

RAPPORT DE PROJET

PLAN :

- 1) Le minimum demandé
- 2) La construction de Niveaux aléatoires
- 3) La gestion des monstres.
- 4) La gestion de l'écran
- 5) Les options de réglage
- 6) La bibliothèque "myrandom"

1) Le minimum demandé.

Nous avons scrupuleusement suivi l'énoncé, sans négliger la documentation ce qui fut notre erreur lors du premier projet.

Le jeu :

Le joueur commence la partie en haut à gauche, et la sortie se trouve toujours en bas à droite. Il se déplace avec les flèches ou avec le pavé Numérique selon sa préférence.

Pour afficher la liste des clés pendant le jeu, vous devez taper 'l' . Et pour sortir 'x'.

La programmation du jeu :

Elle fut en fait assez courte, la librairie liste que nous avons programmée lors du premier projet nous gérait déjà quasiment nos clés. Nous avons tout de même réécrit la fonction d'affichage étant donné la nature de la sortie ncurses, l'utilisation d'un printList aurait provoqué un bug.

Pour pouvoir ajouter des monstres, nous avons utilisé l'option `nodelay(stdscr, TRUE)` ainsi un appel à `getch()` ne bloque pas le jeu et les monstres peuvent continuer à tourner.

La gestion des événements claviers sont gérés par le fichier `player.c` .

2) La construction de niveaux aléatoires.

Ce fut la partie la plus longue du projet. Sur internet nous avons

décortiqué les algorithmes de construction de labyrinthe aléatoire. Celui qui a retenu notre attention est l'algorithme de Trojan. Malheureusement, cet algorithme a pu nous aiguiller quand aux choix de programmation mais n'était pas utilisable tel quel, à cause des portes et des clés qu'il fallait aussi gérer.

La construction en bref :

Tout d'abord est créé un labyrinthe complètement fermé, c'est à dire des cellules vides entourées chacune de 4 murs.

Tout l'algorithme repose sur le fait que l'on casse des murs. La question qu'il faut se poser lorsqu'on casse un mur est de savoir si il existe déjà un chemin entre deux cellules. Si oui, alors on ne casse pas le mur, si non, alors on casse le mur (avec une petite précision que l'on verra plus tard).

L'implémentation traditionnelle de l'algorithme de Trojan consiste à attribuer un représentant pour chaque case du labyrinthe. Si deux cellules ont le même représentant alors on ne fait rien et si elles ont des représentants différents alors on casse le mur et on leur attribue un représentant en commun. Toute fois ce système de représentant est inutilisable dans notre cas, car ce que nous voulons ce n'est pas un simple labyrinthe mais un labyrinthe représenté par des zones qui correspondent à des composantes connexes. Une fois les zones établies nous pourrons placer nos clés et nos porte.

Voilà donc comment se déroule l'algorithme.

Une fois le labyrinthe (fermé) construit, on attribue la valeur -1 à la première cellule en haut à gauche, et la valeur -2 à la dernière cellule en bas à droite. Ainsi l'entrée sera toujours en haut à gauche et la sortie toujours en bas à droite. Puis on attribue à des cellules prises de manière aléatoire des valeurs négatives jusqu'à ce qu'il y est autant de zones que de clés.

Une fois cette première étape achevée, on doit casser les murs, pour se faire on commence par générer un tableau de coordonnées qui correspond à tous les murs possible.

Maintenant on va analyser chaque mur dans un ordre aléatoire.

Il y a plusieurs type de zone :

- zone zéro, je suis une cellule personne ne s'est occupé de moi
- zone positive, j'appartiens à une zone temporaire
- zone négative, j'appartiens à une zone finale.

Voici l'ordre des priorités de la plus prioritaire à la moins prioritaire :
zone négative, zone positive , zone zéro

Donc lorsque je casse un mur, si j'ai une cellule appartenant à une zone plus prioritaire que l'autre je transforme l'autre zone (de l'autre cellule).

Si j'ai deux cellules à priorité égale :

cas 0 0 : je crée une nouvelle zone temporaire qui englobe les deux cellules

cas positif positif : si c'est la même zone je ne fait rien

sinon je casse le mur et transforme le tout

comme une même zone

cas négatif négatif : je ne fais rien.

Maintenant si tout c'est bien passé j'ai autant de composantes connexes que de zones négatives que j'avais attribué au départ. et mon tableau est donc composé uniquement de zéro et de nombre négatifs.

Il s'agit maintenant de créer les portes entre les différentes zones et de leur attribuer pour le moment une valeur neutre (arbitrairement - nombreDeClé-1)

En même temps que je crée les porte j'écris dans un tableau à double entrée les coordonnées de celles-ci. Ainsi dans ce tableau (doorsTab) si à la case i,j il y a une valeur non nulle cette valeur correspond à la coordonnée de la porte entre les zone i et j ,et si il y a un zéro alors il n'y a pas de porte.

*Note importante : Un labyrinthe est plus jolie lorsqu'il est plus large que long, et plutôt que de faire un tableau de tableau de tableau ce qui n'aurait pas été pratique à implémenter, les coordonnées sont donc calculées de la manière suivante yCell + xCell*cols. De toute façon les macros GETL et GETC rendent transparentes cette astuce.*

Nous avons donc maintenant en notre possession un tableau qui donne les portes existantes entre les différentes zones.

Il s'agit de crée un chemin vers la sortie en attribuant des portes de couleurs et en déposant les bonnes clés dans les bonnes zones.

C'est la fonction findKeyPath qui fait ce travail, on lui passe en paramètre une longueur de chemin. Je m'explique, il peut paraître curieux d'imposer une longueur de chemin vers la sortie, mais en fait si les zones -1 et -2 sont connexes je me retrouverai toujours avec une seule porte et ce ne serait pas intéressant pour le joueur. Donc findKeyPath essaye de trouver un chemin de longueur N, si il y parvient à la remonté des appels récursifs il place les clés et les portes.

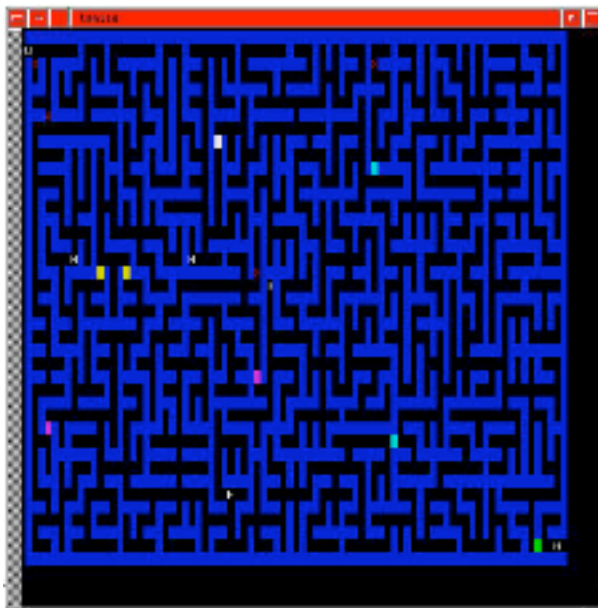
Note : findKeyPath place aussi de fausse portes (de la même couleur qu'une clé en la possession du joueur) pour rendre le jeu plus difficile.

Nous savons qu'un chemin existe la connexité étant garantie par la méthode de construction du labyrinthe. Toutefois rien ne garantit l'existence d'un chemin de longueur imposé. Donc, pour qu'un maximum de portes soient placées, on teste toutes les longueurs possibles en partant de la plus longue vers la plus courtes.

Finalement nous avons un jeu avec la garantie qu'il existe un chemin vers la sortie.

Il a été long et difficile de concevoir un algorithme qui générerait des plateaux agréables a jouer. Il est clair que nous utilisons beaucoup de fonctions récursives avec beaucoup de closes (parce qu'on travaille sur un graphe). Plutôt que de détailler le code de chaque fonction (qui même après explication peut paraître flou), nous avons préféré vous faire ce petit résumé qui devrait vous permettre de mieux saisir la logique du fonctionnement de l'algorithme.

Un exemple de labyrinthe :



3) La gestion des monstres.

La première chose est qu'il faut passer en mode nodelay, pour qu'un appel à getch ne bloque pas le programme.

Le module des monstres a été écrit dans l'optique de futures extensions.

Chaque monstre est représenté par un numéro.

Grâce à ce numéro on peut retrouver : la dernière direction que suivait le monstre et le comportement d'un monstre (c'est à dire le numéro de la fonction qui gère ses déplacement).

Il pourrait paraître plus logique de faire une table de pointeur pour les fonctions de comportement, en fait non, le comportement d'un monstre étant choisis de manière aléatoire travailler directement avec des pointeurs n'aurait pas été pratique.

Pour que les monstres ne soit pas trop rapides à chaque appel on note la date (en microsecondes) du dernier appel. Si le nouvel appel est trop récent on ne fait rien, sinon on rafraîchit la position des monstres.

Choisir une vitesse plus grande aurait permis une plus grande fluidité du jeu, mais nuisait grandement à la jouabilité car il était très difficile d'éviter un monstre. Nous avons donc sacrifié l'aspect graphique au profit du plaisir de jouer.

Enfin, si le joueur touche un monstre il perd.

4) La gestion de l'écran

Cette partie a elle aussi été longue et compliqué à implémenter. Non pas à cause de la complexité du code, mais à cause des fonctions de la librairie ncurses dont le comportement n'est pas toujours clair.

Ne vous fiez donc pas à l'aspect rudimentaire et simplifier du code, car trouver comment gérer l'écran de manière correcte nous a forcé à nous plonger dans la documentation ncurses.

Suite à de nombreux "bricolages" tout marche enfin.

La gestion du redimensionnement :

Le handle qui gère normalement les le signal SIGWINCH n'est pas du tout adapté dans notre cas. En effet, il renvoie le caractère KEY_RESIZE sur l'entrée standard et surtout il bloque le processus courant jusqu'à ce qu'un touche soit pressée (même avec l'option nodelay !!!).

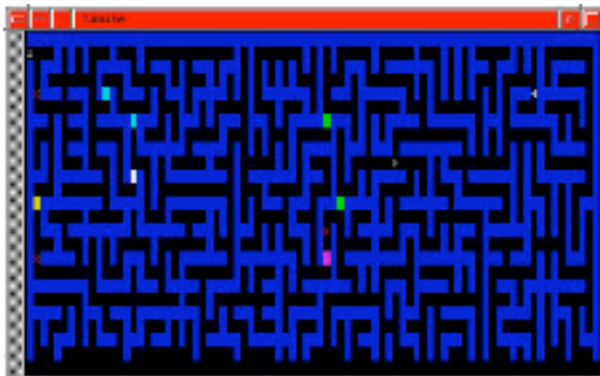
Notre première idée a donc été de redémarrer complètement le mode ncurses. La solution marche mais est lente et provoque parfois des bugs, si l'utilisateur fait des redimensionnement intempestifs. C'est donc la que les choses se compliquent, nous avons trouvé dans le manuel de ncurses ce que fait initscr de manière détaillé. Elle appelle des fonctions bas niveau qui en théorie ne sont pas visible par l'utilisateur. Or ce dont nous nous avons besoin c'est une simple mis à jour des variables d'environnement LINES et COLS. Nous appelons donc setupterm pour y parvenir.

La gestion des écrans du jeu :

Voici la philosophie du jeu , lorsque vous lancez le jeu nous supposons que la taille de votre terminal est agréable pour vous puisque vous avez travaillé avec cette dimension. La dimension de départ de votre terminal sera donc la plus grande à la quelle vous aurez le droit tout au long du jeu. Je m'explique : imaginons que vous lanciez le jeu, si vous redimensionnez vers une taille plus petite et que vous essayez de retrouver votre taille normale, pas de problème. Par contre si vous essayez d'agrandir la taille initiale vous ne verrez pas le reste du labyrinthe. Ces réglages ont été fait dans un but précis, éviter les tricheries. Le but du joueur est de trouver la sortie, tant qu'il ne la voit pas il pourrait être tentant de regarder ou elle se trouve en redimensionnant la fenêtre et donc cela lui est interdit.

La dernière ligne du terminal est réservé aux messages à l'utilisateur, soit lorsqu'il trouve une clé, soit lorsqu'il veut faire l'inventaire des clés etc...

Un exemple de labyrinthe plus grand que l'écran.



5) Les options de départ :

Nous avons triché pour cette partie, nous avons récupéré l'exemple du manuel forms que nous avons modifié pour qu'il aie le comportement voulu.

Ce que nous avons modifié :

Une meilleure gestion des événements clavier pour que l'utilisateur puisse revenir se déplacer et corriger une entrée.

Une simplification de l'affichage pour être sûr que le formulaire s'affiche dans n'importe quel terminal (car on avait eu des problèmes avec les terminaux mac)

6) La bibliothèque myrandom

Les fonctions rand <==> random génère des nombres pseudos aléatoires à partir d'une graine. Malheureusement si vous lancez plusieurs fois votre programme vous risquez de vous retrouver avec la même graine. Nous avons donc décidé de créer notre propre librairie qui se sert de la date actuelle en microsecondes pour initialiser la graine de random.