RAPPORT DE SOUTENANCE n°2

EPITA - Projet S2

Ctrl + Alt + Elite

KOPFF Louis, MITTELBRONN Pierre,
PONSOT Thibault, RAUGER Axel, TOUFIR Bashir

Mars 2025

Table des matières

1 Introduction

Ce rapport de soutenance présente l'avancée du projet de jeu vidéo en 2D intitulé *Syndrome Fear*, un jeu axé sur l'exploration et la résolution d'énigmes.

Le développement du jeu a bien avancé depuis la première soutenance, sur plusieurs plans, notamment celui de la réalisation graphique du jeu ou "design", et la partie programmation et développement.

Le rapport détaille les étapes déjà accomplies, les fonctionnalités développées et les choix techniques et créatifs réalisés jusqu'à présent.

2 Avancée du planning

Lors de la création de ce projet, nous avons défini un planning à travers le diagramme de Gantt, que nous avons suivi jusqu'ici.

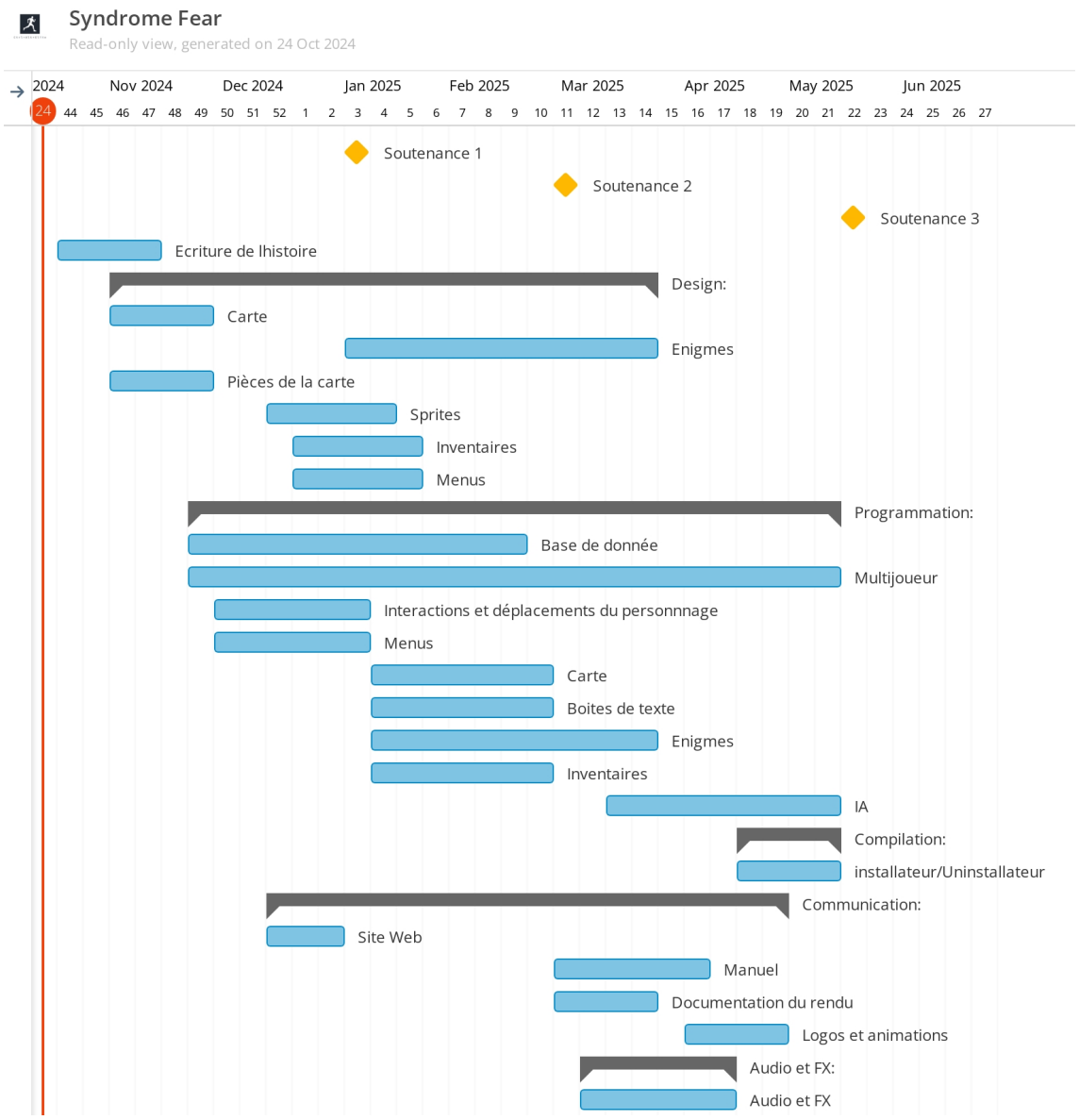

Figure 1: Diagramme de Gantt

Durant cette période, la conception des graphismes du jeu a continué. Nous pouvons citer, notamment, la conception de l'interface utilisateur. Cette interface comprend les menus, les icones de réalisations, ou encore les boutons.

Nous avons également terminé la réalisation de l'inventaire et des menus.

En terme de développement et de programmation, nous avons continué l'implémentation du mode multijoueur, celle de la carte, des énigmes ou encore de l'inventaire.

2.1 Designs du jeu

2.1.1 Les différents niveaux du jeu

Nous avons réfléchi à l'agencement de la carte, afin de permettre une aventure agréable et logique tout au long de l'avancée du joueur. Pour cela, nous avons élaboré le plan le plus logique et le plus cohérent.

Voici donc les différents niveaux réalisés, ainsi que la chambre de départ du joueur :

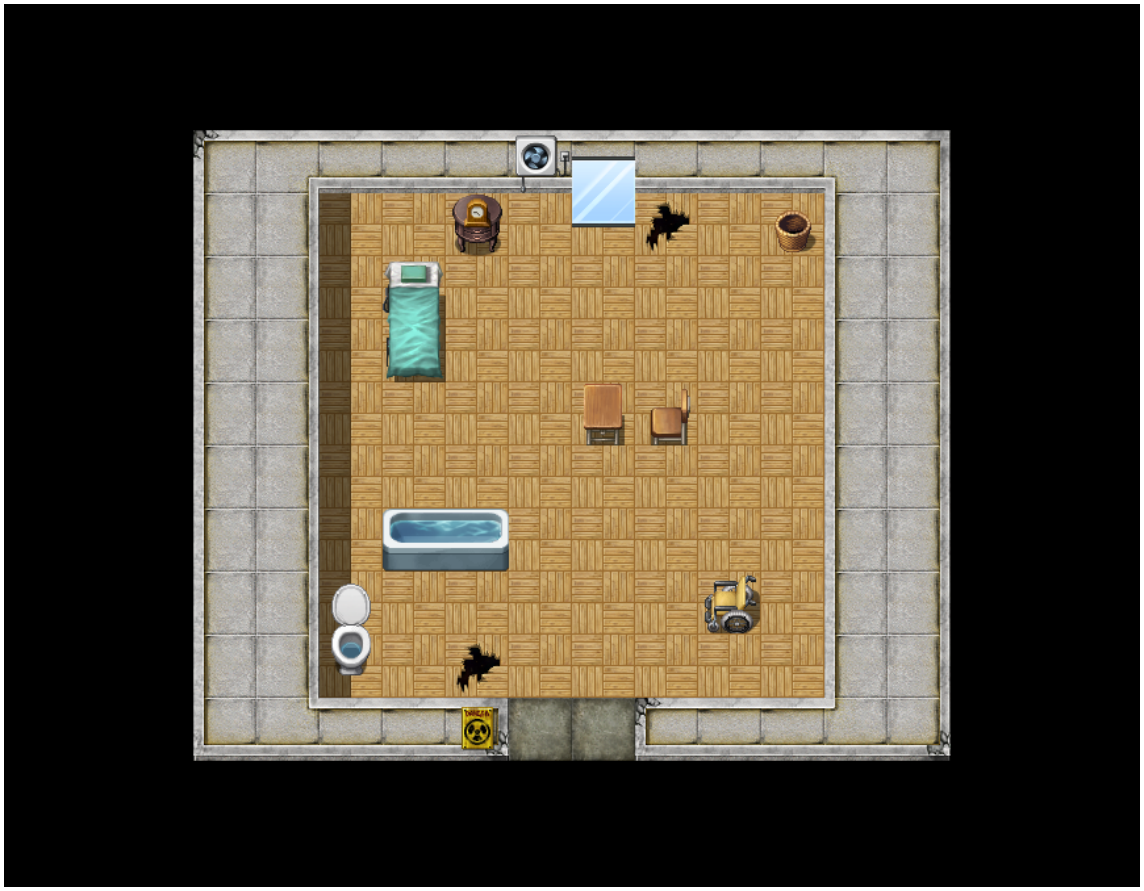

Figure 2: Chambre de départ

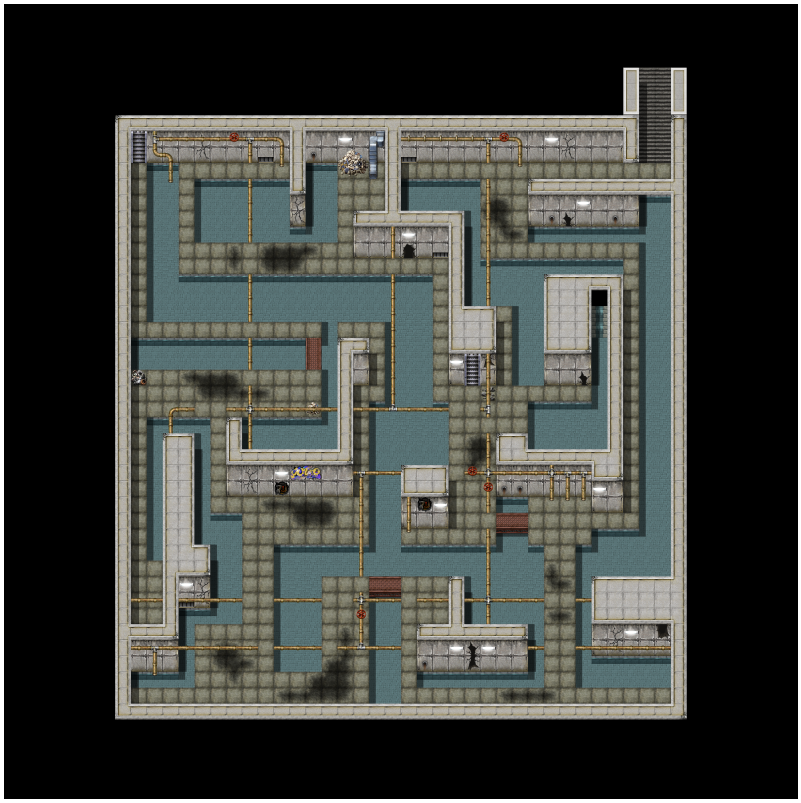

Figure 3: Niveau 2

Figure 4: Niveau 3

Figure 5: Niveau 4

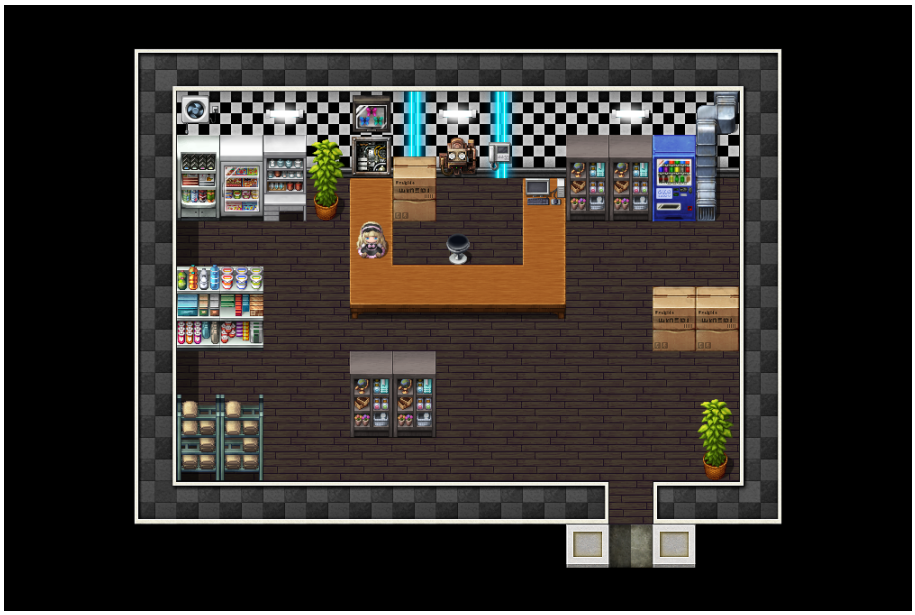

Figure 6: Niveau 5

2.1.2 L'inventaire

L'inventaire est représenté sous la forme d'un carnet pour faciliter l'immersion et offrir une consultation intuitive. Il comporte des cases clairement définies pour les objets collectés et une carte complète immédiatement visible, aidant le joueur à se repérer facilement dans l'environnement.

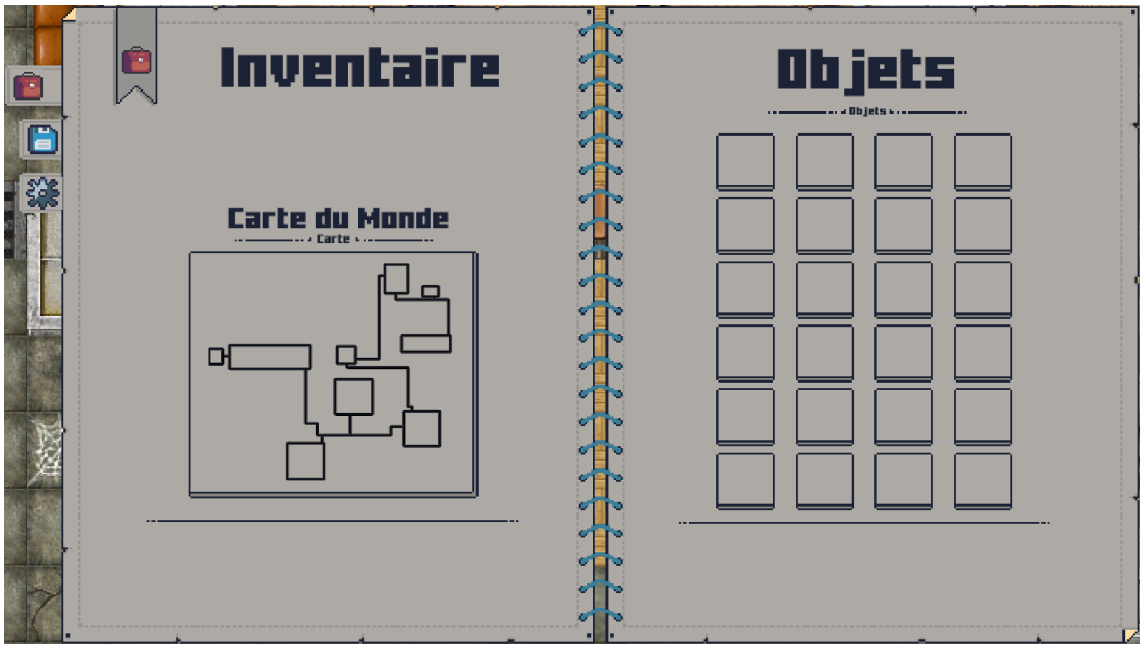

Figure 7: L'interface de l'inventaire

Le format carnet pour l'inventaire n'est pas seulement esthétique, mais aussi pratique. Il permet de regrouper les informations essentielles dans un format facilement accessible, augmentant ainsi la lisibilité et la facilité d'utilisation. L'affichage direct de la carte aide à maintenir l'orientation spatiale du joueur, un point crucial pour éviter les frustrations liées à la navigation.

2.1.3 Les menus

Nous avons effectué un changement radical du menu d'accueil du jeu.

En effet, nous n'étions pas satisfaits de ce menu qui n'était pas assez original, immersif, et thématique. De plus, il n'allait pas avec les autres designs du jeu, comme l'inventaire, qui nous plaisait bien plus.

Le changement de design du menu principal d'une présentation classique à une forme de carnet a donc été fait. Le choix du menu sous forme de carnet renforce l'immersion, donnant au joueur l'impression de consulter un véritable journal personnel. Inspiré directement par des jeux d'enquête et d'horreur classiques, ce choix de design offre une navigation intuitive et familière, en cohérence avec le thème narratif du jeu.

Chaque icône du menu est conçue pour être immédiatement reconnaissable, simplifiant la navigation et les interactions dans le jeu. Par exemple, la roue dentée représente les paramètres, permettant aux joueurs de comprendre rapidement leurs fonctionnalités.

Nous avons également commencé à implémenter les fonctionnalités audio afin de finaliser le menu, et de rendre tous les paramètres fonctionnels.

Figure 8: L'ancien menu

Figure 9: Le nouveau menu

Figure 10: La section des paramètres

Figure 11: Les options du mode multijoueur

2.2 Enigmes

2.2.1 Template

Nous avons commencé par réaliser le template de l'UI de l'énigme. Il est composé de la partie interface qui revient systématiquement. Dans Unity, il est constitué d'éléments UI et TextMeshPro.

Lorsqu'une énigme est lancée, une page cinématique affiche le numéro et le nom de l'énigme.

Figure 12: Affichage du numéro et du nom de l'énigme pendant 3s

Trois secondes plus tard, elle disparaît, laissant place à un écran divisé en trois zones:

- **Un header**, en haut, affichant le titre et le numéro de l'énigme.
- **La zone d'explication**, à droite, contenant le descriptif de l'énigme.
- **La zone principale de l'énigme**, à gauche, où se déroule l'interaction.

L'écran comporte également trois boutons : **Valider**, **Quitter** et **Notes**.

La partie **Notes** était une fonctionnalité essentielle pour nous, car elle permet au joueur de prendre des notes librement, comme avec un crayon sur une feuille. Ces notes sont conservées durant toute la durée de l'énigme, même si le panneau est fermé puis rouvert. Un bouton permet d'effacer les notes pour repartir de zéro.

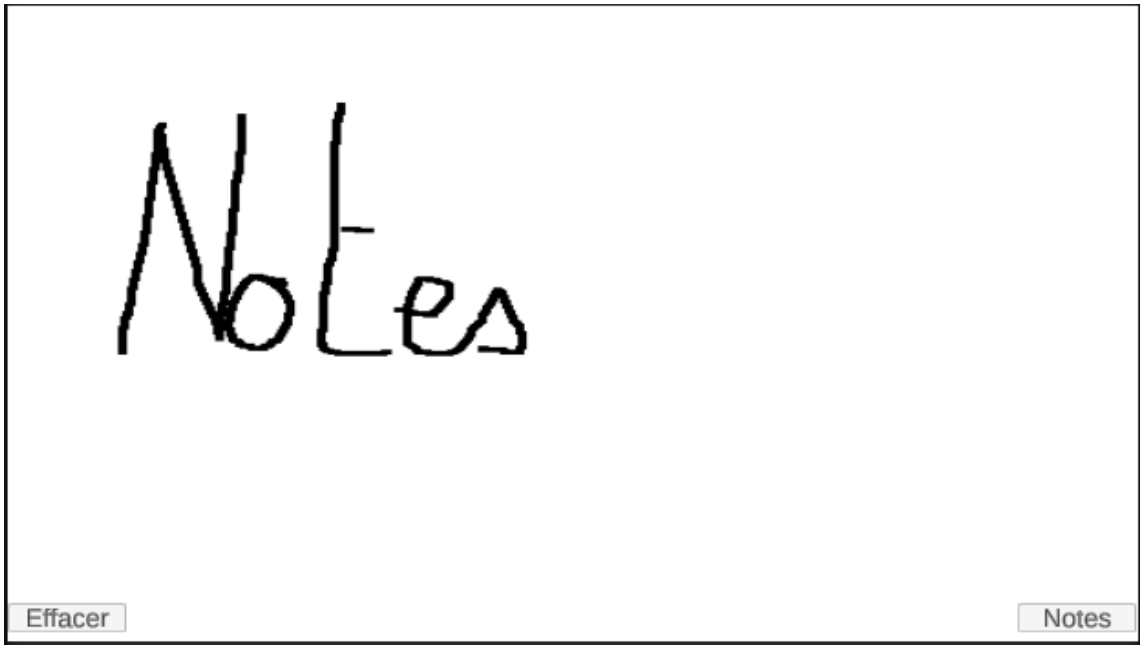

Figure 13: Interface de prise de notes

Afin de réutiliser ce template facilement, nous avons créé un **prefab Unity**, rendant la réutilisation de la partie graphique triviale.

2.2.2 Classe abstraite

Nous avons associé au prefab la création d'une classe abstraite `Enigme` en C#.

Lorsqu'une énigme est chargée, tous les éléments textuels sont placés dynamiquement par le code. L'utilisation d'une classe abstraite nous permet de concevoir facilement différents types d'énigmes avec leurs propres mécaniques.

Voici un extrait de code de la classe `Enigme` :

```
1  //Dependance :
2  using UnityEngine;
3
4  public abstract class Enigme
5  {
6      // Variable :
7      public string s_Title;
8      public string s_Explanations;
9      public int i_Number;
10     public bool b_finished = false;
11
12     protected GameObject PlayField;
13
14     // Constructeur de la classe abstraite Enigme
15     public Enigme(string s_title, string s_explanations,
           int i_number)
16     {
17         s_Title = s_title;
18         s_Explanations = s_explanations;
19         i_Number = i_number;
20         PlayField = EnigmeManager.Instance.PlayField;
21     }
22
23     //Fonctions de la classe abstraite Enigme :
24     public abstract void Initialize();
25
26     public abstract bool Check();
27
28 }
```

Dans la classe énigme deux méthodes sont définies, elles seront "écrasées" dans les classes héritières. La première, Initialize, s'assure d'initialiser les différents champs ainsi que de créer dynamiquement la zone de jeu. La seconde, Check, permet de vérifier l'énigme.

2.2.3 Types d'énigmes à ce jour

Nous avons remarqué que les énigmes sont soit simples à concevoir mais complexes à implémenter, soit l'inverse. Cependant, nous ne voulons pas revoir nos ambitions à la baisse : le jeu proposera donc plusieurs mécaniques d'énigmes différentes dont deux sont présentées ci-dessous.

Actuellement, nous avons déjà un type d'énigme appelé **ClassicText**. Son fonctionnement est simple : les explications présentent une énigme sous forme de texte, et le joueur doit entrer la réponse dans une boîte de texte. En cliquant sur Valider, la réponse entrée est comparée à la réponse attendue. La zone de réponse change de couleur en fonction de la réponse : elle devient verte pour une réponse juste et violette pour une réponse fausse.

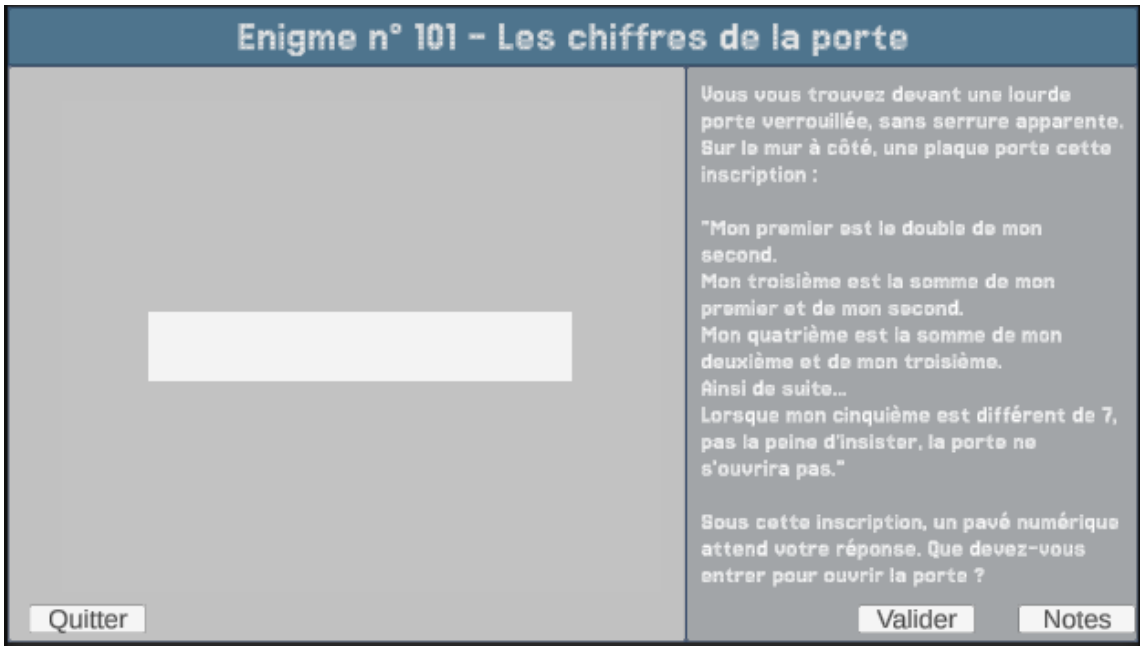

Figure 14: Interface d'une énigme de type ClassicText. D'un type d'énigme à un autre, seule la partie gauche de l'écran différe

Voici à titre d'exemple le code de la classe ClassicText.

```
using System;
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class T_ClassicText : Enigme
{
    private InputField inputField;
    private string _expected;

    public T_ClassicText(string title, string explanations,
        int number, string expected) : base(title,
       explanations,
        number)
    {
        _expected = expected;
    }

    void CreateInputField()
    {
        GameObject inputObject = new GameObject("
           EnigmeInput");
        inputObject.transform.SetParent(PlayField.transform
           , false);

        Image bgImage = inputObject.AddComponent<Image>();
        bgImage.color = new Color(1, 1, 1, 0.8f);

        inputField = inputObject.AddComponent<InputField>()
           ;

        GameObject textObject = new GameObject("Text");
        textObject.transform.SetParent(inputObject.
           transform, false);

        Text inputText = textObject.AddComponent<Text>();
        inputText.font = Resources.GetBuiltinResource<Font
           >("LegacyRuntime.ttf");
        inputText.text = "";
        inputText.color = Color.black;
        inputText.alignment = TextAnchor.MiddleLeft;

        RectTransform textRect = textObject.GetComponent<
           RectTransform>();
```

```
37          textRect.sizeDelta = new Vector2(280, 40);
38          textRect.anchoredPosition = Vector2.zero;
39
40          inputField.textComponent = inputText;
41
42          RectTransform rectTransform = inputObject.
               GetComponent<RectTransform>();
43          rectTransform.sizeDelta = new Vector2(300, 50);
44          rectTransform.anchoredPosition = Vector2.zero;
45
46          Debug.Log("InputField cree dans le PlayField !");
47      }
48
49      public override void Initialize()
50      {
51          PlayField = GameObject.Find("PlayField");
52
53          if (PlayField == null)
54          {
55              Debug.LogError("PlayField introuvable !");
56              return;
57          }
58
59          CreateInputField();
60      }
61
62
63      public override bool Check()
64      {
65          if (PlayField == null)
66          {
67              throw new Exception("Il n'est pas possible de
                   valider l'enigme si le PlayField n'a pas
                   charge correctement");
68          }
69
70          bool result = _expected.ToLower() == inputField.
               text.ToLower();
71          Color newColor;
72          if (result)
73              ColorUtility.TryParseHtmlString("#B2BEB5", out
                   newColor); // Vert si jamais c'est juste
74          else
75              ColorUtility.TryParseHtmlString("#8C92AC", out
                   newColor); // Violet si jamais c'est pas
                   juste
```

```
76
77          inputField.image.color = newColor;
78
79
80          Debug.Log("La reponse est " + result);
81          return result;
82      }
83  }
```

De plus, nous sommes en train de travailler sur un type d'énigme basé sur **Reversi**.

Le **Reversi** est un jeu de stratégie qui se joue sur une grille composée de cases pouvant être dans deux états différents. L'objectif du joueur est de retourner les cases de manière à ce qu'elles soient toutes orientées dans le même sens.

Figure 15: Schéma du jeu Reversi

Le principe est simple : lorsqu'un joueur clique sur une case, celle-ci, ainsi que ses voisines changent d'état. Cela peut entraîner des réactions en chaîne qui modifient une partie du plateau de jeu.

Ce type d'énigme repose donc sur l'anticipation des effets de chaque action du joueur. Trouver la bonne séquence de coups pour parvenir à un plateau entièrement homogène est le défi à relever.

Dans notre implémentation, nous définissons plusieurs niveaux de difficulté pour pouvoir réutiliser plusieurs fois le concept en jouant sur le pattern de départ.

2.2.4 Gestion des énigmes en arrière plan

Il est important de noter que l'entièreté de l'initialisation des différents éléments, y compris la zone de jeu, est gérée directement depuis un script. Il était important, pour simplifier la création d'autres types d'énigmes, d'automatiser le processus de la sorte.

Les énigmes sont stockées dans une liste C# dans un fichier qui s'occupe de la gestion des énigmes. Il permet de :

- Instancier les énigmes et les mettre dans la liste
- Démarrer une énigme à partir de son numéro

L'avantage de cette méthode d'implémentation est qu'il suffit de programmer chaque type d'énigme dans une classe héritière et ensuite il suffit de créer une nouvelle instance pour ajouter une énigme qu'on a conçu sur du brouillon. Par exemple, pour créer une nouvelle énigme ClassicText, il suffit d'utiliser :

```
1    new T_ClassicText(string title, string explanations, int
         number, string expected)
```

en remplaçant respectivement les arguments par le titre, les explications, le numéro de l'énigme et la réponse attendue.

En interne, nous utilisons aussi un tableur avec les mêmes informations. Le numéro de l'énigme permet aussi de les ranger et de les identifier facilement : les 0xx seront des énigmes de fin de niveau, les 1xx sont des ClassicText, les 2xx seront des Reversi et les 9xx sont des énigmes de test. Cette notation va évoluer au fur et à mesure de la création des types d'énigmes restants.

2.3 Développement et programmation

En premier lieu, nous avons tenu compte des commentaires constructifs donnés durant la première soutenance, en corrigeant, par exemple, la vitesse du joueur, et en réglant les problèmes liés aux FPS, ce qui a, par la même occasion, réglé le problème des "à-coups" des déplacements du joueur vu en soutenance.

2.3.1 Mode multijoueur

Le mode multijoueur est une facette importante de notre jeu, et son développement s'étale tout au long du projet.

Notre objectif principal était de stabiliser la connexion entre les joueurs et d'assurer une synchronisation fluide des mouvements tout en garantissant une expérience immersive et individualisée pour chaque participant.

Synchronisation des mouvements

L'un des défis majeurs du multijoueur LAN était d'assurer que chaque joueur puisse voir les déplacements de l'autre en temps réel sans désynchronisation. Grâce à Netcode for GameObjects et à l'utilisation du NetworkTransform , nous avons désormais réussi à synchroniser les mouvements des joueurs de manière efficace. Désormais :

- Chaque joueur peut se déplacer indépendamment, et ses actions sont bien répliquées sur toutes les instances du jeu.
- Nous avons implémenté une logique réseau qui permet au client d'envoyer des requêtes de déplacement au serveur, garantissant ainsi une gestion centralisée des positions et évitant les comportements incohérents.
- La latence reste faible, offrant une expérience fluide et immersive.

Connexion des joueurs

L'établissement de la connexion LAN entre les joueurs est maintenant fonctionnel. Nous avons confirmé que :

- L'hôte peut créer une session de jeu en cliquant sur **Héberger**.
- Un client peut rejoindre la partie en sélectionnant **Rejoindre**, ce qui lui permet de se connecter automatiquement à l'hôte et d'apparaître dans l'environnement de jeu.
- Chaque joueur se voit attribuer un `NetworkObject`, assurant une réplication correcte des entités dans l'univers du jeu.

Cette étape étant validée, nous nous concentrons maintenant sur l'amélioration de la gestion des permissions et de l'autorité réseau pour garantir une communication optimale entre client et serveur.

Désynchronisation des caméras (en cours de développement)

Un autre point essentiel pour l'immersion est la gestion des caméras. Auparavant, tous les joueurs partageaient la même vue, ce qui posait un problème d'individualisation de l'expérience. Pour résoudre cela, nous avons commencé à implémenter une séparation des caméras afin que :

- Chaque joueur dispose de sa propre caméra attachée à son personnage.
- Les autres joueurs ne puissent pas voir à travers la caméra d'un autre joueur.
- L'assignation des caméras soit automatique en fonction de l'identité du joueur sur le réseau (via `IsOwner` dans Netcode).

Pour l'instant, cette fonctionnalité est encore en cours de patch et nécessite quelques ajustements pour assurer un bon fonctionnement, notamment en empêchant toute interférence entre les instances.

Prochaines étapes

Avec ces avancées, notre feuille de route inclut maintenant :

- **La finalisation de la désynchronisation des caméras** pour garantir que chaque joueur ait une vue unique adaptée à son personnage.
- **La correction des derniers problèmes de réplication des données client-serveur**, notamment en affinant les permissions réseau pour éviter d'éventuelles latences ou incohérences.
- **L'optimisation de la synchronisation des mouvements** pour éviter d'éventuels bugs liés aux collisions ou aux décalages mineurs dans l'environnement.
- **L'intégration d'un système de code** pour séparer les parties.

Nous sommes sur la bonne voie pour finaliser cette fonctionnalité et offrir une expérience LAN fluide et immersive dans Syndrome Fear .

2.3.2 Menu

L'implémentation des menus s'est fait en plusieurs temps.
Nous avons commencé par la création du menu multijoueur en ajoutant trois options principales : le mode multijoueur, le mode solo et l'accès aux options. Par la suite, nous avons développé le menu des options, qui comprend plusieurs fonctionnalités : un bouton pour retourner au menu principal, un lien vers un site web, un bouton pour quitter le jeu, ainsi qu'une gestion des paramètres audio. En parallèle de ce menu, nous avons intégré les effets sonores et commencé la gestion des sliders audio. Nous avons aussi travaillé sur le menu d'échappement, en ajoutant un changement d'image dynamique lors de l'interaction avec les boutons situés en haut à gauche de l'écran.

En complément, nous avons mis en place un système d'inventaire permettant la gestion et l'affichage des objets récupérés par le joueur.

2.3.3 Téléportations et collisions

Notre jeu nécessitait un système de collisions, afin d'éviter que les joueurs puissent traverser les objets. Nous avons créé nos collisions sur les cartes du jeu avec le système de collisions d'Unity.

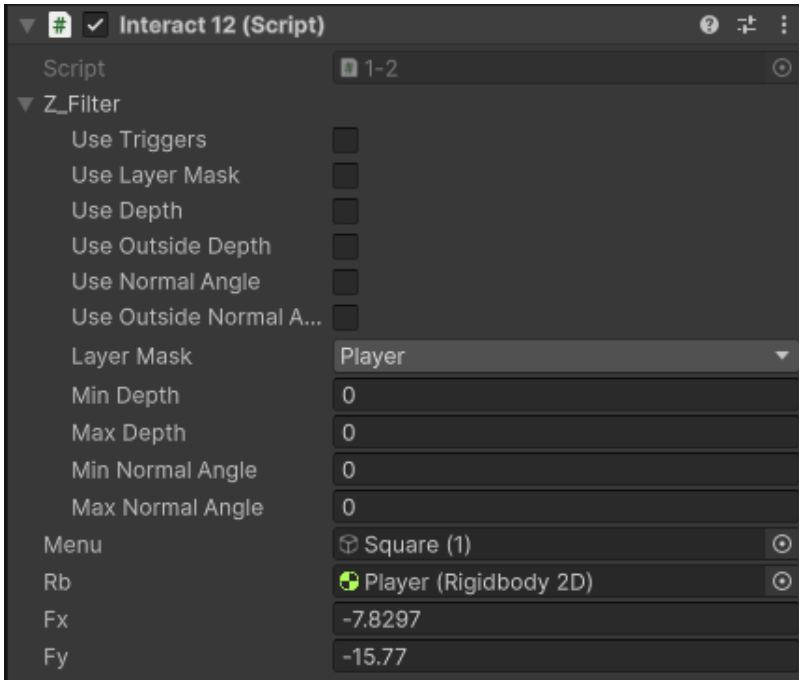

Figure 16: Interface de création d'une collision

De plus, ce système de collisions nous permet de créer un système de téléportations entre les différents niveaux : lorsque le joueur entre en collision avec la CollisionBox d'un GameObject défini, celui-ci exécute un script présent ci-dessous qui "téléporte" au niveau suivant, c'est-à-dire aux coordonnées indiquées en paramètres.

```
// Gestion des teleportations entre chaque niveau
using UnityEngine;
using System.Collections.Generic;

public class Interact12 : MonoBehaviour
{
    // Variable :
    private Collider2D z_Collider;
    [SerializeField]
    private ContactFilter2D z_Filter;
    private List<Collider2D> z_CollidedObjects =  new List<
       Collider2D>(1);
    public GameObject menu;
    public Rigidbody2D rb;
    public float fx;
    public float fy;
    Vector2 vmoov;

    // Initialise z_Collider comme etant le Collider de l'
       objet (le teleporteur).
    private void Start()
    {
        z_Collider = GetComponent<Collider2D>();
    }

    // Repetition :
    private void Update()
    {
        // Recupere les objets en collision avec le
           teleporteur en fonction des filtres rentres dans
            z_Filter et stocke les objets dans
           z_CollidedObjects.
        z_Collider.Overlap(z_Filter, z_CollidedObjects);
        // Parcourt cette liste, afin de recuperer le 1er
           element (notre joueur, theoriquement, au vu des
           filtres rentres), et le deplace aux coordonnees
           voulues.
        foreach(var z in z_CollidedObjects)
        {
            menu.SetActive(false);
            rb.position = new Vector3(fx, fy);
            break;
        }
    }
}
```

2.3.4 Le site internet

Notre site a été mis à jour, afin d'ajouter les différents rapports de soutenance et la version exécutable de notre jeu.

De plus, une version plus responsive est maintenant disponible, pour les utilisateurs mobiles.

Le site Internet est toujours hébergé sur nos propres infrastructures, et le code source est disponible sur GitLab à l'URL suivante : https://gitlab.cri.epita.fr/thibault.ponsot/syndrome_fear_website. Une connexion via la Forge est requise pour accéder au code source.

2.3.5 La base de données

Lors de la création du cahier des charges en octobre, nous avons dû évaluer les technologies à utiliser pour réaliser ce projet. Nous avions choisi d'utiliser une base de données. Cependant, l'expérience nous a appris deux choses : qu'il n'était pas toujours forcément nécessaire de déployer une base de données et que lorsque ce choix s'imposait, il était nécessaire de la créer et de la déployer en parallèle, et non en amont.

Nous avons donc pris la décision de ne pas utiliser de base de données jusqu'à présent, contrairement à ce qui avait été initialement mentionné dans le diagramme de Gantt, sans se fermer la porte à la possibilité d'en implémenter une si nécessaire, dans le cadre du mode multijoueur, par exemple.

3 Planning prévisionnel

Cette dernière partie est la finalisation de notre projet. Voici les tâches que nous allons effectuer dans les semaines à venir :

- La création de l'intelligence artificielle de pathfinding
- La fin de la réalisation des énigmes
- La fin de l'implémentation du mode multijoueur
- La réalisation du didacticiel et le placement des énigmes dans l'espace pour induire un déroulement de l'histoire
- La création d'un installateur et d'un désinstallateur
- La rédaction du manuel, de la documentation
- La réalisation des animations, de l'audio et des effets sonores.

4 Conclusion

Le temps s'écoule rapidement, et nous éprouvons une grande satisfaction ainsi qu'une immense fierté en voyant notre jeu se concrétiser et évoluer.

Nous abordons à présent la phase finale de développement, ultime étape avant la livraison du produit final.

Notre ambition est de proposer un produit d'une qualité irréprochable, et notre équipe mobilise tous les moyens nécessaires pour atteindre cet objectif.