Rapport du projet C++ « Piece Out »

Yuanheng ZHOU, Foucauld DE LARMINAT

Qu’est-ce qu’on a fait ?

La machine des états :

- La `GameStateMachine` utilise le pattern « Singleton », et ça fait le pattern « State » avec tous les `GameState`;
- Le pointeur `GameState*` de l’instance `GameStateMachine::context` pointe vers un **`MainMenuState` au début**, qui contient un `std::vector<Button*>` ;
- Les autres `GameStates` implémentés :
  - les `MainMenuStateButtonHover` et `MainMenuStateButtonPressed` qui héritent de `MainMenuState` ;
  - le `LevelState` qui est virtuel, les `LevelStateIdle` et `LevelStatePieceClicked` qui héritent de `LevelState`, le `LevelStatePieceSliding` qui est de `LevelStatePieceClicked`…
- Les boutons héritent de la classe **`Button`** virtuelle, qui possède les méthodes virtuelles `onMouseEnter()`, `onMouseLeave()`, `onMouseDown()` qui ne renvoient rien, et `activate()` qui renvoie un pointeur vers un `GameState` (oui on pourrait faire mieux, mais dans ce cas on aurait besoin d’un `StateButtonPressed` pour chaque bouton… on veut éviter d’écrire les templates compliqués) ;

Les niveaux :

- **Codage des cases** par des nombres entiers :
  - `0` pour une case vide, `1` pour un mur, et `n+2` pour la pièce d’indice `n` ;
- Les données d’un `Level` sont stockées dans un objet de **classe `LevelData`**, qui contient **deux matrices** (`vector<int> gridFinal` et `vector<int> gridCurrent`), **une liste de `OperatorData`**, et puis **une liste des `PieceColor`**. Comme les positions sont stockées dans deux matrices, une pièce dans sa position initiale peut avoir des cases sur la position finale attendue d’une autre pièce ;
- On **initialise un niveau** de son `LevelData` **lorsqu’il est choisi** (donc si on veut implementer des centaines de niveaux, on n’a pas besoin de calculer les données de toutes les pièces auparavant), et c’est `Level` qui est résponsable de gérer la mémoire de toutes les pièces et tous les opérateurs ;
- On reviens au `MainMenuState` lorsque chaque pièce est à sa positions finales (si cette pièce a une position finale attendue), et le niveau actuel sera détruit.

Les opérateurs :

- `MovementOperator`, `RotationOperator` et `FlipOperator` héritent bien de `Operator` ;
- Comme dans le jeu original, on peut avoir **plusieurs opérateurs de mouvement sur une seule case** d'une pièce, qui fonctionnent différemment des autres opérateurs : une pièce ne peut posséder qu'un seul opérateur de mouvement de chaque sens, et on les active **en maintenant et en déplaçant la souris dans le bon sens** après avoir appuyé sur la pièce. Donc on peut définir une pièce de taille 1x1 que l'on peut déplacer dans toutes les directions ;
- Les opérateurs de rotation et de symétrie fonctionnent plus facilement : ils s'activent lorsque la souris est relâchée sur la bonne case, après que la case occupée par l'opérateur ait été appuyée ;

Le rendu graphique :

- `Button`, `Level`, `Piece` et `Operator` héritent de `DrawableShape`, qui **hérite de `sf::Drawable`**, et la documentation de `sf::Drawable` dit le suivant :

  > Abstract base class for objects that can be drawn to a render target.
  > sf::Drawable is a very simple base class that allows objects of derived classes to be drawn to a sf::RenderTarget.
  > All you have to do in your derived class is to override the draw virtual function.
  > Note that inheriting from sf::Drawable is not mandatory, but it allows this nice syntax "window.draw(object)" rather than "object.draw(window)", which is more consistent with other SFML classes.

- `DrawableShape` contient un `vector<sf::Vertex>` et un `sf::PrimitiveType`. Ici **on veut imiter un `sf::VertexArray`**, mais comme on redéfinit son propre « vertex array », on peut réserver sa mémoire en avance, et on n'a plus besoin de recopier le contenu du `vector<sf::Vertex>` dans `sf::VertexArray` lorsque `push_back()` nécessite d'espace ;
- On n'utilise que les vertices des objets pour les rendre, donc aucune `sf::Texture` utilisée ;
- Mais on a implémenté un **`ResourceManager`** de pattern « Singleton » pour charger la police.

Qu'est-ce qu'il nous manque ?

- **Plus de boutons pour choisir le niveau...**
- **Une manière plus naturelle pour glisser les pièces :**

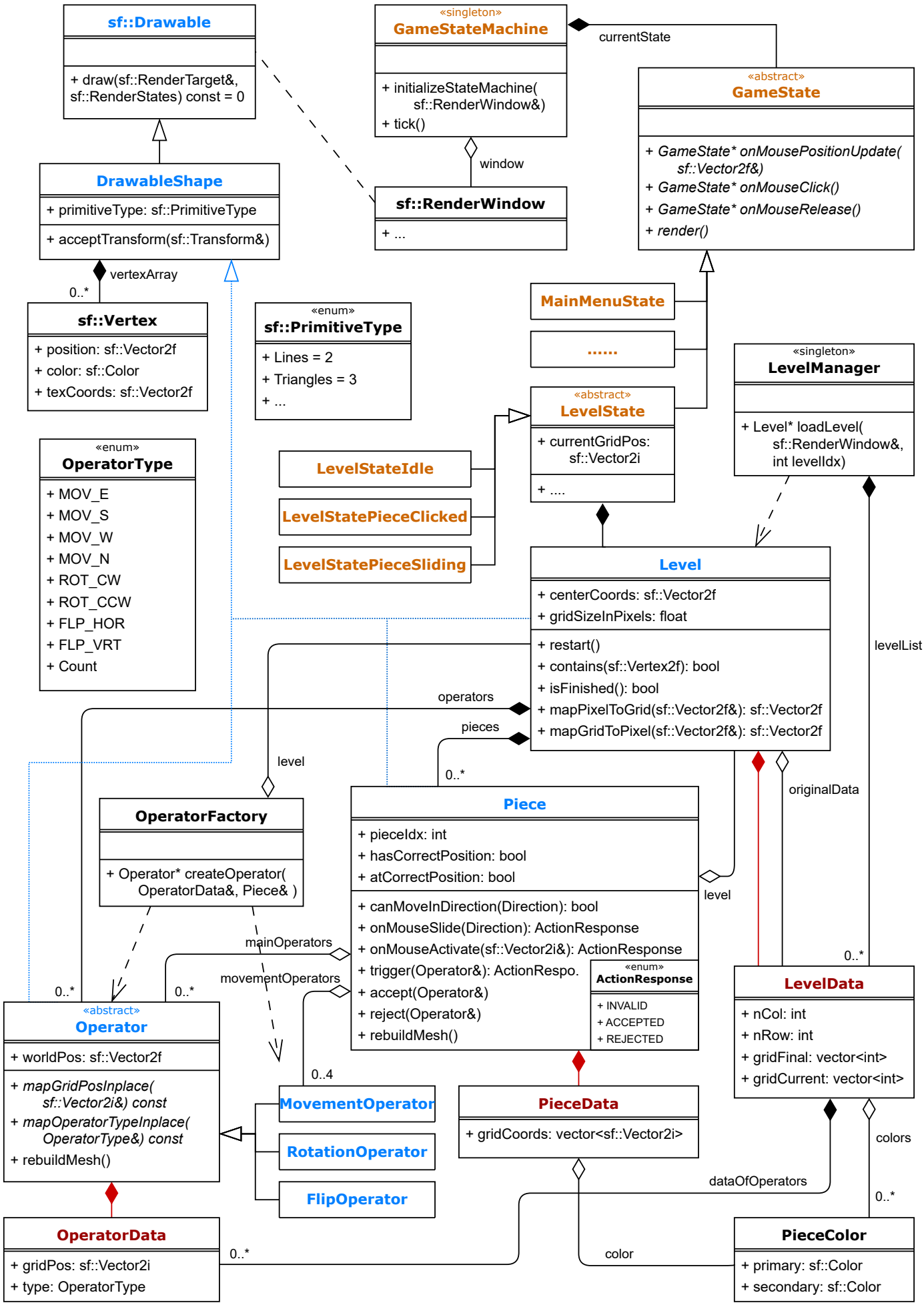
  vous saurez ce que ça signifie si vous essayez de déplacer la pièce rouge de taille 1x1.

- **L'animation :**

  il est déjà possible d'appliquer n'importe quelle transformation affine sur les `DrawableShape`, dont `Level`, `Piece`, `Operator` et `Button` font partie ; mais ça nécessiterait plus d'états pour la `StateMachine`, et on n'a plus de temps pour l'implémenter…

- **Les bouton de redémarrage et d'annulation, et une IA pour résoudre un niveau ;**

  oui, `LevelData` est déjà sous une forme de stockage optimisé, mais il ne reste plus de temps pour les faire…


sf::Drawable
+ draw(sf::RenderTarget&, sf::RenderStates) const = 0
«singleton»
GameStateMachine
+ initializeStateMachine(sf::RenderWindow&)
+ tick()
currentState
«abstract»
GameState
+ GameState* onMousePositionUpdate(sf::Vector2f&)
+ GameState* onMouseClick()
+ GameState* onMouseRelease()
+ render()
window
sf::RenderWindow
+ ...
DrawableShape
+ primitiveType: sf::PrimitiveType
+ acceptTransform(sf::Transform&)
vertexArray
0..*
sf::Vertex
+ position: sf::Vector2f
+ color: sf::Color
+ texCoords: sf::Vector2f
«enum»
sf::PrimitiveType
+ Lines = 2
+ Triangles = 3
+ ...
MainMenuState
.....
«singleton»
LevelManager
+ Level* loadLevel(sf::RenderWindow&, int levelIdx)
«abstract»
LevelState
+ currentGridPos: sf::Vector2i
+ ....
LevelStateIdle
LevelStatePieceClicked
LevelStatePieceSliding
«enum»
OperatorType
+ MOV_E
+ MOV_S
+ MOV_W
+ MOV_N
+ ROT_CW
+ ROT_CCW
+ FLP_HOR
+ FLP_VRT
+ Count
Level
+ centerCoords: sf::Vector2f
+ gridSizeInPixels: float
+ restart()
+ contains(sf::Vertex2f): bool
+ isFinished(): bool
+ mapPixelToGrid(sf::Vector2f&): sf::Vector2f
+ mapGridToPixel(sf::Vector2f&): sf::Vector2f
levelList
operators
pieces
0..*
level
originalData
OperatorFactory
+ Operator* createOperator(OperatorData&, Piece&)
Piece
+ pieceIdx: int
+ hasCorrectPosition: bool
+ atCorrectPosition: bool
+ canMoveInDirection(Direction): bool
+ onMouseSlide(Direction): ActionResponse
+ onMouseActivate(sf::Vector2i&): ActionResponse
+ trigger(Operator&): ActionRespo.
+ accept(Operator&)
+ reject(Operator&)
+ rebuildMesh()
«enum»
ActionResponse
+ INVALID
+ ACCEPTED
+ REJECTED
level
mainOperators
movementOperators
0..*
0..*
0..4
«abstract»
Operator
+ worldPos: sf::Vector2f
+ mapGridPosInplace(sf::Vector2i&) const
+ mapOperatorTypeInplace(OperatorType&) const
+ rebuildMesh()
MovementOperator
RotationOperator
FlipOperator
PieceData
+ gridCoords: vector<sf::Vector2i>
LevelData
+ nCol: int
+ nRow: int
+ gridFinal: vector<int>
+ gridCurrent: vector<int>
colors
0..*
dataOfOperators
OperatorData
+ gridPos: sf::Vector2i
+ type: OperatorType
0..*
color
PieceColor
+ primary: sf::Color
+ secondary: sf::Color