ENSEIRB-MATMECA

Rapport de projet 1A

Kitty Wonderland : A Battle of Wits

Rémi Blaise
Simon Despretz

Chargé de projet :
M. Vinh-Thong Ta

15 décembre 2017

Table des matières

1 Introduction

Kitty Wonderland est un jeu dans lequel des joueurs s'affrontent au tour par tour à l'aide de cartes. Chaque joueur se voit attribuer un mémento dont le contenu est généré aléatoirement en début de partie. Ils piochent alors cinq cartes pour former leur main. Le but d'un joueur va être d'éliminer les autres joueurs jusqu'à être le dernier survivant. Pour cela, à chaque tour, les joueurs vont jouer simultanément une carte de leur main.

Les joueurs évoluent également sur un plateau torique (sur lequel un seul mouvement permet de se déplacer d'un bord à l'autre). Au début du tour, ils se déplacent simultanément d'au plus cinq cases, ou restent à leur position s'ils sont plusieurs à vouloir atteindre une même case.

Un joueur possède trois statistiques fondamentales : la vie, le mana et la génération de mana par tour. Amener un joueur à 0 points de vie l'élimine. Jouer une carte consomme du mana, et la génération de mana par tour permet au joueur de d'augmenter son mana.

Dans notre cas, le comportement des joueurs est simulé par l'ordinateur. Dans un soucis de simplicité, il sera simple et en partie aléatoire.

La première partie consistera en une explication des problématiques de la réalisation de ce projet. Ensuite, la seconde partie présentera les solutions mises en place pour répondre à ces problématiques. Viendront ensuite un retour sur les difficultés principales rencontrées durant ce projet, puis une présentation du résultat obtenu. Finalement, les améliorations possibles du projet seront indiquées.

2 Problématiques

La réalisation de ce projet s'est déroulée en plusieurs étapes appelées "succès", se dévoilant au fur et à mesure de la progression. Il a donc fallu adopter une méthodologie de développement progressive et modulaire.

Le premier succès consiste en l'élaboration de la structure de base du jeu : les systèmes de cartes et de joueurs, les cinq premières cartes, la logique du déroulement du jeu, la détection de la fin de la partie ainsi que l'implémentation des premiers comportements des ordinateurs.

Le deuxième succès introduit la notion de mémento pour chaque joueur dans lequel les joueurs piochent leurs cartes. Les mémentos sont remélangés dès que toutes les cartes ont été piochées.

Le troisième succès apporte au jeu un plateau, le déplacement des joueurs sur ce plateau ainsi qu'une nouvelle carte permettant de tenter d'éliminer ses adversaires en les bloquant.

Le quatrième succès permet aux joueurs d'invoquer des amis grâce à une nouvelle carte. Les amis sont des entités similaires aux joueurs mais aux capacités restreintes.

Chaque succès est accompagné d'une batterie de tests afin de certifier son bon fonctionnement tout au long du développement.

3 Solution

Dans cette partie, la solution logicielle réalisée sera détaillée. Afin de présenter l'évolution du projet, les apports des succès successifs et les adaptations nécessaires seront détaillées ici .

3.1 Succès 0 : Mécaniques de base

3.1.1 Structures de données

Pour les mécaniques de bases du jeu, il a fallu mettre en place plusieurs structures de données :

— la structure *player* qui contient toutes les caractérisques d'un joueur,
— la structure *card* qui contient toutes les caractérisques d'une carte,
— le tableau *players* contenant chaque joueur en jeu.

Structure joueur

Un joueur est alors défini par :
— son nom, un chaine de caratères
— sa vie, un entier
— son mana, un entier
— sa génération de mana, un entier
— sa main, qui contient les cartes qu'il est susceptible de jouer
Son implémentation a donné :

```
struct player {
  int life;
  int mana;
  int mana_regen;
  char name[64];
  const struct card * hand[HAND_SIZE];
};
```

La main devenant un tableau de pointeurs de carte, chaque carte étant une constante pour pour le jeu définie dans un autre fichier.

Structure carte

Une carte est alors définie par :
— son nom, un chaine caratères
— son coût en mana, un entier
— son effet
Son implémentation a donné :

```
struct card {
  char name[64];
  void (* apply) (struct player *, struct player *);
  int cost;
};
```

L'effet étant alors codé comme un pointeur vers une fonction d'application prenant en paramètre un joueur cible et un joueur lanceur à qui les effets seront appliqués. Ceci permet d'avoir une forme générique de fonction d'application.

Le tableau des joueurs

Enfin, le programme utilise un tableau de joueurs comme structure de travail pour toutes les fonctions, ce tableau, *players*, est initialisé en début de jeu. Dans le programme, *players* est un tableau de pointeurs vers des joueurs.

3.1.2 Les cartes

À présent, voici la description des cartes implémentées dans le premier succès :
- Panacea, qui augmente la vie de son lanceur de 10 pour 2 de mana
- Razor, qui diminue la vie de sa cible de 10 pour 2 de mana
- Think, qui augmente la génération de mana de son lanceur de 1 pour 5 de mana
- Steal, qui augmente la génération de mana de son lanceur de 1 et diminue la génération de mana de sa cible de 1, la génération de mana ne pouvant pas être inférieure à 1, pour 10 de mana
- Hell is other, qui tue sa cible pour 100 de mana

Lors du jeu les cartes seront piochées avec des probabilités inversement proportionnelles à leur coût, codée ici par la fonction *draw_random_card*. Cette fonction utilisant l'aléatoire du module *rand* pour sélectionner une carte selon leur probabilité.

3.1.3 Le jeu et son implémentation

Principe du jeu

Le jeu est au tour par tour, chaque tour étant composé des phases suivantes :
- Tous les joueurs jouent une carte s'ils le peuvent
- Le mana est généré
- Les joueurs éliminés sont retirés du jeu
- Test de la fin de la partie

Ces phases sont gérées par une fonction chacune, qui sont appelées par la boucle principale du programme gérée par la fonction principale *play_game* qui s'occupe également l'initialisation des données du jeu (le tableau *players*).

Phase des cartes

Les joueurs n'ayant aucune notion de stratégie, ils jouent de manière aléatoire une carte de leur main dont le cout en mana est inférieur à leur mana. La fonction responsable de cette phase est *play_turn* qui tire aléatoirement une carte à jouer et un joueur cible dans le tableau *players*, pour ensuite appliquer l'effet de la carte grâce au pointeur de fonction de la carte. Le joueur pioche ensuite une carte pour remplacer celle jouée via la fonction *draw_random_card*. Si aucune carte de la main n'est jouable, le joueur ne fait rien. Cependant, il était demandé, dans une version antérieur du succès, de redonner une nouvelle main au joueur s'il ne pouvait pas jouer, cette action pouvant être réalisée en appelant la fonction *deal_hand* qui distribue une main complète à un joueur.

Phase du mana

Le mana est ensuite incrémenté par la fonction *generate_mana* qui travaille sur le tableau *players* et lit la valeur *mana_regen* puis l'ajoute à la valeur *mana* pour chaque joueur du tableau.

Phase d'élimination

La fonction *purge_the_dead* gère cette phase en regardant la vie de chaque joueur et éliminant du tableau les joueurs dont la vie n'est pas strictement positive. Pour éliminer un joueur de *players* un appel à la fonction *die* est réalisé. Cette fonction recopie le pointeur du dernier joueur en vie à la place du pointeur du joueur mort, puis décremente la taille du tableau des joueurs. Le paramètre *n_players*, taille de *players*, est toujours donné en pointeurs aux fonctions afin de pouvoir de modifier la taille d'un tableau.

```
void die(int index_player, struct player* players[], int * n_players) {
  (*n_players)--;
  players[index_player] = players[*n_players];
}
```

Phase de fin

Cette phase est directement géré par *play_game* sa boucle principale testant la valeur de *n_players* et s'arrêtant si elle n'est pas strictement supérieure à un. Il y a alors deux cas possibles :

— Un joueur est encore vivant et il est déclaré vainqueur,
— Tous les joueurs sont éliminés et la partie n'a pas de vainqueur.

3.2 Succès 1 : Mémentos

L'implémentation du mémento a été conçue comme indépendante du reste, notamment de l'implémentation du joueur : toutes les fonctions liées à la gestion d'un mémento

ont été regroupées dans un fichier *deck.c*.

3.2.1 Nouvelle structure de données

Le mémento d'un joueur est un ensemble de cartes.

Dans l'implémentation choisie, le mémento est un tableau des cartes ainsi qu'un indice permettant de mémoriser la position de la prochaine carte à piocher dans ce tableau :

```
struct deck {
  const struct card * cards[DECK_SIZE];
  int index;
};
```

3.2.2 Génération aléatoire du mémento

Précédemment, la fonction *draw_random_card* s'occupant du tirage aléatoire de nouvelles cartes selon leurs probabilités de tirage respectives était appelée lors de la génération de la main du joueur dans la fonction *deal_hand*. Ce succès déplace le tirage aléatoire de nouvelles cartes dans le processus de génération du mémento d'un joueur, effectué dans la fonction *deal_deck*.

La fonction *deal_deck* réinitialise également l'indice du mémento.

3.2.3 Mélange

La fonction *draw_random_card* n'est utilisée qu'une seule fois dans la partie, au moment de l'initialisation des joueurs dans la fonction *play_game*. Cependant, une fois que toutes les cartes d'un mémento ont été piochées, il doit être remélangé : la fonction *shuffle_deck* s'occupe de mélanger un mémento.

L'algorithme choisi pour le mélange opère en parcourant une à une les cartes du mémento et en échangeant leur place avec celle d'une autre carte du mémento sélectionnée aléatoirement. Ainsi, chaque carte s'est trouvée déplacée au moins une fois à une place aléatoire : lors du parcours, la carte s'est soit trouvée déplacée avant que l'indice de parcours n'atteigne la position initiale de la carte dans le mémento, soit se trouve déplacée au moment où l'indice de parcours atteint sa position.

On remarquera que étant donnée une carte dans le mémento, la distribution de probabilité de son emplacement dans le mémento après le mélange n'est pas uniforme. Cependant, tous les emplacements autres que sa position initiale ont une égale probabilité : la petite irrégularité concernant la probabilité que la carte soit remise à sa position initiale n'est pas gênante pour notre mélange.

3.2.4 Pioche

L'essence même de l'existence du mémento est de pouvoir y piocher des cartes. Cette fonctionnalité est assurée par la fonction *draw_card* qui renvoie la carte désignée par l'indice du mémento et incrémente cet indice.

Dans le cas où l'indice atteint la fin du mémento, la fonction s'occupe de le mélanger via un appel à *shuffle_deck* avant de piocher la nouvelle carte. Ainsi, le mélange du mémento est complètement transparent pour son utilisateur : la fonction *draw_card* peut être utilisée sans avoir à se préoccuper d'un éventuel épuisement du mémento.

3.2.5 Modification de la distribution de la main

La fonction *deal_hand* ne doit plus à présent tirer une carte en utilisant la fonction *draw_random_card* mais plutôt la fonction *draw_card* du mémento.

3.3 Succès 2 : Plateau de jeu et caramels mous

3.3.1 Nouvelle structure de données

Ce succès apporte une nouvelle donnée, le plateau : un espace torique sur lequel les joueurs se déplacent. Il a été modélisé comme un tableau à deux dimensions de taille finie, dont l'utilisation ultérieure permettra le passage d'un bord à l'autre. En voici son implémentation :

```
char board[BOARD_SIZE][BOARD_SIZE];
```

Le plateau devient alors une variable globale pour le programme à laquelle toute les fonctions pourront accéder directement. Le choix du tableau de caractères vient du fait que chaque case possède un état libre ou occupé représenté par la valeur 0 ou 1. De plus, pour les fonctions de recherche de positions décrites plus tard, une structure de case contenant ses coordonnées dans *board* à été mis en place :

```
struct cell {
  int x;
  int y;
};
```

Enfin, les joueurs étant placés sur le plateau, ceci ont acquis deux nouveaux attributs dans leur structure : les coordonnées du joueur sur le plateau, sous la forme de deux entiers x et y. Les positions des joueurs sont initialisées par *play_game* via la fonction *init_board* qui donne une position aléatoire à chaque joueur sur le plateau et place toutes les autres cases dans l'état libre.

3.3.2 Une nouvelle carte : Stone

Cette nouvelle carte ayant un coût de 10 en mana permet de placer un caramel mou au contact (sur une case adjacente) de sa cible passant alors la case en question dans l'état occupé jusqu'à la fin de la partie. Stone choisit une case libre au contact de sa cible aléatoirement, si aucune case n'est libre la carte n'a pas d'autres effets que de consommer le mana du lanceur.

3.3.3 Modifications des mécaniques de bases

A présent, les joueurs adoptent une nouvelle phase au début d'un tour de jeu : la phase de déplacement.

Phase de déplacement

Chaque joueur choisit une case accessible qu'il peut atteindre (le déplacement par tour est limité à un nombre fini de franchissement de cases). Si plusieurs joueurs ont choisi la même case, ils restent à leur place initiale. Sinon le joueur se déplace à la case choisie. Cette phase est gérée par la fonction *move_players* qui est à présent appelée par la boucle principale de *play_game*. *move_players* réalise la sélection de chaque joueur aléatoirement à partir du tableau des positions atteignables généré pour chaque joueur par la fonction récursive *add_cells*. Cette fonction calcule les cases atteignables en se déplaçant d'une case à partir d'un tableau de positions de départ avant de recommencer à partir des cases nouvellement atteintes et ce jusqu'à avoir consommé tout le déplacement du joueur.

Nouvelle élimination possible

À présent, l'élimination peut également venir d'une impossibilité à se déplacer : cela signifie que si les quatre cases adjacentes à un joueur sont occupées, il est à présent éliminé. Ce comportement est géré par *move_players*, qui élimine les joueurs ne pouvant choisir aucune nouvelle position par un appel à la fonction *die*. De plus, les joueurs occupant une case du plateau, il a été choisi de ne pas libérer les cases des joueurs à leur élimination. Ainsi, les "cadavres" des joueurs éliminés servent d'obstacles pour le reste de la partie.

3.4 Succès 3 : Appel à un ami

3.4.1 Un nouveau type d'entité : les amis

Jusqu'à présent, les seules entités présentes sur le plateau étaient les joueurs (on ne considère pas les caramels mous comme des entités). L'introduction des amis vient chambouler l'organisation établie : il a fallut généraliser le concept de joueur au concept d'entité, permettant de distinguer deux types d'entités, les joueurs et les amis.

Les modifications suivantes ont donc été apportées :

— les fichiers *player.c* et *player.c* ont été renommés en respectivement *entity.c* et *entity.c*,
— la structure de donnée *player* a été renommée en *entity*,
— un champ *type* a été rajouté à la structure *entity* afin de pouvoir discriminer les joueurs et les amis,
— le tableau *players* a été renommé en *entities*,

— un nombre d'amis en vie *n_friends* vient compléter le nombre de joueurs en vie *n_players*, permettant de parcourir le tableau *entities* grâce au calcul du nombre d'entités en vie *n_players + n_friends*.

De plus, le plateau de jeu, le tableau des joueurs ainsi que sa taille étaient déclarés en tant que variables globales du programme : le nombre de ces variables augmentant encore avec ce succès, et l'ensemble de ces variables représentant le contexte du jeu, il a été choisi dans ce succès de placer toutes ces variables dans une structure *game_data*.

```
struct game_data {
    struct entity * entities[BOARD_SIZE * BOARD_SIZE];
    int n_players;
    int n_friends;
    char board[BOARD_SIZE][BOARD_SIZE];
};
```

Cela permet désormais de passer le contexte du jeu dans la plupart des fonctions, notamment celles responsables des différentes phases de jeu mais également dans les fonctions d'application des cartes, qui deviennent alors des fonctions pures.

Un ami a des caractéristiques et un comportement différent d'un joueur : il apparaît avec 1 point de vie, possède une nouvelle caractéristique appelée durée de vie et ne peut effectuer qu'une seule action par tour, se déplacer ou jouer sa carte. Lors de son invocation, une carte choisie aléatoirement lui est attribuée : elle sera la seule carte qu'il pourra jouer. De plus, on lui attribue également une cible, tirée aléatoirement parmi les entités en jeu : elle sera la cible de toutes les cartes jouées par l'ami.

Les amis ont leur propre algorithme de déplacement, implémenté dans la fonction *choose_friend_move*. Celui-ci consiste à se diriger tout droit vers sa cible. Pour cela, il dresse la liste de toutes les positions atteignables et choisit parmi ces positions celle qui le rapproche le plus de sa cible (on réutilise la fonction utilitaire *add_cells*, renommée dans ce succès avec le nom plus explicite *find_reachable_cells*). Les amis ont 2 points de déplacement.

La structure *player* se retrouve donc avec une partie de ses champs spécifiques à l'usage d'un ami - la cible, la carte et la durée de vie - et une autre partie réservée à l'usage d'un joueur - la main et le mémento.

3.4.2 Une nouvelle carte : Puppy

L'apparition des amis en jeu se fait grâce à une nouvelle carte : Puppy, ayant un coût de 5 mana. Elle a pour effet d'invoquer un ami à proximité du joueur : cet ami aura pour cible la cible de la carte.

3.4.3 Durée de vie et disparition des amis

Une nouvelle phase de jeu apparaît avec l'introduction d'une durée de vie pour les amis. Celle-ci diminue à chaque tour et entraîne la disparition de l'ami lorsqu'elle atteint 0. La fonction implémentant la décrémentation de la durée de vie de tous les amis est

nommée *decay_friends* et est appelée à la fin de chaque tour par la boucle de jeu (dans la fonction *play_game*). La fonction *purge_the_dead* a quant-à-elle comme nouveau rôle de faire disparaître du plateau et de l'état du jeu les amis dont la durée de vie est nulle.

4 Difficultés

Durant ce projet, plusieurs difficultés sont survenues au cours des différents succès à propos de détails techniques sur lesquelles il peut être intéressant de revenir.

4.1 Structure du code

Le code source du programme est séparé en plusieurs fichiers sources contenant différentes partie du code plus ou moins indépendantes :

- *board.c*, qui gère le plateau, son fonctionnement et la phase de déplacement
- *cards.c*, qui gère les cartes, leur définition et leur génération
- *deck.c*, qui gère le mémento et sa manipulation
- *entity.c*, qui gère les différentes entités et leur possible main
- *games.c*, qui gère les mécaniques de jeu globales et la boucle principale du jeu
- *utils.c*, qui gère différentes fonctions utilitaires comme l'aléatoire ou l'arithmétique

Cependant, la séparation du programme en corps élémentaires a engendré quelques problèmes de dépendance. Par exemple, les définitions des joueurs et des cartes se font référence mutuellement, le joueur étant défini avec au moins une carte et la carte étant définie avec un joueur comme cible, ce qui génére une référence circulaire. Pour y remédier, le fichier entête des cartes (*cards.h*) a une structure particulière : il définit tout d'abord le prototype de la structure joueur, puis contient la définition de la structure carte et enfin l'inclusion de *entity.h*.

D'autre part, la méthode de développement en succès consécutifs a demandé l'ajout de nouveaux fichiers sources au fur et à mesure des succès et le déplacement de certaines fonctions. Ainsi, la séparation du mémento des joueurs a rendu l'action de piocher dépendante du mémento et non du joueur, les mémentos étant à présent indépendants des joueurs. Ceci a forcé l'adoption une structure du code flexible et adaptée à recevoir des modifications à chaque nouveau succès, avec l'arrivée de nouvelles mécaniques ou structures.

4.2 Structures de données

L'arrivée de la structure de donnée *game_data* dans la dernière partie du projet est venue répondre aux difficultés apportées par la nécessité de modifier le nombre de joueurs par l'utilisation d'une carte (*puppy*). En effet, les cartes n'avait pas accès à cette variable auparavant et la multiplication de variables globales ne satisfaisait pas l'idée que *play_game* devrait se suffir à elle-même pour gérer une partie complète. L'apparition de *game_data* a demandé une révision en profondeur de tous les accès aux données du jeu dans l'intégralité des fonctions.

Dans le même ordre d'idée, la structure *cell* du deuxième succès est apparue afin de ne plus utiliser de tableaux bidimensionnels de taille variable, qui servaient, dans une première version du code, à stocker les coordonnées d'une case. En effet, les tableaux de structure sont plus simples à utiliser, à déclarer et à utiliser que ceux à deux dimensions.

Enfin, l'initialisation du tableau de pointeurs a posé problème pendant un temps puisque le programme ne générait pas plusieurs pointeurs différents lors de l'initialisation mais recopiait toujours le même pointeur dans toutes les cases du tableau. L'utilisation de *malloc* a alors réglé ce problème de manière efficace et n'a demandé qu'une légère modification de *die* pour libérer l'espace du joueur éliminé via un *free*.

4.3 L'élimination

La fonction *die* qui élimine un joueur possède un comportement particulier : le langage ne permettant pas de supprimer une case d'un tableau, et la taille du tableau étant toujours donnée avec celui-ci, il est possible de "supprimer" une case en diminuant la taille du tableau. Les fonctions utilisant ce tableau par la suite ignoreront alors sa dernière case. Le contenu de la dernière case n'a alors plus qu'à être recopiée dans la case à "supprimer".

Un dernier point délicat reste à aborder : la gestion du cas particulier dans lequel la cible d'un ami viendrait à mourir avant la mort ou la disparition de cet ami. Une vérification est donc faite au sein de la fonction *die* : lorsqu'une entité meurt, on fait également mourir tous les amis ayant pour cible cette entité.

Des difficultés ont été cependant rencontrées dûes au fonctionnement de la fonction *die* : en effet, celle-ci modifiant les indices des joueurs du tableau, ne peut pas s'appeler récursivement. Il a donc été choisi de descendre la durée de vie des amis à 1 afin de laisser la fonction *purge_the_dead* faire mourir les amis. et de les rendre inoffensifs durant l'interval de temps restant en leur attribuant une mana et une régénération de mana de 0.

5 Résultats

Pour chaque succès, des tests unitaires de chacune de ses fonctions ont été mis en place pour vérifier leurs comportements. De plus, l'état des différents joueurs en jeu est envoyé dans la sortie standard à chaque tour via la fonction *display_entities* ce qui permet de suivre l'éxecution du jeu. Enfin, il existe une fonction *display_board* permettant d'imprimer le plateau dans la sortie standard et donc d'avoir un retour sur le jeu.

Toutes les fonctions du programme ont passé leurs tests avec succès et le jeu est prêt à l'exécution dans chacun des succès. Pour la compilation du code ou son exécution, se réferrer au *README*.

6 Améliorations

Dans cette section se trouve plusieurs pistes d'améliorations éventuelles du jeu et du programme.

6.1 Le mémento

Il serait plus réaliste de revoir la structure du mémento de sorte à ce que, lorsque toutes les cartes du mémento ont été piochées et qu'il faut le remélanger, les cartes que le joueur possède encore en main ne soient pas remélangées. En effet, dans l'implémentation actuelle, les cartes en main ne sont pas retirées du mémento lors de son mélange. L'idée serait de retrouver le comportement d'un mémento réel, composé d'une pile de pioche et d'une défausse, et dont seules les cartes de la défausse sont mélangées pour reformer un mémento plein.

On pourrait par exemple sauvegarder dans le mémento un tableau des indices des cartes en main qui serait mis à jour à chaque fois que le joueur pioche, puis avec une légère modification de *shuffle_deck*, placer ces cartes en début de mémento, modifier l'attribut *index* en conséquence et ne mélanger que la fin du mémento (les cartes non en main). De cette façon, le joueur ne pourrait en aucun cas piocher une carte qu'il a déjà en main.

6.2 L'interactivité

Actuellement, le jeu n'est joué que par l'ordinateur. Il pourrait être intéressant de permettre à un ou plusieurs joueurs d'être controllés par des humains. Ainsi, la dimension de jeu serait apparente et permettrait de réaliser des parties entre humains.

Pour ce faire, il serait envisageable de créer un nouvel attribut *décision* : un pointeur vers une fonction permettant de réaliser la prise de décision avec l'intelligence artificielle aléatoire actuelle pour les non-humain et une interaction avec l'utilisateur pour les humains. Le joueur ayant connaissance de ses cartes, ses caractéristiques et pouvant voir le plateau via la fonction *display_board* pourrait donc renseigner sa prochaine position, la carte à jouer et sa cible via l'entrée standard.

6.3 L'intelligence artificielle

Dans le même ordre d'idée, l'ordinateur pourrait posséder une réelle intelligence artificielle avec la mise en place de stratégies. L'ordinateur éviterait notamment de se coller aux cases occupées lors de son déplacement pour éviter l'élimination par blocage. De plus, les cartes ne seraient plus jouées aléatoirement, une Panacea étant sans réel intérêt si le joueur a déjà plusieurs centaines de points de vie. La carte Hell is other est également à jouer immédiatement afin d'éliminer un adversaire très bien protégé. Cela permettrait d'ailleurs de tester l'efficacité de différentes stratégies en faisant s'affronter les différentes intelligences artificielles en tournois.

6.4 Interface graphique

Pour l'instant, l'interface pour observer le jeu est la console, il pourrait être intéressant d'obtenir une interface graphique dynamique mettant à jour le plateau, la liste des joueurs et leurs caractéristiques au fur et à mesure des tours. Ainsi, il serait également possible de cacher les mains des autres joueurs quand ça n'est pas leur tour, introduisant une dimension tactique supérieure ainsi qu'une prise en main plus rapide du jeu par un humain. L'interface pourrait également permettre d'obtenir des rappels sur les règles ou les effet des cartes via des boutons d'aide.

6.5 Classes et équipes

Pour augmenter encore l'esprit de jeu de stratégie, il pourrait être intéressant d'implémenter un mode en équipe où les joueurs pourraient jouer de nouvelles cartes pour aider leurs alliés. Par exemple, une Panacea qui se jouerait sur un allié plutôt que sur soi-même ou une carte permettant d'améliorer la génération de mana de l'équipe entière.

Dans un autre ordre d'idée, on pourrait implémenter différents types de joueurs (des *classes*) avec des caractéristiques initiales différentes et des mémentos spécialisés. Par exemple un joueur guerrier pourrait augmenter sa probabilité d'avoir des Razor alors que le joueur soigneur aurait plus de Panacea. Ainsi la dimension tactique du jeu serait complète et des intelligences artificielles spécifiques pour chacune des classes pourraient être développées.

7 Conclusion

Ainsi, nous avons détaillé les quatre succès, qui ont été réalisés dans le temps imparti. Les fonctions de ce projet ont été regroupées dans différents fichiers suivant la sémantique de leurs rôles, et des tests ont été écrit au fur et à mesure des succès afin de vérifier et d'assurer leur fonctionnement à travers les améliorations suivantes. Nous avons également abordé les difficultés rencontrées et les solutions mises en place pour les surmonter. Enfin, nous avons discuté d'améliorations possibles à apporter au projet.

Le sujet définissait trois succès supplémentaires, qui donnent également de nombreuses idées d'amélioration du jeu. Néanmoins, le quatrième succès demandait d'implémenter le comportement des cartes avec des pointeurs de fonction, choix qui avait été fait dès le début de notre projet. Ainsi, ce succès se trouve automatiquement réalisé bien que nous n'ayons pas ajouté de dossier *achievement4* ni de section relative dans ce rapport.