1ère année informatique

Projet de programmation C

Rapport final du projet

Binôme :
Hilaire Claire
Maghraoui Sahar

Encadrant :
M. Faverge Mathieu

21 décembre 2017

Table des matières

Chapitre 1

Introduction

Le but de ce projet est la modélisation du jeu **"Kitty Wonderland"** en le codant en langage C.

1.1 Kitty Wonderland

Kitty wonderland est un jeu simulant une bataille philosophique entre des chatons et des poneys philosophes. Chaque joueur possède une certaine énergie qui varie au long de la partie et il quitte la partie lorsqu'il est épuisé. Le joueur participe au débat en jouant des cartes (ses arguments dans le débat). Ces cartes influent sur ses attributs : ses idées (nécessaire à l'utilisation d'une carte), son énergie et ceux de son adversaire. Une partie se finit lorsqu'il ne reste plus qu'un joueur (le gagnant), ou que tous le monde à perdu. Le jeu est expliqué plus en détail plus tard (cf 2.1.1).

1.2 Organisation

L'écriture du code est faite via l'éditeur de texte Emacs et la compilation avec gcc. Le partage du code a été rendu possible par l'outil subversion. Le rapport à été réalisé en Latex grâce à l'outil en ligne sharelatex permettant son écriture collaborative. Enfin, les figures ont été réalisées avec draw.io.

1.3 Les étapes

Ce jeu est modélisé en deux versions "Base version" et "Achievement 1". On commencera par "Base version", en expliquant le principe de cette version et la problématique qui en découle, ainsi que notre choix d'implémentation, en précisant la complexité et la corrections des fonctions qui interviennent. On fera de même pour "Achievement 1". Enfin, nous discuterons les fonctions 'tests'.

Chapitre 2

Base version

2.1 Problématiques

2.1.1 Le jeu

Kitty Wonderland est un jeu de cartes où au moins 2 joueurs s'affrontent sur un plateau de jeu. Chaque joueur a des points d'énergie (que l'on peut voir comme les points de vie) et une jauge d'idées (que l'on pourrait aussi désigné comme des points de mana) qui s'incrémentent d'un gain propre au joueur à chaque tour.
Chaque joueur dispose d'une main de 5 cartes, avec un coût en matière d'idées pour chaque carte et une probabilité d'être tirée qui lui est propre. Les cartes disponibles sont :

Carte	Coût	Effet	Probabilité
Kitty Think	5	Augmente le gain du joueur de 1	$\frac{20}{131}$
Kitty Steal	10	Augmente le gain du joueur de 1 et diminue celui de l'advervaire de 1	$\frac{10}{131}$
Kitty Panacea	2	Augmente l'énergie du joueur de 10	$\frac{50}{131}$
Kitty Razor	2	Diminue l'énergie de l'adversaire de 10	$\frac{50}{131}$
Kitty Hell is others	100	Diminue l'énergie de l'adversaire de INT_MAX [1]	$\frac{1}{131}$

Début de partie :
Au début du jeu, chaque joueur dispose de :
50 points d'énergie, 0 idées, un gain de 1 et 5 cartes qui sont tirées aléatoirement dans le momento (Deck).
Tour de jeu :

1. INT_MAX est l'entier maximal atteignable

A chaque tour du jeu, chaque joueur met à jour sa jauge d'idées en fonction de son gain et choisit une carte selon ses idées : s'il en a suffisamment, il joue sa carte, déduit son coût de sa jauge d'idées et la défausse, sinon il passe son tour. À la fin du tour, chaque joueur reconstitue sa main en repiochant des cartes jusqu'à avoir de nouveau une main complète, et si un joueur a perdu tout ses points d'énergie, on le déclare perdant et il ne participe plus à la bataille.

fin de la partie :
Le jeu prends fin lorsqu'il n'y a plus qu'un seul joueur, déclaré alors vainqueur ; ou lorsque tout le monde a été éliminés : partie nulle.

2.1.2 Les impératifs

La modélisation de ce jeu requiert différents points. Tout d'abord, il faut trouver une représentation propice des différents éléments : comment choisir les structures des joueurs, des cartes et du plateau de jeu ? Quel type choisir pour modéliser la main du joueur ? Ensuite, il faut conceptualiser les algorithmes gérant la dynamique du jeu pour qu'ils collent au fonctionnement attendu. De plus, il faut répartir ces fonctions dans plusieurs fichiers sources, et donc trouver un découpage adéquat. Ce découpage ne doit pas interférer avec les appels mutuels des fonctions à la compilation. Enfin, pour gérer la compilation séparée de ces fichiers et ne pas recompiler inutilement des fichiers non modifiés, il faut établir un Makefile gérant les compilations.

2.2 Choix d'implémentation

2.2.1 Modélisation

La modélisation de ce jeu est basé sur des structures.
Une carte est modélisée par une structure **card** qui contient les champs suivants :

- **id** : son identifiant
- **name** : son nom
- **cost** : son coût
- **gain-player** : le gain qu'elle offre au joueur
- **gain-adversary** : le gain qu'elle prend à l'adversaire donc négative
- **energy-player** : l'énergie qu'elle offre au joueur
- **energy-adversary** : l'énergie qu'elle prend à l'adversaire :donc négative

L'ensemble des cartes est catalogué dans le tableau `card_list` contenant chacune des différentes cartes. Pour modéliser l'absence de cartes, on crée la carte fantôme supplémentaire "Null". Chaque carte à un identifiant, qui correspond à sa case dans `card_list`. Ceci permet de confondre les cartes avec leur idendifiants, et ainsi travailler avec des entiers et non avec des structures lourdes.
Remarque : une amélioration possible aurait été de travailler avec des pointeurs plutôt que des identifiants.

Carte	id	Coût	Gain Joueur	Gain Adversaire	Énergie Joueur	Énergie Adversaire
Null	0	0	0	0	0	0
Kitty Think	1	5	1	0	0	0
Kitty Steal	2	10	1	1	0	0
Kitty Panacea	3	2	0	0	10	0
Kitty Razor	4	2	0	0	0	10
Kitty Hell is others	5	100	0	0	0	INT_MAX

Le joueur est de même modélisé par une structure **player** qui contient les champs suivants :

- **energy** son énergie
- **ideas** ses idées
- **gain** son gain
- **state** son état : mort ou vivant.
- **hand** sa main sous forme d'un tableau qui contient les id de ses cartes
- **deck** son momento : c'est un pointeur. Ce champ a été ajouté lors de la réalisation de l'Achievement 1 pour avoir une compatibilité entre les deux versions (cf 3).

Enfin le plateau de jeu est modélisé par une structure qui contient les champs :

- **number_players** : le nombre des joueurs d'une partie.
- **players_alive** : le nombre des joueurs vivants.
- **players** : tableau des joueurs.
- **round** : le tour de jeu en cours.

Le champ round est incrémenté à chaque tour, et permet ainsi de stopper le jeu si la partie est trop longue.

2.2.2 Le découpage des fichiers

L'organisation du code se fait en plusieurs fichiers.
Le projet contient deux headers : `definition.h` qui contient les définitions des structures de base, et `fonctions.h` qui contient les prototypes des toutes les fonctions, permettant les appels mutuels des fonctions. Ces fonctions sont réparties dans trois fichiers source : `board.c`, `cards.c` et `deck.c` (ce dernier a été créé par la pour la compatibilité avec l'Achievement 1, cf 3). Enfin, tous ces éléments sont integrées avec les headers dans le fichiers `game.c` qui gère la boucle de jeu. Ils peuvent aussi être compilés avec les trois fichiers test, `test_deck.c`, `test_board.c`, et `test_cards.c`, pour tester les fonctions prises individuellement (cf 4.2.2) .

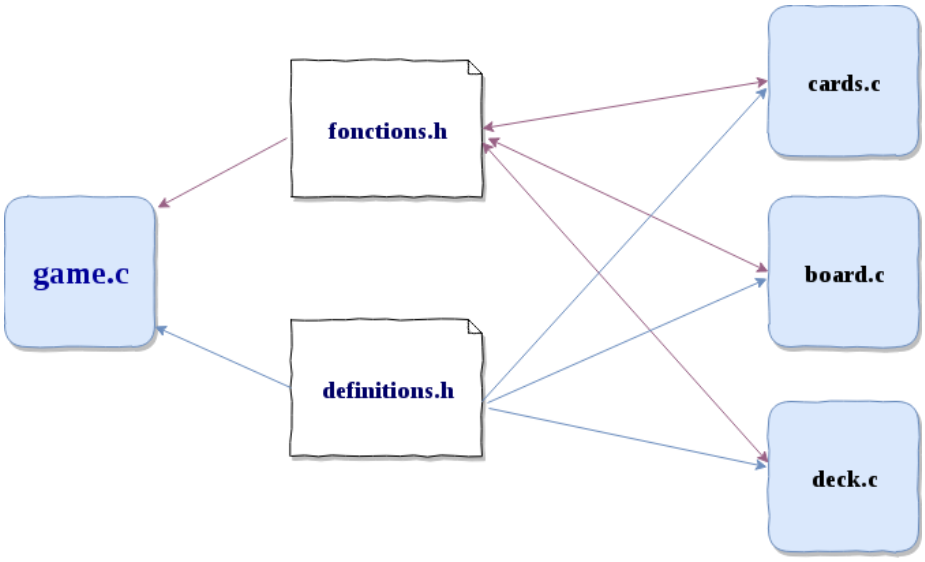

FIGURE 2.1 – découpage des fichiers du Base version

2.2.3 Les fonctions du plateau de jeu

Fonctions d'initialisation du board :

L'initialisation du plateau de jeu se fait à travers la fonction `init_board` , on passe en paramètre le nombre de joueurs number_players qui est aussi égal au début du jeu au nombre de joueurs en vie : players_alive. Pour chacun des es joueurs, on initialise son états : state à 1, pour dire qu'il vivant en début de partie, son énergie à 50 points, son idées à 0, son gain à 1 et sa main vide. Enfin, on fait un tirage aléatoire de 5 cartes pour remplir sa main en faisant appel à la fonction `refill_card` (cf 2.2.4)

Complexité :

Si on note n le nombre de joueurs passé en paramètre, alors puisque au sein de la fonction `init_board` on fait un parcours complet du tableau des joueurs et on fait appel à la fonction `refill_cards`, qui a une complexité constante (cf2.2.4). Quant aux fonctions `init_deck` et `distribute` qui sont aussi appelées par `init_board`, elles ne font rien dans Base version (cf(3)). Donc la fonction `init_board` est de complexité linéaire par rapport au nombre de joueurs $\theta(n)$.

Fonctions de la dynamique du jeu :

A chaque début de tour, il est nécessaire d'augmenter les idées de chaque joueur en fonction de son gain. D'où le besoin de la fonction : `Increase_ideas`. A la fin de chaque tour, il faut également vérifier si l'un des joueurs est mort *i.e.* si son énergie est négative ou nulle, et si c'est le cas, le noté comme mort (changer son champ state pour 0) et décrémenter le nombre des joueurs en vie : c'est le rôle de la fonction `death`.

Complexité :

Les deux fonctions effectuent un parcours complet du tableau des joueurs pour modifier les champs des joueurs, auxquels elles ont accès en temps constant. Elles s'effectuent donc en $\theta(n)$.

Affichage :

Les dernières fonctions se rapportant au plateau de jeu sont des fonctions d'affichage :`display_players` et `announce_results`, permettant respectivement d'afficher l'état de la partie en fin de chaque tour, et d'annoncer le résultat à la fin de la partie. La parti se finit lorsqu'il y a moins de 2 joueurs ou lorsque le maximum de tours de jeu est atteint.

Complexité :

En cas de match nul ou de dépassement du nombre de tours autorisés, la fonction `announce_results` a accès aux données nécessaire à l'annonce du résultat en temps constant par le board. Sinon, elle dois parcourir le tableau des joueurs pour trouver le vainqueur et s'effectue donc en $\theta(n)$. Elle est donc en $\mathcal{O}(n)$.

2.2.4 Les fonctions des cartes

Les choix aléatoires :

Le jeu est grandement basé sur l'utilisation aléatoire des cartes : chaque carte est piochée et jouée de façon aléatoire (dans la mesure des règles d'utilisation). Il est donc nécessaire de créer une fonction `random_card`. La génération aléatoire de chacune des cartes, en respectant les règles de probabilités énoncées plus haut (cf 2.1.1).

Pour cela on génère un nombre r àléatoire entre 0 et 130 :
si r $\in$ [0,20[, alors c'est la carte Kitty Think
si r $\in$ [20,30[, alors c'est la carte Kitty Steal
si r $\in$ [30,80[, alors c'est la carte Kitty Razor
si r $\in$ [80,130[, alors c'est la carte Kitty Panaceas
si r = 130, alors c'est la carte Kitty Hell is Other

Cette fonction permet le tirage aléatoire d'une carte parmi les 5 proposées. Cependant, pour des raisons de compatibilité entre l'Achievement 1 et la Base version (cf 3), elle n'est pas appelée directement pour le tirage d'une carte, mais elle est appelée dans la fonction wrap `draw_card`, qui est la seule fonction non vide de `deck.c`.

Complexité :
La génération d'une carte aléatoire est en temps constant, $\theta(1)$.

Il est aussi nécessaire de choisir un adversaire de façon aléatoire. Le choix de cet adversaire ce fait dans `choose_adversary` selon l'algorithme suivant :

Algorithm 1 algorithme du choix de l'adversaire

Require: joueur numéro p ; n joueurs, a joueurs vivants
{//le joueur peut choisir parmi les a-1 candidats, *i.e.*les joueurs vivants autres que lui : il choisit le r$^{\text{ème}}$ joueur dans cet ensemble des façon aléatoire}
$r \leftarrow$ random[0,a-1]
{//on cherche le numéro adv correspondant au r$^{\text{ème}}$ candidats}
{//d'abord, on passe les r candidats précédants}
while $r \neq 0$ **do**
  if adv est un candidat **then**
    $r \leftarrow r - 1$
  end if
  $adv \leftarrow adv + 1$
end while
{//à présent, le premier joueur vivant autre que p est le bon advesaire}
while adv n'est pas un candidat **do**
  $adv \leftarrow adv + 1$
end while

Remarque :
Dans l'algorithme, on a forcément a-1>0 car pour entrer dans la boucle de jeux, le nombre de joueur doit être ≥ 2 (sinon la partie se serait achevée au tour d'avant), et la procédure de mort n'est appelée qu'à la fin du tour.
Prenons par exemple la situation de la figure 2.2 : il y a 10 joueurs parmi lesquels les joueurs 1, 2 et 6 sont morts. Le joueur 8 peut donc choisir parmi les 6 joueurs 0, 3, 4, 5, 7 et 9. Ces derniers sont renumérotés de 0 à 5 pour le tirage aléatoire, le but de l'algorithme et de retrouver la correspondance entre le numéro tiré et l'identifiant du joueur.
Complexité :
Dans le pire des cas, la fonction va devoir parcourir tout le tableau de joueurs pour trouver l'adversaire. Elle s'effectue donc en $\mathcal{O}(n)$.

Les fonctions dynamiques des cartes :

A chaque tour, chaque joueur choisit une carte : il choisit la première de sa main que lui permettent ses idées selon son coût. Ceci est réalisé dans `select_card`. Le choix de la carte jouée est bien aléatoire puisque les cartes arrivent de façon aléatoire dans sa main avec `draw_card`.
Complexité :
Le choix de la carte nécessite dans le pire des cas un parcours complet de la

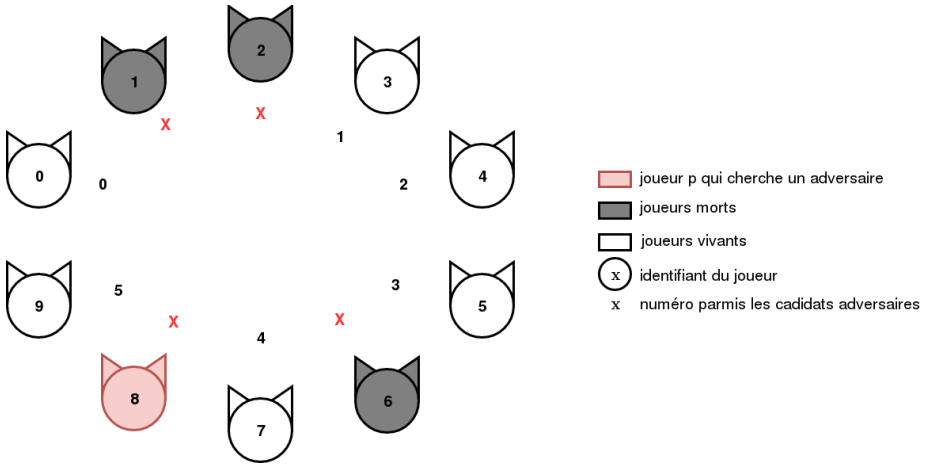

FIGURE 2.2 – exemple de situation de choix d'un adversaire

main en faisant appel à des instructions et des fonctions réalisées en temps constant, mais comme la main est de taille constante (5 cartes), la complexité de `select_card` est en $\theta(1)$.

Une fois que la carte et l'adversaire sont choisis, il faut appliquer les effets de la carte au joueur et à son adversaire, c'est le rôle de `apply_card`.
`Complexité :`
Elle agit sur les champs des joueurs qui sont des action en temps constant, mais c'est elle qui appelle `choose_adversary` qui est en $\theta(n)$, donc elle l'est également.

Enfin, à la fin de chaque tour, le joueur rempli sa main grâce à `refill_cards`. Cette dernière parcourt la main du joueur pour trouver les emplacements vides est appeler **draw_card** si tel est le cas.
Complexité :
`draw_card` est de complexité constante et la main est de taille fini donc le remplissage de la main se fait en temps constant.

Chapitre 3

Achievement 1

3.1 Problématique

Jusqu'à présent, les cartes piochées étaient générées aléatoirement dans une sorte de deck virtuel. À partir de l'achievement 1, chaque joueur se voit attribué un deck de 50 cartes en début de partie. Ainsi, lorsqu'un joueur veut piocher une carte pour remplir sa main, il la pioche au sommet de son deck, et lorsqu'il joue une carte, elle est mise dans sa fausse. Quand son deck est vide, les cartes de sa défausse sont mélangées pour reformer un deck.

Il fallait donc intégrer ce nouvel élément dans la dynamique du Base version, en conservant au maximum sa structure et ses fonctions, et notamment en conservant le fichier principal gérant la boucle de jeu tel quel.

3.2 Choix d'implémentation

3.2.1 Adaptation des fichiers

Pour intégrer ce nouvel élément dans la structure précédente, on utilise les avantages de la compilation séparée :

- on crée un fichier source `deck_ach1.c` contenant la définition de la structure du deck ainsi que les définitions de toutes les fonctions utilisant le deck.
- on crée un fichier source `deck.c` contenant une définition vide de la structure du deck (puisqu'il n'y avait pas de deck dans la Base version) ainsi que les définitions des fonctions de base version équivalentes à celle de `deck_ach1.c`, avec des entêtes identiques, quitte à faire des fonctions vides lorsqu'il n'existe pas de telles équivalences.
- les fonctions se rapportant au decks ayant ainsi le même prototype dans la Base version et dans l'Achievement, on met les prototypes communs dans le header `fonctions.h`. Ainsi on compile avec la source `deck.c` lorsqu'on veut utiliser les fonction pour la Base version, et `deck_ach1.c` pour utiliser

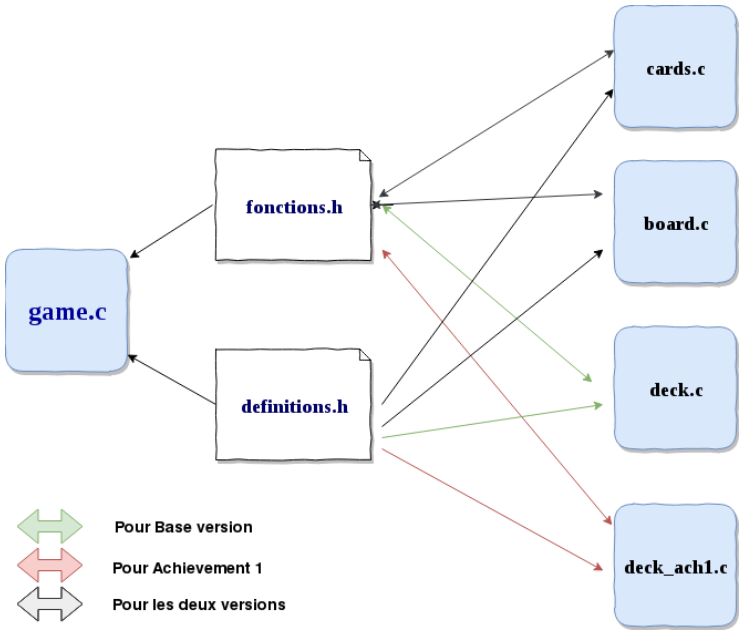

FIGURE 3.1 – Organisation des fichiers

celles de l'Achievement 1.
Ainsi, comme on peut le voir sur la figure 3.1, la génération du jeu dans les deux versions diffère d'un seule fichier, et l'organisation générale reste proche de celle le la Base version (cf 2.1).

Les decks étant tous dans le même fichier source, on y accède seulement grâce à l'adresse de leur zone mémoire, donc on ne les manipule qu'avec des pointeurs. Pour que chaque joueur ait accès à son deck, on a rajouté un champ "deck" qui est un pointeur vers son deck.

3.2.2 Modélisation

Le deck est modélisé par une file d'identifiants de carte : chaque fois que le joueur pioche une carte, elle est défilée de la file, chaque fois qu'il en défausse une, elle est enfilée à la file.

Pour modéliser cette file en C, on prend un tableau de 50 cartes, parcouru par un indice de début **top** et un indice de fin **bottom** (cf 3.3). Lorsqu'un joueur pioche une carte dans son deck, il récupère l'identifiant de la carte désignée par top puis on incrémente ce dernier ; et lorsqu'il se défausse d'une carte après l'avoir jouée, l'identifiant de la carte est enregistrée dans la case désignée par bottom et ce dernier est incrémenté. Les incrémentations se font modulo 50 pour passer de l'indice 49 à 0 et ainsi boucler le tableau sur lui-même. Les cartes allant de top (inclus) à bottom (exclu) sont donc réellement dans le deck, tandis que celles de bottom (inclus) à top (exclu) sont en réalité des places réservées pour

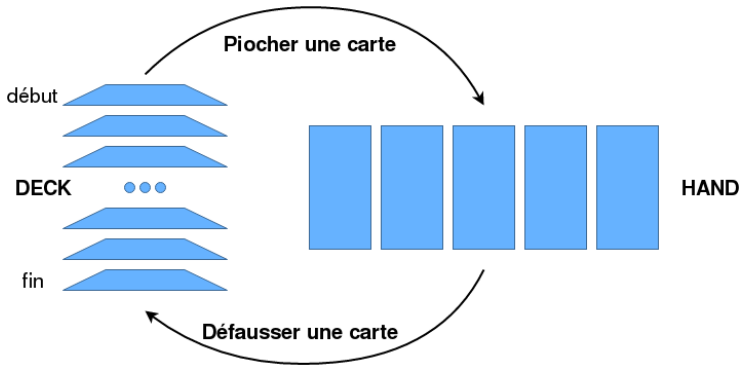

FIGURE 3.2 – représentation en file du deck

les cartes hors du deck (donc dans la main du joueur). Cependant, les identifiants de carte qu'elles contiennent sont quelconques : ce ne sont pas les mêmes que celles dans la main car le joueur ne joue pas ces cartes dans le même ordre qu'il les pioche, d'où l'intérêt d'avoir deux indices.

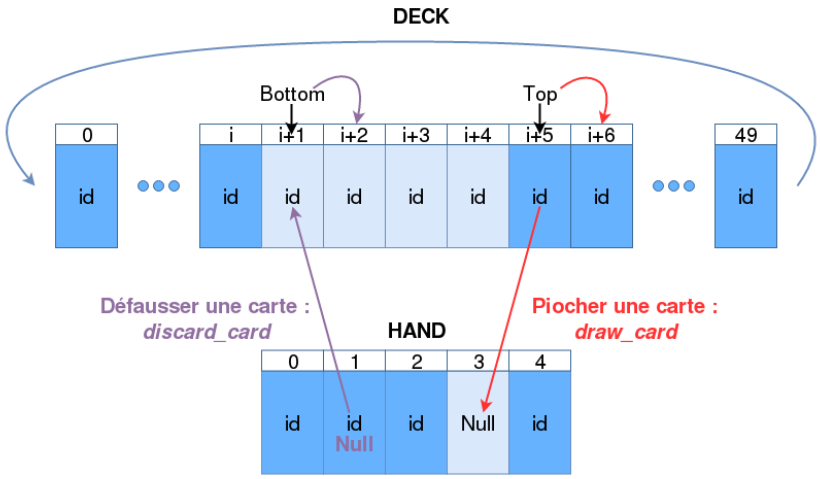

FIGURE 3.3 – représentation avec un tableau du deck

Tous les decks des joueurs sont contenus dans un tableau **decks** statique (sa taille étant fixe, on l'initialise au nombre maximal de joueur autorisé : MAX_PLAYERS). Le champ deck du joueur pointera donc vers la case de ce tableau correspondante.

3.2.3 Les fonctions du deck

Initialisation des decks

L'initialisation des decks se fait en deux temps, la génération de n decks (n étant le nombre de joueur) de 50 cartes prises aléatoirement pour chacun des n joueurs avec `init_decks`, et l'attribution de ces decks à leurs propriétaires. L'attribution des decks est fait dans la fonction `distribute`, qui consiste à copier, pour chaque joueur p, l'adresse du $p^{ième}$ deck dans son champ deck.

Ces initialisations se font à la fin de l'initialisation du board. pour la Base version, les fonctions équivalentes étaient appelées et ne faisaient rien.

Complexité :

L'algorithme d'initialisation `init_decks` parcourt donc les l cartes du deck (l étant la longueur d'un deck), pour chacun des n joueurs, et ne fait appel qu'à des fonctions en temps constant, donc sa complexité est en $\theta(l.n)$. Comme la taille des decks est fixée à l=50, on a du $\theta(n)$.

La fonction `distribute` est aussi un parcours du tableau des n joueurs, avec un affectation en temps constant pour chacun, donc il est aussi en $\theta(n)$.

Remarque : la fonction `init_board` qui les appelle étant déjà en $\theta(n)$ (cf 2.2.3), sa complexité n'est pas modifiée par l'ajout de ces fonctions.

fonctions de la dynamique du deck

Les fonctions `draw_card` et `discard_card` gèrent l'enfilement et le défilement de la file, selon les règles énoncées plus tôt. On aura toujours bottom $\leq$ top car le joueur l'algorithme du jeu ne permet pas au joueur de jouer plus de carte qu'il n'en a pioché dans son deck, avec égalité quand le deck est plein.

Complexité :

le fait d'avoir des indices désignant le début et la fin du deck permet de réaliser ces actions en temps constant.

Pour `discard_card`, il n'y a pas de problème tant que bottom est incrémenté modulo 50.

Pour `draw_card` en revanche, le problème se pose lorsque top atteint la valeur 50 : le deck est vide, il faut mélanger le deck avant de remettre top à 0. Cependant, il faut faire attention à ne pas prendre les cartes entre bottom (qui vaut normalement 45) et top, qui ne sont pas des cartes dans le deck, car cela modifierait la composition du deck. Le mélange est donc réalisé parmi les cartes de 0 à bottom, et non toutes les cartes du deck.

Le mélange suit l'algorithme suivant :

Algorithm 2 Algorithme de mélange du deck

Require: le deck est réduit à ses indices de 0 à bottom
  for i indice parcourrant le deck **do**
    $c \leftarrow$indice aleatoire du deck
    Echanger les éléments aux indices i et c
  end for

Chacune des cartes est échangée au moins une fois avec une cartes prise aléatoirement dans le deck, et en restant entre les indices 0 et bottom, on s'assure bien que seules les cartes présente dans les deck sont prise en compte.

Complexité :

l'algorithme de mélange parcours le deck une fois en faisant des affectations qui se font en temps constant, qui lui même est de taille constante : il est se fait donc en temps constant.

Chapitre 4

Les fonctions Test

4.1 Difficultés et évolution

Au cours de l'encodage du projet, il a fallut tester les fonctions prises individuellement grâce à des fonctions test. Ce sont donc des fonctions qui simulent une situation initiale, appellent la fonction à tester, et permette de comparer le résultat avec la situation attendu.

Au début, ayant mal compris leur fonctionnement attendu, nous avions au début fait des fonctions d'affichage, permettant d'afficher l'état des élément de plateau concerné et s'assurer visuellement de la bonne évolution de cet élément. Ce n'est que par la suite que nous avons repris ces fonctions test pour qu'elles s'occupent elles-même de la comparaison et informe l'utilisateur seulement de la présence où non d'erreur.

Une difficultés supplémentaire a été l'implémentation des fonctions test du deck. En effet, n'ayant pas accès à la structure du deck ou au tableau des deck "decks" (car seules les fonctions du fichier `deck_ach1.c` y ont accès), il a fallut créer des fonctions auxiliaires dans `deck_ach1.c` qui sont utilisées par les fonctions test.

4.2 Implémentation

4.2.1 Découpage des fichiers

Les fonctions test sont réparties en 3 fichiers : `test_deck.c`, `test_board.c`, et `test_cards.c`, testant respectivement les fonctions liées aux decks, à la table de jeu et aux cartes ; les deux derniers faisant les tests avec le deck de la Base version et de l'Achievement 1.

4.2.2 Fonctions

Chacune des fonctions test fonctionne de la façon suivante :

- La fonction est appelée sur un board quelconque initialisé dans le main du fichier dans laquelle elle se trouve.
- Elle modifie ce board pour créer une situation initiale connue, pour éviter toute influence externe sur le résultat, et si possible de placer dans une situation que pourrait poser problème.
- Elle appelle la fonction à tester
- Elle vérifie que la situation est bien conforme à ce qui était attendu.
- Si un point n'est pas conforme, elle envoie un message d'erreur signalant à quel niveau est repéré le problème et retourne 1.
- Si tout est conforme, elle retourne 0.
- La fonction main du fichier compte le nombre de fonction test du fichier signalant une erreur.

La principale difficulté a été pour les test du deck, puisqu'il fallait permettre au fonction test d'accéder aux différents champs des decks alors qu'elle n'avaient pas accès à leur structure. Pour cela, six fonctions auxiliaires ont été ajoutée au fichier source `deck_ach1` : `get_top`, `get_bottom`, `get_ith_cards`, pour accéder respectivement au top, au bottom et à la $i^{\text{ème}}$ carte du deck, et `modif_top`, `modif_bottom`, `modif_ith_card`, permettant de modifier ces mêmes éléments.

De même, une fonction auxiliaire était nécessaire pour la fonction `ditribute_test`. Cette dernière vérifie que les champs deck des joueurs contiennent bien les adresses des deck correspondants, dans le tableau "decks", en comparant lesdites adresses et les contenu des champ deck. Or seules les fonctions de `deck_ach1` ont accès à ce tableau. Elle fait donc appelle à la fonction `get_address` qui est encodé dans `deck_ach1` et retourne cette adresse.

Problème non résolu : la fonction `apply_card`, contient un affichage pour afficher ce que fait chaque joueur lors d'un tour : cet affichage est donc réalisé lors de l'appel des fonctions test. une solution simple aurait été de ne pas réaliser un tel affichage et de se contenter des affichages de `display-player`, mais cela aurait été dommage pour l'exécution des jeu.

Chapitre 5

Conclusion

Ce projet est encore loin d'être parfait. Il aurait pu être amélioré en allant plus loin dans les achievements proposés. Il aurait aussi pu être amélioré en accédant à card_list non pas avec des identifiant des cartes mais avec des pointeurs, ce qui aurait permis de mettre ce tableau en statique dans un fichier source avec toute les fonctions qui y accèdent, comme cela a était fait pour le tableau decks.

Cependant, il a permis de poser un bon nombre de bases en programmation C. Il a permis de se familiariser à la programmation séparée, et notamment avec la création et l'utilisation d'un Makefile. Il a aussi permis de mieux comprendre la logique des structures et des pointeurs. Enfin, ce projet à permis de s'entraîner à la résolution d'un problème complexe en travaillant en équipe.