

PROJET DE C++ 2A

Réalisation d'un RPG Pokémon

MEILAC Adrien
LANGLOIS Romain

Encadrant :
Jean-Baptiste Yunès

SOMMAIRE

Introduction	3
I Description de l'utilisation du jeu	4
1 Présentation de l'interface de combat	4
2 La carte du jeu	6
II Architecture générale du code	8
1 Interaction des classes et des fonctions	8
2 Précisions sur l'interface graphique	9
III Architecture détaillée et problèmes rencontrés	10
1 Méthodes d'accès et de conversion des données (Tools)	10
2 Structures de données (Pokémon)	11
3 Structure du jeu (Battle)	14
4 Fonctions graphiques réalisées en C (Graphics)	15
Conclusion	17

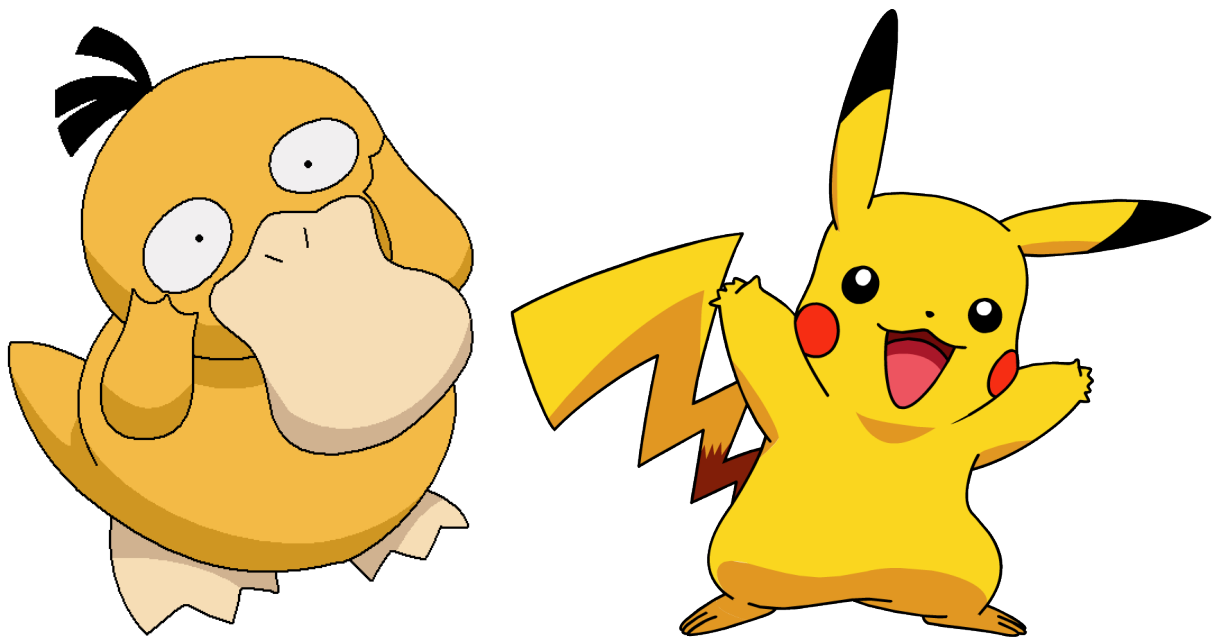
INTRODUCTION

Nous avons depuis notre enfance été plongés dans le monde des RPG. Si nos goûts spécifiques ont pu varier, nous partageons néanmoins une passion toute particulière pour ces jeux souvent inimitables. Nous avons ainsi dans le cadre de ce projet décidé de la faire partager en reprenant un RPG incontournable, Pokémon.

Pokémon a été créé en 1996 par Satoshi Tajiri. Cette franchise a réalisé des records de ventes dans l'histoire des jeux vidéos dès ses premières éditions Rouge et Bleu, vendues à plus de 30 millions d'exemplaires. Aujourd'hui, Pokémon est exploité sous forme d'animés, de mangas, de jeux de cartes et plus récemment d'application mobile avec Pokémon Go et génère plus de 3.3 milliard de chiffre d'affaires (année 2016).

Nous avons entrepris de coder notre RPG en C++ avec une interface graphique en C. Nous avons également utilisé en Python pour pouvoir créer des bases de données propres à partir de données brutes. Nous avons choisi d'utiliser la bibliothèque SDL (associée avec SDL Image et SDL ttf) pour l'interface graphique car cette dernière contient des fonctions adaptées à la création de jeux de plateforme.

La réalisation de ce jeu représentait un immense défi que nous avons décidé de relever. Notre projet Pokémon est une version épurée qui s'inspire à l'original mais qui rappellera des souvenirs aux connaisseurs.



I DESCRIPTION DE L'UTILISATION DU JEU

Pour pouvoir rester au plus proche du jeu original, nous avons utilisé des bases de données créées par les fans sur le site de Pokémon essentials¹. Nous en avons extrait divers éléments graphiques ainsi que des polices de caractère et des fichiers contenant de vraies données sur les Pokémon. C'est pourquoi notre interface graphique et celle du vrai jeu se ressemblent beaucoup.

1 Présentation de l'interface de combat

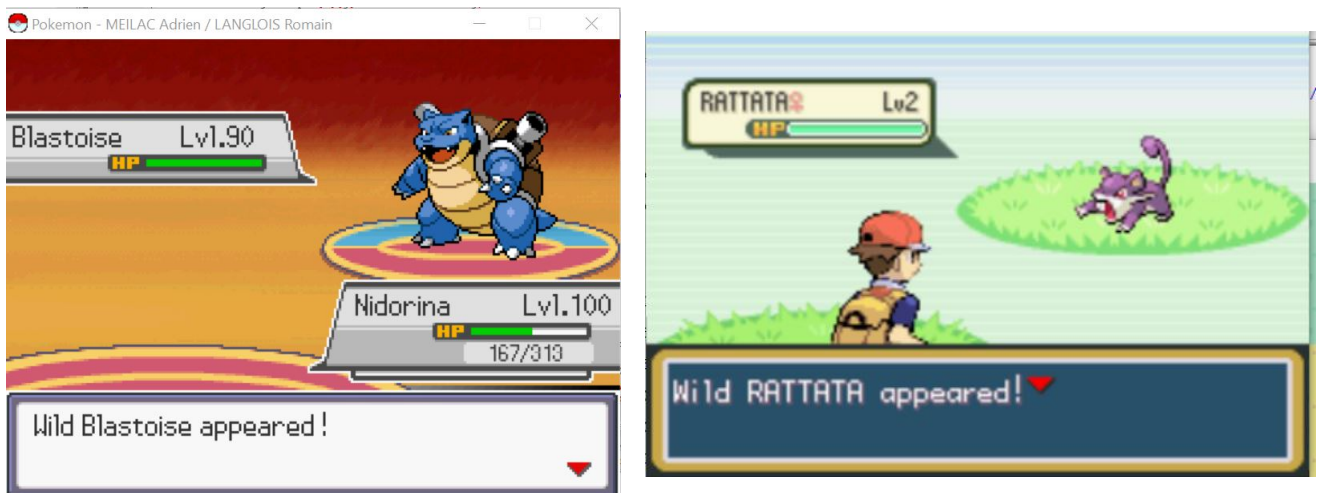


FIG. 1 : Début de combat

Lors de la rencontre d'un pokémon aléatoire, une interface spécifique apparaît après une transition. On peut voir qu'il y a arrière-plan de combat et également le Pokémon ennemi. Une pression de la touche entrée permet alors d'afficher le Pokémon allié et le menu contextuel de combat.

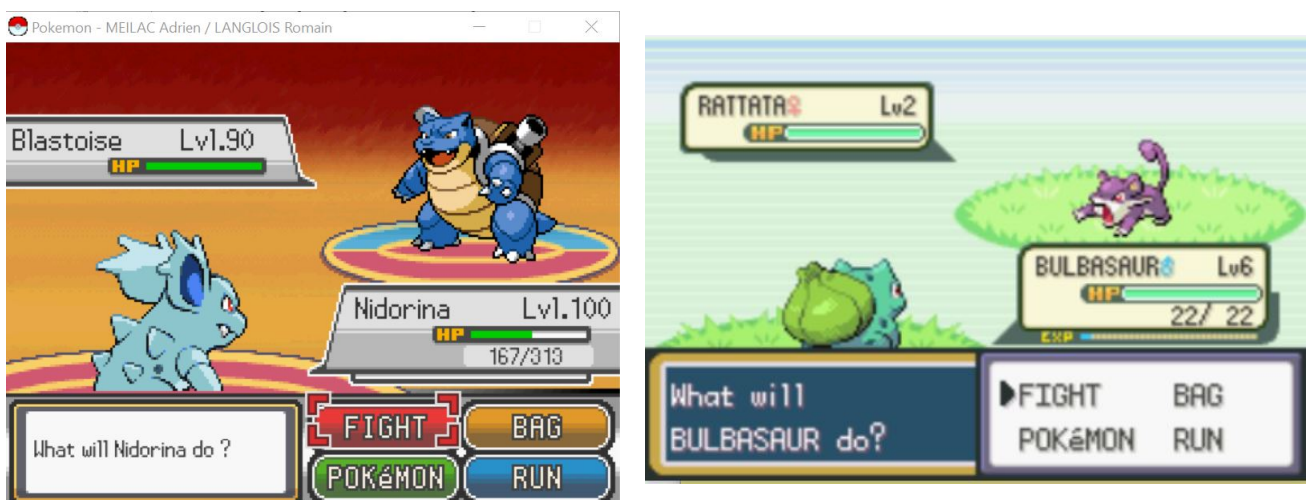


FIG. 2 : Menu principal

Dans ce menu, comme dans le jeu original, le joueur a le choix entre accéder à son inventaire (BAG), choisir son pokémon destiné à combattre (Pokémon), s'enfuir (RUN), ou alors attaquer (FIGHT). Si le joueur décide de s'enfuir, il revient sur la map générale et ses Pokémon gardent les dégâts qu'ils

1. http://Pokémonessentials.wikia.com/wiki/Pok%C3%A9mon_Essentials_Wiki

ont subis lors du combat. Cependant, nous n'avons pas eu le temps de conserver la position du joueur lors du lancement du combat, c'est pourquoi ce dernier est remis à sa position initiale.

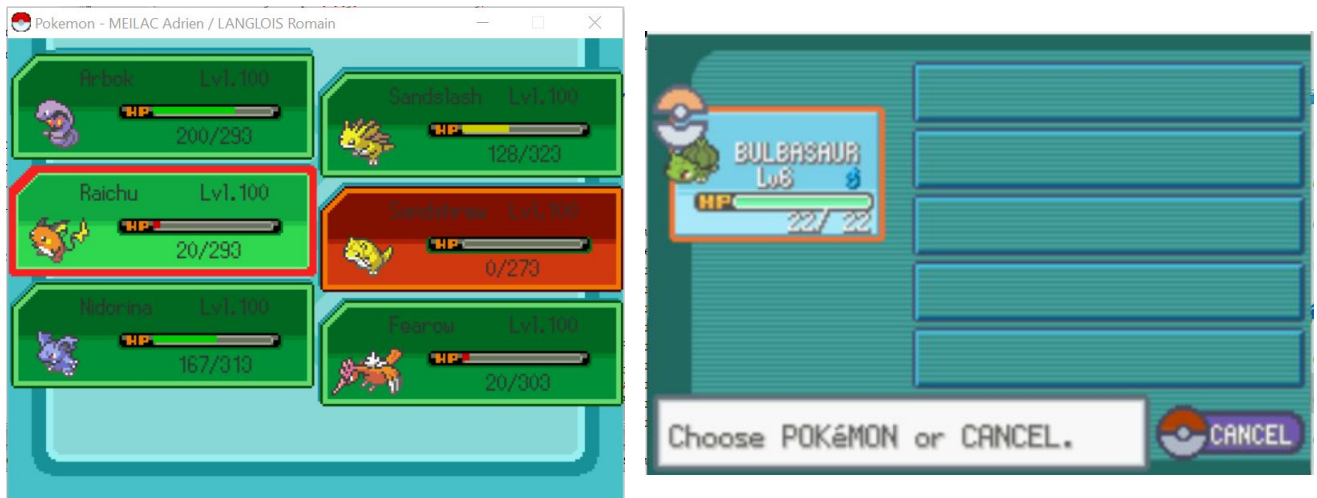


FIG. 3 : Menu du dresseur

Le menu Pokémon, appelé menu du dresseur, permet au joueur de choisir le Pokémon qui va combattre, et de voir l'état de santé ainsi que le niveau de ses Pokémon. Les Pokémon morts apparaissent en rouge et ne peuvent être sélectionnés pour combattre.

Nous avons réalisé un effet visuel sur les barres d'HP. Leur longueur est ainsi adaptée au pourcentage de leur vie restant, et la couleur change : rouge en-dessous de 20%, orange entre 20 et 50%, et vert au-delà. Cet effet est aussi visible sur l'écran de combat principal, ce qui rend le combat plus interactif.

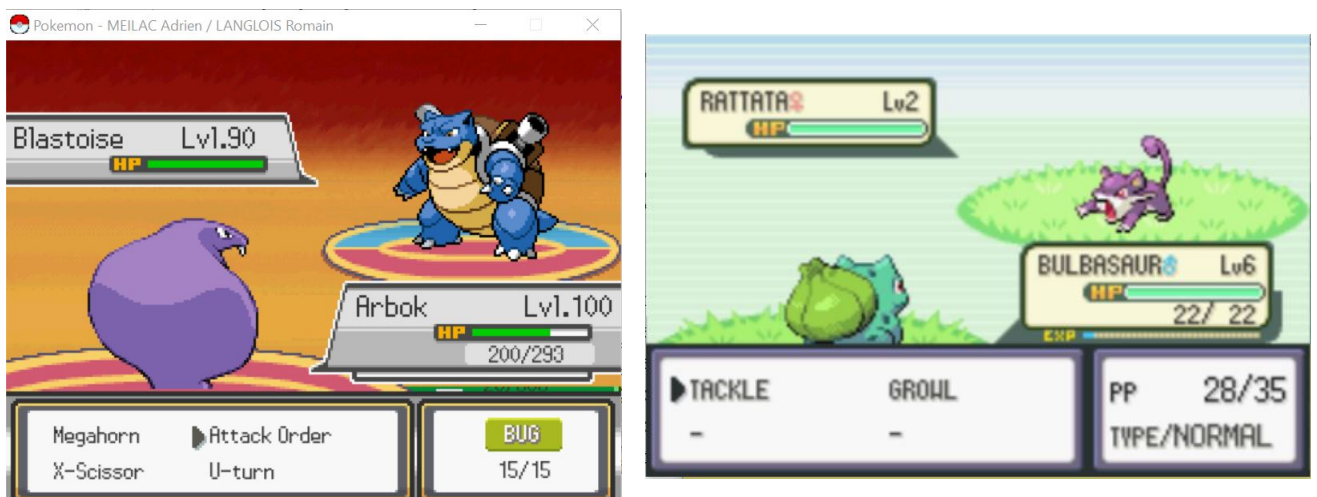


FIG. 4 : Menu de combat

Si le combat est adopté, un menu comprenant les diverses capacités disponibles pour le Pokémon retenu apparaît. Lorsque le joueur sélectionne une capacité à l'aide du curseur, il peut voir son type ainsi que ses Points de puissance (PP). Le combat se déroule ensuite au tour par tour en respectant les priorités d'ordre d'attaque entre les Pokémon. A chaque étape le menu est remplacé par une fenêtre de dialogue indiquant l'attaque utilisée. On voit alors le Pokémon touché perdre ses points de vie progressivement. Certaines capacités, comme Hail (Grêle dans le jeu français) de Nidorina, ne font pas de dégâts à l'adversaire mais ont un effet sur le temps du terrain, qui va se répercuter durant plusieurs tours à l'aide d'un message dans la boîte de dialogue.

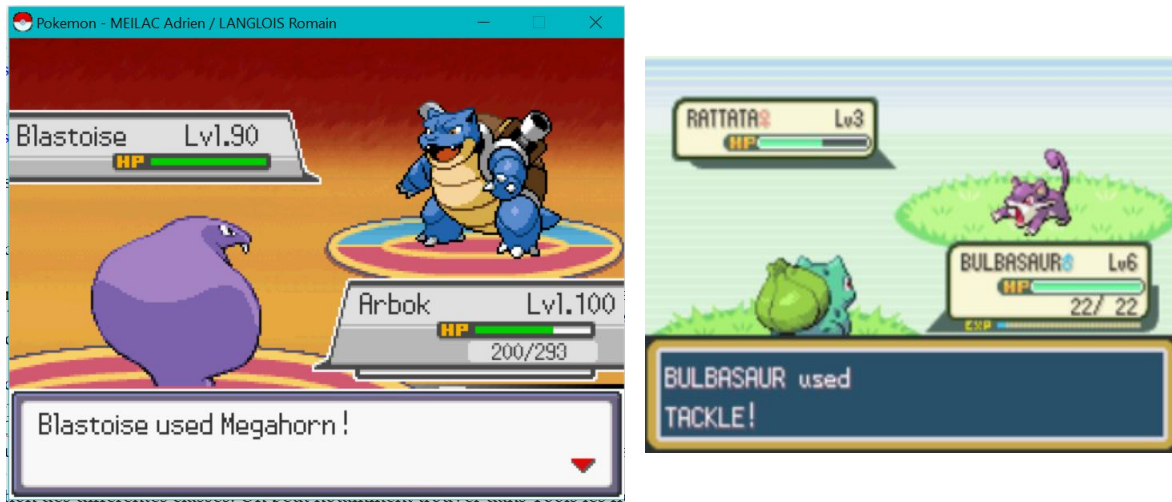


FIG. 5 : Joueur sur la carte Pokémon

Certains des Pokémon meurent ou terrassent leur adversaire en un seul tour. C'est un choix volontaire de notre part, nous avons pensé que la démonstration serait plus fluide s'il y avait possibilité d'observer directement les conséquences d'une victoire ou d'une défaite. Ces paramètres sont modifiables dans le fichier `Player_Pokémon.txt`.

Le pokémon sélectionné par défaut, Nidorina, possède approximativement les stats de son adversaire. Le Pokémon Arbok a été configuré avec des statistiques largement surévalué pour tester les cas de victoires et les autres Pokémon ont des statistiques largement sous évalué pour tester les cas d'échec. Dans le cas d'une victoire, le Pokémon vainqueur reçoit des points d'expérience (exp) dans une fenêtre contextuelle, puis le joueur revient sur la carte. Dans le cas où Pokémon du joueur est terrassé, ce dernier se voit contraint de sélectionner un autre Pokémon dans le menu de dressage pour pouvoir continuer le combat. Si tous les Pokémon décèdent, une boîte de dialogue Game Over apparaît, il convient alors de fermer le jeu pour le relancer.

2 La carte du jeu

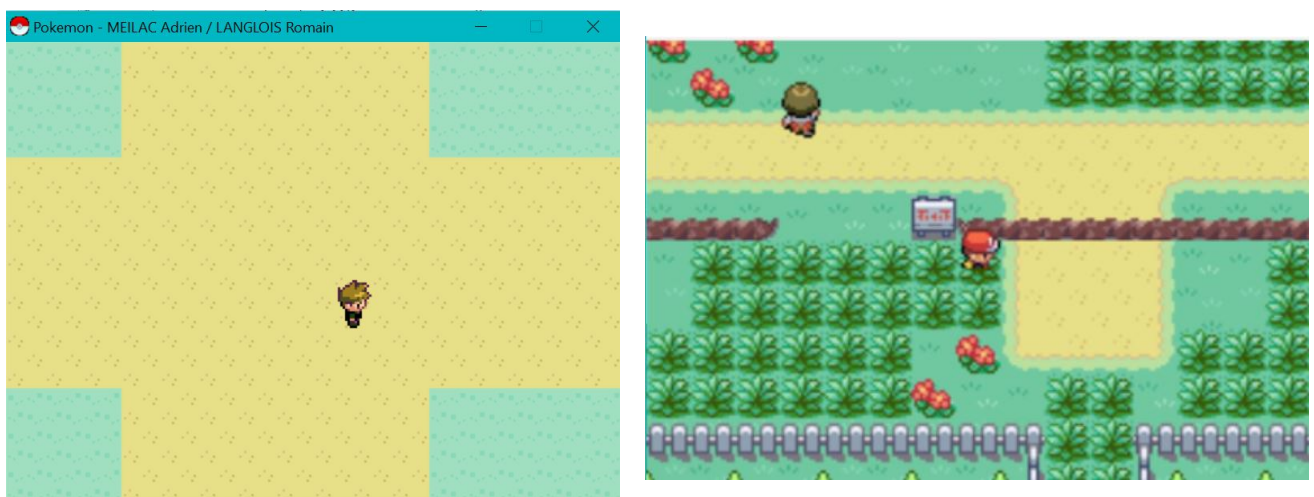


FIG. 6 : Joueur sur la carte Pokémon

Le joueur est également libre de se déplacer sur la carte quand il n'est pas en combat. La pression de la touche Espace lui permet d'afficher un menu. Il peut par exemple afficher ses différents Pokémon disponibles ou alors revenir sur la carte à l'aide du bouton Exit. Une courte description de chaque option est également disponible.

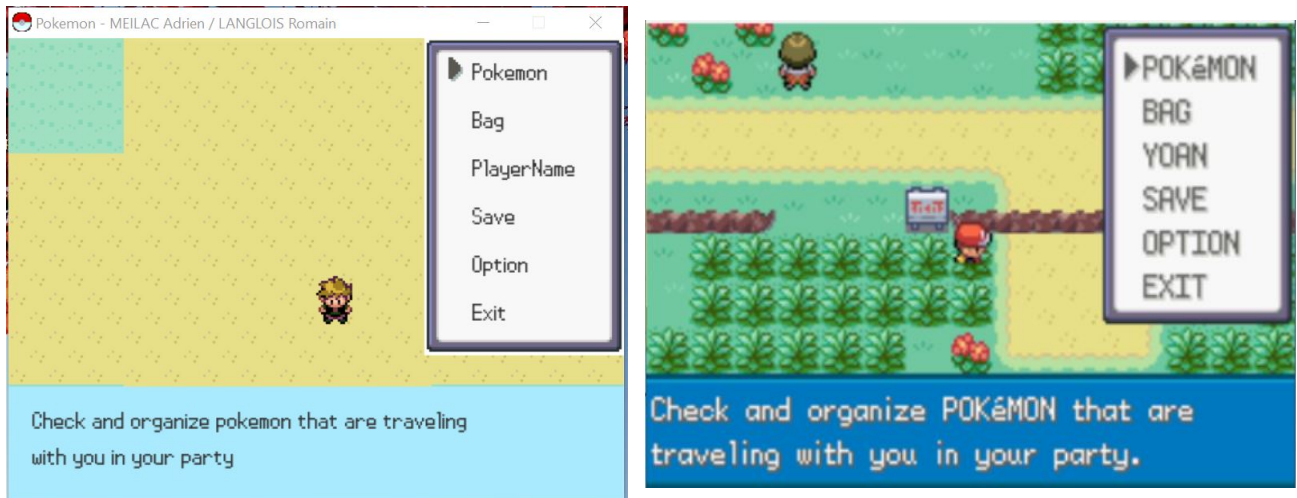


FIG. 7 : Menu sur le terrain

Nous n'avons pas réalisé l'inventaire ni les autres options de ce menu du fait d'un manque de temps, le code existant ayant déjà nécessité beaucoup de lignes, mais nous disposons désormais de tous les outils nécessaires pour réaliser ces améliorations. Pour la même raison, nous n'avons laissé qu'une seule couche d'arrière-plan sur la carte et n'avons pas eu le temps de gérer les déplacements de la carte, ni la liaison entre la carte et des lieux stockés dans les données, ni l'affichage des bâtiments.

II ARCHITECTURE GÉNÉRALE DU CODE

1 Interaction des classes et des fonctions

D'après GitHub, notre code est composé de 35 fichier .cpp et de 23 fichier .h, et notre projet contient environ 14 000 lignes. Pour ne pas se perdre dans notre projet, nous avons donc essayé de hiérarchiser au maximum notre code et de créer le namespace PKMN pour pouvoir reconnaître plus facilement nos fonctions.

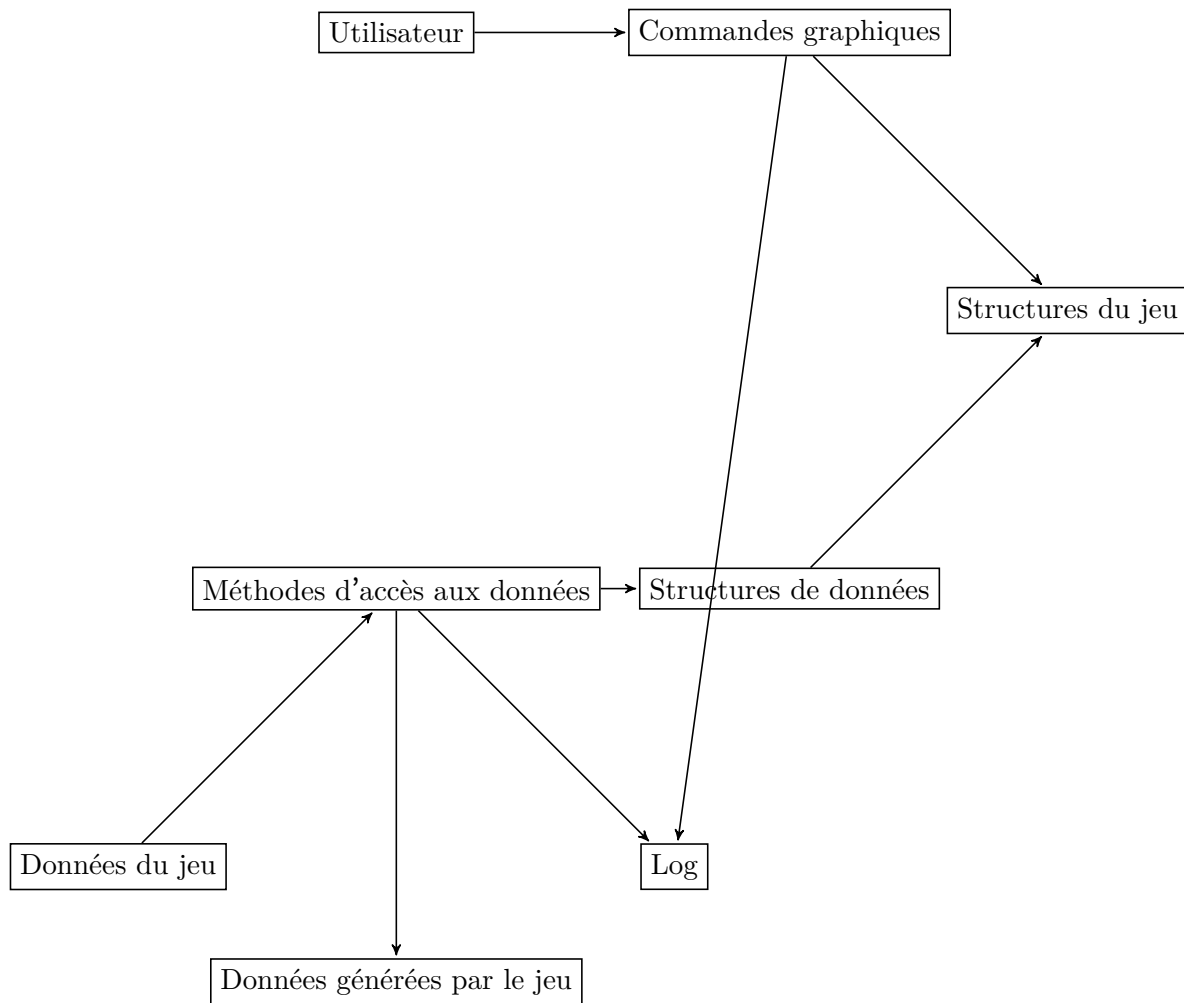


FIG. 8 : Schéma de l'architecture générale du projet

L'organigramme 8 illustre schématiquement les interactions entre les différentes parties de notre code. Lorsque le programme est lancé par l'utilisateur. Ce dernier va commencer par préparer les méthodes d'accès aux données (stockées dans le répertoire Tools) et va remplir la structure de nos données (la plupart des classes que nous avons écrites se trouvent dans Pokémon) avec les données stockées dans les répertoires Data et BackUp.

Ces structures de données sont majoritairement composées d'accesseurs et permettent de modifier les données et de les sauvegarder via les méthodes d'accès. Nous avons essayé de les différencier des structures du jeu (qui correspond principalement aux fichiers BattleWildPokémon et main). Ces dernières utilisent les structures de données mais n'agissent pas directement sur les fichiers.

De l'autre côté, l'utilisateur a uniquement accès aux commandes graphiques (stockées dans Graphics)

et le jeu en lui même est composé de boucles infinies qui s'arrêtent lorsque l'utilisateur ferme la fenêtre. Enfin notre programme rédige un log (stocké dans stdout.txt) qui nous permet de tracer les incohérences dans le gameplay.

2 Précisions sur l'interface graphique

L'interface graphique a été intégralement réalisée en SDL. Nous avons décidé de la séparer de la structure du jeu pour :

- éviter d'avoir des fonctions trop longues,
- distinguer le cœur de notre code qui est en C++ alors que les fonctions graphiques sont en C
- pouvoir utiliser les options avancées du C++ (telles que les classes) qui sont incompatibles avec l'option extern C.
- avoir des fonctions graphiques qui ne touchent pas aux données et réalisent juste de l'affichage

Pour pouvoir réaliser une telle coupure, nous avons créé des fonctions graphiques qui prennent l'écran en paramètre et qui le remplissent. Elles fonctionnent avec des boucles infinies qui vont s'arrêter (soit parce que l'utilisateur a sélectionné un choix, soit spontanément dans le cas d'un effet aléatoire) et renvoyer un "flag" (codé sous forme de string ici) aux structures du jeu qui vont agir en fonction du flag reçu. Nous nous sommes servis du double buffering pour que l'utilisateur ne puisse pas voir les jointures entre nos différents affichages.

L'installation de SDL a été une difficulté à surmonter car les fonctions changent entre les différentes versions de cette bibliothèque. Nous avons préféré installer la version 1.2 qui était la plus documentée. Nous avons installé en plus les librairies SDL_Images pour la gestion des images compressées (jpeg, png,...), et SDL_TTF pour la gestion des polices de caractères (utilisée pour charger les polices de caractères de Pokémon dans nos divers menus et boîte de dialogues).

Cela nous a appris d'apprendre à gérer l'ajout de bibliothèques dans les options du linker. Pour pouvoir faire fonctionner notre code nous avons en effet du rajouter les options -mingw32 -lSDL -lSDLmain -lSDL_Image -lSDL_Ttf. Nous avons aussi découvert l'utilisation des flag pour le compilateur. Nous avons utilisé l'option -std=c++11 pour écrire en C++ 11 ainsi que l'option -m32 qui permet de réaliser des applications 32 bits étant donné que nous utilisons la version 32 bit de SDL. Il ne nous a pas paru nécessaire d'utiliser l'option -O2 pour rendre l'exécutable plus rapide car notre jeu ne prenait quasiment pas de temps de chargement.

Nous n'avons cependant pas réussi à installer la bibliothèque FMOD et à l'utiliser car nous n'avons pas trouvé de bon tutoriel sur le sujet et nous n'arrivons pas à résoudre nos erreurs liés à des problèmes de versions.

III ARCHITECTURE DÉTAILLÉE ET PROBLÈMES RENCONTRÉS

Cette partie rentre plus en détail dans la manière dont notre code fonctionne et explique sa répartition dans les différents répertoires.

1 Méthodes d'accès et de conversion des données (Tools)

Tools contient tous les outils dont nous avons eu besoin lors de notre projet, qui ne sont pas directement relié au mécanisme de fonctionnement de notre jeu, mais qui permettent de simplifier la création des différentes classes. On peut notamment trouver dans Tools les fichiers (et leur header) suivants :

- **Table** : La totalité de nos fichiers sont des tableaux (avec comme délimiteurs ";""). C'est pourquoi il nous a paru essentiel de simplifier la lecture des fichiers en créant une classe qui les lirait. Cette classe est capable d'extraire une ligne, une colonne et surtout de donner un élément à partir de son nom de ligne et de son nom de colonne (surcharge de l'opérateur () car [] ne peut être surchargé qu'avec un argument). Afin de standardiser la saisie, nous avons décidé que la lecture des tableaux donnerait uniquement des chaînes de caractères (et ne chercherait donc pas à donner des nombres). Cette classe nous a posé quelques problèmes lorsque nous avons essayé de l'utiliser en écriture.

```
InternalName;Name;Weaknesses;Immunities;Resistances;IsSpecialType
NORMAL;Normal;;FIGHTING,GHOST;;False
FIGHTING;Fighting;FLYING,PSYCHIC;;ROCK,BUG,DARK;False
FLYING;Flying;ROCK,ELECTRIC,ICE;GROUND;FIGHTING,BUG,GRASS;False
POISON;Poison;GROUND,PSYCHIC;;FIGHTING,POISON,BUG,GRASS;False
GROUND;Ground;WATER,GRASS,ICE;ELECTRIC;POISON,ROCK;False
```

FIG. 9 : Exemple de fichier lu par Table

- **VectorMethod** : Ce fichier contient des fonctions supplémentaires concernant les vecteurs. Notamment, un test `vector_in` qui permet de dire si un élément est dans le vecteur, et une fonction qui permet de découper les chaînes de caractères. Cette dernière sert par exemple lors de la création des objets de type Table.
- **Conversion** contient une collection de fonctions permettant de convertir les objets d'un type à un autre. En effet, le compilateur MinGW fourni par notre IDE CodeBlocks ne nous laissait pas utiliser les fonction `stoi`, `atoi`, `stol` ... qui permettent la conversion directe des types. Nous avons donc créé nos propres fonctions. Ces dernières sont généralement utilisées après la lecture des fichiers contenant des données pour convertir les `std::string` que la classe Table permet d'extraire (et donc de compenser le fait que Table ne soit composé que de `std::string`)
- **StatSet** et **StatSetExt** sont des classes (la seconde héritant de la première) qui permettent de stocker les différentes statistiques de nos Pokémon. Lors de leur création, nous nous sommes aperçus qu'il y avait beaucoup de répétitions d'arguments liés aux points de vie, d'attaque, de défense, d'attaque spéciale, de défense spéciale et de vitesse. Nous avons aussi mis dedans la formule permettant de calculer les statistiques d'un Pokémon.

Si on note

- **Niv**, le niveau d'un Pokémon,
- **PV** sa vie maximum,
- **Stat** la valeur de ses autres statistiques (Attaque Défense, Attaque Spéciale, Défense Spéciale, Vitesse) (cette valeur est celle qui est valable lorsque le Pokémon est soigné),
- **IV** des paramètres pour chacune des statistiques (ces derniers varient entre 0 et 31, ils sont générés aléatoirement lors de la création d'un Pokémon, ils ne sont ni visibles ni modifiables),
- **EV** des paramètres pour chacune des statistiques (ces derniers varient entre 0 et 255, ils valent 0 lors de la création d'un Pokémon et varient en fonction des combats effectués, ils ne sont pas visibles par l'utilisateur, mais ce dernier peut les modifier indirectement),
- **Base** des paramètres pour chacune des statistiques traduisent l'influence de l'espèce sur un Pokémon,

alors les statistiques d'un Pokémon sont définies par :

$$Stat = \left\lfloor \frac{2 * Base + IV + \left\lfloor \frac{EV}{4} \right\rfloor * Niv + 5}{100} * Nat \right\rfloor$$

$$PV = \left\lfloor \frac{2 * Base + IV + \left\lfloor \frac{EV}{4} \right\rfloor * Niv}{100} \right\rfloor + Niv + 10$$

2 Structures de données (Pokémon)

Pokémon est un répertoire qui contient l'architecture des classes que nous remplissons lorsque le programme est lancé. La plupart des classes ont un constructeur qui prend en paramètre un nom interne d'objet dont les caractéristiques sont définies dans un ou plusieurs fichiers. Le constructeur va alors utiliser la fonction Table pour lire la ligne du fichier qui l'intéresse et construire l'objet, ce qui simplifie énormément la déclaration des classes. Nous avons fait notre code de telle sorte que les arguments des uns soient les noms internes des autres ce qui permet de créer automatiquement les objets associés.

Par ailleurs, la plupart des objets ont une structure qui n'est pas aléatoire et qui est totalement déterminée par le nom interne de l'objet, nous avons donc surchargé les opérateurs == entre un objet et un std::string pour simplifier les déclarations d'égalité et rendre le code plus lisible (cela permet d'écrire par exemple if(type == "FIRE") ...)

Voici une description des différentes classes :

- **Type** contient la classe codant le Type des Pokémon ou des attaques. Les méthodes les plus importantes de cette classe sont effectiveness qui donne l'efficacité d'un Type sur un autre (par

exemple, une attaque de type Eau sera très efficace contre un Pokémon de Type Feu), ainsi que `getPathImage` qui permet de donner l'image à afficher pour indiquer le Type d'un Pokémon.

- **Move** contient la classe codant les attaques des Pokémon. Elle est liée au fichier "Data/Move.txt" qui contient une grande quantité d'arguments, parfois très complexes. Nous n'avons donc pas tout utilisé. Move contient des objets issus des classes Flag, Target et DamageCategory qui correspondent à des effets particuliers liés aux attaques. Ces classes offrent une gamme de test booléen pour éviter d'avoir à gérer des notations abstraites.
- **Species** contient les arguments liés à l'espèce d'un Pokémon. Nous avons décidé de sauvegarder les mouvements que peut apprendre un Pokémon (en moyenne, une centaine) sous forme de string pour ne pas utiliser inutilement de la mémoire étant donné que seuls 4 sont utilisés (et sont donc converti en Move dans la classe fille Pokémon).
- **Pokémon** contient principalement les fonctions qui permettent de faire varier les statistiques des Pokémon. On lui associe 2 Types, 4 Attaques, ainsi que deux structures de statistiques, l'une étant l'état normal, et l'autre l'état actuel (En effet, si un Pokémon est blessé, il n'a pas sa vie au maximum, cette information est stockée dans l'état actuel. Toutefois, si il est soigné, il revient à son état normal. Cet état normal peut aussi jouer dans le calcul des dégâts, d'où l'intérêt de le sauvegarder comme un argument).

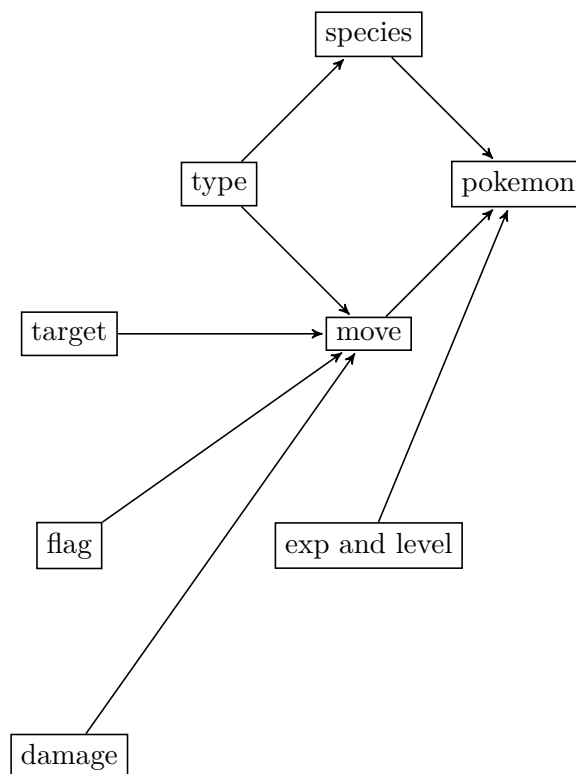


FIG. 10 : Hiérarchie des classes de données

Les formules pour calculer les dégâts dans le vrai Pokémon sont assez compliqués :

Si on note :

- Att la statistique d'attaque si le Move est physique, l'attaque spéciale si l'attaque est spéciale

- Def la statistique de défense si le Move est physique, l'attaque spéciale si l'attaque est spéciale
- Power le pouvoir de base (statistique lié à Move)
- Weather l'effet du temps
- Badge l'effet lié aux badges de l'utilisateur
- Critical l'effet bonus aléatoire
- random , un nombre entre 0.85 et 1
- STAB, l'effet bonus si le Pokémon est du même type que son attaque
- Type, l'effet du type de l'attaque sur le Pokémon

alors

$$Damage = \left\lceil \left(\left(\frac{2 * Niv}{5} + 2 \right) * Power * \frac{Att}{Def} + 2 \right) * Modifier \right\rceil$$

avec

$$Move = Targets * Weather * Badge * Critical * random * STAB * Type * Other$$

nous avons donc choisi de faire des simplifications car il nous était impossible de coder tout ces effets qui ont été rajoutés dans les différentes versions du jeu depuis plusieurs décennies.

3 Structure du jeu (Battle)

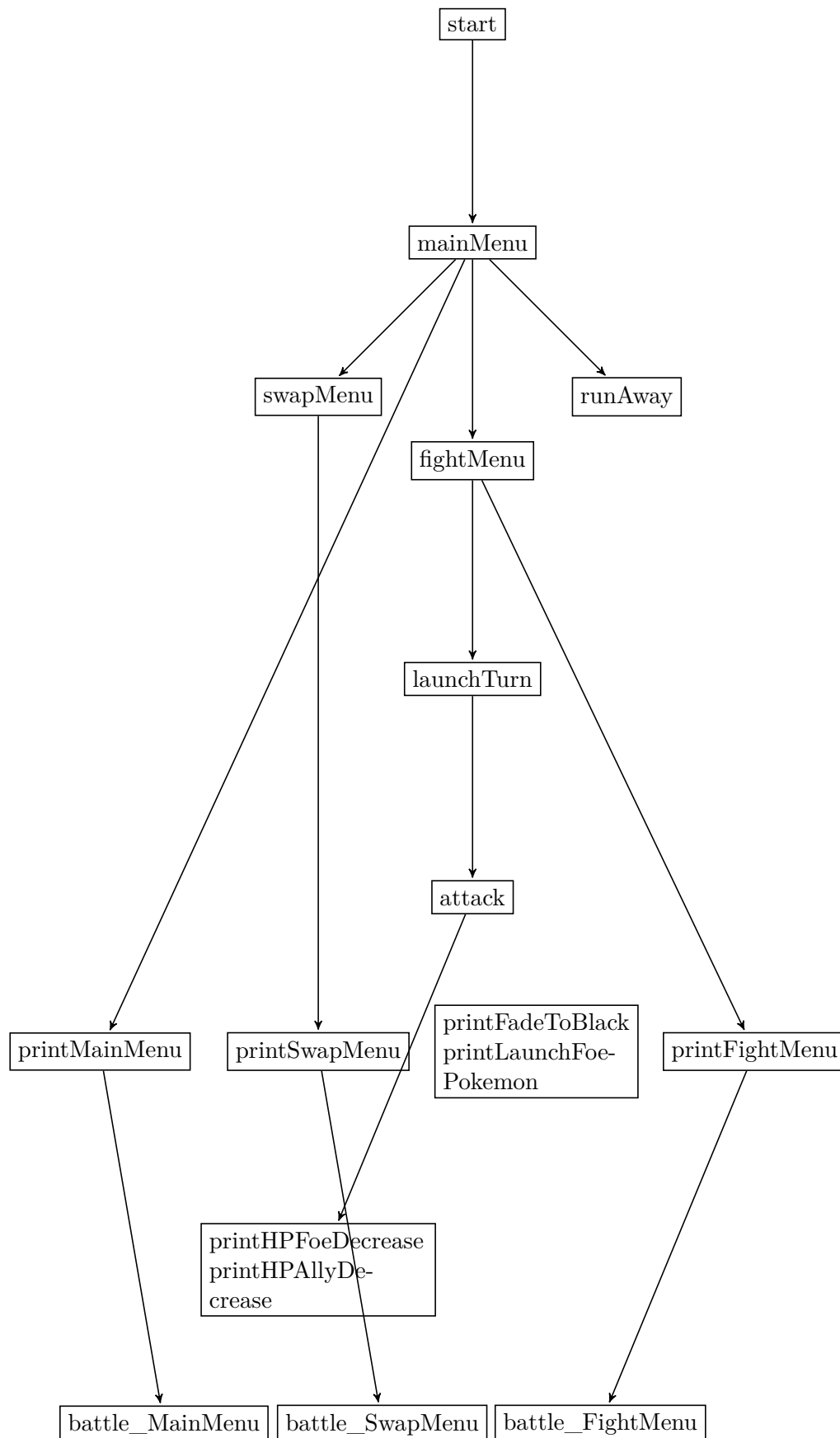


FIG. 11 : Hiérarchie des classes de données

Battle est un répertoire qui contient les fonctions liées aux fonctionnements des combats. Nous avons choisi de séparer les effets visuels (stockés dans Graphics, ce sont des fonctions dont le nom commence par Battle_) du système du jeu en lui même afin de créer des classes de combats qu'on lance via la méthode start et qui sont indépendantes. Nous avons préféré éviter d'avoir des codes trop longs avec des fenêtres graphiques car le mécanisme interne des combats est complexe et cela aurait été une source d'erreur.

A l'intérieur de ce répertoire, nous avons créé deux classes :

- **Player** qui est lié au joueur. Il contient notamment des objets de type Pokémon. Ses caractéristiques sont décrites dans le fichier Player du répertoire BackUp et elles de ses Pokémon dans le fichier Player_Pokemon. Ainsi, si l'on souhaite directement tester des configurations précises ou équilibrer le jeu il n'y a qu'à changer les statistiques dans ces fichiers, les Pokémon sélectionnés ou leurs capacités. Une des méthodes importantes de la classe Player est SwapPokémon, qui permet de changer le Pokémon retenu pour le combat.
- **BattleWildPokémon** est la classe dont l'architecture est décrite par l'organigramme 11. Nous avons décidé de scinder en 3 types les méthodes de cette classe :
 - Les méthodes graphiques commencent par le mot "print" et sont l'équivalent des fonctions graphiques commençant par le mot "Battle_", elles permettent de restreindre les interactions entre la classe et le domaine global.
 - Les méthodes structurelles qui font des choix à partir des flags qu'elle reçoivent des méthodes graphiques qui elles même transfèrent les flags des interfaces de Graphics.
 - Les méthodes de calcul qui ne modifient pas les arguments et qui sont appelées par les méthodes structurelles.

Prenons un exemple pour être plus explicite : Si on utilise la fonction mainMenu, cette dernière fait appel à printMainMenu qui fait appel à Battle_MainMenu. L'utilisateur voit alors le menu et s'il fait un choix à l'aide du bouton entrée, Battle_MainMenu s'arrête et renvoie un std::string contenant son choix à printMainMenu, qui le renvoie à mainMenu. mainMenu va alors choisir l'action suivante en fonction du flag qu'elle aura reçu. Dans cet exemple les actions sont choisir un nouveau Pokémon et faire appel au swapMenu, s'enfuir en appelant runAway ou combattre en appelant fightMenu.

Toutes ses méthodes sont cachées à l'utilisateur (portée private) sauf start qui permet de lancer le combat.

4 Fonctions graphiques réalisées en C (Graphics)

Graphics est un répertoire contenant toutes les fonctions graphiques gérant l'interface. Nous avons tenu à bien segmenter le code et à créer des fonctions qui se contentent d'afficher mais ne touchent pas à nos arguments. Nous avons fait seulement deux entorses à cette règle par souci de simplicité. Il s'agit de hpallydecrease et hpfoedecrease qui impactent respectivement la barre de vie du Pokémon allié et celle du Pokémon ennemi en combat.

Nous avons distingué deux principales natures de graphiques :

- Field (lié à Field.h), qui permet de générer la carte du jeu sur laquelle se déplace le personnage. Nous avons codé le déplacement du personnage et la rencontre de Pokémon sauvages. A l'aide de la touche Espace, le joueur peut afficher un menu lui permettant notamment de consulter ses Pokémon.
- les interfaces de combat (liées à Battle.h), comprenant l'écran de combat et les menus associés. Elles sont créées à l'aide de macro `#define` permettant de redéfinir sur chacune de nos fonctions graphiques les mêmes éléments à quelques variations près. Pour créer une interface de combat, il faut afficher l'arrière plan, la base du Pokémon ennemi, celle du Pokémon allié, le Pokémon ennemi, le Pokémon allié, la barre de menus présente en bas, les deux boîtes de data contenant la vie des Pokémon, ainsi que leurs nom, leurs niveau (Level), l'image de leurs barre de vie, et enfin la vie sous forme d'un entier pour le Pokémon allié. L'utilisation de `#define` nous a donc permis de condenser la déclaration de ces fonctions et de réduire drastiquement leur nombre de lignes. Par exemple, `SET_BATTLE_DATABOX` définit 11 paramètres en une seule ligne.

Pour pouvoir enchaîner les effets visuels, nous avons eu recours au double buffering. En effet, nos fonctions graphiques prennent en paramètre `screen` qui correspond à l'écran global. Elles vont l'utiliser jusqu'au moment où un flag est émis. La fonction graphique suivante reprend alors la variable `screen` et la réutilise. Le double buffering permet de ne pas montrer le nouvel appel aux images de fond et l'utilisateur ne voit aucun changement visuel.

Toutes nos fonctions graphiques n'ont pas de rétroaction, c'est pourquoi la majorité sont de type `void`. D'autre part, celles qui renvoient un flag (les fonctions graphiques non `void`), peuvent le faire de manière souhaitée (l'utilisateur clique sur une option) ou de manière involontaire (déclenchement d'un combat à l'aide d'un événement aléatoire, comme les combats)

CONCLUSION

Dans le cadre de ce projet, nous avons eu l'opportunité d'approfondir notre connaissance du C++ tout en travaillant sur un sujet nous tenant à cœur. Nous avons également exploité les possibilités d'interactions avec d'autres langages, en particulier le C.

Nous avons pu mettre en pratique et approfondir toutes les notions expliquées en cours, allant de la simple création d'une variable à l'usage poussé de classes abstraites.

Notre programme est encore une ébauche d'un RPG Pokémon et nous nous sommes rendu compte que la réalisation d'un tel projet en entier demandait plus de temps, mais n'était pas pour autant plus complexe.

