Projet : Liquid War

Projet à réaliser en binôme, à rendre sur Moodle, avant la date indiquée sur Moodle.
Le langage doit être Java (version 21).
Le code doit être écrit par vous, mais votre programme peut dépendre de bibliothèques tierces (celles indiquées dans le sujet ou des bibliothèques rendant un service équivalent).

I) À propos

Liquid Wars [1] est un jeu de stratégie en temps réel conçu par Thomas Colcombet, puis développé par Christian Mauduit, qui se base sur une armée "fluide" de particules se déplaçant sur une carte en 2D selon un algorithme de plus court chemin.
Le sujet consiste à implémenter ce jeu en Java en tirant le plus partie des possibilités *multi-thread* de la plateforme.

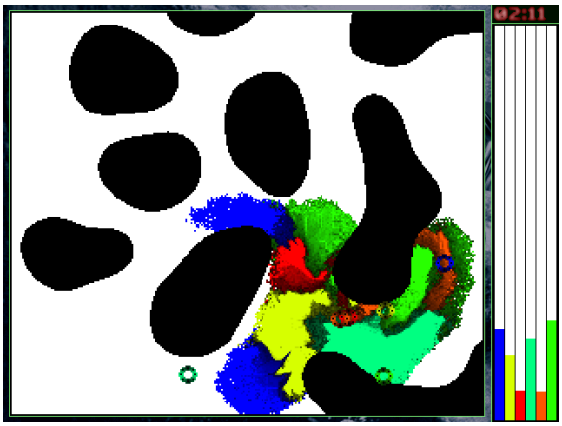

Fig. 1 – Capture d'écran du jeu Liquid War (via Wikipédia)

II) Déroulement du jeu

Chaque équipe possède chacune une armée de particules (pixels colorés, une couleur pour chaque joueur), réparties sur une carte en 2D, jonchées d'obstacles infranchissables.
Un joueur contrôle un curseur/cible qu'il peut déplacer librement sur la carte (typiquement à la souris). Chaque particule devra se diriger vers la cible de son équipe la plus proche, en contournant les obstacles via un algorithme de plus court chemin.
Pour une particule p donnée, les autres particules de la même couleur constituent des obstacles qui seront évités au même titre que les murs, quant aux particules d'une autre couleur (autre joueur), si elles se trouvent sur l'emplacement où p doit aller, selon l'algorithme, elle est alors attaquée et convertie vers la même couleur que p dès lors qu'elle n'a plus d'énergie (le nombre de particules reste constant).
Une partie se termine lorsqu'une équipe contrôle toutes les particules.
L'algorithme est décrit plus précisément ici : `https://ufoot.org/liquidwar/v5/techinfo/algorithm`.
Voilà, dans les grandes lignes ce qu'il en retourne :

Plus court chemin par gradient Le plus court chemin n'est pas recalculé pour chaque particule, mais plutôt, pour chaque équipe, un gradient est calculé, indiquant la distance à la cible la plus proche, via un algorithme de parcours de graphe sur les pixels de la carte (ici, en largeur, pour l'exemple, mais ce sera à vous de voir) :

1. initialement, le score 0 est affecté aux pixels contenant les cibles de l'équipe,

[1] `https://ufoot.org/liquidwar`

2. puis les voisines (sauf obstacles) des cases qui ont des 0 se voient affecter 1,
3. puis on ré-itère : à l'étape i, les voisines (sauf obstacles) non encore marquées des cases de score i reçoivent le score $i+1$, jusqu'à recouvrir toute la carte (sauf les obstacles, qui gardent le score ∞).

Remarque : ce n'est pas exactement l'algorithme implémenté par Christian Mauduit, qui utilise une optimisation (approximative) basée sur des mailles moins fines que de simples pixels. Vous pourrez implémenter la version optimisée en un second temps !

Déplacement des particules Remarque : à tout instant, un pixel contient au plus une particule (mais il peut aussi être vide ou contenir un obstacle).
Pour chaque particule, pour chaque direction de déplacement possible (chaque pixel voisin), évalue dans laquelle de ces 4 situations on est :

- direction principale (valeur de gradient minimale)
- bonne direction (valeur de gradient plus petite que la valeur de gradient du pixel où se trouve la particule)
- direction acceptable (valeur de gradient identique)
- direction impossible (valeur plus grande, qui nous éloigne de l'objectif)

La particule va se comporter comme dans le premier cas qui s'applique, parmi la liste suivante :

1. Si une direction principale est libre, on y va.
2. Si une bonne direction est libre, on y va.
3. Si une direction acceptable est libre, on y va.
4. Si une direction principale est occupée par un ennemi, on l'attaque.
5. Si une bonne direction est occupée par un ennemi, on l'attaque [2].
6. Si une direction principale est occupée par un ami, on lui transfère de notre énergie [3].
7. Sinon, on ne fait rien.

Dans votre implémentation, vérifiez que le nombre de particules ne change pas et que la quantité totale d'énergie est aussi constante.

III) À programmer

La consigne est simple : en utilisant un maximum de principes de POO vus en cours cette année, en particulier le *multithreading*, programmer le jeu de liquid war.
Les fonctionnalités à implémenter (en ordre de priorité) :

1. Algorithme de calcul du gradient.
2. Application des règles de déplacement sur une population de cellules de plusieurs couleurs (étant donné un gradient pour chaque couleur)
3. interface graphique permettant de déplacer une cible à la souris ; les gradients, ainsi que les déplacements des particules sont recalculés à chaque *frame* du jeu (à chaque événement de rafraîchissement de l'UI, typiquement 60 fois par seconde), et l'affichage est mis à jour. On assigne une couleur de base (saturée) à chaque équipe. La couleur exacte sera plus ou moins brillante en fonction de l'énergie de la particule.

[2] C'est à dire qu'on lui vole de l'énergie. Dès que l'énergie de la cible passe sous le seuil minimum, la particule attaquée est convertie vers notre couleur.

[3] Le document ne précise pas, mais on va supposer qu'on ne donne de l'énergie que si on en a plus que le minimum et que la particule à laquelle on donne a moins que le maximum !

4. Implémentation de joueurs contrôlés par algorithmes (des "IA", quoi...). Vous pouvez commencer par des cibles fixes, puis des déplacements aléatoires, puis des stratégies plus intelligentes selon votre inspiration.
5. Multi-joueur local.
6. Optimisations *multithread*ées. Réfléchissez à ce qui peut être parallélisé et à ce qui doit être synchronisé !

   Remarque : il est possible de paralléliser le calcul de gradient, mais c'est probablement très technique (pour le faire à la fois correctement et efficacement).
7. Calcul de gradient amélioré (méthode des mailles).
8. Multi-joueur en réseau.
9. Implémentez d'autres options existant dans Liquid War 6 (ou autre version).

IV) Contraintes techniques

Concurrence Le plus facile est d'utiliser les *threads* virtuels. N'oubliez pas que l'affichage lui-même se fait via le *thread* applicatif de votre toolkit graphique, qu'il faudra donc aussi synchroniser avec vos *threads* de travail.
Si vous faites du réseau, il est aussi à prévoir un *thread* virtuel de plus par *socket* en écoute.
Dans tous les cas, prenez soin de vérifier la bonne synchronisation. Privilégiez les structures immuables et limitez le nombre de variables partagées.

Testabilité Tout le code que vous écrirez devra avoir été programmé de telle sorte à ce que le comportement soit testable.
Le découpage en méthodes doit être suffisamment fin pour permettre des tests unitaires pertinents.
Le découpage en classes et la programmation « à l'interface » doivent permettre d'effectuer facilement des tests d'intégration. Cela peut être fait en remplaçant n'importe quel composant par un composant factice implémentant la même interface.
Les tests devront être écrits, exécutables, et fournis avec le rendu.

V) Aide technique

- Utilisez un système de compilation tel que Maven ou Gradle. Vous faciliterez la gestion des dépendances (exemples fournis).
- Interface graphique : Swing, JavaFX [4] (vous pouvez aussi consulter les transparents du cours sur les IG, qui sont basés sur JavaFX) ou autre. Si JavaFX, utilisez le plugin JavaFX pour Maven ou Gradle, vous vous simplifierez la vie. Pour tout autre toolkit que Swing, l'installation des dépendances doit se faire automatiquement (bref, utilisez Maven ou Gradle).
- Réseau : vous pouvez utiliser une connexion TCP. Regardez la documentation des classes `java.net.Socket` et `java.net.ServerSocket`. Vous trouverez aussi sur votre moteur de recherche préféré plusieurs cours d'enseignants de l'UPC qui expliquent ces classes. Une démo de *chat* TCP vous est fournie en exemple.
- Les tests peuvent être écrits dans un *framework* tel que JUnit5 [5], mais ce n'est pas indispensable. En tout cas, il faut des tests.

[4] `https://openjfx.io/`
[5] `https://junit.org/junit5/`

VI) Critères d'évaluation

- Le projet est rendu dans une archive `.zip`.
- Le projet doit contenir un fichier `README.md` indiquant comment compiler, exécuter et utiliser votre programme et ses tests, décrivant les fonctionnalités effectivement implémentées et expliquant vos choix techniques originaux, le cas échéant.
- Joignez aussi un ou des diagrammes de classe.
- La commande pour compiler ou exécuter doit être simple (s'aider de Gradle, Maven... voire de Makefile).
- La compilation selon ces instructions doit terminer sans erreur ni avertissement.
- L'exécution doit être correcte : elle respecte le cahier des charges et ne quitte pas de façon non contrôlée.
  Notamment, en mode graphique, les exceptions doivent être rattrapées. Dans tous les cas, si le problème n'est pas réparable, un message d'erreur doit être présenté à l'utilisateur. Aucune de ces erreurs ne doit être due à un problème de programmation (comme le fameux `NullPointerException`...).
- Le code respecte les conventions de codage.
- Le code est architecturé intelligemment. Notamment, il faut utiliser des patrons de conception vus en cours ainsi que les constructions les plus appropriées fournies par Java.
- Les classes et méthodes sont documentées (javadoc).
- Des commentaires expliquent les portions de codes qui ne sont pas évidentes.
- Les tests fournis doivent être aussi exhaustifs que possible.
- Un projet qui tenterait de tout faire mais dans lequel rien ne marche bien sera moins bien noté qu'une projet qui ne remplit que la première moitié des objectifs sans avoir de bug. Les objectifs sont incrémentaux. Il est toujours possible de rendre un projet cohérent qui n'aurait atteint que les premiers objectifs, donc profitez-en !