Projet de programmation: Wall Is You

L'objectif de ce projet est d'implémenter un petit jeu graphique de type "puzzle" dont la brique de base est un algorithme de recherche de chemin dans un labyrinthe.

Le jeu *Wall Is You*

Wall Is You[1] est une réinterprétation du scénario classique dans lequel le joueur incarne un aventurier courageux et doit s'engouffrer dans un donjon sinistre pour vaincre les créatures maléfiques qui s'y trouvent. Dans *Wall Is You*, les rôles sont inversés : le joueur joue le rôle d'un donjon bienveillant qui doit aider un aventurier particulièrement stupide à le débarrasser des monstres qui l'habitent.

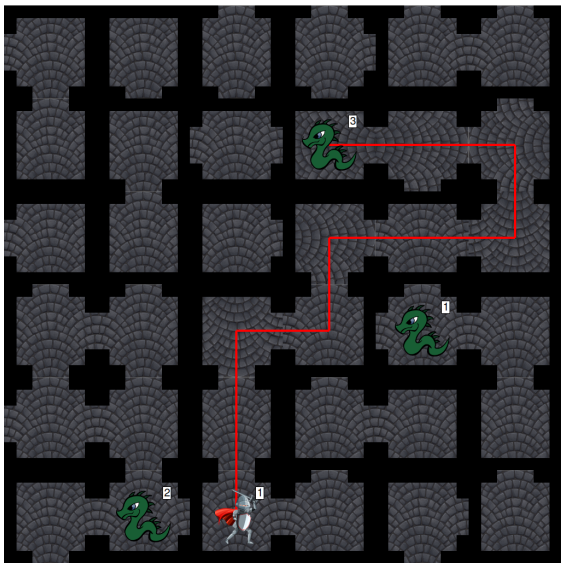


Fig. 1 : Une partie de *Wall Is You* en cours.

Le jeu alterne entre les tours du donjon, contrôlé par le joueur, et les tours de l'aventurier, contrôlé par le programme :

Le donjon

- Le donjon est une grille rectangulaire de *salles*. Chaque salle peut avoir un passage vers chacune des quatre directions cardinales (haut, droite, bas, gauche).

[1] Le nom du jeu est un clin d'oeil au jeu de puzzle *Baba Is You*.

- Les salles peuvent contenir des *dragons*. Chaque dragon a un niveau, et tous les dragons sont de niveaux différents.
- À son tour, le joueur peut cliquer une ou plusieurs fois sur autant de salles qu'il le souhaite pour les faire pivoter. Chaque clic fait pivoter la salle de 90° dans le sens horaire. Plusieurs clics successifs permettent donc de faire pivoter la salle de 180 ou 270° ou de la ramener dans sa position d'origine.

Fig. 2 : Les quatre orientations possibles d'une salle en L.

- Le joueur indique la fin du tour du donjon en appuyant sur la touche Espace.

L'aventurier

- L'*aventurier* se trouve dans l'une des salles du donjon et commence au niveau 1.
- L'aventurier est *orgueilleux* et cherche toujours à aller vers le dragon le plus fort qui lui est accessible. À chaque instant, l'aventurier indique son *intention* par une ligne rouge. Par exemple, sur la Fig. 1, l'intention de l'aventurier est d'affronter le dragon de niveau 3. Si aucun dragon n'est accessible, l'aventurier n'a pas d'intention.
- À son tour, l'aventurier se déplace de salle en salle le long de la ligne rouge, jusqu'à atteindre un dragon. S'il rencontre un dragon de niveau inférieur ou égal au sien, il le tue et gagne un niveau. S'il rencontre un dragon de niveau supérieur, il est tué et la partie est *perdue*.
- La partie est *gagnée* lorsque tous les dragons ont été tués !

Réalisation

Le projet se décompose en trois tâches principales, toutes obligatoires. Les améliorations facultatives sont marquées d'un nombre d'étoiles ($\star$) correspondant approximativement à leur difficulté.

Tâche 1: Réalisation du moteur de jeu

La première tâche du projet consiste à programmer la logique interne du jeu (le *modèle*), c'est-à-dire la partie qui permet de représenter l'état du jeu dans des structures de données appropriées, et de les modifier pour appliquer les effets des actions du joueur et de l'aventurier.

Cette tâche très conséquente est divisée en plusieurs sous-tâches.

Représentation de l'état du jeu

Voici une suggestion de structures de données permettant de représenter l'état du jeu (donjon, personnages). Vous êtes toutefois libres d'en changer, **à condition d'expliquer et de motiver vos choix.**

Chaque **salle** du donjon est représentée par un **tuple de 4 booléens** `(haut, droite, bas, gauche)`, indiquant quels passages sont possibles dans la salle.

Le **donjon** est représenté par une liste `donjon` de listes de salles. Les dimensions des listes correspondent aux dimensions du donjon représenté. La valeur `donjon[i][j]` décrit la salle se trouvant à la ligne `i` et à la colonne `j`.

Par exemple, le donjon ci-dessous est représenté par la structure suivante :

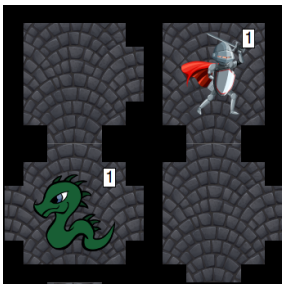


Fig. 3 : Un tout petit donjon

```
donjon = [[(False, True, True , False), (False, False, True, False)],
          [(True , True, False, True ), (True , True , True, False)]]
```

Ainsi, la valeur `donjon[1][0]` représente la salle en bas à gauche du donjon (celle contenant le dragon). Elle vaut `(True, True, False, True)` pour indiquer que la salle permet les passages vers le haut, la droite et la gauche, mais pas vers le bas.

Chaque **personnage** (aventurier ou dragon) est représenté par un dictionnaire ayant les clefs `'position'` et `'niveau'`. La position est représentée par un couple de coordonnées `(ligne, colonne)` et le niveau par un entier. Par exemple, l'aventurier de la Fig. 3 est représenté par le dictionnaire ci-dessous :

```
aventurier =  {
                'position' : (0,1)
                'niveau' : 1
              }
```

Comme le donjon peut contenir plusieurs dragons, les dictionnaires correspondant à chaque dragon sont regroupés dans une liste `dragons`.

Gestion du donjon

Le moteur du jeu doit permettre de faire pivoter les salles du donjon et de savoir facilement si deux salles sont connectées. Ces fonctionnalités seront assurées par les deux fonctions suivantes :

- `pivoter(donjon, position)` : la fonction modifie la liste de listes de salles `donjon` en faisant pivoter la salle se trouvant aux coordonnées `position` de 90 degrés dans le sens horaire. La fonction ne renvoie rien.
- `connecte(donjon, position1, position2)` : la fonction renvoie `True` si les salles de la liste de listes `donjon` aux coordonnées `position1` et `position2` sont *adjacentes* et *connectées* (le passage de la première salle vers la deuxième salle est ouvert, et réciproquement), et `False` sinon.

Intention de l'aventurier

La partie la plus délicate du moteur du jeu est le calcul de l'*intention* de l'aventurier, c'est-à-dire le chemin qu'il compte emprunter lorsque le donjon aura fini de jouer. Pour ce faire, on propose d'écrire une fonction `intention(donjon, position, dragons)` renvoyant un chemin (une liste de positions successivement connectées) menant l'aventurier jusqu'au dragon accessible de plus haut niveau, ou `None` si aucun dragon n'est accessible.

Plusieurs approches sont possibles pour écrire une telle fonction. Nous allons commencer par implémenter un algorithme de recherche naïf (appelé algorithme de recherche *par backtracking*, ou algorithme de *recherche en profondeur*) qui, depuis une position du jeu, va essayer toutes les positions connectées possibles et rechercher récursivement un chemin depuis chaque nouvelle position obtenue.

Pour garantir que l'algorithme ne provoque pas d'appels récursifs infinis, nous allons nous assurer que chaque position est explorée au plus une fois. Pour ce faire, nous allons enregistrer chaque position visitée dans un ensemble `visite` de type `set` et interrompre prématurément les appels récursifs sur des positions déjà visitées.

L'algorithme procède comme suit, à partir de la position `position` donnée en paramètre et de l'ensemble initialement vide `visite` :

1. Si `position` correspond à la position d'un dragon de niveau `niveau`, répondre `([position],niveau)` : on a trouvé un chemin trivial pour atteindre un dragon de niveau `niveau`.
2. Si la position est déjà dans `visite`, répondre `None`. Cette position à déjà été considérée, on interrompt la recherche.
3. On ajoute `position` dans `visite`.
4. On lance récursivement la recherche sur chaque position `cible` connectée à `position`, et on récupère les couples `(chemin, niveau)` ainsi obtenus.
5. Si aucune recherche n'a produit un couple `(chemin, niveau)`, on déduit qu'il n'y a pas de dragon accessible à partir de la position donnée, et on renvoie `None`.
6. Sinon, on récupère le couple `(chemin, niveau)` de plus haut niveau, et on renvoie `([position] + chemin, niveau)` : `niveau` est le niveau du dragon accessible le plus fort, et on a trouvé le chemin `[position] + chemin` pour l'atteindre.

Tour de l'aventurier

Lors de son tour, l'aventurier suit son intention, et en applique les conséquences. Voici une proposition de fonctions à implémenter :

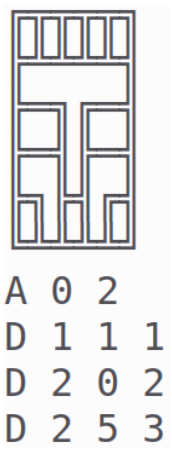
- `rencontre(aventurier, dragons)` : la fonction teste si l'aventurier est à la même position qu'un dragon, et en applique les conséquences : si l'aventurier rencontre un dragon de niveau inférieur ou égal, le dragon est tué (et retiré de la liste) et l'aventurier gagne un niveau. S'il rencontre un dragon de niveau supérieur, l'aventurier est tué (décidez d'une méthode pour le représenter).
- `appliquer_chemin(aventurier, dragons, chemin)` : la fonction déplace l'aventurier le long du chemin, et appelle la fonction `rencontre` quand nécessaire. La fonction ne renvoie rien.
- `fin_partie(aventurier, dragons)` : la fonction renvoie `1` si la partie est gagnée (tous les dragons ont été tués), `-1` si la partie est perdue (l'aventurier a été tué), et `0` si la partie continue.

Tâche 2: Représentation et chargement des donjons

Le but de cette tâche est de permettre au programme de lire et de charger des donjons prédéfinis et stockés dans des fichiers. Ainsi, un donjon de *Wall Is You* sera représenté par un fichier texte constitué de deux parties :

- La première partie représente l'agencement du donjon, dessiné avec les caractères spéciaux ╬, ╠, ╔, ╟, ╥ et leurs rotations. Par exemple, le caractère ╔ représente une salle avec des passages vers la droite et le bas, mais pas vers le gauche ou le haut.
- La seconde partie donne les coordonnées et niveaux des personnages. Par exemple, `A 2 5` indique que l'aventurier se trouve à la ligne 2 et à la colonne 5, tandis que `D 2 0 2` indique que la salle à la ligne 2 et colonne 0 contient un dragon de niveau 2. On ne précise pas le niveau de l'aventurier, qui commence toujours au niveau 1.

Un exemple de fichier avec le donjon correspondant est donné ci-dessous :



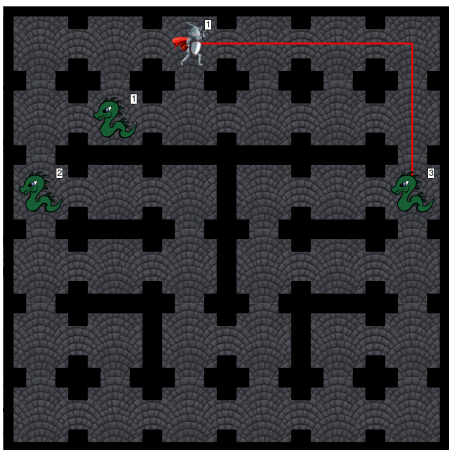


Fig. 4 : Le fichier d'un donjon, et le donjon correspondant.

Votre programme doit permettre de lire des fichiers écrits dans le format spécifié ci-dessus. Plus précisément, vous écrirez une fonction `charger(fichier)` recevant en paramètre le chemin d'accès d'un fichier et renvoyant les structures de données que vous aurez choisies lors de la tâche 1, correctement remplies avec les informations contenues dans le fichier. La fonction renverra `None` si le fichier proposé n'est pas correctement formaté, par exemple s'il contient un caractère inattendu.

Remarque : une grille n'est pas forcément carrée, elle peut aussi avoir une forme rectangulaire.

Tâche 3: Interface graphique

La troisième tâche consiste à programmer l'interface graphique du jeu (la *vue*).

Une interface ergonomique est attendue. Au lancement du programme, le joueur devra être accueilli par un menu lui permettant de charger un donjon de son choix parmi un ensemble de donjons disponibles.

Une fois le jeu lancé, le joueur pourra jouer en cliquant sur les salles à faire pivoter. À chaque coup joué, le programme se chargera de mettre à jour l'intention de l'aventurier et l'affichage en conséquent. Le joueur indiquera ensuite la fin de son tour en appuyant sur la touche `Espace`, et le programme se chargera de jouer et d'afficher le tour de l'aventurier.

Le joueur pourra recommencer le donjon choisi depuis le début avec la touche `R`, et revenir au menu pour choisir un autre donjon avec la touche `Echap`. En cas de victoire ou de défaite, un message approprié doit s'afficher et proposer au joueur de revenir au menu.

Toute la partie graphique du projet doit être réalisée avec la bibliothèque `fltk`.

Tâches complémentaires

Cette section propose des améliorations du projet, à aborder uniquement si toutes les tâches obligatoires ont été **entièrement terminées**.

(⋆⋆) Plus courts chemins

L'algorithme proposé dans la Tâche 1 pour calculer l'intention de l'aventurier ne produit pas nécessairement le chemin le plus court pour rejoindre le dragon visé. Ce défaut conduit à des situations dans lesquelles l'aventurier semble faire des détours incompréhensibles, comme dans le cas ci-dessous :

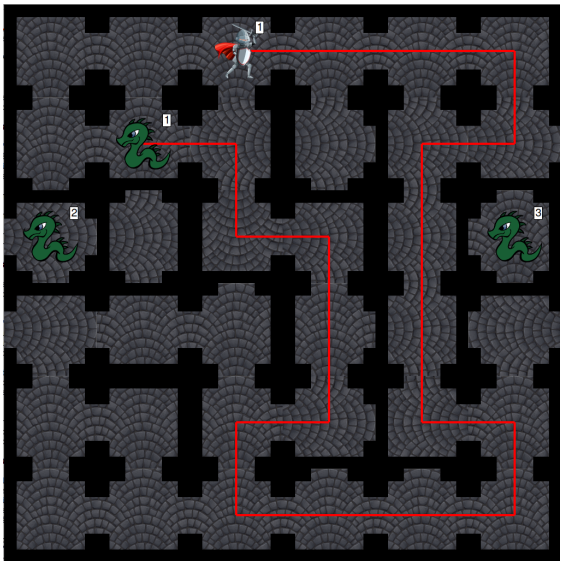


Fig. 5 : Un détour inutile.

Proposez un nouvel algorithme permettant de trouver le chemin le plus court pour aller vers le dragon de plus haut niveau. *(Indice : utilisez une file !)*

(⋆⋆) Trésors

Le jeu présenté ne laisse en fait que très peu de choix au joueur : il s'agit simplement de trouver un chemin allant de l'aventurier au premier dragon, puis du premier dragon au deuxième, etc. On souhaite enrichir le jeu en ajoutant des *trésors*, régis par les règles suivantes :

- Le donjon dispose d'un certain nombre de trésors, fixé à l'avance.
- Pendant son tour, le donjon peut, par un clic droit, placer un trésor dans une salle inoccupée. Il ne peut y avoir au plus qu'un seul trésor présent dans le donjon.
- L'aventurier est cupide, et est attiré en priorité par les trésors, même si des dragons sont accessibles.
- Lorsque l'aventurier rencontre un trésor, il le récupère (le trésor est définitivement retiré) et termine son tour.

Les trésors sont donc un outil pour le donjon, lui permettant d'attirer l'aventurier dans une salle de son choix, et donc de réaliser des itinéraires normalement impossibles.

Dans l'exemple ci-dessous, le joueur a placé un trésor pour attirer l'aventurier jusqu'à la dernière salle de la dernière ligne, puis a ensuite pivoté cette salle pour forcer l'aventurier à combattre le dragon de niveau 1 en premier. On peut facilement se convaincre que ce donjon ne peut pas être résolu sans trésor.

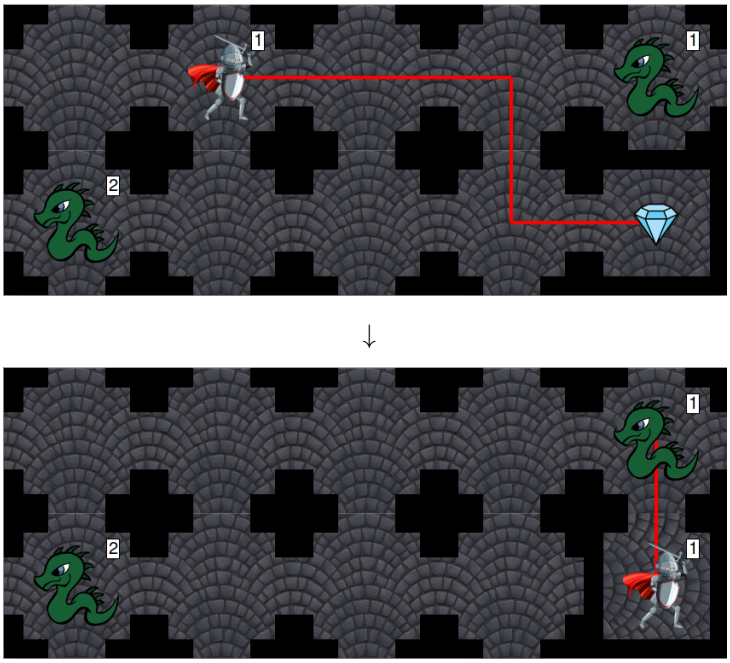


Fig. 6 : Utilisation d'un trésor.

Cette amélioration requiert d'adapter les différentes structures de données et algorithmes proposés, et de proposer des nouveaux donjons prédéfinis avec un nombre de trésors approprié.

Autres améliorations suggérées

Voici quelques autres suggestions d'améliorations moins détaillées. Attention, certaines sont plutôt faciles tandis que d'autres sont assez difficiles. N'hésitez pas à demander conseil à vos enseignants avant de vous lancer sur une de ces pistes.

- ($\star$) Permettre au joueur d'enregistrer une partie en cours ou de charger une partie enregistrée. L'enregistrement devra être fait dans un fichier de manière à pouvoir être récupéré lors d'une autre session de jeu.
- ($\star$) Améliorer l'apparence du jeu : les décors changent d'une salle à l'autre, les dragons tués laissent une dépouille au sol, l'aventurier change d'apparence en montant de niveau…
- ($\star\star$) Implémenter un éditeur de donjons, c'est-à-dire une interface graphique permettant à l'utilisateur de choisir la dimension du donjon, l'agencement des salles, et où placer les dragons et l'aventurier. Une fois la réalisation terminée, la grille sera enregistrée dans un fichier.
- ($\star\star$) Implémenter un mode de jeu "un seul tour" : une fois que le joueur a passé son tour, il ne peut plus jouer jusqu'à la fin de la partie, et l'aventurier doit pouvoir gagner seul. **L'intention de l'aventurier doit être affichée jusqu'à la fin de la partie, et pas seulement jusqu'au premier dragon rencontré.**

- (⋆⋆) Implémenter de nouvelles mécaniques de jeu qui n'attirent pas l'aventurier et n'interrompent pas son tour, mais sont activées lorsqu'il passe dessus. Par exemple : leviers permettant d'ouvrir ou de fermer des portes, elixir permettant de vaincre le prochain dragon même s'il est de niveau supérieur, etc. **Cette amélioration nécessite d'avoir réalisé la tâche "plus courts chemins" et de proposer des donjons adapatés et intéressants.**
- (⋆⋆) Choisir des règles de déplacement intéressantes pour les dragons, et une façon appropriée de les représenter visuellement, à la manière de l'intention de l'aventurier. **Cette amélioration nécessite de proposer des donjons adaptés, et se désactiver automatiquement lorsque le joueur charge un donjon "normal".**
- (⋆⋆⋆) Implémenter une fonction "Indice", indiquant au joueur un chemin possible jusqu'au prochain dragon, même si les salles ne sont pas actuellement correctement tournées.
- (⋆⋆⋆) Implémenter un générateur de niveaux aléatoires. Les niveaux générés doivent toujours admettre une solution.

Toute autre amélioration est envisageable selon vos idées et envies, à condition d'en discuter au préalable avec un de vos enseignants.