Rapport de Soutenance 1
The Maze VR

S-TEAM

18 janvier 2019

Martin Camincher
Geoffroy du Mesnil du Buisson
Lucien Leseigle
Hugo Belot-Deloro

Table des matières

1 Présentation du projet

1.1 Le jeu

The Maze VR est un projet de jeu d'horreur en VR. Celui-ci repose sur le principe de l'exploration de labyrinthe.

1.2 Modes de jeu

1.2.1 Mode histoire

Le joueur apparaîtra sur une grande carte ouverte, depuis laquelle il sera possible d'accéder à plusieurs labyrinthes de plus petite taille. Il devra donc explorer ces labyrinthes afin d'obtenir divers objets lui permettant de progresser sur la carte principale. Plusieurs monstres gêneront sa progression, certains pouvant l'attaquer dans les labyrinthes, d'autres sur la carte extérieure.

1.2.2 Mode multijoueur

Dans ce mode de jeu, deux joueurs s'affronteront. L'un explorera le labyrinthe et tentera de s'en échapper, tandis que l'autre aura un plan du labyrinthe et pourra interagir avec son adversaire en faisant apparaître des monstres ou en activant des pièges, voire même en modifiant la structure du labyrinthe en cours de partie pour le gêner.

1.2.3 Éditeur de Labyrinthe

Il sera possible de créer, exporter et importer des labyrinthes entièrement personnalisés grâce à un éditeur de labyrinthe. Ils seront ensuite jouables en solo ou en multijoueurs.

2 Planning de réalisation

2.1 Avancement du projet

Tâche	Gameplay	Interface	Multi	IA	Graphismes	Audio	Web
Prévisionnel	45%	40%	30%	25%	45%	15%	15%
Effectif	35%	40%	70%	25%	35%	15%	15%

2.2 Taches communes

- Réflexions sur le gameplay
- Réflexions sur le style graphique

2.3 Taches individuelles

Hugo :
- Gameplay de base
- Menus
- Game/Scene Manager
- Début du site web

Martin :
- Générateur de labyrinthe
- Détecteur de structure

Geoffroy : - Création de la carte
- Création du premier labyrinthe
- Création de la structure pour la première IA
- Création de trois types d'arbre

Lucien :

3 Avancement individuel

3.1 Hugo : gameplay de base

3.1.1 Travail effectué

J'ai voulu tout d'abord poser les bases de notre jeu, afin que nous puissions rapidement nous lancer dans le développement des fonctionnalités plus avancées. J'ai donc créé une scène test contenant un sol, un mur et un objet représentant le joueur (1). J'ai commencé par ajouter des contrôles permettant au joueur de se déplacer dans les directions cardinales en lui appliquant une force, puis je les ai modifiés afin que le joueur puisse avancer, reculer et tourner, tout en changeant la méthode de déplacement pour utiliser la classe Transform (2). J'ai ensuite attaché la caméra au joueur afin de fournir une vue à la première personne (3).

Je me suis par la suite tourné vers la création d'un menu principal (4). Celui-ci comporte actuellement des boutons pour charger la scène de test ou un labyrinthe généré grâce au générateur de Martin. Il contient également un bouton permettant d'afficher un menu options (5).

3.1.2 Scène persistante

Je me suis ensuite tourné vers un système d'inventaire, mais celui-ci devant être persistant à travers les changements de scène, j'ai dû créer une scène persistante, qui restera toujours active et gérera entre autres l'inventaire. En suivant divers tutoriels, je me suis aperçu que celle-ci pouvait avoir bien d'autres utilités comme par exemple gérer les transitions entre les autres scènes (6).

3.1.3 Objectifs

Mon prochain objectif est donc un système d'inventaire (8) permettant de ramasser les objets nécessaires à la progression, un système de points de vie pour préparer l'implémentation des monstres et une interface graphique affichant les objets possédés et les points de vie. Le nombre d'objets portés à tout instant sera faible, puisque ceux-ci seront utilisés au fur et à mesure pour débloquer de nouvelles zones de la carte. Il n'y aura donc pas besoin d'un grand inventaire, et celui-ci devrait tenir facilement sur l'affichage tête haute. L'inventaire devra également être persistant, et utilisera donc la scène persistante. Un inspecteur personnalisé (9) a été créé pour faciliter la tâche.

Le système de vies sera implémenté sur un script attaché au préfab Joueur. La façon dont ce système fonctionnera n'est pas encore définie. Quand les monstres auront été ajoutés, un écran type "Game Over" sera créé.
L'interface graphique devra permettre la représentation des points de vie et de l'inventaire. Elle sera minimaliste, pour éviter de briser l'immersion.

3.2 Martin : Générateur de labyrinthe et détecteur de structures

3.2.1 Générateur de labyrinthe

Notre projet étant un jeu de labyrinthe en 3D, il nous est nécessaire de pouvoir créer et manipuler des modèles de labyrinthes tout au long du développement du jeu. Il était pour cela possible d'utiliser Blender afin de créer des labyrinthes en 3D, mais nous ne maîtrisons pas tous cet outil, et cela aurait rendu la tache de créer et manipuler les labyrinthes relativement complexe. J'ai donc développé un générateur de labyrinthe sur Unity (10). Celui-ci prend en paramètres un plan du labyrinthe et une liste de structures préfabriquées. Chaque pixel du plan est observé, et, en fonction de sa valeur RGBA et de sa position, une des structures fournies est placée aux coordonnées indiquées par la position du pixel.
Actuellement, les structures utilisables permettent de placer les murs et le sol du labyrinthe, et font apparaître le joueur, mais des structures plus complexes seront ensuite créés, telles des portes ou des pièges. Les actions réalisables par les structures, comme l'ouverture d'une porte, seront directement intégrées au préfabriqué.
Ce générateur a pour avantage de pouvoir très simplement modifier le labyrinthe, avec une simple manipulation du plan. De plus, grâce au codage RGBA, il est possible d'utiliser au maximum $256^4 \simeq 4$ milliards de structures différentes dans un même labyrinthe, ce qui sera largement suffisant.
Il sera également possible de modifier le générateur, afin d'utiliser plusieurs plans et ainsi concevoir un labyrinthe à multiples étages.

3.2.2 Détecteur de structures

Afin de pouvoir réaliser certaines actions, telles qu'ouvrir une porte, il faut que le joueur soit face à la porte, et donc il faut savoir quelle structure se trouve devant le joueur (12). Pour cela, on utilise un Ray, une ligne infinie ayant une origine, le joueur, et une direction, devant le joueur. Cela nous permet de définir la ligne de vue du joueur. La fonction Physics. Raycast nous permet ensuite de savoir si le Ray a une intersection avec une structure à une certaine distance du joueur. Cela permet ainsi de retrouver la structure que le joueur regarde.

3.2.3 Objectifs

Je cherche maintenant à créer différentes structures préfabriquées, comme des portes ou des pièges, qui permettront d'apporter de la diversité dans les labyrinthes créés par le générateur.

3.3 Geoffroy : design de la carte

3.3.1 La carte dans son ensemble

La carte en totalité sera de taille de un km par un km à échelle humaine. Elle sera divisée en quatre parties. Elle sera sous forme d'un Open World dirigé de telle sorte que le joueur puisse aller où il le souhaite dans la partie de la carte dans laquelle il se trouve, ce qui lui donnera une sensation de liberté tout en étant quand même guidé. Les zones seront dissociables par leurs climats et leurs environnements. Le joueur sera toujours en mesure de savoir dans quelle partie de carte il se trouve.

3.3.2 Les limites de la carte

La carte étant assez grande nous sommes donc contraints de lui mettre des limites de telle sorte que le jeu ne soit pas trop lent puisqu'il devra tourner sur téléphone par conséquent nous sommes obligés de créer des limites physiques au joueur. Pour éviter de casser l'immersion en créant des murs invisibles ou des murs simples, nous avons décidé de créer la carte sur une île en hauteur un peu comme Fortnite ou Apex l'on fait, le joueur pourra donc tomber de la carte/île et mourir s'il ne fait pas assez attention. Un problème que nous avons est que pour la fluidité du jeu, la carte étant très grande il y donc beaucoup de faces sur Blender, or la fluidité du jeu est proportionnelle au nombre de faces c'est pourquoi pour limiter le nombre de faces nous allons essayer de faire comme Minecraft, c'est-à-dire charger et décharger des parties de la carte en fonction de la positon du joueur, de sorte que plus le joueur avance vers l'avant plus la carte se chargera devant lui et se déchargera derrière lui. Afin que le joueur ne remarque pas ce détail, nous limiterons l'horizon du champ de vision du joueur comme s'il y avait du brouillard.

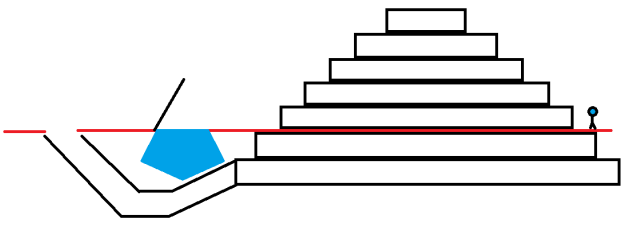
3.3.3 L'organisation de la carte

La carte est divisée en quatre parties : la Forêt, le Désert, les Montagnes et le Marais. Ces zones seront uniquement accessibles au joueur une fois qu'il aura effectué certaines tâches. La première zone sera la Forêt et sera composée du premier labyrinthe, d'une montagne et d'un hangar abandonné, elle sera assez sombre et ressemblera un peu à une forêt hantée.

Pour sortir de la première zone il faudra réunir trois clés : la première se trouvera dans le labyrinthe, la deuxième sera dans une grotte dans la montagne et la dernière se trouvera dans le hangar. Les trois clés serviront à déverrouiller une grille qui nous donne accès à un pont pour pouvoir passer de l'autre côté de la rivière, car les trois premières zones seront séparées par une rivière/lac et le personnage ne pourra pas traverser l'eau avant d'être allé au moins une fois dans la zone quatre.

Une fois passée la première rivière le joueur arrivera dans la deuxième zone. Celle-ci sera composée d'une pyramide, d'une montagne et d'un village, elle se présentera sous une forme un peu aride comme un désert d'orient. La montagne sera cette fois seulement décorative et n'aura pas d'importance pour le joueur, il faudra juste qu'il la traverse.

Pour passer de la deuxième à la troisième zone le joueur devra traverser la pyramide qui est un véritable casse-tête.



Pyramyde.png

Elle est composée de sept étages, tous composés de labyrinthes. Si le joueur passe le labyrinthe sans avoir visité toute la deuxième zone, il sera forcé de faire marche arrière car il ne possédera pas l'équipement nécessaire à sa survie. L'équipement sera dans le village, mais le village n'est pas visible par le joueur à l'entrée de la zone 2, il doit donc le chercher et par conséquent, plusieurs essais seront nécessaires au joueur avant de pouvoir passer à la zone suivante. Pour l'instant seuls deux niveaux de la pyramide ont été réalisés, car cela nécessite un maximum d'optimisation afin que le jeu reste un minimum jouable.

Une fois dans la troisième zone avec l'équipement le joueur aura la possibilité d'abaisser un pont-levis afin de pouvoir transiter entre la zone trois et la zone deux sans avoir à refaire la pyramide dans le sens inverse, mais la clé du pont-levis sera cachée à l'intérieur de la pyramide, il sera donc conseillé au joueur de bien visiter la pyramide afin de ne pas manquer d'indices qui pourront être importants pour la suite. Il est aussi important pour le joueur d'abaisser le pont-levis, car même pour les plus malins qui prendront leurs précautions et qui auront déjà la matériel pour la troisième zone et qui ne feront pas la pyramide en entier, ils se retrouveront très vite bloqués.

Passé ces étapes, le joueur sera dans la troisième zone composée de trois montagnes, deux labyrinthes, la troisième zone sera une zone de glace et de froid rappelant un peu une ère glacière. Le joueur sera contraint de faire les labyrinthes dans un certain ordre. A la fin du premier labyrinthe le joueur trouvera un plan donnant l'emplacement de deux clés. La première se trouvera dans la première zone et ne sera pas accessible tant que la carte n'est pas déverrouillée. La deuxième clé se trouvera dans la deuxième zone et pourra être stockée par le joueur. Une fois les deux clés réunies le joueur aura accès au deuxième labyrinthe.

Le reste de la carte et des objectifs n'a pas encore été imaginé, nous pensons juste que la dernière zone sera une zone marécageuse avec un thème de jungle.

3.3.4 La végétation

Comme dit précédemment nous sommes limités en capacité, par conséquent nous aurons un végétation présente mais faible graphiquement afin de limiter la surcharge. Nous aurons donc cinq à six types d'arbres dont trois ont déjà été réalisés. Les trois types déjà conçus serviront à la première et troisième zones dans les marécage. Nous nous limiterons à quelques plantes pour les raisons précédemment citées.

3.3.5 Les Labyrinthes

Pour l'instant nous n'avons pu construire qu'un seul labyrinthe sur Blender, mais nous avons pour projet de faire l'ensemble des labyrinthes de la pyramide sur Blender, le reste des labyrinthes sera fait grâce au générateur de Martin. Cela est dû aux contraintes que nous voulons rajouter dans la pyramide qui ne pourront pas être résolues par le générateur. Nous avons pris un peu de retard sur le nombre de créations de labyrinthe du fait de la complexité de Blender qui est un logiciel extrêmement complet et qui par conséquent nécessite du temps avant de pouvoir être pris en main.

3.3.6 Habitations

La carte comportera un village et un entrepôt donc d'habitations. Pour l'instant nous ferons une version bêta de l'intérieur des maisons en se limitant à quelques objets. Il y aura quatre à cinq types d'habitations différentes plus celles que nous ferons en annexe en fonction de ce qu'il y aura dans la dernière zone.

3.3.7 Les IA

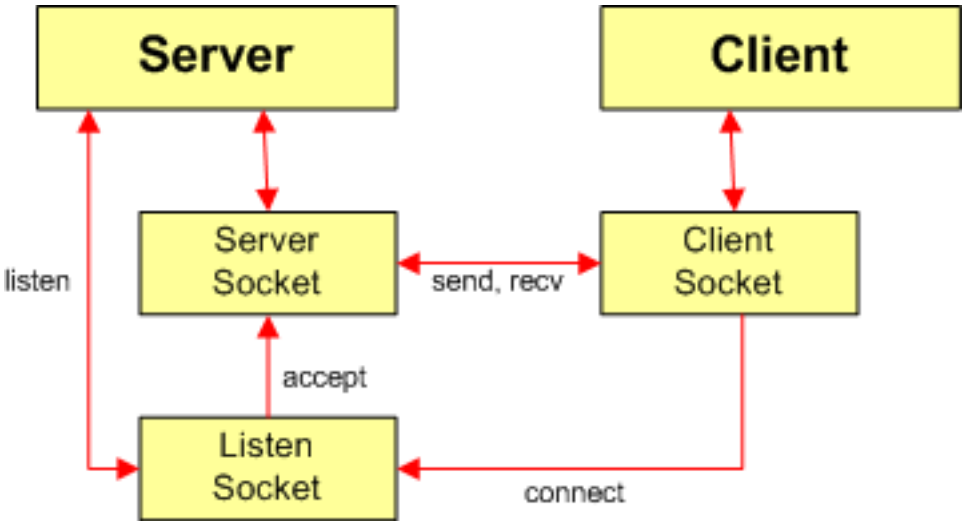
Pour le design de l'IA nous avons un premier prototype mais il y aura quatre à cinq types d'IA d'apparences différentes. Nous ne les avons pas encore imaginé.

3.4 Lucien : multijoueur, IA et Audio

3.4.1 Multijoueur

Le multijoueur doit permettre à deux joueurs maximum d'interagir en temps réel. Le joueur 2 (qui contrôle la carte) déclenche divers

évènements qui seront implémentés dans le code du joueur 1 (VR). Une latence de quelques millisecondes ne devrait pas avoir d'influence négative sur le gameplay. De ce fait, une connexion Client/Serveur basique est suffisante : le joueur VR est le serveur, le joueur 2 est le client. Ils peuvent échanger des messages sous la forme d'une chaîne de caractères, qui correspondront aux événements (par exemple, le message « trap_01 » déclenchera l'action correspondante et activera le piège d'index 01). Dans l'état actuel, le client et le serveur sont entièrement fonctionnels, mais les messages envoyés ne correspondent à aucune action du jeu (ce qui relève de la partie gameplay).



3.4.2 IA

Le comportement de l'intelligence artificielle des ennemis du jeu aura un grand impact sur le gameplay. En effet l'IA ne doit pas être trop performante, mais être le plus effrayant possible. De ce fait, il semble souhaitable de ne pas donner à l'IA trop d'informations qui lui permettraient de trouver le joueur trop rapidement, et sans se montrer dangereuse le reste de la partie. Tout d'abord, voici le comportement idéal de l'IA : Un ennemi est un chasseur, le joueur est sa proie. Il connaît déjà la carte (c'est son terrain de chasse habituel), et il est capable de repérer des indices laissés par le joueur (traces de pas, certains évènements comme l'activation d'un gros piège), dans le but

de le trouver le plus vite possible. Il n'a pas accès à la position du joueur. Un ennemi aura donc accès (au moins) à ces informations, à chaque update :
- design de la carte
- indices et position du joueur s'il est dans sa vision
- l'activation de certains évènements (pièges, spawn, etc... les évènements en question n'ont pas encore été choisi) Et il devra déterminer (au moins) son déplacement. A priori, cela se traduira par une direction x,y dans sa vision, qui sera utilisée par une fonction built-in d'Unity qui gérera le déplacement. De ce fait, une IA pourra être implémentée sous la forme :
- D'un réseau de neurones récurrent, ou LSTM. Il permettra à l'IA de se souvenir et d'appliquer des stratégies complexes dans le temps. Toutefois les librairies de Machine Learning en C# sont peu nombreuses et la plupart ne propose pas de LSTM.
- D'un réseau de neurones basique, certes moins performant mais plus simple à implémenter et qui peut offrir de très bons résultats.
Quel que soit le modèle choisi, il sera si possible entraîné avec un algorithme génétique.
Calqué sur le principe de l'évolution, ce modèle présente divers avantages dans le cadre de notre jeu par rapport aux méthodes classiques pour entraîner les réseaux de neurones :
- Une fonction de fitness complexe permettra de définir un autre objectif à l'IA que simplement de trouver le joueur. On peut par exemple imaginer récompenser l'IA quand le joueur la voit, favorisant une IA qui se montre présente et menaçante et qui ne se contente pas de surprendre le joueur au dernier moment. On peut également définir la fonction de fitness en fonction d'un groupe de plusieurs IA, créant ainsi une intelligence collective (qui se comporterait comme une meute)
- Un fois le modèle créé, il sera simple de modifier la fonction de fitness et de générer de nouvelles configurations changeant le gameplay.
- On peut extraire de chaque génération les individus (réseaux de neurones) qui offrent des performances intéressantes. Cela signifie que l'on entraîne non pas une intelligence la plus performante possible, mais de très nombreux individus de niveau quantifiable et avec un comportement différent.
Ce qui est à faire :

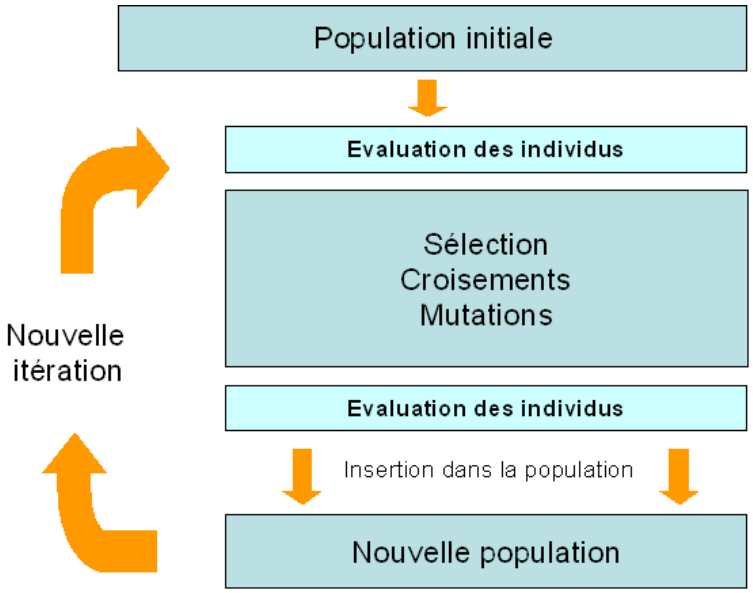
- Choisir une implémentation pour 1 individu : LSTM si possible, sinon NN classique ou algorithme modulaire.
- Définir une fonction de fitness qui récompense les individus
- Implémenter l'algorithme génétique : générer 1000 individus (par exemple) en fonction des générations précédentes (ou bien avec des paramètres aléatoires pour la première génération). Procéder à des mutations, ou changements aléatoires sur certains paramètres de certains individus. Évaluer leurs performances et sélectionner les meilleurs individus pour la génération suivante. De plus, sauvegarder les individus intéressant à implémenter dans le jeu. Puis recommencer à l'étape une, jusqu'à observer une forme d'équilibre (l'espèce arrête d'évoluer).

Une fois implémenté, cet algorithme permettra de choisir entre des milliers d'individus aux comportements différents (et quantifié par la sélection et la fonction de fitness) et de garantir une expérience riche et non monotone. Le but est que le joueur aie l'impression d'être chassé par une créature intelligente, le forçant à être réactif et prudent sur les informations qu'il donne à l'IA (ex : ne pas rester près d'un endroit après y avoir fait beaucoup de bruit en déclenchant un objet par exemple). De plus, dans le mode créateur de cartes, le joueur 2 qui répartit des points dans la création de la partie pourra choisir entre des créatures intelligentes et coûteuses, ou bien moins chères mais au comportement moins optimal, ou encore des créatures qui agissent en groupe...

Chaque IA étant définie par quelques valeurs numériques seulement, les milliers d'individus pourront être stockés dans les fichiers du jeu et être sélectionnés au hasard parmi un groupe de 'niveau' (défini par la fonction de fitness) équivalent. Cela signifie qu'à chaque partie, le joueur aura affaire à des membres différents d'une même espèce, qui a évolué spécialement pour le tuer tout en garantissant une expérience ludique (les ennemis trop forts, trop peu présents, etc... ne seront pas conservés).

J'ai déjà développé un modèle "proof of concept" d'un réseau de neurones LSTM entrainé par un algorithme génétique pour de la prédiction de séquences de texte, mais il ne semble pas fonctionnel (ou bien il n'est pas réaliste de l'entraîner avec aussi peu de puissance de calcul que celle disponible). S'il n'est pas possible d'utiliser un algorithme génétique, j'entraînerai le réseau LSTM avec d'autres mé-

thodes (backpropagation), ou bien je développerai une IA classique de pathfinding (en bénéficiant des fonctions builtin d'Unity).



3.4.3 Musique et Sons

Le moteur sonore utilisé sera le moteur 3D d'Unity. Les fichiers seront stockés (a priori au format wav) avec les fichiers du jeu, et déclenchés avec les évènements correspondants. Différents bruitages seront associés au même événement, et l'un d'entre eux sera choisi aléatoirement, pour rendre le son plus réaliste et moins monotone.
La musique est composée sur Ableton, puis elle sera exportée en pistes séparées dont les volumes (et potentiellement des effets audio type reverb, filtre, delay...) seront modifiés en fonction de l'état actuel du jeu (ex : un thème effrayant sera associé à la présence d'un ennemi, le volume de cette piste augmentera progressivement). Le moteur 3D n'est pas utilisé pour ne pas donner d'information précise au joueur, la musique fait partie de l'ambiance du jeu mais ne doit pas trop faciliter le gameplay). Actuellement, environ 3 min de musique sont composées, de manière à pouvoir boucler sans transition visible.
Le tempo de la musique sera variable si cela est possible dans Unity (il augmentera au fil du jeu, pour signifier le timer qui approche à sa

fin dans les labyrinthes par exemple).
Il faut donc :
- Créer les différents bruitages à l'aide de logiciels de synthèse sonore (Xfer Serum) et de samples
- Continuer la composition de la musique, et implémenter la gestion du volume des pistes dans Unity
- Associer les bruitages à leur événement dans le moteur 3D d'Unity
- Gérer les volumes des pistes de la musique en fonction de la présence d'ennemis, de piège, de l'avancement du temps dans la partie, etc...

4 Conclusion

Les points les plus importants qu'il nous reste à traiter sont la création de l'environnement, les interactions entre le joueur et les objets, monstres et pièges.
Différentes structures doivent être créées pour le générateur, afin d'offrir une plus grande liberté de création pour les labyrinthes.
Le multijoueur doit être incorporé dans le reste du code, et un système de lobby doit être réalisé.
Les différentes IA du jeu ne sont pas encore fonctionnelles, et bien que le code de l'algorithme génétique soit assez avancé, il ne sera pas forcément possible de l'implémenter dans le jeu (des IA plus classiques seront alors réalisées, notamment grâce aux fonctions builtin d'Unity).
La musique du jeu dure actuellement 3 min (en boucle), l'idéal serait d'avoir 10 min de musique environ à boucler.
Les bruitages ne sont pas encore réalisés, ni implémentés dans le moteur 3d d'Unity.

5 Annexes

5.1 Hugo

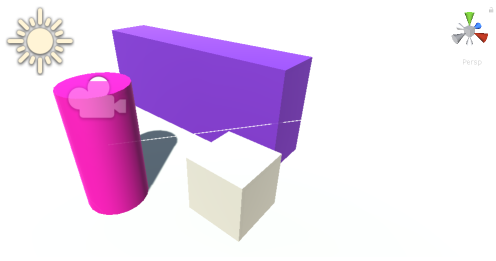


FIGURE 1 – Vue de la scène

```
public class PlayerMovement : MonoBehaviour
{
    private const float MovementAmount = 0.08f;
    private const float RotationAmount = 5f;

    void FixedUpdate()
    {
        int forwardMovement = Convert.ToInt32(Input.GetKey("z")) - Convert.ToInt32(Input.GetKey("s"));
        int rotation = Convert.ToInt32(Input.GetKey("d")) - Convert.ToInt32(Input.GetKey("q"));
        transform.position += transform.forward * MovementAmount * forwardMovement;
        transform.Rotate(Vector3.up * rotation * RotationAmount, Space.World);
    }
}
```

FIGURE 2 – Script des contrôles

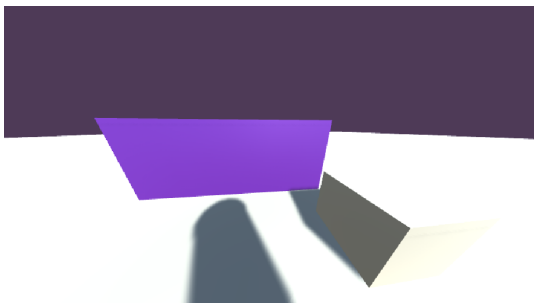


FIGURE 3 – Vue première personne

FIGURE 4 – Menu Principal

FIGURE 5 – Menu Options

```
public class GameManager : MonoBehaviour
{
    public static GameManager instance;
    public string currentSceneName;

    private void Awake()
    {
        if (instance != null)
            Destroy(gameObject);
        else
        {
            instance = this;
            //DontDestroyOnLoad(gameObject);
            SceneManager.LoadSceneAsync("MainMenu", LoadSceneMode.Additive);
            currentSceneName = "MainMenu";
        }
    }

    public void LoadScene(string scene)
    {
        SceneManager.LoadSceneAsync(scene, LoadSceneMode.Additive);
        currentSceneName = scene;
    }

    public void UnloadScene(string scene)
    {
        StartCoroutine(Unload(scene));
    }

    IEnumerator Unload(string scene)
    {
        yield return null;
        SceneManager.UnloadSceneAsync(scene);
    }
}
```

FIGURE 6 – Le Game Manager

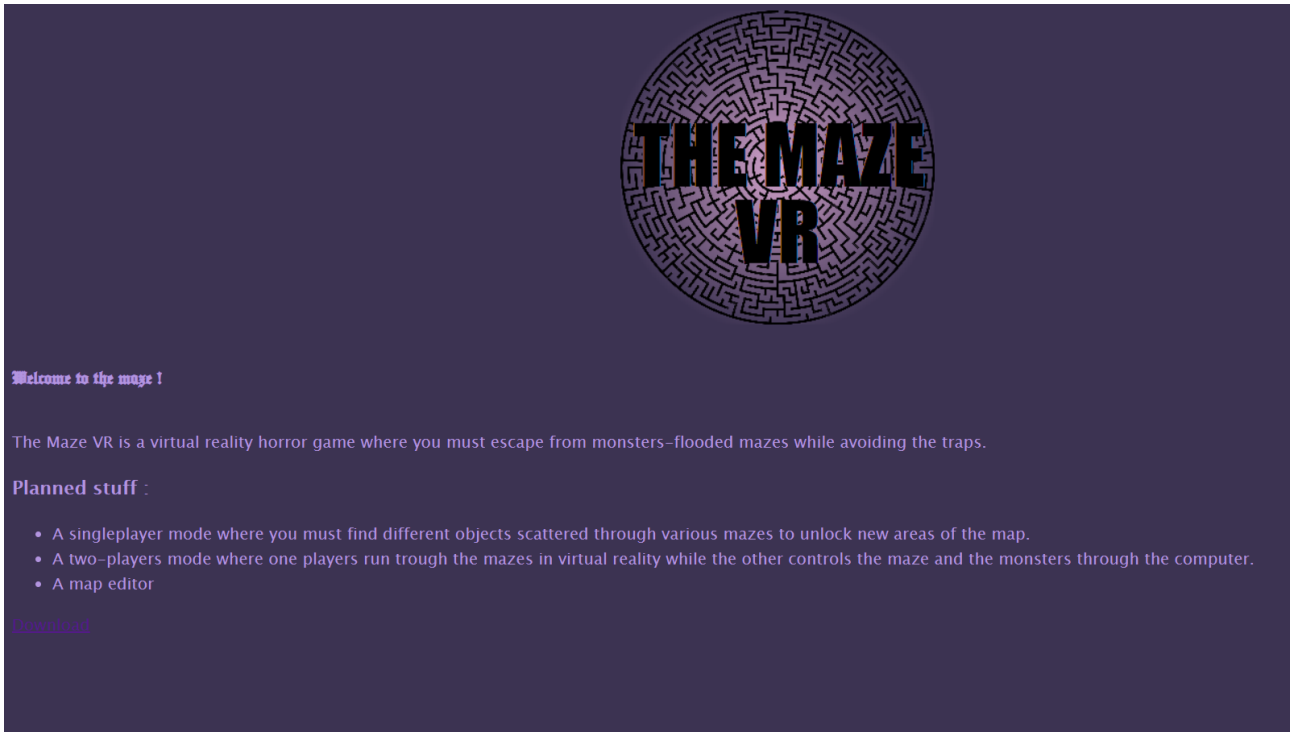


FIGURE 7 – Site Internet

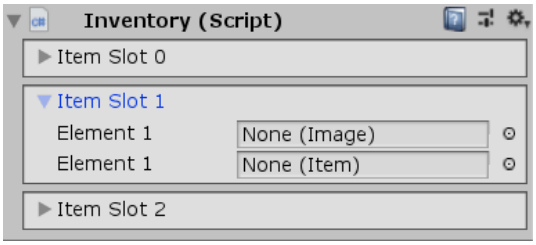
```
public class Inventory : MonoBehaviour
{
    public Image[] itemImages = new Image[MaxItems];
    public Item[] items  = new Item[MaxItems];
    public const int MaxItems = 3;

    public bool AddItem(Item item)
    {
        int i = 0;
        while (i < MaxItems && items[i] != null)
        {
            i++;
        }
        if (items[i] == null)
        {
            items[i] = item;
            itemImages[i].sprite = item.sprite;
            itemImages[i].enabled = true;
            return true;
        }
        return false;
    }

    public bool RemoveItem(Item item)
    {
        int i = 0;
        while (i < MaxItems && items[i] != item)
            i++;
        if (i != MaxItems)
        {
            items[i] = null;
            itemImages[i].sprite = null;
            itemImages[i].enabled = false;
            return true;
        }
        return false;
    }

}
```

FIGURE 8 – Le système d'inventaire




```
[CustomEditor(typeof(Inventory))]
public class InventoryEditor : Editor
{
    private SerializedProperty itemImagesProperty;
    private SerializedProperty itemsProperty;
    private bool[] showItemsSlots = new bool[Inventory.MaxItems];
    private const string inventoryPropItemImagesName = "itemImages";
    private const string inventoryPropItemsName = "items";


    private void OnEnable()
    {
        itemImagesProperty = serializedObject.FindProperty(inventoryPropItemImagesName);
        itemsProperty = serializedObject.FindProperty(inventoryPropItemsName);
    }

    public override void OnInspectorGUI()
    {
        serializedObject.Update();
        for (int i = 0; i < Inventory.MaxItems; i++)
        {
            ItemSlotGUI(i);
        }
        serializedObject.ApplyModifiedProperties();
    }

    private void ItemSlotGUI(int index)
    {
        EditorGUILayout.BeginVertical(GUI.skin.box);
        EditorGUI.indentLevel++;
        showItemsSlots[index] = EditorGUILayout.Foldout(showItemsSlots[index], "Item Slot " + index);
        if (showItemsSlots[index])
        {
            EditorGUILayout.PropertyField(itemImagesProperty.GetArrayElementAtIndex(index));
            EditorGUILayout.PropertyField(itemsProperty.GetArrayElementAtIndex(index));
        }
        EditorGUI.indentLevel--;
        EditorGUILayout.EndVertical();
    }
```

FIGURE 9 – Un custom inspector

5.2 Générateur de labyrinthe

```
using UnityEngine;

[System.Serializable]
public class ListPrefab
{
    public GameObject prefab;
    public string name;
}

using UnityEngine;

public class Generator : MonoBehaviour
{
    public Texture2D map;

    public ListPrefab[] P;

    public void GenerateLab()
    {
        for (int j = 0; j < map.height; j++)
        {
            for (int i = 0; i < map.width; i++)
            {
                Color c = map.GetPixel(i, j);
                if (i % 2 == 0 && j % 2 == 0)
                {
                    if (c == new Color(0,0,0,1))
                        create("Pillar", i, j,false);
                }
                else if (i % 2 == 1 && j % 2 == 0)
                {
                    if (c ==new Color(0,0,0,1))
                        create("Wall", i, j,true);
                }
                else if (i % 2 == 0 && j % 2 == 1)
                {
                    if (c ==new Color(0,0,0,1))
                        create("Wall", i, j,false);
                }
                else if (i % 2 == 1 && j % 2 == 1)
                {
                    if (c==new Color(1,0,0,1))
                        create("Player",i,j,false);
                    create("GroundTile", i, j,false);
                }
            }
        }
    }
```

FIGURE 10 – Générateur de labyrinthe 1/2

```
public void create(string name, int x,int y, bool rotate){
    Vector3 v;
    foreach (ListPrefab G in P)
        {
            if (G.name == name)
            {
                switch (G.name)
                {
                    case ("Pillar"):
                        v = new Vector3(2 * x, (float) 2.5, 2 * y);
                        Instantiate(G.prefab, v, Quaternion.identity, transform);
                        break;
                    case ("Wall"):
                        v = new Vector3(2 * x, (float) 2.5, 2 * y);
                        if (rotate)
                            Instantiate(G.prefab, v,Quaternion.Euler(new Vector3(0,90,0)), transform);
                        else
                            Instantiate(G.prefab, v, Quaternion.identity, transform);
                        break;
                    case ("GroundTile"):
                        v = new Vector3(2*x,0,2*y);
                        Instantiate(G.prefab, v, Quaternion.identity, transform);
                        break;
                    case ("Player"):
                        v=new Vector3(2*x,(float)1.5 ,2*y);
                        Instantiate(G.prefab, v, Quaternion.identity, transform);
                        break;
                }
            }
        }
}
```

FIGURE 11 – Générateur de labyrinthe 2/2

5.3 Détecteur de structures

```
public class PlayerCursor : MonoBehaviour
{
    void Update()
    {
        Vector3 PlayerPosition = transform.position;

        Ray InteractionRay = new Ray(PlayerPosition,transform.forward);

        RaycastHit InteractionRayHit;

        float InteractionRayLength = 2.0f;

        Vector3 InteractionRayEndPoint = transform.forward * InteractionRayLength+PlayerPosition;

        Debug.DrawLine(PlayerPosition,InteractionRayEndPoint);

        bool HitFound = Physics.Raycast(InteractionRay, out InteractionRayHit, InteractionRayLength);
        if (HitFound)
        {
            GameObject HitGameObject = InteractionRayHit.transform.gameObject;

            string HitFeedBack = HitGameObject.name;

            Debug.Log(HitFeedBack);

            if (Input.GetKey("e"))
            {
                //TODO
            }
        }
    }
}
```

FIGURE 12 – Détecteur de structures

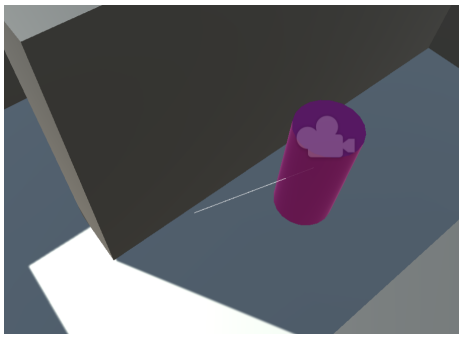


FIGURE 13 – Visualisation du détecteur