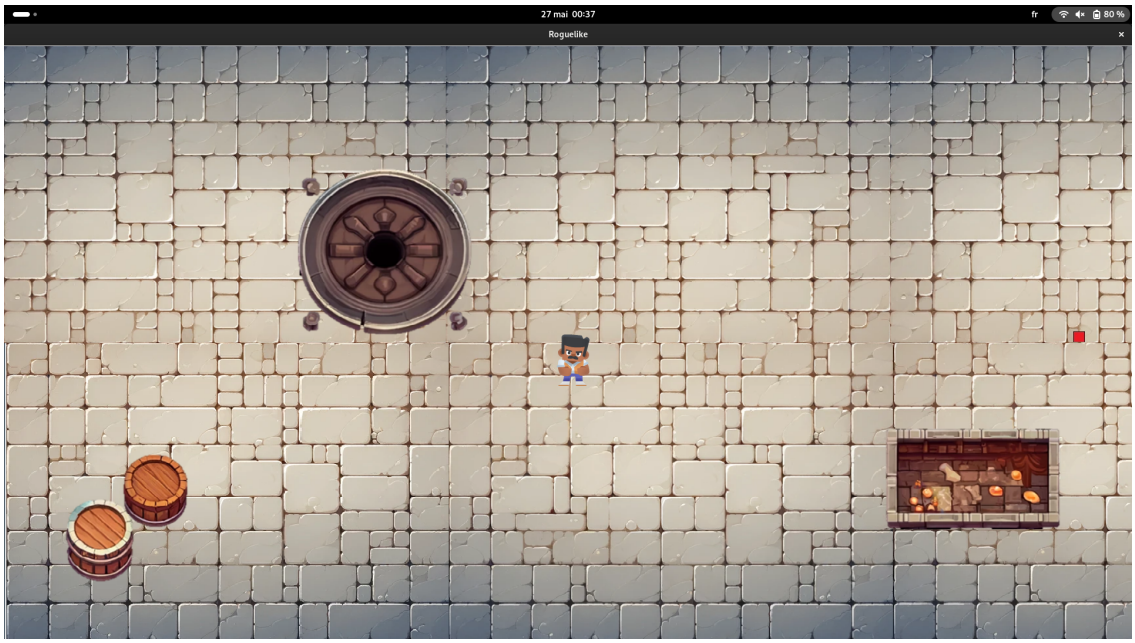
Rapport général : Roguelike



Etudiants :
Beauhorry-sassus Nicolai
Chraibi Driss
Dissoubray Alice
El Omari Marwa
Joma'a Ainji
Pointeau Quentin
Moussu Eloi

Superviseurs :
Xavier Cregut
Olivier Duffort

Table des matières

1 Contexte

L'objectif de ce projet consiste à concevoir un jeu vidéo d'exploration de donjons à plusieurs niveaux générés de manière procédurale en utilisant le langage Java et la librairie LibGDX. Le jeu comporte aussi une partie qui concerne le combat entre le joueur et un monstre.

2 Les principales fonctionnalités

Voici les principales fonctionnalités accessibles à l'utilisateur :

— Lancer une partie : finalisé à la première itération.

— Se déplacer dans une salle : finalisé à la deuxième itération.

— Faire un combat avec un monstre : finalisé à la deuxième itération.

— Naviger des menus (menu général et menu de pause) : finalisé à la troisième itération.

— Passer d'un étage à un autre : finalisé à la troisième itération.

— Utiliser un inventaire : commencé à la troisième itération.

— Utiliser des objets consommables : commencé à la troisième itération.

3 Le découpage de l'application en sous-sytèmes

La classe PlayState est la classe qui relie toutes les pièces du projet. Dans celle-ci on implémente le déplacement du personnage ainsi que la gestion des collisions avec les objets présents dans les salles. De plus, on fait appel à l'algorithme de géneration procédurale qui a pour but de créer les salles d'un étage (et leurs portes) aléatoirement. Aussi, cette classe permet le passage d'une salle à une autre lorsque le personnage s'approche d'une porte, la création et le passage à un étage suivant lorsque le personnage traverse le portail, et finalement, le déclenchement du combat avec le monstre lorsque le joueur s'approche trop près de celui-ci.

Afin de pouvoir réaliser ces fonctionnalités, notre projet a été décomposé en plusieurs sous-parties :

— Un package *personnage* a été créé afin de regrouper une partie des actions relatives aux personnages. Dans ce paquetage nous pouvons gérer la position des personnages (joueur, monstre...), la modification des points de vie ainsi que l'inventaire et les objets qui y figurent.

— Un package *portes* a été créé pour regrouper toutes les portes implémentées. Ce dernier est indispensable pour relier les salles générées entre elles. En effet, il permet d'obtenir les positions des portes générées par l'algorithme de génération procédurale ainsi que les salles qui leurs sont reliées.

— Un package *salle* a été créé afin de regrouper les différents types de salles implémentés : la salle de départ, la salle de monstre, la salle représentant la salle de l'étage suivant...

— Un package *states* qui regroupe tout ce qui est relatif aux menus et au lancement du jeu.

— Les classes *Fight* et *MissileState* qui permettent de gérer le combat entre le joueur et le monstre lorsque celui-ci est déclenché.

— L'implantation de la génération procédurale a nécéssité l'implantation de différents packages afin de pouvoir manipuler "numériquement" les entités étage, salle, porte, case et les couples (case source, case libre) qui est implanté par la classe Position.

4 Les diagrammes de classe

Voici le diagramme de classe illustrant l'architecture de notre jeu :

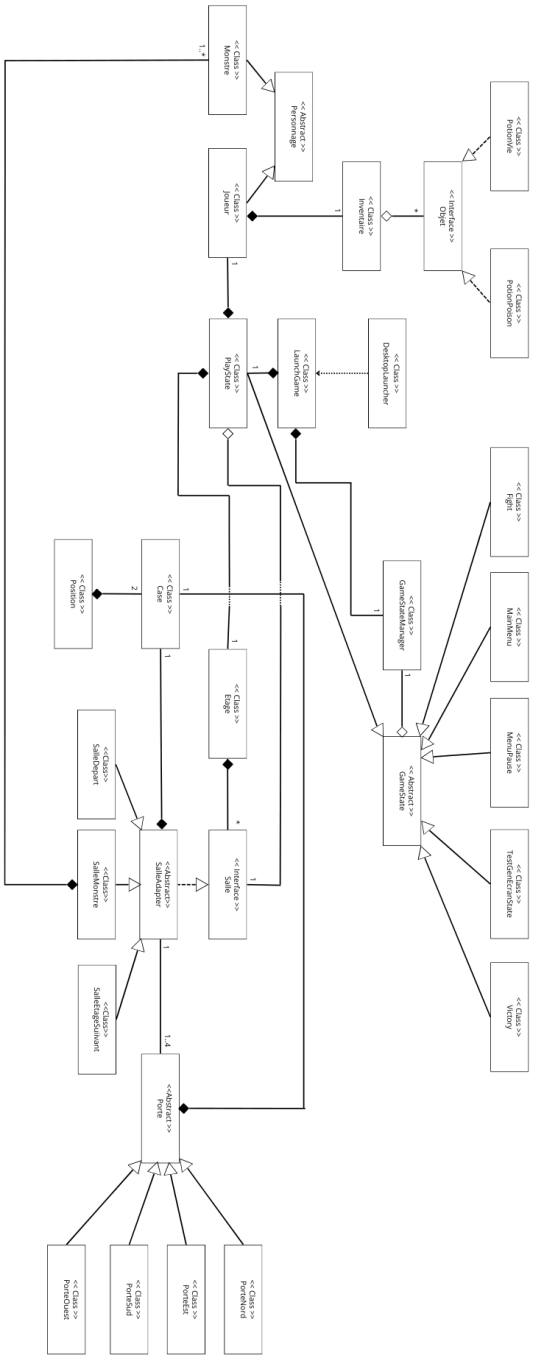


Figure 1 – *Diagramme de classe général*

5 Les principaux choix de conception et réalisation

Choix de la conception de la librairie :

Premièrement, nous avons fait le choix d'utiliser la librairie LibGDX du au fait qu'elle est adaptée aux jeux.

Le jeu commence par la classe LaunchGame, qui commence par créer un premier étage et un personnage qui sera le même tout au long du jeu. Ensuite, la classe jeu lance la classe PlayState qui a pour but de Gérer l'affichage principal du jeu et les interactions avec l'utilisateur ainsi que les mouvements et les collisions du joueur avec les objets et les ennemis. Ce personnage ainsi que l'étage correspondant sont donnés en paramètres.

Choix de conception pour la classe PlayState :

— **Utilisation de `Stage` et `SpriteBatch`** : Utilisation de `Stage` pour gérer les événements d'entrée et de `SpriteBatch` pour dessiner les éléments graphiques, ce qui est courant dans LibGDX pour optimiser le rendu graphique.

— **Gestion des textures** : Les textures sont chargées dans la méthode `init()` et libérées dans la méthode `dispose()` pour gérer efficacement les ressources.

— **Gestion des collisions avec des `Rectangle`** : Utilisation de rectangles pour détecter les collisions entre le joueur et les obstacles, ce qui est une méthode simple et efficace.

— **Déplacement du joueur** : Le déplacement du joueur est basé sur l'entrée utilisateur (touches Q, A, Z, D) et est ajusté en fonction du delta time pour une fluidité de mouvement.

— **Gestion des transitions entre salles** : La classe gère la transition entre les différentes salles et les étages en fonction de la position du joueur par rapport aux portes et aux portails.

Nous pouvons remarquer dans PlayState l'appel à la classe Fight, qui a pour but de gérer les interactions entre le joueur et les monstres pendant le combat tout en affichant la barre de vie du monstre.

Choix de conception pour la classe Fight :

— **Gestion des points de vie** : Affichage de la barre de vie du joueur à l'aide de textures représentant les points de vie (PV).

— **Gestion des missiles** : Les missiles sont créés lorsque l'utilisateur appuie sur la barre d'espace, ajoutés à une liste de missiles, et mis à jour dans la méthode `render()`.

— **Détection des collisions** : Les collisions entre les missiles et les monstres sont détectées pour mettre à jour les points de vie du monstre lorsqu'un missile le touche.

— **Transitions d'écran** : Transition vers un écran de victoire lorsque les points de vie du monstre atteignent zéro.

Les missiles sont gérés par la classe MissileState, qui a pour rôle de représenter un missile tiré par le joueur et de gérer le mouvement du missile et sa suppression lorsqu'il sort de l'écran.

Choix de conception pour la classe MissileState :

— **Constantes et variables** : Utilisation de constantes pour la vitesse du missile et variables pour sa position.

— **Méthodes de mise à jour et de rendu** : Méthodes `maj()` pour mettre à jour la position du missile et `render()` pour le dessiner à l'écran.

— **Gestion de la vie du missile** : Le missile est marqué pour suppression (`enlever`) lorsqu'il sort de l'écran.

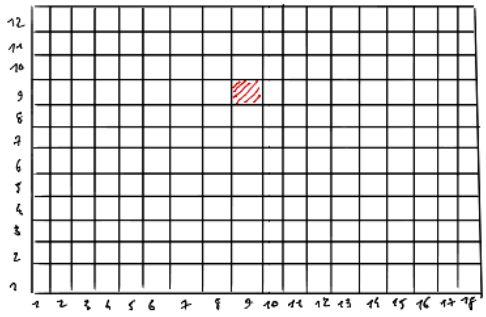
Choix de la conception pour le modèle de l'étage :

Le modèle globale de l'étage sur lequel nous étions partis était beaucoup trop complexe ce qui rendait la tâche de relier les portes des salles entre elles, et ce de manière cohérente, presque impossible. C'est suite à une seconde séance de brainstorming que nous avons mis au point un modèle d'étage plus simple ainsi qu'une première version du raffinage finale. Ainsi, en ce qui concerne le modèle de l'étage, nous avons choisi de le visualiser comme étant une grille géante où chaque case représente un emplacement envisageable pour y mettre une salle.

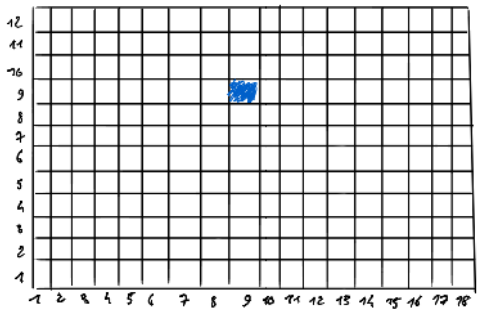
A partir de cet instant, nous avons implanté la première version du raffinage de l'algorithme de génération procédurale puis après avoir aporté de légères modifications durant l'implantation, nous avons concu l'algorithme suivant :

- **R0 :** Génerer de manière procédurale un étage
- **R1 :** Comment **R0** ?
  - **Choisir** une case de départ
  - **Construire** la salle de départ
  - **Choisir** la case suivante
  - **Initialiser** le premier couple (case source, case libre)
  - **TANT QUE** le nombre de salles est inférieure au nombre de salles maximum dans l'étage **FAIRE**
    - **Tirer** un couple (case source, case libre)
    - **Choisir** une nouvelle salle
    - **Relier** la nouvelle salle avec la salle source
    - **Creer** de nouveau couple (case source, case libre) à partir des cases adjacentes de la nouvelle salle
  - **FIN TANT QUE**
  - **Placer** la salle étage suivant

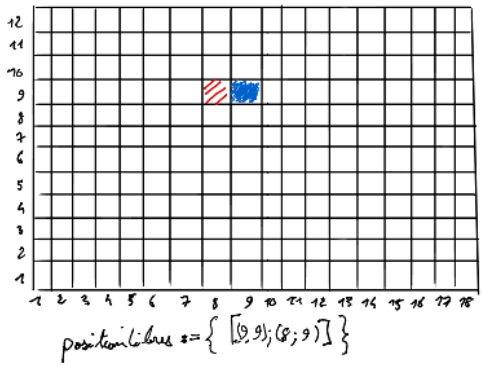
Illustrons cet algorithme dans le cas où le nombre maximum de salles de l'étage est 4



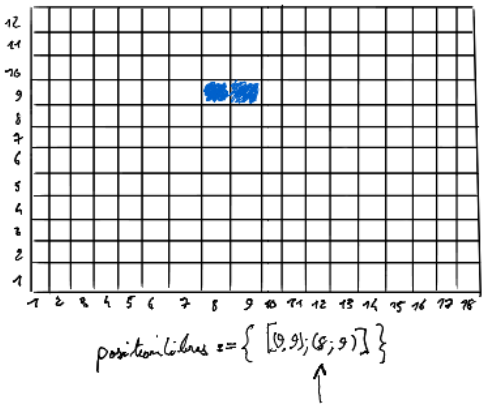
(a) *Illustration de la première action complexe du raffinage*



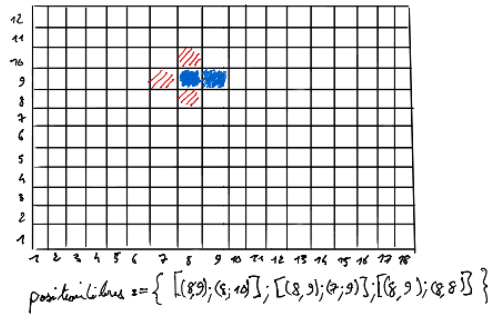
(b) *Illustration de la seconde action complexe du raffinage*



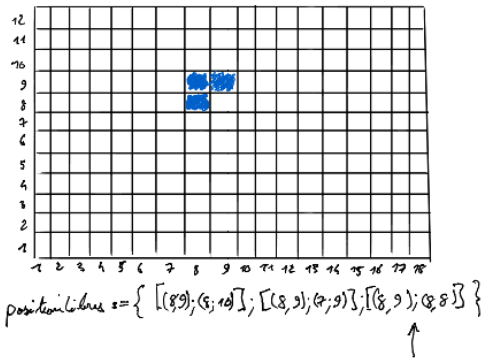
(c) *Illustration de la troisième et quatrième action complexe du raffinage*



(d) *Illustration des trois premières actions complexes du raffinage pour la première itération de la boule TANT QUE*

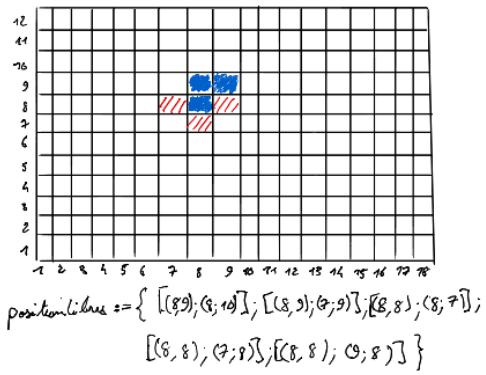


(e) *Illustration de la quatrième action complexe du raffinage pour la première itération de la boule TANT QUE*

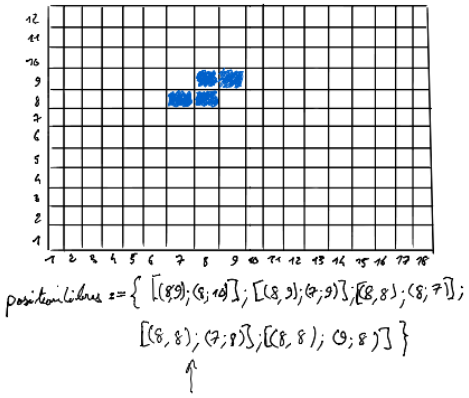


(f) *Illustration des trois premières actions complexes du raffinage pour la deuxième itération de la boule TANT QUE*

Figure 2 – *Illustration de l'algorithme de génération procédurale pour quatre salles dans l'étage (première partie)*



(a) *Illustration de la quatrième action complexe du raffinage pour la deuxième itération de la boucle TANT QUE*



(b) *Illustration de la dernière action complexe du raffinage*

FIGURE 3 – *Illustration de l'algorithme de génération procédurale pour quatre salles dans l'étage (deuxième partie)*

Choix de la conception pour le packetage *personnage* :

En ce qui concerne le packetage *personnage*, le choix a été fait de créer une classe abstaite Personnage, qui permet de regrouper toutes les actions utiles et communes aux classes Joueur et Monstre, qui en sont donc des sous-classes. Par exemple, l'obtention et la modification des coordonnées, des textures ou des points de vie sont implémentés par des méthodes au sein de la classe abstraite Personnage. Les sous-classes Joueur et Monstre implémentent donc des actions plus spécifiques à leur classe comme la gestion de l'inventaire pour la classe Joueur ou des constructeurs plus spécifiques pour les deux sous-classes.

Choix de la conception de l'interface Objet :

Nous avons aussi choisi de créer l'interface Objet qui réalisera tous les objets, soit consommables, soit à effets passifs. Nous avons fait deux réalisations de cette interface pour des objets consommables : un objet PotionDeVie qui permettera au joueur d'augmenter ses points de vie d'une somme pré-définie, ainsi qu'un objet PotionPoison, qui aura l'effet inverse pour le monstre.

Choix de la conception de l'interface Menu :

De même que pour les objets, nous avons fait le choix de créer une interface Menu qui réalisera tous les menus du jeu (menu principal, menu de pause, menu Victoire...).

Choix de la conception de l'invertaire :

Enfin, pour ce qui est de l'inventaire, le choix a été fait (par manque de temps et donc pour un souci de facilité) d'implanter la gestion des stocks par l'utilisation d'une HashMap. Ainsi, l'ajout et le retrait d'objets dans l'inventaire se fait par la gestion de cette HashMap. Malheuresement, nous n'avons pas eu le temps d'incorporer la gestion de l'inventaire dans notre projet. Ainsi, le code de la classe Inventaire n'est pas utilisé dans notre projet.

6 L'organisation de l'équipe

Nous avons organisé plusieurs réunions afin de se répartir les tâches et de garder l'équipe au courant du progrès de chacun d'entre nous. Nous avons aussi utilisé un Trello afin de suivre l'avancement de notre projet de façon continue.

Voici la répartition finale du projet :

— Nicolai et Quentin se sont occupés de tout ce qui touche à la conception et l'implémentation de l'algorithme de génération procédurale.

— Ainji s'est chargé de concevoir et d'implémenter le déplacement du joueur dans la salle grâce aux touches du clavier, ainsi que de la gestion des collisions. Elle s'est également occupée de la création de l'interface de combat entre le joueur et le monstre (envoie de missile grâce à la touche espace et barre de vie du monstre).

— Quentin et Marwa se sont occupés de concevoir et d'implémenter la classe abstraite Personnage ainsi que les sous-classe Joueur et Monstre, et la classe Inventaire, qui n'a pas pu être achevée.

— Nicolai, (avec l'aide d'Ainji) s'est chargé de concevoir et d'écrire le code qui a permis de relier les salles entre elles et de passer à l'étage suivant.

— Marwa s'est occupée de la conception et l'implémentation de l'interface Objet ainsi que ses réalisations.

— Driss s'est occupé de l'implémentation et de l'affichage des menus relatifs au jeu.