

UFR SCIENCES ET TECHNIQUES  
UNIVERSITÉ DU MAINE

## RAPPORT DE PROJET

---

# RogueLike

---

17 décembre 2016

Emeric MOTTIER  
Valentin PELLOIN  
Titouan TEYSSIER

L2 Sciences pour l'ingénieur

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Organisation</b>	<b>3</b>
2.1	Répartition des tâches . . . . .	3
2.1.1	Pourquoi cette répartition . . . . .	3
2.2	Utilisation d'un gestionnaire de versions . . . . .	3
2.3	Utilisation des projets Github . . . . .	3
2.4	Notre boîte à outils . . . . .	3
2.5	Doxygen, CUnit, GDB . . . . .	4
<b>3</b>	<b>Analyse et Conception</b>	<b>5</b>
3.1	Notre cahier des charges . . . . .	5
3.2	Comment jouer ? . . . . .	6
<b>4</b>	<b>Codage, méthode et outil</b>	<b>8</b>
4.1	Structures et énumérations . . . . .	8
4.1.1	Une case de la carte . . . . .	8
4.1.2	Les être vivants . . . . .	8
4.1.3	Et le reste . . . . .	9
4.2	Séparation du code en modules . . . . .	9
4.3	Détail des modules . . . . .	9
4.3.1	La génération des niveaux . . . . .	9
4.3.2	Les lectures et les écritures dans des fichiers . . . . .	10
4.3.3	Les interactions et déplacements . . . . .	10
4.3.4	L'affichage . . . . .	11
4.3.5	Les monstres . . . . .	11
4.3.6	La nourriture et la vie . . . . .	11
4.3.7	L'inventaire du joueur . . . . .	12
<b>5</b>	<b>Résultat et conclusion</b>	<b>13</b>
5.1	Améliorations possibles . . . . .	13
5.2	Apport personnel du projet . . . . .	13
5.3	Cas d'utilisation du débogueur <i>GDB</i> . . . . .	14
5.3.1	Des monstres qui passent à travers les portes ... . . . .	14

# Chapitre 1

## Introduction

Nous avons choisi le jeu *Roguelike* car c'est un jeu que nous trouvons intéressant, puisqu'il est complet, et que c'est un jeu aux possibilités infinies : il est toujours possible d'ajouter de nouvelles actions que le joueur pourra effectuer.

Notre jeu se déroule dans le bâtiment IC<sub>2</sub>. Nous sommes un étudiant, nous partons du rez-de-chaussée, et nous devons aller chercher QUELQUE CHOSE tout en haut, pour le ramener.

Nous devons cependant faire attention aux monstres : des L1, L2, L3, des masters, des doctorants, et certains fantômes : CLAUDE et CHAPPE.

Sur notre chemin, nous pouvons trouver quelques pièges : des flaques d'eau laissées par les femmes de ménages qui nous font glisser, des trous entre les étages qui nous font tomber d'un étage à un autre inférieur, ou des cartes à jouer qui nous sont jetées dessus par des L1.

Durant notre parcours, nous devons aussi tenir compte de notre faim. Nous possédons une barre de vie, lorsqu'elle est à zéro, nous mourons. Pour régénérer de la vie, il y a deux possibilités : ne plus avoir faim (en mangeant de la nourriture, attention, certaines sont empoisonnées), et des seringues de soin à s'injecter directement. Ces objets peuvent être consommés directement sur place quand le joueur le trouve, ou plus tard, en les gardant dans son inventaire.

Enfin, lorsque le joueur apparaît, il ne voit pas entièrement la carte, il doit la découvrir pour cela. Lorsque le joueur a trop faim, en plus de perdre de la vie, il s'évanouit : il se déplace plus difficilement, et perd connaissance de ce qu'il a découvert.

Pour le projet, nous devons au minimum effectuer un jeu qui génère des niveaux (ici, des étages dans notre bâtiment) aléatoires, avec une taille variant en fonction de l'étage où se trouve le joueur. Il était aussi demandé, en fonction de l'avancement du projet, d'ajouter des fonctionnalités supplémentaires : des armes, des monstres, des pièges, ou autres.

## Chapitre 2

# Organisation

ici mettre un petit mot <tout le monde>

### 2.1 Répartition des tâches

Mettre ici notre répartition des tâches. <tout le monde>

#### 2.1.1 Pourquoi cette répartition

<tout le monde>

### 2.2 Utilisation d'un gestionnaire de versions

Pour gérer les versions du projet nous avons utilisé Git et Github. Git pour toute la partie locale à chacune de nos machines. Quand nous codons, nous pouvons ainsi faire des versions du projet régulièrement et revenir en arrière si besoin. Github pour la mise en commun des modifications apportées, ce qui nous permet de travailler ensemble sans nécessairement coder en même temps ou au même endroit. Afin que chacun puisse librement ajouter ses modifications au projet, un seul dépôt Github a été créé et chaque membre de l'équipe a reçu le droit en écriture sur le dépôt.

### 2.3 Utilisation des projets Github

L'utilisation des projets de Github, découvert en conduite de projet, nous a permis de mettre tous nos objectifs sur le projet aussi bien que le code et le rapport. Nous avons pu nous répartir des tâches fait par chacun et ce qui nous reste à faire. Pour les objectifs du code, le lien du projet Github est le suivant : <https://github.com/TitouanT/rogueLike/projects/1>. Et pour les objectifs du rapport : <https://github.com/TitouanT/rogueLike/projects/2>.

### 2.4 Notre boîte à outils

Notre projet a été réalisé en plusieurs modules différents, et l'un d'entre eux est notre boîte à outils. Dedans se trouve de nombreuses fonctions qui nous sont utiles, mais qui ne sont pas pour autant liées à notre projet en particulier : des fonctions de comparaison d'intervalles, d'aléatoires, de caractères, de log

d'erreurs, de fichiers, ...

Nous avons aussi les fonctions essentielles pour l'accès à des listes et des files.

## 2.5 Doxygen, CUnit, GDB

Durant la réalisation de notre projet, nous avons utilisé divers outils d'aide à la programmation et au débogage.

La première chose que nous avons mis en place est la documentation à l'aide de *Doxygen*. C'est un programme qui génère une documentation automatiquement, en fonction des fichiers d'en-têtes et sources. La documentation peut être générée de plusieurs formats, nous avons choisi au format HTML car il est plus facile de s'en servir. Celle-ci est sur internet, à l'adresse suivante : <https://roguelike.vlntn.pw/>. Elle se met à jour automatiquement en fonction de notre code (via un webhook mis en place sur Github).

Ensuite, nous avons utilisé *CUnit*, un framework de tests unitaires pour le C. Toutes nos fonctions de notre boîte à outil ont été testées, avec des assertions que nous jugeons pertinentes (sur des valeurs qui pourraient poser problème dans certaines fonctions, comme des valeurs nulles, négatives, sur des fichiers inexistantes, ...).

Enfin, lorsque nous avons certains bogues que nous n'arrivions pas à résoudre, nous avons utilisé le logiciel de débogage *GDB* (*GNU DeBugger*). Nous n'avons pas réussi à le faire fonctionner dès le début, car nous utilisons la librairie d'affichage *ncurses*<sup>1</sup>, qui utilise déjà le terminal pour afficher notre jeu. En le combinant avec *GDB*, le terminal n'était plus utilisable.

La solution a été d'utiliser deux téléscripteurs (TTY) différents : un pour le jeu, et un pour le débogueur. Dans l'annexe, vous pouvez retrouver 3 exemples de cas où nous nous sommes servis du débogueur.

---

1. Voir la section interactions et déplacements, page 10.

## Chapitre 3

# Analyse et Conception

ici mettre un petit mot

### 3.1 Notre cahier des charges

Avant de commencer à programmer notre jeu, nous nous sommes concertés autour d'un même fichier pour mettre en commun nos idées à propos du jeu. Notre jeu étant très libre dans son fonctionnement, c'était une étape cruciale.

Notre *rogueLike* prend place dans le bâtiment IC<sup>2</sup>. Le joueur démarre le jeu au premier étage. Il doit se rendre tout en haut, aller chercher l'objet, puis revenir en bas.

Chaque étage de ce bâtiment est généré aléatoirement, il doit comporter des pièces et des couloirs. Entre ceux-ci, peut se trouver des portes, ou juste un trou. Les portes sont aléatoirement ouvertes ou fermées, et le joueur peut essayer de les ouvrir manuellement (il a une probabilité de ne pas réussir). Le nombre de pièces par étage est défini en fonction de l'étage : les étages supérieurs sont plus compliqués car ils possèdent plus de pièces. Pour changer d'étages, le joueur doit emprunter des escaliers (un escalier pour monter, et un pour descendre par étage, ceux-ci ne peuvent pas se trouver dans la même pièce).

Lorsque le joueur apparaît, il ne voit que la pièces où il se situe. Dans les couloirs, lorsqu'il progresse, les zones s'éclairent petit à petit, en revanche, dans dès qu'il entre dans une pièce, celle-ci s'éclaire entièrement. Le joueur possède un certain nombre de point de vie (défini à 10) au maximum, lorsque celle-ci est à 0, il meurt. Au fur et à mesure de ses déplacements, il perd de la nourriture (une barre allant de 0 à 100). Lorsqu'il a trop faim (nourriture inférieure à 10), il ne se déplace plus correctement, il perd certaines zones de sa mémoire de la carte (elle ne sont plus éclairées), et il perd de la vie. Le joueur peut trouver de la nourriture aléatoirement par terre (un total de 2 objets de nourriture par pièce, par étage), tout comme il peut trouver des soins de santé pour lui faire régénérer sa vie. L'autre façon de re-gagner de la vie est de manger, ainsi certains point de nourritures seront utilisées pour la vie.

Lorsque le joueur mange de la nourriture, il a une faible probabilité de chance que celle-ci était empoisonnée. Le joueur perd de la vie à chaque déplacement lorsqu'il est empoisonné. Pour ne plus l'être, il doit soit attendre quelques déplacements de ne plus l'être, soit consommer un kit de santé.

Durant son chemin dans le bâtiment, le joueur peut se faire avoir par des pièges. Ceux-ci peuvent, au choix, enlever de la vie, nous faire glisser à travers la pièce, ou nous faire tomber d'un ou plusieurs étages (une chance sur 3 pour chaque).

Les objets tels que la nourriture, les soins de santé, ou les pièges peuvent être récupérés par le joueur dans

son inventaire, pour les déposer ou les utiliser plus tard. L'inventaire du joueur possède 5 *slots*, et il ne peut y déposer qu'un seul objet par *slot*.

Des monstres se trouvent dans le jeu, ceux-ci nous attaquent, en se rapprochant de nous. Nous pouvons aussi les attaquer. Ils possèdent, tout comme nous, une agilité, qui est plus importante pour un monstre évolué. Cette agilité permet de faire plus de dégâts au combat, et de moins en subir. Ces monstres doivent avoir une mini-intelligence artificielle pour qu'ils puissent se déplacer de façon similaire à ce que un véritable joueur pourrait faire.

A tout moment, l'utilisateur peut sauvegarder sa partie sur l'un des trois emplacements de sauvegarde prévus. Il peut aussi quitter le jeu à n'importe quelle étape. Au lancement, l'utilisateur doit avoir un écran lui indiquant les sauvegardes, il peut en charger une, ou en supprimer une. Si il lance le jeu sur un emplacement vide, une nouvelle partie est alors créée.

A la fin du jeu, lorsque le joueur perd, ou gagne, il se retrouve sur l'écran initial des sauvegardes.

Au lancement du jeu, un texte explicatif avec l'objectif de celui-ci doit être affiché. Le jeu est décomposé en trois parties :

- la zone de jeu
- la zone de logs, où se trouve des informations sur ce que le joueur peut effectuer
- la zone de statistiques, qui indique toutes les propriétés du joueur : étage, vie, faim, nombre de déplacements, si il est empoisonné, si il a trouvé l'objet, ...

## 3.2 Comment jouer ?

- Lancement du jeu :  
Pour commencer à jouer, vous devez télécharger le jeu à partir de l'adresse suivante : `rogueLike` ; avec la commande suivante : `git clone git@github.com:TitouanT/rogueLike.git` Vous faites : `cd rogueLike`.  
Puis vous compilez grâce au `makefile` : `make install`.  
Le jeu commence dès que vous faites : `./rogueLike`.
- Les déplacements :  
Nous pouvons gérer nos déplacements sur la carte grâce aux flèches de direction.
- Interactions avec des objets :  
Les interactions avec un objet (serringues de soins, nourriture, escalier) se font avec la touche **entrée**.
- Ouvrir et fermer une porte :  
Si vous souhaitez ouvrir une porte, déplacez-vous devant la porte, appuyer sur la touche **o** et marqué la direction de la porte avec les flèches de direction. Mais pour fermer, c'est le même principe que pour ouvrir une porte sauf que la touche est **c** au lieu de **o**.
- Gestion de l'inventaire :  
Pour voir votre inventaire, vous devez appuyer sur la touche **i**. Pour prendre un objet (serringues de soins, nourriture), vous devez appuyez sur la touche **g** mais pour poser votre inventaire, vous devez appuyez sur la touche «d» et indiquer la case se trouve l'objet.
- Sauvegarder sa partie :  
Vous pouvez sauvegarder la partie à tout moment avec la touche **s**, cette manoeuvre n'arrêtera pas votre expérience de jeu.
- Combattre un monstre :

Pour combattre un monstre, utilisez les flèches de direction et déplacez vous sur le monstre jusqu'à ce qu'il meurt (disparaisse).

Ces commandes sont écrire dans le jeu en appuyant sur la touche ?. Certains commandes sont encore à découvrir comme se déplacer en diagonale pour échapper à des monstres ou des codes de triches peuvent être utiliser grâce à la touche \_, pour les voir faites \_ et taper **help** pour avoir la liste.



## Chapitre 4

# Codage, méthode et outil

ici mettre un petit mot

### 4.1 Structures et énumérations

Nous avons utilisé un certain nombre de structure et d'énumération pour modéliser le rogueLike.

#### 4.1.1 Une case de la carte

La structure la plus importante est celle qui représente une case de la carte. Cette structure contient un champ qui définit son type, un autre qui définit son état ainsi qu'un tableau listant les objets présents sur la case et deux entiers pour savoir si le joueur a découvert cette case et pour connaître le nombre d'objets présents.

Le type est une énumération des différents types de case possible : un vide, un mur, un encadrement de porte, une pièce ou un couloir.

L'état est une énumération utilisée pour les encadrements de portes afin de savoir si il y a une porte, et si elle est ouverte ou fermée. L'état est aussi utilisé pour les pièces afin de savoir si il y a de la lumière ou non dans la pièce.

Un objet est décrit par une structure qui est composée d'un type et d'un entier pour savoir si l'objet a été découvert par le joueur. Le type d'un objet est lui décrit par une énumération et définit les escaliers ascendants et descendants, la nourriture, les kits de santé et les pièges.

#### 4.1.2 Les êtres vivants

Le joueur contrôle un personnage que l'on représente à l'aide d'une structure. Elle permet d'enregistrer son nom, sa position (ligne, colonne et niveau), ses points de vie et d'attaque, son agilité, son expérience, son appétit, sa santé, le nombre de mouvement effectué, un booléen pour savoir si il a trouvé le QUELQUE CHOSE ainsi qu'un tableau d'objets qui représente son inventaire.

Les monstres ont eux aussi une structure qui enregistre leur nom, leur types, leur position (ligne, colonne et niveau), leur points de vie et d'attaque, leur champ de vision, leur agilité et quelques champs supplémentaires qui est utilisé comme mémoire pour leur IA. Les différents types de monstres sont L1, L2, L3, master, doctorant et fantôme.

### 4.1.3 Et le reste

La représentation en matrice des niveau n'étant pas dans tous les cas la plus efficace, nous avons créé une structure de niveau qui contient un tableau de pièces et le nombre de pièces que contient l'étage.

Une pièce est elle-même représentée par une structure qui contient la position de son angle supérieur gauche (ligne et colonne) ainsi que ses dimensions (largeur et hauteur).

Quelque fois, nous avons besoin de manipuler des position (ligne, colonne). Nous avons donc créé une structure de position qui contient une ligne et une colonne.

## 4.2 Séparation du code en modules

Afin d'avoir un code source clair et maintenable, nous l'avons séparé en plusieurs modules. Nous avons un module dédié pour :

- les affichages,
- les interactions (interprète les commandes du joueur),
- les lectures/écritures dans des fichiers,
- la boîte à outils,
- la génération des niveaux,
- et la prise en charge des monstres.

## 4.3 Détail des modules

On met du code comme ça :

Listing 4.1 – programme.c

```
1 #include <stdio.h>
2 #include "maFonction.h"
3
4 int main(){
5
6     int b;
7     b = test(10);
8     return 0;
9 }
```

### 4.3.1 La génération des niveaux

La génération d'un niveau se fait en quelques étapes :

1. Le choix du nombre de pièce à placer en fonction de l'étage qu'il représente (de 0 à 5).
2. Le placement des pièces sur la map.
3. La création des couloirs pour lier les pièces.
4. Le placement des objets (nourriture, escaliers, pièges).

## placement des pièces

Pour placer les pièces, on les places les unes après les autres. Pour chaque pièces, nous commençons par lui donner une position (la position de son angle supérieur gauche). Ensuite nous lui choisissons une dimension (hauteur, largeur) afin qu'elle entre la plus petite et la plus grande pièce possible. Pour finir, nous vérifions qu'elle n'est pas en collision avec une autre pièce. Dans le cas ou elle l'est, on recommence (pas plus de 100 fois).

## création des couloirs

Le but est de créer un réseau de pièces ou aucune pièce ou groupe de pièces ne soit isolé des autres. Pour cela nous connectons les pièces les unes après les autres dans leur ordre de création. L'algorithme qui crée les liens commence par initialiser la première pièces simplement en y ouvrant une porte. Ensuite, pour chaque pièces il procède ainsi :

- Création d'une ouverture dans la pièce.
- Création d'un couloir entre cette ouverture et la case de type couloir ou encadrement de porte la plus proche à l'aide d'un algorithme de recherche de chemin.

### 4.3.2 Les lectures et les écritures dans des fichiers

Nous gérons des sauvegardes sur plusieurs :

- Les fichiers comportant les paramètres l'ensemble des niveaux, soit un fichier texte pour un niveau ; nommé lvl.txt avec lvl variant selon le niveau.
- Un fichier texte comportant les donnees sur les monstres présents dans la partie ; nommé : monster.txt.
- un fichier comportant les données de chaque pièce ; nommé : lvlData.txt.
- un fichier comportant les données du personnage ; nommé : position.txt.

#### Les écritures :

Les niveaux sont sauvegardés lorsque du lancement d'une nouvelle partie, un changement de niveau, lorsque le joueur demande de sauvegarder. Les monstres sont sauvegardés lors de la création d'une nouvelle partie et la sauvegarde demander par le joueur. Les données de chaque pièce sont sauvegardés que lors d'une nouvelle partie. Quand aux données du personnage, elles sont sauvegardés que lorsque le joueur veut sauvegarder, l'apparition de cette sauvegarde montre qu'une partie est sauvegardée : si ce fichier n'existe pas lors du relanceront du jeu, le joueur n'aura pas accès à ses anciennes données.

#### Les lectures :

Les niveaux sont lus si le joueur veut charger une partie sauvegardée, un changement de niveau. Les monstres sont lus lors du chargement d'une partie existante comme les données sur le joueur et les données des pièces.

### 4.3.3 Les interactions et déplacements

Notre jeu devait se jouer de façon simple, le joueur devait pouvoir déplacer son personnage d'un simple appui sur une touche. L'utilisation de simples `scanf()` pour récupérer l'appui sur une touche aurait été difficile à gérer, car ces fonctions sont exécutées uniquement lorsque la personne appuie sur la touche

*entrée*. Pour palier ce problème, nous avons utilisé la librairie *ncurses*, qui est une TUI, une interface utilisateur par terminal. Celle-ci nous permet, entre autres, de récupérer les touches (ou combinaisons de touches) que l'utilisateur effectue sans avoir à attendre qu'il appuie sur *entrée*.

#### 4.3.4 L'affichage

L'affichage s'est aussi faite à l'aide de la librairie *ncurses*, car elle propose des fonctions de création de fenêtres virtuelles, et d'écriture directement sur celles-ci. Nous avons séparé notre jeu en trois zones, qui sont en réalité des fenêtres. Pour l'écran de chargement et de fin de partie (en cas de perte ou de victoire), des messages ASCII sont affichés, ces messages proviennent en réalité de fichiers textes, qui sont lus par le programme et affichés au centre de l'écran. L'écran de sélection quant à lui nous permet de visualiser facilement, avec des couleurs, les parties.

Lorsque le joueur découvre de nouveaux endroits dans les couloirs, l'état de visibilité des cases l'entourant change. Pour les pièces, puisque celles-ci doivent s'allumer entièrement, on vérifie si le joueur se trouve dans une pièce. Dans ce cas, on cherche la taille de la pièce où se trouve le joueur, puis on affiche chaque case de celle-ci.

Au moment d'afficher la carte, on n'oublie pas d'afficher les cases uniquement si elles sont visibles.

#### 4.3.5 Les monstres

Le nombre de monstre dans une partie est choisi aléatoirement entre deux bornes définies en constantes. Les monstres sont équitablement répartis entre les étages proportionnellement au nombres de pièces qu'ils contiennent. Chaque étages a une proportion différentes de chaque monstres, plus l'étage est haut, plus les types de monstres présents sont difficiles à battre.

Pour animer les monstres, nous marquons toutes les cases de la matrice accessible par le joueur avec la distance de celle-ci par rapport au joueur. Ensuite chaque monstre fait un mouvement en direction du joueur si le joueur est dans son champ de vision. Sinon, ils ne bougent pas.

Nous faisons attention à ce que les monstres n'aillent pas sur le joueur et si le joueur est à porté alors ils l'attaquent. Si le joueur fait un déplacement vers une case où se trouve un monstre alors le joueur ne bouge pas et attaque. Ainsi, il n'y a jamais deux êtres vivants sur la même case.

Les fantômes sont particuliers car :

- ils ne visent jamais le joueur,
- ils peuvent se déplacer partout sur la map,
- ils ont un halo lumineux autour d'eux, permettant au joueur d'apercevoir des zones qu'il n'a pas encore découvertes.

#### 4.3.6 La nourriture et la vie

A la génération des niveaux, nous sélectionnons des endroits libres (mais aléatoires) sur la carte, dans des pièces, pour y placer des objets de nourriture, et des objets de santé. Le nombre de ces objets trouvable dans le niveau est défini en fonction du nombre de pièces de celui-ci.

Lorsque le joueur se trouve sur un de ses objets, et qu'il effectue l'action pour s'en servir, sa nourriture ou sa vie est mise à jour. Ensuite, à chaque tour, on vérifie si le joueur possède plus de 90% de nourriture. Dans ce cas, il possède une chance sur trois d'augmenter sa vie (et de diminuer sa nourriture). Si il ne possédait pas 90% de nourriture, alors il possède 1% de chance d'augmenter sa vie.

Lorsque le joueur ne possède plus de vie, il possède 75% de ne pas réussir à se déplacer. Pour l'évanouissement, une position est choisie aléatoirement, puis deux longueurs, entre deux bornes. Le rectangle formé par ces données est ensuite passé en non-visible.

#### **4.3.7 L'inventaire du joueur**

L'inventaire du joueur est en réalité un tableau du même type que les objets de notre carte. Ainsi, lorsqu'on souhaite ajouter un objet à l'inventaire du joueur, il suffit de prendre l'objet, et le mettre dans l'inventaire, à une case disponible.

## Chapitre 5

# Résultat et conclusion

Faire une comparaison avec les objectifs. <titouan>

### 5.1 Améliorations possibles

<tout le monde>

Il aurait été possible d'améliorer notre projet en ajoutant des sons dans le jeu. On aurait aussi pu créer un didacticiel.

Le but aurait été d'ajouter un fond sonore dans le jeu et des sons lorsqu'un monstre attaque, ou est proche du joueur. Pleins de possibilités pourraient s'offrir à nous lorsque que nous trouverons un moyen de lire une bande son (avec un réglage pour de ne pas mettre de son pour ceux qui le souhaitent).

Un didacticiel serait aussi intéressant pour apprendre au joueur à jouer.

Un système de classe aurait pu être intéressant, au début, nous aurions pu choisir d'être un humain, un elfe, un nain, ou autre, avec des caractéristiques différentes (vie, vitesse, attaque, résistance, faim moyenne du joueur, ...).

Enfin, un développement du système d'expérience aurait rendu notre jeu plus jouable : lorsqu'un joueur tue un monstre, son expérience augmente, et plus son expérience est élevée plus il tue facilement d'autres monstres.

### 5.2 Apport personnel du projet

<tout le monde>

# Annexe

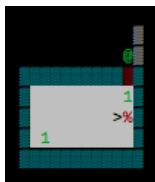
## 5.3 Cas d'utilisation du débogueur *GDB*

Parfois, nous n'arrivons pas à trouver la cause de certains bugs, qui étaient finalement très simple. Pour cela, nous avons utilisé l'outil *GDB*, comme mentionné dans la partie sur les outils, page 4. Afin de l'utiliser avec *ncurses*, voici la marche à suivre :

1. Ouvrir deux terminaux, un qui servira d'interface pour *GDB*, un autre, suffisamment grand pour notre jeu.
2. Sur le terminal du jeu, tapez la commande `tty` afin de connaître le nom du TTY de celui-ci. Tapez ensuite la commande `sleep 10000000` afin de dire au terminal d'attendre longtemps avant de faire une autre action.
3. Sur le terminal du débogueur, tapez le commande `gdb rogueLike` afin de lance *GDB* sur notre programme.
4. Une fois dans *GDB*, tapez `tty <nom du TTY>` avec `<nom du TTY>` celui trouvé à l'étape 2. Il est généralement de la forme `/dev/ttys000`.
5. Vous pouvez maintenant utiliser *GDB*.

### 5.3.1 Des monstres qui passent à travers les portes ...

Il arrivait parfois que des monstres passent à travers des portes fermées. Nous avons lancé *GDB*, et nous nous sommes rendu compte que la fonction qui testait si un monstre pouvait marcher sur une case était inversée dans le cas des portes :



(a) Porte fermée, monstre à l'intérieur



(b) Il a réussi à aller sur la porte

FIGURE 5.1 – Explication du bug

```

Valentin@MBP-Valentin-2 ~/git/rogueLike master » gdb rogueLike
GNU gdb (GDB) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin15.5.0".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from rogueLike...done.
(gdb) tty /dev/ttys001
(gdb) break main
Breakpoint 1 at 0x1000009fb: file src/main.c, line 26.
(gdb) run
Starting program: /Users/Valentin/git/rogueLike/rogueLike

Breakpoint 1, main () at sr
26      if (fileExi
Ajust d'un break au début de la fonction
qui indique si un montre peut se déplacer
(gdb) break monstre.c:213
Breakpoint 2 at 0x10000dbe8: file src/monstre.c, line 213.
(gdb) cont
Continuing.

```

(a) Lancement de *GDB*

```

Breakpoint 2, isItWalkableForAMonster (cell=...) at src/monstre.c:214
214      switch (cell.type) {
(gdb)      L'erreur se situe ici, on compare pas si la porte est fermée :
217          if (cell.state == dNONE) return FALSE;
(gdb)
218          else return TRUE;
(gdb)
222      }
(gdb)

```

(b) Erreur trouvée

FIGURE 5.2 – Résolution du beug

## Nourriture qui baisse trop vite ...

Lors de l'ajout de la nourriture, nous nous sommes vite rendu compte qu'elle diminuait à chaque tour, malgré la probabilité de 70% qu'elle ne diminue pas. Nous avons lancé *GDB*, nous avons tout d'abord vérifié que la nourriture était bien prise en compte dans les caractéristiques du joueur. Ensuite, en regardant de plus près la fonction qui gère la faim, nous avons pu tester manuellement les conditions pour la diminution de la nourriture. Nous nous sommes rendu compte que le *OU* de la condition était incorrect, et aurait du être un *ET*.



```

Indiquez votre code de triche :
I WANT TO EAT

Continuing.

Hardware watchpoint 3: player.food
Old value = 100
New value = 0
cheat (win_logs=0x100503880, win_game=0x1001037b0, map=0x7fff5fb44a70,
      player=0x7fff5fb40ff8, isPlayerInvincible=0x7fff5fb40f28) at src/interactions.c:246
246
}

```

(a) La nourriture est prise en compte

```

(gdb) print player->food
$2 = 99
(gdb) print didItHappen(30)
$3 = 0
(gdb)
$4 = 0
(gdb)
$5 = 1
(gdb)
$6 = 0
(gdb) print didItHappen(100)
$7 = 1
(gdb) print player->food > 0 || didItHappen(100)
$8 = 1
(gdb)
$9 = 1
(gdb) print player->food > 0 || didItHappen(0)
$10 = 1
(gdb) print player->food > 0 || didItHappen(0)
$11 = 1
(gdb) print player->food > 0 && didItHappen(0)
$12 = 0
(gdb)
$13 = 0
(gdb) print player->food > 0 && didItHappen(50)
$14 = 1
(gdb)
$15 = 1
(gdb)
$16 = 1
(gdb)
$17 = 1
(gdb)
$18 = 1
(gdb)
$19 = 0
(gdb)
$20 = 1
(gdb) :

```

(b) Les conditions *OU* sont incorrectes

FIGURE 5.3 – Résolution de la nourriture qui baisse vite