

Projet de semestre **P3** en Informatique

Orientation : Technologies de l'information et de la communication (TIC)

Projet n°232
Jeu mobile : SchoolBoyBattle

Réalisé par
Valentin Prétat

Encadrement Mr. Gobron Stéphane
HE-Arc, HES-SO

Résumé

0.1 Version française

SchoolBoyBattle est un projet de jeu vidéo ayant comme plateforme cible les appareils mobiles, comme les téléphones et les tablettes. Dans ce jeu de 2 à 10 joueurs, deux équipes s'affrontent en récoltant le plus de points possible avant la fin du temps imparti. Pour cela, chaque joueur doit récolter des friandises réparties sur le terrain et les ramener à la base sans se les faire voler par l'autre équipe !

Ce projet a été réalisé avec le moteur de jeu Godot, permettant un développement rapide et simplifié. L'implémentation du mode de jeu multijoueur a été une étape cruciale afin de garantir une qualité minimum lors de la synchronisation des clients et du serveur.

Au final, le jeu est supporté sur téléphone et tablette et est jouable entre plusieurs appareils présents dans le même sous-réseau.

0.2 English version

SchoolBoyBattle is a video game project targeting mobile devices, such as phones and tablets. In this game of 2 to 10 players, two teams compete by collecting as many points as possible before the time runs out. To do this, each player must collect candies spread over the field and bring them back to the base without being stolen by the other team!

This project was made with the Godot game engine, allowing for a quick and simplified development. The implementation of the multiplayer game mode was a crucial step in order to guarantee a minimum quality when synchronizing the clients and the server.

In the end, the game is supported on phones and tablets and is playable between multiple devices on the same network.

Contents

Résumé	i
0.1 Version française	i
0.2 English version	i
Table des acronymes	iv
1 Introduction	1
1.1 Contexte	1
1.2 Problématique générale	1
1.3 Stratégie utilisée	2
1.4 Etat de l'art	3
1.5 Prérequis	3
1.6 Séminaire	4
2 Conception	5
2.1 Fonctionnalités	5
2.2 Maquettes	6
3 Implémentation	8
3.1 Structure des scènes	8
3.1.1 L'instanciation dans Godot	8
3.1.2 Structure des scènes	8
3.1.3 Les Singltons	11
3.2 L'interface graphique	11
3.2.1 Navigation	11
3.2.2 Fonctionnement	12
3.3 Gestion du multijoueur	13
3.3.1 Infrastructure	13
3.3.2 Les rôles d'une instance	16
3.3.3 Création d'une partie	16
3.4 Le joueur	17
3.4.1 Animations	17
3.4.2 Déplacements	18
3.5 Les friandises	19
3.5.1 Gestion des types	19
3.5.2 File de friandises	20
3.5.3 Validation	21

4 Tests et résultats	23
4.1 Résultat	23
4.2 Améliorations	26
5 Discussion	27
5.1 Conclusion	27
5.2 Perspective	27

Table des acronymes

Quelques mots tabulés

HE-Arc	Haute Ecole Arc, Neuchâtel
HES-SO	Université des Sciences et Arts Appliquée de Suisse Occidentale
GDScript	Language de programmation utilisé dans le moteur de jeu Godot
API	Interface de Programmation d'Application
POO	Programmation Orientée Objet
HUD	Heads Up Display (interface d'un jeu affichée sur l'écran)
Instance	Une exécution du programme qui tourne indépendamment des autres

Chapter 1

Introduction

Cette section situe le contexte du projet, présente les problématiques et détaille la méthodologie employée.

1.1 Contexte

Le projet a démarré le mardi 21 septembre 2021 et s'est terminé le jeudi 23 décembre 2021. Les heures de travail étaient réparties entre les cours de Projet P3 les mardi matin et le travail à effectuer à la maison.

Les quatre leçons de cours par semaine le mardi ont permis d'avoir un rendez-vous hebdomadaire entre moi-même et Mr. Gobron afin d'attester de l'avancée du projet et de définir une stratégie pour la suite.

L'objectif de ce projet est la réalisation d'un jeu vidéo ayant comme plateforme cible les appareils mobiles comme les téléphones portables et les tablettes.

Ce projet consiste en un jeu vidéo multijoueur où chaque joueur incarne un écolier évoluant sur le terrain d'une école primaire. Les joueurs sont séparés en deux équipes : une rouge et une noire, représentées respectivement par St-Nicolas et par le Père Fouettard.

Sur le terrain de jeu se trouvent des bonbons de différents types. Le but pour chaque joueur est de ramasser les bonbons présents sur le terrain et de les ramener à sa base, où se trouve St-Nicolas ou Père Fouettard. Dès qu'un bonbon est ramené à la base, il va ajouter un certain nombre de points à l'équipe du joueur qui l'a ramassé. Le nombre de points est défini selon le type et la taille du bonbon ramené. Une partie de jeu dure 3 minutes, et à la fin de ce temps c'est l'équipe qui aura réussi à ramener le plus de points qui remportera la partie. Un classement est également fait entre tous les joueurs afin de déterminer qui a ramené le plus de points durant la partie.

1.2 Problématique générale

Les caractéristiques de ce projet apportent deux problématiques principales :

1. Quelles sont les étapes à suivre pour la création d'un jeu vidéo ? ;
2. Quels sont les enjeux et difficultés de l'implémentation d'un mode multijoueur ?

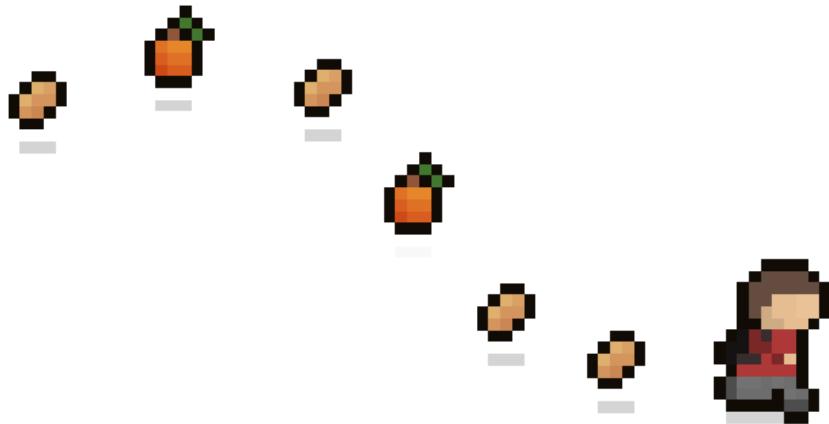


Figure 1.1: Un joueur suivi de sa file de bonbons, composé de petites cacahuètes et de petites mandarines.

Le premier point est une question qui, grâce à l'immense industrie du jeu vidéo, a été soulevée maintes fois. Le résultat est la présence d'un grand nombre de moteurs de jeu sur le marché : des logiciels spécialisés dans la création de programmes vidéo-ludique facilitant la tâche au développeur en incluant des outils répondant aux problématiques récurrentes.

Parmi les plus connus se trouvent Unity, Unreal Engine, Cry Engine, Construct, Godot ou encore Game Maker. Chaque moteur de jeu possède ses propres caractéristiques, rendant la tâche du choix cruciale afin de travailler avec un moteur de jeu adapté au projet escompté.

La question de l'implémentation d'un mode de jeu multijoueur est souvent dirigée par la décision de la première problématique. En effet, certains moteurs de jeu intègrent une API de haut niveau simplifiant grandement l'implémentation d'une synchronisation multijoueur.

1.3 Stratégie utilisée

Parmi les différents moteurs de jeux présents sur le marché, deux d'entre eux ressortent pour la création d'un jeu vidéo en deux dimensions pour téléphone portable : Unity et Godot.

- Unity est un moteur de jeu multi-plateforme développé par Unity Technologies. Sa principale force est sa popularité auprès des développeurs indépendants et des grands studios de production. Le prototypage y est rapide et un grand nombre de plateformes est disponible ;
- Godot Engine est un moteur de jeu multi-plateforme doté de nombreuses fonctionnalités permettant de créer des jeux en 2D et en 3D à partir d'une interface unifiée.^[1] Sa politique de distribution, sa facilité d'utilisation et son prototypage rapide en font un logiciel largement apprécié de sa communauté. Il est entièrement gratuit et open source sous la licence MIT.

Parmi ces deux propositions, le choix du moteur de jeu s'est porté sur Godot.

1.4 Etat de l'art

Le marché du jeu vidéo pour téléphone portable ou tablette est florissant depuis plusieurs années, grâce à l'expansion de l'utilisation de ce type d'appareil. Les créations vidéo ludiques ainsi que les outils de création ciblant ce marché se sont multipliés, et il est aujourd'hui de plus en plus facile et accessible de réaliser un jeu vidéo pour son téléphone.

De part la nature de ce projet, le choix des technologies utilisables est rapidement réduit. Voici les outils utilisés afin de réaliser ce projet :

Aspect technique

- **Godot** est le moteur de jeu qui a été utilisé pour créer ce projet. De part les nombreuses fonctionnalités qui viennent avec, peu d'autres outils ont été nécessaires pour réaliser la partie technique du projet ;
- **L'API multijoueur de Godot** est ce qui a permis de faire communiquer plusieurs instances du programme entre elles afin de créer l'aspect multijoueur en temps réel ;
- **Visual Studio Code** est l'outil utilisé afin de réaliser les opérations avec le repository Git du projet. Son interface facile d'utilisation et sa grande intégration avec le système de gestion de versions est ce qui a poussé à l'utiliser au lieu de l'outil Git intégré à Godot.

Aspect visuel

- **Aseprite** est un logiciel permettant l'édition d'images, conçu pour éditer et animer des pixel-arts ;
- Les ressources visuelles du jeu ont été reprises du projet P2 qui s'est déroulé en 2019 - 2020, puis légèrement modifiée afin d'être intégrable dans Godot.

Outils

- Le **Gitlab** de la HE-Arc est la plateforme utilisée afin de sauvegarder le code du projet avec Git ;
- **Excalidraw** est une application web permettant de réaliser des diagrammes ayant l'aspect d'avoir été dessinés à la main. Excalidraw a été utilisé pour illustrer ce rapport ;
- **Overleaf** est un éditeur LaTeX en ligne, collaboratif et en temps réel. Il a été utilisé pour rédiger ce rapport.

1.5 Prérequis

Comme le projet a été réalisé en grande partie en utilisant Godot, ce document parle d'aspects techniques qui sont spécifiques à ce moteur de jeu.

Afin de comprendre les différents termes énoncés, il est important d'être familier avec les concepts clés de Godot. La documentation officielle [2] possède une page dédiée à cette cause, trouvable à ce lien : https://docs.godotengine.org/en/latest/getting_started/introduction/key_concepts_overview.html

1.6 Séminaire

Les 9 et 17 décembre 2021, un séminaire d'introduction à Godot destiné aux classes INF3dlm-a et INF3dlm-b s'est déroulé à la HE-Arc de Neuchâtel. Ce séminaire a offert aux étudiants en informatique une introduction à un moteur de jeu qui n'est pas utilisé dans le programme annuel du cours d'infographie. Au deuxième semestre, les étudiants découvriront Unity, un moteur largement utilisé dans l'industrie du jeu vidéo. Le but était donc d'offrir aux étudiants la possibilité de faire eux-même un comparatif entre deux moteurs de jeu afin d'élargir leurs connaissances et d'avoir un avis plus critique sur les outils à leur disposition.

Durant ce séminaire, une version simplifiée du Pong a été créé en live pour que la classe puisse recopier les étapes et créer leur version du jeu. Ce séminaire a permis d'introduire les étudiants aux concepts de noeuds, des scènes, de scripting et de collisions dans le moteur de jeu Godot. La figure 1.2 montre le résultat dans l'éditeur de à quoi ressemble le jeu terminé.

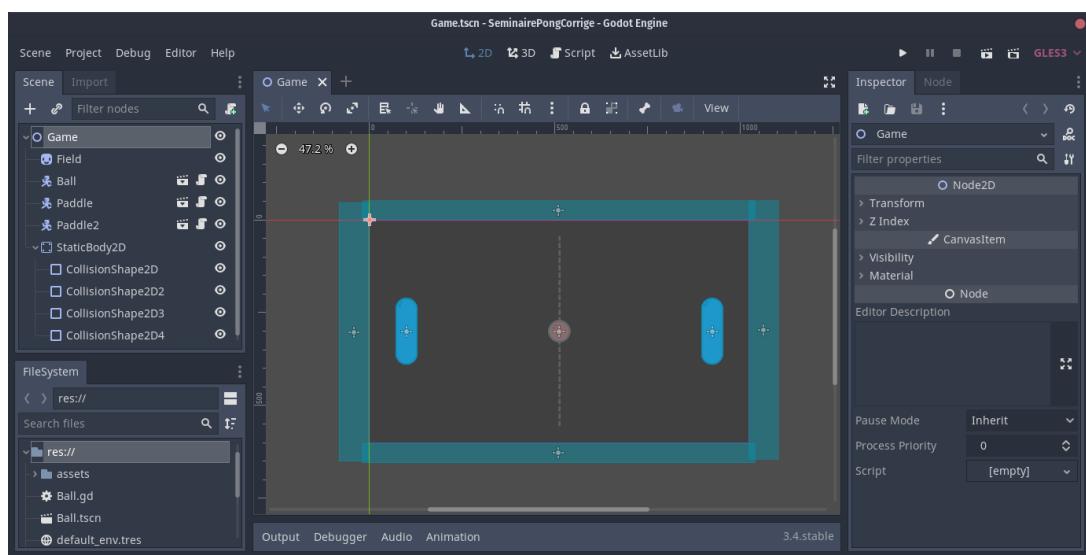


Figure 1.2: Un aperçu du Pong réalisé en classe lors du séminaire.

Chapter 2

Conception

Cette section présente les objectifs attendus et les maquettes représentant une prévisualisation du jeu.

2.1 Fonctionnalités

Le projet contient plusieurs buts de réalisation, représentés par les objectifs primaires et secondaires. Chaque but représente les fonctionnalités minimales à implémenter afin d'avoir un projet cohérent et complet lors de son rendu.

Voici une liste des objectifs primaires et des objectifs secondaires liés à ce projet :

Objectifs primaires

- Recréation du jeu comme il était dans l'ancien projet ;
- Connexion multijoueur ;
- Jeu en deux équipes ;
- Somme des points pour les deux équipes avec écran de victoire en fin de la partie.

Objectifs secondaires

- Scoring (i.e. comptage des points) :
 - Tableau des scores avec les autres joueurs ;
 - Somme des points des parties sauvegardés sur l'appareil.
- Plusieurs cartes disponibles ;
- Vote pour le choix de la carte ;
- Certains bonbons peuvent être mangé impliquant un effet, e.g. :
 - Bonbonbonus :
 - * SpeedBonbon : Le joueur avance plus vite ;
 - * AllInOneBonbon : Les bonbons sont tous cachés pendant un dt dans un unique bonbon, qui peut évidemment être pris ;
 - * BonbonOfSteel : Les bonbons sont imprenables, eg. Chaine en métal ;

- * MagnetBonbon : L'avatar génère un champ de force (rayon) dans lequel les bonbons sont attirés.
- Anti-bonus : avoir des bonbons pièges si on les mange :
 - * GetSlowBonbon : l'avatar est stoppé puis reprend sa vitesse petit à petit ;
 - * UnchairMyHeartBonbon : La chaîne de bonbon explose, les bonbons sont répartis au hasard ;
 - * BlackBonbonOfTheDeath (unique sur tout le jeu et est facilement reconnaissable) : le joueur qui le touche meurt et fait gagner la partie instantanément à l'autre joueur.
- Options de jeux :
 - Reconsidérer l'action de manger des bonbons par récupérer des gemmes -> choix pédagogique où dès lors les bonbons existent mais sont mangés automatiquement ;
 - Plus on amasse de bonbons, plus on est lent ;
 - Plus on mange de super bonbons, plus on grossit et est lent ;
 - Les avatars peuvent se pousser.

2.2 Maquettes

Les figures 2.1 et 2.2 ont été réalisées avant le début du projet. Dès que les ressources visuels du jeu ont été créées, un montage a permis de se fixer un objectif sur l'aspect visuel du jeu et des fonctionnalités à implémenter.



Figure 2.1: Prévisualisation de la zone "intérieur" du jeu.



Figure 2.2: Un aperçu de la partie "extérieur" du jeu, avec les décorations pour la neige.

Chapter 3

Implémentation

Ce chapitre décrit les fonctionnalités principales du projet, les problématiques qu'elles posaient et les solutions utilisées afin de les implémenter.

3.1 Structure des scènes

3.1.1 Linstanciation dans Godot

La structure des composants d'un programme réalisé avec Godot est composée à son plus bas niveau de noeuds et de scènes. Une scène sur Godot peut être considérée comme une classe en POO, et les instanciations de cette scène peuvent être considérées comme étant les objets d'une classe.

Dans Godot, linstanciation d'une scène se poursuit en général par lajout de cette instantiation au *SceneTree* afin que la scène instanciée soit présente dans le programme en cours d'exécution. Le *SceneTree* est une structure sous forme arborescente comportant les noeuds qui seront traités par le moteur; toute scène instanciée n'étant pas ajoutée au *SceneTree* est dite "orpheline", et ne sera pas prise en compte par le moteur de jeu.

Le *SceneTree* contient à sa racine le noeud *root* que le moteur utilise comme point d'entrée pour exécuter récursivement chaque noeud enfant lors de chaque rafraîchissement d'image.

3.1.2 Structure des scènes

Dans ce document, la structure des scènes d'un projet Godot consiste en la représentation de linstanciation de scènes dans d'autre scènes.

Il est important de noter que la représentation des scènes telle que décrite dans ce chapitre n'est pas une représentation des noeuds du *SceneTree*, mais plutôt une explication simplifiée de la structure du programme et de comment s'imbriquent les composants principaux.

Dans le cadre de ce projet, deux structures distinctes sont utilisées dépendant de l'état dans lequel se trouve le jeu :

- Aucune partie n'est en cours, un écran de menu est affiché à l'écran ;
- Une partie est en cours et est affichée à l'écran.

Dans le premier cas où il n'y a pas de partie en cours, la structure des scènes peut être observée sur la figure 3.1. La particularité de cette structure consiste en l'utilisation de deux scènes pour afficher un menu :

1. La scène *MenuContainer* ;
2. Une scène affichant le menu (dans notre cas la scène *Home*).

Cette structure se rapporte au patron de conception appelé "Machine d'état" [3] [4] : chaque menu peut émettre un signal qui est envoyé au *MenuContainer* lui indiquant la nouvelle interface à afficher. L'avantage de cette solution est qu'il est simple d'implémenter des transitions animées entre les différents menus du jeu.[5] Aussi, le fond visuel de tous les menus reste commun car il est défini dans la scène *MenuContainer*.

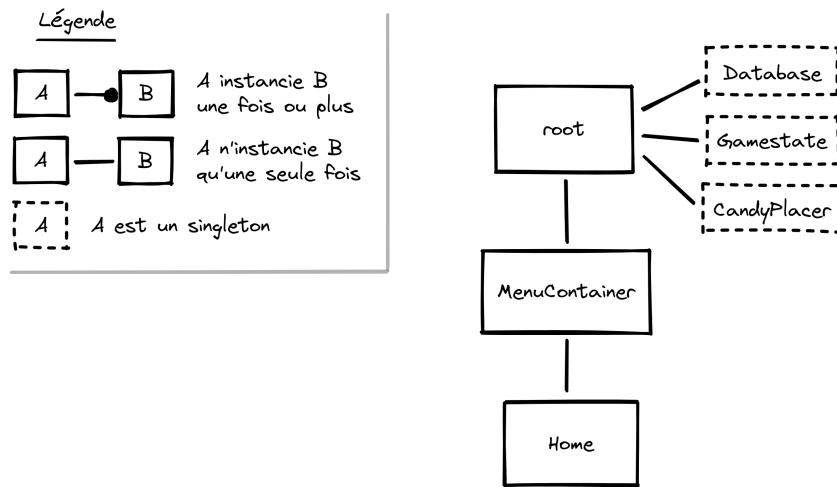


Figure 3.1: Schéma indiquant les instanciations de scènes lorsqu'on se balade dans les menus.

L'élément *root*, quant à lui, n'est pas une scène définie dans le projet mais est le noeud de base de tout programme réalisé avec Godot.

Pour en savoir plus sur le fonctionnement des menus du jeu, voir le chapitre 3.2.2.

Dans le cas où une partie est en cours et s'affiche à l'écran, la figure 3.2 décrit la structure des scènes.

Voici une liste indiquant le rôle de chacune de ces scènes :

- **Game** : S'occupe de l'initialisation d'une partie en instantiant les scènes enfant, en connectant des signaux entre certains objets et en mettant en place le tout selon si le programme exécuté est un serveur ou un client. S'occupe également de compter les points des équipes et de gérer la fin d'une partie ;
- **HUD** : Se trouve sur un *CanvasLayer* différent, faisant que son contenu s'affiche statiquement à l'écran et non dans le système de coordonnées du terrain de jeu. Contient les éléments de l'interface graphique ;
- **Joystick** : Scène contenant le *Joystick* permettant de déplacer le joueur. Envoie un signal à chaque fois que le joueur interagit avec. Le signal est directement connecté au noeud du joueur qui se déplace en conséquence ;

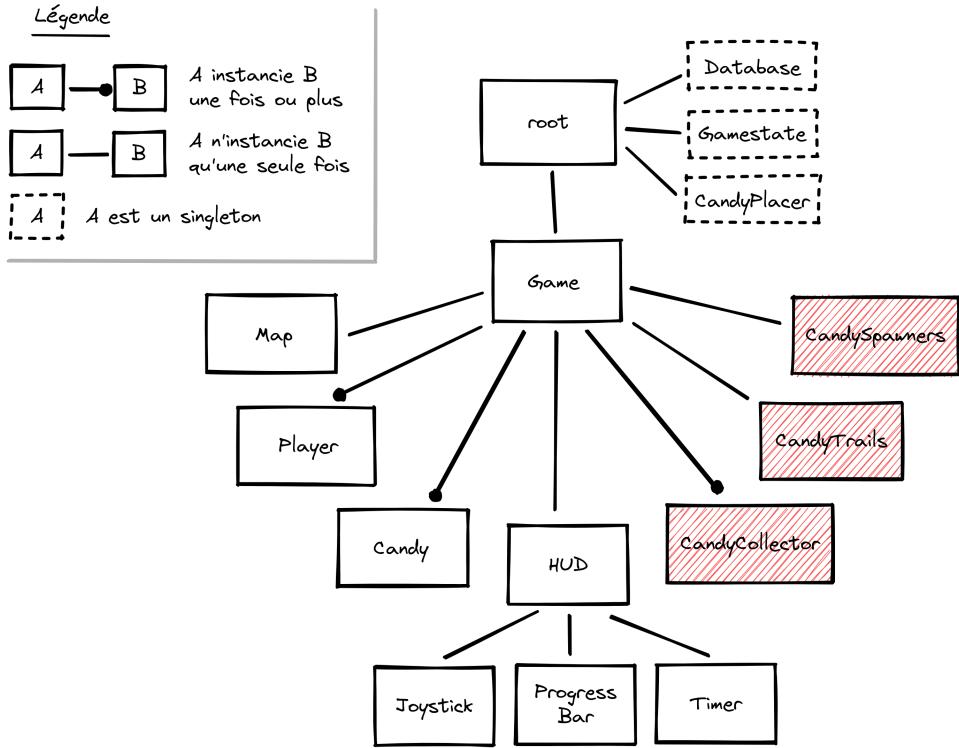


Figure 3.2: Schéma indiquant les instantiations de scènes lorsqu'une partie est en cours.

- **ProgressBar** : Contient la *ProgressBar* en haut de l'interface utilisateur indiquant les points de chaque équipe ;
- **Timer** : Le minuteur indiquant le temps restant à la partie en cours ;
- **Player** : La scène décrivant le joueur. Cette scène est instanciée autant de fois qu'il y a de joueur **sur chaque instance du jeu**. Par exemple, si une partie se déroule avec 4 joueurs, la scène *Player* sera instanciée un total de 16 fois entre tous les appareils jouant au jeu. Cette scène a le rôle de déplacer le joueur selon l'input reçu du *Joystick*, ramasser les friandises avec lesquelles le joueur entre en collision et gérer les animations du joueur selon différents paramètres ;
- **Candy** : Cette scène est la scène contenant tout type de friandise visible sur l'écran. Lors de l'initialisation d'une friandise, le type du noeud racine de la scène définit l'animation à jouer et le nombre de points que la friandise rapporte à l'équipe. Cette scène gère également la capture d'une friandise par le joueur ;
- **Map** : Scène contenant la carte du jeu avec certaines zones définies, comme les points d'apparition des friandises, des joueurs, l'emplacement de St-Nicolas et de Père-Fouettard ainsi que la base de chaque équipe ;
- **CandyTrail** : Scène servant à placer les friandises derrière le joueur en file indienne lorsqu'il les a ramassées ;
- **CandySpawners** : Scène permettant de choisir un point d'apparition pour la prochaine friandise. le *CandySpawner* s'occupe également de prévenir les instances client de faire apparaître une friandise au même endroit ;

- **CandyCollector** : Scène permettant d'animer la récolte des friandises lorsqu'elles sont amenées à la base d'une équipe.

Les scènes avec un fond rouge représentent celles qui ne sont instanciées que sur le serveur.

3.1.3 Les Singlenton

Dans les figures 3.2 et 3.1 du chapitre 3.1.2 se trouvent des rectangles traitillés décrits comme étant des *Singlenton*.

Godot offre un système de *Singleton* appelé "Autoload", permettant à un script d'être appelé statiquement depuis n'importe où dans le projet. Bien que cette fonctionnalité ne devrait pas être sur-utilisée pour garder une implémentation propre, elle est peut s'avérer utile selon les fonctionnalités à implémenter dans le jeu. Dans notre cas, 3 *Autoloads* sont présents dans le projet :

- **Database** : script offrant une multitude de constantes utilisées à plusieurs endroits dans le projet. Ce script s'occupe également d'enregistrer et de lire les données sur le téléphone, comme les points gagnés à chaque partie ;
- **Gamestate** : script gérant la création d'un serveur, sa gestion et la possibilité de se connecter à un serveur si on est un client. Il s'occupe également de démarrer une partie en synchronisant tous les joueurs ;
- **CandyPlacer** : script utilisé uniquement sur le serveur offrant la fonction *place()*, qui va placer une liste de bonbons le long d'une ligne définie par un noeud *Line2D*. Ce script est utilisé depuis la scène *Trail* et depuis la scène *CandyCollector*.

3.2 L'interface graphique

3.2.1 Navigation

Sur la figure 3.3 est décrit un schéma de navigation pour les différents menus du projet.

Parmi ces différents croquis d'interface, quelques observations peuvent être faites. La première est que le bouton *Jouer* de l'écran *Home* envoie à 2 écrans différents. Ceci est dû au fait que le programme sauvegarde en mémoire le pseudonyme de l'utilisateur entre les parties. Ainsi, si l'utilisateur lance le jeu pour la première fois, il lui sera demandé d'entrer un pseudonyme. Si le pseudonyme est déjà défini, le bouton *Jouer* enverra l'utilisateur sur le menu *ChooseMode*.

Ensuite, il est notable que sur l'écran *ChooseMode* se trouvent 3 boutons permettant de choisir un mode de jeu, mais que seul 1 est activé. Les boutons désactivés ajoutent de nouveaux modes de jeu qui n'ont pas été implémentés dans ce projet à la date de son rendu, mais qui pourraient constituer une suite au développement du jeu. Voir le chapitre 4.2 pour plus d'information.

Finalement, deux types de flèches lient les croquis de menus entre eux : les flèches remplies et les flèches pointillées :

- Les flèches remplies représentent les transitions qui surviennent à partir de l'interaction de l'utilisateur, comme la pression d'un bouton ;

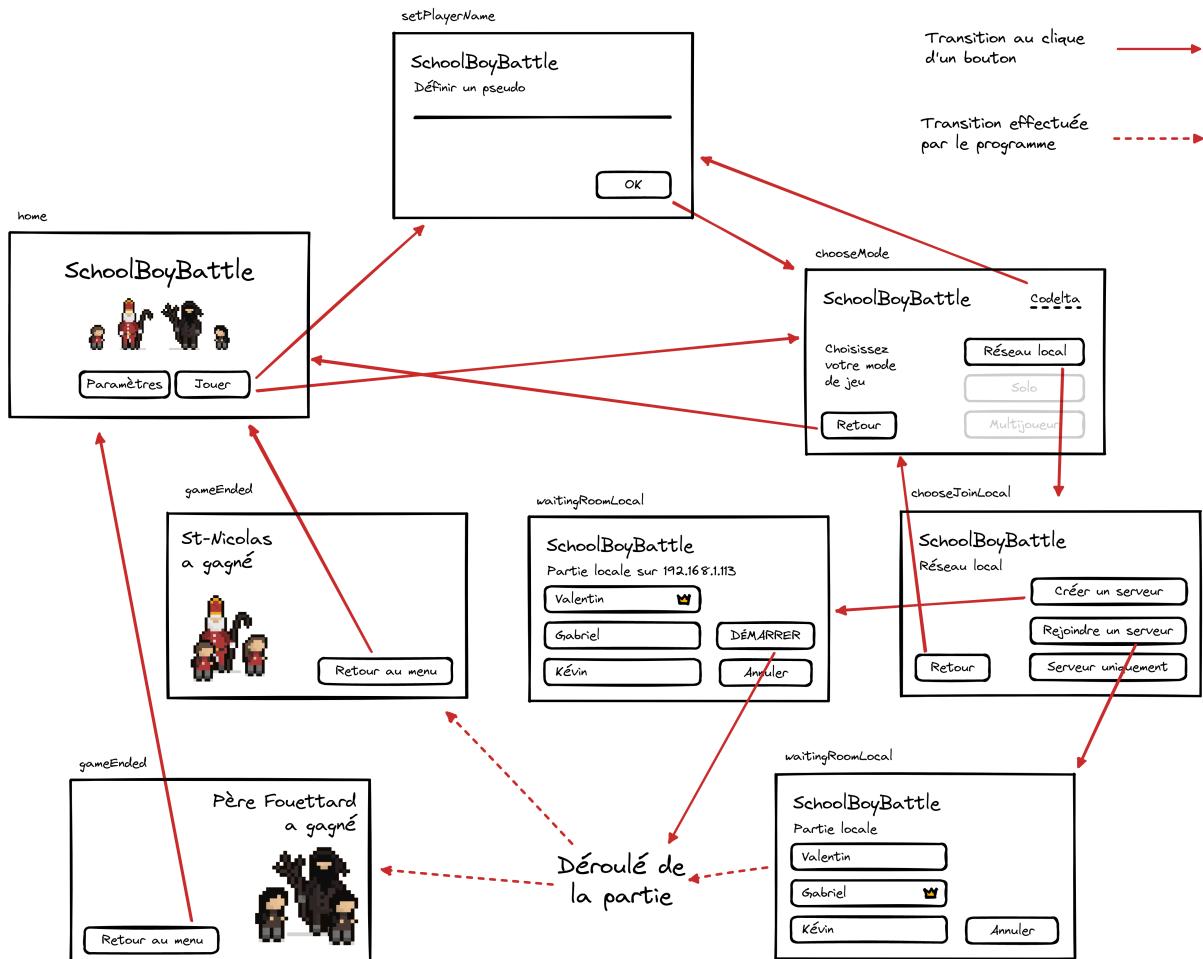


Figure 3.3: Schéma indiquant les transitions et relations entre les différents menus du jeu.

- Les flèches pointillées représentent la transition d'un menu à un autre effectué automatiquement, comme par exemple lorsque la partie démarre après que le serveur l'ait demandé. Tous les clients vont automatiquement changer d'interface pour afficher le jeu sans que l'utilisateur n'ait eu d'interaction à faire.

3.2.2 Fonctionnement

Le fonctionnement de l'interface graphique repose sur l'utilisation d'un patron de conception : la *Machine d'état*.[6] [3]

Ce patron de conception est composé de 3 classes qui interagissent entre elles. La première (dans notre cas *MenuContainer*) est notre classe de base possédant un attribut référençant l'état actuel (dans notre cas le menu actuel).

La deuxième classe est usuellement définie comme une interface ou une classe abstraite. Comme Godot ne possède pas ce type de classe, il s'agit dans notre cas d'une classe simple possédant un signal `set_menu`.

```

1  extends Control
2  class_name Menu
3  signal set_menu(path, direction, parameters)

```

Code 3.1: Code de la classe Menu

La dernière classe représente les différents états que peut avoir notre patron de conception (dans notre cas : les différents menus du jeu). Chacune de ces classes possède dans une ou plusieurs de ses fonctions une ligne de code envoyant le signal *set_menu* au *MenuContainer* lui indiquant de changer de menu.

```

1  extends Menu
2
3  func _on_ButtonPlay_pressed():
4      emit_signal("set_menu", "res://src/ui/menus/choose-mode/
    ChooseMode.tscn", Vector2.DOWN)
5
6  # [...]

```

Code 3.2: Code de la classe Home

Les figures 3.4 et 3.5 représentent l'état de notre *Machine d'état* lorsqu'on se trouve respectivement sur le menu *Home* et sur le menu *WaitingRoom*.

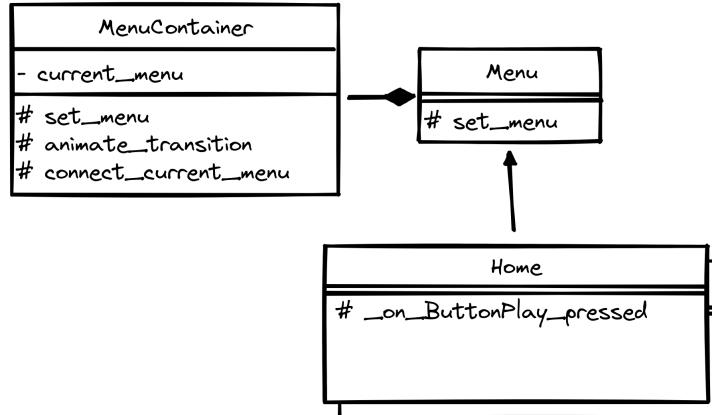


Figure 3.4: Schéma UML où le menu *Home* est affiché.

3.3 Gestion du multijoueur

3.3.1 Infrastructure

Avant même de commencer le développement du projet, prendre la décision de comment fonctionne le multijoueur entre plusieurs utilisateurs a été crucial : l'implémentation de la synchronisation entre des clients est utilisée à beaucoup d'endroits dans le projet. C'est pour cela qu'il est important de définir à l'avance que fait chaque instance du jeu lorsqu'elles communiquent

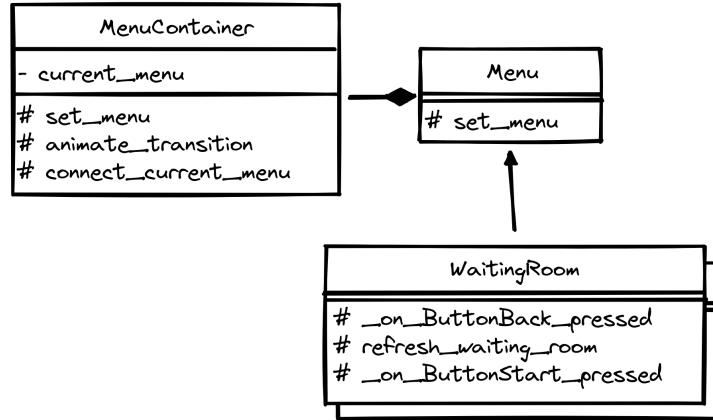


Figure 3.5: Schéma UML où le menu *WaitingRoom* est affiché.

ensemble.[7]

Lors du projet P2 réalisé en fin 2020 et début 2021, l'aspect multijoueur a été implémenté tard dans le développement du jeu. Le résultat était qu'une notable désynchronisation était présente entre tous les clients sur le nombre de points récoltés et sur l'emplacement des différents acteurs sur le terrain de jeu.

Le fonctionnement du multijoueur sur le projet P2 est décrit sur la Figure 3.6.

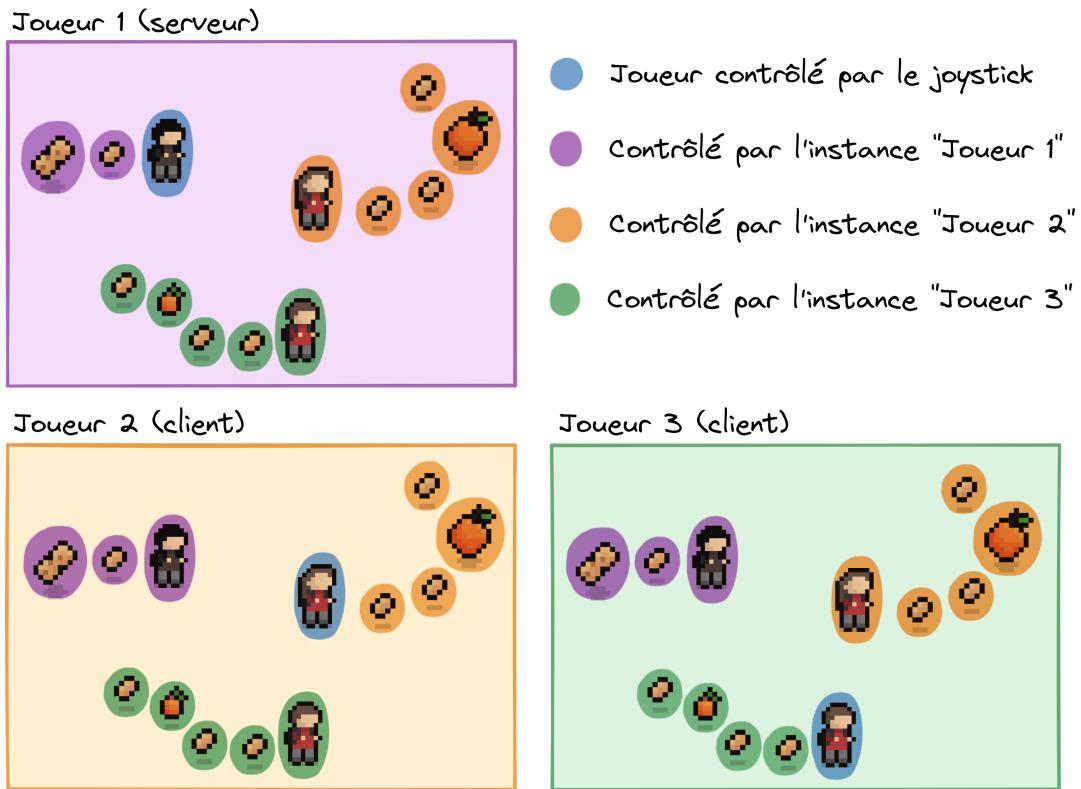


Figure 3.6: Fonctionnement de la synchronisation multijoueur dans le projet P2 en C++ avec Qt.

Dans cette ancienne implémentation, chaque instance s'occupait de placer les friandises récoltées par son joueur à la queue-leu-leu derrière lui, mais également de placer ces friandises au même

endroit sur les autres instances.

Ainsi, chaque instance était responsable de son joueur et des friandises qu'elle avait récolté. Cependant, cette solution supportait mal les problèmes de réseaux qui pouvaient survenir pendant une partie. En effet, il était possible que lorsqu'un joueur A volait des friandises à un joueur B, la désynchronisation causée par le réseau faisait que pour le joueur B aucun bonbon n'avait été volé. Ce problème causait une telle différence de points entre les équipes qu'à la fin de chaque partie les scores finaux étaient différents pour tout le monde.

La principale erreur derrière cette implémentation est le fait que la responsabilité de l'état du jeu était trop partagé entre les différentes instances. Afin de palier aux éventuels problèmes réseaux, la solution est de faire qu'une instance parmi les joueurs soit considérée comme étant l'instance *modèle*. Ainsi, si un joueur prends du retard à cause d'une erreur réseau, il pourra toujours rattraper son retard en se basant sur l'état de l'instance modèle. Cette instance modèle est appelée le **serveur**, et les autres sont les **clients**.

La Figure 3.7 décrit le fonctionnement du multijoueur dans ce projet P3.

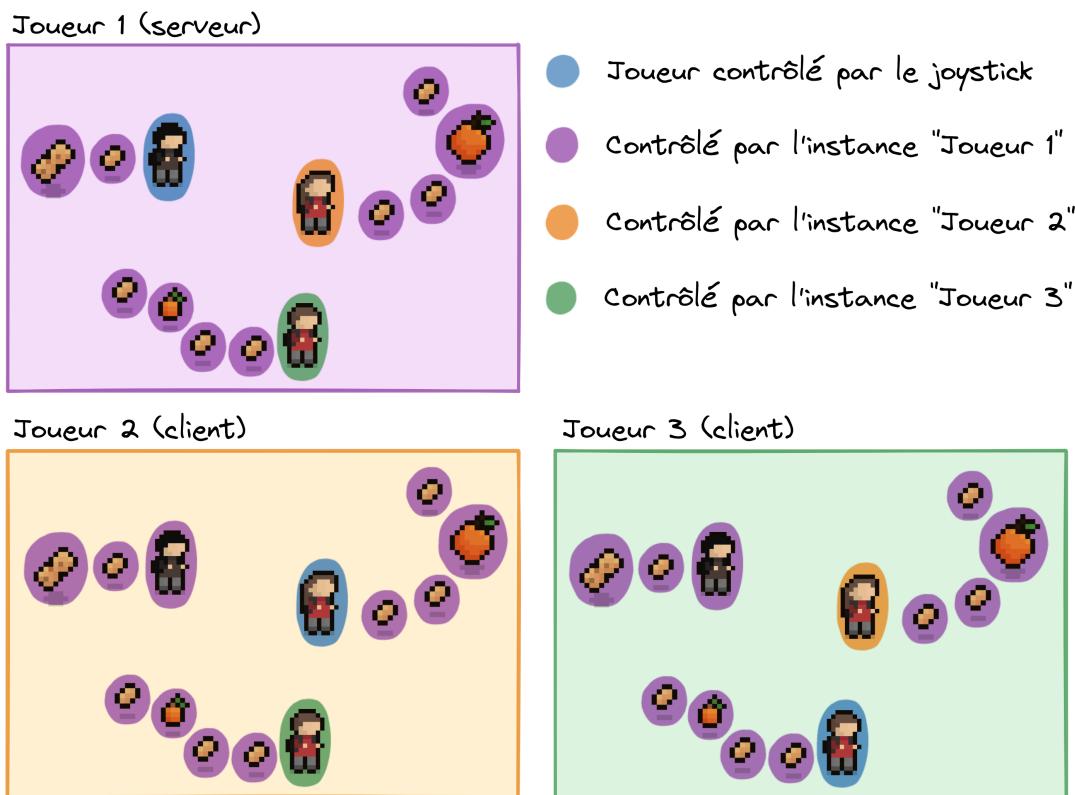


Figure 3.7: Fonctionnement de la synchronisation multijoueur dans le projet P3 développé avec Godot.

Ainsi, avec cette nouvelle implémentation, chaque client ne s'occupe que du déplacement de son propre joueur et de la partager aux autres joueurs.[8]

Le serveur, lui, s'occupe également de placer toutes les friandises de tous les joueurs sur le terrain de jeu en fonction de la position du joueur que chaque instance lui a envoyé.

Cette façon de faire résout notre principal problème de synchronisation : Tant que la latence du réseau n'est pas trop importante, chaque instance peut rattraper son retard en se basant sur l'état que le serveur lui partage.

Cependant, elle apporte également l'inconvénient que le serveur doit être accessible en tout temps. Si le serveur se retrouve déconnecté de la partie, c'est tous les joueurs qui se retrouvent déconnectés en même temps.

3.3.2 Les rôles d'une instance

Avec cette implémentation, chaque instance prends l'un de ces 3 rôles lors d'une partie :

- **Client** : l'instance s'occupe d'envoyer les inputs de son joueur (sa position) au serveur et reçois toutes les informations pour qu'il puisse afficher un état similaire à celui du serveur ;
- **Serveur-client** : c'est lorsqu'un des joueurs de la partie a également le rôle du serveur. Ce joueur aura comme particularité de n'avoir aucune latence, car il est lui-même son propre serveur ;
- **Serveur uniquement** : l'instance ne permet pas de jouer au jeu, mais à la place affiche une vue dézoomée du terrain où évoluent les joueurs. Par la suite, l'objectif des instances *Serveur uniquement* est de faire tourner le programme sur des serveurs dédiés, sans affichage d'interface graphique. Ainsi, les parties de jeu ne seraient plus limitées à un réseau local.

3.3.3 Création d'une partie

L'action de créer une partie est déclenchée par le serveur, lorsque l'utilisateur clique sur le bouton **Démarrer**. À ce moment, le serveur lance la fonction `begin_game()`, présente dans le singleton *Gamestate*.

La figure 3.8 présente les interactions entre le serveur et les clients lors du démarrage d'une partie.

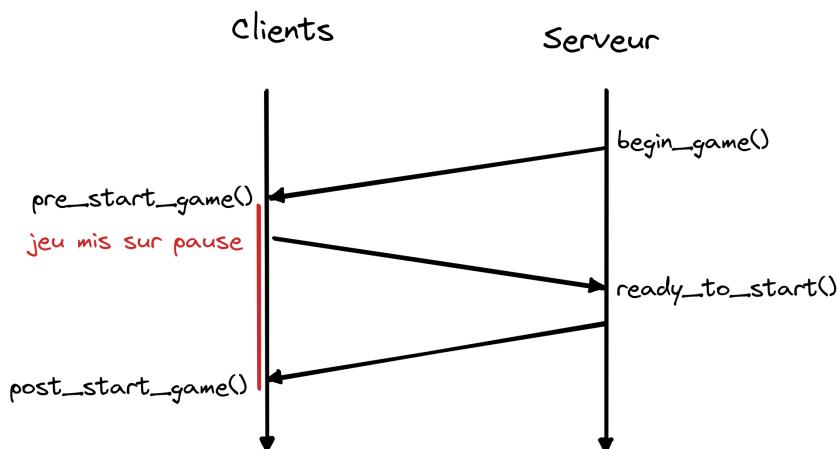


Figure 3.8: Interactions entre les clients et le serveur lors du démarrage d'une partie.

Pour commencer, la fonction `begin_game()` est lancée sur le serveur. Son rôle est de définir et rassembler dans un dictionnaire toutes les informations pour chaque joueur de la partie, comme son équipe, son sexe et son pseudonyme.

Dans la fonction `pre_start_game()`, le jeu est instancié puis mis en pause. Chaque joueur est créé avec les paramètres définit auparavant, et sa relation avec son instance "maître" est définie via la fonction `set_network_master()`. Finalement, quand le client est prêt, la fonction

ready_to_start() du serveur est appelée.

```
1 p.init(p_init["name"], p_init["gender"], p_init["team"],
        spawn_pos)
2 p.set_name(str(id)) # Use unique ID as node name
3 p.set_network_master(id) #set unique id as master
```

Code 3.3: Initialisation d'un joueur, dans la fonction *pre_start_game()*

Le serveur, de son côté, compte le nombre de fois que la fonction *ready_to_start()* a été appelée par un client. Dès qu'elle a été appelée autant de fois qu'il y a de clients dans la partie, elle envoie le message à tout le monde de démarrer la partie. À ce moment, tous les clients ne mettent plus leur jeu sur pause et la partie peut démarrer pour tout le monde.

3.4 Le joueur

Cette section décrit comment sont implémentés les déplacements du joueur en utilisant un joystick, ainsi que la synchronisation des déplacements en multijoueur.

3.4.1 Animations



Figure 3.9: Une des nombreuses feuilles de sprite du joueur.

Le joueur, tout comme les friandises, possède la particularité d'avoir plusieurs types et animations selon les valeurs d'initialisation. Il y a au total 4 types de joueurs qui possèdent chacun leur propres animations selon leurs état :

- Le garçon dans l'équipe rouge ;
- La fille dans l'équipe rouge ;
- Le garçon dans l'équipe noire ;
- La fille dans l'équipe noire.

Et chacun de ces types de joueur possède plusieurs animations, selon son état durant la partie :

- **Idle**, lorsque le joueur est à l'arrêt ;
- **Running**, lorsque le joueur se déplace.

L'animation pour un joueur fille de l'équipe St-Nicolas qui est en train de courir est montrée sur la figure 3.9.

Définir les animations du joueur est une étape qui se déroule à chaque image du jeu : elle se place donc dans le déroulé de la fonction *_process(delta)* de notre script *player.gd*. Dans cette fonction, plusieurs fonctions reliées au joueur sont appelées afin de s'occuper des déplacements

du joueur puis de son animation.

L'étape de l'animation du joueur survient après avoir calculé et effectué ses déplacements. Ainsi, l'animation peut se baser sur des paramètres comme la vitesse ou la position du joueur pour se rendre plus dynamique.

Pour pallier au problème de la multitude de types d'animations à avoir selon le sexe et l'équipe du personnage, Godot nous offre la possibilité de sauvegarder dans un fichier au format *.tres* (visible sur la Figure 3.10) ce qu'on appelle un **SpriteFrames** : une collection d'animations qui peut être dynamiquement chargée durant l'exécution du programme et qui vient se greffer à un noeud de type *AnimatedSprite*.

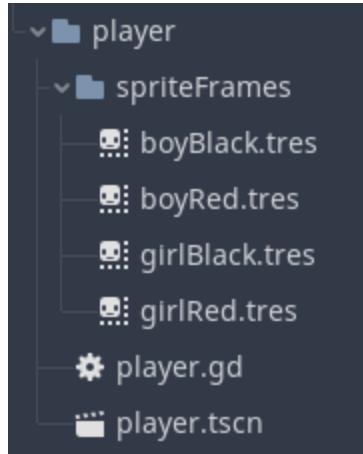


Figure 3.10: Fichiers utilisés pour le joueur.

Ainsi, lors de l'initialisation d'une partie (*c.f.* chapitre 3.3.3), le noeud *AnimatedSprite* de notre scène Player définit quel *SpriteFrames* utiliser pour ses animations.[9]

3.4.2 Déplacements

Les déplacements du joueur est un processus qui implique que plusieurs scènes interagissent entre elles. D'après le schéma 3.2, il s'agit des scènes *Joystick*, *Game* et *Player*.

Afin d'implémenter les déplacements du joueur, il est nécessaire de résoudre un petit problème : Le calcul du vecteur de déplacement créé par le Joystick et le déplacement du joueur à partir d'un vecteur dans la scène Player sont totalement indépendants. La scène Player ne peut pas accéder simplement au vecteur de la scène Joystick : il faut donc faire communiquer ces deux scènes d'une certaine manière.

- Une première façon est d'enregistrer la valeur du vecteur de déplacement dans une variable de la classe Joystick. Puis, en passant une référence de l'objet Joystick au Player lors de l'initialisation, le Player peut reprendre cette valeur à chaque frame afin de se déplacer sur le terrain de jeu ;
- Une deuxième façon est de faire l'inverse : le Joystick possède une référence de l'objet Player et enregistre la valeur du vecteur de déplacement directement dans l'objet Player. Lui n'aurait plus qu'à lire sa propriété pour savoir où se déplacer.

Cependant, aucune de ces solutions ne paraît propre à implémenter : dans les deux cas il faut parcourir le *SceneTree* statiquement pour reprendre l'autre objet. Aussi, chaque scène n'est censé s'occuper que d'elle-même ainsi que de ses scènes enfant. Il n'est donc pas logique que l'objet Joystick ou l'objet Player aient connaissance l'un de l'autre.

Ce problème a été résolu à l'aide des **signaux**, une fonctionnalité de Godot qui permet de faire communiquer des noeuds entre eux.

Connecter le noeud Joystick et le noeud Player se fait à partir d'un noeud qui a accès à ces deux là : le noeud Game. Lors de l'initialisation de la partie, la fonction *set_main_player()* est appelée afin de définir à quel joueur le signal du Joystick doit se connecter.

```

1 func set_main_player():
2     # Connects the joystick's signal to the player of this
3     # instance
4     get_node("CanvasLayer/HUD/Joystick").connect(
5         "use_move_vector",
6         get_node("YSort/Players/" + str(get_tree().
7             get_network_unique_id())),
8         "_on_joystick_event"
9     )

```

Code 3.4: Connexion entre l'objet Joystick et l'objet Player

Ainsi, l'objet Joystick n'a qu'à émettre le signal *move_vector(direction)*. Le signal est connecté à une fonction dans l'objet Player qui sauvegarde ce vecteur dans un membre de la classe. Finalement, dans la fonction *_process(delta)*, le joueur n'a plus qu'à récupérer la valeur de cette variable locale pour savoir dans quelle direction se déplacer.[10]

3.5 Les friandises

3.5.1 Gestion des types

Tout comme les joueurs, les friandises possèdent plusieurs types qui ont chacun leurs propres paramètres :

- Les animations ;
- le nombre de points ;
- l'explosion.

Pour les animations et l'explosion, la même implémentation que l'animation des joueurs a été utilisée (*C.f. chapitre 3.4.1*), chaque type de friandise possédait 3 fichiers :

1. Un fichier en **.gd**, contenant la classe de la friandise (*par exemple SmallPeanut.gd*) ;
2. Un fichier en **.tres** contenant les animations (*par exemple SmallPeanut.tres* ;
3. Un fichier en **.tres** contenant l'explosion en particules lorsqu'un bonbon arrive à la base d'une équipe (*par exemple SmallPeanutExplosion.tres*).

Le déclencheur de la création d'une friandise s'effectue sur le serveur, grâce au script *CandySpawners.gd* qui n'est pas chargé sur les clients. Un type est choisi en utilisant un système de probabilité relié à chaque type de bonbon ainsi un emplacement d'apparition, puis l'information de créer une nouvelle friandise est transmise à tous les clients qui l'ajoutent à leur scène.

3.5.2 File de friandises

La fonctionnalité des friandises suivant le joueur l'ayant ramassé à la queu-leu-leu est une des fonctionnalités primaires de ce projet. Son implémentation est importante, et le choix de comment le faire peut beaucoup changer les stratégies utilisables en jeu.

Dans le projet P2 effectué en 2019 - 2020, la file de bonbons était implémentée d'une façon plus simple que dans ce projet P3 : chaque bonbon était déplacé petit à petit en direction du bonbon qui se trouvait devant lui. Pour atteindre ce résultat, une méthode d'interpolation linéaire avait été mise en place.

Cette solution possédait tout de même quelques problèmes : si le joueur tournait autour d'un mur, la file de friandises avait tendance à couper à travers ce mur au fur et à mesure que le joueur avançait. Aussi, la position de chaque bonbon était déterminée par la position du bonbon d'avant. Si un des joueurs, par un problème réseau, se retrouvait en retard par rapport aux autres et rattrape son retard sur l'état du jeu (causant par la même façon une petite téléportation du joueur dans le jeu pour retrouver son vrai emplacement), toute la file de friandise se retrouvait mal placée. Finalement, lorsque le joueur s'arrêtait, toute la file de friandise se concentrait à sa position.

Afin de pallier à ces problèmes, une toute autre façon de placer les friandises sur le terrain a été implémentée :

Chaque joueur sur le terrain dessine en tout temps une ligne invisible qui trace ses déplacements. À chaque fois que le joueur retourne à sa base, cette ligne est effacée et son traçage repart de zéro.

Cette ligne permet de placer sur le terrain les friandises que le joueur a capturé. À chaque image, le serveur boucle sur chaque friandise à placer et en utilise le singleton *CandyPlacer* afin d'en déterminer la position.



Figure 3.11: Placement des friandises sur le terrain à l'aide de la ligne rendue visible.

Cependant, un problème relié à l'implémentation du multijoueur persiste : Comme vu sur la Figure 3.7, la file de friandises d'une instance client est définie par le serveur. Cela signifie que

pour qu'une instance client voit ses friandises se faire placer derrière son joueur, le jeu effectue :

1. Le joueur sur le client se déplace en suivant la direction du Joystick ;
2. La position de ce joueur est envoyée au serveur ;
3. Le serveur calcule ensuite la position de chaque bonbon pour ce joueur ;
4. La position de chacune des friandises est envoyée au client ;
5. Le client affiche chaque bonbon à la position qu'il a reçue du serveur.

Cette marche à suivre peut paraître longue, mais c'est une concession à faire pour avoir un multijoueur où les problèmes réseaux aient un impact réduit.

Visuellement, la longueur de ce processus est visible sur les clients au moindre problème réseau : les friandises derrière lui auront du mal à le suivre et se déplaceront d'une façon saccadée.

Une bonne solution consiste à ne plus transmettre la position réelle des bonbons entre le serveur et les clients, mais une position "cible". Les clients, eux, utiliseront de l'interpolation linéaire sur la position de chaque bonbon pour qu'il se rapproche de sa position cible. Le résultat est que même si la position cible de chaque bonbon est vulnérable aux bugs de réseau, la position réelle n'en sera que peu impactée, le tout en donnant un côté plus fluide aux déplacements des friandises.

3.5.3 Validation

Lorsque le joueur arrive à sa base, les bonbons se font valider et le nombre de points de son équipe est augmenté du total de points des friandises que ce joueur avait capturé.

Initialement, dès que le joueur entrait en collision avec la zone délimitant sa base, tous les bonbons disparaissaient d'un coup et le nombre de points était immédiatement ajouté. Afin d'ajouter un aspect visuel plus intéressant, il a été choisi d'animer la validation des friandises.

L'animation consiste à faire accélérer la file de friandises en direction de la base de l'équipe dès que le joueur y entre en contacte. À ce moment, la file de friandises ne peut plus être volée par d'autre joueurs.

Durant l'animation, la file de bonbons n'est plus rattachée au joueur. Il faut donc un autre noeud qui va prendre le relais au joueur afin de s'occuper de placer ces bonbons jusqu'à ce que l'animation soit terminée.

La Figure 3.12 montre les 4 étapes qui se déroulent à chaque capture d'une file de friandises. Imaginons un cas où le joueur avec l'ID 1 retourne à sa base :

1. Le joueur s'approche de sa base, sa file de friandises le suit. Le joueur contient un tableau référençant chaque bonbon qu'il possède dans sa file, le serveur utilise ce tableau pour placer les bonbons derrière lui ;
2. Arrivé dans sa base, la ligne *Trail1* est remise à zéro. L'ancienne ligne et le tableau de friandises du joueur sont copiés avant d'être vidés. La ligne *Collect1* est créée, et possède comme variable le tableau copie des bonbons que le joueur avait capturé. La ligne *Trail1* utilise ce tableau pour déplacer les friandises petit à petit vers la base du joueur ;
3. Une fois qu'un bonbon atteint le bout de la ligne *Collect1*, il est supprimé en jouant une petite animation d'explosion ;

4. Une fois que tous les bonbons ont été amenés à la base, la ligne *Collect1* se supprime elle-même. Le processus est prêt à être recommencé la prochaine fois que le joueur atteint la base de son équipe.

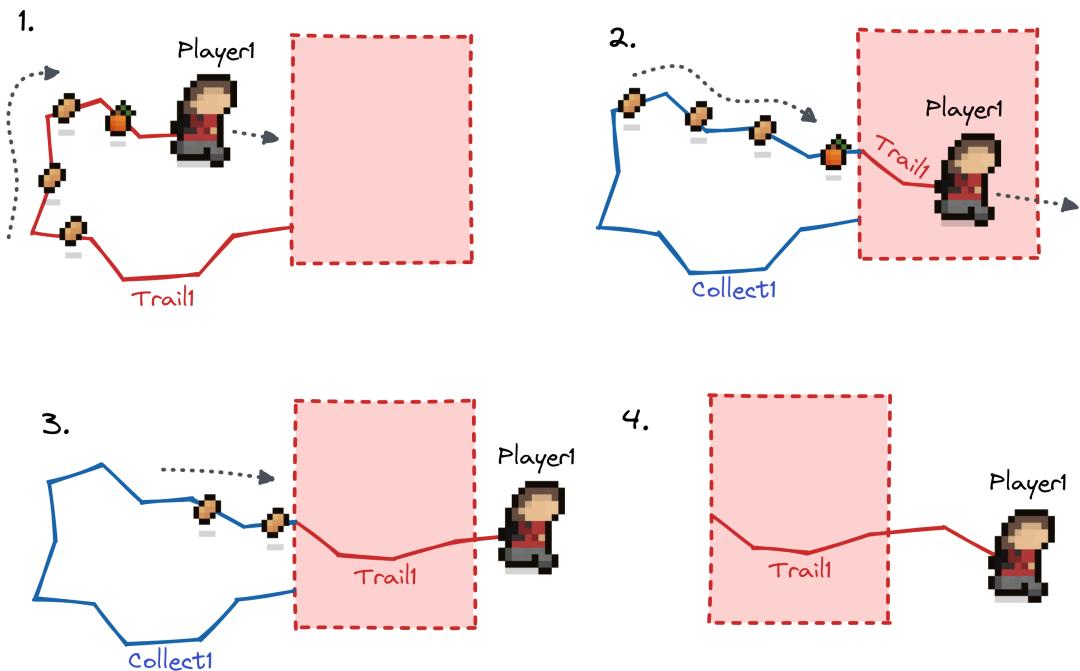


Figure 3.12: Animation de la capture des friandises. Le rectangle rouge représente la base du joueur.

Chapter 4

Tests et résultats

Ce chapitre décrit les différentes fonctionnalités du programme réalisé à la fin du projet et présente quelques améliorations possibles pour un développement futur.

4.1 Résultat

Le programme rendu à la fin de ce projet commence par l'affichage du menu principal visible sur la figure 4.1. Une petite animation se joue avec les personnages à l'écran, puis le joueur est libre de cliquer sur un des deux boutons.

Sur cet écran, le bouton "Paramètres" ne possède aucune action : le menu des paramètres n'a pas été implémenté.



Figure 4.1: Menu d'accueil du jeu. Le bouton "Paramètres" ne fait rien pour l'instant.

Sur la figure 4.2, l'utilisateur est invité à entrer son nom d'utilisateur. Le nom d'utilisateur ne lui est demandé qu'au démarrage du jeu, et il restera enregistré pour toutes les futures parties du jeu.

L'utilisateur aura la possibilité de modifier ce nom d'utilisateur plus tard également.

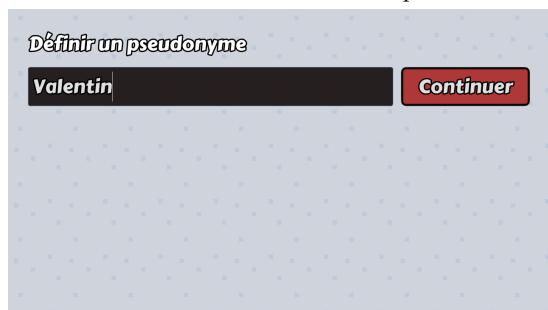


Figure 4.2: Choix du pseudonyme à utiliser.

Sur la figure 4.3, l'utilisateur est invité à choisir son mode de jeu. Pour l'instant, seul le mode "Réseau local" est disponible, permettant de jouer à plusieurs dans le même sous-réseau. Le bouton contenant le nom de l'utilisateur en haut de l'écran permet de revenir à l'écran 4.2 pour modifier son pseudonyme.



Figure 4.3: Menu de choix du mode de jeu.

Lorsque l'utilisateur clique sur le bouton *Réseau local*, il arrive sur le menu 4.4, où il lui est demandé comment compte-t-il rejoindre une partie en locale. Trois choix s'offrent à lui :

1. **Rejoindre un serveur**, permettant au client de se connecter à un serveur créé sur une autre instance ;
2. **Créer un serveur**, lui permettant de partager un serveur à d'autres instances ;
3. **Serveur uniquement**, créant également un serveur mais sans possibilité d'y jouer.

Le bouton retour permet de revenir au menu de la figure 4.3.



Figure 4.4: Menu du mode de jeu "Local".

En cliquant sur le bouton *Rejoindre un serveur*, un popup visible sur la figure 4.5 invite l'utilisateur à entrer l'adresse IP du serveur qu'il souhaite rejoindre.



Figure 4.5: Entrée de l'adresse IP du serveur.

Il est notable que seul le serveur a le bouton *Démarrer la partie*, les clients, eux, ne possèdent que le bouton *Retour* comme montré sur la figure 4.6.



Figure 4.6: Salle d'attente du client.

Dès que l'utilisateur ayant le rôle du serveur clique sur le bouton *Démarrer la partie*, toutes les instances changent de scène pour démarrer le jeu.



Figure 4.7: Salle d'attente du serveur.

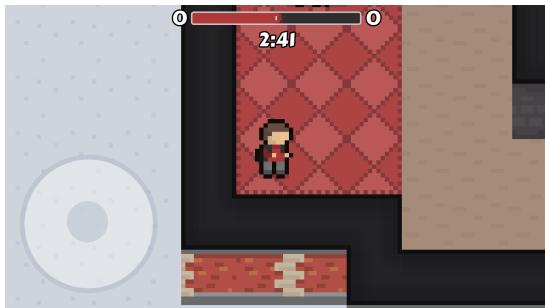


Figure 4.8: Début de la partie pour l'équipe rouge.

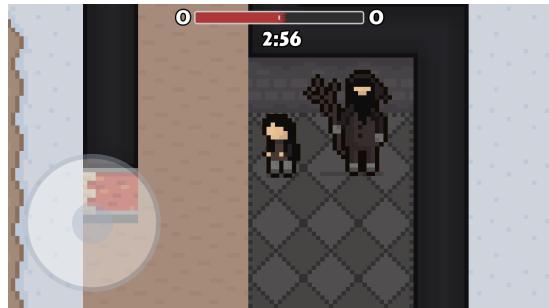


Figure 4.9: Début de la partie pour l'équipe noire.

Lorsqu'un joueur de l'équipe de St-Nicolas ramasse des friandises, elles le suivent à la queue leu leu. Aussi les friandises prennent une texture rougeâtre.



Figure 4.10: Le joueur rouge et sa file de friandises.



Figure 4.11: Le joueur noir et sa file de friandises.

Avoir un changement de couleur lorsque les friandises sont ramassées par les joueurs permettent de voir rapidement si une file de friandise qu'on rencontre sur le jeu est la file d'un joueur ami ou ennemi, comme montré sur la figure 4.13.



Figure 4.12: Les joueurs se rencontrent sur la carte.

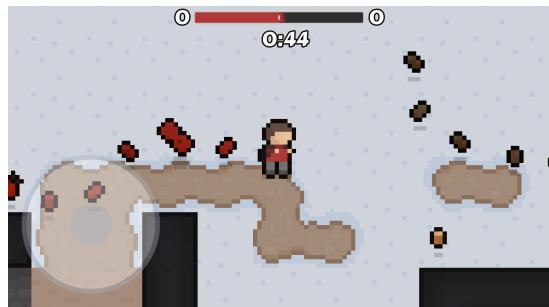


Figure 4.13: Le joueur rouge voit la file de friandises d'un joueur noir.

Finalement, à la fin du temps imparti, la partie se termine et toutes les instances changent de scène pour afficher l'écran de victoire. Si l'équipe de St-Nicolas a gagné, l'écran de la figure 4.14 sera affiché. Si c'est l'équipe de Père Fouettard qui a gagné, c'est l'écran de la figure 4.15 qui sera affiché.



Figure 4.14: L'équipe de St-Nicolas a gagné la partie.
Finalement, le bouton *Retour* ramène l'utilisateur sur l'écran de la figure 4.3.



Figure 4.15: L'équipe de Père Fouettard a gagné la partie.

4.2 Améliorations

Plusieurs améliorations ont été imaginées pour continuer ce projet, et beaucoup d'entre elles peuvent ajouter une dynamique intéressante au jeu :

- **Améliorations du gameplay :**

- Un gain de vitesse temporaire est donné lorsqu'on vole des friandises à un ennemi ;
- Ajouter les modes de jeux multijoueur et solo ;
- Créer une intelligence artificielle pour le monde de jeu solo ;
- Ajouter différents types de friandises spéciales ;
- Donner un rôle à St-Nicolas et Père Fouettard, par exemple si l'équipe de St-Nicolas perds trop face à l'équipe de Père Fouettard, le personnage de St-Nicolas pourrait se déplacer sur le terrain pour aider son équipe ;
- ...

- **Améliorations techniques :**

- Avoir une meilleure gestion des messages pour le réseau, n'en envoyer que lorsque c'est nécessaire ;
- Chiffrer les données sauvegardées sur l'appareil ;
- Ajouter un écran de paramètres avec réglages du volume de la musique.

Chapter 5

Discussion

Cette rubrique propose une synthèse du projet, elle rappelle et mets en évidence les principaux résultats et conclusions.

5.1 Conclusion

Dans ce projet, un jeu vidéo complet a été développé en utilisant le moteur de jeu Godot. Avec l'API multijoueur de haut niveau intégrée au moteur, il a été possible d'intégrer un mode de jeu multijoueur permettant de jouer à plusieurs en connectant différents appareils mobile ensemble.

Dans un premier temps, la phase de conception s'est déroulée avec facilité : comme ce projet reprends globalement les mêmes objectifs que le projet P2 réalisé en fin 2019 et début 2020, le concept du jeu était déjà posé au début de ce projet. La principale différence est donc l'aspect technique : au lieu de développer le projet en C++ avec Qt, c'est avec Godot que le jeu a été construit.

Ensuite, la réalisation du projet s'est déroulée durant les mois d'octobre, novembre et décembre 2021. L'implémentation a commencé par le fonctionnement du multijoueur car c'est un aspect du jeu qui est décisif pour les autres fonctionnalités du projet. Ayant déjà travaillé avec Godot auparavant, la phase d'apprentissage était relativement minime et m'a permis de rapidement avoir un prototype fonctionnel.

Au milieu du mois de décembre, un séminaire d'introduction à Godot a été présenté aux différentes classes d'informatique de 3ème année de la HE-Arc. Sa préparation m'a également permis de réviser les bases du moteur et de mieux comprendre certains concepts fondamentaux qui m'ont été utiles pour le projet SchoolBoyBattle.

L'utilisation d'un téléphone portable et d'une tablette empruntés à la HE-Arc ont permis de tester le programme dans des conditions réelles et de tester la réactivité de l'interface graphique face aux écrans de différentes tailles.

5.2 Perspective

Le concept du jeu étant assez simple dans son exécution, il est tout à fait possible d'y ajouter des fonctionnalités afin de poursuivre son développement.

Un objectif intéressant et atteignable sous quelques mois est de peaufiner le jeu suffisamment afin de le publier sur des magasins d'applications pour téléphone portable, comme le Google Play Store ou l'App Store d'iOS.

References

- [1] Henri Mäkelä. ?“Development of a 3D mahjong video game in Godot Engine”? **in:** (2021).
- [2] Juan Linietsky, Ariel Manzur **and** Godot Community. *Godot Docs 3.2 branch Godot Engine latest documentation*. 2020.
- [3] Richard K Shehady **and** Daniel P Siewiorek. ?A method to automate user interface testing using variable finite state machines? **in:** 1997, **pages** 80–88.
- [4] Paul Adamczyk. ?The anthology of the finite state machine design patterns? **in:** 2003.
- [5] Fei Hu **and** Lixia Ji. ?On the peak-experience in the game GUI design? **in:** 2008, **pages** 204–207. ISBN: 9780769533667. DOI: 10.1109/ICMECG.2008.15.
- [6] Paul Gestwicki. ?A case study approach for teaching design patterns through computer game programming: tutorial presentation? **in:** *Journal of Computing Sciences in Colleges* 24 (1 2008). ISSN: 1937-4771.
- [7] Thorsten Hampel, Thomas Bopp **and** Robert Hinn. ?A peer-to-peer architecture for massive multiplayer online games? **in:** 2006. DOI: 10.1145/1230040.1230058.
- [8] Stefan Fiedler, Michael Wallner **and** Michael Weber. ?A communication architecture for massive multiplayer games? **in:** 2002. DOI: 10.1145/566500.566503.
- [9] Rokas Volkovas **and others**. ?Practical Game Design Tool: State Explorer? **in:** volume 2020-August. 2020. DOI: 10.1109/CoG47356.2020.9231863.
- [10] Ian Millington. *Game Physics Engine Development*. 2010. DOI: 10.1201/b13170.