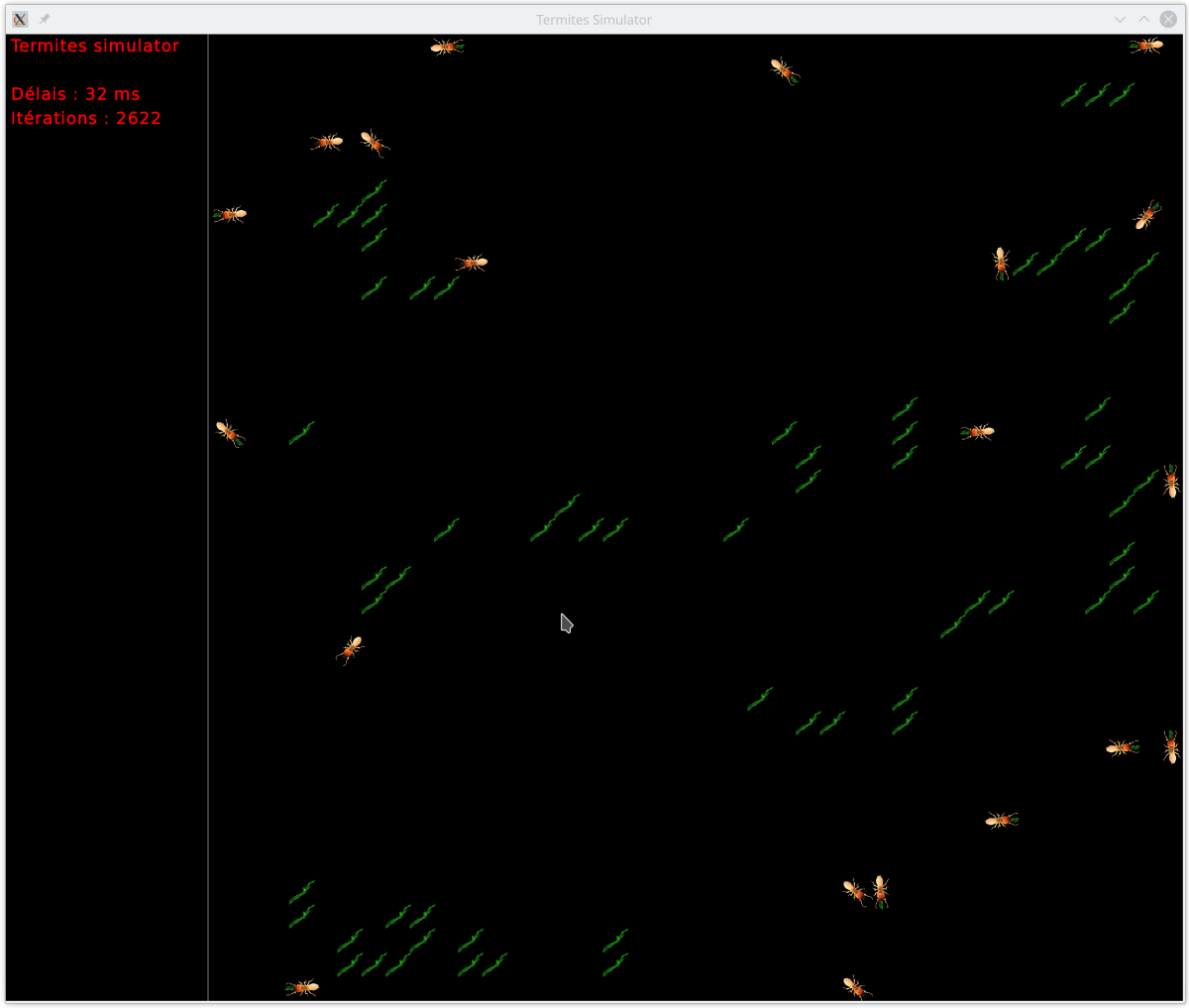
Projet de Programmation

Simulation de termites rassemblant des brindilles

.



1 Principe de l'algorithme mis en œuvre par les termites.

La stigmergie est une méthode de communication indirecte dans un environnement émergent auto-organisé, où les individus communiquent entre eux en modifiant leur environnement. D'après Wikipedia :

> En biologie, la *stigmergie* est un mécanisme de coordination indirecte entre les agents. Le principe est que la trace laissée dans l'environnement par l'action initiale stimule une action suivante, par le même agent ou un agent différent. De cette façon, les actions successives ont tendance à se renforcer et ainsi conduisant à l'émergence spontanée d'activité cohérente, apparemment systématique.

La stigmergie a d'abord été observée dans la nature : les termites, par exemple, utilisent des phéromones pour construire de grandes et complexes structures de terre à l'aide d'une simple règle décentralisée. Chaque termite ramasse un peu de boue autour de lui, y incorpore des phéromones, et la dépose par terre. Comme les termites sont attirés par l'odeur, ils déposent plus souvent leur paquet de boue là où d'autres ont déjà déposé le leur, ce qui forme des piliers, des arches, des tunnels et des chambres.

Dans ce projet, nous allons simuler un comportement particulièrement simple qui ne nécessite pas de phéromone. La tâche est de rassembler des brindilles éparpillées, pour en faire un seul gros tas. Pour ce faire, un termite itère la suite des quatre étapes suivantes :

1. Une marche aléatoire mène le termite vers une brindille.
2. La brindille est ramassée par le termite.
3. Une deuxième marche aléatoire conduit vers un autre tas de brindilles.
4. La brindille portée est posée.

Statistiquement, les petits tas diminuent alors que les gros ont tendance à augmenter. Une fois qu'un seul gros tas est obtenu, il reste en place. L'algorithme est décentralisé et auto-stabilisant. Ce projet est un exemple de simulation de système complexe fait d'un grand nombre d'agents simples. La complexité naît de l'interaction du grand nombre. La compréhension de ce type de système est un défi majeur de l'informatique. La puissance de l'ordinateur moderne est indispensable pour les étudier en permettant leur simulation.

Comportement des termites

Le terrain où évoluent les termites est une grille, implémentée par un tableau à deux dimensions de cases, sur laquelle on pose aléatoirement un certain nombre de termites et de brindilles. On n'autorisera pas les termites à marcher sur les brindilles, ni deux termites à partager une même case, ce qui facilitera l'affichage de la grille.

Un termite ne voit que la case située devant lui mais peut modifier sa direction pour s'orienter vers les 8 points cardinaux {nord-ouest, nord, nord-est, est, sud-est, sud, sud-ouest, ouest}. Il ne peut saisir une brindille que si elle se trouve en face de lui, et ne pourra la reposer que sur une case vide en face de lui.

Les termites se déplacent "en même temps" (on n'attend pas qu'un termite ait trouvé une brindille pour déplacer les autres termites). Cependant comme il n'est pas possible de traiter tous les termites exactement en même temps, on procède étape par étape : on fait faire à chaque termite une étape élémentaire, puis on recomence.

Quelques règles concernant le comportement des termites :

- Lors du déplacement aléatoire, si la case devant est libre, le termite a 90% de chance d'avancer tout droit, sinon il tourne dans une direction au hasard.
- Quand il a ramassé une brindille, il doit attendre un certain nombre de déplacement (par exemple 6) avant de pouvoir reposer la brindille.
- Après s'être déplacé au moins 6 fois, dès qu'il trouve devant lui une autre brindille, il tourne sur place jusqu'à avoir trouvé une case libre et pose sa brindille dans cette case.

  Le termite ne tourne que d'un 8ème de tour à chaque étape de jeu ; il est donc possible qu'il tourne sur sur place pendant un moment avant de trouver une case où poser la brindille. Il est également possible qu'il n'y ai plus aucune case libre car un autre termite l'a enfermé. Dans ce cas, il va tourner sur place jusqu'à ce qu'un autre termite le libère.

  Il est aussi possible qu'en posant sa brindille le termite s'enferme. Dans une amélioration possible de l'algorithme, le termite ne pose la brindille que si il ne s'enferme pas en le faisant, c'est-à-dire s'il existe une autre case voisine libre.
- Il repart ensuite à la recherche d'une autre brindille, mais il ne ramassera pas de nouvelle brindille avant 6 nouveaux déplacements.

Remarque : Le paramètre 6 qui compte le nombre minimal de déplacements sera mis dans une constante de manière à pouvoir essayer ce voir si le comportement est modifié si on le change.

Vous devez réaliser le projet en utilisant les principes vus en info 121, en particulier les types abstraits. Pour vous aider, il y aura deux TD et un TP encadrés, puis probablement une ou deux séances de suivi en TP.

2 TD Types abstraîts pour les termites

L'objectif de ce TD est de faire une analyse ascendante. On cherche la liste des fonctionnalités élémentaires que doivent pouvoir remplir la grille et les termites. Puis, en les combinant, on programme des primitives plus complexes pour aboutir finalement à l'algorithme complet.

2.1 Se repérer sur la grille

Pour se repérer sur la grille, on a besoin d'un type `Coord` pour repérer une case par ses coordonnées et d'un type `Direction` pour représenter une direction parmi nord-ouest, nord, nord-est, est, sud-est, sud, sud-ouest, ouest.

Ce sont des types abstraits tellement simples, qu'ils ne méritent pas vraiment qu'on les traite comme tels. Cependant, il faut s'efforcer de les programmer proprement. En fait, l'analyse réelle commence logiquement par se poser des questions sur le type concret sous-jacent.

- Quels types va-t-on utiliser ?
- En ne considérant que `Coord` et `Direction`, donner la liste des fonctions pour les manipuler (indication : on n'oubliera pas de faire une fonction `egalCoord` qui va nous permettre d'écrire les tests du type abstrait coord).

Les deux types `Coord` et `Direction` seront traîtés en détails dans le premier TP.

2.2 La grille

La grille est un tableau à deux dimensions. Dans chaque case de la grille, il y a plusieurs informations à enregistrer, il faudra donc utiliser un type `Case` qui code les informations nécessaires.

- Quelle(s) information(s) doit enregistrer une case du terrain ?
- Spécifier le constructeur d'une case vide :

Le type `Case` ne sert qu'à la gestion de la grille et n'apparaîtra que dans la déclaration du type concret `Grille` et dans son implémentation. Par conséquent, pour manipuler la grille, il n'y a besoin que de spécifier la grille et la coordonnée de la grille à modifier. Autrement dit, l'utilisateur et le testeur de la grille n'ont pas connaissance du type concret `Case`. Ainsi toutes les fonctions d'interrogation et de manipulation d'une case de la grille devront prendre en paramètre la grille et les coordonnées de la case à interroger ou manipuler.

- Compléter le type abstrait `Grille` ci-dessous. On utilise un pseudo-code dans lequel pour chaque paramètre on doit explicitement écrire Donnée, Résultat ou Donnée-Résultat suivant le mode de passage (ensuite pour votre projet vous devrez traduire ce pseudo-code en C++) :

— Constructeur :

Procédure **initialiseGrilleVide**(_____)
{ Initialise la grille à vide }

— Accès à une case de la grille :

fonction **estVide** (_____) → _____
{ retourne vrai si la grille ne contient ni brindille, ni termite à la position donnée }

fonction **dansGrille** (_____) → _____
{ retourne vrai si la position indiquée est bien dans la grille }

fonction **contientBrindille** (_____) → _____
{ retourne vrai si la grille contient une brindille à la position indiquée }

fonction **numeroTermite** (_____) → _____
{ retourne le numéro du termite qui est dans la grille à la position indiquée, -1 si pas de termite à cet endroit }

— Modifications d'une case de la grille :

procédure **poseBrindille** (_____)
{ pose une brindille dans la grille à la position indiquée }

procédure **enleveBrindille** (_____)
{ enlève de la grille la brindille située à la position indiquée }

procédure **poseTermite** (_____)
{ pose à la position indiquée le termite de numéro donné en 3ème paramètre }

procédure **enleveTermite** (_____)
{ enlève de la grille le termite situé à la position indiquée }

2.3 Les termites

On représentera les termites par un type abstrait `Termite`. Les termites seront rangés dans un tableau et chaque termite aura un numéro correspondant à sa position dans le tableau.

- Quelles informations doit contenir un termite ?
- Quel(s) paramètre(s) passer pour construire un termite ?

Dans le type abstrait Termite, il y a deux types de fonctions : celles qui ne concernent que le termite lui-même, et celles qui concernent également la grille. Nous commençons par les fonctions simples qui ne concernent que le termite lui-même :

fonction **directionTermite**(_____) → _____
{ La direction du termite }

fonction **devantTermite**(_____) → _____
{ Les coordonnées de la case devant le termite }

fonction **porteBrindille**(_____) → _____
{ Est-ce que le termite porte une brindille }

Procédure **tourneADroite**(_____)
{ Le termite tourne à droite }

Procédure **tourneAGauche**(_____)
{ Le termite tourne à gauche }

Procédure **tourneAleat**(_____)
{ Le termite tourne aléatoirement }

Il y a ensuite une deuxième série de fonctions qui manipulent non seulement le termite, mais aussi certaines cases de la grille :

- Les fonctions de test de l'environnement du termite

  fonction **laVoieEstLibre**(_____) → _____
  { la case devant le termite est libre }

  fonction **brindilleEnFace**(_____) → _____
  { la case devant le termite contient une brindille }

  fonction **pasEnferme**(_____) → _____
  { le termine ne s'enferme pas s'il pose une brindille devant lui
  C'est-à-dire, l'une des autres cases voisines est libre }

- Les procédures qui modifient le termite et son environnement :

  procédure **avanceTermite**(_____)
  { le termite avance (la case devant lui est supposé libre) }

  procédure **dechargeTermite**(_____)
  { le termite pose sa brindille devant lui (la case devant lui est supposée libre) }

  procédure **chargeTermite**(_____)
  { le termite prend une brindille (la case devant lui est est supposée contenir une brindille) }

  procédure **marcheAleatoire**(_____)
  { le termite fait un déplacement aléatoire) }

- En utilisant les fonctions des types abstraits `Termites` et `Grille`, écrire les fonctions `laVoieEstLibre`, et `brindilleEnFace`.
- Écrire la procédure `chargeTermite`.
- Écrire la procédure `avanceTermite`.
- Écrire la procédure `marcheAléatoire`.

3 Mise en forme du programme

- On recommande de mettre en début de programme toutes les constantes utilisées, pour pouvoir les voir et les modifier facilement. Quelles sont ces constantes ?
- Comment faut-il faire pour faire apparaître clairement le type abstrait **termite** au niveau du programme C++ ?
- Que faut-il faire pour rendre le programme lisible ?

4 Travail personnel demandé et noté.

Pour vous aider, nous indiquons une suite d'étapes permettant d'avancer progressivement, et aboutissant à un effet testable le jour de la soutenance. D'autre part, deux bugs sont fréquents. Si votre programme plante inopinément, vous avez sans doute tenté d'adresser une case dont les coordonnées sont en dehors du terrain. Si votre programme ne s'arrête jamais, c'est probablement que l'un de vos termites est entré dans une boucle qui ne s'arrête pas. Pour éviter cela, éviter les boucles dans les primitives du termite.

Le barème est donné à titre purement indicatif. De plus, pour pouvoir prétendre avoir les points d'une partie, il ne suffit pas de fournir du code correspondant, mais il faut démontrer sa maîtrise du code.

4.1 Affichage du terrain (3pt)

Cela correspond au travail réalisé dans le TP de démarrage.

4.2 Affichage de la direction des termites (1,5pt)

Modifier le type concret des termites pour stocker leur direction (entier dans 0 .. 7), initialiser celle-ci aléatoirement, et afficher le termite avec le caractère '/', '\' , '-' , '|' suivant sa direction.

Les points de cette question ne peuvent être donnés que si la question suivante est également traitée, de sorte qu'on puisse vérifier visuellement la cohérence du mouvement avec la direction affichée.

Vous devriez avoir un affichage ressemblant à

4.3 Mouvement des termites (3 pt)

Bouger les termites aléatoirement, en modifiant leur direction avec une probabilité de 1 sur 10. Ainsi, en moyenne, le termite va tout droit dix fois de suite. Les termites ne peuvent se déplacer vers une case déjà occupée par une brindille, ou un autre termite. Il faut mettre en place la boucle qui itère une passe, puis demande à l'utilisateur une saisie d'un caractère de contrôle avant de continuer (getcar()). Le programme termine si l'utilisateur tape '.' sinon il continue une passe. Une simple pression sur return provoque directement une nouvelle passe.

4.4 Contournement des obstacles (1pt)

Observer que les termites restent bloqués longtemps sur les obstacles. En particulier la majorité se retrouvent coincés au bord de la grille. Pour éviter cela, les termites mutent, la nouvelle génération décide systématiquement de tourner (au hasard) si un obstacle survient en face.

4.5 Contrôle de l'exécution (1pt)

On souhaite vérifier l'exécution après un grand nombre d'itérations, pour pouvoir s'assurer directement si le regroupement en un seul tas fonctionne, sans avoir à taper un grand nombre de fois return, pour itérer un grand nombre de passes. Utiliser une variable *nbPasse*, et réaliser *nbPasse* entre chaque affichage. Initialiser d'abord directement cette variable à 10,100,1000, puis donner la possibilité à l'utilisateur de multiplier par deux (resp. diviser par deux) cette variable, s'il saisit le caractère '*' (resp. '/'). Cela permet d'accélérer et de ralentir l'exécution.

4.6 Rassembler les brindilles (5 pt)

Modifier le type concret des termites pour ajouter les variables d'état du termite (sablier, brindille, tourneSurPlace). Implémenter *chargerBrindille* et *dechargerBrindille*. Puis l'algorithme principal *rassemblerBrindille*, en prenant progressivement en compte ces nouvelles variables d'état.

4.7 Séparation du type abstrait (2pt)

Séparer le code relatif aux termites dans un fichier termite.cpp, en ajoutant également un fichier termites.hpp pour les signatures des primitives. Cela vous oblige peut-être à passer la grille en paramètre pour les fonctionnalités du termite qui en ont besoin.

4.8 Préserver la liberté du termite (1pt)

Observer, ou tester que les termites peuvent s'enfermer tout seul dans un mur de brindilles, en posant une brindille sur la case par où ils sont arrivés. Modifier *rassemblerBrindille* pour éviter cela. (Cela prend trois lignes de programmes).

4.9 Présentation (1,5 pt)

Ces points récompensent la qualité de présentation orale et du code : modularité, lisibilité, gestion des problèmes rencontrés, affichage d'informations intermédiaires, et surtout, commentaires.

4.10 Bonus (1 points et plus...)

Ces points récompensent les initiatives personnelles, l'originalité, ou les caractéristiques qui distinguent le programme des autres. Voici des suggestions :

- affichage graphique (on suggère l'utilisation de la bibliothèque SFML) ;
- comportement des termites amélioré (les termites peuvent se passer des brindilles de l'un à l'autre) ;
- Grille plus compliquée (quand on sort à droite, on rentre à gauche...), tunnels qui passent directement d'une case à l'autre...

4.11 Optionnel (Pour la gloire et la science ;-))

Ce qui suit est réellement difficile, et n'est pas du tout demandé. C'est la suite logique, et si certains étudiants aiment bien programmer, et ont le temps, ils peuvent s'y frotter. On observe que le tas ne reste pas forcément connexe, une fois qu'il est formé. La raison est que un termite peut enlever une brindille qui sépare le tas en deux. Modifier le comportement pour éviter cela. Tester la connexité du tas, vérifier qu'il reste connexe, utiliser le temps moyen de mise en tas connexe, optimiser les constantes du modèle (durée du sablier, probabilité de tourner) en fonction des densités en termites et brindilles, pour obtenir une mise en tas la plus rapide possible. Commenter les résultats obtenus (comparer avec des vrais termites ? !).