– S P E C T R A L –

Rapport de fin de projet

Vincent Boyer (BOYV23119600)

Christopher Leret (LERC12069705)

Charles Mécheriki (MECC17029701)

Vincent Rabier (RABV22059708)

Concept du jeu

Motivations

Spectral a été pensé comme un *shooter* délibérément non conventionnel, inspiré du genre *tower defense* et à la direction artistique distinctive.

Concevoir un *shooter* avait pour avantage de ne pas s'appuyer sur un dispositif complexe, tant sur le plan des interactions que des animations.
L'idée était d'imaginer un jeu original, qui mette en place des personnages et un décor inattendu, comme a su le faire **The Binding of Isaac** (2011).
Le choix de proposer un décor illustré en 2D, à la manière de **Postal Redux** (2016), offrait l'avantage d'être facilement réalisable, tout en gardant la possibilité de produire un environnement détaillé, avec une identité artistique forte.

L'univers gothique présenté dans Spectral s'inscrit dans une thématique et une esthétique facilement identifiable, loin des archétypes du genre.
Le joueur est immergé dans un monde fantastique au bestiaire original, où les ennemis et les objectifs à défendre sont mis en évidence par des couleurs contrastés.
À titre d'exemple, **Tesla Versus Lovecraft** (2018) a su mettre à profit une esthétique fantastique dans le contexte d'un *shooter*.

Gameplay

Le joueur incarne Prudence, domestique au manoir d'un riche collectionneur.
Depuis son réveil, d'étranges phénomènes ont lieu à l'intérieur du manoir et à ses alentours.
En effet, une mystérieuse lanterne semble provoquer l'assaut de spectres - appelés cauchemars - venus d'une autre dimension par des portails - appelés rêveurs.
Seule la lumière de la lanterne, qui est la cible des cauchemars, peut anéantir les rêveurs lorsqu'elle est mise à leur contact.
Pour cela, Prudence doit prendre et déplacer la lanterne.
Prudence peut également repousser les cauchemars avec son arbalète, mais tant que des rêveurs sont actifs, de nouvelles vagues de cauchemars apparaîtront.

Si les cauchemars parviennent à éteindre la lumière de la lanterne, la partie est perdue.

Le jeu supporte un second joueur, permettant de jouer en mode coopératif. En fonction de la quantité de cauchemars tués, chaque joueur se voit attribuer un score.

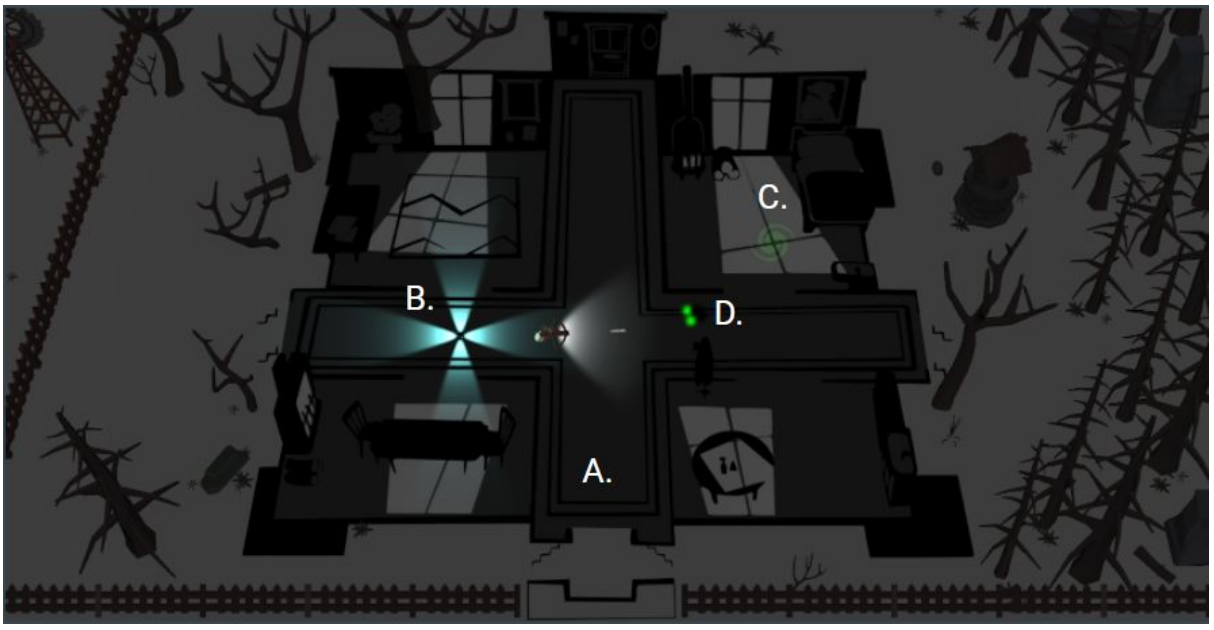


Figure 1. Éléments du Gameplay de Spectral
A. Le manoir. B. La lanterne. C. Un rêveur. D. Un cauchemar

Interface de jeu et menu

Spectral propose 1 scène de jeu et 4 scènes de menu, qui sont respectivement :

- MainScene, la scène de jeu, où on retrouve le personnage du joueur, le manoir, etc. ;
- MainMenu, le menu principal, permettant de lancer une partie ou de consulter les contrôles du jeu ;
- WinMenu, le menu affiché en fin de partie, en cas de victoire ;
- LoseMenu, le menu affiché en fin de partie ne cas de défaite ;
- ControlsMenu, le menu présentant les différents contrôles au joueur.

Dans la scène de jeu, pour inciter à l'immersion, aucun HUD n'a été implémenté. La plupart des informations relatives à l'avancement de la partie sont facilement déductibles par le joueur, notamment les points de vie de la lanterne, qui se reflètent directement par la quantité de lumière qui en est émise.

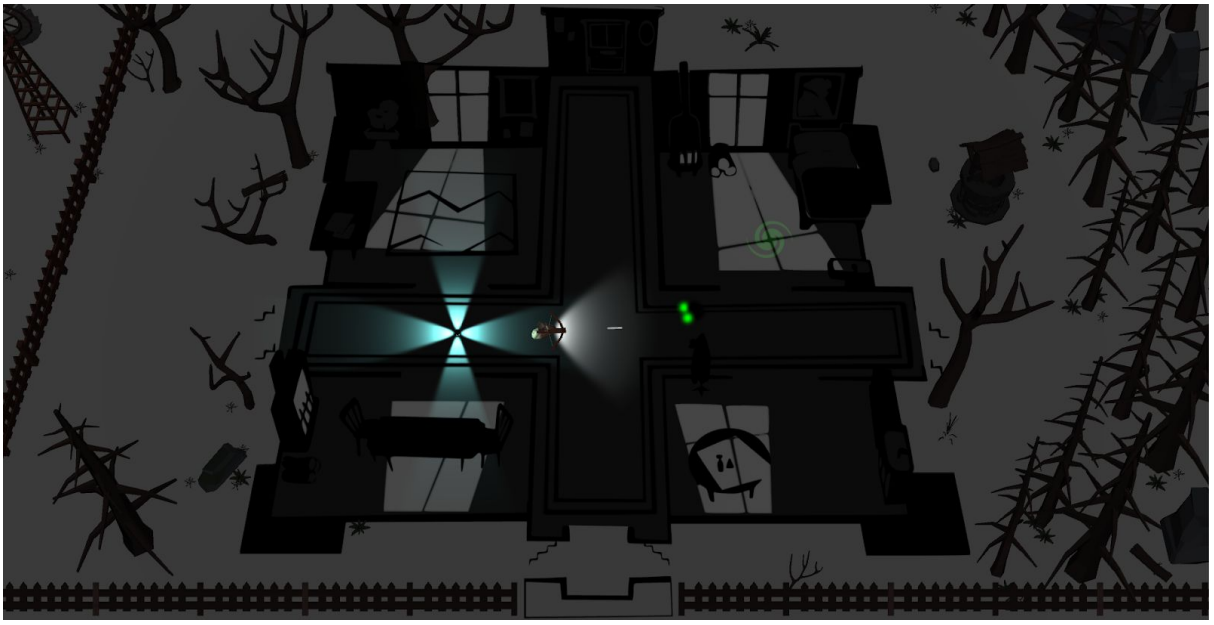


Figure 2. Scène de jeu

Figure 3. Menu principal

Figure 4. Menu de victoire

Figure 5. Menu de défaite

Figure 7. Menu des contrôles

Architecture logiciel du jeu

Présentation de l'architecture du jeu

Le jeu repose sur 5 entités principales :

- Le *GameManager*, reponsable de la résolution du jeu et des évènements ;
- Les *Dreamers*, d'où proviennent les cauchemars ;
- La *Lantern*, élément central du jeu ;
- Les *Nightmares*, qui attaquent la lanterne.

Pour chacune de ces entités, des scripts spécifiques, responsables des différentes interactions, sont rassemblées au sein d'un script général, portant le nom de l'entité.
Ajouté à ces scripts d'interactions sont présents des scripts généraux (*Health*, *Damage*, *Loot*, *Score*), et des scripts utilitaires (*EventManager*, *Event*, *Timer*, *AudioUtility*, *TriggerDelegate*, *Tags*, *Layers*, *Scenes*, etc.) pour répondre à des problématiques récurrentes.

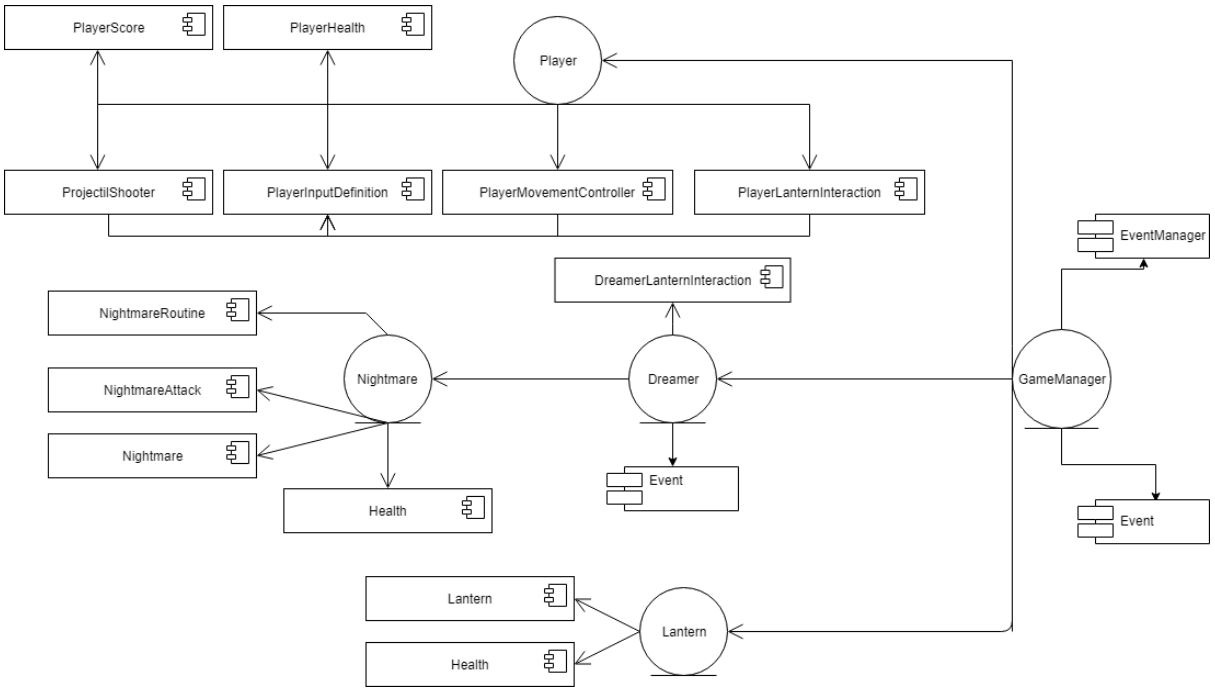


Figure 8. Architecture simplifiée de Spectral

Interaction du joueur

Les interactions du joueur se font grâce au inputs rentrés par l'utilisateur à la manette ou au clavier / souris. Pour simplifier la déclaration des entrées nous avons décidé de créer un composant permettant de réaliser ce comportement : le composant *PlayerInputDefinition*. Ce composant est responsable de définir les input utilisé par le joueur en fonction de si il est le premier, second, troisième ou quatrième joueur.

Pour réaliser ce comportement, nous avons utilisé le système d'input de Unity. Chaque joueur possède des actions qui lui sont propre. Par exemple l'action "*Fire*" (utilisé pour déclencher le tir d'un projectile) est définie plusieurs fois avec un identifiant dans le nom : "Fire1", "Fire2" etc. Ce comportement n'est pas optimal mais le système d'Input n'est pas assez modulable pour faire autrement. Unity va changer la gestion des entrées et le nouveau système est disponible en version beta mais nous n'avons pas voulu prendre le risque d'utiliser un système nouveau et pas en version finale.

Tout les script ayant besoin de connaître l'état d'une touche du clavier ont donc une référence vers le script *PlayerInputDefinition*. C'est donc le cas des scripts *PlayerMovementController* qui gère le déplacement du joueur, *ProjectileShooter* qui gère le tire de projectile et *PlayerLanternInteraction* qui lui est utilisé pour prendre et déposer la lanterne.
Afin de faciliter leur association à un GameObject, tous les scripts associés au bon fonctionnement du personnage du joueur sont regroupé dans le script *Player*.

Gestion de la santé et des dégâts

Dans le jeu, nous avons plusieurs type d'entités pouvant prendre des dégâts : les joueurs, la lanterne et les monstres. Nous avons donc créé un script permettant de gérer la vie et la prise de dégâts de ces types d'entités sans prendre en compte leurs types : le composant *Health***.** Ce composant stocke la vie de l'entité mais aussi des listeners (*delegate* en C#) permettant de notifier les autres composants lors de la prise de dégâts. Ils sont notamment utilisés pour jouer des effets sonore différents en fonction de l'entité qui a reçu des dégâts.

Gestion des dégâts

Pour prendre des dégâts, nous avons opté pour une solution comportant plusieurs composants. Lors de l'action d'attaque d'un joueur, le composant *ProjectileShooter* va faire apparaître un gameobject en lui donnant des données de dégâts. Ces données comportent une référence vers le joueur ayant tiré le projectile ainsi que les dégâts à infliger lors de l'impacte.

Spawner d'ennemis

Les spawners d'ennemis, ou *Dreamers*, sont des entités qui permettent de créer des instances d'ennemis en suivant un pattern donné. Nous pouvons configurer pour chaque spawner le type d'entités qu'ils vont faire spawner. La zone de spawn est configurable et les monstres, ou *Nightmares*, pourront donc apparaître à une position aléatoire dans cette zone.

Gestion de l'état du jeu

Pour gérer l'état interne du jeu nous avons créer un *GameManager*. Ce composant possède une référence vers toute les instances de *Player* ainsi que vers toutes les instances de *Dreamers*. Il possède également une référence vers la lanterne.

Dès lors que la lanterne a perdu toute sa santé, la partie est perdue. En revanche, si tous les *Dreamers* sont désactivés avant que la lanterne s'éteigne, le joueur est victorieux.

Exemples de code

Exemple d'interactions : PlayerMovementController

Pour vous illustrer le code, nous allons vous donner l'exemple de la classe *PlayerMovementController*. Ce classe représente bien ce que nous avons codé dans toutes les autres classes.

Nous allons commencer par la méthode Awake de Unity. Ce méthode nous permet de récupérer les composants de base qui seront utilisés dans cette classe. Ici, le descripteur des inputs, le rigidbody et le contrôleur d'animation. Les autres variables sont des optimisations ou des construction de base.

```
void Awake()
{
    m_playerInputDefinition = GetComponent<PlayerInputDefinition>();
    m_rigidbody = GetComponent<Rigidbody>();
    m_animator = GetComponent<Animator>();

    m_floorMask = Layers.Floor;
    m_camera = Camera.main;
}
```

Figure 9. Implémentation de la méthode Awake() dans le MonoBehaviour Player

La seconde méthode que nous allons donner est la méthode permettant de déplacer le joueur. Cette méthode est invoquée à un interval régulier que nous avons fixé.

```
void Move()
{
    // Stores the input axes.
    float horizontalInput = Input.GetAxis(m_playerInputDefinition.m_inputMovementHorizontal);
    float verticalInput = Input.GetAxis(m_playerInputDefinition.m_inputMovementVertical);

    bool isWalking = horizontalInput != 0 || verticalInput != 0;
    m_animator.SetBool("IsWalking", isWalking);
    m_animator.SetFloat("WalkHorizontal", horizontalInput);
    m_animator.SetFloat("WalkVertical", verticalInput);

    PlayFootstepSoundEffect(isWalking);

    horizontalInput *= m_movementSpeed;
    verticalInput *= m_movementSpeed;

    // Computes the movement.
    Vector3 movementVector = Vector3.forward * verticalInput + Vector3.right * horizontalInput;

    // Moves using Lerp.
    transform.position = Vector3.Lerp(transform.position, transform.position + movementVector,
        Time.deltaTime);
}
```

Figure 10. Méthode Move() du MonoBehaviour Player

Comme vous pouvez le voir, cette méthode permet de récupérer les inputs horizontal et vertical. Nous allons ensuite donner ces information au contrôleur d'animation.
Cette méthode va également jouer les son de mouvement via une audio source présente dans le joueur.
Finalement nous allons déplacer le joueur en fonction du temps de la frame pour ne linéariser le déplacement.

Exemple de coroutine : Dreamer

Les *Dreamers* sont les entités responsables des vagues d'ennemis, et par conséquent de la difficulté du jeu. Ils ont donc été conçu pour être hautement configurables.
Ainsi, il est possible de choisir les différents délais séparant les vagues, les cauchemars de la chaque vague, ou encore l'événement qui déclenche l'activation du *Dreamer*.

```
#region Waves
[Header("Waves")]

[SerializeField]
Wave[] m_waves;

[Serializable]
public struct Wave
{
    public float FirstSpawnDelay;
    public GameObject NightmarePrefab;
    public int Spawns;
    public float SpawnsDelay;
    public int NightmaresPerSpawn;
}

[SerializeField] Event m_activateOnEvent = Event.Void;
[SerializeField] Event m_onDeactivateTriggerEvent = Event.Void;
[SerializeField] bool m_selfDeactivateOnLastWave = false;
#endregion
```

Figure 11. Attributs du MonoBehaviour Dreamer

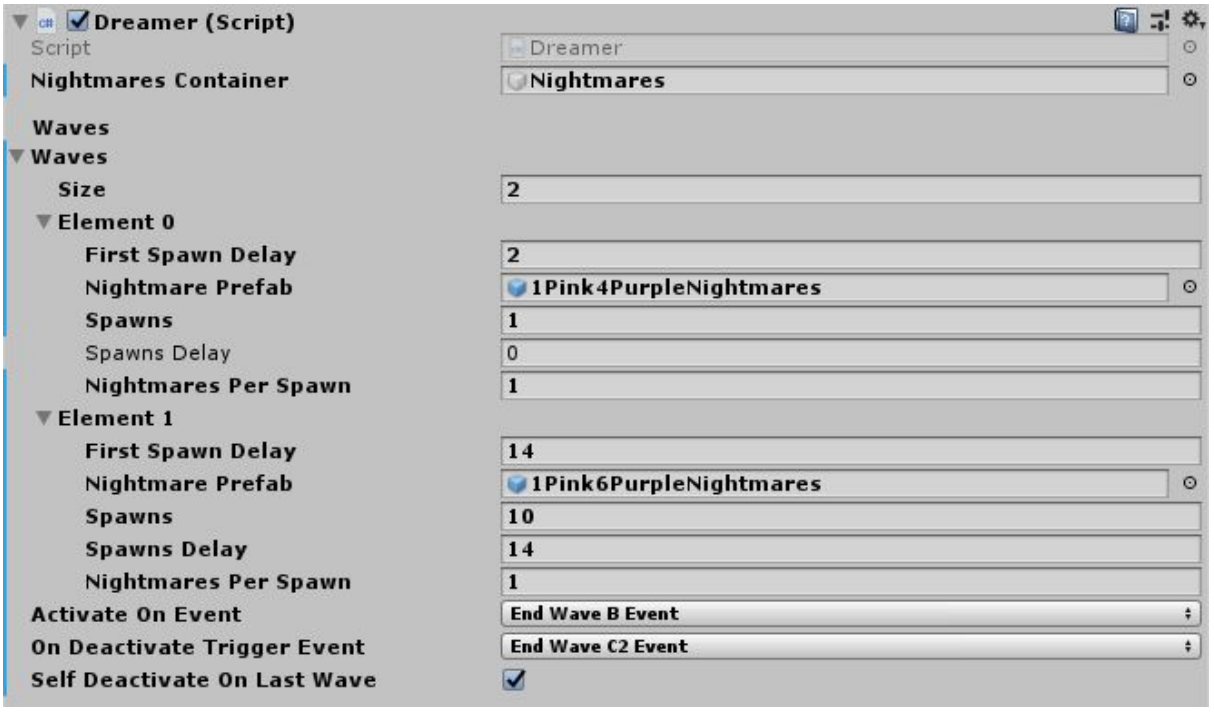


Figure 12. Interface du MonoBehaviour Dreamer dans l'éditeur

Pour déclencher les waves successives d'ennemis, une coroutine est utilisée. Celle-ci gère le suivi des délais séparant les vagues d'ennemis.
Elle est en fonctionnement de l'activation du *Dreamer* jusqu'à sa désactivation.

```
IEnumerator WavesCoroutine()
{
    foreach (Wave wave in m_waves)
    {
        // FirstSpawnDelay
        float waveStartingTime = Time.time + wave.FirstSpawnDelay;

        while (Time.time < waveStartingTime)
        {
            RotationTick(-5);

            yield return new WaitForFixedUpdate();
        }

        for (int i = 1; i <= wave.Spawns; i++)
        {
            while (m_light.intensity < m_lightDefaultIntensity)
            {
                m_light.intensity += m_lightIntensityVariation;

                RotationTick(5.5f);

                yield return new WaitForFixedUpdate();
            }

            // Spawns
            if (!IsInteractingWithLantern)
            {
                for (int j = 1; j <= wave.NightmaresPerSpawn; j++)
                {
                    SpawnNightmare(wave.NightmarePrefab);
                }
            }

            while (m_light.intensity > m_lightMinIntensity)
            {
                m_light.intensity -= m_lightIntensityVariation * 1.7f;

                RotationTick(6);

                yield return new WaitForFixedUpdate();
            }
```

Figure 13. Coroutine WavesCoroutine()du MonoBehaviour Dreamer (début)

```
            // SpawnsDelay
            float nextSpawnTime = Time.time + wave.SpawnsDelay;

            if (i != wave.Spawns)
            {
                while (Time.time < nextSpawnTime)
                {
                    RotationTick(-5);

                    yield return new WaitForFixedUpdate();
                }
            }
        }
    }

    if (m_selfDeactivateOnLastWave)
    {
        SelfDeactivate();
    }

    while (true)
    {
        RotationTick(-5);

        yield return new WaitForFixedUpdate();
    }
}
```

Figure 14. Coroutine WavesCoroutine()du MonoBehaviour Dreamer (fin)

Ce genre de coroutine est extrêmement pratique, puisqu'elle permet de déclencher des méthodes de scripts de manière régulière, et se base sur des valeurs entrées dans l'éditeur (i.e par le level designer).

Choix de conception et technologies utilisées

Mixage audio

Vous avez pu constater que le joueur possédait plusieurs sources audio. Ces sources sont ont toutes une utilité. En effet, chaque clip audio, selon sa nature, est joué sur une source spécifique, elle-même associée à un mixer différent. Cela permet de déclencher des sons avec un volume adapté selon la situation. Nous avons par exemple spécifier à l'engine de baisser la musique d'ambiance lorsque des sons du joueur (tir, prise de dégâts) sont émis.

Lumières et cookies

Spectral repose sur une grande utilisation des composants *Light* de Unity. Ceux-ci sont très versatiles, puisqu'il est possible de leur associer une texture d'image, appelé cookie, qui est affecté par les opérations appliqués à la lumière (intensité, portée, angle de spot, transformation, etc.).
Ainsi, le motif des Rêveurs et les fausses lumières du Manoir sont des cookies de lumières, ce qui permet de les animer de manière fluide et joueur sur leur intégration au décor.

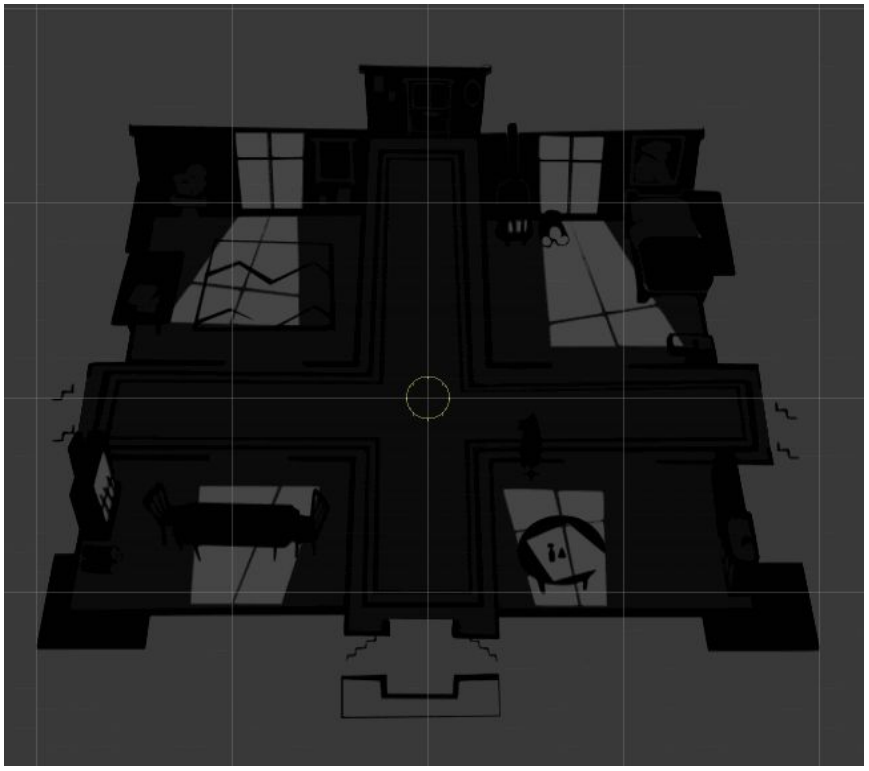


Figure 15. Illustration du potentiel des cookies sur le Manoir

Manoir 2D

Bien que les modèles du jeu soient 3D, l'environnement du manoir est quant à lui basé sur des illustrations.
Cela a permi d'obtenir un rendu avec une vraie personnalité et de

Le revers de la médaille étant que l'ensemble des collisions avec les modèles 3D du niveau doivent donc être simulés en utilisant des *Colliders* supplémentaires.

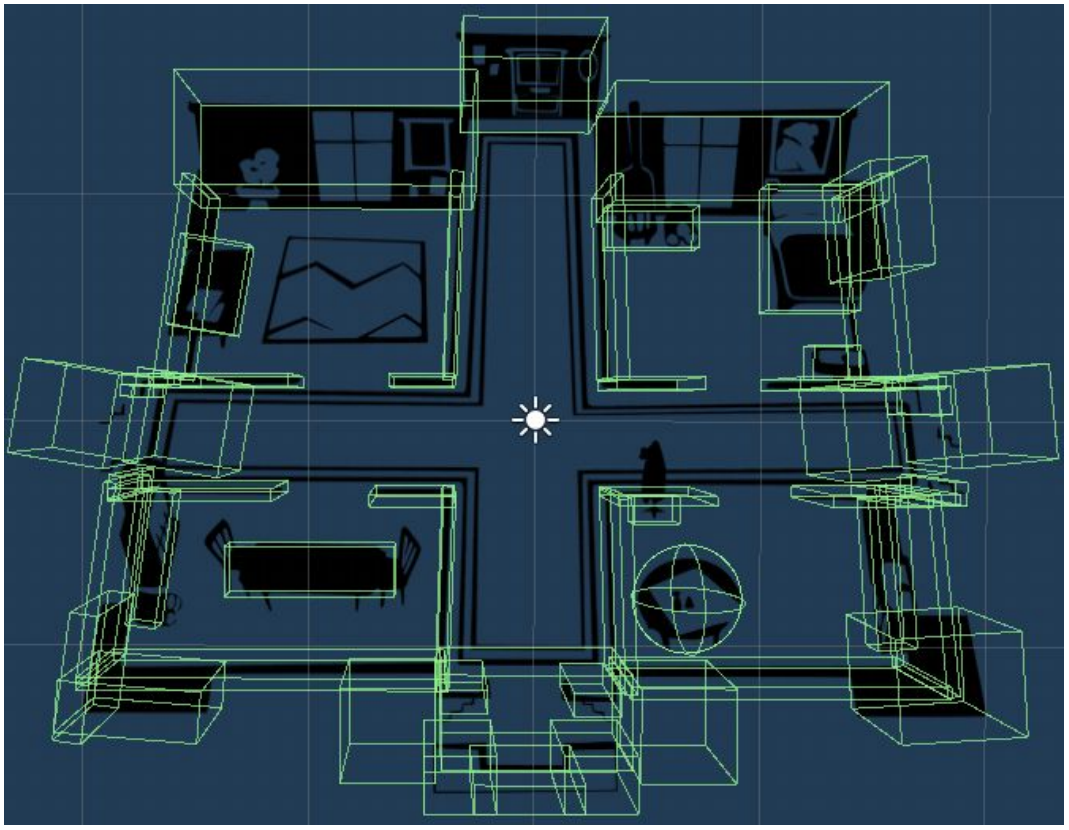


Figure 16. Les Colliders du Manoir

TextMeshPro

Pour les scènes de menu, *TextMeshPro*, un plugin Unity permettant de réaliser des interface contenant des options typographique plus poussées a été utilisé.
Il a notamment permis de faire des effets de *glow* au titre (cf. Figure 3. Menu principal).

Gestionnaire de source GIT LFS

Pour partager nos sources et nos assets, nous avons utilisé le gestionnaire de version GIT. Ce n'est pas le plus utilisé dans l'industrie (le plus utilisé étant Perforce) mais nous avons eu beaucoup d'échos de professionnel de l'industrie qui tente de se séparer de Perforce au profit de GIT car celui ci est plus performant. Le seul problème étant la gestion des assets. Pour gérer ce problème, nous avons fait attention à ne pas modifier les même scène au cours du développement. Nous avons aussi limité la modification des assets et des prefabs.

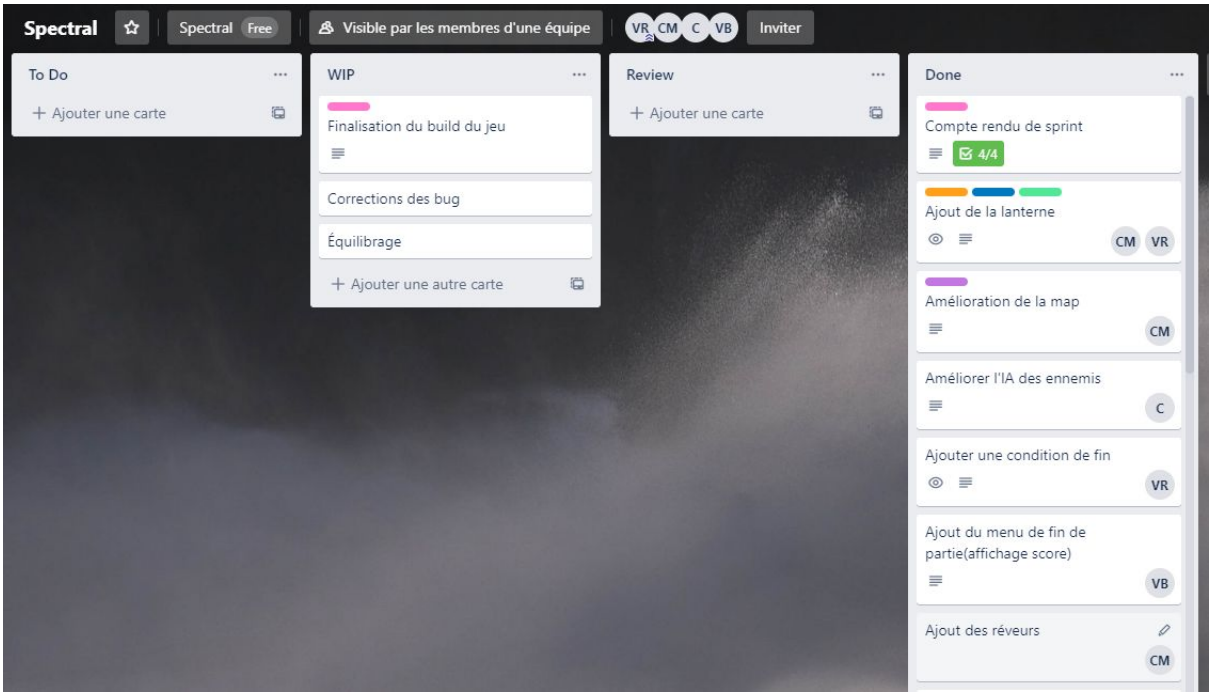
Nous avons utilisé l'extension LFS[1] de Git qui permet de stocker les fichiers lourds (audio, vidéos, assets 3D, etc.) d'une meilleure façon en les convertissant au format texte. Pour utiliser cette extension, nous avons utilisé le repository Git fournis par notre professeur de réseau Valère Plantevin, disponible ici.

[1] https://git-lfs.github.com/

Répartition des tâches

Pour réaliser ce projet nous nous somme répartie les tâches à réaliser durant un sprint au début de celui-ci. Vous pouvez trouver ci-joint le Trello que nous avons utilisé pour garder une trace du travail réalisé par les membres de l'équipe. Un exemple de Trello nous a été donné au début du projet. Cet exemple était beaucoup plus complet mais nous avions déjà commencé à remplir notre tableau nous avons donc choisi de continuer avec celui-ci.

Notre dashboard se compose de 4 colonnes. La "To Do" représente les tâches non assignées qu'il reste à accomplir. Cette colonne était remplie au début de chaque sprint par l'équipe en fonction de la prévision que nous avions réalisée ainsi que des remarques qui ont émergée durant les sprints précédent.
La colonne WIP représente le travail en cours de réalisation par les membres de l'équipe. Les tâches dans cette colonne sont assignées à une ou plusieurs personnes afin de garder une trace des personnes et de leurs travail.
Une fois une tâche accomplie, elle est déplacé dans la colonne "Review" cette colonne nous permet de savoir lorsqu'une personne pense avoir fini. Nous pouvons donc tester la fonctionnalité développée et donner des retour ou des idées d'amélioration si besoin.
Lorsque la tâche est considérée comme "finie" par l'équipe, nous pouvons la déplacer dans la colonne "Done".



Dashboard du Trello du projet

Le nom des tâche étant assez vague, une description précise se trouve dans la description de chaque tâche. Nous avons aussi ajouté des commentaires à certaines tâches afin de proposer des idées ou de donner des remarques sur les tâches (par exemple dans la colonne "Review".

Tests et assurance qualité

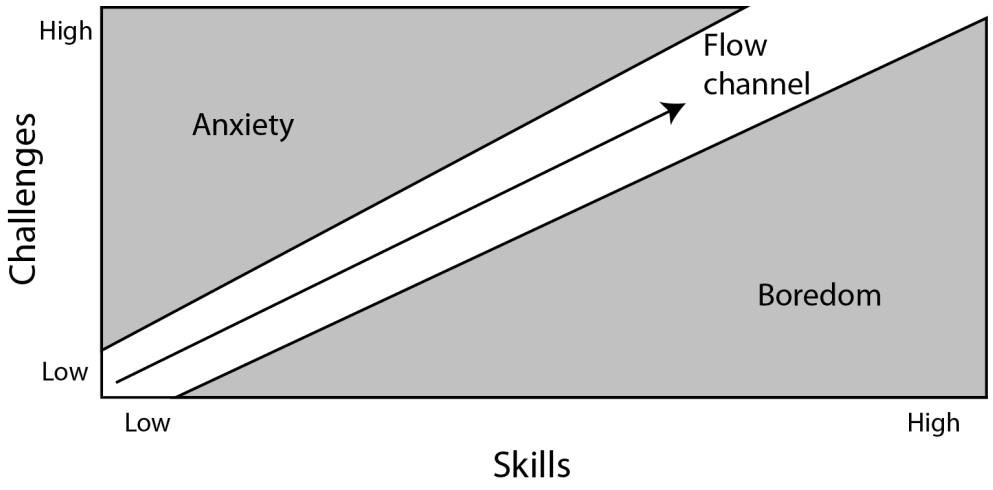
Au fur et à mesure du développement des nouvelles fonctionnalités, plusieurs type de tests étaient effectués.

Tout d'abord, après avoir implémenté une fonctionnalité, chaque développeur devait faire de simples tests au moment de développer leur travail dans leur propre scène. Cela leur permettait de s'assurer du bon fonctionnement de la fonctionnalité.

Ensuite, au moins un second membre de l'équipe devait récupérer le nouveau script ou la nouvelle prefab créée pour la tester dans sa scène. Cela permettait notamment de s'assurer que le script (ou la prefab) conçu fonctionne bien de manière indépendante à son contexte. Par exemple, si un script dépend d'un composant particulier, il est important que soit cette dépendance soit définie en tant qu'attribut serializable (i.e exposé à l'éditeur), soit ce composant soit ajouté au GameObject depuis le script.

Une fois chaque fonctionnalité fonctionnelle, nous avons dû nous assurer que celles-ci le restait tout au long du projet grâce à des **tests de régression**. Pour cela, régulièrement nous avons sollicité les différentes mécaniques du jeu pour vérifier qu'aucun nouveau changement n'affecte les fonctionnalités déjà développées.

Enfin, une fois les différentes mécaniques de jeu mises en places, nous avons passé une étape de **tests d'équilibrage**. Cette fois-ci, l'objectif était de configurer les différentes valeurs du jeu (rythme des vagues de cauchemars, santé des cauchemars, etc.) pour proposer une expérience de jeu à la difficulté croissante, afin de respecter le principe de *flow*.



Lors de cette étape, nous avons fait essayer Spectral à des joueurs extérieur à notre groupe de développement, afin d'obtenir leur ressenti sur la difficulté du jeu, la compréhension du but du jeu, etc.

Conclusion

Ce projet était pour nous tous un premier projet dans le domaine du jeu vidéo. Il a donc été enrichissant sur beaucoup de domaines, principalement concernant le moteur que nous avons utilisé, Unity, et les technologies qui lui sont associées. C'est aussi la première fois que nous avons mis en place les principes de conception de jeux-vidéo.

Pour un premier projet de jeu vidéo, nous sommes fières du résultat. Celui-ci n'est pas parfait et nous pouvons constater que certains éléments aurait pu mieux se dérouler ou qu'avec plus d'expérience nous aurions été bien plus efficaces, notamment en étant plus rigoureux dans la mise en place du travail d'équipe.

Nous avons également été surpris par plusieurs éléments concernant le développement du jeu comme la facilité avec laquelle nous pouvons développer des petit prototype mais la difficulté qui réside dans les détails comme les feedbacks, les indications au joueur etc. Nous avons également été déçu par certaines technologies utilisées par Unity comme leurs gestion des entrées utilisateur qui n'est pas vraiment faite pour gérer le multijoueur local, ou comme le système de particule qui a une interface et des possibilité qui semblent plus limitées que celle de Unreal par exemple.