Gurnaud Jade
Turan Baturay
Kruzic Charlotte

The Legend Of Chris Minnahl

Jeu d'infiltration stratégique

Encadré par Julien Bernard, maître de conférences

Licence 3 Informatique
Université de Franche-Comté
2022-2023

Remerciements

Nous aimerions exprimer ensemble notre sincère gratitude envers notre tuteur, Julien Bernard. Son soutien constant, ses précieux conseils et son expertise ont grandement contribué à la réussite de notre projet tutoré.

Table des matières

Liste des figures

I. Introduction

L'objectif de ce projet était de réaliser un jeu vidéo de type infiltration stratégique, en C++, à l'aide de la bibliothèque Gamedev Framework[1] (gf) : une bibliothèque offrant un cadre de travail pour les développeurs de jeux 2D en C++. Le projet était encadré par Julien Bernard, maître de conférences à l'Université de Franche-Comté.

Un jeu d'infiltration est un jeu vidéo dans lequel le joueur doit tenter de s'infiltrer dans un lieu, en toute discrétion, en évitant d'être repéré par des ennemis, pour réaliser une mission. Pour cela, il dispose généralement de gadgets permettant de l'aider dans sa tâche. En général, cela implique une progression lente et réfléchie.

Nous avons créé *The Legend Of Chris Minnahl*, un jeu d'infiltration se déroulant dans un musée, dans lequel le joueur incarne un voleur, ayant comme but de dérober un ou plusieurs diamants, sans se faire repérer par les gardes du musée.

[1] https://gamedevframework.github.io/v0.22.0/
https://gamedevframework.github.io/

II. Modélisation

1. Les règles du jeu

The Legend Of Chris Minnahl est un jeu se jouant en solitaire.

Au début de la partie, le voleur est positionné sur une case de départ définie. Grâce aux touches directionnelles, l'utilisateur peut se déplacer dans le musée. Il doit d'abord récupérer le ou les diamants, puis rejoindre le point d'arrivée, sans se faire repérer par les différents gardes du musée, qui réalisent des rondes. Les gardes possèdent un champ de vision, si le joueur entre dans cette zone, il est repéré et perd la partie.

Pour aider le joueur, il y a des socles disposés dans le musée, qui lui permettent de se transformer en statue, et donc ne pas être repéré par les gardes. Pour se mettre en statue, il faut appuyer sur la touche espace et être sur un socle.

Le joueur dispose aussi d'une mini-carte, qu'il peut voir en appuyant sur la touche M. Sur celle-ci, on peut voir l'ensemble du musée, à l'exception des gardes.

Si le joueur se fait attraper par un des gardes, il perd. Si le joueur réussit à rejoindre la case d'arrivée, en ayant récupéré tous les diamants du niveau, il gagne. Le joueur ne peut pas sortir du musée sans avoir récupéré tous les diamants.

Figure 1 : Écran de jeu

2. Le choix de la structure

Pour structurer notre code, nous avons organisé les pages de notre jeu de façon indépendante.

Nous avons utilisé le gestionnaire de jeu proposé par la bibliothèque gf : la classe GameManager. Cette classe hérite du SceneManager de gf, le gestionnaire de scènes. Le gestionnaire de jeu possède aussi un gestionnaire de ressource, un objet de la classe ResourceManager de gf (voir Figure 2).

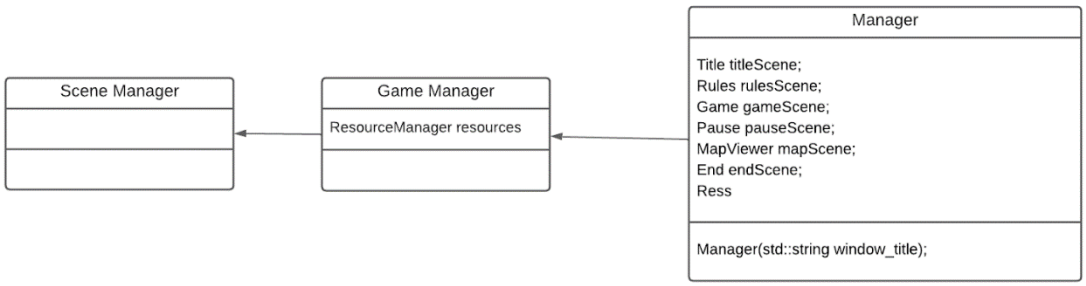


Figure 2 : Diagramme d'héritage de la classe Manager

a. GameManager

La classe SceneManager de gf possède un ensemble de fonctions permettant de gérer les scènes. Il est possible d'en ajouter, d'en enlever, ou d'en remplacer une par une autre. Les scènes sont organisées en pile. La classe GameManager, héritant de cette classe, possède les mêmes fonctionnalités.

C'est donc dans un GameManager, que nous avons instancié et stocké l'ensemble de nos scènes de jeu, à savoir, la page d'accueil, les règles du jeu, l'écran de jeu, celle de la carte réduite, le menu pause et la page de fin.

De plus, nous avons utilisé la fonctionnalité de remplacement des scènes pour passer d'une page de notre jeu à une autre, mais aussi le système de pile pour positionner une scène sur une autre, pour notre menu pause par exemple. C'est aussi cette classe qui nous a permis d'utiliser une boucle de jeu général qui lance, lorsqu'une scène est active, la boucle de jeu de la scène en cours.

b. Scene

La classe Scene de gf possède sa propre boucle de jeu avec des fonctions déjà définies. Une fonction de traitement des événements, du clavier ou de la souris. Une fonction de gestion des actions du joueur, pour les déplacements. Une fonction de mise à jour, une pour l'affichage de la scène, et une pour mettre à jour le tampon d'image en fonction de la taille de la fenêtre de jeu.

Les scènes possèdent aussi des propriétés propres qui permettent de savoir si elles sont actives, si elles sont en pause, si elles sont cachées, ou de les définir ainsi. Chaque scène peut contenir des vues, des actions et des entités. Ces éléments étant automatiquement mis à jour par la boucle de jeu intégrée de la scène.

Nous avons alors pu créer différentes scènes, une pour chaque page/interface de notre jeu. Pour chaque scène, nous avons défini des fonctionnalités, des attributs, des méthodes, mais aussi des vues spécifiques. Nous avons aussi ajouté des actions et des entités propres à chaque scène. Ainsi, elles sont stockées dans la scène correspondante et gérées dans celle-ci. Cela nous a aussi permis de gérer les éléments visuels, d'avoir une meilleure organisation de notre code et donc de mieux nous repérer et de pouvoir y apporter des modifications.

c. ResourceManager

La classe ResourceManager de gf permet, comme son nom l'indique, de gérer les ressources. Elle possède des fonctions qui permettent de récupérer des images, des textures, mais aussi des polices de caractères, stockées à un chemin donné.

Nous avons aussi utilisé ce gestionnaire de ressources intégré pour gérer, de façon optimale, l'ensemble des ressources de notre jeu : les polices de caractères, les fichiers textes et JSON et les images. Cela nous a permis de charger en mémoire l'ensemble des ressources, au lancement du jeu.

3. Les différentes scènes

Les scènes sont gérées grâce à un gestionnaire de scènes, notre classe Manager, qui permet d'effectuer les différentes transitions entre les différentes scènes du jeu (voir Figure 3). C'est lui qui est responsable de l'activation et de la désactivation des scènes.

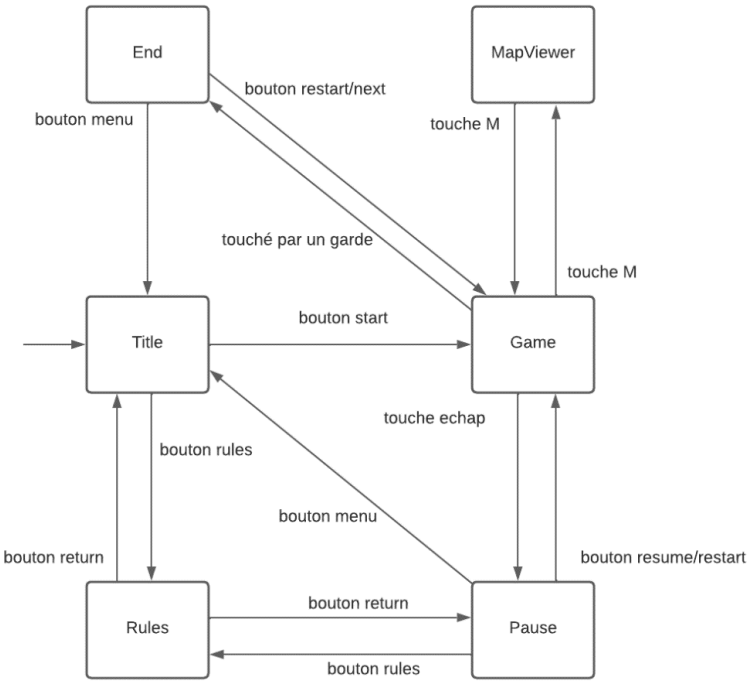


Figure 3 : Organisation des scènes

a. Scene Title

La page d'accueil contient le titre du jeu et trois boutons (voir Annexe 1). Le premier qui permet de démarrer le jeu, le second qui nous dirige vers la page de règles et le dernier qui permet de quitter le jeu.

b. Scene Rules

En ce qui concerne la page des règles du jeu, elle contient un bouton pour retourner à la page précédente et les règles du jeu (voir Annexe 2).

Les règles sont sous la forme d'images accompagnées de mots clés, pour permettre une meilleure lisibilité et une meilleure compréhension par le joueur.

c. Scene Game

La page de jeu affiche le niveau en cours (voir Annexe 3). La vue est centrée sur le joueur et permet de voir qu'une partie de la carte de jeu. Il est possible de visualiser la carte entière, sans les gardes, en cliquant sur la touche M. Cela affiche la scène MapViewer.

Il est aussi possible de mettre le jeu en pause en cliquant sur la touche Echap. Cela affiche la scène Pause.

d. Scene Pause

La page pause dispose de différents boutons (voir Annexe 4). Un pour reprendre la partie en cours, un pour recommencer la partie, un pour visionner les règles du jeu, et enfin un pour revenir au menu.

Cette scène est accessible depuis la page de jeu avec la touche Echap, elle s'affiche par-dessus. La page de jeu est mise en pause et visible en arrière-plan.

e. Scene MapViewer

Sur la page MapViewer est affiché l'ensemble du niveau en cours, sans les gardes (voir Annexe 5). Elle permet au joueur de se repérer dans le niveau.

Depuis cette scène, on peut retourner à la scène de jeu en cliquant sur la touche M.

f. Scene End

La page de fin apparaît à chaque fin de niveau, et affiche si le niveau a été gagné ou perdu (voir Annexe 6). Lors d'une défaite, il y a deux boutons disponibles : un permettant de recommencer le niveau et un autre permettant de retourner au menu. Lors d'une victoire, ces deux boutons sont disponibles, mais il y en a un en plus, permettant de passer au niveau suivant, lorsqu'il en reste.

4. La représentation visuelle

Nous avons choisi d'utiliser une vue top down, pour représenter les niveaux de notre jeu, et ainsi permettre de donner de la perspective au musée.

L'affichage des éléments du niveau, dans la classe Game, se fait en deux fois et est effectué de gauche à droite et de haut en bas. Cela nous permet d'afficher par couches et dans le bon ordre les différents éléments. Nous commençons par l'affichage des éléments au sol : le sol, la case de départ, la case de fin et le socle de la statue. Puis dans une nouvelle boucle nous affichons le joueur ainsi que les gardes et pour terminer les diamants, les vitrines et les murs.

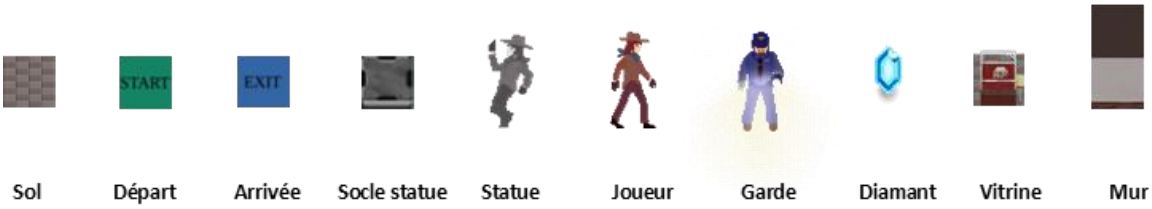


Figure 4 : Sprite du jeu

Pour représenter graphiquement nos différents éléments, nous avons utilisé nos propres sprites dessinés par Baturay, qui sont des éléments graphiques qui peuvent se déplacer sur l'écran (voir Figure 4).

Parmi les différents sprites que nous utilisons, certains changent au cours du jeu. Pour les gardes, nous en avons quatre, un pour chaque direction. Le joueur aussi possède un sprite par direction, mais il en a un en plus, permettant de le représenter lorsqu'il est en statue. Et enfin, il y a le diamant qui ne possède qu'un seul sprite, mais qui lorsqu'il est volé par le joueur disparaît.

5. La physique du jeu

a. Mouvements du joueur et des gardes

Le niveau est représenté sous forme de grille, cependant les mouvements des éléments dynamiques du jeu, le joueur et les gardes, ne sont pas alignés selon cette grille. Cela permet une plus grande liberté de mouvement et la possibilité de se déplacer en diagonale.

b. Gestion des collisions du joueur

L'ensemble des collisions du jeu étant des collisions rectangles-rectangles, nous avons choisi de ne pas utiliser de moteurs physiques et de gérer l'ensemble des collisions nous-mêmes, à l'aide bibliothèque gf, plus particulièrement grâce à la fonction intersects.

A chaque tour de la boucle de jeu, au moment de la mise à jour des éléments de la classe Level, l'ensemble des collisions sont vérifiées. Lorsqu'une collision est trouvée, la zone de collision de l'élément qui est entré en contact avec le joueur est récupérée, cela permet de gérer ce qu'il faut faire avec celle-ci.

Avec les gardes

Les gardes possèdent deux zones de collision, celle du garde lui-même et celle de son faisceau lumineux, avec lesquelles le joueur peut entrer en collision. Lorsque le joueur entre en collision avec une de ces zones, il perd immédiatement.

Avec les murs

Lors d'une collision entre le joueur et un mur, la position du joueur doit être recalculée afin de ne pas traverser le mur. Pour cela, il faut savoir dans quelle direction est la collision : l'axe des abscisses, celui des ordonnés ou les deux. Il faut aussi savoir son sens. Pour cela, nous récupérons la vélocité du joueur. Puis grâce à ces informations, nous pouvons positionner le joueur au niveau du mur et le décaler, de sa taille, dans le bon sens et la bonne direction.

Avec les vitrines

Les collisions entre le joueur et les vitrines fonctionnent sur le même principe que celles avec les murs. Les vitrines et les murs sont les seuls éléments à ne pas pouvoir être traversés par le joueur, cela est dû au fait qu'ils sont définis comme étant solides.

Avec les diamants

Lorsqu'une collision entre le joueur et un diamant est détectée, le diamant change de catégorie, il devient un "objet trouvé", cela engendre plusieurs changements. Tout d'abord, un changement visuel, l'objet disparaît. Il y a aussi l'augmentation du compteur des objets trouvés. Et enfin, si c'était le dernier diamant, le joueur peut maintenant terminer le jeu en se rendant sur la case de fin.

Avec les socles de statue

Les collisions entre le joueur et un socle de statue sont un peu différentes. En effet, lors d'une collision de ce type, rien ne se passe, sauf si l'utilisateur appuie sur la touche espace. Lorsque la touche espace est enfoncée et que le joueur est en collision avec le socle, alors il se transforme en statue.

c. Physique du joueur

Déplacements

Lors de l'appui sur une seule des touches directionnelles, déplacement horizontal ou vertical, le joueur se déplace de 1. Pour éviter que le joueur se déplace de sqrt(2), lorsque l'on appuie sur deux touches directionnelles, déplacement en diagonale, nous avons recalculé sa vitesse déplacement. Pour cela, nous avons utilisé la formule : 1/sqrt(2)

Propriétés de la statue

Lorsque le joueur est en statue, plusieurs de ces propriétés physiques changent. Tout d'abord, il n'entre plus en collisions avec les gardes, cela signifie que s'il passe dans le champ de vision d'un garde, il ne perd pas. De plus, il n'a plus la possibilité de se déplacer, il est figé.

III. Implémentation

1. Initialisation

a. Chargement des niveaux

Lors du lancement du jeu, le premier niveau se charge immédiatement. Cela se fait via un fichier texte externe contenant les données du niveau, ainsi qu'un fichier JSON listant les données des gardes du niveau, ainsi que leurs routines.

Lorsque le joueur atteint la fin du niveau, le niveau suivant est chargé en parsant le fichier texte.

Les niveaux n'ont pas de taille fixe, la largeur et hauteur sont donc calculées en fonction du nombre de lignes dans le fichier texte, et du nombre de symboles à chaque ligne (supposés constants entre chaque ligne). Par la suite, chaque caractère du fichier est lu et la case correspondante est ajoutée dans le niveau.

b. Chargement des classes et entités

Après le chargement du niveau, le joueur est déplacé à la position de départ.

Les gardes sont créées à partir du fichier JSON, assignées à leurs positions de départ et leurs routines leur sont attribuées.

2. Le stockage des données

Pour stocker l'ensemble des données de notre jeu, nous avons utilisé un gestionnaire de ressources, directement intégré à notre gestionnaire de jeu.

Nous avons stocké et organisé l'ensemble de nos données dans le dossier data/TheLegendOfChrisMinnahl. Pour qu'il soit accessible, par le gestionnaire de ressources et pour tous les utilisateurs, nous avons géré la procédure d'installation.

Nous avons différents types de données dans ce jeu : des polices de caractères sous format TrueType font (.ttf), des niveaux sous format texte, des routines de gardes sous format JSON et enfin des images sous format Portable Network Graphics (.png) pour nos sprites.

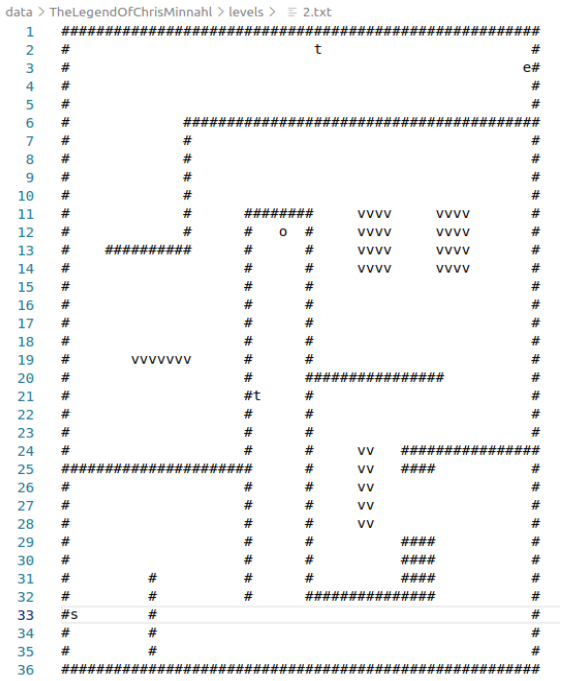
Le gestionnaire de jeu n'est accessible, par référence, que par certaines classes de notre jeu, celles qui ont besoin d'une ou plusieurs ressources.

a. Format des données externes

Niveaux

Les différents niveaux sont stockés dans des fichiers textes en dehors du code source (voir Figure 4). Ils sont nommés en fonction de leur ordre dans le jeu : '1.txt' pour le premier niveau, '2.txt' pour le 2ème, …etc. Le contenu du fichier texte est une représentation texte du niveau, qui sera parsée par le jeu dans la classe Map.

data > TheLegendOfChrisMinnahl > levels > 2.txt



```
1  ##################################################
2  #                         t                      #
3  #                                               e#
4  #                                                #
5  #                                                #
6  #           ########################################
7  #           #                                    #
8  #           #                                    #
9  #           #                                    #
10 #           #                                    #
11 #           #      ########     vvvv     vvvv     #
12 #           #      #   o  #     vvvv     vvvv     #
13 #    ##########    #      #     vvvv     vvvv     #
14 #                  #      #     vvvv     vvvv     #
15 #                  #      #                       #
16 #                  #      #                       #
17 #                  #      #                       #
18 #                  #      #                       #
19 #       vvvvvvv    #      #                       #
20 #                  #      ################        #
21 #                  #t     #                       #
22 #                  #      #                       #
23 #                  #      #                       #
24 #                  #      #     vv   ##############
25 ####################      #     vv   ####        #
26 #                  #      #     vv               #
27 #                  #      #     vv               #
28 #                  #      #     vv               #
29 #                  #      #          ####        #
30 #                  #      #          ####        #
31 #        #         #      #          ####        #
32 #        #         #      ###############        #
33 #s       #                                       #
34 #        #                                       #
35 #        #                                       #
36 ##################################################
```

Figure 5 : Fichier texte de stockage de niveaux

Le parsage se fait par une simple boucle itérant chaque caractère de chaque ligne du tableau. En fonction du caractère lu, la case correspondante est créée.

Dans ce jeu, chaque symbole a une signification spécifique. Le symbole '#' correspond à un mur, le symbole ‘o’ désigne un objet, le symbole ‘t’ symbolise une statue, le symbole ‘s’ indique l'entrée et ‘e’ la sortie et le symbole ‘v’ décrit une vitrine (voir Figure 5).

Gardes

Les données relatives aux gardes sont stockées dans des fichiers JSON, plutôt qu’en format texte, en raison de la complexité des données. Un garde est défini par sa position initiale, et sa liste d’actions, les routines, définies ci-dessous.

Les actions que prennent les gardes sont des dictionnaires, qui sont organisés en listes d’actions pour former la route de chaque garde (voir figure 6). Pour chaque niveau, il y a un fichier JSON équivalent qui contient une liste de gardes sous le format décrit précédemment.

```
{
    "position": {"x": 45, "y": 26},
    "route": [
        {"type":"GO","time":1,"position": {"x": 51, "y": 26}},
        {"type":"WAIT","time":1,"position": {"x": 51, "y": 26}},
        {"type":"GO","time":1,"position": {"x": 51, "y": 31}},
        {"type":"WAIT","time":1,"position": {"x": 51, "y": 31}},
        {"type":"GO","time":1,"position": {"x": 45, "y": 31}},
        {"type":"WAIT","time":1,"position": {"x": 45, "y": 31}},
        {"type":"GO","time":1,"position": {"x": 45, "y": 26}},
        {"type":"WAIT","time":1,"position": {"x": 45, "y": 26}}
    ]
},
```

Figure 6 : Routines des gardes au format JSON

Stockage des routines

Les gardes ont chacun une routine ainsi qu'une position de départ qui leur sont associées. La routine prend la forme d'une liste d'actions à effectuer. Durant le jeu, le garde est positionné à la position de départ et effectue ses actions. Lorsqu'il arrive à la fin de sa liste, il retourne au début.

Les deux actions possibles sont GO et WAIT, qui permettent respectivement de déplacer le garde à une position donnée, ou de le faire attendre sur place.

Chaque action de la routine possède un attribut qui détermine sa durée. Ainsi il est possible d'enchaîner les actions avec des durées différentes.

Cette structure permet de stocker, de manière assez aisée, ces informations dans un fichier au format JSON, d'où l'usage de ce format.

Patrouille

À chaque boucle de jeu, l'état du garde est mis à jour, afin que son action courante se poursuive ou, si la durée de l'action est achevée, de passer à l'action suivante.

Les déplacements sont interpolés en fonction de la position courante, et la position à atteindre en fonction du temps restant.

3. Les structures et les classes

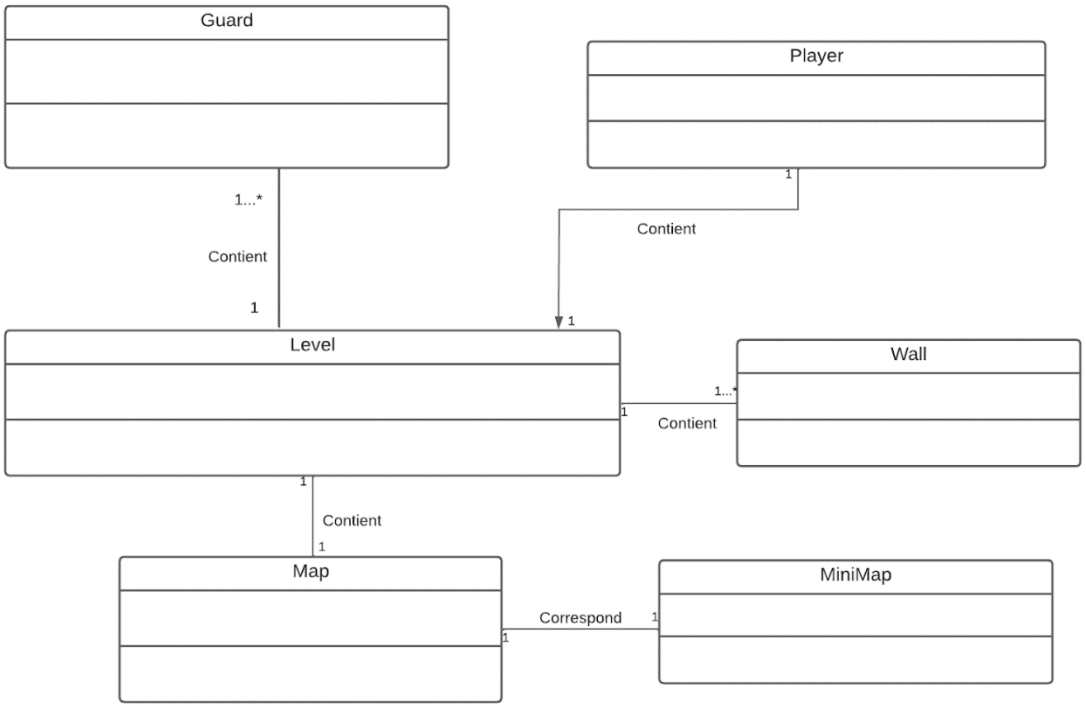


Figure 7 : Organisation des classes

a. Manager

La classe Manager hérite de la classe GameManager de gf (voir Figure 2). Elle gère les ressources, mais aussi les scènes du jeu qu'elle instancie et stocke en attribut publique.

Elle permet aussi d'initialiser la fenêtre du jeu. Cette fenêtre peut être fermée en cliquant sur le bouton avec la croix en haut à droite. Lorsque celle-ci est fermée, le jeu est supprimé. Lors de l'instanciation d'un objet de la classe manager, la fenêtre du jeu est initialisée et la scène de titre est affichée. Le manager est instancié dans la fonction principale.

Principe de la boucle de jeu

Dans notre jeu, la boucle de jeu général est gérée par notre classe Manager. La classe Manager, possédant l'ensemble des scènes de notre jeu, permet l'appel des boucles de jeu des scènes actives. Une boucle de jeu est divisée en 3 temps, que nous allons détailler ci-dessous.

Input

La première étape est la récupération des entrées du joueur. Ces entrées peuvent provenir du clavier ou de la souris, dans notre cas. Une fois lues, ces données pourront être utilisées dans les prochaines étapes pour mettre à jour l'état du jeu.

Update

La deuxième étape consiste à mettre à jour l'état interne des différents éléments du jeu en modifiant les variables qui les représentent, en fonction des entrées récupérées à l'étape précédente.

Draw

La troisième étape consiste à afficher les éléments graphiques du jeu à l'écran.

b. Map

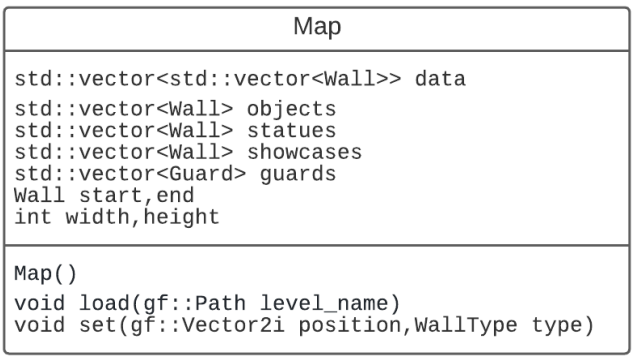


Figure 8 : Classe Map

La classe Map permet de charger le niveau en cours. La carte possède des gardes et l'ensemble des murs : les vitrines, les diamants, les socles de statue, le début et la fin. Elle a une hauteur et une largeur (voir Figure 8).

C’est dans cette classe que les niveaux, définis dans un fichier texte, et les gardes, définis dans un fichier JSON, sont chargés. Elle ne possède pas de fonctions permettant de la mettre à jour, ni de réaliser un affichage.

Nous avons choisi de séparer les données statiques des données dynamiques du jeu, ce qui a amené à la création de cette classe. Les classes Level et Minimap accèdent ainsi aux données de la carte en lecture uniquement.

c. MiniMap

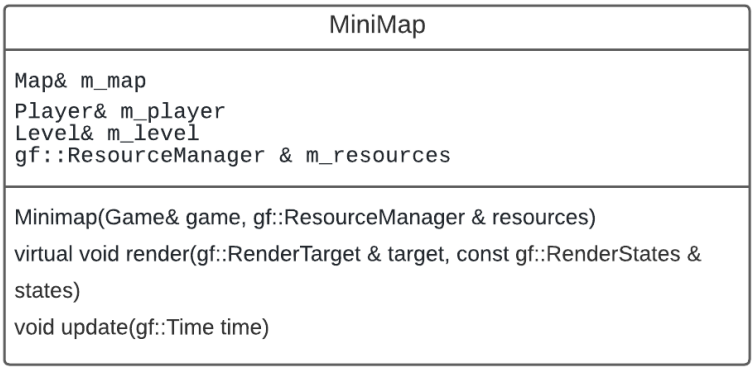


Figure 9 : Classe MiniMap

La classe MiniMap permet de représenter une petite carte entière du niveau en cours.

La mini-carte possède une carte, un joueur, un niveau et l’ensemble des murs. Elle a aussi un gestionnaire de ressources qui donne accès à l’ensemble des textures et sprites (voir Figure 9).

Elle est représentée de la même manière que la classe Level, mais ne permet pas d’effectuer des modifications sur la Map, les gardes ne sont pas non plus stockés. De plus, elle est mise à jour grâce à son élément Level, qui lui permet de récupérer l’ensemble des informations dont elle a besoin pour s’afficher correctement.

d. Level

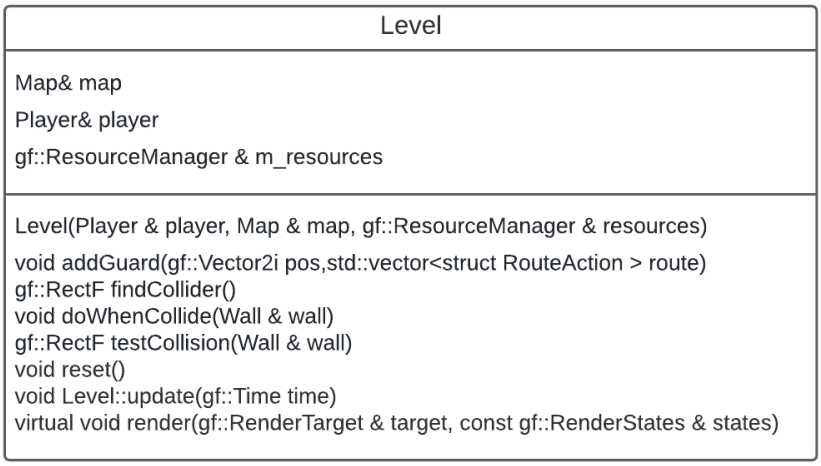


Figure 10 : Classe Level

La classe Level permet de représenter le niveau du jeu accessible depuis la classe Game.

Elle contient une carte, un joueur, un gestionnaire de ressources qui donne accès à l'ensemble des textures et des sprites, des gardes et des diamants. C'est dans cette classe que sont gérées l'ensemble des collisions (voir Figure 10).

Chaque niveau est réinitialisé juste avant de passer au niveau suivant. Le joueur retourne au départ et retrouve une vitesse nulle, les gardes se positionnent et les objets sont remis à leur place.

A chaque tour de la boucle de jeu, le niveau est mis à jour et affiché depuis la scène Game.

Le diamant

Chaque niveau contient au minimum un diamant. Un compteur, en haut à gauche de la fenêtre, permet de savoir le nombre de diamants que le joueur a déjà récupérés par rapport au nombre total présent dans le niveau.

Les diamants sont des objets de la classe Wall et sont stockés dans le niveau en fonction de s'ils ont déjà été récupérés ou non.

e. Wall

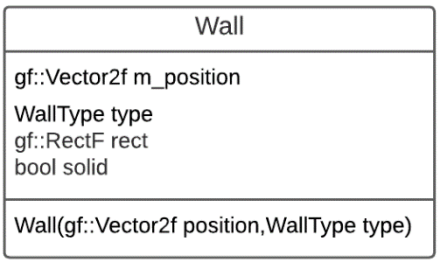


Figure 11 : Classe Wall

La classe Wall permet de représenter les murs. Chaque mur possède une position, une zone de collision, et un type (voir Figure 18). Ils ne changent pas au cours d'un niveau, ils restent fixes.

Il existe différents types de murs, le type :

- solide qui permet de représenter les murs
- vide qui représente les sols,
- début pour le début du niveau,
- fin qui représente la fin du niveau,
- objet qui représente l'objet que le joueur doit voler,
- statue qui représente les socles sur lesquels les joueurs peuvent se transformer en statue,
- vitrine qui représente les vitrines du musée.

En plus de ses différents types, les Wall peuvent être solides ou non. Les vitrines et les murs sont les seuls à être solide et c'est grâce à cette distinction, qu'ils ne sont pas traversables par le joueur.

f. Player

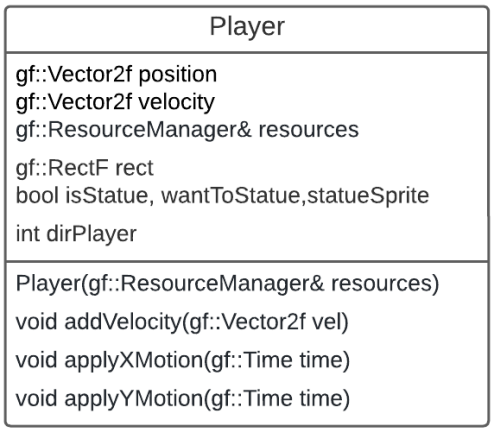


Figure 12 : Classe Player

La classe Player permet de représenter un joueur du jeu.

Un joueur possède une position, une vitesse, une zone de collision et un manager ressource permettant de récupérer les textures pour le représenter (voir Figure 12).

La position du joueur, sa vitesse et sa représentation changent au cours du jeu. Pour le déplacer, l'utilisateur doit utiliser les touches directionnelles. Il peut aussi transformer le joueur en statue s'il est sur un socle et qu'il clique sur la touche espace. Le joueur est instancié dans la scène Game et reste le même tout au long du jeu.

g. Guard

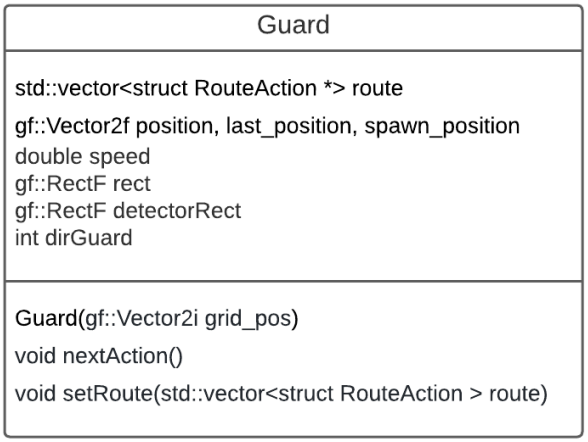


Figure 13 : Classe Guard

La classe Guard permet de représenter un garde d'un niveau.

Un garde possède une position, une vitesse, une direction et une zone de collision (voir Figure 13). Il possède aussi des rectangles de vision que le joueur devra esquiver. Les gardes font partie du niveau courant, puis sont enlevés et remplacés à chaque changement de ce dernier.

Une routine prédéfinie est associée à chaque garde, leur permettant de surveiller le musée avec une trajectoire prédéfinie. Cette routine, stockée en JSON, est extraite puis affectée au garde sous une forme d'une liste de RouteAction (voir Figure 14), une structure créée pour faciliter la gestion des patrouilles.

```
struct RouteAction {
  actionType type;
  float time;
  float cumulated_time;
  gf::Vector2i grid_position;
};
```

Figure 14 : Structure de gestion des patrouilles

La structure possède les informations nécessaires pour les actions. Elle contient le type qui peut être GO ou WAIT et la durée de l'action (en seconde). Elle contient aussi la durée cumulée de l'action, qui permet de savoir si cette dernière est arrivée à terme. Cette valeur est mise à 0 au début de l'action, puis incrémentée avec le temps écoulé (dt) depuis la dernière frame, à chaque tour de boucle. A partir de la durée cumulée et de la durée totale, un pourcentage de progression de l'action courante peut être calculé comme suit :

pourcentage = (durée_totale / durée_cumulée)

Ce pourcentage sert par la suite à calculer la position du garde lorsqu'il se déplace. Enfin, RouteAction enregistre la position où doit se déplacer le garde dans la grille de jeu, au moment de l'action GO.

IV. Conclusion

1. Gestion de projet

Nous avons utilisé différents outils pour nous organiser lors de ce projet.

Pour la partie communication, nous avons créé deux groupes privés sur Discord[2], une plateforme de communication en ligne. Elle fonctionne sur le principe de serveurs, mais il est aussi possible de créer des groupes de chat privé, comme nous l'avons fait. Sur cette plateforme, il est possible de faire des appels vocaux, des appels vidéo, de partager un écran ou encore de parler en chat textuel.

Le premier groupe, où nous étions que les trois, nous permettait d'échanger nos idées, mais aussi de faire des appels, afin de suivre les avancées de chacun. Le second, avec Julien Bernard, nous a permis de poser des questions et de nous aider lors des blocages.

Avant de débuter le projet en C++, nous avons programmé un prototype minimal en python (avec la librairie pygame) pour tester la logique du jeu et l'implémentation des gardes. Une fois que ce premier prototype fut satisfaisant, nous sommes passés à l'implémentation en C++, en prenant le prototype comme modèle.

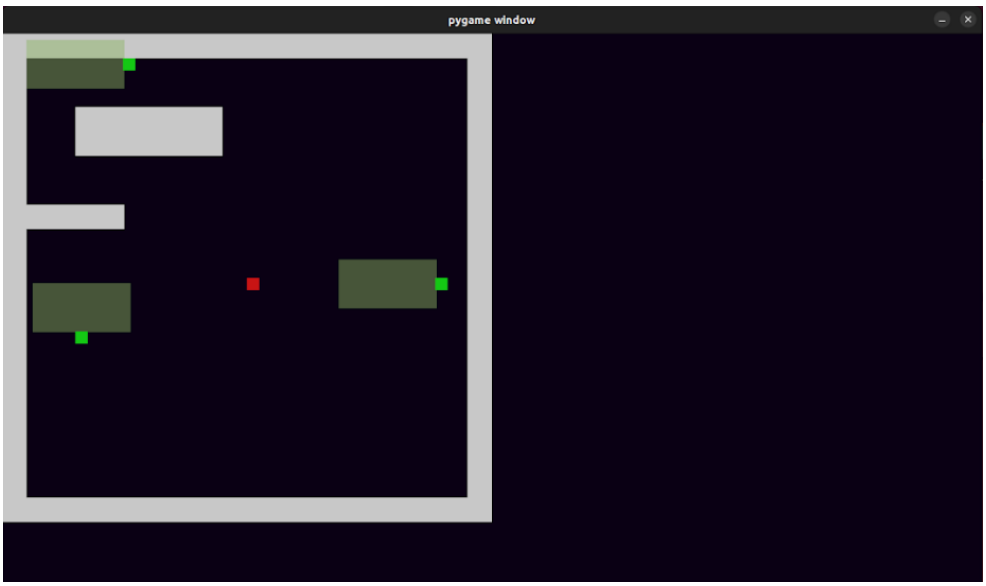


Figure 15 : Prototype Python

Pour la partie programmation, nous avons créé un dépôt sur GitHub[3], un service web d'hébergement permettant de collaborer sur des projets et de gérer les versions. Celui-ci nous a permis de sauvegarder notre projet et de pouvoir nous partager le code facilement. Nous avons créé différentes branches afin de pouvoir fusionner et gérer les conflits plus facilement.

---

[2] https://discord.com/

[3] https://github.com/TuranBaturay/ProjetL3

Nous avons créé différentes branches afin d'organiser les différentes versions du code. La branche "master" permet de sauvegarder les versions jouables de notre projet. Nous avons développé l'ensemble de notre jeu dans la branche "development". Au départ, nous avions seulement cette branche, cela nous a permis de créer la base de notre code. Nous pouvions ajouter des fonctionnalités et les tester, tout en gardant une version sûre et stable sauvegardée sur la branche "master".

A partir de "development", nous avons créé plusieurs branches pour nous permettre de répartir le travail et de pouvoir travailler en même temps. L'organisation en scènes permise par gf, nous a permis de faciliter le partage des tâches sur les différentes pages du jeu.

2. Bilan personnel

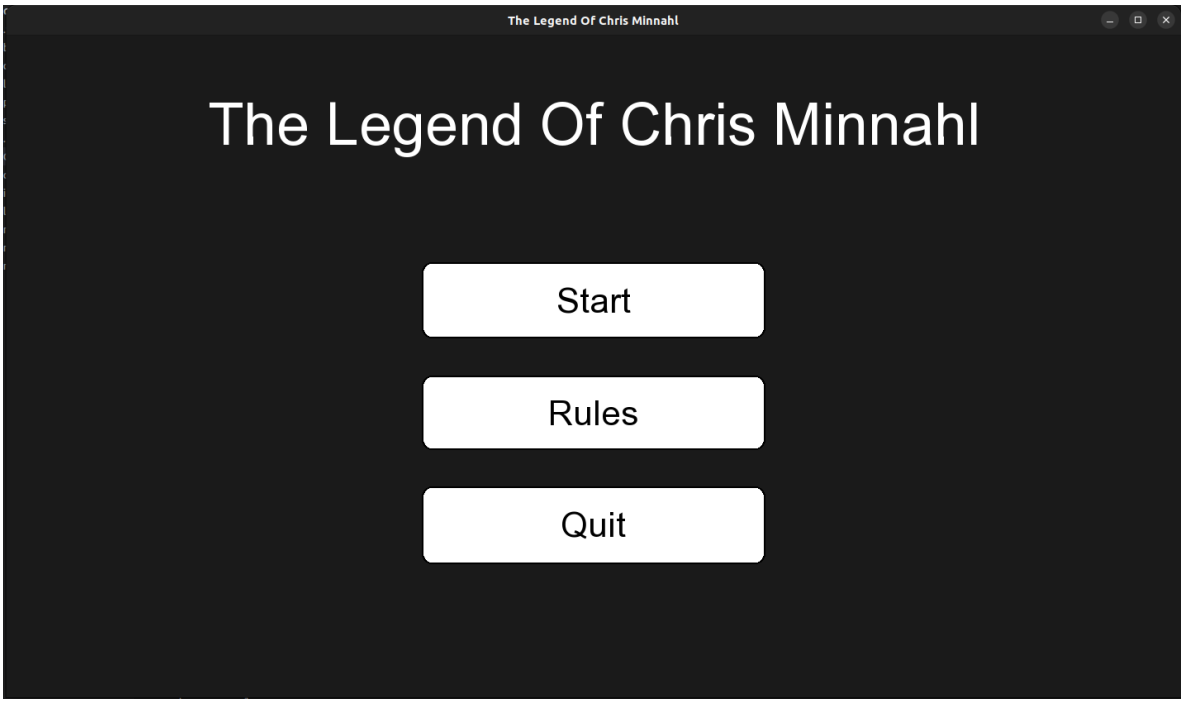
La réalisation de ce projet nous a permis d'acquérir de nouvelles compétences et connaissances. Nous avons ainsi pu comprendre le fonctionnement d'un jeu vidéo, notamment avec sa boucle de jeu. En travaillant en équipe, nous avons appris à collaborer efficacement sur un projet en organisant et en planifiant les différentes tâches. Nous avons également perfectionné notre utilisation du gestionnaire de version Github.

De plus, ce projet nous a permis de développer notre créativité et notre sens de l'innovation en créant notre propre jeu. Nous avons également appris à ajouter des éléments graphiques à notre programme C++, grâce à la bibliothèque Gamedev Framework, que nous avons appris à utiliser lors de ce projet. Nous avons aussi appris à coder en C++.

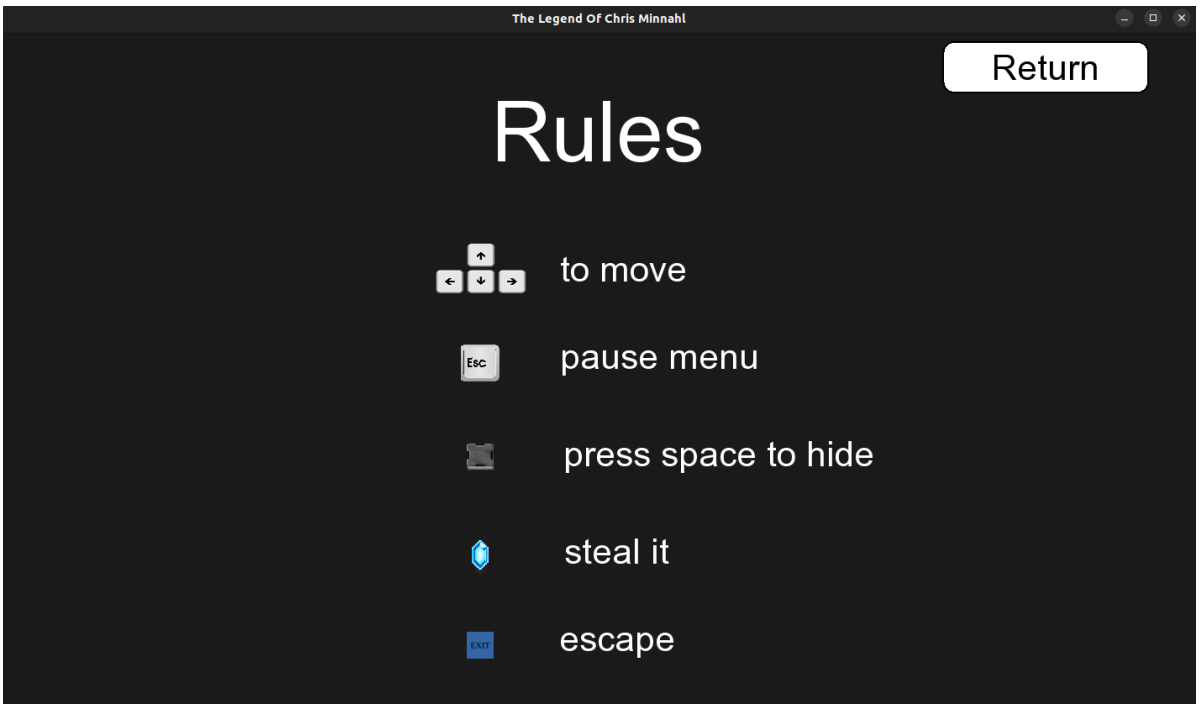
3. Validation du cahier des charges

The Legend Of Chris Minnahl est un jeu d'infiltration stratégique. Le joueur doit voler un trésor et sortir du musée, pour l'aider dans sa quête, il est capable de se transformer en statue, et il a accès à une petite carte. Nous l'avons implémenté en utilisant le langage C++, et la bibliothèque Gamedev Framework. Il répond au cahier des charges.

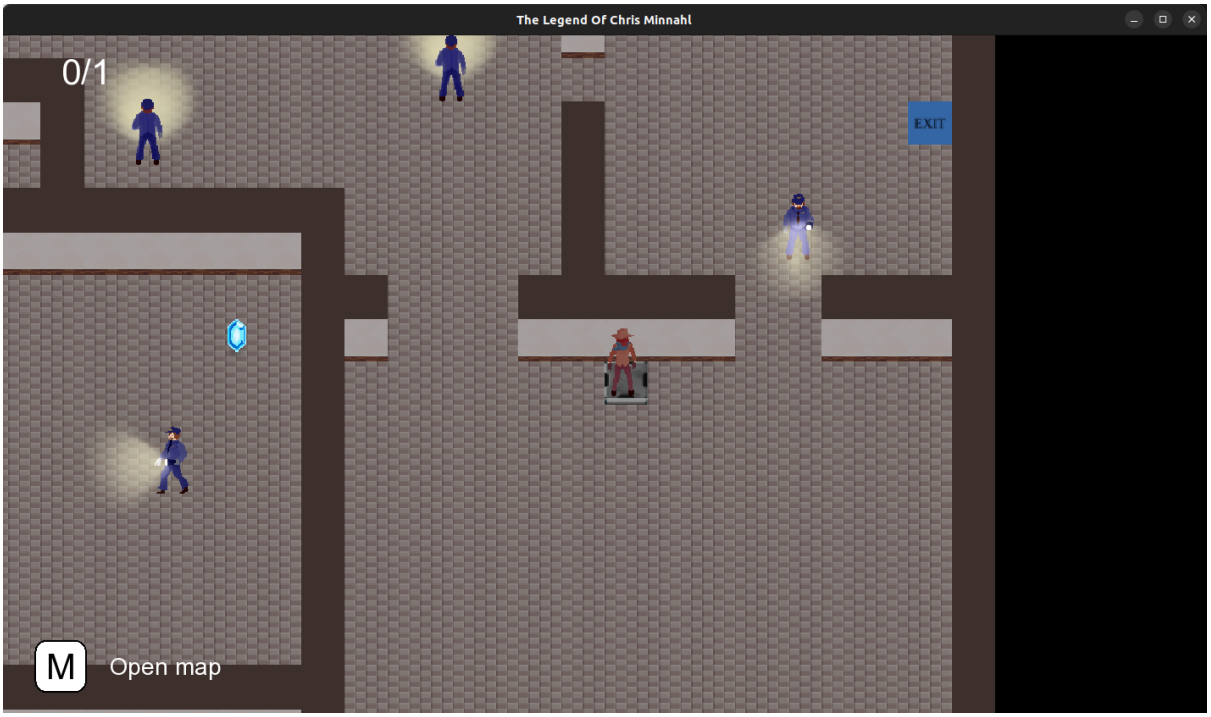
V. Annexes



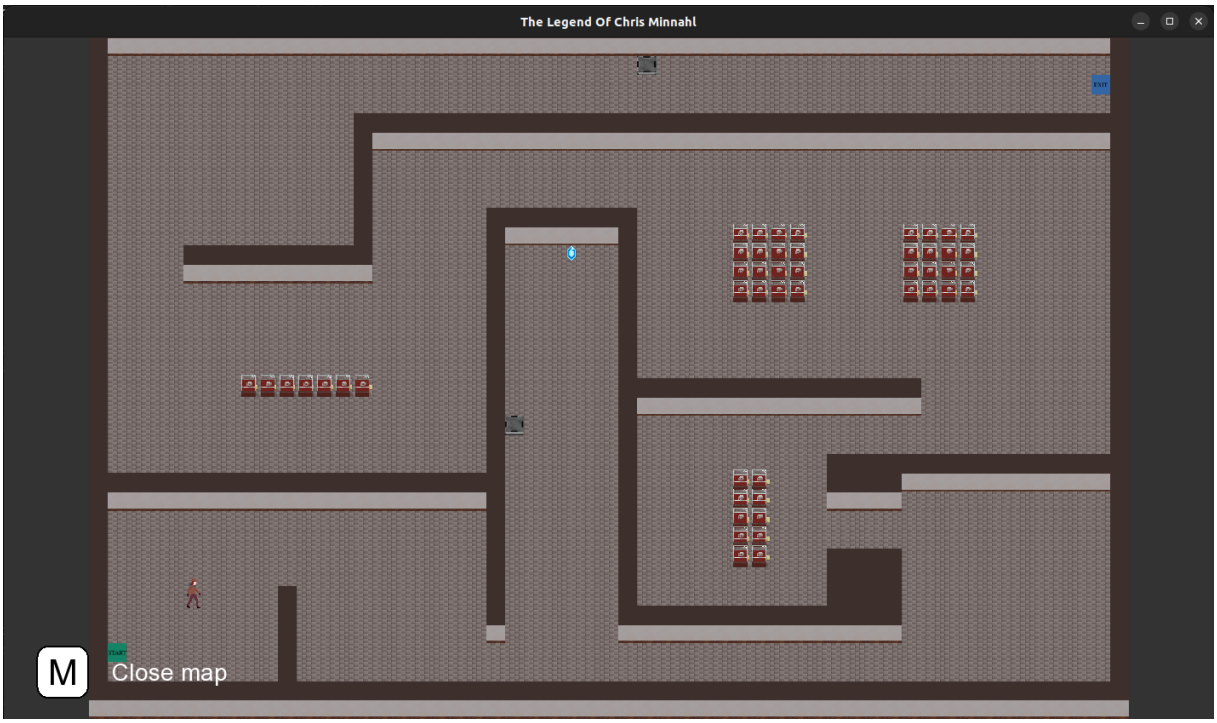
Annexe 1 : Le menu du jeu



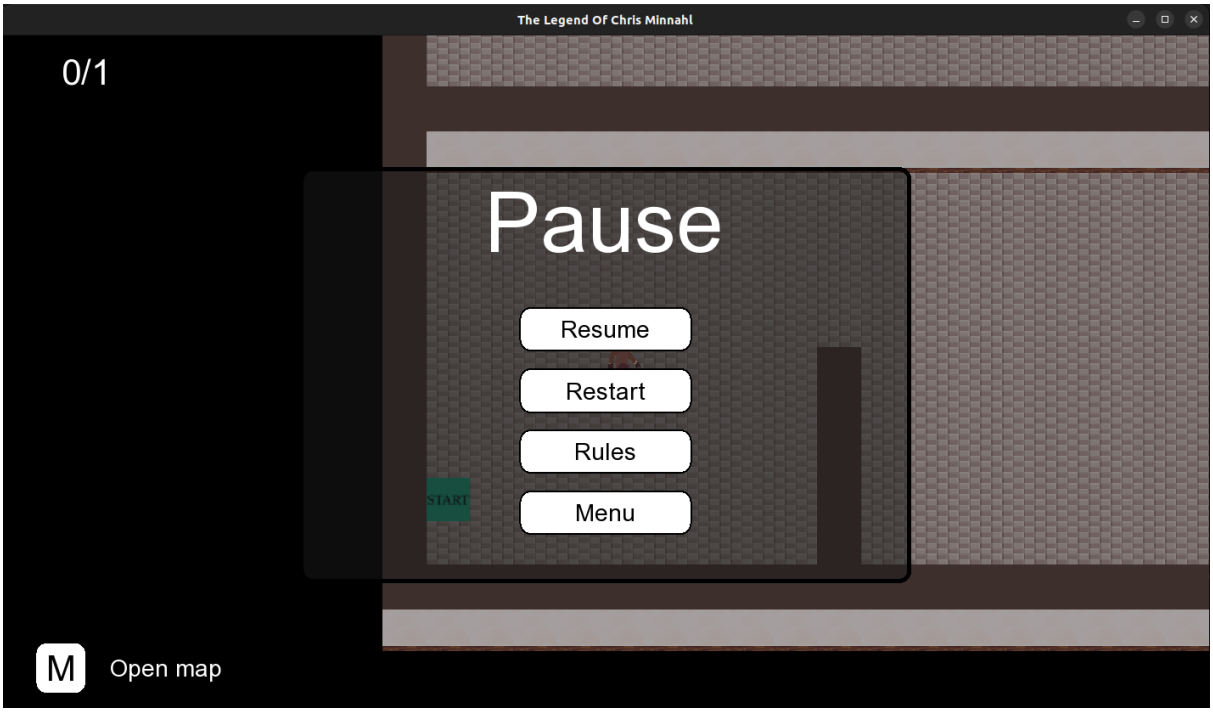
Annexe 2 : La page de règles du jeu



Annexe 3 : L'écran de jeu



Annexe 4 : La mini-map



Annexe 5 : Le menu pause

Annexe 6 : La page de fin