Projet Tchou-Tchou

Manuel développeur

Camille Euvrard
Justine Donnat

Sommaire

TABLE DES FIGURES

INTRODUCTION

Élèves en 4ème année du cycle ingénieur en génie mathématiques et modélisation à Polytech Clermont-Ferrand nous avons conçu un jeu éducatif destiné aux enfants en bas âge nommé **Tchou-Tchou**.

“Tchou-Tchou” est un jeu graphique de type puzzle, codé sous C++ sur l’API Qt qui permet de modéliser et de développer l’application.

Dans le cadre de ce travail, nous avons tout d’abord commencé par modéliser les différentes facettes de notre application en utilisant les diagrammes du langage UML. Durant cette étape nous avons fait des choix de conception et de modélisation afin de faciliter le codage de l’application.

Dans ce manuel, vous trouverez les différentes étapes de notre réflexion matérialisées par des diagrammes UML et les justifications de nos choix de conception.

1. Présentation du jeu

Le jeu consiste à déplacer des rails sur une grille pour combiner les rails et former ainsi un chemin liant l'entrée de la grille à sa sortie. Lorsque le joueur pense avoir trouver ce chemin, il appuie sur un feu. Le train est alors lancé sur le chemin que le joueur aura constitué. Si le chemin est uniforme et arrive à destination, le joueur aura gagné et le feu passera au vert. Dans le cas contraire, le train s'écrasera et le joueur aura perdu la partie.

Lorsque "Tchou-tchou" est lancé, le joueur aura le choix de jouer sur une grille 2x2 ou 3x3, avec des rails de différentes formes : horizontale, verticale et courbés. De plus, la case de départ est indiquée par la position du train et la case d'arrivée par un drapeau rouge. Le joueur peut échanger toute case avec la case vide, en appuyant sur celle-ci. Le joueur cherchera alors à créer un chemin du point de départ au point d'arrivée.

2. Partie analyse et conception du logiciel

Dans cette partie, nous allons expliciter au moyen de différents diagrammes UML [2], la conception du logiciel telle que nous l'avons imaginée.

2.1. Les cas d'utilisation

Tout d'abord nous avons réalisé un diagramme de cas d'utilisation afin de comprendre au mieux les limites du système et ses relations avec l'environnement. De plus, il permet de faire ressortir les acteurs et les fonctions offertes par le système.

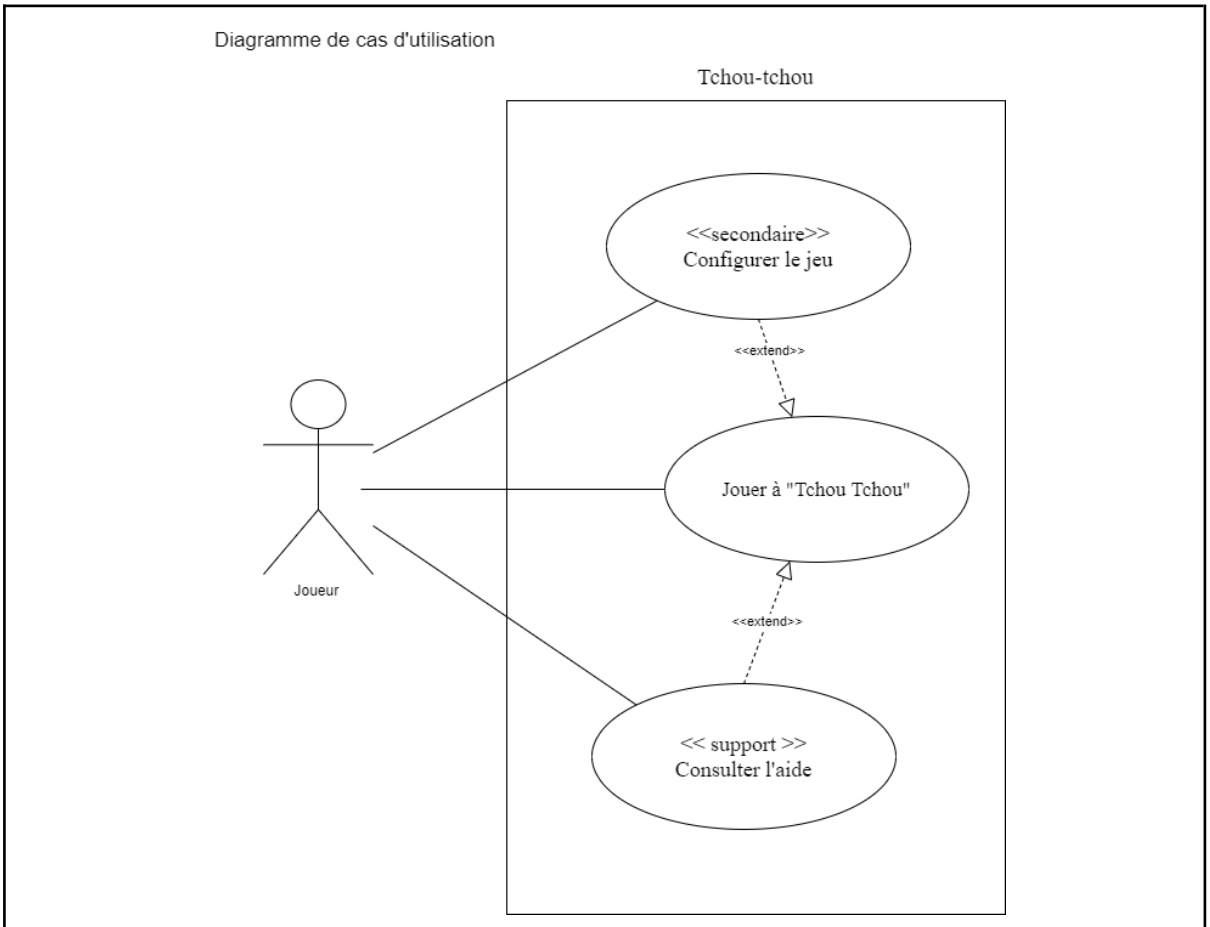

Fig 1 : Diagramme de cas d'utilisation

Quand le joueur démarre le jeu, il a le choix entre commencer à jouer à Tchou-Tchou, configurer le jeu et consulter l'aide.

Cas d'utilisation "Jouer Tchou-Tchou" : Le joueur peut faire une partie. Il lance le jeu Tchou-Tchou avec la configuration par défaut (grille 2x2).
Le joueur n'a plus accès à la configuration une fois le jeu lancé.

Cas d'utilisation “Configurer Jeu” : Le joueur peut configurer le jeu. Il peut choisir la taille de la grille, c'est-à-dire la difficulté (2x2 ou 3x3).

Cas d’utilisation “Consulter l’aide” : Le joueur peut consulter l’aide. Il peut accéder à une fenêtre tutoriel simple lui rappelant les règles du jeu.

2.2. Analyse (les diagrammes de haut niveau)

2.2.1. Les diagrammes de séquence

Nous avons créé un diagramme de séquence qui explique comment l'utilisateur joue une partie de Tchou-Tchou. Ce diagramme sert à décrire les interactions entre les objets en insistant sur l'aspect temporel.

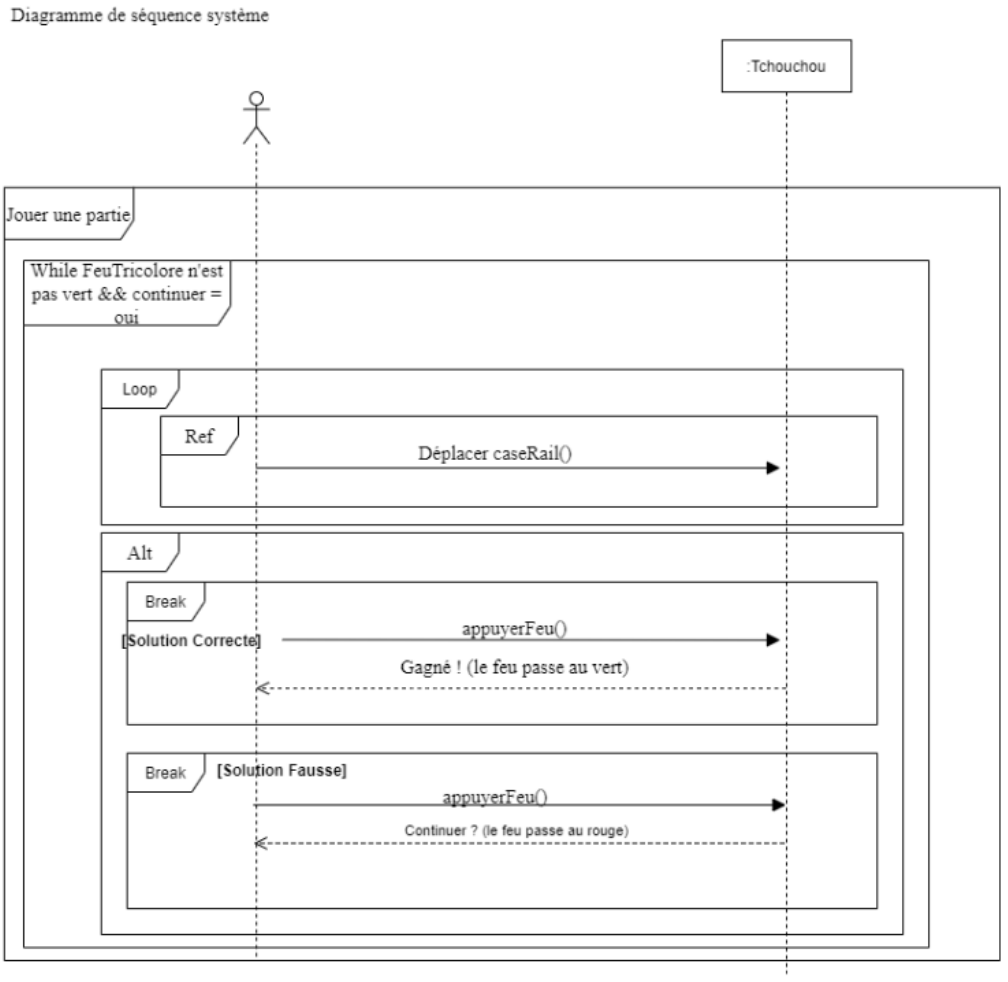

Fig 2 : Diagramme de séquence système de "Jouer une partie"

On souhaite pouvoir déplacer des cases tant que le feu est éteint et que le joueur souhaite continuer à jouer. Quand le joueur pense avoir la bonne solution, il appuie sur le feu. Il gagne si le feu devient vert. Si le feu devient rouge, cela veut dire que son train est bloqué, il a alors le choix de recommencer une partie ou d'arrêter de jouer.

2.2.2. Les diagrammes de classes

Nous avons créé un diagramme de classe d'analyse qui présente le type d'objet que nous utilisons. Il permet également de comprendre les liaisons entre les différentes classes.

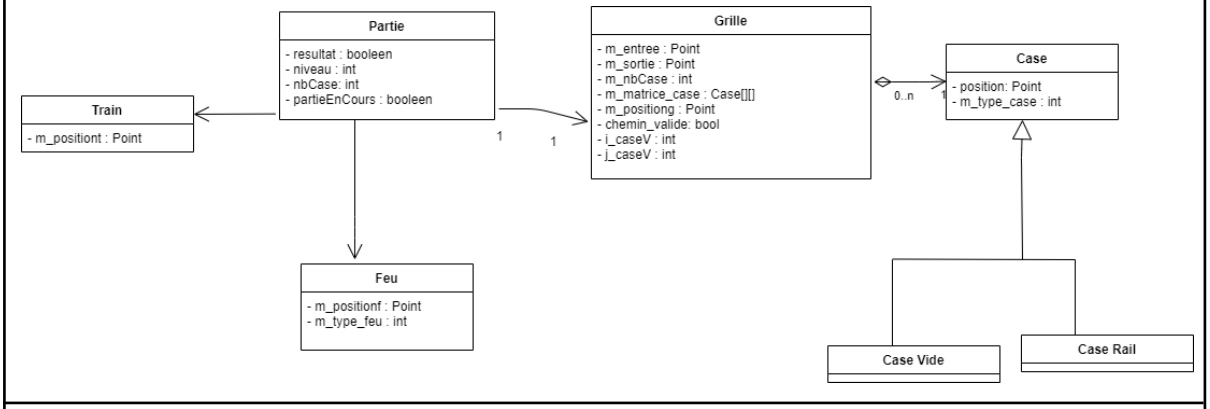

Fig 3 : Diagramme de classe d'analyse

Une partie de Tchoutchou est composée d'une grille, elle-même composée de cases, d'un feu et d'un train. Une case peut être une case rail, ou une case vide.

Classe Grille:

- **Attributs:**
  - m_entree et m_sortie : les coordonnées de l'entrée et de la sortie de la grille que l'utilisateur devra relier.
  - m_nbCase : le nombre de case par lignes et par colonne de la grille
  - m_matrice_case : Stocke les cases de la grille
  - m_positiong : les coordonnées de la grille dans la fenêtre de jeu
  - chemin_valide : true si le chemin emmène le train jusqu'à la sortie
  - i_caseV : ligne de la case vide dans la matriceCase
  - j_caseV : colonne de la case vide dans la matriceCase

Classe Case:

Le jeu est composé de deux types de cases : les cases rail et la case vide.

- **Attributs:**
  - position : les coordonnées de la case dans la fenêtre de jeu
  - m_type_case : type de la case (Fig 7)
    - 0 : case vide
    - 1 : case rail traversé horizontal
    - 2 : case rail traversé vertical
    - 3 : case rail coin bas droit
    - 4 : case rail coin bas gauche
    - 5 : case rail coin haut droit
    - 6 : case rail coin haut gauche

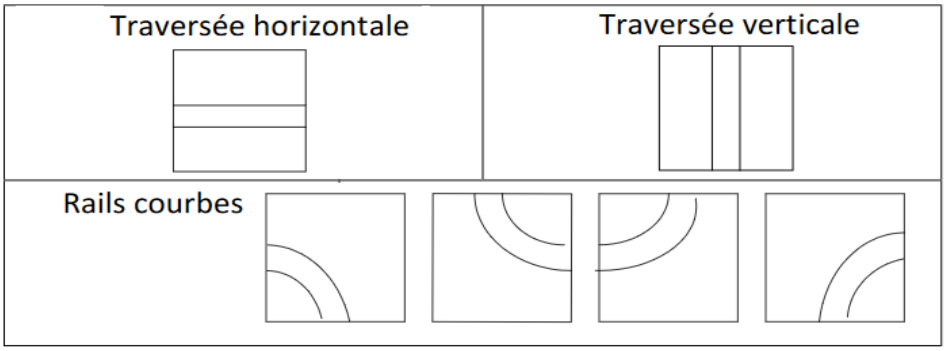

Fig 4: Différents types de rails

Classe Feu:

- **Attributs:**
  - m_ positionf : les coordonnées du feu dans la fenêtre de jeu
  - m_type_feu: entier permettant d'indiquer la couleur du feu
    - 0 : le feu est éteint
    - 1 : le feu est vert
    - 2 : le feu est rouge

Classe Train:

- **Attributs:**
  - m_positiont: coordonnées du train dans la fenêtre de jeu

Classe Partie:

- **Attributs:**
  - résultat : Vrai si la partie est gagnée, faux si le joueur n'a pas de chemin valide
  - niveau : niveau de la partie (facile, moyen, difficile)
  - nbCase : indique le nombre de cases par ligne et par colonne de la grille
  - partieEnCours : retourne Vrai si le joueur continue à jouer, faux si il décide d'arrêter.

Les différentes méthodes des classes sont détaillées lors de la présentation du diagramme de classe de conception (Fig 6).

2.3. Conception (les diagrammes de bas niveaux)

2.3.1. Les diagrammes de séquence

Nous allons détailler les diagrammes de séquence haut niveaux que nous avons créés. Nous présenterons le diagramme de séquence de la fonction *appuyerFeu* (Fig 5) et le diagramme de séquence de *deplacerCase* (Fig 6).

Événement AppuyerFeu(), diagramme de séquence :

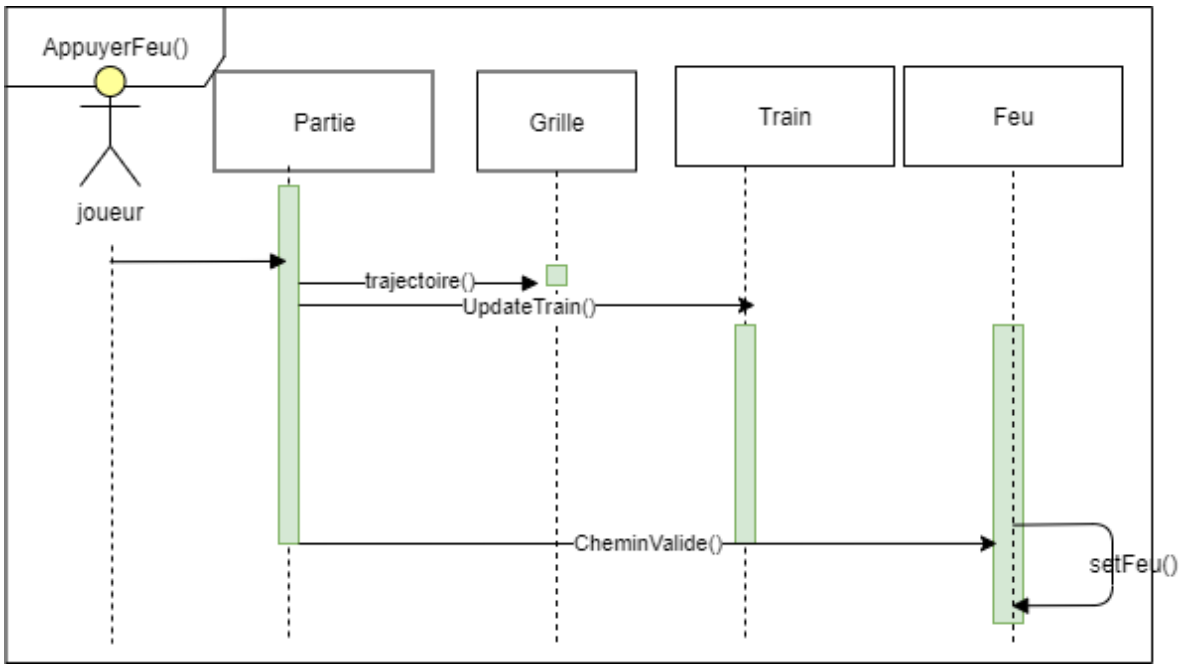

Fig 5: Diagramme de séquence de appuyerFeu

Lorsque le joueur pense avoir trouvé un chemin valide reliant le départ à l'arrivée, il appuie sur le feu.

On commence par établir la trajectoire que le train va emprunter grâce à partie.trajectoire(). Cette fonction va aussi permettre de changer l'attribut résultat de partie à true si le chemin permet au train de se rendre jusqu'à la sortie.

Puis, le train est lancé, même si le chemin est incorrect et la fonction *updateTrain* est utilisée pour faire rouler le train.

Une fois que le train s'est arrêté, on appelle *cheminValide* qui nous renvoie le resultat de la partie. Si le resultat est true, la fonction *setFeu* est appelée pour passer le feu au vert. Si le resultat est false, la fonction *setFeu* est appelée pour passer le feu au rouge.

Une autre fonction que nous avons trouvé intéressante à expliciter est la fonction *deplacerCase*. C'est ce que nous allons donc faire dans la suite avec un diagramme de séquence (Fig 5).

Événement deplacerCase, diagramme de séquence :

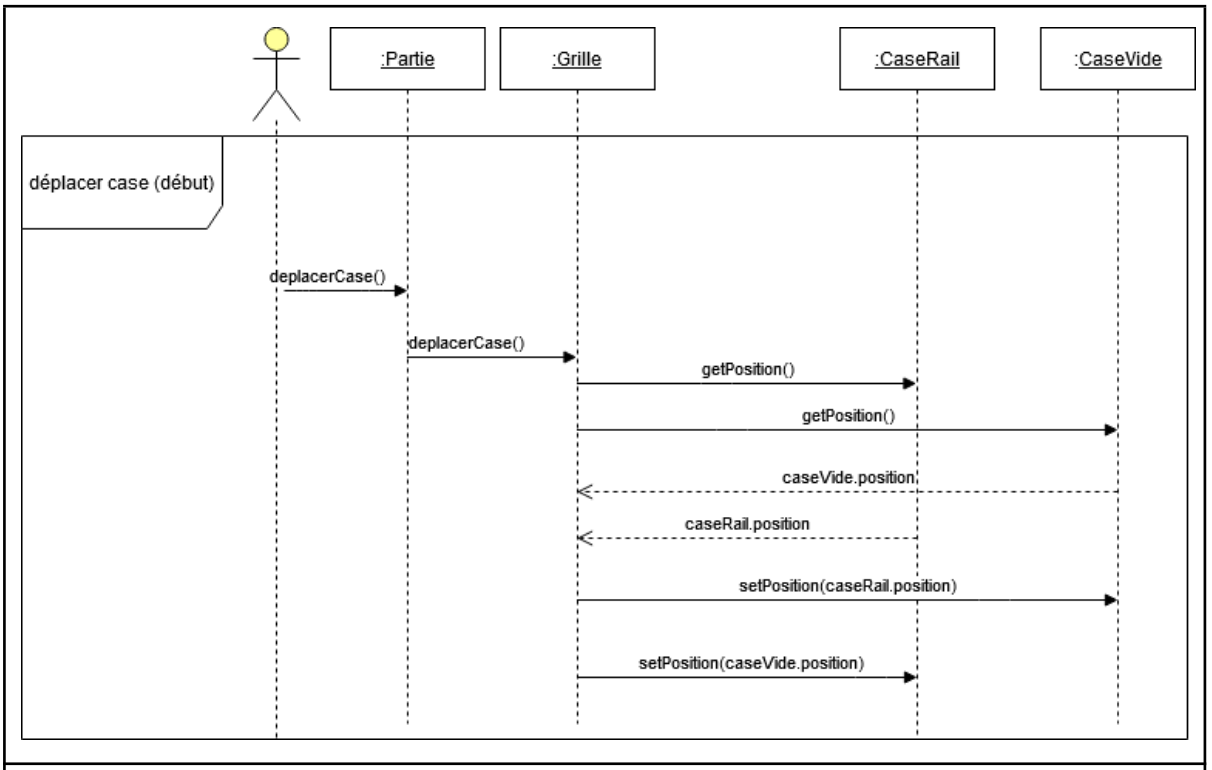

Fig 6 : Diagramme de séquence de deplacerCase

Lorsque le joueur double clique sur une case, l'évènement déplacer Case est déclenché.
La Grille récupère les positions de la case vide et de la case sur laquelle l'utilisateur a cliqué avec les fonctions *getPosition* de Grille. Ensuite, on va échanger les 2 positions des cases avec les fonctions *setPosition* de Grille. L'ancienne position de la case vide devient la position de la case sélectionnée, et inversement.

2.3.2. Les diagrammes de classes

Dans cette partie nous allons présenter le diagramme de de classe de conception (Fig 6) de notre projet.

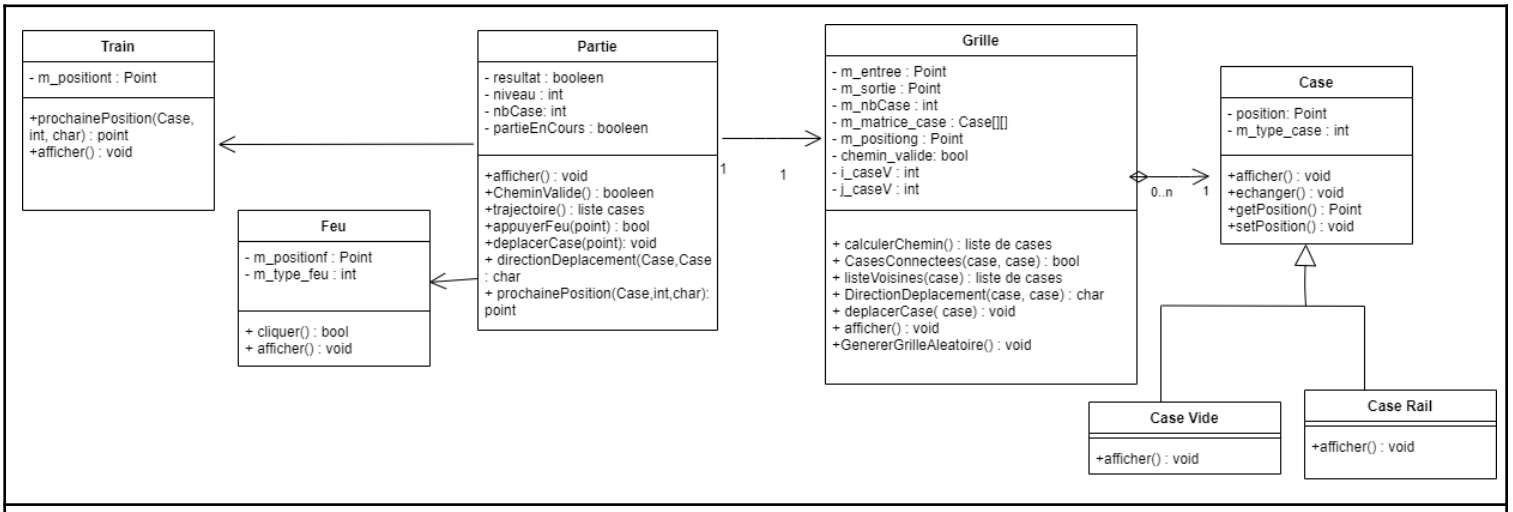

Fig 7 : Diagramme de classe de conception

Classe Grille:

- **Méthodes:**
  - *calculerChemin* : renvoie une liste de cases correspondant au chemin que doit prendre le train
  - *deplacerCase* : échange la position de la case passée en paramètre avec celle de la case vide
  - *genererGrilleAleatoire* : à partir de liste de chemins possibles (donc de cases obligatoires pour chaque chemin), une est tirée aléatoirement puis on mélange la liste en lui ajoutant des cases aléatoires.
  - *caseVoisine* : est utilisée par *calculerChemin*. Renvoie une liste de toutes les cases adjacentes à une case passée en paramètre.
  - *casesConnectees* : est utilisé par *calculerChemin*. Vérifie si le train pourrait se déplacer entre deux cases passées en paramètre.
  - *directionDeplacement* : pour deux cases passées en paramètre, renvoie un caractère (‘h’, ‘b’, ‘d’, ‘g’) qui correspond à la direction du déplacement que le train pourrait effectuer entre les cases.
  - *afficher* : affiche la grille dans la fenêtre de jeu.

Classe Case:

- **Méthodes:**
  - *afficher* : affiche les cases sur la grille.
  - *getPosition* : accesseur nous permettant d'obtenir la valeur de la position
  - *setPosition* : mutateur qui nous permet de modifier la valeur de l'attribut position

Classe Feu:

- **Méthodes :**
  - *afficher* : affiche le feu dans la fenêtre de jeu
  - cliquer : renvoie si le feu a été cliqué ou non

Classe Train:

- **Méthodes:**
  - *prochainePosition* : calcule et renvoie la prochaine position du train dans la fenêtre de jeu
  - *afficher* : affiche le train dans la fenêtre de jeu

Classe Partie:

- **Méthodes:**
  - *afficher* : appelle la méthode afficher() de Grille
  - *cheminValide* : retourne la valeur resultat de Partie
  - *trajectoire* : renvoie une liste de cases que doit emprunter le train pour suivre le chemin
  - *appuyerFeu* : renvoie true si le feu a été cliqué, valeur utilisée pour lancer le train (Fig 4)
  - *deplacerCase* : permet d'échanger la place de la case en passée en paramètre avec celle de la case vide
  - *directionDeplacement* : permet de retourner la direction dans laquelle doit se diriger le train lorsqu'il passe de la première case passée en paramètre à la deuxième case passée en paramètre
  - *prochainePosition* : appelle *prochainePosition* de train, renvoi la prochaine position du train

3. Partie programmation du logiciel

Dans cette partie, le but est d'expliciter ce que l'on a réalisé lors de la programmation du logiciel en Qt et c++.

3.1. L'architecture du logiciel

Dans cette première sous-partie, notre but est de présenter la construction architecturale du logiciel.

3.1.1. Schéma de l'arborescence du logiciel

Tout d'abord, nous présenterons le schéma de l'arborescence du logiciel (Fig 7).

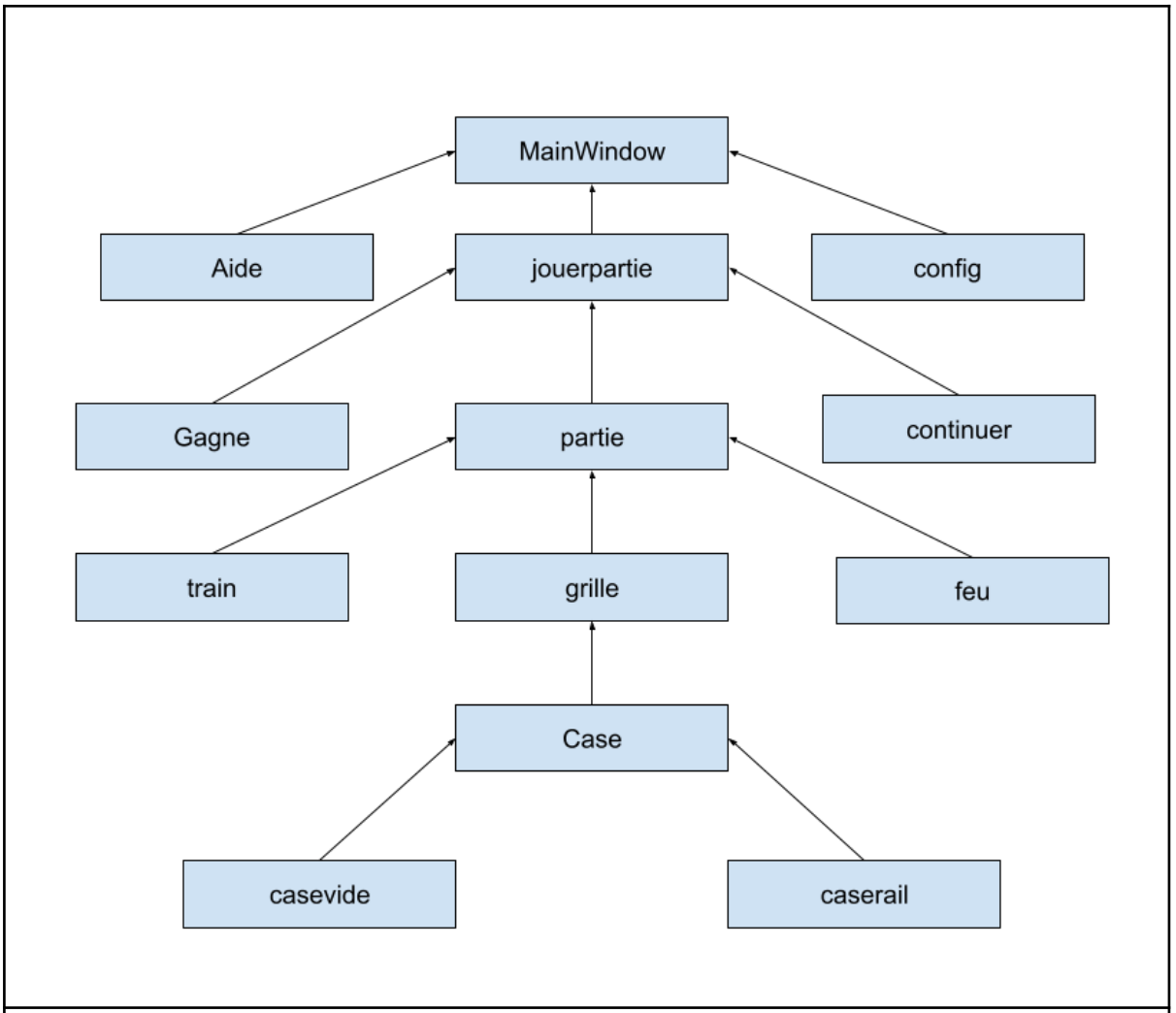

Fig 8 : Schéma de l'arborescence du logiciel

Comme nous pouvons le voir sur le schéma ci-dessus, notre logiciel est composé d'un certain nombre de fichiers.

En premier lieu, nous retrouvons la **MainWindow**, c'est-à-dire la fenêtre principale, qui correspond à notre écran d'accueil.

Puis nous retrouvons les trois fichiers correspondant à nos cas d'utilisation : la **fenêtre d'aide**, la **fenêtre de configuration** du niveau du jeu et enfin **jouerpartie** qui est la fenêtre dans laquelle l'utilisateur peut jouer à TchouTchou.

Le fichier ***jouerpartie*** à besoin d'un certain nombre d'autres fichiers pour fonctionner correctement. Deux fenêtres optionnelles lui sont liées : les fenêtres qui s'ouvrent pour informer l'utilisateur qu'il a ***gagné*** ou **perdu**. La **classe partie** est aussi inclue dans le fichier de ***jouerpartie*** car elle regroupe toutes les autres classes et les méthodes utiles au bon déroulement d'une partie.

La **classe partie** regroupe donc plusieurs classes importantes à la mise en place du jeu : la **classe train**, la **classe feu** et le **classe grille** qui elle-même a besoin d'une autre classe pour son fonctionnement.

La **classe grille** a donc besoin de la **classe abstraite Case** car la grille est composée de cases.

La **classe Case** est une classe abstraite qui est la classe mère des classes ***caserail*** et **casevide**.

C'est donc de cette façon que notre logiciel est organisé, il est important maintenant d'expliquer comment fonctionne notre logiciel.

3.1.2. Description des principales fonctionnalités

Nous allons donc expliciter les différentes classes et fenêtres de notre programme en détaillant quelles méthodes de chacune d'elles sont particulièrement importantes au fonctionnement de notre logiciel.

- **Fenêtre MainWindow** :

Elle ne nous sert que d'écran d'accueil pour le joueur, elle est composée de trois boutons qui lorsque cliqué ouvrent de nouvelles fenêtres.

  - Le bouton aide : qui ouvre le **fenêtre aide**, celle-ci expose à l'utilisateur le fonctionnement et les règles du jeu.
  - Le bouton jouerpartie : qui ouvre la **fenêtre jouerpartie** (celle-ci sera explicitée dans la suite)
  - Le bouton configuration : qui ouvre la **fenêtre configuration** qui permet à l'utilisateur de sélectionner le niveau de la partie à venir. Pour cela il peut cocher niveau 1 ou niveau 2.

- **Fenêtre jouerpartie** :

Elle nous sert à jouer à une partie de TchouTchou pour cela elle utilise la **classe partie**. Mais aussi un certain nombre de méthodes liées à l'aspect graphique du logiciel.

- La méthode PaintEvent : qui permet l'affichage de tous les éléments graphiques comme la grille, le train, le feu ou encore le drapeau indiquant la sortie de la grille
- La méthode MousePressEvent : qui permet de récupérer les endroits de la fenêtre où l'utilisateur a cliqué, utile notamment pour déplacer les cases ou encore pour savoir quand l'utilisateur pense avoir trouvé le bon chemin (il clique sur le feu).
- La méthode updateTrain : qui permet le déplacement du train mais aussi le changement de couleur du feu ainsi que l'ouverture des **fenêtres gagné** ou **continuer**
  - **La fenêtre gagné** ouvre un gif signifiant au joueur qu'il a gagné la partie
  - **La fenêtre continuer** indique au joueur qu'il a perdu et lui demande s'il veut continuer la partie

  C'est aussi cette méthode qui permet au joueur de continuer la partie s'il a perdu et décidé néanmoins de continuer à jouer.

- **Classe partie** :

Elle réunit toutes les classes nécessaires au bon fonctionnement de la partie, c'est-à-dire la **classe grille**, la **classe train** et la **classe feu**. Elle comporte un certain nombre de méthodes mais qui sont juste des méthodes faisant appel aux méthodes des différentes classes dont elle réunit les caractéristiques.
Son constructeur est néanmoins intéressant à détailler. Il prend en paramètre le niveau de la partie et permet de donner la position correcte de la sortie de la grille. Il permet aussi d'initialiser les différents objets venant des autres classes : la grille, le train et le feu.

- **Classe grille** :

Elle permet de manipuler le plateau de cases du jeu. Elle est composée de 8 attributs, de constructeurs, d'un destructeur, de 7 méthodes et d'accesseurs et mutateurs.
Nous allons détailler son constructeur ainsi que certaines méthodes.

- Constructeur de grille : Il prend en argument l'entrée, la sortie, la position de la grille dans la fenêtre, ainsi que le nombre de cases. Nous parcourons le nombre de cases afin de créer une matrice de pointeurs de cases. Nous appelons ensuite la méthode ***genererGrilleAleatoire***. Cette dernière a pour but de remplir la matrice de cases rails (et d'une case vide) de sorte à ce qu'il y ait un chemin valide qui relie l'entrée à la sortie. Il y a quatres combinaisons de cases possibles pour couvrir tous les chemins. On va générer un nombre aléatoirement entre 1 et 4 pour simuler le tirage au sort d'un chemin. En fonction du chiffre obtenu, on va ajouter à une liste les types des cases correspondantes au chemin associé à ce numéro. A noter qu'avec la même combinaison de cases il est possible de faire plusieurs chemins.
  Nous mélangeons ensuite la liste de types pour que la grille change à chaque partie.

Une fois la liste de types de cases obtenue, nous allons pourcourir la matrice et créer des cases rails et la case vide, en fonction de l'emplacement où on se trouve dans la matrice. Nous utilisons la liste de cases pour donner un type aux cases rails dans la matrice.

- deplacerCase : Notre méthode *deplacerCase* permet d'échanger la case vide et la case rail dans la grille. Pour cela, elle prend en argument la position du curseur. Elle regarde sur quelle case se situe le curseur et parcourt la matrice jusqu'à cette case. Elle va ensuite sauvegarder la position et le type de la case à échanger et va créer une nouvelle case vide et une nouvelle case rail en échangeant les positions de ces dernières. Elle met à jour l'emplacement de la nouvelle case vide.
- directionDeplacement : La méthode *directionDeplacement* prend en argument deux cases: c1 et c2. Elle a pour but de donner la direction dans laquelle se trouve la case c2 par rapport à la case c1: droite, gauche, haut ou bas. Elle initialise un entier qui prend un certaine valeur si la case c2 esst à droite/à gauche/en bas/en haut. Nous avons fait en sorte que les quatre valeurs soient différentes. En fonction de cet entier, on retourne une direction.
- listeVoisines : La méthode *listeVoisines* prend en argument une case. Elle a pour but de retourner une liste contenant toutes les cases voisines de cette case sur la grille.
- CasesConnectees : La méthode *CasesConnectees* prend deux cases c1 et c2 en argument. Elle a pour but de renvoyer true si la case c2 est connectée à la case c1, c'est-à-dire que le chemin est valide entre les deux cases, et false sinon. Pour cela elle va prendre en compte le type des deux cases, ainsi que la direction de la case c2 par rapport à la case c1.
- calculerChemin : la méthode *calculerChemin* renvoie une liste de cases qui composent un chemin valide commençant par l'entrée de la grille. On initialise une case courante à l'entrée de la grille. Si la case à l'entrée de la grille est valide, c'est à dire qu'elle est connectée au bord gauche de la grille, et que le rail ne mène pas vers l'extérieur, on l'ajoute à la liste de cases. Si ce n'est pas le cas, on ne continue pas et on renvoie une liste vide. De manière générale, on récupère dans une liste les cases voisines de la case courante, on regarde si celles-ci sont connectées à la case courante et si elles n'appartiennent pas à la liste de cases déjà visitées. Si ces deux conditions sont réunies, on ajoute la case voisine en question à la liste de cases visitées, et elle devient la case courante. Si ce n'est pas le cas, on sort de la boucle et la fonction renvoie la liste de cases visitées (le chemin n'est donc pas entièrement valide car il ne relie pas l'entrée à la sortie). Si nous somme sur la case de la sortie et que le rail mène vers l'extérieur droit de la grille, on ajoute la case de sortie à la liste de cases visitées. Auquel cas le chemin est entièrement valide, on retourne le chemin de cases visitées.

- **Classe train** :

Elle permet la manipulation du train, c'est-à-dire la manipulation de sa position mais aussi son affichage.

- **Classe feu** :

Elle permet la manipulation du feu, c'est-à-dire sa position, sa couleur, son affichage dans la fenêtre.

- **Classe abstraite Case** :

Elle permet de donner la position de n'importe quelle case et son type. Elle comporte la méthode virtuelle *afficher* qui est explicitée dans ses classes filles

  - **Classe caserail** : représente les cases qui comportent des rails, elle permet leur affichage selon leur position et leur type
  - **Classe casevide** : représente la case vide (qui ne comporte pas de rail), elle permet son affichage selon sa position

3.2. Présentation des écrans

Dans cette sous-partie, nous présenterons les différentes fenêtres graphiques que nous obtenons à la suite de la programmation ainsi que les différents éléments de Qt [1] [3] utilisés pour la mise en place de celles-ci.

- **Fenêtre d'accueil**

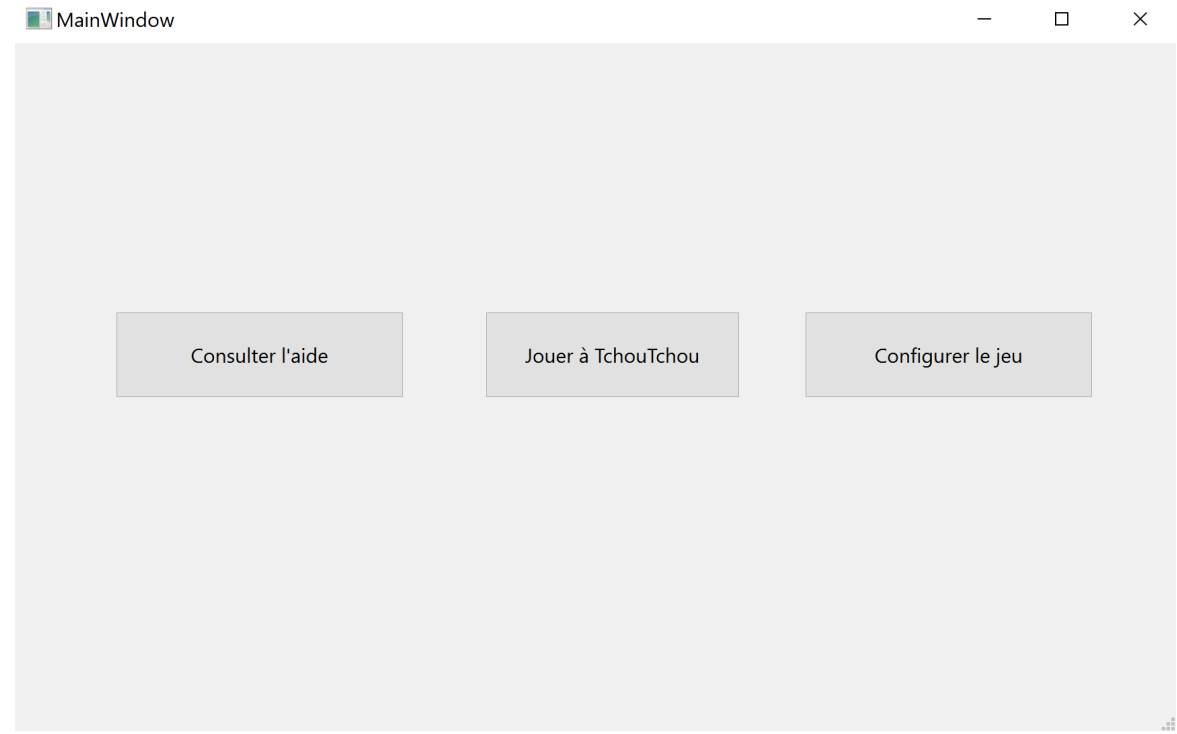

Fig 9 : Fenêtre d'accueil

Pour réaliser cette fenêtre, nous avons :

➔ Créé trois QPushButton
➔ Lié ces trois boutons à l'ouverture de trois nouvelles fenêtres lorsqu'ils étaient cliqués

- **Aide**

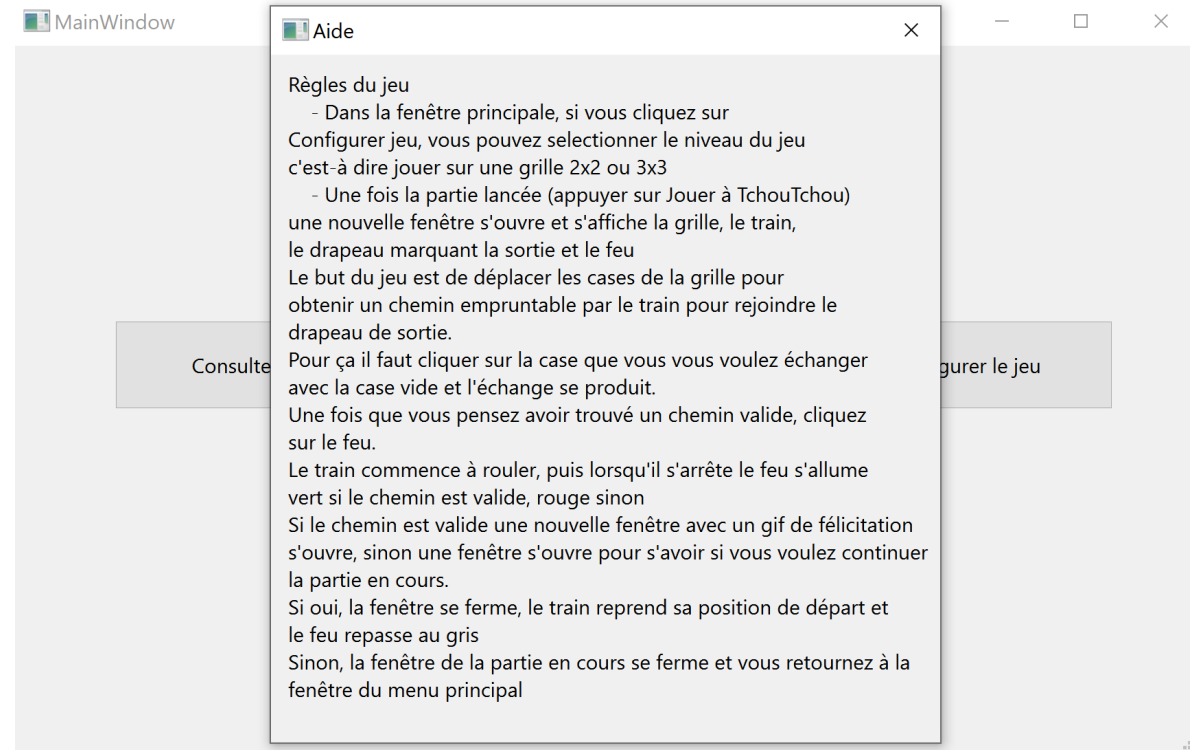

Fig 10 : Écran d'aide

Pour réaliser cette fenêtre, nous avons :

➔ Créé un QLabel où nous avons inclus le texte correspondant aux règles de notre jeu

- **Fenêtre de configuration**

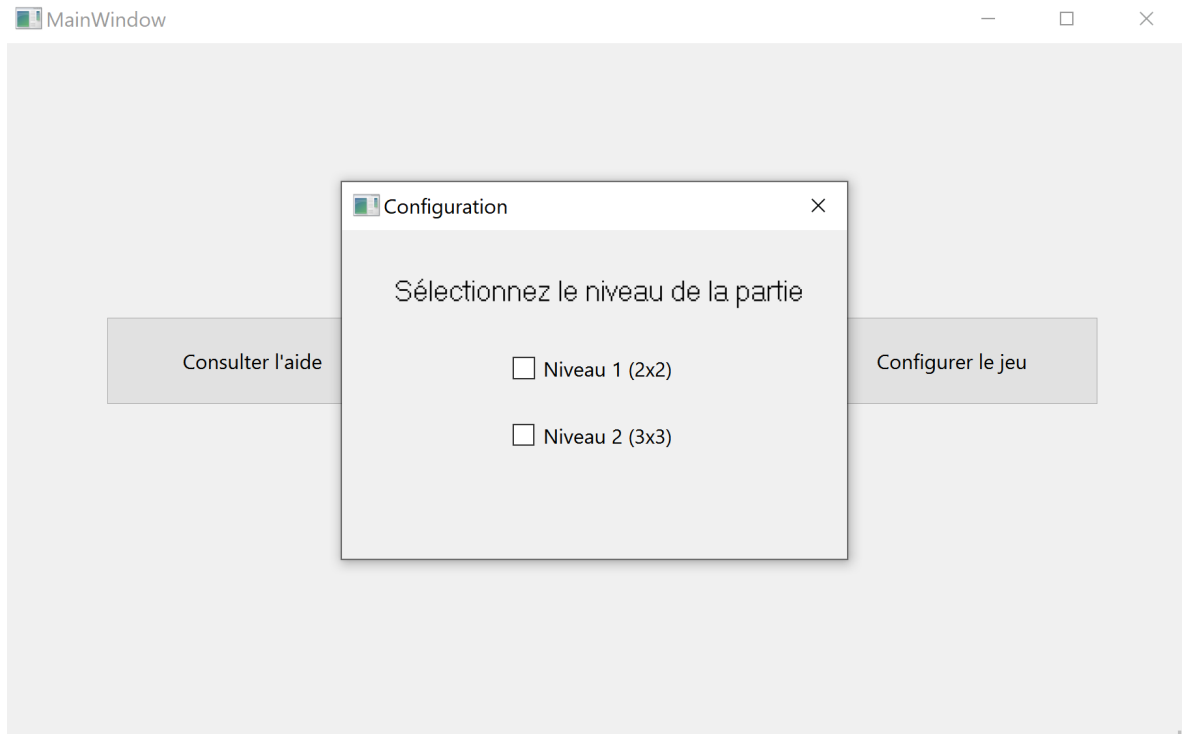

Fig 11 : Écran de configuration

Pour réaliser cette fenêtre, nous avons :

- ➔ Utilisé un Qlabel pour afficher l'instruction "Sélectionnez le niveau de la partie"
- ➔ Utilisé des QCheckBox pour créer les éléments cochables ainsi que des Qlabel pour expliciter à quoi ils correspondaient

- **Jouer à TchouTchou**

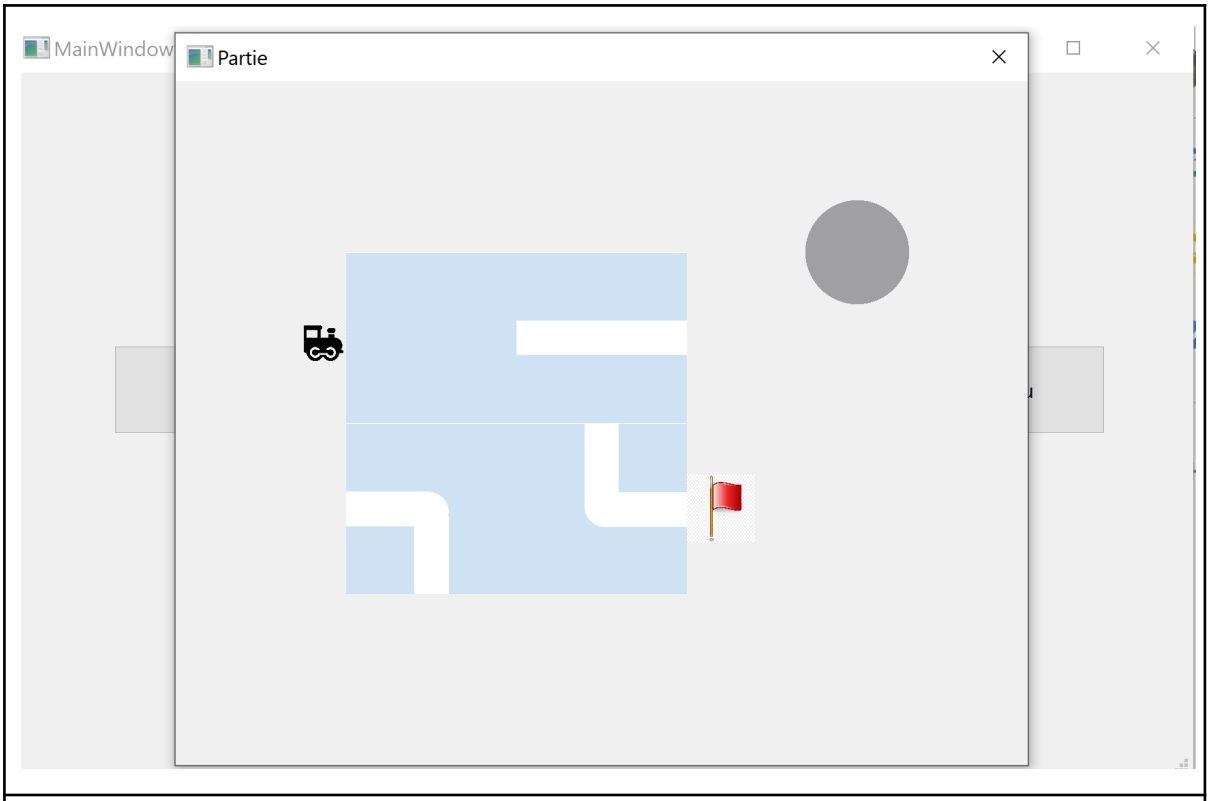

Fig 12 : Écran de la partie

Pour réaliser cette fenêtre, nous avons :

- ➔ Utilisé une méthode PaintEvent pour afficher tous les éléments graphiques : la grille, le train, le feu et le drapeau rouge indiquant la sortie
- ➔ Utilisé une méthode MousePressEvent pour récupérer la position où l'utilisateur cliquait dans la fenêtre afin de pouvoir déplacer les cases ou lancer le train (lorsqu'il cliquait sur le feu)
- ➔ Utilisé une méthode updateTrain qui permettait le déplacement du train ainsi que le changement de couleur du feu et l'ouverture de la fenêtre perdu ou gagné suivant le resultat de la partie
- ➔ Utilisé la classe partie pour exploiter toutes les méthodes liées à la grille, le train ou le feu

- **Gagné**

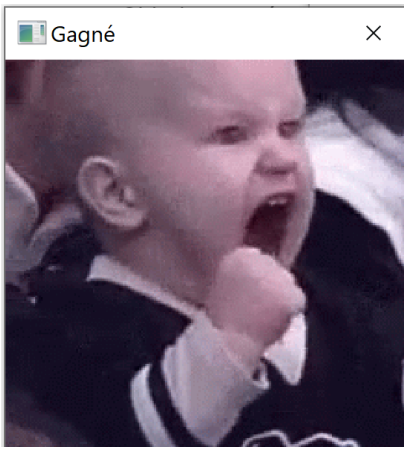

Fig 13 : Écran de victoire

Pour réaliser cette fenêtre, nous avons :

- ➔ Utilisé un QMovie pour l'affichage du gif
- ➔ un QTimer pour lancer le mouvement du gif

- **Continuer**

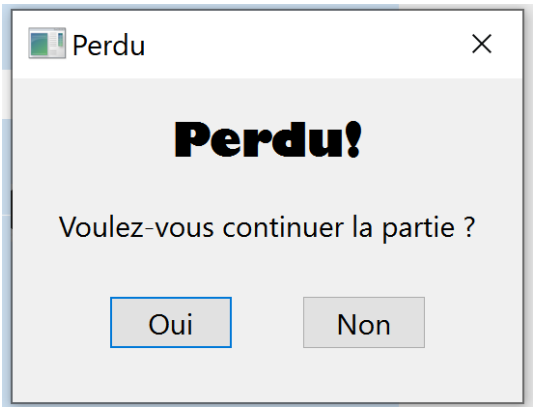

Fig 14 : Écran de défaite

Pour réaliser cette fenêtre, nous avons :

- ➔ Utilisé des QLabel pour annoncer au joueur qu'il avait perdu ainsi que pour lui demander s'il voulait continuer
- ➔ Utilisé des QPushButton pour capter la réponse du joueur

4. Bibliographie

[1] « Qt Documentation ». https://doc.qt.io/.
[2] « Support cours UML», Marinette Bouet, Christophe De Vaulx.
[3] « Développement d'applications graphiques en C++ à l'aide de la bibliothèque graphique Qt», Marinette Bouet, Christophe De Vaulx.
[4] « Programmation Orientée Objet, Le langage C++», Marinette Bouet, Christophe De Vaulx.