Empire of Lionel

Réalisé par :

Gabriel IFRIM
Arthur MACDONALD

Compte rendu de Projet Java
Cours d'approche object

Master 1 - Semestre 7 - Bordeaux

Introduction

Ce projet a été réalisé dans le cadre du semestre 7 du Master Informatique à l'Université de Bordeaux, dans le cadre du cours "Approche Objet" dispensé par Lionel Clément. L'objectif était de concevoir et développer un jeu de gestion utilisant les principes de la programmation orientée objet. Le projet a été mené sur une période d'un mois, en utilisant JavaFX pour l'interface graphique.

Règles du Jeu

Le but du jeu est de faire croître votre village en atteignant 1000 habitants le plus rapidement possible. Pour cela, vous devez gérer vos ressources et vos bâtiments afin d'accueillir de nouveaux villageois et de les faire travailler. Chaque bâtiment a un rôle spécifique : il permet de produire des ressources nécessaires au développement du village et d'offrir des emplois aux habitants.

Gestion des Ressources et des Bâtiments

Les bâtiments disponibles dans le jeu vous permettront de gérer et de produire des ressources (telles que le bois, la nourriture, etc.). Ils offrent également des emplois pour vos habitants. Chaque résident doit être affecté à un bâtiment spécifique, ce qui génère une production de ressources. Plus vous aurez de bâtiments et d'habitants, plus vous pourrez produire de ressources et ainsi recruter de nouveaux habitants, augmentant ainsi votre population.

Gestion de l'Espace

Les bâtiments occupent une place sur une grille limitée, ce qui signifie que vous devez gérer l'espace disponible avec soin. Vous ne pouvez pas simplement construire partout : il vous faudra réfléchir à l’emplacement de chaque bâtiment pour optimiser votre village et garantir une croissance rapide.

Nourriture et Survie

Vos villageois consomment de la nourriture, et s’ils n’ont pas suffisamment de provisions, ils risquent de mourir de faim. Vous devez donc veiller à avoir suffisamment de nourriture en stock pour tous vos habitants. Si vous manquez de nourriture et que vos villageois meurent, cela pourrait entraîner la perte de votre partie.

Game Over

Si vous perdez tous vos habitants (par manque de nourriture, par exemple), le jeu sera terminé. Il est donc crucial de maintenir un équilibre entre la production de ressources, l'extension de votre village, et la gestion de la population pour atteindre l'objectif des 1000 habitants avant qu'il ne soit trop tard.

Contribution au projet

Nous avons collaboré sur tous les aspects du projet, y compris le développement du backend (logique de l'application) et du frontend (interface utilisateur avec JavaFX). Les tâches n'ont pas été spécifiquement divisées, mais ont été abordées conjointement en fonction des besoins du projet à chaque étape.

Bilan du Projet

Nous avons réussi à implémenter toutes les fonctionnalités de base du jeu, y compris la gestion des bâtiments, des ressources, de la production et du temps. L'équilibrage du jeu a été ajusté pour assurer une expérience de jeu minimale viable.

Ce qui a été fait

- Implémentation des fonctionnalités de base du jeu (bâtiments, ressources, production, temps).
- Ajustement de l'équilibrage du jeu.
- Création d'une interface utilisateur (UI) simple en pixel art.
- Implémentation d'un système de production de ressources basé sur les travailleurs individuels.

Stratégies adoptées

- Système de Production de Ressources Indépendant : Chaque résident a un travail individuel et produit des ressources de manière indépendante, ce qui permet une gestion plus fine de la production et de la consommation.
- Interface Utilisateur (UI) Simple : Une UI minimaliste en pixel art a été choisie pour faciliter la compréhension et l'utilisation.

Design Patterns Utilisés

- Singleton : Garantir qu'une classe n'ait qu'une seule instance.
    - Exemple : `SceneManager.getInstance()` pour obtenir l'instance unique de SceneManager.
- Factory : Créer des objets sans spécifier la classe exacte de l'objet à créer.
    - Exemple : `WorkFactory.createWork(WorkType.LUMBERJACK)` pour créer une instance de `Lumberjack`.
- Observer : Permettre à un objet (observateur) de recevoir des notifications de changement d'état d'un autre objet (sujet).
    - Exemple : `GameTime` notifie les objets `Work` (implémentant `TimeObserver`) des changements de temps.
- MVC : Séparer les préoccupations de l'application en trois parties : le modèle (données), la vue (interface utilisateur) et le contrôleur (logique).
    - Modèle : Représente les données de l'application (par exemple, `Resident`, `Building`).
    - Vue : Représente l'interface utilisateur (par exemple, fichiers FXML comme `main.fxml`).

    - Contrôleur : Contient la logique de l'application et interagit avec le modèle et la vue (par exemple `GameManager`)

Ce qu'il reste à faire

- Améliorations Bonus : Implémentation de la gestion des déplacements des résidents et d'un système d'amélioration des bâtiments. Ces améliorations ont déjà commencé à être ajoutées en code et en ressources (images) mais nous n'avons pas eu le temps de finir les implémentations.

Difficultés rencontrées

- JavaFX : La création de l'UI en pixel art avec JavaFX s'est avérée complexe. Nous n'avons malheureusement pas pu obtenir le résultat voulu. En effet JavaFX n'est pas un outil très adapté pour ce genre d'application et nous avons eu des complications pour placer certains éléments sur l'écran.

Améliorations possibles

- Amélioration de l'UI.
- Ajustement supplémentaire de l'équilibrage du jeu.
- Ajout de la fonctionnalité d'upgrade des bâtiments pour qu'ils produisent plus de ressources.
- Ajout d'une représentation visuelle des travailleurs se rendant à leur poste de travail et rentrant chez eux la nuit.

Comment jouer

1. **Utiliser Maven pour exécuter le projet.** Depuis le répertoire racine :
2. `cd elo/`
3. `mvn clean javafx:run`

Structure des Classes et Relations

L'image est également présente depuis la racine dans le dossier /output.

Classes Principales

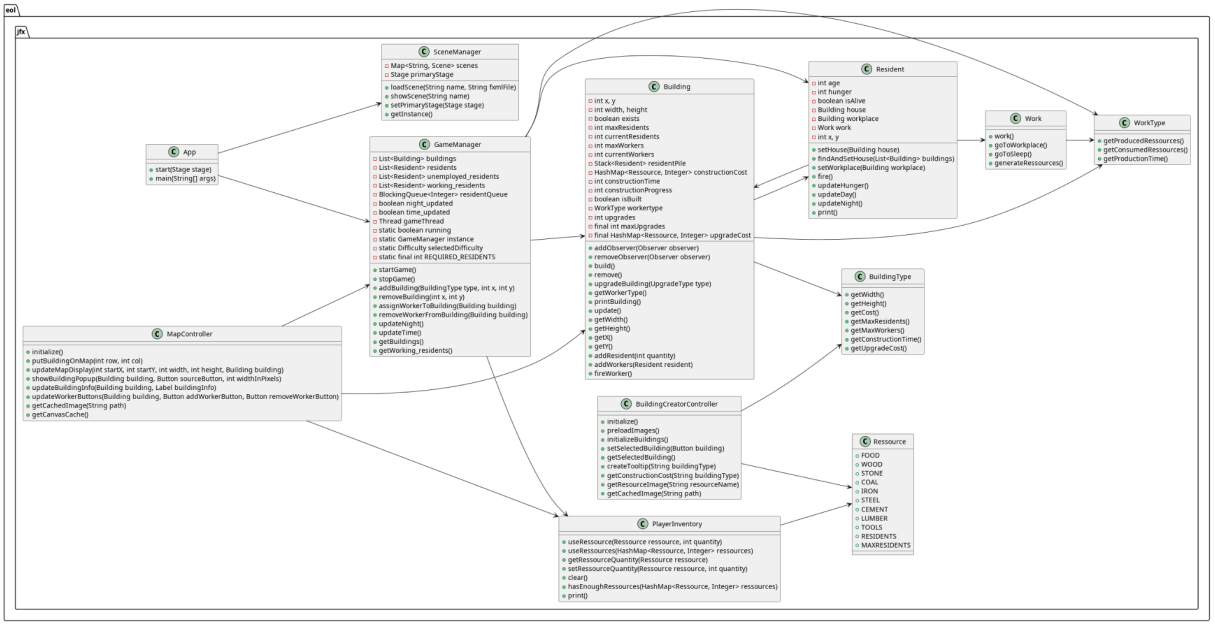
- `Building` : Classe abstraite représentant un bâtiment. Chaque bâtiment a des caractéristiques telles que la taille, le coût de construction, le nombre maximum de résidents et de travailleurs, etc.
- `Resident` : Représente un habitant du village. Chaque résident a un travail et une maison.
- `Work` : Classe abstraite représentant un travail. Chaque travail produit et consomme des ressources.
- `PlayerInventory` : Singleton gérant les ressources du joueur.
- `GameManager` : Singleton gérant l'état du jeu, les bâtiments, les résidents, etc.

- `GridMap` : Gère la grille sur laquelle les bâtiments sont placés.

Relations entre les Classes

- `GameManager` gère les instances de `Building` et `Resident`.
- `Building` utilise `WorkType` pour déterminer le type de travailleur qu'il peut accueillir.
- `Resident` a une référence à `Building` pour sa maison et son lieu de travail.
- `PlayerInventory` gère les ressources consommées et produites par `Work`.

Diagramme de relations



Retour personnel sur le projet

Ce projet a été une expérience extrêmement enrichissante. Il nous a permis de mettre en pratique les concepts de programmation orientée objet que nous avons appris en cours, tout en nous confrontant à des défis réels de développement logiciel. La conception et la mise en œuvre d'un jeu de gestion complexe nous ont permis de développer nos compétences en Java et en JavaFX, ainsi que notre capacité à travailler en équipe. Nous avons particulièrement apprécié la possibilité de voir nos idées prendre forme et de créer un produit fini fonctionnel. Ce projet a été non seulement un excellent exercice académique, mais aussi une expérience très gratifiante sur le plan personnel et professionnel. Nous espérons que notre travail sera apprécié et que le jeu sera amusant à jouer pour tous.