Projet Pokimac

Date de rendu : 24 Janvier 2022
Hello, voilà un doc pour parler de l'affichage console et quelques pitis conseils pour démarrer le projet

Allez surtout voir le doc de Philippe qui est vraiment très cool et plus complet sur comment commencer (vous pouvez lui apporter des cookies aussi pour le remercier🍪)

Le principe du jeu est de réaliser un jeu basé sur l'univers du jeu Pokémon.

L'objectif est de pouvoir se déplacer sur un terrain (en 2D vu du dessus à priori) et de pouvoir rencontrer des pok'imacs présents sur le plateau pour se battre avec eux ou les capturer.
Vous êtes assez libres d'imaginer les interactions avec les pok'imacs mais le minimum est de pouvoir au moins choisir de les capturer ou non (en fonction du nombre de pokiballs qu'il vous reste dans votre inventaire par exemple).

Le déroulement du jeu se fait donc en 2 étapes:

1. Une phase de déplacement sur le plateau de jeu
2. Une phase de capture ou de combat lorsque l'on rencontre un pok'imac

Affichage et console

Une partie importante du jeu est donc l'affichage en console (afficher le plateau, afficher les dialogues de combats ou de rencontres avec les pok'Imacs, etc)

C'est souvent complexe d'un OS à l'autre pour être cross-platform et l'idée n'est pas de perdre trop de temps à faire ça !

C'est pourquoi j'ai fait pour vous une petite lib (un header seulement donc facile à intégrer à vos fichiers) qui abstrait un peu l'interaction avec la console (avec les échappements ANSI et l'api windows pour les pitis curieux 😳).
Cela va vous permettre de manipuler facilement la console pour:

- Effacer le terminal
- Déplacer le curseur et ne pas effacer et réafficher tout si besoin (pour le déplacement du joueur par exemple, il suffit d'effacer l'endroit où il se trouve et l'afficher à sa nouvelle position (cela reste bonus. Si c'est plus simple pour vous de tout effacer et tout réafficher ça marche aussi ☺)
- Afficher des jolies couleurs (pour un RobillZard vert par exemple)
- Récupérer une touche sans l'afficher dans la console (permettant de faire des déplacements sans polluer visuellement la console)
- ...

J'ai fait un gros fichier d'exemple vous permettant de tester les fonctionnalités de la libraire et de voir des codes d'exemple (avec le déplacement d'un joueur notamment 🙃)

Tout est dispo ici : https://github.com/dsmtE/ConsoleUtils (Je n'ai testé que sur Windows et Linux)

Exemple:

La librairie permet d'afficher des couleurs et voici un exemple tiré du cpp:

```
std::cout << "Test 1: Colors" << std::endl;
    for (int i = 0; i < 16; i++) {
        consoleUtils::setColor(static_cast<consoleUtils::Color>(i));
        std::cout << i << " ";
    }
    consoleUtils::resetColors();
    std::cout << std::endl << "You should see numbers 0-15 in different colors." <<
std::endl;
```

Petites explications :

consoleUtils dispose d'un enum, consoleUtils::color :

```
enum class Color { BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, GREY, DARKGREY,
LIGHTBLUE, LIGHTGREEN, LIGHTCYAN, LIGHTRED, LIGHTMAGENTA, YELLOW, WHITE };
```

On peut passer à la fonction **setColor** une valeur de cet enum qui permet donc de définir la couleur du prochain texte à afficher.

> Une partie ici peut faire peur, le "**static_cast**". C'est juste un cast (mais à la manière du c++ un peu plus propre) qui me permet de convertir ou interpréter mon `int i` en un consoleUtils::color pour pouvoir être utilisée.

Je prends ainsi toutes les valeurs de cet enum ci-dessous pour tester l'affichage en couleur. Mais on peut très bien l'utiliser comme ceci :

```
consoleUtils::setColor(consoleUtils::Color::RED);
std::cout << "texte en rouge";
```

Pour effacer l'entièreté de la console il suffit d'appeler la fonction :

```
consoleUtils::clear();
```

Voilà un exemple d'utilisation que je vais expliquer et qui permet de récupérer une touche pour effectuer un déplacement d'un caractère sur la console (ici le "@" représentant le joueur par exemple):

```
int x = 7;
int y = 7;
bool spaceHit = false;

while (!spaceHit) {
    consoleUtils::setCursorPos(x,y); std::cout << '@'; // Output player
    char c = consoleUtils::getChar(); // Wait Get char from keyboard
    consoleUtils::setCursorPos(x,y); std::cout << ' '; // Clean old player position
    switch (c) {
        case 'z': --y; break;
        case 's': ++y; break;
        case 'q': --x; break;
        case 'd': ++x; break;
        case ' ': spaceHit = true; break;
        default: break;
    }
}
```

l'idée ici est de pouvoir faire bouger le personnage sans réafficher tout le plateau de jeu (ce qui n'est pas obligatoire mais intéressant).

Plusieurs choses se passent ici:

- J'ai tout d'abord une boucle et une condition d'arrêt (spaceHit) qui est en quelque sorte ma boucle de jeu
- J'affiche ensuite mon joueur à la bonne position à l'aide de la fonction **setCursorPos** qui permet de déplacer mon curseur dans la console (et ne pas afficher bêtement des espaces et devoir tout réafficher)
- j'attends ensuite l'appui de la prochaine touche pour réagir en fonction.

> La fonction **getChar** permet d'attendre et capturer la prochaine touche du clavier sans l'afficher dans la console (à la différence du **cin** qui affiche l'input dans la console). C'est bien pratique ici pour réagir en fonction de la touche reçue et effectuer le mouvement.

- J'efface ensuite mon personnage à son emplacement actuel (ici un espace mais dans votre cas il faut afficher ce qui se trouve dessous c'est à dire le plateau de jeu par exemple)
- Je peux ensuite effectuer le mouvement et modifier la position (dans les variables **x** et **y** (qui peuvent être dans une struct du joueur pour vous (Philippe l'explique dans son doc (Oui ça commence à faire beaucoup de parenthèses 😖))))
- Je boucle et affiche de nouveau mon joueur à la bonne (nouvelle) position.
- J'attends la prochaine touche , etc
- ....

Vous trouverez plein d'autres exemples dans le fichier cpp fourni à coté de la librairie🙃

Structure du plateau:

le plateau sera surement représenté sous la forme d'un tableau 2D (contenant des caractères pour la position des pok'imacs par exemple)

Voilà un exemple de tableau de jeu:

			P
		X	
	S		

Les caractères suivant représentent respectivement:

- X : Joueur
- P : Pikachu
- S : Salameche

Il faut donc une manière de représenter ce plateau et un tableau semble être intéressant dans ce cas de figure.

⚠ Attention au choix que vous allez faire tableau 1D ou 2D !

Paragraphe tiré du sujet :

> Pour rappel on peut utiliser un tableau comme un pointeur et inversement. Accéder à une case c'est accéder à un certain emplacement mémoire situé après l'adresse pointée par tableau. Dans le cas d'un tableau 2D, le compilateur doit connaitre les dimensions du tableau pour savoir comment indexer les cases. Et donc si vous passez un tableau 2D en paramètre d'une fonction vous devez lui

> renseigner dans les paramètres les dimensions de ce tableau, ce qui fixe la taille de votre tableau. Si vous voulez un code souple, **il faut mieux privilégier les pointeurs (et donc des tableaux 1D) mais à vous de déterminer quelle méthode convient le mieux** à votre projet.

Comme on a pu voir en soutien (pour certains), déclarer un tableau 2D sous cette forme `int tab[height][width];` peut poser des problèmes pour le passer à une fonction.

En effet, sous cette forme, la taille est statique pour le compilateur et il n'est pas possible d'interpréter ce tableau comme un double pointeur mais seulement comme un pointeur vers un tableau de taille statique comme cela:

```
void updateMyTab(char tab[][10], unsigned int height) {
    for (unsigned int i = 0; i < height ; i++) {
        for (unsigned int j = 0; j < 10; j++) {
            ...
        }
    }
}
```

C'est pas très pratique car on aimerait éventuellement pouvoir changer la taille du plateau par exemple (depuis un menu) et pas forcément avoir la taille en dur de notre plateau dans le code et dans la signature de toutes les fonctions qui le manipule.

Il est possible plutôt, d'utiliser un tableau 1D que l'on **affiche** comme un 2D. Cela permet d'avoir la flexibilité de le manipuler comme un pointeur comme on a vu dans les TDs tout en ayant visuellement la même chose.

En considérant que toutes les **n** cases on change de ligne on peut donc accéder à notre position (l, c) (pour **l**igne et **c**olonne) en faisant $\mathrm{tab}[l * \mathbf{n} + c]$au lieu de $\mathrm{tab}[l][c]$

> ⚠ On initialise donc notre tableau pour qu'il ait height*width case:
>
> ```
> char tab[width*height];
> ```
>
> ou avec un malloc:
>
> ```
> char tab[] = (int *)malloc(width * height * sizeof(int));
> ```

On obtient alors:

```
void updateMyTab(char tab[], unsigned int width, unsigned int height) {
    for (unsigned int i = 0; i < height ; i++) {
        for (unsigned int j = 0; j < width; j++) {
            tab[i*width+j] = ...
        }
    }
}
```

Il y a des supers exemples dans le doc de Philippe ☺

Conseils

Les pok'imac sont des créatures qui se différencient par certaines caractéristiques :

- Sa représentation dans la console (soit juste un nom, soit un ascii art)
- Son endurance (qui vont définir des points de vie)

Et pour les plus fou d'entres vous:

- Son espèce (un nom d'espèce)
- Sa force
- Sa défense
- ...

Une structure de type :

```
struct Pokimac {
    int vie;
    ...
}
```

est donc intéressante et vous permettra de manipuler vos pok'imac facilement

Quelques pistes pour aller plus loin:

- Il est possible de créer une **interface graphique** pour les plus fous d'entres vous (avec SDL par exemple comme présenté dans le sujet même si je suis pas très fan de la librairie)
- **les classes**

  SI vous voulez essayer d'utiliser des classes c'est possible (une fonctionnalité du c++ mais pas du c)

  Elles on certains avantages par rapport au structs mais c'est vraiment bonus (et il faudra justifier votre choix) et pas nécessaire pour mener à bien votre projet 😃

  Voila une liste non exhaustive des avantages des classes:

  - Les opérateurs peuvent être crées sur les classes (comme + -)
  - les classes supportent le masquage de données (attributs privés). Les membres d'une structure peuvent être accédés par n'importe quelle fonction indépendamment de sa portée.
  - les classes supportent les membres statiques.
  - Une classe dispose d'un constructeur et d'un destructeur qui peuvent s'avérer bien pratique. (pour s'assurer de la libération de mémoire allouée par exemple 👀)
  - ...
- Voilà une source pour la culture qui explique les différentes façon de manipuler des tableau 2D: https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/ (la version 4 présentée sur le site est un peu tricky et je ne vous conseil pas de l'utiliser)

Je reste avec votre disposition jours et nuits et pendant les vacances (bon nuit abusez pas quand même 😝) pour répondre à vos questions et vous aider si besoin.