TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABBRÉVIATIONS

A	Adénine
ADN	Acide DésoxyriboNucléique
APK	Android Package Kit
C	Cytosine
CO	Cycle d'Orientation
G	Guanine
HEP	Haute École Pédagogique
JSON	Javascript Object Notation
SDK	Software Development Kit
SIB	Swiss Institute of Bioinformatics
T	Thymine

INTRODUCTION

L'origine de l'oxymore « jeu sérieux » remonte à l'époque de Platon et a été utilisé au XVIIème siècle pour décrire le climat qui régnait à la cour du roi de France (Sanchez, 2012), autrement dit, pour désigner une situation où s'opposent la frivolité des comportements et l'importance des enjeux. Selon Sanchez, l'apprentissage est un processus où un apprenant serait amené à adapter sa manière de penser et d'agir dans un contexte de résolution de problème. Il affirme que l'usage dans des classes de jeux vidéo simulant une situation de référence et proposant des défis et des problèmes à résoudre est justifié par un tel point de vue. Cependant, l'interactivité d'un logiciel informatique dépend de la manière dont celui-ci est présenté et des tâches qui sont proposées à l'utilisateur. De plus, il est important d'alterner des phases de jeu et des phases de prise de recul pour conserver le caractère sérieux de la situation et consolider les connaissances acquises. Ainsi, Sanchez propose d'employer l'expression « jeu sérieux » pour décrire une situation ludique intégrant un outil informatique, plutôt que l'outil lui-même.

Dans le contexte de son travail d'enseignant en biologie au secondaire I (cycle d'orientation), l'étudiant en Master à la Haute École Pédagogique Valaisanne (HEP VS) avec qui nous avons collaboré a développé un jeu au format papier permettant d'illustrer des principes simplifiés de la génétique, afin de faire comprendre à ses élèves la répartition aléatoire des gènes des parents à leurs enfants. Autrement dit, pourquoi des frères et sœurs issus de mêmes parents présentent des différences physiques. Les prémisses de la situation qualifiée de « jeu sérieux » par Sanchez existent donc déjà sous la forme de cours de sciences. La HEP VS et la HES-SO Valais souhaitent proposer l'outil informatique correspondant, qui offrirait un caractère ludique à ces cours, un *serious game* destiné à des adolescents de 10 à 15 ans. Dans le cadre de son travail, l'étudiant en Master souhaite, quant à lui, étudier l'évolution de sa manière d'enseigner suite à nos rencontres et discussions autour de la réalisation de cette application.

Un cahier des charges rédigé par l'étudiant en Master nous a été fourni (voir annexe I). Il préconise les objectifs suivants pour le futur jeu :

- L'utilisateur doit pouvoir choisir un père parmi un échantillon proposé
- L'utilisateur doit pouvoir faire de même avec la mère
- L'utilisateur doit pouvoir créer un ou plusieurs enfants à partir des gènes du père et de la mère
- L'utilisateur doit pouvoir rapidement observer les caractéristiques physiques de l'enfant créé
- L'aspect tactile de l'application est vivement souhaité pour favoriser l'apprentissage

Les caractéristiques physiques à observer ont également déjà été définies. Celles-ci sont au nombre de six :

- La couleur de la peau
- La couleur des yeux
- La couleur des cheveux
- La forme de la bouche
- La forme du nez
- Le sexe

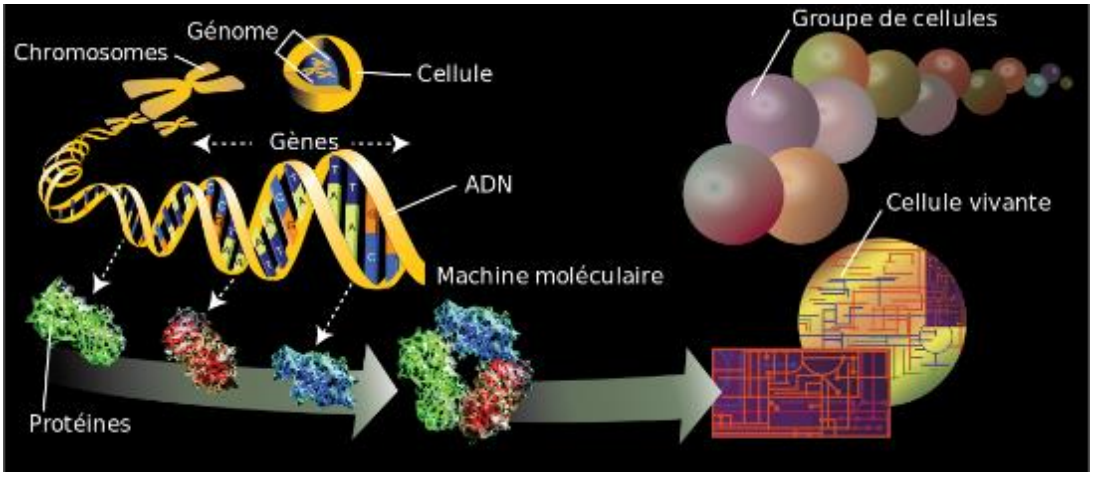
Une table de codage nous a été fournie pour définir la correspondance entre la combinaison de gènes et l'apparence de l'enfant qui en découle. Celle-ci figure en annexe 2 de ce mémoire.

Une étape souhaitée serait le test du prototype par un groupe de jeunes de 10 à 15 ans après la phase de développement. Ceci permettrait d'obtenir un *feedback* et de réunir des propositions d'amélioration du prototype. Cette phase d'évaluation devrait avoir lieu avant la fin de l'année scolaire, pour des raisons de disponibilité des testeurs. Par conséquent, la phase de développement aura pour contrainte supplémentaire d'aboutir à une version du prototype testable avant la fin du mois de mai.

Afin de favoriser la compréhension du jeu et de l'application, nous allons définir les notions de génétiques importantes à connaître dans ce contexte. La génétique consiste à étudier l'hérédité et les gènes (Wikipedia, 2019). La génétique mendélienne, une de ses branches, se concentre sur la transmission des caractères héréditaires entre des individus et leur descendance. Un gène est une unité d'information génétique composée de plusieurs nucléotides : l'adénine, la thymine, la guanine et la cytosine (A, T, G, C). Ils peuvent être codants, ce qui signifie que leur information génétique est utilisée, ou non-codants. Dans le second cas, l'information génétique assure une autre série de fonctions, comme l'activation de l'expression de certains gènes.

L'information génétique est portée par l'acide désoxyribonucléique (ADN), qui est formé de deux chaînes de nucléotides enroulées en double hélice : la chromatide (Wikipedia, 2019). L'ordre et le nombre de nucléotides d'un gène porte l'information génétique : on parle de séquence d'ADN. Deux chromatides identiques constituent un chromosome, forme sous laquelle on trouve l'ADN dans les cellules. L'ADN humain est composé de 23 paires de chromosomes (Wikipedia, 2019).

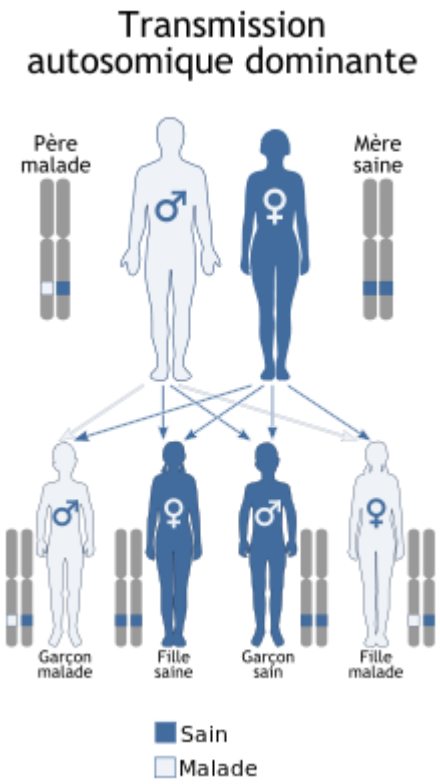
Figure 1 : Représentation de la molécule d'ADN jusqu'à la cellule vivante



Source : https://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9tique

L'hérédité est la transmission de caractéristiques d'individus à leur descendance (Wikipedia, 2019). Il peut s'agir de traits physiques, mais aussi comportementaux. Un exemple souvent utilisé est la couleur des yeux dans l'espèce humaine. Elle dépend d'un ensemble de gènes qui contrôlent la production de protéines dans l'iris, ce qui lui donne sa couleur. La figure 2 présente un exemple similaire, la transmission d'un gène pathologique dominant. On constate que le père a un gène pathologique sur l'une de ses chromatides, et un gène « sain » sur la seconde. Le caractère dominant fait que la présence d'un seul gène pathologique dans la paire de chromatides de la génération suivante entraîne la maladie.

Figure 2 : Transmission d'un gène pathologique dominant.



Source :

https://fr.wikipedia.org/wiki/H%C3%A9r%C3%A9dit%C3%A9#Un_exemple_:_l'h%C3%A9r%C3%A9dit%C3%A9_de_la_couleur_des_yeux

Déroulement du jeu

Dans le jeu créé par l'étudiant en Master, les gènes sont représentés par des symboles ou des dessins sur de petites cartes, et les chromatides par des cordelettes. Un ovale en papier plastifié avec des velcros placés au niveau des yeux, du nez, et de la bouche permet de reconstituer un visage en y plaçant différentes représentations des caractéristiques physiques. Les chromatides de l'enfant sont disposées côte à côte sur la table. Le jeu se déroule comme suit : on choisit les individus parents (un homme et une femme) parmi un échantillon proposé. On prend l'un des deux gènes de la mère codant pour une caractéristique physique spécifique et on le place sur une chromatide du chromosome de l'enfant. On prend un des deux gènes du père codant pour la même caractéristique physique et on le place sur la chromatide libre de l'enfant, à la même hauteur. On réalise ces deux étapes pour toutes les caractéristiques physiques des individus parents. Une fois tous les gènes en place, on reconstitue l'enfant ainsi créé à l'aide d'une table de codage en assemblant les différentes parties du visage correspondantes et on observe le résultat.

1. État actuel de la recherche

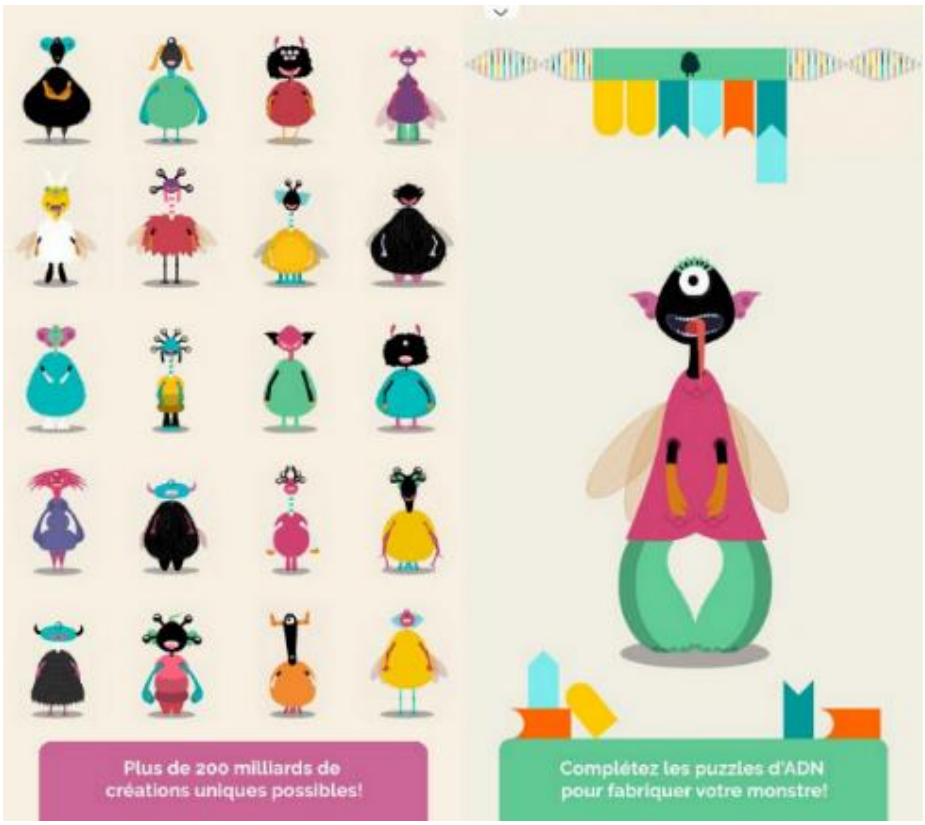
1.1. Applications similaires

Afin de proposer la meilleure alternative numérique pour ce jeu, nous avons réalisé une recherche d'applications similaires, qui proposent de créer un ou plusieurs individus à partir de parents et d'observer leurs caractéristiques. Nous avons également étudié des applications simplement en rapport avec l'ADN.

1.1.1. DNA Play

DNA Play est un jeu payant disponible sur Google Play qui permet de créer des monstres et de les transformer en manipulant leur ADN comme un puzzle. Il s'agit d'une introduction simple au fonctionnement de l'ADN, des gènes et des mutations, destinée à des enfants de 4 à 9 ans (Avokiddo, s.d.). En plus de la création de ces monstres, il est possible de les modifier en temps réel pendant qu'ils prennent part à des activités comme de la danse ou du skateboard, de générer des mutations aléatoires immédiates en touchant une partie de leur corps, de les sauvegarder dans une bibliothèque et d'enregistrer des instantanés. Des images du jeu sont présentées à la figure 3.

Figure 3 : Images du jeu DNA Play

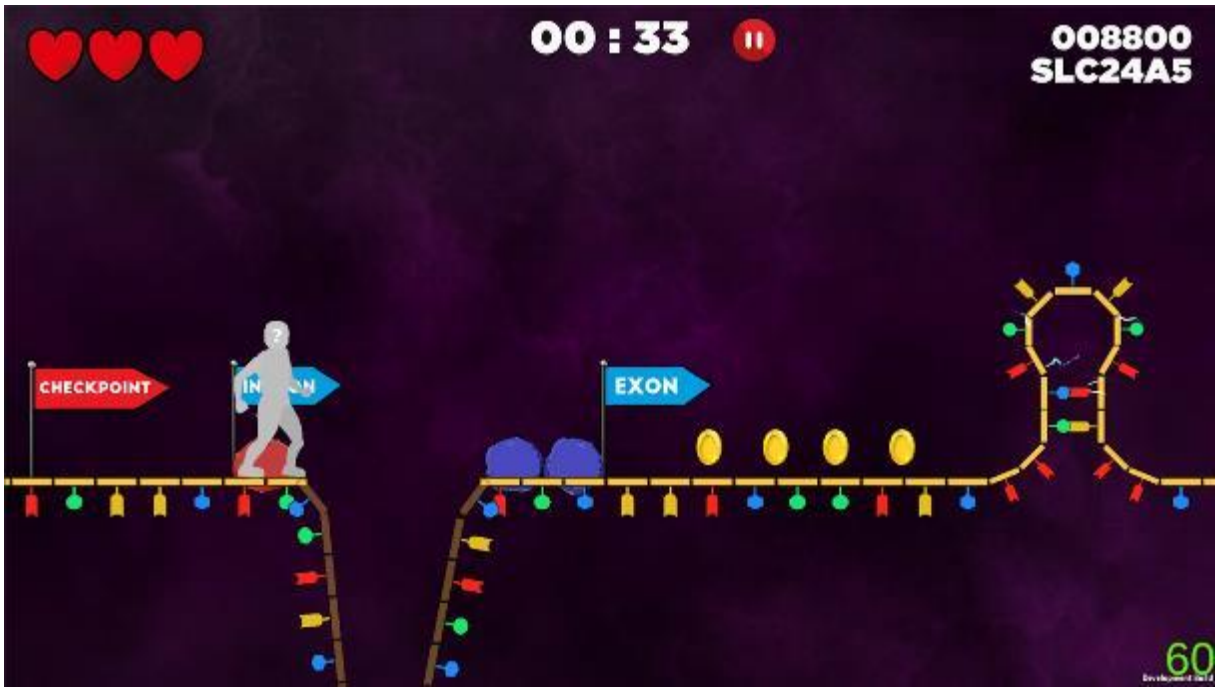


Source : https://play.google.com/store/apps/details?id=com.avokiddo.games.dnaplay

1.1.2. Genome Jumper

Genome Jumper est un jeu gratuit disponible sur Google Play et sur l'App Store qui a été développé par le *Swiss Institute of Bioinformatics* (SIB) afin de promouvoir et d'expliquer la bioinformatique au public. Il nous propose de parcourir des séquences de gènes sous la forme d'un personnage dont l'apparence se modifie en recueillant de petits changements dans notre ADN : des variants (Bioinformatics, s.d.). Le personnage court le long de la séquence d'ADN en évitant les chutes dans les introns (régions non codantes) et en sautant par-dessus des structures secondaires. Le niveau de difficulté augmente au fur et à mesure des séquences parcourues. Une image tirée du jeu est présentée à la figure 4.

Figure 4 : Image de Genome Jumper



Source : https://play.google.com/store/apps/details?id=ch.sib.genomejumperfinal

1.1.3. Pajama

Pajama est un jeu de simulation destiné à des jeunes dès 9 ans permettant de développer un élevage de poissons (*pajamas*) à partir d'un couple (Kasbi, 2012). Après avoir défini les caractéristiques du père et de la mère, on peut observer la descendance obtenue, supprimer certains poissons, les séparer, les accoupler, sans oublier de les nourrir, afin de réaliser des objectifs qui nous sont fournis. Cependant, ce jeu semble être une application web tombée en désuétude, car l'url fournie dans l'article de blog ne mène pas au jeu. Des images du jeu sont présentées à la figure 5.

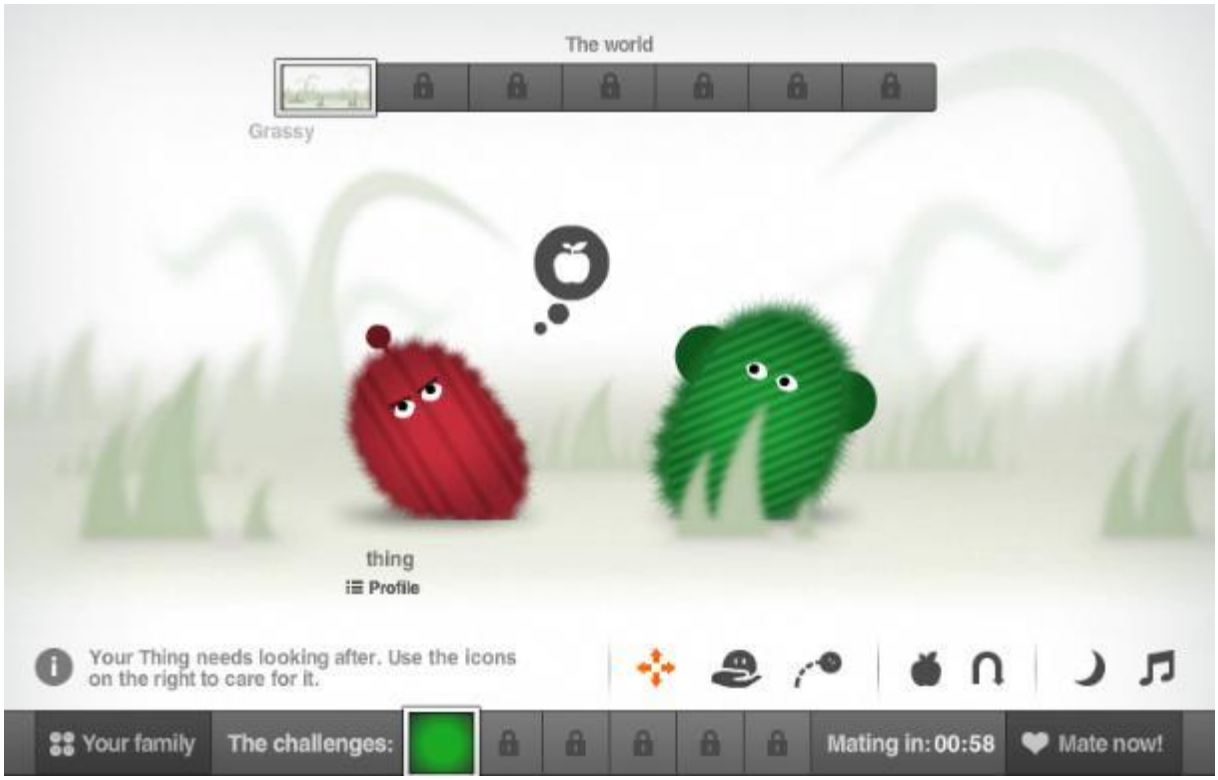
Figure 5 : Images du jeu Pajama

Source : http://blog.seriousgame.be/pajama-un-jeu-de-simulation-sur-la-gntique

1.1.4. Welcome to Thingdom

Welcome to Thingdom est une application web conçue par le musée des Sciences de Londres dans le cadre d'une exposition permanente intitulée *Who am I* (Un jeu amusant sur l'hérédité et la génétique, 2018). Ce jeu sérieux explique de façon ludique et simplifiée ce qu'est l'ADN, à quoi il sert, ainsi que les principes de gènes récessifs et dominants. Pour cela, il s'agit d'élever de mignonnes petites créatures et de les apparier avec d'autres créatures afin de remplir des objectifs précis, comme par exemple, obtenir un enfant d'une forme ou d'une couleur particulière. Avant cela, il faut s'occuper de la créature en la nourrissant et la divertissant en attendant qu'elle soit prête à procréer. Lorsque c'est le cas, elle doit séduire la créature de notre choix en jouant à différents mini-jeux. Après cette période de séduction, l'accouplement a lieu et une jeune créature apparaît. Puis le cycle recommence. Une image tirée du jeu est présentée à la figure 6.

Figure 6 : Image du jeu Welcome to Thingdom



Source : http://whoami.sciencemuseum.org.uk/whoami/thingdom

1.1.5. Analyse des applications

Les caractéristiques des applications étudiées sont regroupées dans le tableau 1. Nous avons considéré celles décrites dans le cahier des charges fourni par l'étudiant, ainsi que quelques attributs supplémentaires tels que la simplicité d'utilisation, l'accessibilité et si l'application est gratuite ou payante. Ces deux derniers critères sont présentés à titre informatif.

On peut constater qu'aucune des applications analysées ne répond à toutes les caractéristiques décrites dans le cahier des charges. Celles qui s'en rapprochent le plus sont Pajama et *Welcome to Thingdom* : elles comportent la notion de transmission de gènes de parents à un enfant et permettent d'en observer l'influence sur son apparence. Cependant, il y a également une notion d'élevage, de devoir prendre soin des individus, qui rallonge et complexifie la phase de création, ce qui n'est pas désiré dans ce contexte. En outre, Pajama ne semble plus être en fonction. À l'opposé, DNA Play est une application rapide et simple à utiliser. Elle est accessible sur Google Play moyennant une petite somme, mais n'intègre pas la notion de parents. *Genome Jumper* non plus, même si le but du jeu est tout de même d'observer des changements d'apparence se produisant en cours de partie. Par conséquent, le développement d'une nouvelle application est tout à fait justifié.

Tableau 1 : Caractéristiques des applications existantes

	DNA Play	Genome Jumper	Pajama	Welcome to Thingdom
Permet de choisir les parents	✗	✗	✓	✓
Permet de choisir les caractéristiques de l'enfant	✗	✗	✓	✓
Permet d'observer l'influence des gènes sur l'apparence	✓	✓	✓	✓
La création est rapide	✓	✗	✗	✗
Est tactile	✓	✓	✗	✗
Est simple d'utilisation	✓	✓	✗	✗
Est accessible	✓	✓	✗	✓
Est payant	✓	✗	✗	✗

Source : données de l'auteur

1.2. Choix de l'outil de développement

Dans le chapitre 1.1.3, nous avons souligné le fait qu'une application tactile était vivement souhaitée, afin de favoriser l'apprentissage du sujet par les élèves du cycle d'orientation (CO). Par conséquent, nous avons décidé de réaliser une application pour tablette. La HES-SO a mis à notre disposition un appareil fonctionnant sous Android afin de nous permettre de tester l'application durant le développement. Le premier critère est donc que l'outil en question permette le développement pour Android. Nous avons également considéré le fait que le développement puisse être multi-plateforme, afin que l'application soit également utilisable sur un autre appareil, comme un iPad, par exemple. Un plus serait que du contenu existant soit disponible et facilement utilisable, notamment pour la partie graphique, comme le visage des parents et de l'enfant. Pour terminer, certains outils de développement nécessiteront une période de formation, selon le langage et la logique de programmation correspondants. Cela devra être pris en considération par rapport aux délais à tenir.

1.2.1. Android Studio

Android Studio est l'outil proposé par Google pour le développement d'applications mobiles (Lancelin-Golbery, 2019). Il fournit un éditeur de texte pour créer du code en Java, un compilateur et des possibilités de tester l'application, que ce soit via un émulateur ou directement sur un appareil. Une application développée à l'aide d'Android Studio est dite native, ce qui signifie qu'elle utilise le *Software Development Kit* (SDK) fourni par Google (Marquez, 2018). Cela permet d'obtenir de meilleures performances. En revanche, l'application ne sera pas compatible avec un appareil ne fonctionnant pas sous Android.

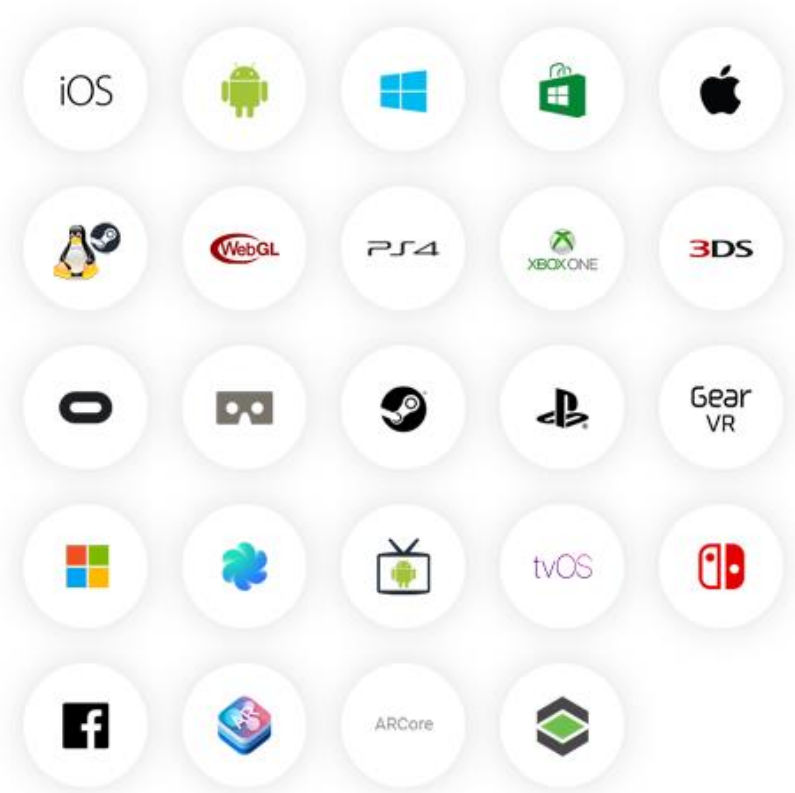
1.2.2. Flutter

Flutter est un *framework*, un ensemble d'outils et de composants logiciels, développé par Google, qui permet de développer des applications natives pour Android et iOS avec le langage de programmation Dart (Marquez, 2018). Ce dernier propose deux modes de fonctionnement : *ahead of time*, avec lequel on réalise les applications natives pour chaque plateforme, et *just in time* qui permet un développement plus rapide en réduisant le temps nécessaire aux *builds*.

1.2.3. Unity

Unity est un éditeur dédié au développement de jeux sur plus de 25 plateformes, telles que iOS et Android, mais également pour des consoles de jeux comme la Playstation 4, la Xbox One ou la Nintendo Switch (Unity Technologies, 2019). Il intègre la gestion de la 2D et de la 3D et propose toute une gamme d'outils de *design*. Il est également relié à un magasin, *l'Asset Store*, qui offre un catalogue important de modules (*assets*) gratuits et payants pour améliorer notre projet.

Figure 7 : Plateformes supportées par Unity



Source : https://unity3d.com/unity

1.2.4. Analyse des outils de développement

Les environnements de développement analysés permettent tous de réaliser une application pour Android. La synthèse des caractéristiques étudiées pour le choix de l'outil que nous allons utiliser dans le cadre de notre travail est présentée dans le Tableau 2. Nous avons utilisé Android Studio à plusieurs reprises dans le courant de notre formation. Son fonctionnement nous est donc connu. Cependant, il ne permet de produire que des applications destinées à Android, tandis que Flutter et Unity sont multi-plateformes. Nous avons découvert Flutter lors de notre recherche d'outils de développement sur Android. Celui-ci nécessiterait une période de formation, pour intégrer l'architecture et la logique de cet environnement d'une part, mais également pour apprendre le langage de programmation utilisé d'autre part. Ceci impliquerait une réduction trop importante du temps dédié au développement pour obtenir un prototype testable dans les délais impartis. Unity, quant à lui, est un éditeur dédié au développement de jeux, ce qui est tout à fait en adéquation avec l'objectif de ce travail. De plus, aucune formation n'est nécessaire, puisqu'il a déjà été utilisé lors d'un projet récent dans le cadre de nos études.

En parcourant *l'asset store* à la recherche de contenu, nous avons trouvé un *asset* permettant la création rapide de personnages très variés, intitulé *Super Visual Chibis*. Les caractéristiques physiques à choisir correspondent à celles mentionnées au chapitre 1.1.3, hormis la forme du nez. L'ensemble de ces paramètres, mais surtout la découverte de cet a*sset* a nettement fait pencher la balance en faveur d'Unity comme environnement de développement. De plus, nous avons estimé que l'apparence des personnages de cet *asset* correspond au public cible. Nous y voyons également l'occasion d'approfondir nos connaissances sur le sujet.

Tableau 2 : Caractéristiques des outils de développement

	Android Studio	Flutter	Unity
Développement pour Android	✓	✓	✓
Multi-plateforme	✗	✓	✓
Contenu existant	✗	✗	✓
Formation pas nécessaire	✓	✗	✓

Source : données de l'auteur

Figure 8 : Image de présentation de *l'asset Super Visual Chibis* sur *l'Asset Store*

Source : https://assetstore.unity.com/packages/2d/characters/super-visual-chibis-94257

2. Modélisation de l'application

La modélisation de l'application a déjà été en partie réalisée par l'étudiant en Master. Ce dernier nous avait transmis des *mockups* en annexe à son cahier des charges. Toutefois, ces représentations de l'application nous avaient semblé quelque peu surchargées, et en accord avec lui, nous avons proposé une nouvelle version plus adaptée au support destiné à accueillir le prototype. Nous avons également listé les fonctionnalités désirées sous forme d'user stories dans un *product backlog*.

2.1. Mockups

Nous avons utilisé Adobe XD afin de pouvoir partager aisément ces mockups avec les parties prenantes du projet. En effet, cet outil propose d'avoir un prototype actif et exécutable partagé dans sa formule starter gratuite (Adobe, 2019). Les écrans constituant l'application figurent dans l'annexe III de ce mémoire.

2.2. Product backlog

Lors de l'établissement de notre cahier des charges, nous avons décidé de travailler selon la méthodologie Agile et d'utiliser une adaptation de *Scrum*. Afin de préciser la forme du prototype à développer, nous avons retranscrit les spécifications de l'application, ainsi que ses fonctionnalités sous forme *d'user stories*. Celles-ci sont regroupées par ordre de priorité dans le tableau 3 et représentent un total de 48 *story points*. L'outil utilisé pour gérer le *product backlog* sera détaillé au chapitre 6. Les u*ser stories* en italique sont des fonctionnalités supplémentaires qui dépassent le cadre des objectifs définis au début du projet. Ainsi, le nombre de *story points* à atteindre est de 33 pour y répondre.

Tableau 3: User stories composant le projet

Numéro	User story	Story points
13	En tant qu'utilisateur, je veux pouvoir accéder à un écran d'accueil pour pouvoir jouer au jeu	1
1	En tant qu'utilisateur, je veux pouvoir choisir le père pour créer un enfant par la suite	3
12	En tant qu'utilisateur, je veux pouvoir choisir la mère pour pouvoir créer un enfant par la suite	3
9	En tant qu'utilisateur, je veux pouvoir créer un enfant pour observer la correspondance entre ses gènes et son apparence	5
3	En tant qu'utilisateur, je veux pouvoir glisser-déposer les gènes des parents correspondants aux caractéristiques physiques sur les chromatides de l'enfant	3

4	En tant qu'utilisateur, je veux pouvoir visualiser l'apparence de l'enfant créé	8
8	En tant qu'utilisateur, je veux pouvoir enregistrer un enfant	2
7	En tant qu'utilisateur, je veux pouvoir consulter une liste des enfants déjà créés	3
16	En tant qu'utilisateur, je veux pouvoir observer un enfant existant pour avoir une vue plus détaillée.	2
5	En tant qu'utilisateur, je veux pouvoir ajouter un nouvel enfant à partir des mêmes parents que l'enfant précédent	2
11	En tant qu'utilisateur, je veux pouvoir retourner à l'écran d'accueil après avoir observé un enfant existant pour pouvoir continuer le jeu	1
6	*En tant qu'utilisateur, je veux obtenir une aide pour comprendre le jeu*	*3*
10	*En tant qu'utilisateur, je veux pouvoir quitter l'écran d'aide pour continuer le jeu*	*1*
14	*En tant qu'utilisateur, je veux pouvoir partager ce que j'ai enregistré pour pouvoir le consulter sur un autre appareil*	*2*
15	*En tant qu'utilisateur, je veux pouvoir importer des enfants pour les observer sur un autre appareil*	*5*
17	*En tant qu'utilisateur, je veux pouvoir accéder à un menu d'importation pour pouvoir importer des enfants créés sur d'autres appareils*	*2*
18	*En tant qu'utilisateur, je veux pouvoir entrer mon nom pour pouvoir retrouver mon travail*	*2*

Source : données de l'auteur

3. Développement

3.1. Organisation du développement

Avant de commencer le développement, nous avons déterminé avec l'étudiant en Master quelles fonctionnalités devraient être mises en place avant de procéder aux tests. En effet, la phase d'évaluation devant se dérouler avant la fin de l'année scolaire des cycles d'orientation, les délais ne permettront pas le développement complet du prototype. Par conséquent, nous avons défini comme objectif une application permettant de choisir les parents et de créer un enfant. La possibilité de sauvegarder l'enfant serait appréciée, mais pas indispensable. Une description plus détaillée de la gestion du projet figure au chapitre 6.

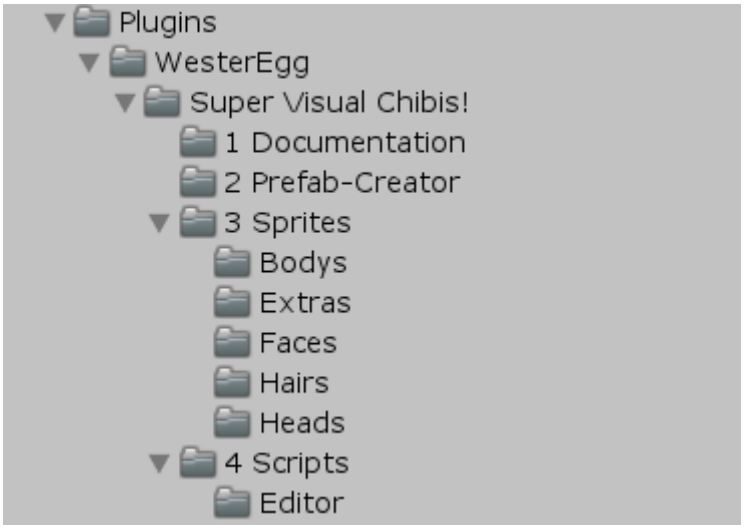
3.2. Étude de l'asset Super Visual Chibis

Dans un premier temps, nous avons étudié cet *asset* dans le but de comprendre son fonctionnement pour nous permettre d'afficher une représentation des individus à observer dans le jeu. Pour ce faire, il a été téléchargé et importé dans un projet pilote qui a ensuite été converti en projet de développement.

3.2.1. Structure

L'asset en question est composé de 4 sous-dossiers représentés dans la figure 9. Le premier contient une documentation sous la forme d'un document pdf décrivant le fonctionnement et la structure de *l'asset*. Le second contient un *prefab*, sorte de modèle d'objet réutilisable dans Unity (Unity Technologies, 2019), ainsi qu'un outil de création de *Chibi*. Le *prefab* est un *GameObject* réunissant les caractéristiques modifiables du *Chibi* : les cheveux, la tête, le corps, les sourcils, les yeux, la bouche et une caractéristique supplémentaire permettant de symboliser son humeur ou son état d'esprit, qualifiée d'extra. L'outil de création permet, comme son nom l'indique, de composer un Chibi sur la base du modèle et de l'ajouter au projet en tant que *GameObject*. Le troisième dossier contient les images et des fichiers xml permettant de lier ces images à leur nom. Le quatrième et dernier dossier contient les scripts, autrement dit le code source permettant à *l'asset* de fonctionner.

Figure 9 : Structure de l'asset importé dans Unity

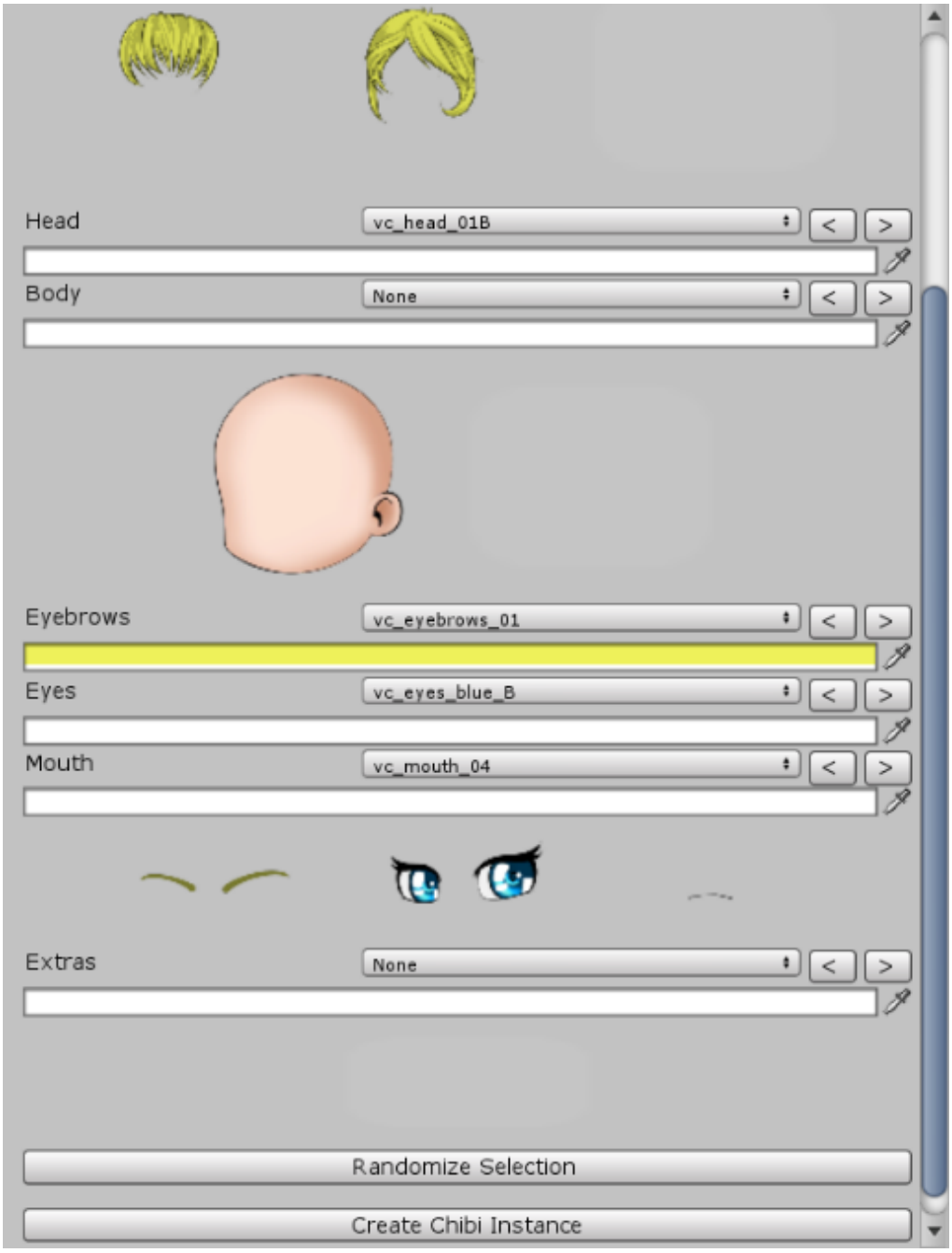


Source : données de l'auteur

3.2.2. Création et affichage d'un Chibi

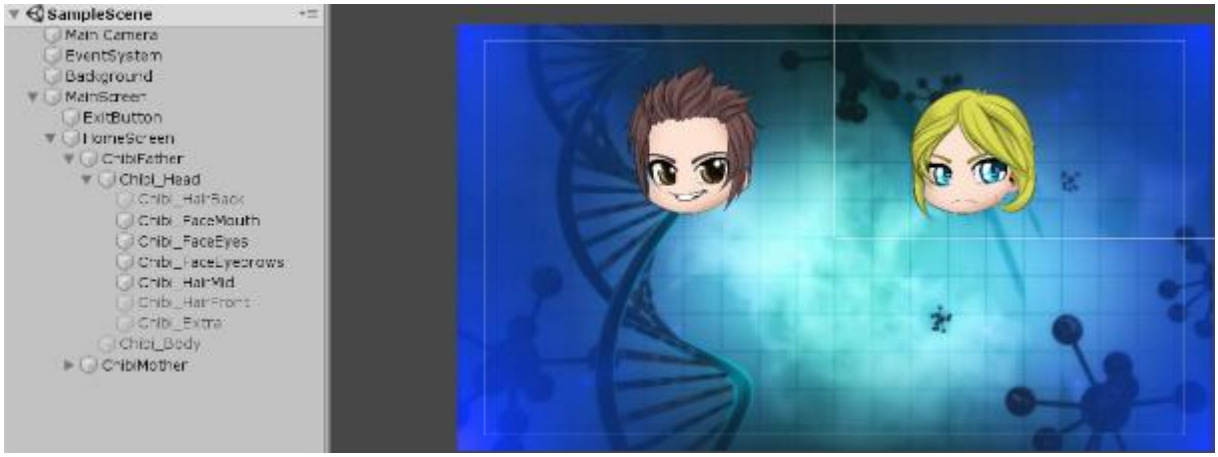
En se positionnant sur le *ChibiCreator* dans le dossier « 2 Prefab-Creator » dans l'arborescence d'Unity, nous accédons à un formulaire de création de *Chibi* proposant également un aperçu de l'apparence du personnage. Ce formulaire est représenté dans la figure 10. Les différentes caractéristiques peuvent être choisies au moyen de listes déroulantes. La couleur de certaines d'entre elles peut être sélectionnée, soit en entrant le code de la couleur (RGBA ou hexadécimal), ou encore à l'aide d'une pipette. Au bas du formulaire figurent deux boutons. L'un permet de choisir aléatoirement les caractéristiques du *Chibi* à créer. Le second instancie le *Chibi* et l'ajoute dans le projet en tant que *GameObject*, comme illustré dans la figure 11. Les caractéristiques peuvent être modifiées après la création du *Chibi*. Elles correspondent à des *Sprites*, des objets graphiques en deux dimensions dans notre cas. Celles qui ne sont pas attribuées sont affichées en grisé, comme le corps dans notre exemple.

Figure 10 : Formulaire de création de Chibi dans Unity.



Source : données de l'auteur

Figure 11 : Chibis générés via l'outil de création.

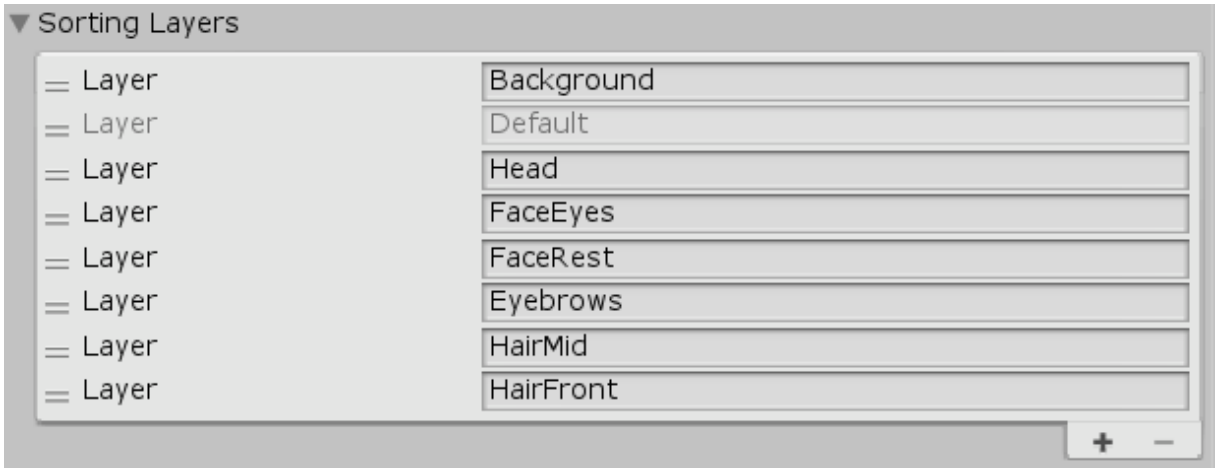


Source : données de l'auteur.

3.2.3. Gestion de la superposition des couches

Lors des essais de l'asset, nous avons constaté quelques problèmes de superposition des éléments composant les *Chibis*. Il arrivait parfois que certains d'entre eux ne soient pas visibles, car ils étaient placés en arrière-plan par rapport à un autre élément empêchant leur affichage. Nous avons donc créé des couches de tri (*sorting layers*) et les avons attribuées aux différentes caractéristiques en nous fiant aux recommandations figurant dans la documentation.

Figure 12 : Couches de tri, la première étant à l'arrière-plan, la dernière à l'avant-plan.



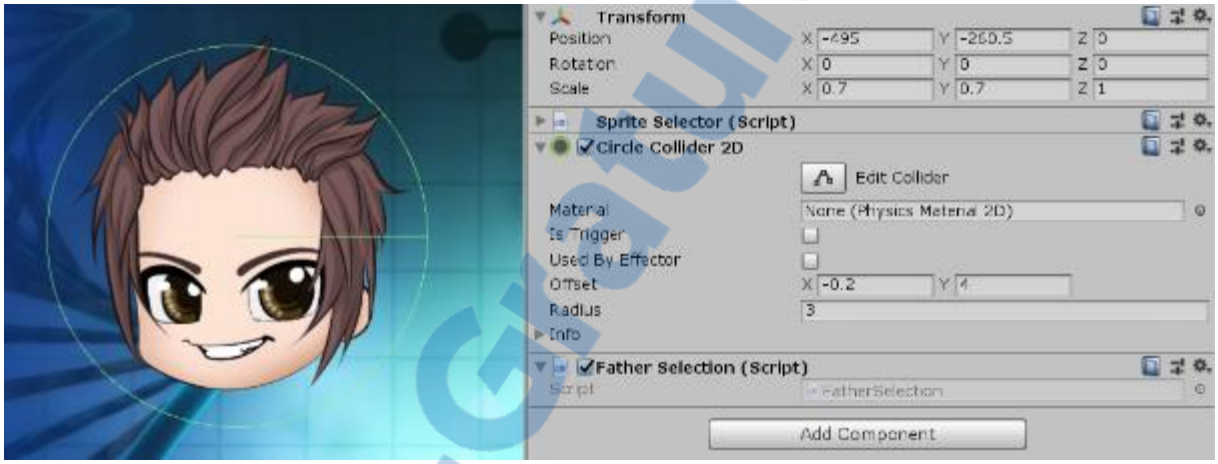
Source : données de l'auteur, adaptées de la documentation de l'asset

3.2.4. Manipulation du Chibi

Après avoir créé notre premier *Chibi*, nous avons cherché à savoir quels types de manipulations étaient possibles et les liens avec le code source. Pour commencer, nous avons rendu le *Chibi* cliquable, afin d'entraîner une action après l'événement « clic ». En effet, seuls les boutons disposent

d'un script avec la méthode *OnClick*(). Pour ce faire, il a fallu ajouter un *Collider* autour du *Chibi*. Il s'agit d'un composant permettant de définir la forme d'un objet dans Unity afin de gérer des collisions. Bien que cela ne soit pas utile dans le cadre de notre projet, il est tout de même nécessaire de déterminer la zone dans laquelle le clic doit être considéré, autrement dit, la forme du *Chibi*. Celle-ci est représentée par un cercle vert (*Circle Collider 2D*) autour de la tête du *Chibi* dans la figure 13.

Figure 13 : *Collider* permettant un clic sur un *Chibi*



Source : données de l'auteur

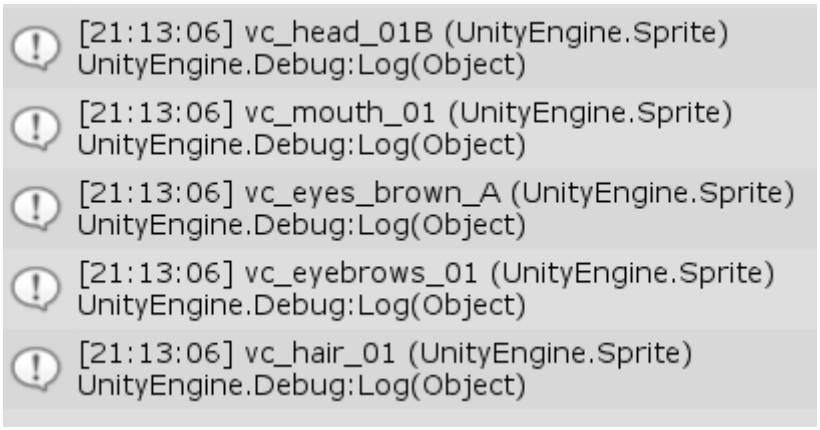
Ensuite, nous avons assigné un script à ce *GameObject* pour y entrer le code correspondant à l'action de cliquer sur la zone précédemment définie à l'aide de la méthode *OnMouseDown*(). Toujours dans le but de réaliser des tests, nous avons cherché à récupérer et stocker le nom des *Sprites* qui composent le *Chibi* et à les afficher dans la console d'Unity. Le code correspondant est représenté dans la figure 14, et le résultat obtenu dans la figure 15.

Figure 14 : Code permettant de récupérer le nom du Sprite correspondant à chaque caractéristique du Chibi en cliquant dessus.

```
private void OnMouseDown()
{
    Component[] components;
    components = this.GetComponentsInChildren<SpriteRenderer>();

    // Get sprite name for each caracteristic
    foreach(SpriteRenderer s in components)
    {
        Debug.Log(s.sprite);
    }

}
```

Source : données de l'auteur

Figure 15 : Affichage de la console.



Source : données de l'auteur.

Une fois cette étape réalisée, nous avons cherché à modifier le *Chibi*. La nouvelle action entraînée par un clic sur le *Chibi* dans le but de tester ce type de manipulation est le changement de la couleur des yeux en bleu. Pour ce faire, nous avons suivi les instructions de la documentation et coché le paramètre « *Enable changing without editor in play* » afin de permettre l'accès à la classe *ChibiCreator* depuis le code. Après étude de la classe *SpriteSelector*, nous en avons déduit le code qui permettrait de changer la couleur des yeux du *Chibi* en cliquant dessus. Celui-ci est illustré dans la figure 16. Tout d'abord, il faut récupérer le *SpriteSelector* du *Chibi* en question. Cette classe contient toutes les caractéristiques (*Sprites*) et les méthodes permettant de les modifier, ainsi que la classe *ChibiCreator* faisant le lien avec les listes de caractéristiques. Ainsi, la dernière ligne de cette méthode signifie que l'on attribue la première occurrence de la liste de couleur des yeux (le bleu) au Sprite correspondant à la couleur des yeux du *Chibi* sur lequel on clique.

Figure 16 : Code permettant de modifier la couleur des yeux du *Chibi* en bleu.

```
private void OnMouseDown()
{
    // Set new eyes color (blue)
    SpriteSelector selector = this.GetComponent<SpriteSelector>();
    ChibiCreator creator = selector.chibiCreator;

    selector.faceEyesRenderer.sprite = creator.FaceEyesSprites[1];

}
```

Source : données de l'auteur.

Dans un deuxième temps, nous avons également essayé de réaffecter des *Sprites* d'un sélecteur à un autre. La méthode présentée dans la figure 17 permet d'utiliser les *Sprites* d'un *Chibi* et de les affecter à un second sans utiliser la liste contenue dans *ChibiCreator*.

Figure 17 : Code permettant d'affecter les *Sprites* par leur nom

```
private void SetParentSprites(SpriteSelector sourceSelector, SpriteSelector destSelector)
{
    destSelector.headRenderer.sprite = sourceSelector.headRenderer.sprite;
    destSelector.faceEyesRenderer.sprite = sourceSelector.faceEyesRenderer.sprite;
    destSelector.faceEyebrowsRenderer.sprite = sourceSelector.faceEyebrowsRenderer.sprite;
    destSelector.faceEyebrowsRenderer.color = sourceSelector.faceEyebrowsRenderer.color;
    destSelector.hairMidRenderer.sprite = sourceSelector.hairMidRenderer.sprite;
    destSelector.hairMidRenderer.color = sourceSelector.hairMidRenderer.color;
    destSelector.hairBackRenderer.sprite = sourceSelector.hairBackRenderer.sprite;
    destSelector.hairBackRenderer.color = sourceSelector.hairBackRenderer.color;
    destSelector.faceRestRenderer.sprite = sourceSelector.faceRestRenderer.sprite;
}
```

Source : données de l'auteur

3.3. Création de l'écran d'accueil

Après avoir effectué ces tests, nous avons procédé à la création du premier écran à afficher après le démarrage de l'application, soit l'écran d'accueil. Celui-ci contient une représentation du père, une autre de la mère, et une liste d'enfants. Cette dernière a été ajoutée plus tard dans le développement et sa création a été détaillée au chapitre 5.4.

3.3.1. Création des parents

Afin de permettre la sélection des parents, il est nécessaire de créer un échantillon de pères et de mères à faire figurer sur l'écran d'accueil. Selon le cahier des charges, on doit proposer cinq possibilités pour chacun des parents. Avant de choisir leurs caractéristiques physiques, nous avons procédé à un état des lieux des caractéristiques des parents et des résultats possibles pour les enfants qui est présenté dans le tableau 4. On peut constater que le nombre de possibilités pour les enfants est plus important que le nombre de caractéristiques de la génération parentale, à l'exception du sexe. Ceci s'explique par le fait qu'une combinaison de gènes différents peut donner un résultat intermédiaire, comme par exemple un gène pour une peau blanche apparié à un gène pour une peau noire correspond à une peau métissée.

Tableau 4 : Liste des caractéristiques physiques des parents et des résultats possibles pour les enfants.

Caractéristiques physiques	Caractéristiques des parents	Résultats possibles
Couleur de la peau	Blanche Noire	Blanche Métissée noire et blanche Noire
Couleur des yeux	Bleu Brun Noir	Bleu clair Vert Bleu foncé Brun clair Brun foncé Noir
Couleur des cheveux	Blond Brun Noir	Blond Brun clair Brun foncé Noir
Forme de la bouche	Forme 1 Forme 2	Forme 1 Forme 2 Forme 3
Sexe	Homme Femme	Homme Femme

Source : données de l'auteur adaptées selon le cahier des charges fourni et le contenu de l'asset

À partir des données du tableau 4, nous avons fait le lien entre les caractéristiques nécessaires et le contenu de *l'asset*. Le tableau 5 présente la correspondance entre chacune d'entre elles et les noms et index des *Sprites* correspondants. Certains liens ont été établis de manière arbitraire, comme par exemple pour la couleur des yeux. Étant donné qu'il n'y avait pas de nuances de brun dans le contenu de l'asset, la couleur orange a été choisie pour représenter le brun clair, et le pourpre pour le brun foncé.

Afin de différencier les *Chibis* masculins et féminins, nous avons choisi de nous fonder sur la forme des yeux et sur la coupe de cheveux. Les *Sprites* des yeux dont le nom se termine par B correspondent à des représentations dont les cils sont plus prononcés, et dont l'apparence est donc plus féminine. La coiffure a également été choisie de manière à donner une apparence plus masculine, respectivement plus féminine au *Chibi*. Un exemple d'apparence est représenté dans la figure 18.

Tableau 5 : Liste des caractéristiques nécessaires pour les parents et les enfants.

Caractéristiques physiques	Résultats possibles	Nom du Sprite	Index du Sprite ou code couleur rgba
Couleur de la peau	Blanche	vc_head_01B	1
	Métissée noire et blanche	vc_head_01C	2
	Noire	vc_head_01D	3
Couleur des yeux	Bleu	vc_eyes_blue_A/B	0/1
	Bleu clair	vc_eyes_lightblue_ A/B	20/21
	Vert	vc_eyes_green_ A/B	12/13
	Brun foncé	vc_eyes_brown_ A/B	4/5
	Brun clair	vc_eyes_oranje_ A/B	24/25
	Noir	vc_eyes_gray_ A/B	16/17
Couleur des cheveux	Blond	-	0.93,0.94,0.35,1
	Brun clair	-	0.52,0.38,0.38,1
	Brun foncé	-	0.2,0.15,0.15,1
	Noir	-	0.18,0.15,0.15,1
Forme de la bouche	Forme 1	vc_mouth_07	6
	Forme 2	vc_mouth_06	5
	Forme 3	vc_mouth_01	0
Sexe	Homme	vc_eyes_****_A	Varie selon la couleur
		vc_hair_13	12
	Femme	vc_eyes_****_B	Varie selon la couleur
		vc_hair_14	13
		vc_hairback _12	11

Source : données de l'auteur

Figure 18: *Chibi* masculin à gauche et *Chibi* féminin à droite.

Source : données de l'auteur

À partir de ces informations, nous avons choisi les caractéristiques de cinq représentants masculins et cinq représentants féminins qui serviront de choix pour les parents. Leurs attributs ont été regroupés dans les tableaux 6 et 7, en tenant compte également de la forme et de la couleur des sourcils. Il s'agit d'un attribut qui ne fait pas partie des caractéristiques à observer par la suite, mais qui nécessite d'être paramétré pour l'affichage. C'est pourquoi nous avons choisi une forme simple ainsi qu'une couleur assortie à celle des cheveux pour plus de commodité. Le résultat visuel de ces caractéristiques est présenté dans les figures 19 et 20.

Tableau 6 : Choix des caractéristiques pour l'échantillon de pères.

#	Couleur de peau	Couleur des yeux	Couleur des cheveux	Forme de la bouche
1	Blanche	Bleus	Bruns	Forme 1
2	Blanche	Bleus	Blonds	Forme 2
3	Noire	Noirs	Noirs	Forme 1
4	Blanche	Bruns	Bruns	Forme 2
5	Blanche	Noirs	Blonds	Forme 1

Source : données de l'auteur

Figure 19 : Échantillon de pères

Source : données de l'auteur

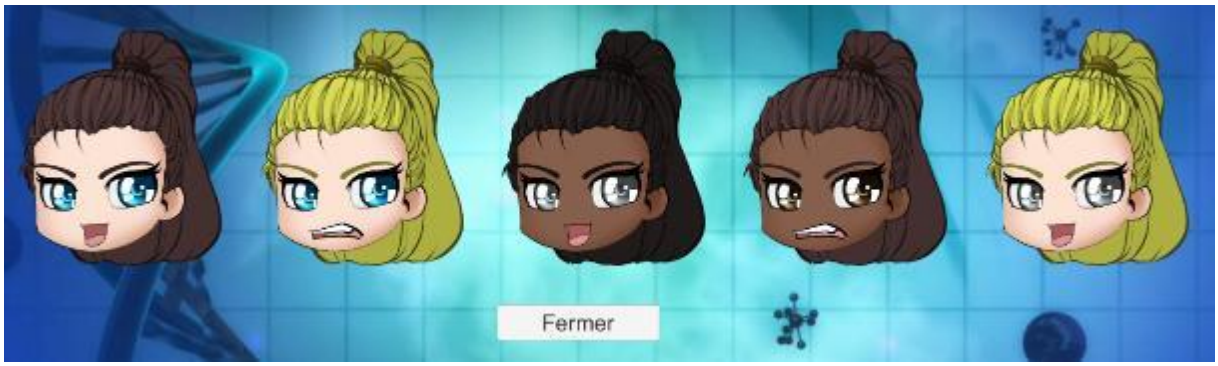
Tableau 7: Choix des caractéristiques pour l'échantillon de mères

#	Couleur de peau	Couleur des yeux	Couleur des cheveux	Forme de la bouche
1	Blanche	Bleus	Bruns	Forme 2
2	Blanche	Bleus	Blonds	Forme 1
3	Noire	Noirs	Noirs	Forme 2
4	Noire	Bruns	Bruns	Forme 1
5	Blanche	Noirs	Blonds	Forme 2

Source : données de l'auteur

Dans le cas des *Chibis* féminins, les cheveux sont constitués de deux *Sprites* : un pour les cheveux sur le crâne (*midhair*), et un pour la queue de cheval (*backhair*).

Figure 20 : Échantillon de mères



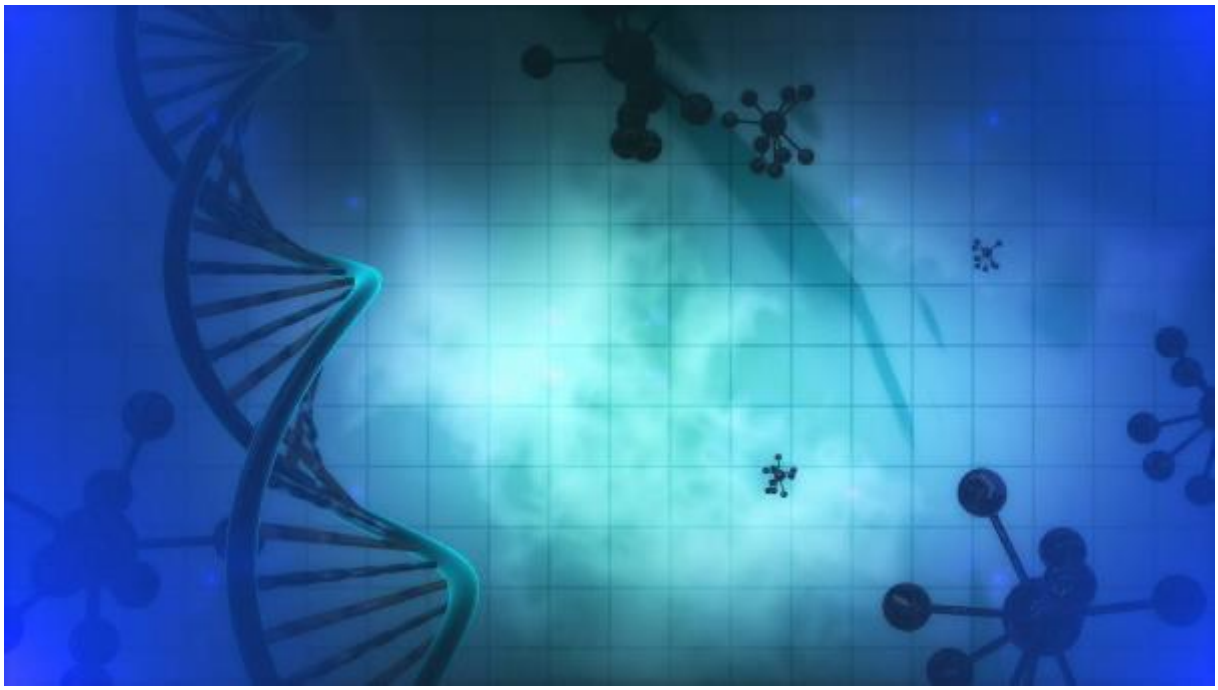
Source : données de l'auteur

Ces caractéristiques ont été regroupées dans la classe *Parents*.cs sous forme de listes publiques et statiques.

3.3.2. Création de l'interface graphique

Pour commencer, nous avons recherché un fond d'écran en accord avec le thème de l'application : la génétique. Nous avons parcouru plusieurs sites proposant des images libres de droit avant de choisir celui qui est présenté dans la figure 21. Elle montre l'écran d'accueil tel qu'il apparaît au démarrage de l'application.

Figure 21: Fond d'écran de l'application



Source : https://pixabay.com/illustrations

Pour construire cet écran, nous avons créé un canevas (*RootCanvas*) qui contiendrait tous les éléments communs de la scène (*MainScene*), tels que l'image d'arrière-plan et le bouton permettant de quitter l'application. Ensuite, nous avons ajouté un canevas (*HomeScreenCanvas*) contenant tous les éléments de l'écran d'accueil, c'est-à-dire la représentation des parents, un *Panel* proposant les échantillons de pères ou de mères, ainsi que le bouton permettant d'accéder à l'écran suivant. La position des composants a été définie à l'aide d'un *RectTransform* et/ou d'un *GridLayout* pour qu'elle s'adapte à différentes résolutions d'écran.

Au démarrage de l'application, le père et la mère sont neutres afin que l'utilisateur fasse un choix parmi les parents proposés et qu'il n'accepte pas simplement une proposition par défaut. En cliquant sur le père, respectivement la mère, on ouvre un panneau proposant un choix de cinq *Chibis*. Le contenu du panneau change en fonction du *Chibi* sur lequel on a cliqué : ils sont masculins si on a cliqué sur le père, et féminins si on a cliqué sur la mère. Tant que les deux parents n'ont pas été

choisis, le bouton permettant d'accéder à l'écran suivant ne s'affiche pas. Le résultat final est présenté à la figure 22.

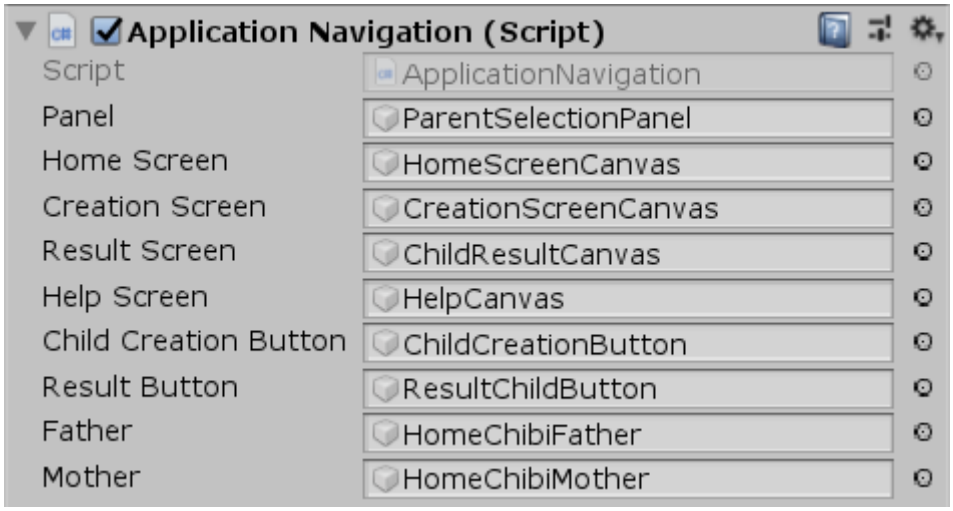
Figure 22: Écran d'accueil avec des *Chibis* neutres

Source : données de l'auteur

3.3.3. Création des scripts

En parallèle à la création de l'interface graphique de l'écran d'accueil, nous avons créé des scripts permettant de naviguer dans l'application et de modifier l'écran. En premier lieu, nous avons créé un script nommé *ApplicationNavigation*.cs pour permettre d'afficher ou de masquer les éléments au démarrage. Pour ce faire, nous avons déclaré des variables publiques de type *GameObject*. De cette façon, elles sont accessibles dans l'inspecteur d'Unity. Pour les affecter, il suffit de glisser déposer les objets dans le champ correspondant, comme illustré dans la figure 23.

Figure 23: Affectation des variables publiques depuis l'inspecteur



Source : données de l'auteur

Par défaut, ce script qui hérite de *MonoBehaviour* comporte deux méthodes : *Start*() et *Update*(). Dans la méthode *Start*(), nous avons masqué les éléments à l'exception des deux parents. Dans la méthode *Update*(), nous avons mis une condition à l'affichage du bouton permettant d'accéder à l'écran suivant. Cette méthode est appelée plusieurs fois par seconde par le programme et permet donc d'observer d'éventuelles modification de l'environnement. Par défaut, les cheveux des deux *Chibis* sont blanc. Tant que la couleur de cheveux des deux *Chibis* n'a pas changé, le bouton reste masqué. La seule façon d'introduire ce changement, c'est de choisir parmi les parents proposés.

Nous avons ensuite créé le script *ParentCreation*.cs, afin d'afficher le panneau proposant l'échantillon de parents et dont le contenu s'adapte en fonction du parent sur lequel on a cliqué. Ce script a été lié au *GameObject* du père ainsi qu'à celui de la mère. La variable publique *currentParent* permet d'adapter le comportement de ce script en fonction de son affectation. Il comporte trois méthodes. La première, *OnMouseDown*() définit l'action qui suit le clic sur l'objet. Elle appelle les deux autres méthodes, *CreateParents*() et *ShowPanel*(). La première des deux permet d'afficher les cinq *Chibis* masculins ou féminins définis au chapitre 3.3.1. Il y a également été indiqué que ce qui différencie un *Chibi* masculin d'un *Chibi* féminin est la forme des yeux. Par conséquent, nous avons mis en place une condition sur le nom du *Sprite* correspondant aux yeux. Si celui-ci se termine par « A », la suite de la méthode sera appliquée pour le père, sinon elle sera appliquée pour la mère. La dernière méthode, *ShowPanel*(), affiche le panneau contenant l'échantillon de parents et masque le père et la mère.

Le script *ParentSelectionPanelBehaviour*.cs permet, quant à lui, de fermer ce panneau. Il comporte deux méthodes, *OnMouseDown*() et *HidePanel*(). La première méthode est une méthode privée appelant la seconde de façon à masquer le panneau et afficher les parents lorsque l'un des cinq *Chibis* est sélectionné. Elle fait l'exact contraire de la méthode *ShowPanel*() du script décrit au

paragraphe précédent. *HidePanel*() a été déclarée comme publique afin de pouvoir l'affecter au bouton « Fermer » du panneau.

Le dernier script créé pour cet écran est *ParentSelection*.cs. Il est utilisé lorsque l'on clique sur l'un des cinq *Chibis* du panneau et est affecté à chacun des cinq *GameObjects* le composant. Il ne comporte qu'une seule méthode, *OnMouseDown*(), qui définit le sexe du *Chibi* sur lequel on a cliqué en vérifiant la dernière lettre du nom du S*prite* correspondant aux yeux. En fonction de la réponse, son apparence est appliquée au père ou à la mère.

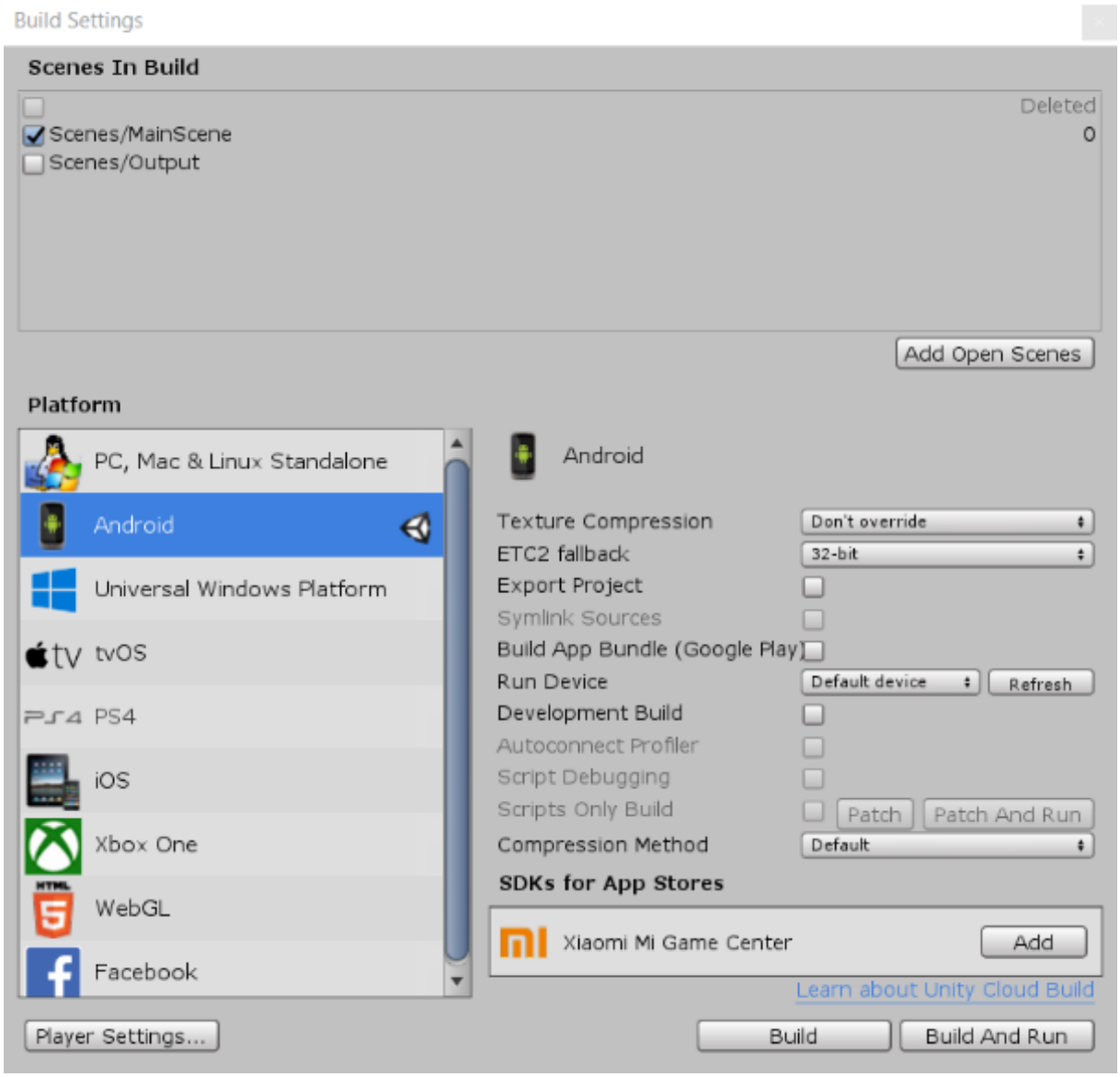
3.4. Test de l'application sur un appareil Android

Selon la documentation, Unity permet d'émuler le comportement d'un appareil Android, mais également de compiler et d'exécuter une application sur un appareil connecté à l'ordinateur via un câble USB. Pour ce faire, il faut configurer les *build settings* ainsi que les *player settings* (Unity Technologies, 2019).

3.4.1. Build settings

Nous pouvons accéder aux paramètres de compilation en cliquant sur *File* puis *Build settings*. La fenêtre concernée est présentée dans la figure 24. Il est important de sélectionner la plateforme Android dans la liste, ainsi que les scènes à intégrer à *l'Android Package Kit* (APK). Les autres paramètres ont été laissés par défaut.

Figure 24: Build settings



Source : données de l'auteur

3.4.2. Player settings

Ensuite, nous avons paramétré les *player settings* en cliquant sur le bouton correspondant sur la fenêtre des *build settings*, en bas à gauche. Il s'agit en réalité d'un sous-menu des paramètres du projet, illustré dans la figure 25.

Figure 25: Player settings

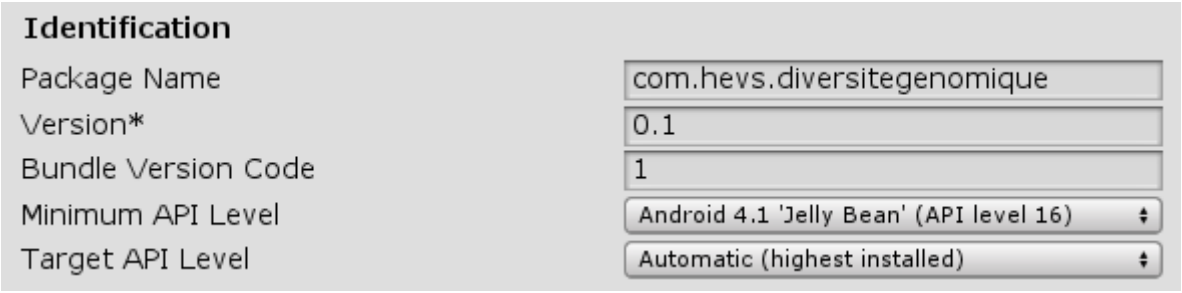
Source : données de l'auteur, l'icône provient de https://icons8.com

Dans la partie supérieure, nous avons entré le nom de l'école, le nom de l'application, choisi une icône et défini une version.

Dans le sous-menu *Resolution and Presentation*, nous avons autorisé la rotation automatique et bloqué l'orientation de l'application sur paysage droit et gauche.

Le sous-menu *Other Settings* comporte plusieurs sections, dont une partie liée à l'identification présentée dans la figure 26. Nous avons ajouté un nom de package et laissé les options relatives au niveau de l'API par défaut.

Figure 26: Paramètres d'identification de l'application



Source : données de l'auteur

3.4.3. Exécution de l'application sur Android

Une fois les paramètres configurés, nous avons activé le débogage USB sur la tablette fournie par la HES-SO puis installé et exécuté l'application pour vérifier son comportement. Nous n'avons pas relevé d'anomalie et avons poursuivi le développement en testant régulièrement l'application, à la fois sur l'émulateur et sur la tablette.

3.5. Écran de création de l'enfant

Cet écran est destiné à créer un enfant en glissant-déposant les gènes des parents pour constituer une représentation de son code génétique. Afin de permettre la navigation entre les deux écrans, une méthode supplémentaire a été ajoutée au script *ApplicationNavigation*.cs, *ShowCreationScreen*(). Elle masque tous les écrans à l'exception de celui de création de l'enfant.

3.5.1. Création de l'interface graphique

Afin de bien séparer les deux écrans, un nouveau canevas a été créé : *CreationScreenCanvas*. Nous avons scindé ce canevas en deux parties : un panneau supérieur pour y faire figurer les parents, et un panneau inférieur pour afficher les gènes.

Nous avons placé deux *Chibis* représentant les parents dans le panneau supérieur, mais sans *Collider* cette fois. Nous avions pensé réutiliser les *Chibis* de l'écran d'accueil en désactivant les *Colliders* pour qu'ils ne soient plus modifiables une fois le choix de leur apparence validé, mais cela génère une erreur car le script qui leur est attaché nécessite qu'ils soient actifs. En outre, utiliser deux canevas distincts permet de bien séparer les deux écrans.

Dans la partie inférieure de l'écran, nous avons divisé le panneau en trois, pour le père, l'enfant et la mère, et chaque partie accueille une grille de dix cases représentant les gènes.

3.5.2. Affichage des gènes

Dans un premier temps, les gènes étaient représentés par des miniatures des *Sprites*. En raison d'un manque de précision dans le descriptif du prototype, nous avions opté pour cette méthode afin que la conception d'un enfant soit plus visuelle. Cependant, lors de la validation du sprint concerné, l'étudiant en Master a demandé que les allèles soient représentés par des lettres, afin d'empêcher les utilisateurs d'anticiper l'apparence qu'aura l'enfant en fonction des gènes choisis. Nous nous sommes référés à la table de codage figurant en annexe. À partir de celle-ci, nous avons établi une liste dans le tableau 8.

Tableau 8 : Liste des gènes possibles pour les caractéristiques physiques choisies.

Caractéristique physique	Gènes possibles
Couleur de la peau	b (blanche), n (noire)
Couleur des yeux	bl (bleu), br (brun), no (noir)
Couleur des cheveux	bd (blond), br (brun), NO (noir, dominant)
Forme de la bouche	f1 (forme 1), f2 (forme 2)
Sexe	x (gène féminin), y (gène masculin)

Source : données de l'auteur adaptées du cahier des charges de l'étudiant en Master

À partir de ces gènes, la combinaison de deux d'entre eux donnera un résultat physique résumé dans le tableau 9. L'ordre de la combinaison n'a pas d'importance, ainsi, bleu + brun ou brun + bleu donneront tous deux des yeux verts. Il est à noter que le gène codant pour des cheveux noirs est dominant dans notre exemple. Ainsi, toute combinaison avec au moins un gène noir donnera des cheveux noirs.

Tableau 9 : Liste des combinaisons de gènes et leurs résultats physiques.

Caractéristiques physiques	Combinaison de gènes	Résultats physique
Couleur de la peau	b+b	Blanche
	b+n ou n+b	Métissée noire et blanche
	n+n	Noire
Couleur des yeux	bl+bl	Bleu clair
	bl+br ou br+bl	Vert
	bl+no ou no+bl	Bleu
	br+br	Brun clair
	br+no ou no+br	Brun foncé
	no+no	Noir
Couleur des cheveux (noir dominant)	bd+bd	Blond
	bd+br ou br+bd	Brun clair

	br+br	Brun foncé
	bd+NO ou NO+bd	Noir
	br+NO ou NO+br	Noir
	NO+NO	Noir
Forme de la bouche	f1+f1	Forme 1
	f2+f2	Forme 2
	f1+f2 ou f2+f1	Forme 3
Sexe	x+y ou y+x	Homme
	x+x	Femme

Source : données de l'auteur adaptées du cahier des charges.

En étudiant de plus près cette table de codage et lors de la réalisation de l'écran de création de l'enfant, nous avons constaté que les échantillons de pères et mères choisis ne permettent pas de proposer une diversité de gènes suffisante pour illustrer le grand nombre de possibilités chez les enfants. En effet, les individus choisis ont tous des paires de gènes identiques pour presque toutes les caractéristiques physiques, hormis les cheveux noirs et le sexe. Par exemple, en choisissant les couleurs de peau blanche et noire pour la génération parentale, le seul choix de l'utilisateur est de prendre un gène « b » pour le parent à peau blanche, et un gène « n » pour le parent à peau noire. Cela génèrera dans tous les cas un enfant à peau métissée noire et blanche. En revanche, en définissant les parents comme ayant une peau métissée noire et blanche, nous favorisons la création d'enfants à peau blanche, noire et métissée noire et blanche. La diversité sera d'autant plus flagrante que l'assortiment de parents à choix sera très semblable. Par conséquent, le choix parmi les caractéristiques physiques des parents a été revu et résumé dans le tableau 10. Nous avons choisi les couleurs et formes correspondant à des paires de gènes différents et donnant ainsi un maximum de possibilités différentes.

Tableau 10 : Caractéristiques physiques des parents.

Caractéristique physique	Couleurs ou formes choisies	Gènes	Sprite ou code couleur rgba correspondant
Couleur de la peau	Métissée noire et blanche	b+n ou n+b	vc_head_01C
Couleur des yeux	Vert	bl+br ou br+bl	vc_eyes_green_ *
	Bleu	bl+no ou no+bl	vc_eyes_blue_ *
	Brun foncé	br+no ou no+br	vc_eyes_brown_ *
Couleur des cheveux	Brun clair	bd+br ou br+bd	0.52,0.38,0.38,1
	Noir	bd+NO ou NO+bd	0.18,0.15,0.15,1
		br+NO ou NO+br	0.18,0.15,0.15,1
Forme de la bouche	Forme 3	f1+f2 ou f2+f1	vc_mouth_01

Sexe	Féminin	x+x	vc_eyes_*****_ B
	Masculin	x+y ou y+x	vc_eyes_*****_ A

Source : données de l'auteur

L'échantillon de parents a par conséquent été modifié selon les informations réunies dans les tableaux 11 et 12. Nous avons également décidé d'augmenter le nombre de parents de l'échantillon à 6, afin de proposer toutes les variantes possibles. L'interface graphique et le code source ont été modifiés en conséquence.

Tableau 11 : Choix des caractéristiques pour l'échantillon de pères.

#	Couleur de peau	Couleur des yeux	Couleur des cheveux	Forme de la bouche
1	Métissée	Vert	Brun clair	Forme 3
2	Métissée	Bleu	Brun clair	Forme 3
3	Métissée	Brun foncé	Brun clair	Forme 3
4	Métissée	Vert	Noir	Forme 3
5	Métissée	Bleu	Noir	Forme 3
6	Métissée	Brun foncé	Noir	Forme 3

Source : données de l'auteur

Tableau 12: Choix de caractéristiques pour l'échantillon de mères.

#	Couleur de peau	Couleur des yeux	Couleur des cheveux	Forme de la bouche
1	Métissée	Vert	Brun clair	Forme 3
2	Métissée	Bleu	Brun clair	Forme 3
3	Métissée	Brun foncé	Brun clair	Forme 3
4	Métissée	Vert	Noir	Forme 3
5	Métissée	Bleu	Noir	Forme 3
6	Métissée	Brun foncé	Noir	Forme 3

Source : données de l'auteur

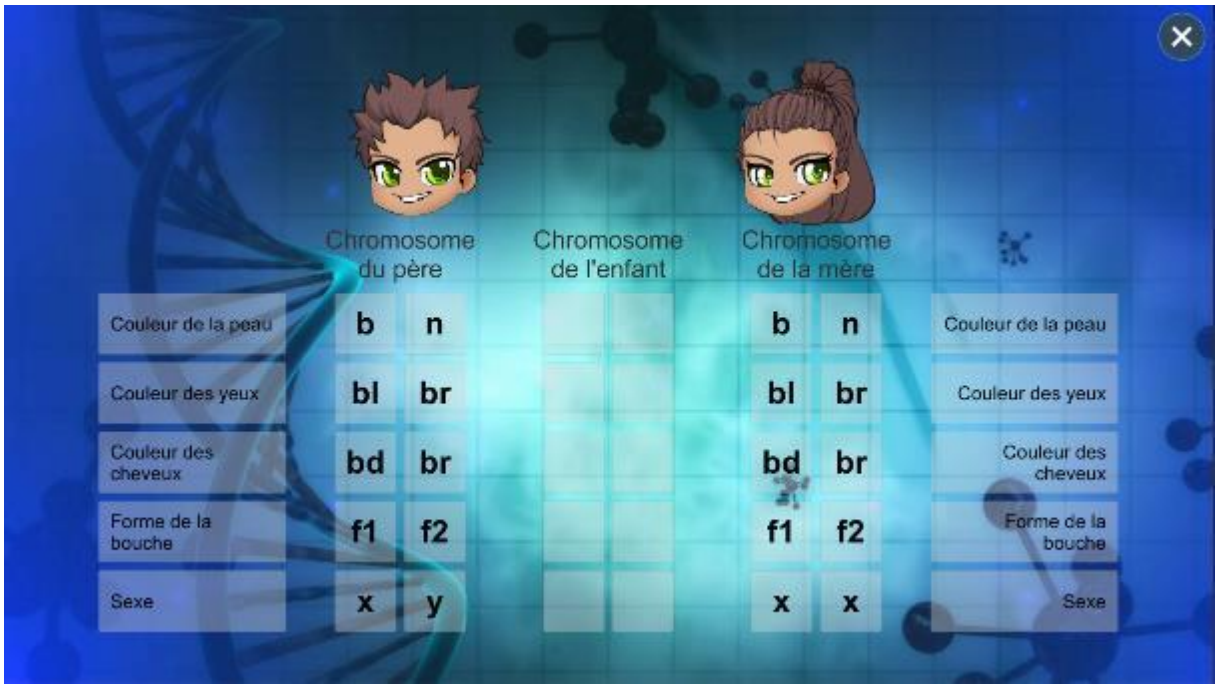
Les résultats physiques correspondants ont été réunis dans la figure 27.

Figure 27 : Apparence des parents à choix

Source : données de l'auteur

Le résultat final pour cet écran est présenté dans la figure 28 ci-dessous.

Figure 28: Écran de création de l'enfant



Source : données de l'auteur

3.5.3. Création des scripts

Afin de générer le contenu de l'écran, nous avons créé le script *ChildrenCreationBehaviour*.cs. Il continent les variables publiques *childrenCreationButton*, *selectedFather*, *selectedMother*,

destFather, *destMother*, *fatherGenes* et *motherGenes* permettant de lier les *GameObjects* nécessaires à son bon déroulement.

Il est composé de la méthode *SetContent*(), qui appelle les autres : *SetParents*(), *SetParentSprites*(…), *SetGenes*() et *SetGeneCode*(…). *SetParents*() et *SetParentSprites*(…) permettent aux *Chibis* de cet écran d'avoir la même apparence que ceux de l'écran d'accueil. *SetGenes*() et SetGeneCode(…) permettent l'affichage des gènes correspondants sur les chromosomes des parents selon la méthode décrite dans le chapitre précédent.

La correction de l'affichage des gènes selon la demande de l'étudiant en Master expliquée au chapitre précédent a également mis en évidence un élément à corriger au niveau de la création de l'échantillon des parents dans l'écran d'accueil. En effet, ceux-ci sont créés à partir d'objets contenant les indices des *Sprites*, alors qu'il faudrait les obtenir à partir de leur code génétique. Cependant, compte tenu des délais à tenir pour pouvoir procéder à l'évaluation de l'application avant la fin de l'année scolaire, nous avons choisi de poursuivre le développement du prototype en incluant une solution intermédiaire pour permettre d'obtenir le code génétique des parents dans l'écran de création de l'enfant à partir du nom des *Sprites*.

Étant donné que la démarche pour afficher l'enfant obtenu serait la même que celle pour afficher les pères et mères dans le panneau de sélection, nous avons décidé de la développer lors de la conception de l'écran de visualisation de l'enfant, pour corriger l'écran d'accueil en conséquence par la suite. Afin de permettre l'utilisation du code génétique, nous avons créé une classe permettant de stocker des listes de paires clé-valeur liant les paires de gènes au *Sprite* correspondant, *GeneCombination*.cs, dont un extrait est présenté dans la figure 29, et la classe *GeneCodeManagement*.cs. Cette classe contient plusieurs méthodes permettant d'obtenir le code génétique à partir du nom du *Sprite* ou de la couleur.

Figure 29: Extrait de la classe GeneCombination.cs

```
public static class GeneCombination
    {
        #region Skin
        public static List<KeyValuePair<string, string>> SKIN_COLOR_GENES = new
List<KeyValuePair<string, string>>
        {
            new KeyValuePair<string, string>("bb", Constants.WHITE_SKIN),
            new KeyValuePair<string, string>("bn",
Constants.MIXED_WHITE_BLACK_SKIN),
            new KeyValuePair<string, string>("nn", Constants.BLACK_SKIN)
        };

        public static List<KeyValuePair<string, int>> SKIN_COLOR_GENES_INDEX = new
List<KeyValuePair<string, int>>
        {
            new KeyValuePair<string, int>("bb", Constants.WHITE_SKIN_INDEX),
```

```
            new KeyValuePair<string, int>("bn",
Constants.MIXED_WHITE_BLACK_SKIN_INDEX),
            new KeyValuePair<string, int>("nn", Constants.BLACK_SKIN_INDEX)
        };
        #endregion

…
    }
```

Source : données de l'auteur

Afin de stocker les deux allèles, nous avons également ajouté une classe, *GenePair*, qui est un objet ayant deux attributs, *gene1* et *gene2*.

Ainsi, les allèles des parents sont récupérés à partir du nom des *Sprites* utilisés pour l'affichage. Cependant, cette méthode comporte un inconvénient : étant donné le caractère dominant du gène pour les cheveux noirs « NO », il y a trois possibilités différentes correspondant aux cheveux noirs, et la méthode utilisée ici retourne toujours la combinaison « NO + NO ». Cela implique que tous les enfants créés à partir de parents aux cheveux noirs auront également les cheveux noirs. Cela va à l'encontre de l'objectif de l'application et nécessitera une correction aussi vite que possible.

3.5.4. Implémentation du glisser-déposer

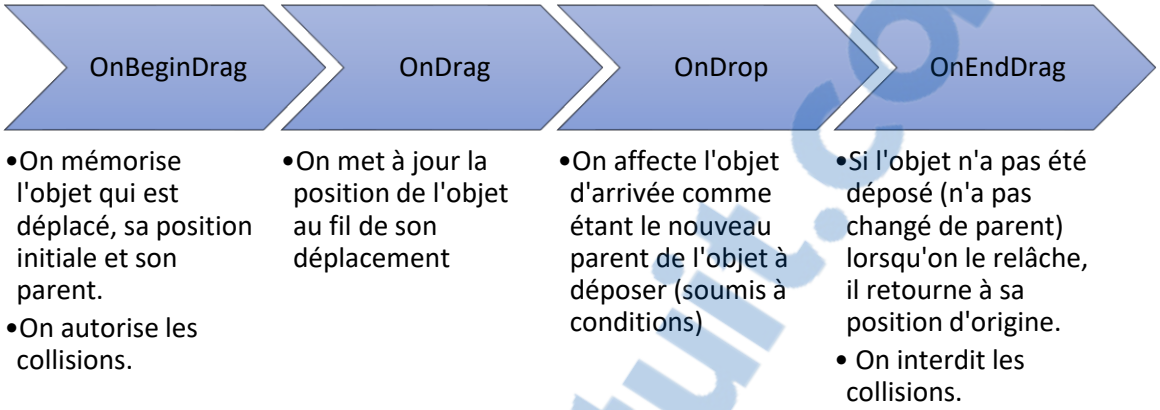
Selon les prescriptions du cahier des charges fourni au début de ce projet, la composition du code génétique de l'enfant doit s'effectuer en glissant-déposant les gènes depuis les chromosomes des parents vers celui de l'enfant.

Après une lecture de la documentation existante sur le sujet, nous avons décidé d'utiliser les interfaces *IBeginDragHandler*, *IDragHandler*, *IEndDragHandler* et *IDropHandler* comme décrit dans un exemple présenté par Kiwasi Games (Games, 2014).

Il s'agit de créer deux classes, pour gérer le glisser (*drag*), respectivement le déposer (*drop*). La première, que nous avons nommée *DragHandler*.cs doit être affectée à tous les objets qui seront glissés-déposés, en l'occurrence, les *GameObjects* correspondant aux codes des gènes chez les parents. La seconde, *DropHandler*.cs, doit être affectée à tous les objets qui accueilleront le glisser-déposer, autrement dit, les cases se trouvant sur le chromosome de l'enfant.

Dans *DragHandler*,cs, l'implémentation des interfaces mentionnées dans le paragraphe précédent permet de redéfinir trois méthodes correspondant aux étapes du glisser : *OnBeginDrag*, *OnDrag* et *OnEndDrag*. *DropHandler*.cs ne nécessite, quant à elle, qu'une seule méthode pour un cas d'utilisation simple : *OnDrop*. Ces méthodes sont utilisées dans l'ordre suivant : *OnBeginDrag*, *OnDrag*, *OnDrop* et *OnEndDrag* et sont décrites dans la figure 30. La gestion des collisions mentionnée plus bas permet au gène d'être accueilli dans le *GameObject* de destination.

Figure 30: Actions constituant un glisser-déposer



Source : données de l'auteur, inspiré de l'exemple utilisé (Games, 2014)

Comme précisé dans la figure 30, l'action de déposer l'objet est soumise à conditions. La première est qu'il ne doit pas y avoir d'objet présent dans la case de destination. Les suivantes concernent des restrictions à appliquer pour éviter des aberrations en rapport à la génétique, et sont spécifiques à notre cas d'utilisation. En effet, lors de la conception d'un enfant, celui-ci reçoit la moitié de son matériel génétique de la mère, et la seconde moitié de son père. Par conséquent, il n'est pas possible de glisser-déposer un gène si l'autre allèle composant la paire a déjà été transmis. De plus, les gènes doivent être déposés dans des cases du même type : par exemple la couleur des yeux. Ainsi, il n'est pas autorisé de glisser-déposer un gène dans une case située sur une autre ligne. Enfin, un gène qui a été déposé sur le chromosome de l'enfant ne peut retourner que sur le chromosome de son parent d'origine. De cette manière, on évite qu'un gène transmis du père à l'enfant ne soit accidentellement placé sur le chromosome de la mère lors du processus.

Afin d'appliquer ces conditions lors du *OnDrop*, nous avons ajouté des *tags* aux cases contenant les gènes (père, mère, enfant). Ils sont ensuite utilisés pour vérifier la provenance et la destination du gène glissé-déposé. Ainsi, il est possible de déposer un gène dans une case de l'enfant ou du parent d'origine uniquement. Pour contrôler le type de gène, nous vérifions que les cases d'origine et de destination portent bien le même nom, à l'exception de l'indice, de manière à ce que le gène puisse être déposé dans n'importe laquelle des cases de l'enfant correspondant au type de gène considéré. Pour terminer, afin de ne pas transmettre deux gènes du même parent, nous contrôlons que la case voisine de celle d'origine n'est pas vide avant d'autoriser le déposer.

Le code rédigé pour implémenter ces conditions est présenté dans la figure 31. Elles sont donc au nombre de quatre :

- La case de destination doit être vide
- La source et la destination doivent porter le même nom

- Le *tag* de destination doit correspondre à celui de l'enfant ou du parent d'origine
- La case voisine de celle d'origine ne doit pas être vide

Figure 31: Conditions d'autorisation du glisser-déposer

```
if (!item && sourceName.Equals(destName) && (destTag.Equals(sourceTag) ||
destTag.Equals(CHILD_TAG)) && siblingSlotChildCount > 0)
        {
            DragHandler.itemBeingDragged.transform.SetParent(transform);

        }
```

Source : données de l'auteur

3.6. Écran de visualisation de l'enfant

L'écran de visualisation de l'enfant permet d'afficher l'apparence de l'enfant constitué dans l'écran précédent et d'observer les différences ou les ressemblances avec ses parents. Comme précédemment, une nouvelle méthode a été ajoutée à *ApplicationNavigation*.cs pour permettre l'affichage de cet écran.

3.6.1. Création de l'interface graphique

Nous avons créé un Canvas nommé *ChildResultCanvas* pour accueillir ce nouvel écran. Il a été divisé en deux parties : le code génétique s'affiche dans la partie gauche, et le résultat visuel à droite. Le code génétique de l'enfant est repris depuis l'écran précédent et affiché sous forme de grille, semblable à celles utilisées dans l'écran de création. À droite, nous présentons les parents, ainsi que l'enfant. Ce dernier est mis en évidence par sa taille, plus importante que celle de ses parents. À ce stade du développement, le bouton figurant en bas à droite de l'écran permet de revenir à l'écran d'accueil. Il sera modifié plus tard pour permettre la sauvegarde du profil génétique de l'enfant. L'interface graphique de cet écran est illustrée dans la figure 32.

Figure 32: Écran de visualisation de l'enfant

Source : données de l'auteur

3.6.2. Création des scripts

Pour permettre l'affichage du contenu de l'écran, nous avons affecté un script nommé *ShowChildResult*.cs à l'écran *ChildResultCanvas*. Il est constitué des méthodes *SetContent*() et *IsMale*(string genderCode). *SetContent*() est utilisé afin de constituer l'apparence de l'enfant à partir de son code génétique. Elle utilise également la méthode *IsMale*(…) afin de différencier le père et la mère lors de l'obtention des *Sprites* correspondant aux yeux et aux cheveux.

Comme précisé dans le chapitre 3.3.1, l'ordre des allèles n'a pas d'influence sur l'apparence. Nous nous affranchissons de ce détail en triant alphabétiquement les allèles avant de les concaténer. Pour ce faire, nous avons ajouté deux méthodes dans la classe *GenePair* : ConcatenanteGenePair() et SortGeneParisAlphabetically(). Ainsi, «no » (gene1) et br » (gene2) donneront « brno », par exemple. Nous avons également créé des tests unitaires afin de vérifier le comportement de cette méthode avec plusieurs combinaisons d'allèles. Les méthodes en question sont présentées dans la figure 33.

Figure 33: Méthode permettant d'obtenir une paire de gènes à partir d'allèles individuels

```
public string ConcatenateGenePairs()
        {
            SortGenePairsAlphabetically();
            return gene1 + gene2;
        }

private void SortGenePairsAlphabetically()
```

```
        {
            List<string> list = new List<string>()
            {
                gene1,
                gene2
            };

            list.Sort();
            gene1 = list[0];
            gene2 = list[1];
        }
```

Source : données de l'auteur

La chaîne de caractère ainsi obtenue est utilisée pour obtenir le *Sprite* correspondant à afficher sur l'écran à l'aide de la classe *GeneCodeManagement.cs*. Ces informations sont récupérées au moyen de la classe *GeneCombination*.cs présentée au chapitre 3.4.2 et en se référant à la classe *Constants*.cs créée au début du projet.

Ainsi, *SetContent*() est constituée de trois parties :

- Affichage des gènes sur le chromosome de l'enfant
- Obtention du nom des *Sprites* à partir du code génétique
- Attribution des *Sprites* au *Chibi* représentant l'enfant

Afin d'affecter les *Sprites* aux *Chibis* correspondant aux parents, nous avons créé le script *TransferParentSprites*.cs. Celui-ci reprend les méthodes décrites dans le chapitre 3.5.3. En effet, il s'agit de la même manipulation réalisée afin d'afficher les parents dans l'écran de création de l'enfant. Nous avons par conséquent décidé de les centraliser dans un script, utilisé par les boutons *ChildCreationButton* et *ResultChildButton*.

À ce stade du développement, nous avons atteint l'objectif fixé pour permettre l'évaluation de l'application par un échantillon d'utilisateurs, à savoir, être en mesure de créer un enfant et de visualiser son apparence. Cependant, nous avons manqué de temps pour corriger le défaut relevé au chapitre 3.5.3, ce qui implique que la création d'enfants aux cheveux noirs est nettement favorisée. Cela introduit un biais dans l'évaluation de l'application, mais permettrait également de vérifier l'implication des testeurs.

4. Évaluation de l'application

La phase d'évaluation de l'application s'est déroulée dans le cadre de révisions de fin d'année au cycle d'orientation de St-Maurice, avec pour testeurs, une classe gérée par l'étudiant en Master collaborant à ce travail.

4.1. Organisation du test

L'organisation de cette phase a fait l'objet d'une séance avec l'étudiant en Master, pendant laquelle nous avons défini le cadre et la manière de tester le prototype. Nous avons été informés que le test ne pourrait se faire que pendant une période de révision (45 minutes)4 normalement consacrée à la préparation des examens. Par conséquent, il faudrait définir une méthode permettant aux étudiants du cycle de se consacrer à leur travail pendant le reste de la période. Nous avons convenu que le meilleur moyen de gagner du temps était de réaliser une présentation de l'application à tous les élèves, puis de constituer des groupes de trois à quatre personnes auxquels nous ferions tester le prototype. Nous obtiendrions un retour d'information en leur demandant de répondre à un questionnaire directement après avoir essayé l'application. Il est à noter que la composition exacte de la classe ne nous était pas connue à ce moment-là.

4.2. Création des formulaires

Lors de la séance d'organisation, nous avons préparé les points devant figurer dans le questionnaire d'évaluation du prototype. Nous avons convenu que l'étudiant en Master se chargerait de formuler les questions étant donné sa meilleure connaissance du public cible. Elles ont pour but d'évaluer la qualité de l'application, mais également la compréhension de son objectif par les élèves. Les questions retenues sont les suivantes :

- L'application est-elle facile d'utilisation ?
- Y a-t-il des fonctionnalités qui n'ont pas fonctionné ?
- Qu'avez-vous remarqué sur l'apparence des enfants créés ?
- Comment pourrions-nous avoir deux enfants totalement identiques ?
- Combien de temps a pris la création d'un enfant ?
- Le glisser-déposer est-il facile à manipuler ?
- Le design de l'application vous a-t-il plu ?
- Avez-vous trouvé cette approche de la génétique intéressante ? Si oui pourquoi ?
- A la suite de cette activité, la transmission génétique vous a-t-elle paru plus concrète ?
- Quel est le but de cette application ?

4.3. Préparation du test

En amont de l'évaluation de l'application, nous avons préparé une seconde tablette, afin de permettre à plusieurs groupes de travailler simultanément. Cependant, lors de l'installation de l'application sur ce nouvel appareil, nous avons constaté des problèmes d'affichage. Le jeu n'est pas centré sur l'écran, et une partie est donc inaccessible et inutilisable pour l'utilisateur, malgré la conception multi-résolution suivie et présentée dans la documentation d'Unity (Unity Technologies, 2019). Par conséquent, le test s'est déroulé sur la première tablette uniquement.

4.4. Déroulement du test

Le test s'est déroulé le mardi 11 juin de 15h20 à 16h05, avec une classe de 3ème année préparant un examen de mathématiques. Elle était constituée de 12 élèves d'une quinzaine d'années.

4.4.1. Démonstration

Comme défini au chapitre 4.1, nous avons réalisé une démonstration de l'application en affichant celle-ci à l'aide d'un *beamer*. L'étudiant en Master a décrit son utilisation pendant que nous en faisions la démonstration et a également expliqué les consignes du test.

4.4.2. Test

Ensuite, nous avons réparti les 12 étudiants présents en groupes de quatre personnes qui ont testé l'application à tour de rôle sur la tablette. Nous avons précisé le scénario en imposant à chaque membre d'un groupe de conserver les mêmes parents et de photographier l'enfant créé dans le but de les comparer d'une fois que chacun avait terminé le test. Ceci a permis de simuler la fonctionnalité manquante de sauvegarde des enfants et de leur affichage dans l'écran d'accueil. Nous avons pris quelques photos, avec le consentement des personnes présentes. L'un des groupes testant l'application est présenté dans la figure 34.

Figure 34: Évaluation de l'application par des élèves du cycle d'orientation de St-Maurice

Source : photo prise par l'auteur, avec le consentement oral des personnes concernées

4.4.3. Retour d'information

Une fois le prototype testé, nous avons distribué les formulaires permettant de récolter les retours d'information. Ceux-ci ont été remplis anonymement par les élèves, ainsi que par le maître de formation de l'étudiant en Master, présent en tant qu'observateur.

4.5. Réponses au questionnaire

Nous avons donc obtenu 13 retours, dont un venant d'un enseignant expérimenté. Les réponses sont détaillées dans les chapitres suivants.

4.5.1. Utilisation de l'application

Les 13 utilisateurs de l'application l'ont trouvée facile d'utilisation.

4.5.2. Fonctionnalités

Aucun des utilisateurs n'a constaté de problème pendant la durée de l'évaluation.

4.5.3. Apparence des enfants

Cinq personnes ont trouvé que les enfants se ressemblent, cinq personnes les ont trouvés différents, deux personnes ont trouvé qu'ils ressemblent aux parents et quatre personnes ont trouvé qu'il manquait des caractéristiques physiques (blond, peau blanche).

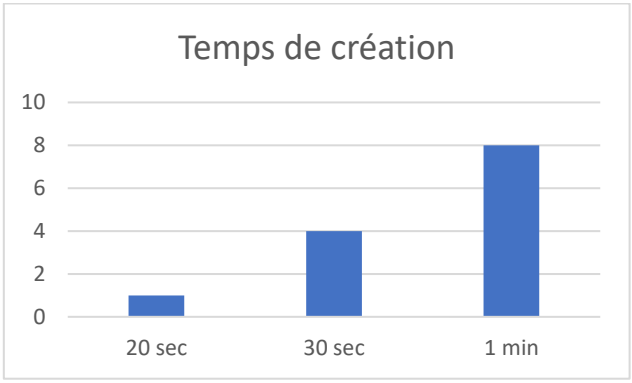
4.5.4. Enfants identiques

L'ensemble des utilisateurs a exprimé l'idée de sélectionner les mêmes gènes.

4.5.5. Temps de création

Le temps estimé par les utilisateurs varie entre 20 secondes et une minute et a été jugé satisfaisant. Nous avons porté les réponses sous forme de graphique à la figure 35.

Figure 35: Répartition du temps de création de l'enfant



Source : données de l'auteur

4.5.6. Glisser-déposer

L'ensemble des utilisateurs a trouvé le glisser-déposer facile à manipuler.

4.5.7. Design de l'application

Trois personnes trouvent l'application un peu sombre. Les dix autres ont apprécié l'apparence des personnages (*Chibis*).

4.5.8. Approche

Les utilisateurs ont trouvé cette approche intéressante, ludique et efficace. Quatre d'entre eux ont compris le caractère dominant des cheveux noirs. L'un d'entre eux a trouvé l'application trop simple. L'enseignant estime que c'est une bonne introduction à la diversité génétique et que l'affichage des enfants créés est nécessaire pour compléter le prototype.

4.5.9. Transmission génétique concrète

L'ensemble des utilisateurs estime avoir mieux compris le sujet grâce à cette application. L'un d'entre eux a fait remarquer que c'était dommage que cela se soit déroulé après l'examen de sciences.

4.5.10. But de l'application

Excepté l'enseignant, aucun des utilisateurs n'a utilisé le terme adéquat de « diversité » génétique. Tous se sont concentrés sur la génétique en général, et l'un d'entre eux a pointé le fait d'apprendre en s'amusant.

4.6. Analyse des réponses

L'étude des réponses du questionnaire montrent que l'application est rapide, facile à utiliser et fonctionne bien. Nous nous permettons cependant de pondérer cette dernière affirmation en raison du problème d'affichage constaté lors de la préparation du test.

Les remarques sur l'apparence des enfants créés montrent que le défaut de conception mentionné au chapitre 3.5.3 a bien entraîné un biais dans l'utilisation de l'application. Cependant, cela nous a également montré que les utilisateurs semblent avoir tendance à choisir des parents différents l'un de l'autre, alors que le cas de figure rendant la diversité génétique évidente serait de choisir des parents semblables. Toutefois, quatre personnes ont relevé le manque d'apparition d'enfants aux cheveux blonds, ce qui montre une certaine attente et également une compréhension intuitive de l'objectif de l'application, même si celui-ci n'a pas été exprimé explicitement en réponse à la dernière question du formulaire.

5. Suite du développement

5.1. Correction de l'écran d'accueil

Avant de poursuivre l'implémentation de nouvelles fonctionnalités, nous avons apporté les modifications nécessaires à l'utilisation du profil génétique pour générer les échantillons de pères et mères dans l'écran d'accueil.

5.1.1. Génération de l'échantillon de pères et mères

Nous avons ajouté deux listes de six profils génétiques dans la classe *Parents*.cs correspondant aux pères et aux mères, présentées dans la figure 36.

Figure 36: Listes des profils génétiques des parents

```
#region genes
        //Gene pairs : skin color, eyes color, hair color, mouth, gender
        public static List<ChibiGenes> FATHER_GENES = new List<ChibiGenes>()
        {
            new ChibiGenes("bn", "blbr", "bdbr", "f1f2", "xy"),
            new ChibiGenes("bn", "blno", "bdbr", "f1f2", "xy"),
            new ChibiGenes("bn", "brno", "bdbr", "f1f2", "xy"),
            new ChibiGenes("bn", "blbr", "brNO", "f1f2", "xy"),
            new ChibiGenes("bn", "blno", "bdNO", "f1f2", "xy"),
            new ChibiGenes("bn", "brno", "brNO", "f1f2", "xy")
        };

        public static List<ChibiGenes> MOTHER_GENES = new List<ChibiGenes>()
        {
            new ChibiGenes("bn", "blbr", "bdbr", "f1f2", "xx"),
            new ChibiGenes("bn", "blno", "bdbr", "f1f2", "xx"),
            new ChibiGenes("bn", "brno", "bdbr", "f1f2", "xx"),
            new ChibiGenes("bn", "blbr", "brNO", "f1f2", "xx"),
            new ChibiGenes("bn", "blno", "bdNO", "f1f2", "xx"),
            new ChibiGenes("bn", "brno", "brNO", "f1f2", "xx")
        };
        #endregion
```

Source : données de l'auteur

Ensuite, nous avons modifié la méthode *CreateParents*() du script *ParentCreation*.cs de façon à nous référer à ces listes pour générer l'apparence des parents dans le panneau de sélection. Nous obtenons les index des *Sprites* à afficher, ainsi que la couleur des cheveux, au moyen de nouvelles méthodes implémentées dans la classe *GeneCodeManagement.cs*, illustrées dans la figure 37.

Figure 37: Méthodes permettant d'obtenir l'apparence physique des parents à partir du code génétique

```
public int GetIndexFromGeneCode(string characteristic, string geneCode, bool isMale)
        {
```

```
            int index = -1;

            switch (characteristic)
            {
                case "skin":
                    index = GetIndex(GeneCombination.SKIN_COLOR_GENES_INDEX,
geneCode);
                    break;
                case "eyes":
                    if (isMale)
                        index = GetIndex(GeneCombination.MALE_EYE_COLOR_GENES_INDEX,
geneCode);
                    else
                        index =
GetIndex(GeneCombination.FEMALE_EYE_COLOR_GENES_INDEX, geneCode);
                    break;
                case "mouth":
                    index = GetIndex(GeneCombination.MOUTH_GENES_INDEX, geneCode);
                    break;
            }
            return index;
        }

public Color GetColorFromGeneCode(string geneCode)
        {
            List<KeyValuePair<string, Color>> list =
GeneCombination.HAIR_COLOR_GENES;
            Color color = new Color();

            foreach (KeyValuePair<string, Color> kvp in list)
            {
                if (kvp.Key.Equals(geneCode))
                {
                    color = kvp.Value;
                }
            }
            return color;
        }

private int GetIndex(List<KeyValuePair<string, int>> list, string geneCode)
        {
            int index = -1;

            foreach (KeyValuePair<string, int> kvp in list)
            {
                if (kvp.Key.Equals(geneCode))
                {
                    index = kvp.Value;
                }
            }
            return index;
        }
```

Source : données de l'auteur

5.1.2. Affichage des gènes dans l'écran de création

Pour permettre la récupération du profil génétique des parents sélectionnés, nous avons créé un script nommé *GameManager*.cs dans le but de transmettre leur index, présenté dans la figure 38. Nous nous sommes inspirés d'un scénario trouvé sur Youtube dont le but est de mettre à jour l'affichage d'un score dans l'interface graphique en transmettant une variable entre deux scripts (Unity Games Programming For Beginners, 2017). Nous avons le même besoin, à la différence que nous ne souhaitons pas afficher les variables en question, mais les utiliser dans le second script.

Figure 38: Script permettant la transmission de variables entre deux scripts

```
public class GameManager : MonoBehaviour
{
    private int mother;
    private int father;

    public int GetMother()
    {
        return mother;
    }

    public void SetMother(int index)
    {
        mother = index;
    }

    public int GetFather()
    {
        return father;
    }

    public void SetFather(int index)
    {
        father = index;
    }
}
```

Source : données de l'auteur

Afin d'accéder aux méthodes de *GameManager*.cs, il est nécessaire de le référencer dans les scripts concernés, à savoir *ParentSelection*.cs et *ChildrenCreationBehaviour*.cs, comme montré dans la figure 39.

Figure 39: Récupération de la classe GameManager dans un script

```
private GameManager gameManager;

    private void Awake()
    {
        gameManager = GameObject.FindObjectOfType<GameManager>();
    }
```

Source : données de l'auteur

Lorsqu'un parent a été choisi, son index est sauvegardé lors de la sélection du *SpriteSelector* du père, respectivement de la mère, figurant sur l'écran d'accueil. L'index correspond au dernier caractère du nom de l'objet contenant le *Chibi* choisi, moins un. Cette démarche est présentée dans la figure 40.

Figure 40: Sauvegarde de l'index

```
if (gender.Equals("A"))
        {
            selector = father.GetComponent<SpriteSelector>();
            index = Int32.Parse(this.name.Substring(this.name.Length - 1)) - 1;
            gameManager.SetFather(index);
        }
        else
        {
            selector = mother.GetComponent<SpriteSelector>();
            index = Int32.Parse(this.name.Substring(this.name.Length - 1)) - 1;
            gameManager.SetMother(index);
        }
```

Source : données de l'auteur

Nous avons corrigé le script *ChildrenCreationBehaviour* en conséquence en ajoutant une variable servant à récupérer l'index sauvegardé précédemment et en l'utilisant pour retrouver le profil génétique du parent sélectionné dans le but d'afficher son code génétique. Afin de récupérer les allèles séparément, nous avons modifié l'affectation des variables *skinPair*, *eyePair*, *hairPair*, *mouthPair* ainsi que *genderPair*. Le code corrigé est présenté dans la figure 41.

Figure 41: Correction du script ChildrenCreationBehaviour.cs

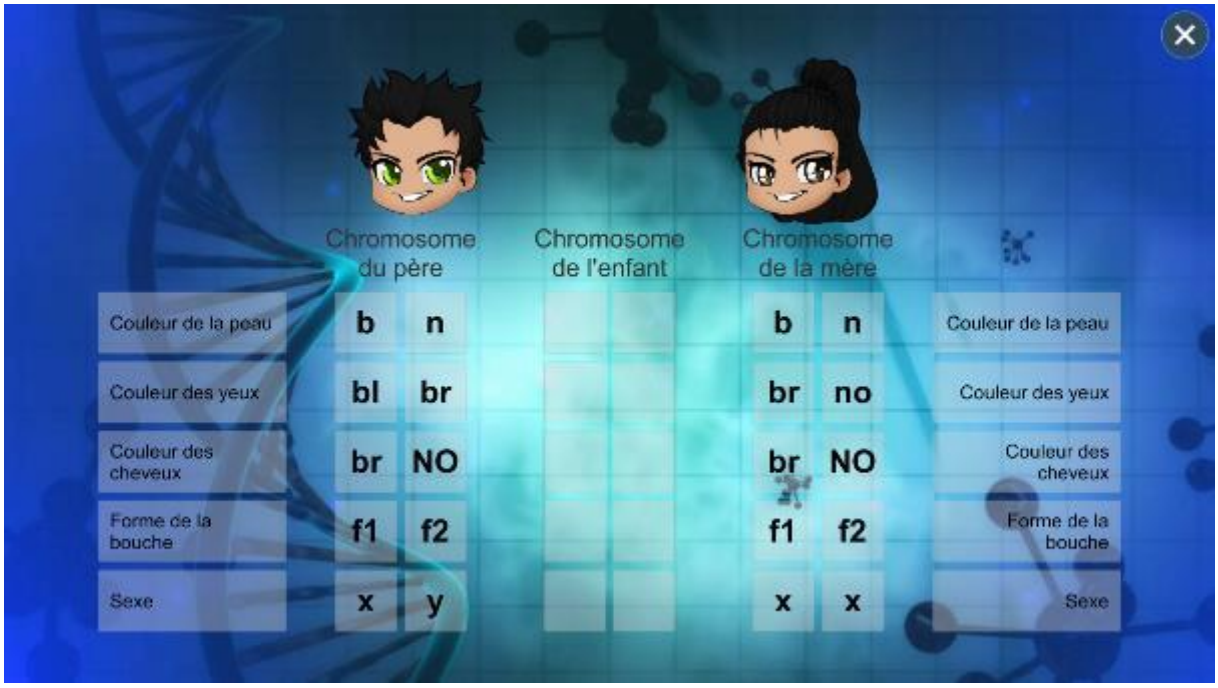
```
if (isFather)
        {
            path = "FatherGenes/";
            fatherIndex = gameManager.GetFather();
            genes = Parents.FATHER_GENES[fatherIndex];
        }
        else
        {
            path = "MotherGenes/";
```

```
            motherIndex = gameManager.GetMother();
            genes = Parents.MOTHER_GENES[motherIndex];
        }

        skinPair = skinPair.SplitGenePair(genes.SkinColor);
        GameObject.Find(path +
"SkinColorGene1/SkinColorGene1").GetComponent<Text>().text = skinPair.gene1;
        GameObject.Find(path +
"SkinColorGene2/SkinColorGene2").GetComponent<Text>().text = skinPair.gene2;

        eyePair = eyePair.SplitGenePair(genes.EyesColor);
        GameObject.Find(path +
"EyeColorGene1/EyeColorGene1").GetComponent<Text>().text = eyePair.gene1;
        GameObject.Find(path +
"EyeColorGene2/EyeColorGene2").GetComponent<Text>().text = eyePair.gene2;

        hairPair = hairPair.SplitGenePair(genes.HairColor);
        GameObject.Find(path +
"HairColorGene1/HairColorGene1").GetComponent<Text>().text = hairPair.gene1;
        GameObject.Find(path +
"HairColorGene2/HairColorGene2").GetComponent<Text>().text = hairPair.gene2;

        mouthPair = mouthPair.SplitGenePair(genes.Mouth);
        GameObject.Find(path + "MouthGene1/MouthGene1").GetComponent<Text>().text =
mouthPair.gene1;
        GameObject.Find(path + "MouthGene2/MouthGene2").GetComponent<Text>().text =
mouthPair.gene2;

        genderPair = genderPair.SplitGenePair(genes.Gender);
        GameObject.Find(path + "GenderGene1/GenderGene1").GetComponent<Text>().text
= genderPair.gene1;
        GameObject.Find(path + "GenderGene2/GenderGene2").GetComponent<Text>().text =
genderPair.gene2;
```

Source : données de l'auteur

Nous avons finalement procédé à des tests afin de vérifier le comportement de l'application, et que les paires de gènes des cheveux noirs ne correspondent plus à la paire « NO + NO ». Il est donc désormais possible de créer des enfants aux cheveux blonds ou bruns à partir de parents aux cheveux noirs, comme illustré dans les figures 42 et 43.

Figure 42: Vérification de l'affichage des gènes pour les cheveux noirs



Source : données de l'auteur

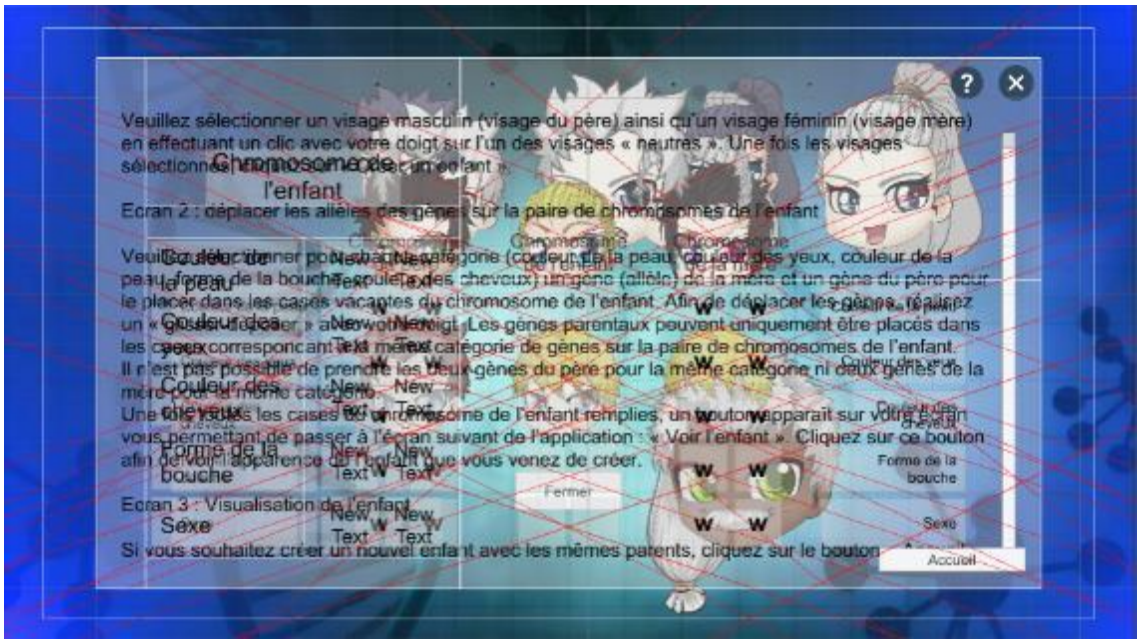
Figure 43: Enfant blond issu de parents aux cheveux noirs

Source : données de l'auteur

5.2. Mise à jour d'Unity

Avant de poursuivre le développement de l'application, nous avons procédé à la mise à jour d'Unity de la version 2018.3.5f1 à la version 2019.1.8f1 afin de bénéficier d'une nouvelle fonctionnalité simplifiant la gestion de l'interface graphique. En effet, la structure de notre application ne comporte qu'une seule scène dans laquelle nous affichons et masquons des canevas. Cependant, à ce stade du développement, il y a quatre canevas superposés : l'arrière-plan, l'écran d'accueil, l'écran de création et l'écran de visualisation. La visibilité de la scène est extrêmement réduite en raison du nombre d'éléments affichés, comme illustré dans la figure 44.

Figure 44: Superposition des canevas sur la scène de l'éditeur



Source : données de l'auteur

Jusqu'à présent, nous avons créé les nouveaux canevas sur une scène alternative, avant de les déplacer sur la scène de l'application une fois l'interface graphique jugée satisfaisante. Toutefois, une fois que des scripts sont associés aux objets, déplacer le canevas dans la seconde scène génère des erreurs dues aux références manquantes. Cette méthode n'est pas adaptée pour la modification des éléments graphiques d'un écran après avoir affecté les scripts et les variables concernés.

Nous nous sommes documentés sur la meilleure façon d'adapter la structure de l'application, en envisageant de placer les canevas dans des scènes séparées. Toutefois, cela entraîne une autre difficulté : la destruction des objets lorsque l'on navigue d'une scène à une autre, ce qui impliquerait de repenser totalement la structure de notre application. En creusant davantage, nous avons trouvé une fonctionnalité mise en place dans une version plus récente d'Unity, permettant d'afficher et de

masquer des éléments du projet dans l'éditeur : *Scene visibility* (Unity Technologies, 2019). Les différentes possibilités d'affichage sont résumées dans la figure 45.

Figure 45: Possibilités offertes par la fonctionnalité scene visibility d'Unity

A		The GameObject is visible, but some of its children are hidden.
B		The GameObject is hidden, but some of its children are visible.
C		The GameObject and its children are visible. This icon only appears when you hover over the GameObject.
D		The GameObject and its children are hidden.

Source : https://docs.unity3d.com/Manual/SceneVisibility.html

Par conséquent, après avoir fait une sauvegarde du projet, nous avons mis à jour l'éditeur à une version plus récente comportant cette fonctionnalité. Comme le montre la figure 46, la création et la maintenance des écrans a été simplifiée grâce à cette démarche. Ceci permettra également de rechercher la cause du problème d'affichage sur la seconde tablette.

Figure 46: Scène de l'éditeur après mise à jour d'Unity et masquage de certains canevas

Source : données de l'auteur

5.3. Sauvegarde des enfants

L'étape suivante de la phase de développement est de sauvegarder les enfants créés afin de les afficher sur l'écran d'accueil. Ceci permettra également d'illustrer la diversité génomique de façon visuelle.

5.3.1. Création du format de sauvegarde

Dans un premier temps, nous nous sommes documentés sur la meilleure façon de sauvegarder les enfants. Il existe deux options : créer une base de données ou sauvegarder les données dans un fichier. La conception d'une base de données nécessite de la modéliser, de la créer, d'implémenter des objets afin de la lier au projet, d'y insérer des données et de créer des requêtes afin de la lire et de la modifier. D'après une discussion sur *Stackoverflow*, cela représente une quantité de travail importante, qui se justifie pour un très grand nombre de données (des milliers), ou si les tables ont des relations complexes (Imapler, 2014). Cependant, elle offre l'avantage de faciliter la recherche. Toujours d'après cette même discussion, la sauvegarde dans un fichier est plus simple à implémenter et ne nécessite pas d'adaptateur. Par ailleurs, Unity a mis à disposition un outil de sérialisation en Javascript Object Notation (JSON) très simple d'utilisation (Unity Technologies, 2019). Nous estimons que le nombre d'enfants à sauvegarder par cours se limitera à plusieurs dizaines. Par conséquent, nous avons opté pour la sauvegarde des enfants dans un fichier, au format JSON.

5.3.2. Sauvegarde dans un fichier

Sur la base de cette décision, nous avons créé un objet sérialisable, *Child*.cs, contenant toutes les propriétés que l'on souhaite sauvegarder : les parents et les caractéristiques physiques.

Figure 47: Objet utilisé pour la sérialisation des enfants

```
[Serializable]
    public class Child
    {
        public string Parents;
        public string SkinColor;
        public string EyesColor;
        public string HairColor;
        public string Mouth;
        public string Gender;

        public Child(string parents, string skinColor, string eyesColor, string
hairColor, string mouth, string gender)
        {
            Parents = parents;
            SkinColor = skinColor;
            EyesColor = eyesColor;
            HairColor = hairColor;
            Mouth = mouth;
            Gender = gender;
        }
    }
```

Source : données de l'auteur

Ensuite, nous avons créé un script nommé *SaveToJsonFile*.cs afin d'implémenter la méthode nécessaire à cette fonctionnalité présentée dans la figure 48. Elle est appelée lorsque l'on clique sur le bouton « Sauvegarder l'enfant ».

Nous avons procédé par étapes, en n'enregistrant qu'un seul enfant dans un premier temps. Puis nous avons fait évoluer la méthode *WriteFile*() pour ajouter l'enfant créé à une liste existante. Pour ce faire, nous vérifions si le fichier existe déjà. Si ce n'est pas le cas, nous créons le fichier et sauvegardons l'enfant. Dans le cas contraire, nous récupérons la liste existante, et la convertissons en liste, comme recommandé dans une discussion sur *Stackoverflow* (Programmer, 2017). Nous ajoutons ensuite l'enfant à la liste et la sérialisons à l'aide d'un outil nommé *JsonHelper* permettant la conversion d'un tableau en JSON, inspiré de cette discussion également. Il est illustré dans la figure 49.

L'emplacement du fichier de destination est défini par l'instruction *Application.persistentDataPath*. Celle-ci correspond à un dossier qui n'est pas écrasé par les mises à jour de l'application, mais qui est accessible par l'utilisateur s'il souhaite le supprimer (Unity Technologies, 2019).

Figure 48: Sérialisation d'un enfant

```
public void WriteFile()
    {
        Child[] children;
        List<Child> childrenList = new List<Child>();
        Child child = gameManager.GetChild();
        string filePath = Application.persistentDataPath + @"/children.json";
        string childrenToJson;

        if (File.Exists(filePath))
        {
            // Get children from file into a list
            string fileContent = File.ReadAllText(filePath);
            children = JsonHelper.FromJson<Child>(fileContent);
            foreach(Child c in children)
            {
                childrenList.Add(c);
            }

            // Add new child
            childrenList.Add(child);

            // Back to array
            children = childrenList.ToArray();
        }

        else
        {
            // Create child array
            children = new Child[] { child };
        }
```

```
        // Convert to json and write to the file
        childrenToJson = JsonHelper.ToJson(children, true);
        File.WriteAllText(filePath, childrenToJson);
    }
```

Source : données de l'auteur

Figure 49: Classe JsonHelper permettant la sérialisation d'un tableau

```
public static class JsonHelper
    {
        public static T[] FromJson<T>(string json)
        {
            Wrapper<T> wrapper = JsonUtility.FromJson<Wrapper<T>>(json);
            return wrapper.Children;
        }

        public static string ToJson<T>(T[] array)
        {
            Wrapper<T> wrapper = new Wrapper<T>();
            wrapper.Children = array;
            return JsonUtility.ToJson(wrapper);
        }

        public static string ToJson<T>(T[] array, bool prettyPrint)
        {
            Wrapper<T> wrapper = new Wrapper<T>();
            wrapper.Children = array;
            return JsonUtility.ToJson(wrapper, prettyPrint);
        }

        [Serializable]
        private class Wrapper<T>
        {
            public T[] Children;
        }
    }
```

Source : code de l'auteur, inspiré de https://stackoverflow.com/questions/36239705/serialize-and-deserialize-json-and-json-array-in-unity

Afin de communiquer les attributs de l'enfant de *ShowChildResult*.cs vers ce script, nous avons mis à jour *GameManager*.cs et ajouté une variable *Child* ainsi qu'un *getter* et un *setter*. Nous avons également ajouté une méthode *GetParents*() concaténant les identifiants des parents de l'enfant.

Pour terminer, nous avons modifié *ShowChildResult*.cs comme illustré dans la figure 50 afin de stocker les caractéristiques de l'enfant via *GameManager*.cs.

Figure 50: Stockage des caractéristiques de l'enfant en vue de la sauvegarde

```
// Display genes
…

// Get sprite name
…

// Prepare Child object for child saving
childToSave = new Child(gameManager.GetParents(), skinCode, eyesCode, hairCode,
mouthCode, genderCode);
gameManager.SetChild(childToSave);

// Set sprite
```

Source : données de l'auteur

Le résultat obtenu en JSON est présenté dans la figure 51.

Figure 51: Contenu du fichier de sauvegarde

```
{
    "Children": [
        {
            "Parents": "00",
            "SkinColor": "nn",
            "EyesColor": "brbr",
            "HairColor": "bdbr",
            "Mouth": "f2f2",
            "Gender": "xy"
        },
        {
            "Parents": "00",
            "SkinColor": "bb",
            "EyesColor": "blbl",
            "HairColor": "bdbd",
            "Mouth": "f1f1",
            "Gender": "xx"
        },
        {
            "Parents": "14",
            "SkinColor": "bn",
            "EyesColor": "nono",
            "HairColor": "brNO",
            "Mouth": "f2f2",
            "Gender": "xy"
        }
    ]
}
```

Source : données de l'auteur

5.4. Affichage des enfants sur l'écran d'accueil

Les informations nécessaires étant disponibles dans le fichier de sauvegarde, nous avons procédé à l'implémentation de l'affichage des enfants sur la page d'accueil. Ceux-ci doivent correspondre aux parents sélectionnés.

5.4.1. Création d'une liste défilante

En nous documentant sur le sujet, nous avons trouvé un exemple se rapprochant de nos besoins, à la différence que la liste est verticale au lieu d'être horizontale. Nous y avons également trouvé une piste afin de corriger le problème d'affichage de l'application sur d'autres appareils. En effet, en paramétrant dans l'inspecteur d'un canevas le mode de rendu de la caméra en *world space*, celle-ci va automatiquement s'adapter aux dimensions du canevas en question (Siddiqui, 2015). Les démarches entreprises pour corriger ce problème sont détaillées dans le chapitre 5.5.

Tout d'abord, nous avons créé un *Panel*, *ChildrenScrollPanel*, dans *HomeScreenCanvas*, auquel nous avons ajouté un *ScrollRect* horizontal et attribué un masque afin de restreindre l'affichage de son contenu à l'intérieur des bornes définissant la liste défilante. Ensuite, nous avons ajouté un second Panel, *ChildrenContentPanel*, dans le premier et nous lui avons attribué un *HorizontalLayout* afin d'organiser le contenu de la liste. En déposant sa référence dans la propriété Content de son parent, nous lui permettons de défiler le long de l'axe horizontal ainsi défini. Le comportement par défaut des éléments contenus dans le *layout* horizontal est de s'étirer jusqu'à atteindre la même largeur que l'élément parent (Siddiqui, 2015). Afin de le redéfinir, nous avons ajouté un *ContentSizeFitter* dans lequel nous avons affecté la valeur de la propriété *Horizontal fit* à *preferred size*, de manière à ce que la largeur du panneau s'adapte à son contenu.

L'étape suivante de l'exemple consiste à créer un *Prefab*, ce dont nous disposons déjà. Toutefois, celui-ci provient de *l'asset* que nous avons acheté et est plus complexe que dans l'exemple. En effet, il est composé d'un *SpriteRenderer*, et afin de pouvoir modifier l'apparence du *Chibi*, il doit également avoir un objet *ChibiCreator* associé. Or, la méthode permettant d'instancier correctement cet objet est privée et inaccessible depuis notre projet. Les *Chibis* ne peuvent être créés qu'à partir de l'éditeur. Par conséquent, nous avons dupliqué un *Chibi* existant jusqu'à obtenir 10 enfants et les avons placés dans *ChildrenContentPanel*. Cela limite le nombre d'enfants à 10 par parents, ce qui correspond à un total de 360 enfants (6 pères x 6 mères x 10 enfants). Cela a été jugé suffisant pour atteindre les objectifs de ce projet, en accord avec l'étudiant en Master.

5.4.2. Affichage des enfants

Afin de permettre l'affichage, nous avons créé le script *OnHomeScreenLoad*.cs dont la méthode *SetContent*() récupère les enfants enregistrés dans le fichier précédemment créé, s'il existe, et les affiche lorsque leurs parents respectifs sont sélectionnés. Lors du démarrage de l'application, aucun

enfant n'apparaît en raison des deux *Chibis* neutres et la liste est masquée. Son contenu est mis à jour à chaque fois que l'on sélectionne un parent ou que l'on charge l'écran d'accueil.

Figure 52: Extrait du script permettant l'affichage des enfants sur l'écran d'accueil

```
public void SetContent()
    {
        #region variables
        string filePath = Application.persistentDataPath + @"/children.json";
        string fileContent;
        Child[] children;
        string parentCode;
        int index = 0;
        SpriteSelector fatherSelector = father.GetComponent<SpriteSelector>();
        SpriteSelector motherSelector = mother.GetComponent<SpriteSelector>();
        List<GameObject> childObjectList = new List<GameObject>
        {
            child1, child2, child3, child4, child5, child6, child7, child8, child9,
child10
        };
        #endregion

        for (int i = 0; i < childObjectList.Count; i++)
        {
            childObjectList[i].SetActive(false);
        }

        if (!fatherSelector.hairMidRenderer.color.Equals(white) &&
!motherSelector.hairMidRenderer.color.Equals(white))
        {
            childrenPanel.SetActive(true);
            parentCode = gameManager.GetParents();

            if (File.Exists(filePath))
            {
                // Get children from file into an array
                fileContent = File.ReadAllText(filePath);
                children = JsonHelper.FromJson<Child>(fileContent);

                // Display child as corresponding Chibi
                for (int i = 0; i < children.Length; i++)
                {
                    if (children[i].Parents.Equals(parentCode))
                    {
                        childObjectList[index].SetActive(true);
                        DisplayChild(children[i], childObjectList[index]);
                        index++;
                    }
                }
            }
        }
    }

    private void DisplayChild(Child child, GameObject childObject)
    {
        SpriteManagement spriteManagement = new SpriteManagement();
        GeneCodeManagement geneCodeManagement = new GeneCodeManagement();
```

```
        SpriteSelector childSelector = childObject.GetComponent<SpriteSelector>();
        bool isMale = geneCodeManagement.IsMale(child.Gender);
        Chibi chibi = geneCodeManagement.GetChibiFromChild(child, isMale);

        spriteManagement.SetSprite(childSelector, isMale, chibi);
    }
```

Source : données de l'auteur

Afin de centraliser la méthode permettant d'attribuer les *Sprites*, *SetSprites*(), nous avons créé une nouvelle classe nommée *SpriteManagement*.cs, présentée dans la figure 53, et modifié le code source en conséquence.

Figure 53: Classe SpriteManagement

```
public void SetSprite(SpriteSelector selector, bool isMale, Chibi chibi)
        {
            ChibiCreator creator;
            creator = selector.chibiCreator;
            if (isMale)
            {
                selector.hairMidRenderer.sprite =
creator.HairMidSprites[Constants.MALE_MIDHAIR_INDEX];
                selector.hairBackRenderer.sprite =
creator.HairBackSprites[Constants.MALE_BACKHAIR_INDEX];
            }
            else
            {
                selector.hairMidRenderer.sprite =
creator.HairMidSprites[Constants.FEMALE_MIDHAIR_INDEX];
                selector.hairBackRenderer.sprite =
creator.HairBackSprites[Constants.FEMALE_BACKHAIR_INDEX];
            }

            selector.headRenderer.sprite = creator.HeadSprites[chibi.SkinColor];
            selector.faceEyesRenderer.sprite =
creator.FaceEyesSprites[chibi.EyesColor];
            selector.faceEyebrowsRenderer.sprite =
creator.FaceEyebrowsSprites[Constants.EYEBROWS_INDEX];
            selector.faceEyebrowsRenderer.color = chibi.HairColor;
            selector.hairMidRenderer.color = chibi.HairColor;
            selector.hairBackRenderer.color = chibi.HairColor;
            selector.faceRestRenderer.sprite = creator.FaceRestSprites[chibi.Mouth];
        }
```

Source : données de l'auteur

Nous avons également mis à jour *ApplicationNavigation*.cs afin d'y référencer le script *OnHomeScreenLoad*.cs pour appliquer la méthode *SetContent*() lors de l'affichage de l'écran d'accueil. Le résultat obtenu est présenté dans la figure 54.

Figure 54: Affichage de la liste des enfants sur l'écran d'accueil

Source : données de l'auteur

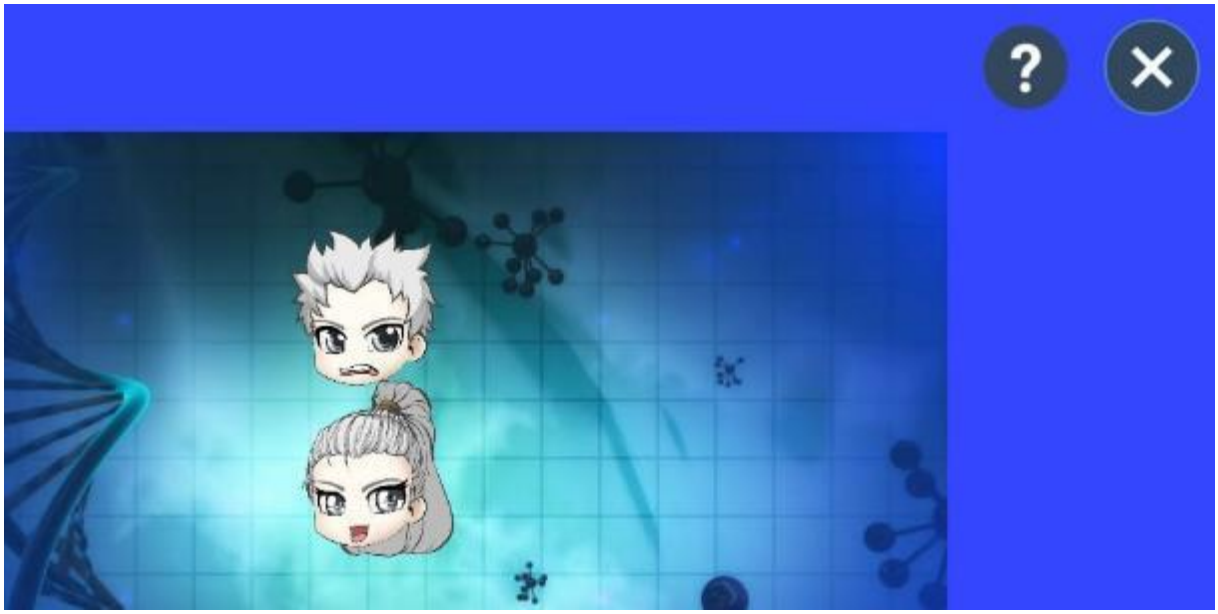
5.5. Test de l'application sur un autre appareil

Afin d'investiguer le problème d'affichage sur la seconde tablette mise à disposition pour le test de l'application, nous l'avons installée et exécutée sur d'autres appareils Android : un Sony Xperia XZ (F8331) fonctionnant sous Android 8.0.0, ainsi qu'un Motorola E5 Play (XT1920-16) fonctionnant, quant à lui, sous Android 8.1.0. L'affichage de l'application sur chacun de ces smartphones est illustré dans les figures 55 et 56.

Figure 55: Essai du prototype sur le Sony Xperia

Source : données de l'auteur

Figure 56: Essai du prototype sur le Motorola E5



Source : données de l'auteur

Il s'avère que notre application a un comportement similaire sur le Motorola et la tablette Samsung mise à disposition pour la phase de test. L'application est décentrée, et elle ne s'adapte pas à l'écran. Afin d'avoir une vue d'ensemble de la situation, nous avons regroupé les résolutions des appareils utilisés dans le tableau 13, ce qui nous a permis de confirmer que le problème y est bien lié.

Tableau 13: Résolution d'écran des appareils utilisés (au format paysage)

Modèle	Résolution
Lenovo Yoga Book (YB1-X90F, appareil principal)	1920x1080
Sony Xperia XZ (F8331)	1920x1080
Samsung Note 10 (SM-P605)	2560x1600
Motorola E5 Play (XT1920-16)	1280x720

Source : données de l'auteur

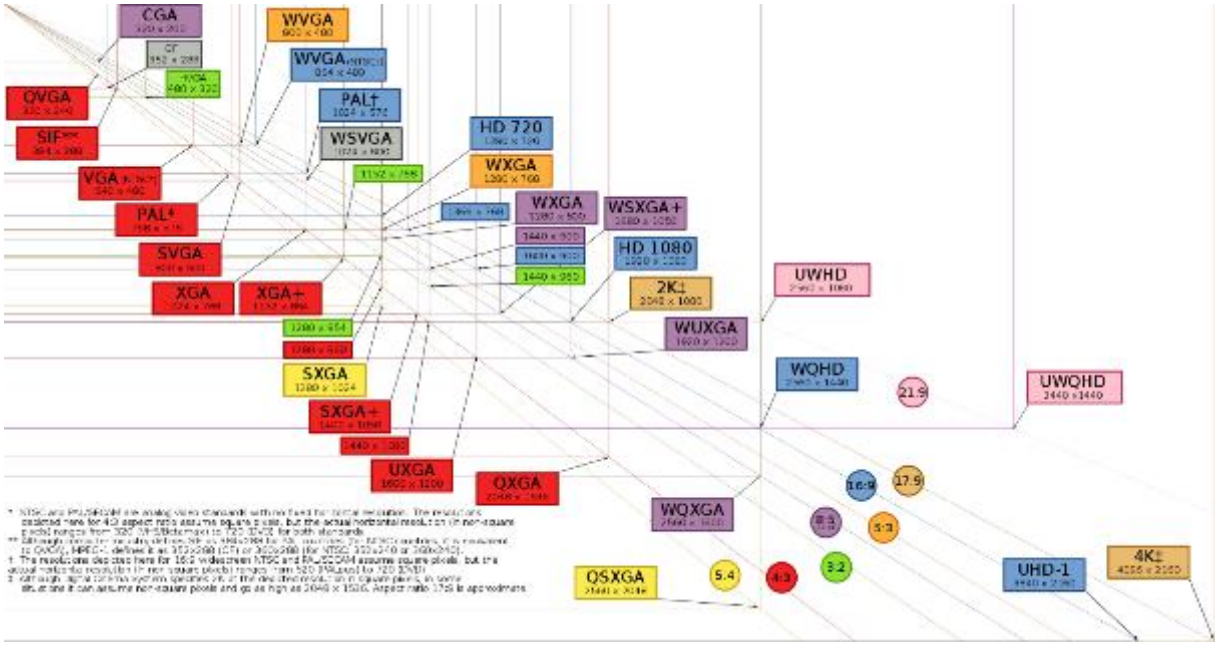
Afin de centrer l'image de l'application, nous avons placé la caméra, *MainCamera*, en tant qu'enfant du canevas racine, *RootCanvas*. Ainsi, il est possible de lui faire hériter des paramètres de positionnement de son parent. Ceci a permis de centrer l'interface utilisateur de l'application, mais le problème d'adaptation à la résolution demeure.

Nous avons commencé par nous renseigner sur les différentes résolutions utilisées pour les tablettes tactiles. D'après Continisio, les résolutions les plus couramment utilisées pour un ordinateur sont 1920×1080 pixels (1080p) et 1280x720 (720p) (Continisio, 19). Il ajoute que la gamme de résolutions est très large pour les smartphones, davantage que pour les ordinateurs, mais ne précise pas ce qu'il en est pour les tablettes.

Nous avons trouvé une vue d'ensemble des définitions d'écran les plus courantes sur Wikipedia, que nous avons reprise dans la figure 57. Elle montre en effet que la gamme disponible est vaste, et qu'il serait plus judicieux d'utiliser une résolution de référence plus élevée, comme par exemple celle du Samsung Note (2560x1600). Ceci est confirmé par le comparatif des tablettes graphiques réalisé par le site lesnumeriques.com, qui indique comme plus haute résolution celle du Samsung (Jacquet, 2019).

Par conséquent, nous avons défini 2560x1600 comme résolution de référence et appliqué la modification citée dans le chapitre 5.4.1 et décrite par M. Siddiqui (Siddiqui, 2015) qui consiste à paramétrer le mode de rendu de la caméra en *world space*. Nous avons ensuite procédé à plusieurs créations d'enfants en simulant les trois différentes résolutions citées dans le tableau 13 et confirmé que l'application s'adapte désormais à ces définitions d'écran. Nous avons également révisé et amélioré l'interface en utilisant des *layouts* verticaux et horizontaux pour optimiser la disposition des éléments graphiques et nous assurer de leur comportement, quelle que soit la résolution de l'écran.

Figure 57: Définitions d'écran les plus courantes



Source : https://fr.wikipedia.org/wiki/D%C3%A9finition_d%27%C3%A9cran

Toutefois, cette mesure a entraîné un effet inattendu sur l'affichage du déplacement du code lors du glisser-déposer. En effet, celui-ci ne suit pas toujours la position de la souris ou du doigt, en fonction de la résolution de l'écran de l'appareil utilisé. Après quelques recherches, nous avons compris que la position de la souris, *Input.mousePosition*, ne correspond pas à la position dans l'espace, *world position*, en raison de ce nouveau paramétrage. En effet, dans une discussion sur le site de la communauté d'Unity, un développeur nommé Robertbu fait mention de la nécessité de convertir des coordonnées en deux dimensions vers un univers en trois dimensions avec ce paramétrage de caméra (Charlesvi, 2013). Le sujet dépassant nos connaissances, nous avons décidé de nous documenter davantage en parallèle au développement. Ce problème n'empêchant pas d'utiliser l'application, nous ne l'avons pas considéré comme prioritaire.

5.6. Affichage des détails d'un enfant sauvegardé

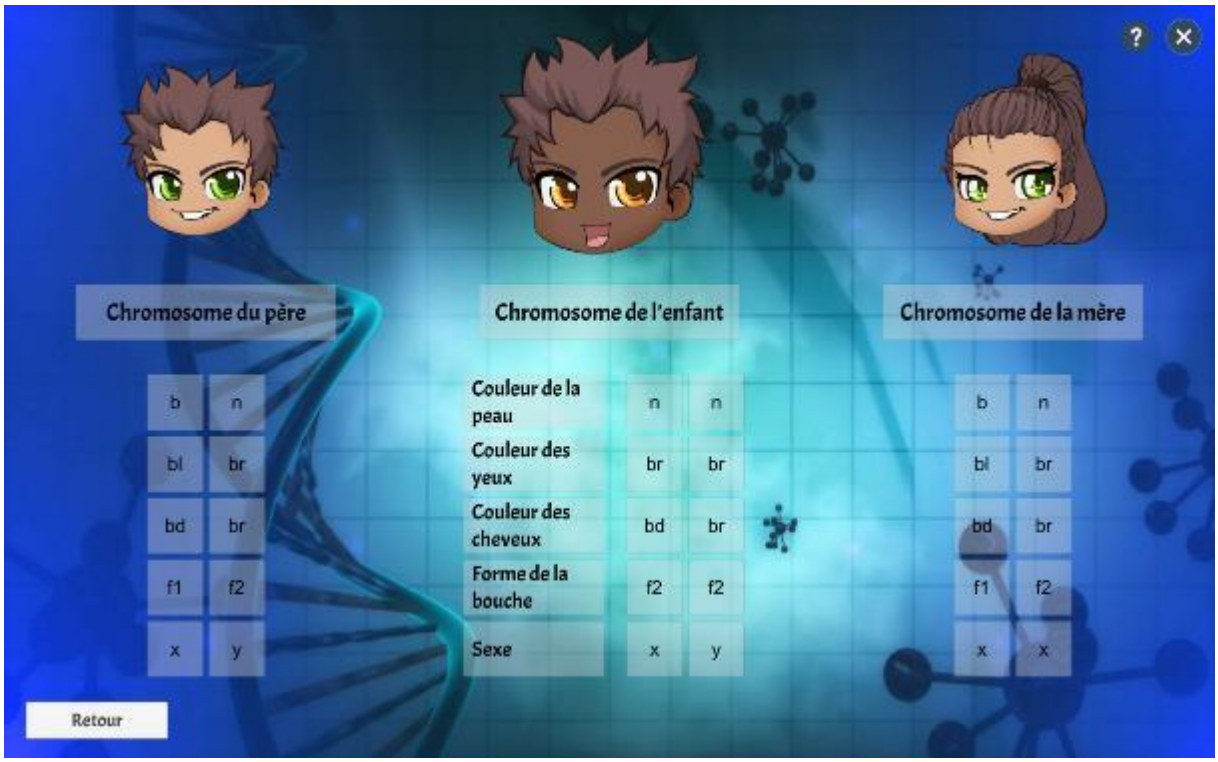
L'étape suivante du développement consiste à afficher les détails d'un enfant en cliquant sur sa représentation dans l'écran d'accueil. Dans un premier temps, nous avons modifié l'écran de visualisation de l'enfant existant, afin de montrer le code génétique des parents et de permettre la comparaison avec celui de l'enfant. Nous avons ensuite adapté le code et la navigation pour afficher un bouton de retour lorsque l'on vient de l'écran d'accueil et le bouton sauvegarder lorsque l'on vient de l'écran de création.

5.6.1. Adaptation de l'écran de visualisation de l'enfant

À la demande de l'étudiant en Master, nous avons remanié l'écran de visualisation de l'enfant créé au chapitre 3.6 afin d'afficher le code génétique des parents. Ainsi, il est possible de comparer à la fois l'apparence et le profil génétique. Pour ce faire, nous avons divisé l'écran en trois zones verticales, elles-mêmes divisées en trois zones horizontales correspondant à la représentation visuelle du parent ou de l'enfant, puis une étiquette indiquant de quel chromosome il s'agit et enfin du code génétique.

Nous avons ajouté un bouton permettant de retourner sur l'écran d'accueil, qui sera masqué lors du processus de création d'un enfant. Le bouton « sauvegarder » sera, quant à lui, indisponible lorsque l'on affichera les détails d'un enfant depuis l'écran d'accueil. Le résultat est représenté dans la figure 58.

Figure 58: Écran de visualisation des détails de l'enfant



Source : données de l'auteur, police récupérée sur https://fonts.google.com

5.6.2. Adaptation du code

Pour commencer, nous avons ajouté une variable dans *GameManager*.cs permettant de stocker la liste des enfants affichés sur l'écran d'accueil, afin de récupérer les caractéristiques à envoyer vers l'écran de visualisation. Celle-ci est affectée lors du chargement de l'écran d'accueil, dans le script *OnHomeScreenLoad*.cs.

Nous avons ensuite créé un script nommé *ShowDetails*.cs qui sera attribué à chaque *Chibi* de la liste des enfants. Ce script comporte la méthode *OnMouseDown*() qui affiche l'écran créé au chapitre 5.6.1 à l'aide de plusieurs méthodes. La première, *GetCharacters*(), récupère les objets *ChibiGenes* correspondant aux parents et à l'enfant depuis *GameManager*.cs. La seconde, *SetSprites*(), applique les *Sprites*, autrement dit, les caractéristiques physiques, correspondant aux codes génétiques à l'aide de la classe *SpriteManagement*.cs.

Les deux suivantes proviennent d'une nouvelle classe, *TransferCode*.cs, créée pour centraliser des méthodes permettant l'affichage des gènes dans un tableau. Étant donné que le matériel génétique de l'enfant correspond à un objet *Child*, et celui des parents à un objet *ChibiGenes*, une méthode a été implémentée pour chaque type d'objet. Celle pour l'objet *Child* est présentée à la figure 59 Pour finir, la dernière méthode fait appel à *ApplicationNavigation.ShowDetailScreen*() et affiche l'écran en question.

Figure 59: Méthode permettant l'affichage des gènes de l'enfant

```
public void ChildTransferCode(int childIndex)
    {
        Transform child = destChild.transform;
        Child childGenes = gameManager.GetChildFromList(childIndex);
        childCodes = new List<string>();

        SplitGenePair(childGenes.SkinColor, childCodes);
        SplitGenePair(childGenes.EyesColor, childCodes);
        SplitGenePair(childGenes.HairColor, childCodes);
        SplitGenePair(childGenes.Mouth, childCodes);
        SplitGenePair(childGenes.Gender, childCodes);

        for (int i = 0; i < child.childCount; i++)
        {
            Text text = child.GetChild(i).GetComponentInChildren<Text>();
            text.text = childCodes[i];
        }
    }
```

Source : données de l'auteur

Il est à noter que pour cet écran, nous avons utilisé *Transform.GetChild*() afin de cibler les lignes des tableaux dans l'interface graphique. Cette méthode est plus appropriée que l'utilisation des variables publiques correspondant aux gènes, puisque nous n'affectons plus qu'un objet par personnage, au lieu de dix. Cela diminue le risque d'erreur lors du glisser-déposer de l'objet dans l'inspecteur. Par conséquent, le procédé auquel nous avons eu recours pour l'écran de visualisation de l'enfant après création nécessite d'être adapté par souci de cohérence et de facilitation de la maintenance du code.

5.6.3. Gestion de la navigation

Afin de masquer les boutons de retour et de sauvegarde en fonction du scénario, nous avons mis à jour le script *ApplicationNavigation*.cs en y ajoutant les variables publiques correspondantes, *SaveButton* et *BackButton*. Ces boutons sont activés, respectivement désactivés, dans les méthodes *ShowResultScreen*() et *ShowDetailScreen*(). Cette dernière est présentée dans la figure 60. Elle participe également à la gestion de l'historique pour l'affichage de l'écran d'aide.

Figure 60: Méthode permettant l'affichage de l'écran de visualisation des détails d'un enfant

```
public void ShowDetailScreen()
    {
        gameManager.SetScreenIndex(3);
        homeScreen.SetActive(false);
        creationScreen.SetActive(false);
        resultScreen.SetActive(true);
        helpScreen.SetActive(false);
        backButton.SetActive(true);
        saveButton.SetActive(false);
    }
```

Source : données de l'auteur

La méthode *LeaveHelpScreen*() a également été mise à jour en ajoutant la condition correspondant à l'écran de visualisation des détails, comme montré dans la figure 61.

Figure 61: Méthode LeaveHelpScreen() après ajout de l'écran de visualisation des détails

```
public void LeaveHelpScreen()
    {
        int index = gameManager.GetScreenIndex();
        switch (index)
        {
            case 0:
                ShowHomeScreen();
                break;
            case 1:
                ShowCreationScreen();
                break;
            case 2:
                ShowResultScreen();
                break;
            case 3:
                ShowDetailScreen();
                break;
        }
        helpScreen.SetActive(false);
    }
```

Source : données de l'auteur

5.6.4. Amélioration de l'écran d'accueil

L'ajout de cette fonctionnalité, avec changement d'écran lors d'un clic sur l'un des enfants de la liste, a amené un défaut de comportement dans l'écran d'accueil. En effet, désormais, lorsque l'on touche l'un des éléments de la liste en voulant la faire défiler, on déclenche inévitablement l'affichage de l'écran de visualisation des détails, même si cela n'est pas désiré.

Une solution possible aurait été l'ajout d'une barre de défilement au-dessus ou au-dessous de la liste. Cependant, avec l'augmentation du nombre d'enfants, nous avons constaté que ce procédé n'était pas optimal. En effet, au-delà d'un certain nombre d'enfants, les éléments situés aux extrémités de la liste sortent de l'écran. Lors de l'implémentation de la fonctionnalité précédente, nous avons remarqué que la présence d'un enfant hors de l'écran n'était pas évidente et que nous avions considéré le second enfant comme étant le premier, par erreur. En outre, en n'ayant qu'une partie des enfants affichés sur l'écran, nous allons à l'encontre de l'objectif de l'application, qui consiste à illustrer la diversité de l'apparence des enfants. Il est donc important d'avoir un aperçu de l'ensemble de la descendance en un seul coup d'œil. Par conséquent, nous avons décidé de supprimer la liste déroulante et de présenter les enfants sous forme de grille. Pour une meilleure expérience utilisateur, nous avons également ajouté un texte annonçant le nombre d'enfants existants par rapport au nombre maximal et masqué le bouton permettant d'accéder à l'écran suivant lorsque le nombre maximal d'enfants est atteint. Le résultat est illustré dans la figure 62.

Figure 62: Écran d'accueil amélioré

Source : données de l'auteur

5.7. Amélioration de l'aide

Afin d'améliorer l'expérience utilisateur, nous avons amélioré l'aide du jeu en faisant correspondre un écran d'aide à chaque écran composant l'application. Puis nous avons adapté la navigation pour gérer l'affichage de ses écrans et le retour à l'écran précédent.

5.7.1. Création des aides correspondant à chaque écran du jeu

Pour commencer, nous avons créé trois écrans dont le contenu diffère selon l'écran depuis lequel on accède à l'aide du jeu. Pour ce faire, nous avons modifié le canevas de l'écran d'aide existant et avons remplacé le contenu par du texte et des images. Une fois la mise en forme jugée satisfaisante, nous avons dupliqué cet écran pour obtenir les écrans d'aide correspondant à l'écran de création de l'enfant et celui de visualisation du résultat. Étant donné que l'écran de visualisation des détails correspond à ce dernier, nous lui avons affecté le même écran d'aide. Ces écrans sont présentés dans les figures 63 à 65.

Figure 63: Aide correspondant à l'écran d'accueil

Source : données de l'auteur

Figure 64: Écran d'aide correspondant à la visualisation de l'enfant

Source : données de l'auteur

Figure 65: Écran d'aide correspondant à l'écran de création de l'enfant

Source : données de l'auteur

5.7.2. Gestion de la navigation

Dans le but d'afficher le bon écran d'aide en fonction de l'écran affiché, nous avons remplacé la variable *helpScreen* du script *ApplicationNavigation*.cs par trois variables correspondant à chacun des écrans d'aide. Nous avons ensuite remplacé chacune des références à la variable *helpScreen* par l'une des nouvelles variables, comme illustré dans la figure 66.

Figure 66: Remplacement de la variable helpScreen

```
81  107         public void LeaveHelpScreen()
    @@ -96,7 +122,10 @@ public void LeaveHelpScreen()
96  122                     ShowDetailScreen();
97  123                     break;
98  124             }
99      -           helpScreen.SetActive(false);
        125 +
        126 +           helpScreenHome.SetActive(false);
        127 +           helpScreenCreation.SetActive(false);
        128 +           helpScreenResult.SetActive(false);
100 129         }
```

Source : copie d'écran de l'auteur provenant de github.com

Pour terminer, nous avons modifié la méthode *ShowHelpScreen*() comme présenté dans la figure 67. Nous avons ajouté une variable permettant de récupérer l'indice de l'écran en cours d'affichage et l'utiliser pour afficher l'écran d'aide correspondant.

Figure 67: Méthode affichant l'écran d'aide correspondant à l'écran actuel

```
public void ShowHelpScreen()
    {
        homeScreen.SetActive(false);
        creationScreen.SetActive(false);
        resultScreen.SetActive(false);
        int index = gameManager.GetScreenIndex();

        switch (index)
        {
            case 0:
                helpScreenHome.SetActive(true);
                break;
            case 1:
                helpScreenCreation.SetActive(true);
                break;
            case 2:
                helpScreenResult.SetActive(true);
                break;
```

```
            case 3:
                helpScreenResult.SetActive(true);
                break;
        }

    }
```

Source : données de l'auteur

5.8. Révision du code

Afin de vérifier la qualité du code et de faciliter sa maintenance, nous avons passé en revue l'ensemble des scripts. Nous avons remanié certaines méthodes pour enlever les doublons, diminuer le nombre de variables à affecter depuis l'inspecteur d'Unity et améliorer la lisibilité du code. Nous avons également supprimé les références inutiles et contrôlé les commentaires.

Le premier script que nous avons remanié est *ShowChildResult*.cs. Au chapitre 5.6.2, nous avons expliqué que l'utilisation de la méthode *transform.GetChild*(id) était mieux adaptée à la maintenance de l'application parce qu'elle ne nécessitait qu'une seule variable publique à affecter dans l'inspecteur. En effet, avant modification de ce script, les dix gènes source étaient référencés par dix variables publiques. Nous avons décidé de ne référencer que leur parent, *CreationChildGenes*, et d'obtenir ses enfants grâce à cette méthode. Nous avons également centralisé la déclaration des variables privées et décomposé la méthode principale en méthodes privées. Ceci a permis de simplifier le code et de le rendre plus lisible, comme illustré à la figure 68. Nous avons également remplacé le *switch/case* par un test avec condition pour construire la liste de paires de gènes.

Figure 68: Méthode principale de ShowChildResult après révision

```
public void SetContent()
    {
        #region Variables
        int geneNumber = sourceChildGenes.transform.childCount;
        SpriteSelector childSelector = child.GetComponent<SpriteSelector>();
        List<GenePair> genePairList = new List<GenePair>();
        Chibi childToDisplay;
        Child childToSave;
        string firstGene = null;
        #endregion

        // Display genes
        for (int i = 0; i < geneNumber; i++)
        {
            string currentGene =
sourceChildGenes.transform.GetChild(i).GetComponentInChildren<Text>().text;
            destChildGenes.transform.GetChild(i).GetComponentInChildren<Text>().text
= currentGene;

            if((i % 2) == 0)
            {
```

```
                firstGene = currentGene;
            }
            else
            {
                genePairList.Add(new GenePair(firstGene, currentGene));
            }
        }

        // Get sprites name
        GetSpriteNames(genePairList);

        // Prepare Child object for child saving
        childToSave = new Child(gameManager.GetParents(), skinCode, eyesCode,
hairCode, mouthCode, genderCode);
        gameManager.SetChild(childToSave);

        // Set sprite
        childToDisplay = new Chibi(skinIndex, eyesIndex, -1, -1, -1, hairColor,
mouthIndex );
        manageSprite.SetSpriteFromChibiObject(childSelector, isMale,
childToDisplay);
    }
```

Source : données de l'auteur

Le second script que nous avons remanié est *ChildrenCreationBehaviour*.cs, et plus précisément, la méthode *SetGeneCode*(…). Elle consistait à afficher le code génétique des parents en utilisant la méthode *GameObject.Find*() et le chemin de l'objet en question. Cette façon de procéder utilisait des chemins relatifs « en dur » et impliquait que le moindre changement dans l'arborescence la mette en échec. Il s'agit d'un bon exemple des inconvénients de la dette technique introduite dans ce projet et expliquée au chapitre 6. Nous avons remplacé ce procédé par l'utilisation d'une liste de *GenePair* et d'une boucle *for*, en faisant correspondre les indices des *GameObjects* destinés à accueillir les gènes et la liste de paires de gènes comme présenté dans le tableau 14. Les figures 69 et 70, quant à elles, illustrent les modifications apportées au code.

Tableau 14: Correspondance des indices

Caractéristiques	Indice de la paire de gènes correspondante	Indices des petits-enfants de FatherGenes/MotherGenes (gène 1, gène2)
Couleur de la peau	0	0, 1
Couleur des yeux	1	2, 3
Couleur des cheveux	2	4, 5
Forme de la bouche	3	6,7
Sexe	4	8, 9

Source : données de l'auteur

Les formules de conversion sont :

- Indice du gène 1 = 2 * indice de la paire de gènes
- Indice du gène 2 = 2 * indice de la paire de gènes + 1

Figure 69: Affectation du texte d'une paire de gènes avant révision

```
GameObject.Find(path + "MouthGene1/MouthGene1").GetComponent<Text>().text =
mouthPair.gene1;
GameObject.Find(path + "MouthGene2/MouthGene2").GetComponent<Text>().text =
mouthPair.gene2;
```

Source : données de l'auteur

Figure 70: Méthode SetGeneCode() du script ChildrenCreationBehaviour après révision

```
private void SetGeneCode(bool isFather, SpriteSelector selector)
    {
        GeneCodeManagement manageCode = new GeneCodeManagement();
        GameObject parentGenes;
        List<GenePair> genePairList = new List<GenePair>();
        GenePair skinPair = new GenePair();
        GenePair eyePair = new GenePair();
        GenePair hairPair = new GenePair();
        GenePair mouthPair = new GenePair();
        GenePair genderPair = new GenePair();
        ChibiGenes genes;

        if (isFather)
        {
            parentGenes = fatherGenes;
            genes = Parents.FATHER_GENES[gameManager.GetFather()];
        }
        else
        {
            parentGenes = motherGenes;
            genes = Parents.MOTHER_GENES[gameManager.GetMother()];
        }

        // Prepare gene codes
        skinPair = skinPair.SplitGenePair(genes.SkinColor);
        genePairList.Add(skinPair);

        eyePair = eyePair.SplitGenePair(genes.EyesColor);
        genePairList.Add(eyePair);

        hairPair = hairPair.SplitGenePair(genes.HairColor);
        genePairList.Add(hairPair);

        mouthPair = mouthPair.SplitGenePair(genes.Mouth);
        genePairList.Add(mouthPair);

        genderPair = genderPair.SplitGenePair(genes.Gender);
```

```
        genePairList.Add(genderPair);

        // Display genes
        for(int i = 0; i < genePairList.Count; i++)
        {
            parentGenes.transform.GetChild(i *
2).transform.GetChild(0).GetComponent<Text>().text = genePairList[i].gene1;
            parentGenes.transform.GetChild(i * 2 +
1).transform.GetChild(0).GetComponent<Text>().text = genePairList[i].gene2;
        }
    }
```

Source : données de l'auteur

Pour terminer, nous avons retiré le suivi du déplacement du code lors du glisser-déposer dans le script *DragHandler*.cs. Le temps nous a manqué pour corriger proprement ce problème, mais son impact sur l'expérience utilisateur a été jugé trop important pour le négliger.

6. Déroulement du projet

6.1. Planification et cahier des charges

La planification et le cahier des charges ont été réalisés durant les quatre premières semaines du travail. Le tableau 15 présente la planification provisoire réalisée avec MS *Project* à partir du cahier des charges rédigé par l'étudiant en Master et de la description du Travail de Bachelor. Comme expliqué au chapitre 3.1, la phase de test du prototype devait être réalisée avant la fin de l'année scolaire pour les cycles d'orientation, raison pour laquelle celle-ci a été prévue au début du mois de juin. La fin du projet a été planifiée pour le début du mois de juillet afin d'avoir un mois de marge, par précaution.

Tableau 15: Extrait de la planification provisoire réalisée sur MS Project

Nom de la tâche	Début	Fin
Travail de Bachelor	Sam 09.03.19	Dim 07.07.19
État de l'art	Sam 09.03.19	Dim 31.03.19
Modélisation du prototype	Sam 06.04.19	Sam 06.04.19
Développement du prototype	Dim 07.04.19	Dim 26.05.19
Test du prototype	Sam 01.06.19	Dim 16.06.19
Amélioration du prototype	Sam 22.06.19	Dim 07.07.19

Source : données de l'auteur

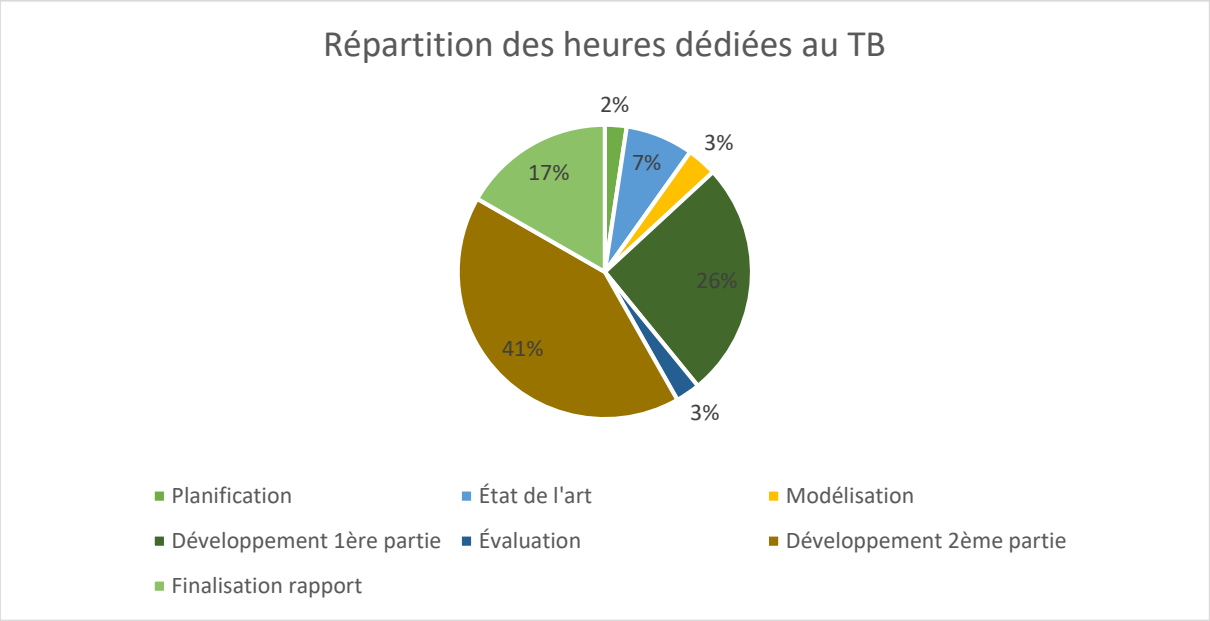
Le tableau 16 présente le déroulement réel du projet et la figure 71 la répartition des heures. La phase de l'état de l'art a commencé environ une semaine plus tard que prévu, en raison du nombre de personnes à réunir pour la première séance (personnes concernées par le travail de Bachelor et le travail de Master). Celle-ci a d'ailleurs dû avoir lieu en l'absence de l'étudiant en Master. On peut constater que la période à laquelle l'évaluation allait se produire a été correctement prédite malgré le peu d'informations à disposition au début du projet. La période des examens de fin de semestre a nécessité la mise en pause du projet, raison pour laquelle la suite du développement s'achève environ deux semaines après la date initialement prévue.

Tableau 16: Déroulement effectif du projet

Nom de la tâche	Nombre d'heures	Début	Fin
Travail de Bachelor	335	21.02.2019	04.08.2019
Planification	8	21.02.2019	13.03.2019
État de l'art	25	18.03.2019	08.04.2019
Modélisation	11	09.04.2019	14.04.2019
Développement 1ère partie	87	20.04.2019	04.06.2019
Évaluation	9	05.06.2019	11.06.2019
Développement 2ème partie	139	24.06.2019	24.07.2019
Finalisation rapport	56	25.07.2019	04.08.2019

Source : données de l'auteur

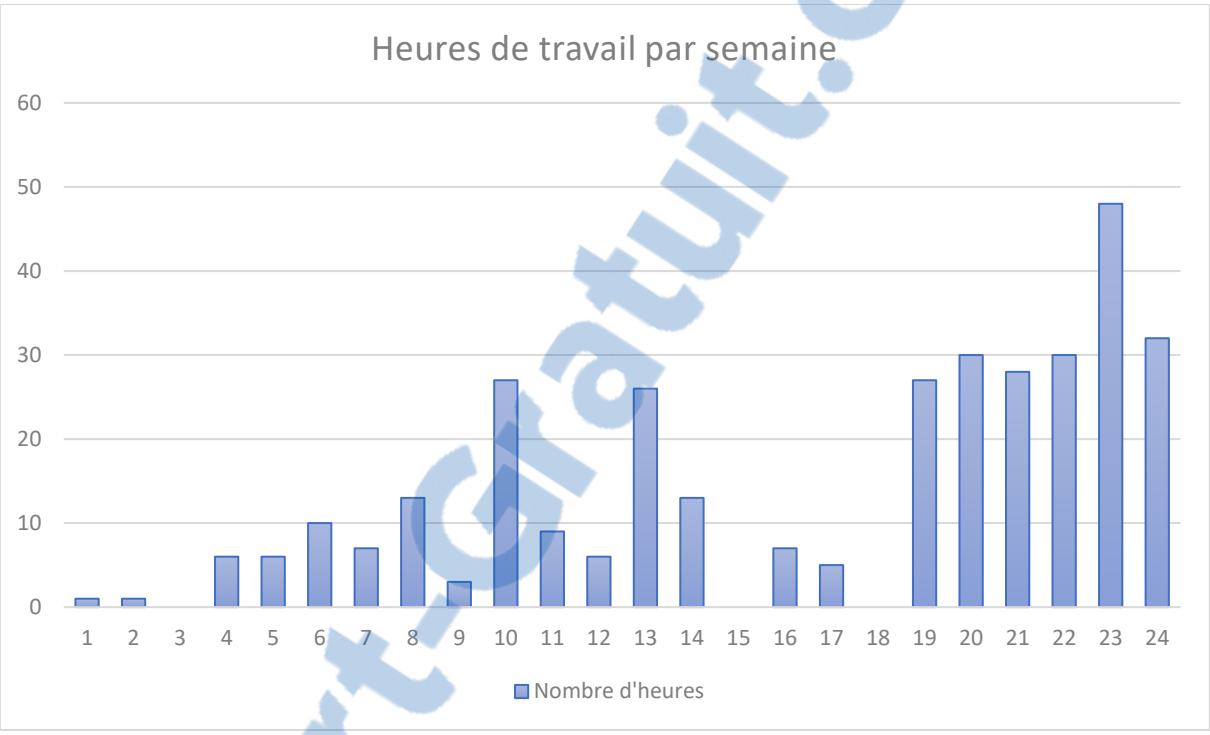
Figure 71: Répartition des heures dédiées au projet



Source : données de l'auteur

La figure 72 présente le nombre d'heures de travail hebdomadaires effectuées pendant l'ensemble du projet. Les valeurs ayant permis la construction de ce graphique se trouvent dans le ficher Excel transmis dans la clé USB accompagnant ce rapport.

Figure 72: Travail hebdomadaire



Source : données de l'auteur

La quinzième semaine coïncide avec la période de rendu des projets du dernier semestre de notre formation, tandis que la dix-huitième semaine correspond aux examens. Notre charge de travail a nécessité de mettre ce projet en attente pendant ces périodes. À partir de la dix-neuvième semaine, les cours se sont terminés et nous avons pu consacrer davantage de temps à notre application.

Autant que possible, la rédaction du rapport a été effectuée au fur et à mesure de l'avancement du projet. Cependant il a été nécessaire de créer une dette technique durant la première partie du développement afin de respecter les délais. Il s'agit de reporter la rédaction de la documentation et de diminuer la qualité du développement au profit d'un avancement plus rapide (Sven Johann, 2013). Cette notion nous avait été présentée par M. David Wannier dans le module 625-1, Organisation du développement logiciel. Elle a permis de respecter les délais, mais a eu pour conséquence plusieurs phases de correction afin de fournir un code de bonne qualité et facile à maintenir.

6.2. Séances avec l'enseignant responsable du projet et avec l'étudiant

Durant ce projet, nous avons, dans la mesure du possible, coordonné les rencontres avec l'enseignant responsable et celles avec l'étudiant en Master afin qu'elles aient lieu la même semaine. Elles se sont déroulées environ tous les quinze jours.

Tableau 17: Séances réalisées durant le projet

Date	Participant
Jeudi 21.02	Enseignants responsables du travail de Bachelor et de Master
Mercredi 27.02	Étudiant en Master
Vendredi 08.03	Enseignant responsable
Jeudi 21.03	Enseignant responsable, assistant
Mercredi 27.03	Enseignants responsables, étudiant en Master
Jeudi 04.04	Assistant
Mercredi 10.04	Étudiant en Master
Jeudi 18.04	Enseignant responsable, assistant
Jeudi 02.05	Enseignant responsable, assistant
Mercredi 08.05	Étudiant en Master
Mercredi 22.05	Étudiant en Master
Jeudi 23.05	Enseignant responsable, assistant
Mercredi 05.06	Étudiant en Master
Jeudi 06.06	Assistant
Jeudi 13.06	Enseignant responsable, assistant
Lundi 08.07	Enseignant responsable
Mardi 09.07	Étudiant en Master
Mercredi 24.07	Enseignant responsable, assistant, étudiant en Master

Source : données de l'auteur

6.3. Gestion du projet

Ce travail a été mené en suivant une adaptation de *Scrum* où l'étudiant en Master jouait le rôle de p*roduct owner*. Ainsi, les rencontres avec ce dernier ayant eu lieu pendant la phase de développement ont fait office de *sprint plannings* et de *sprint reviews*. Nous n'avons, en revanche, pas effectué de *daily scrum* étant donné que le développement a été réalisé par une seule personne.

6.3.1. Outils utilisés

La gestion du développement a été réalisée à l'aide de taiga.io, une plateforme de gestion de projet pour des développeurs et responsables de projet utilisant la méthodologie Agile (Taiga Agile LLC, 2019). Elle a servi à administrer le *product backlog* et à suivre le déroulement des *Sprints*.

L'application a été développée en C# sur Unity 2018.3.5f1, puis 2019.1.8f1 pour bénéficier d'une fonctionnalité récente, en combinaison avec Microsoft Visual Studio Enterprise 2017 version 15.9.7. C'est une configuration que nous avions déjà utilisée lors d'un projet de groupe durant notre formation.

Le versionnage du projet a été effectué avec Github. Par souci de protection de la propriété intellectuelle du contenu payant, le repository que nous avons utilisé est privé.

Le suivi des heures consacrées à ce projet a été consigné dans un fichier Excel qui se trouve dans la clé USB jointe à ce mémoire.

6.3.2. Product backlog

Le *product backlog* a été défini en collaboration avec l'étudiant en Master en début de projet. Les *user stories* qui le composent sont présentées dans les tableaux 18 et19, par ordre de priorité. Les *user stories* du premier tableau correspondent aux objectifs du projet. Elles ont la priorité *MUST* selon la méthode MoSCoW et sont considérées comme vitales. Celles du second tableau ont été ajoutées après la première discussion avec l'étudiant en Master.

Tableau 18: Product backlog initial

Numéro	User story	Story points
13	En tant qu'utilisateur, je veux pouvoir accéder à un écran d'accueil pour pouvoir jouer au jeu	1
1	En tant qu'utilisateur, je veux pouvoir choisir le père pour créer un enfant par la suite	3
12	En tant qu'utilisateur, je veux pouvoir choisir la mère pour pouvoir créer un enfant par la suite	3
9	En tant qu'utilisateur, je veux pouvoir créer un enfant pour observer la correspondance entre ses gènes et son apparence	5
3	En tant qu'utilisateur, je veux pouvoir glisser-déposer les gènes des parents correspondants aux caractéristiques physiques sur les chromatides de l'enfant	3
4	En tant qu'utilisateur, je veux pouvoir visualiser l'apparence de l'enfant créé	8
8	En tant qu'utilisateur, je veux pouvoir enregistrer un enfant	2
7	En tant qu'utilisateur, je veux pouvoir consulter une liste des enfants déjà créés	3
16	En tant qu'utilisateur, je veux pouvoir observer un enfant existant pour avoir une vue plus détaillée.	2

5	En tant qu'utilisateur, je veux pouvoir ajouter un nouvel enfant à partir des mêmes parents que l'enfant précédent	2
11	En tant qu'utilisateur, je veux pouvoir retourner à l'écran d'accueil après avoir observé un enfant existant pour pouvoir continuer le jeu	1
	Total	33

Source : données de l'auteur

Tableau 19: User Story supplémentaires

Numéro	User story	Story points	Priorité selon MoSCoW
6	En tant qu'utilisateur, je veux obtenir une aide pour comprendre le jeu	3	Should
10	En tant qu'utilisateur, je veux pouvoir quitter l'écran d'aide pour continuer le jeu	1	Should
14	En tant qu'utilisateur, je veux pouvoir partager ce que j'ai enregistré pour pouvoir le consulter sur un autre appareil	2	Could
15	En tant qu'utilisateur, je veux pouvoir importer des enfants pour les observer sur un autre appareil	5	Could
17	En tant qu'utilisateur, je veux pouvoir accéder à un menu d'importation pour pouvoir importer des enfants créés sur d'autres appareils	2	Could
18	En tant qu'utilisateur, je veux pouvoir entrer mon nom pour pouvoir retrouver mon travail	2	Could
	Total	15	Could

Source : données de l'auteur

6.3.3. Definition of done

Pour permettre la modification du statut d'une User Story à *closed* sur Taiga, nous avons défini des critères autorisant l'équipe de développement à considérer une fonctionnalité comme prête à être validée par le *product owner*. Celles-ci ont été regroupées dans le tableau 20.

Tableau 20: Definition of done

Numéro	Critère
1	Les critères d'acceptance sont respectés
2	Tous les tests unitaires sont exécutés avec succès

3	La fonctionnalité a été testée par une personne extérieure à l'équipe

Source : données de l'auteur

6.3.4. Sprint 1

Objectif :	Pouvoir choisir les parents et accéder à l'écran de création de l'enfant avec les informations nécessaires (code génétique)
Début :	15 avril
Fin :	30 avril
User stories :	13, 1, 12, 9
Nombre de *story points* :	12

Le nombre de *story points* du premier *sprint* a été défini en divisant le nombre total par le nombre de *sprints* estimé en début de projet, ce qui donne 12. Les *user stories* choisies ont été reprises dans le tableau 21.

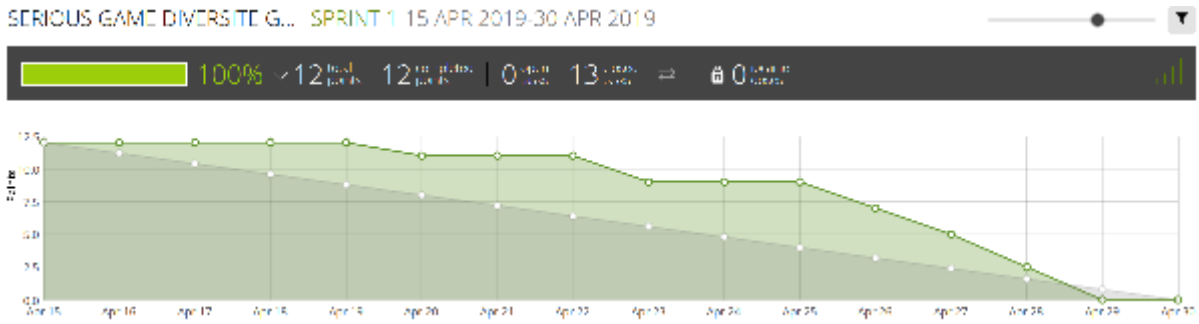
Tableau 21: User Stories du Sprint 1

Numéro	User story	Story points
13	En tant qu'utilisateur, je veux pouvoir accéder à un écran d'accueil pour pouvoir jouer au jeu	1
1	En tant qu'utilisateur, je veux pouvoir choisir le père pour créer un enfant par la suite	3
12	En tant qu'utilisateur, je veux pouvoir choisir la mère pour pouvoir créer un enfant par la suite	3
9	En tant qu'utilisateur, je veux pouvoir créer un enfant pour observer la correspondance entre ses gènes et son apparence	5

Source : données de l'auteur

Le développement s'est déroulé dans les délais impartis, comme illustré dans la figure 73. Le *product owner* a validé toutes les *user stories*. Cependant, il a fait la demande d'afficher les gènes sous forme de code plutôt que d'images pour *l'user story* 9.

Figure 73: Burn-down chart du Sprint 1



Source : taiga.io

6.3.5. Sprint 2

Objectif :	Pouvoir créer un enfant et visualiser le résultat
Début :	15 avril
Fin :	30 avril
User stories :	3, 4
Nombre de *story points* :	11

Étant donné l'avancement du sprint précédent, nous avons choisi de poursuivre au même rythme en choisissant une vélocité à atteindre de 11 *story points*. Les *user stories* correspondantes sont détaillées dans le tableau 22.

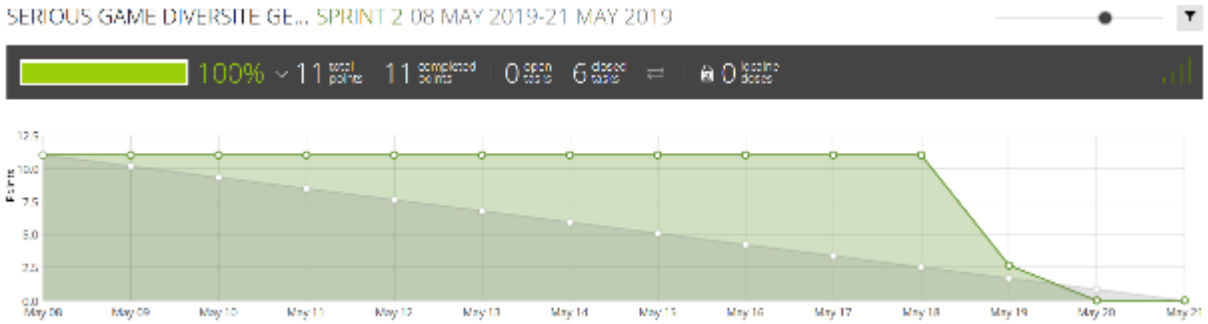
Tableau 22: User Stories du Sprint 2

Numéro	User story	Story points
3	En tant qu'utilisateur, je veux pouvoir glisser-déposer les gènes des parents correspondants aux caractéristiques physiques sur les chromatides de l'enfant	3
4	En tant qu'utilisateur, je veux pouvoir visualiser l'apparence de l'enfant créé	8

Source : données de l'auteur

Les fonctionnalités ont également été validées par le *product owner*, nous octroyant une vélocité de 11 *story points* pour ce *sprint*. Le *burn-down chart* correspondant est représenté à la figure 74. On y constate que les fonctionnalités ont été terminées en fin de période, en raison des corrections à apporter selon la demande du *product owner* au *sprint* précédent.

Figure 74: Burn-down chart du Sprint 2



Source : taiga.io

6.3.6. Sprint 3

Objectif :	Sauvegarder les enfants et visualiser la liste sur l'écran d'accueil, accéder à un écran d'aide
Début :	24 juin
Fin :	8 juillet
User stories :	6, 10, 8, 7
Nombre de *story points* :	9

Ce *sprint* suit la phase de test de l'application et a nécessité des corrections de l'application avant la poursuite du développement. Pour cette raison, nous avons diminué le nombre de *story points* à un total de 9. Les *user stories* sont présentés dans le tableau 23.

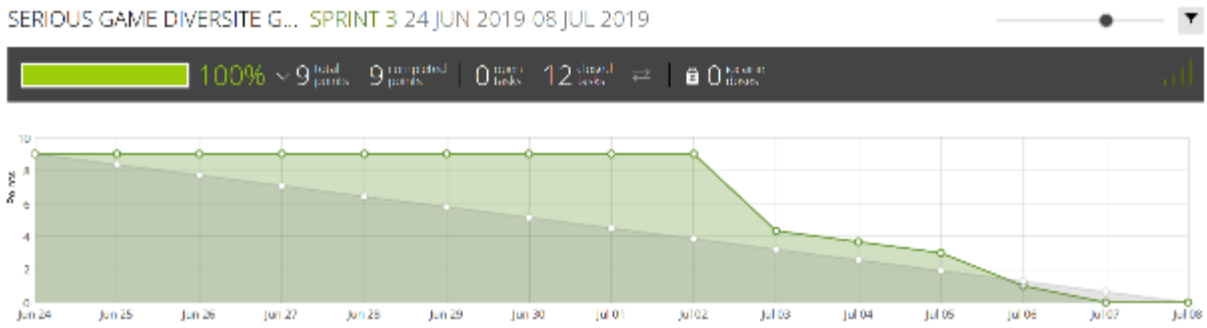
Tableau 23: User Stories du Sprint 3

Numéro	User story	Story points
6	En tant qu'utilisateur, je veux obtenir une aide pour comprendre le jeu	3
10	En tant qu'utilisateur, je veux pouvoir quitter l'écran d'aide pour continuer le jeu	1
8	En tant qu'utilisateur, je veux pouvoir enregistrer un enfant	2
7	En tant qu'utilisateur, je veux pouvoir consulter une liste des enfants déjà créés	3

Source : données de l'auteur

Comme pour le *sprint* précédent, les corrections à apporter ont retardé la livraison des fonctionnalités et celles-ci ont été achevées dans la seconde moitié du temps imparti, comme illustré à la figure 75. Nous avons également consacré plus de temps à la rédaction du rapport

Figure 75: Burn-down chart du Sprint 3



Source : taiga.io

6.3.7. Sprint 4

Objectif :	Voir les détails d'un enfant, améliorer l'écran d'une aide au jeu
Début :	9 juillet
Fin :	23 juillet
User stories :	16, 11, 53
Nombre de *story points* :	7

Après discussion avec le *product owner*, nous avons décidé de stopper le développement de l'application à la fin de ce *sprint*, avec l'ajout d'une dernière fonctionnalité, à savoir, le fait de pouvoir consulter les détails d'un enfant depuis l'écran d'accueil. Nous avons également proposé d'améliorer l'écran d'aide réalisé lors du *sprint* précédent pour une meilleure expérience utilisateur. Pour ce faire, nous avons ajouté une nouvelle *user story* dans le p*roduct backlog*. Les *user stories* correspondantes sont listées dans le tableau 24.

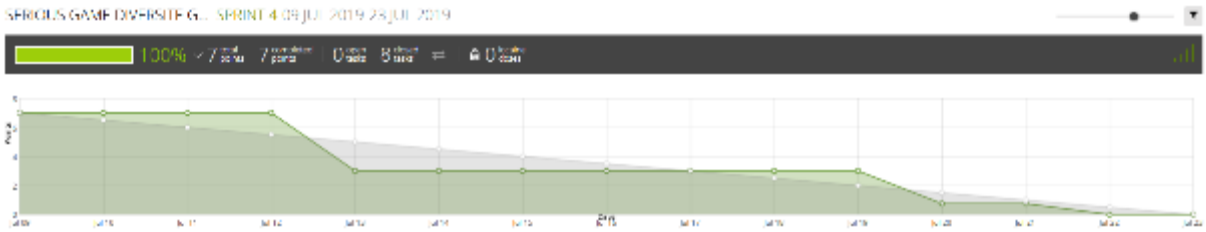
Tableau 24: User Stories du Sprint 4

Numéro	User story	Story points
16	En tant qu'utilisateur, je veux pouvoir observer un enfant existant pour avoir une vue plus détaillée.	3
11	En tant qu'utilisateur, je veux pouvoir retourner à l'écran d'accueil après avoir observé un enfant existant pour pouvoir continuer le jeu	1
53	Amélioration de l'écran d'aide	3

Source : données de l'auteur

Les fonctionnalités ont été achevées dans les délais, en parallèle de la rédaction du rapport, ce qui correspond à une vélocité de 7 pour ce *sprint*. L'accomplissement des fonctionnalités a été présenté à la figure 76.

Figure 76: Burn-down chart du Sprint 4



Source : taiga.io

6.3.8. Évolution du product backlog pendant le projet

Le contenu du *product backlog* a été modifié durant le développement de l'application. Lors du dernier *sprint*, nous avons ajouté une *user story* supplémentaire afin de couvrir l'amélioration de l'écran d'aide. Nous avons également redéfini le nombre de *story points* de *l'user story* 16 en raison des modifications à apporter à l'interface graphique pour afficher le code génétique des parents. Les détails de ces modifications sont présentés dans le tableau 25.

Tableau 25: Modifications apportées au Product backlog

Numéro	User story	Story points estimés au début du projet	Modifications apportées
16	En tant qu'utilisateur, je veux pouvoir observer un enfant existant pour avoir une vue plus détaillée.	2	Nouvelle valeur pour les Story Points : 3
53	En tant qu'utilisateur, je veux pouvoir accéder à plusieurs écrans d'aide afin de mieux connaître le fonctionnement du jeu	-	Nouvelle User Story Story Points : 3

Source : données de l'auteur

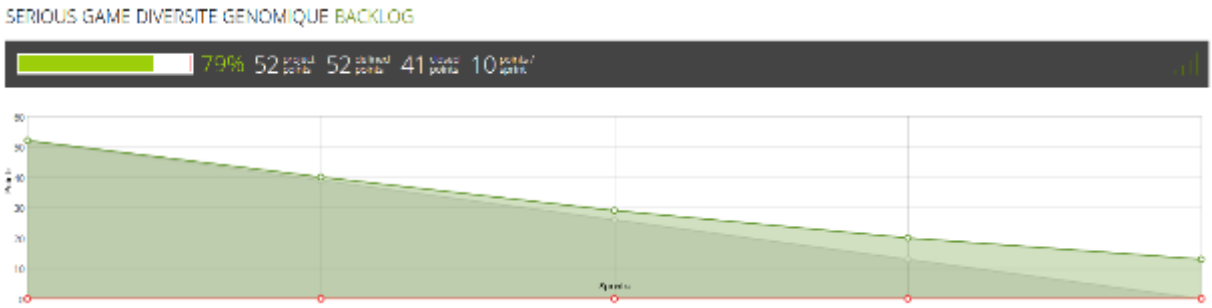
Par conséquent, le nombre de *story point* du projet a été augmenté de 4 pour un total de 52 au lieu de 48.

6.3.9. Vue d'ensemble du projet

La figure 77 montre que 79% du développement a été réalisé, soit 39 *story points* sur 52. Bien que le *product backlog* n'ait pas été entièrement complété, nous avons répondu aux objectifs définis au début du projet, et avons même ajouté une fonctionnalité supplémentaire en intégrant un écran d'aide au jeu.

En outre, l'application réalisée a été estimée comme utilisable en tant qu'outil d'enseignement par le *product owner*. Cela signifie que nous avons produit davantage qu'un prototype et que nous avons été au-delà de ses attentes.

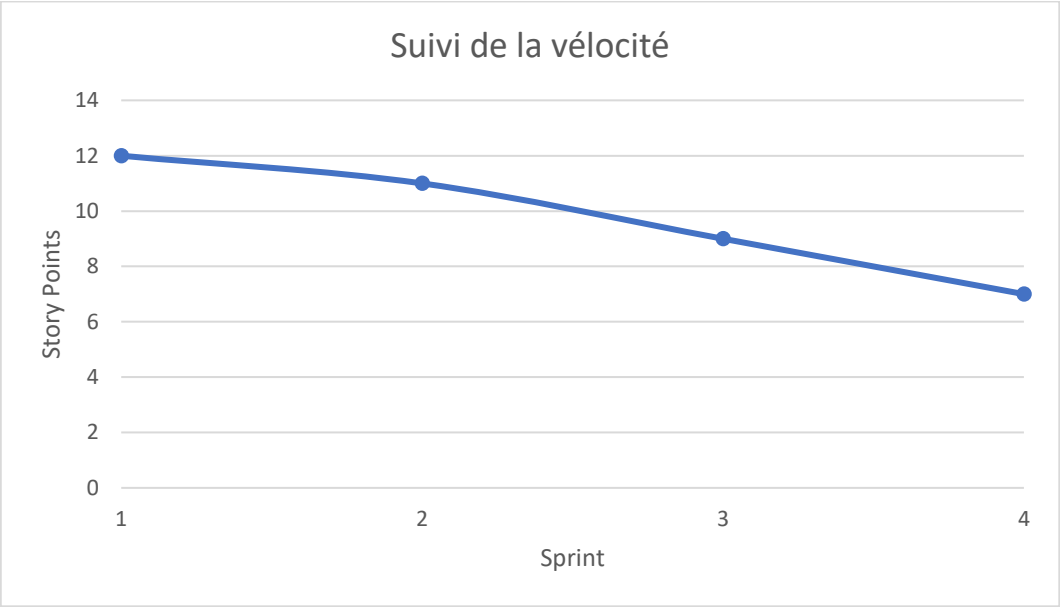
Figure 77: Burn-down chart final de l'ensemble du projet



Source : taiga.io

L'évolution de la vélocité au cours du projet a été représentée dans la figure 78. Elle a diminué au fur et à mesure de l'avancement du projet en passant de 12 *story points* lors du premier *sprint*, à 7 pour le dernier. Toutefois, ces chiffres sont légèrement faussés par la création de la dette technique décrite au chapitre 6.1. En effet, nous avons expliqué que la rédaction du rapport avait été repoussée à plus tard au profit de l'implémentation des fonctionnalités et que le retard a été rattrapé par la suite, dès le 3ème *sprint*. Cela se manifeste par un rendement plus élevé en début de projet, mais ne signifie pas que le travail fourni lors du 3ème et 4ème *sprint* ait été moins important. Celui-ci ne se reflète tout simplement pas dans le suivi de cette valeur.

Figure 78: Suivi de la vélocité



Source : données de l'auteur

7. Bilan

Avant ce travail, nous avions déjà utilisé Unity et Visual Studio afin de développer une application dans le cadre de notre formation. Cependant, ce projet n'avait pas demandé la création d'un programme à partir de zéro. Il s'agissait de concevoir une interface simple à partir de laquelle il était possible de démarrer plusieurs jeux qui ont été achetés sur *l'asset store*. Nous avions également ajouté quelques fonctionnalités telles que l'enregistrement du temps de jeu et avions préparé le transfert de ces informations à une API. En revanche, lors du présent travail, nous avons dû créer l'intégralité de l'interface graphique et du code source. Cela nous a permis d'approfondir nos connaissances de cet éditeur.

Ce travail de fin d'études nous a également permis de gérer un projet individuellement du début jusqu'à la fin. Nous avons pu démontrer notre capacité à estimer notre charge de travail par rapport aux délais à tenir. Les objectifs de chaque *sprint* ont été atteints. Nous avons également pu développer nos talents de négociation, l'étudiant en Master étant très enthousiaste et nous demandant de nombreux ajouts au fil du projet.

La collaboration avec l'étudiant en Master s'est bien passée. Nous nous sommes réunis régulièrement tout au long du projet et avons partagé nos connaissances. Il s'est déclaré satisfait du travail fourni et estime que l'application qui résulte de ce projet est utilisable en l'état. Nous en tirons une grande satisfaction.

Cela n'a pas été sans difficultés, la première à mentionner étant le problème d'adaptation aux différentes résolutions d'écran. Malgré le fait que nous ayons trouvé une solution, celle-ci a eu pour conséquence de perturber l'affichage du déplacement du code lors du glisser-déposer, diminuant de ce fait la qualité de l'expérience utilisateur et le côté intuitif de l'application. Nous aurions aimé disposer de davantage de temps pour trouver une façon de convertir la position de l'objet concerné entre la 2D et la 3D.

Une autre difficulté à mentionner est la gestion des scripts. Nous disposons d'une grande liberté dans l'affectation des scripts, et sans organisation systématique, il est aisé de se retrouver avec des doublons qui compliqueront la tâche lors de la maintenance de l'application. Ceci montre l'importance de la documentation, où nous avons recensé la liaison des scripts aux différents objets composant l'interface graphique.

Nous n'étions également pas satisfaits de la superposition des éléments sur la scène de l'éditeur. Celle-ci l'a rapidement rendue illisible et inutilisable. Ce problème a demandé de nombreuses heures de documentation, pour finalement trouver une solution simple, mais efficace. Bien qu'il ne soit pas recommandé de mettre à jour l'éditeur pendant un projet en cours, dans ce cas précis, cela a été payant.

Nous avons présenté le concept de la dette technique dans le chapitre 6.1. Elle nous a permis de respecter les délais malgré la charge de travail provenant des projets menés en parallèle dans le cadre de notre formation. Cependant, celle-ci a aussi entraîné une charge de travail supplémentaire, lorsqu'il a fallu réviser et améliorer le code après la phase de test. Cela a entraîné un sentiment d'insatisfaction par rapport à la qualité du code, et aussi de surmenage en raison de la nécessité de devoir revenir sur un travail effectué lors des *sprints* précédents tout en devant répondre aux attentes du *product owner* par rapport aux fonctionnalités à ajouter. Cela nous a également permis de mesurer les enjeux de la gestion de projet, en faisant l'expérience de se positionner par rapport aux demandes du client et aux attentes des développeurs qui seront amenés à maintenir cette application.

Toutefois, nous avons apprécié cette expérience professionnelle qui nous a permis de mettre en pratique les connaissances acquises durant les quatre années de cette formation.

CONCLUSION

L'application qui résulte de ce travail répond aux objectifs fixé en début de projet. Elle permet de choisir des parents, de construire le code génétique d'un enfant par glisser-déposer de gènes et de visualiser rapidement son apparence. Il est également possible de comparer les caractéristiques physiques et le code génétique des géniteurs et de leur descendance. En addition à cela, nous avons défini un écran d'aide expliquant l'utilisation de chaque écran, sans pour autant diminuer le côté intuitif et éducatif du jeu. Le *product owner* a estimé qu'elle constitue un outil d'enseignement tout à fait satisfaisant.

Étant donné que la phase de test s'est déroulée avant que le développement ne soit complété, nous pensons qu'il serait pertinent de récolter à nouveau des retours d'information après l'utilisation de cette application dans le contexte de cours de sciences. Ceci permettrait de soulever d'éventuels points à améliorer et de vérifier que l'expérience utilisateur est agréable, du point de vue du public cible.

Comme le *product owner* estime que l'application est suffisamment aboutie pour être utilisée en cours, une prochaine étape consisterait à adapter le jeu pour d'autres plateformes telles que Windows Mobile et iOS. Le choix de l'outils de développement a été effectué en ce sens, et cette démarche ne devrait demander que quelques ajustements.

Des fonctionnalités supplémentaires pourraient être implémentées dans le futur, telles que le partage d'enfants entre plusieurs appareils et la possibilité d'entrer le nom de la personne qui utilise l'application afin de pouvoir retrouver le travail de chaque élève. Le jeu a été développé de manière à ce que le nombre de caractéristiques physiques à observer puisse être augmenté. Cela offrirait une diversité encore plus large d'enfants à observer.

Le déplacement du code lors du glisser-déposer nécessite également d'être amélioré. Nous avons finalement supprimé cette caractéristique, car le problème a été jugé perturbant pour l'expérience utilisateur. Cependant, nous trouverions plus élégant de la rétablir.

RÉFÉRENCES

Adobe. (2019). *Adobe XD*. Récupéré sur Adobe: https://www.adobe.com/fr/products/xd.html#

Android Developers. (2019). *Android Studio*. Récupéré sur Android Developers: https://developer.android.com/studio

Avokiddo. (s.d.). *DNA Play*. Récupéré sur Google Play: https://play.google.com/store/apps/details?id=com.avokiddo.games.dnaplay

Bioinformatics, S. I. (s.d.). *Genome Jumper*. Récupéré sur sib: https://genome-jumper.sib.swiss/fr/

Charlesvi. (2013, septembre 21). *Converting Mouse Position to World. Stationary Camera*. Récupéré sur Unity Community: https://answers.unity.com/questions/540888/converting-mouse-position-to-world-stationary-came.html

Continisio, C. (19, novembre 2018). *Choosing the resolution of your 2D art assets*. Récupéré sur Unity Blog: https://blogs.unity3d.com/pt/2018/11/19/choosing-the-resolution-of-your-2d-art-assets/

Games, K. (2014, décembre 12). Unity UI Drag and Drop Tutorial. Récupéré sur https://www.youtube.com/watch?v=c47QYgsJrWc

Google. (2019). *Acme*. Récupéré sur Google Fonts: https://fonts.google.com/specimen/Acme

Imapler. (2014, septembre 5). *Create local database in unity3d*. Récupéré sur Stackoverflow: https://stackoverflow.com/questions/25668150/create-local-database-in-unity3d

Jacquet, R. (2019, juin 27). *COMPARATIF / Quelle tablette tactile choisir ?* Récupéré sur Les Numériques: https://www.lesnumeriques.com/tablette-tactile/comparatif-choisir-sa-tablette-tactile-connectee-a1048.html

Kasbi, Y. (2012, avril 23). *Pajama, un jeu de simulation sur la génétique*. Récupéré sur Le Blog SeriousGame.be: http://blog.seriousgame.be/pajama-un-jeu-de-simulation-sur-la-gntique

Lancelin-Golbery, M. (2019, janvier 2019). *Comment télécharger et installer Android Studio*. Récupéré sur FrAndroid: https://www.frandroid.com/android/developpement/563584_comment-telecharger-installer-android-studio

Marquez, E. (2018, septembre 24). *Qu'est-ce que Flutter, l'outil permettant de créer des applications Android et iOS ?* Récupéré sur FrAndroid:

https://www.frandroid.com/android/535194_quest-ce-que-flutter-loutil-permettant-de-creer-des-applications-android-et-ios

Philipp. (2015, février 25). *Unity: Make camera center of screen*. Récupéré sur StackExchange: https://gamedev.stackexchange.com/questions/95661/unity-make-camera-center-of-screen

Programmer. (2017, août 23). *Serialize and Deserialize Json and Json Array in Unity*. Récupéré sur Stackoverflow: https://stackoverflow.com/questions/36239705/serialize-and-deserialize-json-and-json-array-in-unity

Sanchez, E. (2012, juillet). *"Serious games" ou "serious play" ? Interactions et apprentissages*. Récupéré sur educ-revues.fr: http://www.educ-revues.fr/ARGOS/AffichageDocument.aspx?iddoc=41674

Siddiqui, D. (2015, octobre 7). *Creating Dynamic Scrollable Lists with New Unity Canvas UI*. Récupéré sur Folio 3: https://www.folio3.com/blog/creating-dynamic-scrollable-lists-with-new-unity-canvas-ui/

Sven Johann, E. W. (2013, août 20). *Gérer la dette technique* . Récupéré sur InfoQ: https://www.infoq.com/fr/articles/managing-technical-debt/

Taiga Agile LLC. (2019). *Home*. Récupéré sur Taiga: https://taiga.io/

Un jeu amusant sur l'hérédité et la génétique. (2018, avril 16). Récupéré sur Thot Cursus: https://cursus.edu/formations/21020/un-jeu-amusant-sur-lheredite-et-la-genetique#.XKEJyqTgqUm

Unity Games Programming For Beginners. (2017, juin 13). How to Pass Variables Between Scripts in C#. Récupéré sur Youtube: https://www.youtube.com/watch?v=ck6OyNBC95c

Unity Technologies. (2019). *Android Player Settings*. Récupéré sur Unity Documentation: https://docs.unity3d.com/Manual/class-PlayerSettingsAndroid.html

Unity Technologies. (2019). *Application.persistentDataPath*. Récupéré sur Unity Documentation: https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html

Unity Technologies. (2019). *Building apps for Android*. Récupéré sur Unity Documentation: https://docs.unity3d.com/Manual/android-BuildProcess.html

Unity Technologies. (2019). *Designing UI for Multiple Resolutions*. Récupéré sur Unity Documentation: https://docs.unity3d.com/Manual/HOWTO-UIMultiResolution.html

Unity Technologies. (2019). *JSON Serialization*. Récupéré sur Unity Documentation: https://docs.unity3d.com/Manual/JSONSerialization.html

Unity Technologies. (2019). *Prefabs*. Récupéré sur Unity Documentation: https://docs.unity3d.com/Manual/Prefabs.html

Unity Technologies. (2019). *Scene visibility*. Récupéré sur Unity Documentation: https://docs.unity3d.com/Manual/SceneVisibility.html

Unity Technologies. (2019). *Unity*. Récupéré sur Unity 3D: https://unity3d.com/unity

Vaccaro, L. (2015, mai 1). *Resize sprite to match camera width*. Récupéré sur Stack exchange: https://gamedev.stackexchange.com/questions/99321/resize-sprite-to-match-camera-width

Wikipedia. (2019, mars 11). *Chromosome*. Consulté le avril 8, 2019, sur Wikipedia: https://fr.wikipedia.org/wiki/Chromosome#Chromosomes_chez_les_procaryotes

Wikipedia. (2019, mars 31). *Génétique*. Consulté le avril 8, 2019, sur Wikipédia: https://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9tique

Wikipedia. (2019, mars 6). *Hérédité*. Consulté le avril 8, 2019, sur Wikipedia: https://fr.wikipedia.org/wiki/H%C3%A9r%C3%A9dit%C3%A9#Un_exemple_:_l'h%C3%A9r%C3%A9dit%C3%A9_de_la_couleur_des_yeux

Wikipedia. (2019, janvier 31). *Méthode MoSCoW*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/M%C3%A9thode_MoSCoW

DÉCLARATION DE L'AUTEUR

Je déclare, par ce document, que j'ai effectué le travail de Bachelor ci-annexé seule, sans autre aide que celles dûment signalées dans les références, et que je n'ai utilisé que les sources expressément mentionnées. Je ne donnerai aucune copie de ce rapport à un tiers sans l'autorisation conjointe du RF et du professeur chargé du suivi du travail de Bachelor, y compris au partenaire de recherche appliquée avec lequel j'ai collaboré, à l'exception des personnes qui m'ont fourni les principales informations nécessaires à la rédaction de ce travail et que je cite ci-après : Nicole Glassey Balet, Zhan Liu.

ANNEXE I : cahier des charges de l'étudiant en Master

1 Introduction

À la suite de mon cours de transposition didactique à la Haute Ecole Pédagogique du Valais (HEPVS), l'enseignante en charge de ce cours m'a proposé de développer une application numérique en lien avec un jeu que j'ai créé pour l'enseignement de la génétique au secondaire I (cycle d'orientation). Le cahier des charges concernera le développement d'une application numérique en lien avec le jeu que j'ai imaginé.

2 Buts du projet (application numérique)

Ce projet a pour objectif d'enseigner ou du moins de faire comprendre aux utilisateurs de l'application la répartition aléatoire des gènes parentaux aux futurs nouveau-nés. Le jeu permettra également de démontrer que deux individus (la même femme et le même homme) peuvent donner naissance à de nombreux individus très différents les uns des autres. Cette application mettra aussi en avant le côté aléatoire de la sélection des gènes de chacun des parents, ce qui découlera sur la diversité génétique (chaque utilisateur sera libre de faire ses propres choix dans la sélection des gènes parentaux). Cette application pourra permettre aux utilisateurs de « jouer » tout en apprenant (serious games).

Ce projet permettra également de rendre un jeu « papier » sous forme numérique. Ainsi, le nombre de pièces (cartes, images, cordelettes, …) composant le jeu sera réduit à une seule application numérique. Cela permettra de nombreux avantages décrits ci-dessous.

- Aucune usure ou perte de matériel (cartes qui se déchirent, perte d'une carte composant le jeu, …).
- Quantité de matériel réduit à une application numérique
- Création instantanée du nouvel individu (enfant)
- Autres ???

Finalement, le projet permettra de faire collaborer des étudiants (ingénieurs) de la HES-SO de Sierre avec un étudiant en Sciences de la HEPVS. Chaque intervenant aura comme objectif de mettre à contribution ses savoirs afin de développer une application numérique.

Je m'occuperai du fonctionnement du jeu (règles), de la création des documents scientifiques (table de codage, code génétique choisi, termes spécifiques à la génétique, …) et du design de l'interface (1er croquis). Les ingénieurs de la HES-SO auront comme tâche de développer une application numérique de ce jeu. C'est-à-dire :

- Créer le code informatique
- Développer le jeu sur une plateforme bien précise (IOS ou autre)
- Permettre aux utilisateurs de bouger des éléments de l'interface de manière tactile

- Créer un code informatique permettant de visualiser instantanément le nouvel individu
- Eventuellement intégrer un moyen de sauvegarder les données (à voir encore)
- …

3 Présentation du projet

Ce projet a pour but de développer une application permettant aux étudiants du secondaire de se familiariser avec certains principes de la génétique. Plus précisément, la **diversité génétique**. En effet, lors de la conception d'un nouvel individu (lors d'une reproduction sexuelle), la moitié des gènes proviennent de la mère alors que l'autre moitié est donnée par le père. De plus, ni la mère ni le père peuvent choisir les gènes qu'ils transmettent à leur futur enfant. C'est cette notion de transmission aléatoire des gènes qui doit être travaillée dans cette application et c'est ce choix aléatoire qui va permettre à chaque être humain d'être différent (diversité génétique). C'est cette conception que l'on souhaite faire apprendre aux étudiants manipulant cette application. Ils pourront également voir que ce sont les chromosomes qui portent l'information génétique (gènes).

Afin de pouvoir bien comprendre l'activité, il est primordial que les étudiants sachent :

- Que les gènes d'un enfant proviennent à la fois de la mère (50%) ainsi que du père (50%).
- Que se sont les chromosomes qui portent les gènes (information génétique)
- Qu'il y a plusieurs gènes sur un même chromosome
- Que les chromosomes sont composés de deux chromatides (un brin du chromosome)

De plus, la génétique est un domaine très complexe, j'ai donc dû réfléchir à rendre ce sujet plus simple à comprendre (vulgariser la matière). J'ai donc opté pour supprimer certains aspects de la génétique afin de faciliter la compréhension de ce sujet aux élèves.

Nous n'aborderons pas les notions suivantes :

- La récessivité et dominance des gènes ni la notion d'allèles
- Le fait que certains traits physiques sont codés par plusieurs gènes
- Les aspects environnementaux pouvant agir sur la génétique
- Les maladies génétiques
- Les OGM (organismes génétiquement modifiés)

Néanmoins, nous initierons le principe de dominance grâce à l'allèle noir des cheveux nommé N (cf. table de codage).

4 Le jeu (application numérique)

Afin de pallier aux difficultés que pourraient rencontrer les étudiants sur ce thème, j'ai pensé à développer un jeu de génétique. Dans le but de ne pas parler de récessivité ni de dominance, j'ai créé en amont une table de codage afin que les élèves n'aient pas à réfléchir aux résultats lorsque l'on combine deux gènes (les utilisateurs ne devront pas définir eux-mêmes l'apparence physique des individus qu'ils vont créer).

Exemple : Le père donne le gène bleu pour les yeux alors que la mère donne un gène brun. L'enfant aura donc les yeux bruns car le brun domine sur le bleu. En créant une table de codage en amont (qui sera intégrée à l'application), l'utilisateur du jeu n'aura pas à réfléchir si le brun domine ou non sur le bleu car l'application le fera automatiquement. Il verra simplement deux gènes associés qui donneront un résultat spécifique.

Cette activité aura pour objectif de démontrer que deux individus peuvent donner naissance à de nombreux individus très différents les uns des autres. Elle mettra également en avant le côté aléatoire de la sélection des gènes parentaux, ce qui découlera sur la diversité génétique.

5 Les règles du jeu

L'utilisateur sera libre dans ses choix afin que le résultat final soit totalement aléatoire d'un utilisateur à un autre. Néanmoins, le jeu possède plusieurs règles décrites ci-dessous

5.1 Règles du jeu

1. Choisir les individus parents (un homme et une femme) parmi l'échantillon proposé.
2. Prendre un des deux gènes de la mère codant pour une caractéristique physique spécifique et le glisser sur une chromatide du chromosome de l'enfant.
3. Prendre un des deux gènes du père codant pour la même caractéristique physique et le glisser sur la chromatide « libre » de l'enfant.
4. Réaliser les étapes 2 et 3 pour toutes les caractéristiques physiques des individus parents.
5. Observer l'enfant crée
6. *Phase facultative* : création d'un nouvel individu avec les mêmes individus parents en cliquant sur le +

Ainsi, lors des résultats, les participants pourront comparer leur enfant avec tous les autres utilisateurs. C'est là qu'interviendra la diversité génétique et que les utilisateurs devront se questionner sur les raisons de toutes ces différences physiques entre les divers enfants créés → répartition aléatoire des gènes parentaux !

Afin de mieux comprendre les règles du jeu, j'ai mis à disposition un exemple de démarche dans les annexes. (*cf. exemple phase 1 ; exemple phase 2*)

6 L'interface (1er croquis)

L'interface que j'ai imaginée possède 4 phases :

- ***Phase 1*** : la sélection des individus parents (*cf. image 1*)
- ***Phase 2*** : Glisser les gènes parentaux sur le chromosome de l'enfant (*cf. image 2*)
- ***Phase 3*** : Visualisation de l'enfant créé avec le visage de l'enfant et la composition du chromosome (*cf. image 3*)
- ***Phase 4*** : Le récapitulatif des actions de l'utilisateur lors des étapes 1 à 3 ET la possibilité de créer un nouvel enfant en cliquant sur le +. (*cf. image 4*)

6.1 La phase 1

Lorsque l'utilisateur débutera le jeu, il devra commencer par choisir deux individus (un homme et une femme) afin de définir qui seront les parents de l'enfant qu'il va créer. Il pourra choisir parmi 5 visages de femme en haut de l'écran et 5 visages d'homme en bas de l'écran. L'utilisateur aura l'obligation de glisser une femme dans le carré de gauche et un homme dans le carré de droite. En aucun cas il sera possible de glisser un visage féminin dans le carré droit et inversement pour les visages masculins.

Dès lors, l'application affichera deux chromosomes portant les gènes respectifs des deux individus choisis. Une fois ses choix faits, l'utilisateur pourra appuyer sur le bouton « SUIVANT » afin de passer à la phase 2. (*cf. exemple phase 1*)

6.2 La phase 2

A ce moment, l'utilisateur verra apparaître à l'écran le chromosome de la femme (la mère) sur la gauche ainsi que le chromosome de l'homme (le père) sur la droite. Au centre de l'écran, l'utilisateur verra en plus grand le chromosome vierge (pour le moment) de l'enfant.

Il devra ensuite glisser un des deux gènes de la mère codant pour une caractéristique particulière sur une chromatide du chromosome de l'enfant. Il glissera ensuite un des deux gènes du père codant pour la même caractéristique sur la 2ème chromatide de l'enfant. L'utilisateur effectuera ces actions pour chaque caractéristique physique des parents.

Il est important que l'utilisateur ne puisse pas glisser les deux gènes de la mère codant pour le même trait physique sur le chromosome de l'enfant. En effet, une fois que l'utilisateur aura « pris » un gène chez la mère, il ne pourra pas prendre le 2ème gène de la mère codant pour cette même caractéristique. Il devra absolument aller chercher

un gène du père codant pour ce trait physique. Une fois qu'il aura distribué les gènes parentaux pour toutes les caractéristiques physiques, l'utilisateur pourra passer à la phase 3 en cliquant sur le bouton « SUIVANT ». (*cf. exemple phase 2*)

6.3 La phase 3

A ce stade, l'application affichera sur la gauche la combinaison de gènes choisit par l'utilisateur et sur le reste de l'écran l'apparence physique de l'enfant qui vient d'être créé.

Dans cette phase, il n'y a aucune manipulation à faire, il suffit d'observer le résultat obtenu suite à l'association de nos deux individus parents. Une fois cette observation faite, l'utilisateur peut passer à la phase 4 en cliquant sur le bouton « SUIVANT ». (*cf. image 3*)

6.4 La phase 4

Dès lors, l'application affichera en haut de l'écran les chromosomes des deux parents sélectionnés et au bas de l'écran un encadré avec les enfants créés. Pour le moment, l'application affichera uniquement un seul enfant dans l'encadré mais l'utilisateur aura la possibilité de cliquer sur le bouton « + » afin de pouvoir créer un nouvel enfant et l'application le redirigera directement à la phase 2. Une fois à la phase 2, l'utilisateur pourra une nouvelle fois choisir aléatoirement les gènes qu'il place sur le chromosome de l'enfant. Et ainsi de suite jusqu'à la création maximale de 6 enfants. (*cf. image 4*)

Idée à développer pour cette phase : permettre à l'utilisateur d'exporter ou de sauvegarder ses résultats…

7 La table de codage

Dans ce jeu, il y aura 6 caractéristiques physiques qui pourront changer d'un individu à un autre. Toutes les apparences physiques sont codées par des gènes. Chaque gène peut s'exprimer sous différentes formes (exemple : gène des yeux → yeux bleus, verts, bruns, …). Chaque individu possède deux allèles pour chaque gène, 1 allèle sur chaque chromatide (un brin du chromosome).

Qu'est-ce qu'un allèle ? Prenons l'exemple du gène qui code pour le sexe de l'individu. Les femmes sont XX et les hommes XY. Les femmes possèdent donc deux allèles identiques qui sont les X alors que les hommes possèdent deux allèles différents, un X et un Y. L'allèle est une sorte de sous-catégorie des gènes.

Les différents gènes du jeu

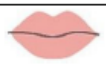
- Couleur de la peau
- Couleur des yeux

- Couleur des cheveux
- Forme de la bouche
- Forme du nez
- Sexe de l'individu

Les allèles de chaque gène

- Peau : Blanche (b), Noire (n1), Asiatique (a) (**3 allèles différents**)
- Yeux : Bleu (bl), Noir (n2), Brun (br1) (**3 allèles différents**)
- Couleur des cheveux : Blond (bd), Brun (br2), Noir (N) (**3 allèles différents**)
- Forme de la bouche : forme 1 (bo1), forme 2 (bo2), forme 3 (bo2) (**3 allèles différents**)
- Forme du nez : Triangle (tr), Rond (ro), Carré (ca) (**3 allèles différents**)
- Sexe de l'individu : X ou Y (**2 allèles différents**)

Les caractéristiques de chaque gène

Caractéristiques physiques	Types de possibilités					
Couleur de la peau	Blanche	Noire	Asiatique	Métissée noir et blanc	Métissée blanc et asiatique	Métissée noir et asiatique
Couleur des yeux	Bleu clair	Bleu foncé	Brun clair	Brun foncé	Vert	Noir
Couleur des cheveux	Blond	Brun clair	Brun foncé	Noir		
Forme de la bouche						
Forme du nez	Triangle pointe vers le bas	Triangle pointe vers le haut	Rond	Ovale en longueur	Ovale en largeur	Carré
Sexe de l'individu	Femme	Homme				

Remarque : Toutes les abréviations sont en minuscule SAUF pour la couleur noire des cheveux. En effet, l'allèle des cheveux noirs possède l'abréviation N, c'est-à-dire que cet allèle est dominant. Chaque fois que cet allèle est associé à un autre allèle, ce sera forcément la couleur noire qui sortira sur l'aspect physique de la personne. Tous les autres allèles sont co-dominants et le résultat de chaque association d'allèles se trouve dans le tableau ci-dessous.

La table de codage

Type de gène	Gène (allèle) du parent 1	+	Gène (allèle) du parent 2	Résultat physique
Gène 1 : Couleur de la peau	Blanche *(b)*	+	Blanche *(b)*	Peau blanche
	Blanche *(b)*	+	Noire *(n1)*	Métissé noir et blanc
	Blanche *(b)*	+	Asiatique *(a)*	Métissé blanc et asiatique
	Noire *(n1)*	+	Noire *(n1)*	Peau noire
	Noire *(n1)*	+	Asiatique *(a)*	Métissé noir et asiatique
	Asiatique *(a)*	+	Asiatique *(a)*	Asiatique
Gène 2 : Couleur des yeux	Bleu *(bl)*	+	Bleu *(bl)*	Bleu clair
	Bleu *(bl)*	+	Brun *(br1)*	Vert
	Bleu *(bl)*	+	Noir *(n2)*	Bleu foncé
	Brun *(br1)*	+	Brun *(br1)*	Brun clair
	Brun *(br1)*	+	Noir *(n2)*	Brun foncé
	Noir *(n2)*	+	Noir *(n2)*	Noir
Gène 3 : Couleur des cheveux	Blond *(bd)*	+	Blond *(bd)*	Blond
	Blond *(bd)*	+	Brun *(br2)*	Brun clair

	Blond *(bd)*	+	Noir *(N)*	Noir
	Brun *(br2)*	+	Brun *(br2)*	Brun foncé
	Brun *(br2)*	+	Noir *(N)*	Noir
	Noir *(N)*	+	Noir *(N)*	Noir
Gène 4 : Forme de la bouche	Forme 1 (f1)	+	Forme 1 (f1)	
	Forme 1 (f1)	+	Forme 2 (f2)	
	Forme 1 (f1)	+	Forme 3 (f3)	
	Forme 2 (f2)	+	Forme 2 (f2)	
	Forme 3 (f3)	+	Forme 2 (f2)	
	Forme 3 (f3)	+	Forme 3 (f3)	
Gène 5 : Forme du nez	Triangle *(tr)*	+	Triangle *(tr)*	
	Triangle *(tr)*	+	Rond *(ro)*	
	Triangle *(tr)*	+	Carré *(ca)*	
	Rond *(ro)*	+	Carré *(ca)*	
	Rond *(ro)*	+	Rond *(ro)*	
	Carré *(ca)*	+	Carré *(ca)*	

Gène 6 : Sexe de l'individu	X	+	X	Femme
	X	+	Y	Homme

Remarque : Une combinaison d'allèles de ce type : ro + ca est identique à la combinaison ca + ro.

8 Conclusion

Les étudiants du secondaire I savent en général qu'ils ressemblent à leurs parents car ils prennent quelque chose d'eux (leurs gènes). Ils savent également que nous sommes tous uniques (empreintes, ADN, …) et que notre apparence physique reflète nos gènes. Cependant, je souhaite qu'il puisse expérimenter cela grâce à cette application afin qu'ils comprennent que la diversité génétique provient du fait que nos gènes sont choisis aléatoirement entre notre mère et notre père.
Ils pourront également expérimenter cela en créant plusieurs enfants issus des mêmes parents. Mon souhait et qu'après qu'ils aient créé 4-5 enfants provenant des mêmes parents qu'ils remarquent que chaque individu créé est unique et différent des autres (*Dans le rectangle de l'image 4*).

Ce projet n'a pas pour objectif de retranscrire à la lettre le fonctionnement de la génétique, car en effet, la génétique n'est pas aussi simple que cela. Mais plutôt de permettre aux jeunes étudiants de comprendre le fonctionnement de base de la distribution des gènes et de découvrir la provenance de la diversité génétique.

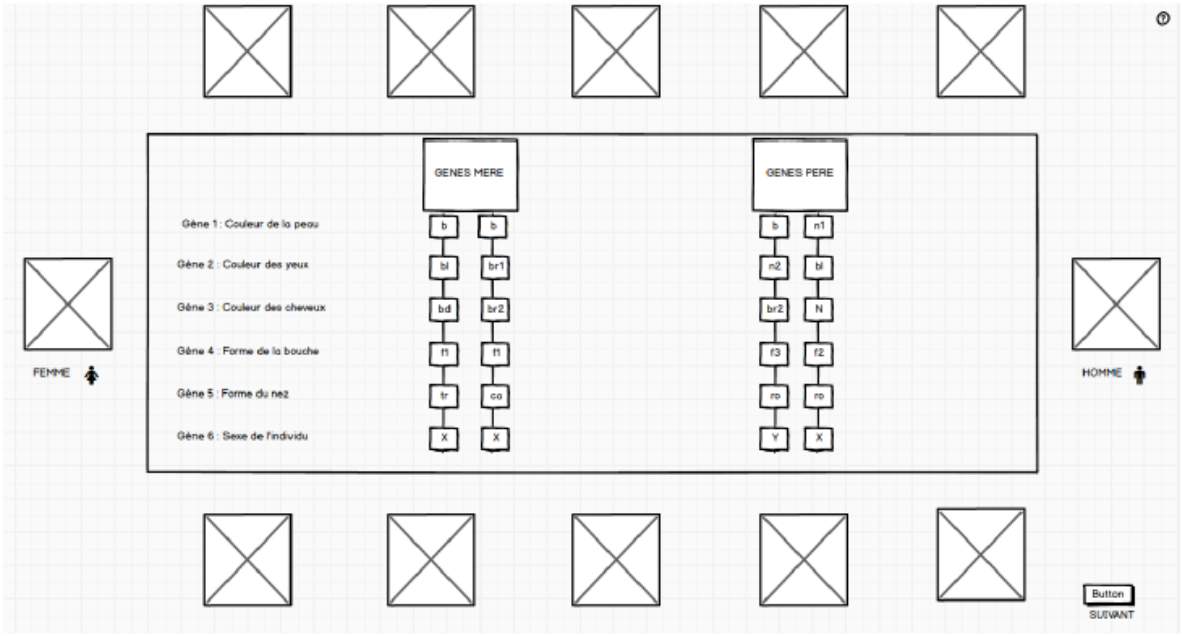
9 Annexes

Voici ci-dessous les différentes annexes :

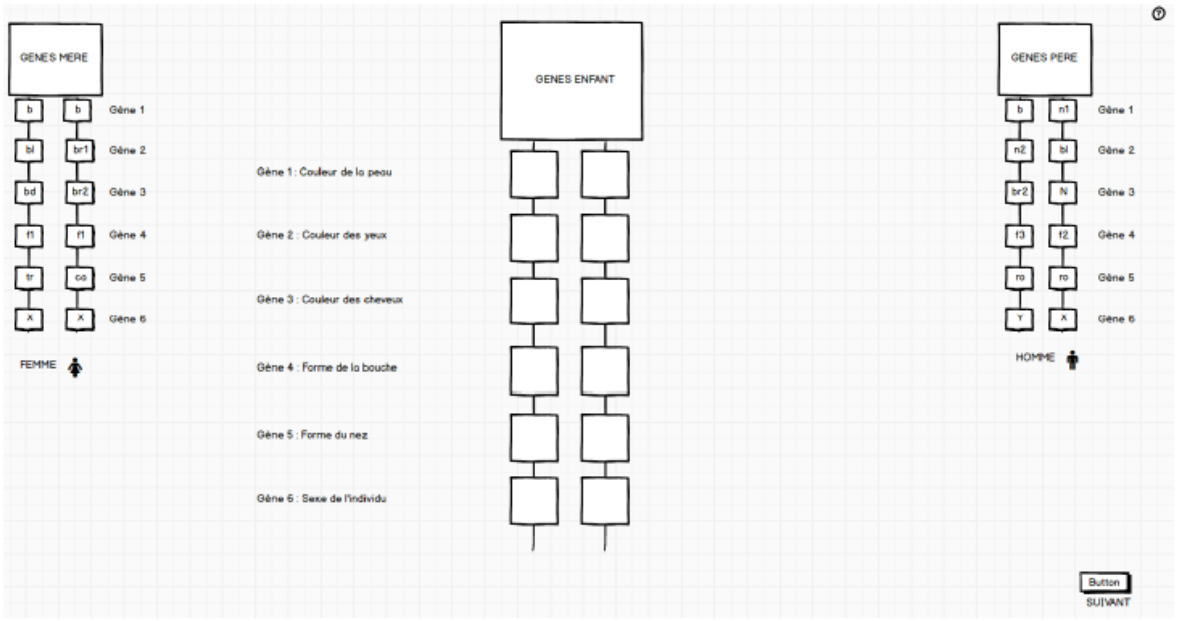
- Image 1
- Image 2
- Image 3
- Image 4
- Exemple phase 1
- Exemple phase 2
- Coordonnées personnelles

ANNEXE II : mockups de l'étudiant en Master

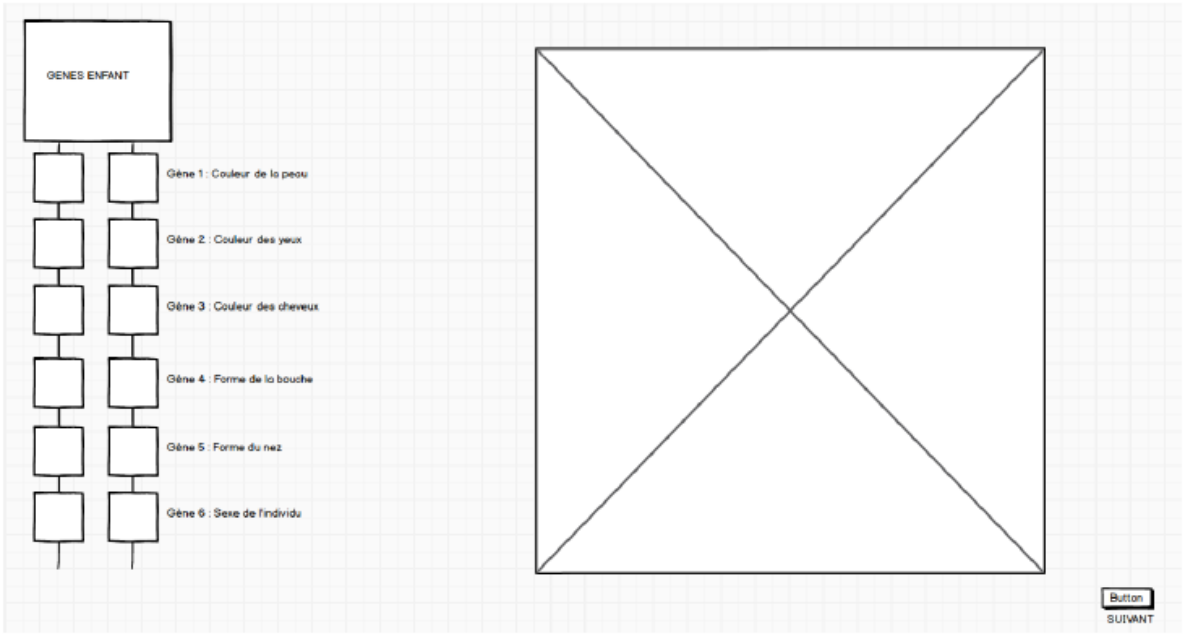
9.1.1 Image 1



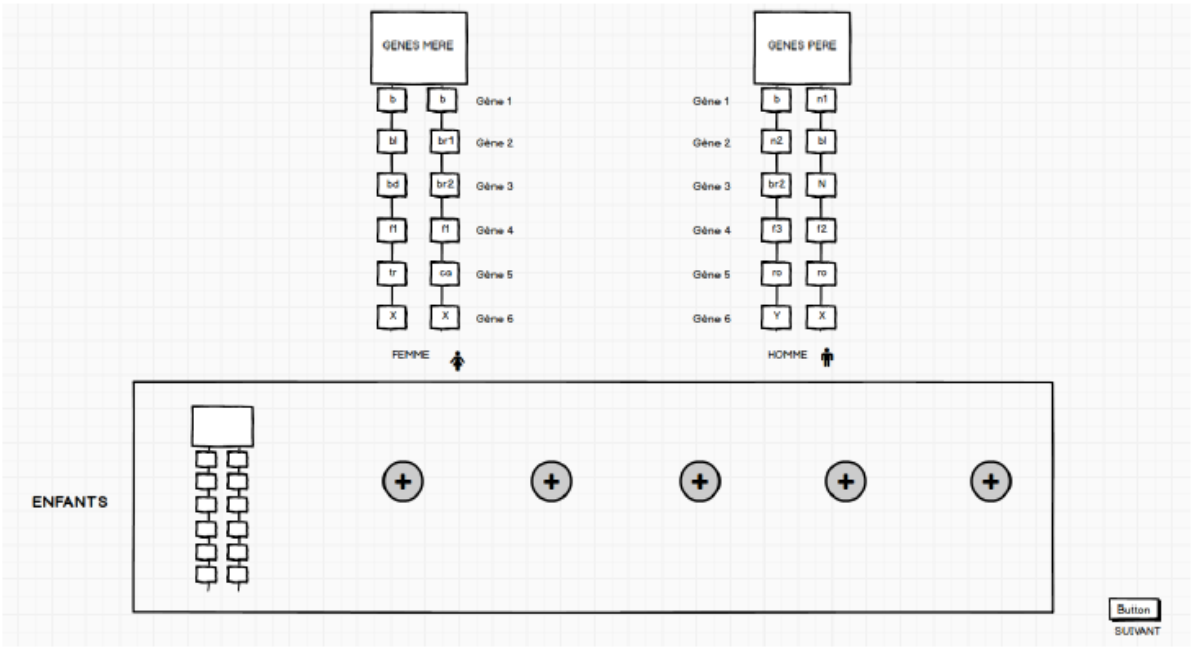
9.1.2 Image 2



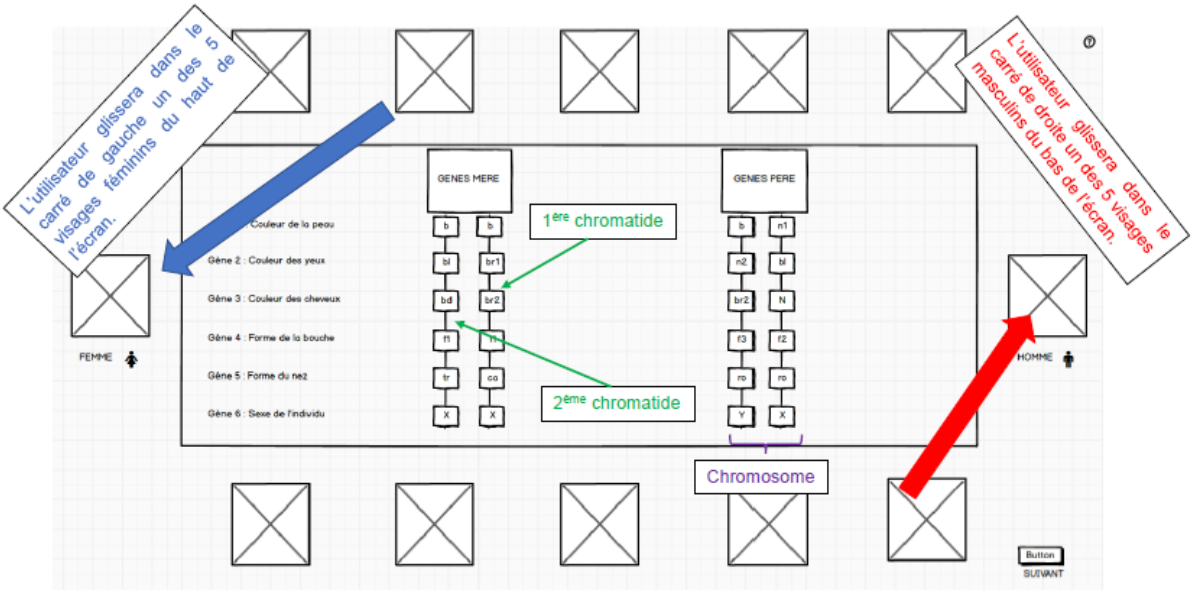
9.1.3 Image 3



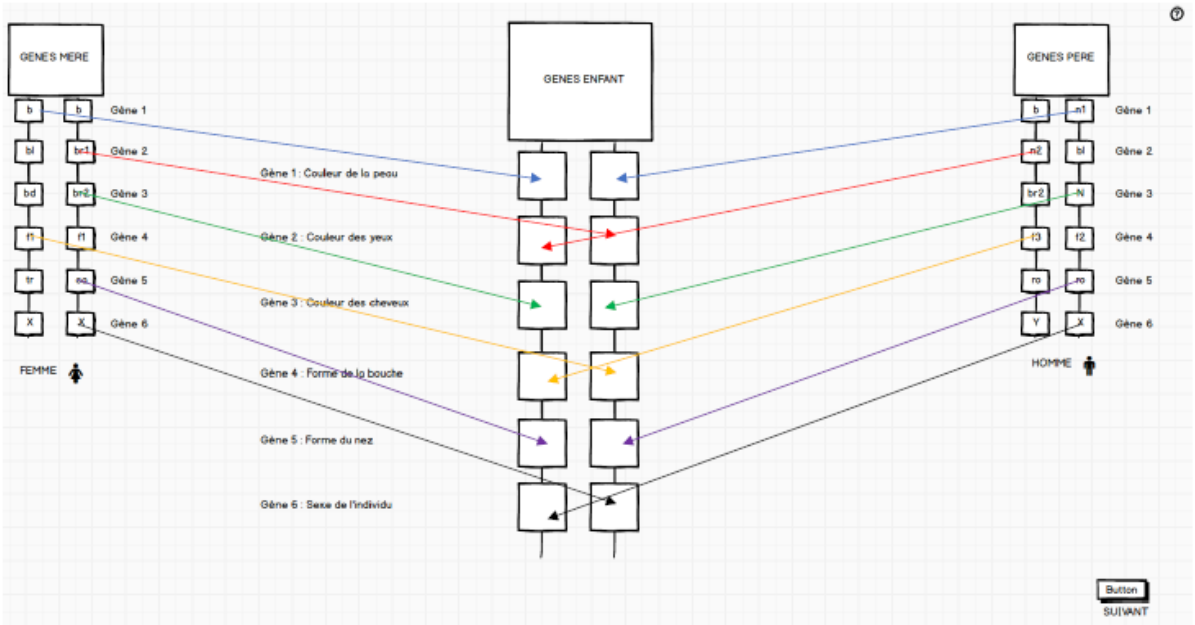
9.1.4 Image 4



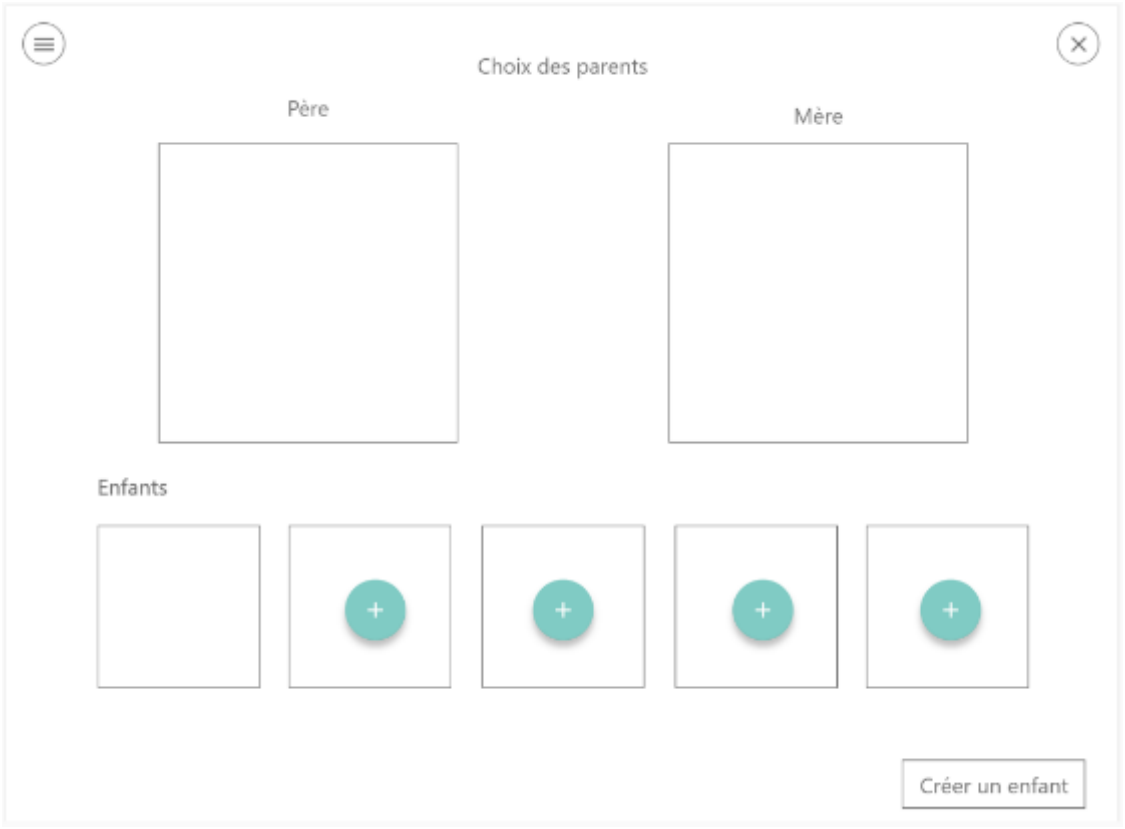
9.1.5 Exemple phase 1

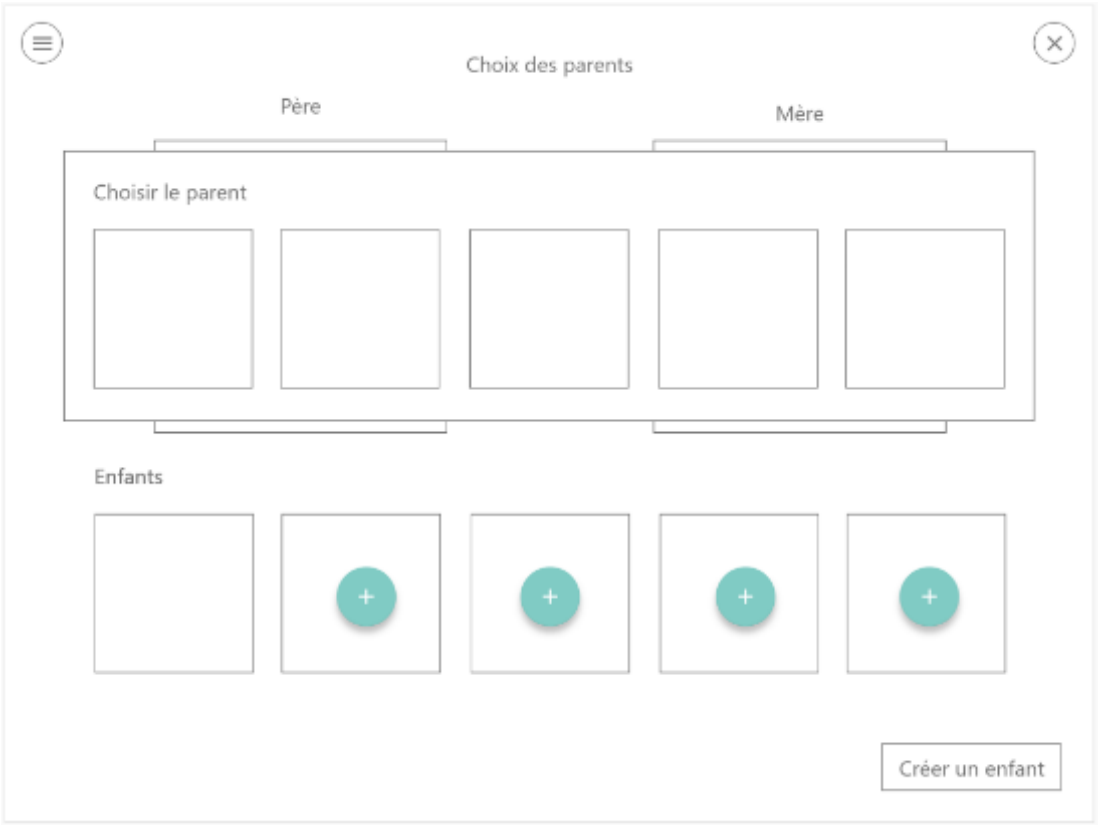


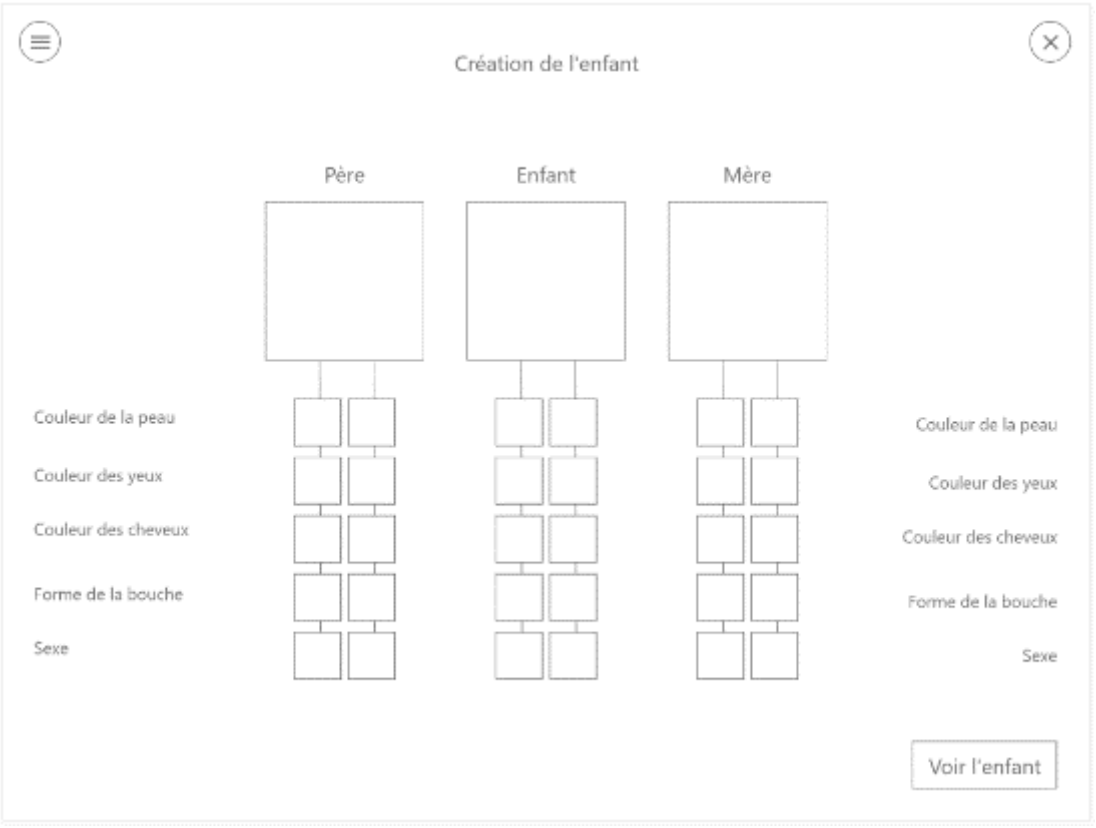
9.1.6 Exemple phase 2

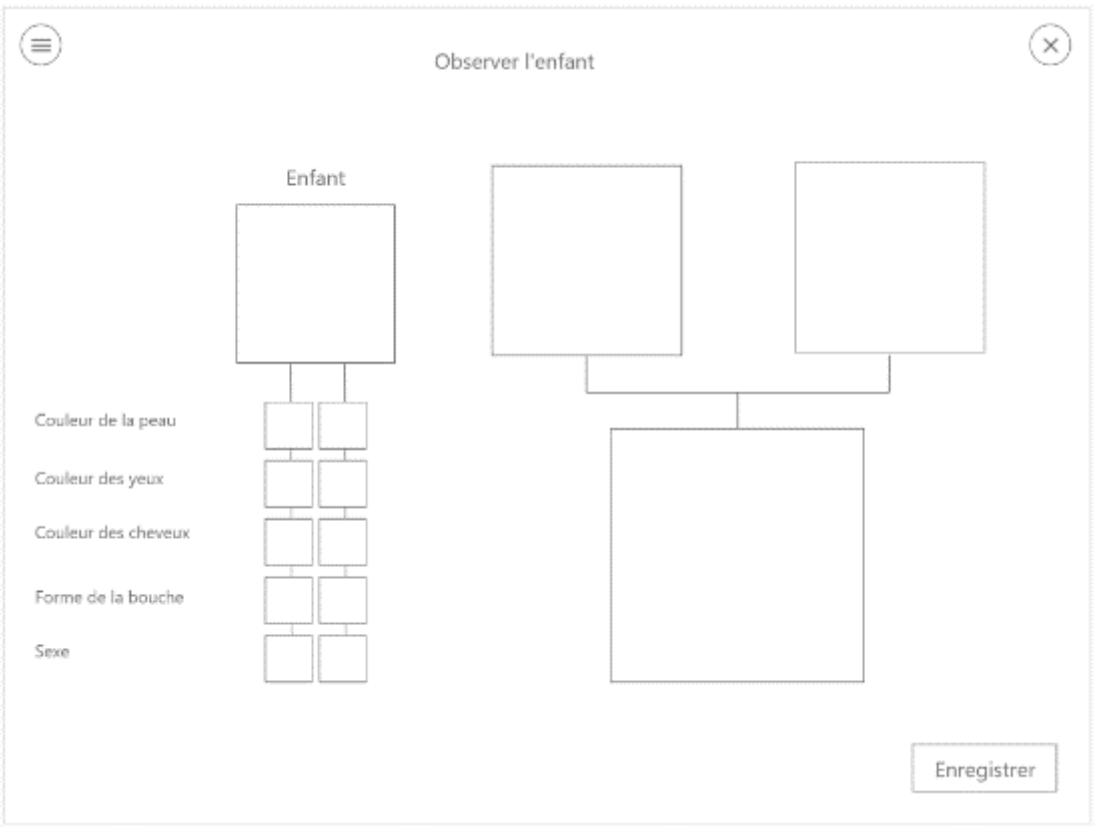


ANNEXE III : mockups






Création de l'enfant
Père
Enfant
Mère
Couleur de la peau
Couleur des yeux
Couleur des cheveux
Forme de la bouche
Sexe
Couleur de la peau
Couleur des yeux
Couleur des cheveux
Forme de la bouche
Sexe
Voir l'enfant


Observer l'enfant
Enfant
Couleur de la peau
Couleur des yeux
Couleur des cheveux
Forme de la bouche
Sexe
Enregistrer

ANNEXE IV : réponses aux questionnaires du test de l'application

Application Génétique

13 fiches ont été remplies : 12 élèves ont répondu au questionnaire et un enseignant

1. L'application est-elle facile d'utilisation ? Oui ☐ Non ☐

Les réponses comprennent : 13 OUI

2. Y a-t-il des fonctionnalités qui n'ont pas fonctionné ? Oui ☐ Non ☐

Les réponses comprennent : 13 NON

3. Qu'avez-vous remarqué sur l'apparence des enfants créés ?

Tous différents
Ils ressemblent à ses parents
Ils se ressemblent beaucoup
Que la répartition des gènes était bien respectée
Manque des caractères physiques (blond, blanc) : 3 fois
Manque des différents caractères physiques (couleur de peau, …)
Ils se ressemblent beaucoup et des fois moins : 3 fois
Ils se ressemblent la plupart du temps
Ils ressemblent beaucoup aux parents mais chaque bébé est différent

4. Comment pourrions-nous avoir deux enfants totalement identiques ?

En sélectionnant à chaque fois les mêmes gènes
Il faut choisir les mêmes gènes
Prendre un gène chez la mère et un gène chez le père qui sont identique
Il faut prendre les mêmes gènes de chaque côté
Jumeau
En mettant des mêmes caractères : 2 fois
En mettant les mêmes chromosomes : 3 fois
En mettant les mêmes chromosomes des mêmes parents
Si nous mettons la même colonne de gènes pour les deux enfants

5. Combien de temps as-tu mis pour créer un enfant ?

20 secondes : 1 fois
30 secondes : 4 fois
1 minute : 8 fois

6. Le « glisser-déposer » est-il facile à manipuler ? Oui ☐ Non ☐

13 oui

7. Le design de l'application vous a-t-il plu ? Oui ☐ Non ☐

10 OUI et 3 NON
Oui car les images sont mignonne et ressemblent au manga
Non car les couleurs sont trop sombres : 2 fois
Les couleurs sont un peu sombres et on ne remarque pas vraiment les décorations (un peu fade)

8. Avez-vous trouvé cette approche sur la génétique intéressante ? Si oui pourquoi ?

A compléter mais bonne introduction à la génétique (diversité)
Oui, c'est bien pour apprendre
Oui, il y a des images, c'est plus intéressant qu'un prof qui parle
Oui, car on comprend bien la différence entre chaque choix
Oui, drôle et ludique : 2 fois
Oui, elle est intéressante et ludique
On a pu voir quel chromosome dominait : 4 fois
Oui, mais un peu simple quand même

9. A la suite de cette activité, la transmission génétique vous a-t-elle paru plus concrète ?

Oui, pour autant qu'elle soit complétée
Oui : 4 fois
Oui j'ai mieux compris (après l'examen, yesss) : 2 fois
Oui, sans m'en rendre compte j'ai assimilé des choses mais dommage car c'est après le test
Oui, plus compréhensible : 4 fois
Ça va, oui

10. Quel est le but de cette application ?

Mettre le doigt sur la diversité génétique
Apprendre comment fonctionne la génétique
D'apprendre plus facilement
Nous faire être plus à l'aise avec la génétique
Comprendre, apprendre en s'amusant : 3 fois
Comprendre la génétique
Comprendre et expliquer la génétique : 4 fois
Nous montrer que nous prenons un gène du papa et de la maman par catégorie

ANNEXE V : documentation de l'application

Outils de développement

Cette application a été développée avec Unity 2019.1.8f1 et Microsoft Visual Studio Enterprise 2017 version 15.9.7.

Elle comporte également un *asset* acheté sur *l'asset store*, *Super Visual Chibis,* utilisé pour représenter les parents et les enfants. La documentation de ce module se trouve dans le dossier Documentation de *l'asset*.

Structure de l'application

L'interface graphique a été réalisée sur Unity. Le code source a été rédigé en C# à l'aide de Visual Studio. La structure de ces deux parties est précisée ci-dessous.

Unity Editor

MainScene :

- EventSystem
- RootCanvas
    - Background
    - HomeScreenCanvas
    - ParentSelectionCanvas
    - CreationScreenCanvas
    - ChildResultCanvas
    - ExitButton
    - HelpButton
    - HelpCanvasHome
    - HelpCanvasCreation
    - HelpCanvasResult
    - MainCamera

HomeScreenCanvas :

- TopCanvas
    - ParentPanel
        - FatherSlot : HomeChibiFather = Chibi prefab
        - MotherSlot : HomeChibiMother = Chibi prefab
    - ChildrenPanel
        - Child[n] = Chibi prefab (0 <= n <= 9)
    - ChildCountCanvas

- BottomCanvas
    - ChildCreationButton
        - Text

ParentSelectionCanvas :

- ParentSelectionPanel
    - ChibiPanel
        - ChibiSlot : ChibiParent[n] (0 < n <= 6)
    - ButtonPanel
        - PanelExitButton

CreationScreenCanvas :

- ChibiPanel
    - FatherSlot
        - CreationChibiFather
    - MotherSlot
        - CreationChibiMother
- Labels
    - FatherChromatides
    - ChildChromatides
    - MotherChromatides
- GenePanel
    - FatherGenesLabel
        - SkinSlot : text
        - EyeSlot : text
        - HairColorSlot : text
        - MouthSlot : text
        - GenderSlot : text
    - FatherGenes
        - SkinColorGene1 : text
        - SkinColorGene2 : text
        - EyeColorGene1 : text
        - EyeColorGene2 : text
        - HairColorGene1 : text
        - HairColorGene2 : text
        - MouthGene1 : text
        - MouthGene2 : text
        - GenderGene1 : text

        - GenderGene2 : text
    - CreationChildGenes
        - SkinColorGene1
        - SkinColorGene2
        - EyeColorGene1
        - EyeColorGene2
        - HairColorGene1
        - HairColorGene2
        - MouthGene1
        - MouthGene2
        - GenderGene1
        - GenderGene2
    - MotherGenes
        - SkinColorGene1 : text
        - SkinColorGene2 : text
        - EyeColorGene1 : text
        - EyeColorGene2 : text
        - HairColorGene1 : text
        - HairColorGene2 : text
        - MouthGene1 : text
        - MouthGene2 : text
        - GenderGene1 : text
        - GenderGene2 : text
    - MotherGenesLabel
        - SkinSlot
        - EyeSlot
        - HairColorSlot
        - MouthSlot
        - GenderSlot
- ButtonPanel
    - ResultChildButton

ChildResultCanvas :

- TopCanvas
    - FatherCanvas
        - ChibiCanvas
            - FatherSlot : ResultChibiFather (prefab)
        - TitleCanvas

- TitlePanel
  - Title
- ChromatidesCanvas
  - FatherGenesCanvas
    - SkinColorGene1 : text
    - SkinColorGene2 : text
    - EyeColorGene1 : text
    - EyeColorGene2 : text
    - HairColorGene1 : text
    - HairColorGene2 : text
    - MouthGene1 : text
    - MouthGene2 : text
    - GenderGene1 : text
    - GenderGene2 : text
  - ChildCanvas : même structure que FatherCanvas
  - MotherCanvas : même structure que FatherCanvas
- BottomCanvas
  - BackButton
  - SaveChildButton

HelpCanvas :

- Panel
  - Canvas
    - Image : ScrollRect et Mask
      - ScrollContentCanvas : texte et images
    - HelpScreenScrollbar
    - Canvas
- HideScreenButton

Visual Studio

Assembly-CSharp : application

- Assets
  - Data
    - Constants.cs : nom et index des sprites de l'*asset* utilisés
    - GeneCombination.cs : listes de KeyValuePair<code génétique, nom du sprite ou index du sprite>
    - Parents.cs : liste de ChibiGenes (caractéristiques physiques) des parents

- Objects
  - Chibi.cs

```
public class Chibi
    {
        public int SkinColor { get; set; }
        public int EyesColor { get; set; }
        public int Eyebrows { get; set; }
        public int Hair { get; set; }
        public int BackHair { get; set; }
        public Color HairColor { get; set; }
        public int Mouth { get; set; }

        public Chibi(int skinColor, int eyesColor, int eyebrows, int
hair, int backhair, Color hairColor, int mouth)
        {
            SkinColor = skinColor;
            EyesColor = eyesColor;
            Eyebrows = eyebrows;
            Hair = hair;
            BackHair = backhair;
            HairColor = hairColor;
            Mouth = mouth;
        }

        public string Print()
        {
            return "skin: " + SkinColor + ", eyes: " + EyesColor + ",
eyebrows: " + Eyebrows + ", hair: " + Hair
                + ", backhair: " + BackHair + ", haircolor: " +
HairColor + ", mouth: " + Mouth;
        }
    }
```

  - ChibiGenes.cs

```
public class ChibiGenes
    {
        public string SkinColor { get; set; }
        public string EyesColor { get; set; }
        public string HairColor { get; set; }
        public string Mouth { get; set; }
        public string Gender { get; set; }

        public ChibiGenes(string skinColor, string eyesColor, string
hairColor, string mouth, string gender)
        {
            SkinColor = skinColor;
            EyesColor = eyesColor;
            HairColor = hairColor;
            Mouth = mouth;
            Gender = gender;
        }
    }
```

- Child.cs

```
    [Serializable]
    public class Child
    {
        public string Parents;
        public string SkinColor;
        public string EyesColor;
        public string HairColor;
        public string Mouth;
        public string Gender;

        public Child(string parents, string skinColor, string
eyesColor, string hairColor, string mouth, string gender)
        {
            Parents = parents;
            SkinColor = skinColor;
            EyesColor = eyesColor;
            HairColor = hairColor;
            Mouth = mouth;
            Gender = gender;
        }

        public string Print()
        {
            return "Parents: " + Parents + ", skin : " + SkinColor +
", eyes: " + EyesColor +
                ", hair: " + HairColor + ", mouth: " + Mouth + ",
gender: " + Gender;
        }
    }
```

- GenePair.cs

```
public class GenePair
    {
        public string gene1 { get; set; }
        public string gene2 { get; set; }

        public string ConcatenateGenePairs()
        {
            SortGenePairsAlphabetically();
            return gene1 + gene2;
        }

        private void SortGenePairsAlphabetically()
        {
            List<string> list = new List<string>()
            {
                gene1,
                gene2
            };

            list.Sort();
            gene1 = list[0];
            gene2 = list[1];
        }
```

```
        public GenePair SplitGenePair(string genePair)
        {
            GenePair pair = new GenePair
            {
                gene1 = genePair.Substring(0, genePair.Length / 2),
                gene2 = genePair.Substring(genePair.Length / 2,
genePair.Length / 2)
            };
            return pair;
        }
    }
}
```

- Scripts
  - ApplicationNavigation.cs : affichage et masquage des écrans
  - ChildrenCreationBehaviour.cs : affichage du contenu de l'écran de création
  - DragHandler.cs : gestion du glisser
  - DropHandler.cs : gestion du déposer
  - GameManager.cs : stockage de variables à transmettre entre scripts
  - OnHomeScreenLoad.cs : affichage des enfants et de leur nombre depuis le fichier
  - ParentCreation.cs : affichage des parents dans le panneau de sélection à partir de la liste dans Parents.cs
  - ParentSelection.cs : affichage du parent sélectionné sur l'écran d'accueil
  - ParentSelectionPanelBehaviour.cs : fermeture du panneau de sélection des parents
  - ResultButtonBehaviour.cs : affichage ou masquage du bouton sur l'écran de création selon le nombre de gènes déposés
  - SaveToJsonFile.cs : sauvegarde de l'enfant créé dans un fichier en json
  - ShowChildResult.cs : affichage de l'enfant sur l'écran de visualisation du résultat, transmission des informations à GameManager en vue de la sauvegarde
  - ShowDetails.cs : affichage du contenu de l'écran de visualisation des détails
  - UndoDragAndDrop.cs : annulation du glisser-déposer quand on quitte le processus de création d'un enfant
- Utils
  - GeneCodeManagement.cs

```
public string GetSpritenameFromGeneCode(string characteristic, string
geneCode, bool isMale)
```

```
public int GetIndexFromGeneCode(string characteristic, string
geneCode, bool isMale)

public Color GetColorFromGeneCode(string geneCode)

public Chibi GetChibiFromChild(Child child, bool isMale)

public Chibi GetChibiFromChibiGenes(ChibiGenes genes, bool isMale)

public bool IsMale(string genderCode)

private GenePair GetGenePairFromSpritename(List<KeyValuePair<string,
string>> list, string spriteName)

private string GetSpritename(List<KeyValuePair<string, string>> list,
string geneCode)

private int GetIndex(List<KeyValuePair<string, int>> list, string
geneCode)
```

- JsonHelper.cs

```
public static class JsonHelper
    {
public static T[] FromJson<T>(string json)
        {
            Wrapper<T> wrapper =
JsonUtility.FromJson<Wrapper<T>>(json);
            return wrapper.Children;
        }

public static string ToJson<T>(T[] array)
        {
            Wrapper<T> wrapper = new Wrapper<T>();
            wrapper.Children = array;
            return JsonUtility.ToJson(wrapper);
        }

public static string ToJson<T>(T[] array, bool prettyPrint)
        {
            Wrapper<T> wrapper = new Wrapper<T>();
            wrapper.Children = array;
            return JsonUtility.ToJson(wrapper, prettyPrint);
        }

        [Serializable]
private class Wrapper<T>
        {
            public T[] Children;
        }
    }
```

- SpriteManagement.cs

```
public void SetSpriteFromChibiObject(SpriteSelector selector, bool
isMale, Chibi chibi)
        {
            ChibiCreator creator;
            creator = selector.chibiCreator;
            if (isMale)
            {
                selector.hairMidRenderer.sprite =
creator.HairMidSprites[Constants.MALE_MIDHAIR_INDEX];
                selector.hairBackRenderer.sprite =
creator.HairBackSprites[Constants.MALE_BACKHAIR_INDEX];
            }
            else
            {
                selector.hairMidRenderer.sprite =
creator.HairMidSprites[Constants.FEMALE_MIDHAIR_INDEX];
                selector.hairBackRenderer.sprite =
creator.HairBackSprites[Constants.FEMALE_BACKHAIR_INDEX];
            }

            selector.headRenderer.sprite =
creator.HeadSprites[chibi.SkinColor];
            selector.faceEyesRenderer.sprite =
creator.FaceEyesSprites[chibi.EyesColor];
            selector.faceEyebrowsRenderer.sprite =
creator.FaceEyebrowsSprites[Constants.EYEBROWS_INDEX];
            selector.faceEyebrowsRenderer.color = chibi.HairColor;
            selector.hairMidRenderer.color = chibi.HairColor;
            selector.hairBackRenderer.color = chibi.HairColor;
            selector.faceRestRenderer.sprite =
creator.FaceRestSprites[chibi.Mouth];
        }
```

- TransferCode.cs

```
private void Awake()
    {
        gameManager = GameObject.FindObjectOfType<GameManager>();
    }

public void ParentTransferCode()

public void ChildTransferCode(int childIndex)

private void GetParentsGenes()

private void SplitGenePair(string genePair, List<string> codes)
```

- TransferParentSprites.cs

```
// Source father and mother
public GameObject sourceFather;
public GameObject sourceMother;
// Destination father and mother
public GameObject destFather;
```

```
public GameObject destMother;

public void SetContent()
    {
        SpriteSelector fatherSelector =
sourceFather.GetComponent<SpriteSelector>();
        SpriteSelector motherSelector =
sourceMother.GetComponent<SpriteSelector>();
        SpriteSelector destFatherSelector =
destFather.GetComponent<SpriteSelector>();
        SpriteSelector destMotherSelector =
destMother.GetComponent<SpriteSelector>();

        SetParentSprites(fatherSelector, destFatherSelector);
        SetParentSprites(motherSelector, destMotherSelector);
    }

private void SetParentSprites(SpriteSelector sourceSelector,
SpriteSelector destSelector)
    {
        destSelector.headRenderer.sprite =
sourceSelector.headRenderer.sprite;
        destSelector.faceEyesRenderer.sprite =
sourceSelector.faceEyesRenderer.sprite;
        destSelector.faceEyebrowsRenderer.sprite =
sourceSelector.faceEyebrowsRenderer.sprite;
        destSelector.faceEyebrowsRenderer.color =
sourceSelector.faceEyebrowsRenderer.color;
        destSelector.hairMidRenderer.sprite =
sourceSelector.hairMidRenderer.sprite;
        destSelector.hairMidRenderer.color =
sourceSelector.hairMidRenderer.color;
        destSelector.hairBackRenderer.sprite =
sourceSelector.hairBackRenderer.sprite;
        destSelector.hairBackRenderer.color =
sourceSelector.hairBackRenderer.color;
        destSelector.faceRestRenderer.sprite =
sourceSelector.faceRestRenderer.sprite;
    }
```

Assembly-CSharp-Editor-firstpass : asset editor

Assembly-CSharp-firstpass : asset sprites et scripts

TBDiversiteGenomiqueTestProject : tests unitaires

- GenePairTest.cs

```
[TestMethod]
public void ConcatenateGenePairsNB()
        {
            GenePair pair = new GenePair();
            string result = "bn";
            pair.gene1 = "n";
            pair.gene2 = "b";
```

```
            Assert.AreEqual(pair.ConcatenateGenePairs(), result);
        }

[TestMethod]
public void ConcatenateGenePairsBN()
        {
            GenePair pair = new GenePair();
            string result = "bn";
            pair.gene1 = "b";
            pair.gene2 = "n";

            Assert.AreEqual(pair.ConcatenateGenePairs(), result);
        }

 [TestMethod]
public void ConcatenateGenePairsNOBR()
        {
            GenePair pair = new GenePair();
            string result = "brno";
            pair.gene1 = "no";
            pair.gene2 = "br";

            Assert.AreEqual(pair.ConcatenateGenePairs(), result);
        }

[TestMethod]
public void ConcatenateGenePairsF2F1()
        {
            GenePair pair = new GenePair();
            string result = "f1f2";
            pair.gene1 = "f2";
            pair.gene2 = "f1";

            Assert.AreEqual(pair.ConcatenateGenePairs(), result);
        }
```

Affectation des scripts dans l'éditeur

- RootCanvas : GameManager, SaveToJsonFile
- HomeScreenCanvas : ApplicationNavigation, OnHomeScreenLoad
- HomeScreenCanvas -> HomeChibis : ParentCreation
- ParentSelectionPanel : ParentSelectionPanelBehaviour
- ParentSelectionCanvas -> each ChibiParent : ParentSelection, ParentSelectionPanelBehaviour
- CreationScreenCanvas : ChildrenCreationBehaviour, ResultButtonBehaviour, TransferParentSprites
- CreationScreenCanvas -> FatherGenes and MotherGenes children : DropHandler
- CreationScreenCanvas -> CreationChildGenes children : DropHandler
- ChildResultCanvas : ShowChildResult, TransferParentSprites, TransferCode

Sauvegarde des enfants

Les enfants sont sauvegardés à l'aide de l'objet Child, du script SaveToJsonFile et de JsonHelper. Le code définissant les parents est issu de la concaténation de leurs index.

L'url du dossier est définie comme suit :

```
string filePath = Application.persistentDataPath + @"/children.json";
```

Cela correspond à :

- Android : /storage/emulated/0/Android/data/<packagename>/files
- iOS : /var/mobile/Containers/Data/Application/<guid>/Documents
- Windows store apps : %userprofile%\AppData\Local\Packages\<productname>\LocalState
- Windows : C:\Users\<userprofile>\AppData\LocalLow\<companyname>\<productname>

Exemple de json obtenu :

```
{
    "Children": [
        {
            "Parents": "00",
            "SkinColor": "nn",
            "EyesColor": "brbr",
            "HairColor": "bdbr",
            "Mouth": "f2f2",
            "Gender": "xy"
        },
        {
            "Parents": "00",
            "SkinColor": "bb",
            "EyesColor": "blbl",
            "HairColor": "bdbd",
            "Mouth": "f1f1",
            "Gender": "xx"
        },
        {
            "Parents": "14",
            "SkinColor": "bn",
            "EyesColor": "nono",
            "HairColor": "brNO",
            "Mouth": "f2f2",
            "Gender": "xy"
        }
    ]
}
```