

PROJET DE C++ 2A

Réalisation d'un RPG Pokemon

MEILAC Adrien
Langlois Romain

Encadrant :
Jean-Baptiste Yunès

SOMMAIRE

I	Architecture du code	3
1	Les différents répertoires	3

I ARCHITECTURE DU CODE

D'après GitHub, notre code est composé de 35 fichier .cpp et de 23 fichier .h, et notre projet contient environ 11 000 lignes. Pour ne pas se perdre dans notre projet, nous avons donc essayé de hiérarchiser notre code.

1 Les différents répertoires

Notre code est réparti dans répertoires suivants.

1.1 *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).
- **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 fournit par notre IDE CodeBlocks ne nous laissait pas utiliser les fonction `stoi`, `atol` ... 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 aperçu que des 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 :

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

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

1.2 *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 tel sorte que les arguments des uns sont 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 déterminé 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 Pokemon ou des attaques. Les méthodes les plus importantes de cette classes 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 liée au fichier "Data/Move.txt" qui contient énormément 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 tests booléens 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 seuls 4 sont utilisés (et sont donc convertis 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 compliquées :

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 liée 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

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

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 tous ces effets qui ont été rajoutés dans les différentes versions du jeu depuis plusieurs décennies.

2 Battle

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.