



*Rapport d'élève ingénieur
Projet de 3^{ème} année
Filière : Génie logiciel et systèmes informatiques*

Développement de mini-jeux pour Android

avec Unity

Présenté par : DESNOS Nathan
 VILLEDIEU DE TORCY Mathieu

Responsable ISIMA : YON Loïc

OCTOBRE 2022 - MARS 2023
120h

Remerciements

Nous tenons à remercier M. Loïc YON, notre tuteur de projet, pour nous avoir encadrés et avoir pris le temps de discuter avec nous de l'avancée de notre projet.

Table des figures

1.1	Console Pocket Dream Console (PDC)	2
1.2	Capture d'écran du menu principal de la console originale	2
1.3	Diagramme de Gantt des tâches réalisées	4
1.4	Composant Transform	5
2.1	Architecture globale d'un jeu	8
2.2	Architecture de l'interface	9
2.3	Contour de la console	10
2.4	Boutons	10
2.5	Menu pause	11
2.6	Diagramme du Launcher (menu de sélection des jeux)	11
2.7	Launcher (menu de sélection des jeux)	12
2.8	Interface de la console complète : Samsung Galaxy Note8	13
2.9	Interface de la console complète : LG nexus 5	13
3.1	Sprite du Player	15
3.2	Animator Controller du Player (Animation Tree)	16
3.3	Plateforme simple : EdgeCollider2D et BoxCollider2D	18
3.4	Sprite de la plateforme tapis roulant	19
3.5	Animation clip de la plateforme tournante	19
3.6	Sprite de la plateforme tournante	20
3.7	Sprite de la plateforme avec des pics	20
3.8	Animation clip de la plateforme de saut	20
3.9	Sprite de la plateforme de saut	21
3.10	Diagramme du GameObject Carte	22
3.11	Script Plateforme Générateur	22
3.12	Disposition des plateformes dans la scène	23
3.13	Positionnement sur l'axe des X	24
3.14	Extremités du composant BoxCollider2D d'une plateforme	24
3.15	Distance entre 2 plateformes	25
3.16	Diagramme du GameObject de l'interface du jeu 100 FLOOR	26
3.17	Interface de jeu 100 FLOOR complète	26
3.18	Sprite de barre de vie	26
3.19	Barre de vie : moitiée pleine	27
3.20	Arrière-plan	28
3.21	Diagramme du GameObject Background	28
3.22	Animation de la couche Maisons	28
3.23	Jeu 100 FLOOR complet dans l'application	29
4.1	Obstacles du niveau BP	30
4.2	Effet de profondeur	31
4.3	Personnage joueur (jeu original)	32
4.4	Superposition lors de la pose de la bombe (jeu original)	32
4.5	Explosion d'une bombe de portée 1 case (jeu original)	33

Résumé

Dans le but de faciliter l'accès aux vieux **jeux vidéo** de la petite console portable **PDC**, nous avons pour objectif de redévelopper quelques-uns de ses **mini-jeux** pour **smartphones Android**. Nous nous concentrerons en particulier sur deux jeux : un **endless-runner** (100 FLOOR) et un jeu d'action dans un labyrinthe (B.P.). Le projet vise à proposer une expérience de jeu similaire à celle des jeux originaux sur les **smartphones**.

Pour développer ces jeux, nous utilisons le moteur de jeu **Unity** et le langage de programmation C# associé.

À ce jour, nous avons créé une application qui reproduit l'apparence de la console **PDC**. Cette application contient un jeu complet en termes de fonctionnalités de gameplay (100 FLOOR), et un autre jeu qui ne possède que des briques de gameplay (B.P.). Néanmoins, l'application rencontre encore quelques bogues mineurs et nécessite des travaux de développement supplémentaires pour assurer la jouabilité des deux jeux dans leur intégralité.

Mot-clés : Unity, smartphone, Android, jeux vidéo, mini-jeux

Abstract

In order to make access to the old **video games** from the small **PDC** portable gaming console easier, we aim to redevelop some of its **mini-games** on **Android smartphones**. In particular, we are focusing on two games : an endless-runner (100 FLOOR) and a maze-action game (B.P.). The project aims to provide a gameplay experience similar to that of the original games, but on **smartphones**.

In order to develop these games, we use the **Unity** game engine and the associated C# programming language.

To this date, we have created an application that replicates the look of the **PDC** console. This application contains one game that is complete in terms of gameplay features (100 FLOOR), and another game that only has gameplay elements (B.P.). However, the application has some minor bugs and requires further development work to ensure the playability of both games in their entirety.

Keywords : Unity, smartphone, Android, video games, mini-games

Table des matières

Remerciements	i
Tables des figures	ii
Résumé	iii
Abstract	iv
Table des matières	v
Introduction	1
1 Présentation du sujet	2
1.1 Console	2
1.2 Environnement et objectifs	3
1.3 Organisation du travail	4
1.4 Concepts d'Unity	5
2 Composants communs	8
2.1 Architecture globale de l'application	8
2.2 Architecture de l'interface	9
2.3 Interface de jeu	12
2.4 Résultats	12
3 Premier jeu : 100 FLOOR	14
3.1 Gameplay	14
3.2 Player	15
3.3 Plateformes	18
3.4 Gestion de la carte	22
3.5 Interface du jeu	26
3.6 Arrière-plan	28
3.7 Résultats	29
4 Deuxième jeu : B.P.	30
4.1 Gameplay	30
4.2 Agencement d'un niveau	30
4.3 Personnages et ennemis	31
4.4 Bombe	33
4.5 Résultats	34
Conclusion	35
Bibliographie	v
Glossary	vi

Introduction

Les consoles portables de poche étaient très populaires pour leur portabilité, mais ont été remplacées par les smartphones. Cependant, les jeux de ces consoles ne sont pour la plupart pas disponibles sur les smartphones ou sont très différents de l'original. C'est le cas de la console portable [PDC](#). Pour permettre aux joueurs de rejouer à ces jeux sur smartphone, nous avons décidé de recréer certains d'entre eux pour les appareils Android en utilisant le [moteur de jeu Unity](#).

L'application permettra de jouer à deux jeux issus cette console : 100 FLOOR (un [endless-runner](#)) et BP (un jeu d'action et de labyrinthe). Elle utilisera les images des jeux originaux pour offrir une expérience de jeu fidèle. En plus de permettre de jouer à ces deux jeux, l'application doit offrir une expérience de jeu similaire à celle à la console [PDC](#) en affichant une représentation visuelle de celle-ci sur le smartphone, avec son contour et ses boutons. L'application doit offrir une architecture permettant le développement indépendant de chaque jeu, ainsi que leur ajout facile dans l'application.

Nous verrons dans un premier temps une présentation plus détaillée des objectifs du projet, ainsi que des concepts importants du [moteur de jeu Unity](#). Nous présenterons ensuite l'architecture du projet et l'agencement des composants communs aux différents jeux. Enfin nous expliquerons le fonctionnement et le développement de chacun des jeux.

1 Présentation du sujet

1.1 Console

La Pocket Dream Console, est une petite console portable développée par Conny dans les années 2004-2005. Elle est vendue en France sous le nom de « PDC ». Cette console de poche (10,3cmx5,2cmx2,4cm) possède un écran LCD couleur de deux pouces et de plusieurs boutons :

- 4 flèches directionnelles
- bouton A
- bouton B
- bouton Menu
- 2 gâchettes sur la partie supérieure de la console
- bouton de contrôle du volume sur le côté droit de la console
- interrupteur On/Off sur la partie inférieure de la console



FIGURE 1.1 – Console PDC

La console existe en plusieurs couleurs et contient de nombreux jeux suivant les versions (30, 50, et 100 jeux).



La console propose des mini-jeux répartis sur le menu principal (cf. [figure 1.2](#)) en plusieurs catégories : Puzzle, Action, Casino, Table, Racing, Relax.

Ces catégories couvrent une grande variété de types de jeux comme des jeux d'énigmes, de réflexion, de combat, de course, de gestion, de plateformes, de hasard, offrant une large gamme de choix aux joueurs. Ces jeux proposent un gameplay simple et ne possèdent pas de sauvegarde.

FIGURE 1.2 – Capture d'écran du menu principal de la console originale

1.2 Environnement et objectifs

La PDC étant une console portable, l'objectif du projet est de redévelopper certains de ses jeux sur smartphone Android. Pour cela, nous avons sélectionné deux jeux sur lesquels nous nous concentrerons.

Le premier est disponible sur toutes les versions de la console, il s'agit du jeu 100 FLOOR, un jeu de plateforme où le joueur doit rester en vie le plus longtemps possible en restant sur des plateformes en mouvement.

Le deuxième est B.P. disponible sur les versions 50 et 100 jeux. Le jeu est de type « Bomberman », dans lequel le joueur doit éliminer les ennemis à l'aide de bombes placées dans un labyrinthe. Pour se déplacer sur la carte, le joueur peut détruire certains murs du labyrinthe en posant les dites bombes. Il doit dans le même temps éviter de toucher les explosions de ses bombes ainsi que les ennemis et leurs attaques, sous peine de perdre une vie et de recommencer le niveau.

Le but du projet est de réaliser une application mobile qui se rapproche au plus près de ce que propose la console. Pour cela, l'interface de la console avec ses boutons sera affichée sur l'écran du smartphone. Le petit format de la console permettra facilement de représenter l'entièreté des contours de la console et de ses détails.

Pour réaliser ce projet, le [moteur de jeu](#) Unity [1] sera utilisé, puisqu'il permet un développement simple de différents types de jeux vidéo avec un déploiement facile sur mobile. De plus, cet outil propose des fonctionnalités de développement intéressantes en termes de débogage ou de visualisons du jeu sur téléphone, notamment grâce à l'application Unity Remote 5 [2] qui permet de tester en temps réel le prototype sur smartphone.

Ici, le projet se concentre sur les smartphones Android. Dans un premier temps, l'ensemble de l'application est prévu et testé pour fonctionner correctement sur un Samsung Galaxy Note 8, mais devra par la suite s'adapter à n'importe quel écran de smartphone Android.

Ce [moteur de jeu](#) est depuis quelques années une référence dans le domaine du développement de jeux vidéo et particulièrement le développement mobile. Il représente environ 60% des jeux mobiles gratuits d'après une étude menée en 2021 [3]. C'est pour nous l'occasion de prendre en main l'outil afin de découvrir et d'exploiter des concepts plus ou moins spécifiques à Unity dans le domaine du jeu vidéo. [4, 5]

1.3 Organisation du travail

L'objectif étant de réaliser deux mini-jeux, nous avons décidé de nous concentrer chacun sur un jeu. Mathieu a réalisé le premier jeu 100 FLOOR et Nathan le deuxième B.P.

En plus des deux jeux, Mathieu a reproduit l'apparence de la console dans l'application en incluant ses contours, ses contrôles, son système de pause pour les jeux et son menu principal qui permet de sélectionner le jeu à lancer. Ces éléments sont essentiels pour offrir une expérience de jeu similaire à celle de la console originale. La gestion des contrôles a été réalisée en début de projet pour permettre une mise en place rapide des jeux, tandis que les autres éléments, qui ont une priorité moindre et n'étant pas essentiels pour le gameplay, ont été développés vers la fin du projet.

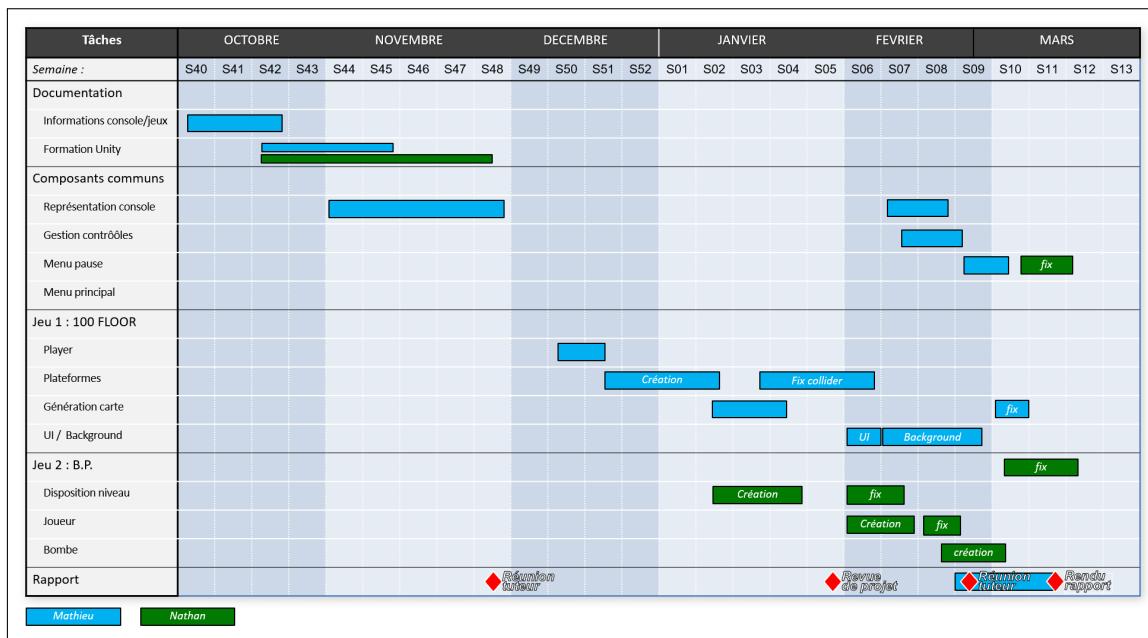


FIGURE 1.3 – Diagramme de Gantt des tâches réalisées

La réalisation d'un jeu vidéo, même s'il s'agit que d'un jeu simple, nécessite beaucoup de temps lorsque l'on souhaite mener le développement jusqu'à son terme. Initialement, nous avons envisagé de créer deux ou trois mini-jeux par personne, mais nous avons rapidement opté pour un seul par personne afin de nous concentrer sur la réalisation de nos mini-jeux respectif et de ne pas précipiter le développement en produisant des jeux incomplets.

1.4 Concepts d'Unity

Unity est un [moteur de jeu](#) pour développer des jeux 3D et 2D. Dans ce projet nous nous concentrons que sur la partie 2D de ce [moteur de jeu](#), même si la plupart des concepts possède leur équivalent en 3D. Le [moteur de jeu](#) Unity propose un environnement de développement complet en fournissant des composants qui implémentent déjà certaines fonctionnalités. Il abstrait une partie de la complexité technique du développement d'un jeu vidéo (grâce aux composants d'affichage, de physique et autres) pour permettre aux développeurs de se concentrer sur la réalisation d'un jeu vidéo et sur ses aspects de gameplay.

GameObject

Le concept de GameObject est central dans Unity. En effet, il s'agit de l'élément de base qui représente tous les objets et entités du jeu.

Ils peuvent être organisés en hiérarchie parent-enfant pour faciliter leur manipulation dans une scène. Ainsi un GameObject peut être composé de plusieurs autres GameObjects. Chaque GameObject est composé d'au moins un composant Transform, qui sert à le situer dans la scène où il est défini. Ces GameObjects peuvent également contenir d'autres composants pour définir leurs apparences, leurs comportements et leurs fonctionnalités.

Que ce soit une caméra, une carte, un cube ou tout autres entités, il s'agit de GameObject ayant au moins un Transform et des composants spécifiques pour définir leur fonctionnement.

Prefab

Unity utilise des Prefab pour stocker les GameObjects avec tout leurs composants et toutes leurs propriétés. Cela facilite la réutilisation des GameObjects et permet de facilement les instancier dans le jeu.

Transform

Le Transform est le composant essentiel pour définir la position, la rotation et la taille d'un GameObject dans une scène. Il est composé de trois vecteurs 3D.

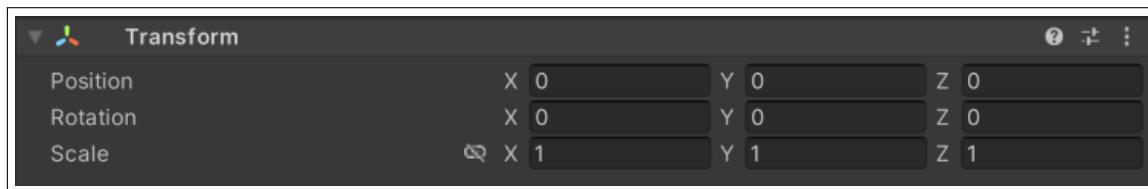


FIGURE 1.4 – Composant Transform

Dans le cas de jeu 2D, la composante Z, pour la position et la taille (scale), est évidemment inutile.

Rigidbody2D

Le Rigidbody2D est un composant qui ajoute de la physique au GameObject en lui attribuant une masse, une vitesse et une accélération. Cela permet de simuler la gravité, les collisions ou bien d'appliquer des forces sur celui-ci. Pour qu'il puisse interagir avec d'autres GameObjects, il doit être associé à un Collider2D.

Collider2D

Le Collider2D est un composant qui est utilisé pour donner une forme physique à un GameObject. Il est utile pour détecter les contacts et collisions physiques entre GameObjects.

Il possède une propriété Trigger qui permet de définir ce composant comme traversable. Cela permet de déclencher une action lorsque le Collider entre en contact avec un autre collider de GameObject sans pour autant provoquer une interaction physique de collision entre les deux GameObjects. Cette propriété est souvent utilisée pour la gestion des événements de jeu lors de contacts entre un objet et une zone définie.

Les Collider2D envoient des messages à Unity, qui appelle automatiquement certaines méthodes dans les scripts attachés aux objets impliqués dans une collision.

Les méthodes appelées lors des collisions sont :

- *OnCollisionEnter2D* : lorsqu'un Collider2D entre en contact avec le Collider2D de cet objet
- *OnCollisionExit2D* : lorsqu'un Collider2D arrête de toucher le Collider2D de cet objet
- *OnCollisionStay2D* : tant qu'un Collider2D touche le Collider2D de cet objet

Les Collider2D, qui ont la propriété Trigger activée, déclenchent eux les méthodes suivantes :

- *OnTriggerEnter2D* : lorsqu'un autre objet entre dans le Collider2D attaché à cet objet
- *OnTriggerExit2D* : lorsqu'un autre objet sort du Collider2D attaché à cet objet
- *OnTriggerStay2D* : tant qu'un Collider2D est dans le Collider2D attaché à cet objet

Il existe aussi différents types de Collider2D, pour s'adapter au mieux à la forme des GameObjects, pour détecter les collisions :

- Circle Collider 2D : forme circulaire
- Box Collider 2D : forme rectangulaire
- Polygone Collider 2D : forme libre
- Edge Collider : forme libre, ou forme non fermée
- Capsule Collider 2D : forme de capsule
- Composite Collider 2D : fusion de plusieurs Collider2D

SpriteRenderer

Le SpriteRenderer est un composant qui permet d'afficher sur la scène un élément graphique, une image ou une texture. Il possède plusieurs propriétés, notamment ce qui concerne les couches d'affichages. Il est possible de choisir sur quelle couche le [sprite](#) sera rendu, et son ordre d'affichage dans cette couche. Cela permet de superposer les éléments graphiques et de les ordonner facilement (fond, décors, obstacles, personnage...) pour choisir quel élément sera affiché devant quel autre élément.

Animation-Animator

Pour les animations, Unity propose un composant Animator. Il est composé d'un Animator Controller qui a la charge de définir les transitions entre les différentes animations suivant certains paramètres. Les différentes animations sont éditées dans Unity via des Animation Clip. Ceux-ci permettent de faire varier plusieurs paramètres au cours du temps, comme le [sprite](#), le collider, leur position, leur taille ou bien de déclencher des appels de méthodes.

AudioSource

L'

Script

Bien que de nombreux composants soient disponibles dans Unity, il existe un composant Script qui permet d'éditer du code en C# (version 9.0). [6] Cela permet de développer le comportement d'un GameObject en créant de nouvelles classes. Ces classes C# dérivent de la classe [MonoBehaviour](#). Il est donc possible d'utiliser certaines méthodes qui seront automatiquement appelée par Unity. C'est le cas pour les méthodes déclenchées par les Collider2D. Il est donc facile d'exécuter du code lorsque deux GameObjects entrent en collision.

Certaines méthodes sont importantes, puisqu'elles sont appelées à des moments spécifiques :

- *Awake* : est appelée quand l'instance du script est chargée
- *Start* : est appelée juste avant le premier appel du script à la méthode d'*Update*
- *Update* : est appelée à chaque [frame](#)
- *FixedUpdate* : est appelée suivant une fréquence fixe
- *OnEnable/OnDisable* : sont appelées à l'activation/désactivation du script

Il est important de noter que la méthode *Update* dépend de la fréquence des [frames](#). Cette fréquence est directement liée à la puissance de la machine qui exécute le jeu. Pour s'affranchir de cette contrainte, il existe deux solutions. Soit multiplier les variables utilisées dans la méthode *Update()* par `Time.deltaTime` (intervalle de temps en secondes entre la dernière [frame](#) et celle actuelle) afin d'uniformiser les valeurs entre une machine peu puissante et une très puissante. Soit effectuer les calculs dans la méthode *FixedUpdate()*, qui elle ne dépend pas de la puissance de la machine puisqu'elle est appelée à la fréquence du système physique qui est fixe. Il est préférable d'effectuer les calculs de physique dans cette dernière méthode.

Canvas

Le Canvas est un composant qui permet de définir une zone où tous les éléments de l'[user interface \(UI\)](#) sont disposés et rendus. Chaque GameObject contenant des éléments UI doit être un enfant du GameObject contenant ce composant. Ces éléments peuvent être des boutons, des images ou bien du texte.

RawImage

Le RawImage est un simple composant qui permet d'afficher une image comme un élément [UI](#).

EventTrigger

EventTrigger est un composant qui reçoit des événements du système d'événements (tels que des déplacements de pointeur, des clics, ...) et appelle des fonctions définies pour chaque événement.

2 Composants communs

2.1 Architecture globale de l'application

Le projet se concentre sur seulement 2 mini-jeux, cependant pour pouvoir ajouter facilement d'autres mini-jeux, il faut que l'application soit facilement modulable et permette de réutiliser les éléments communs tous les jeux.

Les éléments communs à tous les jeux sont :

- * Le système de gestion des contrôles
- * Le système de pause
- * La partie visuelle représentant la console
- * Le menu principal

Tous ces éléments peuvent être implémentés dans des éléments d'interfaces et sont donc disposés dans un Canvas.

La scène d'un jeu est divisée en deux GameObjects principaux. Le premier est le Canvas qui contiendra tous les GameObjects servant pour l'interface, aussi bien celle affichant les éléments du jeu que celle affichant la console et ses boutons. Le deuxième GameObject contiendra tous les éléments de logique du jeu, comme le Player, la carte, etc... Evidement chaque GameObject peut lui aussi être composé d'autres GameObjects.

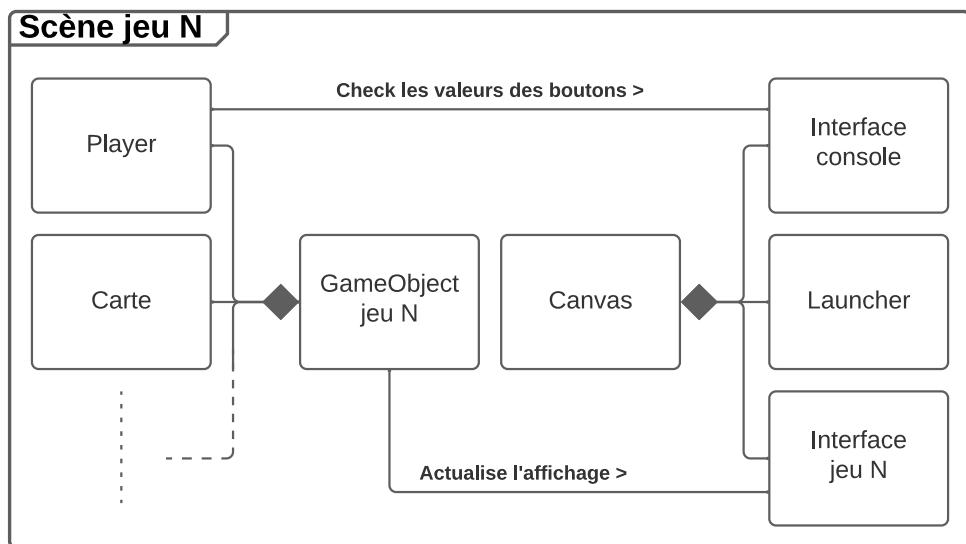


FIGURE 2.1 – Architecture globale d'un jeu

Avec cette architecture de projet, chaque mini-jeu est associé à une scène. Pour en ajouter facilement un autre, il faut rajouter la scène correspondant au nouveau mini-jeu dans le projet. Cette scène sera composée du GameObject jeu correspondant à la logique et au gameplay de celui-ci et du Canvas contenant l'interface pour ce jeu spécifique.

Dans cette solution, les communications entre les objets se font uniquement entre les éléments du GameObject de jeu et les éléments du Canvas. Ce sont les objets du jeu qui viennent vérifier l'état des commandes, s'ils en ont besoin. C'est également eux qui viennent appeler les méthodes de mise à jour des éléments d'interface en fonction des nouvelles valeurs à afficher. De cette façon, tous les éléments de l'interface contenus dans le Canvas n'ont pas besoin de connaître l'implémentation des objets du jeu.

2.2 Architecture de l'interface

L'interface de l'application est représentée par un Canvas. Ce Canvas est composé de 3 GameObjects :

- ➔ Interface console : contient les éléments nécessaires pour afficher le contour, l'ensemble des boutons et le système de pause de la console
- ➔ Interface de jeu : contient les éléments du jeu tels que le score ou la vie du joueur
- ➔ Launcher : contient les éléments du menu principal

L'objectif de cette structure est de pouvoir changer facilement le GameObject *Interface de jeu* dans le Canvas en gardant le reste des éléments communs à tous les jeux (*Interface console* et *Launcher*).

Dans ce projet, les gâchettes, le bouton de volume, ainsi que l'interrupteur de la console ne sont pas modélisés, car ceux-ci se trouvent sur les côtés de la console. Dans la plupart des jeux, ces gâchettes ont souvent les mêmes fonctionnalités que les autres boutons. Elles déclenchent les mêmes actions que les boutons flèches gauche et droite, de même que les boutons A et B et ne sont donc pas nécessaires. De plus, dans le cas des deux mini-jeux reproduits, elles ne sont pas utilisées.

2.2.1 Interface de la console

Le GameObject *Interface console* est composé de plusieurs autres GameObject. Il gère le système de contrôle des boutons et de pause du jeu.

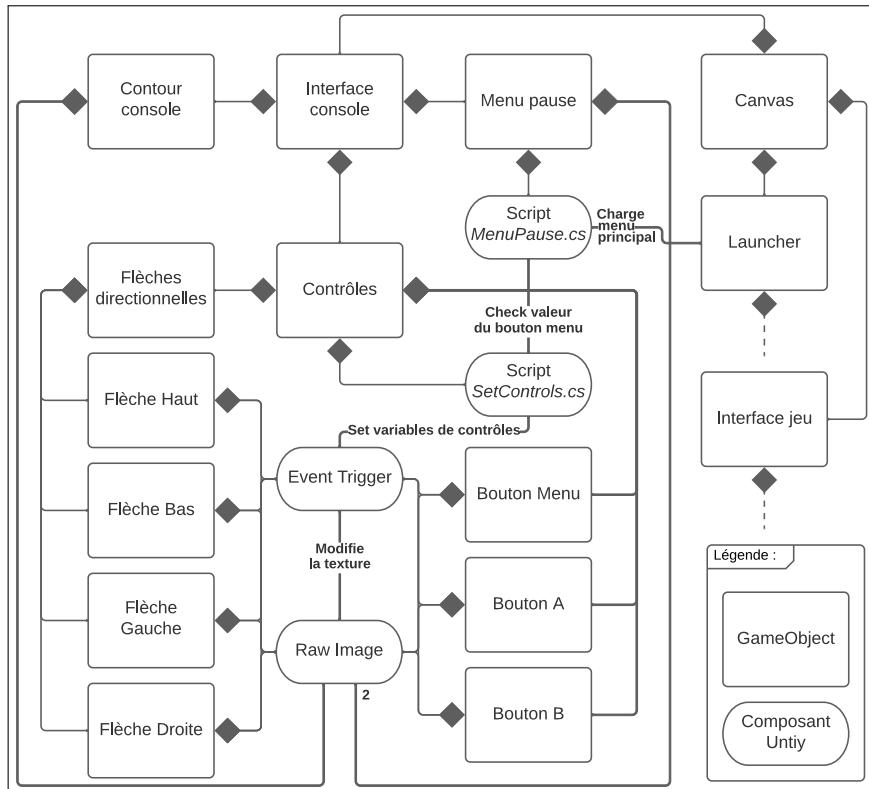


FIGURE 2.2 – Architecture de l'interface

L'objet *ContourConsole*, qui contient juste une image, affiche la structure de la console. Il s'agit juste d'un élément visuel. Aucune interaction n'est possible avec. (cf. figure 2.3)

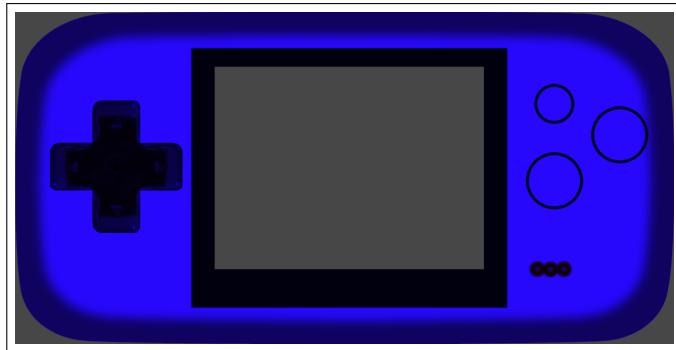


FIGURE 2.3 – Contour de la console

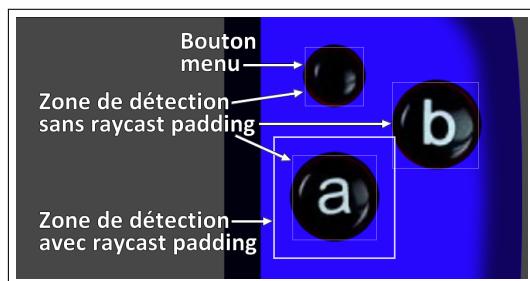
2.2.2 Contrôles

Le GameObject *Contrôles* est composé de plusieurs objets représentant les différents boutons de la console. Pour gérer leur état, il possède un script *SetControls.cs* dans lequel une propriété (booléen) est associée à chaque bouton. De plus, deux autres propriétés (float) sont présentes : *horizontalAxis* et *verticalAxis*. Ces deux propriétés sont mises en place pour connaître respectivement la direction des flèches gauche-droite et haut-bas grâce aux trois valeurs qu'elles peuvent prendre : -1, 0 et 1. Cela facilite l'utilisation et la lecture des contrôles en ne lisant que deux propriétés pour connaître l'état des flèches directionnelles et peut cela être utile dans certain type de jeu.

Les différents GameObjects des boutons sont composés de la même façon. Ils possèdent chacun un composant RawImage pour afficher le *sprite* du bouton et un composant EventTrigger pour détecter les *inputs* sur ce *sprite*. Quatre événements sont détectés par le composant EventTrigger :

- ☒ Pointer Up
- ☒ Pointer Down
- ☒ Pointer Enter
- ☒ Pointer Exit

Les événements Pointer Up/Down se déclenchent lorsqu'il y a respectivement un relâchement ou une pression sur l'image et appellent le setter du bouton correspondant, dans le script *SetControls.cs*, avec la valeur false/true associée. Sur téléphone mobile lorsque l'on glisse son doigt sur le bouton (en ayant appuyé en dehors avant) ou en dehors du bouton (en ayant bien appuyé dessus avant) ce changement d'état n'est pas reconnu par les deux événement Pointer Up/Down. Pour pallier ce problème, deux événements ont été ajoutés dans l'EventTrigger : Pointer Enter/Exit qui détectent lorsque le pointeur (ici le doigt) rentre ou sort dans la zone du bouton.



Pour améliorer la détection des *inputs* sur les boutons, le *raycast padding* (zone de détection) de chaque image de bouton a été augmenté. Ainsi la zone tactile des boutons est légèrement plus large que l'image.

FIGURE 2.4 – Boutons

Le script *SetControls.cs* contient en temps réel l'état des boutons. De cette façon, chaque script de jeux ayant besoin d'utiliser une entrée utilisateur, doit vérifier la propriété associée dans ce script.

2.2.3 Menu de Pause

Le menu de pause de la console est appelé lorsque le bouton menu (cf. [figure 2.4](#)) est pressé. Quand la pause est déclenchée, le script attaché à cet objet modifie la valeur de la variable `Time.timeScale` sur 0 pour stopper le jeu et toutes les actions en cours. Le `Time.timeScale` correspond à la vitesse à laquelle le temps progresse dans le jeu (1 vitesse normale, 0 jeu en pause). Le script affiche également le [sprite](#) du menu en fonction de son état. Le joueur utilise les flèches haut et bas pour se déplacer dedans (cf. [figure 2.5](#)) et le bouton A pour valider son choix.



FIGURE 2.5 – Menu pause

Si l'option QUIT est choisie, alors le script appelle la méthode pour revenir au menu principal de sélection des jeux défini dans le GameObject *Launcher* (cf. [figure 2.7](#)).

2.2.4 Launcher

L'objet *Launcher* correspond au menu principal de la console permettant de sélectionner le mini-jeu à lancer. Il contient également un système qui permet de charger les différentes scènes.

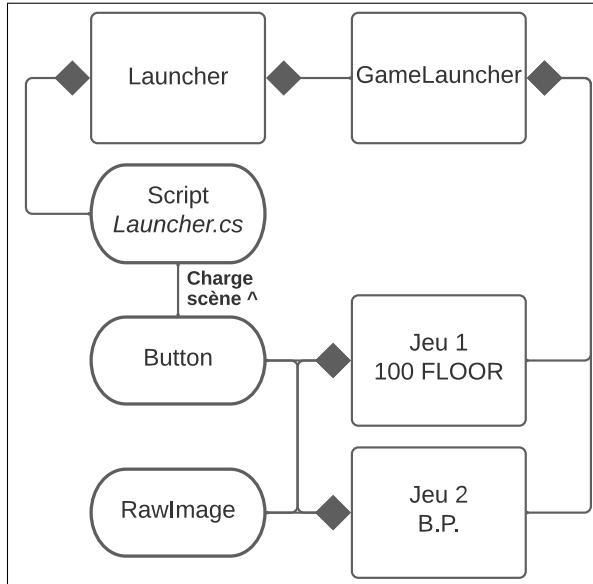


FIGURE 2.6 – Diagramme du Launcher (menu de sélection des jeux)

Il est composé d'un objet (*GameLauncher*) qui contient des boutons pour lancer chaque jeu (objet *Jeu 1 100 FLOOR* et *Jeu 2 B.P.* sur [figure 2.6](#)). Cet objet est activé seulement si l'application se trouve sur la scène du menu principal (cf. [figure 2.7](#)).



FIGURE 2.7 – Launcher (menu de sélection des jeux)

Le script *Launcher.cs* possède une méthode qui s’occupe de charger une scène. C’est cette méthode qui est appelée par les composants Button de chaque jeu avec la valeur de la scène à charger. Une méthode permet de retourner sur le menu principal, en chargeant la scène associée, avec le bouton pause ou lorsque le jeu se termine.

2.3 Interface de jeu

La structure de l’*Interface de jeu* dépend de celui-ci et des paramètres qu’il doit afficher. Cependant, là aussi, ce sont les scripts du jeu qui appellent les méthodes présentes sur les GameObjects dans l’*Interface de jeu* pour mettre à jour les valeurs des textes et images à afficher.

2.4 Résultats

Pour ajouter d’autres mini-jeux, il est nécessaire de modifier l’objet *Launcher*. Il faut rajouter un bouton redirigeant sur la scène correspondant au mini-jeu (cf. figure 2.7). Les boutons de ce menu principal ne sont pas sélectionnables avec les flèches directionnelles et les autres boutons de la console. Ils sont cliquables tactilement grâce au composant Button, qui simplifie la gestion mais ne déclenche pas l’action si le clic est relâché ou commencé en dehors du bouton (contrairement aux boutons classiques de la console avec l’EventTrigger). Cependant bien qu’il soit fonctionnel, ce menu de sélection (l’objet *Launcher*) ne correspond pas à celui de la console d’origine (cf. figure 1.2).

La séparation en deux objets, le Canvas et le GameObject jeu, nous a permis de séparer facilement le projet et de travailler de manière indépendante sur chacun des mini-jeux en facilitant leur intégration par la suite.



FIGURE 2.8 – Interface de la console complète : Samsung Galaxy Note8

Pour l'instant l'adaptation de l'interface de la console ne s'effectue que sur la largeur du smartphone. Pour le modèle de référence du projet, le Samsung Galaxy Note 8, le ratio entre la largeur et la hauteur du téléphone permet d'afficher correctement l'interface. Cependant s'il s'agit d'un smartphone plus petit en haut, comme le LG Nexus 5, alors l'interface du contour de l'écran est tronquée (cf. [figure 2.9](#)).



FIGURE 2.9 – Interface de la console complète : LG nexus 5

Il reste à ajouter un script qui adapte l'application pas seulement par rapport à la largeur du smartphone mais aussi par rapport à sa hauteur, en vérifiant quelle dimension est la plus contrainte.

3 Premier jeu : 100 FLOOR

Le premier jeu reproduit est **100 FLOOR**. Il s'agit d'un jeu de plateforme 2D, de type [endless-runner](#).

3.1 Gameplay

100 FLOOR est un jeu dans lequel le joueur doit s'échapper vers le bas pour éviter d'être touché par les pics situés au-dessus de l'écran qui s'approchent à chaque instant. Pour cela, le joueur doit tomber de plateforme en plateforme tout en restant en vie. Il existe différentes plateformes ayant chacune un effet particulier :

- ⇒ La plateforme simple : qui n'a aucun effet (cf. [figure 3.3](#))
- ⇒ La plateforme tapis roulant : qui déplace le joueur horizontalement (cf. [figure 3.4](#))
- ⇒ La plateforme tournante : qui ralentit le joueur dans sa chute (cf. [figure 3.6](#))
- ⇒ La plateforme avec de pics : qui inflige des dégâts au joueur (cf. [figure 3.7](#))
- ⇒ La plateforme de saut : qui fait rebondir le joueur (cf. [figure 3.9](#))

La partie s'arrête lorsque le joueur sort de l'écran ou bien lorsqu'il n'a plus de vie. Au début de la partie le joueur à dix points de vie.

Les plateformes avec des pics font perdre quatre points de vie au joueur lorsqu'il atterrit dessus. Quant aux pics situés en haut de l'écran, ils diminuent également la vie du joueur de quatre points et le font tomber de sa plateforme. La vie du joueur se régénère automatiquement toutes les deux secondes.

Plus le joueur reste longtemps en vie, plus son score augmente (un point toutes les huit secondes). Son objectif est de faire le score maximal.

3.2 Player

Le *Player* est le GameObject que le joueur contrôle. Pour le déplacer horizontalement le joueur utilise les flèches directionnelles gauche et droite.

3.2.1 Déplacement du Player

Pour permettre un déplacement fluide et cohérent du *Player* en tenant compte des limites de la carte ainsi que des différentes plateformes, Ce GameObject est doté de deux composants Unity : un BoxCollider2D et un Rigidbody2D.

Grâce au Rigidbody2D, il est possible d'appliquer des forces horizontalement sur le *Player*. Il suffit donc de lire les [inputs](#), en particulier la propriété *horizontalAxis* pour connaître la direction dans laquelle déplacer le *Player*.

Il faut ensuite appliquer la force dans la bonne direction au *Player*. Pour cela, il faut effectuer toutes les opérations de calculs et d'affectations de force de déplacement dans la méthode *FixedUpdate()*, du script *Player.cs*, qui est appelée à intervalles de temps fixe. Le calcul s'effectue en plusieurs étapes :

1. Récupération des forces appliquées sur l'objet *Player* dans un vecteur (propriété *velocity* de son Rigidbody2D)
2. Modification de la coordonnée X de ce vecteur en fonction de l'état des flèches gauche et droite (état de la variable *horizontalAxis*)
3. Affectation du nouveau vecteur de force calculé à la propriété *velocity* du Rigidbody2D

Il faut ensuite envoyer à l'Animator des paramètres concernants le déplacement du *Player* pour lancer l'animation adéquate :

4. Définir la valeur de la variable *moveX* dans l'Animator en fonction de la valeur d'*horizontalAxis* (cf. [figure 3.2](#))
5. Définir la valeur de la variable *moveY* dans l'Animator qui indiquera si le *Player* est sur une plateforme ou non (cf. [figure 3.2](#))

Ces différentes étapes permettent de déplacer le *Player* dans la carte, tout en lançant ses animations.

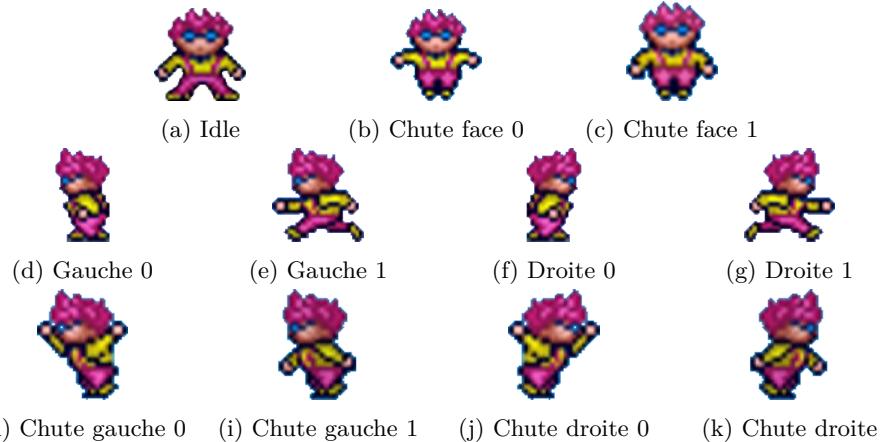


FIGURE 3.1 – Sprite du Player

3.2.2 Animation du Player

Le Player a six animations différentes :

- Idle : player à l'arrêt (cf. [figure 3.1a](#))
- Chute de face (cf. [figure 3.1b](#), [3.1c](#))
- Déplacement sur la gauche (cf. [figure 3.1d](#), [3.1e](#))
- Déplacement sur la droite (cf. [figure 3.1f](#), [3.1g](#))
- Chute en diagonale sur la gauche (cf. [figure 3.1h](#), [3.1i](#))
- Chute en diagonale sur la droite (cf. [figure 3.1j](#), [3.1k](#))

Pour gérer ces animations, le GameObject utilise le component Animator contenant un Animator Controller servant à définir les règles de lecture des animations. Les règles d'animations sont définies dans un AnimationTree, ici de type *2D Freeform Cartesian* (cf. [figure 3.2](#)), grâce à deux paramètres. Dans ce cas-ci, les deux paramètres sont *moveX* et *moveY* et ils sont de type float. Ils sont définis dans le script qui est chargé du déplacement du *Player* vu au-dessus.

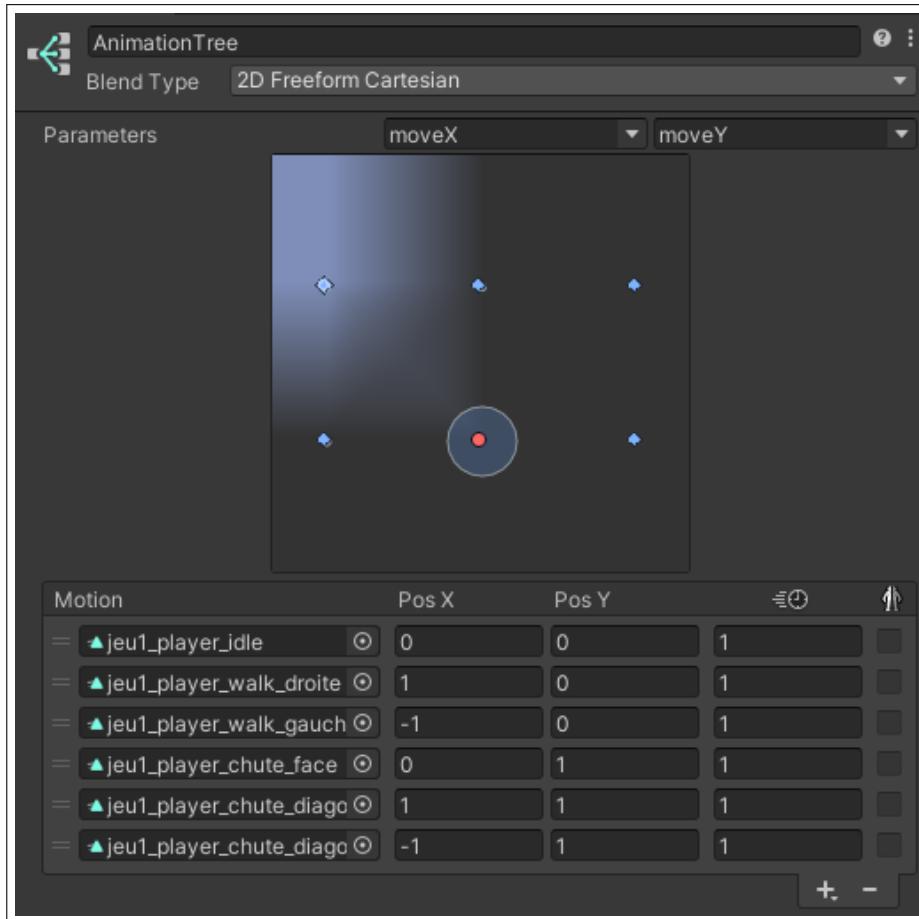


FIGURE 3.2 – Animator Controller du Player (Animation Tree)

Suivant les valeurs de *moveX* et *moveY*, une animation différente est lancée. Ici *moveY* correspond à 1 si le Player est en chute libre ou en train de sauter, et 0 s'il est en contact avec une plateforme. La valeur *moveX* indique la direction dans laquelle le *Player* se déplace.

3.2.3 Vie du Player

Le script *Player.cs* attaché au *GameObject* possède deux méthodes : *PrendreDegats(uint degat)* et *RegenerationVie(uint valeur = 1)*, qui permettent de mettre à jour son attribut *Vie*. Cette valeur est initialisée à 10 au début de partie. En plus de mettre à jour cet attribut, ces méthodes appellent une fonction pour mettre à jour la barre de vie de l'interface de jeu, en fonction de la nouvelle valeur de vie (cf. [figure 3.19](#)).

La vie du joueur est régénérée toutes les deux secondes grâce à un autre *GameObject* nommé *TimeEvent*, qui possède un script chargé d'appeler la méthode *RegenerationVie()* à intervalles de temps réguliers.

L'objet *Zone bas écran* (cf. [figure 3.12](#)) inflige le maximum de dégâts au *Player*, lorsqu'il entre en contact avec. Ce contact indique que le *Player* est sorti de l'écran, déclenchant ainsi la fin du jeu.

Dans la méthode *Update()* qui est appelée à chaque actualisation, l'attribut *Vie* du *Player* est vérifié pour s'assurer qu'il est toujours strictement positif, sinon la fin du jeu est lancée et le retour au menu principal de sélection des mini-jeux (cf. [figure 2.7](#)) est déclenché.

3.3 Plateformes

Le jeu 100 FLOOR comporte différents types de plateformes. Tous les GameObjects des plateformes sont composés de ces différents composants :

- ☒ SpriteRenderer : pour l'affichage de la plateforme
- ☒ BoxCollider2D : pour rendre solide la plateforme et détecter les collisions avec le *Player*
- ☒ EdgeCollider2D : pour la génération de la carte (propriété Trigger activée)
- ☒ Rigidbody2D : pour gérer les interactions physiques avec le *Player*
- ☒ ExtrimitBoxCollider2D.cs : script qui calcule les coordonnées des extrémités du BoxCollider2D
- ☒ MovePlateforme.cs : script qui sert à déplacer la plateforme vers le haut de l'écran
- ☒ AudioSource : contient le son à jouer lorsque le *Player* tombe sur la plateforme
- ☒ AudioPlateforme.cs : script pour déclencher la lecture du son à l'impact du *Player*

3.3.1 Comportement commun

Toutes les plateformes utilisent le script *MovePlateforme.cs* qui s'occupe de gérer leur déplacement vers le haut de l'écran. Le composant EdgeCollider2D, qui a la propriété Trigger activée, permet au script de détecter lorsque le *Player* passe en dessous de la plateforme. En effet, le composant EdgeCollider2D associé à chaque plateforme correspond à une ligne horizontale sur toute la largeur de l'écran située légèrement en dessous du haut du BoxCollider2D de la plateforme (cf. figure 3.3). Lorsque le GameObject *Player* traverse cette ligne, cela déclenche un événement et la méthode associée dans le script, *OnTriggerEnter2D()* est appelée. Dans cette méthode, le BoxCollider2D, qui est chargé des collisions physiques avec le *Player* et de lui appliquer les effets de la plateforme, est désactivé. Cela permet de rendre la plateforme traversable lorsque le joueur est passé en dessous et de ne pas le bloquer lorsqu'il rebondit sur une plateforme de saut.

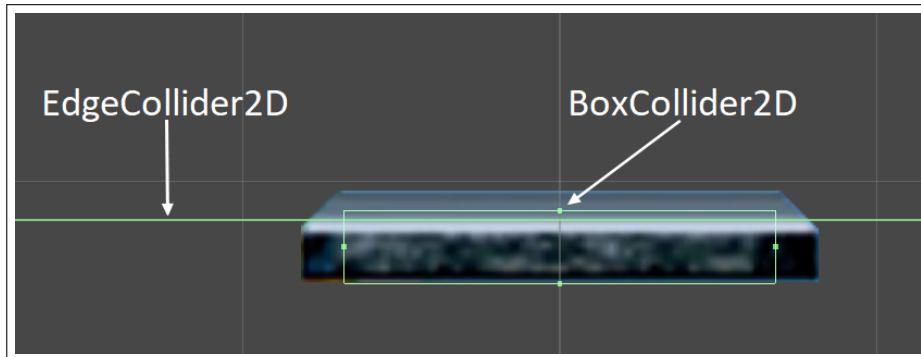


FIGURE 3.3 – Plateforme simple : EdgeCollider2D et BoxCollider2D

Le script *ExtrimitBoxCollider2D.cs* permet d'obtenir les coordonnées exactes du haut, bas, gauche et droite du BoxCollider2D en tenant compte de sa position et de ses dimensions. Ces coordonnées exactes sont utiles pour savoir si le *Player* est en dessous d'une plateforme par exemple. Il ne faut pas seulement comparer les coordonnées des GameObjects entre elles, puisqu'elles indiquent leur centre (cf. figure 3.14). Dans ce cas-ci, il faut comparer les coordonnées des pieds de l'objet *Player* (ie l'extrémité basse de son BoxCollider2D) et celles du haut de la plateforme (ie celle de l'extrémité haute de son BoxCollider2D).

Ce script est également utile pour la génération des plateformes, où il est nécessaire de connaître les extrémités des plateformes, comme nous le verrons plus loin (cf. section 3.4).

Les plateformes utilisent également le script *AudioPlateforme.cs* qui est chargé de détecter la collision avec le *Player* lorsqu'il tombe dessus et de jouer le son associé à l'impact (entre le *Player* et la plateforme spécifique) grâce au composant AudioSource.

3.3.2 Plateforme simple

La plateforme simple (cf. figure 3.3) ne possède aucun autre composant que ceux décrits plus haut. Elle n'a pas d'animation et ne possède pas d'effet particulier quand le *Player* tombe dessus.

3.3.3 Plateforme tapis roulant

La plateforme tapis roulant a pour effet de déplacer le *Player* horizontalement sur la gauche ou sur la droite en continu lorsque celui-ci est dessus.

Pour ce faire, le sens de l'effet de déplacement est défini de manière aléatoire à linstanciation du GameObject. Si le sens de l'effet de déplacement est choisi pour aller vers la gauche, alors un flip sur laxe X est appliqu sur le SpriteRenderer, qui affiche par défaut le *sprite* du tapis roulant avec les flches vers la droite (cf. figure 3.4a), pour effectuer un effet miroir sur le *sprite* (cf. figure 3.4b).

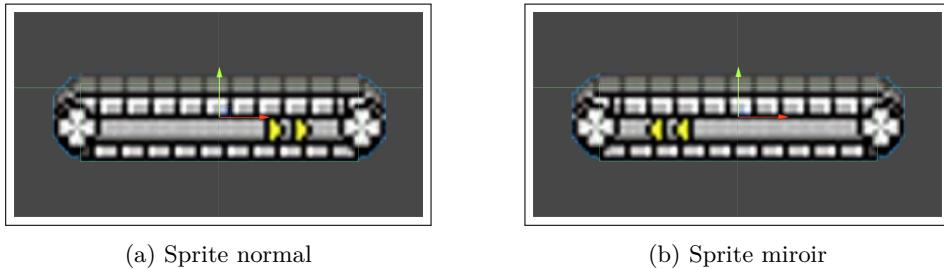


FIGURE 3.4 – Sprite de la plateforme tapis roulant

Le vecteur de force à appliquer sur le Rigidbody2D du *Player* est calculé suivant la vitesse du tapis roulant et son sens. Il ne possède qu'une composante non nulle, sa coordonnée X.

L'effet de la plateforme est appliqué tant que le *Player* reste en contact avec la plateforme. Ainsi, tant que les deux Colliders (celui de la plateforme et du *Player*) restent en contact, la méthode *OnCollisionStay2D()* du script *PlateformeTapisRoulant.cs*, est appelée. Cette méthode applique la force de déplacement calculée sur le *Player*.

3.3.4 Plateforme tourante

La plateforme tournante a pour effet de ralentir le *Player* dans sa chute. Lorsque celui-ci atterrit dessus, il est stoppé, puis après quelques millisecondes, la plateforme devient traversable et le *Player* reprend sa chute.

Pour ce faire, deux composants sont rajoutés : un Animator qui contient lanimation à jouer et le script *PlateformeTournante.cs* qui lance lanimation de la plateforme lorsque le collider du *Player* entre en contact avec. Le Player est stoppé dans sa chute par le BoxCollider2D de la plateforme puis lanimation de la plateforme est jouée.



FIGURE 3.5 – Animation clip de la plateforme tournante

Dans le clip d'animation, le *sprite* de la plateforme est changé (cf. figure 3.6). Un événement d'animation est déclenché (cf. figure 3.5). Celui-ci appelle la méthode (dans *MovePlateforme.cs*) qui permet de désactiver le BoxCollider2D pour permettre à l'objet *Player* de traverser la plateforme et de retomber.

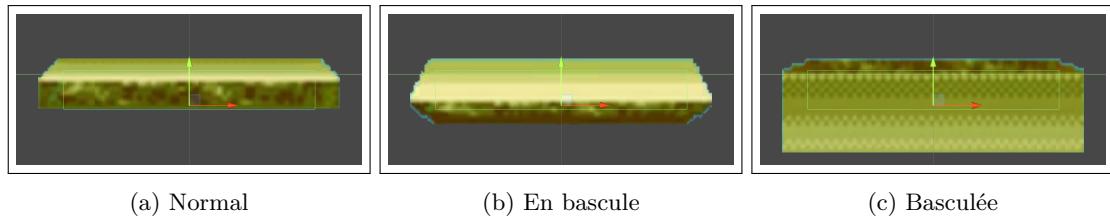


FIGURE 3.6 – Sprite de la plateforme tournante

3.3.5 Plateforme avec des pics

La plateforme avec des pics a pour effet d'infliger des dégâts au *Player*, lorsqu'il atterrit dessus.

Pour ce faire, le script *PlateformePics.cs* a été rajouté. Dans ce script, lorsque la méthode *OnCollisionEnter2D()* est appelée la vie du joueur est diminuée par l'intermédiaire de la méthode *PrendreDegat()*.



FIGURE 3.7 – Sprite de la plateforme avec des pics

Le [sprite](#) de la plateforme possède une animation simple qui est gérée par composant un Animator. L'animation se contente d'alterner les [sprites](#) rendus (cf. [figure 3.7](#)) toutes les cinquante millisecondes.

3.3.6 Plateforme de saut

La plateforme de saut a pour effet de faire rebondir le *Player* lorsqu'il tombe dessus. Comme pour les autres plateformes avec des effets, un Animator a été ajouté, ainsi qu'un script *PlateformeSaut.cs* servant à gérer son effet.

Le script *PlateformeSaut.cs* possède une variable booléenne *Saut* servant à surveiller l'état du *Player* (pour n'appliquer qu'une seule fois son effet).

Il possède également une méthode *AddForceSautPlayer()* qui donne une impulsion vers le haut sur le Rigidbody2D du *Player* pour simuler un rebond. Cette méthode calcule la force relative au déplacement du *Player* par rapport à celle de la plateforme, afin que l'impulsion appliquée soit constante.

C'est le script qui déclenche l'animation lorsque que le *Player* tombe sur la plateforme et qu'il n'est pas déjà en train de sauter (ie variable *Saut* égale à false).

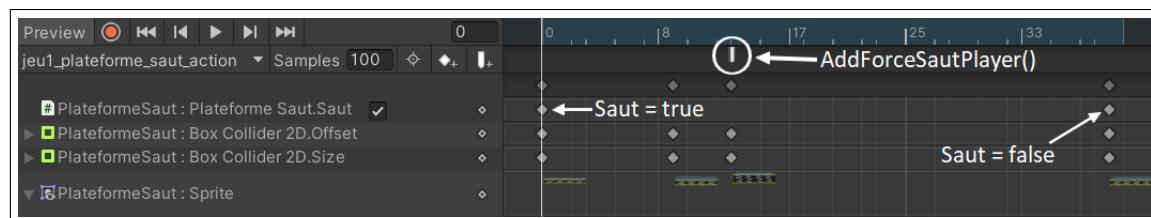


FIGURE 3.8 – Animation clip de la plateforme de saut

L'animation (cf. [figure 3.8](#)) change l'état de la variable *Saut* à true au début de sa lecture pour indiquer que l'animation est en cours et que le joueur est déjà en train de sauter. Cette variable est remise à false à la fin de l'animation et permet donc au script de relancer l'animation pour réappliquer l'effet de rebond. L'effet à appliquer sur le *Player* est déclenché dans l'animation avec un *animation event* qui appelle la méthode *AddForceSautPlayer()*.

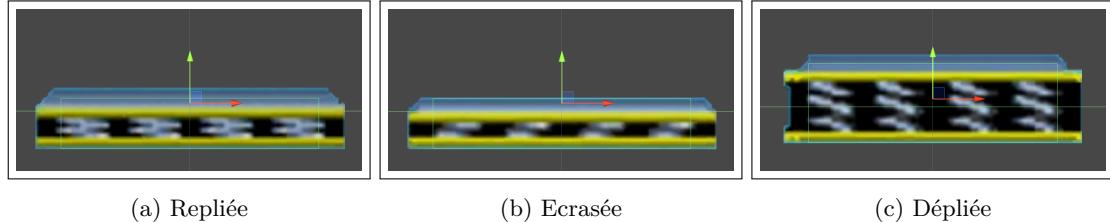


FIGURE 3.9 – Sprite de la plateforme de saut

L'animation change également le [sprite](#) de la plateforme. Le *BoxCollider2D* est adapté suivant le [sprite](#) défini pour suivre la hauteur de l'image : augmentation de la hauteur du collider lorsque le [sprite](#) de la plateforme dépliée est choisi (cf. [figure 3.9c](#)).

3.4 Gestion de la carte

La carte du jeu est composée de plusieurs GameObjects. L'objet *ContourCarte* contient deux objets ayant des BoxCollider2D et des Rigidbody servant à définir les limites de jeu sur les côtés de l'écran et empêcher le joueur de sortir de l'écran sur les cotés (cf. [figure 3.12](#)). Le GameObject *Autres* sert à regrouper des objets nécessaires à la bonne gestion des plateformes que nous verrons un peu plus bas. Ils contiennent également tous un BoxCollider2D et un Rigidbody pour détecter des événements. L'objet *Générateur Plateformes* sert à gérer toutes les méthodes d'instanciation et d'activation des plateformes.

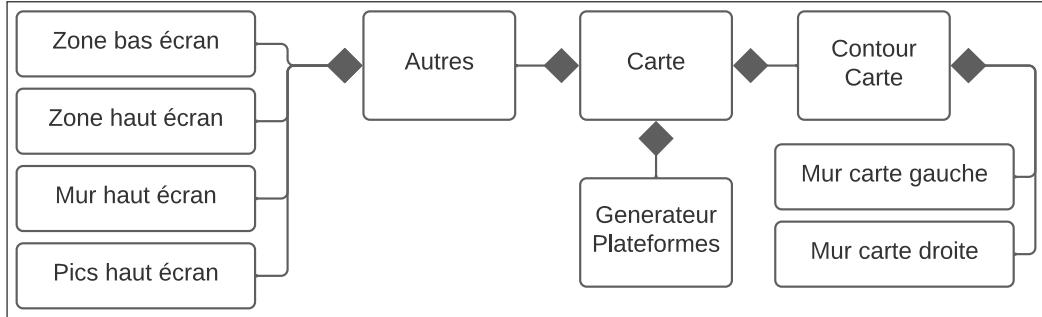


FIGURE 3.10 – Diagramme du GameObject Carte

3.4.1 Instanciation des plateformes

Pour gérer efficacement les objets en mémoire, un [pool](#) de plateformes est mis en place. Au début de la partie, l'ensemble des plateformes sont instanciées et désactivées. Cela permet de réduire les temps de chargement pendant le jeu en évitant de constamment instancier et détruire des objets. De plus, cela permet de limiter l'utilisation de la mémoire vive en évitant trop d'objets actifs simultanément.

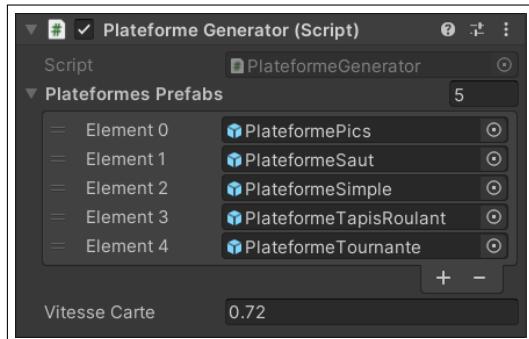


FIGURE 3.11 – Script Plateforme Générateur

C'est l'objet générateur de plateformes qui s'occupe de cette gestion. Le script associé à cet objet possède un tableau contenant les cinq types de plateformes sous forme de prefab (cf. [figure 3.11](#)). C'est à partir de ces prefabs que les [pools](#) de plateformes sont instanciés.

Pour gérer ses [pools](#) de plateformes, le script utilise un tableau de GameObjects à deux dimensions. Chaque ligne contient l'ensemble des plateformes d'un certain type.

Pour choisir une plateforme inutilisée, une méthode *ChoisirPlateformeInPools()* renvoie directement une plateforme inactive dans les différents [pools](#).

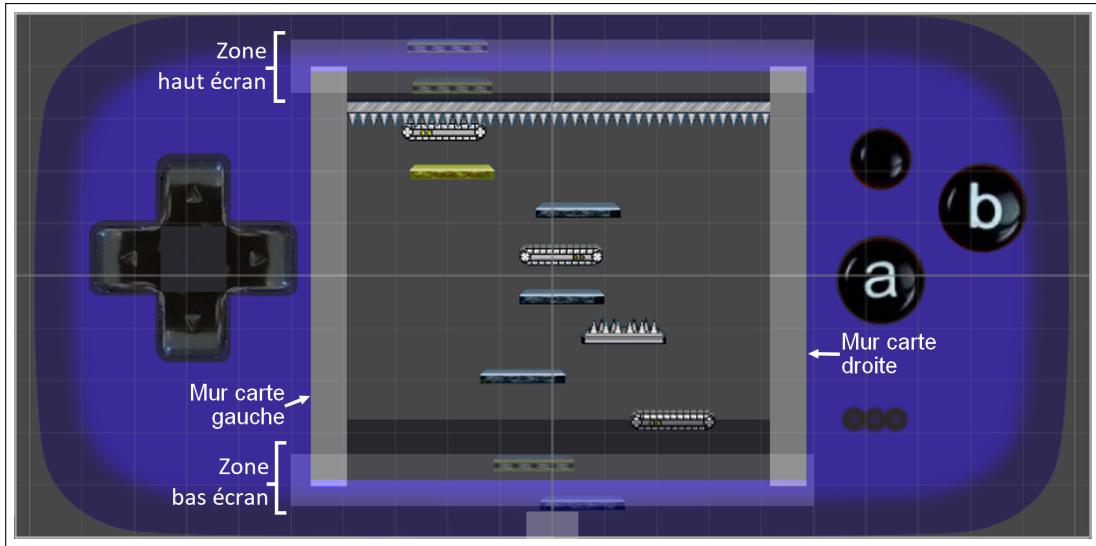


FIGURE 3.12 – Disposition des plateformes dans la scène

Pour les dimensionner correctement ces *pools*, il faut compter le nombre de plateformes nécessaires utilisées en même temps sur une scène lorsque le jeu est lancé et génère les plateformes en continu. Il y environ 8 plateformes visibles à l'écran et 2 plateformes en haut et en bas de la scène non visible, ce qui donne 12 plateformes actives sur la scène. La taille des *pools* de plateformes choisies est donc de 5 par type, ce qui donne 25 plateformes sur l'ensemble des *pools*, donnant une marge de plateformes disponibles largement suffisante.

3.4.2 Génération des plateformes

Pour gérer l'activation des plateformes en continu, l'objet *Zone haut écran* possède un script nommé *RecyclagePlateforme.cs* qui s'occupe de désactiver les plateformes lorsqu'elles entrent en contact avec lui. En fois la plateforme désactivée, il appelle la méthode du *Générateur Plateformes* pour générer de nouvelles plateformes. C'est le composant *EdgeCollider2D* qui déclenche ce contact avec l'objet *Zone haut écran*, étant donné que le *BoxCollider2D* est désactivé lorsque la plateforme passe au-dessus du *Player*.

Lorsque la plateforme est désactivée, elle disparaît visuellement de la scène et n'interagit plus avec les autres objets. La génération d'une nouvelle plateforme consiste en réalité à activer une plateforme qui n'est pas utilisée dans les *pools* de plateformes. Le processus de génération d'une nouvelle plateforme sur la carte passe par plusieurs étapes.

- ❖ Premièrement : une plateforme disponible est récupérée via la méthode *ChoisirPlateformeIn-Pools()*.

Le type de plateforme est défini avec un tirage aléatoire. Ensuite, une plateforme qui n'est pas activée est sélectionnée dans le *pool* de plateformes du type choisi, c'est-à-dire dans une ligne du tableau de *GameObjects*. Si aucune plateforme de ce type n'est disponible, alors le tirage aléatoire est relancé parmi les types de restants. Ce processus est effectué tant que tous les *pools* de plateformes n'ont pas été vérifiés. Comme le nombre total de plateformes est strictement supérieur au nombre de plateformes qui doivent être actives en même temps sur la scène (25 au total pour 12 actives), il y en a toujours une de libre.

- ❖ Deuxièmement : la position d'activation de la plateforme est calculée.

Sur l'axe de Y, la position est calculée par rapport à la dernière plateforme activée. La distance entre deux plateformes est définie par la hauteur du *BoxCollider2D* du *Player*. De plus, cette distance doit être celle entre le bas du *BoxCollider2D* de la dernière plateforme activée et le haut

du BoxCollider2D de la plateforme que l'on veut activer (cf. [figure 3.15b](#)). En effet la position d'un GameObject correspond au centre de celui-ci en général (cf. [figure 3.14](#)) et ne coïncide pas avec les extrémités du collider, c'est pourquoi il faut prendre en compte leur hauteur dans notre calcul. Dans un premier temps, la position calculée sans prendre en compte la hauteur du collider de la plateforme que l'on veut activer (cf. [figure 3.15a](#)). Cette prise en compte sera effectuée avant l'activation avec un décalage de la plateforme.

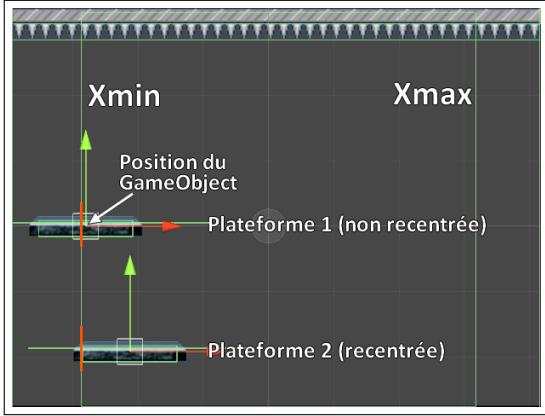


FIGURE 3.13 – Positionnement sur l'axe des X

Sur l'axe X, la position des plateformes est déterminée par un tirage aléatoire entre les deux coordonnées X_{min} et X_{max} , qui représentent les limites de l'espace où les plateformes doivent apparaître. Cependant, cette position ne prend pas en compte la largeur des colliders des plateformes mais cette position sera ajusté par la suite avant l'activation.

Par exemple, sur la [figure 3.13](#), la plateforme 1 est positionnée à droite de la ligne définissant X_{min} , mais elle dépasse sur la gauche au-delà de cette limite en raison de sa largeur.

❖ Troisièmement : le déplacement de la plateforme à la position calculée avec un réajustement.

La plateforme est déplacée à la position calculée, cependant il faut réajuster sa position sur X et Y pour bien prendre en compte les dimensions de son collider. Pour réajuster la position sur Y, la méthode de décalage vers le bas du script *ExtremiteBoxCollider2D.cs*, que toutes les plateformes possèdent, permet de corriger sur Y la position, en déplaçant l'objet vers le bas pour que l'extrémité du BoxCollider2D (cf. [figure 3.14](#)) soit à la position du GameObject (croix rouge sur [figure 3.14](#)). De cette façon, la distance entre les deux extrémités des plateformes est bien celle voulue (cf. [figure 3.15b](#)).

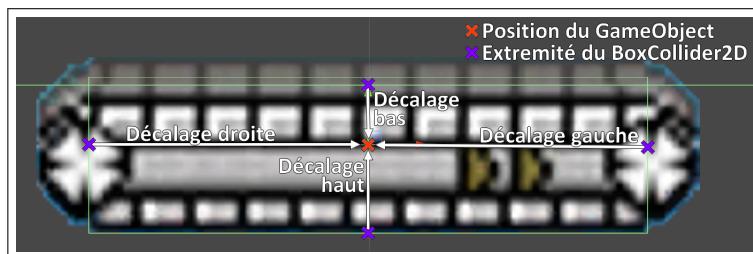


FIGURE 3.14 – Extrémités du composant BoxCollider2D d'une plateforme

Il faut également déplacer la plateforme sur l'axe X. La plateforme est donc décalée sur le côté gauche ou droit suivant si elle dépasse en X_{min} ou X_{max} (de la même manière que pour le décalage vers le bas avec les méthodes du script *ExtremiteBoxCollider2D.cs*). Une fois le décalage effectué, l'entièreté du collider de la plateforme se trouve entre X_{min} et X_{max} (cf. [figure 3.13](#)).

- ❖ Quatrièmement : la plateforme est activée.

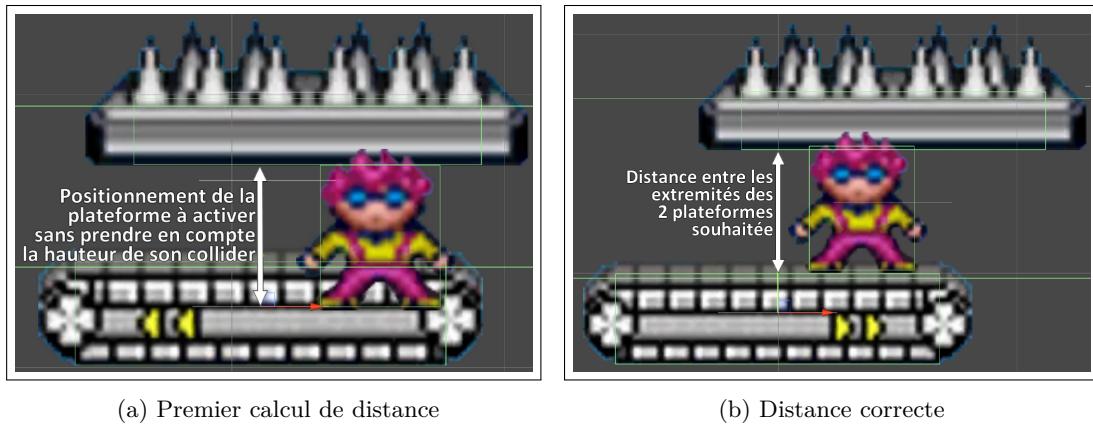


FIGURE 3.15 – Distance entre 2 plateformes

Une fois la mécanique de génération en continu de plateforme définie, il faut générer les premières plateformes au lancement du jeu.

3.4.3 Activation des premières plateformes

Pour générer les plateformes sur l'ensemble de l'écran au début de la partie, il faut calculer le nombre de plateformes nécessaires pour occuper toute la hauteur de l'écran, de la *Zone haut écran* à la *Zone bas écran* (cf. [figure 3.12](#)). Ce calcul est effectué au démarrage du jeu, puis la première plateforme est générée à la position de l'objet *Zone haute écran*. Ensuite, les autres plateformes sont ensuite générées de manière classique, en fonction de la dernière plateforme générée, comme décrit dans le paragraphe précédent. De cette manière, un nombre suffisant de plateformes est activé au début de la partie, ce qui permet de lancer la génération de plateformes en continu.

Afin d'éviter que le joueur commence sur une plateforme ayant un effet et qu'il n'ait pas le temps de réagir, la plateforme sur laquelle il débute, est remplacée par une plateforme simple. Ensuite, le joueur est déplacé sur cette plateforme et la partie commence.

Dans le jeu original, le joueur ne commence pas nécessairement sur une plateforme centrée à l'écran. Afin de faciliter le début de partie, la plateforme de départ est recentrée dans l'écran.

3.5 Interface du jeu

L'interface du jeu est composée de plusieurs GameObjects.

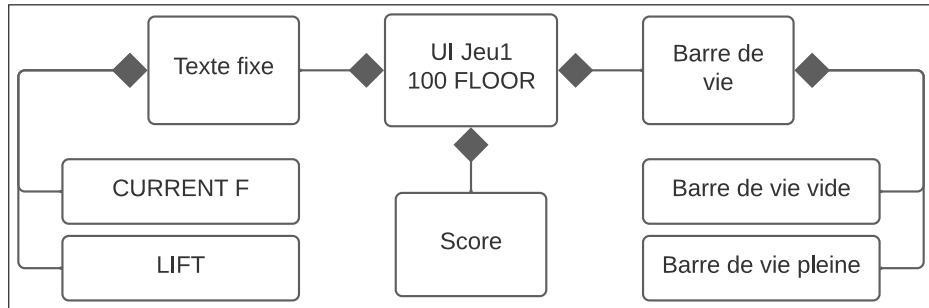


FIGURE 3.16 – Diagramme du GameObject de l'interface du jeu 100 FLOOR

Les objets *CURRENT F* et *LIFT* contiennent simplement un composant TextMeshPro, qui permet d'afficher un élément textuel à l'écran. Ces deux éléments restent constants durant toutes la partie.



FIGURE 3.17 – Interface de jeu 100 FLOOR complète

3.5.1 Barre de vie

Les deux objets *Barre de vie vide* et *Barre de vie pleine* contiennent un composant Image. Contrairement au composant RawImage, celui-ci permet des manipulations plus avancées.

Les deux images associées à ces objets sont les suivantes :



FIGURE 3.18 – Sprite de barre de vie

La barre de vie du joueur est représentée par ces deux objets, l'un contenant la barre de vie totalement vide et l'autre la barre de vie totalement pleine. Les deux objets sont superposés, celui affichant la barre pleine étant placé au-dessus de celui affichant la vide. Un script, attaché à l'objet *Barre de vie*, possède une méthode pour mettre à jour l'affichage en fonction de la valeur de la vie du joueur. Cette mise à jour est effectuée chaque fois que la vie du joueur change de valeur.

Lorsque la valeur envoyée correspond au maximum de vie du joueur, l'image de la barre de vie pleine est affichée à 100%. Mais lorsque la valeur envoyée est 0, l'image n'est plus affichée du tout. Les valeurs intermédiaires font varier le pourcentage d'affichage de l'image horizontalement. Ainsi, pour 50% d'affichage, seule la moitié gauche de l'image est affichée (cf. [figure 3.19](#)).



FIGURE 3.19 – Barre de vie : moitiée pleine

De cette façon, les petits rectangles jaunes de la barre de vie se remplissent ou se vident en fonction de la valeur de la vie du joueur passée en paramètre.

3.5.2 Score

L'objet *Score* est composé d'un composant TextMeshPro qui affiche le score du joueur. Ce score est modifié par le script attaché à cet objet. Ce script est appelé par le GameObject nommé *TimeEvent* qui est chargé d'incrémenter le score toutes les huit secondes.

3.6 Arrière-plan

Dans le jeu original sur console, l'arrière-plan est composé de plusieurs types de nuages qui défilent à différentes vitesses. Pour reproduire cet effet, l'arrière-plan est divisé en quatre couches qui sont affichées dans l'ordre suivant :

1. Fond bleu
2. Nuages
3. Maisons
4. Gros nuages
5. Décors

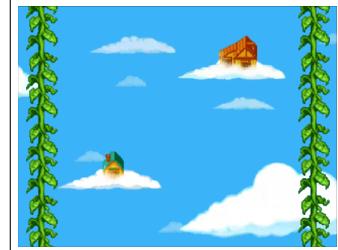


FIGURE 3.20 – Arrière-plan

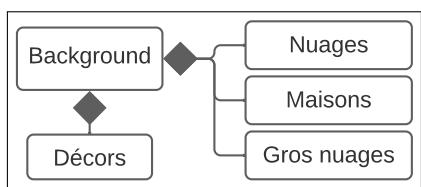


FIGURE 3.21 – Diagramme du GameObject Background

L'arrière-plan est organisé dans un GameObject *Background* (cf. figure 3.21). La première couche d'arrière-plan est directement dans le SpriteRenderer de ce GameObject puisqu'il s'agit d'une simple image bleue immobile. Le GameObject *Décors* est composé lui aussi d'un SpriteRenderer qui affiche les lianes, qui sont également immobiles, sur les côtés de l'écran.

Les trois couches suivantes sont disposées dans trois objets différents avec leur propre Sprite-Renderer et Animator. Ces trois couches ont le même comportement, à savoir un défilement vers le haut en continu mais avec des vitesses légèrement différentes. Ce défilement vers le haut s'effectue par l'intermédiaire d'une animation qui modifie la position des images. Les images de ces trois couches sont composées de deux fois le même motif l'un au-dessus de l'autre. Cela permet de garder une fluidité dans l'animation.

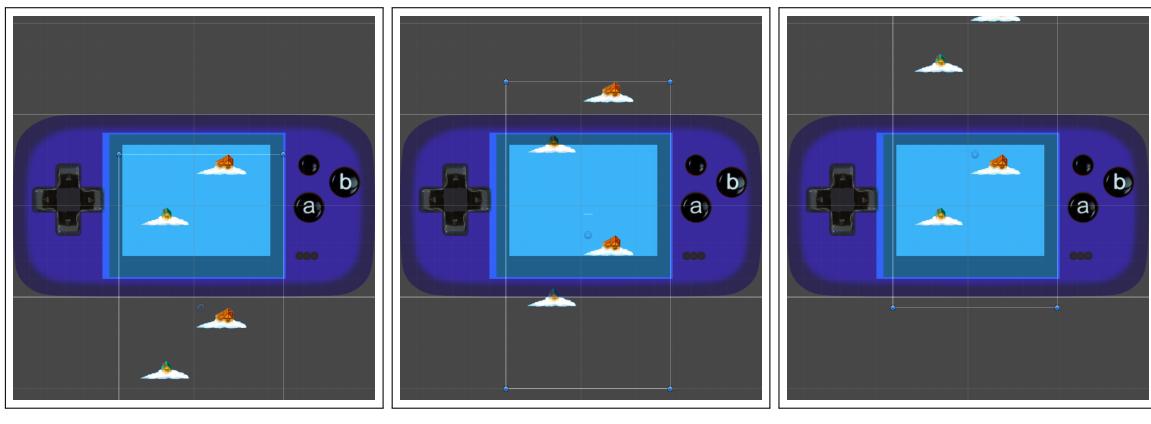


FIGURE 3.22 – Animation de la couche Maisons

En effet, l'animation fait progresser l'image vers le haut de l'écran (cf. figure 3.22b) jusqu'à ce que le début du deuxième motif de l'image arrive en haut de l'écran (cf. figure 3.22c). Quand cela arrive, l'animation redescend toute l'image (cf. figure 3.22a), de manière à ce que le premier motif vienne se placer exactement là où le deuxième motif s'était arrêté. Les deux motifs étant parfaitement identiques, le joueur ne perçoit pas le changement. L'animation peut alors reprendre la progression de toute l'image vers le haut de l'écran. Avec cette méthode, l'image ne sort jamais complètement de l'écran et cela permet de faire défiler indéfiniment l'image sans que le joueur s'en aperçoive.

3.7 Résultats

Le jeu 100 FLOOR est complet en termes de fonctionnalités de gameplay et peut être joué sur téléphone. Cependant, il reste encore quelques améliorations à apporter.



FIGURE 3.23 – Jeu 100 FLOOR complet dans l’application

Tout d’abord, il faudrait une meilleure intégration des sons dans le jeu. En effet, il y a une différence de volume entre certains clips audio qui sont joués dans le jeu. De plus, la musique du jeu n’a pas encore été ajoutée. D’un point de vue visuel, il faudrait ajouter une animation de clignotement du *sprite* du *Player* lorsqu’il prend des dégâts.

Il subsiste encore quelques problèmes de collisions mineurs sur certaines plateformes. Il peut arriver que le joueur traverse la plateforme de saut quand il atterrit sur ses extrémités gauche et droite. De plus, le joueur peut rester collé au mur de la carte sur les côtés de l’écran s’il maintient les boutons pour se déplacer leur direction.

A la différence de la console PDC, dans la version actuelle du jeu, lorsque le joueur meurt, la partie se termine par un retour au menu principal et non sur un écran de score. De plus, contrairement à la version console, le début de la partie est immédiat et les plateformes commencent à bouger dès que la partie commence, sans laisser de temps de préparation au joueur.

4 Deuxième jeu : B.P.

4.1 Gameplay

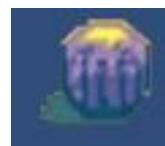
Le deuxième jeu que nous avons tenté de reproduire est nommé BP. Il s'agit d'un jeu similaire au très connu Bomberman : le joueur dirige un personnage dans un labyrinthe peuplé d'obstacles, et peut poser des bombes pour casser certains obstacles en prenant garde à ne pas se trouver dans la zone d'explosion. L'objectif de ce jeu est d'utiliser les bombes pour éliminer les différents ennemis qui peuplent le niveau, sans que ces derniers ne nous touchent en retour. Le joueur dispose de trois essais pour finir le niveau.

4.2 Agencement d'un niveau

Chaque niveau du jeu est représenté sous forme d'un labyrinthe d'obstacle agencé selon un paterne de grille. Il existe deux types d'obstacles, facilement discernables par leur apparence : les obstacles que le joueur peut détruire avec des bombes, et les obstacles indestructibles.



(a) Obstacle indestructible



(b) Obstacle destructible

FIGURE 4.1 – Obstacles du niveau BP

Dans notre implémentation du jeu, l'agencement du niveau est contenu dans un GameObject de type « Tilemap », accompagné de son GameObject parent « Grid ». Ce composant ne fait pas partie des éléments fournis par défaut par Unity, et doit être importé via le package « 2D Tilemap Editor ».

La Tilemap est un GameObject capable de stocker l'emplacement d'un autre type d'élément, les « Tiles » ou « Tile Assets » selon une disposition en grille. Le tout fonctionne comme une grille sur laquelle on peut placer nos différentes tuiles afin de rapidement créer un niveau fonctionnel.

Le composant Grid (attaché au GameObject du même nom) permet de définir l'agencement général de la grille. On peut ainsi régler le type de la grille (rectangulaire, hexagonale, ou isométrique), ainsi que la taille des cases de la grille. Afin de réaliser le jeu BP, nous avons utilisé une grille aux cases rectangulaires dont la dimension en hauteur est plus faible que la largeur. Ainsi, comme nous utilisons des tuiles carrées pour le remplissage de la grille, on obtient un léger effet de profondeur similaire au jeu original.



FIGURE 4.2 – Effet de profondeur

Le GameObject Tilemap est lié à plusieurs composants :

- Le composant « Tilemap » qui définit le rôle du GameObject.
- Le composant « Tilemap Renderer » qui permet de régler divers paramètres sur l'affichage des tuiles à l'écran.
- Le composant « Tilemap Collider 2D » qui permet d'inclure les tuiles de la grille dans les calculs des interactions physiques.

Les Tilemap Assets servent à définir les propriétés basiques des tuiles de la grille, à savoir le [sprite](#) associé et l'emplacement de la boîte de collision de la tuile. Dans notre cas, la boîte de collision est dissociée du sprite de la tuile : la boîte fait les dimensions d'une case de la grille, alors que la partie supérieure du sprite dépasse de la case pour générer l'effet de profondeur abordé précédemment. Ces objets sont stockés sous la forme de Prefab dans l'arborescence du projet.

4.3 Personnages et ennemis

Le personnage joueur et les différents ennemis à éliminer sont fabriqués à partir d'une base commune de 3 composants (déjà abordée précédemment) :

- Le « Sprite Renderer » qui sert pour l'affichage du GameObject à l'écran.
- Le « Collider 2D » qui définit la forme de l'objet lors des interactions physiques en 2D.
- Le « Rigidbody 2D » qui définit les propriétés physiques de l'objet lors des interactions physiques en 2D.
- « L'Animator » qui permet d'animer le sprite du GameObjet.

Le personnage et les ennemis disposent de plus d'un composant script qui définit leur comportement dans le cadre du jeu.

4.3.1 Le personnage joueur

Le personnage du jeu est limité à deux actions : se déplacer, et poser une bombe sur la case actuellement occupée. Le déplacement est géré à chaque boucle de jeu en regardant l'état des boutons de la croix directionnelle. Si on détecte un appui sur l'une des quatre directions de la croix, alors le personnage essaie de se déplacer dans cette direction jusqu'à rencontrer un obstacle ou un ennemi.

Pour la gestion des collisions, le Collider 2D du joueur est aux dimensions d'une case de la grille du niveau comme décrit plus haut. Nous avons choisi d'utiliser un collider en forme de capsule (rectangle avec bords arrondis) afin de pallier à un problème lors du calcul des collisions qui empêchait parfois le joueur de passer entre deux obstacles. De plus, le [sprite](#) du personnage mesure environ deux fois la taille d'une case de la grille, afin de simuler l'effet de vue du dessus.



FIGURE 4.3 – Personnage joueur (jeu original)

Dans le jeu d'origine, le joueur peut poser des bombes en appuyant sur le bouton A. Il dispose initialement d'une seule bombe, mais peut en obtenir d'autres via des bonus dans les obstacles qu'il peut détruire. Pour gérer la pose de bombe, un événement est émis à chaque appui du bouton A, afin d'être reçu et exploité par le GameObject du joueur. À la réception de cet événement, le script tente de créer une bombe dans la case sur laquelle se tient le joueur (il peut se trouver à cheval entre deux cases, auquel cas la case la plus occupée est choisie). Cette bombe est gérée par un GameObject bombe créé à la volée selon un modèle de préfab, comme décrit dans une section suivante.

Lors de la pose d'une bombe, celle-ci ne génère pas encore de collision avec le joueur, effet qui perdure tant que les boîtes de collision de la bombe et du joueur se superposent. Dès que cette superposition s'arrête, la bombe se met à se comporter comme un obstacle pour le joueur.



FIGURE 4.4 – Superposition lors de la pose de la bombe (jeu original)

4.3.2 Ennemis

Les ennemis apparaissent à des endroits prédéfinis au lancement du niveau. Ils se déplacent aléatoirement sur la carte sans pouvoir retirer les obstacles. Ils réalisent occasionnellement une attaque dans une direction ; celle-ci fait perdre le joueur en cas de contact, mais ne compte pas comme faisant partie de l'ennemi lors du calcul des éléments touchés par l'explosion d'une bombe.

Le déplacement des ennemis est relativement simple : Ils se déplacent de case en case de façon sensiblement aléatoire. Ils attaquent aussi à intervalle régulier, avec quelques variations de timing.

4.4 Bombe

Les bombes sont l'outil utilisé par le joueur afin d'avancer dans le jeu et de remporter la partie. Elles lui permettent de détruire certains obstacles et d'éliminer les ennemis. Le joueur débute avec une seule bombe, et la récupère lorsque la bombe explose au bout d'un certain temps. La bombe a initialement une portée de 1 case (elle ne peut toucher que les entités des cases adjacentes à la sienne), mais cette portée peut être améliorée via des bonus récupérés dans certains obstacles détruits. Le nombre de bombe du joueur peut être augmenté de la même manière.

Les bombes sont représentées par un GameObject « Bombe ». Lorsque le personnage pose une bombe, une instance de l'objet est créée aux coordonnées correspondantes et un compte à rebours démarre. Une fois le décompte écoulé, la bombe explose et touche la première entité à sa portée dans les quatre directions de la grille. Le test est effectué en récupérant les coordonnées de chaque case sur la trajectoire de l'explosion, puis en vérifiant si une entité (obstacle, joueur ou ennemi) se trouve sur la case concernée, et ce jusqu'à trouver une entité ou à arriver à la portée maximale. L'opération est répétée pour chaque direction.

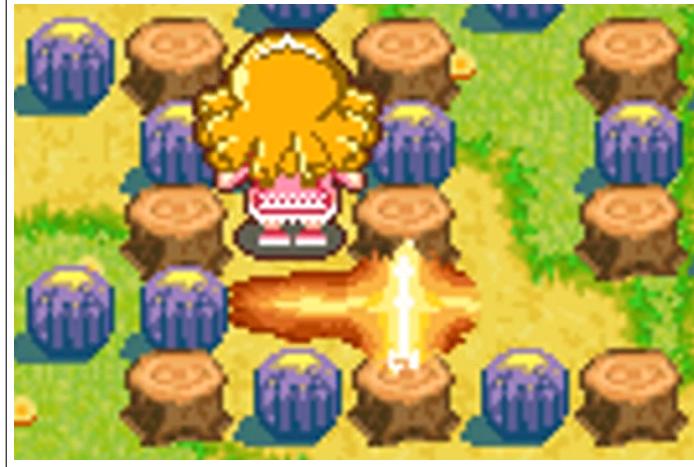


FIGURE 4.5 – Explosion d'une bombe de portée 1 case (jeu original)

Si l'explosion trouve un obstacle, alors on vérifie s'il s'agit d'un obstacle destructible. Le cas échéant, on retire la tuile de l'obstacle de la Tilemap et on a une chance aléatoire de générer un bonus à ramasser à cet endroit.

Si l'explosion touche un ennemi, alors celui-ci joue une animation de mort à l'issue de laquelle le GameObject est détruit. De même, si le personnage joueur est touché, alors il joue une animation de mort avant de perdre une vie. Le niveau est également réinitialisé.

4.5 Résultats

Le jeu BP est encore incomplet. Il manque encore certains éléments de gameplay et de nombreux éléments graphiques. Des complications lors du regroupement du travail ont retardé le développement de certaines fonctionnalités du jeu.

En effet, les différents [sprites](#) n'ont pas encore été animés par manque de temps. Cela inclut les animations de la bombe et de son explosion, ainsi que les attaques des ennemis. L'[UI](#) du jeu, servant notamment à suivre le décompte des bombes possédées par le personnage ainsi que son score dans le niveau, est également manquante. Les sons du jeu ne sont pas non plus présents.

La fin du niveau n'est pour l'instant pas gérée. Le calcul du score n'est pas non plus implémenté. Dans le jeu originel, le joueur pouvait par exemple gagner des points en détruisant les obstacles, en éliminant les ennemis et en ramassant divers bonus.

Conclusion

Bien que la [PDC](#) propose une grande variété de mini-jeux, il est souvent difficile de retrouver une expérience similaire en termes de graphismes et de gameplay pour ses différents jeux.

Cependant, son format compact et la nature des jeux s'adaptent parfaitement à une reproduction sur smartphone. L'objectif du projet était donc de créer une application pour smartphone Android sous le [moteur de jeu](#) Unity permettant de jouer à deux mini-jeux issus de cette console : 100 FLOOR et BP. Une attention particulière devait être apportée pour reproduire l'expérience de jeu de cette petite console.

La majeure partie de l'objectif fixé a été atteinte (source du projet [7]). Tout d'abord, l'apparence générale de la console ainsi que ses boutons sont représentés sur l'écran du smartphone, donnant l'illusion de tenir la console en main. Le menu de pause est identique à celui proposé par la console d'origine, toutefois le menu principal diffère de l'original mais reste néanmoins fonctionnel.

Le jeu 100 FLOOR possède toutes les fonctionnalités essentielles au gameplay mais nécessite malgré tout quelques ajustements pour corriger certains bogues.

Le jeu BP a été partiellement réalisé : seul le déplacement du personnage et l'utilisation des bombes avec les explosions de décors ont été implémentés. Cependant, aucune animation ni [UI](#) n'a été mise en place pour l'instant.

La conception de l'application permet d'ajouter de nouveaux jeux très facilement, avec la réutilisation de composants communs tels que la gestion des contrôles de la console et son affichage.

Ce projet nous a également donné la possibilité de nous familiariser avec les concepts proposés par le [moteur de jeu](#) Unity. Notre expérience durant ce projet, nous a permis de réaliser que la conception de jeux vidéo demande beaucoup de temps pour être menée à bien et fournir un résultat satisfaisant. Nous aurions aimé produire plus de mini-jeux, cependant nous avons dû nous focaliser sur un seul jeu chacun afin de garantir la qualité de leur réalisation. Cette expérience de projet nous a permis de mieux comprendre les défis de la conception de jeux vidéo et nous a donné des bases pour continuer à explorer ce domaine.

Bibliographie

- [1] UNITY TECHNOLOGIES. *Unity documentation*. 2022. URL : <https://docs.unity3d.com/Manual/index.html> (visité le 11/2022).
- [2] UNITY TECHNOLOGIES. *Unity Remote*. 2022. URL : <https://docs.unity3d.com/Manual/UnityRemote5.html> (visité le 11/2022).
- [3] UNITY TECHNOLOGIES. *2021 Gaming Report*. URL : <https://create.unity.com/2021-game-report> (visité le 03/2023).
- [4] Maxime FRAPPAT. *Unity3D : développer en C# des applications en 2D-3D multiplateformes : (iOS, Android, Windows...) / [Maxime Frappat, Jonathan Antoine]*. fre. Epsilon. St Herblain : Éditions ENI, 2016. ISBN : 978-2-7460-9904-3.
- [5] Anthony CARDINALE. *Créez des jeux de A à Z avec Unity : bases et jeux mobiles / par Anthony Cardinale*. fre. 2e édition [mise à jour et enrichie]. Anzin : D-BookeR éditions, 2016. ISBN : 978-2-8227-0535-6.
- [6] MICROSOFT. *Documentation C#*. URL : <https://docs.microsoft.com/fr-fr/dotnet/csharp/> (visité le 11/2022).
- [7] Mathieu VILLEDIEU DE TORCY. *Dépot github du projet*. URL : <https://github.com/MATHVDT/projetZZ3> (visité le 03/2023).

Glossaire

endless-runner Un style de jeu de plateforme dans lequel le personnage du joueur court sans arrêt sur un parcours dangereux et doit être guidé pour sauter, se baisser pour éviter les dangers. [iii](#), [1](#), [14](#)

frame Image à partir d'une séquence d'images qui représentent des graphiques en mouvement. [7](#)

input Entrées utilisateur telles que les touches du clavier, les boutons de la souris et les axes de joystick, pour contrôler le gameplay d'un jeu. Ici cela correspond au pression sur l'écran du smartphone. [10](#), [15](#)

MonoBehaviour La classe MonoBehaviour est héritée par chaque script Unity et fournit un cadre permettant d'attacher les scripts à un GameObject dans l'éditeur, ainsi que des liens vers des événements spécifiques. [7](#)

moteur de jeu Logiciel qui permet de créer des jeux vidéo en fournissant des fonctionnalités de base telles que la gestion de la physique, des graphismes, de l'audio, des entrées utilisateur, etc. [iii](#), [1](#), [3](#), [5](#), [35](#)

Pocket Dream Console (PDC) Petite console de jeu portable. [ii](#), [iii](#), [iv](#), [1](#), [2](#), [3](#), [29](#), [35](#)

pool Ensemble de ressources réutilisables. [22](#), [23](#)

raycast padding Propriété du Collider dans Unity qui ajuste la distance de détection pour éviter les interactions indésirables. [10](#)

sprite Objet graphique 2D qui peut être une image, une texture. [6](#), [10](#), [11](#), [19](#), [20](#), [21](#), [29](#), [31](#), [32](#), [34](#)

user interface (UI) Interface utilisateur, ensemble des éléments visuels et interactifs d'un système informatique permettant à l'utilisateur de communiquer avec celui-ci. [7](#), [34](#), [35](#)