 

Manuel du développeur

Jeu 2048

Une image contenant Graphique, symbole, Police, logo

Le contenu généré par l’IA peut être incorrect. Une image contenant Graphique, cercle, symbole, capture d’écran

Le contenu généré par l’IA peut être incorrect.

***Réalisé par :***

*Nathan Talbot* ***Année Universitaire***

*Léo Bernardin**2024-2025*

Table des matières

[I) Sommaire 3](#_Toc190354528)

[II) Partie analyse et conception du logiciel 4](#_Toc190354529)

[1) Les cas d’utilisation 4](#_Toc190354530)

[2) Le diagramme de classes 7](#_Toc190354531)

[3) Le diagramme de séquences 10](#_Toc190354537)

[III) Partie programmation du logiciel 11](#_Toc190354538)

[1) L’architecture du logiciel 11](#_Toc190354539)

[2) Les points importants du code 12](#_Toc190354540)

[3) Présentation des écrans 17](#_Toc190354547)

1. Sommaire

Durant notre année scolaire 2024-2025, nous avons eu pour projet de programmer le jeu 2048 via le langage C++ et l’API Qt Creator.

Ce document « Manuel du développeur » a pour objectif d’aider n’importe quel programmeur qui voudrait réutiliser le code que nous avons implémenté en utilisant la programmation orientée objet (POO).

De ce fait, ce manuel doit permettre au lecteur de comprendre les différentes parties de notre code nécessitant d’être détaillées afin d’être comprise.

Pour ce faire, nous commencerons par détailler l’analyse et la conception du logiciel.

Dans cette partie, nous présenterons les diagrammes UML que nous avons réalisés afin d’illustrer graphiquement le principe de notre logiciel :

* Le diagramme de cas d’utilisation qui permet de saisir les différentes fonctionnalités offertes à l’utilisateur du logiciel.
* Le diagramme de classe qui spécifie la logique de POO utilisée, en détaillant les différentes classes implémentées dans notre code.
* Le diagramme de séquence permettant de saisir les différentes étapes de l’exécution de notre programme.

Ensuite, nous aborderons plus en détail la partie programmation de notre jeu 2048, en expliquant l’architecture du logiciel, les différents points importants du code, et pour finir, la présentation de tous les écrans de jeu.

1. Partie analyse et conception du logiciel
2. Les cas d’utilisation

Ce diagramme de cas d’utilisation a pour but de présenter les différentes fonctionnalités disponibles pour l’utilisateur lors de l’utilisation du logiciel « 2048 ».

Ces fonctionnalités sont celles que nous avons imaginées afin d’offrir le jeu le plus complet et agréable possible. Cependant, par manque de temps, nous n’avons pas pu toutes les implémenter.

Nous allons donc présenter le fonctionnement de ce logiciel ainsi que les différents cas d’utilisation pouvant se présenter à l’utilisateur, dans le cas où nous aurions implémenté toutes les fonctionnalités présentes dans ce diagramme.

Une image contenant texte, diagramme, croquis, dessin

Description générée automatiquement

Diagramme de cas d’utilisation 2048

Lors du lancement du logiciel, l’utilisateur atterrit sur une page d’accueil, via laquelle il peut configurer la partie et/ou s’authentifier.

La configuration permet de choisir la taille de la grille (le nombre de lignes, qui est égal au nombre de colonnes) et un objectif à atteindre pour gagner la partie (obtenir le bloc 2048, 4096, etc.).

De plus, l’utilisateur peut s’authentifier, c’est-à-dire se créer des identifiants (nom d’utilisateur et mot de passe) ou se connecter avec ses identifiants déjà existants.

Dans le cas où l’utilisateur s’authentifie, alors on propose à l’utilisateur la possibilité de reprendre une partie parmi l’ensemble des parties qu’il a sauvegardées.

Après que l’utilisateur a créé une nouvelle partie ou repris une de ses sauvegardes, il commence alors à jouer au 2048.

Il peut alors jouer (déplacer les blocs), sauvegarder sa partie (en lui assignant un nom de sauvegarde), consulter l’aide du jeu, ou encore réinitialiser la partie (remettre la grille à zéro).

De plus, l’utilisateur peut quitter la partie et revenir à l’écran d’accueil. S’il n’a pas sauvegardé sa partie, alors elle sera perdue et il ne pourra pas la reprendre.

L’ensemble de ces fonctionnalités permettent d’avoir un jeu dynamique et respectant les principes de la plupart des jeux existants (authentification, sauvegarde, configuration du jeu, etc.).

1. Le diagramme de classes

Pour coder ce logiciel, nous avons utilisé le langage C++, et en particulier l’API Qt Creator.

Le C++ offre la possibilité d’utiliser la programmation orientée objet, qui implique donc l’implémentation de classes, qui permettent de modéliser diverses entités du logiciel/jeu.

Afin de présenter ces différentes classes et d’avoir une première approche de leur implémentation, nous avons illustrer graphiquement leur structure via ce diagramme de classe.

Une image contenant texte, capture d’écran, diagramme, Police

Le contenu généré par l’IA peut être incorrect.

Diagramme de classe 2048

1. La classe Coordonnees

Cette classe, utilisée par la plupart des autres classes de ce logiciel, permet de modéliser les coordonnées X et Y relativement aux coordonnées de la fenêtre principale « MainWindow » implémentée via Qt Creator.

Ces coordonnées sont exprimées en pixel, et le coin en haut à gauche de la fenêtre principale correspondant aux coordonnées (0, 0).

1. La classe MainWindow

Cette classe permet d’instancier la fenêtre principale affichée à l’écran lors du lancement du programme.

Elle est implémentée via l’API Qt, et dispose donc de caractéristiques propres à Qt, telles que des méthodes pour définir son affichage, capter des évènements du clavier, etc.

C’est dans cette fenêtre que sont affichés tous les éléments du code, comme la grille du jeu et le score.

On attribue à chaque fenêtre principale une largeur et hauteur, qui permettent de définir sa taille lorsqu’elle est affichée à l’écran de l’ordinateur. Si sa taille dépasse les dimensions de l’écran, alors elle ne sera pas affichée dans son entièreté.

1. La classe Bloc et ses classes dérivées BlocNumerote et BlocNul

La classe bloc est une classe abstraite permettant de modéliser les cases du 2048.

Cette classe mère permet de définir les attributs qui sont communs aux différents types de bloc (aux différentes classes héritées).

Par exemple, chaque bloc possède une couleur, des coordonnées, une fonction pour dessiner le bloc à l’écran, etc.

De plus, on attribue à chaque type de bloc un attribut « type » différent. Ce dernier nous permet de retrouver la classe d’un objet Bloc instancié dont on ne connait pas la classe.

La classe héritée BlocNumerote correspond aux cases du jeu possédant une valeur (une puissance de 2), tandis que la classe CaseVide modélise les cases de la grille dans lesquelles il n’y a pas de blocs numérotés.

Les cases vides sont automatiquement affichées en gris, à l’inverse des cases numérotées dont la couleur dépend de leur valeur.

1. La classe Grille

C’est la classe la plus conséquente du projet en termes de code.

Elle représente la grille du jeu, c’est-à-dire un carré composé de cases numérotées ou vides.

Chaque grille possède un nombre de lignes, qui est égal à son nombre de colonnes car la grille est un carré.

De plus, afin d’afficher la grille à l’écran, on lui attribue des coordonnées et une taille, en pixels, qui sont calculés en fonction de la taille de la fenêtre principale.

En termes de code, la grille est modélisée par une matrice carrée de pointeurs vers des blocs (qu’ils soient numérotés ou vides).

La matrice permet une représentation intuitive de la grille (car elle peut être vue comme un carré composé de blocs), tandis que les pointeurs vers des blocs permettent une gestion dynamique de la mémoire.

Enfin, la grille implémente toutes les fonctions permettant le déroulement de la partie, comme le déplacement des blocs, la création de nouvelles cases numérotées, la fusion de deux blocs, l’affichage de la grille du 2048, etc.

1. La classe Score

Enfin, la classe Score permet, comme son nom l’indique, de modéliser le score de la partie en cours.

Ce score est mis à jour via la classe Grille, et augmente à chaque fois que deux blocs numérotés sont fusionnés.

1. Le diagramme de séquences
2. Partie programmation du logiciel

1) L’architecture du logiciel

Afin de versionner et d’avoir une sauvegarde Cloud de notre projet, nous avons décidé de créer un répertoire distant « repository » GitHub.

Ce repository nommé « 2048-project », contient à sa racine plusieurs sous-dossiers ainsi que des fichiers propres à Git comme le « .gitignore », « README », etc.

Le projet Qt Creator est localisé dans le sous-dossier « src », qui contient donc la partie code du repository.

Pour l’ensemble des classes mères et héritées que nous avons implémentées, nous avons créé un couple (fichier d’entête « .h », fichier source « .cpp »).

De plus, ce dossier « src » contient le programme « main.cpp », permettant de lancer le logiciel.

Afin d’ouvrir le projet dans Qt Creator, il faut ouvrir le fichier « 2048-project.pro », permettant de construire le projet.

La prise en main de ce logiciel est expliquée via le fichier « README » inclus à la racine du projet.

1. Les points importants du code

a) La modélisation de la grille en langage informatique

Dans cette partie, nous allons vous expliquer comment nous avons « traduis » la modélisation de la grille en langage informatique (en C++).

Comme indiqué précédemment, la grille est implémentée comme une matrice contenant des pointeurs vers des blocs, permettant une allocation dynamique de la mémoire.

En effet, que ce soit pour placer un nouveau bloc dans la grille, déplacer un bloc numéroté d’une case à une autre, fusionner deux blocs, etc., cette matrice nous permet une gestion de la mémoire efficace et en accord avec la programmation orientée objet.

Afin de respecter toutes les conventions de la POO, nous avons bien redéfini les destructeurs, constructeurs de recopie, etc., de toutes les classes faisant intervenir des pointeurs.

De plus, cela nous facilite l’affichage de la grille via l’interface Qt Creator puisque, la ligne et la colonne de chaque bloc peuvent être vu comme des coordonnées en pixels dans la fenêtre graphique.

En effet, via la méthode « convertirCoordBloc » de la classe Grille, on peut déterminer les coordonnées en pixel d’un bloc à partir de sa ligne et de sa colonne dans la matrice.

1. La gestion des exceptions

De plus, afin de rendre le logiciel le plus « professionnel » possible, nous avons créé de nombreuses classes « exception » afin de s’assurer de la cohérence du code.

Par exemple, des messages d’erreurs s’affichent si un utilisateur essaye d’accéder à un élément de la matrice en indique une ligne et/ou une colonne négative, s’il essaye de créer un bloc numéroté avec une valeur qui n’est pas une puissance de 2 (incohérent avec le principe du jeu 2048), etc.

1. Le déplacement des blocs

L’implémentation du déplacement des blocs à été une étape délicate du code. En effet, la façon de déplacer les blocs de la grille dépend de la direction dans laquelle les déplacer (\*1). Par exemple, si le joueur souhaite déplacer les blocs de droite à gauche, alors :

* Il faut déplacer les blocs colonne par colonne en regardant les colonnes de droite à gauche,
* Il est inutile d’essayer de déplacer les blocs de la colonne tout à droite, car les blocs sont bloqués par le bord de la grille.

Tandis que si le joueur déplace les blocs de bas en haut :

* Il faut déplacer les blocs ligne par ligne, en allant de haut en bas,
* Il est inutile de regarder la ligne du haut.

Ainsi, vous l’aurez compris, il est important de différencier chaque cas de déplacement de blocs (chaque direction).

Afin de ne pas surcharger de code l’implémentation du déplacement des blocs, nous avons décidé de séparer cette partie en deux méthodes : la méthode « deplacerBloc » permettant de déplacer un unique bloc en particulier et la méthode « deplacerBlocs » permettant d’itérer tous les blocs à déplacer de la grille, et ainsi d’appeler la méthode « deplacerBloc » pour chacun de ces blocs.

La méthode « deplacerBlocs » permet de déterminer, en fonction de la direction, l’ordre pour regarder/itérer les blocs de la grille. C’est-à-dire qu’elle permet de déterminer s’il faut itérer les blocs ligne par ligne ou colonne par colonne, dans quel sens les itérer, par quelle colonne/ligne commencer l’itération, etc.

Cela permet de respecter les conditions énoncées plus haut (cf. \*1).

De plus, pour chaque bloc itéré, on regarde si c’est bien un bloc numéroté (on regarde si son type est bien égal à 2), car il est inutile de déplacer un bloc nul (une case vide).

La méthode « deplacerBloc » permet quant à elle de déterminer le déplacement à effectuer sur le bloc numéroté itéré, soit :

* Aucun déplacement : le bloc que l’on regarde est collé à un autre bloc numéroté (suivant la direction choisie) avec une valeur différente,
* Une fusion de 2 blocs : le prochain bloc (suivant la direction choisie) est un autre bloc numéroté avec la même valeur que le bloc itéré,
* Un transfert du bloc : le bloc itéré est bloqué par un autre bloc numéroté avec une valeur différente ou le bord mais il y a au moins une case vide entre le bloc itéré et le bord ou l’autre bloc numéroté.

De plus, cette méthode s’adapte en fonction de la direction choisie. En effet, on n’essaiera pas de déplacer un bloc de la même manière en fonction de la direction.

Comme pour la méthode « deplacerBlocs » la gestion de la direction est effectuée via la commande « switch ».

Dans tous les cas, pour chaque direction, on détermine la colonne et la ligne de la case où on souhaite déplacer/fusionner le bloc, et en fonction de ces variables, on détermine le type de déplacement à effectuer.

Enfin, il est important de noter que ces deux méthodes prennent en paramètre le booléen « test\_finie » (\*2). Ce dernier permet d’indiquer si on appelle ces méthodes pour réellement déplacer un ou plusieurs blocs, ou bien si on les appelle pour savoir si un ou des déplacements sont possibles.

En effet, si ce booléen vaut true, alors les déplacements ne seront pas réellement effectués par ces deux méthodes, mais ces dernières renverront « true » si un ou des déplacement sont possibles, et « false » sinon.

Cela nous sera très utile pour savoir si la partie est terminée ou non, et c’est d’ailleurs ce que l’on va vous expliquer dans la prochaine partie.

1. Savoir quand la partie est terminée

Nous allons donc maintenant vous expliquer comment nous avons gérer la manière de savoir quand la partie est terminée, c’est-à-dire quand la grille est remplie de blocs et qu’aucune fusion n’est possible.

Nous avons créé la méthode « estFinie », qui est appelée pour déterminer si la partie est terminée.

Premièrement, nous avons utilisé la logique qu’il est inutile de regarder si la partie est terminée dans le cas où la grille n’est pas pleine. Pour savoir si la grille est pleine, nous avons créé l’attribut « m\_nbBlocs », qui correspond au nombre de blocs numérotés présents dans la grille à un instant t.

De ce fait, pour savoir si la partie est terminée, il faut appeler la méthode « estFinie », après chaque déplacement de la part du joueur, seulement dans le cas ou la grille est pleine. La grille est pleine si le nombre de blocs numérotés dans cette dernière est égal au nombre de lignes (ou de colonnes) de la grille au carré.

La méthode « estFinie » renvoie donc « true » si la partie est terminée, c’est-à-dire si aucun déplacement est possible, et elle renvoie « false » si au moins un déplacement est possible.

Pour ce faire, nous allons via une boucle « for », essayer de déplacer les blocs dans les 4 directions possibles via la méthode « deplacerBlocs ».

Pour rappel, cette dernière renvoie « true » si au moins un déplacement est possible et « false » sinon.

Afin d’indiquer à la méthode « deplacerBlocs » que l’on souhaite seulement regarder si un déplacement est possible et non réellement déplacer des blocs, on va lui passer en paramètre le booléen « estFinie » = 1 (cf. \*2).

1. La suppression/ajout des blocs de la mémoire

Afin d’être sûr de bien gérer l’allocation dynamique de la mémoire et ne pas oublier de supprimer des blocs de la mémoire, on a défini la méthode « setBloc » qui est appelée à chaque modification de la matrice, soit à chaque déplacement, fusion, ajout, etc., de blocs.

Cette méthode est simple : on lui passe en paramètre le pointeur vers le nouveau bloc à ajouter, une ligne et une colonne de la matrice, elle supprime de la mémoire le bloc actuellement présent dans la case à modifier, et ajoute le pointeur de bloc dans la case en question.

Par exemple, pour la méthode « transfererBloc », alors on va appeler la méthode « setBloc » en lui indiquant le pointeur du bloc à déplacer, et la colonne et la ligne de la case où déplacer ce bloc. Cette case contenant un bloc nul (avant le transfert), la méthode « setBloc » va alors supprimer ce bloc vide de la mémoire, et mettre dans la matrice, à la colonne et la ligne indiquées, le pointeur du bloc à déplacer.

On va ensuite, via la méthode « transfererBloc », instancier un nouveau bloc vide qu’on va introduire, via la méthode « setBloc », dans la case où était le bloc déplacé (avant le transfert).

1. L’ajout d’un nouveau bloc dans la grille

Nous avons également implémenté une méthode pour placer un nouveau bloc numéroté dans la grille, qui sera appelée lors de l’initialisation de la grille et après chaque déplacement dans une direction si au moins un bloc s’est déplacé.

Lorsqu’on place un nouveau bloc, on lui attribue une valeur aléatoire parmi 2 et 4, avec des probabilités respectives de 9/10 et 1/10. Pour choisir « aléatoirement » cette valeur, nous instancions un entier aléatoire entre 0 et 9 compris, s’il est égal à 0, on attribue la valeur 4 au bloc, sinon 2. Cette méthode permet donc de bien attribuer une valeur « aléatoire ».

De plus, ce nouveau bloc est placé aléatoirement dans une des cases vides de la grille (les éléments de la matrice correspondant à des blocs nuls). Pour ce faire, nous connaissons le nombre de blocs qui sont présents à un instant t dans la grille, qui est égal au nombre de blocs total moins le nombre de blocs numérotés de la grille.

Nous allons donc choisir un nombre aléatoire « n\_alea » entre 1 et ce nombre de blocs vides. La case choisie sera alors la « n\_alea »ème case vide de la grille en comptant les cases lignes par lignes puis colonnes par colonnes. Par exemple, la 2ème case de la grille correspond à la case située première ligne et 2ème colonne.

1. L’affichage graphique des différentes entités du jeu

Pour finir, nous allons expliquer comment nous avons programmé l’affichage graphique du jeu.

Par affichage graphique, nous entendons la visualisation du jeu via la fenêtre principale MainWindow développée par Qt. Pour ce jeu, les éléments affichés dans la fenêtre sont le score de la partie, la grille et les blocs qui la composent.

Ces éléments sont affichés via des méthodes implémentées dans les fichiers d’en-tête de leur classe respective. Ces méthodes sont appelées dans la méthode « paintEvent » de la classe MainWindow, qui permet de redessiner la fenêtre principale. On redessine la fenêtre à chaque fois que le joueur effectue un déplacement.

Ces éléments sont modélisés via des rectangles, des objets de type « QRect » au sens de Qt. Pour les dessiner, on utilise la méthode « fillrect », et on peut également écrire du texte dans ces rectangles via la méthode « drawText » (par exemple pour écrire la valeur d’un bloc, le score, etc.).

La taille et les coordonnées sont déterminées conditionnellement à la taille de la fenêtre principale. En effet, le logiciel est implémenté tel que, en modifiant simplement la taille de la fenêtre, alors la taille et la position de tous les éléments soient modifiés. Par exemple, en diminuant la taille de la fenêtre, la taille de la grille sera diminuée (car sa taille est calculée proportionnellement à celle de la fenêtre), la taille des blocs sera diminuée, la police d’écriture de leur valeur diminuera, etc.

Cela permettra donc à quiconque reprendra le code, de ne pas avoir à modifier manuellement les positions et tailles de tous les éléments de la grille à chaque modification de l’affichage graphique.

1. Présentation des écrans