RUZZLE BOBBLE

Ruzzle Bobble ayant pour vocation à être un jeu vidéo, nous avons imaginé et développé tout un moteur 2D ainsi qu'un lecteur de fichiers sonores.

Interface Graphique

Le moteur graphique est articulé autour d'une classe « Engine » servant de coeur au logiciel. Elle permet de faire le lien entre les différentes séquences (écran de titre, séquence de jeu), l'affichage et le fonctionnement du jeu. L'Engine se charge du rafraîchissement de l'écran de jeu (dans un Thread spécifique) et permet d'amorcer les changements de séquence de jeu et de leur envoyer des signaux (pour exemple, l'Engine peut indiquer à la séquence d'écran de titre que les dictionnaires ont été chargés, la séquence d'écran de titre réagit en faisant apparaitre le bouton permettant le lancement du jeu).

Les séquences de jeu sont des classes héritées d'une classe « Stage ». Une instance de Stage est dotée d'une liste de Sprites et d'une musique associée. Lorsqu'une séquence est chargée dans l'Engine, l'écran de jeu passe en revue la liste de Sprite du Stage et les affiche successivement. A l'initialisation d'un stage, un séquenceur MIDI est lancé pour jouer une musique d'ambiance. C'est aussi à la classe Stage de gérer l'animation et les mouvements des Sprites qui lui sont associés dans des Thread fonctionnant en parallèle du fonctionnement du jeu, en modifiant des propriétés des Sprites de sa liste de Sprites.

Un sprite est une instance ou un objet héritant d'une classe Sprite. Un Sprite est une instance contenant de nombreuses propriétés prises en compte lors de leur affichage : une image, des propriétés de transparence ainsi que ses positions à l'écran. Un sprite possède en outre une liste d'images qui lui sont associées. Lorsqu'elle veut l'animer, une séquence a juste à modifier son image courante en choisissant parmi sa liste de Sprite (ou à appeler une séquence d'animation du sprite ayant été redéfinie). Le positionnement dans l'espace du Sprite est géré par un rectangle, permettant de connaitre sa position dans l'axe X et Y mais aussi de prendre en compte sa largeur et sa hauteur, ce qui est utile lors qu'on veut savoir si le sprite est en collision avec un autre sprite ou un événement lié à l'utilisation de la souris.

L'affichage du jeu est effectué par l'instance d'une classe GameScreen (héritant de la classe JPanel de java). Comme dit précédemment, le Gamescreen affiche la liste de sprite de la séquence de jeu qu'on lui associe en prenant en compte les propriétés desdits Sprite, ceci en une fréquence définie par l'Engine. C'est aussi le Gamescreen qui gère les interactions entre le joueur et la séquence de jeu. Le Gamescreen est ainsi chargé de capter les mouvements et les interactions de la souris avec lui même.

Par exemple, lorsque le joueur clique sur le Gamescreen, il met en revue les sprites associés à la séquence de jeu et vérifie si leur hitbox (le rectangle qui leur est associé) est en collision avec les coordonnées de la souris et, le cas échéant, appelle la méthode associé à l'interaction souris utilisée (le clic souris et le maintien du clic étant différencié). Par exemple, le bouton « Jouer » de l'écran de titre est programmé pour demander à l'Engine de charger une séquence de jeu lorsque l'on clique dessus tandis qu'un Sprite de lettre en partie est programmée pour ajouter sa lettre au mots courant lorsque la souris passe dessus avec un clic maintenu, ce qui permet une saisie plus rapide et intuitive des mots sur la grille de jeu.

Enfin, une classe MusicPlayer est chargée de la gestion des musiques MIDI (via le séquenceur de Java) ainsi que d'autres formats sonores comme les .WAV (via la classe Clip de Java) afin de créer une ambiance et donner vie aux mascottes du jeu : Bub et Bob.

La conception du jeu

RuzzleDictionary :
C'est le dictionnaire sur lequel tout le jeu se base.

Il contient tout les mots possibles de la langue donné par l'utilisateur, et permet donc des vérifications soit pour le jeu en lui même, soit pour les algorithmes utilisé.

Pour garder en mémoire chacun des mots, il utilise un arbre lexicographique, où chaque nœud possède une valeur correspondant à une lettre, et chaque fils correspondant à une suite existante dans la langue.

Chacun des nœuds peut être dans un état terminal (c'est à dire que le chemin parcouru dans l'arbre correspond à un mot) ou dans un état non terminal (donc à l'inverse, le chemin parcouru ne correspond pas à un mot existant dans le dictionnaire).

Lettre :
Cet objet contient toutes les données liées à la lettre, c'est à dire le score (qui change selon la langue du jeu), le caractère correspondant à la lettre, et enfin un tableau contenant la fréquence d'utilisation de la lettre selon la langue du jeu.

Marble :
Cet objet correspond aux cases du jeu, et permet de stocker des informations nécessaires aux algorithmes. Il contient la Lettre associée, les bonus possibles, ainsi que tous les voisins de cette même cases.

Les bonus possibles

- *mot compte double*
- *mot compte triple*
- *lettre compte double*
- *lettre compte triple*

Board :
Il correspond au plateau du jeu, et contient toutes les données nécessaires pour son bon fonctionnement. Ainsi, il représente le plateau à l'aide d'un tableau à deux dimensions de Marble, où chaque coordonnées précise correspondra à une case du plateau.
Le dictionnaire y est aussi présent.

La conception des algorithmes

En préambule, les algorithmes de parcours de dictionnaire génère avant tout une HashMap, qui contient, pour chaque lettre présente sur le plateau, une liste des positions.

Ainsi, si une lettre n'a aucun position sur le plateau, elle n'existera pas dans la HashMap, sauf s'il y a présence de joker. On peut donc dire que s'il y a au moins un joker présent sur le plateau, alors chacune des lettres de l'alphabet existera.

AlgorithmsData :
Cet objet contient toute les informations nécessaires au bon déroulement des algorithmes.

Il y stocke la position courante dans l'arbre, la position courante dans le plateau, le mot en cours de construction, et enfin le chemin parcouru dans l'arbre.

Il était avantageux de passer par cet objet, notamment pour la lisibilité du code lors de l'utilisation de ces algorithmes.

On garde la position courante dans l'arbre pour éviter de devoir le parcourir à chaque fois pour déterminer si le mot en construction en est un. Ainsi, on aura juste à déterminer si le nœud courant est dans un état terminal pour savoir si le mot existe dans le dictionnaire. De plus, il permet de simplifier les cas lors de jokers : on se réduit à tester les fils de la position courante, plutôt que de tester toutes les lettres possibles.

La position courante dans le plateau nous permet de savoir quelle est la dernière case qui a été parcouru, et nous permet donc de retrouver très simplement les voisins.

Le chemin est utile pour nous indiquer si la case que l'on souhaite tester à déjà été visitée. Si elle a déjà été visitée, alors on ne fera rien avec cette case là.

Quatre algorithmes de résolution différents ont été écrit.

1. DFS Grille

(SolveByMarbleGrid ; fonctions : initDfs() et dfs(AlgorithmsData))

La première étape de l'algorithme (qui se situe dans initDfs()), est une simple boucle permettant de cibler chacune des cases, pour pouvoir lancer le dfs avec ce point de départ.

Si la case ciblée est un joker, alors on lance un dfs pour tout l'alphabet, sinon on lance dfs pour la seule lettre de la case.

La seconde étape est la fonction dfs(AlgorithmsData), qui est récursive.

Elle fonctionne en deux points principaux :

- En premier lieu, on teste si l'état du nœud de l'arbre courant est terminal, et si oui on ajoute le mot courant dans la liste.
- Ensuite on teste chacun des voisins de la case courante :
  - Si le voisin se trouve déjà dans le parcours, alors on ne fait rien.
  - Si le voisin est un joker, alors on réenvoie un dfs pour chacun des fils de la position courante dans l'arbre.
  - Sinon, on regarde s'il existe un fils possédant la même lettre que le voisin ; si oui alors on envoie un dfs, sinon on ne fait rien.

2. BFS Grille

(SolveByMarbleGrid ; fonctions : initQueue(integer[]), bfs())
Cet algorithme fonctionne à l'aide d'une file.

La fonction initQueue sert à initialiser cette file.

Ainsi, pour chaque case du plateau, si on tombe sur un joker, alors on ajoute toute les lettres de l'alphabet, sinon on ajoute la lettre tout simplement. Puis on retourne la file.

L'algorithme se résout ensuite, tant que la file n'est pas vide, en 3 étapes :

- On retire la première donnée.
- Si la position courante de l'arbre dans ces données est terminale, alors on ajoute le mot.
- Ensuite pour chaque voisin de la position courante dans le plateau :
  - Si le voisin existe déjà dans le parcours, alors on ne fait rien.
  - Si la lettre du voisin correspond à un joker, alors pour chaque fils de la position courante dans l'arbre, on enfile un objet de donnée mis à jour.
  - Sinon on enfile un objet de donnée avec la nouvelle lettre ajoutée.

3. DFS Dictionnaire

(SolveByDictionary ; initDfs(Tree), dfs(AlgorithmDatas))

En premier lieu, la fonction iniDfs va se charger de faire les appels nécessaires à dfs.

On utilise donc la HashMap qui a été créée pour l'occasion. Ainsi, pour chaque fils de la position de départ de l'arbre, on va tester si des positions existent dans la HashMap.

Si elles existent, alors pour chacune de ces positions, on lance un dfs.
Sinon on ne fait rien.

La suite se situe dans le dfs, qui se résout récursivement, en 2 étapes :

- Si la position courante dans l'arbre est terminale, alors on ajoute le mot dans la liste.
- Pour chaque fils la position courante dans l'arbre :
    - Pour chaque voisins de la case courante dans le plateau :
        - Si la lettre correspond à un joker, ou si la lettre correspond à la position dans l'arbre, alors on relance un dfs en ayant mis à jours les données.

4. BFS Dictionnaire

(SolveByDictionary ; fonctions : initQueue(Tree), bfs())
La fonction initQueue initialise une file à l'aide de la HashMap.

Ainsi, pour chacun des fils de la position initiale de l'arbre, s'il existe des positions dans le plateau, alors pour chacune de ces positions on ajoute une donnée dans la file.

L'algorithme se résout ensuite, tant que la file n'est pas vide, en 3 étapes :

- On retire la première donnée.
- Si la position courante de l'arbre dans ces données est terminale, alors on ajoute le mot.
- Ensuite pour chaque fils de la position courante dans l'arbre :
    - Pour chaque voisin de la case courante dans le plateau :
        - Si le voisin existe déjà dans le parcours, alors on ne fait rien.
        - Si la lettre du voisin correspond à un joker, alors pour chaque fils de la position courante dans l'arbre, on enfile un objet de donnée mis à jour.
        - Sinon on enfile un objet de donnée avec la nouvelle lettre ajoutée.