Tower Defense

& MACHINE LEARNING

Tuteur : THOMAS Vincent

Présentation de notre étude préalable du sujet

Sommaire

01	Principe du jeu	06	Point de passage
02	Et dans notre cas	07	Steering Behaviours
03	Moteur de jeu	08	Algorithmes gérant l'évolution des ennemis
04	Déplacement des ennemis	09	Choix de chemin
05	Machine de Braitenberg	10	Différentes itérations

Principe du jeu

UN TOWER DEFENSE, C’EST QUOI?

- Type de jeu vidéo
- Principe : Défendre une zone
- Difficulté : Vagues successives d’ennemis
- Objectif : Placer des défenses au fur et à mesure pour défendre la zone voulue des ennemis.

Et dans notre cas ?

- Se placer du côté des ennemis.
- Les placements des défenses seront prédéfinis
- Se concentrer sur la manière dont il vont réagir, se déplacer, attaquer
- Voir l'évolution de leur comportement d'une manche à l'autre (après échec)
- Utiliser un algorithme d'apprentissage (machine learning)

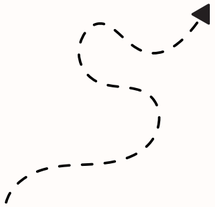
> On utilise alors l'apprentissage par IA

Avoir des ennemis qui s'améliorent de manche en manche	On s'appuie sur les propriétés des ennemis/défenses (points de vie, vitesse de déplacement, d'attaque...)	Définir un ou plusieurs algorithme(s) d'apprentissage parmis ceux étudiés (steering behaviours, points de passage, machines de Braitenberg..) On choisit le Java pour programmer notre jeu
		A tester/comparer/combiner

> Sur quelles caractéristiques se baser?

Propriétés classiques (points de vie, vitesse de déplacement, dégats...)

Principe de type “style Pokémon” avec contre-types



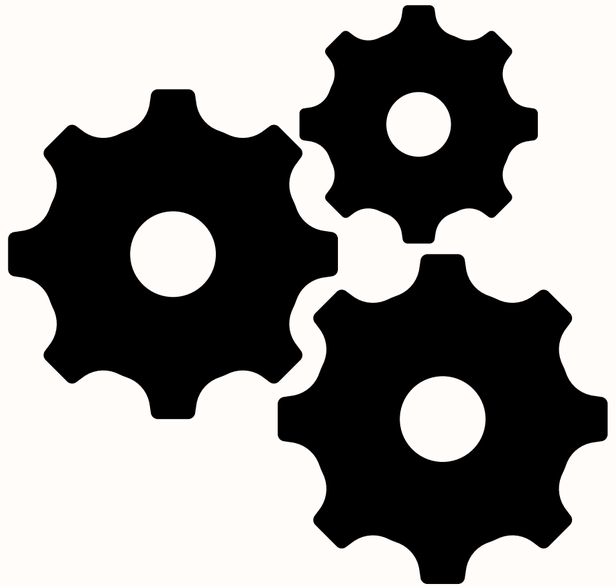
Forme de trajectoire des ennemis
Quelles sont les tendances?

La manière d’attaquer en groupe.
Plusieurs types en une attaque?

Le placement des défenses

Moteur du jeu

- Slick2D
- JavaFX
- Swing



Slick2D

Avantages

- Moteur de jeu dédié, conçu pour les animations et le rendu rapide
- Performant et capable de gérer des jeux avec de nombreux sprites, des effets de particules, et des animations fluides
- Intégration facile de sons et de musique, gestion des contrôles avancés (clavier, souris, manette)
- Supporte les feuilles de sprites et les animations avancées, utiles pour les jeux de type arcade ou plateforme

Inconvénients

- Plus complexe à utiliser que JavaFX et Swing, car il s'agit d'une bibliothèque tierce
- Le développement de Slick2D est arrêté, bien que la bibliothèque reste fonctionnelle
- Documentation et support communautaire plus limités par rapport Swing et JavaFX

Utilisation : Jeux 2D nécessitant des performances élevées et des animations fluides

JavaFX

Avantages

- Bibliothèque connu et déjà utilisé pour des projets comme “Zeldiablo” ou “Trello”
- Supporte les animations et effets graphiques (par exemple les effets de transition ou les rotations).
- Plus performant que Swing pour les jeux simples
- Facile à utiliser pour des jeux 2D simples avec des animations et des éléments visuels plus interactifs.

Inconvénients

- Moins performant que Slick2D pour les jeux nécessitant un haut taux de rafraîchissement.
- Légèrement plus complexe que Swing
- Bien qu'il soit meilleur que Swing pour les animations, il reste limité pour les jeux plus complexes.

Utilisation : Jeux 2D simples avec animations et effets graphiques légers

Swing

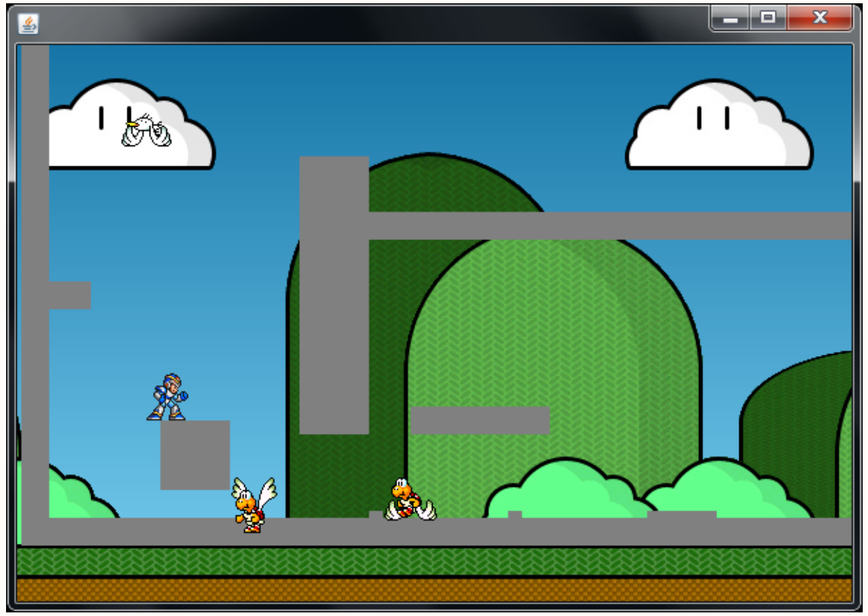
Avantages

- Simplicité et large documentation.
- Parfait pour des interfaces graphiques statiques ou des jeux de type puzzle (par exemple, un jeu de Sudoku, Tic-Tac-Toe).
- Intégration directe avec Java sans nécessiter de bibliothèque tierce.

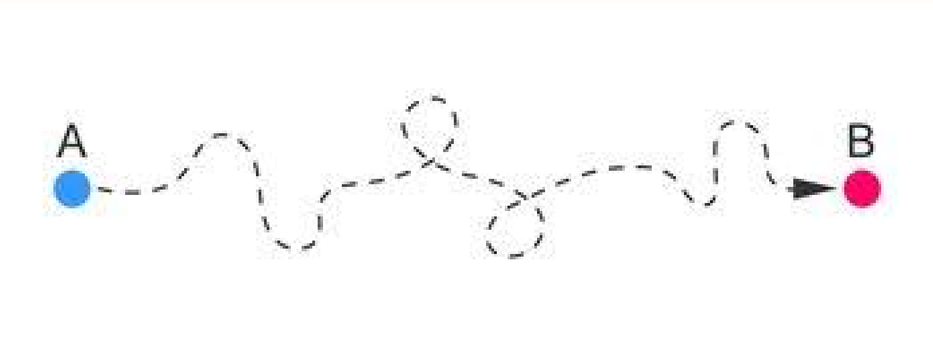
Inconvénients

- Performances limitées pour les jeux avec beaucoup de sprites ou d'animations.
- Pas optimisé pour le rendu graphique accéléré.
- Manque de support pour les effets graphiques modernes (par exemple, des shaders ou des effets de particules).

Utilisation : Jeux simples ou prototypes rapides où les performances ne sont pas une priorité.

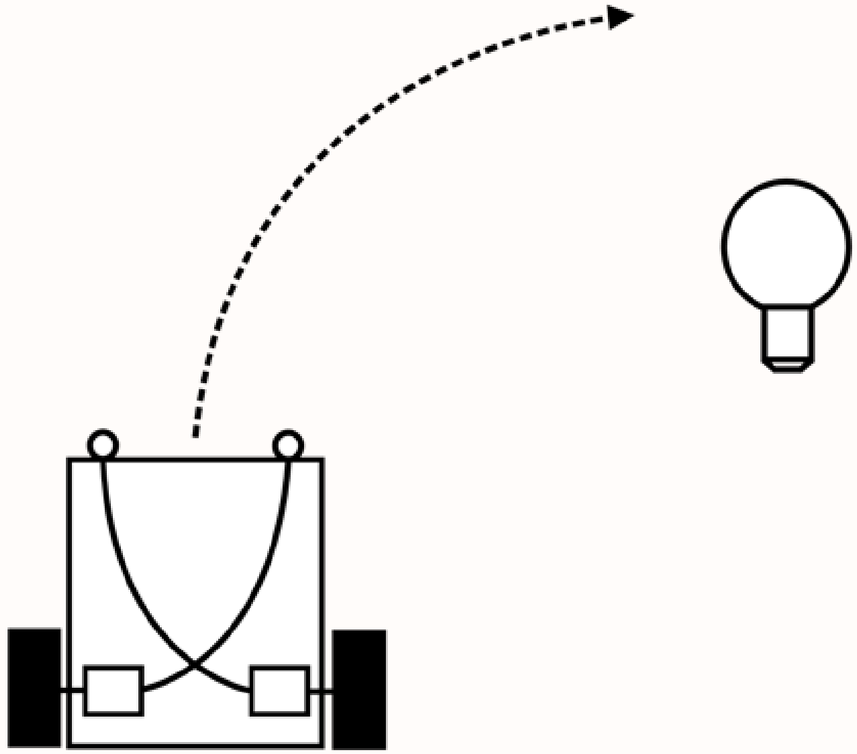


Déplacement des ennemis



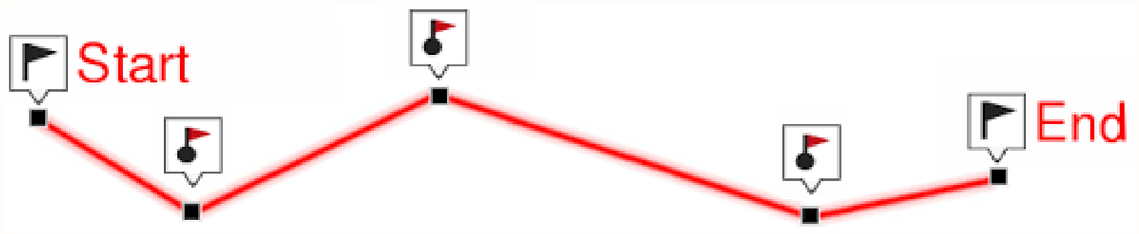
- Machines de Braitenberg
- Points de passage
- Steering Behaviours

Machines de Braitenberg



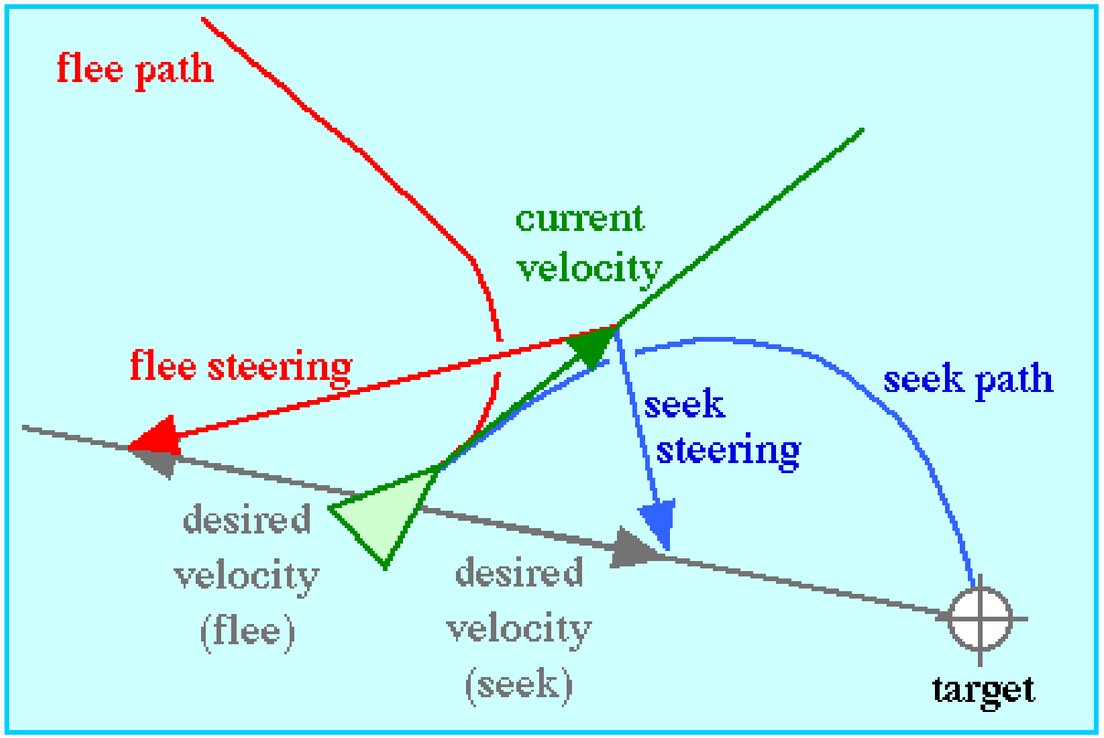
- Permet de simuler un comportement “intelligent” très simplement

Points de passage



- Défini un chemin en séquence de positions
- Très simple à paramétrer

Steering Behaviours



- Permet de simuler des mouvement réalistes
- La vitesse varie grâce à une accélération

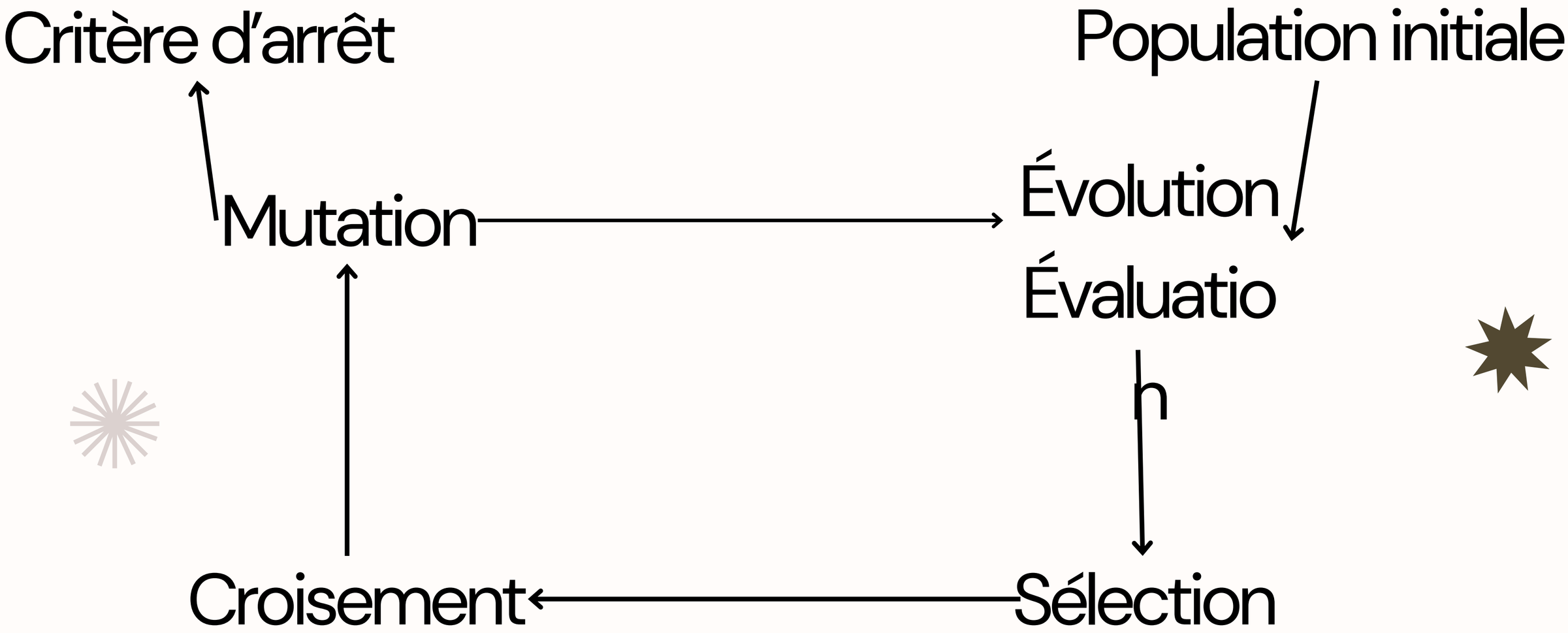
Choix de chemin

- Bellman Ford
- Dijkstra
- A*

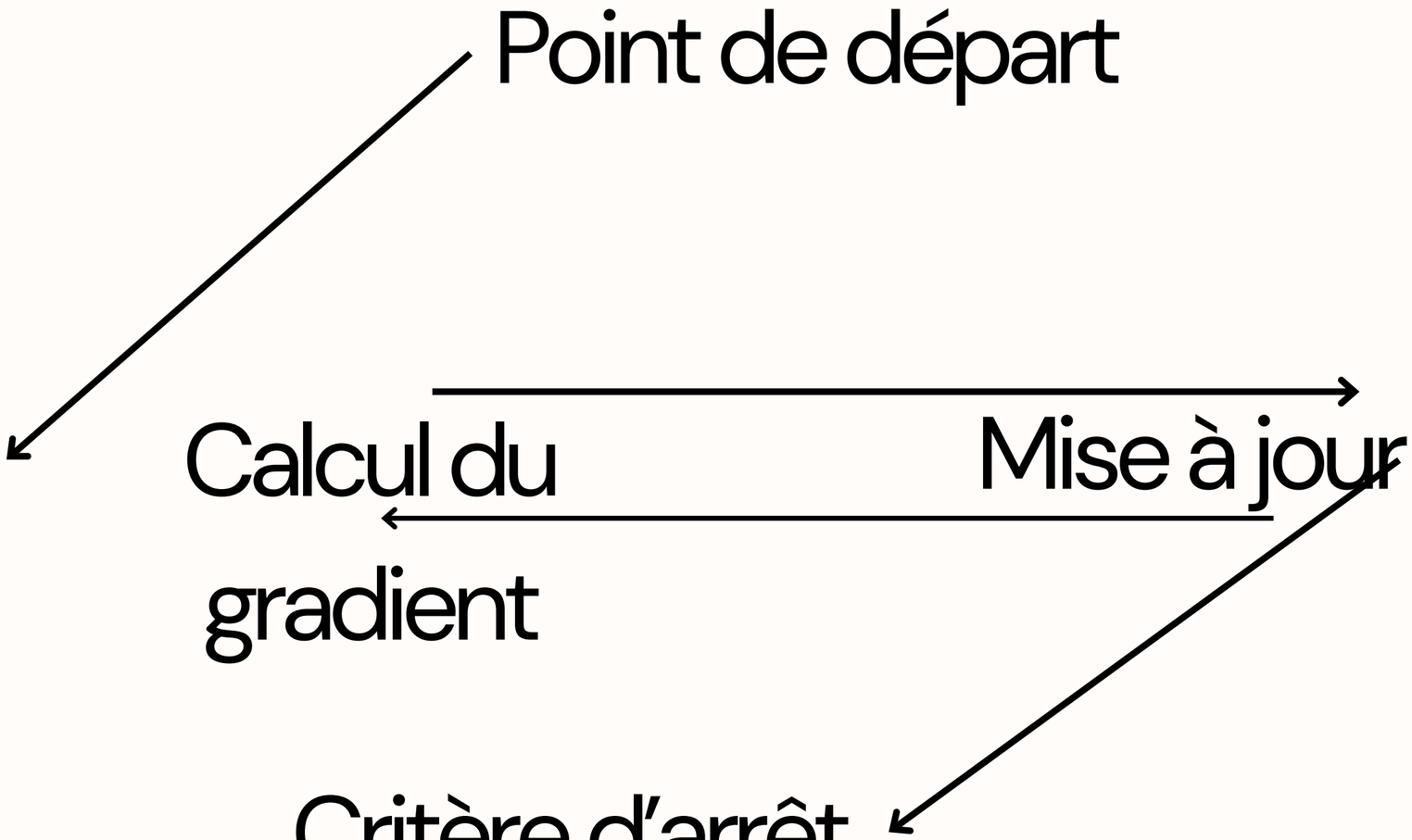
Algorithmes gérant l'évolution des ennemis

- Algorithme évolutif
- Descente de gradient

Algorithme évolutif



Descente de gradient



Nos choix

- Moteur de jeu : **JavaFX**
- Algorithme de déplacement : **point de passage** (pour commencer)
- Algorithme d’évolution : **Descente de gradient**
- Algorithme de choix de chemins : **Dijkstra** (pour commencer)

Itération 1

- Choix, compréhension et implémentation du moteur de jeu.
- Définition des éléments du jeu (défenses, ennemis, labyrinthe).

- **Produit Minimum Viable :** Moteur de jeu opérationnel (une fenêtre s’ouvre)

Itération 2

- Implémentation du déplacement des ennemis.
- Créations de types d’ennemis, des types d’attaques pour préparer l’implémentation d’un algorithme d’évolution.
- **Produit Minimum Viable :** Génération d’une unité statique

Itération 3

- Ajout d’algorithme de choix de chemins.
- Implémentation d’un premier algorithme d’évolution des ennemis.

- **Produit Minimum Viable :** Génération d’une unité et déplacement de cette dernière.

Itération 4

- Finalisation de la première version avec un algorithme de déplacement, de choix de chemin et d’évolution fonctionnelle.
- Préparation de la soutenance

- **Produit Minimum Viable :** Déplacement et évolution fonctionnels)

Itération 5

- Ajout d’un algorithme de déplacement plus sophistiqué

- **Produit Minimum Viable :** Déplacement avancé des ennemis

Itération 6

- Implémentation de l’algorithme d’évolution sur les défenses

- **Produit Minimum Viable :** Défenses adaptatives

Itération 7

- Préparation de la soutenance finale et du document final

- **Produit Minimum Viable :** Version finale avec toutes les fonctionnalités

Merci de nous avoir écouté