Télécom INT
Année scolaire 2024/2025

Projet Informatique 1re Année
PRO 3600

Ducournau Ethan
Prugnat Aurélien
Deleporte Hugo
Benois Loup
Jean-Marie Matisse

Enseignant responsable : Lallet Éric

10/02/2025

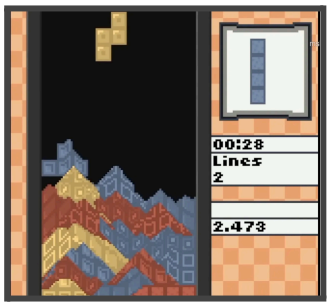
I. Introduction **2**
II. Cahier des charges **3**
III. Gestion de projet **4**
A. Diagramme de Gantt 4
B. Plan de charge 5
C. Suivi d'activité 6
IV. Développement **7**
A. Analyse du problème et spécification fonctionnelle 7
B. Conception préliminaire 8
1. Diagramme de classe 8
2. Fonctionnement 8
C. Conception détaillée 9
1. Lancement du jeu et déroulement général 9
2. Apparition et chute des pièces 10
3. Simulation du sable 10
4. Suppression et traitement des groupes de sable 10
5. Affichage à l'écran 11
6. Contrôles du joueur 11
7. Menus et ambiance 11
D. Choix des outils 12
E. Tests 13
V. Manuel utilisateur **14**
Objectif du jeu 14
Navigation dans les menus 15
Contrôles pendant la partie 16
Fin de partie 17
VI. Conclusion **18**
VII. Bibliographie **19**

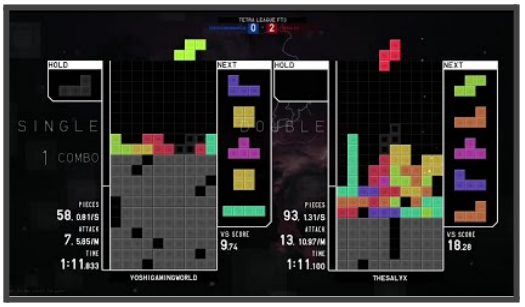
I. Introduction

Le but est de réaliser un jeu sur ordinateur en Java. Pour le jeu, Il s'agit d'un Tetris où les blocs se transforment en sable, soumis à une physique réaliste lorsqu'ils touchent le sol ou d'autres blocs. Pour faire disparaître du sable, du sable d'une même couleur doit toucher simultanément les côtés droit et gauche du niveau.

Dans le cas de la présence d'un mode multijoueur, nous nous inspirerons du jeu web Tetr.io qui implémente des mécaniques de jeu adaptées à un affrontement sous forme de Battle royal (Il s'agit d'un jeu où le but est de rester en vie le plus longtemps).

Sandtris





Tetr.io

II. Cahier des charges

Fonctionnalité	Tests
Implémentation de la physique du sable. Remarque : Chaque grain de sable doit tomber à nouveau lorsque du sable disparaît en dessous.	- Le sable ne doit pas sortir de la fenêtre de jeu - Il doit y avoir des collisions entre les grains de sable - Le bloc doit se transformer en sable
Détection des différentes composantes connexes formées par les différentes couleurs de sable.	- La composante doit savoir si elle touche les bords du terrain - Les diagonales ne doivent pas être considérées
Gestion du score qui prend en compte la taille de la composante qui vient d'être supprimée.	- On marque autant de points que le nombre de grains de sable dans une composante connexe
Implémentation d'une interface de jeu avec : un tableau des "high scores", un menu, un choix de difficulté et le niveau qui est en train d'être joué.	- Le tableau doit être trié par score décroissant - On doit pouvoir changer de menu en cliquant sur les différents boutons.
Un mode multijoueur au format 1vs1 (ou plus) inspiré de "Tetr.io".	- Il faut que les points que l'on obtient viennent provoquer des actions qui dérangent l'adversaire.

III. Gestion de projet

A. Diagramme de Gantt

catégorie	Tache	Semaine 1	Semaine 2	Semaine 3	Semaine 4	Semaine 5	Semaine 6	Semaine 7	Semaine 8	Semaine 9	Semaine 10	Semaine 11	Semaine 12	Semaine 13	Semaine 14	Semaine 15	Semaine 16	Soutenance		
Générale	Squelette du code																			
	Tombé et réaparition des pièces																			
Menu	Menu de démarage																			
	Menu de défaite																			
	Menu de score, Base de donné																			
	Menu de paramètre																			
Partie principale à redécoupé	physique du sable																			**Responsable**
	cassage de la pièce																			
	détection des composants connexes																			Matisse
	Disparition des composants connexes																			Loup
	Condition de défaite																			Hugo
Gestion des pièce	Génération de la forme																			Ethan
	vitesse des pièces																			Aurélien
	Prévisualisation forme + couleur																			
Features	Choix difficulté																			
	Multijoueur locale																			

B. Plan de charge

	PLAN DE CHARGES PRÉVISIONNEL							SUIVI D'ACTIVITÉS (Charge Consommée)						
			Charge en H / Participant							Charge en H / Participant				
Description de l'activité	**Charge en %**	**Charge en H**	**Deleport Hugo**	**Prugnat Aurelien**	**Jean-Marie Matisse**	**Ducournau Ethan**	**Benois Loup**	**Charge en %**	**Charge en H**	**Deleport Hugo**	**Prugnat Aurelien**	**Jean-Marie Matisse**	**Ducournau Ethan**	**Benois Loup**
Total	**100%**	**250**	**50**	**50**	**50**	**50**	**50**	**100%**	**5**	**1**	**1**	**1**	**1**	**1**
Gestion de projets	**11%**	**28**	**7**	**8**	**4**	**5**	**4**	**100%**	**5**	**1**	**1**	**1**	**1**	**1**
Réunion de lancement	**2%**	**5**	1	1	1	1	1	**100%**	**5**	1	1	1	1	1
Planning prévisionnel et Suivi d'activités	**3%**	**8**	0	6	1	1	0	**0%**	**0**	0	0	0	0	0
Réunions de suivi	**2%**	**5**	1	1	1	1	1	**0%**	**0**	0	0	0	0	0
Rédaction	**2%**	**4**	0	0	1	1	2	**0%**	**0**	0	0	0	0	0
Outils collaboratifs (svn, etc.)	**2%**	**6**	5	0	0	1	0	**0%**	**0**	0	0	0	0	0
Spécification	**4%**	**10**	**3**	**2**	**0**	**5**	**0**	**0%**	**0**	**0**	**0**	**0**	**0**	**0**
Définition des fonctionnalités	**4%**	**10**	3	2	0	5	0	**0%**	**0**	0	0	0	0	0
Conception préliminaire	**24%**	**59**	**10**	**10**	**14**	**11**	**14**	**0%**	**0**	**0**	**0**	**0**	**0**	**0**
Définition d'un modèle de données	**10%**	**25**	5	5	5	5	5	**0%**	**0**	0	0	0	0	0
Définition d'un format de fichiers associé au modèle de données	**4%**	**11**	0	2	4	1	4	**0%**	**0**	0	0	0	0	0
Définition des fonctionnalités	**4%**	**11**	3	3	0	5	0	**0%**	**0**	0	0	0	0	0
Définition des modules	**5%**	**12**	2	0	5	0	5	**0%**	**0**	0	0	0	0	0
Conception détaillée	**27%**	**67**	**11**	**15**	**14**	**13**	**14**	**0%**	**0**	**0**	**0**	**0**	**0**	**0**
Définition des classes	**2%**	**4**	2	0	0	2	0	**0%**	**0**	0	0	0	0	0
Définiton des méthodes	**5%**	**12**	3	2	2	3	2	**0%**	**0**	0	0	0	0	0
Définition des tests unitaires	**4%**	**10**	4	2	1	2	1	**0%**	**0**	0	0	0	0	0
Auto-formation	**14%**	**36**	1	10	10	5	10	**0%**	**0**	0	0	0	0	0
Maquettage des interfaces	**2%**	**5**	1	1	1	1	1	**0%**	**0**	0	0	0	0	0
Codage	**22%**	**56**	**13**	**9**	**12**	**10**	**12**	**0%**	**0**	**0**	**0**	**0**	**0**	**0**
Codage des classes	**3%**	**7**	3	2	0	2	0	**0%**	**0**	0	0	0	0	0
Codage des méthodes	**10%**	**25**	10	5	2	6	2	**0%**	**0**	0	0	0	0	0
Codage des tests unitaires	**10%**	**24**	0	2	10	2	10	**0%**	**0**	0	0	0	0	0
Intégration	**0%**	**0**	**0**	**0**	**0**	**0**	**0**	**0%**	**0**	**0**	**0**	**0**	**0**	**0**
Intégration des modules	**0%**	**0**	0	0	0	0	0	**0%**	**0**	0	0	0	0	0
Tests d'intégration	**0%**	**0**	0	0	0	0	0	**0%**	**0**	0	0	0	0	0
Soutenance	**12%**	**30**	**6**	**6**	**6**	**6**	**6**	**0%**	**0**	**0**	**0**	**0**	**0**	**0**
Préparation de la soutenance	**10%**	**25**	5	5	5	5	5	**0%**	**0**	0	0	0	0	0
Soutenance	**2%**	**5**	1	1	1	1	1	**0%**	**0**	0	0	0	0	0

C. Suivi d’activité

Le suivi des activités se fait via les “issues” Github.

IV. Développement

A. Analyse du problème et spécification fonctionnelle

Le projet **Sand Tetris** reprend les principes fondamentaux du célèbre jeu Tetris, tout en y intégrant une mécanique originale basée sur la simulation de sable. L'objectif était de concevoir un gameplay hybride, mêlant réflexion, physique et esthétique.

La principale différence avec un Tetris classique réside dans le comportement des pièces : lorsqu'un bloc en chute entre en contact avec le sol ou avec d'autres blocs déjà posés, il ne se fige pas mais **se désagrège immédiatement pour se transformer en particules de sable**. Ces particules s'accumulent alors naturellement, influencées par la gravité.

Le jeu se déroule sur une **grille rectangulaire verticale**, comme dans Tetris, mais avec une dynamique de jeu totalement modifiée par la nature instable du sable. Cette approche introduit un comportement plus organique et imprévisible, renforçant le défi pour le joueur.

Une règle spécifique renforce la dimension stratégique du jeu : lorsqu'un **amas connexe** de sable de même couleur parvient à **établir une connexion entre les quatre bords de la grille** (haut, bas, gauche et droite), cet amas est automatiquement supprimé. Cette règle incite le joueur à manipuler les pièces non pas pour former des lignes complètes, mais pour créer des connexions globales avec les bords, tout en tenant compte du comportement chaotique du sable.

Les spécifications fonctionnelles majeures retenues pour le développement étaient les suivantes :

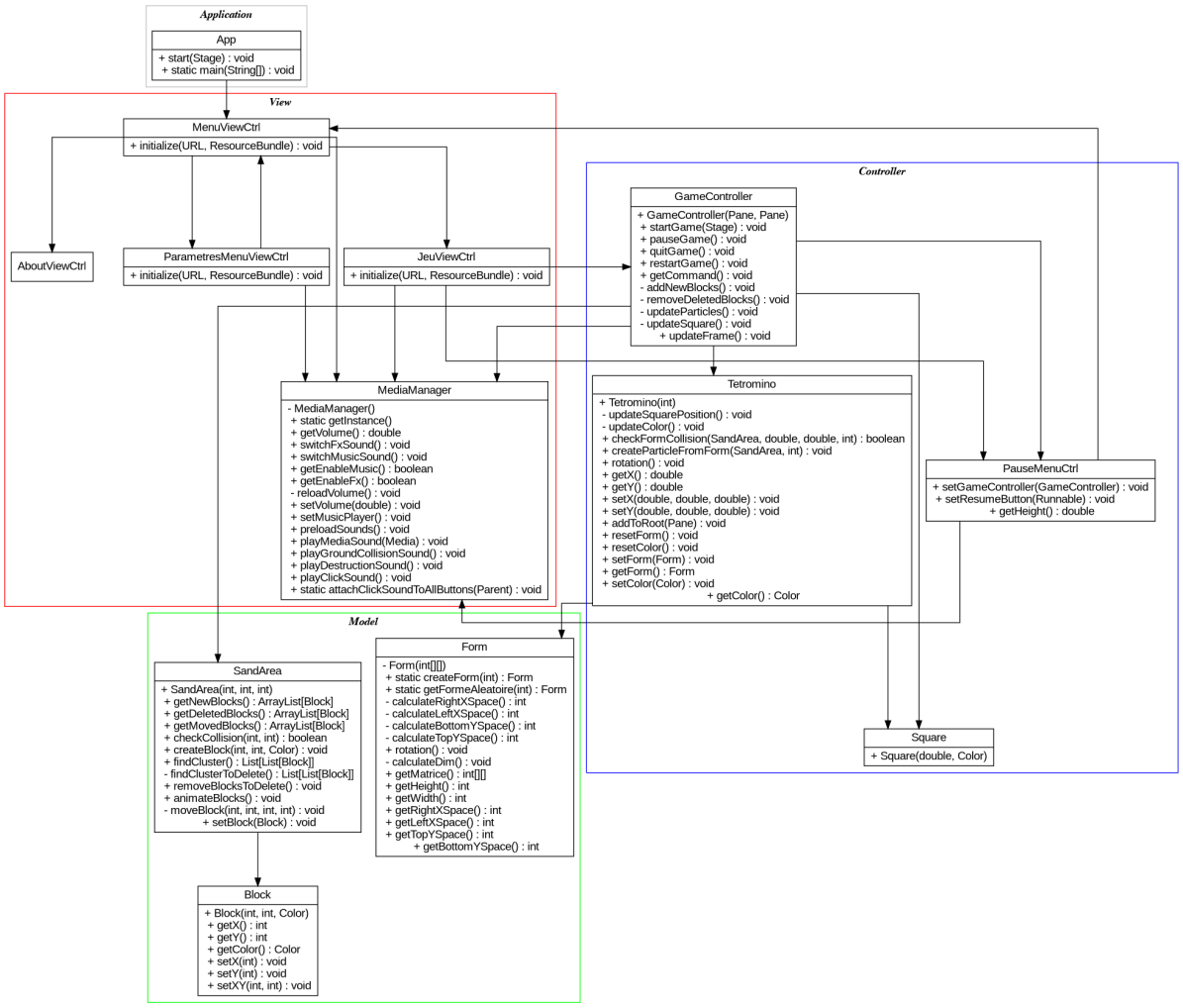
- Détection de collision des blocs avec le sol ou d'autres blocs.
- Désintégration instantanée des blocs en sable au moment de l'impact.
- Simulation de la gravité sur les particules de sable.
- Gestion de la couleur des grains et de leur regroupement.
- Détection et suppression automatique d'un amas connexe de sable de même couleur touchant les quatre bords de la grille.

Ce système de règles, à la fois simple et innovant, constituait le cœur du cahier des charges fonctionnel et a orienté toute la conception de l'application.

B. Conception préliminaire

1. Diagramme de classe

Le diagramme de classe, présenté ci-dessous, expose l'architecture orientée objet adoptée pour ce projet. Elle repose sur une séparation claire entre les responsabilités, notamment à travers l'application du modèle **MVC (Modèle-Vue-Contrôleur)**, afin de distinguer la logique métier de la présentation graphique.



2. Fonctionnement

Le fonctionnement du jeu s'appuie sur une boucle principale animée par une **Timeline JavaFX**, qui exécute régulièrement une fonction d'actualisation du jeu. Cette boucle contrôle à la fois la simulation de la chute des pièces, le comportement du sable et l'affichage.

Une **pièce** est générée aléatoirement (forme et couleur) et placée en haut de la grille. Elle descend selon une **vitesse verticale constante (vy)** jusqu'à entrer en **collision avec le sol ou du sable existant**. Lors de cette collision, la pièce est immédiatement **désintégrée** : elle est remplacée dans la matrice par des **grains de sable colorés**, correspondant à sa forme et sa couleur.

La simulation du **sable** est effectuée dans une matrice dédiée. Chaque grain de sable est soumis à des règles simples :

- Un grain tombe s'il y a un vide directement en dessous.
- Il peut **glisser latéralement** si une **falaise de plus d'une cellule de hauteur** est détectée sur un côté.

Parallèlement, le jeu réalise une **détection de composantes connexes** à chaque itération. Cette détection repose sur un **parcours en profondeur (DFS)** permettant d'identifier les **clusters** de grains de même couleur. Une fois tous les clusters identifiés, chacun est de nouveau parcouru pour vérifier s'il **touche les quatre bords de la grille** (haut, bas, gauche, droite). Si cette condition est remplie, le cluster est automatiquement **supprimé**.

Enfin, l'**affichage** repose sur une architecture distincte du modèle : des **éléments dédiés à la vue** sont générés dynamiquement à chaque mise à jour, reflétant l'état du modèle sans interférer avec sa logique. Cette séparation garantit une meilleure maintenabilité et une clarté accrue dans l'organisation du code.

C. Conception détaillée

Le fonctionnement de Sandtris repose sur une architecture logicielle dite **MVC (Modèle-Vue-Contrôleur)**. Cette structure permet de bien organiser le projet en trois parties : la logique du jeu (le modèle), ce que voit le joueur à l'écran (la vue), et la gestion des interactions (le contrôleur).

1. Lancement du jeu et déroulement général

Lorsque le joueur lance le jeu, celui-ci commence par initialiser l'interface graphique et le menu principal grâce à la classe `App.java`. Dès que la partie est démarrée, une boucle de jeu continue est mise en place via une `Timeline` définie dans `GameController.java`. Cette boucle, qui se répète plusieurs fois par seconde, exécute la méthode `updateGame()`.

Cette méthode orchestre la logique du jeu en appelant successivement plusieurs autres fonctions : elle fait tomber la pièce active à l'aide de `moveDown()`, vérifie si elle est en collision avec d'autres éléments via `checkCollision()`. Cette dernière examine la

position projetée de la pièce et la compare à l'état actuel de la grille. Si un emplacement à venir est déjà occupé, la fonction considère qu'une collision va se produire, et si besoin, la transforme en sable (`transformIntoSand()`) avant de faire apparaître une nouvelle pièce avec `createNewTetromino()`.

2. Apparition et chute des pièces

Les pièces apparaissent en haut de la grille avec une forme et une couleur aléatoires. Les différentes formes sont représentées dans les classes `Tetromino.java` et `Square.java`.

La descente de la pièce est assurée par `moveDown()`, qui ajuste sa position verticale à chaque cycle. Cette méthode vérifie d'abord si l'espace situé juste en dessous de la pièce est libre. Si oui, elle déplace la pièce d'une case vers le bas. Si un obstacle est rencontré (bord inférieur ou sable), elle stoppe le mouvement, signalant ainsi que la pièce doit être transformée. Si une collision est détectée (avec le sol ou du sable existant), la méthode `transformIntoSand()` est appelée. Elle récupère la position des différentes parties de la pièce, puis place à ces emplacements des grains de sable de la même couleur dans la grille. Chaque grain remplace l'ancienne position occupée par la pièce et devient un élément indépendant soumis aux règles de gravité.

3. Simulation du sable

Une fois la pièce désintégrée, chaque grain est intégré dans la grille du jeu, gérée par la classe `SandArea.java`. Le comportement du sable est ensuite simulé dynamiquement par la méthode `updateSand()`.

Cette fonction applique les règles de gravité : elle examine chaque grain de sable pour déterminer s'il peut tomber vers le bas (si la case est vide), ou s'il peut glisser vers la gauche ou la droite si un vide est détecté sur les côtés inférieurs. Cette mécanique simple donne au sable un comportement naturel et fluide.

4. Suppression et traitement des groupes de sable

À chaque étape, le jeu cherche à identifier des groupes de grains de même couleur qui pourraient être supprimés. Pour cela, la méthode `findCluster()` (ou `dfs()`) explore la grille avec un **parcours en profondeur**, une technique permettant de regrouper les grains connectés entre eux par les côtés.

Une fois un groupe identifié, la méthode `checkComponents()` évalue s'il touche simultanément les quatre bords de la grille (haut, bas, gauche et droite). Pour cela, elle passe en revue chaque grain du groupe et note les bords atteints en fonction de leur position. Elle conserve une trace des bords rencontrés, et dès qu'elle détecte que les quatre côtés sont couverts, le groupe est considéré comme valide pour suppression.

Ensuite, la méthode `removeComponent()` est appelée. Cette fonction parcourt tous les grains appartenant au groupe concerné et les retire de la matrice de jeu en les remplaçant par des cellules vides. Elle s'assure également que l'affichage est mis à jour pour refléter

visuellement la suppression du sable. Le score du joueur est alors augmenté en fonction du nombre de grains éliminés.

5. Affichage à l'écran

Ce que le joueur voit à l'écran est mis à jour en continu à partir de l'état de la grille. Les fichiers FXML définissent les éléments graphiques (grille, menu, fond, boutons), tandis que les contrôleurs comme `JeuViewCtrl.java` les animent.

La méthode `updateDisplay()` traduit les données du modèle en affichage visuel : elle colore les cases, affiche les formes, et met à jour les éléments de l'interface selon la progression du jeu. Cette séparation permet de modifier l'apparence sans toucher à la logique interne.

6. Contrôles du joueur

Le joueur peut interagir avec la pièce en cours grâce au clavier. Les touches déclenchent les fonctions suivantes dans `GameController.java` :

- `movedropLeft()` pour déplacer la pièce à gauche,
- `moveRight()` pour aller à droite,
- `moveDown()` pour accélérer la descente,
- `()` pour la faire tomber directement,
- et la touche Échap pour mettre le jeu en pause.

Ces actions sont traitées à chaque cycle pour assurer une réactivité fluide.

7. Menus et ambiance

Les menus (accueil, pause, paramètres) sont conçus en FXML (`MenuView.fxml`, `PauseMenuView.fxml`, etc.) et pilotés par des classes comme `MenuViewCtrl.java`. Ils permettent de naviguer entre les différentes sections du jeu.

L'ambiance sonore est gérée par `MediaManager.java`, qui déclenche des effets sonores lors des actions (clics, suppressions) et joue une musique de fond pendant la partie pour renforcer l'immersion.

En résumé, Sandtris est un jeu qui combine chute de pièces, simulation physique de sable et stratégie de connexion. Chaque fonctionnalité repose sur une méthode spécifique qui réalise sa tâche avec des techniques bien définies (comme la gravité pour `updateSand()`,

ou le parcours en profondeur pour `findCluster()`). Le tout est orchestré dans une architecture claire et modulaire qui sépare la logique, l'affichage et les interactions.

D. Choix des outils

Pour la gestion du code source, nous avons choisi **GitHub** comme plateforme de versionnage. Ce choix s'est imposé naturellement, car tous les membres du groupe étaient déjà familiers avec son fonctionnement. GitHub a permis d'héberger notre dépôt à distance, de suivre les modifications de chacun grâce aux commits, et de collaborer efficacement via les issues et les pull requests. Même si GitLab était une alternative envisageable, notre expérience commune sur GitHub a facilité la communication et la coordination sans nécessiter de temps d'adaptation.

Concernant les environnements de développement, chacun a utilisé l'**IDE** (environnement de développement intégré) qu'il maîtrisait déjà : **IntelliJ IDEA**, **Eclipse**, ou **Visual Studio Code**. Ce fonctionnement distribué n'a posé aucun problème majeur, car le projet étant bien structuré avec Maven, il était facilement portable d'un IDE à un autre. Le format standardisé du projet Java a permis une compatibilité immédiate, sans configuration complexe.

Pour la conception de l'interface graphique, nous avons utilisé **JavaFX** associé à **SceneBuilder**. JavaFX a été préféré à Swing pour plusieurs raisons :

- **Swing est obsolète** : il n'est plus activement maintenu depuis plusieurs années, et son rendu graphique paraît daté.
- JavaFX permet une **meilleure séparation entre la logique métier et l'interface** grâce à son format FXML, ce qui est idéal dans une architecture MVC.
- Il offre une **expérience plus moderne et fluide** pour la création d'interfaces réactives et personnalisées.

Quant à SceneBuilder, cela nous a permis de **concevoir visuellement l'apparence des écrans** (menus, jeu, paramètres) sans avoir à écrire manuellement tout le code de mise en page. Il génère des fichiers `.fxml` qui décrivent **la structure statique de l'interface** :

- quels éléments sont présents (boutons, canevas, labels),
- comment ils sont agencés (grilles, boîtes horizontales/verticales),
- et quels identifiants ils ont pour être utilisés dans le code.

Cet outil s'est révélé très utile pour gagner du temps, visualiser rapidement nos écrans, et les adapter facilement en fonction des retours.

Pour faciliter la compilation et la gestion des dépendances, nous avons utilisé **Maven**. Cet outil nous a permis d'automatiser la construction du projet, de centraliser la configuration et de simplifier le partage de bibliothèques nécessaires, en particulier JavaFX.

E. Tests

Les tests unitaires ont pour but de vérifier le bon fonctionnement de chaque élément du programme de façon isolée. Ils permettent de s'assurer que les différentes fonctions se comportent correctement, même lorsqu'elles sont appelées dans des contextes différents. Cela permet de détecter rapidement les erreurs en cas de modification future du code.

Dans notre projet, l'outil utilisé pour effectuer ces tests est **JUnit**. Il s'agit d'un framework dédié aux tests unitaires en Java. Il permet d'exécuter automatiquement une suite de tests et de vérifier que les résultats obtenus correspondent bien aux résultats attendus. Cela offre un gain de temps significatif et une meilleure confiance dans la fiabilité du code.

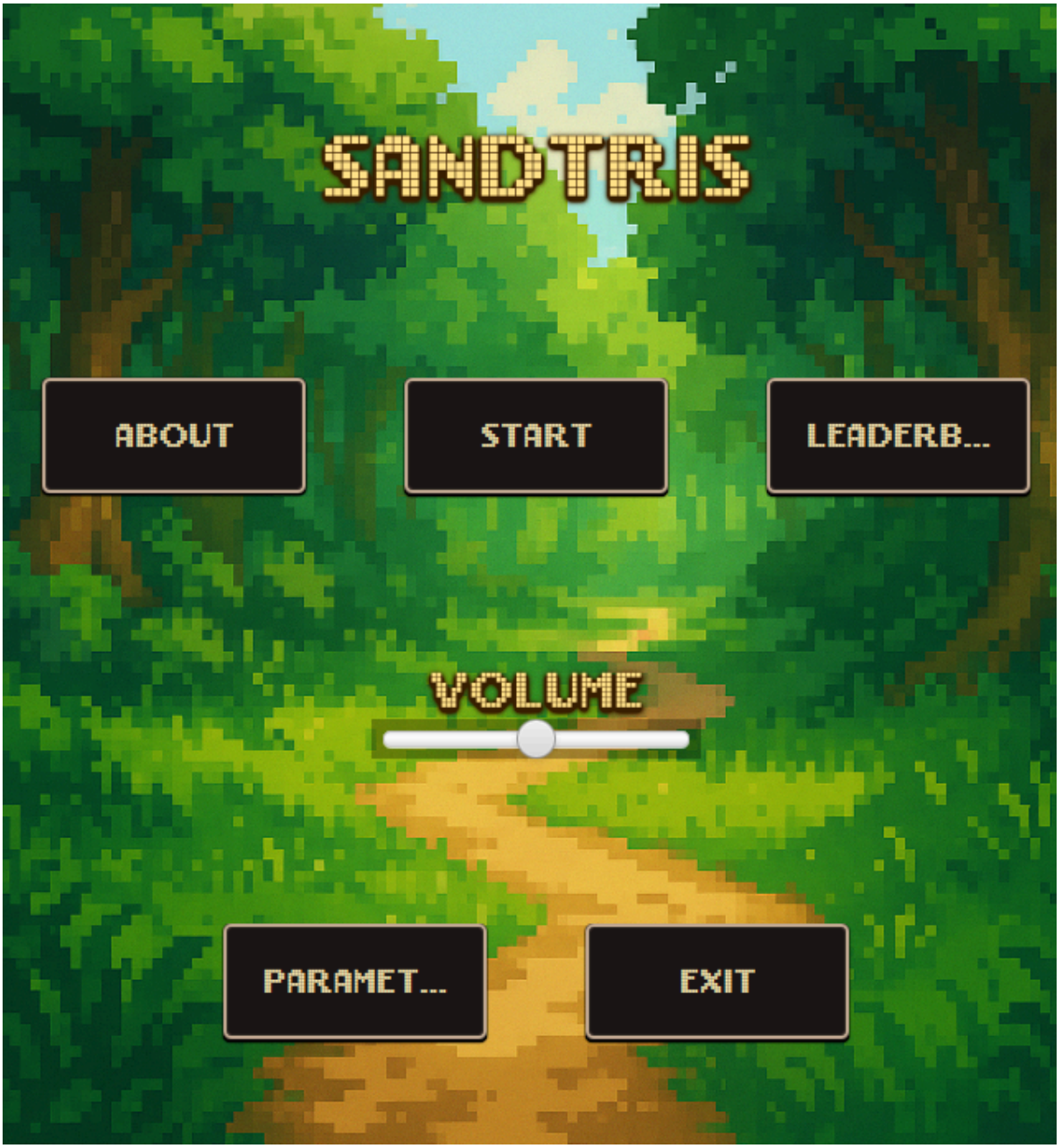
Nous avons décidé de tester en particulier deux fonctions clés du programme : la détection de collision (`checkCollision()`) et la détection de clusters de sable (`findCluster()`).

Ces deux fonctions ont été ciblées car elles jouent un rôle central dans le fonctionnement du jeu. Elles sont sollicitées à chaque cycle de la boucle de jeu, doivent gérer plusieurs cas possibles (différents types de contact, regroupement de couleurs, frontières de la grille), et leur exactitude conditionne directement la stabilité et la jouabilité du système. Tester ces fonctions nous a permis de nous assurer qu'elles réagissent correctement quelle que soit la situation rencontrée par le joueur.

V. Manuel utilisateur

Objectif du jeu

Les pièces descendent automatiquement et se désagrègent en grains de sable dès qu'elles touchent le sol ou d'autres grains. Le but est de former des groupes de sable de même couleur qui touchent **les quatre bords** de la grille (haut, bas, gauche, droite). Ces groupes sont automatiquement supprimés, et le joueur gagne des points selon leur taille.



Navigation dans les menus

Le menu principal s’affiche au démarrage. Le joueur peut :

- **Start** : démarrer une nouvelle partie,
- **Paramètre** : accéder aux options de paramétrage pour activer ou désactiver la musique et les effets sonores,
- **Volume** : régler le niveau sonore global via une barre de volume,
- **Leaderbord** : consulter les scores des différents joueurs,
- **About** : obtenir des informations sur le jeu et le projet,
- **Exit** : quitter le jeu.

Les boutons sont cliquables à la souris, avec des effets sonores associés pour indiquer les interactions.

Contrôles pendant la partie

- la scène a droite en bas du bouton pause permet de voir la prochaine pièce
- en bas s'affiche le score en direct

- Pause : pour faire pause et accéder a l'ecran proposant de soit recommencer, reprendre ou quitter

Une fois la partie commencée, le joueur peut interagir avec la pièce en cours de chute à l'aide des touches suivantes :

- **Flèche gauche** : déplacement vers la gauche,
- **Flèche droite** : déplacement vers la droite,
- **Espace** : fais tourner la pièce de 90°
- **Echap** : pour faire pause

Fin de partie

La partie se termine lorsque les grains de sable atteignent le haut de la grille, bloquant ainsi l'arrivée de nouvelles pièces.

Le joueur peut ensuite retourner au menu principal ou relancer une nouvelle partie.

VI. Conclusion

Le projet Sandtris est inspiré de Tetris avec une innovation sur la physique des blocs. Ce projet peut être considéré comme une réussite. Le jeu fonctionne comme prévu, et nous avons atteint les objectifs que nous nous étions fixés en début de projet. La maquette prévisionnelle a été respectée dans ses grandes lignes, aussi bien sur le plan technique qu'en termes d'organisation du travail.

Tout au long du développement, nous avons su mettre en place une architecture propre et maintenable, en nous appuyant sur le modèle MVC, JavaFX pour l'interface, et Maven pour la gestion du projet. Nous avons également utilisé des outils de versionnage efficaces avec GitHub, et mené des tests unitaires sur des fonctions essentielles afin de garantir la fiabilité du cœur du jeu.

Malgré quelques difficultés liées à la diversité des niveaux techniques dans notre groupe, nous avons su surmonter ces écarts en collaborant étroitement pour combler les lacunes de chacun.

Avec plus de temps, nous aurions aimé aller plus loin, notamment en intégrant un mode multijoueur Ce mode aurait apporté une dimension compétitive supplémentaire intéressante et permis d'élargir les mécaniques de jeu déjà mises en place.

Au final, nous sommes fiers du résultat obtenu : Sandtris est un jeu original, fonctionnel, graphiquement propre, et basé sur une idée de gameplay innovante. Ce projet nous a permis d'acquérir de solides compétences techniques tout en expérimentant un vrai travail en équipe dans un cadre structuré.

VII. Bibliographie

[1] : API java (https://docs.oracle.com/javase/8/docs/api/)