


FACULTÉ DES SCIENCES DE MONTPELLIER

UNIVERSITÉ DE MONTPELLIER
M1 INFORMATIQUE - IMAGINE

Rapport de projet TER :
Développement d'un jeu point and click

HAI823I - PROJET INFORMATIQUE MASTER 1

GROUPE LUCIARINHU

Étudiants :
HUTTE Norman – 21908587
LUCIANI Nicolas – 22200159
MAURIN Christina – 21600781

Encadrante :
MME FARAJ Noura

Mai 2023

Remerciements

Au terme de ce travail, nous tenons à exprimer notre gratitude à notre professeure encadrante, Mme Faraj, pour avoir accepté d'encadrer notre projet ainsi que pour son suivi et ses conseils, qu'elle nous a prodigués tout au long de la période du projet. Nous sommes également reconnaissants envers l'équipe pédagogique de la Faculté des Sciences qui a assuré notre formation. Nous remercions enfin l'ensemble des membres du jury, qui nous font l'honneur de bien vouloir étudier avec attention notre travail.

Table des matières

1 Introduction

1.1 Présentation du sujet

Ce projet s'inscrit dans le cadre de l'unité d'enseignement "HAI823I : Projet Informatique TER" du second semestre du Master 1 IMAGINE et vise à développer un projet en lien avec la spécialité de nos études. Ainsi, nous avons choisi de nous orienter vers le jeu vidéo, et plus particulièrement, le genre "point and click".

Le point and click est un genre de jeu qui mêle aventure et réflexion. La prise en main est simple : on joue avec la souris. Le personnage évolue dans un décor dans lequel on vient interagir afin de progresser dans le scénario. On cherche d'abord des objets puis on les réutilise dans une autre partie de la scène, parfois en combinant ces objets entre eux. Il arrive que certaines actions n'aient aucun effet sur l'avancement dans le jeu et il faudra faire preuve de réflexion pour réussir à se débloquer.

1.2 Motivations du projet

Tout d'abord, il est à noter que nous avons choisi de proposer notre propre sujet. En effet, nous sommes adeptes de jeux d'énigmes, tels que *Rusty Lake*[1], *The Room*[2], *There is no game*[3], *Look INside*[4] et bien d'autres, mais également, l'un de nos membres, Nicolas, a été maître de jeu dans un escape game. Nous voulions ainsi faire à notre tour un jeu dans cette thématique. C'est pourquoi nous avons décidé de soumettre notre propre sujet à notre professeure Mme Faraj.

1. Par Rusty Lake : `https://www.rustylake.com/`
2. Par Fireproof Studios : `https://www.fireproofgames.com/`
3. Par Draw Me A Pixel : `https://drawmeapixel.com/`
4. Par Unexpected : `https://www.unexpected-studio.com/`

2 Analyse

Dans cette partie nous établirons une liste des éléments que nous voulons intégrer dans notre jeu avec un ordre de priorité. En effet, notre jeu n'est pas une reproduction d'un jeu existant mais une page vierge que l'on doit concevoir de A à Z. Il nous faut ainsi déterminer les objectifs à atteindre pour sa conception afin, d'une part, savoir se situer dans notre planning, et d'autre part, plus facilement se répartir le travail. Cette étape est divisée en deux partie.

2.1 Analyse des besoins fonctionnels

- **Structuration :** Réfléchir à la structure de notre jeu :
  - Réfléchir à la scène que l'on souhaite développer.
  - Quels éléments vont débloquer quels autres éléments.
  - Disposition des éléments dans la scène : il ne faut pas que le joueur ait tous les éléments côte à côté.
  - Implémentation de chaque "mini-jeu".
- **Aspect visuel des éléments :** Quelle direction artistique adopter.
- **Adaptation :** Faire en sorte que le code et les visuels soient compatibles.
- **Fin :** Indiquer au joueur qu'il a terminé le jeu.

2.2 Analyse des besoins non-fonctionnels

- **Rejouabilité :** Permettre au joueur de pouvoir jouer une nouvelle partie sans que le gameplay ne soit totalement identique.
- **Menu :** Avoir un menu au lancement du jeu permettant au joueur de sélectionner le bouton "jouer", "options" ou "quitter".
- **Écran pause :** Permettre au joueur de mettre en pause de jeu.
- **Sauvegarde :** Permettre au joueur de quitter le jeu en ayant une sauvegarde de sa progression.
- **Scénario :** Réfléchir à l'histoire que l'on souhaite partager au joueur.
- **Musique :** Ajout d'une musique de fond.

2.3 Lecture de documents dans le cadre du TER

Il nous a été demandé de sélectionner des documents pouvant nous être plus ou moins utiles durant la création de notre projet. Au moment où l'on nous a demandé de sélectionner des articles parmi une liste, nous ne savions pas exactement comment nous allions mettre en place nos éléments et certains ne correspondaient pas du tout à notre genre de jeu. C'est pourquoi nous avons préféré choisir des articles qui étaient plus "généraux".

✦ **Camera control in computer graphics** [5], *M. Christie, P. Olivier & J.M. Normand.*

L'article étudié est un document détaillant et faisant état de l'art des différentes techniques des différents contrôles caméra possibles en infographie. Ces derniers seront bien différents en fonction des domaines appliqués (modélisation, jeux-vidéo, cinéma, ...), présentant des conditions et par extension, des besoins, différents (pour exemple, dans le cas d'un jeu-vidéo, le rendu doit être en temps réel, exigeant des méthodes efficaces mais fondamentalement limitées).

La caméra est donc un élément primordial en 3D, et son contrôle reste un sujet d'étude même aujourd'hui. Différents types et méthodes de contrôles caméra sont évoqués : ‘

- Contrôle direct : l'utilisateur a un contrôle direct sur les propriétés de la caméra, il peut la déplacer, la tourner, zoomer, pivoter, afin d'obtenir le point de vue qu'il désire
- Contrôle "Through-the-Lens" : l'utilisateur a le contrôle sur le point de vue directement, les propriétés de la caméra s'adaptant à celui-ci. L'utilisateur voit alors "à travers la lentille".
- Contrôle assisté : la caméra se concentre sur un objet ou un environnement particulier, s'adaptant alors aux mouvements ou aux propriétés de celui-ci.
- Contrôle automatisé : la caméra a un mouvement prédéfini, des paramètres précis pour un instant t, donnant un effet de "cinématique".

L'article détaille une à plusieurs méthodes pour les différents types de contrôles listés ci-dessus, mais également des cas d'utilisation. Toutefois, bien qu'enrichissant, cet article ne nous a pas été directement utile dans le cas de ce projet. En effet, développant un jeu point and click, les contrôles caméra ne sont pas ordinaires, d'autant

5. (2008, December) In Computer Graphics Forum(Vol. 27, No. 8, pp. 2197-2218). Oxford, UK : Blackwell Publishing Ltd.

plus que nous avons privilégié l'utilisation de plusieurs scènes comportant chacune des caméras fixes, et le passage de l'une à l'autre.

Finalement, cet article peut s'avérer très intéressant dans le cadre d'un jeu hors point and click, puisqu'il couvre un ensemble assez large de genres de jeux (que ce soit des jeux à la première personne, à la troisième personne, ou bien pour des cinématiques). Les informations et connaissances apportées peuvent en revanche, très utile dans le cadre d'un autre projet de programmation 3D et moteur dans le cadre d'un module également suivi ce semestre.

✦ **A survey of interaction techniques for interactive 3D environments** [6], *J. Jankowski & M. Hachet.*

Cet article est un long document détaillé passant en revue différentes techniques pour les environnements 3D interactifs. Voici un rapide résumé des différentes techniques abordées :

— Techniques d'interaction directe et indirecte : directe entre l'utilisateur et l'objet 3D en utilisants des dispositifs tels que des souris, des stylos, suivi de mouvement. Indirecte comme par le biais de menus ou de curseurs permettant une interaction avec l'objet en question.

— Techniques de navigation : se déplacer dans un environnement 3D par la marche, la conduite ou le vol. Notamment avec des claviers ou des contrôleurs de jeu.

— Techniques de sélection et manipulation d'objets : comme son nom l'indique, permet à l'utilisateur de sélectionner et manipuler un objet. Cela inclut la sélection par rayon, encadrement ou par liste. On peut faire une translation, rotation ou un redimensionnement.

— Techniques de communication avec des personnages virtuels : reconnaissance vocale/faciale ou détection de gestes.

— Techniques de manipulation de la caméra : contrôle de la vue et de la caméra comme le panoramique, la rotation ou le zoom, par l'utilisation de claviers ou contrôleurs de jeux, par exemple.

Cet article énonce les différents avantages et inconvénients de chacun des points évoqués et par conséquent, nous a permis de faire le point sur ce que l'on voulait, ou pas, inclure dans notre jeu. Certains points nous semblaient évidents avant même la

6. (2013, May) In Eurographics 2013-STAR.

lecture de l'article. Par exemple, nous ne voulions pas d'interaction avec des personnages, car nous avions parti pris de ne pas en inclure. D'autre part, nous avions déjà travaillé sur l'aspect caméra du jeu, celui-ci se présentant en vue subjective, nous avons réduit l'angle de vue et le déplacement de la caméra dans la scène afin d'éviter l'effet de *motion sickness*[7].

Par ailleurs, cet article est intéressant dans sa globalité car il se destine à une large gamme de genre de jeux ou simulations 3D. Il se prête parfaitement à d'autres projets de programmation 3D.

✦ **Characterizing and measuring user experiences in digital games**[8], *W.A. IJsselsteijn, Y.A.W. de Kort, K Poels, A. Jurgelionis & F. Bellotti.*

Ce texte discute des défis liés à la mesure et à la caractérisation de l'expérience utilisateur dans les jeux vidéo. Les auteurs soulignent la difficulté de trouver des métriques adéquates en raison de la subjectivité inhérente à l'évaluation de l'expérience utilisateur. Chaque utilisateur a sa propre perception de ce qui est amusant, et cela ne constitue pas la seule façon de caractériser l'expérience d'un joueur.

La diversité entre les utilisateurs est une première raison expliquant la difficulté de définir des termes objectifs pour décrire une expérience utilisateur. Des facteurs tels que le sexe, la culture et l'âge ont un impact non négligeable sur les processus physiologiques et psychologiques qui contribuent à une expérience agréable. Les auteurs soulignent également qu'il est difficile d'évaluer une expérience en temps réel, car cela perturberait l'immersion de l'utilisateur. D'autre part, évaluer une expérience après coup nécessiterait un vocabulaire compréhensible par tous, ce qui n'est pas encore le cas, bien que des termes tels que "amusement", "engagement" ou "investissement" soient couramment utilisés par les joueurs et les critiques. La jeunesse de la discipline est également une raison expliquant la difficulté d'une caractérisation et d'une mesure adéquate de l'expérience utilisateur dans les jeux vidéo, bien qu'elle commence à être considérée comme une discipline digne d'études universitaires.

L'article insiste sur l'importance de développer ces mesures, car elles pourraient grandement aider les concepteurs de jeux.

Ensuite, l'article met en évidence la différence entre l'évaluation de l'expérience

7. Cinétose, provoque des maux de tête et nausées.

8. (2007) In R. Bernhaupt, & M. Tscheligi (Eds.), Proceedings of the International Conference on Advances in Computer Entertainment Technology (ACE 2007).

utilisateur dans les jeux vidéo et dans les applications productives. Alors qu'il est possible de mesurer cette expérience dans le cas des applications productives en se basant sur des facteurs principalement liés à l'utilisabilité de l'application, cela ne peut pas s'appliquer aux mesures dans le cadre des jeux vidéo, car l'utilisateur n'a pas d'objectif de productivité en soi. Les jeux vidéo comportent souvent des défis et des éléments contraignants, ce qui les distingue des applications productives.

Le texte présente deux caractéristiques qui émergent de la littérature sur le sujet : le "flow" et l'immersion. Le "flow" est un concept développé par Csikszentmihalyi [9] dans le cadre de ses études sur ce qui rend une expérience agréable pour un individu. Il a identifié plusieurs éléments caractéristiques du "flow", tels que des compétences requises, des objectifs clairement définis avec un feedback immédiat, la capacité de se concentrer rapidement, le contrôle des actions, l'investissement sans effort, la perte de conscience de soi et une distorsion de la perception du temps. Ces éléments sont importants dans l'expérience des joueurs et recherchés par les concepteurs de jeux.

L'immersion est définie comme l'investissement ou l'engagement d'un individu. Le texte présente trois types d'immersion basés sur des études sur les phases de jeu chez les enfants : l'immersion sensorielle, l'immersion basée sur le défi et l'immersion imaginative. Une étude menée sur des joueurs a défini trois niveaux d'immersion : l'engagement, l'implication et l'immersion totale.

Les auteurs concluent en présentant leur processus d'étude de l'expérience utilisateur dans les jeux vidéo, qui comprend l'utilisation d'un questionnaire couvrant un large éventail d'expériences de jeu, du jeu occasionnel à la compétition. Ce questionnaire permettrait une évaluation plus adéquate de l'expérience utilisateur.

Finalement, cet article s'est révélé pertinent dans le cadre d'un projet visant à concevoir un jeu qui ne soit pas frustrant pour les joueurs. Il est important de trouver un équilibre entre la complexité pour les joueurs expérimentés et l'accessibilité pour les débutants. Les récompenses doivent également être proportionnelles. L'évaluation de la rejouabilité du jeu en tenant compte des aspects discutés dans l'article pourrait permettre d'apporter des améliorations au système de jeu.

9. Psychologue hongrois

3 Gestion du projet

3.1 Organisation du travail

Lors des prémices du développement du jeu, nous avons réparti le travail de façon à ce que chacun puisse se concentrer sur un point en particulier semaine après semaine et ne pas s'éparpiller. De plus, ayant d'autres projets où nous étions pas forcément dans le même groupe, cela nous a permis de cloisonner les tâches.

Tout au long du semestre, nous avons pris des rendez-vous avec Mme Faraj toutes les deux semaines environ. Une période de temps qui nous permettait de développer suffisamment le projet en amont pour lui montrer une réelle avancée du projet et poser des questions pertinentes, si nous en avions.

3.2 Répartition du travail dans le temps

Nous avons décidé de nous planifier notre projet dans le temps par l'utilisation d'un diagramme de Gantt. Ce dernier permet de nous situer facilement et visuellement durant la période que dure le projet et estimer si l'on est en avance ou en retard par rapport à certaines tâches.

Nous avons réparti notre temps et notre travail en 4 principaux axes :

- **Mise en place du projet**
  Cette partie concerne le brainstorming du jeu. Nous avons notamment cherché des inspirations que nous avons validé ou réfuté, réfléchi à l'ambiance que nous souhaitions donner.
- **Conception de l'environnement**
  Nous avons implémenté une scène template[10] afin de faire nos premiers tests. Cela implique la gestion des interactions (utilisateur, scène, objet), inventaire, menu.
- **Réalisation du contenu**
  Ensuite nous avons constitué une vraie scène avec sa thématique propre et l'implémentation cohérente des énigmes afin d'un rendu final présentable et fonctionnel.
- **Gestion rapport/soutenance**
  Inclut la prise de notes tout au long du projet afin de retranscrire le plus fidèlement les avancées que nous avons faites dans la rédaction du rapport. Puis, en fin de projet, la préparation de la soutenance.

10. générique

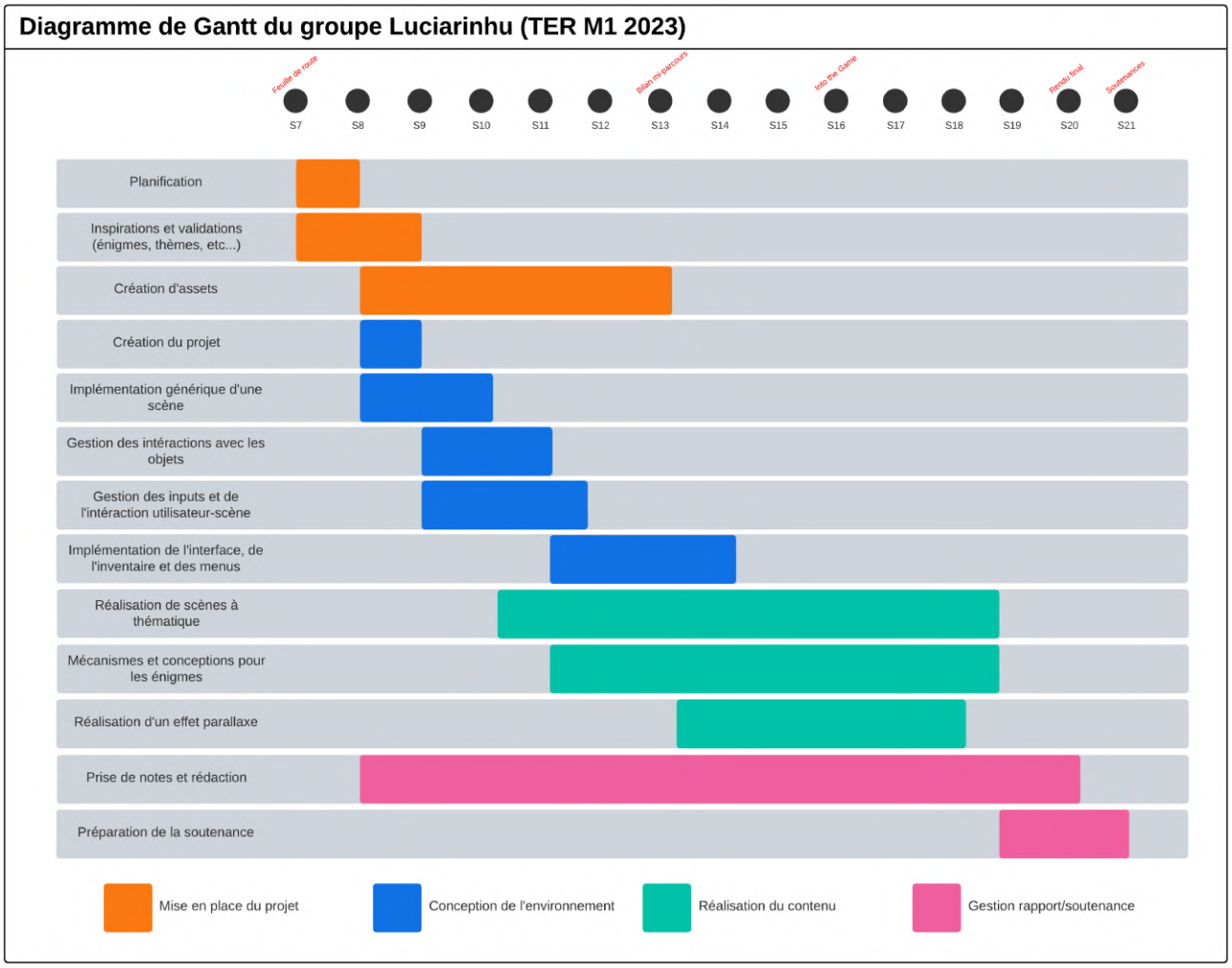

FIGURE 1 – Diagramme de Gantt

3.3 Outils de travail collaboratifs

Nous avons utilisé plusieurs programmes et logiciels pour mener à bien la création de notre projet.

Pour la création et la réalisation des différents assets [11] graphiques, nous avons choisi d'utiliser majoritairement le logiciel *Procreate* [12] sur iPad à l'aide d'un Apple Pencil. Ce choix est motivé par le fait que c'est un outil que possédait déjà Christina et sur lequel elle dessine depuis plusieurs années. De plus, l'iPad étant un objet très léger, il est possible de l'emporter partout et de pouvoir dessiner et/ou rectifier rapidement un élément si on le lui demande.

11. Ressource. Élément pouvant être de type graphique, audio, vidéo ou textuel.
12. Procreate : `https://procreate.com/`

Afin que chacun ait accès à ces éléments, nous avons choisi d'utiliser *Google Drive*[13]. C'était une évidence car, premièrement, il est possible d'enregistrer une version d'une image directement sur le gdrive via Procreate, et dans un second temps, d'avoir le gdrive qui s'actualise sur son bureau sans passer nécessairement par l'ouverture d'un navigateur web.

Pour assembler et coder notre jeu, nous avons dès le départ choisi le moteur de jeu *Unity*[14], Nicolas ayant déjà eu quelques petites expériences dessus. De plus, c'est un moteur gratuit, accessible et "beginner friendly" (on trouve facilement des tutoriels sur internet). Nous avons pu reproduire les scènes que nous avions imaginées sur Procreate, réaliser, coder les énigmes et définir nos interactions jeu/joueur (UI/UX). Il est à noter qu'utiliser Unity implique de coder en C#. Tout cela étant complètement nouveau pour Christina et Norman.

Également, afin que chacun puisse travailler sur une partie du jeu sans impacter le travail d'un autre, nous avons cherché un gestionnaire de versions qui puisse supporter ce projet conséquent. Nous avons donc utilisé *Plastic SCM*[15], qui a notamment été acquis par Unity en 2020.

Pour communiquer entre nous, nous avons utilisé un logiciel que l'on utilisait déjà au quotidien, *Discord*[16]. En effet, il permet d'échanger rapidement, que ce soit sur mobile ou ordinateur. Nous avons ainsi créé un serveur spécialement pour notre projet avec plusieurs salons textuels (planning, général, images, ...) et un salon vocal. Aussi, il nous a permis de discuter facilement avec notre encadrante lorsque nous avions une question ou lors de réunions en distanciel tout en ayant la possibilité de partager notre écran d'ordinateur.

Pour l'écriture du rapport, nous avons pensé qu'il serait plus judicieux de le rédiger à l'aide de *LaTeX*[17], la mise en page se faisant automatiquement. De plus, nous avons de bonnes bases car nous l'avons utilisé dans plusieurs autres matières. Enfin, voulant travailler en même temps sur des machines différentes, nous avons choisi de nous servir du site web *Overleaf*[18].

13. Google Drive : `https://drive.google.com/`
14. Unity : `https://unity.com/`
15. Plastic SCM : `https://plasticscm.com/`
16. Discord : `https://discord.com`
17. LaTeX : `https://www.latex-project.org/`
18. Overleaf : `https://fr.overleaf.com/`

4 Le jeu "Luciarinhu"

4.1 Les bases d'Unity

Pour la compréhension de tout lecteur, voici un récapitulatif des définitions des termes majoritairement utilisés sur le moteur de jeu Unity.

✦ **GameObject**
Un *GameObject* est une entité vide permettant de représenter un objet 2D ou 3D dans l'espace choisi. Il peut contenir plusieurs composants permettant de définir son apparence, ses fonctionnalités et son comportement.

Chaque élément visible du jeu est défini par un *GameObject*. Cela peut être, par exemple, un personnage, un meuble, un objet à ramasser, un mur, un bouton, et cætera...

Les *GameObjects* ont la particularité de pouvoir, si nécessaire, s'organiser en dépendance "parent-enfant", cela permet ainsi de créer des relations spatiales et structurelles entre ces éléments. Les manipulations appliquées au parent seront alors transmises à leurs enfants. Cela facilite la gestion des objets au besoin.

Ci-dessous, quelques composants employés fréquemment :

- **Transform**
  Ce composant est attaché à chaque élément et permet de contrôler sa position, sa rotation et sa mise à l'échelle en X, Y et Z.
- **Sprite Renderer**
  Il permet d'afficher un *Sprite* sur un *GameObject*. Un *Sprite* est une image 2D que l'on utilise pour représenter la surface visible du *GameObject* dans le jeu.
  (L'alter ego du *Sprite* en 3D est le *Mesh*.)
- **Collider**
  Ce composant permet de détecter les collisions entre les objets. Il permet aux objets de réagir entre eux lorsqu'ils rentrent en contact. Il permet également l'interaction utilisateur avec la souris. Il existe plusieurs types de *Colliders* permettant de s'adapter à la forme du *GameObject* : *Box Collider* (zone cubique ou rectangulaire), *Sphere Collider* et d'autres.
- **Script**
  Pour le côté programmation d'objet, Unity utilise le langage C#. Le script

permet de contrôler le comportement des objets ou des interactions dans le jeu.

- **Prefab**
  Un *Prefab* (préfabriqué) est un modèle préfabriqué d'un *GameObject* pouvant être utilisé afin de créer des instances multiples et identiques de celui-ci dans une *Scene*.
- **Scene**
  Une *Scene* est l'espace dans lequel on va disposer nos *GameObjects* pour les placer, les organiser et interagir. Elle peut être un niveau, un menu, ou une autre partie distincte du jeu. Unity permet de créer et gérer plusieurs scènes qui peuvent être chargées ou déchargées dynamiquement pendant l'exécution du jeu
- **UI Element**
  Comme son nom l'indique, il est utilisé pour créer et afficher des éléments d'interface utilisateur. On peut concevoir des interfaces graphiques pour les menus, les boutons, les panneaux, les curseurs, et cætera...

4.2 Création de l'univers du jeu

Lorsque nous avons formé notre groupe au début de l'année, nous n'avions pas encore une idée précise de ce que nous voulions produire mais l'envie commune de faire un jeu axé sur le *point and click*.

Généralement, un jeu *point and click* possède une histoire plus ou moins déployée. Afin de trouver l'inspiration pour nos énigmes, nous avons donc pensé qu'il était primordial d'imaginer un semblant de scénario.

Nous avons ainsi convenu que nous prendrions pour base un scénario où il n'y a qu'un personnage en vue subjective, que l'on joue. Ensuite, nous avons décidé de créer un univers basé sur la temporalité, où notre personnage, baptisée Lucia Rinhu, traverse les époques à chaque pièce parcourue. En effet, l'idée de base était de la faire se réveiller dans une pièce neutre et de lui faire connaître le "passé" avant de sortir vers l'extérieur.

Premièrement, nous avons constitué une première pièce se déroulant dans les années 20.

4.3 Les "Années 20" : version histoire

4.3.1 Conception

Tout d'abord, nous avons défini des éléments à débloquer et débloquant des accès vers d'autres éléments.

En premier lieu et par logique, nous avons décidé que la fin serait déclenchée lorsque l'on arrive à sortir de la pièce, après avoir trouvé la clé qui ouvre le cadenas bloquant la porte. Cette clé se cache dans le tiroir d'une table dont il faut trouver le code à quatre chiffres pour l'ouvrir. Ces quatre chiffres se trouvent à plusieurs coins de la pièce, sur des éléments ou des papiers, dissimulés de façon plus ou moins évidente.

Ci-dessous, un diagramme permettant de visualiser la construction de notre énigme générale.

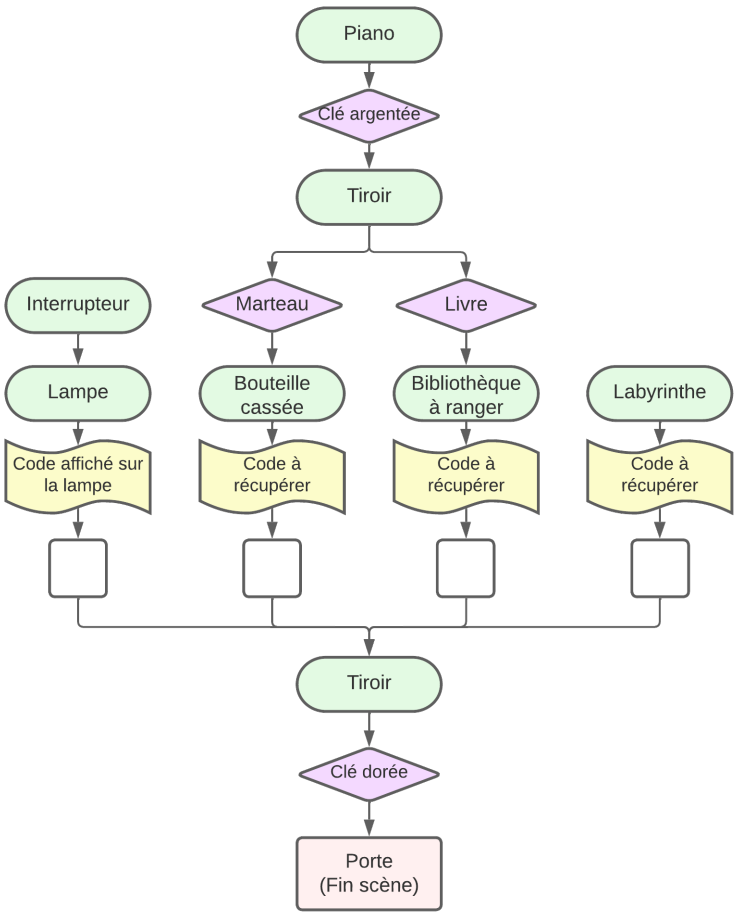

FIGURE 2 – Diagramme des dépendances

Pour cette pièce de type "salon des années 20" de notre jeu, nous avons décidé de représenter chaque pan de mur séparément. Afin de faciliter nos appellations, nous les avons chacune respectivement appelées : P1, P2, P3 et P4.

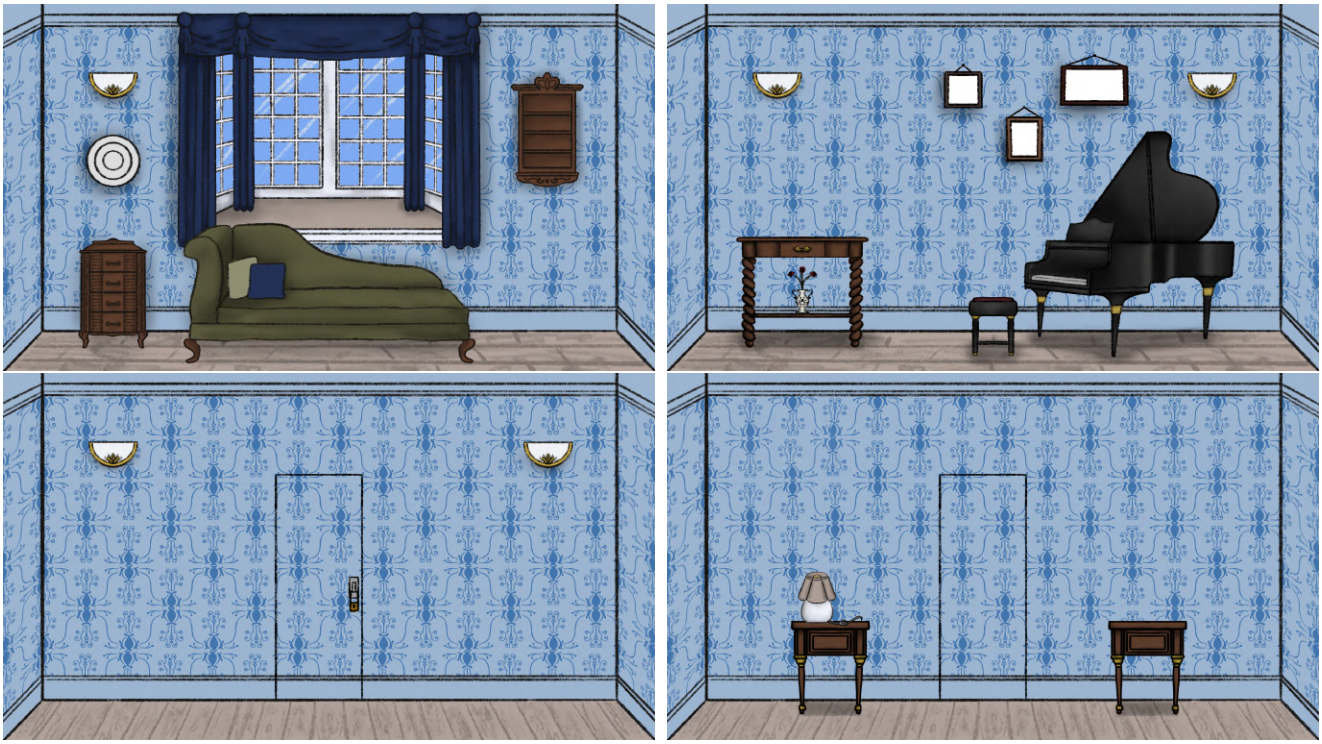

FIGURE 3 – Dessin des murs P1, P2, P3 et P4

P4 est la porte par laquelle on vient d'entrer et dont on ne peut plus retourner, c'est pourquoi il n'y a pas de serrure. P3 est la porte par laquelle nous voulons sortir. Dans notre pièce, les murs P3 et P4 sont à l'opposé l'un et l'autre. Au lancement du jeu, nous faisons face au mur P4, afin d'indiquer implicitement au joueur le but du jeu.

Après la finalisation des dessins de chacune des quatre parties de la pièce, tous réalisés par Christina, nous avons reconstruit notre dessin sur Unity. On est ainsi passé d'un dessin 2D en scène 3D avec ses éléments en 2D [19]. On constate que les éléments ont quelque peu changé de place.

19. Inspiré notamment par *Paper Mario* : `https://fr.wikipedia.org/wiki/Paper_Mario`

FIGURE 4 – Version 3D de P1

Ci-dessous, le *Prefab* que l'on utilise pour chaque mur. Nous avons placé une caméra au centre du sol et par le biais d'un script, nous avons autorisé le mouvement de la caméra par l'utilisateur avec une modification de l'angle de vue en X et Y limitée. Ce mouvement est déclenché lorsque l'on appuie sur le clic droit de la souris puis en bougeant celle-ci dans la direction choisie.

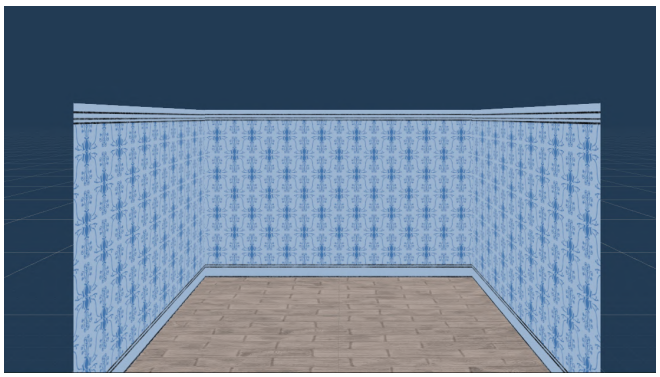

FIGURE 5 – Visuel sur Unity du *Prefab* de chaque pan de la pièce

Ensuite, nous avons créé des scènes "zoomées" pour chaque élément d'énigme. Par exemple, dans la partie de pièce de la figure 4, nous avons une énigme pour la "roue" blanche au mur, la bibliothèque murale et le chiffonnier. Ces scènes zoomées sont des scènes statiques [20] en 2D.

Dans la section suivante, nous allons parler des énigmes de ces objets.

20. Au sens : pas de mouvement de caméra.

4.3.2 Réalisation

➤ La roue "labyrinthe"

Dans cette énigme, le but est de déplacer les trois cercles afin de trouver l'unique solution qui permet d'ouvrir le cercle central, révélant un papier.

Pour cela, nous disposons de trois *GameObject* représentant les trois cercles, ainsi qu'un *GameObject* "manager", responsable de l'ensemble de l'énigme. En effet, les trois cercles disposent tous les trois d'un identifiant respectif (correspondant à leur position plus ou moins interne à la roue) et d'un même script responsable du mouvement de ces derniers par l'utilisateur : celui-ci peut saisir le cercle de son choix, et lui appliquer une rotation intuitivement avec la souris.

Nos cercles sont implémentés à l'origine avec leur position résolvant l'énigme (considérée comme par défaut avec une rotation de 0° en axe Z). Lors du lancement de la partie, une rotation aléatoire est appliquée à chacun des cercles de la roue, permettant un "mélange" du puzzle. L'utilisateur aura alors pour objectif de redonner à chacun des cercles sa position originelle, le script individuel au cercle vérifiant à chacune des rotations appliquées si celle-ci est égale à 0 (nous permettons une marge d'erreur : la retombée exacte sur 0 s'avérant quasiment impossible), et appellera une méthode du "manager" avec leur identifiant, afin d'attester de leur bonne résolution.

Une fois les trois cercles résolus et la bonne réception des trois appels respectifs de chacun, le "manager" validera l'énigme et permettra l'ouverture du centre. Celle-ci est réalisée par l'activation d'un *Sprite* représentant le centre ouvert, et par l'activation seulement à ce moment-là, du *GameObject* correspondant au papier récompense (cela empêche l'utilisateur d'interagir avec et de récupérer celui-ci avant la complétion de l'énigme).

FIGURE 6 – Roue générée aléatoirement

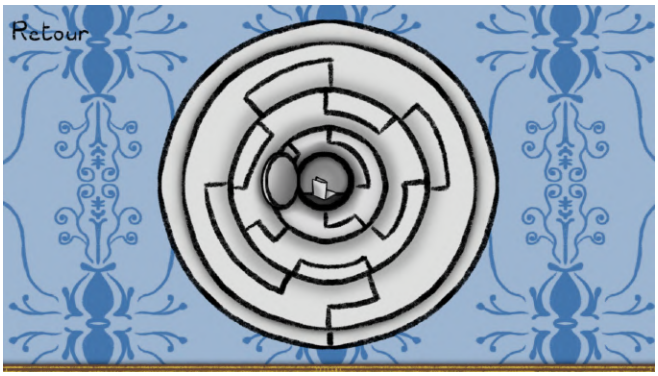

FIGURE 7 – Roue après résolution

➤ La lampe

Cette énigme est la plus simple à deviner pour le joueur et ne requiert aucun objet. Il suffit de cliquer sur la lampe et de mettre son interrupteur sur la position allumée. Un chiffre apparaît alors sur l'abat-jour.

Pour réaliser cela, nous disposons de deux *GameObject* : un pour la lampe, un pour le chiffre résultat de notre énigme. Nous associons le premier à un collider placé sur le bouton de la lampe ainsi qu'à cette dernière un script captant cette interaction utilisateur. Ainsi, à chaque clic sur le bouton, la lampe change d'état en fonction de celui actuel.

Pour représenter ces changements d'état, nous mettons à disposition du *GameObject* de la lampe, deux *Sprites* correspondant respectivement à l'état "éteint" et "allumé". Lors de l'interaction avec le bouton, le script va alors changer le *Sprite* associé au *SpriteRenderer* de la lampe pour y mettre le *Sprite* du nouvel état. De plus, dans le cas d'un passage à l'état "allumé", nous allons rendre visible le *GameObject* associé au chiffre indice, et le désactiver dans le cas inverse. Nous avons choisi d'implémenter notre énigme de cette façon afin de simplifier la randomisation : nous n'avons besoin de générer et de modifier que le *Sprite* du chiffre, sans altérer ceux de la lampe.

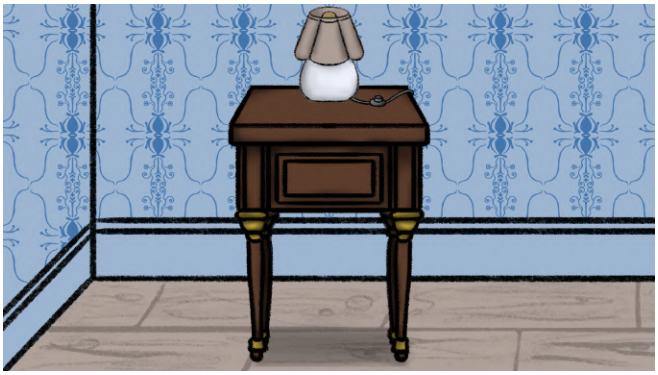

FIGURE 8 – Lampe éteinte

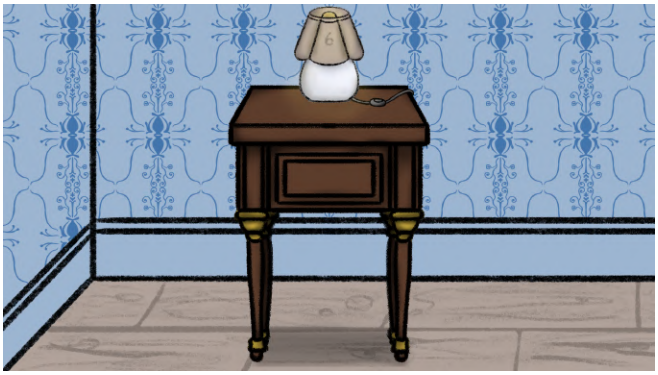

FIGURE 9 – Lampe allumée, code = 6

➤ Le piano

L'énigme du piano incite le joueur à être attentif. En effet, sans connaître la séquence à jouer, il y a peu de chance de réussir à débloquer l'objet caché. Cependant, en regardant les tableaux positionnés à différents endroits, la solution devient triviale.

Pour les non-musiciens, un tableau indique les correspondances entre les touches et les notes (système de notation américain [21]).

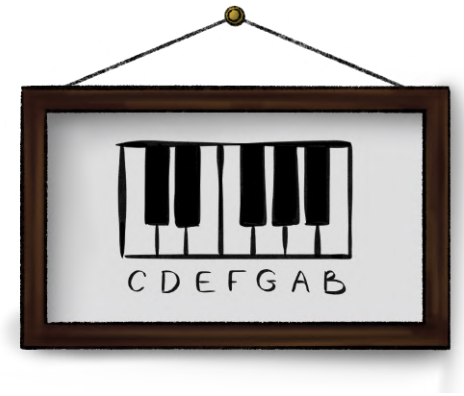

FIGURE 10 – Notes

FIGURE 11 – Mot à jouer

Un autre tableau indique ainsi le mot à jouer. Celui-ci a été randomisé parmi un dictionnaire de mots français que l'on a conçu avec 24 possibilités. Comme chaque lettre a été dessinée, nous avons donc pu modifier le mot dans notre code et calculer l'emplacement de chaque mot (de différentes tailles) pour qu'ils soient correctement positionnés dans le tableau.

Enfin, le dernier indice est un tableau interactif. Il faut cliquer dessus afin d'afficher une transition qui fait disparaître l'une des mains. La main restante correspond à l'endroit où jouer la séquence sur le piano. En effet, notre piano possède deux octaves jouables, il était donc nécessaire d'indiquer cet indice.

FIGURE 12 – Tableau avant de cliquer dessus

21. Do : C, Ré : D, Mi : E, Fa : F, Sol : G, La : A, Si : B.

Nous avons aussi codé les sons produits par le piano. Nous avions appris au semestre précédent, en traitement du signal, qu'il était possible de coder et produire une note de musique en utilisant les filtres fréquentiels [22].

FIGURE 13 – Piano une fois l'énigme résolue

Concernant l'énigme en elle-même, et à l'instar de celle de la roue, nous disposons de scripts individuels, et d'un script "manager" responsable du puzzle. Ces scripts individuels sont appliqués à chacune des touches de notre piano, et sont responsables du son évoqué juste avant qu'elles produisent, mais également de l'interaction avec l'utilisateur. Il pourra, lors d'un clic sur celles-ci, effectuer une pression sur la touche correspondante. De plus, chaque touche possède un identifiant unique en fonction de la note jouée et de son octave.

A chaque pression d'une touche, celle-ci va procéder à l'appel d'une méthode de notre "manager", accompagné de son identifiant. Cette méthode va alors, en fonction de l'identifiant passé en paramètre, reformer une chaîne de caractère. A chaque nouvelle entrée, nous vérifions que notre chaîne formée est bien préfixe de la séquence générée et à réaliser, si ce n'est pas le cas, nous repassons à une chaîne vide, sinon nous ajoutons simplement le caractère. Si la chaîne formée est égale à notre séquence, le "manager" valide l'énigme, et lance l'ouverture du tiroir.

Cette dernière est permise par une translation du *Sprite* du tiroir fermé : celui-ci se voit appliquer un mouvement vers la gauche jusqu'à dépasser un certain point, laissant accès à l'utilisateur la clé qu'il cache. Le *GameObject* correspondant à celle-ci n'est activé qu'une fois l'ouverture lancée, afin d'empêcher toute interaction entre l'utilisateur avant complétion de l'énigme.

22. HAI701I - Cours de M. Strauss

➤ Le chiffonnier

Le chiffonnier dispose d'une serrure. Tant que celle-ci n'est pas débloquée, on ne peut interagir avec les tiroirs du meuble. Après avoir débloqué et récupéré la clé, qui se trouve dans l'énigme du piano, il est possible de l'utiliser sur la serrure du chiffonnier. Cela permet alors d'ouvrir n'importe quel tiroir.

Pour ce faire, nous avons placé quatre *GameObjects* représentant chaque tiroir. Dans ceux-ci nous avons attribué deux *GameObjects* enfants : l'un avec le *Sprite* du tiroir fermé, l'autre avec le *Sprite* du tiroir ouvert. Pour le tiroir du haut, les enfants possèdent tous deux un enfant qui est le *Sprite* de la serrure.

Les deux enfants possèdent un *Collider2D*. En position fermée, lorsque le joueur fait un clic gauche sur un tiroir, le script associé va alors désactiver son *GameObject*. Puis, il active le deuxième *GameObject* enfant associé à la position ouverte. Et vice-versa.

Afin d'organiser et de simuler l'ouverture de chacun des tiroirs indépendamment, nous avons en sorte d'installer les tiroirs à une position différente sur l'axe Z sur notre scène (modification du *transform.position.z*). En effet, si l'on veut ouvrir un tiroir puis ouvrir celui du dessus, il faut que ce dernier masque le contenu du premier. De ce fait, plus un tiroir est haut, plus son Z est légèrement plus proche de celui de la caméra. (De bas en haut on aura : -0.17, -0.18, -0.19 et -0.2.)

FIGURE 14 – Chiffonnier fermé

FIGURE 15 – Chiffonnier ouvert sur les deux éléments à récupérer

➤ La bouteille à casser

Le principe de cette énigme est de briser la bouteille afin d'en récupérer le papier disséminé à l'intérieur.

Cette énigme a une particularité : elle requiert la complétion d'une autre énigme. En effet, un objet marteau peut être obtenu dans une précédente énigme, et nous permet de briser la bouteille. Pour cela, il suffit de sélectionner celui-ci dans l'inventaire, et ensuite cliquer sur la bouteille pour la briser.

En effet, la bouteille est représentée par un *GameObject* possédant un *Collider*. Lors de l'interaction avec l'objet dans l'inventaire place ce dernier dans une variable, on vérifie alors si la valeur de cette variable correspond à l'objet requis (ici le marteau). Si tel est le cas, alors un clic sur la bouteille entraînera le brisement de celle-ci, si non rien ne se passe.

FIGURE 16 – Bouteille avant puis après interaction

➤ La bibliothèque

Lorsque le joueur découvre la bibliothèque murale, il s'aperçoit (ou non) qu'il manque un élément dans celle-ci. En effet, sur l'étagère la plus haute, on peut remarquer qu'un emplacement vide se situe entre deux livres. Tant que ce vide n'aura été comblé, l'énigme ne sera pas résolue. On trouve ce livre dans le chiffonnier en figure 15.

Une fois le livre remis à sa place, on peut reconstituer la séquence valide. Si l'on regarde bien les tranches de chacun des livres, on constate qu'il y a plusieurs niveaux "jaune". Ordonner les livres ne suffit pas à résoudre l'énigme. En effet, les sculptures géométriques doivent elles aussi être triées. On les classe par nombre de côtés ce qui donne : cercle, triangle, carré et pentagone.

Au niveau du code, pour les deux séries, on réutilise le même *Script* "DragAndSwap" [23]. Dans le script, nous avons un tableau de *GameObject* et un de *Colliders*. On choisit le nombre d'objets puis on assigne les éléments correspondants à chaque tableau, dans l'ordre de résolution.

On a ensuite un *Script* "manager" qui va gérer le livre qui est enlevé, générer la séquence aléatoire des livres et valider lorsque les deux énigmes sont complétées.

Après résolution, la porte de l'étagère inférieure coulisse et donne accès à un papier.

FIGURE 17 – Bibliothèque non résolue et sans le livre manquant

FIGURE 18 – Bibliothèque résolue

➤ Le code à 4 chiffres

Après avoir obtenu les papiers et le chiffre sur la lampe, on peut désormais tenter de trouver le code permettant d'ouvrir le tiroir de la table (mur P2 [24]). En regardant au dessus de la porte cadenassée, on observe un tableau dont la disposition des éléments fait penser à celle de notre code.

23. Drag and swap : Glisser-Échanger.
24. Figure 3

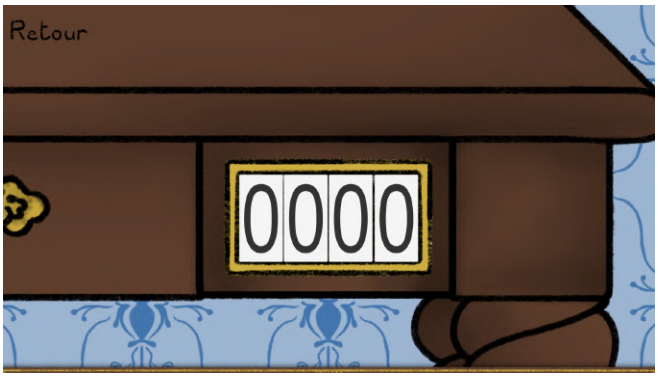

FIGURE 19 – Code à 4 chiffres bloquant l'ouverture du tiroir

FIGURE 20 – Indices position des chiffres

On retrouve ces symboles dans les tableaux éparpillés sur les murs de la pièce. Ces tableaux font références aux endroits où l'on a trouvé chaque chiffre. Également, on note que chaque papier possède un tampon rouge qui fait également allusion à ces emplacements. Par exemple, l'exemple ci-dessous montre un papier avec le chiffre "9", un tampon qui évoque un livre et sur l'un des murs, on a aussi un tableau avec un livre ouvert. On note que le symbole en bas à droite est un noeud papillon et qu'il correspond à la 3ème position du code.

FIGURE 21 – Exemple de liaison papier/tableau

Enfin, lorsque l'on a bien fait le rapprochement entre tous les tampons, les tableaux et les symboles, nous pouvons en déduire l'emplacement de chacun des chiffres du code. Cela dit, il est possible de trouver le code sans connaître tous les emplacements, ce n'est pas un pré-requis que l'on a scripté.

Le code est le validé lorsqu'il n'est plus possible de modifier les chiffres du code.

FIGURE 22 – Ouverture du tiroir de la table

➤ La porte finale

La porte finale représente le dernier obstacle auquel fera face le joueur avant de parvenir à la fin du jeu. Celle-ci consiste simplement en une porte requérant une clé spécifique pour permettre son ouverture. C'est la clé du tiroir à code.

La porte est représentée par un *GameObject* possédant un *Collider* au niveau de sa serrure. A l'instar du puzzle de la bouteille, un objet spécifique est requis afin de permettre l'ouverture : la clé dorée (évoquée précédemment). La sélection de celle-ci combinée avec l'interaction avec le cadenas, entraînera l'ouverture de la porte et par conséquent, la complétion du jeu.

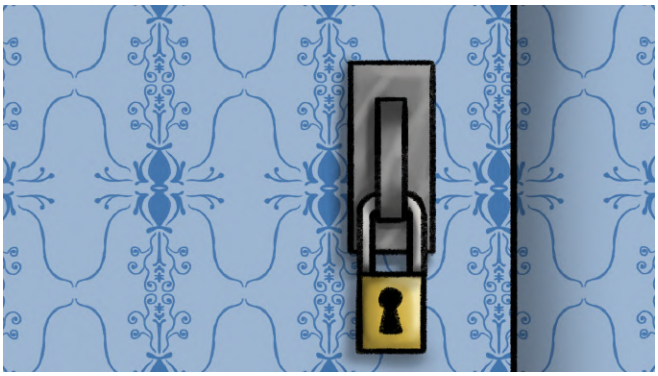

FIGURE 23 – Cadenas de la porte finale avant ouverture

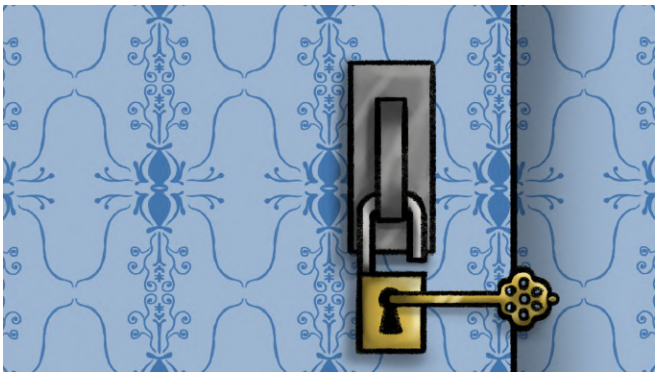

FIGURE 24 – Cadenas de la porte finale après ouverture

4.3.3 Gestion du jeu

➤ Le menu

Lors du lancement du jeu, nous tombons directement sur le menu principal du jeu. Celui-ci est la première scène du jeu, initialisant alors certains éléments requis pour le bon déroulement de notre jeu (tels que le manager d'énigmes, l'inventaire).

Le joueur se voit proposer plusieurs options :

— "Jouer" : une nouvelle partie se lance, envoyant le joueur directement dans la première scène.

— "Charger partie" : envoie le joueur sur le menu de charge de partie (celui-ci est traité plus tard lors de l'étude de la sauvegarde).

— "Options" : prévue pour envoyer le joueur sur un menu d'options lui permettant de changer certains paramètres (le volume sonore, choix de l'affichage fenêtré/plein écran, ...) mais non implémenté.

— "Quitter" : quitte simplement le jeu en fermant celui-ci.

Cela est permis par l'utilisation d'un *Canvas* regroupant plusieurs images cliquables qui serviront de boutons. Le Canvas se voit alors associer un script comportant une méthode spécifique pour chacun des boutons de celui-ci. Nous pouvons alors, par la suite, attribuer à chacun de ces boutons sa méthode respective.

FIGURE 25 – Menu du jeu

➤ Chargeur de scène

Pour passer d'une pièce à l'autre, et donc changer de scène, il faut les charger. C'est le chargeur de scène ou *Level Loader* qui réalise cela.

Le chargeur de scène utilise les *Build Settings*[25] et notamment les indices de scène dans ces paramètres pour charger les différentes scènes. Il dispose de différentes méthodes :

— Charger une scène à un indice donné
— Charger la scène précédente
— Charger la scène "à droite" ou "à gauche" (définies avec les flèches)

Il se charge également de lancer les animations de transitions entre ces scènes. Ces animations sont lancées par des *Coroutine*. Une *Coroutine* est une portion de code qui sera exécutée parallèlement au reste du programme. L'objectif d'utiliser une *Coroutine* ici est que l'animation de transition et le chargement de la scène ne se bloquent pas entre elles. Lancer une animation avec une *Coroutine* nous semblait plus approprié étant donné que réaliser plusieurs actions dans une *Coroutine* est moins pratique en terme de code. Ici le code exécuté en parallèle du chargement de la scène est uniquement le lancement de l'animation.

Cette animation se décompose en deux parties. Une première partie se fait en "fermeture" de la scène courante, et la seconde en "ouverture" de la scène suivante.

Le chargeur de niveau est utilisé par beaucoup d'éléments de nos scènes. Par exemple, tous les objets ayant un *BoxCollider* dont le clic amènera vers une scène "zoomée" sont du type *ObjectCollider* et l'utilisent dans leur méthode *OnMouseDown()*.

➤ L'inventaire

L'inventaire est un élément d'*User Interface (UI)* composé de plusieurs enfants. D'abord, l'arrière plan, défini comme une *UI - Image*. Ensuite, il y a 9 *UI - Boutons*. Ce sont ces boutons qui vont nous permettre d'afficher et d'utiliser les objets (voir Fig.26). L'inventaire possède une instance *statique* de façon à pouvoir y accéder depuis n'importe quelle scène et n'importe quelle classe.

25. Paramètres de Build du jeu. Contient une liste indexée des scènes du jeu

Nous avons défini un *Scriptable Object*[26] *Item*. L'avantage d'utiliser un *Scriptable Object* plutôt qu'une simple classe est d'abord qu'il est possible d'ajouter un élément au menu déroulant du moteur pour créer des éléments. Cela permet de gagner beaucoup de temps sur la création des différents objets. (voir Fig. 27)

Un objet est donc défini comme suit :

— Un identifiant

— Un nom Une image (l'icône à afficher dans l'inventaire)

On a également défini des *Paper Item* qui sont des dérivés d'*Item* auxquels nous avons ajouté le tampon et le chiffre à afficher.

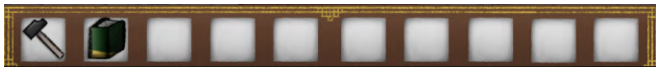

FIGURE 26 – Barre d'inventaire

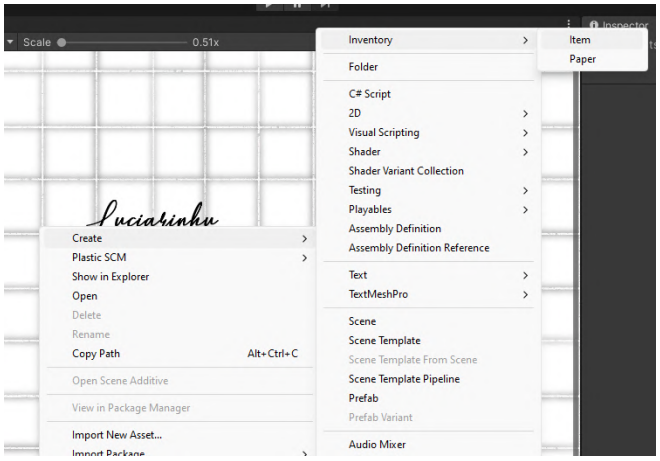

FIGURE 27 – Menu déroulant pour créer un objet

Au lancement du jeu, l'inventaire est "fermé". Pour l'ouvrir (et le refermer) il faut cliquer sur la bande supérieure avec les ornements dorés.

Une fois l'inventaire ouvert, on peut sélectionner un objet. Sélectionner un objet en cliquant dessus, permettra ensuite de l'utiliser dans la partie. Un objet sélectionné

26. Objet programmable et dont les propriétés sont modifiables dans l'*Inspecteur* du moteur

aura son emplacement encadré en doré.

➢ Utilisation d'un objet

Nous avons déjà évoqué comment le jeu gère les interactions utilisateur pour l'utilisation d'objet. C'est au niveau de l'inventaire et d'une autre classe que ces interactions sont intégrées au système de la partie. Cette autre classe est nommée *ItemReceiver*. Une instance de cette classe implémente donc la méthode pour gérer le clic sur le *Collider* de la zone nécessitant un objet. De plus, cette méthode va "utiliser" un objet du côté de l'inventaire. Voici, ci-dessous, la procédure d'utilisation d'un objet.

1. Si un objet est sélectionné et que l'utilisateur clique sur un *Collider* pour l'utiliser, on vérifie si l'identifiant attendu par l'*ItemReceiver* est égal à celui de l'objet.
2. Si ce n'est pas le cas, alors l'objet est désélectionné et rien ne se passe.
3. Sinon, la zone est déverrouillée (mise à jour des booléens nécessaires dans le manager d'énigme, lancement d'animation, ...)
4. L'objet est supprimé de l'inventaire
5. L'inventaire est réorganisé pour ne pas avoir d'emplacement vide entre deux objets.

➢ Utilisation d'un papier

Si l'utilisateur clique sur un papier, la vue change comme montré en Figure 21.

➤ Le manager d'énigmes

Afin de garder une trace de l'avancée globale des énigmes et de notre partie, nous avons implémenté un "manager d'énigmes" regroupe l'ensemble des données de la partie. C'est dans celui-ci que sont regroupées plusieurs informations utiles telles que :

— le statut de complétion des énigmes (des booléens représentant si le puzzle a été complété, utile pour la sauvegarde ou pour les énigmes requérant la complétion d'autres).

— le statut de récupération des objets comme les clés, papiers indices (des booléens représentant si l'objet a été récupéré ou non par le joueur, utile pour la sauvegarde ou pour éviter de générer à nouveau un objet déjà récupéré lors de l'initialisation d'une énigme).

— les séquences ou nombres associés à la randomisation (ce script est responsable de la randomisation des énigmes et des séquences ou nombres qui en découlent, cela permet une centralisation de ces données afin de faciliter la sauvegarde et le déroulement du jeu).

— un chronomètre permettant de calculer le temps en jeu (celui-ci ne se lance qu'à chaque lancement de la partie qu'elle soit nouvelle ou chargée, utile pour la sauvegarde).

Il est initialisé dès le lancement du jeu.

➤ La sauvegarde

Un système de sauvegarde de notre progression est aujourd'hui une composante essentielle dans le jeu-vidéo, et ce fut également un de nos objectifs. Celui-ci permet à l'utilisateur de sauvegarder sa partie et sa progression à tout moment, et de la reprendre quand bon lui semble.

Nous avons implémenté notre système de sauvegarde à l'aide de la classe PlayerPrefs de Unity. Celle-ci permet de stocker dans un registre local, différentes données non encryptées (nous devons alors veilleur à ne pas y rentrer des données sensitives) que ce soit des nombres entiers ou flottants, ou bien des chaînes de caractères. Ces données peuvent être stockées ou récupérées à l'aide d'une clé de référence que l'on choisit lors de la sauvegarde, avec la méthode correspondante. Il nous suffit de cette même clé pour récupérer la valeur stockée avec une autre méthode.

Pour la sauvegarde de partie, nous avons ajouté à notre menu de pause une option "Sauvegarder partie", réalisant ainsi une sauvegarde de l'ensemble des données de notre manager d'énigmes (traité précédemment) avec des clés que nous choisissons. Un message "Sauvegarde effectuée!" s'affiche pendant un cours instant pour notifier au joueur la réussite de la sauvegarde.

Pour la charge de partie, nous avons ajouté un menu de charge, accessible dans le menu principal du jeu. Celui-ci affiche différentes informations sur la partie enregistrée telles que le temps de jeu, l'heure de la dernière sauvegarde (les deux étant formatés de sorte à être stockés sous forme de chaîne de caractères), ou le taux de complétion. Le joueur a alors la possibilité de retourner au menu principale ou bien de charger la partie sauvegardée. Dans ce cas, l'ensemble des informations référencées avec PlayerPrefs seront extraites et chargées au sein du manager d'énigmes, avant de lancer la partie. Si aucune partie n'est sauvegardée, le menu affichera juste un

message le signalant, proposant au joueur de revenir en arrière afin d'en démarrer une nouvelle.

Pour tester cela, nous ajoutons une clé de "référence d'existence" qui est utilisé pour chaque seconde, indépendamment de la configuration de la pièce. On teste alors si une telle clé existe lors de la charge pour déterminer si une partie est ou non sauvegardée.

Cela permet de conserver la génération sur laquelle l'utilisateur jouait avant sa sauvegarde, afin de conserver son avancée et ses repères.

FIGURE 28 – Menu de chargement de partie si une partie est sauvegardée

FIGURE 29 – Menu de chargement de partie si aucune partie n'est sauvegardée

➤ Le manager de jeu

Le manager de jeu ou *GameManager* est un élément essentiel dans un jeu développé sur Unity. Dans notre cas, il permet de conserver des *GameObject* d'une scène à l'autre grâce à une liste *DontDestroyOnLoad*[27]. Dans cette liste se trouve donc les *GameObjects* suivant :

- Manager de jeu
- Manager d'énigme
- Inventaire

27. List à ne pas détruire au chargement d'une scène

4.4 Les "Années 20" : version procédurale

Au cours du projet et des semaines écoulées, nous nous sommes rendus compte qu'il était bien plus intéressant de mettre de côté notre scénario et de plutôt privilégier une autre façon de structurer notre pièce initiale. En effet, changer d'époque et construire un nouveau décor auraient été une perte de temps par rapport à l'ajout de nouveaux éléments dans le premier décor.

C'est pourquoi nous avons décidé de créer une version procédurale de notre pièce. En gardant seulement les murs et les portes, nous avons randomisé l'intérieur de notre décor.

4.4.1 Conception

Tout d'abord, nous avons créé deux nouveaux supports "template" qui permettent de placer une énigme au choix. Le but étant d'avoir un meuble que l'on peut positionner au mur et l'autre sur une table.

FIGURE 30 – Objet sur une table

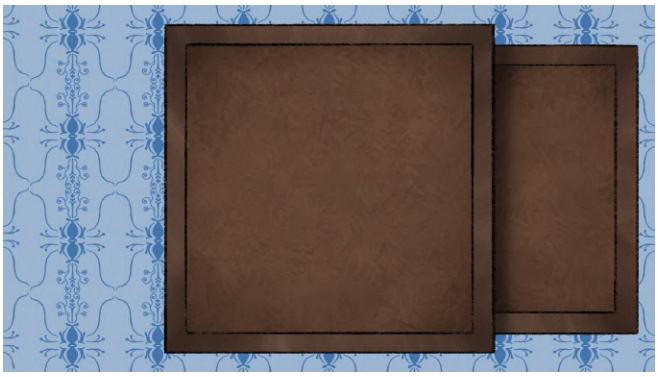

FIGURE 31 – Objet sur le mur avec tiroir droit de résolution ouvert

Après réflexion, nous avons eu différentes idées d'intégration d'énigmes dans ces espaces : un taquin, un jeu mathématique de type *kakuro/perplexed*[28], les tours d'Hanoï, un sudoku et un *Simon*[29]. Nous disposons toujours des énigmes détaillées plus tôt.

28. L'idée est issue d'un ancien projet de TER L2 de Christina encadré par M. Boudet.
29. Jeu consistant à reproduire une série de signaux.

4.4.2 Réalisation

Étant donné que ces énigmes pourront être intégrées dans deux scènes différentes, il a été nécessaire de gérer la position des objets dans la scène dans des *Scripts* et non plus dans l'éditeur de scène du moteur. De plus, nous avons développé ces puzzles dans des scènes uniques, pour pouvoir créer des *Prefab*[30].

➤ Taquin à 8 pièces

Habituellement, le jeu se constitue de 15 pièces. Cependant, il est possible de ne pas réussir facilement à débloquer le jeu, c'est la raison pour laquelle nous avons délibérément choisi de réduire le jeu à 8 pièces. Nous avons volontairement choisi de dessiner les chiffres en écriture romaine.

Au lancement de l'énigme, les positions initiales de chacune des pièces sont générées aléatoirement. On utilise le pseudo-aléatoire du moteur *Unity*, dont la *seed*[31] est dépendante du temps système.

Le jeu est représenté comme une matrice. Du côté du programme, c'est une matrice de positions. Du côté de la scène, chaque pièce est implémentée comme un *GameObject* différent. Ainsi, il est possible d'y ajouter un script pour traiter les interactions avec le joueur. Lorsqu'un clic est détecté par la boîte englobante de l'objet, on cherche s'il est possible de bouger ce dernier selon les règles suivantes :

— Un mouvement est possible si la pièce est adjacente à une case vide.

— Si le mouvement est possible, la position du *GameObject* correspondant est déplacée vers cet emplacement vide.

Pour pouvoir déterminer si le puzzle est complété, nous avons implémenté une matrice résultat contentant les positions attendues. On compare alors la position de chaque pièce avec sa position dans la matrice résultat. Si toutes les pièces sont à leur emplacement, alors le puzzle est complété.

30. Objet de Unity indépendant d'une scène, instanciable en cours d'exécution.
31. Nombre permettant d'initialiser un générateur de nombre pseudo-aléatoire

FIGURE 32 – Taquin à 8 pièces

➤ Jeu mathématique

Le but du jeu est de modifier les quatre nombres afin qu'ils correspondent aux sommes indiquées en colonne ou en ligne. Nous avons décidé d'augmenter la difficulté en mettant des chiffres entre 1 et 99, et non pas 1 et 10.

Nous avons développé le jeu en utilisant un *Canvas* d'interface utilisateur ainsi que des *GameObjects*. Ceux-ci communiquent à travers un script. Le canvas d'interface utilisateur enregistre les clics, et augmente le chiffre sur lequel le joueur a cliqué. Le script s'occupe ensuite d'incrémenter correctement les sommes.

Le puzzle est complété une fois que les sommes sont égales aux sommes affichées en doré. Ces sommes en doré sont générées pseudo-aléatoirement comme pour le Taquin.

FIGURE 33 –

➤ Les tours d'Hanoï

Les tours d'Hanoï était un jeu en bois que possédait Christina enfant. Elle s'est souvenue de l'existence de ce jeu et a proposé d'implémenter le jeu.

Le jeu se compose de la façon suivante :

— 3 disques de tailles différentes

— 3 tours

L'objectif est de placer les 3 disques de taille croissante (du haut vers le bas) de la tour gauche sur la tour droite en respectant les règles suivantes :

— Un mouvement de disque est possible si la tour sur laquelle on veut le placer est vide ou contient uniquement des disques de taille plus grande.

— On ne peut déplacer qu'un disque à la fois, et toujours celui du dessus.

Nous avons créé 3 *GameObjects* vides correspondants à chaque tour. Dans le *GameObject* de la tour gauche, nous avons créé une relation parent-enfant en glissant les trois *GameObjects* disques dans un ordre précis. En effet, le dernier enfant correspond à l'enfant de taille la plus petite. Nous avons mis des *Colliders2D* pour tous les *GameObjects* précédemment cités.

En s'inspirant de ce qui a été implémenté pour la bibliothèque, nous avons implémenté un *Drag and Drop*[32] pour le déplacement de chaque disque. Grâce à la hiérarchie parent-enfant, nous avons pu utiliser un *Script* autorisant uniquement le déplacement du dernier enfant.

Quand l'utilisateur dépose un disque sur une tour, le programme détecte la collision avec cette dernière. Si le mouvement est possible, alors il est réalisé. S'il ne l'est pas, le disque est replacé à sa position initiale.

Grâce aux *Colliders*, nous avons fait en sorte que le disque soit repositionné de manière centrée par rapport au *Collider* du parent. Pour simuler le positionnement en Y, deux options possibles :

— Si le disque est placé sur une tour vide, on copie la position basse du *Collider* de la tour et on décale de - 0.25.

— Si le disque est placé sur un autre enfant, on copie la position haute du *Collider* de celui-ci et on décale de -0.15.

FIGURE 34 – Les tours d'Hanoï

➤ Simon

Le but du jeu est de répéter une séquence de signaux.

La scène se compose alors de 4 ampoules et 4 boutons. Le jeu jouera une séquence aléatoire en allumant les ampoules. Le joueur devra ensuite appuyer sur les boutons dans le même ordre afin de reproduire la séquence. Le puzzle est complété quand 3 séquences ont correctement été reproduites. Ces trois séquences aléatoires sont de longueur croissante (3, 4 puis 5). Si le joueur se trompe, la séquence en cours est rejouée.

Chaque bouton est un *GameObject* avec son *BoxCollider2D*. Une classe *Manager* utilise les interactions pour mettre à jour l'état de l'énigme (l'indice de la séquence, et l'indice du bouton dans cette séquence). L'implémentation de ce jeu impliquait l'utilisation du temps, de l'attente passive. Cette attente a été implémentée à l'aide de *Coroutine* de *Unity*. Il est donc possible de lancer et stopper ces exécutions à volonté, et d'attendre les entrées utilisateurs sans geler le reste du jeu.

FIGURE 35 – Simon

32. Drag and drop : Glisser-Déposer

➤ Sudoku

Le sudoku est une grille de chiffre (de taille 9×9) à compléter en respectant les règles suivantes :

- Les 9 chiffres vont 1 à 9.
- Chaque ligne et chaque colonne doit contenir les 9 chiffres.
- Les lignes et les colonnes ne peuvent donc pas contenir deux fois le même chiffre.
- Pour chacune des 9 sous-grilles de taille 3×3 que contient la grille, on doit trouver les 9 chiffres.
- Ces sous-grilles ne peuvent non plus contenir deux fois le même chiffre.

Un générateur aléatoire de sudoku est une tâche ardue et nous avons décidé de considérer le problème autrement. En effet, la génération aléatoire de sudoku peut être très gourmande en temps d'exécution. De plus, la résolution d'un sudoku peut l'être tout autant et notre objectif n'est pas d'imposer au joueur cette tâche.

C'est pourquoi nous avons décidé d'implémenter une grille complète. Ensuite, le programme tire aléatoirement 9 cases (le tirage se réalise comme pour les jeux précédents), et retire le chiffre qui s'y trouve. L'objectif du joueur sera alors de compléter ce sudoku. La tâche demandée ici est bien moins complexe, et cela garantit aussi que le sudoku soit toujours résoluble.

Le sudoku est donc un *GameObject* parent des 81 chiffres (chacun dans un *GameObject* en soi). Chaque chiffre est un *Prefab* avec un *BoxCollider2D* et un script. Ce script définit si un chiffre est "modifiable" ou non, et gère l'interaction utilisateur. Si le joueur clique sur un chiffre rouge, sa valeur est incrémentée. En fonction de la propriété "modifiable" le *Sprite* du chiffre est un *Sprite* rouge.

FIGURE 36 – Sudoku avec valeurs en rouge à modifier

4.4.3 Génération procédurale de scène

Pour générer procéduralement notre partie, il faut d'abord générer une séquence d'énigme. On a alors décidé d'implémenter cette génération en arrière-plan, "hors" du jeu, puis d'intégrer la séquence dans un second temps.

➤ Génération procédurale de séquence

Nous avons d'abord défini trois classes pour représenter les éléments d'une partie.

— La classe *ContainerItem* représente les objets de la partie, tels que les clés, les papiers. L'objectif de cette classe est de représenter les dépendances *Key-Lock* ou *Clé-Cadenas* que l'on détaillera ensuite.

— La classe *Container* représente une énigme ou un meuble contenant un objet (le chiffonnier n'est pas une énigme en soi mais est important dans la séquence de résolution de la partie).

— La classe *Sequence*, contient une liste de *Container* et représente donc la séquence d'énigmes.

Notre objectif sera donc de générer une séquence aléatoire, en respectant les dépendances.

➢ Dépendances

Une dépendance *Clé-Cadenas* est assez intuitive. Un objet est verrouillé et a besoin d'un autre objet pour être déverrouillé. Dans notre cas, ceci ne concerne pas nécessairement des clés et des cadenas. Par exemple, la bouteille doit être "déverrouillée" (ici cassée plus exactement) par un marteau.

Nous avons alors dans un premier temps défini tous ces paramètres. Autrement dit, nous avons implémenté un programme *Randomizer*, les *Container* et *ContainerItem* de notre partie ainsi que les dépendances suivantes :

— Le chiffonnier a besoin de la clé en acier pour être ouvert

— La bouteille a besoin du marteau pour être cassée

— La bibliothèque a besoin du livre pour être complétée.

"Respecter" une dépendance signifie donc que si un conteneur a besoin d'une clé, alors un autre la contiendra. Il faudra également faire attention à éviter les dépendances croisées : par exemple, la clé du chiffonnier qui se trouve dans la bibliothèque, alors que le chiffonnier contient le livre.

L'algorithme de génération procédurale de séquence fonctionnera comme suit :

1. Définitions des objets et des dépendances entre eux
2. Tirage aléatoire de k conteneurs
3. Résolution des dépendances
4. Résolution des objets nécessaire au code final
5. Vérification de la séquence (Retour en 2. si la séquence n'est pas valide)

➢ Résolution

Nous avons défini deux méthodes de "résolution". Nous définissons "résolution" par l'ajout aux conteneurs des objets nécessaires au respect des dépendances. Pour le code final, nous avons besoin de 4 chiffres. Ces 4 chiffres peuvent être trouvés soit sur 4 papiers différents, soit sur 3 papiers différents et sur la lampe. Donc il est nécessaire de s'assurer que le bon nombre de papiers est ajouté à la séquence.

Algorithm 1 Résolution des dépendances

Input: Sequence S
Output: Sequence S modifiée

```
for Conteneur c in sequence.conteneurs do
    for Object o in c.dependances do
        Créer une liste R de conteneur pouvant résoudre la dépendance
        for Conteneur r in sequence.conteneurs do
            if r peut contenir o then
                Ajouter r à R
            end if
            Mélanger R
            Ajouter o à R[0]
        end for
    end for
```

La résolution pour les papiers fonctionne exactement de la même manière, mais au lieu de parcourir les dépendances de chacun des conteneurs de la séquence, on boucle avec 3 ou 4 papiers.

➢ Mélange

Algorithm 2 Mélange

Input: Liste L d'éléments T
Output: Liste L mélangée

```
n ← L.nombreElements
while n > 1 do                                   ▷ x
    n ← n − 1
    k ← nombrePseudoAleatoire ∈ [0, n + 1]
    value ← L[k]
    L[k] ← L[n]
    L[n] ← value
```

➢ Vérification de la séquence

Pour vérifier la séquence on suit la procédure suivante :

1. Parcours de tous les conteneurs, et vérification du respect des dépendances (i.e. si un autre conteneur de la séquence contient effectivement la clé)
2. Vérification du nombre de papiers (ou chiffres) contenus dans la séquence (bien égal à 4)

➤ Intégration dans la scène

Une fois la séquence générée, il faut intégrer les conteneurs et les objets dans les scènes.

Pour cela, nous avons d'abord ajouté à chacune de nos scènes des *GameObject* vides, qui vont accueillir les *Prefab* des différents conteneurs. Ces *Prefab* ont été créés de façon à devoir modifier le moins de chose possible dans leurs propriétés. A l'instanciation, on leur donne une position et une rotation, puis on affecte le *GameObject* vide comme parent. La problématique est donc maintenant la suivante : dans quels emplacements place-t-on les *Prefab* ?

On ajoute une propriété à un conteneur ; un booléen indiquant s'il peut être placé sur un mur. Ensuite, on définit une liste d'emplacements tels qu'un emplacement contient un indice de scène, un booléen indiquant si c'est un emplacement mural, et un indice d'emplacement dans la scène. En effet, il y a plusieurs emplacements par scène. Comme la séquence est déjà aléatoire, nous n'avons pas besoin d'affecter aléatoirement ces emplacements. Nous avons décidé de parcourir simplement la séquence et d'affecter le premier emplacement possible. Cependant, nous avons tout de même décidé que le piano (s'il est présent dans la séquence) restera à sa position habituelle, du fait de la taille imposante du *Sprite*. Nous trouvions dommage qu'il se place ailleurs.

Un emplacement peut accueillir un conteneur selon les règles suivantes :

— Si l'emplacement ne contient pas déjà un conteneur
— Si l'emplacement est mural et si le conteneur peut être placé au mur

Avec ces règles on peut ensuite instancier les différents *Prefabs* au chargement de la scène à laquelle leur emplacement appartient. Les méthodes réalisant cela sont placées dans une classe *PropsManager* dont le *GameObject* est dans la liste *DontDestroyOnLoad* du *GameManager*. Malgré tout, quelques ajustements sont ensuite

nécessaires pour garder les scènes agréables visuellement.

➢ Intégration des objets

Les objets ont aussi un *Prefab* attribué. Grâce à notre séquence, on sait où chacun des objets doit se trouver. Alors, on peut directement parcourir cette liste et instancier les *Prefab* au chargement de la scène correspondante.

Finalement, il a fallu apporter quelques ajustements aux différents gestionnaires d'énigmes pour prendre en compte cette génération procédurale.

5 Annexe

Ci-dessous, quelques extraits de *Scripts*. Notre projet contient d'autres scripts, mais par soucis de lisibilité, nous avons mis qu'un code pour chaque objet de la version histoire, ensuite, le randomizer et le conteneur de la version procédurale et également ajouté les scripts qui concernent la gestion du jeu (manager, caméra, ...).

WheelRotation.cs (roue en page 15)

```
using UnityEngine;

public class WheelRotation : MonoBehaviour{
    private bool completed = false;
    private float margin = 0.05f;
    private Vector3 _lastMousePosition;
    public float rotationSpeed = 10.0f;
    public float maxRotationAngle = 360.0f;
    public float minMouseDelta = 0.1f;
    public Sprite[] spriteArray;

    private GameObject managerObject;
    private WheelManager manager;
    private int index;

    private void SetRandomSprite(int n){
        SpriteRenderer spriteRenderer = gameObject.GetComponent <SpriteRenderer>();
        spriteRenderer.sprite = spriteArray[n - 1];
    }

    private void Start(){
        managerObject = GameObject.Find("Manager");
        manager = managerObject.GetComponent<WheelManager>();
        int random_set = manager.random_set;
        SetRandomSprite(random_set);
        float random_rotation = Random.Range(0.0f, 360.0f);
        gameObject.transform.localRotation = Quaternion.Euler(0, 0, random_rotation);
        string name = gameObject.name;
        if (name == "Wheel_2") index = 2;
        if (name == "Wheel_3") index = 3;
        if (name == "Wheel_4") index = 4;
    }

    private void OnMouseDown(){
        _lastMousePosition = Input.mousePosition;
    }
```

```
37
38      private void OnMouseUp(){
39          _lastMousePosition = Input.mousePosition;
40          Quaternion currentRotation = transform.rotation;
41          float rotationValue = currentRotation[2];
42          if ((rotationValue > (0.0f - margin)) && (rotationValue < (0.0f + margin))){
43              completed = true;
44              manager.SetWheelCorrect(index);
45          }
46      }
47
48      private void OnMouseDrag(){
49          if (!completed){
50              Vector3 objectPosition = Camera.main.WorldToScreenPoint(transform.position);
51              Vector3 mouseDelta = Input.mousePosition - objectPosition;
52
53              if (mouseDelta.magnitude < minMouseDelta) return;
54
55              float angle = Vector3.Angle(transform.right, mouseDelta);
56
57              float rotationAmount = angle * rotationSpeed * Time.deltaTime;
58              rotationAmount = Mathf.Clamp(rotationAmount, -maxRotationAngle,
    ↪  maxRotationAngle);
59
60              Vector3 cross = Vector3.Cross(transform.right, mouseDelta);
61              float direction = Mathf.Sign(cross.z);
62
63              transform.Rotate(Vector3.forward, direction * rotationAmount, Space.World);
64
65              _lastMousePosition = Input.mousePosition;
66          }
67      }
68  }
```

LampeSwitchController.cs (lampe en page 16)

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class LampeSwitchController : MonoBehaviour{
6       public Sprite light_on, light_off;
7       public GameObject number;
8       private bool isOn = false;
9
```

```
    void Start(){
        isOn = A20_Manager.instance.light_on;
        gameObject.GetComponent<SpriteRenderer>().sprite = isOn ? light_on : light_off;
        number.SetActive(A20_Manager.instance.light_on);
    }

    public void OnMouseDown(){
        if(isOn){
            isOn = false;
            gameObject.GetComponent<SpriteRenderer>().sprite = light_off;
            number.SetActive(isOn);
        }else{
            isOn = true ;
            gameObject.GetComponent<SpriteRenderer>().sprite = light_on;
            number.SetActive(isOn);
        }
        A20_Manager.instance.light_on = isOn;
    }
}
```

PianoFrameManager.cs (piano en page 16)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PianoFrameManager : MonoBehaviour{
    private Dictionary<char, string> sprite_dictionary;

    void Awake(){
        sprite_dictionary = new Dictionary<char, string>();
        string path_start = Application.dataPath + "/Images/A20/P1/";
        sprite_dictionary.Add('A', "A20_lettre_A");
        sprite_dictionary.Add('B', "A20_lettre_B");
        sprite_dictionary.Add('C', "A20_lettre_C");
        sprite_dictionary.Add('D', "A20_lettre_D");
        sprite_dictionary.Add('E', "A20_lettre_E");
        sprite_dictionary.Add('F', "A20_lettre_F");
        sprite_dictionary.Add('G', "A20_lettre_G");

        GameObject a20ManagerObject = GameObject.Find("A20_Manager");
        A20_Manager a20_manager = a20ManagerObject.GetComponent<A20_Manager>();
        string sequence = a20_manager.pianoGenerator.sequence_to_display;
        int seq_length = sequence.Length;

```

```
        Sprite sprite;
        GameObject spriteObject;
        SpriteRenderer spriteRenderer;
        string path;
        float sequence_extension = 0.4375f * seq_length;

        Sprite[] sprites = new Sprite[seq_length];
        GameObject[] spriteObjects = new GameObject[seq_length];
        SpriteRenderer[] spriteRenderers = new SpriteRenderer[seq_length];

        for (int i = 0; i < seq_length; i++){
            sprite_dictionary.TryGetValue(sequence[i], out path);
            sprites[i] = Resources.Load<Sprite>(path);
            spriteObjects[i] = new GameObject("Lettre_n"+i.ToString());
            spriteRenderers[i] = spriteObjects[i].AddComponent<SpriteRenderer>();
            spriteRenderers[i].sprite = sprites[i];
            spriteObjects[i].transform.position = new Vector3((0.5f * i) -
↪   (sequence_extension/2), -3.75f, 1);
        }
    }
}
```

OuvertureTiroir.cs (chiffonnier en page 19)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class OuvertureTiroir : MonoBehaviour{
    public GameObject TiroirFerme;
    public GameObject TiroirOuvert;

    void Start(){
        TiroirOuvert.SetActive(false);
        TiroirFerme.SetActive(true);
    }

    public virtual void OnMouseDown(){
        if(!A20_Manager.instance.silverlock_opened) return;
        if(TiroirFerme.activeSelf){
            TiroirFerme.SetActive(false);
            TiroirOuvert.SetActive(true);
        }else{
            TiroirFerme.SetActive(true);
            TiroirOuvert.SetActive(false);
```

```
        }
    }
}
```

BottleController.cs (bouteille en page 19)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BottleController : ItemReceiver
{
    public GameObject paper;

    void Start(){
        SpriteRenderer renderer = gameObject.GetComponent<SpriteRenderer>();
        renderer.sprite = A20_Manager.instance.bottle_broken ? afterUse : beforeUse;
        gameObject.transform.position = A20_Manager.instance.bottle_broken ? new
↪   Vector3(72.2f, 3.5f, 33.84f) : new Vector3(68.92f, 6.8f, 35.25f);
    }

    public override bool Accept(int id){
        base.Accept(id);
        A20_Manager.instance.bottle_broken = hasAccepted;
        paper.SetActive(hasAccepted);
        if (hasAccepted) gameObject.transform.position = new Vector3(72.2f, 3.5f, 33.84f);
        return hasAccepted;
    }

    public override void OnMouseDown(){
        if(A20_Manager.instance.bottle_broken) return;
        base.OnMouseDown();
    }
}
```

DragAndSwap.cs (bibliothèque en page 20)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DragAndSwap : MonoBehaviour{
    public GameObject[] objectsToAssign;
    public Collider2D[] collidersToAssign;
```

```
8
9
10      private Vector3 initialPosition;
11      private Vector3 finalPosition;
12      private bool isDragging = false;
13      private int initialIndex = -1;
14
15      private void Start(){
16          initialPosition = transform.position;
17      }
18
19      private void OnMouseDown(){
20          if(A20_Manager.instance.bookcase_completed) return;
21          if (GetComponent<Collider2D>().OverlapPoint(
22          Camera.main.ScreenToWorldPoint(Input.mousePosition))){
23              initialPosition = transform.position;
24              initialIndex = -1;
25              for (int i = 0; i < objectsToAssign.Length; i++){
26                  if (objectsToAssign[i] == gameObject){
27                      initialIndex = i;
28                      break;
29                  }
30              }
31              isDragging = true;
32          }
33      }
34
35      private void OnMouseDrag(){
36          if(A20_Manager.instance.bookcase_completed) return;
37          if (isDragging){
38              transform.position = Camera.main.ScreenToWorldPoint(new
    ↪  Vector3(Input.mousePosition.x, Input.mousePosition.y, 10));
39              transform.position = Vector3.Lerp(transform.position, finalPosition, 0.1f);
40          }
41      }
42
43      private void OnMouseUp(){
44          if(A20_Manager.instance.bookcase_completed) return;
45          isDragging = false;
46          bool isColliding = false;
47          for (int i = 0; i < objectsToAssign.Length; i++){
48              if (objectsToAssign[i] != gameObject &&
    ↪  GetComponent<Collider2D>().IsTouching(collidersToAssign[i])){
49                  Vector3 temp = objectsToAssign[i].transform.position;
50                  objectsToAssign[i].transform.position = initialPosition;
51                  transform.position = temp;
```

```
52                  isColliding = true;
53                  break;
54              }
55          }
56
57          if (!isColliding){
58              if (initialIndex >= 0){
59                  transform.position = initialPosition;
60              } else {
61                  transform.position = Vector3.Lerp(transform.position, initialPosition,
    ↪  0.1f);
62              }
63          }
64      }
65  }
```

UICodeLock.cs (code à 4 chiffres en page 21)

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.UI;
5   using UnityEngine.EventSystems;
6   using TMPro;
7   using System;
8
9   public class UICodeLock : MonoBehaviour{
10      public static bool isCodeValid = false;
11      public GameObject canvas;
12
13      public void OnDigitClicked(){
14          if(isCodeValid) return;
15          GameObject digit = EventSystem.current.currentSelectedGameObject;
16          int current = Int32.Parse(digit.GetComponentInChildren<TextMeshProUGUI>().text);
17          int next;
18          if(current == 9) next = 0;
19          else next = current+1;
20          digit.GetComponentInChildren<TextMeshProUGUI>().text = next.ToString();
21          check();
22      }
23
24      private void check(){
25          Button number0 = canvas.GetComponentsInChildren<Button>()[0];
26          Button number1 = canvas.GetComponentsInChildren<Button>()[1];
27          Button number2 = canvas.GetComponentsInChildren<Button>()[2];
```

```
        Button number3 = canvas.GetComponentsInChildren<Button>()[3];

        int n0, n1, n2, n3;
        n0 = Int32.Parse(number0.GetComponentInChildren<TextMeshProUGUI>().text);
        n1 = Int32.Parse(number1.GetComponentInChildren<TextMeshProUGUI>().text);
        n2 = Int32.Parse(number2.GetComponentInChildren<TextMeshProUGUI>().text);
        n3 = Int32.Parse(number3.GetComponentInChildren<TextMeshProUGUI>().text);

        if(n0 == 4 && n1 == 5 && n2 == 9 && n3 == 8){
            isCodeValid = true;
            A20_Manager.instance.tableCode_correct = true;
        }
    }
}
```

FinalDoor.cs (porte finale en page 23)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FinalDoor : ObjectCollider{
    void Start(){
        gameObject.SetActive(A20_Manager.instance.finalLock_opened);
    }
}
```

Randomizer.cs (Mélange de la version procédurale en page 38)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;
using System;

public class Randomizer{
    ContainerItem paper = new ContainerItem(0,"Paper");
    ContainerItem silverkey = new ContainerItem(1,"SilverKey");
    ContainerItem number = new ContainerItem(2,"Number");
    ContainerItem hammer = new ContainerItem(3,"Hammer");
    ContainerItem book = new ContainerItem(4,"Book");

    int indexLight = -1;
    public Randomizer(){}
```

```
16
17      public Sequence Generate(){
18          Sequence sequence;
19          do{
20              sequence = GenerateSequence(6);
21              if(isPossibleValidSequence(sequence)) {
22                  ResolveDependencies(sequence);
23                  ResolvePapers(sequence);
24              }
25              else break;
26          }while(!isValidSequence(sequence));
27
28          foreach (Container container in sequence.containers){
29              Debug.Log("Container " + container.id + " - " + container.name + ":");
30              Debug.Log("- Objects " + string.Join(", ", container.objects));
31              Debug.Log("- Dependencies " + string.Join(", ", container.needs));
32              Debug.Log("- Key:" + container.key);
33              Debug.Log("- Is Locked " +  container.isLocked);
34          }
35          return sequence;
36      }
37
38      private List<T> Shuffle<T>(List<T> list){
39          int n = list.Count;
40          while (n > 1){
41              n--;
42              int k = (int) UnityEngine.Random.Range(0,n+1);
43              T _value = list[k];
44              list[k] = list[n];
45              list[n] = _value;
46          }
47          return list;
48      }
49
50      private bool isPossibleValidSequence(Sequence sequence){
51          foreach (Container container in sequence.containers){
52              if (container.isLocked &&
    ↪ !container.canSatisfyDependency(sequence.containers)){
53                  return false;
54              }
55          }
56          return true;
57      }
58
59      private bool isValidSequence(Sequence sequence){
60          int nbPaper = 0;
```

```
61          foreach (Container container in sequence.containers){
62              if(container.objects.Count <=0) return false;
63              if (container.isLocked &&
↪   container.IsDependencySatisfied(sequence.containers)){
64                  container.isLocked = false;
65              }
66          }
67          foreach(Container container in sequence.containers){
68              if(container.isLocked == true) return false;
69              nbPaper += CountItem(container, paper);
70              if(container.objects.Contains(number)) nbPaper++;
71          }
72          return nbPaper >=4;
73      }
74
75
76      private void ResolveDependencies(Sequence sequence){
77          foreach(Container container in sequence.containers){
78              foreach(ContainerItem dependency in container.needs){
79                  List<Container> possibleResolvers = new List<Container>();
80                  foreach(Container resolver in sequence.containers){
81                      if(resolver.isAddSafe(dependency)){
82                          if(!container.objects.Contains(resolver.key))
83                              possibleResolvers.Add(resolver);
84                      }
85                  }
86                  possibleResolvers = Shuffle(possibleResolvers);
87                  if(possibleResolvers.Count > 0)
↪   possibleResolvers[0].objects.Add(dependency);
88              }
89          }
90      }
91
92      private void ResolvePapers(Sequence sequence){
93          int papernb ;
94          if(indexLight != -1){
95              sequence.containers[indexLight].objects.Add(number);
96              papernb = 3;
97          }else papernb = 4;
98
99          for(int k = 0; k<papernb; k++){
100             List<Container> resolvers = new List<Container>();
101             foreach(Container container in sequence.containers){
102                 if(container.isAddSafe(paper)){
103                     resolvers.Add(container);
104                 }
```

```
            }
            resolvers = Shuffle(resolvers);
            if(resolvers.Count> 0) resolvers[0].objects.Add(paper);
        }
    }

    private int CountItem(Container container, ContainerItem item){
        int counter = 0;
        foreach(ContainerItem possible in container.objects){
            if(possible.Equals(item)) counter++;
        }
        return counter;
    }

    private Sequence GenerateSequence(int length){
        Container piano = new Container(0);
        piano.name = "Piano";
        piano.isLocked = false;
        piano.canContain.Add(paper); piano.canContain.Add(silverkey);
        piano.maxObjects = 1;

        Container wheel = new Container(1);
        wheel.name ="Wheel";
        wheel.isLocked = false;
        wheel.canContain.Add(paper);
        wheel.maxObjects = 1;

        Container light = new Container(2);
        light.name = "Light";
        light.isLocked = false;
        light.canContain.Add(number);
        light.maxObjects = 1;

        Container furniture = new Container(3);
        furniture.name = "Chiffonier";
        furniture.isLocked = true;
        furniture.key = silverkey;
        furniture.needs.Add(silverkey);
        furniture.canContain.Add(paper);
        furniture.canContain.Add(book);
        furniture.canContain.Add(hammer);
        furniture.maxObjects = 2;

        Container bookcase = new Container(4);
        bookcase.name = "Bookcase";
        bookcase.isLocked = true;
```

```
151         bookcase.key = book;
152         bookcase.needs.Add(book);
153         bookcase.canContain.Add(silverkey);
154         bookcase.canContain.Add(hammer);
155         bookcase.canContain.Add(paper);
156         bookcase.maxObjects = 1;
157         bookcase.canBeOnWall = true;
158 
159         Container bottle = new Container(5);
160         bottle.name = "Bottle";
161         bottle.isLocked = true;
162         bottle.key = hammer;
163         bottle.needs.Add(hammer);
164         bottle.canContain.Add(paper);
165         bookcase.maxObjects = 1;
166 
167         Container sudoku = new Container(6);
168         sudoku.name = "Sudoku";
169         sudoku.isLocked = false;
170         sudoku.canContain.Add(book);
171         sudoku.canContain.Add(silverkey);
172         sudoku.canContain.Add(paper);
173         sudoku.canContain.Add(hammer);
174         sudoku.maxObjects = 1;
175         sudoku.canBeOnWall = true;
176 
177         Container taquin = new Container(7);
178         taquin.name = "Taquin";
179         taquin.isLocked = false;
180         taquin.canContain.Add(book);
181         taquin.canContain.Add(silverkey);
182         taquin.canContain.Add(paper);
183         taquin.canContain.Add(hammer);
184         taquin.maxObjects = 1;
185         taquin.canBeOnWall = true;
186 
187         Container perplexed = new Container(8);
188         perplexed.name = "Perplexed";
189         perplexed.isLocked = false;
190         perplexed.canContain.Add(book);
191         perplexed.canContain.Add(silverkey);
192         perplexed.canContain.Add(paper);
193         perplexed.canContain.Add(hammer);
194         perplexed.maxObjects = 1;
195         perplexed.canBeOnWall = true;
196 
```

```
197         Container hanoi = new Container(9);
198         hanoi.name = "Hanoi";
199         hanoi.isLocked = false;
200         hanoi.canContain.Add(book);
201         hanoi.canContain.Add(silverkey);
202         hanoi.canContain.Add(paper);
203         hanoi.canContain.Add(hammer);
204         hanoi.maxObjects = 1;
205         hanoi.canBeOnWall = true;
206 
207         Container simon = new Container(10);
208         simon.name = "Simon";
209         simon.isLocked = false;
210         simon.canContain.Add(book);
211         simon.canContain.Add(silverkey);
212         simon.canContain.Add(paper);
213         simon.canContain.Add(hammer);
214         simon.maxObjects = 1;
215         simon.canBeOnWall = true;
216 
217         List<Container> containers = new List<Container>(){
218             piano, wheel, light, furniture, bookcase, bottle, sudoku, taquin, perplexed,
    ↪  hanoi, simon
219         };
220         UnityEngine.Random.InitState((int)System.DateTime.Now.Ticks);
221 
222         Sequence sequence = new Sequence();
223         List<int> added = new List<int>();
224         int counter = containers.Count;
225         int k = 0;
226         containers = Shuffle(containers);
227         for(int i = 0; i<length; i++) {
228             sequence.containers.Add(containers[i]);
229         }
230 
231         for(int i = 0 ; i<sequence.containers.Count; i++){
232             if(sequence.containers[i].name == "Light") indexLight = i;
233         }
234         return sequence;
235     }
236 }
```

ContainerItem.cs (Conteneur d'item de la version procédurale en page 36)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ContainerItem{
    public string name {get;set;}
    public int id{get;set;}

    public ContainerItem(int _id, string _name){
        id = _id; name = _name;
    }

    public bool Equals(ContainerItem other){
        if(other == null) return false;
        else return this.id == other.id;
    }

    public override string ToString(){
        return name;
    }

}

public class Container{
    public int id { get; set;}
    public string name;
    public List<ContainerItem> objects { get; set;}
    public int indexInSequence { get; set;}
    public ContainerItem key { get; set;}
    public bool isLocked { get; set;}

    public bool canBeOnWall;

    public List<ContainerItem> needs { get; set;}
    public List<ContainerItem> canContain {get; set;}
    public int maxObjects;

    public Container(int _id){
        id = _id;
        objects = new List<ContainerItem>();
        canContain = new List<ContainerItem>();
        needs = new List<ContainerItem>();
        indexInSequence = -1;
        key = null;
        isLocked = false;
        maxObjects = 0;
```

```
47      }
48
49      public bool IsDependencySatisfied(List<Container> previous){
50          foreach(ContainerItem dependency in needs){
51              bool isDependencyMet = false;
52              foreach(Container container in previous){
53                  if(container.objects.Contains(dependency)){
54                      isDependencyMet = true;
55                      break;
56                  }
57              }
58              if(!isDependencyMet) return false;
59          }
60          return true;
61      }
62
63      public bool canSatisfyDependency(List<Container> previous){
64          foreach(ContainerItem dependency in needs){
65              bool isDependencyMet = false;
66              foreach(Container container in previous){
67                  if(container.canContain.Contains(dependency)){
68                      isDependencyMet = true;
69                      break;
70                  }
71              }
72              if(!isDependencyMet) return false;
73          }
74          return true;
75      }
76
77      public bool isAddSafe(ContainerItem item){
78          return canContain.Contains(item) && objects.Count < maxObjects;
79      }
80  }
81
82  public class Sequence{
83      public List<Container> containers {get; set;}
84      public Sequence(){
85          containers = new List<Container>();
86      }
87
88      public Container FindById(int id){
89          return containers.Find(c => (c.id == id));
90      }
91  }
```

LevelLoader.cs (Chargement de scène en page 25)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class LevelLoader : MonoBehaviour{
    public float transitionTime = 1f;
    public Animator transition;
    public static int previous = 0;
    public static Dictionary<int, int> history  = new Dictionary<int, int>();

    void Update(){
        if(Input.GetKeyDown("left")) LoadNextLevel(-1);
        if(Input.GetKeyDown("right")) LoadNextLevel(1);
    }

    public void LoadNextLevel(int increment){
        int levelIndex = SceneManager.GetActiveScene().buildIndex + increment;
        if(levelIndex >= 5) levelIndex = 1;
        else if(levelIndex < 1) levelIndex = 4;
        StartCoroutine(LoadLevel(levelIndex));
    }

    public void LoadPreviousLevel(){
        LoadLevelAt(history[SceneManager.GetActiveScene().buildIndex]);
    }

    public void LoadLevelAt(int levelIndex){
        StartCoroutine(LoadLevel(levelIndex));
    }

    IEnumerator LoadLevel(int levelIndex){
        if(history.ContainsKey(levelIndex)){
            if(SceneManager.GetActiveScene().buildIndex < history[levelIndex]){
                history[levelIndex] = SceneManager.GetActiveScene().buildIndex;
            }
        }else{
            history.Add(levelIndex, SceneManager.GetActiveScene().buildIndex);
        }
        transition.SetTrigger("Start");
        yield return new WaitForSeconds(transitionTime);
        SceneManager.LoadScene(levelIndex);
    }
```

```
}
```

A20_Manager.cs (Manager de la pièce 29)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class A20_Manager : MonoBehaviour
{
    // Puzzles randomization
    public PianoGenerator pianoGenerator;

    // Puzzles state
    public bool piano_completed;
    public bool ragman_opened;
    public bool bottle_broken;
    public bool wheel_completed;
    public bool tableCode_correct;
    public bool finalLock_opened;
    public bool silverlock_opened;
    public bool light_on;
    public bool bookcase_completed;
    public bool book_placed;

    // Items to pick
    public bool hammer_taken; // marteau dans le chiffonnier

    public bool silver_key_taken; // clé argentée (pour le chiffonnier) dans le piano
    public bool golden_key_taken; // clé dorée (pour la porte finale) dans le tiroir à code

    public bool bottle_paper_taken; // papier dans la bouteille en verre (n2)
    public bool bookcase_paper_taken; // papier dans la bibliothque (n3)
    public bool wheel_paper_taken; // papier dans la roue (n4)
    public bool book_taken;

    public int index_missingBook;

    public static A20_Manager instance;

    void Awake()
    {
        if(instance != null){
            Debug.Log("More than one instance of inventory");
            return;
```

```
        }
        instance = this;

        // Puzzles randomization
        pianoGenerator = new PianoGenerator();

        // Puzzles state
        piano_completed = false;
        ragman_opened = false;
        bottle_broken = false;
        wheel_completed = false;
        tableCode_correct = false;
        finalLock_opened = false;
        light_on = false;
        silverlock_opened = false;
        bookcase_completed = false;
        book_taken = false;

        // Items states
        hammer_taken = false;
        silver_key_taken = false;
        golden_key_taken = false;
        bottle_paper_taken = false;
        bookcase_paper_taken = false;
        wheel_paper_taken = false;
        book_placed = false;
    }
}
```

CameraController.cs (Caméra en page 14)

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraController : MonoBehaviour{
    private Vector3 origin, difference;
    private Transform reset;
    private bool isInSlideMode = false;
    private bool isDragged;

    void Start(){
        reset = Camera.main.transform;
    }
```

```
15
16
17      void LateUpdate(){
18          if(isInSlideMode){
19              if(Input.GetMouseButton(1)){
20                  difference = Camera.main.ScreenToWorldPoint(Input.mousePosition) -
    ↪  Camera.main.transform.position;
21                  if(!isDragged){
22                      isDragged = true;
23                      origin = Camera.main.ScreenToWorldPoint(Input.mousePosition);
24                  }
25              } else isDragged = false;
26
27              if(isDragged) {
28                  float X_offset = Math.Max(-18.2659f, Math.Min((origin - difference).x,
    ↪  18.2659f));
29                  Camera.main.transform.position = new
    ↪  Vector3(X_offset,Camera.main.transform.position.y, Camera.main.transform.position.z);
30              }
31          } else {
32              if(Input.GetMouseButton(1)){
33                  if(((Camera.main.transform.eulerAngles.y + Input.GetAxis("Mouse X")) <=
    ↪  8.0f || (Camera.main.transform.eulerAngles.y + Input.GetAxis("Mouse X"))>= 354.0f) &&
34                      ((Camera.main.transform.eulerAngles.x + Input.GetAxis("Mouse Y")) <=
    ↪  8.0f || (Camera.main.transform.eulerAngles.x + Input.GetAxis("Mouse Y")) >= 354.0f))
35                      Camera.main.transform.eulerAngles += new Vector3(Input.GetAxis("Mouse
    ↪  Y"), Input.GetAxis("Mouse X"), 0);
36              }
37          }
38      }
39  }
```

6 Conclusion

6.1 Bilan

Nous allons maintenant tracer le bilan de notre projet.

Si l'on revient sur la partie 2 "Analyse", nous avons réussi à implémenter et rendre fonctionnel le coeur de notre jeu. Nous avons une scène complète avec une suite d'éléments à débloquer pour finir le jeu. Notre jeu a une direction artistique qui lui est propre, chaque élément est cohérent dans l'ensemble. Nous avons parfois dû modifier l'aspect de ces éléments afin de s'accommoder avec la partie technique.

En ce qui concerne les besoins dits non-fonctionnels, nous avons réussi à donner une certaine rejouabilité à notre jeu grâce à la partie procédurale. Nous avons un menu, un écran de pause et une sauvegarde. Les points manquants sont le manque de scénario et la musique mais c'étaient des points facultatifs qui n'impactaient pas la jouabilité.

6.2 Connaissances et apprentissages

Ce projet nous a permis de mettre en pratique et de consolider nos connaissances accumulées tout au long de notre parcours en master IMAGINE.

Il nous a permis de découvrir et nous rendre compte de tous les processus que comporte la création d'un jeu, notamment dans le genre point and click. En effet, il faut beaucoup de réflexion en amont avant de commencer à coder afin d'avoir un jeu cohérent et fonctionnel. Ce n'est pas aussi évident que cela pourrait laisser penser lorsqu'on joue à des jeux de ce genre.

Ce projet nous a aussi mené à réfléchir à ce que la génération procédurale pouvait amener au genre. Lorsque nous avons conçu le jeu, nous l'avons d'abord conçu de façon statique et chaque élément avait sa place, dans la pièce et dans la suite d'énigme. Nous avons dû conceptualiser à nouveau la partie pour étudier les portions qui pouvaient être générées procéduralement. Ce fut un réel défi car il a fallu repenser plusieurs fois nos algorithmes. En effet, les besoins du projet, que ce soit en terme de temps ou de fonctionnalité, nécessitaient des algorithmes rapides et fiables. Nous avons dû trouver un compromis entre jouabilité, rapidité et aléatoire. On se rend compte aujourd'hui qu'il serait même possible d'ajouter encore des dépendances, des objets ou conteneurs dans la partie sans que cela n'altère trop l'expérience du joueur.

Ce projet a été une véritable source d'apprentissage quant à la manipulation d'un outil de travail tel que Unity. En effet, c'est un moteur de jeu dont nous avions que peu voire pas du tout de connaissance et nous avons appris à comprendre son fonctionnement tout au long de notre processus créatif. De plus, nous allons devoir apprendre à l'utiliser au semestre prochain et cette expérience nous donne l'avantage considérable de connaître de bonnes bases.

6.3 Perspectives

Pour la continuité de notre projet, nous pouvons imaginer quelques améliorations sur le jeu pour le faire évoluer. En effet, nous avons eu plusieurs projets qui se sont entrecroisés durant le semestre, ce qui nous a demandé de gérer notre temps différemment. Si nous avions eu la possibilité d'accorder notre disponibilité seulement à la création du jeu, comme le ferait un petit studio de jeu indépendant, nous aurions voulu ajouter d'autres fonctionnalités.

Tout d'abord, nous aurions aimé pouvoir enrichir le scénario que nous avons décrit au tout début du rapport. Avec le thème de la temporalité, il est possible d'imaginer plusieurs autres scènes à conceptualiser, et donc, des mécaniques de jeu propres à chacune des époques. Également, nous aurions aimé raconter l'histoire par de petits éléments à trouver tels que des lettres, des documents, des photos, qui auraient expliqué la présence de notre personnage dans cette pièce.

Dans notre diagramme de Gantt, nous avions inclus une partie sur de la parallaxe qui n'aurait pas été utilisable sur les assets actuels et nous avons préféré abandonner l'idée pour se concentrer sur la génération procédurale. Cela aurait été possible dans le cas d'une scène présentant davantage de profondeur (par exemple, une scène en extérieur).

Nous avons aussi pensé à d'autres énigmes cependant celles-ci se sont révélées assez techniques pour le temps que nous disposions. De plus, il aurait été intéressant d'ajouter plus d'éléments en 3D.