Le Mans Université

Institut Informatique Claude Chappe

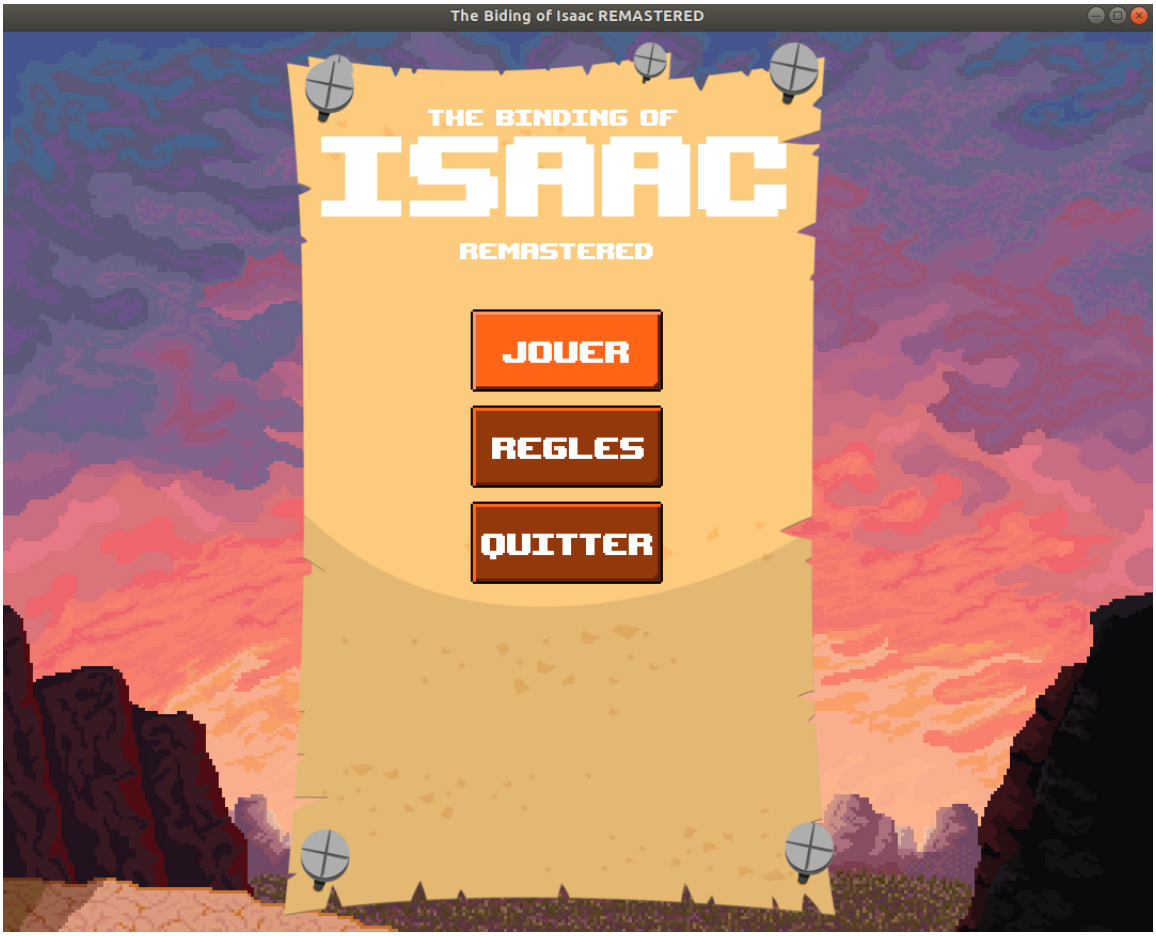
Le Mans Université

Licence Infomatique *2ème année*

Compte rendu : The Binding of Isaac Remastered

Paul Parent du Chatelet, Paul Chauvière, David Chanteau, Adrien Pitault

April 22, 2019



1 Introduction

The Binding of Isaac est un rogue-like au gameplay simple. Nous contrôlons un personnage qui se déplace de salles en salles en tirant sur des monstres et en cherchant des items afin de pouvoir tuer les boss et avancer le plus loin possible dans ce labyrinthe. Certaines salles se ferment derrière le joueur tant qu'il n'a pas tué tous les monstres dans celle-ci et il y a même un système de salle cachée que le joueur peut trouver en cassant les murs avec des bombes. Le joueur gagne lorsqu'il a battu tous les boss de la map créée aléatoirement.
Dans notre version nous avons décidé de simplifier toutes ces caractéristiques et de les adapter afin de pouvoir développer un jeu dans le temps qui nous était donné.
L'objectif de notre jeu est donc de trouvé une clé et tuer tous les monstres dans le labyrinthe. Si ces deux conditions sont complétées, le portail de la salle final s'ouvre et le joueur remporte la partie.

Le cahier des charges pour notre programme était le suivant :
- Création aléatoire de la map
- Génération des salles et affichage : portes, monstres et salle finale
- Contrôle personnage : déplacements et changement de salle, attaques
- Projectiles tirés par le joueur et certains monstres
- Comportement des monstres

Idées bonus :
- Interface graphique (passage à des coordonnées et des hitbox)
- Minimap
- Echelle de difficulté
- Salles de boss
- Items

Notre jeu sera donc une version nettement simplifiée au gameplay bien moins nerveux mais aura un aspect plus "old school".

Contents

2 Organisation du travail

2.1 Planification et répartition du travail

L'organisation induit l'efficacité et la qualité de travail. C'est pour cela que pour mener à bien notre projet, l'organisation de notre temps et de notre travail était primordial.
Avant de nous pencher sur le codage des fonctionnalités que nous voulions, nous avons élaboré un cahier des charges, contenant les idées (triées) et les objectifs de notre jeux.
Nous avons tout d'abord choisis de réaliser les caractéristiques du type de jeux dont nous nous inspirions, le rogue-like. Cela comprend donc la génération d'une map aléatoire en début de partie, une génération de salle selon la map, le contrôle d'un joueur qui pourra se balader de salle en salle, ou encore créer des monstres avec un comportement spécifique, visant à empêcher le joueur de finir le jeux.
Nous avons également listé différentes tâches bonus, dont la réalisation ont dépendu du bon avancement de notre projet, ou bien d'un changement de priorité de nos principales fonctionnalités.

Une fois le cahier des charges complété, il fallait donc nous répartir la réalisation des tâches.
Nous avons donc réalisé un planning sur Google Sheet(*Cf. Figure 6 : Planning du projet*), nous permettant de nous tenir au courant de l'évolution de notre projet, savoir quel tâche est terminée, qui est en retard pour que nous puissions l'aider.
Le planning englobe la totalité de la durée du projet, ainsi nous nous fixions un certain nombre de semaines chacun pour la réalisation de nos tâches. Même si nous étions parfois plusieurs sur une tâche, l'un de nous en était le responsable. Nous avons également établi un code couleur avec une échelle de 0 à 100% pour avoir un visuel rapide et précis sur notre avancement. Une colonne de commentaires à également été ajoutée pour chacune des tâches afin que nous indiquions nos idées pour améliorer certaines taches, indiquer ce qu'il restait à implémenter etc...
Nous avons également établis des échéances représentant une étape majeure dans l'avancée du projet.
La première étant de permettre au joueur de se déplacer entre les différentes salles d'une map générée aléatoirement en début de partie. La deuxième était d'avoir terminé une version entièrement jouable dans le terminal, puis la troisième était de réaliser une version avec un portage SDL.

Durant la totalité de notre projet, nous avons également eu recourt à différents outils, rendant ainsi notre travail plus efficace.
Google Drive était notre principal outil, nous y déposions chaque semaine des dossiers représentatifs de nos mises en commun et de l'avancement globale du projet. Ces dossiers étaient renommés, de sortes que nous puissions voir immédiatement la date de dépôt, l'ajout majeur réalisé ainsi que la personne qui l'a déposé.
Nous avons également utilisé, comme demandé, GitHub pour y déposer nos fichiers. Cependant, nous ne nous en somme pas majoritairement servis car nous connaissions déjà parfaitement le fonctionnement de Google Drive, nous le trouvions plus visuel et simple d'utilisation.
Ayant chacun travaillé en dehors des séances déjà dédiées au projet, il était important de pouvoir garder contact pour se poser des questions etc. Disposant chacun de Messenger (Facebook), nous y avons donc créer une conversation prévu à cet effet.

Enfin, le dernier outil nous ayant aidé dans notre organisation à été Doxygen. Nous l'avons manipulé pour commenter notre code et ainsi avoir un visuel sur toutes nos fonctions. Cela nous a donc permis, en cas de doute sur le fonctionnement d'une fonction ou encore le contenu d'un structure, de nous y retrouver très rapidement.

3 Développement

3.1 Génération de la map

Comme précisé dans le cahier des charges, notre jeu nécessite une génération aléatoire de la map. Cependant, avant de se lancer dans cette tâche, il est nécessaire de définir clairement ce qu'est une map. Pour notre projet, une map est définie comme étant une matrice composée de salles (structures de salles en l'occurrence) rattachées les unes aux autres, formant ainsi un labyrinthe.
Ainsi l'initialisation de la map est assez simple et demande seulement la mise à NULL de toute les cases de la map sauf celle au centre de la matrice où une salle est créée, celle où le joueur apparaît en début de partie.

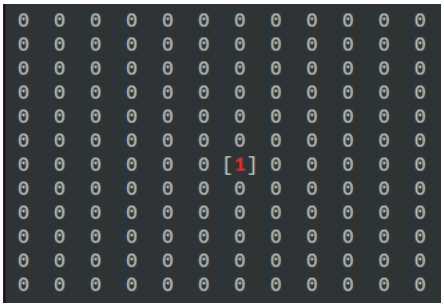


Figure 1: Placement de la première salle

L'initialisation faite, il faut maintenant générer le reste de la map de manière procédurale. L'ajout des coordonnées des salles possibles dans une liste se fait par rapport aux salles déjà existantes sur les cases vides adjacentes (la diagonale n'est donc pas prise en compte) puis via le parcours de cette liste qui ajoute ainsi une salle à ces coordonnées dans la matrice de la map.
Cependant il faut d'abord indiquer que l'ajout des coordonnées dans la liste est possible. De cette manière nous avons déterminé quatre cas éliminatoires :

- les coordonnées sont en dehors de la matrice
- la valeur de la matrice à ces coordonnées est différente de NULL
- les coordonnées sont déjà dans la liste de coordonnées
- l'ajout de ces coordonnées feraient apparaître un carré de 4 salles

Une fois l'ajout des coordonnées adjacentes à la salle étudiée finis, le choix des coordonnées où la salle sera créée est effectué avec la fonction rand de la bibliothèque "time.h" afin de déterminer quel élément de la liste est pioché.
La salle est ajouté à la matrice de la map et l'ajout des autres salles se fait ainsi de suite en positionnant toujours les coordonnées sur la dernière salle créée dont les coordonnées sont ôtées de la liste.

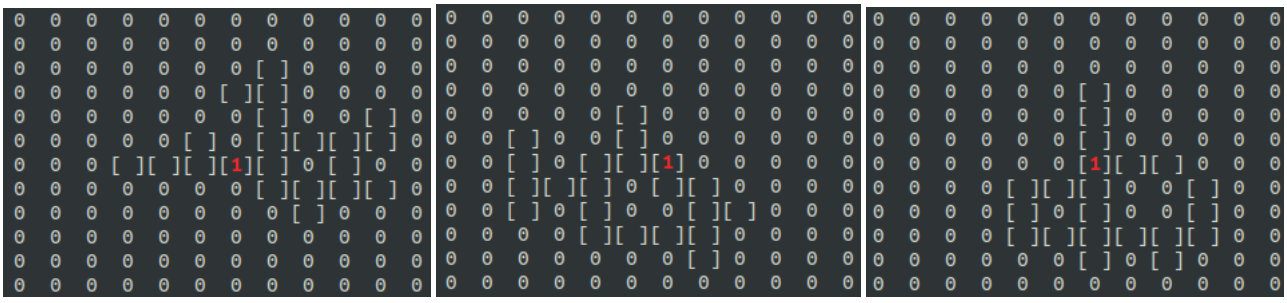


Figure 2: Génération aléatoire de plusieurs map

La matrice ainsi obtenue est celle utilisée lors de l'affichage de la mini-map afin de permettre au joueur de se repérer dans son environnement. Dans ce but, chaque structure salle possède un état qui est soit [VU,CACHE,VIDE].
Ainsi une salle ayant l'état VU est une salle déjà visitée ou étant adjacente à celle où nous nous situons (jaune). Une salle ayant l'état VIDE est une salle dont tout les monstres ont été tués (verte). Et une salle ayant l'état CACHE est une salle n'ayant pas encore été VU (invisible).
Dans la version finale du jeu, la mini-map est évolutive, elle se découvre en même temps que le joueur voyage de salle en salle.

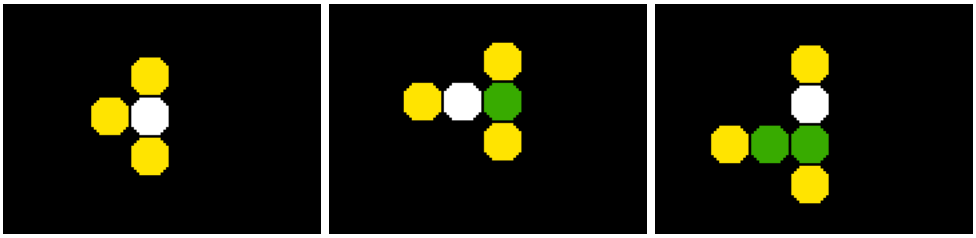


Figure 3: Exemple de mini-map en évolution

3.2 Génération d'une salle

Les salles sont créées lors de la génération de la map, mais la matrice m_salle, qui indique tout ce qui se trouve dans la salle (mur, joueur, clé, monstre ...), reste vide.
Il faut donc remplir la matrice de chaque structure salle. Pour cela nous avons créé la fonction init_salle qui prend en paramètre la matrice map ainsi que les coordonnées de la salle sur la matrice map.

3.2.1 Création des murs et des portes

La création des mur est assez simple et est identique pour toutes les salle. On place les murs aux extrémités de la salle, pour cela il suffit d'initialiser les cases de la matrice où x est égale à zéro ou à la largeur de la matrice - 1 et les cases où y est est égale à zéro ou à la longueur de la matrice - 1 à la valeur MUR .

Pour les portes on regarde pour côté si il y a un salle de créé, si c'est la cas on créer une porte de ce côté. Les portes sont créé à la place des murs au milieu de la longueur ou de la largeur de la matrice salle en fonction du côté où elle sont créés.

3.2.2 Création des obstacles

Pour la création des obstacles nous avons décidé de créer les obstacles à la main, mais pour ne pas que toutes les salles soit identiques ou pour ne pas avoir à faire un paterne d'obstacles pour chaque salle, nous avons décidé de séparer la salle en quatre partie et de faire des paternes obstacles différents pour chaque partie. Actuellement, nous avons trois paternes différents pour chaque parties de la salle soit 12 au total mais il est possible d'en ajouter assez facilement.
Ces paternes sont stockés sous forme de coordonnées dans un fichier txt nommé pattern.txt. Pour créer les obstacles d'une salle nous voulons donc tirer au hasard un des trois paternes pour chaque parties de la salle, nous obtenons ainsi une génération pseudo-aléatoire avec un total de 81 possibilités de salles différentes .
Pour tirer au sort puis recopier le paterne dans la salle, nous avons définit une matrice m_paterne [4] [NB_PATTERN] [M] [N] avec :

- 4 en première dimension pour chaque partie de la salle
- NB_PATTERN en deuxième dimension pour chaque paterne
- M et N pour représenter une salle et stocker les paternes

On commence par initialiser toutes les valeurs de m_pattern à zéro puis on appelle la fonction remplir_pattern qui va nous permettre de charger tous les paternes d'obstacles pour chaque partie de la salle dans m_pattern. Pour cela, cette fonction lis le fichier pattern.txt. On initialise deux entiers coin et choix à zéro, la variable coin indique la partie de la salle qui est remplit et choix le paterne d'obstacle.
On lis les coordonnées deux par deux avec un entier x qui prend le 1^{er} caractère lu et un entier y le deuxième. Si x et y sont positifs, on on fait m_pattern[coin][choix][x][y] = 1 ; pour indiquer qu'il y a un obstacle à ces coordonnées.
Si x et y sont égale à -1, on incrémente choix de un pour passer au paterne suivant. Si x et égale a -1 et y a -2, on incrémente coin de un pour changer de partie et on remet choix à zéro. Puis on lis les deux prochains caractères tant que coin est inférieur à 4 et que choix est inférieur a NB_PATTERN. Maintenant que m_pattern contient tous les patterns, on prend quatre nombres aléatoires entre zéro et NB_PATTERN pour chaque partie de la salle. On parcours m_pattern et regarde si l'un des quatre patterns est égale a 1 et ce pour toute coordonnées i et j. Si c'est la cas on affecte à la salle aux coordonnées i,j la valeur MUR pour préciser qui il a un obstacle a cette endroit de la salle. Cas de la salle finale :
La salle finale étant spéciale,les générations des obstacles est différente, on appelle la fonction init_salle_finale qui charge des obstacles prédéfinies et spécifiques à cette salle.

3.3 Contrôle du personnage

La structure joueur est composée de plusieurs variables renseignant :
- Son nombre de point de vie (pv)
- La salle dans laquelle il se trouve (x_map, y_map)
- Où il se situe dans cette salle (x_salle, y_salle)
- Dans quelle direction l'utilisateur veut le faire bouger (x_mouv, y_mouv)
- Quelle est sa dernière direction (x_direction, y_direction)
- S'il possède la clé pour sortir (cle)
- S'il a gagné la partie (victoire)

En premier lieu, dans la version de notre jeu qui se jouait dans le terminal, nous voulions récupérer les caractères saisis au clavier afin de savoir sur quelle touche le joueur appuyait. Les fonctions **scanf** et **getchar** étant bloquantes elles cassaient le dynamisme du jeu. L'affichage étant aussi limité nous nous sommes intéressés à la librairie ncurses. La fonction **getch** de cette librairie permettait de récupérer les entrées du clavier de manière non bloquante et donc plus réactive, comme nous le voulions.

Afin de "stabiliser" l'affichage dans le terminal nous avions décider de limiter le jeu à 20 boucles par seconde. En effet si l'on ne limitait pas, l'affichage était illisible. Seulement cette limite a posé un problème dans la récupération des touches. Lorsque l'on envoyait beaucoup de caractère différents au programme (en maintenant une touche enfoncée par exemple) ces derniers s'accumulaient puisqu'ils ne pouvaient êtres lus qu'à la vitesse de 20 par seconde. Certains étaient donc interprété même après que l'utilisateur ait relâché la touche. Ce mécanisme posait un vrai problème de jouabilitée puisqu'il cassait la réactivité du jeu. Afin de régler ce problème nous avons ajouté un **flushinp** qui a pour effet de vider le buffer. Ainsi notre fonction récupérait toujours la dernière touche appuyée par l'utilisateur ou bien aucune le cas échéant.

Voici à quoi ressemblent les contrôles de notre jeu :
- Z : se déplacer vers le haut
- Q : se déplacer vers la gauche
- S : se déplacer vers le bas
- D : se déplacer vers la droite
- ESPACE : tirer un projectile

Dans la fonction **controle_joueur** on commence donc par récupérer la direction choisie. On met dans la strucure joueur le choix qui a été fait en changeant les variables x_direction, y_direction, x_mouv et y_mouv.

Si c'est un tir, le projectile est tiré immédiatement dans la dernière direction connue c'est-à-dire x_direction et y_direction puisque dans ce cas là x_mouv et y_mouv sont mis à 0 pour que le joueur ne se déplace pas.

Si aucune touche n'est appuyée le joueur ne bougera pas et ne fera rien.

Ensuite on fait appel à la fonction **deplacer_joueur** à qui on passe les salles et le joueur. Cette fonction va commencer par vérifier le cas de victoire du joueur. Elle vérifie donc si le joueur est bien en possession de la clé, s'il a tué tous les monstres et qu'il veut se déplacer sur la porte de sortie. Le cas échéant, la fonction vérifie que la case sur laquelle veut se déplacer le joueurs n'est pas occupée par un projectile, un monstre, un obstacle ou un mur.
Dans le cas où le joueur veut se déplacer sur une porte la fonction va gérer le changement de salle en le plaçant de manière cohérente dans la salle dans laquelle il voulait aller.

3.4 Gestion des monstres

3.4.1 Création des monstres

Dans un rogue-like, les difficultés que peut rencontrer le joueur lors de son expérience dépendent généralement de ses combats avec des monstres qui ont pour but est d'empêcher le joueur d'atteindre ses objectifs.
Les comportements des monstres peuvent se manifester de différentes manières :

- blocage
- tir sur le joueur
- etc.

Tout d'abord, nous avons défini un type **t_monstre** dont la structure est constitué des coordonnées du monstre dans la salle, des coordonnées utiles à son déplacement, sa dernière direction (verticale ou horizontal) et enfin, sa vitesse.
L'ajout des monstres se fait dans la fonction d'initialisation de la salle. Ils sont placés de manière aléatoire dans la matrice lorsque l'emplacement est une case disponible (différentes d'un mur ou d'une porte). On remplis également une file qui contient tout les monstres présent dans la map. Lorsqu'un monstre sera tué, une entité de la file sera alors retiré.

3.4.2 Déplacement des monstres

Rentrons maintenant dans le coeur du mécanisme gérant le comportement des monstres.
Dans notre projet, nous avons simplement voulus que les monstres se déplacent en direction du joueur en empruntant le chemin le plus court dans la matrice. Nous souhaitions également que les monstres aient la capacité d'esquiver les projectiles du joueur, afin de rajouter une légère difficulté.
Nous avons écrit une fonction permettant de créer une matrice, qui contiendra la distance entre le joueur et chaque case de la matrice.

Dans une première boucle, nous initialisons cette matrice. La case correspondant au joueur contiendra le chiffre 1, chaque case disponible contiendra 0, puis tout les obstacles, murs, portes, projectiles, et portails de sortie, se verront attribuer le chiffre -1 (CF. *Figure 7 : Une salle, affichage Ncurces* et *Figure 8 : Matrice distance correspondante à la salle de la Figure 7*)
Dans une seconde boucle, tant que la matrice n'est pas entièrement remplie, on commence par se positionner sur la case du joueur. Si, une case adjacente est un 0, alors on l'incrémente par rapport à n, le nombre de tour de la boucle. Nous recommençons ensuite cette boucle, qui pour chaque tour, incrémente elle aussi n.
On obtient une matrice qui, à chaque déplacement du joueur, se remplis de la sorte, indiquant aux monstres le chemin à emprunter pour rejoindre le joueur.

Nous avons également rajouté une condition, précisant que si la dernière direction du monstre était verticale, le prochain mouvement sera horizontale (et inversement). Cela permet d'éviter aux monstres de se déplacer uniquement par grandes lignes droites. Le comportement des monstres se fait donc de manière plus naturel.

3.5 Gestion des projectiles

Nous avons défini les projectiles comme étant des objets qui se déplacent de case en case sur une trajectoire linéaire et à une vitesse fixe. En partant de cette définition nous avons créé une structure projectile comprenant des coordonnées sur la matrice salle, une direction en X, une direction en Y et une vitesse.
Enfin, pour pouvoir avoir plusieurs projectiles dans une salle , nous avons ajouté une liste de structures projectile à la structure salle. Ainsi, pour chaque salle donné nous avons accès à la liste de tous ses projectiles.

3.5.1 Création des projectiles

Pour créer un projectile, il nous faut la salle dans laquelle il sera créé, les coordonnées dans cette salle , une direction en X et en Y ainsi qu'une vitesse. Toutes ces informations sont rentrées en paramètres dans la fonction **tirer_projectile**(salle, x, y, directionX, directionY, vitesse). C'est dans cette fonction que les champs de la structure projectile sont initialisés et que le projectile est ajouté à la liste. Dans la cas d'un projectile tiré par le joueur, on veut qu'il soit tirer dans la direction vers laquelle le joueur regarde. Pour cela nous avons besoin des coordonnées du joueur (que nous avons) ansi que sa direction (que nous n'avons pas). Nous avons donc ajouté les champs direction_X et direction_Y à la structure joueur.

Ainsi à la création d'un projectile par le joueur, on donne la salle dans laquelle est le joueur, les coordonnées du joueur dans la salle, sa direction en X et en Y ainsi que vitesse quelconque. Ce qui donne **tirerprojectile**(map[joueur-¿x_map][joueur-¿y_map], joueur-¿x_salle,joueur-¿y_salle, joueyr-¿direction_x, joueur-¿direction_y).

Lors de l'appelle de la fonction **tirerprojectile** le projectile est crée sur la case devant le joueur, pour cela on initialise les coordonnées du projectile aux coordonnées du joueur + direction du joueur. Cela nous oblige à vérifier que la case sur laquelle le projectile doit être créé est vide au risque de détruire des élément de la salle. Dans le cas où la case n'est pas vide, le projectile n'est pas créé et si cette case est occupée par un monstre, le monstre est supprimé.

3.5.2 Déplacement des projectiles

Une fois créés, les projectiles se déplacent dans la salle jusqu'à ce qu'ils rencontrent un obstacle. Les projectiles sont gérés par la fonction **deplacer_projectile** qui prend en paramètres : - La salle courante, ainsi nous avons accès à la listes de projectiles et des monstres
- Le joueur
- Un compteur i qui est incrémenté de 1 a chaque appelle de la fonction

Chaque salle pouvant contenir plusieurs projectiles, on parcours l'ensemble de la liste de projectiles dans la fonction **deplacer_projectile**. Ainsi on peut les traiter un à un.
Avant d'en déplacer un, on vérifie s'il doit l'être. Cela dépend de sa vitesse, si il a une vitesse de 1, il se déplace à chaque appel de **deplacer_projectile**, si il a une vitesse de 2 il se déplace une fois sur 2, si il a une vitesse de 3 il se déplace une fois sur 3 etc. Pour savoir si on peut le déplacer, on utilise le compteur i, si i modulo vitesse du projectile est égale a zéro alors on déplace le projectile.

Pour déplacer le projectile on additionne simplement ses coordonnées avec sa direction, ainsi on obtient ses nouvelles coordonnées. Mais avant d'assigner la case de ses nouvelles coordonnées à un projectile, il faut vérifier qu'elle soit vide. Si la case est vide, on y place le projectile, sinon il y a plusieurs cas possibles :
- La case est occupée par un obstacle, alors on supprime le projectile
- La case est occupée par le joueur, alors on supprime le projectile et on retire un point de vie au joueur
- La case est occupée par un monstre, alors on supprime le projectile et on parcourt la liste des monstres pour trouver le monstre en question et le supprimer.

3.6 Adaptation SDL

Une fois le jeu jouable dans le terminal, il nous a paru important de le rendre esthétique et agréable à jouer. Pour ce faire, l'utilisation de la librairie SDL semble appropriée. Seulement, n'étant pas familier avec ce nouvel outil, le premier objectif n'a pas été d'avoir un beau rendu, mais un rendu fonctionnel ouvrant ensuite la voie à un jeu potentiellement magnifique. L'affichage en SDL n'est au final que la pose de texture sur un rendu final.

3.6.1 Création du menu

Notre menu est composé de plusieurs images de fond ainsi que d'un titre et de trois boutons.
Le premier bouton lance le jeu, le second ouvre une nouvelle fenêtre sur les règles et le dernier ferme le jeu. Voulant un jeu se jouant exclusivement au clavier, le parcours du menu se fait de la même manière avec les flèches.
Lors du parcours du menu, une variable de position comprise entre un et trois permet de savoir quelle action sera effectuée lors de la validation lorsque la touche Entrée est appuyée. Cette variable de position (pos) est incrémentée ou décrémentée si l'on appuie sur la flèche du haut ou du bas et revient à un si elle dépasse trois et passe à trois si elle devient inférieur à 1.
Chaque bouton étant associé à une valeur de pos, il est alors possible de changer l'image du bouton (la même en plus claire) afin de préciser quelle sélection est faite lors de l'appuie sur Entrée. Par défaut le bouton sélectionné est le premier. La fenêtre concernant les règles utilise le même parcours.

Figure 4: Changement de sélection du bouton dans le menu

3.6.2 Affichage du jeu

Afin de passer à un affichage en SDL pour la salle, il a fallu rajouter un rendu dans les paramètres de la fonction afficher_salle() et changer le type de valeur renvoyer par cette fonction qui est devenu un rendu également(SDL_Renderer). L'affichage du jeu est en 16/9 environ.
Passé en SDL insinue de passer d'une matrice avec des coordonnées à une fenêtre composée de pixels. Il faut donc définir la taille des images qui remplacent les cases de la matrice en fonction de ce qu'elles contiennent. Ainsi plusieurs images ont été sélectionnées pour représenter le contenu de la matrice, une image pour les portes, une autre pour les projectiles, une autre pour les monstres, etc.
Afin d'obtenir les textures de ces images, deux fonctions sont utilisées.
La première tex_img_png(), retourne une texture à partir d'une surface qui est elle générée à partir d'une image dont le chemin est renseigné dans les paramètres de la fonction. La seconde init_res(), initialise une liste de texture en utilisant la première fonction, dans le but de les afficher sur le rendu final.
L'affichage des salles est donc au final l'affichage d'images les unes à la suite des autres en fonction du contenu présent dans la matrice de la salle.
L'affichage de la mini_map à droite de la salle se fait de la manière, un parcours de matrice et on affiche telle ou telle image selon son contenu.
L'affichage des coeurs et de la clé se fait en fonction de la structure joueur.

Figure 5: Affichage d'une salle en SDL

3.6.3 Récupération des touches

Le passage de l'affichage en mode console à celui en mode SDL à également changé la méthode de récupération des touches. Dans le jeu en mode console, la récupération des touches se faisait via la fonction getch. Dans le jeu avec la SDL, la récupération des touches se fait grâce au paramètre event.key.keysym.sym d'un SDL_Event event. Ce paramètre correspond à une touche appuyée et est un entier. Cet entier est passé en paramètre de la fonction controle_joueur() comme étant un SDL_Keycode.
Le même traitement est effectué dans le reste de la fonction controle_joueur() à l'exception que la fonction flushinp n'est plus utilisée et que les flèches remplacent z,q,s,d.

4 Résultat et conclusion

Nous sommes globalement satisfait de notre travail tout au long de ce projet, en effet nous avons réussis à intégrer toutes les fonctionnalités de base que nous avions définis dans le cahier des charges.
Nous avons également réussi à intégrer des fonctionnalités bonus tels que la mini map, l'interface graphique grâce à la SDL2, l'ajout de vie qui apparaissent lorsque l'on tue les monstres et la clé qu'il faut ramasser pour sortir.

Le planning défini en début de projet a été respecté jusqu'au bout. La répartition des tâches ayant été claire en début de projet cela nous a permis de nous fixer des objectifs précis et atteignable dans les temps impartis. Si un collègue bloquait sur sa partie, la plupart du temps à cause d'une erreur dans la réflexion ou un oublie, alors le reste du groupe était là pour l'aider. Cette manière de travailler a aidé au bon déroulement du projet.

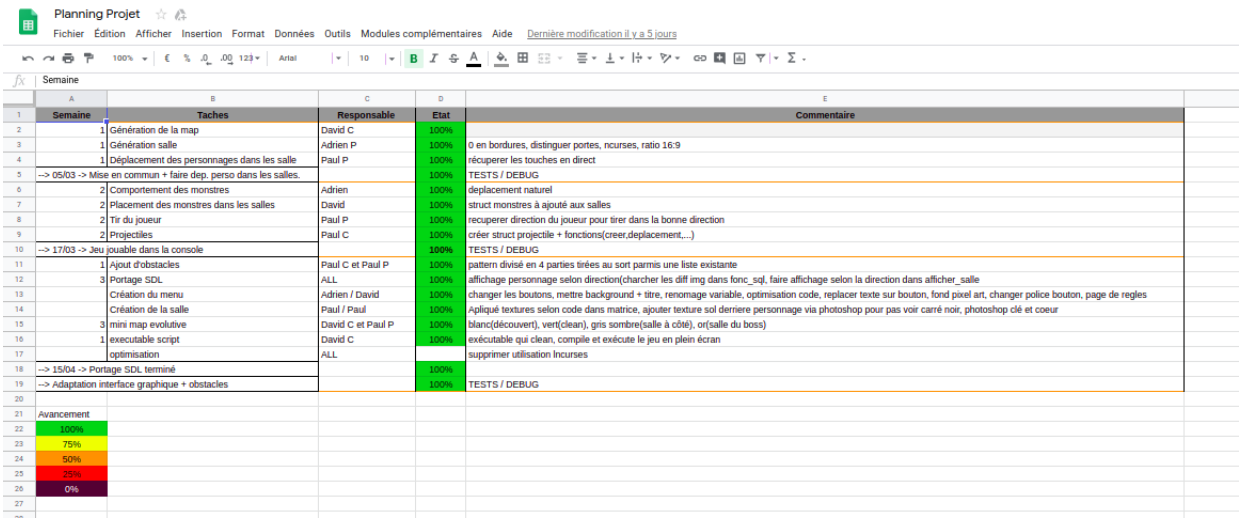
Même si l'ensemble de notre cahier des charges (et même plus) a été effectué, notre projet est encore perfectible.
Certains détails lui font effectivement défauts, comme par exemple le déplacement des personnages qui n'est pas fluide. Beaucoup d'améliorations sont encore possibles:

-l'ajout d'une bande sonore
-l'ajout de boss
-l'ajout de texte et de doublage
-une version en coop ou en versus en réseau

Mener un projet comme celui-ci était quelque chose de nouveau pour chacun d'entre nous, nous avons donc tous beaucoup appris de ces trois mois de travail en groupe.
Tout d'abord d'un point de vu technique avec l'utilisation de nouveaux outils comme Doxygen, Ncurces, la SDL et enfin l'utilisation plus poussé d'outils que nous connaissions déjà, tout particulièrement les makefiles, primordiaux dans ce genre de projet. Nous avons aussi beaucoup appris sur l'organisation, se fixer des objectifs réalistes, penser un programme en plusieurs parties en autonomie.
La réalisation de ce projet nous a également permis de réaliser qu'il est parfois difficile d'avancer au même rythme que les autres.

Si nous avions à refaire un projet du même type à l'avenir, nous serions beaucoup mieux préparés et efficaces qu'il y à trois mois.

5 Annexes



Planning Projet

Fichier Édition Afficher Insertion Format Données Outils Modules complémentaires Aide Dernière modification il y a 5 jours

Semaine

	A	B	C	D	E
1	**Semaine**	**Taches**	**Responsable**	**Etat**	**Commentaire**
2	1	Génération de la map	David C	100%	
3	1	Génération salle	Adrien P	100%	0 en bordures, distinguer portes, ncurses, ratio 16:9
4	1	Déplacement des personnages dans les salle	Paul P	100%	récuperer les touches en direct
5	--> 05/03 -> Mise en commun + faire dep. perso dans les salles.			100%	TESTS / DEBUG
6	2	Comportement des monstres	Adrien	100%	deplacement naturel
7	2	Placement des monstres dans les salles	David	100%	struct monstres à ajouté aux salles
8	2	Tir du joueur	Paul P	100%	recuperer direction du joueur pour tirer dans la bonne direction
9	2	Projectiles	Paul C	100%	créer struct projectile + fonctions(creer,deplacement,...)
10	--> 17/03 -> Jeu	jouable dans la console		100%	TESTS / DEBUG
11	1	Ajout d'obstacles	Paul C et Paul P	100%	pattern divisé en 4 parties tirées au sort parmis une liste existante
12	3	Portage SDL	ALL	100%	affichage personnage selon direction(charcher les diff img dans fonc_sql, faire affichage selon la direction dans afficher_salle
13		Création du menu	Adrien / David	100%	changer les boutons, mettre background + titre, renomage variable, optimisation code, replacer texte sur bouton, fond pixel art, changer police bouton, page de regles
14		Création de la salle	Paul / Paul	100%	Apliqué textures selon code dans matrice, ajouter texture sol derriere personnage via photoshop pour pas voir carré noir, photoshop clé et coeur
15	3	mini map evolutive	David C et Paul P	100%	blanc(découvert), vert(clean), gris sombre(salle à côté), or(salle du boss)
16	1	executable script	David C	100%	exécutable qui clean, compile et exécute le jeu en plein écran
17		optimisation	ALL		supprimer utilisation lncurses
18	--> 15/04 -> Portage SDL terminé			100%	
19	--> Adaptation interface graphique + obstacles			100%	TESTS / DEBUG
20					
21	Avancement				
22	100%				
23	75%				
24	50%				
25	25%				
26	0%				
27					

Figure 6: Planning du projet

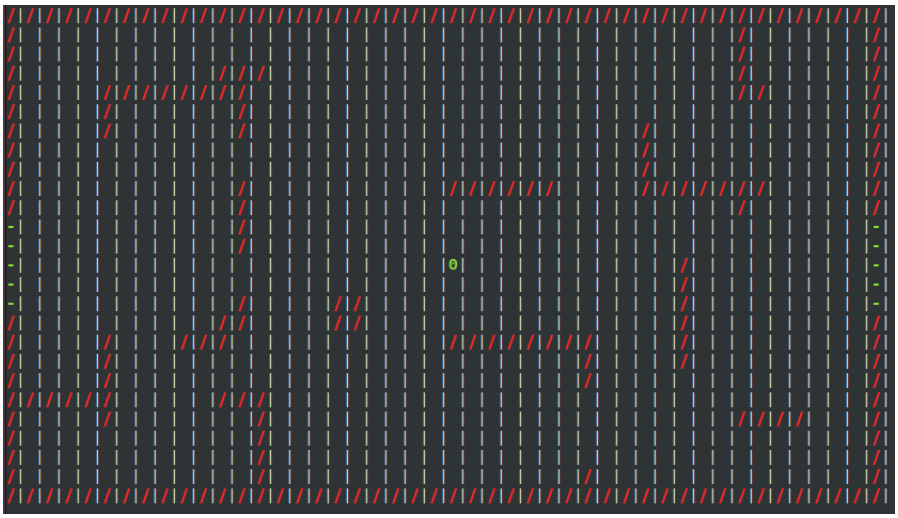


Figure 7: Une salle, affichage Ncurces

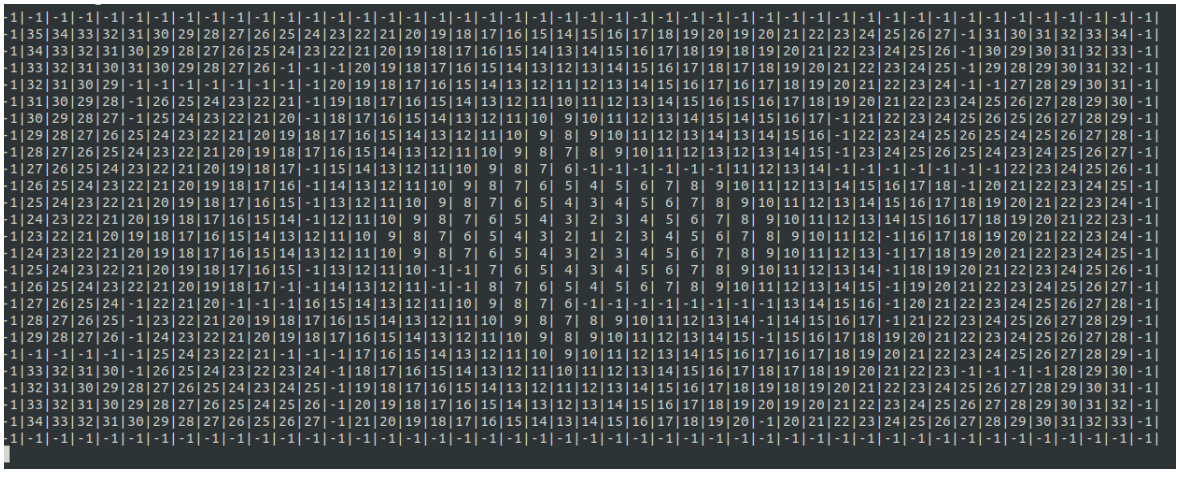


Figure 8: Matrice distance correspondante à la salle de la figure 5