RAPPORT DE PROJET

BUCHERT Timothé - CAMON Baptiste

1. Description de l'application développée

L'application que nous avons développée modélise la représentation du jeu Monopoly en C++, en version coupe du monde de football :

- ❖ Les propriétés de maisons sont remplacées par des stades de football, les monopoles sont donc deux ou trois stades d'un même pays
- ❖ Les propriétés de gares sont remplacées par des archives des finales historiques de la coupe du monde (1998, 2006, 2014, 2018 (allez les bleus !))
- ❖ Les cases électricité et eau du jeu de base sont conservées, et ajusté au contexte (payer des réparations d'eau / d'électricité au(x) stade(s))
- ❖ La case prison est remplacée par une case "carton rouge"
- ❖ Les cases caisse de communauté et chance sont toutes remplacées par des cases Loterie :
  - Vous devez cotiser pour l'entretien du terrain : - 50 M
  - Vous avez gagné le concours de dribble : + 30 M
  - Vous récupérez vos gains sur le pari du match en cours : + 20 M
  - Tacle par derrière : carton rouge
  - Un joueur vient de faire faute sur vous : allez directement à la case départ (et empochez vos 200 M) pour vous rétablir

- ❖ Le principe des cases taxes et parc gratuit reste inchangé, seule la case parc gratuit est renommée "Terrain libre" et permet d'encaisser l'argent des taxes si le joueur tombe dessus

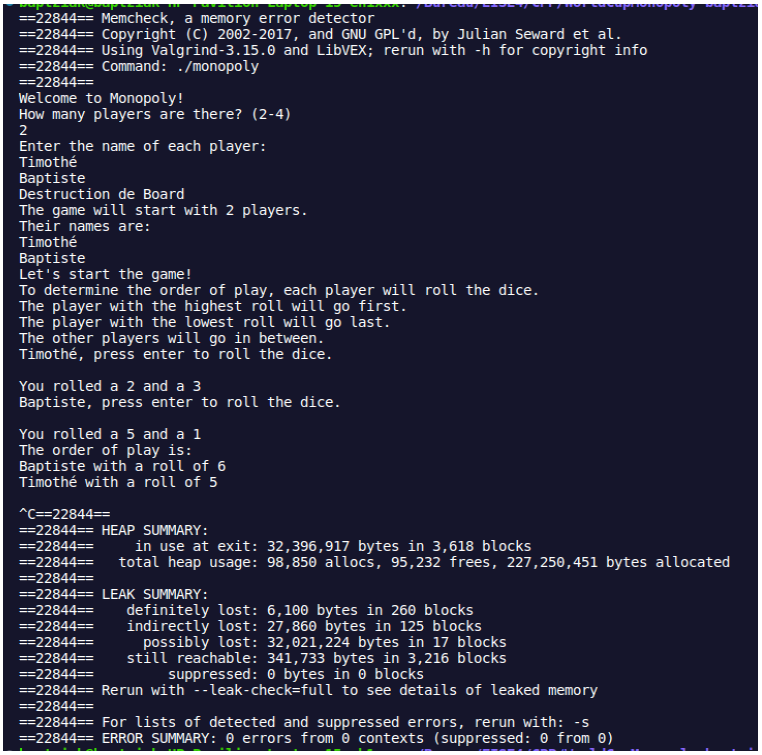
En dehors de ça, le déroulement du jeu reste exactement le même dans les règles du Monopoly classique.

2. Mise en valeur de l'utilisation des contraintes

- 8 Classes : comme vous pouvez le constater dans le diagramme UML, nous avons bien plus que 8 classes (qui était le minimum), mais cela était plus pratique pour nous !
- 3 Niveaux de hiérarchie : nous avons bien respecté une modélisation a 3 niveaux de hiérarchie minimum. Les classes de propriétés de stades avaient elles aussi à l'origine une classe mère différente de Box, qui s'appelait StadiumBox, mais celle-ci à été supprimée pour des soucis de modélisation. Dans l'esprit, il n'y avait donc que les cases multiples (type loterie) ou représentant une possession (vidéo, réparation, stade…) qui avaient une classe mère différente de Box. Cela explique pourquoi en dehors des cases stade, les cases unique type case départ ou terrain libre héritent directement de Box.
- 2 surcharges d'opérateurs : nos deux surcharges d'opérateurs sont situées dans le header de la classe Board, et nous permettent d'afficher le nombre de joueurs, ainsi que de pouvoir

rentrer dans le terminal le nombre de joueurs dans l'attribut prévu à cet effet au sein de la classe Board. (nbPlayers)

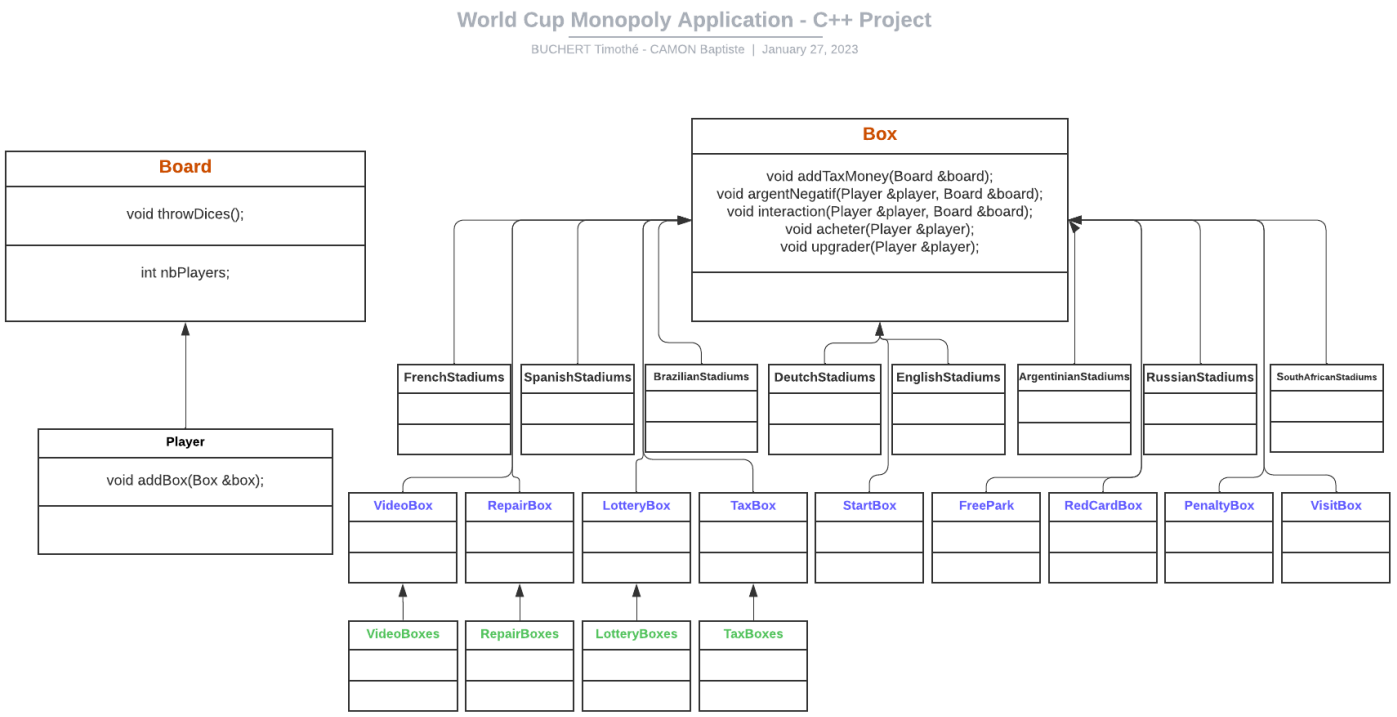
- 2 conteneurs différents de la STL : nous avons choisi d'utiliser les deux conteneurs suivants : vector et map. En effet, les vecteurs nous permettent aisément de manipuler les possessions des joueurs, ainsi que les joueurs du plateau, dans la classe Board. D'autre part, le map nous permet d'accéder à tout moment à une case en fonction de son nom et de son numéro de case, défini manuellement lors de la création de chaque case (en fonction bien sûr de sa position sur le plateau).
- pas d'erreur avec valgrind : comme vous pouvez le constater sur la capture d'écran ci-dessous (qui correspond à un lancement de l'application avec valgrind, une rapide exécution de celle-ci puis un arrêt forcé), il n'y a pas d'erreur dans notre code, même si on peut constater quelques petites fuites mémoires.



Les tests unitaires ont été mis en place, ils sont accessibles dans le dossier de l'application, exécutables avec la commande make test, puis ./test.

Notre makefile contient une règle "all" et une règle "clean".

3. Diagramme UML de l'application



4. Procédures d'installation (bibliothèques...) et d'exécution du code

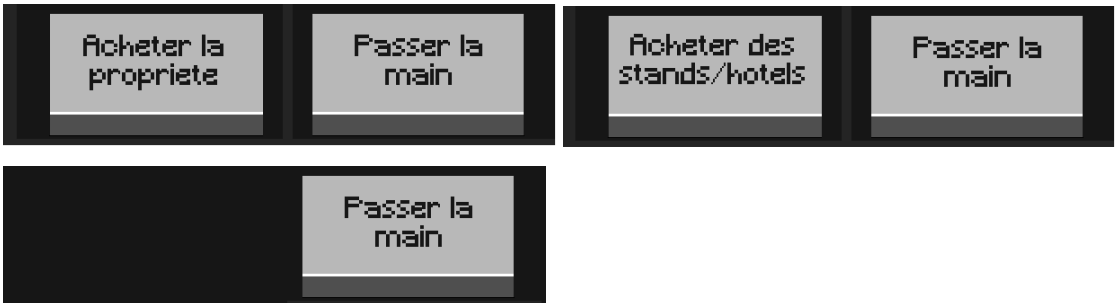
Le projet utilise une seule bibliothèque externe: SDL2 que l'on peut installer dans un terminal Linux. L'utilisation d'SDL2 ici est justifiée par le fait qu'il s'agisse d'un jeu de plateau, avec peu d'animations.

L'actualisation de l'affichage graphique est gérée à l'intérieur de manageRedraw() et l'initialisation du chargement de toutes les textures et de la création de la fenêtre sont gérés à l'intérieur de init_sdl().

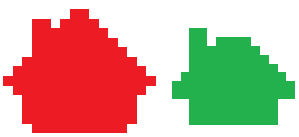
Avant de détailler les fonctions, on peut s'intéresser au découpage de l'affichage et les différents plans:

1. D'abord un tableau vide est présenté en arrière plan:

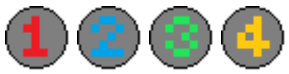
2. En fonction des interactions nécessaires, on ajoute un calque:



3. En parallèle on regarde l'intégralité des niveaux de chaque case afin d'afficher les stands (en vert) ou Hotel (en rouge) en ajoutant un chiffre représentant le nombre de stands:



4. Ensuite on place les joueurs:



5. Si un évènement nécessite un message en plus, on fait apparaître un calque gris ou l'affichage des dés en complément:

⚀⚁⚂⚃⚄⚅

<u>Au niveau du code:</u>

la fonction `manageredraw()` **commence avec cette boucle:**

```
for (auto it = board->getPlayers().begin(); it != board->getPlayers().end(); ++it)
```

Qui met à jour les coordonnés des sprites pour leur affichage plus tard.

Ensuite on affiche le fond (1)

```
SDL_RenderCopy(renderer, texture_monopolyboard, NULL, NULL);
```

Puis on parcourt toutes les cases afin d'afficher si besoin le niveau des cases (3):

```
SDL_RenderCopy(renderer, texture_maison, NULL, &Posup);
myRenderText("1", x + 20, y + 20);
```

`myRenderText()` **est une fonction permettant d'afficher un texte mis en paramètre à des coordonnées.**

On affiche ensuite les joueurs (4):

```
SDL_RenderCopy(renderer, texture_p1, NULL, &PosP1);
```

Puis en fonction de la valeur retourné par la fonction interaction on modifie l'affichage (2):

```
if (affichage_interactions == 1)
        SDL_RenderCopy(renderer, texture_passerlamain, NULL, NULL);
    if (affichage_interactions == 2)
        SDL_RenderCopy(renderer, texture_acheterpropriete, NULL, NULL);
    if (affichage_interactions == 3)
        SDL_RenderCopy(renderer, texture_acheterstands, NULL, NULL);
    if (affichage_interactions == 4){
        SDL_RenderCopy(renderer, texture_passerlamain, NULL, NULL);
        SDL_RenderCopy(renderer, texture_gray, NULL, NULL);
        myRenderText((board->getMessage()).c_str(), 225, 450);
        myRenderText((board->getMessage2()).c_str(), 225, 490);
        myRenderText((board->getMessage3()).c_str(), 225, 530);
    }
```

Enfin on affiche tous les différents textes de la case en cours et de l'événement qu'il vient de se passer:

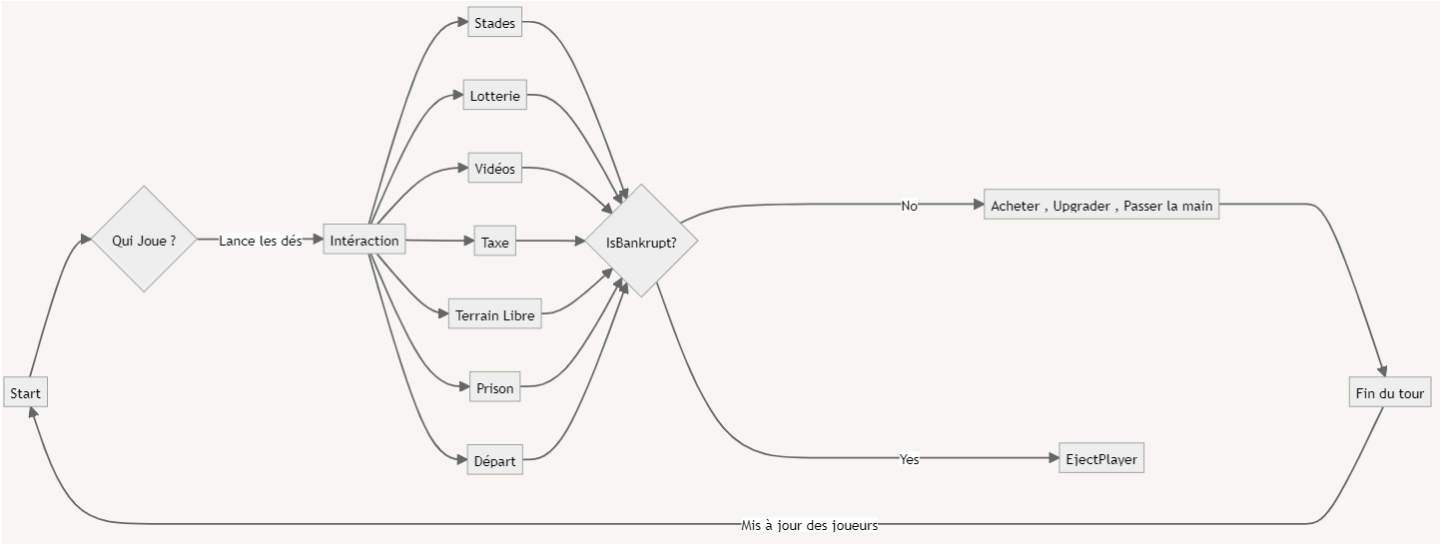
```
myRenderText((std::to_string(board->getPlayers()[board->getWhosPlaying()].getMoney())+" M").c_str(), 1198, 163);
```

Puis on termine la fonction par l'affichage des dés:

```
SDL_RenderCopy(renderer, texture_de1, NULL, &De2);
```

Cette externalisation de cette fonction nous permet de l'appeler que lorsque qu'un événement nécessitant le changement d'affichage a été effectué. Limitant ainsi les usages des ressources.

Pour ce qui est du fonctionnement d'une partie, le code peut être représenté de cette manière:



5. Les parties de l'implémentation dont nous sommes les plus fiers

Les parties de l'implémentation dont nous sommes les plus fiers sont la modélisation graphique de l'application, dans le main, ainsi que la fonction "Box::interaction".

- Pourquoi la modélisation graphique ? Parce qu'elle est magnifique pardi !

Plus sérieusement, le rendu graphique nous donne vraiment de quoi être fier de notre projet car il représente la consécration et l'aboutissement de celui-ci. En quelque sorte, il nous permet de tester la viabilité de notre application, et de constater certaines pistes d'améliorations qui pourraient nous permettre de peut-être, commercialiser un petit jeu vidéo indépendant un jour !

- Pourquoi la fonction Box::interaction ?

D'abord parce qu'elle représente l'esprit de la modélisation de notre jeu en C++ : l'interaction entre les joueurs et les cases sur lesquelles ceux-ci tombent.

Elle est en quelque sorte le noyau de notre projet, car c'est la méthode la plus importante et la plus réfléchie de notre projet.
C'est dans cette méthode qu'est modélisé, individuellement, à l'aide du type enum, le rôle de chaque type de cases.
En parlant de lui, ce type enum contient donc les noms de tous les types de cases (propriété, loterie, case départ…) afin de pouvoir différencier l'action à effectuer en fonction des cases.

En bonus, nous trouvons que les cases loterie que nous avons implémentées à la place des cases classiques de happening (chance, caisse de communauté) sont très originales, et nous font beaucoup rire ! (mais ce n'est pas pour autant que le tacle par derrière deux pieds décollés est autorisé… attention à vous car vous pourriez finir exclu !)