# Projet C++ : Des chiffres

Thomas Aubertier
Pascal Pham

Polytech Sorbonne Université, MAIN4
Janvier 2024

## Contents

**1 Description de l'application développée** **2**

**2 Mettre en valeur l'utilisation des contraintes** **3**

**3 Diagramme UML de l'application** **4**

**4 Procédure d'installation des bibliothèques** **5**

**5 Procédure d'exécution du code** **5**

**6 Tests unitaires** **6**

**7 Les parties de l'implémentation dont nous sommes le plus fier** **7**

# 1 Description de l'application développée

L'application développée est un jeu interactif basé sur un concept simple, conçu pour tout type de public (enfants, adolescents, adultes). À partir des nombres proposés et des 4 opérations de l'arithmétique élémentaire, l'objectif de ce jeu de chiffres est d'atteindre un objectif donné, ou du moins, s'en rapprocher.

Plusieurs contraintes et particularités existent :

- Chaque calcul est effectué indépendamment, dans n'importe quel ordre. Une fois l'opération effectuée, le résultat est ajouté à la liste des nombres disponibles.
- La division n'est possible que si le reste est nul et le dénominateur est non nul. Dans le cas contraire, rien ne se produit.
- Un nombre n'est utilisable qu'une seule fois. Vous pouvez également ne jamais l'utiliser.

Lorsque vous lancez l'application, vous vous trouvez dans le menu du jeu. 4 options s'offrent à vous. Les chiffres marqués sur les 4 boutons correspondent au nombre de "nombres proposés" pour le jeu de chiffres. Plus cette valeur est élevée, plus le jeu est *simple*, car, il existera plus de combinaisons possibles pour se rapprocher de la valeur recherchée. En cliquant sur l'un de ces 4 boutons, la partie se lance.

Maintenant, vous pouvez observer en haut à gauche, l'objectif à atteindre, qui est un nombre aléatoire entre 100 et 999, et votre résultat, commençant à 0. En bas de la fenêtre, il y a 4 boutons correspondant aux opérations et au milieu de la fenêtre, des boutons bleus contenant les nombres proposés. Le nombre de boutons bleus dépend du choix fait dans le menu.

Pour jouer, sélectionnez un de ces boutons, choisissez ensuite l'opération élémentaire, et enfin un autre bouton de nombre. Cela créera un nouveau bouton correspondant au résultat de l'opération entre ces 2 nombres. Les boutons de nombre deviennent rouges, une fois cliqués, signifiant qu'ils ne sont plus utilisables. Lorsqu'on clique sur un bouton utilisable, il apparait vert, cela permet d'indiquer la dernière action réalisée.

En bas à gauche se trouvent les boutons pour les opérations et à droite les 2 boutons d'action : Valider et Effacer. Le bouton "Effacer" réinitialise toutes les actions réalisées précédemment, les boutons crées disparaissent de la fenêtre. Le bouton "Valider" permet de soumettre votre réponse, on ne peut alors plus revenir en arrière, et le jeu se termine. Donc, avant de valider, il faut sélectionner un bouton de nombre car le résultat correspond à la valeur du bouton. Sinon, votre résultat reste constamment à 0. Cela signifie également que pour avoir le résultat le plus proche possible de l'objectif, il faut un bouton contenant une valeur approchée de l'objectif.

Le score affiché correspond à la différence entre la valeur objective et votre résultat. Ce score détermine également ce qui sera affiché ! Enfin, si vous voulez relancer une nouvelle partie, il suffit d'appuyer sur le bouton "Rejouer" en bas de la fenêtre. Dans le cas où vous souhaitez changer la difficulté du jeu, vous pouvez revenir au menu en cliquant sur "Menu".

# 2 Mettre en valeur l'utilisation des contraintes

Afin de développer le jeu et pour bien séparer les tâches à réaliser, créer plus de 8 classes n'est pas réellement une contrainte. D'une part, nous allons créer des classes correspondant à des catégories de bouton, et pour mettre en place le jeu de chiffres, nous avons besoin d'un nombre important de boutons que cela soit pour les calculs, ou pour rendre le jeu plus agréable. D'autre part, le jeu contient plusieurs scènes : au début un menu (`StartScene`), ensuite la scène pour la phase de jeu (`NumberScene`), et enfin une scène pour le résultat (`EndScene`). Comme ces 3 scènes sont significativement différentes et que les boutons à gérer changent, il est préférable de créer des classes pour chaque scène. Cela permet également dans un premier temps de construire le jeu, puis, dans un deuxième temps de l'améliorer. Il s'agit d'une structure usuelle dans le développement de jeu. À noter que la classe mère `Scene` ne sert qu'à charger la police du texte.

Le code contient 3 niveaux de hiérarchie : en effet, chaque type d'opération correspond à une classe dérivée de la classe `OperationButton`, qui dérive de la classe `Button`.

Le deuxième niveau de hiérarchie permet de distinguer les propriétés des différents types de bouton, par exemple, le sprite appliqué sur les boutons de numéro n'est pas le même que celui appliqué sur les boutons d'opération. Ensuite, comme énoncé dans la description de l'application, la division ne fonctionne pas toujours, contrairement aux 3 autres opérations. Le troisième niveau de hiérarchie permet de distinguer leurs propriétés avec l'utilisation d'une fonction virtuelle "`compute`". Elle est redéfinie pour chaque opération : pour la division, elle annule l'action réalisée si le résultat n'est pas un nombre entier ou le dénominateur est nul, sinon l'opération est effectuée, tandis que pour les 3 autres opérations, l'opération sera forcément réalisée.

L'utilisation des vecteurs est particulièrement utile pour gérer autant de boutons à la fois. Par exemple, un vecteur permet de stocker les boutons de même type, on peut donc leur fournir les mêmes propriétés (taille, couleur, police). On peut ensuite faire la même action sur chacun à l'aide d'une boucle. Il y a également le fait qu'on peut utiliser les indices pour pouvoir aligner les boutons, au lieu de les placer au cas par cas. Ici, il y a 6 boutons par rangée et sont équidistants. Enfin, la méthode `emplace_back` trivialise l'insertion d'un nouveau bouton dans la liste.

On s'est également servi des maps pour stocker les données des différentes images/sprites utilisées, notamment pour leur position, taille avec la map `imageData`. En effet, comme l'image affichée dépend du score obtenu à la fin, on se retrouve donc avec des images ayant des propriétés différentes. On a donc ajusté leur position et leur taille par esthétisme et récupérer ces données en fonction de l'image que l'on veut afficher est très facile à partir de `imageData`. Il est important de déclarer les `sf::Texture` en `static` car ce sont des structures lourdes en ressources.

# 3 Diagramme UML de l'application

Le diagramme UML est disponible dans le dossier via l'image "`uml.png`". Pour le générer, nous avons utilisé l'outil hpp2plantuml, qui créé un fichier source "`uml.puml`".

Voici les commandes qui ont été utilisées :

- `hpp2plantuml -i "*.hh" -o uml.puml` : Génère le fichier source en analysant tous les fichiers headers .hh dans le répertoire courant.
- `plantuml -tpng uml.puml` : Convertit le fichier en image .png. Il est aussi possible de générer un version en .svg en utilisant le flag `-tsvg` à la place.

Attention : le fichier `endscenes.hh` a tendance à créer une erreur lors de son analyse. Il est alors préférable de retirer le mot-clé `static` de la map pour corriger le bug.

Remarque : l'image est bien trop grande pour être affichée dans ce PDF.

# 4 Procédure d'installation des bibliothèques

Le jeu nécessite une interface simple permettant une interaction avec l'utilisateur, d'où l'utilisation de la bibliothèque graphique SFML. Pour l'installer sur Linux, il suffit de rentrer la commande "sudo apt-get install libsfml-dev" dans le terminal. D'autres options existent et sont expliquées sur ce site. Pour que la bibliothèque SFML soit reconnue lors de la compilation, la commande lors de la compilation doit comporter à la fin "-lsfml-graphics -lsfml-window -lsfml-system".

# 5 Procédure d'exécution du code

Vérifiez avant si le Makefile est bien dans le même répertoire que l'ensemble des fichiers du projet.

- `make clean` pour effacer tous les fichiers objet et exécutables,
- `make jouer` pour compiler et lancer l'exécutable "cestincroyable.out",
- `make test` pour exécuter les tests unitaires.

# 6 Tests unitaires

Les tests unitaires du projet ont été réalisés dans le fichier `Tests_Unitaires.cc`, en s'inspirant de la structure des `TestCase.cc` des précédents TP.

- Par soucis d'automatisation, aucun test n'est relatif à l'appui de bouton. Seules les fonctions de calcul et les getters sont isolées.
- Les deux opérateurs relatifs à la classe `NumberScene` servent uniquement à ces tests. En effet, tous les composants de la scène sont directement accessibles dans ses méthodes, et le terminal n'est pas utilisé en tant normal pour afficher de l'information.
- Le programme s'arrête lorsque l'un des tests échoue (il suffit de changer les égalités pour s'en rendre compte). Chaque test passé affiche un ".". Chaque section entièrement validée affiche un "OK".
- Les tests concernent très largement `NumberScene` et ses boutons, car il s'agit des comportements les plus complexes et les plus sujets à l'erreur. Beaucoup des autres méthodes n'interagissent pas avec des composants majeurs du jeu.

# 7 Les parties de l'implémentation dont nous sommes le plus fier

- **La répartition des rôles des boutons :** Chaque comportement principal dispose de sa propre classe, ce qui évite l'utilisation de fonctions externes ou de "hardcoder" chaque bouton à la main en vrac lors de la déclaration. De plus, cette disposition permet de limiter de recopier la plupart des fonctions dans chaque classe enfant.

- **L'économie des ressources :** Comme dit précédemment, le jeu utilise le plus de méthodes et arguments `static` possible. Cela permet de réduire le stockage des ressources les plus coûteuses, en plus de mettre en valeur ce qui est commun et ce qu'il ne l'est pas.

- **La liste des boutons opératoires :** Cette liste doit être déclaré du type `OperationButton`, malgré le fait qu'aucun de ses éléments n'en soit directement une instance (étant des classes enfant). Pour que l'appel de la méthode `compute` référence bien celle des classes enfant et pas celle d'`OperationButton`, on a dû y référencer les pointeurs des objets afin que le typage des boutons soit conservé. Cela a permis de consever l'aspect très pratique du vecteur pour itérer sur les boutons, malgré les contraintes de type.

- **Les ajustements de l'affichage graphiques :** Tout d'abord, chaque type de bouton possède sa manière de centrer le texte dans son rectangle (les méthodes `draw`). L'opérateur a une position constante (`string` de taille 1), et le nombre peut être placé selon sa longueur grâce à un calcul de taille. En revanche, nous avons été contraint pour les boutons génériques (`ActionButton`) de spécifier le décalage à la main car certains caractères prennent plus de place que d'autres. Enfin, les arrières-plans on pu être réalisés en pixelart à petite échelle avant d'être redimensionnés, tout en évitant toute compression et déformation.