Sevi DERVISHI

Adrian HEOUAIRI

Interfaces graphiques : projet Energy

Partie 1 : modèle

Level

Un niveau qu'on joue ou qu'on édite est représenté par un objet de la classe Level. Un level possède un numéro qui correspond à son numéro de fichier.

Board

Un level contient un objet de la classe Board, qui représente le plateau de jeu. Une board possède une géométrie, de l'enum Geometry = { SQUARE, HEXAGON }. La géométrie de la board détermine la géométrie des tuiles qu'elle contient. Une board stocke ses tuiles (ses cases) dans une List<List<Tile>>. Le fait d'utiliser des List au lieu d'un tableau 2D permet de faire varier la taille de la board (nous permettons cela dans l'éditeur de niveaux). Nous passerons en revue les méthodes de Board après avoir présenté les autres classes.

Tile

Un objet de la classe Tile représente une tuile du plateau de jeu. Une tile possède une géométrie de l'enum Geometry. Elle possède une List<Connector> qui est toujours de taille 4 ou 6 selon la géométrie. Cette liste représente les bords connectés de la tuile. Le composant au centre d'une tuile est déterminé par un attribut de type enum Component = { SOURCE, LAMP, WIFI, EMPTY }. Enfin, soit une tuile est alimentée, soit elle ne l'est pas. Cet état peut être obtenu par la méthode isPowered() de Tile.

Connector

Un Connector représente un bord d'une tuile. Ce bord peut soit avoir un fil, soit ne pas en avoir. Cela est représenté par un booléen exists dans Connector. Un connector possède une référence vers sa tuile parente, cela est utile pour l'algorithme de propagation d'énergie. Un connecteur possède une référence vers un autre connecteur, appelé neighbor. Neighbor est le connecteur qui se situe en face d'un connecteur. Par exemple, le neighbor du connecteur Est d'un carré est le connecteur ouest du carré à la droite du premier carré. Connaitre son neighbor est très utile dans l'algorithme de propagation. Enfin, un connecteur possède une direction de l'interface Direction décrite ci-dessous.

Direction

Direction est une interface qui nous permet de faire une abstraction entre les 4 directions (nord, est, sud, ouest) pour les côtés d'un carré et les 6 directions pour un hexagone. On peut demander à une direction la liste des directions, qui renvoie soit 4 soit 6 éléments, on peut demander son ordinal (valeur entre 0 et 5 pour un hexagone par exemple). Une direction a la méthode getHeightOffset(int

columnIndex). Cela nous donne de combien il faut se déplacer verticalement dans List<List<Tile>> de Board pour atteindre la case pointée par une direction : par exemple, pour nord, cela retourne -1. Cela est utile par exemple pour nord ouest avec un hexagone : selon la parité de la colonne, il faut se déplacer de 0 ou -1. On a également getWidthOffset() qui est analogue. Ces deux méthodes nous permettent donc d'obtenir les coordonnées de la case au nord-ouest d'une tuile sur une board hexagonale par exemple. Cela est utile pour déterminer le neighbor de chaque connector. On a enfin la méthode getOppositeDirection() qui retourne la direction opposée.

L'interface Direction est implémentée par les enum Direction4 et Direction6, pour représenter les côtés d'un carré ou d'un hexagone respectivement.

Retour sur Tile

Une tile possède les méthodes rotateClockwise() et rotateCounterClockwise(), qui fait tourner la tuile d'un quart ou d'un sixième de tour selon sa géométrie. rotateClockwise() ne fait que déplacer le booléen exists sur le connecteur suivant pour chaque connecteur : par exemple pour un carré dont seul le connecteur nord existe, un appel à rotateClockwise() met exists à false au nord et exists à true à l'est. Ainsi, chaque connector peut toujours garder le même neighbor. rotateCounterClockwise() est simplement implémenté comme 3 ou 5 appels à rotateClockwise(), selon la géométrie.

Une tile possède une méthode cycleComponent() qui change le composant au milieu d'une tuile dans un ordre cyclique prédéfini.

Retour sur Board

Une board possède 8 méthodes qui permettent d'ajouter ou enlever une ligne ou une colonne de tuiles en haut, en bas, à gauche ou à droite. Ceci est utilisé dans l'éditeur de niveaux. Une board possède une méthode getTilesWithComponent(Component) qui retourne toutes les tuiles avec un composant spécifique. Une board possède une méthode isSolved() qui retourne true si toutes les lampes sont allumées, même si la board a 0 lampes. Une board a une méthode shuffle() qui appelle rotateClockwise() un nombre aléatoire de fois sur chaque tuile.

Algorithme de propagation d'énergie

À chaque fois qu'un changement se produit sur la board, on éteint toutes les tuiles (sauf les sources) et on lance l'algorithme de propagation d'énergie. Il s'agit d'un algorithme simple :

```
pour chaque tuile t contenant une source : t.propagateEnergyToNeighbors()
oneWifiIsPowered = false
pour chaque tuile t contenant un wifi :
       si t est allumée alors oneWifiIsPowered = true
si oneWifiIsPowered :
       pour chaque tuile t contenant un wifi : allumer t
       pour chaque tuile t contenant un wifi : t.propagateEnergyToNeighbors()
```

Et le code de propagateEnergyToNeighbors() dans Tile :

```
List<Tile> connectedNeighborsNotPowered = new ArrayList<>();
pour chaque connecteur c de la tuile :
       si c et c.neighbor existent et la tuile de c.neighbor est éteinte :
              ajouter la tuile de c.neighbor à connectedNeighborsNotPowered
pour chaque tuile t dans connectedNeighborsNotPowered : allumer t
pour chaque tuile t dans connectedNeighborsNotPowered : t.propagateEnergyToNeighbors()
```

Partie 2 : utils

Pour écrire un Level dans un fichier ou obtenir un Level à partir d'un fichier, nous avons la classe LevelConverter. Lors du parsing d'un fichier de niveau, nous validons les lignes par expression régulière. Lorsqu'on démarre le programme pour la première fois, les niveaux fournis dans la banque 1 sont copiés dans le répertoire ~/.energy, qui constitue la banque 2.

Partie 3 : vue (interface graphique)

Nous avons MainMenuView et BankSelectionView qui sont codés simplement avec un BoxLayout vertical. Pour représenter les boutons, nous utilisons des JLabel au lieu de JButton car ceux-ci sont plus faciles à styliser. Nous avons BankView (qui prend en paramètre un numéro de banque) et affiche la liste des niveaux dans un GridLayout ainsi que des boutons pour jouer, éditer si banque 2, etc. Nous avons enfin PlayingLevelView et EditingLevelView, qu'on affiche lorsqu'on est en train de jouer ou d'éditer un niveau. Ces deux vues contiennent une BoardView et des boutons pour effectuer certaines actions comme retour. Une BoardView est un JPanel dont la méthode paintComponent() est redéfinie pour afficher le plateau de jeu. Nous utilisons un algorithme complexe qui permet de centrer horizontalement et verticalement le plateau au milieu de la BoardView, de préserver les proportions d'une tuile (les carrés ne sont jamais déformés en rectangle, pareil pour les hexagones), et la taille d'une tuile s'adapte à la taille de la BoardView (la fenêtre peut donc être redimensionnée). Nous obtenons les fils électriques, composants et contours à dessiner grâce à une classe SpriteBank qui les extrait de l'image png fournie avec getSubimage().

Partie 4 : respect du pattern MVC

Board et Tile implémentent respectivement les interfaces ReadOnlyBoard et ReadOnlyTile. Cela permet de donner à la vue une vision en lecture seule de ce qu'elle affiche. De plus, Board implémente l'interface BoardObservable, qui lui permet de maintenir une liste d'observateurs et de les notifier avec notifyObservers(). Les observateurs implémentent l'interface BoardObserver qui possède la méthode « void update(ReadOnlyBoard) ». BoardView et PlayingLevelView implémentent BoardObserver. Cela permet à BoardView de redessiner le plateau à chaque appel de notifyObservers(), et à PlayingLevelView de vérifier si le joueur vient de gagner la partie pour passer au niveau suivant.

Partie 5 : contrôleurs

À chaque fois que BoardView dessine le plateau, nous stockons dans une map en attribut, pour chaque tuile t, l'entrée suivante : clé = coordonnées en pixels dans la BoardView du centre de t,

valeur = ligne et colonne de la tuile t dans la Board. Pour identifier la tuile dans laquelle l'utilisateur a cliqué nous cherchons simplement le centre de tuile le plus proche du clic en itérant sur la map. On obtient donc la ligne et la colonne de la tuile sur laquelle il faut agir. En mode édition, l'action est différente selon la zone dans laquelle on clique à l'intérieur de la tuile. Après avoir déterminé la tuile dans laquelle l'utilisateur a cliqué, nous déterminons la zone : si l'utilisateur a cliqué dans le cercle de rayon 20 % de de la longueur de la tuile et positionné au centre de la tuile, nous considérons qu'il a cliqué au milieu de la tuile et nous changeons le composant de la tuile avec cycleComponent(). S'il a cliqué en dehors de ce cercle, on utilise la fonction atan2 pour déterminer l'angle formé entre l'horizontale et la droite qui passe par le clic et le centre du polygone. Selon cet angle et selon la géométrie de la tuile, on toggle l'existence du connecteur qui se situe dans cette zone. Par exemple, s'il clique sur la région nord de la tuile, on toggle le connecteur nord.