

Fourmis

Concepts abordés

- Développer dans le cadre d'un *framework* qui impose une structure et un comportement particuliers.
- Implémenter une *interface*
- Swing : Création de composants, capture d'événements, dessin et animation
- Structures et algorithmes orientés-objet complexe
- `java.io.StreamTokenizer`
- Créer un JAR

Consignes générales de travail

- Ce projet est à préparer et à rendre en binôme (voir sur Campus EFREI les conditions de remise du rapport de TP).
- En plus des consignes habituelles, il faut tout de suite comprendre que ce projet est plus complexe que les TPs précédents.
- Ce projet impose une *framework* que vous devez respecter. Prenez le temps de déchiffrer sa structure et son fonctionnement avant d'y intégrer votre code.
- Prenez également le temps de réfléchir avant de construire: Concevez votre structure de classes, imaginez vos objets en mémoire, testez vos algorithmes ... et *ensuite* codez-les. Profitez de l'UML pour exprimer et valider vos idées avec vos professeurs.

Présentation

Objectifs

Le but de ce projet est simuler la quête de nourriture d'une colonie de fourmis sur un terrain accidenté. Le terrain est un espace rectangulaire composé de cases qui peuvent être des obstacles, la fourmilière, ou libres avec ou sans nourriture. Le comportement de chaque fourmi et de partir de la fourmilière à la recherche de nourriture et de la ramener quand il la trouve. Dans sa forme plus perfectionnée (facultatif pour cet exercice), les fourmis peuvent également déposer des phéromones sur les cases pour signaler la présence de nourriture aux autres fourmis. L'intérêt "scientifique" de ce genre de simulation est de mieux comprendre le fonctionnement de la nature. Cette simulation en particulier permet de comprendre comment une colonie de fourmis arrive à ramasser avec une efficacité étonnante la nourriture essentielle à sa survie.

Framework imposé

Ce projet impose un *framework* que vous devez compléter, sans pour autant le modifier. Il est défini dans le paquetage `simengine` et ses sources sont distribuées dans le JAR `fourmiz.jar` à télécharger sur Campus EFREI.

`simengine` fournit:

- `class JeuPanel`, qui implémente le moteur de simulation et des méthodes (`start`, `stop`) pour le contrôler, et sert également comme un composant Swing capable d'afficher un terrain à simuler;
- `interface TerrainDeJeu`, qui définit le comportement qu'un terrain doit implémenter pour être prise en charge par le moteur de `JeuPanel`;
- `class Utilitaire`, qui fournit quelques méthodes qui vous seront utiles.

Notez que `simengine` n'implémente pas:

- le dessin des fourmis ou des cases
- les algorithmes de comportement des fourmis.
- l'interface graphique de contrôle du simulateur par l'utilisateur.

Votre travail sera, dans un premier temps, d'implémenter l'interface `simengine.TerrainDeJeu` pour y définir le stockage et dessin du terrain, et les actions et l'affichage des fourmis; dans un deuxième temps, de créer une fenêtre graphique Swing contenant un `JeuPanel` et les contrôles nécessaires pour permettre à l'utilisateur de le contrôler.

Méthode de travail et de rendu

Vous devez créer toutes vos classes dans un paquetage nommé `fourmiz`. Vous ne devez pas:

- créer d'autres paquetages,
- placer des fichiers dans le paquetage par défaut,
- modifiez le paquetage `simengine`.
-

Votre paquetage `fourmiz` doit être rendu sous forme d'un fichier JAR portant les noms des membres du binôme, et doit impérativement contenir les sources de vos classes (merci de vérifier - rendre uniquement des fichiers Java compilés donnera lieu à une note de 0).

Ne cédez pas à la tentation d'"améliorer" `simengine`. Votre paquetage `fourmiz` sera testé avec le `simengine` d'origine, donc toutes nouvelles dépendances échoueront !

Fichiers fournis fournis

Le fichier JAR `fourmiz.jar` contient :

- les sources du paquetage `simengine`,
- une application Swing d'exemple (double-cliquer dessus pour lancer l'application).
- trois fichiers de données de terrain (`terrain.dat`, `terrain2.dat`, `terrain3.dat`) qui définissent chacun un terrain différent, et que votre programme doit pouvoir charger, afficher et simuler.
-

Exercices

Exercice 1: Charger et afficher le terrain

A la fin de cette question, votre programme devrait être capable de charger un terrain depuis un fichier et l'afficher. Les fourmis sont également affichées dans la case fourmilière.

- Importez et examinez les fichiers fournis dans l'archive `fourmiz.jar`
- Assurez-vous de comprendre la structure et le fonctionnement du framework défini dans `simengine`. Prêtez surtout attention à l'interface `TerrainDeJeu` et ses méthodes qu'il va falloir implémenter.
- Concevez un modèle de données orienté-objet pour un terrain de jeu. Cela exigera notamment la création d'une classe pour chaque entité du domaine, par exemple `Fourmi`, `Case`, `Fourmilière`, ... Veillez à ce que votre modèle soit assez riche en associations pour réaliser toutes les méthodes de l'interface `simengine.TerrainDeJeu`.
- Codez vos classes. Concentrez-vous sur les éléments nécessaires à l'initialisation et l'affichage de vos objets. Le comportement des fourmis est à laisser aux exercices 2 et 3.
- Créez une classe `MonTerrain`. Cette classe devrait :
 - implémenter l'interface `simengine.TerrainDeJeu` .
 - fournir une méthode de chargement d'un terrain depuis un fichier nommé qui initialise votre structure de données définie ci-dessus. Votre méthode doit lancer une exception si le fichier terrain est invalide (manque de cases, plusieurs exactement une fourmilière, ...) ou si une erreur de lecture survient. Un exemple

- de cette méthode (sans détection d'erreurs) est donné dans la classe `simengine.Utilitaire`. Copiez-la et adaptez-la à vos besoins.
- déléguer un maximum de son travail à ses objets composants.
 - Une fois l'exercice terminé, vérifiez que votre implémentation est bien orienté-objet: Vos données sont-elles bien encapsulées ? Les méthodes sont-elles placées dans les bonnes classes ? ... Cela sera pris en compte dans la note.

Exercice 2: La recherche de nourriture

A la fin de cette question, votre programme devrait afficher le mouvement des fourmis dans leur recherche de nourriture. Les fourmis sont tous placés dans la fourmilière au début de la simulation. A chaque pas (« step ») de la simulation, chaque fourmi choisit aléatoirement l'une des directions définies dans la Figure 1. La direction choisie modifie la position (X,Y) d'une fourmi par les valeurs dX et dY définies dans les Figures 2 et 3.

```
0 1 2
7 X 3
6 5 4
```

Figure 1: directions

```
-1 0 +1
-1 X +1
-1 0 +1
```

Figure 2: modification de X (dX)

```
-1 -1 -1
0 X 0
+1 +1 +1
```

Figure 3: modification de Y (dY)

Afin d'éviter un mouvement trop erratique, la nouvelle direction sera choisie en fonction de l'ancienne, avec une pondération qui favorise un mouvement dans une ligne droite. Pour cela chaque direction possible se voit attribuer une pondération en fonction de sa rotation par rapport à l'ancienne direction. On utilise une rotation 0 pour garder la même direction, ± 4 pour faire marche arrière, ± 1 pour aller en diagonale en avant, etc. La Figure 4 définit les pondérations pour chacune des 8 rotations possibles (les rotations ± 4 sont considérées comme une seule et même rotation).

Rotation	0	± 1	± 2	± 3	± 4	
Pondération		12	2	1	1	0

Figure 4: pondérations de chaque rotation par rapport à l'ancienne direction pendant la recherche de nourriture

Par exemple, si l'ancienne direction d'un fourmi est 1, il y aura 12 chances sur 20 ($12 \times 1 + 2 \times 2 + 2 \times 1 + 2 \times 1 + 1 \times 0$) que la nouvelle direction soit également 1 (rotation 0), et 2 chances sur 20 que sa nouvelle direction soit 2 (rotation +1), etc. Il n'y a aucune chance que la fourmi fait marche arrière (rotation ± 4). La classe **simengine.Utilitaire** fournit une méthode **randomPondere** qui génère une valeur aléatoire dans l'intervalle 0..N-1, d'après un tableau de pondérations de taille N. Si une direction n'est pas possible (obstacle ou sortie du monde), sa pondération correspondante est annulée.

Ci-dessous est donné l'algorithme complète pour la génération de la nouvelle position d'une fourmi en fonction de son ancienne position et direction. A noter : quand une fourmi sort de la fourmilière elle doit choisir une direction totalement aléatoire.

- Implémentez les méthodes nécessaires pour réaliser le comportement d'une fourmi. Réfléchissez bien comment placer vos méthodes dans les classes appropriées.
- Si ce n'est pas fait, implémentez la méthode `step` de votre classe `MonTerrain`.

Objective : calculer la nouvelle position (newX,newY) d'un fourmi

Pre-requisites :

```
prev_dir : 0..7 // ancienne direction du fourmi
currentX : int // position X actuelle du fourmi
currentY : int // position Y actuelle du fourmi
dX[8] = { -1, 0, 1, 1, 1, 0, -1, -1 } // dX[i] = modification de X pour direction i
dY[8] = { -1, -1, -1, 0, 1, 1, 1, 0 } // dY[i] = modification de Y pour direction i
rot[8] = { 12, 2, 1, 1, 0, 1, 1, 2 } // rot[i] = pondération pour direction i quand prev_dir=0
```

Return :

```
int newX, int newY; // la nouvelle position du fourmi
```

Variables :

```
weights[8] : int // weights[i] = la pondération de direction i
new_dir : 0..7 // la nouvelle direction
```

Algorithm :

```
pour i de 0 to 7 faire
    weights[i] = rot[(8+i-prev_dir)%8]
    si blockAt(currentX+dX[i], currentY+dY[i]) est infranchissable alors
        weights[i] = 0
fini
finpour
new_dir = Utilitaire.randomPondere(weights)
newX = currentX + dX[new_dir]
newY = currentY + dY[new_dir]
```

Exercice 3: Retour à la fourmilière

A la fin de cette question vos fourmis devraient pouvoir récupérer de la nourriture et la ramener à la fourmilière par le chemin le plus court. Nous allons définir deux états pour une fourmi, qui détermineront comment elle se déplace :

- A la recherche de nourriture
- Retour à la fourmilière

Dans l'état « à la recherche de nourriture », la fourmi se déplace comme décrit dans la Question 2. En plus, quand elle détecte de la nourriture sur son bloc actuel, elle en récupère une unité, se retourne, et passe dans l'état « retour à la fourmilière ». Dans l'état « retour à la fourmilière », la fourmi se déplace vers la fourmilière par le chemin le plus court. Arrivée à la fourmilière, elle dépose son unité de nourriture, fait demi-tour, et repasse dans l'état « à la recherche de nourriture ». La Figure 6 décrit un algorithme pour déterminer la direction qui mène vers la fourmilière par le chemin le plus court. Une fois cette direction connue, la fourmi choisit une direction aléatoire qui favorise cette direction, utilisant les pondérations de rotation données par le Figure 7. A noter: Cet algorithme exige qu'une fourmi connaisse à tout moment la position de sa fourmilière dans le terrain.

1. Écrivez une méthode qui implémente l'algorithme donné ci-dessous

Objective: Calculer la direction (0..7) qui mène à la fourmilière

Principal: Calculer le vecteur direction normalisé et déterminer à quelle direction il corresponde

Pre-requisites:

```
aX,aY : int // position actuelle du fourmi
fX,fY : int // position de la fourmilière
dX[8] = { -1, 0, 1, 1, 1, 0, -1, -1 }
dY[8] = { -1, -1, -1, 0, 1, 1, 1, 0 }
```

Return:

```
direction : 0..7 // la direction qui mène à la fourmilière
```

Algorithm:

```
int tempX = fX - aX;
int tempY = fY - dY;
float distance = sqrt(tempX*tempX + tempY*tempY);
tempX = (int)round(tempX/distance);
tempY = (int)round(tempY/distance);
for (int i=0; i<8; i++) {
    // déterminer la direction correspondante
    if (tempX==dX[i] && tempY==dY[i]) direction = i;
}
return direction;
```

Rotation	0	± 1	± 2	± 3	± 4
Probability	200	32	8	2	1

Figure 7: pondération de chaque rotation par rapport à la direction menant à la fourmilière pendant le retour à la fourmilière

Exercice 4: L'interface utilisateur Swing

La classe `simengine.JeuPanel` fournit plusieurs méthodes qui permettent de contrôler le simulateur (`init()`, `start()`, `stop()`, etc). Dans la Question 1 vous avez également implémenté dans `MonTerrain` une méthode permettant de charger un terrain depuis un fichier. Dans cette question vous devez compléter votre programme de test avec une interface plus élaborée, comprenant les contrôles suivants :

1. Des `JButton` pour démarrer, arrêter ou faire avancer d'un pas la simulation.
2. Un `JSlider` pour contrôler la vitesse des pas de la simulation.
3. Un `JCheckBox` pour afficher/cacher la grille.
4. Un `JButton` pour charger un monde depuis un fichier, associé à un `JFileChooser` pour laisser à l'utilisateur le choix du fichier à charger.

Vous devez également contraindre les actions de l'utilisateur. Par exemple, empêcher le démarrage d'une simulation avant qu'un terrain ne soit chargé. Cela peut se faire facilement par moyen la méthode `setEnabled(boolean):void`, implémentée par la plupart des composants Swing. La Figure 8 définit les contraintes à imposer sur l'utilisateur, par moyen d'un *statechart*.

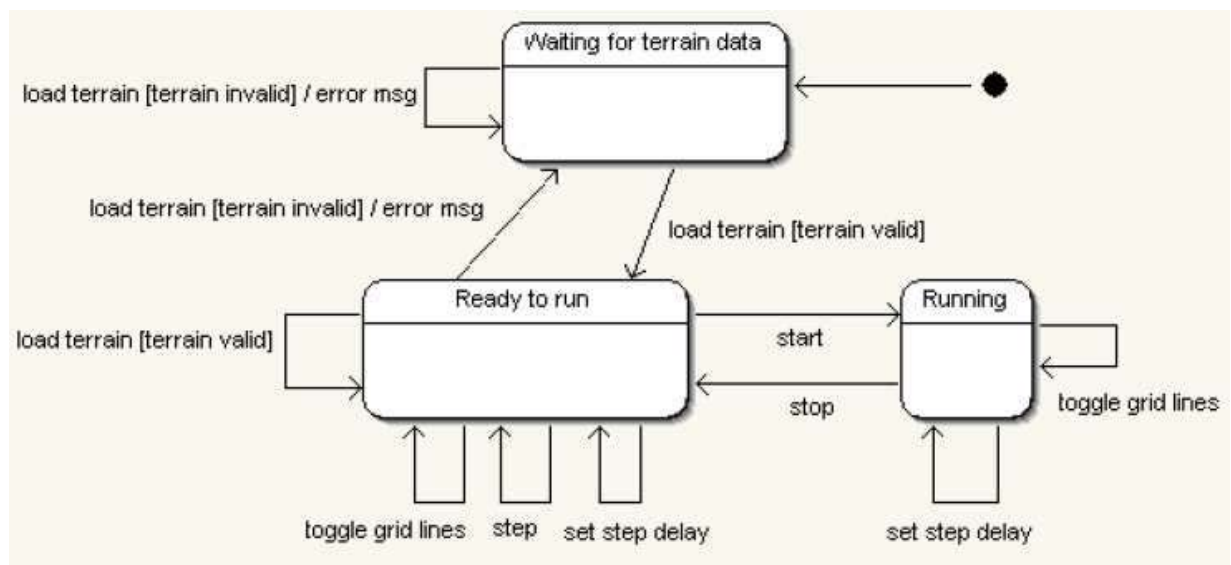


Figure 8: *state chart* définissant le comportement voulu du programme

Exercice 5: Les phéromones (facultatif, pour des points bonus)

Les fourmis (les vrais) utilisent des phéromones pour signaler un chemin qui mène vers de la nourriture, afin de permettre aux autres fourmis de la retrouver plus rapidement. Dans cette question nous implémentons cette idée dans notre simulation. Quand une fourmi retourne vers la fourmilière chargée de nourriture, elle dépose 10 unités de phéromones sur chaque bloc traversé et 5 unités sur chaque bloc environnant. Un bloc peut stocker jusqu'à 100 unités de phéromones, son niveau de saturation. Les phéromones stockées s'évaporent dans le temps, à une vitesse de 1 unité par 20 pas de simulation. Quand une fourmi est à la recherche de nourriture, elle suit les phéromones stockées dans les blocs. Le problème est rendu plus difficile par le fait qu'elle doit suivre les phéromones dans la direction opposée de la fourmilière. Afin de faire cela, elle calcule la direction opposée de la fourmilière, et choisit une direction aléatoire pondérée par la rotation relative à cette direction et la quantité de phéromones détectées dans chaque direction.

- Modifier le comportement de votre fourmi dans l'état en sorte qu'elle détecte et suit les phéromones lors de la recherche de nourriture, et les dépose sur les cases lors de son retour à la fourmilière (il faudrait ajouter un mécanisme de stockage pour chaque case).

- Trouvez un moyen d'afficher la quantité de phéromones présente dans chaque case (un dégradé de couleurs donne un joli résultat).
- Ajoutez le code nécessaire pour faire évaporer les phéromones dans le temps.

Exemple:

1. Si la direction vers la fourmilière est 6, alors la direction opposée est 2.

$directionOpposée = (directionFourmilière + 4) \% 8.$

=> La direction 2 aura donc une pondération de rotation de 12, comme d'habitude.

2. La pondération de rotation de la direction est ensuite multipliée par la quantité de phéromones présente sur le bloc correspondant à cette direction.

=> Si le bloc correspondant à la direction 2 contient 50 phéromones, il aura une pondération de 600.

3. **Utilitaires.randomPondere** peut alors être utilisé pour générer la nouvelle direction du fourmi.