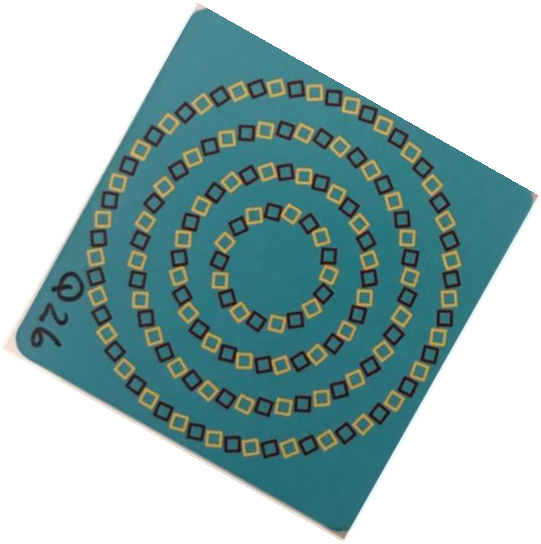
haute école
neuchâtel berne jura
arc ingénierie
saint-imier le locle delémont

Projet autonome N°215

TINY 3D SERIOUS GAME 4 SENIOR



Auteur

Gabriel Griesser

Classe

INF3dlm-a

Présenté à

Senn Julien

Gobron Stéphane

Le 25.01.2019

Abstract

This document describes the implementation of the autonomous project of the 1st semester of the 3rd year. The latter consists in creating an application and following the guidelines of a specification for a given project. The project will take 15 weeks to complete, from **September 18, 2018 to January 25, 2019**. A minimum of 126 hours of work was required. This project, which was partially carried out in class and at home, requires autonomy and regularity in its progress.

Once the project was chosen and assigned, I thought about how to approach the subject. The latter was to create a 3D Serious game for seniors. After a few days of reflection, I looked at a classic game using the Unity software: **Memory**.

The version of this project has been revised. This Serious game is played alone, has different levels/characteristics and allows the user to work on visual and memory perception. The goal is to find all pairs of cards as quickly as possible. The game can be played with 16, 20 or 24 cards. Each level is distinguished by the little difference between its cards, and certain features (music, visual effects, choice of card backs, etc.) improve the immersion of the Serious game.

One of the subtleties of this Serious game is the images used. Indeed, I have chosen 5 series of images whose differences are confusing. As the images chosen for the cards are all carefully selected, the user will need to focus more to identify differences between several cards. He aims to find all pairs of cards in a minimum amount of time. At the end of the game, a score will be calculated according to the time used.

This document will describe, inter alia, the progress, design of the project and implementation of the program. But also the description of the main classes, problems encountered and results obtained and the use of the software.

Table des matières

1. INTRODUCTION 1

2. ANALYSE 2

3. CONCEPTION 3

3.1. CAHIER DES CHARGES ET SPÉCIFICATIONS DU PROJET 3
3.2. DIAGRAMME DE GANTT 3
3.3. MODÉLISATION 5
3.3.1. Diagramme UML 5
..... 6
3.3.2. Convention de codage 7
3.3.3. Use Case 7
3.3.4. Déroulement du jeu 8

4. IMPLÉMENTATION 9

4.1. LOGICIEL 9
4.1.1. Unity 9
4.2. TABLEAU DE JEU ET CARTE 10
4.3. GÉNÉRATION DU BOARD 12
4.4. LOGIQUE DU JEU 13
4.5. NIVEAUX 16
4.6. MENU 19
4.6.1. Boutons 19
4.6.2. Menu d'accueil 20
4.6.3. Sélection du niveau 21
4.6.4. Sélection du nombre de cartes 21
4.6.5. Options 22
4.7. CLASSE STATIQUE 23
4.8. EFFET FLIP ET COROUTINES 23
4.9. MUSIQUE ET EFFETS SONORE 24
4.10. RÉSULTATS 25

5. PROBLÈMES RENCONTRÉS 26

5.1. GÉNÉRATION DU BOARD 26
5.2. EFFET FLIP 27
5.3. DÉSACTIVATION DU CLIC SUR LES CARTES QUAND ELLES SE RETOURNENT 27
5.4. *BUILD* DU PROJET 29

6. AMÉLIORATIONS FUTURES 30

7. CONCLUSION 31

8. SOURCES 31

9. ANNEXES 32

1. Introduction

Pendant le semestre d'automne de la 3ème année, les étudiants de la HE-ARC doivent choisir un projet parmi une liste donnée, rédiger un cahier des charges conforme aux objectifs du projet et développer l'application choisie. Si l'application n'est pas imposée, le langage, l'objectif final et certains critères le sont.

L'objectif de mon projet est le suivant : **Réaliser un (tout) petit Serious game pour senior sous Unity avec des objets 3D.**

Je me suis donc orienté vers la réalisation d'un **Serious game 3D** développé en **C#** en utilisant le moteur de jeu **Unity**. Ce Serious game reprend le principe du célèbre jeu *Memory*, mélangeant mémoire et perception visuelle.

Le Memory est un jeu de société composé de paires de cartes portant des illustrations identiques. Ces cartes sont mélangées puis déposées face cachée. A tour de rôle, chaque joueur retourne 2 cartes de son choix. Si ces dernières sont identiques, il les ramasse, les conserve et a la possibilité de rejouer. Si les cartes ne sont pas identiques, il les retourne faces cachées. Quand toutes les cartes ont été trouvées, le joueur ayant le plus de paires de carte gagne la partie.

Dans mon cas, le concept du jeu a été revisité afin de remplir la condition concernant le public visé. Il possèdera entre autres 5 niveaux, chacun avec 16, 20 ou 24 cartes.

Le délai pour ce projet est le 25 janvier 2019. Nous avons donc en tout 15 semaines, chacun comptabilisant environ 3 heures de travail en salle (horaire fixe) et environ 5 heures et demi chez soi.

2. Analyse

Parmi les nombreuses possibilités offertes par le monde de l'informatique en infographie, les directives du projet demandent l'utilisation du moteur graphique Unity, ce dernier imposant le langage de programmation C#.

Avant de se lancer dans le projet, j'ai réfléchi à la meilleure manière d'aborder le sujet, d'atteindre le public ciblé (senior) avec un Serious game développé sous Unity. L'idée du *Memory* m'est venue 1 semaine après la date de début de projet. Ce jeu, en une version revisitée faisant travailler à la fois la mémoire et la perception visuelle, correspondait parfaitement aux attentes : cibler un public senior via un Serious game.

Après avoir réfléchi sur comment implémenter ce projet et comment atteindre l'objectif sans se perdre en chemin, j'ai pu rédiger le cahier des charges et les spécifications figurant en annexes pour avoir l'idée optimale pour le déroulement du projet. Ces derniers documents décrivent entre autres la façon dont le Serious game verra le jour, les objectifs principaux et secondaires et les différents points chauds.

Avant d'attaquer le projet, j'ai défini une convention de codage et réfléchi sur la meilleure manière d'aborder la conception.

Finalement, j'ai dessiné les différents schémas nécessaires pour une meilleure réussite du projet. Ces schémas, qui sont le diagramme UML, le *Use case* et le diagramme de Gantt, ont bien entendu évolués tout au long du projet. Ils m'ont néanmoins permis d'y voir plus clair et donc de ne pas me perdre en cours de route.

Figure 1: Logo Unity

3. Conception

Cette partie est consacrée à la conception du projet. Premièrement, j'ai rédigé le cahier des charges et les spécifications du projet. Ensuite, après avoir défini une première liste de tâche nécessaires au projet, j'ai modélisé ce dernier sous forme de diagramme UML afin d'avoir une meilleure vision de l'implémentation.

3.1. Cahier des charges et spécifications du projet

Le cahier des charges défini clairement les objectifs à atteindre. Ces derniers sont détaillés dans la spécification du projet. Ces deux documents m'ont permis d'obtenir une meilleure définition du but, des objectifs et des détails concernant le projet.

Dans le cahier des charges, j'ai défini les objectifs principaux, secondaires, le but du projet, la description de ce dernier et la méthode de codage. Il servira à quiconque le lit, d'avoir une idée globale du projet et des objectifs à atteindre. J'ai pu y intégrer les objectifs principaux accompagnés de quelques objectifs optionnels.

Dans les spécifications du projet, j'ai rédigé plus en détail le déroulement de l'application. Cela me permet de faire un effet « entonnoir » dans la conception. Le cahier des charges permettant d'avoir une idée du projet final dans les grandes lignes, ce 2ème document, quant à lui, décrit avec un maximum de précision le déroulement de la conception. J'y ai inclus notamment les spécificités du projet, le déroulement d'une partie et les différents objectifs accompagné de leurs points chauds. Cela m'a permis d'attaquer la suite avec un maximum de compréhension.

3.2. Diagramme de Gantt

Le diagramme de Gantt est une manière de décrire, d'attribuer et de visualiser les différentes tâches nécessaires au bon déroulement au projet. La 1ère version de ce planning a été réalisé la 2ème semaine du projet. Il a donc évolué pendant ces 15 semaines pour donner cette version ci-dessous, certaines tâches ne correspondent pas au temps donné (marge d'erreur pendant la réalisation du projet, certaines tâches me demandaient plus de temps et d'autres moins) pendant que d'autres le sont. La liste complète des tâches a été réalisée dans le temps imparti.

	Plan...	Nom de tâche	Durée	Début	Fin
1		**Memory classique**	**51 jours**	**25.09.2018**	**04.12.2018**
2		**Cartes**	**51 jours**	**25.09.2018**	**04.12.2018**
3		Récupération images carte standard (heures) + dos...	6 jours	25.09.2018	02.10.2018
4		Création des cartes	11 jours	02.10.2018	16.10.2018
5		Mélange des images pour chaque partie (effet aléat...	7 jours	06.11.2018	14.11.2018
6		Gestion cartes (clic, retournement, etc.)	16 jours	13.11.2018	04.12.2018
7		**Plateau de jeu**	**32 jours**	**25.09.2018**	**07.11.2018**
8		Image plateau de jeu	5 jours	25.09.2018	01.10.2018
9		Création 3D du plateau	6 jours	02.10.2018	09.10.2018
10		Mise en place des cartes sur le plateau	16 jours	02.10.2018	23.10.2018
11		Positionnement des cartes 4x4, 5x4, 6x5 (statique !)	22 jours	09.10.2018	07.11.2018
12		**Logique du jeu**	**36 jours**	**16.10.2018**	**04.12.2018**
13		Indexage cartes	11 jours	30.10.2018	13.11.2018
14		Comparaison des cartes (par ID)	11 jours	16.10.2018	30.10.2018
15		Ajout système de temps	15 jours	14.11.2018	04.12.2018
16		Ajout système de score	15 jours	14.11.2018	04.12.2018
17		Check fin de jeu	11 jours	20.11.2018	04.12.2018
18		**Memory spécial**	**30 jours**	**04.12.2018**	**14.01.2019**
19		**Menu**	**9 jours**	**04.12.2018**	**14.12.2018**
20		Menu pour jouer (sélection nombre cartes, niveau)	9 jours	04.12.2018	14.12.2018
21		Menu options (Dos de carte, volume, about)	9 jours	04.12.2018	14.12.2018
22		Menu fin de jeu (sauvegarde, nouvelle partie, quitter)	9 jours	04.12.2018	14.12.2018
23		Récupération/Création images spéciales (scan, Ishiha...	6 jours	14.12.2018	21.12.2018
24		Ajout de 2 niveaux (Scan des images de Mr. Senn)	10 jours	17.12.2018	28.12.2018
25		Peaufinement, amélioration board/cartes	21 jours	17.12.2018	14.01.2019
26		Ajout de musiques et bruitages	9 jours	01.01.2019	11.01.2019
27		**Améliorations**	**41 jours**	**27.11.2018**	**22.01.2019**
28		Ajout 3ème niveau (Ishihara)	5 jours	25.12.2018	31.12.2018
29		Option changement dos de carte	10 jours	14.12.2018	27.12.2018
30		Effet flip	27 jours	27.11.2018	02.01.2019
31		Sauvegarde	8 jours	11.01.2019	22.01.2019
32		**Annexe**	**86 jours**	**25.09.2018**	**22.01.2019**
33		Réalisation rapport	56 jours	06.11.2018	22.01.2019
34		Spécification du projet	4 jours	25.09.2018	28.09.2018

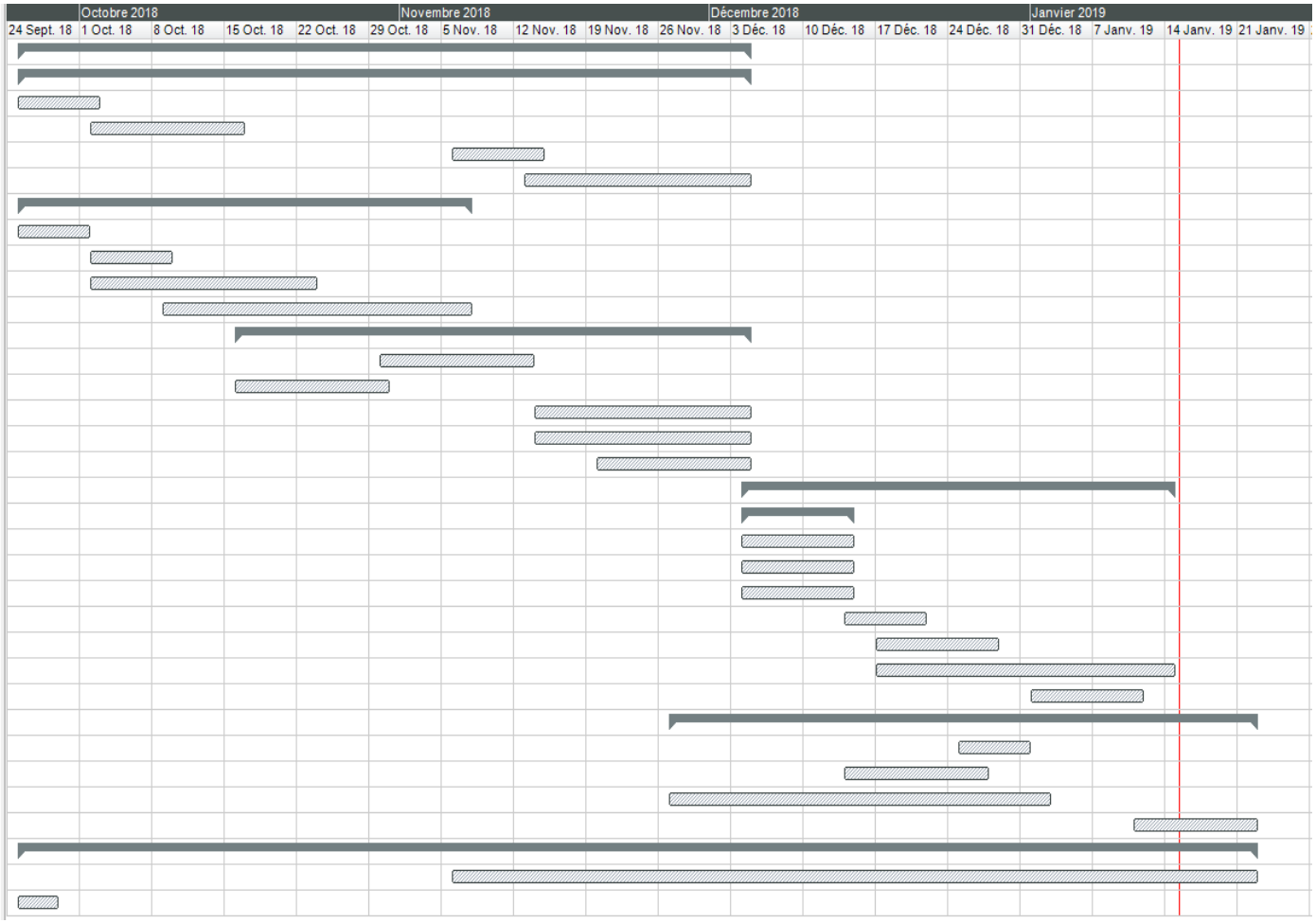


Figure 2 : Diagramme de Gantt

3.3. Modélisation

3.3.1. Diagramme UML

Le diagramme UML, bien que très simple, m'a permis de poser sur papier ma représentation du projet. Chaque modification, ajout de classe ou suppression de composant était reporté sur mon diagramme. Ce dernier est très simplifié. En effet, il est difficile de représenter toutes les liaisons multiples entre objets, composants et scripts de Unity sur un simple diagramme. J'ai donc décidé d'y afficher uniquement les classes, méthodes, propriétés et attributs utilisés. Certaines classes héritent de **MonoBehaviour** étant donné leurs rôles. D'autres, qui ne sont pas reliés à un GameObject, sont de simples classes du langage C#.

Le diagramme est divisé en 3 parties, respectivement les 3 scènes du jeu. Le menu, le jeu et la scène de fin.

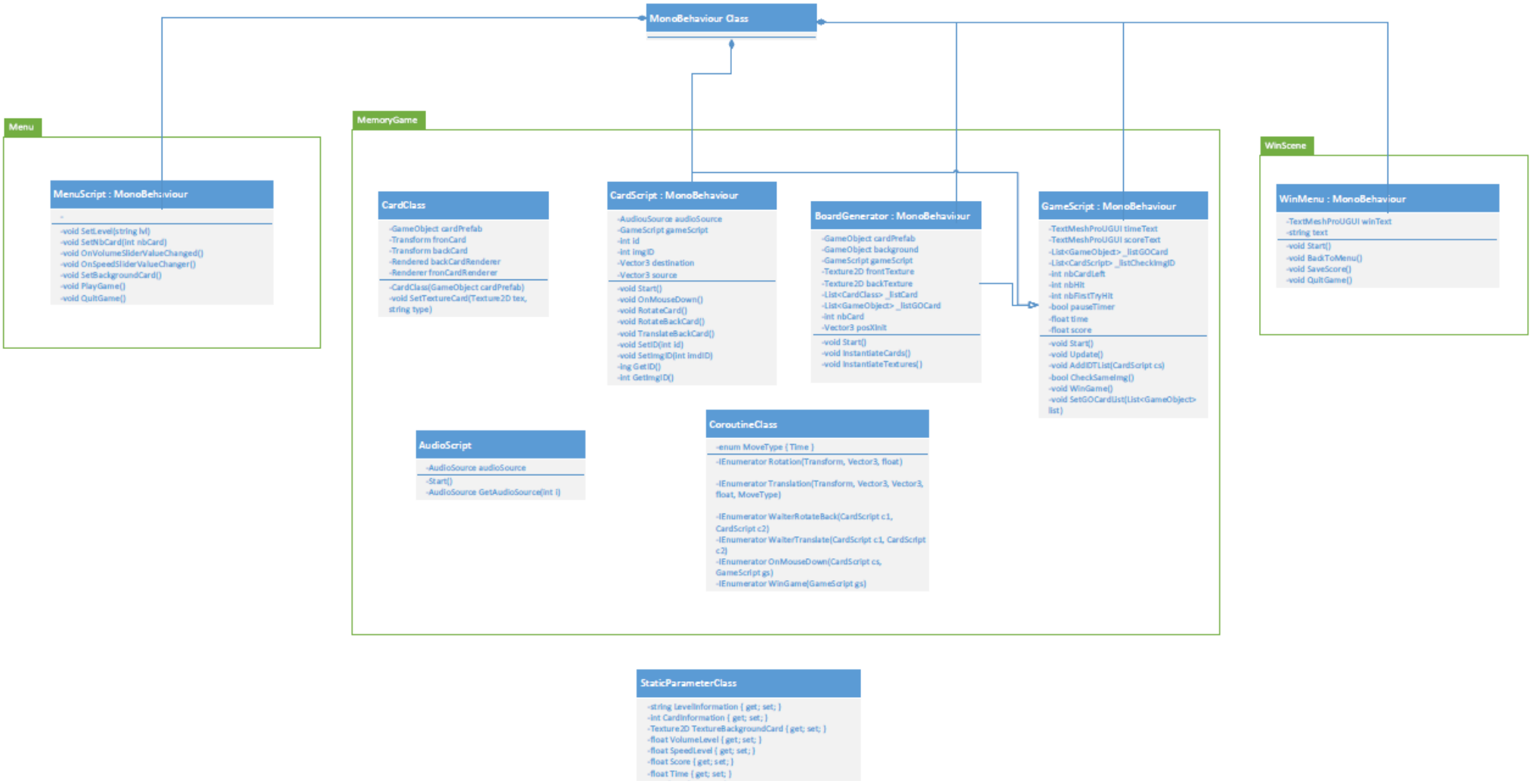


Figure 3: Diagramme UML

3.3.2. Convention de codage

- Les noms des classes sont en anglais, commencent par une majuscule et peuvent contenir une ou plusieurs majuscule → *BoardGenerator*
- Les noms des attributs et objets commencent par une minuscule et peuvent contenir une majuscule. Ils sont écrits en anglais → *cardPrefab*
- Les noms des *Lists* commencent par un tiret en bas puis une minuscule et peuvent contenir une majuscule. Ils sont écrits en anglais → *_listGOCard*
- Les noms des méthodes commencent par une majuscule et peuvent contenir une ou plusieurs majuscules. Ils sont écrits en anglais → *InstantiateCards()*

3.3.3. Use Case

Le *use case*, ou diagramme de cas d'utilisation, défini la manière d'utiliser le programme et les différentes actions possibles. Nous remarquons ici une certaine linéarité. Effectivement, le programme n'a qu'une seule façon d'être joué. Chaque partie renvoi le joueur au même processus final.

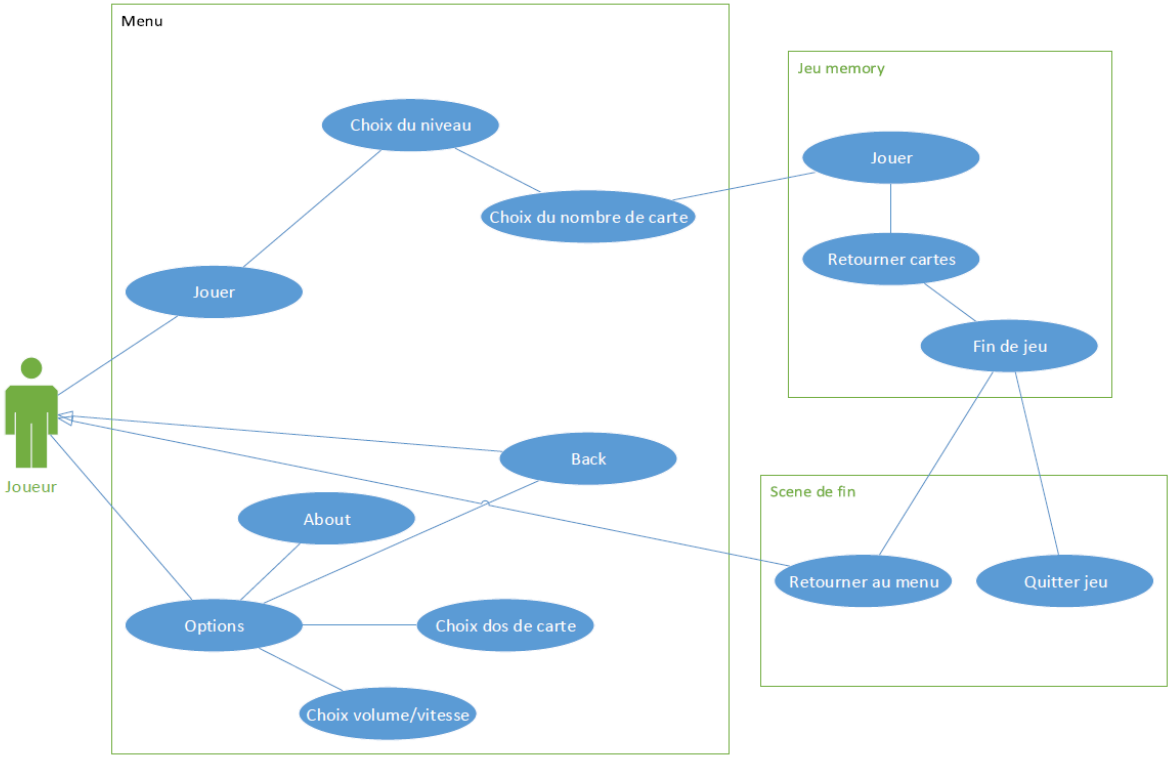


Figure 4: Cas d'utilisation

3.3.4. Déroulement du jeu

L'utilisation de ce Serious game est assez basique. Le jeu est en effet linéaire et déroule comme cela (certains processus ne sont pas visibles par le joueur).

- Démarrage du jeu, passage par le menu
- **Menu**
  - Choix des options → Sauvegarde des valeurs dans la classe statique
    - Changement dos de cartes
    - Modification du volume
    - Modification du temps de retournement des cartes
  - Sélection du niveau → Sauvegarde dans classe statique
  - Sélection du nombre de cartes → Sauvegarde dans classe statique
  - Jouer
- **Jeu**
  - Récupération des valeurs de la classe statique
  - Création des cartes
    - Position en X
    - Position en Y
    - Instanciation des textures d'images et dos de carte
    - Placement sur le board
  - Déroulement du jeu
    - Clic sur une carte
      - Retournement de la carte
      - Si c'est la 1ère, ne rien faire
      - Si c'est la 2ème, check des images
        - Si c'est les mêmes images, replacement des cartes faces visibles et incrémentation du score de 100
        - Sinon, replacement face cachée
    - Temps incrémenté en boucle
    - Check du nombre de carte en boucle
      - Si = 0, alors fin de jeu
        - Stop timer
        - Sauvegarde temps, score, nombre de coups
        - Passage à la scène de fin
- **Fin du jeu**
  - Affichage du temps, du nombre de coup total et du score final
  - Retourner au menu d'accueil
  - Quitter le jeu

4. Implémentation

Cette partie du rapport décrit l'implémentation du projet. Le logiciel utilisé, la façon d'atteindre le but principal du Serious game, puis les différentes parties du programme, leurs spécificités et le rôle de chacune seront détaillés. Chaque partie du jeu a été implémentée, modifiée et enfin améliorée et finalisée. Certaines sections ne respectent donc pas la même chronologie que le développement lui-même.

4.1. Logiciel

4.1.1. Unity

Unity est un moteur de jeu multiplateforme. Le logiciel a la particularité d'utiliser un éditeur de script compatible mono (C#). Unity, couplé au langage de programmation C# permet de développer des applications graphiques. Il permet de créer des scènes supportant l'éclairage, des textures, caméras, musique et vidéos. Ce moteur graphique est répandu dans l'univers des jeux-vidéos avec notamment le célèbre *Hearthstone : Heroes of Warcraft* ou encore *Pokemon GO*.

Avant d'entamer la partie suivante, il est préférable d'expliquer en quelques lignes les principes de développement en Unity.

- Tout objet (cube, lumière, canvas, personnage, etc.) créé et instancié depuis Unity s'appelle un **GameObject**.
- Ces *GameObject* possèdent des **composants** (*colliders*, position, source audio, script, image etc.) qui permettent d'implémenter la réelle fonctionnalité de ce dernier.
- Les **scripts** attachés au *GameObject* écrivent vos propres ajouts de fonctionnalités de l'éditeur Unity. Ils permettent entre autres la gestion complète du *GameObject* auquel il est rattaché.
- Les scripts sont codés en **C#** et sont dérivé de la classe **MonoBehaviour**. Tout script Unity touchant directement aux *GameObject* doit hériter de cette classe.
- Les *GameObject* peuvent être « sauvegardé » comme **prefab**.
- Un **prefab** est un système de Unity qui permet de créer, configurer et stocker un *GameObject* complet avec tous ses composants, ses valeurs et ses propriétés. Cela assure une duplication parfaite du *GameObject* et permet d'éviter de créer 2 fois le même *GameObject* pour un projet.
- Les **coroutines** permettent de mettre en pause une exécution du code, effectuer une action décrite puis poursuivre la lecture du code là où cette dernière avait été mise en pause.

Nous voilà avec une base nécessaire pour la compréhension de la suite de cette section.

4.2. Tableau de jeu et carte

Ce projet a été développé en plusieurs parties. La première consistait à développer les GameObject suivants : le **background** et la **card**. Ils correspondent respectivement au tableau de jeu et à une carte.

Pour le background, c'est un simple GameObject de type *Cube* sur lequel une image a été appliquée. La modification de l'élément rotation de son composant *Transform* ajoute un effet 3D faisant penser à une table sur laquelle on joue.

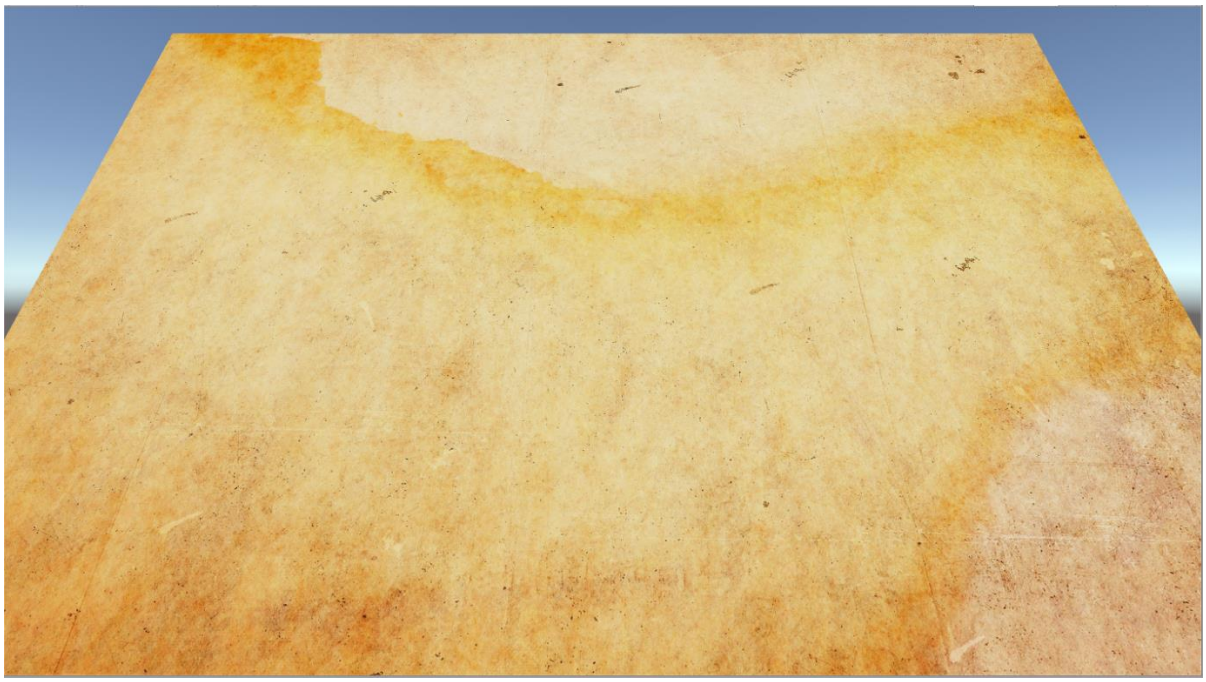


Figure 5: Tableau de jeu vide

La carte quant à elle est composé de 2 *Cube*. Le cube principal (*Front*) et le dos de carte (*Back).* Couplés et redimensionnés, ces 2 GameObject offrent un objet carré représentant une carte dont chaque côté correspondra à une image. Le côté visible, le *front*, affichera le l'image de carte parallèlement à l'autre côté qui affichera le dos de la carte.

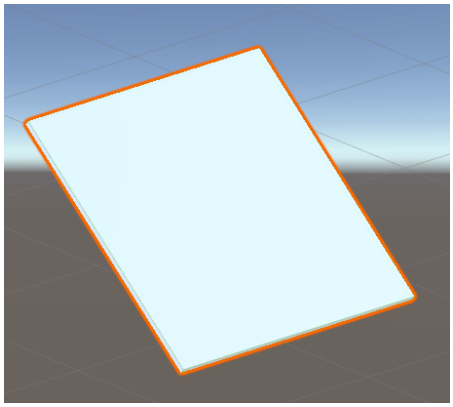


Figure 6: GameObject card

Une fois cette carte créée, il a fallu lui ajouter un script : **CardScript.cs**. En effet, la carte sera soumise à différentes opérations et modification durant le déroulement du jeu. Ce script gère donc toute action touchant directement à la carte. Voici sa composition principale :

- Attributs
  - **id** → id de la carte.
  - **imgID** → id de l'image liée à la carte. Les imgID varient entre 8 et 12, respectivement correspondant au nombre de carte total divisé par 2 (16, 20 ou 24 maximum).
- Méthodes
  - ***Start()***
    est une méthode d'initialisation. Elle est appelée à chaque initialisation du GameObject auquel est rattaché le script.
  - ***OnMouseDown()***
    Gère le clic du bouton de la souris sur une carte.
  - ***RotateCard()***, ***RotateBackCard()***, et ***TranslateBackCard()***
    Ces fonctions sont utilisées pour les effets *flip* de la carte. A chaque clic, la carte se retournera tout en se rapprochant du joueur pour qu'il puisse mieux distinguer l'image se trouvant dessus. Également après le test de similarité entre les 2 cartes, ces dernières doivent soit être replacées faces visibles sur le board, soit être retournées et replacées faces cachées. C'est ces fonctions, appelant des *coroutine* qui gèrent ces différents effets. L'effet *flip* a été implémenté à la fin du projet, par conséquent, ces fonctions aussi.

Ce script, attaché à la carte, permet donc de finaliser le GameObject pour en faire un prefab. En effet, le cahier des charges stipule une utilisation minimale de 16 cartes, il a fallu une base qui pourrait être réutilisable dans n'importe quel cas. C'est ici que le prefab intervient.

Une dernière classe, non MonoBehaviour a été créée pour le bon fonctionnement de la carte. La classe **CardClass.cs**. Cette dernière va permettre l'accès au GameObject *card* notamment par la modification de ses textures depuis l'extérieur. Voici les méthodes la composant :

- ***CardClass(GameObject cardPrefab)***
  Constructeur de la classe. Chaque carte aura sa propre instance de classe *CardClass*. Pour que la liaison soit faite, je passe le prefab de ma carte en paramètre du constructeur.
- ***SetTextureCard(Texture2D tex, string type)***
  Placer une texture passée en paramètre sur un côté (avant ou arrière) de la carte. La partie avant correspond à l'image de la carte, la partie arrière au dos de carte.

Cette classe sera utile pour la section suivante. Je possède maintenant un plateau et un prefab de carte tout deux implémentés.

4.3. Génération du Board

Que faire avec un tableau de jeu et des cartes vides ? Finaliser ces dernières et les placer faces cachées sur le board bien entendu. C'est ici la deuxième partie de mon développement. Pour y parvenir, j'ai créé un GameObject vide et invisible qui contiendra les scripts nécessaires au jeu : le **GameController**

Une fois mon tableau de jeu et mon prefab de carte créés, il a fallu placer les images sur les cartes et positionner ces dernières sur le background. C'est ici le rôle du script **BoardGenerator.cs.** Cette classe est rattachée au **GameController** décrit ci-dessus. Elle permettra notamment l'instanciation et le placement de toutes les cartes sur le board, mais aussi la gestion des textures et le mélange des cartes à chaque partie. La classe **BoardGenerator** est composée des éléments principaux suivants :

- Attributs
  - **cardPrefab** et **background**
    GameObject représentant respectivement la carte/le tableau de jeu
  - **gameScript**
    Lien direct vers le script *GameScript*. Ce dernier sera détaillé plus tard.
  - **_listCard** et **_listGOCard**
    Liste contenant respectivement chaque carte représentée par la classe *CardClass* et chaque carte représentée par son GameObject.
  - **postXInit**
    Position initiale, statique. En haut à gauche du board.
- Méthodes
  - ***Start()***
    Comme pour la section précédente, cette méthode sert de point de démarrage à la classe. C'est la première à être appelée.
  - ***InstanciateCards()***
    C'est ici que les cartes sont instanciées. Tant au niveau GameObject que script (*CardClass*). Les cartes sont placées faces cachées sur le background, à leur position initiale, donnée statiquement. Je rempli également les listes avec les objets créés dans cette fonction. Finalement, je passe la liste des GameObject à ma classe de jeu, *GameScript*.
  - ***InstantiateTexture()***
    Instanciation des textures. Chaque texture correspond à un numéro entre 1 le nombre de carte. Je place ensuite 2 fois chaque numéro dans ma liste *_listImg*. Je vais ensuite mélanger cette dernière grâce à la fonction ***ShuffleList(List<int> list)***.
    Une fois fait, je possède une liste d'entier, de 1 à *nbCard*, dont chacun est rentré 2 fois, mélangée aléatoirement. Il ne me reste plus qu'à piocher les images (numérotées de **1.png** à **12.png**) en parcourant la liste et à appliquer ces dernières en tant que texture via la méthode *SetTextureCard()* de la classe *CardClass*.

Pour le dos de carte, la fonction check si un dos de carte a été choisi dans le menu. Si c'est le cas, elle l'appliquera à chaque carte de la même méthode que pour les images.

- ***PositionColCard()*** et ***PositionRowCard(int nbRow, Vector3 updPosY)***
  Ce sont ces 2 méthodes qui vont positionner les cartes. Ces dernières étant toutes instanciées au même endroit dans la méthode *InstanciateCards().*
  PositionColCard() permettra de créer 4 colonnes de cartes. En fonction du nombre de cartes, elle attribuera un nombre de ligne et un espacement statique qui seront passés en paramètre à la méthode suivante.
  PositionRowCard() espacera chaque ligne de cartes par *updPosY* passé en paramètre.

Figure 8 : Placement des cartes avec la fonction PositionColCard()

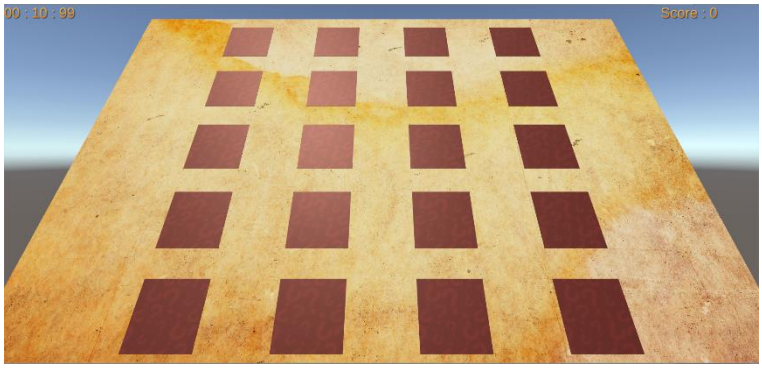


Figure 7 : Placement des cartes avec la fonction PositionColCard() et PositionRowCard()

Pour résumer, ce **BoardGenerator** permet de créer tout l'aspect visuel et graphique du jeu. Il va créer chaque carte une par une, leur attribuer certaines propriétés, placer les textures dessus puis les poser sur le board en fonction du nombre de carte. Il ne reste plus que la logique du jeu pour avoir une version jouable. C'est donc le point suivant de ce rapport.

4.4. Logique du jeu

Pour implémenter la logique du jeu, il me fallait un script, similaire au **BoardGenerator**, c'est-à-dire rattaché à aucun GameObject visible. J'ai donc créé le script **GameScript.cs**, rattaché au même GameObject que la classe de génération de board. C'est donc dans ce GameObject **GameController** que sera également géré la logique du jeu. Cette logique implémente plusieurs choses.

1. Le check des cartes à chaque retournement d'une paire de carte.
2. La mise à jour du temps et score contenus dans un *canvas.*
3. Le *win condition*.

Comme décrit, c'est ici la classe principale du jeu. Elle est composée d'une méthode appelée *Update().* Cette méthode, qui est la plus employée pour tout script implémenté, est lancée à chaque *frame* du script si ce dernier est MonoBehaviour. Ci-dessous est détaillé la composition principale du script implémentant la logique du jeu, le **GameScript.**

- Attributs
  - **timeText** et **scoreText**
    GameObject relié directement aux 2 éléments textuels du canvas, respectivement le temps de jeu et le score du joueur.
  - **AudioSource**
    Composant utilisé pour jouer un son si les 2 cartes retournées sont les mêmes. Idem, avec un son différent, si les 2 cartes ne sont pas identiques.
  - **_listGOCard** et **_listCheckImgID**
    La première est la liste de cartes, passé en paramètre depuis le script BoardGenerator. Elle permettra la suppression de chacune en cas de fin de jeu.
    La deuxième liste contient les scripts des 2 cartes retournées. Elle a donc une capacité maximale de 2 cartes.
  - **nbCardLeft, nbHit, nbFirstTryHit, pauseTimer, time and score**
    Divers attributs nécessaires au bon déroulement du script. Leur nomination est en adéquation avec leur rôle. La différence entre *nbFirstTryHit* et *nbHit* repose sur le fait que la première variable est incrémentée uniquement quand 2 cartes retournées sont les mêmes. Contrairement à *nbHit* qui est incrémenté de 1 chaque fois que 2 cartes sont testées, indépendamment du résultat du test.
- Méthodes
  - ***Start()***
    Comme pour la section précédente, cette méthode sert de point de démarrage à la classe. C'est la première à être appelée. Elle va initialiser les variables de la classe.
  - ***Update()***
    Cette fonction est la plus répandue dans l'implémentation des scripts. Si le script est *MonoBehaviour*, et donc qu'il est relié à un GameObject, *Update()* sera appelée à chaque frame.
    C'est donc ici que j'effectue mes tests. A chaque frame, je check et, si besoin, met à jour le score. Idem pour le temps de jeu, ce timer *time* est incrémenté puis affiché dans le canvas
    J'y fait également le check du nombre de carte restant. Si ce dernier est à 0, alors je stoppe le timer et lance le processus de fin de partie.
  - ***AddIDToList(CardScript cs)***
    Cette méthode est utilisée pour gérer la carte sur laquelle le joueur a cliqué Cette dernière est passée en paramètre par le biais de son script, *CardScript*. *AddIDToList()* possède un *switch()* qui va tester le nombre de cartes dans la liste *_listCheckImgID*. 3 cas sont possibles :
    - La carte est la 1ère des 2. Je l'ajoute donc à la liste et sort de la boucle.
    - La carte est la 2ème carte sur laquelle on clique. Je l'ajoute à la liste et lance le traitement de reconnaissance d'ID via la méthode *CheckSameImg()*. Quelle que soit la réponse de cette dernière, je vide la liste pour permettre un nouveau remplissage.

- Pour des raisons de sureté, j'ai également checké si la liste contenait déjà 2 images. Dans ce cas, elle est vidée pour accueillir la nouvelle carte sélectionnée.

- ***CheckSameImg()***

  Ici, je récupère l'ID de l'image des 2 cartes contenues dans la liste *_listCheckImgID*. Si cet ID est le même, le *nbFirstTryHit* augmente de 1. Un son *FlipWin* est joué et la fonction retourne *true*. Cela signifie que les images des 2 cartes sont identiques. Dans le cas contraire, un son *FlipLose* est joué et la fonction retourne *false.*

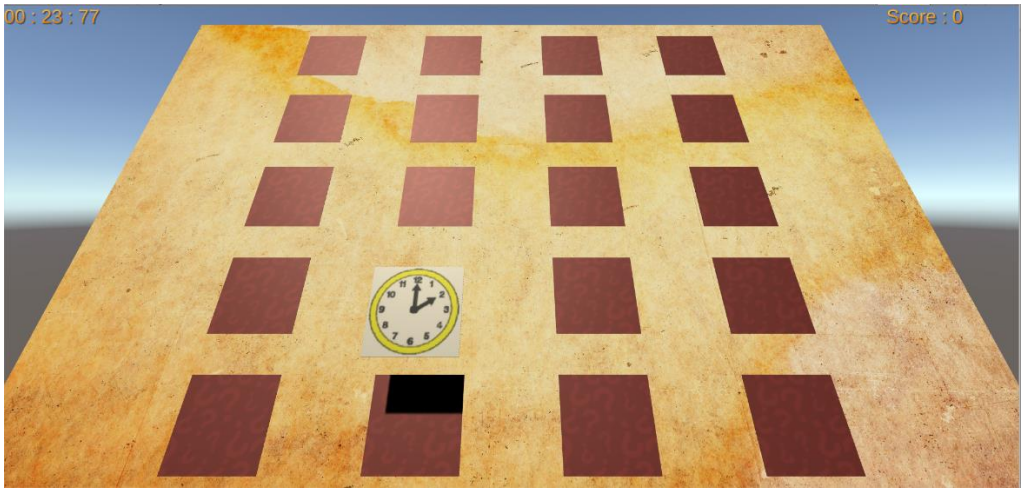


Figure 9: Une carte est retournée

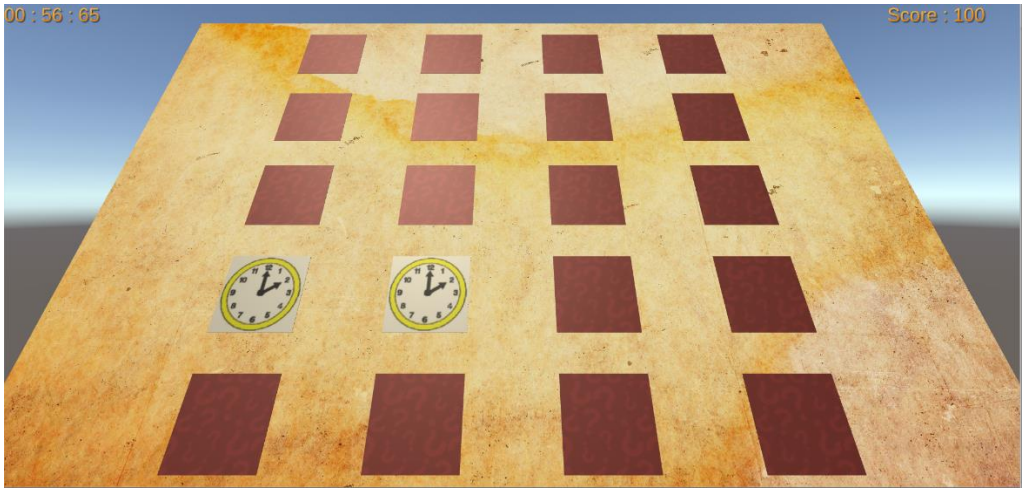


Figure 10: La 2ème carte retournée possède la même image. CheckSameImgID() return true et le score s'incrémente de 100.

- ***WinGame()***

  Méthode de fin de jeu. Sauvegarde les données dans une classe statique (détaillée plus tard), puis charge la scène *WinScene*. Cette scène, créée bien plus tard dans le projet, est différente de notre scène actuelle. En effet, elle ne contiendra pas les cartes faces cachées mais les informations sur le jeu, le nombre de coup total (nous voyons ici l'utilité de la variable *nbHit*), et le score final. Ce score est calculé en fonction du temps. Plus le temps utilisé

sera court, plus haut sera le score final. Une possibilité de revenir au menu est également affichée.

Figure 11: Scène de fin de partie

L'explication sur la logique du jeu prend fin ici. Cette classe est en quelque sorte la classe mère. C'est elle qui permettra les différentes actions en fonction du choix de l'utilisateur, gérant à la fois le temps, le score, le test des 2 cartes retournées et la fin du jeu. Jusqu'ici, mon Serious game était encore très *light*. Je ne possédais pas de menu principal, l'effet *flip* n'était pas implémenté, le son non plus et la réponse à une fin de jeu s'affichait dans la console. Plusieurs améliorations graphiques ont donc vu le jour. Effectivement, je ne possédais qu'une seule scène, la *MemoryGame*. J'ai donc mis en place premièrement mon menu d'accueil, suivi de mon menu de fin de jeu. A cela s'est ajouté le son puis l'effet flip, venu tout à la fin.

4.5. Niveaux

Les niveaux sont, pour ce Serious game, considéré comme des images. Elles doivent être numérotées de **1.png** à **12.png**. Le chargement, via la méthode *InstanciateTexture()* de la classe BoardGenerator se fait comme cela :

- Chaque numéro de 1 à 12 est rentré 2 fois dans la liste *_listImg.*
- La liste est mélangée aléatoirement grâce à la méthode *ShuffleList()*.
- Dans une boucle, le programme va parcourir la liste et charger comme une texture le fichier ***Resources/CardTexture/PATHTOLEVEL/_listImg[i]***.
- Il positionnera ensuite une texture sur une carte puis continuera le parcours de la boucle jusqu'à ce que le nombre de card *nbCard* soit atteint.

Par exemple, pour charger la carte 4 du niveau *Hour,* le chemin sera le suivant : ***Resources/CardTexture/CardHour/4***

J'ai commencé le développement avec des images représentant des heures sur un site libre de droit.

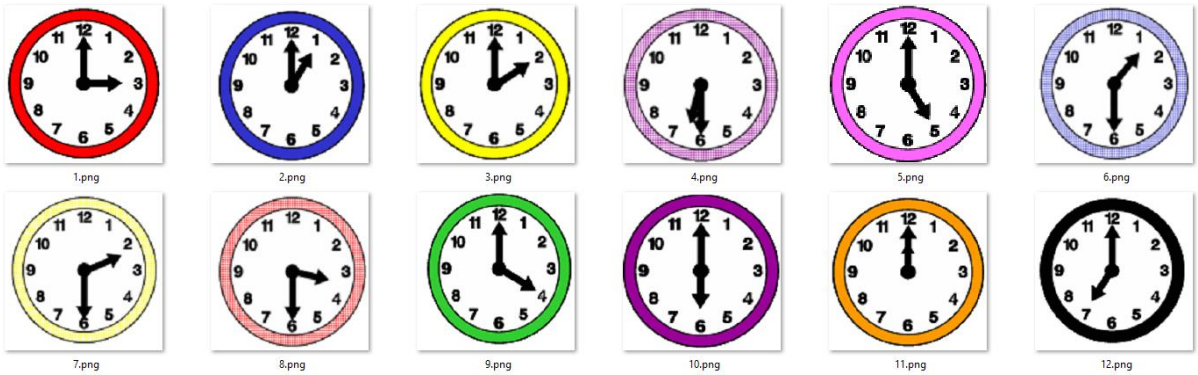


Figure 12: Niveau 1, « Hour »

Le 2ème et 3ème niveau ont été rajoutés une fois le jeu en version alpha. A ce stade, je suis capable de jouer une partie avec 16, 20 ou 24 cartes (variable à changer en brute dans le code) sur un seul niveau.

Mon expert, Mr Julien Senn, m'a apporté un jeu de carte venant de la ludothèque de la Chaux-de-Fonds. Ces cartes possèdent toutes des images qui pouvaient me servir de niveau dans mon jeu. J'ai donc scanné 24 de ces cartes pour les séparer en 2 niveaux distincts.

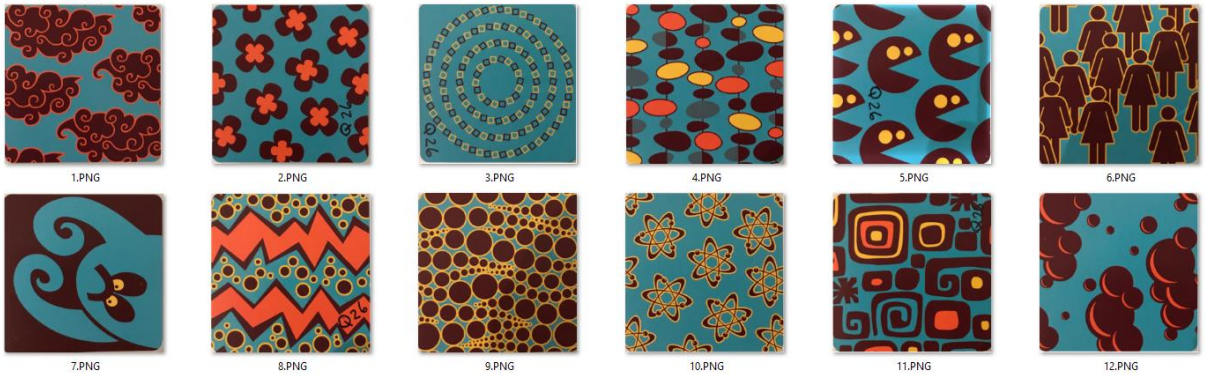


Figure 13: Niveau 2, « Blue »

Figure 14: Niveau 3, « Orange »

Une fois le scan de ces cartes fini, j'ai testé les nouveaux niveaux en changeant dans le code la variable pointant sur le chemin d'accès au dossier des images. Aucun problème détecté, le programme charge correctement les images. L'algorithme de récupération des cartes marche quelques soient les images. J'ai donc récupéré, sur le site https://www.colour-blindness.com/colour-blindness-tests/ishihara-colour-test-plates/ les images du **test de vision Ishihara** pour en faire mon 4ème niveau.

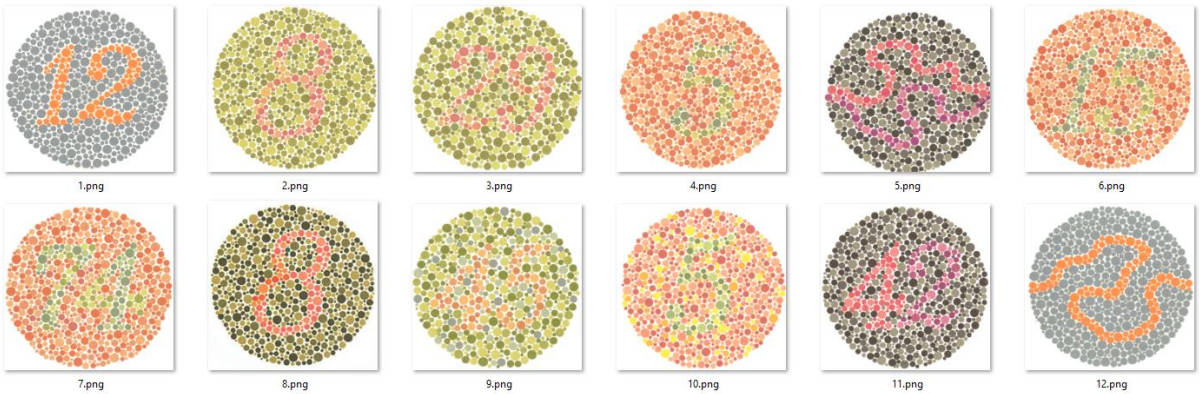


Figure 15: Niveau 4, « Ishihara »

Mon dernier niveau vient du site web http://www.jeu-test-ma-memoire.com/. C'est un mélange de figures rouges et blanches. Ce dernier est crédité à ©Istockphoto.

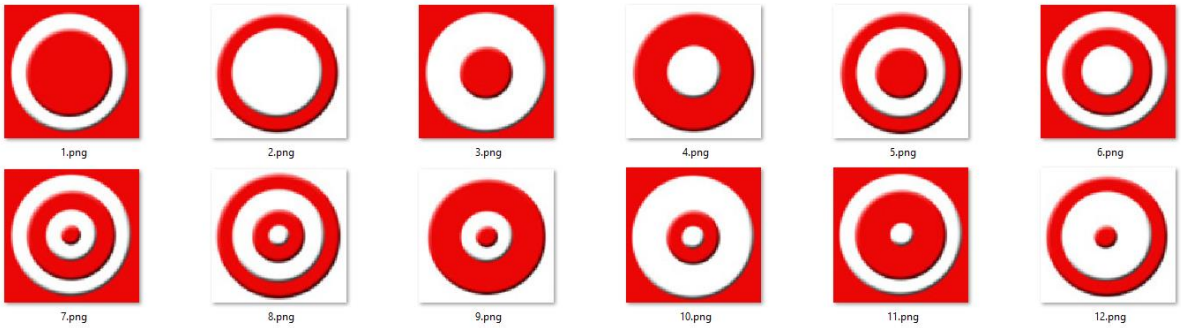


Figure 16: Niveau 5, « Red »

Le choix des niveaux a été très minutieux. Malgré la simplicité dont peut paraître la tâche, il me fallait faire des choix d'images en visant le public senior. C'est pourquoi ce Serious Game ne possède pas de niveau basique contenant des images de chat, souris, maison, fruit, etc.

Après divers tests, j'ai remarqué que Unity prend également en charge les images en **jpeg/jpg**.

4.6. Menu

Le menu du jeu a été implémenté après la logique et les 3 premiers niveaux. Je possède donc à ce stade un jeu fonctionnel dont la plupart des problèmes ont été résolus. Il me fallait un point d'ancrage, permettant à la fois de choisir le niveau et le nombre de cartes. J'y ai rajouté quelques options qui seront détaillées ci-dessous.

Ce menu est en réalité une seconde scène, la première étant le jeu du memory. En changeant l'ordre de chargement des scènes dans le *Unity Editor*, il est facile de forcer une scène à s'afficher avant une autre. Cette scène est composée principalement d'un canvas regroupant les différents menus et d'un script, **MenuScript.cs** permettant le bon chargement des options choisies par le joueur.

4.6.1. Boutons

Chaque GameObject bouton possède un composant *button* permettant de lancer un évènement. Ici, nous avons le bouton de sélection du niveau *Ishihara*. Au clic sur ce dernier, le bouton va charger la méthode *SetLevel(string lvl)* de la classe **MenuScript.cs** avec comme paramètre *lvl = CardIshihara/*. Il désactivera ensuite le GameObject regroupant les boutons de sélection du niveau (*LevelMenu*) et chargera le GameObject de sélection des cartes (*CardMenu*).

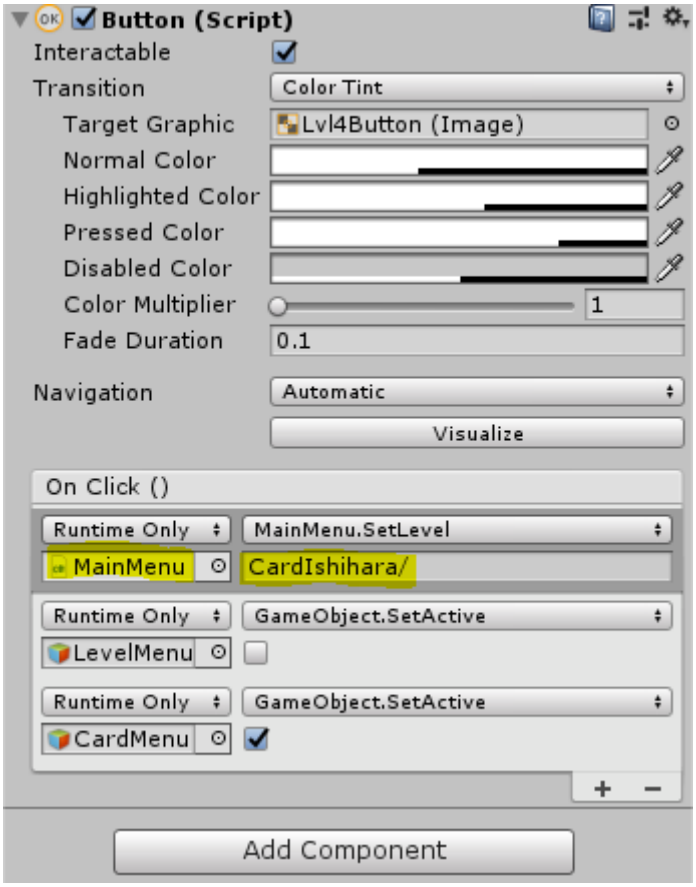


Figure 17: Composant bouton

4.6.2. Menu d'accueil

C'est le menu principal. Trois boutons sont affichés permettant respectivement de jouer, modifier les options ou quitter le jeu. Chaque menu correspond à un GameObject contenu dans le canvas principal.

Le GameObject **MainMenu** contiendra les boutons *Play, Option* ou *Quit*. C'est ici que sera lié le script **MenuScript.cs**. Ce dernier possède une méthode pour chaque choix fait par l'utilisateur.

Le **LevelMenu** contient les différents boutons pour la sélection du niveau.

C'est dans **CardMenu** que la sélection du nombre de carte se fait par le biais de 3 boutons.

OptionsMenu est le GameObject qui regroupe les différentes possibilités d'option pour l'utilisateur. Nous retrouvons donc le bouton permettant de changer de dos de carte, les slider ainsi que le bouton *About*.

Chaque choix fait par l'utilisateur, valeur modifiée ou clic sur un bouton est géré par une méthode dans le script **MenuScript**. Ce script, lié au GameObject **MainMenu** s'occupe de sauvegarder les options faites par le joueur.

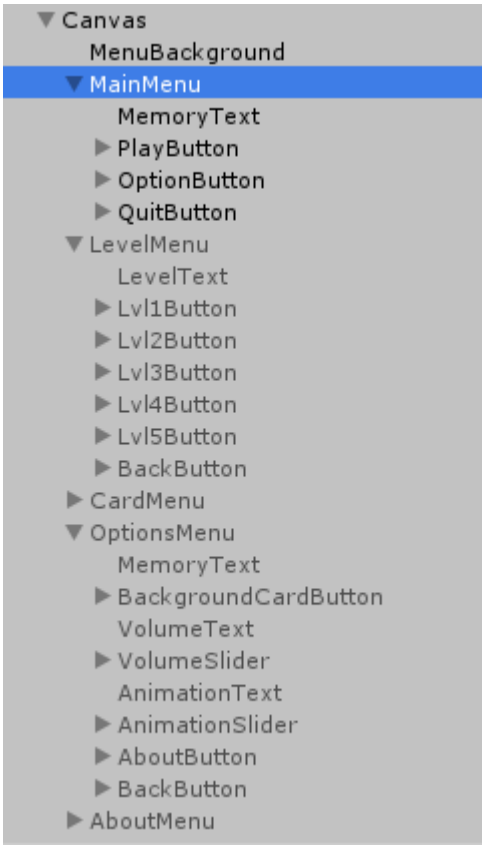


Figure 18: Composition du menu

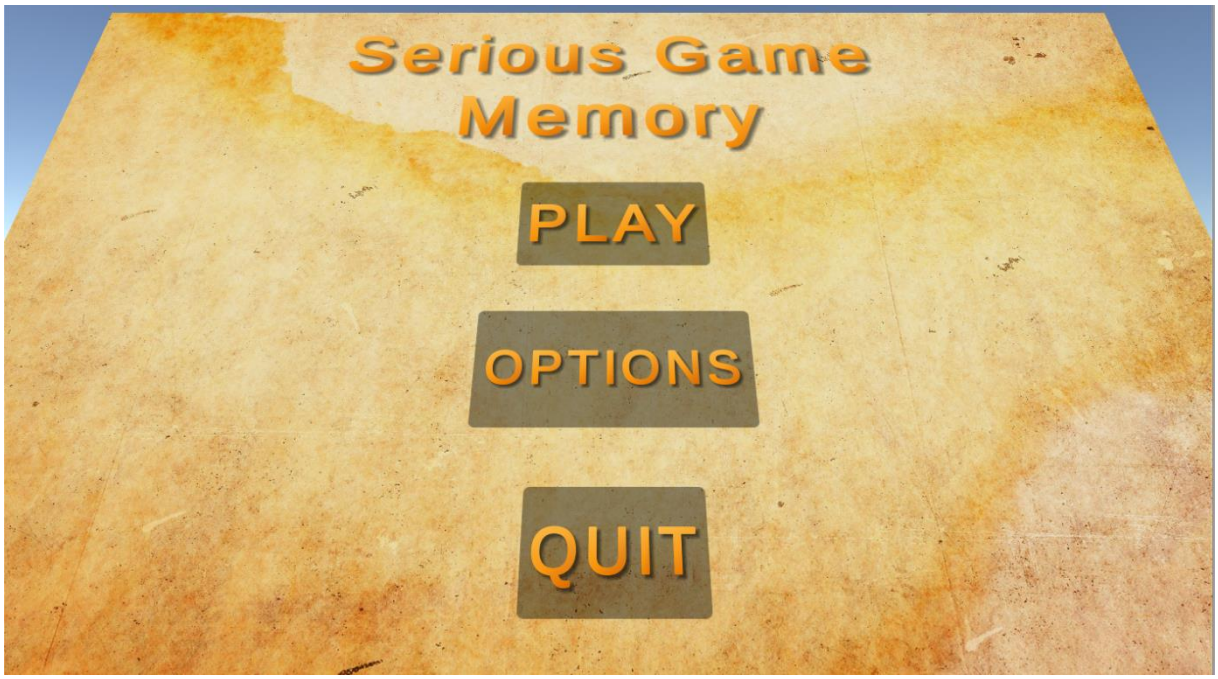


Figure 19: Menu principal

4.6.3. Sélection du niveau

C'est ici que se fait la sélection du niveau. Chaque bouton est assigné à un changement de valeur. Au choix d'un niveau, la méthode *SetLevel(string lvl)* est chargé et s'occupe de sauvegarder le niveau choisi dans une classe statique détaillée plus tard.

Figure 20: Sélection du niveau

4.6.4. Sélection du nombre de cartes

La sélection du nombre de carte vient directement après le choix du niveau. Les 3 possibilités sont évidentes, 16, 20 ou 24 cartes au total. Le clic sur le bouton chargera la méthode *SetNbCard(int nbCard)* du script.

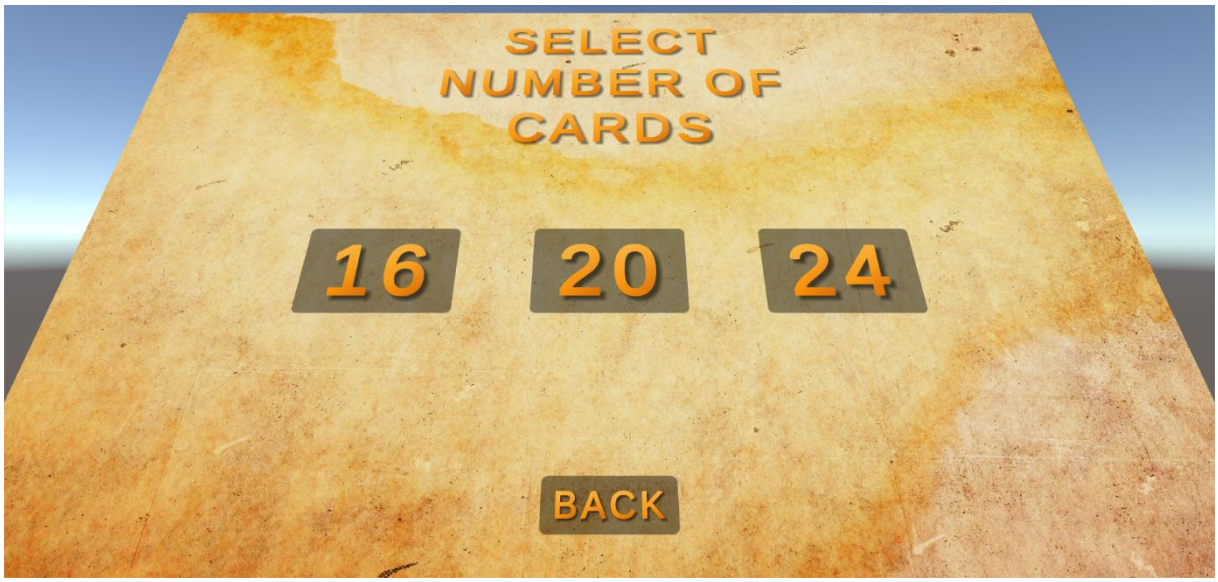


Figure 21: Menu de sélection de cartes

4.6.5. Options

Ce menu option offre 3 choix à l'utilisateur. Le premier est de changer l'image du dos des cartes. Lancé par la fonction *SetBackgroundCard()* qui permet à l'utilisateur de choisir une image .png pour son dos de carte, elle sauvegardera ce choix dans la classe **StaticParameter-Class** dont les attributs seront lus dans **le BoardGenerator.**

La deuxième fonction permet de changer le niveau du volume. En effet, la lecture d'une musique pour le menu et pour le jeu fait partie intégrante du projet. C'est donc ici, par le biais de la méthode *OnVolumeSliderValueChanged()* que tous les composants AudioSource de mon jeu verront leur niveau de volume changé.

La dernière fonction permet de changer la vitesse de retournement des cartes. En positionnant le slider à sa plus petite valeur, les cartes resteront moins longtemps face visible, ajoutant donc une difficulté à ce Serious game par la rapidité du retournement. Évidemment, le retournement se produit que lorsque 2 cartes retournées ne sont pas identiques. La méthode *OnSpeedSliderValueChanged()* s'occupe de la sauvegarde de la valeur du slider.

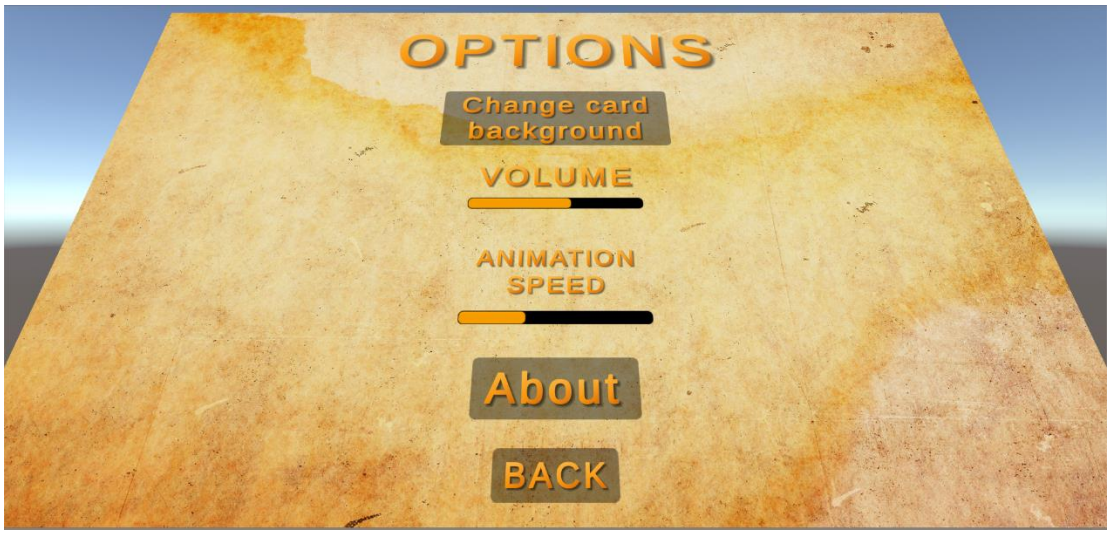


Figure 22: Menu options

4.7. Classe statique

Comme expliqué dans la section précédente, la sauvegardes des choix de l'utilisateur se fait grâce à une classe statique. Cette dernière, qui n'est bien entendu pas MonoBehaviour est composée uniquement de propriétés statiques accessibles depuis n'importe quelle autre classe. Il est donc aisé de sauvegarder les options et les charger depuis une autre classe à un moment souhaité. Par exemple, la propriété *LevelInformation* stock un string distinct pour le niveau choisi depuis le menu. Cette propriété est ensuite chargée dans la classe **BoardGenerator** qui se chargera de récupérer le chemin contenu dans la propriété, chargeant ainsi le niveau choisi par le joueur.

```
public static class StaticParameterClass
{
    #region properties
    public static string LevelInformation { get; set; }
    public static int CardInformation { get; set; }
    public static Texture2D TextureBackgroundCard { get; set; }
    public static float VolumeLevel { get; set; }
    public static float Score { get; set; }
    public static string Time { get; set; }
    public static int NbHit { get; set; }
    #endregion
}
```

Figure 23: Classe statique

4.8. Effet flip et Coroutines

Comme expliqué dans la section 4.1.1, les coroutines permettent de mettre sur pause l'exécution du programme, de lancer un bout de code et de reprendre le code principal. Dans ce projet, ces fonctions ont toutes été implémentées dans une classe à part : la **CoroutineClass.** Ce système de coroutine repose sur une instruction magique : ***yield return null.***

Ici, la lecture du code est mise en pause et reprendra à la prochaine frame. L'instruction ***yield return new WaitForSecondsRealTime(float seconds)*** permet au programme de mettre le code en pause et de reprendre après le nombre de seconde passé en paramètre.

C'est elle qui gère entre autres les effets de mouvement de la carte et les temps de pause dans le jeu. Elle implémente entre autres les fonctions suivantes :

- **Rotation(Transform thisTransform, Vector3 degrees, float time)**
  Retourne la carte de 180° (*degrees*) dans un temps donné (*time*).
- **Translation(Transform thisTransform, Vector3 startPos, Vector3 endPos, float value, MoveType moveType)**
  Déplace la carte d'un point à un autre (startPos to endPos) dans un temps donné (*value*).
- **WaiterRotateBack(CardScript c1, CardScript c2)**
  Lance la fonction *RotateBackCard* depuis le script **c1** et **c2**. Cette fonction est lancée quand les 2 cartes ne sont pas identiques.
- **WaiterTranslate(CardScript c1, CardScript c2)**
  Lance la fonction *TranslateBackCard* depuis les **CardScript** permettant de reposer les 2 cartes faces visibles si ces dernières sont identiques.
- **OnMouseDown(CardScript cs, GameScript gs)**
  Lance le système de rotation de la carte **cs**, attend un temps de 0,65 secondes puis ajoute la carte à la liste pour checker les ID (dans la classe **GameScript**).
- **WinGame(GameScript gs)**
  A la fin du jeu, attend 1 seconde avant de lancer la scène de fin de jeu via la méthode *WinGame* de la classe **GameScript**.

Sur papier ces enchainements de coroutine semblent biscornus, quand le jeu est lancé, ces instructions permettent d'avoir un effet *flip* joli en adéquation avec le jeu Memory. Elles permettent aussi d'avoir un jeu dans lequel aucune action n'ira bloquer une autre. Tout s'enchaine sans problème grâce aux coroutines.

4.9. Musique et effets sonore

Les musiques et effets sonores sont eux aussi en adéquation avec le projet. Nous comptons en tout 2 musiques et 4 effets sonores. Les musiques proviennent du site internet https://www.playonloop.com tandis ce que les effets viennent de https://www.zapsplat.com. Ces deux sites sont libres de droits pour les projets qui ne sont pas distribués commercialement.

La 1ère musique est incluse dans le composant *AudioSource* du GameObject *MenuAudioSource* de la scène **Menu**. C'est en effet la musique de fond jouée pendant que le joueur est sur le menu d'accueil.

La 2ème musique est incluse de la même manière dans un GameObject *MainAudioSource* de la scène de jeu, **MemoryGame**. C'est la musique lancée pendant la partie de memory.

Les sons sont inclus directement dans leur scripts respectifs car ils sont répartis en plusieurs niveaux. Les 3 premiers sons sont joués respectivement pour le retournement de la carte,

quand les 2 cartes sont identiques et quand les 2 cartes ne sont pas identiques. Le dernier effet sonore est joué lors de la victoire.

4.10. Résultats

Le projet est maintenant complet, du menu de démarrage à la fin du jeu en passant par la création du board et l'implémentation des divers algorithmes de sélection et logique. Une refactorisation complète ainsi que l'ajout de commentaires XML ont été réalisés les 2 dernières semaines de cours.

Pour ce projet, j'ai aussi utilisé un asset pour la police d'écriture du texte. Il est disponible sur ce lien https://assetstore.unity.com/packages/essentials/beta-projects/textmesh-pro-84126. Chaque texte (score, temps, boutons, etc.) reprend le même style de police. Le dossier final du projet se décrit de cette façon :

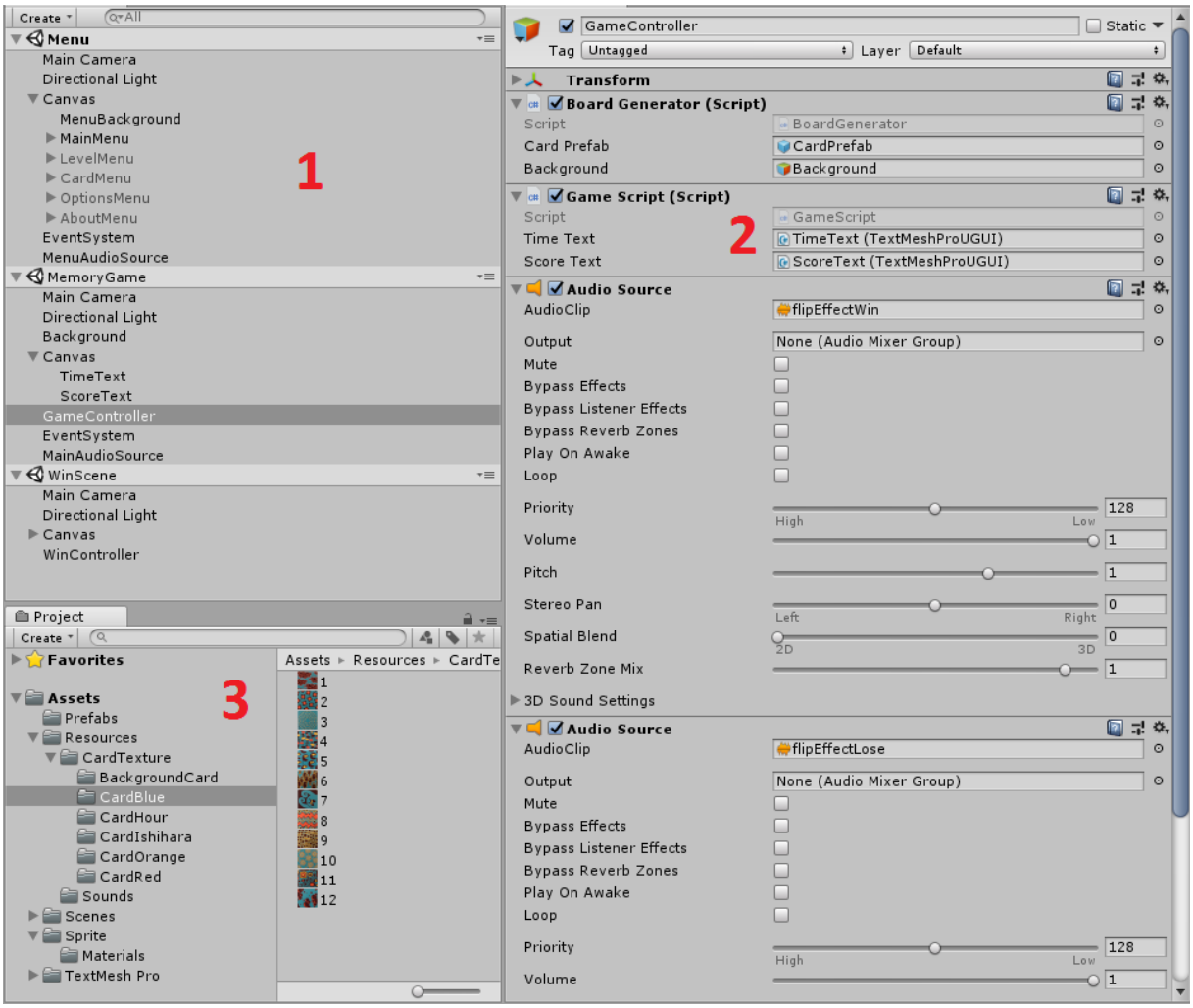


Figure 244: Structure du projet

1. Scènes et objets
    a. Scène **Menu**, par son nom le menu de démarrage, est composé du canvas principal sur lequel les différents menus sont regroupés
    b. La scène principale, le **MemoryGame** qui est composé du background dans lequel les cartes seront dynamiquement instanciées, d'un canvas contenant le score et le temps, du Gameobject *GameController* et de la source audio pour la musique du jeu.
2. Composants
    a. Sur cette image sont affichés les composant du *GameController*. Comme ce dernier est vide, rien de graphique ou visuel n'est ajouté à ses composants. Cependant, nous pouvons clairement voir les 2 scripts nécessaires à la génération du board (**BoardGenerator**) et au déroulement de la partie (**GameScript**). Les derniers composants sont les sources audio jouées lors d'un bon ou mauvais check des cartes retournées.
3. Assets
    a. Ce dossier est une visualisation complète de tous les fichiers nécessaires au projet.
    b. Le dossier **Prefabs** contient uniquement le GameObject *CardPrefab*.
    c. **Resources** contient toutes les ressources nécessaires au jeu (images, sons). Au moment du choix du niveau, le programme parcourra le chemin jusqu'au dossier souhaité.
    d. C'est dans **Scenes** que seront regroupées les 3 scènes de jeu, la scène du menu, celle du jeu et celle de la fin de jeu.
    e. Les seuls **Sprites** utilisés sont respectivement le dos de carte par défaut et le logo de la HE-ARC affiché dans le menu *About*.
    f. Enfin, comme expliqué ci-dessous, ce projet utilise l'asset gratuit **TextMesh Pro**, rendant les différents textes du jeu en adéquation avec le thème.

C'est aussi dans ce dossier Assets que seront mis les différents scripts du jeu.

5. Problèmes rencontrés

Durant le déroulement du projet, j'ai rencontré quelques problèmes qui, pour la plupart, ont été résolu. Grâce à ma charge de travail régulière, aucun de ces problèmes ne m'a bloqué pendant longtemps.

5.1. Génération du board

Le premier problème se trouvait dans la partie de création du board. Comment lier le tableau de jeu avec mon prefab de carte et comment placer ces dernières sur mon board ?

J'ai essayé plusieurs versions tout en créant au fur et à mesure mon script **BoardGenerator**. Placement dynamique, instanciation des cartes différemment, ne pas les lier en tant que fils du board mais cela ne me convenait pas. Après plusieurs essais, je suis parti sur une version statique. En effet, mon jeu ne comprend que 16, 20 ou 24 cartes. J'ai donc 3 possibilités différentes et c'était plus simple de coder chaque possibilité différemment.

J'ai donc pu créer un nombre de carte adéquat, les instancier et toutes les positionner sur le board à la même place. Comme chaque carte se place de la même manière, il me fallait une méthode qui convenait à toutes les cartes. C'est là que j'ai pensé à procéder en 2 étapes : 2 méthodes *PositionColCard()* et *PositionRowCard()* qui gèrent respectivement le placement en colonne, puis le placement en lignes.

Problème résolu, seul bémol : Si je souhaite un jeu avec un nombre de carte différent, je dois modifier ces méthodes de placement.

5.2. Effet flip

Le plus gros problème de mon projet était l'animation pour le retournement des cartes. Comment donner un effet flip joli pour l'utilisateur ?

J'ai passé beaucoup de temps sur ce problème. J'ai commencé par créer mes propres fonction de retournement avec le *Time.deltaTime* de Unity. Malheureusement, cela donnait toujours un résultat faussé. La carte ne se retournait pas complètement ou ne se retournait que pour une frame.

Après plusieurs heures de recherche, j'ai trouvé un script, **MoveObject.cs** sur ce site http://wiki.unity3d.com/index.php/MoveObject. Il utilise les coroutines et cela m'a beaucoup aidé. J'ai donc intégré ce script à mon projet, modifié quelques fonctions et mon retournement fonctionnait.

J'en ai profité pour ajouter toutes mes fonctions coroutines dans cette classe. Comme j'avais déjà utilisé cette fonctionnalité dans mon code, la refactorisation était simple et ne m'a pas pris beaucoup de temps.

5.3. Désactivation du clic sur les cartes quand elles se retournent

Le problème, qui n'a toujours pas été résolu, se trouve dans la suite d'évènement du retournement de deux cartes.

Lorsqu'un joueur clique sur une carte, un processus de rotation et translation est lancé pour afficher son image. C'est la méthode *RotateCard()* du script **CardScript.** Jusque-là tout va bien.

Lorsqu'un joueur retourne 2 cartes et que ces dernières ne sont pas identiques, un processus de rotation et de translation est lancé pour reposer les cartes faces cachées sur le board. C'est le rôle de la méthode *RotateBackCard()* du script **CardScript**.

Le problème survient ici : si le joueur clique sur une carte pendant le processus *RotateBackCard()*, cette dernière lance une rotation *RotateCard()* du même script alors que le processus d'avant n'est pas fini !

Figure 255: Retournement d'une carte

Figure 26: Les 2 cartes sont différentes. Appel de RotateBackCard()

Figure 267: En cliquant sur celle de droite pendant son replacement, bug

J'ai essayé plusieurs méthodes pour arranger ça. La première étant de désactiver le *collider* sur une carte. Malheureusement cela ne marche par car la réactivation de ce dernier ne se fait pas correctement. L'instruction est la suivante :

```
public void RotateBackCard()
{
    GetComponent<Collider>().enabled = false;
    StartCoroutine(CoroutineClass.Rotation(transform, new Vector3(0, 180, 0), StaticParameterClass.SpeedLevel + 0.1f));
    StartCoroutine(CoroutineClass.TranslateTo(transform, source, 1.0f, CoroutineClass.MoveType.Time));
    GetComponent<Collider>().enabled = true;
}
```

Figure 278: Désactivation du collider, rotation/translation puis réactivation du collider

Mais le problème est toujours présent.

J'ai essayé de jongler avec ce collider, en le désactivant au début de la méthode *RotateCard()* puis en le réactivant à la fin de la méthode *RotateBackCard()*. La désactivation du *collider* se passe bien, mais la réactivation se fait trop tôt.

Une autre solution essayée était la désactivation complète du GameObject. Cela n'a pas fonctionné car ce dernier doit changer de position et de rotation dans la méthode et elle nécessite un GameObject activé.

Arrivé à la fin du temps donné, le problème persiste. Je pense qu'avec un peu plus de temps j'aurais pu réactiver le *collider* au bon moment mais cela demande surement beaucoup de modification de méthodes, notamment en passant le *collider* entre les différents scripts.

5.4. *Build* du projet

Ce problème m'est arrivé durant la dernière semaine avant le rendu de ce projet. Je devais générer un exécutable en passant par le *Build settings* de l'éditeur Unity. Je sélectionne donc le chargement de mes scènes. Mon **Menu** arrive en premier, ma **MemoryGame** en second puis vient finalement la **WinScene** à la fin du jeu.

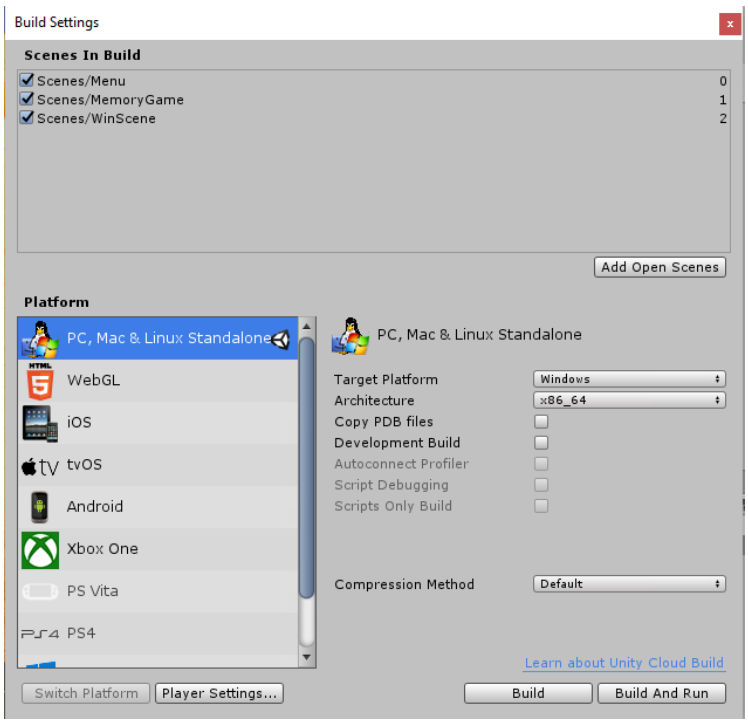


Figure 289: Build settings

Au moment du *build*, une erreur s'affiche me disant que la classe **EditorUtility** n'est pas accessible en mode déploiement. Après quelques recherches, je trouve la source du problème qui est la boite de dialogue pour choisir son dos de carte. En effet, j'utilise une *OpenFilePanel* héritant directement de la classe **EditorUtility** et cette dernière n'est disponible qu'en mode éditeur et pas en mode exécutable.

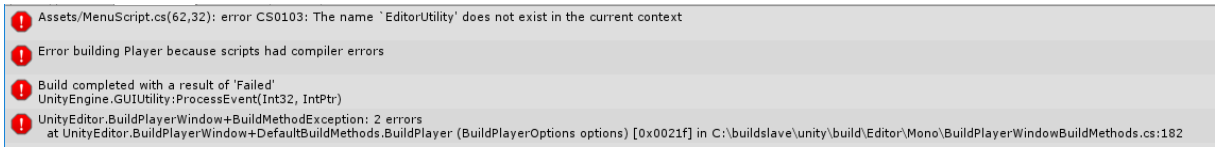


Figure 3029: Erreur de build

Après quelques recherches, je suis tombé sur un projet github https://github.com/gkngkc/UnityStandaloneFileBrowser qui contient le code d'une simple fenêtre de sélection de fichiers. J'ai donc repris son code pour l'intégrer à mon projet. Le changement se fait au niveau de la méthode *SetBackgroundCard()* du script **MenuScript**. Au lieu d'ouvrir une *OpenFilePanel* héritant de EditorUtility, j'ouvre un même panel héritant de la classe **StandaloneFileBrowser** téléchargée depuis le lien ci-dessus.

Après quelques modifications en fonction de cette nouvelle classe, le code marche parfaitement bien et j'ai pu construire l'exécutable et ses fichiers nécessaires.

```
public void SetBackgroundCard()
{
    Texture2D texture = new Texture2D(100, 100);

    //Used only in Editor mode
    //string pathToTexture = EditorUtility.OpenFilePanel("Overwrite with png", "", "png");

    // Open file with filter
    var extensions = new[] {
    new ExtensionFilter("Image Files", "png", "jpg", "jpeg" ),
    };

    var pathToTexture = StandaloneFileBrowser.OpenFilePanel("Open File", "", extensions, true);

    if (pathToTexture.Length != 0)
    {
        var fileContent = File.ReadAllBytes(pathToTexture[0]);
        texture.LoadImage(fileContent);
        StaticParameterClass.TextureBackgroundCard = texture;
    }
}
```

Figure 31: OpenFilePanel pour sélectionner un dos de carte personnalisé

6. Améliorations futures

Les objectifs principaux et quelques objectifs secondaires sont remplis, mais le projet peut encore être amélioré. Comme noté dans mon cahier des charges et mes spécifications de projet, je pourrais ajouter une fonction de sauvegarde. Cette dernière à déjà été ajouté en tant que bouton et fonction (ces 2 ont été désactivés pour le rendu final) mais l'implémentation n'est pas encore faite. Il suffirait juste d'enregistrer le score du joueur, son nom ainsi que le temps utilisé dans un fichier. Ce dernier pourrait être chargé dans une zone de texte à la fin de chaque partie. Après en avoir longtemps parlé avec mon superviseur, nous avons jugé préférable de ne pas se lancer dedans à la fin, de peur de manquer de temps pour une tâche aussi chronophage.

Aussi, une amélioration et refonte graphique pourrait voir le jour. Pourquoi pas créer ses propres dos de cartes, modifier son background avec *Blender*.

Une tâche qui m'aurait également intéressé serait de choisir son propre set de carte. Avec une modification de la classe **StandaloneFileBrowser**, il serait facile d'ajouter une option à la sélection de niveau permettant de choisir 12 cartes. Ces 12 cartes seraient ensuite stockées dans la liste *_listImg* puis chargées à chaque instanciation d'une carte en tant que texture. Il faudrait bien évidemment effectuer de multiples tests en vérifiant que le nom des carte soit bien entre 1 et 12 et qu'elles soient de type image (jpg, jpeg, png, etc.)

Une dernière amélioration, qui consiste plutôt en une modification, serait d'ajouter une version joueur vs joueur. Le jeu du Memory se joue en réalité à 2 joueurs ou plus. Comme expliqué dans mon introduction, ce Serious game est une version revisité. Mais rien n'empêche de lui ajouter une version joueur vs joueur. Cela demanderait énormément de temps, en effet, la logique du jeu resterait la même mais l'ajout d'un 2ème joueur demande une modification et refonte complète des autres classes. Cette amélioration serait très chronophage.

7. Conclusion

Pour conclure ce rapport, plusieurs choses sont à noter. La première est que tous les objectifs principaux ont été remplis. Ce Serious game regroupe toutes les fonctionnalités principales décrites dans le cahier des charges et les spécifications du projet.

Plus encore, j'ai eu le temps d'y ajouter quelques améliorations tout au long de ces 15 semaines. En commençant par la rédaction du cahier des charges jusqu'à l'implémentation de celui-ci en passant par des diagrammes, plannings et multiples problèmes, ce projet a été finalisé et rendu dans les temps. Mon organisation du travail m'a permis de ne jamais être bloqué pendant plusieurs semaines sur une même tâche. En effet, même si le planning de Gantt n'a pas été respecté à la lettre, ma régularité pour l'avancement de ce projet m'a permis d'obtenir un résultat final concluant.

Les améliorations étant multiples, je regrette quand même de n'avoir pas eu assez de temps pour finaliser la sauvegarde du score et régler le problème du retournement des cartes décrit à la section 5.4. Ajouté à cela, une refonte graphique de meilleure qualité (changement de fond, personnalisation du background, meilleur placement du score/texte) aurait pu voir le jour avec quelques semaines à disposition en plus.

La réalisation de ce Serious game m'a permis non seulement de peaufiner mes connaissances actuelles en Unity, mais aussi d'apprendre de nouvelles fonctionnalités qu'offre ce moteur de jeu comme le système de coroutines, de canvas ou la création de menu.

8. Sources

Documentation Unity : https://docs.unity3d.com/ScriptReference/index.html

Forum Unity : https://answers.unity.com/index.html

GitHub gnkgnc : https://github.com/gkngkc/UnityStandaloneFileBrowser

TextMeshPro : https://assetstore.unity.com/packages/essentials/beta-projects/textmesh-pro-84126

Jeu de memory : http://www.jeu-test-ma-memoire.com/jeux-de-memory/memory-a-imprimer

Diverses aides et démarche à suivre : https://stackoverflow.com/ et https://forge.ing.he-arc.ch/projects/he-arc-inf-1718/wiki/Directives_TA

9. Annexes

Ci-joint :

- Cahier des charges
- Spécification du projet
- Journal de travail
- Diagramme UML des classes
- Diagramme de cas d'utilisation
- Planning de Gantt
- Code source du jeu
- Exécutable sous formes de .exe compressé dans un dossier .zip
- Présentation