I.U.T de Laval
Département INFO
52 Rue des Docteurs
Calmette et Guérin
53000 LAVAL
Téléphone : 02 43 59 49 01

LIUM
Site de Laval CERIUM²
15 Rue des Docteurs
Calmette et Guérin
53000 LAVAL
Téléphone : 02 43 83 38 58

Mise en œuvre d'un **système de traçage** des actions dans un jeu sérieux d'apprentissage et **visualisation** partielle de ces traces pour le superviseur au sein du projet ***Escape it!***

Rapport de stage en vue de l'obtention du Diplôme Universitaire de Technologie Informatique

Réalisé par **Louis Euvrard**

Stage effectué du 6 Avril au 19 Juin et soutenu le 22 Juin 2020

Maître de stage : **M. Pierre Laforcade**
Tuteur IUT : **M. Yann Walkowiak**

IUT Laval
Le Mans Université

LIUM
Laboratoire d'Informatique
Université du Mans

Résumé

Le LIUM (Laboratoire d'Informatique de l'Université du Mans) se consacre à la recherche, menée par deux équipes : l'équipe LST (Langage and Speech Technology) et IEIAH (Ingénierie des Environnement Informatiques pour l'Apprentissage Humain). C'est dans le cadre de celle-ci que j'ai pu réaliser ce stage, sur le projet de jeu sérieux *Escape it!*, sous la tutelle de M. Pierre Laforcade qui est le principal responsable de ce projet. Ce projet a été conçu pour offrir un outil aux parents d'enfants atteints de TSA (Troubles du Spectre Autistique) facilitant l'acquisition de compétences visuelles, outil pouvant être suggéré et suivi par un thérapeute. L'application se présente sous la forme d'un jeu mobile, reprenant un concept *escape-the-room*. Le sujet du stage proposait de fournir un système de traçage des parties de l'enfant, pour permettre au parent / superviseur de suivre plus en détail sa progression. Au cours de mon stage, j'ai pu développer un système de sauvegarde des traces de jeu d'un enfant, ainsi qu'une interface pour voir les différentes sessions, et un système de Replayer permettant de revoir en détail les parties de l'enfant.

Mots-clefs du stage :

- Jeu sérieux
- Objectif thérapeutique
- Autisme
- Unity
- Sérialisation / Désérialisation

Remerciements

Avant d'entrer en détail dans le rapport, je souhaiterai remercier un certain nombre de personnes.

Merci tout d'abord au corps enseignant de l'IUT, pour la qualité de l'apprentissage pendant ces deux années de DUT, qui ont été des outils nécessaires pour le bon déroulement et la réussite de ce stage.

Merci à Pierre Laforcade, mon maître de stage, pour son sujet de stage et son encadrement, qui m'a permis de progresser et de passer 11 semaines de travail très agréables et formatrices.

Merci à Yann Walkowiak pour son tutorat et ses observations sur la progression de mon stage, qui m'ont permis de me recentrer sur l'essentiel au bon moment.

Merci à Ludovic Hamon pour son investissement pendant la recherche des stages, ainsi que son écoute et son temps pour résoudre les quelques problèmes que j'ai pu rencontrer pendant ce stage.

Et merci à Quentin Couland, pour le soutien moral et ces conversations que nous avons pu avoir durant ces deux années. Encore félicitations pour ta thèse, Docteur.

IUT Laval
Le Mans Université

LIUM
Laboratoire d'Informatique
Université du Mans

Sommaire

1. Résumé p.2
2. Remerciements p.3
3. Introduction p.6
4. Présentation du contexte de recherche p.7
   - 4.1. Le laboratoire LIUM p.7
   - 4.2. L'équipe IEIAH p.8
5. Contexte du sujet de stage p.9
   - 5.1. Le projet *Escape it!* p.9
   - 5.2. Le jeu sérieux *Escape it!* p.9
   - 5.3. De nouveaux besoins p.10
   - 5.4. Missions du stage p.11
   - 5.5. Organisation et méthodes p.11
6. Travaux réalisés p.13
   - 6.1. Le système de traçage p.13
     - 6.1.1. Définition et explication des traces p.13
     - 6.1.2. Récupération des traces p.17
     - 6.1.3. Persistance des traces p.19
     - 6.1.4. Exploitation des traces p.21
   - 6.2. Le replayer p.26
     - 6.2.1. Complétion du fonctionnement et lien entre les traces récupérées p.26
     - 6.2.2. Définition et explication des traces p.28
     - 6.2.3. Récupération des traces p.31
     - 6.2.4. Persistance des traces p.31
     - 6.2.5. Rediffusion des traces p.32
7. Conclusion p.33
   - 7.1. Travaux réalisés p.33
   - 7.2. Bilan personnel p.34
8. Glossaire p.36
9. Bibliographie & webographie p.37

Table des figures

Figure 1 - Description d'une scène (CSEDU'18 - Escape it!) 10
Figure 2 - Maquette de l'écran des traces journalières 13
Figure 3 - Maquette de l'écran des traces des sessions 13
Figure 4 - Maquette de l'écran des traces par compétences 14
Figure 5 - Maquette de l'écran des traces des niveaux 14
Figure 6 - Maquette de l'écran des traces des niveaux, en fonction d'une compétence 14
Figure 7- Structure de la classe DayTrack 15
Figure 8- Structure de la classe SessionTrack 16
Figure 9 - Structure de la classe LevelTrack 16
Figure 10 - Structure de la classe LevelTrackSkill 17
Figure 11 - Structure de la classe SkillTrack 17
Figure 12- Séquence d'écrans permettant de lancer une partie 17
Figure 13 - Écran de lancement d'un niveau 18
Figure 14 - Cas où le bouton d'aide est pressé 18
Figure 15 - Cas de niveau terminé, avec porte ouverte 19
Figure 16 - Exemple de fonctions de sauvegarde et de chargement (ici pour un SkillTrackDB) 20
Figure 17 - Fichier de sauvegarde des DayTracks 21
Figure 18 - Fichier de sauvegarde des SkillTracks 21
Figure 19- Prefab d'une DateView 22
Figure 20 - Prefab d'une SessionView 22
Figure 21 - Prefab d'une LevelView 22
Figure 22 - Prefab d'une SkillLevelView 22
Figure 23 - Contenu d'un script de View 22
Figure 24 - Écran d'affichage des DayTracks 23
Figure 25 - Écran d'affichage des SessionsTracks 24
Figure 26 - Écran d'affichage des LevelTracks 24
Figure 27 - Écran d'affichage des SkillTracks (NOTE : cet écran est statique) 25
Figure 28 - Écran d'affichage des LevelSkillTracks 25
Figure 29 - Prefab d'une SkillTrack nulle 26
Figure 30 - Écran d'affichage d'une SkillTrack nulle 26
Figure 31- Panel du replayer 26
Figure 32 - Hiérarchie des scènes lors de la diffusion d'un replay 27
Figure 33 - Action d'une TouchTrack 29
Figure 34 - Détail de la classe TouchTrack 29
Figure 35 - Détail de la classe HiddenObjectTouchTrack 29
Figure 36 - Détail de la classe DragAndTouchTrack 30
Figure 37 - Détail de la classe VectorTime 30
Figure 38 - Intégration des sous types en XML 31
Figure 39 - Définition d'un tableau de sous types pour la sérialisation 31

Introduction

Ce stage de 11 semaines a été réalisé dans le but de valider la deuxième année de mon DUT Informatique, à l'IUT de Laval. J'ai travaillé pour le LIUM, le Laboratoire d'Informatique de l'Université du Mans. Malheureusement, en raison de la crise sanitaire de Covid-19, ce stage n'a pas pu se réaliser en présentiel, au sein des locaux du CERIUM[2], le site lavallois du LIUM, mais s'est déroulé en télétravail.

Le LIUM divise ses recherches en deux parties, correspondant aux deux équipes de chercheurs qui y travaillent : l'équipe LST (Langage and Speech Technology) et l'équipe IEIAH (Ingénierie des Environnements Informatiques pour l'Apprentissage Humain). Mon sujet de stage entrait dans le cadre du projet de jeu sérieux *Escape it!* mené par M. Pierre Laforcade, membre de l'équipe IEIAH.

Cette application sur appareils mobiles (iOS) a été imaginée pour essayer de répondre à un besoin thérapeutique, qui est de faciliter l'apprentissage de compétences visuelles élémentaires pour les enfants atteints de TSA, et permettre un suivi de la progression d'un enfant dans son apprentissage. Ce suivi a justement fait l'objet d'un besoin de sauvegarder les traces de l'enfant, ses interactions avec le jeu, et de permettre au parent / superviseur de visualiser ces traces, via des écrans qui les récapitulent.

Pour ce faire, j'ai suivi le schéma suivant, que j'ai appliqué à deux reprises, une première fois pour tracer les sessions, et une seconde fois pour tracer les parties en elles-mêmes, pour le Replayer :

- La définition des traces
- La récupération des traces
- La persistance des traces
- L'exploitation et la visualisation des traces

Dans la suite de ce rapport, vous trouverez en section 3 une présentation du contexte de stage et du laboratoire. La section 4 est dédiée au contexte du sujet de stage. La section 5 aborde les travaux que j'ai pu réaliser pendant ce stage. La section 6 comporte une conclusion de ce rapport, ainsi qu'un bilan personnel de ce stage. Le glossaire se situe à la section 7, et les bibliographies et webographies à la section 8.

IUT Laval
Le Mans Université

LIUM
Laboratoire d'Informatique
Université du Mans

Présentation du contexte de recherche

1. Le Laboratoire LIUM

Le Laboratoire d'Information de l'Université du Mans (LIUM) a été créé il y a environ 30 ans. La majorité des enseignants-chercheurs en informatique de l'Université du Mans y travaillent, et sont répartis sur deux sites :

- Le site du Mans et l'Institut Informatique Claude Chappe (IC2)
- Le site de Laval dans les bâtiments de l'IUT, et le CERIUM[2]

Les travaux du LIUM portent sur deux thématiques que sont l'Ingénierie des Environnements Informatiques pour l'Apprentissage Humain (IEIAH) et le Traitement Automatique de la Langue (LST), chacun des deux thèmes ayant une équipe de chercheurs qui lui est associé.

Le LIUM est composée de 60 personnes, dont 28 enseignants-chercheurs, 26 doctorants et post-doctorants, et 4 personnels administratifs et techniques.

Les chercheurs et enseignants-chercheurs du LIUM collaborent activement avec d'autres laboratoires, français et internationaux. Le LIUM s'est également associé à des partenaires industriels, comme Airbus, Orange Lab, Voxolab ou encore OpenClassrooms.

Le LIUM travaille activement sur un total de 20 projets actuellement : 12 pour l'équipe IST et 7 pour l'équipe IEIAH, les deux équipes travaillant ensemble sur le projet *PASTEL*.

Toutes ces informations proviennent et sont consultables sur le site de l'Université du Mans, dans la rubrique dédiée au LIUM.

2. L'équipe IEIAH

L'équipe IEIAH dirigée par Sébastien George axe ses recherches sur l'élaboration d'un socle scientifique pour le développement de systèmes informatiques pour l'enseignement et l'apprentissage. Les travaux de l'équipe IEIAH collabore activement avec des enseignants et formateurs dans la phase de conception d'un EIAH, et permettre d'avoir des logiciels les plus efficaces possibles, grâce à la consultation des concernés. La conception d'un EIAH se fait par un processus itératif passant par trois étapes qui sont l'adaptation en temps réel des activités d'enseignement et d'apprentissage, puis l'analyse des usages, et la réingénierie de l'EIAH.

L'équipe IEIAH est actuellement composée de 15 membres permanents, et 14 membres temporaires. L'équipe s'occupe actuellement de 8 projets, qui sont : *VEA*, *Ludifik'action*, *écri+*, *Escape it!*, *PASTEL*, *TGRIS*, *TurtleTablet*, *ReVeRIES*, le tout consultable sur le site du LIUM, dans la rubrique dédiée à l'équipe IEIAH.

Contexte du sujet de stage

1. Le projet *Escape it!*

Le stage tient place dans le cadre du projet *Escape it!*. Ce projet de développement de jeu sérieux sur appareils mobiles a été lancé en 2017 par les membres de l'équipe IEIAH du LIUM, en collaboration avec des membres de l'association Cocci'Bleue (spécialistes et parents), une association non lucrative qui s'implique dans l'accompagnement des personnes atteintes de TSA (Troubles du Spectre de l'Autisme), et la sensibilisation à l'autisme. Le projet n'a pas de financement propre (hormis les stages financés à 50% par l'Université du Mans et à 50% par l'Université Bretagne-Loire), et est né des récents plans autismes, recommandations de la Haute Autorité de Santé (HAS) et nouvelles mesures prises par d'autres pays, qui mettaient en avant la notion d'apprentissage pour favoriser l'inclusion scolaire, en apprenant à l'enfant les compétences élémentaires requises pour cette inclusion. L'objectif du projet Escape it! vise à supporter l'apprentissage des compétences visuelles (association d'objets, tri, etc) en proposant un jeu sérieux qui peut être proposé par un thérapeute, et poursuivi sous la supervision des parents à la maison.

2. Le jeu sérieux *Escape it!*

Selon Wikipédia, un jeu sérieux est « une activité qui combine une intention « sérieuse » – de type pédagogique, informative, communicationnelle, marketing, idéologique ou d'entraînement – avec des ressorts ludiques. » *Escape it!* rentre donc dans cette catégorie, car ce jeu a été conçu avec des objectifs thérapeutiques en tête.

Le jeu est prévu pour appareils mobiles (sous iOS à l'heure où j'écris ce rapport), et présente au joueur un scénario (une suite de niveaux), allant de 3 à 5 niveaux successifs, chaque niveau représentant une pièce d'une maison en 2D isométrique (parmi ces pièces on retrouve des chambres, une cuisine, une salle de bain, un salon et un garage), avec des graphismes cartoon.

Le but du jeu est d'interagir et déplacer des objets du niveau (en *drag and drop*) de façon à résoudre une sorte de puzzle un peu à la manière d'une *escape room* qui permettra d'ouvrir la porte de la pièce en question, qui marque la fin d'un niveau. Chacun de ces puzzles est basé sur 7 compétences visuelles empruntées au guide ABLLS-R et qui ont été réadaptées pour le jeu, et dont la difficulté augmente en fonction des succès de l'enfant et qui permettent de définir un profil particulier pour ce dernier.

Ces 7 compétences dont il est question sont les suivantes :

- B3 : Apparier des objets à des objets identiques
- B4 : Apparier des objets à des images
- B8 : Classer des items non identiques
- B9 : Placer des pièces sur un modèle
- B13 : Reproduire une séquence visuelle en présence du modèle
- B19 : Trier par catégorie
- B25 : Sériation

La création d'un scénario et la génération des niveaux qui le composent est gérée par un serveur distant, qui récupère les données du profil de l'enfant et crée des niveaux adaptés à ce profil.

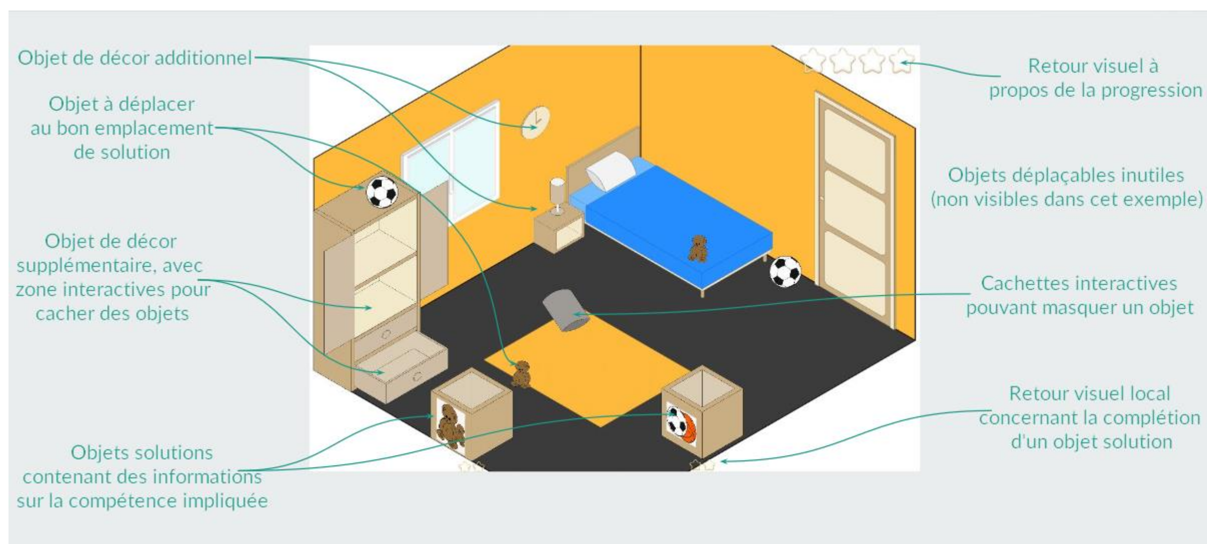

Figure 1 - Description d'une scène (CSEDU'18 - Escape it!)

3. De nouveaux besoins

Dans l'état actuel du projet, il restait encore à mon arrivée quelques besoins, auxquels mon sujet de stage devait répondre :

- Le besoin pour un parent superviseur de constater les sessions de jeu d'un enfant, ainsi que ses progrès, grâce à un système répertoriant les traces de l'enfant au quotidien pendant ses sessions (répertorier combien de sessions ont été joués au cours d'une journée, à quelle heure, etc.).

- Le besoin pour le développeur d'être notifié d'un niveau mal conçu et de recevoir celui-ci, pour pouvoir effectuer les modifications nécessaires et mettre à jour l'application.

Cependant, lors d'une de nos premières réunions, M. Laforcade m'a expliqué un besoin supplémentaire, qui ne figurait pas sur le sujet de stage original, besoin qui est le suivant :

- Le besoin pour un parent superviseur de pouvoir voir directement une session de jeu, répétant les actions de l'enfant lors de sa partie, via un système de *Replayer*.

4. Missions du stage

Pendant ce stage, j'ai eu pour mission de répondre à ces besoins, en implémentant à l'application les fonctionnalités nécessaires.

En ce qui concerne le besoin de supervision, il me fallait développer un moyen de tracer les sessions de jeu, ainsi que gérer leur stockage en XML et leur affichage via l'interface graphique du jeu.

Pour la partie replayer, M. Laforcade s'était chargé de concevoir une preuve de concept, sur laquelle je m'appuierais pour poursuivre mes travaux. Avant de travailler directement sur le replayer, il me fallait développer un moyen de stocker la composition des niveaux et de les regénérer depuis l'interface graphique que j'aurais déjà réalisée auparavant. Je devais aussi faire en sorte de récupérer et stocker les traces de l'enfant pendant sa partie (où a-t-il cliqué, quel objet a-t-il déplacé, etc.) dans l'objectif de les faire se réaliser en temps réel.

Concernant le besoin du développeur d'être notifié d'un niveau mal généré, il me fallait envoyer au serveur le fichier contenant un niveau possédant une erreur de conception rendant impossible sa complétion (un objet solution du puzzle qui se voit caché par un élément du décor par exemple), pour ensuite l'envoyer au développeur.

Préciser ce qu'est un niveau mal conçu

5. Organisation et méthodes

En raison des conditions particulières et des réglementations mises en place pendant la période du stage, celui-ci s'est fait en télétravail. M. Laforcade a pris l'initiative de créer

un serveur sur Discord dédié au stage, sur lequel nous pourrions échanger quotidiennement à ce sujet, et où je pourrais lui faire part de l'avancée de mes travaux. Nous avons beaucoup communiqué grâce au tchat vocal de Discord, où nous échangions sur la suite des événements et la façon de procéder, et j'ai très souvent envoyé des vidéos pour montrer directement les modifications apportées au projet.

En raison du télétravail, j'ai dû utiliser mon PC personnel pour travailler sur ce projet (CPU Intel Core i7-4790K, GPU NVidia GeForce GTX 970, 16Go de RAM en DDR3 à 1600MHz, SSD de 128Go et HDD de 1To, le tout sous Windows 10). Les logiciels utilisés durant ce stage sont Unity 2019.2.18f1 et Visual Studio Community pour tout ce qui est de l'ordre du développement, respectivement sous Unity (conception de prefabs, d'assets, etc.) et en C#, et Visual Studio Code, pour l'analyse et modification des fichiers XML.

Travaux réalisés

A l'origine, dans le sujet de stage ne figuraient que deux besoins majeurs. Un besoin de collecte / visualisation des traces, et un besoin de pouvoir avertir le développeur d'une mauvaise conception d'un niveau. Cependant, comme expliqué plus haut, un troisième besoin a fait son apparition, et s'est trouvé plus « prioritaire » que le besoin de notification, qui n'a, à l'heure de la rédaction de ce rapport, pas encore été traité.

1. Le système de traçage

Le système de traçage figurant dans le sujet du stage consistait à récupérer les données concernant les différentes parties de jeu d'un enfant, leur persistance et leur exploitation, via une interface graphique. La suite de cette section abordera chacun de ses points.

1.1. Définition et explication des traces

Tout d'abord, pour définir les traces qu'il fallait collecter, M. Laforcade et moi avons eu une première réunion, pour parler en détail de ce premier système de traces. M. Laforcade avait déjà prévu des maquettes des écrans finis, et de ce que devait permettre ce système de traçage.

Figure 2 - Maquette de l'écran des traces journalières

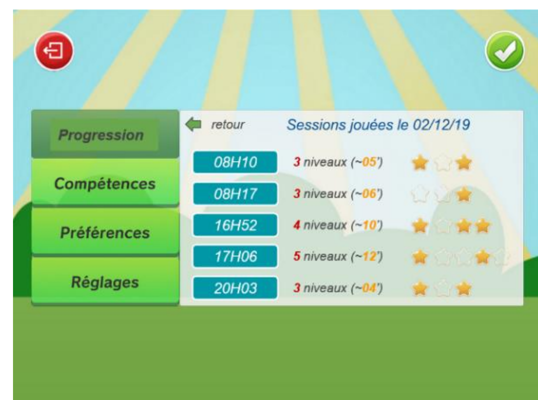

Figure 3 - Maquette de l'écran des traces des sessions

Figure 5 - Maquette de l'écran des traces des niveaux

Figure 4 - Maquette de l'écran des traces par compétences

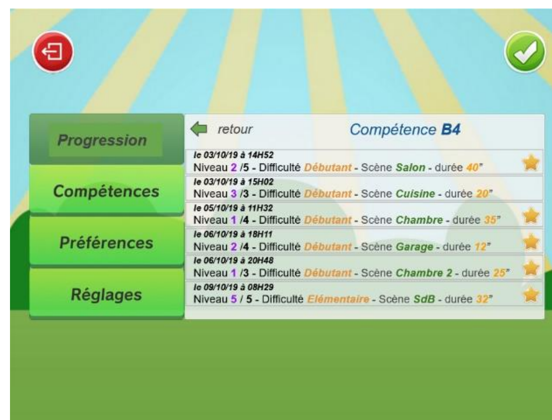

Figure 6 - Maquette de l'écran des traces des niveaux, en fonction d'une compétence

Je me suis basé sur chacune de ces figures (hormis la figure 5) pour déduire les données qu'il fallait tracer.

Sur la *figure 2* on peut voir qu'il faut tracer une ***date***, un nombre de ***sessions***, et une ***durée***.

Sur la *figure 3* il est question de tracer une ***heure***, une ***quantité de niveaux***, une ***durée***, et une ***quantité de succès, ou non*** (les étoiles indiquent si l'enfant a fini un niveau sans aide, ou non).

Sur la *figure 4*, on trouve les différentes informations d'un niveau, c'est-à-dire son ***ordre dans la série***, la ***compétence impliquée***, la ***difficulté***, la ***pièce dans laquelle s'est déroulé le niveau***, le ***temps mis à finir le niveau***, et le ***succès ou non de celui-ci***.

La *figure 6*, elle, regroupe des informations des trois énoncées plus haut, mais les organise de façon différente, en les triant en fonction de ***la compétence impliquée dans les niveaux***.

Ces maquettes m'ont permis de définir 4 types de traces :

- Les traces des journées, trace collectée à chaque première partie d'une journée.
- Les traces des sessions, trace collectée à chaque lancement d'une partie.
- Les traces des niveaux, trace collectée à chaque niveau d'une partie.
- Les traces par compétences, trace collectée à chaque niveau d'une partie également.

Les parties suivantes décrivent en détail le modèle objet de chacune de ces traces.

a) Les traces des journées

```
public class DayTrack
{
    private List<SessionTrack> m_Sessions;
    private string m_Date;
    private string m_DateToSave;

    1 référence
    public DayTrack()[...]

    1 référence
    public DayTrack(SDayTrack s)[...]

    1 référence
    public void AddSession(SessionTrack session)[...]

    4 références
    public List<SessionTrack> GetSessions()[...]

    1 référence
    public void Recap()[...]

    4 références
    public string GetDate()[...]

    4 références
    public string GetDateToSave()[...]

    3 références
    public int GetDuree()[...]

    0 références
    public string GetTime()[...]
}
```

Figure 7- Structure de la classe DayTrack

Les traces des journées ont été modélisées via l'objet *DayTrack*.

Celui-ci contient une liste de *SessionTracks* (les traces de sessions), ainsi que deux dates, qui sont en réalité la même mais avec un format différent (l'une est prévue pour l'affichage, l'autre pour la sauvegarde).

Les autres attributs évoqués plus haut (le nombre de session et la durée) sont des attributs déduits du contenu de la liste de sessions, et ne sont pas des attributs contenus à même la classe.

b) Les traces des sessions

```
public class SessionTrack
{
    private List<LevelTrack> m_Levels;
    private int m_NbLevels = 0;
    private string m_Heure, m_HeureToSave;

    1 référence
    public SessionTrack(int nbLevels)...

    1 référence
    public SessionTrack(SSessionTrack s)...

    4 références
    public int GetDuree()...

    0 références
    public string GetTime()...

    1 référence
    public void AddLevel(LevelTrack level)...

    2 références
    public List<LevelTrack> GetLevels()...

    1 référence
    public void Recap()...

    2 références
    public int GetNbLevels()...

    1 référence
    public void GetStatus()...

    3 références
    public string GetHeure()...

    4 références
    public string GetHeureToSave()...
```

Figure 8- Structure de la classe SessionTrack

Les traces des journées ont été modélisées via l'objet *SessionTrack*.

Celui-ci contient une liste de *LevelTracks* (les traces des niveaux), un entier déterminant la quantité de niveaux prévue, et permettant de vérifier que chaque niveau d'une a bien été ajouté, ainsi que deux heures, à formats différents pour les mêmes raisons que le *DayTrack*.

Les attributs évoqués lors de l'analyse des maquettes ne figurant pas ici (durée, quantité de succès) sont déduits de la liste de niveaux, et ne sont pas inhérents à cette classe.

c) Les traces des niveaux

```
public class LevelTrack
{
    private string m_Competence, m_Difficulte, m_Salle;
    private bool m_Succes = true;
    private int m_Duree, m_NumNiveau;

    1 référence
    public LevelTrack(string competence, string difficulte, string salle, int index)...

    2 références
    public LevelTrack(SLevelTrack s)...

    1 référence
    public void SetDuree(System.DateTime debut, System.DateTime fin)...

    5 références
    public int GetDuree()...

    2 références
    public int GetNumNiveau()...

    0 références
    public string GetTime()...

    3 références
    public string GetCompetence()...

    1 référence
    public string GetDifficulte()...

    1 référence
    public bool GetSucces()...

    1 référence
    public string GetSalle()...

    3 références
    public string Recap()...

    1 référence
    public void GotHelp()...

    0 références
    public void SetCompetence(string competence)...
}
```

Figure 9 - Structure de la classe LevelTrack

Les traces des niveaux ont été modélisées via l'objet *LevelTrack*.

Celui-ci contient toutes les données propres à un niveau (sa difficulté, la salle où le niveau s'est tenu, la compétence impliquée, ainsi que son ordre dans la session, e.g. s'il est le premier d'une session, ou le deuxième). L'objet contient également des informations relatives à la performance de l'enfant, grâce à la durée mise à finir le niveau, et aussi en affichant si l'enfant a demandé de l'aide pour finir le niveau.

d) Les traces par compétences

Les traces par compétences ont demandé un modèle légèrement différent de celui des autres traces. Deux objets ont été créés pour la création de celles-ci.

```
public class LevelTrackSkill
{
    public string m_Date;
    public string m_Heure;

    public int m_NumNiveau;
    public int m_TailleSession;

    public LevelTrack m_Level;

    1 référence
    public LevelTrackSkill(LevelTrack l){...}

    1 référence
    public LevelTrackSkill(SLevelTrackSkill l){...}
}
```

Figure 10 - Structure de la classe LevelTrackSkill

Un premier objet *LevelTrackSkill*, qui stocke les données d'un niveau, ainsi que les données relatives aux *SessionTracks* et *DayTracks* dans lesquels le niveau est situé, qui sont la date, l'heure, la taille de la session et l'emplacement du niveau dans cette même session.

```
public class SkillTrack
{
    public string m_SkillName;
    List<LevelTrackSkill> m_Levels;

    0 références
    public SkillTrack(string skillName){...}

    0 références
    public SkillTrack(SSkillTrack s){...}

    0 références
    public List<LevelTrackSkill> GetSkillTracks(){...}

    0 références
    public void AddSession(LevelTrackSkill l){...}
}
```

Figure 11 - Structure de la classe SkillTrack

Le second objet, *SkillTrack*, est celui qui stocke tous les *LevelTrackSkills* dont la compétence impliquée est la même que celle contenu dans cet objet (grâce à l'attribut *m_SkillName*)

1.2. Récupération des traces

Chacune des différentes traces est créée à différents moments d'une partie.

Les premières traces à se créer sont les *DayTracks* et *SessionTracks*, qui se créent au même instant.

Figure 12- Séquence d'écrans permettant de lancer une partie

Ces deux traces se créent au moment où le joueur arrive sur le 3e écran de la *figure 12* ci-dessus. L'objet *GameManager* (c'est un objet qui existe aussi longtemps que l'application est lancée, responsable de beaucoup de choses dans le jeu) contient une *DayTrack* statique, qui s'initialise dès le premier lancement d'une journée. Une *SessionTrack* se crée après, et initialise ses attributs de date, et de quantité de niveaux.

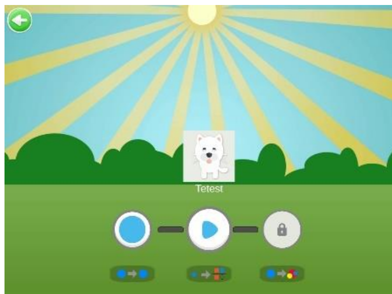

Figure 13 - Écran de lancement d'un niveau

Le *LevelTrack* est initialisé après que l'enfant ait cliqué sur le bouton *Play* de cet écran sur la *figure 13*, et ses attributs liés au niveau sont définis dès l'appui de ce bouton, qui lance l'initialisation des éléments du niveau. Un chronomètre se lance aussi après la pression du bouton, et servira à la définition du temps mis à compléter le niveau.

Si jamais l'enfant est aidé par un superviseur, alors celui-ci devra appuyer sur le bouton, pour indiquer qu'une aide a été fournie (l'enfant peut s'aider tout seul en appuyant plusieurs fois sur le bouton, pour déverrouiller une option permettant d'ouvrir la porte lui-même). Dès que l'enfant appuie sur ce bouton, le *LevelTrack* en prend note, et retiendra que l'enfant a reçu de l'aide.

Figure 14 - Cas où le bouton d'aide est pressé

Une fois les différents objets à leur place, la porte s'ouvre (voir *figure 15* ci-dessous). Lorsque l'enfant clique dessus, il passe au niveau suivant. Le chronomètre s'arrête aussi à cet instant précis, l'attribut de durée du *LevelTrack* en cours est défini, et le niveau est ajouté à la *SessionTrack* en cours.

Figure 15 - Cas de niveau terminé, avec porte ouverte

A ce moment-là se crée aussi un *LevelSkillTrack* relatif au niveau qui vient d'être complété, et qui récupère les données des *DayTrack* et *SessionTrack* courants. Le *LevelSkillTrack* va s'ajouter à la liste de niveau du *SkillTrack* correspondant à la même compétence. Si jamais c'est la première fois que la compétence est mise en œuvre dans un niveau, un nouveau *SkillTrack* est créé et associé à cette compétence, avant d'y ajouter le *LevelSkillTrack* en question par la suite.

1.3. Persistance des traces

Concernant la sauvegarde des traces, je me suis inspiré du travail déjà effectué pour la sauvegarde des profils, et ai repris le même modèle.

J'ai eu accès à l'outil *XmlSerializer*, qui permet de grandement faciliter la sérialisation et désérialisation d'objets, mais il nécessite de suivre un certain nombre de règles :

- Un attribut sérialisable doit avoir une portée *publique*
- Un tableau ou une liste d'attributs sérialisables doit être précédée de l'instruction *[XmlArray("Nom")]*
- Une classe sérialisable doit avoir un constructeur vide

Une fois ces règles décelées, j'ai commencé par créer un nouvel objet par trace existante, qui a pour but d'être sérialisé et stocké dans un fichier XML, et qui a les mêmes attributs que son équivoque modèle. Cet objet porte le même nom que l'objet modèle, mais se distingue avec un « S » au tout début (*DayTrack* -> *SDayTrack*, *LevelTrack* -> *SLevelTrack*, etc.). J'ai utilisé ce procédé pour deux raisons :

- La première, c'est que c'était déjà comme ceci que fonctionnait la sauvegarde des profils, donc ça m'a un peu facilité le travail et m'a donné directement une piste

(n'ayant jusqu'à lors que très peu travaillé avec la persistance des données, je ne savais pas vraiment au début comment m'y prendre).

- J'y ai constaté l'avantage de pouvoir me repérer plus facilement lors du développement, c'est-à-dire que je visualise directement ce que je manipule, un objet « courant », utilisé lors d'une partie, ou un objet « persistant », déjà sauvegardé ou destiné à l'être.

J'ai aussi créé deux objets, *DayTrackDB* et *SkillTrackDB*, qui regroupent toutes les traces, correspondant respectivement aux jours et aux compétences, et sont associés à un profil, grâce à un identifiant (un entier, égal à l'identifiant d'un profil). Ces objets sont instanciés une seule fois, et accessibles de partout dans le code.

En ce qui concerne la sauvegarde par jour, c'est à la fin de chaque session que la sauvegarde s'effectue (lorsque le joueur appuie sur la porte du dernier niveau). Les traces du jour courant sont mises à jour, puis ajoutées dans le *DayTrackDB* instancié. L'existence du jour dans les données sauvegardées est vérifiée, et en fonction de s'il existe ou non, le jour courant devient une nouvelle entrée, ou bien concatène celle déjà existante.

Pour le *SkillTrackDB*, c'est un peu différent. L'objet initial contient une liste de *SkillTracks* vide, et est mise à jour à chaque niveau terminé. Le *LevelTrack* du niveau est utilisé pour pouvoir récupérer la compétence en cours, et en fonction de si elle a déjà été utilisée ou non, créer une nouvelle *SkillTrack* adaptée à cette compétence dans le *SkillTrackDB* ou compléter celle déjà existante.

Pour chacun de ses deux objets, il y a un fichier approprié, contenant toutes les traces correspondantes, dans lequel les sauvegardes se font et écrasent les traces précédentes, et desquels on extrait les données à chaque lancement, pour pouvoir créer les deux instances de *DayTrackDB* et *SkillTrackDB* complètes, et comprenant toutes les traces conservées à ce jour.

```
2 références
public static void SaveAllSkillTracks(SkillTrackDB tracks)
{
    SkillTrackDB skillTrackDB = new SkillTrackDB(tracks);
    XmlSerializer serializer = new XmlSerializer(typeof(SkillTrackDB));
    FileStream stream = new FileStream(Application.persistentDataPath + "/XMLFiles/Tracks/child0" + Authentication.currentProfile.GetId() + "SkillTracks.xml", FileMode.Create);
    serializer.Serialize(stream, skillTrackDB);
    stream.Close();
}

2 références
public static void LoadAllSkillTracks()
{
    if (File.Exists(Application.persistentDataPath + "/XMLFiles/Tracks/child0" + Authentication.currentProfile.GetId() + "SkillTracks.xml"))
    {
        SkillTrackDB temp;
        XmlSerializer serializer = new XmlSerializer(typeof(SkillTrackDB));
        FileStream stream = new FileStream(Application.persistentDataPath + "/XMLFiles/Tracks/child0" + Authentication.currentProfile.GetId() + "SkillTracks.xml", FileMode.Open);
        temp = serializer.Deserialize(stream) as SkillTrackDB;
        stream.Close();
        m_SkillTrackDB = new SkillTrackDB(temp);
        m_SkillTrackDB.SetId();
    }
    else
    {
        SkillTrackDB skills = new SkillTrackDB();
        SaveAllSkillTracks(skills);
        LoadAllSkillTracks();
    }
```

Figure 16 - Exemple de fonctions de sauvegarde et de chargement (ici pour un SkillTrackDB)

Il existait déjà dans le code des fonctions de sérialisation et de désérialisation, grâce à l'outil *XmlSerializer*, qui permet de convertir en XML un objet (dont le type est indiqué

dans le constructeur) via la fonction *Serialize()*, son type, et un fichier, dans lequel écrire le XML. La désérialisation elle fonctionne de la même façon, seulement la fonction *Deserialize()* n'attend que le fichier à désérialiser, et en extrait un objet du type précisé dans le constructeur du *XmlSerializer*. Il suffit juste de modifier le type à gérer lors de la construction du *XmlSerializer* et de la déclaration / l'attribution des objets à sauvegarder / charger.

Chacun des deux objets *DB*, *DayTrackDB* et *SkillTrackDB*, sont sauvegardés dans deux fichiers qui leurs sont propres, et répertorient toutes les traces liées à un profil depuis sa création.

Figure 17 - Fichier de sauvegarde des DayTracks

Figure 18 - Fichier de sauvegarde des SkillTracks

1.4. Exploitation des traces

Une fois que toutes ces traces peuvent être sauvegardées et chargées, je me suis chargé de m'occuper de la résolution du besoin de visualisation, en fournissant une interface graphique affichant toutes ces traces.

Avant de commencer à créer les interfaces telles qu'elles, j'ai dû d'abord créer des *prefabs* de chaque élément représentant une trace, pour pouvoir les instancier depuis le code. Chacun de ces prefabs est une *view* de la trace qui lui correspond (une *DayView* affiche les informations d'une *DayTrack*, une *SessionView* celles d'une *SessionTrack*, etc.).

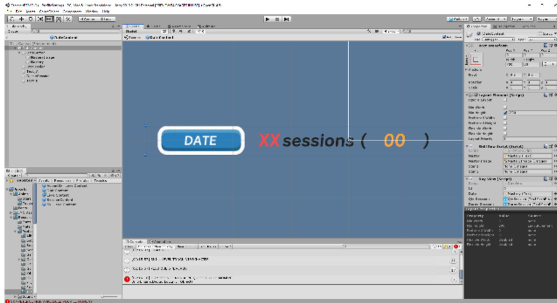

Figure 19- Prefab d'une DateView

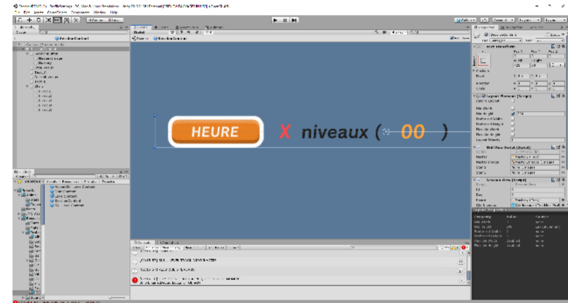

Figure 20 - Prefab d'une SessionView

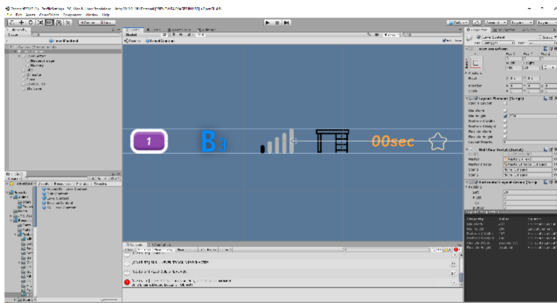

Figure 21 - Prefab d'une LevelView

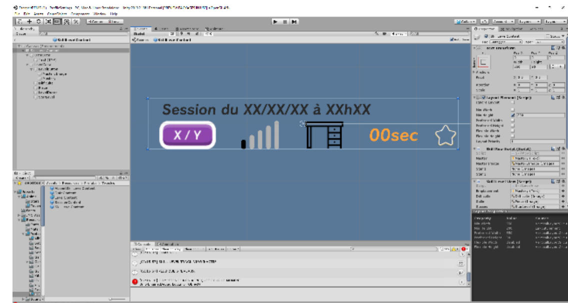

Figure 22 - Prefab d'une SkillLevelView

A noter que les prefabs des *LevelView* et *SkillLevelView* diffèrent des maquettes proposées dès le début du stage par M. Laforcade. Le fonctionnement du texte dans Unity a fait que le design proposé en maquettage fonctionnait, mais offrait un rendu visuellement disgracieux.

En effet, la taille dans laquelle tient un texte est fixe, et l'espace inoccupé reste inoccupé et ne s'adapte pas à la taille du texte (ce n'est pas responsive). Certains mots étant considérablement plus longs que d'autres (par exemple pour la difficulté, entre « avancé » et « intermédiaire »), cela donnait des *views* inconstantes avec beaucoup d'espace vide.

Nous avons donc discuté de ce problème avec M. Laforcade, et avons décidé d'opter pour des icônes, dont les tailles seraient identiques pour pallier à ce problème d'inconstance.

```
public class LevelView : MonoBehaviour
{
    public string m_Date, m_Heure;
    public SLevelTrack m_Level;

    public Text id;
    public Image m_Competence, m_Difficulte, m_Salle, m_Succes;
    public TextMeshProUGUI m_Duree;
}
```

Figure 23 - Contenu d'un script de View

Chaque prefab contient un script répertoriant tous les assets le constituant, qui nécessitent d'être modifiés, en fonction de ce qui est contenu dans la trace associée.

Le script montré sur la *figure 20* ci-contre montre comment se structure un script de *view* (ici une *LevelView*). Tous les éléments du prefab sont accessibles depuis ce script, ce qui

permet de générer une *view* adaptée aux traces correspondantes (afficher les bonnes icônes).

Chaque bouton sur les *views* permet de conserver une variable identifiant quelles traces on doit charger (si on clique sur un *DayTrack*, on obtient la journée, si on clique sur un *SessionTrack* on obtient l'heure, etc.), qui se conserve d'un écran un autre et permet de générer des bonnes *views*.

Les 5 prochaines figures exposeront les interfaces finales.

Figure 24 - Écran d'affichage des DayTracks

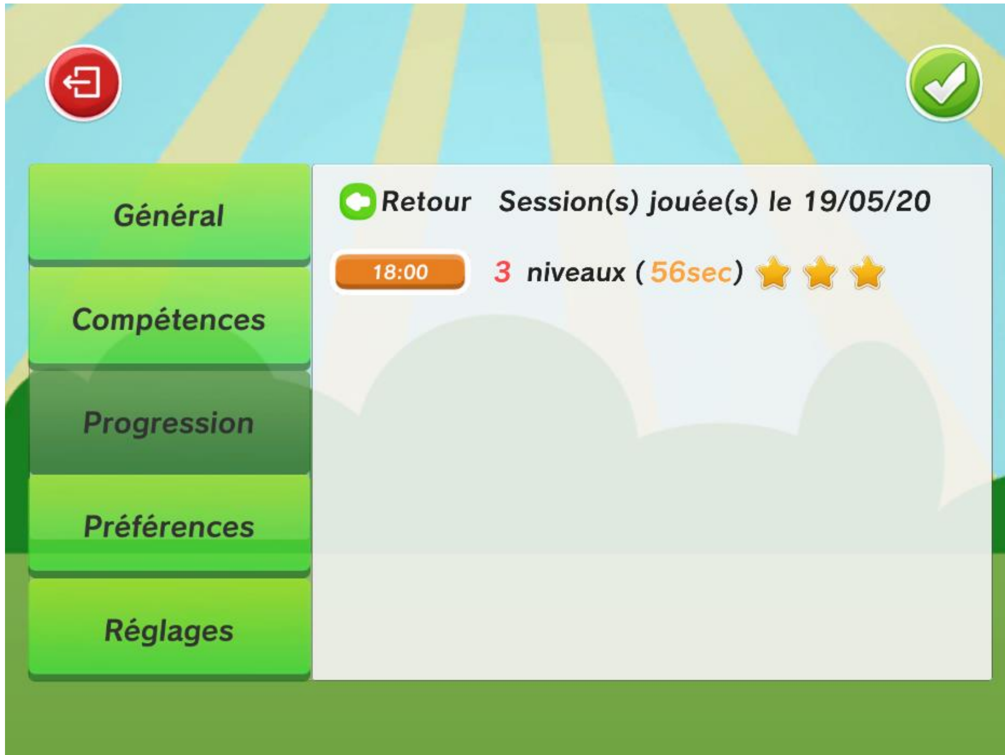

Figure 25 - Écran d'affichage des SessionsTracks

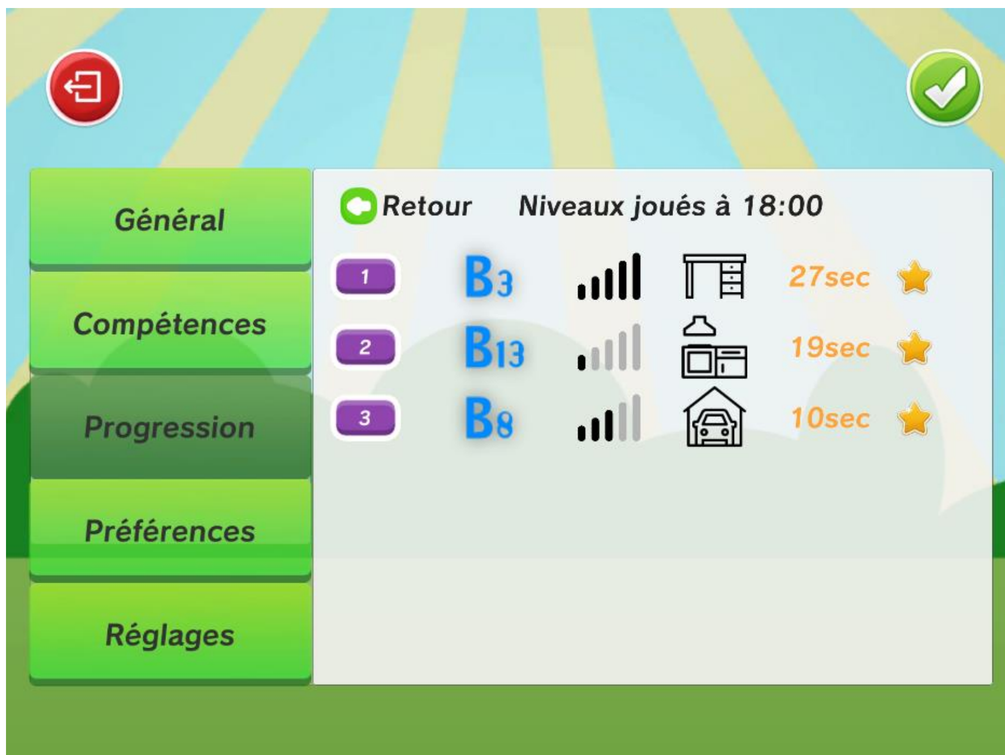

Figure 26 - Écran d'affichage des LevelTracks

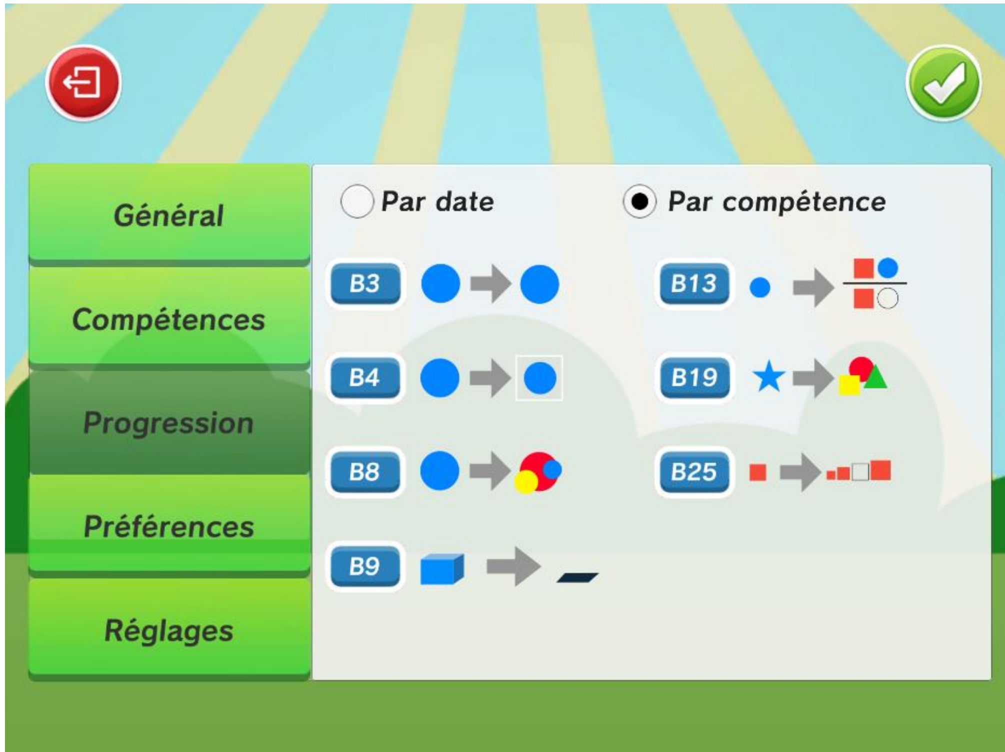

Figure 27 - Écran d'affichage des SkillTracks (NOTE : cet écran est statique)

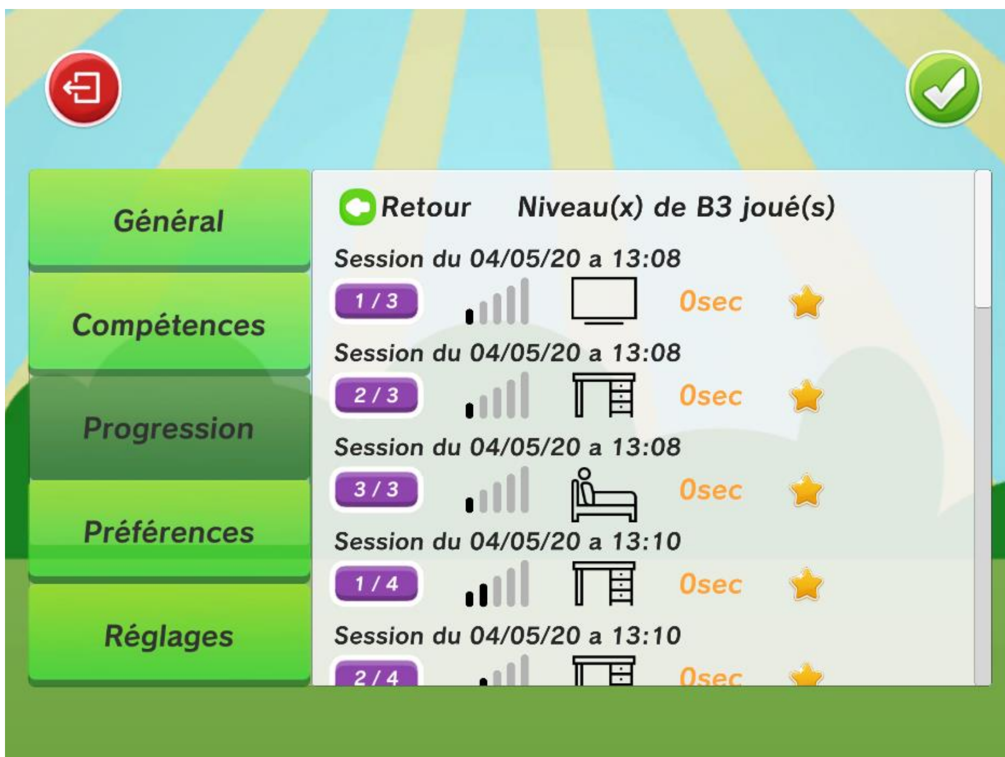

Figure 28 - Écran d'affichage des LevelSkillTracks

J'ai aussi été confronté au cas où le joueur cliquait sur une compétence qui n'a toujours pas été mise en œuvre sur l'écran des *SkillTracks*. Le jeu essayait de charger une *SkillTrack* qui n'existait pas, et cela provoquait une erreur (de type *NullReferenceException*). Pour résoudre ce problème, j'ai ajouté un prefab indiquant qu'aucune session n'a sollicité la compétence pour le moment.

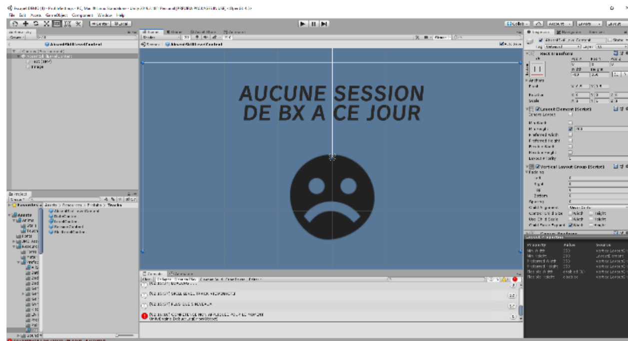

Figure 29 - Prefab d'une SkillTrack nulle

Figure 30 - Écran d'affichage d'une SkillTrack nulle

2. Le Replayer

Le replayer est une fonctionnalité qui doit permettre de rediffuser une partie de l'enfant, ses interactions avec un niveau, et ce depuis les écrans de traces que j'avais réalisé auparavant.

2.1. Complétion du fonctionnement

Lorsque j'ai commencé à travailler sur le Replayer, M. Laforcade avait déjà proposé un système fonctionnel. Ce modèle permettait de rejouer un niveau en mode replay juste après l'avoir terminé, et de reproduire un certain nombre d'actions stockées provisoirement. Le superviseur doit pouvoir interagir de différentes façons avec la rediffusion, et ces interactions sont possibles grâce à un panel. Voici comment se présente le menu

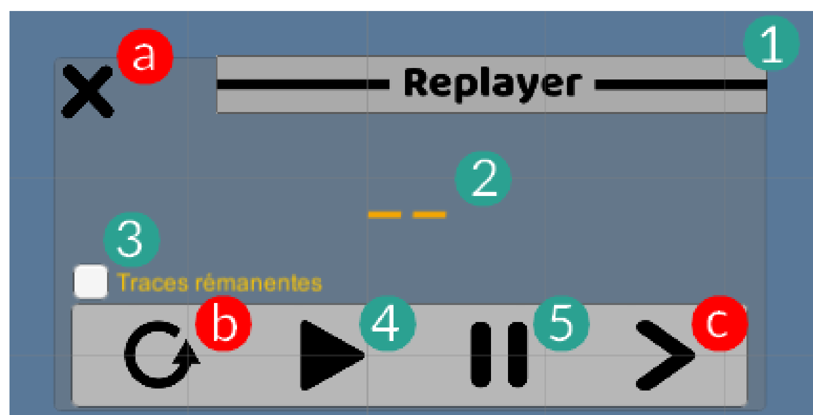

Figure 31- Panel du replayer

La *figure 31* répertorie toutes les parties interactives du replayer.

Les éléments annotés d'une bulle verte sont ceux qui fonctionnaient déjà, et ont été implémentés par M. Laforcade, qui sont les suivants :

1. Cette partie du replayer permet de déplacer le panel dans la scène.
2. Ce texte affiche le temps écoulé.
3. Ce bouton permet de laisser affichées les inputs de l'enfant.
4. Ce bouton permet de lancer le replay.
5. Ce bouton permet de mettre en pause le replay.

Les rouges sont celles sur lesquelles j'ai travaillé :

a. Pouvoir quitter le replayer (et revenir sur l'écran des traces des niveaux).
b. Rejouer la scène.
c. Avancer directement à la prochaine trace.

Avant d'entamer cette partie, je devais faire la jonction entre l'onglet de progression et le replayer, c'est-à-dire permettre de lancer un niveau déjà joué depuis l'écran de *LevelTracks*.

Pour ce faire, j'ai créé une fonction de sauvegarde me permettant de sauvegarder la constitution de chaque niveau dans un fichier unique lors de leur création, que je sauvegarde dès le lancement d'une session. J'ai ensuite fait en sorte de charger le niveau adapté en chargeant la disposition du niveau en question via les différentes informations transmises entre les écrans de l'onglet Progression (la date et l'heure). Une fois le bon fichier, je lance la génération du niveau.

Pour permettre de quitter le replayer et revenir directement à l'écran où l'utilisateur s'est arrêté, j'ai opté pour une approche asynchrone de la gestion de scènes. Au moment de lancer le niveau, là où une partie normale est la seule scène active, ici la partie en mode replay est la seule *visible*. La scène précédente, *ProfileSettings* (la scène d'affichage des traces) est toujours présente dans la hiérarchie des scènes, mais est simplement masquée. On constate sur la *figure 32* qu'il y a 3 scènes dans la hiérarchie. Ainsi, lorsque l'utilisateur presse le bouton pour quitter le replayer, la scène du replay se décharge et disparaît, tandis que celle des paramètres se réaffiche, dans l'état dans lequel elle a été laissée.

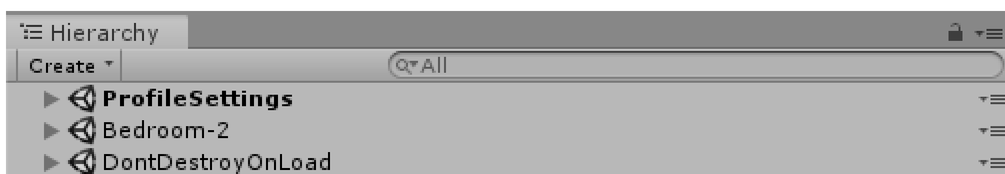

Figure 32 - Hiérarchie des scènes lors de la diffusion d'un replay

Concernant le bouton de redémarrage du replay, il s'est agit de simplement recharger la scène du replay, en se basant sur son nom. Toutefois, à noter que ce rechargement est en vérité feint, car recharger une scène ne peut pas se faire de façon asynchrone, et supprime la scène *ProfileSettings* de la hiérarchie, laissant seule la scène du replay (ce qui fait que la scène *ProfileSettings* n'est pas ré-affichable, et mène à une erreur). La scène du replay est déchargée de façon asynchrone, puis rechargée de façon asynchrone également juste après. La nuance peut paraître fine, mais a bien demandé une bonne heure avant d'être trouvée.

Concernant le bouton permettant d'avancer à la prochaine trace, il a suffi de donner au chronomètre la valeur de temps de la trace à l'index 0 de la liste de traces du replayer, qui est la prochaine à arriver chronologiquement, puis la supprimer juste après.

2.2. Définition et explication des traces

Dans le cas du replayer, les traces font ici allusion aux interactions possibles pendant une partie. Ces interactions héritent toutes d'une classe *AbstractTrack*, qui ne contient qu'un seul attribut, représentant à quel moment l'interaction a eu lieu, et plusieurs méthodes, dont une méthode Action(), permettant de simuler l'action du joueur lors du replay. *AbtractTrack* est une classe abstraite, dont les sous classes redéfinissent la méthode Action(). Il faut donc créer une sous classe par interaction à tracer, et redéfinir la méthode Action() à chaque fois.

Ces sous classes sont les suivantes :

- *TouchTrack*, qui correspond à un tap de l'enfant sur l'écran.
- *HiddenObjectTouchTrack*, qui correspond à un tap de l'enfant sur une cachette.
- *DoorTouchTrack* qui correspond à un tap de l'enfant sur la porte de sortie.
- *HelpTouchTrack* qui correspond à si l'enfant a reçu de l'aide ou non.
- *HintTouchTrack* qui correspond à si l'enfant a utilisé un indice.
- *OpenTouchTrack* qui correspond à si l'enfant a ouvert la porte (via le menu d'aide).
- *DragAndDropTrack* qui correspond à un drag and drop de l'enfant.

Les 4 traces *DoorTouchTrack*, *HelpTouchTrack*, *HintTouchTrack* et *OpenTouchTrack* sont des traces qui ne font que redéfinir la méthode Action() pour simuler les interactions avec les éléments dont elles dépendent (ouvrir la porte, activer le menu d'aide, etc.).

La *Touchtrack* est la forme la plus simple d'*AbstractTrack*. Elle ne contient en plus d'un temps deux coordonnées, x et y, indiquant sa position. L'action qu'elle déclence est l'apparition rapide d'un cercle aux coordonnées de la trace (voir *figure 33* ci-dessous).

Figure 33 - Action d'une TouchTrack

```
public class TouchTrack : AbstractTrack
{
    [XmlAttribute("X_Position")]
    public double x;
    [XmlAttribute("Y_Position")]
    public double y;

    [XmlAttribute("Element")]
    private GameObject pointer = null;

    0 références
    public TouchTrack() { }

    3 références
    public TouchTrack(Vector3 origin)[...]

    26 références
    override public string ToString()[...]

    10 références
    public override void action()
    {
        Debug.Log(this.ToString());

        if (pointer != null)
            GameObject.Destroy(pointer);

        GameObject parent = GameObject.Find("Managers");
        pointer = GameObject.Instantiate(Resources.Load("Prefabs/AllScenes/TouchPointer")) as GameObject;
        pointer.transform.SetParent(parent.transform);

        Camera cam = (GameObject.Find("Main Camera")).GetComponent("Camera") as Camera;
        Vector3 screenPos = cam.ScreenToWorldPoint(new Vector3((float)x, (float)y, cam.nearClipPlane));
        pointer.transform.position = screenPos;
    }
}
```

Figure 34 - Détail de la classe TouchTrack

La *HiddenObjectTouchTrack* elle, possède le nom du *GameObject* sur lequel l'enfant a appuyé. Son action consiste à retrouver le *GameObject* dans la scène grâce à son nom, et à activer la fonction permettant de révéler le contenu de la cachette. L'action affiche elle aussi un cercle à l'emplacement du *GameObject*, de la même façon que le *TouchTrack*.

```
public class HiddenObjectTouchTrack : AbstractTrack
{
    [XmlAttribute("Object")]
    public string name;

    [XmlAttribute("Element")]
    private GameObject pointer = null;

    0 références
    public HiddenObjectTouchTrack() { }

    1 référence
    public HiddenObjectTouchTrack(string name)
    {
        this.name = name;

        TimeSpan diff = System.DateTime.Now - TrackingServices.instance.GetZeroTime();
        float time = (float)diff.TotalMilliseconds / 1000;

        SetTimeInSeconds(time);
    }

    26 références
    override public string ToString()[...]

    10 références
    public override void action()
    {
        GameObject go = GameObject.Find(name);
        DragNDropData script = go.GetComponent<DragNDropData>();
        script.showHiddenElements();


        if (pointer != null)
            GameObject.Destroy(pointer);

        GameObject parent = GameObject.Find("Managers");
        pointer = GameObject.Instantiate(Resources.Load("Prefabs/AllScenes/TouchPointer")) as GameObject;
        pointer.transform.SetParent(parent.transform);

        Camera cam = (GameObject.Find("Main Camera")).GetComponent("Camera") as Camera;
        Vector3 screenPos = cam.ScreenToWorldPoint(new Vector3((float)go.transform.position.x, (float)go.transform.position.y, cam.nearClipPlane));
        pointer.transform.position = screenPos;
    }
}
```

Figure 35 - Détail de la classe HiddenObjectTouchTrack

IUT Laval
Le Mans Université

LIUM
Laboratoire d'Informatique
Université du Mans

Le *DragAndDropTrack* lui est plus compliqué. Il possède une liste de *VectorTime*, une classe qui contient une position et un attribut de temps. Ces positions dans le temps permettent de « tracer » la trajectoire du drag and drop. Le *DragAndDropTrack* possède aussi le nom de l'objet déplacé, et sa position initiale, au cas où le drag and drop ne se termine pas au bon endroit, et où l'objet retourne à sa position d'origine. La classe contient justement aussi un booléen indiquant si le drag and drop a été concluant ou non. Il y a aussi un booléen indiquant si l'objet disparaît dans l'objet solution, et une chaîne de caractère qui correspond au *layer* final (cela permet de faire en sorte que l'objet apparaisse sur le bon plan). L'action du *DrangAndDropTrack* va récupérer le *GameObject* par son nom, et le déplace à la position du premier *VectorTime* de la liste, qui est supprimé juste après. Si l'objet est déplacé jusqu'à un objet solution et que la liste de *VectorTime* est vide, alors son layer change, et s'il doit disparaître alors il devient inactif. Si l'objet n'est pas déplacé jusqu'à un objet solution, sa position est réinitialisée.

```
public class DragAndDropTrack : AbstractTrack
{
    [XmlArray("Positions")]
    public List<VectorTime> m_Positions;

    public Vector3 m_OriginalPosition;

    [XmlAttribute("Nom Objet")]
    public string m_GameObjectName;

    [XmlAttribute("LayerFinal")]
    public string m_FinalLayer = "";

    [XmlAttribute("DropValide")]
    public bool m_IsGood = false;

    [XmlAttribute("Disparait")]
    public bool m_Disappear = false;

    0 références
    public DragAndDropTrack() { }

    1 référence
    public DragAndDropTrack(Vector3 position, string name)...

    2 références
    public void IsGood()...

    1 référence
    public void AddVectorTime(Vector3 v)...

    26 références
    override public string ToString()...


    10 références
    public override void action()
    {
        GameObject gObj = GameObject.Find(m_GameObjectName);

        gObj.GetComponent<SpriteRenderer>().sortingLayerName = "draggable";

        gObj.transform.position = m_Positions[0].position;

        m_Positions.RemoveAt(0);

        if (m_IsGood)
        {
            gObj.GetComponent<SpriteRenderer>().sortingLayerName = m_FinalLayer;
        }

        if (m_IsGood && m_Positions.Count == 0 && m_Disappear)
            gObj.SetActive(false);


        if (!m_IsGood && m_Positions.Count == 0)
        {
            gObj.transform.position = m_OriginalPosition;
        }

    }
}
```

Figure 36 - Détail de la classe DragAndTouchTrack

```
public class VectorTime
{

    public Vector3 position;
    public float time;

    0 références
    public VectorTime() { }

    2 références
    public VectorTime(Vector3 v, float time)
    {
        this.position = v;
        this.time = time;
    }
}
```

Figure 37 - Détail de la classe VectorTime

Toutes ces traces sont instanciées dans la classe *TrackingServices*, un singleton qui contient la liste d'*AbstractTracks* qui ont été réalisés pendant toute la durée d'un niveau.

2.3. Récupération des traces

Pour la récupération des traces, on peut faire une distinction entre deux catégories. Une première regroupant toutes les interactions avec le niveau directement, et une avec les éléments d'interface.

La première regroupe les *TouchTrack*, *HiddenObjectTouchTrack* et *DragAndDropTrack*, qui sont des traces identifiées et créées grâce au script qui permet de gérer les différentes interactions que l'utilisateur peut effectuer avec le jeu. Une trace se créée dès lors qu'elle est détectée, et est complétée jusqu'à ce qu'elle soit estimée terminée par le script en ce qui concerne le drag and drop.

La seconde regroupe les quatre autres, *DoorTouchTrack*, *HelpTouchTrack*, *HintTouchTrack* et *OpenTouchTrack*, qui elles sont créées grâce aux boutons auxquels elles correspondent. A chaque fois qu'un de ces boutons est pressé, la trace associée est créée.

2.4. Persistance des traces

Concernant la persistance des traces, chaque niveau possède son propre fichier stockant les traces du joueur. Pour ce faire, j'ai utilisé le même procédé que précédemment et ait sauvegardé la liste d'*AbstractTracks* de l'instance de *TrackingServices*, cependant avec quelques contraintes supplémentaires.

Le fait d'utiliser des sous types d'une classe abstraite oblige à suivre d'autres règles, pour permettre la sérialisation. Il faut préciser au-dessus de chaque sous classe l'instruction *[XmlInclude(typeof(AbstractTrack)]* (voir *figure 38* ci-contre).

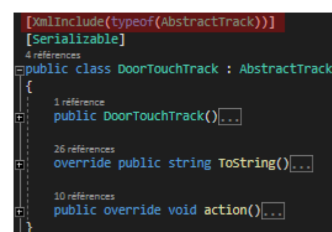

Figure 38 - Intégration des sous types en XML

Il faut aussi préciser dans un tableau de *Types* chaque sous classe d'*AbstractTrack* existante, et ajouter ce tableau au constructeur du *XmlSerializer* (voir *figure 39*).

Figure 39 - Définition d'un tableau de sous types pour la sérialisation

2.5. Rediffusion des traces

Plutôt que de vous décrire comment se déroule un replay en détail, textuellement et en décrivant chacune des actions, je vous propose une démonstration en vidéo de comment fonctionne le replay, qui sera, je l'espère, plus représentative du résultat obtenu.

En voici le lien : https://youtu.be/I242iK0GMtA

On constate sur la vidéo que les traces de la partie jouée sont stockées dès sa fin, et que son replay est accessible directement. Une fois dans celui-ci, on peut voir les différentes traces et interactions avec le niveau, qui s'enchaînent avec fluidité.

Cependant il est de bon aloi de remarquer le manque de certains feedbacks visuels et sonores au replay. En effet, en l'état, le replayer n'utilise pas encore complètement les contrôles du jeu pour fonctionner, et feint de jouer au jeu, au lieu d'y jouer réellement. Ce qui fait que chaque action est en réalité simulée de façon à correspondre visuellement à ce qu'elle doit faire, mais n'agit pas vraiment sur le jeu.

Pour illustrer ce propos, dans le replay du niveau dans la cuisine par exemple, lorsque la pomme glisse dans la coupe (l'objet solution), elle est masquée au moment où elle arrive à la dernière position, et le code du jeu ne reconnait pas que la pomme est dans la coupe, ce qui :

- Oblige à masquer manuellement la pomme.
- N'active pas les animations des étoiles et n'affiche pas la progression en temps réel.
- N'est pas systémique et flexible du tout. Chaque nouvelle action programmée nécessite d'être resimulée, plutôt que de juste se produire naturellement.

Durant la dernière semaine qui suit le rendu de ce rapport, j'envisage d'essayer de résoudre ce problème de système, et de faire en sorte que le niveau se joue réellement.

Conclusion

1. Travaux réalisés

La motivation derrière *Escape it!* est un objectif thérapeutique. L'application aspire à fournir une aide à l'apprentissage aux enfants autistes en leur permettant de faire travailler des compétences visuelles qui peuvent se montrer plus difficile à assimiler. Le support mobile et tactile permet de faciliter cet apprentissage, grâce à l'aspect jeu sérieux du projet, qui permet à l'enfant d'apprendre avec un modèle d'essais et d'erreurs.

Mon sujet de stage exprimait un besoin de pour suivre les progrès de l'enfant via un système de traçage. Pendant les 11 semaines de mon stage, bien que n'ayant pas pu travailler activement dans les tâches propres à la recherche, j'ai pu concevoir et développer deux systèmes de traçage différents :

- Les traces des sessions de jeu
- Le Replayer

Ces deux systèmes m'ont permis de réaliser des travaux touchant à une certaine variété de domaines, qui sont la compréhension et description d'un besoin, la récupération de données à instants clefs du jeu, la persistance de données et la réalisation d'interfaces sous Unity. Tout ceci a résulté d'un système de traçage permettant de suivre et constater les progrès d'un enfant, à différents niveaux de détail, allant de savoir à quels jours il a joué, jusqu'à reproduire ses parties en temps réel.

J'ai aussi pu réaliser des petits ajustements dans le projet, des corrections de bugs çà et là, et proposer des améliorations sur certains aspects du jeu.

Cependant le projet est encore améliorable. Tout d'abord par le biais du système de notification, que je n'ai du coup pas traité pendant le stage, au jour du rendu de ce rapport, qui permettrait d'avoir, au fil des améliorations, la génération de niveaux la plus pertinente et efficace possible, diminuant au maximum le risque de blocage d'un enfant, qui pourrait freiner son apprentissage.

Le jeu étant prototypé pour des thérapeutes, ils seront les principaux concernés quant à l'expression de nouveaux besoins pour le développement du jeu, et son évolution, en espérant le voir un jour dépasser le statut de prototype et devenir un outil thérapeutique utilisé par le grand public.

2. Bilan personnel

Cette section me permettra de faire un bilan de ce que m'a apporté ce stage personnellement, selon trois axes, que sont ma vision de l'informatique, les connaissances que j'ai pu acquérir, et les méthodes de travail que j'ai pu tirer de ce stage.

a. Vision de l'informatique

N'ayant jamais été très familier avec la recherche dans le monde informatique, ce stage a été l'occasion pour moi de comprendre quels sont les enjeux que celle-ci peut soulever, en l'occurrence de l'aspect thérapeutique d'un jeu sérieux, et des moyens mis en œuvre (collaboration avec des spécialistes, des associations) pour son développement. Malheureusement, en l'absence de présentiel, je n'ai pas pu échanger avec les différents chercheurs et doctorants du LIUM, ce qui aurait été l'occasion pour encore plus étoffer ma compréhension de la recherche.

Sachant également que je souhaite travailler dans l'industrie du jeu vidéo, ce stage m'a permis de me conforter davantage dans cette voie, car pendant ces 11 semaines, même si mon objectif était toujours la récupération de traces, j'ai pu effectuer des travaux très variés (persistance de données, interfaces, travail sur les scènes, etc.), alors que je n'ai travaillé que sur un seul aspect de l'application.

b. Connaissances acquises

J'ai beaucoup progressé dans la compréhension de comment fonctionne un système de persistance des données en XML dans un jeu, domaine dans lequel je ne m'étais jamais réellement plongé jusqu'à présent dans mes divers projets étudiants, ou personnels.

J'ai aussi enrichi ma maîtrise de Unity, notamment sur tout ce qui concerne la création d'interface, que je n'avais jusqu'alors qu'un petit peu effleuré dans les différents projets que j'ai pu mener sur ce moteur de jeu, mais également avec des petits fragments de connaissances et de fonctionnement du logiciel, çà et là, qui sont toujours bonnes à prendre et utiliser.

J'ai également acquis des compétences plus pragmatiques et passe partout, comme par exemple comment s'intégrer à un projet déjà entamé, comment s'approprier une grande quantité de code déjà existante et l'étendre, qui sont des compétences que je n'ai jamais vraiment eu besoin d'exploiter avant de réaliser ce stage, et qui s'avèrent cruciales.

c. Méthodes de travail

Les circonstances particulières pendant lesquelles s'est tenu le stage ont fait que malheureusement, ces 11 semaines de travail se sont faites en télétravail, et non en présentiel. Malgré tout, ce format de travail m'a tout de même obligé à m'imposer un

certain rythme, une certaine rigueur au quotidien pour faire mes heures de travail journalières. Cela dit, passé la demi seizaine de semaines, le télétravail a montré ses limites. Mon accommodation au confinement a fini par s'ébrécher par moments, et je me suis vu faire des entorses à mon rythme de sommeil, qui ont résulté de matinées plus difficiles que d'autres. Mais dans l'ensemble, je me suis fait force pour maintenir un rythme régulier, et ces écarts étaient malgré tout assez rares.

Grâce à Discord, M. Laforcade et moi avons pu communiquer quasi journalièrement, j'ai pu lui faire part de l'avancée de mes travaux, à chaque progression notable, et nous avons souvent pu discuter de comment mettre en place de nouvelles fonctionnalités, ce qui nous a souvent permis de confronter nos idées. Ces échanges m'ont rapidement permis de me rendre compte d'un certain manque d'esprit de test lorsque je développais (en raison de certains problèmes imprévus, soulevés par M. Laforcade qui testait nombre de cas que je n'anticipais pas), réflexion que j'ai tenté d'améliorer et d'appliquer avec sérieux à chaque nouvelle étape du développement.

Glossaire

Scène : Une scène est une composition de différents assets et objets du jeu.

Niveau : Scène jouable

Asset : Un asset dans Unity est n'importe quel objet pouvant être utilisé dans une scène (une image, un son, etc.).

GameObject : Un GameObject est une entité existant dans le cadre du jeu.

Prefab : Asset de Unity créé et enregistré, pouvant être réutilisé pour être instancié depuis un script.

Bibliographie & Webographie

Site du LIUM	https://lium.univ-lemans.fr/
Site d'*Escape it!*	https://projets-lium.univ-lemans.fr/escapeit/
Présentation CSEDU'18	https://projets-lium.univ-lemans.fr/escapeit/wp-content/uploads/sites/11/2018/04/CSEDU18-ESCAPE-IT.pdf
Article du Cénop sur les TSA	https://cenop.ca/troubles-comportement/tsa-trouble-spectre-autisme.php
Site de l'association Cocci'Bleue	http://www.coccibleue.fr/
Article Wikipédia sur les jeux sérieux	https://fr.wikipedia.org/wiki/Jeu_s%C3%A9rieux