Ant Colony Rush
Rapport de projet de programmation concurrentielle et interface interactive -

Esther DEBA

Ania GAROUI

Johan MARTIN-BORRET

Amine OUATMANI

Université Paris-Saclay

L3 Informatique

2025

Table des matières

1 Introduction

Histoire

Il y a fort longtemps, la Reine des Fourmis découvrit un territoire exceptionnellement fertile. D'une simple clairière, elle fit le berceau d'un nouvel empire, certain que rien ne viendrait perturber la quiétude et la prospérité de son peuple. Les fourmis y vécurent des années dans la plus parfaite paix, s'agrippant à chaque brin d'herbe et butinant chaque parcelle de terre à la recherche de ressources toujours plus précieuses.

Hélas, un soir où grondait l'orage, la nature se déchaîna et une tempête sans précédent dévasta les alentours. Au milieu des bourrasques et des fracas célestes, un hôte imprévu et pour le moins *indésirable* fut emporté jusqu'aux abords du nid : un crapaud imposant, aux yeux globuleux et à l'appétit insatiable. Dès cet instant, l'insouciance des fourmis prit fin et un climat de terreur s'installa.

Sous la menace permanente du prédateur, chaque instant de récolte et de déplacement devint un jeu dangereux. Rien n'attise autant la peur qu'un batracien aux aguets, prêt à fondre sur la moindre fourmi égarée. Les récoltes furent alors menées en hâte, dans une tension constante : plus question de se prélasser au soleil ou de festoyer comme autrefois. Dans la cité, les couloirs du nid résonnaient de rumeurs et de chuchotements inquiets. L'idée d'un *exode* germa dans l'esprit de la Reine : pourquoi ne pas accumuler suffisamment de ressources pour bâtir un nouveau foyer, loin du danger ?

Depuis, les fourmis n'ont plus qu'un seul objectif : amasser autant de nourriture et de matériaux que possible, afin de pouvoir *déménager* et échapper définitivement aux assauts du crapaud. Chaque jour, elles s'organisent, élaborent des stratégies, explorent plus loin, bravent l'inconnu, animées par l'espoir d'une nouvelle terre «safe». Au cœur de cet univers, vous êtes invité à prendre part à leur destinée, à déjouer les plans du crapaud et à orchestrer la grande évasion des fourmis. Le temps presse, car dans ce monde impitoyable, seule la vigilance et la persévérance permettront à la colonie de survivre et de se forger un avenir meilleur.

Présentation du jeu

Ant Colony Rush est un jeu de simulation stratégique captivant, plaçant le joueur à la tête d'une colonie de fourmis déterminée à prospérer dans un environnement rempli d'opportunités et de dangers. Votre objectif :

- **Collecter** autant de ressources que possible.
- **Gérer** l'énergie et la santé des fourmis pour qu'elles accomplissent leurs tâches.
- **Éviter** ou contourner le crapaud, dont le champ de vision et la faim menacent en permanence les ouvrières.
- **Améliorer** et agrandir le nid en recrutant de nouvelles fourmis, afin de récolter plus vite et survivre plus longtemps.
- **Atteindre** un niveau de ressources suffisant pour migrer vers un nouveau territoire et mettre la colonie à l'abri du danger.

Ant Colony Rush propose ainsi une expérience immersive unique, combinant gestion de ressources, stratégie en temps réel, et prise de décisions sous pression. Chaque choix compte, faisant de chaque partie une aventure pleine de suspense et de rebondissements.

Cadre de réalisation

Ce jeu a été développé dans le cadre d'un projet universitaire supervisé par Monsieur Nicolas Sabouret, dans son module de Programmation Concurrente et Interfaces Interactives. L'objectif pédagogique était double : nous initier aux principes de la programmation concurrente à travers les threads et maîtriser la création d'interfaces interactives avec Swing en Java. En parallèle, ce projet nous a permis de développer des compétences essentielles en gestion de projet et en travail collaboratif, incluant la rédaction technique, l'organisation méthodique et la planification efficace des tâches.

2 Analyse globale

Cette analyse présente les principales fonctionnalités à développer pour le projet « Ant Colony Rush » ainsi que leur niveau de difficulté et leur priorité. L'approche se décline en quatre grands pôles : Interface et Navigation, Dynamique des Personnages et Comportements, Gestion des Structures et des Ressources, et Fonctionnalités Complémentaires.

2.1 Interface et navigation

2.1.1 Menu principal et sélection de la difficulté

Le menu principal permet au joueur de démarrer la partie et de sélectionner le niveau de difficulté. Cette fonctionnalité est relativement simple à mettre en œuvre.

— **Difficulté :** *Facile*
— **Priorité :** *Moyenne*

2.1.2 Panneaux de contrôle du jeu (Nid, abris, ressources)

Les panneaux de contrôle fournissent une vue d'ensemble en temps réel des éléments stratégiques du jeu (score, nid, ressources, etc.). Leur mise à jour fluide et l'interaction intuitive avec l'utilisateur sont essentielles.

— **Difficulté :** *Moyenne*
— **Priorité :** *Haute*

2.1.3 Option de pause et gestion du chronomètre

Permettre la mise en pause du jeu ainsi que la gestion d'un chronomètre assure une meilleure flexibilité pour le joueur.

— **Difficulté :** *Facile*
— **Priorité :** *Faible*

2.1.4 Sauvegarde du score et du meilleur record

La sauvegarde et la mise à jour des scores, ainsi que la mémorisation des records, sont essentielles pour la rejouabilité.

— **Difficulté :** *Facile*
— **Priorité :** *Faible*

2.2 Dynamique des personnages et comportements

2.2.1 Les fourmis

Les fourmis constituent le cœur dynamique du jeu : elles doivent se déplacer, s'animer, consommer et régénérer de l'énergie, et parfois mourir.

— **Difficulté :** *Moyenne*
— **Priorité :** *Haute*

2.2.2 Le crapaud

Le crapaud est un personnage clé dont le comportement est plus complexe : il se déplace de façon autonome, interagit avec les fourmis et les ressources, et change d'état (faim, sommeil).

— **Difficulté :** *Haute*
— **Priorité :** *Haute*

2.3 Gestion des structures et des ressources

2.3.1 Nid

Le nid est la structure centrale qui accueille les fourmis et gère le score.

— **Difficulté :** *Facile*
— **Priorité :** *Haute*

2.3.2 Abris

Les abris servent à stocker les fourmis et disposent d'une capacité définie. Le développement est similaire à celui du nid et s'appuie sur des éléments visuels simples.

— **Difficulté :** *Facile*
— **Priorité :** *Haute*

2.3.3 Gestion des ressources

Ce module est le plus complet et complexe dans cette catégorie :

a. **Génération initiale et en cours de partie** : La création de ressources se fait aléatoirement et doit respecter un nombre maximal et éviter toute collision.
b. **Animation de pousse** : Ce volet demande un soin particulier pour assurer une transition visuelle fluide.
c. **Déplacement vers le nid** : La logique de mouvement des ressources doit être coordonnée avec l'arrivée au nid et s'articuler autour d'algorithmes précis.

— **Difficulté :** *Haute*
— **Priorité :** *Haute*

2.4 Fonctionnalités complémentaires

2.4.1 Histoire du jeu

L'introduction narrative permet d'immerger le joueur dans l'univers. L'aspect rédactionnel et l'intégration dans l'interface (écrans dédiés ou séquences introductives) impliquent une difficulté modérée.

— **Difficulté :** *Modérée*
— **Priorité :** *Faible*

2.4.2 Effets sonores et musique de fond

L'ambiance sonore (effets et musique) renforce l'immersion.

— **Difficulté :** *Faible à Moyenne*
— **Priorité :** *Moyenne à Haute*

2.4.3 Mode jour/nuit

Le mode jour/nuit, qui modifie l'atmosphère visuelle.

— **Difficulté :** *Moyenne*
— **Priorité :** *Haute*

2.5 Pluie

L'ajout d'effets météorologiques, tel que la pluie, constitue un supplément esthétique.

— **Difficulté :** *Moyenne*
— **Priorité :** *Faible*

L'analyse met en évidence que la priorité doit être donnée aux mécanismes centraux du jeu : l'interface de contrôle, la dynamique des fourmis et du crapaud, ainsi que la gestion des structures (nid, abris) et des ressources. Ces éléments, jugés faciles à mettre en œuvre ou moyennement complexes, constituent la base d'une expérience de jeu fluide et stratégique. À ce socle s'ajoutent des fonctionnalités complémentaires – telles que l'histoire, l'ambiance sonore, le mode jour/nuit et les effets météorologiques – qui, bien que non critiques pour la jouabilité, enrichiront l'immersion et la profondeur narrative du jeu.

3 Organisation et Plan de Développement

Durant la réalisation du projet, nous avons employé plusieurs outils afin d'optimiser notre organisation et assurer un suivi efficace des différentes tâches. Parmi ces outils, nous avons utilisé principalement :

— **Un tableau Trello** permettant une gestion visuelle des tâches à réaliser, en cours et achevées. (Voir capture d'écran ci-dessous).

— **Un serveur Discord** facilitant la communication instantanée et la collaboration entre les membres de l'équipe.
— **Un diagramme de GANTT** indispensable à la planification temporelle du projet.

Le diagramme de GANTT nous a particulièrement aidés à répartir clairement les responsabilités et à suivre l'état d'avancement du projet tout au long de son développement. Cet outil s'est avéré crucial pour respecter les délais impartis.

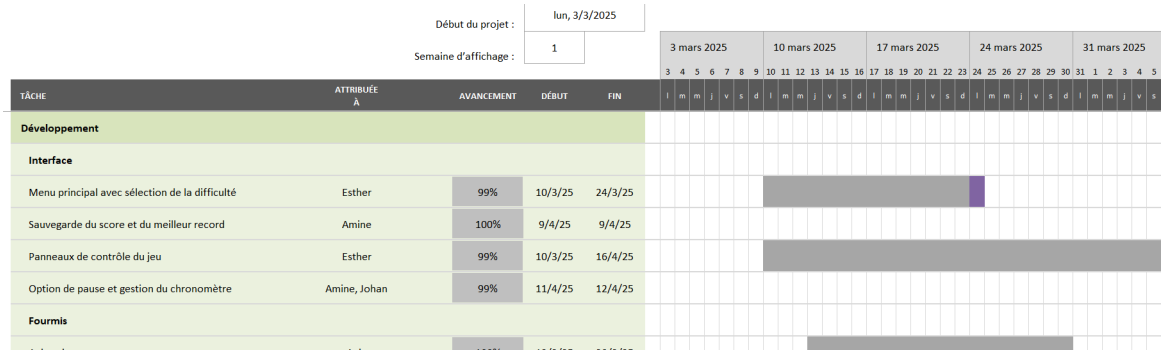


FIGURE 1 – Diagramme de GANTT utilisé pour le suivi du projet

Note : Le diagramme complet est présent dans le dépôt Git, dans le dossier `doc/`.

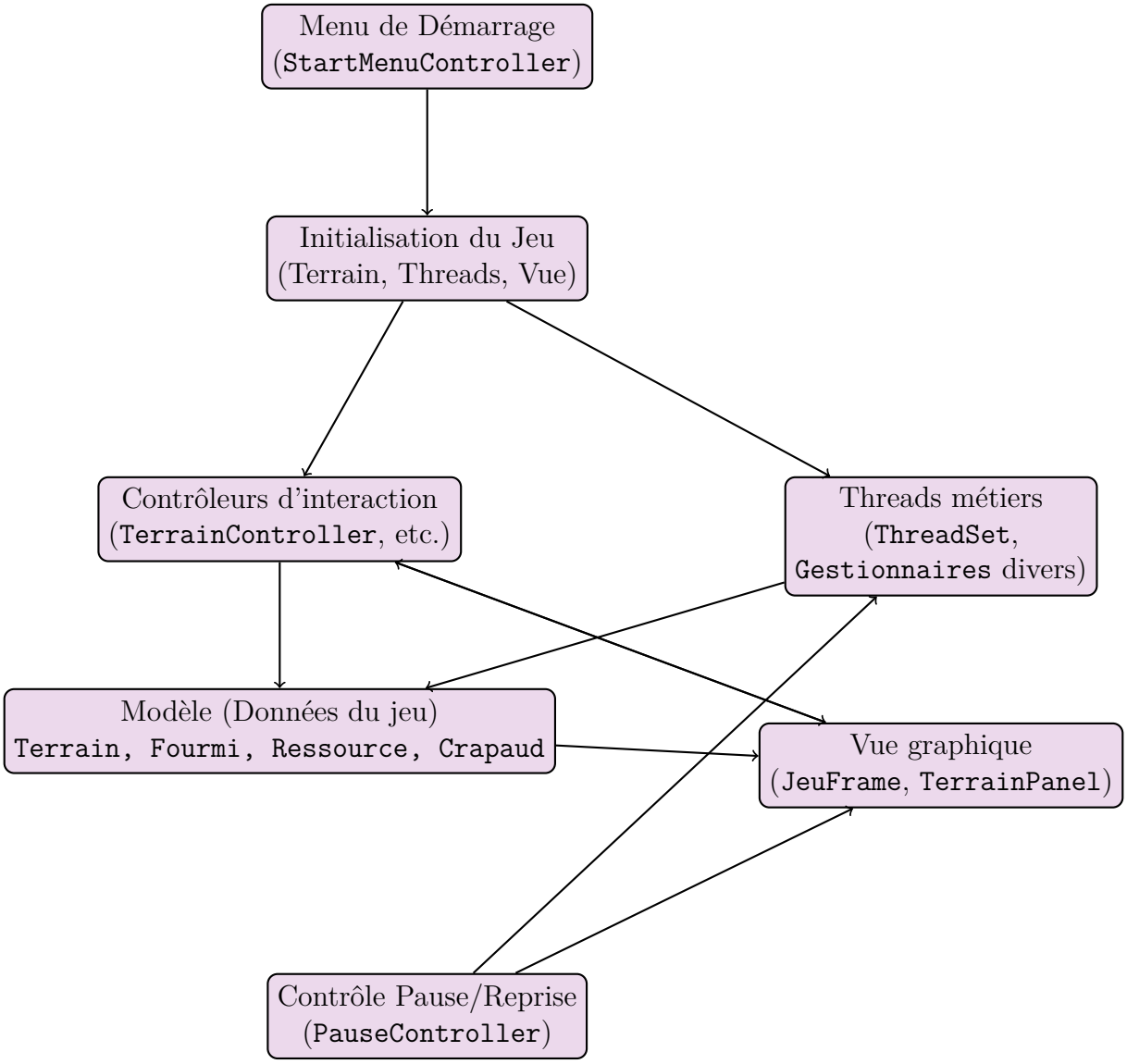
4 Conception générale

Le projet *Ant Colony Rush* est une simulation en temps réel basée sur l'architecture **MVC** (Modèle-Vue-Contrôleur). Il est conçu autour de plusieurs blocs fonctionnels qui communiquent pour gérer la logique de jeu, l'affichage et les interactions.

4.1 Structure du Projet

— **Modèle** (`model.*`) : contient la logique métier du jeu (terrain, entités, score, threads).
— **Vue** (`view.*`) : affiche les éléments visuels du jeu (terrain, interface graphique, animation).
— **Contrôleur** (`controller.*`) : gère les événements utilisateurs (clavier, souris), initie le jeu, contrôle les flux entre la vue et le modèle.

4.2 Blocs Fonctionnels et Communication



4.3 Explication de l'architecture

— Le **menu de démarrage** permet de choisir la difficulté et de lancer une partie via le `StartMenuController`

— L'**initialisation du jeu** prépare le terrain, les entités, les gestionnaires de threads, la fenêtre de jeu (`JeuFrame`) et la vue (`TerrainPanel`)
— Les **contrôleurs d'interaction** comme `TerrainController` ou `PauseController` gèrent les clics souris, les touches clavier, et déclenchent des actions sur le modèle ou les threads
— Les **threads métiers** sont regroupés dans un objet centralisé (`ThreadSet`) qui contrôle l'exécution parallèle des composants comme :
  — `GestionnaireEnergie`
  — `GestionnaireDeplacement`
  — `GestionnaireCrapaud`
  — `GestionnaireRessources`
  — `GestionnaireFourmisMortes`
  — `GestionnaireSelection`
  — `GestionScore`

  Ces gestionnaires héritent d'un thread commun (`InterruptibleThread`) et peuvent être mis en pause/repris collectivement via `PauseController`
— Le **modèle** contient toutes les entités (terrain, fourmis, ressources, crapaud, score, etc.) mises à jour par les threads et contrôlées par les événements.
— La **vue** affiche les entités du modèle et reçoit les événements utilisateur pour les rediriger vers les bons contrôleurs

4.4 Threads principaux

Voici les principaux threads utilisés dans la simulation :

— `GestionnaireEnergie` : met à jour régulièrement l'énergie des fourmis.
— `GestionnaireDeplacement` : anime les déplacements sur le terrain.
— `GestionnaireCrapaud` : fait évoluer le comportement du prédateur.
— `GestionnaireRessources` : ajoute ou fait croître les ressources sur le terrain.
— `GestionnaireFourmisMortes` : gère la suppression visuelle des fourmis décédées.
— `GestionnaireSelection` : affiche les entités sélectionnées dans la vue.
— `GestionScore` : permet la création de nouvelles fourmis quand le score le permet.

5 Conception détaillée

5.1 Menu principal et sélection de la difficulté

Structures de données utilisées

— `MenuDemarrage` : Composant Swing représentant l'écran de démarrage.
— `DifficultePanel` : Panneau qui permet au joueur de choisir le niveau de difficulté (Facile, Moyen, Difficile).
— `StartMenuController` : Contrôleur qui gère les actions du menu, lit le choix effectué dans `DifficultePanel` et lance la partie en configurant le `Terrain` avec la difficulté sélectionnée.

Constantes du modèle

— Les dimensions de l'interface (par exemple, taille fixe de la fenêtre dans `MenuDemarrage`).
— Les valeurs préconfigurées dans l'énumération `Difficulte` (nombre initial de fourmis, abris, ressources).

Algorithme abstrait

1. À l'ouverture de l'application, le `MenuDemarrage` est affiché.
2. L'utilisateur sélectionne la difficulté à l'aide du `DifficultePanel`.
3. En cliquant sur le bouton « Start », le `StartMenuController` récupère la difficulté choisie et crée le `Terrain` en appelant le constructeur approprié (par exemple, `new Terrain(difficulte)`).
4. Le contrôleur lance ensuite l'initialisation des gestionnaires et la création de la fenêtre de jeu (`JeuFrame`), qui intègre l'interface complète.

Conditions limites à respecter

— La difficulté doit être correctement sélectionnée (aucun choix null ou invalide).
— La configuration du `Terrain` doit prendre en compte la difficulté choisie, c'est-à-dire que le nombre d'éléments (fourmis, abris, ressources) correspond aux paramètres définis dans l'énumération `Difficulte`.

Utilisation par les autres fonctionnalités

Le résultat de cette étape (la configuration initiale du `Terrain` et le choix de difficulté) est utilisé par :

— Les gestionnaires de la simulation (énergie, déplacement, ressources, etc.) qui récupéreront ces paramètres.
— L'interface de jeu, qui se construit selon la configuration initiale.

5.2 Panneaux de contrôle du jeu

Structures de données utilisées

— `ConteneurPanneauDeControle` : Panneau global qui regroupe la partie fixe (tableau de bord) et la partie dynamique (détails sur les objets sélectionnés).
— `PanneauDeTableauDeBord` : Affiche en temps réel les informations du jeu (score, temps écoulé, informations sur le crapaud).

— `PanneauCartesFourmis` : Gère l'affichage des informations des fourmis (via des cartes individuelles, par exemple `CardFourmis`).

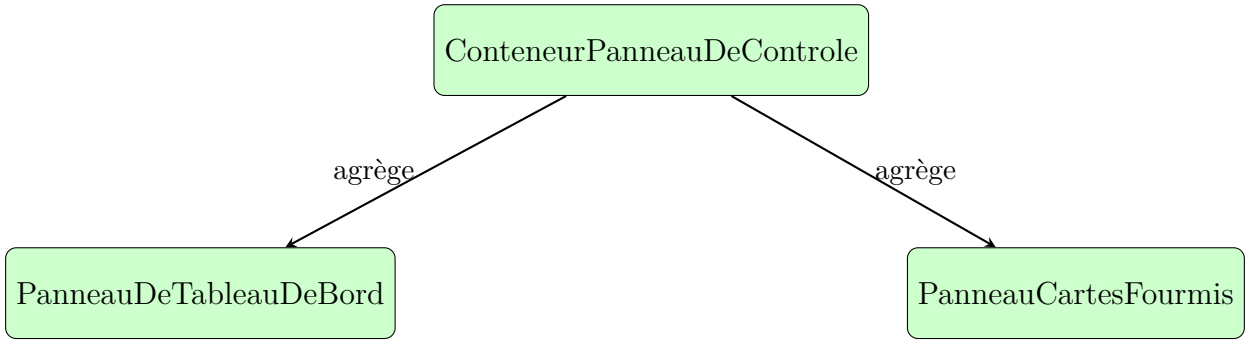
Constantes du modèle

— Dimensions fixes des panneaux et des composants graphiques.
— Paramètres pour l'affichage des barres (exemple : plage de la barre d'énergie de 0 à `Fourmi.MAX_ENERGIE`).

Algorithme abstrait

1. À l'initialisation du jeu (via `JeuFrame`), le `ConteneurPanneauDeControle` est créé et positionné sur le côté de la fenêtre.
2. Le `PanneauDeTableauDeBord` est intégré en haut du conteneur pour afficher les informations statiques telles que le score et le chronomètre.
3. Une zone dynamique est prévue pour afficher des détails sur les objets (nid, abris, ressources) lorsque l'utilisateur clique sur ces derniers dans le `TerrainPanel`.
4. Les mises à jour se font par des appels réguliers (notamment via un Swing Timer dans `PanneauDeTableauDeBord`) pour rafraîchir l'affichage.

Diagramme de classes (simplifié)



Conditions limites à respecter

— Les informations affichées (score, temps, état du crapaud) doivent être synchronisées avec le modèle en temps réel.
— Les tailles et positions des panneaux doivent respecter les contraintes d'affichage pour éviter un chevauchement des éléments.

Utilisation par les autres fonctionnalités

Les panneaux de contrôle servent de point d'affichage central et de rétroaction pour le joueur. Ils communiquent avec :

— Les contrôleurs du terrain qui déclenchent l'affichage de détails lors d'un clic (par exemple, la sélection d'un Nid ou d'une Ressource).
— Les gestionnaires de score et d'énergie qui mettent à jour dynamiquement les valeurs affichées.

5.3 Option de pause et gestion du chronomètre

Structures de données utilisées

— `Timer` Swing : utilisé pour assurer une actualisation régulière du chronomètre et, dans certains cas, pour gérer des transitions visuelles (comme le mode nuit dans `TerrainPanel`).
— Une variable d'état dans la fenêtre de jeu (`JeuFrame`) : cette variable indique si le jeu est en pause ou non, afin que l'interface s'adapte (affichage d'un écran de pause, désactivation temporaire de certains contrôles, etc.).
— Un ensemble de threads regroupés dans la classe `ThreadSet` : il permet de contrôler centralement la mise en pause et la reprise des processus qui animent le jeu (gestion de l'énergie, déplacements, etc.).
— Une classe abstraite `InterruptibleThread` : chaque thread héritant de cette classe dispose d'un flag `isRunning`, qui détermine s'il continue ou non son exécution.

Constantes du modèle

— Intervalle de mise à jour du chronomètre : par exemple, 1000 ms, tel que défini dans le module chargé de l'affichage du temps (ou dans un éventuel `PanneauDeTableauDeBord`).
— Dans `TerrainPanel`, le `Timer` est également utilisé pour la transition fluide du mode nuit, avec une incrémentation définie par l'intervalle et un pas de variation de l'opacité.

Algorithme abstrait

L'algorithme de la gestion du chronomètre et de la pause s'articule autour des actions de l'utilisateur et de la propagation de l'état aux différents composants du jeu :

1. Au lancement du jeu, le chronomètre démarre et est mis à jour régulièrement grâce à un `Timer` Swing.
2. Lorsqu'une action de pause est déclenchée (par exemple, par le biais d'un bouton dans l'interface), le `PauseController` intercepte cet événement.
3. Le `PauseController` appelle la méthode `playPauseAll()` de la classe `ThreadSet`. Cette méthode parcourt l'ensemble des threads du jeu et, pour chacun, inverse l'état du flag `isRunning` :
   — Si le thread est actif, la méthode `pause()` est appelée (le flag passe à `false`), suspendant ainsi son exécution.
   — Si le thread est déjà en pause, la méthode `play()` est appelée pour le reprendre (le flag repasse à `true`).
4. Parallèlement, la fenêtre de jeu (`JeuFrame`) met à jour son état via ses méthodes `pause()` ou `resume()`, affichant éventuellement une interface de pause et ajustant la gestion du chronomètre pour conserver la valeur actuelle sans réinitialisation.
5. Lors de la reprise, le `Timer` redémarre, le chronomètre se poursuit, et l'ensemble des threads reprend son exécution normale.

Conditions limites à respecter

— Le `Timer` doit pouvoir être arrêté puis redémarré sans perdre la valeur actuelle du chronomètre.
— La mise en pause ne doit pas interférer avec d'autres processus critiques, notamment ceux assurant la mise à jour graphique (comme la transition en mode nuit dans `TerrainPanel`).
— Chaque thread doit vérifier régulièrement l'état du flag `isRunning` pour garantir une suspension réactive et éviter tout décalage dans la synchronisation.

Utilisation par les autres fonctionnalités

La gestion combinée de la pause et du chronomètre impacte plusieurs modules du jeu :

— La suspension temporaire du chronomètre permet au joueur de reprendre la partie sans perte de temps, garantissant la cohérence du suivi de la durée de jeu.
— Les modules de gestion de l'énergie, du déplacement, et d'autres mécanismes critiques de la simulation se voient suspendus grâce à l'inversion de l'état des threads dans `ThreadSet`.
— Les composants d'affichage, tels que le `RedessinJeuFrame`, doivent interroger l'état de pause afin de suspendre ou d'ajuster leurs mises à jour pendant la pause.

Détails de conception de l'option pause

L'architecture de l'option pause repose sur une séparation claire entre la gestion des événements d'interface et le contrôle des processus de fond :

— **PauseController** : ce contrôleur se charge d'écouter les actions de l'utilisateur sur les boutons « Pause » et « Reprendre ». À chaque action, il déclenche `threadSet.playPauseAll()` pour basculer l'état des threads, puis appelle soit `gameFrame.pause()` soit `gameFrame.resume()` pour mettre à jour l'interface graphique.
— **ThreadSet** : ce composant regroupe l'ensemble des threads qui orchestrent la simulation (énergie, déplacement, etc.). Sa méthode `playPauseAll()` parcourt tous les threads et inverse leur état, assurant ainsi une gestion homogène de la pause dans tout le moteur de jeu.
— **InterruptibleThread** : chaque thread dérivé de cette classe utilise le flag `isRunning` pour déterminer s'il doit poursuivre ou suspendre son exécution. Ce mécanisme simplifié garantit une interaction cohérente entre la logique métier et les commandes de pause.

5.4 Sauvegarde du score et du meilleur record

Structures de données utilisées

— `Score` : classe qui encapsule la valeur du score courant et gère sa modification (`augmenterScore`, `diminuerScore`, etc.).
— Un module de sauvegarde (potentiellement une classe `Sauvegarde`) qui réalise les opérations d'entrée/sortie sur un fichier.

Constantes du modèle

— Chemin du fichier de sauvegarde (ex : `cheminFichier = "scores.dat"`).
— Valeurs seuils pour l'enregistrement des records.

Algorithme abstrait

1. Pendant ou à la fin de la partie, le score courant est récupéré depuis l'objet `Score`.
2. Un module de sauvegarde est appelé pour écrire cette valeur dans un fichier, éventuellement en comparant avec le record existant.
3. Lors du lancement du jeu, le module de sauvegarde lit le fichier pour récupérer et afficher le meilleur record.

5.5 Les Fourmis : Caractéristiques, Déplacements, Animation et Mort

Structures de données utilisées

— **Classe `Fourmi`** (voir les sources) :
  — Attributs : position (`x, y`), énergie (`energie`) avec une valeur maximale (`MAX_ENERGIE` = 100), et un identifiant unique.
— **Classes de déplacement** :
  — `Deplacement` (voir les sources) et ses sous-classes (ex. `DeplacementSimple`) qui gèrent l'animation du déplacement de la fourmi d'un point de départ à un point d'arrivée.
— **Composants d'animation** :
  — Système d'animation via `SpriteAnimation` pour gérer la direction et le rafraîchissement des images associées à la fourmi.
  — Composants d'affichage tels que `CardFourmis` pour représenter graphiquement l'état d'une fourmi.

Constantes du modèle

— `Fourmi.MAX_ENERGIE` = 100.
— `Deplacement.VITESSE` = 5, représentant le pas de déplacement.

Algorithme abstrait (avec accent sur les déplacements)

1. **Initialisation :** Une fourmi est instanciée avec une position initiale et une énergie maximale.
2. **Déclenchement du déplacement :** Lorsqu'une fourmi doit se déplacer (ex. pour aller chercher une ressource ou retourner au Nid), le contrôleur crée un objet de type `DeplacementSimple`.
3. **Calcul de la trajectoire :**
   — La position actuelle de la fourmi est comparée à la destination.
   — La méthode `updatePosition()` ajuste (en incrémentant ou décrémentant) la coordonnée `x` ou `y` de la valeur `VITESSE` jusqu'à atteindre la destination.
   — La méthode `updateDirection()` ajuste la direction de l'animation (droite, gauche, haut ou bas) en fonction du déplacement.
4. **Animation et mise à jour graphique :** Le système d'animation `SpriteAnimation` est mis à jour continuellement pour refléter le mouvement, et l'affichage se rafraîchit via un thread de redessin (ex. dans `RedessinJeuFrame`).
5. **Gestion de la mort :** Si l'énergie atteint zéro (mise à jour par `GestionnaireEnergie` via `decrEnergie()`) ou la fourmi entre dans le champs de vision du prédateur, la méthode `isDead()` renvoie vrai et la fourmi est retirée de la simulation (gestion effectuée par `GestionnaireFourmisMortes`).

Conditions limites à respecter

— La fourmi ne doit pas dépasser la destination (les coordonnées sont plafonnées dans `updatePosition()`).
— Le déplacement doit être interrompu si l'énergie tombe à 0 avant l'arrivée.

— Les collisions ou restrictions (ex. déplacement vers un abri plein ou un Nid depuis un Nid) doivent être vérifiées par `DestinationSelectionnee`.

Interactions avec les autres fonctionnalités

— **Gestion du score :** Les fourmis transportant des ressources contribuent à l'augmentation du score du Nid à leur retour.
— **Interface :** L'information sur l'énergie et le déplacement est affichée via `CardFourmis` dans le panneau de contrôle.
— **Synchronisation :** Les déplacements sont synchronisés avec le redessin de la fenêtre (ex. via `TerrainController` et le thread de redessin).

5.6 Le Crapaud : Déplacements, Animation, Caractéristiques

Structures de données utilisées

— **Classe `Crapaud` :**
  — Attributs : position (`x, y`), vitesse (`dx, dy`), plage de vision (`visionRange`), état de satiété, et indicateur d'endormissement.
— **Animation :**
  — Composant d'animation via `CrapaudIdleAnimation` pour actualiser l'image affichée.

Constantes du modèle

— `MAX_SATIETE` = 10, `SATIETE_AU_REVEIL` = 5, et `SEUIL_FAIM` = 3.
— `DUREE_SIESTE` = 10000 ms.

Algorithme abstrait

1. **Mouvement aléatoire :** À chaque mise à jour (`update()`), le crapaud ajuste sa position en ajoutant `dx` et `dy`.
2. **Changement de direction :** Une modification aléatoire de direction intervient occasionnellement via `randomizeDirection()`.
3. **Vérification des limites du terrain :** Les coordonnées sont vérifiées et ajustées pour maintenir le crapaud dans le champ du terrain.
4. **Interaction avec les objets :** Le crapaud détecte les collisions avec ressources, abris ou Nid, et modifie son comportement (ex. saute pour éviter un abri).
5. **Gestion de la satiété et de l'état :** `update()` vérifie l'état de satiété. Si insuffisante, le crapaud consomme une ressource ou une fourmi, et si la satiété dépasse `MAX_SATIETE`, il passe en mode sommeil via `fallAsleep()`.

Conditions limites à respecter

— Le crapaud ne doit pas quitter les limites définies par `Terrain.LARGEUR` et `Terrain.HAUTEUR`.
— La transition entre états (éveil, faim, sommeil) doit être fluide avec un recalibrage correct de la satiété.

Interactions avec les autres fonctionnalités

— **Interférence avec fourmis et ressources :** Le crapaud peut intercepter les fourmis en déplacement ou consommer des ressources, ce qui influe sur le score et le transport.
— **Mise à jour graphique :** L'animation et les déplacements sont affichés dans le `TerrainPanel`.

5.7 Le Nid et les Abris

Structures de données utilisées

— **Classe `Nid` :**
  — Stocke le score accumulé, la liste des fourmis et fournit une méthode pour ajouter une nouvelle fourmi.
— **Classe `Abri` :**
  — Définit une capacité maximale servant à stocker un nombre limité de fourmis.

Constantes du modèle

— Capacité d'un abri définie lors de sa création.
— Règles de modification de l'énergie des fourmis lorsqu'elles se trouvent dans le Nid ou l'Abri.

Algorithme abstrait

1. **Initialisation :** À la création du `Terrain`, le `Nid` est positionné aléatoirement, et les abris sont ajoutés.
2. **Ajout de fourmis :** Le `Nid` fournit une méthode `ajouterFourmi()` qui ajoute une fourmi dès que le score est suffisant.
3. **Gestion de la capacité des abris :** Chaque abri vérifie via `isPlein()` si la capacité est atteinte.

Conditions limites à respecter

— La capacité des abris ne doit pas être dépassée.
— Les modifications d'énergie (augmentation d'énergie dans le Nid) doivent rester cohérentes.

Interactions avec les autres fonctionnalités

— Les fourmis retournant au Nid contribuent à l'augmentation du score et bénéficient d'une recharge énergétique.
— Les abris restreignent l'accès, influant sur l'organisation de la colonie.

5.8 Les Ressources : Animation de pousse, Génération initiale et en cours de partie

Structures de données utilisées

— **Classe `Ressource` :**
  — Attributs : poids, valeur nutritive, et indicateur de déplacement.

— **Gestionnaire de Ressources** (`GestionnaireRessources`) :
  — Thread qui surveille et ajoute des ressources pendant la partie.
— **Modules d'animation de pousse** :
  — Gèrent l'animation de croissance des ressources (de la graine à la ressource mature).

Constantes du modèle

— `Terrain.NB_RESSOURCES_MAX` = 40.
— Valeurs prédéfinies pour le poids et la valeur nutritive (définies dans des tableaux dans `GenerationRessource`).

Algorithme abstrait

1. **Génération initiale :** À l'initialisation du `Terrain`, la méthode `ajouterRessources(nb)` est appelée pour créer un nombre de ressources déterminé par la difficulté.
2. **Génération en cours de partie :** Le `GestionnaireRessources` exécute une boucle qui, à intervalle régulier (par exemple, toutes les 2 secondes), vérifie le nombre de ressources existantes et ajoute de nouvelles ressources ou ressources temporaires selon une probabilité.
3. **Animation de pousse :** Une fois générées, les ressources passent par une phase d'animation (de la graine à la ressource mature) avant d'être considérées comme exploitables.
4. **Transport de la ressource** :
   — **Deplacement** est une classe abstraite regroupant les fonctionnalités de base permettant à une entité de se déplacer sur le terrain (via `updatePosition()` et `updateDirection()`). Elle définit également un attribut statique `VITESSE` qui module le pas de mouvement.
   — **DeplacementSimple** est utilisé pour les déplacements classiques des fourmis. Une instance de cette classe est créée lorsqu'une fourmi doit se déplacer d'un point A à un point B, et elle conserve une référence vers la fourmi concernée.
   — **DeplacementRessource** est dédié aux déplacements liés au transport de ressources. Il utilise la même logique de déplacement que `Deplacement` mais ajoute la gestion spécifique au transport d'une `Ressource`, afin que celle-ci soit acheminée vers le `Nid` et le score mis à jour.
   — Les classes **Fourmi** et **Ressource** contiennent les données propres à ces entités et interagissent avec les classes de déplacement pour orchestrer les mouvements sur le terrain.

   Ce diagramme illustre ainsi clairement la relation hiérarchique entre les classes de déplacement et montre comment elles sont utilisées pour gérer à la fois le mouvement des fourmis et le transport des ressources, assurant une cohérence dans l'animation et la mise à jour du modèle de jeu.

Conditions limites à respecter

— Le nombre total de ressources ne doit pas excéder $\texttt{NB}_RESSOURCES_MAX.Lesressourcestemporaire$
— Les ressources doivent être placées sans chevauchement excessif avec d'autres objets sur le terrain.

Interactions avec les autres fonctionnalités

— Une fois prêtes, les ressources déclenchent le transport par les fourmis vers le Nid, ce qui influence le score.

— Le module de génération en cours de partie s'intègre à la logique globale du Terrain, renouvelant les ressources disponibles durant la simulation.

5.9 Fonctionnalités Complémentaires

Objectif : Apporter un enrichissement esthétique et immersif au jeu.

Histoire et Narration

— Définition d'un scénario introductif qui présente l'univers du jeu.
— Affichage sous forme de séquence ou d'écran dédié au lancement.

Effets Sonores et Musique de Fond

— Utilisation de la classe `MusicPlayer` pour la gestion des musiques et des effets sonores.
— Changement dynamique de l'ambiance sonore (ex. lors du passage du mode jour à nuit).

Mode Jour/Nuit

— Implémenté dans `JeuFrame` et `TerrainPanel` via l'ajustement de l'opacité de l'affichage.
— Synchronisation avec les effets sonores (transition des bruits de jour et de nuit).

— **Difficulté :** Moyenne
— **Priorité :** Haute (pour l'immersion)

Pluie

— Mise en œuvre d'un overlay graphique ou d'un système de particules simple pour simuler la pluie.
— Fonctionnalité optionnelle pour la version initiale.

— **Difficulté :** Faible à Moyenne
— **Priorité :** Faible

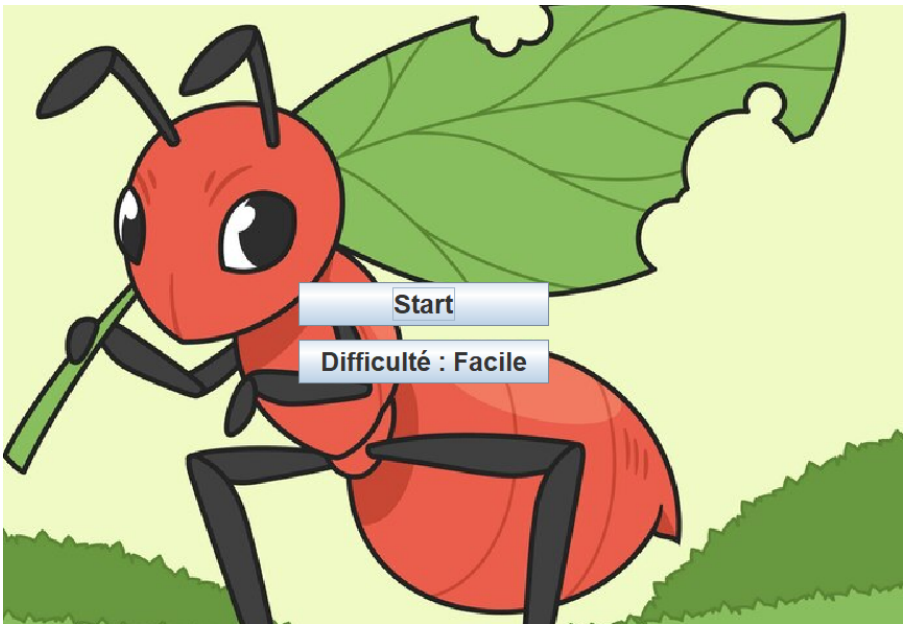
La conception détaillée présentée ci-dessus permet de comprendre comment chaque fonctionnalité du projet *Ant Colony Rush* a été pensée et réalisée. L'approche en blocs fonctionnels montre l'architecture globale du système, tandis que les sous-sections dédiées offrent un éclairage précis sur les structures de données, les algorithmes abstraits et les conditions limites nécessaires pour garantir une simulation robuste et immersive. Cette documentation constitue une base solide pour la validation, la maintenance et l'évolution future du projet.

6 Résultats

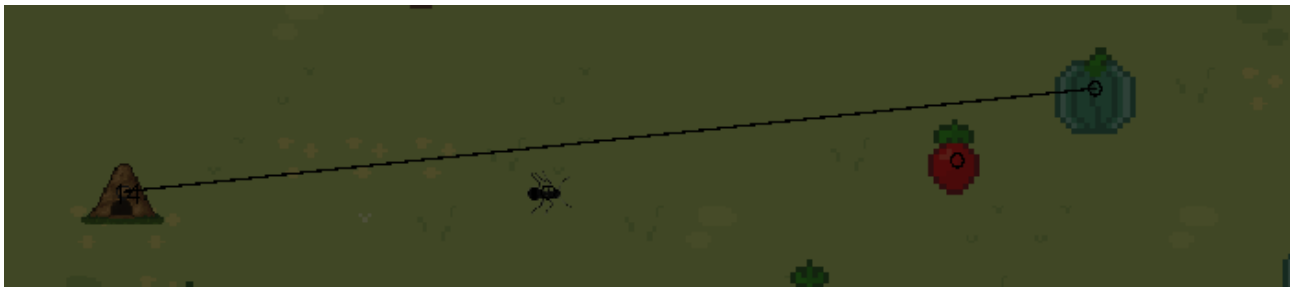
La version (presque) finale du jeu présente une interface graphique dynamique et immersive, intégrant plusieurs modules essentiels qui se combinent pour offrir une expérience de simulation riche et interactive. L'interface principale se compose notamment d'un panneau de contrôle complet, d'un menu de démarrage intuitif et d'une simulation nocturne où l'on peut observer le déplacement d'une fourmi, ainsi que la présence du crapaud et l'affichage de son champ de vision.

6.1 Panneau de contrôle

6.2 Menu de démarrage



6.3 Simulation nocturne et déplacement d'une fourmi



6.4 Crapaud et son champ de vision

La mise en œuvre de ces composants repose sur une architecture logicielle robuste, garantissant une synchronisation efficace entre l'interface utilisateur et les processus en arrière-plan, notamment via l'utilisation des threads contrôlés par `ThreadSet` et la gestion de la pause par `PauseController`. Les effets visuels, tels que la transition en mode nuit et l'animation dynamique des entités, renforcent l'immersion et la qualité de la simulation proposée par le jeu.

7 Conclusion et perspectives

Dans le cadre de ce projet, nous avons réalisé une simulation interactive d'une colonie de fourmis en temps réel. Nous avons mis en place une architecture robuste basée sur le modèle MVC, combinée à une gestion concurrente via des threads dédiés. Notre prototype permet de gérer les déplacements des fourmis, l'interaction avec le crapaud, la collecte et le transport des ressources, ainsi que la mise en pause et la gestion du score.

Les principales difficultés rencontrées ont été le travail de groupe, la gestion du temps et la résolution de bugs, en particulier ceux liés à la programmation concurrente. Pour y remédier, nous avons adopté une méthode de travail collaborative efficace (outils de gestion de version, réunions régulières et communication continue) et développé des mécanismes de synchronisation comme la classe abstraite `InterruptibleThread` et un `ThreadSet` centralisé. Ces solutions nous ont permis de stabiliser l'exécution simultanée des différents processus et d'assurer un fonctionnement fluide du jeu.

Ce projet nous a permis d'acquérir une meilleure maîtrise du travail en groupe, de la communication et de la gestion de projet. Nous avons également renforcé nos compétences en programmation concurrente ainsi qu'en développement d'interfaces interactives, des domaines essentiels pour la création de systèmes réactifs et performants.

En perspective, plusieurs axes d'amélioration majeurs sont envisageables :

- Une refonte graphique complète afin d'améliorer l'aspect visuel du jeu, avec des animations plus fluides et des effets visuels attractifs.
- L'enrichissement des interactions avec le joueur par l'ajout de nouvelles mécaniques de jeu et de scénarios interactifs.
- L'optimisation de la gestion concurrente pour fluidifier davantage les transitions et les mises à jour en temps réel.

Ces évolutions permettront de rendre le jeu encore plus immersif et de faciliter l'extension des fonctionnalités existantes.