# Projet commun POO – Conduite de Projet

Jeu d'artillerie 2D (type Worms)

Le but de ce projet est d'implémenter en Java un jeu *Worms*-like : des équipes de *vers* s'affrontent sur un terrain 2D destructible, à tour de rôle, en utilisant un petit arsenal soumis à une physique simple et à des collisions. À titre d'illustration (et sans obligation de reproduire ni le graphisme, ni toutes les fonctionnalités), vous pouvez jeter un œil à une version jouable en ligne du jeu original [1]. Les sections suivantes détaillent les contraintes que votre implémentation devra respecter.

## 1 Généralités

- Le projet doit être réalisé en groupes de 4 à 6 personnes.
- Les soutenances auront lieu la semaine du 12 janvier 2025 ; votre dernière *release* devra être effectuée au plus tard le **5 janvier 2026 à 22h**.
- Les consignes pratiques relatives à la soutenance seront communiquées dans l'espace Moodle du cours « Conduite de projet ».

## 2 Description générale du jeu

On considère un monde virtuel en 2D, vertical, contenant dans sa partie inférieure un plan d'eau (l'océan), surmonté d'une bande de terrain destructible (le sol), elle-même surmontée d'un espace vide (le ciel). Deux équipes de *vers* s'y déplacent ; chaque ver peut atteindre des positions avantageuses, sauter et tirer sur les vers de l'équipe adverse.

Chaque équipe dispose d'un petit *arsenal* (par ex. bazooka, grenade) et d'un *inventaire* indiquant les munitions restantes. Les tirs obéissent à un *modèle physique simplifié* : un projectile est lancé avec une vitesse initiale (déterminée par l'angle de tir et la puissance de l'arme), subit une accélération verticale constante due à la gravité et, si l'option est activée, une accélération horizontale constante modélisant le vent. Le mouvement est arrêté au premier contact avec le terrain ou l'eau. À l'impact (ou après une temporisation configurable selon l'arme), le projectile déclenche son effet (par ex. explosion), inflige des dégâts aux vers dans son rayon d'action et peut optionnellement creuser le terrain [2].

Le jeu se déroule au tour par tour. Chaque tour est limité par une durée (jauge de temps affichée). Par défaut, un seul ver par équipe joue à chaque tour : le ver actif peut se déplacer, viser et utiliser une arme jusqu'à expiration du temps. Les règles déterminant le ver actif sont libres : rotation fixe (les vers d'une équipe jouent successivement tour après

1. Worms (version classique en ligne)
2. Recommandation : la modification du terrain peut être traitée dans un second temps (extension), après une première version jouable sans déformation.

tour, toujours dans le même ordre), sélection manuelle à la souris, ou tout autre mécanisme clair pour le joueur. En extension, vous pouvez permettre à une équipe d'activer plusieurs vers au cours d'un même tour, tant que la jauge de temps d'équipe n'est pas épuisée. À l'expiration du temps, le tour s'achève. Avant de passer au tour suivant, les effets des actions en cours sont menés à leur terme : par exemple le déplacement d'un projectile, puis son explosion, la destruction éventuelle d'une portion de terrain (si l'extension est implémentée) et la chute d'un ou de plusieurs vers, etc.

Les points de vie (PVs) diminuent à la suite des explosions ou des chutes. Un ver est éliminé à 0 PV. S'il tombe dans l'eau ou hors de la carte, il est éliminé immédiatement (équivalent à PVs réduits à 0). La victoire est obtenue lorsque tous les vers adverses sont éliminés.

Nous vous laissons le choix des paramètres de partie (taille de carte, nombre de vers, puissance des armes, durée du tour, etc.). De même, les aspects purement esthétiques (graphismes, animations) sont libres et ne seront pas évalués, vous avez le droit d'utiliser des graphismes disponibles en ligne[3]. Vous pouvez également, si vous le souhaitez, utiliser des bibliothèques (par exemple de gestion de la physique ou toute autre bibliothèque utile) ; les bibliothèques employées doivent être mentionnées dans le rapport (nom, version et lien). En revanche, votre implémentation du jeu en Java devra respecter les contraintes décrites dans les sections qui suivent.

## 3 Implémentation du jeu en Java

Votre projet devra présenter à l'utilisateur une interface graphique réalisée à l'aide des bibliothèques `awt` et `swing` *uniquement* (pas de `javafx`). L'usage du clavier est limité aux *touches directionnelles* (déplacements et saut). Toutes les autres actions (sélection d'arme, visée, réglage de puissance via clic prolongé, tir, interactions avec l'interface) doivent pouvoir se faire à la *souris*.

**Vue textuelle (obligatoire).** Afin d'encourager une bonne séparation *modèle / vue*, vous devrez fournir une version textuelle minimale (console) capable de : (i) lancer une partie, (ii) afficher l'état essentiel (équipe active, ver actif, points de vie, informations de tour), (iii) exécuter les actions de base (déplacements, tir, fin de tour). Cette vue sert au prototypage et aux tests précoces ; elle doit réutiliser le même modèle que la vue graphique.

Prévoyez un moyen simple de choisir la vue, par ex. un argument de ligne de commande ou une option dans l'écran d'accueil (voir section 3.1).

3. Par exemple : `https://www.spriters-resource.com/pc_computer/wormsgeddon/asset/13597/`.

### 3.1 L'écran d'accueil

Au lancement du programme, on présentera à l'utilisateur un écran d'accueil contenant quatre boutons permettant respectivement :

- de poursuivre une partie sauvegardée lorsque celle-ci existe (voir ci-dessous),
- de démarrer une nouvelle partie,
- d'accéder à une page de réglages (taille de carte, nombre de vers par équipe, durée d'un tour, etc.),
- de quitter l'application.

L'activation du premier ou du second bouton permettra à l'utilisateur d'accéder à l'écran de jeu décrit à la section suivante.

**Sauvegarde de la partie.** Il vous est demandé d'implémenter un système de sauvegarde locale dans des fichiers texte. Avant de quitter l'application (ou via un menu « Sauvegarder »), on proposera au joueur d'enregistrer l'état courant de sa partie dans un fichier `sauvegarde.txt`. Le format est libre, mais doit être purement textuel et lisible dans un éditeur ordinaire. Vous pouvez autoriser (ou non) la possibilité de quitter au milieu d'un tour. Si vous autorisez la sortie en cours de tour, la sauvegarde doit inclure la durée restante du tour (jauge de temps) et permettre une reprise exacte de cette durée au chargement. Sinon, la sauvegarde se fait en fin de tour.

Le fichier doit contenir au minimum : une représentation de la carte (eau, terre, vide), les paramètres de partie (durée restante du tour si elle est gérée, les états des vers (positions, points de vie), les inventaires des deux équipes (munitions restantes), les projectiles en cours le cas échéant, ainsi que l'information sur le prochain joueur à jouer (ordre des tours). Au lancement suivant, si `sauvegarde.txt` existe, un bouton « Poursuivre la partie » restaurera l'état spécifié par son contenu.

### 3.2 L'écran de jeu

L'écran de jeu affiche au minimum un terrain 2D destructible, un ciel, de l'eau en bas de la carte et des vers. L'affichage doit s'adapter à des dimensions raisonnables (au choix). Les tirs suivent une physique simple (gravité) et peuvent être influencés par un vent optionnel.

- Au lancement d'une partie, le terrain et l'eau (en bas de la carte) sont visibles, ainsi que les vers de chaque équipe.
- Les vers sont générés au début de la partie puis posés sur le terrain, sur des zones d'apparition choisies de manière à éviter l'eau immédiate et les positions impossibles (enterrées, sans appui). La méthode de sélection exacte est libre.
- Défilement de la carte : l'écran doit pouvoir se déplacer pour explorer la carte au-delà de la zone de départ. Le zoom est optionnel (à votre convenance).

## 3.3 L'inventaire

L'inventaire d'armes et d'outils est *commun à toute l'équipe*. Pendant son tour, un joueur peut consulter cet inventaire, choisir un objet et l'utiliser. Les munitions (ou équivalents) diminuent à l'usage ; certaines entrées peuvent être illimitées.

Vous choisissez quels objets existent dans votre version (armes, outils, objets posables simples). Ils doivent permettre d'illustrer concrètement l'utilisation d'un inventaire *d'équipe* et des mécanismes POO attendus ci-dessus.

### 3.3.1 Actions possibles

Se déplacer, sauter, viser, tirer, poser/activer un objet, passer son tour. Les effets (trajectories, dégâts, temporisations) sont libres, sous réserve d'une lecture simple en jeu. Le vent est optionnel : s'il est activé, il peut influencer certaines trajectoires.

## 3.4 Fonctionnalités supplémentaires possibles

Les idées suivantes (liste non-exhaustive) complètent l'implémentation minimale et chaque nouvel ajout sera valorisé dans la note finale. Elles doivent être activables/désactivables dans les options avant de lancer une partie. Les fonctionnalités actives courantes et les valeurs de leurs paramètres devront bien sûr être inscrites dans le fichier de sauvegarde, et récupérées au chargement.

- **Paramétrage de base.** Ajuster des paramètres simples tels que :
  - la *durée du tour* (temps dont dispose un joueur pour agir) ;
  - les *points de vie initiaux* (plus ou moins élevés) ;
  - les *munitions disponibles* au départ (plus ou moins nombreuses) ;
  - les *dégâts de chute* (tomber de haut blesse plus ou moins) ;
  - le *tir allié* (*friendly fire*) : les tirs peuvent ou non blesser les vers de sa propre équipe.
- **Activation du vent.** (oui/non), indication visuelle de sa direction et de son intensité courantes, et effet sur les projectiles.
- **Génération de cartes.** Offrir plusieurs « thèmes » de terrain (îlots séparés, grottes, ponts) et ajouter un *décor* purement visuel (ciel, nuages, éléments de fond).
- **Armes supplémentaires.**
  - **Ricochet.** Arme qui peut appliquer son effet à un autre ver proche.
  - **Projectile à retardement.** Explose après un compte à rebours.
  - **Mine.** Posée sur le sol, elle explose si l'on s'en approche.
  - **Tourelle.** Elle attaque les vers ennemis qui s'approchent pendant leur déplacement.

  - **Corps à corps.** S'utilise au contact et déplace le ver ciblé.
- **Objets utilitaires.**
  - **Poutrelle.** Une barre fixe placée sur la carte pour créer un pont ou un abri.
  - **Outils de déplacement.** Grappin, jet-pack...
  - **Outils de terrain.** Foreuse.
  - **Soins.**
- **Adversaires contrôlés par l'ordinateur (IA simple).** Un « bot » choisit une action basique : sélectionner une arme, viser approximativement, éviter de tomber à l'eau. L'objectif est d'avoir un partenaire ou un adversaire minimal pour tester le jeu en solo.
- **Progression entre manches (optionnel).** Entre deux parties ou manches, afficher un petit écran permettant de dépenser des *points* gagnés (par exemple) pour racheter des munitions, des outils ou des soins. Le barème et le nombre de points sont à votre choix.
- **Inventaire élaboré.**
  - **Caisse.** Apparaît périodiquement sur la carte et permet d'ajouter un objet à son inventaire.
  - **Fabrication.** Fusionne plusieurs objets de l'inventaire en un autre.
  - **Inventaire séparé par ver.**
- **Audio et effets visuels.** Ajouter des sons (tir, explosion, saut) et de petites particules visuelles (fumée, éclats), sans ralentir le jeu.

# 4 Rendu et évaluation

Le travail doit être réalisé tout au long de la suite du semestre via un projet `GitLab` pour chaque groupe. L'usage de ce gestionnaire de version permettra d'éviter toute perte de données, et d'accéder si besoin à vos anciennes versions.

- Votre dépôt `Git` doit être hébergé par le serveur `GitLab` de l'UFR d'informatique :

  `https://moule.informatique.univ-paris-diderot.fr`

  Aucun autre code ne sera pris en compte, même hébergé par un autre serveur.
- Votre dépôt doit être rendu **privé** dès sa création, avec accès uniquement aux membres du groupe. Vous ajouterez les enseignants de ce cours en qualité de rapporteur. Tout code laissé en accès libre sur Moule ou ailleurs sera considéré comme une incitation à la fraude, et sanctionné.
- Votre projet fera l'objet d'une évaluation continue pendant les trois TP de conduite de projet.

- A la fin du projet nous consulterons l'historique de votre dépôt, afin de vérifier la régularité de votre travail et l'équilibre dans le groupe. Un historique trop réduit pourra être pénalisant, et en cas de trop grand déséquilibre, les notes pourront être individualisées.
- Il va de soi que votre travail doit être strictement personnel : aucune communication de code ou d'idées entre les groupes, aucune "aide" externe ou entre groupes. Nous vous rappelons que la fraude à un projet est aussi une fraude à un examen, passible de sanctions disciplinaires pouvant aller jusqu'a l'exclusion définitive de toute université.

## 4.1 Rendu

Votre projet `GitLab` doit contenir :

- les sources et les ressources de votre programme (fichiers `.java`, images, etc.). Ne polluez pas votre dépôt avec des fichiers inutiles : utilisez un fichier `.gitignore` pour exclure les fichiers générés.
- un fichier nommé `README` qui indique comment on compile et on se sert de votre programme (compilation, exécution, utilisation). Faites en sorte que son utilisation soit la plus simple possible. Ne pensez pas que nous utilisons le même environnement de développement que celui que vous avez choisi.
- un rapport au format `PDF` d'au moins cinq pages **rédigées**, expliquant les parties du cahier des charges qui ont été traitées, les problèmes rencontrés et une brève chronologie indiquant comment et quand ils ont été résolus, les problèmes encore connus, et les pistes d'extensions que vous n'auriez pas encore implémentées. Il devra impérativement contenir une représentation graphique du modèle des classes (le plus simple est qu'elle soit manuscrite, puis scannée). Le rapport n'est en aucun cas une impression de votre code !
- un fichier de sauvegarde de votre jeu avancé (`.txt`) pour nous permettre de tester toutes les fonctionnalités du jeu.
- toute autre ressource utile pour rendre la soutenance fluide.

Le correcteur doit pouvoir tester votre travail [4] en simplement lisant votre fichier `README` pour y trouver la ligne de commande qui lance le programme.

4. Testez vous même votre rendu sur une autre machine ou pour le moins dans un dossier séparé de celui où vous avez travaillé. Si nous n'arrivons pas à exécuter votre code vous serez évidement fortement pénalisés.

# 5 Attendus spécifiques pour les étudiants qui suivent le cours de conduite de projet

Dans cette partie, nous détaillons les attentes spécifiques pour les étudiants inscrits à l'UE « Conduite de projet ». Tous les projets seront évalués sur la qualité du code et sur les fonctionnalités implémentées. Cependant, les étudiants qui suivent cette UE devront en plus respecter des consignes supplémentaires, qui seront évaluées dans le cadre de l'UE.

Les étudiants qui ne suivent pas cette UE ne sont pas tenus d'utiliser des outils spécifiques pour les tests, la CI/CD ou la gestion des tâches ; cela reste néanmoins conseillé, car il est difficile de produire un code de qualité sans une bonne organisation et sans tests.

Les étudiants qui suivent l'UE « Conduite de projet » mais qui ont déjà validé l'UE POO pourront concentrer leur travail sur la mise en place des outils de compilation, des tests, de la documentation et des scripts de CI/CD. Le développement fonctionnel du jeu ne leur sera pas demandé.

## 5.1 Méthodologie Agile

Vous devrez adopter une méthodologie de développement Agile, en particulier Scrum. Vous constituerez un backlog produit, que vous affinerez au fil du projet. Vous organiserez votre travail en sprints d'une durée d'environ deux semaines, avec une réunion de planification au début de chaque sprint et une revue de sprint à la fin. Les séances de TP de « Conduite de projet » seront utilisées pour ces réunions.

## 5.2 Gestion des tâches

Vous utiliserez l'outil de gestion des tâches de `GitLab` pour suivre l'avancement de votre projet. Vous créerez des issues pour chaque tâche à accomplir, que vous assignerez aux membres de l'équipe. Vous utiliserez des étiquettes pour catégoriser les tâches (par exemple : *bug*, *feature*, *documentation*, etc.). Vous utiliserez des *milestones* pour organiser les tâches par sprint. Vous utiliserez des *boards* pour visualiser l'état d'avancement des tâches (par exemple : *To Do*, *In Progress*, *Done*).

## 5.3 Workflow Git

Vous utiliserez des branches dédiées pour le développement de nouvelles fonctionnalités ou la correction de bugs. Chaque branche sera nommée de manière descriptive (par exemple : `feature/ajout-inventaire`, `bugfix/collision-sol`). Vous utiliserez des *merge requests* pour intégrer les modifications dans la branche principale (`main` ou `master`). Chaque *merge request* sera revue par au moins un autre membre de l'équipe avant d'être fusionnée. Seules les *merge requests* qui passent les tests automatiques seront fusionnées. Vous écrirez des messages de commit clairs et descriptifs. Les fonctionnalités, une fois testées et validées, seront fusionnées dans la branche principale à la fin de chaque sprint.

### 5.4 Documentation

Vous documenterez votre code en utilisant des commentaires Javadoc pour toutes les classes et méthodes publiques. Vous rédigerez un fichier `README.md` à la racine de votre dépôt, qui expliquera comment compiler, exécuter et utiliser votre programme. Vous tiendrez un journal de bord dans un fichier `CHANGELOG.md`, qui décrira les principales modifications apportées à chaque version de votre projet. Vous documenterez également les décisions de conception importantes dans un fichier `DESIGN.md`. Vous listerez les auteurs et leurs contributions dans un fichier `AUTHORS.md`. Vous adopterez un style de code cohérent, en suivant les conventions de codage Java (par exemple, nommage des classes, méthodes, variables, indentation, etc.). Optionnellement, vous pouvez utiliser un système de vérification de style de code (comme Checkstyle) pour automatiser la vérification de la conformité au style. Celui-ci pourra être intégré au pipeline CI/CD. Vous choisirez une licence appropriée pour votre projet (par exemple, MIT, Apache 2.0, GPLv3, etc.) et l'inclurez dans un fichier `LICENSE` à la racine de votre dépôt. Optionnellement, vous pouvez ajouter un fichier `CONTRIBUTING.md` pour expliquer comment contribuer au projet (par exemple, comment créer une issue, une merge request, etc.).

### 5.5 Tests

Vous écrirez des tests unitaires pour les classes et méthodes principales de votre projet. Vous utiliserez un framework de tests tel que JUnit pour automatiser l'exécution des tests.

Optionnellement, vous pouvez également écrire des tests de robustesse pour vérifier le comportement de votre programme dans des conditions inattendues, notamment pour gérer les erreurs potentielles dans le fichier de sauvegarde des parties.

Optionnellement, vous pouvez écrire des tests de performance pour mesurer le temps d'exécution de certaines opérations critiques, comme le rendu graphique ou la gestion des collisions.

Optionnellement, vous pouvez mettre en place des tests système pour vérifier le bon fonctionnement de l'ensemble du jeu, par exemple en simulant une partie complète entre deux équipes contrôlées par l'ordinateur.

L'évaluation tiendra compte non seulement de la présence de tests, mais aussi de leur qualité et de leur couverture. Vous pouvez utiliser des outils de couverture de code pour mesurer la couverture de vos tests (tels que JaCoCo ou Cobertura).

### 5.6 Système de compilation, intégration continue et déploiement continu

Vous utiliserez un système de compilation automatisé, tels que Maven ou Gradle, pour gérer les dépendances et automatiser le processus de compilation et l'exécution des tests. Vous intégrerez l'exécution des tests dans votre pipeline CI/CD afin de garantir qu'ils sont lancés à chaque modification du code. Une fois par sprint, vous créerez une *release* de

votre projet en utilisant les fonctionnalités de `GitLab` pour taguer la version et générer des artefacts téléchargeables (par exemple, un fichier JAR exécutable).