ISIMA ZZ2 2023-2024

Murder Project

de ABDUL-SALAM Sami, DUPOIS Thomas et
JOUEN Martin

Janvier 2024

Rapport présenté dans le cadre du cours d'Architecture logicielle et qualité
Enseignant : ANTUNES Benjamin

Table des matières

Table des figures

1 Introduction

L'objet de ce projet est de mettre en œuvre un petit logiciel proposant une simulation multi-agents.

Dans notre cas, on simulera le jeu du *Murder* étant composé de plusieurs agents ayant chacun des rôles différents ainsi que des comportements différents en fonction du contexte. Le principe du jeu ainsi que les rôles des différents agents seront décrits ultérieurement dans ce rapport.

On présentera dans un premier temps le cahier des charges afin de décrire ce qui est souhaité au niveau de la simulation et des agents.

Puis, on verra les outils utilisés lors de la réalisation de ce projet.

Ensuite, on analysera un diagramme de classe correspondant à la structure du code.

Enfin, on notera des cas rencontrés dans les résultats de la simulation.

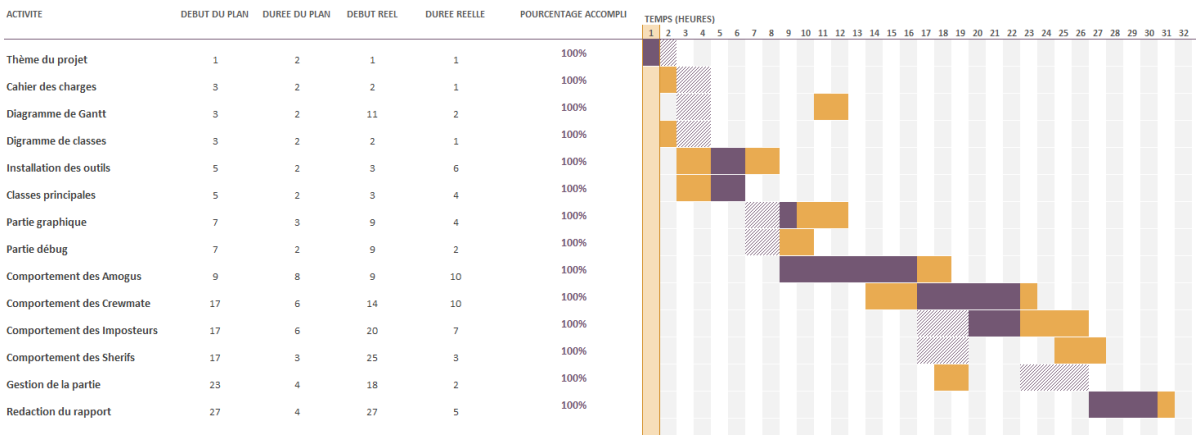
2 Cahier des Charges

L'objectif de ce travail pratique est d'observer l'évolution d'un phénomène ou d'une situation à travers une simulation.
Ce projet constitue une simulation du jeu du *murder*.
Nous avons un nombre défini de « Amogus » parmi lesquels sont présents x crewmates, y sherifs et se cachent z importeurs.
L'objectif des imposteurs est d'éliminer l'ensemble des autres Amogus tandis que le rôle des shérifs est de démasquer les imposteurs afin de les tuer.
Concernant les crewmates, l'objectif est de survivre mais aussi d'accomplir un nombre v de tâches.
La simulation se termine lorsque tous les crewmates sont morts ou lorsque tous les imposteurs ont été tués.
Lorsque un crewmate termine ses tâches, il devient shérif et obtient la possibilité de tuer un imposteur.
En début de partie : Les tâches doivent être générées et disposées aléatoirement dans la map (de préférence, on fera en sorte que les tâches ne soient pas trop proches les unes des autres).
Les crewmates, imposteurs et shérifs possèdent chacun un champ de vision et un champ d'interaction.
Le champ de vision leur permet d'être conscient des autres Amogus qui les entourent. Le champ d'interaction est le rayon dans lequel les Amogus peuvent interagir avec des tâches ou tuer un autre Amogus.
On définit un nombre initial de crewmates, de shérifs et d'imposteurs.
Pour chaque crewmate est attribué une liste de tâches à réaliser. Une même tâche peut être réalisée par différents crewmates.
Un crewmate réalise ses tâches selon sa position dans la map et son champ de vision. S'il repère une tâche non réalisée aux alentours, il se dirige vers cette dernière et l'effectue.
A chaque instant, les crewmates ont un pourcentage de chance de se déplacer aléatoirement dans la zone et on aussi un pourcentage de chance d'aller effectuer leur première tâche.
Une tâche nécessite un certain temps pour être achevée, pendant ce temps, le crewmate réalisant une tâche ne peut pas se déplacer.
Les shérifs parcourent la map de manière aléatoire et agissent en conséquence des évènements qu'ils observent.
Les imposteurs se déplacent, soit aléatoirement, soit vers les tâches de manière à rencontrer des crewmates ou shérifs isolés. Si un imposteur a une faible probabilité d'être repéré, alors il a une forte probabilité de tuer le crewmate en question.
Chaque crewmate et shérif possède une jauge de suspicion pour chaque autre joueur. Une fois shérifs, si la barre de suspicion pour un x joueur est trop haute, alors le shérif peut avoir une forte probabilité de tuer ce joueur.
La jauge de suspicion augmente en fonction du nombre de cadavres découverts et des Amogus aux alentours.
La simulation prends fin à partir du moment où tous les imposteurs ont été tué,

ou tous les crewmates ont été tués.

A noté : Les Crewmates sont en bleu, les Shérifs sont en vert et les Imposteurs sont en rouge. Les tâches quant à elles, sont représentées par des leviers.

3 Diagramme de Gantt



ACTIVITE	DEBUT DU PLAN	DUREE DU PLAN	DEBUT REEL	DUREE REELLE	POURCENTAGE ACCOMPLI
Thème du projet	1	2	1	1	100%
Cahier des charges	3	2	2	1	100%
Diagramme de Gantt	3	2	11	2	100%
Digramme de classes	3	2	2	1	100%
Installation des outils	5	2	3	6	100%
Classes principales	5	2	3	4	100%
Partie graphique	7	3	9	4	100%
Partie débug	7	2	9	2	100%
Comportement des Amogus	9	8	9	10	100%
Comportement des Crewmate	17	6	14	10	100%
Comportement des Imposteurs	17	6	20	7	100%
Comportement des Sherifs	17	3	25	3	100%
Gestion de la partie	23	4	18	2	100%
Redaction du rapport	27	4	27	5	100%

FIGURE 1 – Diagramme de Gantt via Excel

4 Outils utilisés

4.1 C++

Nous avons pris la décision d'utiliser le langage C++ pour la conception de ce projet pour sa vitesse d'execution. (plus rapide que le Java).

4.2 Git

Afin d'optimiser le travail d'équipe, nous avons utilisé git afin de travailler en parallèle sur des ressources différentes du projet. Le projet en lui-même est stocké sur GitHub.

4.3 CMake

CMake est un outil de génération de fichiers de configuration pour la construction de logiciels. Il simplifie le processus de compilation en générant des Makefiles ou des projets pour différents environnements de développement, permettant aux développeurs de compiler leur code de manière cohérente sur différentes plates-formes.

4.4 Raylib

Raylib est une bibliothèque graphique légère et simple à utiliser pour le langage de programmation C et C++. Elle est conçue pour faciliter le développement de jeux et d'applications graphiques. Raylib offre des fonctionnalités telles que la gestion de fenêtres, le dessin 2D et 3D, l'audio, l'entrée utilisateur, et d'autres, le tout avec une syntaxe simple. Son objectif est de permettre aux développeurs de créer rapidement des applications graphiques tout en restant accessible aux débutants.

4.5 Mersenne Twister

Le Mersenne Twister est un algorithme de génération de nombres pseudo-aléatoires largement utilisé en informatique. Il se distingue par sa longue période et sa qualité de distribution, ce qui en fait un choix populaire pour diverses applications nécessitant des nombres aléatoires. Cependant, il est déterministe, ce qui signifie que connaître son état interne permet de prédire les valeurs générées. Nous avons donc décidé d'utiliser la librairie Mersenne Twister afin de d'avoir une génération de nombres pseudo-aléatoire de qualité.

5 Classes implémentées

5.1 Diagramme de classe

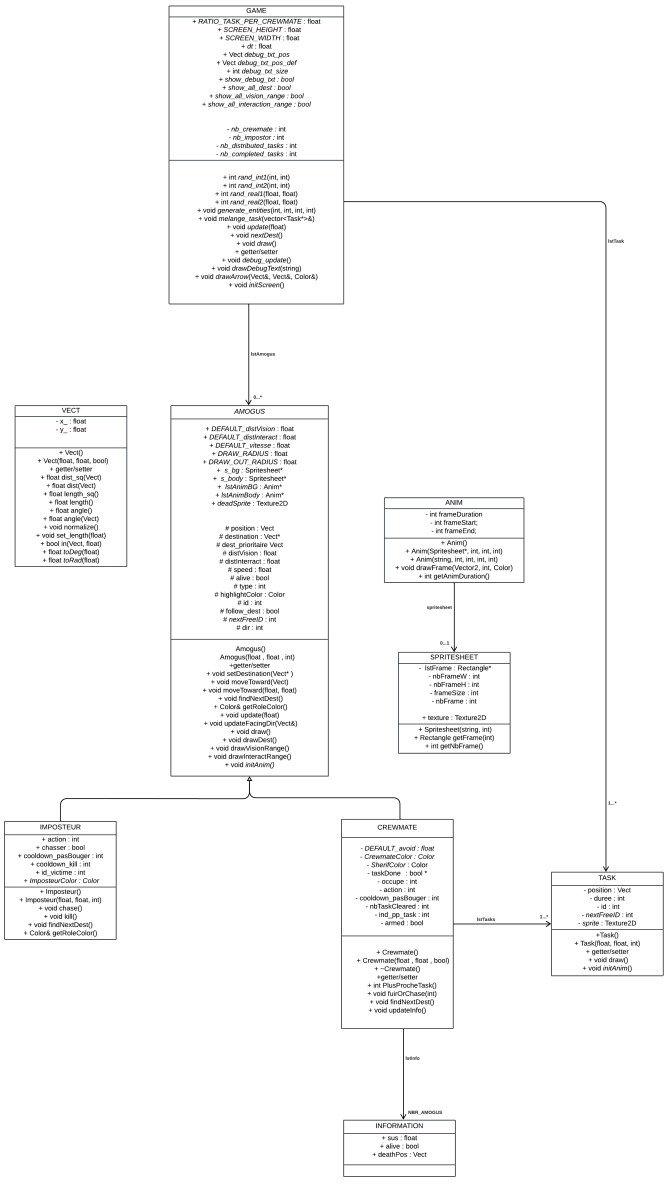


FIGURE 2 – Diagramme de classe UML via Lucidspark

5.2 Description des classes

Voici une description du diagramme décrivant les divers informations et comportements des classes implémentées dans ce projet.
Nous allons analyser dans un premier temps les classes qui représentent les agents de la simulation.
Puis, l'analyse se portera sur les classes tierces, celles qui complètent la structure de donnée et la mise en forme du logiciel.
Au niveau du diagramme, une flèche d'une classe à l'autre correspond à une agrégation, c'est-à-dire une association entre deux classes de sorte que pour les classes A et B telles que : $A \rightarrow B$, alors cela signifie qu'une instance de A contient un certain nombre d'instances de B (par exemple, la classe A possède une liste de type B).

5.2.1 Classes Agents

Voici les classes concernées :

— **Amogus**

— **Crewmate**

— **Imposteur**

La classe **Amogus** décrit l'information et le comportement de toutes les entités de la simulation, qu'importe son rôle. Cette classe sera mère des classes Crewmate et Imposteur qui elles, possèdent des spécificités puisqu'elles jouent des rôles différents.
Au niveau de ces attributs (concernant l'information), on a toute une partie concernant l'aspect visuel de l'entité, leurs types correspondent à des classes qui seront décris dans les classes tierces :

— `s_bg` : Sprite de la visière de l'Amongus.

— `s_body` : Sprite du corps de l'Amongus.

— `lstAnimBG` : Rectangle indiquant la position des sprites de la visière.

— `lstAnimBody` : Rectangle indiquant la position des sprites du corps.

— `deadSprite` : Image de mort d'un Amongus.

— `highlightColor` : Couleur de l'Amongus.

Puis, on a une partie dédiée aux caractéristiques physiques de la simulation telles que :

— `position` : Position actuelle de l'entité

— `destination` : Destination de priorité normale

— `dest_prioritaire` : Destination de plus grande priorité

— `distVision` : Champ de vision

— `distInterract` : Champ d'interraction

— `speed` : Vitesse de déplacement

— `alive` : Booléen décrivant l'état de vie de l'entité

— `type` : Entier décrivant si l'entité est de classe Crewmate ou de classe Imposteur

— `follow_dest` : Booléan décrivant si l'entité suit une destination de haute priorité ou non

— `dir` : Entier décrivant la direction auquelle se dirige l'entité

Le reste des attributs correspondent à des informations utiles lors de divers traitements en mémoire lors de l'exécution du programme.

Au niveau des méthodes (concernant le comportement), nous avons :

— `findNextDest` : Permet de déterminer la prochaine destination de l'entité en fonction du contexte, elle sera redéfinit dans les classes filles.

— `moveToward` : Change les attributs de l'entité liés au positionnement (`destination, dest_prioritaire, follow_dest`)

— `update` : Modifie la position actuelle de l'entité en fonction de la destination

— `updateFacingDir` : Modifie la direction en fonction de la destination

— `draw` : Dessine l'entité en fonction de la position actuelle après traitement

— `drawDest` : Dessine un trait entre l'entité et sa prochaine destination.

— `drawVisionRange` : Dessine un cercle autour de l'entité correspondant à son champ de vision.

— `drawInteractRange` : Dessine un cercle autour de l'entité correspondant à son champ d'interraction.

— `initAnim` : Initialise les animations des entités.

La classe **Crewmate** s'agit d'une classe fille de **Amogus**, c'est-à-dire qu'elle hérite des informations et comportements de la classe mère.
Au niveau des attributs, nous avons :

— `DEFAULT_avoid` : Valeur correspondant à la limite de suspicion envers une autre entité auquel on peut réagir ou non en fonction du contexte.

— `CrewmateColor` : Couleur des Crewmate ne possédant pas d'arme.

— `SherifColor` : Couleur des Crewmate possédant une arme.

— `taskDone` : Pointeur de valeurs booléennes décrivant si une tâche a été réalisé par l'entité ou non.

— `occupe` : Entier décrivant si l'entité est occupé à réaliser une tâche ou non (0 et 1).

— `action` : Entier décrivant plusieurs comportements en fonction du contexte :

- ➔ 0 : Neutre, moment de prise de décision.
- ➔ 1 : Se dirige vers la tâche la plus proche.
- ➔ 2 : Se déplace aléatoirement.
- ➔ 3 : Ne bouge pas pendant un certain temps.
- ➔ 4 : Pour une entité non armé, elle s'éloigne d'une autre entité jugée dangereuse.
- ➔ 5 : Pour une entité armé, elle poursuit une autre entité jugée dangereuse.

— `cooldown_pasBouger` : Entier correspondant au nombre d'unité de temps d'attente lorsqu'on se situe à l'action 3.

— `nbTaskCleared` : Nombre de tâche réalisé par l'entité.

— `ind_pp_task` : Indexe de la tâche la plus proche de l'entité.

— `armed` : Booléen décrivant si l'entité est armé ou non.

Au niveau des méthodes, nous avons :

— `updateInfo` : Observe les autres entités autour de lui et met à jour les informations qu'elle dispose.

— `fuirOrChase` : En fonction du contexte et du rôle du Crewmate, on prend la décision de fuir ou de poursuivre une autre entité jugée dangereuse.

— `PlusProcheTask` : Recherche la tâche la plus proche de l'entité

La classe **Imposteur** s'agit également d'une classe fille de **Amogus**, c'est-à-dire qu'elle aussi hérite des informations et comportements de la classe mère.

Au niveau des attributs, nous avons :

— `action` : Entier correspondant à divers comportements en fonction du contexte :

- ➔ 0 : Neutre, prise de décision en cours.
- ➔ 1 : Se dirige vers une tâche aléatoire.
- ➔ 2 : Se déplace aléatoirement.
- ➔ 3 : Ne bouge pas pendant une certaine durée.
- ➔ 4 : Se dirige vers une victime et la tue si elle est à portée.

— `cooldown_pasBouger` : Entier correspondant au nombre d'unité de temps d'attente lorsqu'on se situe à l'action 3.

— `cooldown_kill` : Temps d'attente entre deux meurtres.

— `id_victime` : Indexe de la prochaine victime.

Au niveau des méthodes, nous avons :

— `chase` : Se met à pourchasser une entité en fonction du nombre d'entités autour de la proie

— `kill` : Se met en état de chasse et place sa prochaine destination à la position de sa proie.

5.2.2 Classes Tierces

Les classes tierces représentent des classes utiles dans la mise en forme de la simulation, que ce soit dans l'aspect visuel ou alors dans la mise en place d'une structure de donnée cohérente avec ce que l'on simule.

La classe **Vect** s'agit d'une classe caractérisant la position des entités et des tâches de la simulation. Ces attributs corresponds aux coordonnées de l'élément dans un plan à deux dimensions. La classe permet de réaliser des calculs tels que :

- Sa distance par rapport à un autre vecteur
- Sa longueur
- L'angle par rapport à un autre vecteur (l'axe horizontal par défaut)
- Normaliser
- Observer si un autre vecteur appartient au cercle tracé autour du vecteur en fonction du rayon
- Convertir en degré ou en radiant

La classe **Information** est une classe simple, sans aucunes méthodes, contenant 3 champs :

- `sus` : Flottant estimant le niveau de suspicion d'une entité donné.
- `alive` : Booléen nous donnant l'état de vie d'une entité donné.
- `deathPos` : Position du cadavre (si l'entité est morte).

La classe **Task** est une classe jouant un rôle important dans le comportement des différents agents. Il s'agit d'une entité fixe auxquels les **Crewmate** se doivent atteindre afin de réaliser leurs objectifs. C'est pour cela que cette classe dispose des attributs :

- `position` : Position de la tâche.
- `duree` : Entier correspondant à la durée qu'un **Crewmate** met à réaliser cette tâche.
- `sprite` : Texture de la tâche à afficher.
- `id` : Indexe de la tâche.

La classe **Game** est la classe clé du programme, elle gère le déroulement de la simulation, elle initialise les différents **Amogus** et **Task** en fonction de certains paramètres tels que le nombre de **Crewmate** avec et sans arme, le nombre d' **Imposteur**, le nombre de **Task** global et le nombre de **Task** qu'un **Crewmate** doit réaliser. Les positions des éléments sont aléatoires ainsi que les tâches attribuées aux différents **Crewmate**.
Ensuite, elle s'occupe en permanence d'exécuter les méthodes de déplacement

des **Amogus** et de mettre à jour les informations des **Amogus** ainsi que l'interface graphique à chaque images. C'est aussi cette classe qui permet d'afficher les directions prises par les entités ainsi que les différents cercles afin de mieux observer la simulation.

La classe **Spritesheet** correspond aux informations necessaires lorsqu'on souhaite afficher une image ou alors une animation image par image dans une interface graphique. Voici ce que représente visuellement un spritesheet :

FIGURE 3 – Spritesheet d'un Amogus

La classe dispose des attributs :

— `lstFrame` : Correspond aux différents positions des rectangles des différentes images de l'animation.

— `nbFrameW` : Nombre de colonnes.

— `nbFrameH` : Nombre de lignes.

— `frameSize` : Largeur d'une image.

— `nbFrame` : Nombre d'images.

— `texture` : Texture du spritesheet.

La classe **Anim** est une classe qui se charge de dessiner les différentes images de l'animation en prenant en compte dans le spritesheet où l'animation commence et où elle se termine en fonction de la direction auquel se dirige l'**Amogus**.

5.3 Interprétation des résultats

5.3.1 1 Imposteur vs 1 Shérif

Nous avons fait en sorte que la distance d'interaction du Shérif soit plus importante que celle de l'imposteur, par conséquent le Shérif gagne à chaque fois.

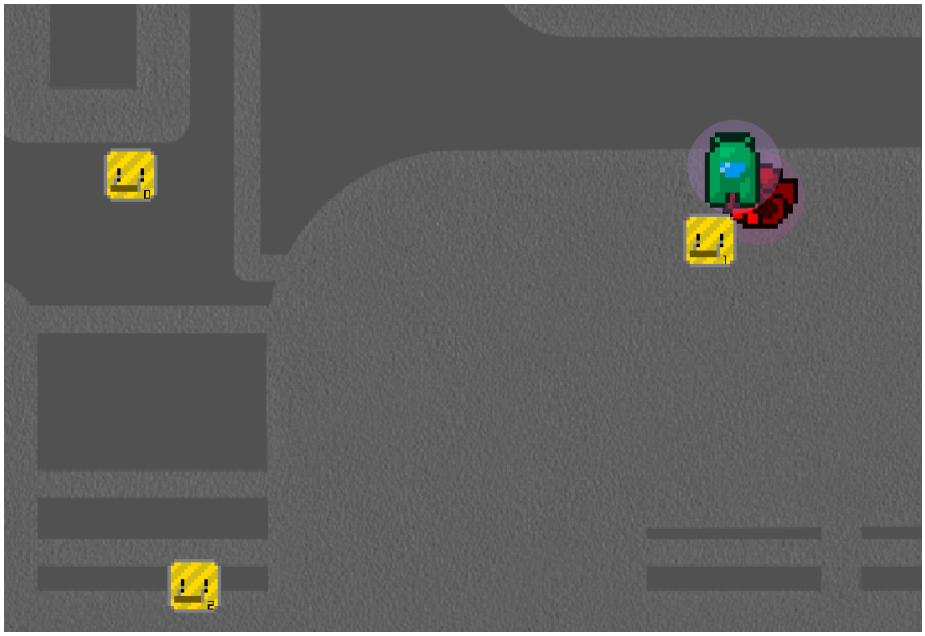


FIGURE 4 – Capture de la situation Imposteur vs Shérif

5.3.2 5 Crewmates vs 5 Imposteurs

Dans ce cas là, la victoire entre les imposteurs et les crewmates est répartis équitablement étant donné qu'il y a une grande part de hasard concernant les kills.

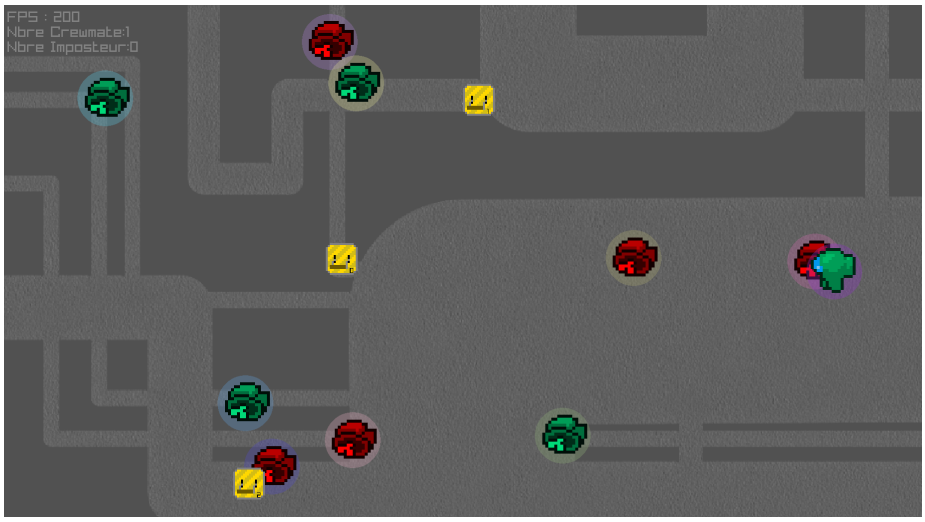


FIGURE 5 – Capture de la situation 5 Crewmates vs 5 Imposteurs

5.3.3 9 Crewmates vs 1 Imposteurs

Dans ces conditions, après de nombreux tests, il s'avère que les imposteurs gagnent la partie plus souvent. On suppose que cela doit être dû au fait qu'un grand nombre de crewmates sont assassinés pendant leurs tâches.

FIGURE 6 – Enter Caption

5.4 Note

Il est possible d'afficher les directions des Amongus en tapant ctrl+d au clavier.

6 Conclusion

En conclusion, ce projet nous a permis de développer nos compétences que ce soit l'apprentissage du c++, du concept de programmation orientée objet, l'utilisation de nouveaux outils tels que gdb, valgrind et de l'implémentation d'une interface graphique.
Nous sommes parvenus à réaliser une simulation multi-agents avec des agents pouvant intéragir entre eux ou avec l'environnement avec :

- Un **Crewmate** se déplaçant vers des **Tasks** pour accomplir sa mission tout en se méfiant des autres personnages.
- Un **Shérif** analysant les différents personnages afin d'éliminer les **Imposteurs** tout en évitant de s'en prendre à un innocent.
- Un **Imposteur** cherchant à éliminer les **Crewmates** en cherchant des cibles isolées.

Tout cela en jouant avec les probabilités afin d'éviter des comportements trop linéaires.