Projet de Programmation C : Jeu d'infiltration

L'objectif de ce projet est de développer une application graphique d'un jeu d'infiltration. Il s'agit typiquement d'un jeu d'action, dans lequel le personnage doit accomplir des tâches sans être détecté.

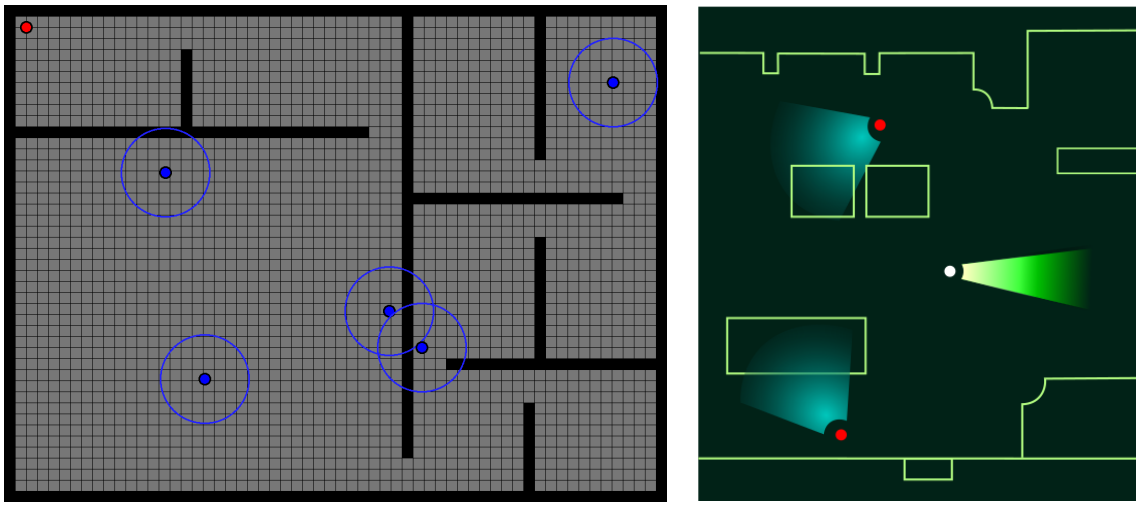

Figure 1: Gauche : une démo simple d'un jeu d'infiltration. Droite : Vision "Soliton Radar" de *Metal Gear Solid*, image par Jetijones sur Wikipedia, sous CC-BY 3.0 unported.

1 Description et déroulement du programme

Une fois le jeu lancé, une fenêtre graphique contenant un menu s'ouvre et permet à l'utilisateur de commencer une nouvelle partie de jeu ou de régler des options. Au début de la partie, le personnage se trouve à l'entrée d'une salle avec des reliques et des gardiens. Le plan de la salle est affiché, avec les positions du personnage, des gardiens et des reliques. Le but du personnage est d'infiltrer la salle, de récupérer toutes les reliques et de revenir à l'entrée, sans être repéré par les gardiens. Pour accomplir cette tâche, le personnage possède des compétences magiques. Lorsque le personnage passe sur une tuile, il ramasse les traces de mana qui s'y trouvent. En cas de besoin, il peut consommer ce mana pour se rendre invisible ou pour s'accélérer, mais la durée de l'effet est limitée par la réserve de mana du personnage. Quand le personnage réussit sa tâche, le programme affiche son temps et les meilleurs temps des autres joueurs.

2 Mécanisme du jeu

2.1 Terrain et gardiens

Le jeu se déroule dans une grille de cases de taille 60×45, murs extérieurs compris. Elle est subdivisée par des murs en plusieurs compartiments, pavés par des tuiles (les cases de la grille) et reliés par des ouvertures. Dans la salle, il y a 3 **reliques** (représentées par des tuiles spéciales) et quelques **gardiens** (5 par exemple). L'entrée de la salle est une région 2×2 en haut à gauche à l'intérieur de la salle, au centre de laquelle le personnage est initialement placé. La subdivision en compartiments, les reliques et les positions initiales des gardiens sont générées aléatoirement. Le personnage et les gardiens sont tous représentés par des disques dont le diamètre est de même longueur que le côté d'une case.

Pour que les mouvements des gardiens et du personnage soient fluides, **votre programme doit effectuer 60 frames (mise-à-jours) par seconde**. La fluidité est cruciale pour ce jeu. Pour calibrer les mouvements, fixons une certaine vitesse v sur laquelle les calculs de vitesse des gardiens et du personnage sont basés. **Cette vitesse v est cruciale pour la bonne expérience du jeu, et il est de votre responsabilité de la calibrer soigneusement.** Dans la suite, toutes les valeurs sont des suggestions, et il est conseillé de calibrer ces valeur pour que le jeu soit agréable à jouer. **Toute modification de valeur doit être impérativement justifiée dans le rapport et sous forme de commentaires dans le code source.**

Les gardiens sont des golems avec une vision à 360 degrés, mais ils ne peuvent pas voir le personnage au travers des murs, ou si la **distance euclidienne** est supérieure à une certaine valeur dépendant de la situation. S'ils détectent le personnage, alors la partie est perdue. Les gardiens se déplacent de façon aléatoire, avec deux modes :

- **Mode normal** : Le gardien se déplace dans une des directions cardinales avec une vitesse aléatoire fixée entre $0{,}3v$ et $0{,}8v$, et a une distance de détection de 4 cases. Il a une probabilité faible (par exemple $1/50$, à calibrer) de changer aléatoirement de direction et de vitesse à chaque frame. Quand il rencontre un mur, il change aussi sa direction et sa vitesse.
- **Mode panique** : Le gardien se déplace à vitesse v, et a une distance de détection étendue à 6 cases. Il change aléatoirement sa direction et sa vitesse à chaque frame comme dans le mode normal, sauf qu'il n'attend plus de rencontrer un mur, mais le fait dès qu'un mur est à une case de distance. Après 30 secondes, il revient au mode normal. L'interface doit présenter des indications visuelles claires, par exemple un cadre rouge ou clignotant, pour signaler que le mode panique est activé.

Tous les gardiens entrent en mode panique si l'un entre eux voit qu'une relique a disparu. Comme il y a 3 reliques, les gardiens entrent au plus 3 fois en mode panique. Si

pendant le mode panique un des gardiens voit qu'une autre relique a aussi disparu, la durée du mode panique est réinitialisée à 30 secondes. Pour rendre le jeu plus facile, la limite de détection de chaque gardien, qui est un cercle, doit être dessinée.

2.2 Mouvement et compétence de personnage

Le personnage est initialement placé au centre de l'entrée, et se déplace avec les flèches directionnelles ou les touches ZQSD du clavier. La vitesse maximale du personnage est $0{,}9v$, plus rapide qu'un gardien en mode normal, mais moins qu'en mode panique. Cependant, quand le personnage démarre ou change de direction, sa vitesse initiale commence à $0{,}1v$, et accélère de $0{,}03v$ par frame (ou une autre valeur qui vous semble appropriée) où la touche de direction reste enfoncée.

Au début, les tuiles contiennent des traces de mana, et le personnage les absorbe en marchant dessus. La présence ou l'absence de mana sur une tuile est indiquée par un changement de couleur. Le joueur peut dépenser ce mana pour deux compétences :

- **Accélération surchargée** : déclenchée en enfonçant l'une des touches Shift, cette compétence permet au personnage de dépasser la vitesse de $0{,}9v$ pour atteindre une vitesse maximale de $1{,}2v$. Cette compétence consomme l'équivalent de mana de deux tuiles par frame.
- **Invisibilité** : déclenchée par la touche Espace, cette compétence rend le personnage non détectable, en consommant à chaque frame le mana d'une tuile.

Le mana dépensé retourne sur des tuiles choisies aléatoirement et dont les traces de mana avaient été absorbées. Les compétences ne peuvent pas être utilisées si le personnage n'a pas assez de mana.

3 Génération de terrain

La grille étant de taille 60×45, en prenant en compte les murs extérieurs, l'espace libre initial est de taille 58×43. Nous appliquons l'algorithme récursif suivant à cet espace libre pour générer les murs intérieurs et ainsi constituer les compartiments et les ouvertures. Cet algorithme garantit que les côtés de chaque compartiment sont au moins d'une longueur minimale fixée `minside`. La valeur suggérée de `minside` est 9, mais vous pouvez calibrer cette valeur autrement en motivant votre choix.

1. Supposons que l'espace libre est de taille $x \times y$. On suppose ici que $x > y$. Sinon, le raisonnement est le même en inversant x et y.
2. Si $x < 2 \times \texttt{minside} + 1$, comme une subdivision créera un espace libre trop petit, l'algorithme termine.

3. Si $x < 4 \times \texttt{minside}$, comme l'espace libre a une taille raisonnable, alors l'algorithme termine avec une probabilité $1/2$.

4. Sinon, on crée deux compartiments en divisant la longueur x en deux. Les deux moitiés créées peuvent être de longueurs différentes, mais chacune doit être plus grande que `minside`. Les deux espaces obtenus doivent être séparés par un mur d'une case de largeur, avec une ouverture de 3 cases à une des extrémités choisie aléatoirement. Puis on relance l'algorithme récursivement sur les deux compartiments générés.

Après la subdivision, les gardiens et les reliques sont placés aléatoirement à l'intérieur des compartiments, donc pas sur les ouvertures, et aussi à une distance euclidienne d'au moins 20 cases de la position initiale du personnage. Voir la partie gauche de la Figure 1 pour un exemple de résultat de l'algorithme.

4 Détection de collision avec les murs

Pour une première version, il est conseillé de commencer par implanter un déplacement discret, qui rend la détection de collision avec les murs très facile : il suffit de ne pas y entrer ! Cependant, pour que le jeu soit plus naturel, on vous demande d'implanter le déplacement continu, avec une détection de collision correctement calculée.

Comme les gardiens et le personnage sont représentés par des disques avec des déplacements quasi-continus, la détection des collisions avec les murs ne se réduit pas à un test simple. Il est en effet nécessaire de mesurer la distance entre l'agent (gardien ou personnage) et les cases de mur dans la direction du déplacement. Un agent est représenté par un disque dont le diamètre est de même longueur que le côté d'une case.

On commence par établir un système de coordonnées de la fenêtre. Le point d'origine est le coin en haut à gauche de la grille, avec l'axe x vers la droite et l'axe y vers le bas. L'unité de distance est le côté d'une case. Considérons la case se trouvant sur la colonne i et la ligne j. Les coordonnées de son coin en haut à gauche sont alors (i, j). Par abus de notation, on notera cette case (i, j), la case en haut à gauche étant la case $(0, 0)$.

Supposons que le centre du disque est au point (x_0, y_0). On suppose que le disque ne chevauche aucune case de mur, et que le disque avance vers le bas. Il est clair que le centre du disque se situe dans la case $(\lfloor x_0 \rfloor, \lfloor y_0 \rfloor)$, où $\lfloor z \rfloor$ est la partie entière de z. Quand le disque se déplace vers le bas, c'est-à-dire quand sa coordonnée y augmente, il y a trois possibilités de rencontre avec un mur :

(a) Si la case immédiatement en bas $(\lfloor x_0 \rfloor, \lfloor y_0 \rfloor + 1)$ est une case de mur, alors la coordonée y du centre du disque ne peut pas augmenter au-delà de $\lfloor y_0 \rfloor + 1/2$, pour éviter de croiser le bord situé sur la ligne $y = \lfloor y_0 \rfloor + 1$.

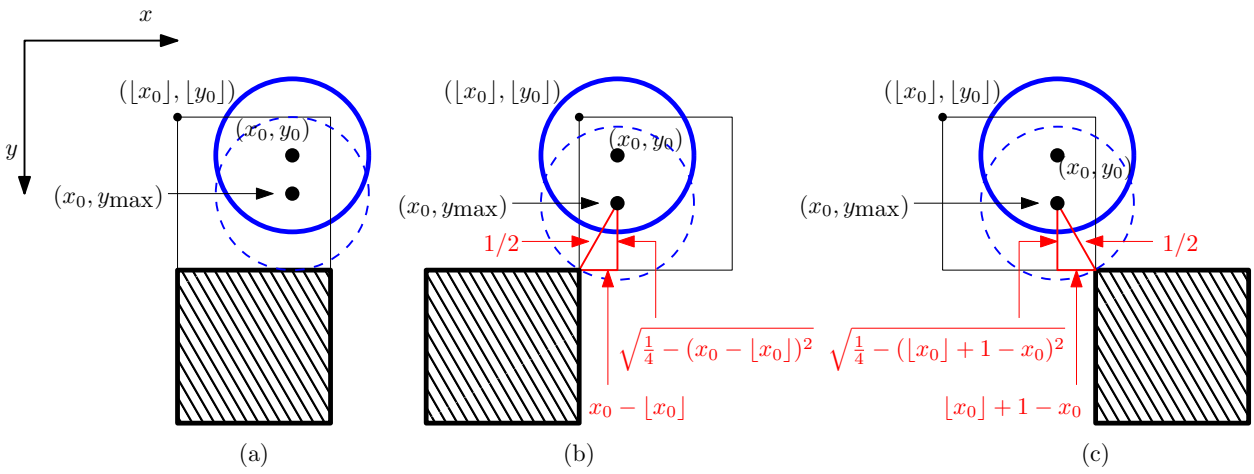

Figure 2: Trois cas de collision possible

(b) Si la case en bas à gauche $(\lfloor x_0 \rfloor - 1, \lfloor y_0 \rfloor + 1)$ est une case de mur, et que $x_0 < \lfloor x_0 \rfloor + 1/2$, alors la coordonée y du centre du disque ne peut pas aller au-delà de

$$\lfloor y_0 \rfloor + 1 - \sqrt{\frac{1}{4} - (x_0 - \lfloor x_0 \rfloor)^2},$$

pour éviter le chevauchement avec le point $(\lfloor x_0 \rfloor, \lfloor y_0 \rfloor + 1)$.

(c) Si la case en bas à droite $(\lfloor x_0 \rfloor + 1, \lfloor y_0 \rfloor + 1)$ est une case de mur, et que $x_0 \geq \lfloor x_0 \rfloor + 1/2$, alors la coordonée y du centre du disque ne peut pas aller au-delà de

$$\lfloor y_0 \rfloor + 1 - \sqrt{\frac{1}{4} - (\lfloor x_0 \rfloor + 1 - x_0)^2},$$

pour éviter le chevauchement avec le point $(\lfloor x_0 \rfloor + 1, \lfloor y_0 \rfloor + 1)$.

La Figure 2 illustre les trois cas possibles. Attention, il se peut qu'un déplacement tombe simultanément dans **plusieurs cas** ! Une fois que les limites de déplacement sont calculées, le disque peut être déplacé jusqu'à la limite. Si on n'est dans aucun cas, alors le disque peut être déplacé jusqu'à ce que son centre arrive dans la case en dessous, et on doit alors réévaluer les conditions pour continuer le déplacement. Les déplacements dans les autres directions cardinales sont traités de façon similaire.

5 Détection du personnage par les gardiens

Pour simplifier la programmation, on suppose qu'un gardien ne détecte le personnage que s'il voit directement son centre, c'est-à-dire que le segment reliant leur centres (la ligne de vue) ne traverse pas de case de mur. Ainsi, le problème de détection se réduit au calcul des cases traversées par un segment donné.

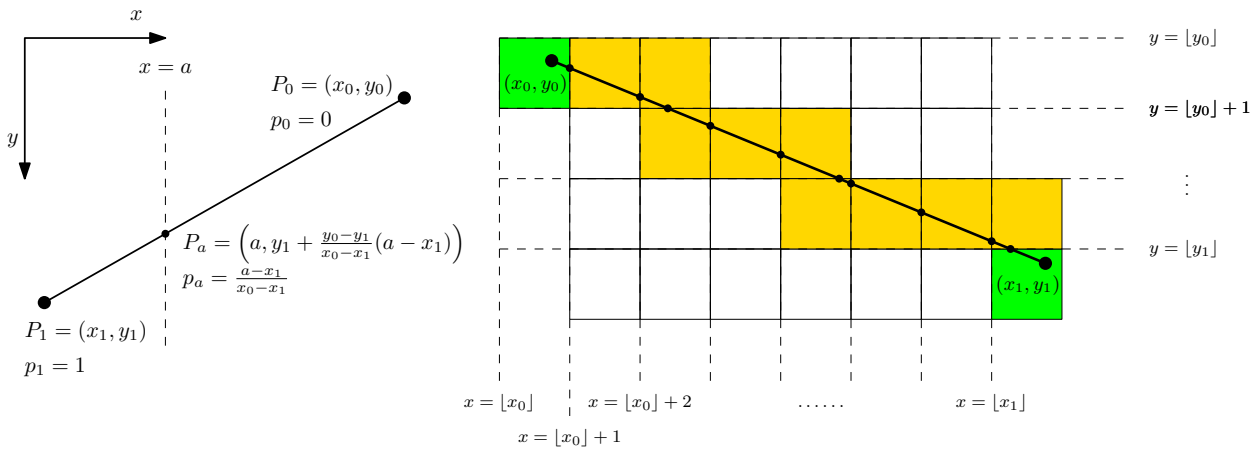

Figure 3: Calcul des intersections et des cases traversées par un segment

Supposons que le personnage se situe à $P_0 = (x_0, y_0)$ et le gardien à $P_1 = (x_1, y_1)$. Il faut alors calculer les cases croisées par le segment $[P_0, P_1]$. En géométrie analytique, les points du segments ont des coordonnées de la forme suivante :

$$(px_0 + (1-p)x_1, py_0 + (1-p)y_1)\,, \text{ avec } 0 \leq p \leq 1.$$

On vérifie que la valeur $p = 0$ donne bien le point P_0, et $p = 1$ donne bien P_1. Ici, la valeur p représente la position relative du point sur le segment $[P_0, P_1]$. Il est clair que, si le point est dans une certaine position relative (représenté par la valeur de p) en abscisse entre x_0 et x_1, alors il est dans la même position relative en ordonnée entre y_0 et y_1, d'où la formule.

Pour trouver les cases traversées, on commence par trouver les côtés des cases qui sont traversées. Comme ce sont des segments sur une droite de la forme $x = a$ ou $y = b$, il suffit de calculer les intersections entre ces droites et le segment $[P_0, P_1]$. Plus précisément, pour calculer l'intersection de $x = a$ avec le segment $[P_0, P_1]$, il suffit de trouver la valeur p_a correspondant à $x = a$:

$$p_a x_0 + (1 - p_a)x_1 = a \Leftrightarrow p_a = \frac{a - x_1}{x_0 - x_1}.$$

Donc l'intersection se trouve au point (a, y_a) avec

$$y_a = p_a y_0 + (1 - p_a)y_1 = y_1 + \frac{y_0 - y_1}{x_0 - x_1}(a - x_1).$$

Il faut bien vérifier que $0 \leq p_a \leq 1$ pour garantir que le point d'intersection se trouve à l'intérieur du segment $[P_0, P_1]$. L'intersection au point (a, y_a) se trouve sur le côté allant de $(a, \lfloor y_a \rfloor)$ à $(a, \lfloor y_a \rfloor + 1)$. Les deux cases adjacentes à ce segment sont donc les cases $(a - 1, \lfloor y_a \rfloor)$ et $(a, \lfloor y_a \rfloor)$, représentées par les coordonnées de leur coin en haut à gauche.

Si l'une de ces cases est un mur, alors la ligne de vue du gardien est interrompue et le joueur n'est pas détecté.

Le cas des droites de la forme $y = b$ est traité de façon similaire. Voir la partie gauche de la Figure 3 pour une illustration.

Pour trouver tous les cases traversées, il suffit répéter ce calcul pour toutes les droites entre P_0 et P_1, et ainsi trouver tous les points d'intersection entre le segment et un côté de case. Certaines cases peuvent avoir plusieurs points d'intersection, et donc être traitées plusieurs fois. Voir la partie droite de la Figure 3 pour un exemple. Si n'importe laquelle de ces cases est un mur, alors le gardien ne voit pas le personnage. Sinon, le joueur est détecté et la partie est perdue.

6 Tableau de classement

À la fin de la partie, on affiche le temps réalisé par le joueur et la quantité de mana dépensée. Si la partie est gagnée, le joueur peut rentrer son nom dans le tableau des meilleurs scores, affiché en fin de partie. Il y a deux classements : par meilleur temps et par meilleure consommation de mana. Les classements doivent être enregistrés dans un **fichier binaire** pour décourager la triche. On peut limiter la longueur du nom rentré pour simplifier l'enregistrement.

7 Gestion de framerate

Pour gérer le taux de mise-à-jour de l'affichage, deux choses sont fondamentales :

- Votre programme ne demande pas trop de ressources.
- Vous limitez le nombre d'appels à la fonction `MLV_update_window`.

Votre fonction main ou bien une autre fonction gérant globalement votre jeu devra comporter une boucle maîtresse qui gère le jeu frame par frame. Tant que le joueur ne veut pas quitter, on met l'affichage à jours, on résout tous les événements sur la frame, puis on calcule le temps passé et on ajuste la vitesse du jeu en attendant quelques millisecondes.

```
1 /* Main loop over the frames ... */
2 while(!quit){
3     /* Some declaration of variables */
4     ...
5
6     /* Get the time in nanoscond at the start of the frame */
7     clock_gettime(CLOCK_REALTIME, &end_time);
```

```
8       /* Display of the current frame, sample function */
9       /* THIS FUNCTION CALLS ONCE AND ONLY ONCE MLV_update_window */
10      draw_window(&param, &grid);
11
12      /* We get here some keyboard events */
13      event = MLV_get_event(...);
14
15      /* Dealing with the events */
16      ...
17      /* Move the entities on the grid */
18      ...
19      /* Collision detection and other game mechanisms */
20      ...
21
22      /* Get the time in nanosecond at the end of the frame */
23      clock_gettime(CLOCK_REALTIME, &new_time);
24      /* Compute the time spent for the current frame */
25      frametime = new_time.tv_sec - end_time.tv_sec;
26      frametime += (new_time.tv_sec - end_time.tv_sec) / 1.0E9;
27
28      /* Force the program to spend at least 1/60 second in total */
29      extratime = 1.0 / 60 - frametime;
30      if(extratime > 0){
31          MLV_wait_milliseconds((int)(extratime * 1000));
32      }
33  }
```

Pour que la fonction `clock_gettime` soit disponible sous ANSI, il faut définir au début du fichier d'en-tête le macro `_POSIX_C_SOURCE` avec la valeur `199309L`. Pour plus d'information, voir `man 3 clock_gettime`.

8 Interface graphique

Pendant une partie, toute la salle (grille 60×45, murs extérieurs compris) doit être affichée. Les déplacements des gardiens et du personnage doivent être fluides. Le contrôle du personnage doit être réactif. **La fluidité et la réactivité ont un impact important sur la jouabilité du jeu, et seront prises en compte dans l'évaluation finale.** La zone de détection de chaque gardien, dont le rayon peut changer pendant la partie, doit être dessinée avec un cercle. L'interface doit indiquer clairement quand les gardiens sont en mode panique. De même, l'utilisation des compétences (invisibilité ou accélération) doit être clairement représentée, par exemple en modifiant l'apparence du personnage.

Pour réaliser l'interface graphique, vous utiliserez obligatoirement la bibliothèque graphique C de l'université : `libMLV`. Une documentation de cette bibliothèque est accessible à l'adresse `http://www-igm.univ-mlv.fr/~boussica/mlv/api/French/html/index.html`. Elle est déjà installée sur toutes les machines de l'université. Pour l'installer sur votre propre machine sur Ubuntu (et d'autres distributions avec `apt`), il suffit d'installer `libmlv3` et `libmlv3-dev` avec `apt`. Pour les autres distributions, il y a normalement les mêmes paquets dans le système de gestion de paquets. Si vous êtes sur Windows, il est conseillé d'installer Ubuntu avec WSL (*Windows Subsystem of Linux*), ou bien d'utiliser une machine virtuelle. Si vous avez un Mac, alors la seule solution sera d'installer une machine virtuelle.

Ce choix de la libMLV est une obligation[1]. Il y a mieux, il y a aussi moins bien... On attend de vous que vous vous familiarisiez avec une bibliothèque écrite par un autre, et que sa documentation vous suffise à construire une véritable application graphique.

9 Modularité

Votre projet doit organiser ses sources sémantiquement. Votre code devra être organisé en modules rassemblant les fonctionnalités traitant d'un même domaine. Un module sans entêtes pour le main, un module pour le moteur du jeu, un module pour la gestion du terrain et un module graphique sont un minimum. Le moteur du jeu sera vraisemblablement très volumineux et on peut même imaginer le subdiviser au besoin (module pour les gardiens, module pour les collisions, module pour les reliques...).

Votre projet devra être compilable via un Makefile qui exploite la compilation séparée. Chaque module sera compilé indépendamment et seulement à la fin, une dernière règle de compilation devra assembler votre exécutable à partir des fichiers objets en faisant appel au linker. Le flag -lMLV est normalement réservé aux modules graphiques ainsi qu'à l'assemblage de l'exécutable (sinon, c'est que votre interface graphique dégouline de partout et que vos sources sont mal modulées).

10 Pour aller plus loin

Pour les plus récalcitrants d'entre vous, toute amélioration sera considérée comme un plus pour l'évaluation. Voici quelques suggestions possibles de raffinements pour ce projet. Ces propositions sont gradués avec un indice de difficulté entre parenthèse.

[1]Certains cas particuliers justifient d'utiliser la bibliothèque `libsdl2`, dont la prise en main est plus difficile. Cette permission ne sera accordée qu'au cas par cas, après discussion avec le responsable du cours. Toute utilisation de `libsdl2` sans autorisation sera sanctionnée par une note de 0/20 à l'ensemble du projet.

- (blob) Ajouter des effets visuels et/ou sonores pour la récupération de mana et l'utilisation des compétences.
- (blob) Ajouter d'autres types de gardiens avec des règles de déplacement différentes. Les mouvements produits doivent être convaincant, et ne doivent pas dégrader la jouabilité.
- (gobelin) Permettre un choix du niveau de difficulté, par exemple en modifiant le nombre de gardien, la vitesse des gardiens, la durée du mode panique, etc. Ces choix ne doivent pas dégrader la jouabilité.
- (gobelin) Ajouter un mécanisme d'évaporation de mana : le mana libéré par l'utilisation des compétences est volatile et s'évapore après un certain temps. Par contre, le mana présent initialement sur le plateau ne s'évapore pas.
- (paladin) Ajouter des mini-jeux (type *Among Us*, par exemple relier des cables de couleurs différentes, parier des objets, *etc.*) à la récupération des reliques pour "forcer le coffre", pendant lequel les gardiens bougent et détectent normalement. Le joueur doit donc trouver un moment opportun et se dépêcher de résoudre le mini-jeu sans être capturé.
- (paladin) Proposer un autre algorithme intéressant pour la génération du terrain, sans modifier les autres éléments du jeu. Typiquement, il faut que toutes les tuiles soient accessibles, et que la jouabilité ne soit pas perturbée.
- (paladin) Modifier le mécanisme de détection de sorte que les gardiens détectent le personnage s'ils voient n'importe quel point de son disque, et pas seulement son centre.
- (dragon) Modifier le mécanisme de détection pour le rendre plus réaliste, en introduisant la notion de direction de vue pour les gardiens. Ainsi, les gardiens ne voient plus à 360 degrés, mais leur champs de vision est limité à un éventail centré dans la direction où ils regardent. Pour la présentation graphique, on peut se contenter de dessiner les deux rayons qui délimitent le champs de vision. Le comportement des gardiens sera modifié pour ajouter une phase pendant laquelle le gardien s'arrête et tourne sur lui-même pour regarder dans toutes les directions. La largeur du champs de vision sera aussi élargie pendant le mode panique.
- (dragon) Au lieu de dessiner le rayon de détection, dessiner la vraie zone de détection, en prenant en compte les zones cachées par les murs. De plus, la zone doit être dessinée en semi-transparence. Attention, le bord de la zone de détection sera un mélange de courbes et de segments rectilignes.
- (Jörmungandr) Combiner correctement les deux améliorations précédentes.

Attention, **tout changement déteriorant la jouabilité sera lourdement pénalisé** ! Faites attention à bien calibrer les paramètres et les nouveaux mécanismes !

11 Conditions de développement

Le but de ce projet est moins de pondre du code que de développer le plus proprement possible. C'est pourquoi vous développerez ce projet en utilisant un système de gestion de versions Git. Le serveur à disposition des étudiants de l'Université est disponible à cette adresse : `https://forge-etud.u-pem.fr/`. Comme nous attendons de vous que vous le fassiez sérieusement, votre rendu devra contenir un dump du fichier de logs des opérations effectuées sur votre projet, extrait depuis le serveur de gestion de versions ; afin que nous puissions nous assurer que vous avez bien développé par petites touches successives et propres (commits bien commentés), et non pas avec un seul commit du résultat la veille du rendu. L'évaluation tient compte de la capacité du groupe à se diviser équitablement le travail.

Voici quelques **remarques importantes** :

- L'intégralité de votre application doit être développée exclusivement en langage C (la documentation technique dynamique fait évidemment exception). Toute utilisation de code dans un autre langage (y compris C++) vaudra 0 pour l'intégralité du projet concerné.
- En dehors des bibliothèques standards du langage C et de `libMLV`, il est interdit d'utiliser du code externe : vous devrez tout coder vous-même. Toute utilisation de code non développé par vous-même vaudra 0 pour l'intégralité du projet concerné.
- Tout code commun à plusieurs projets vaudra 0 pour l'intégralité des projets concernés.

12 Conditions de rendu

Vous travaillerez en **binôme** et vous lirez avec attention la Charte des Projets. Il faudra rendre au final une archive `tar.gz` de tout votre projet (tout le contenu de votre projet `git`), les sources de votre application et ses moyens de compilation. Il sera alors crucial de lire des recommandations et conseils d'utilisation de `git` sur la plate-forme `e-learning`. Vous devrez aussi donner des droits d'accès à votre chargé de TP et de cours à votre projet via l'interface `redmine`.

Un exécutable devra alors être produit à partir de vos sources à l'aide d'un `Makefile`. Naturellement, toutes les options que vous proposerez (ne serait-ce que `--help`) devront être gérées avec `getopt` et `getopt_long`.

La cible `clean` doit fonctionner correctement. Les sources doivent être propres et bien commentées. C'est bien de se mettre un peu à l'anglais si possible.

Votre archive devra aussi contenir :

- Un fichier `log_dev` correspondant au dump des logs de votre projet (nom des commits, qui? et quand?), extrait depuis le serveur de gestion de versions que vous aurez utilisé.
- Un fichier `Makefile` contenant les règles de compilation pour votre application ainsi que tout autre petit bout de code nécessitant compilation (comme les tests par exemple).
- Un dossier `doc` contenant la documentation technique de votre projet ainsi qu'un fichier `rapport.pdf` contenant votre rapport qui devra décrire votre travail. Si votre projet ne fonctionne pas complètement, vous devrez en décrire les bugs.
- Un dossier `src` contenant les sources de votre application.
- Un dossier `include` contenant tous les headers de vos différents modules.
- Un dossier `bin` contenant les fichiers objets générés par la compilation séparée.
- Aucun fichier polluant du type `bla.c~` ou `.%smurf.h%` généré par les éditeurs. Votre dossier doit être propre !
- Si possible, la partie `html` générée par l'utilitaire `doxygen` à partir de votre application de visualisation. Ceci est optionnel mais tellement plus propre.

Sachant qu'un script automatique scrupuleux va analyser et corriger votre rendu avant l'oeil humain, le non-respect des précédentes règles peut rapidement avorter la correction de votre projet. Le respect du format des données est une des choses des plus critiques.