

Projet de Programmation | SR1a

Par Mohamed Ben El Mostapha, Clément Blavy, Anthony Fernandes et Yacob Zitouni

Sujet

Résumé des fonctionnalités implémentées

Présentation du jeu

Commandes

Menus

Armes

Ennemis

Améliorations

Scénarios

Fonctionnalités prévues mais non implémentées

Organisation

Utilisation des issues et de labels

Git flow

Conventions

Structure globale du Projet

Structure du Modèle

Gestion des ressources

Modélisation de la carte et de tuiles

Scénarios de partie

Hierarchie des entités

Mouvement et physique

Utilisation de la programmation orientée événements

Implémentation en elle-même

Implémentation des murs cassable

Gestion de l'IA

Utilisation du Flood Fill pour l'IA

Utilisation de l'algorithme de Bresenham modifié

Modélisation des armes

Utilisation des design patterns

Composition pattern

Strategy pattern

Factory pattern

Observer pattern

Structure de la vue

Look and feel

Menus jetables

Sélecteur de ressource
Vue en jeu
Animations
Éditeur de cartes
Modèle de l'éditeur
Vue de l'éditeur
Difficultés rencontrées

Sujet

Ce projet consiste à créer un jeu de tir en vue de dessus où le joueur affronte des ennemis dans diverses arènes. Chaque arène est composée d'une grille de tuiles avec des sols, des murs destructibles et des obstacles infranchissables pour les personnages mais pas pour les projectiles. Les personnages peuvent tirer des projectiles et poser des pièges pour vaincre les ennemis. Des améliorations sont attendues, telles que des comportements d'ennemis variés, des effets de projectiles et de pièges diversifiés, ainsi que des éléments décoratifs. Des fonctionnalités supplémentaires, comme des effets graphiques, un menu, un éditeur de niveau et le support multijoueur local, sont recommandées. Le travail est organisé en deux phases : une phase initiale de développement d'un prototype minimal suivi d'une période d'améliorations.

Résumé des fonctionnalités implémentées

- Variété de cartes chargeable via des fichiers textes
- Scénarios de partie personnalisables à l'aide de fichiers XML
- Carte composées de tuiles aux comportements variés
- Murs cassables
- Plusieurs types d'armes et de projectiles
- Plusieurs types d'ennemis avec différentes intelligences artificielles
- Mode marathon (mode de jeu infini)
- Fenêtre redimensionnable et menus s'adaptant dynamiquement
- Menus d'accueil et de customisation de la partie
- Éditeur de carte et exportation en fichiers texte
- Effets graphiques (animations, traces de pas...)
- Jeu transportable dans un fichier `.jar`

Présentation du jeu

L'objectif du joueur est de survivre le plus longtemps aux vagues successives et d'en tuer le plus possible avant de mourir. Pour cela il a la possibilité de ramasser différentes armes qui apparaîtront dans l'arène au cours de la partie.

Commandes

- Déplacements : ↑ Z ← Q ↓ S → D
- Changement d'arme : A
- Attaque : clic gauche
- Ramasser une arme : E
- Attaquer : clic gauche
- Viser : déplacer la souris

Menus

À l'exécution, la fenêtre du jeu s'ouvre sur le menu principal. Celui-ci donne accès sélecteur de niveau et à l'éditeur de carte intégré.

Le sélecteur de niveau permet de choisir une carte (depuis la liste des cartes par défaut ou bien depuis l'ordinateur en cliquant sur "open"), ainsi qu'un scenario (également depuis une liste ou l'ordinateur) qui détermine comment les armes et les ennemis apparaissent. Le mode Marathon est un scenario par défaut qui n'a pas de fin et une difficulté croissante.

L'éditeur de carte permet de créer des cartes, à partir de zéro ou en se basant sur des cartes existantes, et de les exporter pour pouvoir les ensuite utiliser dans le menu de lancement d'une partie.

Armes

Le joueur commence la partie avec un couteau équipé et peut tenir jusqu'à deux armes. La plupart des armes ont une jauge indiquant le temps de recharge de l'arme. Ci-dessous la liste des armes et une courte description :

Nom de l'arme	Description
Couteau	Attaque dans la direction du déplacement du joueur, ou de son curseur s'il est immobile. L'attaque inflige des dégâts et applique un saignement qui inflige des dégâts sur la durée.

Nom de l'arme	Description
Poseur de pièges simples	Pose un piège simple à la position du joueur. Lorsqu'un ennemi marche sur un piège, il subit des dégâts. Aucun temps de recharge. Après la troisième utilisation, l'arme disparaît.
Pistolet	Tire vers le curseur. Très faible temps de recharge.
Pistolet rebondissant	Tire 4 projectiles rapides qui rebondissent 4 fois avant de disparaître. Temps de recharge moyen.
Fusil à pompe	Tire 5 projectiles aléatoirement dans un cône en face du joueur au moment où il enfonce le clic et au moment où il le relâche. Temps de recharge moyen.
Mitrailleuse gatling	Tire à haute cadence en face du joueur tant que le clic est enfoncé. Au bout d'une certaine durée, le joueur doit s'arrêter de tirer pour permettre à l'arme de se recharger.
Lance-flammes	Lance des flammes à très haute cadence dans un grand cône en face du joueur. Au bout d'une certaine durée, le joueur doit s'arrêter de tirer pour permettre à l'arme de se recharger. Les flammes font peu de dégâts mais brûlent les ennemis, ce qui leur inflige des dégâts sur la durée.
Lance-roquettes	Lance un projectile lent qui explose à l'impact. Peut infliger des dégâts au joueur. Très long temps de recharge.
Fusil gravitationnel	Tire un trou noir qui s'arrête après une courte durée. Une fois arrêté, le trou noir aspire les ennemis à proximité et leur inflige de faibles dégâts.
Lance-tornades	Lance une tornade qui inflige de faibles dégâts dans une grande zone et pousse les ennemis qu'elle rencontre. Temps de recharge moyen

Ennemis

Le jeu possède trois types d'ennemis, chacun avec leur propre comportement.

Nom de l'ennemi	Description
Ennemi de base	Se déplace vers le joueur et lui inflige des dégâts au corps à corps.

Nom de l'ennemi	Description
Ennemi explosif	Se déplace rapidement vers le joueur et explose à son contact.
Ennemi intelligent	Reste à distance et tire sur le joueur.
Ennemi à amélioration	Fuit le joueur tant qu'il est loin, l'attaque au corps à corps s'il est proche. Donne des améliorations quand il meurt. Possède beaucoup de points de vies.

Améliorations

Nom de l'amélioration	Description
Vitesse	Augmente la vitesse du joueur
Dégâts	Multiplie les dégâts infligés par le joueur
Santé	Augmente la santé maximum du joueur
Régénération	Octroie au joueur une régénération passive

Scénarios

Plutôt que des apparitions complètement aléatoire, les ennemis et les armes apparaissent selon un scénario. Nous avons placé quelques scénarios par défaut dans le jeu, mais n'importe qui peut créer les siens et les charger depuis l'écran de lancement puisqu'ils sont stockés sous forme de fichiers XML. De plus, un scénario spécial "Marathon" permet de jouer indéfiniment avec une difficulté augmentant progressivement au cours du temps.

Pour un exemple, voir le titre scénario de [partie plus bas](#).


Fonctionnalités prévues mais non implémentées

- Mode multijoueur : le modèle ne fait que peu de différences entre le joueur et les autres entités, il devrait donc être facile à implémenter. Le contrôleur du joueur est également conçu pour nécessiter peu de modifications lors de l'ajout. Cependant, on rencontre une autre limitation simple mais contraignante: on ne peut avoir qu'une seule souris à la fois. Une solution serait d'utiliser une manette, mais swing ne les gère pas. Il faudrait donc une implémentation personnalisée pour chaque plateforme. Nous avons donc jugé qu'il était préférable de se concentrer sur d'autres fonctionnalités.


Organisation

Utilisation des issues et de labels

Le travail était réparti en utilisant la création d'issues bien descriptives, qui isolaient un problème précis à traiter :

Closed Opened 1 month ago by  BEN EL MOSTAPHA mohamed Reopen issue New issue

Improve Shooting Mechanics




Expected features :

- Shoot in the direction of the cursor
- Display the weapon cool down
- Display weapon name on the screen

Pour être le plus clair possible, on a profiter des labels dans gitlab :

Handling Tile Graphics

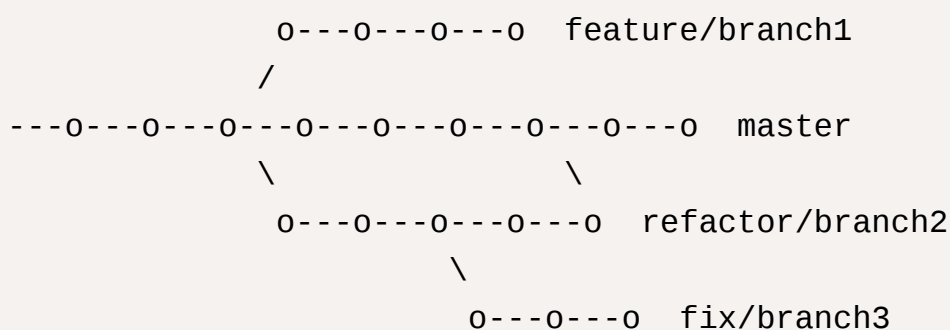
#14 · opened 2 months ago by BEN EL MOSTAPHA mohamed  Ghost Map GUI priority 1

Proposal for Project Structure and Package Organization

#1 · opened 2 months ago by ZITOUNI Yacob discussion

Git flow






Nous avons utilisé une version simplifiée de GitFlow , comprenant une branche master ainsi que des branches feat, refactor et fix :








Notre processus de progression implique l'ouverture de demandes de fusion (Merge Requests) que chacun d'entre nous vérifie. Après des discussions autour des changements, la branche est fusionnée

Exemple :

ZITOUNI Yacob @zitouni added **implementation** label 2 months ago






 BEN EL MOSTAPHA mohamed @benelmos · 2 months ago Maintainer    

Looks good. However, I do think it's generally a good idea to separate the view updater from the model updater. The reason being that a problem in the view (e.g significant delays), would stop the game logic from being updated, which could lead to all kinds of bugs later on. That's my only remark.






 ZITOUNI Yacob @zitouni · 2 months ago Maintainer    

True,i agree we should do that.

I realize that it that case, all we have to do is create 2 GameLoop objets (1 for the model and 1 for the loop)

 BEN EL MOSTAPHA mohamed @benelmos · 2 months ago Maintainer    

That's right actually, sounds good.

 ZITOUNI Yacob @zitouni · 2 months ago Maintainer    

[@fernanda](#) suggested we use java.util.Timer instead of Swing's timer

the main difference is that code ran in swing's timer is directly ran in the EDT. But using a swing component for the model also feels wrong, and in any case calls to repaint() on the view are already ran in the EDT

ZITOUNI Yacob @zitouni added 11 commits 2 months ago
[Compare with previous version](#)

ZITOUNI Yacob @zitouni changed the description 2 months ago

ZITOUNI Yacob @zitouni changed target branch from **develop** to **master** 2 months ago

Conventions

Commits :

Avoir des commits clairs et compréhensibles est primordial dans un projet d'équipe. À cet effet, nous avons décidé de nous fixer une convention pour éviter la confusion et mettre à profit les messages de commit.

Description :

```
<type> (<scope facultatif>): <description>
Ligne de séparation vide
<corps facultatif>
Ligne de séparation vide
<pied de page facultatif>
```

Exemple d'utilisation :

refactor: use canEnterConditions set

This commit changes the way the canEnter method is implemented in the TileModel class. Instead of using a method, a set of predicates is used in the canEnter function. This allows for more flexibility in the conditions that can be checked.

Branches :

De même pour les branches, il faut veiller à ce que les noms soient aussi informatifs que possible pour éviter tout malentendu possible. Pour cela, on s'est fixé cette convention.

Description :

```
git branch <category/reference/description-in-kebab-case>
```

Exemple d'utilisation :

feat/issue-26/knife-weapon
ae60397d · Merge remote-tracking branch 'origin/master' into feat/issue-26/knife-weapon · 1 month ago

78 4

Merge request

Compare

Structure globale du Projet

Pour l'organisation générale du projet nous avons opté pour Maven's standard directory layout :

```
└─ src
  └─ main
    ├── java      # Main code
    └─ resources  # Main resources
  └─ tests
    ├── java      # Test code
    └─ resources  # Test specific resources
└─ design        # Sources for resources that we design
                  # (e.g. .xcf files)
```

Pour l'organisation des packages, nous avons utilisé la structure suivante :


```

controller
model
├─ ingame          # Core model for in-game functionaliti
|                  # (after we start a
├─ config          # Storage of user-defined configuratio
|                  # (if implemente
└─ level (or map)  # Models for levels/maps, and logic to
                   # them from file

gui
├─ animations      # Everything concerning animations
├─ editor          # Map editing menu
├─ ingame          # In-game rendering (map, enemies...)
└─ launcher        # Game launcher menus

util               # Miscellaneous classes/tools

```

▼ Arborescence complète des répertoires du projet

```

.
├─ designs
│   └─ icons
├─ docs
└─ src
    └─ main
        ├── java
        │   ├── controller
        │   ├── gui
        │   │   ├── animations
        │   │   ├── editor
        │   │   ├── ingame
        │   │   │   ├── effects
        │   │   │   ├── entity
        │   │   │   ├── footprints
        │   │   │   └─ tile
        │   │   └─ launcher
        │   ├── model
        │   │   ├── ingame
        │   │   └─ entity

```

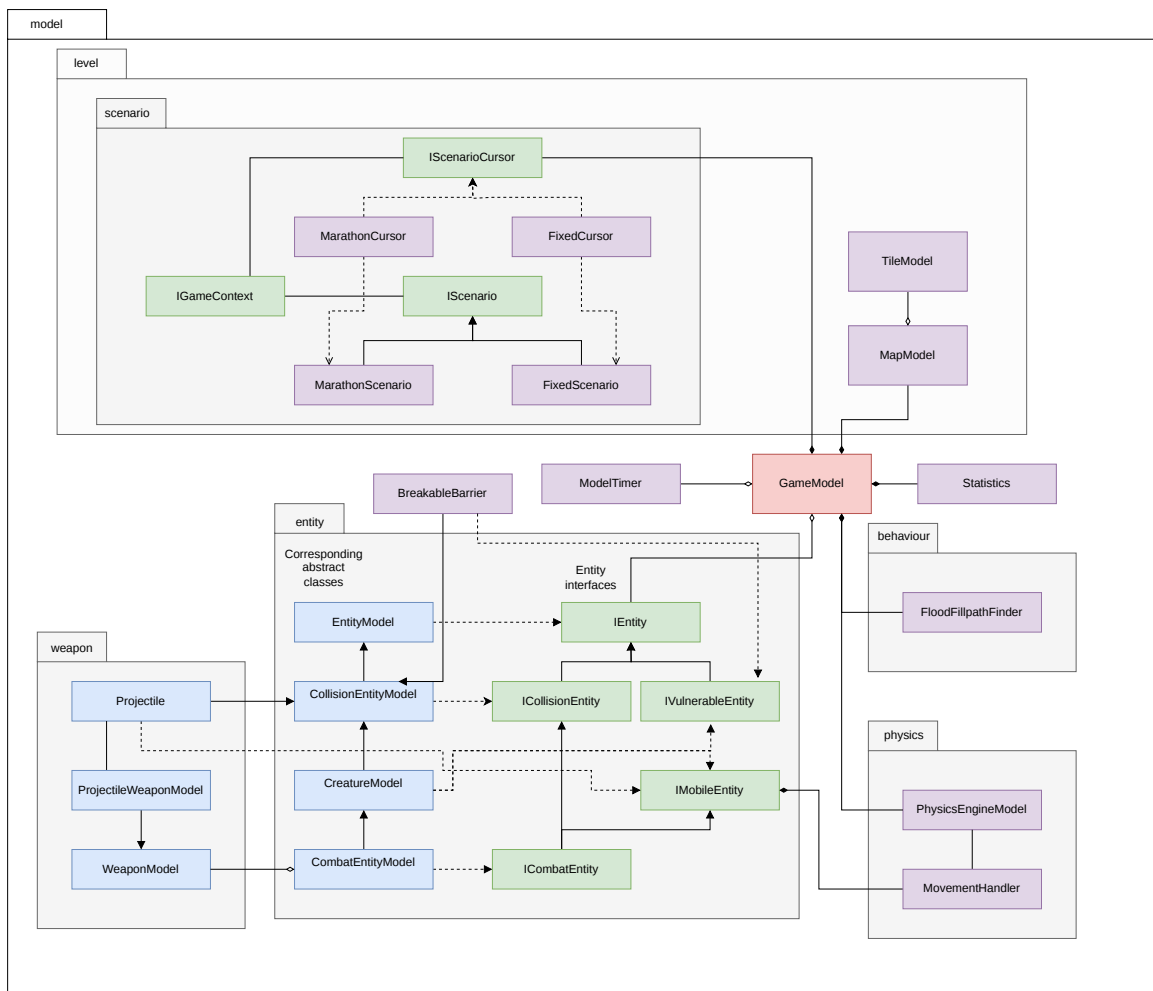
```

|   |   |   |   └─ behavior
|   |   |   └─ physics
|   |   └─ weapon
|   └─ level
|       └─ scenario
|       └─ tiles
└─ util
└─ resources
    └─ gui
        └─ editor
        └─ ingame
            └─ entity
                └─ animations
                └─ sprites
                    └─ explosion
                    └─ knife_slash
                    └─ player1
                        └─ left
                        └─ right
                        └─ up
                    └─ weapon
            └─ footprints
            └─ tile
                └─ animations
                └─ sprites
                    └─ grass
                    └─ water
            └─ laf
                └─ arrow
                └─ border
                └─ icon
            └─ launcher
    └─ model
        └─ level
            └─ maps
            └─ scenario

```

Structure du Modèle

▼ Diagramme des **principales classes** de `model`. Toutes les classes ne sont pas incluses, par soucis de lisibilité, seules celles dont nous avons jugé la présence dans le diagramme bénéfique pour la compréhension ont été incluses.



Bleu: classes abstraites, Vert: interfaces, Violet: implémentations, Rouge: GameModel

Gestion des ressources

Java distingue deux types de ressources :

- Les ressources internes au projet, chargées via la méthode `Class.getResourceAsStream` et indépendantes du système de fichiers de l'ordinateur car lues depuis le classpath. Ces ressources servent pour les fichiers qui sont livrés dans le projet et donc potentiellement incluses dans un `.jar` (d'où l'importance d'être indépendant du système de fichier, les

fichiers dans une archive n'existent pas sur le disque). C'est comme ça que sont lus les cartes et les scénarios par défaut du jeu.

- Les ressources externes au projet, qui sont des fichiers sur le disque et qui peuvent être lues avec de multiples classes (nous utilisons la classe `File`). C'est ce type de ressource qui est utilisé pour lire les cartes et les scénarios customisés créés par les joueurs et enregistrés dans des fichiers texte/XML.

Pour concilier ces deux visions, les ressources qui peuvent être internes ou externes sont abstraites dans une unique classe `Resource` pouvant être construite avec un `File` ou bien avec un chemin dans le classpath et une fonction `asStream`. Peu importe la façon dont elle est construite, le contenu d'une ressource est lu sous forme d'un `InputStream` créé grâce à la fonction `Resource.asStream`. Ces ressources peuvent être réutilisées autant de fois que nécessaire, ce qui est particulièrement pratique quand le joueur veut relancer la même partie.

Pour les ressources exclusivement internes qui sont lues de nombreuses fois (c'est à dire les images et les animations), nous avons mis en place un système de cache pour éviter de devoir parser et charger en mémoire le même fichier encore et encore. Le gain de performance est devenu particulièrement notable quand les animations ont été ajoutées au jeu, car sans cache, chaque image d'une animation doit être rechargée plusieurs fois par seconde.

Les deux classes de cache, `ImageCache` et `AnimationCache`, fonctionnent sur le même principe : elles sont munies d'un dictionnaire qui relie le nom d'une image/animation à l'objet la représentant. Pour changer une image/animation, il suffit d'utiliser la fonction `load...` du cache associé, cette fonction vérifie d'abord si l'image/animation existe déjà dans le cache (et la renvoie le cas échéant) avant de lancer le parsing du fichier.

Modélisation de la carte et de tuiles

En dehors du jeu, les cartes sont stockées dans des fichiers texte selon le format spécifié par le fichier `docs/mapFormat.md`. De tels fichiers peuvent facilement être créés avec l'éditeur inclus dans le jeu et être chargés en jeu via le sélecteur de ressources.

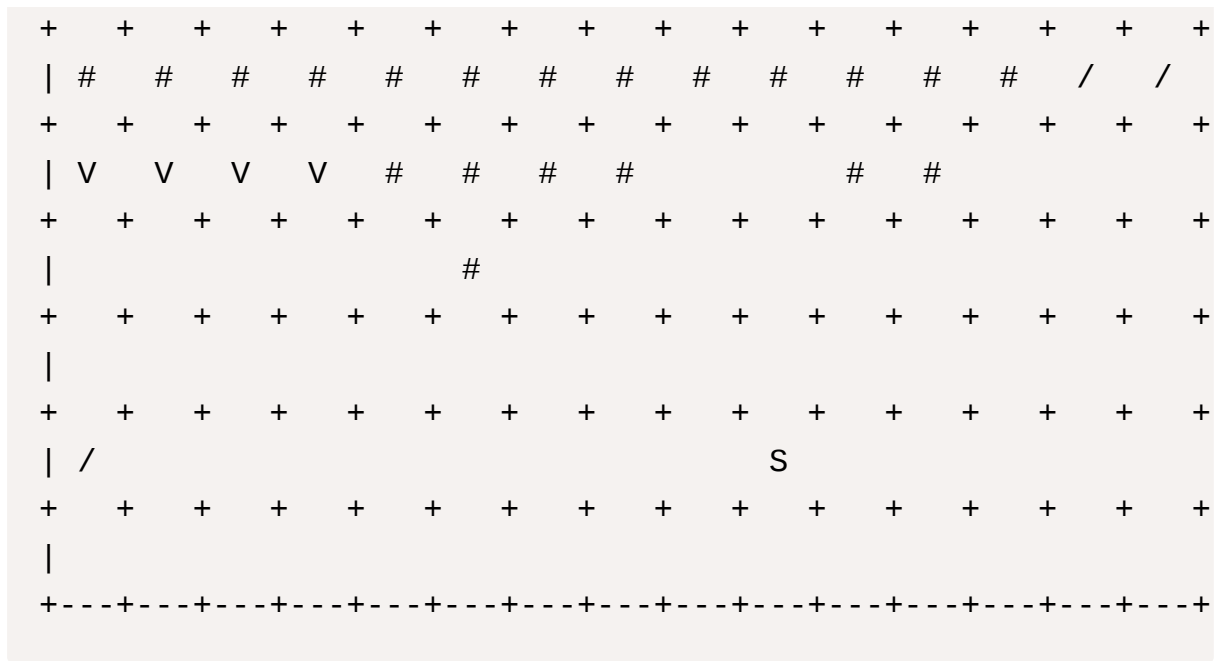
Une fois la carte chargée en jeu, une tuile est représentée par une instance d'une sous classe de la classe abstraite `TileModel`. Chaque tuile contient l'ensemble des entités collisionnables se trouvant sur cette tuile (utilisé pour économiser des calculs dans le moteur physique) ainsi qu'un ensemble de prédicats qu'une entité doit respecter pour entrer sur cette tuile (encore une fois, utilisé par le moteur physique). Certaines tuiles apparaissent également avec une entité `BreakableBarrier` à l'intérieur, qui empêche tout entité d'entrer dans la tuile tant qu'elle n'est pas cassée.

Voici la liste des types de tuiles et leur conditions d'entrée :

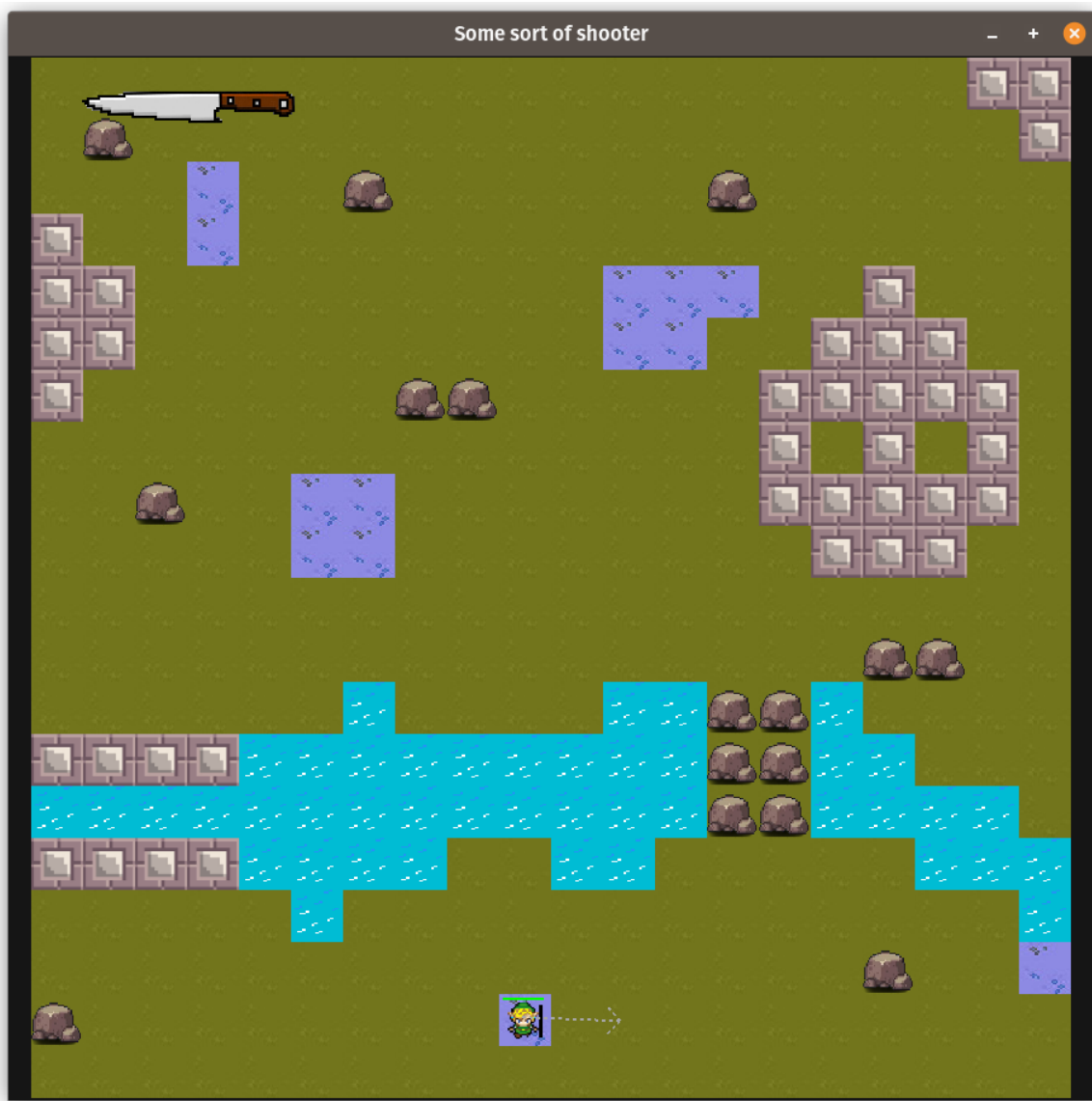
Type de tuile	Entités pouvant y entrer
Standard	Toutes les entités
Vide	Aucune entité
Eau	Uniquement les projectiles
Protégée (Safe)	Joueur uniquement

Ces tuiles sont rassemblées dans la classe `MapModel` qui permet de les manipuler collectivement pour facilement y obtenir et y déplacer les entités. Voici un exemple de carte extrait du code du jeu (il s’agit de la map “ruins”) :

```
+-----+
|
+ + + + + + + + + + + + + + + +
| /
+ + + + + + + + + + + + + + + +
|          S          /          /
+ + + + + + + + + + + + + + + +
| V          S
+ + + + + + + + + + + + + + + +
| V  V          S  S  S
+ + + + + + + + + + + + + + + +
| V  V          S  S
+ + + + + + + + + + + + + + + +
| V          /  /          V
+ + + + + + + + + + + + + + + +
|          V
+ + + + + + + + + + + + + + + +
|          /          S  S          V
+ + + + + + + + + + + + + + + +
|          S  S
+ + + + + + + + + + + + + + + +
|
+ + + + + + + + + + + + + + + +
|
+ + + + + + + + + + + + + + + +
|          #          #  #  /  /
+ + + + + + + + + + + + + + + +
| V  V  V  V  #  #  #  #  #  #  #  #  /  /
```



Et son interprétation une fois en jeu :



Scénarios de partie

L'apparition des ennemis et des armes n'est pas complètement aléatoire mais suit un scénario. Nous avons créé quelques scénarios stockés dans les ressources du jeu sous forme de fichiers XML, mais l'utilisateur peut choisir de créer les siens et les charger. Nous avons fait le choix du XML car les outils de parsing pour ce format sont déjà présent par défaut dans le JDK. Il y a aussi un scénario spécial "marathon" sans fin et avec une difficulté croissante.

Chaque scénario est une succession de `IGameContext` pendant des intervalles de temps donné. Chaque `GameContext` décide la fréquence d'apparition des armes et des ennemis, ou bien indique qu'il faut faire apparaître un groupe d'armes/ennemis une seule fois (OneShot). Les scénarios peuvent soit boucler lorsqu'ils arrivent à la fin (pour des parties infinies), soit arrêter le jeu à la fin, soit maintenir leur dernier état.

Pour cela, nous avons créé une classe générique `IntervalMap` (dans le package `utils`) qui associe des intervalles à des valeurs et gère les comportements en fin de scénario. On utilise ensuite un `IntervalMapCursor` pour le parcourir facilement en avançant dans le temps. On a ainsi une classe `FixedScenario` qui étend `IntervalMap<Double, IGameContext>` et `FixedCursor` qui étend `IntervalMapCursor<Double, IGameContext>`.

Concernant le mode marathon, un scénario spécial est intégré programmatiquement. Ce scénario ne stocke pas grand chose, mais il permet de créer un curseur correspondant, qui lui gère une augmentation progressive de la difficulté au cours du temps. Les probabilités d'apparition d'ennemis sont déterminées à partir d'une loi binomiale.

Exemple de scénario XML (plus de détails sont disponibles [dans la documentation](#) et dans le fichier `docs/scenarioFormat.md`):

```
<scenario endReachedBehaviour="LOOPING">
  <!-- Interval 1, starts at 0, ends at 3, spawns one
  pistol, one knife, and one bandage -->
  <interval type="oneshot" time="3">
    <weapons>
      <weapon name="Pistol"/>
      <weapon name="Knife"/>
    </weapons>
    <enemies>
    </enemies>
    <miscs>
      <misc name="Bandages"/>
    </miscs>
  </interval>
  <!-- Interval 2, starts at 3, ends at 10, spawns Walk
  ingEnemy at a rate of 0.3 per second, and SmartEnemy at 0.1
  per second -->
  <interval type="fixed" time="10">
    <weapons>
    </weapons>
    <enemies>
      <enemy rate="0.3" name="WalkingEnemy"/>
      <enemy rate="0.1" name="SmartEnemy"/>
    </enemies>
    <miscs>
    </miscs>
  </interval>
</scenario>
```



```
</interval>
</scenario>
```

Hiérarchie des entités

Java ne possède qu'une mécanique d'héritage simple pour les classes, ce qui n'est pas assez pour représenter des entités qui peuvent avoir des comportements similaires mais qui ne sont pas pour autant des cas particuliers les uns des autres. Par exemple, parmi les entités à collision, certaines sont mouvantes et vulnérables, d'autres sont uniquement vulnérables et d'autres encore uniquement mouvantes.

La solution trouvée pour tout de même lier ces classes est d'utiliser une double hiérarchie avec d'un côté des interfaces utilisant de l'héritage multiple, et de l'autre côté des classes utilisant un héritage simple et implémentant uniquement les interfaces qui les caractérisent.

Cette approche permet une grande extensibilité. Par exemple, pour ajouter une entité qui serait vulnérable mais sans collision (il faudrait peut-être lui faire des dégâts en tuant d'autres entités, en tapant des piliers...), il suffit de faire en sorte qu'elle étende `EntityModel` en implémentant uniquement `IVulnerableEntity` et le tour est joué.

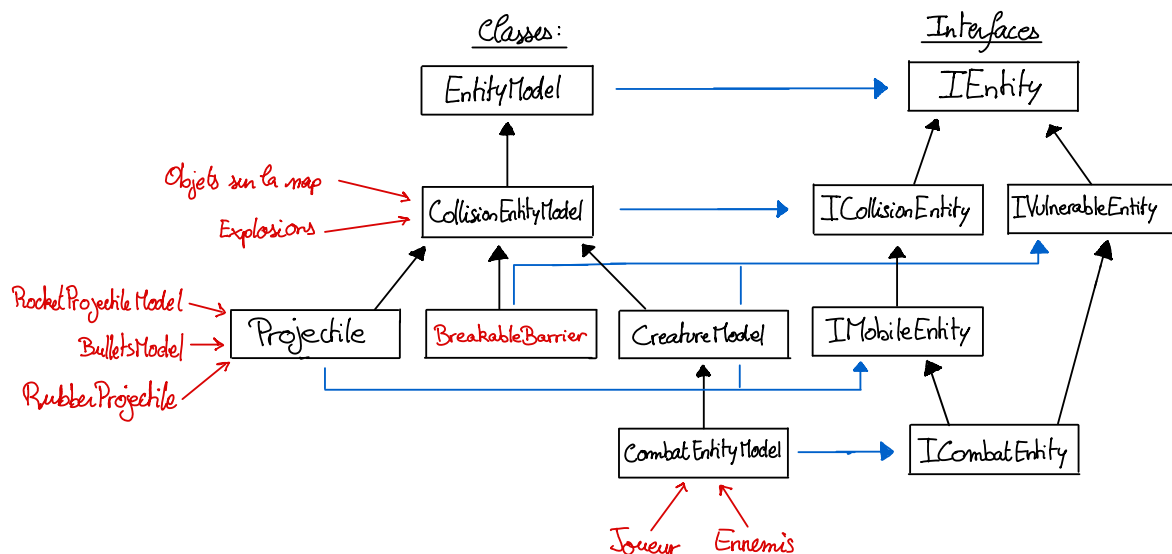


Schéma simplifié de la hiérarchie des entités. Le noir de présente les interfaces, les classes abstraites et les relations d'héritages entre elles. Le bleu de présente les relations d'implémentation. Le rouge représente les classes concrètes utilisées par le jeu et l'héritage de ces classes.

Mouvement et physique

Utilisation de la programmation orientée événements

Dans ce projet, nous avons implémenté la détection de collisions en utilisant la programmation orientée événements (event-driven). Cette approche nous permet de gérer les interactions entre les différents objets du jeu de manière efficace et réactive. Chaque `CollisionEntityModel` est équipée de détecteurs d'événements (listeners), et le moteur de physique se charge d'envoyer les événements lorsqu'une collision est détectée. Les collisions entre entités sont représentées par des `CollisionEvent` et celles déclenchées lorsqu'une tuile bloque un mouvement par `BlockedMovementEvent`. Ainsi, chaque entité obtient un pointeur vers l'autre entité impliquée dans la collision et peut choisir comment réagir (attaque, ramassage d'arme, restauration de points de vies...).

Implémentation en elle-même

Le mouvement est modélisé à l'aide du `movementHandler` des `IMobileEntity`. Celui-ci qui utilise les champs `speed` et `directionVector` pour déplacer l'entité dont in a charge. L'utilisation de la composition dans ce cas est judicieuse, ce qui est confirmé lors de l'implémentation des projectiles. Ces derniers doivent utiliser ce système de mouvement sans nécessairement faire partie des entités vivantes/vulnérables.

Le `movementHandler` délègue toute la physique (les collisions) au moteur de physique en lui faisant des demandes de déplacement via la méthode `PhysicsEngine.move`. Ce dernier gère le déclenchement des différents événements (collision, mouvement bloqué...) ainsi que leur transmission aux listeners de chaque entité. Cette approche garantit l'extensibilité et maximise la réutilisation du code.

Avoir un objet comme `RicochetListener` ou `DamageListener` que l'on peut ajouter à la volée au listeners de n'importe quelle entité (en combinaison avec d'autres listeners si besoin) est très approprié pour un jeu comme le nôtre où nous voulons que différentes entités aient des comportements et des physiques différentes (par exemple la `RubberBall` ou le `RocketProjectile`).

Remarque :

La détection de collisions a été énormément optimisée, passant d'une complexité de $O(n^2)$ à $O(k)$, où k représente le nombre d'entités autour la tuile concernée et n le nombre d'entités sur la carte. Tout cela grâce au stockage des `CollisionEntity` dans les tuiles, ce qui permet de vérifier uniquement les tuiles autour de l'entité concernée.

Implémentation des murs cassable

Nous avons envisagé d'utiliser l'objet `TileModel` pour représenter de telles tuiles. Cependant, Mohamed s'est rendu compte qu'il serait beaucoup plus judicieux d'avoir une entité vulnérable qui représente ces murs. Ce choix maximise la réutilisation de notre moteur physique et de la détection de collisions.

Cette implémentation implique également l'utilisation de `TileModel.canEnterConditions`, l'ensemble des prédicats que doit respecter une entité pour entrer sur une tuile. L'entité représentant le mur cassable se contente d'ajouter une condition d'entrée impossible sur la tuile où il se trouve et de la retirer quand il disparaît.

Gestion de l'IA

Utilisation du Flood Fill pour l'IA

Bien que nous ayons envisagé (et même implémenté) l'algorithme A* pour le pathfinding. Sur recommandations de notre chargée de projet, nous avons finalement décidé d'utiliser l'algorithme du `FloodFill`. Cet algorithme présente comme avantage principal ne calcule qu'une seule fois le chemin vers le joueur pour toutes les entités, contrairement à A* qui nécessite des calculs individuels et uniques à chaque entité. Les calculs du flood fill sont mis à disposition pour toutes les entités dans la classe `FloodFillPathFinder` pour qu'elle puisse les utiliser pour s'orienter sur la carte.

L'implémentation a été réalisée à l'aide d'un objet `Node` contenant une valeur `int` et des coordonnées représentant les emplacements où l'entité peut changer de direction, ainsi qu'un objet `NodeGrid`. La classe `FloodFillPathFinder` contient également des prédicats sur des coordonnées qui ont été utilisés pour éviter la superposition des entités. Ces prédicats peuvent théoriquement être utilisés pour bien d'autres choses, car ils sont vérifiés tout au long du calcul.



Affichage de debug montrant la valeur calculée par le flood fill pour chaque case de la grille.

Utilisation de l'algorithme de Bresenham modifié

Le comportement de `SmartEnemyModel` diffère légèrement des autres. Il suit le même algorithme de remplissage par diffusion que les autres ennemis jusqu'à ce qu'il détecte qu'il est entré dans la ligne de vue du joueur. Cette détection se fait à l'aide de l'algorithme de Bresenham modifié.

L'algorithme fournit une bonne approximation d'une ligne de vue idéale possible et ne prend donc pas en compte toutes les tuiles en contact avec cette ligne idéale. Cependant, il suffit de vérifier toutes les tuiles tangentes à cette ligne approximative.



Visualisation de la ligne de vue d'un ennemi intelligent ciblant le joueur

Modélisation des armes

Les entités de combats ont un attribut pouvant contenir un objet de type `WeaponModel`, représentant une arme. Pour placer les armes pouvant être ramassée sur la carte, nous avons simplement créé des entités spéciales contenant une arme, et ces entités réagissent aux collisions avec le joueur en lui donnant l'arme.

Les armes ont toutes une fonction d'action nommée `attack()`. Pour les armes dont le temps de recharge est non nul, elle `.start()` un `ModelTimer` qui empêchera toute nouvelle action tant qu'il ne sera pas terminé.

Les armes implémentant

`ProjectileWeaponModel` génèrent toutes des projectiles et doivent donc implémenter un `ProjectileSupplier` appelé `createProjectile()` ainsi qu'une méthode `fire()` qui va générer un projectile et le lancer.

Les armes implémentant

`ContinuousFireWeapon` appellent `attack()` lors de l'appui et de la relâche du clic gauche. Le premier appel donne au booléen `isFiring` la valeur `true` et lance la méthode `fire()`. Celle-ci va vérifier `isFiring` et dans ce cas générer un projectile, puis enclencher un `ModelTimer` qui appellera à `fire()` à nouveau lorsqu'il sera fini. Elles possèdent aussi une jauge de surchauffe qui empêche l'appel de `fire()` si trop pleine.

Utilisation des design patterns

Nous avons mis à profit toutes nos connaissances en programmation orientée objet en utilisant plusieurs design patterns.

Composition pattern

La composition a été privilégiée dans notre projet en raison des limitations imposées par l'héritage simple. L'utilisation de l'héritage dans la hiérarchie des entités a été soigneusement planifiée et complétée par l'utilisation d'interfaces. Par exemple, `CreatureEntity` a été défini à l'aide des deux interfaces `IVulnerable` et `IMobileEntity`. Cette séparation s'est avérée utile lorsque nous avons voulu définir des projectiles qui sont des entités mobiles mais non vulnérables.

Strategy pattern

Pour appliquer facilement des effets aux images des animations, nous avons décidé d'utiliser le design pattern Strategy en définissant l'interface fonctionnelle `ImageLoader`.

```
Anthony Fernandes, 2 hours ago | 3 authors (You and others)
1 package util;           You, yesterday • feat: better better graphics
2
3 import java.awt.Image;
4
5 Anthony Fernandes, 2 hours ago | 3 authors (Yacob Zitouni and others)
6 /**
7  * Interface for loading images.
8  */
9 @FunctionalInterface
10 public interface ImageLoader {
11     Image load(String path, Class<?> resourceBase);
12 }
```

Il suffit de passer à l' `AnimationManager` l' `ImageLoader` approprié à l'effet souhaité.

```
private ImageLoader reg = ImageCache::loadImage;
private ImageLoader neg = ImageCache::loadNegativeImage;
```

Factory pattern

Pour centraliser la création de différents objets, nous avons opté pour le design pattern Factory. La facilité d'utilisation et la flexibilité de ce pattern nous ont permis d'ajouter de nouvelles fonctionnalités très rapidement, notamment pour les `EntityRenderer`.

```

public static EntityRenderer make(IEntity entity) {
    return switch (entity) {
        // Player
        case PlayerModel e → new PlayerRenderer(e);

        // Enemies
        case WalkingEnemyModel e → new VulnerableSpriteRenderer(e, path:"sprites/EyeBallEnemy.png");
        case SmartEnemyModel e → new VulnerableAnimatedRenderer(e, path:"animations/smart_enemy.xml");
        case ExplodingEnemy e → new VulnerableAnimatedRenderer(e, path:"animations/exploding_enemy.xml");

        // Bullets
        case BulletsModel e → new SpriteRenderer(e, path:"sprites/weapon/bullet.png", e.getMovementHandler().getDirectionVector().getAngle());
        case RubberProjectile e → new SpriteRenderer(e, path:"sprites/weapon/rubberball.png");
        case RocketProjectileModel e → new AnimatedEntityRenderer(e, path:"animations/rocket_projectile.xml", e.getMovementHandler().getDirectionVector().getAngle());
        case BlackHoleProjectile e → new AnimatedEntityRenderer(e, path:"animations/blackhole.xml");

        // Damage zones
        case ExplosionZoneEntity e → new AnimatedEntityRenderer(e, path:"animations/explosion_zone.xml");
        case KnifeZoneEntity e → new AnimatedEntityRenderer(e, path:"animations/knife_zone.xml", e.getDirection().getAngle());
        case BlackHoleZone e → new EntityRenderer(e);
        case SimpleTrap e → new SpriteRenderer(e, path:"sprites/weapon/simple_trap_placer.png");

        // Mics entities
        case BreakableBarrier e → {
            Image full = ImageCache.loadImage(path:"sprites/breakablebarrier.png", resourceBase:PlayerRenderer.class);
            Image mid = ImageCache.loadImage(path:"sprites/breakablebarrier_2.png", resourceBase:PlayerRenderer.class);
            Image weak = ImageCache.loadImage(path:"sprites/breakablebarrier_3.png", resourceBase:PlayerRenderer.class);

            yield new ConditionalRenderer(e, Map.of(
                toRender → ((IVulnerableEntity) toRender).getHealth() ≥ 60, full,
                toRender → ((IVulnerableEntity) toRender).getHealth() ≥ 30, mid,
                toRender → true, weak
            ));
        }
        case FirstAidKit e → new SpriteRenderer(e, path:"sprites/firstaid.png");
        case WeaponEntity e → new SpriteRenderer(e, String.format("sprites/weapon/%s.png", e.getWeapon().getIdentifier()));
        default → throw new IllegalArgumentException("Unknown entity model: " + entity.getClass().getName());
    };
}

```

Observer pattern

Ce pattern a été employé en conjonction avec les événements, notamment avec les listeners. Par exemple, pour les `CollisionListener`, on les notifie au moment de la détection de collision. Une approche similaire a été adoptée pour les autres couples event-listener.

```

Set<ICollisionEntity> collidedEntities = getCollidedEntities(entity);
if (!collidedEntities.isEmpty()) {
    // create a collision event
    CollisionEvent event = new CollisionEvent(entity, collidedEntities);
    // notify the entity's collision listeners
    entity.notifyCollisionListeners(event);
}
}

```

Structure de la vue

Naturellement, nous avons séparé les composants servant aux menus de lancement du jeu, à l'éditeur de carte, et à l'affichage jeu en lui-même dans différents packages.

Look and feel

La customisation globale et en profondeur de l'apparence d'une application swing est une tâche surprenamment difficile. Le style d'une application swing est régi par le look and feel qui lui est appliqué. La première idée était donc de simplement utiliser le look and feel par défaut du système d'exploitation de la machine que laquelle est lancé le jeu, mais notre chargée de projet nous a conseillé de plus travailler le visuel.

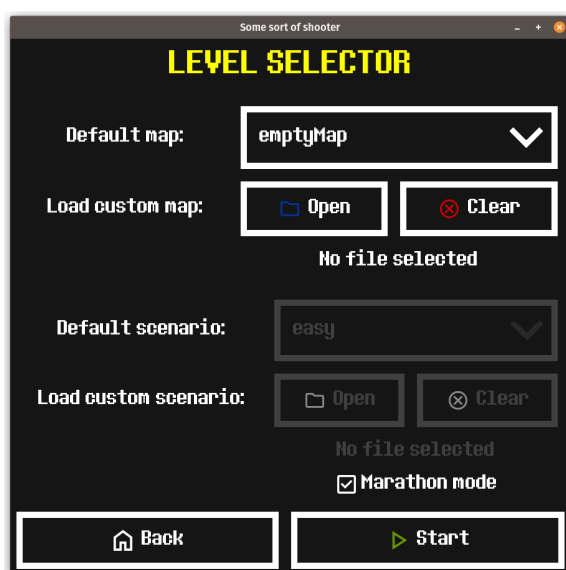
On a donc voulu modifier l'un des look and feel par défaut fourni avec le JDK mais aucun d'entre eux ne propose de personnalisation suffisamment en profondeur (comme par exemple changer les flèches des barres de défilement). Et il a fallu se rabattre sur la création d'un look and feel entièrement nouveau, customisé à notre goût.

Nous avons alors découvert que swing propose le look and feel Synth. Ce dernier ne fournit en réalité que le “feel” et c’est au programmeur de fournir le “look” à l’aide d’un fichier XML (cette fonctionnalité ne fois pas souvent être utilisée, au vu de l’absence presque totale de documentation à ce sujet). Le fichier XML pour notre projet est

`resources/gui/laf/lookAndFeel.xml`, il donne grâce aux éléments `<style>` et `<bind>` les différents styles à appliquer aux composants de l’interface.

Le style global adopté est un fond uni presque noir avec le texte en blanc et le jaune pour la couleur indiquant l’interactivité, ces couleurs sont spécifiées avec les éléments `<color>`.

Nous avons opté pour des boutons sans fond mais avec d’épaisses bordures dessinées avec des `<imagePainter>`. C’est également dans ce fichier que sont chargés les icônes pour le sélecteur de fichier, les flèches des liste déroulantes et les cases à cocher. Toutes ces icônes viennent de la collection Material Icons mise à disposition pour les développeurs par Google sur son site [Google Fonts](https://material.io/icons).



Menu de chargement d’une partie illustrant presque tous les composants utilisés dans le jeu



Sélecteur de fichier très customisé

Menus jetables

Les menus ont été conçus pour être des `JPanel` jetables, ils sont créés à la demande du `MainController` et immédiatement chargés comme `contentPane` de la `MainFrame` puis sont

détruits par le garbage collector à chaque fois qu'un nouveau menu est créé et affiché à sa place.

Les menus sont chargés à l'aide des fonctions `load...` de la classe `MainController`, ce sont ces fonctions qui sont enregistrées en tant qu'action listener sur la plupart des boutons des menus.

Le menu de lancement de partie est également muni de deux sélecteur de ressources permettant de changer les cartes et les scénarios. Le menu de fin affiche les statistiques dans un `StatsPanel` dédié pour un alignement plus facile et une potentielle réutilisation du composant dans un menu qui afficherait l'historique des statistiques.

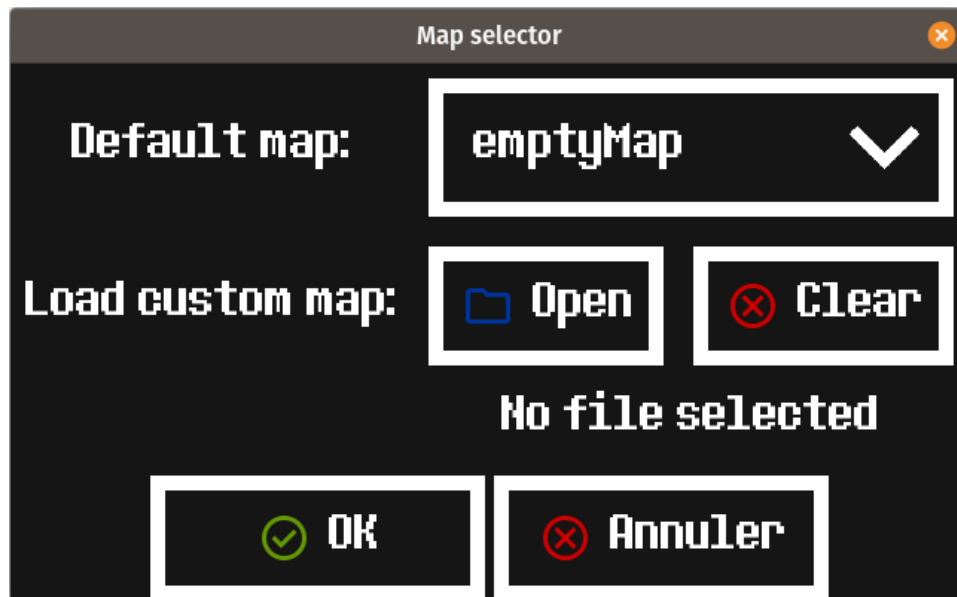
Sélecteur de ressource

Le sélecteur de ressource est un composant particulièrement important de ce projet, il permet au joueur de charger une `Resource` dans le jeu. Comme expliqué dans la partie concernant le modèle, il y a deux types de ressources et le sélecteur de ressource permet de choisir les deux types de ressources.

- Les ressources internes sont proposées au joueur via un menu déroulant dont les valeurs sont remplies à l'aire de l'index passé en argument au constructeur du sélecteur. Par exemple le fichier `resources/gui/launcher/mapIndex` contient la liste de toutes les cartes internes au jeu proposées via le menu déroulant.
- Les ressources externes sont lues à l'aide d'un `JFileChooser`, un composant par défaut de swing mais fortement customisé d'un point de vue visuel. Une fois qu'un fichier est changé, il est prioritaire sur la déroulante qui passe en mode désactivé. Le bouton clear permet de supprimer le fichier sélectionné et de redonner la main à la liste déroulante.

Quand le joueur clique sur le bouton OK, le menu renvoie une `Ressource` correspondant au fichier sélectionné et le programme reprend son exécution.

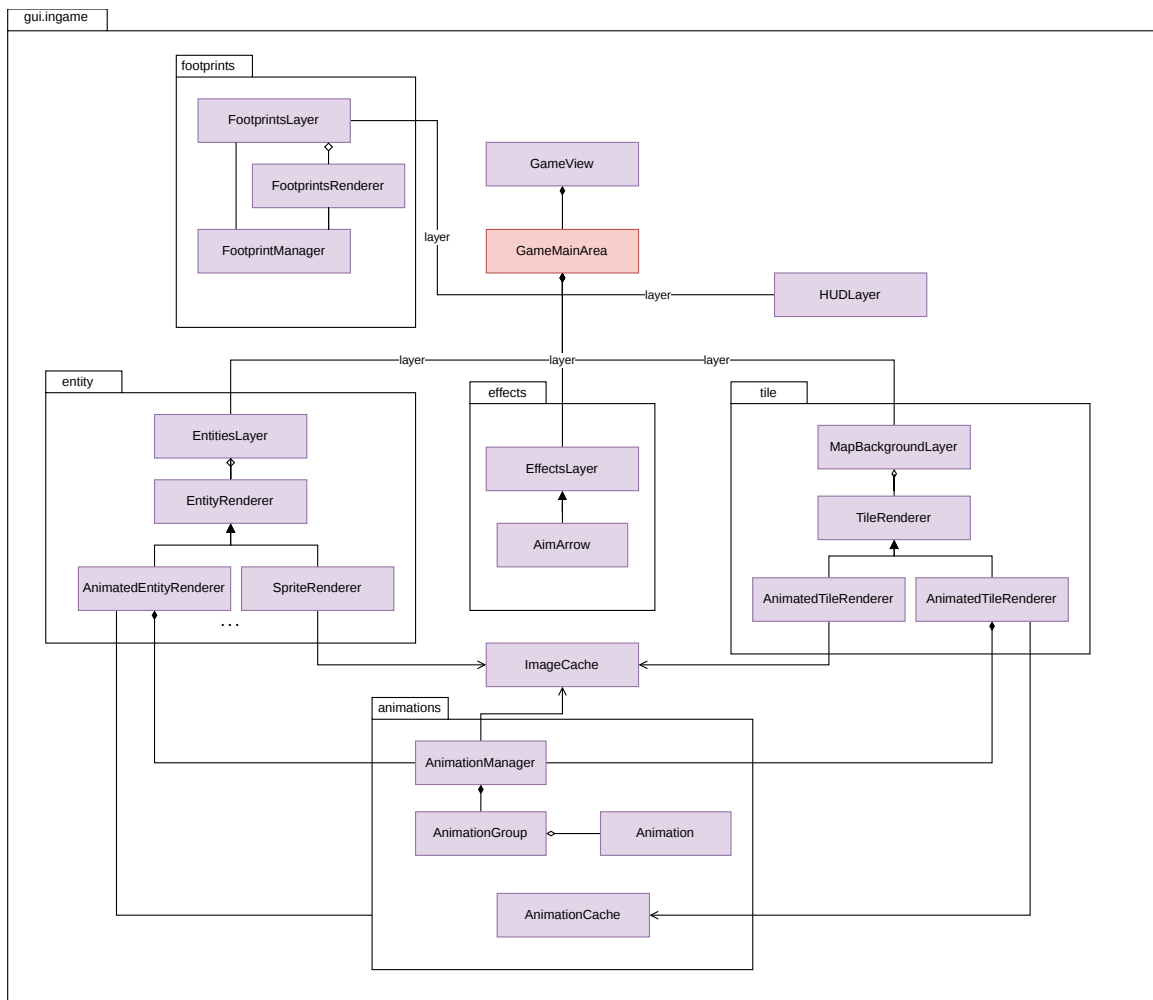
Le sélecteur de scénario sous classe le sélecteur de ressource pour y ajouter une case à cocher permettant de choisir le mode marathon. Cette case est prioritaire sur la liste et sur le fichier.



Le sélecteur de ressource dans sa propre fenêtre utilisé par l'éditeur de carte

Vue en jeu

- ▼ Diagramme des **principales classes** de `gui.ingame`. Toutes les classes ne sont pas incluses, par soucis de lisibilité, seules celles dont nous avons jugé la présence dans le diagramme bénéfique pour la compréhension ont été incluses.



Le composant maître de la vue en jeu est dans la classe `GameView`. Il se charge de redimensionner la zone d’affichage du jeu selon la taille de la fenêtre tout en conservant ses proportions grâce à un `LayoutManager` que nous vont nous même conçu, le `RatioLayout`. C’est aussi ce composant qui se charge d’instancier un contrôleur pour les entrées clavier/souris.

La zone d’affichage à proprement parler est gérée par `GameMainArea`, Plutôt que de placer tous les composants du jeu sur un seul panel, nous avons préféré placer chaque catégorie (fond de la carte, traces de pas, entités, effets, HUD) sur un panel différent et les superposer grâce à un `JLayeredPane`. Cela permet non seulement de mieux séparer les composants, mais aussi de contrôler leur superposition facilement. Par exemple la flèche de visée s’affichera toujours par-dessus les ennemis.

Le fond de carte est géré par `MapBackgroundLayer`. Il utilise simplement un `GridLayout` pour disposer les tuiles, affichées par des `TileRenderer`.

Le layer le plus important est l’ `EntitiesLayer`. Il a accès aux `Set` du modèle comportant l’ensemble des entités. À chaque mise à jour de l’affichage, il va se charger de créer et supprimer des `EntityRenderer` pour refléter les changements du modèle. Chaque renderer se

charge d'afficher une et une seule entité, en réagissant aux changements de l'entité si besoin (comme les points de vie).

Animations

Concernant les animations, nous avons un système à part entière consistant en un `AnimationManager` qui reçoit un groupe d' `Animation` , chaque animation consistant en une association entre un intervalle de temps et un chemin vers une image. Elles utilisent donc les `IntervalMap` , tout comme les scénarios. L' `AnimationManager` permet de facilement récupérer l'image à afficher à un moment donné, et permet aussi de changer d'animation. Le stockage des animations se fait dans des fichiers XML, présents dans `resources/gui/ingame/<catégorie>/animation` . Ceci facilite énormément la modification d'animations sans avoir à modifier le code, quelque chose qui serait primordial dans la cadre d'un projet géré par plusieurs équipes avec des compétences différentes. Ils sont chargés par la classe `AnimationCache` , qui se charge aussi de les stocker en mémoire pour ne pas avoir à les relire dans le système de fichier lorsque nécessaire. Les `EntityRenderer` et `TileRenderer` peuvent alors instancier un `AnimationManager` en lui fournissant les animations demandées, et elles n'auront plus qu'à : 1. le mettre à jour à chaque tick 2. récupérer l'image à afficher 3. demander un changement d'animation.

```
<animationGroup default="walk_down">
  <animation id="idle_up" endReachedBehaviour="INFINITE">
    <frame time="0.1" path="sprites/player1/up/up1.png"/>
  </animation>
  <animation id="walk_up" endReachedBehaviour="LOOPING">
    <frame time="0.1" path="sprites/player1/up/up1.png"/>
    <frame time="0.2" path="sprites/player1/up/up2.png"/>
    <frame time="0.3" path="sprites/player1/up/up3.png"/>
    <frame time="0.4" path="sprites/player1/up/up4.png"/>
    <frame time="0.5" path="sprites/player1/up/up5.png"/>
    <frame time="0.6" path="sprites/player1/up/up6.png"/>
    <frame time="0.7" path="sprites/player1/up/up7.png"/>
    <frame time="0.8" path="sprites/player1/up/up8.png"/>
    <frame time="0.9" path="sprites/player1/up/up9.png"/>
    <frame time="1.0" path="sprites/player1/up/up10.png"/>
  </animation>
</animationGroup>
```

Éditeur de cartes

Comme le jeu, le modèle est divisé en un modèle et une vue.

Modèle de l'éditeur

En interne, la carte en cours d'édition est représentée simultanément de deux manières différentes. D'une part comme un tableau de caractères qui sera utilisé pour écrire la carte, et d'autre part par un tableau de `TileModel` et un tableau de `IEntity` utilisés pour l'affichage.

La synchronisation entre les deux représentations est assurée par la fonction `updateSquare` qui met simultanément à jour le caractère et les deux tableaux. La conversion de caractère à objets réutilise les fonctions de `MapModel` pour avoir le minimum de changements à faire en cas d'ajout de nouvelle tuile.

Le changement de contenu d'une tuile se fait grâce aux fonctions `nextType` et `prevType` qui font défiler les caractères dans l'ordre suivant :

1. Standard
2. Eau
3. Vide
4. Mur cassable
5. Protégée

Le point d'apparition du joueur, unique dans une carte, doit être placé à part par la fonction `setSpawn` qui s'assure qu'il existe un unique spawn à chaque instant.

La lecture de la carte se fait via la fonction `readFile` très similaire au constructeur de `MapModel`. Une fois la carte terminée, elle est écrite dans le fichier spécifié par le joueur à l'aide d'un `PrintWriter` et des fonctions de formatage classiques, `printf` et `println`.

Vue de l'éditeur

La vue de l'éditeur est composée de deux parties, le menu et la grille.

Le menu permet de choisir la taille de la carte à éditer en utilisant les deux spin box en haut de l'écran. Les boutons en bas de l'écran appellent les fonctions du modèle mentionnées plus haut pour nettoyer la grille, ouvrir un fichier ou sauvegarde la carte.

La grille est chargée d'afficher la carte actuelle. Pour se faire elle dispose des `TileRenderer` dans un `GridLayout` à la façon du `MapBackgroundLayer` mais avec un peu plus d'espace entre les tuiles pour pouvoir bien les distinguer. C'est également à la charge de la grille de gérer

l'interactivité de l'éditeur. Pour se faire, des `MouseListener`s sont ajoutés à chaque tuile qui faire passer le prochain type avec un clic droit, le précédent avec un clic gauche et placer le point d'apparition du joueur avec un clic molette.



L'éditeur de carte avec la carte "ruins" chargée

Difficultés rencontrées

- Dès la première semaine, nous avons constaté un changement significatif par rapport au projet du premier semestre : les réunions étaient désormais espacées d'une semaine plutôt que deux. Cela nous a obligés à adopter un rythme de travail un peu plus régulier. Bien que nous ayons rencontré quelques difficultés au début, nous avons progressivement réussi à nous adapter. Nous avons tout de même été ralentis sur certaines semaines

lorsque la charge de travail combinée des autres matières d'informatique et de licence de maths devenait élevée.

- Lors de l'élaboration du modèle, nous avons exploré plusieurs idées. Au début, nous souhaitions maintenir le concept d'encapsulation au maximum et éviter les "God Objects" en veillant à ce que le modèle de jeu ne soit pas accessible partout. Par exemple, les `ModelTimer` devaient être mises à jour par les composants qui les créaient, plutôt qu'être directement mis à jour par le `GameModel`. Cependant, cela entraînait une surcharge de travail importante. De même, nous nous sommes parfois trop concentrés sur des optimisations inutiles. Sur les recommandations de notre chargée de projet, nous avons donc décidé de simplifier le modèle.
- La création des graphismes a été assez ardue. En effet, on nous avait fait comprendre au cours du projet que les graphismes n'étaient pas essentiels pour le résultat final. Cependant, lors des dernières séances, on nous a fait remarquer que le rendu était peu esthétique. Nous avons donc dû refaire intégralement les graphismes très rapidement pour les améliorer et aussi démontrer la facilité avec laquelle on peut les modifier avec le chargement d'animations à partir des XML. Cela n'a pas été évident sans utiliser des animations déjà existantes sur Internet simplement en raison de notre faible expérience artistique.