École nationale
de la statistique
et de l'administration
économique

université
PARIS-SACLAY

Projet de C++ 2A

Réalisation d'un RPG Pokemon

MEILAC Adrien
Langlois Romain

Encadrant :
Jean-Baptiste Yunès

Sommaire

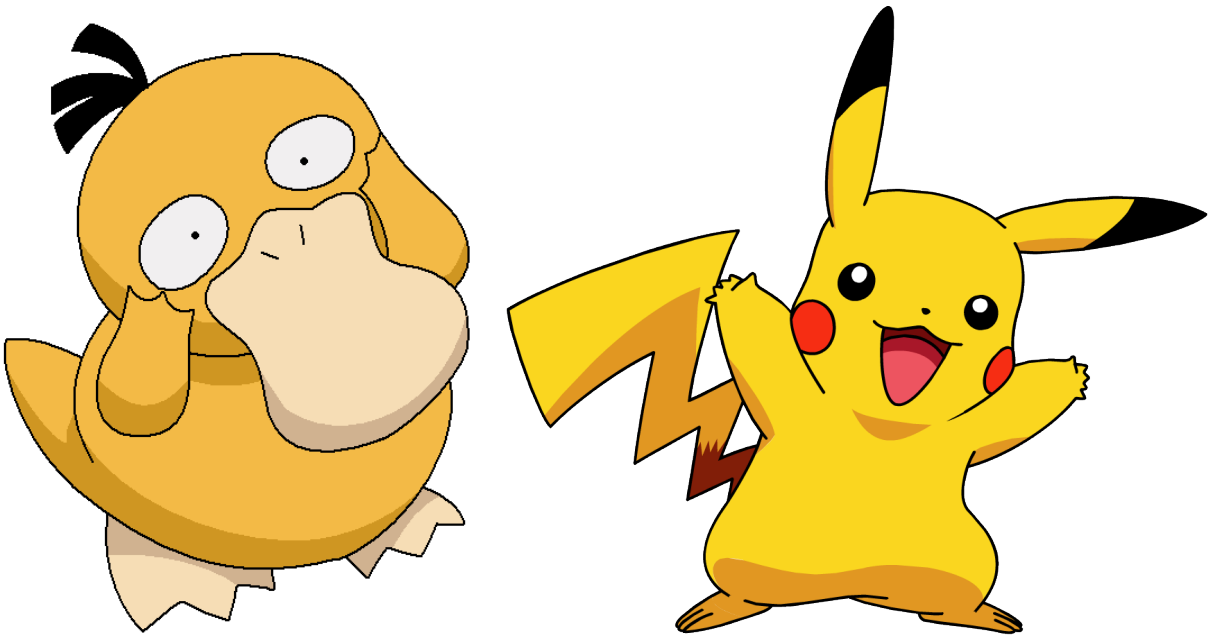
Introduction 3
I Description de l'utilisation du jeu 4
1 Présentation de l'interface de combat 4
2 La carte du jeu 6
II Architecture générale du code 8
III Architecture détaillée et problèmes rencontrés 9
1 Méthodes d'accès et de conversion des données (Tools) 9
2 Structures de données (Pokemon) 11
3 Structure du jeu (Battle) 13
4 Fonctions graphiques réalisées en C (Graphics) 14
Conclusion 15

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.

Pokemon 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 essentials. 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

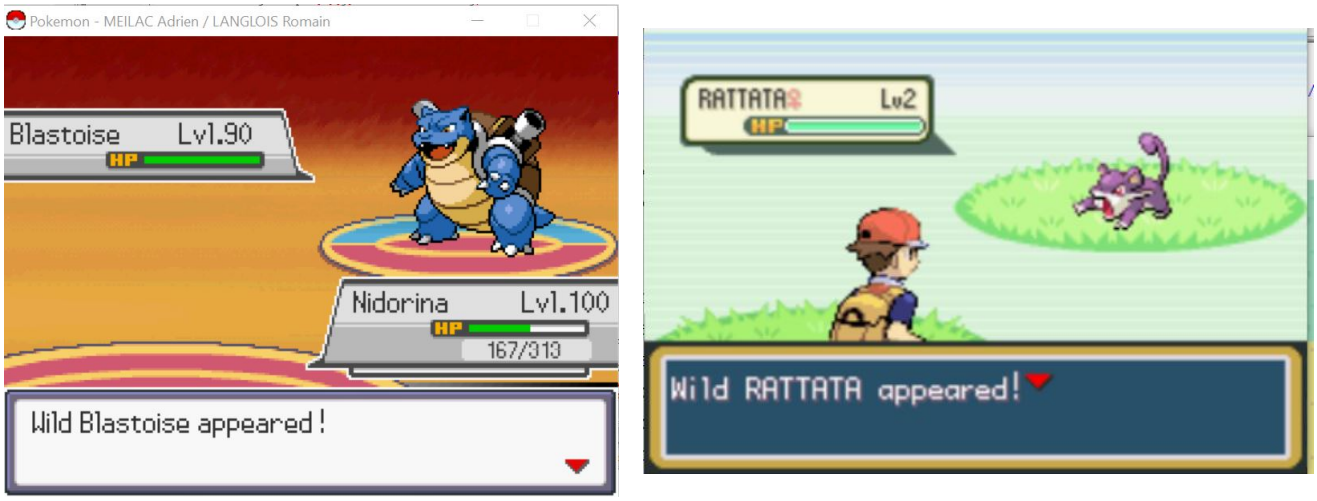


Fig. 1 : Début de combat

Prenons par exemple la rencontre d'un pokémon aléatoire, générée par une loi binomiale lors du déplacement sur la carte. Une fenêtre de dialogue va apparaître, ainsi que l'arrière-plan de combat et également le Pokémon ennemi. Une pression de la touche entrée permet alors d'afficher le menu contextuel de combat.

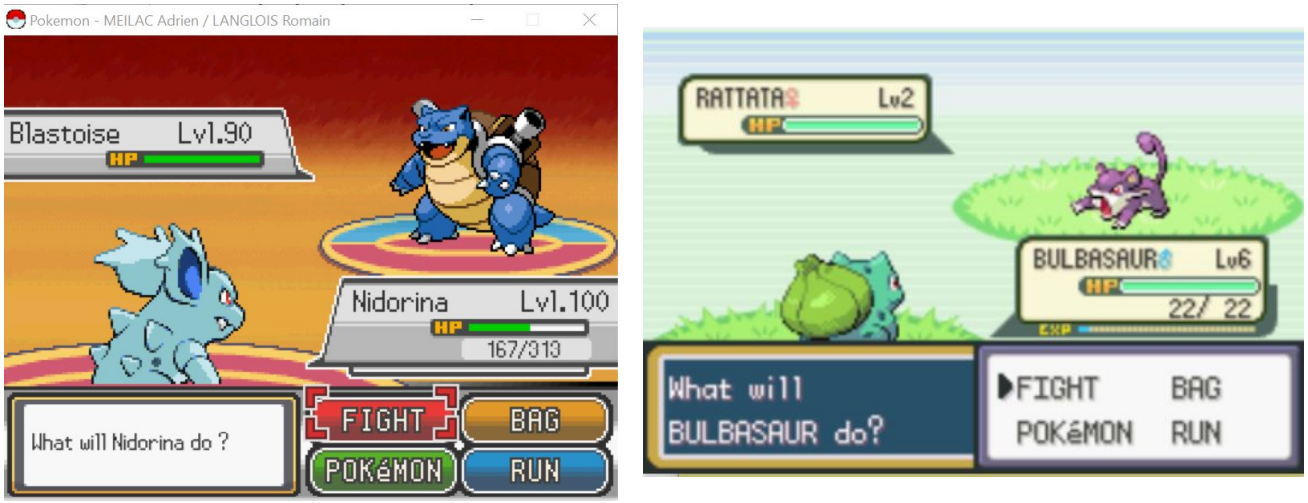


Fig. 2 : Menu principal

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 (POKEMON), s'enfuir (RUN), ou alors attaquer (FIGHT). Si le joueur décide de s'enfuir, il revient sur la map générale et ses Pokemon gardent les dégâts qu'ils

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.

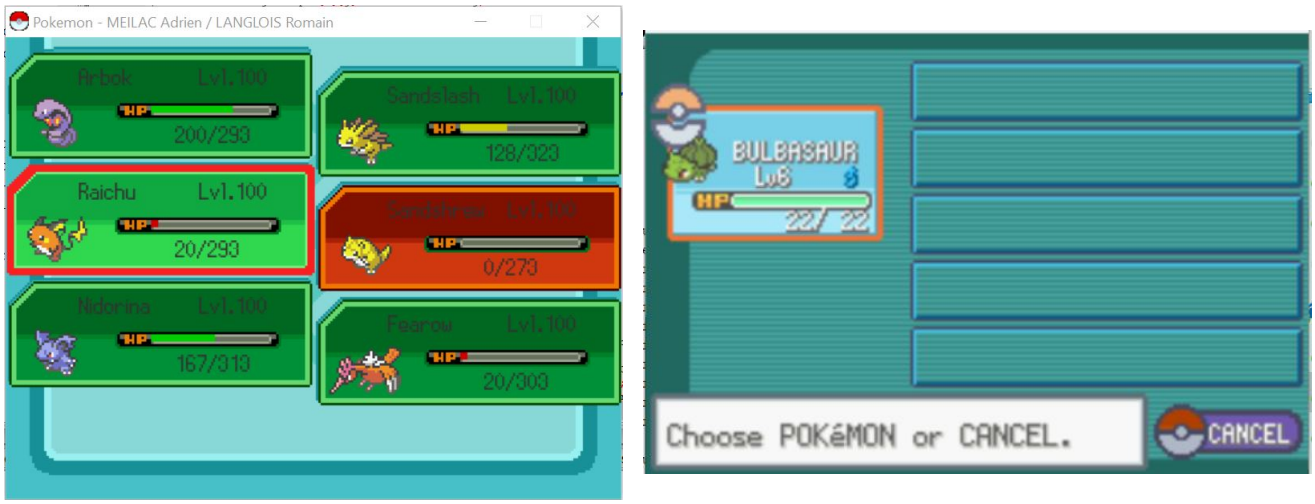


Fig. 3 : Menu du dresseur

Le menu Pokemon, appelé menu du dresseur, permet au joueur de choisir le Pokemon qui va combattre, et de voir l'état de santé ainsi que le niveau de ses Pokemon. Les Pokemon 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.

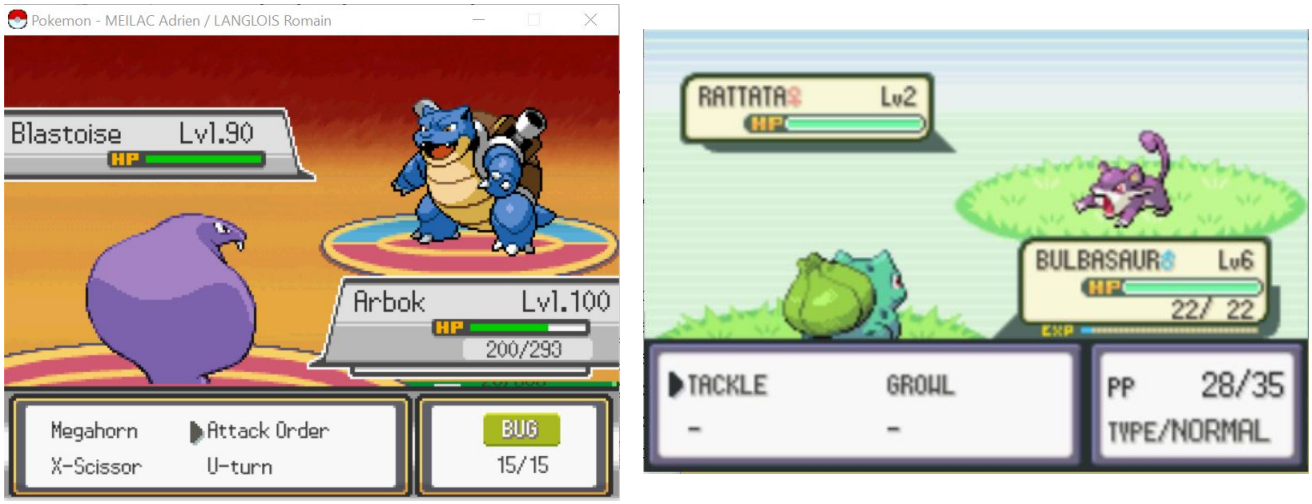


Fig. 4 : Menu de combat

Si le combat est adopté, un menu comprenant les diverses capacités disponibles pour le Pokemon 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 Pokemon. A chaque étape le menu est remplacé par une fenêtre de dialogue indiquant l'attaque utilisée. On voit alors le Pokemon 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.

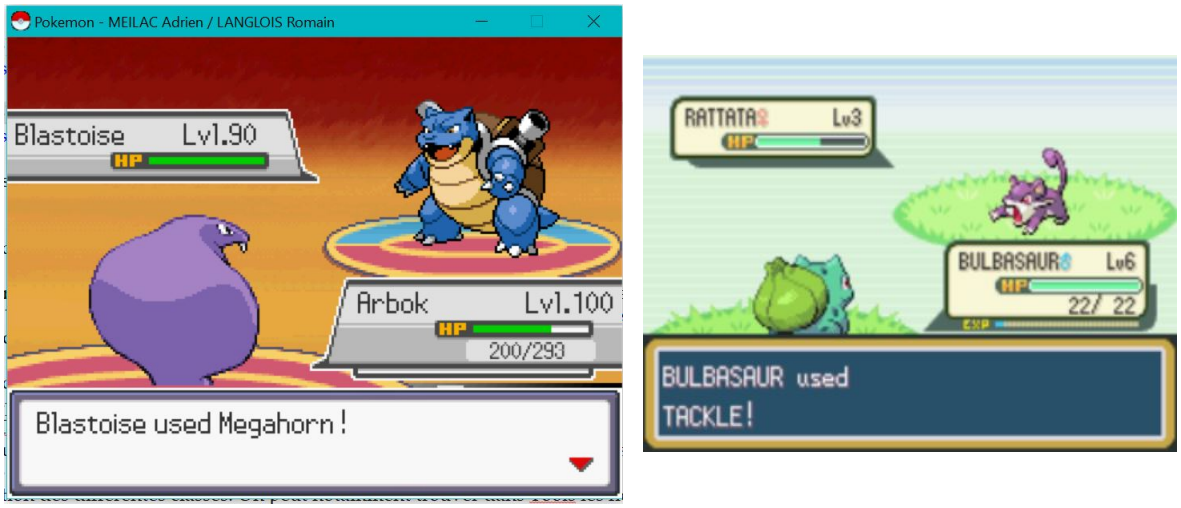


Fig. 5 : Joueur sur la carte Pokémon

Certains des Pokemon 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_Pokemon.txt.

Le pokémon sélectionné par défaut, Nidorina, possède approximativement les stats de son adversaire. Le Pokemon Arbok a été configuré avec des statistiques largement surévalué pour tester les cas de victoires et les autres Pokemon 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 map. Dans le cas où Pokemon du joueur est terrassé, ce dernier se voit contraint de sélectionner un autre Pokemon dans le menu de dressage pour pouvoir continuer le combat. Si tous les Pokemon décèdent, une boite de dialogue Game Over apparait, 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 map 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 Pokemon disponibles ou alors revenir sur la map à l'aide du bouton Exit. Une courte description de chaque option est également disponible.

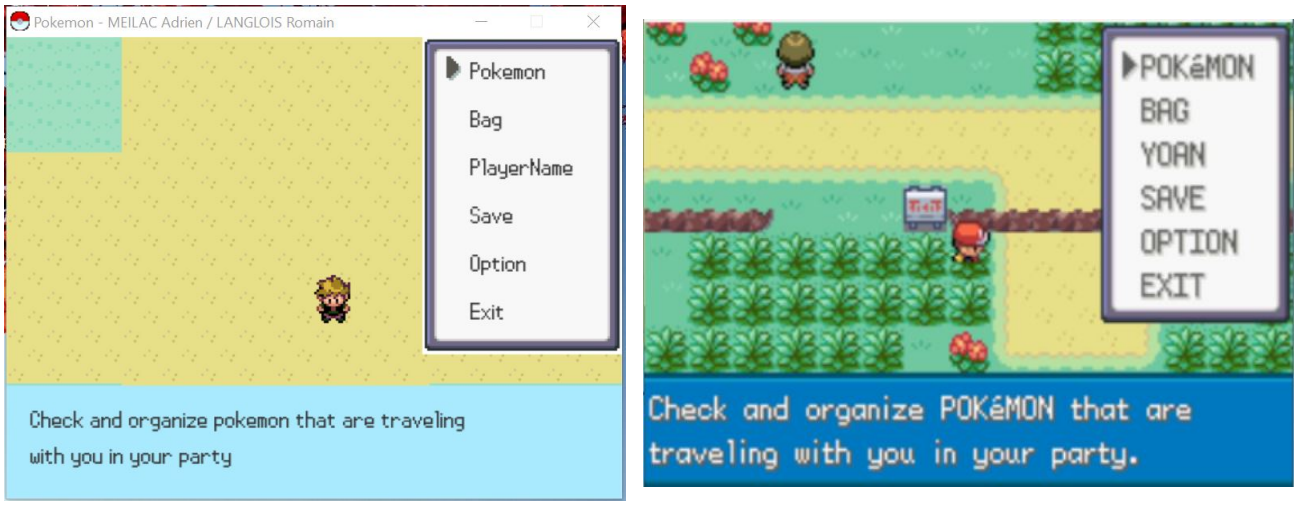


Fig. 7 : Menu sur le terrain

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 map 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

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.

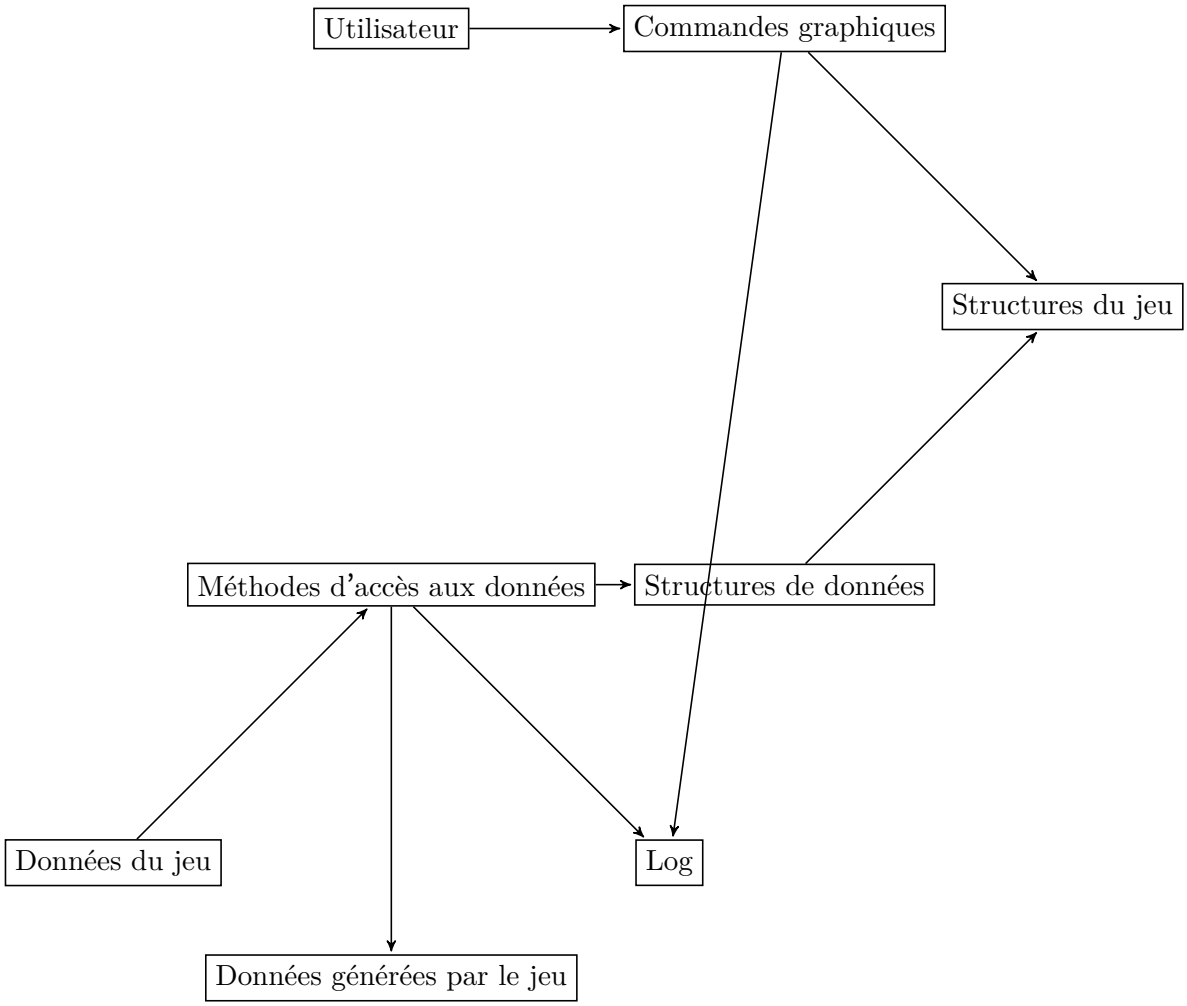


Fig. 8 : Schéma de l'architecture générale du projet

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 différentier des structures du jeu (qui correspond principalement aux fichiers BattleWildPokemon 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). Nous avons décider de les 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 de classe 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.
Enfin notre programme rédige un log (stocké dans stdout.txt) qui nous permet de tracer les incohérences dans le gameplay.
Le jeu en lui même est une boucle infinie qui s'arrête lorsque l'utilisateur ferme la fenêtre.

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 permettre 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)

- **Random** qui contient des fonctions plus simples pour faire appel à l'aléatoire. Nous avons eu des difficultés à utiliser la fonction std ::uniform_real_distribution, c'est pourquoi nous avons créé une approximation de cette dernière à l'aide de rand()

- **StatSet** et **StatSetExt** sont des classes (la seconde héritant de la première) qui permettent de stocker les différentes statistiques de nos Pokemon. 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 Pokemon.
  Si on note

  – **Niv**, le niveau d'un Pokemon,

  – **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 Pokemon 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 Pokemon, 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 Pokemon 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 Pokemon,

  alors les statistiques d'un Pokemon sont définies par :

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

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

2 Structures de données (Pokemon)

Pokemon 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 operateurs == 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 Pokemon 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 Pokemon de Type Feu), ainsi que getPathImage qui permet de donner l'image à afficher pour indiquer le Type d'un Pokemon.

- **Move** contient la classe codant les attaques des Pokemon. Elle est lié 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 au 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 Pokemon. Nous avons décidé de sauvegarder les mouvements que peut apprendre un Pokemon (en moyenne, une centaine) sous forme de string pour ne pas utiliser inutilement de la mémoire étant donné que seul 4 sont utilisés (et sont donc converti en Move dans la classe fille Pokemon).

- **Pokemon** contient principalement les fonctions qui permettent de faire varier les statistiques des Pokemon. 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 Pokemon 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).

Les formules pour calculer les dégâts dans le vrai Pokemon sont assez compliqués :
Si on note :

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

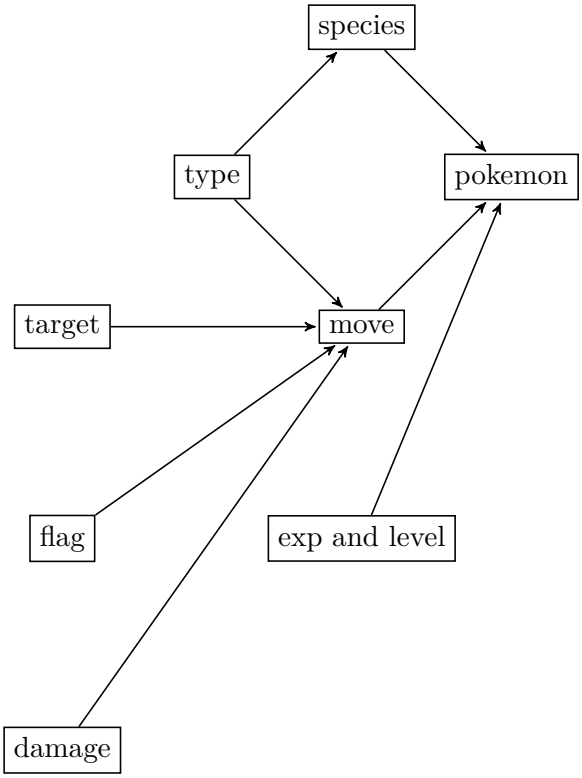


Fig. 10 : Hiérarchie des classes de données

- 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 Pokemon est du même type que son attaque
- Type, l'effet du type de l'attaque sur le Pokemon

alors

$$Damage = \left\lfloor \left(\frac{\left(\frac{2 * Niv}{5} + 2 \right) * Power * \frac{Att}{Def}}{50} + 2 \right) * Modifier \right\rfloor$$

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)

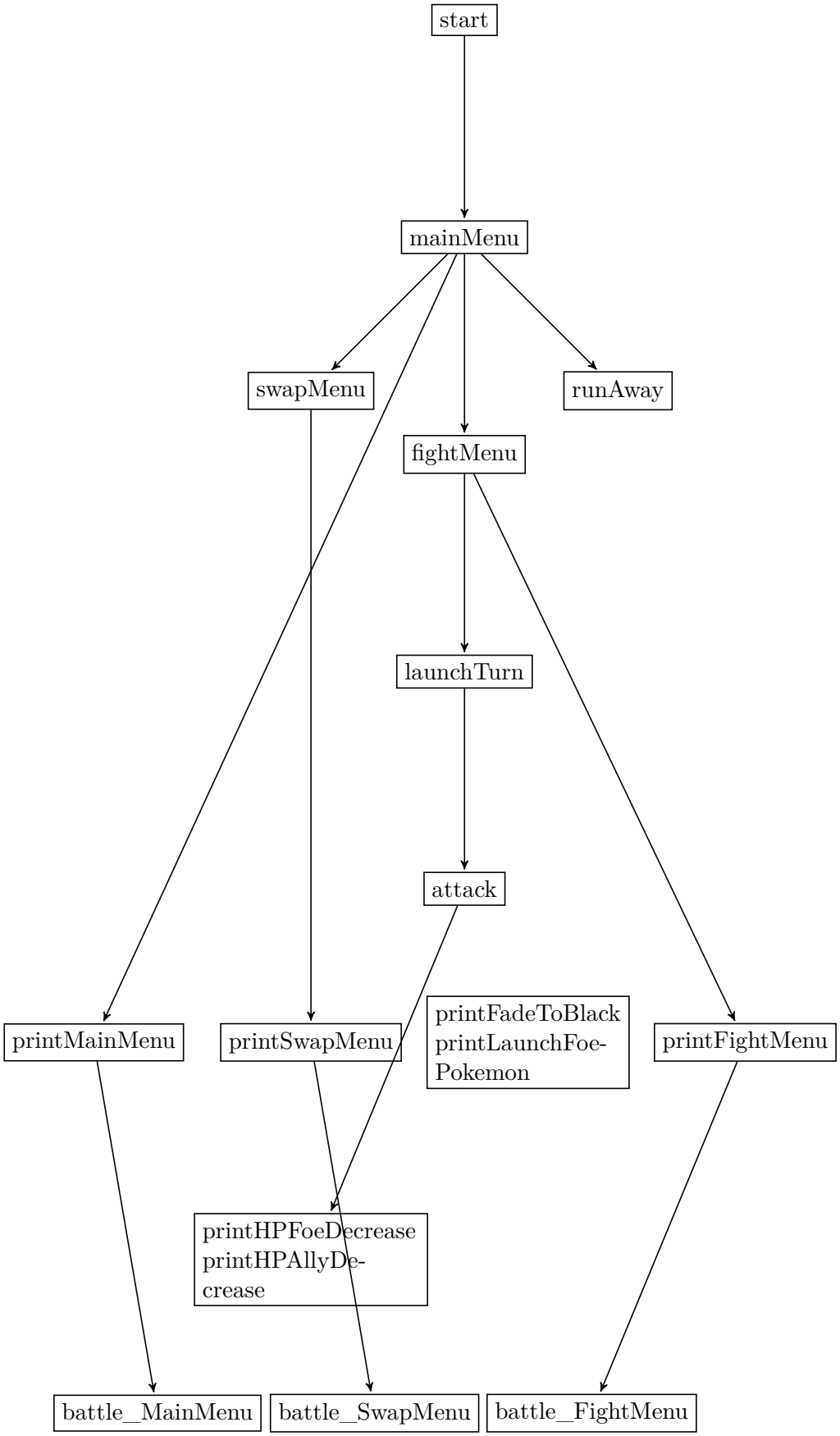


Fig. 11 : Hiérarchie des classes de données

Battle est un répertoire qui contient les fonctions liées aux fonctionnements des combats. Nous avons choisi de séparer les effets visuels (stockés dans Graphics) 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 long avec des fenêtres graphiques car le mécanisme interne des combats est complexe et cela aurait été une source d'erreur.

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 Pokemon allié et celle du Pokemon ennemi en combat.
Nous avons distingué deux principales natures de graphiques :

- Field, 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 Pokemon sauvages. A l'aide de la touche Espace, le joueur peut afficher un menu lui permettant notamment de consulter ses Pokemon.
- l'interface de combat, comprenant l'écran de combat et les menus associés. L'interface de combat est créée à l'aide de macro #define permettant de redefinir 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 Pokemon ennemi, celle du Pokemon allié, le Pokemon ennemi, le Pokemon allié, la barre de menus présente en bas, les deux boites de data contenant la vie des Pokemon, 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 Pokemon allié.

Les menus proposés permettent entre autres de choisir le Pokemon qui combat, de s'enfuir, ou de choisir ses attaques dans le cas du choix du fight menu. Grâce à un usage de la technique de double buffering, alors même que les menus sont gérés par des fonctions différentes, la transition à l'écran s'effectue sans changement visuel pour l'utilisateur.
La plupart des fonctions utilisées sont des fonctions soit de type void, soit renvoyant un flag. Par exemple, l'affichage d'un menu peut être un flag demandé par un utilisateur, tel le Fight menu lors d'un clic sur Fight. Toutes ces fonctions prennent en paramètre l'écran, et décident tour à tour ce qu'elles appliquent dessus. A l'opposé de ces flags volontaires, certains flags telle la rencontre avec un Pokemon, simulée par une loi binomiale, sont involontaires. Une des grosses difficultés du code était de ne pas avoir une seule fonction massive de plusieurs milliers de lignes. Nous avons donc compartimenté notre code en de nombreuses fonctions, ce qui a également aidé au respect du principe de segmentation du code évoqué plus haut. On peut par exemple l'observer dans la fonction LaunchFoePokemon, où 5 lignes de SET_BATTLE servent à faire appel à de nombreuses fonctions définies dans Battle.

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. Ces possibilités, couplées à la programmation modulaire mise en œuvre tout le long du code, nous ont enseigné la réalisation d'un code en équipe. 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 virtuelles. Notre RPG est fonctionnel, mais nous pouvons envisager des perspectives d'amélioration en faisant un RPG Pokemon complet.