Document de conception - Projet EVHI

Projet EVHI de :
Cédric Cornede
Damien Legros
Petar Calic

Master ANDROIDE
Université Sorbonne Université

Table des matières

1 Évolution de l'aspect global du projet

Dans le cahier des charges, nous avions définis plusieurs types d'énigmes que nous pourrions proposer aux joueurs offrant des gameplays différents dans chaque énigme. Cependant suite aux contraintes de temps du projet, nous avons finalement décidé de nous concentrer sur un seul type d'énigme pour passer plus de temps sur l'adaptation de la difficulté.

Nous avons sélectionnée l'énigme suivante, étant celle qui offre le plus de possibilités d'adaptation parmi les 4 énigmes décrites dans le cahier des charges. L'environnement de l'énigme comprend les éléments suivants :

— Un pistolet que le joueur peut prendre et avec lequel il peut tirer.

— Un personnage non jouable (PNJ) à qui le joueur pourra s'adresser ou tirer dessus.

— Un boîtier contenant un interrupteur pour obtenir la clé pour ouvrir la porte.

— Une cible sur laquelle le joueur pourra tirer.

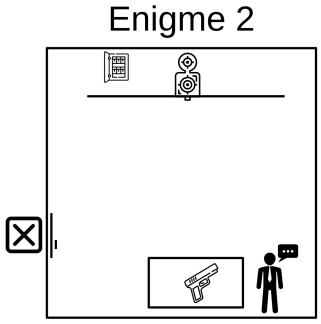


FIGURE 1 – Schéma de l'énigme sélectionnée

Le joueur aura 4 manières différentes d'ouvrir la porte :

1. Discuter avec le PNJ et répondre à son énigme à choix multiples pour ouvrir la porte. (**ACTION SOCIALIZER**)
2. Tirer sur le PNJ avec le pistolet. Une fois fait, le PNJ lâchera une clé pour ouvrir la porte. (**ACTION KILLER**)
3. Tirer sur la cible avec le pistolet, le score du joueur augmentera à chaque cible touchée jusqu'à donner la clé pour ouvrir la porte. (**ACTION ACHIEVER**)
4. Trouver et ouvrir le boîtier, enclencher le bouton du boîtier puis ouvrir la porte avec la clé obtenu. (**ACTION EXPLORER**)

Les positions des objets dans la salle seront placés de manière aléatoire pour avoir une expérience différente à chaque fois. N'ayant plus qu'une énigme les salles se suivront avec les 4 mêmes actions possibles. Le niveau de difficulté de chaque action est choisi par rapport aux actions prises précédemment par le joueur (On augmentera la difficulté d'une action qui est souvent effectuée).

Une adaptation rigide entre les salles de la difficulté sera donc implémenté en plus d'une adaptation dynamique au sein de chaque salle.

2 Modélisation des données

Notre système comporte différentes données. Deux fichiers seront utilisés pour sauvegarder les informations nécessaire à l'adaptation au cours du jeu :

— *rigidAdaptationData.txt*

— *dynamicAdaptationData.txt*

Ces fichiers permettront de stocker les deux profils d'adaptation du joueur au cours de la partie. Des fonctions de sauvegarde et de lecture seront utilisées pour accéder aux fichiers et les enrichir ainsi que pour récupérer leurs données. Un électroencéphalogramme sera utilisé pour récupérer des informations en plus des entrées de l'ordinateur.

Une classe **SaverClass** sera utilisée pour stocker l'ensemble des fonctions d'écriture et de lecture des fichiers de données.

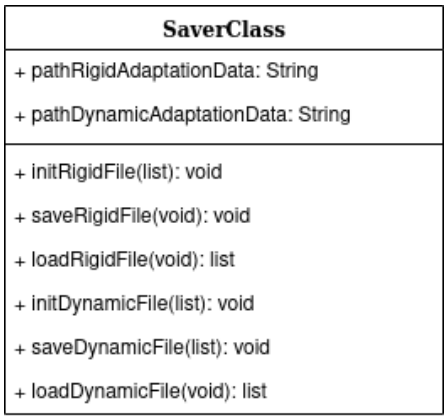


FIGURE 2 – Diagramme de la classe **SaverClass**

Pour l'adaptation rigide de la difficulté, correspondant à la difficulté choisi pour chaque action possible dans une énigme, les données seront stockés dans *rigidAdaptationData.txt*.

Les données seront modélisés dans une classe serializable **RigidAdaptationClass** pour pouvoir sauvegarder les variables de la classe dans le fichier.

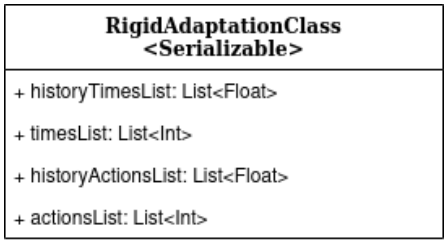


FIGURE 3 – Diagramme de la classe **RigidAdaptationClass**

Ces variables enregistrées seront utilisées pour calibrer la difficulté rigide de chaque action au début de chaque énigme.

— *historyTimesList* sera utilisée pour suivre l'évolution du choix des actions au cours du jeu et garder

une trace des actions prises depuis le début jusqu'à la fin (si le joueur effectue 2 énigmes en **KILLER** et 1 en **SOCIALIZER** on aura donc $['KILLER','KILLER','SOCIALIZER']$ par exemple).

— *timesList* sera utilisée pour stocker les actions prises dans le jeu, on y stockera le nombre des actions de chaque type (si le joueur effectue 2 énigmes en **KILLER** et 1 en **SOCIALIZER** on aura donc $[2, 0, 1, 0]$ par exemple). Cela permettra de savoir la difficulté rigide à mettre pour chaque type d'action.

— *historyTimesList* sera utilisée pour suivre l'évolution des temps passé par énigme au cours du jeu et garder une trace des temps depuis le début jusqu'à la fin (si le joueur effectue 2 énigmes en **KILLER** en 30.5 secs et 26.3 secs et 1 en **SOCIALIZER** 29.4 secs on aura donc $[30.5, 26.3, 29.4]$ par exemple).

— *timesList* sera utilisée pour stocker les actions prises dans le jeu, on y stockera le nombre des actions de chaque type (si le joueur effectue 2 énigmes en **KILLER** en 30.5 secs et 26.3 secs et 1 en **SOCIALIZER** 29.4 secs on aura donc $[57, 0, 29, 0]$ par exemple). Cela permettra de savoir la difficulté rigide à mettre pour chaque type d'action.

Pour l'adaptation dynamique de la difficulté, correspondant à l'évolution de la difficulté donnée à chaque action possible dans une énigme, les données seront stockées dans *dynamicAdaptationData.txt*.

Les données seront modélisés dans une classe serializable **DynamicAdaptationClass** pour pouvoir sauvegarder les variables de la classe dans le fichier.

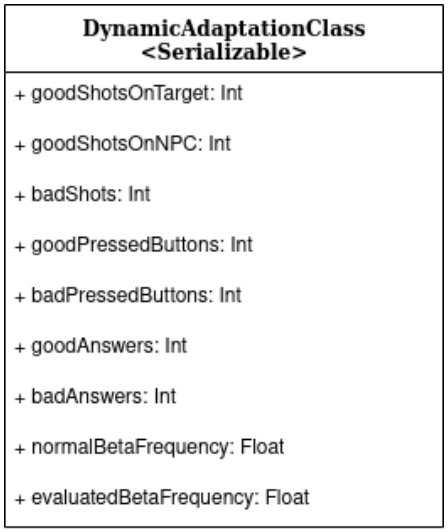


FIGURE 4 – Diagramme de la classe **DynamicAdaptationClass**

Ces variables enregistrées seront utilisées pour calibrer la difficulté dynamique de chaque action toutes les secondes au cours de chaque énigme. Sauf pour l'électroencéphalogramme où les données de *evaluatedBetaFrequency* sont seulement mis à jour toutes les 10 secondes.

— Pour chaque action *normalBetaFrequency* sera comparé avec *evaluatedBetaFrequency* pour savoir si le joueur à une forte ou faible concentration et savoir comment la difficulté doit être calibrée.

— Pour les actions de type **ACHIEVER** le ratio du nombre de *goodShotsOnTarget* et de *badShots* sera utilisé pour savoir comment la difficulté du tir sur les cibles doit être calibrée (Précision du tir ou vitesse des cibles par exemple).

— Pour les actions de type **KILLER** le ratio du nombre de *goodShotsOnNPC* et de *badShots* sera utilisé pour savoir comment la difficulté du tir sur les cibles doit être calibrée (Précision du tir ou vitesse du PNJ par exemple).

— Pour les actions de type **EXPLORER** le ratio du nombre de $goodPressedButtons$ et de $badPressedButtons$ sera utilisé pour savoir comment la difficulté de l'intervalle dans lequel le bouton du boîtier peut être utilisé doit être calibrer (Intervalle de temps d'interaction avec le bouton par exemple).

— Pour les actions de type **SOCIALIZER** le ratio du nombre de $goodAnswers$ et de $badAnswers$ sera utilisé pour savoir comment la difficulté des questions doit être calibrée (Temps de réponse par question par exemple).

2.1 Initialisation des données

Une fonction **initRigidFile()** sera utilisée pour initialiser les données au début du jeu de la classe **RigidAdaptationClass**. Les informations suivantes seront initalisées :

$$historyActionsList \leftarrow []$$

$$actionsList \leftarrow [0, 0, 0, 0]$$

$$historyTimesList \leftarrow []$$

$$timesList \leftarrow [0, 0, 0, 0]$$

Une fonction **initDynamicFile()** sera utilisée pour initialiser les données au début du jeu de la classe **DynamicAdaptationClass**. Les informations suivantes seront initialisées :

$$goodShotsOnTarget \leftarrow 0$$

$$goodShotsOnNPC \leftarrow 0$$

$$badShots \leftarrow 0$$

$$goodPressedButtons \leftarrow 0$$

$$badPressedButtons \leftarrow 0$$

$$goodAnswers \leftarrow 0$$

$$badAnswers \leftarrow 0$$

$$normalBetaFrequency \leftarrow calibratedBetaFrequency$$

$$evaluatedBetaFrequency \leftarrow null$$

Au début du jeu une brève explication des consignes sera présenté au joueur, pendant cette explication le jeu calibrera le rythme des ondes bêta de l'électroencéphalogramme du joueur en situation "normal". Cette calibration sera stockée dans la valeur *calibratedBetaFrequency* pour l'initialisation comme ci-dessus.

Si le joueur s'écarte trop de cette mesure le jeu le détectera alors comme une forte concentration ou inversement une faible concentration.

```
public void InitRigidFile()
{
    FileStream file;

    if(File.Exists(pathRigidAdaptationData)) file = File.OpenWrite(pathRigidAdaptationData);
    else file = File.Create(pathRigidAdaptationData);

    RigidAdaptationClass data = new RigidAdaptationClass([], [0, 0, 0, 0], [], [0, 0, 0, 0]);
    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(file, data);
    file.Close();
}
```

FIGURE 5 – Exemple de code pour **initRigidFile()**

2.2 Sauvegarde des données

Une fonction **saveRigidFile()** sera utilisée pour sauvegarder à chaque fin d'énigme les différentes informations dans la classe **RigidAdaptationClass**. Les informations suivantes sont mis à jour :

$$historyActionsList \leftarrow historyActionsList.add(lastAction)$$

$$actionsList \leftarrow actionsList[lastAction] + 1$$

$$historyTimesList \leftarrow historyTimesList.add(lastAction)$$

$$timesList \leftarrow timesList[lastAction] + 1$$

Une fonction **saveDynamicFile()** sera utilisée pour sauvegarder à chaque seconde les différentes informations dans la classe **DynamicAdaptationClass**. Les informations suivantes sont mis à jour :

$$goodShotsOnTarget \leftarrow goodShotsOnTarget + lastSecGoodShotsOnTarget$$

$$goodShotsOnNPC \leftarrow goodShotsOnNPC + lastSecGoodShotsOnNPC$$

$$badShots \leftarrow badShots + lastSecBadShots$$

$$goodPressedButtons \leftarrow goodPressedButtons + lastSecGoodPressedButtons$$

$$badPressedButtons \leftarrow badPressedButtons + lastSecBadPressedButtons$$

$$goodAnswers \leftarrow goodAnswers + lastSecGoodAnswers$$

$$badAnswers \leftarrow badAnswers + lastSecBadAnswers$$

Comme dit précédemment, sauf pour l'électroencéphalogramme où les données de $evaluatedBetaFrequency$ sont seulement mis à jour toutes les 10 secondes.

$$evaluatedBetaFrequency \leftarrow lastSecEvaluatedBetaFrequency$$

```
public void SaveRigidFile(actionString, actionIndex, time)
{
    FileStream file;

    if(File.Exists(pathRigidAdaptationData)) file = File.OpenRead(pathRigidAdaptationData);
    else
    {
        Debug.LogError("File not found");
        return;
    }

    BinaryFormatter bf = new BinaryFormatter();
    RigidAdaptationClass oldData = (RigidAdaptationClass) bf.Deserialize(file);
    file.Close();

    historyActionsList = oldData.historyActionsList.add(actionString);
    actionsList = oldData.actionsList;
    actionsList[actionIndex].add(1);
    historyTimesList = oldData.historyTimesList.add(time);
    timesList = oldData.timesList;
    timesList[actionIndex].add(int(time));

    if(File.Exists(pathRigidAdaptationData)) file = File.OpenWrite(pathRigidAdaptationData);
    else file = File.Create(pathRigidAdaptationData);

    RigidAdaptationClass data = new RigidAdaptationClass(historyActionsList, actionsList);
    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(file, data);
    file.Close();
}
```

FIGURE 6 – Exemple de code pour **saveRigidFile()**

2.3 Chargement des données

Une fonction **loadRigidFile()** sera utilisée pour récupérer les données au début de chaque énigme pour l'initialisation de la difficulté des actions lors de la génération de l'énigme. Aucune des valeurs n'est modifiées dans la classe **RigidAdaptationClass**.

Une fonction **loadDynamicFile()** sera utilisée pour récupérer la fréquence normal du joueur au début de chaque énigme. Les informations suivantes de la classe **DynamicAdaptationClass** sont aussi réinitialisées :

$$goodShotsOnTarget \leftarrow 0$$

$$goodShotsOnNPC \leftarrow 0$$

$$badShots \leftarrow 0$$

$$goodPressedButtons \leftarrow 0$$

$$badPressedButtons \leftarrow 0$$

$$goodAnswers \leftarrow 0$$

$$badAnswers \leftarrow 0$$

$$evaluatedBetaFrequency \leftarrow null$$

```
public void LoadRigidFile()
{
  FileStream file;

  if(File.Exists(pathRigidAdaptationData)) file = File.OpenRead(pathRigidAdaptationData);
  else
  {
      Debug.LogError("File not found");
      return;
  }

  BinaryFormatter bf = new BinaryFormatter();
  RigidAdaptationClass oldData = (RigidAdaptationClass) bf.Deserialize(file);
  file.Close();

  historyActionsList = oldData.historyActionsList;
  actionsList = oldData.actionsList;
  historyActionsList = oldData.historyTimesList;
  actionsList = oldData.timesList;
  historyTimesList = oldData.historyTimesList.add(time);
  timesList = oldData.timesList;
  timesList[actionIndex].add(int(time));

  return [historyActionsList, actionsList, historyTimesList, timesList];
}
```

FIGURE 7 – Exemple de code pour **loadRigidFile()**

3 Modélisation de l'utilisateur

3.1 Diagramme déroulement de l'énigme

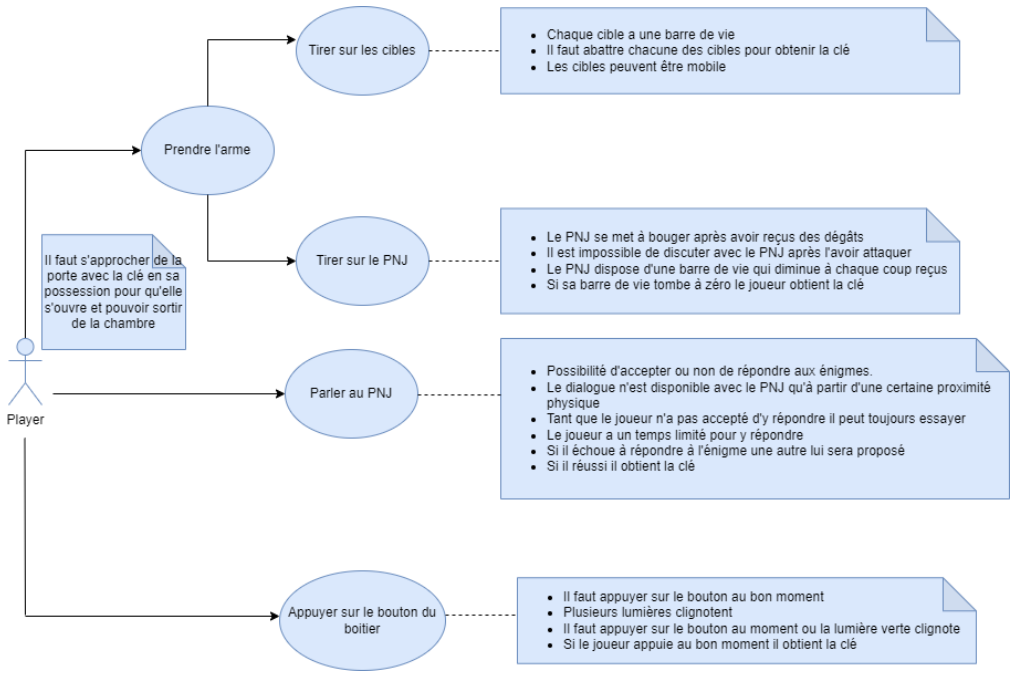


FIGURE 8 – Diagramme du joueur lors d'une énigme

3.2 Le joueur

Une classe **PlayerClass** sera utilisée pour stocker les données tirées des profils ainsi que les fonctions permettant de traduire les profils en choix pour la génération de la difficulté des énigmes et pour l'adaptation au cours de ces énigmes.

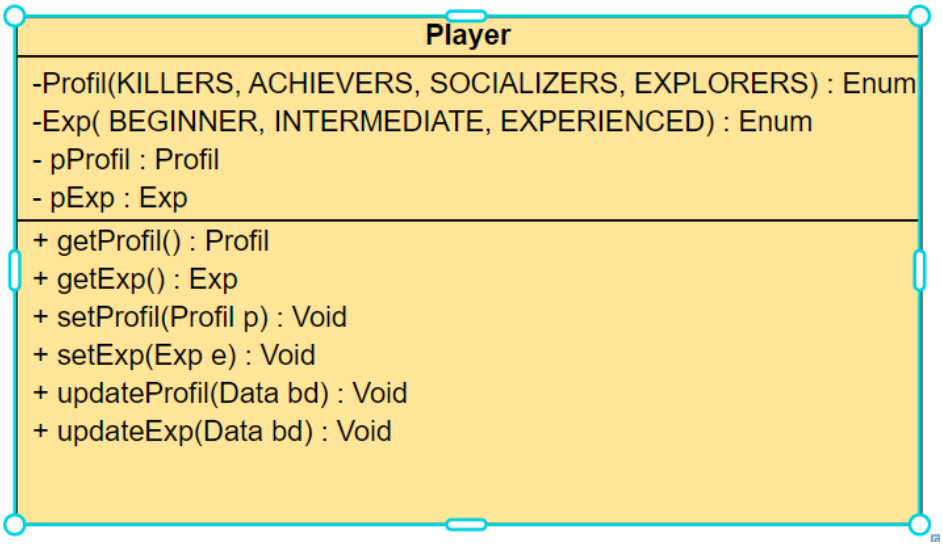


FIGURE 9 – UML : Player class

Une fonction **setExplorerDifficulty()** sera utilisée pour choisir le niveau de difficulté par rapport aux nombre d'actions **EXPLORER** effectuées sur l'ensemble des actions choisies depuis le début du jeu (profil d'adaptation rigide). Augmenter la difficulté augmentera le nombre d'interrupteurs à enclencher pour obtenir la clé et ouvrir la porte.

Une fonction **setWindowToPressButton()** sera utilisée pour choisir la fenêtre de temps pour presser le bouton dans le boîtier. Dépendra du nombre de pression ratées et réussies sur le bouton ainsi que la fréquence des ondes bêtas du joueur (profil d'adaptation dynamique).

Une fonction **setSocializerDifficulty()** sera utilisée pour choisir le niveau de difficulté par rapport aux nombre d'actions **SOCIALIZER** effectuées sur l'ensemble des actions choisies depuis le début du jeu (profil d'adaptation rigide). Augmenter la difficulté augmentera le nombre de questions à répondre pour obtenir la clé et ouvrir la porte.

Une fonction **setAnsweringTime()** sera utilisée pour choisir le temps disponible pour répondre à une question. Dépendra du nombre de questions ratés et réussies du PNJ ainsi que la fréquence des ondes bêtas du joueur (profil d'adaptation dynamique).

Une fonction **setAchieverDifficulty()** sera utilisée pour choisir le niveau de difficulté par rapport aux nombre d'actions **ACHIEVER** effectuées sur l'ensemble des actions choisies depuis le début du jeu (profil d'adaptation rigide). Augmenter la difficulté augmentera le nombre de cibles à toucher pour obtenir la clé et ouvrir la porte.

Une fonction **setAccuracy()** sera utilisée pour choisir la précision du pistolet. Dépendra du nombre de tirs ratées et réussi sur la cible ainsi que la fréquence des ondes bêtas du joueur (profil d'adaptation dynamique).

Une fonction **setKillerDifficulty()** sera utilisée pour choisir le niveau de difficulté par rapport aux nombre d'actions **KILLER** effectuées sur l'ensemble des actions choisies depuis le début du jeu (profil d'adaptation rigide). Augmenter la difficulté augmentera la nombre de fois qu'il est nécessaire de toucher le PNJ pour obtenir la clé et ouvrir la porte.

Une fonction **setPNJSpeed()** sera utilisée pour choisir la vitesse du PNJ lorsqu'il cherche à s'enfuir après s'être fait tirer dessus. Dépendra du nombre de tirs ratés et réussis sur le PNJ ainsi que la fréquence des ondes bêtas du joueur (profil d'adaptation dynamique).

3.3 Les statistiques du joueur

Pour la modélisation du score on aura une barre en haut à gauche du joueur qui augmentera à chaque action *Achiever* du joueur. Pour implémenter la barre de score on utilisera un *GameObject* : *Image* avec un slider pour régler la progression de la barre et si la barre de score atteint 100% on augmente d'un point le profil *Achiever*. On aura donc dans un *Canvas* qui s'affichera à l'écran pendant la partie de l'utilisateur un indicateur de texte qui indiquera au joueur si il a la clé lui permettant d'ouvrir la porte ainsi que sa barre de score :

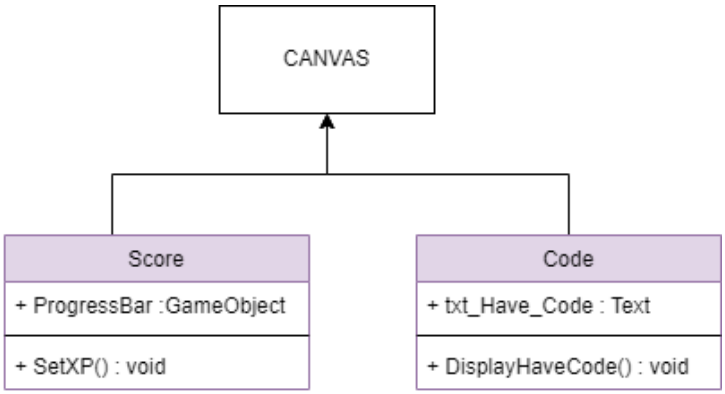


FIGURE 10 – Diagramme de classe pour les statistiques du joueur

4 Modélisation de l'environnement

4.1 Environnement 3D

Le design et le graphisme du jeu n'étant pas l'objet de notre étude, nous utiliserons des assets déjà existant pour construire le jeu plus rapidement. Nous allons utiliser des assets pour les éléments du jeu suivant :

- L'environnement, c'est à dire les différentes chambres 3D dont il faut sortir.
- Les armes.
- Les personnages non-joueurs

Pour la construction globale de notre environnement 3D et de nos mécaniques de jeu nous utiliserons un projet de UnityLearn qui se nomme "FPS Microgame" :

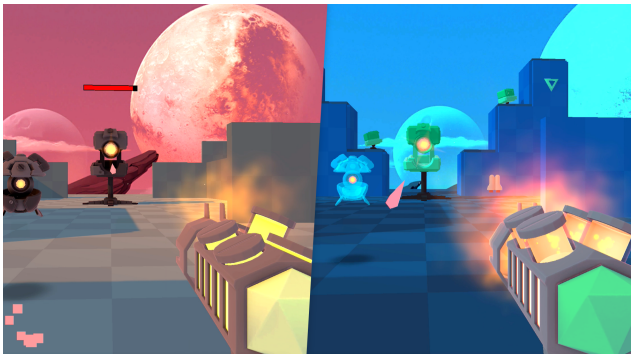


FIGURE 11 – Image du projet FPS Microgame

Le modèle FPS Microgame est un jeu de tir à la première personne en 3D accessible et qui est facilement modifiable et personnalisable. Voici un diagramme UML représentant l'architecture global de ce projet :

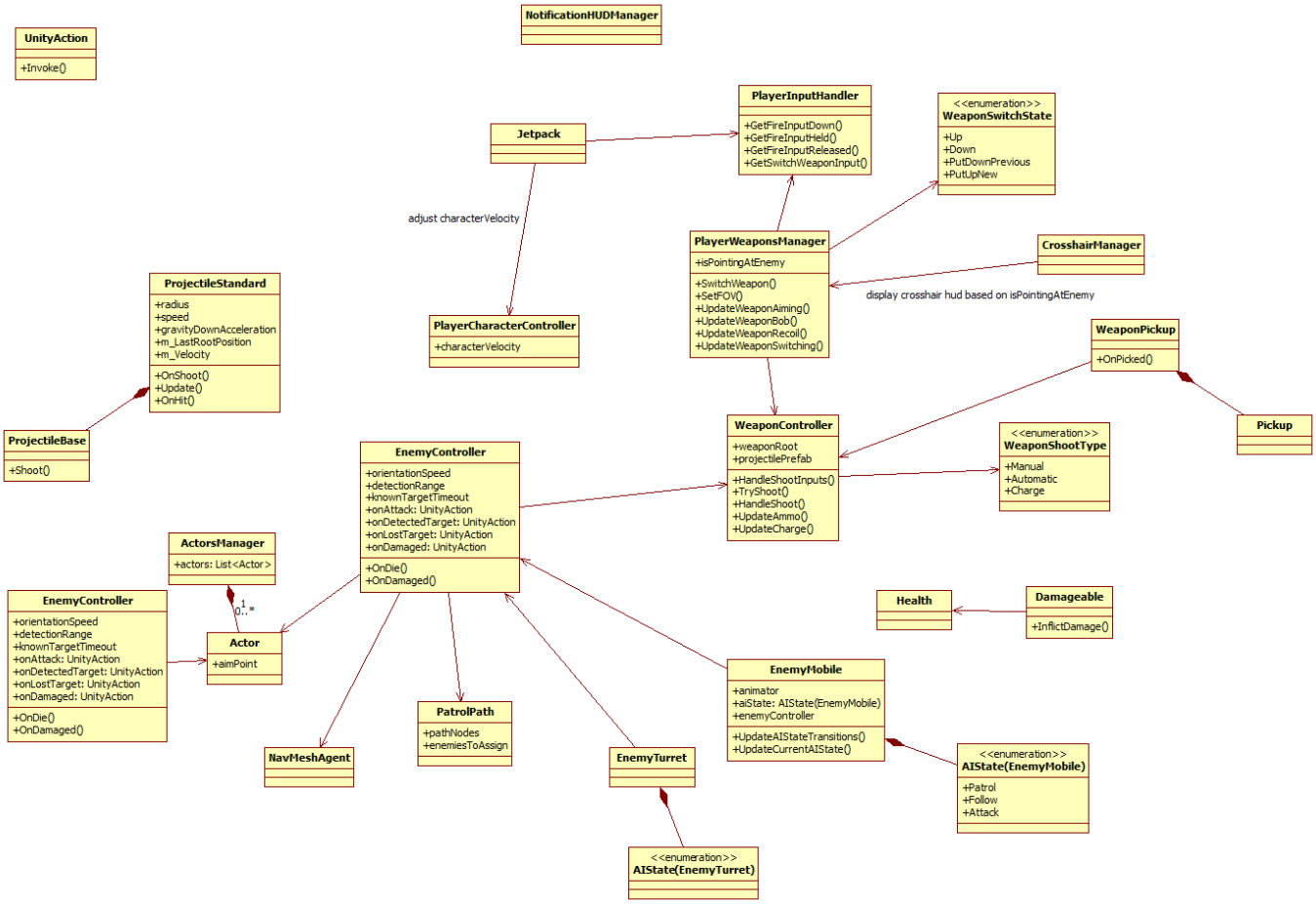


FIGURE 12 – Diagramme UML du projet FPS Microgame

4.2 Salle des énigmes

Une énigme se déroulera dans une salle qui lui est associé. Le jeu consistera en une succession de salle, à chaque énigme que le joueur aura effectué on aura une génération dynamique pour la prochaine salle dans laquelle le joueur entrera. La prochaine salle sera déterminée en fonction des choix et de la performance du joueur sur la salle précédente. La génération de salle peut se représenter de cette façon :

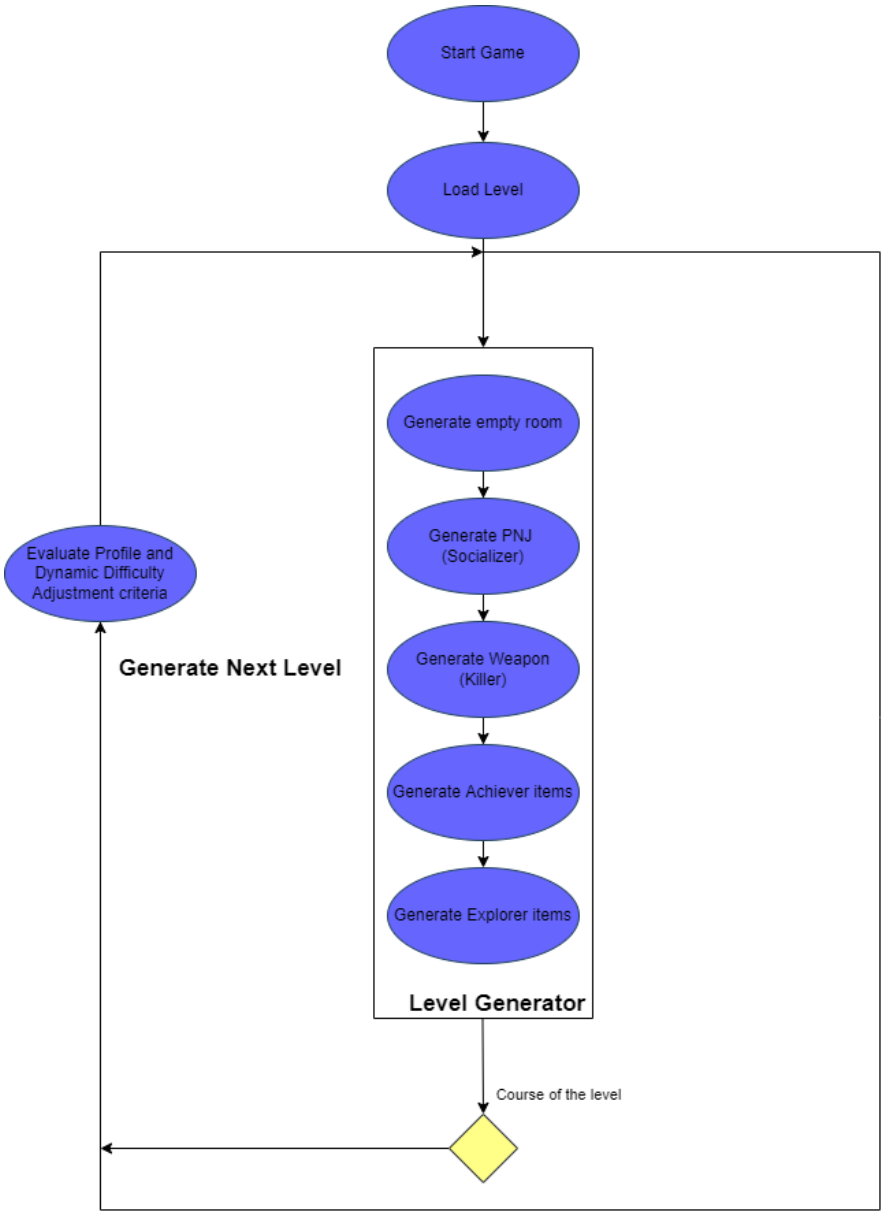


FIGURE 13 – Diagramme d'activité séquentiel représentant la génération de niveau

4.3 Personnage non-joueur : PNJ

Dans chaque énigme on aura un personnage non jouable à qui le joueur pourra s'adresser. On associera un script à ce PNJ qui déclenchera un *Canvas* de texte pour lui proposer un dialogue débouchant sur l'obtention de la clé de la porte si le joueur répond bien à la question. Le *Canvas* de texte s'activera si le *BoxCollider* du PNJ détecte celui de notre joueur.

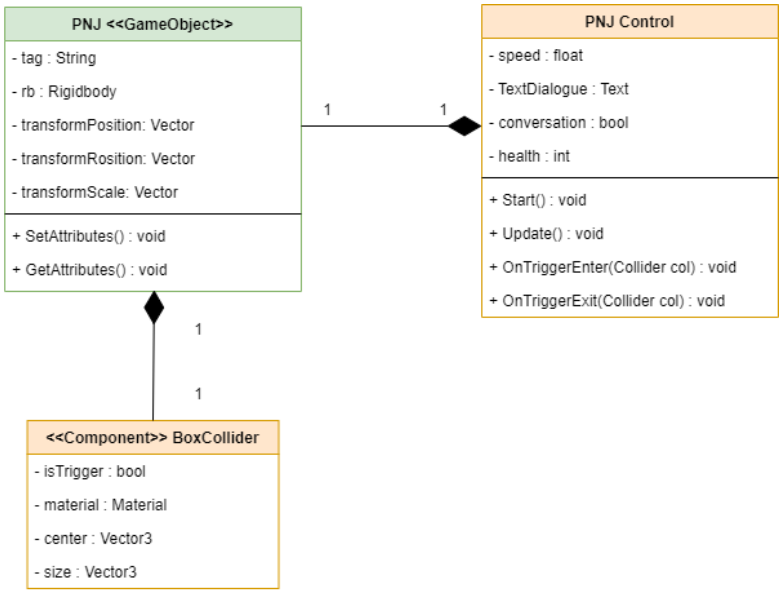


FIGURE 14 – Diagramme de classe pour le personnage non-joueur

Le PNJ aura un attribut speed pour sa vitesse de déplacement en cas d'attaque par le joueur ainsi qu'un booléen indiquant si il est ou non dans une conversation avec le joueur.

Pour les cibles des énigmes on réutilisera les ennemis du FPS Microgame.

4.3.1 Classe Dialogue

Cette classe permettra de gérer le dialogue avec le PNJ, elle comportera plusieurs attributs :

— Le texte correspondant à l'éngime proposé par le PNJ
— Un niveau de difficulté pour l'énigme
— Un temps de réponse que le joueur aura pour répondre à la question
— Une mode de réponse associé à la question (réponse unique, questionnaire à choix multiples...)

Le dialogue sera donc structuré de cette manière :

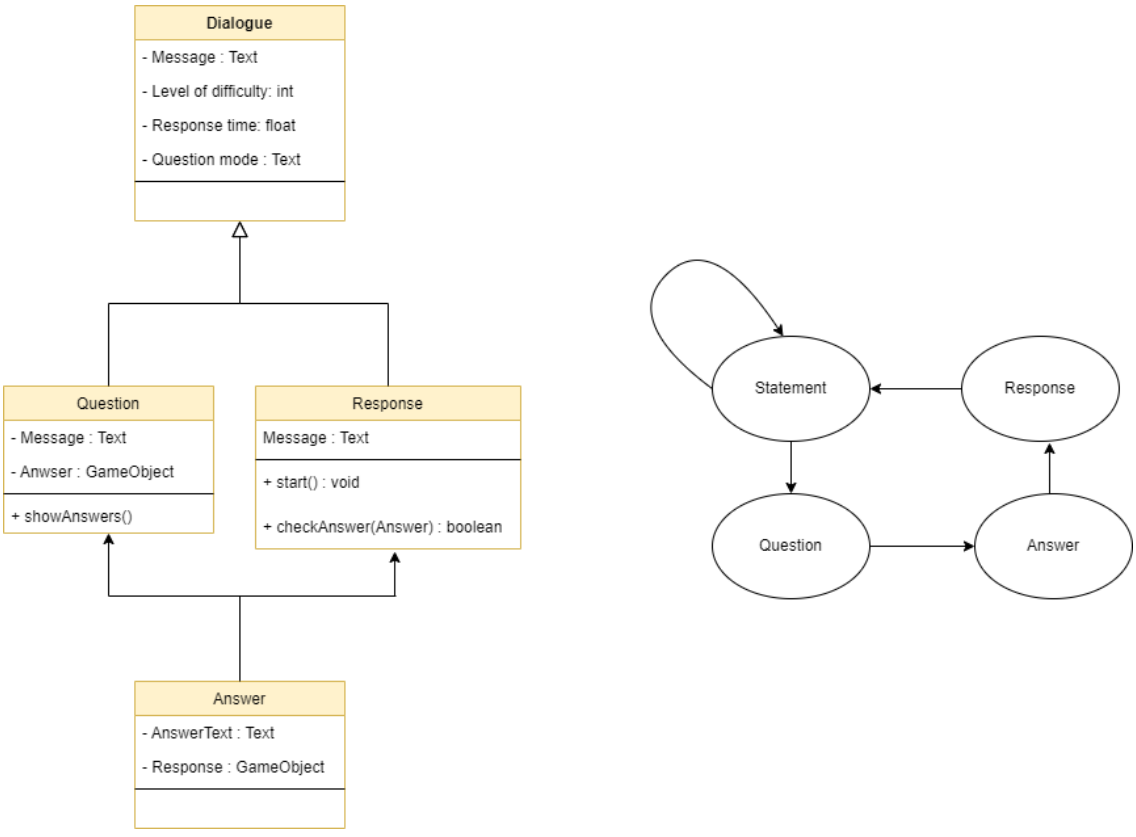


FIGURE 15 – Diagramme de classe pour le dialogue

4.3.2 Health

Il existe une classe Health, qui représente la santé des PNJ. La classe Health est une classe simple qui contient la santé maximale, la santé actuelle, si le PNJ est invincible, une méthode pour soigner ou subir des

dégâts. Elle contient également un indicateur *IsCritical*, qui est vrai lorsque la vie passe en dessous d'un certain ratio. Pour diminuer la santé d'un PNJ, il y a deux façons :

— Appeler la méthode *OnDamaged* de la classe Health.
— Par le biais de Damageable, appelez la méthode *OnDamaged* de la classe Health.

Le numéro 2 est utilisé pour les projectiles.

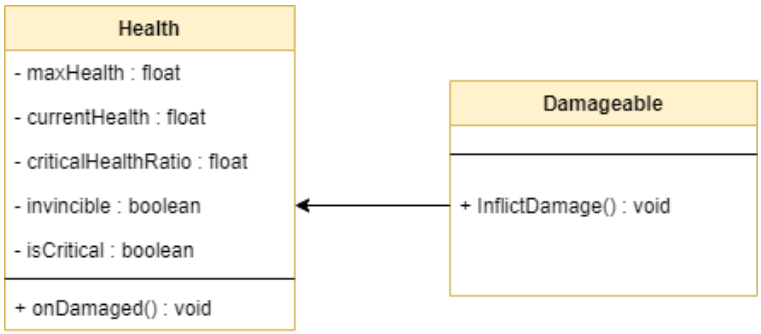


FIGURE 16 – Diagramme de classe pour la santé

4.4 Armes

Le joueur pourra prendre en sa possession une arme, un pistolet dans notre cas. L'arme possédera un viseur avec lequel le joueur pourra zoomer et dézoomer pour augmenter sa précision. Le joueur pourra tirer autant de fois qu'il le veut une fois en possession de l'arme.

Les projectiles qui sortent de l'arme sont des *Meshes* en forme de bâton avec un matériau. Ils ont tous le même script Projectile, avec des paramètres tels que la vitesse, les dégâts, s'ils sont affectés par la gravité, ainsi que SFX (effet sonore) et VFX (effet visuel, c'est-à-dire effets de particules) à l'impact.

Il existe deux classes, ProjectileBase et ProjectileStandard. ProjectileBase définit la position initiale, la direction sur Shoot :

```
public class ProjectileBase : MonoBehaviour
{
    public GameObject owner { get; private set; }
    public Vector3 initialPosition { get; private set; }
    public Vector3 initialDirection { get; private set; }
    public Vector3 inheritedMuzzleVelocity { get; private set; }
    public float initialCharge { get; private set; }

    public UnityAction onShoot;

    public void Shoot(WeaponController controller)
    {
        owner = controller.owner;
        initialPosition = transform.position;
        initialDirection = transform.forward;
        inheritedMuzzleVelocity = controller.muzzleWorldVelocity;
        initialCharge = controller.currentCharge;

        if (onShoot != null)
        {
            onShoot.Invoke();
        }
    }
}
```

FIGURE 17 – Code de la classe des projectiles

Lorsque quelqu'un appelle la méthode Shoot dans ProjectileBase, le callback OnShot dans ProjectileStandard est appelé.

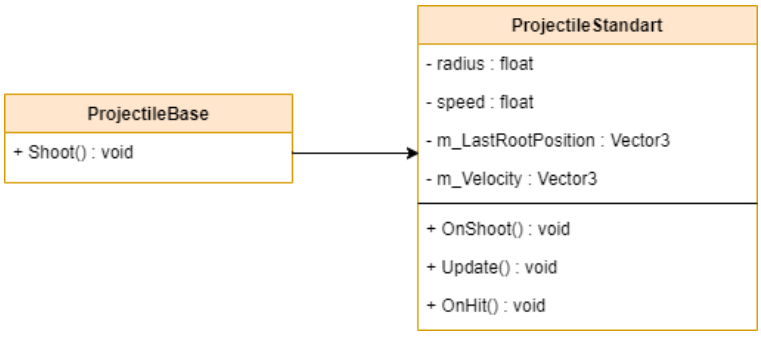


FIGURE 18 – Diagramme de classe des projectiles

Lorsque l'utilisateur tire, plusieurs fonctions sont appelées et le callback onShoot est appelé. Dans OnShoot, sont définis *shootTime*, *velocity*, *m_LastRootPosition*... En particulier *m_LastRootPosition* sert à suivre une position et est utilisé dans la détection de collision :

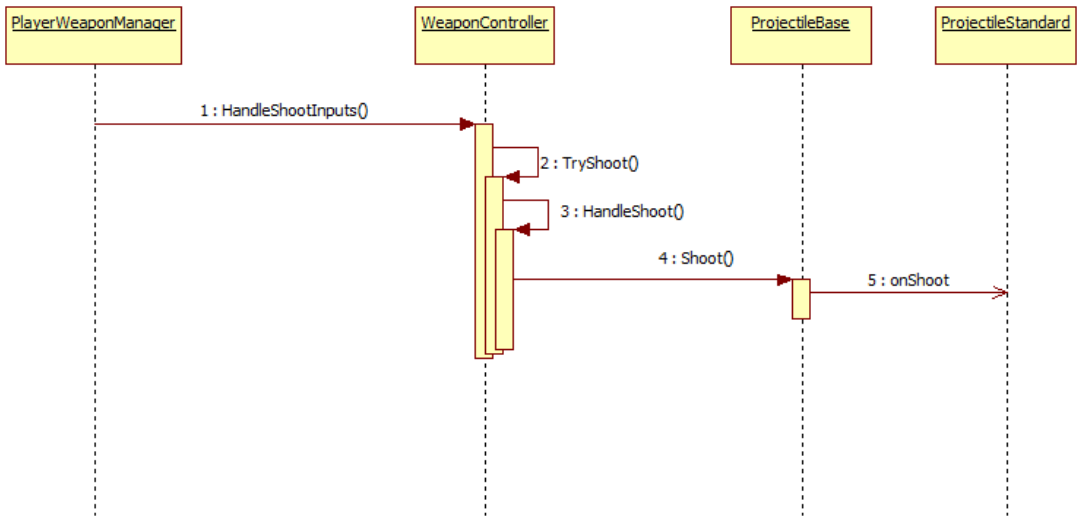


FIGURE 19 – Diagramme déroulement d'un tir

La détection de coup se fait en trouvant tous les objets dans sa trajectoire entre sa dernière position connue et sa position actuelle. Les dommages directs sont simples, on recherche l'objet avec lequel le projectile est entré en collision, on obtient son composant *Damageable* et on appelle la méthode pour infliger les dégâts :

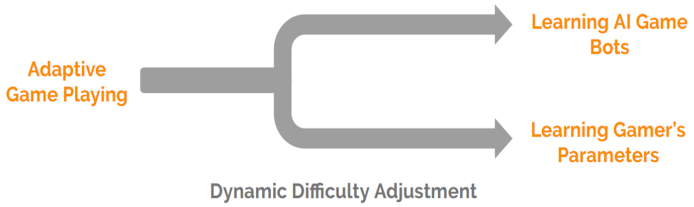
```
Damageable damageable = collider.GetComponent<Damageable>();
if (damageable)
{
  damageable.InflictDamage(damage, false, m_ProjectileBase.owner);
}
```

FIGURE 20 – Code pour infliger des dégats

5 Mécaniques d'adaptation

5.1 Ajustement dynamique de la difficulté (DDA)

Le concept de DDA, qui consiste à fournir un jeu réactif à l'utilisateur, est généralement réalisé par deux approches de base :



Nous nous intéresserons dans notre projet à l'adaptation via les Learning Gamer's Parameters. En mettant constamment à jour divers paramètres via une série d'instructions prédéfinies et en gardant une trace des réponses du joueur, la difficulté du jeu peut être adaptée au niveau de compétence du joueur. De tels paramètres de jeu (par exemple, la vitesse du PNJ si on l'attaque, la vitesse des projectiles à détruire, etc.) peuvent être modifiés dynamiquement pendant le jeu pour manipuler le niveau de difficulté du jeu.

Les paramètres du joueur sont calculés à l'aide de certaines statistiques (par exemple, le nombre de tentatives de résolution d une énigme, le niveau de concentration à chaque niveau..), puis transmis à un classificateur d'apprentissage.

L'adaptation de notre jeu se fera principalement sur la difficulté. A la fin de chaque énigme le joueur aura un profil déterminé par le Gameplay qu'il aura préféré/choisi précédemment, la difficulté associé à ce/ces types de gameplay sera alors augmenter pour offrir un meilleur défis sur ce qu'aime le joueur : c'est ce qu'on appellera **l'adaptation rigide**. Durant une énigme la difficulté pourra être adapter en temps réel pour le joueur si on juge qu'il est en trop grande difficulté ou que cela est trop facile pour lui, : c'est ce qu'on appellera **l'adaptation dynamique**. L'adaptation lors d'une partie se déroulera de cette manière :

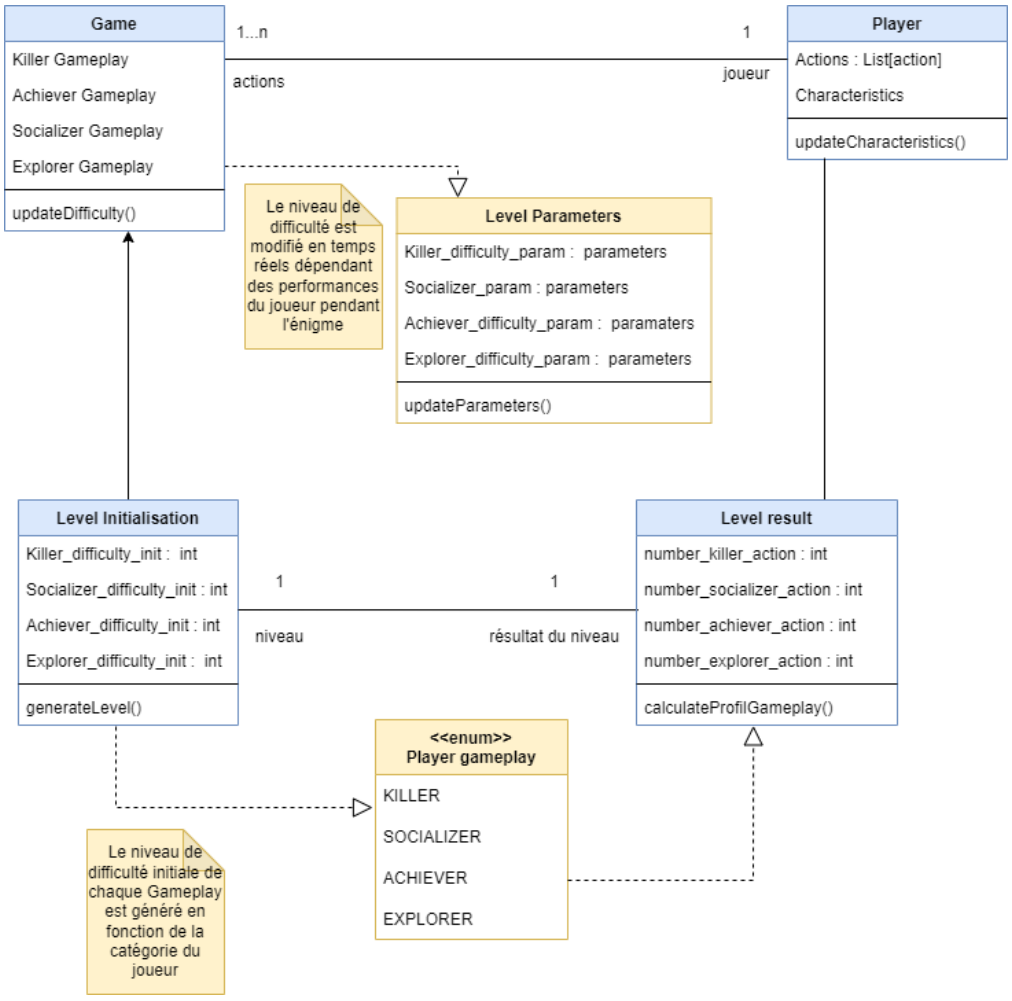


FIGURE 21 – Diagramme de classe montrant les différents types d'adaptations lors d'une partie

Pour déterminer le Gameplay que le joueur préfère on procédera de la manière suivante :

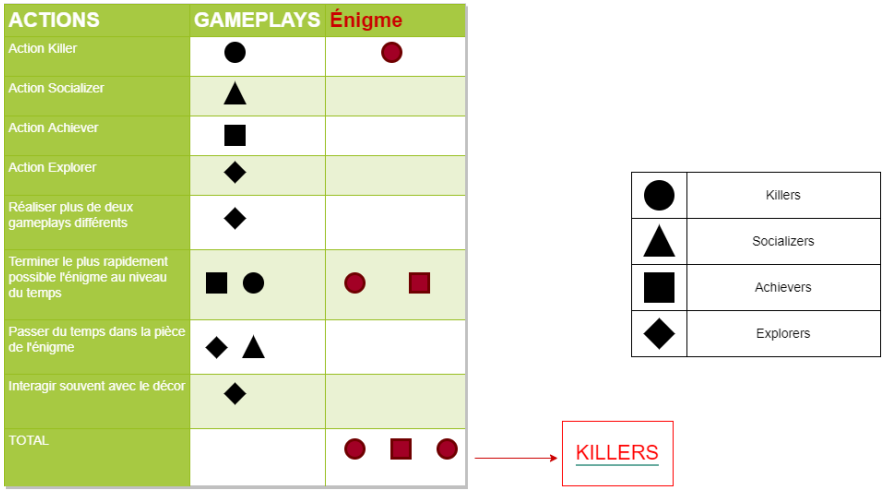


FIGURE 22 – Tableau représentant la sélection du profil concernant le gameplay

En prenant par exemple des valeurs de 1 pour chaque symbole pour le tableau exemple du dessus, on déterminera le profil du gameplay de l'utilisateur de cette manière :

$$2◯ > 1□ > 0\ △ \geqq 0◇$$

Donc le profil *Killer* (◯) serait sélectionné et on augmentera la difficulté associé à ce gameplay sur l'énigme qui suit

Si il y avait une égalité, par exemple avec les valeurs suivantes :

$$2◯ = 2□ > 0\ △ \geqq 0◇$$

Donc le profil *Killer* (◯) et le profil *Achiever* (□) seront sélectionnés et on augmentera la difficulté pour ces deux gameplays sur l'énigme suivante.

Architecture du modèle d'adaptation :
Tout d'abord on place le joueur dans une chambre ou les énigmes sont de niveau "facile". Cette phase sert a apprendre le joueur les commandes et l environnements de notre jeux. (Les règles de jeux). Faire de l'adaptation a ce stade n aurait pas beaucoup d effet.Une fois cette partie terminer on fait une partie de "calibration" ou selon les performances du joueur on le placera dans une configuration du jeux selon la classe d'expérience correspondante(Débutant, Intermédiaire, Expert). L'adaptation dynamique n'aurait pas de sens si on fourni au joueur une difficulté complètement inadapté. Ces classes de niveau de difficulté moyenne selon la classe, sont le point de départ pour chaque joueur.

Partie de calibration :
Des statistiques préalable vont être faites dans lesquels on enregistrera certains paramètres qui nous permettent de juger la performances du joueur. On prendra le temps qu'il a fallu au joueur pour compléter une

manière de résoudre l'énigme. Par exemple pour le jeux de cible, le chronomètre débute après le premier tire, et s'arrête quand la dernière cible à été atteinte. On fait joueur 15 joueurs notre jeux. 5 qui n'ont pas d'expériences dans les jeux vidéos. 5 intermédiaires et 5 expert en FPS. Ainsi nous pourrons déterminer les tranches de paramètres approximatif du jeux auxquels appartiennent les joueurs. Ces données seront utilisées lors de la phase de calibration pour ajuster les paramétres de bases suivant :

SOCIALIZER	EXPLORER	ACHIEVER	KILLER
Nb de questions	Nb d'intérupteurs	Nb de cibles	Nb de coups nécessaire

On souhaite adapter les paramètres pour que toute classe de joueur prenne 30 secondes pour résoudre une énigme par exemple.

Adaptation rigide :
La première étape de l'adaptation de la difficulté est l'adaptation rigide. Quand on parle d'adaptation rigide on parle de la difficulté prédéfini lorsque le joueur entre dans une nouvelle salle. En effet même si la difficulté peut s'adapter au cours de l'énigme que le joueur est en train de réaliser (avec **l'adaptation dynamique** lorsqu'il entre dans une nouvelle salle une difficulté est déterminé en fonction des choix de gameplay de l'utilisateur sur les dernières énigmes. Elle se fait par rapport au goût du joueur et elle est statique. C'est à dire à chaque niveau (chambre) suivant, on augmente la difficulté de la manière de résoudre l'énigme qu'a choisit le joueur précédemment. C'est paramètres ne varient pas dynamiquement pendant le jeu et reste les même durant toute l'énigme. Voici les paramètres influencés :

SOCIALIZER	EXPLORER	ACHIEVER	KILLER
Nb de questions	Nb d'intérupteurs	Nb de cibles	Nb de coups nécessaire

Algorithm 1 updateRigidDifficulty()

$if(Player.getProfil() == SOCIALIZER)\{;$
$SetSocializerDifficulty(GetSocializerDifficulty()++);$
$\}$
$if(Player.getProfil() == EXPLORER)\{;$
$SetExplorerDifficulty(GetExplorerDifficulty()++);$
$\}$
$if(Player.getProfil() == ACHIEVER)\{;$
$SetAchieverDifficulty(GetAchieverDifficulty()++);$
$\}$
$if(Player.getProfil() == KILLER)\{;$
$SetKillerDifficulty(GetKillerDifficulty()++);$
$\}$

Ce sont ces opérations Set qui vont directement manipuler le jeu pour ajuster la difficulté. SetAchiever-Difficulty() par exemple, ajuste la vitesse des cibles.

Ensuite lors de la construction de chaque niveau on fait appel à chaque Guetteur pour savoir le nombre de cibles, d'interrupteurs a construire...

Adaptation dynamique :

L'étape suivante est la phase d'adaptation fine et dynamique de la difficulté. Ainsi chaque des quatre manière de résoudre l'énigme a ça propre formule qui permet d'ajuster la difficulté en fonction des performances du joueur. L'adaptation se fait donc de manière indépendante pour les quatre profils. En revanche l'équation pour l'adaptation de la difficulté est inspirée de la même idée générale :

$$D_{t+1} = -1 * dN_t - dC_t + D_t$$

dN_t étant la variation du nombre de tentatives à résoudre l'énigme (par exemple la variation du nombre du nombre de badShots sur les cibles) et dC_t la variation du niveau de concentration par rapport à un temps t précédant (par exemple, il y a 10 secondes).

En pratique : La manière de résoudre l'énigme de manière ACHIEVER est celle de tirer sur les cibles ainsi : On part par exemple d'un joueur qui est classé comme ayant un expérience intermédiaire. $D_0 = 5$ initialement. Dans les premières dix secondes il rate 4 fois la cible et ça concentration est au niveau 2. $D_1 = 5$ En suite il fait 2 tentatives de moins en ratant 2 balles de moins dans les 10 prochaines secondes donc $dV = -2$, ça concentration baisse de 1 donc $dC = -1$. Donc $D_2 = 8$. Le niveau de difficulté doit augmenter de 3 et ainsi les paramètres de difficulté augmentent aussi. Ainsi la vitesse de déplacement des cibles augmente. La clé est que le joueur ne se rend pas compte que la difficulté varie, on pourra donc rajouter ou enlever des cibles dans la prochaine salle.

Algorithm 2 updateDifficulty()

$D = getDifficulty()$;
$N = diffN()$;
$C = diffC()$;
$newD = D - C - N$;
$setD(newD)$;
$if(D <= newD) : Difficulty++()$;
$else\ :\ Difficulty--()$

Algorithm 3 diffN() : int

$N = getN()$;
$newN = this.badShots$;
$setN(newN)$;
$return\ (newN - N)$;

Algorithm 4 Difficulty++()

$Cible.MovementSpeed+ = 1$

Voici un tableau résumant les paramètres affécté par l'adaptation dynamique dans le temps :

SOCIALIZER	EXPLORER	ACHIEVER	KILLER
Temps de réponse restant	Taille de la fenêtre de temps	Vitesse des cibles	Vitesse du PNJ

Références

[1] J. Baumeister et al. « Influence of phosphatidylserine on cognitive performance and cortical activity after induced stress ». In : *Nutritional Neuroscience* 11.3 (1er juin 2008). Publisher : Taylor & Francis _eprint : https ://doi.org/10.1179/147683008X301478, p. 103-110. issn : 1028-415X. doi : 10.1179/147683008X301478. url : https://doi.org/10.1179/147683008X301478.

[2] *Bitalino*. url : https://bitalino.com/documentation#documention---tabs.

[3] Jaziar Radianti et al. « A systematic review of immersive virtual reality applications for higher education : Design elements, lessons learned, and research agenda ». In : *Computers & Education* 147 (1er avr. 2020), p. 103778. issn : 0360-1315. doi : 10.1016/j.compedu.2019.103778. url : https://www.sciencedirect.com/science/article/pii/S0360131519303276.

[4] *Richard A. Bartle : Players Who Suit MUDs*. url : https://mud.co.uk/richard/hcds.htm.