Rapport projet C++ M1 2022

-

Dylan KELLER – 21961014 – M1 SDD
Paul GANGNEUX – 21958573 – M1 GENIAL

-

Ce document constitue notre rapport et compte rendu concernant le projet de Langages Objet Avancés (C++) pour le premier semestre du M1 Informatique de l'Université Paris Cité, année 2022-2023. Voici ce qu'il contient :

Sommaire

Eléments traités

Nous avons respecté et rempli le cahier des charges. Notre code accueille l'utilisateur dans un menu où il a le choix parmi trois jeux : Dominos, Trax, et Carcassonne.

Ces derniers ont été implémentés et fonctionnent complètement et correctement. Le projet entier tourne sur interface graphique, basé sur SFML.

Le menu ne comporte pas de section « options » car cela n'a pas été jugé nécessaire. Le programme gère les changements de résolution lui-même, par exemple.

Description de la vue

La boucle principale de notre programme se trouve dans la fonction main, et, dans l'ordre, elle effectue les actions suivantes:

- Traite les inputs
- Change l'état du modèle en fonction des inputs
- Met à jour l'affichage en fonction du modèle

Pour ce faire on a créé la classe DrawableState (dans le répertoire gameview) qui implémente les fonctions:

```
virtual int handleEvents(sf::Event& event);
virtual void changeState();
virtual void draw();
```

Ainsi, tous les écrans affichés (jeux, menus) sont hérités de cette classe, et la boucle principale peut appeler ces fonctions.

handleEvents peut renvoyer des valeurs spéciales qui peuvent avoir comme effet de dire au main de quitter le programme, ou de changer de DrawableState (pour passer d'un menu à un jeu par exemple)

Pour les menus, il n'y a aucun modèle complexe et l'état est défini côté vue, mais pour les jeux, la classe GameView s'occupe de faire le lien entre modèle et vue. lorsque changeState() est appelé, des fonctions du modèle sont exécutes, et les variables gérant l'affichage sont modifiés en fonction des valeur de retour de ces fonctions. Ainsi, c'est toujours GameView qui appelle des fonctions du modèle, et jamais l'inverse.

Pour l'affichage d'objets (texte, tuiles, image, autre), on a créé une classe DrawObject, dont le but premier et de pouvoir lier les objets affichable pour que leur transformations soir hérités de l'objet parent. ainsi, si un objet A est parent de B, et qu'on applique une rotation à A, alors cette rotation sera aussi appliquée à B (dans les faits, au moment d'afficher B, l'objet compose la transformation finale à partir de sa transformation locale et de la combinaison de celle de ses parents).

Cela permet par exemple aux dominos d'avoir des nombres affichés qui suivent la tuile, ou même à toutes les tuiles posées de se déplacer quand on déplace un objet parent qui sert de caméra.

Tous les objets affichables héritent de DrawObject, ce qui permet de redéfinir des opérations comme rotation / translation qui pourraient vouloir se comporter différemment en fonction de l'objet (par exemple, quand on fait une rotation sur un domino, on ne veut pas que la rotation soit appliquée aux nombres qui y sont liés)

Tous les DrawableState ont un objet qui reste au centre de l'image même quand la fenêtre est redimensionnée, et dont n'importe quel nouvel objet peut être enfant. Cela permet d'avoir un affichage plus dynamique et joli.

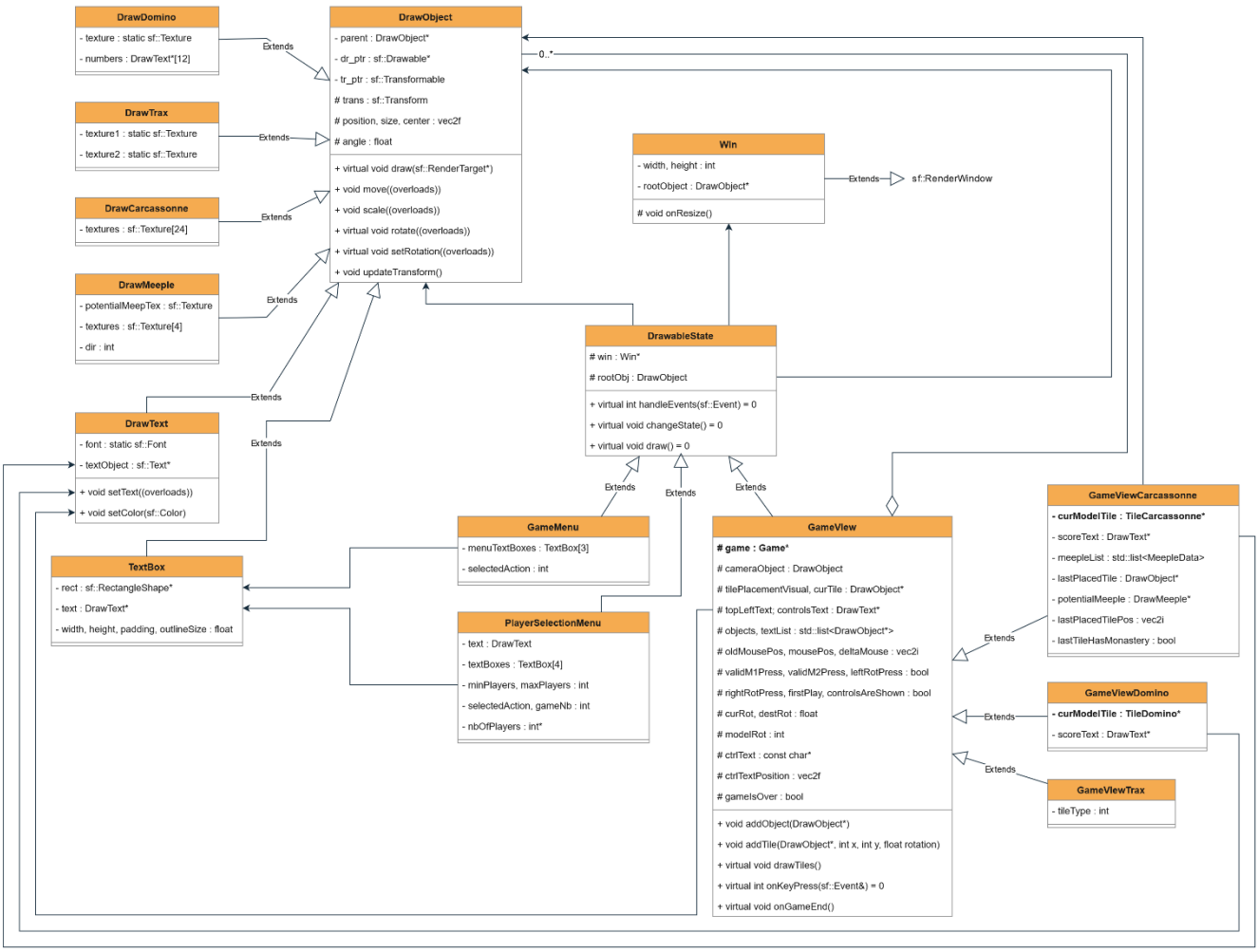

diagramme UML de la vue

Notes sur l'UML : lorsqu'une classe en a une autre en attribut (donc une simple relation 1:1) on s'est contenté d'une flèche noire sans cardinalité pour ne pas polluer le diagramme.
Une version plus grande des deux UML est disponible dans le dépôt.

Description du modèle

Les principales classes du modèle sont Game et Tile. La logique du jeu est décrite dans Game, et dans les classes qui en héritent. c'est dans GameCarcassone et GameTrax que seront écrits les parcours de graphe par exemple, tout comme les fonctions qui calculent le score des joueurs.

Tile définit la logique des tuiles, et contient leurs données. C'est dans Tile que sera définit la fonction qui permet de savoir si deux tuiles peuvent être placées l'une à côté de l'autre par exemple.

Le modèle évolue à chaque fois qu'un joueur appelle une action de l'instance de la classe héritée de Game, comme placeTile() ou discardTile().

Nous avons aussi créé une classe RelativeVector, qui permet de placer des objets à un indice négatif dans un vecteur. Ça nous a été utile car on ne sait pas d'avance dans une partie quelle sera la tuile à la position en haut à gauche, ainsi on définit la première tuile placée comme étant à la position (0,0), et les positions des autres tuiles sont relatives à la première.

Nos fonctions les plus importantes se trouvent dans la classe GameCarcassone, où nous avons dû effectuer des parcours de graphe pour calculer les scores et vérifier si une partie du terrain (ville, route) est complète ou non.

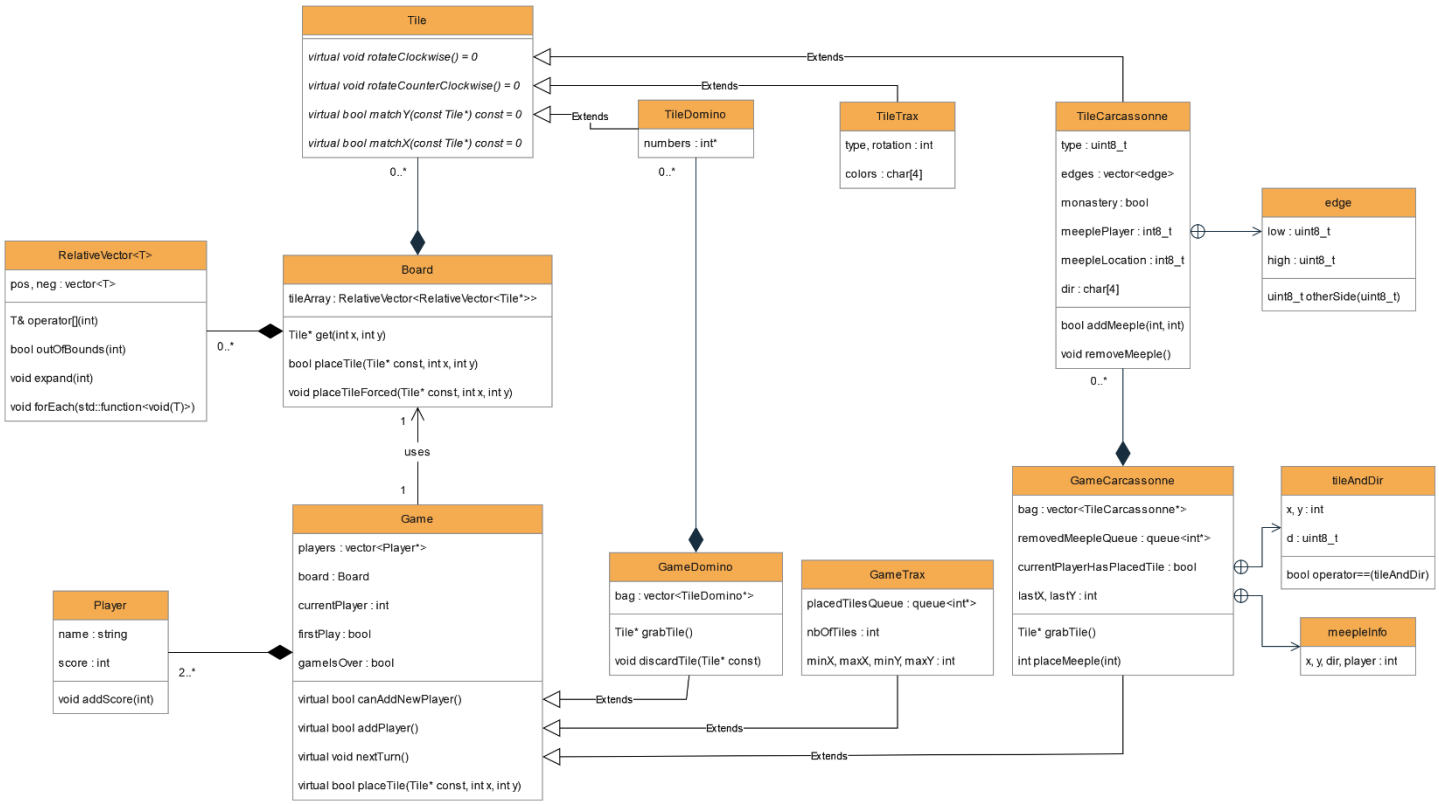

diagramme UML du modèle

Problèmes rencontrés et extensions non implémentées

Aucun problème ou bug n'a été rencontré lors des nombreux tests de la version finale de notre projet.

Nous n’avons pas implémenté d’extension aux jeux, cependant tout le projet a été codé de façon à ce que des extensions puissent être ajoutées aisément. Par exemple, notre façon d’implémenter les tuiles du jeu Carcassonne nous permettrait d’ajouter les extensions « paysages complémentaires » et « rivières » avec très peu de changements à notre code, grâce à notre système d’arêtes. De même, ajouter un quatrième jeu ne poserait aucun souci vis-à-vis des autres jeux.