.

Figure 6.3: Animation par défaut du *SpriteFrame* du *Block?*.

6.1.2. Animer des scintillements dans un cadre. On va définir une animation de quatre scintillements en les plaçant dans des quadrants de l'image, et où chaque animation de scintillement commence à des frames différentes. On uilise les 5 images de scintillements de la Figure 6.4 permettant, si utilisées l'une après l'autre, de créer une animation d'un scintillement.

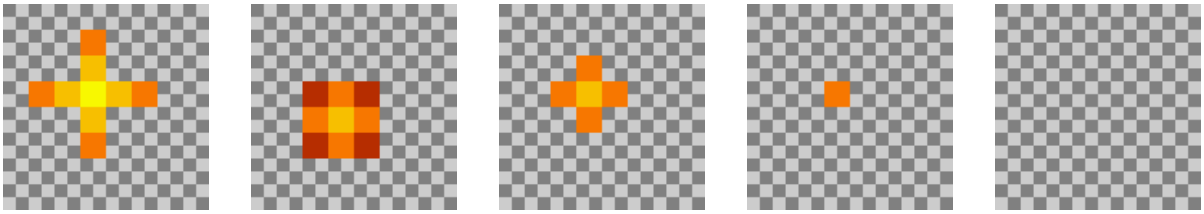


Figure 6.4: Images du scintillement

Dans cette scène, nous allons créer un nœud principal de type *Node2D*, que nous renommerons en *Firework*, auquel nous attacherons (pour le moment) un nœud fils de type *AnimatedSprite*, que nous appellerons *Flicker1*. Créons lui une nouvelle *SpriteFrames* contenant les 5 images de la Figure 6.4. Activez ensuite le champ *Playing*, puis fixez le champ *Speed Scale* à 0.1. Attachez ensuite un *GDScript* nommé *Flicker1.gd* au nœud *Flicker1* (attention, pas encore au *Firework*), voir Figure 6.5. Ensuite, dupliquez 3 fois le nœud *Flicker1* pour obtenir des *AnimatedSprite* nommés *Flicker2*, *Flicker3* et *Flicker4*. Attachez ensuite au nœud *Firework* le *GDScript* suivant, voir Figure 6.6.

```gdscript
extends AnimatedSprite

func _process(delta):
    if self.frame == 4:
        self.playing = false
```

Figure 6.5: *GDScript* associé au nœud *Flicker1*.

```gdscript
extends Node2D

var init_states=[1,2,3,4]
var shuffle=true
var speed=3

func _ready():
    init_states.shuffle()
    $Flicker1.frame = init_states.pop_front()
    $Flicker1.speed_scale= speed
    $Flicker2.frame = init_states.pop_front()
    $Flicker3.frame = init_states.pop_front()
    $Flicker4.frame = init_states.pop_front()
```

Figure 6.6: *GDScript* associé au nœud *Firework*.

Si vous exécutez alors la scène, vous verrez les quatres scintillements qui s'effectuent en décalé, par exemple comme en Figure 6.8.

Notez que l'initialisation de la variable *self.frame* à false indique que l'animation s'arrêter de jouer. Les lignes 4 et 5 de *Flicker1.gd* indiquent donc que les animations s'arrêront de jouer après 4 frames. On peut augmenter cette valeur si on veut que les animations jouent plus longtemps, voir s'en affranchir si on veut une animation en continu. Dans ce cas, il suffit de laisser la vairable *self.playing* à true. Dans *Firework.gd*, le tableau *init_states* correspond aux valeurs de la première Frame d'animation de chacun des *Flicker*. La fonction *pop_front* est une fonction de tableau qui permet de renvoyer son premier élément et le supprimer dans le tableau; ainsi ligne $*Flicker1.frame = init_states.pop_front()* indique que la frame d'initialisation de *Flicker1* sera la première valeur du tableau *init_states*. La fonction *shuffle* permet de mélanger de me manière aléatoire les éléments du tableau, histoire d'ajouter un peu d'imprévu dans l'ordre d'activation des animations. Il est également possible de contrôler la vitesse d'activation des animations (*Speed scale*) à l'intérieur du *GDScript* comme présenté en Figure 6.7.

```gdscript
extends Node2D

var init_states=[1,2,3,4]
var shuffle=true
var speed=3

func _ready():
    init_states.shuffle()
    $Flicker1.frame = init_states.pop_front()
    $Flicker1.speed_scale= speed
    $Flicker2.frame = init_states.pop_front()
    $Flicker3.frame = init_states.pop_front()
    $Flicker4.frame = init_states.pop_front()
```

Figure 6.7: Réglage de la vitesse d'activation des *Frame*.

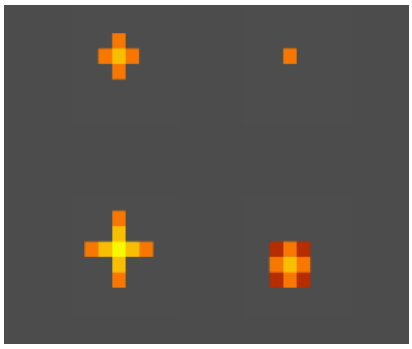


Figure 6.8: Illumination des scintillements.

Puisqu'il y a plusieurs scripts en jeu dans cette scène, nous allons également ici donner les équivalents *VisualScript*. Commencez par reprendre la même arborescence, avec un nœud principal *Firework* de type *Node2D* et un nœud fils *Flicker1* de type *AnimatedSprite*. Associez un Visual script *Flicker.vs* à ce dernier, et reproduire le script présenté en Figure 6.9. Dupliquez ensuite le nœud *Flicker1* pour créer *Flicker2*, *Flicker3* et *Flicker4*, et déplacez les comme dans le cas précédent.

Ensuite, attachez un Visual Script *Firework.vs* au nœud *Firework*, dans lequel il faut ajouter la fonction *physics_process*, ainsi que les variables *shuffle* de type booléen à true, et *init_states*

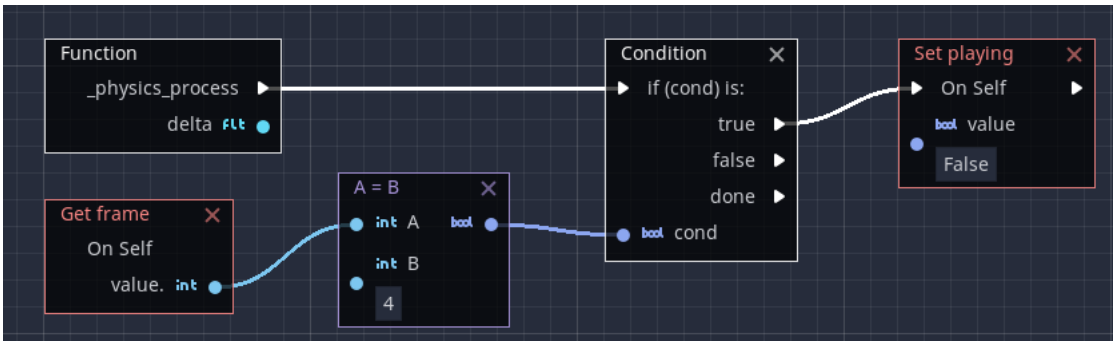


Figure 6.9: *VisualScript* associé au *Flicker*.

de type array à $[1, 2, 3, 4]$. Pour définir les valeurs d'un tableau, il faut d'abord définir sa taille, puis pour chaque élément cliquer sur l'icône de modification, choisir le type de la valeur (par exemple *int*), puis la fixer. Reproduire ensuite le *VisualScript* présenté en Figure 6.10. Pour récupérer la fonction *Set frame*, il faut glisser-déplacer en maintenant la touche Ctrl depuis l'onglet *Frame* de l'inspecteur associé à un Flicker. Attention, les quatre fonctions *Set frame* apparaissant ici sont des fonctions provenant respectivement de l'inspecteur pour *Flicker1*, *Flicker2*, *Flicker3* et *Flicker4* !

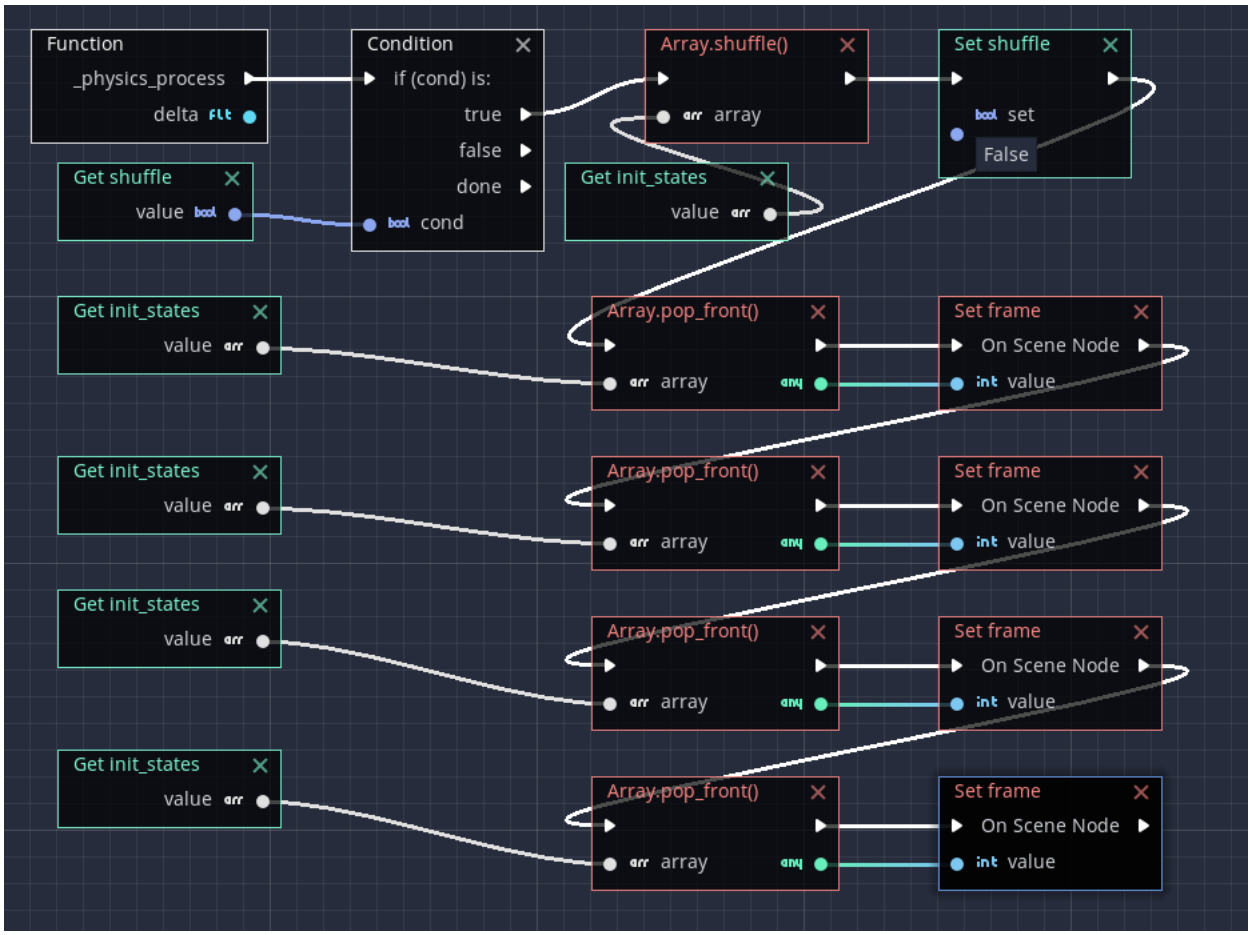


Figure 6.10: *VisualScript* associé au *Firework*.

6.2. DÉPLACEMENT ET ANIMATION

6.2.1. Déplacer un *sprite* au clavier. On veut maintenant ajouter un *sprite* correspondant à un personnage, par exemple avec une texture de Mario, qui se déplacera à gauche et à droite par action des touches flèche gauche et flèche droite. Créons donc un nœud principal de type *Node2D*. **Pour modéliser un personnage, nous allons utiliser un nœud de type *KynematicBody2D***. Ici, vous avez un symbole d'avertissement stipulant que ce nœud n'a pas de forme. On peut ainsi créer un nœud fils de type *CollisionShape2D*, mais puisque nous ne l'utiliserons pas ici on peut s'en passer pour le moment. Ajouter un nœud *Sprite* en tant que fils, et lui ajouter la texture *Mario.jpg* de la figure 6.11. Avec l'inspecteur, fixer les coordonnées x et y du champ *Position* de *Mario* à 300 et 200.

Figure 6.11: Texture de *Mario*.

Voici ensuite la procédure à suivre:

- Si ce n'est pas déjà bon, modifier les actions *Left*, *Right* et les associer aux touches flèches gauche et flèche droite dans *Projet*, *Paramètres du projet*, *Contrôles*.
- Attacher un script *VisualScript* au nœud *Mario*.
- Ajouter une variable *speed* de type *Vector2* initialisée à $(200, 200)$ et une variable *velocity* de type *Vector2* initialisée à $(0, 0)$.
- Créer une fonction *get_input* à l'aide du symbole $+$, et ajouter la fonction *_physics_process*. Réaliser les graphes correspondant aux Figures 6.12 et 6.13.
- Exécuter, et appuyer sur les touches flèche gauche et flèche droite pour déplacer le *Mario*.

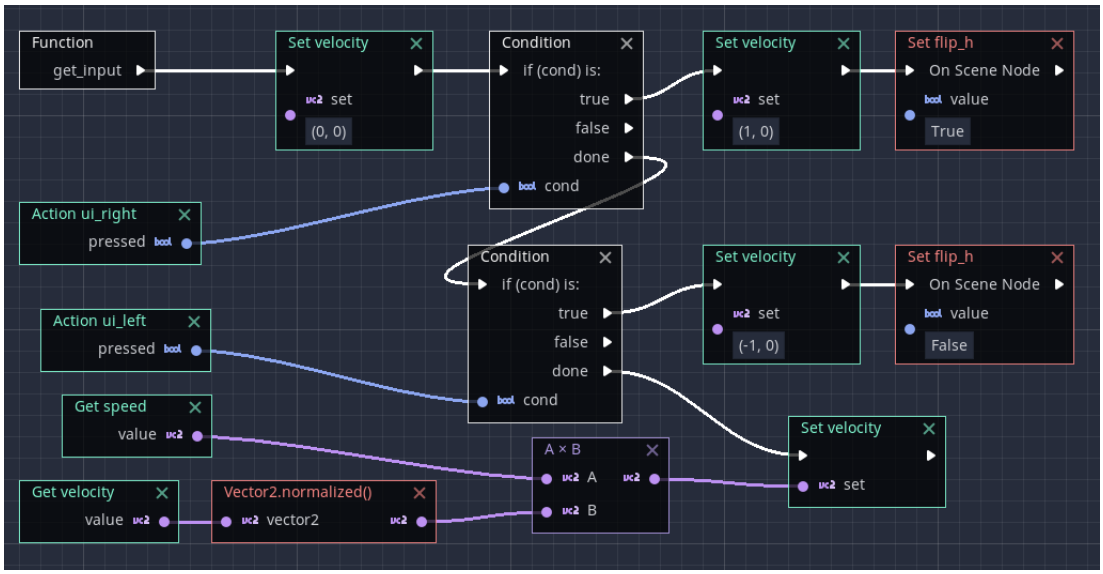


Figure 6.12: Fonction *get_input*.

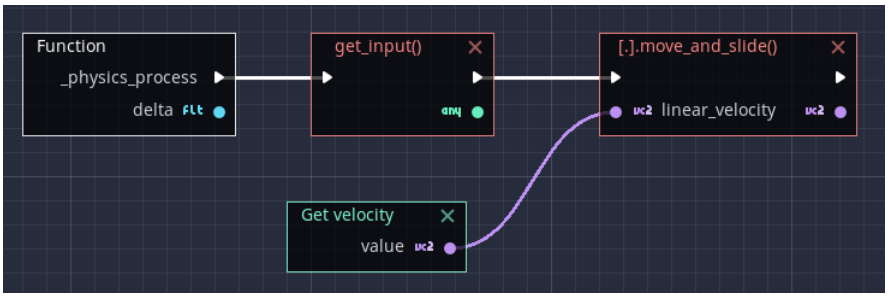


Figure 6.13: Procédure de déplacement de *Mario*.

La fonction *Setflip_h* est associée au nœud *Sprite*, elle permet d'orienter le *Sprite* horizontalement dans le sens du déplacement. Pour l'ajouter au *VisualScript*, il suffit de sélectionner le *Sprite*, et de faire glisser la propriété *Flip H* de l'onglet *Offset* de l'inspecteur. Pour appeler la fonction *get_input* dans le script de *_physics_process*, il suffit de glisser-déplacer la fonction *get_input* depuis le gestionnaire de fonctions. La fonction *Action* permet de récupérer les touches clavier pressées. On peut modifier la touche d'action voulue dans l'interprète. L'opération $A \times B$ sur le type *Vector2* est une multiplication terme à terme.

La fonction *_physics_process* est celle qui gère le déplacement physique de notre personnage. Elle est appelée à la fréquence d'affichage (c'est-à-dire 60 fois par seconde). Il est possible de modifier la fréquence d'appel avec le paramètre *delta*, dont la valeur par défaut est de 0.01666 pour 60 appels par seconde. Le *Vector2* en paramètre de *move_and_slide* est une vitesse en pixels par seconde. L'aide de la fonction *move_and_slide* (qui prend en argument un *Vector2* de vélocité et potentiellement pleins d'autres) indique que cette fonction déplace l'objet physique selon un vecteur, et qu'en cas de collision avec un autre objet, il glissera le long de cet objet plutôt que de s'arrêter immédiatement. Si l'autre objet est de type *KinematicBody2D* ou *RigidBody2D*, il sera aussi affecté par cette collision. Voici le *GDScript* équivalent à ce *VisualScript* en Figure 6.14.

```
1  extends KinematicBody2D
2
3  var speed : Vector2
4  var velocity : Vector2
5
6  func _ready():
7      speed = Vector2(700,700)
8      velocity = Vector2(0,0)
9
10 func get_input():
11     velocity = Vector2(0,0)
12     if Input.is_action_pressed('ui_right'):
13         velocity = Vector2(1,0)
14         $Sprite.flip_h = true
15     if Input.is_action_pressed('ui_left'):
16         velocity=Vector2(-1,0)
17         $Sprite.flip_h = false
18     velocity = speed * velocity.normalized()
19
20 func _physics_process(_delta):
21     get_input()
22     velocity=move_and_slide(velocity)
```

Figure 6.14: *GDScript* pour déplacement gauche-droite de *Mario*.

6.2.2. Déplacer un *sprite* associé à plusieurs images. On veut maintenant reprendre le script précédent, mais ajouter une action lorsque l'on appuie sur la touche flèche du bas : celle de baisser notre Mario. Pour ceci, nous avons besoin d'une deuxième image de Mario lorsqu'il se baisse, voir Figure 6.15.

Figure 6.15: Texture de *Mario* baissé.

Nous allons utiliser les mêmes variables, les mêmes fonctions et la même arborescence que dans la Section 6.2.1. Il va cependant falloir ajouter une condition dans la fonction *get_input* lorsque la touche flèche vers le bas est pressée. Pour modifier la texture en fonction du résultat du test, il faut utiliser la fonction *Preload* pour charger l'image, et la fonction *Set texture* pour changer l'image associée. Voici le *GDScript* associé :

```
1  extends KinematicBody2D
2
3  var speed : Vector2
4  var velocity : Vector2
5
6  func _ready():
7      speed = Vector2(700,700)
8      velocity = Vector2(0,0)
9
10 func get_input():
11     velocity = Vector2(0,0)
12     if Input.is_action_pressed('ui_right'):
13         $Sprite.texture = preload("res://Images/Mario.png")
14         velocity = Vector2(1,0)
15         $Sprite.flip_h = true
16     if Input.is_action_pressed('ui_left'):
17         $Sprite.texture = preload("res://Images/Mario.png")
18         velocity=Vector2(-1,0)
19         $Sprite.flip_h = false
20     if Input.is_action_pressed('ui_down'):
21         $Sprite.texture = preload("res://Images/Mario_down.png")
22     if Input.is_action_pressed('ui_up'):
23         $Sprite.texture = preload("res://Images/Mario.png")
24     velocity = speed * velocity.normalized()
25
26 func _physics_process(_delta):
27     get_input()
28     velocity=move_and_slide(velocity)
```

Figure 6.16: *GDScript* pour l'animation de Mario qui se baisse.

C'est le premier exemple où nous commençons directement par le *GDScript* et non par le *VisualScript*, puisque le code est plus simple à légèrement modifier, et cela évite de recréer toute l'arborescence ! Dorénavant, nous allons nous concentrer sur le *GDScript*, en vue du cours *Programmation dans les moteurs de jeux*. Entraînez-vous à recréer le *VisualScript* associé !

6.2.3. Déplacer un *Sprite* avec plusieurs animations.. On va maintenant utiliser un *AnimatedSprite* qui va être sujet à 3 animations: deux animations à une image et une animation à deux images. Appuyer sur les touches flèche gauche et flèche droite permet respectivement de déplacer le *Sprite* vers la gauche et la droite, appuyer sur la flèche du bas permet de baisser *Mario*. Enfin, si aucune touche n'est pressée, *Mario* reprend sa position de départ, debout à l'arrêt.

On utilise les images de *Mario* de la section précédente, et les deux images de *Mario* situées ci-dessous permettant, si utilisées l'une après l'autre, de créer une animation de marche pour *Mario*.

Créez un nœud *Main* de type *Node2D*, attachez-lui un nœud fils de type *KinematicBody2D* à renommer en *Mario*, et attachez à ce nœud un nœud fils de type *AnimatedSprite*. Notez que vous pouvez ignorer l'avertissement de forme que vous donne le *KinematicBody2D*, cela ne nous sera pas utile, ou alors vous pouvez ajouter un nœud de type *CollisionShape2D* comme précédemment. L'arborescence de la scène est présentée en Figure 6.17. Sélectionnez le *AnimatedSprite*, et ajoutez une animation en créant un nouveau *SpriteFrame* dans le champ *Frames*. Á l'aide du bouton *Nouvelle Animation* (tout à gauche), créez alors 3 animations, appelées Stand, Down et Walk. Ajoutez les images de la section précédentes dans Stand et Down, puis les deux nouvelles images dans l'animation Walk, voir Figure 6.18.

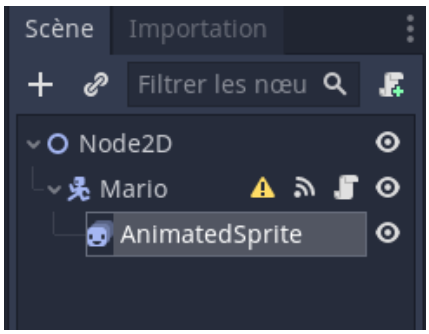


Figure 6.17: Arborescence de la scène.

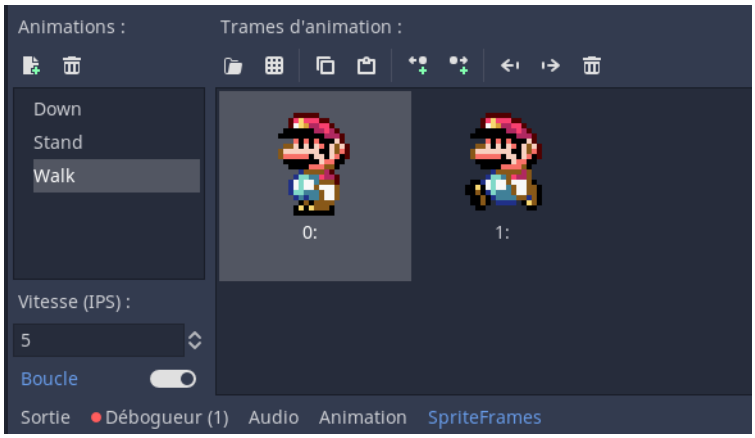


Figure 6.18: *SpriteFrame* contenant les 3 animations.

On va maintenant attacher à un script au nœud Mario. Nous pouvons repartir du *GDScript* de la section précédente, voir Figure 6.16, et le modifier comme en Figure 6.19. On notera que dans le cas des animations avec une seule frame, à savoir Stand et Down, il n'y a pas besoin de remettre la variable $*AnimatedSprite.playing* à *true*.

6.2.4. Placer des décors et une caméra 2D**.** Reprenons notre *AnimatedSprite* Mario de la Section 6.2.3, et plaçons le maintenant dans un décor, de sorte que la vue présentée par l'interface

```gdscript
extends KinematicBody2D

var speed : Vector2
var velocity : Vector2

func _ready():
    speed = Vector2(300,300)
    velocity = Vector2(0,0)

func get_input():
    velocity = Vector2(0,0)
    var no_action_pressed = true
    if Input.is_action_pressed("ui_left"):
        no_action_pressed = false
        $AnimatedSprite.playing = true
        $AnimatedSprite.animation = "Walk"
        velocity = Vector2(-1,0)
        $AnimatedSprite.flip_h = false
    if Input.is_action_pressed("ui_right"):
        no_action_pressed = false
        $AnimatedSprite.playing = true
        $AnimatedSprite.animation = "Walk"
        velocity = Vector2(1,0)
        $AnimatedSprite.flip_h = true
    if Input.is_action_pressed("ui_down"):
        no_action_pressed = false
        $AnimatedSprite.animation = "Down"
    if no_action_pressed:
        $AnimatedSprite.animation = "Stand"
    velocity = speed * velocity.normalized()

func _physics_process(_delta):
    get_input()
    velocity=move_and_slide(velocity)
```

Figure 6.19: *GDScript* du mouvement de *Mario* avec les 3 animations.

soit centrée sur le personnage, et le reste au fil de son déplacement. Ainsi, le déplacement de la caméra suit le déplacement du personnage. Nous allons utiliser un décor classique des jeux Mario, comme en Figure 6.20.

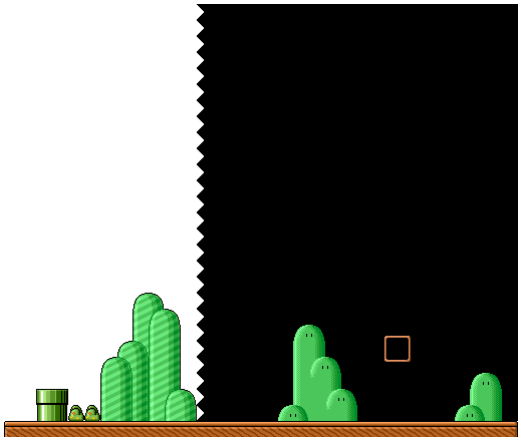


Figure 6.20: Décor utilisé.

Créez ainsi un nœud principal de type *Node2D*, renommé en *Main*, et attachez lui en fils:

- un nœud *Mario* de type *KynematicBody2D* auquel on attache un nœud fils de type *AnimatedSprite* dont on peut définir les animations Stand, Down et Walk comme en Section 6.2.3.
- un nœud de type *Sprite* pour ajouter le décor, en lui associant en texture l'image voulue, ici celle de la Figure 6.20.
- un nœud de type *Camera2D* (qui force l'écran à défiler suivant ce nœud), pour lequel il faut cocher le champ *Current*. Afin que la caméra soit centrée sur *Mario*, régler dans l'inspecteur la position x et y de la caméra à la même que celle de *Mario*.

Une fois cette arborescence établie, attachez un *GDScript* nommé *Mario.gd* en copiant/collant celui de la Section 6.2.3, puis en le modifiant comme en Figure 6.21.

Puisque le *GDScript* est attaché au nœud *Mario*, il faut repasser par la racine pour retrouver le nœud *Camera* et lui affecter une nouvelle position, d'où la ligne 10- du script.

```gdscript
extends KinematicBody2D

var speed : Vector2
var velocity : Vector2
var camera_node

func _ready():
    speed = Vector2(200,200)
    velocity = Vector2(0,0)
    camera_node = get_node("/root/Main/Camera2D")

func get_input():
    velocity = Vector2(0,0)
    var no_action_pressed = true
    if Input.is_action_pressed('ui_left'):
        no_action_pressed = false
        $AnimatedSprite.playing = true
        $AnimatedSprite.animation = "Walk"
        velocity = Vector2(-1,0)
        $AnimatedSprite.flip_h = false
    if Input.is_action_pressed('ui_right'):
        no_action_pressed = false
        $AnimatedSprite.playing = true
        $AnimatedSprite.animation = "Walk"
        velocity = Vector2(1,0)
        $AnimatedSprite.flip_h = true
    if Input.is_action_pressed('ui_down'):
        no_action_pressed = false
        $AnimatedSprite.animation = "Down"
    if no_action_pressed:
        $AnimatedSprite.animation = "Stand"
    velocity = speed * velocity.normalized()

func _physics_process(_delta):
    get_input()
    velocity=move_and_slide(velocity)
    camera_node.position.x = self.position.x
```

Figure 6.21: *GDScript* du déplacement de Mario avec caméra centrée.

Remarque. On aurait également pu utiliser deux scripts dissociés : le premier étant le même que celui de la Section 6.2.3 et toujours attaché au nœud *Mario*, et le second attaché au nœud *Camera* dans lequel on récupèrerait la position de *Mario* pour définir la nouvelle position de la caméra. Une autre solution est de garder le *GDScript* de la Section 6.2.3 attaché à *Mario* sans le modifier, et d'attacher un *GDScript* au nœud *Main* dans lequel on récupèrerait la position de *Mario* pour définir la position de la caméra. Vous pouvez alors par exemple vous amuser à rajouter les scintillements de la Section 6.1.2, avec le même décalage dans les animations.

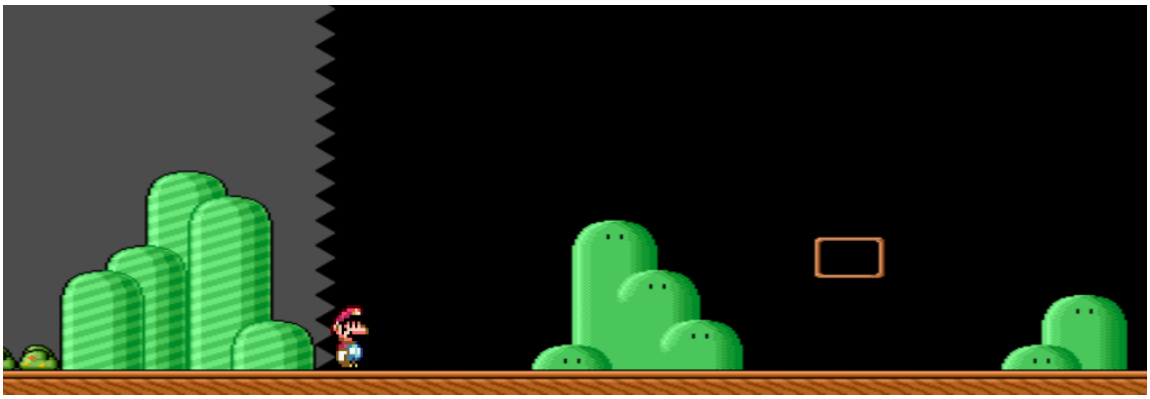


Figure 6.22: Résultat de Mario dans le décor.

Si on veut ensuite faire interagir certains éléments du décor avec *Mario*, notamment le tuyau vert à gauche qui agit comme un mur, on peut ensuite intégrer un nœud de type *StaticBody2D*, puis une forme de *CollisionShape2D* rectangulaire sur ce tuyau... et du coup également une sur le Mario, sinon il n'y aura aucune collision !

6.3. TILEMAPS

Un jeu vidéo à base de tuiles (*tiles*) est un type de jeu vidéo dans lequel l'aire de jeu est constituée de petites images carrées (ou moins souvent, rectangulaires, parallélogrammatiques

ou hexagonales) disposées sur une grille, appelée *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 appelée *sprite-sheet* (atlas de texture). 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. Tous les éléments d'un *sprite-sheet* sont en général disposés de manière plus visuelle dans une image appelée *tile-sheet*.

Les jeux basés sur des tuiles simulent généralement une vue de dessus ou une vue de profil et sont presque toujours en deux dimensions. Dans *Godot*, la grille sur laquelle on va insérer les tuiles pour dessiner l'aire de jeu est appelée une *TileMap*. **Il est possible d'ajouter des fonctionnalités à une tuile afin de gérer les collisions, occlusions, et de lui ajouter un schéma de déplacement**. Vous pourrez utiliser par la suite les *tile-sheets* créées durant le cours de création de ressources; en attendant nous utiliserons des *tile-sheets* libres de droits, par exemple via le site Kenney. Dans cette Section, nous utiliserons la *tile-sheet* ci-dessous pour créer des décors adaptés à notre Mario.

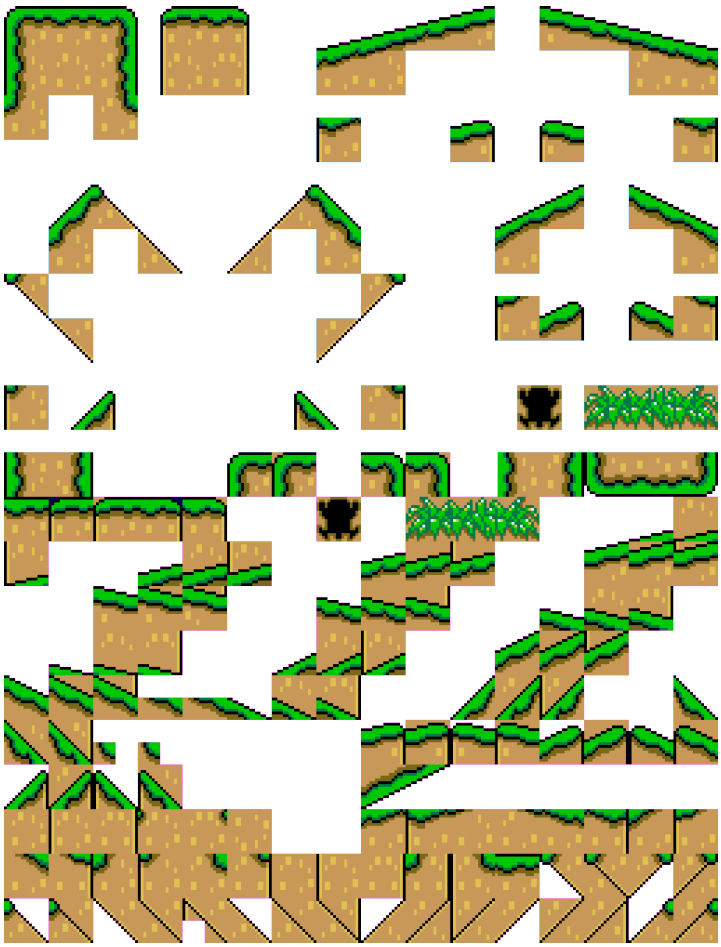


Figure 6.23: *Tile-sheet* utilisée.

6.3.1. Utiliser une *TileMap*. Nous allons créer un projet qui utilise une *TileMap*. Pour cela, il faut ajouter le *sprite-sheet* que nous voulons utiliser dans le gestionnaire de fichiers du projet. Avant de commencer, il faut sélectionner le *sprite-sheet* depuis le gestionnaire de fichiers du projet, puis aller dans l'onglet *Importer* pour désactiver la propriété *filtre*, et ré-importer l'image, voir Figure 6.24. Cette action est à réaliser à cause du comportement par défaut de *Godot* lors de l'import d'images 2D. Ce dernier applique un filtre par interpolation qui a des conséquences néfastes sur les bordures entre les tuiles. Or, l'utilisation d'un *tileset* impose que les tuiles voisines s'accordent parfaitement ensemble.

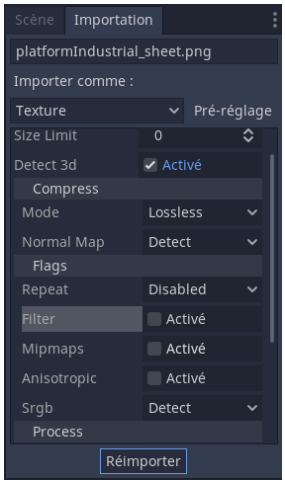


Figure 6.24: Ré-importation de la *sprite-sheet*.

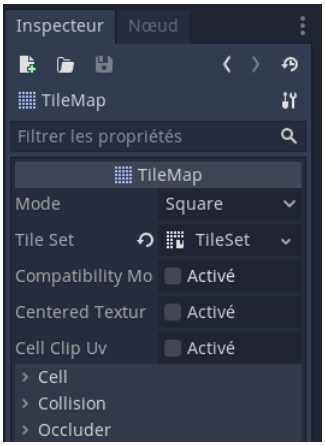


Figure 6.25: Création d'un nouveau *TileSet*

Il faut ensuite ajouter un nœud de type *TileMap* à notre scène. 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 créer un nouveau, voir Figure 6.25. Un *TileSet* est une ressource qui contient les données des tuiles: textures, informations de collision, etc. Lorsque le jeu tourne, la *TileMap* génère un unique objet en combinant toutes les tuiles.

Une fois le nouveau *TileSet* créé (ou chargé si on réutilise un issu d'un précédent projet *Godot*), en allant cliquer sur le *TileSet*, le panneau d'édition apparaît comme en Figure 6.26.

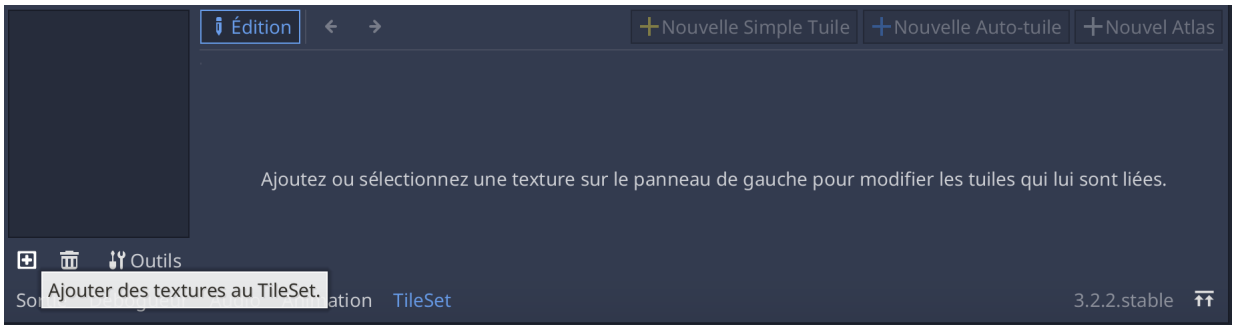


Figure 6.26: Panneau d'édition du *TileSet*.

En cliquant sur le bouton + en bas à gauche, on peut alors ajouter toutes les textures nécessaires à la création. Dans notre cas, il ne s'agit que d'un seul fichier. À partir des images chargées, nous pouvons créer nos tuiles, en cliquant sur le bouton *Nouvelle tuile simple*, voir

Figure 6.27. Pour faciliter la sélection des éléments, il est préférable de cliquer sur *Activer l'aimantation et afficher la grille* en sélectionnant *Nouvelle auto tuile*. Ceci sera très utile si la *tile-sheet* choisie est bien quadrillée, moins sinon... On peut aussi essayer de régler la taille du quadrillage à la min à partir de l'onglet *Cell* de l'inspecteur du nœud *TileMap*. **Dans notre cas, on choisira un quadrillage de taille** 32×32. Lorsque la grille apparaît, si le quadrillage est satisfaisant il suffit de sélectionner sur l'image l'ensemble des cellules qui constituent la tuile, et de re-cliquer sur *Nouvelle auto tuile* pour valider la création de la nouvelle tuile. On peut également contrôler la taille de la sélection choisie dans la *TileSet* à partir de l'onglet *Selected cell* de l'inspecteur. Un rectangle jaune entour l'ensemble des tuiles créées, voir Figure 6.27. Une fois les tuiles voulues sélectionnées, nous pouvons sortir de l'onglet d'édition et positionner alors les éléments voulus sur la grille comme en Figure 6.29. Les tuiles choisies sont également accessibles sur la droite de la scène, voir Figure 6.28. En cliquant sur l'une d'elles, on peut la placer à l'endroit voulu. En cas d'erreur, un clic droit sur la tuile ajoutée permet sa suppression.

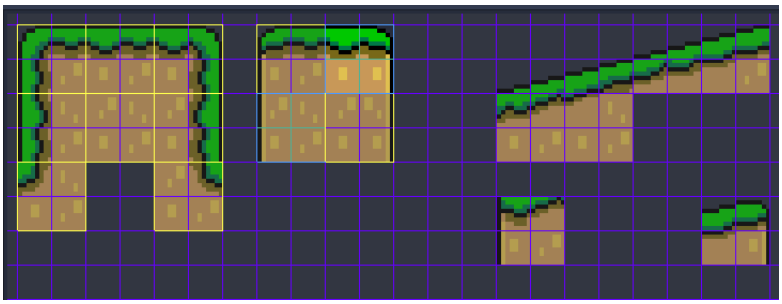


Figure 6.27: Création et sélection de tuiles.

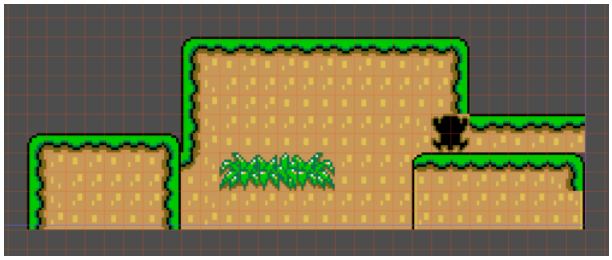


Figure 6.28: Positionnement des tuiles sélectionnées.

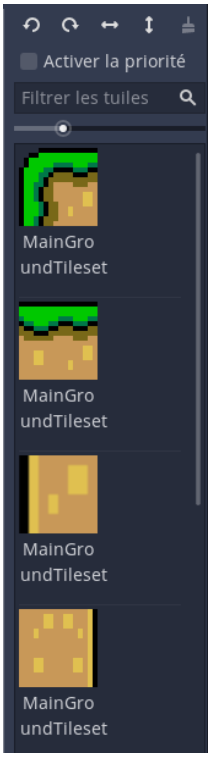


Figure 6.29: Accès aux tuiles créées.

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, voir Figure 6.30. Le bouton *Créer un nouveau rectangle* nous permet alors de définir une *CollisionShape2D* à notre tuile, comme nous le faisions précédemment pour des murs.

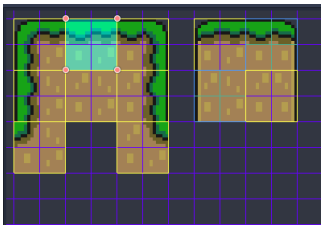


Figure 6.30: Création d'une tuile avec *CollisionShape*.

6.3.2. 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 fonction du contenu des tuiles voisines. Leur fonctionnement est très bien décrit sur le site officiel de *Godot*, voir cette page. Le choix des tuiles est contrôlé par des *bitmasks*. Les *bitmasks* peuvent être ajoutés en cliquant sur "Bitmask", puis en cliquant sur des parties des tuiles pour ajouter ou supprimer des bits dans le masque. En général, les bits sélectionnés dans un masque correspondront à des parties secondaires dans le gameplay, notamment les endroits creux du décor, où le personnage ne marchera jamais, qui pourront être générés automatiquement. Chaque fois que Godot met à jour une cellule en utilisant une *autotile*, il crée d'abord un modèle basé sur les cellules adjacentes qui sont déjà définies. Ensuite, il recherche dans l'*autotile* une seule tuile avec un *bitmask* correspondant au modèle créé. Si aucun masque binaire correspondant n'est trouvé, la tuile "icon" sera utilisée à la place. Si plus d'un masque binaire correspondant est trouvé, l'un d'entre eux sera sélectionné au hasard, en utilisant les priorités de la tuile.

Commençons par un exemple simple de génération de plateformes avec une forme choisie, à partir d'une image simple:

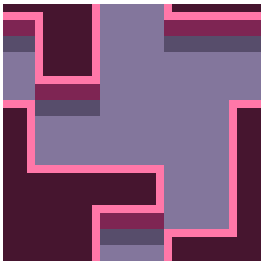


Figure 6.31: Image pour générer une *Autotile*.

Comme pour les tuiles classiques, il faut 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 sélectionner l'ensemble des tuiles qui vont permettre de définir le *bitmask* et qui seront utilisées dans la *map*. Dans notre cas, nous sélectionnons toute l'image comme unique tuile comme en FIgure 6.32, en prenant soin de régler dans l'onglet *Snap options* de la *TileSet* les paramètres de *Step* à 8 et 8 et dans l'onglet *Selected Tile* les paramètres *Subtile Size* à 8 et 8, pour calquer les dimensions de l'image. Il faut ensuite dessiner les tuiles qui serviront à définir le bitmask, et nous sélectionnerons les tuiles en dehors de la plateforme comme en Figure 6.33.

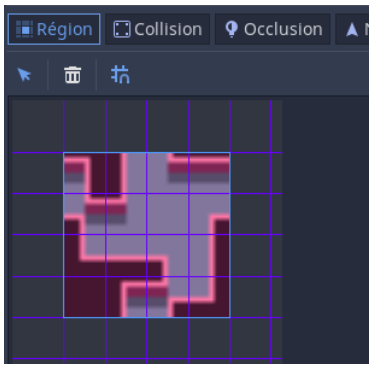


Figure 6.32: Création d'une nouvelle tuile automatique.

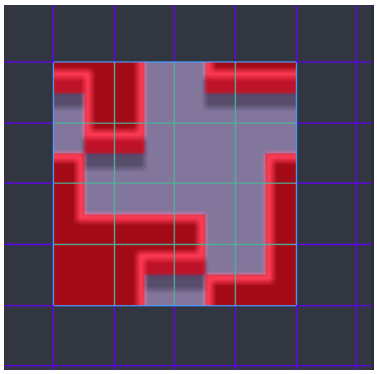


Figure 6.33: Sélection du *bitmask*.

On peut ensuite sélectionner le nœud *TileMap*, et commencer à dessiner des tuiles à partir de notre unique tuile. En dessinant certains contours, les tuiles vont s'adapter automatiquement pour générer des formes comme par exemple en Figure 6.34.

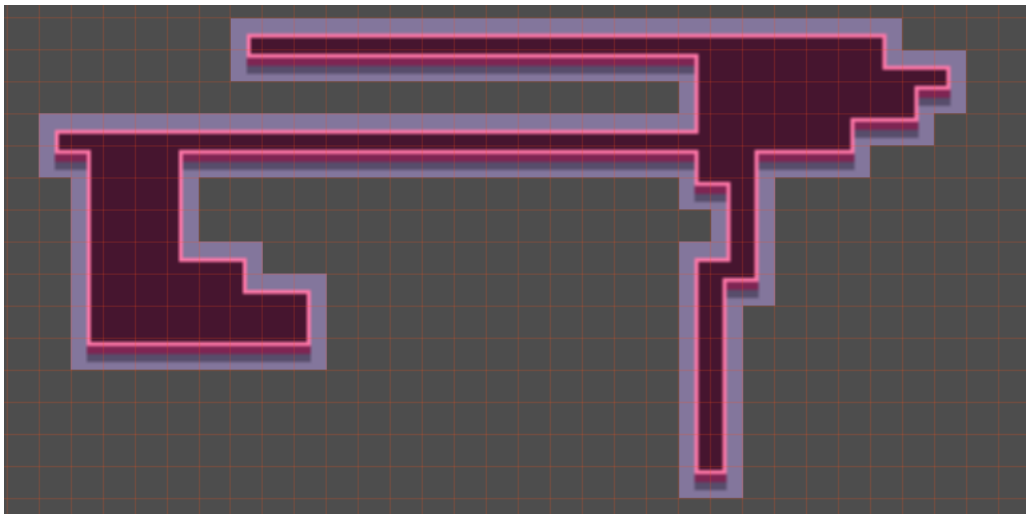


Figure 6.34: Génération de plateforme automatique.

Notons que *Godot*offre la possibilité de créer d'autres types de *bitmask*, notamment des 3 × 3, mais cela imposerait un plus grand nombre de tuiles utilisables. Le principe est en revanche le même, vous pouvez essayer ! N'oubliez également pas de créer des formes de collisions sur les tuiles qui seront utilisées. Dans notre cas, on pourra utiliser une *PolygonShape2D* pour matcher la forme de collision avec les contours de la plateforme.

6.4. PARALLAX BACKGROUND

Dans cette Section, l'objectif est de placer le personnage de Mario de la Section 6.2.4 dans une *TileMap* comme en Section 6.3, et de rajouter un fond composé d'une couche se déplaçant moins vite que le décor de premier plan. L'image utilisée en fond est la suivante, présentée en Figure 6.35, de dimension 2048 × 1728.

Pour créer cette scène, enregistrez depuis Moodle le projet intitulé *Parallax* dans votre répertoire de projets *Godot*, combinant les scènes *Mario.tscn* de la Section 6.2.4 et *TileMap.tscn*

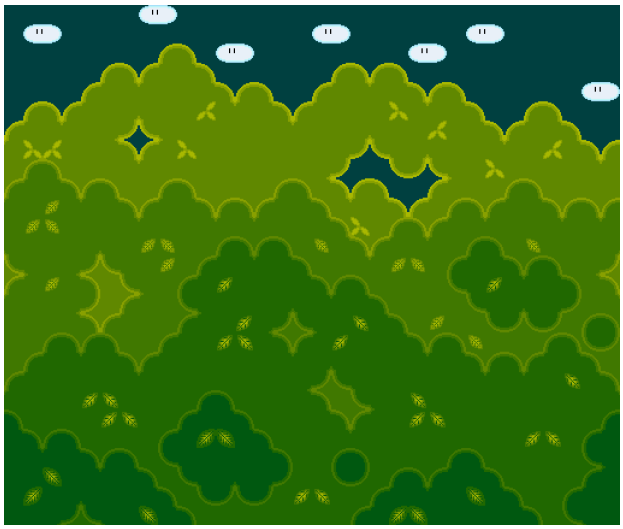


Figure 6.35: Image en fond.

de la section 6.3. Voici ensuite la procédure à suivre:

1. Créer ensuite une nouvelle scène intitulée *Parallax.tscn*, dans laquelle il faut ajouter un nœud de type *ParallaxBackground*, puis un nœud enfant de type *ParallaxLayer* et un *Sprite* dans lequel on ajoutera comme texture l'image de la Figure 6.35.

2. Dans le champ *Motion* du nœud *ParallaxLayer*, fixer la valeur de *Scale* à 0.5 en x et la valeur de *Mirroring* à 2048 en x, ceci modifiera la vitesse de défilement, et permettra à l'image de se répéter une fois arrivé au bout.

3. Dans la scène principale, instancier la scène *Parallax.tscn*.

Vous pouvez ensuite exécuter la scène principale, pour constater la différence de vitesse de déplacement du fond, ainsi que les bonnes transitions du *Background* avec le *Mirroring*.

Chapitre 7

Physique du mouvement d'un *Sprite*

7.1. Amortir les déplacements

Dans cette Section, on souhaite définir le déplacement d'un Sprite de Mario vers la gauche et la droite comme en Section 6.2.1, mais on souhaite également pouvoir amortir son déplacement. L'amortissement est réalisé avec une interpolation linéaire, il permet d'obtenir une accélération et une décélération progressives au début et à la fin du déplacement.

Créer un nouveau projet, ajouter un nœud principal de type *Node2D* auquel on ajoute un nœud fils de type *KinematicBody2D*, puis un Sprite en fils auquel on ajoute la texture de Mario de la section 6.2.1. Attacher le GDScript au nœud *Mario* comme présenté en Figure 7.1.

```
1  extends KinematicBody2D
2
3  var velocity : Vector2
4  var WALK_LEFT : Vector2
5  var WALK_RIGHT : Vector2
6
7  func _ready():
8      velocity = Vector2(0,0)
9      WALK_LEFT = Vector2(-100,0)
10     WALK_RIGHT = Vector2(100,0)
11
12
13 func get_input():
14     if Input.is_action_pressed("ui_right"):
15         velocity = velocity + WALK_RIGHT
16         $Sprite.flip_h = true
17     if Input.is_action_pressed("ui_left"):
18         velocity = velocity + WALK_LEFT
19         $Sprite.flip_h = false
20     velocity.x= lerp(velocity.x,0,0.2)
21
22 func _physics_process(delta):
23     get_input()
24     velocity=move_and_slide(velocity)
```

Figure 7.1: GDScript d'amortissement du mouvement.

Contrairement à la Section 6.2.1, la variable *velocity* n'est pas normalisée, et la coordonnée *velocity.x* varie entre -100 et 100. La fonction *lerp* permet de faire une interpolation linéaire de la variable x (abscisse) du vecteur *velocity*: c'est elle qui amortir les déplacements en faisant

évoluer *velocity* vers 0 avec le temps. On ne modifie pas la variable y (ordonnée) de ce vecteur. On peut accéder facilement de cette manière à une coordonnée d'un *Vector2* pour la modifier; en *VisualScript*, voici l'équivalent permettant de modifier une coordonnée:

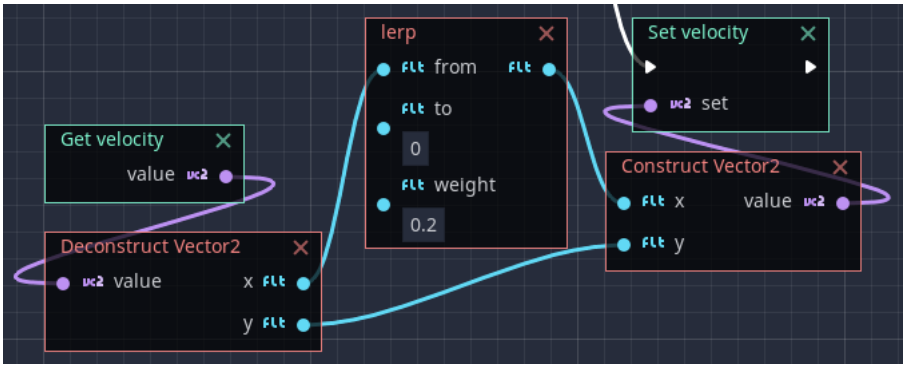


Figure 7.2: Modification d'une coordonnée d'un vecteur en VisualScript

7.2. Collisions 2D

7.2.1. Détecter les collisions entre *Sprite* et *TileMap*. Récupérer les scènes *Mario.tscn*, *Tilemap.tscn* et *Parallax.tscn* de la Section 6.4, ainsi que tous les assets nécessaires (les images d'animation de Mario, le fond d'écran et la *tile-sheet*), et les copier dans le répertoire Courant du nouveau projet. On peut alors instancier ces scènes dans la scène principale pour les réutiliser. Modifier ensuite les tuiles de la *TileMap* pour créer un décor autour du Mario avec une bordure gauche et une bordure droite comme en Figure 7.3.

Figure 7.3: *TileMap* avec bordures.

Pour obtenir une délimitation vers les deux bordures, il suffit pour les deux tuiles correspondant à des bordures montantes d'ajouter des rectangles de collision aux tuiles comme en Figure 7.4.

Il faut aussi bien veiller à ce que le *Mario* ait lui-même une forme de collision. On peut garder une forme de collision rectangulaire, ou essayer d'être plus précis en définissant une forme de collision composée de deux cercles comme en Figure 7.5. La définition de formes de collision qui ne dépassent pas du sprite permet d'éviter la détection de fausses colisions.

Figure 7.4: Ajout des formes de collision sur les tuiles.

Selon le jeu, les décors et l'interaction souhaitée, il faut donc choisir le niveau de précision de détection de collision que l'on souhaite.

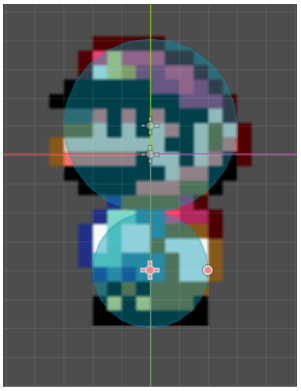


Figure 7.5: Formes de collision circulaires.

7.2.2. Coordonner animation, collision et supression. Créer un nouveau projet reprenant toute la scène principale précédente, à savoir le *Mario.tscn*, le *TileMap.tscn* et le *Parallax.tscn*, ainsi que tous les assets nécessaires. On va vouloir ensuite ajouter deux pièces comme en Figure 7.6 entre les deux bordures comme en Figure 7.7 et faire en sorte que la pièce disparaît quand Mario entre en collision avec cette dernière. Créer un nœud principal de type *Node2D*, renommé en Main, puis instancier en scène enfant *Mario.tscn*,*TileMap.tscn* et *Parallax.tscn*.

Figure 7.6: Image de la pièce.

Figure 7.7: Scène de Mario avec les deux pièces.

- Créer une scène *Coin.tscn* dont le nœud principal est un nœud de type *Area2D*, délimitant une zone de la scène. Lui ajouter un nœud enfant *AnimatedSprite* et ajouter une animation *Turn* composée des quatre images en Figure 7.2.2. Ajouter également un nœud enfant de type *CollisionShape2D*, et matcher la forme de la pièce hors de l'animation. Activer ensuite le champ *Playing* du *AnimatedSprite*.

Figure 7.8: Animation *Turn* de la pièce.

- Ensuite, connecter un signal au nœud de type *Area2D* en double-cliquant sur la fonction *body_entered*. Cela a pour effet d'ajouter un script au nœud principal de la scène *Coin.tscn*, dans lequel apparaît la fonction *_on_Coin_body_entered*. Ajouter à l'intérieur de cette fonction la commende *queue_free()* permettant la suppression de l'objet courant, comme en Figure 7.9.

```
1  extends Area2D
2
3  func _on_Coin_body_entered(body):
4      queue_free()
```

Figure 7.9: *GDScript* de suppression de la pièce.

Vous pouvez ensuite exécuter et voir que lorsque Mario entre dans la *CollisionShape2D* délimitée par une pièce, celle-ci se supprime.

7.2.3. Ajout d'un scintillement. On va maintenant repartir de la scène précédente, en ajoutant une animation de scintillement lorsque *Mario* entre en collision avec la pièce: la pièce disparaît, puis des étoiles scintilles puis disparaissent avec l'animation présentée en Figure 7.2.3.

Ajouter au nœud *AnimatedSprite* de la scène *Coin.tscn* l'Animation *Firework* définie par les images présentées en figure 7.2.3, qui remplaceront l'animation *Turn* de la pièce lorsque Mario entrera en collision.

Figure 7.10: Animation *Firework* de la pièce.

Modifier ensuite le script *Coin.gd* comme présenté en Figure 7.11. On ajoute une variable booléenne qui est initialisée à *false* et va passer à *true* lorsque Mario entre en collision avec la pièce, afin de lancer la fin de l'animation. Lorsque le signal *on_Coin_body_entered* est déclenché, on bascule vers l'animation *Firework*. Dans la fonction *physics_process*, on décrète que l'animation du *Firework* s'arrêtera au bout de 4 frames (on aurait pu choisir une valeur plus grande), où on supprime alors la pièce entièrement. Les commandes *randf()* permettent de choisir un nombre réel aléatoire entre 0 et 1: les lignes 10 à 13 permettent d'ajouter de l'aléa dans l'exécution de l'animation. On aura parfois une symétrie horizontale, parfois une symétrie verticale, parfois les deux, parfois rien...

```gdscript
extends Area2D

var ends : bool

func _ready():
	ends = false

func _on_Coin_body_entered(body):
	$AnimatedSprite.animation="Firework"
	if randf() > 0.5:
		$AnimatedSprite.flip_h=true
	if randf() > 0.5:
		$AnimatedSprite.flip_v=true
	$AnimatedSprite.frame = 0
	ends=true

func _physics_process(delta):
	if ends == true:
		if $AnimatedSprite.frame == 4:
			queue_free()
```

Figure 7.11: *GDScript* de suppression de la pièce avec scintillement.

7.2.4. Mouvement automatique et détection de bords de plateforme. On veut maintenant placer notre personnage de *Mario* dans une *TileMap* contenant 5 plateformes comme en figure 7.12. On va ensuite placer un personnage de *Koopa* sur les quatre plateformes en hauteur, et on voudra que ces *Koopa* se déplacent automatiquement sur leur plateforme en faisant une ronde de gauche à droite, et en réalisant un demi-tour lorsqu'ils détectent le bord de la plateforme.

Pour limiter les déplacements d'un *Koopa*, on va placer autour de lui deux zones *Area2D* au niveau des bords des plateformes, comme en Figure 7.13. Chaque *Koopa* sera ainsi associé à deux zones qui déclenchent un signal pour réaliser un demi-tour et inverser le sens de déplacement.

Figure 7.12: Scène avec deux *Koopa*.

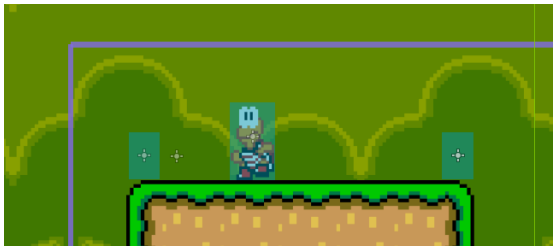


Figure 7.13: Ajoute de deux *Area2D* autour de chaque *Koopa*.

- Créer un nouveau projet, dans lequel on ajoutera les scènes *Mario.tscn*, *Parallax.tscn* et *TileMap.tscn* ainsi que tous les assets nécessaires. Modifier la *TileMap* pour obtenir une map semblable à celle de la Figure 7.12.

- Créer une scène *Koopa.tscn* dont le nœud principal est un *KinematicBody2D*, puis lui ajouter en nœud enfant un *AnimatedSprite* auquel on associera les animations *Turn* et *Walk* au *Koopa* comme présentées en Figures 7.14 et 7.15.

Figure 7.14: Animation *Turn*.

Figure 7.15: Animation *Walk*.

- Ajouter un script *Koopa.gd* au nœud principal de la scène *Koopa.tscn* comme en Figure 7.16. La variable *turning* permet de détecter lorsque le *Koopa* doit se tourner: tant qu'elle est positive, on va changer l'animation en *Turn*. Sinon, l'animation par défaut est *Walk*, et on suppose que le *Koopa* avance vers la droite avec une vélocité constante $(100, 0)$ (et dans ce cas on active le flip horizontal) et vers la gauche avec une vélocité constante $(-100, 0)$ (et dans ce cas on désactive le flip horizontal).

- Ajouter à gauche du premier *Koopa* une *Area2D* nommée *Area2D-L*, puis depuis l'onglet Nœud de l'inspecteur, connecter le signal *body_entered* au **premier** Koopa. La fonction *_on_Area2D_body_entered* s'ajoute alors au script *Koopa.gd*, compléter cette fonction comme en Figure 7.17. L'idée est que lorsque *Koopa* entre en collision avec une de ces zones, sa vélocité soit inversée (afin qu'il change de direction) ainsi que son flip horizontal. On règle également la variable *turning* à 20 pour produire une pause lorsque le *Koopa* se retourne.

```
1  extends KinematicBody2D
2
3  var speed : Vector2
4  var velocity : Vector2
5  var turning : int
6
7  func _ready():
8      speed=Vector2(100,0)
9      velocity=Vector2(100,0)
10     turning=1
11
12 func _physics_process(delta):
13     if turning>0:
14         $AnimatedSprite.animation="Turn"
15         turning -= 1
16     else:
17         $AnimatedSprite.animation="Walk"
18     if velocity.x>0:
19         $AnimatedSprite.flip_h=true
20     else:
21         $AnimatedSprite.flip_h=false
22     velocity = speed * velocity.normalized()
23     if not(turning>0):
24         velocity = move_and_slide(velocity)
```

Figure 7.16: Mouvement du *Koopa*.

```
12 func _on_Area2DL_body_entered(_body):
13     velocity.x= -velocity.x
14     turning = 20
```

Figure 7.17: Changement de vélocité en cas de collision avec l'*Area2D-L*.

7.3. SAUTS, GESTION DE LA GRAVITÉ

7.3.1. Physique des objets. On a déjà précédemment vu les effets de la gravité en *Godot*, notamment sur les objets de type *RigidBody2D* comme nos balles du Chapitre 4. On a également modifié certains paramètres physiques sur ces balles, comme les forces exercées, leur capacité de rebond, etc.

En *Godot*, on peut définir les contraintes imposées au mouvement des corps avec ou sans collision, avant ou après une collision, en pesanteur ou en apesanteur. Dans la Section précédente, nous avons étudié les collisions en 2D pour des *Area2D* et des *KinematicBody2D*. Mais en fonction de notre jeu, les types d'objets peuvent être différents. Avec les fonctionnalités du moteur physique, nous laissons au moteur de jeu le contrôle de certaines réactions physiques. Lorsque les objets ne bougent pas, il devient plus pertinent d'utiliser des nœuds de type *StaticBody2D*. Lorsque les objets bougent mais ne sont pas contrôlés directement par le joueur, on utilisera plutôt des *RigidBody2D*. Le fait que les objets ne bougent pas signifie que le moteur physique ne cherche pas à les faire bouger, mais on peut tout de même les faire bouger en leur associant une vélocité. Les valeurs *constant_linear_velocity* et *constant_angular_velocity* définissent respectivement la vitesse linéaire et la vitesse angulaire des objets *StaticBody2D*. Dans l'onglet *PhysicsMaterial*, les champs *Mass*, *Friction* et *Bounce* définissent les propriétés physiques des corps: le poids, les forces de frottement, l'élasticité. Il est également possible de définir des propriétés physiques ambiantes dans les propriétés du projet. On déplace un *StaticBody2D* par application de forces ou par collision avec d'autres objets.

Pour les *RigidBody2D*, il existe quatre comportements prédéfinis. Le type *Rigid* réagit aux collisions et aux forces, c'est le comportement par défaut. Le type *Static* se comporte comme un *StaticBody2D* Le type *Character* est un *RigidBody2D* sans *AngularVelocity* (c'est-à-dire sans mouvement de rotation). Le type *Kinematic* se comporte comme un *KinematicBody2D*.

7.3.2. Sauts et chutes. Dans cette Section, on va reprendre les scènes *Tilemap.tscn*, *Mario.tscn* et *Parallax.tscn* de la Section 6.4 et placer *Mario* dans une *tilemap* correspondant à une plateforme avec deux hauteurs de sol, comme présenté en Figure 7.18. On souhaite que le déplacement de Mario soit limité par les bordures, et que lorsque l'on appuie sur la flèche du haut, le *Mario* saute selon une jolie courbe, avec une vitesse nulle à sa hauteur maximale pendant un instant. De plus, on souhaite que Mario puisse chuter de la plateforme, avec une accélération vers le bas.

Figure 7.18: *Tilemap* avec différentes hauteurs de plateforme.

- Créer un nœud principal de type *Node2D*, et lui instancier les scènes *Mario.tscn*, *Tilemap.tscn* et *Parallax.tscn*.
- Dans le nœud *AnimatedSprite* de la scène *Mario.tscn*, ajouter deux animations *Jump* et *Fall* présentées en Figures 7.19 et 7.3.2.

Figure 7.19: Animation *Jump*.

Figure 7.20: Animation *Fall*.

- Modifier la fonction *get_input* du script *Mario.gd* comme présenté en Figure 7.3.2. La première partie de cette fonction fixe l'animation à *Stand* lorsque aucune touche n'est pressée. La deuxième partie définit, comme dans les projets précédents, l'animation en fonction des touches clavier pressées: selon l'input, on ajoute la variable *GO_LEFT* ou *GO_RIGHT* à *velocity*. La troisième partie de la fonction applique en permanence une gravité à *velocity* en lui ajoutant le vecteur *DOWN*, qui est orienté vers le bas. La réalisation d'un saut est conditionné par sur la flèche du haut ainsi que la fonction *is_on_floor* qui permet de s'assurer que le personnage est sur le sol au moment du saut. Supprimer cette fonction autorise de cumuler plusieurs sauts d'affilée. L'utilisation de *is_on_floor* implique de définir la direction des collisions en utilisant la fonction *move_and_slide*, avec un second paramètre précisant la direction vers le haut.

```
1  extends KinematicBody2D
2
3  var speed : Vector2
4  var velocity : Vector2
5  var GO_LEFT : Vector2
6  var GO_RIGHT : Vector2
7  var DOWN : Vector2
8  var JUMP : Vector2
9
10 func _ready():
11     speed = Vector2(200,200)
12     velocity = Vector2(0,0)
13     GO_LEFT = Vector2(-200,0)
14     GO_RIGHT= Vector2(200,0)
15     JUMP=Vector2(0,-1000)
16     DOWN=Vector2(0,30)
```

Figure 7.21: Variables du script.

```
func get_input():
    if (not(Input.is_action_pressed("ui_right")) and
    not(Input.is_action_pressed("ui_left"))):
        $AnimatedSprite.animation="Stand"
    if Input.is_action_pressed("ui_right"):
        velocity = velocity + GO_RIGHT
        $AnimatedSprite.animation="Walk"
        $AnimatedSprite.flip_h=true
    if Input.is_action_pressed("ui_left"):
        velocity = velocity + GO_LEFT
        $AnimatedSprite.animation="Walk"
        $AnimatedSprite.flip_h=false
    if Input.is_action_pressed("ui_down") and is_on_floor():
        $AnimatedSprite.animation="Down"
    velocity=velocity+DOWN
    if Input.is_action_pressed("ui_up") and is_on_floor():
        velocity=velocity+JUMP
```

Figure 7.22: Nouvelle fonction *get_input*.

- Ajouter la fonction *set_animation* permettant de fixer l'animation à *Jump* lorsque Mario saute (*i.e.* sa vélocité en y est strictement négative); ou à *Fall* lorsque Mario chute (*i.e.* sa vélocité en y est strictement positive). Modifier ensuite la fonction *physics_process* précédente comme présenté en 7.23.

```
func set_animation():
    if velocity.y > 0:
        $AnimatedSprite.animation="Fall"
    if velocity.y < 0:
        $AnimatedSprite.animation="Jump"

func _physics_process(_delta):
    get_input()
    velocity = move_and_slide(velocity,Vector2(0,-1))
    velocity.x = lerp(velocity.x,0,0.2)
    set_animation()
    $AnimatedSprite.offset=$AnimatedSprite.position
```

Figure 7.23: Fonctions *set_animation* et *physics_process*.

7.4. Pentes avec gravité

On place maintenant le *Mario* dans une *Tilemap* correspondant à une plateforme avec une pente, comme en Figure 7.24. Pour créer cette pente, il faut ajouter des nouvelles tuiles à notre *TileMap* correspondant à la pente vers la droite, et ajouter des formes de collision qui matchent le mieux possible la surface de la tuile qui sera un sol, comme en Figure 7.4.

Figure 7.24: Nouvelle *TileMap* avec pente.

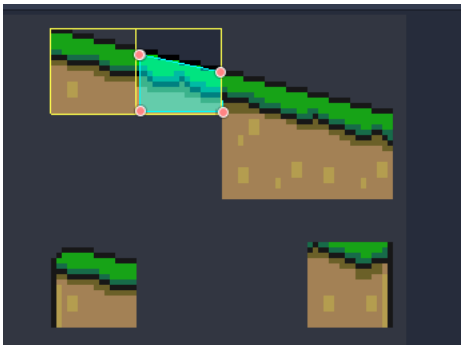


Figure 7.25: Création de la pente.

Si on exécute le jeu avec le *GDScript* précédent, on constate alors les problèmes suivants:

(1) Si on arrête Mario sur la pente, il glisse dans le sens de la pente.

(2) Quand Mario descend la pente, il est animé avec *Fall*.

(3) Quand Mario monte sur la pente, il est animé avec *Jump*.

(4) Quand Mario est sur la pente, il est légèrement au dessus du sol.

Pour résoudre (2) et (3), il suffit d'ajouter une conditon *not(is_on_floor)* dans la fonction *set_animation*. Pour résoudre (1), on ajoute un troisième paramètre à la fonction *move_and_slide* dans la fonction *physics_process*. Ce paramètre est nommé *stop_on_slope*: fixé à True, il permet à Mario de ne pas glisser dans la pente lorsque sa vélocité est nulle. L'application de la gravité créant un vecteur vélocité non nul, on ajoute une condition *is_on_floor* dans la partie de la fonction *get_input* associée à l'animation *Stand* (c'est-à-dire quand aucune touche n'est

pressée). Ceci permet d'avoir une vélocité nulle dans ce cas, et donc de ne plus glisser sur la pente.

```
func set_animation():
    if not(is_on_floor()):
        if velocity.y > 0:
            $AnimatedSprite.animation="Fall"
        if velocity.y < 0:
            $AnimatedSprite.animation="Jump"
```

Figure 7.26: Ajout d'une condition dans *set_animation*.

```
func get_input():
    if (not(Input.is_action_pressed("ui_right")) and
    not(Input.is_action_pressed("ui_left"))):
        $AnimatedSprite.animation="Stand"
        if is_on_floor():
            velocity=Vector2(0,0)
    if Input.is_action_pressed("ui_right"):
        velocity = velocity + GO_RIGHT
        $AnimatedSprite.animation="Walk"
        $AnimatedSprite.flip_h=true
    if Input.is_action_pressed("ui_left"):
        velocity = velocity + GO_LEFT
        $AnimatedSprite.animation="Walk"
        $AnimatedSprite.flip_h=false
    if Input.is_action_pressed("ui_down") and is_on_floor():
        $AnimatedSprite.animation="Down"
    velocity=velocity+DOWN
    if Input.is_action_pressed("ui_up") and is_on_floor():
        velocity=velocity+JUMP
```

Figure 7.27: Nouvelle fonction *get_input*.

```
func _physics_process(_delta):
    get_input()
    velocity = move_and_slide(velocity,Vector2(0,-1),true)
    velocity.x = lerp(velocity.x,0,0.2)
    set_animation()
    $AnimatedSprite.offset=$AnimatedSprite.position
```

Figure 7.28: Nouvelle fonction *physics_process*.

Pour résoudre (4), on ajoute au début de la fonction *get_input* un décalage en y (pouvant être variable selon la position de vos objets) si le sol sous Mario n'est pas horizontal, ce qu'on peut vérifier en regardant si la fonction *get_floor_normal* renvoie un vecteur différent de $(0, -1)$. On obtient alors la dernière fonction *get_input* présentée en Figure 7.29.

```
func get_input():
    $AnimatedSprite.position=Vector2(0,0)
    if (is_on_floor() and get_floor_normal() != Vector2(0,-1)):
        $AnimatedSprite.position=Vector2(0,3)
    if (not(Input.is_action_pressed("ui_right")) and
    not(Input.is_action_pressed("ui_left"))):
        $AnimatedSprite.animation="Stand"
        if is_on_floor():
            velocity=Vector2(0,0)
    if Input.is_action_pressed("ui_right"):
        velocity = velocity + GO_RIGHT
        $AnimatedSprite.animation="Walk"
        $AnimatedSprite.flip_h=true
    if Input.is_action_pressed("ui_left"):
        velocity = velocity + GO_LEFT
        $AnimatedSprite.animation="Walk"
        $AnimatedSprite.flip_h=false
    if Input.is_action_pressed("ui_down") and is_on_floor():
        $AnimatedSprite.animation="Down"
    velocity=velocity+DOWN
    if Input.is_action_pressed("ui_up") and is_on_floor():
        velocity=velocity+JUMP
```

Figure 7.29: Nouvelle fonction *get_input*.

7.5. RAMASSER ET LANCER UN OBJET

7.5.1. Ramasser une carapace de Koopa. Dans cette Section, l'objectif est d'apprendre à ramasser et lancer des objets. On va pour ce faire placer une carapace de *Koopa* sur la même plateforme que notre *Mario*. On voudra que lorsque Mario est juste à côté de la carapace, il puisse la ramasser avec la touche A, puis se déplacer avec la carapace dans les mains comme en Figure 7.30.

Figure 7.30: Mario avec une carapace de Koopa dans les mains.

Voici ci-dessous la procédure à suivre pour cette scène:

- Reprendre les scènes *Mario.tscn*, *Parallax.tscn* et *Tilemap.tscn* précédentes, et les instancier en fils du nœud principal de type *Node2D*.
- **Dans la scène principale** (afin de pouvoir connecter les *Area2D* qu'on va créer au Mario), ajouter un nœud de type *KinematicBody*, renommé en *Shell*.
- Pour limiter la prise en main de la carapace, on place des *Area2D* à gauche et à droite de la carapace (nommées *LeftCatch* et *RightCatch*) et une *Area2D* sur la carapace comme présenté en Figure 7.31. On peut également lui ajouter une forme de collision, ici de préférence de forme elliptique. L'arborescence de la scène est ainsi présentée en Figure 7.32.

Figure 7.31: Placement des *Area2D* autour de la carapace.

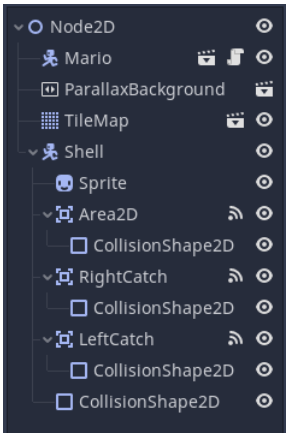


Figure 7.32: Arborescence de la scène.

- Sur les zones *RightCatch*, *LeftCatch* et *Area2D*, connecter les signaux *body_entered* et *body_exited* au nœud Mario. Ceci fera apparaître les fonctions concernées dans le script *Mario.gd*. On va vouloir que l'entrée dans ces zones déclenche un signal qui mettra un jour une variable booléenne; et quand les variables auront les bonnes valeurs et que le joueur appuie sur A, la carapace passera dans les mains de Mario.

- Dans la scène *Mario.tscn*, ajouter les animations *Stand-Hold* et *Walk-Hold* comme présenté en Figure 7.33 et 7.34.

Figure 7.33: Animation *Stand-Hold*.

Figure 7.34: Animation *Walk-Hold*.

- Dans le script *Mario.gd*, ajouter les variables de type booléen suivantes:

  1. *can_catch* qui va détecter quand Mario est en collision avec *LeftCatch* ou *RightCatch*, autrement dit quand Mario est assez proche de la carapace pour l'attraper;
  2. *holding_shell* qui va tester si Mario porte la carapace;
  3. *overlapping_shell* qui teste si Mario est en collision avec *Area2D*;

  On va également récupérer une variable de type *node*, qui nous permet d'accéder à un nœud du projet qui n'est pas dans la scène *Mario.tscn* où se situe le script *Mario.gd*, ici le nœud *Shell* dont on voudra modifier la position lorsque Mario ramasse la carapace. La définition des variables est présentée en Figure 7.35.

```
var can_catch: bool
var holding_shell: bool
var looking_left: bool
var overlapping_shell: bool

onready var node_shell = get_node("/root/Node2D/Shell")
```

Figure 7.35: Nouvelles variables de *Mario.gd*.

- Modifier les fonctions associées aux signaux de *LeftCatch*, *RightCatch* et *Area2D* comme présenté en Figure 7.36.

- Définir des fonctions *anim_hold*, *anim_walk_left* et *anim_walk_right* qui vont activer les animations adéquates en fonction de la valeur des variables booléennes, comme présenté en Figure 7.37.

```
func _on_RightCatch_body_entered(_body):
    can_catch = true

func _on_LeftCatch_body_entered(_body):
    can_catch = true

func _on_RightCatch_body_exited(_body):
    can_catch = false

func _on_LeftCatch_body_exited(_body):
    can_catch = false

func _on_Area2D_body_entered(_body):
    overlapping_shell=true

func _on_Area2D_body_exited(_body):
    overlapping_shell=false
```

Figure 7.36: Signaux des diverses Area2D.

```
func animate_hold():
    if holding_shell == true:
        $AnimatedSprite.animation="Stand-Hold"
    else:
        $AnimatedSprite.animation="Stand"

func animate_walk_left():
    $AnimatedSprite.flip_h=false
    looking_left=true
    if holding_shell==true:
        $AnimatedSprite.animation="Walk-Hold"
    else:
        $AnimatedSprite.animation="Walk"

func animate_walk_right():
    $AnimatedSprite.flip_h=true
    looking_left=false
    if holding_shell==true:
        $AnimatedSprite.animation="Walk-Hold"
    else:
        $AnimatedSprite.animation="Walk"
```

Figure 7.37: Fonctions qui déclenchent les animations adéquates.

- Définir une fonction *hold_shell* qui va régler la position de la carapace en fonction de celle du Mario lorsque Mario porte la carapace, puis modifier la fonction *physics_process* en ajoutant un appel à *hold_shell* comme présenté en Figure 7.38.

- Modifier la fonction *get_input* en prenant en compte les nouvelles fonctions d'animation, et en indiquant que lorsque la touche A est pressée et que Mario peut attraper la carapace, alors il l'attrape. La fonction est présentée en Figure 7.39.

```
func hold_shell():
    if holding_shell==true:
        if looking_left==true:
            node_shell.position = node_mario.position + Vector2(-44,0);
        else:
            node_shell.position = node_mario.position + Vector2(42,0);


func _physics_process(_delta):
    get_input()
    velocity = move_and_slide(velocity,Vector2(0,-1),true)
    velocity.x = lerp(velocity.x,0,0.2)
    hold_shell()
    set_animation()
    $AnimatedSprite.offset=$AnimatedSprite.position
```

Figure 7.38: Fonction *hold_shell*.

```
func get_input():
    $AnimatedSprite.position=Vector2(0,0)
    if (is_on_floor() and get_floor_normal() != Vector2(0,-1)):
        $AnimatedSprite.position=Vector2(0,3)
    if (not(Input.is_action_pressed("ui_right")) and
    not(Input.is_action_pressed("ui_left"))):
        animate_hold()
        if is_on_floor():
            velocity=Vector2(0,0)
    if Input.is_action_pressed("ui_right"):
        velocity = velocity + GO_RIGHT
        animate_walk_right()
    if Input.is_action_pressed("ui_left"):
        velocity = velocity + GO_LEFT
        animate_walk_left()
    if Input.is_action_pressed("ui_down") and is_on_floor():
        $AnimatedSprite.animation="Down"
    velocity=velocity+DOWN
    if Input.is_action_pressed("ui_up") and is_on_floor():
        velocity=velocity+JUMP
    if Input.is_action_just_pressed("Catch"):
        if can_catch==true:
            holding_shell=true
```

Figure 7.39: Nouvelle fonction *get_input*.

Notez que le décalage de la position de la carapace du vecteur $(-44, 0)$ ou $(42, 0)$ par rapport au Mario dépend des réglages de votre projet et des dimensions de votre objet; à vous de trouver la position qui semble la meilleure...

Remarque importante : Vous pouvez alors enregistrer la scène *Shell.tscn* pour la réutiliser plus tard. Toutefois, si vous exécutez le jeu, Mario ne ramassera plus la carapace puisque les signaux associés aux *Area2D* ont été déconnectés... Pour résoudre ce problème, il faudra rendre la scène instanciée locale par un clic droit sur la scène, puis cliquer sur *Rendre local*, et enfin reconnecter les signaux à la main au Mario.

7.5.2. Lancer une carapace de Koopa. On reprend la scène précédente, en ajoutant une action supplémentaire: lorsque Mario porte la carapace et que le joueur appuie sur la touche Z, Mario lance la carapace comme présenté en Figure 7.40.

Figure 7.40: Mario lance la carapace de Koopa.

On va faire en sorte que, lorsque la carapace sort de l'écran et ne revient jamais, elle sera supprimée. Pour savoir si un objet sort de l'écran, on utilisera un nœud de type *VisibilityNotifier2D*.

- Repartir du projet précédent, et ajouter un nœud fils de type *VisibilityNotifier2D* au nœud *Shell*.
- Dans les paramètres du projet, ajouter la prise en compte de l'action *Throw*, associée à la touche Z.
- Dans le script *Mario.gd*, ajouter une nouvelle variable booléenne *throwing_shell* initialisée à *false*, qui va tester si Mario lance la carapace ou non.
- Ajouter au nœud *Shell* un script *Shell.gd* pour gérer le déplacement de la carapace et sa suppression de la scène, puis connecter le signal *screen_exited* du nœud *VisibilityNotifier2D* au nœud Shell. Dans la fonction *_on_VisibilityNotifier2D_screen_exited* créée dans le script, appeler la fonction *queue_free*.
- Dans *Shell.gd*, ajouter la fonction *physics_process* comme présenté en Figure 7.41.
- Dans *Mario.gd*, ajouter la prise en compte de la touche clavier Z à la fin de la fonction *get_input* comme présenté en Figure 7.42.
- Ajouter la prise en compte de la variable *throwing_shell* à la fin de la fonction *hold_shell* comme présenté en Figure 7.43.

```
extends KinematicBody2D

var speed : Vector2
var velocity : Vector2

func _ready():
    speed=Vector2(200,200)
    velocity=Vector2(0,0)

func _on_VisibilityNotifier2D_screen_exited():
    queue_free()

func _physics_process(_delta):
    velocity=move_and_slide(velocity)
```

Figure 7.41: Script *Shell.gd*

```
    if Input.is_action_just_pressed("Throw"):
        throwing_shell=true
```

Figure 7.42: Nouvelle action *get_input*.

```
func hold_shell():
    if holding_shell==true:
        if looking_left==true:
            node_shell.position = node_mario.position + Vector2(-44,0);
        else:
            node_shell.position = node_mario.position + Vector2(42,0);
    if throwing_shell==true:
        holding_shell=false
        if looking_left==true:
            node_shell.velocity=Vector2(-1,0)
        else:
            node_shell.velocity=Vector2(1,0)
        node_shell.velocity=node_shell.speed * (node_shell.velocity).normalized()
        throwing_shell=false
```

Figure 7.43: Nouvelle fonction *hold_shell*.

Notez que dans la fonction *hold_shell*, lorsque la variable *throwing_shell* est initialisée à *true*, ce sont les variables *speed* et *velocity* du Script *Shell.gd* que nous devons modifier. On peut accéder à ces variables depuis notre nœud *node_shell* déclaré dans le script. Ainsi, pour accéder à la variable *velocity* de *Shell.gd*, il suffit d'appeler *node_shell.velocity*.

En exécutant la scène, on peut alors s'apercevoir qu'on peur lancer la carapace; elle s'arrête si elle entre en collision avec une plateforme qui a une forme de collision; et si elle sort de l'écran, elle est supprimée. La carapace lancée a une vitesse constante déterminée par la variable *speed* de *Shell.gd*. On peut aussi atténuer sa vitesse de déplacement en réalisant une interpolation linéaire de la variable *node_shell.velocity*.

7.6. COORDONNER LES MOUVEMENTS DE DEUX *Sprite*

Dans cette Section, on veut définir une animation d'un personnage de *Mario* placé sur un *Yoshi*. On veut de plus que le *Yoshi* puisse se déplacer à gauche ou à droite, et qu'il puisse gonfler et dégonfler ses joues en appuyant respectivement sur les touches A et Z. Voici l'image utilisée pour le *Sprite* de *Mario*.

Figure 7.44: *Sprite* de Mario sur le Yoshi.

Voici également les animations de course du Yoshi, sans les joues gonflées puis avec :

Figure 7.45: Course du Yoshi

Figure 7.46: Course du Yoshi avec les joues gonflées

Voici ensuite la procédure à suivre:

- Récupérer les scènes *Tilemap.tscn* et *Parallax.tscn* de la Section 6.4, ainsi que tous les assets nécessaires (le fond d'écran et la *tile-sheet*), et les copier dans le répertoire Courant du nouveau projet. On peut alors instancier ces scènes dans la scène principale pour les réutiliser.

- Créer un nœud principal de type *Node2D* avec un fils *KinematicBody2D* en fils, puis deux nœuds *AnimatedSprite*, un pour le Yoshi et un pour le Mario.

- Définir quatre animations pour le Yoshi: *Idle* avec la première image, *IdleLoaded* avec la quatrième image, *Run* avec dans l'ordre les images $2, 3, 2$ et 1 et *RunLoaded* avec les images $5, 6, 5$ et 4.

- Dans les paramètres du projet, ajouter la prise en compte des actions associées aux touches A et Z en les nommant respectivement *Load_dragon* et *Dragon*, comme en Figure 7.47.

Figure 7.47: Associer des nouvelles actions au projet.

- Dans un script, on voudra régler les paramètre *Offset* du Yoshi à $(2, 0)$ quand il va vers la gauche, et $(88, 0)$ quand il va vers la droite. Mario a un *offset* de $(0, -14)$ quand le Yoshi est comme dans les images 1 et 4, et un *offset* de $(0, -10)$ pour les autres images. Notez que ces valeurs peuvent changer en fonction du positionnement de vos *AnimatedSprite*, à vous de tester la meilleure configuration. Ces variations d'*offset* produisent deux effets quand le Yoshi avance: Marion monte et descend, et sa tête oscille vers l'avant. Pendant un cycle de *Run* du Yoshi, les positions de Mario sont bas, bas, bas, haut et les positions de la tête du Yoshi sont arrière, avant, arrière, avant.

- Attacher ensuite au nœud *Main* le GDScript composé des fonctions suivantes:

```gdscript
extends Node2D

var dragon_loaded : bool
var velocity : Vector2
var speed : Vector2

func _ready():
    dragon_loaded = false
    velocity = Vector2(0,0)
    speed = Vector2(200,200)

func set_yoshi_idle():
    if dragon_loaded:
        $MarioOnYoshi/Yoshi.animation="IdleLoaded"
    else:
        $MarioOnYoshi/Yoshi.animation="Idle"
```

```gdscript
func set_yoshi_animation():
    if dragon_loaded:
        $MarioOnYoshi/Yoshi.animation="RunLoaded"
    else:
        $MarioOnYoshi/Yoshi.animation="Run"

func set_mario_offset():
    if $MarioOnYoshi/Yoshi.animation == "Idle":
        $MarioOnYoshi/Mario.offset=Vector2(0,-14)
    if $MarioOnYoshi/Yoshi.animation == "Run":
        if $MarioOnYoshi/Yoshi.frame == 3:
            $MarioOnYoshi/Mario.offset=Vector2(0,-14)
        else:
            $MarioOnYoshi/Mario.offset=Vector2(0,-10)
```

```
33 func get_input():
34     velocity = Vector2(0,0)
35     if !Input.is_action_pressed("ui_left") and !Input.is_action_pressed("ui_right"):
36         set_yoshi_idle()
37     if Input.is_action_pressed("ui_left"):
38         velocity=Vector2(-1,0)
39         set_yoshi_animation()
40         $MarioOnYoshi/Mario.flip_h = false
41         $MarioOnYoshi/Yoshi.offset = Vector2(2,0)
42         $MarioOnYoshi/Yoshi.flip_h = false
43     if Input.is_action_pressed("ui_right"):
44         velocity=Vector2(1,0)
45         set_yoshi_animation()
46         $MarioOnYoshi/Mario.flip_h = true
47         $MarioOnYoshi/Yoshi.offset = Vector2(88,0)
48         $MarioOnYoshi/Yoshi.flip_h = true
49     velocity = speed*velocity.normalized()
50     if Input.is_action_pressed("Dragon"):
51         dragon_loaded=false
52     if Input.is_action_pressed("Load_dragon"):
53         dragon_loaded=true
54
55 func _physics_process(_delta):
56     get_input()
57     set_mario_offset()
58     velocity = $MarioOnYoshi.move_and_slide(velocity)
```

Figure 7.48: *GDScript* du déplacement du Mario sur le Yoshi.

Chapitre 8

Ennemis et combat

8.1. Tuer un Koopa

Dans cette section, on veut ajouter des fonctionnalités à la scène présentée en Section 7.5.2: avoir un ou des Koopa(s) qui se baladent sur des plateformes, et qu'on puisse les tuer de deux manières différentes:

(1) en leur lançant une carapace;

(2) en leur sautant sur la tête.

Voici la procédure à suivre:

- Créer un nouveau projet repartant de la scène précédente. Pour ce faire, on va récupérer les scènes *Mario.tscn*, *Tilemap.tscn*, *Parallax.tscn*, *Main.tscn* et tous les assets nécessaires, et reconnecter les signaux des Area2D *LeftCatch*, *RightCatch* et *Area2D* au Mario.
- Insérer la scène *Koopa.tscn*, le script *Koopa.gd* et les images d'animation du projet de la Section 7.2.4, puis ajouter les nœuds *Area2DL* et *Area2DR* et les connecter comme en Section 7.2.4, de sorte que le Koopa se déplace sur sa plateforme.
- Pour (1), on va ajouter un nœud de type *Area2D* nommé *Killzone* au nœud *Shell*, qui va délimiter la zone de collision qui lui permettra de tuer le Koopa, puis connecter le signal *body_entered* de cette zone au nœud *Koopa*. Ceci a pour effet de créer la fonction *_on_Killzone_body_entered* dans le script *Koopa.gd*, que vous pouvez modifier comme en Figure 8.1. La variable booléenne *koopa_dead* a été ajoutée au script et initialisée à *false*.

```
func _on_Killzone_body_entered(body):
    if body.get_name()=="Koopa":
        koopa_dead=true
```

Figure 8.1: Script de la mort d'un Koopa recevant une carapace.

- Pour (2), on va ajouter un nœud de type *Area2D* au dessus de la tête du *Koopa* comme en Figure 8.2. Connecter le signal *body_entered* de cette *Area2D* au nœud *Koopa*, puis instancier la fonction *_on_TopKillzone_body_entered* comme présenté en Figure 8.3. La variable booléenne *koopa_shell* est initialisée à *false*.

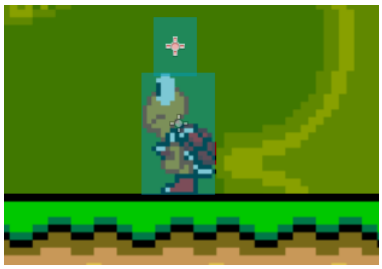


Figure 8.2: Position de la *TopKillzone*.

```
func _on_TopKillzone_body_entered(body):
    if body.get_name()=="Mario":
        koopa_shell=true
```

Figure 8.3: Script de la mort d'un Koopa quand Mario lui saute sur la tête.

- On peut alors ajouter une fonctionnalité supplémentaire: que le Koopa devienne une carapace lorsque Mario le tue en lui sautant dessus... Pour ce faire, on pourrait utiliser une fonction *replace_by* qui permet de remplacer un nœud par un autre, mais ce n'est pas optimal. Une autre méthode naïve est d'ajouter une animation supplémentaire, nommée *Shell*, au *AnimatedSprite* de notre Koopa, et que lorsque la variable *koopa_shell* passe à *true*, on joue l'animation *Shell*. Il faut alors mettre à jour la fonction *physics_process* comme en Figure 8.4. Toutefois, ce n'est pas totalement satisfaisant car la nouvelle carapace ne correspond pas au nœud *Shell* précédemment créé, et ne peut donc pas être ramassée puis lancée en l'état.

```
func _physics_process(delta):
    if turning>0:
        $AnimatedSprite.animation="Turn"
        turning -= 1
    else:
        $AnimatedSprite.animation="Walk"
    if velocity.x>0:
        $AnimatedSprite.flip_h=true
    else:
        $AnimatedSprite.flip_h=false
    velocity = speed * velocity.normalized()
    if not(turning>0):
        velocity = move_and_slide(velocity)
    if koopa_dead==true:
        queue_free()
    if koopa_shell==true:
        $AnimatedSprite.animation="Shell"
        velocity=Vector2(0,0)
        node_koopa.position = node_koopa.position + Vector2(0,8)
```

Figure 8.4: Fonction *physics_process* du script *Koopa.gd*.

Une meilleure méthode est en fait de travailler directement avec le nœud *Koopa*, de lui ajouter une animation *Shell* avec l'image de la carapace, et de reprendre ce qui a été fait dans les Sections 7.5.1 et 7.5.2 à partir du nœud *Koopa*. Toutefois, ceci est un peu plus fastidieux puisqu'il faut que s'assurer que *Koopa* est bien en animation *Shell* avant que *Mario* puisse ramasser la carapace, et donc gérer des variables booléennes supplémentaires. L'arborescence d'une scène avec ces fonctionnalités est présentée en Figure 8.5. Dans cette scène, Mario peut tuer deux Koopas qui deviennent des carapaces, puis récupérer leur carapace et la lancer pour tuer l'autre Koopa. Les scripts correspondants *Mario.gd* et *Shell.gd* sont disponibles sur cette page.

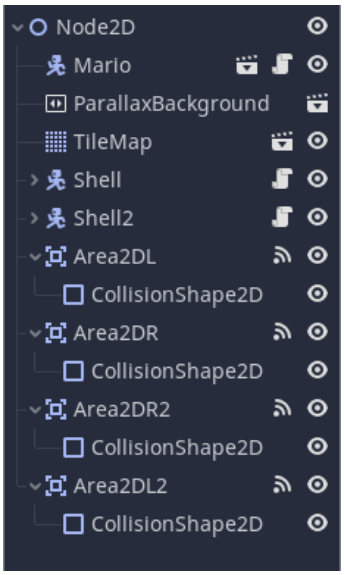


Figure 8.5: Arborescence du projet avec un *Koopa* qui devient une carapace ramassable.

8.1.1. Utilisation de Groupes. Dans notre script précédent, on vérifie plusieurs fois lors des détection de signaux *body_entered* associés aux *Area2D* que le *body* en question est bien soit Mario, soit Koopa avant de faire une action. Ceci peut devenir fastidieux lorsque l'on veut disposer de plusieurs personnages, ou de plusieurs ennemis...

Godot propose un moyen de regrouper plusieurs nœuds selon un Tag, via la structure de groupe. Un groupe va regrouper plusieurs nœuds, en général du même type, afin de simplifier la gestion automatique de ces nœuds, par exemple gérer les collisions ou les déplacements de plusieurs ennemis à la fois. La documentation de *Godot* sur la notion de Groupe est disponible sur cette page.

Un groupe est créé en ajoutant un nœud à un nouveau nom de groupe depuis l'onglet *Nœud* de l'inspecteur en haut à droite comme en Figure 8.6. Dans cet onglet, il faut insérer le nom du nœud de la scène pour lequel on veut créer un groupe, par exemple *Player* ou *Mario*, puis cliquer sur *Ajouter*. Cela aura pour effet de faire apparaître le nouveau groupe dans la liste, comme en Figure 8.7.

Figure 8.6: Onglet *Groupes* de l'inspecteur.

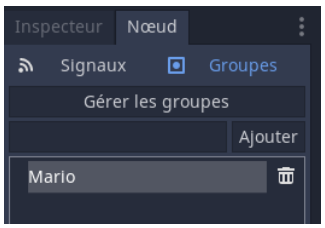


Figure 8.7: Ajout d'un nouveau groupe correspondant au nœud *Mario*.

En cliquant sur le bouton *Gérer les groupes*, on peut ajouter ou supprimer des nœuds aux groupes existants. La fenêtre de gestion de Groupe, présentée en Figure 8.8, est composée de 3 colonnes : la première est la liste des groupes utilisés dans la scène, la deuxième est la liste des nœuds qui ne font pas partie du groupe sélectionné à gauche, la troisième est la liste des nœuds du groupe.

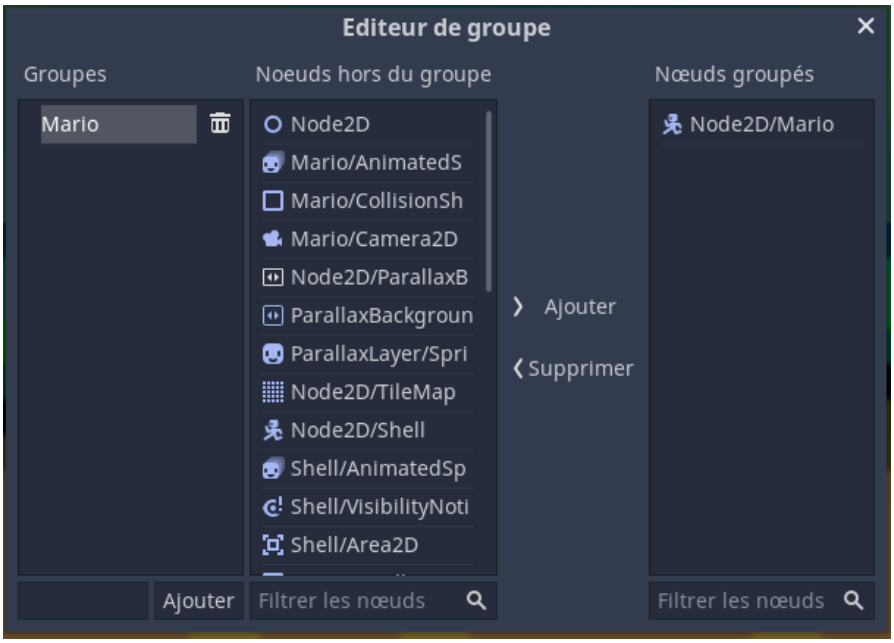


Figure 8.8: Fenêtre de gestion de Groupes.

On peut aussi gérer les groupes à l'intérieur d'un script :

- pour ajouter le nœud courant d'un script à un groupe *Group*, on peut utiliser l'instruction

```
add_to_group("Group")
```

- pour accéder aux nœuds à l'intérieur d'un nœud *Group*, on peut utiliser l'instruction

```
var nodes = get_tree().get_nodes_in_group("Group")
```

- pour remplacer notre commande `if body.get_name()=="Mario"` par une commande plus générale, on peut utiliser l'instruction

```
if body.is_in_group("Group").
```

  pour déclencher le signal d'entrée dans l'*Area2D*. Ainsi, tous les nœuds du groupe déclencheront le signal.

8.2. Barre de vie

Dans cette section, on va ajouter une barre de vie qui correspond à l'état de santé de notre personnage. Pour ce faire, on va repartir de la scène précédente (en ne gardant qu'un seul Koopa), et on va ajouter le fait que lorsque Mario entre dans la *Killzone* centrale du Koopa, il va perdre de la vie. Voici la procédure à suivre:

- Ajouter dans la scène principale un nœud de type *CanvasLayer*, puis lui ajouter un nœud enfant de type *TextureProgress*, renommé en *HealthBar*, qui designera notre barre de vie. Pour créer une barre de vie qui se met à jour avec un tel nœud, il nous suffira de

deux images (ou trois, selon ce que vous souhaitez en faire): l'image de la barre de vie vide, et l'image de la barre de vie pleine. On pourrait aussi imaginer charger une image correspondant aux différents états de la barre de vie, mais ce serait plus lourd et compliqué, d'autant que le nœud *TextureProgress* permet plus de possibilités.

- Dans l'onglet *Textures* du nœud *TextureProgress*, ajouter les images correspondant à la barre de vie vide dans les champs *Under* et *Over*, et la barre de vie pleine dans le champ *Progress*, comme présenté en Figure 8.9.

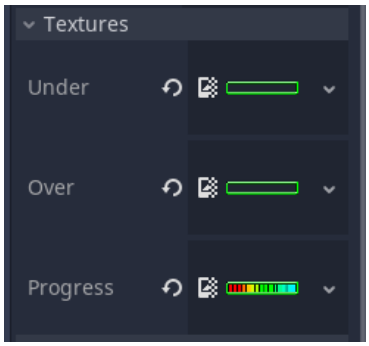


Figure 8.9: Réglage des textures de notre *Healthbar*.

Voici ci-dessous les textures utilisées pour notre barre de vie, trouvées sur cette page.

Figure 8.10: Barre de vie vide.

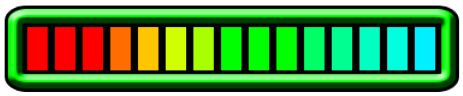


Figure 8.11: Barre de vie pleine.

- Dans le champ *Range* de l'inspecteur, présenté en Figure 8.12, les onglets *Min Value* et *Max Value* nous permettent de choisir les valeurs minimale (correspondant à l'état vide) et maximale (correspondant à l'état plein) pour la donnée que contiendra la barre, dans notre cas un nombre de points de vie. Puisque la barre choisie présenté 15 traits de vie, nous choisirons une valeur maximale de 150, et nous décrémenterons la vie de 30 par 30.

  L'onglet *Value* correspond à la valeur actuellement stockée; si vous modifiez cette valeur, vous pouvez constater l'évolution du remplissage de la barre de vie.

- Dans le script *Mario.gd*, ajouter une variable *HP* de type *int*, initialisée à 150 (ou la valeur maximale que vous avez choisi dans le *TextureProgress*. Ensuite, modifier la fonction *_on_Killzone_body_entered* du nœud *Shell.gd* comme en Figure 8.13, afin d'implémenter le fait que Mario perd des points de vie lorsqu'il entre dans la *Killzone* du *Koopa*.

  où la variable *node_mario*, correspondant au nœud *Mario*, a été récupérée grâce à l'instruction

```
onready var node_mario = get_node("/root/Node2D/Mario")
```

  Lors de l'éxécution, Mario aura alors tous ses points de vie, mais dès que vous entrez en collision avec Koopa sans le tuer par le dessus, la barre de vie diminuera. Cependant,

Figure 8.12: Réglages des paramètres de la *Healthbar* dans *Range*.

```
func _on_Killzone_body_entered(body):
    if body.get_name() == "Mario":
        node_mario.HP -= 30
```

Figure 8.13: Modification du script *Shell.gd*.

pour le moment il ne se passera rien une fois vos points de vie diminués à zéro... Une première possibilité est de recharger la scène courante depuis le départ lorsque c'est le cas, par exemple en ajoutant une fonction *death()* dans le script *Mario.gd* comme présenté en figure 8.14, et de l'appeler dans la fonction *physics_process*. Nous verrons par la suite dans le chapitre 9 comment ajouter un menu *Game Over*, qui permet de relancer la scène après un certain temps.

```
func death():
    if HP == 0:
        get_tree().reload_current_scene()
```

Figure 8.14: Rechargement de la scène lorsque les HP tombent à 0.

8.3. Système de tir ou d'attaque

Dans cette Section, nous allons voir comment ajouter un système d'attaque à notre personnage, par exemple en lançant un objet, en tirant avec une arme, ou dans notre cas en balançant une boule de feu à l'aide d'un *Yoshi*. Pour ce faire, on va repartir du projet avec le *Mario* sur le *Yoshi* présenté en 7.6. Importer les scènes *Main.tscn* (dans laquelle se trouve le nœud *MarioOnYoshi* si il n'a pas été instancié), *Parallax.tscn*, *TileMap.tscn* de ce projet ainsi que les assets nécessaires.

Dans les paramètres du projet, réattribuer les touches d'action *Load_dragon* et *Dragon* aux touches A et Z respectivement. On veut ajouter une action supplémentaire: lorsque le Yoshi relâche ses joues après pression sur Z, il lance une boule de feu comme présenté en Figure 8.15.

- Créer une nouvelle scène *Fireball.tscn* dans laquelle on ajoute un nœud principal de type *KinematicBody2D*, renommé en *Fireball*, puis 3 nœuds enfant: un *AnimatedSprite*, un

Figure 8.15: Yoshi lance une boule de feu en appuyant sur Z.

CollisionShape2D et un *VisibilityNotifier2D*, qui permettra de supprimer les boules de feu lorsqu'elles sortent de l'écran. Dans le *AnimatedSprite*, ajouter une animation *Idle* correspondant aux deux images présentées en Figure 8.16. L'arborescence de la scène est présentée en Figure 8.17.

Figure 8.16: Animation *Idle* de la boule de feu.

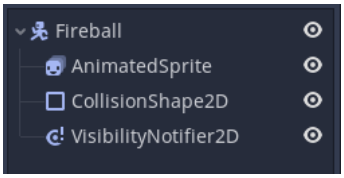


Figure 8.17: Arborescence de la scène *Fireball*.

- Dans le script *Main.gd*, ajouter une variable *dragon_velocity* de type *Vector2*. Cette variable va définir la direction des *Fireball*: si elle vaut $(1, 0)$, le Yoshi lancera une boule de feu vers la droite, si elle vaut $(-1, 0)$, il lancera une boule de feu vers la gauche. Puisque le Yoshi est de base orienté vers la gauche, elle sera initialisée à $(-1, 0)$.

- Attacher au nœud *Fireball* un script *Fireball.gd*, dans lequel on ajoute des variables *speed* et *velocity* de type *Vector2*, initialisées respectivement à $(300, 300)$ et $(0, 0)$. Ajouter ensuite une fonction *init()* comme présentée en Figure 8.18.

  Connecter ensuite le signal *screen_exited* du nœud *VisibilityNotifier2D* au nœud *Fireball*, puis compléter la fonction associée du script *Fireball.gd* comme présenté en Figure 8.19. Ajouter ensuite la fonction *physics_process* comme présentée en Figure 8.20.

- Dans le script *Main.gd*, modifier la fonction *get_input* pour ajouter le changement de la variable *dragon_velocity* lorsque les touches gauche ou droite sont pressées: lorsque c'est la touche gauche, *dragon_velocity* passera à $(-1, 0)$ et lorsque c'est la touche droite, elle passera à $(1, 0)$.

```
func init():
    if velocity == Vector2(0,0):
        velocity=node_main.dragon_velocity
    if velocity == Vector2(1,0):
        $AnimatedSprite.flip_h = true
        $AnimatedSprite.offset = Vector2(100,-40)
    if velocity == Vector2(-1,0):
        $AnimatedSprite.flip_h = false
        $AnimatedSprite.offset = Vector2(-100,-40)
    $AnimatedSprite.visible = true
    velocity = speed * velocity.normalized()
```

Figure 8.18: Fonction *init()* de *Fireball.gd*.

```
func _on_VisibilityNotifier2D_screen_exited():
    queue_free()
```

Figure 8.19: Supression de la *Fireball* hors de l'écran.

```
func _physics_process(delta):
    init()
    velocity=move_and_slide(velocity)
```

Figure 8.20: Fonction *physics_process*.

- Dans *Main.gd*, ajouter la fonction *dragon()* présentée en Figure 8.21.

```
func dragon():
    var fireball = preload("res://Fireball.tscn").instance()
    add_child(fireball)
    if $MarioOnYoshi/Mario.flip_h == true and $MarioOnYoshi/Yoshi.flip_h == true:
        fireball.position = $MarioOnYoshi.position + Vector2(60,0)
    else:
        fireball.position = $MarioOnYoshi.position
```

Figure 8.21: Fonction *dragon()* de *Main.gd*.

Notez que cette fonction présente une fonctionnalité important de *Godot*. L'appel à cette fonction créé une variable *fireball* qui est en fait une nouvelle instance (d'où l'utilisation de la fonction *instance()* de la scène chargée en paramètres, à savoir *"res://Fireball.tscn"*.

La fonction *add_child* ajoute cette nouvelle instance de la scène en enfant du nœud principal de la scène. Chaque appel à *dragon()* aura donc pour effet de créer une nouvelle copie de la scène *Fireball.tscn* dans notre jeu. Les lignes restantes de cette fonction servent à positionner les boules de feu créées en fonction de la position de notre Mario sur le Yoshi, avec la commande *fireball.position*.

Pour faire en sorte que ces boules de feu tuent des ennemis, il faut ensuite reprendre le principe de la Section 8.1.

8.4. Détection du personnage par un ennemi

Dans cette Section, on veut ajouter une fonctionnalité aux *Koopas* de la fonction 8.1 : qu'ils puissent nous détecter lorsque l'on s'approche d'eux et nous suivre (ou nous attaquer en nous retirant des HP). Pour ce faire, on va repartir du projet de la Section 8.2 avec un seul Koopa et une barre de vie.

- Créer un nouveau projet incorporant les scènes *Mario.tscn*, *Tilemap.tscn*, *Parallax.tscn* et *Main.tscn* ainsi que les différents scripts et assets. Ajouter les actions *Catch* et *Throw* associées aux touches A et Z respectivement.

- Ajouter une nouvelle *Area2D* (en nœud fils du nœud principal) autour du *Koopa*, nommé *DetectZone*, qui correspondra à la zone à l'intérieur de laquelle le Koopa pourra vous détecter, comme en Figure 8.22. Faire en sorte que cette zone soit suffisamment haute pour que le Koopa puisse encore vous détecter lorsque vous sautez. Cette étape est optionnelle si vous souhaitez que l'ennemi puisse vous suivre sur toute la carte. Connecter ensuite les signaux *body_entered* et *body_exited* de cette *Area2D* au nœud *Shell*.

Figure 8.22: Zone de détection autour du Koopa.

- Dans le script *Shell.gd*, ajouter une variable *following* de type *int*, initialisée à false, puis modifier les fonctions *_on_DetectZone_body_entered* et *_on_DetectZone_body_exited* comme en Figure 8.23.

```
func _on_DetectZone_body_entered(body):
    if body.get_name()=="Mario":
        following = true

func _on_DetectZone_body_exited(body):
    if body.get_name()=="Mario":
        following = false
```

Figure 8.23: Signaux associés à la *DetectZone*.

- Modifier ensuite la fonction *physics_process* du script *Shell.gd* comme présenté en Figure 8.24. Les lignes importantes correspondant à ce script sont les lignes 88 à 97. C'est la variable nouvellement créée *follow* qui va donner la direction à suivre à l'ennemi lorsque *following* vaut true: on va calculer un vecteur qui détermine la différence de position entre le joueur et l'ennemi, et on va se déplacer selon ce vecteur avec la vélocité *velocity_run*.

  Dans ce script, la variable *velocity_run* est initialisée à $(200, 100)$ mais le déplacement initial du Koopa sur sa plateforme remet sa vélocité en y à 0. Ainsi, le Koopa sera en mesure de nous suivre sur sa plateforme, avec la bonne orientation (lignes 95 et 97), mais toutefois, il restera sur le sol. Si on veut que le *Koopa* puisse suivre nos déplacements en l'air, il faut influer sur sa vélocité en y. Une telle modification est présentée en Figure 8.25.

  La ligne 96 est celle qui influe sur la vélocité verticale de notre Koopa, dépendant de la vélocité du Mario. Avec une telle affectation, le Koopa pourra suivre les sauts de notre personnage, en sautant toutefois un peu moins haut. Cependant, il peut ainsi y avoir une

```gdscript
func _physics_process(delta):
    if koopa_is_shell==true:
        $AnimatedSprite.animation="Shell"
        velocity_throw=move_and_slide(velocity_throw)
    else:
        if following == true:
            turning = 0
            $AnimatedSprite.animation="Walk"
            var follow = (node_mario.position - position).normalized()
            var vel = follow * velocity_run
            move_and_slide(vel,Vector2(-1,0))
            if follow.x > 0:
                $AnimatedSprite.flip_h = true
            else:
                $AnimatedSprite.flip_h = false
        else:
            if turning>0:
                $AnimatedSprite.animation="Turn"
                turning -= 1
            else:
                $AnimatedSprite.animation="Walk"
                if velocity_run.x>0:
                    $AnimatedSprite.flip_h=true
                else:
                    $AnimatedSprite.flip_h=false
            velocity_run = speed_run * velocity_run.normalized()
            if not(turning>0):
                velocity_run = move_and_slide(velocity_run)
```

Figure 8.24: Fonction *physics_process* de suivi du Mario.

```gdscript
func _physics_process(delta):
    print(velocity_run)
    var test = is_on_floor()
    print(test)
    if koopa_is_shell==true:
        $AnimatedSprite.animation="Shell"
        velocity_throw=move_and_slide(velocity_throw)
    else:
        if following == true:
            turning = 0
            $AnimatedSprite.animation="Walk"
            var follow = (node_mario.position - position).normalized()
            var vel = follow * velocity_run
            vel.y = node_mario.velocity.y + 300
            move_and_slide(vel,Vector2(-1,0))
            if follow.x > 0:
                $AnimatedSprite.flip_h = true
            else:
                $AnimatedSprite.flip_h = false
```

```gdscript
    else:
        if turning>0:
            $AnimatedSprite.animation="Turn"
            turning -= 1
        else:
            $AnimatedSprite.animation="Walk"
            if velocity_run.x>0:
                $AnimatedSprite.flip_h=true
            else:
                $AnimatedSprite.flip_h=false
        velocity_run = speed_run * velocity_run.normalized()
        if not(turning>0):
            velocity_run = move_and_slide(velocity_run)
```

Figure 8.25: Modification de la fonction *physics_process* pour que Koopa suive les sauts.

collision dans les airs, vous faisant perdre des points de vie ou tuer le Koopa. En l'état, le défaut subsistant est que si vous sortez de la *DetectZone* alors que Koopa est en l'air, il ne retombera jamais puisqu'on influe jamais sur sa vélocité en y lorsque *following* est à *false*. Une manière de résoudre ceci serait d'équipper le *Koopa* d'une gravité ambiante dès le début, et de l'ajouter constamment lorsqu'il n'est pas sur le sol, comme pour notre Mario.

8.5. Projectiles aléatoires et rebonds

On va créer une scène dans laquelle un Mario va se déplacer verticalement, en grimpant et en descendant sur une grille dans toutes les directions. La grille sera délimitée par des bordures rouges. Toutes les deux secondes, une boule de lave apparaît en haut à droite de l'écran, et sera lancée avec un angle initiale de déplacement vers la gauche; elles entreront en collision avec les murs pour rebondir dessus. On fera aussi en sorte que, lorsque la boule de lave touche le Mario, il clignote pendant quelques secondes. Le décor est présenté en Figure 8.26.

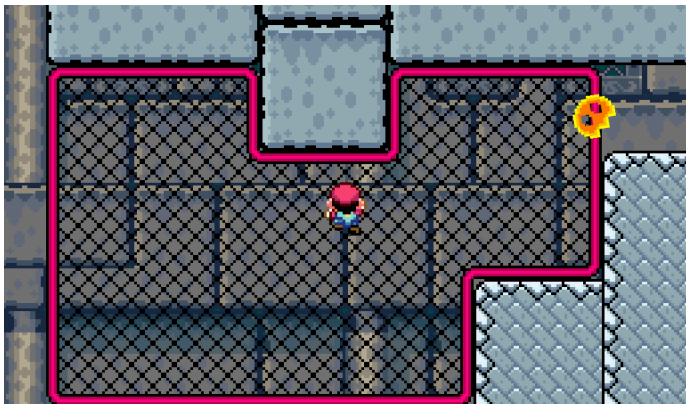


Figure 8.26: Mario sur une grille, avec apparition d'une boule de lave.

8.5.1. Arbre de scène et scripts. L'arborescence de la scène est présentée en Figure 8.27. La scène principale est composée d'un nœud principal de type *Node2D* équipé d'une scène instanciée *Mario.tscn*, une scène instanciée *Grid.tscn* correspondant à la *Tilemap* associée à la grille, une scène instanciée *Parallax.tscn* correspondant au décor de fond, présenté en Figure 8.28, et un nœud enfant de type *Timer* qui permettra de gérer le temps d'apparition des bulles de lave. Les propriétés de *Mirroring* du *ParallaxLayer* seront réglées à 2048 en x et 1168 en y.

Figure 8.27: Arborescence de la scène.

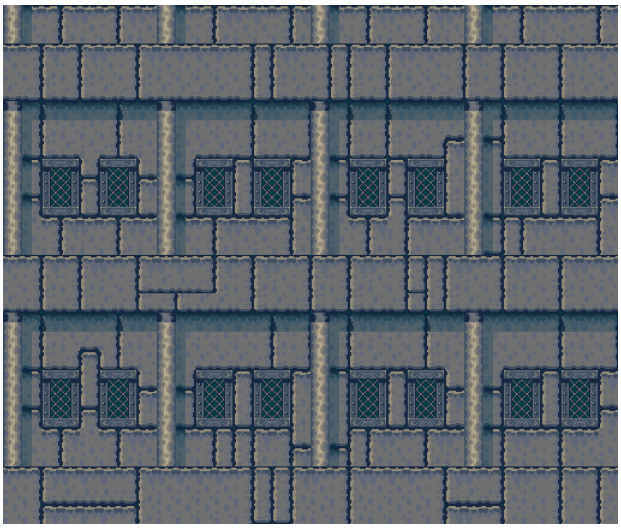


Figure 8.28: Image de décor du *ParallaxBackground*.

La *TileSheet* utilisée pour construire nos murs et nos grilles est présentée en Figure 8.29. Les tuiles utilisées respectivement pour la scène *Grid.tscn* et *Walls.tscn* sont présentées en Figure 8.30. Elles permettent de construire les maps de la grille et des murs présentées respectivement en Figures 8.31 et 8.32.

Figure 8.29: *TileSheet* utilisée pour les murs et les grilles.

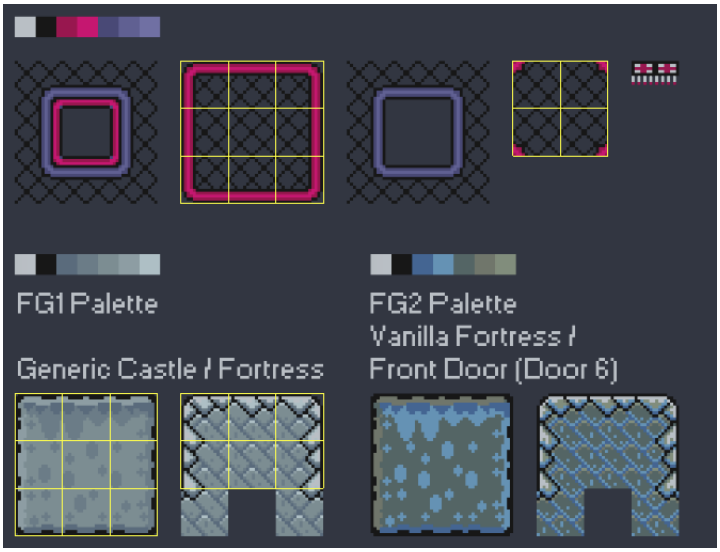


Figure 8.30: Tuiles utilisées dans nos *Tilemaps*.

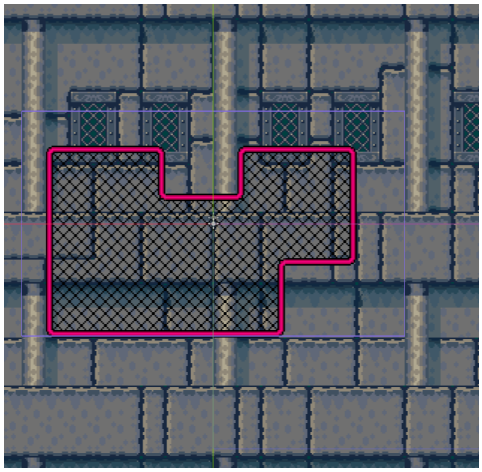


Figure 8.31: *Tilemap* de *Grid.tscn*.

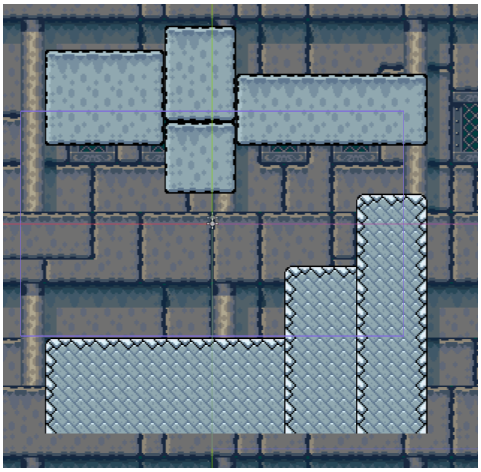


Figure 8.32: *Tilemap* de *Wall.tscn*.

L'arborescence de la scène *Mario* est présentée en Figure 8.33. Le nœud *AnimatedSprite* est équipée d'un *SpriteFrames* contenant une animation *Climb*, constituée des deux images présentées en Figure 8.34 (disponibles sur Moodle). La forme de collision de la *MarioArea2D* est calquée sur la *CollisionShape* du Mario. On règlera le champ *Vitesse* de l'animation à 10 images par seconde pour augmenter la vitesse d'enchaînement des images.

On va ensuite créer une nouvelle scène, nommée *LavaBubble.tscn*, pour ajouter les boules de lave. L'arborescence de la scène est présentée en Figure 8.35, le nœud principal de cette scène étant de type *KinematicBody2D*. Le *AnimatedSprite* est équipé d'un *SpriteFrame* contenant une

Figure 8.33: Arborescence de *Mario.tscn*.

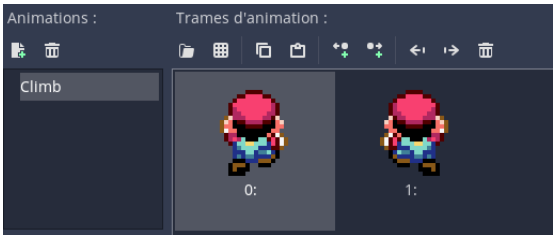


Figure 8.34: Animation *Climb* de Mario.

animation *Idle* constituée des deux images présentées en Figure 8.36. La forme de collision de la *ZigZagArea2D* est calquée sur la zone de collision de la boule de lave, qui est de forme circulaire pour matcher la forme de la boule.

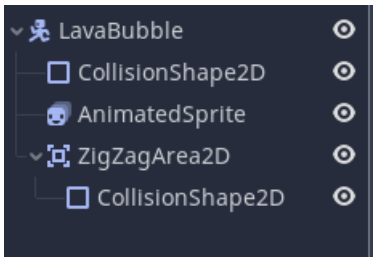


Figure 8.35: Arborescence de *LavaBall.tscn*.

Figure 8.36: Animation *Idle* de la *LavaBall*.

Après avoir aligné les formes de collision, désélectionner le champ *Visible* du nœud *LavaBall* pour le rendre invisible, de sorte que la boule de lave soit invisible tant qu'elle n'est pas à sa position initiale. Puis, suivre la procédure suivante pour les scripts de notre jeu:

- Créer un script *LavaBall.gd*, dans lequel on initialise des variables *init_done*, de type booléen initialisée à *false*; *speed* de type *Vector2* initialisée à $(1, 1)$ et *velocity* de type *Vector2* initialisée à $(0, 0)$.
- Définir les fonctions *init()*, *animate()*, *finish()* et *physics_process* présentées respectivement en Figures 8.37, 8.38, 8.39, 8.40.

```
func init():
    if not(init_done):
        position = Vector2(400,-100)
        visible = true
        velocity = Vector2(1,1).rotated(rand_range(0.78,3.92))
    init_done = true
    velocity = speed * velocity.normalized()
```

Figure 8.37: Fonction *init()*.

```
func animate():
    var rot = (velocity.angle() * 180 / PI) + 135
    rotation_degrees = rot
```

Figure 8.38: Fonction *animate*.

Dans la fonction *init*, la position $(400, -100)$ place les boules de lave en haut à droite de la grille. Pour la rotation du vecteur vélocité, on remarque qu'avec une initialisation de *velocity* à $(1, 1)$, le vecteur a un angle de 45°, et donc en appliquant une rotation entre 0.78 et 3.92 radians, on obtient un angle initial entre 90 et 270°. Dans la fonction *animate*, la fonction *Vector2.angle()* retourne un angle en radians, on obtient ainsi la

```
func finish():
    if position.distance_to(Vector2(0,0)) > 1000:
        queue_free()
```

Figure 8.39: Fonction *finish()*.

```
func _physics_process(delta):
    init()
    if move_and_collide(velocity) != null:
        velocity = velocity.bounce(move_and_collide(velocity).norma
    animate()
    finish()
```

Figure 8.40: Fonction *physics_process*.

valeur équivalente en degrés en multipliant par 180 et en divisant par π. L'ajout du 135° permet d'obtenir une boule de lave orientée vers la droite. Dans la fonction *finish*, on impose que si la *LavaBubble* est trop loin de la coordonnée $(0, 0)$, elle est supprimée de la scène.

Dans la fonction *physics_process*, la fonction *move_and_collide* renvoie l'objet en collision lorsqu'il y a collision, et on applique à *velocity* une nouvelle valeur correspondant au rebond (avec la fonction *Vector2.bounce*) selon la normale du point de collision. En l'absence de collision, la fonction *mode_and_collide* renvoie `null`.

- Créer un script *Mario.gd*, y ajouter des variables *hit_delay* de type *int* initialisée à 0, *speed* de type *Vector2* initialisée à $(200, 200)$ et *velocity* de type *Vector2* initialisée à $(0, 0)$.
- Connecter le signal *_on_MarioArea2D_area_shape_entered* au nœud Mario, puis modifier la fonction nouvellement créer dans le script en affectant à *hit_delay* la valeur 80. Ajouter les fonctions *hit_effect*, *get_input* et *physics_process* comme présentées en Figure 8.41, 8.42 et 8.43.

```
func hit_effect():
    if hit_delay > 0:
        hit_delay -= 1
        if hit_delay % 4 == 0:
            if $AnimatedSprite.visible == true:
                $AnimatedSprite.visible = false
            else:
                $AnimatedSprite.visible = true
    if hit_delay == 0:
        $AnimatedSprite.visible = true
```

Figure 8.41: Fonction *hit_effect()*.

```
func get_input():
    velocity = Vector2(0,0)
    if not(Input.is_action_pressed("ui_down")) and \
       not(Input.is_action_pressed("ui_left")) and \
       not(Input.is_action_pressed("ui_right")) and \
       not(Input.is_action_pressed("ui_up")):
        $AnimatedSprite.playing = false
    else:
        $AnimatedSprite.playing = true
    if Input.is_action_pressed("ui_right"):
        velocity += Vector2(1,0)
    if Input.is_action_pressed("ui_left"):
        velocity += Vector2(-1,0)
    if Input.is_action_pressed("ui_down"):
        velocity += Vector2(0,1)
    if Input.is_action_pressed("ui_up"):
        velocity += Vector2(0,-1)
    velocity = speed * velocity.normalized()
    print(velocity)
```

Figure 8.42: Fonction *get_input*.

```
func _physics_process(delta):
    get_input()
    velocity = move_and_slide(velocity)
    hit_effect()
```

Figure 8.43: Fonction *physics_process* de *Mario.gd*.

La fonction *get_input* définit l'action du Mario selon les touches du clavier pressées, et modifie le vecteur *velocity* en fonction. Si aucune des touches directionnelles n'est pressée, on désactive l'animation *Climb*, et on l'active si une de ces touches est pressée. La fonction *hit_effect* a pour but d'ajouter l'effet de clignotement en fonction des collisions : quand *hit_delay* est strictement supérieure à 0, Mario clignote pour indiqué qu'il a été touché par un objet. L'effet de clignotement est généré par alternance du champ *visible* du Mario à `true` et `false`. Quand *hit_delay* vaut 0, Mario est visible et ne clignote pas.

- Dans le nœud *Timer*, cocher l'option *Autostart* de l'inspecteur, puis régler le champ *Wait Time* à 2. Créer un script *Main.gd* attaché au nœud principal, et connecter le signal *_on_Timer_timeout* du *Timer* au nœud *Main*. Compléter la fonction ainsi créée dans le script comme présenté en Figure 8.44.

```
func _on_Timer_timeout():
    var lavabub = preload("res://LavaBubble.tscn").instance()
    add_child(lavabub)
```

Figure 8.44: Fonction *_on_Timer_timeout* du script *Main.gd*.

Cette fonction instancie une nouvelle scène *LavaBubble.tscn* toutes les 2 secondes, lorsque le signal de *timeout* du *Timer* est activé, et l'ajoute en enfant du nœud principal de la scène comme en Section 8.3.

8.5.2. Collisions avec *layer* et *masks*. Pour gérer les collisions de manière optimale, nous allons utiliser des *layer* et *masks* de collision. Ces propriétés sont accessibles depuis l'onglet *PhysicsBody2D* de l'inspecteur d'un nœud pouvant être soumis à des propriétés physiques, voir Figure 8.45.

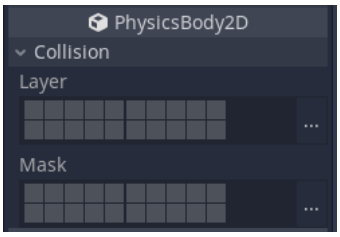


Figure 8.45: Onglet *PhysicsBody2D* de l'inspecteur.

Ces propriétés permettent de régler manuellement, pour un objet donné, quels objets peuvent interagir (c'est-à-dire entrer en collision) avec lui et quels objets ne peuvent pas. Pour un projet donné, on va associer à chaque nœud (ou groupe de nœuds) disposant d'une physique un *Layer* associé, depuis l'onglet *Layer Names, 2d Physics* des paramètres du projet (comme en Section 9.3, voir Figure 9.11). Ainsi, dans notre cas, le *Layer1* sera associé au Mario, le *Layer2* sera associé à la grille, le *Layer3* sera associé aux murs et le *Layer4* sera associé à la boule de lave. Ainsi, dans l'onglet *PhysicsBody2D* du nœud Mario, on cochera la case associée au *Layer1*, et on peut vérifier en passant la souris sur cette case qu'elle correspond bien au *Layer* Mario, comme en Figure 8.46.

Pour un objet donné avec le bon *Layer*, on va ensuite régler le *Mask* de collision associé. Notez que les cases de l'onglet *Mask* sont équipées des mêmes noms que celles des *Layer*. Ainsi, si on veut que Mario interagisse avec la grille, on cochera la case associée à *Grid* dans

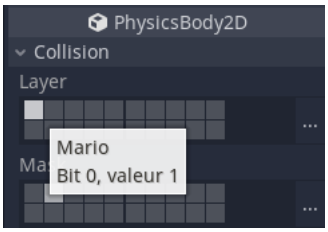


Figure 8.46: Réglage du *Layer* de Mario.

l'onglet *Mask* du Mario. Si on ne veut pas qu'il interagisse avec les murs, on gardera la case associée à *Wall* de l'onglet *Mask* du Mario décochée.

Dans notre cas, on va donc régler les quatre *Layer* et *Mask* des nœuds respectifs *Mario*, *Grid*, *Wall* et *LavaBubble* comme présenté en Figures 8.47, 8.48, 8.49 et 8.50 respectivement.

Figure 8.47: Mario.

Figure 8.48: Grid.

Figure 8.49: Wall.

Figure 8.50: LavaBubble.

Ainsi, Mario (1) entrera uniquement en collision avec les grilles (2); les grilles (2) entreront en collision avec elles-même (2); les murs (3) entreront en collision avec eux-même (3) et les boules de lave (4) entreront en collision avec les murs (3).

Les formes de collision sur les tuiles associées aux murs et aux grilles sont présentées en Figure 8.51, 8.52, 8.53 et 8.54.

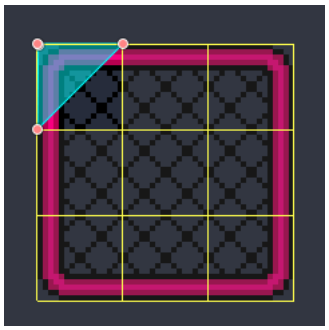


Figure 8.51: Coin de grille.

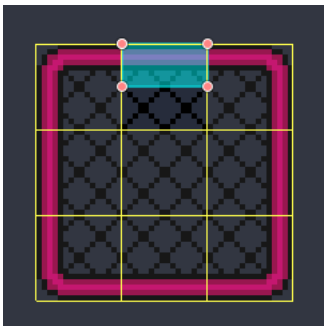


Figure 8.52: Bord de grille.

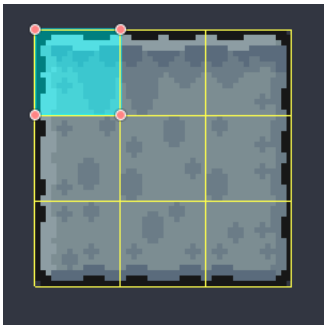


Figure 8.53: Coin de mur.

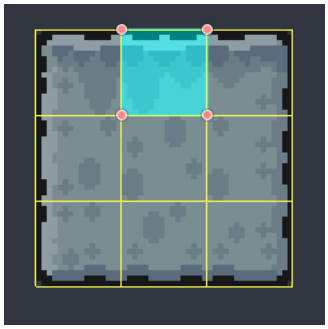


Figure 8.54: Bord de mur.

Chapitre 9

Changement de scène

Dans ce Chapitre, nous allons découvrir comment changer de scène principale en cours d'exécution d'un jeu. Ceci permet par exemple de développer un jeu avec plusieurs niveaux, avec changement de niveau automatique lorsqu'un personnage atteint un objectif.

9.1. Bouton de changement de scène

Dans ce premier exemple, on va créer deux scènes très simples: une scène verte nommée *GreenScene.tscn*, et une scène jaune, et on voudra passer de l'une à l'autre en cliquant sur un bouton. On va ainsi créer une scène verte comprenant un nœud principal de type *CanvasLayer*, auquel on va attacher 3 nœuds fils: un nœud *ColorRect*, un nœud *Label* et un nœud *Button* que l'on renomme en *ChangeSceneButton*. Dessiner ensuite un rectangle de couleur verte dans tout le cadre de la scène à l'aide des propriétés du *ColorRect*, comme en Figure 9.2.

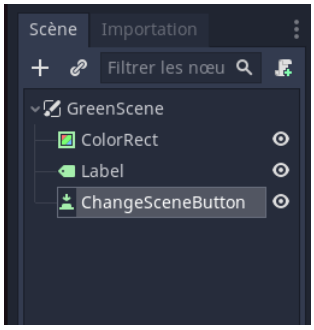


Figure 9.1: Arborescence de la scène verte.

Figure 9.2: Scène verte.

Créer ensuite une autre scène *RedScene.tscn* avec la même arborescence, mais en réglant la couleur du *ColorRect* à rouge. Attacher ensuite un GDScript au nœud principal de la scène *GreenScene.tscn* comme en Figure 9.3.

La fonction *_on_ChangeSceneButton_pressed* permet d'attacher un signal au bouton de la scène *GreenScene.tscn* (comme nous l'avions déjà fait précédemment), et la fonction *get_tree().change_scene* est la fonction qui nous permet d'aller vers une autre scène de notre projet: ici on lui demande de changer vers la scène rouge. Vous pouvez alors exécuter le jeu, et voir qu'un clic sur le bouton

```
1  extends CanvasLayer
2
3  func _on_ChangeSceneButton_pressed():
4      get_tree().change_scene("res://RedScene.tscn")
5
```

Figure 9.3: Changement de scène simple.

permet de passer de la scène verte à la rouge. En revanche, si vous créez le même script pour la scène rouge en voulant passer vers la verte, vous pouvez remarquer qu'on ne peut pas passer de vert à rouge, puis revenir vers vert. L'appel à une telle fonction a pour effet de réinitialiser la scène que l'on quitte, qui n'est alors plus accessible... C'est suffisant si on veut évoluer de niveau en niveau, mais pas totalement si on veut pouvoir revenir dans des endroits de notre jeu précédemment explorés.

9.2. GÉRER UN *SceneSwitcher*

La méthode la plus efficace pour gérer un jeu comprenant plusieurs niveaux ou scènes, est de gérer manuellement un nœud qui fait office de *Scene switcher* (changeur de scène). Dans une autre scène, créer un nœud de type *Node* simple, et le renommer en *SceneSwitcher*, auquel on ajoute en nœud fils la scène correspondant au niveau principal, ici par exemple *GreenScene.tscn*.

On va modifier le script *GreenScene.gd* précédent par le script suivant, présenté en Figure 9.4. La ligne 3 permet d'initialiser un signal, qui sera émis lorsque le bouton de la scène ambiante sera pressé. Par ailleurs, la ligne 4 est utile pour gérer plusieurs niveau, elle permet d'ajouter au nœud *CanvasLayer* une variable exportable de type *String*, qui correspondra au nom du niveau représenté par la scène, et à laquelle nous pourrons accéder. On peut alors modifier pour chaque scène du projet le nom du niveau, comme en figure 9.5. Nommez alors le niveau correspondant à la scène *GreenScene.tscn* (resp. *RedScene.tscn*) en *GreenScene* (resp. *RedScene*).

```
1  extends CanvasLayer
2
3  signal level_changed
4  export (String) var level_name = "level"
5
6  func _on_ChangeSceneButton_pressed():
7      emit_signal("level_changed",level_name)
```

Figure 9.4: GDScript pour émettre le signal d'un niveau.

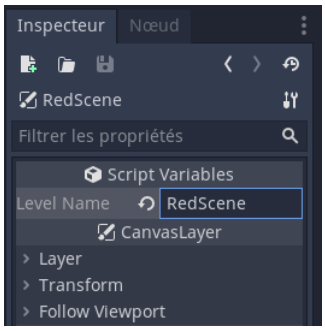


Figure 9.5: Attribution du nom de niveau.

Une fois ceci fait, on peut attacher un GDScript au *SceneSwitcher* comme en Figure 9.6, permettant d'obtenir une pile d'appels vers différents niveaux.

```gdscript
extends Node

onready var current_level = $GreenScene

func _ready():
    current_level.connect("level_changed",self,"handle_level_changed")

func handle_level_changed(current_level_name: String):
    var next_level
    var next_level_name : String
    match current_level_name:
        "GreenScene":
            next_level_name = "RedScene"
        "RedScene":
            next_level_name = "GreenScene"
        _:
            return
    next_level=load("res://"+next_level_name+".tscn").instance()
    add_child(next_level)
    next_level.connect("level_changed",self,"handle_level_changed")
    current_level.queue_free()
    current_level = next_level
```

Figure 9.6: GDScript du *SceneSwitcher*

On crée une variable *current_level* qui stocke le nom du niveau actuel. La fonction *handle_level_changed* permet de modifier la valeur du niveau suivant (grâce aux variables *next_level_name* et *next_level*) en fonction du niveau actuel. La fonction *match* correspond à une disjonction de cas sur la valeur de la variable *current_level_name*; le dernier cas correspond à un match qui ne contient pas un nom de niveau, et dans ce cas on ne renvoie rien. On ajoute ensuite le *next_level* dans l'arborescence des niveaux à visiter, puis on remplace le niveau actuel par ce dernier.

9.3. CHANGEMENT DE NIVEAU DEPUIS UNE ZONE

Nous allons apprendre à gérer un changement de niveau lorsqu'un *Sprite* rentre dans une zone, par exemple lorsque Mario passe une porte. Pour ce faire, créer un nouveau projet et dans le répertoire de ce nouveau projet, importer les scènes *Mario.tscn*, le script *Mario.gd* et les images d'animation de Mario correspondant à la Section 6.2.4. Renommez le nœud principal en *Level1*, puis choisissez d'enregistrer le nœud Mario comme scène en cliquant sur le bouton en Figure 9.8.

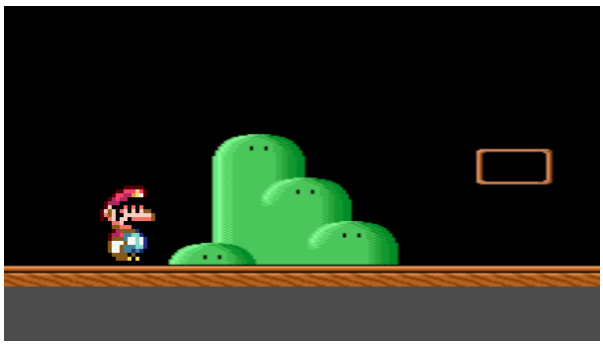


Figure 9.7: Mario dans le décor précédent.

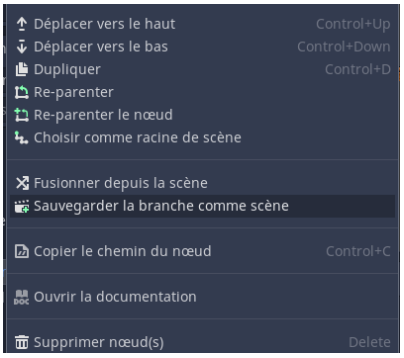


Figure 9.8: Enregistrement de Mario en *Mario.tscn*

Créer ensuite une autre scène *Level2.tscn* dont le nœud principal est un *Node2D*, puis instanciez lui la scène *Mario.tscn* en nœud fils, ainsi qu'un nœud Sprite pour ajouter le nouveau décor présenté en Figure 9.9.

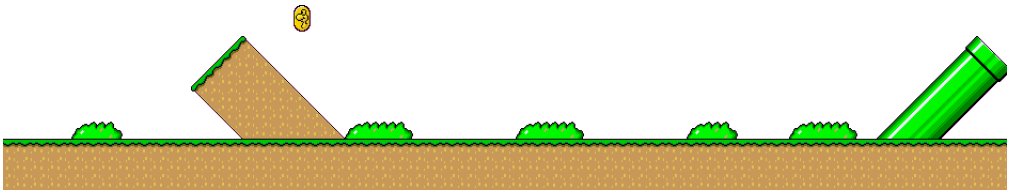


Figure 9.9: Décor du *Level2*

L'idée est ensuite de créer une zone à la droite du décor correspondant au *Level1*, et de faire en sorte qu'en rentrant dans cette zone, on change vers la scène *Level2*.

Ajouter un nœud de type *Area2D* en tant que fils du nœud principal, puis ajoutez lui une *CollisionShape2D* de forme rectangulaire, et placez là où vous souhaitez obtenir le changement de niveau, par exemple comme en Figure 9.10.

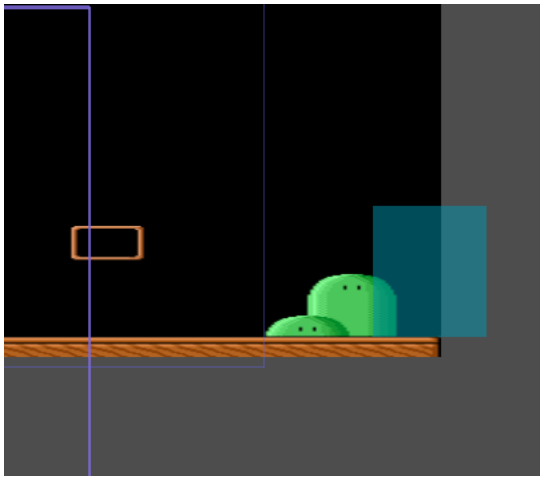


Figure 9.10: Zone de changement de niveau.

Il est alors préférable de sauvegarder le nœud *Zone* comme une autre scène, si on souhaite réutiliser ce genre de zone de changement de niveau par ailleurs. Notez qu'il connecter notre nœud à un signal, depuis l'onglet Nœud en haut à droite (au niveau de l'inspecteur), doublez cliquez sur le signal *body_entered* et connectez le. S'ouvre alors la fenêtre de GDScript avec une première fonction instanciée. Complétez alors ce Script comme en Figure 9.13. Ajouter dans les paramètres du projet, onglet *Général*, section *Layer Names*, choix *2d Physics* deux layers; un pour le *Mario*, un pour la *Zone* comme en Figure 9.11. Ensuite, dans l'onglet *Collision* de l'inspecteur correspondant à ces deux nœuds, assigner le bon *layer* dans l'onglet *Layer*, et

assigner le *Mask* opposé (Mario pour la Zone, Zone pour le Mario). Notez alors que dans la scène *Zone.tscn*, on peut modifier dans l'inspecteur la variable *Next Scene Path*, et même lui indiquer un chemin dans l'arborescence de notre projet ! Choisissez de lui assigner la scène *Level2.tscn* comme en Figure 9.12.

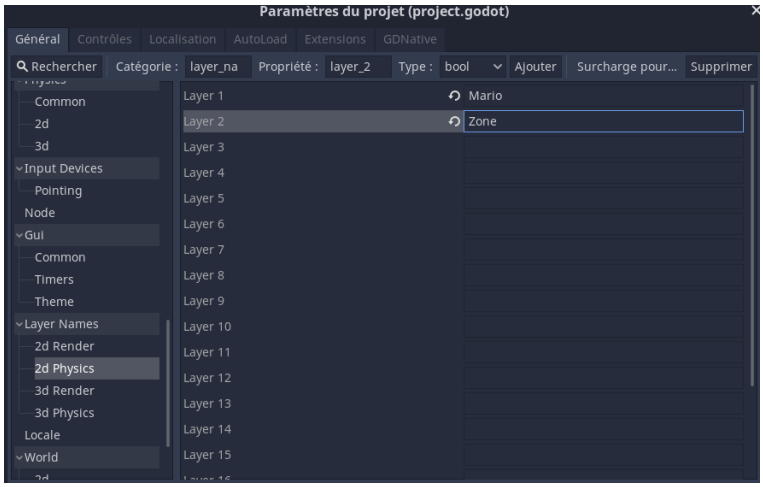


Figure 9.11: Ajout des *Layer* de collision.

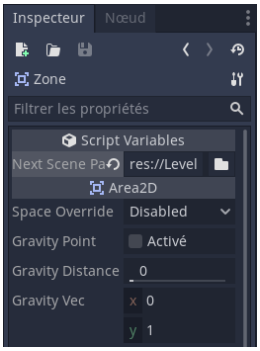


Figure 9.12: Choix du chemin vers la scène suivante.

```
1  tool
2  extends Area2D
3
4  export (String,FILE) var next_scene_path = ""
5
6  func _get_configuration_warning():
7      if next_scene_path == "":
8          return "next_scene_path must be set"
9      else:
10         return ""
11
12
13 func _on_Zone_body_entered(body):
14     if get_tree().change_scene(next_scene_path) != OK:
15         print("Error: Unavailable scene!")
```

Figure 9.13: GDScript pour une zone de changement de niveau.

Notez qu'on retrouve la fonction *get_tree* dans la fonction *_on_Zone_body_entered*, qui permet de changer de niveau vers la valeur de *next_scene_path*. Le mot clé *tool* et la fonction *get_configuration_warning* permet d'afficher un message d'erreur à l'écran lorsque le nom de la scène suivant n'est pas instancié correctement.

Vous pouvez alors exécuter le jeu, et voir que lorsque vous déplacez le Mario vers la droite dans le *Level1*, on est immédiatement envoyé dans le *Level2*. Notez que vous pouvez également ajouter une Zone à gauche du *Level2* permettant de faire le chemin inverse, en modifiant la variable *Next Scene Path* de cette nouvelle zone vers le *Level1.tscn*. Pour le moment, l'endroit où atterit le Mario dans le *Level2* dépend uniquement de sa position dans la scène *Level2.tscn*, puisqu'on réalise juste un simple changement de scène. On pourrait automatiser ce processus en créant un script attaché au nœud principal qui contient une variable *player_initial_position* et une variable *player_spawn_location*, et changer la valeur de ces variables dans la fonction *_on_Zone_body_entered* et dans le Script de déplacement du personnage.

9.4. MENUS DE JEU

Dans cette Section, on va apprendre à créer des menus de jeu, notamment un menu de démarrage, offrant le choix à l'utilisateur entre lancer le jeu et quitter; et un menu *Game Over* lorsque le joueur meurt. On va repartir du projet de la Section 8.4 avec Mario équipé d'une barre de vie, et Koopa qui détecte et attaque Mario lorsqu'il entre dans une zone autour de lui. On va renommer la scène *Main.tscn* en *Level1.tscn*.

9.4.1. Menu de démarrage. Créer une nouvelle scène nommée *StartMenu.tscn* dont le nœud principal est un nœud de type *Control* renommé en *StartMenu*. Ajouter un nœud enfant de type *VBoxContainer*, qui dictera la position de nos boutons sur l'écran à partir du bouton *Disposition sur l'écran* en haut au centre de la fenêtre, voir Figure 9.14.

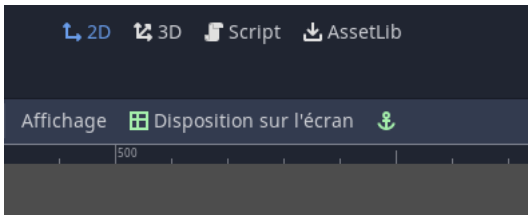


Figure 9.14: Disposition des boutons sur l'écran.

Ajouter ensuite des nœuds enfant de type *Button* à ce *VBoxContainer*, un pour chaque bouton souhaité dans le menu. Dans un premier temps, nous ajouterons un bouton *Launch Game* et un bouton *Quit*, centrés sur l'écran. On peut ajouter un Sprite et une police différente pour nos boutons afin de customiser le menu, mais nous nous en passerons ici.

Attacher un script *StartMenu.gd* au nœud principal, puis connecter le signal *_on_LaunchButton_pressed* au nœud *StartMenu*. De même, connecter le signal *_on_QuitButton_pressed* à ce nœud. Modifier ensuite le script *StartMenu.gd* comme indiqué en Figure 9.15. La première fonction indique que l'on chargera la scène *Level1.tscn* en appuyant sur *LaunchButton*. La seconde indique qu'on quittera le jeu en appuyant sur *QuitButton*.

```
extends Control

func _on_LaunchButton_pressed():
    get_tree().change_scene("res://Level1.tscn")

func _on_QuitButton_pressed():
    get_tree().quit()
```

Figure 9.15: Script du *StartMenu*.

On peut également ajouter la fonction *ready()* présentée en Figure 9.16 afin de permettre le choix des boutons au clavier : appuyer sur haut et bas permet d'aller respectivement vers le bouton du dessous ou du dessous, appuyer sur entrée revient à cliquer sur le bouton. Pour régler l'ordre de défilement des boutons au clavier, dans l'onglet *Focus* de l'inspecteur du *LauchButton*, il faut régler le *NeighbourBottom* et *NeighbourTop* (puisqu'on a que deux boutons, on veut passer de l'un à l'autre en appuyant sur haut ou bas) comme *QuitButton*, et réciproquement, comme en Figure 9.17.

```gdscript
func _ready():
    $VBoxContainer/LaunchButton.grab_focus()
```

Figure 9.16: Sélection des boutons du menu au clavier.

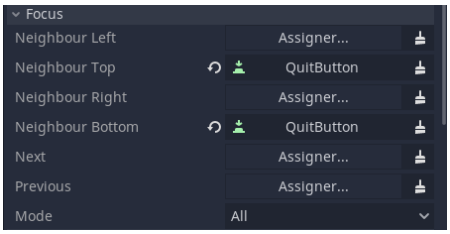


Figure 9.17: Choix des voisins du *LaunchButton*.

9.4.2. Menu Game Over. On veut maintenant faire en sorte que lorsque notre personnage meurt (soit parce que ses points de vie sont à 0, soit parce qu'il tombe indéfiniment), apparaisse un menu *Game Over*, qui relance la scène courante (dans notre cas *Level1.tscn*) au bout de 5 secondes.

Créer une nouvelle scène *GameOver.tscn*, dont l'arborescence est présentée en Figure 9.18. La texture utilisée pour le Sprite est présentée en Figure 9.19

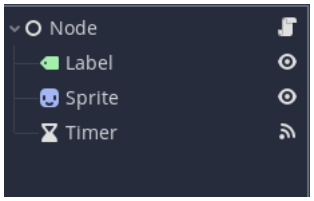


Figure 9.18: Arborescence de *GameOver.tscn*.

Figure 9.19: *Sprite* du menu *Game Over*.

Créer un script *Node.gd* associé au nœud *Node*, puis connecter le signal *_on_Timer_timeout* à ce nœud. Modifier la fonction *_on_Timer_timeout* et ajouter les fonctions *ready()* et *physics_process* comme présentées en Figure 9.20.

```gdscript
var seconds : int

func _ready():
    $Timer.start(5)

func _physics_process(delta):
    seconds = $Timer.get_time_left()
    $Label.text = "Restart in " + str(seconds) + " seconds"

func _on_Timer_timeout():
    get_tree().change_scene("res://Level1.tscn")
```

Figure 9.20: Script *GameOver.gd*.

La fonction *ready* déclenche un *Timer* à 5 secondes. Dans la fonction *physics_process*, on récupère le temps restant dans une variable *seconds*, et on modifie le texte du Label pour indiquer "Restart in x seconds" où x est la valeur contenue dans la variable *seconds*. Lorsque le *Timer* est arrivé à 0, on recharge la scène *Level1.tscn*.

9.5. CHECKPOINTS

On repart du projet de la Section 9.4 avec le menu *Start* et le menu *Game Over*. On veut ajouter un *Checkpoint* dans notre *Level1.tscn* qui fait en sorte que, après la prochaine apparition du *Game Over*, on repartira de ce *Checkpoint*.

Ajouter en fils du nœud principal un nœud de type *Area2D* renommé en *Checkpoint*. Lui ajouter en enfant un *AnimatedSprite* avec un *SpriteFrame* contenant l'animation *Idle* constituée des 3 images présentées en Figure 9.21, et une *CollisionShape2D* circulaire, qui matche la forme du Sprite. On le place sur la plateforme horizontale après le premier Koopa. Enregistrer la scène *Checkpoint.tscn* pour pouvoir éventuellement instancier d'autres *Checkpoints*.

Figure 9.21: Images d'animation du *Checkpoint*.

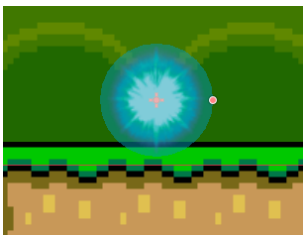


Figure 9.22: *Area2D* et zone de collision autour du *Checkpoint*.

Pour rendre notre *Checkpoint* fonctionnel, nous avons besoin d'enregistrer dans une variable la position du dernier *Checkpoint* visité. Cependant, après passage vers le menu *Game Over* et rechargement de la scène *Level1.tscn*, nous voulons que la valeur de cette variable puisse être conservée et non pas restaurée. Il faut donc utiliser une **variable globale**, qui sera définie dans un script indépendant, non associé à un nœud de la scène, appelé un **Autoload**.

- Créer un nouveau script *Autoload.gd* en cliquant droit dans le gestionnaire de scène, puis en sélectionnant *Nouveau script*. Ce script est donc actuellement présent dans notre projet, tout en étant attaché à aucun nœud. Dans *Autoload.gd*, définir une variable *last_checkpoint*, de type *Vector2*. Le script *Autoload.gd* ne contiendra que la définition de cette variable, voir Figure 9.23. Il faut ensuite ajouter ce script comme **Autoload**. Pour ce faire, aller dans les paramètres du projet et dans l'onglet *Autoload*, chercher le script *Autoload.gd* et cliquer sur Ajouter. Vérifier que la case *Singleton* est bien cochée. La fenêtre de gestion des *Autoload* est présentée en Figure 9.24.

- Dans la scène *Level1.tscn*, ajouter un nœud de type *Position2D* en enfant du nœud principal, et centrer ce nœud *Position2D* sur le personnage de Mario. Nous récupèrerons la position de ce nœud pour connaître la position initiale de notre personnage, avant passage à un checkpoint. On peut donc initialiser la variable *last_checkpoint* à *node_position.position*, où *node_position* désigne ce nœud *Position2D*. Dans la fonction *ready()* du script *Mario.gd*, ajouter l'instruction

  ```
  Autoload.last_checkpoint = node_position.position
  ```

  Notez qu'après ajout du script *Autoload.gd* dans les paramètres du projet, on peut accéder aux variables du script avec la commande *Autoload.variable*. En fait, cette étape d'initialisation n'est pas indispensable, mais il est préférable de l'ajouter, d'autant que cela permet également de scripter la position d'apparition du personnage à l'origine dans la scène.

```gdscript
1  extends Node
2
3  var last_checkpoint : Vector2
```

Figure 9.23: Script *Autoload.gd*.

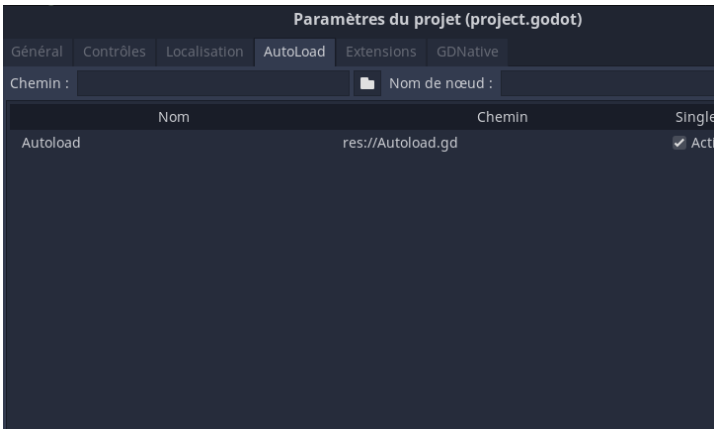


Figure 9.24: Fenêtre de gestion des *Autoload*.

- Ajouter un script *Level1.gd* au nœud principal de la scène *Level1.tscn*. Ce script va permettre de régler la position de notre personnage dans la scène en fonction de *Autoload.last_checkpoint*. Il est présenté en Figure 9.25.
- Ajouter un script *Checkpoint.gd* au nœud *Checkpoint*. Connecter le signal *on_body_entered* de l'*Area2D* associée au nœud *Checkpoint*, puis modifier le script comme présenté en Figure 9.26.

```gdscript
1  extends Node2D
2
3  func _enter_tree():
4      if Autoload.last_checkpoint:
5          $Mario.global_position = Autoload.last_checkpoint
```

Figure 9.25: Script *Level1.gd*.

```gdscript
1  extends Area2D
2
3  func _on_CheckPoint_body_entered(body):
4      if body.get_name()=="Mario":
5          queue_free()
6          Autoload.last_checkpoint = global_position
```

Figure 9.26: Script *Checkpoint.gd*.

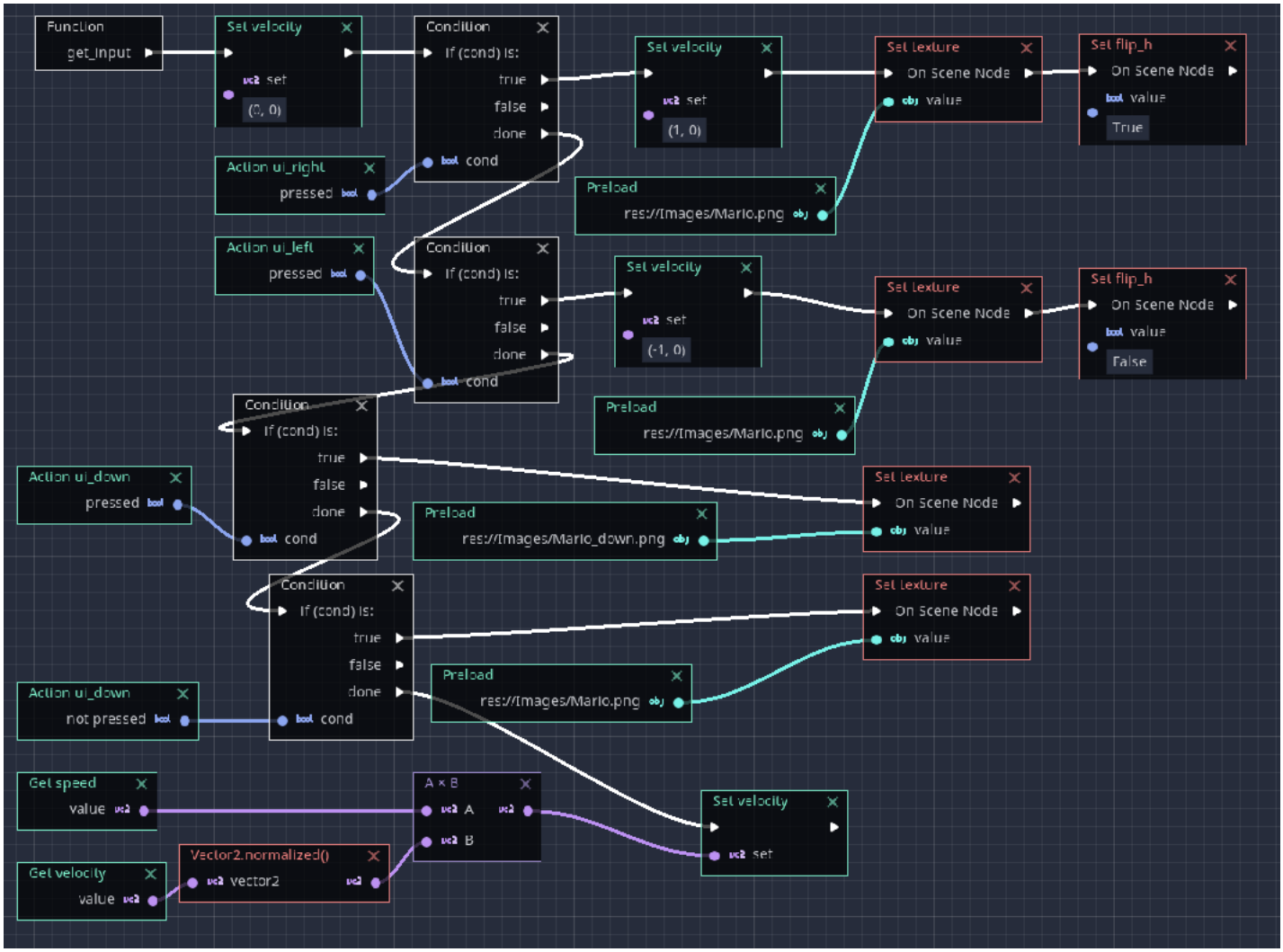
À la ligne 5 du script *Checkpoint.gd*, au lieu de supprimer directement le nœud *Checkpoint*, on pourrait aussi ajouter une animation comme en Section 7.2.3. Après exécution du projet, lorsque le Mario passe par le *Checkpoint*, celui-ci disparaît, et si Mario meurt par la suite, le rechargement de la scène après le menu *Game Over* se fera à partir de la position du *Checkpoint*. Vous pouvez également dupliquer le nœud *Checkpoint* pour obtenir d'autres points de sauvegarde; après chaque mort, le Mario reprendra toujours la position du dernier *Checkpoint* traversé.

Chapitre 10

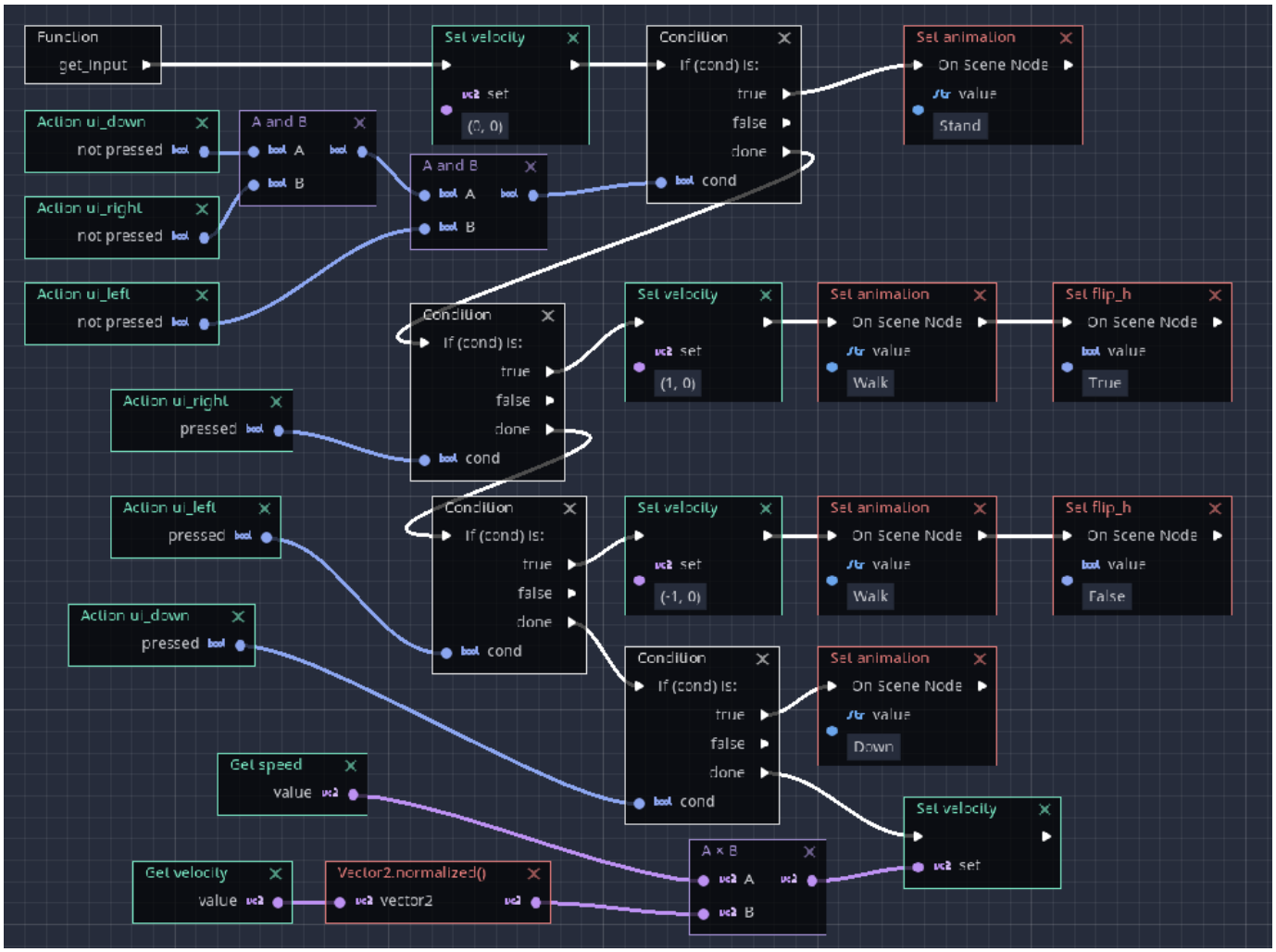
Quelques VisualScripts

10.1. VisualScripts du Chapitre 6

10.1.1. Section 6.2.2 (Mouvement de Mario avec plusieurs images). Pour obtenir la fonction *Set texture*, il faut sélectionner le nœud Mario et faire glisser/déplacer le champ *Texture* depuis l'inspecteur; idem pour *Set flip_h*.



10.1.2. Section 6.2.3 (Déplacement de Mario avec animations Stand, Walk et Down). Attention, si vous reprenez le *VisualScript* de la section précédente 6.2.2, pensez à changer les fonctions *Set flip_h*, puisque celles-ci sont associées au nœud *Sprite*, et non *AnimatedSprite*; il faut donc bien reprendre les fonctions *Set flip_h* associées à ce nœud. Pour ajouter les fonctions *Set animation*, sélectionnez le nœud *AnimatedSprite*, choisissez l'animation que vous souhaitez dans le champ *Animation*, et faites ensuite glisser-déplacer ce champ dans le *VisualScript*.



10.1.3. Section 6.2.4 (Déplacement de Mario dans un décor). Vous pouvez réutiliser le *VisualScript* utilisé pour la Section 6.2.3 (voir ci-dessus), et garder la fonction *get_input* intacte. Il faut ensuite changer la fonction *physics_process* comme en Figure 10.1.

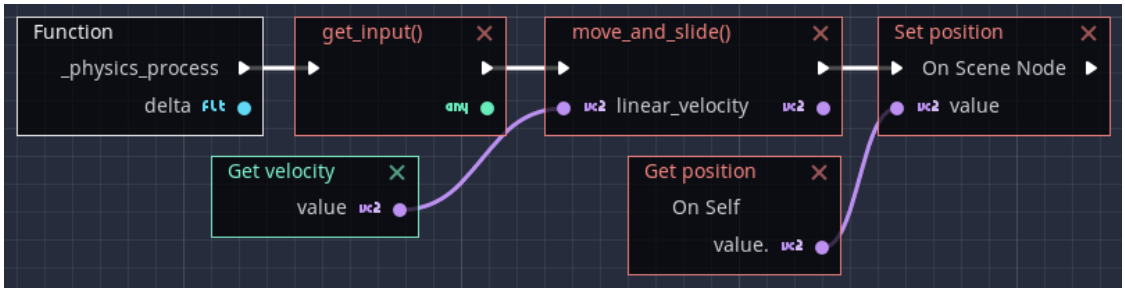


Figure 10.1: Fonction *physics_process* pour suivre le déplacement de Mario à la caméra.

La fonction *Get position* mentionne *On self* pour indiquer qu'elle s'applique au nœud auquel est attaché le script (c'est-à-dire le nœud *Mario*), tandis que la fonction *Set position* mentionne *On Scene Node* pour indiquer qu'elle s'appliquer à un nœud de la scène (le nœud *Camera2D*).