

Rapport de projet

# RogueLike

Projet Programmation Orientée Objet PRINTEMPS 2021

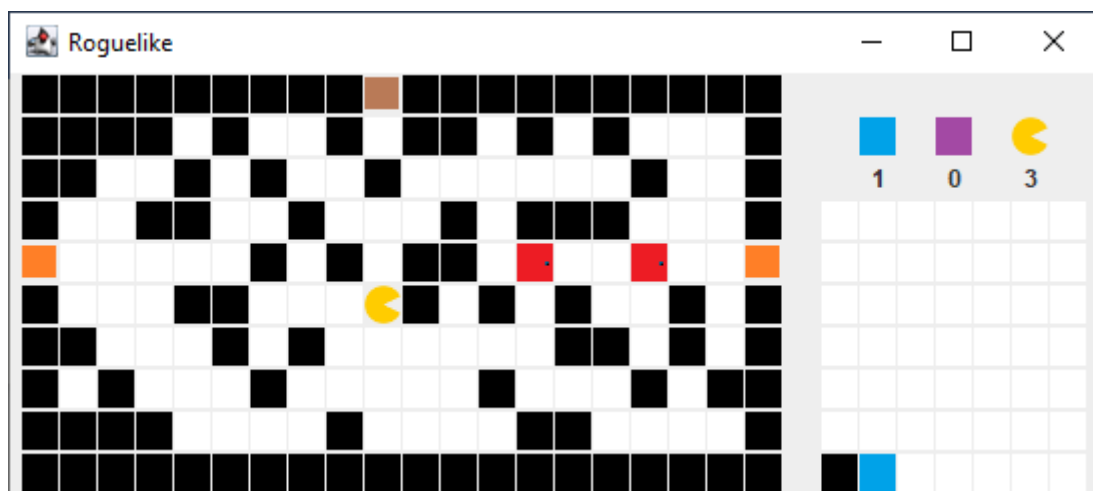
Fourkane SAID ALI | Alexandre DURY  
11802776 | 11919722

# Sommaire









Introduction	2
Fonctionnalités communes	2
Inventaire	2
Portes	2
Clés et Capsules	3
Coffres	3
Cases Normales, Murs, Dalles et vides	3
Salle	3
Niveau	4
Héros	4
Interaction avec les objets	5
Jeu	5
Extensions	5
Le scrolling	5
Affichage de la mini-carte et de l'inventaire	5
Le saut avec contraintes	6
Fin de partie	6
Victoire	6
Défaite	7
Conclusion	7

## Introduction

Ce projet a été réalisé par Alexandre DURY et Fourkane SAID ALI. Nous avons réalisé quasiment la totalité du projet ensemble. L'un d'entre nous était en partage d'écran et nous avons implémenté chacune des classes ensemble. Lorsque nous lançons une partie, un niveau composé de salles est généré aléatoirement. L'objectif est de trouver la salle finale. Chaque salle est générée aléatoirement et le joueur ne voit que la salle dans laquelle il se trouve. Il a également accès à un inventaire où les carrés bleus représentent les capsules d'eau, les carrés violets représentent le nombre de clés et l'icône du héros représente le nombre de saut qu'il peut réaliser dans la salle actuelle. Les portes sont représentées comme des carrés marrons s'ils ne sont pas verrouillés et oranges s'ils sont verrouillés et les dalles sont représentées comme des carrés rouges. Au-dessous de l'inventaire, le joueur a également accès à une mini-carte qui représente la salle où se trouve le joueur en bleu et les salles qu'il a visitées en noir.



Interface de jeu

 Capsule d'eau / Salle actuelle (mini-carte)	 Coffre	 Joueur / Nombre de sauts restants (inventaire)	 Porte déverrouillée
 Clé	 Dalle	 Mur / Vide / Salle visitée (mini-carte)	 Porte verrouillée

Légende

## Fonctionnalités communes

Nous avons réalisé l'ensemble des fonctions communes tout en respectant le modèle MVC proposé.

### Inventaire

Nous avons décidé de réaliser une classe afin de représenter l'inventaire. Cette classe contient un entier représentant le nombre de clés et un autre contenant le nombre de capsules ainsi que des accesseurs de ces données. Nous avons également implémenté des fonctions afin d'ajouter ou supprimer un nombre de clés ou de capsules. Un héros possède une instance de la classe Inventaire.

### Portes

Notre classe Porte hérite de la classe EntiteStatique. Elle possède un booléen qui permet de différencier les portes verrouillées des portes non verrouillées. Elle possède également un constructeur qui verrouille aléatoirement une porte avec une probabilité de succès de 25%. Elle possède également une fonction membre qui permet de déverrouiller une porte. Une porte n'est traversable que si elle n'est pas verrouillée.

## Clés et Capsules

Les clés et les capsules d'eau sont toutes les deux des classes héritant d'EntiteStatique et traversables. Elles n'ont aucun autre attribut. Lorsque le héros se trouve dans la même case qu'une clé ou une capsule, cette case devient une case normale et l'inventaire est modifié afin d'ajouter l'objet ramassé dans l'inventaire.

## Coffres

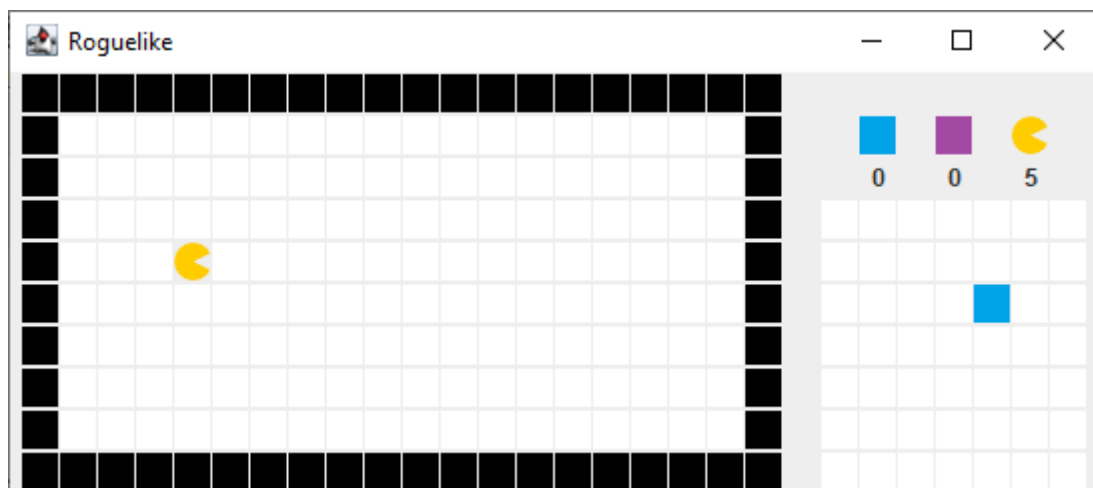
Nous avons implémenté une classe Coffre héritant de classe EntiteStatique et contenant un entier représentant le nombre de clés qu'il contient et un autre entier contenant le nombre de capsules. La classe contient également une fonction d'initialisation qui initialise entre 0 et 2 capsules et clés. Dans le même principe que pour les clés et les capsules, les coffres sont traversables, et une fois que le héros se trouve dans la même case qu'un coffre, son contenu est ajouté à l'inventaire et le coffre disparaît.

## Cases Normales, Murs, Dalles et vides

Nous avons une classe CaseNormale, une classe Dalle et une classe Mur. Elles héritent toutes de la classe EntiteStatiques. Les cases normales sont traversables tandis que les murs et les dalles ne sont pas traversables. Nous avons décidé de ne pas créer de classe pour le vide, mais d'utiliser à la place un Mur puisqu'ils ont les mêmes caractéristiques et nous les avons représentés de la même manière. Lorsque nous utilisons une capsule d'eau sur une dalle, en étant orienté vers la dalle et en pressant la touche « e », la dalle devient une case normale, et est alors traversable.

## Salle

Dans notre classe salle, qui représente une salle, nous avons deux variables représentant la dimension en hauteur et en largeur d'une salle. La salle contient également une instance du jeu actuelle, un tableau 2D d'entités statiques qui permet de représenter ce que contient chacune des cases du jeu. Nous avons également implémenté un booléen estVisite qui permet de définir si nous avons visités une salle. Cette variable, initialement à false, sert pour l'affichage de la mini-carte. Toutes les salles sont initialisées comme étant des salles ne contenant que des cases normales avec des murs aux extrémités.



Une salle initialisée

Si la salle dans laquelle le joueur se trouve n'est ni la salle de départ, ni la salle finale, la fonction générationSalle est appelée. Cette fonction permet de générer aléatoirement une salle en plaçant aléatoirement un certain nombre de murs représentant le vide. Nous avons une variable nbMurs

contenant le nombre de cases vides à initialiser. L'algorithme de génération prend un emplacement aléatoire, et si cet emplacement n'est pas déjà un mur, il place un mur. Cet algorithme permettait de générer aléatoirement des salles cependant, nous pouvions nous retrouver avec des salles où il nous était impossible d'atteindre certaines portes et il était également possible de se retrouver avec un regroupement de murs qui nous empêchait d'atteindre certains objets. Pour cela, nous avons rajouté une fonction membre qui calcule le nombre de cases voisines d'un emplacement n'étant pas une case normale et si ce nombre dépassait 3, on ne générerait pas de murs. Autrement dit, un mur ne peut être généré que si la case est une case normale et qu'elle n'est pas entourée de murs. Par la suite, nous plaçons aléatoirement les ramassables et les dalles et une fois que toutes les salles ont été créées, nous générons les portes.



Une salle générée

## Niveau

Notre classe Niveau représente l'ensemble d'un étage de jeu. Il contient les dimensions x et y du niveau, un entier représentant le nombre de salles à générer, un tableau 2D de Salle et un autre tableau 2D d'entiers qui sert à représenter les emplacements des salles ainsi que le type de salle. Niveau possède également une instance de jeu. Tout comme pour les salles, nous avons implémenté une génération de niveaux. Pour cela, nous nous servons du tableau 2D de salles qui initialise toutes les valeurs à 0. Ensuite, nous plaçons aléatoirement un emplacement dans ce tableau d'entier. Nous lui attribuons la valeur 1 qui correspond à la salle de départ. Ensuite, nous créons des salles aléatoirement si elles n'ont qu'une seule salle voisine afin d'éviter d'avoir des regroupements de salles, et de permettre à toutes les salles d'être accessibles depuis n'importe quelle autre salle. Ces salles ont pour valeur 2 dans le tableau d'entiers. Enfin, nous créons de la même manière la salle finale représentée par la valeur 3. Une fois le tableau 2D d'entiers initialisé, il ne reste plus qu'à créer toutes les salles dans le tableau 2D de salles et initialiser toutes les portes.

2 2 2 2	2 2 2 2	2 2 2
2 1 2 2	2 2	2 2 2
2 2 2 2 2	2 1 2 2 2 2	2 2
2 2 2	2 2 2	2 2 2
3 2 2	2 2 2 2 3	2 2 3 2 2
2 2 2	2	2 2 2 2
2 2	2 2 2	2 1 2 2 2

Exemples de génération de niveau

## Héros

La classe Héros correspond au joueur. Elle possède des entiers `x` et `y` qui sont la position du joueur dans la salle, des entiers `x_salle` et `y_salle` qui représentent la position de salle du héros dans le niveau. Nous avons également rajouté un entier `orientation` qui permet de définir l'orientation actuelle du joueur. Chaque valeur représente une orientation (0 pour le nord, 1 pour l'est, 2 pour le sud et 3 pour l'ouest). Cette fonctionnalité servira pour l'usage des clés et des capsules. Le héros possède également une variable `nombre de sauts` qui indique le nombre de saut qu'il peut encore réaliser dans la salle actuelle.

## Interaction avec les objets

Nous avons implémenté toutes les interactions entre le héros et les différents objets du jeu. Pour ce qui concerne les objets ramassable (coffres, clés et capsules d'eau), comme cela a été précisé précédemment, lorsque le héros se trouve au contact avec un de ces objets, c'est-à-dire lorsque le héros se déplace vers une case contenant l'objet, il est automatiquement ramassé, et son contenu est rajouté à son inventaire. Lorsque le joueur se trouve à côté d'une dalle, il ne peut pas la traverser. Cependant, s'il possède au moins une capsule d'eau, qu'il est orienté vers la dalle et appuie sur la touche « e », cette dalle devient une case normale et le joueur perd une capsule. Dans le même principe, si le joueur se trouve orienté vers une porte verrouillée et qu'il possède une clé, il peut l'utiliser en appuyant sur la touche « d ». Pour ce qui concerne les cases vides, nous avons implémenté une extension de saut que nous détaillerons plus tard. Lorsque le joueur se trouve dans l'emplacement d'une porte non verrouillée, il est automatiquement téléporté vers la salle voisine correspondante, juste à côté de la porte vers la salle précédente. Lorsque le joueur quitte une salle, toutes les capsules d'eau qui ont été générées dans la salle disparaissent et sont perdues.

## Jeu

La classe Jeu comporte toutes les instances nécessaires pour le déroulement d'une partie, c'est-à-dire un héros et un niveau contenant l'ensemble des salles.

## Extensions

Nous avons également rajouté certaines extensions au projet que nous allons présenter.

### Le scrolling

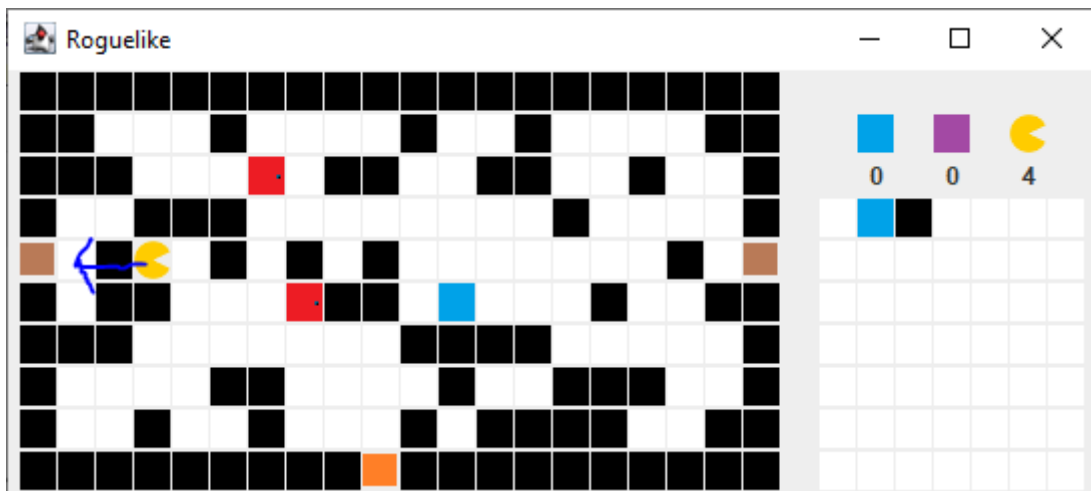
Le scrolling est une des premières fonctionnalités que nous avons réalisées. Comme nous avons directement créé la classe Niveau contenant le tableau de salle, il nous suffisait juste d'ajouter les positions de la salle actuelle du joueur dans la classe héros et d'afficher cette salle dans la fonction `mettreAJourAffichage` de `VueControleur`.

### Affichage de la mini-carte et de l'inventaire

Nous avons décidé d'afficher en permanence l'inventaire ainsi que d'ajouter une mini-carte afin de mieux nous repérer dans le jeu et savoir quelles salles nous avons visitées. Pour cela, nous avons décidé d'augmenter la valeur de `SIZE_X` de 8 et nous avons placé les icônes de la clé, de la capsule ainsi que l'icône du héros afin de représenter le nombre de clés, de capsules et le nombre de sauts que le joueur peut encore effectuer. En ce qui concerne l'affichage de la mini-carte, on n'affiche que les salles visitées. Pour cela, on se sert de la valeur du booléen `estVisite` de la classe `Salle`. Ce booléen est initialement à `false` et est ensuite passé à `true` au moment où le héros entre dans cette salle. Afin de se repérer plus facilement, la salle actuelle est représentée d'une autre couleur que les autres salles. Cet affichage est effectué dans `VueControleur` dans la fonction `mettreAJourAffichage` afin de respecter le modèle MVC du projet.

## Le saut avec contraintes

Lorsque le joueur se trouve devant une case vide et que la case suivante est traversable, il peut sauter le vide pour atteindre cette salle. Cette fonctionnalité est nécessaire puisqu'avec notre génération de salle, on doit souvent avoir besoin de sauter un vide afin d'accéder à des éléments. Nous avons implémenté cette fonctionnalité dans la classe héros, dans la fonction saut. Cette fonction n'est appelée que si le joueur souhaite se déplacer vers un emplacement non traversable. Le héros ne peut effectuer que 5 sauts lorsqu'il entre dans une salle. Dans la fonction saut, on vérifie d'abord que le joueur peut encore sauter, puis selon l'orientation du joueur, si la case en face de lui est une case vide, et la case qui suit est traversable, le joueur a la possibilité de sauter, et on décrémente le compteur de saut.

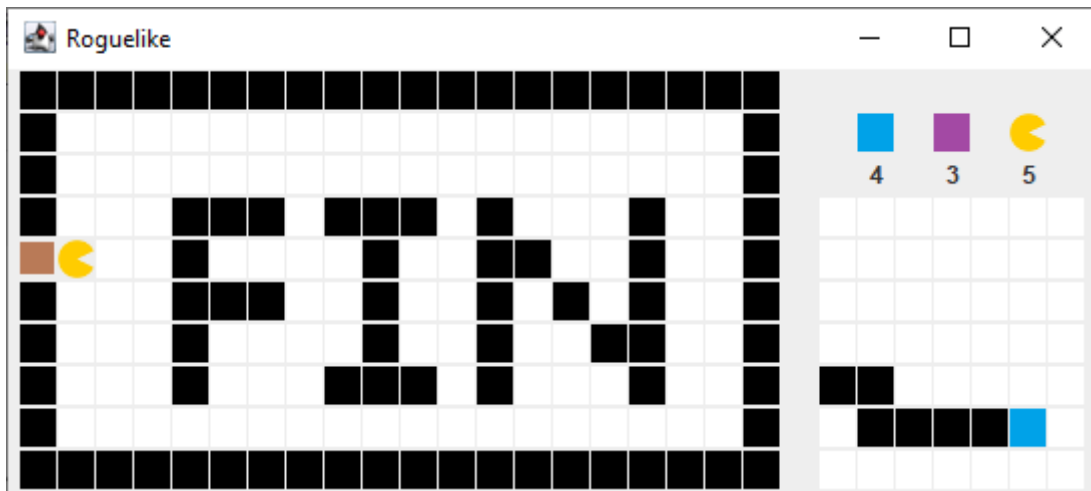


Exemple de situation où le saut est nécessaire

## Fin de partie

### Victoire

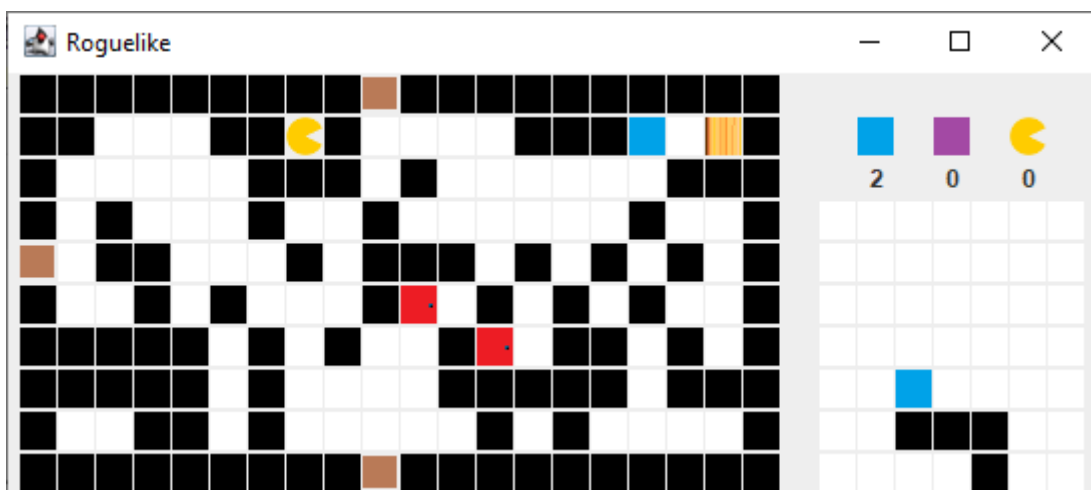
Le joueur remporte une partie s'il parvient à trouver la salle finale. Nous avons décidé d'écrire le code permettant d'initialiser cette salle dans la fonction `initialiserMurSalleFinal` de la classe `Salle`. Elle est appelée dans la classe `Niveau` pour la salle ayant comme valeur 3 dans le tableau d'entiers représentant le niveau.



Victoire du joueur

## Défaite

Bien que rien n'indique que le joueur a perdu la partie, on peut supposer que le joueur a perdu la partie s'il se retrouve bloqué par des trous et qu'il ne peut plus effectuer de sauts. Dans cette situation, il n'aura pas d'autre choix que d'appuyer sur le bouton « r » afin de recommencer une nouvelle partie.



Défaite du joueur

## Conclusion

Notre projet nous permet de réaliser une partie complète du jeu en respectant le cahier des charges ainsi que le modèle MVC. Par manque de temps, nous n'avons cependant pas pu implémenter de véritables conditions de victoire et défaite. Notre code possède également certaines parties pouvant être supprimé comme l'instance Inventaire présente dans la classe Jeu, inutilisée car la classe héros en possède un, et la classe Ramassable qui n'est pas utilisée. Nous avons donc décidé de ne pas les mentionner dans le rapport et le diagramme de classe. Nous avons également orienté notre code le plus objet possible en créant des classes lorsque cela nous semblait judicieux tout en se servant de l'héritage.